



SAPIENZA
UNIVERSITÀ DI ROMA

Visita in Ampiezza Ottimizzata per Direzione

Facoltà di Ingegneria dell'informazione, informatica e statistica
Corso di laurea in Informatica

Simone Salvatore Vivacqua
Matricola 2095596

Relatrice
Prof.^{ssa} Tiziana Calamoneri

Anno Accademico 2024/2025

Tesi non ancora discussa

Visita in Ampiezza Ottimizzata per Direzione
Sapienza Università di Roma

© 2025 Simone Salvatore Vivacqua. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Email dell'autore: s.simone.vivacqua@gmail.com

*Dedicato a
tutte le persone che mi hanno supportato in questo percorso, colleghi e non.*

Sommario

Contesto In questo lavoro di tirocinio si sono affrontate le problematiche relative alla visita in ampiezza di grafi di grandi dimensioni. In particolare, ci si è soffermati su due algoritmi noti in letteratura e le loro versioni multi-thread: Visita in Ampiezza classica e Visita in Ampiezza Ottimizzata per Direzione.

Obiettivi Dopo avere inquadrato proprietà e tipologie di grafi (diametro ridotto, presenza di grandi centri nevralgici e coefficiente di raggruppamento) e analizzato le principali strutture di rappresentazione (matrice di adiacenza, liste di adiacenza, matrice di adiacenza compressa), si passerà al confronto dei due algoritmi e delle loro versioni multi-thread, sotto ogni tipologia di grafo e struttura dati riportata, collegando tali caratteristiche all'efficienza degli algoritmi.

Metodi Implementate le varie strutture dati per la rappresentazione dei grafi in linguaggio C, si procede a generare e/o importare le varie tipologie di grafi, si confrontano Visita in Ampiezza classica e Visita in Ampiezza Ottimizzata per Direzione rispetto a spazio, prestazioni e parallelizzazione, con particolare oculatezza per quest'ultima, adottando operazioni atomiche per evitare le contese, memoria dedicata per thread e scheduling dinamico.

Risultati Su grafi grandi e sparsi a basso diametro, la Visita in Ampiezza Ottimizzata per Direzione riduce sensibilmente le ispezioni ridondanti nelle fasi centrali, ottenendo tempi di esecuzione più bassi per ogni struttura dati utilizzata. Si riscontra inoltre un miglioramento più marcato nelle prestazioni in modalità multi-thread rispetto alla versione classica. Al contrario, per i grafi regolari il vantaggio risulta limitato e, in alcuni casi, persino negativo.

Conclusione La combinazione di rappresentazioni compatte e scelte algoritmiche adattive consente di avvicinare il lavoro effettivo al numero di vertici nelle fasi più costose, massimizzando la località e limitando la sincronizzazione. Le tecniche descritte costituiscono una base solida e riutilizzabile per l'elaborazione multi-thread di grafi reali di grande scala, confermando quanto è detto nella teoria.

Indice

Acronomi	v
1 Introduzione	1
2 Visita in Ampiezza dei Grandi Grafi	3
2.1 Definizioni di base sui grafi, proprietà strutturali e impatto sulla Visita in Ampiezza	3
2.2 Tipologie di grafi nelle applicazioni reali	4
2.3 Sintesi delle sfide dei grandi grafi	6
2.4 Strutture dati atte alla rappresentazione dei grafi	7
2.4.1 Matrice di Adiacenza	8
2.4.2 Liste di Adiacenza	8
2.4.3 Matrice di Adiacenza Compressa	9
2.4.4 Considerazioni finali sulle strutture dati atte alla rappresentazione dei Grafi	10
3 Algoritmo di Visita in Ampiezza	12
3.1 BFS classica: formulazione e comportamento	12
3.1.1 Criticità della Visita in Ampiezza classica	14
3.2 Ottimizzazione della direzione: idea, strategia e costo	14
3.2.1 Euristica per la scelta della direzione	17
4 Visita in Ampiezza su architetture multi-thread	18
4.1 Strutture dati, organizzazione della memoria e cenni di gestione multi-thread	18
4.2 Visita in Ampiezza ottimizzata per Direzione multi-thread	19
4.3 Organizzazione pratica dei dati, Analisi dei costi e scalabilità	21
5 Risultati Sperimentali	24
5.1 Metodologia di Test	24
5.2 Versioni dell'Algoritmo di Visita in Ampiezza	25
5.3 Sperimentazioni con Grafi Generati	25
5.3.1 Risultati per i Grafi Regolari Casuali	25
5.3.2 Il modello di Newman–Watts	30
5.3.3 Risultati per i grafi generati utilizzando il modello Newman–Watts	31
5.3.4 Il modello di Barabási-Albert	35

5.3.5	Estensione del modello BA: il modello di Holme–Kim	37
5.3.6	Risultati per i grafi generati utilizzando il modello Barabási-Albert	37
5.3.7	Risultati per i grafi generati utilizzando il modello Holme–Kim	40
5.4	Sperimentazioni con Dataset	45
5.4.1	Grafi della raccolta SNAP	45
5.4.2	Grafi della raccolta Network Repository	47
6	Conclusioni	53
A	Implementazione	55
A.1	Strutture dati per la Visita in Ampiezza	55
A.2	Visita in Ampiezza sequenziale	57
A.3	Visita in Ampiezza multi-thread	60
A.3.1	Libreria OpenMP: definizione e funzioni	60
A.4	Verifica della correttezza dell’algoritmo	63
A.4.1	Veridicità dell’implementazione: Confronto	64
A.5	Generazione dei modelli noti di Grafi	65
Bibliografia		69

Capitolo 1

Introduzione

L’analisi e l’elaborazione di grafi costituiscono uno dei pilastri fondamentali dell’informatica moderna, sia teorica che applicata. Molti problemi del mondo reale possono essere modellati come grafi: una rappresentazione estremamente versatile, che consente di affrontare in maniera unitaria sfide provenienti da domini tra loro molto diversi. Tra i contesti più significativi si annoverano, ad esempio, la pianificazione dei trasporti e la determinazione di percorsi ottimali tra due punti minimizzando costi e tempi, come avviene nei sistemi di navigazione o nella logistica, l’analisi delle reti sociali e delle dinamiche di marketing per identificare connessioni non esplicite, e quindi non osservabili direttamente, tra utenti o clienti e suggerire relazioni o prodotti in base a interessi comuni, la progettazione di reti di telecomunicazione resilienti ed efficienti per ridurre la latenza nelle trasmissioni di dati, e perfino l’elaborazione del linguaggio naturale, dove i testi possono venire rappresentati come grafi semantici per poi essere analizzati.

Tra i numerosi algoritmi su grafi, la *Visita in Ampiezza* (Breadth–First Search, BFS) riveste un ruolo centrale: rappresenta un paradigma di esplorazione semplice ma potente, alla base di molte applicazioni, dalla visita del cammino minimo a procedure di analisi su reti complesse. Tuttavia, quando le dimensioni dei grafi crescono fino a includere milioni o perfino miliardi di nodi ed altrettanti archi, l’esecuzione sequenziale diventa rapidamente impraticabile e onerosa. È in questo contesto che entrano in gioco le tecniche di ottimizzazione per direzione e di parallelizzazione, in questo caso a memoria condivisa (multi-thread), concepite rispettivamente per ridurre il numero di esplorazioni necessarie e per sfruttare l’hardware multi–core al fine di abbattere drasticamente i tempi di calcolo.

Sarà posta anche particolare attenzione alla memorizzazione dei dati in memoria, aspetto cruciale nei grandi grafi, con un confronto tra le strutture dati più note per la loro rappresentazione.

Si propone, quindi, di studiare e implementare versioni sia sequenziali sia multi–thread dell’algoritmo di Visita in Ampiezza, nonché la sua versione *Ottimizzata per Direzione*⁽⁴⁾, con enfasi su grafi di reti reali, per poi effettuare un confronto tra le diverse versioni, sia sul piano teorico sia su quello implementativo e prestazionale.

La scelta di affrontare dapprima l'algoritmo classico e successivamente la sua versione ottimizzata per direzione e quella multi-thread nasce dalla volontà di disporre di un termine di paragone solido e di valutare progressivamente, su basi empiriche, i vantaggi offerti dalle varie ottimizzazioni. L'attività svolta ha quindi permesso di analizzare in maniera critica le proprietà strutturali dei grafi reali e le loro implicazioni sul comportamento della Visita in Ampiezza, di approfondire le principali tecniche di parallelizzazione multi-thread applicabili a questo algoritmo con particolare riferimento al modello di programmazione OpenMP, di sviluppare e confrontare implementazioni sequenziali e multi-thread su grafi di dimensioni realistiche, e di valutare sperimentalmente le prestazioni identificando colli di bottiglia e le strutture dati più adatte alla rappresentazione dei grafi.

Per l'implementazione dell'algoritmo sarà utilizzato il linguaggio di programmazione C, al fine di garantire un controllo fine sulle risorse computazionali e sulle strutture dati.

Capitolo 2

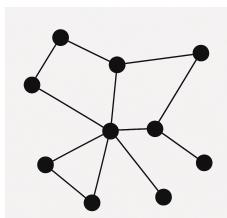
Visita in Ampiezza dei Grandi Grafi

La comprensione del comportamento dell'algoritmo di Visita in Ampiezza (BFS) passa attraverso l'analisi delle principali proprietà strutturali delle reti a cui esso si applica.

Inoltre, nell'ambito della Visita in Ampiezza riveste un ruolo centrale il concetto di frontiera, che definiremo come l'insieme dei vertici raggiunti ma non ancora completamente esplorati, ossia i nodi la cui esplorazione dei vicini è in corso, ed è un insieme ben definito del grafo.

A ogni iterazione, la frontiera viene aggiornata includendo tutti i vertici adiacenti a quelli attualmente presenti nella frontiera, purché non siano stati visitati in precedenza. In altre parole, a ogni passo vengono analizzati tutti i vertici che si trovano alla medesima distanza dalla sorgente, corrispondenti a un determinato livello di profondità, prima di procedere con il livello successivo.

2.1 Definizioni di base sui grafi, proprietà strutturali e impatto sulla Visita in Ampiezza



Un grafo $G = (V, E)$ è un insieme di vertici V collegati da archi E ; può essere orientato, se gli archi hanno direzione specifica, oppure non orientato, se rappresenta relazioni simmetriche.

Formalmente, nel modello a livelli di Beamer et. al.⁽⁴⁾ la frontiera è un insieme ben definito a ogni passo.

Sia $s \in V$ la sorgente. Definiamo la frontiera e i non visitati per livelli come

$$F_0 = \{s\}, \quad U_0 = V \setminus F_0,$$

e, per $i \geq 0$,

$$F_{i+1} = \{ u \in U_i \mid \exists v \in F_i \text{ con } (v, u) \in E \}, \quad U_{i+1} = U_i \setminus F_{i+1}.$$

Allora $F_i = \{ v \in V \mid \text{dist}_s(v) = i \}$ e gli insiemi F_i sono a due a due disgiunti.

Dal momento che il presente lavoro si concentra sull'analisi dell'algoritmo di *Visita in Ampiezza*, si farà riferimento alla classificazione delle proprietà strutturali dei grafi proposta da Scott Beamer, Krste Asanović e David Patterson⁽⁴⁾, particolarmente rilevanti per lo studio delle prestazioni di tale algoritmo in differenti contesti applicativi, e sono: il *diametro* e la *distribuzione dei gradi*. Il diametro rappresenta la distanza massima, in termini di archi, fra due vertici qualunque ed è indicativo della profondità della rete. Grafi con basso diametro possono essere esplorati in pochi livelli, mentre grafi con diametro elevato richiedono numerosi passi per raggiungere tutti i nodi. La distribuzione dei gradi, invece, descrive come gli archi sono distribuiti tra i vertici: reti con distribuzione uniforme presentano nodi con simile numero di adiacenze, mentre reti sbilanciate contengono pochi nodi con grado elevatissimo (hub) e molti con grado molto basso. Questa caratteristica influenza l'ampiezza della frontiera e, di conseguenza, il carico computazionale di ciascun passo dell'algoritmo.

Un ulteriore indicatore di rilievo è il *coefficiente di raggruppamento* (clustering), che misura la tendenza dei vicini di un nodo a essere a loro volta collegati tra loro, formando così dei “triangoli” o comunità locali. Valori elevati del coefficiente di raggruppamento indicano reti con forte struttura comunitaria, come i grafi sociali, mentre valori bassi corrispondono a reti più sparse e casuali, inoltre un alto valore può influenzare l'espansione della Visita in Ampiezza favorendo la scoperta di nodi già vicini tra loro e modificando la crescita della frontiera: questo aspetto diventa particolarmente rilevante nelle reti in cui brevi distanze medie si combinano ad un alto coefficiente di raggruppamento (2.2).

Queste proprietà strutturali (diametro, distribuzione dei gradi e coefficiente di raggruppamento) non sono semplici definizioni teoriche ma variabili concrete che determinano le prestazioni reali dell'algoritmo. Ad esempio, grafi con basso diametro e hub molto connessi producono frontiere ampie già nelle prime iterazioni, rendendo più onerosa l'esplorazione; reti con alto coefficiente di raggruppamento possono generare numerosi archi ridondanti verso nodi già scoperti, aumentando il numero di controlli e accessi in memoria necessari a ciascun passo. Comprendere questi fenomeni è essenziale per valutare correttamente le ottimizzazioni della Visita in Ampiezza, come l'approccio bottom-up o l'euristica che sceglie la strategia ad ogni iterazione dell'algoritmo introdotta nel capitolo successivo.

2.2 Tipologie di grafi nelle applicazioni reali

Le reti osservate nel mondo reale non sono omogenee: presentano proprietà strutturali differenti che incidono in maniera significativa sul comportamento della Visita in Ampiezza. Comprendere le tipologie più comuni di grafi aiuta a interpretare i

risultati sperimentali e a scegliere strategie di esplorazione appropriate. La Tabella 2.1 riassume alcune tipologie comuni e le loro caratteristiche rilevanti.

Una prima classe è rappresentata dai *Grafi Regolari*, in cui ciascun vertice ha lo stesso numero di adiacenze. Formalmente, un grafo si dice *k-regolare* se ogni vertice ha esattamente grado k . Nel caso non orientato e semplice, vale la relazione

$$2e = \sum_{v \in V} (v) = nk \quad \Rightarrow \quad k = \frac{2e}{n},$$

che collega il numero totale di archi e , i vertici n e il grado comune k . I Grafi Regolari casuali mantengono la condizione di regolarità ma assegnano in modo casuale le adiacenze, producendo reti prive di grandi centri di collegamento (hub) predominanti e con distribuzione dei gradi strettamente concentrata attorno a k . Verranno presi in considerazione grafi con componenti piccole, per evidenziare alcune specifiche limitazioni.

Dal punto di vista della Visita in Ampiezza, la regolarità del grado tende a generare frontiere che crescono in modo prevedibile e relativamente omogeneo fra i livelli. L'assenza di vertici con connettività anomala evita picchi improvvisi nel numero di archi da esplorare e facilita un bilanciamento del carico più stabile anche in ambiente multi-thread. D'altra parte, quando il diametro è non trascurabile, l'esplorazione può richiedere più livelli per raggiungere tutti i nodi: la Visita in Ampiezza resta ordinata e ben distribuita, ma il numero di iterazioni può aumentare in funzione della profondità della rete con un conseguente aumento del tempo totale di esecuzione.

Una seconda tipologia è quella dei *Grafi Small-World* sono caratterizzati da una breve distanza media tra coppie di nodi (diametro basso), la quale cresce lentamente con n (es. $\log n$ o inferiore), a cui si combina un alto coefficiente di raggruppamento.

Questo implica che, nel contesto della Visita in Ampiezza, pur essendo che i nodi sono tendenzialmente organizzati in gruppi locali densi, è possibile raggiungere qualsiasi vertice del grafo attraverso un numero ridotto di passaggi. Al tempo stesso, l'alto coefficiente di raggruppamento dei vertici concentra numerosi archi all'interno di comunità locali, incrementando la probabilità di incontrare vicini già scoperti nelle fasi iniziali. Ne risulta una dinamica in cui la copertura cresce rapidamente, ma i controlli su archi ridondanti possono aumentare, con effetti sensibili sul numero di accessi in memoria e sulla località degli stessi. In ambiente multi-thread, la concentrazione di connessioni all'interno di comunità può semplificare la partizione dei dati, ma richiede comunque attenzione nell'evitare contese quando più thread aggiornano strutture condivise legate alla frontiera.

Questi grafi sono utilizzati per modellare molte reti reali, quali reti sociali, reti neurali e sistemi di comunicazione, in cui la struttura locale compatta e la presenza di brevi percorsi medi tra nodi risultano determinanti per l'efficienza e la robustezza della rete.

Infine, vi sono i grafi *scale-free*, i quali presentano una distribuzione dei gradi che segue una legge di potenza: la probabilità che un vertice abbia grado k decresce come $P(k) \sim k^{-\gamma}$, con esponente γ spesso compreso tra 2 e 3. Questa caratteristica produce

un elevato numero di vertici a basso grado e pochi vertici ad altissimo grado, detti *hub* o grandi centri di collegamento, che fungono da snodi centrali della connettività. Il meccanismo di *attaccamento preferenziale* (preferential attachment) è un processo generativo sequenziale che conduce a tale distribuzione: i nuovi vertici tendono a collegarsi con maggiore probabilità ai nodi più connessi. È importante osservare che il modello di base ad attaccamento preferenziale non implica necessariamente un elevato coefficiente di raggruppamento; varianti con formazione di triadi invece lo introducono, avvicinandosi meglio alle reti sociali reali.

Sull'esecuzione della Visita in Ampiezza, la presenza dei *grandi centri di collegamento* (hub) ha due effetti principali. Da un lato accelera l'espansione iniziale della frontiera: pochi passi sono sufficienti a raggiungere un'ampia frazione dei vertici, dato che i centri fungono da snodo principale il quale si collega con una buona parte degli altri vertici del grafo. Dall'altro lato, i grandi centri di collegamento concentrano un numero enorme di adiacenze che, in implementazioni multi-thread, possono causare squilibri marcati nel carico: alcuni thread si trovano a processare liste di vicini molto lunghe mentre altri terminano rapidamente, aumentando le contese per le risorse di memoria e peggiorando la scalabilità dell'algoritmo. Inoltre, nelle fasi centrali della visita, molte ispezioni degli archi incidenti ai grandi centri di collegamento finiscono su vertici già visitati, aumentando i controlli ridondanti, rendendo l'algoritmo particolarmente lento. L'ottimizzazione della direzione (scelta tra top-down / bottom-up) che analizzeremo più avanti nasce anche per mitigare questo fenomeno, riducendo il numero di archi effettivamente controllati quando la frontiera si fa molto estesa.

I modelli di reti complesse consentono di rappresentare un'ampia gamma di sistemi reali, dalle reti sociali alle infrastrutture di comunicazione (come Internet) fino alle reti sociali, in cui la presenza di grandi centri di collegamento (hub) rivestono un ruolo cruciale per la struttura e le funzionalità del sistema.

Queste classi, discusse anche in letteratura da Beamer⁽⁴⁾, costituiscono modelli di riferimento per valutare l'efficacia delle diverse strategie della Visita in Ampiezza. Nei capitoli successivi vedremo come l'ottimizzazione della direzione e l'approccio bottom-up possano mitigare gli effetti negativi di alcune di queste caratteristiche strutturali.

Tabella 2.1 Principali classi di grafi e loro caratteristiche

Classe	Distribuzione dei gradi	Effetto sulla BFS
Regolare	Uniforme	Frontiera bilanciata, più iterazioni
Small-world	Diametro basso, alto coeff. di raggrup.	Copertura in meno iterazioni
Scale-free	Legge di potenza (hub)	Frontiera esplode, carico sbilanciato

2.3 Sintesi delle sfide dei grandi grafi

Abbiamo presentato le proprietà strutturali dei grafi, quali il diametro, distribuzione dei gradi e del coefficiente di raggruppamento, raccogliamo ora in modo unitario le principali criticità che emergono quando la dimensione del grafo cresce: si tratta di

problemi trasversali, indipendenti dal modello specifico, che influenzano direttamente l'efficienza della Visita in Ampiezza (BFS).

Il primo vincolo è lo *spazio*: rappresentazioni dense richiedono $\mathcal{O}(n^2)$ memoria e diventano rapidamente impraticabili, mentre schemi sparsi riducono l'occupazione a $\mathcal{O}(V + E)$ ma introducono nuove tensioni sulla struttura dei dati e sulla località degli accessi.

A ciò si somma l'*accesso non sequenziale*: la Visita in Ampiezza attraversa adiacenze disposte lontano in memoria, con un aumento dei mancati ritrovamenti in cache (cache miss) e della pressione sulla banda (rendendo necessario l'accesso diretto alla RAM, che può risultare da 10 a 100 volte più lenta rispetto alla cache L1⁽¹⁾).

Nei contesti paralleli a memoria condivisa (multi-thread) emergono *sbilanciamento del carico*, dovuto all'eterogeneità dei gradi, in particolare in presenza di hub, e *contese* su strutture condivise, che impongono sincronizzazioni e riducono l'efficienza. In sistemi distribuiti, il costo di *comunicazione* tra nodi di calcolo può diventare dominante.

Infine, la *dinamicità* topologica richiede strutture aggiornabili con costi locali, mentre la *dimensione della frontiera* e la *debole località* dei grafi sparsi determinano il numero di controlli ridondanti su archi e la qualità del riuso della cache complessiva, attraverso tecniche di precaricamento (prefetching) delle risorse⁽⁵⁾.

Tabella 2.2 Sintesi delle problematiche dei grandi grafi

Sfida	Impatto principale	Dominante in
Costo di memoria	Rappresentazione non allocabile in RAM	Tutti i contesti
Accesso non sequenziale	Cache miss, rallentamenti	Sequenziale e multi-thread
Squilibrio del carico	Bassa efficienza dei core	multi-thread
Contesa/sincronizzazione	Attese e overhead	multi-thread
Comunicazione	Latenza di rete	Distribuito
Dinamicità topologica	Necessità di aggiornamenti locali	Grafi dinamici
Frontiera molto grande	Costi di gestione/copia	Sequenziale e multi-thread
Località debole	Precaricamento inefficiente	Tutti i contesti

Le criticità riassunte in Tabella 2.2 spiegano perché, su grandi reti reali, l'appuccio top-down tenda a eseguire molte verifiche ridondanti sugli archi e a soffrire la latenza di memoria nelle fasi centrali. Questi limiti giustificano l'*ottimizzazione della direzione*, con il cambio di direzione verso la modalità bottom-up quando la frontiera diventa ampia, e l'impiego mirato del parallelismo a memoria condivisa, che esamineremo nei capitoli successivi.

2.4 Strutture dati atte alla rappresentazione dei grafi

La scelta della struttura dati per rappresentare un grafo incide in modo determinante sulle prestazioni degli algoritmi di visita, in particolare quando si opera su reti di

grandi dimensioni e in ambienti multi-thread. Discuteremo in maniera comparativa i tre schemi più comuni evidenziando come le loro proprietà e di come influenzino l'esecuzione della Visita in Ampiezza (BFS). La Figura 2.1 mostra, su un piccolo esempio, come gli stessi dati vengano organizzati nei tre formati.

2.4.1 Matrice di Adiacenza

La *Matrice di Adiacenza* è una delle possibili rappresentazioni per grafi orientati e non, formalmente definita come una matrice binaria $A \in \{0, 1\}^{V \times V}$, dove V rappresenta l'insieme dei vertici del grafo. L'elemento A_{ij} assume valore pari a 1 se e solo se esiste un arco diretto dal vertice v_i al vertice v_j ; in caso contrario, $A_{ij} = 0$.

Dal punto di vista di occupazione in memoria, la matrice di adiacenza richiede una quantità di memoria pari a $\mathcal{O}(n^2)$, indipendentemente dalla densità del grafo. Questo rende la rappresentazione poco adatta a grafi di grandi dimensioni e particolarmente inefficiente per grafi sparsi, dove la maggior parte delle celle risulta nulla, comportando uno spreco significativo delle risorse.

L'accesso agli archi è invece molto efficiente, in quanto può avvenire in tempo costante $O(1)$, facilitando la verifica dell'esistenza di un arco specifico. Tuttavia, l'iterazione sui vertici adiacenti a un nodo richiede un tempo lineare rispetto al numero di nodi $O(n)$, poiché è necessario scorrere l'intera riga della matrice.

Questo schema presenta regolarità nell'accesso quando si percorrono righe contigue, ma la Visita in Ampiezza raramente accede a righe in sequenza perfetta: i salti fra righe diverse, determinati dalla dinamica della frontiera, amplificano la distanza tra gli indirizzi effettivamente referenziati e riducono la località spaziale. Ne conseguono mancate *corrispondenze nella cache* (cache miss) più frequenti e pre-caricamento delle risorse inefficace, soprattutto su grafi grandi, con un conseguente effetto negativo sul *ritmo di esecuzione delle istruzioni* e sulla latenza percepita dalla CPU. La rigidità strutturale, infine, rende onerose le modifiche dinamiche: inserire o rimuovere vertici o archi implica riallocazioni del grafo. In sintesi, lo schema è indicato quasi esclusivamente per grafi molto densi, a scopo didattico, o algoritmi che usano direttamente la struttura a matrice.

2.4.2 Liste di Adiacenza

La rappresentazione tramite liste di adiacenza memorizza, per ciascun vertice, l'elenco dei suoi vicini. Indichiamo con **teste** (cime delle liste) un array di lunghezza n che, per ciascun vertice v_i , contiene un riferimento (o indice) al primo elemento di una lista dei suoi vicini.

Questo formato memorizza soltanto gli archi effettivamente presenti e richiede, in termini asintotici, spazio $\mathcal{O}(V + E)$. Rispetto alla matrice di adiacenza ($\mathcal{O}(V^2)$), comporta un risparmio decisivo su reti sparse.

La contropartita è la scarsa località: le strutture basate su puntatori collocano in memoria elementi logicamente vicini ma fisicamente lontani, riducendo il *riuso della cache* e l'efficacia dei meccanismi di *precaricamento* (prefetching). Ulteriori costi di overhead derivano dai puntatori stessi.

La visita dei vicini di un vertice v richiede tempo $\mathcal{O}(\deg(v))$, ossia il grado del nodo.

Un punto di forza della lista è la buona adattabilità a grafi dinamici: aggiungere o rimuovere archi è in genere un'operazione locale che non richiede la ricostruzione dell'intera struttura.

Sotto parallelismo a memoria condivisa (multi-thread), la suddivisione del lavoro per vertici è naturale, ma l'eterogeneità dei gradi può indurre squilibri significativi fra thread: alcuni esauriscono rapidamente le proprie liste, altri restano vincolati a vertici ad alto grado.

Queste limitazioni possono essere mitigate adottando rappresentazioni più efficienti, come per la Matrice di Adiacenza Compressa, che conserva i vantaggi spaziali delle liste di adiacenza migliorando la località dei dati e supportando l'elaborazione multi-thread.

2.4.3 Matrice di Adiacenza Compressa

La matrice di adiacenza Compressa (CSR) rappresenta un compromesso: immagazzina i vicini di ciascun nodo come un intervallo (offset) contiguo all'interno di un array compatto, ottenuto previa ordinamento degli archi.

Indichiamo con `offset` un array di lunghezza $n + 1$ e con `adj` un array di lunghezza e tale che i vicini di v_i corrispondano all'intervallo `adj[offset[i] : offset[i+1]]`. Questo formato, ampiamente utilizzato anche nella rappresentazione di matrici sparse⁽⁶⁾, consente di memorizzare esclusivamente gli archi effettivamente presenti, tale organizzazione comporta un significativo risparmio di memoria, riducendo lo spazio richiesto rispetto alla Matrice di Adiacenza classica da $\mathcal{O}(V^2)$ a $\mathcal{O}(V + E)$.

L'accesso agli archi incidenti a un vertice può avvenire in tempo $\mathcal{O}(d)$ nel caso di scansione lineare, con d grado del vertice, oppure in $\mathcal{O}(\log d)$ qualora venga utilizzata una struttura ausiliaria per la ricerca binaria, dove d rappresenta il grado del vertice. L'iterazione sull'insieme dei nodi adiacenti è invece sempre di complessità lineare $\mathcal{O}(d)$.

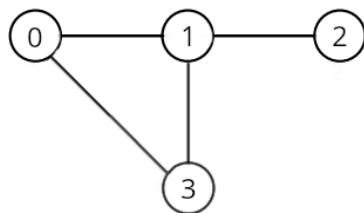
Sulle gerarchie di memoria moderne, la Matrice di Adiacenza Compressa risulta particolarmente favorevole: gli accessi sequenziali su `adj` attivano efficacemente il precaricamento nella cache(prefetching), riducendone i *mancati usi* velocizzando l'operazione di elaborazione dei dati da parte della CPU. Ciò si traduce in miglioramenti tangibili nelle fasi della Visita in Ampiezza dominate da letture ripetute di vicini.

Prendendo in considerazione un ambiente multi-thread, la struttura dati a Matrice Compressa si dimostra altamente efficiente: la ripartizione per blocchi contigui dell'array `adj` minimizza conflitti sulle cache e facilita una distribuzione più regolare del carico, anche se l'asimmetria dei gradi resta una fonte di squilibrio, favorendo l'esecuzione multi-thread su CPU multicore (ma anche per sistemi fortemente paralleli, come ad esempio GPU⁽⁴⁾). Per tali motivi, è ampiamente adottato in implementazioni di algoritmi fortemente paralleli di Visita in Ampiezza (BFS) su

grafi di grandi dimensioni.

