Algoritmo di Kruskal vs Algoritmo di Prim

Laboratorio di Algoritmi - Esercizio E

Samuel Bruno

3 Dicembre 2022

Abstract

L'obiettivo di questa relazione è quello di analizzare le differenze tra l'algoritmo di Kruskal con struttura dati Union-Find e l'algoritmo di Prim con code di priorità.

1 Introduzione

Gli algoritmi di **Kruskal** e **Prim** sono algoritmi ottimi utilizzati per calcolare l'albero di copertura minimo (o anche MST, Minimum Spanning Tree) di un grafo non orientato. Sono algoritmi cosidetti "greedy", ovvero, algoritmi che effettuano tante scelte ottime locali sperando di ottenere una soluzione ottima globale. Si consideri un grafo non orientato e connesso dove V rappresenta il numero di nodi ed E il numero di archi. Ad ogni arco è associato un peso: lo scopo dei due algoritmi è quello di trovare un albero ricoprente di peso minimo, cioè quello in cui la somma dei pesi sia minima.

L'algoritmo di Kruskal, consiste in 3 passi:

i.) Mantiene una foresta, cioè un insieme di alberi: inizialmente ogni albero è un nodo isolato. Inoltre gli archi vengono ordinati in ordine crescente di costo;

ii.) Successivamente viene analizzato ogni arco singolarmente, inserendo nella soluzione un arco sicuro, ovvero, l'arco più leggero che non forma cicli con gli archi precedentemente selezionati (quindi archi presenti in alberi differenti) e lo aggiunge alla foresta. Il numero di alberi della foresta viene così decrementato di uno senza introdurre cicli;

iii.) Quando non ci sono più archi che connettono alberi differenti, l'algoritmo termina. La foresta contiene di conseguenza un unico albero che è anche l'albero di copertura minimo.

L'algoritmo di Prim, invece, lavora in questo modo:

i.) L'algoritmo parte da un albero che contiene, in quel momento, un unico nodo radice;

ii.) Ad ogni passo, l'algoritmo aggiunge, come arco sicuro, l'arco più leggero che connette un nodo dell'albero costruito con un nodo del grafo non presente nell'albero.

iii.) L'algoritmo termina quando non esistono più nodi del grafo non presenti nell'albero costruito. L'albero così ottenuto è un MST.

Il confronto verrà effettuato proprio sull'applicazione dei seguenti algoritmi. In particolare, verranno analizzati, attraverso un grafico che sintetizza il loro costo computazionale al variare del numero di nodi presenti nel grafo. Date le basi, possiamo procedere a descriverne la struttura dati.

2 Struttura dati

L'algoritmo di **Kruskal** viene solitamente rappresentato attraverso la struttura dati **Union-Find**. Questa è una struttura che mantiene una collezione $S = S1, S2, ..., Sk$ di **insiemi dinamici disgiunti**. Ciascun insieme è identificato da un **rappresentante** (che è un elemento dell'insieme) ed ogni elemento è rappresentato da un oggetto **x**.
Definiamo le operazioni dell'algoritmo:

i.) **Union(x,y):** Unisce due diversi sottoinsiemi in un unico sottoinsieme ed elimina i due sottoinsiemi dalla collezione S.

ii.) **Find(x):** Determina in quale sottoinsieme si trova l'elemento x e restituisce il rappresentante di quell' insieme.

iii.) **MakeSet(x):** Crea un nuovo insieme il cui unico elemento e rappresentante, è x.

Per quanto riguarda **Prim** la struttura adottata è quella dell'**Heap**, in particolare il **min-heap**. Un heap binario è una struttura dati composta da un array che può essere considerato come un albero binario quasi completo. Ad ogni nodo dell'albero corrisponde un elemento dell'array che memorizza il valore del nodo, mentre con albero quasi completo si intende che tutti i livelli, tranne l'ultimo, sono completi: possono mancare solo alcune foglie consecutive a partire dall'ultima foglia a destra. Per rappresentare un min-heap ogni valore del nodo deve soddisfare la seguente proprietà: ogni nodo i diverso dalla radice è tale che $A[parent[i]] \leq A[i]$. Supporta le seguenti operazioni principali:

i.) **Insert(A,x):** Inserisce l'elemento x in A.

