Relazione esercizio 3-4

Componenti del gruppo
Carlino Mattia (matricola: 913397)
Ferreri Federico (matricola: 929655)
Fontana Anna (matricola: 912716)

ESERCIZIO 3

*Si implementi la struttura dati Union-Find Set (con le euristiche di **unione per rango** e **compressione del cammino**). La struttura dati deve permettere di inserire oggetti di tipo generico e non prevedere un insieme iniziale finito di elementi.*

Scelte implementative
Si è scelto di implementare la struttura dati UnionFindSet utilizzando due HashMap poiché queste consentono di effettuare le operazioni di inserimento, cancellazione e ricerca di un elemento con tempo costante (O(1)). La prima HashMap serve per tenere traccia del nodo padre (rappresentante). L'altra HashMap, invece, serve per memorizzare il valore del rango necessario per l'implementazione dell'euristica di unione per rango.

Osservazioni
Per quanto riguarda la variabile rango viene utilizzata solo quella del rappresentante dell'insieme (cioè dell'elemento in testa). Una volta che due insiemi disgiunti vengono fusi con la union, il rango dell'elemento non in testa non viene più utilizzato.

ESERCIZIO 4

*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*

Descrizione e scelte implementative generali.

Si sceglie di utilizzare una HashMap che ha come chiave il tipo generico del nodo (o vertice) e come valore una lista di archi. Questo è necessario in quanto le operazioni di inserimento, cancellazione e ricerca di una lista associata alla chiave nodo (in questo caso il tipo generico V rappresenta il tipo di un nodo del grafo) possono essere fatte con tempo costante O(1). Per mantenere la lista di archi associati a un nodo invece non sarebbe sensato utilizzare un'ulteriore HashMap (sia perché le complessità richieste sono ottenute utilizzando semplicemente una lista) sia perché la struttura dati deve essere ottimale per grafi sparsi e quindi si assume ci saranno pochi archi associati a un nodo.

Infine, per quanto riguarda la rappresentazione di un arco del grafo si crea una classe Edge. Qui un arco viene definito con tre parametri: il nodo iniziale (non necessario in realtà, ma inserito per comodità), il nodo finale e l'etichetta dell'arco.

Scelte implementative per la complessità delle operazioni.

- *Creazione di un grafo vuoto – O(1)*

  Costruttore della classe: prende come parametro una stringa che dice se il grafo creato è "directed" oppure non diretto (se la stringa è diversa da "directed", in ogni caso, viene creato un grafo non diretto). Si inizializzano gli attributi privati della classe.

- *Aggiunta di un nodo – O(1)*

  Se il nodo non esiste viene inserito nella HashMap con l'istruzione put (tempo costante O(1)). Se il nodo esiste già allora non viene inserito e il metodo ritorna false.

- *Aggiunta di un arco – O(1)*

  Se i due nodi tra cui si vuole inserire l'arco non esistono vengono creati chiamando il metodo per aggiungere un nodo. Se l'arco esiste già non viene aggiunto e il metodo ritorna false. L'utilizzo della HashMap permette di eseguire questa operazione con tempo costante O(1): date le chiavi (nodi) con l'operazione get si può ottenere la lista di archi associata al nodo e con l'istruzione add della lista l'arco viene aggiunto in fondo alla lista (tempo di inserimento in lista è O(1)).

- *Verifica se il grafo è diretto – O(1)*

  É sufficiente ritornare l'attributo 'directed' della classe.

- *Verifica se il grafo contiene un dato nodo – O(1)*

  Il metodo containsKey(node) della HashMap permette di trovare con tempo costante O(1) se un nodo è già stato inserito.

- *Verifica se il grafo contiene un dato arco – O(1) (*)*

  Questo metodo ha complessità costante O(1) solo se si assume che il grafo sia sparso: in questo caso l'istruzione contains legata alla lista degli archi può essere considerata di tempo costante poiché si assume ci siano pochi archi per un dato nodo.

- *Cancellazione di un nodo – O(n)*

Occorre sia eliminare il nodo sia eliminare qualsiasi arco contenga lo stesso nodo come nodo iniziale e/o finale (sia iniziale che finale nel caso di un nodo che punta a se stesso). Nel caso in cui il grafo sia diretto si deve scorrere la lista di archi (ottenuta grazie al metodo che ritorna la lista di archi che ha complessità O(n)) e rimuovere l'arco quando il nodo di partenza o quello di arrivo sono uguali al nodo da rimuovere. Questo metodo ha complessità O(n) poiché: O(n) (metodo che ritorna una lista di archi) + O(n) (ciclo che scorre la lista) = O(n).
Questa soluzione andrebbe bene sia per grafi diretti che non diretti, ma si decide di gestire il caso in cui il grafo sia non diretto in modo più efficiente: in un grafo non diretto, per definizione, il nodo non potrà essere presente solamente come nodo finale, quindi si può sfruttare il metodo che ritorna la lista di adiacenza di un nodo e iterare su questa lista di nodi. La complessità è sempre O(n), ma si suppone che la lista di adiacenza di un nodo contenga un numero di elementi minore rispetto alla lista di tutti gli archi del grafo.

