Triangle Counting: conteggio e analisi dei triangoli presenti in grafi di larga scala

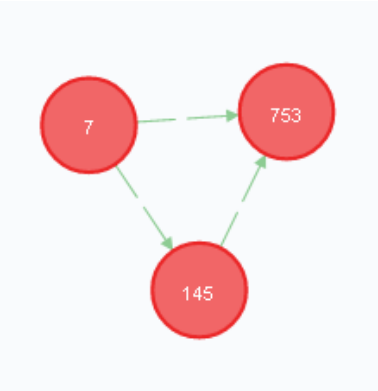
Vincenzo Germinara

Aprile 2021

1 Introduzione

L'algoritmo FFF_k, basato sul paradigma *MapReduce*, è stato sviluppato per risolvere il problema di contare il numero di k-cliques in grafi su larga scala.
Un programma *MapReduce* è composto da diversi round. Ogni round è concettualmente suddiviso in tre fasi consecutive: *map, shuttle e reduce*. I valori di input/output di ogni round , così come i dati intermedi scambiati tra mapper e reducer, vengono memorizzati come coppie $\langle key; value \rangle$. Nella fase map le coppie sono distribuite arbitrariamente tra i mapper e la funzione map definita dal programmatore viene applicata a ciascuna coppia.

Nello sviluppo di questo progetto è stato preso come modello l'algoritmo FFF_k per risolvere il problema Triangle Counting, che è un caso speciale del "k-cliques counting", ovvero si tratta del caso in cui si considera k=3.
L'algoritmo Triangle Counting conta il numero di triangoli presenti nel grafo. Un triangolo è un insieme di tre nodi in cui ogni nodo ha un arco che lo collega agli altri due.



Esempio di triangolo

2 Triangle Counting

L' algoritmo sviluppato per il Triangle counting, così come l'algoritmo FFF_k, fa uso del principio *total order* $\prec$ per decidere quale nodo è responsabile per il conteggio del triangolo. In particolare, si ritiene come *nodo responsabile* per il conteggio il più piccolo nodo che fa parte del triangolo.

L'algoritmo FFF_k lavora su tre round:

Map 1	**Reduce 1**
input $\langle (u,v)\,;0\,\rangle$ **if** $u \prec v$ **then** emit $\langle u; v\,\rangle$	input $\langle u; \Gamma^+(u)\,\rangle$ **if** $\lvert\Gamma^+(u)\rvert \geq k-1$ **then** emit $\langle u; \Gamma^+(u)\,\rangle$
Map 2	**Reduce 2**
input $\langle u; \Gamma^+(u)\,\rangle$ or $\langle (u,v)\,;0\,\rangle$ **if input of type** $\langle (u,v)\,;0\,\rangle$ and $u \prec v$ **then** emit $\langle (u,v)\,;\$\,\rangle$ **if input of type** $\langle u; \Gamma^+(u)\,\rangle$ **then** for each $x_i, x_j \in \Gamma^+(u)$ s.t. $x_i \prec x_j$ do emit $\langle (x_i,x_j)\,;u\,\rangle$	input $\langle (x_i,x_j)\,;\{u_1,...,u_k\} \bigcup \$\,\rangle$ **if** input contains \$ **then** emit $\langle (x_i,x_j)\,;\{u_1,...,u_k\}\,\rangle$
Map 3	**Reduce 3**
input $\langle (x_i,x_j)\,;\{u_1,...,u_k\}\,\rangle$ **for** $h \in [1,k]$ **do** emit $\langle u_h; (x_i,x_j)\,\rangle$	input $\langle u; G^+(u)\,\rangle$ let $q_{u,k-1}$ = number of $(k-1)$-cliques in $G^+(u)$ emit $\langle u; q_{u,k-1}\,\rangle$

L'algoritmo che è stato implementato per il Triangle counting, rispetto al FFF_k differisce nella fase finale *Map3*, in quanto si vuole risolvere il problema di contare il numero dei triangoli per ogni nodo e non il numero totale di clique.
Inoltre, a differenza di quanto proposto nel paper "Clique counting in MapReduce: theory and experiments", tutte le implementazioni sono state realizzate

