TRIANGLE COUNTING IN UN GRAFO UTILIZZANDO IL PARADIGMA MAPREDUCE

A PREPRINT

Luigi Barba
Università degli Studi di Roma la Sapienza
Laurea Magistrale Scienze Statistiche

Francesco Guglielmino
Università degli Studi di Roma la Sapienza
Laurea Magistrale Scienze Statistiche

November 20, 2020

1 Introduzione

Il problema di contare il numero di sottografi di dimensione relativamente piccola in grafi di grande dimensione ha attirato molte attenzioni da parte della comunità scientifica negli ultimi anni. Difatti il conteggio di triangoli - o, più generalmente, di cricche (in inglese, *cliques*) - ha molte applicazione nei più svariati campi, dall'informatica (spam detection o social network analysis) alla biologia. Il problema in questi casi è il costo computazionale: difatti il conteggio di sottostrutture in grafi molto grandi può richiedere molti calcoli e, quindi, tanto tempo per essere portato a termine.

In questo progetto abbiamo implementato - in Java e Spark - un algoritmo MapReduce che aggira questo problema definendo una relazione d'ordine all'interno del grafo. Quest'algoritmo - chiamato FFF_k - è preso dal paper di Irene Finocchi, M. Finocchi e E. G. Fusco ***Clique Counting in MapReduce: theory and experiments (2015)*** e verrà utilizzato più specificatamente per il calcolo del numero di triangoli all'interno del grafo preso in considerazione.

L'algoritmo viene applicato su grafi non- direzionati: nel caso in cui il grafo fosse direzionato, basta preprocessarlo per renderlo non-direzionato (tale procedura non modifica il numero di triangoli presenti all'interno del grafo originale).

2 Preliminari Matematici

Prima di addentrarci nell'analisi più strettamente algoritmica è bene introdurre dei concetti (matematici) che saranno fondamentali per i nostri obiettivi.

Per prima cosa, utilizzeremo il simbolo q_k per indicare il numero di k-cricche all'interno del grafo: più specificatamente quindi q_3 sarà il numero di triangoli e q_2 il numero di archi.

$\Gamma(u)$ sarà l'intorno del nodo u ovvero l'insieme dei nodi collegati ad u tramite un arco e $d(u) = |\Gamma(u)|$ (grado di u).

Definiamo inoltre una relazione d'ordine totale sui nodi di G: se $x, y \in V(G)$, allora $x \prec y$ se $d(x) < d(y)$ o se $d(x) = d(y)$ e $x < y$.

Denotiamo inoltre con $\Gamma^+(u)$ l'intorno superiore di u, ovvero l'insieme dei nodi $x \in \Gamma(u)^+$ tali che $u \prec x$. Simmetricamente, $\Gamma^-(u) = \Gamma(u) \setminus \Gamma^+(u)$.

Infine, dati due grafi $G(V, E)$ e $G_1(V_1, E_1)$, G_1 è un sottografo di G se $V_1 \subseteq V$ e $E_1 \subseteq E$. G_1 è un sottografo indotto di G se, oltre ad essere un sottografo, per ogni $u, v \in V_1$, si ha che $(u, v) \in E_1$ **se e solo se** $(u, v) \in E$. Il sottografo indotto dall'intorno superiore $\Gamma^+(u)$ è denotato con $G^+(u)$.

Come precisazione finale, i nodi saranno etichettati con il loro grado: questo sarà molto utile ai fini della riduzione del costo computazionale.

3 L'algoritmo

Algorithm 1 : FFF_k

Map 1: input $\langle (u,v); \emptyset \rangle$
  if $u \prec v$ **then** emit $\langle u; v \rangle$

Reduce 1: input $\langle u; \Gamma^+(u) \rangle$
  if $|\Gamma^+(u)| \geq k-1$ **then** emit $\langle u; \Gamma^+(u) \rangle$

Map 2: input $\langle u; \Gamma^+(u) \rangle$ or $\langle (u,v); \emptyset \rangle$
  if input of type $\langle (u,v); \emptyset \rangle$ 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 $\langle (x_i, x_j); u \rangle$

Reduce 2: input $\langle (x_i, x_j); \{u_1, \ldots, u_k\} \cup \$ \rangle$
  if input contains $ **then**
    emit $\langle (x_i, x_j); \{u_1, \ldots, u_k\} \rangle$

Map 3: input $\langle (x_i, x_j); \{u_1, \ldots, u_k\} \rangle$
  for $h \in [1, k]$ **do**
    emit $\langle u_h; (x_i, x_j) \rangle$

Reduce 3: input $\langle u; G^+(u) \rangle$
  let $q_{u,k-1}$ = number of $(k-1)$-cliques in $G^+(u)$
  emit $\langle u; q_{u,k-1} \rangle$

Figure 1: L'algoritmo secondo la filosofia MapReduce

L'idea è di utilizzare la relazione totale $\prec$ per decidere quale nodo di un dato triangolo Q è responsabile del suo conteggio, ovvero evitiamo di ricontare due o più volte lo stesso triangolo.

