

Progettazione di Algoritmi

Marco Casu



Contents

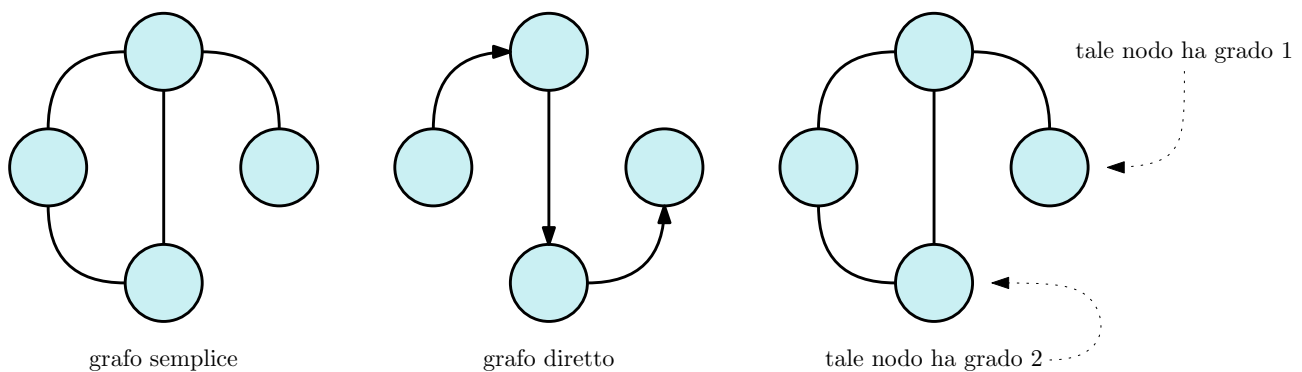
1	Grafi	3
1.1	Introduzione e Definizioni	3
1.2	Rappresentazione Fisica	4
1.3	Ricerca di un Ciclo	4

1 Grafi

1.1 Introduzione e Definizioni

Un grafo, è una coppia (V, E) , dove V è un insieme di *nodi o vertici*, ed E un insieme di archi che collegano i nodi. Un grafo è detto **semplice** se, per ogni coppia di nodi, essi sono collegati da al massimo un arco, e non esistono dei cicli su un singolo nodo. Nel corso ci occuperemo di *visitare* i grafi in profondità ed in ampiezza (concetti che verranno ripresi più in avanti).

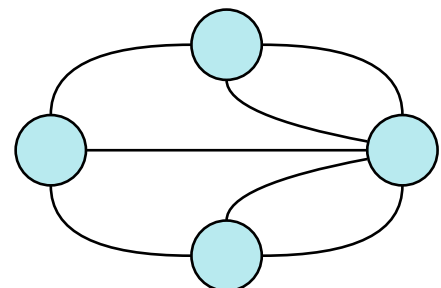
Un grafo, può vedere i suoi archi *orientati*, in questo caso si dice che il grafo è **diretto**. Due nodi sono **adiacenti** se collegati da un arco, ed il **grado** di un nodo non è altro che il numero di nodi adiacenti ad esso.



Esiste un problema classico dal 1700, noto come *problema dei ponti di Königsberg*, si consideri la seguente città posta nei pressi di un fiume che la divide in diversi settori, collegati da appositi ponti, rappresentata con il seguente grafo :



si rappresenta :



Ci si chiede se è possibile passeggiare per la città, visitando tutti i settori, senza passare per due volte sullo stesso ponte. Consideriamo il modello del grafo, una passeggiata su un grafo non è altro che una sequenza ordinata di vertici ed archi che si alternano, come : $v_0, e_1, v_1, \dots, e_k, v_k$. Esiste una passeggiata su questo grafo, ossia una sequenza che non vede ripetizioni degli archi?

Osservazione : Per visitare un nodo è necessario passare per due archi, uno entrante ed uno uscente. Se entriamo in un nodo di grado 3, resterà un arco non visitato, per visitarlo sarà necessario entrarvi nuovamente da tale arco, per poi uscire da un altro precedentemente già visitato (questo ovviamente se non si comincia la passeggiata dal nodo in questione).

Ci rende chiaro il seguente fatto : Se il grado di un nodo x è dispari, a meno che la passeggiata non inizi o finisca su x , uno dei suoi archi verrà attraversato più di una volta. *Eulero* studiò questo problema, si dice infatti che la passeggiata su un grafo è **euleriana** se non si passa 2 volte sulle stesso arco.

Si consideri però il seguente grafo :



Pur vedendo ognuno dei suoi nodi avere grado pari, ossia 2, tale grafo non permette alcuna passeggiata aleatoria, in quanto non è *connesso*.

Un grafo si dice **connesso** se, per ogni coppia di vertici, essi sono collegati da una passeggiata, ossia è possibile raggiungere un vertice partendo da un altro. Le precedenti osservazioni ci portano al seguente risultato.

Teorema (Eulero) : Un grafo ha una passeggiata euleriana se e solo se è connesso, ed esistono al massimo 2 vertici di grado dispari.

Il fatto che sono concessi 2 vertici di grado dispari, è dato dal fatto che essi saranno l'inizio e la fine della passeggiata.

