VISITE:

- **BFS**: visita per cercare il cammino di lunghezza minima da un nodo di partenza in un grafo NON PESATO.

  UTILIZZA: vettore distanze, predecessori, colori. → **LIVELLI**

  PROCEDIMENTO:

  - coloro un nodo di NERO quando ho colorato di grigio tutti i suoi ADIACENTI

  - memorizzo solo gli archi che mi fanno scoprire qualcosa, ovvero GRIGIO-BIANCO, con coda FIFO

  - Al termine di BFS(G,s), il vettore π contiene i predecessori nel cammino minimo tra s e i vari nodi

  COMPLESSITÀ:

  con Mg = $\Theta(|V|^2)$
  con Lg = $O(|V| + |E|)$

  PSEUDOCODICE:

CODICE:

```
BFS (G,s){          % G grafo S nodo      → ti esplora una sola comp. CONNESSA e ti calcola le distanze da un nodo
  INIZIALIZZAZIONE
  for (each u ∈ V){ % inizializzo                      ⎤
      color [u] ← B                                    ⎥ Θ(|V|)
      d [u] ← +∞                                       ⎥
      π [u] ← NIL                                      ⎥
  }                                                    ⎦
  Q ← ∅     % coda                                     ⎤
  color [s] ← G  % setto il colore grigio al nodo di partenza  ⎥ Θ(1)
  d[s] ← 0   % s dista 0 da se stesso                  ⎥
  Enqueue (Q,s)  % inserisco s in coda                 ⎦
  While (Q ≠ ∅){  O(|V|) volte il while (tutti i nodi della comp. connessa di s)
      u ← head(Q)            % estraggo in testa alla coda
                  ADIACENZE
      For (each v ∈ Adj(u)){  O(|V|) volte il for        ⇒ sembra essere O(|V|²)
          if (color[v] = B){   % se il nodo adiacente è bianco
              color[v] ← G     % GRIGIO
              d[v] ← d[u]+1    % la distanza del nodo adiacente da s è distanza di u+1
              π[v] ← u         % salvo in π[v] il suo "parent" che ti da il cammino minimo
              Enqueue(Q,v)
          }
      color[u] ← N   % ho finito di esplorare gli adiacenti di u, lo coloro di NERO
      Dequeue (Q)    % e lo rimuovo dalla LISTA
  }
}
```

→ arrivo solo ai nodi raggiungibili da s, non è detto che visiti tutti i nodi

- **DFS** : seguo un cammino in profondità' e faccio backtracking quando serve.
  cerca proprietà globali, es. cicli o topological sort

  UTILIZZA: vettore distanze, predecessori, colori, vettori dei tempi → **PROFONDITÀ**

  PROCEDIMENTO:

  i vettori dei tempi mi segnano quando un nodo passa da bianco-grigio (inizio[]) e da grigio-nero (fine[])

  COMPLESSITÀ:

  con Mg = $\Theta(|V|^2)$
  con Lg = $\Theta(|V| + |E|)$

  PSEUDOCODICE:

```
DFS(G){
    for(each v ∈ V){
        color[v] ← BIANCO          Θ(|V|)
        π[v] ← NIL
    }
    TIME ← 1
    % se mi sono rimasti nodi bianchi, la visita può ripartire
    for(each v ∈ V){
        if(color[v] = B){
            TIME ← DFS_Visit(G,v,TIME)
        }   └ con la 1ª chiamata esploro tutta la COMP. CONNESSA,
    }         se rimane qualche nodo fuori (NON CONNESSO), richiamo su di lui
}
```

```
DFS_Visit(G,v,TIME){
    color[v] ← GRIGIO
    i[v] ← TIME                    Θ(1)
    TIME ++
    for(each u ∈ Adj[v]){          % come la post-order, prima vado fino in fondo,
                                     poi risalgo
        if(color[u] = BIANCO){
            π[u] ← v
            TIME ← DFS_visit(G,u,TIME)   Θ(|V|)
        }   └ richiamata dal for principale (DFS) solo una volta sarebbe O(|V|), ma alla fine
              sarà chiamata per tutti i nodi → Θ(|V|)
    }
    f[v] ← TIME
    TIME++                         Θ(1)
    color[v] ← NERO
    return TIME
}
```

$\Theta\left(\sum_{v \in V} |Adj[v]|\right)$

COMPLESSITÀ: $\Theta(|V|) + \Theta(|V|) + \Theta\left(\sum_{v \in V} |Adj[v]|\right)$

operaz $\Theta(1)$

quanto costa con MG / LG

Teorema del cammino bianco : se da v ad a il cammino e’ tutto bianco allora a e’ discendente di v

Archi:

- **GRIGIO-GRIGIO**: back edge, sono dei **cicli**. Il numero di back edge e di cicli non coincide per forza, infatti i back edge sono O(n^2) mentre i cicli sono O(2^n)

- **GRIGIO-NERO**: possono essere **CROSS-EDGE** (se i vettori dei tempi sono scollegati) o **FORWARD-EDGE** (vettori dei tempi compresi)

Proprietà: ogni arco aciclico ha almeno un nodo senza archi uscenti e un nodo senza archi entranti.

