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)

Prova del 23-06-2020 Compito B – a.a. 2019-20 – Tempo: 2 ore e 30 minuti – somma punti: 32

Istruzioni

Occorre rispondere a partire dal testo delle domande.

Avviso importante. La risposta ai quesiti 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.

Quesito 1: Progetto algoritmi C/Java [soglia minima: 5/30]

In questo problema si fa riferimento a grafi semplici (possibilmente) diretti e *non pesati*. I grafi sono rappresentati attraverso liste di adiacenza. A ciascun nodo u è dunque associata una lista collegata contenente $grado(u)$ elementi, ciascuno dei quali è il riferimento a uno dei vicini (uscenti) di 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. 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 alcuni di questi potrebbero non essere necessari.

Ciò premesso, rispondere al seguenti punto:

1. Implementare il metodo `public LinkedList<V> viciniEntroK(Graph<V> g, GraphNode<V> s, int k)` (funzione `linked_list * vicini_entro_k(graph *g, graph_node *s, int k)` in C) della classe `GraphServices` (modulo `graph_services` in C) che dati un grafo, un nodo sorgente e un intero k, restituisce la lista dei valori associati ai nodi che si trovano a distanza *al più k dalla sorgente in termini di numero minimo di archi attraversati*. Ad esempio, per il grafo in Fig. 1, qualora $k = 1$ e il nodo sorgente fosse 7 il metodo/funzione dovrebbe restituire la lista corrispondente ai seguenti nodi (l'ordine non è importante):

```
7 11 8
```

Se invece $k = 3$ e il nodo sorgente fosse ancora 7 il metodo/funzione dovrebbe restituire la lista corrispondente ai seguenti nodi:

```
7 8 11 9 2 10
```

Il costo dell'algoritmo deve essere asintoticamente ottimale. In particolare, parte del punteggio sarà assegnato a soluzioni che non esplorarano nodi inutilmente. Ad esempio, se avessimo $k = 10$ e il grafo contenesse 10^9 nodi, vorremmo che il numero di nodi esplorati fosse nell'ordine delle decine o delle centinaia e non del miliardo.

Punteggio: [10/30]

Suggerimento. Si suggerisce di usare il campo intero `timestamp` della classe `GraphNode` (omonimo campo di `struct graph_node` in C). Si ricordi anche che per calcolare percorsi minimi su grafi non pesati è sufficiente un algoritmo molto più semplice di quello di Dijkstra o equivalente.

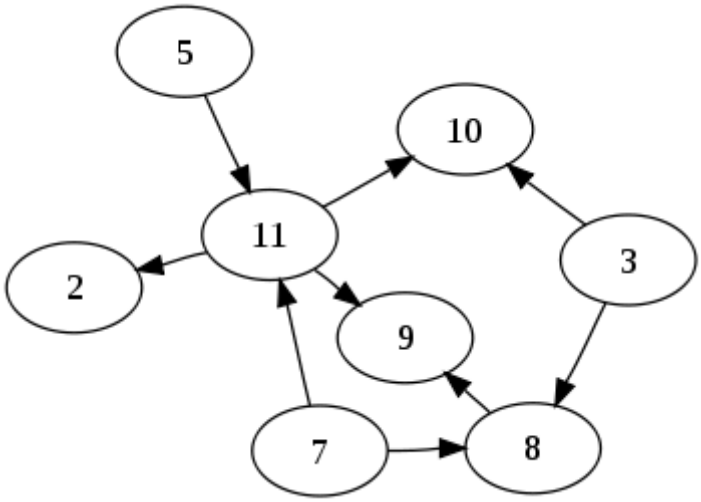


Fig. 1. Grafo diretto con più componenti connesse semplici.

Quesito 3: Algoritmi

1. Si consideri una tabella hash a *indirizzamento aperto* di dimensione $N = 11$ e a chiavi intere. Si supponga che la funzione di compressione sia $f(x) = x \mod N$ e che le collisioni siano risolte mediante *esplorazione lineare*. Dare un esempio di aggregazione (clustering) *primaria*, generata dall'inserimento in successione di 4 coppie (chiave, valore).

   Occorre spiegare cos'è l'aggregazione e perché si verifica nell'esempio proposto.

