Progettazione di Algoritmi

Marco Casu

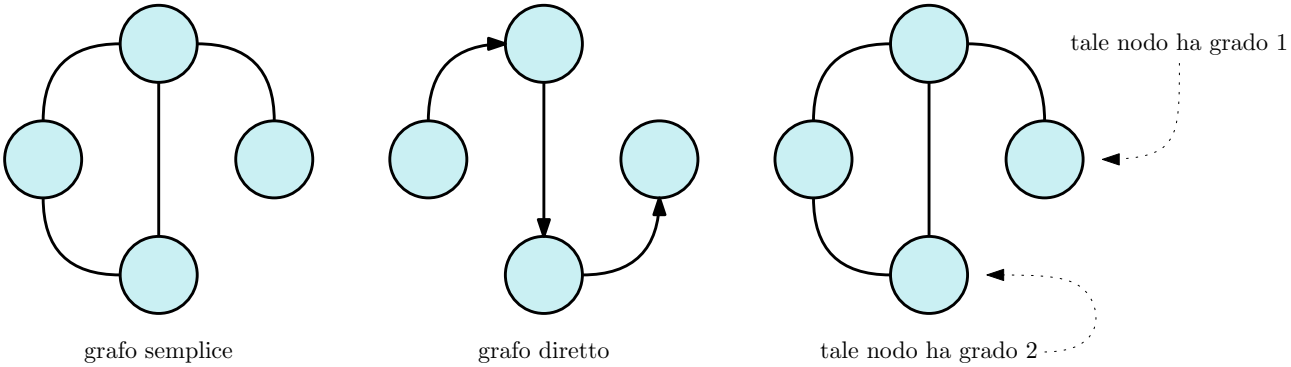
Contents

1 Grafi

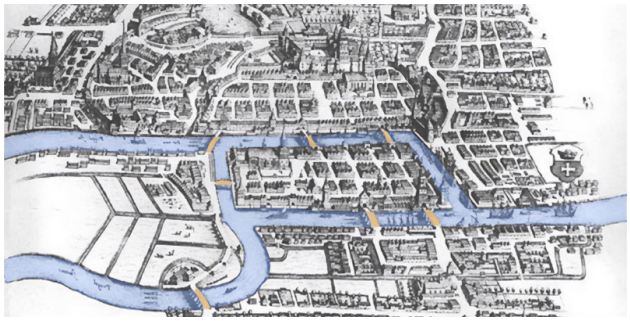
1.1 Introduzione e Definizioni

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).

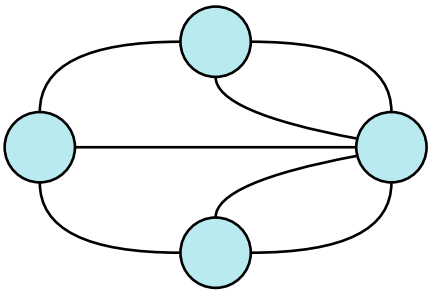
Un grafo, può vedere i suoi archi *orientati*, in questo caso si dice che il grafo è **diretto**. Due nodi sono **adiacenti** se collegati da un arco, ed il **grado** di un nodo non è altro che il numero di nodi adiacenti ad esso.



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 :

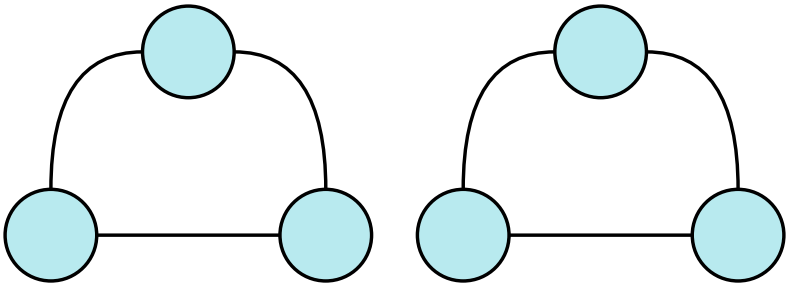


Ci si chiede se è possibile passeggiare per la città, visitando tutti i settori, senza passare per due volte sullo stesso ponte. Consideriamo il modello del grafo, una passeggiata su un grafo non è altro che una sequenza ordinata di vertici ed archi che si alternano, come : $v_0, e_1, v_1 \ldots, e_k,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

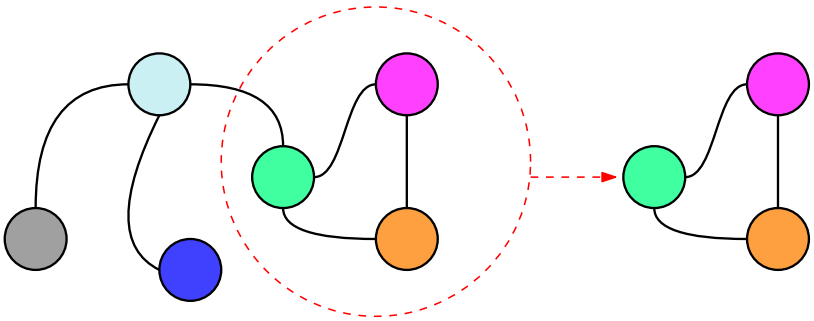
  Le dimensioni della struttura dati sono $O\big(n + \sum_{v \in V(G)} \deg(v)\big)$.

