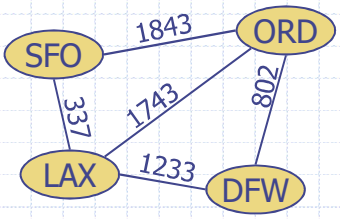
Graphs



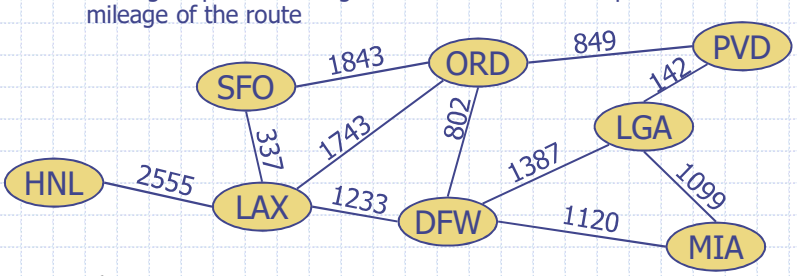
1

1

Come funziona? Fare esempi

Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices — VERTICI
  - E is a collection of pairs of vertices, called edges — ARCHI
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



Aeroporti: Archi = distanza / costo voli: Se ho costo voli posso avere più archi che collegano stessi nodi

2

In ogni Vertice c'è un valore: il contenuto è qualsiasi cosa, un generico elemento E, ma senza chiave, non mi interessa. In generale non abbiamo punto di ingresso o di uscita.

NOTA: Arco pesato: costo per passare su quell'arco.
Arco non pesato: peso 0.

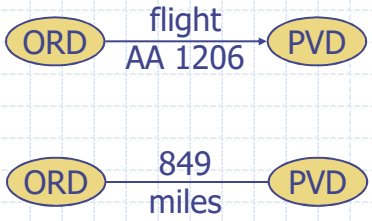
V è un set di nodi; significa che non possiamo avere duplicati
E è una collezione: collezione di coppia di vertici. Posso avere duplicatiaaa
e anche un nodo collegato a sé stesso.

NOTA: Percorso più breve non per forza unico

Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network

Arco come coppia di vertici Sorgente-destinazione

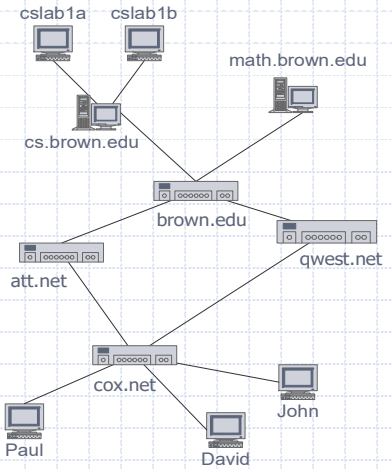


3

3

Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram

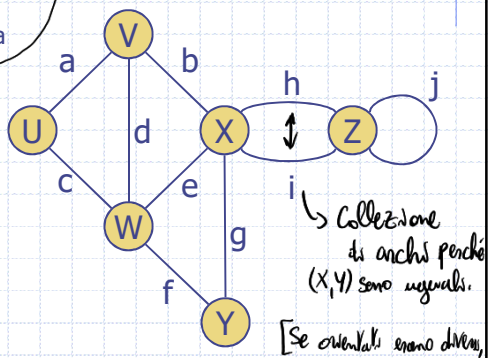


4

4

Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



Arco a: U e V sono i suoi punti terminali. → Estremi arco

Archi che vocano vertice X.

Collezione di archi perché (X,Y) sono ugualmente.

[Se orientati erano diversi, ma comunque //]

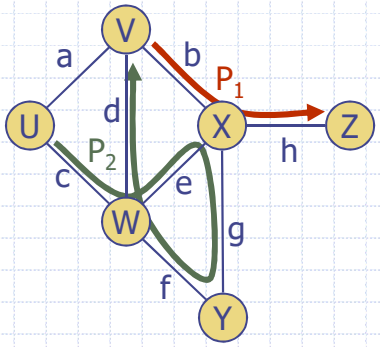
Se sono connessi da un solo arco.

5

5 Nota: Se grafo è orientato si parla di incidenza in ingresso o incidenza in uscita
Rango in ingresso / Rango in uscita

Terminology (cont.)

