1 I Grafi

1.1 Introduzione

Cosa sono i grafi?

Un grafo è una struttura che serve per rappresentare relazioni tra elementi.
I grafi sono considerati insiemi di entità, dette **nodi** o **vertici**, che sono collegate a coppie tra loro da delle relazioni. Queste relazioni vengono dette **lati** o **archi**, e a questi possono essere attribuiti dei **pesi**, dando origine a grafi detti **grafi pesati**.

Un **grafo orientato** G è dunque una coppia (V, E) dove V è un insieme finito ed E è una relazione binaria in V.
Gli elementi dell'insieme V sono detti **vertici**, l'insieme V è quindi chiamato **insieme dei vertici** di G. Allo stesso modo gli elementi dell'insieme E sono detti **archi** e l'insieme E è quindi chiamato **insieme degli archi** di G.

Grafi orientati e non orientati

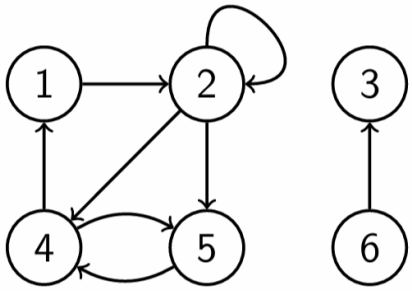
In un **grafo orientato**, ogni arco ha una direzione: significa che il collegamento va da un vertice a un altro, in un verso preciso.
In un **grafo non orientato**, invece, gli archi non hanno direzione: il collegamento tra due vertici vale in entrambi i sensi.
Quindi, mentre in un **grafo orientato** $G = (V, E)$, l'insieme degli archi E è composto da coppie ordinate di vertici, in un **grafo non orientato** $G = (V, E)$, l'insieme degli archi E è composto da coppie di vertici non ordinate.

1.1.1 Definizioni utili

- In un grafo **non orientato**, quando un arco (u, v) collega due vertici (in questo caso u e v), diciamo che l'arco è **incidente** nei vertici u e v, e che u e v sono **adiacenti** tra loro (cioè "collegati"). In questo caso, dato che il grafo non è orientato, l'adiacenza è bidirezionale, perchè anche l'arco è percorribile in entrambe le direzioni (da u a v e da v a u).
- In un grafo **orientato**, l'arco $[u, v]$ rappresenta la connessione dal vertice u al vertice v. In questo caso diciamo che l'arco è **incidente** nei vertici u e v ma con verso diverso, **uscente** da u ed **entrante** in v. In questo caso l'adiacenza è monodirezionale, segue il verso dell'arco, quindi diciamo che u è **adiacente** a v ma non è necessariamente vero il contrario (cioè a meno che non esista anche un arco $[v, u]$).
- Il **grado** di un vertice in un grafo **non orientato** coincide con il numero di archi incidenti in esso. Se un vertice ha grado 0, è **isolato**.
- In un grafo **orientato**, per un vertice, possone essere distinti tre tipi di grado. Il **grado uscente** di un vertice, è il numero di archi uscenti da esso; mentre il **grado entrante** è il numero di archi entranti nel vertice. Il **grado** "totale" del vertice è dato dalla somma di grado entrante e grado uscente, quindi dalla somma di tutti gli archi incidenti in esso.
- Un **sottografo** è un grafo formato da un sottoinsieme dei vertici e degli archi di un dato grafo, che mantiene la stessa struttura di connessioni.

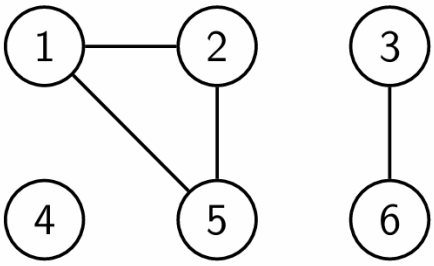
- Un arco che entra ed esce dallo stesso vertice è detto **cappio**.

1.1.2 Esempi



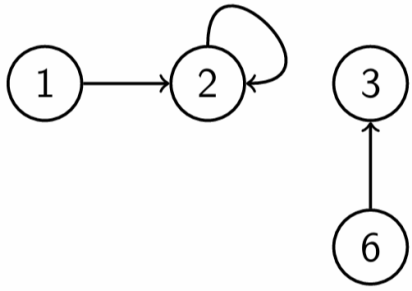
In questo caso abbiamo un grafo **orientato** con 6 nodi ($V = \{1, 2, 3, 4, 5, 6\}$) e 8 archi $E = \{[1,2], [2,2], [2,4], [2,5], [4,1], [5,4], [6,3]\}$.

Dalle definizioni precedenti possiamo notare che l'arco $[2,2]$ è un **cappio**.



In questo caso abbiamo un grafo **non orientato** con 6 nodi ($V = \{1, 2, 3, 4, 5, 6\}$) e 4 archi $E = \{(1,2), (1,5), (2,5), (3,6)\}$.

Dalle definizioni precedenti possiamo notare che il vertice 4 è **isolato**.



In questo caso abbiamo un **sottografo** del grafo orientato del primo esempio.

1.1.3 Cammini e raggiungibilità

Cosa si intende per *cammino*?

Un **cammino** è un modo per "muoversi" all'interno di un grafo passando da un vertice all'altro seguendo gli archi che li collegano.
Per essere più precisi, un cammino di lunghezza k che va da un vertice u a un vertice u' (in un grafo $G = (V, E)$), è una sequenza di vertici $\langle v_0, v_1, v_2, ..., v_k \rangle$ che rispetta tre condizioni:

- Il primo vertice è il punto di partenza: $u = v_0$;
- L'ultimo vertice è il punto di arrivo: $u' = v_k$;
- Ogni coppia di vertici consecutivi nella sequenza è collegata da un arco del grafo, cioè: $(v_{i-1}, v_i) \in E$ per $i = 1, 2, ..., k$

Conoscendo la definizione di **cammino**, possiamo dare la definizione di **lunghezza del cammino**. La lunghezza di un cammino è semplicemente il numero di archi che lo compongono. Quindi, se ho k archi, il cammino avrà lunghezza k.
Possiamo inoltre dire che un cammino è **semplice** se **tutti i vertici che attraversa sono distinti**, cioè non si passa mai due volte dallo stesso vertice.

Cosa si intende per *raggiungibilità*?

Diciamo che un vertice u' è **raggiungibile** da un altro vertice u attraverso un cammino p, se esiste un cammino p che parte dal vertice u e arriva al vertice u'.
Se il grafo è orientato il cammino p si indica $u \xrightarrow{p} u'$ (con la freccetta ondulata) e indica quindi un cammino che rispetti la direzione degli archi.

Cosa è un *sottocammino*?

Un **sottocammino** si un cammino $p = \langle v_0, v_1, v_2, ..., v_k \rangle$, è una sottosequenza contigua dei suoi vertici. Questo vuol dire che non posso prendere ad esempio il primo e l'ultimo vertice saltando quelli in mezzo. Quindi ad esempio $\langle v_3, v_4, v_5 \rangle$ è un sottocammino di p, mentre $\langle v_3, v_6 \rangle$ non lo è.

Cosa è un *ciclo*?

In un grafo orientato $G = (V, E)$, un **ciclo** C di lunghezza $k > 2$ è una sequenza di nodi $u_0, u_1, ..., u_k$ tale che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k-1$ e $u_0 = u_k$. ($k > 2$ esclude cicli banali composti da coppie di archi (u, v) e (v, u), che sono onnipresenti nei grafi non orientati)

Cicli: altre definizioni

In altre parole, in un **grafo orientato**, un cammino $\langle v_0, v_1, v_2, ..., v_k \rangle$ forma un **ciclo** se $v_0 = v_k$ e il cammino **contiene almeno un arco**.
Quindi il **cappio** (o loop) che abbiamo visto prima, è un ciclo di lunghezza 1.
Inoltre se il cammino che forma il ciclo è **semplice**, anche il ciclo viene detto semplice.

Invece, in un **grafo non orientato**, un cammino $\langle v_0, v_1, v_2, ..., v_k \rangle$ forma un **ciclo** (semplice) se: $v_0 = v_k$, $k \geq 3$, $v_1, ..., v_k$ sono distinti.
Si dice inoltre che un **grafo orientato senza cappi** è **semplice** mentre un grafo senza cicli è **aciclico**.

1.1.4 Connessione

La definizione si connessione è diversa per grafo orientato (per il quale bisogna fare una distinzione in più tra connessione **debole** e **forte**) e non orientato:

Quando un grafo non orientato di dice *connesso*?

Un grafo non orientato si dice **connesso** se ogni coppia di vertici è collegata tramite un cammino (eventualmente minimo). Ovvero se, per ogni coppia di vertici distinti $u, v \in V$ esiste almeno un cammino che collega u e v.

