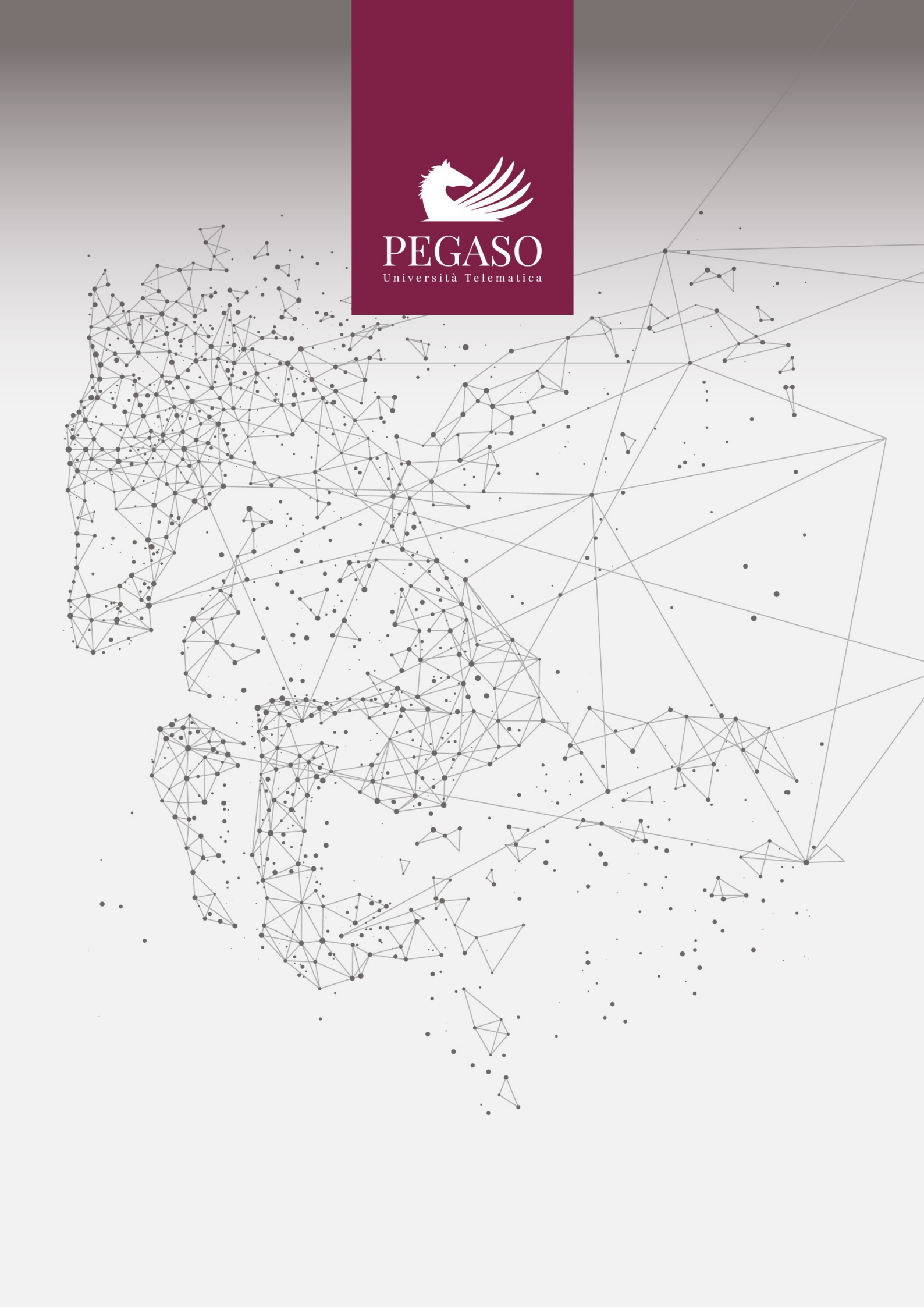
PEGASO
Università Telematica

# Indice

# 1. Visita in profondità

La visita in profondità (depth first search, DFS) è spesso una "subroutine" della soluzione di altri problemi; è utilizzata per coprire l'intero grafo e non solo i nodi raggiungibili da una singola sorgente (diversamente da BFS).

La BFS esplora tutti i nodi raggiungibili dal nodo di partenza, ma solo quelli delle componenti connesse che contengono il nodo di partenza, e non esplora necessariamente tutti i nodi del grafo contemporaneamente quando il grafo è non connesso: se dunque il grafo è una foresta, la BFS esplora solo la porzione di grafo raggiungibile da nodo di partenza. Ad esempio, supponiamo di avere un grafo non connesso con due componenti connesse, e di eseguire la BFS partendo da un nodo nella prima componente: la BFS esplorerà tutti i nodi della prima componente connessa, ma non esplorerà i nodi della seconda componente connessa, a meno che non si esegua una nuova BFS partendo da un nodo della seconda componente.

