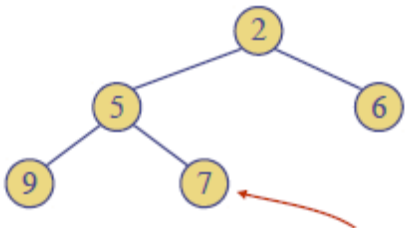
Consideriamo la seguente equazione ricorsiva di costo: C(n)<=c+C(n/2) → C(n)=O($\log_2$n). Se la associamo alla ricerca di un elemento in un array, il costo sarà logaritmico nella dimensione dell'input (considerata come il numero di elementi di un array). Se la associamo a una funzione di elevamento a potenza con n parametro di elevazione, il costo sarà lineare nella dimensione dell'input (considerata come il numero di bit necessari a salvare in memoria l'input, ovvero l'esponente).

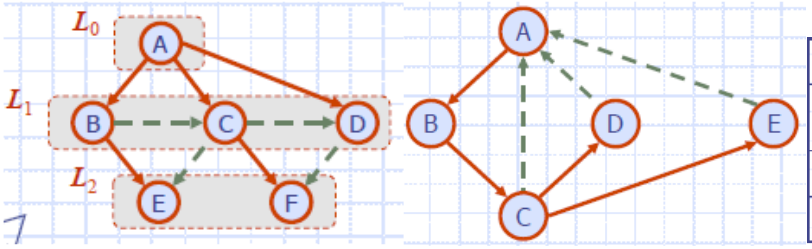


Heap: Viene utilizzato per rappresentare code di priorità. In ogni nodo la chiave associata al nodo è <= di quelle associate ai figli. L'altezza sarà uguale alla parte intera del $\log_2$n. Inserimento: si inserisce l'elemento nell'ultimo posto disponibile e si fa upheap. RemoveMin: in un heap l'elemento con chiave minima si trova nella radice. Viene rimosso, sostituito con l'ultimo elemento dell'heap e si fa downheap.

Mappe ordinate. Alberi binari di ricerca si usano per rappresentare mappe ordinate. In un albero binario di ricerca BST ho chiavi sottoalbero sinistro <= chiave radice <= chiavi sottoalbero destro.

Alberi AVL. Il fattore di bilanciamento β di un nodo è h(sx)-h(dx). Un albero bilanciato in altezza ha $|\beta|$<=1 per ogni nodo.

Grafi: Un grafo può essere connesso o non connesso, può essere diretto o non diretto. Un albero è un particolare grafo connesso, non diretto e senza cicli. Una foresta è un insieme di alberi, di conseguenza è un grafo non connesso. Dato un grafo connesso posso sempre trovare un albero ricoprente.



Applicazioni	DFS	BFS
Foresta ricoprente, cammini, cicli, componenti connesse	√	√
Cammini minimi		√
Componenti biconnesse	√	

BFS(G) for all vertice v in G: set v UNEXPLORED; (for all vertice v in G: if V is UNEXPLORED:) BFS(G,v)
BFS(G,v)
new linkedlist L; L.add(s); s.setexplored; (s.timestamp/s.count/s.print); while (!L.isEmpty)
{v=L.remove; for all vertici u adiacenti a v: {
If u is unexplored and not in L: {L.add(u); set u explored; (u.timestamp/u.count/u.print)}}}

La DFS chiama ricorsivamente se stessa all'interno della funzione su ogni nodo adiacente a quello visitato.

```
public static <V> void bfs(Graph<V> g) {
        for(Node<V> n : g.getNodes()) {
                if(n.stato == Node.Stato.UNEXPLORED)
                        bfsFromNode(g, n);}}
private static <V> void bfsFromNode(Graph<V> g, Node<V> source) {
        if(!(source.stato == Node.Stato.UNEXPLORED))
                return;
        Queue<Node<V>> queue = new ArrayDeque<Node<V>>();
        source.stato = Node.Stato.EXPLORED;
        queue.add(source);
        while(!queue.isEmpty()) {
                Node<V> u = queue.remove();
                System.out.println(u.getElement());
                for(Edge<V> e : g.getOutEdges(u)) {
                        Node<V> v = e.getTarget();
                        if(v.stato == Node.Stato.UNEXPLORED){
                                v.stato = Node.Stato.EXPLORED;
                                queue.add(v);}}}}
```

