Orale Algoritmo 2

# Componenti fortemente connesse

## Ordine topologico

Un ordine topologico è un ordinamento lineare dei nodi di un grado tali che se contiene un arco (u,v), allora il nodo u appare prima di v, può essere visto come un ordinamento dei vertici in linea in modo da avere tutti gli archi uscenti che partono da un nodo di sinistra e arrivano a uno di destra, esso è possibile solamente quando il grafo è DAG, ovvero che è orientato e senza cicli. Un grafo è DAG quando la visita DFS non genera archi all’indietro.

### Algoritmo

| topSort()  sortNum=getOrder()-1  for i=0 to n-1;  getTopSort(i) | getTopSort(int source)  S=S U {source}  per ogni vicino di source  if(vicino non scoperto)  getTopSort(vicino)  ts[source]=sortNum--; |
| --- | --- |

L’algoritmo esegue una visita DFS per calcolare il tempo di completamento della visita di un nodo, una volta calcolato essa verrà messo in testa a una lista la quale verrà restituita alla fine. La complessità dell’algoritmo è O(n+m) perchè occorre lo stesso tempo per effettuare una visita DFS più un tempo costante per inserire i nodi nella lista. Quando un grafo è DAG, l’algoritmo produce sempre un’ordine topologico.

#### Dimostrazione

Dato un qualsiasi arco (u,v), quando esso verrà ispezionato dalla visita DFS, il nodo v non potrà essere non terminato, dato che sarebbe un antenato di u contraddicendo la tesi. Se v è non scoperto, diventa discendente di u e di conseguenza viene dopo nell’ordine topologico. Se invece è nero, la sua visita è già stata completata e quindi la sua posizione è già impostata. Dal momento che stiamo ancora visitando il nodo u, bisogna assegnargli una posizione, essa però sarà minore di quella di v. Quindi ogni arco (u,v) ha sempre il tempo di completamento di u maggiore di quello di v.

Componenti fortemente connesse

topSort()

sortNum=getOrder()-1

for i=0 to n-1;

getTopSort(i)

getTopSort(int source)

S=S U {source}

per ogni vicino di source

if(vicino non scoperto)

getTopSort(vicino)

ts[source]=sortNum--;

L’algoritmo esegue una visita DFS per calcolare il tempo di completamento della visita di un nodo, una volta calcolato essa verrà messo in testa a una lista la quale verrà restituita alla fine. La complessità dell’algoritmo è O(n+m) perchè occorre lo stesso tempo per effettuare una visita DFS più un tempo costante per inserire i nodi nella lista. Quando un grafo è DAG, l’algoritmo produce sempre un’ordine topologico.

Dimostrazione

Dato un qualsiasi arco (u,v), quando esso verrà ispezionato dalla visita DFS, il nodo v non potrà essere non terminato, dato che sarebbe un antenato di u contraddicendo la tesi. Se v è non scoperto, diventa discendente di u e di conseguenza viene dopo nell’ordine topologico. Se invece è nero, la sua visita è già stata completata e quindi la sua posizione è già impostata. Dal momento che stiamo ancora visitando il nodo u, bisogna assegnargli una posizione, essa però sarà minore di quella di v. Quindi ogni arco (u,v) ha sempre il tempo di completamento di u maggiore di quello di v.

Componenti fortemente connesse

Una componente fortemente connessa è un sottografo massimale di un grafo (gli insiemi dei nodi e degli archi sono sottoinsiemi di quelli del grafo) in cui, per ogni coppia di nodi, esiste un cammino orientato che non esce dalla propria componente. Q

Una componente fortemente connessa è un sottografo massimale di un grafo (gli insiemi dei nodi e degli archi sono sottoinsiemi di quelli del grafo) in cui, per ogni coppia di nodi, esiste un cammino orientato che non esce dalla propria componente. Quindi all’interno di una componente i nodi sono tutti connessi. Questo vuol dire che se isolo una SCC dal grafo e si visita in qualsiasi modo, essa visiterà tutti i nodi appartenenti a essa per via della definizione di SCC. Un grafo può contenere più SCC connesse tra loro, esse formano il grafo delle SCC, un grafo DAG (quindi ammette ordine topologico) i cui nodi rappresentano le SCC. Esso viene utilizzato perchè gli archi del grafo connettono nodi di una componente connessa a quelli di un’altra, rappresentano quindi il passaggio che l’algoritmo utilizza per passare da una componente a un’altra.

