

Sorting Strutture dati

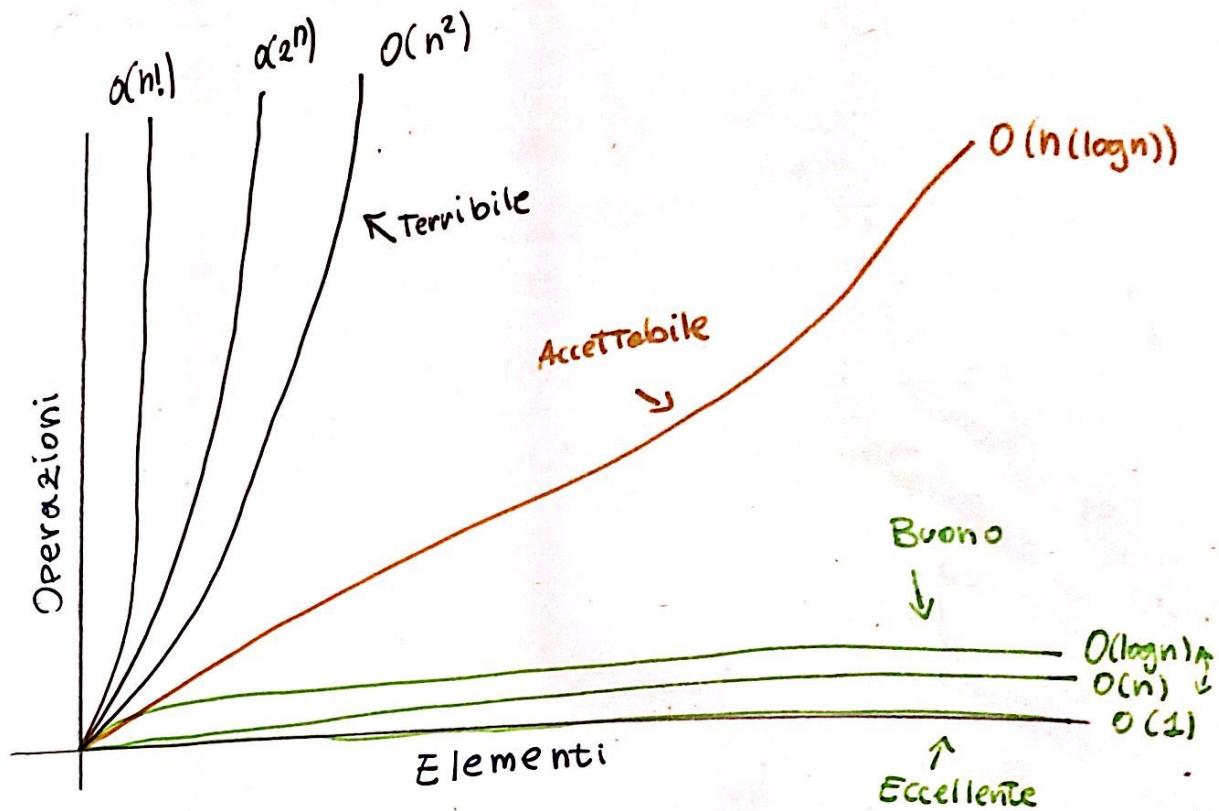
Struttura	Time Complexity				Space Complexity			
	Average				Worst			
	Indexing	Search	Insert	Delete	Indexing	Search	Insert	Delete
Basic Array	$O(1)$	$O(n)$	—	—	$O(1)$	$O(n)$	—	—
Dynamic Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	—	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
HashTable	—	$O(1)$	$O(1)$	$O(1)$	—	$O(n)$	$O(n)$	$O(n)$
BST	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
RB Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	—	$O(\log n)$	$O(\log n)$	$O(n)$

Ricerca

Algoritmo	Struttura	Time complexity		Space
		Average	Worst	
DFS	Grafo - Albero	—	$O(E + V)$	$O(V)$
BFS	Grafo - Albero	—	$O(E + V)$	$O(V)$
Binary Search	Array ordinato	$O(\log n)$	$O(\log n)$	$O(1)$
Linear (BF)	Array	$O(n)$	$O(n)$	$O(1)$
Dijkstror (min-heap)	Grafo	$O((V + E)\log V)$	$O((V + E)\log V)$	$O(V)$
Dijkstra (priority q.)	Grafo	$O(V ^2)$	$O(V ^2)$	$O(V)$
Bellman Ford	Grafo	$O(V \cdot E)$	$O(V \cdot E)$	$O(V)$

Sorting

Algoritmo	Struttura	Time Complexity			Space Complexity
		Best	Average	Worst	
QuickSort	Array	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
MergeSort	Array	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
HeapSort	Array	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
BubbleSort	Array	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	Array	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \cdot k)$



Ordine di crescita	Nome	Esempio
1	Costante	Somme di due num.
$\log n$	logaritmico	Ricerca binaria
n	lineare	Ricerca minimo
$n \log n$	linearitmico	MergeSort
n^2	Quadratico	controllo coppie
n^3	Cubico	Controllo triplette
2^n	Esponenziale	Controllo sottoinsiemi

Hash Map
Salva i dati con copie chiave - valore. Le funzioni di hash accettano una chiave e ritornano un output unico che è associato solo a quella chiave.

