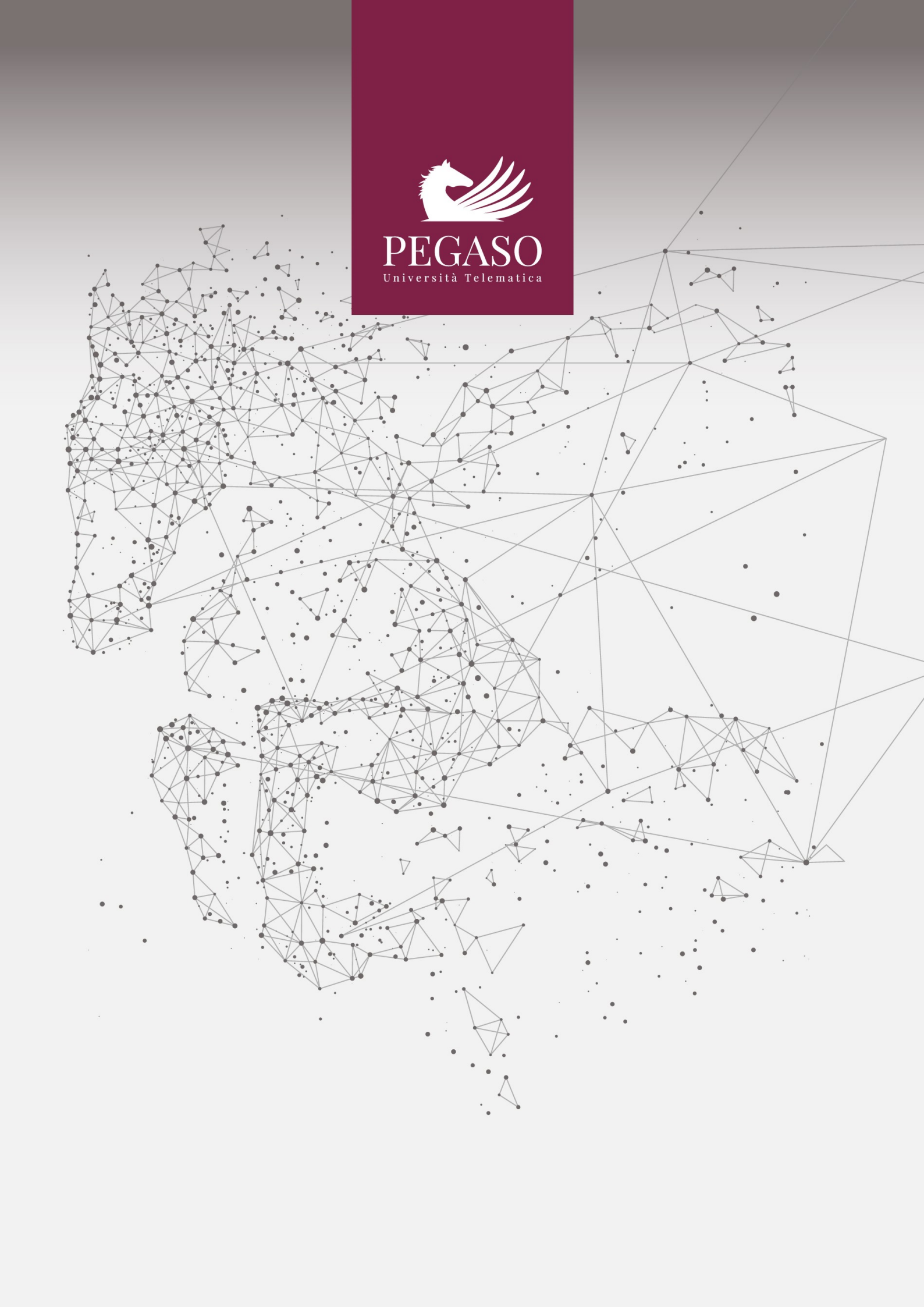
PEGASO
Università Telematica

Indice

1. Albero di copertura DF

L'albero di copertura DF viene costruito in modo che ogni nodo del grafo sia raggiungibile da un nodo radice dell'albero (che rappresenta il nodo di partenza dell'algoritmo); tale albero viene chiamato "di copertura" perché copre appunto tutti i nodi del grafo (relativi alla componente connessa del grafo).

In un albero di copertura DFS (Depth-First Search), il termine **marcato** si riferisce ai nodi che sono stati visitati durante l'esecuzione dell'algoritmo (durante la visita in profondità, quando si visita un nodo, viene segnato come "marcato" e viene aggiunto all'albero di copertura DFS).