La matrice di adiacenza compressa presenta però anche diverse criticità. Inizialmente, è *difficile da aggiornare dinamicamente*, risultando quindi più adatta a grafi statici. La sua costruzione richiede una fase di *pre-elaborazione* che include l'ordinamento degli archi, comportando una complessità aggiuntiva sia in termini di tempo che di spazio. Inoltre, la distribuzione del carico di lavoro tra i thread può non essere bilanciata, soprattutto nei grafi in cui i vertici presentano gradi molto diversi tra loro, rendendo complicata una suddivisione uniforme del lavoro.

Grafo originale:



Matrice di adiacenza:

0	1	2	3
0	[0, 1, 0, 1]		
1	[1, 0, 0, 1]		
2	[0, 1, 0, 0]		
3	[1, 1, 0, 0]		

Lista di adiacenza:

0: 1, 3
1: 0, 3
2: 1
3: 0, 1

Matrice Compressa:

offset:	[0 , 2 , 4 , 5]
adiacenze:	[1, 3, 0, 3, 1, 0, 1]

0 1 2 3

Figura 2.1 Esempio di rappresentazione del medesimo grafo con Matrice di Adiacenza, Liste di Adiacenza e Matrice di Adiacenza Compressa.

2.4.4 Considerazioni finali sulle strutture dati atte alla rappresentazione dei Grafi

Per la Visita in Ampiezza su grafi statici, grandi e sparsi, la Matrice di Adiacenza Compressa offre il miglior compromesso tra spazio e prestazioni grazie alla località degli accessi durante l'iterazione dei vicini e alla buona *scalabilità* in ambiente multi-thread. Le liste di adiacenza rimangono una scelta valida quando si privilegia la flessibilità agli aggiornamenti e si accetta un minor sfruttamento della cache; la matrice di adiacenza, infine, è indicata quasi esclusivamente per grafi molto densi o per contesti nei quali l'accesso $O(1)$ all'arco e la struttura matriciale siano requisiti centrali.

In definitiva, *non esiste una rappresentazione universalmente ottimale* per la memorizzazione dei grafi; ciascuna struttura presenta *vantaggi e limiti* specifici. La scelta della strategia più efficiente dipende pertanto dal contesto applicativo e dalle caratteristiche del grafo da elaborare.

Tabella 2.3 Confronto tra le principali strutture dati per la rappresentazione dei grafi.

Struttura	Memoria	Località	Dinamismo	multi-thread
Matrice di adiacenza	$\Theta(n^2)$	Buona	No	Media
Liste di adiacenza	$\Theta(n + m)$	Pessima	Sì	Buona ma variabile
Matrice di a. compressa	$\Theta(n + m)$	Ottima	No	Buona/Ottima se bilanciato

Capitolo 3

Algoritmo di Visita in Ampiezza

L'algoritmo di *Visita in Ampiezza* (BFS) costituisce una procedura fondamentale per l'esplorazione di un grafo a partire da un nodo sorgente. Tale algoritmo è frequentemente impiegato come base per numerose altre tecniche di elaborazione e analisi dei grafi, spiegandone dunque la rilevanza, tra cui la determinazione del cammino minimo in grafi non pesati, l'analisi della connettività in reti complesse, la navigazione di strutture dati come alberi, nonché in sistemi di raccomandazione e nell'analisi delle reti sociali.

3.1 BFS classica: formulazione e comportamento

Il principio di funzionamento dell'algoritmo di Visita in Ampiezza si fonda sull'aggiornamento iterativo della *frontiera* dell'esplorazione: esplora il grafo a partire da una sorgente s procedendo per *livelli* di distanza: a ogni iterazione vengono visitati tutti i vertici alla stessa distanza da s prima di passare al livello successivo, mantenendo nella *frontiera* i vertici scoperti ma non ancora completamente esplorati; per ciascun vertice in frontiera si scandisce l'elenco dei vicini, marcando come visitati quelli non ancora scoperti e aggiornando i relativi *genitori* (*parents*). Questo schema assicura che il primo arrivo a un vertice avvenga lungo un cammino minimo in numero di archi.

Pseudocodice Visita in Ampiezza

```
function breadth-first-search(vertices, source)
    frontier ← {source}
    next ← {}
    parents ← [-1, -1, ..., -1]
    while frontier != {} do
        top-down-step(vertices, frontier, next, parents)
        frontier ← next
        next ← {}
    end while
    return tree
```

Nel modello *top-down* ogni passo parte dalla frontiera corrente, spingendo l'esplorazione verso i vicini dei nodi contenuti in essa.

Pseudocodice top-down.

```
function top-down-step(vertices, frontier, next, parents)
    for v ∈ frontier do
        for n ∈ neighbors[v] do
            if parents[n] = -1 then
                parents[n] ← v
                next ← next ∪ {n}
            end if
        end for
    end for
return
```

Per l'analisi dei vari algoritmi della Visita in Ampiezza, adottiamo la notazione di Beamer et al.⁽⁴⁾ studiandone il lavoro (confronti di adiacenza): al livello i siano F_i la frontiera, $n_f(i) = |F_i|$ la sua cardinalità ed indicando con d il grado del nodo v . Il numero complessivo di accessi alla memoria al livello i risulta quindi

$$m_f(i) = \sum_{v \in F_i} d(v). \quad (3.1)$$

ossia il numero di adiacenze incidenti alla frontiera (archi che il passo top-down esamina).

Denotando con $G_s = (V_s, E_s)$ il componente raggiungibile a partire dalla sorgente s , con $n = |V_s|$ e $m = |E_s|$, e considerando che nel modello top-down ogni arco incidente a un vertice in frontiera è potenzialmente esaminato una volta, mentre nei grafi non orientati l'arco può essere visto da entrambi gli estremi, il lavoro complessivo sarà quindi di

$$\sum_i m_f(i) = \begin{cases} m & (\text{orientato}) \\ 2m & (\text{non orientato}). \end{cases} \quad (3.2)$$

mentre il costo asintotico, il quale risulta proporzionale al lavoro, per entrambi i tipi di grafi equivale a

$$\Theta\left(\sum_i m_f(i)\right) = \Theta(m) = \Theta(V + E) \quad (3.3)$$

Ulteriori costi ammontano a $O(n)$ per le assegnazioni dei padri e a $O(\sum_i |F_{i+1}|) = O(n)$ per la costruzione delle frontiere successive.

Sommendo quindi su tutti i livelli, con rappresentazioni a liste di adiacenza e Matrici di Adiacenza Compresse la Visita in Ampiezza richiede tempo $\Theta(V + E)$, in quanto visita al più ogni arco una volta, mentre è di $\Theta(V^2)$ per rappresentazioni a

matrici di adiacenza. Nel caso non orientato ogni arco può essere esaminato fino a due volte; nel migliore dei casi teorici, invece, gli archi *necessari* a connettere $|V|$ vertici sono $|V| - 1$. In pratica, l'esecuzione top-down reale finisce per ispezionare un insieme di archi ben più ampio del minimo indispensabile, proprio nelle fasi in cui la frontiera è più estesa. Il costo in memoria è di $\Theta(V)$ per il vettore dei padri.

3.1.1 Criticità della Visita in Ampiezza classica

Nei grafi di grandi dimensioni e fortemente sbilanciati nella distribuzione dei gradi, la fase centrale della Visita in Ampiezza tende a dedicare una quota rilevante del tempo a verificare archi che conducono a vertici già visitati. Quando la frontiera si amplia, la probabilità di incontrare vicini già scoperti cresce rapidamente e una parte sostanziale delle ispezioni diventa ridondante. Questo effetto è particolarmente marcato nelle reti con basso diametro e presenza di *grandi centri di collegamento* (hub, si vedano le tipologie discusse nel Capitolo 2.2), dove molti vertici vengono raggiunti nelle prime iterazioni e la frontiera raggiunge in breve dimensioni elevate.

A queste considerazioni si sommano le criticità architettoniche già evidenziate nella sintesi sui grandi grafi (Capitolo 2.3): gli accessi ai dati risultano irregolari e scarsamente localizzati, aumentano rapidamente i mancati accessi in cache e la latenza di memoria diviene il fattore dominante. L'effetto combinato, ridondanza nelle ispezioni e debole località, rende la Visita in Ampiezza classica *vincolata dalla memoria* nelle fasi centrali.

In conclusione, l'obiettivo dell'ottimizzazione non è cambiare la complessità dell'algoritmo, che rimane asintotica lineare, bensì diminuire il numero di archi *effettivamente controllati* quando la frontiera è ampia e migliorare la località degli accessi, ottenendo comunque dei benefici. Le strategie basate sul cambio di direzione (*bottom-up* e la Visita in Ampiezza Ottimizzata per Direzione) nascono precisamente per intervenire su questi due punti.

3.2 Ottimizzazione della direzione: idea, strategia e costo

Lo schema top-down diventa inefficiente quando la frontiera supera una certa soglia⁽⁴⁾: molte ispezioni finiscono su vertici già raggiunti. La strategia bottom-up inverte l'ottica: invece di partire dai vertici in frontiera per esplorare tutti i loro vicini, parte dall'insieme dei vertici *non visitati* e verifica se ciascuno di essi sia adiacente ad almeno un vertice in frontiera. Appena trova un vicino in frontiera, marca il vertice come scoperto e passa oltre, evitando di controllare altri vicini inutilmente. Questo riduce, in media, il numero di controlli su archi nelle fasi in cui la frontiera è grande.

La *Visita in Ampiezza Ottimizzata per Direzione* (DO-BFS) alterna dinamicamente top-down e bottom-up in base a una condizione di soglia sulla frontiera (scelta euristica). Nella pratica, si inizia in top-down (quando la frontiera quindi è

piccola) e si passa al bottom-up quando la frontiera cresce oltre una certa frazione dei vertici non ancora visitati; più avanti quando la frontiera torna a ridursi, si ritorna al top-down.

Di seguito è presente lo pseudocodice della Visita in Ampiezza Ottimizzata per Direzione, i dettagli implementativi sono rimandati all'Appendice.

Pseudocodice Visita in Ampiezza Ottimizzata per Direzione.

```
function breadth-first-search(vertices, source)
    frontier ← {source}
    next ← {}
    parents ← [-1, -1, ..., -1]
    while frontier != {} do
        choice ← heuristic-choice(choice, vertices, frontier)
        if choice ← 'TOP-DOWN':
            top-down-step(vertices, frontier, next, parents)
        if choice ← 'BOTTOM-UP':
            bottom-up-step(vertices, frontier, next, parents)
        frontier ← next
        next ← {}
    end while
    return tree
```

Pseudocodice bottom-up.

```
function bottom-up-step(vertices, frontier, next, parents)
    for u ∈ unvisited_vertices do:
        if parents[u] = -1 then
            for neighbor n ∈ neighbors[u] do
                if n ∈ frontier then
                    parents[u] ← n
                    next ← next ∪ {u}
                    break
                end if
            end for
        end if
    end for
    return
```

Questo schema rende esplicito il punto di forza del bottom-up: la possibilità di “fermarsi anticipatamente” (*break*) al primo vicino in frontiera e, quindi, visitare meno archi quando la frontiera è vasta.

Il costo del solo top-down è proporzionale agli archi della componente connessa della sorgente: ogni arco incidente ai vertici in frontiera viene in media controllato almeno una volta e, nei grafi non orientati, potenzialmente due (3.2).

Il bottom-up, al contrario, sostituisce molte di queste ispezioni con ricerche da vertici non ancora visitati, che si arrestano al primo riscontro di adiacenza con la

frontiera, riducendo il numero di controlli complessivi proprio quando la frontiera è ampia.

I grafi a basso diametro e con distribuzione del grado non uniforme (con grandi centri di collegamento) costituiscono il caso d'uso privilegiato: l'espansione iniziale è rapida, la frontiera diventa grande e la *scelta euristica* porta benefici significativi.

Una grande criticità della Visita in Ampiezza Ottimizzata per Direzione sono invece i grafi sparsi e non connessi (basso grado, alto diametro, scarso fattore di raggruppamento), in cui la frontiera rimane ridotta per gran parte dell'esplorazione.

Supponiamo che al livello i vi siano $u = |U_i|$ vertici non ancora visitati. Il *limite superiore* del lavoro nel passo bottom-up è

$$m_u(i) = \sum_{u \in U_i} d(u) = u \cdot \bar{d}(U_i) \leq u \cdot \Delta, \quad (3.4)$$

dove $\bar{d}(U_i)$ è il grado medio sui nodi non ancora visitati e Δ il grado massimo, ma grazie all'arresto anticipato (`break`) il lavoro effettivo per livello è in media molto inferiore; quando la frontiera è ampia, si avvicina a un termine lineare in $|U_i|$.

Il limite inferiore è uguale $n - 1$, pari al numero di archi dell'albero di visita. In pratica, il lavoro effettivo si colloca fra $n - 1$ e m (o $2m$ non orientato), con scostamento che cresce nelle fasi centrali su grafi con frontiere ampie.

Il lavoro è nel complesso è quindi pari a, denotando con $G_s = (V_s, E_s)$ la componente connessa raggiungibile a partire dalla sorgente s , con $n = |V_s|$ e $m = |E_s|$, e sia $D = \max_{v \in V_s} \text{dist}(s, v)$, allora

$$\sum_i m_u(i) \quad (3.5)$$

dove, a differenza di 3.2, $m_u(i)$ è un limite superiore per livello; sommando sui livelli lo stesso vertice può contribuire più volte, quindi $m_u(i)$ può superare $m_f(i)$ anche di un fattore pari al numero di livelli (diametro della componente connessa), quindi $m_u(i)$ è potenzialmente maggiore di m .

Il costo computazionale sequenziale del bottom-up è dell'ordine di

$$O\left(\sum_i m_u(i)\right). \quad (3.6)$$

proporzionale al lavoro. Ulteriori costi sono $O(n)$ per le assegnazioni dei padri e a $O(\sum_i |F_{i+1}|) = O(n)$ per la costruzione delle frontiere successive.

Dunque, la stima del lavoro e della complessità asintotica dell'algoritmo di Visita in Ampiezza Ottimizzata per Direzione, si esprimono, rispettivamente, nelle seguenti forme:

$$\sum_i \min(m_f(i), m_u(i)) \quad (3.7)$$

Nel passo bottom-up al livello i , il lavoro upper bound è $m_u(i) = \sum_{u \in U_i} d(u)$, ma grazie all'arresto anticipato, il lavoro *effettivo* per livello risulta spesso molto

inferiore. Quando la frontiera è ampia (tipico delle reti a basso diametro), la maggior parte dei vertici non visitati trova rapidamente un genitore: in pratica, il lavoro per livello si avvicina a un termine lineare in $|U_i|$.

Con questa lettura, il lavoro complessivo della Visita in Ampiezza Ottimizzata per Direzione può essere descritta come la somma di tre fasi:

$$\underbrace{\sum_{i \text{ iniziali}} m_f(i)}_{\text{frontiere piccole: regime top-down}} + \underbrace{\sum_{i \text{ centrali}} |U_i|}_{\text{frontiere ampie: regime bottom-up}} + \underbrace{\sum_{i \text{ finali}} m_f(i)}_{\text{coda sottile: ritorno top-down}} .$$

In particolare, poiché ogni vertice entra una sola volta in U_i , $\sum_i |U_i| = n - 1$.

Nel complesso vale

$$n-1 \leq \sum_i \min(m_f(i), m_u(i)) \leq \min\left(\sum_i m_f(i), \sum_i m_u(i)\right) \leq \begin{cases} m & (\text{orientato}) \\ 2m & (\text{non orientato}). \end{cases}$$

Mentre il costo asintotico della Visita in Ampiezza Ottimizzata per Direzione risulta

$$\Theta\left(\sum_i \min(m_f(i), m_u(i))\right) \quad (3.8)$$

A ogni iterazione, l'algoritmo valuta dinamicamente quale strategia comporti un costo computazionale inferiore, per farlo un'euristica ben definita è fondamentale.

3.2.1 Euristica per la scelta della direzione

La transizione dinamica tra le due strategie è guidata da una euristica che stima, a ogni livello, il lavoro atteso in top-down rispetto al bottom-up e sceglie la modalità che richiede meno controlli sugli archi. L'idea, proposta da Beamer et al⁽⁴⁾, vale sia per grafi non orientati sia per grafi orientati (in questi ultimi è necessario disporre anche del grafo inverso per applicare il passo bottom-up).

Sia F la frontiera corrente e U l'insieme dei vertici non ancora visitati. Indichiamo con

$$m_f = \sum_{v \in F} d(v) \quad \text{e} \quad m_u = \sum_{u \in U} d(u)$$

il numero di controlli d'arco stimati, rispettivamente, per esplorare i vicini dei vertici in frontiera (top-down) e per cercare, per ciascun non visitato, almeno un vicino in frontiera (bottom-up). Poiché nel bottom-up l'ispezione di un non visitato termina al primo riscontro di un vicino in frontiera, m_u è in realtà una *stima superiore*; il parametro $\alpha > 0$ compensa questa sovrastima.

Al livello i , con $n_f(i) = |F_i|$ e $n_u(i) = |U_i|$, e con $m_f(i) = \sum_{v \in F_i} d(v)$, $m_u(i) = \sum_{u \in U_i} d(u)$ (stima superiore per via dell'arresto anticipato), la strategia ibrida usa:

$$\text{passa a bottom-up se } m_f(i) > \frac{m_u(i)}{\alpha}, \quad \text{torna a top-down se } n_f(i) < \frac{n_u(i)}{\beta},$$

con $\alpha = 14$ e $\beta = 24$ come valori robusti riportati in⁽⁴⁾.

Capitolo 4

Visita in Ampiezza su architetture multi-thread

La Visita in Ampiezza (BFS) è un algoritmo fortemente dipendente dalla gerarchia di memoria: gli accessi ai nodi adiacenti risultano irregolari e caratterizzati da scarsa località. L'introduzione del parallelismo, anche a memoria condivisa, può offrire vantaggi rilevanti: durante ciascun livello di esplorazione, i nodi appartenenti alla frontiera possono essere processati in maniera indipendente, consentendo una suddivisione del lavoro tra più thread; ma al tempo stesso accentua tali limitazioni: se non gestite correttamente, i miglioramenti ottenibili restano marginali. Inoltre, l'esecuzione multi-thread comporta ulteriori sfide legate al bilanciamento del carico, alla contesa e alla sincronizzazione.

4.1 Strutture dati, organizzazione della memoria e cenni di gestione multi-thread

La scelta di adeguate strutture dati è parte fondamentale di queste accortezze. Consideriamo la rappresentazione del grafo in formato di *Matrici di Adiacenza Compressa* (la più indicata per il contesto). Sia `offset` un vettore di lunghezza $|V|+1$ e `adj` un vettore di lunghezza $|E|$; i vicini di v sono memorizzati nell'intervallo semiaperto $[\text{offset}[v] : \text{offset}[v+1])$ del vettore `adj`. Questa struttura, contigua e compatta, riduce i mancati riscontri in cache durante la scansione dei vicini e si presta tanto al passo *top-down* quanto al passo *bottom-up*.

Lo stato dell'esplorazione include:

- `parents`: vettore di interi inizializzato a -1 (non visitato), aggiornato in modo atomico al primo arrivo;
- `frontier` e `next_frontier`: vettori compatti per la frontiera corrente e quella successiva; nella versione multi-thread si usano *frontiere locali*, una per thread con un consolidamento finale;
- `frontier_bitset`: rappresentare la frontiera anche come maschera binaria consente test di appartenenza $O(1)$ tramite operazioni *bitwise* e migliorando anche la località (l'intera maschera può risiedere in cache);

- contatori ausiliari (dimensione della frontiera, numero di non visitati) e, se richiesto, somme di gradi per l'euristica di cambio di direzione, tenuti in memoria per un accesso costante.

Il contesto considerato è quello della memoria condivisa su architetture multi-core. Più *thread* eseguono in concorso nello stesso spazio di indirizzamento, accedendo a strutture comuni (grafo, frontiere), con la necessità però, di orchestrare con attenzione sincronizzazione, contesa e località.

Il modello di esecuzione è *a livelli*: ciascun livello elabora la frontiera corrente e costruisce la frontiera successiva. Le sincronizzazioni globali si collocano ai confini tra livelli, mentre all'interno del livello si privilegia la *sincronizzazione minimale* tramite operazioni atomiche. L'assegnazione del padre (parents) di un vertice viene realizzata con un'unica istruzione di tipo *compare-and-swap* (CAS), che garantisce che il primo thread che scopre il vertice ne stabilisca il genitore; gli altri thread osservano l'aggiornamento e proseguono senza lock esplicativi. Per ridurre la contesa sulle strutture condivise, ciascun thread produce la propria porzione di frontiera successiva in un contenitore *locale*, che viene consolidato al termine del livello con una fusione lineare.

Per minimizzare l'*overhead* di sincronizzazione si adottano vettori allineati a multipli della dimensione di linea di cache, si evitano scritture *falsamente condivise* (false-sharing) e si aggregano le riduzioni (somme e minimi/massimi) in buffer locali per poi combinarle con un'unica fase.

Le scritture falsamente condivise si verificano quando thread distinti aggiornano variabili diverse che risiedono nella stessa *linea di cache*⁽¹¹⁾, i core devono continuamente invalidarsi a vicenda quella linea per mantenere la coerenza, generando traffico inutile tra le cache che può rallentare molto l'esecuzione, anche se in realtà i thread non stanno condividendo davvero la stessa variabile.

Infine, la scelta della *pianificazione* (scheduling) dei thread influisce direttamente sul bilanciamento del carico. L'eterogeneità dei gradi induce porzioni di lavoro molto differenti; per questo si preferisce una politica dinamica con *chunk* di dimensione moderata, che ripartisce progressivamente il lavoro fra i thread.

4.2 Visita in Ampiezza ottimizzata per Direzione multi-thread

Presentiamo ora le tre varianti della Visita in Ampiezza: top-down multi-thread, bottom-up multi-thread ed infine la combinazione dinamica di entrambe, la *Visita in Ampiezza Ottimizzata per Direzione multi-thread*. Gli pseudocodici sono dati a livello concettuale; i dettagli d'implementazione e la libreria *OpenMP* sono specificati nell'appendice.

Nel passo top-down la sorgente della computazione è la frontiera. La frontiera corrente viene ripartita tra i thread; per ciascun vertice assegnato si scorrono i vicini

nella matrice compressa e, per ogni vicino non ancora visitato, si tenta l'assegnazione del padre con un operazione atomica (compare-and-swap). In caso di successo, il vicino viene inserito nella frontiera locale del thread. Al termine, tutte le frontiere locali sono fuse nell'ordine del numero dei thread per poi formare la frontiera successiva. Questa modalità è particolarmente efficace nelle fasi iniziali, quando la frontiera è piccola e il rapporto segnale/rumore (nuove scoperte rispetto a controlli ridondanti) è elevato.

Pseudocodice in C semplificato del passo top-down multi-thread con frontiere locali.

```
function top-down-step(vertices, frontier, next, parents)
    #pragma omp parallel for schedule(dynamic, CHUNK)
    for v ∈ frontier do:
        for e ∈ [offset[v], offset[v+1]) do:
            u ← adj[e]
            if (__sync_bool_compare_and_swap(&parents[u], -1, v)) do:
                local_frontier[omp_get_thread_num()] ← push(u)
                //omp_get_thread_num = numero del thread corrente
            end if
        end for
    end for
    // Fusione delle frontiere locali → next_frontier
```

Nel passo bottom-up la sorgente della computazione è l'insieme dei non visitati. L'insieme dei vertici non ancora scoperti viene ripartito tra i thread; per ciascun vertice assegnato si cercano i vicini fino al primo riscontro di appartenenza alla frontiera corrente (verifica in $O(1)$ su `frontier_bitset`). Al primo riscontro si assegna il padre e si inserisce il vertice nella frontiera locale. Grazie all'arresto anticipato, il numero di controlli su archi si riduce proprio quando la frontiera è molto estesa, cioè nelle fasi centrali dell'esecuzione. Inoltre, a differenza del top-down multi-thread, non necessita dell'uso di alcuna operazione atomica su `parents`.

Pseudocodice in C semplificato del passo bottom-up multi-thread con BitSet di frontiera.

```
function bottom-up-step(vertices, frontier, next, parents)
    #pragma omp parallel for schedule(dynamic, CHUNK)
    for u in unvisited_vertices do:
        for e in range offset[u] do:
            v ← adj[e]
            if frontier_bitset(v) exists do:
                parents[u] ← v; // primo riscontro
                local_frontier[omp_get_thread_num()] ← push(u)
                break; // arresto anticipato
            end if
        end for
    end for
    // Fusione delle frontiere locali → next_frontier
```

La Visita in Ampiezza Ottimizzata per Direzione multi-thread mantiene lo stesso codice della versione sequenziale (3.3), perciò non verrà riportato.

In ambito multi-thread l'euristica preserva il suo razionale e aggiunge un vantaggio pratico: in bottom-up la suddivisione per non visitati riduce contese e favorisce la scalabilità su reti con grandi centri di collegamento (hub).

Si passa a bottom-up quando $m_f > m_u/\alpha$, mentre si torna a top-down quando $|F| < |U|/\beta$, con parametri α, β , i quali valgono rispettivamente 14 e 24, valori ritenuti migliori e robusti dopo varie sperimentazioni⁽⁴⁾.

In pratica, le somme di gradi sono mantenute e aggiornate lungo i livelli con riduzioni locali tramite OpenMP; la costruzione del `frontier_bitset` precede il passo bottom-up ed è lineare nella dimensione della frontiera ($O(|F_i|)$ per livello, con azzeramento del bitset).

4.3 Organizzazione pratica dei dati, Analisi dei costi e scalabilità

L'organizzazione dei dati è fondamentale per un effettivo guadagno nelle prestazioni di una configurazione multi-thread.

L'efficienza richiede che il costo aggiuntivo delle strutture ausiliarie rimanga marginale rispetto al lavoro di esplorazione. La costruzione del `frontier_bitset` procede azzerando i bit della frontiera precedente e impostando i bit corrispondenti alla frontiera corrente; l'operazione è contigua e beneficia della sequenzialità delle scritture. Le frontiere locali vengono allocate una sola volta e riutilizzate livello dopo livello, dimensionate per assorbire picchi di inserimenti senza riallocazioni troppo frequenti.

Per mitigare la *falsa condivisione* (false sharing) in cache tra i thread, si allineano i puntatori delle frontiere locali alla dimensione della linea di cache.

Le riduzioni di m_f e m_u vengono effettuate in due tempi: accumulo locale durante la scansione e combinazione globale in coda al livello.

La scalabilità reale è limitata da tre fattori principali: (i) *ridondanza* nelle ispezioni di archi nelle fasi centrali (mitigata dal bottom-up); (ii) *contesa* su strutture condivise (mitigata tramite funzioni atomiche compare-and-swap, frontiere locali e riduzioni) e *pianificazione* dei thread adeguata; (iii) *debole località* che rende la Visita in Ampiezza sensibile alla banda di memoria più che al numero di core.

In pratica, su macchine multi-core, top-down scala bene nei primi livelli (frontiera piccola), mentre bottom-up è preferibile nelle fasi centrali con frontiere ampie e reti a basso diametro; la versione Ottimizzata per Direzione massimizza il minimo tra i due costi per livello e tende a produrre i migliori tempi complessivi.

Con l'analisi dei costi condotta per livello, nel passo top-down il costo computazionale multi-thread, indicato con C_{TD} , è proporzionale al lavoro, come per la versione sequenziale $m_f(i)$ (3.1), divenendo circa

$$C_{TD}(i) \approx \frac{\phi(i)}{p} m_f(i) + O(\text{sync}(i)) + O(\text{mem}), \quad (4.1)$$

dove p è il numero di thread e $\phi(i) \geq 1$ il fattore di sbilanciamento che misura quanto il lavoro del livello i è distribuito male tra i thread; a questo si sommano i costi aggiuntivi, come costi di sincronizzazione (una operazione atomica compare-and-swap riuscita per ogni vertice scoperto, al più $n - 1$, e fusione delle frontiere locali e riduzioni) e i costi *mem*, incorporando l'effetto della banda limitata di memoria, per via di località e affinità.

Il costo complessivo del Top-down, risulta approssimativamente di

$$\sum_i C_{\text{TD}}(i) = \sum_i \frac{\phi(i)}{p} m_f(i) + \sum_i O(\text{sync}(i)) + O(\text{mem}) \quad (4.2)$$

Per il passo bottom-up per livello, il costo della versione sequenziale $m_u(i)$ (3.4) diviene, indicato con C_{BU} , circa di

$$C_{\text{BU}}(i) \approx \frac{1}{p} m_u(i) + O(\text{sync}(i)) + O(\text{mem}), \quad (4.3)$$

considerando i thread bilanciati, ai quali si sommano marginali costi aggiuntivi, che includono: con *mem* la costruzione/azzeramento del **frontier_bitset** con complessità in upper bound $O(|F_i|)$ per livello, circa $O(n)$, oltre alla località e affinità in memoria; e con *sync* la fusione delle frontiere locali e riduzioni.

Conseguentemente, il costo totale del Bottom-up con thread idealmente bilanciati, è approssimato a

$$\sum_i C_{\text{BU}}(i) = \sum_i \frac{1}{p} m_u(i) + \sum_i O(\text{sync}(i)) + O(\text{mem}) \quad (4.4)$$

Considerando, infine, la Visita in Ampiezza Ottimizzata per Direzione, e dividendo idealmente per il numero di thread p tenendo conto dei costi di sincronizzazione e memoria, si ottiene una stima al livello:

$$C(i) \approx \frac{\phi(i)}{p} \min(C_{\text{TD}}(i), C_{\text{BU}}(i)) + C_{\text{sync}}(i) + C_{\text{mem}}(i), \quad (4.5)$$

dove $C_{\text{sync}}(i)$ riassume il contributo della operazione atomica (compare-and-swap), fusione delle frontiere locali e riduzioni mentre $C_{\text{mem}}(i)$ la gestione del *bitset* e la località in memoria. In pratica, la scalabilità è condizionata da tre fattori: la riduzione della ridondanza nelle fasi centrali (a favore del bottom-up), il bilanciamento del carico in presenza di grandi centri di collegamento (favorito da *chunk* dinamici e dalla ripartizione per non visitati) e la banda di memoria effettivamente disponibile (migliorato dall'allocazione contiguo).