### Proprietà delle SCC

* Date due SCC c1 e c2 connesse nel seguente modo: c1 → c2, la visita DFS del primo nodo di c1 termina dopo quella dei nodi di c2, con questa proprietà, i nodi che non hanno archi entranti terminano la visita per ultimi e viceversa per i nodi che non hanno archi uscenti. Questa regola permette di definire un ordine di terminazione dei nodi nella visita DFS, detto ordine di fine visita, che corrisponde all’ordine topologico alla rovescia se il grafo è DAG;
* Date due componenti c1 e c2 e le rispettive coppie di nodi u1, v1 e u2, v2, se esiste un arco che connette u1 a u2, allora non può esiste un arco che connette v2 a v1;
* Date due componenti connesse c1 e c2, se esiste un arco (u,v) in cui u appartiene a c1 e v a c2, allora f(c1)<f(c2) in cui f è la minima posizione di ogni nodo appartenente alla SCC data;
* Date due SCC c1 e c2 connesse e distinte, se esiste un arco (u,v) appartenente al grafo trasposto in cui u appartiene a c1 e v a c2, allora (fc1)>f(c2)

### Algoritmo di Kosarasu

l’algoritmo di Kosarasu è un algoritmo che calcola le SCC in un grafo orientato, esso utilizza il grafo delle SCC perchè:

* il grafo delle SCC è DAG;
* dato un grafo formato da due componenti c1 e c2 e un arco (u,v) tale che u appartiene a c1 e v in c2, se visitiamo in DFS il grafo completamente, essa produrrà dei tempi di terminazione tali che quelli di nodo di c1 sono maggiori rispetto a quelli di c2, di conseguenza terminano dopo rispetto a questi ultimi.

Dato che il grafo delle SCC è un DAG, esso ammette ordine topologico, infatti se si visita in DFS questo grafo, si otterrà un certo tempo di terminazione. Il grafo delle SCC serve quindi come dimostrazione del ragionamento dell’algoritmo: dal grafo delle SCC si quindi un ordine topologico, si può osservare che, ribaltando l’ordine topologico, si ha che una situazione in cui la visita di un nodo parte, si ferma su quella componente e dal ciclo esterno partirà un’altra visita da un’altro nodo. Si vuole quindi forzare la visita a passare dai tempi di terminazione inversi scoperti in precedenza, per farlo si utilizza un grafo trasposto, un grafo con gli stessi nodi dell’originale ma coi nodi girati al contrario. Se la seconda DFS si fa prendendo le sorgenti dall’ordine di fine visita inverso, la visita parte da quelle componenti e si ferma in essa perchè, dato che parte da un nodo di una certa SCC, la visita prende tutti i nodi di quella SCC, per tutti i nodi che stanno “al confine” della componente, se esiste un’arco che permette alla visita di uscire dalla SCC per andare in un altra, in teoria esce e quindi non riuscirebbe a fermarsi, quindi viene utilizzato il grafo delle SCC per “delimitare” i nodi appartenenti a una stessa SCC e utilizzo il grafo trasposto per impedire alla visita di uscire da essa (altrimenti non riuscirebbe a capire quali nodi apparterrebbero alla SCC) perchè, quelli che sono archi uscenti dalla SCC nel grafo normale, in quello trasposto sono archi entranti, quindi dal punto di vista della SCC non cambia niente, dato che rimane tale indipendentemente dal fatto che il grafo sia trasposto o meno, cambiano invece gli archi tra una componente e un’altra, quindi tutti gli archi uscenti diventano entranti e viceversa permettono di conseguenza di fermare la visita all’interno delle SCC, al termine tutte le chiamate vengono “rilasciate” fino ad arrivare alla sorgente e vengono segnate nella rispettiva SCC. Infine il ciclo esterno sceglierà il prossimo nodo dell’ordine di terminazione e ricomincerà la visita, esso apparterrà a un’altra SCC e quindi visiterà tutti i nodi nei “limiti”.