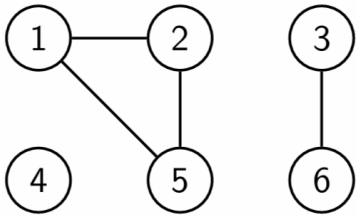
N.B. I vertici non devono essere connessi tutti direttamente (non devono necessariamente essere *adiacenti*), basta che ci sia un cammino che ti permetta di raggiungere un qualunque nodo (vertice) del grafo partendo da un qualunque altro nodo (vertice) dello stesso grafo.

Quando un grafo **orientato** si dice *connesso*?

Un grafo orientato si dice **fortemente connesso** se due vertici qualsiasi sono raggiungibili l'uno dall'altro; quindi esiste un cammino orientato da u a v e viceversa.
È detto invece **debolmente connesso** se, ignorando la direzione degli archi (cioè trattandolo come se fosse non orientato), il grafo risultante è connesso.

Un'altra definizione importante è quella di **componente connessa**. Le **componenti connesse** di un grafo "sono le classi di equivalenza dei vertici secondo la relazione di (mutua) raggiungibilità"; ma vediamo cosa vuol dire per grafi orientati e non.

Quando un grafo non orientato **non è connesso**, può essere suddiviso in più parti isolate, dette **componenti connesse**. In pratica, all'interno di un grafo non connesso, possiamo individuare diversi sottografi massimali che, presi singolarmente, sono connessi. Diciamo sottografi **massimali** perché questo implica che l'aggiunta di un altro qualsiasi nodo del grafo non gli fa più rispettare la condizione di connessione. Facciamo un esempio:

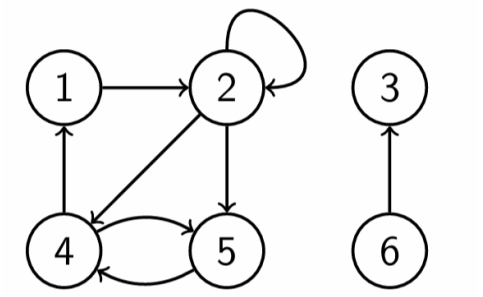


In questo esempio i nodi $\langle 1, 2, 5 \rangle$ formano una componente connessa. Infatti se prendiamo il sottografo formato solo da questi 3 vertici, questo risulta connesso, e non possiamo aggiungere nessun altro vertice del grafo continuando ad avere una condizione di connessione.
Se invece prendiamo come sottografo quello formato dai vertici $\langle 1, 2 \rangle$ e la connessione tra i due, questo non rappresenta una componente connessa in quanto se aggiungiamo il vertice 5 e i rispettivi archi di collegamento con i vertici 1 e 2, continua a mantenere la proprietà di connessione.
In questo esempio abbiamo difatti 3 componenti connesse:

- $\langle 1, 2, 5 \rangle$
- $\langle 3, 6 \rangle$
- $\langle 4 \rangle$

Grazie a questa definizione possiamo dire che un grafo non orientato è **connesso** se ha **una sola componente connessa**.

Dato che un grafo orientato può essere debolmente o fortemente connesso, dobbiamo dare la definizione di **componente fortemente connessa**. Funziona esattamente come nel caso precedente, solo che adesso gli archi sono direzionali, quindi il sottografo che prendiamo in esame dev'essere fortemente connesso. Anche in questo caso, un grafo si dice **fortemente connesso** se presenta **una sola componente fortemente connessa**. Anche in questo caso facciamo un esempio:



Anche in questo caso abbiamo 3 componenti fortemente connesse:

- $\langle 1, 2, 4, 5 \rangle$
- $\langle 3 \rangle$
- $\langle 6 \rangle$

3 e 6 non formano una componente fortemente connessa perché il nodo 3 è raggiungibile dal 6 ma non viceversa.
Invece $\langle 1, 2, 4, 5 \rangle$ formano una componente fortemente connessa perché ogni nodo è raggiungibile da un'altro dell'insieme. Ad esempio il nodo 1 è

collegato direttamente con il nodo 2 e, passando da questo nodo si possono raggiungere sia il nodo 4 che il nodo 5. Stesso ragionamento per gli altri nodi.

1.1.5 Dimensioni del grafo

Per descrivere un grafo, sono necessarie due quantità fondamentali: il numero di nodi e di archi nel grafo. Se si considerano $n = |V|$ come il numero totale di nodi nel grafo e $m = |E|$ come il numero totale di archi del grafo, è possibile definire alcune relazioni importanti tra queste due quantità, a seconda che il grafo sia orientato o meno:

- In un **grafo non orientato**, il numero di archi possibili è $m \leq \frac{n(n-1)}{2}$ perché ogni coppia di nodi può essere collegata al massimo da un arco. Quindi il numero di archi cresce, al massimo, proporzionalmente a n^2 ($O(n^2)$)
- In un **grafo orientato**, ogni coppia di nodi può avere due archi distinti (uno per ciascuna direzione), quindi il numero di archi è $m \leq n^2 - n$, che cresce anch'esso come n^2 ($O(n^2)$)

Quando si analizzano algoritmi che operano sui grafi, la loro complessità viene spesso espressa in funzione sia del numero di nodi n sia del numero di archi m. Ad esempio, un algoritmo può avere complessità $O(m + n)$, cioè proporzionale alla somma di nodi e archi del grafo, e questo può avvenire quando l'algoritmo deve "guardare" tutti i nodi e tutti gli archi almeno una volta.

1.1.6 Grafi sparsi e densi

Possiamo "catalogare" un grafo in base alle sue caratteristiche.
Ad esempio quando un grafo ha un arco tra tutte le coppie di nodi, (cioè quando ogni nodo è collegato "direttamente" a **tutti** gli altri nodi del grafo) è detto **completo**.
Un grafo inoltre può essere **sparso** o **denso**:

- Un grafo è detto **sparso** se ha pochi archi; grafi con ad esempio $m = O(n)$ o $m = O(n \log n)$ sono considerati sparsi.
- Un grafo è detto **denso** quando ha molti archi, come ad esempio $m = \Omega(n^2)$

Possiamo quindi dire che un grafo **completo** è anche **denso**, perché è quello che contiene il numero massimo di archi possibili rispetto ai nodi ed ha quindi la densità più alta che un grafo può avere. Ovviamente non è vero anche il contrario, un grafo denso non necessariamente è completo.

1.1.7 Isomorfismo

Introduciamo ora questa relazione tra 2 grafi:

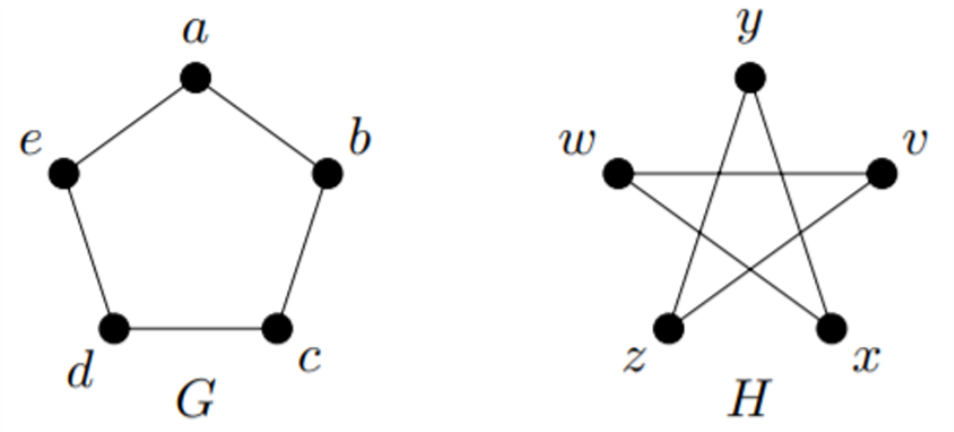
Quando si dice che due garfi sono *isomorfi*?

Due grafi $G = (V, E)$ e $G' = (V', E')$ sono **isomorfi** se esiste una funzione biiettiva (cioè una corrispondenza uno-a-uno) $f : V \to V'$ tale che $(u, v) \in E$ e se e soltanto se $(f(u), f(v)) \in E'$.
Cioè se è possibile rietichettare i vertici di G come i vertici di G' mantenendo gli archi corrispondenti in G e G'.

In altre parole possiamo dire che due grafi sono isomorfi se:
- hanno lo stesso numero di vertici e di archi;
- la connessione tra i vertici è la stessa, anche se i nomi o la disposizione dei vertici cambiano.

Quindi se due vertici sono adiacenti in un grafo lo saranno anche nell'altro.

