Graph

Corso: Strutture Dati

Docente: Annalisa De Bonis

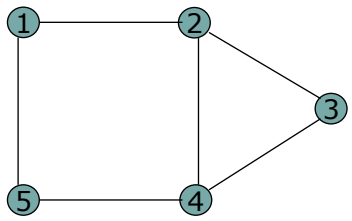
Grafi

- I grafi sono una particolare struttura dati usata spesso in informatica
  - Esempio più noto di grafo: World Wide Web
- Algoritmi che lavorano su grafi sono spesso fondamentali in molti campi dell'informatica
  - Esempio: i metodi di routing per l'instradamento di pacchetti su Internet hanno bisogno di conoscere i cammini più brevi per andare da un nodo all'altro della rete

Grafi

- Un grafo G=(V,E) è una coppia di insiemi
  V = insieme di nodi, chiamati vertici
  E = insieme di coppie di nodi, chiamati archi

Es: grafo



V={1,2,3,4,5}

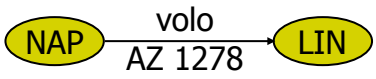
E={(1,2),(2,3),(2,4),(3,4),(4,5),(5,1)}

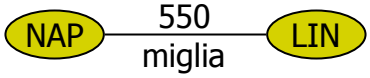
Applicazioni

- Circuiti elettronici
- Reti di trasporto
  - Rete autrostradali
  - Rete di rotte aeree
- Reti di computer
  - LAN
  - Internet
  - Web
- Database
  - Diagramma delle relazioni

Grafi direzionati e non

- Arco direzionato
  - Coppia ordinata di vertici (u,v)
  - u è l'origine
  - v è la destinazione
  - Esempio: un volo
- Arco non direzionato
  - Coppia non ordinata di vertici (u,v)
  - Esempio: una rotta aerea
- Grafo direzionato
  - Tutti gli archi sono direzionati
  - Esempio: rete dei voli
- Grafo non direzionato
  - Tutti gli archi sono non direzionati
  - Esempio: rete delle rotte aeree

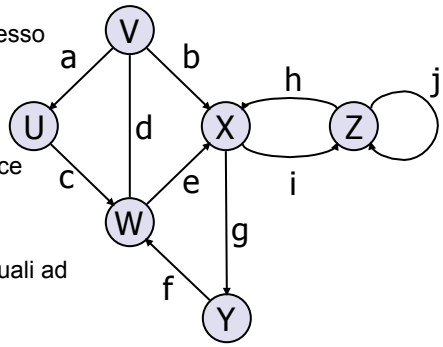




- NB: un grafo non direzionato può essere trasformato in un grafo direzionato sostituendo ogni arco (u,v) con i due archi direzionati (u,v) e (v,u)

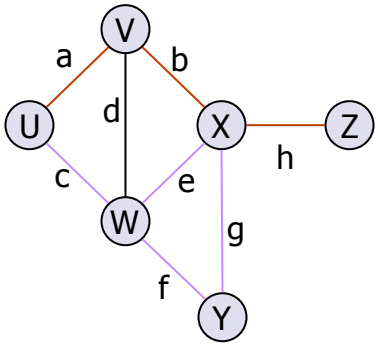
Terminologia

- Estremità di un arco
  - Vertici collegati dall'arco u e v
  - Es: **W** e **X** sono le estremità dell'arco **e**
- Archi incidenti su un vertice u
  - Archi con una delle due estremità uguali a u
  - Es: **e**, **b**, **g**, **h** ed **i** incidono su **X**
- Vertici adiacenti
  - Vertici che sono le estremità di uno stesso arco
  - Es: **W** e **X** sono adiacenti
- Grado di un vertice
  - Numero di archi che incidono sul vertice
  - Es: **X** ha grado 5
- Autociclo
  - Arco che ha entrambe le estremità uguali ad uno stesso vertice
  - Es: **j** è un autociclio