La visita **DFS** genera dunque l'albero dei cammini DFS:

- Tutte le volte che viene incontrato un arco che connette un nodo marcato ad uno non marcato, esso viene inserito nell'albero T e viene detto **arco dell'albero**. Gli archi (u, v) **non inclusi** nell'albero possono essere divisi in tre categorie:
  - Se u è un **antenato** di v in T, (u, v) è detto **arco in avanti** (se l'arco è esaminato passando da un nodo di T ad un suo discendente, che non sia figlio);
  - Se u è un **discendente** di v in T, (u, v) è detto **arco all'indietro** (se l'arco è esaminato passando da un nodo di T ad un altro nodo che è suo antenato);
  - Altrimenti, viene detto **arco di attraversamento.**

Costruiamo l'algoritmo per gestire la visita DFS utilizzando le seguenti strutture dati:

- $counter$: contatore
- $d[]$: discovery
- $f[]$: finish

Per tenere traccia di quando un nodo viene scoperto nella visita, si utilizza una struttura **discovery**; la struttura **finish** indica invece che tutte le visite dei nodi adiacenti sono state completate.

```
DFS_SCHEMA(Graph G, Node u, int &counter, int[] d, int[] f)
{ visita il nodo u (pre-order) }
counter = counter + 1;
d[u] = counter
foreach v in G.adj(u)
     { visita l'arco (u, v) (qualsiasi) }
     if d[v] == 0
          { visita l'arco (u, v) (albero) }
          DFS_SCHEMA(G, v, counter, d, f)
     else if d[u] > d[v] and f[v] == 0
          { visita l'arco (u, v) (arco all'indietro) }
```

```
     else if d[u] < d[v] and f[v] != 0
          { visita l'arco (u, v) (arco in avanti) }
     else
          { visita l'arco (u, v) (arco di attraversamento) }
{ visita il nodo u (post-order) }
counter = counter + 1;
f[u] = counter
```

2. Applicazioni delle proprietà degli archi

Le proprietà degli archi possono essere usate per "costruire" algoritmi più efficienti. Partendo dalle proprietà che abbiamo introdotto attraverso l'analisi degli esempi precedenti, possiamo enunciare il seguente **teorema**: data una visita **DFS** di un grafo $G = (V, E)$, per ogni coppia di nodi $(u, v) \epsilon V$, solo una delle condizioni seguenti è vera:

1. Gli intervalli $[d[u], f[u]]$ e $[d[v], f[v]]$ sono non-sovrapposti; (u, v) non sono discendenti l'uno dell'altro nella foresta DF
2. L'intervallo $[d[u], f[u]]$ è contenuto in $[d[v], f[v]]$; u è un discendente di v in un albero DF
3. L'intervallo $[d[v], f[v]]$ è contenuto in $[d[u], f[u]]$; v è un discendente di u in un albero DF

Da precisare che non esiste una sola visita DFS di un grafo ma ne possono esistere differenti: dipende infatti da dove si parte e come si scelgono i nodi adiacenti da visitare per primi.

Dimostriamo "intuitivamente" il teorema precedente; dato il seguente grafo:

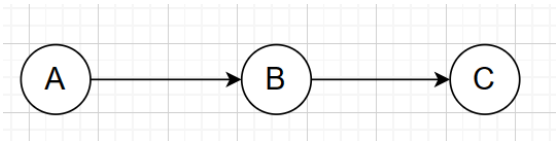

Figura 1: Digrafo

Eseguiamo la visita partendo dal nodo A:

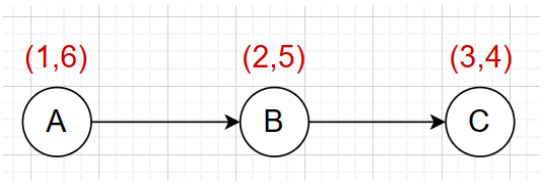

Figura 2: visita DFS

Se prendiamo in esame l'intervallo (2, 5) possiamo certamente dire che questo "contiene" l'intervallo (3, 4); ci troviamo nella condizione 3 del teorema precedente, e cioè possiamo dire che il nodo B contiene il nodo C e dunque C è un discendente di B, e B è un antenato di C; allo stesso modo A è un antenato di B e C e B e C sono discendenti di A.

L'idea è quella per cui quando mettiamo un finish time su un nodo, questo finish è certamente più grande dei time dei discendenti del nodo in questione. Se non si rientra nei casi 2 e 3 del teorema (che sono uno il duale dell'altro) ci troviamo nello scenario 1 in cui non c'è discendenza tra le coppie di nodi.

Se riprendiamo il seguente grafo:

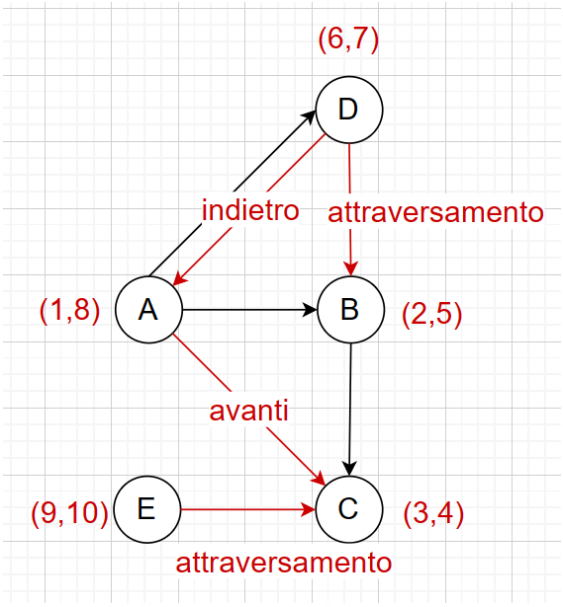

Figura 3: DFS del grafo

Possiamo dire ad es. che:

- A contiene B, C, D (cioè l'intervallo di A contiene gli intervalli di B, C e D)
- B contiene C
- A non contiene E

Un ulteriore **teorema** rilevante in tale contesto è il seguente: un grafo orientato è aciclico se e solo se non esistono archi all'indietro nel grafo.