Collisioni di hash

Le collisioni si verificano quando una funzione di hash ritorna lo stesso output per due input diversi. Le collisioni si risolvono con il **Separate Chaining**, dove gli elementi vengono inseriti in una lista (quando si verifica una collisione) oppure con il **Linear Probing**; in questo caso inseriamo la chiave, se siamo in presenza di una collisione, nella posizione utile successiva all'index di hash.

Clustering

In cluster c'è un blocco di elementi, aggiungendo delle chiavi potremmo andare ad incrementare la grandezza di un cluster.

Quando elimino nel **Linear Probing** DEVO effettuare il re-hashing di tutte le chiavi rimanenti.

Resize - S.C.

L'obiettivo è mantenere $N/M = \text{costante}$

- Raddoppio la grandezza M quando $N/M >= 8$
- Dimezzo M quando $N/M <= 2$
- Rieffettuo d'hashing

Resize - L.P.

L'obiettivo è mantenere $N/M <= \frac{1}{2}$

- Raddoppio M quando $N/M >= \frac{1}{2}$
- Dimezzo M quando $N/M <= \frac{1}{8}$
- Rieffettuo d'hashing

Time Complexity

- indexing : $O(1)$
 - Ricerca : $O(1)$
 - inserimento : $O(1)$
- } de Hash Map sono estremamente efficienti!

Informazioni importanti Algoritimi e Strutture

Arrays

Gli array salvano i dati degli elementi basandosi su un indice, solitamente che inizia da 0. Questo tipo di struttura è una delle più vecchie e anche più usata.

Time Complexity

- indexing $O(1)$
- ricerca $O(n)$
- inserimento $O(n)$

Resize Arrays

Quando effettuiamo il resize, non aumentiamo le dimensioni di 1, ma **Raddoppiamo** l'array iniziale. Lo stesso si applica quando dobiamo rridurlo:

- Push(): raddoppiamo quando è pieno
- Pop(): dimezziamo quando è pieno per $\frac{1}{4}$.

→ lo stack è sempre pieno tra il 25% e 100%.

Linked List

Salva i dati attraverso dei Nodi che puntano ad altri Nodi.

root \rightarrow [1] \rightarrow [5] \rightarrow [4] \rightarrow [3] \rightarrow NULL

Time Complexity

- indexing: $O(n)$
- ricerca: $O(n)$
- inserimento: $O(n)$

} Nota come una struttura lenta

Albero Binario

E' un tipo di struttura dati ad ALBERO, dove ogni nodo ha al massimo DUE Figli. Questi due figli sono posti a sinistra e a destra del nodo padre.

Gli alberi sono progettati per la ricerca ed ordinamento. Gli alberi sono inoltre usati per creare i BST, ovvero un albero che effettua dei confronti, nell'inserimento, per decidere se aggiungere l'elemento a sx o Dx di ogni nodo.

Time Complexity

- inoltre: $O(\log n)$
 - ricerca: $O(\log n)$
 - inserimento: $O(\log n)$
- } BST, le operazioni sono basate sulla search(), che è $O(\log n)$

Ricerca - Breadth First Search BFS

E' un algoritmo che ricerca all'interno di un albero (o grafo) ricorrendo prima in ogni Livello, partendo dalla root.

Questo alg. Trova ogni nodo sullo stesso livello; facendo ciò traccia i nodi figli dei nodi del livello corrente.

Quando ha finito di esaminare un livello si sposta sul nodo più a sinistra del prossimo livello.

da BFS usa una Queue per salvare le informazioni sull'albero; per questo motivo usa più memoria delle DFS.

Ricerca - Depth First Search DFS

da DFS, al contrario della BFS, ricerca andando in profondità: arriva prima al nodo più a sinistra, quando raggiunge la fine del ramo, torna indietro ricorsivamente provando gli eventuali nodi destro, per prima, e poi sinistro. Il nodo più a destra è esaminato per ultimo.

BFS - time Complexity

- Ricerca BFS: $O(|E| + |V|)$

DFS - time Complexity

- Ricerca DFS: $O(|E| + |V|)$

Pro e Cons

da BFS viene usata per alberi che sono più 'larghi' che profondi, questo perché controlla livello per livello.

da DFS, invece, in caso di alberi molto profondi, potrebbe andare in profondità inutilmente.

DFS & BFS

	Tempo	Spazio
DFS	$O(E+V)$	$O(\text{Altezza})$
BFS	$O(E+V)$	$O(\text{Lunghezza})$

Altezza: Altezza max dell'albero

Lunghezza: Massimo numero di nodi in un singolo livello.

