9) Grafi

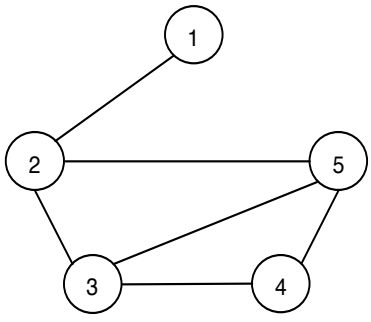
In questo capitolo vengono illustrati i ***grafi***, importantissime strutture discrete che da un lato esibiscono proprietà di grande interesse per la matematica (in particolare per la teoria dei grafi) e, dall'altro, permettono di modellare una grande varietà di problemi.

9.1 Definizioni e semplici proprietà

Un ***grafo*** *G = (V, E)* è costituito da una coppia di insiemi, l'insieme finito *V* di ***nodi***, o ***vertici***, e l'insieme finito *E* di coppie non ordinate di nodi, dette ***archi*** o ***spigoli***. Laddove non sorgano equivoci, gli archi vengono generalmente rappresentati come coppie racchiuse tra parentesi tonde – ad es. *(u, v)* - sebbene, essendo coppie non ordinate, sarebbe più corretto racchiuderle tra parentesi graffe – ad es. *{u, v}*. Si dice che un arco è **incidente** ai suoi due estremi.

Usualmente, gli interi *n* ed *m* indicano rispettivamente il numero di nodi e di archi di cui è composto il grafo.

E' facile vedere che il massimo numero di archi è *n(n - 1)/2*. Nel caso in cui *m = O(n)* diremo che il **grafo** è **sparso**.



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

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

Un arco *(u, v)* con *u = v* è detto ***cappio***. Un arco che compare più di una volta in *E* è detto ***arco multiplo***. Un ***grafo semplice*** non contiene cappi né archi multipli.

Un ***grafo orientato*** (detto anche ***digrafo***) è definito in modo analogo al grafo, ma gli archi sono delle coppie ordinate e sono detti ***archi orientati***.

Un grafo è detto ***pesato*** se ad ogni arco è associato un valore numerico detto ***peso***.

In questa brevissima trattazione, ometteremo di trattare grafi con cappi e archi multipli, grafi orientati e grafi pesati, e quindi si userà la sola parola *grafo* per intendere la locuzione *grafo semplice non orientato e non pesato*.

Il numero di archi incidenti in un certo nodo *v* è detto ***grado*** *di v (**degree(v)**)*.

Teorema: *Dato un grafo non orientato G, la somma dei gradi di tutti i nodi è pari a 2m. Dato un grafo orientato G, la somma dei gradi entranti di tutti i nodi è pari alla somma dei gradi uscenti di tutti i nodi, che è pari ad m.*

Dimostrazione. Contiamo in due modi diversi le coppie *(nodo, arco)* che sono in relazione di incidenza, cioè contiamo in due modi le coppie dell'insieme

$$A=\{(x,e): x\in V, e\in E, x\in e\}.$$

Fissato un nodo $x\in V$, il numero di archi ad esso incidenti è dato dal suo grado. Quindi: $|A|=\sum_{x\in V}\deg ree(x)$.

Per ogni arco $e\in E$, esistono esattamente due nodi ad esso incidenti.

Quindi: $|A|=2|E|$.

Unendo le due uguaglianze, si ottiene la tesi.

Il caso orientato si dimostra analogamente. **CVD**

Corollario (detto *Regola della stretta di mano***)**. *Sia G=(V,E) un grafo finito. Il numero dei nodi di G che hanno grado dispari è pari.*

Dimostrazione. $\sum_{x\in V}\deg ree(x)=\sum_{x\in V,\ \deg ree(x)\ pari}\deg ree(x)+\sum_{x\in V,\ \deg ree(x)\ dispari}\deg ree(x)$. Ma per il teorema precedente si ha: $\sum_{x\in V}\deg ree(x)=2|E|$

che è un numero pari. Poiché la somma di un numero pari di dispari è pari, mentre la somma di un numero dispari di dispari è dispari, *G* deve avere un numero pari di nodi di grado dispari. **CVD**

Una ***passeggiata*** in *G* dal nodo *x* al nodo *y* è una sequenza $x_1, e_1, x_2, e_2, \ldots, x_t, e_t, x_{t+1}$ in cui si alternano nodi ed archi del grafo con $x_i \in V$, $e_i \in E$, $x = x_1$, $y = x_{t+1}$ e tale che per $i = 1, 2, \ldots, t$ si abbia $\{x_i, x_{i+1}\} = e_i$. La ***lunghezza della passeggiata*** è *t*, ovvero il numero dei suoi archi (incluse le eventuali ripetizioni).

Un ***cammino*** da *x* a *y* è una passeggiata da *x* a *y* in cui non ci siano ripetizioni di archi: se $i \neq j$ allora $e_i \neq e_j$.

Un cammino in cui primo e ultimo nodo coincidono si chiama ***circuito***.

Un ***cammino semplice*** è un cammino in cui non ci siano ripetizioni di nodi:

se $i \neq j$ allora $x_i \neq x_j$, ad eccezione al più di *1* e $t + 1$; nel caso in cui $x_1 = x_{t+1}$ il cammino semplice si chiama ***ciclo***.

Proposizione. *Se in G esiste una passeggiata da x a y allora esiste un cammino semplice da x a y.*

Dimostrazione. Se nella passeggiata $x = x_1, e_1, x_2, e_2, \ldots, x_t, e_t, x_{t+1} = y$ esistono indici $i \neq j$ tali che $x_i = x_j = z$, allora sia i' il più piccolo indice per cui $z = x_{i'}$, e sia j' l'indice più grande per cui $z = x_{j'}$. Sostituendo la sezione della passeggiata $x_{i'}, e_{i'}, x_{i'+1}, \ldots, x_{j'}$ con il solo nodo *z* si ottiene una nuova passeggiata, visto che in questa nuova sequenza ogni arco presente ha gli stessi vicini di prima. D'altronde in questa passeggiata il nodo *z* non è più ripetuto. Iterando questa procedura con gli altri nodi ripetuti presenti, si arriva ad un cammino semplice. **CVD**

Si dice che *y* è ***raggiungibile*** da *x* se $y = x$ oppure se esiste una passeggiata da *x* a *y*.

- La relazione di raggiungibilità tra nodi ha la proprietà ***riflessiva*** per definizione perché *x* è raggiungibile da *x*.

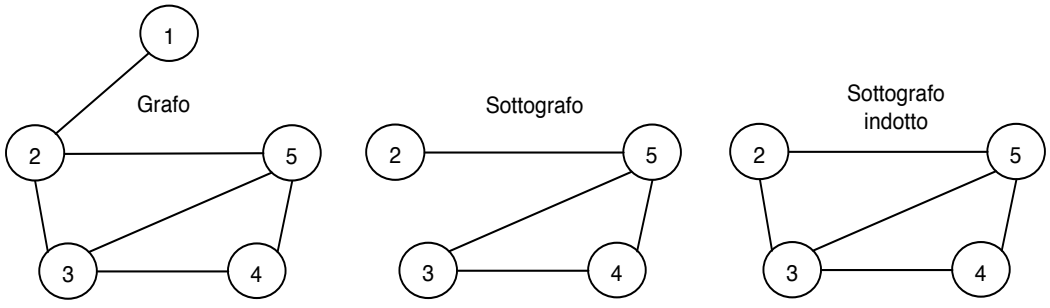
Pag 157

- La relazione è ***simmetrica*** perché se $x_1, e_1, x_2, e_2, \dots, x_t, e_t, x_{t+1}$ è una passeggiata da x ad y, allora $x_{t+1}, e_t, x_t, \dots, x_2, e_1, x_1$ è una passeggiata da y ad x.
- Inoltre la relazione di raggiungibilità è anche ***transitiva***: se c'è una passeggiata da x a y e una passeggiata da y a z, allora z è raggiungibile da x attraverso la passeggiata che si ottiene concatenando la passeggiata da x a y con quella da y a z.

La relazione di raggiungibilità è perciò una relazione di equivalenza: le classi di questa relazione di equivalenza determinano una partizione dei nodi. Tutti i nodi di una stessa classe sono raggiungibili da ogni altro nodo della classe. Ogni classe della partizione induce un sottografo chiamato ***componente connessa***. Un grafo in cui la partizione in questione ha una sola classe si chiama ***connesso***.

Non è restrittivo supporre che i grafi considerati siano connessi: se così non fosse, si potrebbero considerare separatamente, una per una, tutte le loro componenti connesse.

Dato un grafo $G = (V, E)$, un **sottografo** di G è un grafo $G' = (V', E')$ tale che $V \subseteq V'$ ed $E \subseteq E'$. Dato un sottinsieme V' di V, si definisce **sottografo indotto** da V' il sottografo di G che ha come insieme dei nodi l'insieme V' e come insieme degli archi l'insieme di ***tutti*** gli archi di G che collegano nodi in V'. Un **sottografo** $G' = (V', E')$ è **ricoprente** per G se $V' = V$.



Pag 158

9.2 Rappresentazione in memoria di grafi

Vi sono diverse tecniche per rappresentare un grafo in memoria; tra esse:

- liste di adiacenza;
- matrice di adiacenza;
- matrice di incidenza;
- lista di archi.

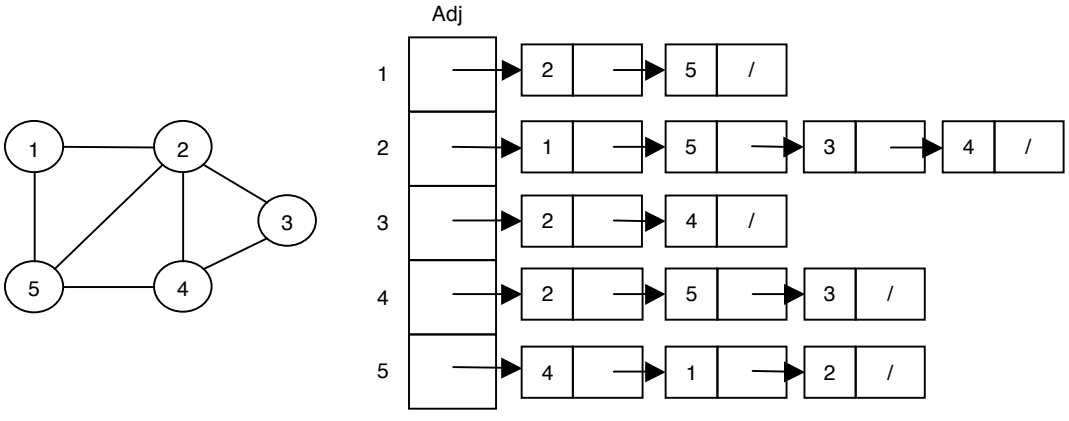
Le prime due tecniche sono quelle più spesso utilizzate in pratica.

Come vedremo nel seguito, e come abbiamo già visto nel caso della memorizzazione di alberi, non c'è un modo per memorizzare grafi che risulti migliore in assoluto: è la natura del problema da risolvere che determina la scelta della tecnica di rappresentazione del grafo più idonea alla soluzione.

9.2.1 Liste di adiacenza

Questa memorizzazione consiste in un vettore *Adj[]* di *|V|* puntatori, uno per ciascun nodo in *V,* che rappresentano la testa di *|V|* liste.

