Progettazione di Algoritmi

Simone Lidonnici

12 luglio 2024

Indice

1 Teoria dei grafi **3**
- 1.1 Tipi di grafi . . . 3
  - 1.1.1 Grafi diretti e non diretti . . . 3
  - 1.1.2 Passeggiate e cammini . . . 3
  - 1.1.3 Grafi connessi e fortemente connessi . . . 4
  - 1.1.4 Grafi ciclici . . . 4
- 1.2 Rappresentare un grafo . . . 5
  - 1.2.1 Matrici di adiacenza . . . 5
  - 1.2.2 Liste di adiacenza . . . 5
- 1.3 Trovare il ciclo in un grafo . . . 6
- 1.4 DFS (Ricerca in profondità) . . . 7
  - 1.4.1 DFS ottimizzata . . . 8
  - 1.4.2 DFS ricorsiva . . . 9
  - 1.4.3 DFS in grafi diretti . . . 9
  - 1.4.4 Componenti e DFS con componenti . . . 10
- 1.5 Ordinare un grafo . . . 10
  - 1.5.1 Trovare l'ordine topologico in grafi diretti . . . 11
  - 1.5.2 Trovare l'ordine topologico in grafi non diretti . . . 12
- 1.6 Intervalli di visita e tipi di archi . . . 13
  - 1.6.1 Tipi di archi . . . 14
  - 1.6.2 Algoritmo per controllare i tipi di archi . . . 15
- 1.7 Alberi di visita e cicli . . . 16
  - 1.7.1 Grafi non diretti . . . 16
  - 1.7.2 Grafi diretti . . . 16
  - 1.7.3 Vettore dei padri . . . 17
- 1.8 Ponti . . . 18
  - 1.8.1 Algoritmo per trovare i ponti . . . 19
- 1.9 Componenti fortemente connessi . . . 19
  - 1.9.1 Contrazione di un componente . . . 20
  - 1.9.2 Algoritmo per trovare i componenti . . . 21
  - 1.9.3 Algoritmo di Tarjan . . . 22
- 1.10 BFS (Ricerca in ampiezza) . . . 24
  - 1.10.1 Algoritmo della BFS . . . 24
  - 1.10.2 Distanza fra insiemi di nodi . . . 25
  - 1.10.3 Grafi pesati . . . 26
  - 1.10.4 Calcolare distanze pesate (Dijkstra) . . . 26

2 Algoritmi Greedy **28**
- 2.1 Alberi di copertura . . . 30
  - 2.1.1 Algoritmo di Kruskal . . . 30
- 2.2 Grafi con pesi negativi . . . 31
- 2.3 Algoritmo di Prim . . . 31

3 Algoritmi divide et impera **33**
3.1 Teorema principale 33
3.2 Esercizi divide et impera 34
3.2.1 Sottoarray di somma massima 34
3.2.2 Valore singolo in un array 35
3.3 Elemento maggioritario in un array 36

4 Programmazione dinamica **37**
4.1 Esercizi di programmazione dinamica 38
4.1.1 Ottimizzare lo spazio su un disco 38
4.1.2 Cammini colorati su una scacchiera 39
4.1.3 Ottimizzare il peso in uno zaino 40

1

Teoria dei grafi

Definizione di Grafo

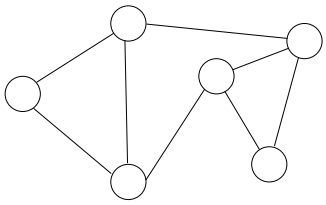
Un **grafo** G è una coppia (V, E) in cui V è un insieme di nodi e E un insieme di archi che collegano due nodi. Un grafo si dice **semplice** se:

- Non ha cappi, cioè nessun nodo è collegato con se stesso
- Ogni coppia di nodi è collegata da massimo un arco

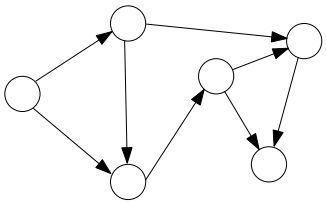
1.1 Tipi di grafi

1.1.1 Grafi diretti e non diretti

I grafi possono essere di due tipologie in base a se gli archi sono **orientati**, cioè partono da un nodo e arrivano ad un altro senza essere percorribili al contrario. Se il grafo ha archi orientati si dice **diretto**.



Grafo non diretto



Grafo diretto

1.1.2 Passeggiate e cammini

Nodi adiacenti

Due nodi collegati da un arco si dicono **adiacenti** (o vicini) e l'arco che li collega viene detto incidente. Per indicare che due nodi sono adiacenti scriviamo $x \backsim y$.
Si definisce il grado di un nodo $\deg(x)$ come il nomero dei suoi nodi adiacenti, uguale al numero di archi incidenti.

Definizione di passeggiata

Una **passeggiata** su un grafo è una sequenza di archi e nodi:

$$v_0 e_1 v_1 e_2 \dots e_n v_n$$

In cui ogni arco e_i collega il nodo v_{i-1} al nodo v_i.
Un **cammino** è una passeggiata in cui non si ripetono i nodi.

1.1.3 Grafi connessi e fortemente connessi

Definizione di grafo connesso

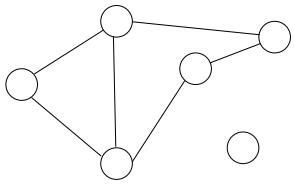
Un grafo G si dice **connesso** se per qualsiasi coppia di nodi esiste un cammino che li collega:

$$\forall v_i, v_j \in V(G) \exists \text{cammino} | v_1 \to v_j \vee v_j \to v_i$$

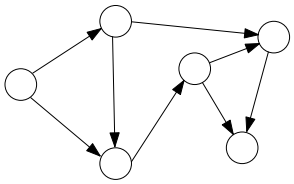
Un grafo G si dice **fortemente connesso** se per qualsiasi coppia di nodi esiste un cammino che li collega partendo da entrambi i nodi:

$$\forall v_i, v_j \in V(G) \exists \text{cammino} | v_1 \to v_j \wedge v_j \to v_i$$

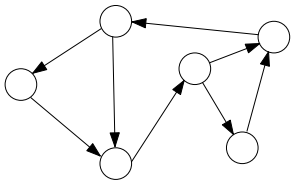
Nel caso di grafi non diretti ogni grafo connesso è anche fortemente connesso.



Grafo non connesso



Grafo connesso



Grafo fortemente connesso

Esiste un tipo specifico di passeggiata detta **passeggiata Euleriana** in cui si attraversano tutti i nodi una sola volta. Può esistere una passeggiata Euleriana in un grafo solo se il grafo è connesso e ci sono al massimo 2 nodi con grado dispari, che saranno inizio e fine.

1.1.4 Grafi ciclici

Definizione di grafo ciclico

Un grafo G è ciclico se esiste un sottogrppo connesso in cui ogni vertice ha grado ≥ 2.
Se nel grafo tutti i vertici hanno grado ≥ 2 allora il grafo è sicuramente ciclico.

$$\forall v \in V(G) \deg(v) \geq 2 \implies G \text{ ciclico}$$

In un grafo diretto se ogni nodo ha almeno un arco uscente allora il grafo è ciclico.

