TRIANGLE COUNTING GRAPH: MAP-REDUCE ALGORITHM

Carmelina Fascione, Manuela Casole, Mariapia Angelino

Abstract

Il conteggio del numero di cliques, ovvero il numero di cricche, all'interno di grafi di grandi dimensioni rappresenta un problema di rilevante importanza per molte analisi applicative reali (dall'informatica alla medicina). Per tale motivo negli ultimi decenni, è stato oggetto di attenzione per la maggior parte della comunità scientifica che, in particolar modo, si è concentrata sul problema del costo computazionale che si riscontra con grafi molto grandi, dal momento che il conteggio di sottostrutture in un grafo può richiedere un numero elevato di operazioni e pertanto lunghe tempistiche.

*Con l'obiettivo di risolvere questi problemi viene proposta una soluzione in parallelo usando il paradigma MapReduce. Viene considerato l'algoritmo FFF_k proposto da Irene Finocchi, M. Finocchi e E.G. Fusco nell'articolo '**Clique counting in MapReduce: theory and experiments** ' (2015).*

Introduzione

L'algoritmo FFF_k è stato implementato in Java-Spark considerando il caso specifico del conteggio di k-cricche con k=3, quindi, del conteggio del numero di triangoli presenti all'interno di un grafo di grandi dimensioni.

In particolare, l'algoritmo FFF_k consente di ridurre il costo computazionale sfruttando la logica del calcolo distribuito tramite il paradigma MapReduce. Il paradigma di programmazione MapReduce consente l'elaborazione di grandi quantità di dati lavorando in parallelo, quindi sul DB e in memoria, sfruttando il vantaggio di avere tempi di risposta molto brevi, lavorando in memoria, ma allo stesso tempo conservando anche lo storico dei dati.

Il paradigma MapReduce consiste nel suddividere l'operazione di calcolo in diverse parti processate in modo autonomo dai vari nodi che compongono il sistema.

Ogni algoritmo viene eseguito in più round, ogni round consiste nell'esecuzione di un passo di map e uno di reduce, separati da un passo automatico di shuffle. Il passo di map prende in input delle coppie $\langle chiave\ k, valore\ v \rangle$, una alla volta, e restituisce una o più coppie intermedie. Il passo di shuffle si occupa di mandare tutte le coppie intermedie che hanno la stessa chiave a un unico reducer. Dopo che tutte le map avranno esaurito il loro input, i reducer riceveranno le coppie intermedie corrispondenti e potranno cominciare la loro computazione. Al termine della fase di Reduce, il loro output rappresenterà il risultato finale dell'algoritmo o l'input per il prossimo round.

Per poter sfruttare la logica del calcolo distribuito è stato utilizzato il framework open-source Apache Spark. Quando viene utilizzato Spark si vanno a creare delle collezioni di oggetti, chiamate RDD, che sono delle strutture dati partizionate automaticamente da Spark per poi essere distribuite tra i diversi nodi.

Richiami matematici

Per poter comprendere al meglio la logica dell'algoritmo sono necessari alcuni richiami matematici.

Un **grafo G(V, E)** è una struttura dati composta da:

- un insieme finito di nodi (o vertici) V
- un insieme finito di archi E dove ciascun arco E = (u,v) descrive una relazione tra una coppia di nodi u e v.

Dato un grafo ***G***, siano ***u*** i nodi in ***G***, si definisce:

- ***Γ(u)*** l'insieme dei nodi collegati ad ***u*** tramite un arco, ossia l'intorno di ***u***;
- ***d(u)*** = ***|Γ(u)|*** il grado di ***u***, ossia, il numero di nodi presenti nell'intorno ***Γ(u)***;
- una relazione d'ordine totale sui nodi di ***G***: per ogni $x, y \in V(G)$, $x \prec y$ se e solo se ***d(x)*** < ***d(y)*** o se ***d(x)*** = ***d(y)*** e $x < y$ (dove queste ultime rappresentano le etichette associate ai nodi che supponiamo siano univoche);
- ***Γ⁺ (u)*** l'intorno superiore di ***u***, ovvero l'insieme dei nodi $x \in \Gamma^+(u)$ tali che $u \prec x$.
- ***G⁺(u)*** sottografo indotto dall'intorno superiore $\Gamma^+(u)$.

