Università degli Studi di Messina

Breadth-First Search con il paradigma Map/Reduce

Daryn Calangi - Mariachiara Trifirò

Università degli studi di Messina

Presentation Overview

Campi di applicazione

L'implementazione distribuita dell'algoritmo trattato, trova applicazione in vari ambiti: l'identificazione delle potenziali aree di guasto e la gestione dei nodi critici all'interno di reti, l'analisi dei *path length* all'interno dei Social Network.

Campi di applicazione

- Wireless Multi-hop Networks (WMhN);
- Computing Reachability Preserving Graph;
- Online Social Networks (OSN).

Descrizione del problema

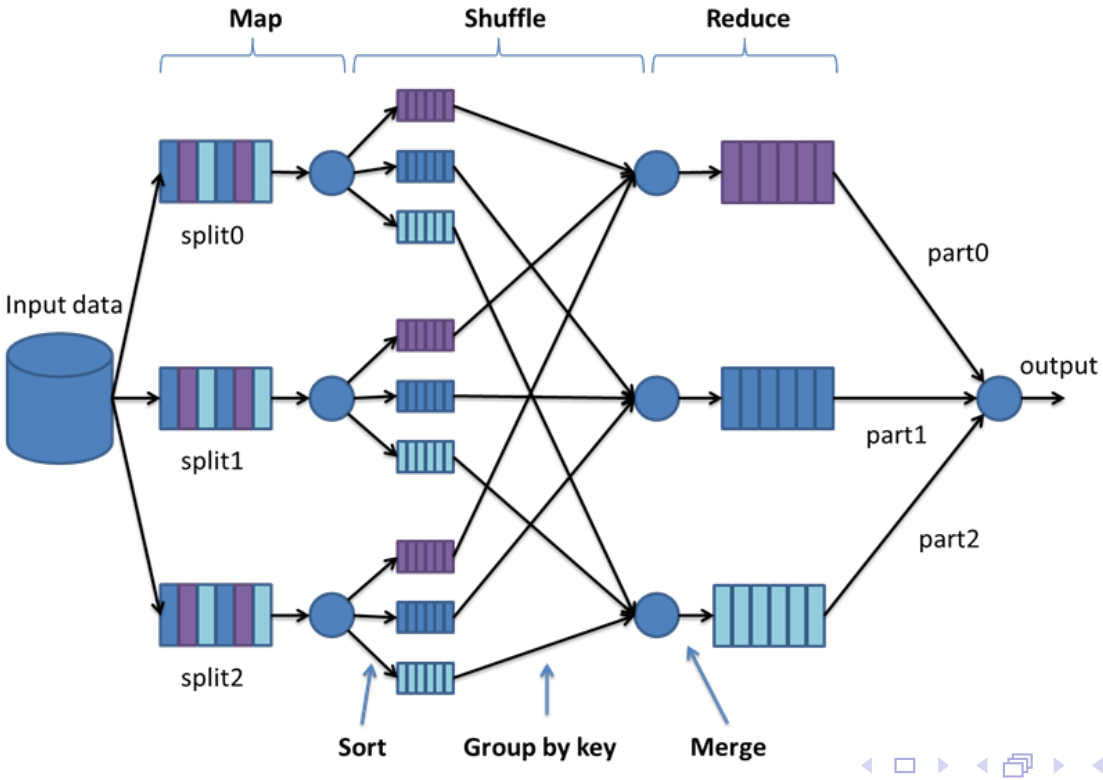
Il problema affrontato tratta l'implementazione dell'algoritmo **Breadth First Search** in ambiente distribuito sulla base del paradigma di programmazione **Map/Reduce** sfruttando il framework **Ray**.

BFS

La ricerca in ampiezza (**BFS**, *Breadth First Search*) è un algoritmo di ricerca non informato che esamina tutti i nodi di un grafo o di un albero in modo sistematico ed esaustivo.