DFS

- Migliore quando l'obiettivo è più vicino alle root.
- Stack \rightarrow LIFO
- Ricerca Preorder, inorder, Postorder
- Va in profondità
- Ricorsiva
- Veloce

BFS

- Migliore quando l'obiettivo è lontano
- Queue \rightarrow FIFO
- Ricerca Level order
- Va in lunghezza
- Iterativa
- Lenta

Binary Search

	Tempo	Big O
BS	$O(\log n)$	

Spazio

$O(1)$

Heaps

Gli heaps possono essere di due tipi: **MinHeap** e **MaxHeap**, la differenza tra i due è che il primo elemento può essere il più piccolo o più grande della collezione.

In un **MinHeap** tutti gli elementi sono più piccoli dei loro figli. Quindi, scendendo lungo l'albero, gli elementi diventano sempre più grandi.

Questo però non ci dice che la struttura sia ordinata, sappiamo infatti che la **root** è l'elemento più piccolo, ma non se gli altri elementi sono ordinati.

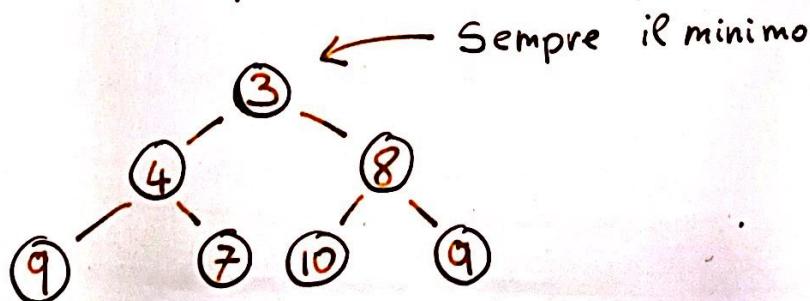
Abbiamo un ordinamento completo solo quando effettuiamo il **pop()** dell'elemento minimo, questo perché dopo ogni **pop()** la struttura deve essere 'riarrangiata', ponendo il min in prima pos.

Inserimento

Quando inseriamo un elemento, esso viene sempre posto nel primo posto libero allo fine della struttura. Se l'elemento non rispetta l'ordinamento (dove essere più grande del suo genitore) allora si effettua un'operazione chiamata **Bubble-up**, che fa 'salire' l'elemento fino allo posto che rispetta l'ordinamento.

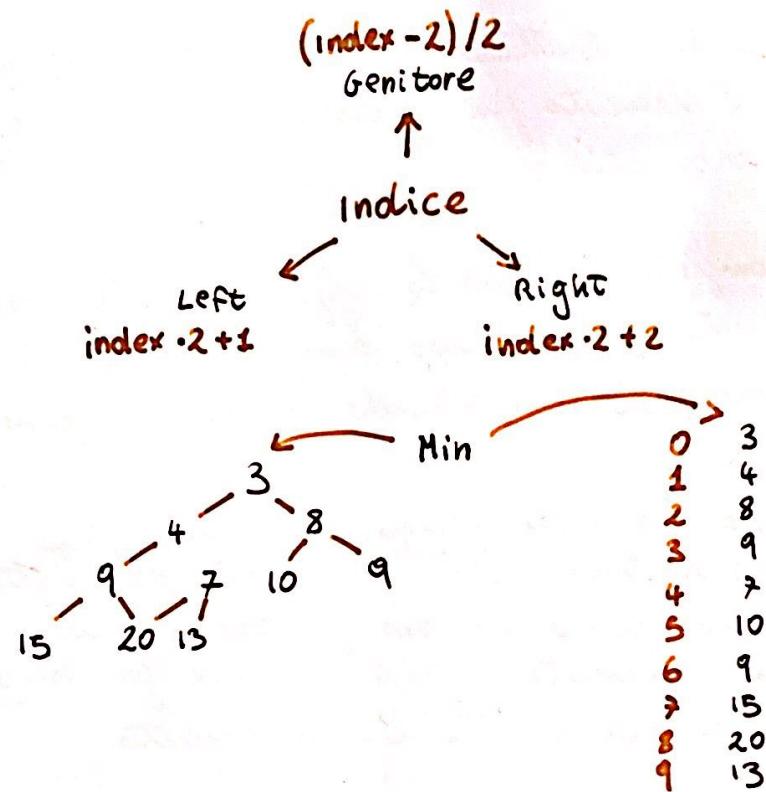
Rimozione

Per rimuovere un elemento scambiamo la root con l'ultimo nodo, rimoviamo l'ultimo nodo (la vecchia root) e successivamente effettuiamo il **Bubble-down** della root. L'operazione fa 'scendere' l'elemento finché non rispetta le condizioni.



Implementazione

Potremmo pensare di implementare la struttura usando gli oggetti, e creando una classe `Node`, ma esiste un modo migliore per farlo: guardando la struttura ci risulta semplice capire che gli elementi (padre e figli) si troveranno sempre in una posizione precisa, e facciamo quindi determinare la posizione di ogni nodo con una semplice equazione:



Binary Search
da BS usa la struttura dati array che è stata precedentemente ordinata. L'idea generale è quella di suddividere l'array ed ogni passaggio a metà, usare solo la metà dove potrebbe essere l'elemento che cerchiamo.

