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 14-09-2021 – 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 *diretti*. In particolare, ogni nodo ha associata la lista dei vicini uscenti (vicini ai quali il nodo è connesso da archi uscenti) . Il grafo è descritto nella classe `Graph.java` (Java) e nel modulo `graph.c` (C). Il generico nodo è descritto nella classe `GraphNode` (Java) e in `struct graph_node` (C). .

Sono inoltre già disponibili le primitive di manipolazione del grafo: creazione di grafo vuoto, lista dei vicini uscenti ed entranti di ciascun nodo (quest'ultima non necessaria per lo svolgimento di questo esercizio), inserimento di un nuovo nodo, inserimento di un nuovo arco, get label/valore (stringa) di un dato nodo, cancellazione arco, cancellazione nodo (e relativi archi incidenti), cancellazione grafo, stampa grafo. Per dettagli sulle segnature di tali primitive, così come per dettagli su quali porzioni di memoria allocata vengono liberate dalle varie primitive di cancellazione, si rimanda ai file sorgente distribuiti.

In particolare, per Java si rimanda alle classi `Graph` e `GraphNode` (quest'ultima classe interna di e contenuta nel file `Graph.java`) e ai commenti contenuti in `Graph.java`. Per C si rimanda all'header `graph.h`.

Si noti che le primitive forniscono un insieme base per la manipolazione di grafi, ma i problemi proposti possono essere risolti usandone solo un sottoinsieme.

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:

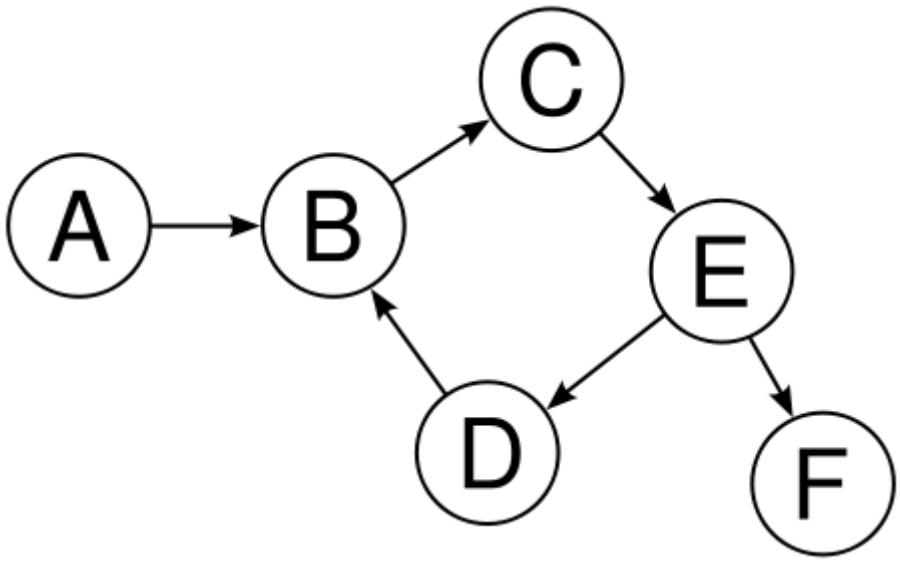

Figura 1. Esempio di grafo diretto pesato. La componente *fortemente connessa* contenente il nodo **B** include il sottoinsieme {**B**, **C**, **D**, **E**} dei nodi, mentre la componente fortemente connessa cui appartiene il nodo **A** è {**A**}.

1. Implementare la funzione/metodo `boolean cycles(Graph<V> g)` della classe `GraphServices` (o la funzione `int cycles(graph* g, graph_node* source)` del modulo `graph_services` in C) che, dato un grafo `g`, restituisce `true` (`1` in C) se il grafo `g` è un DAG (Directed Acyclic Graph) e `false` (`0` in C) altrimenti . Ad esempio, per il grafo della Figura 1 il metodo (la funzione) dovrebbe restituire `false` (`0`), in quanto è chiaramente presente un ciclo diretto che coinvolge i vertici B, C, E, D.

   Suggerimenti.

   - Si tenga presente che si chiede soltanto di verificare se il grafo sia un DAG o meno, non si chiede di calcolare un ordinamento topologico.
   - Si ricordi che per verificare la presenza di cicli in grafi diretti si può fare uso di una DFS opportuna, usando in modo appropriato il campo `state` della classe `GraphNode` (`struct graph_node` in C).

   Punteggio: [10/30]

Quesito 2: Algoritmi

1. Si consideri un grafo $G = (V, E)$ avente n vertici, *non diretto e completo* (ossia, avente un arco non diretto per ogni possibile coppia (u, v) di vertici). Si caratterizzino le proprietà degli alberi di visita DFS (Depth First Search) e BFS (Breadth First Search o visita in ampiezza) effettuati a partire da un generico vertice $s \in V$. In particolare per ciascuna delle due visite dare:

- L'altezza dell'albero di visita
- Il numero di foglie dell'albero di visita
- Il grado minimo e il grado massimo dell'albero di visita

Nota: la domanda è semplice e richiede soltanto la conoscenza delle proprietà degli algoritmi di visita. Gli studenti non si lascino impressionare dalla formulazione generica della domanda.

