Alcune applicazioni delle visite

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **22(b)** **[12/12/23]**

SAPIENZA
UNIVERSITÀ DI ROMA

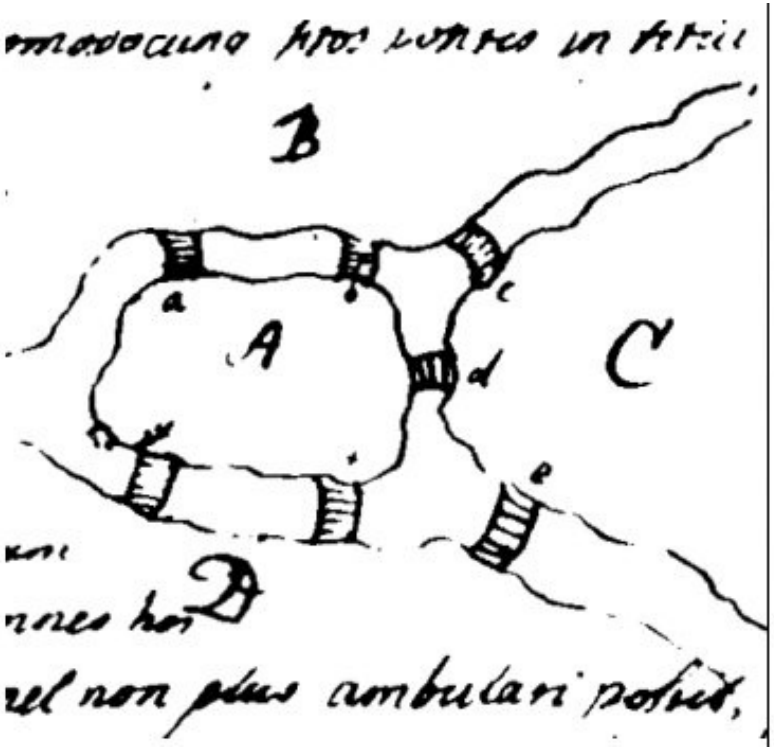
Visite: raccogliere informazioni

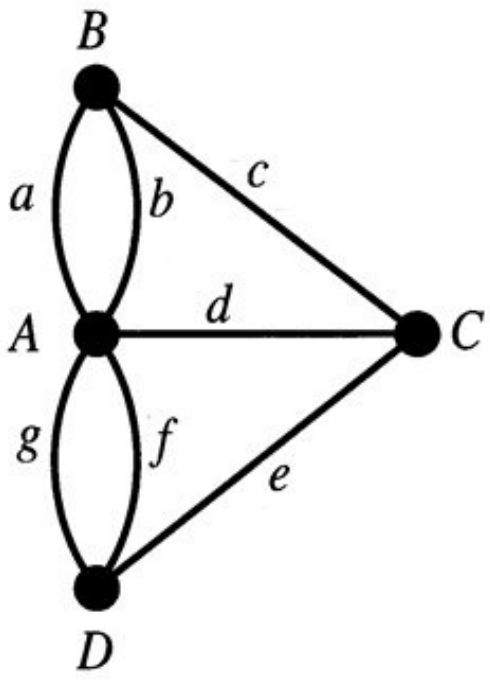
Durante una visita si possono **raccogliere informazioni** sui **nodi** (o più raramente sugli archi) attraverso **vettori indicizzati sui nodi** (o archi) come abbiamo fatto con i **tempi di ingresso/uscita**.

Queste informazioni permettono spesso di **risolvere un problema sui grafi con una** (o più) **visita**, ma senza dover ricalcolare più volte le stesse cose (**programmazione dinamica**).

Vediamo nel seguito diversi esempi di questo fatto.

Ovviamente, a **seconda del problema**, occorre capire quale sia l'**informazione giusta** da considerare.





distanze minime (numero di archi)

Cammino minimo da s a t

Il problema di trovare un (singolo) **cammino minimo** in **numero di archi** da s a t può **sembrare** un problema **più semplice** della **visita** di un intero grafo.

Tuttavia **non si può sapere** per **dove passi** tale cammino.

Quindi, **è necessario** fare una **BFS radicata in s** che si può arrestare non appena venga trovato t, tenendo **traccia delle distanze**.

Il modo più semplice è tenere un vettore d e assegnare $d[z] = d[v] + 1$ a tutti i vicini z di v.

Il nodo s da cui si inizia la visita ha distanza 0.

L'informazione nel vettore d **sussume l'informazione di marked**, se inizializziamo d a un valore distanza impossibile, al solito -1.

