Conteggio di Triangoli all'Interno di Grafi Massivi con MapReduce

---

Progetto per Sistemi di Elaborazione di Grandi Quantità di Dati

Xu Sunyi

VR435956

UNIVERSITÀ DEGLI STUDI DI VERONA
Anno Accademico 2019/2020

Indice

1 Introduzione

Per il progetto è stato sviluppato un programma che implementa l'algoritmo TTP, proposto dagli autori in [1], e compact forward, proposto in [2], per computare il numero di triangoli presenti all'interno di un grafo massivo utilizzando il framework MapReduce. L'obiettivo del progetto è quello di sviluppare un'applicazione in grado di elaborare grandi quantità di dati in parallelo in un sistema distribuito.

1.1 Struttura

Nella Sezione 2 viene introdotto formalmente il problema del conteggio dei triangoli mentre nella Sezione 3 viene descritto l'algoritmo TPP e il suo funzionamento. Il tool sviluppato viene presentato nella Sezione 4, dove si è posta particolare attenzione alle scelte implementative adottate. Il testing e la valutazione delle performance del programma vengono descritte nella Sezione 5. La relazione si conclude con la Sezione 6 dove viene mostrato un algoritmo per MapReduce per il calcolo di quadrati all'interno di un grafo. Esso è basato sull'idea del partizionamento e della classificazione in diverse tipologie di TTP.

2 Background

2.1 Definizione del problema

Definiamo ora formalmente il problema dei triangoli che vogliamo risolvere.

Definition 2.1. (Triangolo) Si consideri un grafo indiretto $G = (V, E)$ dove V è l'insieme dei vertici e E è l'insieme degli archi. Siano n e m la cardinalità di V e E rispettivamente. Siano $v, w \in V$ due nodi del grafo, (u, w) rappresenta l'arco che collega i due vertici, $(u, w) \in E$. Sia $N(v)$ l'insieme dei vicini di v, cioè l'insieme di tutti i vertici che hanno un arco con quest'ultimo e $d(V)$ il grado di v, i.e. $d(v) = |N(v)|$. Un triangolo, denotato da $\Delta(u, v, w)$ è un set di tre nodi $u, v, w \in V$ tali che $(u, v), (u, w), (v, w) \in E$.

Il nostro obiettivo è quello di trovare, dato un grafo G definito come sopra, il numero totale di triangoli presenti in esso. Per comprendere l'algoritmo proposto dagli autori in [1] bisogna prima definire alcune importanti nozioni che verranno usate nel resto della relazione.

2.2 Partizioni e archi

Definition 2.2 (Partizione o 1-partizione)**.** Dato un intero ρ, una partizione (o 1-partizione), denotato da $G_i = (V_i, E_i)$ o $G_j = (V_j, E_j)$, è un sottografo di G tale che $V_i \cap V_j = \emptyset$ dove $0 < i < j \leq \rho$, e $\bigcup_{i=1}^{\rho} V_i = V$ dove $0 < i \leq \rho$.

ρ è il numero di partizioni in cui viene suddiviso il grafo.

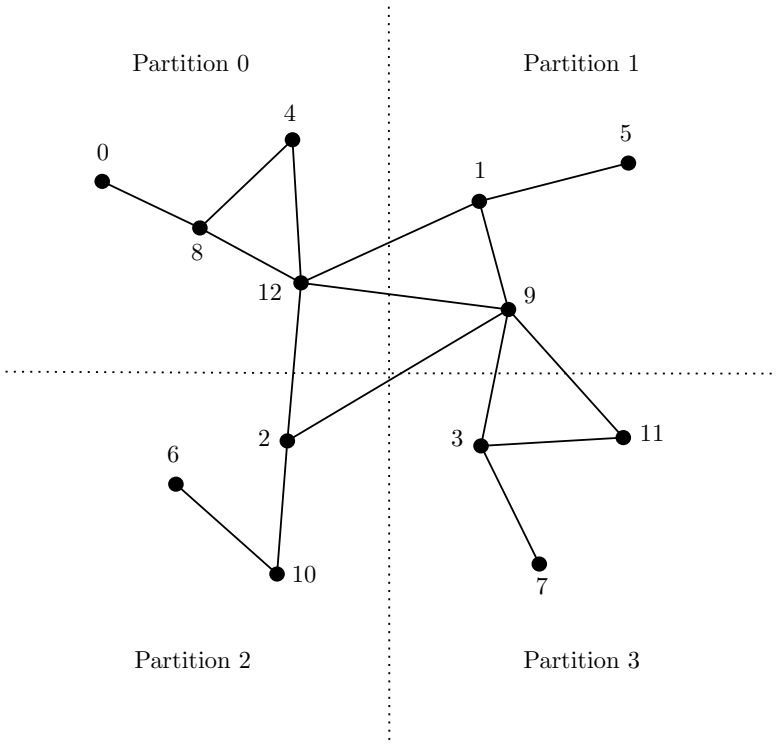


Figura 1: Grafo G suddiviso in $\rho = 4$ partizioni

Definition 2.3 (2-partizione e 3-partizione)**.** Siano G_i, G_j e G_k tre partizioni diverse. Una 2-partizione, denotata da $G_{ij} = (V_{ij}, E_{ij})$, è un sottografo indotto di G su $V_i \cap V_j$. Similmente, una 3-partizione, denotata da $G_{ijk} = (V_{ijk}, E_{ijk})$, è un sottografo di G su $V_i \cap V_j \cap V_k$.

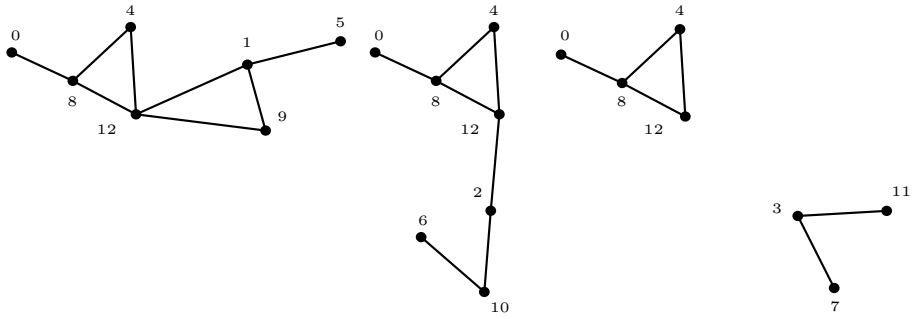
Il numero di possibili 2-partizioni e 3-partizioni per un grafo è di $\binom{\rho}{2}$ e $\binom{\rho}{2}$ rispettivamente. Prendiamo in considerazione il grafo G rappresentato in Figura 1. Esso è formato da 12 nodi, suddivisi in 4 partizioni, e avrà $\binom{4}{2} = 6$ 2-partizioni (Figura 2) e $\binom{4}{3} = 4$ 3-partizioni.

