Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Primitive per l'Analisi di Grandi Grafi in MapReduce

Relatore:
Prof. Andrea Alberto Pietracaprina

Correlatore:
Prof. Geppino Pucci

Laureando:
Gianmaria Parigi Bini

Anno accademico 2015/2016

ai miei genitori

a Silvia

Abstract

I grafi sono strutture storicamente utilizzate nel mondo dell'Informatica: gli esempi più famosi sono il Web e i social network. Tuttavia i grafi hanno trovato applicazioni in numerosi altri campi, tra cui fisica, biologia, chimica o bioinformatica.

Queste reti hanno una caratteristica comune: stanno rapidamente diventando sempre più grandi. Si pensi alla sola Facebook, che ha annunciato recentemente di avere 1,55 miliardi di utenti attivi.

E' di fondamentale importanza sviluppare allora degli algoritmi in grado di processare rapidamente questi grafi. La manipolazione di grafi di queste dimensioni, utilizzando una singola macchina, o è semplicemente impossibile oppure richiede macchine molto costose. MapReduce è un modello di computazione che permette di realizzare algoritmi paralleli per processare grandi moli di dati utilizzando un'architettura distribuita.

In questa tesi sono stati sviluppati ed ottimizzati diversi algoritmi MapReduce per calcolare due particolari proprietà di un grafo: il *diametro*, che rappresenta la distanza massima esistente nel grafo, utile ad esempio per studiare l'evoluzione di una rete, e la *centralità*, che permette di quantificare l'importanza relativa di un nodo all'interno della rete. Particolare enfasi è stata posta nell'applicazione di tecniche di *clustering*, che permette di diminuire la taglia di un grafo.

Indice

Capitolo 1

Introduzione

Nell'ultimo decennio i grafi hanno assunto un ruolo di primo piano nel mondo dell'informatica e non solo, grazie alla loro presenza in quasi ogni ambito di ricerca. L'esempio più classico di grafo è il Web, dove ogni pagina è un nodo del grafo e un link è rappresentato come un arco tra due nodi.

Tra questi, particolare rilevanza stanno assumendo le reti sociali come Facebook o Twitter. Possiamo rappresentare infatti con dei grafi qualsiasi tipo di rete, sia fisica che virtuale, come reti stradali, reti telefoniche, o le interazioni tra proteine in una cellula.

Non solo i grafi si stanno espandendo a nuove applicazioni, come la bioinformatica, ma le reti associate stanno diventando sempre più grandi. Basti pensare, ad esempio, che la sola Facebook ha al momento 1,55 miliardi di utenti attivi e ognuno di questi ha centinaia di amicizie, producendo quindi un grafo con centinaia di miliardi di archi. Per questi motivi, lo sviluppo di algoritmi efficienti in grado di analizzare grandi grafi ha importanti ricadute sia commerciali che di ricerca.

Il primo problema che affronteremo è il calcolo del *diametro* di un grafo, ovvero la distanza massima esistente tra due nodi della rete. Questa proprietà ha importanti applicazioni in diversi campi, tra cui la ricerca operativa, la biologia o la chimica, ad esempio per determinare il numero di reazioni necessarie alla produzione di un particolare metabolita.

Il secondo problema che affronteremo è quello del calcolo della *centralità* di un nodo del grafo, che in qualche misura quantifica l'importanza del nodo all'interno della rete. Di particolare interesse è il calcolo dell'insieme dei k nodi più importanti, chiamati Top-k. Le applicazioni sono limitate soltanto dalla fantasia. Ad esempio, quando facciamo una ricerca su Google ci vengono restituiti i risultati ordinati per centralità decrescente, oppure siamo interessati a trovare le persone più influenti all'interno di una rete sociale. Ancora, potremmo voler determinare quali computer all'interno di una rete sono importanti per mantenere il traffico regolare.

Per ognuno di questi problemi si sono sempre sviluppati algoritmi estremamente efficienti, ottimizzati per uno specifico problema e per la specifica macchina su cui l'al-

goritmo dev'essere eseguito.
Ci accorgiamo però che esiste un primo problema: gli algoritmi efficienti su un particolare hardware spesso non lo sono su macchine diverse e per processare grandi grafi c'è bisogno di macchine molto costose.
Esiste poi un secondo problema: cosa succede se il grafo da analizzare diventa troppo grande per essere processato dalla macchina? Spesso, piuttosto di effettuare un upgrade dell'hardware è più conveniente sostituire l'intera macchina, e a quel punto quella vecchia diventa inutilizzabile.

Per ovviare a queste problematiche nel 2004 è stato introdotto *MapReduce*, un paradigma di programmazione che permette di sviluppare algoritmi paralleli in grado di girare su un *cluster* di macchine. In questo modo, quando il problema in input diventa troppo grande, invece di sostituire le macchine è sufficiente *aggiungerne* altre, al fine di aumentare la potenza di calcolo complessiva.

Obiettivo di questa tesi sarà quindi lo sviluppo e lo studio di diversi algoritmi MapReduce (implementati su *Apache Spark*) in grado di risolvere efficacemente i due problemi descritti: diametro e centralità.

Per far ciò, verranno migliorate alcune soluzioni esistenti e verranno introdotti diversi nuovi algoritmi basati sul *clustering* di un grafo, ovvero il partizionamento del grafo in moduli denominati *cluster*. In particolare, verranno studiati per la prima volta in letteratura alcuni approcci basati sul clustering per determinare i Top-k del grafo.

Lavori precedenti

Il partizionamento di un grafo in cluster per risolvere problemi complessi è un approccio molto studiato in letteratura [1, 10, 18, 8]. In particolare, algoritmi di clustering MapReduce sono stati studiati in [7, 9, 8]. In [8] gli autori hanno ottenuto un algoritmo in grado di approssimare il diametro di grafi pesati entro un fattore $1,5$ in un numero di iterazioni parallele sub-lineare al diametro.

Un altro filone di ricerca per ottenere il diametro di un grafo è quello basato sull'utilizzo di contatori, ad esempio [19, 15, 2]. Per quanto riguarda l'architettura di nostro interesse, il primo algoritmo sviluppato è stato HADI [15] ma sono poi stati sviluppati algoritmi più efficienti, sfruttando contatori HyperLogLog [3] in [7] e [5]. In quest'ultimo lavoro, in particolare, è stata sfruttata l'implementazione molto efficiente di HyperLogLog utilizzata in HyperANF [2]. HyperANF, seppur molto veloce, è un algoritmo sviluppato per macchine multiprocessore dotate di larga memoria condivisa, pertanto non è di nostro interesse.

Il tema della centralità è un problema che sta suscitando grande dibattito, in quanto non esiste una definizione perfetta per qualsiasi applicazione. Tuttavia, studi recenti [3] hanno portato alla ribalta l'Harmonic Centrality. Un algoritmo efficiente in grado di calcolare questa definizione di centrality è [4], dagli autori di [2], tuttavia è ancora destinato a costose macchine multiprocessore.

Per quella che è la nostra conoscenza, non esistono algoritmi MapReduce in grado di approssimare l'Harmonic Centrality di un nodo, né esistono in generale evidenze di approcci basati su clustering per il calcolo di centrality.

Risultati raggiunti

E' stata realizzata per la prima volta una implementazione di contatore *HyperLogLog* ottimizzata per l'utilizzo in *Spark*, che ha permesso di analizzare grandi grafi in tempi considerevolmente minori di quanto precedentemente richiesto. Sono stati poi introdotti diversi nuovi algoritmi.

Innanzitutto è stato sviluppato un nuovo algoritmo di clustering, *Degree Clustering*, che si è rivelato particolarmente efficace per la stima del diametro nelle reti sociali rispetto ai precedenti algoritmi. Per migliorare la stima del diametro anche sulle reti stradali, è stato sviluppato un nuovo algoritmo ottimizzato che permette ai cluster di "sovrapporsi". A prezzo di un leggero aumento del tempo di esecuzione, questo algoritmo ha permesso di migliorare la stima del diametro in reti stradali di un ordine di grandezza rispetto ai classici algoritmi di clustering. E' stato possibile infatti portare la stima del diametro, in alcuni casi, da un errore relativo del 37% a meno del 5%.

Infine, per la prima volta è stato sviluppato un approccio basato sul clustering per calcolare i Top-k di un grafo, utilizzando un clustering per fasi chiamato *Progressive Clustering*. E' stato così possibile ottenere, in pochi minuti, i 10/20 nodi più importanti di un grafo di milioni di nodi con una percentuale di accuratezza superiore al 90%.

Struttura della tesi

Nel Capitolo 2 verranno fornite alcune definizioni di base sulla terminologia che verrà usata nel resto della tesi, quindi segue una breve descrizione sul paradigma MapReduce e sulle caratteristiche chiave di Apache Spark, che è il framework utilizzato per implementare e testare i vari algoritmi descritti in questo lavoro.

Nel Capitolo 3 verranno descritti e confrontati tre diversi algoritmi che saranno poi usati per calcolare diametro e centralità di un grafo: due algoritmi di clustering, *Randomized Clustering* e *Degree Clustering*, di cui l'ultimo originale, e un più classico algoritmo basato sui contatori *HyperLogLog*.

Nel Capitolo 4 verranno quindi applicati i tre algoritmi precedentemente descritti per il calcolo del diametro di un grafo. In particolare, si potranno apprezzare i miglioramenti portati dal nuovo *Degree Clustering* sulle reti sociali, mentre per migliorare la stima del diametro sulle reti stradali verrà introdotto un nuovo algoritmo chiamato *Multicoloring*.

Segue nel Capitolo 5 una seconda applicazione di questi algoritmi, per individuare i Top-k di un grafo utilizzando l'*Harmonic Centrality*. Per la prima volta, verranno

descritti approcci basati su un clustering per fasi per rifinire le aree più importanti del grafo, fino ad ottenere una buona approssimazione dei Top-k desiderati.

Infine, nel Capitolo 6 verranno riassunti i molteplici risultati raggiunti in questa tesi e verranno delineati i possibili sviluppi, sia teorici che pratici.

Capitolo 2

Nozioni preliminari e ambiente di programmazione

2.1 Definizioni di base

Un *grafo* è una struttura che descrive una relazione tra oggetti. Gli oggetti sono chiamati *vertici* (o *nodi*) e una relazione è rappresentata da un *arco* (o *lato*) tra due vertici. Si distinguono due tipi basilari di grafi, i grafi *orientati* (o grafi *diretti*) e i grafi *non orientati* (o grafi *indiretti*). Un grafo non diretto (dove un lato costituisce un collegamento senza verso tra i due nodi) è rappresentato matematicamente come $G = (V, E)$, dove V è l'insieme dei vertici ed $E = VxV$ è l'insieme di archi. Un grafo si definisce *pesato* se gli archi sono abbinati ad un numero denominato *peso*. Un grafo non orientato si definisce *connesso* se ogni nodo è raggiungibile da qualsiasi altro nodo. Un *percorso* di lunghezza n in G è dato da una sequenza di vertici $v_0, v_1, ..., v_n$ (non necessariamente tutti distinti) e da una sequenza di archi che li collegano $(v_0, v_1), (v_1, v_2), ..., (v_{n-1}, v_n)$. I vertici v_0 e v_n si dicono estremi del percorso. Un percorso con i lati a due a due distinti tra loro prende il nome di *cammino*. Il *cammino minimo* tra due nodi è il cammino più breve (il numero di archi, se grafo indiretto) esistente tra i due nodi. Possono coesistere multipli cammini minimi tra due nodi, se questi hanno pari lunghezza.

Definizione 3.1. Il *grado* di un nodo v è pari al numero di archi incidenti in v.

Definizione 3.2. La *distanza* tra due nodi in un grafo non pesato è equivalente al numero di archi nel cammino minimo che li collega.

Definizione 3.3. Il *diametro* di un grafo G è equivalente alla distanza massima presente tra una qualsiasi coppia di vertici in G.

2.2 Modello di calcolo MapReduce

MapReduce è un paradigma di programmazione con cui è possibile processare parallelamente grandi moli di dati, utilizzando un insieme numeroso di computer (nodi) complessivamente denominato *cluster*.

E' stato inizialmente pubblicato da Google [11] ma ha rapidamente conosciuto una grande popolarità ed è stato implementato in librerie dei più svariati linguaggi. Il nome deriva dalle funzioni *map* e *reduce* tipicamente presenti nei linguaggi funzionali. L'approccio in sé non è innovativo poiché funzioni di map e reduce in ambito distribuito erano già esistenti in altri modelli (e.g. *MPI*), ma l'articolo ha introdotto concetti importanti per quanto riguarda scalabilità e tolleranza agli errori.

Le funzioni, operanti su tuple $< Key, Value >$, sono così definite.

- La funzione *map* riceve in input una tupla $< K_{in}, V_{in} >$ e restituisce in output un numero finito (anche nullo) di tuple $< K_{out}, V_{out} >$.
- La funzione *reduce* riceve in input una chiave K con la lista di tutti i valori associati a quella chiave, quindi riduce la lista di valori a un numero minore (o uguale) di tuple.

I dati da dare in input al programma, solitamente, sono già distribuiti tra i nodi del cluster. Un algoritmo MapReduce si compone infine di 3 fasi.

1. **Map**: ogni nodo processa i propri dati in input applicando la funzione *map*, producendo delle tuple $< K, V >$.
2. **Shuffle**: le tuple vengono smistate nella rete e quelle con la stessa chiave vengono raccolte sullo stesso nodo, secondo una determinata logica.
3. **Reduce**: dopo che le tuple con la stessa chiave sono state raggruppate, ogni nodo potrà processare le liste che gli sono state destinate utilizzando la funzione *reduce*.

La maggior parte dei problemi non può essere però risolta con un singolo ciclo di questa procedura, quindi solitamente si applicano più *round* consecutivi, che ricevono in input l'output del round precedente.

L'articolo originale [11] non fornisce però un modello in grado di analizzare complessità spaziale e temporale di un algoritmo MapReduce definito per round. Un modello in grado di sopperire a questa mancanza, sfruttante come uniche informazioni la memoria locale m disponibile per ogni nodo e la memoria totale M del cluster, è stato definito in [20]. Nel modello si suppone che l'algoritmo analizzato utilizzi per ogni round memoria locale $O(m)$ e memoria globale $O(M)$. Infine, è richiesto che le funzioni di map e reduce siano polinomiali rispetto alla taglia dell'input. Per ulteriori dettagli il lettore è invitato a consultare [20].

2.3 Apache Spark

Esistono diverse implementazioni del modello MapReduce. Una delle prime e la più utilizzata è Apache Hadoop, che tuttavia prevede la memorizzazione su disco dei dati per tutte le fasi intermedie della computazione. In algoritmi iterativi ad alto numero di round, la memorizzazione su disco diventa un enorme collo di bottiglia.

Si è scelto pertanto in questo lavoro di utilizzare Apache Spark, un framework MapReduce innovativo che negli ultimi anni ha cominciato a sostituire Hadoop. Come vedremo, infatti, Spark — seppur imperfetto — è in grado di risolvere i problemi più gravi delle altre implementazioni.

Il framework è sviluppato in *Scala*, un linguaggio funzionale derivato da *Java*, ma è possibile sviluppare applicazioni utilizzando anche *Java* e *Python*.

2.3.1 Caratteristiche

Spark ha introdotto il concetto di *RDD* (Resilient Distributed Dataset) [22], che permette di mantenere in memoria i dati necessari all'esecuzione senza essere obbligati a passare per il disco. Dato che, in mancanza di salvataggio su disco ad ogni fase, non abbiamo più la garanzia di poter recuperare i dati aggiornati in caso di fallimenti, l'*RDD* memorizza lo "storico" delle operazioni effettuate sui dati, in modo che eventualmente si possano ripetere per riottenere i dati persi.

Un altro concetto importante introdotto da Spark è quello di *lazy evaluation*: quando a dei dati vengono applicate diverse operazioni in serie, queste non vengono eseguite finché i risultati dell'operazione non sono effettivamente utilizzati. La conoscenza dell'intera sequenza di operazioni da effettuare sui dati permette all'*engine* di applicare opportune ottimizzazioni e velocizzarne l'esecuzione. Un RDD, se non esplicitamente mantenuto in memoria, verrà ricalcolato ogni volta che verrà utilizzato.

Il programma è controllato da un nodo *master*, mentre i nodi che processano i dati sono definiti *worker*. Il processo eseguito sul nodo master, che esegue la funzione *main*, è definito invece *driver program*.

2.3.2 Principali operazioni

In tabella 2.1 elenchiamo le più importanti funzioni messe a disposizione da Spark.