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 p 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.

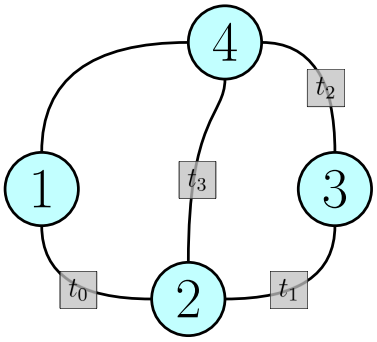


Consideriamo adesso un problema, vogliamo definire un algoritmo che, dato in input un grafo $G = (V, E)$, dove ogni vertice ha grado maggiore o uguale a 2, restituisca in output un qualsiasi ciclo presente nel grafo, mantenendo un costo computazionele $O(n + m) = O(|V| + |E)$.

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 :



Ha prodotto l'array $V = [1, \underline{2, 3, 4, 2}]$
è il ciclo

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.

```
CercaCiclo(graph G){
    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
}
```

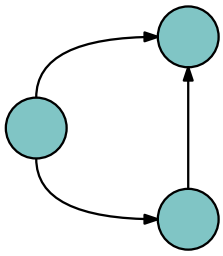
Qual'è la complessita di tale algoritmo? Entrambi i cicli `while` eseguono $O(n)$ iterazioni, il fatto è che, nel primo ciclo while, il controllo `next∉W` deve scorrere comunque tutto il vettore, rendendo il costo dell'algoritmo $O(n^2)$, non rispettando le specifiche iniziali, ossia $O(n+m)$.

1.4 Cammini sui Grafi

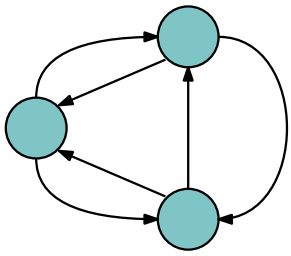
Un **cammino**, non è altro che una passeggiata su un grafo in cui non si passa mai più di una volta sullo stesso vertice, ossia una passeggiata senzza ripetizioni di vertici o archi.

Osservazione : Siano x ed y due nodi di un grafo, se esiste una passeggiata da x ad y, allore esiste anche un cammino.

Nei grafi diretti vale la stessa regola, con ovviamente il vincola che bisogna rispettare l'orientazione degli archi. Un grafo diretto si dice **fortemente connesso** se, per ogni coppia di vertici x, y, esiste un cammino da x ad y e viceversa.



non è fortemente connesso



è fortemente connesso

Un noto problema è il seguente, dato un grafo G e due veritici x, y, esiste un cammino da x ad y? In generale, il carico di lavoro per controllare ciò, equivale al carico di lavoro necessario per controllare tutti i nodi che possono essere "raggiunti" partendo da x.

Prendo quindi un vertici x e trovo tutti i vertici y per i quali esiste un cammino fra essi, per fare ciò, occorre **visitare** il grafo, e può essere fatto in due modi differenti.

1.4.1 Depth-First Search

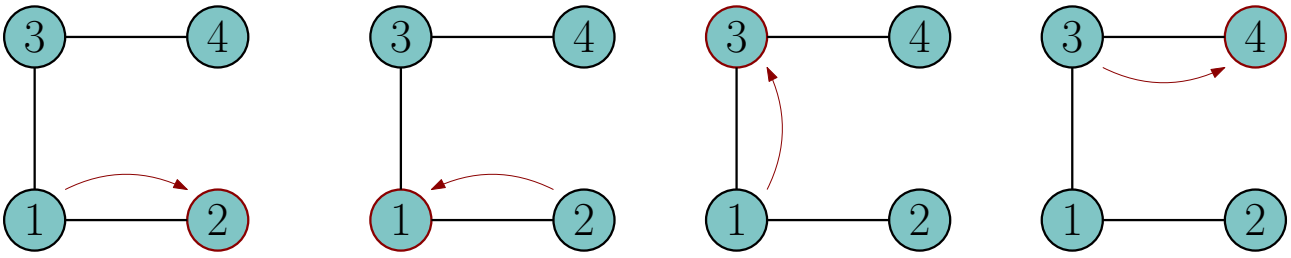
Abbreviato **DFS**, tale algoritmo rappresenta la visita su un grafo in *profondità*. Partendo da un qualsiasi vertice x, inizio a visitare randomicamente uno dei vertici adiacenti, per poi proseguire da esso. Se ad un certo punto non vi sono nuovi vertici da visitare, si esegue il cosiddetto *back tracking*, controllando i nodi a ritroso e cercando dei nuovi vertici. Risulta quindi naturale l'uso di uno *stack* per poter implementare tale ricerca. L'algoritmo alla fine visiterà ogni nodo per la quale esiste un cammino dal nodo iniziale.

