Implentazione struttura *2-hop cover* tramite *RXL* basato su ordinamento con algoritmo *SamPG*

Fabrizio D'Ascenzo, Ottorino Colantoni

26/04/2020

Introduzione

Una *query sulla distanza* riguarda la distanza tra due vertici di un grafo, cioè la lunghezza di un cammino minimo tra di essi. Senza dubbio, il rispondere a tali query è una delle principali operazioni sui grafi. Per esempio, nei social network, la distanza tra due utenti è considerata per consentire ad un utente di trovare altri utenti o contenuti ad esso più correlati. Nei *web graph*, la distanza tra due pagine web è un indicatore di rilevanza ed è usata per determinare le pagine web più correlate a quella che si sta visitando attualmente. I grafi relativi a tali applicazioni sono anche noti come "*complex graph*".

Per rispondere alle query sulla distanza, ovviamente, si possono usare algoritmi come *BFS - Breadth First Search* o *Dijkstra*. Il problema è che tali algoritmi richiedono più di un secondo per grafi molto ampi come i *complex graph*, cioè sono troppo lenti per essere utilizzati come blocchi fondamentali delle applicazioni sopra citate. Infatti, la risposta a query sulla distanza dovrebbe richiedere un tempo nell'ordine dei microsecondi. Un altro possibile approccio, completamente opposto agli algoritmi appena citati, è quello di calcolare le distanze tra ogni coppia di vertici in anticipo e memorizzarle. Nonostante con questo approccio possiamo rispondere alle query istantaneamente, è anch'esso inutilizzabile poiché sia il tempo di preprocessing che la quantità di memoria richiesta sono quadratiche nel numero dell'input. L'idea, quindi, è quella di cercare una soluzione intermedia tra i due approcci.

Notazione

Sia $G = (V, E)$ un grafo con V come insieme dei vertici ed E come insieme degli archi. Indichiamo con:

- $n = |V|$.
- $m = |E|$.

- $N_G(v) = \{u \in V | (u, v) \in E\}$ l'insieme dei *vicini* di un vertice $v \in V$.
- $d_G(u, v)$ la distanza tra due vertici u e v. Se u e v sono disconnessi $d_G(u, v) = \infty$.
- $P_G(s, t) \subseteq V$ come l'insieme di tutti i vertici in un cammino minimo tra s e t. Quindi:

$$P_G(s, t) = \{v \in V | d_G(s, v) + d_G(v, t) = d_G(s, t)\}.$$

1 Pruned Landmark Labeling

Il metodo proposto in [1] è un metodo *esatto*. Ciò vuol dire che restituisce ogni volta una risposta corretta alla query che gli viene sottoposta. Si ricorda, che ai metodi esatti si contrappongono metodi *approssimati* che, invece, non restituiscono sempre una risposta corretta (ma solitamente si comportano meglio in termini di scalabilità).

Il metodo è basato sulla nozione di *distance labeling* o *distance-aware 2-hop cover*. L'idea del *2-hop cover* è la seguente:

"*Per ogni vertice u, prendiamo un insieme $C(u)$ di vertici in modo che, ogni coppia di vertici (u, v) abbia almeno un vertice $w \in C(u) \cap C(v)$ in un percorso minimo tra di essi. Per ogni vertice u e un vertice $w \in C(u)$, precomputiamo la distanza $d_G(u, w)$. Definiamo, inoltre, l'insieme $L(u) = \{(w, d_G(u, w))\}_{w \in C(u)}$ come la* ***label*** *di u. Usando le label, è facile capire che si può rispondere ad una query sulla distanza tra due vertici u e v $QUERY(u, v, L)$ come:*

$$d_G(u, v) = \ min\{\delta + \delta' | (w, \delta) \in L(u), (w, \delta') \in L(v)\}$$

in un tempo pari a $\mathcal{O}(|L(u)| + |L(v)|)$. L'insieme delle label $L = \{L(u)\}_{u \in V}$ è detto ***2-hop cover***".

Ciò che viene proposto in [1] è un approccio denominato *Pruned Landmark Labeling* - PLL.

In poche parole, sia $V = v_1, v_2, ..., v_n$, partendo da un index vuoto L_0 e supponendo di eseguire una BFS a partire da ogni vertice v_i nell'ordine $v_1, v_2, ..., v_n$, si calcola l'index L_k a partire dall'index L_{k-1} nel modo seguente:

- se stiamo visitando un vertice u con distanza δ da v_k e

  $$QUERY(v_k, u, L_{k-1}) \leq \delta$$

  allora "riduciamo" u, cioè la coppia (v_k, δ) non viene aggiunta a $L_k(u)$, quindi $L_k(u) = L_{k-1}(u)$, e non attraversiamo nessun arco uscente da u.
- se invece $QUERY(v_k, u, L_{k-1}) > \delta$ allora $L_k(u) = L_{k-1}(u) \cup \{(v_k, \delta)\}$

- per ogni vertice u che non viene visitato alla k-th iterazione della BFS $L_k(u) = L_{k-1}(u)$.

In *Fig. 1* è riportato lo pseudocodice dell'algoritmo.