1.2 Rappresentare un grafo

1.2.1 Matrici di adiacenza

I grafi possono essere rappresentati con delle matrici di adiacenza in cui se v_i è adiacente a v_j la matrice conterrà 1 nella posizione (i, j) e nella posizione (j, i):

	v_1	...	v_i	...	v_j	...	v_n
v_1	0						
...		0					
v_i			0		1		
...				0			
v_j			1		0		
...						0	
v_n							0

Costo per controllare se x è vicino di y: $O(1)$
Spazio necessario per l'archiviazione: $O(n^2)$
Nel caso di grafi diretti la matrice conterrà 1 nella posizione (i, j) se l'arco parte da i e arriva a j (non sarà più simmetrica).

1.2.2 Liste di adiacenza

Per rappresentare i grafi si può anche usare una lista di adiacenza in cui ogni nodo ha una lista contenente tutti i suoi vicini:

$$v_1.\text{neighbors} = [\dots]$$
$$\dots$$
$$v_n.\text{neighbors} = [\dots]$$

Nel caso di un grafo diretto, ogni nodo avrà due liste:

- v_i.neighbors_out che contiene i nodi collegati da archi uscenti da v_i
- v_i.neighbors_in che contiene i nodi collegati da archi entranti in v_i

Costo per controllare se x è vicino di y: $O(n)$
Spazio necessario per l'archiviazione: $O(n^2)$
Lunghezza della lista di vicini di un determinato nodo v_i: $\deg(v_i)$
Grandezza totale delle liste: $O(n) + O(\sum_{i=1}^{n} \deg(v_i)) = O(n + m)$

1.3 Trovare il ciclo in un grafo

Dato un grafo G in cui ogni vertice ha grado ≥ 2, l'algoritmo per trovare il ciclo:

Algoritmo: Ricerca di un ciclo in un grafo G

Input:

- G: grafo

Output:

- C: nodi che formano il ciclo

```
def FindCiclo(G):
    x=V[0]
    C=[x]
    current=x
    next=x.neighbors[0]
    while next not in C : // finchè non trovo un nodo già visitato
        C.append(next)
        current=next
        if current.neighbors[0]!=C[-2] :
            next=current.neighbors[0]
        else :
            next=current.neighbors[1]
    while C[0]!=next :
        C.pop(0)
    return C
```

1.4 DFS (Ricerca in profondità)

La **DFS** (Depth first search) è un modo per visitare un grafo che consiste nel partire da un nodo e spostarsi in un vicino casuale non ancora visitato e nel caso tutti i vicini di un nodo siano già stati visitati ritornare al nodo precedente. Per implementare questo roll-back si utilizza uno Stack. L'algoritmo ritorna tutti i nodi visitabili dal nodo di partenza, quindi nel caso di grafo non connesso, ritornerà solo i vertici nel sottografo contenente il nodo di partenza.

Algoritmo: DFS

Input:

- G: grafo
- x: nodo di partenza

```
def DFS(G, x):
    Vis=set(x)
    Stack S=[x]
    while len(S)!=0 :
        y=S.top()
        if ∃ z in y.neighbors | z ∉ Vis :  // O(deg(y) · n)
            Vis.add(z)
            S.push(z)
        else :
            S.pop()
    return Vis
```

Dimostrazione per assurdo:
Supponiamo esista $y | \exists$cammino $x \to y$ ma $y \notin$ Vis e sia i un indice per cui $v_i \in \text{Vis} \wedge v_{i+1} \notin \text{Vis}$.
$v_i \in \text{Vis} \implies \begin{cases} v_i \text{ è stato inserito in } S \\ v_i \text{ è stato tolto da } S \end{cases} \implies$ ogni vicino di v_i è stato inserito in Vis $\implies v_{i+1}$ è stato inserito in Vis

1.4.1 DFS ottimizzata

L'algoritmo di base della DFS è poco ottimizzato per via del costo dell'if che richiede $O(\deg(y) \cdot n)$, per ottimizzarlo si cambia la struttura di Vis rendendolo un array lungo n in cui:

$$Vis[v] = \begin{cases} 0 & v \text{ non è stato visitato} \\ 1 & v \text{ è stato visitato} \end{cases}$$

Con questo cambiamento l'algoritmo diventa:

Algoritmo: DFS ottimizzata

Input:

- G: grafo
- x: nodo di partenza

```
def DFS_ott(G, x):
    Vis[x]=1
    Stack S=[x]
    while len(S)!=0 :
        y=S.top()
        if Vis[y.neighbors[0]]==1 :
            z=y.neighbors[0]
            Vis[z]=1
            S.push(z)
        y.neighbors.remove(0)
        if len(y.neighbors==0) :
            S.pop()
    return Vis
```

Avendo tutto costo $O(1)$ tranne il ciclo while con costo $O(n+m)$, l'algoritmo ha costo complessivo $O(n+m)$.

1.4.2 DFS ricorsiva

Della DFS si può fare anche una versione ricorsiva:

Algoritmo: DFS ricorsiva

Input:

- G: grafo
- x: nodo di partenza
- Vis: array nodi visitati

```
def DFS_ric(G, x):
    Vis[x]=1
    for y in x.neighbors :
        if Vis[y]==0 :
            DFS_ric(G, y, Vis)
    return Vis
```

Il costo di questo algoritmo è $O(n + m)$.

1.4.3 DFS in grafi diretti

Nel caso di grafi diretti bisogna cambiare l'algoritmo per controllare solo gli archi uscenti e non quelli entranti quando si cambia nodo:

Algoritmo: DFS

Input:

- G: grafo
- x: nodo di partenza

```
def DFS_dir(G, x):
    Vis[x]=1
    Stack S=[x]
    while len(S)!=0 :
        y=S.top()
        if ∃ z in y.neighbors_out | Vis[z]==0 :
            Vis[z]=1
            S.push(z)
        else :
            S.pop()
    return Vis
```

1.4.4 Componenti e DFS con componenti

Definizione di componente

Un **componente** è l'insieme di nodi di un sottografo connesso, però non connesso al resto del grafo.

$$Comp[x] = \text{nodi nello stesso componente che contiene } x$$
$$Comp[x] = Comp[y] \iff x, y \text{ appartengono allo stesso sottografo}$$

L'algoritmo che visita tutti i componenti è una modifica della DFS ricorsiva in cui:

$$Comp[v] = \begin{cases} 0 & v \text{ non è ancora stato visitato} \\ i & v\text{è nel componente } i \end{cases}$$

Si aggiunge inoltre una funzione per cambiare componente in cui si trova il nodo corrente:

Algoritmo: DFS per trovare componenti

Input:
- G: grafo

```
def CComp(G):
    comp_count=0
    for x in V :
        if Comp[x]==0 :
            comp_count+=1
            DFS_ric_comp(G, x, Comp, comp_count)
    return Comp


def DFS_ric_comp(G, x, Comp, comp_count):
    Comp[x]=comp_count
    for y in x.neighbors :
        if Comp[y]==0 :
            DFS_ric_comp(G, y, Comp, comp_count)
    return Comp
```

1.5 Ordinare un grafo

Un grafo diretto G ha un **ordine topologico** se esiste un ordine per cui ogni nodo ha archi uscenti che vanno solo verso nodi successivi nell'ordine e archi entranti solo da nodi precedenti nell'ordine. Inoltre:

$$G \text{ ciclico} \iff \nexists \text{ ordine topologico}$$

Corollario:

$$G \text{ non ciclico} \implies \exists v \in V | v \text{ non ha archi uscenti}$$

