

Algorithm Engineering

Roberto Di Stefano

INDICE

I	Introduzione	2
I-A	Argomento di studio e stato dell'arte	2
I-B	Nozioni preliminari	3
I-C	Algoritmo GLR e risultati noti	4
I-C.1	Pseudocodice	4
I-C.2	Complessità temporale	4
I-C.3	Prestazioni dell'algoritmo	4
I-D	Obbiettivo dello studio	5
II	Valutazione degli esperimenti	6
II-A	Ambiente di Test	6
II-A.1	Configurazione dell'ambiente	6
II-A.2	Implementazioni degli algoritmi	6
II-A.3	Generazione dei grafi	8
II-A.4	Generazione delle partizioni	9
II-A.5	Configurazione degli esperimenti	10
II-B	Impostazione degli esperimenti	11
II-C	Analisi della correttezza	15
II-C.1	Esecuzione su istanze di input semplici	15
II-C.2	Confronto tra le implementazioni	15
II-D	Analisi del tempo di esecuzione	17
II-D.1	Analisi dell'algoritmo non ottimizzato	17
II-D.2	Analisi dell'algoritmo ottimizzato	21
III	Conclusioni	27

I. INTRODUZIONE

A. Argomento di studio e stato dell'arte

In un generico grafo i nodi con maggiore capacità di diffusione vengono chiamati influential node. Identificare gli influential node e ordinare i nodi di un grafo in base alla loro capacità di diffusione sta diventando sempre più utile in molteplici ambiti:

- La diffusione di epidemie può essere modellata opportunamente attraverso un grafo, nel quale i singoli individui o gruppi di individui vengono rappresentati tramite nodi, le interazioni tra individui vengono rappresentati tramite archi e gli influential node in questo modo andranno a rappresentare gli individui più critici per la diffusione del contagio;
- Un social network può essere rappresentato tramite un grafo dove ciascun nodo rappresenta un profilo, gli archi rappresentano le relazioni di amicizia, e gli influential node in questo modo rappresenteranno i profili con il maggior numero di connessioni o relazioni, che quindi eserciteranno un'influenza significativa sulla rete sociale e sono di conseguenza di interesse per la diffusione di informazioni quali pubblicità o propaganda politica.

Negli anni i ricercatori hanno ideato diverse metriche di centralità che permettessero di misurare il grado di influenza di ciascun nodo di un grafo e di conseguenza di individuare gli influential node. Storicamente uno delle prime misure di centralità e anche una delle più semplici, è stata la degree centrality [3] che consiste nel classificare i nodi sulla base del numero di vicini di ciascuno, quanti più vicini ha un nodo quanto più sarà influente. Questo tipo di metrica, per quanto richieda poco tempo per essere calcolata, non tiene conto della struttura globale della rete e di conseguenza ne risente in termini prestazionali. Infatti un nodo che ha pochi vicini molto influenti nella pratica è più influente di un nodo che ha tanti vicini periferici ma, utilizzando la degree centrality, otterrebbe un valore di centralità minore rispetto al secondo. Per risolvere questo problema è stata introdotta la closeness centrality [7, 1] la quale misura per ciascun nodo il numero di passi necessari per raggiungere in media tutti gli altri nodi. la closeness centrality è una metrica molto efficiente su piccoli grafi ma risulta impraticabile su grafi più grandi in quanto presenta una complessità temporale troppo elevata. Altri articoli che presentano metriche di centralità che meritano di essere citati sono i seguenti [5, 6, 9, 4, 2]. Nessuno degli articoli finora citati permette allo stesso tempo di considerare la struttura globale del grafo e di essere applicato su reti complesse in tempi ragionevoli, questo è l'obiettivo che si pone la metrica di centralità Gateway Local Rank (GLR) proposta da C.Salavati, A. Abdollahpouri e Z. Manbari nel 2019 [8]. GLR mira a migliorare le prestazioni temporali di closeness centrality mantenendo la stessa idea di base ma calcolando, per ciascun nodo, invece della distanza verso tutti gli altri nodi la distanza verso un insieme di nodi ritenuti rilevanti. I nodi rilevanti sono identificati suddividendo il grafo in community e applicando una metrica di centralità su ciascuna delle community; il nodo più influente di ciascuna community farà parte dell'insieme dei nodi rilevanti come anche il nodo di ciascuna community avente più vicini fuori dalla propria community. in questo studio è stato analizzato l'algoritmo GLR, implementandolo e testando le sue performance temporali infine, è stata proposta una variante applicabile solo ai grafi non orientati che migliora le prestazioni in termini di tempo di esecuzione. L'algoritmo per calcolare GLR e i risultati ottenuti per quest'ultimo nell'articolo [8] sono spiegati meglio nella sezione (Sez. I-C).

B. Nozioni preliminari

In questa sezione verranno presentate una serie di nozioni fondamentali per comprendere l'implementazione dell'algoritmo GLR e i test eseguiti.

Definizione I-B.1. La modularità è una misura di quanto un grafo sia suddiviso in moduli (community). I grafi con un alto valore di modularità hanno molti archi che connettono i nodi interni a ciascun modulo e pochi che connettono nodi appartenenti a moduli diversi. la modularità può essere calcolata attraverso questa formula:

$$Q = \frac{1}{2m} \sum_{i,j} (A_{i,j} - \gamma \frac{k_i k_j}{2m}) \delta(c_i, c_j)$$

dove m è la somma dei pesi di tutti gli archi (o il numero degli archi se il grafo non è pesato), A è la matrice di adiacenza del grafo, k_i è il grado di i , γ è un parametro chiamato resolution parameter infine $\delta(c_i, c_j)$ è pari a uno se i nodi i e j sono nello stesso modulo altrimenti è pari a zero.

Definizione I-B.2. Il Louvain community detection è un algoritmo per suddividere un grafo in community. L'idea è quella di determinare la partizione del grafo con il massimo valore di modularità ottenibile su quel grafo. Per ottenere questa partizione viene utilizzato un algoritmo euristico che di conseguenza non trova la soluzione ottimale. L'algoritmo può essere suddiviso in due fasi che vengono ripetute iterativamente:

- **Prima Fase:**

- 1) Si crea una community per ciascun nodo composta solo dal nodo stesso;
- 2) Per ciascun nodo si calcolano le variazioni di modularità che si avrebbero rimuovendo il nodo dalla propria community e aggiungendolo a ciascuna delle community vicine;
- 3) Una volta calcolate tutte le variazioni, il nodo viene aggiunto alla community vicina che massimizza la modularità. Se non è possibile nessun miglioramento allora il nodo viene mantenuto in una community separata.
- 4) Si ripetono i passi precedenti fino a che nessuna variazione sulla struttura delle community porti ad un incremento della modularità.

- **Seconda Fase:**

- 1) Si riduce ciascuna community ad un singolo nodo sostituendo gli archi che connettono nodi appartenenti a community differenti con un singolo arco pesato.
- 2) Si esegue nuovamente la prima fase sul grafo così ottenuto.

N.B. L'algoritmo così fatto ha una complessità nel caso peggiore pari a $O(n \cdot \log(n))$.

Definizione I-B.3. BalancedClustering è una classe di algoritmi per determinare una partizione di un grafo nei quali si cerca di assegnare a ciascuna community esattamente lo stesso numero di nodi ($\frac{n}{k}$). In queste tipologie di algoritmi il numero di community è uno dei parametri da impostare.

Definizione I-B.4. Nella teoria dei grafi, betweenness centrality (BC) è una misura della centralità in un grafo basata sui percorsi più brevi. definendo con $\sigma_{u,z}$ il numero di cammini minimi che connettono il nodo u con il nodo z e, con $\sigma_{u,z}(v)$ il numero di cammini minimi che connettono il nodo u con il nodo z passando per v allora:

$$BC(v) = \sum_{\forall s \in V, \forall t \in V \text{ t.c. } s \neq t \neq v} \frac{\sigma_{u,z}(v)}{\sigma_{u,z}}$$

N.B. L'algoritmo per calcolare la BC ha una complessità nel caso peggiore pari a $O(n \cdot m)$.

Definizione I-B.5. Nella teoria dei grafi, local betweenness centrality (LBC) è una variante della misura della centralità BC. Rispetto a BC, in LBC vengono considerati solo i cammini minimi più corti di uno specifico threshold (θ). definendo con $\sigma_{u,z}$ il numero di cammini minimi che connettono il nodo u con il nodo z e, con $\sigma_{u,z}(v, \theta)$ il numero di cammini minimi che connettono il nodo u con il nodo z passando per v più corti di θ allora:

$$LBC(v) = \sum_{\forall s \in V, \forall t \in V \text{ t.c. } s \neq t \neq v} \frac{\sigma_{u,z}(v, \theta)}{\sigma_{u,z}}$$

C. Algoritmo GLR e risultati noti

1) *Pseudocodice*: Di seguito viene riportato lo pseudo codice dell'algoritmo GLR:

Algoritmo I-C.1: GLR

Input: Un grafo $G = (V, E)$ con $|V| = n$, $|E| = m$

Output: I nodi del grafo ordinati in base alla misura di centralità GLR.

```

1: function GLR (G: Graph ) :
2:    $C = \text{Detect communities}$ 
3:   foreach  $c$ : Community in  $C$  :
4:     foreach  $i$ : Node in  $c$  :
5:        $LBC_i^c = \text{compute LBC}$ 
6:        $\text{outDegre}_i = \text{inter-community link of node } i$ 
7:        $K_c = \arg \max_i LBC_i^c$ 
8:        $G_c = \arg \max_i \text{outDegre}_i$ 
9:   foreach  $i$  in  $V$  :
10:     $GLR_i = \text{compute GLR according to}$ 
11:    ranking = ranking node base on  $GLR$  values
12:   return ranking

```

L'algoritmo si articola in una sequenza di passi:

- **riga 2:** L'algoritmo per prima cosa calcola una partizione del grafo. Per calcolare la partizione può essere utilizzato qualsiasi algoritmo di community detection ma in particolare nell'articolo [8] viene utilizzato l'algoritmo Louvain community detection (Def. I-B.2);
- **righe 3-5 e 7:** Per ciascun nodo di ciascuna community si calcola il valore di centralità, con la metrica LBC (Def. I-B.5), considerando esclusivamente il sottografo indotto dai nodi appartenenti alla community. Quindi si determina il nodo critico della community c che è il nodo $K_c \in c$ con il massimo valore di LBC (nella discussione delle prossime sezioni, per semplificare la scrittura, ci si riferirà ai nodi K_c chiamandoli nodi critici);
- **righe 3-6 e 8:** Per ciascun nodo di ciascuna community si calcola il numero di vicini esterni alla community (inter-community degree) quindi si determina il nodo gateway come il nodo, interno alla community, che presenta il massimo valore di inter-community degree. Se più di un nodo ha un valore di inter-community pari a quello massimo, allora si sceglie come gateway quello che fra loro risulta più vicino al nodo critico determinato al passo precedente;
- **righe 9-11:** per ciascun nodo v del grafo viene calcolato il valore GLR_v in accordo alla seguente formula:

$$GLR_v = \frac{1}{\alpha_1 \sum_{u \in \Gamma_k} d(v, u) + \alpha_2 \sum_{p \in \Gamma_G} d(v, u)}$$

dove Γ_k è l'insieme dei nodi critici, Γ_G è l'insieme dei nodi gateway, $d(v, u)$ è la lunghezza del cammino minimo tra il nodo v e il nodo u infine i parametri α_1 e α_2 servono a regolare il peso delle due componenti.

Dopo aver calcolato per ciascun nodo v il valore GLR_v si ordinano i nodi in base alla loro influenza in accordo al valore GLR_v . Più GLR_v sarà grande più il nodo v sarà influente.

Per semplificare la discussione delle prossime sezioni e per coerenza con quanto fatto in fase di analisi si individuano tre subroutine all'interno dell'algoritmo:

- **Partition computation:** riga 2 dell' algoritmo (Alg. I-D);
- **Nodes computation:** righe 3-8 dell'algoritmo (Alg. I-D);
- **GLR:** righe 9-11 dell'algoritmo (Alg. I-D);

2) *Complessità temporale*: Nell'articolo [8] viene calcolata anche la complessità temporale dell'algoritmo che risulta essere secondo l'autore pari a $O(C \cdot n^2)$ dove C è il numero di community e n il numero di nodi del grafo. l'operazione dominante è il calcolo di GLR ripetuto per ciascuno dei nodi che appunto ha complessivamente, secondo l'autore dell'articolo, una complessità di $O(C \cdot n^2)$. Tuttavia l'autore non spiega quale algoritmo utilizzare per ottenere tale complessità e di conseguenza nella mia implementazione ho utilizzato la BFS (dato che tutti i test sono stati eseguiti su grafi non pesati) ripetuta per ciascuna dei nodi e terminando l'esplorazione una volta raggiunti tutti i nodi target (nodi critici e gateway). Rieseguendo l'analisi computazionale asintotica si ottiene quindi una complessità pari a $O(n \cdot (n + m))$, dal momento che $m = O(n^2)$ e che $C \ll n$, la complessità della mia implementazione risulta maggiore di quella stimata nell'articolo.

3) *Prestazioni dell'algoritmo*: Per valutare la qualità della soluzione ottenuta grazie all'algoritmo GLR, nell'articolo [8], sono riportati i risultati ottenuti eseguendo l'algoritmo a confronto con diverse altre metriche di centralità. Il confronto più interessante è quello con closeness centrality (CC) in quanto GLR si pone come sua alternativa con complessità temporale minore; si osserva che GLR ha performance simili a quelle di CC e spesso anche migliori.