Definition 2.4 (Arco interno/esterno)**.** Dato un arco (u, v), esso è:

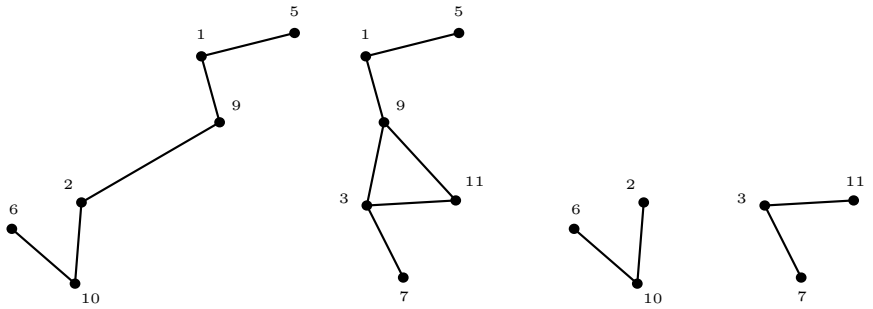
- arco interno se u e v appartengono alla stessa partizione, cioè $u, v \in G_i$;
- arco esterno se u e v appartengono a due partizioni differenti, $u \in G_i, v \in G_j, i \neq j$.

Definition 2.5 (3'-partizione)**.** Una 3'-partizione, denotata da G'_{ijk} , è un sottografo di una 3-partizione senza archi interni.

Figura 3 mostra tutte le possibili 3'-partizioni di G.

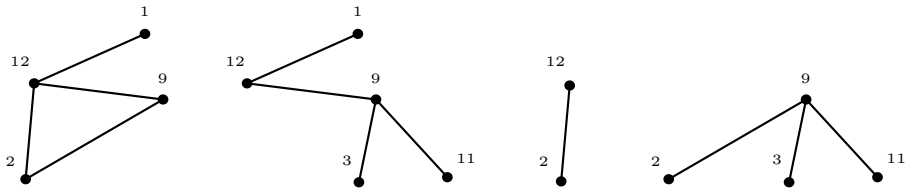


(a) G_{01} (b) G_{02} (c) G_{03}



(d) G_{12} (e) G_{13} (f) G_{23}

Figura 2: 2-partizioni del grafo G



(a) G_{012} (b) G_{013} (c) G_{023} (d) G_{123}

Figura 3: 3'-partizioni del grafo G

3 Algoritmo TTP - Triangle Type Partition

3.1 Tipi di triangoli

Partizionando un grafo si possono distinguere 3 tipi diversi di triangoli $\Delta(u, v, w)$ in base alle partizioni a cui appartengono i suoi vertici.

Definition 3.1 (Tipi di triangoli)**.**

- Type-1. Se i nodi u, v e w del triangolo $\Delta(u, v, w)$ sono nella stessa partizione, il tipo del triangono è Type-1.
- Type-2. Se due vertici del triangolo fanno parte della stessa partizione mentre il terzo appartiene a una partizione diversa, il triangolo è Type-2.
- Type-3. Se tutti e tre i nodi sono in partizioni diverse, il triangolo è Type-3.

Nel caso del triangolo di Type-1 tutti gli archi sono interni. Al contrario, gli archi di un Type-3 sono tutti esterni mentre nel Type-2 un arco è interno mentre gli altri due sono esterni. Per esempio, in Figura 1, il triangolo $\Delta(4, 8, 12)$ è un Type-1, $\Delta(1, 9, 12)$ e $\Delta(3, 9, 11)$ sono Type-2 mentre $\Delta(2, 9, 12)$ è un triangolo Type-3. I vari tipi di triangoli possono essere computati in diverse k-partizioni. In particolare, dato un triangolo $\Delta(u, v, w)$:

- se il triangolo è di tipo Type-1 allora possiamo trovarlo in G_i e in tutte le 2-partizioni e 3-partizioni che contengono la partizione i.
- se il triangolo è di tipo Type-2, dove i e j sono le due partizioni a cui appartengono i vertici, allora lo possiamo trovare nella 2-partizione $G_{i,j}$ e in tutte le 3-partizione che contengono le partizioni i e j.
- se il triangolo è invece di tipo Type-3, esso sarà presente sia in G_{ijk} che G'_{ijk}, dove i, j e k sono le partizioni a cui appartengono i vertici. Questo perchè i Type-3 contengono solo archi esterni.

3.2 Idea generale dell'algoritmo

L'idea di base dell'algoritmo TTP è di ripartire tutti gli archi del grafo nelle rispettive 2-partizioni e 3'-partizioni e di computare in esse i vari tipi di triangoli. I Type-1 e Type-2 verranno calcolati nei primi mentre i Type-3 nei secondi. Come abbiamo visto precedentemente, G'_{ijk} contiene solo archi esterni. Non è dunque possibile trovare al suo interno triangoli Type-2 e Type-1 visto che questi hanno almeno un arco interno. Ogni Type-2 viene calcolato una sola volta in un'unica 2-partizione mentre i Type-1 possono trovarsi in $(\rho - 1)$ 2-partizioni. Questo significa che ogni triangolo Type-1 comparirà $(\rho - 1)$ volte nel nostro output. Per questi ultimi bisognerà quindi aggiustare il loro valore nel conteggio che, invece di essere 1, sarà $\frac{1}{\rho-1}$. Siccome i Type-1 vengono già calcolati nelle 2-partizioni, è ridondante ricalcolarli nelle 1-partizioni. In sintesi quindi:

- i triangoli Type-1 vengono computati nelle 2-partizioni e contati con valore $\frac{1}{\rho-1}$;
- i triangoli Type-2 vengono trovati nelle 2-partizioni e conteggiati come 1;
- i triangoli Type-3 vengono cercati nelle 3'-partizioni e contati come 1.

Come possiamo osservare in Figura 2, $\Delta(1,9,12)$ e $\Delta(3,9,11)$ si trovano solo in G_{01} e G_{13} rispettivamente. $\Delta(4,8,12)$ si trova invece in 3 2-partizioni: G_{01}, G_{02} e G_{03}. Contando ogni istanza di questo triangolo come 1/3, la somma dà come risultato 1. Esso viene dunque considerato correttamente una volta sola nel conteggio totale dei triangoli del grafo. Il triangolo $\Delta(2,9,12)$ si trova invece solo in G'_{012} (Figura 3).

3.3 Schema MapReduce

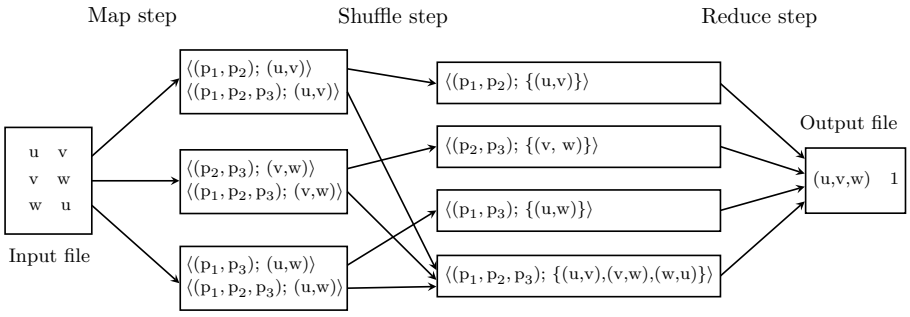


Figura 4: Schema MapReduce di TTP

Figura 4 mostra lo schema MapReduce generale per l'algoritmo. Nella fase di Map ogni mapper riceve in input una riga del file di input e produce in output una coppia $\langle key; value\rangle$ dove:

- *key* è una tupla formata da due oppure tre elementi, rappresentanti rispettivamente una 2-partizione o una 3'-partizione;
- *value* è l'arco preso in considerazione.