La strategia è quella di dividere il grafo in vari sottografi, sfruttando tale frammentazione per calcolare agevolmente il numero di triangoli. Più specificatamente useremo come "partizione" del grafo i sottografi $G^+(u)$ indotti da $\Gamma^+(u)$, questo per ogni nodo u del grafo, e conteremo dunque i triangoli a partire da ogni sottografo in modo indipendente.

L'algoritmo si può idealmente dividere in 3 rounds: essi sono facilmente traslabili in un algoritmo MapReduce.

- **Round 1:** (*calcolo degli intorni superiori*)

  In questo round il nostro obiettivo principale è quello di "mappare" ogni nodo u in $\Gamma^+(u)$ e di fare un reduce sulla cardinalità degli intorni superiori. Il calcolo di $\Gamma^+(u)$ è reso possibile dall'etichetta di ogni nodo relativa al grado: difatti, a partire da essa, è banale stabilire se $x \prec y$. Dunque, nel linguaggio di MapReduce, il map manda l'arco (x, y) in $\langle x; y \rangle$ se $x \prec y$, e il reduce utilizza la chiave x per eliminare le coppie $\langle x; \Gamma^+(x) \rangle$ tali per cui $|\Gamma^+(x)| \leq 2$.

- **Round 2:** (*preliminari per il calcolo dei sottografi*)

  L'obiettivo di questo round è quello di associare ogni nodo arco (x, y) con l'insieme dei nodi u tali per cui $G^+(u)$ contiene (x, y). Questo passo è fondamentale per il calcolo dei sottografi indotti $G^+(u)$, utilizzati a loro volta per il calcolo finale dei triangoli nel grafo.

  In tale round la tupla $\langle u; \Gamma^+(u) \rangle$ viene "mappata" (utilizzando il gergo di MapReduce) in $\langle (x, y); u \rangle$, per ogni coppia (x, y) che appartiene al prodotto cartesiano di $\Gamma^+(u) \times \Gamma^+(u)$ tale che $x \prec y$. Inoltre vi è un secondo map, che produce, per ogni arco (x, y) tale per cui $x \prec y$, la coppia $\langle (x, y); \$ \rangle$. Tale map sarà utile nel secondo reduce dell'algoritmo, dove si dovranno cercare solo le coppie di nodi che formano gli archi del grafo in relazione $\prec$.

- **Round 3:** (*calcolo degli archi negli intorni superiori del grafo*)

  Per ogni nodo u conteremo il numero dei triangoli cui u contribuisce utilizzando $\prec$ e facendo leva sul numero di archi presenti in $G^+(u)$.

  Il terzo (e ultimo) map riceve in input la chiave (x, y) e emette la coppia $\langle u; (x, y) \rangle$ per ogni nodo u tale per cui $G^+(u)$ contiene (x, y). Il terzo e ultimo reduce riceverà dunque l'intera lista degli archi in $\Gamma^+(u)$, dunque è possibile ricostruire $G^+(u)$; dunque siamo in grado di calcolare (localmente) il numero di archi nei sottografi indotti dagli intorni superiori e, di conseguenza, trovare il numero dei triangoli per cui u è responsabile.

4 L'algoritmo MapReduce e l'implementazione in Java

L'implementazione dell'algoritmo è stata realizzata utilizzando il paradigma MapReduce, con Java e Spark, su un sistema distribuito.

L'algoritmo è stato in prima istanza applicato a un grafo non-direzionato preso dalla *Stanford Large Network Dataset Collection* (SNAP) chiamato loc-Gowalla, il quale possiede 196591 nodi e 950327 archi. Tale grafo è basato sul social network Gowalla: i suoi nodi rappresentano gli utenti e gli archi l'amicizia che lega due utenti.