Per ciascun nodo $v \in V$, *Adj[v]* punta ad una lista i cui elementi contengono gli indici di tutti i nodi *u* tali che esiste un arco *(u, v)* in *G*. In altre parole, *Adj[v]* punta ad una lista che contiene gli indici di tutti i nodi adiacenti a *v* in *G*.



Si noti che un arco *(u, v)* in un grafo non orientato appare due volte:

- esiste l'indice *u* nella lista puntata da *Adj[v]*;
- esiste l'indice *v* nella lista puntata da *Adj[u]*.

Viceversa, nel caso dei grafi orientati, l'arco *(u, v)* uscente da *u* ed entrante in *v* appare una sola volta:

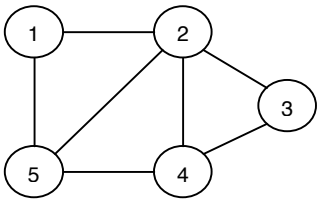
- esiste l'indice *v* nella lista puntata da *Adj[u]*.

Per rappresentare un grafo pesato è sufficiente aggiungere ad ogni elemento delle liste un campo che contiene il peso dell'arco.

9.2.2 Matrice di adiacenza

Questa rappresentazione consiste in una matrice *A* di n^2 numeri interi, nella quale:

- *A[i, j] = 1* se e solo se l'arco *(i, j)* $\in$ *E(G)*;
- *A[i, j] = 0* altrimenti



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

Si noti la simmetria rispetto alla diagonale principale. Essa è dovuta al fatto che, essendo il grafo non orientato, l'arco *(u, v)* e l'arco *(v, u)* coincidono.

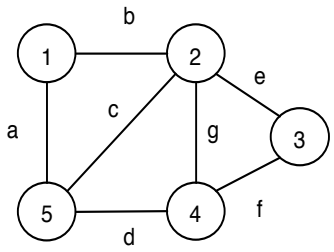
Nel caso di grafo orientato, la definizione è identica (ma questa volta la coppia che rappresenta l'arco è ordinata!), e quindi la matrice non risulta più simmetrica.

Per rappresentare un grafo pesato è sufficiente memorizzare nella matrice i pesi degli archi anziché il valore 1.

9.2.3 Matrice di incidenza

Questa rappresentazione consiste in una matrice *A* di *n* righe (una per ogni nodo) ed *m* colonne (una per ogni arco), nella quale in ogni colonna vi è un 1 in

corrispondenza delle righe relative ai due nodi in cui l'arco in questione è incidente.



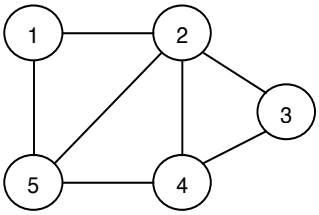
	a	b	c	d	e	f	g
1	1	1					
2		1	1		1		1
3					1	1	
4				1		1	1
5	1		1	1			

Per rappresentare un grafo pesato è sufficiente memorizzare nella matrice i pesi degli archi anziché il valore 1. Per memorizzare un grafo orientato, basta rappresentare in modo differente il nodo di partenza e quello di destinazione di ciascun arco (ad esempio con -1 e 1, rispettivamente).

9.2.4 Lista di archi

Questa rappresentazione consiste semplicemente in un vettore che contiene tutti gli archi del grafo.

Ogni elemento del vettore rappresenta un arco *(u, v)* mediante la coppia degli indici dei nodi *u* e *v* in cui esso è incidente.



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

Per rappresentare un grafo pesato è sufficiente aggiungere ad ogni elemento del vettore un campo che contiene il peso dell'arco. Un grafo orientato verrà

rappresentato allo stesso modo, ma gli archi andranno considerati come orientati dal primo al secondo indice.

9.2.5 Confronti fra le rappresentazioni

In questo paragrafo confrontiamo le varie rappresentazioni di grafi dal punto di vista dell'occupazione di memoria e del costo computazionale di due semplici operazioni.

Occupazione di memoria	
Liste di adiacenza	$\Theta(n + m)$
Matrice di adiacenza	$\Theta(n^2)$
Matrice di incidenza	$\Theta(nm)$
Lista di archi	$\Theta(m)$

Dato il nodo *v*, calcolare *degree(v)*	
Liste di adiacenza	$\Theta(degree(v))$ Si scorre la lista puntata da Adj(v)
Matrice di adiacenza	$\Theta(n)$ Si scorre la v-esima riga della matrice
Matrice di incidenza	$\Theta(m)$ Si scorre la v-esima riga della matrice
Lista di archi	$\Theta(m)$ Si scorre l'intera lista di archi

Dati i nodi *u* e *v*, l'arco (u, v) ∈ E?	
Liste di adiacenza	$O(degree(u))$ Si scorre al più l'intera lista puntata da *Adj(u)*
Matrice di adiacenza	$\Theta(1)$ Si ispeziona l'elemento *M[u,v]* della matrice
Matrice di incidenza	$O(m)$ Si scorre la *u*-esima riga della matrice; quando si trova un 1 si verifica se nella nella riga *v*-esima della stessa colonna è posto un 1
Lista di archi	$O(m)$ Si scorre al più l'intera lista di archi

Pag 162

Si noti che, in generale, un costo di *Θ(degree(v))* è molto più favorevole rispetto sia a *Θ(n)* che a *Θ(m)*, anche perché di solito lo stesso costo va conteggiato su tutti i nodi, e la somma dei gradi di tutti i nodi di un grafo non orientato:

$$\sum_{i=1}^{n} degree(v_i)$$

è pari a *2m*.

9.3 Visita di grafi

Analogamente a quanto già illustrato a proposito degli alberi, anche nel caso dei grafi un'operazione basilare, che spesso è il presupposto per poter risolvere il problema dato, è l'accesso sistematico a tutti i nodi, uno dopo l'altro, al fine di poter effettuare una specifica operazione (che dipende ovviamente dal problema posto) su ciascun nodo, attraversando tutti gli archi, ossia la ***visita del grafo***.

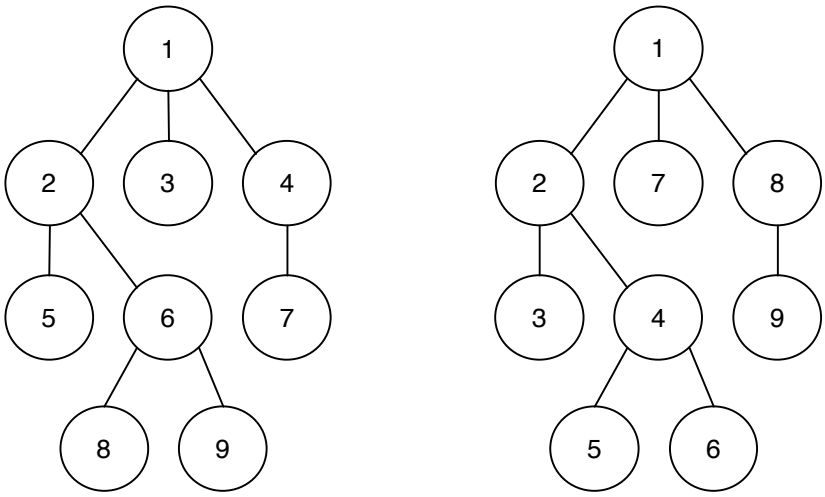
Partendo da uno specifico nodo, detto ***nodo di partenza***, e di solito identificato con *s*, si può desiderare di accedere a tutti i nodi che si trovano a una certa distanza da esso prima di accedere a quelli situati a distanza maggiore, e in questo caso si parla di ***visita in ampiezza (BFS, Breadth-first search)***; oppure si vuole accedere ai nodi allontandosi sempre di più dal nodo di partenza finché è possibile, e in questo caso si parla di ***visita in profondità (DFS, depth-first search)***.

La visita in profondità è molto adatta ad essere realizzata in modo ricorsivo ma può, come vedremo, essere definita in modo iterativo con l'ausilio di una pila.

Viceversa, la visita in ampiezza è poco adatta a un'implementazione ricorsiva e viene generalmente realizzata in modo iterativo con l'ausilio di una coda.

Per capire di cosa stiamo parlando, ricordiamo che un albero è un grafo con certe proprietà (connesso e aciclico), e vediamo come vengono implementate le due visite ora menzionate su questi grafi molto semplici.

Nella figura seguente, i valori associati ai nodi indicano l'ordine col quale i nodi stessi vengono visitati. A sinistra è illustrato il percorso seguito dalla visita in ampiezza e a destra quello seguito dalla visita in profondità.



Queste due visite sono concettualmente piuttosto semplici però, nel momento in cui si passa dagli alberi ai grafi, ci sono alcune complicazioni in più: dato che nei grafi possono essere presenti dei cicli (assenti negli alberi) la visita deve essere in grado di "ricordarsi" se un determinato nodo *v* è stato già visitato, poiché è possibile che diversi cammini che originano nel nodo di partenza conducano a *v*, e in tal caso il nodo *v* sarà "scoperto" più volte. Per questa ragione, nelle visite di grafi è necessario gestire alcune informazioni aggiuntive.

9.3.1 Alberi di visita

Indipendentemente dalla politica usata per determinare l'ordinamento nella scoperta di nuovi nodi, osserviamo che le visite hanno come prima conseguenza quella di verificare la connessione del grafo di partenza infatti, se partendo da un qualunque *s* si riescono a scoprire tutti gli *n* nodi, non si può che concludere che il grafo è connesso. Viceversa, se si scopre solo un sottinsieme dei nodi, allora sapremo che quel sottinsieme induce la componente connessa contenente *s* e, per trovare le altre componenti, bisogna cominciare una nuova visita da un nodo non ancora visitato. Sarà necessario cominciare tante nuove visite quante sono le componenti connesse del grafo.

Pag 164

Mettiamoci ora nell'ipotesi che il grafo sia connesso e consideriamo il sottografo del grafo di partenza che ha come insieme di nodi l'intero insieme *V*, mentre l'insieme degli archi è costituito dall'insieme degli archi utilizzati dalla visita per scoprire nuovi nodi.

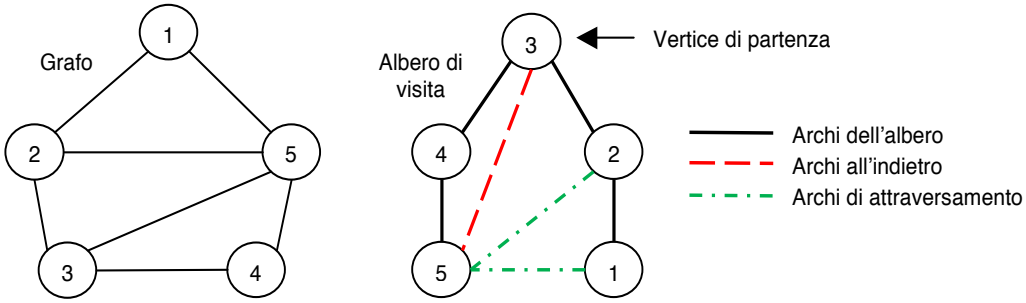
E' piuttosto semplice convincersi del fatto che tale sottografo è un albero. Infatti:

- è connesso in quanto, se *G* è connesso, allora ogni nodo è raggiungibile da *s* mediante un cammino, e questo risulterà durante la visita;
- è aciclico, infatti se per assurdo non lo fosse, potremmo ordinare i nodi che fanno parte del ciclo secondo la loro distanza da *s*; allora, il nodo più lontano da *s* risulterebbe avere due nodi dal quale è stato scoperto, e questo è ovviamente assurdo.