Algoritmo

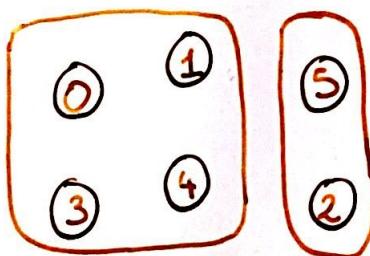
- Confronta la Key con l'el mid dell'array
- se la Key è maggiore cerca a destra
- Se la Key è minore cerca a sinistra
- Se la Key è uguale l'ho trovato.

Time complexity

Siccome dividiamo l'array ad ogni passaggio, l'algoritmo è molto efficiente con un tempo di esecuzione pari a $O(\log n)$.

Union Find

Immaginiamo di avere dei gruppi di oggetti, un esempio potrebbe essere il seg:



Una volta che abbiamo raggruppato gli oggetti, potremmo voler sapere alcune proprietà degli oggetti, come ad esempio

- Quanti gruppi ci sono?
- Due oggetti appartengono allo stesso gruppo?

La struttura supporta due operazioni:

1. Union(x, y) → unisce i gruppi contenenti x e y.



2. Find(x) → trova il gruppo a cui x appartiene.

Utilizzi

L'Union-Find può essere usato in molte applicazioni, tra cui:

- Trovare le componenti connesse in un grafo.
- Reti di amici in un Social Network.
- Diversi pixel dello stesso colore in un'immagine.

Algoritmi di Ordinamento

	Migliore	Media	Spazio
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n)$
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(1)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(\log n)$
Insertion	$O(n)$	$O(n^2)$	$O(1)$
Selection	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble	$O(n)$	$O(n^2)$	$O(1)$

Merge Sort

L'algoritmo è basato sui confronti: divide l'intero insieme di dati in gruppi di massimo due elementi. Confronta ogni numero uno alla volta, ponendo il più piccolo alla sinistra della coppia.

Una volta che tutte le coppie sono ordinate confronta gli elementi più a sinistra con le altre coppie a sinistra, creando un gruppo ordinato di quattro con gli elementi più piccoli a sx e i più grandi a dx.

Questo procedimento viene ripetuto finché non rimane un solo insieme o gruppo.

Analisi

Siccome l'algoritmo divide i dati da confrontare ad ogni iterazione, ha un tempo di esecuzione linearitico. Anche nel caso peggiore garantisce lo stesso tempo: $O(n \log n)$

Quick Sort

Anche il qs è un algoritmo basato sui confronti. Divide l'intero insieme di dati in due selezionando l'elemento Medio, e posizionando tutti gli elementi minori allo sx, e tutti i maggiori a dx.

Continua con lo stesso processo sul lato Sinistro finché si ritrova a confrontare solo due elementi, a quel punto il lato sx è ordinato. Viene ripetuta la stessa operazione sul lato dx.

Extra

Anche se il qs ha lo stesso notazione Big $O(n \log n)$ di molti altri algoritmi di ordinamento, è spesso più veloce nelle pratiche di molti altri, come il Merge Sort.

- Caso peggiore: $O(n^2)$

Grafi non direzionali

Un grafo non direzionale è composto da un insieme di vertici ed un insieme di archi.

I grafi sono stati usati in centinaia di applicazioni, come nella comunicazione nei circuiti e trasporti.

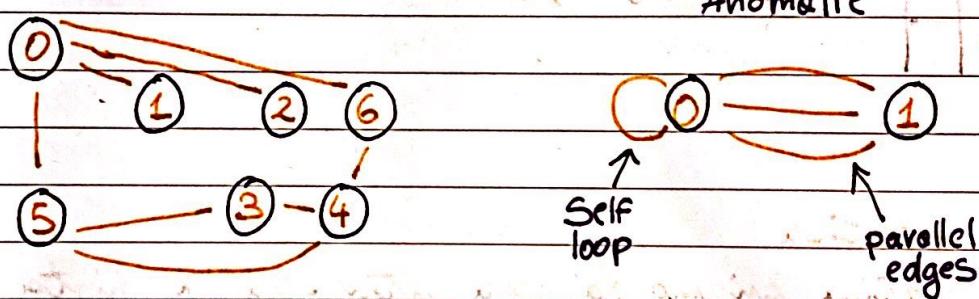
Terminologia dei grafi

Path: Sequenza di vertici connessi da archi.

Ciclo: Un path il quale primo ed ultimo node coincidono.

Rappresentazione

Per rappresentare i grafi in modo semplice gioiamo usare degli interi compresi tra 0 e $V-1$, e collegarli tramite delle linee:



Matrice di adiacenza

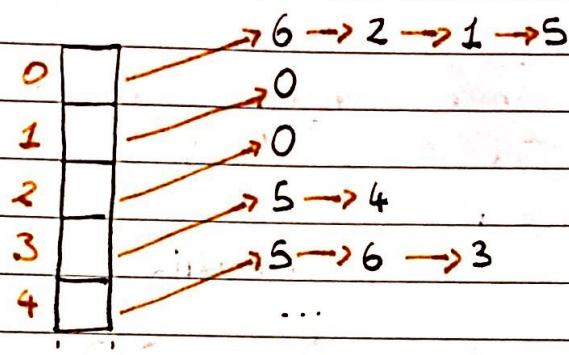
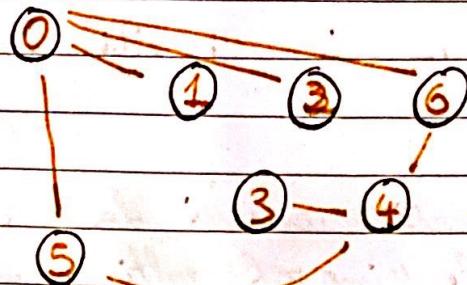
Usiamo un array di booleani bidimensionale V per V e per ogni arco $v-w$ nel grafo: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$.

