Strutture Dati, Algoritmi e Complessità

Esercitazione del 16 Aprile 2024

Scopo di questa esercitazione è realizzare una struttura dati per rappresentare un grafo *non diretto* e implementare una visita DFS. Nel codice della soluzione verrano usati i valori `{UNEXPLORED, EXPLORING, EXPLORED}` (tipo enumerabile) come corrispondenti agli attributi Bianco, Grigio e Nero usati sul libro di testo per descrivere lo stato dei nodi durante una visita.

Task 1. Rappresentazione di un grafo

Un grafo è un TDA che contiene un insieme V di valori, chiamati nodi, e un insieme E di coppie di nodi, chiamati archi. La generica coppia $(u, v) \in E$ rappresenta un arco che può essere percorso in entrambe le direzioni, da u a v e viceversa. Ogni nodo ha associato un valore di tipo generico (se in C, il valore è un `void *`). Per mantenere traccia degli archi in un grafo, sono possibili diverse implementazioni. In questa esercitazione, è consigliato l'uso di liste di adiacenza: ogni nodo mantiene una lista dei suoi nodi vicini. Per chi sviluppa l'esercizio in C, il codice ausiliario fornisce una semplice implementazione di una linked list. Per chi sviluppa l'esercizio in Java, si può far uso di una qualunque implementazione di una struttura
collegata fornita dal Java Collection Framework (ad esempio, `LinkedList`).

Task 1

Specifiche. Scrivere un modulo C con intestazione `graph.h` (rispettivamente una classe generica Java `Graph.java`) contenente le seguenti funzioni (metodi di classe se Java):

- una funzione/costruttore che construisce un grafo vuoto:

```
graph * graph_new();

public Graph();
```

- una funzione/metodo che restituisce i nodi presenti nel grafo:

```
linked_list * graph_get_nodes( graph * g );

public List <GraphNode<V>> getNodes();
```

  La lista collegata che viene restituita contiene i nodi del grafo (se C `graph_node`, se Java `GraphNode`).

- una funzione/metodo che restituisce la lista dei nodi vicini di un nodo:

```
linked_list * graph_get_neighbors(graph * g , graph_node * n);

public List <GraphNode<V >> getNeighbors(GraphNode <V> n);
```

- una funzione/metodo che aggiunge un nodo al grafo:

```
graph_node * graph_add_node(graph * g , void * value);

public GraphNode <V> addNode( V value);
```

e restituisce il nodo creato.

- una funzione/metodo che aggiunge un arco al grafo:

```
void graph_add_edge(graph * g , graph_node * v1 , graph_node * v2);

public void addEdge(GraphNode <V> v1 , GraphNode <V> v2);
```

Suggerimento: la funzione deve inserire un arco navigabile in entrambi i versi.

- una funzione/metodo che restituisce il valore associato a un nodo:

```
void * graph_get_node_value(graph_node * n);

public V getNodeValue(GraphNode <V> n);
```

- una funzione/metodo che elimina un arco dal grafo:

```
void graph_remove_edge(graph * g , graph_node * v1 , graph_node * v2);

public void removeEdge(GraphNode <V> v1 , GraphNode <V> v2);
```

- una funzione/metodo che elimina un nodo dal grafo (e tutti gli archi ad esso incidenti):

```
void graph_remove_node(graph * g , graph_node * v);

public void removeNode(GraphNode <V> v);
```

- (solo in C) una funzione che elimina il grafo:

```
void graph_delete(graph * h);
```

Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `graph`.

Task 2. Input/Output di un Grafo

In questo esercizio, occorre implementare una procedura di input/output testuale di un grafo, secondo una rappresentazione standard. In particolare, useremo una rappresentazione per lista di archi. Ad esempio: lista di archi, preceduta da una riga contenente numero di nodi e numero di archi.

5 8

1 2

1 4

2 3

3 4

3 5

4 5

4 2

3 1

In questo caso, abbiamo 5 nodi e 8 archi. Questi ultimi sono gli archi {1, 2}, {1, 4}, {2, 3} ecc.

Specifiche. Estendere il modulo C con intestazione `graph.h` (rispettivamente la classe generica Java `Graph.java`) definito nell'esercizio precedente in modo che includa:

- una funzione/metodo che legge un grafo da file:

```
graph * graph_read_ff(FILE * input);

public readFF(File input);
```

- una funzione/metodo che stampa a video un grafo utilizzando la rappresentazione descritta sopra:

  ```
  void graph_print(graph * g);

  public String toString(Graph g);
  ```

Task 3. Componenti connesse

In questo esercizio, occorre identificare e contare le componenti connesse di un grafo *non orientato*. Dato un grafo G, una componente connessa di G è un sottografo S in cui: (i) esiste un cammino che connetta qualsiasi coppia di nodi in S; (ii) nessun nodo in S è connesso con un nodo in G che non appartenga a S. Ad esempio, il grafo in Fig. 1 consiste di 3 componenti connesse.