Algorithm 1 Pruned BFS from $v_k \in V$ to create index L'_k.

```
1: procedure PrunedBFS(G, v_k, L'_{k-1})
2:     Q ← a queue with only one element v_k.
3:     P[v_k] ← 0 and P[v] ← ∞ for all v ∈ V(G) \ {v_k}.
4:     L'_k[v] ← L'_{k-1}[v] for all v ∈ V(G).
5:     while Q is not empty do
6:         Dequeue u from Q.
7:         if Query(v_k, u, L'_{k-1}) ≤ P[u] then
8:             continue
9:         L'_k[u] ← L'_{k-1}[u] ∪ {(v_k, P[v_k])}
10:        for all w ∈ N_G(v) s.t. P[w] = ∞ do
11:            P[w] ← P[u] + 1.
12:            Enqueue w onto Q.
13:    return L'_k
```

Algorithm 2 Compute a 2-hop cover index by pruned BFS.

```
1: procedure Preprocess(G)
2:     L'_0[v] ← ∅ for all v ∈ V(G).
3:     for k = 1, 2, ..., n do
4:         L'_k ← PrunedBFS(G, v_k, L'_{k-1})
5:     return L'_n
```

Figura 1: Algoritmo Pruned BFS

Finora abbiamo supposto di visitare i vertici nell'ordine $v_1, v_2, ..., v_n$ ma, ovviamente, possiamo sceglierne un altro qualsiasi. Tuttavia, è importante notare che, come riportato in [1], l'ordine con il quale vengono considerati i vertici nella Pruned BFS è cruciale per le performance del metodo. In generale, un buon ordine dei nodi è quello nel quale all'inizio vengono considerati vertici che incidono su molti cammini minimi. Infatti, in questo modo, i primi vertici che verranno esaminati copriranno gran parte dei cammini minimi e l'algoritmo eviterà di visitare molti dei vertici successivi.

Il problema, adesso, è quindi quello di determinare un "buon ordine" dei vertici.

1.1 Strategie per il *Vertex Ordering*

In [1] sono proposte diverse strategie per determinare l'ordine dei vertici con il quale eseguire la *Pruned BFS*:

- **Random**: i vertici sono ordinati in modo randomico.
- **Grado**: i vertici sono ordinati in modo decrescente in base al loro grado. L'idea alla base è che, i vertici con grado maggiore, hanno un numero maggiore di connessioni e quindi incideranno su molti più cammini minimi rispetto a vertici con grado minore. Tuttavia, come affermato in [2], questo approccio si comporta bene per grafi non pesati e non orientati con un diametro piccolo, ma non è robusto.
- **Centralità (*closeness centrality*)**: i vertici sono ordinati a partire da quelli più vicini al centro della rete fino a quelli più lontani. L'idea alla base è simile a quella dell'ordinamento per grado, se un vertici è più centrale, esso incide su più cammini minimi. Calcolare esattamente la centralità è computazionalmente costoso. Viene quindi suggerito di approssimarla, calcolando la distanza rispetto ad un sottoinsieme di vertici selezionati randomicamente.

2 SamPG

Nella sezione precedente sono stati presentati alcuni algoritmi per risolvere il problema del "Vertex Ordering", tuttavia tali algoritmi funzionano bene solamente su alcune tipologie di grafi e dunque risultano soluzioni poco robuste.

In [3] viene, quindi, presentato un nuovo algoritmo di ordinamento in grado di trovare delle buone etichette per varie classi di grafi. In particolare, la versione base di tale algoritmo indica che l'i-esimo vertice più importante è colui che copre il maggior numero di cammini minimi non coperti dai vertici precedentemente estratti. Il primo passo per poter implementare tale regola è quello di calcolare per ogni nodo un albero dei cammini minimi attraverso l'algoritmo di Dijkstra. Indichiamo con T_s l'albero dei cammini minimi radicato nel nodo sorgente s. Poiché soltanto i nodi compresi nel percorso s-t in T_s possono coprire la coppia [s,t], allora il numero di discendenti di un nodo v in T_s rappresenta il numero di cammini minimi coperti da v. Indicheremo quindi la somma dei cammini minimi coperti da v in ogni albero dei cammini minimi con $\sigma(v)$. Ad ogni iterazione l'algoritmo seleziona il vertice v che possiede il massimo valore di $\sigma(v)$. Prima di procedere ad una nuova iterazione l'algoritmo rimuove tutti i sottoalberi radicati nel nodo v estratto ed aggiorna i valori di $\sigma(\cdot)$ per ogni nodo discendente o antenato di v.

È facile notare che tale algoritmo è computazionalmente costoso sia in termini di spazio che di tempo ed inoltre nel caso in cui i cammini minimi non sono unici, l'algoritmo sottostima il numero di coppie coperte da un nodo estratto e ciò può portare a generare etichette più grandi del necessario. Per migliorare tali risultati possiamo usare una forma ibrida di tale algoritmo che prevede prima

l'uso del *basic algorithm* ed in seguito l'uso della tecnica **PL**(*Pruned Labeling*) per calcolare l'etichettatura. Tale algoritmo, che prende il nome di *HybPG*, presenta comunque un elevato costo computazionale. Appunto per questo motivo nell'articolo [2] vengono presentati alcuni miglioramenti ad esso applicabili. Questi miglioramenti prevedono di creare una stima di $\sigma(v)$, indicata con $\tilde{\sigma}(v)$, sufficientemente precisa da poter individuare quali sono i nodi più importanti nella rete. Un approccio possibile per realizzare tale stima è quello di realizzare $k << n$ alberi dei cammini minimi partendo da nodi sorgenti estratti randomicamente dal grafo. A tal punto impostiamo $\tilde{\sigma}(v)$ come il numero dei cammini minimi coperti da v nei k alberi realizzati. Ogni volta che un nodo viene estratto eseguiamo le stesse procedure indicate nel *basic algorithm* e cioè rimuoviamo i sottoalberi radicati in v ed aggiorniamo i valori di $\tilde{\sigma}(\cdot)$ per i discendenti e antenati di v.

