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 09-06-2023 – a.a. 2022-23 – Tempo: 2 ore – somma punti: 32 - Compito B

Istruzioni

Lanciare la macchina virtuale Oracle VirtualBox e lavorare all'interno della cartella `Esame`. Per prima cosa compilare il file `studente.txt` con le informazioni richieste. In particolare, cognome, nome, matricola, email, esame (vecchio o nuovo), linguaggio in cui si svolge l'esercizio 1 (Java o C). Per quanto riguarda il campo esame (vecchio o nuovo), possono optare per il vecchio esame gli studenti che nel periodo che va dal 2014-15 al 2017-18 (estremi inclusi) sono stati iscritti al II anno.

Nota bene. Per ritirarsi è sufficiente rinominare il file `studente.txt` in `studenteritirato.txt`, *senza* cancellarlo.

Come procedere. Nella cartella `Esame` trovate i) il testo del compito, ii) il file `studente.txt` sopra citato, iii) due sottocartelle `C-aux` e `java-aux`, contenenti il codice di supporto e lo scheletro delle soluzioni per il quesito di programmazione (quesito 1). Svolgere il compito nel modo seguente:

- Per il quesito 1, Alla fine la cartella `C-aux` (o `java-aux`, a seconda del linguaggio usato) dovrà contenere le implementazioni delle soluzioni al quesito 1, ottenute completando i relativi metodi (o le relative funzioni) contenuti nei file forniti dai docenti; si ricorda ancora una volta che la cartella `java-aux` (o `c-aux`) deve trovarsi all'interno della cartella `Esame`.
- Per i quesiti 2 e 3, creare due file `probl2.txt` e `probl3.txt` contenenti, rispettivamente, gli svolgimenti dei problemi proposti nei quesiti 2 e 3; i tre file devono trovarsi nella cartella `Esame`. È possibile integrare i contenuti di tali file con eventuale materiale cartaceo *unicamente per disegni, grafici, tabelle, calcoli*.

Attenzione: i simboli < e > usati nei nomi dei file fanno parte del linguaggio dei metadati e non debbono essere inclusi nei nomi reali.

Avviso importante 1. Per svolgere il quesito di programmazione *si consiglia caldamente l'uso di un editor di testo (ad esempio Geany) e la compilazione a riga di comando*, strumenti più che sufficienti per lo svolgimento del compito. La macchina virtuale mette a disposizione diversi ambienti di sviluppo, quali Eclipse e Javabeans. *Gli studenti che li usano lo fanno a proprio rischio*. In particolare, *se ne sconsiglia l'uso qualora non se ne abbia il pieno controllo e certamente se non si è già in grado di sviluppare servendosi unicamente di un editor e della compilazione a riga di comando*. Qualora lo studente intenda comunque usare un ambiente di sviluppo integrato, si raccomanda di controllare che i file vengano effettivamente salvati nella cartella `java-aux` (o `c-aux`, a seconda del linguaggio usato), in quanto vi è il rischio concreto di perdere il proprio lavoro. In generale, file salvati esternamente alla cartella `Esame` andranno persi al termine della prova e quindi non saranno corretti.

Avviso importante 2. Il codice *deve compilare* correttamente (warning possono andare, ma niente errori di compilazione), altrimenti riceverà 0 punti. Si raccomanda quindi di scrivere il programma in modo incrementale, ricompilando il tutto di volta in volta.

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) e dei vicini entranti (nodi che hanno archi diretti verso il nodo considerato). 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. che corrisponde al grafo usato nel programma di prova (`Driver.java` o `driver`):

