

Grafi direzionati

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 percorso non c'è vedi se ne 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 oneway.

Problemi generali

- **S-T** esiste un percorso da S a T?
- **S-T più corto** esiste percorso più corto da S a T?
- **ciclo direzionato** c'è un ciclo direzionato 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 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 direz.

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 inraggiungibile.

Breadth first search

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

Applicazioni della BFS

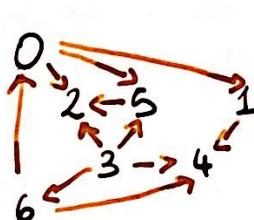
→ Shortest paths da Sorgenti multiple

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'unico direzione.



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

Ci diciamo quindi che in questo caso i archi 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. a x , allora v è F.C. a x .

Algoritmo di Kosaraju-Sharir

Base delle componenti fortemente connesse in G sono le steme 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:

0 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 visitore nodi non ancora visitati.
- Ignoro i nodi a cui è già associato 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 più arco di peso minore di nero.
- Ripetiamo finché $V-1$ vertici sono colorati di nero.

In pratica ...

l'algoritmo 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 prossimo all'albero T a meno che non crea 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

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

- V : faccio 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 contenenti 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 le UF se i due archi sono connesi
- Se non sono connesi, 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
delete min	E	$\log E$
Union	V	$\log^* V$
Connected	E	$\log^* V$

- Copio tutti gli Edges nella PQ
- Potrei dover far scendere il primo el per tutta l'altezza dell'heap
- Altezza dell'albero
- 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 esattamente un endpoint int.
- Ritetiamo 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, aggiungendolo 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 grafici NON orientati, in questo caso analizziamo i grafici **orientati**.

Ci sono diverse varianti di questo problema:

- **Sorgente Singola** Da un vertice s a tutti gli altri.
- **Collegamento Singolo** Da tutti i vertici ad un singolo vertice s .
- **Sorgente - collegamento** Da un vertice s ad un vertice t .
- **Tutte le coppie** 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)	// $v \rightarrow w$
int	From ()	// Vertice v
int	To ()	// Vertice w
double	weight ()	// peso dell'arco

Rappresentazione

Anche in questo caso usiamo la **lista di adiacenza**, da lista funzione come nel caso dei grafici orientati semplici, aggiungendo l'arco solo al nodo da cui esce e' 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 $s \rightarrow v$

Iterable <Directed Edge> pathTo (int v) // path da $s \rightarrow v$

Ovvio che queste due strutture sono dei semplici Arrays:

- $\text{distTo}[v]$ lunghezza dello SP
- $\text{edgeTo}[v]$ d'ultimo arco nullo SP da $s \rightarrow v$

edgeTo[] distTo[]

0	null	0
1	$s \rightarrow 1$ 0.32	1.05
2	$0 \rightarrow 2$ 0.26	0.26
3	$7 \rightarrow 3$ 0.37	0.97
:	:	:

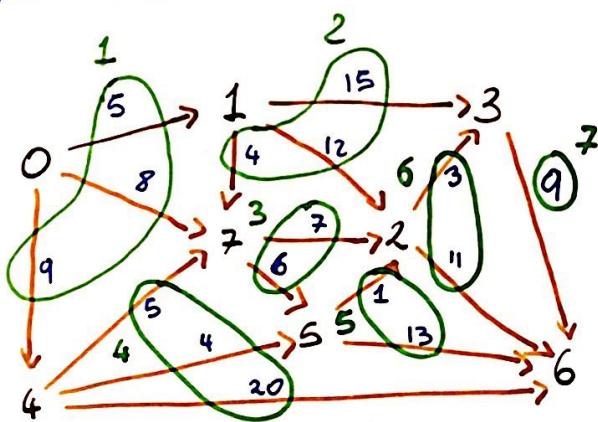
Algoritmo di Dijkstra

- 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 nel 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 $\text{disto}[w]$ $\text{edgeTo}[w]$.

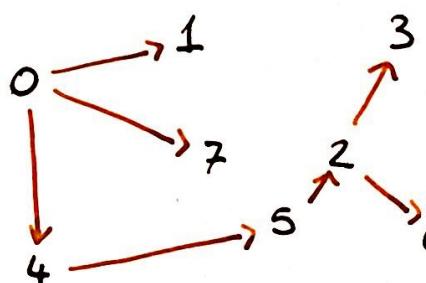
In poche parole... le procedure di rilessamento ci servono ad aggiornare l'arco che collega v a w SE ne troviamo uno di peso minore.



v	disto[v]	edgeTo
0	0.0	-
1	5.0	0->1
2	17.0	1->2
3	15.0	1->3
4	14.0	1->4
5	13.0	1->5
6	25.0	4->6
7	26.0	4->7
	39.0	5->6
	8.0	0->7

* Il primo passaggio ci dice di settare il primo nodo 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