esso la fa eseguendo tre passaggi:

* Calcola l’ordine di fine visita DFS visitando il grafo;
* Calcola il grafo trasposto, un grafo simile a quello originale in cui tutti gli archi hanno la direzione invertita, i vertici u e v sono raggiungibili l’un l’altro in un grafo se e solo se lo sono anche in quello trasposto.
* Visita del grafo trasposto seguendo l’ordine di fine visita dato, essa può essere sia in ampiezza che in profondità.

L’algoritmo di Kosarasu calcola sempre le componenti connesse in un grafo.

### Dimostrazione

L’ipotesi induttiva è che le prima k alberi DFS corrispondono a delle SCC, la base si trova ponendo k=0, quindi è banale. Supponendo che i k alberi siando SCC e si considera il (k+1) esimo albero prodotto, supponendo che il vertice u di quest’ultimo albero appartenga alla componente C, si ha che f(C)<f(C)’ per ogni componente C’ diversa da C non ancora visitata. Quando viene visitato il vertice u, tutti gli altri nodi della componente C non sono ancora stati scoperti e quindi sono discendenti di u, tutti gli archi del grafo trasposto che partono da C devono essere diretti verso componenti già visitate. In questo modo nessun vertice di qualsiasi componente diversa da C sarà discendente di u durante la visita del grafo trasposto. Di conseguenza i vertici di ogni albero nel grafo trasposto radicato in u formano una SCC.

P e NP

# Dimostrazione della complessità dello zaino

La complessità dell’algoritmo che risolve il problema dello zaino è =(n\*C) in cui n è il numero di oggetti e C è la capacità dello zaino, essa è lineare sul numero di celle della tabella ma per quanto riguarda la capacità C, quanto spazio si occupa? si occupa spazio logC, dato che viene scritto in binario,al posto dello spazio lineare per gli n oggetti, Quindi la complessità O(nC) diventa esponenziale proprio per questo motivo, dato che C occupa spazio logaritmico e quindi la complessità equivale a O(2^C).

# Introduzione ai problemi difficili

I problemi trattati finora ammettono algoritmi la cui complessità è deterministica/polinomiale, sfortunatamente esistono problemi in cui può esistere un algoritmo che richiede un tempo esponenziale oppure può non esserci affatto.I problemi possono essere di:

* Decisione: quando restituisce un valore binario, permettono la verifica di un’istanza x dell’input del problema;
* Ricerca: data un’istanza x, il problema deve restituire una soluzione s tale che la coppia (x,s) appartenga al problema;
* Ottimizzazione: data un’istanza x il problema deve restituire la soluzione migliore possibile. Ogni problema di ottimizzazione può essere espresso in forma decisionale, questo significa che il problema di decisione non è tanto difficile quanto quello di ottimizzazione e che se riuscissimo a trovare un algoritmo che risolve il problema di ottimizzazione, allora riusciamo a risolvere anche il rispettivo problema di decisione.