Nella fase di shuffle e sort, eseguita da combiners/partitioners, i valori vengono aggregati in base alla chiave e mandati ai reducer. Questo significa che vengono prodotti dati intermedi della forma $\langle(key); \{value1, value2, value3, ...\}\rangle$. Ogni reducer riceve in input una coppia del tipo $\langle key; valueSet\rangle$ prodotta nella fase precedente. A ogni reducer viene assegnata un sottografo (2-partizione o 3'-partizione) con i relativi nodi e archi da computare. In output vengono poi generate coppie $\langle key; value\rangle$ dove:

- *key* è una tupla di tre vertici che rappresenta un triangolo;
- *value* è il valore del triangolo per il conteggio totale.

I risultati della fase di reduce vengono salvati sul file di output (Figura 5).

4 Implementazione del Tool

In questa sezione verranno descritte le classi del tool e in generale le scelte progettuali adottate per ottimizzare il conteggio dei triangoli.

4.1 Dataset di input

Il programma riceve in input un semplice file di testo rappresentante il grafo che si vuole elaborare.

Formato del Dataset I file di input, per essere considerati validi, devono essere nel formato:

$$u\ v$$
$$v\ w$$
$$w\ u$$

dove ogni nodo viene univocamente identificato da un numero intero positivo. Ognuna delle m righe rappresenta un arco i cui vertici vengono separati da un tabulatore. Gli archi non devono essere ripetuti, i.e. se esiste l'arco (u, v) non deve essere presente (v, u). Inoltre gli autoanelli, e.g. (u, u), devono essere eliminati dal grafo poichè alternano le metriche utilizzate dall'algoritmo compact forward per il calcolo dei triangoli (Sezione 4.6.1). Si è scelto questo formato per il file di input poichè la maggior parte dei dataset analizzati sono già rappresentati in questo modo. I file che invece sono formattati diversamente sono stati modificati prima di eseguire il programma. È indifferente l'ordine in cui archi e nodi compaiono.

Provenienza dei Dataset Tutti i dataset utilizzati per il progetto sono liberamente reperibili sul sito della Stanford University[3]. Ogni network ha una sua pagina dove, oltre a una breve descrizione dei dati raccolti, sono presenti alcune informazioni statistiche sul grafo quali il numero di nodi e archi, il diametro, il coefficiente di clustering medio e il numero di triangoli nel grafo. Il dato di maggiore rilevanza per il nostro progetto risulta sicuramente quest'ultimo. È importante notare che il numero di triangoli presente sul sito viene calcolato considerando il grafo di input come indiretto, cioè non orientato. Di conseguenza, possiamo accettare in input per il tool anche i grafi che vengono classificati come diretti. Non tutti i dataset riportano il conteggio dei triangoli. Si sono dunque scelti per il testing solo quelli in cui è presente il dato, per poter avere un riscontro finale con il risultato del programma.

4.1.1 Differenze nel Formato dei File di Input

Nei file dei grafi orientati un arco (u v) indica che è presente un arco che va da u a v mentre per i grafi non orientati possiamo avere due notazioni diverse. In alcuni dataset, per evitare ridondanza, gli archi non sono ripetuti: se è presente (u v) non sarà elencato (v u), poichè viene implicitamente assunto che esista.

Questa è la notazione scelta per esempio per il set di *Facebook* [4]. In quello di *California road network* [5] invece, per ogni arco $(u\ v)$, esiste anche $(v\ u)$.

Script in Python e Bash Per rendere omogeneo il formato dei file viene utilizzato uno script Python e comandi bash. Lo script in Python (Listing 1) prende in input il file del dataset e si occupa di ordinare i nodi di ogni arco in ordine crescente, i.e. $u < v$. Gli autoanelli vengono ignorati.

```python
import sys

if len(sys.argv) != 2:
    sys.exit()

with open(sys.argv[1]) as f:
    for line in f:
        nodes = line.split()
        if nodes[0] == nodes[1]:
            continue
        print("\t".join(sorted(nodes)))
```

Listing 1: Pyhton script

L'output stream del programma viene dato in input al comando bash sort -u che ordina gli archi ed elimina gli eventuali duplicati (Listing 2). Il risultato viene quindi scritto su file.

```
python3 script.py file_input.txt | sort -u > output_file.txt
```

Listing 2: Comandi Bash

Lo script e i comandi bash impiegano solo pochi secondi di esecuzione per produrre il file di output del formato corretto. Testando con un file da 1,3 GB [6], il risultato viene prodotto in meno di 6 minuti. Il parsing iniziale dell'input permette di velocizzare in maniera sensibile l'operazione di Map in quanto vengono eliminati la metà degli archi presenti in un file per un grafo indiretto con duplicati. Anche l'eliminazione degli autoanelli, che sono comunque inutili per trovare triangoli all'interno di un grafo, permette di ridurre il tempo di esecuzione. É stato scelto di gestire il file come stream (tramite lo script Python) per evitare di gestirlo interamente nella memoria fisica (RAM) e poter elaborare dataset di dimensioni notevoli. Per evitare di creare un file temporaneo con i dati ottenuti dallo script, si è deciso di redirigere direttamente lo stream verso l'applicativo sort. Inoltre, la scelta è ricaduta su sort e non sull'implementazione di un nuovo programma perchè esso è già ottimizzato per la gestione di file di grosse dimensioni tramite parallelizzazione (e.g. –parallel n).

4.2 Avvio del programma

Per avviare il programma bisogna dare i seguenti parametri in input (in ordine):

- Numero di partizioni in cui suddividere il grafo;
- Numero di reducers;
- File di input in HDFS;
- Cartella di output in HDFS;
- Cartella di output nel file system locale.

Eccetto il numero di partizioni, tutti gli altri parametri sono opzionali. Se non specificati, verranno utilizzati quelli di default specificati nella classe *Driver* (Sezione 4.3). Il numero di reducers è settato di default a 1 mentre la cartella di output in HDFS è creata automaticamente nel formato 'yyyyMMdd HHmmss' in base al timestap corrente. Listing 3 e 4 mostrano degli esempi di comandi da terminale per eseguire il programma.

```
[rootquickstart /]# hadoop jar BigData.jar Driver 5
```

Listing 3: Esecuzione con valori di default

```
[rootquickstart /]# hadoop jar BigData.jar Driver 5 1 /user/sunyi/
    ttp/input/general_rel.txt
```

Listing 4: Esecuzione con 3 parametri in input

4.3 Classe *Driver*

La classe *Driver* si occupa di configurare e inoltrare il job che deve essere eseguito dal cluster Hadoop. La classe, oltre a impostare i vari parametri dati in input, imposta la classe Mapper e la classe Reducer per il job, che nel nostro caso sono rispettivamente *Map* e *Reduce*. Alle due classi viene dato in input il numero di partizioni in cui deve essere suddiviso il grafo. La classe *Reduce* produce in output un file, memorizzato nell'HDFS, che elenca tutti i triangoli del grafo e il loro valore per il conteggio. Per ottenere il numero totale dei triangoli bisogna sommare tutti i singoli valori presenti nell'output. Per fare ciò si potrebbe utilizzare un altro processo MapReduce in cascata a quello utilizzato per computare i triangoli. Questo metodo risulta però più oneroso rispetto a una semplice lettura in locale del file, quindi si è preferita quest'ultima soluzione. Il tool si occupa dunque di copiare il file di output dall'HDFS al file system locale, nel nostro caso di Cloudera, e poi di calcolare il numero totale di triangoli. Questo passaggio di copiatura è necessario perchè non è possibile leggere direttamente il file di output quando quest'ultimo si trova nel filesystem di Hadoop.