Lista di Archi	***GRAFI*** (deg grado nodo)
Iteratore su Archi nodo(v)	O(m)
sono Adiacenti(v, w)	O(m)
Inserisci Vertice	O(1)
Inserisci Arco	O(1)
Rimuovi Vertice(v)	O(m)
Rimuovi Arco	O(1)
Spazio	O(n+m)
Lista di Adiacenza	
Iteratore su Archi nodo(v)	O(n + deg(v))
sono Adiacenti(v, w)	O(min{deg(v), deg(w)}
Inserisci Vertice	O(1)
Inserisci Arco	O(1)
Rimuovi Vertice(v)	O(deg(v))
Rimuovi Arco	O(1)
Spazio	O(n+m)
Matrice di Adiacenza	
Iteratore su Archi nodo(v)	O(n)
sono Adiacenti(v, w)	O(1)
Inserisci Vertice	O(n)
Inserisci Arco	O(1)
Rimuovi Vertice(v)	$O(n^2)$
Rimuovi Arco	O(1)
Spazio	$O(n^2)$
Operazioni Varie	
DFS / DFS-Diretta	O(n+m)
BFS	O(n+m)
Kosaraju(ConnettivitàForte)	O(n+m)
Chiusura Transitiva	Lista:O(n(n+m)),Mat:$O(n^3)$

CAMMINI MINIMI	
Dijkstra (connes, non neg)	O((n+m)*logn)
Bellman-Ford(nocicli,neg)	O(n*m)
Tra tutti i nodi	O(n*(n+m)*logn)

Sequenza	***UNION-FIND***
Find	O(1)
Makecluster	O(1)
Union	O(n.elem.con id modifica)
Collegata	Ove a e b sono liste
Find	O(1)
Makecluster	O(1)
Union	O(min{a.size, b.size})
Analisi ammortizzata	O(k+nlogn)
Alberi	
Find	O(h)
Makecluster	O(1)
Union no ottimizzazioni	O(h)
Union con ottimizzazioni	O(klog*n)

AVL	
Altezza	O(logn) sempre
Rotazioni	O(1)
Find, add, remove	O(logn)

MST	
Prim-Jarnik	O((n+m)*logn
Kruskal	O((n+m)*logn

Liste NON Ordinate	***CODE***
inserimento	O(1)
remove Min	O(n)
Minimo	O(n)
Liste Ordinate	
Inserimento	O(n)
remove Min	O(1)
Minimo	O(1)
Heap	
Inserimento – Up-Heap	O(logn)
Rimozione – Down-Heap	O(logn)
Replace	O(logn)
Heapsort	O(nlogn)
Heapify	O(n)
Raddoppio dim array	O(nlogn)

Ordinamento in Place	
Insertion sort	$O(n^2)$
Array2heap	O(nlogn)
Selection sort	$O(n^2)$
Merge sort	O(nlogn)
Radix Sort	
Bucket sort	O(n+N) ove n/N fase 1/2
Radix Sort	O(n+N) ove n/N fase 1/2
Quick Sort	
Caso peggiore	$O(n^2)$
Tempo atteso	O(nlogn)

HASHING	
Calcolo polinomio	$O(n^2)$
Regola di Horn	O(n)
Costi Peggiori / Attesi	
Get	O(n) / O(1)
Put	O(n) / O(1)
remove	O(n) / O(1)

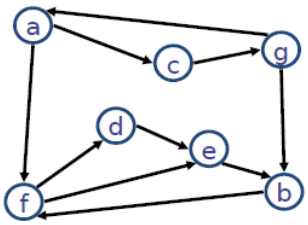
MAPPE con Liste	
Put	O(1)
Get	O(n)
Remove	O(n)

Liste INSIEMI	
Find, add, remove	O(n)
Fusione generica	$O(n_a+n_b)$ ove n_a/ n_b dim.
Spazio usato	O(n)

MAPPE ordinate	
Get	O(logn)
Put	O(n)assente / O(logn)presente
Remove	O(n)
RANGE QUERY	O(logn+s)s è costo accesso

BST	H altezza albero (fra n e logn)
Ricerca	O(h)
Inserimento	O(h)
Rimozione	O(h)
Range Query	O(h+s) s n.chiavi intervallo
Spazio Richiesto	O(n)

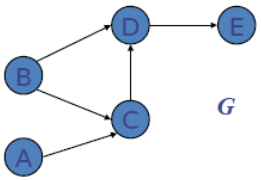
La proprietà di connettività forte si verifica quando ciascun vertice è raggiungibile da ogni altro vertice. Si può controllare se un grafo rispetta questa proprietà scegliendo un vertice e facendo una DFS, se al termine di questa esiste ancora un vertice non visitato allora la proprietà non è rispettata; successivamente si invertono tutti i vertici del grafico e si esegue nuovamente una DFS, se al termine esiste ancora un vertice non visitato allora la proprietà non è rispettata. Se entrambe le DFS vanno a buon fine allora possono restituire vero. La complessità di questo algoritmo è O(n+m).