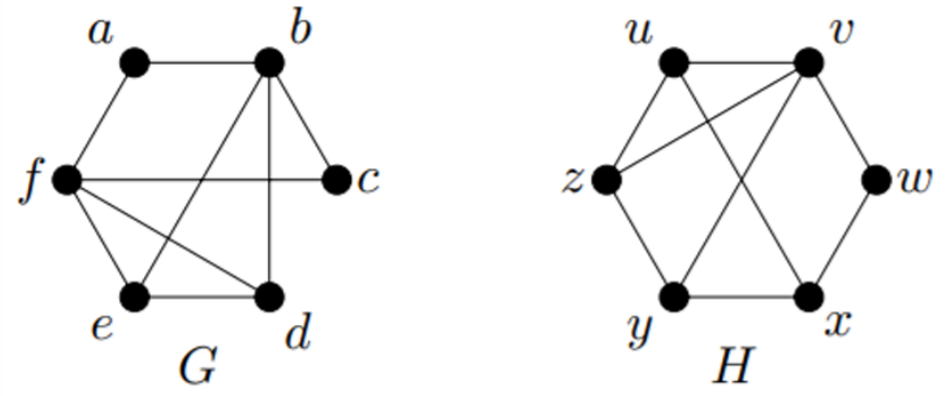
Per fissare meglio il concetto vediamo alcuni esempi:



Questi due grafi sono **isomorfi**. Innanzitutto possiamo notare che hanno lo stesso numero di nodi e di archi, oltretutto tutti e 5 i nodi di entrambi i grafi hanno valenza (o grado) 2.
Nel garfo G abbiamo il nodo a adiacente ai nodi b ed e, il nodo c adiacente a b e d e quest'ultimo adiacente al nodo e; nel grafo H, se partiamo dal nodo v, notiamo che questo è adiacente ai nodi w e z, il nodo x adiacente a w e y e questultimo adiacente al nodo z.

Possiamo quindi vedere una corrispondenza del tipo:
- $a \Longleftrightarrow v$ quindi definiamo $f(a) = v$
- $b \Longleftrightarrow w$ quindi definiamo $f(b) = w$
- $c \Longleftrightarrow x$ quindi definiamo $f(c) = x$
- $d \Longleftrightarrow y$ quindi definiamo $f(d) = y$
- $e \Longleftrightarrow z$ quindi definiamo $f(e) = z$

Vediamo un altro esempio:



In questo caso invece i due grafi **non sono isomorfi**. Infatti, nonostante abbiano entrambi 6 nodi e 9 archi, il grafo G ha 2 vertici di valenza 2 (a e c), 2 vetici di valenza 3 (d ed e) e 2 vertici di valenza 4 (b ed f), mentre il grafo H ha un vertice di valenza 2 (w), 4 vertici di valenza 4 (u, x, y e z) quindi non è possibile trovare un'equivalenza come invece è stato fatto nel caso precedente. (Ricordiamo che il grado di valenza dei vertici in un grafo non orientato è il numero di archi incidenti in esso)

1.1.8 Relazioni tra alberi e grafi

Esistono delle relazioni tra alberi e grafi, infatti, quando un grafo rispetta determinate caratteristiche, può essere considerato un albero.
Il primo tipo di albero che andiamo a considerare, quello con i requisiti meno stringenti, è il cosiddetto *albero libero*.

Quando un grafo è un albero libero?

Un grafo per poter essere considerato un **albero libero** deve rispettare due requisiti fondamentali:

- **deve essere connesso** $\rightarrow$ deve essere possibile raggiungere qualsiasi nodo partendo da un qualsiasi altro nodo, seguendo gli archi;
- **deve essere aciclico** $\rightarrow$ non devono esistere cicli, quindi percorsi chiusi che ti permettano, partendo da un nodo, di tornare allo stesso senza ripercorrere gli stessi archi.

Queste due proprietà possono essere riassunte dicendo che il numero di archi m deve essere esattamente uno in meno rispetto al numero dei nodi n: $m = n - 1$. Infatti, se ci fossero meno archi il grafo non sarebbe connesso, e se ce ne fossero di più il grafo conterrebbe almeno un ciclo.

Una volta che si ha in mente cosa è un albero libero, possiamo capire cosa si intende per *albero radicato*.

Quando invece è un albero radicato?

Un **albero radicato** infatti è un albero libero nel quale è stato scelto un nodo come radice. A questo punto possiamo immaginare i nodi come disposti su più livelli, in base alla loro distanza dalla radice. Avremo quindi la radice su un livello, i suoi figli sul livello successivo, i figli dei suoi figli in quello dopo ancora, e così via. Viene stabilito quindi un ordinamento "verticale" tra i nodi.

A partire dagli alberi radicati possiamo definire un'ulteriore tipo di grafi che possono essere visti come alberi, ossia gli *alberi ordinati* (*ordered tree*)

Quando è un albero ordinato?

Un **albero ordinato** è un albero radicato nel quale viene deciso un ordine preciso tra i nodi sullo stesso livello, quindi tra i nodi figli di un certo nodo. In questo caso quindi il grafo dovrà rispettare tutte le caratteristiche elencate per i precedenti due, viene però aggiunto un ordinamento "orizzontale" tra i nodi.

Possiamo ora passare alla definizione di ciò che viene detta *foresta*.

Cosa è una foresta?

Una **foresta** è banalmente un insieme di alberi. È quindi un grafo non connesso, che presenta diverse componenti connesse, che rispettano le caratteristiche elencate sopra per poter essere considerati alberi.

1.2 Specifica

Prima di iniziare diamo una brevissima spiegazione di ciò che intendiamo con "specifica" in questo contesto. Una specifica è una descrizione formale delle operazioni che una struttura dati deve offrire, senza dire come è implementata.
In pratica daremo una descrizione astratta delle operazioni disponibili e dei loro effetti, senza entrare nel dettaglio della loro implementazione.

Altra piccola parentesi: poiché ogni grafo non orientato G può essere visto come un grafo orientato G', ottenuto da G, sostituendo ogni arco $[u, v]$ con i due archi (u, v) e (v, u), sarà fornita una specifica solo per grafi orientati, che saranno chiamati semplicemente grafi.

1.2.1 Grafi dinamici

Nella versione più generale, il grafo è una struttura dati dinamica che permette di aggiungere e rimuovere nodi e archi,per la quale non esiste uno standard universalmente adottato. Possiamo quindi vedere le principali operazioni che possono essere fatte sui grafi:

```
1 Graph()                      //Crea un nuovo grafo
2 SET V()                      //Restituisce l'insieme di tutti i nodi
3 int size()                   //Restituisce il numero dei nodi
4 Set adj(NODE u)              //Restit. l'insieme dei nodi adiacenti a u
5 insertNode(NODE u)           //Aggiunge il nodo u al grafo
6 insertEdge(NODE u, NODE v)   //Aggiunge l'arco (u,v) al grafo
7 deleteNode(NODE u)           //Rimuove il nodo u dal grafo
8 deleteEdge(NODE u, NODE v)   //Rimuove l'arco (u,v) dal grafo
```

1.2.2 Specifica ridotta - senza rimozioni

In alcuni casi non è necessaria una struttura dinamica completa. Spesso, infatti, si utilizza un grafo che viene caricato inizialmente e che non viene più modificato, tranne eventualmente per l'aggiunta di nuovi nodi o archi. In questi casi utilizziamo grafi non dinamici, in cui sono permessi solo inserimenti ma non rimozioni.

```
1 Graph()                      //Crea un nuovo grafo
2 SET V()                      //Restituisce l'insieme di tutti i nodi
3 int size()                   //Restituisce il numero dei nodi
4 Set adj(NODE u)              //Restit. l'insieme dei nodi adiacenti a u
5 insertNode(NODE u)           //Aggiunge il nodo u al grafo
6 insertEdge(NODE u, NODE v)   //Aggiunge l'arco (u,v) al grafo
```

Questo ha conseguenze anche sull'implementazione del grafo stesso.

1.3 Memorizzazione

La memorizzazione di un grafo può avvenire tramite uno dei due possibili approcci:
- tramite **matrici di adiacenza**
- tramite **liste di adiacenza**

1.3.1 Matrici di adiacenza

La memorizzazione di un grafo tramite matrici di adiacenza, è uno dei modi più semplici per rappresentare un grafo.

