Esercitazione 8
Grafi - SSSP e MST

Corso di Fondamenti di Informatica II

Algoritmi e strutture dati

A.A. 2017/2018

1 Giugno 2018

Sommario

Scopo di questa esercitazione è implementare le operazioni di *single source shortest path* e *minimum spanning tree*.

Per svolgere l'esercitazione è necessario collegarsi al sito:
`www.sites.google.com/diag.uniroma1.it/crocefederico/materiale`
e scaricare i file contenuti nella cartella dedicata a questa esercitazione.

Attenzione: L'esercitazione può essere svolta sia in linguaggio C che in Java. E' altamente consigliato sviluppare le soluzioni in laboratorio usando il linguaggio C e svolgere l'esercitazione in Java come esercizio per casa. Inoltre è altamente consigliato di far girare le soluzioni in C sotto `valgrind`.

1 Visita in ampiezza

Dato un grafo diretto G, l'esercizio richiede di sviluppare un algoritmo (`bfs`) che faccia una visita in ampiezza dei nodi di G. L'output della procedura sarà una stampa a video della sequenza dei nodi visitati, ordinata secondo l'ordine di visita. Ad esempio, invocando la procedura sul grafo rappresentato in Fig. 1 si può ottenere la stampa:

```
1
2
3
4
```

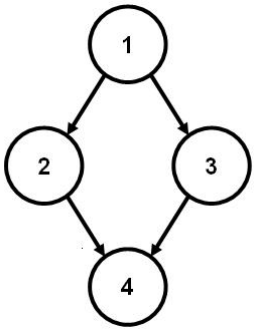

Figura 1: Un esempio di grafo diretto.

Materiale di supporto: Viene fornito il modulo `graph.h` (`Graph⟨V⟩`) che rappresenta un grafo semplice orientato, a pesi interi. `V` è il tipo di dato usato per etichettare i nodi del grafo. Il grafo è rappresentato mediante lista di incidenze: ogni nodo ha associata una lista di tutti i suoi **archi** (esplicitati nel tipo `Edge⟨V⟩`) uscenti.

Specifica. Realizzare la funzione `bfs` descritta nel header `graph_services.h` (`GraphServices.java`) che, preso in ingresso un grafo, stampa a video l'ordine di accesso dei nodi del grafo secondo una visita in ampiezza.

```
void bfs(graph* g);

public static <V> void bfs(Graph<V> g);
```

Si ricorda che il grafo potrebbe non essere fortemente connesso.

Suggerimento: Si consiglia di utilizzare una struttura **Queue** per gestire l'ordine di visita dei nodi del grafo.

Si proceda a testare il codice sviluppato utilizzando `driver.c` oppure `Diver.java`.

2 Single Source Shortest Path

L'algoritmo di Dijkstra risolve il problema *single source shortest path* per grafi pesati, se tutti i pesi degli archi sono positivi. Si chiede di realizzare una funzione (`sssp`) che, preso in ingresso un grafo ed un nodo *sorgente*, stampi a video la distanza minima dal nodo indicato a tutti gli altri nodi del grafo. Un nodo non raggiungibile da quello *sorgente* può essere indicato con distanza *infinita*. Ad esempio, invocando la procedura sul grafo in Fig. 2 avendo S come nodo *sorgente*, si ottiene la stampa:

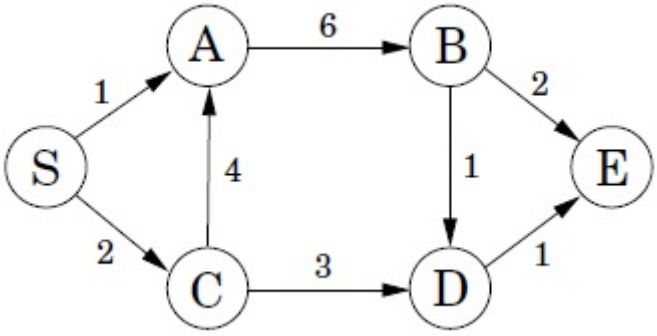

Figura 2: Un grafo diretto con gli archi a pesi positivi.

```
S 0
A 1
C 2
B 7
D 5
E 6
```

`L'ordine non è importante.`

Materiale di supporto: viene fornita l'implementazione di un `MinHeap` per la gestione di una coda con priorità, sia nel linguaggio C che in Java.