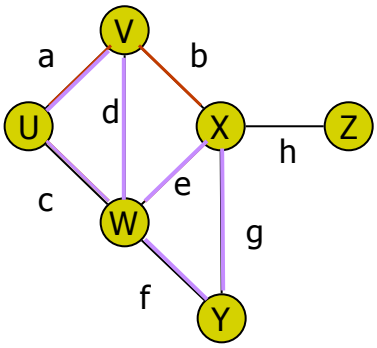
Terminologia

- Percorso
  - Sequenza che comincia e finisce con un vertice e in cui
  - si alternano vertici ed archi
  - ciascun arco è situato tra le sue stremità
- Percorso semplice
  - percorso in cui tutti i vertici e tutti gli archi sono distinti
- Esempi
  - P =(U,a,V,b,X,h,Z) è un percorso semplice
  - P= (U,c,W,f,Y,g,X,e,W) è un percorso che non è semplice



Terminologia

- Ciclo
  - Percorso che inizia e finisce nello stesso vertice
- Ciclo semplice
  - Ciclo in cui tutti i vertici sono distinti ad eccezione del primo e dell'ultimo
- Esempi
  - C=(V,b,X,e,W,c,U,a,V) è un ciclo semplice
  - C=(V,d,W,e,X,g,Y,f,W,c,U,a,V) è un ciclo che non è semplice



Proprietà

- Σ_v grado(v) = 2|E|
  - in quanto ogni arco incide su due vertici e quindi è contato due volte
- In un grafo non direzionato (senza autocicli) si ha

  $|E| \leq |V|(|V|-1)/2$
  - In quanto $|V|(|V|-1)/2$ è il numero massimo di coppie non ordinate che si possono formare con |V| elementi
- In un grafo direzionato si ha $|E| \leq |V|^2$
  - In quanto $|V|^2$ è il numero massimo di coppie ordinate che si possono formare con |V| elementi

Il TDA Graph: metodi di accesso

- numVertices()
  - restituisce il numero di vertici del grafo
- numEdges()
  - restituisce il numero di archi del grafo
- endVertices(e)
  - Restituisce un array contenente le due estremità di e
- opposite(v, e)
  - Restituisce il vertice incidente su e opposto al vertice v
- areAdjacent(v, w)
  - Restituisce true se e solo i vertici v e w sono adiacenti

Il TDA Graph: metodi che restituiscono collezioni iterabili

- incidentEdges(v)
  - Restituisce una collezione iterabile degli archi incidenti su v
- vertices()
  - Restituisce una collezione iterabile iteratore dei vertici nel grafo
- edges()
  - Restituisce una collezione iterabile degli archi nel grafo

Il TDA Graph: Metodi di aggiornamento

- replace(v, x)
  - Sostituisce l'elemento nel vertice v con x e restituisce l'elemento memorizzato precedentemente in v
- replace(e, x)
  - Sostituisce l'elemento nell'arco e con x e restituisce l'elemento memorizzato precedentemente in e
- insertVertex(o)
  - Inserisce restituendolo in output un vertice che memorizza l'elemento o
- insertEdge(v, w, o)
  - Inserisce restituendolo in output un arco (v,w) che memorizza l'elemento o
- removeVertex(v)
  - Cancella il vertice v ed i suoi archi incidenti e restituisce in output l'elemento di v
- removeEdge(e)
  - Cancella l'arco e restituisce in output l'elemento di e

Vertici e Archi

- Ciascun vertice e ciascun arco corrisponde ad una posizione all'interno del grafo

L'interfaccia graph

```
public interface Graph<V, E> {
  public int numVertices();

  public int numEdges();

  public Iterable<Vertex<V>> vertices();

  public Iterable<Edge<E>> edges();

  public V replace(Vertex<V> p, V o) throws
    InvalidPositionException;

  public E replace(Edge<E> p, E o) throws
    InvalidPositionException;

  public Iterable<Edge<E>> incidentEdges(Vertex<V> v)throws
    InvalidPositionException;
```

L'interfaccia graph

```
public Vertex[] endVertices(Edge<E> e) throws InvalidPositionException;

 public Vertex<V> opposite(Vertex<V> v, Edge<E> e)
  throws InvalidPositionException;

 public boolean areAdjacent(Vertex<V> u, Vertex<V> v)
  throws InvalidPositionException;

 public Vertex<V> insertVertex(V o);

 public Edge<E> insertEdge(Vertex<V> u, Vertex<V> v, E o)
  throws InvalidPositionException;

 public V removeVertex(Vertex<V> v) throws InvalidPositionException;

 public E removeEdge(Edge<E> e) throws InvalidPositionException;
}
```