D. Obiettivo dello studio

Nell'articolo [8], nel quale l'algoritmo GLR è stato presentato, sono stati eseguiti una serie di esperimenti che avevano il principale obiettivo di valutare la qualità della soluzione offerta dall'algoritmo GLR (Alg. I-D) al problema di ordinare i nodi di un grafo in base alla loro capacità di diffusione. Non è stata fatta invece un'analisi sperimentale della complessità temporale, che è invece l'oggetto della presente relazione. Inizialmente l'obiettivo dello studio, oltre a essere quello di accertare con esperimenti pratici la complessità temporale, era anche quello di valutare se cambiando l'algoritmo di centralità (LBC) utilizzato localmente all'interno delle community si riuscisse ad ottenere ulteriori miglioramenti in termini temporali, senza perdere eccessivamente sulla qualità della soluzione. Fin dai primi esperimenti si è però subito osservato che il calcolo locale (righe 3-8 Alg. I-D) era molto efficiente e che quasi la totalità del tempo veniva speso invece nel calcolo dei valori GLR di ciascun nodo (righe 9-11 Alg. I-D). Infatti mentre il calcolo locale spesso impiegava meno di un secondo il calcolo dei valori GLR impiegava talvolta anche minuti. Quindi dopo aver verificato la complessità temporale dell'algoritmo e dopo aver notato la problematica sopra espressa la concentrazione si è spostata nel cercare un modo per calcolare i valori GLR in modo più efficiente. Il risultato di questa analisi è stato quello di determinare un modo più efficiente per calcolare i valori GLR applicabile ai grafi non orientati. Infine anche per quest'ultima versione dell'algoritmo sono stati eseguiti dei test per verificare la complessità temporale teorica.

Determinare i valori GLR di ciascun nodo è oneroso in quanto bisogna calcolare le distanze ($d(v, u)$) da ciascun nodo del grafo v ai nodi di interesse (nodi critici e gateway). Per far ciò in grafi orientati bisogna applicare n volte la BFS ovvero una volta per nodo del grafo. Nel caso di grafi non orientati invece si può sfruttare la proprietà secondo cui $d(v, u) = d(u, v)$ e quindi invece di eseguire n volte la BFS la si può eseguire $2C$ volte ovvero, una volta per ogni nodo di interesse. In questo modo invece di calcolare la distanza da tutti i nodi del grafo ai nodi di interesse si calcola la distanza dai nodi di interesse a tutti i nodi del grafo quindi, la complessità temporale della subroutine GLR sarà pari a $O(C \cdot (n + m))$. Per una migliore comprensione vengono mostrati di seguito gli pseudo codici sia della versione non ottimizzata sia di quella ottimizzata.

Algoritmo I-D.1: GLR_subroutine

Input: Un grafo $G = (V, E)$, la lista dei nodi critici K e la lista dei gateway $Gateways$ con $|V| = n$, $|E| = m$

Output: lista di $GLR_v = \frac{1}{\alpha_1 \sum_{u \in \Gamma_k} d(v, u) + \alpha_2 \sum_{p \in \Gamma_G} d(v, u)}$

```

1: function GLR_subroutine (G: Graph, K: List<  $K_c$  >, Gateways: List<  $G_c$  >) :
2:   foreach  $v$ : Node in  $V$  :
3:      $distK_v$  = use BFS to compute distances between  $v$  and all nodes  $\in K$ 
4:      $distG_v$  = use BFS to compute distances between  $v$  and all nodes  $\in Gateways$ 
5:      $sumDistK_v$  = sum( $distK_v$ )
6:      $sumDistG_v$  = sum( $distG_v$ )
7:      $GLR_v = 1 / (\alpha_1 \cdot sumDistK_v + \alpha_2 \cdot sumDistG_v)$ 
8:   return L s.t  $GLR_v \in L \Leftrightarrow v \in V$ 

```

Algoritmo I-D.2: GLR_subroutine (ottimizzata)

Input: Un grafo $G = (V, E)$, la lista dei nodi critici K e la lista dei gateway $Gateways$ con $|V| = n$, $|E| = m$

Output: lista di $GLR_v = \frac{1}{\alpha_1 \sum_{u \in \Gamma_k} d(v, u) + \alpha_2 \sum_{p \in \Gamma_G} d(v, u)}$

```

1: function GLR_subroutine (G: Graph, K: List<  $K_c$  >, Gateways: List<  $G_c$  >) :
2:   foreach  $c$ : Community in  $C$  :
3:      $distK_c$  = use BFS to compute distances between  $K_c$  and all nodes  $\in V$ 
4:      $distG_c$  = use BFS to compute distances between  $G_c$  and all nodes  $\in V$ 
5:   foreach  $v$ : Node in  $V$  :
6:      $sumDistK_v = 0$ 
7:      $sumDistG_v = 0$ 
8:     foreach  $c$ : Community in  $C$  :
9:        $sumDistK_v += distK_c(v)$ 
10:       $sumDistG_v += distG_c(v)$ 
11:      $GLR_v = 1 / (\alpha_1 \cdot sumDistK_v + \alpha_2 \cdot sumDistG_v)$ 
12:   return L s.t  $GLR_v \in L \Leftrightarrow v \in V$ 

```

Lo stesso principio può essere sfruttato per abbassare il tempo necessario a determinare i nodi di gateway. Infatti quando più di un nodo ha lo stesso inter-community degree, nel caso di grafi non orientati, invece di calcolare la distanza tra ciascun candidato gateway e il nodo critico della community di interesse, si può calcolare la distanza tra il nodo critico e tutti i candidati gateway eseguendo a prescindere una sola volta la BFS invece di eseguirla una volta per candidato gateway.

N.B. Per quanto l'algoritmo ottimizzato funzioni formalmente solo su grafi non orientati, si osserva che lo stesso principio può essere generalizzato anche a grafi orientati applicando le subroutine ottimizzate invece che sul grafo G , sul grafo G' avente gli stessi nodi e gli stessi archi orientati però nella direzione opposta a quelli del grafo G .

II. VALUTAZIONE DEGLI ESPERIMENTI

A. Ambiente di Test

1) *Configurazione dell'ambiente*: Il codice per l'implementazione dell'algoritmo e per impostare gli esperimenti è disponibile su GitHub (link). Si osserva un'organizzazione in directory per favorire la fruizione del codice:

analysis	add statistics to experiments	2 days ago
c++_implementation	done exp1 replicated	3 days ago
double_experiment	done exp4 repetition	2 days ago
graphs	setup exp4	last week
partial_results/partitions	setup exp3	last week
python_implementation	done exp1 replicated	3 days ago
results	add statistics to experiments	2 days ago
utility	delete useless file (next exp4)	4 days ago
README.md	added run instructions	5 months ago
requirements.txt	update requirements.txt	5 months ago

Fig. 1: Organizzazione del codice

Si rimanda al file README.md, mostrato in figura (Fig. 1) e presente sulla repository, per la configurazione dell'ambiente e per l'installazione delle librerie necessarie.

2) *Implementazioni degli algoritmi*: I due algoritmi (GLR con subroutine ottimizzate e non Sez. I-D) sono entrambi stati implementati sia in python che in C++. In particolare si è scelto di utilizzare python per la sua semplicità e versatilità che ha permesso di avere in tempi rapidi un'implementazione funzionante, quindi eseguire dei test prima di implementare gli algoritmi in C++, che per quanto più efficiente e quindi più adatto per valutare le performance in un caso reale, richiede generalmente più tempo per lo sviluppo.

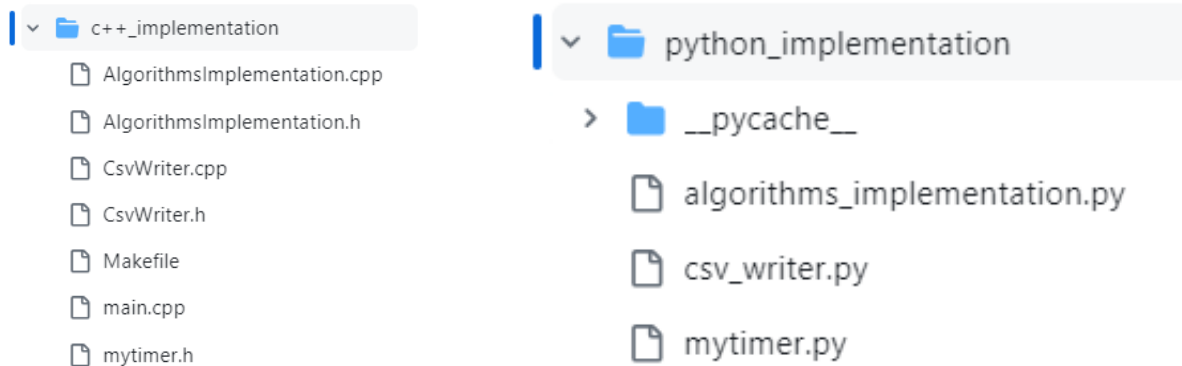


Fig. 2: File di implementazione degli algoritmi

Come si può osservare dalla figura (Fig. 2) l'organizzazione dei file dedicati all'algoritmo è la medesima. Per python il main è il file algorithms_implementation.py mentre per C++ il main è il file main.cpp. Inoltre ci sono una serie di file di ausilio per il salvataggio dell'output su file csv e per misurare le performance temporali. In più per l'implementazione C++ è stato necessario il file MakeFile necessario per compilare il codice in una tra tre possibili modalità:

- **debug**: utilizzato per scopi di debug;
- **release**: utilizzato per ottenere il massimo delle performance;
- **valgrind**: non rilevante ai nostri scopi.

Per quanto riguarda l'interfaccia di input si è scelto di utilizzare le opzioni in fase di esecuzione:

```
python3 algorithms_implementation.py -g <GraphPath> -f <Flag> -p <PartitionPath> -u
./main -g <GraphPath> -f <Flag> -p <PartitionPath> -u true
```

le opzioni che vengono passate in fase di esecuzione permettono di impostare diversi aspetti dell'esecuzione:

- **-g** Bisogna passare il path del file dove è salvato il grafo in formato METIS (in caso di esecuzione python se non passato viene richiesto attraverso standard input);
- **-f** può essere passata qualsiasi stringa che sarà utilizzata in fase di salvataggio dei risultati in modo da poter ricercare i risultati in base a tale valore (di default è impostata a "prova");
- **-p** Se passata la partizione non verrà generata ma letta da file. Questo è stato utile in fase di esperimenti per riuscire ad ottenere gli stessi risultati anche in esecuzioni differenti, infatti il Louvain community detection ha una componente randomica non controllabile attraverso l'interfaccia del metodo che lo implementa. (se non passata la partizione verrà generata in fase di esecuzione);
- **-u** Serve per discriminare se utilizzare o meno nell'esecuzione l'ottimizzazione per grafi non orientati, se l'opzione viene passata allora si utilizzerà la versione dell'algoritmo ottimizzata (se si usa python non è necessario passare alcun valore mentre se si usa C++ si deve passare un valore ma qualsiasi valore venga passato verrà sempre eseguito la versione dell'algoritmo ottimizzata, per eseguire la versione non ottimizzata è sufficiente non inserire l'opzione).

Per quanto riguarda l'interfaccia di output si è scelto di salvare tutti i risultati su file in modo da renderli fruibili in fase di analisi.

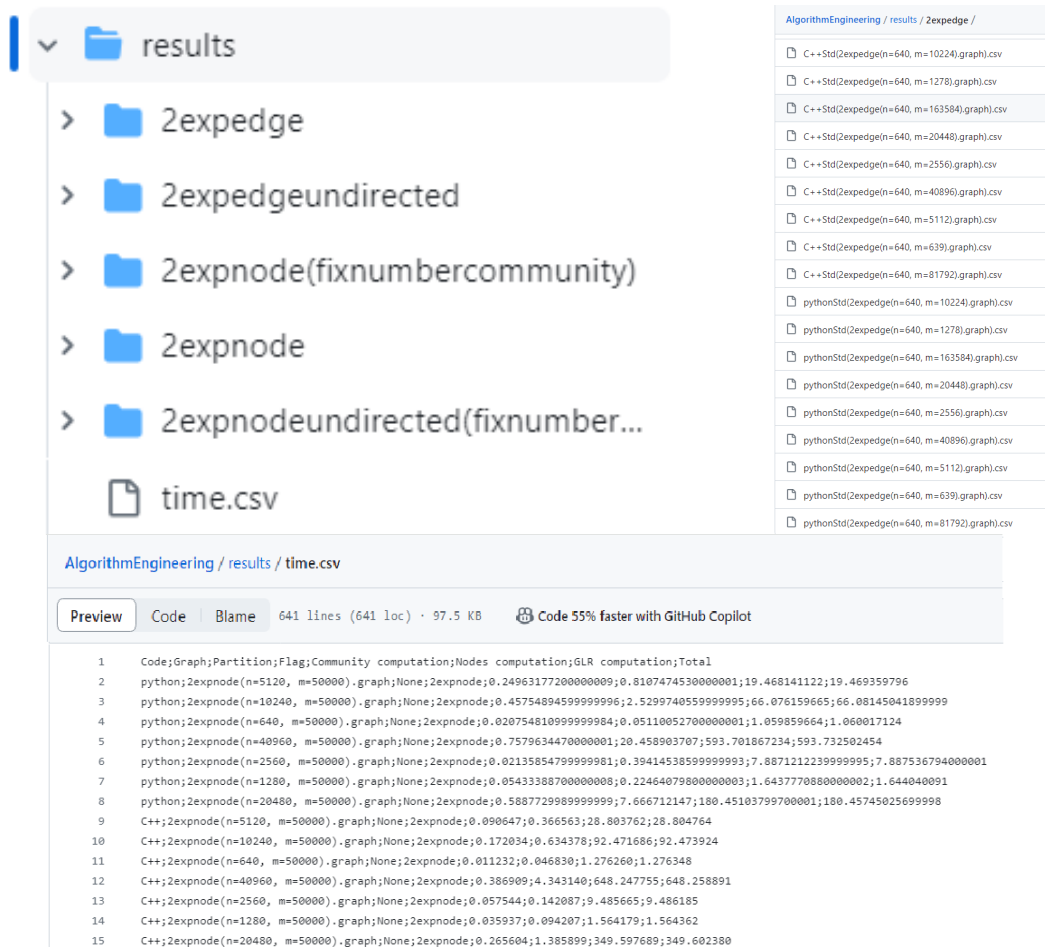


Fig. 3: Organizzazione dei risultati

Una volta terminata l'esecuzione degli algoritmi i risultati vengono salvati in un file nella cartella results il cui nome è composto da un prefisso ,dipendente dal linguaggio di programmazione utilizzato, seguito dal nome del grafo su cui è stato applicato l'algoritmo. I risultati che volevano essere salvati permanentemente sono stati poi spostati in sotto cartelle aventi come nome la flag associata all'esperimento. Per quanto riguarda invece le performance temporali, ad ogni esecuzione viene aggiunta una riga riportante il tempo complessivo e i tempi parziali necessari a completare le sotto procedure.