Stiamo dicendo che due vertici sono adiacenti se, scambiamoli gli indici, il valore nella matrice è 1 in entrambi.

True

Lista di adiacenza

Per una rappresentazione con matrice di adiacenza, manteniamo un array di liste vertice - indice



Nella Pratica

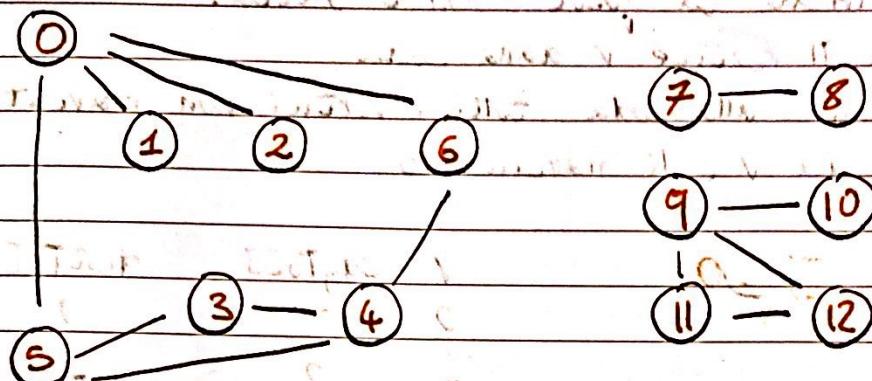
Nella pratica usiamo la rappresentazione a lista di adiacenza, questo perché la rappresentazione è semplice da implementare e funziona bene.

Questo tipo di implementazione si basa sull'iterare sui vertici adiacenti a v .

DFS nei grafi

Per visitare un vertice v :

- Marchiamo v come visitato
- Visitiamo ricorsivamente tutti i vertici adiacenti a v .



V $\text{marked}[]$ $\text{edgeto}[]$

0 T 3 -

1 T 2 0

2 T 1 0

3 T 5 6

4 T 6 -

5 T 4 -

6 T 0 -

7 F - -

8 F - -

9 F - -

10 F - -

11 F - -

12 F - -

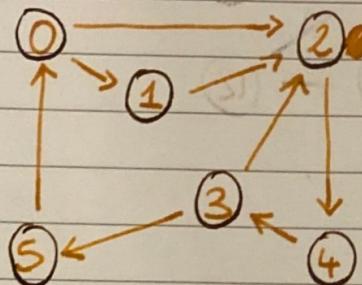
Vertici raggiungibili
da zero.

BFS nei grafi

Nel caso delle ricerche in ampiezza non usiamo uno stack, che nelle DFS usavamo attraverso la ricorsione, ma usiamo una coda esplicita.

Ripetiamo finché la coda non è vuota:

- Rimuoviamo il vertice v dalla coda
- Aggiungiamo alla coda tutti i vertici non marcati che puntano da v e li marchiamo



v	edgesTo[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

Bisogna notare che la BFS NON è ricorsiva, a differenza della DFS.

Grafi direzionali

Ci sono diverse applicazioni in cui sono utili archi orientati invece di archi NON orientati. Ci sono anche differenze tra i due tipi di grafi, e la maggior parte delle implementazioni dei grafi non direzionati possono essere usate per i grafi direzionati.

Un **Digraph** è formato da un insieme di vertici connessi a coppie da archi **direzionati**, ovvero un arco va da un nodo sorgente ad uno destinazione.

Per questo motivo, poniamo, per ogni nodo, definire due tipi di grado:

- **inDegree**: Il numero di archi entranti.
- **outDegree**: Il numero di archi uscenti.

Un percorso direzionato è un percorso che va da un nodo sorgente ad un nodo destinazione. In questo caso poniamo non c'è viaggio se lo percorriamo al contrario, come succede nei grafi non direzionati.

Applicazioni

- Reti stradali: i nodi o vertici sono gli **incroci** e gli archi le strade. In questo caso poniamo rappresentare le strade one way.

Problemi generali

- **S → t** esiste un percorso da **s** a **t**?
- **S → t più corto** ~~c'è~~ percorso più corto da **s** a **t**?
- **ciclo direzionale** c'è un ciclo direzionale nel grafo?
- **ordinamento topologico** puo' il grafo essere disegnato in modo che tutti gli archi puntino in un unico verso?

Rappresentazione: Matrice di adiacenza

Anche in questo caso poniamo mantenere un array di liste vertice-indice d'unica differenza: c'è che se abbiamo un arco del tipo:

 Ci troveremo l'elemento **6** nella lista di adiacenza di **9**, ma non viceversa.