Il TDA Directed Graph

- In aggiunta ai metodi del TDA Graph, il TDA Directed Graph ha anche i seguenti metodi:
  - isDirected(e)
    - restituisce vero se l'arco e è direzionato
  - insertDirectedEdges(u,v,o)
    - aggiunge al grafo l'arco direzionato (u,v) avente come elemento o
- Si noti che se e è un arco direzionato allora il metodo endVertices(e) deve restituire un array A tale che A[0] contiene l'origine dell'arco e A[1] contiene la destinazione dell'arco.

Il TDA Directed Graph

- Potrebbe essere utile aggiungere i metodi
  - inIncidentEdges(v)
    - restituisce una collezione iterabile degli archi entranti in v
  - outIncidenceEdges(v)
    - restituisce una collezione iterabile degli archi uscenti da v

L'interfaccia DirectedGraph

```
public interface DirectedGraph<V, E> extends
   Graph<V,E>{

 public boolean isDirected(Edge<E> e);

 public insertDirectedEdge(Vertex<V> u, Vertex<V>v,V o);

}
```

Grafi pesati e non

- Rappresenteremo i grafi pesati e quelli non pesati nello stesso modo
- Nel caso di un grafo pesato, l'elemento memorizzato nella posizione Edge sarà il peso dell'arco stesso
- Nel caso di un grafo non pesato, l'elemento contenuto nella posizione Edge sarà null

Implementazioni di Graph

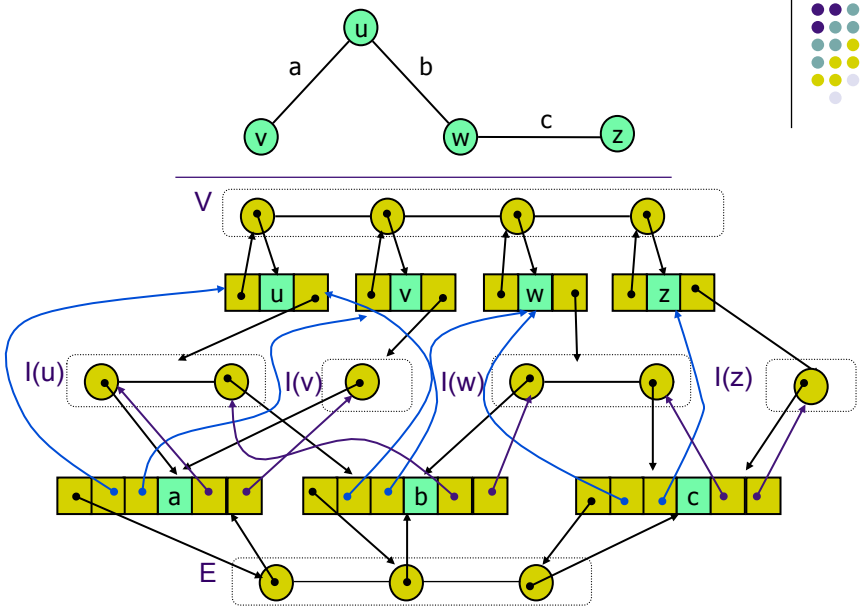
- Le loro implementazioni cambiano a seconda della rappresentazione che si utilizza per Graph
  - Lista di archi (*Edge List*)
  - Matrice di Adiacenza (*Adjacent Matrix*)
  - Liste di Adiacenza (*Adjacent List*)

Liste di adiacenza

- Consiste di
  - collezione di tutti i vertici
  - collezione di tutti gli archi,
  - per ciascun vertice v, collezione I(v) degli archi incidenti su v
- Ciascun vertice v contiene un riferimento alla collezione I(v) degli archi incidenti su v
- Ciascun arco e=(u,v) contiene i riferimenti alle posizioni (o alle entrate) relative ad e nelle collezioni I(u) e I(v)
- Tipicamente le collezioni I(v) vengono rappresentate con delle liste.

