POLITECNICO
DI TORINO
Dipartimento
di Automatica e Informatica

L’ADT Grafo

Gianpiero Cabodi e Paolo Camurati

Perché la Teoria dei Grafi?

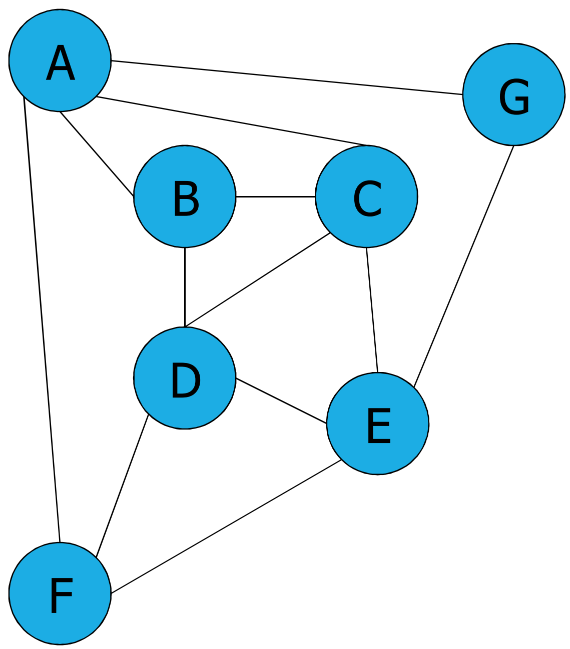
- Moltissime applicazioni pratiche
- Centinaia di algoritmi
- Astrazione interessante e utilizzabile in molti domini diversi
- Ricerca attiva in Informatica e Matematica discreta.

Complessità dei problemi

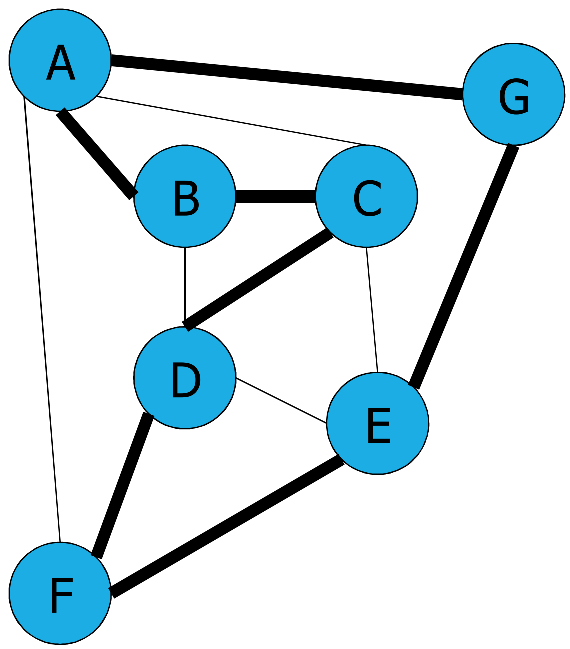
- problemi facili:
  - determinare se un grafo è connesso
  - determinare la presenza di un ciclo
  - individuare le componenti fortemente connesse
  - individuare gli alberi minimi ricoprenti
  - calcolare i cammini minimi
  - determinare se un grafo è bipartito
  - trovare un cammino di Eulero
- problemi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

- problemi intrattabili:
  - cammini a lunghezza massima
  - colorabilità
  - clique massimale
  - ciclo di Hamilton
  - problema del commesso viaggiatore
- problemi di complessità ignota: isomorfismo di due grafi F e G:
  - isomorfismo fra *G* e *H:* applicazione biiettiva *f* dai vertici di *G* ai vertici di *H* tale che vi sia un arco dal vertice *u* al vertice *v in G se e solo se* c'è un arco dal vertice *f*(*u*) al vertice *f*(*v*) in *H*
  - non esiste un algoritmo polinomiale, ma non è stato provato che il problema è NP-completo.

Ciclo di Hamilton



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


A
G
B
C
D
E
F

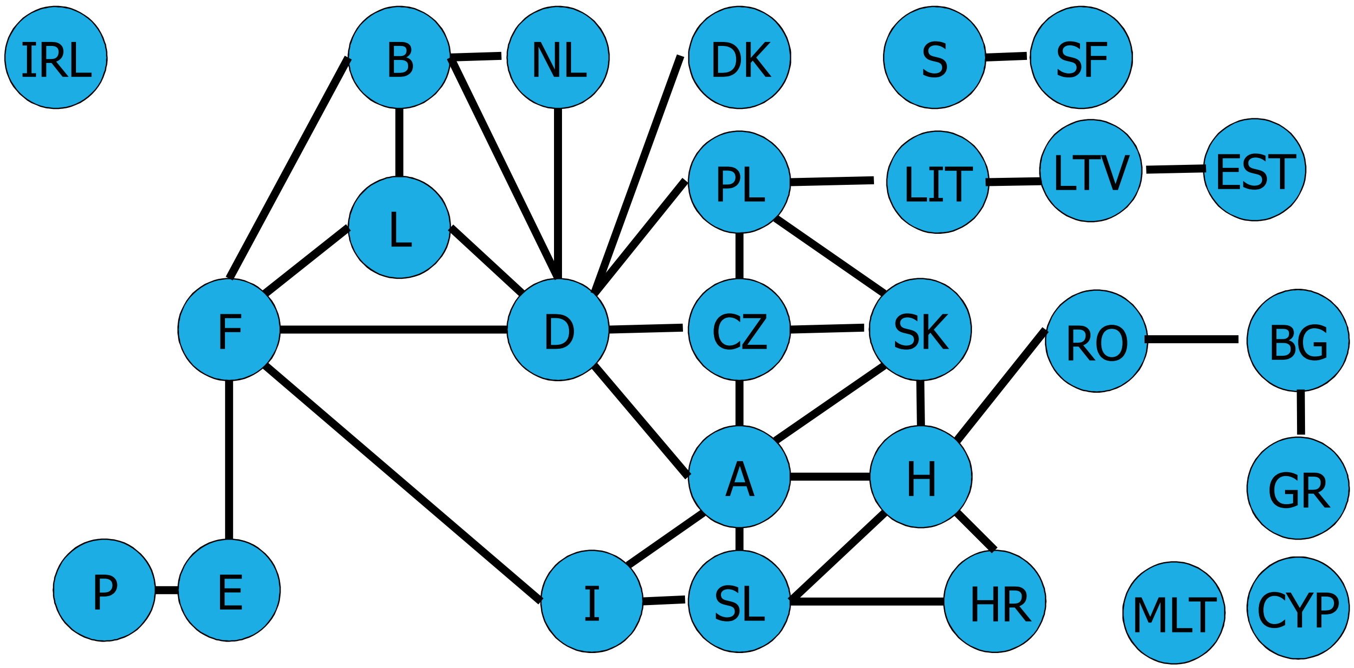
Colorabilità

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.