Pseudocodice

`Input` : Un grafo $G = (V, E)$, ed un vertice x.
`Output` : L'insieme dei vertici visitati partendo da x.

```
DFS(graph G, vert x){
    S : stack = {x}
    Vis :  set = [x]     // l'insieme che conterrà l'output
    while(S≠ ∅){
        y=S.top()
        if(∃z adiacente ad y∧x≠Vis){
            Vis.add(z)
            S.push(z)
        }
        else{
            S.pop()
        }
    }
    return Vis
}
```

Esempio di applicazione (il nodo di partenza è il nodo 1) :



primo passo
Vis = [1,2]

secondo passo
Vis = [1,2]

terzo passo
Vis = [1,2,3]

quarto passo
Vis = [1,2,3,4]

L'output dell'algoritmo sarà proprio l'insieme `Vis`, contenente tutti i nodi raggiungibili dal vettore input, bisogna dimostrare che l'algoritmo sia corretto, mostrando che ogni vertice raggiungibile da x è in `Vis`.

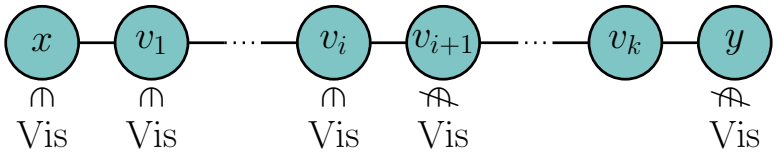
Dimostrazione : Supponiamo per assurdo che vi sia un vertice y tale che, esiste un cammino

da x ad y e che y non sia presente in Vis.

$$\exists y | x \to y \wedge y \notin \text{Vis}$$

Essendo x il vertice di partenza, esso sicuramente si troverà in Vis, per costruzione dell'algoritmo. Questo vuol dire che esiste un vertice nel cammino, per la quale vale la seguente proprietà :

Siano $v_1 \dots v_k$ vertici nel cammino $x \to y$, $\exists v_i | v_i \in \text{Vis} \wedge v_{i+1} \notin \text{Vis}$



Essendo v_i in Vis, vuol dire che ad un certo punto è stato nel top dello stack, ma v_{i+1} è adiacente a v_i, quindi da quest'ultimo l'algoritmo avrà selezionato ad un certo punto v_{i+1}, per poi proseguire da esso, per costruzione, sarà inserito in Vis, ma ciò è in contraddizione con l'ipotesi iniziale che y non è in Vis. ■

Questo algoritmo presenta un problema cruciale, non è efficiente, infatti risulta particolarmente pesante il controllo `if(∃z adiacente ad y∧x≠Vis)`, che ha costo computazionale $O(\deg(y)) + O(n)$. L'algoritmo va migliorato, al posto di un set, è possibile utilizzare un array nella seguente maniera : sarà composto da $n := |V|$ elementi inizializzato con tutti 0, si avrà che $array[i] = 1 \iff i$ fa parte dell'output.

Pseudocodice

```
DFS2(graph G, vert x){
    S : stack = {x}
    Vis :  int[n] = [0,0...0]     // L'array in questione
    while(S≠ ∅){
        y=S.top()
        if(Vis[y.adiacenti[0]]==0){     // Trova un adiacenta non ancora controllato
            z=y.adiacenti[0]
            Vis[z]=1
            S.push(z)
            y.adiacenti.remove(0)
        }
        else{
            y.adiacenti.remove(0)
        }
        if(y.adiacenti≠0){S.pop()}
    }
    return Vis
}
```

Si è nell'ipotesi in cui il grafo è implementato con le liste di adiacenza, infatti si noti come ogni vertice presenta il campo `adiacenti`. Per rendere più efficiente il tutto senza dover controllare ogni volta se un nodo è stato già visitato, semplicemente si rimuove dalla lista di adiacenza, ed ogni volta se ne prende il primo di tale lista che sicuramente non è stato ancora visitato, rendendo costante tale operazione.

Qual'è ora il costo computazionale? Quante volte viene eseguito il ciclo `while`? Rispondere a ciò risulta difficile, piuttosto ci si chiede quanto lavoro devo fare nel ciclo per ogni vertice? Per ognuno di essi, si esegue un numero limitato di volte il comando `S.top()`. Nello specifico, si esegue tante volte quanto è il grado del vertice, risulta naturale che la complessità finale sia :

$$O(n) + O\big(\sum_{v \in V(G)} \deg(v)\big) = O(n + |E|) = O(n + m) \text{ costo lineare}$$

Lo stesso algoritmo, si presta in maniera piuttosto naturale ad essere implementato in maniera ricorsiva, permettendo l'omissione dell'utilizzo di uno stack.

Pseudocodice

```
DFSRec(graph G, vert x,int[n] Vis){
    Vis[x]=1
    for each y∈x.adiacenti{     // per ogni adiacente di x
        if(Vis[y]==0){
            DFSRec(G,y,Vis)
        }
    }
}
```

Il ciclo `for each y∈x.adiacenti` considera ogni adiacente di x una volta sola, facendo lo stesso lavoro di "cancellazione" dei vicini già controllati, la complessità rimane la medesima.