ii.) **Minimum(A):** Restituisce l'elemento di A con chiave minima.

iii.) **Extract-Min(A):** Elimina e restituisce l'elemento di A con chiave minima.

iv.) **Decrease-Key(A, x, k):** Decrementa il valore della chiave di x ad un nuovo valore k (k deve essere $\leq$ dell'attuale valore della chiave di x).

v.) **Heapify(A, i):** Assume che i sottoalberi sinistro e destro del nodo i siano degli heap. Se questo si verifica, il metodo sposta l'elemento A[i] lungo un cammino dell'albero in modo da ristabilire la proprietà di heap.

3 Prestazioni attese

Il caso peggiore della complessità dell'algoritmo di **Kruskal** con implementazione Union-Find è $O(ElogV)$, poichè è necessario ordinare gli archi, mentre per quanto riguarda **Prim** dipende dall'implementazione della struttura dati ausiliaria utilizzata: in questo caso avendo utilizzato le code con priorità il tempo totale di esecuzione è **asintoticamente uguale** a Kruskal.

4 Documentazione del progetto

Nel programma ho voluto prendere in considerazione il confronto tra l'algoritmo di Kruskal, l'algoritmo di Prim con l'utilizzo della libreria **heapq** e sempre l'algoritmo di Prim ma questa volta con l'utilizzo della struttura dati **min-heap** implementata a mano. Il programma è formato in totale da 5 files:

i.) **graph.py:** Nel file sono presenti le classi Node, Edge e Graph contenenti l'implementazione del grafo.

ii.) **kruskal.py:** Nel file sono presenti la classe UnionFind contente l'implementazione della medesima struttura dati e la classe Kruskal dove si applica l'algoritmo.

iii.) **primHeapq.py:** Nel file è presente la classe Prim dove si applica l'algoritmo con l'utilizzo della libreria heapq per implementare le code a priorità.

iv.) **primMinHeap.py:** Nel file è presente la classe Prim dove si applica l'algoritmo implementando (senza l'utilizzo di librerie) la struttura dati min-heap.

v.) **main.py:** Nel file viene costruito il grafo e viene applicato il confronto tra i tre algoritmi.

Il progetto presenta quindi la seguente struttura:

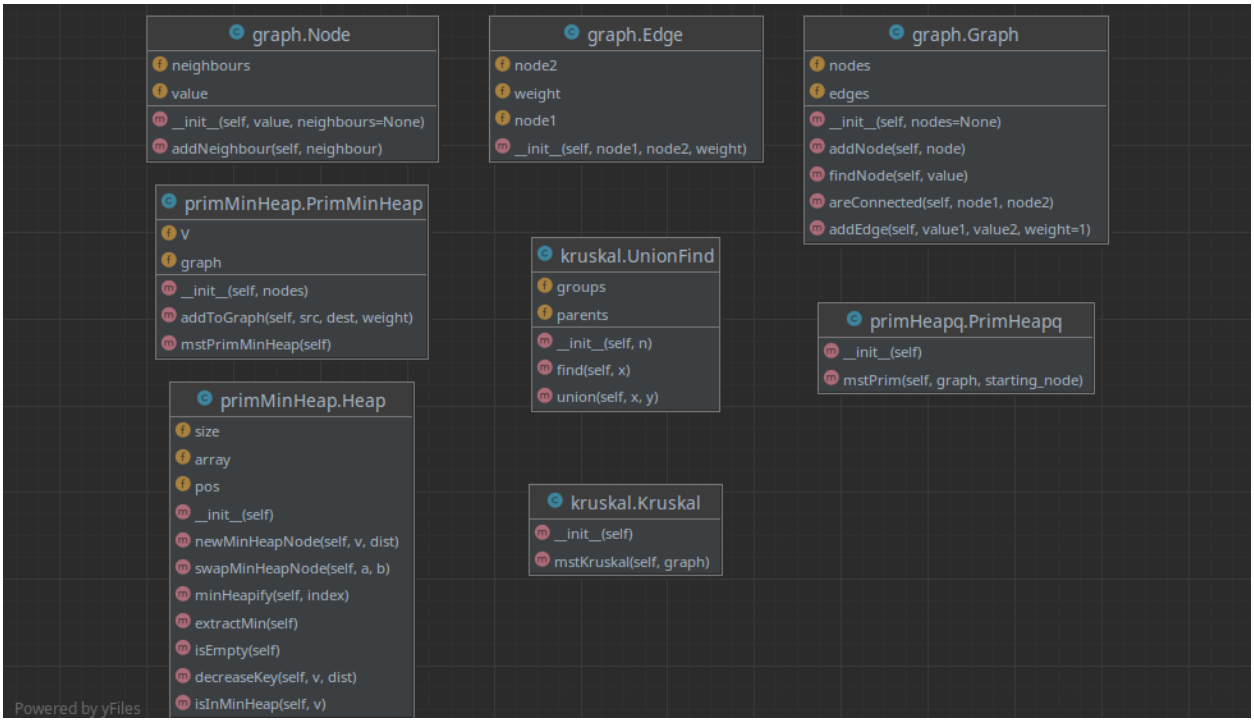

Figure 1: Diagramma delle classi

4.1 Classi e metodi

4.1.1 Classe Node

E' la classe che implementa il singolo nodo del grafo. I suoi attributi sono: value e neighbours (lista dei vicini di un nodo). Fornisce il metodo **addNeighbour** che aggiunge il vicino di un nodo.

4.1.2 Classe Edge

E' la classe che definisce l'arco. I suoi attributi sono: node1, node2 e weight (peso del nodo).

4.1.3 Classe Graph

E' la classe che implementa il grafo. Presenta l'attributo nodes, una lista contente i nodi. Fornisce i seguenti metodi:

- **addNode:** Inserisce un nuovo nodo nel grafo.
- **findNode:** Ricerca un nodo nel grafo.
- **areConnected:** Controlla se due nodi sono connessi tra di loro.
- **addEdge:** Inserisce un arco nel grafo.

4.1.4 Classe UnionFind

E' la classe che implementa la medesima struttura dati. Presenta gli attributi **parents** (un set che rappresenta un puntatore al padre di un nodo) e **groups** (rappresenta la dimensione della collezione, in particolare l'i-esimo elemento di *parents* rappresenta il padre dell'i-esimo nodo). Fornisce i metodi *union* e *find* il cui approfondimento è già stato descritto nella *sezione 2*.

4.1.5 Classe Kruskal

E' la classe che implementa l'algoritmo di Kruskal attraverso il metodo chiamato **mstKruskal**.

4.1.6 Classe PrimHeapq

E' la classe che implementa l'algoritmo di Prim attraverso l'utilizzo della libreria heapq.

4.1.7 Classe Heap

E' la classe che implementa la relativa struttura dati per implementare l'algoritmo di Prim.

4.1.8 Classe PrimMinHeap

E' la classe che implementa l'algoritmo di Prim utilizzando la struttura dati dell' Heap.

4.1.9 Classe Main

In questa classe avviene la costruzione del grafo ed il confronto tra i tre algoritmi. Vedremo nella prossima sezione i risultati di tali esperimenti.

5 Risultati Sperimentali

Gli esperimenti sono stati effettuati su una sequenza crescente di nodi che ad ogni ciclo vengono generati ed inseriti randomicamente in un grafo (partendo da un minimo di 3 nodi). I nodi sono memorizzati in un array contententi ogni lettera dell'alfabeto, in totale quindi 26. I test sono stati eseguiti su un HP Pavilion DV6 le cui specifiche sono:
Processore Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz 3.10 GHz Dual-core Quad-thread
RAM 8 GB 1333 Mhz
Sistema Operativo EndeavourOS - kernel Linux

Le misurazioni effettuate riguardano il tempo di esecuzione degli algoritmi nella CPU e il numero di nodi del grafo in quel determinato ciclo. I vari test sono stati eseguiti molteplici volte in modo tale da verificare la costanza dei risultati.
Nelle tabelle di seguito gli ultimi 3 test eseguiti (con l'asse delle ordinate contente i tempi espressi in **secondi**):