Altre operazioni interessanti che mette a disposizione Spark, slegate però dal concetto di RDD, sono la funzione *broadcast*, che permette di distribuire un oggetto in sola lettura dal *master* a tutti i *worker* — e quindi rendere disponibile il suo accesso all'interno delle varie funzioni distribuite —, e la funzione *accumulator*, che permette di

Tabella 2.1: Principali operazioni sugli RDD

Funzione	**Descrizione**
count()	Restituisce il numero di elementi nell'RDD.
countByKey()	Restituisce una mappa di valori $(key, count)$ con il numero di elementi per ogni chiave esistente.
filter(func)	Restituisce un nuovo RDD con gli elementi che soddisfano il predicato indicato.
flatMap(func)	Applicando la funzione func trasforma ogni elemento dell'RDD in una lista di elementi (anche vuota).
join(otherRDD)	Dato in input un altro RDD della forma $< K, V_2 >$, restituisce un nuovo RDD di tuple $< K, (V_1, V_2) >$ dove V_1 è il valore nell'RDD chiamante e V_2 è il valore che assume K in $otherRDD$.
map(func)	Restituisce un nuovo RDD di pari grandezza, applicando ad ogni elemento la funzione *func*.
reduce(func)	Aggrega gli elementi di un RDD applicando la funzione *func*, che riceve in input due tuple e ne restituisce una. L'operazione deve essere commutativa e associativa.
reduceByKey(func)	Aggrega gli elementi con la stessa chiave applicando la funzione $func : VxV => V$.

inizializzare sul *master* un contatore che può essere incrementato da qualsiasi *worker*. Per ulteriori dettagli è possibile consultare la documentazione ufficiale[1].

2.4 Configurazione adottata

Una delle problematiche maggiori di Spark è la configurazione, per la quale non esistono regole precise né tantomeno una configurazione ideale, ma bisogna arrivare tramite un processo di *trial-and-error* alla miglior configurazione possibile per una specifica applicazione.

Il cluster su cui sono stati effettuati gli esperimenti è dotato di 16 macchine con processore quad core Intel Nehalem i7-950, 16 GB di RAM e disco SSD.

In Spark generalmente è consigliato utilizzare un numero di partizioni dei dati uguale al doppio dei processori disponibili. Nel nostro caso, essendo processori dotati di Hyper Threading, abbiamo quindi impostato un parallelismo pari a 256.

Dagli esperimenti è risultato che distribuire i dati utilizzando una funzione *hash* funziona meglio per le reti sociali, mentre per le reti stradali (dove solitamente i ver-

[1] http://spark.apache.org/docs/latest/

tici adiacenti hanno chiavi "vicine") funziona meglio uno shuffle manager basato sul *sorting*.

I vari algoritmi sono stati implementati utilizzando le funzioni native di Spark poiché l'implementazione specifica per lo sviluppo di algoritmi di graph analytics, GraphX, è risultata meno performante. Infine, è risultato più efficiente utilizzare un unico RDD per rappresentare il grafo, associando ad ogni chiave (un vertice del grafo) la lista delle adiacenze di quel nodo.

Tabella 2.2: Configurazione di Spark

spark.driver.memory	4g
spark.executor.memory	14g
spark.driver.maxResultSize	2g
spark.default.parallelism	256
spark.rdd.compress	true
spark.shuffle.memoryFraction	0.4
spark.shuffle.blockTransferService	nio
spark.shuffle.manager	hash
spark.shuffle.consolidateFiles	true

Capitolo 3

Algoritmi fondamentali

3.1 Clustering

L'obiettivo primario di questa tesi è lo studio di tecniche di clustering applicate al calcolo di percorsi minimi. In questa sezione verranno quindi illustrati due diversi approcci: uno basato sulla scelta casuale dei nodi da cui generare i cluster ed uno basato su una scelta ordinata per *grado* del nodo.

Il clustering permette di ridurre la dimensione di un grafo, facendo crescere contemporaneamente da alcuni nodi selezionati delle palle, dette *cluster*, finché ogni nodo del grafo è associato ad un cluster. A quel punto è possibile ricavare il *grafo quoziente* (che definiremo in 4.2.1) ed utilizzarlo al posto del grafo originale negli algoritmi di nostro interesse, velocizzandone quindi l'esecuzione.

3.1.1 Randomized Clustering

Randomized Clustering[1] è un algoritmo MapReduce di clustering inizialmente pubblicato nel 2014 [9] da Matteo Ceccarello, Andrea Pietracaprina e Geppino Pucci, per grafi non diretti e non pesati, ma poi rielaborato [8] per funzionare anche nel caso pesato.

Si tratta del primo algoritmo parallelo in grado di approssimare il diametro di un grafo in un numero di iterazioni inferiore al diametro stesso, utilizzando al contempo spazio lineare. Seppur fornisca teoricamente un'approssimazione entro un fattore $O(\log^3 n)$, sperimentalmente si ottiene un rapporto di approssimazione entro $1,5$.

[1] Non è il nome scelto dagli autori. E' usato in questa sede per distinguerlo dal secondo clustering.

Pseudocodice e descrizione

Algoritmo 1 Randomized Clustering

Input: Grafo $G = (V, E)$ e t target del grafo quoziente
Output: Grafo processato

$C \leftarrow \emptyset$ // *cluster selezionati*
$V' \leftarrow \emptyset$ // *nodi coperti*
$batchDim \leftarrow \dfrac{t}{\log_2(|V|/t)}$
while $|V - V'| + |C| > t$ **do**
  Scegli ogni nodo in $V - V'$ come nuovo centro con probabilità $\dfrac{batchDim}{|V - V'|}$
  Aggiungi a C i nuovi centri selezionati
  Fai crescere in parallelo tutti i cluster in C finché non sono stati coperti almeno $|V - V'|/2$ nuovi nodi
  $V' \leftarrow$ nodi coperti dai cluster in C
end
$C \leftarrow C \cup (V - V')$
return C

L'algoritmo, invece di selezionare tutti i centri desiderati all'inizio, li seleziona per fasi per permettere sia di coprire le regioni sparse del grafo con nuovi cluster che di mantenere il raggio di ogni cluster, ovvero la distanza massima di un nodo dal proprio centro, limitato. Ad ogni iterazione si selezionano dei nuovi centri con probabilità via via crescente, quindi si fanno crescere in parallelo tutti i cluster selezionati fino a quel momento, finché non sono stati coperti almeno $|V - V'|/2$ nuovi nodi dall'inizio dell'iterazione.

Quando il numero di cluster selezionati (C) e il numero di nodi scoperti ($V - V'$) scende sotto la taglia target t, possiamo aggiungere tutti i nodi ancora scoperti come *singleton* cluster e restituire infine l'insieme dei cluster.

Per ulteriori dettagli è possibile consultare l'articolo originale [8].

3.1.2 Degree Clustering

Degree Clustering nasce invece durante il lavoro di questa tesi da alcune evidenze sperimentali emerse durante l'applicazione di clustering per il calcolo della centrality, argomento che sarà esaminato nel Capitolo 5. In particolare nelle reti sociali si è trovata una forte correlazione tra grado del nodo e importanza dello stesso all'interno della rete. Dato che i nodi ad alto grado tendono a formare degli *hub* naturali all'interno di

queste tipologie di grafi, si è deciso allora di provare a sfruttarli come unico set iniziale di centri da cui far crescere i cluster.

Questo metodo di selezione dei centri, non più randomizzato, ha due immediate ricadute pratiche:

1. Le regioni dense vengono coperte molto in fretta;

2. Le regioni sparse vengono coperte lentamente.

Selezionare tutti i centri all'inizio in ordine di grado decrescente ci permette infatti, nelle reti sociali, di approssimare meglio le distanze nelle regioni dense del grafo dove, essendo anche quelle più popolate, un piccolo errore nell'approssimazione delle distanze si può tradurre in un grande errore nel valore di centrality di un nodo. Tale fattore è invece meno d'impatto nel calcolo del diametro poiché, invece di misurare l'importanza relativa tra nodi, si limita ad approssimare una proprietà del grafo che è molto influenzata dalle regioni sparse dello stesso.

Per risolvere il secondo problema senza compromettere i vantaggi che derivano dal selezionare il maggior numero possibile di centri alla prima iterazione, è possibile inserire un numero adeguato di nuovi centri ogniqualvolta la velocità di copertura di nuovi nodi dovesse rallentare troppo.

Pseudocodice e descrizione

Possiamo subito notare che l'algoritmo è profondamente diverso da *Randomized Clustering*. Ci sono ad esempio quattro costanti (*J, K, L, M*) da fissare per regolare il funzionamento dell'algoritmo e bilanciare i due fattori precedentemente discussi: contenimento del numero di centri e velocità di copertura delle regioni sparse.

Innanzitutto, l'algoritmo seleziona tutti i t centri desiderati all'inizio (righe 4–6), quindi entra nel ciclo in cui i cluster vengono fatti crescere — ed eventualmente aggiunti — finché il grafo non è completamente coperto.

All'inizio di ogni iterazione viene memorizzato il numero di nodi scoperti. Dopo aver fatto crescere di un passo tutti i cluster presenti (ovvero un singolo scambio di messaggi tra nodi confinanti) viene effettuato un controllo (riga 12) sul numero di nuovi nodi coperti: se questo scende al di sotto di una quota determinata dal massimo tra una frazione dei nodi precedentemente scoperti e una frazione dei nodi del grafo[2], allora vengono aggiunti dei nuovi centri. Per la taglia dei nuovi centri da inserire, di nuovo, si individua il massimo tra una frazione dei nodi scoperti e una frazione della taglia t.

L'obiettivo è mantenere il numero finale di cluster il più vicino possibile al target iniziale, senza rallentare troppo il processo di clustering.

[2]E' importante considerare anche i nodi totali del grafo, poiché nelle fasi finali dell'algoritmo una frazione dei nodi scoperti rischia di diventare troppo piccola.

Regolando opportunamente i parametri è possibile far sì che nelle reti ad alto coefficiente di espansione, come quelle sociali, non vengano aggiunti nodi durante l'esecuzione dell'algoritmo, ma contemporaneamente nelle reti a bassa espansione, come quelle stradali, vengano aggiunti nodi in grado di mantenere il processo di copertura ad un ritmo adeguato. In quest'ultima tipologia di reti, infatti, aggiungere qualche centro in più di quelli inizialmente previsti non è un problema poiché, essendo reti sparse, l'esecuzione successiva di altri algoritmi basati sul grafo quoziente (come Dijkstra) non verrà eccessivamente rallentata. La scelta dei parametri varia anche in funzione del problema per cui si desidera applicare il clustering.

Algoritmo 2 Degree Clustering

Input: Grafo G e t target del grafo quoziente
Output: Grafo processato

1: $C \leftarrow \emptyset$ // *cluster selezionati*
2: $V' \leftarrow \emptyset$ // *nodi coperti*
3: $minUpdated \leftarrow |V| \cdot J$
4: Seleziona t nuovi centri in ordine di grado decrescente da V
5: Aggiungi a C i nuovi centri selezionati
6: $V' \leftarrow$ nodi coperti dai cluster in C
7: **while** $|V - V'| > 0$ **do**
8: $\quad uncovered \leftarrow |V - V'|$
9: $\quad$ Fai crescere in parallelo tutti i cluster in C di un'unità
10: $\quad V' \leftarrow$ nodi coperti dai cluster in C
11: $\quad$ **if** $(uncovered - |V - V'|) < \text{MAX}(uncovered \cdot K, minUpdated)$ **then**
12: $\quad\quad batchDim \leftarrow \text{MAX}(uncovered \cdot L, t \cdot M)$
13: $\quad\quad$ Seleziona $batchDim$ nuovi centri in ordine di grado decrescente da $V - V'$
14: $\quad\quad$ Aggiungi a C i nuovi centri selezionati
15: $\quad\quad V' \leftarrow$ nodi coperti dai cluster in C
16: $\quad$ **end**
17: **end**
18: **return** C

Per produrre i risultati descritti in questa tesi, in particolare, sono stati scelti i seguenti parametri: $J = 1/1500$, $K = 5/200$, $L = 2/1000$ e $M = 1/8$.

L'algoritmo è *pseudodeterministico* poiché, a parità di grado, i nodi non vengono scelti secondo un particolare ordine. In reti con un numero elevato di nodi con lo stesso grado (ad esempio le reti stradali, dove tutti i nodi hanno grado ridotto) la selezione di un certo numero di nodi di grado "elevato" potrebbe ridursi a una selezione casuale tra

molti nodi. Al contrario, in reti con una varianza elevata del grado, l'algoritmo tende a ripetere le stesse scelte da un'iterazione all'altra.

Inizialmente la selezione dei centri per grado era stata implementata sfruttando la selezione per fasi dell'algoritmo *Randomized Clustering*. Successivamente, si è visto che la selezione in un unico set iniziale produceva risultati sperimentali migliori sulle reti sociali (mentre sulle reti stradali la differenza è irrilevante, a causa della selezione quasi casuale di cui abbiamo appena discusso) e quindi si è arrivati alla forma finale dell'algoritmo qui descritta.

3.1.3 Confronto dei due algoritmi

Esaminiamo il comportamento dei due algoritmi eseguendoli su grafi diversi e variando al contempo la taglia del grafo quoziente (indicata tra parentesi).

Osserviamo che i tempi di esecuzione sono paragonabili: nelle reti stradali prevale leggermente Randomized Clustering, al contrario nelle reti sociali prevale Degree Clustering. A causa della politica scelta per la selezione dei centri, nel grafo quoziente ottenuto tramite Degree Clustering i lati sono più numerosi: circa il doppio nelle reti stradali e quasi un fattore 100 nelle reti sociali. Osserviamo ad esempio che nel grafo *livejournal* il numero di centri selezionati è comparabile, ma nel grafo quoziente di Degree Clustering sono presenti quasi 2 milioni di lati contro i 31 mila di Randomized Clustering.

Tabella 3.1: Confronto tempo e lati quoziente: Degree vs Randomized

Dataset	Tempo (s)		Nodi quoz. (m)		Lati quoz. (m)	
roadPA (5K)	52	50	6	4.8	30	18
roadPA (10K)	50	37	11.2	9.5	57.2	36.2
roadPA (15K)	45	32	16.3	12.7	84	47
roadTX (5K)	65	60	6.4	4.8	30.5	16.3
roadTX (10K)	58	48	12	9.2	58	33
roadTX (15K)	48	45	17	14	84	52
orkut (2K)	34	40	2	1.1	3619	37
orkut (5K)	44	47	5	1.2	13840	227
livejournal (2K)	30	33	2	1.8	1982	31
livejournal (5K)	29	34	5.1	3.1	5854	160

Possiamo osservare (tabelle 3.2 e 3.3) che Degree Clustering è più efficace di Randomized Clustering nel mantenere i cluster di dimensione limitata, sia nel raggio che nella taglia (numero di nodi appartenenti al cluster). Taglia e raggio dei cluster hanno infatti media e mediana simili: i valori sono quindi ben distribuiti senza particolari eccessi. Tali valori sono inoltre sempre minori in Degree rispetto a Randomized, tranne

nel grafo *orkut* che è un caso particolare essendo estremamente denso. In Randomized Clustering sembra infatti esserci una rilevante disparità tra cluster piccoli e cluster grandi, con un numero elevato di cluster piccoli e alcuni cluster di dimensione molto elevata.

Tabella 3.2: Confronto taglia cluster: Degree vs Randomized

Dataset	Media		Mediana		Max	
roadPA (5K)	178	225	119	9	1900	5500
roadPA (10K)	96	114	69	8	950	2800
roadPA (15K)	66	85	50	6	700	1800
roadTX (5K)	206	290	109	10	2600	7200
roadTX (10K)	115	148	77	8	1600	4300
roadTX (15K)	79	95	57	7	1200	2600
orkut (2K)	1536	2900	338	1	347K	140K
orkut (5K)	613	2526	105	1	143K	62K
livejournal (2K)	1824	3000	270	1	114K	500K
livejournal (5K)	772	1279	126	1	67K	250K

Tabella 3.3: Confronto raggi cluster: Degree vs Randomized

Dataset	Media		Mediana		Max	
roadPA (5K)	12	7.8	13	4	30	40
roadPA (10K)	9.6	6.3	10	3	23	28
roadPA (15K)	8.5	5.7	9	3	20	24
roadTX (5K)	12	8.5	13	4	37	47
roadTX (10K)	10.2	7	11	3	28	33
roadTX (15K)	9	6	9	3	23	27
orkut (2K)	2.9	0.7	3	0	5	4
orkut (5K)	2.4	1.6	2	0	4	4
livejournal (2K)	4	0.8	4	0	7	7
livejournal (5K)	3.6	1.4	4	0	7	7

3.2 Probabilistic Counting: HyperBall

Per indagare le proprietà di un grafo non pesato è possibile utilizzare un secondo metodo di computazione, sfruttando i cosiddetti *contatori probabilistici*. L'idea è quella di assegnare ad ogni nodo del grafo un set in cui memorizzare tutti i nodi presenti entro una determinata distanza. Non possiamo però memorizzare gli identificatori

dei nodi in modo diretto, perché avremmo bisogno di spazio lineare per ogni nodo del grafo e quindi spazio quadratico nel complesso. Utilizzando invece dei contatori probabilistici, possiamo contare elementi appartenenti a insiemi di cardinalità molto ampia usando solo una piccolissima frazione dello spazio che richiederebbe invece un contatore lineare.

3.2.1 HyperLogLog Counter

Introdotti nel 2007 da Flajolet e altri [12], i contatori *HyperLogLog* sono a tutt'oggi la soluzione di riferimento per il conteggio probabilistico di elementi in un insieme. Permettono infatti di contare elementi appartenenti ad insiemi di cardinalità n utilizzando spazio nell'ordine di $\log \log n$, da cui il nome.

