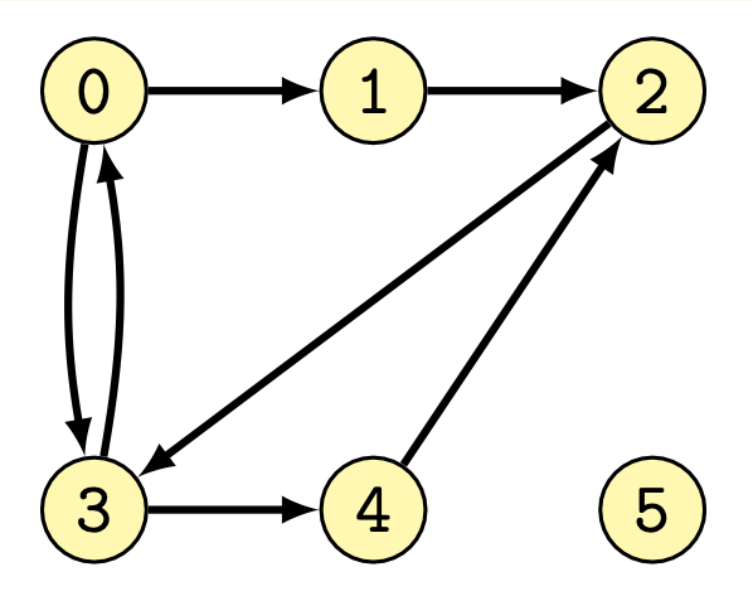
Implementazione di grafi in Go

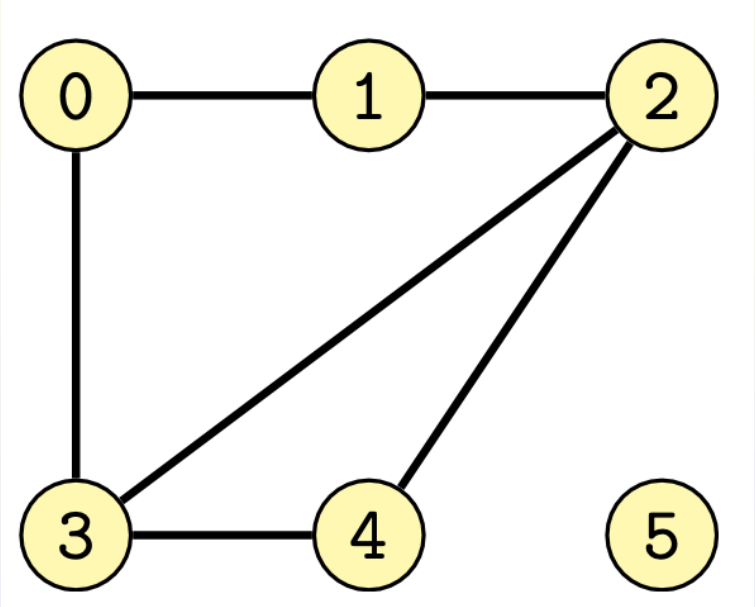
Violetta Lonati

Università degli studi di Milano
Dipartimento di Informatica

Laboratorio di algoritmi e strutture dati
Corso di laurea in Informatica

Esempi di grafo orientato e non orientato



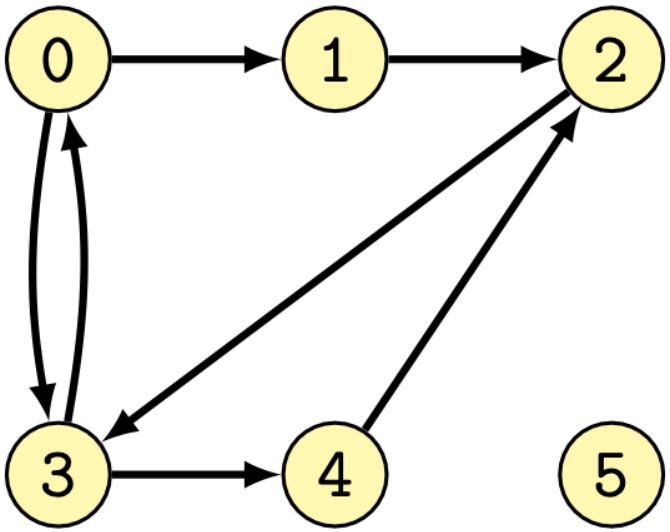


Matrice di adiacenza

Si può implementare in Go con una slice di slice

$$m_{uv} = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$

$\text{Spazio} = n^2 \text{ bit}$



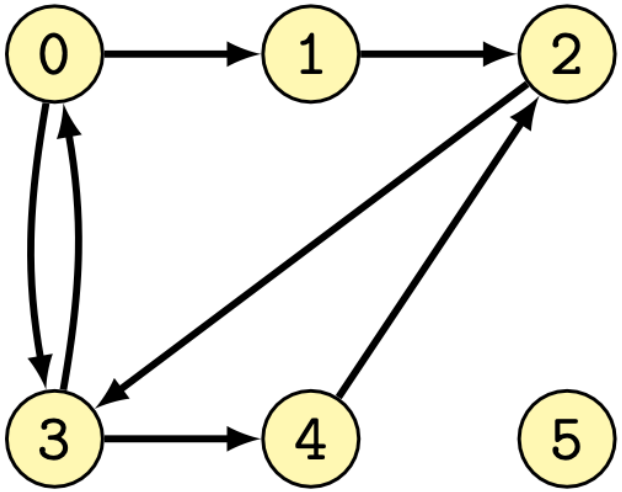
	0	**1**	**2**	**3**	**4**	**5**
0	0	1	0	1	0	0
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	1	0	0	0	1	0
4	0	0	1	0	0	0
5	0	0	0	0	0	0

