Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

L'esercitazione di questa settimana prevede di sviluppare alcuni programmi che elaborano grafi non orientati; gli esercizi si basano principalmente sull'uso delle visite in profondità e in ampiezza.

Per semplicità assumeremo sempre che i nodi del grafo siano rappresentati dagli interi 0, 1, ... n, dove $n-1$ è il numero dei vertici del grafo.

1 Rappresentazione di grafi

Innanzitutto bisogna predisporre alcune funzioni fondamentali che permettono di creare un grafo, aggiungere nodi e archi. L'interfaccia potrebbe essere di questo tipo:

```
typedef struct graph *Graph;

/* crea un grafo con n nodi */
Graph graph_new( int n );

/* distrugge il grafo g */
void graph_destroy( Graph g );

/* inserisce l'arco (v,w) nel grafo g */
void graph_edgeinsert( Graph g, int v, int w );

/* legge da standard input un grafo (specificare il formato!!) */
Graph graph_read( void );

/* stampa su standard output un grafo (specificare il formato!!) */
void graph_print( Graph g );
```

Implementate la precedente interfaccia usando le due rappresentazioni standard dei grafi.

Matrice di adiacenza

In questo caso potete definire il tipo seguente:

```
struct graph {
  int n, m;           /* n = num vertici, m = num lati */
  int **A; /* matrice di adiacenza */
};
```

Nella definizione della funzione `graph_new( int n )`, andrà definita una variabile `Graph g`, allocato lo spazio necessario per una **`struct`** `graph`, quindi allocato lo spazio per per gli `n` vertici:

```
  int i; Graph g;

  /* alloca lo spazio per una struct graph */
```

```
  g = malloc( sizoef(struct graph) );

  /* alloca lo spazio per un array di n puntatori a int */
  g -> A = calloc( n, sizeof(int *) );

  for ( i = 0; i < n; i++ )
    /* alloca lo spazio per l'array A[i] di n interi, inizialmente nulli */
    g -> A[i] = calloc( n, sizeof( int ) );
  g -> n = n;
```

Array di liste di adiacenza

In questo caso invece potete definire i tipi seguenti:

```
struct listnode {
  struct listnode *next;
  int v;
};

struct graph {
  int n, m;
  struct listnode **A;
}
```

e allocare lo spazio per un array di n puntatori a liste di adiacenza (o, meglio, puntatori a teste di liste di adiacenza):

```
  g -> A = calloc( n, sizeof( struct listnode* ) );
```

2 Visite di grafi

Implementate la visita DFS (depth-first search, ovvero visita in profondità) e la visita BFS (breadth-first search, ovvero visita in ampiezza) con entrambe le rappresentazioni.

Ricordate che per la visita DFS potete usare la ricorsione oppure una pila di supporto, mentre per la BFS potete usare una coda.

3 Proprietà di grafi

Scrivete delle funzioni o dei programmi client che permettano di svolgere le seguenti operazioni su grafi. Tali operazioni possono essere efficientemente eseguite sfruttando le visite di grafi implementate in precedenza.

Attenzione: i prototipi delle funzioni potranno richiedere altri argomenti oltre a quelli indicati nelle specifiche delle operazioni!

1. **gen (*p*)** genera un grafo casuale, a partire dalla probabilità p compresa tra 0 e 1 (inclusi).
   Il modello matematico di riferimento è il seguente: si considerano tutti i possibili archi includendoli nel grafo con probabilità p. Più esplicitamente, per ogni possibile coppia di vertici, si genera un numero reale compreso tra 0 e 1; se questo è minore di p si inserisce l'arco, altrimenti non lo si inserisce.
   NB: potete usare questa operazione per generare grafi su cui testare la correttezza dei vostri programmi!

2. **degree** (*v*) calcola il grado del vertice v.
   Si ricorda che il *grado* di un vertice è definito come il numero di vertici ad esso adiacenti.

3. **path** (*v*, *w*) testa l'esistenza di un cammino semplice che collega i vertici *v* e *w*.
   Si ricorda che un cammino si dice *semplice* quando attraversa ogni vertice al più una volta. il numero di vertici ad esso adiacenti.

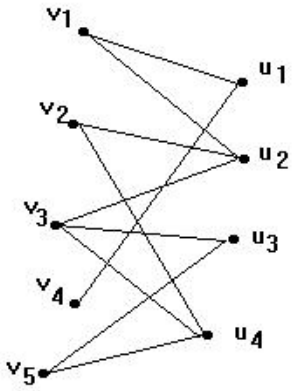
4. **ccc** conta il numero di componenti connesse di un grafo.
   Si ricorda che si chiama *componente connessa* di un grafo ogni insieme massimale di vertici connessi tra loro da un cammino.

5. **cc** (*v*) stampa l'elenco dei vertici della componente fortemente connessa contenente *v*;

6. **span** (*v*) calcola uno spanning tree con radice *v* e lo stampa nella rappresentazione a sommario;
   Si ricorda che si definisce *spanning tree* (in italiano, *albero di copertura*) un albero che ha per nodi tutti e soli i vertici del grafo. Osservate che per ottenere uno spanning tree con radice *v* è sufficiente eseguire una visita della componente connessa contenente *v* stampando ad ogni passo l'arco attraversato. Che tipo di visita si deve eseguire per avere la garanzia di ottenere uno spanning tree di altezza minimale?

7. **shortestpath** (*v*, *w*) calcola il cammino più breve che collega i vertici *v* e *w*.

8. **twocolor** testa se il grafo è bicolorabile.
   Si ricorda che un grafo si dice *bicolorabile* quando è possibile assegnare ad ogni vertice del grafo uno dei due colori bianco o nero in modo che due nodi vicini abbiano sempre colori diversi. Quando un grafo è bicolorabile, si dice anche che è *bipartito*. Ad esempio, il grafo nell'illustrazione è bipartito (basta colorare i vertici v_i di bianco e i vertici u_i di nero).



   Osservate che per verificare questa proprietà del grafo è sufficiente eseguire una visita in profondità, assegnando colori alternati ai vertici che si visitano man mano.

9. **oddcycles** testa se il grafo contiene cicli di lunghezza dispari.
   Si ricorda che un *ciclo* è un cammino che parte e finisce nello stesso vertice. Prima di implementare questa operazione, osservate quale relazione c'è tra questa proprietà e la precedente!