Per la descrizione che segue, supponiamo di avere una funzione di hash $h(x) : M \rightarrow [0, 1, .., 2^m - 1]$ e una funzione $\rho(y)$ che data una stringa di bit restituisce l'indice del bit 1 meno significativo presente nella stringa ($\rho(0001...) = 4$). Per ogni elemento x dell'insieme M con la cardinalità da calcolare applichiamo la funzione di hash, ottenendo $y = h(x)$. Suddividiamo quindi i valori di hash in $m = 2^b$ sottoinsiemi, $M^1...M^m$, in base ai primi b bit meno significativi del valore y.

Il valore che memorizzeremo per ogni registro R_j sarà infine

$$R_j = \max_{y \in M^j} \rho(y) \tag{3.1}$$

Algoritmo 3 HyperLogLog

Input: Elemento $x \in M$ da inserire e set $R_1...R_m$ di registri

$h \leftarrow \text{HASH}(x)$
$j \leftarrow 1 + < h_1...h_b >_2$
$R_j \leftarrow \text{MAX}(R_j, \rho(h_{b+1}h_{b+2}...))$

L'algoritmo sfrutta l'idea che ogni sottoinsieme M^j, idealmente, conterrà n/m elementi. Allora il valore del registro R_j ci aspettiamo che sia vicino a $\log_2(n/m)$, perciò la media armonica del valore 2^{R_j} dovrebbe essere nell'ordine di n/m. Moltiplicando quindi la media armonica per m ed un fattore correttivo determinato dagli autori, otteniamo infine la seguente stima di n

$$N = \frac{\alpha_m m^2}{\sum_{j=1}^{m} 2^{-R_j}} \qquad \text{dove} \qquad \alpha_m = \left(m \int_0^\infty \left(log_2 \frac{2+u}{1+u} \right)^m du \right)^{-1} \tag{3.2}$$

L'analisi operata dagli autori stabilisce che i contatori HyperLogLog abbiano un errore standard di $1,04/\sqrt{m}$, che ci permette ad esempio di stimare cardinalità nell'or-

dine di 10^9 con un errore standard del 2% utilizzando solo $m = 2048$ registri (circa 1,5 kilobyte di memoria).

Per ulteriori dettagli è possibile consultare l'articolo originale [12].

3.2.2 Pseudocodice e descrizione

Definiamo quindi l'algoritmo HyperBall, che ci permetterà di determinare, per ogni nodo del grafo, il numero di vertici ad una determinata distanza da esso.

Algoritmo 4 HyperBall

Input: G grafo da processare e $b = \log_2 m$, logaritmo registri
Output: Grafo processato

```
1:  for each vertex v in G do
2:      v ← new HyperLogLogCounter(b)
3:  end
4:  while esistono contatori aggiornati do
5:      for each vertex v in G do
6:          if contatore di v aggiornato then
7:              Invia una copia del contatore a tutti i vicini di v
8:          end
9:      end
10:     for each vertex v in G do
11:         if v ha ricevuto almeno un contatore then
12:             counter ← counter ∪ contatori ricevuti
13:             Fai qualcosa con il contatore aggiornato
14:         end
15:     end
16: end
17: return G
```

L'algoritmo (righe 1–3) crea per ogni nodo del grafo un nuovo contatore (dotati della stessa funzione di hash). Successivamente entra nel ciclo principale (righe 4–16), che si ripete finché almeno un contatore viene aggiornato (ovvero aumenta la propria cardinalità). All'inizio dell'iterazione i-esima ogni nodo, se all'iterazione precedente si è aggiornato, invia ai propri vicini una copia del contatore. Successivamente, tutti i contatori destinati allo stesso nodo vengono inizialmente uniti tra loro, quindi uniti al contatore del nodo destinatario.

L'unione tra contatori diversi avviene semplicemente effettuando l'operazione binaria *OR* tra i registri corrispondenti.

Memorizzando la cardinalità del proprio contatore prima e dopo l'istruzione 12, è possibile ottenere il numero approssimato di nodi a distanza i. Come vedremo nel Capitolo 5, è possibile sfruttare tale informazione per calcolare la *centrality* di ogni nodo.

Capitolo 4

Diametro

4.1 HyperBall

L'utilizzo di contatori HyperLogLog per la stima del diametro è un problema ampiamente studiato e ormai consolidato in letteratura. La prima implementazione efficiente è stata *HyperANF*, ad opera di Boldi e Vigna [2], sviluppata in Java ed ottimizzata per essere eseguita su una singola workstation multiprocessore dotata di memoria condivisa.

Una prima implementazione su Apache Spark è stata invece sviluppata in [7], successivamente è stata oggetto di un lavoro approfondito in [5]. L'argomento è stato tuttavia ripreso in questo lavoro poiché c'erano grossi margini di miglioramento rispetto alle soluzioni già esistenti.

Miglioramenti rispetto alle implementazioni esistenti

L'autore [5] ha scelto di sfruttare l'implementazione dei contatori HyperLogLog sviluppata per HyperANF, associando quindi ad ogni vertice del grafo un diverso oggetto e creandone delle copie ad ogni scambio di contatori.

L'oggetto sviluppato da Vigna e Boldi, oltre ai registri, contiene altre variabili utilizzate per le funzioni di unione e stima della cardinalità. Queste variabili sono però uguali in tutti gli oggetti. Inoltre, tali oggetti non sono stati pensati per un utilizzo distribuito, quindi ogni volta che si vuole effettuare l'unione di due diversi contatori non viene sfruttato lo spazio già allocato ma bisogna creare diversi nuovi oggetti. In Spark ogni oggetto prima di essere trasmesso deve essere serializzato, quindi all'aumentare del volume dei dati segue un aumento più che lineare dei tempi di comunicazione. La creazione di numerosi oggetti intermedi, infine, obbliga il *Garbage Collector* della JVM[1] a intervenire più frequentemente.

[1] Java Virtual Machine

Un risultato importante raggiunto in questa tesi è stato quindi quello d'aver sviluppato un'implementazione *ad-hoc* di HyperLogLog adatta all'utilizzo su Spark, che permetta cioè uno scambio efficiente di contatori tra nodi.
Invece di scambiare e unire i registri attraverso l'uso di pesanti oggetti innestati, infatti, è possibile utilizzare dei semplici array di interi. Inoltre, invece di associare ad ogni nodo del grafo un diverso oggetto, è sufficiente mantenere memorizzati solamente i registri tramite un array. In tal modo riduciamo sia il volume di dati "fissi", che quelli temporanei. Per poter effettuare poi le operazioni di unione tra registri è possibile utilizzare delle funzioni ottimizzate che limitano al massimo utilizzo di memoria e creazione di nuovi oggetti.

Questi accorgimenti hanno portato ad un miglioramento della velocità d'esecuzione, in alcuni casi, di un fattore 10^2. Per ulteriori dettagli tecnici è possibile consultare il codice sorgente[3].

4.1.1 Algoritmo

L'algoritmo è lo stesso illustrato nella sezione 3.2.2.
L'idea alla base del suo utilizzo per il calcolo del diametro è la seguente:

1. All'inizio dell'algoritmo ogni vertice possiede come unica informazione l'identificatore di sé stesso.
2. All'inizio dell'iterazione i-esima, $i \geqslant 1$, ogni vertice sarà a conoscenza dei nodi entro distanza $i - 1$ da esso e invierà ai nodi confinanti il proprio contatore.
3. All'iterazione i-esima ogni vertice v riceverà i contatori dei propri vicini. Facendone l'unione insiemistica, si otterrà infine l'insieme dei nodi entro distanza i da v.
4. Se all'iterazione i-esima, nel vertice v, l'unione degli insiemi ricevuti non ha portato alla scoperta di nuovi nodi rispetto a quelli ottenuti all'iterazione $i - 1$, significa che non esistono nodi a distanza maggiore di $i - 1$ da v. In tal caso, il raggio $r(v)$ di v è uguale a $i - 1$.
5. Se all'iterazione i-esima nessun contatore viene aggiornato, significa che il raggio massimo del grafo, e quindi il diametro, è pari a $i - 1$.

Correttezza. La prova è banale. Se esistesse un nodo a distanza i da v, infatti, questo sarebbe giunto a v attraverso il percorso minimo (di lunghezza i) dopo i iterazioni dell'algoritmo. Altrettanto banalmente si dimostra che non può esistere un nodo x a

[2] Purtroppo non è possibile riportare un confronto sistematico a causa di cambiamenti hardware della piattaforma.

[3] https://github.com/gparigibini

distanza $i + 1$. In tal caso, infatti, detto y il nodo che precede x nel percorso minimo da x a v, all'iterazione i il nodo v avrebbe ricevuto y. Iterando il ragionamento si può ottenere il risultato.

4.1.2 Risultati e considerazioni

Prima di esaminare le prestazioni dell'algoritmo, facciamo una breve considerazione sull'applicabilità dello stesso.

L'algoritmo ha un numero di iterazioni pari al diametro + 1. Su grafi dal grande diametro, come le reti stradali, l'algoritmo dovrà quindi eseguire centinaia di iterazioni prima di terminare, impiegandoci diverse decine di minuti, il che rende l'approccio non praticabile su reti di questo tipo.
Inoltre, seppur lo spazio occupato dai registri sia nell'ordine delle centinaia di byte, esso può diventare facilmente un problema nelle reti dense dove i lati sono circa il quadrato dei nodi, poiché ad ogni iterazione vengono generati tanti contatori quanti sono i lati.

Infine, ma non meno importante, questo approccio non funziona nel caso di grafi pesati.

Tabella 4.1: Risultati HyperBall
$\log_2 reg$: diametro (tempo)

Dataset	Basso $\log_2 reg$	Alto $\log_2 reg$
dblp	8: 19 (45)	10: 20 (55)
livejournal	6: 15 (109)	8: 15 (139)
orkut	6: 8 (133)	8: 8 (255)
roadPA	8: 777 (2090)	•
roadTX	8: 1052 (3287)	•
roadCA	8: 850 (3002)	•

In tabella 4.1 sono rappresentate alcune esecuzioni dell'algoritmo su 6 grafi, utilizzando (dove sensato) numeri diversi di registri. Ad esempio, nella prima cella (*8: 19 (45)*) per il grafo *dblp* abbiamo eseguito HyperBall con $reg = 2^8$ registri e abbiamo ottenuto una stima del diametro uguale a 19, impiegandoci 45 secondi.

Per le reti sociali abbiamo eseguito l'algoritmo con un numero "basso" di registri e con un numero "alto", in relazione alla taglia del grafo. Per le reti stradali invece, a causa del tempo impiegato con un numero "basso" di registri, non ha avuto senso replicare l'esperimento con un numero maggiore.

Come ci aspettavamo, sulle reti stradali anche se le singole iterazioni sono veloci, il loro numero elevato rende l'approccio impraticabile. Sulle reti sociali di bassa cardinalità l'algoritmo è invece competitivo, riuscendo a trovare una buona stima del

diametro in poche decine di secondi. Sulle reti sociali di cardinalità alta, invece, il peso dei numerosi messaggi in circolazione ad ogni iterazione rende l'algoritmo meno competitivo rispetto alle altre soluzioni, come avremo modo di esaminare alla sezione 4.2.3.

4.2 Degree Clustering

4.2.1 Approccio

Applicando l'algoritmo 2 (o 1) ad un grafo G otteniamo C, un grafo con gli stessi archi e vertici dell'originale, dove però in ogni nodo è memorizzata una coppia che indica il centro del cluster di appartenenza e la distanza da esso. L'idea è di creare a partire da C un **grafo quoziente** Q in cui un cluster sia rappresentato da un nodo e i cluster confinanti, cioè collegati da almeno un lato, siano collegati da un lato di peso adeguato nel grafo quoziente.

Definiamo allora l'algoritmo 5, che accetta in input C e restituisce il grafo quoziente Q. Osserviamo che vengono esaminati tutti i lati del grafo clusterizzato e, se gli estremi di un arco appartengono a due cluster differenti, viene creato nel grafo quoziente un lato tra i centri dei due cluster, di peso uguale alla somma delle distanze dei due nodi dai rispettivi centri più il peso dell'arco sotto esame.

Algoritmo 5 Estrazione grafo quoziente

Input: C, grafo clusterizzato
Output: Grafo quoziente

```
Q ← ∅
for each lato (u,v) in C do
    c_u ← centro di u
    c_v ← centro di v
    if c_u ≠ c_v then
        d_u ← distanza da u a c_u
        d_v ← distanza da v a c_v
        Aggiungi lato (c_u, c_v) di peso (d_u + d_v + 1) a Q
    end
end
Per ogni coppia (u,v) in Q collegata da più lati mantieni quello col peso minore
return Q
```

Definizione 4.1. Il *grafo quoziente* di un grafo G processato da un algoritmo di clustering è il grafo i cui nodi corrispondono ai cluster in G e i cui lati connettono cluster che in G contengono nodi adiacenti.

Applicando un algoritmo di *All-Pairs-Shortest-Paths*, infine, è possibile ottenere una stima della vera distanza tra i centri dei cluster. Con algoritmo *APSP* si intende un algoritmo in grado di calcolare la distanza da ciascun nodo verso ogni altro nodo della rete. In questa sede, tale risultato viene raggiunto attraverso *n* applicazioni di *Dijkstra*, a partire da ogni nodo del grafo. Il cammino minimo di lunghezza massima ottenuto con questa procedura, però, non è indicativo del vero diametro del grafo poiché non abbiamo preso in considerazione i *raggi* dei cluster. Definiamo allora l'algoritmo 6, in grado di ottenere un *upper bound* del vero diametro, considerando sia la distanza tra centri che il loro raggio.

Algoritmo 6 Approssimazione Diametro

Input: C, grafo clusterizzato
Output: Diametro

$Q \leftarrow$ Estrai grafo quoziente da C
$rads \leftarrow$ Per ogni cluster in C trova il nodo più distante // *mappa (clusterCenter, radius)*
$dists \leftarrow \text{APSP}(Q)$ // *matrice distanze*
$max \leftarrow 0$
for each $(src, dst, distance)$ **in** $dists$ **do**
    $max \leftarrow \text{MAX}(max, rads(src) + distance + rads(dst))$
end
return max

4.2.2 Risultati e confronto con Randomized Clustering

Confrontiamo adesso i risultati sperimentali ottenuti con Degree Clustering rispetto a quelli ottenuti tramite Randomized Clustering con le tabelle 4.2 – 4.5. Per far ciò, esaminiamo la stima del diametro ottenuta per sei grafi, 3 reti sociali e 3 reti stradali, per diverse taglie del grafo quoziente, riportando inoltre anche il tempo impiegato dall'algoritmo. Come diametro di riferimento useremo la stima ottenuta grazie all'algoritmo *Double-SSSP*, che defineremo formalmente al paragrafo 4.2.3.

Le stime ottenute sono indicate con un intervallo con i valori minimo e massimo ottenuti tramite diverse iterazioni (da un minimo di 5 ad un massimo di 10), allegando infine tra parentesi il tempo mediano in secondi dell'intera procedura (clustering + APSP).

E' bene ricordare, come spiegato alla fine della sezione 3.1.2, che in certe situazioni anche l'algoritmo Degree Clustering si riduce ad una scelta pseudocasuale dei centri, quindi avremo, anche in questo caso, dei risultati diversi da un'iterazione all'altra, seppur in misura inferiore rispetto a Randomized Clustering.

Tabella 4.2: Taglia 1.000

Dataset	Diametro	Rand. Clust. (t)	Degree Clust. (t)
dblp	23	[34-40] (14)	25-26 (23)
livejournal	21	[29-34] (42)	23-24 (40)
orkut	9	[17-22] (50)	13 (37)
roadPA	794	[1194-1283] (64)	[1090-1130] (64)
roadTX	1064	•	[1511-1559] (94)
roadCA	852	•	[1354-1430] (141)

Tabella 4.3: Taglia 5.000

Dataset	Diametro	Rand. Clust. (t)	Degree Clust. (t)
dblp	23	[36-45] (12)	24-25 (38)
livejournal	21	[30-35] (40)	23-24 (127)
orkut	9	[18-22] (45)	13 (220)
roadPA	794	[1194-1318] (60)	[1106-1130] (62)
roadTX	1064	[1505-1602] (77)	[1544-1562] (69)
roadCA	852	•	[1412-1467] (93)

Tabella 4.4: Taglia 10.000

Dataset	Diametro	Rand. Clust. (t)	Degree Clust. (t)
dblp	23	[36-42] (13)	24-25 (44)
livejournal	21	[30-35] (49)	24 (513)
orkut	9	[19-23] (46)	14 (836)
roadPA	794	[1270-1325] (45)	[1076-1089] (58)
roadTX	1064	[1572-1620] (51)	[1506-1513] (67)
roadCA	852	•	[1378-1425] (90)

Tabella 4.5: Taglia 15.000

Dataset	Diametro	Rand. Clust. (t)	Degree Clust. (t)
dblp	23	[36-42] (13)	24 (51)
livejournal	21	[30-35] (49)	•
orkut	9	[19-23] (46)	•
roadPA	794	[1251-1342] (42)	[1033-1093] (41)
roadTX	1064	[1548-1636] (54)	[1435-1490] (54)
roadCA	852	[1502-1523] (70)	[1341-1388] (73)

Reti stradali Osserviamo che la stima del diametro nelle reti stradali, usando l'algoritmo Randomized Clustering, sembra essere invariante alla taglia del grafo quoziente. Il tempo invece, all'aumentare della taglia, diminuisce lievemente ma non ci sono differenze significative[4].