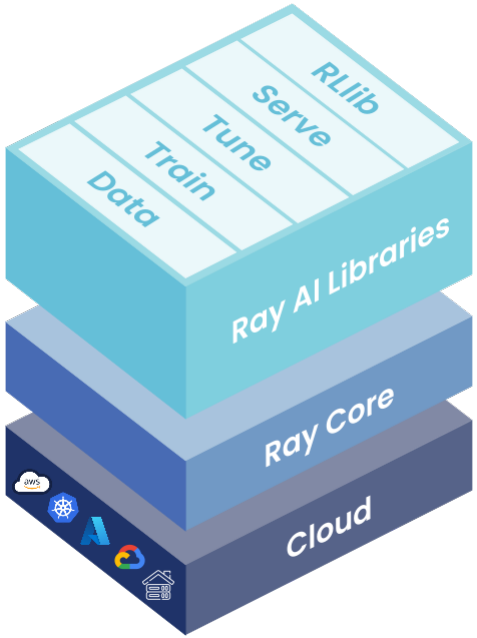
Map/Reduce

Il concetto su cui si basa il Map/Reduce consiste nel suddividere l'operazione di calcolo in più partizioni processate in modo autonomo. Una volta completata l'esecuzione di ciascuna porzione, i vari risultati parziali vengono *ridotti*, cioè ricomposti, ad un unico risultato finale.



Ray

Ray è un framework open-source progettato per supportare l'esecuzione distribuita di applicazioni che richiedono una potenza di calcolo non indifferente mantenendo alte le prestazioni.



Due implementazioni a confronto

L'obiettivo del progetto è analizzare due tipologie di implementazione:

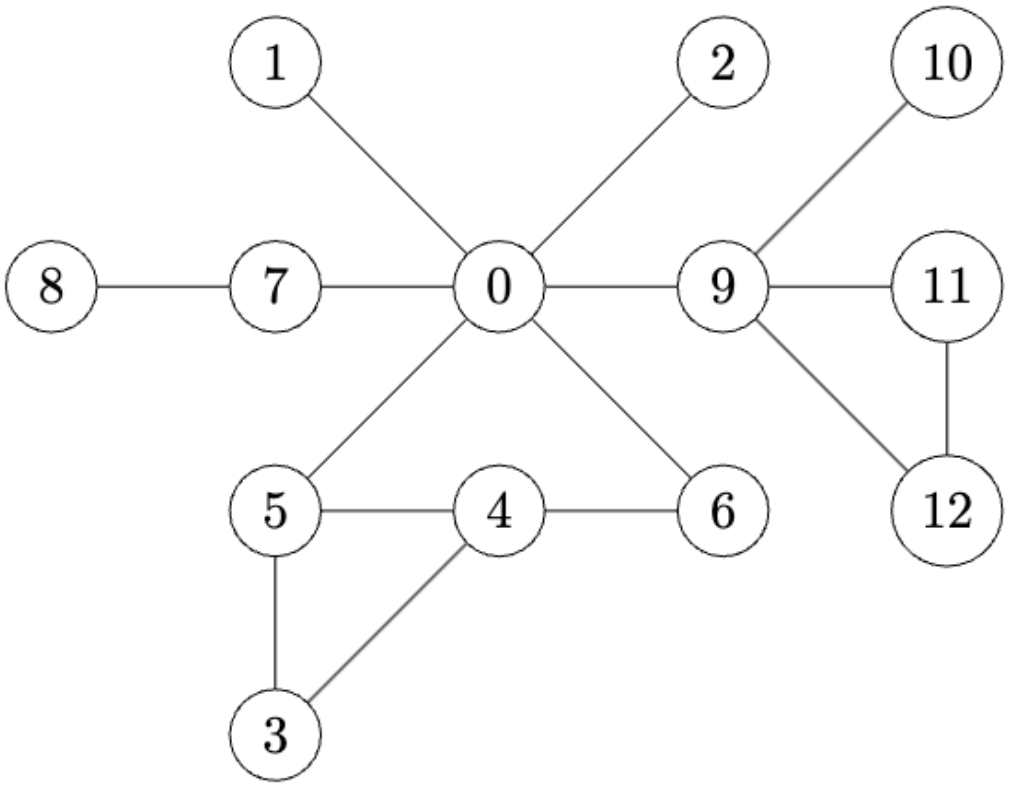
- BFS **Sequenziale**
- BFS **Parallelo**

Paradigma di programmazione

Entrambi i codici realizzati si basano su una logica orientata agli oggetti (**OOP**, *Object Oriented Programming*), utilizzando il linguaggio di programmazione Python. Sono state implementate due classi principali: **`Graph`** e **`Node`**.