Una volta che ci siamo convinti che tale sottografo è un albero, possiamo osservare che è, in realtà, un **albero ricoprente,** poiché il suo insieme dei nodi coincide con quello dell'intero grafo.

Gli archi del grafo che fanno parte anche dell'albero ricoprente sono chiamati appunto **archi dell'albero**. Tutti gli altri archi, denominati genericamente **archi non dell'albero**, possono essere partizionati in due sottinsiemi: **archi all'indietro** e **archi di attraversamento**:

- gli archi all'indietro sono quelli che congiungono due nodi che sono uno l'antenato dell'altro nell'albero di visita;
- gli archi di attraversamento sono tutti gli altri.



9.3.2 Visita in ampiezza (BFS)

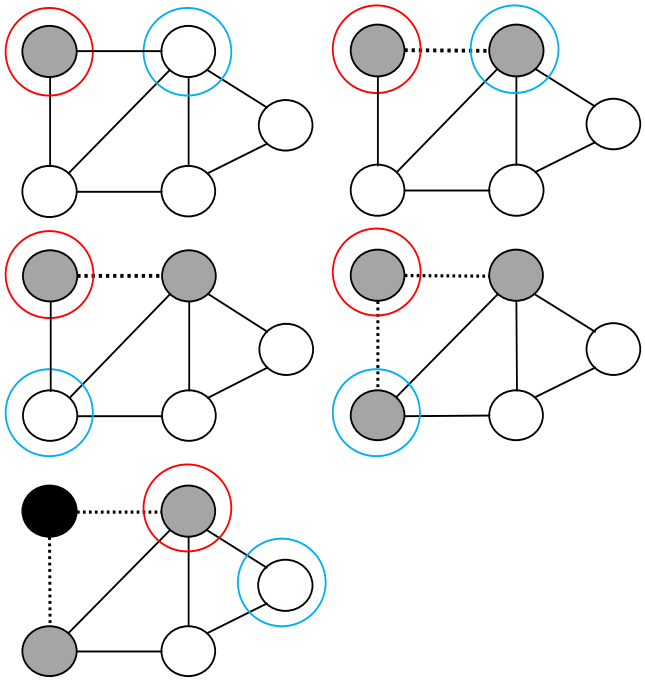
Dato un grafo *G* ed uno specifico ***nodo di partenza s***, la visita in ampiezza (BFS, Breadth-first search) si muove sistematicamente lungo gli archi di *G* per

"scoprire", e successivamente visitare, ogni nodo *v* raggiungibile da *s*. Contestualmente costruisce un ***albero breadth-first*** (che inizialmente consiste del solo nodo s, la sua radice) che, alla fine, contiene tutti i nodi raggiungibili da *s*. Come vedremo più avanti, per ogni nodo *v* raggiungibile da *s,* il cammino da *s* a *v* nell'albero breadth-first è un cammino minimo da *s* a *v*, ossia un cammino che contiene il minimo numero possibile di archi.

La visita in ampiezza espande la "frontiera" fra nodi già scoperti e nodi non ancora scoperti, allontanandola uniformemente e progressivamente dal nodo di partenza. In particolare, la visita scopre tutti i nodi a distanza *k* dal nodo di partenza prima di scoprire anche un solo nodo a distanza *k + 1*.

Come accennato nel paragrafo precedente, è necessario tenere traccia dei nodi già visitati per evitare di visitarli più di una volta. A tale scopo i nodi vengono opportunamente ***marcati***.

Nel corso della scansione dei nodi adiacenti ad un nodo *u* già marcato (quindi già scoperto – cerchiato di rosso nella figura), ogni volta che un nuovo nodo non marcato *v* viene scoperto (nodi cerchiati in azzurro nella figura), il nodo v viene marcato (reso grigio in figura) e l'arco *(u, v)* viene aggiunto all'albero breadth-first (arco tratteggiato nella figura). Non appena fra tutti gli adiacenti di *u* non vi è più alcun nodo bianco, il lavoro sul nodo *u* è terminato (*u* è reso nero in figura) e si prosegue passando ad un altro nodo marcato.



L'albero breadth-first si memorizza mediante un campo aggiuntivo in ogni nodo, il campo ***predecessore***: aggiungere l'arco *(u, v)* all'albero breadth-first significa memorizzare nel campo *predecessore(v)* l'indice *u*. Poiché ogni nodo viene marcato solo una volta, ogni nodo avrà un solo predecessore e questo garantisce l'assenza di cicli nell'albero breadth-first: infatti, se vi fosse un ciclo, almeno un nodo del ciclo avrebbe due predecessori.

Si noti che il predecessore di un nodo nell'albero è, di fatto, suo padre, se l'albero viene considerato come radicato in *s.*

Lo pseudocodice riportato nel seguito assume che il grafo *G = (V, E)* in input sia rappresentato mediante una generica memorizzazione, ed il modo di ricercare gli adiacenti di un nodo dato non è quindi specificato.

Lo pseudocodice utilizza alcune informazioni aggiuntive, necessarie al suo corretto funzionamento:

- un campo *marked[v]* in ciascun nodo per gestire la marcatura dei nodi;
- un campo *pred[v]* in ciascun nodo per memorizzare il predecessore (inizialmente *NULL*);
- una coda Q per gestire l'ordine di visita dei nodi.

```
Funzione Breadth-first_search (G: grafo; q: puntatore a coda)
       Assegna a tutti i nodi v              //inizializzazione
              marked[v] ← false; pred[v] ← NULL
       scegli il nodo di partenza s         //si inizia da s
       marked[s] ← true; Enqueue(q, s)
       finché la coda Q non è vuota
              u ← Dequeue(q)               // u è il nodo su cui lavorare
              visita il nodo u             //operazione generica
              per ogni nodo v adiacente ad u
                     if marked[v] = false then
                            marked[v] = true; Enqueue(q, v)
                            pred[v] ← u
       return
```

Studiamo ora il costo computazionale della visita in ampiezza.

L'inizializzazione ha costo $O(|V|)$. Dopo tale fase, nessun nodo viene più "smarcato", per cui ogni nodo viene marcato, inserito nella coda (e quindi estratto) al massimo una volta. Le operazioni di inserimento ed estrazione dalla coda hanno costo $\Theta(1)$, quindi il tempo totale dedicato alle operazioni sulla coda è $O(|V|)$.

Per calcolare il costo computazionale della seconda parte della visita (in cui si scorre ogni nodo *v* adiacente ad *u*), bisogna tener conto della struttura dati a disposizione.

Infatti, se ci troviamo su liste di adiacenza, la lista di ciascun nodo è scorsa una volta sola, quando il nodo è estratto dalla coda. Poiché la somma delle lunghezze di tutte le liste di adiacenza è $\Theta(|E|)$, il tempo dedicato alle scansioni delle liste è dello stesso ordine. Dunque il costo computazionale in tal caso è $\Theta(|V| + |E|)$.

Pag 168

Se ci troviamo invece su matrice di adiacenza, trovare tutti gli adiacenti di un nodo costa $\Theta(|V|)$, e questa operazione deve essere ripetuta per ogni nodo. Ne consegue un costo computazionale di $\Theta(|V|^2)$.

Se ci troviamo, infine, su matrice di incidenza, per trovare tutti i nodi adiacenti, devo scorrere la riga relativa ad *u* e, per ogni 1 trovato, scorrere la colonna corrispondente, per un costo di $\Theta(|E|)+deg(v)O(|V|)$. Poiché questa operazione va ripetuta per tutti i nodi, il costo complessivo è di $\Theta(|E|\,|V|)$.

La visita in ampiezza ha molte caratteristiche peculiari, che possono poi essere sfruttate per risolvere dei problemi opportuni, come vedremo.

Innanzi tutto, osserviamo che viene operata una politica di tipo FIFO (First In First Out) e questo viene confermato dall'utilizzo della coda.

Inoltre, l'albero di visita che scaturisce da una BFS ha una struttura partcolare.

Infatti si può osservare che, rispetto a tale albero, **nell'insieme degli archi non dell'albero non sono presenti archi all'indietro e gli archi di attraversamento collegano nodi sullo stesso livello o su due livelli adiacenti**. Per convincersi che fra gli archi non dell'albero non ci sono archi all'indietro, basta considerare il fatto che, quando ci si trova su un nodo *u,* si visitano tutti i suoi adiacenti non visitati, perciò non è possibile trovare nel sottoalbero di *u* un nodo *v*, scoperto quindi solo successivamente, che sia adiacente a *u*, perché altrimenti tale nodo sarebbe stato scoperto come adiacente di *u* e l'arco (u, v) farebbe parte dell'albero.

Per un motivo analogo gli archi di attraversamento non possono collegare nodi che non siano sullo stesso livello o su livelli consecutivi: infatti, se consideriamo due nodi *u* e *v* sui livelli *i* ed $i + j$, $j > 1$, rispettivamente, questi non possono in alcun modo essere collegati da un arco non dell'albero, perché allora il nodo *v* a livello $i + j$ sarebbe stato scoperto come figlio di *u* e l'arco *(u, v)* farebbe parte dell'albero.

Inoltre, **la visita in ampiezza individua i cammini più corti da *s* ad ogni altro nodo, e questi sono i cammini sull'albero**. Per convincersi di questo, si

consideri un qualunque nodo *v*, ed il cammino più corto che collega *v* ad *s*. Tale cammino può essere costituito da archi dell'albero, archi di attraversamento tra nodi sullo stesso livello ed archi di attraversamento tra nodi su livelli consecutivi. Gli archi dell'albero e quelli di attraversamento tra nodi su livelli adiacenti portano da un livello *i* ad un livello *i + 1*, mentre gli archi di attraversamento che collegano nodi sullo stesso livello non fanno procedere verso livelli successivi. Ne consegue che questo ipotetico cammino più corto non potrà che essere della stessa lunghezza, o addirittura più lungo, del cammino che da *v* risale sull'albero verso *s*.

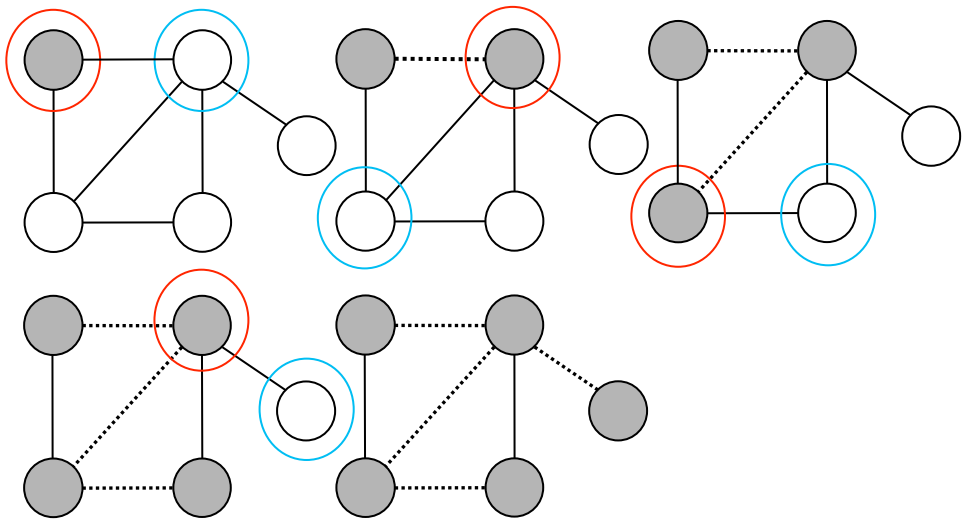
9.3.3 Visita in profondità

Dato un grafo *G* ed un nodo di partenza s, la visita in profondità (DFS, Depth-first search), a differenza di quella in ampiezza, esplora il grafo allontanandosi sempre più dal nodo di partenza finché è possibile, per poi ripartire da nodi più vicini a quello di partenza solo quando non è possibile allontanarsi ancora.