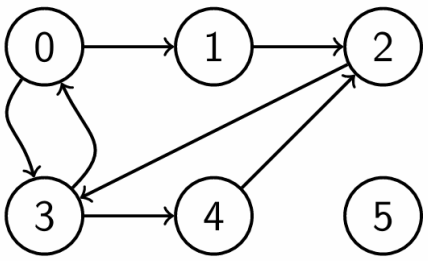
Matrice di adiacenza

Dato un grafo $G = (V, E)$, la **matrice di adiacenza** $M = [m_{uv}]$, è una matrice $n \times n$ tale che

$$m_{uv} = \begin{cases} 1 & se\ (u, v) \in E \\ 0 & se\ (u, v) \notin E \end{cases}$$

Grafi orientati

Quando il grafo è orientato lo spazio in memoria occupato dalla matrice è n^2 bit



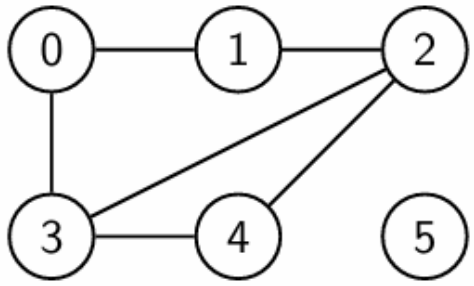
	0	1	2	3	4	5
0	0	1	0	1	0	0
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	1	0	0	0	1	0
4	0	0	1	0	0	0
5	0	0	0	0	0	0

Come possiamo vedere dall'esempio, abbiamo un bit per ogni possibile arco, che è a 0 se l'arco tra quei nodi non esiste ed è a 1 quando l'arco esiste. In questo caso abbiamo i nodi di partenza degli archi scritti in colonna (che "numerano" quindi le righe) e i nodi di arrivo scritti in riga (che "numerano" quindi le colonne). Possiamo quindi vedere come esiste un arco da 0 a 1 in quanto il bit nella posizione [0][1] (riga 0 e colonna 1) è a 1, ma non esiste il collegamento inverso, da 1 a 0, in quanto il bit [1][0] (riga 1 e colonna 0) è a 0. Diventa quindi intuitivo capire che, se nel grafo non sono presenti *cappi* (quelli che si hanno quando un nodo punta a se stesso) la diagonale principale sarà composta da soli 0, proprio come nell'esempio.

Grafi non orientati

Quando il grafo non è orientato lo spazio in memoria occupato dalla matrice può essere di n^2 bit se vengono memorizzate tutte le celle della matrice come nel caso precedente, ma può essere inferiore se si nota che la matrice risultante sarebbe simmetrica rispetto alla diagonale principale, e quindi gli archi verrebbero memorizzati 2 volte: $m_{uv} = m_{vu}$.
Per poter capire di quanti bit abbiamo bisogno, dobbiamo prima sapere se i *cappi* sono ammessi o meno, e quindi se dobbiamo o no memorizzare anche la diagonale.
Con $\frac{n^2}{2}$ possiamo memorizzare uno dei "triangoli" sopra o sotto la diagonale, più metà diagonale.
Se abbiamo quindi bisogno di memorizzare anche la diagonale abbiamo bisogno di altri $\frac{n}{2}$ bit e raggiungiamo un totale di $\frac{n(n+1)}{2}$ bit.
Se invece non abbiamo bisogno della diagonale, non abbiamo bisogno nemmeno di quei bit

che ne memorizzavano metà, e li dobbiamo quindi sottrarre da $\frac{n^2}{2}$, ottenendo un totale di $\frac{n(n-1)}{2}$ bit necessari.



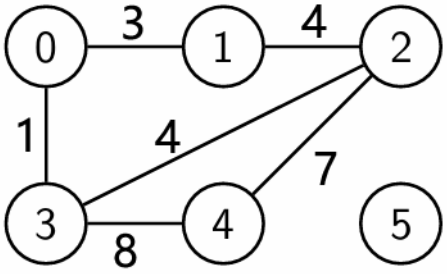
	0	1	2	3	4	5
0		1	0	1	0	0
1			1	0	0	0
2				1	0	0
3					1	0
4						0
5						

Grafi pesati

Quando abbiamo un grafo pesato andiamo a memorizzare nella matrice il peso p attribuito a quell'arco e non i precedenti 0/1 che indicavano solo l'esistenza o non esistenza dell'arco.
La definizione della matrice di adiacenza diventa quindi:

$$m_{uv} = \begin{cases} p & se\ (u,v) \in E \\ 0\ o\ \pm\infty & se\ (u,v) \notin E \end{cases}$$

Il valore associato ad un arco non esistente cambia in base al problema, l'importante è che non sia un valore ammesso per i pesi degli archi esistenti.



	0	1	2	3	4	5
0		3	0	1	0	0
1			4	0	0	0
2				4	7	0
3					8	0
4						0
5						

Lo spazio che occupano in memoria è quindi il numero di bit visti nei casi precedenti (che sarebbe il numero di celle che vado a memorizzare), moltiplicato per il numero di bit scelti per la memorizzazione dei pesi. Se quindi ho valori dei pesi ammessi solo da 0 a 15 userò 4 bit per ogni arco, e ogni cella della matrice varrà 4 bit, per questo moltiplico il numero delle celle per il numero dei bit. Prima invece ogni singola cella era da un solo bit, quindi non abbiamo moltiplicato per 1.

Complessità

Con l'implementazione tramite matrice verificare la presenza di un determinato arco richiede un tempo $O(1)$. Invece iterare su tutti gli archi richiede un tempo $O(n^2)$ sia per un grafo denso, che per uno sparso in quanto bisogna iterare in ogni caso su tutte le celle della matrice. Per questo l'utilizzo di un'implementazione tramite matrice è consigliata **solo in caso di grafi densi**. Inoltre un altro problema relativo a quest'implementazione è che, se si ha un numero elevato di nodi, n^2 può diventare enorme; si pensi che con soli 1000 nodi abbiamo 10^6 bit che sono circa 125KB e con 10.000 nodi arriviamo a 12,5MB.

1.3.2 Liste di adiacenza

Un altro modo per rappresentare un grafo, è tramite le **liste di adiacenza**.
L'idea alla base è quella di registrare solo i collegamenti che esistono davvero: per ogni nodo si mantiene una lista dei suoi vicini, invece di rappresentare tutte le possibili coppie di nodi come farebbe una matrice.

Lista di adiacenza

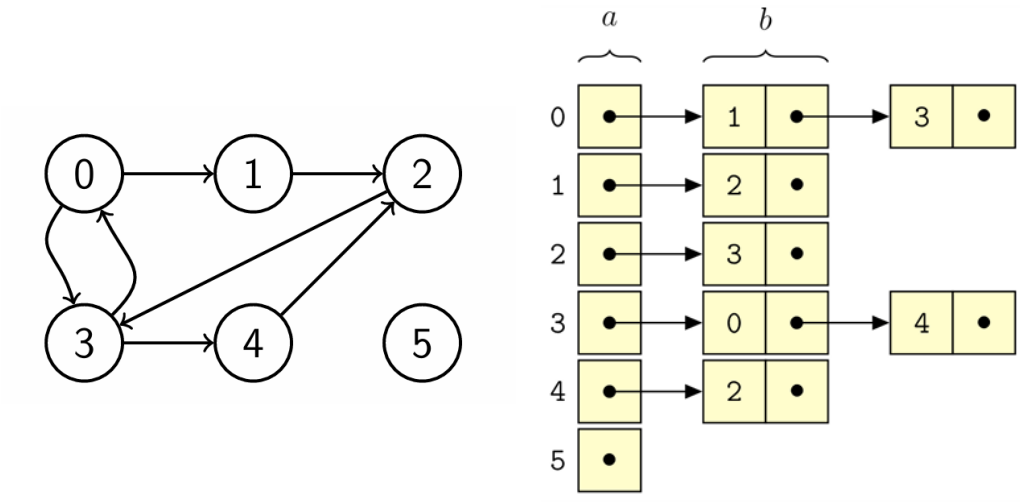
Una **lista di adiacenza** è un modo per rappresentare un grafo associando a ogni nodo l'elenco dei nodi a cui è collegato tramite un arco.
Si usa quindi un vettore con n celle, una per ciascun nodo, dove ogni cella conterrà il riferimento alla lista contenente tutti i nodi adiacenti al nodo in esame, quindi gli elementi $G.adj(u) = \{v \mid (u, v) \in E\}$.
Solitamente vengono realizzate con delle **linked list** dove ogni elemento punta al successivo, anche se questa scelta non è obbligata, infatti qualunque struttura che permetta di memorizzare i vicini del nodo è adatta.

In questo modo la struttura è compatta ed efficiente, soprattutto quando il grafo è **sparso**, e permette di accedere facilmente ai nodi adiacenti e di iterare sugli archi del grafo.

Grafi orientati

Cosa rappresenta $G.adj(u) = \{v \mid (u, v) \in E\}$ in questo caso?

$G.adj(u)$ rappresenta l'insieme dei nodi del grafo G adiacenti al nodo u. Cioè l'insieme dei nodi che vogliamo memorizzare per ogni nodo del grafo. La formula ci dice che questi nodi rispettano la caratteristica $\{v \mid (u, v) \in E\}$, cioè si trovano nell'insieme formato da tutti i nodi v tali che l'arco **da** u **a** v esista nell'insieme degli archi di G.