Grafo

La classe **`Graph`** realizza il grafo fornendo in ingresso il percorso del file, quindi del dataset contenente i nodi e le connessioni tra questi.



Vertex	Neighbours
0	1, 2, 5, 6, 7, 9
1	0
2	0
3	4, 5
4	3, 5, 6
5	0, 3, 4
6	0, 4
7	0, 8
8	7
9	0, 10, 11, 12
10	9
11	9, 12
12	9, 11

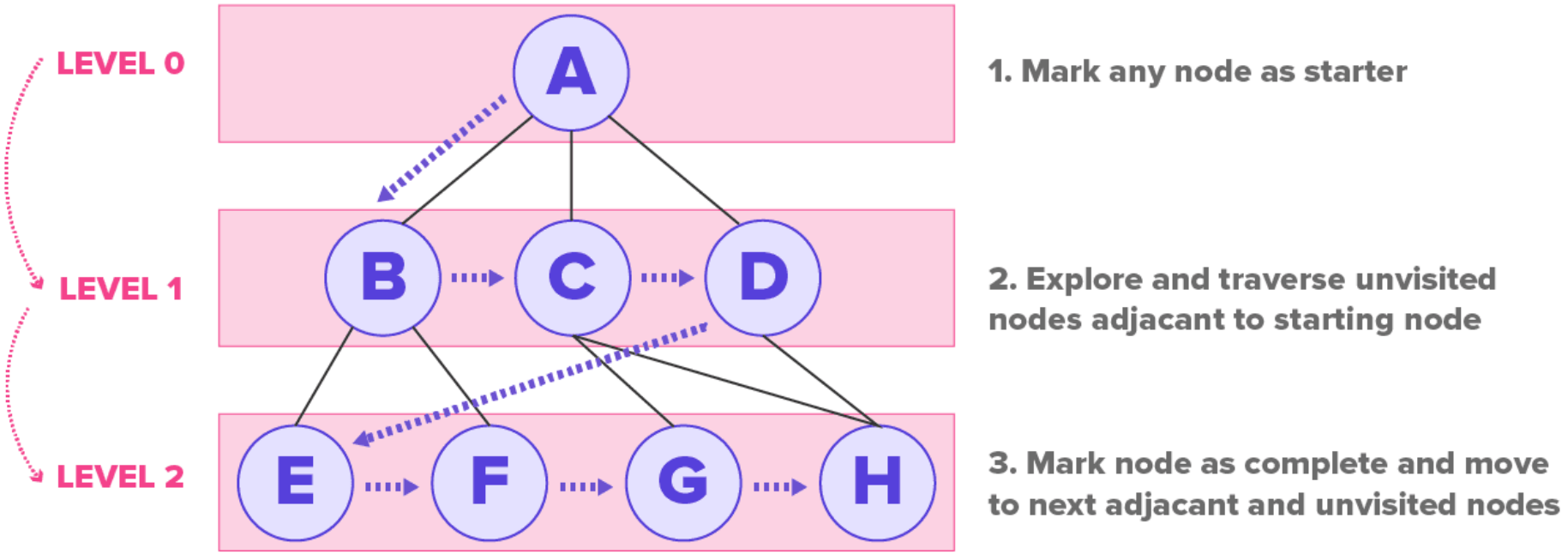
Nodo

La classe **`Node`** è caratterizzata dalle seguenti variabili di istanza:

- **`id`**
- **`source`**
- **`color`** che può essere
  1. WHITE
  2. GRAY
  3. BLACK
- **`dist`**
- **`neighbours`**
- **`path_list`**

BFS Sequenziale

In questa implementazione è stata mantenuta la logica classica dell'algortimo BFS che attraversa il grafo seguendo un ordine del tipo **FIFO** (*First In First Out*) come mostrato in figura.