3) *Generazione dei grafi*: All'interno della cartella utility è presente la sotto cartella graph_generators che mantiene i file python per la generazione dei grafi utilizzati negli esperimenti. Nello sviluppo del generatore dei grafi sono stati utilizzati i principi dell'OOP per rendere semplice l'aggiunta di nuovi generatori e per fare altrettanto nella configurazione degli stessi. A questo scopo è stato inserito un file di configurazione tramite il quale si può scegliere il generatore da utilizzare e i parametri da passare.

```

1  {
2    "generator": "StdGraphGenerator" # Nome del generatore da utilizzare,
3    "StdGraphGenerator": { # Impostazioni del generatore StdGraphGenerator
4      "graph_flag": "2expedge" # Prefisso del file in cui salvare il grafo,
5      "result_folder": "../../graphs/2expedge" # Cartella dove salvare i grafi,
6      "n": 640 # Numero di nodi del primo grafo che si andrà a generare,
7      "m": 639 # Numero di archi del primo grafo che si andrà a generare,
8      "doubling_n_factor": 1 # Fattore moltiplicativo per aumentare il numero di nodi ad ogni iterazione,
9      "doubling_m_factor": 2 # Fattore moltiplicativo per aumentare il numero di archi ad ogni iterazione,
10     "max_number_of_graph": 10 # Massimo numero di grafi da generare
11   },
12   "AlbertGraphGenerator": { # Impostazioni del generatore AlbertGraphGenerator
13     "graph_flag": "Albert", # Prefisso del file in cui salvare il grafo,
14     "result_folder": "../../graphs/albert" # Cartella dove salvare i grafi,
15     "start_node_number": 160 # Numero di nodi iniziali,
16     "max_node_number": 30000 # Numero di nodi finali
17   }
18 }

```

Fig. 4: File di configurazione per la generazione di grafi con spiegazione di ciascun parametro

Si sono implementati due generatori:

- **StdGraphGenerator**: generatore personalizzato di grafi non orientati, non pesati e completamente connessi utile per avere controllo sul numero di nodi e archi facendoli variare attraverso un parametro moltiplicativo configurabile;
- **AlbertGraphGenerator**: usa al suo interno il generatore di libreria "BarabasiAlbertGenerator" facendo raddoppiare il numero di nodi ad ogni nuova generazione.

Per gli scopi degli esperimenti fatti sono stati generati due tipologie di grafi entrambi con il generatore "StdGraphGenerator". Di seguito vengono riportate le due configurazioni utilizzate.

```

1  {
2    "generator": "StdGraphGenerator",
3    "StdGraphGenerator": {
4      "graph_flag": "2expnode",
5      "result_folder": "../../graphs/2expnode",
6      "n": 640,
7      "m": 5000,
8      "doubling_n_factor": 2,
9      "doubling_m_factor": 1,
10     "max_number_of_graph": 10
11   },
12   "AlbertGraphGenerator": {
13     "graph_flag": "Albert",
14     "result_folder": "../../graphs/albert",
15     "start_node_number": 160,
16     "max_node_number": 30000
17   }
18 }

```

Fig. 5: Genera massimo 10 grafi randomici completamente connessi con $m=5000$ e $n = 640 \cdot 2^i$ con $0 \leq i < 10$ e $m > n \cdot (n-1)/2$ (I grafi sono salvati in "graphs/2expnode")

```

1  {
2    "generator": "StdGraphGenerator",
3    "StdGraphGenerator": {
4      "graph_flag": "2expedge",
5      "result_folder": "../../graphs/2expedge",
6      "n": 640,
7      "m": 639,
8      "doubling_n_factor": 1,
9      "doubling_m_factor": 2,
10     "max_number_of_graph": 10
11   },
12   "AlbertGraphGenerator": {
13     "graph_flag": "Albert",
14     "result_folder": "../../graphs/albert",
15     "start_node_number": 160,
16     "max_node_number": 30000
17   }
18 }

```

Fig. 6: Genera massimo 10 grafi randomici completamente connessi con $n=640$ e $m = 639 \cdot 2^i$ con $0 \leq i < 10$ e $m > n \cdot (n-1)/2$ (I grafi sono salvati in "graphs/2expedge")

4) *Generazione delle partizioni*: All'interno della cartella utility è presente la sotto cartella partition_generators che mantiene i file python per la generazione delle partizioni utilizzate negli esperimenti. Analogamente al caso dei generatori dei grafi, anche per quelli delle partizioni si è cercato di rendere il più semplice possibile la definizione di nuovi generatori e la configurazione di quelli esistenti. A questo scopo è stato inserito un file di configurazione tramite il quale si può scegliere il generatore da utilizzare e i suoi parametri.

```

1 {
2   "filter_keyword": "2expnode" # Utile per filtrare i grafi che non contengono questa stringa nel loro nome,
3   "strategy_name": "BalancedClusteringDoubling" # Nome del generatore da utilizzare ,
4   "strategies": {
5     "PLM": { # Configurazione del generatore PLM
6       "graphs_path": "../graphs/2expnode" # Cartella dove trovare i grafi per cui calcolare la partizione,
7       "output_folder": "../partial_results/partitions/2expnode" # Cartella dove salvare la partizione generata
8     },
9     "BalancedClusteringDoubling": { # Configurazione del generatore BalancedClusteringDoubling
10      "graphs_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph" # Grafo per cui calcolare la partizione,
11      "min_number_communities": 2 # Numero di community che deve avere la prima partizione generata,
12      "max_number_communities": 128 # Raddoppiare il numero di community e generare nuove partizioni finchè il numero di community è minore di questo valore,
13      "output_folder": "../partial_results/partitions" # Cartella dove salvare la cartella <graphs_name> dove salvare i risultati
14    },
15    "BalancedClustering": { # Configurazione del generatore BalancedClustering
16      "graphs_path": "../graphs/2expnode" # Cartella dove trovare i grafi per cui calcolare la partizione,
17      "number_communities": 10 # Numero di community che devono avere le partizioni generate,
18      "output_folder": "../partial_results/partitions" # Cartella dove salvare la cartella partition<number_communities> dove salvare i risultati
19    }
20  }
21 }

```

Fig. 7: File di configurazione per la generazione di partizioni con spiegazione di ciascun parametro

Si sono implementati tre generatori:

- **PLM**: genera per ciascun grafo, passato come parametro, una partizione utilizzando la funzione di libreria PLM ovvero L'algoritmo Louvain community detection (Def. I-B.2);
- **BalancedClustering**: genera per ciascun grafo, passato come parametro, una partizione utilizzando la funzione di libreria makeContinuousBalancedClustering ovvero un algoritmo di tipo BalancedClustering (Def. I-B.3);
- **BalancedClusteringDoubling**: genera per un singolo grafo, passato come parametro, una serie di partizioni raddoppiando ad ogni generazione il numero di community utilizzando la funzione di libreria makeContinuousBalancedClustering.

Per gli scopi degli esperimenti fatti sono stati generati due tipologie di grafi. Di seguito vengono riportate le due configurazioni utilizzate.

```

1 {
2   "filter_keyword": "2expnode",
3   "strategy_name": "BalancedClusteringDoubling",
4   "strategies": {
5     "PLM": {
6       "graphs_path": "../graphs/2expnode",
7       "output_folder": "../partial_results/partitions/2expnode"
8     },
9     "BalancedClusteringDoubling": {
10      "graphs_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph",
11      "min_number_communities": 2,
12      "max_number_communities": 128,
13      "output_folder": "../partial_results/partitions"
14    },
15     "BalancedClustering": {
16       "graphs_path": "../graphs/2expnode",
17       "number_communities": 10,
18       "output_folder": "../partial_results/partitions"
19     }
20   }
21 }

```

Fig. 8: Genera una serie di partizioni con $1 \leq i \leq 7$ dove il numero di community $k = 2 \cdot i$ (Le partizioni sono salvati in "partial_results/partitions/2expnode(n=20480, m=50000)")

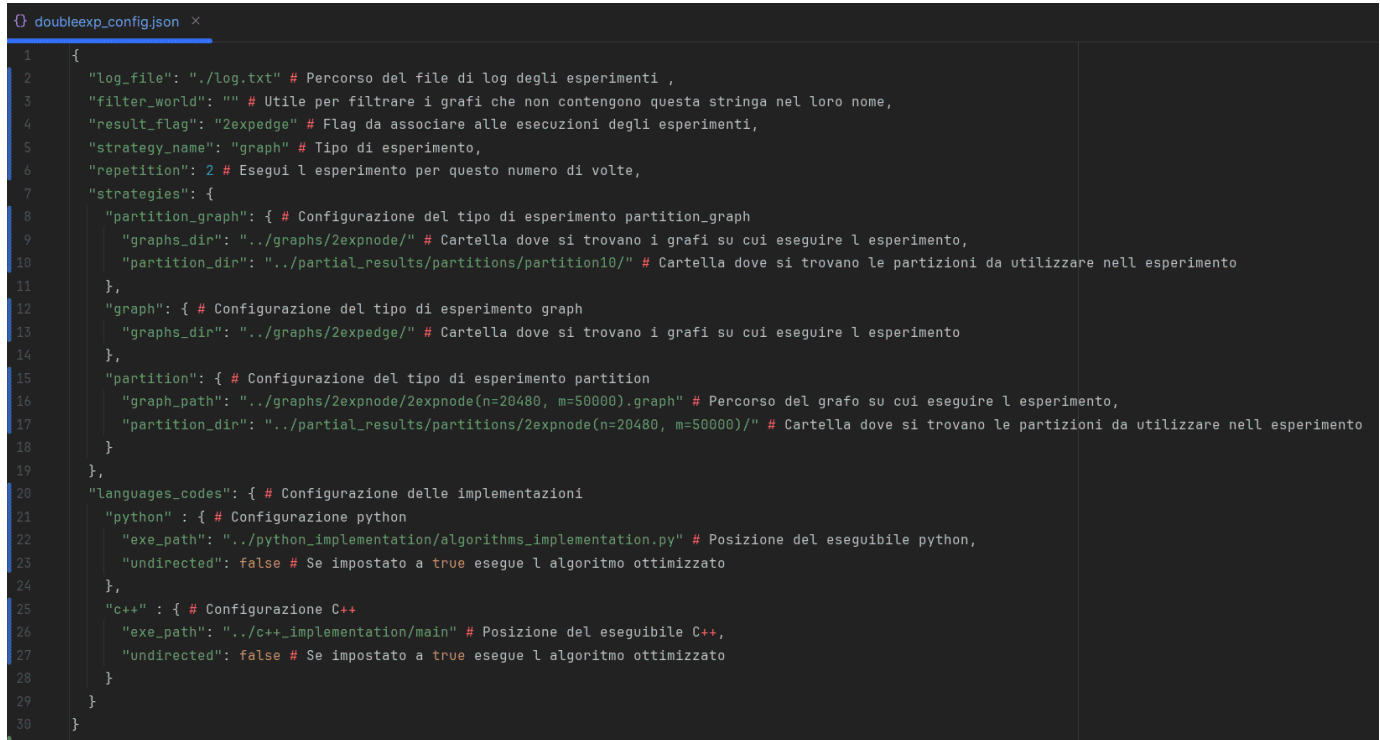
```

1 {
2   "filter_keyword": "2expnode",
3   "strategy_name": "BalancedClustering",
4   "strategies": {
5     "PLM": {
6       "graphs_path": "../graphs/2expnode",
7       "output_folder": "../partial_results/partitions/2expnode"
8     },
9     "BalancedClusteringDoubling": {
10      "graphs_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph",
11      "min_number_communities": 2,
12      "max_number_communities": 128,
13      "output_folder": "../partial_results/partitions"
14    },
15     "BalancedClustering": {
16       "graphs_path": "../graphs/2expnode",
17       "number_communities": 10,
18       "output_folder": "../partial_results/partitions"
19     }
20   }
21 }

```

Fig. 9: Genera una partizione per ciascun grafo in "graphs/2expnode" dove il numero di community $k = 10$ (I grafi sono salvati in "partial_results/partitions/partition10")

5) *Configurazione degli esperimenti*: All'interno della cartella "double_experiment" sono presenti i file che servono per eseguire gli esperimenti. come anche nel caso dei generatori è implementato in maniera tale che sia facilmente estendibile e si può facilmente scegliere l'esperimento da eseguire grazie ad un file di configurazione.



```

1 {
2   "log_file": "./log.txt" # Percorso del file di log degli esperimenti ,
3   "filter_world": "" # Utile per filtrare i grafi che non contengono questa stringa nel loro nome,
4   "result_flag": "2expedge" # Flag da associare alle esecuzioni degli esperimenti,
5   "strategy_name": "graph" # Tipo di esperimento,
6   "repetition": 2 # Esegui l'esperimento per questo numero di volte,
7   "strategies": {
8     "partition_graph": { # Configurazione del tipo di esperimento partition_graph
9       "graphs_dir": "../graphs/2expnode/" # Cartella dove si trovano i grafi su cui eseguire l'esperimento,
10      "partition_dir": "../partial_results/partitions/partition10/" # Cartella dove si trovano le partizioni da utilizzare nell'esperimento
11    },
12    "graph": { # Configurazione del tipo di esperimento graph
13      "graphs_dir": "../graphs/2expedge/" # Cartella dove si trovano i grafi su cui eseguire l'esperimento
14    },
15    "partition": { # Configurazione del tipo di esperimento partition
16      "graph_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph" # Percorso del grafo su cui eseguire l'esperimento,
17      "partition_dir": "../partial_results/partitions/2expnode(n=20480, m=50000)/" # Cartella dove si trovano le partizioni da utilizzare nell'esperimento
18    }
19  },
20  "languages_codes": { # Configurazione delle implementazioni
21    "python": { # Configurazione python
22      "exe_path": "../python_implementation/algorithms_implementation.py" # Posizione del eseguibile python,
23      "undirected": false # Se impostato a true esegue l'algoritmo ottimizzato
24    },
25    "c++": { # Configurazione C++
26      "exe_path": "../c++_implementation/main" # Posizione del eseguibile C++,
27      "undirected": false # Se impostato a true esegue l'algoritmo ottimizzato
28    }
29  }
30 }

```

Fig. 10: File di configurazione per gestire gli esperimenti con spiegazione di ciascun parametro

Il parametro più rilevante è la strategia da utilizzare. In particolare la strategia stabilisce come vengono scelti i grafi e le partizioni da utilizzare negli esperimenti. Per gli scopi di questa relazione sono stati definiti tre strategie:

- **graph**: Si esegue l'algoritmo su tutti i grafi presenti nella cartella passata come parametro senza passare la partizione come opzione quindi generandola in fase di esecuzione.
- **partition**: Si esegue l'algoritmo sul grafo passato come parametro e sulle partizioni presenti nella cartella passata come parametro.
- **partition_graph**: Si esegue l'algoritmo su tutti i grafi e partizioni, aventi lo stesso nome, presenti nelle cartelle passate come parametro.