Guardando, invece, l'ultima colonna con i risultati ottenuti tramite Degree Clustering, è possibile notare che la stima del diametro è leggermente migliore rispetto al concorrente mantenendo tempi di esecuzione simili. Per quanto riguarda le taglie molto piccole, però, questa competitività nel tempo di clustering è ottenuta tramite l'aggiunta automatica di nuovi centri nelle prime fasi: le reti stradali con taglia molto piccola, infatti, hanno un tasso di copertura troppo lento usando un solo set iniziale di nodi, quindi l'algoritmo comincia ad aggiungere nuovi centri che portano la taglia finale a circa 3000 cluster invece dei 1000 desiderati. Per quanto riguarda le taglie più elevate, invece, l'algoritmo non ha bisogno di aggiungere forzatamente dei nuovi centri durante l'esecuzione e arriva a completamento con una taglia finale della grandezza desiderata.

Reti sociali Se nelle reti stradali il vantaggio di Degree Clustering su Randomized Clustering è risibile, nelle reti sociali possiamo osservare un comportamento molto differente.
Vediamo infatti che in queste reti il diametro ottenuto tramite Degree Clustering è decisamente migliore di quello ottenuto tramite Randomized Clustering, producendo infatti una stima molto vicina al vero diametro, per qualsiasi taglia target.
Possiamo però osservare che il tempo di esecuzione nelle reti dense (come *Orkut*) cresce molto velocemente al salire della taglia: tale fenomeno è dovuto al fatto che, selezionando i nodi per grado, si formano dei cluster fortemente interconnessi che vanno a creare un grafo quoziente quasi *completo* (in cui ogni cluster ha un collegamento diretto con ogni altro cluster) e quindi, pur essendo la fase di clustering molto veloce in tutti i casi, cresce rapidamente il tempo richiesto da Dijkstra per il calcolo dei cammini

[4]Non è stato possibile applicare Randomized Clustering ad alcune reti stradali con target basso poiché l'algoritmo non è stato originariamente sviluppato per funzionare con grafi disconnessi.

minimi. Infatti, non sono stati ottenuti i dati per quei due grafi con taglia 15.000 perché troppo dispendiosi e inutili ai fini dell'analisi.

Conclusioni Possiamo vedere che esiste quindi un pattern di applicazione per quanto riguarda i due algoritmi: se Randomized Clustering ha risultati quasi invarianti sia nel tempo che nella qualità della stima per entrambe le tipologie di grafi, Degree Clustering esibisce invece comportamenti molto differenti a seconda della tipologia di grafo.

Per le reti sociali, si può infatti ottenere un'ottima stima del diametro in poco tempo usando una taglia target bassa. Nelle reti stradali, al contrario, il tempo e la qualità dell'approssimazione migliorano al crescere della taglia.

4.2.3 Confronto con altri algoritmi

Come anticipato, per la stima del vero diametro è stato utilizzato l'algoritmo *Double-SSSP* [5]. Confronteremo quindi i migliori risultati ottenuti con approcci basati su clustering con gli algoritmi Double-SSSP e HyperBall.

Double-SSSP Definiamo l'algoritmo Double-SSSP in questo modo: prendiamo un nodo x a caso nella rete (nel nostro caso selezioniamo il nodo di grado maggiore), troviamo il nodo y più distante di esso tramite un'esecuzione di Dijkstra, quindi eseguiamo una nuova istanza di Dijkstra a partire da y. Il cammino minimo più lungo trovato, infine, sarà la nostra stima del diametro. L'implementazione distribuita di tale algoritmo richiede al più un numero di iterazioni pari al doppio del *costo* del diametro. In un grafo non pesato, come quelli qui testati, il costo del diametro è equivalente al numero di archi. Non è ovviamente un approccio praticabile sui grafi ad alto diametro, ma diventa molto competitivo sui grafi ad alto coefficiente d'espansione producendo al contempo un'ottima stima del diametro. Ovviamente ne è stata data una implementazione distribuita, funzionante attraverso lo scambio di messaggi tra le varie macchine.

Seppur tale stima rappresenti un *lower bound* al vero diametro, lo useremo come valore di riferimento poiché il vero diametro è a noi sconosciuto. [6]

Tabelle Per le conclusioni ottenute precedentemente, come risultati rappresentativi per i due algoritmi di clustering sono stati scelti quelli ottenuti con taglia 1000 per le reti sociali e quelli di taglia 15.000 per le reti stradali. Per l'algoritmo HyperBall

[5] Single Source Shortest Paths

[6] I ricercatori del sito di Stanford da cui sono stati prelevati i dataset, infatti, forniscono una stima del diametro che è sistematicamente minore del lower bound calcolato con questo algoritmo, quindi errata.

Tabella 4.6: Confronto dei diametri ottenuti

Dataset	Double-SSSP	HyperBall	Rand. Clust.	Degree Clust.
dblp	23	19-20	[34-40]	25-26
livejournal	21	15	[29-34]	23-24
orkut	9	8	[17-22]	13
roadPA	794	777	[1251-1342]	[1033-1093]
roadTX	1064	1052	[1548-1636]	[1435-1490]
roadCA	852	850	[1502-1523]	[1341-1388]

Tabella 4.7: Confronto dei tempi (secondi)

Dataset	Double-SSSP	HyperBall	Rand. Clust.	Degree Clust.
dblp	42	45	14	23
livejournal	55	109	50	40
orkut	39	133	42	37
roadPA	1347	2090	42	41
roadTX	1937	3287	54	54
roadCA	1510	3002	70	73

sono invece state considerate le iterazioni più veloci disponibili con basso numero di registri.

I tempi indicati in Tabella 4.7 sono invece stati calcolati, come per le tabelle precedenti, prendendo la mediana di più iterazioni, scartando così eventuali casi anomali dovuti a dinamiche interne al sistema.

Conclusioni Con queste due ultime tabelle riassuntive, possiamo vedere facilmente che Degree Clustering è un algoritmo assolutamente competitivo sia nelle reti stradali che nelle reti sociali.

Nelle reti sociali otteniamo infatti una stima superiore di soli 2-3 punti rispetto al presunto diametro in un tempo inferiore all'algoritmo di riferimento. Da alcune evidenze qui non riportate con ulteriori grafi, è risultato che tale competitività con Double-SSSP, al crescere della taglia e del diametro della rete sociale in input, è destinata a migliorare.
Si tenga inoltre presente che il diametro nel grafo quoziente è calcolato applicando un APSP: possiamo ottenere in certi casi una significativa compressione del tempo finale se, invece di calcolare i cammini minimi a partire da ogni nodo del grafo quoziente, applichiamo lo stesso procedimento descritto per Double-SSSP. In tal caso, però, perdiamo la garanzia teorica di ottenere un *upper bound* al vero diametro, ma nella pratica

la stima ottenuta in tal modo differisce da quella calcolata tramite APSP di pochi punti e, anzi, sono spesso uguali.

Nelle reti stradali i vantaggi del Degree Clustering sul Randomized Clustering, seppur presenti, sono meno significativi. Gli approcci basati su clustering sono ad oggi l'unico approccio pratico per indagare le proprietà di grafi ad alto diametro, quindi assume particolare importanza la qualità della stima ottenuta in questo tipo di grafi. Com'è possibile vedere in Tabella 4.6, nel migliore dei casi (*roadPA*) siamo in grado di ottenere una stima con un errore relativo del 30% rispetto al (presunto) diametro, errore abbastanza elevato da rendere potenzialmente inutile la stima in alcune applicazioni. Vediamo allora se e come è possibile ottenere una buona approssimazione anche su questo tipo di reti.

4.3 Multicoloring: oltre il semplice clustering

4.3.1 Motivazioni

Come abbiamo visto, *Degree Clustering* è in grado di ottenere una buona stima del diametro, in grafi sparsi, in una frazione del tempo richiesto dagli altri approcci. In grafi densi invece, seppur l'approccio basato su clustering rimanga competitivo sia come tempi sia come qualità della stima, l'approccio *Double-SSSP* riesce ad ottenere una stima leggermente migliore del diametro in tempi paragonabili a quelli richiesti dal clustering.

Se quindi consideriamo come target primario di questo approccio i grafi ad alto diametro, potremmo chiederci come migliorarne l'errore relativo della stima, dove pochi punti percentuali si traducono in una grande differenza assoluta, senza dilatare troppo il tempo di esecuzione.
Per calcolare il cammino minimo tra due generici nodi x e y del grafo originale, infatti, nel grafo quoziente siamo obbligati a passare attraverso i centri dei vari cluster che separano i due nodi. Ogni volta che "attraversiamo" un cluster paghiamo quindi un errore, che può essere anche significativo.

In generale, la taglia del grafo quoziente ha poca influenza sull'errore finale. Usando un clustering *fine*, infatti, riduciamo l'errore commesso attraversando un cluster, ma lo ripetiamo più volte. Utilizzando un clustering *coarse*, invece, il cammino minimo attraversa meno cluster ma questi avranno anche un errore più elevato rispetto al caso precedente.

Se vogliamo quindi migliorare la stima dei cammini minimi, ogni cluster dovrà acquisire informazioni anche sui cluster vicini non immediatamente confinanti, quindi dobbiamo abbandonare l'idea di avere cluster completamente disgiunti.

Dijkstra a distanza limitata

Con l'implementazione attuale, nel momento in cui il grafo è completamente coperto — ovvero tutti i nodi appartengono ad un cluster — l'algoritmo termina. Si potrebbe pensare allora di modificare l'algoritmo in modo che i messaggi con le distanze da ogni centro continuino a propagarsi anche se il grafo è già coperto, fino a raggiungere una determinata distanza massima. In tal modo otterremo la distanza esatta di tutti i nodi entro il raggio massimo da noi indicato, per il centro di ogni cluster.

Purtroppo un approccio del genere non è praticabile poiché, ad esempio, nelle reti sociali entro pochi passi è possibile raggiungere la maggior parte dei nodi presenti nella rete, quindi il numero di messaggi scambiati, anche con un raggio massimo molto basso, esplode velocemente.

Dijkstra a memoria limitata

Invece di limitare la distanza massima percorribile da un messaggio, potremmo pensare di limitare la quantità di distanze memorizzabili nella mappa di ogni nodo. In tal modo, nel momento in cui un nodo con la mappa già piena riceve un messaggio da un nuovo centro, esso non verrebbe ritrasmesso. Se invece la mappa di un nodo non è piena, oppure se il messaggio contiene una distanza minore da un centro già memorizzato, il nodo aggiorna la distanza e la ritrasmette come ci aspetteremmo che faccia. L'idea alla base è quella che i nodi nelle zone dense del grafo, dove i cluster sono già densamente connessi, si satureranno in fretta, mentre i messaggi nelle zone sparse del grafo, dove i cammini minimi vengono approssimati peggio, avranno modo di percorrere distanze più lunghe prima di saturare la memoria dei nodi.

Purtroppo anche tale approccio non è praticabile. L'algoritmo, infatti, termina quando tutti i nodi raggiungono la memoria massima indicata e quindi non vengono più generati nuovi messaggi da trasmettere. In grafi molto sparsi come le reti stradali, però, affinché venga saturata la memoria dei nodi periferici, potrebbero essere necessarie molte iterazioni dell'algoritmo. Indicando un numero troppo limitato di distanze memorizzabili per mitigare questo effetto, invece, annulleremmo i benefici di questo approccio.

Vale la pena notare che il clustering semplice, cioè a cluster disgiunti, è ottenibile con questo algoritmo semplicemente assegnando un solo registro in cui memorizzare le distanze, ovvero solo quella dal proprio centro.

Un semplice compromesso

Possiamo allora pensare di prendere il meglio dei due approcci, cioè il controllo sul numero massimo di iterazioni dell'algoritmo e il controllo sul numero di messaggi scambiabili, e fonderli in un unico algoritmo. Assegniamo allora ad ogni nodo una memoria limitata in cui poter memorizzare i messaggi ricevuti e, allo stesso tempo, evitiamo che

l'algoritmo esegua troppe iterazioni per arrivare a saturare la memoria di nodi molto periferici, interrompendolo non appena tutto il grafo viene coperto.

In tal modo avremo un numero di iterazioni pari a quelle che avremmo avuto col clustering semplice e, nel mentre, ci assicuriamo che il tempo richiesto per ogni iterazione non esploda, limitando il numero di registri.

4.3.2 Pseudocodice e descrizione

Il nome *Multicoloring* deriva proprio dal fatto che, se immaginiamo di colorare tutti i nodi appartenenti allo stesso cluster di un determinato colore, allora un nodo viene "colorato" più volte, tante quante sono i registri disponibili.

L'algoritmo di clustering è lo stesso illustrato al Capitolo 3 con la differenza che ciascun nodo, invece di memorizzare e trasmettere solo la distanza dal proprio centro, memorizza e ritrasmette tutte le distanze che riceve fino al riempimento dei registri assegnati. A quel punto, eventuali messaggi ricevuti vengono semplicemente ignorati. Definiamo questo algoritmo con la funzione $DegreeClustering(G, t, k)$, dove G è il grafo da processare, t è la taglia del grafo quoziente, k è il numero di registri da utilizzare.

Lo pseudocodice è banale, l'algoritmo vero e proprio (particolarmente complicato per ottimizzarlo al meglio e non interessante in questa sede) è invece visionabile all'indirizzo[7].

Più interessante è, invece, esaminare come viene costruito il grafo quoziente a partire dal grafo processato.

Algoritmo 7 Multicoloring + Grafo quoziente

Input: G, grafo da processare. t, taglia grafo quoziente. k, numero massimo di registri
Output: Grafo quoziente

```
1:  C ← DegreeClustering(G, t, k)
2:  Q ← Estrai grafo quoziente da C
3:  for each vertex v in C do
4:      for each (src, dist) in v.map do
5:          Aggiungi lato (src, c_v) di peso (dist + d_v) a Q
6:      end
7:  end
8:  Per ogni coppia (u,v) in Q collegata da più lati
9:     mantieni quello col peso minore
10: return Q
```

[7] https://github.com/gparigibini

Come già illustrato, applichiamo intanto l'algoritmo 5 per estrarre i lati a cavallo di cluster immediatamente confinanti. Successivamente processiamo tutti i vertici del grafo: ciascun vertice v potrà contenere al più K coppie *(src, dist)*, dove *src* è un centro di un cluster e *dist* è ovviamente la distanza da *src* a v. Creiamo allora un lato tra il centro c_v di v e il centro *src*, avendo la distanza contenuta nel messaggio e la distanza del nodo dal proprio centro. Infine, siccome ogni coppia di centri sarà verosimilmente collegata da molti lati diversi, manteniamo solo quelli dal peso minore che saranno effettivamente utilizzati per il calcolo dei cammini minimi.

4.3.3 Risultati sperimentali

Esaminiamo nel dettaglio il comportamento di *Multicoloring* sul grafo *roadPA*, operando su clustering di taglia 10 mila e 15 mila. Per ogni riga della tabella sono elencati il massimo numero di registri utilizzabili da un nodo, il tempo richiesto dal solo *clustering* (tempo incomprimibile) e il tempo finale, ovvero *clustering + APSP*.

Se non siamo interessati ad ottenere un provabile upper bound, infatti, invece di utilizzare $|clustering|$-volte Dijkstra (per ogni nodo del grafo quoziente) possiamo limitarci ad applicarlo solo due volte (come in Double-SSSP) in locale sul processo *driver*, ed ottenere infine una stima che di solito differisce da quella ottenuta tramite APSP di poche unità, impiegandoci però meno di un secondo.

Nelle ultime due colonne sono indicati, invece, in migliaia il numero di lati ottenuti nel grafo quoziente finale e l'errore relativo rispetto al (presunto) diametro. Si noti che Multicoloring di registro massimo 1 equivale al semplice Degree Clustering.

In tabella 4.10 sono stati riportati alcuni risultati ottenuti su altri grafi, tra cui due reti sociali e le altre due reti stradali studiate in questa tesi. In particolare, sono stati indicati anche i lati del grafo quoziente originale (cioè dopo clustering semplice, in migliaia) per confrontarli poi con i lati del grafo quoziente ottenuto con l'applicazione di Multicoloring.

