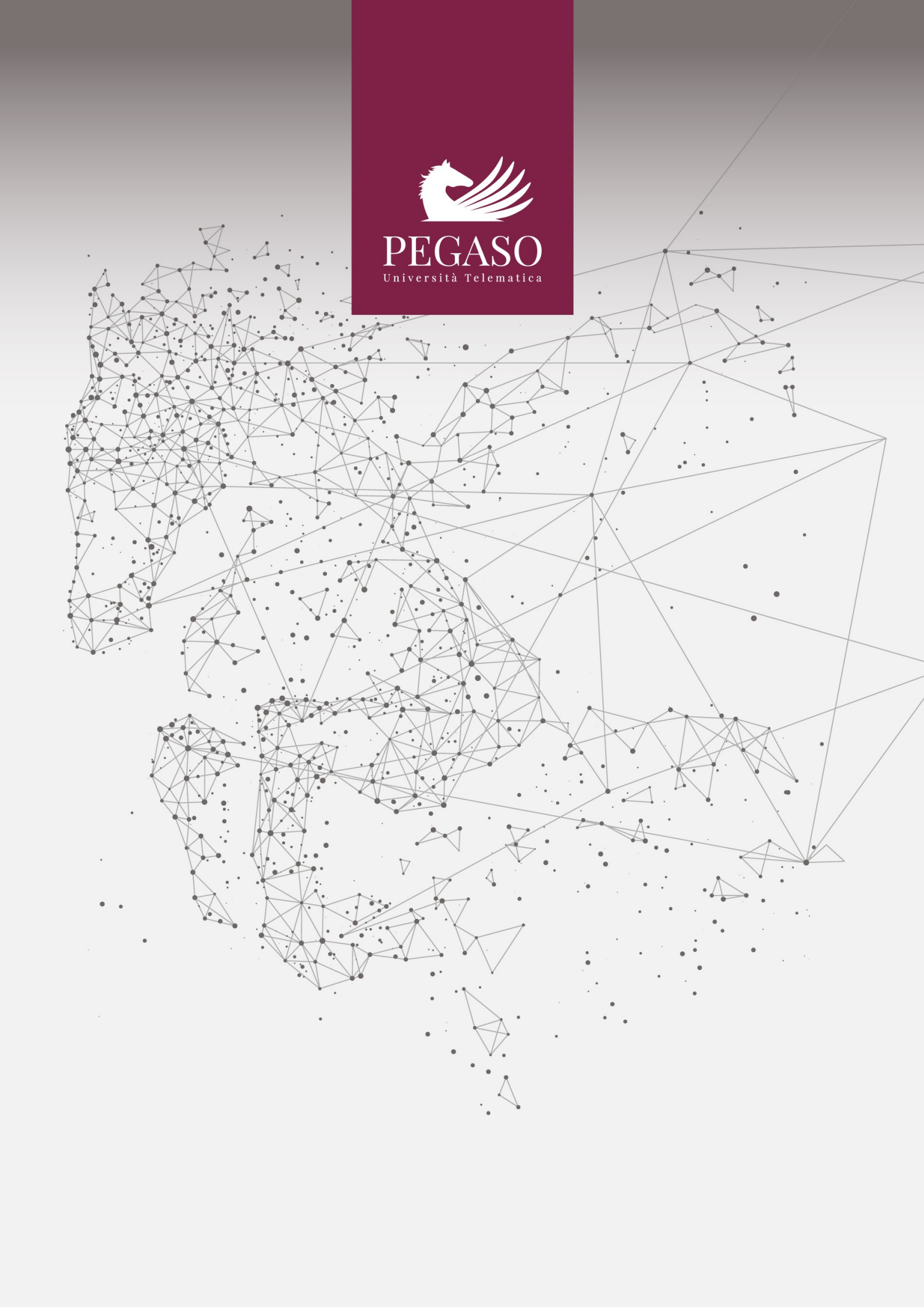

PEGASO
Università Telematica

Indice

1. Grafo

Un grafo è una struttura relazionale formata da un numero finito V di **vertici** (**nodi**) e un numero finito E di **segmenti** (**archi**) che collegano ogni nodo agli altri.