- Path Percorso
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



Tutti i suoi vertici e anche distinti tra loro

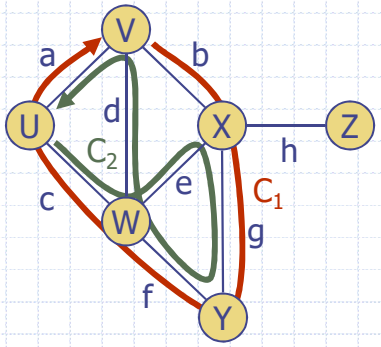
6

6 Nota: Un percorso non necessariamente è unico tra 2 vertici
2 nodi non necessariamente hanno un percorso.

Ciclo: un qualsiasi percorso che chiude da un vertice e ritorna nello stesso vertice

Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle (Non vi è alcuna ripetizione di vertici o archi)
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



7

7

m= archi n=nodi

Properties

Numero di nodi che entrano

Property 1

Somma dei gradi dei vertici è 2m

$\sum_v \deg(v) = 2m$ → archi

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

$m \leq n\,(n-1)/2$

Proof: each vertex has degree at most $(n-1)$

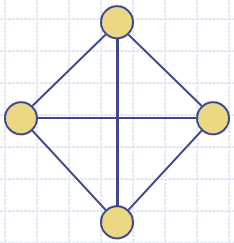
What is the bound for a directed graph?

→ Unico modo per limitare superiormente il numero di archi

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v

Example



- $n = 4$
- $m = 6$
- $\deg(v) = 3$

8

8

Complessità legata al grado (limitato da num. archi)
numero di archi (limitato da num. di nodi)
numero di nodi

→ Grado complessivo = 3

Subset; non prendiamo tutto il set.

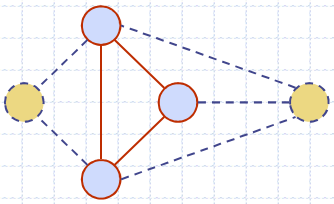
Subset perché facciamo riferimento a grafo semplice (NO self loop e anche doppi)

Subgraphs

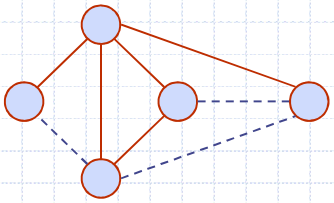
- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G

Sottografo ricoprente

Ha stessi vertici ma non stessi archi



Subgraph



Spanning subgraph

9

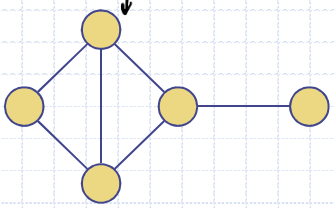
Piccolezza:
non bisogna specificare che subset di archi deve avere endpoints

9

Connectivity

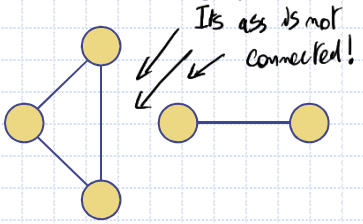
- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G

Grafo connesso



Connected graph

Its ass is not connected!



Non connected graph with two connected components

10

10 Questi 2 sono tra loro grafi connessi: Sottografi connessi; Componenti connesse.
Ma è esteso al massimo possibile, prendendo quanti più nodi possibili.

GRAFO CONNESSO: Intero grafo è componente connessa.

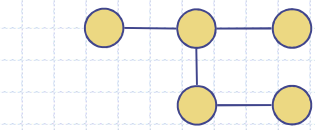
NOTA: Se grafo non orientato, se esiste percorso da A a B allora vale il viceversa

Trees and Forests

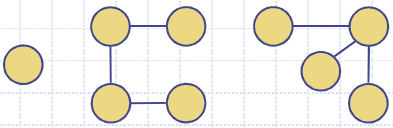
- A (free) tree is an undirected graph T such that → No direzione
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees

Non c'è radice: No concetto di punto di partenza.