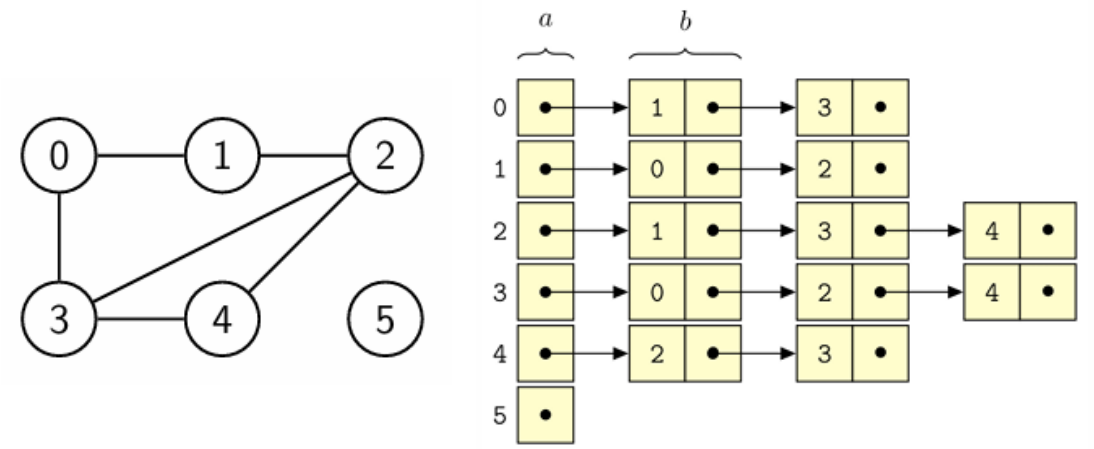


Lo spazio totale è dato da una parte fissa, un puntatore per ogni nodo ($a \times n$ bit, dove a è il numero di bit che occupa ogni cella, quindi ogni puntatore), e da una parte variabile, un elemento per ogni arco ($b \times m$ bit, dove b è il numero di bit che occupa ogni cella della lista, quindi bit del valore + bit del puntatore). Sommando otteniamo $an + bm$ bit occupati. Siccome a e b sono costanti, questo corrisponde a uno spazio $\Theta(n + m)$, cioè proporzionale al numero di nodi e di archi del grafo.

Grafi non orientati

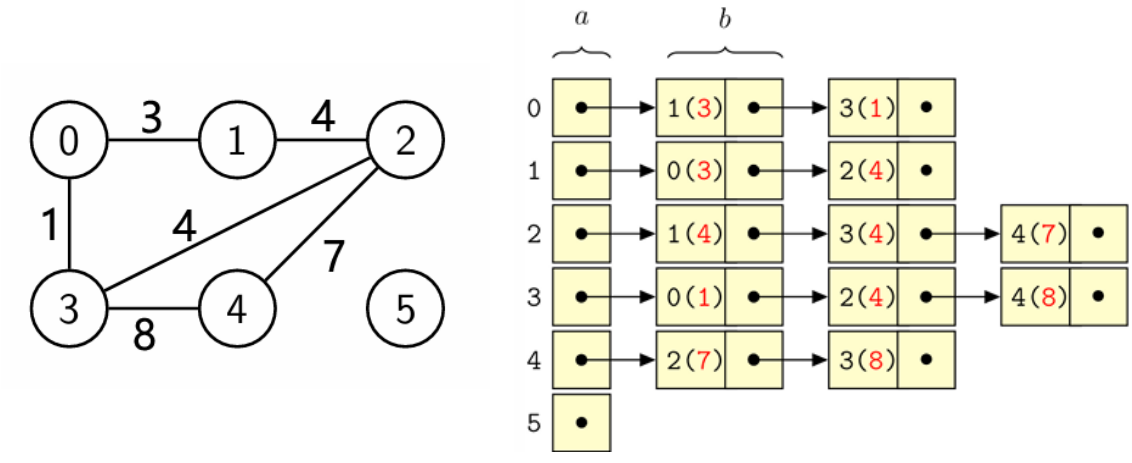
Cosa rappresenta $G.adj(u) = \{v \mid (u, v) \in E\}$ in questo caso?

Vale lo stesso ragionamento visto precendentemente, però cambia leggermente l'interpretazione di $\{v \mid (u, v) \in E\}$, in questo caso si riferisce all'insieme formato da tutti i nodi v tali che l'arco (u, v) esista nell'insieme degli archi di G.
Inoltre il fatto che il grafo non è orientato implica la scrittura doppia di ogni arco, una per ogni nodo coinvolto.



In questo caso lo spazio tortale è dato da $a\,n + 2 \cdot b\,m$ bit, il "2" è dato dal fatto che ogni arco viene memorizzato 2 volte. In ogni caso, siccome a b e 2 sono costanti, questo corrisponde a uno spazio $\Theta(n + m)$, cioè proporzionale al numero di nodi e di archi del grafo, proprio come nel caso precedente

Grafi pesati



In questo caso il grafo non è orientato e quindi valgono le stesse cose dette precedentemente. L'unica differenza è che dobbiamo memorizzare anche il peso di ogni arco, quindi la costante b coltre a contenere i bit necessari per nodo + puntatore, ha anche i bit necessari per memorizzare il peso. Quindi sempre uno spazio $\Theta(n + m)$.

Complessità

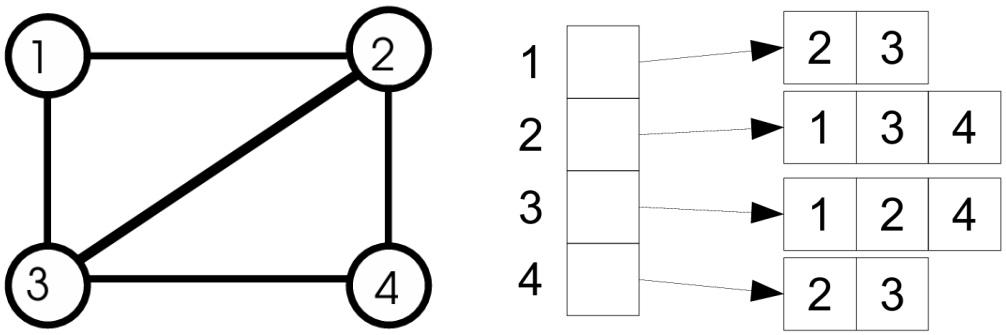
Come visto prima lo spazio in memoria richiesto è $\Theta(n)$ per il vettore con i nodi e $\Theta(|G.adj(u)|)$ per ciascuna lista con un totale di $\Theta(\, n + \sum_{u \in V} (|\, G.adj(u)\,|)\,) = \Theta(n + m)$, che è ottimo.
La verifica della presenza di un arco richiede tempo $O(n)$, poiché è necessario scorrere la lista dei vicini di u. Invece, l'iterazione su tutti gli archi del grafo richiede tempo $O(n + m)$, dato che si attraversano tutte le liste di adiacenza.
Queste complessità valgono allo stesso modo per grafi orientati, non orientati e pesati, poiché l'eventuale presenza di pesi non modifica la struttura delle liste ma solo il contenuto di ogni elemento. È ideale per i grafi **sparsi**.

1.3.3 Implementazione

Mentre l'implementazione di una matrice è fissa, esistono diversi modi per implementare una **lista di adiacenza**. La tabella seguente mostra le possibili strutture dati con cui è possibile implementarle:

Struttura	Java	C++	Python
Lista collegata	LinkedList	list	
Vettore statico*	[ ]	[ ]	[ ]
Vettore dinamico**	ArrayList	vector	list
Insieme	HashSet TreeSet	set	set
Dizionario	HashMapTreeMap	map	dict

In questo corso, se non diversamente specificato, si farà riferimento a un'implementazione basata su **vettori di adiacenza**.



Vettori di adiacenza

La rappresentazione tramite **vettori di adiacenza** utilizza un vettore, statico o dinamico, `adj` di dimensione n, indicizzato dai nodi del grafo (cioè ogni indice del vettore corrisponde al nodo con quell'identificatore).
Ogni elemento `adj[u]` contiene un vettore di adiacenza (il riferimento al suo primo elemento), che memorizza tutti i nodi v adiacenti a u.

Inoltre assumeremo che:

- La classe `Node` sia uguale a `int` e che di conseguenza l'accesso alle informazioni abbia costo $O(1)$;
- Le operazioni per aggiungere nodi e archi abbiano costo $O(1)$ (per i vettori dinamici, anche se il costo dell'`append` non vale sempre $O(1)$, si fa riferimento all'analisi ammortizzata, dove il costo rimane $O(1)$);
- Dopo l'inizializzazione, il grafo sia **statico**, quindi una volta creato non cambia più (ad esempio viene mandato in input come file di testo).

Riassumendo

Per semplicità, assumiamo che il grafo sia rappresentato con liste di adiacenza basate su vettori, che i nodi siano semplici interi, che accedere e inserire elementi costi $O(1)$, e che il grafo non cambi mentre eseguiamo gli algoritmi.

