Argomenti Orale Algoritmi 1

# Algoritmo di Kosarasu

L’algoritmo di Kosarasu è un algoritmo che permette di trovare le componenti fortemente connesse in un grafo orientato, esso le sa attraverso tre passaggi:

1. Calcolo dell’ordine di fine visita (ordine in cui terminano i nodi, equivalente all’ordine topologico al contrario se il grafo è aciclico) con la visita DFS (non si può ottenere con la visita BFS);
2. Calcolo del grafo trasposto, ovvero un grafo con lo stesso numero di nodi del precedente ma con tutti gli archi invertiti;
3. Visita del grafo trasposto utilizzando l’ordine di fine visita (la seconda visita è indifferente perché si basa su quest’ultimo).

## Visita BFS

La visita BFS è un tipo di visita di un grafo in cui vengono scoperti e visitati tutti i vicini del nodo dato per poi proseguire fino a quando non li ho scoperti tutti. Per fare ciò, la visita si serve di una coda in cui memorizzare ed estrarre i nodi da visitare. Una proprietà di questa visita è che i nodi inseriti nella coda corrispondo a quelli del livello successivo dell’albero di visita rispetto a quello che si sta visitando.

## Visita DFS

La visita DFS è un tipo di visita in cui i nodi vengono scoperti a partire dal “fondo” e man mano risalendo il percorso fatto dal grafo. L’algoritmo, alla scoperta di un nuovo nodo, esegue una chiamata ricorsiva su esso e mettendo in pausa la visita del precedente (la visita però non è ancora terminata). È possibile sfruttare questa peculiarità utilizzandola quando si è in presenza di cicli.

## Rilevazione di cicli nei grafi

La rilevazione di cicli in grafi orientati e non è possibile attraverso gli alberi di visita ricavabili dalle visite, da essi infatti si può constatare che:

* L’albero di visita di un grafo orientato è formato da 3 tipi di archi tra quelli non considerati dall’albero: gli archi in avanti/all’indietro permettono di salire/scendere di livello mentre quelli trasversali connettono due nodi fratelli.
* Dal punto precedente, si può dire che un grafo orientato ha un ciclo se è presente un arco all’indietro che connette due nodi.
* In un grafo non orientato, l’albero di visita non considera solo gli archi trasversali o quelli in avanti/all’indietro a seconda dell’algoritmo di vista utilizzato (rispettivamente BFS per il primo caso e DFS per il secondo), si può quindi dire che un grafo ha un ciclo se l’albero BFS non considera un arco trasversale oppure se l’albero DFS non ne considera uno in avanti/all’indietro.
* La peculiarità della visita DFS permette la rilevazione di cicli in ogni tipo di grafo.

La ricostruzione d’un ciclo avviene tramite l’array in padri, in cui si continua a inserire nodi finché non arrivo alla radice oppure arrivo al padre del nodo di partenza.

## Ordine Topologico

L’ordine Topologico è una rappresentazione del grafo orientato in cui i nodi vengono disposti in odo da avere tutti gli archi che partono da un nodo di sinistra e arrivano a un di destra. Per permettere ciò, il grafo deve essere aciclico, ovvero non deve presentare cicli al suo interno (DAG).

## Componenti Fortemente Connesse (SCC)

Una Componente Fortemente Connessa di un grafo orientato è una componente in ogni per ogni coppia di nodi appartenente a essa, esiste un cammino orientato. Sfruttando queste componenti è possibile, dato un grafo orientato, ricavare il grafo delle SCC, ovvero una rappresentazione del grafo originale in cui si utilizzano le SCC come singoli nodi. Il grafo delle SCC è sempre DAG e ha la seguente proprietà: se esiste un arco che parte da una SCC A e arriva in B, l’ultimo nodo a terminare sarà in A, questo succede perché.