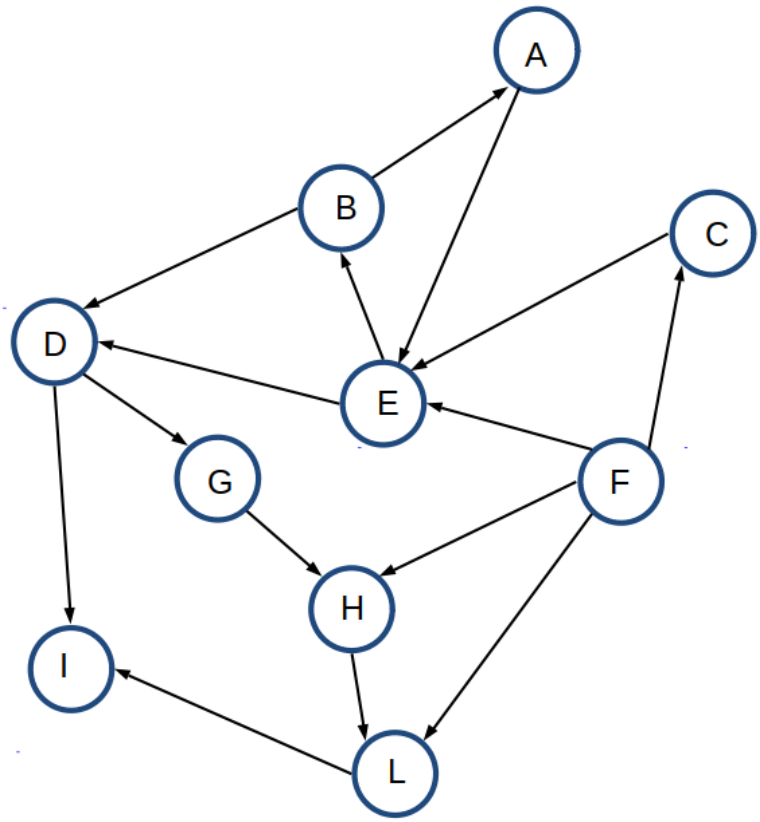


Figura 1. Esempio di grafo diretto non pesato. Le distanze minime dal nodo **A** sono [**A**:0, **B**:2, **E**:1, **D**:2, **G**:3, **H**: 4, **I**: 3, **L**: 5], mentre le distanze minime dal nodo **H** sono [**H**:0, **L**:1, **I**:2]. Si noti che in entrambi i casi considerati, non tutti i nodi del grafo sono raggiungibili dal nodo di partenza.

1. Implementare la funzione/metodo `static <V> void distances(Graph<V> g)` della classe `GraphServices` (o `void distances(graph* g)` del modulo `graph_services` in C) che, dato un grafo `g` stampa, *per ogni* nodo `source` di `g`, l'elenco (eventualmente vuoto) di tutti i nodi *raggiungibili* da `source` e, per ognuno di questi, la sua *distanza minima* da `source`, intesa come il *numero minimo di archi da attraversare* (l'ordine di stampa non è importante). Per un esempio, si faccia riferimento alla Figura 1.

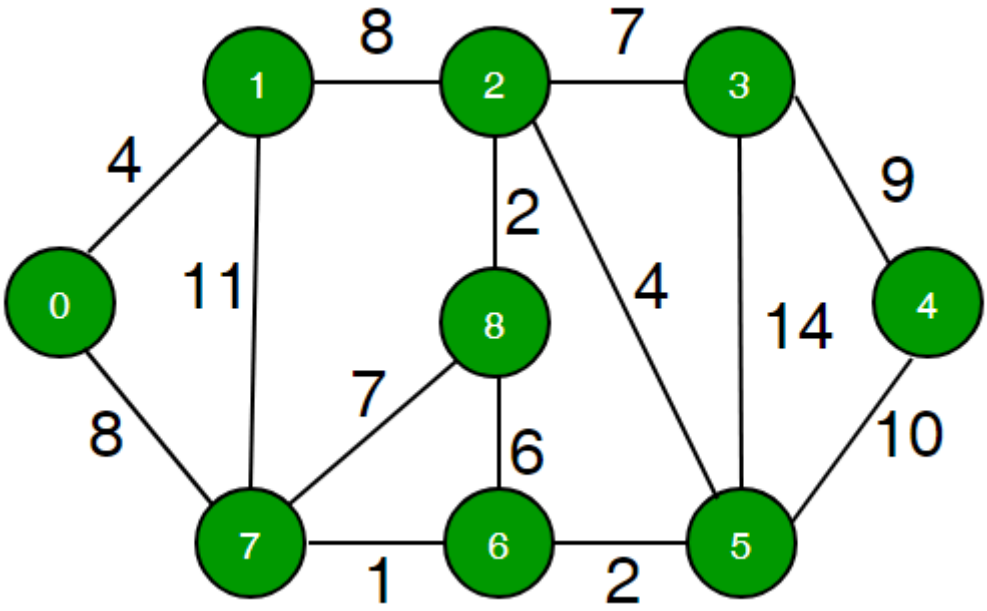
   La valutazione delle risposte degli studenti terrà conto dell'efficienza delle soluzioni proposte.