1.3.4 Iterazione su nodi e archi

Precedentemente abbiamo discusso delle complessità per matrici e liste di adiacenza. Ora risulta quindi utile formalizzare lo schema di iterazione tipico utilizzato negli algoritmi sui grafi.

L'iterazione su tutti i nodi del grafo, rappresentata dallo pseudocodice seguente, ha costo $O(n)$, moltiplicato per il costo dell'operazione eseguita su ciascun nodo. Se l'operazione all'interno del ciclo ha costo c allora avremo un costo $O(n \cdot c)$; bisogna tenere a mente però che, se tale costo è costante per ogni nodo ($c = O(1)$), esso può essere trascurato e si ottiene nuovamente $O(n)$.

```
foreach u ∈ G.V() do
    /* Esegui operazioni sul nodo u */
end for
```

Se ho invece bisogno di iterare su tutti i nodi e tutti gli archi del grafo, operazione rappresentata dal seguente pseudocodice, avrò un costo computazionale $O(n+m)$ se il grafo è implementato tramite liste di adiacenza, e $O(n^2)$ se rappresentato tramite matrici.

```
foreach u ∈ G.V() do
    /* Esegui operazioni sul nodo u */
    foreach v ∈ G.adj(u) do
        /* Esegui operazioni sull'arco (u, v) */
    end for
end for
```

1.4 Visite

Visite: definizione generale

Una *visita* di un grafo è una procedura sistematica per esplorare un grafo, visitando almeno una volta ogni suo nodo ed arco.

Noi andremo però ad esaminare un problema più specico che presenta due famiglie principali di soluzione.

Caso di studio

Dato un grafo $G(V, E)$ e un vertice (radice o sorgente) $r \in V$, visitare una ed una sola volta tutti i nodi del grafo che possono essere raggiunti da r

Per "risolvere" il problema abbiamo due strade percorribili:
- **Visita in ampiezza - BFS** (Breadth-First Search);
- **Visita in profondità - DFS** (Depth-first Search);

che verranno aprofondite nel corso di questo capitolo.

Intanto rivediamo l'idea generale dietro queste famiglie di algoritmi che abbiamo già visto parlando degli alberi:

BFS

In questo genere di algoritmi la visita dei nodi avviene per livelli: prima si visita la radice, poi i nodi a distanza 1 dalla radice, poi quelli a distanza 2 ecc.
Una possibile applicazione è quella del calcolo dei cammini più brevi a partire da una singola sorgente.

DFS

In questo tipo di algoritmi la visita dei nodi avviene in profondità: a partire da un nodo, si visita uno dei suoi nodi adiacenti e si continua l'esplorazione lungo un cammino finché possibile, prima di tornare indietro.
È tipicamente implementata in modo ricorsivo ed è utilizzata, tra le altre cose, per l'ordinamento topologico e per il calcolo delle componenti connesse e fortemente connesse.

1.4.1 Visita: più complessa di quanto sembri

Per visitare un grafo si potrebbe pensare di ciclare per ogni nodo e per ogni arco, come nello pseudocodice seguente. Così facendo però non si tiene in considerazione la struttura del grafo e si itera su tutte le possibili coppie di nodi, non solo le coppie adiacenti. L'ieraione avviene quindi senza un criterio preciso.

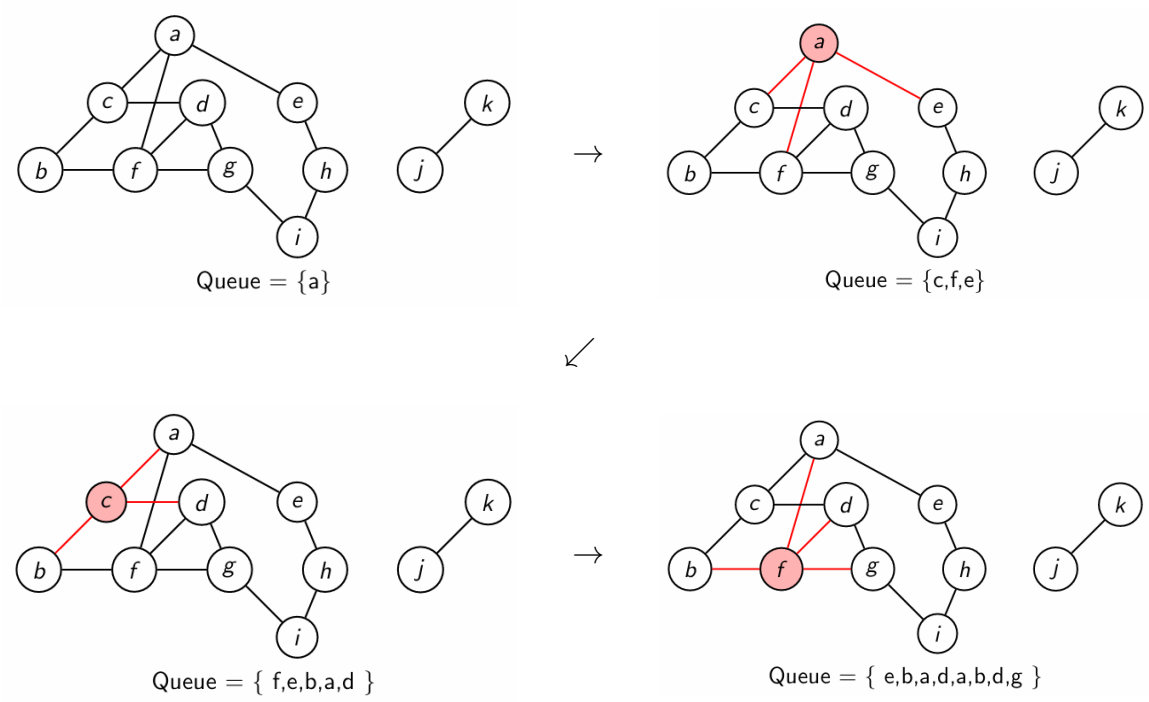
Visit(Graph G)

```
foreach u ∈ G.V() do
    /* Visita il nodo u */
    foreach v ∈ G.V() do
        /* Visita l'arco (u, v) */
        // anche se, come abbiamo detto prima, la coppia (u, v) non corrisponde
        //necessariamente ad un arco; non è detto che u e v siano adiacenti
    end for
end for
```

Si potrebbe quindi pensare di visitare il grafo trattandolo come un albero, visitando quindi i nodi vicini al nodo in esame. Questo approccio è trattato dal codice seguente dove r è il nodo di partenza scelto come radice e Q è la coda in cui andremo a memorizzare i nodi vicini

```
BFSTraversal(Graph G, int r)
  QUEUE Q = Queue()
  Q.enqueue(r)    // inserire r in fondo alla coda
  while not Q.isEmpty() do
      Node u = Q.dequeue()    // prendere il primo elemento in testa alla coda
      /* visita il nodo u */
      foreach v ∈ G.adj(u) do
          Q.enqueue(v)    // inserire i vicini di u in fondo alla coda
      end for
  end while
```

Questa soluzione potrebbe sembrare corretta a primo impatto ma non tiene conto dei **nodi già visitati**. Quindi, se ad esempio avessimo un grafo che contiene un ciclo, la coda crescerebbe in maniera sconsiderata e continueremmo a **visitare i nodi all'infinito**.
Vediamo un esempio passo passo:



Andando avanti il prossimo nodo esplorato sarebbe e, e si aggiungerebbero alla coda i nodi a e h. Già da qui notiamo che molti nodi si ripetono nella coda, e ogni volta che vengono rivisitati aggiungono i propri vicini alla coda, che riaggiungeranno a loro volta anche il nodo che li ha "chiamati", andando avanti in un loop infinito.

1.4.2 Algoritmo generico di attraversamento

Per risolvere i problemi di terminazione e ridondanza visti in precedenza, si introduce un algoritmo di attraversamento generico. La novità cruciale è l'uso di un meccanismo di marcatura (riga 3 e 10): un nodo viene aggiunto alla struttura di supporto solo se non è stato ancora visitato, garantendo che ogni vertice venga elaborato una sola volta.

visit(GRAPH G)

```
1:  SET S = Set()    // Insieme generico
2:  S.insert(r)    // Da specificare
3:  /* marca il nodo r */
4:  while S.size() > 0 do
5:      NODE u = S.remove()    // Da specificare
6:      /* visita il nodo u */
7:      foreach v ∈ G.adj(u) do
8:          /* visita l'arco (u, v) */
9:          if v non è ancora stato marcato then
10:             /* marca il nodo v */
11:             S.insert(v)    // Da specificare
12:         end if
13:     end for
14: end while
```

Questo algoritmo è un algoritmo generico in quanto abbiamo un insieme generico S per memorizzare i nodi, e non sono state specificate le modalità di inserimento e rimozione di un nodo dal suddetto insieme. Vediamo quindi quando questo si comporta come un algoritmo **BFS** e quando come uno **DFS**:

BFS vs DFS

- **BFS**: Quando l'insieme S è una **coda**, la logica di inserimento e rimozione è di tipo **FIFO** (*First In First Out*), la visita diventa una **visita in ampiezza**, perché andiamo a prendere il nodo che si trova nella coda da più tempo, e visitiamo quindi la coda per livelli.
- **DFS**: Quando l'insieme S è una **pila**, la logica di inserimento e rimozione è di tipo **LIFO** (*Last In First Out*), la visita diventa una **visita in profondità**, perché andiamo a prendere l'ultimo nodo inserito e ci addentriamo quindi il più possibile nel grafo prima di tornare indietro.