B. Impostazione degli esperimenti

In questa sezione vengono riportate le configurazioni degli esperimenti fatti, sia in forma testuale che con i valori da assegnare ai parametri del file di configurazione (Path: double_experiment/doubleexp_config.json) per replicare l'esperimento. In totale sono stati fatti 7 doubling experiment, 4 per verificare le performance dell'algoritmo non ottimizzato e 3 per analizzare la versione ottimizzata. Ciascuno degli esperimenti è stato realizzato allo scopo di analizzare l'influenza di ciascun parametro sulle performance temporale dei due algoritmi, inoltre alcuni dei risultati sono stati utilizzati anche in fase di analisi della correttezza per assicurarsi che tutte le implementazioni restituiscano lo stesso risultato. Si osserva che tutti gli esperimenti sono stati eseguiti più volte per assicurarsi una discreta accuratezza statistica e riproducibilità.

Esperimento 1. :

```
doubleexp_config.json
1 {
2   "log_file": "./log.txt",
3   "filter_world": "",
4   "result_flag": "2expnode",
5   "strategy_name": "graph",
6   "repetition": 3,
7   "strategies": {
8     "partition_graph": {
9       "graphs_dir": "../graphs/2expnode/",
10      "partition_dir": "../partial_results/partitions/partition10/"
11    },
12    "graph": {
13      "graphs_dir": "../graphs/2expnode/"
14    },
15    "partition": {
16      "graph_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph",
17      "partition_dir": "../partial_results/partitions/2expnode(n=20480, m=50000)/"
18    }
19  },
20  "languages_codes": {
21    "python": {
22      "exe_path": "../python_implementation/algorithms_implementation.py",
23      "undirected": false
24    },
25    "c++": {
26      "exe_path": "../c++_implementation/main",
27      "undirected": false
28    }
29  }
30 }
```

- **Algoritmo:** GLR non ottimizzato per l'esecuzione su grafi non orientati;
- **Performance metric:** tempo di esecuzione;
- **Performance indicator:** tempo di CPU;
- **Factor:** numero di nodi del grafo (n) e numero di archi del grafo (m);
- **Noise parameter:** numero di community della partizione generata (C) e numero di nodi per partizione (n_c);
- **Design Point:**
 - $n=(640,1280,2560,5120,10240,20480,40960)$;
 - $m=50000$.
- **Obiettivo:** verificare come varia il tempo di esecuzione dell'algoritmo al variare del numero di nodi del grafo su cui l'algoritmo viene eseguito.

Esperimento 2. :

```
doubleexp_config.json
1 {
2   "log_file": "./log.txt",
3   "filter_world": "",
4   "result_flag": "2expnode(fixnumbercommunity)",
5   "strategy_name": "partition_graph",
6   "repetition": 3,
7   "strategies": {
8     "partition_graph": {
9       "graphs_dir": "../graphs/2expnode/",
10      "partition_dir": "../partial_results/partitions/partition10/"
11    },
12    "graph": {
13      "graphs_dir": "../graphs/2expnode/"
14    },
15    "partition": {
16      "graph_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph",
17      "partition_dir": "../partial_results/partitions/2expnode(n=20480, m=50000)/"
18    }
19  },
20  "languages_codes": {
21    "python": {
22      "exe_path": "../python_implementation/algorithms_implementation.py",
23      "undirected": false
24    },
25    "c++": {
26      "exe_path": "../c++_implementation/main",
27      "undirected": false
28    }
29  }
30 }
```

- **Algoritmo:** GLR non ottimizzato per l'esecuzione su grafi non orientati;
- **Performance metric:** tempo di esecuzione;
- **Performance indicator:** tempo di CPU;
- **Factor:** numero di nodi del grafo (n), numero di archi del grafo (m) e numero di community delle partizioni (C);
- **Noise parameter:** numero di nodi per partizione (n_c);
- **Design Point:**
 - $n=(640,1280,2560,5120,10240,20480,40960)$;
 - $m=50000$;
 - $C=10$.
- **Obiettivo:** verificare come varia il tempo di esecuzione dell'algoritmo al variare del numero di nodi del grafo su cui l'algoritmo viene eseguito eliminando, tra i parametri di disturbo, il numero di community della partizione utilizzata. I risultati ottenuti sono stati utilizzati anche in fase di analisi della correttezza per assicurarsi che tutte le implementazioni a partire dallo stesso input restituivano lo stesso output.

Esperimento 3. :

```

doubleexp_config.json x
1 {
2   "log_file": "./log.txt",
3   "filter_world": "",
4   "result_flag": "2expcommunity",
5   "strategy_name": "partition",
6   "repetition": 3,
7   "strategies": {
8     "partition_graph": {
9       "graphs_dir": "../graphs/2expnode/",
10      "partition_dir": "../partial_results/partitions/partition10/"
11    },
12    "graph": {
13      "graphs_dir": "../graphs/2expnode/"
14    },
15    "partition": {
16      "graph_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph",
17      "partition_dir": "../partial_results/partitions/2expnode(n=20480, m=50000)/"
18    }
19  },
20  "languages_codes": {
21    "python": {
22      "exe_path": "../python_implementation/algorithms_implementation.py",
23      "undirected": false
24    },
25    "c++": {
26      "exe_path": "../c++_implementation/main",
27      "undirected": false
28    }
29  }
30 }

```

- **Algoritmo:** GLR non ottimizzato per l'esecuzione su grafi non orientati;
- **Performance metric:** tempo di esecuzione;
- **Performance indicator:** tempo di CPU;
- **Factor:** numero di nodi del grafo (n), numero di archi del grafo (m) e numero di community della partizione generata (C);
- **Noise parameter:** numero di nodi per partizione (n_c);
- **Design Point:**
 - $n=20480$;
 - $m=50000$;
 - $C=(2,4,8,16,32,64,128)$
- **Obbiettivo:** verificare come varia il tempo di esecuzione dell'algoritmo al variare del numero di community della partizione utilizzata.

Esperimento 4. :

```

doubleexp_config.json x
1 {
2   "log_file": "./log.txt",
3   "filter_world": "",
4   "result_flag": "2expedge",
5   "strategy_name": "graph",
6   "repetition": 3,
7   "strategies": {
8     "partition_graph": {
9       "graphs_dir": "../graphs/2expnode/",
10      "partition_dir": "../partial_results/partitions/partition10/"
11    },
12    "graph": {
13      "graphs_dir": "../graphs/2expedge/"
14    },
15    "partition": {
16      "graph_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph",
17      "partition_dir": "../partial_results/partitions/2expnode(n=20480, m=50000)/"
18    }
19  },
20  "languages_codes": {
21    "python": {
22      "exe_path": "../python_implementation/algorithms_implementation.py",
23      "undirected": false
24    },
25    "c++": {
26      "exe_path": "../c++_implementation/main",
27      "undirected": false
28    }
29  }
30 }

```

- **Algoritmo:** GLR non ottimizzato per l'esecuzione su grafi non orientati;
- **Performance metric:** tempo di esecuzione;
- **Performance indicator:** tempo di CPU;
- **Factor:** numero di nodi del grafo (n) e numero di archi del grafo (m);
- **Noise parameter:** numero di community della partizione generata (C) e numero di nodi per partizione (n_c);
- **Design Point:**
 - $n=640$;
 - $m=(639,1278,2556,5112,10224,20448,40896,81792,163584)$.
- **Obbiettivo:** verificare come varia il tempo di esecuzione dell'algoritmo al variare del numero di archi del grafo su cui l'algoritmo viene eseguito.

Esperimento 5. :

```

doubleexp_config.json x
1 {
2   "graphs_dir": "../graphs/2expnode/",
3   "log_file": "../log.txt",
4   "filter_world": "",
5   "result_flag": "2expnodeundirected(fixpartition)",
6   "load_partition": true,
7   "strategy_name": "partition_graph",
8   "repetition": 10,
9   "strategies": {
10    "partition_graph": {
11      "graphs_dir": "../graphs/2expnode/",
12      "partition_dir": "../partial_results/partitions/partition10/"
13    },
14    "graph": {
15      "graphs_dir": "../graphs/2expedge/"
16    },
17    "partition": {
18      "graph_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph",
19      "partition_dir": "../partial_results/partitions/2expnode(n=20480, m=50000)/"
20    }
21  },
22  "languages_codes": {
23    "python": {
24      "exe_path": "../python_implementation/algorithms_implementation.py",
25      "undirected": true
26    },
27    "c++": {
28      "exe_path": "../c++_implementation/main",
29      "undirected": true
30    }
31  }
32 }

```

- **Algoritmo:** GLR ottimizzato per l'esecuzione su grafi non orientati;
- **Performance metric:** tempo di esecuzione;
- **Performance indicator:** tempo di CPU;
- **Factor:** numero di nodi del grafo (n), numero di archi del grafo (m) e numero di community delle partizioni (C);
- **Noise parameter:** numero di nodi per partizione (n_c);
- **Design Point:**
 - $n=(640,1280,2560,5120,10240,20480,40960)$;
 - $m=50000$;
 - $C=10$.
- **Obbiettivo:** verificare come varia il tempo di esecuzione dell'algoritmo al variare del numero di nodi del grafo su cui l'algoritmo viene eseguito eliminando, tra i parametri di disturbo, il numero di community della partizione utilizzata. I risultati ottenuti sono stati utilizzati anche in fase di analisi della correttezza per assicurarsi, che tutte le implementazioni a partire dallo stesso input restituivano lo stesso output.

Esperimento 6. :

```

doubleexp_config.json x
1 {
2   "graphs_dir": "../graphs/2expedge/",
3   "log_file": "../log.txt",
4   "filter_world": "",
5   "result_flag": "2expcommunityundirected",
6   "load_partition": true,
7   "strategy_name": "partition",
8   "repetition": 10,
9   "strategies": {
10    "partition_graph": {
11      "graphs_dir": "../graphs/2expnode/",
12      "partition_dir": "../partial_results/partitions/partition10/"
13    },
14    "graph": {
15      "graphs_dir": "../graphs/2expedge/"
16    },
17    "partition": {
18      "graph_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph",
19      "partition_dir": "../partial_results/partitions/2expnode(n=20480, m=50000)/"
20    }
21  },
22  "languages_codes": {
23    "python": {
24      "exe_path": "../python_implementation/algorithms_implementation.py",
25      "undirected": true
26    },
27    "c++": {
28      "exe_path": "../c++_implementation/main",
29      "undirected": true
30    }
31  }
32 }

```

- **Algoritmo:** GLR ottimizzato per l'esecuzione su grafi non orientati;
- **Performance metric:** tempo di esecuzione;
- **Performance indicator:** tempo di CPU;
- **Factor:** numero di nodi del grafo (n), numero di archi del grafo (m) e numero di community della partizione generata (C);
- **Noise parameter:** numero di nodi per partizione (n_c);
- **Design Point:**
 - $n=20480$;
 - $m=50000$;
 - $C=(2,4,8,16,32,64,128)$
- **Obbiettivo:** verificare come varia il tempo di esecuzione dell'algoritmo al variare del numero di community della partizione utilizzata.

Esperimento 7. :

```

doubleexp_config.json
1  {
2    "graphs_dir": "../graphs/2expedge/",
3    "log_file": "../log.txt",
4    "filter_world": "",
5    "result_flag": "2expedgeundirected",
6    "load_partition": true,
7    "strategy_name": "graph",
8    "repetition": 10,
9    "strategies": {
10     "partition_graph": {
11       "graphs_dir": "../graphs/2expnode/",
12       "partition_dir": "../partial_results/partitions/partition10/"
13     },
14     "graph": {
15       "graphs_dir": "../graphs/2expedge/"
16     },
17     "partition": {
18       "graph_path": "../graphs/2expnode/2expnode(n=20480, m=50000).graph",
19       "partition_dir": "../partial_results/partitions/2expnode(n=20480, m=50000)/"
20     }
21   },
22   "languages_codes": {
23     "python": {
24       "exe_path": "../python_implementation/algorithms_implementation.py",
25       "undirected": true
26     },
27     "c++": {
28       "exe_path": "../c++_implementation/main",
29       "undirected": true
30     }
31   }
32 }

```

- **Algoritmo:** GLR ottimizzato per l'esecuzione su grafi non orientati;
- **Performance metric:** tempo di esecuzione;
- **Performance indicator:** tempo di CPU;
- **Factor:** numero di nodi del grafo (n) e numero di archi del grafo (m);
- **Noise parameter:** numero di community della partizione generata (C) e numero di nodi per partizione (n_c);
- **Design Point:**
 - $n=640$;
 - $m=(639,1278,2556,5112,10224,20448,40896,81792,163584)$.
- **Obbiettivo:** verificare come varia il tempo di esecuzione dell'algoritmo al variare del numero di archi del grafo su cui l'algoritmo viene eseguito.