# Problema del commesso viaggiatore

Il problema del commesso viaggiatore (TSP) è un problema difficile in cui, dato un grafo non orientato, completo e con pesi positivi, bisogna percorrere tutti i nodi minimizzando le distanze. Nel suo complesso TSP non ammette approssimazioni per nessuna R, esiste però un caso in cui si può ottenere una soluzione vicina a quella ottimale: il TSP metrico, ovvero il TSP in cui è presente la disuguaglianza triangolare applicata agli archi (il lato più lungo di un triangolo deve essere minore o uguale della somma degli altri 2).

Supponiamo che S\* sia la soluzione ottimale del TSP metrico. Dato che S\* è un cammino chiuso, se togliamo un arco qualsiasi da esso, si ottiene un albero ricoprente T. Il costo di T è sempre minore o uguale a quello di S\*, dato che, togliendo un arco, tutti gli archi sono positivi e possono toglierne uno di peso 0 nel caso limite.

( c(S\*) >= c(T) )

Un primo tentativo di approssimazione si può fare calcolando MST del grafo (A): in questo modo però il commesso percorrerebbe tutti gli archi 2 volte, dato che A non è un cammino chiuso. Dato che A è MST, si può dire che c(T) >= c(A) perché A è il più piccolo tra tutti gli alberi ricoprenti.

Prendiamo ora il cammino più lungo che il commesso può percorrere (W) e, confrontandolo col cammino di A, si ha che c(W)= 2\*c(A) per il motivo spiegato in precedenza. Dato che il grafo è completo, è possibile “tagliare” e, se si può fare, conviene farlo? Sì e in questo modo la lunghezza del tour del commesso diminuisce. Con il nuovo percorso tagliato (W’) si ha che c(W’) <= c(W) per disuguaglianza triangolare.

Dato che prendere le scorciatoie conviene, prendiamole sempre! Per farlo si utilizza l’ordine dei nodi scoperti della visita DFS, in questo modo si ottiene un tour S in cui c(S) <= c(W)<=2\*c(A) dato che prendo tutte le scorciatoie per diminuire la distanza. Dato che, come detto prima, c(S\*)>=c(T)>=c(A), si può dire che c(S)<=c(W)=2\*c(A)<=2\*c(S\*). In conclusione, dato che il tour di S non può essere peggiore del doppio di quello di S\*, si può dire che c(S)<=2\*c(S\*) in cui 2 è il cosiddetto fattore di approssimazione. Tuttavia non riesco a dimostrare se esiste un percorso migliore calcolabile col medesimo modo ma, dato che MST non ha a che fare con i cammini minimi, non riuscirei a dimostrare che la soluzione trovata sia ottimale.

TEORIA DELLE COMPLESSITà E PROBLEMI DIFFICILI

La teoria della complessità é un ramo dell’informatica che studia la complessità dei problemi e le risorse minime per la risoluzione delle sue istanze, essa non dipende nè dal tipo di linguaggio di programmazione, nè dal tipo di calcolatore utilizzato. La complessità di un problema é data dalla complessità dell’algoritmo più efficiente che riesce a risolverlo, essa è detta polinomiale quando viene risolto in un tempo accettabile.

La classe P é quella classe che contiene problemi in cui la dimostrazione é possibile sia matematicamente che “empiricamente”, ovvero con l’ausilio di algoritmi con complessità accettabile. Non tutti i problemi, però, non sono ancora risolvibili o addirittura si pensa non esista una soluzione, nonostante esista una dimostrazione matematica semplice, questo tipo di problemi appartiene alla classe NP (i cosiddetti “problemi difficili”).