Il costo complessivo della Visita in Ampiezza Ottimizzata per Direzione sarà circa di

$$\sum_i C(i) \approx \frac{1}{p} \sum_i \min(C_{\text{TD}}(i), C_{\text{BU}}(i)) + \sum_i C_{\text{sync}}(i) + \sum_i C_{\text{mem}}(i). \quad (4.6)$$

considerando un buon bilanciamento tra i diversi thread.

Per quanto riguarda la *correttezza degli algoritmi*, le varianti multi-thread preservano la validità della Visita in Ampiezza a livelli: l'adozione dell'approccio bottom-up non modifica le distanze minime, mentre la non-deterministicità si limita all'*albero dei padri*, a causa dell'ordine di scoperta dei nodi entro lo stesso livello. Ai fini della verifica è stato utilizzato l'array delle *profondità* (distance-from-source) come criterio robusto, invariato tra diverse esecuzioni e varianti, confrontando infine gli *output* di profondità.

Capitolo 5

Risultati Sperimentali

Al fine di valutare le prestazioni degli algoritmi di visita su grafi, con particolare attenzione alle varianti *Top-Down* e *Visita in Ampiezza Ottimizzata per Direzione (Direction-Optimizing Breadth-First Search)*, sono stati condotti esperimenti su due tipologie di grafi: *grafo sintetici*, generati artificialmente, e *grafo reali*, ottenuti da portali e repository pubblici.

I dataset utilizzati per i grafi reali sono stati scaricati da fonti autorevoli nel contesto della ricerca scientifica, tra cui il *Stanford Network Analysis Project (SNAP)*⁽³⁾ e il *Network Repository*⁽²⁾. Tali piattaforme mettono a disposizione insiemi di dati di natura strutturale (principalmente grafi), appositamente progettati per scopi accademici e sperimentali.

Il formato più comune di tali dataset è un file di testo, in cui ogni riga rappresenta un arco del grafo secondo la notazione:

```
<nodo_sorgente> <nodo_destinazione>
```

All'interno dell'implementazione sperimentale, è stata sviluppata un'interfaccia per il parsing di tali file. Tale interfaccia si basa su una funzione che legge il file riga per riga, estraendo gli archi e memorizzandoli in una struttura dati adatta alla rappresentazione del grafo, come ad esempio le liste di adiacenza o la matrice compressa.

5.1 Metodologia di Test

Per ciascun grafo, l'algoritmo è stato eseguito più volte variando il nodo sorgente utilizzato per l'avvio dell'esplorazione. I nodi sorgente sono stati selezionati in maniera casuale, ma memorizzati in modo da garantire la consistenza tra i test comparativi. Questo ha reso possibile l'esecuzione di ciascun algoritmo sul medesimo sottoinsieme di nodi sorgente, assicurando così l'equità della valutazione sperimentale tra le diverse varianti implementative.

Per ogni combinazione di algoritmo e grafo, sono stati registrati i seguenti indicatori prestazionali: **Tempo minimo** di esecuzione tra tutte le sorgenti testate; **Tempo massimo** di esecuzione; **Tempo medio** di esecuzione.

5.2 Versioni dell'Algoritmo di Visita in Ampiezza

Nel presente studio sono state analizzate sei diverse varianti dell'algoritmo di Visita in Ampiezza (*Breadth-First Search*), ciascuna caratterizzata da una diversa rappresentazione del grafo e, in alcuni casi, da specifiche ottimizzazioni orientate alle prestazioni.

Le versioni considerate sono le seguenti:

- A. *BFS con Matrice di Adiacenza*
- B. *BFS con Liste di Adiacenza*
- C. *BFS con Matrice di Adiacenza Compressa*
- D. *BFS ottimizzata per Direzione su Matrice di Adiacenza*
- E. *BFS ottimizzata per Direzione su Liste di Adiacenza*
- F. *BFS ottimizzata per Direzione su Matrice di Adiacenza Compressa*

Oltre alle versioni sequenziali, è stata studiata anche la controparte *multithread* di ciascuna variante, eseguita con un numero fisso di thread pari a 8. Ciò ha permesso di valutare l'impatto del parallelismo su prestazioni e scalabilità in scenari multi-core, mettendo in luce, in alcuni casi, anche le relative criticità. In linea con Beamer *et al.*⁽⁴⁾, in contesti fortemente paralleli ci si attendono benefici significativi, soprattutto quando le frontiere sono ampie e i costi di sincronizzazione sono contenuti.

5.3 Sperimentazioni con Grafi Generati

I grafi caratterizzati da basso diametro e da una distribuzione del grado secondo una legge di potenza risultano particolarmente adatti a tecniche di ottimizzazione quali la **Visita in Ampiezza ottimizzata per Direzione** (*Direction-Optimizing BFS*).

Per analizzare il comportamento e l'efficacia dell'algoritmo in differenti contesti strutturali, sono stati considerati sia grafi regolari, con particolare attenzione agli effetti delle componenti sconnesse, sia grafi generati secondo modelli noti, quali il modello di Newman–Watts^(?), grafi a basso diametro, e Holme–Kim⁽⁷⁾, evoluzione del modello di Barabási-Albert⁽⁸⁾, noti per la crescita preferenziale e la struttura a rete complessa.

5.3.1 Risultati per i Grafi Regolari Casuali

L'algoritmo è stato analizzato secondo diverse metodologie, applicate a ciascuna delle versioni implementate. In particolare, sono state considerate le seguenti metriche: *Tempo medio di esecuzione* al variare del numero di nodi n , mantenendo fisso il numero di archi $e = 5n$; $e = 2\log n$; $e = \frac{1}{3}n^2$.

I risultati ottenuti sono rappresentati graficamente in funzione della crescita di n :

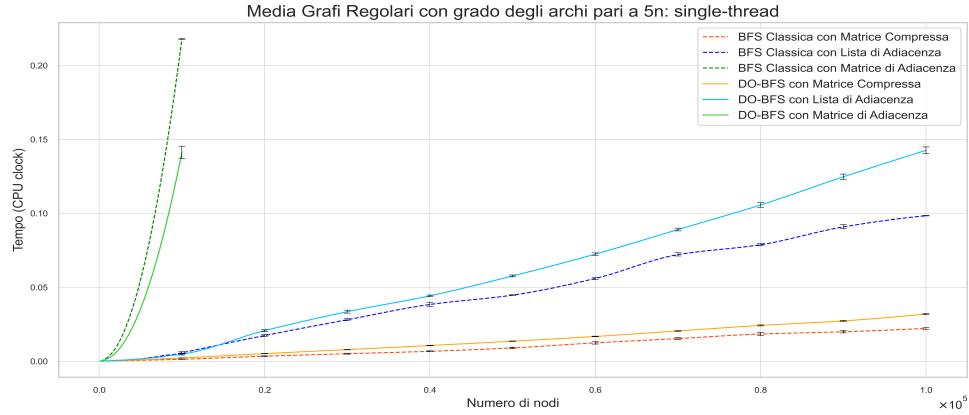


Figura 5.1 Prestazioni su grafi regolari: tempo medio vs n con numero di archi $e = 5n$, fino a $n = 1 \times 10^5$.

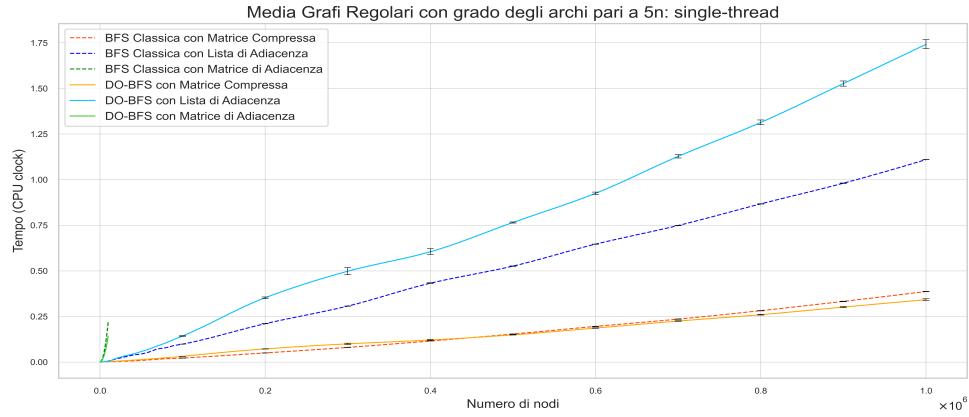


Figura 5.2 Prestazioni su grafi regolari: tempo medio vs n con numero di archi $e = 5n$, fino a $n = 1 \times 10^6$.

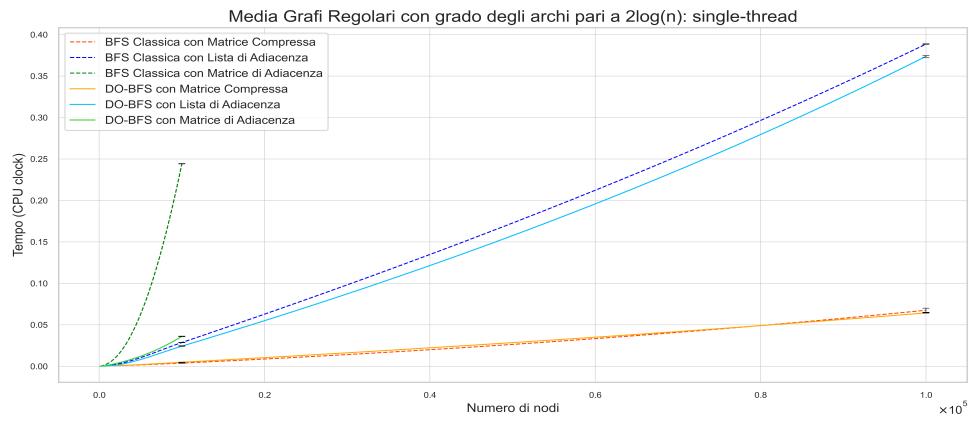


Figura 5.3 Prestazioni su grafi regolari: tempo medio vs n con numero di archi $e = 2\log n$, fino a $n = 1 \times 10^5$.

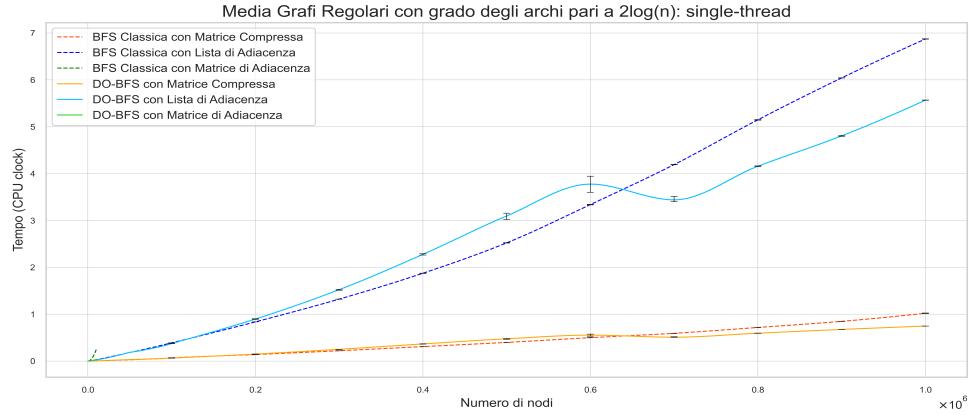


Figura 5.4 Prestazioni su grafi regolari: tempo medio vs n con numero di archi $e = 2\log n$, fino a $n = 1 \times 10^6$.

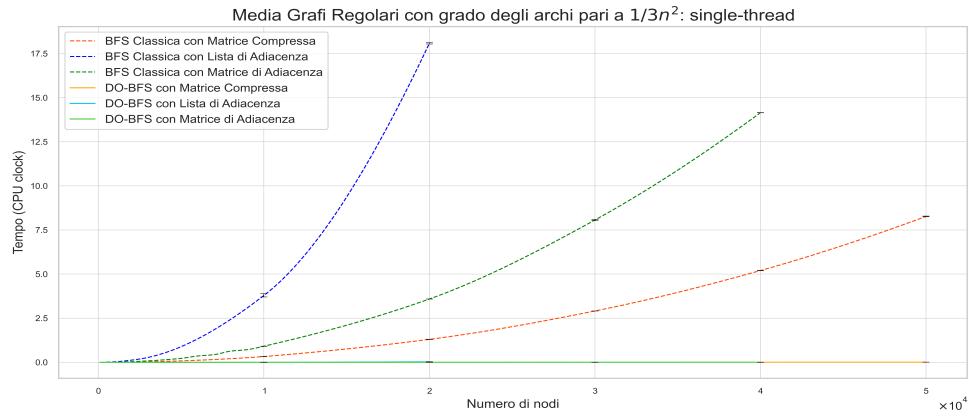


Figura 5.5 Prestazioni su grafi regolari: tempo medio vs n con numero di archi $e = \frac{1}{3}n^2$, fino a $n = 0.5 \times 10^5$.

Di seguito anche le versioni multi-thread, con $p = 8$:

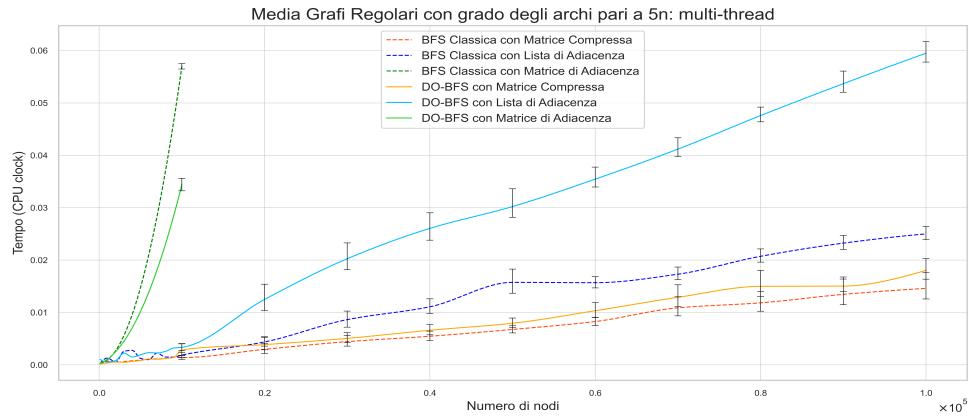


Figura 5.6 Prestazioni su grafi regolari: tempo medio vs n con numero di archi $e = 5n$, fino a $n = 1 \times 10^5$.

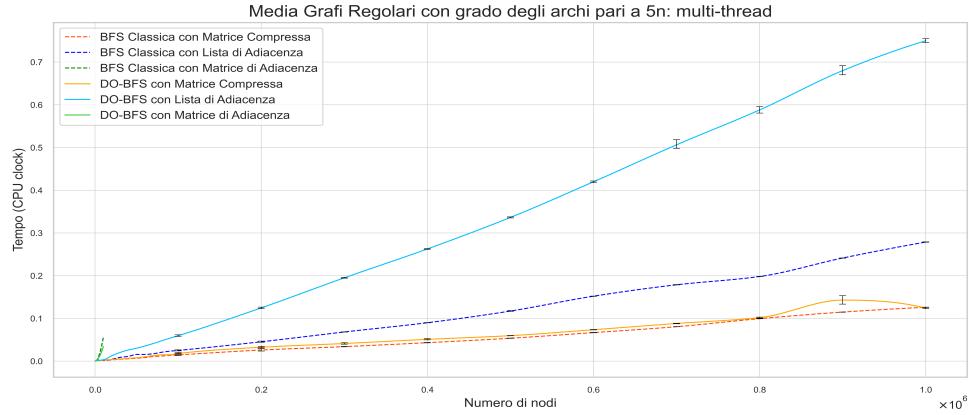


Figura 5.7 Prestazioni su grafi regolari: tempo medio vs n con numero di archi $e = 5n$, fino a $n = 1 \times 10^6$.

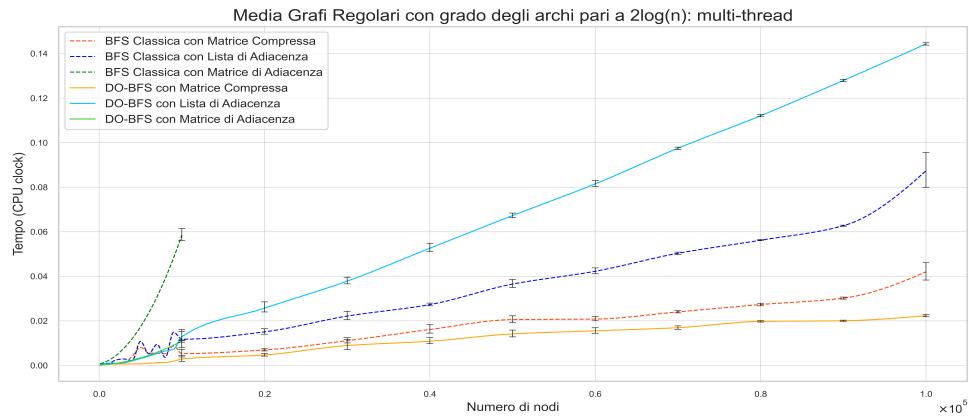


Figura 5.8 Prestazioni su grafi regolari: tempo medio vs n con numero di archi $e = 2\log n$, fino a $n = 1 \times 10^5$.

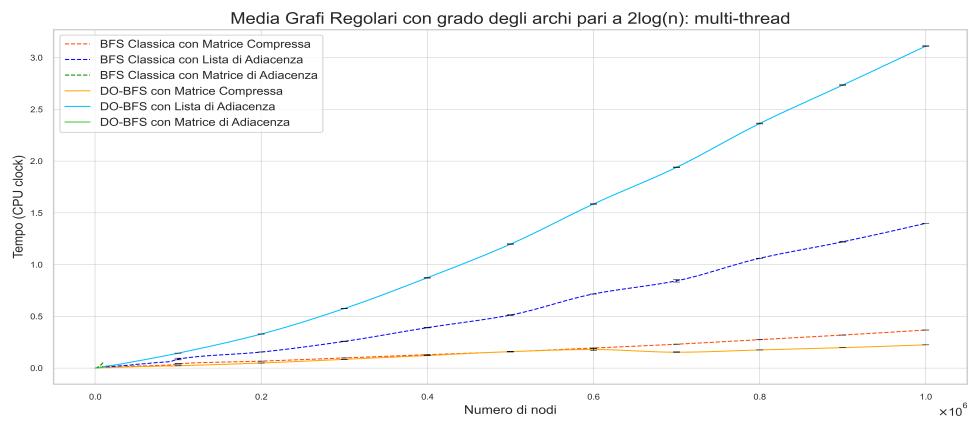


Figura 5.9 Prestazioni su grafi regolari: tempo medio vs n con numero di archi $e = 2\log n$, fino a $n = 1 \times 10^6$.

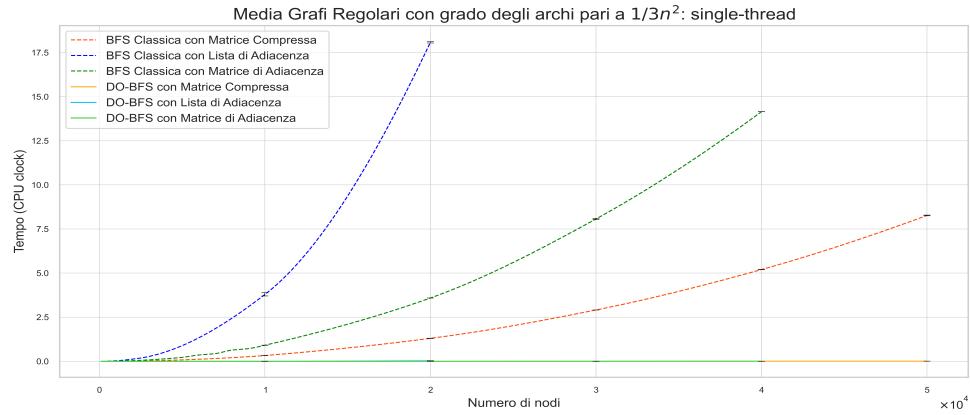


Figura 5.10 Prestazioni su grafi regolari: tempo medio vs n con numero di archi $e = \frac{1}{3} n^2$, fino a $n = 0.5 \times 10^5$.

Inoltre, è stato confrontato il numero di adiacenze totali per entrambe le versioni della *Visita in Ampiezza*, risultando al variare di n in media:

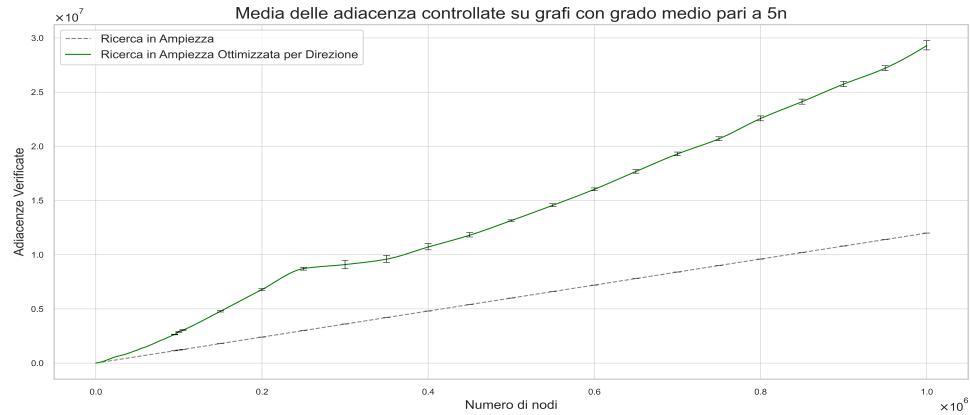


Figura 5.11 Grafico del numero di adiacenze dei grafi regolari all'aumentare dei nodi n con $e = 5n$, fino a $n = 1 \times 10^6$.

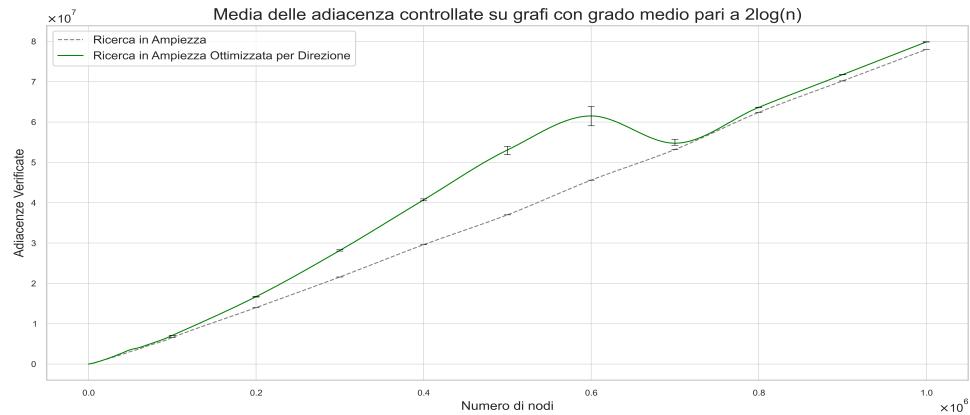


Figura 5.12 Grafico del numero di adiacenze dei grafi regolari all'aumentare dei nodi n con $e = 2\log(n)$, fino a $n = 1 \times 10^6$.

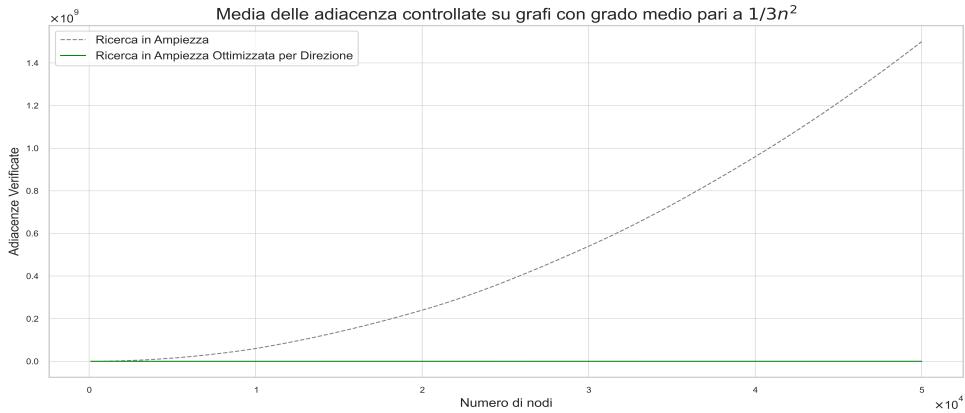


Figura 5.13 Grafico del numero di adiacenze dei grafi regolari all'aumentare dei nodi n con $e = \frac{1}{3} n^2$, fino a $n = 1 \times 10^6$.

La differenza di prestazioni tra la Visita in Ampiezza classica e quella ottimizzata per direzione non risulta significativa, anzi, nei grafi poco connessi (ordine di $5n$ archi) e in di componenti connesse piccole, l'approccio bottom-up ha mostrato la sua debolezza, con un numero di adiacenze controllate maggiore di quelle esplorate dal top-down.

Nel caso delle liste di adiacenza, inoltre, le prestazioni peggiorano ulteriormente, con una Visita in Ampiezza classica fino a due volte più veloce rispetto alla versione ottimizzata per direzione. Tale risultato può essere attribuito a diversi fattori legati alla struttura delle liste di adiacenza: implicano accessi puntuali e iterazioni per nodo con molte dereferenziazioni, causando salti di cache e overhead; di conseguenza, l'ottimizzazione per direzione, che richiede scansioni rapide degli archi dalla frontiera, risulta meno efficace quando le liste sono di lunghezza variabile e sparse in memoria.

Tuttavia, un miglioramento delle prestazioni è osservabile nelle matrici di adiacenza compresse nei grafi più connessi e densi. Sebbene inizialmente il vantaggio sia minimo o addirittura negativo per grafi con pochi nodi, esso cresce all'aumentare della dimensione del grafo. Questo avviene perché le componenti connesse crescono, e diventano pari o quasi al grafo nella sua interezza. Solo in presenza di grandi dimensioni la struttura e dinamica dell'algoritmo emergono chiaramente, permettendo di sfruttare appieno le ottimizzazioni implementate e osservare guadagni significativi in termini di efficienza computazionale. Inoltre aumenta in base al coefficiente del grado dei nodi.

5.3.2 Il modello di Newman–Watts

Il modello di Newman–Watts⁽⁹⁾ (NW) genera grafi *small-world* ed è una variante del modello Watts–Strongatz, scelto per conservare l'elevato *coefficiente di raggruppamento* tipico di questi grafi e, al contempo, ridurre drasticamente la *distanza media* tra coppie di nodi introducendo alcune scorciatoie che collegano punti lontani nel grafo (archi che collegano nodi distanti).

Composizione e proprietà Sia n il numero di nodi e k (pari) il grado regolare del reticolo iniziale su anello: ogni nodo è connesso ai suoi $k/2$ vicini immediati a sinistra e $k/2$ a destra. Dato un parametro $p \in [0, 1]$, per ogni coppia non adiacente di nodi si aggiunge, in modo indipendente, un arco *scorciatoia* con probabilità p . Il reticolo regolare garantisce un coefficiente di raggruppamento (clustering) elevato, mentre pochi archi scorciatoia casuali sono sufficienti a ridurre il diametro e la distanza media, producendo la tipica transizione small-world 2.2.

Il modello Newman–Watts conserva buona parte del raggruppamento del reticolo iniziale e, per valori modesti di p , ottiene una distanza media che cresce lentamente con n (regime small-world) senza indurre una distribuzione del grado a legge di potenza: la distribuzione resta “stretta” attorno a k con una coda leggermente allungata dovuta agli archi scorciatoia, in contrasto con i modelli *scale-free* (Barabasi–Albert/Holme–Kim).

5.3.3 Risultati per i grafi generati utilizzando il modello Newman–Watts

Per ogni variante implementata abbiamo applicato la stessa procedura di analisi ai grafi Newman–Watts, valutando in particolare il *tempo medio di esecuzione* al variare del numero di nodi n , mantenendo fissati il grado iniziale k e la probabilità di ri-collegamento p , di seguito i risultati single-thread:

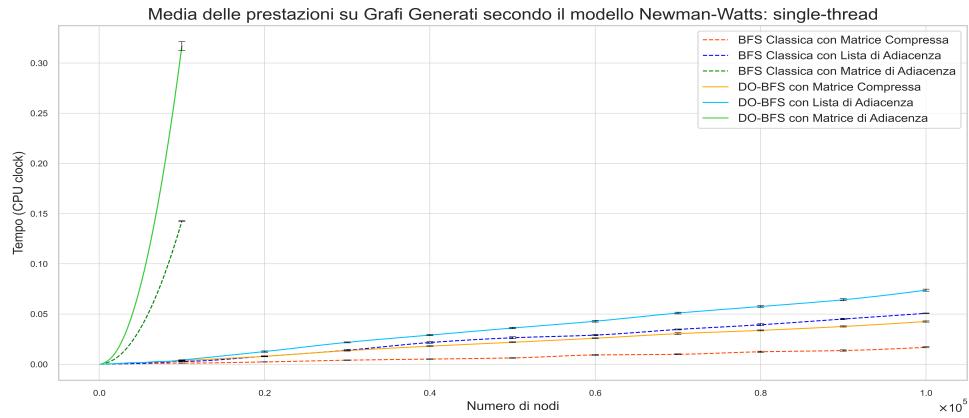


Figura 5.14 Prestazioni su grafi Newman–Watts: tempo medio vs n con parametri $k = 10$ e $p = 0.07$, fino a $n = 1 \times 10^5$.

