Esercizio 4: libreria per struttura dati grafo e algoritmo di *Kruskal*

Testo dell'esercizio

Si implementi una libreria che realizza la struttura dati **Grafo** in modo che **sia ottimale per dati sparsi**. La struttura deve consentire di rappresentare sia grafi diretti che grafi non diretti.

L'implementazione deve essere generica sia per quanto riguarda il tipo dei nodi, sia per quanto riguarda le etichette degli archi.

La struttura dati implementata dovrà offrire (almeno) le seguenti operazioni:

- Creazione di un grafo vuoto – O(1)
- Aggiunta di un nodo – O(1)
- Aggiunta di un arco – O(1)
- Verifica se il grafo è diretto – O(1)
- Verifica se il grafo contiene un dato nodo – O(1)
- Verifica se il grafo contiene un dato arco – O(1) (*)
- Cancellazione di un nodo – O(n)
- Cancellazione di un arco – O(1) (*)
- Determinazione del numero di nodi – O(1)
- Determinazione del numero di archi – O(n)
- Recupero dei nodi del grafo – O(n)
- Recupero degli archi del grafo – O(n)
- Recupero nodi adiacenti di un dato nodo – O(1) (*)
- Recupero etichetta associata a una coppia di nodi – O(1) (*)

(*) quando il grafo è veramente sparso, assumendo che l'operazione venga effettuata su un nodo la cui lista di adiacenza ha una lunghezza in O(1).

Si implementi l'**algoritmo di Kruskal** per la determinazione della minima foresta ricoprente di un grafo.

L'implementazione dell'algoritmo di Kruskal dovrà utilizzare la struttura dati **Union-Find Set** implementata nell'esercizio precedente.

La struttura dati Grafo e l'algoritmo di Kruskal dovranno essere utilizzati con i dati contenuti nel file italian_dist_graph.csv.

Ogni record contiene i seguenti dati:

- località 1: (tipo stringa) nome della località "sorgente". La stringa può contenere spazi, non può contenere virgole;
- località 2: (tipo stringa) nome della località "destinazione". La stringa può contenere spazi, non può contenere virgole;
- distanza: (tipo float) distanza in metri tra le due località.

Un'implementazione corretta dell'algoritmo di Kruskal, eseguita sui dati contenuti nel file italian_dist_graph.csv, *dovrebbe determinare una minima foresta ricoprente con 18.640 nodi, 18.637 archi (non orientati) e di peso complessivo di circa 89.939,913 Km.*

Libreria della struttura dati ed **algoritmo** utilizzato

L'esercizio è diviso in 3 parti:

- **Libreria per la gestione del grafo**: la libreria comprende una suite completa di metodi per gestire la struttura dati, dalla sua inizializzazione alla restituzione dei vertici adiacenti, il tutto rispettando almeno le complessità richieste
- **Algoritmo di Kruskal**: la libreria precedente viene in seguito utilizzata nell'algoritmo di Kruskal, il quale ci permette di calcolare il Minimum Spanning Tree di un grafo non diretto passato come parametro
- **Utilizzo di grafo e Kruskal**: infine, le due classi precedenti vengono utilizzate per la creazione di un grafo e il successivo calcolo del MST basandosi sui dati del file csv passato come parametro (*italian_dist_graph.csv*).

Guida all'utilizzo

Abbiamo tre modi per eseguire questo esercizio, due *mirati* ed uno *generale*:

- Il primo, il più veloce, è lanciare il comando ***ant***. Il comando creerà le cartelle *classes* e *build* nel caso non esistano e eseguirà il ***jar per le Junit*** e il ***run per il main*** per la classe che utilizza il csv**.**
- Il secondo modo è lanciare il comando ***ant runJunit***. Anche questo comando creerà le cartelle *classes* e *build* nel caso non esistano, però eseguirà solamente il ***jar per le Junit***.

- Il terzo metodo è lanciare il comando ant ***runJar.*** Anche questo comando creerà le cartelle *classes* e *build* nel caso non esistano, ed eseguirà solamente ***il main dell'esercizio***.

Decisioni di Sviluppo

Libreria per struttura dati grafo

Il primo quesito che ci siamo posti nell'implementazione è stato: come rappresentare le liste di adiacenza per ogni vertice della struttura dati? Abbiamo ovviato per un **Hashmap** (tenendo in considerazione i tipi generici per vertice ed etichetta) avente come chiave **il vertice stesso**, e come valore associato una **LinkedList** di oggetti **Node** (Node rappresenta una classe "contenitore" contenente il vertice adiacente e la rispettiva etichetta dell'arco).

In corso di realizzazione, abbiamo aggiunto una struttura dati ausiliaria (un **HashSet**) contente tutti gli archi del grafo, senza eventuali duplicati nel caso di un grafo non diretto. In particolare, l'HashSet in questione contiene oggetti **Edge**, una classe che permette di formattare in modo efficace un arco con vertice-sorgente-destinazione, etichetta, ed un attributo booleano che indica se è diretto o non diretto. Quest'ultimo dato gioca un ruolo fondamentale nel riconoscimento della direzione dell'arco e del calcolo dell'HashCode, in modo tale che restituisca il medesimo risultato in un arco di ritorno nel caso di *arco non diretto*.

Riguardo ai metodi della libreria, oltre a quelli richiesti dalla consegna, abbiamo deciso di aggiungere un **ulteriore costruttore: si tratta di un metodo che permette la creazione di un grafo a partire da una collezione di oggetti Edge**, al fine di rendere il codice più snello e leggibile in alcuni punti del progetto (ad esempio, nel programma main con lettura di file csv).

Algoritmo di Kruskal

Partendo dalla libreria precedente e dalle nozioni acquisite dal corso di teoria, abbiamo creato una classe separata per l'implementazione dell'algoritmo di Kruskal e la gestione dei risultati generati. Per risultati intendiamo proprio gli attributi della classe, ovvero: il **MST** stesso, rappresentato da una **LinkedList di oggetti Node**; il numero di vertici all'interno del MST; il peso complessivo del MST, rappresentato da una variabile Double (assumiamo che le etichette

del grafo in input siano numeriche, in particolare con numeri in virgola mobile a doppia precisione).

Per quanto riguarda l'*algoritmo stesso di Kruskal* all'interno della classe, gli **input sono composti da un ArrayList degli archi dell'arco** (debitamente ordinati per il loro peso) **e una struttura dati UnionFind Set inizializzata con tutti i vertici disgiunti**. Quest'ultima struttura, in un package apposito, e' realizzata con 2 HashMap per tenere traccia di tutti gli insiemi e del loro relativo nodo padre e rank; in aggiunta, sono applicate le tecniche di compression path e rank per ottimizzare i rispettivi metodi findSet() e Union(), sfruttati dall'algoritmo di Kruskal per verificare la presenza di un cammino già esistente nel MST da un vertice all'altro.