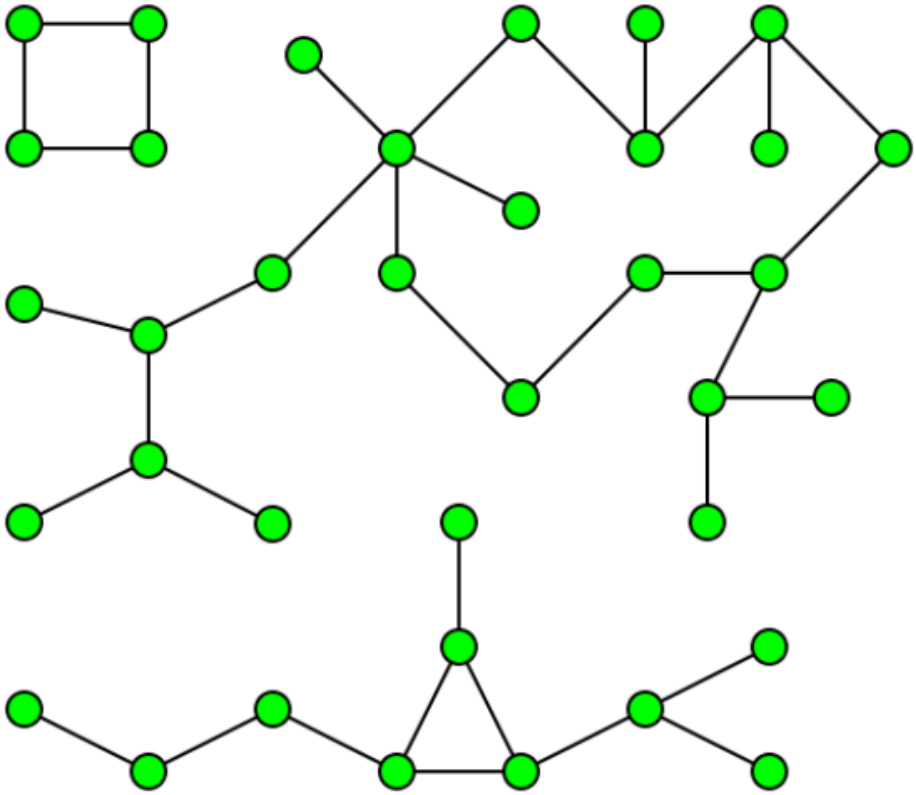


Figura 1: Un grafo con 3 componenti connesse.

Suggerimento: Si consiglia di contare ed identificare le componenti connesse di un grafo facendo uso della visita in profondità (DFS). Si noti che nulla proibisce di usare un'altro tipo di visita, ad esempio la visita in ampiezza (BFS) per risolvere questo problema in grafi *non orientati* (si noti che nel caso di grafi orientati, il metodo più efficiente per calcolare le componenti fortemente connesse è basato sulla DFS).

Specifiche. Estendere il modulo C con intestazione `graph.h` (rispettivamente la classe generica Java `Graph.java`) definito nell'esercizio precedente in modo che includa:

- una funzione/metodo che conti il numero di componenti connesse di un grafo:

```
int graph_n_con_comp(graph *);

public int nConComp();
```