BFS sequenziale: pseudocodice

```
1:  queue.put(source)
2:  while queue is not empty do
3:      parent ← queue.get()
4:      for each v ∈ L_u do                    ▷ L_u is the adjacency list of u
5:          if v.color == WHITE then
6:              v.color ← GRAY
7:              v.dist ← parent.dist + 1
8:              v.path_list.extend(parent.path_list)
9:              queue.put(v)
10:         end if
11:     end for
12:     parent.color ← BLACK
13: end while
```

BFS Parallelo

Come accennato in precedenza, nell'approccio distribuito è stata implementata una logica basata sul paradigma Map/Reduce suddividendo il grafo di partenza per darlo in pasto alle routine di **`map`** e **`reduce`** in grado di operare in parallelo grazie anche ad una fase di **`shuffling`** intermedia.

Schema Map/Reduce

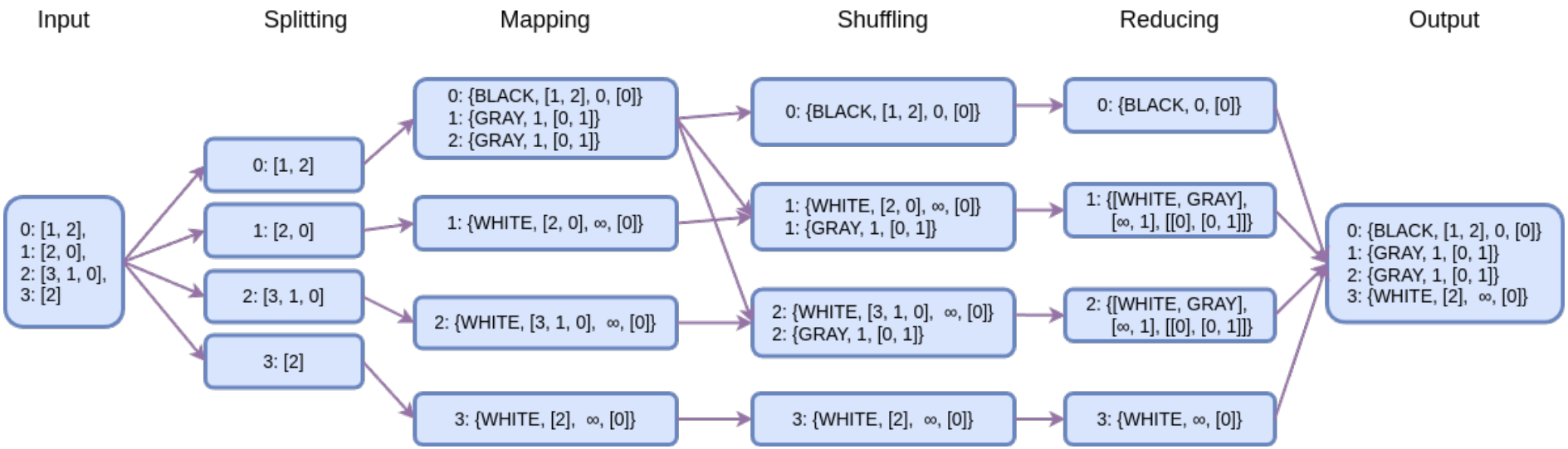


Figure: Schema Map/Reduce

Map

La fase di mapping consiste nell'emissione di un insieme di tuple con l'id del vertice come chiave e il vertice stesso come valore. Se il vertice risulta essere *GRAY*, i suoi vicini vengono aggiunti al set di risultati come nuovi vertici da esplorare con distanza incrementata e percorso modificato.

Map

```
1:  Input: key (ID of the current node); value (node information)
2:  Output: Node – a new node is emitted
3:
4:  u ← (Node)value;
5:  if u.color == GRAY then                          ▷ mark v as unexplored
6:      for each v ∈ L_u do                          ▷ L_u is the adjacency list of u
7:          v ← new Node();
8:          v.color ← GRAY;
9:          for each c ∈ P_u do                      ▷ P_u is the path list of u
10:             v.P_v.add(c);
11:         end for
12:         emit(v.ID, v);
13:     end for
14:     u.color ← BLACK;                             ▷ mark u as explored
15: end if
16: emit(key, u);
17: return ;
```

Shuffle

I risultati della fase di mapping vengono riorganizzati in un dizionario annidato.