1.5.1 Trovare l'ordine topologico in grafi diretti

Per trovare l'ordine topologico in grafi diretti si usa un'algoritmo:

Algoritmo: DFS per trovare l'ordine topologico in grafi diretti

Input:
- G: grafo

```
def DFS_ord(G):
    l=[]
    while len(G)!=0 :  // O(n)
        x=no_archi(G)
        l.insert(x,0)
        elimina(x)
    return l


def no_archi(G): // O(n)
    for v in V :
        if len(v.neighbors_out)==0 :
            return v


def elimina(x):  // O(m)
    for e in E :
        if x in e :
            E.remove(e)
```

Il ciclo while esegue n volte le funzioni no_archi e elimina, quindi il costo dell'algoritmo sarà: $O(n(n+m))$

1.5.2 Trovare l'ordine topologico in grafi non diretti

Per trovare l'ordine topologico in grafi non diretti si usa un'algoritmo:

Algoritmo: DFS per trovare l'ordine topologico in grafi non diretti

Input:

- G: grafo

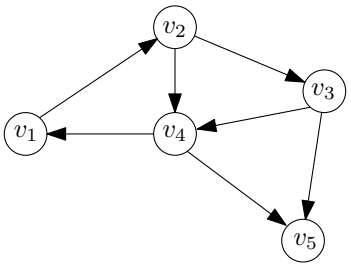
```
def ord_top(G):
    L=[]
    for v in V :
        if Vis[v]==0 :
            DFS_ord(G, v, Vis, L)
    return L


def DFS_ord(G, v, Vis, L):
    Vis[v]=1
    for w in v.neighbors :
        if Vis[w]==0 :
            DFS_ord(G, w, Vis, L)
    L.insert(v,0)
```

1.6 Intervalli di visita e tipi di archi

Dato un grafo G aggiungiamo un contatore C alla DFS, che parte da 1 e viene aumentato di uno ogni volta che si visita un nodo nuovo.
Ad ogni nodo $v \in V$ associamo:

- $t(v)$: valore di C quando v viene visitato per la prima volta
- $T(v)$: valore di C quando v viene rimosso dallo Stack
- $\text{Int}(v) = [t(v), T(v)]$

Esempio:



Una possibile tabella contenente gli intervalli usando una DFS partendo da v_1 è:

v	$t(v)$	$T(v)$
v_1	1	5
v_2	2	5
v_3	3	5
v_4	5	5
v_5	4	4

Dalla tabella e dal grafico possiamo osservare che:

- $t(v_i) \neq t(v_j)\ \forall i, j$
- $t(v_i) \leq T(v_i)$
- $t(v_i) = T(v_i) \iff v_i$ non ha archi uscenti e non è radice
- v_i radice $\iff \text{Int}(v_i) = [1, n]$ con G che ha n nodi

Inoltre confrontando gli intervalli tra due nodi v_1 e v_2 ci sono 3 possibilità:

- $\text{Int}(v_1) \subset \text{Int}(v_2)$
- $\text{Int}(v_1) \supset \text{Int}(v_2)$
- $\text{Int}(v_1) \cap \text{Int}(v_2) = \emptyset$

1.6.1 Tipi di archi

Albero di visita

Un **albero di visita** è un sottografo connesso e aciclico composto solo dagli archi che sono stati usati per raggiungere i vertici visitati. Nel caso di grafi diretti viene detto **arborescenza** ed è un'albero con tutti gli archi orientati dalla radice verso le foglie.

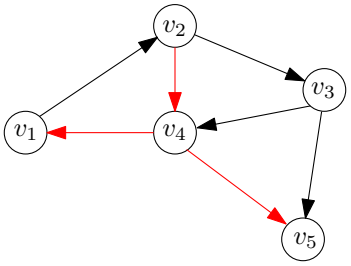
Preso un'arborescenza A creata tramite una DFS su un grafo G, ogni arco $(v_i, v_j) \in E$ non in A può essere classificato in 3 categporie:

1. **Arco all'indietro**: se va da un discendente ad un antenato, cioè $\text{Int}(v_i) \subset \text{Int}(v_j)$
2. **Arco in avanti**: se va da un antenato a un discendente, cioè $\text{Int}(v_i) \supset \text{Int}(v_j)$
3. **Arco di attraversamento**: se i due nodi non hanno correlazioni, cioè $\text{Int}(v_i) \cap \text{Int}(v_j) = \emptyset$

Nei grafi non diretti non essendoci differenza tra gli archi (v_i, v_j) e (v_j, v_i), l'unico caso possibile è che sia un arco all'indietro perché:

$$t(v_i) < t(v_j) \implies \text{Int}(v_i) \subset \text{Int}(v_j)$$

Esempio:



Gli archi non presenti nell'arborescenza A sono (v_2, v_4), (v_4, v_1) e (v_4, v_5).
Questi archi sono classificati:

- (v_2, v_4) è in avanti perchè [2,5]⊃[4,5]
- (v_4, v_1) è indietro perchè [5,5]⊃[1,5]
- (v_4, v_5) è di attraversamento perchè [5,5]∩[4,4]= ∅

1.6.2 Algoritmo per controllare i tipi di archi

Per controllare i tipi di archi usiamo un'algoritmo modificato della DFS che da in output 3 insiemi *Back*, *Forward* e *Cross* che contengono rispettivamente gli archi appartenenti alle tre categorie. Aggiungo un contatore C e anche due array t e T in cui segno gli intervalli dei vari nodi.

Algoritmo: DFS per classificare gli archi

Input:

- G: grafo
- x: nodo di partenza

```
def DFS_archi(G, x):
    C=0
    Vis[x]=1
    t[x]=1
    Stack S=[x]
    while len(S)!=0 :
        y=S.top()
        while len(y.neighbors_out)!=0 :
            z=y.neighbors_out[0]
            y.neighbors_out.remove(0)
            if Vis[z]==0 :
                C+=1
                t[z]=C
                Vis[z]=1
                S.push(z)
                break
            if t[z]<t[y] and T[z]==0 :
                Back.add((y,z))
            elif t[z]<t[y] and T[z]!=0 :
                Cross.add((y,z))
            else :
                Forward.add((y,z))
        if y==S.top() :
            S.pop()
            T[y]=C
    return Back, Cross, Forward
```

1.7 Alberi di visita e cicli

1.7.1 Grafi non diretti

Dato un grafo non diretto G connesso con un albero di visita T generato da una DFS, allora:

$$\exists \text{ arco all'indietro} \iff G \text{ ciclico}$$

1.7.2 Grafi diretti

Dato un grafo diretto G con un'arborescenza T generata da una DFS, definiamo che:

- un nodo u è discendente di un altro nodo v se esiste un cammino $v \to u$, cioè $\text{Int}(u) \subseteq \text{Int}(v)$
- un nodo v è antenato di un altro nodo u se un arco (u, v) è un arco all'indietro. Gli antenati di u sono tutti i nodi nel cammino radice $\to u$

Anche in questo caso:

$$\exists \text{ arco all'indietro} \iff G \text{ ciclico}$$

Esempio:
Un pozzo universale è un nodo x per cui:

- $\nexists (x, y) \in E \; \forall y \in V(G)$
- $\exists (y, x) \in E \; \forall y \in V(G)$

Scrivere un algoritmo con costo $O(n)$ per stabilire se esiste un pozzo universale avendo in input il grafo come matrice di adiacenza. La matrice se ci fosse un pozzo x sarebbe:

	v_1	...	x	...	v_n
v_1	0		1		
...		0	1		
x	0	0	0	0	0
...			1	0	
v_n			1		0

Il codice dell'algoritmo:

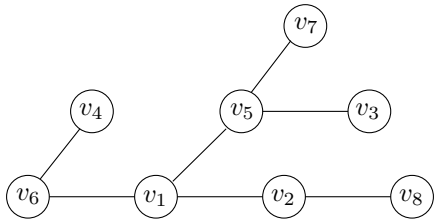
Algoritmo: Ricerca di un pozzo

Input:
- M: matrice di adiacenza del grafo

```
def SearchPozzo(M):
    pozzo=1
    for i in range(2,n) :
        if M[pozzo][i]==1 :
            pozzo=i
    for i in range(n) :
        if M[pozzo][i]==1 :
            return False
        if M[i][pozzo]==0 and i!=pozzo :
            return False
    return pozzo
```

1.7.3 Vettore dei padri

Un modo di salvare un albero di visita è il vettore dei padri, cioè un vettore P in cui $P[v] =$ nodo tramite cui si è arrivati a v. Per la radice $P[v] = v$.
Esempio:



n questo caso partendo da v_6 il vettore dei padri sarebbe:

$$P = [6, 1, 5, 6, 1, 6, 5, 2]$$

Per trovare gli antenati di un nodo v si può usare un algoritmo con costo $O(n)$:

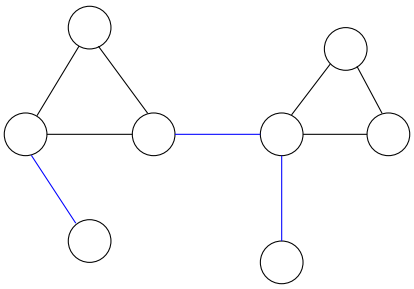
Algoritmo: Trovare gli antenati di un nodo v

```
def Ant(P, v):
    A.add(v)
    while P[v]!=v :
        A.add(P[v])
        v=P[v]
    return A
```

1.8 Ponti

Definizione di ponte

Dato un grafo non diretto G, si dice **ponte** un arco che se tolto fa diventare il grafo non connesso:



Per controllare se un determinato arco (u, v) è un ponte lo elimino e controllo se esiste un altro cammino $u \to v$:

- esiste $\implies$ (u, v) non ponte
- non esiste $\implies$ (u, v) ponte

Se volessimo trovare tutti i ponti in un grafo controllando ogni arco il costo computazionale sarebbe $O(m(n + m))$.

Dato T l'albero di visita di una DFS su un grafo G:

$$(u, v) \text{ ponte} \iff \nexists \text{ arco all'indietro da } T_v \text{ a fuori } T_v$$

Dove T_v è l'insieme dei discendenti di v.

1.8.1 Algoritmo per trovare i ponti

Dato un grafo G per trovare tutti i ponti si usa un'algoritmo che tiene segnato con $Back[v]$ il punto più indietro che si può raggiungere da un determinato nodo v:

Algoritmo: DFS per trovare i ponti

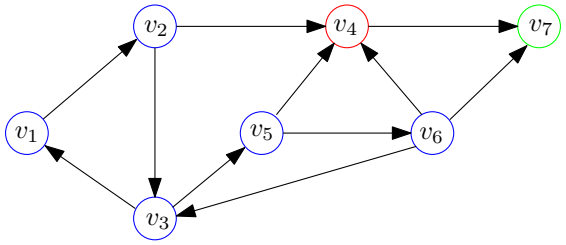
```
def Ponti(G):
    C=0
    v=V[0]
    DFS_ponte(G, v, v, t, C, P, Ponti)
    return Ponti


def DFS_ponte(G, u, v, t, C, P, Ponti):
    C+=1
    t[v]=C
    Back[v]=t[v]
    for u in v.neighbors_out :
        if t[u]==0 :
            P[u]=v
            DFS_ponte(G, v, u, , C, P, Ponti)
            if Back[u]<Back[v] :
                Back[v]=Back[u]
        elif u!=P[v] and t[u]<Back[v] :
            Back[v]=t[u]
    if Back[v]==t[v] :
        P.add((u,v))
```

1.9 Componenti fortemente connessi

Definizione di componente fortemente connesso

In un grafo G un **componente fortemente connesso** è un sottografo massimale (con massimo numero di nodi) fortemente connesso. Due componenti fortemente connessi non hanno nodi in comune. Un nodo singolo non facente parte di nessun componente è anch'esso un componente perchè si può raggiungere da solo.

Esempio:



In questo caso possiamo dividere il grafo in 3 componenti fortemente connessi:

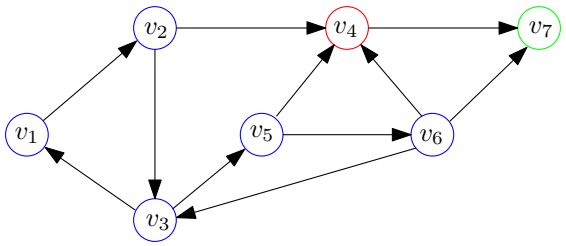
- $H_1 = \{v_1, v_2, v_3, v_5, v_6\}$
- $H_2 = \{v_4\}$
- $H_3 = \{v_7\}$

1.9.1 Contrazione di un componente

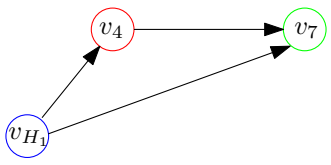
Preso un grafo diretto G e un componente fortemente connesso H, possiamo contrarre H in n solo nodo ottenendo un grafo $G/V(H)$. Il grafo dopo questo processo di contrazione conterrà:

- Nodi: $(V(G) - V(H)) + v_H$
- Archi:
  - $\{(x, y) \in E(G) | x, y \notin V(H)\}$
  - $\{(v_i, v_H)$ se $\exists (v_i, x) \in E(G) | x \in V(H) \wedge v_i \notin V(H)\}$
  - $\{(v_H, v_i)$ se $\exists (x, v_i) \in E(G) | x \in V(H) \wedge v_i \notin V(H)\}$

Esempio:



In questo grafo se contraiamo il componente $H_1 = \{v_1, v_2, v_3, v_5, v_6\}$ il grafo $G/V(H_1)$ diventa:



1.9.2 Algoritmo per trovare i componenti

Dato un grafo G con diverse componenti $H_1, \ldots, H_k$, comprimendo un determinato componente H_i, nel grafo risultante $G/V(H)$ avrò le componenti $H'_1, \ldots, H'_k$ tali che:

$$H'_j = \begin{cases} H_j & j \neq i \\ (H_i/V(H_i)) \cap v_{H_i} & i = j \end{cases}$$

Questo passaggio di contrazione viene applicato in un algoritmo ricorsivo per trovare tutti componenti:

Algoritmo: Trovare i componenti fortemente connessi in un grafo

```
def CompFort(G):
    if ∄ ciclo in G :
        return {{v}|v ∈ V(G)}// insieme di insiemi
    else :
        C=ciclo
    G=G/V(C)
    H_1, ..., H_k=CompFort(G)
    for i in range(k) :
        if v_C ∉ H_i :  // v_C = nodo creato comprimendo C
            H'_i = H_i
        else :
            H'_i = (H_i − {v_C}) ∪ V(C)
```

Il costo di questo algoritmo è $O(n(n+m))$.