C. Analisi della correttezza

Prima di eseguire qualsiasi altro esperimento, è stata eseguita un'analisi della correttezza delle implementazioni. Per il problema in analisi non è possibile calcolare la soluzione esatta, ma esistono solamente una serie di metriche che cercano, attraverso delle euristiche, di misurare la capacità di diffusione di ciascun nodo e quindi di determinare una soluzione più o meno accurata e in generale differente da metrica a metrica. Per questa ragione è stato complicato eseguire un test di correttezza e in particolare di validazione della soluzione e si è scelto di muoversi aumentando quanto possibile la confidenza sulla correttezza dei risultati ottenuti. A questo scopo si è scelto di eseguire le seguenti analisi:

- Si è eseguito l'algoritmo su istanze di input semplici per le quali la soluzione è intuibile;
- Si sono confrontate le soluzioni delle diverse implementazioni a fronte della stessa istanza di input;
- Si è eseguito un processo di verifica inserendo degli assert nel codice attivabili e disattivabili per evitare che potessero avere influenza sugli esperimenti successivi relativi al calcolo del tempo di esecuzione.

I risultati che verranno discussi nel seguito possono essere trovati nel notebook salvato nella repository del progetto "analysis/correcness.ipynb"

1) Esecuzione su istanze di input semplici : Si sono selezionate due tipologie di grafi:

- Un semplice cammino da nove nodi in quanto i nodi più vicini a tutti gli altri e quindi quelli con migliore capacità di diffusione sono i nodi centrali;
- Un grafo a stella composto da un anello centrale di otto nodi e otto nodi periferici ciascuno vicino ad un solo nodo appartenente all'anello. In questo tipo di grafo i nodi dell'anello sono tutti equidistanti a tutti gli altri nodi e quindi sono ugualmente influenti mentre i nodi periferici sono tutti equamente ininfluenti.

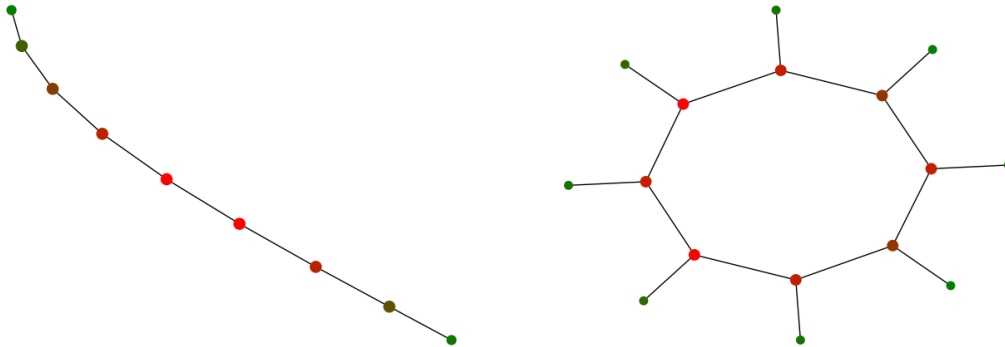


Fig. 11: Si sono utilizzati i colori per indicare il valore di centralità di ciascun nodo. Più il nodo è rosso più alto sarà il suo valore di centralità, viceversa più il nodo è verde più basso sarà il suo valore di centralità.

2) *Confronto tra le implementazioni* : Per verificare che tutte e quattro le implementazioni producano risultati coerenti in risposta alla stessa istanza di input, sono stati confrontati i risultati dell'Esperimento 2 (Exp. 2), ottenuti dalle implementazioni non ottimizzate di entrambi i linguaggi sui grafi salvati nella cartella "graphs/2expnode" della repository del progetto, con i risultati dell'Esperimento 5 (Exp. 5), ottenuti eseguendo le implementazioni ottimizzate sugli stessi grafi. Si osserva che i risultati contengono diversi nodi con lo stesso valore di centralità e che, confrontando i risultati ottenuti dalle diverse implementazioni dell'algoritmo, l'ordine relativo tra questi nodi non è sempre lo stesso. Questa non è una problematica infatti la differenza di ordinamento è presente solo tra nodi ugualmente influenti per i quali il principio di ordinamento è arbitrario e ininfluente ai fini della correttezza della soluzione. Per questa ragione per eseguire un confronto adeguato si è scelto di raggruppare i nodi sulla base del valore di centralità e di confrontare gli insiemi così ottenuti. Confrontando in questa maniera i risultati ottenuti dall'implementazione python ottimizzata con quelli dell'implementazione python non ottimizzata, si osserva che i risultati sono esattamente identici. Lo stesso vale se si confronta l'implementazione C++ ottimizzata con quelli dell'implementazione C++ non ottimizzata. Tuttavia, confrontando i risultati delle implementazioni python con quelli delle implementazioni C++, si riscontra una discrepanza in due grafi specifici. Per approfondire le cause di questa differenza, è stato condotto un processo di debug. Dal processo di debug si evince che la criticità è legata alla selezione dei nodi di gateway e, in particolare, nel caso in cui esista più di un nodo avente il massimo valore di inter community degree e che siano ugualmente distanti al nodo critico. La criticità è legata all'ordine con cui si itera sulla struttura dati dei due linguaggi. In ogni caso i nodi selezionati come gateway nei due casi, sono ugualmente buoni e di conseguenza, per quanto i risultati siano diversi, sono entrambi corretti.

```

Thread 1 "main" hit Breakpoint 3, algorithms_implementation::computeCommunityGateway (graph=0x55555554e10, communityGraph=0x55555580)
800, communityNodes::set with 565 elements = {...}, maxICL_node={...} at algorithms_implementation.cpp:190
90
(gdb) print(maxICL_node)
55 = 14532; maxICL_list = [[0] = 11505, [1] = 11917, [2] = 13119, [3] = 14057, [4] = 14532]
(gdb) continue
Continuing.

Thread 1 "main" hit Breakpoint 4, algorithms_implementation::computeCommunityGateway (graph=0x55555554e10, communityGraph=0x55555580)
800, communityNodes::set with 565 elements = {...}, maxICL_node={...} at algorithms_implementation.cpp:101
101
(gdb) print distance
55 = 0
(gdb) print minDistance
55 = 0
(gdb) print *node1
55 = (unsigned long A) @0x5555557f0780: 11505
(gdb) print node1
55 = 11505
(gdb) continue
Continuing.

Thread 1 "main" hit Breakpoint 4, algorithms_implementation::computeCommunityGateway (graph=0x55555554e10, communityGraph=0x55555580)
800, communityNodes::set with 565 elements = {...}, maxICL_node={...} at algorithms_implementation.cpp:101
101
(gdb) print distance
55 = 0
(gdb) print minDistance
55 = 0
(gdb) print node1
55 = 11917
(gdb) print maxICL_node
55 = 11505
(gdb)
(Pdb) print(max_ICL_nodes)
[14532, 13119, 11505, 11917, 14057]
(Pdb) continue
> /home/roberto/coding/AlgorithmEngineering/python_implementation/algorithms_implementation.py(65)compute_community_gateway()
-> if max_ICL_node == -1 or distance < min_distance:
(Pdb) print(max_ICL_node)
-1
(Pdb) print(distance)
17.0
(Pdb) print(min_distance)
0
(Pdb) print(node)
14532
(Pdb) continue
> /home/roberto/coding/AlgorithmEngineering/python_implementation/algorithms_implementation.py(65)compute_community_gateway()
-> if max_ICL_node == -1 or distance < min_distance:
(Pdb) print(max_ICL_node)
14532
(Pdb) print(distance)
5.0
(Pdb) print(min_distance)
17.0
(Pdb) print(node)
13119
(Pdb)

```

Fig. 12: A sinistra vediamo il debug su codice C++ mentre a destra quello su codice python. Ci troviamo nella sotto routine per calcolare il gateway di una delle community, all'interno della community esiste più di un nodo con lo stesso valore di inter-community degree per cui bisogna selezionare come gateway il nodo tra i papabili più vicino al nodo critico della community. Il nodo più vicino dista 5 ma esiste più di un nodo tra i papabili gateway che dista 5 dal nodo critico. Entrambi gli algoritmi selezionano come gateway il primo tra i papabili che dista 5 dal nodo critico, dato che le due implementazione testano i nodi in un ordine diverso verrà selezionato un diverso nodo di gateway pur essendo entrambi ugualmente buoni (l'implementazione C++ selezionerà il nodo 11917 come gateway, mentre l'implementazione python selezionerà il nodo 14532)

D. Analisi del tempo di esecuzione

In questa sezione verranno riassunti i risultati più importanti ottenuti dagli esperimenti svolti. Tutte le analisi sono state svolte considerando il tempo di esecuzione medio ottenuto a seguito della ripetizione multipla degli stessi esperimenti. Volendo visionare la totalità dei risultati li si può trovare nel notebook salvato nella repository del progetto "analysis/double_experiment.ipynb".

1) *Analisi dell'algoritmo non ottimizzato*: In questa sezione viene analizzato il tempo totale di esecuzione dell'algoritmo non ottimizzato per l'esecuzione su grafi non orientati

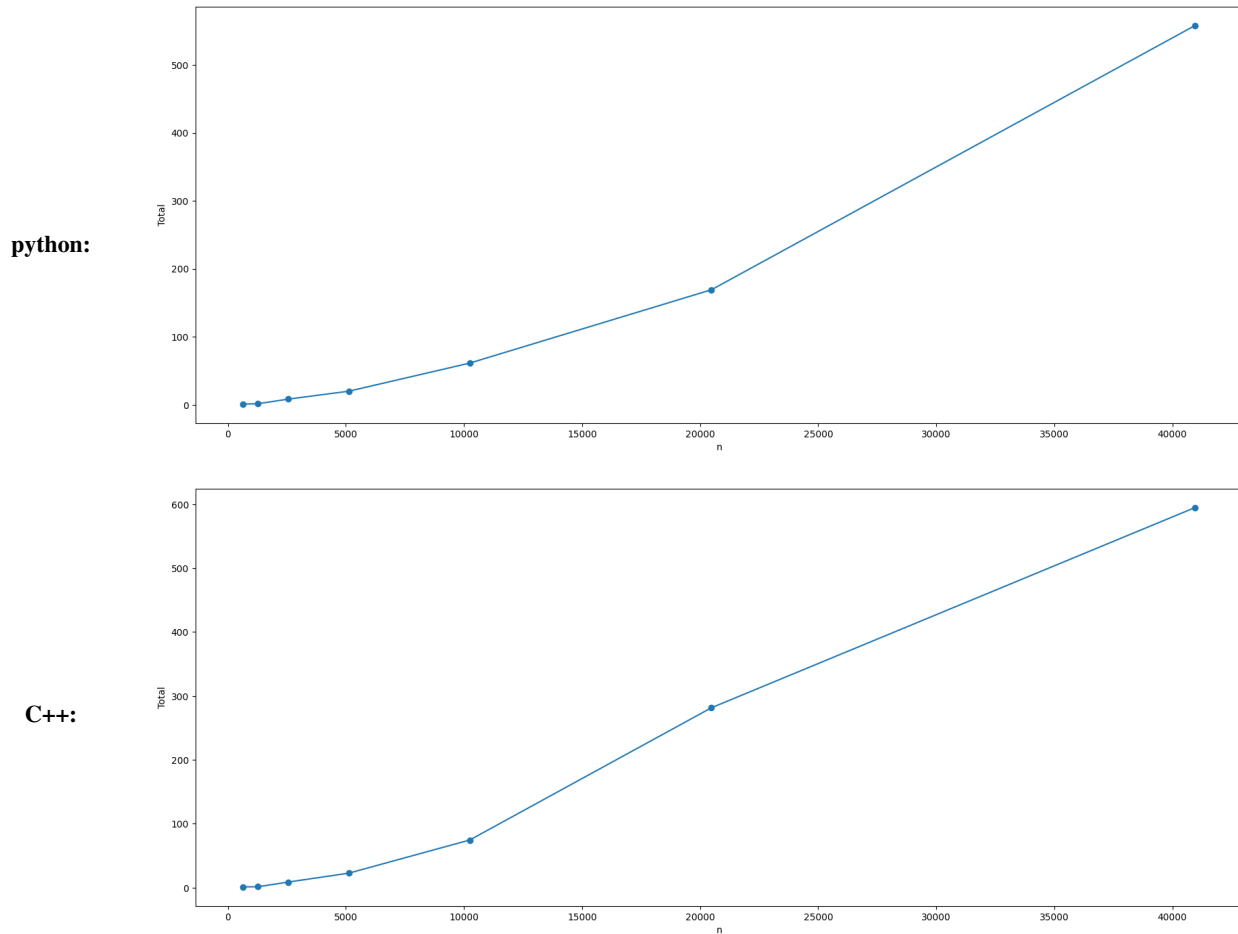


Fig. 13: Grafici rappresentanti i risultati relativi al tempo totale ottenuti dall'esecuzione dell'algoritmo non ottimizzato nell'esperimento 1 (Exp. 1)

Dall'esecuzione del primo esperimento (Exp. 1) si ottengono i seguenti risultati:

n	Total_py	Total_cpp
640	1.167380	1.064801
1280	1.669524	1.379706
2560	8.446346	8.709029
5120	20.054920	22.610006
10240	61.390214	74.451392
20480	169.258372	281.508535
40960	557.888292	594.784175

Si osserva che in entrambi i linguaggi al raddoppiare del numero di nodi, il tempo totale quasi sempre triplica. Stando all'analisi asintotica la dipendenza dai nodi dovrebbe essere quadratica quindi, il tempo totale dovrebbe quadruplicare al raddoppiare dei nodi. L'analisi asintotica considera il caso peggiore per cui si può concludere che l'analisi pratica conferma i risultati teorici.

