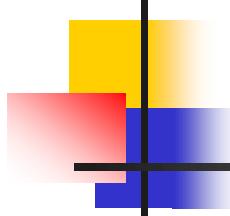


# L'ADT grafo

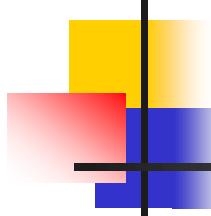
Gianpiero Cabodi e Paolo Camurati  
Dip. Automatica e Informatica  
Politecnico di Torino





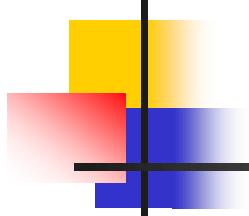
# Perché la Teoria dei Grafi?

- Moltissime applicazioni pratiche
- Centinaia di algoritmi
- Astrazione interessante e utilizzabile in molti domini diversi
- Ricerca attiva in Informatica e Matematica discreta.

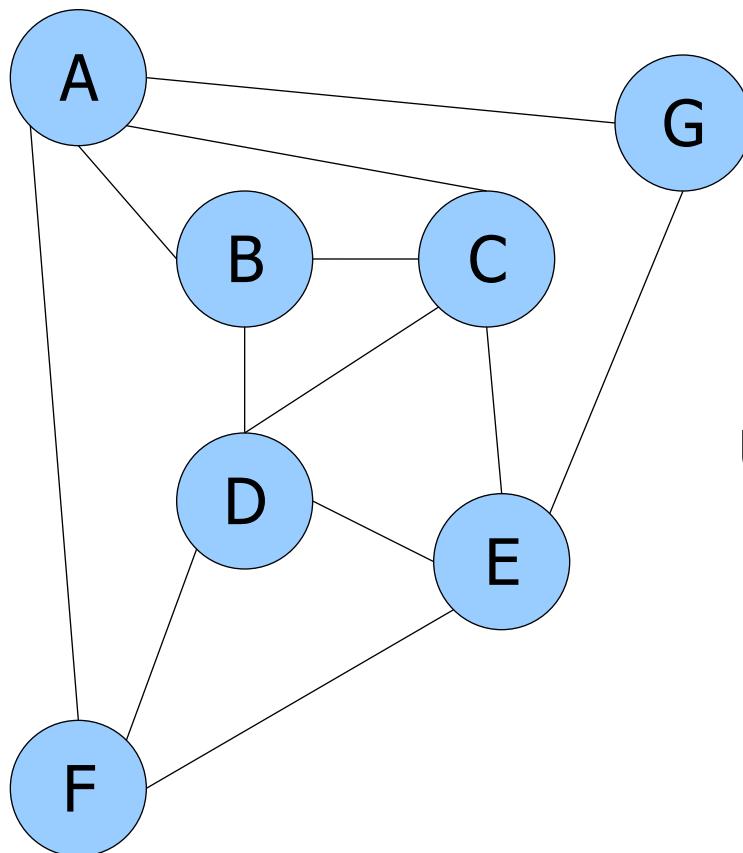


# Complessità dei problemi

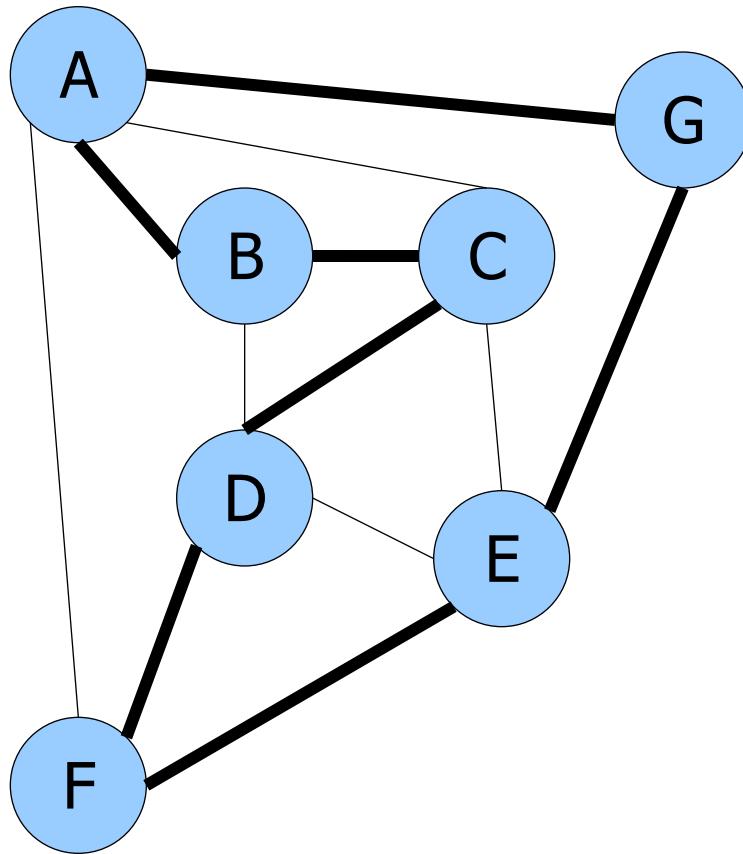
- problemi facili:
  - determinare se un grafo è connesso
  - determinare la presenza di un ciclo
  - individuare le componenti fortemente connesse
  - individuare gli alberi minimi ricoprenti
  - calcolare i cammini minimi
  - determinare se un grafo è bipartito
  - trovare un cammino di Eulero
- problemi trattabili:
  - planarità di un grafo
  - matching

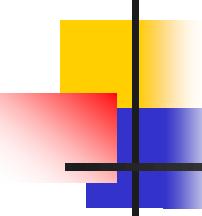
- 
- problemi intrattabili:
    - cammini a lunghezza massima
    - colorabilità
    - clique massimale
    - ciclo di Hamilton
    - problema del commesso viaggiatore
  - problemi ignoti:
    - isomorfismo di due grafi

# Ciclo di Hamilton



Dato un grafo non orientato  $G = (V, E)$ , esiste un ciclo semplice che visita ogni vertice una e una sola volta?



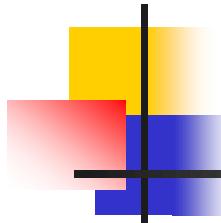


# Colorabilità

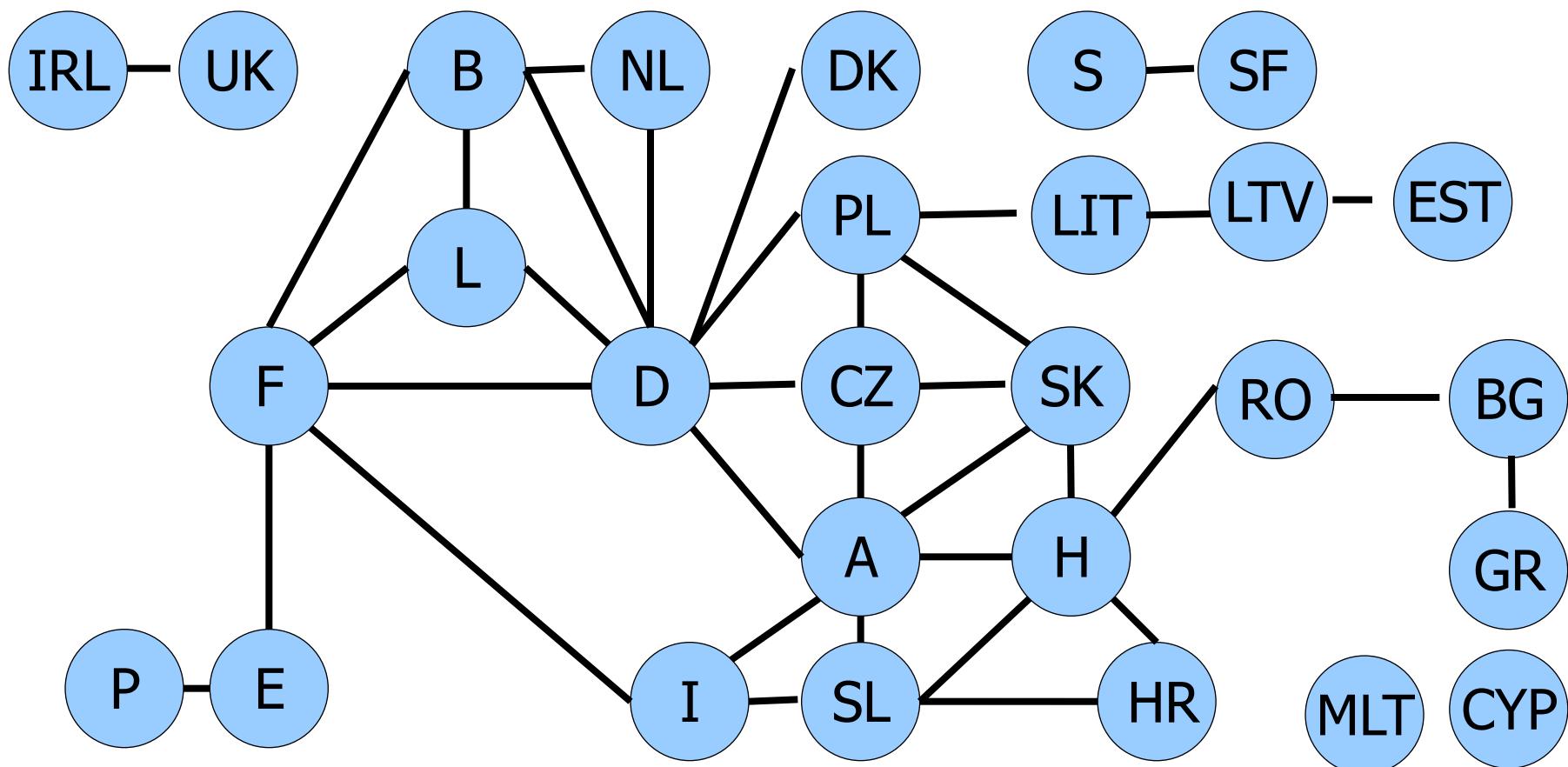
Dato un grafo non orientato  $G = (V, E)$ , quale è il minimo numero di colori  $k$  necessario affinché nessun vertice abbia lo stesso colore di un vertice ad esso adiacente?

Si dice «planare» un grafo che, se disegnato su di un piano, non ha archi che si intersecano.

Le mappe geografiche sono modellate come grafi planari.

