# Fondamenti di Programmazione 2

Laboratorio

25 Novembre 2022

**Utilizzare le classi `Grafo`, `GrafoNonOrientato` viste a lezione oppure implementare le proprie, che espongano (almeno) l'interfaccia:**

```
public:
// Restituisce il numero di nodi del grafo
unsigned n() const;

// Restituisce il numero di archi del grafo
unsigned m() const;

// Restituisce true se e solo se
// l'arco (i,j) è nel grafo
bool operator(unsigned i, unsigned j) const;

// Inserisce (b=True)/Rimuove (b=False)
// l'arco (i,j) nel/dal grafo
void operator(unsigned i, unsigned j, bool b);
```

## Esercizio 1

Implementare i seguenti metodi per la classe `Grafo` e `GrafoNonOrientato`:

```
// Restituisce il grado del nodo `x`
unsigned grado(unsigned x) const;

// Restituisce il grado dei nodi
// i-esimo valore è il grado del nodo `i`
vector<unsigned> gradi() const;

// Restituisce il vicinato del nodo `x`
vector<unsigned> vicinato(unsigned x) const;
```

## Esercizio 2

Scrivere una funzione con segnatura:

```
void stampaGrafo(const Grafo& g)
```

che stampi:

- il numero di nodi di `g`
- il grado di ogni nodo in `g`
- il numero di archi di `g`
- la lista di tutti gli archi in `g`

## Esercizio 3

Scrivere una funzione con segnatura:

```
pair<unsigned, vector<unsigned>> getNodiConGradoMassimo(const GrafoNonOrientato& g)
```

che restituisca una coppia contenente il grado massimo in `g` e tutti i nodi con grado massimo.

## Esercizio 4

Scrivere una funzione con segnatura:

```
bool stessoNumeroNodiStessoGrado(const GrafoNonOrientato& g, const GrafoNonOrientato& h)
```

che restituisca `true` se e solo i grafi `g` e `h`, per ogni possibile grado, hanno lo stesso numero di nodi.

## Esercizio 5

Scrivere una funzione con segnatura:

```
bool almenoUnNodoAdiacenteATutti(const GrafoNonOrientato& g)
```

che restituisca `true` se e solo se in `g` esiste almeno un nodo connesso a tutti gli altri nodi.

## Esercizio 6

Scrivere una funzione con segnatura:

```
pair<unsigned,unsigned> getCoppiaPiuAdiacenti(const GrafoNonOrientato& g)
```

che restituisca la coppia di nodi che hanno il maggior numero di vicini in comune in `g`.

## Esercizio 7

Scrivere una funzione con segnatura:

```
bool connesso(const GrafoNonOrientato& g)
```

che restituisca `true` se e solo se per ogni coppia $(u, v)$ di nodi in `g` esiste un percorso che connette $u$ a $v$, e viceversa.

## Esercizio 8

Scrivere una funzione con segnatura:

```
bool inUnCiclo(const GrafoNonOrientato& g, unsigned n)
```

che restituisca `true` se e solo se il nodo `n` di `g` fa parte di un ciclo di `g`.

## Esercizio 9

Scrivere una funzione con segnatura:

```
bool proprieta_1(const Grafo& g, unsigned k)
```

che restituisce `true` se e solo se in `g` ci sono almeno `k` coppie di nodi che hanno almeno un nodo adiacente in comune. **La coppia** $(i, j)$ **va considerata identica alla coppia** $(j, i)$**.**

## Esercizio 10

Scrivere una funzione con segnatura:

```
bool proprieta_2(const Grafo& g, vector<unsigned> pesi)
```

che restituisca `true` se e solo se per ogni nodo di `g` il prodotto tra il proprio grado e il proprio peso, che per il nodo `i`-esimo di `g` è contenuto in `pesi[i]`, è maggiore della somma dei pesi dei nodi ad esso adiacenti.

## Esercizio 11

L'*algoritmo di Dijkstra* è un algoritmo per il calcolo di cammini minimi in grafi (orientato o non orientato) con pesi non negativi. Possiamo utilizzarlo per il calcolo del cammino minimo da un nodo del grafo (`src`) verso tutti gli altri nodi del grafo (raggiungibili da esso).

L'algoritmo prende in input:

- Un grafo (orientato o non orientato), `graph`
- Una matrice di costi `weights`, dove `weights[i][j]` rappresenta il costo dell'arco `i->j`.
- Un nodo iniziale `src`

L'output dell'algoritmo è:

- Un vettore `cost`, dove `cost[i]` rappresenta il costo (somma del costo degli archi) del percorso minimo tra `src` e `i`
- Un vettore `prev` dove `prev[i] = j`. Il predecessore del nodo `i` nel percorso a costo minimo che connette `src` a `i` è `j`.

```
def dijkstra_all_nodes_single_source(graph, weights, src):
    cost = [None] * g.num_nodes
    cost[src] = 0

    prev = [None] * g.num_nodes

    # Q è una coda con priorità che gestisce coppie (nodo, costo).
    # durante l'esecuzione dell'algoritmo, il primo nodo della coda Q
    # è il nodo (correntemente) raggiungibile a minor costo
    Q = priority_queue()

    # inizializzata con la sorgente della visita
    Q.push((src, 0))

    while not Q.empty():
        # estraggo il nodo a minimo costo dalla coda con priorità
        top_node, top_cost = Q.pop()

        # per ogni vicino del nodo estratto
        for v in g.vicinato(top_node):
            # se è raggiungibile ad un costo minore
            # di quello corrente
            node_cost = cost[top_node] + weights[top_node][v]
            if cost[v] is None or node_cost < cost[v]:
                # aggiorno cost[v]
                cost[v] = node_cost
                parent[v] = top_node
```

```
    # restituisco `cost` e `prev`
    return costo, prev
```

Scrivere una funzione C++ che implementi l'algoritmo di Dijkstra.

**Suggerimento:** Il codice è molto simile ad una visita DFS/BFS, ad eccezione di qualche struttura dati in più (l'analogo dei vettori `cost` e `prev` nello pseudocodice) e l'utilizzo di una coda con priorità invece di uno `stack`/`queue`. Per implementare la coda con priorità, può tornare utile `std::priority_queue<NodeCost>`, dove `NodeCost` è una classe/struct ausiliaria (munita di `operator<`) che gestisce una coppia nodo-costo.