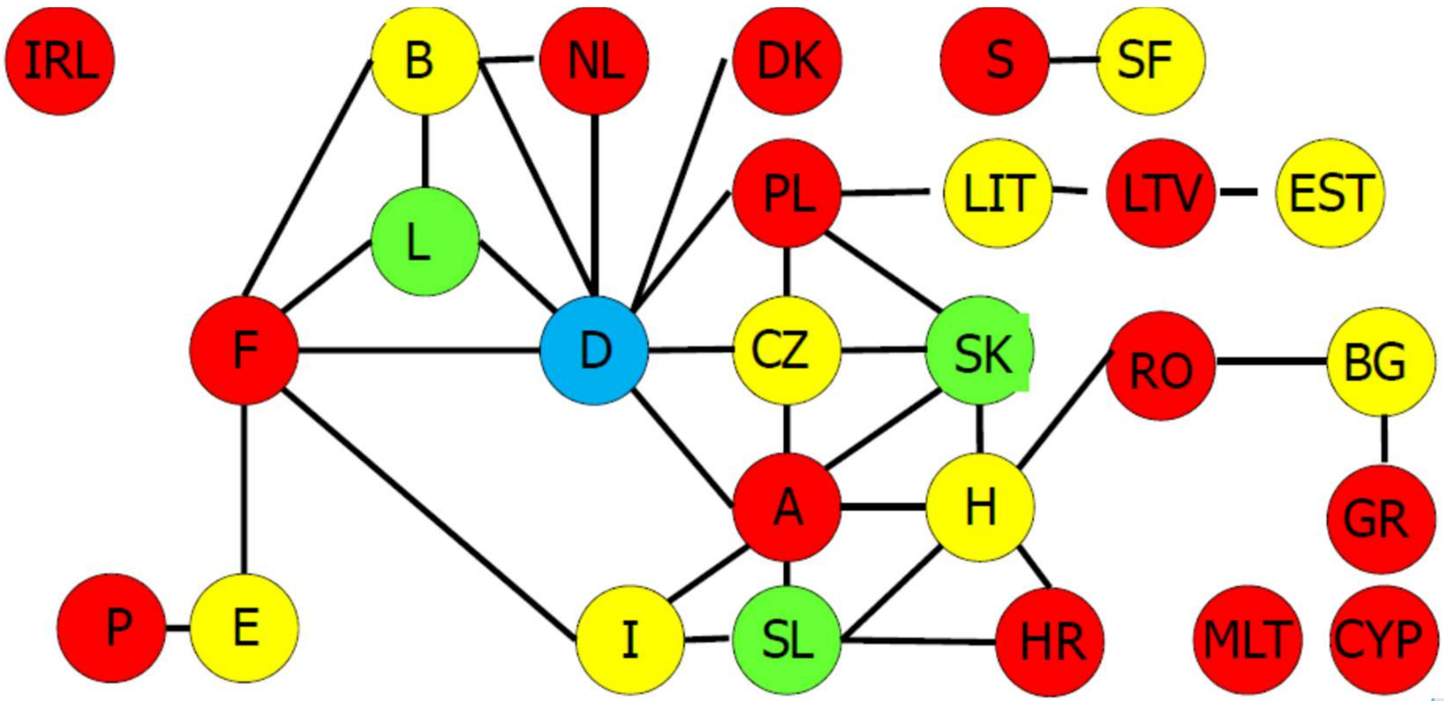
Le mappe geografiche sono modellate come grafi planari.

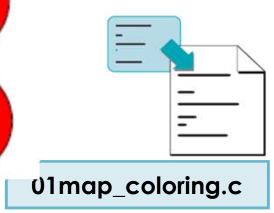
L’UE a 27 stati



Colorabilità: k=4

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

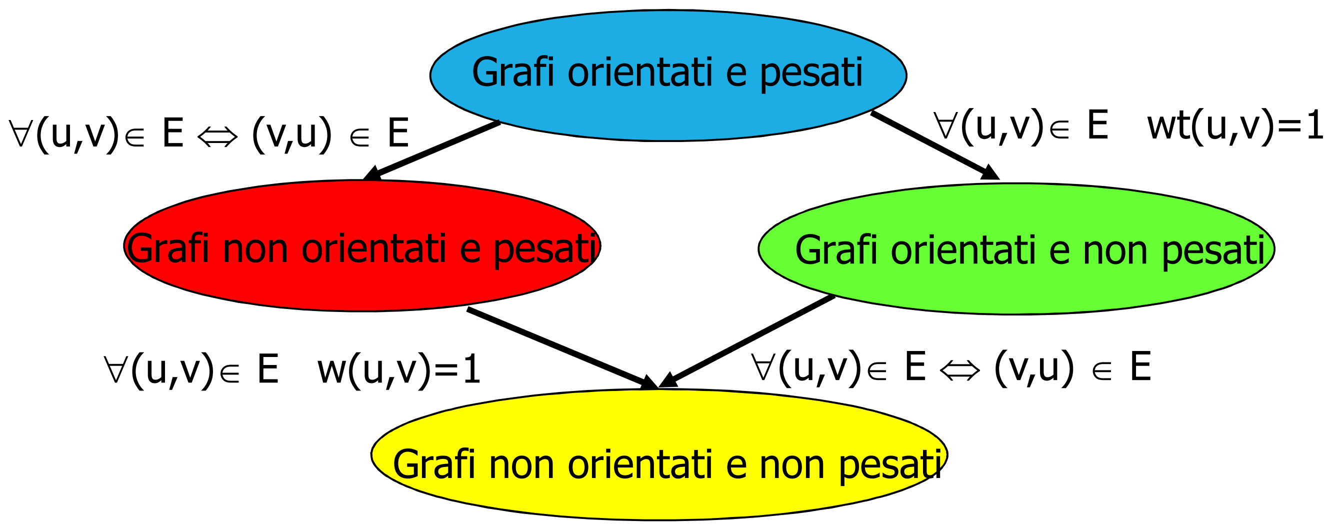




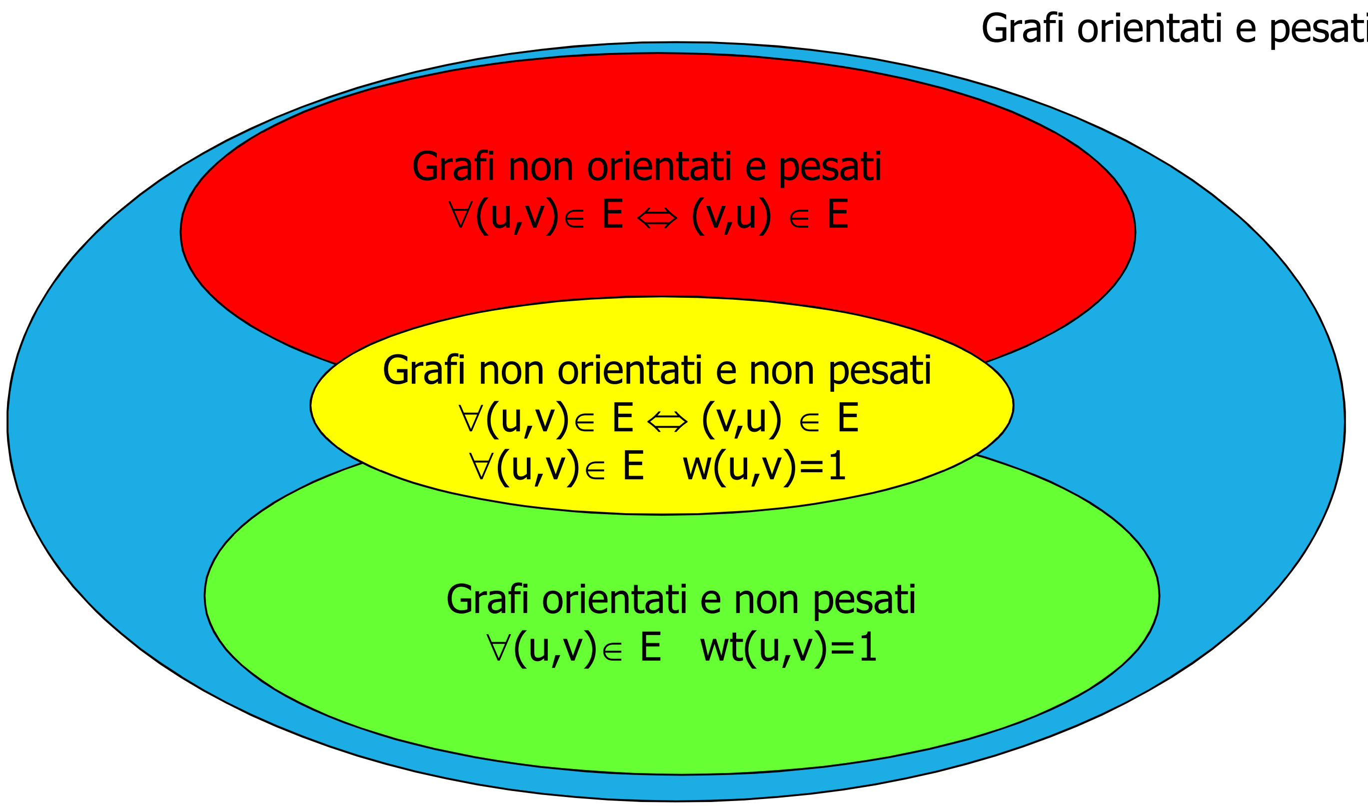
L’ADT Grafo

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

Tipologie (visione a grafo)



Tipologie (visione a insieme)



Ipotesi

- Il modello più generale è il grafo orientato e pesato. Per efficienza si implementano ADT specifici per le 4 tipologie
- Dopo l’inizializzazione:
  - grafi statici: non si aggiungono né si cancellano né vertici né archi
  - grafi semi-statici: non si aggiungono né si cancellano vertici, si possono aggiungere o cancellare archi
  - grafi dinamici: si possono aggiungere e cancellare sia vertici, sia archi
- Nel Corso si considerano solo grafi semi-statici, in cui i vertici vengono cancellati «logicamente», aggiungendo un campo per marcare se è cancellato o meno.

