Laboratorio Di Algoritmi e Strutture Dati Anno 2021/2022

Stefano Peddoni Matr 839138

Davide Laguardia Matr 822358

Relazione Esercizio 3 – 4

Nell'esercizio 3 è stato implementato un Heap minimo che verrà poi utilizzato all'interno di Dijkstra.

Nell'esercizio 4 è stata implementata la struttura di un Grafo.

- Qui di seguito elencheremo i metodi usati dall'Hashtable all'interno di Graph:
    - Private Hashtable<T, Vertex<T,S>> hash;

METODO	COMPLESSITA'	DESCRIZIONE
hash.put	O(1)	Viene utilizzata per inserire la coppia <chiave, valore>.
hash.containsKey	O(1)	Viene utilizzata per verificare se la chiave è contenuta nel grafo e ritorna se è presente
hash.get	O(1)	Viene utilizzata per verificare la chiave e restituisce il valore associato.
hash.size	O(1)	Restituisce la dimensione del grafo
hash.remove	O(1)	Viene utilizzata per rimuovere l'elemento dal grafo
hash.keySet	O(1)	Restituisce una vista delle chiavi

Inoltre, è stata utilizzata anche un'interfaccia Set: Set in Java **è un'interfaccia che fa parte di Java Collection Framework e implementa l'interfaccia Collection**. Una raccolta di set fornisce le caratteristiche di un set matematico. Un set può essere definito come una raccolta di oggetti non ordinati e non può contenere valori duplicati. E' stato quindi scelto di utilizzare la Set, al posto della List proprio per il fatto che la Set viene definita come una raccolta di oggetti senza duplicati, cosa che invece non permette List.

Le seguenti operazioni richieste sono:

Creazione di un grafo vuoto: In **graph**, è stato creato un grafo vuoto attraverso una Hashtable, un parametro direct che essendo boolean determina se il grafo menzionato è diretto (TRUE) oppure non diretto (FALSE). La complessità del problema è di O(1) non avendo cicli.

Aggiunta di un nodo: In **addVertex**, aggiungo un nodo all'interno del grafo, ritorno TRUE se il parametro v viene aggiunto al grafo, FALSE nel caso contrario. La complessità asintotica del problema è di O(1) non avendo cicli.

Aggiunta di un arco: In **addEdge**, aggiungo un arco all'interno del grafo. In questo caso nel metodo addEdge vado a verificare se il grafo è orientato o non orientato. La complessità del problema è di O(1) non avendo cicli.

Verifica se il grafo è diretto: In **isDirect**, verifico se il grafo in questione è diretto o no. Nel primo caso ritorna TRUE, nel secondo caso FALSE. La complessità del problema è di O(1) non avendo cicli.

Verifica se il grafo è indiretto: In **isUndirect**, verifico se il grafo in questione è indiretto o no. Nel primo caso ritorna TRUE, nel secondo caso FALSE. La complessità del problema è di O(1) non avendo cicli.

Verifica se il grafo contiene un dato nodo: In **containsVertex**, verifico se un dato nodo è contenuto all'interno del grafo. In questo caso verifico se vertexToFind è contenuto all'interno di esso. La complessità del problema è di O(1) non avendo cicli.

Verifica se il grafo contiene un dato arco: In **containsEdge**, verifico se un dato arco è contenuto all'interno del grafo. In questo caso verifico se source e destination sono contenuti all'interno di esso. La complessità del problema è di O(1) non avendo cicli.

Cancellazione di un nodo: In **removeVertex**, passiamo come parametro v l'elemento che vogliamo cancellare. La complessità asintotica del problema è O(n) poiché all'interno del metodo abbiamo un ciclo for.

Cancellazione di un arco: In **removeEdge**, passiamo come parametro source e destination gli elementi che vogliamo cancellare. La complessità del problema è di O(1) non avendo cicli.

Determinazione del numero di nodi: In **getNumberVertex** determino il numero di nodi presenti nel grafo. La complessità asintotica è O(1) non avendo cicli.

Determinazione del numero di archi: In **getNumerEdges** determino il numero di archi presenti nel grafo. La complessità asintotica del problema è di O(n) avendo un ciclo while all'interno.

Recupero dei nodi del grafo: In questo caso abbiamo creato due funzioni che permettono di recuperare la sorgente del vertice con **getVertex** e **getAllVertex** per recuperare tutti i nodi presenti nel grafo. La complessità asintotica del problema è O(n) avendo un ciclo for all'interno.

Recupero degli archi del grafo: In questo caso abbiamo creato due funzioni chiamate **getEdge** prendendo in considerazione un singolo arco e **getEdges** per recuperare tutti gli archi presenti nel grafo. La complessità asintotica del problema è O(n) avendo un ciclo for all'interno.

Recupero nodi adiacenti di un dato nodo: In **getAdjacentVertex**, recupero tutti i nodi adiacenti presenti nel grafo. La complessità asintotica del problema è O(1).

Recupero etichetta associata a una coppia di nodi: Utilizziamo **getVertexLabel** e **getAdjacentVertexLabel**, recupero l'etichetta a,b associata ad una coppia di nodi presenti nel grafo. La complessità asintotica del problema è O(1).

Implementazione di Dijkstra:

L'algoritmo di Dijkstra è un algoritmo per trovare i percorsi più brevi tra i nodi in un grafo. Per un dato nodo sorgente nel graph, l'algoritmo trova il percorso più breve tra quel nodo e ogni altro nodo. L'algoritmo di Dijkstra si basa sul principio del rilassamento, in cui valori più accurati sostituiscono gradualmente un'approssimazione alla distanza corretta fino a raggiungere la distanza più breve.

Partendo dalla libreria creata all'interno di Graph e all'interno di Heap e dalle nozioni acquisite dal corso di teoria, abbiamo creato una classe separata per l'implementazione dell'algoritmo di Dijkstra, e convertito lo pseudo codice visto in codice Java.

Dopo aver implementato l'algoritmo di Dijkstra, abbiamo creato una classe DijkstraMain contente l'avvio dell'algoritmo per verificare il percorso tra Torino e Catania tramite il file csv.

Per poter avviare sia l'esercizio 3 che l'esercizio 4 è stato creato un file readme all'interno delle due cartelle con la spiegazione per l'esecuzione del programma contenente i test.

Risultati: Come sorgente si è scelta Torino, mentre come destinazione si è scelta Catania; il risultato del percorso tra le due città risulta essere di 1207.68Km