Un primo problema legato a tale strategia riguarda i vertici meno significativi. Durante l'esecuzione dell'algoritmo i vari alberi si riducono e dunque non è possibile estrarre da essi sufficienti informazioni per poter gestire i nodi meno significativi. Per risolvere questo problema è possibile creare nuovi alberi da nuove radici durante l'esecuzione dell'algoritmo. Tali alberi possono essere realizzati sfruttando l'algoritmo di Dijkstra ma in aggiunta possiamo escludere da essi tutti i vertici già coperti dai precedenti nodi estratti. Visto che per potare i vertici già coperti abbiamo bisogno di etichette parziali, ogni volta che estraiamo un nodo v eseguiamo un iterazione di PL ed aggiungiamo v a tutte le etichette rilevanti. Gli alberi così generati conterranno solamente i cammini minimi non ancora coperti e saranno sempre più piccoli con il proseguire dell'algoritmo. Un criterio d'arresto alla generazione di nuovi alberi può essere stabilito in base al numero dei nodi contenuti nei vari alberi, ad esempio ci si può fermare quando la somma dei nodi è maggiore di $10kn$.

Un ulteriore problema è legato alla varianza di $\tilde{\sigma}(v)$. Infatti campionando i k alberi da realizzare c'è la possibilità di sovrastimare l'importanza di un nodo radice o comunque di uno vicino ad essa. Una soluzione robusta a tale problema è quella di mantenere c differenti $\tilde{\sigma_1}(v)$,$\tilde{\sigma_2}(v)$,...,$\tilde{\sigma_c}(v)$, per una qualsiasi costante c (l'importante è che il numero di alberi campionati $k >= c$), al posto di uno soltanto. Il contatore $\tilde{\sigma_i}(v)$ contiene tutti i discendenti presenti nell'albero T_j per cui $i = (j \ mod \ c)$. Attraverso tali contatori possiamo eliminare i due che contengono i valori più alti, che potrebbero rappresentare dei valori anomali, e calcolare la media dei restanti. Il vertice da selezionare è quello che presenta la media più elevata. Nel caso di pareggio tra due vertici selezioniamo colui che massimizza $\tilde{\sigma}(v) = \sum_{i=1}^{c} \tilde{\sigma_i}(v)$.

Nel complesso, tale algoritmo di ordinamento prende il nome di *SamPG* e rientra nell'insieme delle strategie note come *Robust Exact Labeling* (RXL).

3 Implementazione

In questa sezione riportiamo la nostra implementazione della struttura dati *2-hop cover* mediante tecnica *RXL*, basata sull'utilizzo dell'algoritmo *SamPG* per

il calcolo dell'ordinamento e delle label. In particolare, inizialmente analizzeremo alcune parti salienti del codice, realizzando di fatto un collegamento con ciò che è stato introdotto nelle sezioni precedenti, e in una seconda parte, invece, ci soffermeremo sull'analisi degli iperparametri (e.g. numero di campioni iniziali, numero di contatori ecc.) e sugli effetti che essi hanno sulla qualità delle label e sui tempi richiesti dall'algoritmo.

3.1 Parti salienti del codice

In *Code* 1 è riportato il frammento di codice che, ricevuti in input il grafo in oggetto e tutti gli altri iperparametri, di cui parleremo successivamente, produce in output la struttura dati *2-hop cover*.

Più precisamente, dopo aver inizializzato gli oggetti necessari alla lettura del grafo, a riga 6, vengono istanziati i primi k alberi. A partire da tali campioni, nel ciclo *for* che si svolge da riga 17 a riga 28, vengono eseguite nell'ordine le seguenti operazioni:

1. si estrae il nodo v ritenuto di maggiore importanza, secondo le informazioni contenute negli alberi campioni presenti attualmente nella "foresta". Note le problematiche legate all'estrazione del massimo (esposte nella sezione 2), è data la possibilità di utilizzare un numero contatori c stabilito in input. Ricordiamo che, dato un nodo v, ogni contatore $\sigma_i(v)$ contiene il numero di discendenti del nodo v negli alberi $j = i \ mod \ c$.

2. Vengono create le label per il nodo appena estratto mediante *Pruned Labeling*.

3. A questo punto, vengono rimossi in tutti i campioni i sottoalberi radicati nel nodo v, estratto come massimo, e vengono quindi aggiornati i relativi contatori dei discendenti. Dopo varie iterazioni, può verificarsi la condizione in cui tutti gli alberi del set sono "vuoti", ovvero contengono la sola radice (i.e. tutti i contatori $\sigma_i(v) = 0$). In questa condizione, non è possibile individuare un nodo con più discendenti rispetto agli altri e di conseguenza nessun sottoalbero viene eliminato. Appunto per questo motivo, l'operazione di aggiornamento viene eseguita soltanto se tale condizione non sussiste.

4. L'ultima operazione eseguita nel ciclo è quella di incrementare il numero di alberi campionati. In riga 24 vengono quindi aggiunti, ad ogni iterazione, un numero di alberi stabilito in input. Questa operazione viene eseguita fintantoché la cardinalità della foresta non raggiunge un valore soglia, anch'esso stabilito in input.

Si noti che le operazioni 1 e 2 vengono eseguite per un numero di iterazioni pari al numero dei nodi, questo perché è condizione sufficiente, per la correttezza delle label, che ogni nodo sia estratto almeno una volta.