Implementazione di Vertex e di Edge

- L'implementazione di Vertex contiene
  - riferimento all'elemento,
  - riferimento a I(v)
    - **protected** PositionList<Edge<E>> incEdges;
  - riferimento alla sua posizione nella lista dei vertici
    - **protected** Position<Vertex<V>> loc;
- L'implementazione di Edge contiene
  - riferimento all'elemento
  - riferimenti alle posizioni relative ad e=(u,v) nelle liste I(u) e I(v)
    - **protected** Position<Edge<E>>[] Inc
  - riferimento alla sua posizione nella lista degli archi
    - **protected** Position<Edge<E>> loc;
  - riferimenti alle posizioni dei vertici u e v nella lista dei vertici
    - **protected** MyVertex<V>[] endVertices



Bozza della classe AdjacencyListGraph

```
public class AdjacencyListGraph<V,E> implements Graph<V,E> {

 protected NodePositionList<Vertex<V>> VList;// lista dei vertici
 protected NodePositionList<Edge<E>> EList;// lista degli archi

 /** costruttore di default: crea un grafo vuoto */
 public AdjacencyListGraph() {
   VList = new NodePositionList<Vertex<V>>();
   EList = new NodePositionList<Edge<E>>();
 }
```

Bozza della classe AdjacencyListGraph

```
public Iterable<Vertex<V>> vertices() {
   return VList;
 }

public Iterable<Edge<E>> edges() {
   return EList;
 }
```


Bozza della classe AdjacencyListGraph

```
public Edge<E> insertEdge(Vertex<V> v, Vertex<V> w, E o)
  throws InvalidPositionException {
  MyVertex<V> vv = checkVertex(v);
  MyVertex<V> ww = checkVertex(w);
  MyEdge<E> ee = new MyEdge<E>(v, w, o);

 /** Inserimento dell'arco nelle liste di incidenza dei vertici */
  Position<Edge<E>> pv = vv.insertIncidence(ee);
  Position<Edge<E>> pw = ww.insertIncidence(ee);
 /**  setta i riferimenti alle position dell'arco nelle liste di incidenza dei
   vertici */
  ee.setIncidences(pv, pw);

  EList.addLast(ee); //aggiunge l'arco alla lista degli archi EList
  Position<Edge<E>> pe = EList.last();
  ee.setLocation(pe); //setta il riferimento alla position dell'arco in EList
  return ee;
}
```

Bozza della classe AdjacencyListGraph

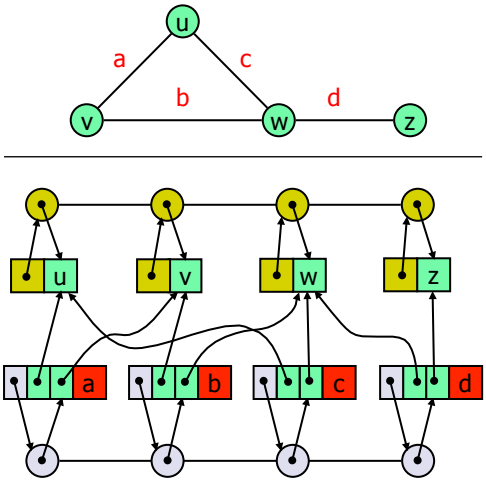
```
public E replace(Edge<E> p, E o) throws
  InvalidPositionException {
  E temp = p.element();
  MyEdge<E> ee = checkEdge(p);
  ee.setElement(o);
  return temp;
}
```


Altre rappresentazioni

- Per rappresentare i grafi potremmo utilizzare
  - La lista di archi
  - La matrice di adiacenza

Rappresentazione mediante lista di archi

- Il grafo è rappresentato tramite due liste
  - Una lista (Node List) o un vettore (Array List) conserva i vertici del grafo
  - Un'altra lista (Node List) o vettore (Array List) conserva gli archi del grafo
- Per effettuare facilmente la ricerca degli archi si potrebbe utilizzare un dizionario per rappresentare la collezione degli archi invece che una lista.
  - L' entrata del dizionario associata all'arco e ha come chiave l'elemento dell'arco e e come valore l'arco e.
  - Si può fare la stessa cosa per i vertici