- **Round 0:** *Pulizia e pre-processing del grafo*

  Per prima cosa salviamo tale grafo utilizzando una ***JavaPairRDD<Integer,Integer>*** in Java, facendo parsing sui nodi e dividendoli utilizzando il comando $split(" ")$. Inoltre, come suggerito dal paper, etichettiamo ogni nodo con il proprio grado; questo è stato fatto calcolando in prima battuta il grado di ogni nodo del grafo tramite un MapReduce, che ha come chiave i nodi, producendo una ***JavaPairRDD<Integer,Integer>*** con chiave i nodi u e valore il grado $d(u)$. Successivamente creiamo una struttura $(u, (v, d(u)))$ facendo un join con u in chiave. Scambiando u con v rieseguiamo il join con la struttura $(nodo, grado)$ e otteniamo il grafo avente i nodi etichettati con il loro grado facendo una semplice operazione di pulizia tramite il metodo *aggiustaCoppie*, che richede un mapToPair.

  La struttura dati ottenuta è dunque una ***JavaPairRDD<Tuple2<Integer, Integer>, Tuple2<Integer, Integer»***, dove la prima tupla ha come chiave il primo nodo x dell'arco (x, y) e come valore il suo grado e - in modo analogo - la seconda possiede y e $d(y)$.

- **Round 1: Map 1**

  Filtriamo gli archi secondo l'ordinamento totale descritto dal paper secondo due condizioni:

  -Archi il cui nodo di arrivo ha grado maggiore di quello di partenza.

  -Archi che hanno nodi con grado uguale ma label del secondo nodo maggiore rispetto alla label del primo.

  L'operazione è eseguita in Java utilizzando due filter diversi sulla JavaPairRDD ottenuta dopo il Round 0. Le due ***JavaPairRDD<Tuple2<Integer, Integer>, Tuple2<Integer, Integer»*** derivanti dai filter verrano poi unite con un comando $union$: questo è il risultato dato dal map 1. Dunque abbiamo una struttura $\langle (u, d(u)), (v, d(v)) \rangle$ tale che $u \prec v$.

  Osserviamo come l'etichetta dei nodi sia di fondamentale importanza per il successo del primo map: senza di essa sarebbe molto più complicato stabilire le relazioni $u \prec v$.

- **Round 1: Reduce 1**

  Raggruppiamo il precedente risultato per chiave ottendendo $\Gamma^+(u)$. Questa idea è applicata in Java utilizzando *map1.groupByKey()*, ottenendo dunque una struttura dati di tipo ***JavaPairRDD<Tuple2<Integer,Integer>, Iterable<Tuple2<Integer,Integer»>***. Per concludere il reduce 1 basta filtrare le coppie $\langle u; \Gamma^+(u) \rangle$ tali per cui $|\Gamma^+(u)| \geq 2$, ricordando che con u s'intende la coppia formata dalla label identificativa del nodo e dal suo grado.

- **Round 2: Map 2**

  Facciamo il map sugli archi restituendo $\langle (u, v), \$ \rangle$ se $u \prec v$, altrimenti $\langle (u, v), 0 \rangle$: tale map è realizzato creando una nuova classe, chiamata *Dollar*, che opera sul grafo pre-processato; questo è il primo dei due map che costituiscono il secondo map. Il risultato è chiamato map2-1.

  Adesso prendiamo l'output del primo reduce nella forma $\langle u; \{x_1, x_2, ..., x_n\} \rangle$ dove le x rappresentano i vicini di u con grado maggiore ed estraiamo le differenti coppie. Vogliamo quindi ottenere $\langle (x_1, x_2); u \rangle$, $\langle (x_1, x_3); u \rangle$, $\langle (x_2, x_3); u \rangle$ etc..., ovvero tutte le coppie ottenibili da $\Gamma^+(u)$ tali per cui $x_i \prec x_j$. Per fare ciò scriviamo un metodo, *estraiCoppiePlus*, che ritorna una ***JavaPairRDD<Tuple2<Tuple2<Integer, Integer>, Tuple2<Integer, Integer», Tuple2<Integer, Integer»***, chiamata *coppie*, da una ***JavaPairRDD<Tuple2<Integer,Integer>, Iterable<Tuple2<Integer,Integer»>***. Terminiamo dunque il *map2* raggruppando per chiave u i nodi che formano $G^+(u)$. tramite un *groupByKey* sulla variabile *coppie* e facendo un *join* con *map2-1*: tale operazione è fatta per identificare le coppie di nodi che formano effettivamente un arco.