Il tool genera le seguenti statistiche in output:

- Nome del file in input;
- Numero totale di triangoli del grafo;
- Numero di triangoli di tipo 2 e 3 con relativo valore di conteggio (i.e. 1);
- Numero di triangoli di tipo 1 con relativo valore di conteggio (i.e. $\frac{1}{\rho-1}$);
- Tempo di esecuzione del job MapReduce in formato hh:mm:ss e in millisecondi.

Per calcolare il numero di triangoli totali si è applicata la seguente formula:

$$(lines\_read - not\_ones) + (not\_ones * \frac{1}{\rho - 1})$$

dove *lines_read* indica il numero totale di righe del file di output mentre *not_ones* indica il numero di triangoli che compaiono nel file con valore diverso da 1. Il risultato della prima parentesi è il numero di triangoli di tipo 2 e 3 mentre il risultato della seconda parentesi è il numero di triangoli di tipo 1. Inizialmente veniva eseguita una semplice sommatoria dei valori dei singoli triangoli del file di output ma il risultato non era preciso. La perdita di precisione era dovuta agli arrotondamenti implementati dalle variabili Java, sia di tipo float che di tipo double. Utilizzando la formula di cui sopra invece si ottiene sempre il risultato corretto e preciso. Il tempo di esecuzione del solo job viene calcolato usando il metodo $System.currentTimeMillis$ di Java.

4.4 Classe *Map*

La classe *Map* implementa il mapper per il modello MapReduce. Ogni istanza di *Map* riceve in input una riga del file di input, che rappresenta un arco (u, v) del grafo, e produce in output coppie $\langle (a, b); (u, v) \rangle$ oppure $\langle (a, b, c); (u, v) \rangle$. a, b e c indicano le partizioni che compongono la 2 o 3'-partizione, cioè G_{ab} oppure G'_{abc}. È importante notare che gli indici devono sempre rispettare la relazione $a < b < c$. I due vertici u e v vengono assegnati a una partizione tramite una funzione di hashing $P(v)$ che produce un intero nell'intervallo [0, ρ - 1]. Per l'implementazione si è scelta la funzione modulo per suddividere i nodi. Se l'arco è interno, cioè entrambi i nodi sono nella stessa partizione, allora verranno prodotte $\rho - 1$ coppie in output $\langle (a, b); (u, v) \rangle$, dove $a, b \in [0, \rho - 1]$ soddisfano la relazione $\{P(u), P(v)\} \subseteq \{a, b\}$. Prendiamo per esempio l'arco (4, 8) del nostro esempio (Figura 1). Siccome è interno, $P(4) = P(8) = 0$, il mapper emetterà le 3 coppie $\langle (0, 1); (4, 8) \rangle$, $\langle (0, 2); (4, 8) \rangle$ e $\langle (0, 3); (4, 8) \rangle$. Per gli archi esterni invece viene prodotta un'unica coppia $\langle (a, b); (u, v) \rangle$, dove a e b indicano le due partizioni dei nodi, e $\rho - 2$ coppie $\langle (a, b, c); (u, v) \rangle$ dove $a, b, c \in [0, \rho - 1]$ soddisfano la relazione $\{P(u), P(v)\} \subseteq \{a, b, c\}$. Per esempio il mapper produrrà le coppie $\langle (1, 2); (2, 9) \rangle$, $\langle (0, 1, 2); (2, 9) \rangle$ e $\langle (1, 2, 3); (2, 9) \rangle$ per l'arco (2, 9).

4.5 Classe *Reduce*

La classe *Reduce* implementa il reducer per MapReduce. Ogni istanza della classe riceve in input una coppia $\langle key; valueSet \rangle$, dove *key* è la partizione e *valueSet* è l'insieme dei suoi archi. Supponiamo che un reducer riceva in input la coppia $\langle$(0, 1, 2); (1, 12), (2, 12), (2, 9), (9, 12). Questo significa che deve computare la 3'-partizione G'_{012} che comprende gli archi (1, 12), (2, 12), (2, 9) e (9, 12). Per creare il sottografo e calcolarne i triangoli viene utilizzata la classe *Graph*. Una volta ottenuti i triangoli, il reducer emette coppie $\langle$(u, v, w); value$\rangle$. Come visto precedentemente, *value* vale 1 per i triangoli Type-2 e Type-3, mentre per quelli Type-1 $value = \frac{1}{\rho - 1}$.

4.6 Classe *Graph*

La classe *Graph* viene utilizzata per generare un grafo indiretto dagli archi in input e computarne il numero di triangoli utilizzando un algoritmo in-memory. Il grafo viene rappresentato tramite una lista di adiacenza, implementata nel codice usando un un array bidimensionale di interi (ogni vertice viene denotato da un numero intero positivo). L'indice i indica il nodo che stiamo considerando mentre j specifica il j-esimo vicino del vertice i. I vicini di ogni nodo sono memorizzati nell'array di adiacenza in ordine crescente. Consideriamo per esempio un nodo 1 che è connesso con i vertici 3, 6 e 2 e sia $adjArray$ il nostro array bidimensionale di adiacenza. Per trovare i vicini di 1 bisognerà accedere alla i-esima riga di $adjArray$, con $adjArray[1][0] = 2$, $adjArray[1][1] = 3$, $adjArray[1][2] = 6$. L'ordinamento dei vertici viene fatto per esigenze dell'algoritmo di computazione dei triangoli, compact forward (Sezione 4.6.1), che ha bisogno che i nodi siano ordinati mediante una funzione $\eta()$. Un array bidimensionale risulta la struttura dati più adatta per le nostre esigenze. Rispetto a una matrice di adiacenza richiede meno memoria, presumendo i grafi in input come sparsi. Inoltre una matrice non sarebbe efficiente per elencare i vicini di un vertice. Supponiamo per esempio di dover trovare il primo vicino del nodo $v = 1$. In una matrice dovremmo scorrere in ordine tutta la riga $i = 1$ fino a trovare il primo nodo che è connesso a v, mentre con la lista basta accedere a $adjArray[1][0]$. Per l'implementazione si potrebbe anche usare un array monodimensionale in cui vengono memorizzati tutti i vicini di ogni nodo, uno dopo l'altro. Separatamente si avrebbe bisogno di un array di supporto di indici che specifica, per ogni vertice, dove trovare i suoi vicini all'interno del primo array. Questa metodologia è quella suggerita dall'autore in [2]. Si è però preferito usare una lista di adiacenza implementata con un array bidimensionale poichè il suo utilizzo è più immediato con i due indici e non richiede particolarmente più memoria dei due array separati. Come già discusso precedentemente, nei file di input ogni coppia di nodi compare solo una volta. Ogni volta che riceviamo un arco (*u, v*) quindi, viene generato e aggiunto anche l'arco (*v, u*). Supponiamo di ricevere in input gli archi (1, 12), (2, 12), (2, 9), (9, 12)(G'_{012} di Figura 3). Se dai

nodi in input costruissimo direttamente l'array di adiacenza, esso risulterebbe:

$$
\begin{aligned}
i = 0 &\to \emptyset & 7 &\to \emptyset \\
1 &\to 12 & 8 &\to \emptyset \\
2 &\to 9, 12 & 9 &\to 12 \\
3 &\to \emptyset & 10 &\to \emptyset \\
4 &\to \emptyset & 11 &\to \emptyset \\
5 &\to \emptyset & 12 &\to 1, 2, 9 \\
6 &\to \emptyset
\end{aligned}
$$

