Esame di

Algoritmi e strutture dati (parte di Fondamenti di informatica II 12 CFU)
Algoritmi e strutture dati (V.O., 5 CFU)
Algoritmi e strutture dati (Nettuno, 6 CFU)

Appello del 19-07-2021 – a.a. 2020-21 – Tempo: 2 ore e 30 minuti – somma punti: 32

Istruzioni

1. Per prima cosa si prega di indicare all'inizio del compito i) anno di immatricolazione; ii) se il proprio esame di Fondamenti 2 consiste di Algoritmi e Strutture Dati + Modelli e Linguaggi (*in tal caso scrivere MOD*) oppure Algoritmi e Strutture Dati + Progettazione del Software (*solo studenti che nel periodo che va dal 2014-15 al 2017-18 sono stati iscritti al II anno, estremi inclusi*). *Nel secondo caso scrivere PSW*.
2. **Nota bene.** Per ritirarsi è sufficiente scriverlo all'inizio del file contenente le vostre risposte.

Avviso importante. La risposta al quesito di programmazione va data nel linguaggio scelto (C o Java), usando le interfacce messe a disposizione dal docente. *Il programma va scritto usando l'editor interno di exam.net*.

Mentre non ci aspettiamo che produciate codice compilabile, una parte del punteggio sarà comunque assegnata in base alla leggibilità e chiarezza del codice che scriverete, *a cominciare dall'indentazione*, oltre che rispetto ai consueti requisiti (aderenza alle specifiche ed efficienza della soluzione proposta). Inoltre, errori grossolani di sintassi o semantica subiranno penalizzazioni. *Si noti che l'editor di exam.net consente di indentare i programmi*.

Anche le risposte ai quesiti 2 e 3 vanno scritte usando l'editor interno di exam.net. È possibile integrare i contenuti di tali file con eventuale materiale cartaceo scansionato *unicamente per disegni, grafici, tabelle, calcoli*.

Quesito 1: Progetto algoritmi C/Java [soglia minima per superare l'esame: 5/30]

In questo problema si fa riferimento a grafi semplici, in generale diretti e semplicemente connessi. I grafi sono rappresentati attraverso liste di adiacenza. In particolare, a ciascun nodo u sono associate due liste collegate di elementi: ciascun elemento della prima lista è il riferimento a uno dei vicini (uscenti) di u, mentre ciascun elemento della seconda lista è il riferimento a un vicino entrante, ossia un nodo che ha un arco diretto verso u. I nodi sono rappresentati dalle classi/strutture `GraphNode/graph_node` , mentre il grafo è rappresentato dalle classi/strutture `Graph/graph` . La gestione delle liste di nodi deve essere effettuata mediante il tipo `linked_list` (C) o la classe `java.util.LinkedList` (Java); per la classe `LinkedList` i principali metodi per la gestione dovrebbero essere noti allo studente, ma alcuni di essi sono riportati in fondo a questo documento per comodità. Analogamente, in fondo a questo documento sono descritti i principali metodi per accedere al tipo `linked_list` in C o per usare iteratori sul tipo `linked_list`.

Le interfacce delle classi/moduli che implementano il grafo e i suoi nodi sono descritti nell'appendice di questo documento. Sono riportati alcuni campi/metodi/funzioni utili allo svolgimento degli esercizi proposti. Si noti che in generale, soltanto alcuni dei campi/funzioni/metodi presenti sono necessari per lo svolgimento dell'esercizio.

Tutto ciò premesso, risolvere al calcolatore quanto segue, in Java o C, con l'avvertenza che gli esempi dati nel testo che segue fanno riferimento alla figura seguente:

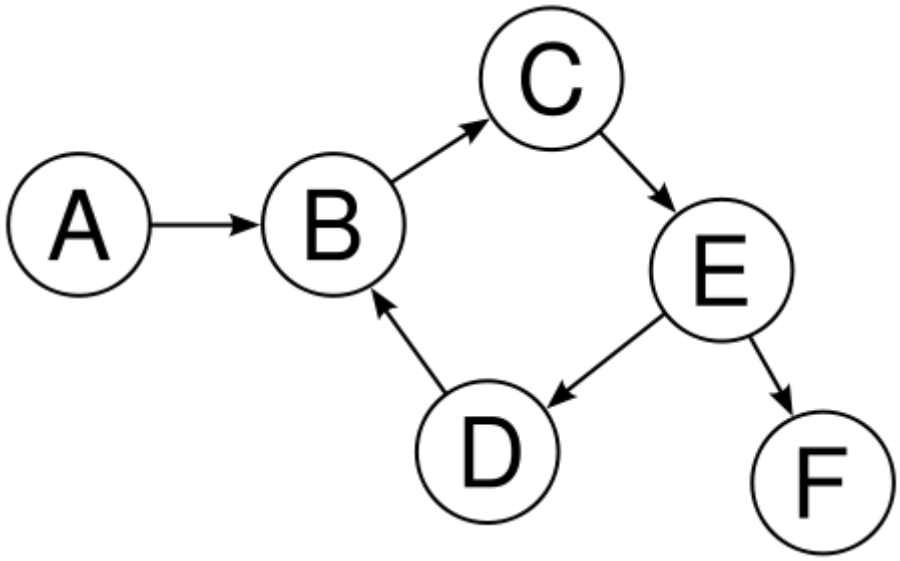

Figura 1. Esempio di grafo diretto pesato. La componente *fortemente connessa* contenente il nodo **B** include il sottoinsieme {**B**, **C**, **D**, **E**} dei nodi, mentre la componente fortemente connessa cui appartiene il nodo **A** è {**A**}.

1. Implementare la funzione/metodo `scc(Graph<V> g, Node<V> source)` della classe `GraphServices` (o la funzione `void scc(graph* g, graph_node* source)` del modulo `graph_services` in C) che, dato un grafo `g` e un (oggetto) nodo `source` , stampa la lista dei nodi della componente *fortemente connessa* alla quale appartiene il nodo `source` . Ad esempio, rispetto alla Figura 1, se `source` fosse il nodo B la funzione/metodo dovrebbe stampare (l'ordine in cui vengono stampati i valori dei nodi non è importante)

   ```
   {B, C, D, E}
   ```

   Se invece `source` fosse il nodo A, allora il metodo/funzione dovrebbe stampare `{A}` in quanto la componente fortemente connessa cui appartiene A contiene soltanto A.

   Suggerimento. Si ricorda agli studenti che la componente fortemente connessa cui appartiene un nodo u può essere individuata eseguendo due visite a partire da u. La prima visita avviene sul grafo di partenza, la seconda sul grafo ottenuto invertendo le direzioni degli archi (che in questo problema sono diretti). I nodi che sono raggiungibili in entrambe le visite appartengono alla componente fortemente connessa di cui fa parte u.

   Nota: Si noti che `GraphNode` (Java) `graph.h` e `graph.c` (C) mettono a disposizione primitive per accedere alla lista dei vicini uscenti ed entranti di ciascun nodo.
   Punteggio: [10/30]

Quesito 2: Algoritmi

1. Si consideri il grafo non diretto e pesato (*i pesi sugli archi non sono rilevanti ai fini di questo esercizio*) nella figura sottostante.