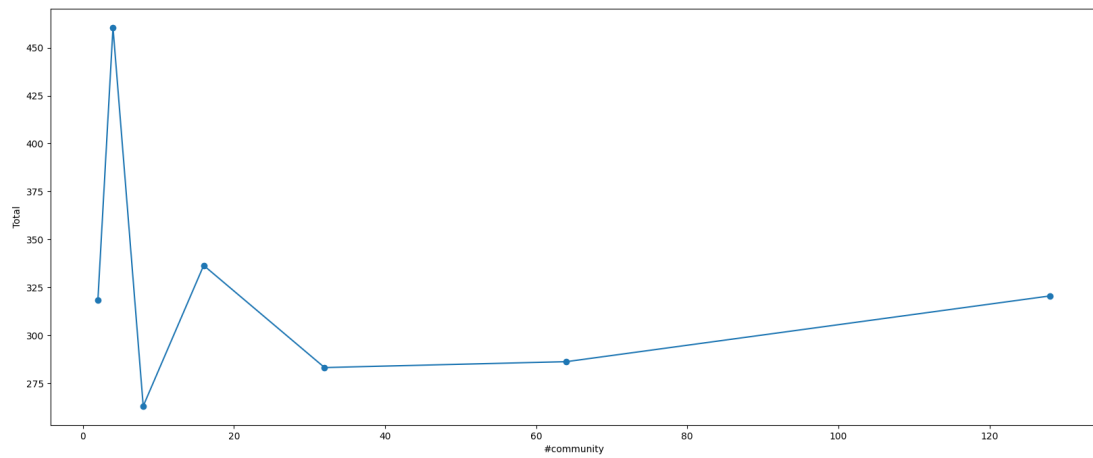
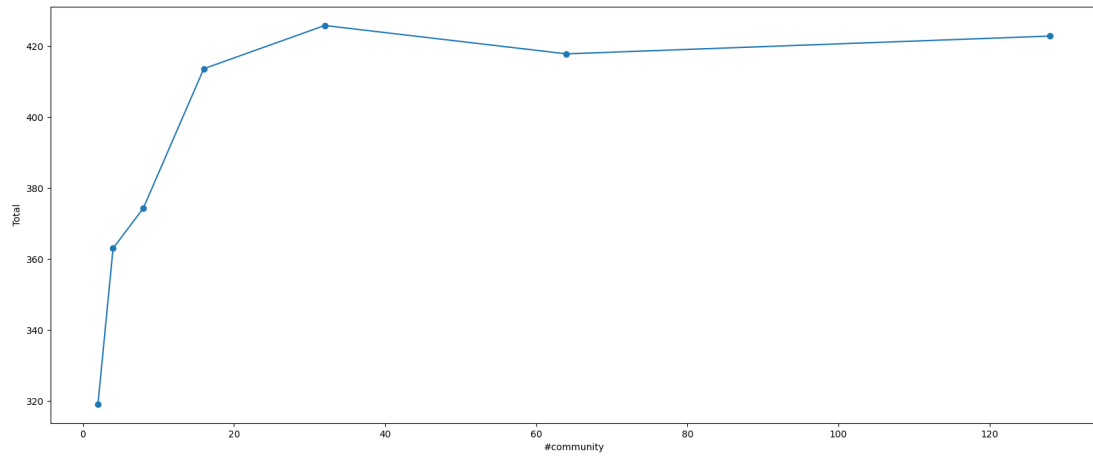
python:**C++:**

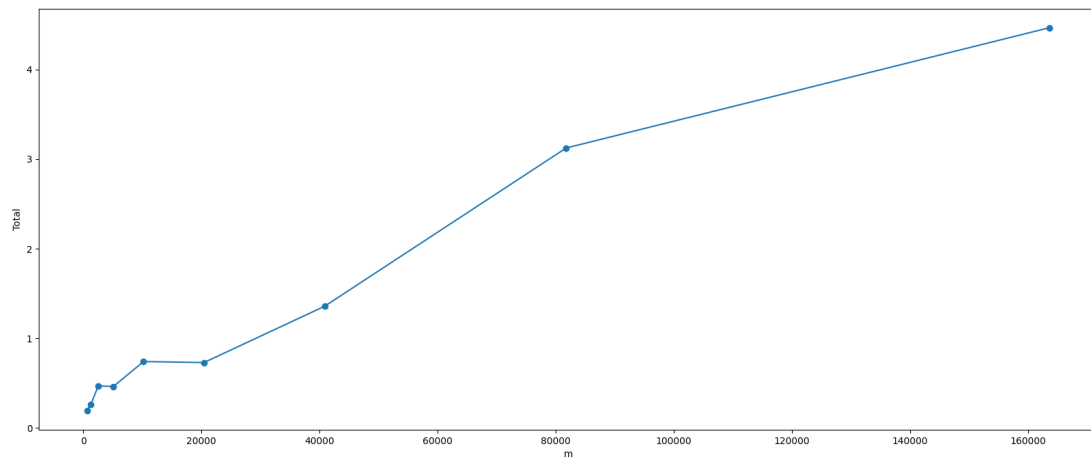
Fig. 14: Grafici rappresentanti i risultati relativi al tempo totale ottenuti dall'esecuzione dell'algoritmo non ottimizzato nell'esperimento 3 (Exp. 3)

Dall'esecuzione del terzo esperimento (Exp. 3) si ottengono i seguenti risultati:

C	Total.cpp	Total.py
2	319.132959	318.461162
4	363.030826	460.514745
8	374.343763	263.209204
16	413.659477	336.513530
32	425.853062	283.275303
64	417.855339	286.347214
128	422.874115	320.609069

analizzando i risultati sembra non esserci una dipendenza diretta del numero di community sul tempo di esecuzione. Questo conferma che con l'implementazione adottata la complessità temporale non corrisponde a quella stimata dall'articolo ma al contrario è coerente con l'analisi riportata nella sezione (Sez. I-C.2).

python:



C++:

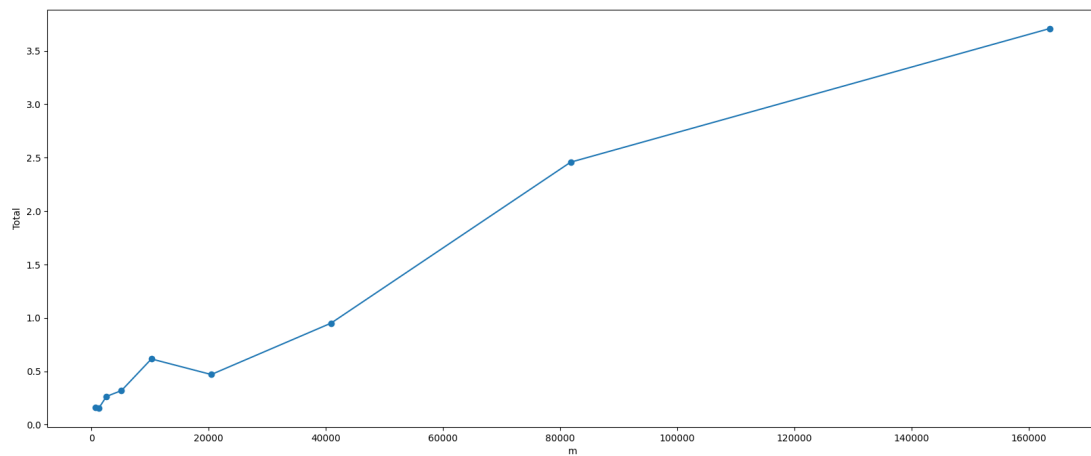


Fig. 15: Grafici rappresentanti i risultati relativi al tempo totale ottenuti dall'esecuzione dell'algoritmo non ottimizzato nell'esperimento 4 (Exp. 4)

Dall'esecuzione del quarto esperimento (Exp. 4) si ottengono i seguenti risultati:

m	Total_py	Total_cpp
639	0.193301	0.159482
1278	0.259008	0.156320
2556	0.467429	0.264430
5112	0.461079	0.318398
10224	0.739244	0.614976
20448	0.727850	0.469246
40896	1.356652	0.949992
81792	3.122732	2.456812
163584	4.464180	3.707897

si osserva che l'andamento non è ben definito ma comunque sempre crescente, anche se non sempre, il tempo di esecuzione spesso raddoppia al raddoppiare del numero di archi suggerendo una dipendenza pressochè lineare.

Volendo confrontare i tempi di esecuzione dell'implementazione python con quelli dell'implementazione C++ si osserva che sono molto simili e che spesso ma non sempre impiega meno l'implementazione python. Analizzando i tempi di esecuzioni delle subroutine si osserva che la più onerosa e anche l'unica per cui python impiega meno tempo rispetto a C++ è la subroutine GLR. La subroutine GLR è molto semplice e si limita quasi esclusivamente a chiamare la funzione "MultiTargetBFS" della libreria networkit. Si osserva che networkit, pur offrendo un'interfaccia python, è implementata in C++ e quindi non è così strano che si ottengano prestazioni simili nei due linguaggi.

Per analizzare la distribuzione del tempo speso, si è deciso di suddividere l'algoritmo in tre subroutine in accordo a quanto spiegato nella sezione (Sez. I-C).

Dall'esecuzione del primo esperimento (Exp. 1) si ottengono i seguenti risultati parziali:

Python Implementation				C++ Implementation			
n	Partition computation	Nodes Computation	GLR	n	Partition computation	Nodes Computation	GLR
640	0.022486	0.025503	1.119161	640	0.009149	0.018364	1.037249
1280	0.078723	0.154475	1.436054	1280	0.021063	0.041740	1.316830
2560	0.082281	0.350901	8.012729	2560	0.032369	0.074373	8.602079
5120	0.574619	0.733126	18.746002	5120	0.072951	0.233212	22.303442
10240	0.469772	2.464258	58.452894	10240	0.136679	0.409448	73.904358
20480	0.647963	6.029022	162.574754	20480	0.200575	1.114705	280.191227
40960	0.667566	19.946980	537.244402	40960	0.369819	3.565615	590.843554

Si osserva che per entrambe le implementazioni quasi la totalità del tempo viene speso nella subroutine GLR che di conseguenza segue gli stessi andamenti discussi per il tempo totale dell'algoritmo. Per quanto riguarda invece la subroutine Partition computation, guardando i dati ottenuti per l'implementazione C++ si osserva che al raddoppiare dei nodi il tempo speso per il calcolo della partizione raddoppia questo suggerisce una dipendenza lineare. Guardando infine alla subroutine Nodes Computation si osserva che al raddoppiare dei nodi il tempo speso sembra triplicare questo suggerisce un andamento quadratico, che infatti torna con l'analisi asintotica dal momento che raddoppiando i nodi del grafo in generale, raddoppino anche i nodi presenti in ciascuna delle community.

Dall'esecuzione del quarto esperimento (Exp. 4) si ottengono i seguenti risultati parziali:

Python Implementation				C++ Implementation			
m	Partition computation	Nodes Computation	GLR	m	Partition computation	Nodes Computation	GLR
639	0.020609	0.012587	0.159911	639	0.047481	0.003086	0.108838
1278	0.013093	0.020132	0.225616	1278	0.008339	0.007069	0.140848
2556	0.144080	0.037438	0.285733	2556	0.008083	0.032346	0.223941
5112	0.040933	0.017673	0.402268	5112	0.013738	0.005649	0.298955
10224	0.188965	0.039228	0.510872	10224	0.149101	0.012285	0.453536
20448	0.121965	0.038748	0.566876	20448	0.044407	0.015249	0.409539
40896	0.269335	0.039935	1.047210	40896	0.013636	0.024670	0.911632
81792	0.465223	0.112184	2.545160	81792	0.042813	0.030514	2.383431
163584	0.840655	0.105689	3.517631	163584	0.023799	0.025816	3.658205

Anche in questo caso la subroutine GLR è estremamente prevalente e segue quindi gli andamenti del tempo complessivo. Per quanto riguarda le altre subroutine invece non si riesce ad identificare una dipendenza precisa rispetto al numero di archi, il tempo speso per il calcolo della partizione rimane pressochè costante mentre il tempo speso per la subroutine Nodes computation cresce al crescere degli archi ma in un modo poco definito.

2) Analisi dell'algoritmo ottimizzato: In questa sezione viene analizzato il tempo totale di esecuzione dell'algoritmo ottimizzato per l'esecuzione su grafi non orientati

Prima di avviare qualsiasi discussione, è importante sottolineare che, a differenza di quanto accadeva nella versione non ottimizzata dell'algoritmo, nella versione ottimizzata nessuna subroutine, compresa la GLR, domina il calcolo della complessità temporale; tutte le subroutine contribuiscono in modo significativo. Le complessità delle tre subroutine, seguendo la stessa suddivisione dell'analisi precedente, risultano infatti essere:

- **Partition computation:** $O(n \cdot \log(n))$;
- **Nodes computation:** $O(C \cdot (n_c^2 \cdot \log(n_c) + n_c))$ dove n_c è il massimo numero di nodi presenti in una community;
- **GLR:** $O(C \cdot (n + m))$

La complessità dell'algoritmo scaturisce quindi dalla somma di queste tre componenti.

