

Laboratorio di algoritmi e strutture dati

Esercizi su implementazione di grafi*

Docente: Violetta Lonati

1 Rappresentazione di grafi

1.1 Slice di liste di adiacenza

Nel caso, semplificato, in cui i vertici del grafo siano identificati semplicemente da numeri naturali progressivi $0, 1, 2, \dots$, possiamo rappresentare la relazione di adiacenza con un *vettore di liste di adiacenza*. In questo caso l'insieme dei vertici viene rappresentato implicitamente: basta memorizzare il numero di vertici con una variabile intera n e l'insieme dei vertici è dato implicitamente dai numeri da 0 a $n-1$.

Realizzate questa implementazione dei grafi riutilizzando le funzioni già scritte per implementare liste concatenate.

- Definite un tipo `graph` come segue

```
type grafo struct {  
    n      int // numero di vertici  
    adiacenti []LinkedList  
}
```

NB: n è pari alla lunghezza della slice `adiacenti`

- Scrivete una funzione con segnatura `nuovoGrafo(n int) *graph` che restituisce l'indirizzo di un nuovo grafo con n vertici.
- Scrivete una funzione per leggere un grafo da standard input. Specificate voi il formato atteso, ad esempio: un numero n che indica il numero di vertici seguito da una serie di coppie di indici, una coppia per riga). La funzione dovrà allocare una slice di puntatori a lista, e inizializzare opportunamente tali puntatori. Dovrà poi inserire gli archi (inserendo in testa alla lista di adiacenza opportuna)
- Scrivete una funzione per stampare un grafo. Specificate voi il formato, ad esempio: l'indice di un vertice su ogni riga, seguito dagli indici dei vertici a lui adiacenti.
- Scrivete una funzione che dati due interi x e y stabilisca se c'è un arco tra x e y .

*Ultimo aggiornamento: 20 novembre 2023 - 13:21:58

1.2 Struttura per i vertici, con slice dei vertici adiacenti

Nel caso in cui si debbano memorizzare ed elaborare anche informazioni relative ai vertici del grafo, può essere interessante rappresentare i vertici esplicitamente definendo un tipo strutturato:

```
type vertice struct {
    valore item
    chiave string
    adj    []*vertice // insieme dei vertici adiacenti
}
```

La chiave `chiave` identifica il vertice (nel caso sopra si tratta di una stringa, ma potrebbe essere di altro tipo); le altre informazioni relative al vertice sono raggruppate sotto il campo `valore` di tipo `item`. A seconda dei casi, `item` potrebbe essere un tipo built-in o un tipo più articolato, ad esempio a sua volta un'altra struttura. Per rappresentare la relazione di adiacenza usiamo il campo `adj`, definito come una slice di puntatori: se `v` è un vertice, allora `v.adj` è la slice dei puntatori ai vertici adiacenti a `v`.

Tipicamente, può essere utile poter cercare efficientemente un vertice in un grafo; per questo può essere utile raggruppare l'insieme dei vertici in una mappa che ad ogni chiave associa il suo vertice (o meglio, un puntatore al vertice):

```
vertici map[string]*vertice
```

In alternativa può bastare avere una slice dei (puntatori ai) vertici:

```
vertici []*vertice
```

NB: anche l'insieme dei vertici adiacenti (campo `adj` della struttura `vertice`) può essere rappresentato con una mappa invece che con una slice, se ci aspettiamo che tali insiemi siano lunghi e vogliamo rendere più efficiente la ricerca all'interno di un insieme di adiacenza.

Esempio di applicazione Assumiamo per esempio di avere un grafo (orientato) in cui ogni vertice rappresenta un utente di una rete sociale (tipo Twitter). Ogni vertice ha un nome, un'età, e una serie di hobby (rappresentiamoli come come slice di stringhe: ogni stringa descrive un hobby). C'è un arco dal vertice *A* al vertice *B* se l'utente rappresentato dal vertice *A* segue l'utente rappresentato dal vertice *B*.

Realizzate un'implementazione dei grafi usando la rappresentazione descritta qui sopra:

- Scrivete una funzione con segnatura `graphNew(n int) *graph` che restituisce l'indirizzo di un nuovo grafo con `n` vertici.
- Scrivete una funzione per leggere un grafo da standard input (specificate voi il formato atteso).
- Scrivete una funzione per stampare un grafo (specificate voi il formato).
- Scrivete una funzione che data una stringa *A* stampi gli hobby dell'utente di nome *A* e l'elenco di tutti gli hobby delle persone **che seguono** *A*.