1.5 BFS - Breadth-First Search

Abbiamo detto quindi che gli algoritmi **BFS** effettuano una **visita in ampiezza** del grafo, per memorizzare i nodi utilizzano una **coda** e di conseguenza la logica di inserimento e rimozione è di tipo **FIFO**. Vediamo adesso quali sono gli obbiettivi di questi algoritmi:

Obbiettivi

- Visitare i nodi a distanze crescenti $\rightarrow$ visitare tutti i nodi a distanza k dalla sorgente prima di quelli a distanza $k+1$

- Calcolare il cammino più breve da r a tutti gli altri nodi → Le distanze sono misurate come il numero di archi attraversati
- Generare un **albero breadth-first** → Generare un albero contenente tutti i nodi raggiungibili da r, tale per cui il cammino dalla radice r al nodo u nell'albero corrisponda al cammino più breve da r a u nel grafo.

1.5.1 Algoritmo BFS

```
bfs(GRAPH G, NODE r)
 1: QUEUE Q = Queue()    // Coda perché è un BFS
 2: Q.enqueue(r)
 3: boolean[] visited = new boolean[G.size()]    // Vettore per marcare i nodi
 4: foreach u ∈ G.V() − {r} do
 5:     visited[u] = false
 6: end for
 7: visited[r] = true
 8: while not Q.isEmpty() do
 9:     NODE u = Q.dequeue()
10:     /* visita il nodo u */
11:     foreach v ∈ G.adj(u) do
12:         /* visita l'arco (u, v) */
13:         if not visited[v] then
14:             visited[v] = true
15:             Q.enqueue(v)
16:         end if
17:     end for
18: end while
```

Come abbiamo già ripetuto varie volte, e come si può vedere dallo pseudocodice, questo algoritmo utilizza una **coda** (FIFO) come struttura dati di supporto e questo garantisce la visita a livelli (i nodi più vicini alla radice vengono aggiunti alla coda e visitati prima di quelli più lontani). Il processo si divide in tre parti principali:

- Si marca la sorgente come visitata e la si inserisce in coda;
- Si estrae un nodo dalla coda e si esplorano tutti i suoi vicini non ancora marcati;
- Ogni vicino scoperto viene marcato e aggiunto in fondo alla coda per essere processato successivamente.

Questo metodo garantisce che un nodo venga scoperto solo attraverso il cammino minimo possibile (in termini di numero di archi). Se esistesse un cammino più breve per raggiungere v, la BFS lo avrebbe già incontrato in un livello precedente.

1.5.2 Numero di Erdős

La storia della matematica è piena di personaggi eccentrici; Paul Erdős è uno di questi. Fu un matematico estremamente prolifico, che arrivò a pubblicare circa 1500 articoli, con più di 500 collaboratori diversi.

Come tributo scherzoso alla sua estrema prolificità, è nato il concetto di “numero di Erdős”, un coefficiente assegnato ad ogni autore scientifico per misurare la sua "distanza" da Erdős (`https://en.wikipedia.org/wiki/Erdos_number`).

Ragionamento dietro il numero di Erdős

- Erdős ha `distance = 0` o `erdos = 0` *
- I co-autori dei suoi articoli hanno `distance = 1`
- Chi è co-autore di un co-autore di Erdős e non è direttamente coautore di Erdős, avrà `distance = 2`
- Chi è co-autore di qualcuno con `distance = k` e non è direttamente coautore di Erdős, avrà `distance = k + 1`
- Tutti gli altri avranno `distance` = ∞

*Il professore chiama questa variabile `erdos` mentre il libro la chiama `distance`

Possiamo quindi riscrivere l’algoritmo sfruttando queste definizioni e otteniamo :

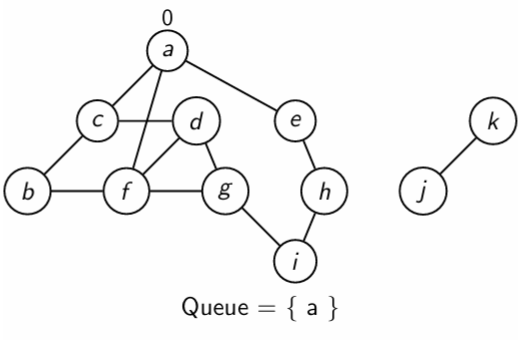
distance(GRAPH G, NODE r, **int**[] $distance$)

```
1:  QUEUE Q = Queue()
2:  Q.enqueue(r)
3:  boolean[] visited = new boolean[G.size()]    // Il prof la mette ma in questo caso è inutile
4:  foreach u ∈ G.V() − {r} do
5:      distance[u] = ∞
6:  end for
7:  distance[r] = 0
8:  while not Q.isEmpty() do
9:      NODE u = Q.dequeue()
10:     foreach v ∈ G.adj(u) do
11:         if distance[v] == ∞ then    // Se il nodo v non è stato scoperto
12:             distance[v] = distance[u] + 1
13:             Q.enqueue(v)
14:         end if
15:     end for
16: end while
```

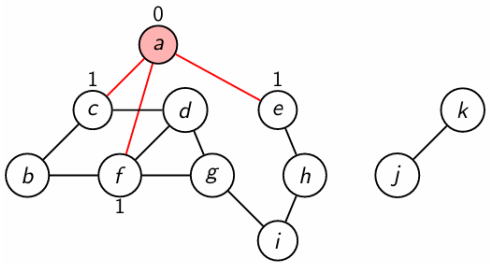
Come si può evincere dal commento, la riga 3, nonostante il professore la inserisca nello speudocodice, è completamente inutile in questo contesto perché `visited` non viene mai usato in questo codice in quanto, per verificare se il nodo è già stato scoperto, si controlla la sua distanza.
Se un nodo ha distanza infinita, non è mai stato inserito in coda; non appena viene scoperto, riceve un valore numerico finito ($distance[v] = distance[u] + 1$) e non verrà mai più processato. Questo garantisce che ogni nodo venga aggiornato solo la prima volta che viene raggiunto, ovvero lungo il cammino minimo dalla sorgente.

Esempio

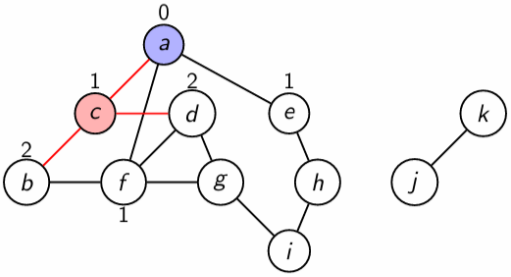
Vediamo ora un esempio visivo di come vengono calcolate le distanze con questo genere di algoritmi:



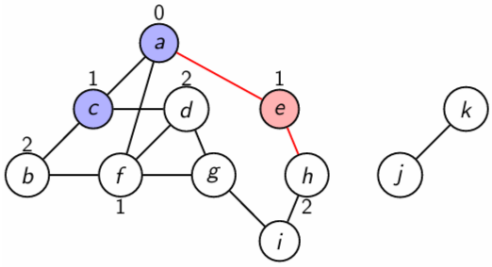
Queue = { a }



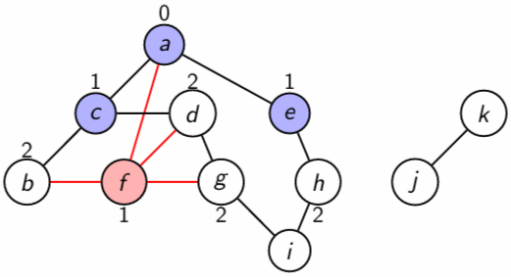
Queue = { c, e, f }



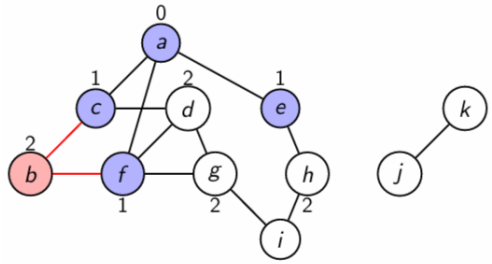
Queue = { e, f, b, d }



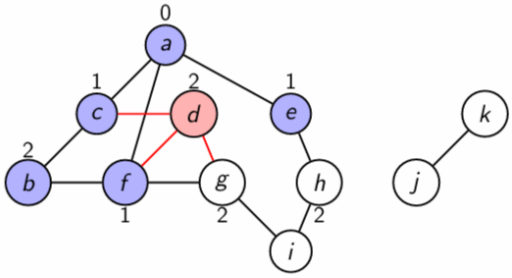
Queue = { f, b, d, h }



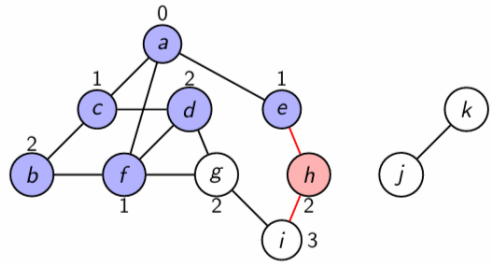
Queue = { b, d, h, g }



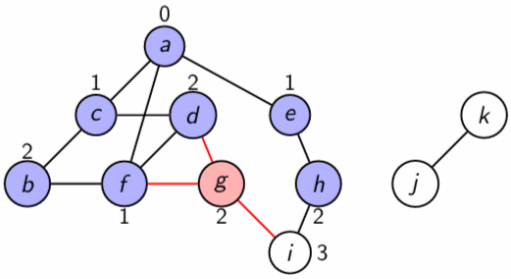
Queue = { d, h, g }



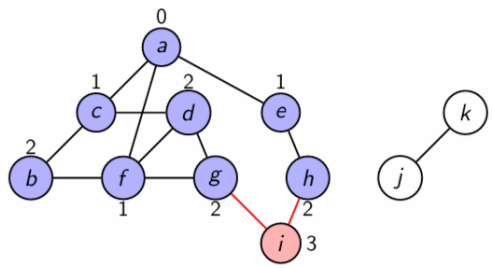
Queue = { h, g }



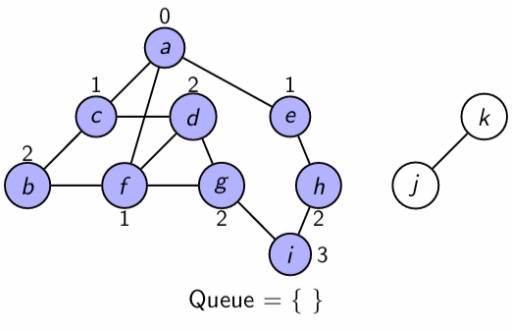
Queue = { g, i }



Queue = { i }



Queue = { i }



1.5.3 Albero BFS

Albero BFS

L'**Albero BFS** è un albero di copertura radicato in r che viene costruito dinamicamente durante la visita in ampiezza, utilizzando un vettore dei padri per tracciare il percorso più breve dalla sorgente a ogni nodo.

Infatti, oltre a visitare i nodi di un grafo, l'algoritmo BFS permette di definire una struttura ad albero, detta **Albero BFS**, che memorizza i cammini minimi (in termini di numero di archi) partendo da una radice r verso tutti i nodi raggiungibili.
Ricordiamo che, in un grafo non pesato, il **cammino minimo** coincide sempre con il cammino che attraversa il **minor numero di archi**.

Passo 1: Memorizzazione e Costruzione

distance(GRAPH G, NODE r, **int**[] $distance$, NODE[] $parent$)

```
1:  QUEUE Q = Queue()
2:  Q.enqueue(r)
3:
4:  foreach u ∈ G.V() − {r} do
5:      distance[u] = ∞    // Si inizializzano tutte le distanze a ∞
6:  end for
7:
8:  distance[r] = 0    // Si mette la distanza della radice pari a 0
9:  parent[r] = nil    // Si mette il padre della radice come null
10:
11: while not Q.isEmpty() do
12:     NODE u = Q.dequeue()
13:     foreach v ∈ G.adj(u) do
14:         if distance[v] == ∞ then    // Se il nodo v non è stato scoperto
15:             distance[v] = distance[u] + 1
16:             parent[v] = u
17:             Q.enqueue(v)
18:         end if
19:     end for
20: end while
```

Questo codice è come quello visto in precedenza, al quale però sono state aggiunte le parti scritte in rosso. VEdiamo la vogica che c'è dietro:

L'albero viene memorizzato in un vettore `parent`, dove ogni cella `parent[v]` contiene il nodo u che ha scoperto v. Durante l'esecuzione della BFS, quando esploriamo un nodo u e troviamo un vicino v non ancora visitato (`distance[v] = ∞`):

- Si aggiorna la distanza: `distance[v] = distance[u] + 1;`
- Si imposta il padre: `parent[v] = u`.

Passo 2: Ricostruzione del cammino

Per visualizzare il percorso minimo dalla radice `r` a un nodo destinazione `s`, si utilizza la procedura ricorsiva `printPath`. Questa risale il vettore dei padri fino alla radice e stampa i nodi durante il ritorno della ricorsione, garantendo l'ordine corretto.

printPath(NODE r, NODE s, NODE[] $parent$)

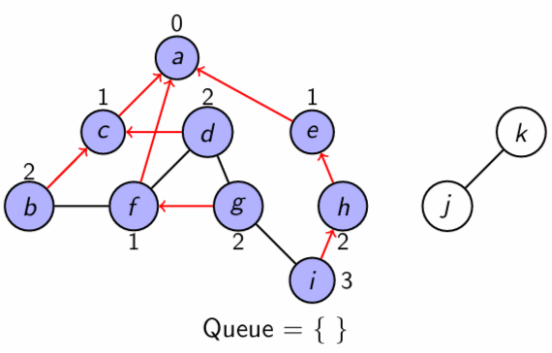
```
1: if r == s then    // Caso base: siamo arrivati alla radice
2:     print s
3: else if parent[s] == nil then    // Il nodo s non è raggiungibile da r
4:     print "error"
5: else   // Risalita ricorsiva verso la radice
6:     printPath(r, parent[s], parent)
7:     print s    // Stampa il nodo durante il ritorno della ricorsione
8: end if
```

L'ordine delle istruzioni nel blocco else è fondamentale. Chiamando la funzione ricorsivamente su `parent[s]` prima di stampare `s`, ci assicuriamo che la stampa avvenga in ordine "top-down" (dalla radice alla destinazione), sfruttando lo stack delle chiamate della ricorsione per invertire l'ordine di risalita dei padri.

Appunto di Gemini sulla complessità spaziale

- Oltre alla memoria per il grafo, l'albero BFS richiede uno spazio aggiuntivo di $O(V)$ per memorizzare il vettore parent;
- La funzione printPath richiede uno spazio di stack ricorsivo proporzionale alla profondità dell'albero (nel caso peggiore $O(V)$).

Se ci riferiamo al grafo dell'esempio precedente, una volta terminata la visita avremo una figura come la seguente, nella quale gli archi rossi rappresentano il vettore dei padri:



1.5.4 Complessità BFS

(Ammetto di aver copiato e incollato da Gemini ciò che segue)

L'algoritmo BFS è estremamente efficiente, con una complessità temporale pari a $O(n + m)$. Questo risultato lo rende un algoritmo lineare rispetto alla dimensione della rappresentazione del grafo (nodi + archi).
Il motivo di questa efficienza risiede nella gestione oculata delle risorse durante la visita:

- **Sui nodi (n)**: Grazie all'uso della coda e al controllo della distanza (o del colore), ogni nodo viene inserito e rimosso dalla coda una sola volta. Non ci sono quindi ricalcoli inutili sullo stesso vertice.
- **Sugli archi (m)**: Ogni volta che un nodo viene estratto dalla coda, l'algoritmo esamina i suoi vicini. Questo significa che ogni arco viene analizzato una sola volta (nel caso di grafi orientati) o al massimo due (in quelli non orientati).
  Tecnicamente, il numero di archi analizzati corrisponde alla somma dei gradi uscenti (d_{out}) di tutti i nodi.
  $$m = \sum_{u \in V} d_{out}(u)$$

In sintesi, la BFS impiega un tempo proporzionale alla "grandezza" del grafo, rendendola una scelta ottimale per trovare cammini minimi in strutture dati di grandi dimensioni, a patto di utilizzare una lista di adiacenza per l'esplorazione dei vicini.

1.6 DFS - Depth-First Search

1.6.1 BFS vs DFS

1.6.2 Iterativa, stack esplicito, pre-order

1.6.3 DFS e componenti connesse

1.6.4 DFS e grafi non orientati aciclici

1.6.5 DFS e grafi orientati aciclici (DAG)

1.6.6 Schema

1.6.7 Classificazione degli archi

1.7 Ordinamento topologico