Code 1: "RXL"

```
void runRXL(std::string graph_location,int num_samples,int
    num_counters,int num_newsamples,int max_numtrees,std::string
    output_location){

     NetworKit::Graph *graph;
     Auxiliary::read(graph_location, false, &graph);
     SamPG *spg = new SamPG(num_samples, num_counters, graph);
     spg->createForest();
     int max;

     std::pair <std::vector<custom_node>, std::vector<custom_node>>
         keeper;
     keeper.second.resize(graph->upperNodeIdBound());
     Labeling *labeling = new Labeling(graph->isDirected());
     Labeling_Tools *lt = new Labeling_Tools(graph, labeling, keeper);

     ProgressStream builder_(graph->numberOfNodes());
     builder_.label() << "Building WEIGHTED UNDIRECTED labeling for "
         <<graph->numberOfNodes()<< " vertices";

     for (int i = 0; i < graph->numberOfNodes(); i++) {
         max = spg->maxDescNode();
         lt->add_node_to_keeper(max, i);
         lt->weighted_build_RXL();
         if(!spg->isEnded()) {
             spg->updateForest(max);
         }
           if (spg->getNumSamples()< max_numtrees) {
               spg->encreaseForest(num_newsamples, labeling);
           }
         ++builder_;
     }

     InputOutput* io = new InputOutput();
     io->printLabelsOnFile(labeling, output_location);

}
```

In *Code* 2 è invece riportato il codice che implementa una versione "modificata" dell'*algoritmo di Dijkstra*. La particolarità di tale implementazione riguarda il fatto che, passando il booleano "*pruned*" uguale a *true*, nella fase di rilassamento degli archi si calcola la distanza dal nodo attuale al nodo vicino che si sta considerando e:

- se la distanza attuale è minore o uguale a quella risultante dalle label già calcolate, si aggiunge l'arco all'albero dei camini minimi;

- se la distanza attuale è maggiore di quella risultante dalle label già calcolate, viceversa.

In particolare, questa versione *Pruned* dell'algoritmo di Dijkstra viene utilizzata nella funzione *EncreaseForest()* per la creazione di nuovi alberi per il set di campioni e attraverso il pruning si evita di considerare cammini già coperti.

Code 2: "Dijkstra"

```
void Dijkstra::runDijkstra(Tree* treeDijkstra, NetworKit::Graph* graph,
    bool pruned , Labeling* index ) {

    boost::heap::fibonacci_heap<heap_dijkstra>* pq = new
        boost::heap::fibonacci_heap<heap_dijkstra>();
    boost::heap::fibonacci_heap<heap_dijkstra>::handle_type* handles =
        new
        boost::heap::fibonacci_heap<heap_dijkstra>::handle_type[graph->upperNodeIdBound()];
    std::vector<int> distances;
    int size= graph->numberOfNodes();
    distances.resize(size,INF);
    int source = treeDijkstra->getRoot()->ID;
    distances[source] = 0;
    handles[source] = pq->push(heap_dijkstra(treeDijkstra->getRoot(),0));
    int wuv;
    while(pq->empty() == false){
        treeNode* current = pq->top().node;
        int distance = pq->top().prio;
        pq->pop();
        graph->forNeighborsOf(current->ID, [&](int v) {
            wuv = graph -> weight(current->ID,v);
            if(distances[v] == INF) {
                if(!pruned || (pruned &&
                    index->query(source,v)>(distance+wuv))) {
                    distances[v] = distance + wuv;
                    treeNode *new_child = new treeNode();
                    new_child->ID = v;
                    handles[v] = pq->push(heap_dijkstra(new_child,
                        distances[v]));
                    treeDijkstra->addNode(new_child, current);
                }
            }
            else if(wuv+distance< distances[v]) {
                distances[v] = distance + wuv;
                (*handles[v]).prio = distances[v];
                pq->decrease(handles[v]);
                treeDijkstra->updateFather(current,(*handles[v]).node);
            }
        });
    }
}
```

3.2 Analisi degli iperparametri e test dell'implementazione

Come esposto precedentemente il funzionamento dell'algoritmo si basa su alcuni iperparametri passati in input dall'utente. Nel dettaglio, questi iperparametri sono:

- ***Numero iniziale dei campioni***: determina il numero di alberi che vengono inizialmente creati e dal quale si estrae il primo nodo con il massimo numero di discendenti. Abbiamo notato che è necessario, all'inizio, avere un set di campioni sufficientemente grande. Infatti, in questo modo, è garantito che il primo nodo estratto sia un nodo incidente su un elevato numero di cammini minimi che, quindi, non verranno inclusi nella costruzione dei nuovi alberi.

- ***Numero di contatori***: la scelta del numero di contatori va effettuata tenendo conto che, maggiore è il numero dei contatori e migliore sarà la stima del nodo "più importante" (i.e. quello con più discendenti) ad ogni iterazione ma, minore è il numero dei contatori e minore è il tempo di convergenza del metodo. Detto ciò, occorre trovare un compromesso tra costo e accuratezza. In [3] viene consigliato un numero di contatori pari a 16.

- ***Numero di nuovi alberi generati ad ogni iterazione***: in tal caso il compromesso va trovato tra il costo del metodo e il numero delle label. Infatti, soprattutto nelle prime iterazioni, è fondamentale aggiungere un numero consistente di alberi, al fine di migliorare la stima dei successivi nodi estratti e quindi diminuire il numero delle label create. Tuttavia, è ovvio che dovendo costruire ad ogni iterazione un numero consistente di alberi, l'algoritmo impiegherà un tempo maggiore.