Dimostriamo il **SE**. Consideriamo il seguente scenario:

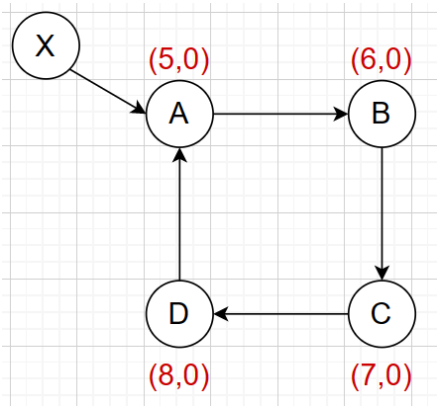

Figura 4: dimostrazione SE

Supponiamo che si arrivi al nodo A con un discovery time pari a 5 (può essere qualsiasi valore); proseguiamo visitando i discendenti di A ed inseriamo i discovery time dei discendenti. Arrivati all'arco (D, A) si ha che: $d(D) > d(A)$ e $t(A) = 0$ pertanto si ha che tale arco è un arco all'indietro.

Possiamo concludere dunque che **SE** c'è un ciclo (e l'arco (D, A) determina la chiusura del ciclo) c'è un arco all'indietro.

Dimostriamo il **SOLO SE**. Considerando l'esempio precedente; se esiste un arco all'indietro (l'arco (D, A) identificato in precedenza), allora esiste anche un cammino da A a D (quello che passa per B e C) che poi si congiunge all'arco all'indietro e determina dunque il ciclo.

Possiamo determinare l'algoritmo per l'identificazione di un ciclo in un grafo, correlandolo alla presenza di un arco all'indietro:

```
HAS_CYCLE(Graph G, Node u, int &counter, int[] d, int[] f)
{ visita il nodo u (pre-order) }
counter = counter + 1
d[u] = counter
foreach v in G.adj(u)
     if d[v] == 0
          if HAS_CYCLE(G, v, counter, d, f)
               return true
     else if d[u] > d[v] and f[v] == 0
          return true
counter = counter + 1
f[u] = counter
return false
```

come si può vedere dall'algoritmo, la condizione:

```
if d[u] > d[v] and f[v] == 0
```

è proprio quella che è relativa all'identificazione dell'arco all'indietro.

3. Ordinamento topologico

Dato un grafo diretto ed aciclico $G = (V, E)$, un ordinamento topologico per questo grafo G è un ordinamento lineare dei suoi nodi tale che se $(u, v) \epsilon E$, allora u appare prima di v nell'ordinamento.

Da un punto di vista "visivo" questo significa sostanzialmente "srotolare" un grafo da sinistra a destra, e di fatto il risultato finale può essere paragonato all'avere un progetto con i vari task in cui l'arco rappresenta la dipendenza (l'avvio di una attività solo dopo che è finita la precedente o le precedenti).

ATTENZIONE: può non esistere un unico ordinamento topologico per un grafo; inoltre, se un grafo contiene un ciclo, non può esistere un ordinamento topologico

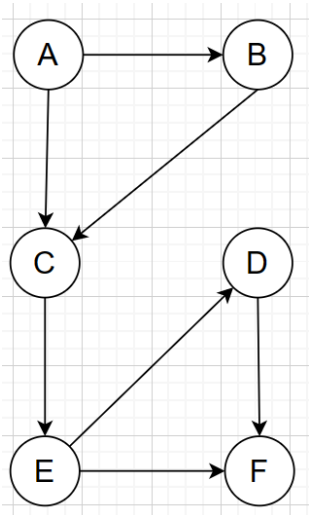

Figura 5: Grafo Orientato Aciclico

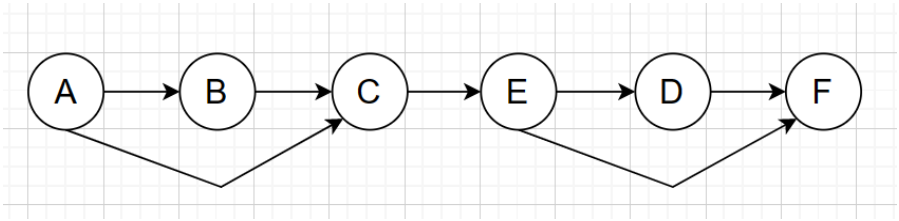

Figura 6: Ordinamento Topologico

Se volessimo scrivere un algoritmo per individuare un ordinamento topologico, iniziamo con l'individuare una soluzione naive (ingenua):

