Strutture Dati

Lezione 15
Grafi

Oggi parleremo di …

- Rappresentazione dei grafi
  - matrici di adiacenza (fatto!)
  - liste di adiacenza
    - concatenate
    - sequenziali
  - multiliste di adiacenza.
- Operazioni
  - visita in profondità

Liste di adiacenza (concatenate)

- Per ogni vertice si costruisce una lista concatenata contenente i lati in uscita dal vertice.
- Se abbiamo *n* nodi e *l* lati questa rappresentazione richiede *n* nodi di testa e 2*l* nodi di lista.

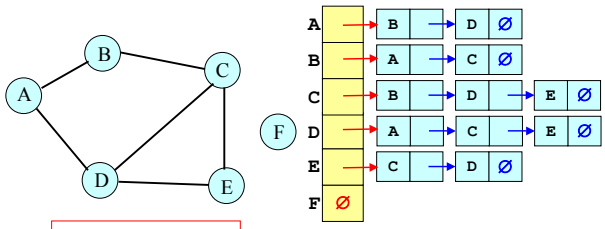
```
#define MAX_VERTICI 10

typedef struct nodo{
        int vertice;
        struct nodo *link;
      } *node_pointer;

node_pointer grafo[MAX_VERTICI];
```

`grafo(v)` rappresenta la lista di tutti i vertici adiacenti a `v`

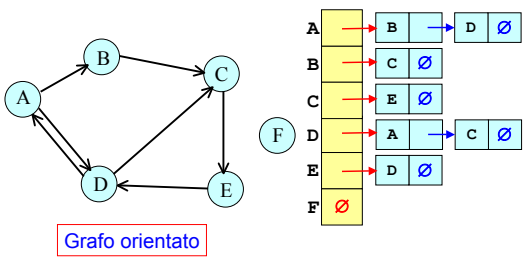
Liste di adiacenza (concatenate)



Grafo non orientato

Lo spazio occupato è $O(n+l)$.

Liste di adiacenza (concatenate)



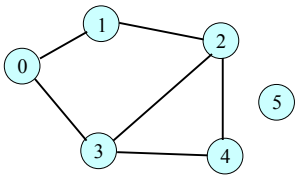
Grafo orientato

Lo spazio occupato è $O(n+l)$.

Liste di adiacenza (sequenziali)

- Si impaccano i nodi della lista in un array *nodo*[ ]
  - *nodo*[*i*] = punto di partenza della lista di adiacenza del vertice *i*, per *i* = 0, …, *n*-1
  - *nodo*[*n*] = *n* + 2*l* + 1.
- I vertici adiacenti al vertice *i* saranno memorizzati in *nodo*[*i*], …, *nodo*[*i*+1]-1.

Liste di adiacenza (sequenziali)



0	7	10	2
1	9	11	1
2	11	12	3
3	14	13	4
4	17	14	0
5	19	15	2
6	19	16	4
7	1	17	2
8	3	18	3
9	0		

$n + 2l + 1 = 6 + 2 \times 6 + 1 = 19$

Lo spazio occupato è $O(n+l)$.

Multiliste di adiacenza

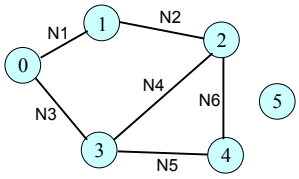
- Nelle liste un lato (v1,v2) appare sia nella lista di v1 che nella lista di v2.
- E' possibile rappresentare un grafo partendo dai lati invece che dai vertici.
- Ciò consente di creare una lista i cui i nodi "appartengono" a più liste (multilista).

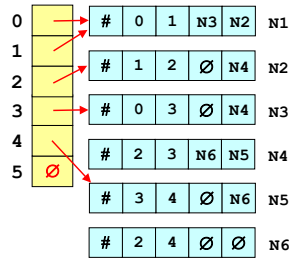
```
typedef struct lato{
        short int marcato;
        int vertice1, vertice2;
        lato_pointer percorso1;
        lato_pointer percorso2;
      } *lato_pointer;

lato_pointer grafo[MAX_VERTICI];
```

marcato	vertice1	vertice2	percorso1	percorso2

Multiliste di adiacenza





Le liste sono:

vertice 0: N1 → N3
vertice 1: N1 → N2
vertice 2: N2 → N4 → N6
vertice 3: N3 → N4 → N5
vertice 4: N5 → N6
vertice 5: NULL

Risparmio spazio!

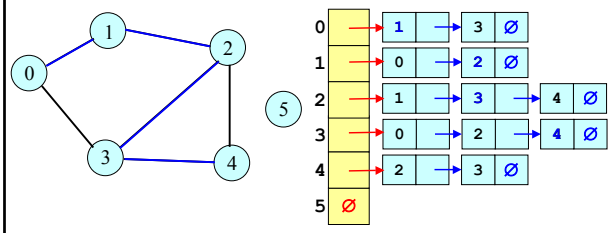
Operazioni sui grafi

- Dato un grafo $G=(V,L)$ e un vertice v, visitare tutti i vertici raggiungibili da v
  - ricerca in profondità
  - ricerca in ampiezza.
- Ricercare tutte le componenti connesse di un grafo.
- Determinare l'albero derivato (spanning tree) di un grafo.

Visita in profondità di un grafo

- La ricerca in profondità è simile alla visita preorder di un albero.
- Si parte da un vertice v, visitandolo.
- Si visita il primo vertice non visitato w adiacente a v.
- Si salva la posizione corrente nella lista di adiacenza di v ponendola nello *stack*.
- Si prosegue sino a raggiungere un nodo per il quale tutti i vertici adiacenti sono stati visitati
  - si risale il cammino sino al primo nodo che ammette un vertice non ancora visitato e si ricomincia seguendo un nuovo cammino con lo stesso criterio.
- L'attraversamento termina quando tutti i percorsi ignorati nella visita sono stati considerati (quando lo stack è vuoto).

Visita in profondità di un grafo



La sequenza dei vertici raggiungibili da 0 in profondità è:

0 1 2 3 4

Visita in profondità di un grafo

```
#define  MAX_VERTICI 30
#define FALSE 0
#define TRUE 1

typedef struct nodo{
        int vertice;
        struct nodo *link;
      } *node_pointer;

nodo_pointer grafo[MAX_VERTICI];
int visitato[MAX_VERTICI];
```

```
void rpr(int v)
/*ricerca in profondità di un grafo*/
{
 nodo_pointer w;

 visitato[v]=TRUE;
 printf("%d  ",v);
 for(w=grafo[v];w; w=w->link)
    if(!visitato[w->vertice]) rpr(w->vertice);
}
```

La complessità con le liste di adiacenza è $O(l)$.

La complessità con le matrici di adiacenza è $O(n^2)$.