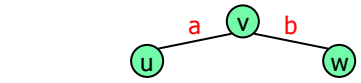


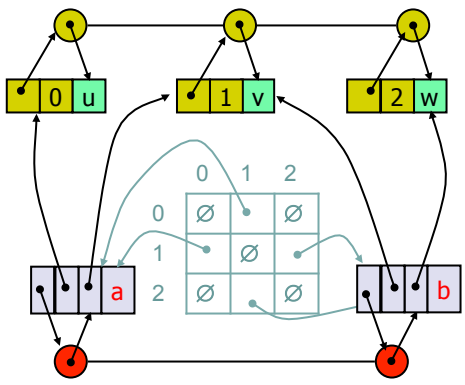
Rappresentazione mediante lista di archi

- Richiede di scandire tutti gli archi del grafo per trovare gli archi incidenti su un certo vertice

Matrice di adiacenza

- Lista degli archi
  - Come per il caso precedente
- Lista dei vertici
  - Sequenza di vertici
  - Ogni vertice contiene una chiave intera (indice)
- Matrice di adiacenza
  - Contiene un riferimento all'arco per vertici adiacenti
  - Contiene null per vertici non adiacenti





Matrice di adiacenza

- Permette di scoprire velocemente se due vertici sono adiacenti
- Richiede spazio $O(n^2)$

n = # nodi —— m = #archi

	Lista di archi	Liste di Adiacenza	Matrice di Adiacenza
Spazio	$n + m$	$n + m$	n^2
incidentEdges(***v***)	***m***	1	***n***
areAdjacent (***v, w***)	***m***	min(grado(***v***), grado (***w***))	1
insertVertex(***o***)	1	1	n^2
insertEdge(***v, w, o***)	1	1	1
removeVertex(***v***)	***m***	grado(***v***)	n^2
opposite(***v***,***e***), endVertices(***e***)	1	1	1
replace(***e,x***), replace(***v,x***)	1	1	1
removeEdge(***e***)	1	1	1

Il Design Pattern Decorator

- Serve ad “attaccare” informazioni extra ad un oggetto
  - Le informazioni vengono aggiunte dinamicamente
  - Ciascuna informazione consiste di una coppia (attributo, valore)
    - Esempio: Per implementare la DFS e la BFS è utile “decorare’’ i vertici e gli archi con l’attributo ***esplorato*** a cui è associato un valore booleano.

Il Design Pattern Decorator

- Alternative:
  - Avremmo potuto inserire il campo **esplorato** di tipo boolean nell’implementazione di Vertex ed Edge
    - Problema: l’implementazione del grafo non è più generica ma dipende dall’uso che intendiamo fare del grafo. Ad esempio, nel nostro caso vogliamo effettuare una visita DFS del grafo
  - Avremmo potuto usare una tabella hash contenente gli archi e i vertici esplorati
    - Problema: nel caso pessimo tempo non costante per marcare “esplorato’’ un vertice o un arco

Il TDA Decorable Position

- Possiamo realizzare un decoratore per un contenitore posizionale definendo il TDA Decorable Position che unisce i metodi del TDA Map a quelli del TDA Position:
  - element()
  - size()
  - isEmpty()
  - entries() : restituisce tutte le coppie (attributo,valore) associate alla posizione
  - get(**a**) : restituisce il valore dell'attributo **a**
  - put(**a**,**x**): pone il valore di **k** uguale a **x**
  - remove(**a**): rimuove dalla posizione l'attributo **a** e il suo valore

L'interfaccia DecorablePosition

```
public interface DecorablePosition<E>
      extends Position<E>, Map<Object,Object> {
}
```

Implementazione di Prim e Dijkstra con AdaptablePriorityQueue

- Possiamo attaccare a ciascun vertice u l'entrata associata ad u nella coda a priorità Q
  - Il vertice u conterrà l'attributo ENTRY il cui valore è l'entrata (k(u),u) associata ad u nella coda a priorità:
    - Inserimento di u nella coda a priorità:
      - Q.insert(k,u) restituisce la LocationAwareEntry e associata al vertice in Q
      - u.put(ENTRY, e) associa al vertice u l'attributo ENTRY avente come valore l'entrata e.