TOPOLOGICAL SORT:

- se il grafo e’ **aciclico**: quando faccio la DFS(G) ogni volta che un nodo diventa nero vuol dire che non ha archi uscenti verso altri nodi percio’ lo inserisco in una coda e procedo iterativamente cosi togliendo sempre gli archi neri → Θ(|V| + |E|)

- se il grafo e’ **ciclico**: devo trovare le componenti fortemente connesse (massimo numero di nodi mutuamente raggiungibili) → grafo Gscc e poi diventa aciclico.
  Dato che con DFS(G) controllo solo gli archi di un verso quindi e’ possibile che non tutti gli alberi di fine visita rappresentino le componenti connesse, potrebbero essercene di più anche dentro un singolo albero.

  Il nodo con tempo di fine visita maggiore si trova in una scc senza archi entranti.
  → algoritmo di **KOSARAJU**

ALGORITMO DI KOSARAJU: per trovare le componenti s.c.c in un grafo orientato

PROCEDIMENTO:

- eseguo DFS(G) e così mi salvo il vettore dei tempi di fine visita (in ordine decrescente)
- calcolo DFS(G^{-1}) (invertendo gli archi del grafo originale) e nel for prelevo gli archi dal vettore dei tempi di fine visita di DFS(G)

COMPLESSITÀ: con Lg = Θ(|V| + |E|)

GRAFI NON ORIENTATI

alberi (liberi) : grafo non orientato, aciclico e connesso (ha una sola componente connessa) e non ha una root precisa

PROPRIETÀ:

1 - G è un albero (connesso aciclico)

2 - $\forall u, v \in V$ $\exists$ un UNICO CAMMINO tra u e v

3 - G è CONNESSO e se viene rimosso un arco qualsiasi, G si sconnette

4 - G è CONNESSO e #archi = #nodi - 1 => $|E| = |V| - 1$

5 - G è ACICLICO e $|E| = |V| - 1$

6 - G è ACICLICO e aggiungendo un qualsiasi arco G diventa CICLICO

per testare se G e’ un albero basta verificare una di queste 6 proprieta.

MINIMUM SPANNING TREE (nei grafi **NON ORIENTATI**, CONNESSI e **PESATI**)

sottografo minimo, lo scheletro che connette tutti i nodi con peso minimo.

funzione peso $W(G) = \sum_{\{x,y\} \varepsilon E} W(\{x, y\})$ la somma dei pesi di tutti gli archi in G

non esiste un unico MST.

DEFINIZIONI:

arco **SAFE-EDGE**: dato MST T=(V, E' ,W) di G=(V,E,W) e $A \subseteq E' \subseteq E$ sottoinsieme di una soluzione possibile, allora {x,y} e’ safe per A se $A \cup \{x, y\}$ e’ sottoinsieme di una possibile soluzione, ovvero $A \subseteq E''$ di un possibile MST T=(V, E'' ,W)

la parte difficile e’ trovare gli archi safe.

TAGLIO: e’ una partizione di V in 2 blocchi, (S, V/S) e {x,y} **attraversa il taglio** se $x \in S,\ \ y \in S/V$ .Un insieme A **rispetta un taglio** se ogni arco di A non attraversa il taglio.

arco **LEGGERO**: {u,v} e' leggero per il taglio (S,V/S) se {u,v} attraversa il taglio ed e' di peso minimo tra tutti quelli che lo attraversano.

TEO: Dato G=(V,E,W) , $A \subseteq T\ MST\ di\ G$, dato {u,v} e dato (S,V/S).
Se **A rispetta il taglio** e **{u,v}** e' arco **leggero**, allora **{u,v} e' safe** per A

LEMMA: Dato T SPANNING Tree di G e {x,y}$\notin$T e {u,v}$\in T$ e {u,v} si trova sul cammino x-y
allora $T' = (T \setminus \{x,y\}) \ \cup \{\{x,y\}\}$ e' SPANNING TREE di G

COR: Dato $A \subseteq T\ MST$ e {u,v} e' di peso minimo tra tutti gli archi che collegano le comp.connesse allora $A \cup \{\{x,y\}\} \ \subseteq T'$ MST

esistono due algoritmi per calcolare MST di G : - KRUSKAL - PRIM

ALGORITMO DI KRUSKAL (utilizza il COROLLARIO)

A e' inizialmente vuoto, quindi ogni nodo di G e' una componente connessa distinta (MAKE).

Iterativamente aggiungo ad A un arco di peso minimo tra tutti gli archi del grafo evitando gli archi che ho già' messo in A e gli archi che collegano due nodi già' connessi (UNION).