In tal modo, avranno **distanza negativa solo i nodi non ancora scoperti**, e quindi vale la proprietà:

$$marked[u] = \text{TRUE} \quad \text{se e solo se} \quad d[u] \geq 0$$

Cammino minimo da s a t

Nel codice sottostante, la visita si arresta **non appena viene trovato** il nodo di **arrivo** t.

Ovviamente **t potrebbe essere anche l'ultimo nodo da visitare**, quindi la complessità è la stessa di una visita, cioè $\Theta(n + m)$.

Ma cosa calcoliamo se lasciamo andare questa visita fino alla fine?

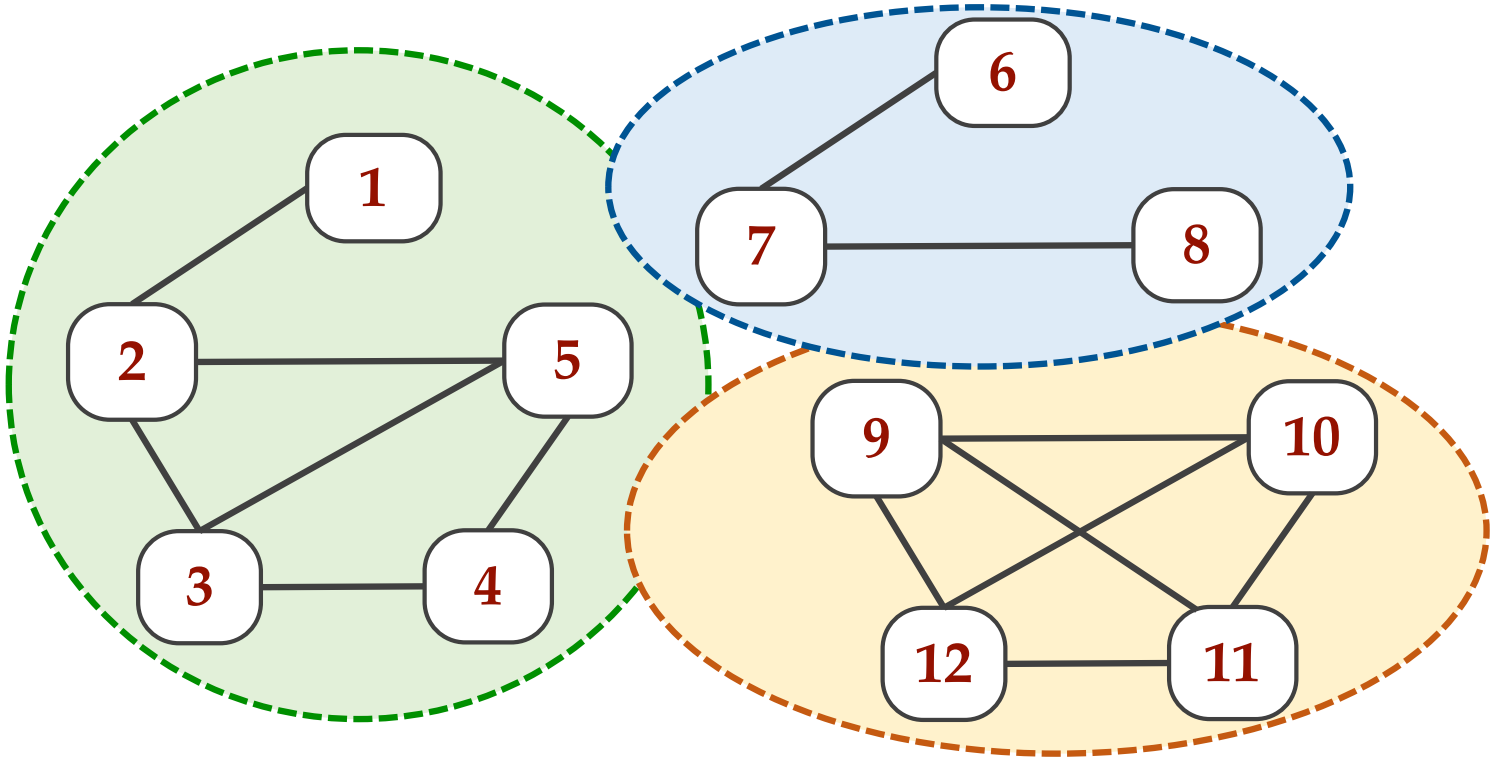
```
def dist(G, s, t):
    Q = newQueue()
    d = init(|G.V|, -1)
    d[s] = 0
    enqueue(Q, s)
    while not isEmpty(Q):
        u = dequeue(Q)
        forall v ∈ G.adj(u):
            if d[v] < 0:
                enqueue(Q, v)
                d[v] = d[u] + 1
                if v == t: return d[v]
    return d[t] # -1 se t non ragg.
```

Distanza minima da s a tutti i nodi

L'algoritmo precedente ci suggerisce che durante una **BFS** di un grafo G possiamo calcolare **le distanze dalla radice s** della visita a **tutti gli altri nodi** di G.