- ***Numero massimo di alberi***: con il quale si può specificare la cardinalità massima del set di alberi campioni. Raggiunta infatti la dimensione specificata in input, l'algoritmo non genererà più nuovi alberi. Questo parametro è necessario poiché da un certo in poi, la generazione di nuovi alberi non apporterebbe alcun beneficio ma, al contrario, sovraccaricherebbe il metodo.

In generale, non esiste una sola combinazione di valori, per questi iperparametri, che garantisce buoni risultati per tutti i grafi, ma ovviamente vanno scelti volta per volta in base alle caratteristiche del grafo stesso.

A seguire riportiamo una serie di esperimenti da noi svolti, sia per testare il corretto funzionamento dell'algoritmo, sia per verificare la corrispondenza tra gli effetti della variazione degli iperparametri, prevista in linea teorica, con i risultati effettivamente ottenuti. In particolare, gli esperimenti sono stati svolti con il seguente setup:

- **Grafo**: *caida-big.dat.hist*, pesato, non diretto, $|V| = 32000$ e $|E| = 40204$;

- **Hardware**:
  - CPU: Intel® Core™ i7-4710HQ CPU @ 2.50GHz 8 core;
  - GPU: Nvidia® GeForce® GTX 850M;
  - RAM: 16 GB di SDRAM DDR3-1600 (2 x 8 GB).

Per lo svolgimento degli esperimenti abbiamo variato un iperparametro alla volta, testando una serie di valori crescenti, e mantenendo fissi tutti gli altri. Di ogni esperimento abbiamo conservato il miglior valore ottenuto, che è stato poi fissato negli esperimenti successivi. Nello specifico, l'ordine di test degli iperparametri è stato: *numero di campioni iniziali*, *numero di campioni aggiunti ad ogni iterazione* e *massimo numero di campioni* (**N.B.** il numero dei contatori non è stato testato, poiché un valore era espressamente suggerito da [3]). Per ognuno degli esperimenti sono riportati i grafici relativi a:

- tempo di creazione della foresta di campioni;
- tempo di creazione dei nuovi campioni;
- tempo totale richiesto dall'algoritmo;
- numero di label generate.

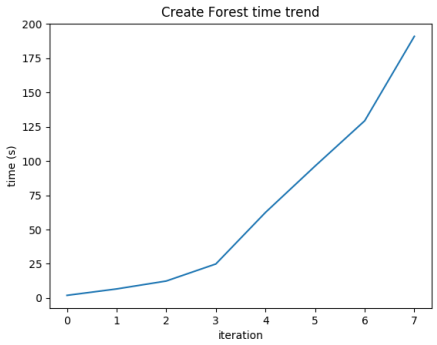
Per ogni esperimento sono inoltre riportati nelle tabelle i valori utilizzati per gli iperparametri in ciascuna iterazione.

Riassumendo, ciò che è possibile osservare nei grafici è che:

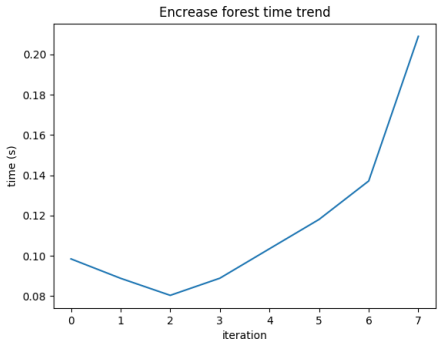
- per quanto riguarda il numero iniziale dei campioni, come abbiamo già detto precedentemente, avere un numero sufficientemente consistente di alberi all'inizio del metodo permette di ottenere un minor numero di label ma, come si può notare nelle figure 2a, 2c e 2d, se troppo consistente incrementa notevolmente il tempo richiesto e non apporta alcun miglioramento sul numero delle label.
- Anche nel caso del secondo iperparametro, come risulta dai grafici nelle figure 3b, 3c e 3d, incrementando il numero degli alberi aggiunti ad ogni iterazione si ottiene una diminuzione del numero delle label generate al costo di un maggior tempo di esecuzione. Inoltre, si noti che da un certo valore in poi (in tal caso 50 alberi) il miglioramento ottenuto nel numero delle label è relativamente ridotto.
- Infine, come è prevedibile, considerando un maggior numero di alberi totali il numero delle label tende a diminuire. Viceversa, per quanto riguarda il tempo di esecuzione.

Al termine di tali esperimenti, i valori che abbiamo ritenuto "ottimali' per il grafo considerato sono:

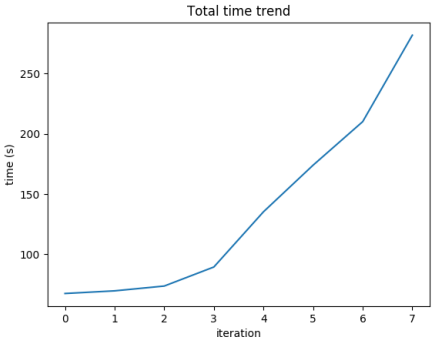
- Numero di alberi iniziali: 200;
- Numero di alberi generati ad ogni iterazione: 20;
- Numero massimo di alberi considerati: 5000.