   Nota: Per tenere traccia della distanza dei vari nodi dalla sorgente si consiglia di usare il campo `int timestamp` della classe `Graph.Node` (`int timestamp` di `struct graph_node`). Si consiglia di definire un metodo/funzione ausiliario che, dato `g` e un generico nodo `source`, restituisce (o modifica) una lista contenente i nodi raggiungibili da `source`.

   Punteggio: [10/30]

Quesito 2: Algoritmi

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

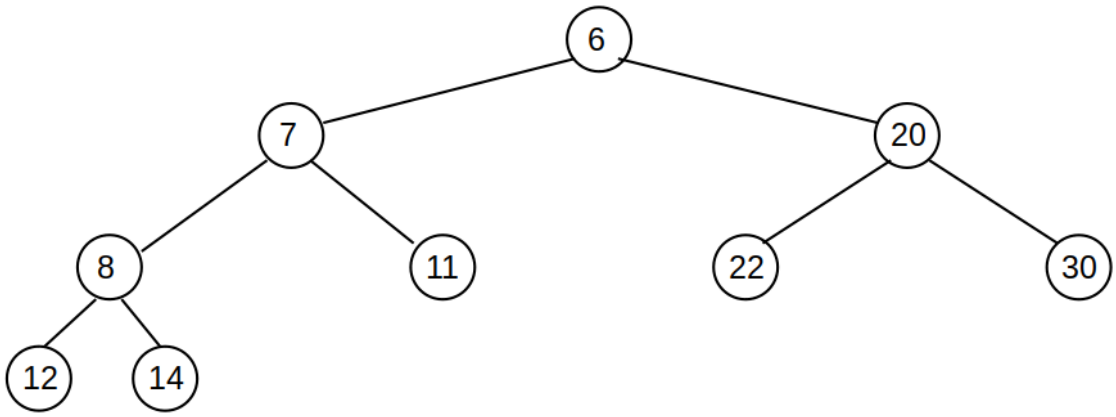


   Si mostri l'evoluzione dell'algoritmo di Kruskal per il grafo in figura. In particolare, per ogni iterazione *i* dell'algoritmo occorre specificare i) l'insieme C delle partizioni (o cluster) presenti *all'inizio* dell'iterazione *i*-esima e ii) l'arco che viene aggiunto all'albero minimo ricoprente T nel corso dell *i*-esima iterazione.

Iterazione	C	Arco aggiunto
1		
2		
....	...	...

   Punteggio: [8/30]

2. Si consideri il seguente heap minimale:



Si descriva l'evoluzione dell'heap a seguito della *rimozione* del nodo avente chiave 7. In particolare, occorre i) descrivere ogni passo di aggiornamento dell'heap a seguito della rimozione del nodo e del ripristino della condizione di heap, ii) descrivere l'albero finale risultante.

Oltre che alla correttezza, la risposta sarà valutata in base alla completezza e alla chiarezza espositiva. Risposte vaghe o generiche subiranno penalizzazioni. Non occorre scrivere tanto ma scrivere bene.

Punteggio: [7/30]

Quesito 3:

1. Si consideri il seguente algoritmo per l'ordinamento di una lista inizialmente disordinata, contenente chiavi appartenenti a un universo totalmente ordinato (ad esempio, chiavi intere):

PQ sort

```
// Usiamo una coda di priorità Q
Input: <lista non ordinata S>
Output: <lista a ordinata rispetto alle chiavi degli elementi in S>
while (!S.isEmpty):
    Q.insert(S.remove())
    // Assumiamo che gli elementi di S siano coppie (k, v)
    // S.remove() rimuove da S l'elemento in testa
while (!Q.isEmpty):
    a.add(Q.removeMin()) //add aggiunge alla fine della lista
return a // a è la lista ordinata delle chiavi inizialmente presenti in S
```