La classe P è la classe contenente i problemi risolvibili in tempo polinomiale, ogni problema ha sia una dimostrazione matematica che un algoritmo buoni, La classe NP invece è formata da problemi in cui la dimostrazione è buona, questo significa che è possibile verificare la “bontà” di una soluzione data un’istanza x attraverso un’ulteriore oggetto detto certificato, il costo della verifica é polinomiale e può essere utilizzato per caratterizzare la complessità del problema stesso. Solitamente i problemi NP sono nella forma “Esiste questo tale che …”, questo ragionamento è applicabile anche ai problemi decisionali di P, ad esempio: esiste un MST tale che il suo peso sia minore del massimo di una variabile data? Questo problema potrebbe quindi essere un problema NP, ma si sa che MST è un problema P, in questo caso è possibile ignorare la verifica e calcolare MST. Questo problema è quindi un problema in P ma è possibile vederlo anche come un problema NP, questo è possibile farlo con tutti i problemi in P ignorando la verifica ( e quindi tutte le informazioni aggiuntive) e effettuandola con un algoritmo polinomiale che conosciamo. Si può quindi dire che P è contenuto in NP, NP significa non deterministico/polinomiale e si riferisce ai metodi utilizzati dai problemi di questa classe: oltre alle normali istruzioni, essi hanno anche delle funzioni che “indovinano” un percorso e da lì prosegue la computazione. E’ possibile dividere un’algoritmo NP in due parti:

* Fase non deterministica: si sfrutta una funzione che “indovina” la costruzione di un certificato che descrive una soluzione data un’istanza;
* Fase deterministica: l’algoritmo verifica che il certificato sia quello dell’istanza data.

I problemi NP sono tutti risolvibili utilizzando una ricerca esaustiva e scegliendo la soluzione migliore, tuttavia la complessità è esponenziale a meno che non vi siano determinati input.

La caratterizzazione dei problemi più difficili è data da NP-completezza: se esiste un algoritmo polinomiale che risolve un problema NP, allora esiste anche per gli altri. Per capire quanto un problema è più difficile di un altro si utilizza la riconducibilità, esse permette di conservare le istanze vere/false per un problema anche per quello a cui riconduce: dati due problemi P1 e P2 (cond(P1,P2)) se esiste una funzione che dato un’input di P1 ne restituisce uno di P2, essa deve soddisfare le seguenti proprietà:

* f deve essere polinomiale;
* per ogni coppia input-soluzione (i,s) appartiene a P1 se e solo se (f(i),s)) appartiene a P2;

Di conseguenza se esiste un algoritmo per risolvere P2, lo si può utilizzare anche per P1, questo significa che P2 è difficile almeno quanto P1, quindi se P2 è in P, allora lo è anche P1. Un problema P1 è detto NP-Hard se ogni problema p in NP è riconducibile a P in modo polinomiale, riuscendo almeno in parte a risolverlo utilizzando parte del metodo di P1. Un problema é NP completo quando appartiene a NP ed è NP-Hard, esso contiene tutti la difficoltà della classe NP e una sua risoluzione porta ad avere P=NP. Di fronte a un problema NP si possono utilizzare i seguenti approcci:

* ricerca esaustica: anche se migliorabile con euristiche, ha complessità esponenziale;
* ricerca locale: si prende un soluzione e si migliora, la soluzione ottenuta sarà molto vicina a quella ottimale;
* approssimazione: si cerca una soluzione non ottimale ma molto vicina a essa.

# Algoritmo di Floyd-Warshall

L’algoritmo di Floyd\_warshal è un algoritmo che permette il calcolo di cammini minimi per ogni coppia di nodi in asserza di cicli negativi nel grafo. Pero ogni k si indica d(x,y,k) come la distanza x-y passante per nodi minori o uguali a k (cammino k.vincolato), per ogni coppia di nodi di nodi esiste sempre un cammino k-vincolato, in più ogni sottocammino del cammino è esso stesso un cammino k-vincolato, quindi, dato un cammino c(x,y,k), si ha che:

* se c non contiene k, allora c è anche un cammino k-1 vincolato;
* se invece c contiene k, allora c è formato dai cammino x-k e k-y che sono cammini k-1 vincolati;

In base a questi due punti, si ha che:

d(x,y,k)=min( d(x,y,k-1), d(x,k,k-1) + d(k,y,k-1))