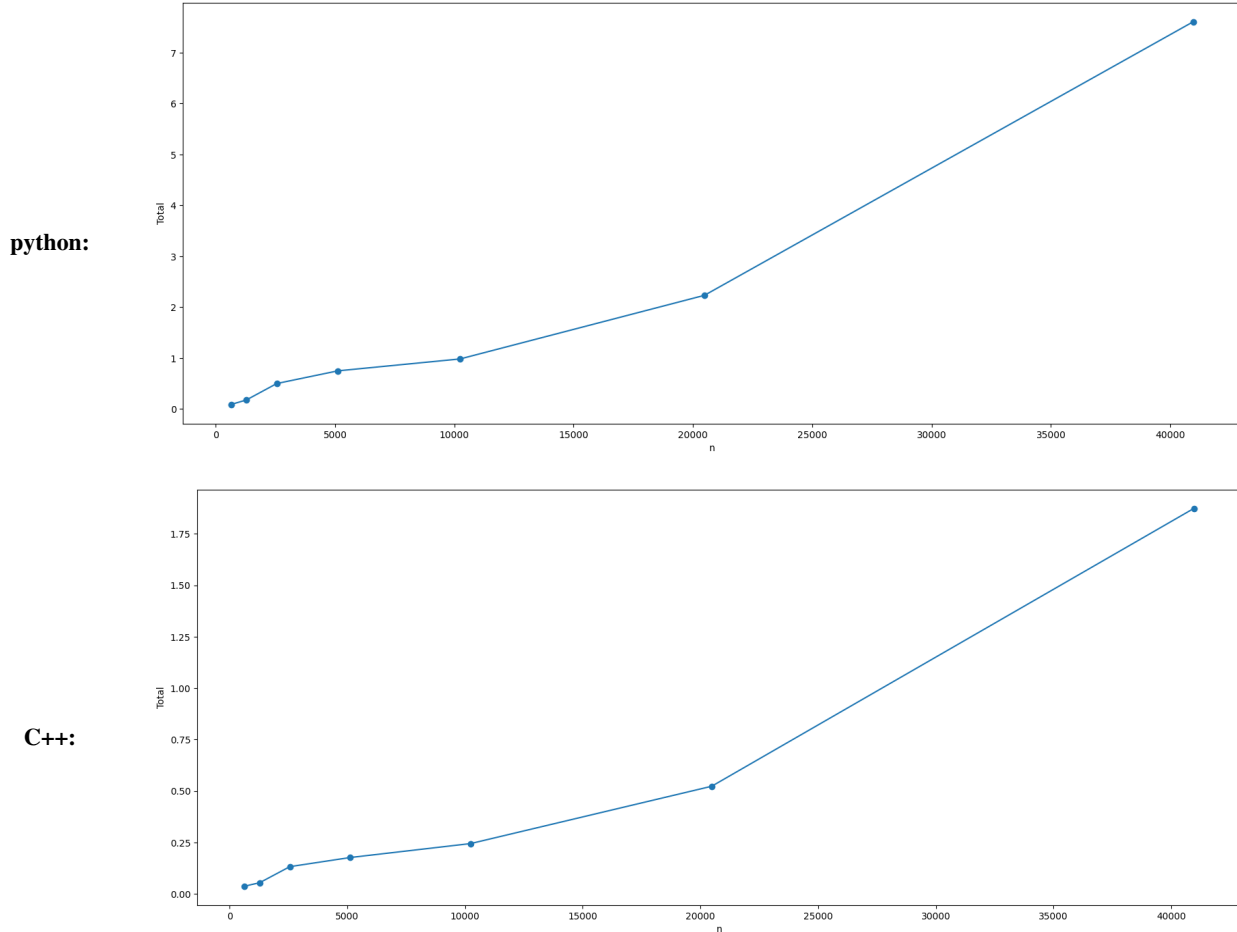


Fig. 16: Grafici rappresentanti i risultati relativi al tempo totale ottenuti dall'esecuzione dell'algoritmo ottimizzato nell'esperimento 5 (Exp. 5)

Dall'esecuzione del quinto esperimento (Exp. 5) si ottengono i seguenti risultati:

n	Total_py	Total_cpp
640	0.089964	0.037715
1280	0.177190	0.054657
2560	0.501007	0.132585
5120	0.750087	0.176896
10240	0.983712	0.244762
20480	2.232958	0.523230
40960	7.604349	1.873114

Si osserva che al raddoppiare dei nodi il tempo di esecuzione circa raddoppia (ad eccezione di quando si passa da 20480 nodi a 40960) per cui si può riscontrare una dipendenza lineare dal numero di nodi del grafo.

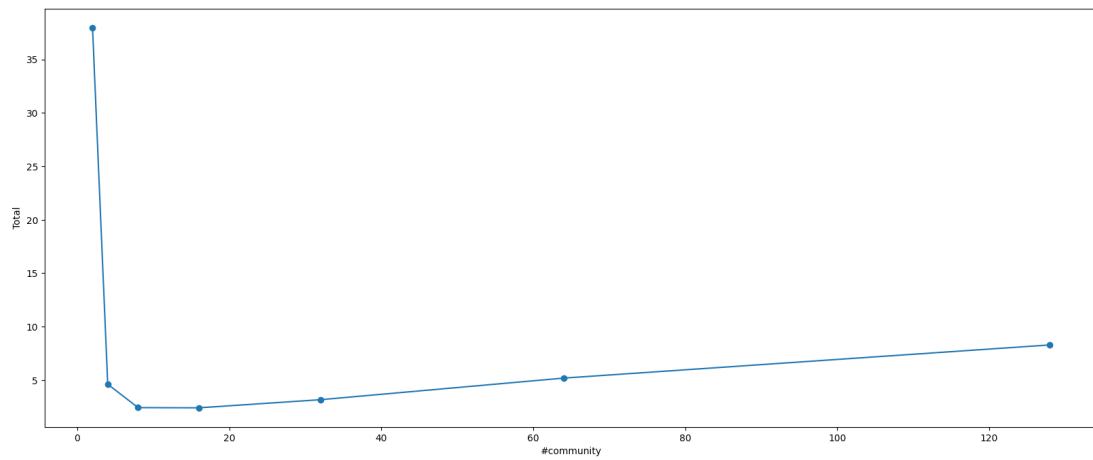
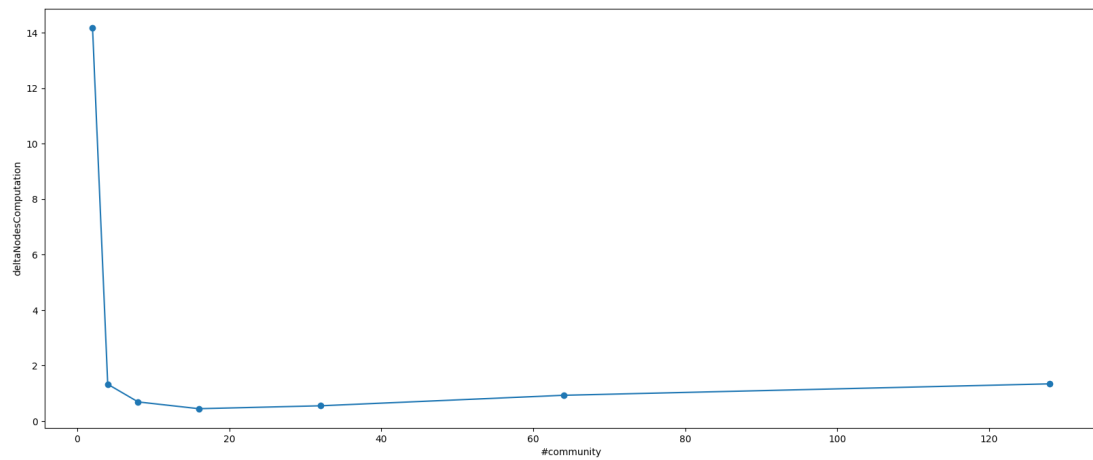
python:**C++:**

Fig. 17: Grafici rappresentanti i risultati relativi al tempo totale ottenuti dall'esecuzione dell'algoritmo ottimizzato nell'esperimento 6 (Exp. 6)

Dall'esecuzione del sesto esperimento (Exp. 6) si ottengono i seguenti risultati:

C	Total.py	Total.cpp
2	37.986110	14.212054
4	4.609648	1.348745
8	2.415110	0.727738
16	2.400550	0.506903
32	3.161341	0.651209
64	5.181738	1.082843
128	8.286098	1.629769

ad eccezione del picco iniziale il tempo di esecuzione sembra crescere linearmente con il numero di nodi, per quanto riguarda l'implementazione python, mentre sembra rimanere pressochè costante e addirittura diminuire nel caso dell'implementazione C++. Il picco iniziale potrebbe essere giustificato dal fatto che con sole due community, il numero di nodi di ciascuna potrebbe essere abbastanza grande da rendere significativo il tempo speso nella subroutine Nodes computation.

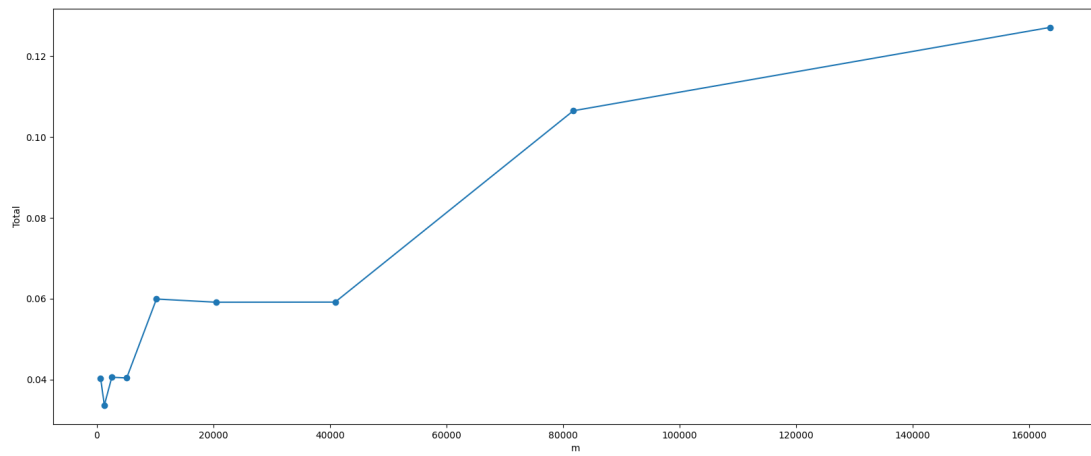
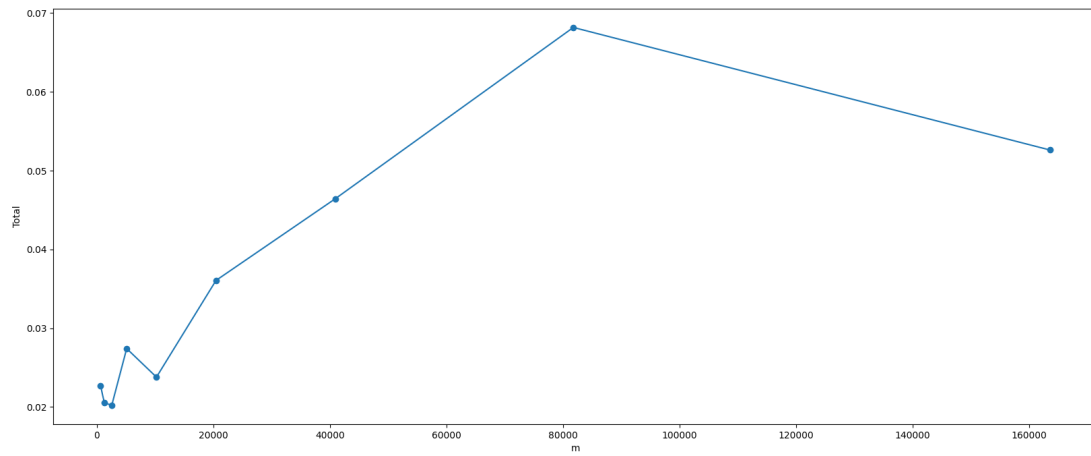
python:**C++:**

Fig. 18: Grafici rappresentanti i risultati relativi al tempo totale ottenuti dall'esecuzione dell'algoritmo ottimizzato nell'esperimento 7 (Exp. 7)

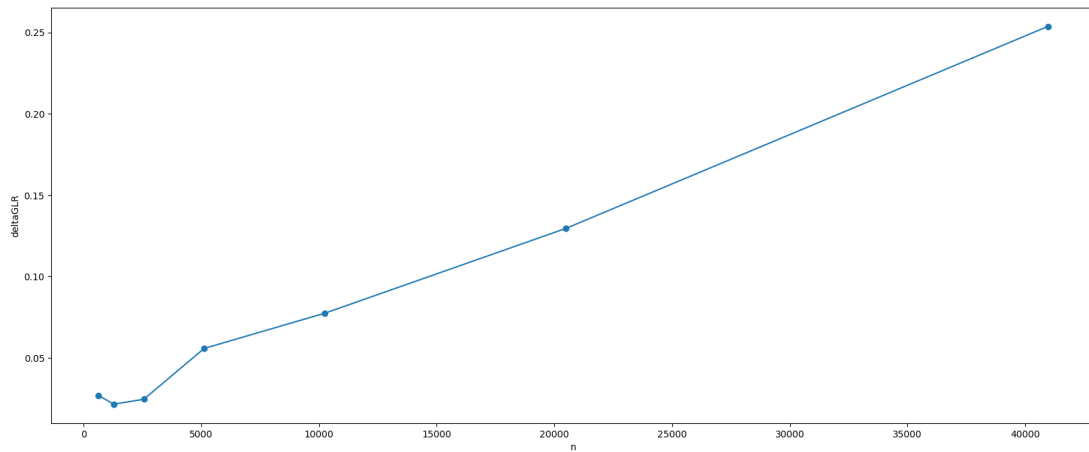
Dall' esecuzione del sesto esperimento (Exp. 7) si ottengono i seguenti risultati:

m	Total_py	Total_cpp
639	0.040213	0.022667
1278	0.033624	0.020512
2556	0.040539	0.020234
5112	0.040360	0.027413
10224	0.059883	0.023793
20448	0.059098	0.036056
40896	0.059127	0.046401
81792	0.106469	0.068138
163584	0.127077	0.052599

Il tempo di esecuzione rispetto al numero di archi sembra rimanere pressochè costante pur crescendo di poco.