In particolare, se la visita in profondità -muovendosi lungo l'arco *(u, v)* del grafo- "scopre" il nodo *v*, proseguirà percorrendo uno degli archi uscenti da *v* e non percorrendo uno degli altri archi uscenti da *u*.

Contestualmente, la visita in profondità costruisce un ***albero depth-first*** che ha come radice il nodo *s* e contiene tutti i nodi raggiungibili da *s*.

Come nel caso della visita in ampiezza, è necessario tenere traccia dei nodi già visitati per evitare di visitarli più di una volta. A tale scopo i nodi vengono marcati, con una logica analoga a quella utilizzata nella visita in ampiezza, solamente la prima volta che vengono incontrati.



Il lavoro su un nodo *u* già marcato (quindi già scoperto – nodo cerchiato di rosso nella figura) consiste nella ricerca di un suo adiacente non marcato *v* (nodi cerchiati in azzurro nella figura). Il nodo v viene marcato (reso grigio in figura) e l'arco *(u, v)* viene aggiunto all'albero depth-first (arco tratteggiato nella figura). Il lavoro prosegue quindi sul nodo *v* anziché (come nella ricerca

breadth-first) su un ulteriore adiacente di *u*. Si torna a lavorare su nodi precedenti solo quando non è più possibile proseguire il lavoro allontanandosi ulteriormente dal nodo di partenza.

Anche l'albero depth-first si realizza mediante un campo aggiuntivo in ogni nodo, il campo ***predecessore***: aggiungere l'arco *(u, v)* all'albero depth-first significa memorizzare nel campo *predecessore(v)* l'indice *u*. Poiché ogni nodo viene marcato solo una volta, ogni nodo avrà un solo predecessore (che, anche in questo caso è, di fatto, il padre nell'albero radicato in *s*) e questo garantisce l'assenza di cicli nell'albero depth-first.

La visita in profondità può essere espressa ricorsivamente in modo semplice e diretto, analogamente a quanto già discusso in relazione alle visite di alberi.

Lo pseudocodice riportato nel seguito assume che il grafo *G = (V, E)* in input sia rappresentato mediante una generica memorizzazione. Le informazioni aggiuntive, necessarie al suo corretto funzionamento, sono le stesse già discusse per la visita in ampiezza:

- un campo *marked[v]* in ciascun nodo per gestire la marcatura dei nodi;
- un campo *pred[v]* in ciascun nodo per memorizzare il predecessore (inizialmente *NULL*).

Inoltre, lo pseudocodice presuppone che al momento della chiamata iniziale (alla quale si passa il nodo di partenza) tutti i nodi siano già inizializzati con il campo *marked* a *false* ed il campo *pred* a *NULL*.

```
Funzione Depth_first_search_ricorsiva (u: nodo)
       marked[u] ← true
       visita il nodo u                          //operazione generica
       per ogni nodo v adiacente ad u
              if marked[v] = false then
                     pred[v] ← u
                     Depth_first_search_ricorsiva (v)
       return
```

Pag 172

Per quanto riguarda il costo computazionale della visita ricorsiva valgono le seguenti considerazioni, supponendo di avere un grafo connesso dato mediante liste di adiacenza:

- per ogni nodo v_i raggiungibile dal nodo di partenza vi è una e una sola chiamata ricorsiva, che avviene quando il nodo viene scoperto per la prima volta (e quindi non è ancora marcato): $\Theta(n)$;
- nella chiamata ricorsiva relativa al nodo v_i si scandisce la lista dei suoi adiacenti: $\Theta(degree(v_i))$.

Quindi il costo totale è:

$$\Theta(n) + \Theta(\sum_{i=1}^{n} degree(v_i)) = \Theta(n+m)$$

Analogamente al caso della visita in ampiezza, il costo della visita in profondità quando il grafo sia memorizzato su matrice di adiacenza è $\Theta(n^2)$ mentre è $\Theta(nm)$ nel caso di grafo memorizzato su matrice di incidenza.

Anche la visita in profondità ha delle interessanti proprietà.

Innanzi tutto è caratterizzata da una politica di tipo LIFO (Last In First Out), tanto è vero che, come vedremo, può essere realizzata con l'ausilio di una pila.

Anche l'albero di visita che scaturisce da una DFS ha una struttura partcolare.

Infatti si può osservare che, rispetto a tale albero, **nell'insieme degli archi non dell'albero non sono presenti archi di attraversamento**. Infatti, è assurdo pensare che esista un tale arco *(u, v)* poiché, durante la visita in profondità, il nodo *v* sarebbe trovato come discendente di *u* o viceversa (a seconda che la visita passi prima per *u* o prima per *v*) e quindi tale arco farebbe parte dell'albero.

Questa proprietà implica il seguente:

Teorema. *Dato un grafo G, o G ha un cammino di lunghezza k o* $|E| = O(kn)$.

Dimostrazione. Se *G* ha un cammino lungo *k* abbiamo finito, quindi supponiamo che ogni cammino sia di lunghezza $< k$. Limitiamo $|E|$ contando il numero di antenati di ogni nodo, che corrisponde al suo massimo grado possibile. Infatti, non esistendo archi di attraversamento fra quelli non

dell'albero, ogni nodo può essere collegato ad altri nodi esclusivamente da archi dell'albero ed archi all'indietro, ossia archi che lo collegano a propri antenati. Poiché tutti i cammini sono più corti di *k*, il numero degli antenati di ciascun nodo è < k, da cui segue che il numero di archi è minore di *kn*. CVD

9.3.4 Somiglianze fra la visita in ampiezza e la visita in profondità

Anche se non è molto evidente dallo pseudocodice illustrato nei due paragrafi precedenti, la visita in ampiezza e quella in profondità sono molto strettamente correlate fra loro.

In effetti, con un piccolo sforzo, le due visite possono essere organizzate nello stesso identico modo, con l'unica differenza della struttura dati d'appoggio utilizzata: una coda nella visita in ampiezza ed una pila nella visita in profondità (le differenze sono evidenziate in grassetto sottolineato).

Perché sia possibile questa identità nella struttura dello pseudocodice è necessario inserire ed estrarre dalle strutture dati d'appoggio archi anziché nodi, altrimenti la visita in profondità non coincide funzionalmente con quella definita ricorsivamente.

Pseudocodice della visita in ampiezza

```
Funzione Breadth-first_search (G: grafo; q: puntatore a coda)
      Assegna a tutti i nodi v
            marked[v] ← false; pred[v] ← NULL
      scegli il nodo di partenza s
      marked[s] ← true
      visita s
      per ogni nodo t adiacente ad s
            Enqueue(q, arco(s, t))
      finché la coda Q non è vuota
            arco(x, y) ← Dequeue(q)
            if (marked[y] = false)
                  marked[y] ← true; pred[y] ← x; visita(y)
                  per ogni nodo z adiacente ad y
                        Enqueue(q, arco(y, z))
      return
```

Pseudocodice della visita in profondità

```
Funzione Depth-first_search (G: grafo; p: puntatore a pila)
        Assegna a tutti i nodi v
                marked[v] ← false; pred[v] ← NULL
        scegli il nodo di partenza s
        marked[s] ← true
        visita s
        per ogni nodo t adiacente ad s
                Push(p, arco(s, t))
        finché la pila P non è vuota
                arco(x, y) ← Pop(p)
                if (marked[y] = false)
                        marked[y] ← true; pred[y] ← x; visita(y)
                        per ogni nodo z nella adiacente ad y
                                Push(p, arco(y, z))
        return
```

Costo computazionale di entrambe le visite

L'inizializzazione ha costo $\Theta(|V|) = \Theta(n)$. Dopo tale fase, nessun nodo viene più "smarcato", per cui ogni nodo viene marcato una sola volta, e ciò significa che, per ciascun nodo, uno solo degli archi in esso incidenti viene inserito nella (e quindi estratto dalla) struttura dati d'appoggio. Le operazioni di inserimento ed estrazione dalla struttura dati d'appoggio hanno costo $\Theta(1)$, quindi il tempo totale dedicato alle operazioni sulla struttura dati d'appoggio è $\Theta(n)$.

L'insieme degli adiacenti di ciascun nodo è scorso una volta sola, quando l'arco incidente nel nodo (arco che, per quanto detto sopra, è l'unico ad essere stato inserito) viene estratto dalla struttura dati d'appoggio.

La scansione di tale insieme però ha un costo che dipende dalla struttura dati utilizzata per l'implementazione. In particolare per un grafo con n nodi ed m archi:

- liste di adiacenza: per ogni nodo v_i si scandisce la lista dei suoi adiacenti, con un costo $O(degree(v_i))$, quindi il costo totale è $\Theta(n) + \Theta(\sum_{i=1}^{n} degree(v_i)) = \Theta(n) + \Theta(m)$;

- matrice di adiacenza: per ogni nodo v_i si scandisce la riga i-esima della matrice di adiacenza, con un costo $\Theta(n)$, quindi il costo totale è $\Theta(n^2)$;
- matrice di incidenza: per ogni nodo v_i si scandisce la riga i-esima della matrice di incidenza, con un costo $\Theta(m)$ e, per ciascun 1, si scorre al più l'intera colonna corrispondente; quindi per ogni nodo si spende Θ(m)+deg(v)O(n); sommano tutti i contributi, il costo totale è $\Theta(nm)$;
- lista di archi: per ogni nodo v_i si scandisce l'intero vettore contenente gli archi, con un costo $O(m)$, quindi il costo totale è $O(nm)$.