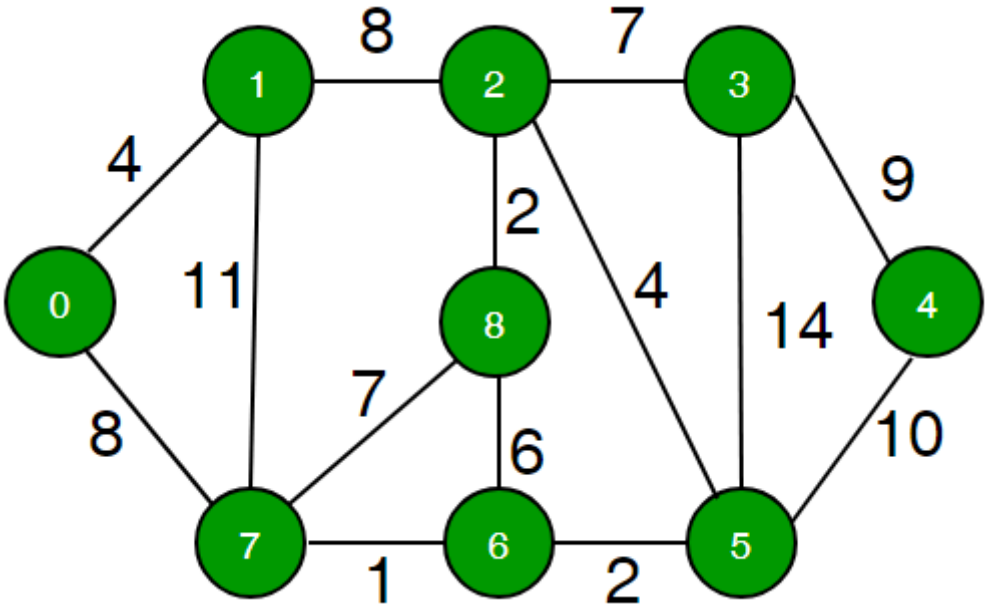

Figura 1

Disegnare un possibile albero di visita in *postordine* o DFS (si noti che sono possibili più alberi di visita in postordine ma non tutti gli alberi ricoprenti del grafo corrispondono a una visita in postordine) *a partire dal nodo avente etichetta* i, dove $i = (n + 7) \mod 8$, ed n è l'ultima cifra del vostro numero di matricola (quella più a destra). Per la precisione occorre: i) disegnare l'albero e ii) associare a ogni nodo (alla sua etichetta numerica) una delle prime lettere dell'alfabeto, in modo che "A" sia l'etichetta associata al *primo nodo visitato*, "B" al secondo e così via fino all'ultimo nodo visitato, che riceverà etichetta "I".

Nota bene: si noti che l'ordine di visita dei nodi non corrisponde necessariamente a quello in cui i nodi sono raggiunti per la prima volta. Ciò dipende dal tipo di visita effettuato

Nota: non usare la propria matricola per rispondere al quesito comporta una penalizzazione.

Punteggio: [5/30]

2. Si consideri la seguente versione dell'algoritmo di Dijkstra, del tutto simile a quella vista a lezione, se non per l'importante differenza che questa versione usa una lista disordinata invece cha una coda di priorità per tenere traccia dell'insieme dei nodi ancora da esaminare:

```
Algorithm ShortestPath(G, s):
    Input: Grafo G non diretto con pesi non-negativi interi sugli archi,
vertice
    sorgente s
    Output: per ogni vertice v di G, la lunghezza D[v] del cammino minimo
da s a v
    // D è un vettore indicizzato dalle etichette dei nodi
    Inizializza D[s] = 0 e D[v] = MAXINT per ogni vertice v != s.
    // L è una lista disordinata contenente i vertici di G
    while L is not empty do
        //Rimuove da L il nodo u a distanza stimata minima
        u = L.removeMin( )
```

```
        for (ogni arco (u, v) tale che v appartiene a L)
            // Esegui il rilassamento sull'arco (u, v)
            if D[u] + w(u, v) < D[v] then
                D[v] = D[u] + w(u, v)
    return the label D[v] of each vertex v
```

Nell'algoritmo, `removeMin()` è una funzione che rimuove da `L` e restituisce il nodo avente valore di `D[v]` minimo tra quelli dei nodi ancora presenti in `L`.

Si calcoli la complessità asintotica di caso peggiore dell'algoritmo così ottenuto.

Occorre dare un'argomentazione quantitativa, rigorosa (prova) e chiara, giustificando adeguatamente la risposta. Il punteggio dipenderà soprattutto da questo.

Punteggio: [7/30]

Punteggio: [5/30]

3. Si consideri ancora una volta il grafo non diretto e pesato della Fig. 1 sopra (stavolta i pesi sono importanti per risolvere l'esercizio).

   Si dimostri che l'arco $(5, 3)$ di peso 14 non può far parte di alcun cammino minimo che porta dal nodo 0 al nodo 3.