1.9.3 Algoritmo di Tarjan

Dato un grafo G con componente fortemente connesso C, definiamo come C-radice il nodo v appartenente a C che è stato visitato per primo dalla DFS.
Preso v nodo C-radice di un componente C e definendo $T(v)$ l'insieme dei discendenti di v nell'arborescenza T e $C(v)$ il componente in cui si trova v allora:

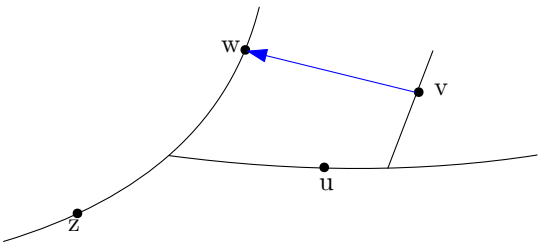
1. $C(v) \subseteq T(v)$
2. Prese $v_1, ..., v_k$ tutte le C-radici in $T(v)$ allora $T(v) = C(v_1) \cup ... \cup C(v_k)$

Tramite queste proprietà possiamo usare un'altro algoritmo per trovare i componenti fortemente connessi:

Algoritmo: Trovare i componenti fortemente connessi in un grafo

```
def SCC(G):
    Stack C=[]
    for v in V(G)|Vis[v]==0 :
        DFS_SCC(G, v, C, output)


def DFS_SCC(G, v, C, output):
    Vis[v]=1
    C.push(v)
    for u in v.neighbors_out|Vis[u]==0 :
        DFS_SCC(G, u, C, output)
    if v è C-radice :  // vedremo dopo come si fa
        X=[]
        w=C.pop()
        X.append(w)
        while w!=v :
            w=C.pop()
            X.append(w)
        output.add(X)
    return output
```

Un nodo u non è C-radice se nella chiamata ricorsiva con radice u viene attraversato un arco (v, w) tale che w è stato già visitato ma il suo componente non ancora stabilito.
Esempio:



Se esiste (v, w) allora la C-radice z di $C(w)$ deve essere un antenato di u e quindi $z, u, v, w \in C(w)$.

Per controllare se un nodo è una C-radice utilizziamo *back* per segnare il punto più indietro raggiungibile da un arco (v, w) in cui:

- v un nodo dentro la chiamata di u
- w è un nodo già visitato ma con componente ancora non individuato

Inotre utilizziamo un array CC in cui:

$$CC[u] = \begin{cases} 0 & \text{non visitato} \\ -t & \text{visitato al tempo } t \text{ ma con componente non identificato} \\ t & \text{componente a cui appartiene} \end{cases}$$

Date queste considerazioni possiamo riscrivere l'algoritmo precedente con costo $= (n + m)$:

Algoritmo: Algritmo di Tarjan

Input:

- G: grafo
- u: nodo radice
- CC: array per segnare i componenti
- S: Stack
- cont_n: contatore tempi di visita
- cont_comp: contatore componenti

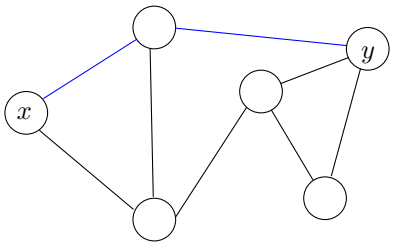
```
def DFS_SCC(G, u, CC, S, cont_n, cont_comp):
    cont_n+=1
    CC[u]=-cont_n
    S.push(u)
    back=cont_n
    for v in u.neighbors_out :
        if CC[v]==0 :
            b=DFS_SCC(G, v, CC, S, cont_n, cont_comp)
            back=min(b, back)
        elif C[v]<0 :
            back=min(back,-CC[v])
    if back==-CC[u] :
        cont_comp+=1
        w=S.pop()
        CC[w]=cont_comp
        while w!=u :
            w=S.pop()
            CC[w]=cont_comp
    return back
```

1.10 BFS (Ricerca in ampiezza)

Dstanza fra due nodi

La distanza fra due nodi x, y è definita come il minimo numero di archi in un cammino $x \to y$ e si scrive $\text{dist}(x, y)$.

Esempio:



In questo caso $\text{dist}(x, y) = 2$.

1.10.1 Algoritmo della BFS

La BFS (breadth first search) è un metodo di visita di un grafo che consiste nel partire da un nodo e controllare prima tutti i nodi con distanza 1 (i vicini), poi tutti quelli con distanza 2 e così via. Con questo algritmo siamo sicuri di sapere sempre la distanza minima di tutti i nodi dalla radice.
Viene implementato tramite un vettore dei padri inizializzato a -1 e usando un array Dist per segnare la distanza. Il costo dell'algoritmo è $O(n + m)$.

Algoritmo: BFS

```
def BFS(G, x):
    P[x]=x
    Queue Q
    Q.enqueue(x)
    while len(Q)!=0 :
        v=Q.dequeue()
        for w in v.neighbors :
            if P[w]==-1 :
                Q.enqueue(w)
                Dist[w]=Dist[v]+1
                P[w]=v
    return P, Dist
```

Dato un grafo G e due nodi x, y esiste sempre un nodo z vicino di y tale che:

$$\text{dist}(x, z) = \text{dist}(x, y) - 1$$

Se $\text{dist}(x, y) = 1 \implies z = x$

Esercizio esempio:
Modificare la BFS per contare anche il numero di cammini possibili $x \to y$ di lunghezza minima:

Algoritmo: Numero di cammini possibili tra due nodi di lunghezza minima

```
def BFS(G, x):
    nCamm[x]=1
    P[x]=x
    Qeue Q
    Q.enquque(x)
    while len(Q)!=0 :
        v.dequeue()
        for w in v.neighbors :
            if P[w]==-1 :
                Q.enqueue(w)
                Dist[w]=Dist[v]+1
                P[w]=v
                nCamm[w]=1
            elif Dist[v]==Dist[w]-1 :
                nCamm[w]+=nCamm[v]
    return P, Dist, nCamm
```

1.10.2 Distanza fra insiemi di nodi

Se vogliamo trovare la distanza minima tra due insiemi di nodi X e Y dobbiamo trovare il minimo tra tutte le distanze che comprendano un nodo di X e uno di Y.
Per fare ciò usiamo una versione modificata della BFS:

Algoritmo: Disanza tra insiemi di nodi

```
def BFS_set(G, X, Y):
    Queue Q
    for x in X :
        Q.enqueue(x)
        Dist[x]=0
    while len(Q!=0) :
        v=Q.dequeue()
        for w in v.neighbors :
            if Dist[w]==-1 :
                Dist[w]=Dist[v]+1
                Q.enqueue(w)
    minimo=∞
    for y in Y :
        minimo=min(Dist[y],minimo)
    return minimo
```

1.10.3 Grafi pesati

Definizione di peso

Un **grafo pesato** è un grafo in cui ogni arco ha associato un numero detto **peso**. Il peso è definito:

$$w : E(G) \to \mathbb{R}^+$$

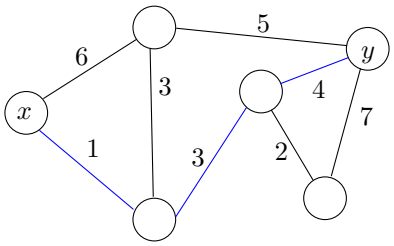
Si definisce, al posto della distanza, il peso di un cammino scritto $\text{dist}_w(x, y)$, cioè la somma tra i pesi di tutti gli archi percorsi in quel cammino, e la distanza diventa quindi il cammino con peso minimo.
Dato un cammino P il suo peso sarà:

$$w(P) = \sum_{e \in P} w(e)$$

Presi due nodi qualsiasi x, y vale che:

1. $\text{dist}_w(x, x) = 0$
2. $\text{dist}_w(x, y) > 0 \iff x \neq y$
3. $\text{dist}_w(x, y) \leq \text{dist}_w(x, z) + \text{dist}_w(z, y)\ \forall z \in V(G)$

Esempio:



In questo caso il cammino di peso minimo $\text{dist}_w(x, y) = 1 + 3 + 4 = 8$.