Come possiamo vedere alcune righe rimangono completamente vuote perchè non abbiamo il nodo corrispondente a quelle posizioni. Verrebbe quindi allocata una struttura dati più grande del necessario. Per 4 nodi viene creato un array per contenere 13 oggetti. È facile capire che questo problema verrebbe esasperato all'estremo con nodi dal valore sempre maggiore. La soluzione ideale è di rinominare i nodi da 0 a $n' - 1$, dove n' è il numero di vertici. La rinominazione però non viene fatta in maniera casuale ma seguendo un preciso criterio: il nodo con il grado maggiore diventerà il nodo 0, il secondo sarà 1, quello con meno archi sarà $n' - 1$ e così via. In pratica si riordinano i nodi per numero decrescente di vicini. Questo viene fatto perchè l'algoritmo compact forward [2] richiede questa enumerazione dei vertici. Definiamo $d(u)$ come la funzione che ritorna il numero di nodi connessi al nodo u. $\eta()$ è una funzione iniettiva tale che $d(u) > d(v)$ implica che $\eta(u) < \eta(v)$ per tutti i $v, u \in V(G)$. Anche se non è richiesto di rinominare i vertici, basta semplicemente poterli enumerare, è molto più conveniente farlo piuttosto che creare, per esempio, una struttura di supporto che mappi ogni nodo al loro ordinamento su $\eta()$. L'array di adiacenza viene costruito subito dopo aver cambiato nome ai vertici, nel costruttore della classe. Tornando all'esempio precedente, il vertice con grado maggiore risulta 12 (nuovo nodo 0) con 3 archi, a seguire abbiamo 2 (nuovo nodo 1) e 9 (nuovo nodo 2) con 2 archi ciascuno e infine 1 (nuovo nodo 3) che ha un solo vicino. L'array di adiacenza risulta:

$$
\begin{aligned}
i = 0 &\to 1, 2, 3 \\
1 &\to 0, 2 \\
2 &\to 0, 1 \\
3 &\to 0
\end{aligned}
$$

Quando due nodi hanno lo stesso numero di archi, come il nodo 2 e 9 dell'esempio, viene scelto come tie breaker il valore numerico dei nodi.

4.6.1 Algoritmo Compact Forward

L'algoritmo utilizzato per il conteggio dei triangoli nel reducer è il compact forward (Alg. 1), scelto dagli autori in [1]. Lo pseudocodice proposto in [2] è

stato implementato nel metodo *compactForward*. L'idea generale dell'algoritmo è quella di enumerare tutti i nodi in base a una funzione iniettiva η e di trovare, tramite l'array di adiacenza, tutti i triangoli $\Delta(u, v, w)$ dove $\eta(u) > \eta(v) > \eta(w)$. Ricordiamo che la funzione η numera i nodi in ordine decrescente di vicini, in modo tale che se v ha più vicini di u, allora $\eta(v) < \eta(u)$. Le righe 1-2 di Alg. 1 vengono già eseguite nel costruttore della classe *Graph*. Si è preferito fare questo step lì invece che nell'algoritmo vero e proprio perchè risultava superfluo rinominare una prima volta a caso i nodi per creare l'array di adiacenza per poi farlo un'altra volta nel metodo *compactForward*. In Alg. 1 si usa la notazione $\eta(u) > \eta(v)$, ma avendo già rinominato i nodi scriveremo semplicemente $u > v$. Per ogni nodo v del grafo, presi in ordine crescente, si considerano solo i loro vicini u tali che $u > v$ (riga 4) e si cercano, scorrendo l'array di adiacenza, tutti i nodi w tali per cui valgono le seguenti proprietà (righe $5 - 12$):

- $w \in N(u)$;
- $w \in N(v)$;
- $w < v$.

dove $N(u)$ indica i vicini di u. In pratica si stanno cercando tutti i nodi w che siano sia vicini di u che di v, ma con una condizione aggiuntiva, cioè $u > v > w$. L'ordinamento in base al numero di vicini ci permette di analizzare prima i vertici più connessi. Questo risulta vantaggioso perchè essi hanno una maggiore probabilità di formare un triangolo all'interno del grafo. Le altre due condizioni, $u > v$ (riga 4) e $w < v$ (riga 7), vengono poste in modo che ogni triangolo venga trovato una sola volta. Riprendendo il nostro esempio e considerando la 3'-partizione G_{012}, il triangolo formato dai tre vertici 0, 1 e 2 (originariamente nodo 12, 2 e 9 rispettivamente) viene ottenuto solo quando $u = 2$, $v = 1$ e $w = 0$. L'algoritmo quindi ritorna solo $\Delta(2, 1, 0)$ mentre tutti gli altri triangoli, come per esempio $\Delta(2, 0, 1)$ e $\Delta(0, 1, 2)$, non sono considerati validi. Naturalmente abbiamo bisogno che i triangoli ritornati al reducer utilizzino la vecchia nomenclatura piuttosto che i nuovi nomi assegnati. Per fare ciò viene usato il metodo $convertNames$ che, usando una struttura dati ausiliaria, riporta i vertici al loro nome originale. Il reducer ottiene quindi il triangolo $\Delta(9, 2, 12)$ come risultato della computazione.

Algorithm 1: Compact forward algorithm [2]

Input: Array di adiacenza di G
Output: Lista dei triangoli di G

```
1  Numerare i vertici con una funzione iniettiva η() tale che d(u) > d(v)
     implica che η(u) < η(v) per tutti i v, u ∈ V(G)
2  Ordinare l'array di adiancenza in base alla funzione η()
3  foreach v ∈ V(G) presi in ordine crescente di η() do
4      foreach u, η(u) > η(v) ∈ N(v) do
5          u' ← primo vicino di u
6          v' ← primo vicino di v
7          while esiste un vicino di u oppure v non ancora considerato &&
             η(u') < η(v) && η(v') < η(v) do
8              if η(n') < η(v') then
9                  u' ← prossimo vicino di u
10             else if η(n') < η(v') then
11                 v' ← prossimo vicino di v
12             else
13                 output triangolo {u, v, u'}
14                 u' ← prossimo vicino di u
15                 v' ← prossimo vicino di v
16             end
17         end
18     end
19 end
```

5 Testing e Valutazione delle Performance

Correttezza Per testare la correttezza dei risultati calcolati dal tool sviluppato si sono utilizzati una ventina di dataset provenienti da [3], scelti in base ai criteri riportati nella Sezione 4.1. Non si è stati in grado di eseguire file superiori a *Googlewebgraph* perchè il job si interrompeva a causa di risorse non sufficienti. Tutti i risultati ottenuti di conteggio dei triangoli corrispondono a quelli attesi.

Set Up Non avendo a disposizione il cluster universitario, si è utilizzato un computer HP Spectre 360 con 8 GB di RAM e processore Intel i5-5200U a 2.20 GHz x 4. Sul PC è installato Docker con un container Cloudera.

Performance In Tabella 1 sono riportati i dataset significativi utilizzati per le nostre analisi di performance con i relativi dati. Ricordiamo che il numero di archi si riferisce agli archi univoci del grafo, quindi senza ripetizioni di coppie di nodi, il numero di triangoli è calcolato considerando il grafo come indiretto, la dimensione del file si riferisce a quella del file scaricato online. I tempi di

esecuzione dei processi di Map e Reduce sono stati presi dall'output prodotto da Hadoop, visualizzabile al termine dell'esecuzione da terminale (*Total time spent by all map tasks, Total time spent by all reduce tasks*). Il tempo totale di esecuzione del job è preso dalle stime calcolate dalla classe *Driver* (Sezione 4.3). Esso tiene in considerazione anche il tempo per settare il job, distribuire le righe del file di input alle varie istanze di Map, etc. Ogni dataset è stato testato più volte utilizzando un numero variabile di partizioni in cui suddividere il grafo. I valori mostrati nella tabella risultano da una semplice media dei valori ottenuti dalle diverse esecuzioni. N/A indica che i valori non sono disponibili poichè il job non è stato portato correttamente a termine. Questo dipende dal fatto che le computazioni richiedono più risorse di quante ne siano disponibili sul pc.