Time Complexity

	Spazio	insert	Arco da v a w ?	Iterare sui vertici uscenti da v
lista di archi	E	1	·E	E
Matrice di adj	V ²	1	1	V
Lista di adj	E+V	1	outdegree(v)	outdegree(v)

↳ Soluzione ottimale

Depth FIRST Search

Il funzionamento è identico alla DFS dei grafici NON direzionali.
Inoltre, la DFS è un algoritmo pensato proprio per i grafici reg.

Applicazioni della DFS nei Dgraphs

Ogni programma è un Dgraph.

- **Vertice** : blocco basilese di istruzioni
- **Arco** : Salto (jump)

- Eliminazione di codice 'morto'

Trovare ed eliminare una porzione di codice non raggiungibile.

- Identificazione di loop infinito

Determinare se un'uscita è irraggiungibile.

Breadth first search

Anche in questo caso il funzionamento è identico alle prec. implementazioni.

Applicazioni della BFS

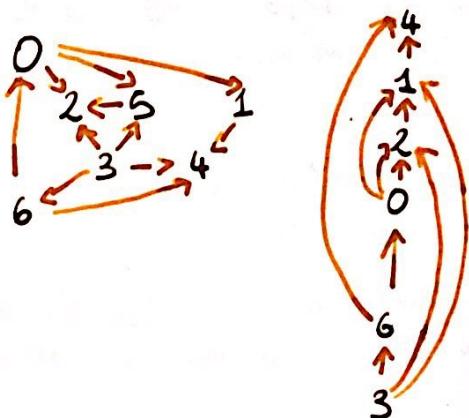
→ Shortest paths da Sorgenti multiple

dato un Digraph ed un insieme di sorgenti, trovare il percorso minimo da ogni vertice nell'insieme, ad ogni altro vertice.

Ordinamento topologico

Viene effettuato sui grafici direzionali aciclici.

d'obiettivo c'è quello di ridisegnare tutti i vertici ed archi del grafo in modo che gli archi puntino in un'unica direzione.



- Per determinare i cicli possiamo usare la DFS ed una struttura dati in cui vedo se segnare tutti i nodi in cui sono arrivato; se nella struttura compare due volte lo stesso nodo, allora esiste un ciclo.
- d'ordinamento topologico è UNICO

topological Sort

- Eseguiamo la DFS
- Ritomiamo i vertici nella struttura postorder, in reverse.

Per eseguire l'algoritmo, quindi, usiamo un ^{array} per tenere tracce degli nodi visitati, ed ogni volta che arriviamo ad un nodo da cui non gioiamo visitare altri nodi (non già visitati) aggiungiamo il nodo allo stack e torniamo indietro.

Rilevare cicli in un Digraph

In Digraph ha un ordinamento topologico se non c'è presente un ciclo.

Dimostrazione

- Se c'è presente un ciclo, l'ordinamento topologico è impossibile.
- Se Non c'è presente un ciclo, un algoritmo basato sullo DFS troverà l'ordinamento topologico.

Ordinamenti della DFS

da DFS visita ogni vertice esattamente una volta. L'ordine in cui lo fa, quindi, può essere importante.

Ordini

- Preorder : l'ordine in cui la dfs() è chiamata.
- Postorder: l'ordine in cui la dfs() ritorna.
- Reverse postorder: l'inverso in cui la dfs() ritorna.

```
private void dfs(...) {  
    marked[v] = true;  
    Preorder.enqueue(v);  
    for(int w : g.adj(v))  
        if(!marked[w])  
            dfs(g, w);  
    Postorder.enqueue(v);  
    reversePostorder.push(v);  
}
```

Componenti fortemente connesse

In un Digraph, un vertice v e w sono fortemente connessi se c'è un cammino direzionato sia da v a w che da w a v .

Proprietà chiave

Copiamo quindi che in questo caso i cicli sono fondamentali, dato che due vertici sono F.C. se e solo se c'è presente un ciclo direzionato che li contiene entrambi.

Inoltre la **Strong connectivity** è una relazione di **equivalenza**:

- v è F.C. a v
- Se v è F.C. a w , allora w è F.C. a v .
- Se v è F.C. a w , e w è F.C. ad x , allora v è F.C. ad x .

Algoritmo di Kosaraju-Sharir

Base dei componenti fortemente connesse in G sono le stene di G^R .

In un grafo inverso gli archi sono orientati al contrario. Grafo inverso.

L'idea consiste nel calcolare l'ordine topologico (**reverse postorder**) nel Kernel DAG.

Dopo che eseguiamo la DFS, considerando i vertici nell'ordine topologico.
↑
nel grafo iniziale.

Esecuzione

1. Eseguiamo l'ordinamento topologico sul grafo inverso.

Averemo uno stack del tipo:

① 2 4 5 3 11 9 12 ...

2. Eseguiamo la DFS in G , visitando i vertici non marcati in **Reverse postorder** di G^R . Ripetiamo quindi l'ordine trovato nel passo precedente.