Specifica: Realizzare la funzione con segnatura indicata di seguito, descritta nel header `graph_services.h` (`GraphServices.java`) che, preso in ingresso un grafo ed un nodo, stampa a video le distanze di tutti i nodi del grafo da quello dato in input, secondo la formattazione dell'esempio.

```c
void single_source_shortest_path(graph* g, graph_node* source);
```

```java
public static void <V> sssp(Graph<V> g, Node<V> source);
```

Suggerimento per C: è possibile tenere traccia della distanza di ciascun nodo dal nodo *source* dato in input alla funzione, facendo uso del campo `int dist`; presente in ogni nodo del grafo. In Java, è possibile fare uso del *Java Collection Framework* per ottenere un risultato simile.

Si proceda a testare il codice sviluppato utilizzando `driver.c` oppure `Diver.java`.

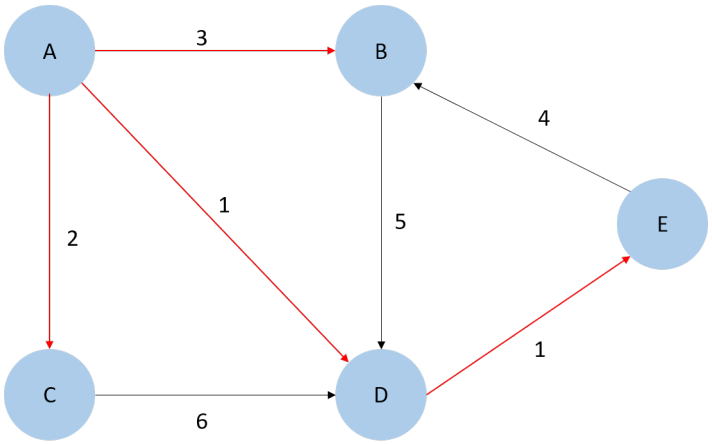

Figura 3: Un grafo diretto con gli archi a pesati. In rosso viene indicato il suo minimum spanning tree.

3 Minimum Spanning Tree

Il problema del *minimum spanning tree* consiste nel calcolare, per un dato grafo pesato, un sottografo aciclico a peso minimo che ricopra tutti i nodi del grafo originale. L'algoritmo di Kruskal risolve questo problema facendo uso delle primitive di `Union – Find`. Si faccia riferimento alle slides del corso per i dettagli su tali primitive e sulle possibili implementazioni ed ottimizzazioni. Si chiede di realizzare una funzione (`mst`) che, preso in ingresso un grafo con gli archi pesati G, stampi a video il sottoinsieme degli archi di G che costituiscono un *minimum spanning tree* del grafo. L'ordine degli archi stampati non è rilevante. Ad esempio, invocando la procedura sul grafo in Fig. 3 si ottiene la stampa:

```
a b
a c
a d
d e
```

`L'ordine non è importante.`

Materiale di supporto: viene fornita l'implementazione delle primitive `Union – Find` per operare su insiemi disgiunti. Si noti che tale implementazione gestisce insiemi di numeri. E' possibile mappare ogni nodo del grafo ad un numero facendo uso del campo `int map`; presente in ogni nodo del grafo.

Specifica. Realizzare la funzione `mst` descritta nel header `graph_services.h` (`GraphServices.java`) che, preso in ingresso un grafo, stampa a video un sottoinsieme degli archi del grafo che compongono un *minimum spanning tree*.

```
void mst(graph* g);

public static void <V> mst(Graph<V> g);
```

Suggerimento: si consiglia di utilizzare una coda con priorità per rappresentare gli archi del grafo da scegliere.

Si proceda a testare il codice sviluppato utilizzando `driver.c` oppure `Diver.java`.

Riferimenti bibliografici

[1] M. T. Goodrich, R. Tamassia and M. H. Goldwasser. *Algoritmi e strutture dati in Java*. Apogeo Education - Maggioli Editore, 2015.