Tabella 4.8: roadPA - 10K

Registri	Diametro	T. clust. (s)	Tot. (s)	Lati (migliaia)	Err. rel.
1	[1076-1089]	45	49	57	35.5%
5	[967-976]	58	65	123	21.8%
10	[895-917]	59	68	214	12.7%
30	[852-868]	65	84	449	7.3%
50	[850-859]	68	91	560	7.1%
100	[849-858]	75	109	621	6.9%

Tabella 4.9: roadPA - 15K

Registri	Diametro	T. clust. (s)	Tot. (s)	Lati (migliaia)	Err. rel.
1	[1033-1093]	42	51	83	30.0%
5	[940-967]	53	68	176	18.4%
10	[902-913]	56	74	304	13.6%
30	[844-862]	59	93	648	6.3%
50	[836-846]	67	102	776	5.3%
100	[831-845]	76	119	833	4.7%
500	[831-845]	78	121	835	4.7%

Tabella 4.10: Altri risultati

Dataset	Registri	Diametro	T. clust.	Tot. (s)	Lati orig.	Lati finali
dblp (1K)	10	24	32	35	189	370
livejournal (1K)	10	22-23	43	47	671	740
roadTX (15K)	30	1185	65	102	84	662
roadCA (15K)	30	1017	100	145	91	789

Reti stradali Per le reti ad alto diametro, obiettivo primario di questo nuovo algoritmo, il miglioramento è notevole. A fronte di un piccolo incremento del tempo di esecuzione siamo infatti in grado di migliorare di quasi un ordine di grandezza la qualità dell'approssimazione del diametro.
Ad esempio, nel grafo *roadPA* siamo passati da un errore relativo di oltre il 37% della versione base (nel caso peggiore) ad un errore del 4.7% aumentando il tempo di clustering di soli 30 secondi.
In Tabella 4.9 Notiamo che, aumentando i registri, ad un certo punto la qualità della stima si assesta e anche il tempo smette di crescere: dato che l'algoritmo termina non appena ogni nodo è associato ad un cluster, se all'algoritmo diamo un numero massimo di registri sufficientemente alto, quasi nessuno di questi riuscirà a saturarsi prima che l'algoritmo termini, comportamento confermato anche dai lati ottenuti nel grafo quoziente.

Reti sociali Nella sezione precedente avevamo già appurato la bontà della stima sulle reti sociali ottenuta attraverso *Degree Clustering*, perciò non ci sorprende osservare (Tabella 4.10) che con *Multicoloring* il miglioramento è marginale, a spese però del tempo di esecuzione che cresce rapidamente all'aumentare del numero di registri.

Conclusioni Nelle reti stradali il contributo portato da Multicoloring è determinante, riuscendo ad avvicinare la stima del diametro a poche decine di distanza da

quella vera (su diverse centinaia) in un tempo assolutamente competitivo. Questo algoritmo si configura quindi come la migliore soluzione — per rapporto qualità / costo — esistente al momento per il calcolo del diametro in reti fortemente sparse, nella nostra architettura di riferimento.

Nelle reti sociali invece, almeno per i 3 grafi scelti in questa tesi, il contributo portato da Multicoloring è marginale. Alcuni test preliminari suggeriscono però che in reti sociali dal diametro più elevato di quelle qui presentate potrebbe essere ancora vantaggioso applicare questo algoritmo. Ulteriori studi potranno confermare o smentire queste evidenze.

Un passo successivo potrebbe essere, attraverso un *sampling* preliminare del grafo per calcolarne il grado di espansione, determinare automaticamente il miglior numero di registri da utilizzare per avvicinare sufficientemente la stima del diametro a quello reale, senza compromettere però più del dovuto il tempo necessario.

Capitolo 5

Centrality

Dopo aver studiato nei dettagli l'applicazione di tecniche di clustering al calcolo di una proprietà globale del grafo, come è il diametro, vediamo ora com'è possibile applicare questi concetti per lo studio della *centrality*, che mira invece a determinare l'importanza di un nodo all'interno della rete.

In particolare, in questa tesi studieremo l'**Harmonic Centrality**, poiché si è rivelata una misura al contempo robusta e affine alle problematiche già studiate per il diametro.

5.1 Definizioni

La nozione di centralità di un nodo nasce dai sociologi durante il secondo dopoguerra, ma una sua formalizzazione avviene solo nel 1979 ad opera di Freeman [13].
Idealmente, ogni nodo possiede un certo grado di rilevanza o importanza all'interno del dominio — sociale o meno — della rete in questione ed è naturale aspettarsi che questa rilevanza si manifesti attraverso la struttura del relativo grafo; la centralità di un nodo mira quindi a fornire una misura quantitativa di questa importanza, relativamente a tutti gli altri nodi presenti nella rete.

La quasi totalità delle misure di centralità studiate si basa sul concetto di cammino minimo precedentemente formalizzato: un nodo "importante" dovrebbe avere una distanza media dagli altri nodi più piccola rispetto ad un nodo periferico, che dovrà invece percorrere mediamente una distanza più lunga per raggiungere un qualsiasi altro nodo.

Sfortunatamente, una misura di centralità ottima in un contesto potrebbe essere non adeguata in un altro; se così non fosse non esisterebbe d'altronde un tale numero di misure sviluppate.

Possiamo quindi suddividere le misure di centralità in due categorie: le misure *geometriche*, basate unicamente sul numero di nodi presenti ad una determinata distanza,

e quelle *path-based*, dipendenti invece dai percorsi di tutti i cammini minimi presenti nel grafo.

5.1.1 Misure geometriche

Indegree

Indegree, o grado entrante, è una misura geometrica poiché interpretabile come il numero di nodi a distanza 1 ed è anche la prima misura di centralità mai utilizzata. Nonostante la sua semplicità e apparente banalità, è un'ottima base di partenza per molte analisi e in certi casi riesce anche ad ottenere risultati migliori di misure più complicate.

Closeness

La *closeness* di un nodo x è definita come il reciproco della somma delle distanze da tutti i nodi y del grafo, ovvero

$$\frac{1}{\sum_y d(y,x)} \tag{5.1}$$

Un nodo periferico, come detto precedentemente, avrà distanze più lunghe a denominatore rispetto ad un nodo centrale, risultando quindi in una closeness minore. Si noti che le distanze sono definite dal nodo y verso il nodo x: in un grafo non orientato il cammino minimo che collega due nodi è uguale in entrambe le direzioni, in un grafo orientato invece ci interessa sapere la lunghezza del percorso che un generico nodo y deve percorrere per raggiungere x. La definizione inoltre, per avere senso, necessita che il grafo sia fortemente connesso altrimenti alcune distanze saranno ∞, risultando in uno score nullo.
Per ovviare a questo problema, viene spesso considerata la seguente versione corretta

$$\frac{1}{\sum_{d(y,x)<\infty} d(y,x)} \tag{5.2}$$

Con questa definizione viene naturalmente introdotto un forte bias verso i nodi raggiunti da pochi nodi, ovvero un basso *coreachable set*, poiché lo score massimo è 1 ed ogni distanza aggiunta andrà a diminuirlo, perciò non è possibile utilizzare questa misura per analisi rigorose.

Lin's index

Una soluzione ai problemi della closeness centrality appena descritti, gestione degli infiniti e delle differenti taglie dei coreachable set, è stata sviluppata da Nan Lin [16] con la seguente misura

$$\frac{|\{y : d(y,x) < \infty\}|^2}{\sum_{d(y,x)<\infty} d(y,x)} \tag{5.3}$$

Innanzitutto, una prima moltiplicazione per la taglia del coreachable set di x permette di normalizzare la metrica per tutto il grafo, ottenendo la distanza media; successivamente, l'elevamento a potenza permette di assegnare uno score più alto a nodi che si trovano in regioni più popolate del grafo.
Seppur questa definizione sembra risolvere tutte le problematiche riscontrate, uno studio condotto da Boldi e Vigna [3], basato su alcuni assiomi intuitivi da loro formalizzati che una buona misura di centralità dovrebbe rispettare, ci ha portato a scartarla e concentrare invece gli sforzi sulla prossima misura.

Harmonic

Inizialmente introdotta nel 2009 [21], questa misura permette di gestire in modo pulito la presenza di molti nodi non raggiungibili tra loro nel grafo. E' stato osservato [17] che in presenza di distanze infinite la media armonica si comporta meglio della media aritmetica, inoltre è l'unica centralità in grado di soddisfare i tre assiomi nello studio di Vigna et al. Per ulteriori dettagli, è possibile consultare [3]. L'*harmonic centrality* è quindi così definita

$$\sum_{y \neq x} \frac{1}{d(y,x)} \tag{5.4}$$

La robustezza della misura alle diverse tipologie di grafo, la facilità di approssimazione e la sua affinità con le altre tematiche affrontate in questo lavoro ci hanno portato a sceglierla come oggetto di questo studio, a discapito di altre misure molto studiate dalla comunità internazionale, di seguito descritte per dovere di completezza.

5.1.2 Misure path-based

A differenza delle misure geometriche, quelle path-based non dipendono semplicemente dal numero di nodi presenti ad una determinata distanza, quanto dalla struttura generata dai cammini minimi tra questi.
La metrica più studiata in letteratura è quella della *betweenness* ma vedremo come anche la grande famiglia delle misure spettrali sia rappresentabile in questa categoria.

Betweenness

Formalizzata da Freeman nel 1977 [14], la *betwenness centrality* quantifica il numero di volte che un particolare nodo x viene attraversato da un qualsiasi cammino minimo tra due coppie di nodi nella rete. In particolare, definendo con σ_{yz} il numero di cammini minimi esistenti tra i nodi y e z, e con $\sigma_{yz}(z)$ il numero di questi che attraversano il nodo x, possiamo definire la betweenness centrality come

$$\sum_{y,z \neq x, \sigma_{yz} \neq 0} \frac{\sigma_{yz}(z)}{\sigma_{yz}} \tag{5.5}$$

Questa misura assume una importanza particolare nelle applicazioni rappresentabili come problemi di flusso: rimuovendo dalla rete i nodi con betweenness più alta, infatti, si eliminano i principali *hub* attraverso cui transitano i dati, portando rapidamente al malfunzionamento o completa dissoluzione della rete.

PageRank e altre misure spettrali

Le misure spettrali misurano l'influenza di un nodo in una rete con l'idea che un collegamento ad un nodo molto importante dia un contributo maggiore rispetto ad un collegamento ad un nodo con uno score più basso. Lo score di ogni nodo, cioè, dipende dagli score di tutti gli altri nodi della rete. Possiamo allora definire lo score di un nodo i come

$$x_i = \frac{1}{\lambda} \sum_j a_{i,j} x_j \tag{5.6}$$

dove λ è una costante e $a_{i,j}$ è un elemento della matrice di adiacenza $\mathbf{A}$, uguale a 1 se i e j sono connessi da un arco e 0 altrimenti. Dall'equazione precedente si ottiene facilmente

$$\mathbf{Ax} = \lambda \mathbf{x} \tag{5.7}$$

da cui il nome di misure spettrali, ovvero dipendenti dall'autovettore sinistro della matrice di adiacenza.

L'esempio più famoso di misura spettrale è *PageRank*, utilizzato nel primo algoritmo di ranking di Google e formalizzato da Page e Brin [6] come il vettore

$$\mathbf{p} = \alpha \mathbf{p} \overline{A} + (1 - \alpha)\mathbf{v} \tag{5.8}$$

dove $\overline{A}$ è la matrice di adiacenza con righe normalizzate, α è il *damping factor* e $\mathbf{v}$ è il *preference vector*, una distribuzione utilizzabile per favorire determinate pagine. Non facendo parte degli obiettivi di questo lavoro, per ulteriori dettagli è possibile consultare l'articolo originale [6]. Ad ogni modo, l'idea di PageRank è quella di assegnare ad ogni pagina Web la probabilità che un generico utente ci arrivi iniziando a navigare da una pagina qualsiasi, con la probabilità, in ogni step, che l'utente clicchi su un nuovo link pari al damping factor.

5.2 HyperBall

Dopo aver inizialmente definito l'algoritmo HyperBall nella sezione 3.2.2, abbiamo visto una sua prima applicazione nel calcolo del diametro di un grafo, semplicemente contando il numero di iterazioni necessarie alla terminazione dell'algoritmo.

Avevamo visto nella descrizione dell'algoritmo 3 che, all'interno del ciclo alle righe 11–14, è possibile determinare il numero di nodi ad una specifica distanza dal nodo x. A questo punto è banale adattarlo per il calcolo dell'Harmonic Centrality, effettuando le seguenti modifiche.

1. Per ogni nodo, oltre al contatore HyperLogLog associamo anche una variabile *harmonic* inizializzata a 0 che conterrà l'Harmonic parziale del nodo.

2. All'iterazione i-esima, prima di effettuare l'unione tra il contatore del nodo x e quelli ricevuti dai vicini (riga 12), salviamo la cardinalità corrente del contatore. Calcolando la cardinalità dopo l'unione ed effettuando la differenza, saremo in grado di ottenere una stima N_i del numero di nodi a distanza i da x.

3. Aggiorniamo l'Harmonic del nodo x eseguendo $harmonic = harmonic + N_i/i$.

In tal modo, senza modificare il tempo richiesto dall'algoritmo HyperBall ma solo aumentando leggermente lo spazio, siamo in grado di calcolare con precisione variabile l'Harmonic Centrality di tutti i nodi del grafo.

La stima ottenuta con questo metodo è sufficientemente buona, tant'è che sarà utilizzata come valore di riferimento per esaminare quanto siano buoni i valori ottenuti invece tramite approcci basati su clustering.

E' giusto ricordare però che questo algoritmo è utilizzabile solo su grafi non pesati: su grafi pesati, infatti, la distanza di un nodo non corrisponde al numero di iterazioni che questo ha impiegato per diffondersi all'interno della rete.

5.3 Utilizzo semplice del clustering

5.3.1 Metrica di confronto

Al contrario del diametro, dove per misurare la bontà di un approccio basta semplicemente calcolare l'errore relativo della stima rispetto al vero diametro, decisamente più impegnativa è la scelta di un metodo affidabile in grado di valutare l'efficacia di un algoritmo nel calcolo della centrality.

Bisogna innanzitutto chiedersi quale sia lo scopo primario di una misura di centrality. Sicuramente, presi due nodi qualsiasi del grafo, ci aspettiamo che il più importante tra i due sia quello con la centrality più alta. Quindi non è importante la "scala" della centrality approssimata rispetto alla centrality reale, quanto che il valore approssimato mantenga l'ordine relativo corretto tra nodi. Inoltre, in problemi di questo tipo solitamente si è interessati ad ottenere la lista dei soli k nodi più importanti, piuttosto che mantenere l'esatto ordine relativo tra ogni coppia possibile.

Si pensi ad esempio ai risultati di una ricerca su Google: non è fondamentale che il sesto risultato più importante sia visualizzato in sesta posizione piuttosto che una posizione prima o dopo, quanto piuttosto che i 10 risultati più rilevanti siano effettivamente restituiti nella prima pagina e non nelle pagine successive.

Definiamo allora il concetto di Top-k.

Definizione: Top-k. Dato un insieme V di nodi dotati di *score*, definiamo l'insieme Top-k, $k \geqslant 1$, come

$$\text{Top-}k = \{v \in V : score(v) \geqslant score(v_k)\} \tag{5.9}$$

dove v_k è il k-esimo elemento nella lista ordinata per score decrescente dei nodi in V.

Poniamoci un obiettivo al contempo ambizioso e realistico: dato un grafo in input, vogliamo calcolarne i Top-k nell'ordine di una parte su un milione. Su un grafo di 1 milione di nodi, quindi, saremo interessati a trovare i 10–100 nodi più importanti.

5.3.2 Calcolo della centrality per cluster

Se vogliamo utilizzare il clustering di un grafo per calcolare la centrality dei propri nodi, il metodo più immediato è quello di calcolare la centrality approssimata solo per i centri dei cluster e quindi associare ad ogni nodo la centrality del proprio centro. Essendo un generico nodo vicino al proprio centro, infatti, la distanza dagli altri nodi del grafo, e quindi la differenza di Harmonic Centrality tra i due, non potrà essere troppo dissimile.

Algoritmo 8 Calcolo centrality per cluster

Input: G, grafo clusterizzato con n cluster
Output: H, array di grandezza n con la centrality dei centri

```
1:  function ClusterHarmonic(G)
2:      quotient ← Estrai grafo quoziente da G
3:      dists ← APSP(quotient)
4:      size ← Estrai grandezza cluster // mappa (cluster, taglia)
5:      radius ← Estrai raggio cluster // mappa (cluster, raggio)
6:      H ← Array[n]
7:      for i ← 1 to n do
8:          for j ← 1 to n do
9:              if i = j then
10:                 H(i) ← H(i) + size(i) / 1/2 radius(i)
11:             else if dists(i, j) < ∞ then
12:                 H(i) ← H(i) + size(j) / dists(i, j)
13:             end
14:         end
15:     end
16:     return H
17: end function
```

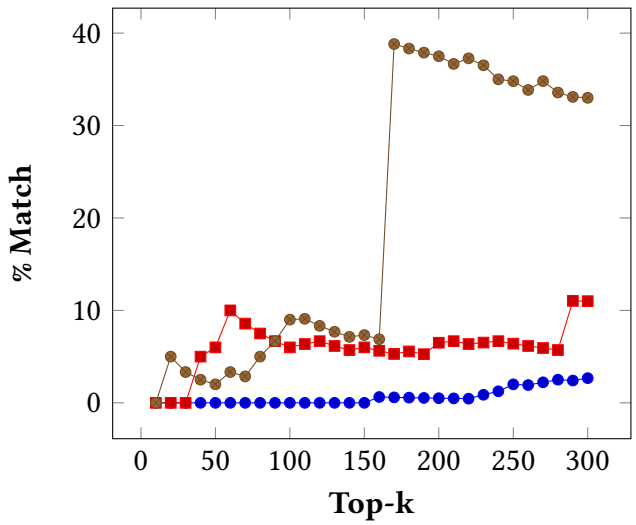
I contributi provenienti dal proprio cluster, per limitare la complessità computazionale, li supponiamo a distanza $raggio/2$. Non è un problema fare questa assunzione per le reti a bassa espansione, poiché è stato osservato sperimentalmente che per un generico nodo i contributi provenienti dal proprio cluster contribuiscono al valore finale di centrality solo per una piccola frazione. La quasi totalità del valore finale di centrality deriva infatti dai nodi appartenenti agli altri cluster del grafo. Pertanto, per meglio approssimare il valore finale di centrality è più importante calcolare correttamente le distanze con i cluster esterni. I contributi provenienti da cluster distinti, dato che non conosciamo la distanza effettiva dei singoli nodi, sono approssimabili usando come distanza media quella dal centro del loro cluster. Questa strategia non è ideale per le reti sociali, ma come vedremo più avanti su queste reti ci sono anche altri problemi che rendono gli approcci basati su clustering complicati, pertanto ci concentreremo sulle reti stradali.