Tree



Forest

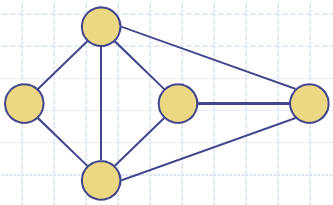
11

11

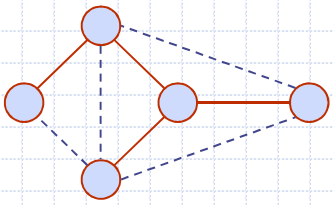
Spanning Trees and Forests

Ricoprente: Prende vertici ma no archi

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

12

12

Possiamo dare maggior peso all'arco che al grafo.

Vertices and Edges

- A **graph** is a collection of **vertices** and **edges**.
- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.
- A **Vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code)
  - We assume it supports a method, element(), to retrieve the stored element.
- An **Edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method.

13

13

iteration = ITERABILE che scorro

Graph ADT

numVertices( ):	Returns the number of vertices of the graph.
vertices( ):	Returns an iteration of all the vertices of the graph.
numEdges( ):	Returns the number of edges of the graph.
edges( ):	Returns an iteration of all the edges of the graph.
getEdge(u, v):	Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
endVertices(e):	Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
opposite(v, e):	For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
outDegree(v):	Returns the number of outgoing edges from vertex v.
inDegree(v):	Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
outgoingEdges(v):	Returns an iteration of all outgoing edges from vertex v.
incomingEdges(v):	Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
insertVertex(x):	Creates and returns a new Vertex storing element x.
insertEdge(u, v, x):	Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
removeVertex(v):	Removes vertex v and all its incident edges from the graph.
removeEdge(e):	Removes edge e from the graph.

Può dare 2 risultati diversi se è vero

Passo vertice

Sempre voglio grafo semplice oppure se vertice non esiste errore

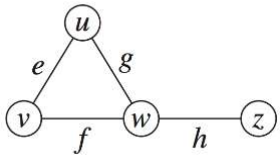
14

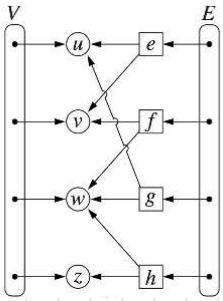
14

Aggiunta: No doppio, no self loop e no aggiunta con nodi che non esistono

Edge List Structure

- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects



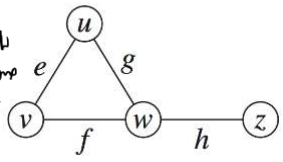


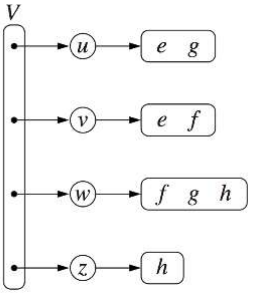
15

15

Adjacency List Structure

- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



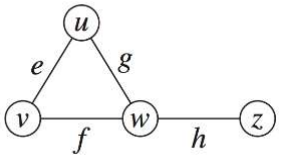


16

16

Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge



	0	1	2	3
u → 0		e	g	
v → 1	e		f	
w → 2	g	f		h
z → 3			h	

17

Come accedo ai nodi se sono numeri?

17 Traduco nodi in numeri (che stanno ordinati): mxm con m numero di vertici
Riga e colonna corrispondo ai vertici e ogni incrocio collegato da elemento arco.
Eliminare nodo ⇒ tolgo riga e colonna per quel nodo. Tempo costante.
Problema, spazio occupato è duplicato: 2 nodi e 2 archi. (Se grafo non è semplice, disegnarlo pure, non necessariamente vuota)
Se grafo orientato aggiungo + un numero direzione. (Se archi pesati ho il peso)

Performance

Lista di adiacenza e lista di archi

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$?	n^2
incidentEdges(v)	m	$\deg(v)$ [1]	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	* n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

elenco → mi devo scorrere tutti gli archi : m

Scorrere tutta la riga della matrice

Vado su riga i colonna

1: Scorro tutti archi per capire se sono entrambi

Minimo tra gradi: li scorro contemporaneamente per vedere se è lo stesso arco

18

18

1: grado rappresenta numero degli archi incidenti e quindi faccio quell'elenco.