- **Round 2: Reduce 2** Il prossimo passo consiste nel filtrare le coppie di nodi appartenenti al neighborhood dei nodi $\{u_1, \ldots, u_n\}$ tenendo solo quelle che costituiscono un arco (u, v) all'interno di V tale che $u \prec v$. Partendo dall'RDD $\langle (u, v), \$ \rangle$ contenente in chiave gli archi del grafo che soddisfano tale proprietà e la stringa "$" in valore, mediante un join (che nella sua espressione base filtra solo gli elementi appartenenti all'intersezione delle due RDD) con l'RDD ottenuta nel passo precedente, raggiungiamo l'obiettivo. Adesso basta semplicemente riordinare l'RDD con un *mapToPair* nella forma $\langle (xi, xj), \{u1, ...uk\} \rangle$ togliendo il "$" dal momento che ha esaurito la sua funzione.

- **Round 3: Map 3** Nel round 3 avverrà il conteggio dei triangoli. Prima di ciò occorre scomporre le righe dell'RDD di partenza in più righe: vogliamo che ciascun elemento di $\{u1, \ldots un\}$ venga spostato da valore a chiave, mentre la coppia di nodi (x_i, x_j) a cui è associato venga messo in valore. Per far ciò ci avvaliamo di un *flatMapToPair* che applica un metodo con il tale obiettivo, *terzoMap()*. Inoltre, avendo adesso di nuovo i nodi ui in chiave, possiamo raggruppare mediante *groupByKey* e ottenere l'RDD $\langle u; G^+(u) \rangle$ dove $G^+(u)$ è il sottografo indotto dal nodo u con gli archi (x_i, x_j) appartenenti ai nodi di $\Gamma^+(u)$. In altre parole, ogni coppia di nodi di $G^+(u)$ è un lato di un triangolo nel grafo di partenza. L'ultimo Reduce, quindi, ha lo scopo di estrarre da ogni riga dell'RDD precedentemente ottenuta il nodo u e la dimensione di $G^+(u)$, che è il numero di triangoli dei quali u è "responsabile".

- **Round 3: Reduce 3** Per concludere, contiamo i triangoli estraendo la taglia di ciascun sottografo, sommando tutti i valori attraverso un semplice reduce e stampando a schermo il risultato. Terminato l'algoritmo, riprendiamo immediatamente il tempo di esecuzione e, per calcolare il tempo totale impiegato, gli sottraiamo il valore memorizzato prima dell'esecuzione. Convertiamo la misura da millisecondi a minuti e stampiamo entrambe le misure.

4.1 Pre-processing dei vari grafi

Nel caso di loc-Gowalla il grafo è già non-direzionato, ma - ovviamente - non tutti i grafi lo sono.

L'operazione di pre-processing è essenziale per la buona riuscita dell'algoritmo: ad esempio, nel caso da noi analizzato del dataset di Google - web-Google - abbiamo dovuto rendere il grafo non-direzionato tramite vari $mapToPair$ e $union$, creando dei duplicati dei nodi (ad esempio, se l'arco $(2, 1)$ è presente nel grafo, allora facciamo "comparire" anche $(1, 2)$). In seguito abbiamo eliminato i duplicati tramite un reduce sulla chiave data dalle coppie di nodi (x, y) presenti nella struttura dati derivante dall'operazione di $union$ utilizzando un $groupByKey$.

Il caso di com-Youtube è stato invece più semplice: difatti il grafo aveva già una struttura "implicitamente" indiretta, e quindi lo abbiamo reso simile a loc-Gowalla scambiando i suoi nodi (x, y) e facendo una $union$ tra il grafo originale e quello con i nodi scambiati.

4.2 Esperimenti computazionali

Qui sotto sono visibili i risultati ottenuti mediante l'applicazione dell'algoritmo su un computer medio-performante. In particolare sono riportati:

- Il tempo impiegato (in formato minuti:secondi)
- Il numero di nodi
- Il numero di archi
- Il numero di triangoli all'interno del grafo.

Grafo	Nodi	Archi	Triangoli	Tempo Impiegato
LocGowalla	196591	950327	2273138	1:42
Youtube	1134890	2987624	3056386	3:03
Google	875713	4322051	13391903	4:13

Da notare che il numero di triangoli (e - quindi - il tempo di esecuzione) è correlato positivamente con il numero di archi, non con quello dei nodi: difatti il grafo web-Google ha meno nodi di com-Youtube, ma il primo ha più triangoli del secondo.