   Si noti che occorre dare una dimostrazione formale di questo fatto, sfruttando le proprietà dei cammini minimi. Il punteggio dipenderà dal rigore dell'argomentazione proposta e dalla chiarezza espositiva. Non occorre scrivere tanto ma scrivere bene.

   Punteggio: [5/30]

Quesito 3:

Si supponga di avere due liste non ordinate contenenti rispettivamente n_1 e n_2 coppie `(k, e)`, dove `k` è un identificatore univoco di tipo stringa ed `e` è il riferimento a un oggetto (ad esempio una pagina Web).

- Si proponga il più efficiente algoritmo possibile che, prese in ingresso le due liste, restituisca il sottoinsieme delle chiavi presenti in entrambe le liste.
- Si calcoli il costo asintotico dell'algoritmo proposto.

  Punteggio: [3/30]

Occorre descrivere chiaramente l'algoritmo proposto e indicare eventuali strutture dati ausiliarie usate. Occorre dare un'argomentazione quantitativa per il costo asintotico. Il punteggio dipenderà molto dalla chiarezza espositiva e dalla solidità delle argomentazioni proposte.

Appendice: interfacce dei moduli/classi per il quesito 2

Di seguito sono descritti i campi/moduli/funzioni delle classi Java o moduli C che si suppone siano utilizzabili.

Interfacce Java

In Java si suppone sia suppone siano già implementate e disponibili le classi `GraphNode` e `Graph`, che rispettivamente implementano il generico nodo e un grafo non diretto. Ovviamente, in Java si assumono disponibili tutte le classi delle librerie standard, come ad esempio `java.util.LinkedList` ecc.

Classe GraphNode

```java
public class GraphNode<V> implements Cloneable{
    public static enum Status {UNEXPLORED, EXPLORED, EXPLORING}

    protected V value; // Valore associato al nodo
    protected LinkedList<GraphNode<V>> outEdges; // Lista dei vicini uscenti
    protected LinkedList<GraphNode<V>> inEdges; // Lista dei vicini entranti

    // keep track status
    protected Status state; // Stato del nodo
    protected int timestamp; // Campo intero utilizzabile per vari scopi

    @Override
    public String toString() {
        return "GraphNode [value=" + value + ", state=" + state + "]";
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return (GraphNode<V>) this;
    }
}
```

Metodi messi a disposizione dalla classe Graph. Di seguito si descrivono le interfacce dei metodi utili alla risoluzione degli esercizi e messi a disposizione dalla classe `Graph`.

```java
// Restituiscee una lista di riferimenti ai nodi del grafo
public List<GraphNode<V>> getNodes();

// Restituisce una lista con i riferimenti dei vicini uscenti del nodo n
(outEdges nella classe GraphNode)
public List<GraphNode<V>> getOutNeighbors(GraphNode<V> n);

// Restituisce una lista con i riferimenti dei vicini entranti del nodo n
(inEdges nella classe GraphNode)
public List<GraphNode<V>> getInNeighbors(GraphNode<V> n);
```

Metodi potenzialmente utili della classe LinkedList.

```java
// Appende e alla fine della lista. Restituisce true
boolean add(E e);

// Rimuove e restituisce l'elemento in testa alla lista.
E remove(); // E indica il tipo generico dell'elemento
```

Scheletro della classe GraphServices

Di seguito, lo scheletro della classe `GraphServices` con le segnature dei metodi che essa contiene.

```java
public class GraphServices<V>{

  public static <V> void scc(Graph<V> g, Graph.GraphNode<V> source) {
        /* DA IMPLEMENTARE */
  }

}
```

Interfacce C

graph.h (solo tipi principali)

```c
#include "linked_list.h"
#include <stdio.h>

typedef enum {UNEXPLORED, EXPLORED, EXPLORING} STATUS;

/**
* Grafo semplice non diretto rappresentato mediante lista delle adiacenze.
*/

typedef struct graph {
    linked_list* nodes;     // lista di graph_node
    int n_nodes;
    int n_edges;
} graph;

typedef struct graph_node {
    int key; // progressivo creazione, a partire da zero
    int timestamp;
    STATUS state;
    int value; // valore associato al nodo - naturale letto da file
    linked_list* out_edges;  // lista di adiacenza - archi uscenti
    linked_list* in_edges;  // lista di adiacenza - archi entranti
} graph_node;
```