Id nodo	Informazioni
0	{...}
1	{...}

→

Proprietà	Valore
colore	GRAY
distanza	0
vicini	[1, 2, 5]
cammino	[0]

Reduce

Quest'ultima routine acquisisce i risultati ottenuti dalla fase precedente e mantiene solo l'istanza del nodo che presenta la distanza minore, ovvero il cammino minimo trovato finora. Restituisce una lista di nuovi nodi aggiornati da fornire in ingresso alla funzione **`map`** nella prossima iterazione.

Reduce

```
1:  Input: values (nodes from shuffling phase)
2:  Output: nodes (list of updated nodes)
3:
4:  nodes ← []
5:  for each v ∈ values do
6:      newNode ← new Node();
7:      newNode.dist ← min(v.dist);
8:      index ← index(v.dist);
9:      newNode.color ← setToDarkest(v.color);
10:      newNode.path_list ← v.path_list[index];
11:      nodes.add(newNode);
12: end for
13: return nodes;
```

Esempio di esecuzione

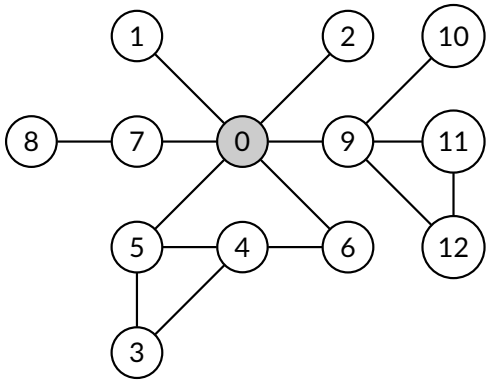


Figure: Iterazione 0

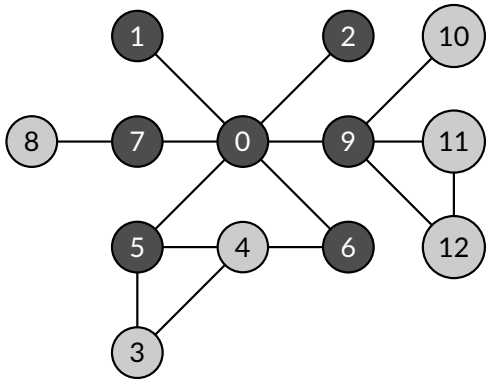


Figure: Iterazione 2

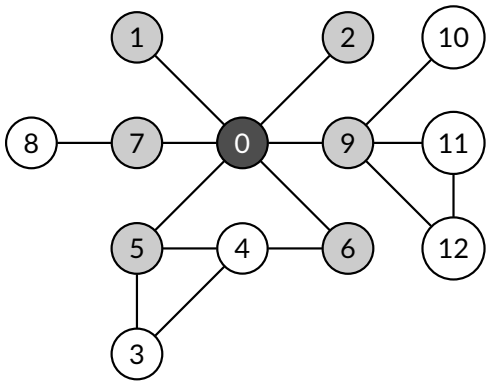


Figure: Iterazione 1

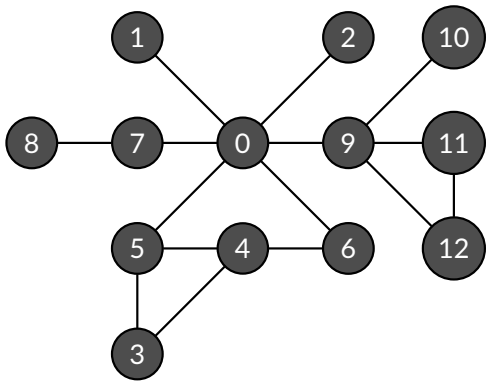


Figure: Iterazione 3

Dataset utilizzati

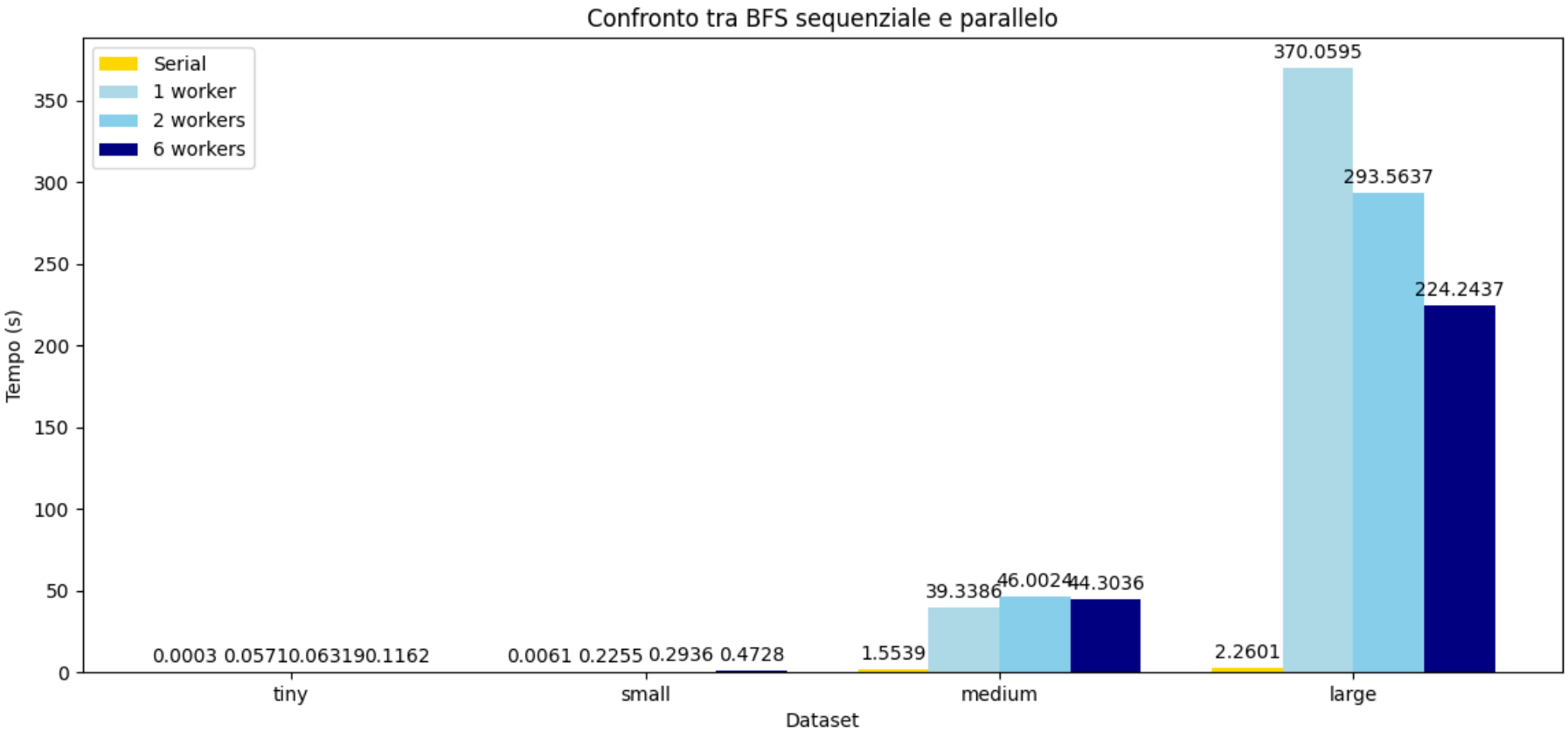
Dataset	Nodes	Edges
tiny	13	15
small	250	1273
medium	77.360	905.468
large	325.729	1.497.134

Table: Dimensioni dei dataset

Formato dei dataset utilizzati

FromNodeID	ToNodeID
0	5
4	3
0	1
9	12
6	4
5	4
0	2
11	12
9	10
0	6
7	8
9	11
5	3
0	7
0	9