Così facendo, tutti i nodi appartenenti allo stesso cluster avranno uguale centrality, quindi per definizione di Top-k saranno inseriti nell'insieme finale tutti i nodi appartenenti al cluster del k-esimo elemento.

Risultati

Occupiamoci per il momento solamente delle reti stradali, che sono il target primario di questo approccio.

Figura 5.1: roadPA - 15K



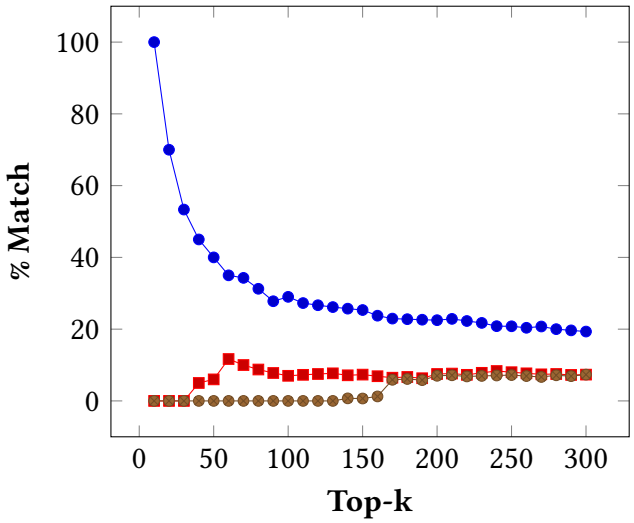
In Figura 5.1 sono rappresentate diverse esecuzioni dell'algoritmo Degree Clustering con taglia 15.000. In ascissa troviamo il numero di Top-k selezionati, da 10 a

300, mentre in ordinata è rappresentata la percentuale di match ottenuta, ovvero quale porzione dei veri Top-k sono compresi nei Top-k approssimati restituiti dall'algoritmo.

Possiamo vedere che, nei casi più fortunati, otteniamo un match del 40% quando iniziamo a considerare centinaia di Top-k. Le discontinuità rappresentate, ovvero i miglioramenti improvvisi seguiti da un matching decrescente fino alla discontinuità successiva, sono dovute alla definizione 5.3.1 di Top-k: in questa fase tutti i nodi appartenenti allo stesso cluster hanno uguale centrality, quindi quando ne viene incluso uno vengono inclusi anche tutti gli altri nodi appartenenti a quel cluster. Perciò, l'inclusione di un nuovo cluster migliora improvvisamente la percentuale di match e poi, finché non viene incluso il successivo, la percentuale decresce.

Esaminiamo adesso in Figura 5.2 cosa succede applicando l'algoritmo Multicoloring di stessa taglia e limitato a 50 registri.

Figura 5.2: roadPA - 15K - 50 registri



La situazione è migliorata, ma le percentuali di match sono ancora troppo basse per un utilizzo realistico e inoltre sono state ottenute includendo molti più nodi di quelli richiesti. In certi casi vengono restituiti infatti anche 5 volte gli effettivi Top-k, a causa dell'inclusione forzata degli interi cluster.

Se vogliamo quindi ottenere i pochi nodi più importanti su milioni un approccio approssimato per cluster non può funzionare: bisogna ottenere un valore univoco per singolo nodo.

Calcolo della centrality per vertice

Invece di calcolare la centrality soltanto per i centri dei cluster, proviamo a calcolarla per ogni nodo. Dato un nodo x, per calcolare la distanza dal nodo y possiamo utilizzare

l'upper bound $dist(x, C_x) + dist(C_x, C_y) + dist(C_y, y)$ dove C_x e C_y sono i centri dei due nodi, la distanza di un nodo dal proprio centro è già conosciuta e la distanza tra i due centri viene calcolata applicando *APSP* sul grafo quoziente.

Algoritmo 9 Calcolo centrality per vertice

Input: $G = (V, E)$, grafo clusterizzato con n vertici
Output: H, array di grandezza n con la centrality dei nodi

```
1:  function VertexHarmonic(G)
2:      quotient ← Estrai grafo quoziente da G
3:      dists ← APSP(quotient)
4:      H ← Array[n]
5:      for each i in V do
6:          C_i ← centro del nodo i
7:          for each j in V do
8:              C_j ← centro del nodo j
9:              if i ≠ j ∧ dists(C_i, C_j) < ∞ then
10:                 totalDist ← dist(i, C_i) + dists(C_i, C_j) + dist(j, C_j)
11:                 H(i) ← H(i) + 1 / totalDist
12:             end
13:         end
14:     end
15:     return H
16: end function
```

Purtroppo tale approccio non è praticabile. Sulle reti sociali i contributi provenienti da cluster esterni sono meglio approssimati utilizzando per ogni nodo l'upper bound appena descritto, ma nelle reti stradali invece, target primario dei nostri algoritmi, aumentiamo inutilmente tutte le distanze e peggioriamo le performance rispetto all'utilizzo di una distanza media. Inoltre, è un algoritmo estremamente lento rispetto agli altri approcci.

5.3.3 Altri approcci

Ricapitolando, l'approssimazione per cluster ha fornito qualche risultato interessante sulle reti stradali, al contrario delle reti sociali dove l'approccio è risultato inutilizzabile. L'approssimazione per vertice invece sperimentalmente ha portato qualche miglioramento sulle reti sociali ma ha peggiorato le performance sulle reti stradali.

Per ottenere un algoritmo valido per qualsiasi tipo di grafo e al contempo migliorare i risultati, si è tentato di prendere in considerazione più iterazioni dello stesso algo-

ritmo, considerando come valore finale di centrality, per ogni nodo, il valore massimo individuato o il valore medio della centrality di quel nodo attraverso le varie iterazioni.

Purtroppo nessun approccio basato su iterazioni multiple ha portato a miglioramenti consistenti, quindi anche questa strada è stata abbandonata.

5.4 Progressive Clustering

Dato che siamo interessati a trovare i Top-k di un grafo, piuttosto che cercare di approssimare bene la centrality di tutti i nodi del grafo è conveniente concentrarsi solo su quei nodi che potrebbero appartenere ai Top-k desiderati e migliorare la stima di quest'ultimi.

Introduciamo quindi il concetto di *Progressive Clustering*: l'idea è quella di partire da un clustering *coarse* del grafo, rifinendo man mano le zone più importanti fino ad arrivare ad una buona approssimazione della centrality di una frazione sufficiente di nodi.

5.4.1 2-Steps Clustering

Il primo algoritmo è chiamato *2-Steps Clustering* poiché, a partire da un grafo già processato, passa direttamente a rifinire un determinato numero di singoli nodi. Altri approcci, come vedremo tra poco, passano invece per una ulteriore fase intermedia di clustering.

Algoritmo

Algoritmo 10 2-Steps Clustering

Input: G, grafo clusterizzato con n nodi. k, numero Top-k (minore del 5% di n)
Output: Top-k

1: $clusterCentralities \leftarrow$ ClusterHarmonic(G)
2: $clusterTopK \leftarrow$ cluster appartenenti al Top-5% di $clusterCentralities$
3: $newG \leftarrow$grafo quoziente di $G \cup$ singoli nodi dei cluster in $clusterTopK$
4: $centralities \leftarrow \emptyset$
5: **for each** single vertex v **in** $newG$ **do**
6: Esegui Dijkstra a partire da v in $newG$
7: Calcola l'Harmonic Centrality h di v
8: Aggiungi h a $centralities$
9: **end**
10: **return** Top-k in $centralities$

L'algoritmo, partendo da un grafo G precedentemente processato da Degree Clustering o Randomized Clustering, calcola l'Harmonic Centrality per i centri dei cluster utilizzando l'algoritmo 8, quindi estrae i cluster appartenenti al 5% più elevato. E' importante ricordare che in questa fase ogni nodo del grafo originale viene considerato con centrality uguale a quella del proprio centro, quindi estrarre il primo 5% significa estrarre i cluster per centrality decrescente finché la somma delle loro cardinalità non supera il 5% del numero originale di vertici.

A questo punto viene costruito un nuovo grafo quoziente, formato dal precedente grafo quoziente e i singoli nodi appartenenti ai cluster selezionati dai Top-5%. Possiamo considerare i singoli nodi aggiunti come *singleton* cluster, ovvero cluster composti da un singolo nodo.

Successivamente, calcoliamo le distanze a partire da ogni nodo v appena aggiunto verso il resto del grafo e ne calcoliamo la centrality, tenendo conto delle opportune taglie dei cluster. Un "vecchio" cluster C_i contribuirà quindi per $size(C_i)/dist(v, C_i)$ mentre un *singleton* y contribuirà semplicemente per $1/dist(v, y)$.

Infine, estraendo i Top-k dalle centrality appena calcolate, avremo il risultato finale.

La soglia del 5% è stata scelta empiricamente poiché rappresenta un buon compromesso tra qualità finale dei Top-k restituiti e complessità computazionale. Infatti, ogni *singleton* aggiuntivo implica un SSSP[1] a partire da esso. In un grafo di 1 milione di nodi significa eseguire 50 mila SSSP finali.

Risultati

Reti stradali Osserviamo il comportamento dello step aggiuntivo rispetto ai risultati rappresentati in Figura 5.1.

Possiamo vedere nella Figura 5.3 che la situazione è molto migliorata, in quanto la percentuale di match peggiore ottenibile è pari al 60%. C'è troppa varianza da un'iterazione all'altra per poter definire tuttavia l'algoritmo stabile.

E' giusto ricordare però che questa volta non stiamo osservando percentuali falsate, poiché adesso nei Top-k restituiti ci sono effettivamente k nodi: stiamo infatti lavorando con nodi di cui abbiamo calcolato la centrality singolarmente, non per cluster.

Provando ad applicare come algoritmo di clustering *Multicoloring* con 50 registri (Figura 5.4) vediamo che la situazione migliora poiché esistono ancora picchi minimi attorno al 50/60%, ma sono casi isolati, e la percentuale media di match si è alzata all'85%. In Figura 5.5 è stato applicato l'algoritmo su un grafo processato con Randomized Clustering, sempre con target 15 mila nodi. Osserviamo che, a causa della natura fortemente randomica dell'algoritmo, si ottengono risultati estremamente discordanti.

[1] Single Source Shortest Paths

Picchi minimi del 50% sono però ancora troppo bassi per rendere l'algoritmo utilizzabile in qualsiasi situazione, quindi dobbiamo migliorare ulteriormente le performance sulle reti stradali.

Reti sociali Testiamo infine l'algoritmo 2-Steps Clustering sulla rete sociale *dblp*: in Figura 5.6 abbiamo applicato l'algoritmo al grafo processato con Degree Clustering (target 5000) mentre in Figura 5.7 abbiamo utilizzato Randomized Clustering. Possiamo osservare che, ad esclusione del primo 66% di match per i Top-3, con Degree Clustering abbiamo ottenuto risultati eccellenti, pertanto la strategia di selezionare i centri per grado è sicuramente efficace per individuare i Top-k nelle reti sociali. Non è stato possibile però applicare l'algoritmo ad altre reti sociali più grandi poiché in quest'ultime i Top-k si trovano in molti cluster diversi (a differenza delle reti stradali), quindi la strategia di prendere il primo 5% non funziona. Bisognerebbe infatti aumentare o la percentuale finale di nodi da prendere in considerazione, o il numero target di cluster. In entrambi i casi esplode velocemente il tempo di computazione, quindi l'approccio dev'essere ripensato per reti sociali massive.

Figura 5.3: roadPA 15K - Degree - 162 s

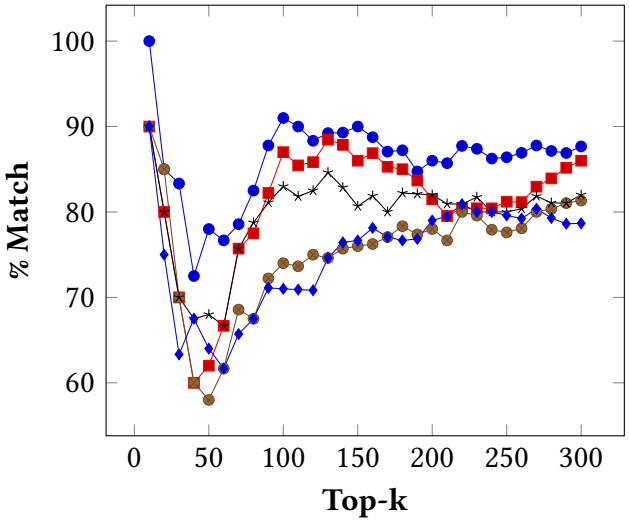


Figura 5.4: roadPA 15K - Multi 50 reg - 350 s

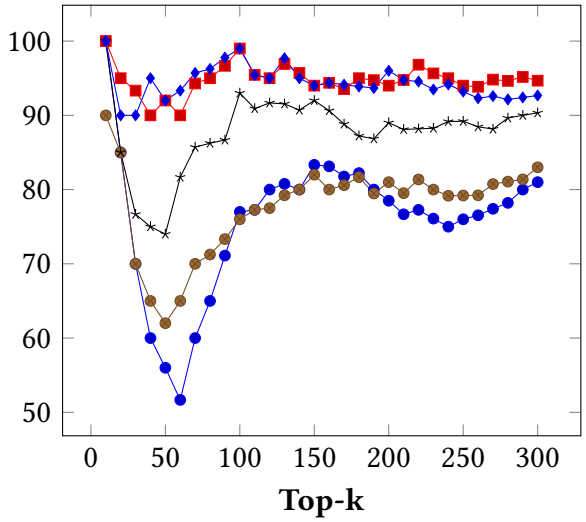


Figura 5.5: roadPA 15K - Random - 170 s

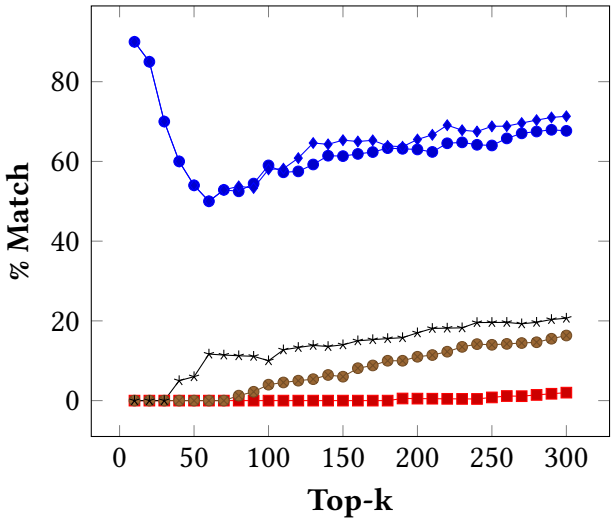


Figura 5.6: dblp 5K - Degree - 70s

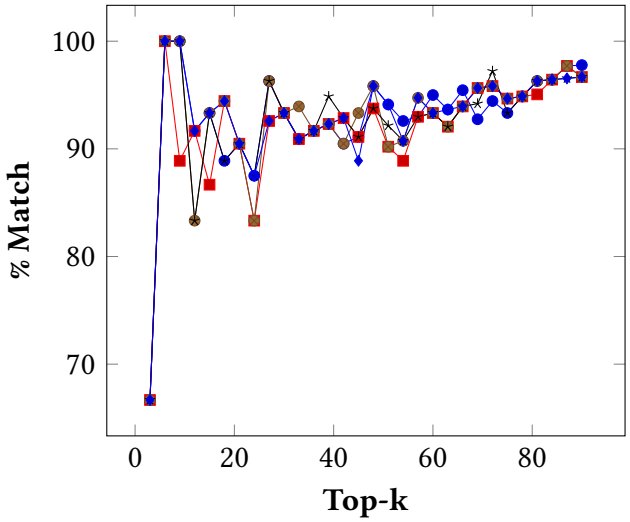
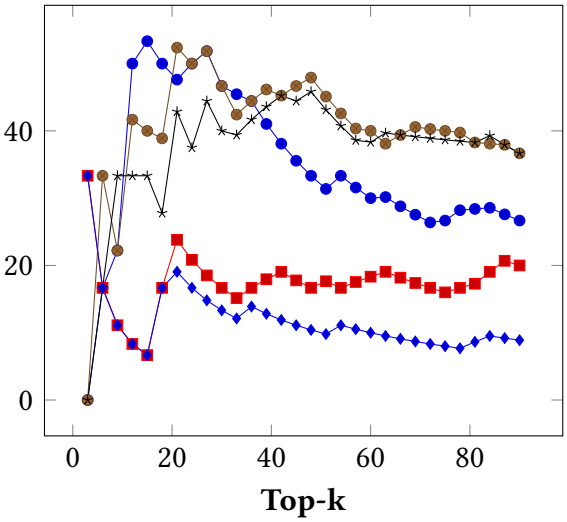


Figura 5.7: dblp 5K - Random - 65s



5.4.2 3-Steps Clustering

Aumentare la percentuale di nodi da rendere *singleton* cluster per poi eseguire un *SS-SP* a partire da ciascuno di essi (da noi empiricamente fissata al 5%), migliorerebbe sicuramente la percentuale di match ma aumenterebbe rapidamente il tempo richiesto dall'algoritmo.