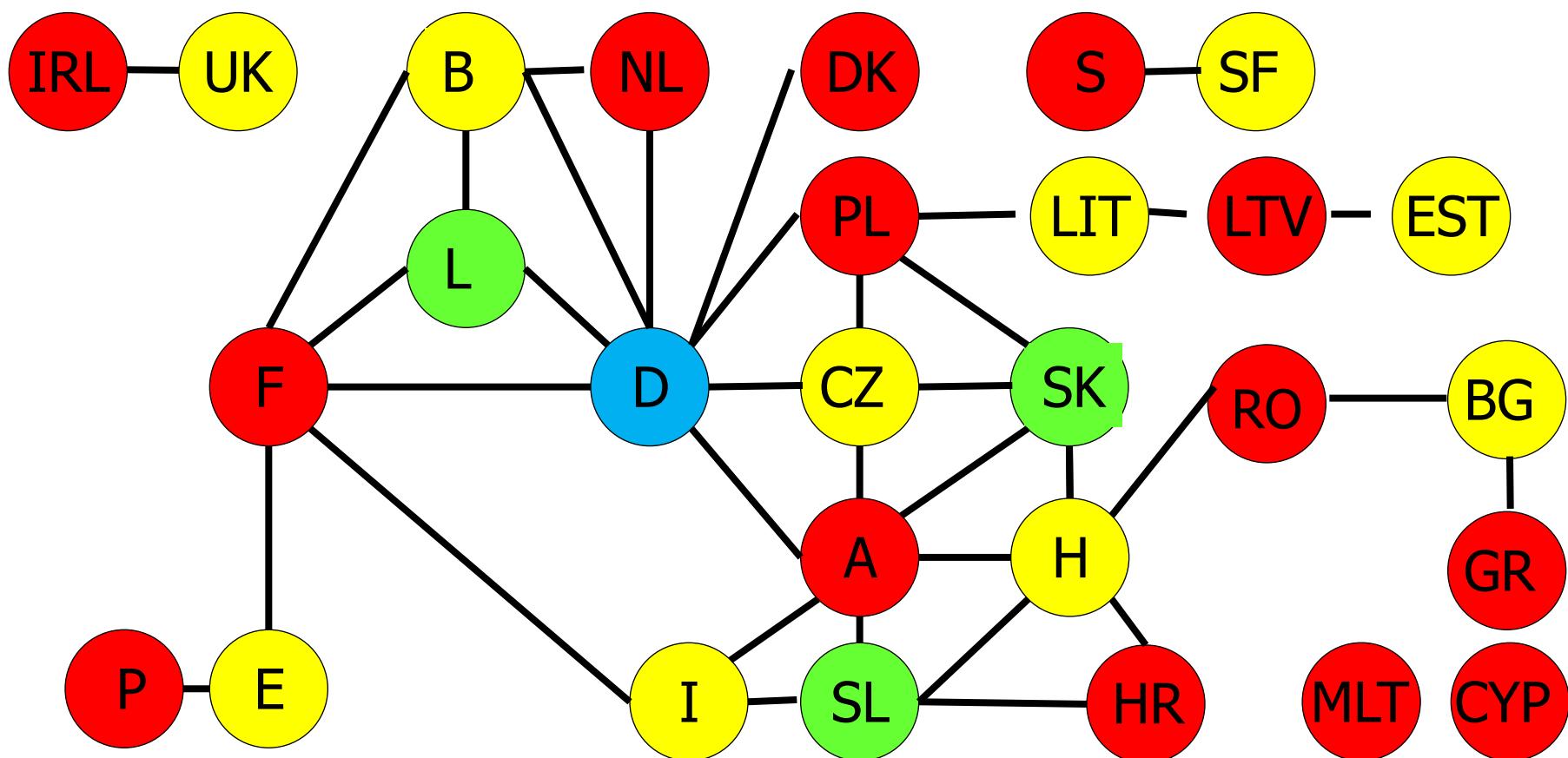


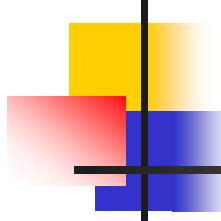
# L'UE a 28 stati



# Colorabilità: $k=4$

Per i grafi planari si può dimostrare che servono al più 4 colori.





# L'ADT grafo

---

- Ipotesi
- Interfaccia
- Lettura da file
- Rappresentazione:
  - matrice delle adiacenze
  - lista delle adiacenze
  - (elenco di archi)

# Riassunto Tipologie di Grafo

Grafi orientati e pesati

Grafi non orientati e pesati

$$(u,v) \in E \Leftrightarrow (v,u) = \in E$$

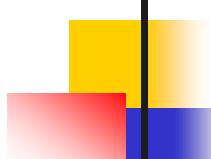
Grafi orientati e non pesati

$$\forall (u,v) \in E \quad \text{wt}(u,v)=1$$

Grafi non orientati e non pesati

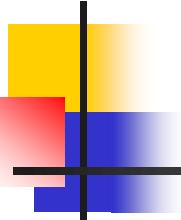
$$\forall (u,v) \in E \quad w(u,v)=1$$

$$(u,v) \in E \Leftrightarrow (v,u) = \in E$$



# Ipotesi

- Il modello più generale è quello di grafo orientato e pesato, ma per efficienza si considerano separatamente le 4 tipologie
- Grafi statici: non si aggiungono né si cancellano vertici, si possono aggiungere o cancellare archi
- I nomi dei vertici sono stringhe, internamente sono interi:
  - tabella di simboli per metterli in corrispondenza biunivoca con gli interi. Negli esempi per semplicità si usa un vettore non ordinato per implementarla
  - i vertici come interi permettono un accesso diretto tramite indice in un vettore (costo  $O(1)$ ).



- Il numero di vertici  $|V|$  che serve per inizializzare grafo e tabella di simboli può essere:
  - noto per lettura o perché compare come intero sulla prima riga del file da cui si legge
  - ignoto, ma sovrastimabile: se il grafo è dato come elenco di archi
    - con una prima lettura si determina il numero di archi e  $|V|$  si sovrasta come 2 volte il numero di archi, ipotizzando che ogni arco connetta vertici distinti
    - con una seconda lettura si inseriscono i vertici su cui insistono gli archi in una tabella di simboli, ottenendo il numero di vertici distinti  $|V|$  esatto e la corrispondenza vertice – intero.

# Quasi ADT Edge

grafi pesati

**Edge.h**

```
typedef struct edge { int v; int w; int wt; } Edge;  
Edge EDGEcreate(int v, int w, int wt);
```

Edge per grafi non pesati

v	w
---	---

Edge per grafi pesati

v	w	wt
---	---	----

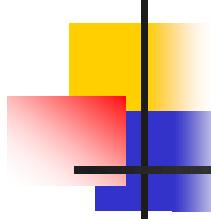
**Edge.c**

```
Edge EDGEcreate(int v, int w, int wt) {  
    Edge e;  
    e.v = v; e.w = w; e.wt = wt;  
    return e;  
}
```

# ADT di I cat. Grafo

## Graph.h

```
#include "Edge.h"
#include "ST.h"
#include "Queue.h"
typedef struct graph *Graph;
Graph GRAPHinit(int);
void GRAPHread(Graph);
void GRAPHwrite(Graph);
void GRAPHinsert(Graph);
void GRAPHremove(Graph);
void GRAPHshow(Graph);
void GRAPHedges(Graph, Edge *);
int GRAPHpath(Graph);
int GRAPHpathH(Graph); ← grafi non orientati
void GRAPHbfs(Graph);
void GRAPHdfs(Graph G);
int GRAPHcc(Graph G); ← grafi orientati
int GRAPHscc(Graph G); ← grafi orientati
```



# Lettura di un grafo da file

- il file contiene la lista degli archi
- è già noto il numero di vertici  $|V|$
- si scandiscono gli archi:
  - per ciascuno dei vertici su cui l'arco insiste, tramite la tabella di simboli se ne ricava l'indice se il vertice è già presente o lo si inserisce attribuendogli un indice se non lo è ancora.

```
void GRAPHread(Graph G) {  
    char name[MAX], src[MAX], dst[MAX];  
    int id1, id2, wt; ← grafi pesati  
    FILE *fin;  
    printf("Input file name: "); scanf("%s", name);  
    fin = fopen(name, "r");  
    if (fin == NULL) exit(-1);  
    while(fscanf(fin, "%s %s %d", src, dst, &wt) == 2) {  
        id1 = STsearch(G->tab, src);  
        if (id1 == -1) id1 = STinsert(G->tab, src);  
        id2 = STsearch(G->tab, dst);  
        if (id2 == -1) id2 = STinsert(G->tab, dst);  
        if (id1 < 0 || id2 < 0) continue; ↓  
        insertE(G, EDGEcreate(id1, id2, wt));  
    }  
    return;  
}
```

# Scrittura di un grafo su file

```
void GRAPHwrite(Graph G) {  
    int i;  
    char name[MAX];  
    FILE *fout;  
    Edge a[G->E];  
    printf("Input file name: ");  
    scanf("%s", name);  
    fout = fopen(name, "w");  
    if (fout == NULL)  
        exit(-1);  
    GRAPHedges(G, a);  
    for (i = 0; i < G->E; i++)  
        fprintf(fout, "%s %s %d\n", STretrieve(G->tab, a[i].v),  
                STretrieve(G->tab, a[i].w),  
                a[i].wt);  
    fclose(fout);  
    return;  
}
```

grafi pesati

# Inserzione di archi

```
void GRAPHinsert(Graph G) {  
    char src[MAX], dst[MAX];  
    int id1, id2, wt;  
    printf("Insert first node = ");  
    scanf("%s", src);  
    printf("Insert second node = ");  
    scanf("%s", dst);  
    printf("Insert weight = ");  
    scanf("%d", &wt);  
    id1 = STsearch(G->tab, src);  
    id2 = STsearch(G->tab, dst);  
    if (id1 < 0 || id2 < 0)  
        return;  
    insertE(G, EDGEcreate(id1, id2, wt));  
    return;  
}
```

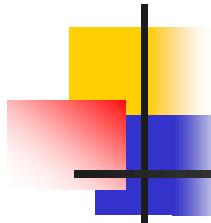
grafi pesati

# Rimozione di archi

```
void GRAPHremove(Graph G) {  
    int id1, id2; char src[MAX], dst[MAX];  
    printf("Insert first node = ");  
    scanf("%s", src);  
    printf("Insert second node = ");  
    scanf("%s", dst);  
    id1 = STsearch(G->tab, src);  
    id2 = STsearch(G->tab, dst);  
    if (id1 < 0 || id2 < 0)  
        return;  
    removeE(G, EDGEcreate(id1, id2, 0));  
    return;  
}
```

grafi pesati: il  
peso è fittizio





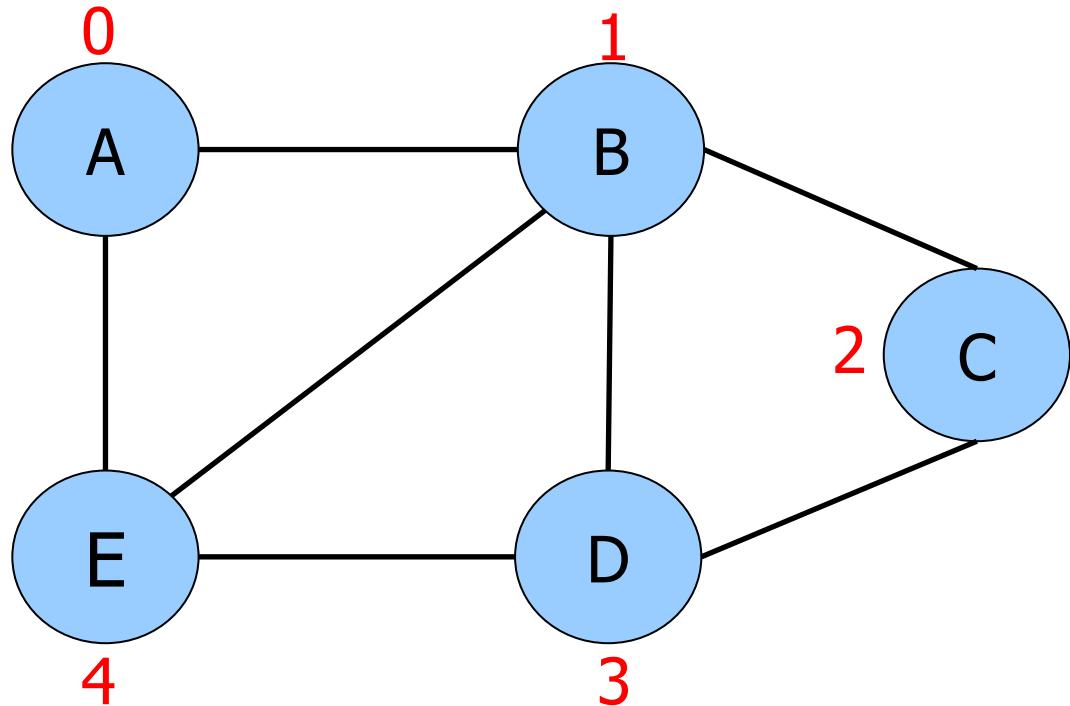
# Rappresentazione

## Matrice di adiacenza

Dato  $G = (V, E)$ , la matrice di adiacenza è:

- matrice  $\text{adj}$  di  $|V| \times |V|$  elementi
- $\text{adj}[i,j] = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{se } (i, j) \notin E \end{cases}$
- grafi non orientati:  $\text{adj}$  simmetrica
- grafi pesati:  $\text{adj}[i,j]=\text{peso dell'arco } (i,j) \text{ se esiste, 0 non è un peso ammesso.}$

# Non orientato non pesato

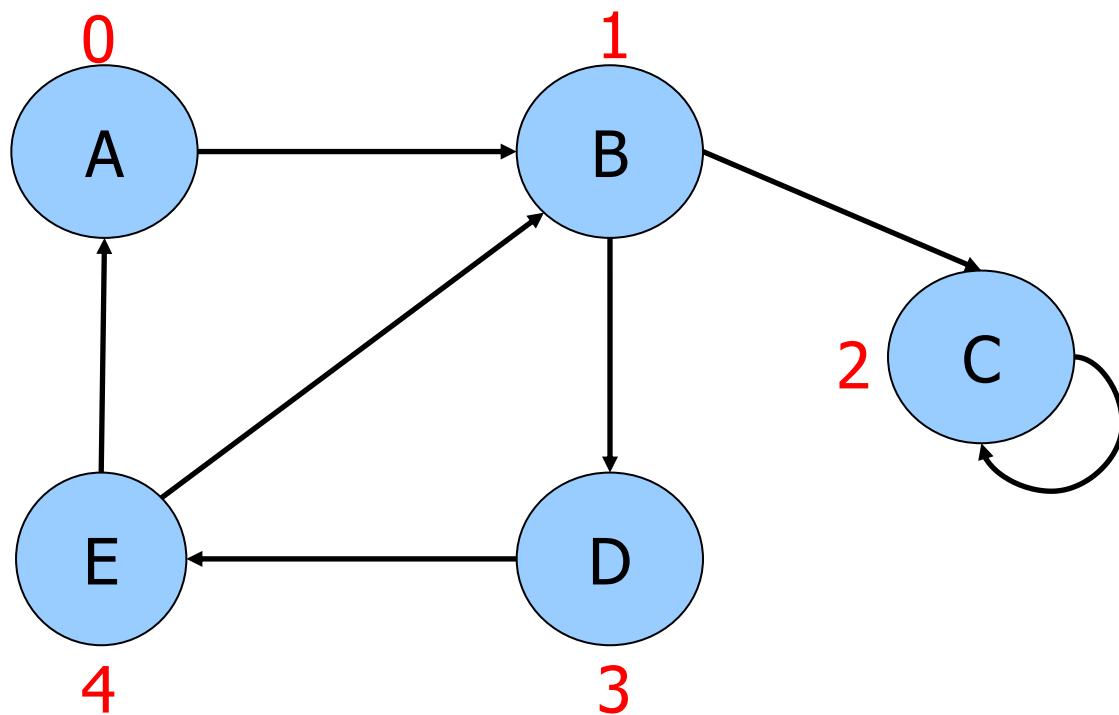


G->adj

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

in.txt	ST
0	A
1	B
2	C
3	D
4	E

# Orientato non pesato



in.txt

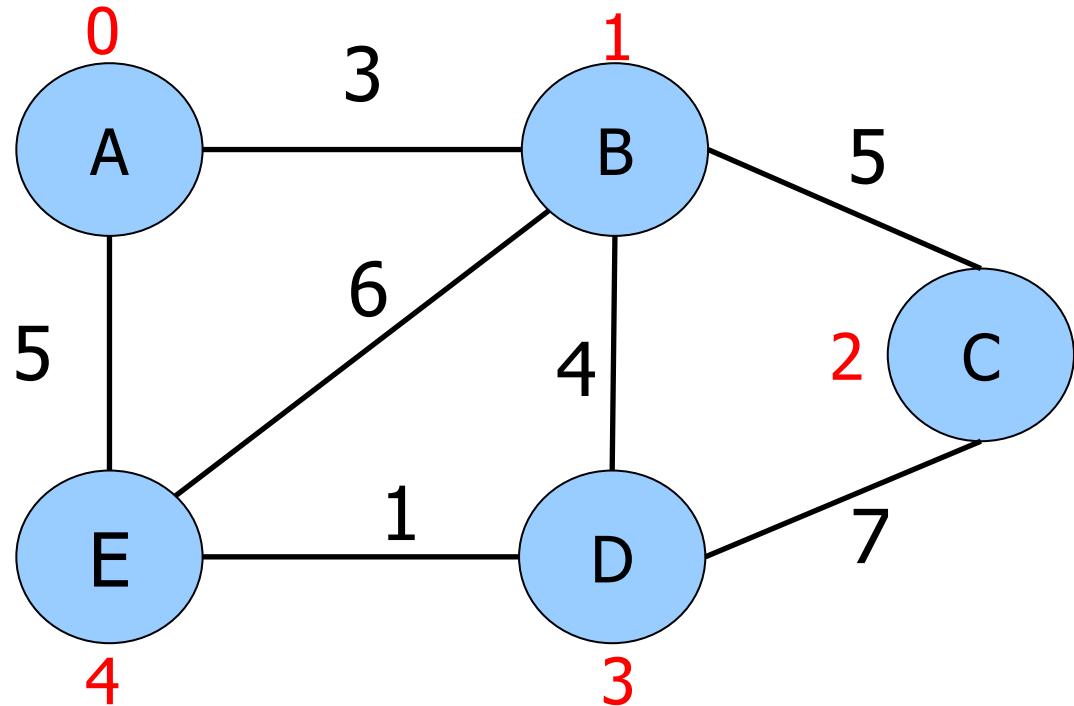
ST
A
B
C
D
E

0 A  
1 B  
2 C  
3 D  
4 E

G->adj

	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	1	0
2	0	0	1	0	0
3	0	0	0	0	1
4	1	1	0	0	0

# Non orientato pesato



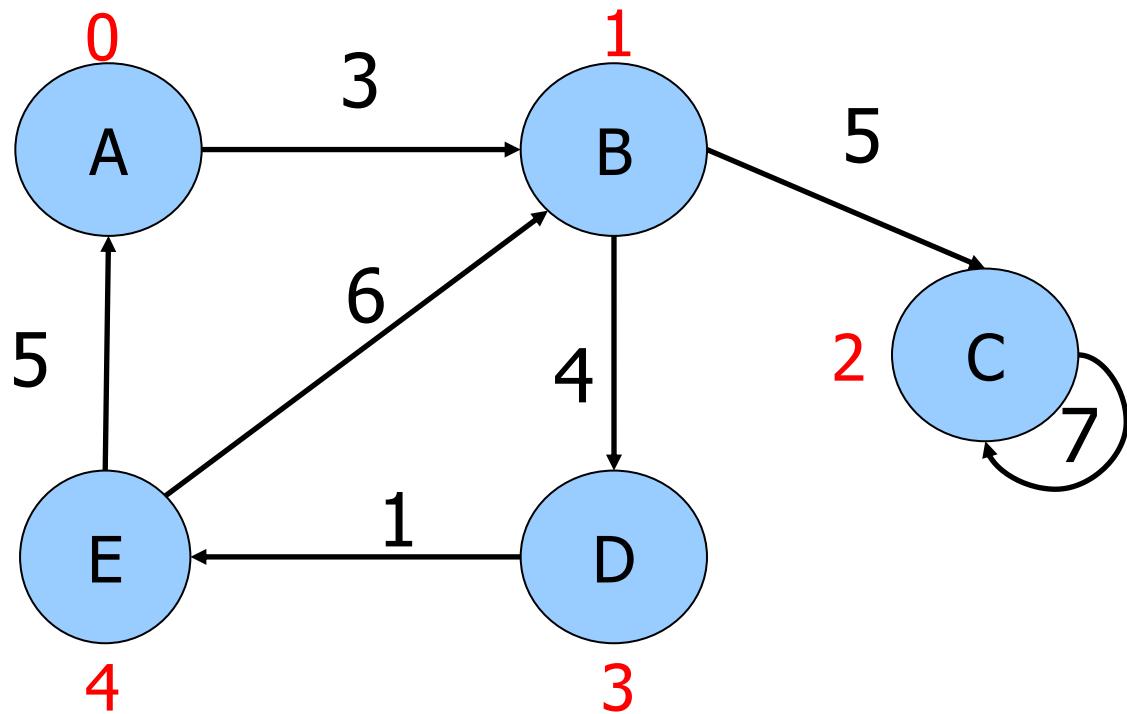
in.txt

ST	A	B	C	D	E
0	A				
1		B			
2			C		
3				D	
4					E

G->adj

0	1	2	3	4	
0	0	3	0	0	5
1	3	0	5	4	6
2	0	5	0	7	0
3	0	4	7	0	1
4	5	6	0	1	0

# Orientato pesato



in.txt

ST	A	B	C	D	E
0	A				
1		B			
2			C		
3				D	
4					E

G->adj

0	1	2	3	4
0	0	3	0	0
1	0	0	5	4
2	0	0	7	0
3	0	0	0	0
4	5	6	0	0

## Graph.c

```
#include <stdlib.h>
#include <stdio.h>
#include "Graph.h"
#define MAX 10

struct graph {int v; int E; int **adj; ST tab;};

int **MATRIXint(int r, int c, int val) {
    int i, j;
    int **t = malloc(r * sizeof(int *));
    for (i=0; i < r; i++) t[i] = malloc(c * sizeof(int));
    for (i=0; i < r; i++)
        for (j=0; j < c; j++)
            t[i][j] = val;
    return t;
}
```