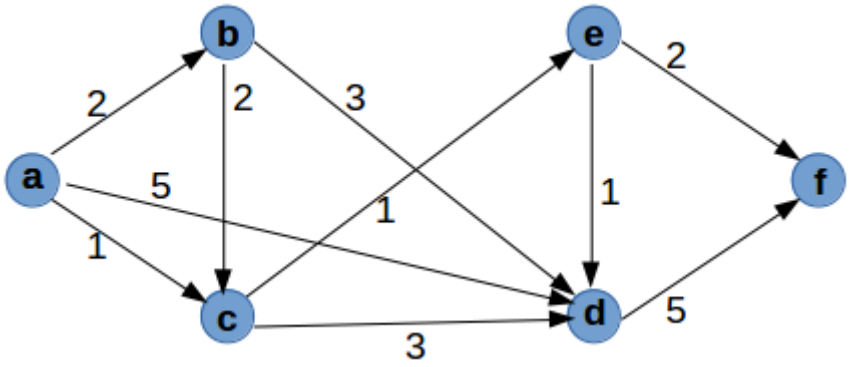
   Punteggio: [5/30]

2. Si consideri una *coda di priorità minimale* realizzata con una *lista non ordinata*, accessibile *soltanto* a partire dalla testa. Supponendo che la coda di priorità inizialmente contenga n coppie (chiave, valore), si valuti il costo di n operazioni consecutive di tipo `removeMin()`, ossia di *rimozione* dalla coda di priorità dell'elemento avente chiave minima tra quelle presenti.

   Occorre dare un'argomentazione quantitativa, giustificando adeguatamente la risposta.

   Punteggio: [5/30]

3. Si consideri il grafo diretto e pesato in Figura 2.



Si mostri l'evoluzione dell'algoritmo di Dijkstra per il grafo in Figura 2 a partire dal nodo **b**. In particolare, per ogni iterazione *i* dell'algoritmo occorre specificare i) il sotto-insieme Q dei *nodi* del grafo già estratti dalla coda di priorità usata dall'algoritmo e ii) la distanza minima *stimata* di ciascun nodo dalla sorgente **b**.

Punteggio: [5/30]

Quesito 4:

Una società pubblica deve realizzare una dorsale autostradale che garantisca il collegamento tra n località principali. Il costo per collegare le località u e v con un tratto autostradale si può assumere pari alla distanza chilometrica w_{uv} tra di esse. Ai fini dell'esercizio si può supporre che l'input corrisponda a un grafo stradale (completo) $G = (V, E)$ non diretto e pesato (con i pesi w_{uv}). Ciò premesso si risponda alle domande seguenti:

- Descrivere un algoritmo (pseudo-codice) che restituisca le coppie di località da connettere mediante un tratto autostradale in modo tale che: 1) ogni località sia raggiungibile da ogni altra località mediante una successione di una o più tratte autostradali; 2) il costo complessivo delle tratte autostradali da realizzare sia minimo. Per essere precisi, il costo complessivo è pari a $Costo(C) = \sum_{(u,v)\in C} w_{uv}$, dove C è l'insieme delle coppie tra ciascuna delle quali verrà realizzato un tratto autostradale, e si vuole scegliere C in modo da minimizzare $Costo(C)$.

  Punteggio: [4/30]

- Descrivere un algoritmo (pseudo-codice) che, preso in input un grafo stradale $G = (V, E)$, determini e restituisca l'arco $e \in E$ tale che, in caso di blocco della circolazione sul tratto autostradale corrispondente ad e , il valore minimo di $Costo(C)$ calcolato sul grafo $\hat{G} = (V, E - \{e\})$ sia massimo. Nel caso e non sia unico, l'algoritmo deve determinare e restituire l'elenco di tutti tali archi. *Anche se non necessario, si può supporre che ogni taglio del grafo G consista di almeno 2 archi*.

  Punteggio: [3/30]

Nota bene: i) La descrizione può usare primitive non elementari, ad esempio corrispondenti ad algoritmi noti studiati nel corso. In tal caso, è sufficiente specificare chiaramente di quali algoritmi si tratta; ii) lo pseudo-codice privo di indentazione sarà penalizzato.

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

```
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 nodi adiacenti

    // 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 del nodo n
public List<GraphNode<V>> getNeighbors(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.

```java
public class GraphServices<V>{

  public LinkedList<V> viciniEntroK(Graph<V> g, GraphNode<V> s, int k) {
        /* 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; // naturale letto da file
    linked_list* out_edges;  // lista di adiacenza
} graph_node;
```

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);
```

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"

linked_list vicini_entro_k(graph *g, graph_node *s, int k) {
        /* DA IMPLEMENTARE */
}
```