Di seguito un esempio di grafo:

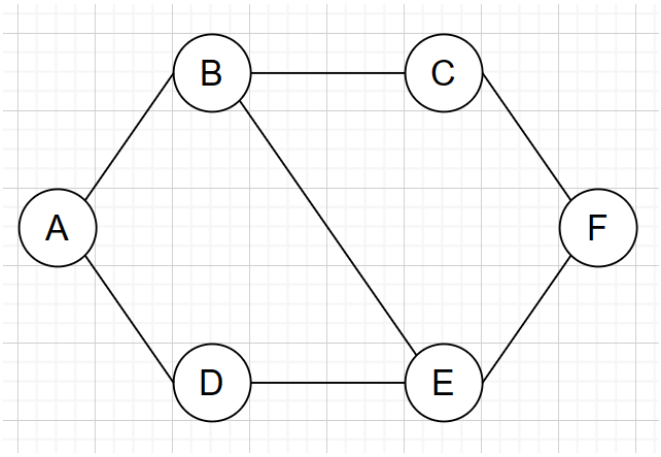


Figura 1: Esempio di grafo

Tale grafo può essere dunque identificato come: $G = (V, E)$

- $V(G) = \ vertici\ del\ grafo\ = \ \{A, B, C, D, E, F\}$
- Ordine del grafo = numero dei nodi

Relativamente agli archi, un arco è una relazione binaria tra 2 nodi:

- $\alpha = (a,\ b)$ con $\alpha \in E$, $a \in V$ e $b \in V$; a e b sono nodi del grafo e sono detti "estremi" dell'arco α
- $E(G) = \ archi\ del\ grafo\ = \ \{(A,B),\ (B,C),\ (C,F),\ (F,E),\ (E,D),\ (D,A),\ (B,E)\}$

Tale "relazione binaria" può essere definita come coppia **ordinata** o **non ordinata** a seconda se sia indicato un verso. Nel caso di coppie non ordinate, le generiche coppie (a, b) e (b, a) indicano lo stesso arco del grafo. Viceversa, nel caso delle coppie ordinate le coppie (a, b) e (b, a) indicano un verso differente sullo stesso arco, rispettivamente a → b e b → a.

Possiamo inoltre parlare di "orientamento" del grafo, in particolare:

- **Grafo orientato (grafo diretto o digrafo)**: è un grafo i cui archi hanno una direzione ed un verso; in questo caso gli archi sono frecce
- **Grafo non orientato**: è un grafo i cui archi hanno con una relazione biunivoca tra i nodi

Il **grado** di un nodo è il numero di archi incidenti su di esso.

Un grafo con un arco fra tutte le coppie di nodi è detto **completo**.

In un grafo completo con m nodi:

$$m = numeri\ di\ archi = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Un grafo è detto **connesso** se, per ogni coppia di vertici $(u, v) \in V$ esiste un cammino che collega u a v.

Cammino: il cammino da un nodo di origine X ad un nodo di destinazione Y è una sequenza di nodi adiacenti per andare da X ad Y, senza nodi ripetuti

Percorso: il percorso da un nodo di origine X ad un nodo di destinazione Y è una sequenza di nodi adiacenti per andare da X ad Y con alcuni nodi ripetuti

Quando si parla di cammino ci si riferisce sia al caso di grafo orientato che non orientato.

A volte ci si riferisce al cammino senza nodi ripetuti come **cammino semplice** mentre al percorso come **cammino** generico.

Lunghezza (**cardinalità**): numero degli archi nella sequenza (sia nel caso di cammino che di percorso)

Distanza: numero di nodi della sequenza

Il **cammino ottimale** ed il **percorso ottimale** sono rispettivamente il cammino ed il percorso con il costo più basso.

Solitamente ci si riferisce al termine cammino quando si considera un grafo orientato: in un **grafo orientato** $G = (V, E)$, un **cammino di lunghezza** $\boldsymbol{k}$ è una sequenza di vertici $u_0, u_1, \dots, u_k$, tale che $(u_i, u_{i+1}) \epsilon E\ per\ 0 \leq i \leq k-1$.