- Trovare un nodo senza archi entranti
- Aggiungere questo nodo nell'ordinamento e rimuoverlo, insieme a tutti i suoi archi
- Ripetere questa procedura fino a quando tutti i nodi sono stati rimossi

Applichiamo questa soluzione al grafo precedente:

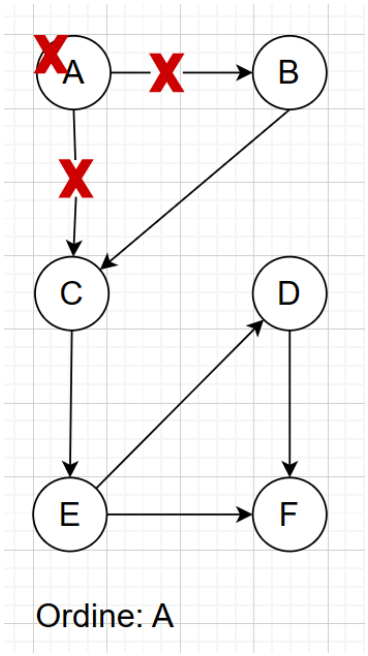

Figura 7: Ordinamento topologico – Step1

Abbiamo solamente il nodo A senza archi entranti quindi lo scelgo, lo aggiungo nell'ordinamento e cancello tutti i suoi archi: (A, B) e (A, C).

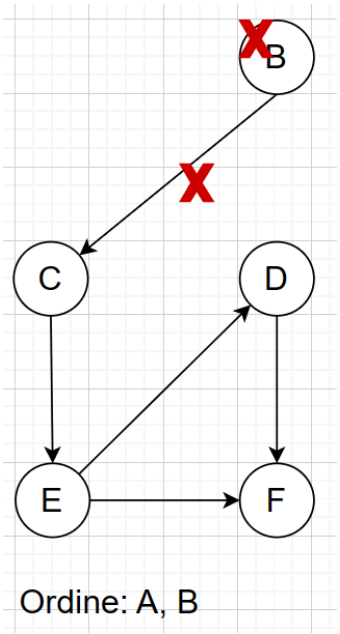

Figura 8: Ordinamento topologico – Step2

Abbiamo solamente il nodo B senza archi entranti quindi lo scelgo, lo aggiungo nell'ordinamento e cancello tutti i suoi archi: (B, C).

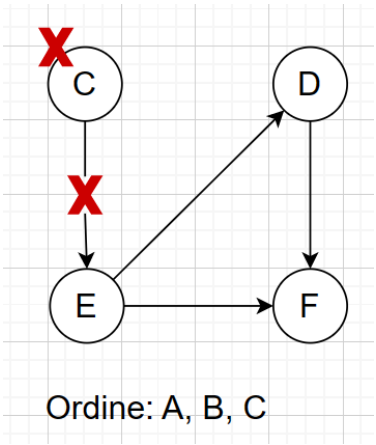

Figura 9: Ordinamento topologico – Step3

Abbiamo solamente il nodo C senza archi entranti quindi lo scelgo, lo aggiungo nell'ordinamento e cancello tutti i suoi archi: (C, E).

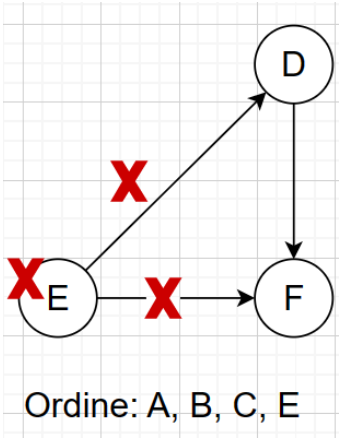

Figura 10: Ordinamento topologico – Step4

Abbiamo solamente il nodo E senza archi entranti quindi lo scelgo, lo aggiungo nell'ordinamento e cancello tutti i suoi archi: (E, D) ed (E, F).

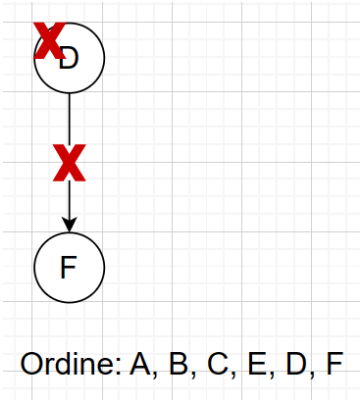

Figura 11: Ordinamento topologico – Step5

Abbiamo solamente il nodo D senza archi entranti quindi lo scelgo, lo aggiungo nell'ordinamento e cancello tutti i suoi archi: (D, F). A questo punto ci resta solo il nodo F che inserisco nell'ordinamento.

Nella procedura abbiamo sempre avuto un solo nodo alla volta senza archi entranti quindi non abbiamo dovuto mai eseguire alcuna scelta; tuttavia, se fosse accaduto che vi fossero più nodi con la stessa caratteristica, avrei potuto scegliere a piacere con quale proseguire (questo è il motivo per cui non esiste di fatto un solo ordine topologico).