La DFS invece, partendo da un nodo qualsiasi, esplora l'intera componente connessa che contiene quel nodo, e poi passa alla successiva, fino a esplorare tutte le componenti. Quindi la DFS esplora una componente connessa alla volta, e per ogni componente connessa genererà un proprio albero DF (albero depth-first) separato, che coprirà solo i nodi di quella componente.

Occorre tuttavia fare attenzione: la visita DFS genera di fatto una **foresta DF** (depth-first) $G_f = (V, E_f)$, cioè una **collezione di alberi DF**. Si parla di "collezione" perché di fatto non esiste un solo modo di visitare in profondità l'albero e questo dipende da dove parto e come scelgo di volta in volta i "vicini" dei nodi da visitare; inoltre ogni visita può generare più alberi, uno per ogni componente connessa.

In altre parole: ogni grafo può essere visitato in DFS in diversi modi e pertanto la visita DFS determina una foresta DFS; se poi il grafo non è connesso ogni visita non determinerà un solo albero DF ma più alberi DF, uno per ogni componente connessa del grafo; ogni albero DF nella foresta può essere un albero di copertura solo per la componente connessa che rappresenta.

La struttura dati che può essere utilizzata per implementare tale visita può essere una pila esplicita o implicita (cioè attraverso la ricorsione).

Di seguito l'implementazione DFS in pre-ordine:

```
DFS_preorder(grafo G, nodo r, boolean[] visited)
visited[u] = true
{visita il nodo u}
foreach v in G.adj(u)
     if !visited[v]
          {visita l'arco (u,v)}
          DFS_preorder(G, v, visited)
```

Di seguito l'implementazione DFS in post-ordine:

```
DFS_postorder(grafo G, nodo r, boolean[] visited)
visited[u] = true
foreach v in G.adj(u)
     if !visited[v]
          {visita l'arco (u,v)}
          DFS_postorder(G, v, visited)
{visita il nodo u}
```

La complessità della visita è: $O(m + n)$

Eseguire una DFS basata su chiamate ricorsive può essere rischioso in grafi molto grandi e connessi: è possibile che la profondità raggiunta sia troppo grande per la dimensione dello stack del linguaggio; in tali casi, si preferisce utilizzare una BFS oppure una DFS basata su stack esplicito.

In generale, la dimensione dello stack è determinata dal sistema operativo e dalla configurazione del compilatore. Tuttavia, la dimensione dello stack può anche essere influenzata dal modo in cui il linguaggio di programmazione gestisce le chiamate di funzione e le variabili locali.

Ad esempio, alcuni linguaggi di programmazione utilizzano una memoria di allocazione automatica (come C e C++) per le variabili locali, che vengono memorizzate nello stack: in questo caso, la dimensione dello stack può essere influenzata dal numero e dalle dimensioni delle variabili locali utilizzate nel codice.

Altri linguaggi di programmazione, come Java, utilizzano invece una gestione della memoria a livello di oggetto: in questo caso, le informazioni relative alle chiamate di funzione e alle variabili locali vengono memorizzate nell'heap, che ha dimensioni diverse rispetto allo stack.

Di seguito sono riportate alcune informazioni sulle dimensioni dello stack per alcune piattaforme comuni:

- In **Windows**, lo stack ha una dimensione predefinita di 1 MB; tuttavia, questa dimensione può essere aumentata o diminuita utilizzando la funzione $SetThreadStackGuarantee$. Inoltre, la dimensione massima dello stack per un processo può essere specificata durante la creazione del processo;

- In **Linux**, la dimensione dello stack predefinita è di 8 MB; tuttavia, è possibile modificare la dimensione dello stack utilizzando il comando $ulimit\ -s$;
- In **macOS**, la dimensione dello stack predefinita è di 8 MB; come in Linux, la dimensione dello stack può essere modificata utilizzando il comando $ulimit\ -s$;
- In **iOS**, la dimensione dello stack predefinita varia in base al dispositivo e alla versione del sistema operativo; ad esempio, su iPhone 11 con iOS 13, la dimensione dello stack è di circa 2 MB;
- In **Android**, la dimensione dello stack predefinita varia in base alla versione del sistema operativo e alla dimensione del dispositivo; in generale, la dimensione dello stack è di circa 2 MB.

Di seguito l'implementazione della DFS iterativa in pre-ordine:

```
DFS_iterative_preorder(grafo G, nodo r)
stack S = new stack()
S.push(r)
boolean[] visited = new boolean[G.V()]
foreach u in G.V()-{r}
     visited[u] = false;
while not S.isEmpty()
     node u = S.pop()
     if !visited[u]
          {visita il nodo u}
          visited[u]=true
          foreach v in G.adj(u)
               {visita l'arco (u,v)}
               S.push(v)
```

In questa procedura, un nodo può essere inserito nella pila più volte: il controllo se un nodo è già stato visitato viene fatto all'estrazione e non all'inserimento.