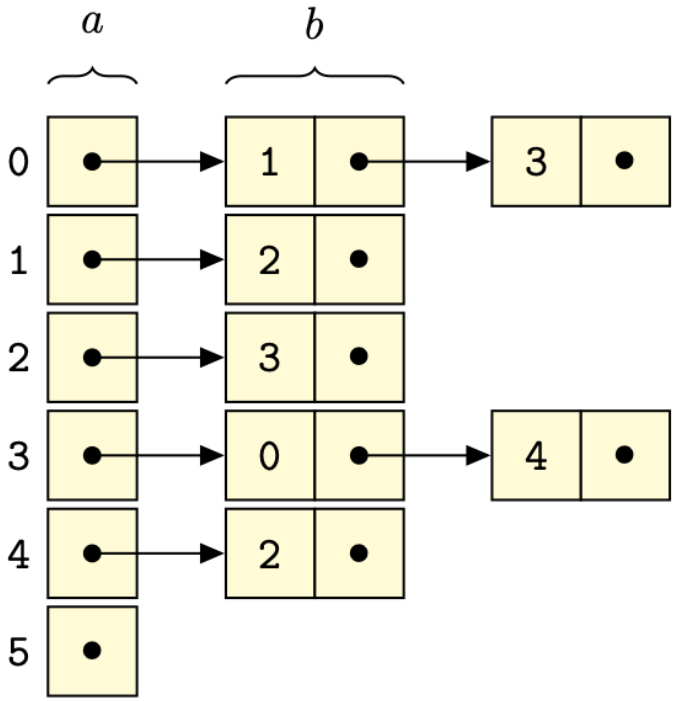
Array di liste di adiacenza

Consideriamo per ora il caso orientato

$G.\text{adj}(u) = \{v | (u, v) \in E\}$

$\text{Spazio} = an + bm$ bit





Array di liste di adiacenza

- Se i nodi sono identificati da numeri progressivi da 0 a n, allora l'insieme dei nodi è rappresentato implicitamente da una variabile intera con valore n.
- L'insieme degli archi, ovvero la relazione di adiacenza del grafo, è rappresentata tramite liste di adiacenza; ogni vertice ha la sua lista di adiacenza, formata dall'insieme dei nodi collegati tramite un arco uscente da i.
- Le liste sono raccolte in un array di n elementi. L'elemento i-esimo dell'array punta al primo elemento della lista di adiacenza del nodo i.

Implementazione “alla lettera”

```
type graph struct {
  n   int
  adj []*listNode
}

type listNode struct {
  item int
  next *listNode
}
```

Riuso dell'implementazione delle liste

E' possibile riutilizzare le funzioni già implementate con questi tipi e segnature:

```
type listNode struct {
  item int
  next *listNode
}
type linkedList struct {
  head *listNode
}
func addNewNode(l *linkedList, n int)
```

NB: in questo caso la slice dovrà contenere liste e non puntatori a nodi:

```
type graph struct {
  n          int
  adij []linkedList
}
```

Slice invece di liste concatenate

Poiché in Go è facile rappresentare insiemi dinamici con le slice, le liste di adiacenza possono essere sostituite da slice.

```
type adjSet []int // insieme di adiacenti

type graph struct {
  n   int
  adj []adjSet // vettore degli insiemi di adiacenza
}
```

`adj[i]` è una slice di interi, che contiene gli indici dei nodi adiacenti al nodo `i`.

Implementazione semplice con mappe

Se i vertici non sono semplici numeri, invece della slice di insiemi di adiacenza si può usare una mappa:
Se ad esempio i vertici sono identificati da stringhe, si può definire:

```
type graph map[string][]string
```

Ad ogni vertice (identificato da una stringa) la mappa associa una slice contenente i vertici vicini (ciascuno di essi è identificato anch'essp da una stringa).

Anche in questa implementazione l'insieme dei vertici non è definito esplicitamente, ma risulta definito implicitamente dall'insieme delle chiavi della mappa (qui: un insieme di stringhe).

Implementazione con struttura vertice

Se ai vertici del grafo sono associate informazioni più articolate (che quindi non possono essere usate come chiavi di una mappa), possiamo invece usare una struttura `vertice`:

```
type vertice struct {
  valore item
  k      chiave
  adj    []*vertice // insieme dei v. adiacenti
}
```

Questa implementazione ricorda l'implementazione degli alberi binari con puntatori ai figli destro e sinistro (invece dei puntatori ai figli, qui abbiamo una slice di puntatori ai vertici adiacenti).

Implementazione con struttura vertice (continua)

Nella struttura vertice abbiamo questi campi:

- `valore` raccoglie le varie informazioni relative a un vertice (il tipo `item` dipende dal tipo di informazioni, potrebbe anche essere un puntatore ad un'altro tipo di struttura)
- `k` è la chiave che identifica il vertice in modo univoco
- `adj` è una slice che contiene gli indirizzi degli altri vertici

Per rappresentare il grafo basta una mappa che associa, ad ogni chiave, il suo vertice (nel caso dell'albero binario bastava avere la radice, ma qui dobbiamo avere tutto l'insieme dei vertici):

```
type graph map[chiave]*vertice
```