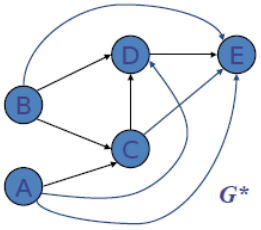
{ a , c , g }

{ f , d , e , b }

Si possono trovare anche le componenti (sottografi) fra di loro fortemente connesse in un grafo. Si utilizza l'algoritmo di Kosaraju: si esegue inizialmente una DFS del grafo inserendo i vertici visitati in una pila. Successivamente si esegue una DFS sul grafo trasposto per ogni vertice nella pila, l'insieme dei nodi trovati è una componente fortemente connessa. Complessità è O(n+m).

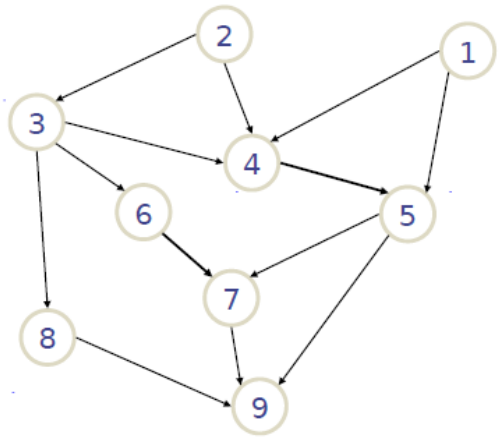


G



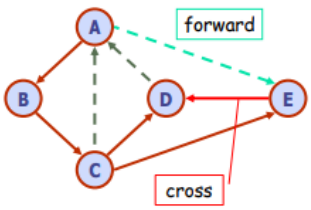
*G**

La chiusura transitiva di un digrafo G è il digrafo G* tale che abbia gli stessi vertici e se G ha un cammino diretto da u a v, allora G* ha un arco diretto da u a v. Ci permette di sapere in tempo costante se un vertice è raggiungibile da un altro vertice (utilizzando ad esempio una matrice di adiacenza). Si può fare la DFS da ognuno dei vertici ma ha costo O(n(n+m)), altrimenti si può considerare la matrice di adiacenza. Il quadrato della matrice di adiacenza ci dice quali nodi posso raggiungere in due passi, il problema principale è l'utilizzo di spazio per memorizzare le matrici, oltre al costo del calcolo dei prodotti fra matrici. Si può usare anche l'algoritmo di Floyd-Warshall per la chiusura transitiva. Si numerano i vertici, si considerano i cammini che usano 1,2,…,k come vertici intermedi. C'è un arco diretto da i a j in G_k se: c'è un arco diretto (i,j) in G_{k-1} o se ci sono archi diretti (i,k) e (k,j) in $G_{k'1}$. Il costo di questo algoritmo è vario, può essere migliore o peggiore dei metodi visti prima.