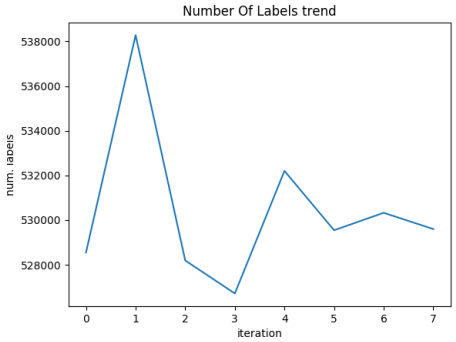
(a) Tempo per la creazione della foresta



(b) Tempo creazione nuovi campioni



(c) Tempo totale

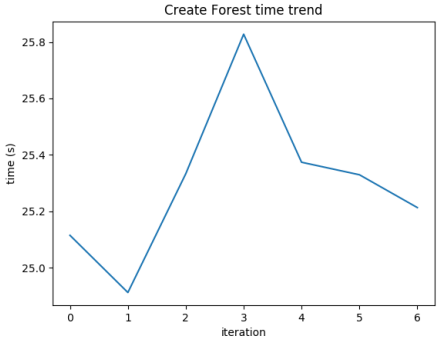


(d) Numero di label

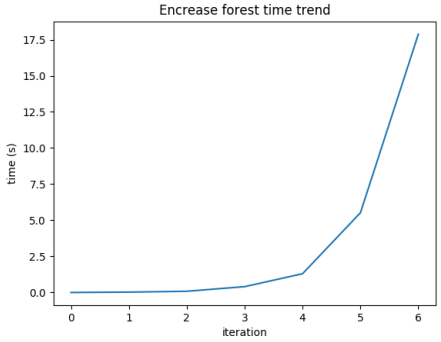
Figura 2: Esperimento su numero di campioni iniziali

Iter./Iperparam.	**Campioni Iniziali**	**Numero di contatori**	**Nuovi Campioni**	**Max Numero Campioni**
0	16	16	20	2000
1	50	16	20	2000
2	100	16	20	2000
3	200	16	20	2000
4	500	16	20	2000
5	750	16	20	2000
6	1000	16	20	2000
7	1500	16	20	2000

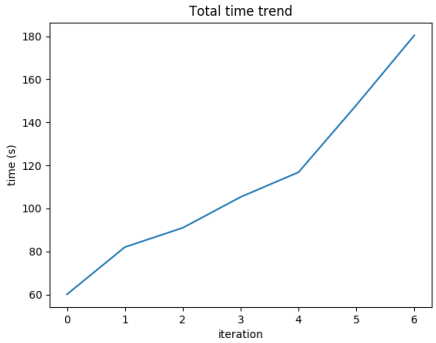
Tabella 1: Variazione Numero di campioni iniziali



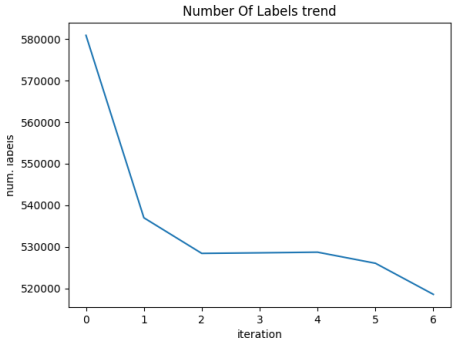
(a) Tempo per la creazione della foresta



(b) Tempo creazione nuovi campioni



(c) Tempo totale

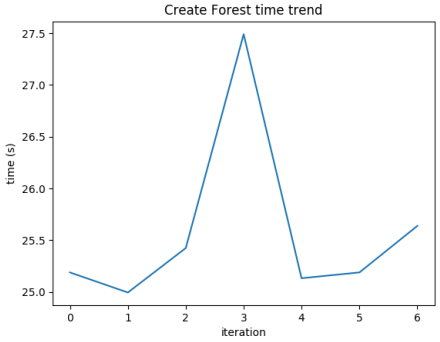


(d) Numero di label

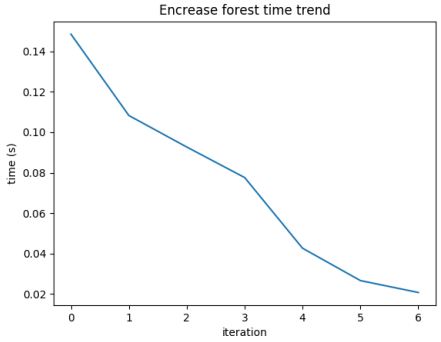
Figura 3: Esperimento su numero di campioni aggiunti ad ogni iterazione

Iter. /Iperparam.	**Campioni Iniziali**	**Numero di contatori**	**Nuovi Campioni**	**Max Numero Campioni**
0	200	16	1	2000
1	200	16	10	2000
2	200	16	20	2000
3	200	16	50	2000
4	200	16	100	2000
5	200	16	250	2000
6	200	16	500	2000

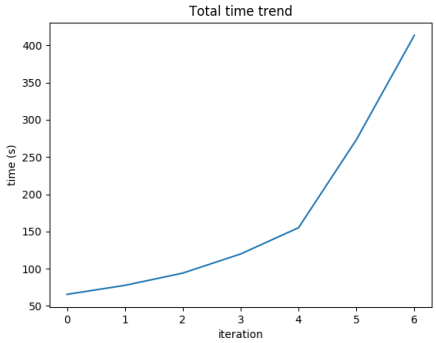
Tabella 2: Variazione numero di campioni inseriti