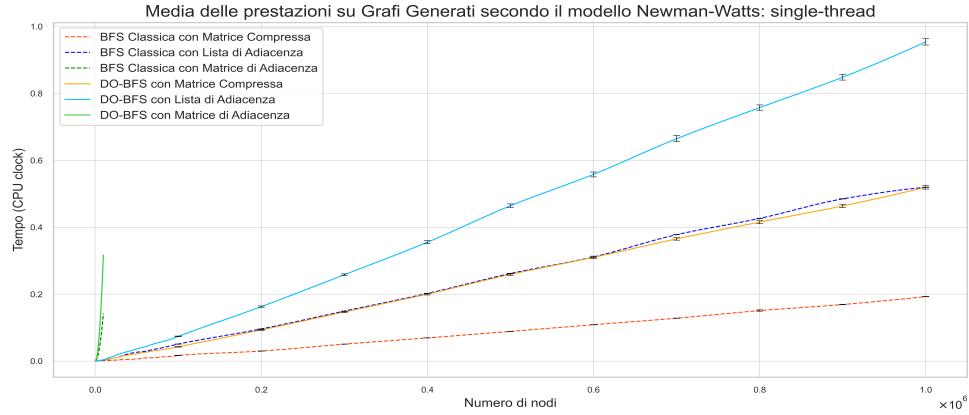


Figura 5.15 Prestazioni su grafi Newman-Watts: tempo medio vs n con parametri $k = 10$ e $p = 0.07$, fino a $n = 1 \times 10^6$.

e le rispettive versioni multi-thread:

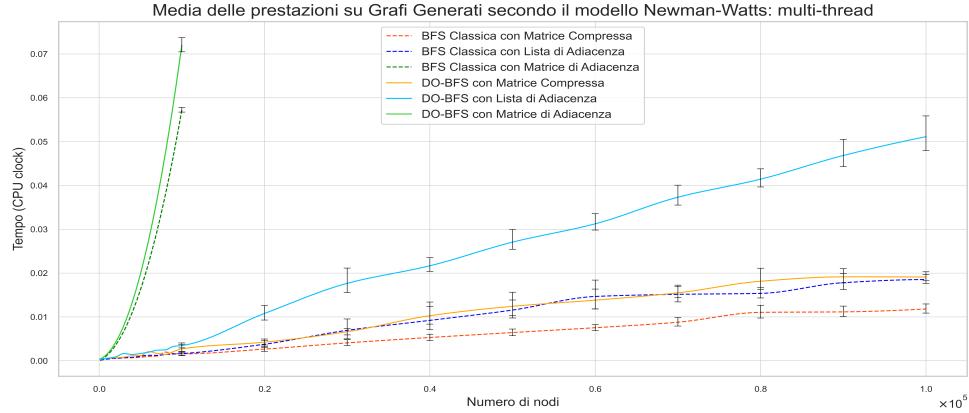


Figura 5.16 Prestazioni multi-thread su grafi Newman-Watts: tempo medio vs n con parametri $k = 10$ e $p = 0.07$, fino a $n = 1 \times 10^5$.

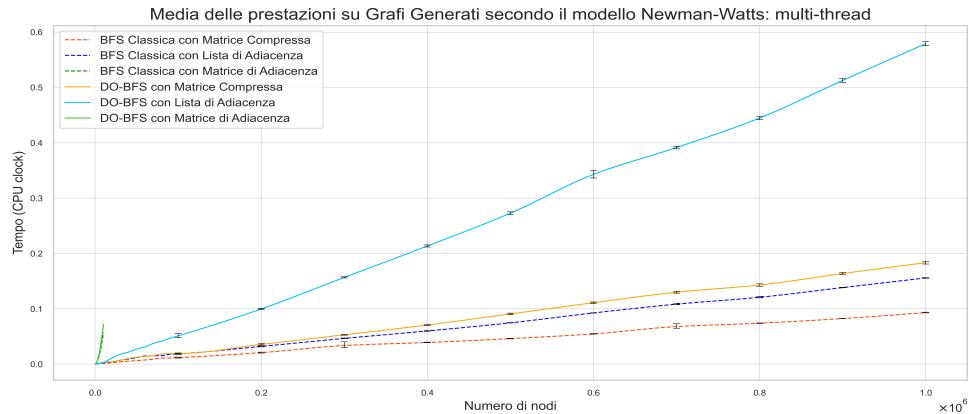


Figura 5.17 Prestazioni multi-thread su grafi Newman-Watts: tempo medio vs n con parametri $k = 10$ e $p = 0.07$, fino a $n = 1 \times 10^6$.

Anche per questo modello è stato studiato il numero di adiacenze controllato da ciascuna versione, ottenendo:

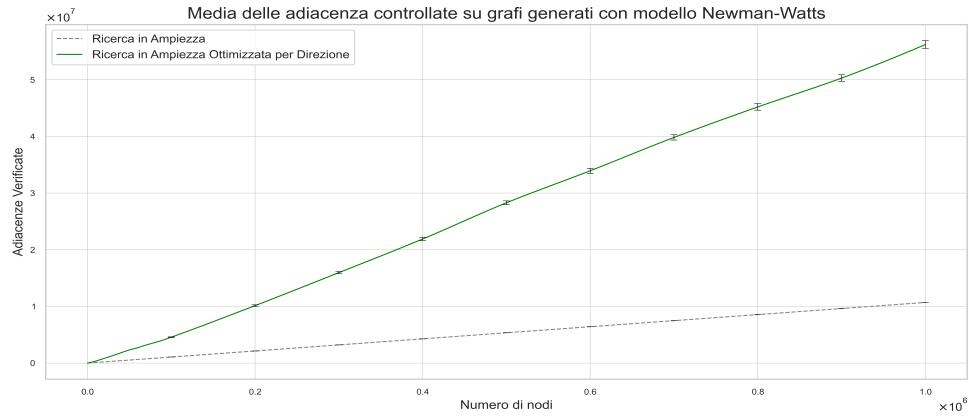


Figura 5.18 Grafico del numero di adiacenze dei grafi Newman–Watts all'aumentare dei nodi n con parametri $k = 10$ e $p = 0.07$, fino a $n = 1 \times 10^6$.

Nei grafi sintetici generati secondo il modello di Newman–Watts, con parametri $k = 10$ e $p = 0.07$, il diametro è ridotto ma il coefficiente di raggruppamento resta moderato e mancano hub marcati; nelle fasi centrali la probabilità di arresto precoce nel passo bottom–up non è sufficiente a ridurre il lavoro rispetto al top–down, e i costi di gestione (bitset, scansione dei non visitati) tendono a prevalere.

Ne deriva che, nelle fasi centrali, i vertici non ancora visitati incontrano con frequenza non sufficiente un vicino già in frontiera: la probabilità di fermarsi anticipatamente tramite passo bottom–up non è abbastanza alta da ridurre sensibilmente i confronti di adiacenza rispetto al passo top–down.

Ecco quindi alcune ricerche effettuate aumentando il parametro p :

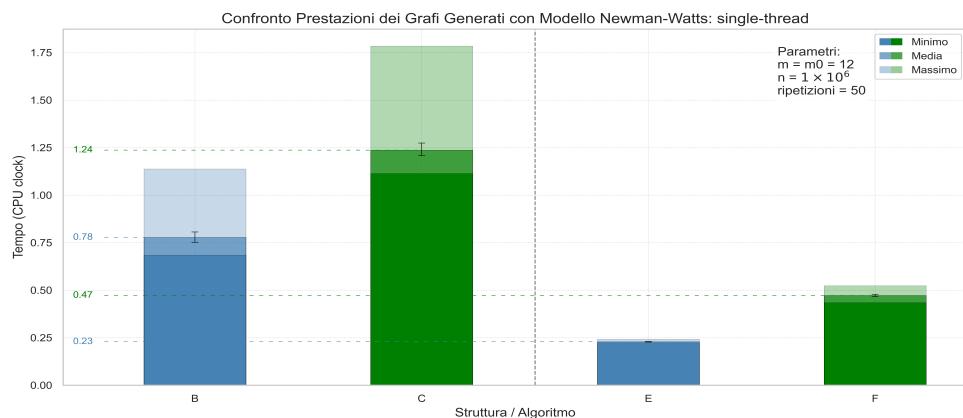


Figura 5.19 Grafico prestazionale dei grafi Newman–Watts con parametri $k = 10$ e $p = 0.17$, ed $n = 1 \times 10^6$.

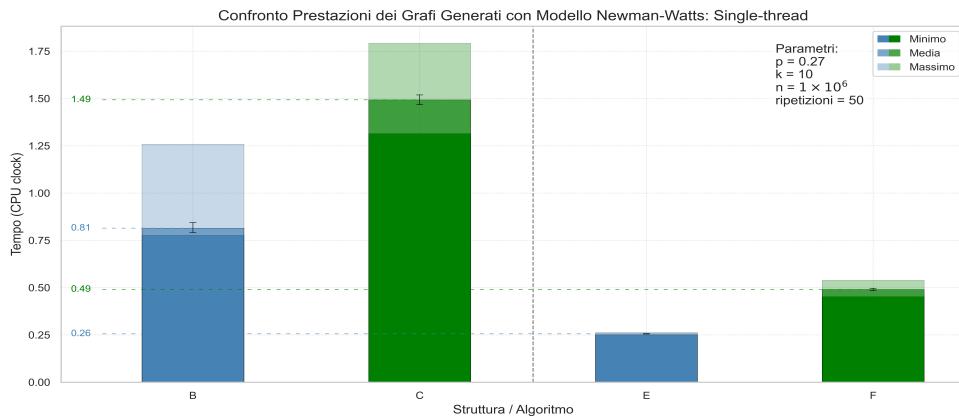


Figura 5.20 Grafico prestazionale dei grafi Newman–Watts con parametri $k = 10$ e $p = 0.27$, ed $n = 1 \times 10^6$.

e le rispettive versioni multi-thread:

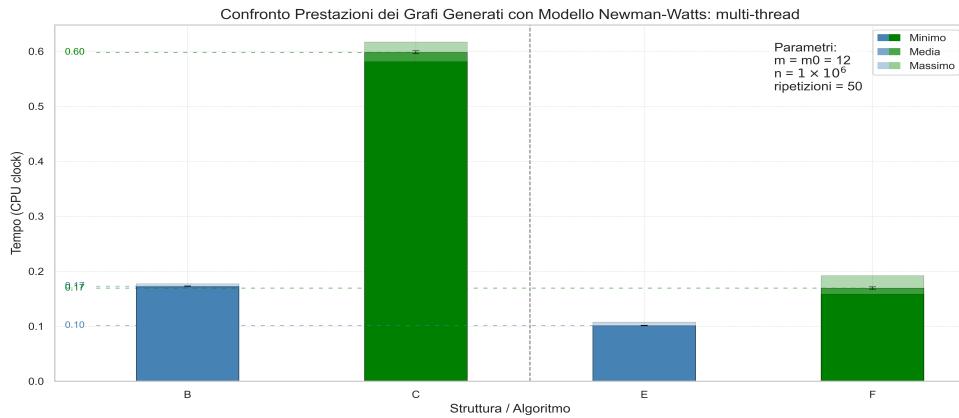


Figura 5.21 Grafico prestazionale dei grafi Newman–Watts multi-thread con parametri $k = 10$ e $p = 0.17$, ed $n = 1 \times 10^6$.

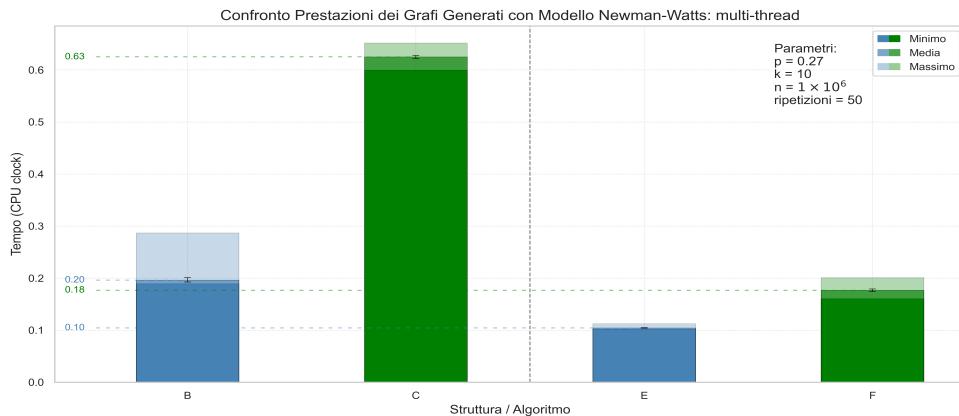


Figura 5.22 Grafico prestazionale dei grafi Newman–Watts multi-thread con parametri $k = 10$ e $p = 0.27$, ed $n = 1 \times 10^6$.

In presenza di un grafo con coefficiente di raggruppamento più alto le prestazioni della *Visita in Ampiezza Ottimizzata per Direzione* migliorano sensibilmente, con anche migliorie del lavoro sui test di adiacenze:

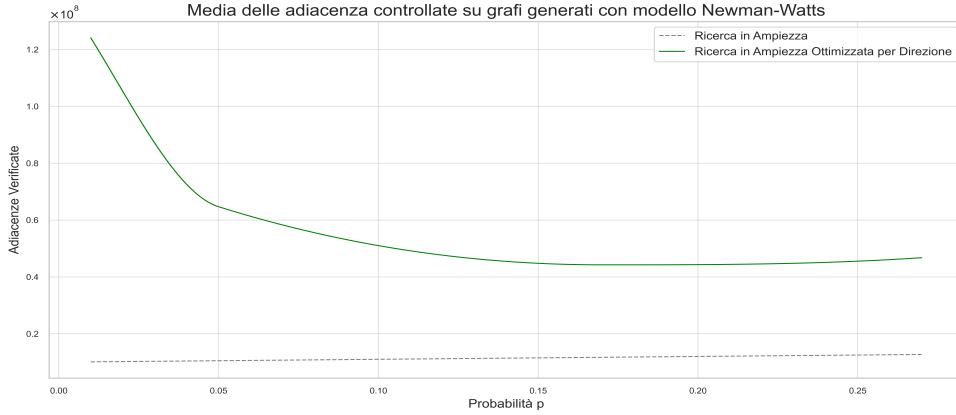


Figura 5.23 Confronto delle adiacenze Newman–Watts con p crescente e parametro $k = 10$, ed $n = 1 \times 10^6$.

5.3.4 Il modello di Barabási-Albert

Il modello di *Barabási-Albert* (BA)⁽⁸⁾ descrive un processo di generazione di grafi scale-free, caratterizzati da una distribuzione del grado che segue una legge di potenza. In tali grafi, la maggior parte dei nodi possiede un basso numero di connessioni, mentre un numero ristretto di nodi altamente connessi, detti *hub*, concentra una parte significativa delle connessioni complessive.

La costruzione del grafo secondo il modello BA si fonda su due principi fondamentali: la **crescita incrementale** e l'**attaccamento preferenziale**.

Il processo ha inizio con un grafo iniziale costituito da m_0 nodi connessi tra loro a formare una *clique completa*, ossia un sottografo completamente connesso. A ogni passo temporale, un nuovo nodo viene aggiunto al grafo e collegato a m nodi preesistenti, dove $m \leq m_0$, proprietà della *crescita incrementale*.

In questo contesto, il parametro m rappresenta il numero minimo di archi con cui ciascun nuovo nodo viene connesso al grafo, mentre m_0 indica il numero di nodi che costituiscono il sottografo iniziale completamente connesso.

La selezione dei nodi a cui il nuovo nodo si collega avviene secondo un criterio di *attaccamento preferenziale*: la probabilità $\Pi(i)$ che un nodo i venga scelto come punto di connessione è proporzionale al suo grado k_i , secondo la seguente relazione:

$$\Pi(i) = \frac{k_i}{\sum_j k_j},$$

dove la sommatoria al denominatore è calcolata su tutti i nodi esistenti al momento dell'inserimento del nuovo nodo. Questo meccanismo implica che i nodi con un grado elevato tendano a ricevere un numero crescente di connessioni nel tempo, favorendo la formazione di hub e conferendo al grafo una struttura scale-free.

Proprietà principali

Il modello di Barabási-Albert presenta diverse proprietà strutturali caratteristiche che emergono direttamente dai meccanismi di crescita e attaccamento preferenziale su cui si basa.

Una delle caratteristiche più rilevanti è la **distribuzione del grado**, che segue approssimativamente una legge di potenza. Più precisamente, la probabilità che un nodo abbia grado k è data da:

$$P(k) \sim k^{-3},$$

comportando la presenza di pochi nodi altamente connessi, e un'ampia maggioranza di nodi con grado relativamente basso. Questa distribuzione è responsabile della struttura scale-free tipica del modello.

Dal punto di vista topologico, la rete esibisce anche un **piccolo diametro**, proprietà nota come small-world. In particolare, la distanza media tra coppie di nodi cresce in maniera logaritmica rispetto alla dimensione del grafo, seguendo un ordine di grandezza pari a $\mathcal{O}(\log n)^{(8)}$. Ciò implica che, anche in grafi di grandi dimensioni, qualsiasi nodo può essere raggiunto da un altro in un numero relativamente ridotto di passaggi.

Infine, la rete mostra una marcata **robustezza strutturale**. Essa risulta infatti resiliente rispetto alla rimozione casuale di nodi, mantenendo intatte gran parte delle sue proprietà topologiche. Tuttavia, risulta significativamente più vulnerabile in caso di rimozione mirata degli hub, i quali giocano un ruolo cruciale nella connettività dell'intero grafo.

Limiti strutturali del modello di Barabási-Albert

Il modello di Barabási-Albert è uno dei modelli più noti per la generazione di reti scale-free. Nonostante la sua capacità di riprodurre alcune proprietà strutturali fondamentali osservate in reti reali, esso presenta numerose limitazioni che ne compromettono l'efficacia nella rappresentazione accurata di fenomeni topologici complessi.

Una prima criticità risiede nel *basso coefficiente di clustering* generato dal modello. In particolare, si osserva una scarsa tendenza dei vicini di un nodo a essere reciprocamente connessi, fenomeno che contrasta con quanto rilevato in molte reti reali, dove i collegamenti tra vicini sono comuni e strutturalmente significativi.

Direttamente correlata a questa carenza è la *bassa densità di triangoli*, ovvero la scarsa presenza di sottostrutture locali formate da tre nodi reciprocamente connessi. I triangoli sono elementi fondamentali per la formazione di strutture locali coese e per la modellazione di processi dinamici come la diffusione di informazioni, il contagio epidemico e l'influenza sociale.

Infine, il modello Barabási-Albert non è in grado di generare spontaneamente *strutture a comunità*, ovvero insiemi di nodi caratterizzati da una densa connettività interna e connessioni relativamente deboli con il resto del grafo. Tali strutture sono invece una caratteristica ricorrente nelle reti reali, dove le comunità rappresentano unità funzionali, organizzative o sociali ben definite.

In sintesi, sebbene il modello di Barabási-Albert sia efficace nel descrivere l'emergere di distribuzioni di grado a legge di potenza, esso non riproduce fedelmente molte delle caratteristiche strutturali delle reti reali. Per analisi più accurate o simulazioni algoritmiche è pertanto opportuno considerare modelli più sofisticati, come quello di Holme-Kim⁽⁷⁾.

5.3.5 Estensione del modello BA: il modello di Holme-Kim

Il modello di Holme-Kim (HK)⁽⁷⁾ rappresenta un'estensione naturale del modello di Barabási-Albert, progettata con l'obiettivo di includere nelle reti scale-free una proprietà rilevante spesso osservata nelle reti reali: un elevato coefficiente di clustering.

Il modello HK conserva il meccanismo di attaccamento preferenziale tipico del modello BA, ma introduce una modifica semplice ed efficace al processo di crescita. In particolare, dopo aver connesso un nuovo nodo a un nodo esistente secondo la regola dell'attaccamento preferenziale, con una certa probabilità p viene aggiunto un secondo collegamento verso un vicino del nodo appena collegato. Questo meccanismo, noto come *triad formation step*, consente la formazione di triangoli, incrementando significativamente il clustering locale della rete.

Dal punto di vista topologico, il modello HK mantiene: la distribuzione del grado secondo legge di potenza; il piccolo diametro medio tipico dei grafi scale-free.

A differenza del modello BA, esso consente di controllare e incrementare il coefficiente di clustering tramite il parametro p . All'aumentare di p , la rete tende a formare un numero maggiore di triangoli, migliorando l'aderenza alle proprietà locali osservate nelle reti reali.

È importante notare che, pur migliorando il clustering, il modello HK non modifica in modo sostanziale altri limiti strutturali del modello BA, come l'assenza di comunità. Di conseguenza, non rappresenta una replica fedele delle reti sociali reali, e i risultati ottenuti possono differire da quelli osservati nei dati empirici.

5.3.6 Risultati per i grafi generati utilizzando il modello Barabási-Albert

Per tutte le varianti implementate dell'algoritmo abbiamo applicato la stessa procedura di analisi ai grafi del modello Barabasi, valutando inizialmente il *tempo medio di esecuzione* al variare del numero di nodi n , mantenendo fissati il grado iniziale $m = m_0 = 3$, di seguito i risultati single-thread:

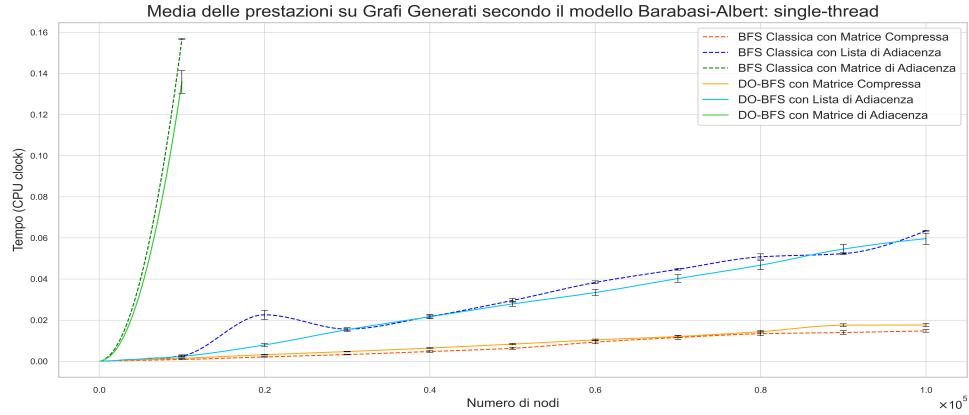


Figura 5.24 Prestazioni su grafi Barabási–Albert: tempo medio vs n con parametri $m = m_0 = 3$, fino a $n = 1 \times 10^5$.

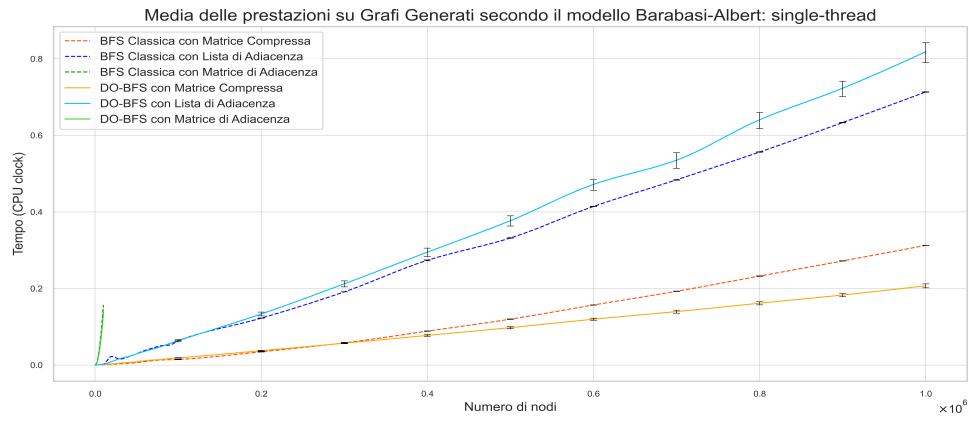


Figura 5.25 Prestazioni su grafi Barabási–Albert: tempo medio vs n con parametri $m = m_0 = 3$, fino a $n = 1 \times 10^6$.

e le rispettive versioni multi-thread:

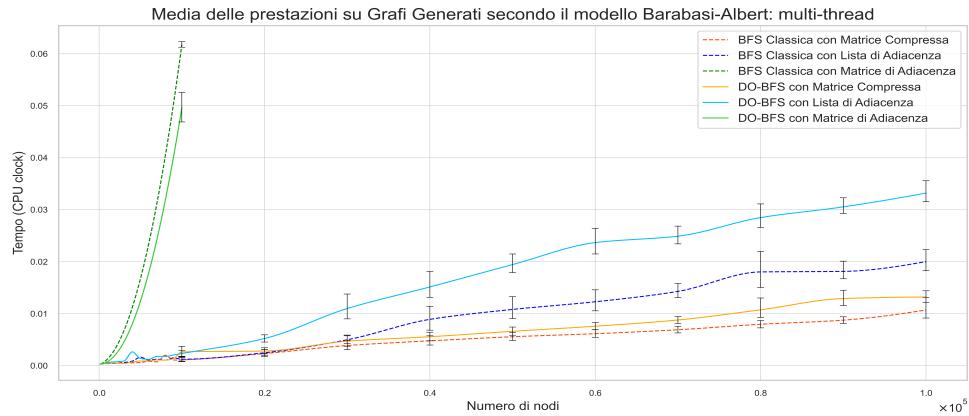


Figura 5.26 Prestazioni multi-thread su grafi Barabási–Albert: tempo medio vs n con parametri $m = m_0 = 3$, fino a $n = 1 \times 10^5$.

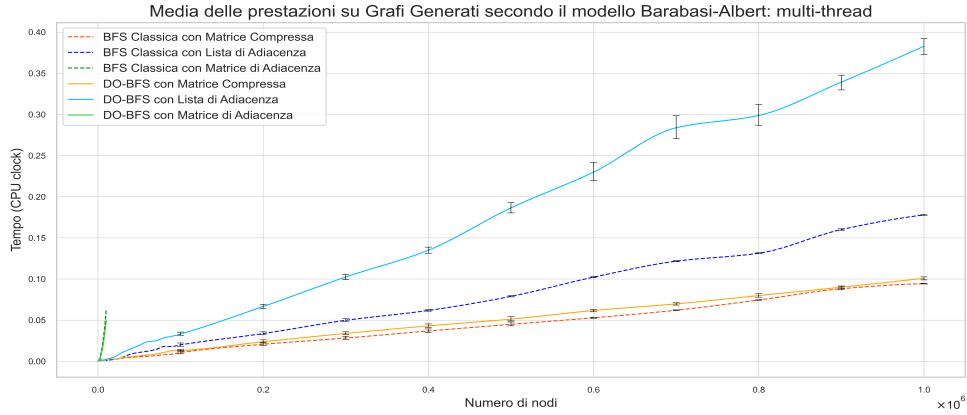


Figura 5.27 Prestazioni multi-thread su grafi Barabási–Albert: tempo medio vs n con parametri $m = m_0 = 3$, fino a $n = 1 \times 10^6$.

Nel caso dell'utilizzo delle liste di adiacenza, si osserva un peggioramento delle prestazioni dell'algoritmo DO-BFS rispetto alla Visita in Ampiezza classica (BFS). Analogamente, l'incremento prestazionale ottenuto mediante l'impiego della matrice di adiacenza compressa risulta trascurabile per un numero di nodi non sufficientemente alto.

Questo comportamento può essere attribuito alla natura fortemente sparsa dei grafi generati tramite il modello di Barabási–Albert, con parametro $m = m_0 = 3$. In tali grafi, la frontiera raramente raggiunge dimensioni sufficienti da rendere vantaggioso lo switch alla modalità *bottom-up*. Di conseguenza, l'algoritmo di Visita in Ampiezza ottimizzata per Direzione tende a rimanere prevalentemente in modalità *top-down*, oppure effettua il passaggio alla modalità *bottom-up* solo per un numero molto limitato di iterazioni. Il miglioramento complessivo rispetto all'approccio *top-down* puro risulta pertanto modesto.

Va inoltre considerato che ogni iterazione della Visita in Ampiezza ottimizzata per Direzione introduce un lieve overhead computazionale, dovuto alla necessità di stimare il lavoro residuo e di valutare, tramite euristiche, la direzione ottimale da adottare nella successiva iterazione. Questo ulteriore costo, se non bilanciato da un guadagno significativo nella fase *bottom-up*, contribuisce a ridurre l'efficacia complessiva dell'approccio di Visita in Ampiezza ottimizzata per Direzione in questo specifico contesto.

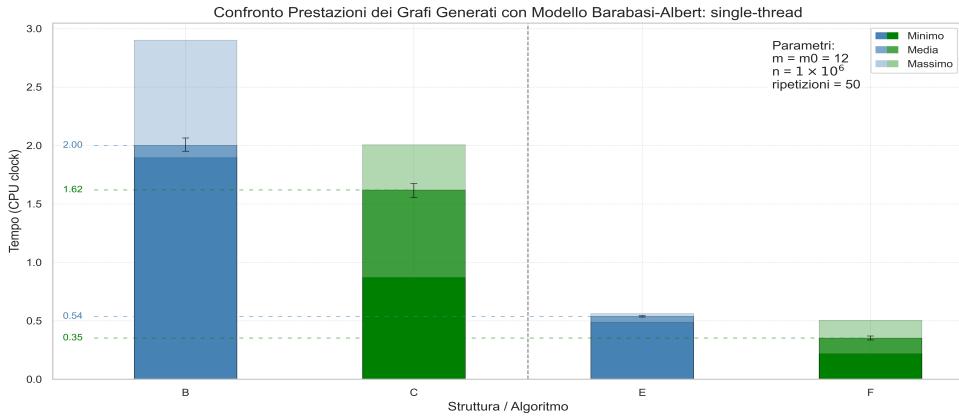


Figura 5.28 Grafico prestazionale dei grafi Barabási–Albert single-thread con parametri $m = m_0 = 12$, ed $n = 1 \times 10^6$.