Numero di vertici

Numero di archi

Matrice

ST

Attenzione: si possono generare cappi!

```
Graph GRAPHinit(int v) {  
    Graph G = malloc(sizeof *G);  
    if (G == NULL)  
        return NULL;  
    G->V = V;  
    G->E = 0;  
    G->adj = MATRIXint(v, v, 0);  
    G->tab = STinit(v);  
    if (G->tab == NULL)  
        return NULL;  
    return G;  
}  
void insertE(Graph G, Edge e) {  
    int v = e.v, w = e.w, wt = e.wt;  
    if (G->adj[v][w] == 0)  
        G->E++;  
    G->adj[v][w] = 1; G->adj[v][w] = wt;  
    G->adj[w][v] = 1; G->adj[w][v] = wt;  
}
```

grafi pesati

grafi non orientati

grafi non orientati pesati

```
void removeE(Graph G, Edge e) {  
    int v = e.v, w = e.w;  
    if (G->adj[v][w] != 0)  
        G->E--;  
    G->adj[v][w] = 0;  
    G->adj[w][v] = 0;  
}
```

grafi non orientati

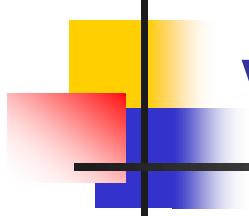
```
void GRAPHedges(Graph G, Edge *a) {  
    int v, w, E = 0;  
    for (v=0; v < G->V; v++)  
        for (w=v+1; w < G->V; w++)  
            for (w=0; w < G->V; w++)  
                if (G->adj[v][w] !=0)  
                    a[E++] = EDGEcreate(v, w, G->adj[v][w]);  
    return;  
}
```

grafi orientati

grafi pesati

```
void GRAPHshow(Graph G) {  
    int i, j;  
    printf("Graph has %d vertices, %d edges \n", G->V, G->E);  
    for (i=0; i < G->V; i++) {  
        printf("%s: ", STretrieve(G->tab, i));  
        for (j=0; j < G->V; j++)  
            if (G->adj[i][j] != 0)  
                printf("%s %d ", STretrieve(G->tab, j), G->adj[v][w]);  
        printf("\n");  
    }  
}
```

grafi pesati



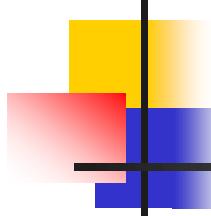
# Vantaggi/svantaggi

- Complessità spaziale

$$S(n) = \Theta(|V|^2)$$

⇒ vantaggiosa SOLO per grafi densi

- No costi aggiuntivi per i pesi di un grafo pesato
- Accesso efficiente (**O(1)**) alla topologia del grafo.



# Rappresentazione

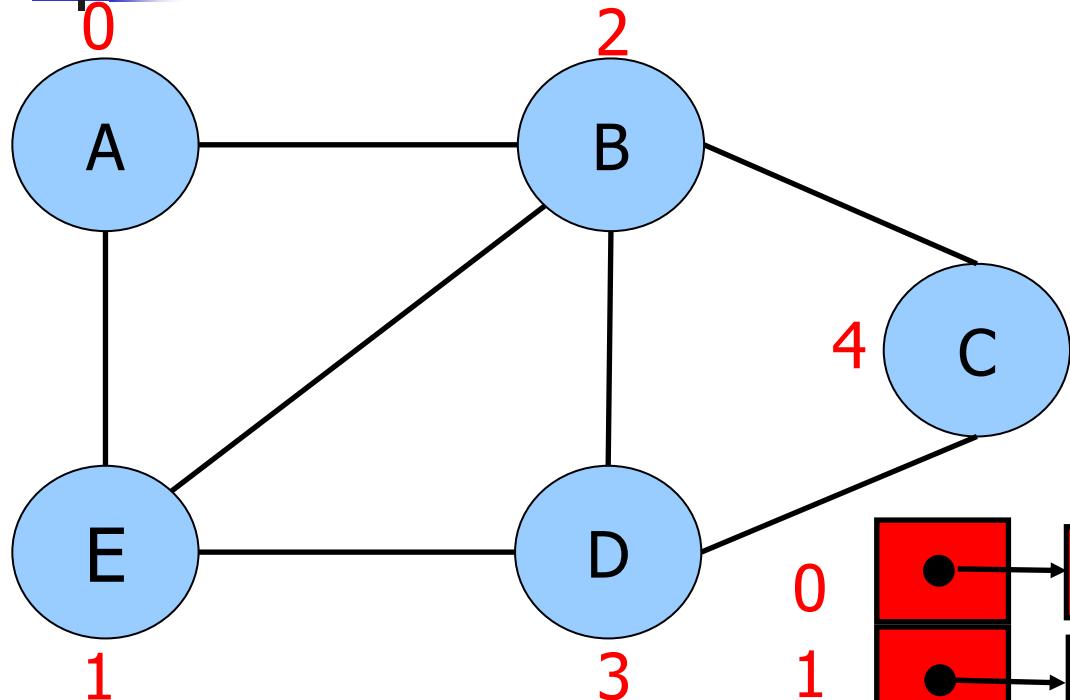
Conoscendo il numero di vertici, si può implementare con un vettore di liste, altrimenti lista di liste

## List di adiacenza

Dato  $G = (V, E)$ , la lista di adiacenza è:

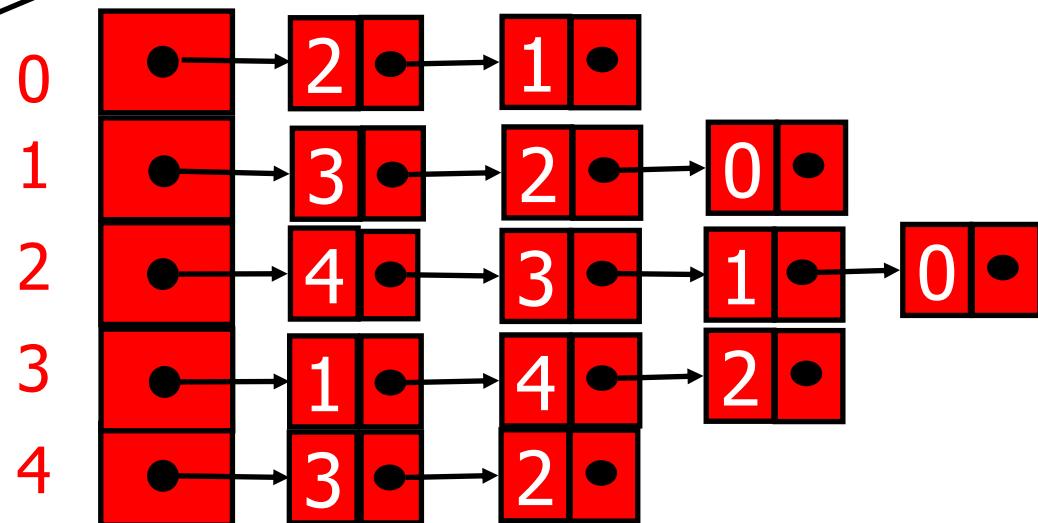
- un vettore  $A$  di  $|V|$  elementi. Il vettore è vantaggioso per via dell'accesso diretto
- $A[i]$  contiene il puntatore alla lista dei vertici adiacenti a  $i$ .

# Non orientato non pesato

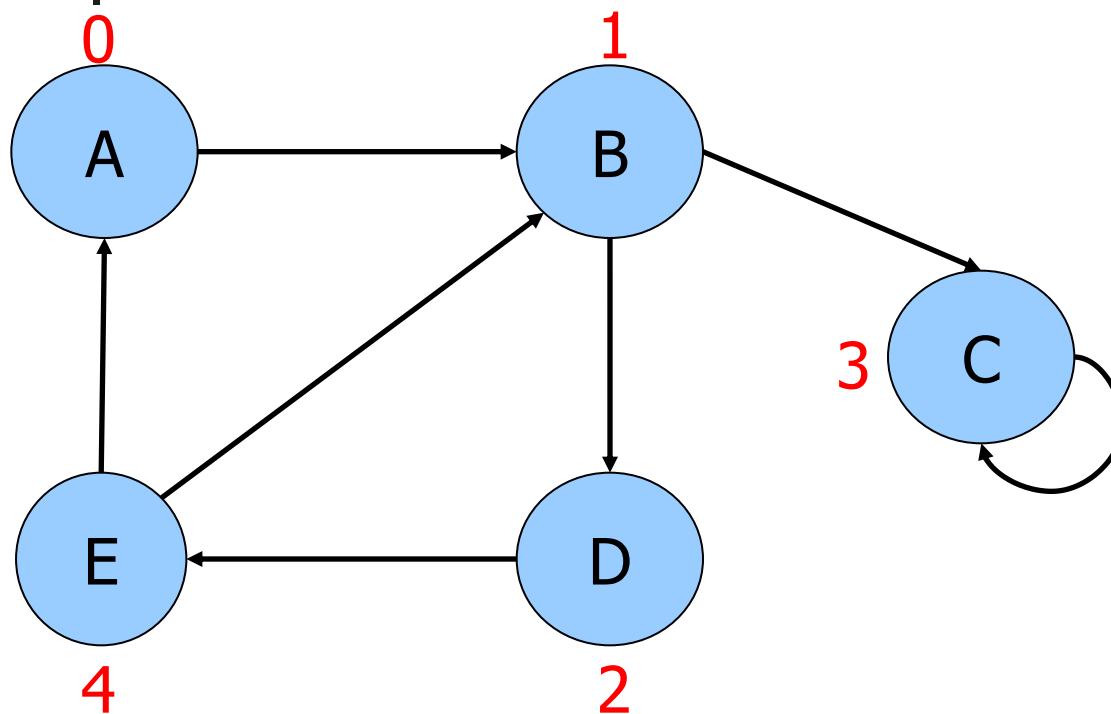


in.txt	ST
A E	0 A
A B	1 E
B E	2 B
B D	3 D
B C	4 C
C D	
D E	

G->adj



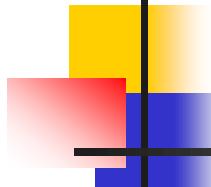
# Orientato non pesato



in.txt	ST
A	0
B	1
D	2
C	3
E	4

G->adj

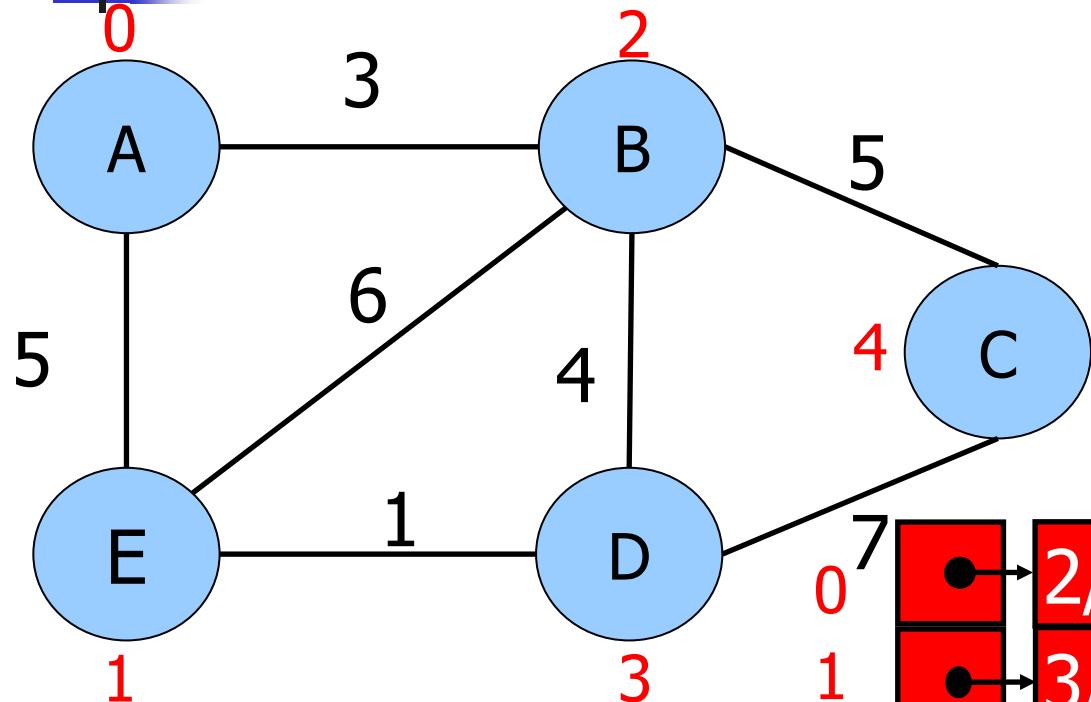
0	1	2	3	4
1	0	3	4	0
3	0	0	3	0
4	0	0	0	4
0	0	0	0	1