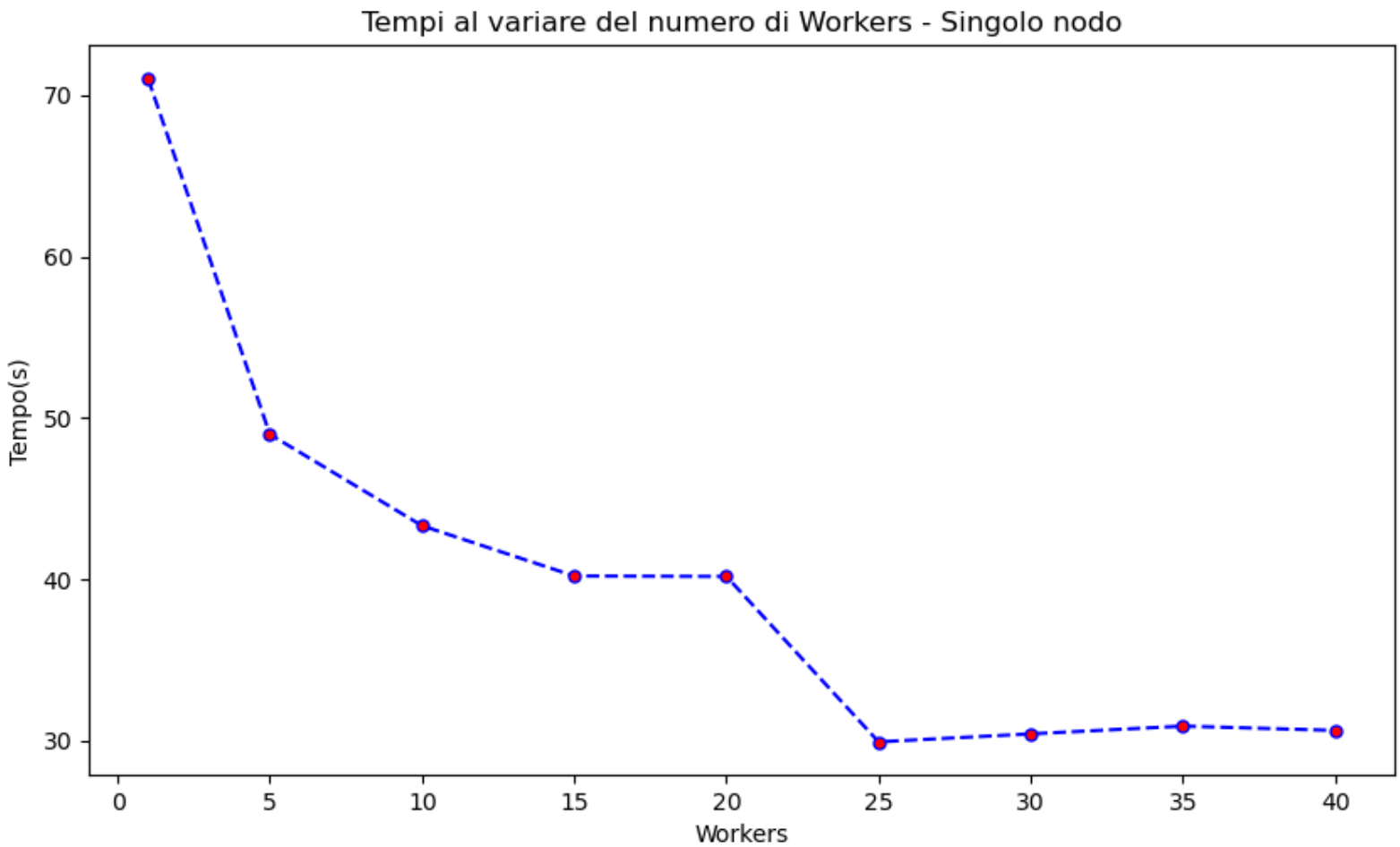
Risultati ottenuti



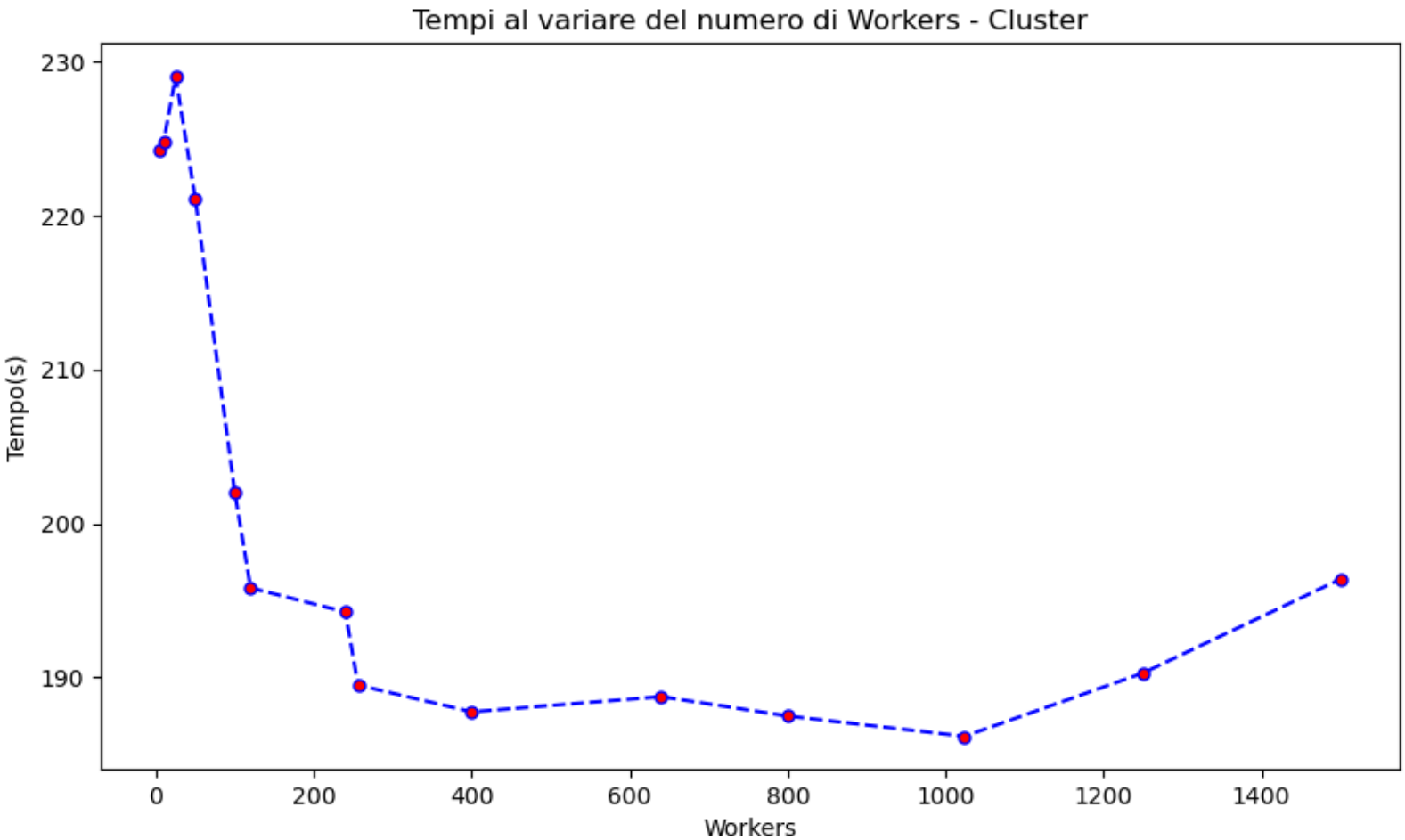
Risultati ottenuti

Dopo aver constatato la maggiore efficienza dell'implementazione sequenziale si è cercato di trovare la quantità di partizioni che ottimizza il più possibile la variante parallela ottenendo i seguenti risultati lanciando il codice rispettivamente in un cluster contenete un singolo nodo e in quello completo.

Risultati ottenuti



Risultati ottenuti



Conclusioni

Non è stato trovato il dataset che ha permesso di individuare il *cut off*, cioè il punto in cui le prestazioni dell'algoritmo BFS implementato in modo sequenziale divergono significativamente rispetto a quelle ottenute implementando l'algoritmo in modo distribuito.

Sviluppi futuri

Implementare l'algoritmo BFS in contesti *distribuiti* risulta essere sfidante a causa della comunicazione di rete, vengono proposte diverse soluzioni. Tra queste:

- Identificare e filtrare visite ai vertici non necessarie per ridurre le comunicazioni.
- Sfruttare architetture gerarchiche moderne per aggregare informazioni e eliminare visite inutili.

Istruzioni di avvio

Per eseguire il codice appena descritto, il comando da lanciare è del seguente tipo:

`BFS_map_reduce.py -p dataset.txt -w NUM_WORKERS`

https://github.com/Riachi02/BFS-MapReduce.git