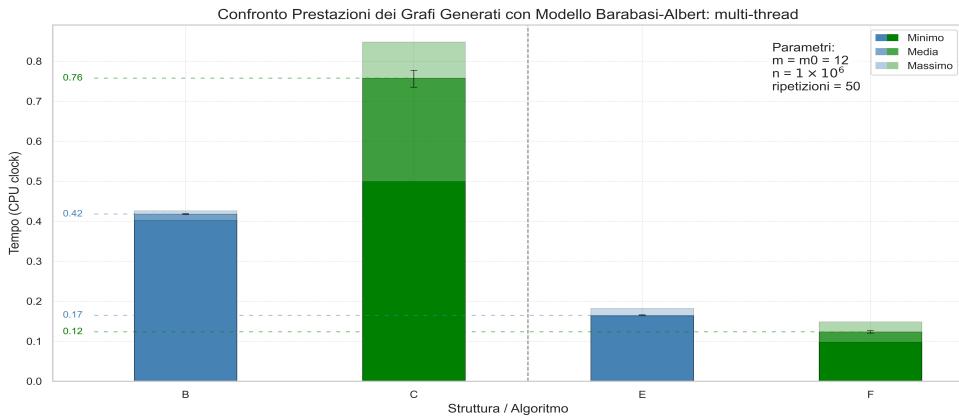


Figura 5.29 Grafico prestazionale dei grafi Barabási–Albert multi-thread con parametri $m = m_0 = 12$, ed $n = 1 \times 10^6$.

Sia nella versione single-thread che in quella multi-thread, l'aumento del grado medio dei nodi, ottenuto modificando i parametri m_0 e m , ha mostrato un lieve miglioramento delle prestazioni della rappresentazione tramite Matrici di Adiacenza Compresse nella Visita in Ampiezza ottimizzata per Direzione. Tuttavia, anche in tali condizioni, tale struttura dati continua a risultare meno efficiente rispetto alle liste di adiacenza in ambiente multi-thread. Questi risultati suggeriscono che, con tali parametri, l'assetto di Barabási–Albert resta troppo sparso per valorizzare pienamente la Visita in Ampiezza Ottimizzata per Direzione; all'aumentare di m il vantaggio cresce, ma resta sensibile alla rappresentazione dati.

5.3.7 Risultati per i grafi generati utilizzando il modello Holme–Kim

In continuità con l'analisi svolta nella sezione precedente, l'algoritmo è stato sottoposto a valutazione attraverso diverse metodologie sperimentali, applicate a ciascuna delle sue varianti. In questa fase, tuttavia, si abbandona l'approccio puramente accademico adottato in precedenza, in cui veniva analizzato anche il comporta-

mento dell'algoritmo su grafi di dimensioni contenute, a favore di un'impostazione sperimentale più orientata alla valutazione delle prestazioni in scenari realistici.

Nel modello di Holme–Kim si considerano i seguenti parametri: n indica il numero totale di nodi del grafo; m_0 è il numero di nodi del grafo iniziale (tipicamente una clique completa); m è il numero di archi che ciascun nuovo nodo introduce collegandosi a nodi già presenti, con scelta basata sull'attaccamento preferenziale. Il parametro p rappresenta la probabilità di *formazioni di triangoli*: dopo il primo collegamento per attaccamento preferenziale, per ciascuno dei restanti $m - 1$ collegamenti si aggiunge, con probabilità p , un arco verso un vicino del nodo appena collegato (chiudendo un triangolo); con probabilità $1 - p$ si continua invece con un'ulteriore scelta per attaccamento preferenziale.

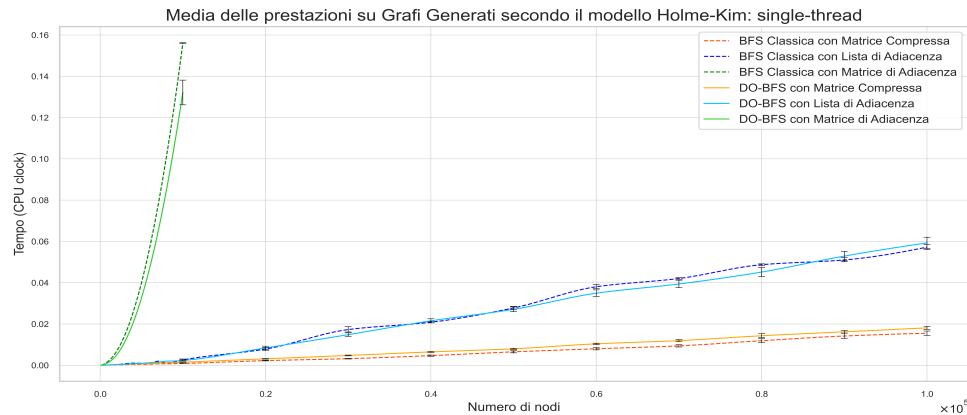


Figura 5.30 Prestazioni su grafi Holme–Kim: tempo medio vs n con parametri $m = m_0 = 3$ e $p = 0.1$, fino a $n = 1 \times 10^5$.

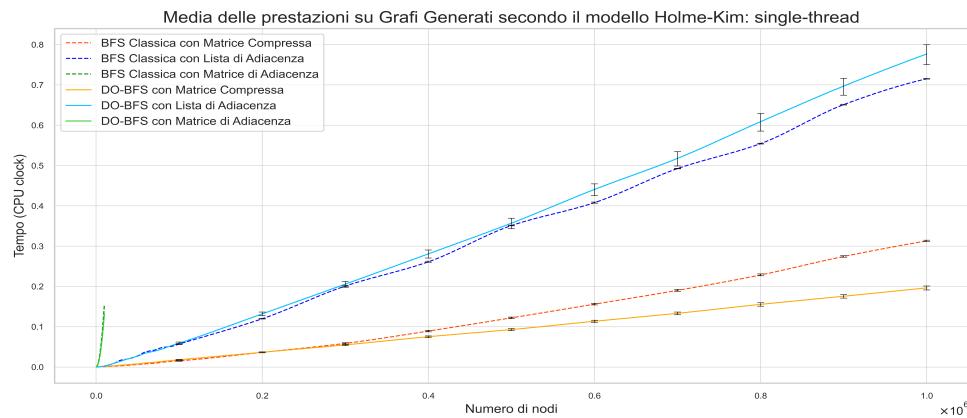


Figura 5.31 Prestazioni su grafi Holme–Kim: tempo medio vs n con parametri $m = m_0 = 3$ e $p = 0.1$, fino a $n = 1 \times 10^6$.

e le rispettive versioni multi-thread:

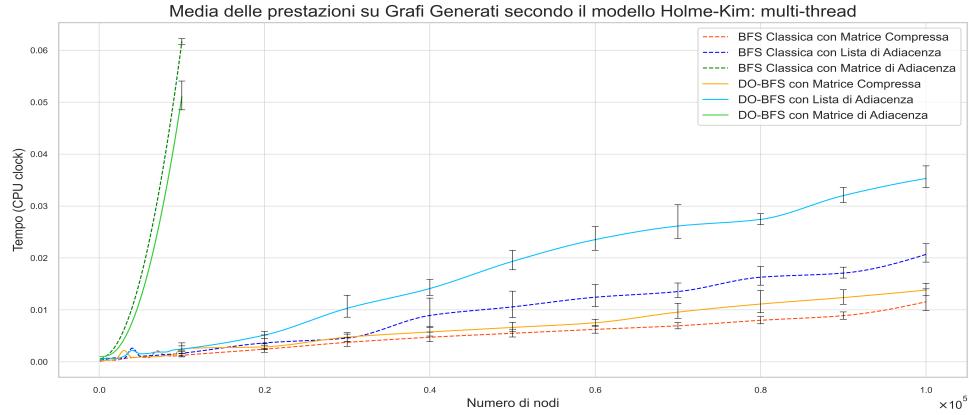


Figura 5.32 Prestazioni multi-thread su grafi Holme–Kim: tempo medio vs n con parametri $m = m_0 = 3e$ $p = 0.1$, fino a $n = 1 \times 10^5$.

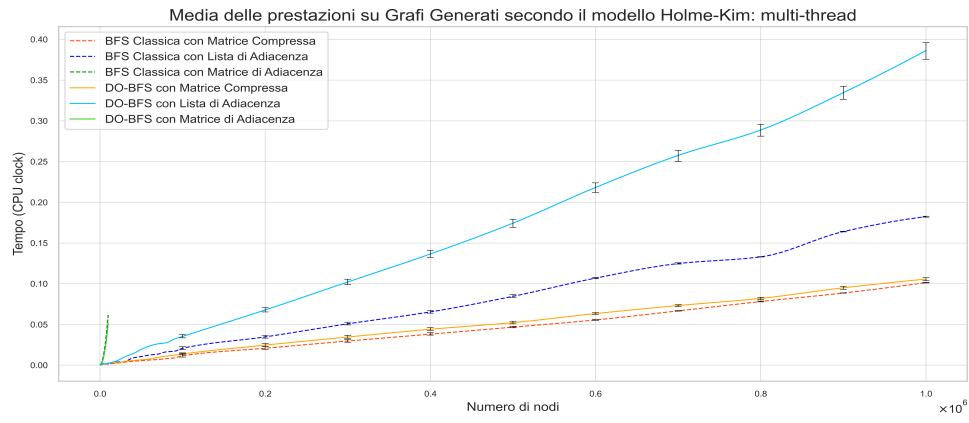


Figura 5.33 Prestazioni multi-thread su grafi Holme–Kim: tempo medio vs n con parametri $m = m_0 = 3$ e $p = 0.1$, fino a $n = 1 \times 10^6$.

In ambienti multi-threaded, seppur con capacità di parallelismo limitata (fino a 8 thread su 4 core fisici), l’accelerazione ottenuta nei nostri esperimenti risulta pari a circa $\times 2$ per la Visita in Ampiezza classica (BFS) con Matrici di Adiacenza Comprese, con un peggioramento nelle prestazioni delle liste di adiacenza.

Il carico di lavoro diminuisce oltre una certa soglia per $m = m_0$, le prestazioni migliorano sensibilmente per la matrice di adiacenza compressa, anche trattandosi di una rete con una cardinalità media dei nodi vicina a $m = 3$.

Sono stati poi condotti i test per i singoli grafi generati per n fissa, al variare di $m = m_0$:

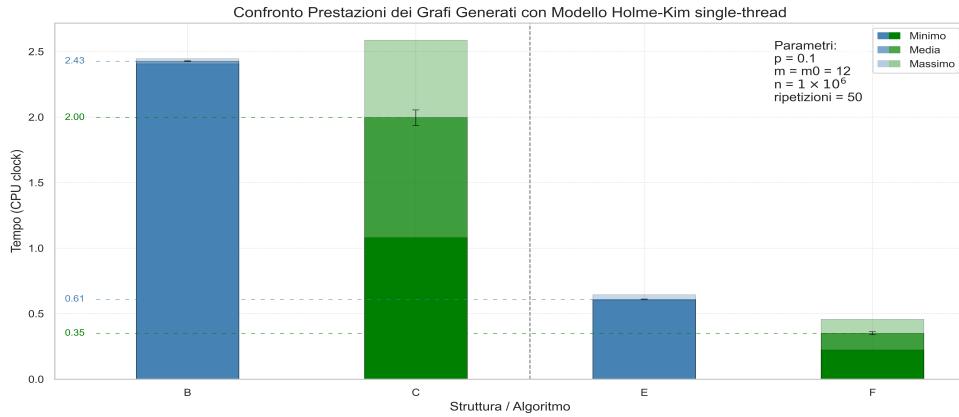


Figura 5.34 Grafico prestazionale dei grafi Holme–Kim con parametri $m = m_0 = 12$ e $p = 0.1$, ed $n = 1 \times 10^6$.

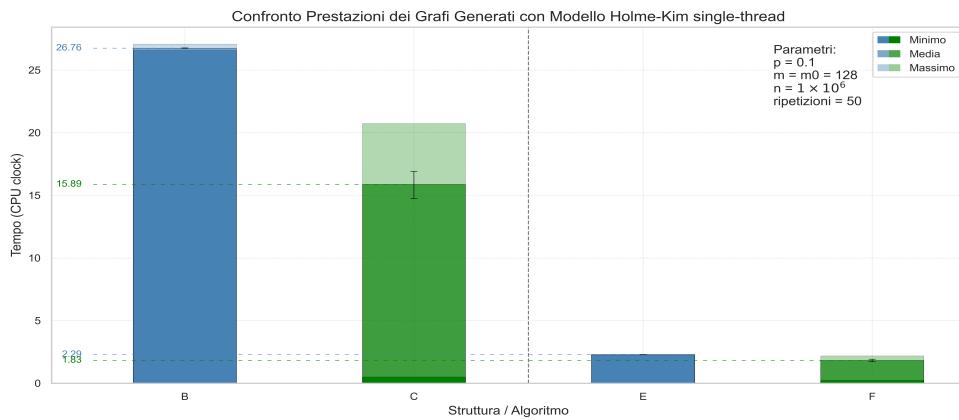


Figura 5.35 Prestazioni su grafi Holme–Kim: tempo medio vs n con parametri $m = m_0 = 128$ e $p = 0.1$, fino a $n = 1 \times 10^6$.

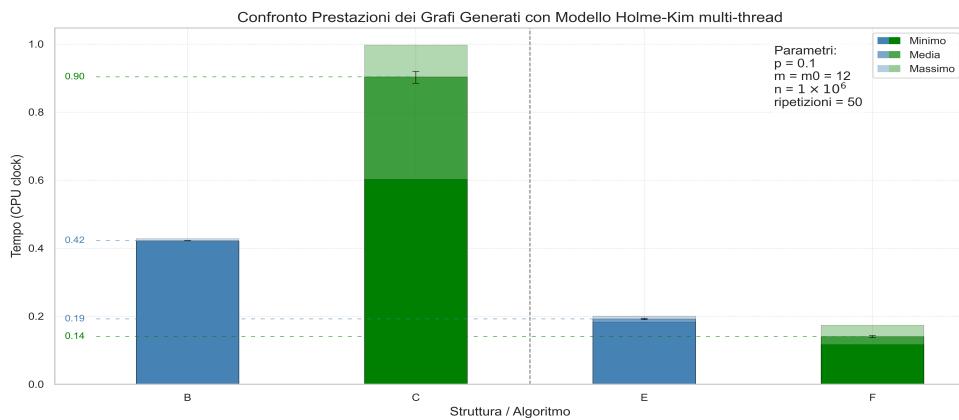


Figura 5.36 Grafico prestazionale dei grafi Holme–Kim multi-thread con parametri $m = m_0 = 12$ e $p = 0.1$, ed $n = 1 \times 10^6$.

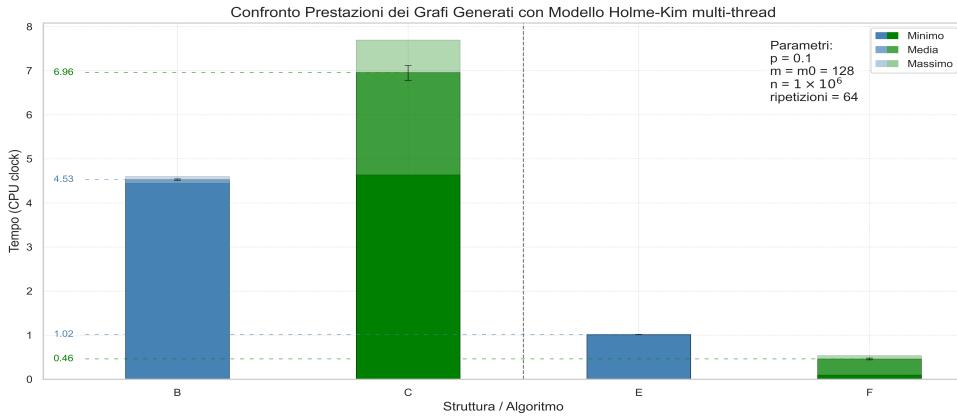


Figura 5.37 Prestazioni multi-thread su grafi Holme–Kim: tempo medio vs n con parametri $m = m_0 = 128$ e $p = 0.1$, fino a $n = 1 \times 10^6$.

con analisi del lavoro in termine di adiacenze verificate:

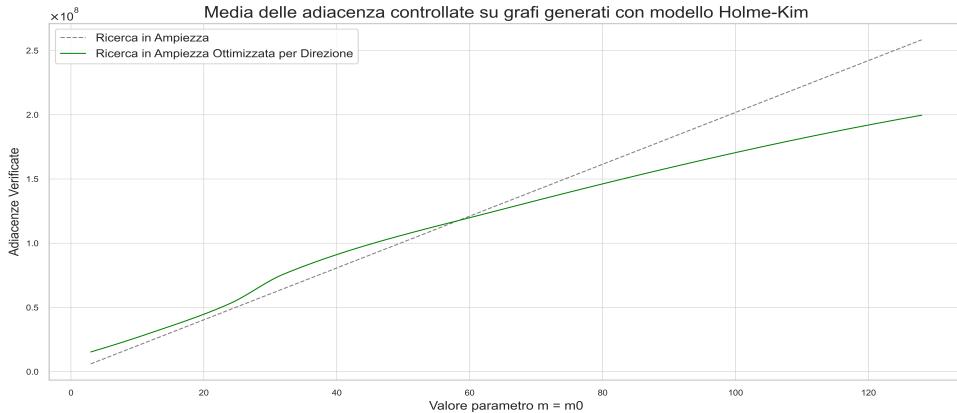


Figura 5.38 Confronto delle adiacenze Holme–Kim al variare di $m = m_0$ e con parametro $p = 0.1$, ed $n = 1 \times 10^6$.

Si osserva una maggiore variabilità delle prestazioni dell’algoritmo quando viene utilizzata la rappresentazione tramite matrice di adiacenza compressa, mentre l’utilizzo delle liste di adiacenza mostra una maggiore stabilità nei tempi di esecuzione. Tuttavia, in media, le prestazioni risultano superiori nel caso delle Matrici di Adiacenza Compresse.

Tali benefici si amplificano ulteriormente nella versione *Visita in Ampiezza ottimizzata per Direzione*, dove la presenza della modalità *bottom-up* consente di sfruttare in modo più efficiente il parallelismo anche a memoria condivisa. In questo contesto, le matrici di adiacenza compresse risultano complessivamente più performanti rispetto alle liste, grazie alla natura sequenziale e regolare degli accessi nella fase *bottom-up*, che si presta particolarmente bene all’esecuzione parallela.

L’incremento dei parametri del modello di generazione comporta una crescita esponenziale del numero di archi, mantenendo fisso il numero di nodi. Questo porta alla formazione di grafi caratterizzati da un numero crescente di hub ad alto grado, e quindi da un aumento del grado medio complessivo, inoltre garantisce un diametro medio basso.

Tale evoluzione strutturale ha un impatto positivo sull'efficacia della Visita in Ampiezza ottimizzata per Direzione (*DO-BFS*), in quanto la presenza di nodi ad alto grado favorisce la transizione verso la modalità *bottom-up*, che risulta particolarmente efficiente in questi contesti.

Come evidenziato dai risultati sperimentali, il beneficio ottenuto con la Visita in Ampiezza Ottimizzata per Direzione rispetto alla Visita in Ampiezza classica, inizialmente limitato a circa $\times 2$ in ambiente sequenziale e $\times 3$ in configurazione multi-thread, raggiunge un massimo di $\times 7$ su singolo thread e $\times 11$ in ambiente multi-thread, a parità del numero di nodi, per grafi sintetici sempre più densi e connessi su questi tipi di esperimenti.

5.4 Sperimentazioni con Dataset

La funzione responsabile della conversione del grafo in una specifica struttura dati varia in base alla rappresentazione desiderata, la metodologia di sperimentazione è la stessa usata per i singoli grafi al variare di uno o più parametri.

5.4.1 Grafi della raccolta SNAP

I grafi della raccolta SNAP⁽³⁾ variano dal piccolo al su larga scala, spaziando da poche migliaia fino a centinaia di milioni di vertici e archi, e coprono domini eterogenei (social, web, citazioni, collaborazioni, reti stradali, comunicazioni). Possono essere diretti o non diretti, talvolta pesati o temporali, e presentano caratteristiche strutturali diverse (small-world, distribuzioni heavy-tailed/scale-free, diversa densità e clustering), rendendoli adatti a valutare algoritmi come la Visita in Ampiezza e la Visita in Ampiezza Ottimizzata per Direzione. Sono stati scelti principalmente grafi web e di comunità.

Siano d il diametro del grafo, k il suo coefficiente di raggruppamento, n il numero di nodi e e il numero di archi:

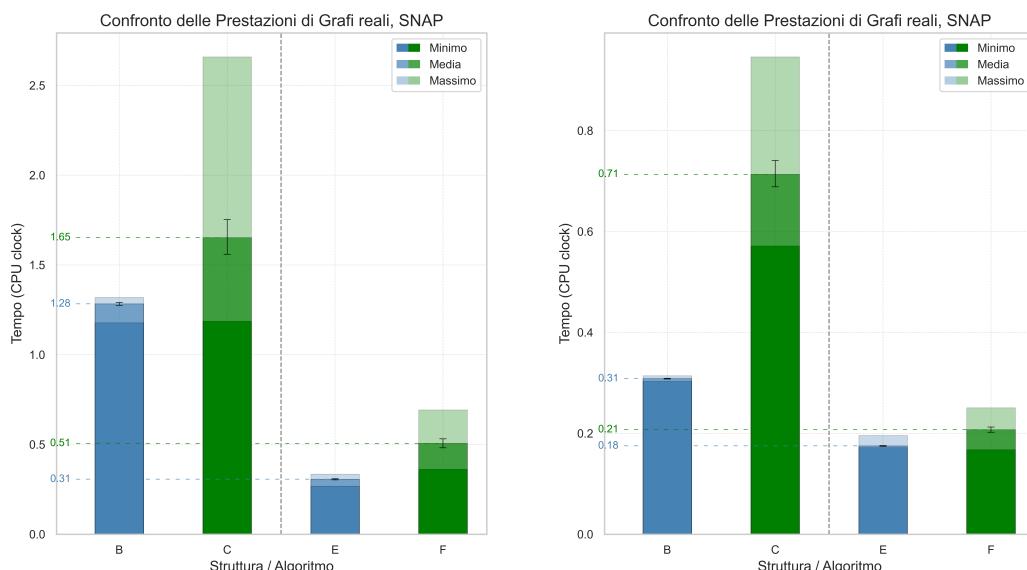


Figura 5.39 as-Skitter, Grafo della topologia di Internet, confronto single-thread (a sinistra) e multi-thread (a destra) con: $k = 0.2581$, $d = 25$, $n = 1.6 \times 10^6$, $e = 11 \times 10^6$.

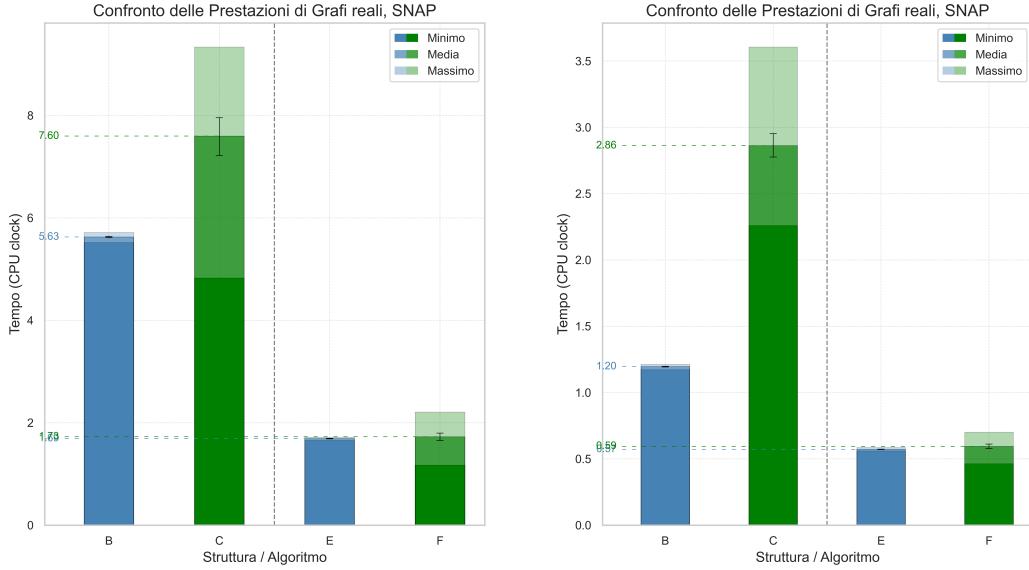


Figura 5.40 com-LiveJournal, Grafo di una comunità Internet, confronto single-thread e multi-thread con: $k = 0.2843$, $d = 17$, $n = 4 \times 10^6$, $e = 34 \times 10^6$.

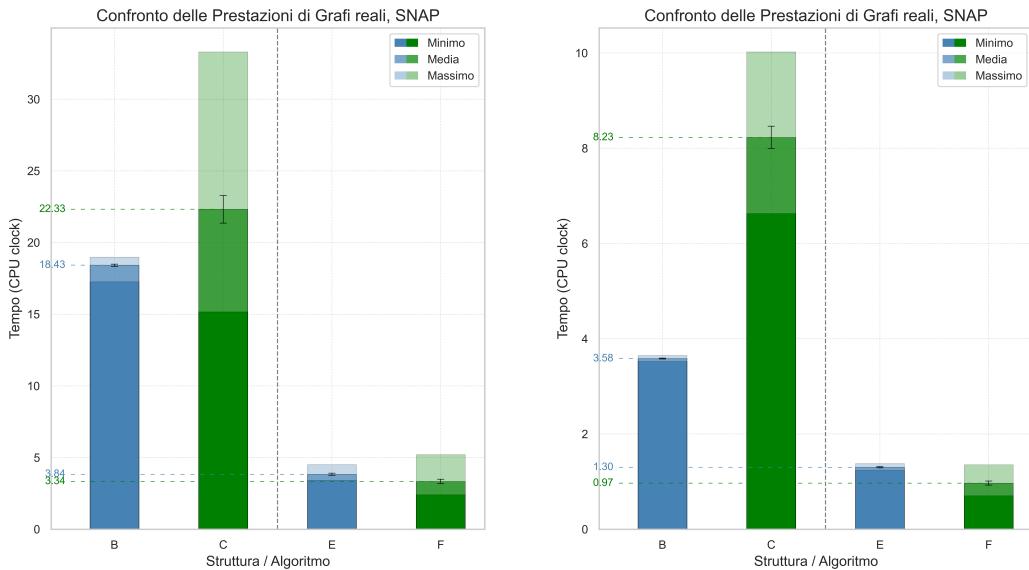


Figura 5.41 com-Orkut, Grafo di un social-network, confronto single-thread e multi-thread con: $k = 0.1666$, $d = 9$, $n = 3.07 \times 10^6$, $e = 117.18 \times 10^6$.

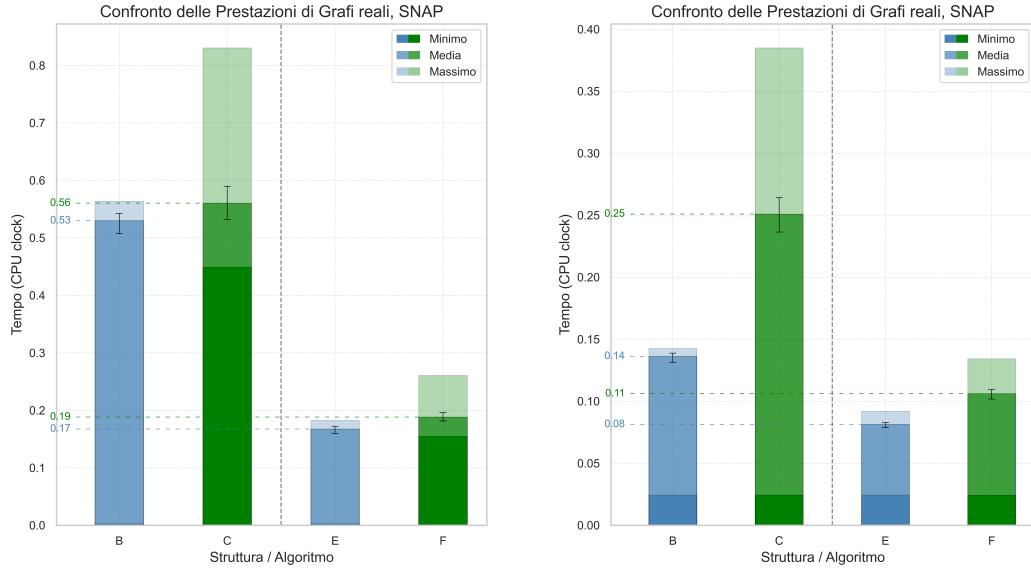


Figura 5.42 com-Youtube, Grafo di un social-network, confronto single-thread e multi-thread con: $k = 0.0808$, $d = 20$, $n = 1.1 \times 10^6$, $e = 29.8 \times 10^6$.

5.4.2 Grafi della raccolta Network Repository

I grafi della collezione Network Repository⁽²⁾ coprono un grande spazio di dimensioni (da poche migliaia a decine di milioni di vertici e archi) e di domini applicativi, tra cui reti sociali, web, biologia, reti stradali e neurali. Le caratteristiche strutturali variano ampiamente (small-world, scale-free, sparsi e densi), rendendo la collezione adatta a valutare in modo comparativo algoritmi di Visita in Ampiezza Ottimizzata per Direzione su scenari eterogenei.