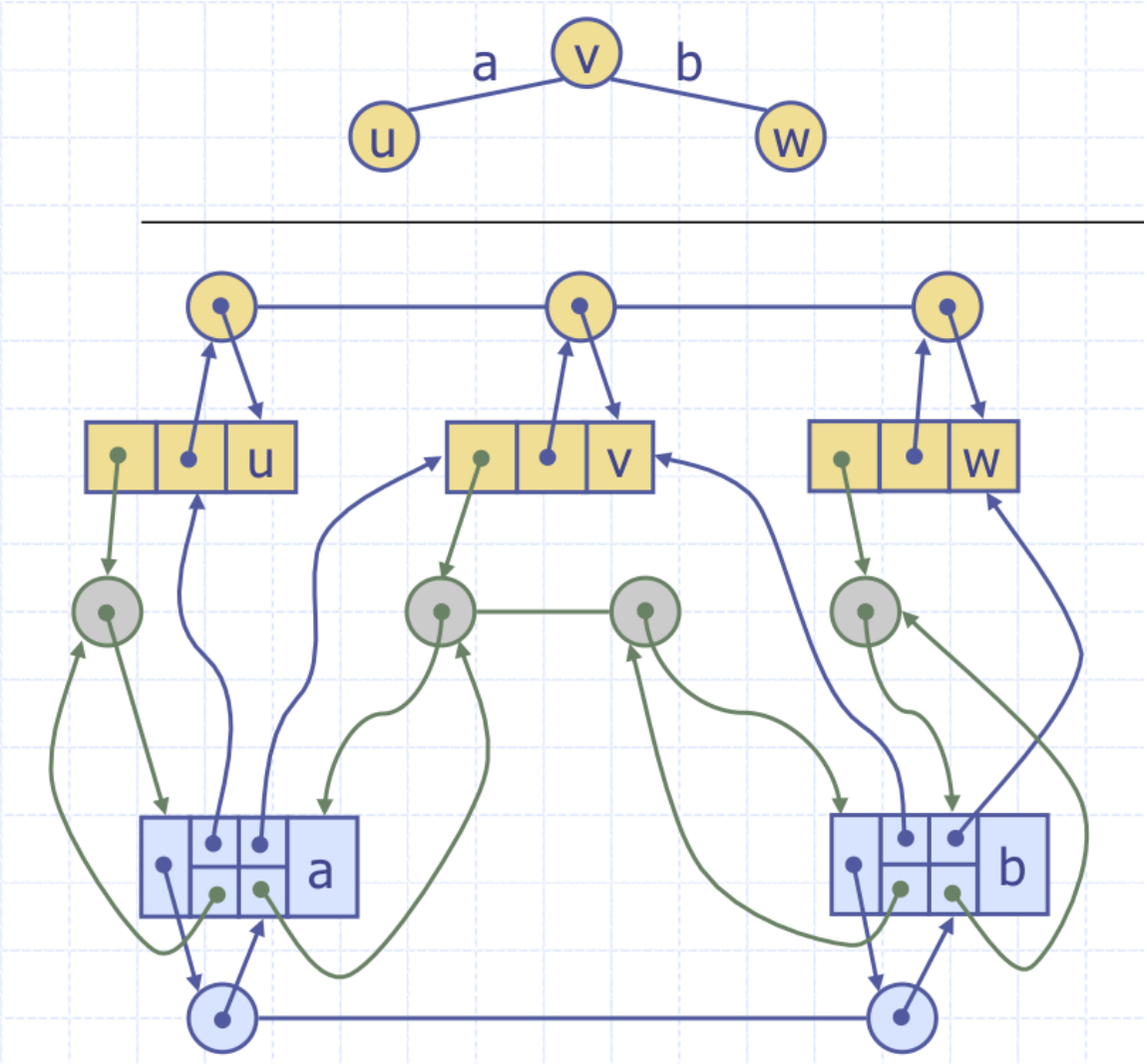
- una funzione che restituisca una lista i cui elementi sono le varie componenti connesse di un grafo:

```
linked_list * graph_get_con_comp(graph *);

public List<Graph> getConComp();
```

Task 4. Rappresentazione esplicita degli archi (per casa)

In generale, rappresentare gli archi esplicitamente può essere utile in alcuni casi. Ad esempio, lo è quando gli archi possono avere informazioni associate (quali ad esempio un peso). In tal caso, la lista di adiacenza di un nodo può essere realizzata semplicemente come una lista di riferimenti. In tal caso, ogni elemento della lista conterrà un riferimento a un oggetto "arco", che a sua volta conterrà le informazioni relative all'arco (ad esempio, un peso e/o un'etichetta), nonché riferimenti indietro alla lista dei nodi e alle liste di adiacenza dei nodi suoi estremi. A sua volta, gli oggetti arco possono essere collegati tra loro in una lista. La struttura risultante è quella della figura riportata di seguito:



Si riscriva il codice dei Task 1, 2, e 3, in modo tale che l'implementazione del grafo sia aderente alla rappresentazione collegata data in figura.

Per quanto riguarda l'informazione da associare agli archi, si supponga che essa sia un intero (peso), inizializzato a 1 per tutti gli archi. In tal modo, si potrà riutilizzare direttamente il codice di test dell'esercitazione precedente.

In un secondo momento, modificare i file di input, il codice di prova e il vostro codice, in modo che il grafo letto da input sia, per ogni riga, nel formato `<vertice_1> <vertice_2> <peso>`