- Parto quindi dal primo nodo (0) ed eseguo la DFS. Ogni nodo raggiunto dalla DFS avrà lo stesso ID.
- Ad ogni raggiungimento ci fermiamo quando torniamo all'elemento iniziale, oppure quando non siamo visitate nodi non ancora visitati.
- Ignoro i nodi a cui c'è già assegnato un ID.

Minimum Spanning tree

Un albero di copertura è un sottografo che è:

- Connesso
- Aciclico
- Include tutti i vertici

Dato un grafo non orientato G con dei pesi degli archi positivi, il nostro goal è trovare un albero di copertura di peso minimo. Il costo complessivo, dato dalla somma di tutti gli archi compresi nel sottoalbero, deve essere minimo.

Algoritmo di Greedy

Assunzioni di semplificazione

- Il grafo è connesso
- I pesi degli archi sono diversi

Proprietà di 'TAGLIO'

- Un taglio in uno grafo è una partizione dei suoi vertici in due insiemi non vuoti
- Un 'crossing edge' connette un vertice di un insieme con un vertice in un altro.

Proprietà: Dato un qualsiasi taglio, il crossing edge di peso minimo è nel MST.

Algoritmo

- Iniziamo con tutti i vertici colorati di grigio.
- Troviamo un taglio con nessun crossing edge nero. Coloriamo il suo arco di peso minore di nero.
- Ripetiamo finché $V-1$ vertici sono colorati di nero.

In pratica ...

d'olgoritmo in pratica effettua dei tagli che attraversano gli archi del grafo e lo dividono in due sottografi.

Succivamente si trova l'arco di peso minimo (tra quelli colpiti dal taglio) e lo si aggiunge al MST.

La condizione di terminazione è che dobbiamo avere $V-1$ archi. Ad esempio, con un grafo di 7 nodi dobbiamo avere 6 archi.

Algoritmo di Kruskal

Con l'algoritmo di Kruskal consideriamo gli archi in ordine Ascendente di peso.

- Aggiungiamo il primo all'albero T a meno che non crei un ciclo.

Per prima cosa bisogna effettuare un ordinamento per peso degli archi. Successivamente l'esecuzione dell'algoritmo è semplice: continuo ad aggiungere gli archi all'MST finché non non crea un ciclo.

Challenge di implementazione

la sfida da si affronta con questo algoritmo è quella di determinare se l'arco da aggiungere crea un ciclo oppure no.

- V : dobbiamo la DFS da v per controllare se w è raggiungibile
- $\log^* V$: Usare la struttura dati Union-Find

Soluzione efficiente

Usando la struttura Union-Find otteniamo la soluzione più efficiente.

- Manteniamo un insieme per ogni componente连通 in T
- Se v e w sono nello stesso insieme, allora aggiungere $v-w$ creerebbe un ciclo.
- Per aggiungere $v-w$ a T , effettuiamo il merge degli insiemi contenuti v e w

Implementazione

• Usiamo una Priority Queue per ordinare gli archi.

- Usiamo la struttura UnionFind, inizialmente aggiungendo tutti gli archi del grafo.
- Verifichiamo attraverso la UF se i due archi sono connetti.
- Se non sono connetti, effettuiamo il merge dei due insiemi v e w ed aggiungiamo all'MST l'arco.

Tempo di esecuzione

L'algoritmo di Kruskal calcola l'MST in un tempo proporzionale a $E \log E$, nel caso peggiore.

Perché?

operazione	Frequenza	Tempo per op	
Build pq	1	E	→ Copio tutti gli Edges nella PQ
delete min	E	$\log E$	→ Potrei dover far scendere il primo el per tutta l'altezza dell'Heap
Union	V	$\log^* V$	→ Altezza dell'albero
connected	E	$\log^* V$	→ Altezza dell'albero

Algoritmo di Prim

- Iniziamo con vertice \emptyset e costruiamo l'albero T in maniera greedy.
- Aggiungiamo a T l'arco di peso minimo con estremante un endpoint int.
- Ripetiamo finché non abbiamo $V-1$ archi.

A differenza dell'algoritmo di Kruskal che fa crescere l'albero a macchia di leopardo, l'algoritmo di Prim fa crescere l'albero a macchia d'olio.

Esecuzione

Anche in questo caso ordiniamo gli archi a seconda del loro peso. Partiamo quindi da quelli di peso minore, partendo da zero (in questo caso). Aggiungo quindi l'arco all'MST.

A questo punto sono in un nuovo nodo; devo considerare tutti gli archi uscenti dal nodo, e selezionare quello di peso minore, aggiungerlo all'MST.

Continuo in questo modo sempre scegliendo l'arco di peso minore, da prendere dalla coda a priorità.

00:33 L 23

Shortest Paths

Se nelle lezioni precedenti abbiamo visto i percorsi su grafi NON orientati, in questo caso analizziamo i grafi orientati.

Ci sono diverse varianti di questo problema:

- Sorgente Singola
- collegamento Singolo
- Sorgente - collegamento
- tutte le coppie

Da un vertice s a tutti gli altri.

Da tutti i vertici ad un singolo vertice s .