- *Cancellazione di un arco – O(1) (*)*

  La complessità del metodo è costante per quanto riguarda grafi sparsi poiché si può supporre che l'istruzione get della HashMap che prende in input la chiave ritorni una lista di archi associati al nodo ritorni una lista con pochi elementi e, quindi, anche l'istruzione remove (che normalmente ha complessità O(n) si può considerare con tempo costante O(1)).

- *Determinazione del numero di nodi – O(1)*

  Si ritorna il valore della chiamata al metodo size della HashMap che rappresenta il numero di chiavi (e quindi di nodi) inserite.

- *Determinazione del numero di archi – O(n)*

  Si ritorna il valore della variabile intera che conta il numero di archi del grafo. Questa variabile ausiliaria viene inizializzata a 0 nel costruttore e incrementata di 1 ogni volta che si aggiunge un arco nel grafo.

- *Recupero dei nodi del grafo – O(n)*

  Si crea una lista inizialmente vuota che verrà ritornata dal metodo. Con complessità O(1) dell'operazione keySet l'HashMap ottiene la lista di chiavi (nodi) inserite. Con complessità O(n) la lista viene scorsa e ogni nodo viene aggiunto alla lista da ritornare.

- *Recupero degli archi del grafo – O(n)*

  Si utilizza una variabile di supporto di tipo HashSet contenente la lista di tutti gli archi del grafo. Ogni volta che inseriamo un arco lo aggiungiamo anche in

questo set e quando rimuoviamo un arco lo rimuoviamo anche dal set. Quindi, nel metodo, si inizializza una lista vuota e scorrendo la variabile di tipo HashSet si aggiunge ogni volta un arco alla lista da ritornare. L'inserimento e la rimozione di un elemento da un HashSet ha tempo costante O(1) e quindi non modifica le complessità dei metodi.

- *Recupero nodi adiacenti di un dato nodo – O(1) (*)*

  Si crea una lista inizialmente vuota che verrà ritornata dal metodo. Con complessità O(1) si ottiene la lista di archi associati a un nodo grazie all'operazione get della HashMap. Con l'ipotesi che il grafo sia sparso questa lista conterrà pochi elementi e quindi scorrere la lista si può assumere abbia complessità costante O(1). Ad ogni iterazione si aggiunge il nodo finale dell'arco alla lista da ritornare. La variabile booleana pointToItself viene utilizzata nel caso in cui il grafo sia non diretto e ci sia un elemento che punti a se stesso per evitare di inserirlo due volte nella lista.

- *Recupero etichetta associata a una coppia di nodi – O(1) (*)*

  Con complessità O(1) si ottiene la lista di archi associati a un nodo grazie all'operazione get della HashMap. Con l'ipotesi che il grafo sia sparso questa lista conterrà pochi elementi e quindi scorrere la lista si può assumere abbia complessità costante O(1). Se il nodo finale di un arco è uguale a quello di cui si vuole trovare la coppia di nodi allora si ritorna il valore dell'etichetta.

  Infine, si è deciso di aggiungere alla libreria il seguente metodo poiché potrebbe essere utile a un utente.

- *Recupero degli archi di un nodo del grafo – O(1)*

  Dopo aver fatto gli opportuni controlli di esistenza del parametro passato si ritorna la lista di archi associata alla chiave (nodo)**.**

Algoritmo di Kruskal

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

Si è dovuta modificare la classe Edge inizialmente creata in modo che implementi l'interfaccia Comparable<Edge<V,E>>: si è aggiunto il metodo compareTo che compara le etichette di due archi. Questo è necessario per poter ordinare la lista di archi del grafo grazie al metodo sort della classe Collection.

Osservazioni

Del file italian_dist_graph.csv si evince che, come riportato dal testo, il dataset è poco accurato: alcuni archi vengono ripetuti scambiando l'ordine delle due località, ma con la stessa distanza tra le due località. Infatti il file riporta inizialmente 56640 righe che fanno pensare a 56640 archi che verranno inseriti, ma invece solo 48055 saranno archi del grafo iniziale creato.
Dopo aver calcolato la minima foresta ricoprente con l'algoritmo di Kruskal si nota che, come ci si aspetta, il peso totale del grafo, così come il numero di archi, è notevolmente diminuito. Inoltre, il grafo creato inizialmente (così come quello ottenuto ora) è un grafo sconnesso con tre sottografi connessi (foresta con tre alberi): il numero di archi, infatti, è inferiore al numero di archi necessari per connettere tutti i nodi di un grafo connesso (numero di archi = numero di nodi -1).