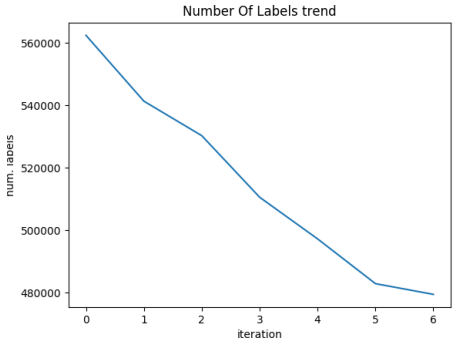
(a) Tempo per la creazione della foresta



(b) Tempo creazione nuovi campioni



(c) Tempo totale



(d) Numero di label

Figura 4: Esperimento su massimo numero di campioni

Iter. /Iperparam.	**Campioni Iniziali**	**Numero di contatori**	**Nuovi Campioni**	**Max Numero Campioni**
0	200	16	20	1000
1	200	16	20	1500
2	200	16	20	2000
3	200	16	20	3000
4	200	16	20	5000
5	200	16	20	10000
6	200	16	20	15000

Tabella 3: Variazione Max Numero Campioni

Utilizzando i valori trovati con gli esperimenti precedenti, ne abbiamo condotto un altro comparando i risultati ottenuti con la nostra implementazione di *RXL* con i risultati ottenuti utilizzando *PLL* basato su un ordinamento per grado. Nello specifico, abbiamo sottoposto alle due implementazioni delle query generate in modo randomico e senza ripetizioni. Abbiamo ritenuto che un numero accettabile di query fosse pari al 10% del numero dei nodi del grafo (nel nostro caso quindi 3200 query).

I risultati ottenuti sono riportati nel grafico in figura 5 e nella tabella sottostante e, possiamo dire, che rispecchiano ciò che effettivamente ci aspettavamo. Infatti, dal grafico in figura 5, si nota fin da subito la differenza nei tempi di risposta tra le due implementazioni ($\approx 1\ \mu s$ in media) e, dalla tabella, risulta il minor numero di label ma al costo di un maggior tempo di preprocessing.

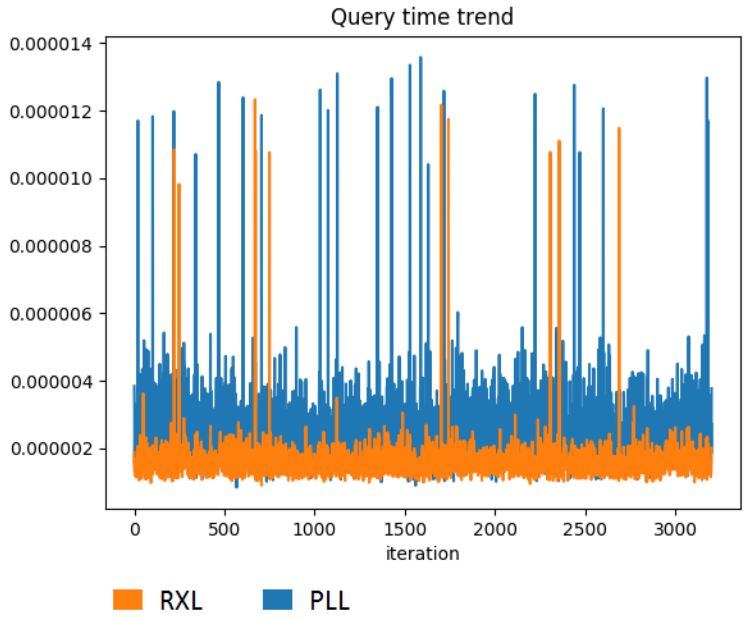


Figura 5: Tempo di risposta alle query: RXL vs PLL

Metodo / Risultato	**Tempo di preprocessing**	**Numero di label**	**Tempo medio di query**
PLL	11.5273 s	1407800	2.59475×10^{-6} s
RXL	159.532 s	496264	1.59176×10^{-6} s

Tabella 4: Risultati RXL vs PLL su grafo “Caida-big-dat”

Al fine di validare i risultati ottenuti e testare l'algoritmo su grafi con caratteristiche differenti abbiamo condotto un ulteriore esperimento. In particolare, abbiamo scelto di eseguire il nostro algoritmo su grafi con un elevato rapporto tra numero di archi e numero di nodi, dato che proprio su tali grafi ci aspettavamo di ottenere ottimi risultati, infatti in questa tipologia di grafi è molto probabile che uno stesso nodo incida su un elevato numero di cammini minimi. A tal proposito abbiamo utilizzato dei grafi relativi alle relazioni tra utenti social presi dal sito http://networkrepository.com. Avendo trovato esclusivamente grafi non pesati abbiamo inserito un valore randomico come costo dell'arco in modo tale da simulare una distanza tra due nodi connessi (il costo potrebbe rappresentare ad esempio un valore calcolato sulla base del numero di amici in comune, il numero di "like" ecc.).

In tabella 5 sono riportati i valori degli iperparametri utilizzati per ciascun grafo (**N.B.** tali valori sono stati dedotti come visto per il grafo "*Caida-big-dat*"), in tabella *6* e nei grafici in figura 6 possono essere osservati i risultati degli esperimenti.