Scheletro del modulo C graph_services

Di seguito lo scheletro del metodo `graph_services.c` e le segnature delle funzioni da implementare.

```c
#include "graph.h"

void scc(graph* g, graph_node* source) {
        /* DA IMPLEMENTARE */
}
```

linked_list.h (solo parte)

```c
typedef struct linked_list_node {
    void *value;
    struct linked_list_node *next;
```

```
    struct linked_list_node *pred;
} linked_list_node;

typedef struct linked_list {
    linked_list_node *head;
    linked_list_node *tail;
    int size;
} linked_list;

typedef struct linked_list_iterator linked_list_iterator;

/**********************
      linked_list
**********************/
/**
Crea una nuova lista.
*/
linked_list * linked_list_new();

/**
Aggiunge in testa alla lista ll, un nodo che punta a value.
*/
void linked_list_insert_head(linked_list* ll, void* value);

/**
Aggiunge in coda alla lista ll, un nodo che punta a value.
*/
void linked_list_insert_tail(linked_list* ll, void* value);

/**
Come linked_list_insert_tail(linked_list* ll, void* value).
*/
void linked_list_add(linked_list * ll, void * value);


/**
Aggiunge alla lista ll un nodo che punta a value, subito dopo predec
*/
void linked_list_insert_after(linked_list * ll, linked_list_node *predec, void
* value);


/**
Rimuove dalla lista ll il nodo in testa e ritorna il valore puntato da tale
nodo.
*/
void *linked_list_remove_head(linked_list* ll);

/**
Rimuove dalla lista ll il nodo in coda e ritorna il valore puntato da tale
nodo.
*/
void* linked_list_remove_tail(linked_list * ll);

/**
Ritorna un puntatore al valore puntato dal nodo in input.
*/
void *linked_list_node_getvalue(linked_list_node* node);
```

```
/**
Ritorna la dimensione della lista ll.
*/
int linked_list_size(linked_list *ll);


/**
Ritorna 1 se la linked list contiene value, 0 altrimenti.
*/
int linked_list_contains(linked_list *ll, void *value);


/**
Stampa a video una rappresentazione della lista ll.
*/
void linked_list_print(linked_list *ll);

/**
Distrugge la lista ll e libera la memoria allocata per i suoi nodi. Nota che la
funzione
non libera eventuale memoria riservata per i valori puntati dai nodi della
lista.
*/
void linked_list_delete(linked_list *ll);


/**********************
  linked_list_iterator
**********************/
/**
Crea un nuovo iteratore posizionato sul primo elemento della lista ll.
*/
linked_list_iterator * linked_list_iterator_new(linked_list *ll);

/**
Ritorna 1 se l'iteratore iter ha un successivo, 0 altrimenti.
*/
int linked_list_iterator_hasnext(linked_list_iterator* iter);


/**
Muove l'iteratore un nodo avanti nella lista e ritorna il valore puntato dal
nodo
appena oltrepassato, o NULL se l'iteratore ha raggiunto la fine della lista.
*/
//void * linked_list_iterator_next(linked_list_iterator * iter);
linked_list_node * linked_list_iterator_next(linked_list_iterator * iter);

/**
Rimuove dalla lista il nodo ritornato dall'ultima occorrenza della funzione
linked_list_iterator_next.
*/
void *linked_list_iterator_remove(linked_list_iterator * iter);

/**
Ritorna il valore puntato dal nodo su cui si trova attualmente l'iteratore
iter.
```

```
*/
//void * linked_list_iterator_getvalue(linked_list_iterator *iter);

/**
Distrugge l'iteratore e libera la memoria riservata. Nota che questa operazione
non ha nessun effetto sulla lista puntata dall'iteratore.
*/
void linked_list_iterator_delete(linked_list_iterator* iter);
```