Vertici: informazioni rilevanti:

- interi per identificarli ai fini dell'algoritmo
- stringhe per identificarli ai fini dell'utente memorizzate in tabelle di simboli:
  - esterne al grafo
  - interne al grafo

  cui si accede mediante 2 chiavi (intero e stringa).

Tabella di simboli con funzioni:

- `STsearch` da chiave (nome) a intero (indice)
- `STsearchByIndex` da chiave (indice) a stringa (nome)

Possibili soluzioni:

- estensione della tabella di simboli se implementata con vettore con la `STsearchByIndex`
- tabella standard (BST, hash table) e vettore a lato per la ricerca da indice a chiave

- possibili implementazioni:
  - tabella di simboli estesa: vettore non ordinato di stringhe: l'indice del vettore coincide con l'indice del vertice che non è memorizzato esplicitamente. Ricerca inversa con scansione lineare
  - BST o tabella di hash: l'indice del vertice è memorizzato esplicitamente. `STsearchByIndex` con scansione lineare di un vettore di corrispondenza indice-chiave
- negli esempi: tabella di simboli estesa interna all'ADT grafo realizzata come vettore non ordinato con indice del vertice coincidente con quello del vettore
- nei laboratori: tutte le tipologie.

- 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 sovrastima 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 nella tabella di simboli, ottenendo il numero di vertici distinti |V| esatto e la corrispondenza nome del vertice – indice.

- Formato del file di input:
  - numero V di vertici sulla prima riga
  - V righe con ciascuna il nome di un vertice
  - numero indefinito di righe con coppie vertice-vertice per gli archi (con peso se grafo pesato).
- con file in formato standard:
  - `GRAPHload`/`GRAPHstore` di libreria
  - altrimenti `fileRead` ad hoc nel `main`.

- Il grafo è un ADT di I classe
- Gli archi sono definiti con `typedef` in `Graph.h` e sono quindi visibili anche al client
- La tabella di simboli è un ADT di I classe
- Altre eventuali collezioni di dati sono ADT di I classe (code, code a priorità, etc.)

ADT di I classe Grafo

Graph.h

```
typedef struct edge { int v; int w; int wt; } Edge;
typedef struct graph *Graph;
Graph GRAPHinit(int V);
void  GRAPHfree(Graph G);
void  GRAPHload(FILE *fin);
void  GRAPHstore(Graph G, FILE *fout);
int   GRAPHgetIndex(Graph G, char *label);
void  GRAPHinsertE(Graph G, int id1, int id2; int wt; );
void  GRAPHremoveE(Graph G, int id1, int id2);
void  GRAPHedges(Graph G, Edge *a);
int   GRAPHpath(Graph G, int id1, int id2);
void  GRAPHpathH(Graph G, int id1, int id2);
void  GRAPHbfs(Graph G, int id);
void  GRAPHdfs(Graph G, int id);
int   GRAPHcc(Graph G);
int   GRAPHscc(Graph G);
```

grafi pesati

indice dato nome

grafi pesati

grafi non orientati

grafi orientati

Lettura/scrittura di un grafo da/su file

- Se il file contiene la lista degli archi in formato «standard» ha senso offrire `GRAPHload`/ `GRAPHstore`. In alternativa il `client` implementa la sua funzione di lettura/scrittura
- è già noto il numero di vertici |V|
- lettura dei vertici e inserzione nella tabella di simboli
- lettura degli archi e inserzione nel grafo

- Scrittura: si scandiscono gli archi:
  - con `GRAPHedges` si accumulano gli archi in un vettore
  - per ciascuno dei vertici su cui l'arco insiste, tramite la tabella di simboli, dato l'indice si ricava il nome.

```
Graph GRAPHload(FILE *fin) {
  int V, i, id1, id2, wt;
  char label1[MAXC], label2[MAXC];
  Graph G;
  fscanf(fin, "%d", &V);
  G = GRAPHinit(V);
  for (i=0; i<V; i++) {
    fscanf(fin, "%s", label1);
    STinsert(G->tab, label1, i);
  }
  while(fscanf(fin,"%s %s %d", label1, label2, &wt) == 3) {
    id1 = STsearch(G->tab, label1);
    id2 = STsearch(G->tab, label2);
    if (id1 >= 0 && id2 >=0)
      GRAPHinsertE(G, id1, id2, wt);
  }
  return G;
}
```

grafi pesati

```
void GRAPHstore(Graph G, FILE *fout) {
  int i;
  Edge *a;

  a = malloc(G->E * sizeof(Edge));

  GRAPHedges(G, a);

  fprintf(fout, "%d\n", G->V);
  for (i = 0; i < G->V; i++)
    fprintf(fout, "%s\n", STsearchByIndex(G->tab, i));
  for (i = 0; i < G->E; i++)
    fprintf(fout, "%s  %s %d\n",
            STsearchByIndex(G->tab, a[i].v),
            STsearchByIndex(G->tab, a[i].w), a[i].wt);
}
```

dipende dalla rappresentazione

grafi pesati

Inserzione/rimozione archi

dipende dalla rappresentazione

```
void GRAPHinsertE(Graph G, int id1, int id2, int wt) {
  insertE(G, EDGEcreate(id1, id2, wt));
}

void GRAPHremoveE(Graph G, int id1, int id2) {
  removeE(G, EDGEcreate(id1, id2, 0));
}
```

grafi pesati

Rappresentazione

Matrice di adiacenza

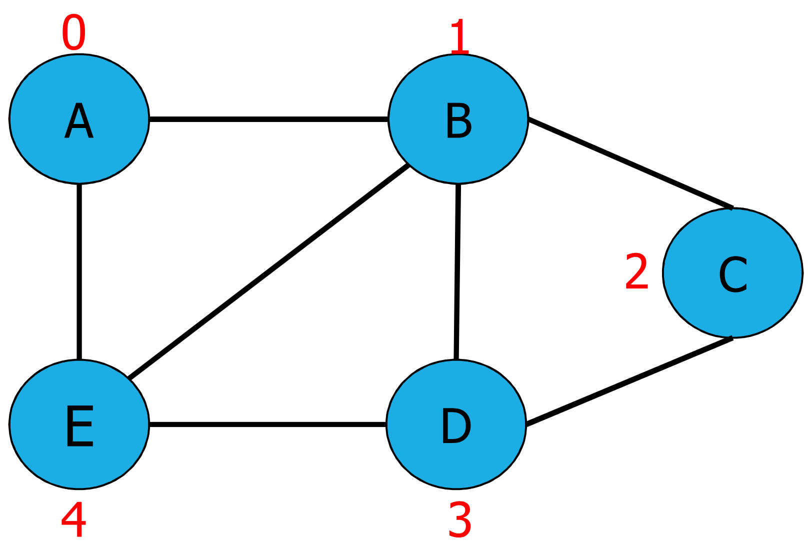
Dato G = (V, E), la matrice di adiacenza è:

- matrice adj di |V| x |V| elementi

$$adj[i,j] = \begin{cases} 1 \text{ se } (i, j) \in E \\ 0 \text{ se } (i, j) \notin E \end{cases}$$

- grafi non orientati: adj simmetrica
- grafi pesati: adj[i,j]=peso dell'arco (i,j) se esiste, 0 non è un peso ammesso.