- Scrivete una funzione che data una stringa *A* stampi gli hobby dell'utente di nome *A* e l'elenco di tutti gli hobby delle persone **seguite** da *A*. Quale è la più complessa tra questa e l'operazione precedente?

2 Visite di grafi

L'implementazione delle visite andrà adattata a seconda di come si è scelto di rappresentare il grafo. Qui faremo riferimento a una rappresentazione che fa uso di una mappa (diversa quindi dalle due rappresentazioni proposte nelle sezioni precedenti). E' opportuno provare a implementare le varie visite anche con altre rappresentazioni.

Consideriamo adesso un caso semplice, in cui ogni vertice del grafo è identificato da una stringa; rappresentiamo il grafo soltanto tramite una mappa che descrive la relazione di adiacenza:

```
type grafo map[string][]string
```

La mappa associa ad ogni vertice (identificato da una stringa) la slice dei vertici adiacenti (anch'essi identificati da stringhe). Più precisamente, se il vertice *v*, identificato dalla stringa *s*, ha per adiacenti i vertici *v*₁, *v*₂, ..., *v*_n, identificati rispettivamente dalle stringhe *s*₁, *s*₂, ..., *s*_n, allora la mappa associa a *s* la slice contenente le stringhe *s*₁, *s*₂, ..., *s*_n.

L'insieme dei vertici non è definito esplicitamente, ma risulta definito implicitamente dall'insieme delle chiavi della mappa (in questo caso, un insieme di stringhe).

2.1 Visita in profondità con funzione ricorsiva

Implementiamo la visita in profondità con una funzione ricorsiva:

```
func dfs1(g grafo, v string, aux map[string]bool) {
    fmt.Println(v)
    aux[v] = true
    for _, v2 := range g[v] {
        if aux[v2] != true {
            dfs1(g, v2, aux)
        }
    }
}
```

La funzione fa uso di una struttura di supporto *aux* che serve a ricordare quali vertici sono già stati visitati. Qui usiamo una mappa da stringa a bool, poiché i vertici sono identificati semplicemente da stringhe). Prima di invocare la funzione, *aux* andrà allocata e inizializzata opportunamente.

NOTA: se si usasse la rappresentazione descritta nella parte 1.1, sarebbe più coerente usare una slice ausiliaria (i vertici sono individuati dagli indici interi) invece di una mappa.

2.2 Visita in ampiezza

Per implementare la visita in ampiezza usiamo una coda. Realizziamo la coda usando una slice di stringhe: inseriamo (*enqueue*) in fondo con `append` e estraiamo (*dequeue*) dall'inizio con `[1:]`

```
func bfs1(g grafo, v string, aux map[string]bool) {
    coda := []string{v}
    aux[v] = true

    for len(coda) > 0 {
        v := coda[0]
        coda = coda[1:]
        fmt.Println("\t", v)

        for _, v2 := range g[v] {
            if !aux[v2] {
                coda = append(coda, v2)
                aux[v2] = true
            }
        }
    }
}
```

La visita (in questo caso stampiamo la stringa che identifica il vertice) avviene quando si estraggono gli elementi dalla coda. Quando inseriamo un vertice nella coda lo marchiamo come visitato (usando la solita mappa `aux`).

2.3 Visita in profondità con pila di supporto

Sostituendo la coda con una pila (basta fare anche l'estrazione dal fondo invece che dall'inizio, come per l'inserimento), si ottiene un'implementazione non ricorsiva della visita in profondità.

3 Proprietà dei grafo

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

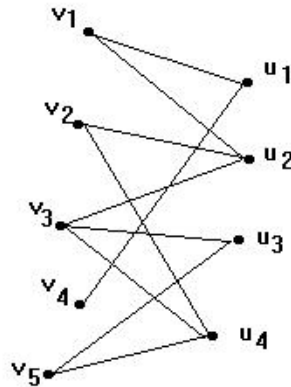
Scegliete voi quale rappresentazione del grafo usare; provate a implementare la stessa operazione con rappresentazioni reverse; valutate quale rappresentazione

Attenzione: le signature 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 (non orientato).
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 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. **twocolor** testa se il grafo è bicoloreabile.
Un grafo si dice *bicoloreabile* quando è possibile assegnare ad ogni vertice del grafo uno dei due colori bianco o nero in modo che due vertici vicini abbiano sempre colori diversi. Quando un grafo è bicoloreabile, 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.

8. **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!