5 Implementazione Database su Neo4J

Per quanto riguarda l'implementazione del Database in Neo4J, adoperando il metodo delle Prepared Statement. Per prima cosa occorre creare i nodi di V. Salviamo in memoria la lista dei nodi del grafo con i relativi gradi, informazioni contenute in *cards1*. Creiamo quindi un ciclo *for* che scorre ogni elemento della lista e lo inserisce in una query Cql con l'intento di creare un nodo. Dalla lista prendiamo l'id denominato *p1* per salvarlo sotto la proprietà "nodo" mentre il grado, denominato *p2*, sotto la proprietà "grado". Una cosa del tutto analoga avviene per la creazione degli archi di E. Prendiamo in memoria le informazioni contenute in *edges*, che contiene Tuple con nodo di partenza e nodo di arrivo per ciascun arco. Dopodichè scriviamo un ciclo *for* che scorre la lista degli archi denominati *p1* e *p2* con all'interno una query Cql che selezioni i nodi la cui proprietà "nodo" corrisponde ai due valori indicizzati e che quindi crei un arco. Il Database contenente nodi e archi è quindi pronto.

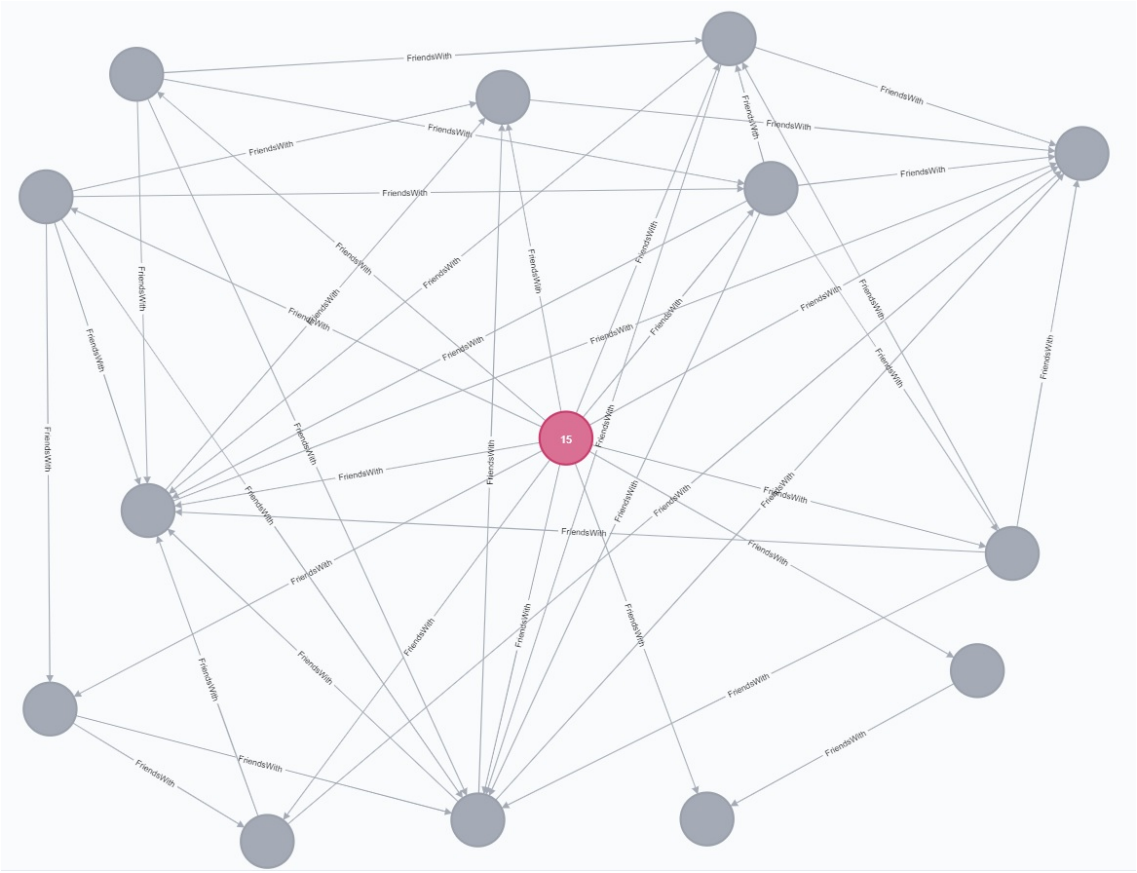

Figure 2: I triangoli del nodo 37938 dati da $G^+(37938)$