Analisi dei risultati ottenuti Le conclusioni tratte si basano sui dati generati sul pc descritto sopra. Non avendo a disposizione più macchine, le istanze di Map e Reduce vengono eseguite in successione, una dopo l'altra. Facendo gli stessi test su un sistema distribuito (cluster) si potrebbero ottenere risultati molto diversi. Aumentando il numero di partizioni aumenta anche il numero di coppie $\langle key; valueSet \rangle$ generate in output dal mapper, ma la crescita è lineare ($O(m\rho)$) [1]. Il numero di controlli che il mapper deve fare per creare le suddette coppie ha una crescita quasi cubica in ρ, ($O(m\rho^3)$ [1]), anche se in pratica è molto minore. L'algoritmo compact forward ha complessità $O(m^{\frac{3}{2}})$. Aumentando il numero di partizioni, diminuisce il numero di archi che ogni reducer deve elaborare. In media la dimensione dell'input di ogni reducer è di $O(\frac{m}{p^2})$. Con l'aumentare del numero di partizioni aumenta però anche il calcolo computazionale richiesto per la fase di shuffle e il numero di istanze di reducer, e quindi il tempo di esecuzione. Come notiamo dai dati riportati dalla Tabella 1, all'aumentare del numero di partizioni, aumenta sempre anche il tempo di esecuzione per i processi di Map e Reduce e di conseguenza quelli totali del job. É interessante notare come il processo di Map richiede più tempo del processo di Reduce all'aumentare del numero di partizioni. Questo comportamento si può notare in Tabella 1 nelle prestazioni dei dataset di *EU email communication network, California road network* e *Google web graph* a partire da 20 partizioni in input. Questo è causato dal fatto che aumentando il numero di partizioni ogni reducer riceve meno archi in input da elaborare mentre i mapper hanno più computazioni da svolgere per creare tutte le coppie $\langle key; valueSet \rangle$. Figura 5 e Figura 6 mostrano le variazioni nei tempi di esecuzione rispetto al numero di archi e al numero di partizioni. Nel primo grafico si è utilizzata una scala logaritmica sia per le ascisse che per le ordinate a causa della considerevole differenza di valore delle variabili. Utilizzando la regressione lineare è possibile identificare una retta che descrive la crescita lineare del tempo di esecuzione in relazione al numero di archi, dimostrandone la crescita direttamente proporzionale. La stessa operazione è stata eseguita anche per evidenziare la relazione tra tempi di esecuzione e numero di partizioni. Esempi di regressione lineare sui dati sono mostrati in Figura 7 e Figura 8.

	Statistiche						Tempo di esecuzione (in ms)		
Dataset	Tipo	Nodi	Archi	Triangoli	Dim.file	Partizioni	Map	Reduce	Job
GR-QC collaboration network	Indiretto	5 242	14 496	48 260	351,7 kB	5	3 653	4 466	22 594
						10	4 076	4 860	23 328
						20	4 159	5 075	23 530
						35	4 786	5 581	25 712
Facebook social circles	Indiretto	4 039	88 234	1 612 010	854,4 kB	5	4 136	7 717	27 820
						10	5 019	8 548	28 765
						20	7 077	9 849	32 871
						35	13 875	13 275	43 264
EU email communication network	Diretto	265 214	420 045	267 313	5,0 MB	5	5 912	9 236	31 635
						10	12 419	13 756	40 667
						20	23 919	18 876	57 488
						35	44 759	28 493	87 763
California road network	Indiretto	1 965 206	2 766 607	120 676	87,8 MB	5	31 000	59 321	105 056
						10	75 150	95 984	186 291
						20	199 909	160 874	374 824
						35	N/A	N/A	N/A
Google web graph	Diretto	875 713	5 105 039	13 391 903	75,4 MB	5	45 565	78 734	140 467
						10	126 615	130 171	272 419
						20	326 607	250 832	593 540
						35	N/A	N/A	N/A