1.10.4 Calcolare distanze pesate (Dijkstra)

Un problema con i grafi pesati è quello di non poter calcolare la distanza pesata tra due nodi, neanche se vicini perchè potrebbe esserci un cammino con più archi ma peso minore.

Peso minimo

Dato un grafo pesato G e un nodo x, scriviamo $\alpha_i = w(x, v_i)$ per ogni nodo v_i vicino di x:

$$\min(\alpha_1, \ldots, \alpha_k) = \alpha_i \iff \text{dist}_w(x, v_i) = \alpha_i$$

Questa regola può essere generalizzata anche per un insieme per cui è nota la distanza da x, cioè dato un insieme R di vertici per cui è nota la distanza da x e (u, v) l'arco che minimizza $\text{dist}_w(x, u) + w(u, v)$ con $e \in R \wedge v \notin R$ allora:

$$\text{dist}_w(x, v) = \text{dist}(x, u) + w(u, v)$$

In questo modo da un insieme R di nodi di cui è nota la distanza da un nodo x, si può sempre aggiungere a R un nodo vicino ad un qualsiasi nodo di R. Questo si può fare con un algoritmo chiamato Dijkstra:

Algoritmo: Dijkstra

```
def Dijkstra(G, x):
    R=set(x)
    while R!=V(G) :
        minimo=∞
        min_arco=None
        for (v,u) in E(G)| v in R and u not in R :
            if Dist[v]+w(v,u)<minimo :
                minimo=Dist[v]+w(v,u)
                min_arco=(v,u)
        R.add(min_arco.u)
        Dist[min_arco.u]=minimo
    return Dist
```

La complessità è $O(n(n+m))$, ma può essere ottimizzato usando un min heap H per memorizzare gli archi:

Algoritmo: Dijkstra ottimizzato

```
def Dijkstra(G, x):
    Dist[x]=0
    R=set(x)
    P[x]=x
    for v in V(G) :
        if v!=x :
            H.insert(v, key=∞)
        else :
            H.insert(v, key=0)
    while len(H)!=0 :
        v=H.extract_min()
        Dist[v]=H.key(v)
        R.add(v)
        for u in v.neighbors :
            if u not in R and Dist[u]>Dist[v]+w(v,u) :
                Dist[u]=Dist[v]+w(v,u)
                H.update_key(u, Dist[u])
                P[u]=v
    return Dist
```

Il costo di questa versione ottimizzata è di $O((n+m)\cdot \log n)$.

2

Algoritmi Greedy

Definizione di algoritmo Greedy

Un algoritmo greedy è un algoritmo che partendo da una soluzione non ottimale (solitamente vuota) controlla tutti i possibili passi che si possono fare per estendere la soluzione e per ogni passo se è fattibile viene aggiunto alla soluzione. Alla fine la soluzione trovata sarà sicuramente possibile ma va dimostrato che è ottimale.

Per dimostrare che la soluzione trovata è anche ottimale:

1. Dimostrare che la soluzione trovata rispetti le caratteristiche previste
2. Dimostrare che ogni istanza della soluzione (soluzione dopo ogni iterazione) è contenuta nella soluzione ottimale:
   Supponendo che l'istanza $\texttt{Sol}_k$ sia contenuta nella soluzione ottimale $\texttt{Sol}^*$ bisogna dimostrare che anche l'istanza $\texttt{Sol}_{k+1}$ sia contenuta in una soluzione ottimale. Di solito questa soluzione ottimale consiste in $(\texttt{Sol}^* - x) \cup y | x \in \texttt{Sol}^* \wedge y \notin \texttt{Sol}^*$
3. Dimostrare che la soluzione output dell'algoritmo sia uguale alla soluzione ottimale che la contiene

Esempio:
Dato un insieme I di intervalli $I_1, ..., I_n$ nella forma $I_i = [a_i, b_i]$, scrivere un algoritmo che trovi il numero massimo di intervalli disgiunti possibile.
Per farlo prima ordiniamo gli intervalli in base alla fine in modo crescente.

Algoritmo: Massimo numero di intervalli disgiunti

Input:
- I: insieme di intervalli

```
def max_int_disj(I):
    I.sort(key= lambda x[b]:x)
    Sol=set()
    right_bound=−∞
    for Int in I :
        if Int[a]>right_bound :
            Sol.add(Int)
            right_bound=Int[b]
    return Sol
```

Dimostrazione:

1. La soluzione contiene sicuramente intervalli disgiunti, quindi rispetta le condizioni
2. Supponiamo esista una soluzione ottimale $\texttt{Sol}^*$ per cui $\texttt{Sol}_k \subseteq \texttt{Sol}^*$:
   - Caso base:
     $\texttt{Sol}_0 = \emptyset \subset \texttt{Sol}^*$
   - Ipotesi induttiva:
     Supponiamo sia vero per qualsiasi soluzione precedente a $\texttt{Sol}_k$, dobbiamo dimostrare che $\texttt{Sol}_k \subseteq \texttt{Sol}^* \implies \texttt{Sol}_{k+1} \subseteq \texttt{Sol}^*$
   - Dimostrazione induttiva:

     $$\texttt{Sol}_{k+1} = \begin{cases} \texttt{Sol}_k & \exists I_i \in \texttt{Sol}_k | I_{k+1} \cap I_i \neq \emptyset \\ \texttt{Sol}_k \cup I_{k+1} & \forall I_i \in \texttt{Sol}_k \implies I_i \cap I_{k+1} = \emptyset \end{cases}$$

     Nel primo caso $\texttt{Sol}_{k+1} = \texttt{Sol}_k \subseteq \texttt{Sol}^*$
     Nel secondo caso se $\texttt{Sol}_{k+1} \not\subseteq \texttt{Sol}^* \implies \exists I_j \in \texttt{Sol}^* \wedge I_j \notin \texttt{Sol}_k | I_j \cap I_{k+1} \neq \emptyset$, inoltre $j > k+1$ perchè sennò I_j sarebbe già in $\texttt{Sol}_k$ e quindi in $\texttt{Sol}_k + 1$. Essendo $j > k+1$ nell'algoritmo verrà preso prima I_{k+1} di I_j quindi $(\texttt{Sol}^* - I_j) \cup I_{k+1}$ è una soluzione ottimale che contiene $\texttt{Sol}_{k+1}$.
3. Supponendo che l'output dell'algoritmo $\texttt{Sol}_n$ sia diverso dalla soluzione ottimale $\texttt{Sol}^*$ allora $\exists I_i \in \texttt{Sol}^* | I_i \notin \texttt{Sol}_n$ ma per cui $I_i \cap I_j = \emptyset \ \forall I_j \in \texttt{Sol}_n$ essendo $\texttt{Sol}_n \subseteq \texttt{Sol}^*$, ma allora alla i-esima iterazione, precedente alla fine dell'algoritmo, I_i dovrebbe essere in $\texttt{Sol}_n$ quindi $\texttt{Sol}_n = \texttt{Sol}^*$

2.1 Alberi di copertura

Minimum Spanning Tree

Un **albero di copertura** in un grafo G è un sottografo T aciclico e tale che $V(T) = V(G)$. Si chiama **Minimum Spanning Tree (MST)** un albero di copertura con peso minimo. Un qualsiasi sottografo connesso che contiene tutti i nodi e ha peso minimo è sempre un MST.