Siano d il diametro del grafo, k il suo coefficiente di raggruppamento, n il numero di nodi e e il numero di archi, sui **Grafi di reti internet** abbiamo:

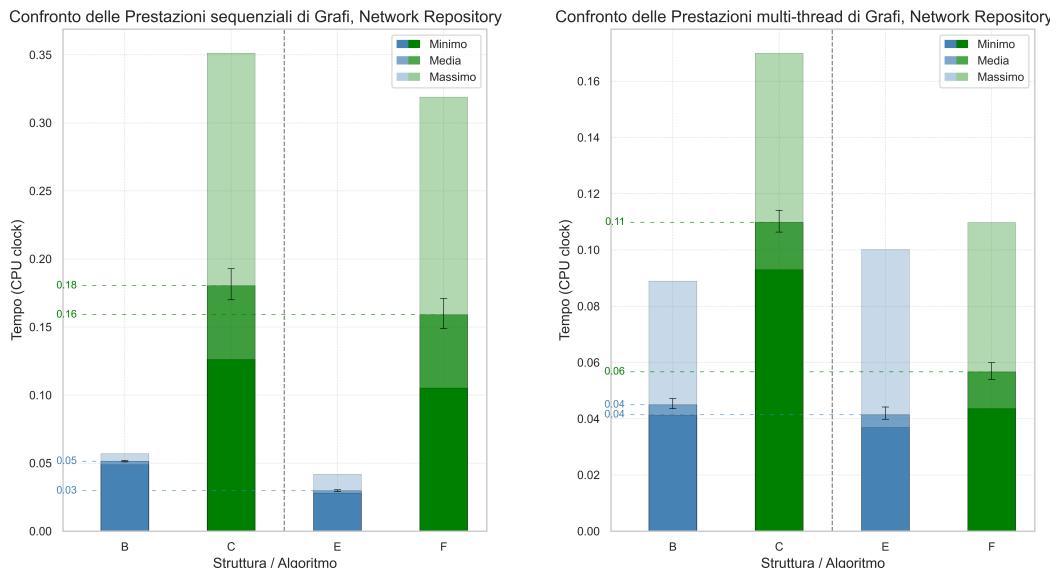


Figura 5.43 Grafo web arabian, confronto single-thread (a sinistra) e multi-thread (a destra) con: $k = 0.6531$, $d = 1 \times 10^3$, $n = 1.64 \times 10^5$, $e = 2 \times 10^6$.

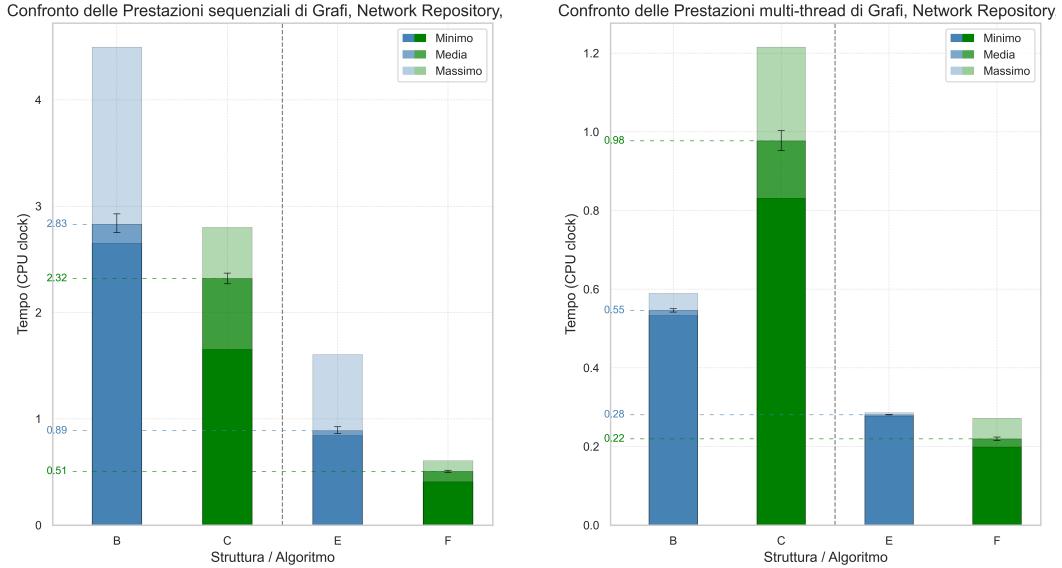


Figura 5.44 Grafo web uk, confronto single-thread e multi-thread con: $k = 0.99$, $d = 850$, $n = 1.3 \times 10^5$, $e = 12 \times 10^6$.

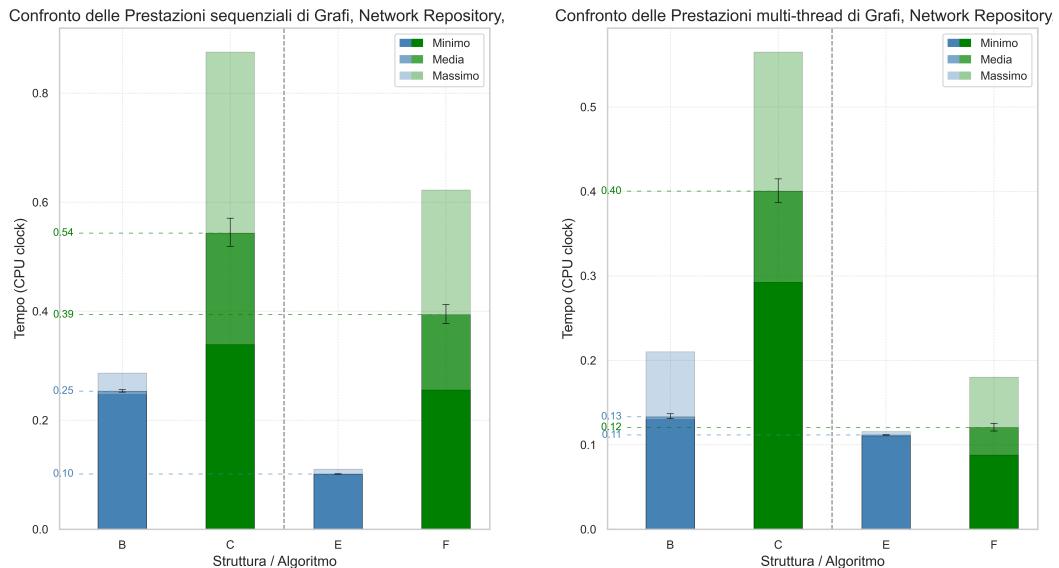


Figura 5.45 Grafo web google, confronto single-thread e multi-thread con: $k = 0.57$, $d = 6 \times 10^3$, $n = 9.16 \times 10^5$, $e = 5 \times 10^6$.

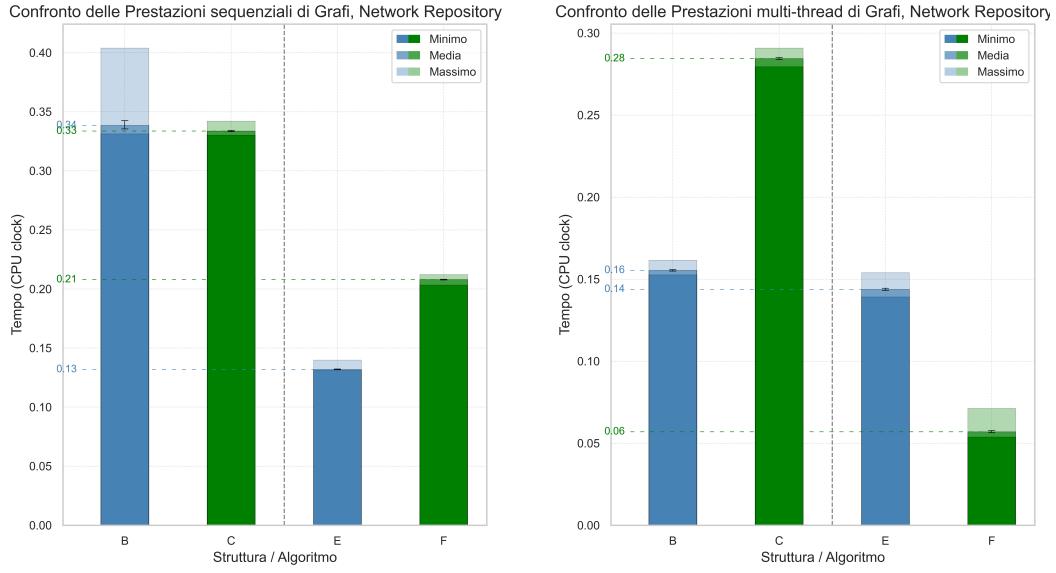


Figura 5.46 Grafo web Stanford, confronto single-thread e multi-thread con: $k = 0.59$, $d = 39 \times 10^3$, $n = 2.82 \times 10^5$, $e = 5 \times 10^6$.

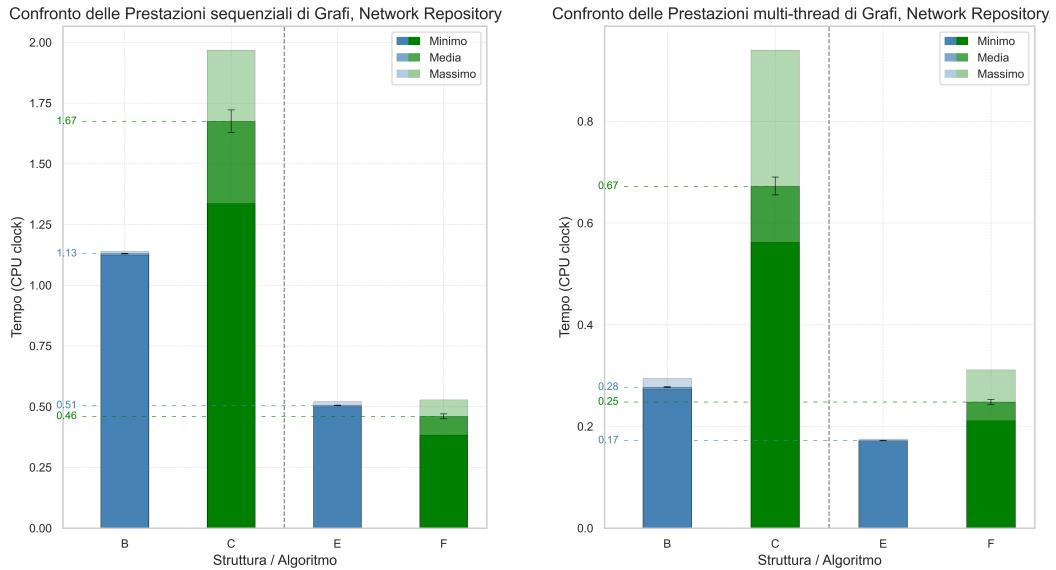


Figura 5.47 Grafo web it, confronto single-thread e multi-thread con: $k = 0.82$, $d = 469$, $n = 5.09 \times 10^5$, $e = 7 \times 10^6$.

Sperimentazioni su grafi di reti neurali

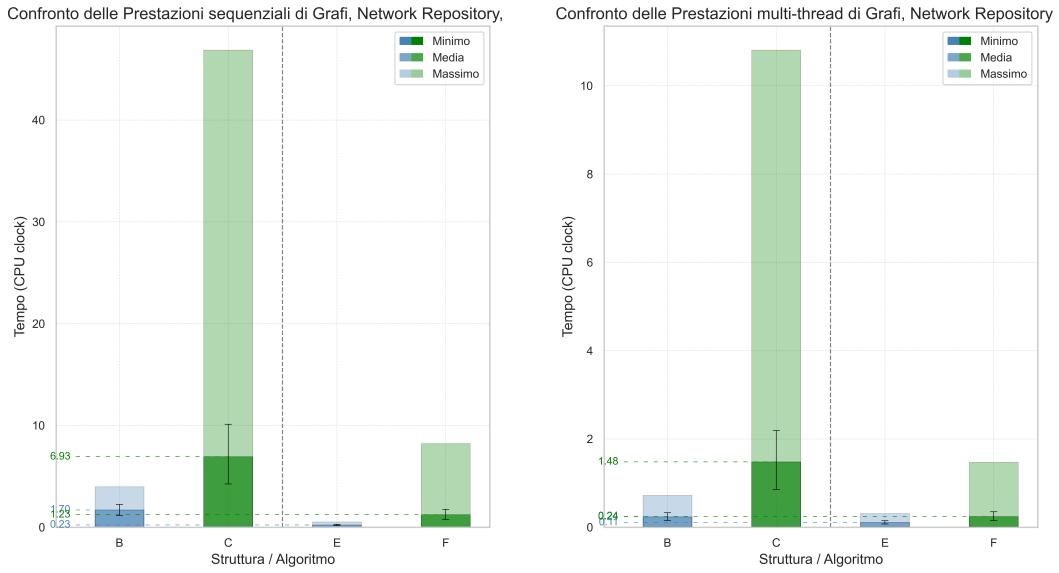


Figura 5.48 human-BNU-1-0025864-session-1-bg, confronto single-thread (a sinistra) e multi-thread (a destra) con: $k = 0.520314$, $d = 15.1 \times 10^3$, $n = 6.96 \times 10^5$, $e = 1.44 \times 10^6$.

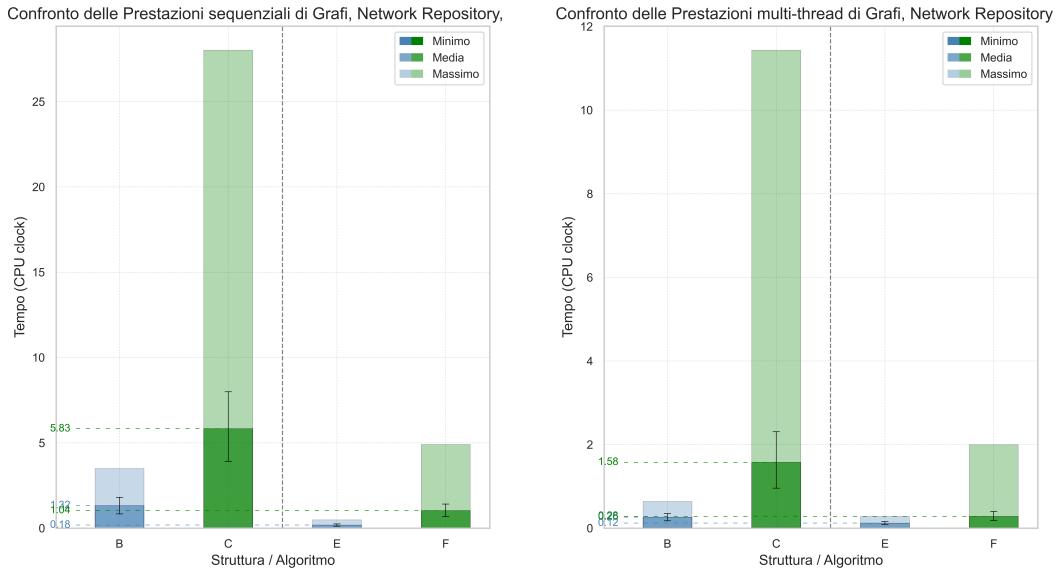


Figura 5.49 human-BNU-1-0025864-session2-bg, confronto single-thread e multi-thread con: $k = 0.4729$, $d = 8.4 \times 10^3$, $n = 6.94 \times 10^5$, $e = 1.33 \times 10^6$.

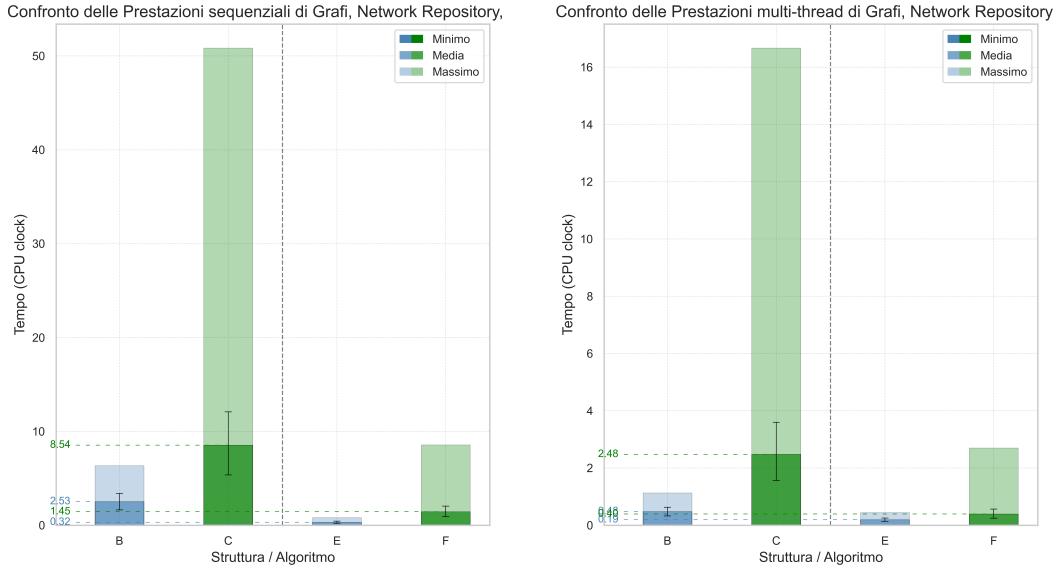


Figura 5.50 human-BNU-1-0025865-session-1-bg, confronto single-thread e multi-thread con: $k = 0.3429$, $d = 8.3 \times 10^3$, $n = 7.34 \times 10^5$, $e = 1.66 \times 10^6$.

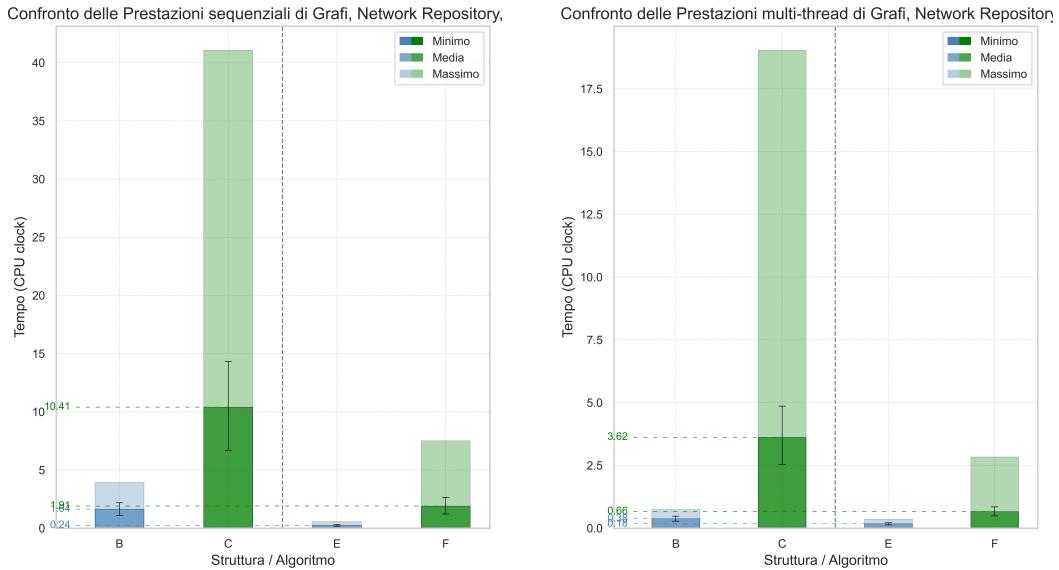


Figura 5.51 human-Jung2015-M87127667, confronto single-thread e multi-thread con: $k = 0.5729$, $d = 9.7 \times 10^3$, $n = 8.53 \times 10^5$, $e = 7.39 \times 10^5$.

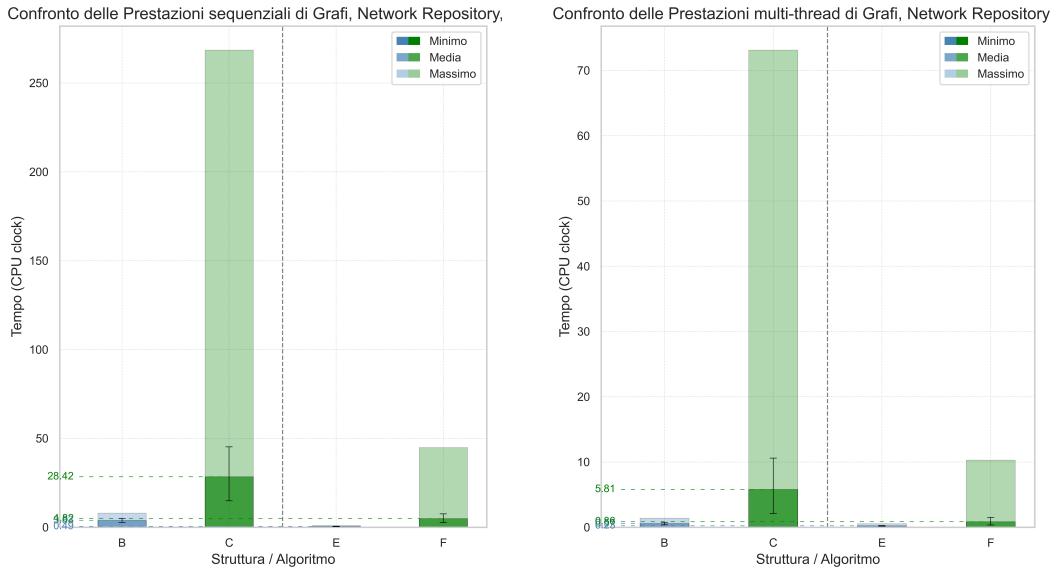


Figura 5.52 human-Jung2015-M87127677, confronto single-thread e multi-thread con: $k = 0.5017$, $d = 10.5 \times 10^3$, $n = 7.92 \times 10^5$, $e = 208.9 \times 10^6$.

Nei grafi neurali reali il grado medio è spesso molto basso (2–4), infatti la densità media dei grafi presi in considerazione è dell’ordine di 5×10^{-4} , e le frontiere della Visita in Ampiezza rimangono sottili lungo gran parte dell’esecuzione, implica che gli intervalli matrice compressa sono corti, quindi il vantaggio di località sequenziale è limitato.

In questo regime, il vantaggio tipico della Matrice di Adiacenza Compressa (la scansione contigua di lunghe porzioni scritte in memoria) si riduce drasticamente: per ogni vertice si leggono pochissimi vicini, per cui il costo fisso di accedere a `offset[v]` e di sostenere eventuali mancati riscontri in cache diventa proporzionalmente elevato. Inoltre, quando il grafo è così sparso, la penalità delle liste di adiacenza si attenuano, migliorandone le prestazioni.

Capitolo 6

Conclusioni

In questa relazione abbiamo analizzato il comportamento della Visita in Ampiezza (BFS) a confronto con la sua variante, la Visita in Ampiezza Ottimizzata per Direzione (DO-BFS), valutandone le prestazioni sia in ambito sequenziale sia multi-thread su grafi reali e sintetici.

L’analisi sperimentale mostra che la Visita Ottimizzata per Direzione tende a superare la Visita in Ampiezza classica su grafi di grandi dimensioni con basso diametro e distribuzioni di grado sbilanciate (scale-free), come quelli generati dal modello di Holme–Kim. In tali condizioni, la strategia bottom-up riduce sensibilmente il numero di adiacenze effettivamente controllate nelle fasi centrali dell’esplorazione, soprattutto quando la frontiera è ampia.

Inoltre, anche quando il lavoro (numero di adiacenze verificate) non diminuisce, si osserva comunque un miglioramento del tempo di esecuzione: nel passo *bottom-up* ogni controllo è più “leggero” di quello *top-down* (test di appartenenza su `bitset` in $O(1)$, poche scritture), e la scansione sequenziale di strutture allocate contiguamente sfrutta meglio la gerarchia di memoria, aumentando l’efficienza della cache quando la frontiera è ampia.

L’efficacia della scelta dinamica della direzione è ancora più marcata in ambiente multi-thread, dove la combinazione di frontiere compatte (`bitset`), partizionamento bilanciato e buona località dei dati consente di sfruttare meglio la gerarchia di memoria.

I risultati sui grafi reali delle collezioni SNAP e Network Repository sono coerenti con il quadro teorico: in media si osservano accelerazioni dell’ordine di 2×–4× rispetto alla Visita in Ampiezza classica, con punte maggiori nei nostri esperimenti fino a 9×–10× su reti grandi, sparse, a basso diametro e con presenza di centri nevralgici. In contesti multi-thread, i guadagni aumentano ulteriormente grazie alla riduzione del lavoro ridondante e alla migliore saturazione della banda di memoria. L’entità precisa del beneficio dipende tuttavia dalla topologia del grafo e dall’implementazione (rappresentazione dei dati, euristica e politiche di pianificazione).

Nei grafi molto sparsi, come quelli sulle reti neurali, i benefici della Visita in Ampiezza Ottimizzata per Direzione sono più contenuti ma non nulli: il bottom-up può comunque trarre vantaggio da un coefficiente di raggruppamento elevato, che aumenta la probabilità di arresto anticipato. In questi scenari la *Matrice di Adiacenza*

cenza Compressa risulta meno efficiente: il basso grado medio e l'irregolarità degli accessi riducono lo sfruttamento della località e la banda effettiva, mentre il costo fisso di setup/scansione degli intervalli (**offset**) pesa di più. Con gradi molto piccoli e distribuzioni spezzate, le *Liste di Adiacenza* possono quindi risultare competitive (o superiori), grazie a iterazioni puntuale senza attraversamenti di ampie regioni contigue poco utili.

Per i grafi sintetici la tendenza è coerente: su grafi regolari molto sparsi (grado medio inferiore a $\log n$) o su small-world con coefficiente di raggruppamento troppo basso e con l'assenza di centri nevralgici, la Visita in Ampiezza Ottimizzata per Direzione offre vantaggi limitati o nulli; al crescere della densità locale o della presenza di centri nevralgici, i benefici diventano via via più evidenti (con riduzioni del lavoro misurabili in termini di adiacenze esaminate).

Nel complesso, l'evidenza sperimentale conferma che la scelta Ottimizzata per Direzione rappresenta un'evoluzione efficace della Visita in Ampiezza tradizionale: riduce il lavoro nelle fasi critiche, scala bene in su di una versione multi-thread rispetto alla classica visita top-down, e beneficia di rappresentazioni dati contigue e compatte. L'impatto finale dipende dalla morfologia del grafo (diametro, coefficiente di raggruppamento, distribuzione dei gradi) e dalla qualità dell'implementazione (euristica di cambio direzione, gestione delle frontiere, politiche di bilanciamento, strutture dati utilizzate per i grafi).

Appendice A

Implementazione

L'algoritmo è stato implementato sia in versione *sequenziale* sia in versione *multi-thread*, utilizzando il linguaggio di programmazione a basso livello C. L'obiettivo principale di questa fase è analizzare l'impatto dell'ottimizzazione della direzione nei due contesti computazionali, valutandone l'efficacia con ciascuna delle strutture dati adottate per la rappresentazione dei grafi.

A.1 Strutture dati per la Visita in Ampiezza

La funzione principale della Visita in Ampiezza (Breadth-First Search, BFS) è condivisa tra la versione *classica* e quella con *ottimizzazione della direzione*, così come tra le implementazioni sequenziale e multi-thread. Le differenze principali riguardano l'accesso e la gestione delle strutture dati impiegate per rappresentare il grafo.

Per tale motivo, in questa sezione viene presentata un'unica versione della funzione, ovvero quella basata sull'utilizzo della rappresentazione tramite matrici di adiacenza compresse (CSR), in quanto ritenuta sufficientemente rappresentativa della logica generale adottata.

Implementazione: Visita in Ampiezza con Matrici di Adiacenza Comprese.

```

1 int* aBFSHybridSearch(Graph *graph, int source, bool hybrid) {
2     if(graph == NULL || graph->numVertices == 0 || graph->all_out_edges == 
3         NULL)
4         exit(1);
5     //inizializzo le strutture dati
6     Parents *parents = newParents(graph->numVertices);
7     initParents(parents, graph->numVertices, graph->numEdges, source);
8     Frontier *frontier = newFrontier(graph->numVertices);
9     addToFrontier(frontier, source);
10
11    if(hybrid) {      //0 = bfs classica, 1 = bfs ibrida
12        bool current_direction = 0; //0 = top-down-step, 1 = bottom-up-step
13        HeuristicData heuristic;
14        initHeuristic(&heuristic);
15
16        while ( !FrontierIsEmpty(frontier) ) { //finche' frontiera non vuota
17            if( current_direction = next_direction(&heuristic,
18                current_direction, graph->numVertices) == 0 )
19                top_down_stepA(graph, frontier, parents);
20            else
21                top_up_stepA(graph, frontier, parents);
22        }
23    }
24
25    return parents;
26}

```

```

19         bottom_up_stepA(graph, frontier, parents);
20         updateSizesA(graph, frontier, parents, &heuristic); //aggiorna
21         dati euristica
22     }
23     } else {
24         while ( !FrontierIsEmpty(frontier))
25             classic_top_down_stepA(graph, frontier, parents);
26     }
27     freeAll(frontier); //dealloca la frontiera
28     return parents->array;
}

```

La struttura **Graph** (la quale usa Matrici di Adiacenza Comprese) è definita in **header.h**, insieme alle strutture **ListGraph** e **MatrixGraph**:

```

1 struct Graph {
2     int numVertices;      //numero dei nodi del grafo
3     long numEdges;        //numero degli archi del grafo
4     int *out_degree, *all_out_edges, *offset;
5     bool directed;        //indica se il grafo e' orientato
6 };
7 typedef struct Graph Graph;
8
9 struct ListGraph {
10    int numVertices;
11    long numEdges;
12    LinkedList ** all_out_edges;    //array di puntatori a liste collegate
13    bool directed;
14 };
15 typedef struct ListGraph ListGraph;
16
17 struct MatrixGraph {
18    int numVertices;
19    long numEdges;
20    bool *adj_matrix; //unico array per continuita' su memoria, non uso
21    //array di puntatori ad array
22    int *out_degree;   //contiene il grado di ogni nodo
23    bool directed;
24 };
25 typedef struct MatrixGraph MatrixGraph;

```

Le strutture dati ausiliarie utilizzate nell'implementazione dell'algoritmo sono le seguenti:

- *Parents*: si tratta di una **struct** che include un array contenente i nodi predecessori di ciascun nodo e un campo denominato **total_edges_degree**, che rappresenta la somma dei gradi dei nodi non ancora visitati. Quest'ultimo parametro si rivela particolarmente utile per ridurre il dispendio computazionale durante la selezione della direzione ottimale della visita.
- *Frontier*: rappresenta la frontiera dell'esplorazione BFS. È implementata come una coda basata su array, gestita mediante due indici, **front** e **rear**. La stessa struttura viene riutilizzata per rappresentare sia la frontiera corrente sia quella successiva (indicata come **next**), contribuendo a una gestione più efficiente e compatta del codice.