Tra le subroutine, la più interessante da analizzare è la GLR, la cui modifica ha determinato una notevole riduzione del tempo di esecuzione complessivo.

python:



C++:

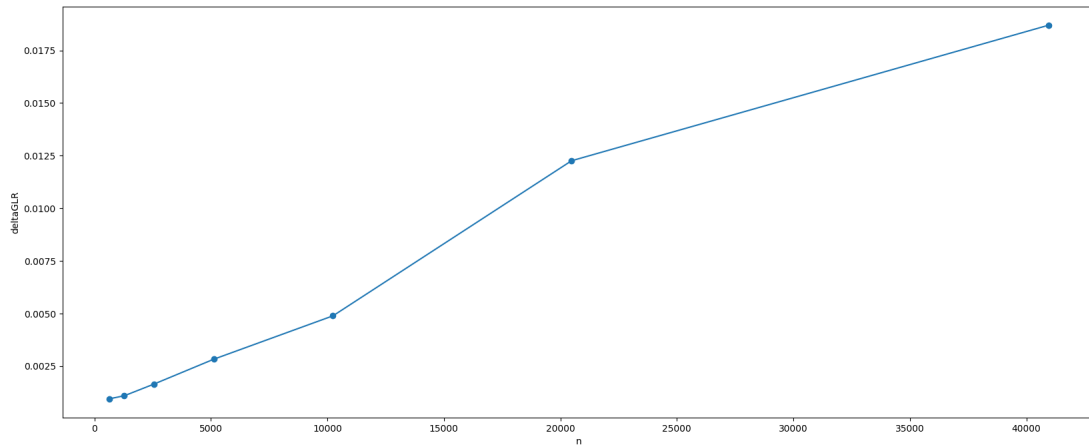


Fig. 19: Grafici rappresentanti i risultati relativi al tempo di esecuzione della subroutine GLR ottenuti dall'esecuzione dell'algoritmo ottimizzato nell'esperimento 5 (Exp. 5)

Dall'esecuzione del quinto esperimento (Exp. 5) si ottengono i seguenti risultati:

n	GLR_py	GLR_cpp
640	0.026825	0.000942
1280	0.021476	0.001084
2560	0.024503	0.001642
5120	0.055722	0.002821
10240	0.077366	0.004886
20480	0.129475	0.012257
40960	0.253683	0.018703

Si osserva che al raddoppiare dei nodi raddoppia anche il tempo di esecuzione questo è sintomo di una dipendenza lineare del tempo di esecuzione con il numero di nodi del grafo.

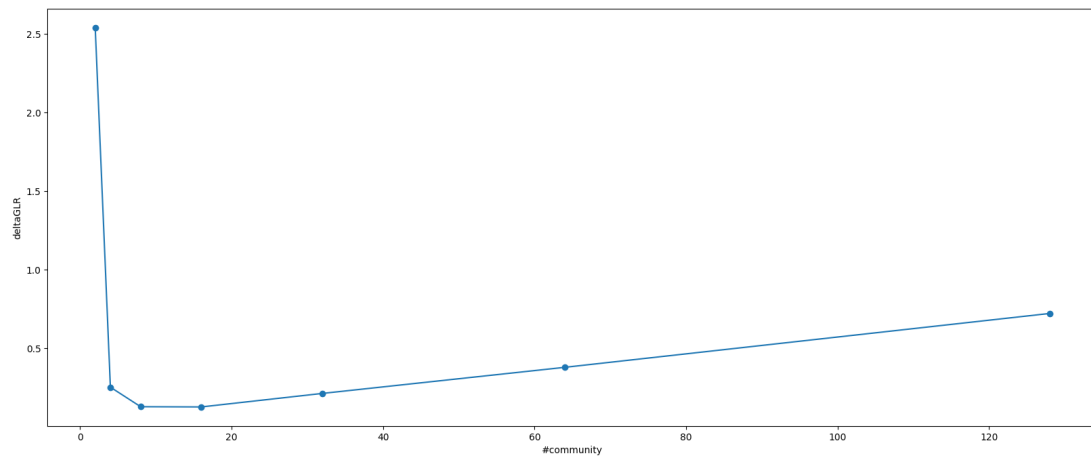
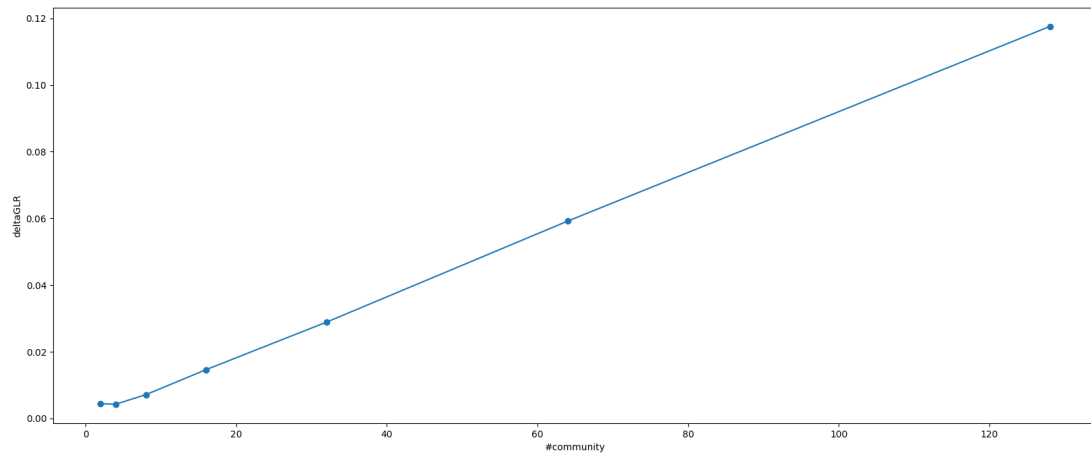
python:**C++:**

Fig. 20: Grafici rappresentanti i risultati relativi al tempo di esecuzione della subroutine GLR ottenuti dall'esecuzione dell'algoritmo ottimizzato nell'esperimento 6 (Exp. 6)

Dall'esecuzione del sesto esperimento (Exp. 6) si ottengono i seguenti risultati:

C	GLR_py	GLR_cpp
2	2.540200	0.004374
4	0.251745	0.004242
8	0.127229	0.007097
16	0.125894	0.014598
32	0.211995	0.028869
64	0.378064	0.059167
128	0.720723	0.117562

Analizzando i risultati si nota una dipendenza lineare rispetto al numero delle community della partizione come atteso dai risultati dell'analisi asintotica.

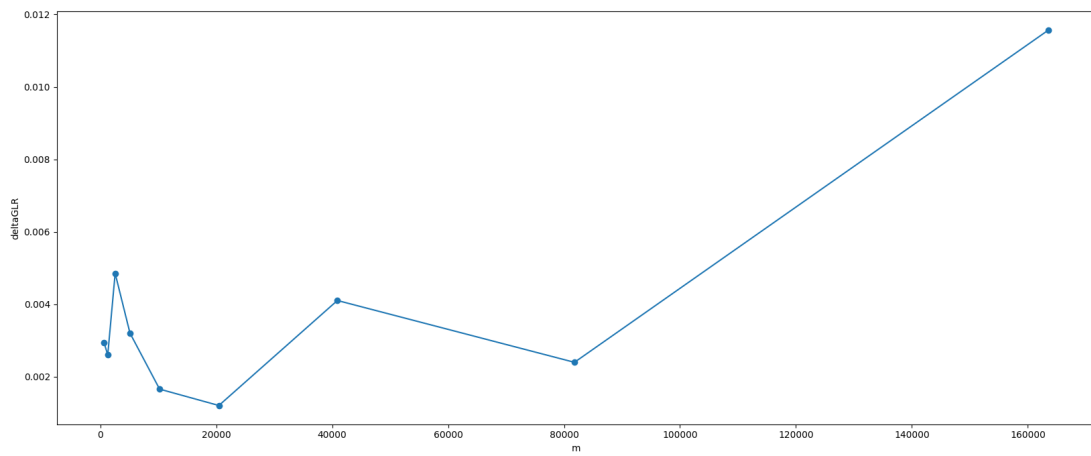
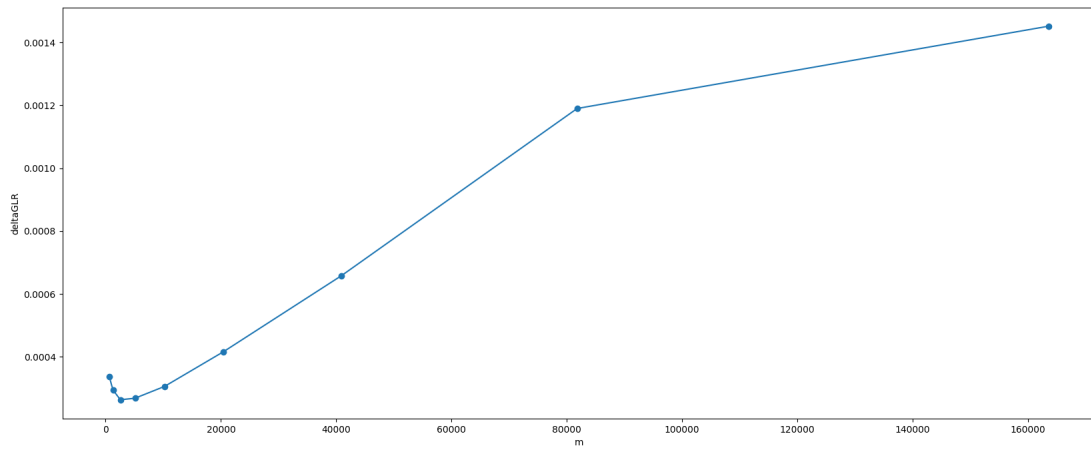
python:**C++:**

Fig. 21: Grafici rappresentanti i risultati relativi al tempo di esecuzione della subroutine GLR ottenuti dall'esecuzione dell'algoritmo ottimizzato nell'esperimento 7 (Exp. 7)

Dall'esecuzione del settimo esperimento (Exp. 7) si ottengono i seguenti risultati:

m	GLR.py	GLR.cpp
639	0.159911	0.108838
1278	0.225616	0.140848
2556	0.285733	0.223941
5112	0.402268	0.298955
10224	0.510872	0.453536
20448	0.566876	0.409539
40896	1.047210	0.911632
81792	2.545160	2.383431
163584	3.517631	3.658205

osservando i risultati, anche se l'andamento non è propriamente stabile, si può identificare, specialmente guardando all'andamento dell'implementazione C++, una dipendenza lineare tra il tempo di esecuzione e il numero di archi del grafo.

III. CONCLUSIONI

In questa relazione è stata verificata sperimentalmente come la complessità temporale dell'algoritmo per il calcolo della metrica Gateway Local Rank (GLR) dipenda dalle caratteristiche del grafo e della partizione utilizzati. Un altro importante risultato è stato aver ottimizzato l'implementazione dell'algoritmo ottenendo una variante che risolve lo stesso problema su grafi orientati con performance molto migliori. Tutte le implementazioni sono state realizzate sia in python che in C++ ciò ha permesso di confrontare le performance ottenute dai due linguaggi. Per quanto riguarda l'implementazione della versione non ottimizzata, python sembra sorprendentemente più efficiente, probabilmente perché utilizza la libreria Networkit, che è scritta in C++ ma offre un'interfaccia python. Tuttavia, nell'implementazione ottimizzata per grafi non orientati, il C++ mostra prestazioni superiori. Tutti i test riportati in questa relazione sono stati eseguiti su grafi non pesati, prossimi esperimenti potrebbero riguardare test su grafi pesati modificando quindi ulteriormente l'implementazione in modo da utilizzare Dijkstra invece della BFS. Un'altra analisi interessante potrebbe riguardare la generalizzazione della versione per grafi non orientati anche a grafi orientati, creando a runtime un grafo equivalente (G') avente gli archi orientati all'opposto rispetto al grafo originale (G) in questo modo la distanza tra i due nodi v e u calcolata sul grafo G' sarà uguale alla distanza tra i nodi u e v calcolata sul grafo G e quindi, per calcolare la distanza tra i nodi del grafo e i nodi di interesse (nodi critici e gateway) sul grafo G , si potrà calcolare sul grafo G' la distanza tra i nodi di interesse e tutti i nodi del grafo, ottenendo teoricamente lo stesso beneficio in termini temporali che abbiamo ottenuto per i grafi non orientati con l'aggiunta dell'onere di creare il grafo G' .

BIBLIOGRAFIA

- [1] Chavdar Dangalchev. “Residual closeness in networks”. In: *Physica A: Statistical Mechanics and its Applications* 365.2 (2006), pp. 556–564. ISSN: 0378-4371. DOI: <https://doi.org/10.1016/j.physa.2005.12.020>. URL: <https://www.sciencedirect.com/science/article/pii/S0378437105012768>.
- [2] Yong Deng, Yang Liu, and Deyun Zhou. “An improved genetic algorithm with initial population strategy for symmetric TSP”. In: *Mathematical problems in engineering* 2015.1 (2015), p. 212794.
- [3] Linton C. Freeman. “Centrality in social networks conceptual clarification”. In: *Social Networks* 1.3 (1978), pp. 215–239. ISSN: 0378-8733. DOI: [https://doi.org/10.1016/0378-8733\(78\)90021-7](https://doi.org/10.1016/0378-8733(78)90021-7). URL: <https://www.sciencedirect.com/science/article/pii/0378873378900217>.
- [4] Ren Jiadong et al. “Identifying influential nodes in weighted network based on evidence theory and local structure”. In: 11.05 (2015), p. 1765.
- [5] Maksim Kitsak et al. “Identification of influential spreaders in complex networks”. In: *Nature physics* 6.11 (2010), pp. 888–893.
- [6] Linyuan Lü et al. “Leaders in social networks, the delicious case”. In: *PloS one* 6.6 (2011), e21202.
- [7] Gert Sabidussi. “The centrality index of a graph”. In: *Psychometrika* 31 (1966). URL: <https://doi.org/10.1007/BF02289527>.
- [8] Chiman Salavati, Alireza Abdollahpouri, and Zhaleh Manbari. “Ranking nodes in complex networks based on local structure and improving closeness centrality”. In: *Neurocomputing* 336 (2019). Advances in Graph Algorithm and Applications, pp. 36–45. ISSN: 0925-2312. DOI: <https://doi.org/10.1016/j.neucom.2018.04.086>. URL: <https://www.sciencedirect.com/science/article/pii/S092523121831275X>.
- [9] Kuang Zhou et al. “Median evidential c-means algorithm and its application to community detection”. In: *Knowledge-Based Systems* 74 (2015), pp. 69–88. ISSN: 0950-7051. DOI: <https://doi.org/10.1016/j.knosys.2014.11.010>. URL: <https://www.sciencedirect.com/science/article/pii/S095070511400402X>.