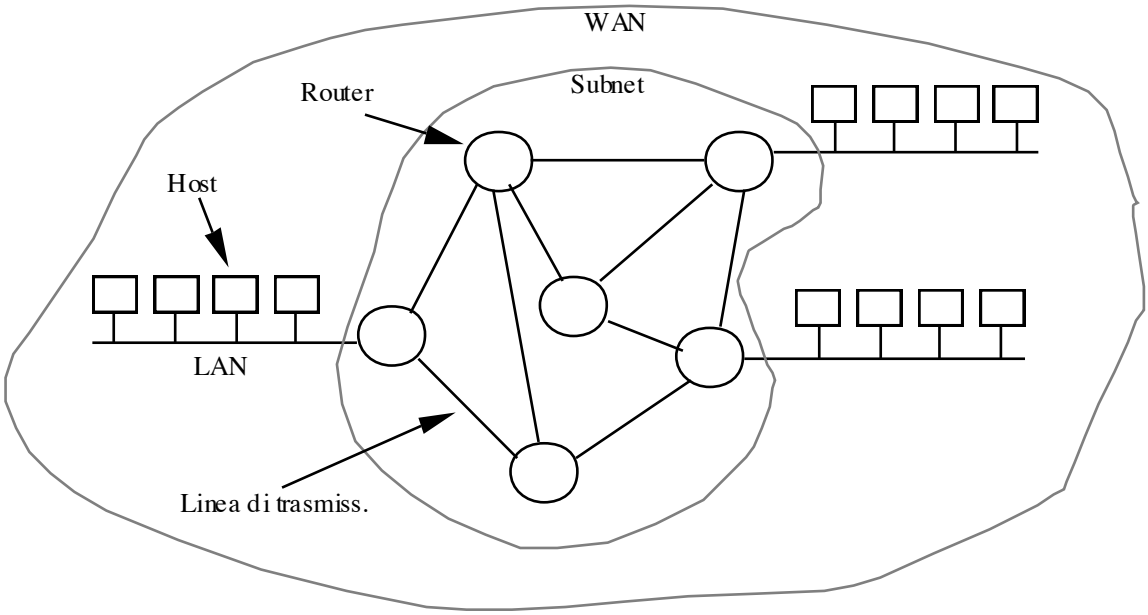
9.4 Reti e grafi

Numerose reti di importanza vitale per la nostra società sono modellate per mezzo di grafi. Si pensi ad esempio alle:

- reti autostradali;
- reti ferroviarie;
- reti per la distribuzione dell'energia elettrica;
- reti telefoniche;
- la rete planetaria Internet.

Nell'ambito di tali strutture un problema di grande rilevanza è determinare il **cammino di costo minimo** fra un punto ed un altro nella rete (ossia fra un vertice ed un altro nel grafo che modella la rete). In tali scenari ogni arco del grafo ha associato un **costo di attraversamento**, di norma strettamente positivo, la cui determinazione dipende da una combinazione delle grandezze in gioco nello specifico contesto (as es. distanza chilometrica e costi di trasporto e pedaggio nelle reti stradali e ferroviarie, ampiezza di banda trasmissiva e costi di affitto delle linee nelle reti telefoniche e di elaboratori, ecc.).

In particolare, la rete Internet è organizzata secondo questo schema:



Ciò che consente il dialogo fra i vari host sparsi per tutto il pianeta è una infrastruttura di rete, chiamata **subnet di comunicazione**, costituita da centinaia di migliaia di apparati (detti **router**) che sono collegati fra loro mediante una rete di collegamenti punto-a-punto e che si incaricano di inoltrare tutti i dati trasmessi da ogni host sorgente ad ogni host di destinazione. E' evidente come sia di cruciale importanza che tali dati vengano instradati, nella subnet, lungo la via più conveniente fra quelle che consentono di raggiungere la destinazione.

Esistono vari algoritmi per la ricerca dei cammini migliori, noi vedremo l'algoritmo di Dijkstra (1956).

9.4.1 Algoritmo di Dijkstra per la ricerca dei cammini minimi

Dato un grafo orientato e pesato G con pesi degli archi non negativi ed uno specifico vertice di partenza s, l'algoritmo di Dijkstra si muove sistematicamente lungo gli archi di G per costruire un albero dei cammini minimi (che inizialmente consiste del solo vertice s, la sua radice) che, alla fine, contiene tutti i vertici

raggiungibili da s. Per ogni vertice v raggiungibile da s il cammino da s a v nell'albero dei cammini minimi è un cammino di peso minimo da s a v, ossia un cammino che ha la seguente proprietà:

- la somma dei pesi degli archi che costituiscono il cammino è minore o uguale alla somma dei pesi degli archi di qualunque altro cammino da s a v.

L'algoritmo di Dijkstra visita i nodi nel grafo, in maniera simile a una ricerca in ampiezza o in profondità.

In ogni istante, l'insieme dei vertici del grafo è diviso in tre parti:

- l'insieme ***VIS*** dei nodi visitati, per i quali è ormai fissato definitivamente il cammino minimo;
- l'insieme ***F*** dei nodi di frontiera, che sono successori dei nodi visitati, per i quali ancora non è definitivo il cammino minimo;
- i nodi sconosciuti, che sono ancora da esaminare.

Per ogni nodo v, l'algoritmo tiene traccia:

- di un valore *distanza(v)*, che rappresenta il costo noto finora del cammino minimo da s al vertice v;
- dell'indice *predecessore(v)*, il cui ruolo è identico a quanto discusso nelle visite in ampiezza e profondità: esso identifica il predecessore di v nell'attuale cammino minimo da s a v;
- di un valore *vis(v)*, che indica se il nodo è in VIS oppure no.

L'algoritmo consiste nel ripetere la seguente iterazione finché l'insieme F non si svuota:

- si prende dall'insieme F un qualunque nodo v avente *distanza(v)* minima;
- si sposta v da F in VIS;
- si inseriscono tutti i successori di v ancora sconosciuti in F;
- per ogni successore w di v (w∈F) si aggiornano i valori *distanza(w)* e *predecessore(w)*. L'aggiornamento viene effettuato con la regola:

distanza(w) = minimo fra *distanza(w)* e (*distanza(v)* + peso dell'arco (u,w))

- se il valore di *distanza(w)* è stato effettivamente modificato, allora *predecessore(w)* viene posto uguale a v, il che ci permette di ricordare che, al momento, il cammino di peso minimo che conosciamo per arrivare da s a w ha come penultimo nodo v.

Pag 178

L'algoritmo segue un'idea piuttosto naturale: se sappiamo che con peso pari a distanza(v) possiamo arrivare fino a v, allora arrivare a w non può costare più di arrivare a v e spostarsi da v lungo l'arco (v,w) fino a w.

Si noti che una fondamentale differenza con la visita in ampiezza è legata al fatto che dobbiamo mantenere i nodi in F ordinati rispetto alla loro distanza nota da s, il che implica che la struttura d'appoggio usata nella visita non può più essere una coda ma diviene una coda con priorità, nella quale l'elemento da estrarre è quello avente distanza minima dal vertice s di partenza.

Si noti che l'algoritmo funziona anche su grafi non orientati: è sufficiente sostituire, ad ogni arco non orientato del grafo, una coppia di archi orientati in senso opposto l'uno all'altro ed aventi ciascuno lo stesso peso dell'arco non orientato che sostituiscono.

Pseudocodice

Lo pseudocodice riportato nel seguito assume che il grafo *G = (V, E)* in input sia rappresentato mediante liste di adiacenza.

Lo pseudocodice utilizza alcune informazioni aggiuntive, necessarie al suo corretto funzionamento:

- un campo *dist[v]* in ciascun vertice per gestire il costo del cammino minimo da s a v (inizialmente posto a ∞) ;
- un campo *pred[v]* in ciascun vertice per memorizzare il predecessore (inizialmente posto a *NULL*);
- un campo *vis[v]* in ciascun vertice per indicare se il vertice è o no in VIS (inizialmente posto a *false*);
- una coda con priorità q per gestire l'ordine di visita dei vertici: i vertici presenti via via nella coda q sono quelli dell'insieme *F*.

```
Funzione Dijkstra (G: grafo pesato con pesi non negativi; Q: coda con priorità)
      Assegna a tutti i nodi v
            dist [v] ← ∞
            pred[v] ← NULL
            vis[v] ← false
      scegli il nodo di partenza s
      dist[s] ← 0
      Enqueue(Q, s)
      finché la coda Q non è vuota
            v ← Dequeue(Q) // v è il vertice su cui lavorare
            vis[v] ← true
            //per il vertice v il cammino minimo è ora quello definitivo
              per ogni vertice w nella lista di adiacenza di v tale che
              vis[w]=false
                   if dist[w] > dist[v] + peso_arco(v,w) then
                         dist[w] = dist[v] + peso_arco(v,w)
                         pred[w] ← v
                   if (w non è nella coda q)
                         Enqueue(Q, w)
                   else   "aggiusta la posizione di w in Q"
      return
```

Esempio

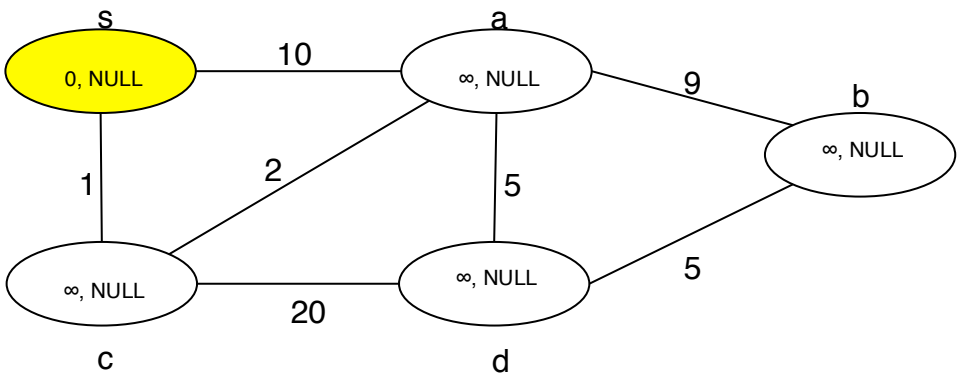
Nell'esempio si adotta la seguente simbologia:

- il vertice s è il vertice di partenza;
- i vertici verdi sono quelli in VIS;
- i vertici gialli sono quelli in F;
- i vertici arancio sono vertici già presenti in F i cui valori vengono aggiornati durante un'iterazione;
- all'interno di ogni vertice v si indicano i valori di *distanza(v)* e *predecessore(v);*
- gli archi spessi sono quelli dell'albero dei cammini minimi.

Pag 180

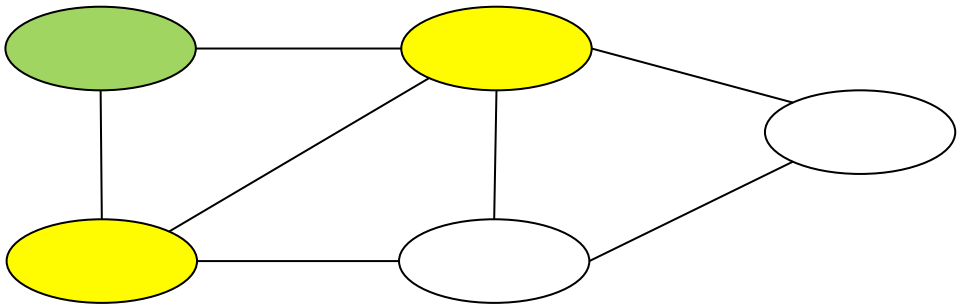
Inizializzazione

Dopo l'inizializzazione la situazione è questa (s è l'unico vertice nella coda):