In un grafo non orientato si parla invece di catena: in un **grafo non orientato** $G = (V, E)$, una **catena di lunghezza** $\boldsymbol{k}$ è una sequenza di vertici $u_0, u_1, \dots, u_k$, tale che $[u_i, u_{i+1}] \epsilon E\ per\ 0 \leq i \leq k-1$.

Distinguiamo inoltre tra **ciclo** e **circuito**:

- in un **grafo orientato** $G = (V, E)$, un **ciclo di lunghezza** $\boldsymbol{k}$ è una sequenza di vertici $u_0, u_1, \dots, u_k$, tale che $(u_i, u_{i+1}) \epsilon E\ per\ 0 \leq i \leq k-1$, $u_0 = u_k\ e\ k > 2$
- in un **grafo non orientato** $G = (V, E)$, una **catena di lunghezza** $\boldsymbol{k}$ è una sequenza di vertici $u_0, u_1, \dots, u_k$, tale che $[u_i, u_{i+1}] \epsilon E\ per\ 0 \leq i \leq k-1$, $u_0 = u_k\ e\ k > 2$

Un ciclo è detto **semplice** se tutti i suoi nodi sono distinti (ad esclusione del primo e dell'ultimo.

Un grafo senza cicli è detto **aciclico**; un grafo orientato aciclico è chiamato **DAG** (**Directed Acyclic Graph**).

In teoria dei grafi, definiamo inoltre un grafo come **pesato** se ogni arco ha associato un peso o un costo numerico. Questo valore numerico rappresenta una qualche misura di distanza, costo, tempo o altra grandezza associata all'arco del grafo.

In termini di "dimensioni" di un grafo abbiamo:

- Numero di nodi: $n = |V|$
- Numero di archi: $m = |E|$

La complessità nei grafi è espressa in termini sia di n che di m (ad es. $O(n + m)$)

Alcune relazioni fra n e m:

- In un grafo non orientato: $m \leq \frac{n(n-1)}{2} = O(n^2)$
- In grafo orientato: $m \leq n^2 - n = O(n^2)$

Informalmente (anche se non c'è accordo sulla definizione), un grafo si dice:

- **sparso** se ha "pochi archi"; i grafi con $m = O(n)$ o $m = O(n \log_2 n)$ sono considerati sparsi
- **denso** se ha "tanti archi"; i grafi con $m = \Omega(n^2)$ sono considerati densi

2. Lista e matrice di adiacenze

Per sviluppare un grafo in un programma informatico si utilizzano due metodi.

- **Lista di adiacenze**: i nodi del grafo sono memorizzati in una variabile array; ogni elemento dell'array indica un nodo del grafo e contiene la lista delle sue connessioni (archi)
- **Matrice di adiacenze**: i nodi e gli archi sono memorizzati in una matrice quadrata $(n\ x\ n)$ dove le n righe indicano i nodi di origine e le n colonne quelli di destinazione; ogni elemento della matrice indica se i due nodi sono collegati tra loro (1) oppure no (0). Da notare che nel caso di grafo non orientato, la matrice risultante è una **matrice triangolare superiore**

Consideriamo il seguente grafo:

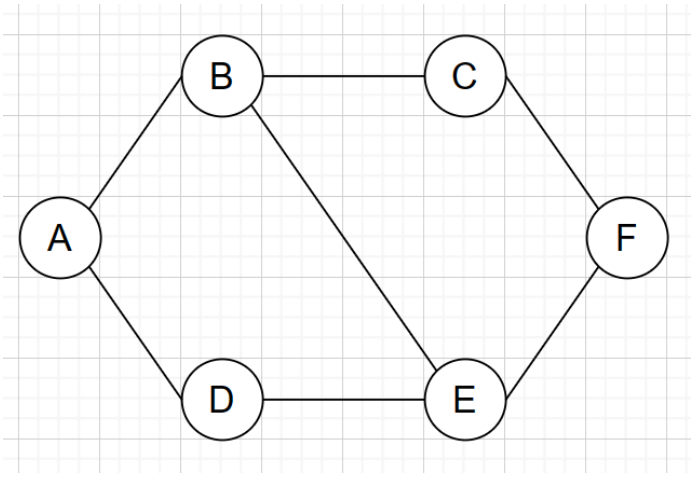


Figura 2: Esempio di grafo

La rappresentazione mediante lista di adiacenze sarebbe:

Nodo	Lista di adiacenze
A	B, D
B	A, C, E
C	B, F
F	C, E
E	B, D, F
D	A, E

La rappresentazione mediante matrice di adiacenze sarebbe invece:

	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	0	1	0
C	0	1	0	0	0	1
D	1	0	0	0	1	0
E	0	1	0	1	0	1
F	0	0	1	0	1	0

Consideriamo il seguente grafo orientato:

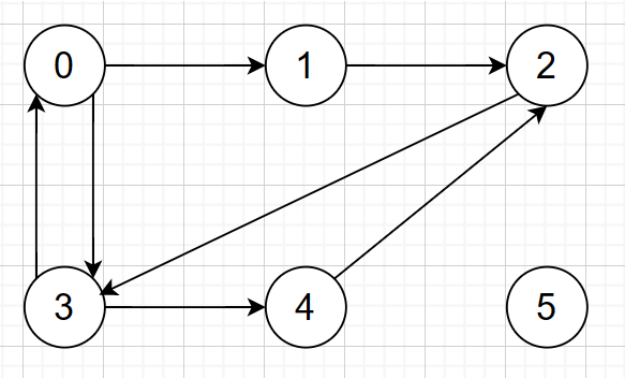


Figura 3: Grafo orientato

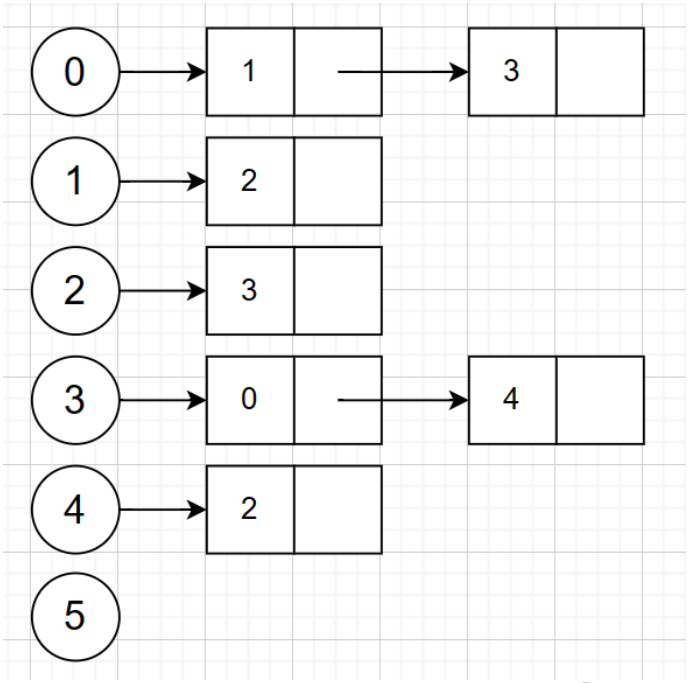
Tale grafo:

- È un grafo orientato
- Possiede 6 nodi
- Possiede 7 archi
- Non è completo
- Non è connesso

La rappresentazione mediante matrice di adiacenze è la seguente:

	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

La rappresentazione mediante lista di adiacenze è:



Nel caso di matrice delle adiacenze, lo spazio "occupato" è $O(n^2)$ (essendo n il numero di nodi); nel caso di lista di adiacenze invece, lo spazio occupato è $O(n+m)$, essendo (essendo n il numero di nodi ed m il numero di archi).

La complessità $O(n+m)$ deriva dal fatto che per costruire una lista di adiacenza, per ogni nodo dobbiamo attraversare tutti gli archi che partono da tale nodo e aggiungere alla lista di adiacenza i nodi di arrivo di ciascuno di questi archi.

In totale, quindi, per costruire tutte le liste di adiacenza, dobbiamo attraversare tutti gli archi del grafo una sola volta, ovvero la complessità dell'operazione è $O(m)$.