Una soluzione più efficiente rispetto a quella naive è quella basata su DFS: eseguire una DFS nel quale l'operazione di visita consiste nell'aggiungere il nodo in testa ad una lista, "at finish time". La lista così ottenuta contiene la sequenza dei nodi ordinati per tempo decrescente di fine.

Tale metodo è corretto poiché quando un nodo arriva a "finish time", significa che tutti i suoi discendenti sono stati scoperti ed aggiunti alla lista: aggiungendolo in testa alla lista garantisco che il nodo è in ordine corretto.

Da notare che aggiungerlo in testa ad una lista è equivalente ad aggiungerlo in uno stack (in cui vale la politica LIFO – Last In First Out).

Analizziamo l'algoritmo:

```
Stack topological_sort(Graph G)
     Stack S = Stack()
     boolean[] visited = boolean[G.V()]
     foreach u in G.V()
          visited[u] = false
     foreach u in G.V()
          if !visited[u]
               ts_dfs(G, u, visited, S)
     return S
ts_dfs(Graph G, Node u, boolean[] visited, Stack S)
     visited[u] = true
     foreach v in G.adj(u)
          if not visited[v]
               ts_dfs(G, v, visited, S)
     S.push(u)
```

Facciamo un esempio sul seguente grafo:

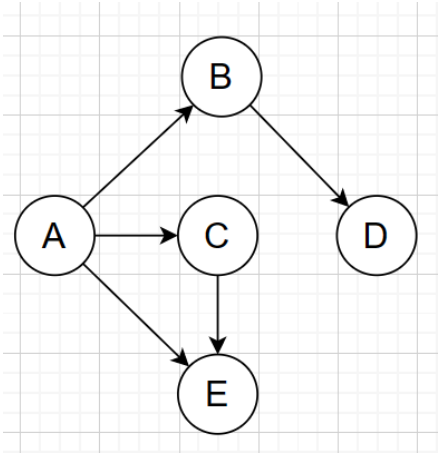

Figura 12: Grafo per analisi DFS

Eseguiamo l'analisi DFS:

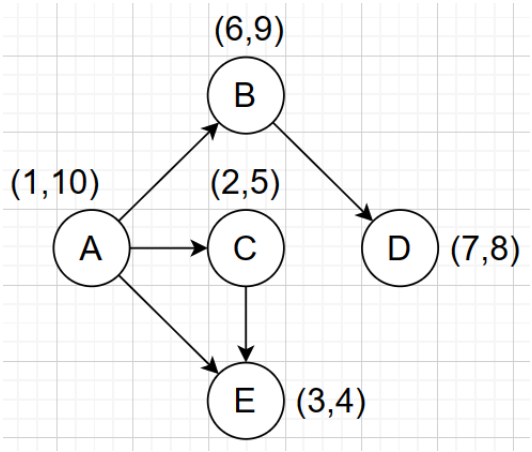

Figura 13: DFS

In base all'algoritmo introdotto in precedenza, l'inserimento nello stack avviene ogni volta che c'è un finish time e pertanto i nodi saranno inseriti nello stack nel seguente ordine:

```
S.push(E)
S.push(C)
S.push(D)
S.push(B)
S.push(A)
```

Gli elementi nello stack saranno dunque collocati nel seguente ordine:

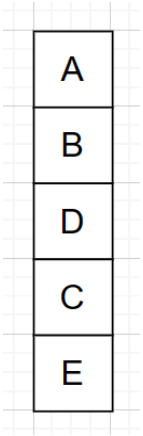

Figura 14: stack

Di seguito l'implementazione in Python:

```
def topological_sort(graph):
    stack = []
    visited = {v: False for v in graph}

    for vertex in graph:
        if not visited[vertex]:
            ts_dfs(graph, vertex, visited, stack)

    stack.reverse()
    return stack

def ts_dfs(graph, vertex, visited, stack):
    visited[vertex] = True
    for neighbor in graph[vertex]:
        if not visited[neighbor]:
            ts_dfs(graph, neighbor, visited, stack)
    stack.append(vertex)
```

4. Componenti connesse

Molti algoritmi che operano sui grafi iniziano decomponendo il grafo nelle sue componenti: l'algoritmo viene poi eseguito su ognuna delle componenti ed infine i risultati vengono poi ricomposti assieme.

Introduciamo il concetto di raggiungibilità.

In un digrafo, se esiste un cammino c tra i nodi u e v, si dice che v è raggiungibile da u tramite c; allo stesso modo in un grafo non orientato se esiste una catena c tra i vertici u e v, si dice che v è raggiungibile da u tramite c.