1.2 Rappresentazione Fisica

Che struttura dati possiamo utilizzare per rappresentare un grafo? Vediamo due alternative :

- **Matrice di Adiacenza** - Utilizziamo una matrice $n \times n$, dove n è il numero di nodi del grafo. Nella posizione i, j ci sarà 1 se il vertice v_i è adiacente al vertice v_j , altrimenti 0. Il costo di "check" per l'adiacenza di due vertici è costante, basta consultare un'entrata della matrice, nonostante ciò, lo spazio che occupa tale rappresentazione è $O(n^2)$.
- **Liste di Adiacenza** - Ad ogni vertice del grafo è associata una lista, contenente tutti i suoi vertici adiacenti, per controllare se due vertici sono adiacenti, è necessario fare una ricerca lineare su tale lista, ed ha costo $O(deg(v))$, dove v è il vertice sulla quale si sta effettuando la ricerca, ed è ovviamente limitato da $n - 1$ (numero di vertici).

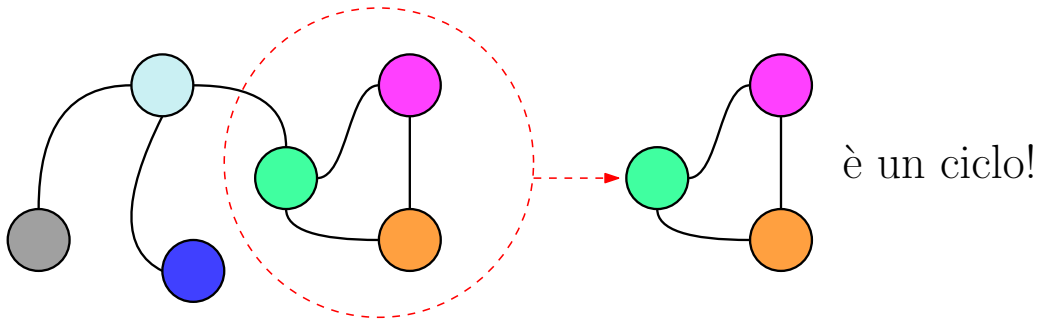
Le dimensioni della struttura dati sono $O(n + \sum_{v \in V(G)} deg(v))$.

Nel caso in cui un grafo dovesse vedere ogni vertice adiacente a tutti gli altri, la ricerca costerebbe $O(n)$ e le dimensioni sarebbero $O(n^2)$, ciò differisce però dal caso reale, la rappresentazione con liste di adiacenza risulta un buon compromesso fra costo computazionale e dimensioni. Sarà usuale denotare m il numero di archi e n il numero di vertici. Le liste di adiacenza occupano quindi spazio $O(n + m)$, si osservi inoltre la seguente identità :

$$\sum_{v \in V(G)} deg(v) = 2 \cdot m \text{ dove } m := |E|$$

1.3 Ricerca di un Ciclo

Definizione : Un *ciclo* in un grafo, non è altro che un *sottografo connesso* dove ogni vertice è di grado 2. Identifica un "cammino circolare", e la ricerca dei cicli nei grafi è un problema molto noto.



Consideriamo adesso un problema, vogliamo definire un algoritmo che, dato in input un grafo $G = (V, E)$, dove ogni vertice ha grado maggiore o uguale a 2, restituisca in output un qualsiasi ciclo presente nel grafo, mantenendo un costo computazionale $O(n + m) = O(|V| + |E|)$.

Si consideri la seguente *idea* informale di soluzione :

Ogni vertice ha almeno 2 nodi adiacenti, è quindi sempre possibile entrare in un vertice ed uscirne da un arco diverso da quello dalla quale si è entrati. Si parte da un qualsiasi vertice nel grafo, e si procede selezionando uno qualsiasi dei due nodi adiacenti successivi, almeno uno dei due non sarà quello dalla quale si è entrati, procederemo in questa maniera camminando in maniera casuale sul grafo, finchè non troveremo un nodo che è stato già visitato in precedenza, ciò indica che si è eseguito un cammino ciclico.

Utilizzeremo un vettore con lo scopo di salvare i nodi visitati, il ciclo sarà rappresentato dai nodi presenti nel vettore, partendo dall'ultimo elemento, continuando a ritroso fino a trovare il nodo identico all'ultimo. Si consideri il seguente esempio in cui gli archi sono contrassegnati dall'iterazione dell'algoritmo nella quale sono stati attraversati :



Una volta completato la ricerca del ciclo, elimineremo dal vettore tutti gli elementi a partire dal primo fino all'elemento antecedente a quello identico all'elemento finale.

Pseudocodice

Input : Un grafo $G = (V, E)$.

Output : I nodi di un sottografo di G che è un ciclo.

```

x = V[random] // Un vertice a caso
W=[x] // Inizializzo il vettore output
current = V
y=adiacente di x // Un adiacente a caso
next=y

```

```

while(next ∉ W){
    W.append(next)
    current=next
    if (1° adiacente di current≠W[W.lenght-2]){ // Il penultimo
        next = 1° adiacente di current
    }else{next = 2° adiacente di current
    }
}
while(W[0]≠next){
    W.remove(W[0]) // Rimuove il primo elemento
}
return W

```

Qual'è la complessità di tale algoritmo? Entrambi i cicli `while` eseguono $O(n)$ iterazioni, il fatto è che, nel primo ciclo while, il controllo `next ∉ W` deve scorrere comunque tutto il vettore, rendendo il costo dell'algoritmo $O(n^2)$, non rispettando le specifiche iniziali, ossia $O(n + m)$.