Progetto di Algoritmi
A.A. 2016-2017

Mattia Nocerino 818089
Angelo Savarino 814991
Elisa Solinas 811737

Updated on 22 giugno 2017

Indice

1 Esercizio 1: Algoritmi di ordinamento

Gli algoritmi di ordinamento `SelectionSort` e `InsertionSort` hanno complessità quadratica, per cui i risultati ottenuti sul file `records.csv`(composto da 20.000.000 di record) non sono stati soddisfacenti (nessuna delle nostre macchine è riuscita a completare l'ordinamento in meno di 10 minuti).

Attraverso l'algoritmo `QuickSort`, con scelta random del `pivot`, i risultati ottenuti sono stati i seguenti:

- Ordinamento sul primo campo: 65390 ms
- Ordinamento sul secondo campo: 25794 ms
- Ordinamento sul terzo campo: 38633ms

La complessità del `QuickSort` nel caso peggiore è n^2, mentre è $n \log n$ nel caso medio. È importante notare, però, che è molto poco probabile che si verifichi il caso peggiore, in particolare sfruttando la **randomizzazione**, cioè scegliendo il pivot in maniera casuale.

2 Esercizio 2: Albero n-ario

L'albero n-ario è stato implementato attraverso la classe `Tree`, che ha come attributo un elemento `root`, di tipo `Node`.
Il tipo `Node` contiene l'etichetta del nodo (il suo valore), un puntatore al padre e un puntatore al fratello

destro. Contiene, inoltre, il grado (`grade`) del nodo, ossia il numero di figli del nodo.

Il metodo `getBinaryTree()`, che ci permette di costruire un albero binario a partire dall'albero n-ario, trasforma l'albero in una lista, la ordina utilizzando l'algoritmo `quickSort` implementato nell'esercizio precedente, in modo da inserire i nodi nell'ordine corretto, utilizzando il metodo `binaryAdder()`.

3 Esercizio 3: Verifica MaxHeap

La classe `MaxHeap` è una classe statica, che contiene due metodi equivalenti, uno iterativo e uno ricorsivo, per verificare se l'array passato come parametro rappresenta o meno un `MaxHeap`.

4 Esercizio 4: Struttura UnionFind

La struttura `UnionFind` è stata implementata attraverso una struttura `HashMap`, avente come **chiavi** gli elementi dell'insieme e come **valori** strutture `UFNode`, contenenti il rango della chiave e il rappresentante dell'insieme cui la chiave appartiene.

5 Esercizio 5: Grafo

5.1 Implementazione della struttura grafo diretto

Per la rappresentazione di un grafo diretto utilizziamo la lista di adiacenza, implementata attraverso una `HashMap`, in cui le chiavi rappresentano i vertici del grafo, mentre i valori rappresentano attraverso dei nodi gli archi connessi a ciascun vertice-chiave.

5.2 Implementazione della struttura grafo non diretto

I grafi non diretti vengono rappresentati attraverso la classe `UndirectedGraph`, che è una sottoclasse di `DirectedGraph`, descritto nel paragrafo precedente.
I metodi implementati nella classe `UndirectedGraph` sostituiscono i metodi della classe `DirectedGraph` sono i seguenti:

- Il metodo `addEdge`, che per ogni arco $u \to v$, aggiunge sia (u, v) che (v, u), in modo da rappresentare un grafo non orientato.
- Il metodo `weight()`, che divide a metà il peso del grafo, in quanto ciascun arco non orientato viene rappresentato mediante un doppio arco.

5.3 Implementazione dell'algoritmo di Kruskal

Il costruttore della classe `MSTKruskal` prende in input un grafo non diretto G e produce una lista degli archi che compongono l'albero minimo di copertura di G.
L'algoritmo è stato implementato utilizzando una `UnionFind`, in cui ciascun insieme rappresenta una componente connessa del grafo.
Inoltre, l'algoritmo sfrutta il metodo `getEdges()` della classe `DirectedGraph`, che restituisce un `ArrayList` ordinato contenente tutti gli archi di G.