Infatti, se lasciamo completare la visita, il vettore d conterrà tutte **le distanze minime** da s a v, **per ogni v**, sempre al costo di $\Theta(n + m)$.

```
def dist(G, s):
    Q = newQueue()
    d = init(|G.V|, -1)
    d[s] = 0
    enqueue(Q, s)
    while not isEmpty(Q):
        u = dequeue(Q)
        forall v ∈ G.adj(u):
            if d[v] < 0:
                enqueue(Q, v)
                d[v] = d[u] + 1
    return d # vettore distanze
```



Componenti Connesse

Componenti Connesse

Le componenti connesse si calcolano **decorando ciascun nodo** con un **numero** che **rappresenta la componente connessa** a cui appartiene.

Teniamo un **vettore *cc* indicizzato sui nodi**, inizializzato a tutti 0. Per tutti i nodi u scoperti nella visita della prima componente connessa, assegneremo 1 a $cc[u]$. Poi troveremo il primo nodo tale che $cc[v]=0$ e faremo partire una nuova visita da v.

È **indifferente in questo problema** usare una **BFS oppure una DFS**. Tutte le operazione necessarie hanno un costo dominato dal costo $\Theta(n + m)$ della visita.

```
def componentiConnesse(G):
    cc = allocZ(numNodi[G], 0)
    c = 1
    for v = 1 to numNodi[G]:
        if cc[v] == 0:
            dfsCC(G, v, cc, c)
            c = c + 1
    return cc, c
```

```
def dfsCC(G, s, cc, c):
    cc[u] = c
    forall v ∈ G.adj(s):
        if cc[v] == 0:
            cc[v] = c
            dfsCC(G, v, cc, c)
```

Liste di nodi delle CC

Osservate che **per trovare il prossimo nodo** da cui cominciare la visita di una nuova componente connessa, durante la visita **si scorre il vettore *cc* una sola volta da sinistra a destra**, con un costo complessivo (**analisi aggregata**) di $\Theta(n)$ per questa operazione.

Possiamo generare un vettore di liste *lcc* tale che *lcc*[1] è la lista dei nodi della prima componente connessa, *lcc*[2] la lista dei nodi della seconda e così via…

Il vettore *lcc* può essere anche caricato scorrendo una sola volta *cc* alla fine della visita, quindi con un costo $\Theta(n)$.

```
def listaCC(G):
    c, cc = componentiConnesse(G)
    lcc = allocaV(c)
    for i=1 to n:
        lcc[cc[i]] = cons(i, lcc[cc[i]])
    return lcc
```

Cicli

Il nodo s appartiene a un ciclo?

La presenza di un **arco di attraversamento** (nella BFS) o di un **arco all'indietro** (in una DFS) **manifestano sempre la presenza di un ciclo** (del resto si tratta di un arco da aggiungere a un albero…).

Questo fatto può essere usato facilmente per **determinare se un grafo sia aciclico o meno** in un grafo non orientato.

Nei grafi orientati, la questione è leggermente più complessa.

O se sia un albero (in questo caso dobbiamo verificare anche che sia connesso).

Tuttavia, **non è banale** (anche se non difficile) **ricostruire il ciclo** da **un arco di attraversamento di una BFS** (esercizio).

Di conseguenza, se ci chiediamo se uno specifico nodo *s* appartenga a un ciclo, qual è la cosa più comoda?

Fare una **DFS radicata in *s*** e verificare se ci sia **un arco all'indietro che arriva in *s***.

Ciclo contenente s: pseudocodice

Scriviamo il codice che è ancora una volta una variazione della DFS. È necessario trasportare tra i parametri il nodo *s* per riconoscerlo, eventualmente, quando lo trovo con un arco all'indietro.

Ma se volessi il ciclo come lista di nodi?

La costruisco "al ritorno".

```
# prima chiamata
def sInCiclo(G, s):
    VIS = ∅
    return dfsCiclo(G, s, s, VIS)
```

```
def dfsCiclo(G, s, u, VIS):
    forall v ∈ adj(u):
        if v == s: return True, NULL
        if v ∉ VIS
            VIS = VIS ∪ {v}
            b, L = dfsCiclo(G, s, v, VIS)
            if b: return True, cons(v, L)
            # sospendo la ricerca
            # costruisco il ciclo all'indietro
    return False, NULL
```