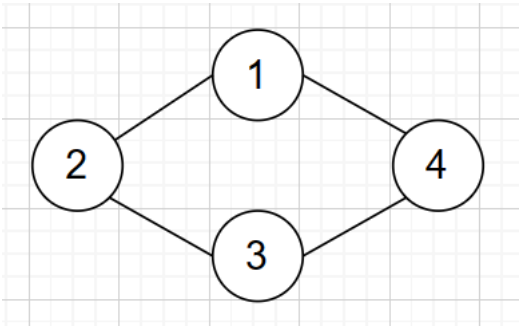

Figura 15: raggiungibilità

Nel grafo in figura si ha (ad es.) che il nodo 1 è **raggiungibile** dal nodo 4 e viceversa. Nel seguente grafo invece:

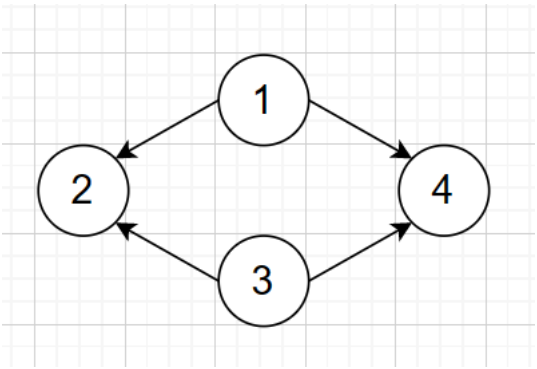

Figura 16: raggiungibilità

il nodo 4 è raggiungibile dal nodo 1 ma non viceversa.

Sappiamo che un grafo non orientato è **connesso** se e solo se esiste un cammino da ogni vertice ad ogni altro vertice. In altre parole: un grafo non orientato si dice connesso se esiste un percorso non interrotto tra qualsiasi coppia di nodi del grafo ed è dunque possibile raggiungere ogni nodo del grafo partendo da qualsiasi altro nodo, attraverso una serie di archi che non si interrompono.

Un grafo $G' = (V', E')$ è una **componente connessa** di G se e solo se è un sottografo di G **fortemente connesso** e **massimale**:

- G' è un **sottografo** di G $(G' \subseteq G)$ se e solo se $V' \subseteq V$ e $E' \subseteq E$; in altre parole, G' è un grafo che deriva da G rimuovendo alcuni dei suoi nodi e archi
- G' è **fortemente connesso**: ciò significa che esiste un percorso diretto tra ogni coppia di nodi distinti in G'. In un grafo orientato, si parla di fortemente connesso se esiste un percorso diretto non interrotto tra qualsiasi coppia di nodi del grafo: in questo caso, è possibile raggiungere ogni nodo del grafo partendo da qualsiasi altro nodo, seguendo solo archi diretti (in pratica, la differenza tra connessione e forte connessione dipende dal fatto che nei grafi orientati gli archi hanno una direzione e quindi è possibile raggiungere un nodo solo seguendo gli archi nella direzione corretta)
- G' è **massimale** se e solo se non esiste un sottografo G'' di G che sia connesso e "più grande" di G', ovvero tale per cui $G' \subseteq G'' \subseteq G$; in altre parole: non esiste un altro sottografo fortemente connesso di G che sia propriamente più grande di G', ovvero non esiste un sottografo fortemente connesso di G che contenga G' come sottoinsieme proprio

Consideriamo il seguente grafo:

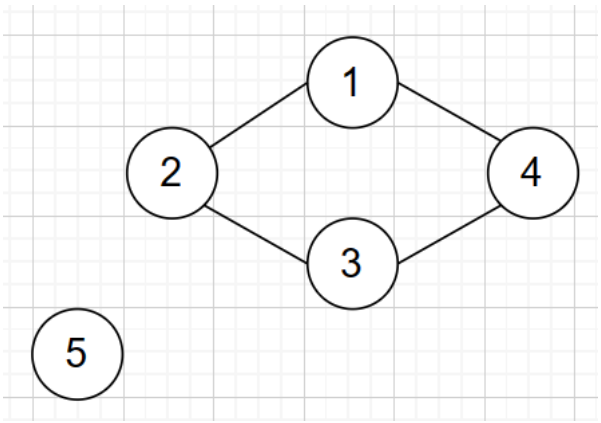

Figura 17: Grafo

Il seguente sottografo G' è fortemente connesso e massimale:

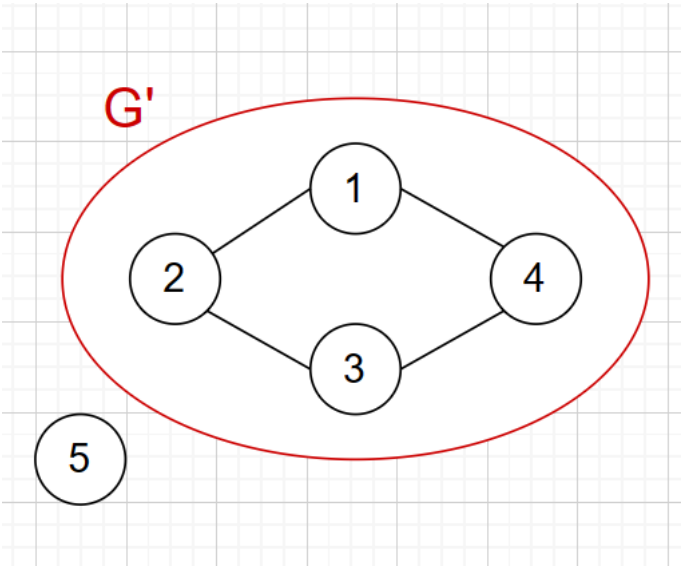

Figura 18: sottografo connesso e massimale

In realtà non essendo un grafo orientato si parla semplicemente di "connessione".