Introduciamo allora un nuovo approccio denominato *3-Steps Clustering*.

Come suggerisce il nome, tra il clustering iniziale e la fase finale di esecuzione degli SSSP introduciamo una fase intermedia di clustering più "fine" di un adeguato sottografo della rete.

Algoritmo

Algoritmo 11 3-Steps Clustering

Input: G, grafo. t, taglia clustering. k, numero Top-k (minore di 5%)
Output: Top-k

1: $clustered \leftarrow$ CLUSTERING(G, t) // *clustering iniziale*
2: $clusterCentralities \leftarrow$ CLUSTERHARMONIC$(clustered)$
3: $subgraph \leftarrow$ cluster appartenenti al Top-33% di $clusterCentralities$
4: $refined \leftarrow$ CLUSTERING$(subgraph, t)$ // *clustering di refinement*
5: $clustered \leftarrow clustered \cup refined$
6: $clusterCentralities \leftarrow$ CLUSTERHARMONIC$(clustered)$
7: $clusterTopK \leftarrow$ cluster appartenenti al Top-5% di $clusterCentralities$
8: $newG \leftarrow$grafo quoziente di $clustered \cup$ singoli nodi dei cluster in $clusterTopK$
9: $centralities \leftarrow \emptyset$
10: **for each** single vertex v **in** $newG$ **do**
11: Esegui Dijkstra a partire da v in $newG$
12: Calcola l'Harmonic Centrality h di v
13: Aggiungi h a $centralities$
14: **end**
15: **return** Top-k in $centralities$

L'algoritmo riceve in input il grafo originale, quindi ne esegue un primo clustering utilizzando l'algoritmo desiderato (*Degree Clustering* con o senza *Multicoloring*, oppure *Randomized Clustering*) e il target t desiderato per il grafo quoziente. Viene quindi calcolata l'Harmonic centrality dei centri dei cluster utilizzando l'algoritmo 8; da questi si selezionano i cluster con centrality più alta, fino a raggiungere il 33% dei nodi originali del grafo.

Si procede quindi (riga 4) ad effettuare un reclustering di questo 33% più importante della rete, utilizzando lo stesso algoritmo e lo stesso target della prima fase. Viene aggiornato quindi il grafo processato (*clustered*): i nodi esclusi dal sottografo mantengono il centro individuato con il clustering originale, i nodi che sono invece stati riprocessati vengono memorizzati con i nuovi centri individuati. Otteniamo quindi un grafo con al più $2t$ cluster.

A questo punto applichiamo la terza ed ultima fase, utilizzando la stessa procedura descritta per l'algoritmo *2-Steps Clustering*. Preleviamo i cluster formanti il Top-5% del nuovo grafo (riga 7) quindi trasformiamo i nodi da cui sono formati in *singleton* cluster. Possiamo costruire infine il grafo quoziente finale e calcolare il valore di centrality per ogni nodo individuato nell'ultima fase.

Risultati

Come abbiamo già esaminato alla sezione 5.3.2, l'approssimazione di centrality per cluster non è adatta per estrarre i Top-k di bassa cardinalità, poiché il risultato è falsato dall'inclusione dei cluster interi, pertanto non è interessante esaminare le percentuali di match ottenute dopo la seconda fase.

Esaminiamo invece il risultato finale, ovvero quello ottenuto dopo la terza ed ultima fase, con i grafici da Figura 5.8 a Figura 5.21.

Per ogni grafico è indicato il grafo, la taglia target (15 mila per le reti stradali), l'algoritmo di clustering utilizzato con eventuale numero di registri, la percentuale finale di nodi trasformati in singleton cluster e il tempo totale in secondi.

Reti stradali In particolare, sono stati esaminati i grafi *roadPA* e *roadTX* utilizzando gli algoritmi *Degree Clustering*, *Multicoloring* e *Randomized Clustering*, studiando per ognuno di essi il comportamento con percentuale finale dell'1% e del 5%.

Esaminando i vari grafici è però complicato trarre delle conclusioni certe. In particolare, il miglioramento portato dall'esaminare nel dettaglio i Top-5% piuttosto che i Top-1% è discutibile, a fronte di un tempo raddoppiato.

Anche l'applicazione dell'algoritmo Multicoloring non sempre ha prodotto miglioramenti rilevanti: in generale, sembra migliorare l'approssimazione dei Top-k più piccoli, garantendo un match minimo del 70%, ma per i Top-k di taglia superiore il vantaggio su Degree Clustering non è evidente.

Randomized Clustering ha una varianza estremamente elevata per i Top-k più piccoli, ad esempio si può notare in Figura 5.16 e 5.17 che la percentuale di match per i Top-10 di *roadPA* può variare dallo 0% al 100%. Sulla rete roadTX, invece, non ci sono differenze apprezzabili rispetto a Degree Clustering.

Reti sociali Come discusso nella sezione 5.4.1, l'unica rete sociale esaminata è dblp. Nelle reti sociali i Top-k di nostro interesse sono distribuiti in molti cluster differenti

quindi nei grafi massivi, dove ogni cluster contiene molti nodi, i Top-(1–5)% da rifinire vengono saturati in fretta prendendo pochi cluster, che contengono però solo una piccola frazione dei Top-k che vogliamo individuare. La soluzione quindi è aumentare drasticamente il numero di cluster, in modo da renderli molto piccoli e permettere quindi una selezione più fine nel terzo step, oppure aumentare la percentuale finale da prendere in considerazione. In ogni caso, il tempo richiesto esplode rapidamente.