Tabella 1: Dataset significativi e relativi dati

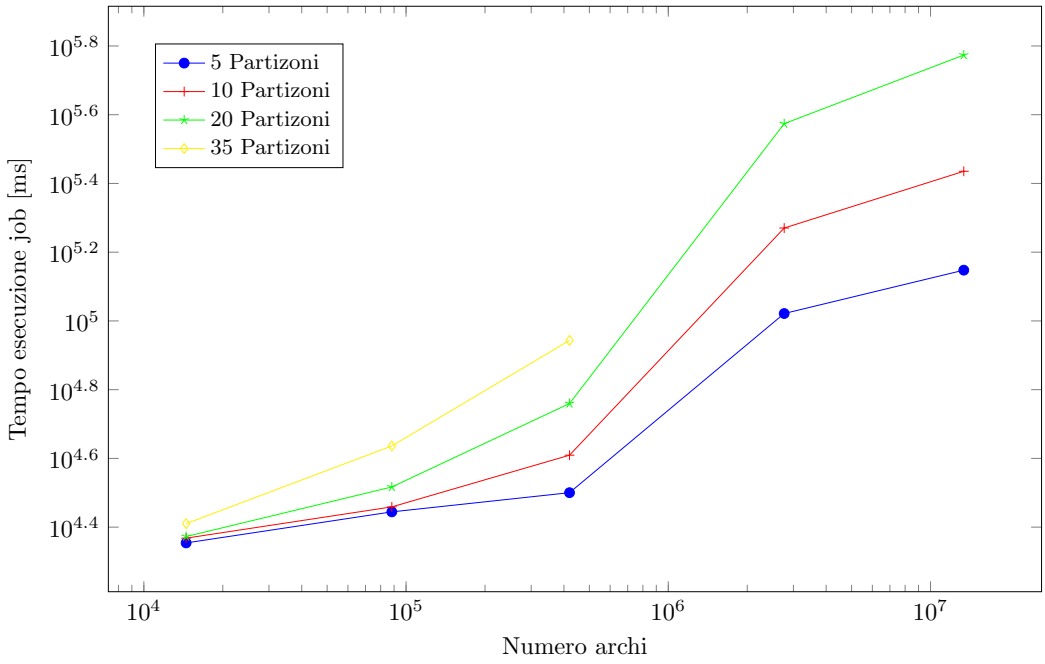


Figura 5: Tempi di esecuzione al variare del numero di archi

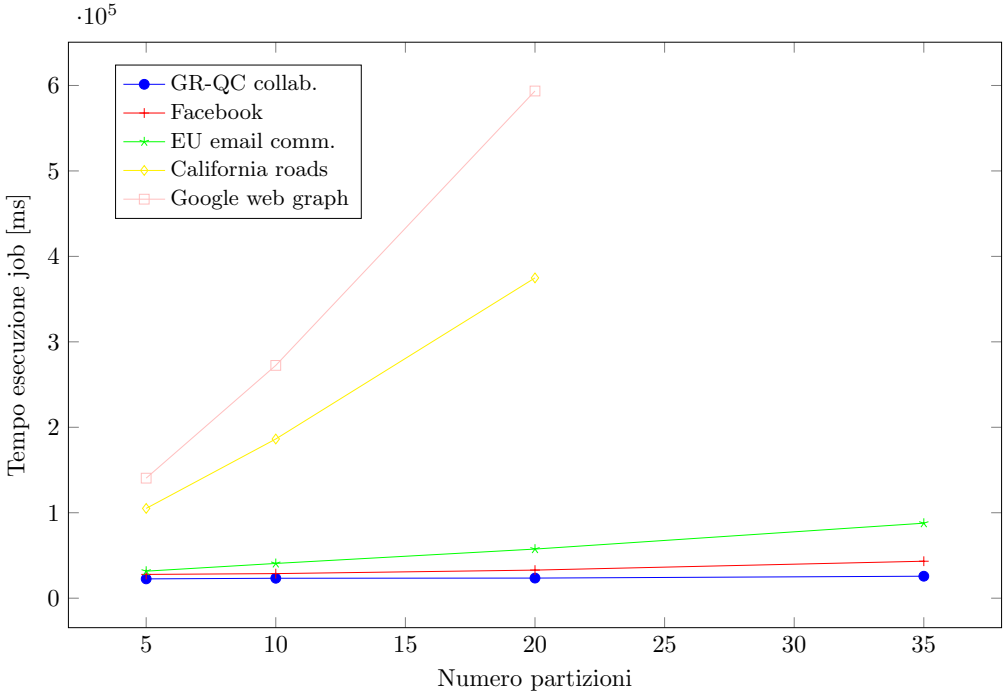


Figura 6: Tempi di esecuzione al variare del numero di partizioni

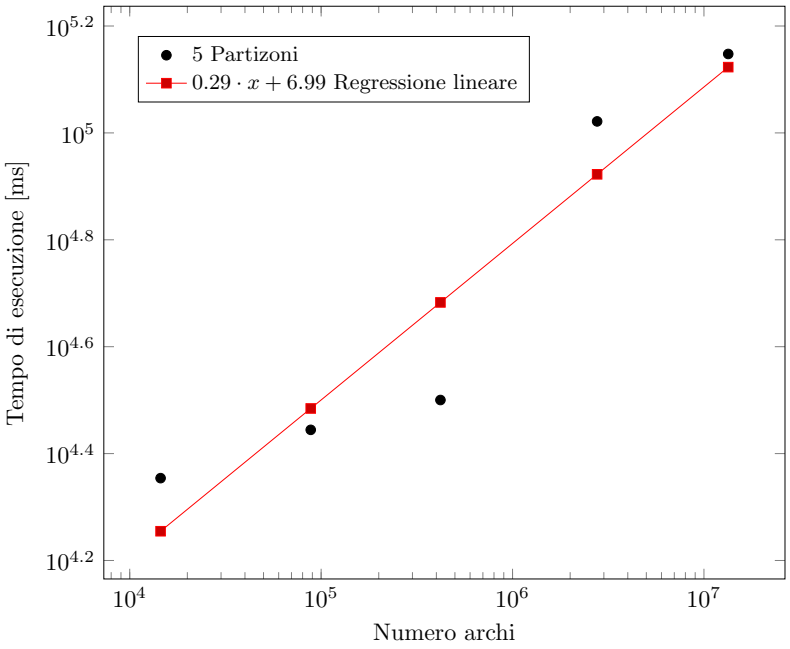


Figura 7: Regressione lineare al variare degli archi

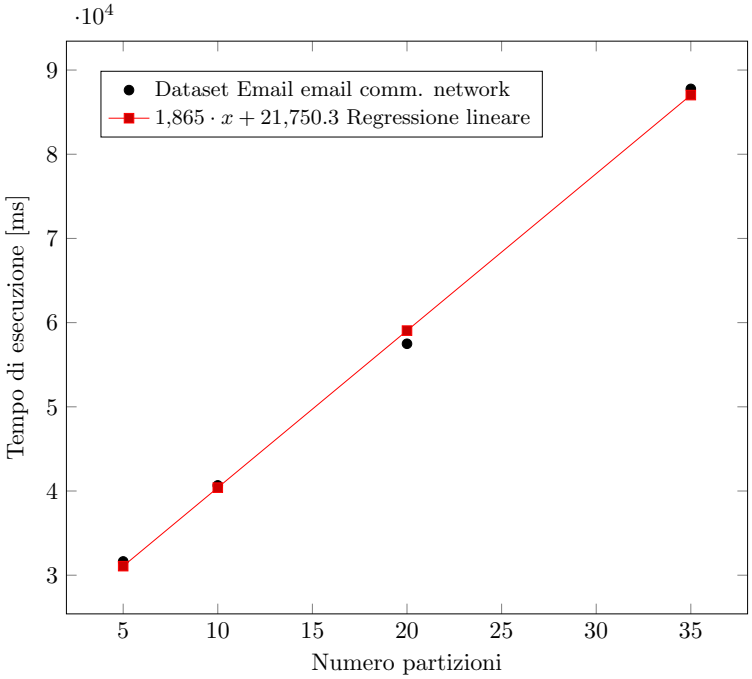


Figura 8: Regressione lineare del dataset in base al numero di partizioni

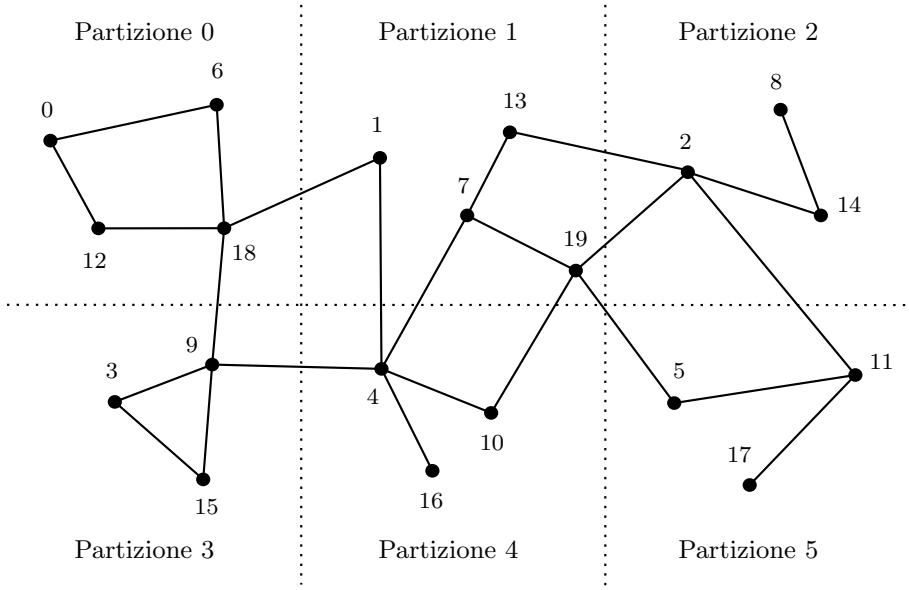


Figura 9: Grafo G' suddiviso in $\rho = 6$ partizioni

6 Possibili estensioni del tool

6.0.1 Algoritmi per il conteggio dei triangoli

Per il nostro tool abbiamo implementato l'algoritmo in-memory compact forward per computare i triangoli all'interno del sottografo assegnato poichè risultava uno dei migliori in termini di tempo e spazio. Naturalmente qualsiasi algoritmo, sia in-memory che non, può essere utilizzato per fare questa operazione e utilizzata al posto di quello scelto. La struttura modulare della classe *Graph* permette di implementarli facilmente, senza doversi preoccupare nè di creare l'array di adiacenza (costruttore) nè di convertire i nodi ai loro nomi originali (funzione $convertNames$).