Non orientato non pesato



Lista archi

A	B
B	C
B	D
A	E
C	D
B	E
D	E

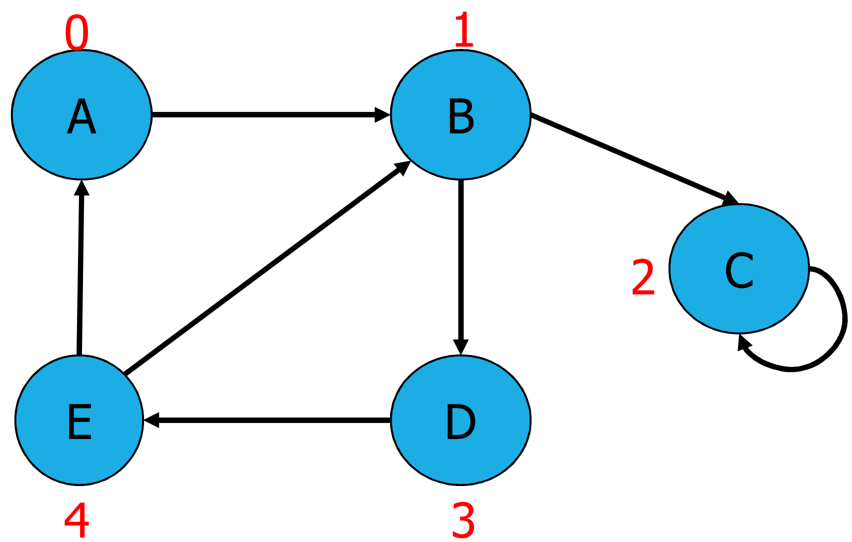
ST

0	A
1	B
2	C
3	D
4	E

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

Orientato non pesato



Lista archi

A B
B C
C C
B D
D E
E B
E A

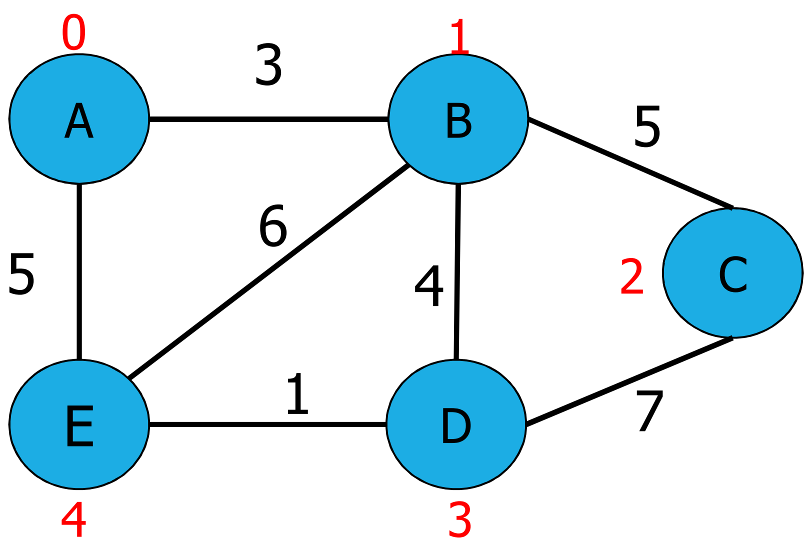
ST

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



Lista archi

A B 3
B C 5
B D 4
A E 5
C D 7
B E 6
D E 1

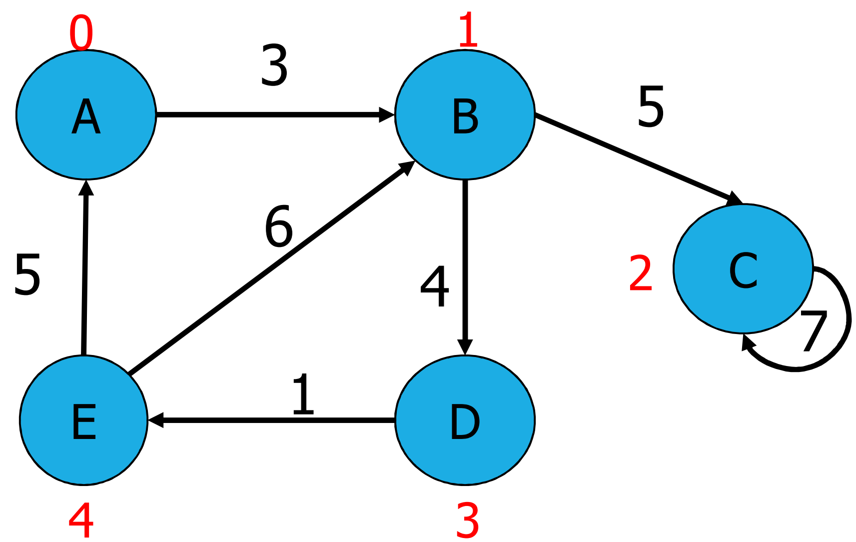
ST

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



Lista archi

A B 3
B C 5
B D 4
E A 5
E B 6
C C 7
D E 1

ST

0	A
1	B
2	C
3	D
4	E

G->adj

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

Graph.c

numero di archi

matrice di adiacenza

tabella di simboli

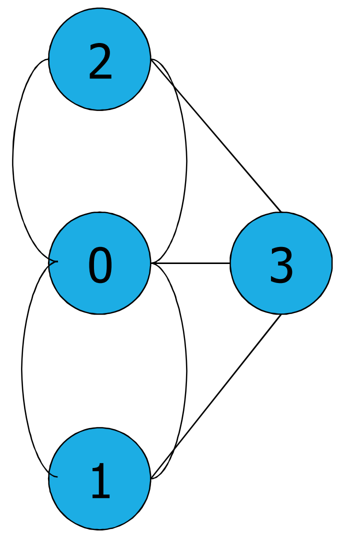
numero di vertici

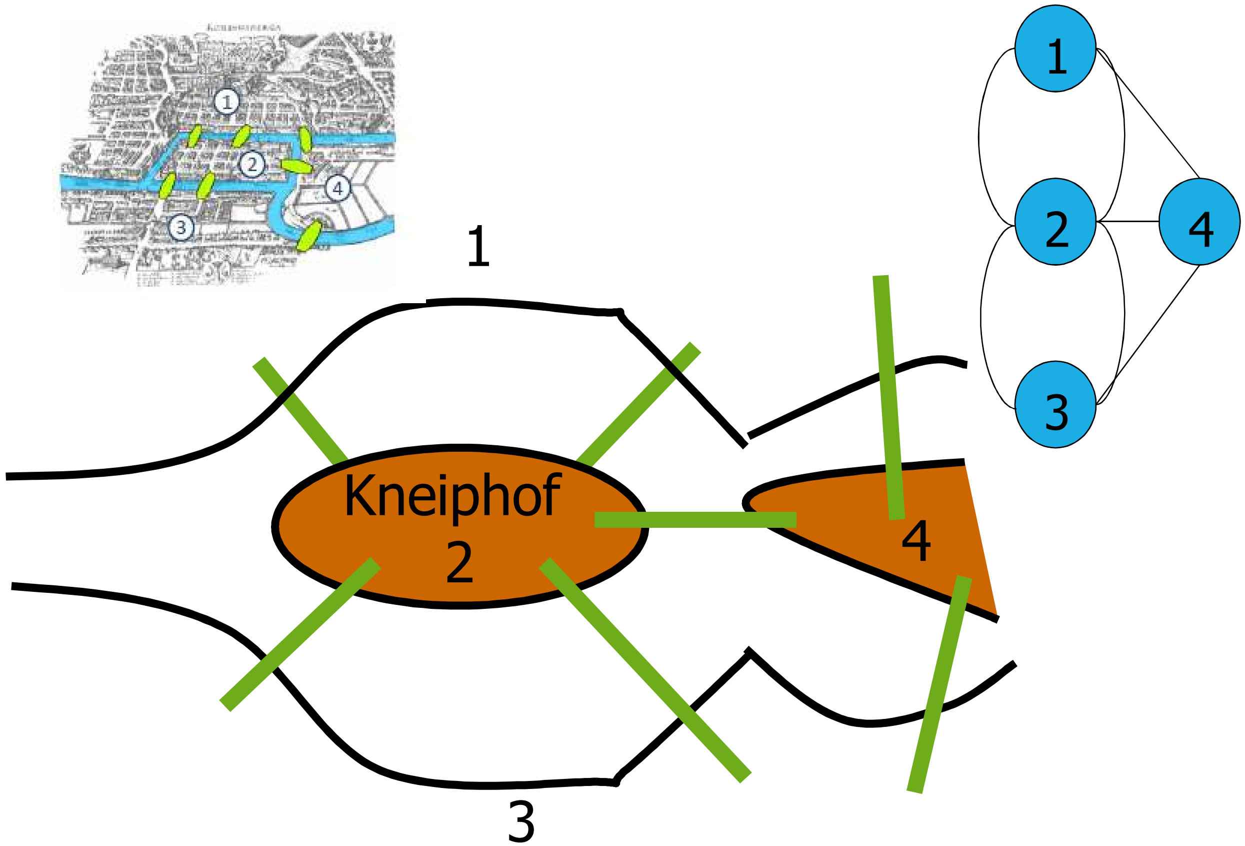
```
...
struct graph {int V; int E; int **madj; ST tab;};

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