La complessità è $O(m + n)$:

- $O(m)$ visite degli archi
- $O(m)$ inserimenti, estrazioni
- $O(n)$ visite dei nodi

Nella visita in pre-ordine, i nodi vengono visitati in ordine "pre-ordine" (ovvero, il nodo radice viene visitato per primo, seguito dai suoi figli), mentre nella visita in post-ordine, i nodi vengono visitati in ordine "post-ordine" (ovvero, i figli di un nodo vengono visitati prima del nodo stesso).

```
 def dfs(graph, start):
    visited = set()
    stack = [start]
    visited.add(start)

    while stack:
        vertex = stack.pop()
        print(vertex, end=' ')

        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append(neighbor)


def dfs_recursive(graph, start):
    visited = set()
    dfs_visit(graph, start, visited)
    return visited

def dfs_visit(graph, vertex, visited, parent=None):
    visited.add(vertex)
    for neighbor in graph[vertex]:
        if neighbor not in visited:
            if dfs_visit(graph, neighbor, visited, vertex):
                return True
        elif neighbor != parent:
            return True
    return False
```

# 2. 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
```

Analizziamo l'algoritmo sul seguente grafo:

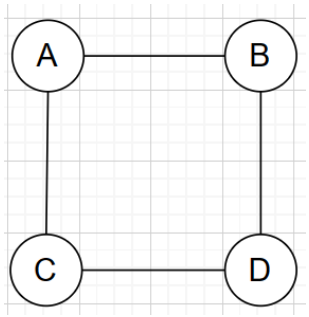


**Figura 1:** analisi DFS

Inizialmente il discovery ed il finish sono pari a 0 per ogni nodo, ed il contatore è anch'esso pari a 0. Inserisco per ogni nodo la coppia $(d, f)$ nel grafo:

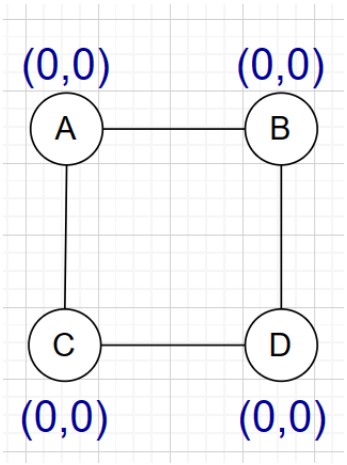


**Figura 2:** analisi DFS con d ed f espliciti su ogni nodo

Supponiamo che $A$ sia la radice. I passi da seguire sono i seguenti:

- Visitiamo il nodo $A$
- Incrementiamo il contatore del tempo di scoperta a 1 ($counter = 1$) e lo assegniamo al valore $d[A] = 1$
- Esaminiamo i nodi adiacenti a $A$ e incontriamo $B$ e $C$. Visitiamo l'arco $(A, B)$ e l'arco $(A, C)$ in qualsiasi ordine