L'interfaccia Vertex

```
public interface Vertex <E>extends
  DecorablePosition<E>{}
```

La classe MyPosition

```
protected static class MyPosition<T>    extends
    HashTableMap<Object,Object> implements DecorablePosition<T> {

  protected T elem;

   public T element() {
     return elem;
   }

  public void setElement(T o) {
     elem = o;
   }
}
```


La classe MyVertex

```
protected  class MyVertex<V>  extends MyPosition<V>
    implements Vertex<V> {
  /** riferimento alla lista di incidenza del vertice */
  protected PositionList<Edge<E>> incEdges;
  /**  riferimento alla posizione del vertice nella lista dei vertici. */
  protected Position<Vertex<V>> loc;

 MyVertex(V o) {
    elem = o;
    incEdges = new NodePositionList<Edge<E>>();
  }
public int degree() {
    return incEdges.size();
  }
 public Iterable<Edge<E>> incidentEdges() {
    return incEdges;
  }
```

La classe MyVertex

```
/** Inserisce un arco nella lista di incidenza. */
  public Position<Edge<E>> insertIncidence(Edge<E> e) {
    incEdges.addLast(e);
    return incEdges.last();   }

/** Rimuove un arco dalla lista di incidenza. */
  public void removeIncidence(Position<Edge<E>> p)
 {   incEdges.remove(p);   }

  public Position<Vertex<V>> location() {   return loc;   }

  /** Setta la posizione del vertice nella lista dei vertici. */
  public void setLocation(Position<Vertex<V>> p) { loc = p;}

 public String toString() {  return elem.toString();   }
}
```


La classe MyEdge

```
protected class MyEdge<E> extends MyPosition<E>
    implements Edge<E> {
  /** estremità dell'arco. */
  protected MyVertex<V>[] endVertices;
  /** posizioni relative all'arco nelle liste di incidenza delle
   estremità dell'arco */
  protected Position<Edge<E>>[] Inc;
  /** posizione dell'arco nella lista degli archi. */
  protected Position<Edge<E>> loc;

 MyEdge (Vertex<V> v, Vertex<V> w, E o) {
    elem = o;
    endVertices = (MyVertex<V>[]) new MyVertex[2];
    endVertices[0] = (MyVertex<V>)v;
    endVertices[1] = (MyVertex<V>)w;
    Inc = (Position<Edge<E>>[]) new Position[2];
  }
  /** restituisce un array contenente le estremità dell'arco */
  public MyVertex<V>[] endVertices() {
    return endVertices;
  }
```

La classe MyEdge

```
public Position<Edge<E>>[] incidences() {
    return Inc;
  }

public void setIncidences(Position<Edge<E>> pv, Position<Edge<E>> pw) {
    Inc[0] = pv;
    Inc[1] = pw;
  }
public Position<Edge<E>> location() {
    return loc;
  }
public void setLocation(Position<Edge<E>> p) {
    loc = p;
  }
public String toString() {
    return element() + "(" + endVertices[0].toString() +
"," + endVertices[1].toString() + ")";
  }
}
```


I metodi checkPosition e checkVertex

```
protected MyPosition checkPosition(Position p)
  throws InvalidPositionException {
  if (p == null || !(p instanceof MyPosition))
    throw new InvalidPositionException("posizione non valida");
  return (MyPosition) p;
}


protected MyVertex<V> checkVertex(Vertex<V> v)
  throws InvalidPositionException {
 if (v == null || !(v instanceof MyVertex))
  throw new InvalidPositionException("vertice non valido");
 return (MyVertex<V>) v;
}
```

Esercizio

- Scrivere la classe AdjacencyListGraph che implementa l'interfaccia Graph mediante liste di adiacenza (da inserire nel progetto)
- Scrivere la classe AdjacencyListDGraph che implementa l'interfaccia Directed Graph mediante liste di adiacenza