Inoltre, dobbiamo creare una lista di adiacenza per ogni nodo del grafo, ovvero dobbiamo attraversare tutti i nodi del grafo e per ciascuno di essi creare una lista di adiacenza. Pertanto, la complessità dell'operazione di costruzione della lista di adiacenza è proporzionale al numero di nodi del grafo, ovvero la complessità è $O(n)$.

Quindi, sommando le due complessità, otteniamo una complessità totale di $O(n + m)$ per la costruzione di una lista di adiacenza. Questa è una complessità molto efficiente per la rappresentazione di grafi di grandi dimensioni, in quanto è proporzionale al numero totale di archi e nodi del grafo.

Nel nostro esempio, l'occupazione di spazio con matrice di adiacenze è pari a 36; nel caso di lista di adiacenze è pari a 13.

Qualora il grafo considerato non fosse orientato:

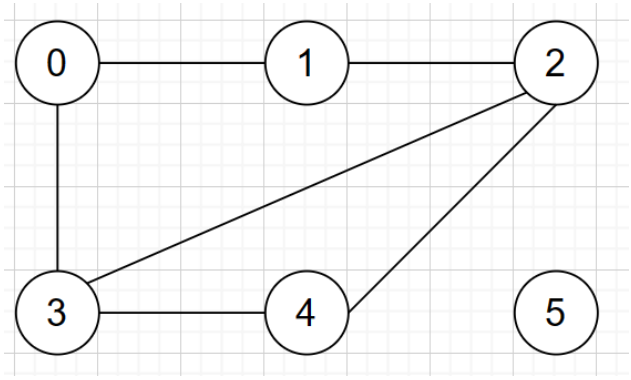


Figura 4: Grafo non orientato

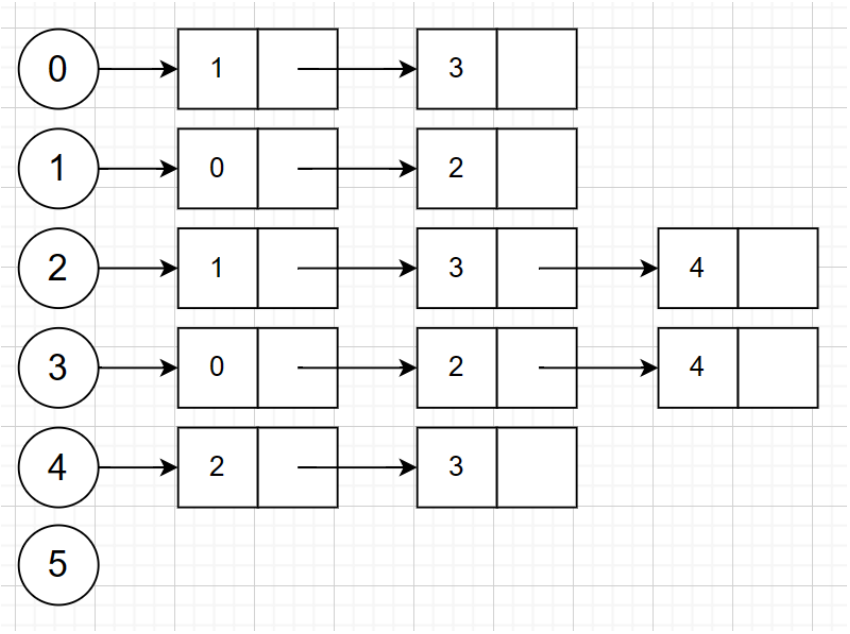
La rappresentazione mediante matrice di adiacenze è la seguente:

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

Si ha cioè una matrice triangolare superiore e l'occupazione di spazio è pari a:

$n(n-1)/2$

La rappresentazione mediante matrice di adiacenze è invece:



È anche possibile associare dei "pesi" agli archi: in tal caso si parlerà di grafo "pesato". Tale peso può essere rappresentato mediante una funzione di costo: $p: V \times V \rightarrow R$, dove R è l'insieme dei numeri reali; quando tra due vertici non esiste un arco, il peso è infinito.