Prima iterazione

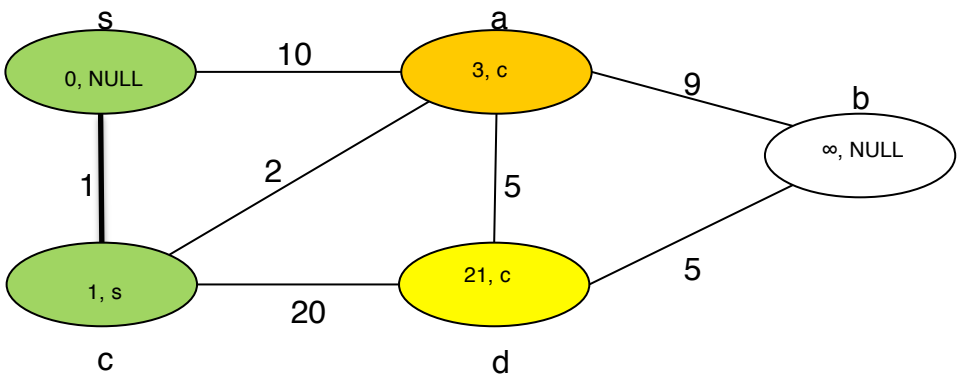
Nella prima iterazione del while si estrae dalla coda la sorgente (che quindi viene spostata in VIS) e si mettono in coda (e quindi in F) i suoi adiacenti, aggiornandone i valori:



Seconda iterazione

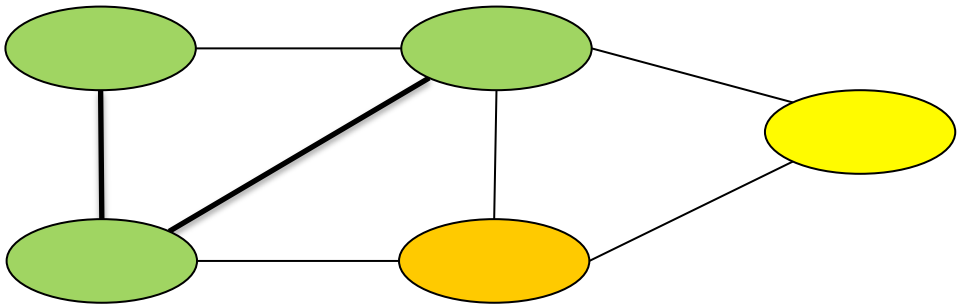
Nella seconda iterazione fra i vertici in F si sceglie quello a distanza minima (c), che viene spostato in VIS, si aggiungono i nuovi vertici (d) scoperti tramite c e si aggiornano i valori di quelli (a) già in F:

Pag 181



Terza iterazione

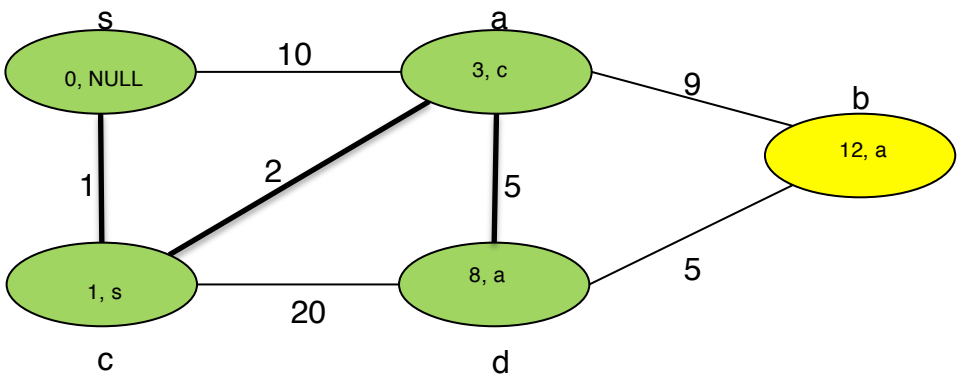
Nella terza iterazione fra i vertici in F si sceglie quello a distanza minima (a), che viene spostato in VIS, si aggiungono i nuovi vertici (b) scoperti tramite a e si aggiornano i valori di quelli (d) già in F:



Quarta iterazione

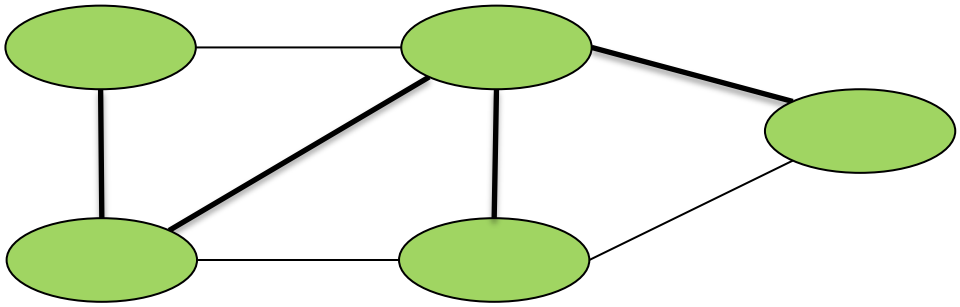
Nella successiva iterazione fra i vertici in F si sceglie quello a distanza minima (d), che viene spostato in VIS, e si aggiornano i valori di quelli già in F (non c'è più alcun nuovo vertice da scoprire). In questo caso però l'aggiornamento non modifica i valori del vertice b dato che *distanza(d) + peso arco (d,b) > distanza(b)*:

Pag 182



Quinta ed ultima iterazione

Fra i vertici in F si sceglie quello a distanza minima (b), che è anche l'ultimo presente nella coda. Esso viene spostato in VIS e si termina:



Costo computazionale

L'inizializzazione ha complessità $O(|V|)$.

L'algoritmo compie successivamente $O(|V|)$ iterazioni, dato che ad ogni iterazione si fissa il cammino minimo per un nuovo vertice (quello estratto dalla coda).

Nel complesso di tali iterazioni si effettuano:

- $O(|V|)$ inserzioni nella coda (ogni vertice è inserito una sola volta);
- $O(|V|)$ estrazioni dalla coda (ogni vertice è estratto una volta sola);
- $O(|E|)$ aggiornamenti di un vertice nella coda (un aggiornamento possibile per ogni arco del grafo).

I costi delle varie operazioni sulla coda dipendono dalla sua implementazione.

Se la coda è implementata ad esempio tramite un vettore non ordinato:

- l'inserimento ha costo $\Theta(1)$;
- l'estrazione ha costo $O(|V|)$;
- l'aggiornamento di un vertice ha costo $\Theta(1)$.

Dunque in tal caso la complessità risulta $O(|V| + |V| + |V|^2 + |E|) = O(|V|^2 + |E|) = O(|V|^2)$.

Viceversa, se la coda è implementata tramite uno heap:

- l'inserimento, con riaggiustamento dello heap, ha costo $O(\log|V|)$;
- l'estrazione, con riaggiustamento dello heap, ha costo $O(\log|V|)$;
- l'aggiornamento di un vertice, con riaggiustamento dello heap, ha costo $O(\log|V|)$.

e la complessità diviene $O(|V| + |V| \log|V| + |V| \log|V| + |E| \log|V|) = O((|V| + |E|) \log|V|)$.

Si noti che questa complessità è migliore della precedente se il grafo è sparso, ossia ha pochi archi: in tal caso, se ad esempio $|E| = O(|V|)$ la complessità totale è $O((|V|\log|V|)$.

Viceversa, essa diviene peggiore della precedente se il grafo è molto denso, ossia molto ricco di archi, poiché in tal caso $|E| = O(|V|^2)$ e il costo totale diviene $O(|V|^2 \log|V|)$.

Correttezza dell'algoritmo

La correttezza dell'algoritmo di Dijkstra si dimostra per induzione:

L'algoritmo è banalmente corretto quando:

- |VIS| = 1: VIS contiene solo la sorgente, che è a distanza zero da se stessa;
- |VIS| = 2: VIS contiene solo la sorgente s ed il vertice collegato ad s dall'arco di peso minimo fra quelli uscenti da s;

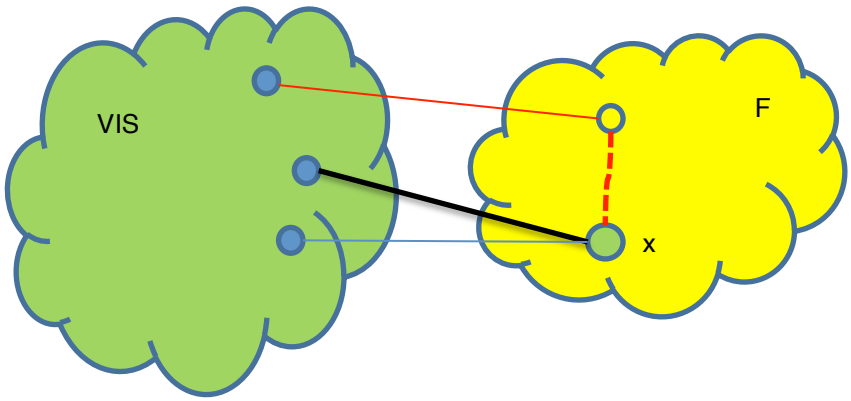
Ora, sia IVISI = k; il prossimo vertice che verrà inserito in VIS è un vertice x (in verde nella figura), fra quelli in F, tale che:

- x è raggiungibile direttamente da un vertice in VIS;

Pag 184

- *distanza(x)* ha valore minimo fra i vertici in F.

Se esistesse un cammino C che permette di raggiungere x con un peso complessivo strettamente minore di *distanza(x)*, esso dovrebbe necessariamente passare anche per altri vertici in F dato che passando per i soli vertici in VIS la distanza minima è appunto *distanza(x)*. Ma tale cammino sarebbe costituito da una porzione in VIS, da un arco che porta da VIS a F (in rosso nella figura), più una porzione in F (in rosso tratteggiato nella figura). Dato che la somma del costo della porzione di cammino in VIS più il costo dell'arco da VIS ad F non può essere minore di *distanza(x)* (altrimenti x non sarebbe il nodo scelto) ne segue che la porzione in F dell'ipotetico cammino C dovrebbe avere peso negativo, il che contraddice l'ipotesi di partenza che tutti gli archi del grafo abbiano peso ≥ 0.



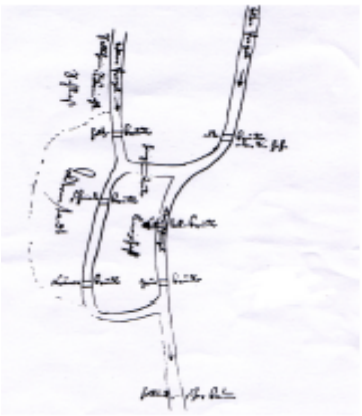
Dunque anche il nuovo insieme VIS, avente cardinalità k + 1, contiene vertici per i quali è determinato il cammino minimo da s.

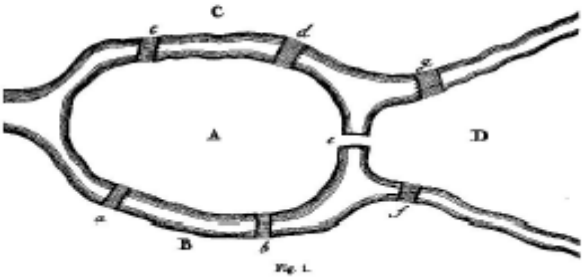
CVD

9.5 Alcuni problemi classici sui grafi

9.5.1 Grafi euleriani

La nascita della moderna teoria dei grafi si fa risalire al famoso articolo di Eulero del 1736 sul problema dei ponti di Königsberg. Il problema consiste nel fare un giro a piedi della cittadina passando una ed una sola volta attraverso ciascun ponte e ritornando al punto di partenza. Nel suo articolo, Eulero risolve il suo problema passando dallo specifico problema dei ponti di Königsberg al problema generale su qualunque grafo ponendo così le basi della teoria dei grafi. E' curioso osservare che egli non utilizza mai la visualizzazione dei grafi per spiegare i suoi risultati: nel suo lavoro, gli unici disegni rappresentano la mappa di Königsberg.