in.txt

ST	
0	A
1	E
2	B
3	D
4	C

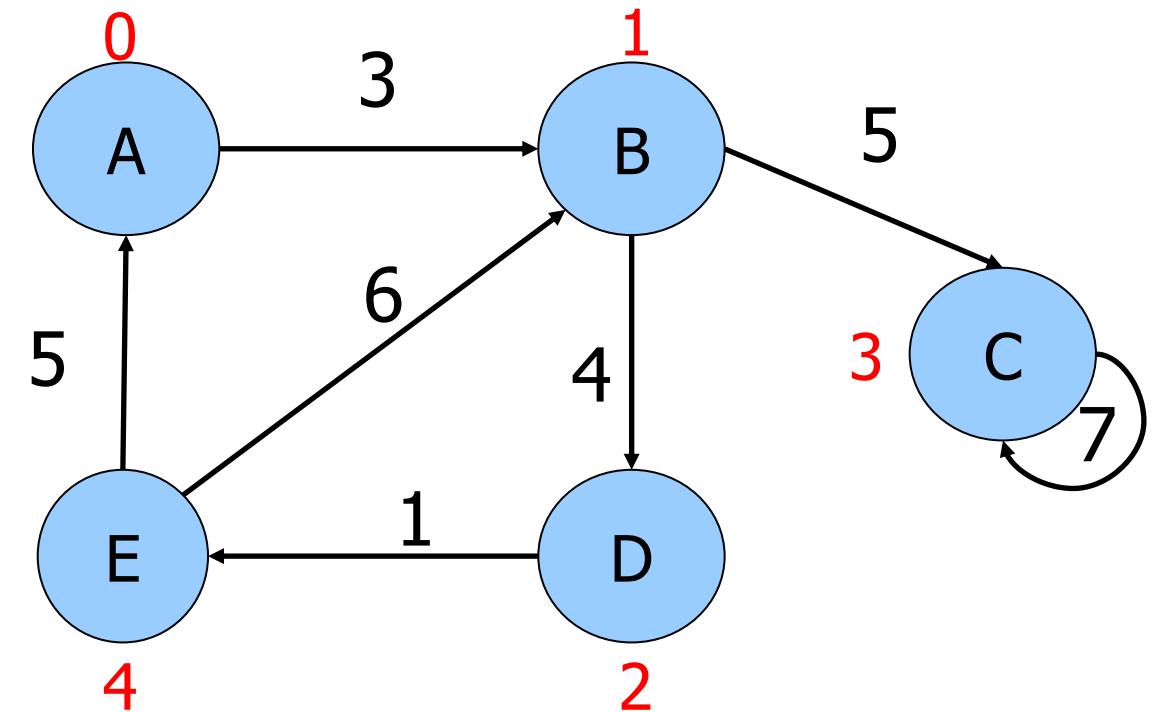
## Non orientato pesato



G->adj

0	0	2/3	1/5			
1	1	3/1	2/6	0/5		
2	2	4/5	3/4	1/6	0/3	
3	3	1/1	4/7	2/4		
4	4	3/7	2/5			

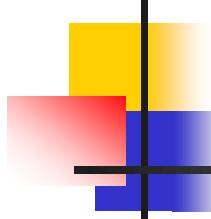
# Orientato pesato



	in.txt	ST
0	A B 3	A
1	B D 4	B
2	C C 7	D
3	B C 5	C
4	D E 1	E
	E B 6	
	E A 5	

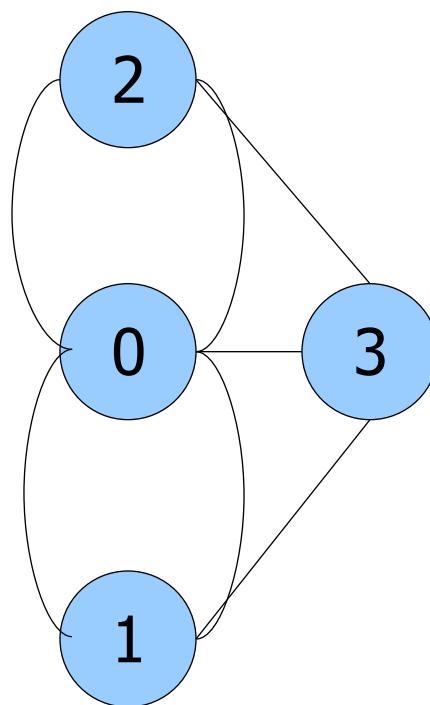
G->adj

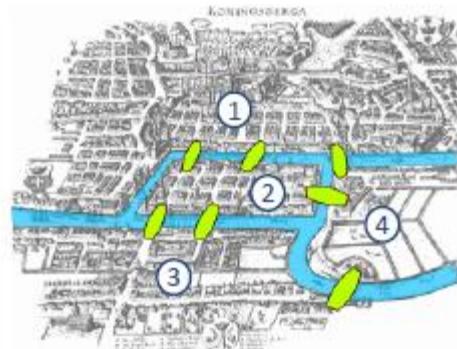
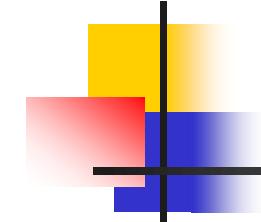
0	• →	1/3 •
1	• →	3/5 • → 2/4 •
2	• →	4/1 •
3	• →	3/7 •
4	• →	0/5 • → 1/6 •



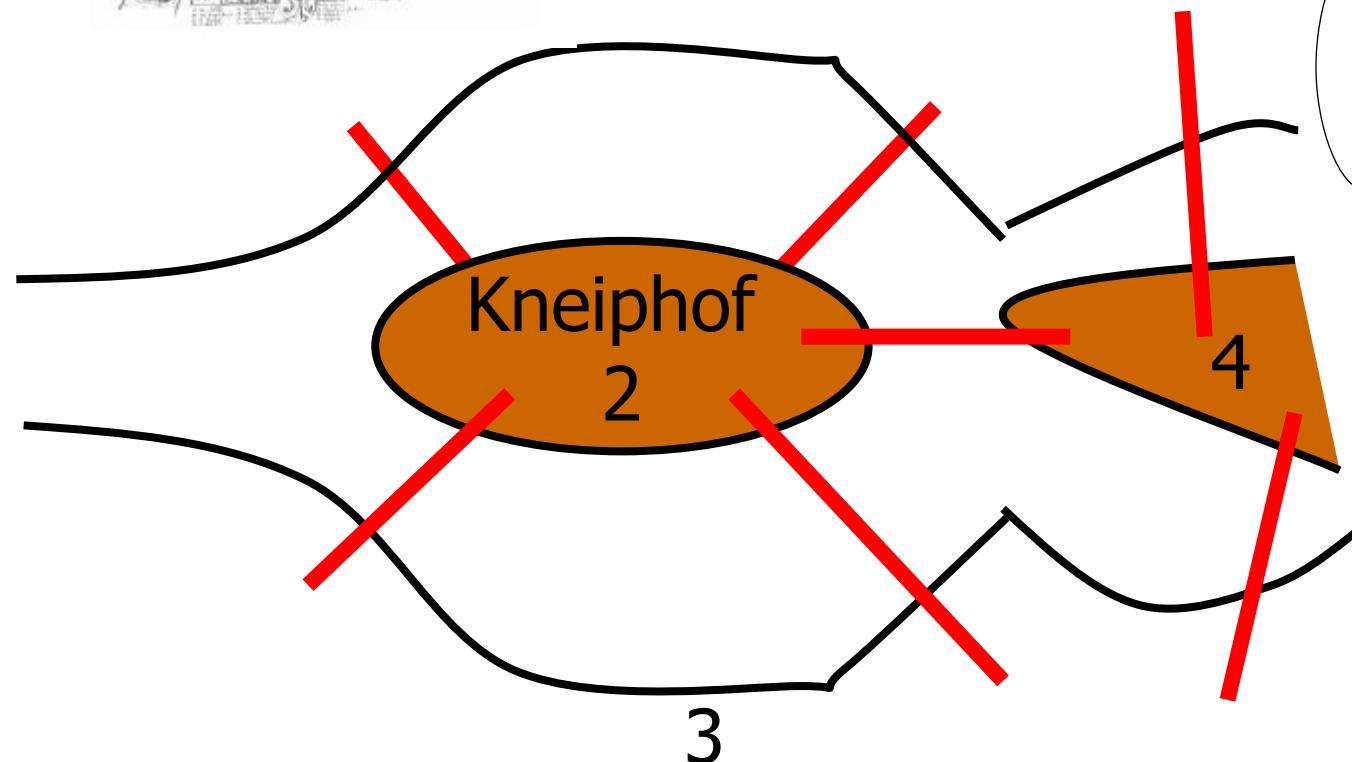
# Esempio: il multigrafo

Multigrafo: archi multipli che connettono la stessa coppia di vertici. Si può generare se in fase di inserzione degli archi non si scartano quelli già esistenti.