2.1.1 Algoritmo di Kruskal

L'algoritmo di Kruskal permette dato un grafo connesso di ottenere l'MST.
Per farlo dobbiamo ordinare gli archi per peso crescente.

Algoritmo: Algoritmo di Kruskal

```
def Kruskal(G):
    E.sort(key= w(e):e)
    Sol=set()
    for e in E(G) :
        if Sol ∪ e non contiene cicli :
            Sol.add(e)
```

Dimostrazione:

1. G è connesso quindi $\forall v \in V(G)$ esiste almeno un arco che collega v preso dal ciclo, quindi $V(\texttt{Sol}_m) = V(G)$. Per lo stesso principio $\texttt{Sol}_m$ è anche connesso, quindi $\texttt{Sol}_m$ è un albero di copertura di G.

2. Supponiamo che esista una soluzione ottimale $\texttt{Sol}^*$ per cui $\texttt{Sol}_k \subseteq \texttt{Sol}^*$:

   - Caso base:
     $\texttt{Sol}_0 = \emptyset \subset \texttt{Sol}^*$
   - Ipotesi induttiva:
     Supponiamo sia vero per qualsiasi soluzione precedente a $\texttt{Sol}_k$, dobbiamo dimostrare che $\texttt{Sol}_k \subseteq \texttt{Sol}^* \implies \texttt{Sol}_{k+1} \subseteq \texttt{Sol}^*$
   - Dimostrazione induttiva:

     $$\texttt{Sol}_{k+1} = \begin{cases} \texttt{Sol}_k & \texttt{Sol}_k \cup e_{k+1} \text{ contiene cicli} \\ \texttt{Sol}_k + e_{k+1} & \texttt{Sol}_k \cup e_{k+1} \text{ non contiene cicli} \end{cases}$$

     Nel primo caso $\texttt{Sol}_{k+1} = \texttt{Sol}_k \subseteq \texttt{Sol}^*$
     Nel secondo caso se $\texttt{Sol}_{k+1} \not\subseteq \texttt{Sol}^* \implies \exists e_j \in \texttt{Sol}^* \wedge e_j \notin \texttt{Sol}_k | \texttt{Sol}^* \cup e_j$ è ciclico, ma visto che $e_j \notin \texttt{Sol}_k$ ed essendo gli archi in ordine di peso, vuol dire che $w(e_{k+1}) \leq w(e_j)$ e quindi $(\texttt{Sol}^* - e_j) \cup e_{k+1}$ è una soluzione ottimale che contiene $\texttt{Sol}_{k+1}$.

3. Esiste quindi una soluzione $\texttt{Sol}^*$ che contiene l'output dell'algoritmo $\texttt{Sol}_m$ ed essendo $\texttt{Sol}_m$ un albero di copertura allora $\texttt{Sol}_m = \texttt{Sol}^*$.

2.2 Grafi con pesi negativi

Se consideriamo un grafo pesato G con pesi anche negativi, cioè:

$$w : E(G) \to \mathbb{R}$$

Per trovare un sotto-grafo connesso H che contiene tutti i vertici con peso minimo dobbiamo modificare l'algoritmo precedente:

Algoritmo: Algoritmo di Kruskal con pesi negativi

```
def Kruskal_neg(G):
    E.sort(key= w(e):e)
    Sol={x ∈ E(G)|w(x) < 0}
    for e in E(G) :
        if ∄ cammino (x → y) in Sol :
            Sol.add(e)
    return Sol
```

2.3 Algoritmo di Prim

L'algoritmo di Prim permette, dato un grafo connesso di ottenere l'MST.
Per farlo dobbiamo ordinare gli archi per peso crescente.

Algoritmo: Algoritmo di Prim

```
def Prim(G):
    E.sort(key= w(e):e)
    Vis[x]=1
    Sol=set()
    while ∃ v| Vis[v]=0 :
        for (u,v) in E(G) :
            if Vis[u]==1 :
                Sol.add((u,v))
                Vis[v]=1
    return Sol
```

Il costo computazionale di questo algoritmo è $O(nm)$.

Dimostrazione:

1. G è connesso quindi $\forall v \in V(G)$ esiste almeno un arco che collega v preso dal ciclo, quindi $V(\texttt{Sol}_m) = V(G)$. Per lo stesso principio $\texttt{Sol}_m$ è anche connesso, quindi $\texttt{Sol}_m$ è un albero di copertura di G.

2. Supponiamo che esista una soluzione ottimale $\texttt{Sol}^*$ per cui $\texttt{Sol}_k \subseteq \texttt{Sol}^*$:

   - Caso base:
     $\texttt{Sol}_0 = \emptyset \subset \texttt{Sol}^*$
   - Ipotesi induttiva:
     Supponiamo sia vero per qualsiasi soluzione precedente a $\texttt{Sol}_k$, dobbiamo dimostrare che $\texttt{Sol}_k \subseteq \texttt{Sol}^* \implies \texttt{Sol}_{k+1} \subseteq \texttt{Sol}^*$
   - Dimostrazione induttiva:

     $$\texttt{Sol}_{k+1} = \begin{cases} \texttt{Sol}_k & \nexists v | Vis[v] = 0 \\ \texttt{Sol}_k + e_{k+1} & \exists e_{k+1} = (u, v) | Vis[v] = 0 \wedge Vis[u] = 1 \end{cases}$$

     Nel primo caso $\texttt{Sol}_{k+1} = \texttt{Sol}_k \subseteq \texttt{Sol}^*$
     Nel secondo caso se $\texttt{Sol}_{k+1} \not\subseteq \texttt{Sol}^* \implies \exists e_j \in \texttt{Sol}^* \wedge e_j \notin \texttt{Sol}_k | e_j = (x, v)$ per un qualche nodo x, ma visto che $e_j \notin \texttt{Sol}_k$ ed essendo gli archi in ordine di peso, vuol dire che $w(e_{k+1}) \leq w(e_j)$ e quindi $(\texttt{Sol}^* - e_j) \cup e_{k+1}$ è una soluzione ottimale che contiene $\texttt{Sol}_{k+1}$.

3. Esiste quindi una soluzione $\texttt{Sol}^*$ che contiene l'output dell'algoritmo $\texttt{Sol}_m$ ed essendo $\texttt{Sol}_m$ un albero di copertura allora $\texttt{Sol}_m = \texttt{Sol}^*$.

3

Algoritmi divide et impera

Gli algoritmi divide et impera consistono nel dividere il problema totale in sotto-problemi che vengono risolti ricorsivamente. Per calcolare il costo computazionale di questi problemi si utilizzano le equazioni di ricorrenza.