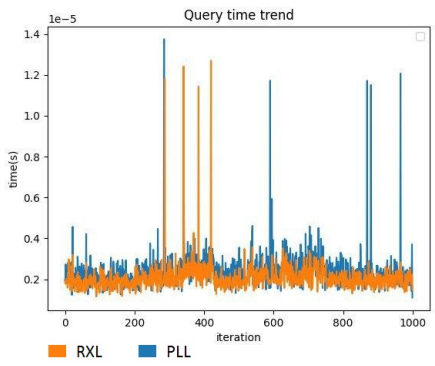
Come è possibile vedere, anche in tal caso *RXL* produce un numero di label e richiede un tempo medio di query inferiore rispetto a *PLL* ma, a costo di un maggiore tempo di preprocessing. Da notare inoltre che la differenza tra i due metodi tende a diminuire quando il grafo considerato è relativamente piccolo, rendendo di fatto ingiustificato il maggiore tempo preprocessamento richiesto da *RXL*.

Grafo	**Campioni Iniziali**	**Numero di contatori**	**Nuovi Campioni**	**Max Numero Campioni**
ff-10000	60	16	10	1500
Simmons81	80	16	10	700
Middlebury45	20	16	5	500
Howard90	20	16	5	150

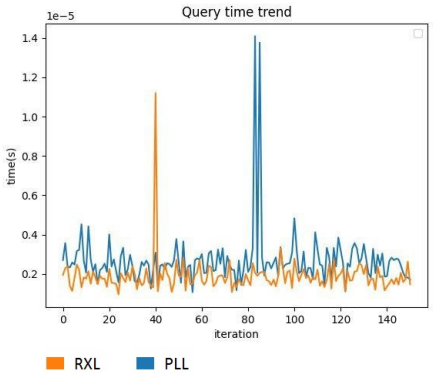
Tabella 5: Valori iperparametri utilizzati sui diversi grafi

Grafo		**T. preproc. (s)**		**Num. Label**		**AVG query (μs)**	
Nome	($\|V\|, \|E\|$)	**PLL**	**RXL**	**PLL**	**RXL**	**PLL**	**RXL**
ff-10000	(10000, 44392)	3.43	31.73	306204	255053	2.39	2.08
Simmons81	(1518, 32988)	2.06	4.17	72833	49588	2.70	1.90
Middlebury45	(3075, 124610)	14.99	54.57	331877	208384	5.13	3.55
Howard90	(4047, 204850)	21.73	93.25	345994	232958	5.81	4.39

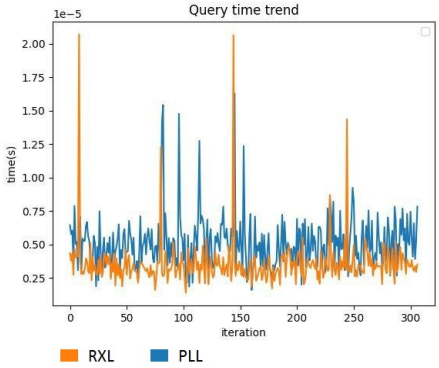
Tabella 6: Risultati RXL vs PLL



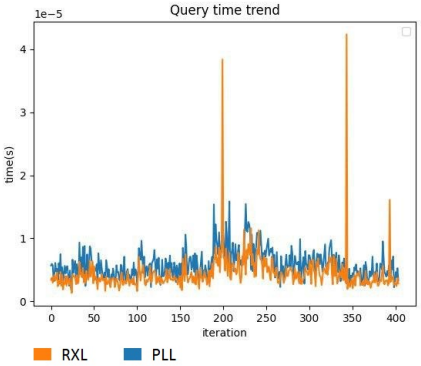
(a) ff-10000



(b) Simmons81



(c) Middlebury45



(d) Howard90

Figura 6: Tempo di risposta alle query su più grafi: RXL vs PLL

Conclusioni e sviluppi futuri

Riassumendo, il nostro lavoro è partito da una analisi teorica della struttura dati *2-hop cover* ed in particolare di una implementazione basata sulla tecnica *RXL*. Il passo successivo è stata l'implementazione di tale tecnica, limitandoci ai soli grafi indiretti, pesati. Una volta verificato il corretto funzionamento dell'implementazione, abbiamo effettuato una serie di test per valutare pro e contro di questa tecnica, anche mettendola a confronto con la più classica *PLL*. I risultati ottenuti sono stati in linea con quanto appreso in [3], cioè, in generale, una migliore qualità delle label generate e un minor tempo medio di risposta alle query, a fronte di un maggior tempo di preprocessamento.

In virtù dei risultati ottenuti, crediamo che possibili sviluppi futuri siano:

- estendere la nostra implementazione anche a grafi diretti;
- diminuire il tempo richiesto dalla fase di preprocessing parallelizzando su più core e/o dispositivi il metodo;
- migliorare la selezione dei nodi nel momento in cui non si è più in grado di estrarre un nodo con il massimo numero di discendenti dal set dei campioni (i.e quando gli alberi nel set contengono la sola radice). Infatti, nella nostra implementazione, per semplicità, raggiunto tale limite i nodi rimanenti vengono estratti in sequenza. Un possibile miglioramento potrebbe essere quello di realizzare un mix con l'ordinamento per grado.
- considerare l'ipotesi di realizzare una tecnica di aggiornamento "pesato" degli alberi nella foresta in seguito all'estrazione di un nodo.

Riferimenti bibliografici

[1] Yoichi Iwate Takuya Akiba and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. 2013.

[2] Thomas Pajor Daniel Delling, Andrew V. Goldberg and Renato F. Werneck. Robust distance queries on massive networks. *Microsoft Research*, 2014.

[3] Delling D. Goldberg A.V. Werneck R.F. Abraham, I. Hierarchical hub labelings for shortest paths. *Microsoft Research*, 2012.