Punteggio: [5/30]

2. Si consideri il seguente algoritmo, descritto in codice Java:

```
static int countSubSeqs(int a[], int i, int j, int n){
        if (n == 1)
            return 1;
        if (n <= 0)
            return 0;

        int count = countSubSeqs(a, i + 1, j, n - 1) + countSubSeqs(a, i, j
- 1, n - 1);

        if (a[i] == a[j])
            count++;

        return count;
    }
static int countSubSequences(int a[]){
        return countSubSeqs(a,0,a.length-1,a.length);
}
```

   Si calcoli la complessità asintotica di caso peggiore dell'algoritmo così ottenuto in funzione della dimensione dell'input.

   Nota: gi studenti dovrebbero sapere che non è necessario comprendere esattamente *cosa fa* l'algoritmo per rispondere correttamente alla domanda.

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

   Punteggio: [5/30]

3. Si consideri il grafo non diretto e pesato nella figura sottostante.

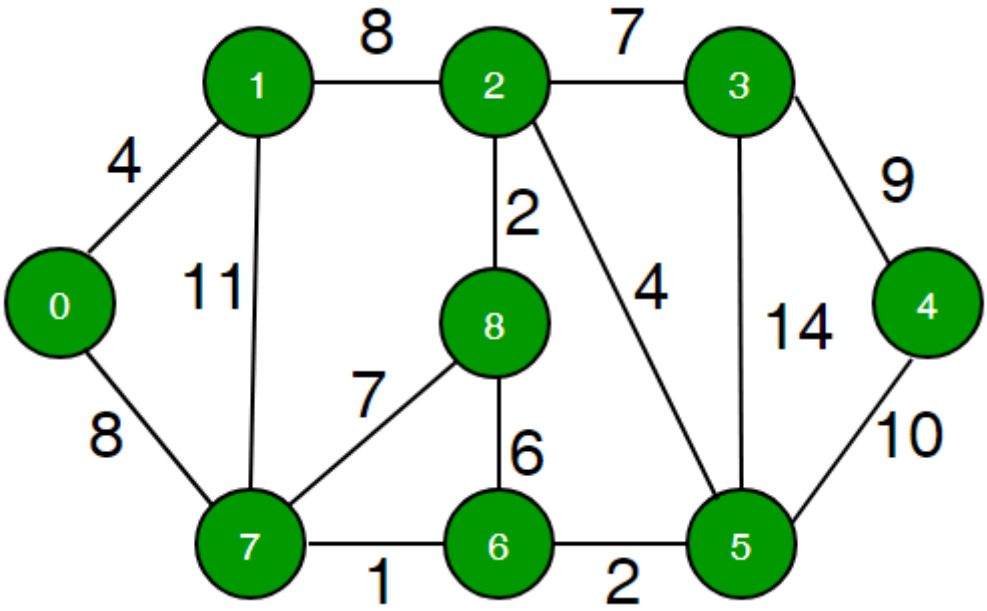

Figura 1

Si dimostri che l'arco $(5, 3)$ di peso 14 non può far parte di alcun *albero minimo ricoprente* per il grafo in figura.

Si noti che occorre dare una dimostrazione formale di questo fatto, sfruttando le proprietà degli alberi minimi ricoprenti. 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 *ordinate* ℓ_1 ed ℓ_2, contenenti rispettivamente n_1 e n_2 stringhe (l'ordinamento è ovviamente quello lessicografico). Si supponga che una stringa possa apparire *al più una volta* in ciascuna lista.

- Si proponga il più efficiente algoritmo possibile che, prese in ingresso le due liste, restituisca una lista ℓ *ordinata*, contenente le stringhe che compaiono in ℓ_1 ma non in ℓ_2. Ad esempio, se $\ell_1 = \{a, ac, b, bc, d, eh\}$ e $\ell_2 = \{a, ad, bc, eh\}$, allora $\ell = \{ac, b, d\}$.

  Nota: è possibile e preferibile (ma non obbligatorio) non usare ulteriori strutture dati oltre alle liste.

  Punteggio: [4/30]

- Si calcoli il costo asintotico dell'algoritmo proposto.

  Punteggio: [3/30]

Occorre descrivere chiaramente l'algoritmo proposto (pseudo-codice), indicando 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, nonché dalla semplicità dell'algoritmo proposto.

Appendice: interfacce dei moduli/classi per il quesito 1

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

```
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

    // 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`.

```
// 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);
```

Metodi potenzialmente utili della classe LinkedList.

```
// 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.

```
public class GraphServices<V>{

  public static <V> boolean cycles(Graph<V> g) {
        /* DA IMPLEMENTARE */
  }

}
```

Interfacce C

graph.h (solo tipi principali)

```
#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
} graph_node;
```

Scheletro del modulo C graph_services

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

```
#include "graph.h"

int cycles(graph* g) {
        /* DA IMPLEMENTARE */
}
```

linked_list.h (solo parte)

```
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);
```