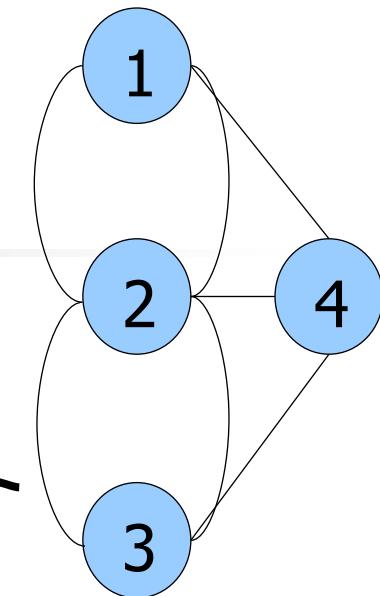




1



I ponti di Königsberg  
(Eulero, 1736)



## Graph.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "Graph.h"
#define maxv 100
#define MAX 10

typedef struct node *link;
struct node { int v; int wt; link next; };

struct graph{int v; int E; link *adj; ST tab; link z;};

link NEW(int v, link next) {
    link x = malloc(sizeof *x);
    x->v = v; x->next = next; x->wt = wt;
    return x;
}
```

Numero di vertici

Numero di archi

Lista

sentinella

ST

grafi pesati

```

Graph GRAPHinit(int v) {
    int v;
    Graph G = malloc(sizeof *G);
    if (G == NULL) return NULL;
    G->V = V; G->E = 0; G->z = NEW(-1, NULL);
    G->adj = malloc(G->V*sizeof(link));
    for (v = 0; v < G->V; v++)
        G->adj[v] = G->z;
    G->tab = STinit(v);
    if (G->tab == NULL) return NULL;
    return G;
}
void GRAPHedges(Graph G, Edge *a) {
    int v, E = 0;
    link t;
    for (v=0; v < G->V; v++)
        for (t=G->adj[v]; t != G->z; t = t->next)
            if (v < t->v) a[E++] = EDGEcreate(v, t->v);
}

```

# Attenzione: si possono generare sia cappi sia archi ripetuti!

```
void GRAPHshow(Graph G) {  
    int v;  
    link t;  
    printf("%d vertices, %d edges \n", G->V, G->E);  
    for (v=0; v < G->V; v++) {  
        printf("%s:\t", STretrieve(G->tab, v));  
        for (t=G->adj[v]; t != G->z; t = t->next)  
            printf("(%s %d)", STretrieve(G->tab, t->v), t->wt);  
        printf("\n");  
    }  
}  
  
void insertE(Graph G, Edge e) {  
    int v = e.v, w = e.w, wt = e.wt;  
    G->adj[v] = NEW(w, wt, G->adj[v]);  
    G->adj[w] = NEW(v, wt, G->adj[w]);  
    G->E++;  
}
```

grafi pesati

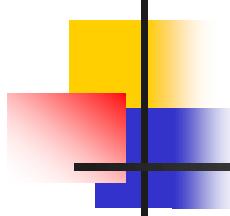
grafi non orientati

```

void removeE(Graph G, Edge e) {
    int v = e.v, w = e.w;
    link x;
    if (G->adj[v]->v == w) {
        G->adj[v] = G->adj[v]->next;
        G->E--;
    }
    else
        for (x = G->adj[v]; x != G->z; x = x->next)
            if (x->next->v == w) {
                x->next = x->next->next;
                G->E--;
            }
    if (G->adj[w]->v == v)
        G->adj[w] = G->adj[w]->next;
    else
        for (x = G->adj[w]; x != G->z; x = x->next)
            if (x->next->v == v) x->next = x->next->next;
}

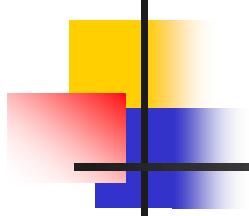
```

grafi non orientati



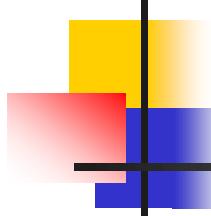
# Vantaggi/svantaggi

- Grafi non orientati:  
elementi complessivi nelle liste =  $2|E|$
- Grafi orientati:  
elementi complessivi nelle liste =  $|E|$
- Complessità spaziale  
 $S(n) = O(\max(|V|, |E|)) = O(|V+E|)$   
 $\Rightarrow$  vantaggioso per grafi sparsi



## ■ Svantaggi:

- verifica dell'esistenza di arco  $(v,w)$  mediante scansione della lista di adiacenza di  $v$
- uso di memoria per i pesi dei grafi pesati.



# Generazione dei grafi

## Tecnica 1: archi casuali

- Vertici come interi tra 0 e  $|V|-1$
- generazione di un grafo casuale a partire da  $E$  coppie casuali di archi (interi tra 0 e  $|V|-1$ )
  - possibili archi ripetuti (multigrafo) e cappi
  - grafo con  $|V|$  vertici e  $|E|$  archi (inclusi cappi e archi ripetuti)

Tecnica 2: archi con probabilità  $p$

- si considerano tutti i possibili archi  $|V|^*(|V|-1)/2$
- tra questi si selezionano quelli con probabilità  $p$
- $p$  è calcolato in modo che sia

$$|E| = p * (|V|^*(|V|-1)/2)$$

quindi

$$p = 2 * |E| / (|V| * (|V| - 1))$$

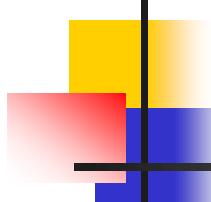
- si ottiene un grafo con in media  $|E|$  archi
- non ci sono archi ripetuti.

```

int randv(Graph G) {
    return G->V * (rand() / (RAND_MAX + 1.0));
}
Graph GRAPHrand1(Graph G, int v, int E) {
    while (G->E < E)
        insertE(G, EDGEcreate(randv(G), randv(G)));
    return G;
}