L'algoritmo

L'algoritmo FFF_k, con k=3, sfrutta la relazione di ordine totale $\prec$ per identificare il nodo responsabile del conteggio di un dato triangolo. È necessario definire questa relazione per evitare di ricontare più volte lo stesso triangolo in quanto il grafo di partenza è non direzionato e ciascun arco appare nel dataset due volte.

La strategia prevede di dividere l'intero grafo in vari sottografi, chiamati $G^+(u)$ indotti da $\Gamma^+(u)$, ossia l'intorno superiore del nodo u; e contare indipendentemente il numero di triangoli presenti in ciascun sottografo.

L'algoritmo prevede tre rounds:

- Round 1: "calcolo degli intorni superiori"

  L'obiettivo è quello di calcolare l'intorno superiore di ciascun nodo utilizzando l'informazione fornita dal grado. In particolare:

  - la fase di Map consente di ottenere, per ciascun arco $(x\ y)$, una Tupla $\langle x, y\rangle$ imponendo la condizione di ordine totale, quindi occorre verificare $x \prec y$;
  - nella fase di Reduce si aggregano tutti i nodi aventi la medesima chiave. L'insieme dei nodi in valore definirà l'intorno superiore.

- Round 2: "intersezione intorni inferiori"

  Obiettivo: associare ciascun arco all'insieme dei nodi *u* tale che $G^+(u)$ contiene l'arco.

  - In questo round l'input del Map può essere di due tipi: $\langle u, \Gamma^+(u)\rangle$ oppure $\langle (x, y), \emptyset\rangle$.
    Se l'input è del tipo $\langle u, \Gamma^+(u)\rangle$ in questa fase la Tupla viene rielaborata e il risultato sarà una lista di Tuple che presenta in chiave tutte le possibili combinazioni a 2 a 2 dei nodi contenuti nell'intorno superiore $\langle (x, y), u\rangle$;
    Se invece l'input è del tipo $\langle (x, y), \emptyset\rangle$ per ogni arco (x, y) sarà restituita una Tupla contenente il simbolo $ in valore laddove è rispettata la condizione di ordine totale.
  - Nella fase di Reduce si uniscono i risultati prodotti nel Map e si cercano le coppie di nodi che formano effettivamente un arco, guardando al simbolo $.

- Round 3: "Conteggio degli archi contenuti negli intorni superiori del grafo"

  Per ciascun nodo *u* si conteranno i triangoli, di cui il nodo è responsabile, a partire dal conteggio degli archi presenti in $G^+(u)$. In particolare:

  - La fase di Map prevede che per ogni arco (x, y) sia emessa una Tupla $\langle u, (x, y)\rangle$;
  - Il Reduce riceve come input l'intera lista di archi contenuti nell'intorno superiore $\Gamma^+(u)$, sarà possibile ricostruire $G^+(u)$.

  A partire dal conteggio del numero di archi, presenti nei sottografi indotti dagli intorni superiori, è possibile identificare il numero di triangoli di cui il nodo *u* è responsabile

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 $\langle u; q_{u,k-1} \rangle$

Implementazione dell'algoritmo in Java

L'algoritmo è stato applicato a un grafo non-direzionato reperito dalla Stanford Large Network Dataset Collection (SNAP) chiamato loc-Brightkite, il quale possiede 58228 nodi e 214078 archi; Brightkite era un social network basato sulla posizione condivisa dagli utenti al momento del login, i nodi all'interno del grafo rappresentano gli utenti mentre gli archi la connessione che lega due utenti.

Si tratta di un grafo non direzionato, ossia gli archi presenti sono caratterizzati da una relazione biunivoca, quindi $(u, v) = (v, u)$.

Per l'implementazione è necessaria una fase preliminare di pre-processing del grafo in quanto l'algoritmo richiede una struttura specifica per la Tupla da fornire in input al Map1.