in Java utilizzando Spark invece di Hadoop e sono stati utilizzati data set di grafi diretti.
Come data set iniziale è stato utilizzato un grafo reale "p2p-Gnutella08" preso dalla libreria di grafi SNAPA, si tratta di un grafo diretto di 6301 nodi e 20777 archi presi dal network *Gnutella*.
I dati in questione sono stati forniti in formato ".txt", quindi dopo aver caricato i dati in memoria, é stata utilizzata la classe "EstraiNodi", che fa parte della fase *Map1*. La classe "EstraiNodi", dato in input la stringa contenente la coppia di nodi, estrae i valori "FROM" e "TO" di ogni arco in una JavaRDD di elementi di tipo *Edge*.
In seguito, è stata creata una JavaPairRDD avente come chiave la coppia (FROM;TO) e come valore "0" per poi lavorare sulle chiavi.
Le chiavi, che rappresentano le coppie di nodi per ogni arco, sono state ordinate secondo il principio *total order* $\prec$ con l'utilizzo della classe "ExtractOrderedEdges2", che per ogni arco (x;y) restituisce la coppia $\langle x, y \rangle$ tale che $x \prec y$. Queste coppie vengono salvate in una nuova JavaPairRDD $\langle String, String \rangle$ dEdgesOrdered2. A questa JavaPairRDD viene applicata la funzione *distinct()* per considerare ciascun arco una sola volta, poiché tra due nodi è possibile avere due archi con direzione opposta, che però una volta ordinati si ottiene due volte lo stesso arco.

Nella prima fase di *Reduce1*, presi in input gli archi ordinati "dEdgesOrdered2", attraverso la funzione *reduceByKey()*, per ciascun nodo di partenza è stato trovato il relativo sottoinsieme dei nodi di arrivo adiacenti, e infine, attraverso la funzione *filter()* sono state conservate solo le coppie $\langle u; \Gamma^+(u) \rangle$ tali che $|\Gamma^+(u)| \geq 2$ poiché, come detto in precedenza, un triangolo è un insieme di tre nodi.

La fase *Map2* prevede due operazioni distinte:

- nella prima le coppie di nodi per ogni arco, sono state ordinate secondo il principio total order $\prec$ con l'utilizzo della classe "ExtractOrderedEdges", che per ogni arco (x_i, x_j) restituisce la coppia $\langle (x_i, x_j) ; \$ \rangle$ tale che $x_i \prec x_j$
- nella seconda operazione, con l'utilizzo della classe "ExtractPairs2", dato in output la JavaPairRDD che ha come chiave il nodo di partenza u e come valore il sottoinsieme di nodi di arrivo adiacenti $\Gamma^+(u)$, per ogni nodo di partenza u mi ricavo tutte le possibile coppie di nodi adiacenti, tenendo conto del principio total order $\prec$, e ottengo una nuova JavaPairRDD che ha come chiave la coppia di nodi adiacenti ad u e come valore u

Nella fase successiva *Reduce2*, per ogni coppia di nodi adiacenti (x_i, x_j) si cerca l'insieme dei *nodi responsabili* $\{u_1, ..., u_k\}$, con l'utilizzo della funzione *reduceByKey()*, in modo che, successivamente per ogni coppia di nodi adiacenti si controlla una sola volta se è presente un arco che le colleghi. Infine, vengono conservate solo le coppie $\langle (x_i, x_j) ; \{u_1, ..., u_k\} \rangle$ tali che tra x_i e x_j esiste un arco, tale condizione è stata verificata con un'operazione di JOIN tra la Ja-

vaPairRDD $\langle (x_i, x_j) ; \{u_1, ..., u_k\} \rangle$ e la JavaPairRDD $\langle (x_i, x_j); \$ \rangle$. Un JOIN tra due JavaPairRDD $\langle K1, V1 \rangle e \langle K1, V2 \rangle$ restituisce solo le coppie delle due JavaPairRDD che hanno la stessa chiave in nuova JavaPairRDD $\langle K, (V1, V2) \rangle$

In seguito, nella fase *Map3*, processiamo tutte le *tuple* contenti $\langle (x_i, x_j) ; \{u_1, ..., u_k\} \rangle$. Per ogni *nodo responsabile* $u_k \in \{u_1, ..., u_k\}$, attraverso la classe "EstraiNodiResponsabili", si ottiene una nuova JavaPairRDD avente come chiave il *nodo responsabile* u_k e come valore la coppia di nodi adiacenti (x_i, x_j). In questo modo abbiamo ottenuto tutte le triple di nodi che formano i triangoli presenti nel grafo.