* devo aggiungere riga e colonna

Devo solo creare nel rappresentato; Rimozione è doppia di removeIncidentEdges(v)

areAdjacent è importante, perché si usa Matrice.
Si usa quando attraversare l'insieme di vertici è raro.

Interface

```java
/**
 * Interfaccia per un vertice di un grafo
 * @param <V> Tipo dell'elemento contenuto nel vertice
 */
public interface Vertex<V> {
    /** Returns the element associated with the vertex. */
    V getElement();
}

/**
 * Interfaccia relativa al vertice
 * @param <E> tipo di dato dell'elemento contenuto nell'arco
 */
public interface Edge<E> {
    /** Returns the element associated with the edge. */
    E getElement();
}
```

19

19

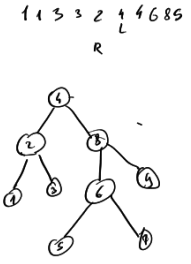
Graph interface

```java
/**
 * An interface for a graph structure.
 */
public interface Graph<V,E> {

    /** Returns the number of vertices of the graph */
    int numVertices();

    /** Returns the number of edges of the graph */
    int numEdges();

    /** Returns the vertices of the graph as an iterable collection */
    Iterable<Vertex<V>> vertices();

    /** Returns the edges of the graph as an iterable collection */
    Iterable<Edge<E>> edges();

    /**
     * Returns the number of edges leaving vertex v.
     * Note that for an undirected graph, this is the same result
     * returned by inDegree
     * @throws IllegalArgumentException if v is not a valid vertex
     */
    int outDegree(Vertex<V> v) throws IllegalArgumentException;

    /**
     * Returns the number of edges for which vertex v is the destination.
     * Note that for an undirected graph, this is the same result
     * returned by outDegree
     * @throws IllegalArgumentException if v is not a valid vertex
     */
    int inDegree(Vertex<V> v) throws IllegalArgumentException;

    /**
     * Returns an iterable collection of edges for which vertex v is the origin.
     * Note that for an undirected graph, this is the same result
     * returned by incomingEdges.
     * @throws IllegalArgumentException if v is not a valid vertex
     */
    Iterable<Edge<E>> outgoingEdges(Vertex<V> v) throws IllegalArgumentException;
```

Check per vedere se modo è corretto o meno

20

20



Graph interface (2)

```
/**
 * Returns an iterable collection of edges for which vertex v is the destination.
 * Note that for an undirected graph, this is the same result
 * returned by outgoingEdges.
 * @throws IllegalArgumentException if v is not a valid vertex
 */
Iterable<Edge<E>> incomingEdges(Vertex<V> v) throws IllegalArgumentException;

/** Returns the edge from u to v, or null if they are not adjacent. */
Edge<E> getEdge(Vertex<V> u, Vertex<V> v) throws IllegalArgumentException;

/**
 * Returns the vertices of edge e as an array of length two.
 * If the graph is directed, the first vertex is the origin, and
 * the second is the destination.  If the graph is undirected, the
 * order is arbitrary.
 */
Vertex<V>[] endVertices(Edge<E> e) throws IllegalArgumentException;

/** Returns the vertex that is opposite vertex v on edge e. */
Vertex<V> opposite(Vertex<V> v, Edge<E> e) throws IllegalArgumentException;

/** Inserts and returns a new vertex with the given element. */
Vertex<V> insertVertex(V element);

/**
 * Inserts and returns a new edge between vertices u and v, storing given element.
 *
 * @throws IllegalArgumentException if u or v are invalid vertices, or if an edge already exists between u and v.
 */
Edge<E> insertEdge(Vertex<V> u, Vertex<V> v, E element) throws IllegalArgumentException;

/** Removes a vertex and all its incident edges from the graph. */
void removeVertex(Vertex<V> v) throws IllegalArgumentException;

/** Removes an edge from the graph. */
void removeEdge(Edge<E> e) throws IllegalArgumentException;
}
```

21

↳ Controlla se esiste arco

Ritorna array; Vertici estremi di arco sono sempre 2. Non ha senso ArrayList

↳ elemento con vertice creato

Arco duplicato ma Vertice possibile.

21

Per evitare di avere 2 vertici con lo stesso elemento devo implementarlo io. Lascia aperta possibilità di implementazione.