PROCEDIMENTO:

- Ordino gli archi in ordine CRESCENTE
- Li scandisco in ordine di peso
- Arrivo all'arco {x,y} e devo valutare se x e y sono già' connessi → **insiemi disgiunti**

COMPLESSITÀ: $O(|E| * log(|E|))$

PSEUDOCODICE:

```
CODICE:  Kruskal( G=(V,E,W)){   % G connesso => |E| >= |V|-1
           Sort(E,W)   % es. con HeapSort  O(|E| log|E|)
           A <- Ø
           For each (v ∈ V){
               MAKE(v)   % creo le singole comp. connesse
           }
                            secondo l'ordinamento
           For (each {u,v} ∈ E){
                              RAPPRESENTANTE
              if ( FIND(u) ≠ FIND(v)){ % se hanno rappr. diversi non sono della stessa comp.
                 A <- A ∪ {{u,v}}
                 UNION(u,v)
              }
           }
           return A
        }
```

costo M/U/F con UNIONBYRANK e PATH C.

costi: $O(|E| \log|E|) + O(|E| \cdot \alpha(|V|,|E|)) = O(|E| \log|E|)$ e $|E| \leq |V|^2 \Rightarrow O(|V|^2 \log|V|^2) = O(|E| \log|V|)$

Se G ha più MST possibili allora in G devono essere presenti archi con lo stesso peso.

ALGORITMO DI PRIM (utilizza il TEO)

Utilizzo due array, in ogni istante devo sapere quali archi attraversano il taglio e qual è' di peso minimo → **per ogni nodo memorizzo qual è il nodo che lo collega al MST e quanto pesa il loro arco**.

$\pi[v]$ = nodo che collega v al MST

$key[v]$ = peso dell'arco che collega v al MST

PROCEDIMENTO:

Inizialmente $\forall v\ \pi[v] = NIL$ e $key[v] = \infty$

- In ogni istante devo trovare v come $key[v]$ minima tra i nodi restanti → **minHeap**
- Controllo se vanno modificate le info per i nodi in Adj[v]

PSEUDOCODICE:

CODICE:

```
Prim ( G = (V,E,W), s){
    For (each v ∈ V){          Θ(|V|)
        π[v] ← NIL
        Key[v] ← ∞
    }
    Key[s] ← 0
    Q ← V      % Q coda con priorità Key
    Buied MimHeap (Q,Key)   % costruisce uma mimHeap da Q basandosi su Key  Θ(|V|)
    while ( Q ≠ ∅ ){
        u ← ExtractMim (Q)   O(log|V|)
        for (each v ∈ Adj[u]){
                                  peso arco {u,v}
            if ( v ∈ Q && Key[v] > W({u,v})){
                Key[v] ← W({u,v})
                π[v] ← u
            % Decrease Key (Q,v)
            }
        }
}
```

E l’MST e’ memorizzato in π e il peso del MST e' $\sum_{i \varepsilon V}(key[i])$

COMPLESSITÀ: $O(|E| * log(|E|))$

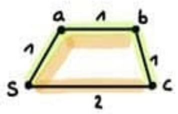
GRAFI ORIENTATI PESATI (peso positivo)

PROPRIETÀ: se prendo un cammino minimo da s a v allora tutti i suoi sotto-cammini sono cammini minimi (non vale il viceversa).

Se tutti gli archi hanno costo 1 allora si trasforma in una BFS

Differenza tra Dijkstra (**SSSP**) e Prim (**MST**):

esempio:



MST 3 → entità che collega e crea l'infrastruttura con il costo minimo → realizzo la CONNESSIONE

SSSP 4 → cammino minimo per ogni nodo → qualcuno ha già realizzato l'infrastruttura e devo prendere il percorso più breve

Con MST : cerco il minimo scheletro per collegare tutti i nodi;
Con SSSP : fissato un nodo s determino $\forall v$ nodo del grafo un cammino minimo da s a v.

ALGORITMO DI DIJKSTRA

Dijkstra(G,s) parte da un nodo s (start) e calcola le distanze minime di tutti i nodi (ovviamente solo quelli raggiungibili) da s. Simile a Prim ma tiene in memoria il peso del cammino corrente.

PROCEDIMENTO:

- Ad ogni iterazione estrae dalla coda (che viene ordinata in base al vettore delle distanze da s) un nodo ed esegue una DFS su tutti i suoi adiacenti.

- Ogni volta che trova un cammino (es. s → a → y) il cui costo (sommato all’arco preso in considerazione) e’ minore rispetto alla distanza del nodo y da s allora aggiorna il vettore dei predecessori e delle distanze.

- Una volta che ha terminato di esplorare gli adiacenti, allora estrae il nodo iniziale dalla coda e riordina in base alla distanza

COMPLESSITÀ: $O(|E| * log(|V|))$

PSEUDOCODICE:

```
CODICE: Dijkstra ( G = (V,E,W), s){
            for (each v ∈ V){        Θ(|V|)
                π[v] ← NIL
                d [v] ← ∞    % peso del cammino minimo da v a s
            }
            d[s] ← 0
            Q ← V    % Q coda con priorità Key
            BuildMinHeap(Q,Key)    % costruisce una minHeap da Q basandosi su Key  Θ(|V|)
            while ( Q ≠ ∅){
                u ← ExtractMin(Q)   O(log|V|)
                for (each v ∈ Adj[u]){
                                              peso arco {u,v} + camm. min
                    if( v ∈ Q && d[v] > d[u] + W{u,v})
                            d[v] ← W({u,v}) + d[u]
                            π[v] ← u
                        % DecreaseKey (Q,v)
                    }
                }
            }
        }
```