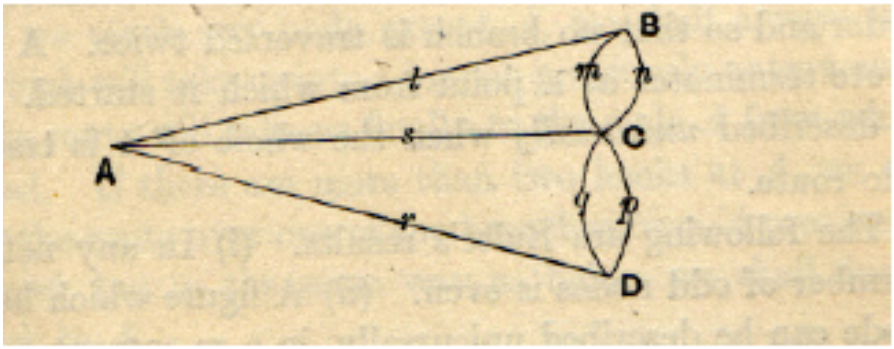




Una spiegazione di questa assenza di interesse per la visualizzazione di grafi può essere trovata nel primo paragrafo del suo articolo, in cui Eulero richiama la visione di Leibnitz di un nuovo tipo di geometria senza misure o grandezze. In effetti, Leibnitz sottolinea l'utilità di una nuova caratteristica, completamente diversa dall'algebra, che trae molto vantaggio da una rappresentazione mentale molto naturale, ma *senza figure*. Eulero ha familiarità con il pensiero di Leibnitz, il quale ribadisce che ne' figure ne' modelli dovrebbero legare l'immaginazione. Eulero è dunque un convinto oppositore della visualizzazione dei grafi per descrivere o risolvere problemi di teoria dei grafi.

Pag 186

Dunque, probabilmente, è solo 150 anni fa che appare la prima raffigurazione di grafo astratto che rappresenta il problema dei ponti di Königsberg (qui sotto), ad opera di W.W. Rouse Ball (1850-1925), e il suo disegno appare in un libro sugli svaghi matematici: le varie zone della cittadina sono rappresentate da nodi ed i ponti sono rappresentati da archi, che collegano opportunamente coppie di zone.



Vediamo le considerazioni di Eulero riguardo questo problema.

Definizione: Un *circuito euleriano* è un cammino che tocca tutti gli archi una e una sola volta tornando al punto di partenza. Un grafo *G* si dice *grafo euleriano* se ammette un circuito euleriano.

Teorema (di Eulero)**.** *Un grafo G=(V,E) connesso è euleriano se e solo se tutti i suoi nodi hanno grado pari.*

Dimostrazione. Condizione necessaria: Sia *G* euleriano. Allora, per definizione esiste un circuito euleriano *C* che attraversa una e una sola volta tutti gli archi del grafo. Il circuito *C* tocca tutti i nodi di *G*. Inoltre *C* entra ed esce in ogni nodo (tranne che per il primo ed ultimo nodo da cui esce all'inizio e in cui entra alla fine) senza mai passare due volte sullo stesso arco. Quindi tutti i nodi devono avere necessariamente grado pari.

Condizione Sufficiente: Per ipotesi, ogni nodo di *G* ha grado pari. Costruiamo un circuito euleriano. Sia *x* un nodo di *G*. Usciamo da *x* attraverso un arco,

Pag 187

entrando in un altro nodo *y* di *G*. Poiché *y* ha grado pari, possiamo uscire da *y* con un arco diverso da quello con cui siamo entrati. Possiamo ripetere il procedimento senza passare mai due volte su uno stesso arco e, poiché il grafo è finito, prima o poi torneremo sul nodo *x*. Abbiamo così costruito un ciclo, che non è necessariamente euleriano, perché potremmo non aver attraversato tutti gli archi del grafo. Allora consideriamo il grafo *G*' ottenuto da *G* cancellando gli archi del ciclo. Se tutti i nodi di *G*' hanno grado 0 allora il ciclo costruito è euleriano. Altrimenti consideriamo un nodo *x*' appartenente al ciclo rimosso che abbia ancora grado positivo. Tale nodo deve esistere perché *G* è connesso. Ripetiamo il procedimento sul grafo *G*' a partire da *x*'. Otteniamo un nuovo ciclo, che interseca il precedente nel nodo *x*'. L'unione dei due cicli è essa stessa un ciclo. Iterando il procedimento, prima o poi tutti gli archi del grafo verranno usati, ottenendo così un ciclo euleriano. **CVD**

9.5.2 Grafi bipartiti ed accoppiamenti

Definizione. Un grafo si dice *bipartito* se l'insieme dei nodi si può ripartire in due sottoinsiemi disgiunti V_1 e V_2 in modo che gli archi uniscano un nodo di V_1 con uno di V_2.

Teorema: *Un grafo è bipartito se e solo se non contiene cicli dispari.*

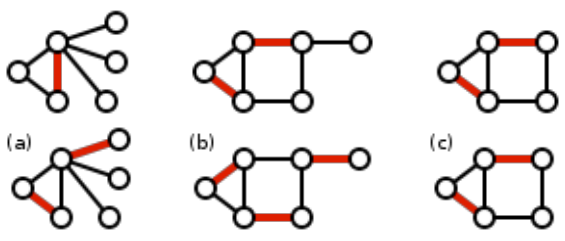
Dimostrazione: (<=) Sia *G=(V,E)* un grafo senza cicli dispari, e mostriamo che esso è bipartito. Ovviamente, un grafo è bipartito se tutte le sue componenti connesse sono grafi bipartiti oppure grafi banali (composti da un unico nodo), quindi non è restrittivo assumere che *G* sia connesso. Sia *T* un albero ricoprente di *G*, e consideriamo la sua radice *r*. Per ogni nodo *v* del grafo, esiste in *T* un unico cammino da *v* ad *r*. Partizioniamo i nodi di *G* a seconda che la loro distanza da *r* sia pari o dispari, e dimostriamo che *G* è bipartito secondo tale partizione. Per fare ciò dimostriamo che, comunque si scelga un arco del grafo, i suoi estremi giacciono in partizioni differenti. Sia *e=(x,y)* un arco di *G*; se *e* è un arco di *T*, allora non è restrittivo assumere che *x* sia il padre di *y*, e quindi *x*

appartiene al cammino che va da *y* ad *r* e, più precisamente, precede *y* nel cammino; quindi *x* ed *y* stanno in partizioni differenti. Infine, sia *e=(x,y)* un arco di *G* ma non di *T*; allora, il cammino da *r* ad *x* su *T*, il cammino da *r* ad *y* su *T*, e l'arco *e* formano un ciclo *C*; tale ciclo è pari per ipotesi, pertanto deve necessariamente essere che uno tra i due cammini da *r* ad *x* e ad *y* sia pari, mentre l'altro sia dispari, cioè *x* ed *y* stanno in partizioni differenti.

(=>) Sia *G=(V,E)* bipartito, pertanto $V=V_1 \cup V_2$. Consideriamo un ciclo di G; esso passa per un qualche nodo *v* in V_1, ed il nodo *w* che segue *v* nel ciclo deve necessariamente appartenere a V_2, per l'ipotesi che il grafo è bipartito; tale ragionamento può essere iterato, pertanto il nodo *u* successivo a *w* nel ciclo appartiene a V_1, il nodo *x* successivo ad *u* nel ciclo appartiene a V_2, e così via. Il ragionamento viene iterato fino a quando, seguendo in ordine i nodi del ciclo, non si torna al nodo di partenza *v*. Essendo stati costretti a passare da una partizione all'altra ad ogni passo, il ciclo non può che essere pari. **CVD**

Definizione. In un grafo, un *accoppiamento* (*matching*) *M* è un sottoinsieme di archi disgiunti, cioè senza estremi in comune.

Di solito, dato un grafo, si cerca un accoppiamento di massima cardinalità. Poiché non sembrano esistere algoritmi semplici per risolvere questo problema, ci si accontenta a volte di determinare un accoppiamento massimale, anziché massimo. In figura sono riportati a sinistra degli accoppiamenti massimali di tre grafi, mentre a destra degli accoppiamenti massimi degli stessi grafi.



Particolare interesse in letteratura ha l'accoppiamento di grafi bipartiti.

Definizione. Dato un grafo bipartito $G=(V_1 \cup V_2, E)$, un suo accoppiamento M si dice *completo* se $|M|=|V_1|$, dove V_1 è l'insieme con cardinalità minore tra i due in cui è stato partizionato l'insieme dei nodi V.

Teorema (di Philip Hall)**.** *Sia $G=(V_1,V_2,E)$ un grafo bipartito con $|V_1| \leq |V_2|$. Allora G ha un accoppiamento completo tra V_1 e V_2 se e solo se per ogni insieme S di k nodi di V_1 vi sono almeno k nodi di V_2 adiacenti ad uno dei nodi di S.*

Sia $adj(S)$ l'insieme dei nodi di V_2 adiacenti a qualche nodo di S, cioè $adj(S) = \{v \in V_2 : \exists u \in S, (u,v) \in E\}$, allora la condizione di Hall diventa: $|S| \leq |adj(S)|$ per ogni $S \subseteq V_1$.

Dimostrazione. Condizione necessaria: Se G ha un accoppiamento completo M ed S è un qualsiasi sottoinsieme di V_1, allora ogni nodo in S è accoppiato da M con un differente nodo in $N(S)$, tale che $|S| \leq |adj(S)|$.

Condizione sufficiente: Per assurdo, sia verificata la condizione di Hall, ma non esista un accoppiamento completo, cioè sia M un accoppiamento massimo, con $|M|<|V_1|$; dimostriamo che esiste un accoppiamento M' con $|M'|=|M|+1$. L'idea della dimostrazione è costruire un cammino i cui nodi siano alternativamente in M e fuori di M.

Diremo, per abuso di linguaggio, che un nodo appartiene ad M se è estremo di un arco di M. Per ipotesi, $|M|<|V_1|$, e dunque esiste un $u_0 \in V_1$, tale che $u_0 \notin M$. Sia $S = \{u_0\}$; per tale insieme vale che $1= |S| \leq |adj(S)|$ per ipotesi, e quindi esiste un nodo $v_1 \in V_2$ adiacente ad u_0. Se $v_1 \notin M$, allora $M' = M \cup \{e\}$, dove $e = (u_0,v_1)$, è l'accoppiamento richiesto. Quindi $v_1 \in M$, per qualche $u_1 \in V_1$, allora prendendo $S=\{u_0,u_1\}$ si ha $2= |S| \leq |adj(S)|$, e deve esistere un altro nodo v_2, distinto da v_1, e adiacente ad u_0 o ad u_1. Se $v_2 \notin M$, si hanno due possibilità:

- v_2 adiacente ad u_1. Consideriamo il cammino v_2,u_1,v_1,u_0, nel quale l'arco $e=(u_1,v_1)$ sta in M, mentre $f=(u_1,v_2)$ e $g=(u_0,v_1)$ no. Formiamo allora il nuovo accoppiamento M' aggiungendo a M gli archi f e g e togliendo e. M' ha un arco in più di M.
- v_2 adiacente ad u_0. Formiamo il nuovo accoppiamento M' aggiungendo ad M l'arco (u_0,v_2). Anche qui M' ha un arco in più di M.