Analizziamo, adesso, le diverse fasi che caratterizzano l'algoritmo:

- **Pre-proccessing del grafo**:
  Innanzitutto, per ogni arco presente nel grafo, viene creata una *JavaPairRDD<Integer, Integer>* utilizzando la funzione MapToPair. La stringa viene splittata rispetto al separatore " ", passando da una struttura del tipo $e1 = (n1, n2)$ ad una Tupla $\langle n1, n2 \rangle$.
  Viene fatta, poi, una trasformazione GroupByKey che consente di raggruppare tutti i valori associati alla medesima chiave. Nel caso specifico si ottiene una Tupla contenete in valore la lista di nodi legata al nodo in chiave attraverso un arco; quindi, ad esempio a partire da $\langle n1, n2 \rangle$, $\langle n1, n3 \rangle$ si ottiene $\langle n1, \{n2, n3\} \rangle$.
  Segue una trasformazione FlatMapToPair che permette di contare il numero di nodi contenuti in valore per calcolare il grado del nodo in chiave, quindi, l'output sarà del tipo:
  $\langle n1, \langle n2, grado(n1) \rangle \rangle$, $\langle n1, \langle n3, grado(n1) \rangle \rangle$.
  Infine, la trasformazione MapToPair consente di invertire l'ordine dei nodi in modo tale da ottenere in valore il nodo ed il suo relativo grado, quindi $\langle n2, \langle n1, grado(n1) \rangle \rangle$, $\langle n3, \langle n1, grado(n1) \rangle \rangle$.
  Per calcolare il grado del nodo che ora appare in chiave si segue una procedura analoga a partire dalla trasformazione GroupByKey; la struttura che si ottiene alla fine di questa fase è la seguente:
  $\langle \langle n2, grado(n2) \rangle, \langle n1, grado(n1) \rangle \rangle$, $\langle \langle n3, grado(n3) \rangle, \langle n1, grado(n1) \rangle \rangle$.
- **Round 1**:
  Per prima cosa gli archi vengono filtrati in modo tale che sia rispettata la condizione di ordinamento totale $\prec$, ossia il nodo in chiave sarà il non preferito. A tal fine, viene eseguita una doppia operazione di filter, selezionando prima gli archi che hanno grado del nodo in entrata strettamente minore del grado del nodo in uscita e poi quelli con uguale grado ma etichetta numerica del nodo in entrata inferiore a quella del nodo in uscita; attraverso l'unione delle due strutture si avrà $\langle \langle n1, grado(n1) \rangle, \langle n2, grado(n2) \rangle \rangle$, n2 rappresenta il nodo preferito (Map1).
  A quest'ultimo output viene, poi, applicata la trasformazione GroupByKey con lo scopo di ottenere l'intorno superiore del nodo in chiave, dato proprio dall'insieme dei nodi adiacenti:

$\langle\langle n1, grado(n1)\rangle, \Gamma^+(n1)\rangle$ (Reduce 1).

- **Round 2**:

  vengono distinti, innanzitutto, due tipi di input:

  - Input 1: a partire dal risultato del Map1 ($\langle\langle n1, grado(n1)\rangle, \langle n2, grado(n2)\rangle\rangle$), viene applicata una trasformazione MapToPair per inserire all'interno della Tupla il simbolo "$", il quale viene usato per indicare che per la Tupla in Input è già stata verificata la condizione di ordinamento totale, $n1 \prec n2;$ il risultato ottenuto è il seguente: $\langle\langle n1, grado(n1)\rangle, \langle n2, grado(n2)\rangle\rangle, \$\rangle$.
  - Input 2: al risultato del Reduce1($\langle\langle n1, grado(n1)\rangle, \Gamma^+(n1)\rangle$) si applica la trasformazione MapToPair per invertire di posizione chiave e valore. I nodi preferiti, presenti in Iterable, vengono poi combinati a due a due attraverso la trasformazione FlatMapToPair e l'utilizzo di un metodo implementato nella nuova classe Java 'coppieNodi'; il risultato sarà una lista di Tuple con la seguente struttura: *JavaPairRDD<Tuple2<Tuple2<Integer, Integer>, Tuple2<Integer, Integer>>, Tuple2<Integer, Integer>>*, dato l'esempio considerato le Tuple saranno del tipo: $\langle\langle x1, grado(x1)\rangle, \langle x2, grado(x2)\rangle\rangle, \langle n1, grado(n1)\rangle$ dove $x1\ , x2 \in \Gamma^+(n1)$, e così per tutte le altre combinazioni di nodi appartenenti all'intorno. (Map2).