Nell'ultima fase, la fase *Reduce3* che viene implementata in modo diverso rispetto all'algoritmo FFF_k, si processano tutti i *nodi responsabili* di ciascun triangolo, ottenuti tramite la funzione *keys()* applicata alla JavaPairRDD ottenuta nella fase precedente. Ciascun *nodo responsabile* andrà ad alimentare, sotto forma di chiave, una nuova JavaPairRDD a cui assoceremo il valore 1. Infine, si raggruppano tutte le coppie $\langle u_k, 1 \rangle$ che hanno come chiave lo stesso nodo "responsabile"u_k, attraverso la funzione reduceByKey(), e si sommano i valori 1. Si ottiene cosi una nuova JavaPairRDD$, Integer \rangle$ dNodiresponsabili4 che avrà come chiave il nodo "responsabile" u_k e come valore il numero di triangoli di cui è responsabile.

E per ultimo è stato calcolato il numero totale dei triangoli presenti nel grafo considerato. Da dNodiResponsabili4, attraverso la funzione *values()*, per ogni coppia $\langle NodoResponsabile, numtriangoli \rangle$ sono stati estratti il numero dei triangoli, per poi essere sommati attraverso la funzione *reduce()*.

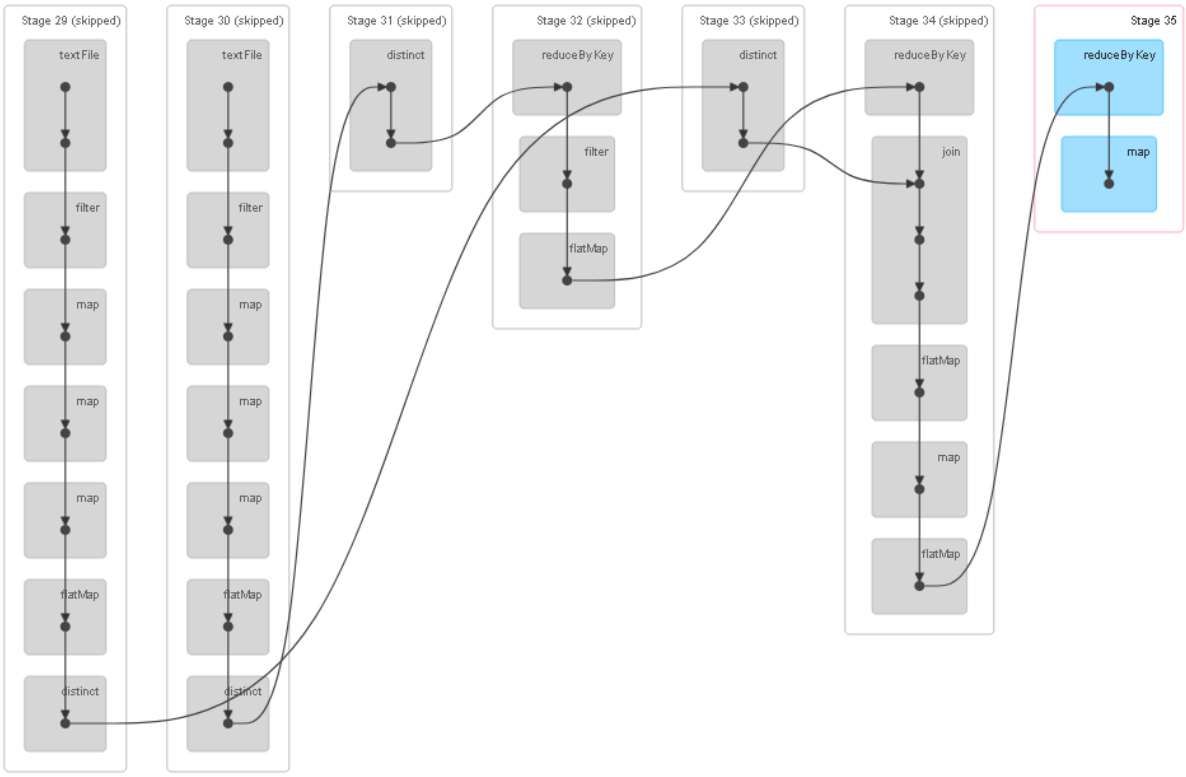
3 Analisi dell'algoritmo

L'algoritmo è stata eseguito su una macchina con 4 core, e quindi si può considerare come un sistema distribuito con 4 nodi.
Le operazione implementate su Java sono state divise in 8 *job* e sono state inviate al framework Spark per essere eseguita. Ogni *job* può essere suddiviso in *stage*, e in ogni stage possiamo avere che il data set viene suddiviso in più pezzi e ciascun pezzo viene processato in parallelo. In questo caso i job sono stati divisi in un totale di 14 stage.