Si supponga che la coda di priorità Q (così come la lista S) sia realizzata usando una *lista non ordinata, accessibile a costo costante soltanto in testa o coda*. In particolare, `Q.insert(x)` aggiunge `x` alla fine della lista che implementa Q, mentre `Q.remove()` rimuove dalla lista e restituisce l'elemento in testa. `Q.removeMin()` estrae da `Q` l'elemento avente chiave minima e il suo costo di caso peggiore è quello determinato dall'implementazione di Q descritta sopra. Supponendo che S contenga n chiavi, si calcoli la complessità asintotica di caso peggiore dell'algoritmo PQ sort.

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

Punteggio: [4/30]

2. Si consideri il problema dell'ordinamento topologico in grafi diretti (Topological Sort).

   i) Si definisca formalmente il problema, introducendo e definendo la notazione eventualmente necessaria.

   ii) Si dimostri che se un grafo diretto ammette un ordinamento topologico allora è privo di cicli (diretti).

   Il punteggio dipenderà da correttezza formale e chiarezza espositiva.

   Punteggio: [3/30]

Occorre descrivere chiaramente l'algoritmo proposto e indicare eventuali strutture dati ausiliarie usate. Occorre dare un'argomentazione quantitativa e rigorosa 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 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.

Alcuni campi della classe Node

```
    /* Classe interna che descrive il generico nodo del grafo, con liste dei
vicini uscenti ed entranti */
    public static class Node<V> implements Cloneable {
        public enum Status {UNEXPLORED, EXPLORED, EXPLORING}

        protected V value;
        protected LinkedList<Node<V>> outEdges;
        protected LinkedList<Node<V>> inEdges;

        protected Status state; // tiene traccia dello stato di esplorazione
        protected int map; // utile in partition union e find
        protected int timestamp; // utile per associare valori interi ai
vertici
        protected int dist; // utile per memorizzare distanze in algoritmi per
cammini minimi

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

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

Alcuni 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`.

```
    /**
     * Restituisce una collezione contenente i nodi del grafo
     *
     */
    public List<Node<V>> getNodes() {
        // Implementazione
    }

    /**
     * Restituisce una lista contenente i vicini uscenti del nodo dato
     *
     */
    public List<Node<V>> getOutNeighbors(Node<V> n){
        // Implementazione
    }

    /**
     * Restituisce una lista contenente i vicini entranti del nodo dato
     *
     */
    public List<Node<V>> getInNeighbors(Node<V> n){
        // Implementazione
    }
```

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 {
    //Altri campi

    public static <V> void reachable(Graph<V> g) {

        // DA IMPLEMENTARE
    }
    // Possibili altri metodi (se necessari)
}
```

Interfacce C

graph.h (solo porzioni principali del file)

```c
typedef enum { UNEXPLORED, EXPLORED, EXPLORING } STATUS;

typedef struct graph_node {
    void *value;
    linked_list *out_edges;
    linked_list *in_edges;

    STATUS status;
    int timestamp;
} graph_node;

typedef struct graph_prop {
    int n_vertices;
    int n_edges;
} graph_prop;

typedef struct graph {
    linked_list* nodes;
    graph_prop* properties;
} graph;

/**
Returns the list of vertices of g.
*/
linked_list * graph_get_nodes(graph * g);

/**
Returns a list containing all the outgoing edges from n in g.
*/
linked_list * graph_get_out_neighbors(graph_node * n);

/**
Returns a list containing all the ingoing edges from n in g.
*/
linked_list * graph_get_in_neighbors(graph_node * n);
```

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_services.h"

#include <stdio.h>
#include <stdlib.h>

void distances(graph* g) {
    /**
        Implementazione
    */
```

```
}

/**
Altre funzioni (se necessarie)
*/
```

linked_list.h (solo parte)

```
/**
Le strutture di seguito riportate sono definite all'inizio di linked_list.c
Si noti che linke_list_iterator mette a disposizione funzioni per
l'aggiornamento
e lo scorrimento di liste
*/
typedef struct linked_list_node linked_list_node;
typedef struct linked_list linked_list;
typedef struct linked_list_iterator linked_list_iterator;
```