- Essendo $d[B]$ e $d[C]$ ancora pari a 0, visitiamo entrambi gli archi $(A, B)$ e $(A, C)$ come archi dell'albero e chiamiamo ricorsivamente la funzione $DFS\_SCHEMA$ su $B$ e $C$.

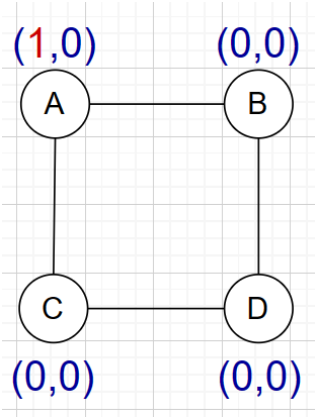


**Figura 3:** d(A)=1

- Iniziamo la visita del nodo $B$, incrementiamo il contatore del tempo di scoperta a 2 ($counter = 2$) e lo assegniamo al valore $d[B] = 2$

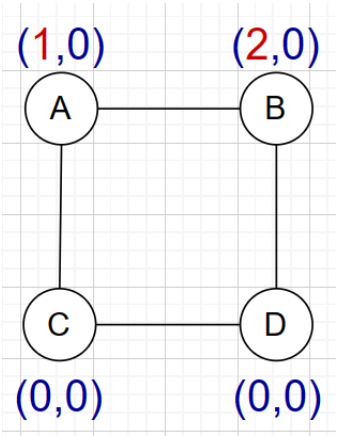


**Figura 4:** d(B)=2

- Esaminiamo i nodi adiacenti a B e incontriamo A e D
- Iniziamo con $A$: visitiamo l'arco $(B, A)$; $d[B] > d[A]$ ed essendo $f[B] = 0$ l'arco è all'indietro e non va inserito nell'albero
- Proseguiamo con D: l'arco $(B, D)$ è invece un arco dell'albero, poiché $d[A]$ non è 0 e $d[D] = 0$; chiamiamo ricorsivamente la funzione $DFS\_SCHEMA$ su $D$
- Iniziamo la visita del nodo $D$, incrementiamo il contatore del tempo di scoperta a 3 ($counter = 3$) e lo assegniamo al valore $d[D] = 3$

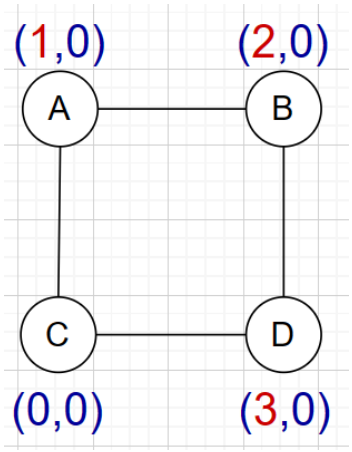


**Figura 5:** d(D)=3

- Esaminiamo i nodi adiacenti a D e incontriamo B e C
- Iniziamo con $B$: visitiamo l'arco $(D,B)$; $d[D] > d[B]$ ed essendo $f[D] = 0$ l'arco è all'indietro e non va inserito nell'albero
- Proseguiamo con C: l'arco $(D,C)$ è invece un arco dell'albero, poiché $d[D]$ non è 0 e $d[C] = 0$; chiamiamo ricorsivamente la funzione $DFS\_SCHEMA$ su $C$
- Iniziamo la visita del nodo $C$, incrementiamo il contatore del tempo di scoperta a 4 ($counter = 4$) e lo assegniamo al valore $d[C] = 4$

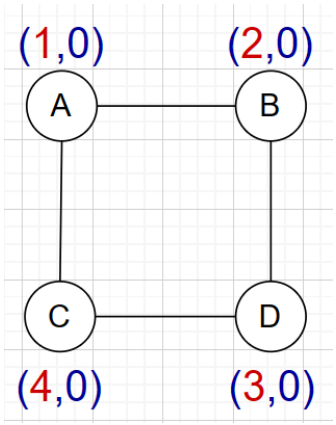


**Figura 6:** d[C]=4

- Esaminiamo i nodi adiacenti a C e incontriamo A e D
- Iniziamo con $A$: visitiamo l'arco $(C,A)$; $d[C] > d[A]$ ed essendo $f[A] = 0$ l'arco è all'indietro e non va inserito nell'albero