```
Graph GRAPHinit(int V) {
  Graph G = malloc(sizeof *G);
  G->V = V;
  G->E = 0;
  G->madj = MATRIXint(V, V, 0);
  G->tab = STinit(V);
return G;
}

void GRAPHfree(Graph G) {
  int i;
  for (i=0; i<G->V; i++)
    free(G->madj[i]);
  free(G->madj);
  STfree(G->tab);
  free(G);
}
```

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.





I ponti di Königsberg
(Eulero, 1736)

grafi non orientati

grafi pesati

grafi non orientati pesati

```
static void  insertE(Graph G, Edge e) {
  int v = e.v, w = e.w, wt =e.wt;
  if (G->madj[v][w] == 0)
    G->E++;
  G->madj[v][w] = 1; G->madj[v][w] = wt;
  G->madj[w][v] = 1; G->madj[w][v] = wt;
}
int GRAPHgetIndex(Graph G, char *label) {
  int id;
  id = STsearch(G->tab, label);
  if (id == -1) {
    id = STcount(G->tab);
    STinsert(G->tab, label, id);
  }
  return id;
}
```

Attenzione:

- si possono generare cappi!
- non si possono generare multigrafi!

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

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->madj[v][w] !=0)
        a[E++] = EDGEcreate(v, w, G->madj[v][w]);
  return;
}
```

grafi non orientati

grafi orientati

grafi pesati

Vantaggi/svantaggi

- Complessità spaziale

  $S(n) = \Theta(|V|^2) \Rightarrow$ vantaggiosa SOLO per grafi densi
- No costi aggiuntivi per i pesi di un grafo pesato
- Accesso efficiente ($O(1)$) alla topologia del grafo (adiacenza di 2 vertici).

Rappresentazione

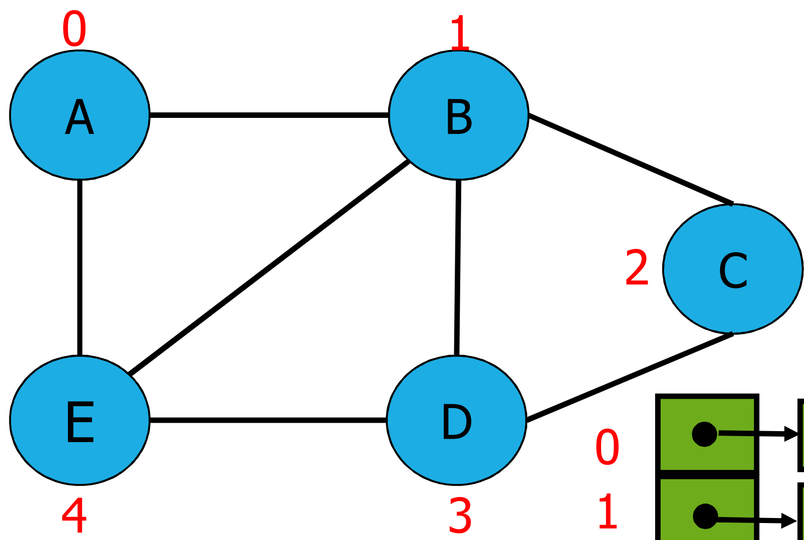
Conoscendo il numero di vertici, si può implementare con un vettore di liste, altrimenti lista di liste

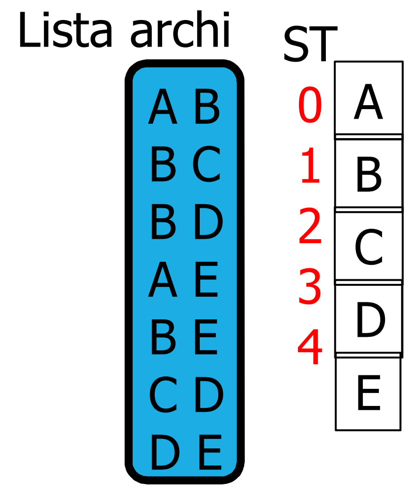
Lista di adiacenza

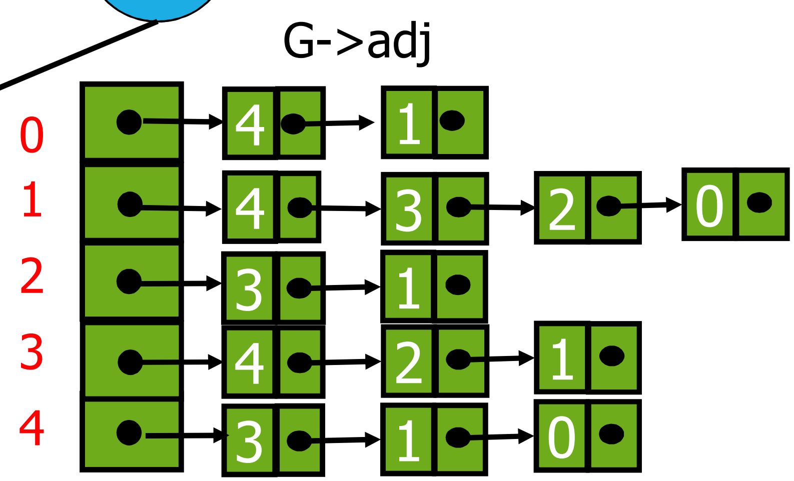
Dato G = (V, E), la lista di adiacenza è:

- 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.