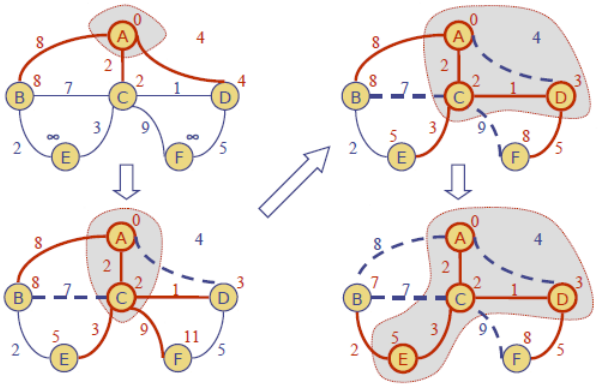


Ordinamento topologico: numera i vertici in modo tale che (u,v) in E implichi u<v. Un digrafo ammette un ordinamento topologico se e solo se è un DAG (digrafo aciclico). Se un grafo è un DAG esiste almeno un vertice che non ha archi entranti e almeno un vertice che non ha archi uscenti. Se si può definire un ordinamento topologico allora il grafo sarà privo di cicli. Si prende il vertice che non ha archi entranti e gli si assegna l'etichetta 1, poi si rimuove e rimane un altro DAG con un vertice senza nodi entranti a cui si assegna l'etichetta 2. Si prosegue così fino ad ottenere un ordinamento topologico del grafo iniziale. Si può eseguire questa procedura anche inversa, partendo dai vertici senza nodi uscenti e numerandoli a partire da n. Nella realtà non si crea veramente un secondo grafo da cui rimuovere di volta in volta i vertici ma si assegna a ciascuno di loro una variabile binaria 'rimosso'. Il costo di questo algoritmo è O(n+m). Si può implementare anche modificando una DFS.

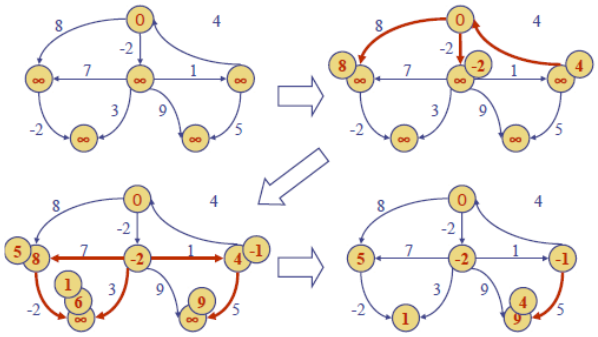
- Archi DFS tree:
  - *Tree, Discovery edges*
- Archi non-tree (in Digraph):
  - *Back edges*, verso antenato
  - *Forward edges*, verso discendente
  - *Cross edges*, verso altro vertice
- In Graph (non orientato): solo discovery e back



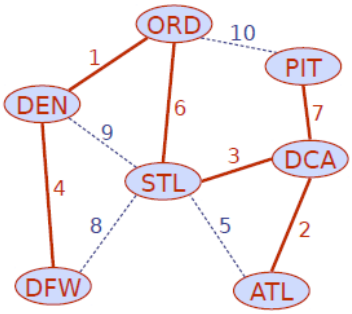
Cammini minimi in un grafo pesato. La visita in ampiezza BFS di un grafo a partire da un nodo s identifica cammini con il numero minimo di archi da s a qualsiasi altro vertice raggiungibile. Ci interessa però il caso in cui ogni arco abbia un peso diverso. Definiamo w(u,v) il peso dell'arco (u,v). Nel caso dei grafi diretti può non esistere un cammino fra due vertici ma possono anche esistere più cammini minimi di peso uguale. Ogni sottocammino di un cammino minimo è a sua volta un cammino minimo. Dato un grafo ed un nodo sorgente generalmente viene richiesto di calcolare un albero di cammini minimi dalla sorgente a tutti i vertici raggiungibili. Il risultato di quest'algoritmo è l'etichettatura di ogni nodo con la lunghezza del cammino dalla sorgente. d[u] è la distanza minima dalla sorgente s al nodo u se e solo se per ogni (v,u) d[u]<=d[v]+w(v,u). Algoritmo di Dijkstra: dato un vertice calcolare l'albero dei cammini minimi da s ad ogni altro vertice. Assumiamo il grafo connesso, i pesi non negativi e il grafo non diretto. Si imposta tutte le d[u] ad un valore massimo. Si fa crescere un sottoinsieme di vertici S che all'inizio include solo s e alla fine include tutti i vertici. Ad ogni passo si aggiunge a S il vertice u che non gli appartiene e che ha d[u] minima. Si aggiornano anche le etichette dei vertici adiacenti a u. La d di un vertice va diminuendo finchè alla fine è quella del cammino minimo. Una volta messo un nodo in S non modifico più la sua d[u]. Il passo chiave di questo algoritmo è il rilassamento, ovvero quando aggiorno



d[z]=min(d[z], d[u]+w[u,z]). Si implementa con una coda di priorità che rende meno costosa la scelta del nodo con d[u] minore, è da considerare che avremo bisogno di modificare le chiavi per cui serve un heap flessibile. La complessità dell'algoritmo di Dijkstra è O((m+n)logn) sfruttando una coda di priorità. Se voglio calcolare le distanze minime fra ogni coppia di vertici posso eseguire l'algoritmo di Dijkstra n volte (una per ogni possibile vertice sorgente) con costo O(n(n+m)logn).