6.0.2 MapReduce per calcolare quadrati

L'idea degli autori in [1], cioè creare le partizioni e suddividere i triangoli in tipi diversi da computare separatamente, può essere estesa a qualsiasi tipo di struttura ciclica composta da n' nodi. Un triangolo in un grafo non è altro che un ciclo formato da 3 vertici. Supponiamo di volere trovare, invece di triangoli, i quadrati all'interno di un grafo G.

Definition 6.1 (Quadrato)**.** Dato un grafo $G = (V, E)$, un quadrato, denotato con $\square(u, v, w, x)$, è un set di quattro nodi $u, v, w, x \in V$ tali che $(u, v), (v, w)$, $(w, x), (x, u) \in E$.

Oltre alle 2 e 3-partizioni ci servirà definire una 4-partizione.

Definition 6.2 (4-partizione e 4'-partizione)**.** Siano G_i, G_j, G_k e G_l quattro partizioni diverse. Una 4-partizione, denotata da $G_{ijkl} = (V_{ijkl}, E_{ijkl})$, è un sottografo di G su $V_i \cap V_j \cap V_k \cap V_l$. Una 4'-partizione, G'_{ijkl}, è un sottografo di una 4-partizione senza archi interni.

Come per i triangoli, partizionando un grafo possiamo distinguere 4 tipi diversi di quadrato $\square(u, v, w, x)$ in base alla suddivisione dei loro archi.

Definition 6.3 (Tipi di quadrati)**.**

- Type-1. Se il quadrato è formato da soli archi interni, cioè tutti i nodi sono nella stessa partizione, allora esso è di tipo Type-1.
- Type-2. Se due archi sono interni e due sono esterni allora il quadrato è un Type-2. In questo caso avremo i nodi suddivisi in due sole partizioni.
- Type-3. Se il quadrato ha un solo arco interno allora è un tipo Type-3. I nodi sono distribuiti su 3 partizioni diverse.
- Type-4. Se tutti gli archi sono esterni (vertici in 4 partizioni distinte) allora il quadrato è un Type-4.

Prendiamo in considerazione il grafo G' in Figura 9 suddiviso in 6 partizioni. Il quadrato $\square(0, 6, 12, 18)$ è di tipo Type-1, $\square(4, 7, 10, 19)$ e $\square(2, 7, 13, 19)$ sono Type-2, $\square(2, 5, 11, 19)$ è un Type-3 mentre $\square(1, 4, 9, 18)$ è un Type-4. A seguito un piccolo schema riassuntivo che indica in che partizioni ogni tipo di quadrato viene computato e il suo valore per il conteggio.

Tipo quadrato	Partizioni	Valore nel conteggio
Type-1	3-partizione	$\binom{\rho-1}{2}$
Type-2	3-partizione	$\rho - 2$
Type-3	3-partizione	1
Type-4	4'-partizione	1

Tabella 2: Schema riassuntivo di tipi e valori per i quadrati

Analogamente ai triangoli, i quadrati con tutti gli archi esterni vengono calcolati in una 4'-partizione mentre tutti gli altri in una 3-partizione. Per i Type-1 e Type-2 è ridondante calcolarli anche nelle 1 e 2-partizioni. Prendiamo in esame i quadrati della Figura 9:

- $\square(0, 6, 12, 18)$ è un Type-1 ed è computato nelle 3-partizioni G_{012}, G_{013}, $G_{014}, G_{015}, G_{023}, G_{024}, G_{025}, G_{034}, G_{035}, G_{045}$;

- $\square(4, 7, 10, 19)$ è un Type-2 ed è calcolato in $G_{014}, G_{124}, G_{134}, G_{145}$;
- $\square(2, 7, 13, 19)$ è un Type-2 e si trova in $G_{012}, G_{123}, G_{124}, G_{125}$;
- $\square(2, 5, 11, 19)$ è un Type-3 e viene calcolato solo in G_{125};
- $\square(1, 4, 9, 18)$ è un Type-4 ed è computato solo in G'_{0134}.

Per quanto riguarda il MapReduce avremo che:

- Ogni mapper riceve in input un arco (u, v):
  - se l'arco è interno allora vengono generate in output $\binom{\rho-1}{2}$ coppie del tipo $\langle (a, b, c); (u, v)\rangle$ dove $a, b, c \in [0, \rho - 1], a < b < c$ soddisfano la relazione $\{P(u), P(v)\} \subseteq \{a, b, c\}$;
  - se l'arco è esterno allora vengono generate in output $\rho - 1$ coppie $\langle (a, b, c); (u, v)\rangle$ dove $a, b, c \in [0, \rho - 1], a < b < c$ soddisfano la relazione $\{P(u), P(v)\} \subseteq \{a, b, c\}$ e $\binom{\rho-2}{2}$ coppie $\langle (a, b, c, d); (u, v)\rangle$ dove $a, b, c, d \in [0, \rho - 1], a < b < c < d$ soddisfano la relazione $\{P(u), P(v)\} \subseteq \{a, b, c, d\}$.
- Ogni reducer riceve in input una coppia $\langle key; valueSet\rangle$ dove key indica una 3 o 4'-partizione e $valueSet$ contiene la lista degli archi del sotto-grafo. Una volta computati i quadrati verranno prodotte in output le coppie $\langle (u, v, w, x); value\rangle$, dove $value$ dipenderà dal tipo del quadrato come indicato in Tabella 2.

Consideriamo gli archi (7, 19) e (1, 18) del nostro grafo di esempio G'. Per il primo arco avremo che verranno prodotti, dalla classe Map, le coppie:
$\langle (0, 1, 2); (7, 19)\rangle, \langle (0, 1, 3); (7, 19)\rangle, \langle (0, 1, 4); (1, 19)\rangle, \langle (0, 1, 5); (1, 19)\rangle,$
$\langle (1, 2, 3); (1, 19)\rangle, \langle (1, 2, 4); (1, 19)\rangle, \langle (1, 2, 5); (1, 19)\rangle, \langle (1, 3, 4); (1, 19)\rangle,$
$\langle (1, 3, 5); (1, 19)\rangle, \langle (1, 4, 5); (1, 19)\rangle$.
Per (1, 18) invece avremo: $\langle (0, 1, 2); (1, 18)\rangle, \langle (0, 1, 3); (1, 18)\rangle, \langle (0, 1, 4); (1, 18)\rangle,$
$\langle (0, 1, 5); (1, 18)\rangle, \langle (0, 1, 2, 3); (1, 18)\rangle, \langle (0, 1, 2, 4); (1, 18)\rangle, \langle (0, 1, 2, 5); (1, 18)\rangle,$
$\langle (0, 1, 3, 4); (1, 18)\rangle, \langle (0, 1, 3, 5); (1, 18)\rangle, \langle (0, 1, 4, 5); (1, 18)\rangle$.

Riferimenti bibliografici

[1] Ha-Myung Park and Chin-Wan Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi, editors, *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 539–548. ACM, 2013.

[2] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, 2008.

[3] Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data/index.html`.

[4] Social circles: Facebook. `http://snap.stanford.edu/data/ego-Facebook.html`.

[5] California road network. `https://snap.stanford.edu/data/roadNet-CA.html`.

[6] Social circles: Google+. `http://snap.stanford.edu/data/ego-Gplus.html`.