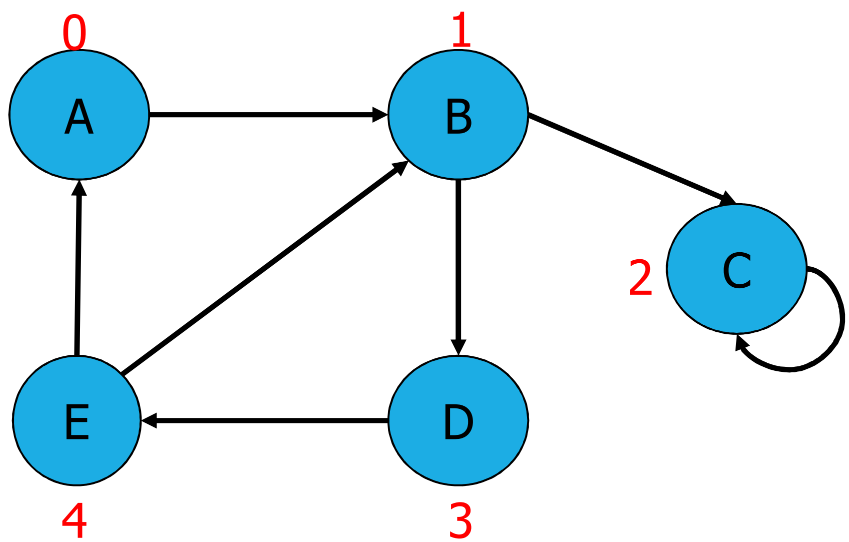
Non orientato non pesato







Orientato non pesato



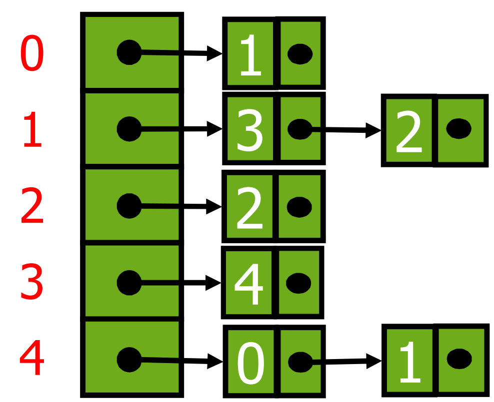
Lista archi

A B
B C
B D
C C
D E
E B
E A

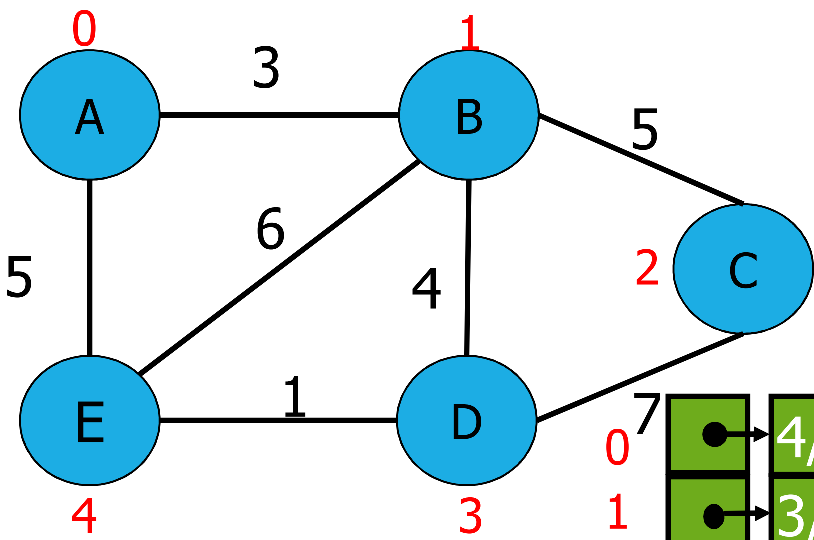
ST

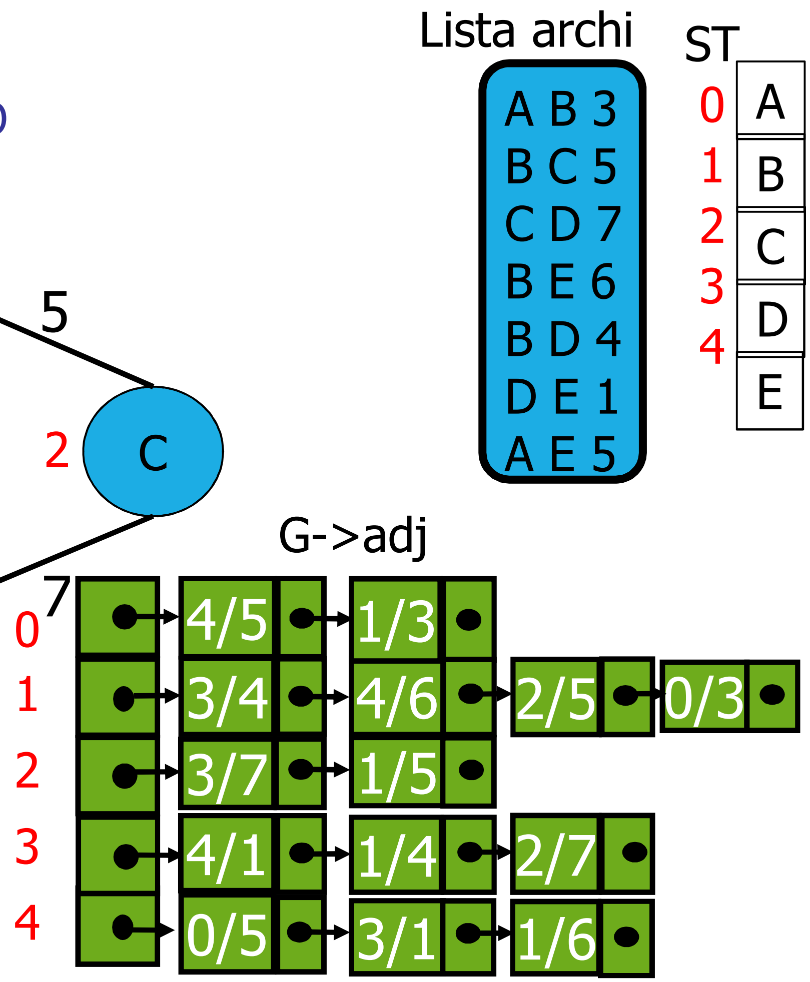
0	A
1	B
2	C
3	D
4	E

G->adj

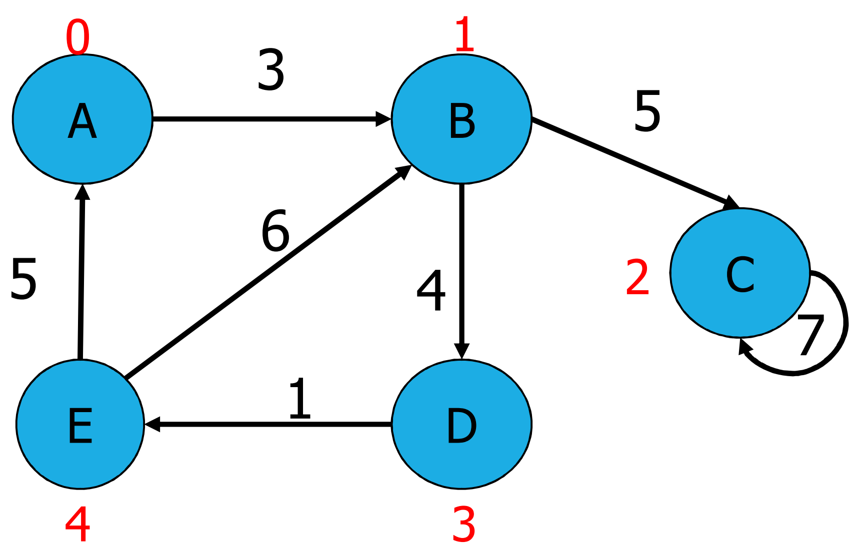


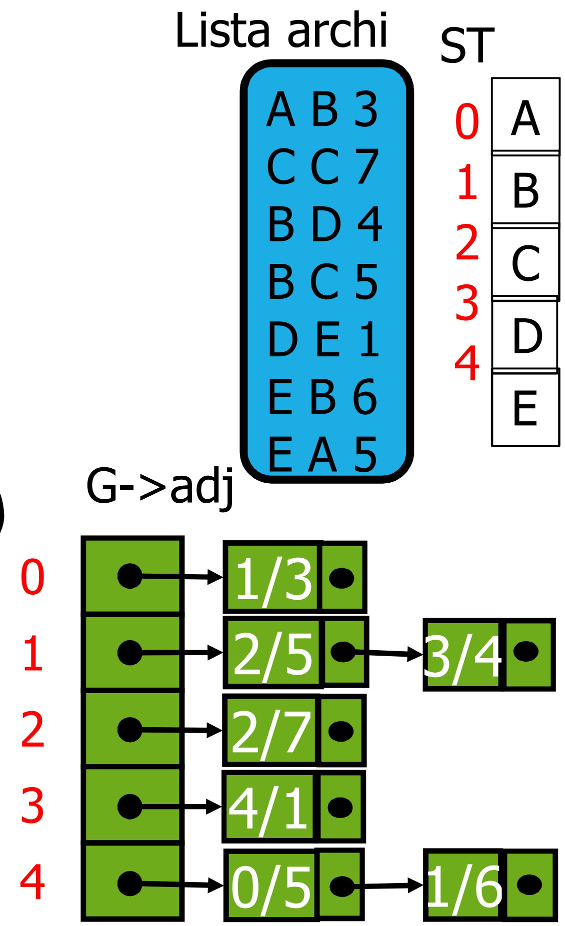
Non orientato pesato





Orientato pesato





Graph.c

```c
typedef struct node *link;

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

struct graph{int V; int E; link *ladj; ST tab; link z;} ;

static link NEW(int v, int wt, link next) {
  link x = malloc(sizeof *x);
  x->v = v; x->wt = wt; x->next = next;
  return x;
}
static Edge EDGEcreate(int v, int w, int wt) {
  Edge e;
  e.v = v; e.w = w; e.wt = wt;
  return e;
}
```

numero di vertici

numero di archi

lista di adiacenza

tabella di simboli

sentinella

grafi pesati

```c
Graph GRAPHinit(int V) {
  int v;
  Graph G = malloc(sizeof *G);
  G->V = V;
  G->E = 0;
  G->z = NEW(-1, -1,  NULL);
  G->ladj = malloc(G->V*sizeof(link));
  for (v = 0; v < G->V; v++)
    G->ladj[v] = G->z;
  G->tab = STinit(V);
return G;
}
```

```c
void GRAPHfree(Graph G) {
  int v;
  link t, next;
  for (v=0; v < G->V; v++)
    for (t=G->ladj[v]; t != G->z; t = next) {
      next = t->next;
      free(t);
    }
  STfree(G->tab);
  free(G->ladj); free(G->z); free(G);
}
void  GRAPHedges(Graph G, Edge *a) {
  int v, E = 0;
  link t;
  for (v=0; v < G->V; v++)
    for (t=G->ladj[v]; t != G->z; t = t->next)
      if (v < t->v)  a[E++] = EDGEcreate(v, t->v, t->wt);
}
```

grafi pesati

grafi non orientati

Attenzione: si possono generare cappi!

grafi pesati

```
static void  insertE(Graph G, Edge e) {
  int v = e.v, w = e.w, wt = e.wt;
  G->ladj[v] = NEW(w, wt, G->ladj[v]);
  G->ladj[w] = NEW(v, wt, G->ladj[w]);
  G->E++;
}
```

grafi non orientati (pesati)

```c
static void  removeE(Graph G, Edge e) {
  int v = e.v, w = e.w;  link x, p;
  for (x = G->ladj[v], p = NULL; x != G->z; p = x, x = x->next) {
    if (x->v == w) {
      if (x == G->ladj[v]) G->ladj[v] = x->next;
      else p->next = x->next;
      break;
    }
  }
  for (x = G->ladj[w], p = NULL; x != G->z; p = x, x = x->next) {
    if (x->v == v) {
      if (x == G->ladj[w]) G->ladj[w] = x->next;
      else p->next = x->next;
      break;
    }
  }
  G->E--;   free(x);
}
```

grafi non orientati

Vantaggi/svantaggi

Vantaggi:

- 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'adiacenza di 2 vertici v e w mediante scansione della lista di adiacenza di v
- uso di memoria per i pesi dei grafi pesati.