Consideriamo ad es. il seguente grafo pesato:

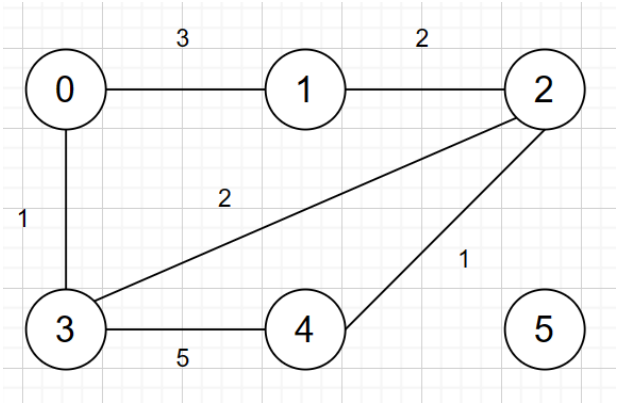


Figura 5: Grafo pesato

La rappresentazione mediante matrice di adiacenze è la seguente:

	0	1	2	3	4	5
0	∞	3	∞	1	∞	∞
1		∞	2	∞	∞	∞
2			∞	2	1	∞
3				∞	5	∞
4					∞	∞
5						∞

Possiamo quindi riassumere dicendo che:

- **Matrici di adiacenza**
  - Spazio richiesto $O(n^2)$
  - Verificare se u è adiacente a v richiede tempo $O(1)$
  - Iterare su tutti gli archi richiede tempo $O(n^2)$
  - Ideale per grafi densi
- **Liste di adiacenza**
  - Spazio richiesto $O(n+m)$
  - Verificare se u è adiacente a v richiede tempo $O(n)$
  - Iterare su tutti gli archi richiede tempo $O(n+m)$
  - Ideale per grafi sparsi

3. Implementazione

Di seguito il codice in Python per implementare le varie tipologie di Grafo

Grafo Non Orientato

```
from collections import defaultdict

class GraphNotOriented:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    def print(self):
      for vertex in g.graph:
        print(vertex, ":", g.graph[vertex])
```

Grafo Orientato

```
class GraphOriented:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)

    def print(self):
      for vertex in g.graph:
        print(vertex, ":", g.graph[vertex])
```

Grafo Pesato Orientato

```
class GraphWeightedOriented:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)

    def add_edge(self, u, v, weight):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append((v, weight))

    def print(self):
      for u in self.graph:
          print(u, end=": ")
          for v, weight in self.graph[u]:
              print(f"({v}, {weight})", end=" ")
          print()
```

Grafo Pesato Non Orientato

```
class GraphWeightedNotOriented:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)

    def add_edge(self, u, v, weight):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append((v, weight))
        self.graph[v].append((u, weight))

    def print(self):
      for u in self.graph:
          print(u, end=": ")
          for v, weight in self.graph[u]:
              print(f"({v}, {weight})", end=" ")
          print()
```

Di seguito il codice per creare un grafo:

```
g = GraphWeightedOriented(4)
g.add_edge(0, 1, 7)
g.add_edge(0, 2, 10)
g.add_edge(1, 3, 3)
g.add_edge(2, 3, 2)

lista_di_adiacenze= {
    0: [(1, 7), (2, 10)],
    1: [(3, 3)],
    2: [(3, 2)],
    3: []
}

g.print()
```

Come si può vedere dal precedente esempio, è possibile creare il grafo partendo da una lista di adiacenze oppure aggiungendo di volta in volta i vertici / archi.

Consideriamo di seguito l'algoritmo per determinare se ad esempio un **grafo non orientato** ha un ciclo:

```
boolean hasCycleRec(Graph G, Node u, Node p, boolean[] visited)
     visited[u] = true
     foreach v in G.adj(u) – {p}
          if visited[v]
               return true
          else if hasCycleRec(G, v, u, visited)
               return true
     return false

boolean hasCycle(Graph G)
     boolean[] visited = new boolean[G.V()]
     foreach u in G.V()
          visited[u] = false
     foreach u in G.V()
          if !visited[u]
               if hasCyclerec(G, u, null, visited)
                    return true
     return false
```

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione;
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione;
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley;
- Sedgewick: Algoritmi in C, Pearson, 2015;
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.