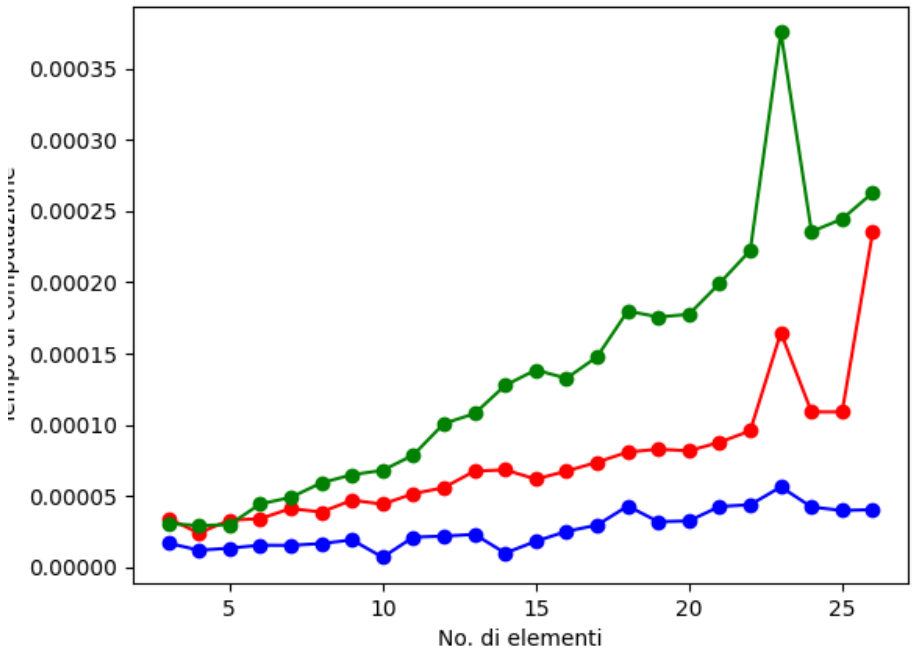

Figure 2: In rosso Kruskal ed in blu Prim

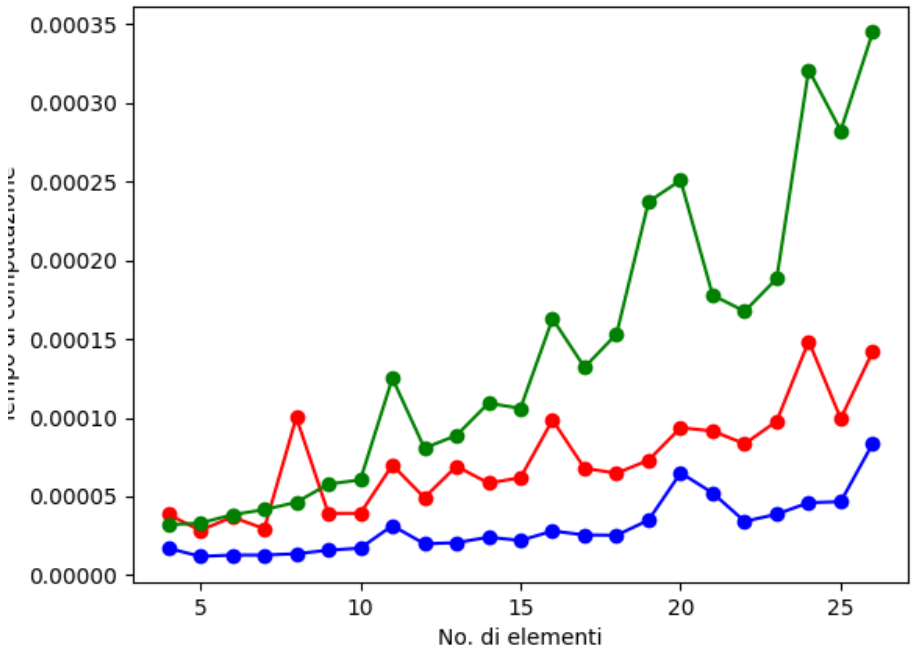

Figure 3: In rosso Kruskal ed in blu Prim

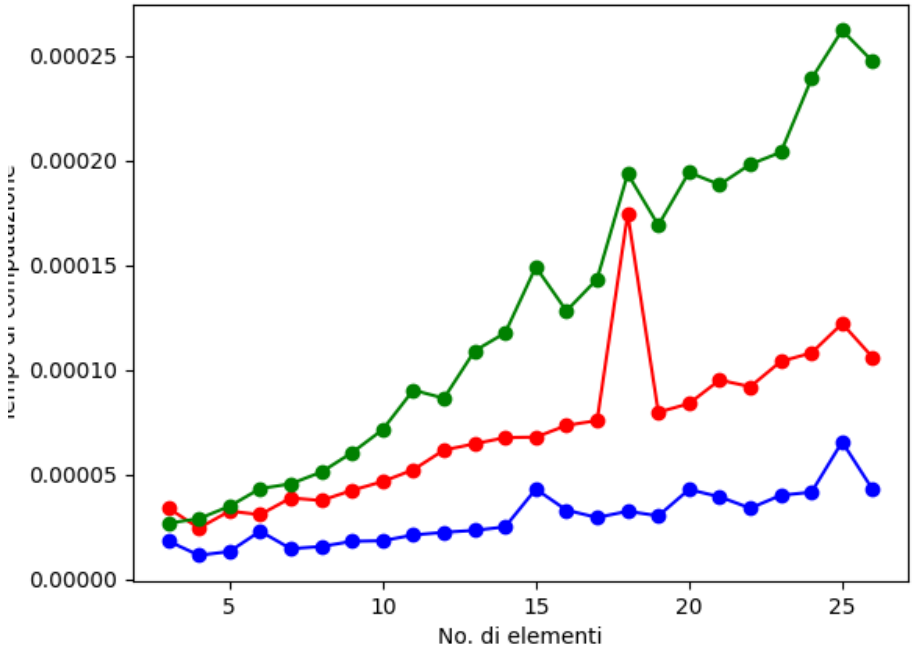

Figure 4: In rosso Kruskal ed in blu Prim

5.1 Analisi dei risultati

Come possiamo vedere nei grafici in figura i risultati rimangono coerenti dopo ogni test, con tutti gli algoritmi che seguono un andamento logaritmico del tipo $y = f(xlog(n))$ che quindi confermano le prestazioni attese nella *sezione 3*. Come dimostrato dai grafici, si può quindi evincere che **Prim con heapq** sia più **efficiente** di **Prim con min-heap** (data anche la probabile non ottima implementazione del codice) ma anche di Kruskal ed in particolare (*Figura 2*) è possibile notare quanto quest'ultimo cresca più **velocemente** rispetto a Prim che invece tende ad essere più **costante** e crescere più **lentamente** nel tempo. E' interessante notare quanto la libreria *heapq* sia molto ben ottimizzata, rispetto invece all'implementazione a codice del *min-heap*.

6 Conclusione

In conclusione dagli esperimenti effettuati notiamo come nella totalità dei risultati **Prim con heapq** abbia la meglio tra i due algoritmi, questo ci farebbe pensare che quindi sia più conveniente utilizzarlo. Se di **pro**, l'algoritmo di Prim, ha **l'efficienza** dalla sua, non si può dire lo stesso sulla **difficoltà di implementazione**. Per questo **Kruskal** risulta alla fine dei conti essere più **efficace**, sia in termini di risultati, che essendo espressi in unità di tempo particolarmente piccole si riconosce essere ugualmente molto efficiente, sia in termini di facilità di implementazione dell'algoritmo. Solitamente, quello che si usa fare è utilizzare Kruskal quando il grafo è **rado**, cioè con un numero ridotto di archi, del tipo $E = O(V)$, quando gli archi sono già ordinati o se possiamo ordinarli in tempo lineare. Invece, utilizzare Prim quando il grafo è **denso**, cioè con un numero elevato di archi, come $E = O(V^2)$.
Per riassumere: l'algoritmo di Prim è significativamente più veloce al limite quando si ha un grafo molto denso con molti più archi che vertici, Kruskal, invece, si comporta meglio in situazioni più tipiche perché utilizza strutture dati più semplici.