Generazione dei grafi

In generale i grafi sono modelli di situazioni reali e vengono formiti come dati di ingresso. In caso contrario, si possono generare dei grafi senza alcuna relazione ad un problema specifico.

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)

grafi non orientati

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)
    GRAPHinsertE(G, 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)
        GRAPHinsertE(G, 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? Non è richiesto trovarli tutti.

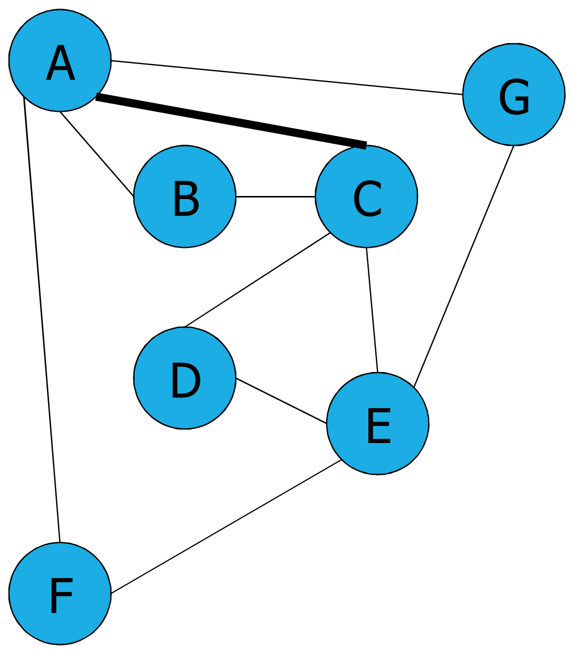
Se il grafo non orientato è connesso $\Rightarrow$ il cammino esiste per definizione, basta trovarne uno qualsiasi senza altri vincoli se non essere semplice. Non serve backtrack.

Se il grafo non orientato non è connesso $\Rightarrow$ il cammino esiste per definizione se i vertici sono nella stessa componente connessa, altrimenti non esiste. Non serve backtrack.

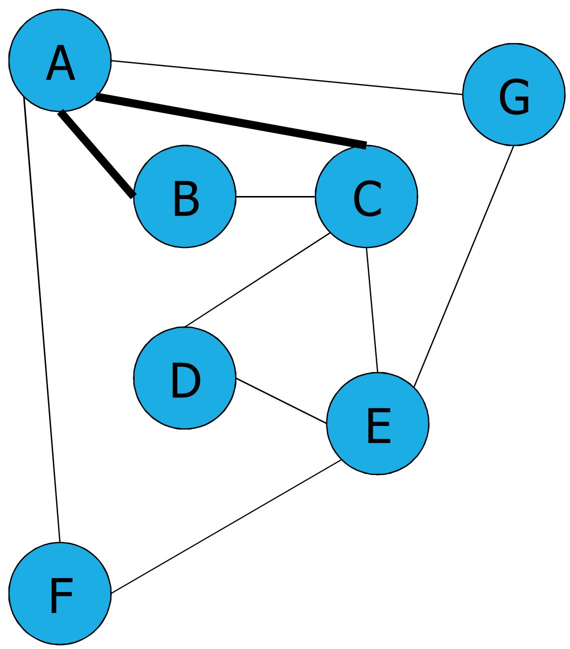
$\exists$ p: $v \to_p w$

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

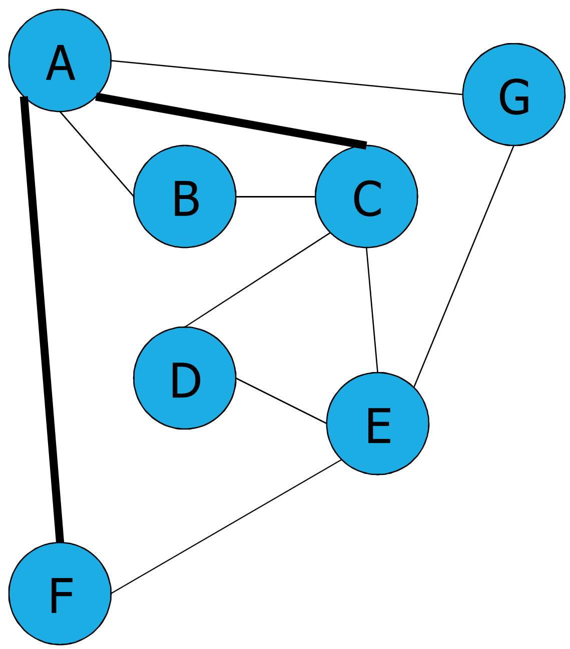
Esempio



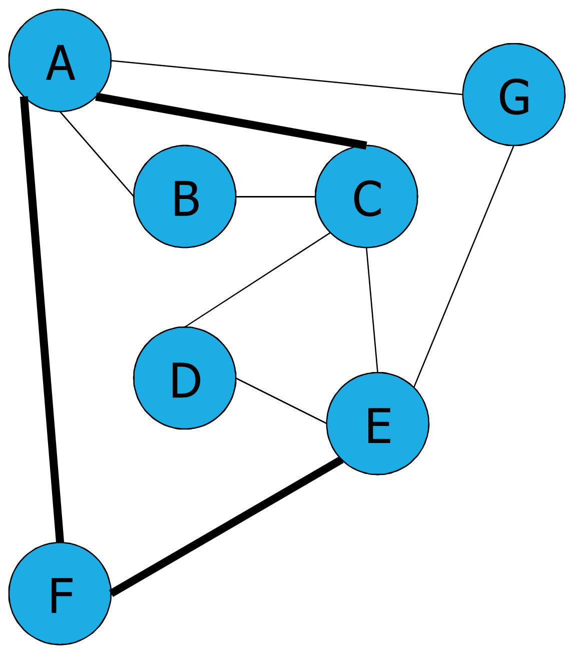
$\exists p$: $C \rightarrow_p G$?
passo 1:
vertici adiacenti a C
non ancora visitati:
A, B, D, E
seleziono A



$\exists p$: $A \rightarrow_p G$?
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



$\exists p\text{:}\ \ F \rightarrow_p G?$

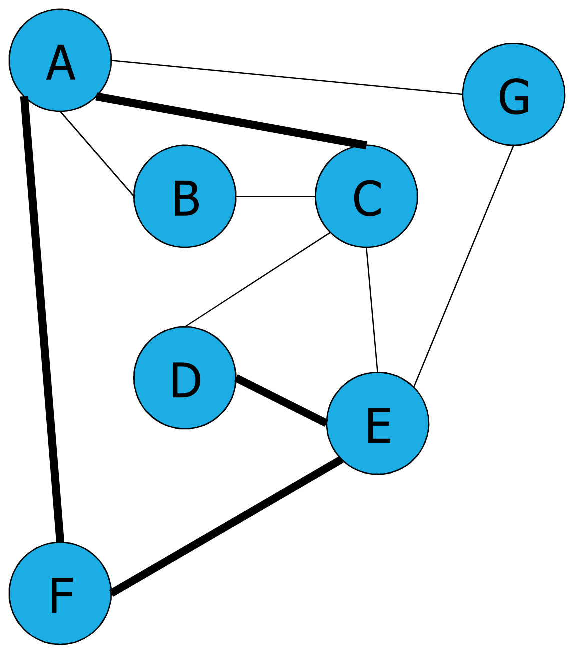
passo 4:

vertici adiacenti a F

non ancora visitati:

E

seleziono E



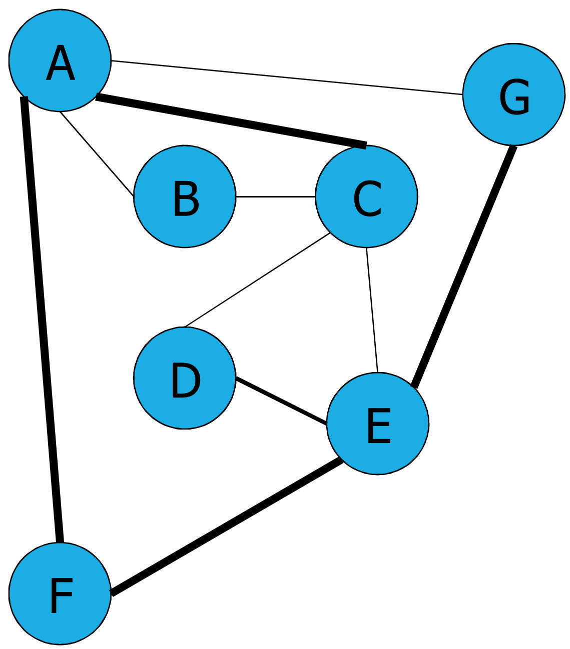
$\exists p\text{:}\ E \rightarrow_p G?$

passo 5:

vertici adiacenti a E non ancora visitati:

D, G

seleziono D



$\exists p$: D $\rightarrow_p$ G?

passo 6:

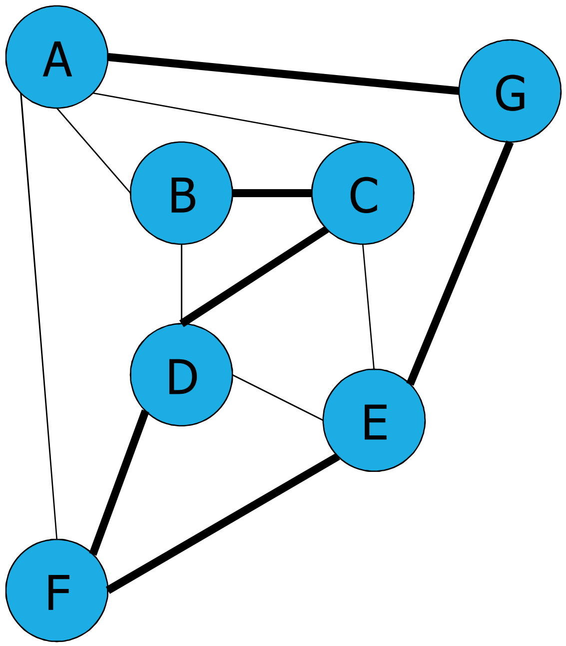
vertici adiacenti a D non ancora visitati: nessuno

La ricorsione torna a E e seleziona **G**

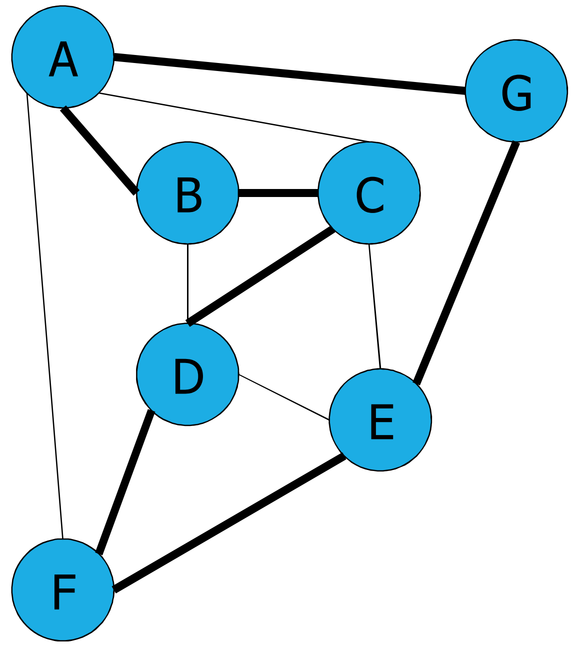
Il Cammino di Hamilton

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!

```
void GRAPHpath/GRAPHpathH(Graph G, int id1, int id2) {
  int t, found, *visited;

  visited = malloc(G->V*sizeof(int));
  for (t=0; t<G->V; t++)
    visited[t]=0;

  if (id1 < 0 || id2 < 0)
    return;
  found = pathR/pathRH(G, id1, id2, G->V-1, visited);
  if (found == 0)
    printf("\n Path not found!\n");
}
```

Attenzione: per sinteticità si viola la sintassi del C

```
static 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->madj[v][t] == 1)
      if (visited[t] == 0)
        if (pathR(G, t, w, visited)) {
          printf("(%s, %s) in path\n",
                  STsearchByIndex(G->tab, v),
                  STsearchByIndex(G->tab, t));
          return 1;
        }
  return 0;
}
```

matrice delle adiacenze

stampa gli archi del cammino in ordine inverso

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

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

matrice delle adiacenze

stampa gli archi del cammino in ordine inverso

backtrack

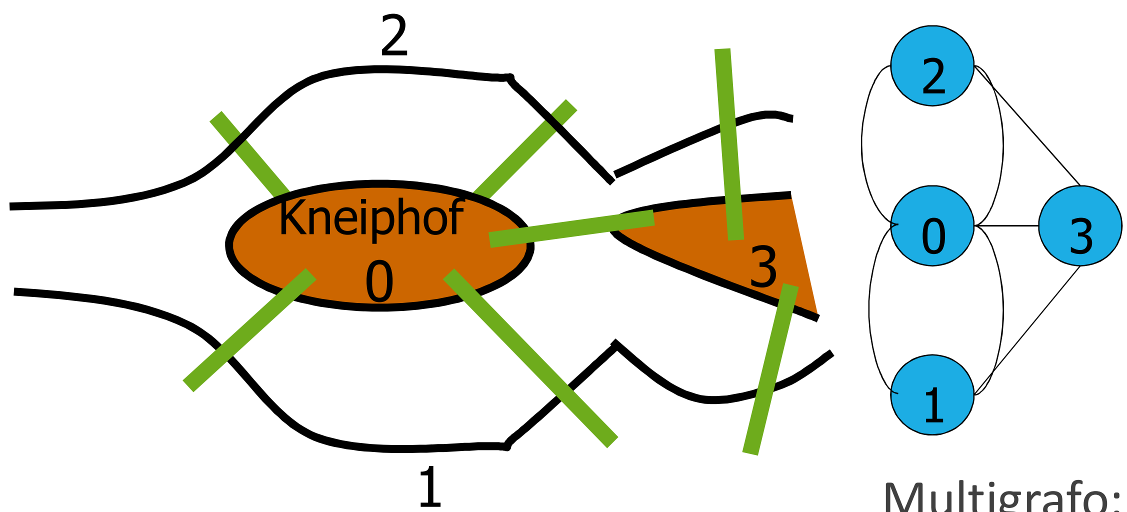
Il Cammino di Eulero

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**.

Lemmi

- 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

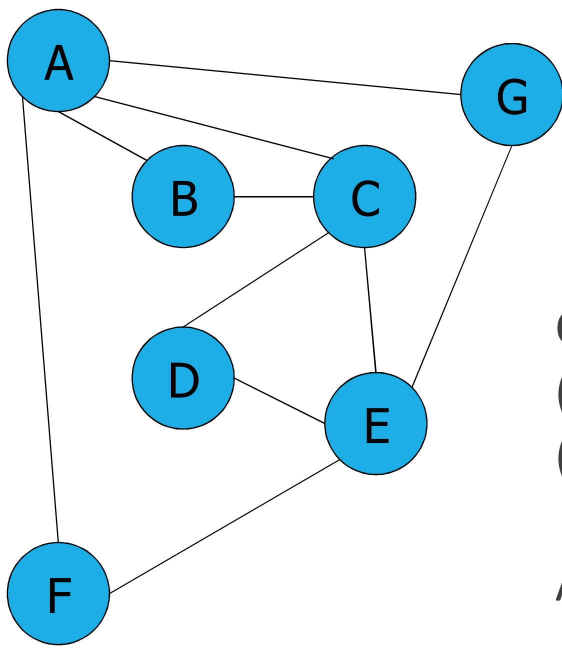


$\nexists$ né cammini, né cicli di Eulero

Multigrafo: archi multipli che connettono la stessa coppia di vertici

Esempio: verifica dell'esistenza del 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, A)

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

Esercizi di teoria

- 10. Visite dei grafi e applicazioni
  - 10.1 Rappresentazioni