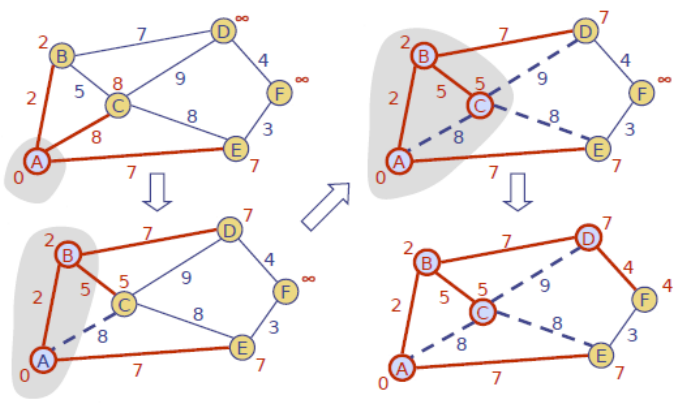


L'algoritmo di Bellman-Ford funziona nel caso di grafi diretti privi di cicli con pesi negativi e per grafi non diretti con pesi positivi. Ha complessità O(m*n). Per mantenere l'albero dei cammini minimi devo salvare in v anche il nodo che mi ci ha fatto arrivare con percorso minimo (lo faccio durante il rilassamento se aggiorno d[v]). Altrimenti possiamo salvare il predecessore di ogni vertice nell'albero dei cammini minimi in un array esterno.

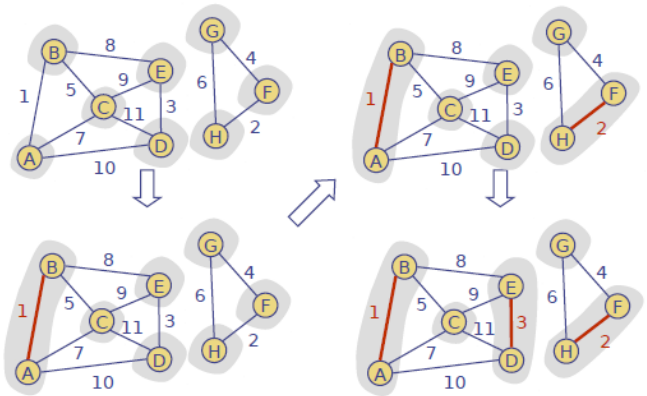


Alberi minimi ricoprenti (Minimum Spanning Tree). Consideriamo grafi non diretti con pesi positivi. Vogliamo un albero che tocchi tutti i nodi tale che la somma dei pesi degli archi scelti sia minima (peso complessivo). Proprietà:

- Di ciclo: per ogni arco f di C w(f)<=w(e) (con un MST T, un arco e non in T, un ciclo T unito ad e C), ovvero il peso di e deve essere maggiore degli altri, altrimenti potremmo ottenere un albero ricoprente di peso minore.
- Di taglio: un taglio è una partizione di V in due sottoinsiemi V1 e V2, l'arco del taglio è qualsiasi arco con un estremo in V1 e l'altro in V2. Sia e l'arco di peso minimo del taglio (V1, V2), allora esiste un MST che contiene e.



Algoritmo di Prim-Jarnik: prendiamo un qualsiasi nodo di partenza e tutti i suoi archi. Assegniamo a lui valore zero e agli altri il peso dell'arco. Poi si sceglie il nodo collegato dall'arco di peso minimo e si mette come visitato, si aggiunge l'arco al MST. Si aggiornano le distanze dei nodi non visitati a lui adiacenti se sono minori di quelle dal primo nodo. Poi scelgo l'arco con peso minore che collega un nodo visitato e uno non visitato e lo aggiungo al MST, segno il nodo come visitato. Proseguo fino ad aver visitato tutti i nodi e ottengo un MST. Il costo di quest'algoritmo è O((n+m)logn).



Algoritmo di Kruskal: ogni nodo parte da solo, poi si sceglie l'arco di peso minimo e si uniscono due nodi con quello. Poi si controlla il successivo arco con peso minimo e, se non unisce due nodi già legati, si usa per unire due nodi o gruppi di nodi. Si prosegue fino ad ottenere un MST. Ha complessità O((m+n)logn).