Da un vertice s ad un vertice t .

Tra tutte le coppie di vertici.

Restrizioni

- Pesi NON negativi.
- Pesi Euclidiani.
- Pesi Arbitrari.

cicli

- Nessun ciclo orientato.
- Nessun ciclo negativo.

API

	DirectEdge (int v, int w, double weight)	// v->w
int	From()	// Vertice v
int	To()	// Vertice w
double	weight()	// peso dell'arco

Rappresentazione

Anche in questo caso usiamo la lista di adiacenza. La lista funziona come nel caso dei grafi orientati semplici, aggiungendo l'arco solo al nodo da cui esso è uscente.

L'unica differenza è l'oggetto **Directed Edge**, ovvero l'oggetto che contiene le informazioni del nodo di partenza, di arrivo ed il peso dell'arco stesso.

Single source S.P.

Per trovare lo S.P. da un singolo vertice a tutti gli altri abbiamo bisogno di due strutture dati:

double distTo(int v) // distanza da s-v

Iterable <Directed Edge> pathTo(int v) // path da s-v

Ovvio che queste due strutture sono dei semplici arrays:

- **distTo[v]** lunghezza dello SP
- **edgeTo[v]** d'ultimo arco sullo SP da s-v

edgeTo[] distTo[]

0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
:	:	:

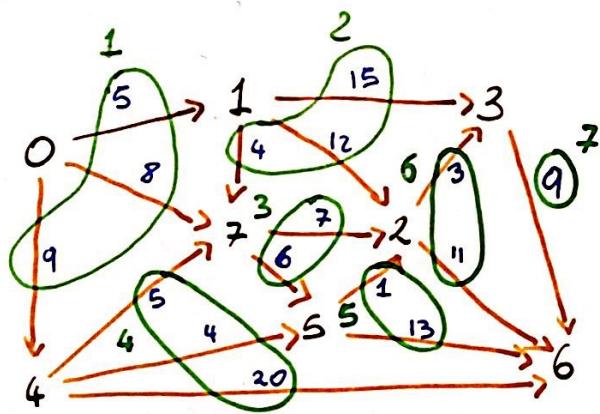
Algoritmo di Dijkstra

- Consideriamo i vertici in ordine crescente di distanza da s
- Aggiungiamo i vertici all'albero e rilessiamo tutti gli archi che partono da quel vertice.

Rilessare un arco $e = v \rightarrow w$

- $\text{disto}[v]$ è la lunghezza del path più corto conosciuto da s a v
- $\text{disto}[w]$
- $\text{edgeTo}[w]$ è l'ultimo arco sul path più corto conosciuto da s a w
- Se $e = v \rightarrow w$ ci restituisce un path più corto a w attraverso v , allora aggiorniamo sia $\text{disto}[w]$ $\text{edgeTo}[w]$.

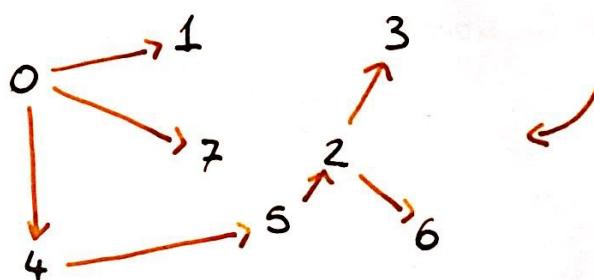
In poche parole... da procedura di rileggimento ci serve solo aggiornare l'arco che collega v a w SE ne troviamo uno di zero minore.



V	distTo[]	edgeTo
0	0.0	-
1	5.0	0->1
2	17.0 / 15.0 / 14	1->2 / 3->2 / 5->2
3	30.0 / 17	1->3 / 2->3
4	9.0	0->4
5	14.0 / 13.0	14->5 / 0->5 / 4->5
6	29.0 / 26.0 / 25.0	4->6 / 5->6 / 2->6
7	8.0	0->7

* Il primo passaggio ci dice di settare
Il primo modo a 0 $\Rightarrow \text{disto}[0] = 0$
e tutte le altre distanze a $+\infty$.

- Aggiungo 0 al Tree e considero tutti gli archi uscenti da zero.
- Considero l'arco che ha la distanza minore dal costituendo tree.
- Considero quindi 1, lo aggiungo al Tree e considero i suoi archi uscenti.
- Vado avanti e considero 7 ed i suoi archi uscenti. Troviamo un percorso per arrivare a 2 minore di quello già esistente, quindi aggiorniamo.
- Continuo in questo modo



Percorso minimo da un nodo a tutti gli altri

Calcolare un Minimum Spanning Tree (MST)

Prim vs Dijkstra

Entrambi gli algoritmi sono nella stessa famiglia di algoritmi, ed entrambi calcolano un **Albero di copertura**, quello che li distingue è la **regola** che sceglie il prossimo vertice nell'albero:

- **Prim:** Scegliamo il vertice più vicino all'albero
→ Archi **NON orientati**
- **Dijkstra:** Scegliamo il vertice più vicino alla sorgente
→ Archi **orientati**

01:10 L 24