Consideriamo il seguente esempio:

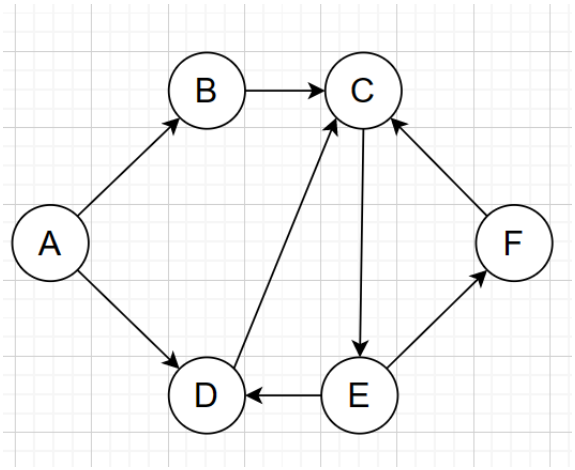

Figura 19: Grafo Orientato

In questo grafo riconosciamo 3 componenti fortemente connesse:

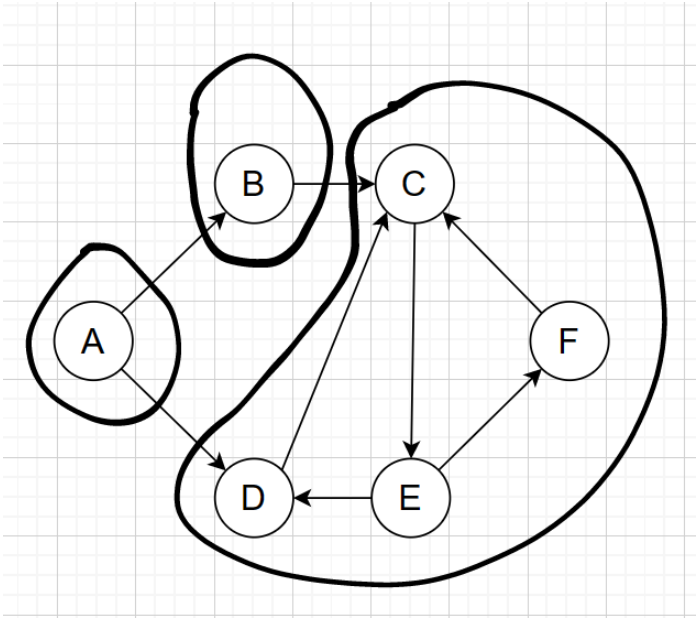

Figura 20: componenti fortemente connesse

A e B, sebbene nodi "singoli" siano ognuno un grafo (con un nodo) connesso e massimale. Il sottografo (C, D, E, F) è una componente fortemente connessa in quanto da ognuno dei nodi posso raggiungere tutti gli altri (del sottografo).

Consideriamo il problema della verifica se un grafo non orientato è connesso e dell'identificazione delle componenti connesse di cui è composto.

Un grafo è connesso se, al termine della DFS, tutti i nodi sono stati marcati, altrimenti, una singola passata non è sufficiente e la visita deve ripartire da un nodo non marcato, scoprendo una nuova porzione del grafo.

Al fine di identificare le componenti connesse abbiamo bisogno di

- Vettore id degli identificatori di componente
- $id[u]$ è l'identificatore della componente connessa a cui appartiene u

Di seguito l'algoritmo cc (connected components):

```
int[] cc(Graph G)
     int[] id = new int[G.V()]
     foreach u in G.V()
          id[u] = 0
     int counter = 0
     foreach u in G.V()
          if id[u] == 0
               counter += 1
               ccdfs(G, counter, u, id)
```

```
     return id

ccdfs(Graph G, int counter, Node u, int[] id)
     id[u] = counter
     foreach v in G.adj(u)
          if id[v] == 0
               ccdfs(G, counter, v, id)
```

Ecco di seguito l'applicazione dell'algoritmo:

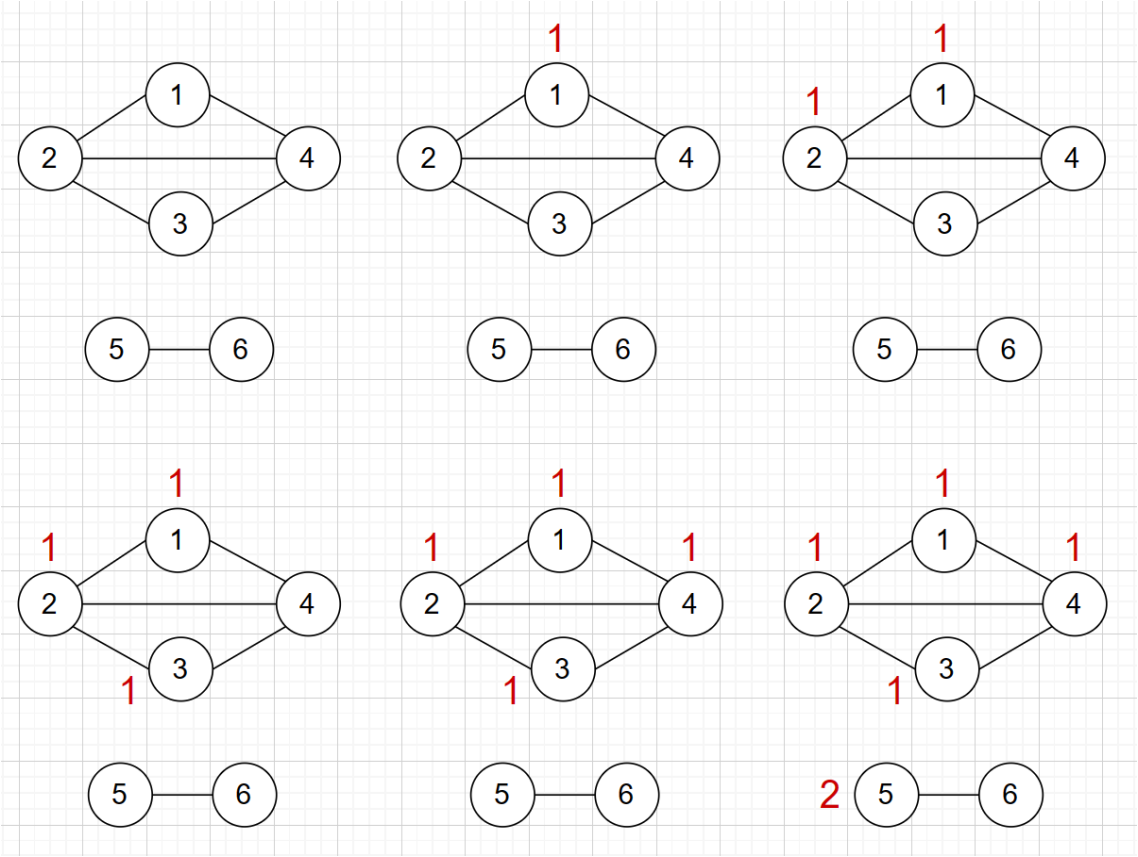

Figura 21: Calcolo delle componenti connesse

Il risultato finale è il seguente:

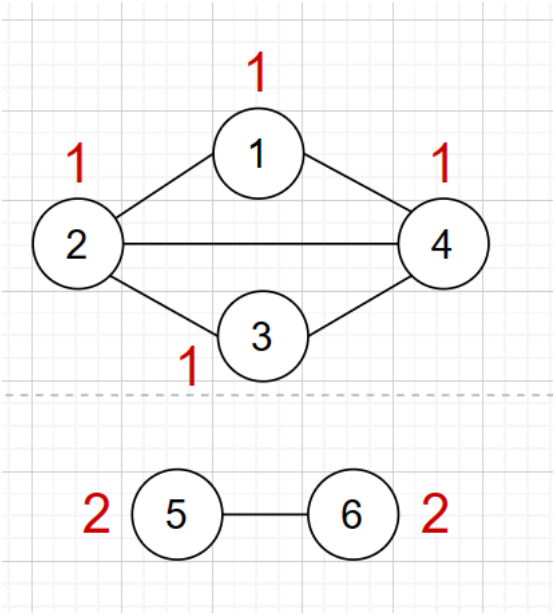

Figura 22: componenti connesse

Tuttavia, questo algoritmo ha un problema: il risultato dipende dal nodo di partenza.

Per superare questo problema occorre introdurre un nuovo algoritmo: **Kosaraju**.

Di seguito i passi dell'algoritmo:

- eseguire una visita DFS del grafo G
- calcolo del grafo trasposto G_t
- eseguire una visita DFS sul grafo G_t utilizzando l'algoritmo precedente cc, esaminando i nodi nell'ordine inverso di tempo di fine della prima visita
- le componenti connesse (e i relativi alberi DF) rappresentano le componenti fortemente connesse di G

Quindi l'algoritmo da implementare (scc = strongly connected components) è il seguente:

```
int[] scc(Graph G)
      Stack S = topological_sort(G)   // prima visita
      GT = transpose(G)          // grafo trasposto
      return cc(GT,S)            // seconda visita
```

Di fatto:

- topological_sort genera uno stack con "i nodi nell'ordine inverso di tempo di fine della prima visita"
- mediante la funzione di trasposizione ottengo il grafo trasposto (con gli archi invertiti)

- la funzione cc (connected components) rispetto a quella vista in precedenza ha in ingresso lo stack che determina in che ordine effettuare la visita (quindi anziché usare un foreach sui nodi, li visita estraendoli dallo stack)

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione;
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione;
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley;
- Sedgewick: Algoritmi in C, Pearson, 2015;
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.