3.1 Teorema principale

Data un'equazione di ricorrenza nella forma:

$$T = \begin{cases} T(n) = aT(\frac{n}{b}) + f(n) \\ T(1) = \Theta(1) \end{cases}$$

Con $a \geq 1$ e $b > 1$.
Per risolverla ci sono vari casi:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{se } f(n) = O(n^{\log_b(a)-\epsilon}) \\ \Theta(n^{\log_b a} \cdot \log n) & \text{se } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{se } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ e } a \cdot f(\frac{n}{b}) \leq c \cdot f(n) \end{cases}$$

3.2 Esercizi divide et impera

3.2.1 Sottoarray di somma massima

Preso un array in input composto da numeri interi, trovare in $\Theta(n \log n)$ il sottoarray di somma massima.
Soluzione:

Algoritmo: Sottoarray di somma massima

```
def max_subarray(A, a, b):
    if a==b :
        return A[a],0
    m=(a+b)//2
    sx=max_subarray(A, a, m)
    dx=max_subarray(A, m+1, b)
    somma, pref, suff=0
    for i in range(a, m) :
        somma+=A[i]
        pref=max(pref, somma)
    somma=0
    for i in range(m+1, b) :
        somma+=A[i]
        suff=max(suff, somma)
    return max(sx, dx, pref+suff)
```

Costo computazionale:

$$\begin{cases} T(n) = 2T(\frac{n}{2}) + \Theta(n) \\ T(1) = \Theta(1) \end{cases}$$

$$n^{\log_b a} = n^{\log_2 2} = n \implies f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(n \log n)$$

3.2.2 Valore singolo in un array

Dato un array ordinato di lunghezza dispari in cui ogni valore compare 1 o 2 volte, trovare in $\Theta(\log n)$ un valore che appare una sola volta.
Soluzione:

Algoritmo: Valore singolo in un array

```
def single(A, a, b):
    if a==b :
        return A[a]
    m=(a+b)//2
    if A[m]!=A[m+1] and A[m]!=A[m-1] :
        return A[m]
    // nei 4 casi vado sempre dove la parte rimanente da controllare è
    dispari
    if A[m]==A[m+1] :
        if m%2!=0 :
            return single(A, a, m-1)
        else :
            return single(A, m+2, b)
    elif A[m]==A[m-1] :
        if m%2==0 :
            return single(A, a, m-2)
        else :
            return single(A, m+1, b)
```

Costo computazionale:

$$\begin{cases} T(n) = T(\frac{n}{2}) + \Theta(1) \\ T(1) = \Theta(1) \end{cases}$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \implies f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(\log n)$$

3.3 Elemento maggioritario in un array

Dato un array trovare in $\Theta(n \log n)$ un elemento che appare più di $\frac{n}{2}$ volte all'interno dell'array.
Soluzione:

Algoritmo: Elemento maggioritario in un array

```
def major(A, a, b):
    if a==b :
        return A[a]
    m=(a+b)//2
    sx=major(A, a, m)
    dx=major(A, m+1, b)
    count_sx, count_dx=0
    for i in range(a,b) :
        if sx!=None and A[i]==sx :
            count_sx+=1
        if dx!=None and A[i]==dx :
            count_dx+=1
    if count_sx>=m+1 :
        return sx
    if count_dx>=m+1 :
        return dx
    return None
```

Costo computazionale:

$$\begin{cases} T(n) = 2T(\frac{n}{2}) + \Theta(n) \\ T(1) = \Theta(1) \end{cases}$$

$$n^{\log_b a} = n^{\log_2 2} = n \implies f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(n \log n)$$

4

Programmazione dinamica

La programmazione dinamica è un approccio algoritmico basato sulla risoluzione di un problema partendo da soluzioni dello stesso problema ma di dimensioni più piccole. A differenza dell'approccio divide et impera in cui il problema di solito si risolve tramite la ricorsione, nella programmazione dinamica si usano solitamente matrici per considerare tuti i casi possibili.

4.1 Esercizi di programmazione dinamica

4.1.1 Ottimizzare lo spazio su un disco

Dato un disco di capacità C e n file di dimensione $s_1, ..., s_n$, trovare l'insieme di file che ottimizza lo spazio del disco (occupa più spazio possibile).
Soluzione:
Definiamo una matrice T di grandezza $(n+1) \times (C+1)$ in cui:

- $T[k, \alpha]$ = capacità massima che si può riempire in un disco con capacità α e usando i primi k file
- $T[0, \alpha] = 0 \; \forall \alpha$
- $T[k, 0] = 0 \; \forall k$

Gli altri valori della matrice verranno riempiti:

$$T[k,\alpha] = \begin{cases} \max(T[k-1,\alpha], T[k-1,\alpha-s_k]+s_k) & s_k \leq \alpha \\ T[k-1,\alpha] & s_k > \alpha \end{cases}$$

Algoritmo: Ottimizzare lo spazio su un disco

Input:

- C: capacità del disco
- S: insieme contente i pesi dei file

```
def DiskSpace(C, S):
    n=len(S)
    T=(n+1)×(C+1)// inizializzata a 0
    for i in range(n) :
        for j in range(C) :
            if S[i]>C :
                T[i][j]=T[i-1][j]
            else :
                T[i][j]=max(T[i-1][j],T[i-1][j-S[i]]+S[i])
    maxS=T[n][C]
    Sol=set()
    for k in range(n,0,-1) :
        if T[k][maxS]!=T[k-1][maxS] :
            Sol.add(S[k])
            maxS-=S[k]
    return Sol, T[n][C]
```

Il costo dell'algoritmo è $O(nC)$ ma l'input è $n \log(C)$ quindi l'algoritmo è esponenziale rispetto all'input.

4.1.2 Cammini colorati su una scacchiera

In una scacchiera $n \times n$ in cui ogni casella è colorata di rosso o di blu, cioè:

$$C[i,j] = \begin{cases} \text{rosso} \\ \text{blu} \end{cases}$$

Una pedina che parte da $(0,0)$ deve arrivare a (n,n) potendo attraversare solo caselle blu e potendosi muovere solo in basso o a destra. Si vuole sapere il numero di cammini possibili.
Soluzione:
Definiamo una matrice T di grandezza $n \times n$ in cui:

- $T[i,j]$ = numero di cammini possibili da $(0,0)$ a (i,j)
- $T[0,j] = \begin{cases} 1 & T[0,j-1] = 1 \wedge C[0][j] = \text{blu} \\ 0 & \text{altrimenti} \end{cases}$
- $T[i,0] = \begin{cases} 1 & T[i-1,0] = 1 \wedge C[i][0] = \text{blu} \\ 0 & \text{altrimenti} \end{cases}$

Gli altri valori della matrice verranno riempiti:

$$T[i,j] = T[i-1,j] + T[i,j-1]$$

Algoritmo: Cammini colorati su una scacchiera

```
def nCamm(C):
    n=len(C)
    T=n×n// inizializzata a 0
    if C[0][0]==rosso :
        return 0
    else :
        T[0][0]=1
    for j in range(n) :  // riempe la prima riga
        if T[0][j-1]==1 and [0][j]==blu :
            T[0][j]=1
        else :
            T[0][j]=0
    for i in range(n) :  // riempe la prima colonna
        if T[i-1][0]==1 and [i][0]==blu :
            T[i][0]=1
        else :
            T[i][0]=0
    for i in range(1,n) :
        for j in range(1,n) :
            T[i][j]=T[i-1][j]+T[i][j-1]
    return T[n][n]
```

4.1.3 Ottimizzare il peso in uno zaino