▾ Completed Jobs (8)

Job Id ▾	Description
7	reduce at CliqueCounting.java:213 reduce at CliqueCounting.java:213
6	take at CliqueCounting.java:201 take at CliqueCounting.java:201
5	take at CliqueCounting.java:182 take at CliqueCounting.java:182
4	take at CliqueCounting.java:169 take at CliqueCounting.java:169
3	take at CliqueCounting.java:144 take at CliqueCounting.java:144
2	take at CliqueCounting.java:134 take at CliqueCounting.java:134
1	take at CliqueCounting.java:126 take at CliqueCounting.java:126
0	take at CliqueCounting.java:110 take at CliqueCounting.java:110

Nell'immagine seguente, invece, vengono riportato i blocchi d' istruzioni eseguiti in ogni stage per ottenere il numero complessivo dei triangoli presenti nel grafo.



I job che hanno richiesto più tempo sono il job 4 e il job 6. Nel job 4 vengono create due JavaPairRDD, la prima $\langle A; Bordinati, \$ \rangle$ e la seconda $\langle A; B, NodiResponsabili \rangle$ per poi eseguire l'operazione di *JOIN*, mentre nel job 6 a ogni nodo responsabile vengono associate le coppie di nodi che contribuiscono a formare un triangolo. Quindi ci si aspetta che le prestazioni dipendano dal numero di archi e dal numero di triangoli presenti nel grafo.
Per verificare le prestazioni del codice implementato, abbiamo eseguito il codice per diversi grafi reali diretti di diverse dimensione, presi dalla libreria di grafi SNAPA.

Nella tabella vengono riportati le dimensioni dei vari grafi considerati e i tempi di esecuzione dell'algoritmo:

Grafo	Nodi	Archi	Triangoli	Tempo
p2p-Gnutella08	6'301	20'777	2'383	3 sec
p2p-Gnutella31	62'586	147'892	2'024	12 sec
amazon0302	262'111	1'234'877	717'719	1,8 min
amazon0312	400'727	3'200'440	3'686'467	19,6 min

Come possiamo vedere i tempi di esecuzione sono ottimi per i primi 3 grafi, nonostante le dimensioni molto diverse. Mentre nell'ultimo caso si ha un risultato peggiore poiché, come si è detto in precedenza il job 4 è quello che richiede più tempo e in questo caso essendo il numero di nodi molto elevato si necessita molto più tempo per la verifica della presenza o meno degli archi (8,6 min). E inoltre, in questo grafo abbiamo anche un numero molto più elevato di triangoli. (7,6 min).
In tutti e quattro i casi sono stati utilizzati al più 2 core.

4 Analisi con Neo4j

Per continuare l'analisi dei triangoli presenti nel grafo è stato fatto uso anche di Neo4j,
Neo4j è un database a grafo NoSQL. Viene introdotto nel 2000 per superare le limitazioni dei DBMS relazionali nel modellare relazioni che si mappano bene su grafi, che nel caso dei DMBS richiederebbero JOIN ricorsivi.
In Neo4j sono stati prima importati tutti i nodi presenti nel grafo reale "p2p-Gnutella08" per poi creare gli archi ordinati tra i nodi.
Dopo aver salvato in memoria i *nodi responsabili* contenuti in dNodiResponsabili2 come chiave, in Neo4j ad ogni *nodo responsabile* è stata aggiunta la proprietà *nodoResponsabile=True*, per poi, attraverso una *query*, ottenere quanti e quali sono i nodi che sono considerati "responsabili".
Poiché l'obiettivo è quello di analizzare i triangoli presenti nel grafo, dopo aver ottenuto le triple di nodi che formano i triangoli, ovvero la JavaPairRDD dNodiResponsabili2 avente come chiave il *nodo responsabile* u_k e come valore la coppia di nodi adiacenti (x_i, x_j), è stata salvata in memoria una copia($List\langle Tuple2\langle String, String\rangle\rangle$ *archi*). In seguito su Neo4j, ad ogni nodo appartenente ad un triangolo è stata aggiunta una nuova *label* contenente una stringa identificativa del triangolo a cui appartiene. Attraverso delle *query* sulle *label*, è stato possibile individuare quanti e quali sono i nodi che non hanno partecipato ad alcun triangolo, quali sono i nodi che hanno partecipato a più di 100 triangoli e quali e quanti sono i nodi che hanno partecipato un numero di triangoli pari al numero massimo di volte che un nodo ha partecipato. Infine, si è voluto vedere quali sono i triangoli a cui hanno partecipato i nodi con un numero di triangoli pari al numero massimo.