  A seconda del tipo di input si procede con la fase di Reduce nel seguente modo:

  - Input 1: si considerano solo le coppie che presentano il simbolo '$' in valore, questo garantisce che l'arco in chiave non verrà considerato più volte per il calcolo finale del numero di triangoli.
  - Input 2: attraverso una trasformazione GroupByKey si raggruppano in valore tutti i nodi collegati al medesimo arco, l'output avrà una struttura del tipo: *JavaPairRDD<Tuple2<Tuple2<Integer, Integer>, Tuple2<Integer, Integer>>, Iterable<Tuple2<Integer, Integer>>>*, dove all'interno dell'iterable ci saranno tutti i nodi collegati all'arco $\langle x1, x2\rangle$ (Reduce2).

  Infine, per mezzo di un join, i due output vengono uniti, e viene eliminato il simbolo "$", tramite la trasformazione MapToPair, avendo completato la sua funzione.

- **Round 3**:

  A partire dall'output del Reduce2, vengono scambiate le posizioni chiave-valore, usando un MapToPair, la struttura ottenuta sarà: *JavaPairRDD<Iterable<Tuple2<Integer*, Integer>>, *Tuple2<Tuple2<Integer*, *Integer>*, *Tuple2<Integer*,*Integer>>>*.

  Dall'Iterable si estraggono individualmente i nodi attraverso la trasformazione FlatMapToPair e la creazione del metodo 'iterableNodi', l'output avrà la seguente struttura *JavaPairRDD<Tuple2<Integer, Integer>, Tuple2<Tuple2<Integer, Integer>, Tuple2<Integer, Integer>>>*, nel caso considerato si ha: $\langle\langle n1, grado(n1)\rangle, \langle\langle x1, grado(x1)\rangle, \langle x2, grado(x2)\rangle\rangle$ dove n1 è il primo nodo dell'Iterable,si procede allo stesso modo per tutti gli altri nodi contenuti nell'Iterable (Map3).

  A questo punto, si raggruppano per chiave tutti gli archi collegati allo stesso nodo usando un GroupByKey .

  Si va ad individuare la taglia dell'Iterable, quindi, il numero di archi presenti in valore, attraverso la funzione MapToPair; in questo modo si identifica il numero di triangoli per cui il nodo in chiave è responsabile.

  Infine, utilizzando un Reduce, si vanno a sommare tutti i valori ottenuti per procedere al conteggio del numero di triangoli presenti nel grafo (Reduce 3).

Applicando il seguente algoritmo al dataset loc-Brightkite il risultato ottenuto indica che il numero di triangoli all'interno del grafo è `494728.`

Implementazione su Neoj

I dati sono stati riportati sul Graph-DB Neo4j per mezzo di query implementate direttamente in java.

Neo4j è un database NoSQL di tipo Graph-based che consente di rappresentare i dati attraverso un modello avente una struttura a grafo, questi possono essere interrogati mediante un linguaggio orientato all'esplorazione dei grafi.

Tra i vantaggi di Neo4j vi è la possibilità di analizzare la struttura e le connessioni di un nodo, senza che le prestazioni siano influenzate dalla taglia del dataset complessivo (performance) e la possibilità di far evolvere la struttura del grafo, aggiungendo nuove proprietà o relazioni senza compromettere l'istanza sottostante (flessibilità).

Per quanto riguarda la fase di progettazione si individuano le istanze del dataset e per ciascuna di esse di introduce un nodo; le relazioni tra istanze sono descritte mediante archi e le due estremità dell'arco coincidono con le due entità coinvolte nella relazione.

In questo caso per poter costruire i nodi si utilizza la query: *"create (m {nodo: "+n1+"})"*, dove n1 è l'elemento che appare in chiave nella Tupla, definita nella fase di pre-processing (freq), avente la seguente struttura JavaPairRDD<Integer, Iterable<Tuple2<Integer, Integer>>> ,

Mentre per la creazione degli archi si considera la Tupla creata all'inizio del lavoro per poter estrarre gli archi dal dataset, la query utilizzata è la seguente: *"match (n),(m) " + "where n.nodo = "+n1+" and m.nodo = "+n2+" " + "create (n)-[a:nearby]->(m) "*.