Graph GRAPHrand2(Graph G, int v, int E) {
    int i, j;
    double p = 2.0 * E / (v * (v-1));
    for ( i = 0; i < v; i++)
        for ( j = i+1; j < v; j++)
            if (rand() < p * RAND_MAX)
                insertE(G, EDGEcreate(i, j));
    return G;
}

```



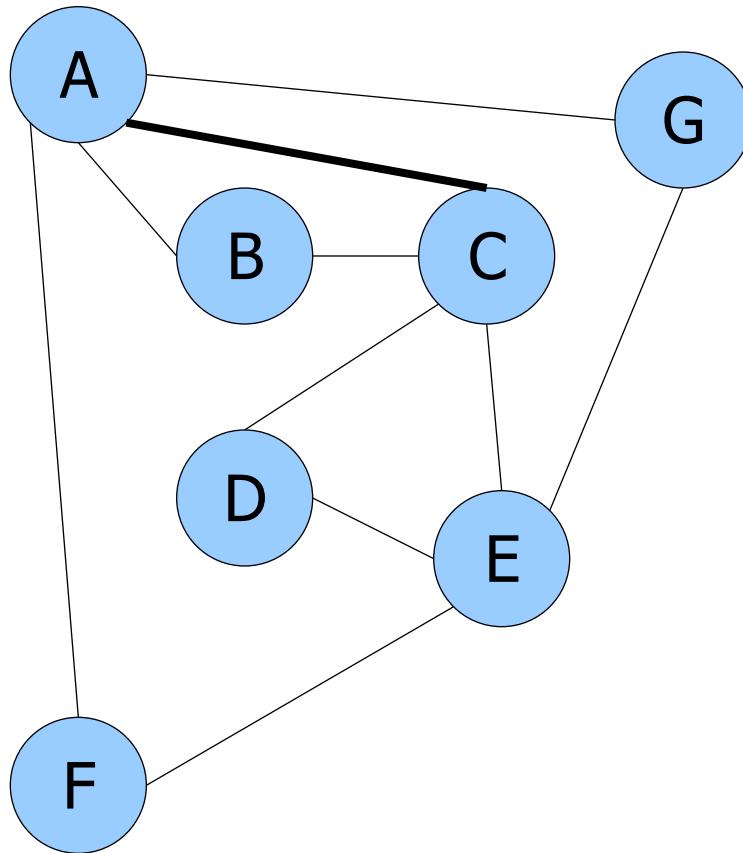
# Cammino semplice

Dato un grafo non orientato  $G = (V, E)$  e 2 suoi vertici  $v$  e  $w$ , esiste un cammino semplice che li connette?

$\exists p: v \rightarrow_p w$

- $\forall$  vertice  $t$  adiacente al vertice corrente  $v$ , determinare ricorsivamente se esiste un cammino semplice da  $t$  a  $w$
- array `visited[maxv]` dichiarato come static in `Graph.c` per marcare i nodi già visitati
- cammino visualizzato in ordine inverso
- complessità  $T(n) = O(|V+E|)$

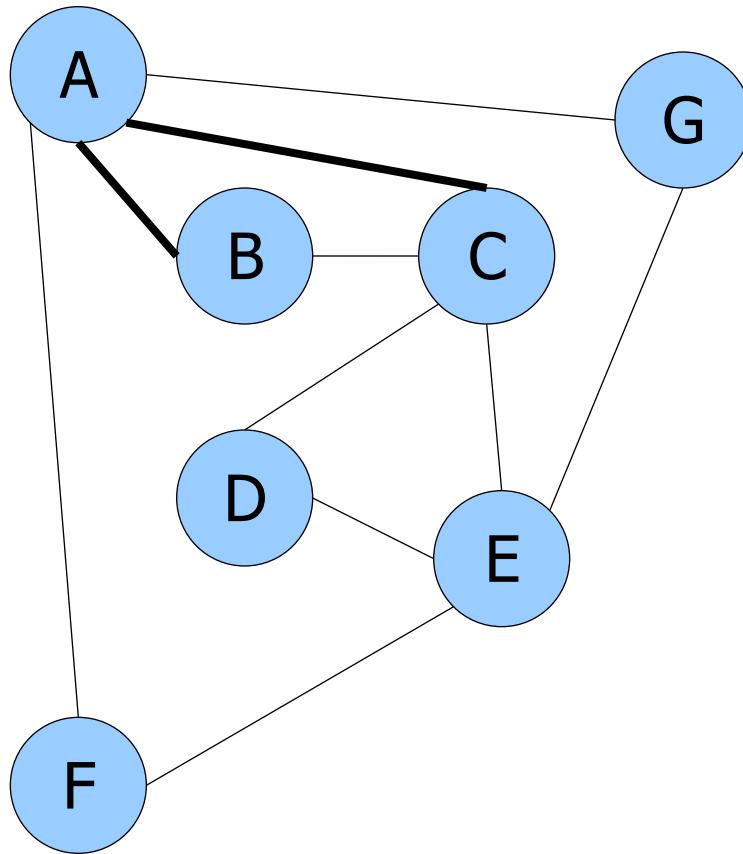
# Esempio



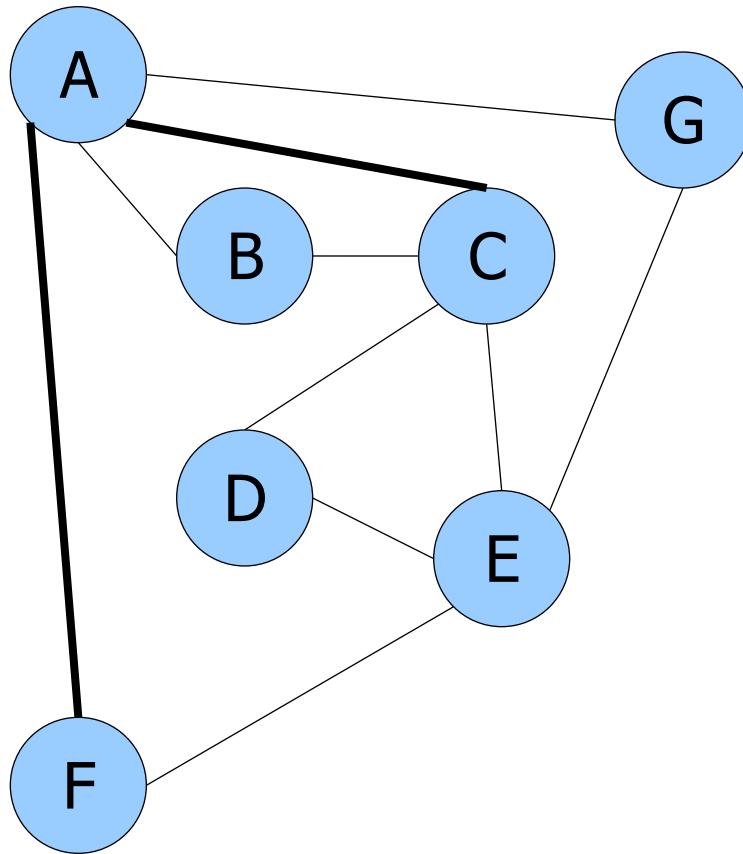
$\exists p: C \xrightarrow{p} G?$

passo 1:

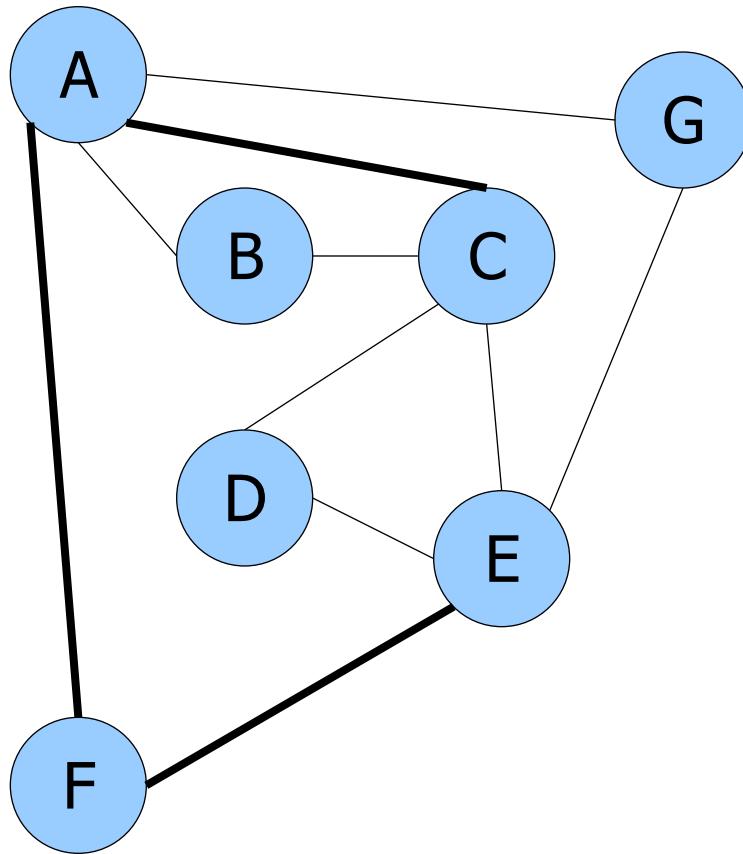
vertici adiacenti a C  
non ancora visitati:  
A, B, D, E  
seleziono A



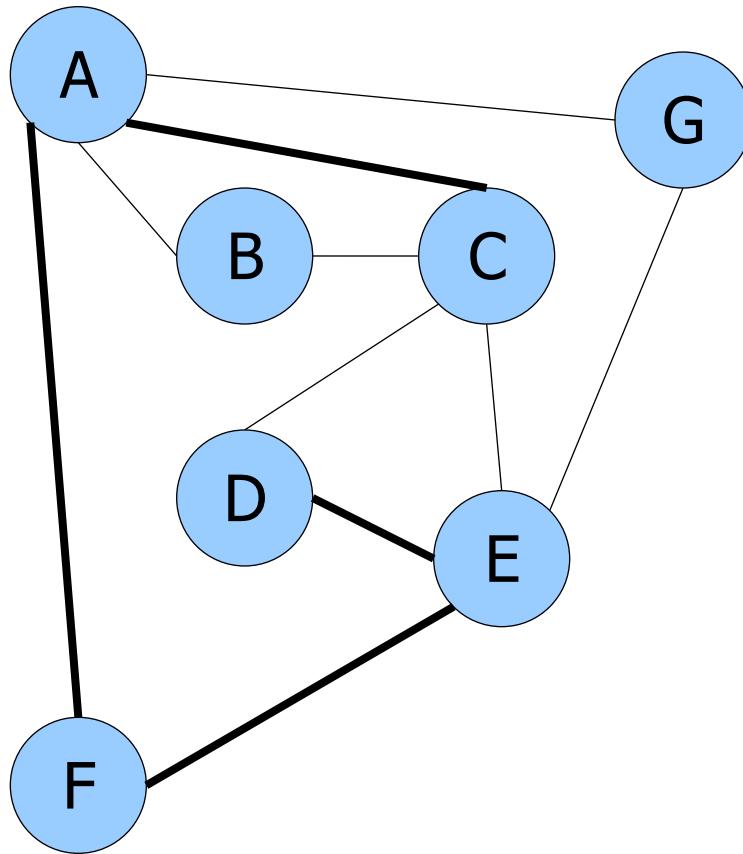
$\exists p: A \xrightarrow{p} G?$   
passo 2:  
vertici adiacenti a A  
non ancora visitati:  
B, F, G  
seleziono B



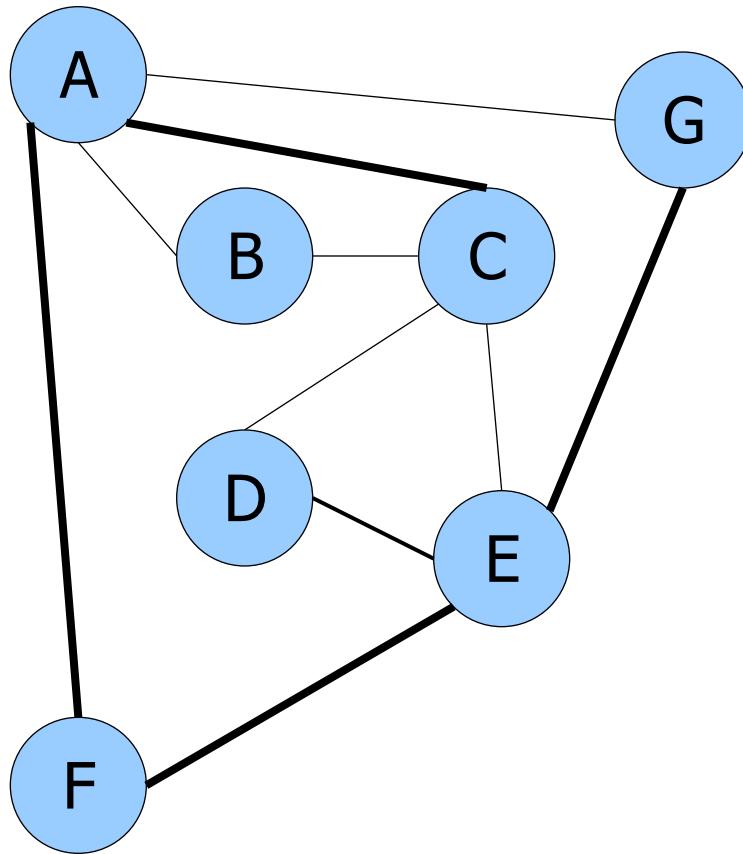
$\exists p: B \rightarrow_p G?$   
passo 3:  
vertici adiacenti a B  
non ancora visitati:  
nessuno  
La ricorsione torna a  
A e seleziona F



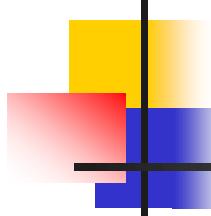
$\exists p: F \rightarrow_p G?$   
passo 4:  
vertici adiacenti a F  
non ancora visitati:  
E  
seleziono E



$\exists p: E \rightarrow_p G?$   
passo 5:  
vertici adiacenti a E  
non ancora visitati:  
D, G  
seleziono D



$\exists p: D \rightarrow_p G?$   
passo 6:  
vertici adiacenti a D  
non ancora visitati:  
nessuno  
La ricorsione torna a  
E e seleziona G

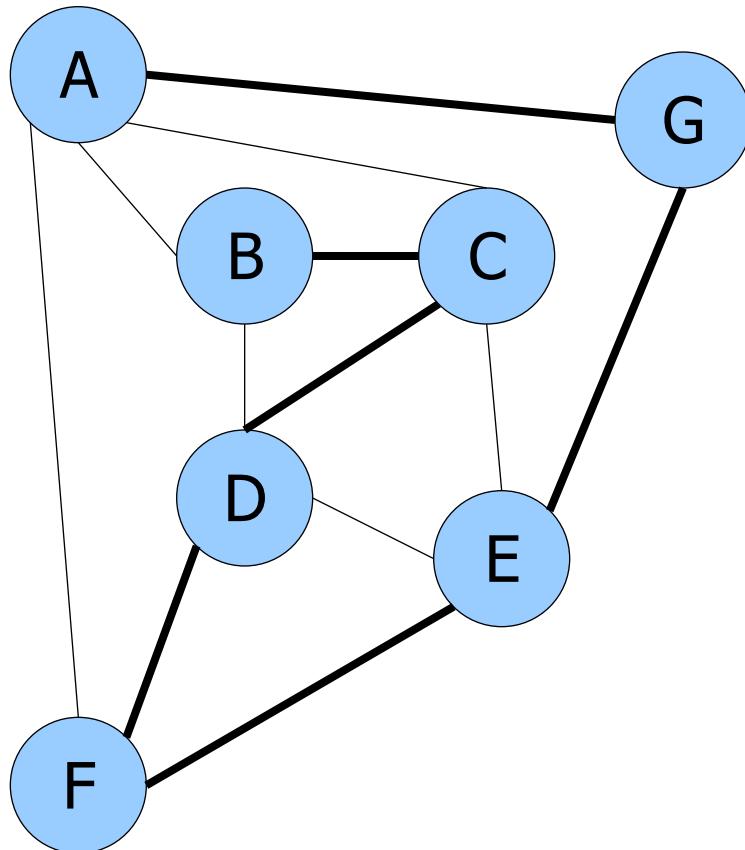


# Cammino di Hamilton

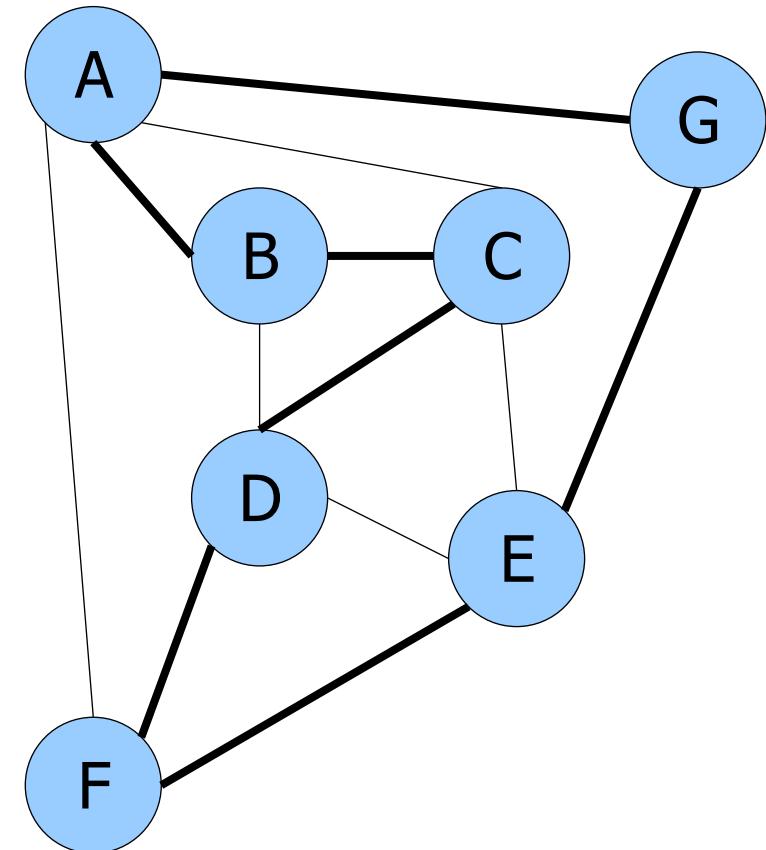
Dato un grafo non orientato  $G = (V, E)$  e 2 suoi vertici  $v$  e  $w$ , se esiste un cammino semplice che li connette visitando ogni vertice una e una sola volta, questo si dice **cammino di Hamilton**.

Se  $v$  coincide con  $w$ , si parla di **ciclo di Hamilton**.

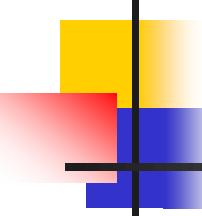
# Esempio



Cammino di Hamilton tra A e B



Ciclo di Hamilton



# Algoritmo

Cammino di Hamilton tra v e w:

- ∀ vertice t adiacente al vertice corrente v, determinare ricorsivamente se esiste un cammino semplice da t a w
- ritorno con successo se e solo se la lunghezza del cammino è  $|V|-1$
- set della cella dell'array visited per marcare i nodi già visitati
- reset della cella dell'array visited quando ritorna con insuccesso (backtrack)
- complessità **esponenziale!**

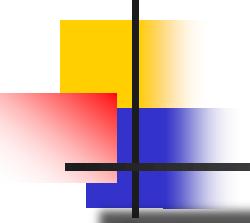
```