Figura 5.8: roadPA 15K - Degree - 1% - 140s

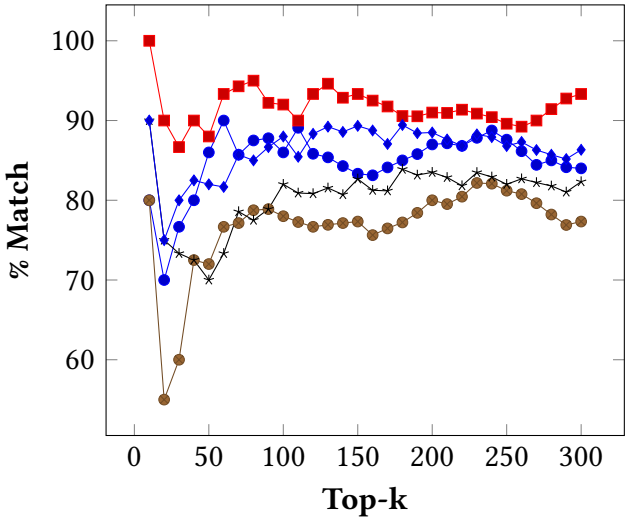


Figura 5.9: roadPA 15K - Degree - 5% - 260s

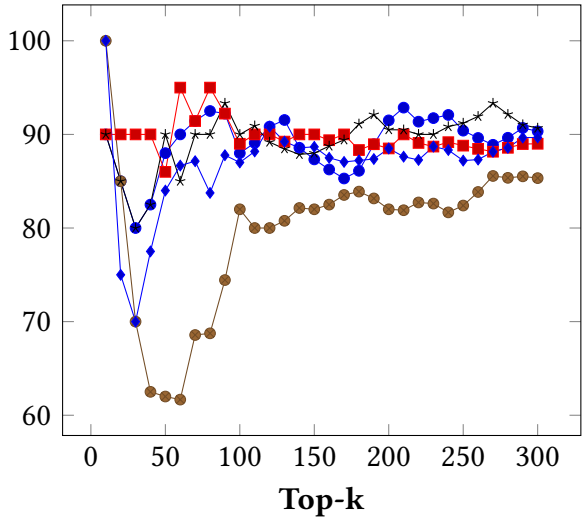


Figura 5.10: roadPA 15K - Multi 50 - 1% - 340s

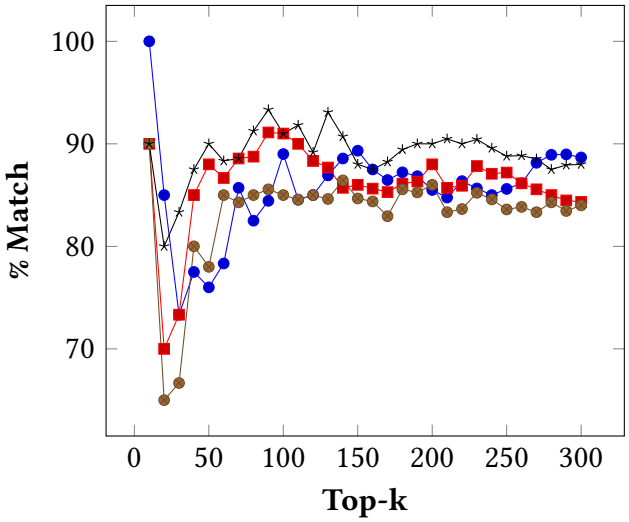


Figura 5.11: roadPA 15K - Multi 50 - 5% - 670s

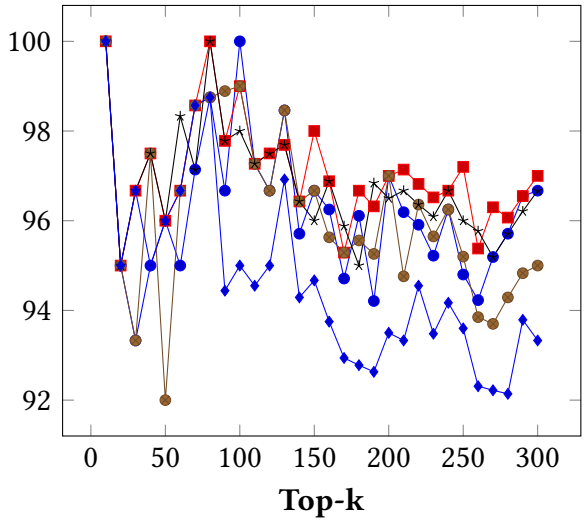


Figura 5.12: roadTX 15K - Degree - 1% - 140s

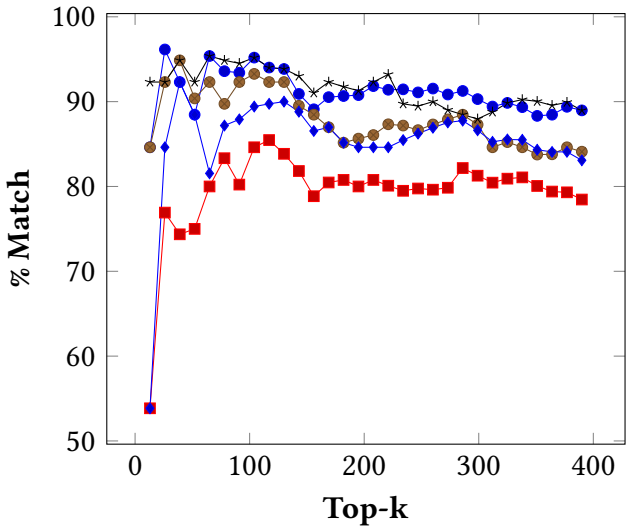


Figura 5.13: roadTX 15K - Degree - 5% - 330s

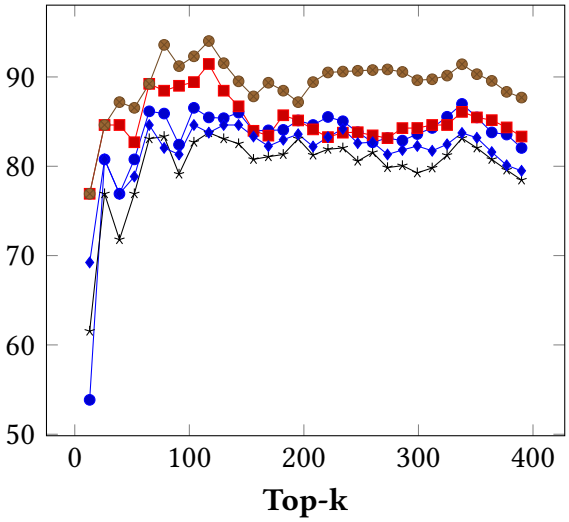


Figura 5.14: roadTX 15K - Multi 50 - 1% - 350s

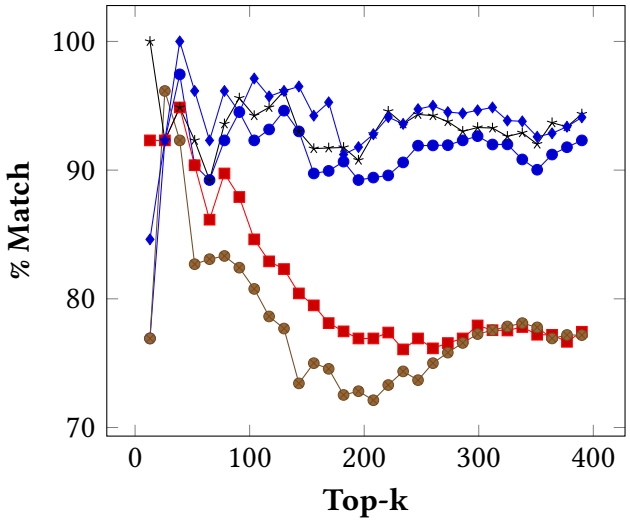


Figura 5.15: roadTX 15K - Multi 50 - 5% - 800s

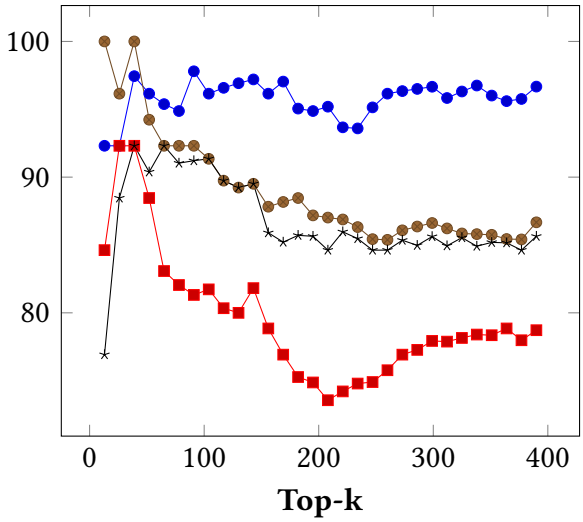


Figura 5.16: roadPA 15K - Random - 1% - 87s

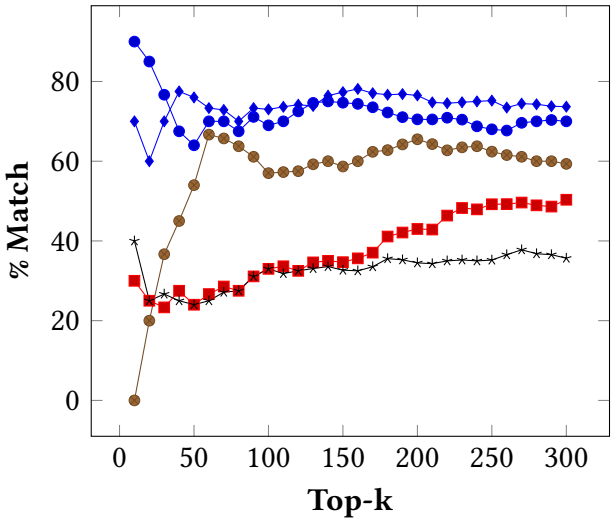


Figura 5.17: roadPA 15K - Random - 5% - 240s

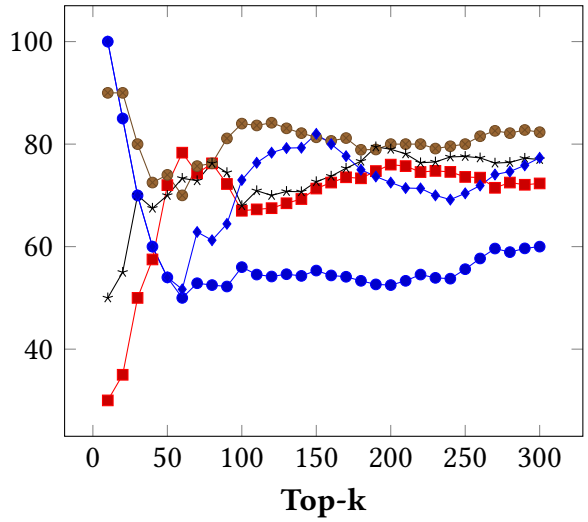


Figura 5.18: roadTX 15K - Random - 1% - 111s

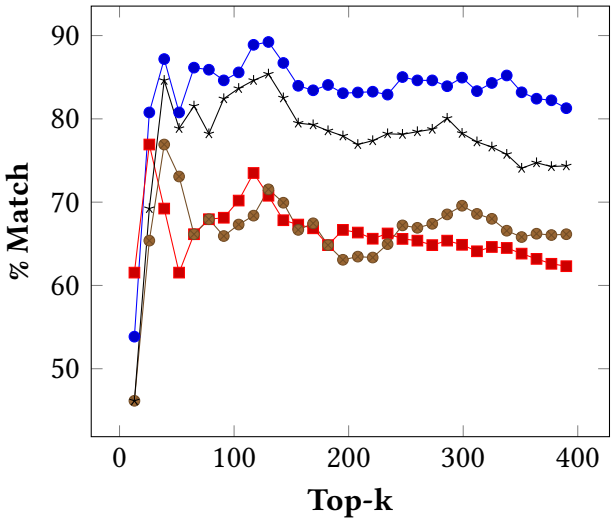


Figura 5.19: roadTX 15K - Random - 5% - 320s

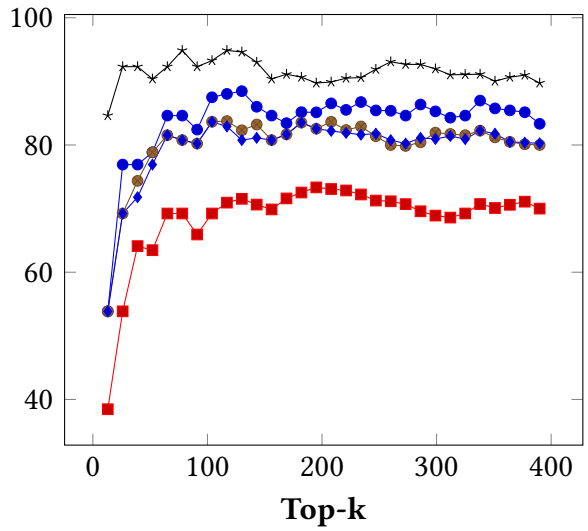


Figura 5.20: dblp 5K - Degree - 5% - 105s

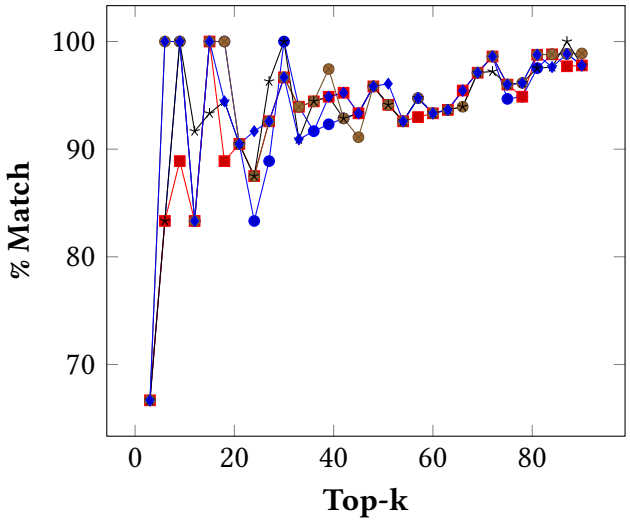
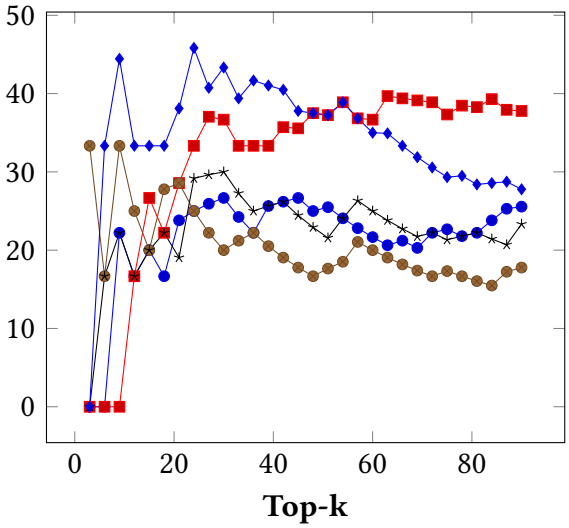


Figura 5.21: dblp 5K - Random - 5% - 95s



Capitolo 6

Conclusioni

6.1 Risultati ottenuti

Durante questo lavoro sono stati migliorati alcuni strumenti già esistenti e ne sono stati introdotti di nuovi. Di seguito verranno quindi riepilogati i risultati raggiunti in questa tesi.

HyperLogLog E' stata sviluppata una implementazione efficiente su Spark di contatori HyperLogLog, adatta per algoritmi iterativi dove è necessario trasferire e unire molti contatori diversi. Le attuali implementazioni sono usate infatti in sistemi di *logging* per contare pochi eventi in un flusso di informazioni, non adatte quindi per inizializzare un gran numero di oggetti.

Con questa implementazione adatta all'utilizzo su grafi non pesati, è possibile calcolare efficacemente non solo diametro o Harmonic centrality, ma in generale qualsiasi misura basata sulla distanza tra nodi.

Degree Clustering La decomposizione di un grafo in cluster permette di affrontare efficacemente il calcolo di misure pesanti, quali il diametro e in generale la distanza tra due nodi qualsiasi.

Nel tentativo di rendere più stabili i risultati ottenibili tramite l'algoritmo Randomized Clustering, si è dapprima modificato l'algoritmo in modo che selezionasse i nodi per grado invece che casualmente, mantenendo però il meccanismo di selezione dei centri per fasi. I risultati non soddisfacenti tuttavia hanno spinto in direzione della versione finale dell'algoritmo Degree Clustering, dove il maggior numero possibile di centri viene selezionato all'inizio e il loro numero non viene incrementato finché il tasso di copertura mantiene un ritmo sufficiente.

Questa scelta è stata premiante soprattutto nelle reti sociali, nelle quali grazie a questo nuovo algoritmo è possibile ottenere un'ottima stima del diametro (entro poche

unità da quello reale) in un tempo equivalente a quello delle soluzioni meno efficaci precedentemente esistenti.

Multicoloring Il vantaggio dell'algoritmo Degree Clustering è meno evidente invece nelle reti stradali, dove tutti i nodi hanno basso grado quindi la loro selezione secondo questo valore è circa equivalente ad una scelta casuale.

Con questo presupposto si è sviluppato l'algoritmo Multicoloring. L'algoritmo permette ai cluster di continuare a crescere in parallelo anche oltre i propri confini, terminando non appena ogni nodo è assegnato ad almeno un cluster.

Come studiato nel paragrafo 4.3.3 l'approccio si è rivelato vincente per ottenere un'ottima stima del diametro, infine, anche sulle reti stradali. Pur richiedendo un tempo di esecuzione leggermente maggiore rispetto alle tecniche classiche di clustering a causa, a parità di iterazioni, del maggior numero di messaggi in circolazione nel sistema, il miglioramento ottenuto sulla qualità della stima finale lo ripaga ampiamente.

Si configura quindi — per quella che è la nostra conoscenza — come il miglior algoritmo per calcolare il diametro di un generico grafo, scegliendo opportunamente target del grafo quoziente e memoria massima per nodo.

Centrality Seppur il clustering di un grafo permetta di realizzare un buon "oracolo" per la distanza tra due nodi qualsiasi (le due distanze dal proprio centro più la distanza tra i due centri), la sua efficacia diventa discutibile per il calcolo della centrality di un nodo. Nelle misure di centrality basate sulla distanza tra nodi, infatti, un errore trascurabile in una singola distanza accumulato molte volte rischia di rendere la misura inutilizzabile.

Abbiamo quindi sviluppato un algoritmo che permette, per la prima volta, di calcolare i Top-k di un grafo sfruttando un clustering progressivo. Partendo da un clustering a "grana grossa" e migliorando per stadi la stima delle distanze nelle aree più importanti, infatti, è possibile ottenere con buona precisione la lista dei Top-k, con k nell'ordine delle decine in grafi con milioni di nodi, in pochi minuti.

6.2 Sviluppi futuri

In questo lavoro sono stati affrontati diversi problemi e sono state di conseguenza sviluppate soluzioni differenti. Proprio per la diversità delle tematiche affrontate, non è stato possibile esaminare ogni problema nei più piccoli dettagli, ma si è voluto piuttosto dimostrare che approcci basati sul clustering sono efficaci nell'affrontare problematiche differenti.

In questa tesi sono stati testati solo grafi non diretti e non pesati. Seppur tutti gli algoritmi sviluppati siano in grado teoricamente di funzionare con grafi pesati (tranne, per definizione, HyperBall), non abbiamo prove della loro efficacia in questo caso.

Elenchiamo quindi una lista dei possibili sviluppi per ogni algoritmo introdotto in questo lavoro.

Degree Clustering Seppur l'algoritmo si sia rivelato efficace nella soluzione dei problemi preposti, meriterebbe uno studio più formale del suo comportamento al variare della tipologia del grafo. In particolare, come già detto, la selezione per grado nelle reti stradali ha vantaggi incerti rispetto alla selezione casuale e forse in queste reti la differenza comportamentale è imputabile semplicemente alla selezione iniziale dei centri invece che per fasi. Inoltre, non c'è una garanzia teorica sul numero finale di cluster a differenza di Randomized Clustering, ma in entrambi i casi non c'è alcun controllo sul numero di archi che saranno presenti nel grafo quoziente — che sono il fattore più importante negli algoritmi di calcolo dei cammini minimi — pertanto è un problema trascurabile.

Multicoloring Gli sforzi maggiori sono stati dedicati a rendere efficiente questa particolare versione distribuita di Dijkstra, non è stato pertanto possibile testare questo approccio anche con l'algoritmo Randomized Clustering, che potrebbe beneficiarne allo stesso modo di Degree Clustering.

Un passo importante per l'evoluzione ulteriore di questo algoritmo è però quello di realizzare, tramite un *sampling* preliminare del grafo per determinarne il grado di espansione, un metodo automatico per calcolare il target e la memoria massima per nodo più adatti al grafo in input, in relazione anche alle capacità computazionali della macchina in uso.

Inoltre, è verosimile aspettarsi dei discreti miglioramenti se, invece di terminare l'algoritmo non appena tutti i nodi sono assegnati ad un centro, gli permettessimo di continuare per qualche iterazione, lasciando così tempo ai nodi coperti per ultimi di ricevere più informazioni sui cluster confinanti. Sarebbe quindi interessante determinare, in relazione al grafo in input, il miglior compromesso tra numero di iterazioni aggiuntive e miglioramento della stima finale.

Centrality Innanzitutto, le performance degli algoritmi dovrebbero essere ricalcolate operando un confronto con valori di benchmark affidabili, poiché i valori di centrality calcolati tramite contatori HyperLogLog — anche con un numero elevato di registri — si sono rivelati instabili.

Dopo aver ottenuto dei valori di benchmark più affidabili per determinare la bontà degli approcci, sarebbe utile condurre uno studio approfondito delle varie combinazio-

ni ottenibili, variando ad esempio la percentuale di rete su cui effettuare il reclustering o la percentuale finale di nodi da trasformare in *singleton* cluster.

In questa sede, infatti, questi parametri sono stati scelti empiricamente in relazione alla potenza di calcolo disponibile, quindi il passo successivo dovrebbe permettere di ottenere delle regole con cui determinare, in base a grafo e potenza disponibile, i parametri che producono il miglior compromesso.

Infine, bisognerebbe modificare l'algoritmo in modo da migliorarne il comportamento sulle reti sociali: abbiamo notato che la strategia di selezionare i nodi per grado in questo tipo di grafi sia vincente, ma in queste reti ad alta espansione un contributo non trascurabile al valore finale di centrality arriva dal proprio cluster. Pertanto, invece di supporre tutti i nodi dello stesso cluster a distanza $raggio/2$ si potrebbero intanto calcolare questi contributi a distanza esatta per il centro del cluster. Rimane però il problema che, a differenza delle reti stradali, in quelle sociali i Top-k d'interesse tendono a distribuirsi in un numero maggiore di cluster, quindi bisognerebbe pensare ad un nuovo approccio.

Lista degli Algoritmi

Bibliografia

[1] G. E. Blelloch, A. Gupta, I. Koutis, G. L. Miller, R. Peng, and K. Tangwongsan. Near linear-work parallel sdd solvers, low-diameter decomposition, and low-stretch subgraphs. *SPAA*, 13–22, 2011.

[2] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. HyperANF: Approximating the Neighbourhood Function of Very Large Graphs on a Budget. 2011.

[3] Paolo Boldi and Sebastiano Vigna. Axioms for Centrality. *Internet Mathematics 10*, 2014.

[4] Paolo Boldi and Sebastiano Vigna. In-core computation of geometric centralities with HyperBall: A hundred billion nodes and beyond. *Proc. of 2013 IEEE 13th International Conference on Data Mining Workshops (ICDMW 2013)*, 2013.

[5] Marco Boscolo Anzoletti. Stima del diametro di un grafo con il framework Apache Spark. *Tesi magistrale*, 2015.

[6] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998.

[7] Matteo Ceccarello. Efficient MapReduce Algorithms for computing the diameter of very large graphs. *Tesi magistrale*, 2013.

[8] M. Ceccarello, A. Pietracaprina, G. Pucci, and E. Upfal. A Practical Parallel Algorithm for Diameter Approximation of Massive Weighted Graphs. 2015.

[9] M. Ceccarello, A. Pietracaprina, G. Pucci, and E. Upfal. Space and time efficient parallel graph decomposition, clustering, and diameter approximation. *ACM SPAA*, 2015.

[10] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM J. Comput.*, 28(1):210–236, 1998.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] Philippe Flajolet, Éric Fusy, Olivier Gandouet. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. *Proceedings of the 13th conference on analysis of algorithm (AOFA 07)*, 127–146, 2007.

[13] L. Freeman. Centrality in social networks: Conceptual clarification. *Social Networks*, 1(3):215–239, 1979.

[14] L. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[15] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2), 2011.

[16] Nan Lin. *Foundations of Social Research*. McGraw-Hill, New York, 1976.

[17] Massimo Marchiori and Vito Latora. Harmony in the small-world. *Physica A: Statistical Mechanics and its Applications 285*, 3(4):539–546, 2000.

[18] G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decompositions using random shifts. *SPAA*, 196–203, 2013.

[19] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. Anf: a fast and scalable tool for data mining in massive graphs. *KDD*, 81–90, 2002.

[20] Andrea Pietracaprina, Geppino Pucci, Matteo Riondato, Francesco Silvestri, and Eli Upfal. Space-round tradeoffs for mapreduce computations. *Proceedings of the 26th ACM international conference on Supercomputing, ICS '12*, 235–244, New York, NY, USA, 2012.

[21] Yannick Rochat. Closeness centrality extended to unconnected graphs: The harmonic centrality index. *Applications of Social Network Analysis*, ASNA 2009.

[22] Matei Zaharia, Mosharaf Chowdhury, Ion Stoica et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *NSDI 2012*, 2012

Appendice A

Dataset

In questa sezione descriviamo brevemente i dataset utilizzati per testare i vari algoritmi in questo lavoro.

I dataset sono stati prelevati dall'archivio *SNAP* di Stanford[1] e rielaborati, poiché nella loro versione originale sono assenti alcuni lati. Questi grafi sono infatti non diretti, quindi un arco indiretto tra due nodi x e y, nella pratica, viene rappresentato con due archi diretti (x, y) e (y, x).

In Tabella A.1 sono elencati i grafi con numero di nodi, numero di lati (diretti, quindi il doppio degli archi non diretti) e presunto diametro[2].

Sono state analizzate 3 reti sociali. *DBLP* rappresenta la rete degli autori di articoli e pubblicazioni di *Computer Science*, dove ogni autore è rappresentato da un nodo e due autori sono collegati se hanno pubblicato almeno un articolo assieme. *LiveJournal* è una piattaforma gratuita di blogging, dove gli utenti possono stringere amicizia. *Orkut* invece era un social network ad opera di Google, ora chiuso, dove gli utenti potevano stringere amicizia al pari del moderno Facebook.

Sono poi presenti le reti stradali *roadPA*, *roadTX*, *roadCA*, che rappresentano rispettivamente le strade degli stati statunitensi Pennsylvania, Texas e California. Un incrocio è rappresentato nel grafo da un nodo, quindi una strada consiste in un arco tra due nodi. Come intuibile, sono reti ad alto diametro poiché essendo reti "fisiche", a differenza delle reti sociali, non possono esistere archi che connettono direttamente due nodi distanti.

[1] http://snap.stanford.edu/data

[2] Come spiegato al paragrafo 4.2.3 non conosciamo il reale diametro di queste reti ma è stato possibile calcolarne una buona stima.

Tabella A.1: Statistiche grafi

	Nodi	Lati	Diametro
DBLP	317.080	4.199.464	23
LiveJournal	3.997.962	69.362.378	21
Orkut	3.072.441	234.370.166	9
roadPA	1.088.092	6.167.592	794
roadTX	1.379.917	7.686.640	1064
roadCA	1.965.206	11.066.428	852