Allora deve risultare $v_2 \in M$; v_2 è adiacente ad u_0 o ad u_1 con un arco che non sta nell'accoppiamento (in quanto $u_0 \notin M$ ed $u_1 \in M$ e u_1 è accoppiato con v_1), ma poiché sta in M deve stare su un arco di M. Dunque esiste $u_2 \in M$, $u_2 \neq u_0,u_1$, adiacente a v_2. Prendiamo $S=\{u_0, u_1, u_2\}$. Avendosi $|\{u_0, u_1, u_2\}| = 3$ esiste un terzo nodo $v_3 \in V_2$ adiacente ad uno dei tre. Sia $v_3 \notin M$. Vi sono 3 casi:

- v_3 adiacente ad u_2. Nel cammino v_3,u_2,v_2,u_0 l'arco (u_2,v_2) appartiene all'accoppiamento M, gli altri due no. Si ottiene un nuovo accoppiamento M' togliendo ad M l'arco (u_2,v_2) e aggiungendo gli altri due, e si ha $|M'|=|M| + 1$.
- v_3 adiacente ad u_1. Nel cammino v_3,u_1,v_1,u_0, cambiando stato al primo e al terzo arco si ottiene $|M'|=|M| + 1$.
- v_3 adiacente ad u_0. Nel cammino v_3,u_0,v_1,u_1 si proceda come nei casi precedenti cambiando stato a (v_3,u_0).

Continuando in questo modo, per la finitezza del grafo, si arriva necessariamente ad un nodo v_r che non appartiene ad M. Ognuno dei nodi v_i è adiacente ad almeno uno tra $u_0,u_1,\ldots,u_{i-1}$. Come nel caso $r = 2$ si ha un cammino $v_r,u_s,v_s,u_t,v_t,\ldots,v_m,u_0$, nel quale glia archi $e_i=(u_i,v_i)$ appartengono ad M, mentre gli $e_j = (v_j,u_j)$ no. Costruiamo allora un nuovo accoppiamento M' togliendo da M gli e_i e aggiungendo gli e_j. Poiché i due archi esterni (v_r,u_s) e (v_m,u_0) sono in M', questo accoppiamento contiene un arco in più di M. **CVD**

Corollario. *Sia $G=(V_1, V_2, E)$ un grafo bipartito k-regolare con $|V_1|=|V_2|$. Allora G contiene k accoppiamenti completi.*

Dimostrazione. Sia S un sottinsieme di V_1. L'insieme *adj(S)* avrà al più $k|S|$ nodi (se ciascun nodo in *adj(S)* ha grado 1 nel sottografo indotto da $S \cup adj(S)$)

Pag 191

ed almeno *|S|* nodi (se ciascun nodo in *adj(S)* ha grado *k* nel sottografo indotto da $S \cup adj(S)$). In tutti i casi la condizione di Hall è verificata, e quindi esiste un accoppiamento completo, che può essere rimosso dal grafo dando luogo ad un nuovo grafo *(k-1)*-regolare. Per esso possiamo ripetere il ragionamento, giungendo all'asserto. **CVD**

9.5.3 Colorazione di grafi e grafi planari

Definizione. Dato un grafo *G*, esso è *k-colorabile* se ad ognuno dei suoi nodi è possibile assegnare uno di *k* colori (interi da 1 a *k*) in modo tale che due nodi adiacenti qualsiasi non abbiano lo stesso colore. Se *G* è *k*-colorabile ma non *(k-1)*-colorabile, diciamo che *k* è il numero cromatico di *G*, indicato con $\chi(G)$.

E' facile vedere che se il grafo è il completo su *n* nodi, il suo numero cromatico è $\chi(G)=n$, perciò è possibile costruire grafi con numero cromatico arbitrariamente grande. Invece, $\chi(G)=1$ se e solo se *G* è un grafo che non contiene archi, cioè è un grafo nullo. Infine, $\chi(G)=2$ se e solo se *G* è un grafo bipartito, e la sua bipartizione corrisponde ai nodi che possono essere colorati con ciascuno dei due colori; pertanto *G* ha numero cromatico 2 se e solo se non contiene cicli dispari. E' interessante notare che gli alberi sono grafi con numero cromatico uguale a 2.

Sfortunatamente, possiamo dire poco a riguardo del numero cromatico di un grafo arbitrario; se il grafo ha *n* nodi allora ovviamente $\chi(G) \leq n$, e se il grafo contiene come sottografo un grafo completo di *r* nodi, allora $\chi(G) \geq r$, ma queste limitazioni possono essere arbitrariamente lontane.

Teorema. *Se G è un grafo il cui grado massimo di un nodo è d, allora G è (d+1)-colorabile.*

Dimostrazione. Procediamo per induzione sul numero *n* di nodi di *G*.

Pag 192

L'affermazione è bnalmente vera se *n=1*, poiché il grafo non ha archi e basta un colore.

Sia *G* un grafo con *n* nodi; se rimuoviamo da esso uno dei suoi nodi con tutti i suoi archi incidenti otteniamo un grafo con *n-1* nodi che, per ipotesi induttiva, può essere *(d+1)*-colorato. Riaggiungendo al grafo il nodo precedentemente tolto, esso ha al più *d* vicini colorati con al più *d* colori diversi. Pertanto esiste sempre un ulteriore colore non usato con cui il nodo può essere colorato. **CVD**

Con un'analisi più approfondita questo teorema può essere rafforzato nel seguente risultato:

Teorema. *(di Brooks) Se G è un grafo di massimo grado d, allora G è d-colorabile a meno che non si verifichi una delle seguenti condizioni:*

(i) G ha il grafo completo con d+1 nodi come componente connessa
(ii) d=2 e G ha un ciclo di lunghezza dispari come componente.

Questi due teoremi sono utili se i gradi dei nodi sono approssimativamente uguali; d'altra parte, se il nostro grafo ha pochi nodi di grado alto, allora questi teoremi ci dicono poco: si pensi alla stella (un nodo connesso a *k* nodi di grado 1), che per il teorema di Brooks è *k*-colorabile, ma che in effetti è bipartito, quindi 2-colorabile.

Questa discrepanza tra limitazione superiore e numero cromatico effettivo è molto ridotta se restringiamo la nostra attenzione ai grafi planari.

Definizione. Un *grafo planare* è un grafo che può essere disegnato sul piano in modo che due archi qualsiasi (o meglio, le curve che li rappresentano) non si incrocino mai eccetto che in un nodo al quale siano entrambi incidenti.

In un grafo planare, oltre ai nodi e agli archi, rimangono individuate le *facce*, definite come segue.

Definizione. Un punto del piano *x* si dice *disgiunto da grafo planare G* se *x* non rappresenta né un nodo né un punto che giace su un arco di *G.* Dato un punto *x* disgiunto da *G*, definiamo la *faccia di G contenente x* come l'insieme dei punti disgiunti da *G* che possono essere raggiunti da *x* con una curva di Jordan i cui punti sono tutti disgiunti da *G*.

In alternativa, possiamo dire che due punti del piano *x* ed *y* sono in relazione se sono entrambi disgiunti da *G* e possono essere congiunti da una curva di Jordan i cui punti siano tutti disgiunti da *G*; questa è una relazione di equivalenza sui punti del piano disgiunti da *G*, e le corrispondenti classi di equivalenza sono chiamate *facce di G*.

Esattamente una delle facce di ogni grafo planare è illimitata, detta *faccia infinita.* In effetti la faccia infinita non ha niente di speciale rispetto alle altre facce; per convincersi di ciò è sufficiente riportare il disegno del grafo sulla superficie di una sfera e riproiettare il grafo sul piano tangente alla sfera nel polo sud da un altro punto: la faccia che contiene il punto di proiezione diventerà la faccia infinita.

Teorema. *(di Eulero) Sia G un grafo planare connesso con n nodi, m archi ed f facce. Allora:*

$$n+f=m+2.$$

Dimostrazione. Si proceda per induzione sul numero degli archi *m.*

Se *m=0*, allora *n=1* (perché *G* è connesso) ed *f=1* (l'unica faccia è quella infinita), e quindi l'asserto è vero.

Supponiamo ora il teorema vero se *G* ha *m-1* archi ed aggiungiamo un nuovo arco a *G.* Due possibilità possono verificarsi: (i) se il numero dei nodi rimane

Pag 194

immutato l'aggiunta di questo arco non può che portare allo spezzamento di una faccia in due nuove facce; (ii) se il numero dei nodi aumenta di 1, l'arco connette un nodo preesistente al nuovo nodo e il numero delle facce rimane immutato.

Detto *G'* il nuovo grafo, con *n'* nodi, *m'* archi ed *f'* facce, e ricordando che *m'=m+1,* si ha:

nel caso (i): *n'=n* ed *f'=f*+1, perciò: *n'+f'=n+f+1=m+2+1=m'+2;*

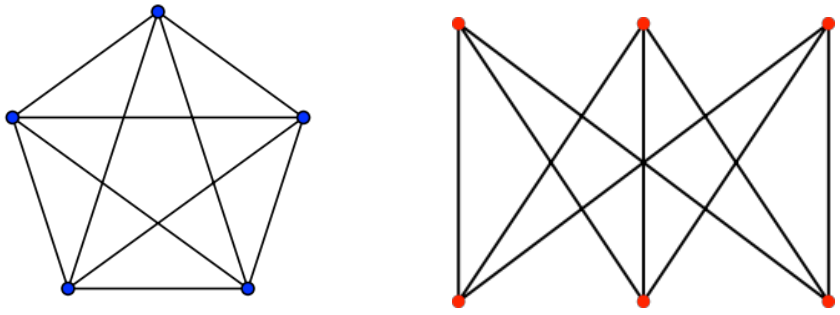
nel caso (ii): *n'=n+1* ed *f'=f*, perciò: *n'+f'=n+1+f=m+2+1=m'+2.*

In entrambi i casi il teorema è dimostrato. **CVD**

Corollario. *Se G è un grafo planare connesso con n nodi ed m archi, n≥3, allora m≤3n-6. Se G è anche bipartito, allora m≤2n-4.*

Dimostrazione. Poiché ogni faccia di *G* è limitata da almeno 3 spigoli *3f ≤2m*. Combinando questa disuguaglianza con il teorema di Eulero si ha il risultato. La disuguaglianza sui bipartiti si ottiene nello stesso modo ricordando che essi non contengono cicli dispari e quindi ogni faccia è limitata da almeno 4 archi. **CVD**

Corollario. I grafi K_5 e $K_{3,3}$ (rappresentati in figura) sono non planari.



Dimostrazione. Se per assurdo i due grafi fossero planari per essi dovrebbe valere il teorema di Eulero ed il suo corollario, mentre applicando le

disuguaglianze del corollario precedente a K_5 si ottiene la relazione assurda 10≤9. Applicando la disuguaglianza $4f \leq 2m$ a $K_{3,3}$ si ha $2f \leq 9$ mentre il teorema di Eulero ci dice che $f=5$. **CVD**

Tornando alla colorazione dei grafi, nel caso dei grafi planari, è possibile enunciare come teorema un risultato che per molti anni è stato una congettura:

Teorema. *(dei 4 colori) Ogni grafo planare è 4-colorabile.*