int GRAPHpath/GRAphpathH(Graph G) {
    int id1, id2, t, found, *visited;
    char src[MAX], dst[MAX];
    visited = malloc(G->v*sizeof(int));
    for (t = 0 ; t < G->v ; t++)
        visited[t] = 0;
    printf("Insert first node = "); sca
    printf("Insert second node = "); sc
    id1 = STsearch(G->tab, src);
    id2 = STsearch(G->tab, dst);

    if (id1 < 0 || id2 < 0)
        return 0;
    found = pathR/pathRH(G, id1, id2, visited);
    if (found == 0)
        printf("\n Path not found!\n");
    return found;
}

```

Attenzione: per  
sinteticità si viola  
la sintassi del C



Matrice delle  
adiacenze

```
int pathR(Graph G, int v, int w, int *visited) {  
    int t;  
    if (v == w)  
        return 1;  
  
    visited[v] = 1;  
  
    for (t = 0 ; t < G->v ; t++)  
        if (G->adj[v][t] == 1)  
            if (visited[t] == 0)  
                if (pathR(G, t, w, visited)) {  
                    printf("(%s, %s) in path\n", STretrieve(G->tab, v),  
                           STretrieve(G->tab, t));  
                    return 1;  
                }  
    return 0;  
}
```

Stampa gli archi  
del cammino in  
ordine inverso

```

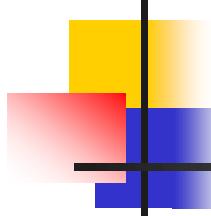
int pathRH(Graph G, int v, int w, int d, int *visited) {
    if (v == w) {
        if (d == 0) return 1;
        else return 0;
    }
    visited[v] = 1;
    for (t = 0; t < G->V; t++)
        if (G->adj[v][t] == 1)
            if (visited[t] == 0)
                if (pathRH(G, t, w, d-1, visited)) {
                    printf("(%s, %s) in path \n",
                           STretrieve(G->tab, v),
                           STretrieve(G->tab, t));
                    return 1;
                }
    visited[v] = 0;
    return 0;
}

```

Matrice delle  
adiacenze

Intero che indica  
quanto manca a un  
cammino lungo  $|V|-1$

Stampa gli archi  
del cammino in  
ordine inverso



# Cammino di Eulero

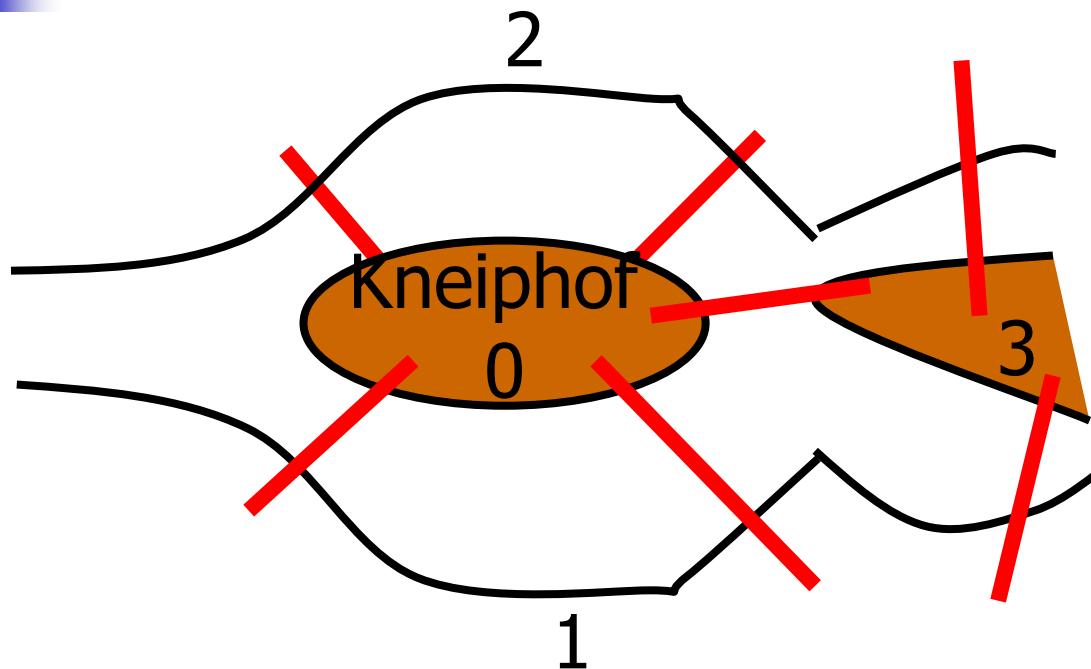


Dato un grafo non orientato  $G = (V, E)$  e 2 suoi vertici  $v$  e  $w$ , si dice **cammino di Eulero** un cammino (anche non semplice) che li connette attraversando ogni arco una e una sola volta.

Se  $v$  coincide con  $w$ , si parla di **ciclo di Eulero**.

- Un grafo non orientato ha un ciclo di Eulero se e solo se è connesso e tutti i suoi vertici sono di grado pari
- Un grafo non orientato ha un cammino di Eulero se e solo se è connesso e se esattamente due vertici hanno grado dispari.

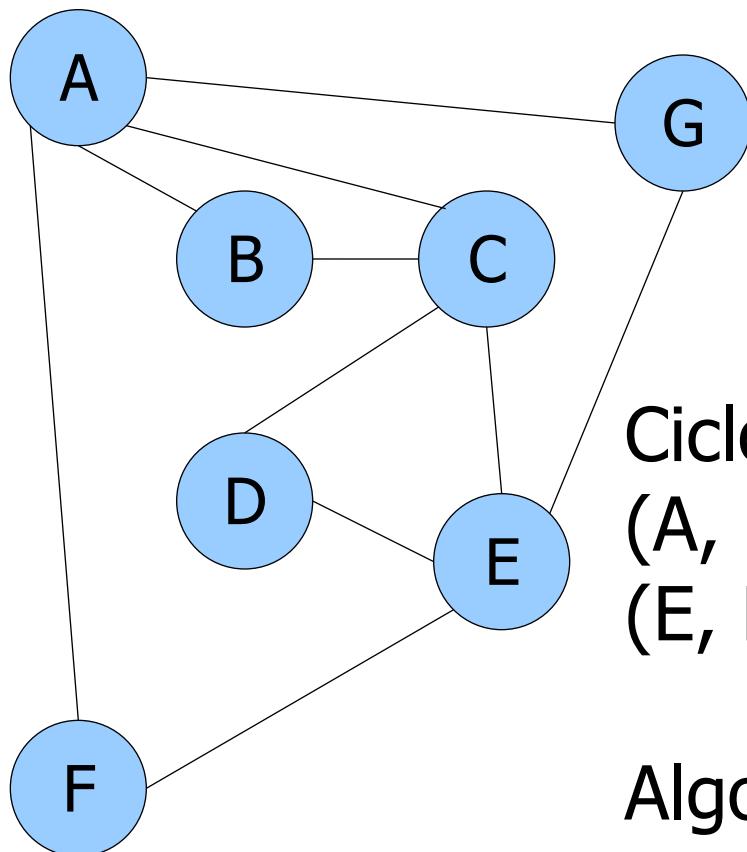
# I ponti di Königsberg



$\nexists$  né cammini, né cicli di Eulero

Multigrafo: archi multipli che connettono la stessa coppia di vertici

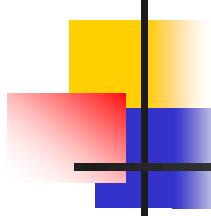
# Esempio: verifica della esistenza di ciclo di Eulero



nodo	deg
A	4
B	2
C	4
D	2
E	4
F	2
G	2

Ciclo di Eulero:  
(A, B), (B, C), (C, A), (A, G), (G, E)  
(E, D), (D, C), (C, E), (E, F), (F, G)

Algoritmo di complessità  $O(|E|)$



# Riferimenti

---

- L'ADT grafo non orientato:
  - Sedgewick Part 5: 17.2
- Rappresentazione dei grafi:
  - Sedgewick Part 5: 17.3, 17.4
  - Cormen 23.1
- Generazione di grafi:
  - Sedgewick Part 5: 17.6
- Cammini semplici, di Hamilton, di Eulero:
  - Sedgewick Part 5: 17.6
  - Cormen 36.2, 36.5.4