- Proseguiamo con $D$: visitiamo l'arco $(C,D)$; $d[D] > d[C]$ ed essendo $f[C] = 0$ l'arco è all'indietro e non va inserito nell'albero

- Terminiamo così la visita di $C$: incrementiamo il contatore ($counter = 5$) ed assegno la fine a $C$: $f[C] = 5$

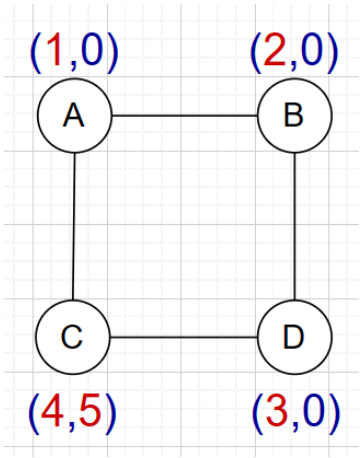


**Figura 7:** f[C]=5

- A questo punto di prosegue a ritrovo fino ad avere:

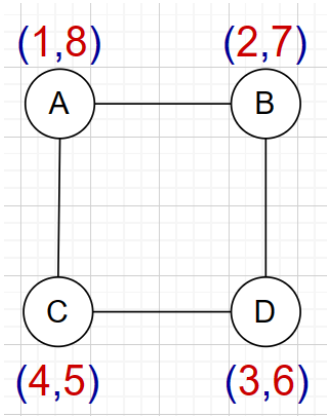


**Figura 8:** schema finale

# 3. Esempi – albero di copertura DF

Consideriamo ora il seguente grafo:

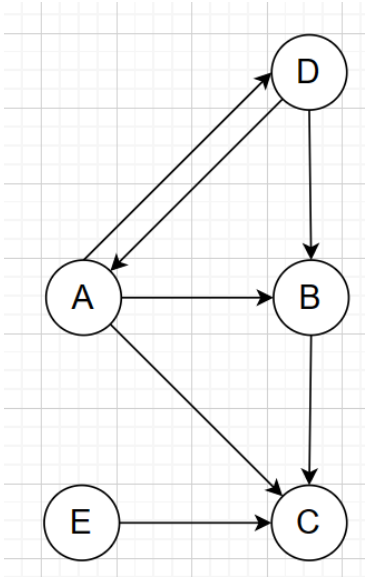


**Figura 9:** digrafo

Partiamo dalla radice $A$. I nodi adiacenti di $A$ sono $B$, $C$ e $D$.

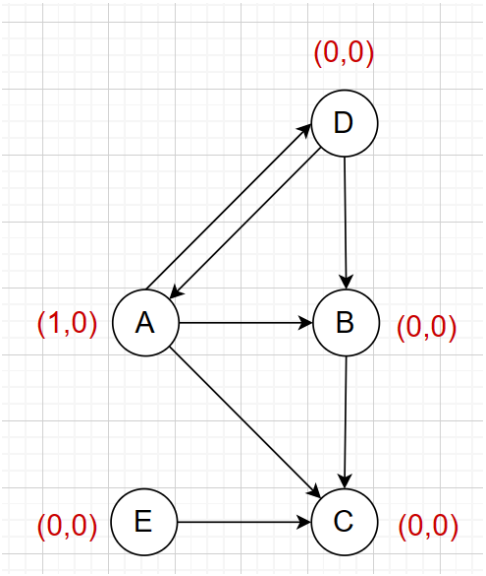


**Figura 10:** d[A]=1

Procediamo in ordine alfabetico, pertanto iniziamo con $B$. Incrementiamo il contatore a 2, assegniamo il discovery a $B$:

- $counter = 2$
- $d[B] = 2$

Vi è solo un nodo adiacente a $B$, cioè $C$ pertanto procedo direttamente con questo. Incrementiamo il contatore a 3, assegniamo il discovery a $C$:

- $counter = 3$
- $d[C] = 3$

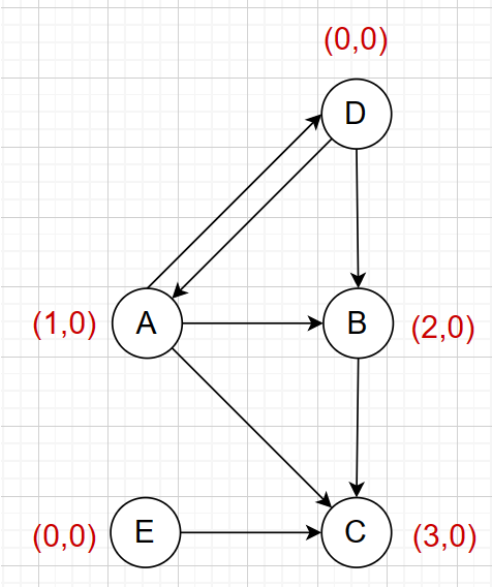


**Figura 11:** d[C]=3

Non vi sono nodi "vicini" a $C$ pertanto posso incrementare il counter e settare il suo finish time:

- $counter = 4$
- $f[C] = 4$

Procedo a ritroso con $B$ considerando gli eventuali altri vicini di $B$ che però non ci sono; pertanto, posso incrementare il counter e settare il suo finish time:

- $counter = 5$
- $f[B] = 5$

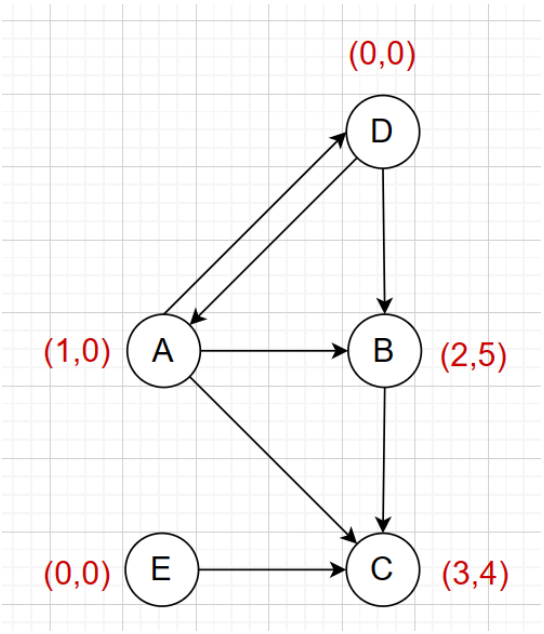


**Figura 12:** Finish time su C e B

Procedo a ritroso con $A$ considerando gli altri vicini ancora da analizzare e cioè $C$ e $D$. Procedo in ordine alfabetico e quindi analizzo $C$. Visito l'arco $(A, C)$ e stavolta il nodo di destinazione è già stato visitato con $d[C] > d[A]$ e $f[C]! = 0$.

Come indicato dall'algoritmo:

```
else if d[u] < d[v] and f[v] != 0
     { visita l'arco (u, v) (arco in avanti) }
```

Quindi l'arco $(A, C)$ punta ad un "discendente" di $A$ ed è dunque un "arco in avanti": non "visito" dunque il nodo di destinazione e procedo con l'arco $(A, D)$. Incrementiamo il contatore a 6, assegniamo il discovery a $D$:

- $counter = 6$
- $d[D] = 6$

I nodi adiacenti di $D$ sono $A$ e $B$. Procediamo in ordine alfabetico, pertanto iniziamo con $A$. In questo caso abbiamo che $d[D] > d[A]$ ed $f[A] = 0$.

Come indicato dall'algoritmo:

```
else if d[u] > d[v] and f[v] == 0
     { visita l'arco (u, v) (arco all'indietro) }
```

Quindi l'arco $(D, A)$ punta ad un "antenato" di $A$ (va dal discendente $D$ all'antenato $A$) ed è dunque un "arco all'indietro": non "visito" dunque il nodo di destinazione e procedo con l'arco $(D, B)$.

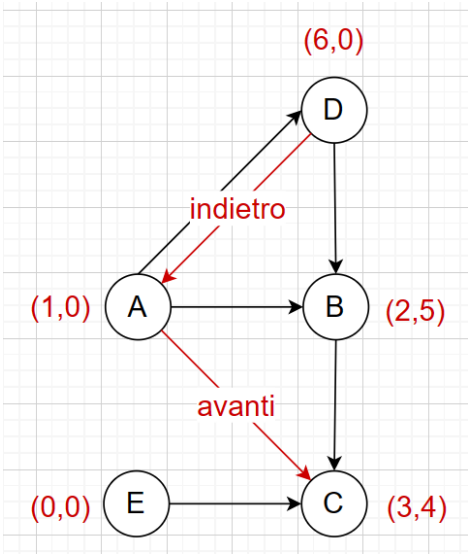


**Figura 13:** Analisi arco (D, B)

Analizzando l'arco $(D, B)$ mi rendo conto che $d[D] > d[B]$ tuttavia $f[B]! = 0$ e quindi finisco nella casistica di arco di attraversamento (i nodi $D$ e $B$ sono di fatto fratelli, non essendo nessuno dei 2, antenato o discendente dell'altro).

Non vi sono nodi "vicini" a $D$ da visitare pertanto posso incrementare il counter e settare il suo finish time:

- $counter = 7$
- $f[D] = 7$

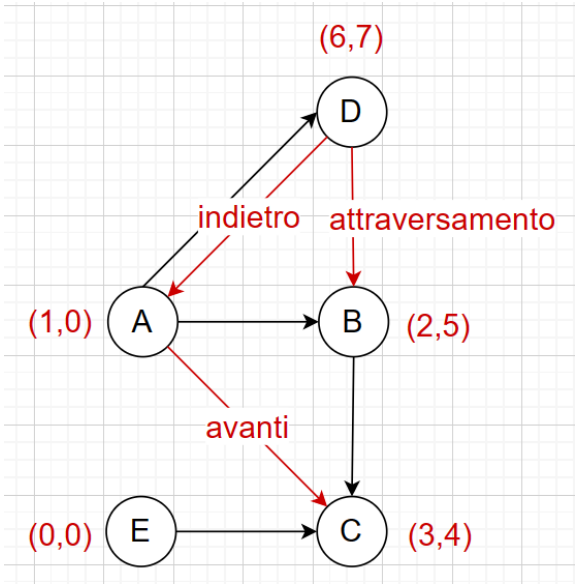


**Figura 14:** f[D]=7

Procedo a ritroso su $A$ e non vi sono nodi "vicini" a $A$ da visitare pertanto posso incrementare il counter e settare il suo finish time:

- $counter = 8$
- $f[A] = 8$

A questo punto la visita riparte con il prossimo nodo "disponibile" che è il nodo $E$ e che parte, dunque, dal contatore incrementato a 9. L'unico vicino di $E$ è il nodo $C$ ma ci troviamo nuovamente nella situazione in cui $d[E] > d[C]$ ma $f[C]! = 0$ pertanto l'arco $(E, C)$ è di attraversamento e dunque si finisce con il settaggio del finish time di $E$.

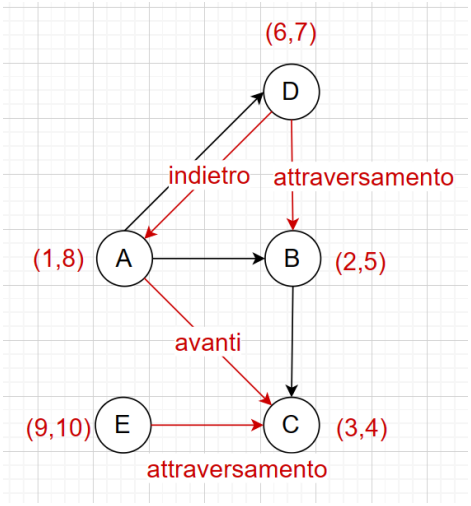


**Figura 15:** f[E]=10

Riportiamo di seguito un altro esempio:

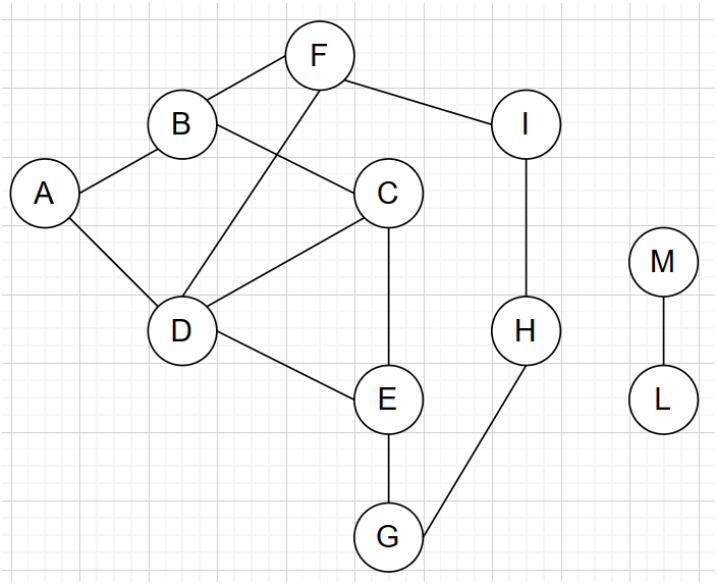


**Figura 16:** Grafo di analisi

Consideriamo $F$ come nodo iniziale; si procede finchè non si arriva alla seguente situazione:

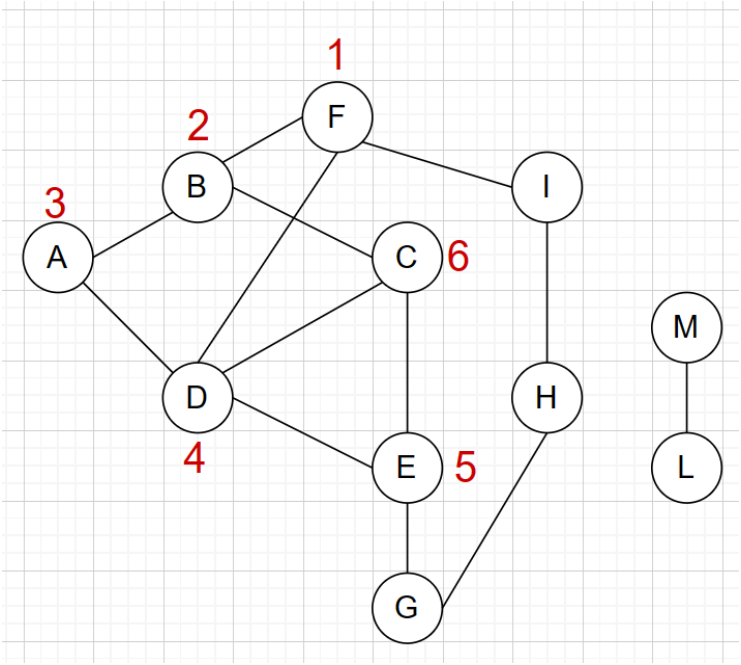


**Figura 17:** Visita del grafo in DFS

Siamo giunti a:

- $counter = 6$
- $d[C] = 6$

Gli archi adiacenti ad $C$ sono $B$ ed $E$; supponiamo di procedere con $B$. Esaminiamo dunque l'arco $(C, B)$; in questo caso $d[B] = 2$ quindi ci troviamo nel caso:

```
else if d[u] > d[v] and f[v] == 0
     { visita l'arco (u, v) (arco all'indietro) }
```

si procede con l'esaminare l'arco $(C, B)$ (arco all'indietro) ed eseguo le 2 istruzioni:

counter = counter + 1;

f[u] = counter

Pertanto:

- $counter = 7$
- $f[C] = 7$

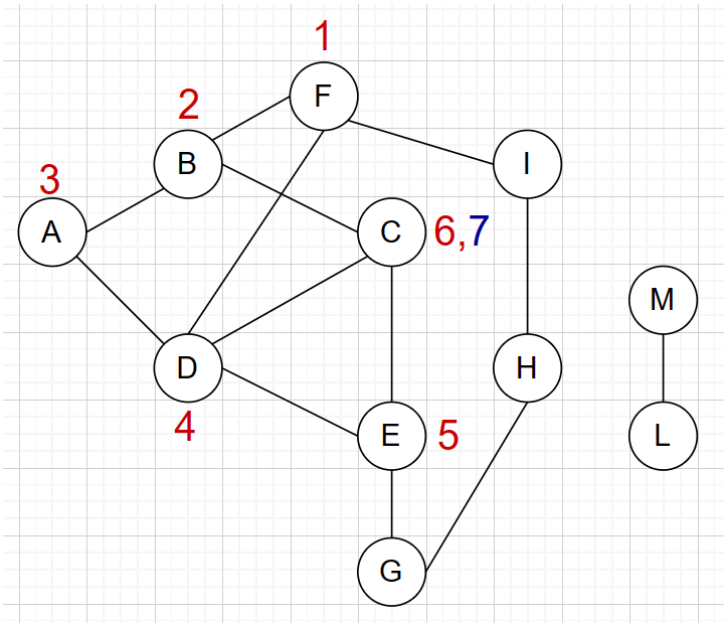


**Figura 18:** DFS: settaggio del finish in C

Si prosegue con E fino ad arrivare al seguente scenario:

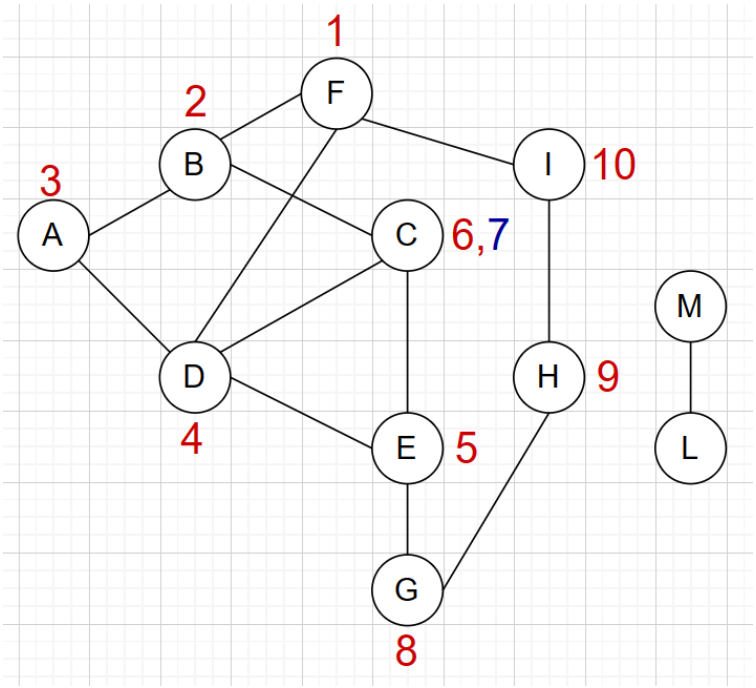


**Figura 19:** d[I]=10

Seguendo il ragionamento precedente si procede con il settaggio dei finish:

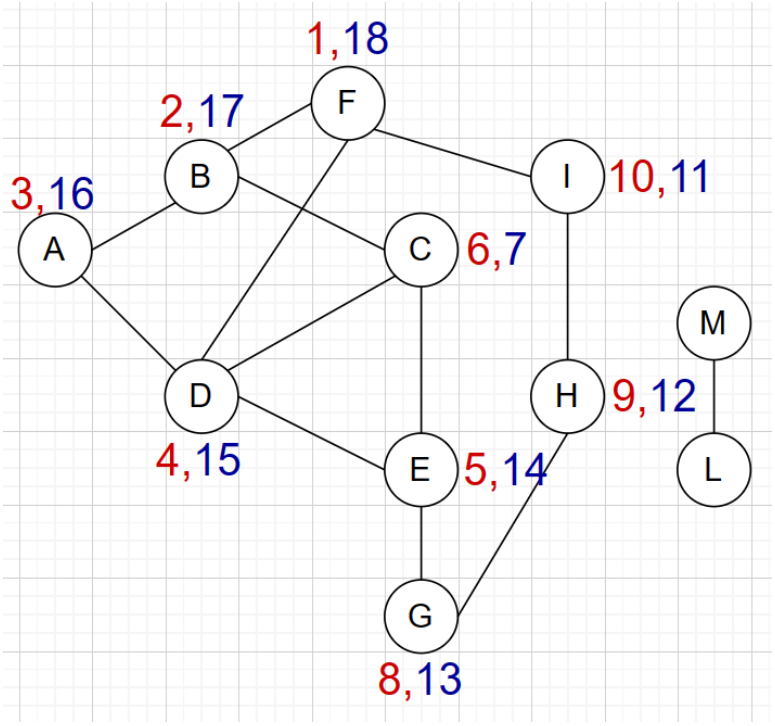


**Figura 20:** f[F]=18

A questo punto si può terminare con gli ultimi 2 nodi:

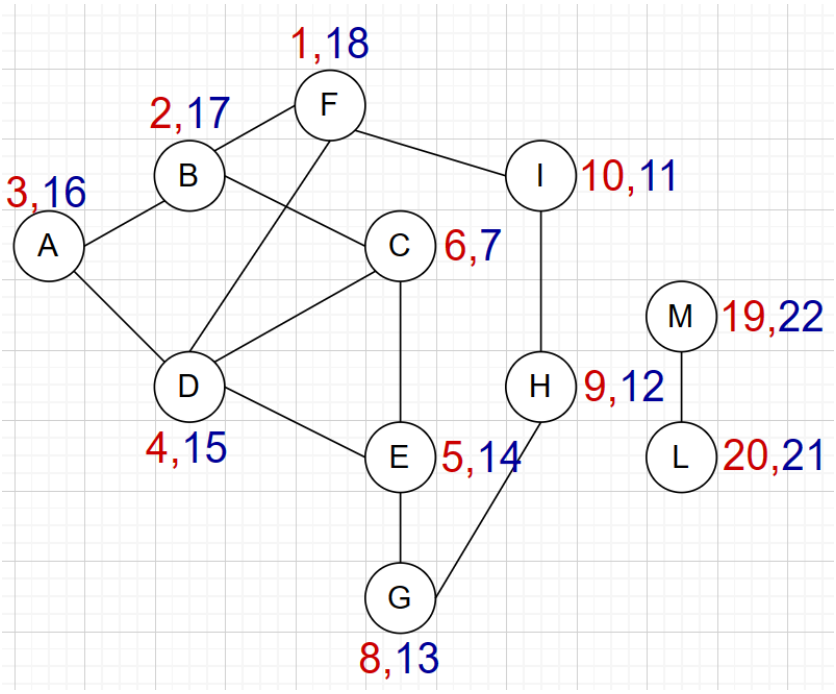


**Figura 21:** DFS finale

# 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.