- *HeuristicData*: **struct** che aggrega un insieme di parametri ausiliari utilizzati per il calcolo delle euristiche, necessari all’ottimizzazione della strategia di esplorazione.

```

1 struct Frontier {
2     struct ArrQueue* queue;
3     struct Bitset* set;      //utilizzato per ottimizzazione
4 };
5 typedef struct Frontier Frontier;
6
7 struct Parents {
8     int *array;
9     Bitset64 *unvisited;
10    long total_edges_degree;
11 };
12 typedef struct Parents Parents;
13
14 struct Heuristic {
15     int frontier_size;       //grandezza della frontiera corrente e
16     int frontier_prev_size;   //dell’iterazione precedente
17     int edges_frontier_size; //out-degree dei nodi in frontier
18     int edges_parents_size;  //out-degree dei nodi non ancora esplorati
19 };
20 typedef struct Heuristic HeuristicData;

```

A.2 Visita in Ampiezza sequenziale

L’implementazione sequenziale dell’algoritmo prevede due varianti principali della Visita in Ampiezza (Breadth-First Search, BFS):

- la Visita in Ampiezza *classica*, con il solo approccio top-down;
- la Visita in Ampiezza con *ottimizzazione della direzione* (Direction-Optimized BFS, DO-BFS).

Analogamente alla funzione principale, ciascuna di queste varianti è stata implementata per ogni tipo di struttura dati utilizzata nella rappresentazione del grafo. Le differenze tra le implementazioni, tuttavia, risultano marginali per quanto riguarda il codice, e riguardano principalmente le modalità di accesso e gestione dei dati del grafo.

Per tale motivo, in questa sezione verrà presentata un’unica versione per ciascuna delle due varianti, utilizzando come riferimento la rappresentazione basata su Matrice di Adiacenza Compressa, considerata sufficientemente espressiva e rappresentativa.

Implementazione: Top-down con Matrici di Adiacenza Comprese.

```

1 void top_down_stepA(Graph *graph, Frontier* frontier, Parents* parents) {
2     initFrontierBits(frontier);
3     int curr_node, curr_node_degree, out_edge;
4     size_t i;
5     int curr_frontier_size = getFrontierLen(frontier);
6
7     for(i = 0; i < curr_frontier_size; i++) {
8         curr_node = getFromFrontier(frontier); // == nodo corrente
9         curr_node_degree = graph->out_degree[curr_node];

```

```

10
11     for(int edge = 0; edge < curr_node_degree; edge++) {    // se il nodo
12         ha almeno un arco, e per ogni arco del nodo corrente
13         out_edge = graph->all_out_edges[graph->offset[curr_node] + edge
14             ]; //nodo raggiunto dal nodo corrente dall'arco edge
15         if(getParents(parents, out_edge) == -1) { //se non gia'
16             visitato, allora
17             setParents(parents, out_edge, curr_node, graph->out_degree[
18                 out_edge]); //lo visito
19             addToFrontier(frontier, out_edge); //lo inserisco in
16                 frontier
17             }
18         }
19     }

```

Si sottolinea che l'implementazione sequenziale è stata progettata con l'obiettivo di essere facilmente estendibile ad un contesto multi-thread. In particolare, l'algoritmo Ottimizzato per Direzione è stato modellato sin dall'inizio per supportare un'efficiente parallelizzazione tra i vari thread, mentre una versione classica avrebbe richiesto modifiche sostanziali alla logica di esecuzione per ottenere prestazioni comparabili in ambiente concorrente.

Per la versione con *Ottimizzazione della Direzione*, la componente centrale è rappresentata dalla funzione *euristica next_direction*, la cui logica si ispira direttamente all'approccio descritto da Beamer et al.⁽⁴⁾. Questa funzione è responsabile della selezione dinamica della direzione più conveniente per l'esplorazione (top-down o bottom-up), sulla base dello stato corrente del grafo e della frontiera.

Implementazione: Euristica.

```

1   bool next_direction( HeuristicData *h, bool current_direction, int
2       total_nodes) {
3       int param_A = 14; //parametri empirici suggeriti nel paper
4       int param_B = 24;
5
6       if (current_direction == 0) { //se mi trovo in top-down-step
7           //se mf > mu/param_A e frontiera crescente
8           if(h->edges_frontier_size > h->edges_parents_size/param_A
9               &&
10                  h->frontier_size > h->frontier_prev_size)
11                  return 1; //vado in bottom-up
12              else
13                  return 0; //rimango in top-down
14          }
15          else { //se mi trovo in bottom-up-step
16              //se nf < n/param_B e frontiera decrescente
17              if(h->frontier_size < total_nodes/param_B
18                  &&
19                     h->frontier_size <= h->frontier_prev_size)
20                     return 0; //vado in top-down
21                 else
22                     return 1; //rimango in bottom-up
23      }

```

Per quanto riguarda l'implementazione dell'algoritmo di *Visita in Ampiezza ottimizzata per Direzione*, si fa uso di una funzione *top-down*, identica alla precedente (A.2), e di una nuova funzione di *bottom-up*:

Implementazione: Bottom-up con Matrici di Adiacenza Compresse.

```

1 void bottom_up_stepA(Graph *graph, Frontier *frontier, Parents* parents) {
2     int curr_node, curr_node_degree, out_edge;
3     int numUnvisitedBlock = parents->unvisited->block_size;
4     int curr_frontier_size = getFrontierLen(frontier);
5
6     //creo nuovo bitset, così da usare il vecchio per le verifiche, ed il
7     //nuovo per gli insert
8     Bitset *frontier_bitset = frontier->set;      //bitset precedente
9     frontier->set = newBitSet(frontier_bitset->size); //nuovo bitset
10
11    // per ogni parola della bitmap "unvisited"
12    for (int w = 0; w < numUnvisitedBlock; ++w) {
13        uint64_t block = parents->unvisited->bits[w];      // copia locale per
14        // estrarre i bit a 1
15        while (block) {
16            unsigned b = ctz64(block);           // posizione del prossimo bit a
17            1 in [0..63]
18            size_t curr_node = (w << 6) + b;      // indice globale del
19            // vertice
20            if (curr_node >= ngraph->numVertices) break;          // per
21            // l'ultima parola parziale
22
23            if (getParents(parents, curr_node) == -1) {
24                curr_node_degree = graph->out_degree[curr_node];
25                for (int edge = 0; edge < curr_node_degree; edge++) {
26                    out_edge = graph->all_out_edges[graph->offset[curr_node] +
27                        edge];
28                    if (is_set(frontier_bitset, out_edge)) { //se presente in
29                        frontier
30                        setParents(parents, curr_node, out_edge, curr_node_degree
31                        );
32                        addToFrontier(frontier, curr_node);
33                        break;          //basta un genitore, passo al prossimo nodo
34                    }
35                }
36            }
37            block &= (block - 1); //imposto a 0 il bit appena elaborato, passa al
38            //prossimo 1 dello stesso blocco
39        }
40        frontier->queue->front += curr_frontier_size;      //resetto la frontiera
41        deallocBitset(frontier_bitset);          //dealloco il precedente bitset
42    }
43 }
```

Per un'implementazione efficiente della fase *bottom-up*, è stata introdotta una struttura dati a basso livello, il **BitSet**, che rappresenta la frontiera come maschera di bit indicizzata sui nodi. In questo modo il test di appartenenza alla frontiera avviene in tempo $O(1)$ tramite operazioni *bitwise*, evitando di scansionare l'intera frontiera ad ogni verifica e preservando i benefici della strategia *bottom-up*.

Inoltre si tratta di una struttura che non appesantisce ulteriormente l'algoritmo, un **BitSet** può essere considerato come un array di valori booleani, con la differenza cruciale che ciascun elemento occupa esattamente *un bit* di memoria, anziché un intero byte come avviene per i tipi *bool* standard in C. Questa caratteristica rende il **BitSet** in grado di rappresentare informazioni binarie con un elevato grado di efficienza spaziale, la compattezza, oltretutto, permette di aumentare la località dei dati, facilitando l'allocazione dell'intera struttura nella cache della CPU.

A.3 Visita in Ampiezza multi-thread

La versione multi-thread dell'algoritmo è stata realizzata mediante l'utilizzo di *OpenMP*, una libreria standard che consente di distribuire il carico computazionale su più thread esecutivi⁽¹⁰⁾, con particolare attenzione alla gestione delle contese e strutture dati.

A.3.1 Libreria OpenMP: definizione e funzioni

OpenMP (*Open Multi-Processing*) è un'API progettata per facilitare la programmazione parallela su sistemi a memoria condivisa, tipicamente CPU multicore. Essa consente di esprimere il parallelismo tramite direttive `#pragma` inserite all'interno del codice sorgente in linguaggi quali C, C++ o Fortran, mantenendo elevata la leggibilità e la compatibilità con versioni sequenziali⁽¹⁰⁾.

Una delle direttive più diffuse è `#pragma omp parallel for`, che consente di parallelizzare in modo semplice ed efficiente i cicli `for`. Ad esempio:

```

1 #pragma omp parallel for
2 for (int i = 0; i < N; i++) {
3     a[i] = b[i] + c[i];
4 }
```

In questo modo, il lavoro di iterazione viene suddiviso tra più thread. Il compilatore e il runtime OpenMP si occupano di assegnare automaticamente le iterazioni ai vari thread.

Mentre, con la clausola `private` indica che ogni thread deve possedere una propria istanza locale di una variabile. Tale meccanismo è fondamentale per prevenire condizioni di race, poiché evita che più thread accedano e modifichino simultaneamente la medesima variabile condivisa.

```

1 int sum = 0;
2 #pragma omp parallel for private(sum)
3 for (int i = 0; i < N; i++) {
4     sum = a[i] + b[i];
5     c[i] = sum;
6 }
```

Nell'esempio riportato, ogni thread dispone di una copia privata della variabile `sum`, garantendo l'assenza di conflitti e condizioni di race durante l'esecuzione parallela.

Quando più thread devono aggiornare una variabile condivisa, possono insorgere condizioni di race che compromettono la correttezza del calcolo. La clausola `reduction` permette di effettuare in modo sicuro operazioni di riduzione (ad esempio somma, massimo, minimo), garantendo l'integrità dei dati e la correttezza del risultato finale.

```

1 int sum = 0;
2 #pragma omp parallel for reduction(+:sum)
```

```

3   for (int i = 0; i < N; i++) {
4     sum += a[i];
5 }
```

Ogni thread mantiene una copia locale della variabile `sum`, le quali vengono infine aggregate in modo corretto per ottenere il risultato finale.

Inoltre, OpenMP consente di specificare la strategia di assegnazione del lavoro attraverso la clausola `schedule`. Le modalità più comuni sono:

- *Statico* (`schedule(static, chunk)`): divide il ciclo in blocchi di dimensione fissa (`chunk`) e li assegna in modo equo ai thread. È efficiente quando ogni iterazione ha un costo simile.
- *Dinamico* (`schedule(dynamic, chunk)`): assegna dinamicamente blocchi di iterazioni ai thread man mano che terminano il lavoro. È utile in presenza di carico irregolare (es. iterazioni con tempi di esecuzione variabili).

Esempio:

```

1 #pragma omp parallel for schedule(dynamic, 4)
2 for (int i = 0; i < N; i++) {
3   process(i);
4 }
```

Altre funzioni utili, integrate in OpenMP e definite in `omp.h` sono:

- `omp_get_num_threads()`: restituisce il numero di thread attivi.
- `omp_get_thread_num()`: restituisce l'identificativo del thread corrente.
- `omp_get_max_threads()`: restituisce il numero massimo di thread disponibili.

Possiamo affermare, quindi, che OpenMP rappresenta una soluzione potente e accessibile per migliorare le prestazioni di applicazioni computazionalmente intensive, mantenendo il codice relativamente semplice e comprensibile.

Il codice del bottom-up in parallelo ne è un buon esempio:

Implementazione: Bottom-up con Matrici di Adiacenza Compresse multi-thread.

```

1 void bottom_up_stepA(Graph *graph, Frontier *frontier, Parents* parents) {
2   int curr_node, curr_node_degree, out_edge, unv_index = 0;
3   int numUnvisitedBlock = parents->unvisited->block_size;
4   int curr_frontier_size = getFrontierLen(frontier);
5
6   #pragma omp parallel for schedule(dynamic, NUM_CHUNK) private(curr_node,
7                                     curr_node_degree, out_edge)
8   for (int w = 0; w < numUnvisitedBlock; ++w) {
9     uint64_t block = parents->unvisited->bits[w];      // copia locale per
10    estrarre i bit a 1
11    while (block) {
12      unsigned b = ctz64(block);                      // posizione del prossimo bit a
13      1 in [0..63]
```

```

11         size_t curr_node = (w << 6) + b;           // indice globale del
12             vertice
13         if (curr_node >= graph->numVertices) break;          // per l
14             'ultima parola parziale
15
16         int tid = omp_get_thread_num();
17
18         curr_node_degree = graph->out_degree[curr_node];
19         for(int edge = 0; edge < curr_node_degree; edge++) {
20             out_edge = graph->all_out_edges[ graph->offset[curr_node] +
21                 edge];
22             if( is_set(frontier->set, out_edge) ) {
23                 setParents(parents, curr_node, out_edge, graph->
24                     out_degree[curr_node]);
25                 chunkEnqueue(local_frontier[tid], curr_node);
26                 break;
27             }
28         }
29     } //inizializza la frontiera, inserendo poi in essa tutti gli elementi in
29     local_frontier
30     updateFrontier(frontier);
31 }
```

Mentre il codice della funzione principale della Visita in Ampiezza rimane invariato (A.1), la principale differenza rispetto alla versione sequenziale del bottom-up risiede nell'utilizzo di una struttura dati *dedicata* per ogni thread: la `local_frontier`.

Questa è implementata come un array di puntatori a *chunk*, di dimensione pari a `p`, dove `p` corrisponde al numero massimo di thread disponibili sulla macchina utilizzata. Tale struttura consente a ciascun thread di disporre di uno spazio di memoria esclusivo per la scrittura, eliminando così la necessità di sincronizzazione o di gestione concorrente durante l'accesso a risorse condivise. Inoltre utilizza tecniche di padding in modo da evitare il false sharing tra diversi thread.

Al contrario, nella versione multi-thread della fase *top-down*, a causa della sua struttura algoritmica, è necessaria una gestione concorrente delle risorse, rendendo inevitabile la sincronizzazione tra thread, anche se è possibile evitare una condizione di race tramite l'uso di una operazione atomica (compare-and-swap), a differenza di quanto avviene nella fase *bottom-up*.

Implementazione: Top-down con Matrici di Adiacenza Compresse multi-thread.

```

1 void top_down_stepA(Graph *graph, Frontier* frontier, Parents* parents) {
2     if(local_frontier == NULL) {
3         local_frontier = initLocalFrontier(graph->numVertices,
4                                         LOCAL_FRONTIER_ALLLOC_REDUCER);
5     }
6     int curr_node, curr_node_degree, out_edge;
7     size_t i;
8     int curr_frontier_size = getFrontierLen(frontier);
9     int edges_to_remove = 0;
10
11 #pragma omp parallel for schedule(dynamic, NUM_CHUNK) private(curr_node,
12     curr_node_degree, out_edge) reduction(+:edges_to_remove)
13 for(i = 0; i < curr_frontier_size; i++) {
14     int tid = omp_get_thread_num();
15     curr_node = frontier->queue->value[i];
```

```

14     curr_node_degree = graph->out_degree[curr_node];
15
16     for (int edge = 0; edge < curr_node_degree; edge++) { // se il nodo
17         ha almeno un arco, e per ogni arco del nodo corrente
18         out_edge = graph->all_out_edges[graph->offset[curr_node] + edge];
19         if (_sync_bool_compare_and_swap(&parents->array[out_edge],
20                                         -1, curr_node)) { //se il nodo non e' gia' stato
21             esplorato (-1) ritorno true e lo modifco (curr_node)
22             edges_to_remove += graph->out_degree[out_edge]; //sommo i
23             gradi dei nodi esplorati
24             chunkEnqueue(local_frontier[tid], out_edge);
25             clear_bit64(parents->unvisited, out_edge); //rimuovo dal
26             bitset dei non visitati il nodo
27         }
28     }
29 }
30 parents->total_edges_degree == edges_to_remove; //rimuovo la
31 somma dei gradi dei nodi esplorati dal totale
32 updateFrontier(frontier); //inizializza la frontiera, inserendo
33 poi in essa tutti gli elementi in local_frontier
34 }
```

Poiché l'algoritmo inizia sempre con una visita *top-down*, la *local frontier* viene inizializzata in questa fase, utilizzando un parametro denominato **LOCAL_FRONTIER_ALLOC_REDUCER**, che definisce il numero di chunk per thread.

Per quanto riguarda la sincronizzazione durante la scrittura concorrente sull'array dei padri, è stata preferita una funzione atomica rispetto all'utilizzo di un mutex. Le funzioni atomiche sono particolarmente affidabili se impiegate per operazioni semplici e operano a livello hardware, garantendo prestazioni significativamente superiori rispetto ai meccanismi di locking a livello software come i mutex. In particolare, l'operazione atomica blocca la risorsa a livello hardware al momento della verifica della condizione di input, ovvero il confronto tra il valore corrente e un valore atteso, restituendo **true** solo se l'aggiornamento è stato effettuato con successo da questa funzione.

Tuttavia, anche questa soluzione introduce un certo overhead computazionale; di conseguenza, la fase *bottom-up* risulta maggiormente adatta a un'efficiente parallelizzazione.

A.4 Verifica della correttezza dell'algoritmo

Un aspetto cruciale nello sviluppo di un algoritmo è la verifica della sua correttezza. Nonostante l'implementazione qui presentata sia basata su un algoritmo consolidato, ovvero la *Direction-Optimizing BFS* proposta da Beamer et al.⁽⁴⁾, si tratta comunque di una realizzazione originale.

A tal fine, la correttezza dell'algoritmo sarà valutata mediante il confronto con il classico algoritmo di Visita in Ampiezza, *Top-Down*. Durante l'esecuzione, i dati relativi alla profondità di esplorazione di ciascun nodo del grafo saranno memorizzati in un array dedicato.

Un ulteriore metodo di verifica consiste nell'analisi delle proprietà fondamentali dei grafi esplorati, al fine di garantire la coerenza e la correttezza dell'algoritmo.

- Ogni nodo raggiungibile dalla sorgente deve essere correttamente contrassegnato come visitato;

- Le distanze calcolate devono rispettare la proprietà caratteristica del grafo, ovvero:

$$dist(v) = dist(u) + 1 \quad \forall (u \rightarrow v) \text{ esplorato durante la BFS};$$

- La Visita in Ampiezza deve esplorare i livelli del grafo in ordine crescente di distanza, senza saltare nodi o livelli più prossimi a favore di nodi più lontani.

A.4.1 Veridicità dell'implementazione: Confronto

Il metodo di verifica infine adottato consiste in una leggera modifica all'implementazione dell'algoritmo, che consente di memorizzare, per ciascun nodo, la profondità (ovvero la distanza) dalla sorgente. Tale informazione viene salvata utilizzando una struttura dati semplice e lineare, generalmente un array indicizzato per nodo.

```

1 struct VerticesDepth {
2     int depth;
3     int *array_depth;
4 };
5 typedef struct VerticesDepth VerticesDepth;
```

Si tratta di un approccio valido ai fini della verifica, a differenza dell'array dei padri, il quale non risulta adatto per un confronto diretto, in quanto:

- L'Array dei padri è non deterministico, l'ordine di esplorazione dei nodi appartenenti allo stesso livello può variare in funzione dell'ordine in cui i nodi sono organizzati nella struttura dati, oppure, nel caso di esecuzione parallela, in base al completamento asincrono dei task da parte dei singoli thread;
- Al contrario, utilizzando la stessa strategia di visita (BFS), la distanza tra il nodo sorgente e ciascun nodo esplorato deve necessariamente rimanere invariata, costituendo una proprietà intrinseca del grafo. Pertanto, l'array delle distanze risulta un criterio affidabile per la verifica della correttezza dell'implementazione.

A tal fine, l'implementazione utilizza una variabile `depth` per tenere traccia della profondità corrente dell'esplorazione, inizializzata a 0 per il nodo sorgente e incrementata a ogni iterazione, sia nel caso della strategia *Top-Down* che in quella *Bottom-Up*. Tale valore viene assegnato a ciascun nodo al momento della sua prima visita e memorizzato all'interno di un array delle profondità, denominato `array_depth`, di dimensione n , dove n rappresenta il numero totale di nodi del grafo.

Al termine dell'esecuzione, viene stampato unicamente l'array delle profondità risultante dalla Visita in Ampiezza per ciascuna variante dell'algoritmo. I risultati vengono inoltre salvati su file di testo, mediante un semplice comando di input del tipo:

```
./aBFS ../grafici/soc-LiveJournal1.txt 0 2 1 > output_A_depths.txt
```

Infine si confrontano le stringhe dei file di output, risultando tutte identiche se corretto.

A.5 Generazione dei modelli noti di Grafi

Di seguito sono riportate le funzioni principali per la generazione dei grafi secondo i modelli noti:

Implementazione: Generatore grafi del modello Newman–Watts.

```

1 void generateWattsUndirectedGraphs(char *location, int nodes) {
2     int edge_count = 0;           // contatore degli archi
3     double p = PROBABILITY_WATTS; // probabilita' p
4     int N = nodes;              // numero totale di nodi
5     int k = K_EDGESPERNODES_WATTS; // numero di archi per nodo k
6
7     if (k % 2 != 0 || k < 2 || k >= N) {
8         perror("Errore: richiedi k pari, k >= 2 e k < N.\n");
9         exit(1);
10    }
11
12    FILE *filePointer = NULL;
13    EdgeHash *new_hash = malloc(sizeof(EdgeHash)); // tabella hash usata
14    // per il test sull'esistenza di un arco
15    hash_init(new_hash, compute_hash_capacity(N, 0, k, p+0.1)); // compute_hash_capacity calcola il valore atteso del numero di
16    // archi totale
17
18    filePointer = fopen(location, "w");
19    if (!filePointer) { //se il file non e' valido fermo il programma
20        perror("fopen\n");
21        freeHash(new_hash);
22        exit(1);
23    }
24
25    //prima aggiungo solo legami clockwise , senso orario successivo (
26    //quindi legami successivi al nodo (k/2), essendo un grafo non
27    //orientato sto aggiungendo anche i legami precedenti per ogni nodo
28    //)
29    printf("Generando grafo NW...\n\n");
30    for(int i = 0; i < N; i++) {
31        for(int d = 1; d <= k/2; d++) {
32            int j = (i + d) % N;
33            uint64_t key = encode_edge(i, j); //genero chiave hash
34            hash_insert(new_hash, key); //aggiungo la chiave
35            edge_count+=2;
36            fprintf(filePointer, "%d %d\n", i, j); //scrivo nel file l'
37            //arco i,j
38        }
39    }
40
41    const long S = llround(p * (double)N * (double)(k/2)); //calcolo
42    //scorciatoie attese
43    const long MAX_ATTEMPS = S ? (50*S + 1000) : 0; //creo un limite
44    //ai tentativi per alleggerire l'esecuzione
45    int added = 0, attempts = 0;
46    srand(time(NULL)); //inizializzo la funzione per i valori
47    //casuali
48
49    //aggiungo scorciatoie casuali
50    while(added < S) {
51        if(attempts++ > MAX_ATTEMPS) {
52            perror("Raggiungto limite aggiunta scorciatoie!!!\n");
53            break;
54        }
55        int u = random() % N;
56
57        for(int d = 1; d <= k/2; d++) {
58            int v = (u + d) % N;
59            uint64_t key = encode_edge(u, v);
60            hash_insert(new_hash, key);
61            edge_count+=2;
62            fprintf(filePointer, "%d %d\n", u, v);
63        }
64    }
65
66    //scrivo l'hash nel file
67    for(int i = 0; i < N; i++) {
68        for(int j = i + 1; j < N; j++) {
69            uint64_t key = encode_edge(i, j);
70            hash_insert(new_hash, key);
71        }
72    }
73
74    //scrivo l'hash nel file
75    for(int i = 0; i < N; i++) {
76        for(int j = i + 1; j < N; j++) {
77            uint64_t key = encode_edge(j, i);
78            hash_insert(new_hash, key);
79        }
80    }
81
82    //scrivo l'hash nel file
83    for(int i = 0; i < N; i++) {
84        for(int j = i + 1; j < N; j++) {
85            uint64_t key = encode_edge(i, j);
86            hash_insert(new_hash, key);
87        }
88    }
89
90    //scrivo l'hash nel file
91    for(int i = 0; i < N; i++) {
92        for(int j = i + 1; j < N; j++) {
93            uint64_t key = encode_edge(j, i);
94            hash_insert(new_hash, key);
95        }
96    }
97
98    //scrivo l'hash nel file
99    for(int i = 0; i < N; i++) {
100        for(int j = i + 1; j < N; j++) {
101            uint64_t key = encode_edge(i, j);
102            hash_insert(new_hash, key);
103        }
104    }
105
106    //scrivo l'hash nel file
107    for(int i = 0; i < N; i++) {
108        for(int j = i + 1; j < N; j++) {
109            uint64_t key = encode_edge(j, i);
110            hash_insert(new_hash, key);
111        }
112    }
113
114    //scrivo l'hash nel file
115    for(int i = 0; i < N; i++) {
116        for(int j = i + 1; j < N; j++) {
117            uint64_t key = encode_edge(i, j);
118            hash_insert(new_hash, key);
119        }
120    }
121
122    //scrivo l'hash nel file
123    for(int i = 0; i < N; i++) {
124        for(int j = i + 1; j < N; j++) {
125            uint64_t key = encode_edge(j, i);
126            hash_insert(new_hash, key);
127        }
128    }
129
130    //scrivo l'hash nel file
131    for(int i = 0; i < N; i++) {
132        for(int j = i + 1; j < N; j++) {
133            uint64_t key = encode_edge(i, j);
134            hash_insert(new_hash, key);
135        }
136    }
137
138    //scrivo l'hash nel file
139    for(int i = 0; i < N; i++) {
140        for(int j = i + 1; j < N; j++) {
141            uint64_t key = encode_edge(j, i);
142            hash_insert(new_hash, key);
143        }
144    }
145
146    //scrivo l'hash nel file
147    for(int i = 0; i < N; i++) {
148        for(int j = i + 1; j < N; j++) {
149            uint64_t key = encode_edge(i, j);
150            hash_insert(new_hash, key);
151        }
152    }
153
154    //scrivo l'hash nel file
155    for(int i = 0; i < N; i++) {
156        for(int j = i + 1; j < N; j++) {
157            uint64_t key = encode_edge(j, i);
158            hash_insert(new_hash, key);
159        }
160    }
161
162    //scrivo l'hash nel file
163    for(int i = 0; i < N; i++) {
164        for(int j = i + 1; j < N; j++) {
165            uint64_t key = encode_edge(i, j);
166            hash_insert(new_hash, key);
167        }
168    }
169
170    //scrivo l'hash nel file
171    for(int i = 0; i < N; i++) {
172        for(int j = i + 1; j < N; j++) {
173            uint64_t key = encode_edge(j, i);
174            hash_insert(new_hash, key);
175        }
176    }
177
178    //scrivo l'hash nel file
179    for(int i = 0; i < N; i++) {
180        for(int j = i + 1; j < N; j++) {
181            uint64_t key = encode_edge(i, j);
182            hash_insert(new_hash, key);
183        }
184    }
185
186    //scrivo l'hash nel file
187    for(int i = 0; i < N; i++) {
188        for(int j = i + 1; j < N; j++) {
189            uint64_t key = encode_edge(j, i);
190            hash_insert(new_hash, key);
191        }
192    }
193
194    //scrivo l'hash nel file
195    for(int i = 0; i < N; i++) {
196        for(int j = i + 1; j < N; j++) {
197            uint64_t key = encode_edge(i, j);
198            hash_insert(new_hash, key);
199        }
200    }
201
202    //scrivo l'hash nel file
203    for(int i = 0; i < N; i++) {
204        for(int j = i + 1; j < N; j++) {
205            uint64_t key = encode_edge(j, i);
206            hash_insert(new_hash, key);
207        }
208    }
209
210    //scrivo l'hash nel file
211    for(int i = 0; i < N; i++) {
212        for(int j = i + 1; j < N; j++) {
213            uint64_t key = encode_edge(i, j);
214            hash_insert(new_hash, key);
215        }
216    }
217
218    //scrivo l'hash nel file
219    for(int i = 0; i < N; i++) {
220        for(int j = i + 1; j < N; j++) {
221            uint64_t key = encode_edge(j, i);
222            hash_insert(new_hash, key);
223        }
224    }
225
226    //scrivo l'hash nel file
227    for(int i = 0; i < N; i++) {
228        for(int j = i + 1; j < N; j++) {
229            uint64_t key = encode_edge(i, j);
230            hash_insert(new_hash, key);
231        }
232    }
233
234    //scrivo l'hash nel file
235    for(int i = 0; i < N; i++) {
236        for(int j = i + 1; j < N; j++) {
237            uint64_t key = encode_edge(j, i);
238            hash_insert(new_hash, key);
239        }
240    }
241
242    //scrivo l'hash nel file
243    for(int i = 0; i < N; i++) {
244        for(int j = i + 1; j < N; j++) {
245            uint64_t key = encode_edge(i, j);
246            hash_insert(new_hash, key);
247        }
248    }
249
250    //scrivo l'hash nel file
251    for(int i = 0; i < N; i++) {
252        for(int j = i + 1; j < N; j++) {
253            uint64_t key = encode_edge(j, i);
254            hash_insert(new_hash, key);
255        }
256    }
257
258    //scrivo l'hash nel file
259    for(int i = 0; i < N; i++) {
260        for(int j = i + 1; j < N; j++) {
261            uint64_t key = encode_edge(i, j);
262            hash_insert(new_hash, key);
263        }
264    }
265
266    //scrivo l'hash nel file
267    for(int i = 0; i < N; i++) {
268        for(int j = i + 1; j < N; j++) {
269            uint64_t key = encode_edge(j, i);
270            hash_insert(new_hash, key);
271        }
272    }
273
274    //scrivo l'hash nel file
275    for(int i = 0; i < N; i++) {
276        for(int j = i + 1; j < N; j++) {
277            uint64_t key = encode_edge(i, j);
278            hash_insert(new_hash, key);
279        }
280    }
281
282    //scrivo l'hash nel file
283    for(int i = 0; i < N; i++) {
284        for(int j = i + 1; j < N; j++) {
285            uint64_t key = encode_edge(j, i);
286            hash_insert(new_hash, key);
287        }
288    }
289
290    //scrivo l'hash nel file
291    for(int i = 0; i < N; i++) {
292        for(int j = i + 1; j < N; j++) {
293            uint64_t key = encode_edge(i, j);
294            hash_insert(new_hash, key);
295        }
296    }
297
298    //scrivo l'hash nel file
299    for(int i = 0; i < N; i++) {
300        for(int j = i + 1; j < N; j++) {
301            uint64_t key = encode_edge(j, i);
302            hash_insert(new_hash, key);
303        }
304    }
305
306    //scrivo l'hash nel file
307    for(int i = 0; i < N; i++) {
308        for(int j = i + 1; j < N; j++) {
309            uint64_t key = encode_edge(i, j);
310            hash_insert(new_hash, key);
311        }
312    }
313
314    //scrivo l'hash nel file
315    for(int i = 0; i < N; i++) {
316        for(int j = i + 1; j < N; j++) {
317            uint64_t key = encode_edge(j, i);
318            hash_insert(new_hash, key);
319        }
320    }
321
322    //scrivo l'hash nel file
323    for(int i = 0; i < N; i++) {
324        for(int j = i + 1; j < N; j++) {
325            uint64_t key = encode_edge(i, j);
326            hash_insert(new_hash, key);
327        }
328    }
329
330    //scrivo l'hash nel file
331    for(int i = 0; i < N; i++) {
332        for(int j = i + 1; j < N; j++) {
333            uint64_t key = encode_edge(j, i);
334            hash_insert(new_hash, key);
335        }
336    }
337
338    //scrivo l'hash nel file
339    for(int i = 0; i < N; i++) {
340        for(int j = i + 1; j < N; j++) {
341            uint64_t key = encode_edge(i, j);
342            hash_insert(new_hash, key);
343        }
344    }
345
346    //scrivo l'hash nel file
347    for(int i = 0; i < N; i++) {
348        for(int j = i + 1; j < N; j++) {
349            uint64_t key = encode_edge(j, i);
350            hash_insert(new_hash, key);
351        }
352    }
353
354    //scrivo l'hash nel file
355    for(int i = 0; i < N; i++) {
356        for(int j = i + 1; j < N; j++) {
357            uint64_t key = encode_edge(i, j);
358            hash_insert(new_hash, key);
359        }
360    }
361
362    //scrivo l'hash nel file
363    for(int i = 0; i < N; i++) {
364        for(int j = i + 1; j < N; j++) {
365            uint64_t key = encode_edge(j, i);
366            hash_insert(new_hash, key);
367        }
368    }
369
370    //scrivo l'hash nel file
371    for(int i = 0; i < N; i++) {
372        for(int j = i + 1; j < N; j++) {
373            uint64_t key = encode_edge(i, j);
374            hash_insert(new_hash, key);
375        }
376    }
377
378    //scrivo l'hash nel file
379    for(int i = 0; i < N; i++) {
380        for(int j = i + 1; j < N; j++) {
381            uint64_t key = encode_edge(j, i);
382            hash_insert(new_hash, key);
383        }
384    }
385
386    //scrivo l'hash nel file
387    for(int i = 0; i < N; i++) {
388        for(int j = i + 1; j < N; j++) {
389            uint64_t key = encode_edge(i, j);
390            hash_insert(new_hash, key);
391        }
392    }
393
394    //scrivo l'hash nel file
395    for(int i = 0; i < N; i++) {
396        for(int j = i + 1; j < N; j++) {
397            uint64_t key = encode_edge(j, i);
398            hash_insert(new_hash, key);
399        }
400    }
401
402    //scrivo l'hash nel file
403    for(int i = 0; i < N; i++) {
404        for(int j = i + 1; j < N; j++) {
405            uint64_t key = encode_edge(i, j);
406            hash_insert(new_hash, key);
407        }
408    }
409
410    //scrivo l'hash nel file
411    for(int i = 0; i < N; i++) {
412        for(int j = i + 1; j < N; j++) {
413            uint64_t key = encode_edge(j, i);
414            hash_insert(new_hash, key);
415        }
416    }
417
418    //scrivo l'hash nel file
419    for(int i = 0; i < N; i++) {
420        for(int j = i + 1; j < N; j++) {
421            uint64_t key = encode_edge(i, j);
422            hash_insert(new_hash, key);
423        }
424    }
425
426    //scrivo l'hash nel file
427    for(int i = 0; i < N; i++) {
428        for(int j = i + 1; j < N; j++) {
429            uint64_t key = encode_edge(j, i);
430            hash_insert(new_hash, key);
431        }
432    }
433
434    //scrivo l'hash nel file
435    for(int i = 0; i < N; i++) {
436        for(int j = i + 1; j < N; j++) {
437            uint64_t key = encode_edge(i, j);
438            hash_insert(new_hash, key);
439        }
440    }
441
442    //scrivo l'hash nel file
443    for(int i = 0; i < N; i++) {
444        for(int j = i + 1; j < N; j++) {
445            uint64_t key = encode_edge(j, i);
446            hash_insert(new_hash, key);
447        }
448    }
449
450    //scrivo l'hash nel file
451    for(int i = 0; i < N; i++) {
452        for(int j = i + 1; j < N; j++) {
453            uint64_t key = encode_edge(i, j);
454            hash_insert(new_hash, key);
455        }
456    }
457
458    //scrivo l'hash nel file
459    for(int i = 0; i < N; i++) {
460        for(int j = i + 1; j < N; j++) {
461            uint64_t key = encode_edge(j, i);
462            hash_insert(new_hash, key);
463        }
464    }
465
466    //scrivo l'hash nel file
467    for(int i = 0; i < N; i++) {
468        for(int j = i + 1; j < N; j++) {
469            uint64_t key = encode_edge(i, j);
470            hash_insert(new_hash, key);
471        }
472    }
473
474    //scrivo l'hash nel file
475    for(int i = 0; i < N; i++) {
476        for(int j = i + 1; j < N; j++) {
477            uint64_t key = encode_edge(j, i);
478            hash_insert(new_hash, key);
479        }
480    }
481
482    //scrivo l'hash nel file
483    for(int i = 0; i < N; i++) {
484        for(int j = i + 1; j < N; j++) {
485            uint64_t key = encode_edge(i, j);
486            hash_insert(new_hash, key);
487        }
488    }
489
490    //scrivo l'hash nel file
491    for(int i = 0; i < N; i++) {
492        for(int j = i + 1; j < N; j++) {
493            uint64_t key = encode_edge(j, i);
494            hash_insert(new_hash, key);
495        }
496    }
497
498    //scrivo l'hash nel file
499    for(int i = 0; i < N; i++) {
500        for(int j = i + 1; j < N; j++) {
501            uint64_t key = encode_edge(i, j);
502            hash_insert(new_hash, key);
503        }
504    }
505
506    //scrivo l'hash nel file
507    for(int i = 0; i < N; i++) {
508        for(int j = i + 1; j < N; j++) {
509            uint64_t key = encode_edge(j, i);
510            hash_insert(new_hash, key);
511        }
512    }
513
514    //scrivo l'hash nel file
515    for(int i = 0; i < N; i++) {
516        for(int j = i + 1; j < N; j++) {
517            uint64_t key = encode_edge(i, j);
518            hash_insert(new_hash, key);
519        }
520    }
521
522    //scrivo l'hash nel file
523    for(int i = 0; i < N; i++) {
524        for(int j = i + 1; j < N; j++) {
525            uint64_t key = encode_edge(j, i);
526            hash_insert(new_hash, key);
527        }
528    }
529
530    //scrivo l'hash nel file
531    for(int i = 0; i < N; i++) {
532        for(int j = i + 1; j < N; j++) {
533            uint64_t key = encode_edge(i, j);
534            hash_insert(new_hash, key);
535        }
536    }
537
538    //scrivo l'hash nel file
539    for(int i = 0; i < N; i++) {
540        for(int j = i + 1; j < N; j++) {
541            uint64_t key = encode_edge(j, i);
542            hash_insert(new_hash, key);
543        }
544    }
545
546    //scrivo l'hash nel file
547    for(int i = 0; i < N; i++) {
548        for(int j = i + 1; j < N; j++) {
549            uint64_t key = encode_edge(i, j);
550            hash_insert(new_hash, key);
551        }
552    }
553
554    //scrivo l'hash nel file
555    for(int i = 0; i < N; i++) {
556        for(int j = i + 1; j < N; j++) {
557            uint64_t key = encode_edge(j, i);
558            hash_insert(new_hash, key);
559        }
560    }
561
562    //scrivo l'hash nel file
563    for(int i = 0; i < N; i++) {
564        for(int j = i + 1; j < N; j++) {
565            uint64_t key = encode_edge(i, j);
566            hash_insert(new_hash, key);
567        }
568    }
569
570    //scrivo l'hash nel file
571    for(int i = 0; i < N; i++) {
572        for(int j = i + 1; j < N; j++) {
573            uint64_t key = encode_edge(j, i);
574            hash_insert(new_hash, key);
575        }
576    }
577
578    //scrivo l'hash nel file
579    for(int i = 0; i < N; i++) {
580        for(int j = i + 1; j < N; j++) {
581            uint64_t key = encode_edge(i, j);
582            hash_insert(new_hash, key);
583        }
584    }
585
586    //scrivo l'hash nel file
587    for(int i = 0; i < N; i++) {
588        for(int j = i + 1; j < N; j++) {
589            uint64_t key = encode_edge(j, i);
590            hash_insert(new_hash, key);
591        }
592    }
593
594    //scrivo l'hash nel file
595    for(int i = 0; i < N; i++) {
596        for(int j = i + 1; j < N; j++) {
597            uint64_t key = encode_edge(i, j);
598            hash_insert(new_hash, key);
599        }
600    }
601
602    //scrivo l'hash nel file
603    for(int i = 0; i < N; i++) {
604        for(int j = i + 1; j < N; j++) {
605            uint64_t key = encode_edge(j, i);
606            hash_insert(new_hash, key);
607        }
608    }
609
610    //scrivo l'hash nel file
611    for(int i = 0; i < N; i++) {
612        for(int j = i + 1; j < N; j++) {
613            uint64_t key = encode_edge(i, j);
614            hash_insert(new_hash, key);
615        }
616    }
617
618    //scrivo l'hash nel file
619    for(int i = 0; i < N; i++) {
620        for(int j = i + 1; j < N; j++) {
621            uint64_t key = encode_edge(j, i);
622            hash_insert(new_hash, key);
623        }
624    }
625
626    //scrivo l'hash nel file
627    for(int i = 0; i < N; i++) {
628        for(int j = i + 1; j < N; j++) {
629            uint64_t key = encode_edge(i, j);
630            hash_insert(new_hash, key);
631        }
632    }
633
634    //scrivo l'hash nel file
635    for(int i = 0; i < N; i++) {
636        for(int j = i + 1; j < N; j++) {
637            uint64_t key = encode_edge(j, i);
638            hash_insert(new_hash, key);
639        }
640    }
641
642    //scrivo l'hash nel file
643    for(int i = 0; i < N; i++) {
644        for(int j = i + 1; j < N; j++) {
645            uint64_t key = encode_edge(i, j);
646            hash_insert(new_hash, key);
647        }
648    }
649
650    //scrivo l'hash nel file
651    for(int i = 0; i < N; i++) {
652        for(int j = i + 1; j < N; j++) {
653            uint64_t key = encode_edge(j, i);
654            hash_insert(new_hash, key);
655        }
656    }
657
658    //scrivo l'hash nel file
659    for(int i = 0; i < N; i++) {
660        for(int j = i + 1; j < N; j++) {
661            uint64_t key = encode_edge(i, j);
662            hash_insert(new_hash, key);
663        }
664    }
665
666    //scrivo l'hash nel file
667    for(int i = 0; i < N; i++) {
668        for(int j = i + 1; j < N; j++) {
669            uint64_t key = encode_edge(j, i);
670            hash_insert(new_hash, key);
671        }
672    }
673
674    //scrivo l'hash nel file
675    for(int i = 0; i < N; i++) {
676        for(int j = i + 1; j < N; j++) {
677            uint64_t key = encode_edge(i, j);
678            hash_insert(new_hash, key);
679        }
680    }
681
682    //scrivo l'hash nel file
683    for(int i = 0; i < N; i++) {
684        for(int j = i + 1; j < N; j++) {
685            uint64_t key = encode_edge(j, i);
686            hash_insert(new_hash, key);
687        }
688    }
689
690    //scrivo l'hash nel file
691    for(int i = 0; i < N; i++) {
692        for(int j = i + 1; j < N; j++) {
693            uint64_t key = encode_edge(i, j);
694            hash_insert(new_hash, key);
695        }
696    }
697
698    //scrivo l'hash nel file
699    for(int i = 0; i < N; i++) {
700        for(int j = i + 1; j < N; j++) {
701            uint64_t key = encode_edge(j, i);
702            hash_insert(new_hash, key);
703        }
704    }
705
706    //scrivo l'hash nel file
707    for(int i = 0; i < N; i++) {
708        for(int j = i + 1; j < N; j++) {
709            uint64_t key = encode_edge(i, j);
710            hash_insert(new_hash, key);
711       }
712    }
713
714    //scrivo l'hash nel file
715    for(int i = 0; i < N; i++) {
716        for(int j = i + 1; j < N; j++) {
717            uint64_t key = encode_edge(j, i);
718            hash_insert(new_hash, key);
719       }
720    }
721
722    //scrivo l'hash nel file
723    for(int i = 0; i < N; i++) {
724        for(int j = i + 1; j < N; j++) {
725            uint64_t key = encode_edge(i, j);
726            hash_insert(new_hash, key);
727       }
728    }
729
730    //scrivo l'hash nel file
731    for(int i = 0; i < N; i++) {
732        for(int j = i + 1; j < N; j++) {
733            uint64_t key = encode_edge(j, i);
734            hash_insert(new_hash, key);
735       }
736    }
737
738    //scrivo l'hash nel file
739    for(int i = 0; i < N; i++) {
740        for(int j = i + 1; j < N; j++) {
741            uint64_t key = encode_edge(i, j);
742            hash_insert(new_hash, key);
743       }
744    }
745
746    //scrivo l'hash nel file
747    for(int i = 0; i < N; i++) {
748        for(int j = i + 1; j < N; j++) {
749            uint64_t key = encode_edge(j, i);
750            hash_insert(new_hash, key);
751       }
752    }
753
754    //scrivo l'hash nel file
755    for(int i = 0; i < N; i++) {
756        for(int j = i + 1; j < N; j++) {
757            uint64_t key = encode_edge(i, j);
758            hash_insert(new_hash, key);
759       }
760    }
761
762    //scrivo l'hash nel file
763    for(int i = 0; i < N; i++) {
764        for(int j = i + 1; j < N; j++) {
765            uint64_t key = encode_edge(j, i);
766            hash_insert(new_hash, key);
767       }
768    }
769
770    //scrivo l'hash nel file
771    for(int i = 0; i < N; i++) {
772        for(int j = i + 1; j < N; j++) {
773            uint64_t key = encode_edge(i, j);
774            hash_insert(new_hash, key);
775       }
776    }
777
778    //scrivo l'hash nel file
779    for(int i = 0; i < N; i++) {
780        for(int j = i + 1; j < N; j++) {
781            uint64_t key = encode_edge(j, i);
782            hash_insert(new_hash, key);
783       }
784    }
785
786    //scrivo l'hash nel file
787    for(int i = 0; i < N; i++) {
788        for(int j = i + 1; j < N; j++) {
789            uint64_t key = encode_edge(i, j);
790            hash_insert(new_hash, key);
791       }
792    }
793
794    //scrivo l'hash nel file
795    for(int i = 0; i < N; i++) {
796        for(int j = i + 1; j < N; j++) {
797            uint64_t key = encode_edge(j, i);
798            hash_insert(new_hash, key);
799       }
800    }
801
802    //scrivo l'hash nel file
803    for(int i = 0; i < N; i++) {
804        for(int j = i + 1; j < N; j++) {
805            uint64_t key = encode_edge(i, j);
806            hash_insert(new_hash, key);
807       }
808    }
809
810    //scrivo l'hash nel file
811    for(int i = 0; i < N; i++) {
812        for(int j = i + 1; j < N; j++) {
813            uint64_t key = encode_edge(j, i);
814            hash_insert(new_hash, key);
815       }
816    }
817
818    //scrivo l'hash nel file
819    for(int i = 0; i < N; i++) {
820        for(int j = i + 1; j < N; j++) {
821            uint64_t key = encode_edge(i, j);
822            hash_insert(new_hash, key);
823       }
824    }
825
826    //scrivo l'hash nel file
827    for(int i = 0; i < N; i++) {
828        for(int j = i + 1; j < N; j++) {
829            uint64_t key = encode_edge(j, i);
830            hash_insert(new_hash, key);
831       }
832    }
833
834    //scrivo l'hash nel file
835    for(int i = 0; i < N; i++) {
836        for(int j = i + 1; j < N; j++) {
837            uint64_t key = encode_edge(i, j);
838            hash_insert(new_hash, key);
839       }
840    }
841
842    //scrivo l'hash nel file
843    for(int i = 0; i < N; i++) {
844        for(int j = i + 1; j < N; j++) {
845            uint64_t key = encode_edge(j, i);
846            hash_insert(new_hash, key);
847       }
848    }
849
850    //scrivo l'hash nel file
851    for(int i = 0; i < N; i++) {
852        for(int j = i + 1; j < N; j++) {
853            uint64_t key = encode_edge(i, j);
854            hash_insert(new_hash, key);
855       }
856    }
857
858    //scrivo l'hash nel file
859    for(int i = 0; i < N; i++) {
860        for(int j = i + 1; j < N; j++) {
861            uint64_t key = encode_edge(j, i);
862            hash_insert(new_hash, key);
863       }
864    }
865
866    //scrivo l'hash nel file
867    for(int i = 0; i < N; i++) {
868        for(int j = i + 1; j < N; j++) {
869            uint64_t key = encode_edge(i, j);
870            hash_insert(new_hash, key);
871       }
872    }
873
874    //scrivo l'hash nel file
875    for(int i = 0; i < N; i++) {
876        for(int j = i + 1; j < N; j++) {
877            uint64_t key = encode_edge(j, i);
878            hash_insert(new_hash, key);
879       }
880    }
881
882    //scrivo l'hash nel file
883    for(int i = 0; i < N; i++) {
884        for(int j = i + 1; j < N; j++) {
885            uint64_t key = encode_edge(i, j);
886            hash_insert(new_hash, key);
887       }
888    }
889
890    //scrivo l'hash nel file
891    for(int i = 0; i < N; i++) {
892        for(int j = i + 1; j < N; j++) {
893            uint64_t key = encode_edge(j, i);
894            hash_insert(new_hash, key);
895       }
896    }
897
898    //scrivo l'hash nel file
899    for(int i = 0; i < N; i++) {
900        for(int j = i + 1; j < N; j++) {
901            uint64_t key = encode_edge(i, j);
902            hash_insert(new_hash, key);
903       }
904    }
905
9
```

```

48         int v = random() % N;
49         if (u == v) continue;
50
51         uint64_t key = encode_edge(u, v);
52         if (!hash_lookup(new_hash, key)) {
53             hash_insert(new_hash, key);
54             fprintf(filePointer, "%d %d\n", u, v);
55             edge_count += 2;
56             added++;
57         }
58     }
59
60     freeHash(new_hash);
61     fclose(filePointer);
62
63     //creo un file contenente le informazioni principali del grafo (
64     // numero di nodi, archi e tipologia -> direzionato o non)
65     char meta_data[256];
66     snprintf(meta_data, sizeof(meta_data), "%smeta", location);
67     filePointer = fopen(meta_data, "w");
68     fprintf(filePointer, "%d %d %d\n", N, edge_count/2, GRAPHTYPE);
69     //2 non direzionato, 1 direzionato
70     fclose(filePointer);
71     printf("File con %d archi scritto correttamente.\n", edge_count);
72     return ;
73 }
```

Implementazione: Generatore grafi del modello Barabási–Albert/Holme–Kim.

```

1 void generateBarabasiHolmeUndirectedGraph(char *location, int nodes) {
2
3     int edge_count = 0;
4     double p = PROBABILITY;
5     int N = nodes; // numero totale di nodi
6     int m0 = (int)STARTNODES_BARABASI; // numero iniziale di nodi
7     int m = (int)EDGEPERNODES_BARABASI; // numero di connessioni per
8     //ogni nuovo nodo
9     if (m > m0 || m0 >= N) {
10         printf("Errore: richiedi m <= m0 e m0 < N.\n");
11         return ;
12     }
13     EdgeList *new_elist = initEdgeBuffer(N);
14     if (new_elist == NULL) {
15         perror("GraphGen: genBarabasi: out of memory!!\n\n");
16         exit(1);
17     }
18     EdgeHash *new_hash = malloc(sizeof(EdgeHash));
19     hash_init(new_hash, compute_hash_capacity(N, m0, m, p));
20     int *degrees = calloc(N, sizeof(int)); //array dei gradi per ogni
21     //nodo n
22     FILE *filePointer = NULL;
23     int last_used_node = 0;
24
25     filePointer = fopen(location, "w");
26     if (!filePointer) {
27         perror("fopen\n");
28         freeAll(new_hash, degrees);
29         exit(1);
30     }
31     srand(time(NULL));
32 }
```

```

33     printf("Generando grafo BA/HK...\n\n");
34     // Inizializza grafo di partenza come clique di m0 nodi
35     for (int i = 0; i < m0; i++) {
36         for (int j = i + 1; j < m0; j++) {
37             add_edge(new_elist, i, j);
38             uint64_t key = encode_edge(i, j);
39             hash_insert(new_hash, key);
40             edge_count += 2;
41             degrees[i]++;
42             degrees[j]++;
43             fprintf(filePointer, "%d %d\n", i, j);
44         }
45     }
46
47     // Aggiunta dei nuovi nodi
48     for (int new_node = m0; new_node < N; new_node++) {
49         int added = 0;
50         while (added < m) {
51             int chosen = choose_node(degrees, edge_count, new_node);
52             // Evita self-loop e duplicati
53             if (new_node == chosen) continue;
54             uint64_t key = encode_edge(new_node, chosen);
55             if (hash_lookup(new_hash, key)) continue;
56
57             // Aggiungi arco
58             add_edge(new_elist, new_node, chosen);
59             hash_insert(new_hash, key);
60             edge_count += 2;
61             degrees[new_node]++;
62             degrees[chosen]++;
63             fprintf(filePointer, "%d %d\n", new_node, chosen);
64             added++;
65
66             // Holme Kim formazioni triadi con probabilita' p
67             if (((((double) random() / RAND_MAX) < p) && new_elist->size
68                 > 0) {
69                 // scegli un vicino casuale di "chosen"
70                 int u_rand = -1, v_rand = -1, w = -1;
71                 const int MAX_TRIES = 16;      // volte che prova a cercare
72                 un arco di chosen
73                 for (int i = 0; i < MAX_TRIES; i++) {
74                     pick_random_edge(new_elist, &u_rand, &v_rand);
75                     if (u_rand == chosen) { w = v_rand; break; }
76                     if (v_rand == chosen) { w = u_rand; break; }
77                 }
78
79                 if (w >= 0) {
80                     // evita self-loop e duplicati
81                     if (new_node == w) continue;
82                     uint64_t key2 = encode_edge(new_node, w);
83                     if (hash_lookup(new_hash, key2)) continue;
84
85                     // Aggiungo arco
86                     add_edge(new_elist, new_node, w);
87                     hash_insert(new_hash, key2);
88                     edge_count += 2;
89                     degrees[new_node]++;
90                     if (w < 0 || w>=N) {
91                         printf("GraphGenerator: generateBarabasi: w=%d
92                             fuori range [0, %d]!\n", w, N);
93                         exit(1);
94                     }
95                     degrees[w]++;
96                     fprintf(filePointer, "%d %d\n", new_node, w);
97                     added++;
98                 }
99             }
100         }
101     }
102 }
```

```

96             }
97         }
98     }
99     freeAllStruct(new_hash, degrees, new_elist);
100    fclose(filePointer);
101    //creo un file contenente le informazioni principali del grafo (
102    //numero di nodi, archi e tipologia -> direzionato o non)
103    char meta_data[256];
104    snprintf(meta_data, sizeof(meta_data), "%smeta", location);
105    filePointer = fopen(meta_data, "w");
106    fprintf(filePointer, "%d %d %d\n", N, edge_count/2, GRAPHTYPE);
107    //2 non direzionato, 1 direzionato
108    fclose(filePointer);
109    printf("File con %d archi scritto correttamente.\n", edge_count);
110    return ;
111}

```

Inoltre, si utilizza una funzione costruita in C per importare i grafi generati dai modelli o scaricati delle repository esterne, ad esempio semplificato, nel caso delle liste di adiacenza, e considerando grafi non orientati, tale funzione è:

```

1 void ImportUndirectedListGraph(ListGraph *graph, char *string){
2     FILE * filePointer = fileOpen(string);
3     int numVertices, numToRead;
4     long numEdges;
5     char buffer[32];
6
7     if(fgets(buffer, sizeof(buffer), filePointer) != NULL)
8         if(sscanf(buffer, "%d %ld %d", &numVertices, &numEdges, &numToRead)
9             != 3 || (numToRead < 2 || numToRead > 3) ) exit(1);
10
11    initListGraph(graph, numVertices, false);
12    int node1, node2; //legge due numeri, che siano 2 o 3 per riga
13
14    if(numToRead == 2)
15        while( fscanf(filePointer, "%d %d", &node1, &node2) == 2)
16            doubleListEdge(graph, node1, node2);
17    if(numToRead == 3)
18        while( fscanf(filePointer, "%d %d %*d", &node1, &node2) == 3)
19            doubleListEdge(graph, node1, node2);
20    if(!feof(filePointer)) {
21        perror("Attenzione! Lettura interrotta prima della fine del file
22                !\n");
23        exit(1);
24    }
25    fclose(filePointer);
26    if(graph->numEdges != numEdges*2) {
27        perror("\nErrore nel parsing! Numero di archi non corrisponde!\n\
28                \n");
29        exit(1);
30    }
31}

```

Bibliografia

- [1] Bevin Brett. Memory performance in a nutshell, 6 2016. Technical article.
- [2] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.
- [3] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- [4] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2012. Disponibile a <https://ieeexplore.ieee.org/document/6468458>.
- [5] Sally A McKee. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 2004(1):19–25, 2004.
- [6] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2 edition, 2003. Disponibile a https://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf.
- [7] Petter Holme and Beom Jun Kim. Growing scale-free networks with tunable clustering. *Physical Review E*, 65(2):026107, 2002.
- [8] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
- [9] M. E. J. Newman and D. J. Watts. Renormalization group analysis of the small-world network model. *Physics Letters A*, 263(4–6):341–346, 1999.
- [10] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.
- [11] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994. Discussione ed evidenze sperimentalì su false sharing e località spaziale.

Ringraziamenti

Desidero ringraziare innanzitutto la relatrice di questa tesina, la Prof.^{ssa} Tiziana Calamoneri, per la disponibilità e gentilezza dimostrate durante la stesura del lavoro, e per l'avermi guidato e supportato nella fase più importante del mio percorso accademico.

Ringrazio anche la mia famiglia per il sostegno quotidiano, dalle piccole attenzioni ai grandi aiuti.

Infine, ringrazio colleghi e amici per il loro supporto e per lo spirito di crescita condivisa, che considero un valore fondamentale per lo sviluppo personale e della comunità.