I problemi di classe P sono una relazione P che contiene I x S, in cui I é l’insieme delle istanze d’ingresso mentre S quello delle soluzioni, quindi, data un’istanza d’ingresso i e una soluzione s, si può dire che P restituisce 1 se (i,s) appartiene a P, restituirà 0 altrimenti. Da queste informazioni, possiamo dire che i problemi possono essere:

* Di decisione: problemi in cui la soluzione é booleana, essi richiedono se, data l’istanza, essa soddisfa certe proprietà.
* Di ricerca: problemi in cui, data un’istanza, si restituisce una soluzione ammissibile, ovvero quando (i,s) appartiene a P.
* DI Ottimizzazione: problemi in, data l’istanza, si deve restituire la miglior soluzione possibile, ovvero quella ottimale tale che (i, s\*) appartiene a P in cui s\* é la migliore delle soluzioni.

Dai punto elencati prima, é possibile intuire che:

* un’ istanza d’un problema di decisione può essere ricavata da un’ulteriore istanza dello stesso problema di ricerca/ottimizzazione.
* Nel caso in cui esiste una soluzione a un problema di ottimizzazione, allora esiste anche una soluzione per il relativo problema di decisione utilizzando lo stesso algoritimo.
* un problema di decisione non può essere più difficile del relativo problema di ottimizzazione.

La verifica di una soluzione d’un problema data un’istanza si fa attraverso un testimone, esso dipende dall’istanza del problema che si considera (ad esempio, nel caso dello zaino non frazionario, il testimone é un insieme di oggetti che esso può contenere), il quale ne verrà verificata l’esistenza e, nel caso sia presente, se fatta in tempo polinomiale.

PROBLEMA: VERTEX COVER

Dato un grafo G, R è un ricoprimento di vertici se, per ogni arco (u,v).

R è un sottoinsieme dell’insieme dei nodi.

Esempio: In un museo in cui i nodi sono le porte e gli archi sono le sale, si vuole mettere un certo numero di custodi, ognuno che controlla le sale adiacenti alla porta cui é assegnato. Voglio che ci sia almeno un custode per ogni sala del museo.

Il problema di ottimizzazione é il ricoprimento R del grafo in cui R è il minimo numero di custodi.

Il problema di decisione, invece, ci dice se esiste un ricoprimento R in cui la cardinalità é minore di un valore k. Il testimone del problema è il ricoprimento stesso.

La verifica del problema di decisione é, data un istanza x=(grafo,k), per dimostrare che la ricerca del testimone avvenga in tempo polinomiale. Si guada ogni arco e si verifica se i nodi appartengono al ricoprimento (è un problema in P perché ci metto “poco tempo” a scorrere il grafo).

Il fatto di poter utilizzare un unico algoritmo per risolvere sia il problema di ottimizzazione che il relativo problema di decisione, é possibile quindi, dati problemi A e B in cui B è contenuto in A e un’istanza Xa, si converte Xa in un istanza Xb e la si dà in pasto al problema B. In questo modo se ottengo una soluzione per il problema B, la ottengo anche per A. La difficoltà del problema A sarà al massimo pari a quella di B e non può essere peggio. Questa trasformazione (fatta in tempo polinomiale) é detta riduzione in cui tutte le istanze vere vengono conservate (ogni istanza del primo problema rimane tale anche per il secondo).

I problemi NP hard (problemi difficili per la classe NP) sono particolari problemi in cui, per ogni altro problema A di NP, é possibile ridurre in modo polinomiale A a quello principale. In poche parole, un problema A è NP hard se per ogni problema A’ in NP, riesco in parte a risolverlo in modo polinomiale e in parte col metodo utilizzato da A. Questa tecnica è utile perchè in questo modo posso risolvere vari problemi in NP a meno di una prima risoluzione che, in essenziale, può essere polinomiale.

Se il problema é NP hard é anche in NP, il problema é NP completo, ovvero contiene tutta la difficoltà della classe NP. Nel caso venisse risolto un problema NP completo in tempo polinomiale, in automatico verrebbero risolti tutti i problemi della classe NP in tempo polinomiale e, di conseguenza, l’insieme P sarebbe uguale a NP. Se volessimo dimostrare che P é diverso da NP, dovremmo prendere un problema NP completo e dimostrare che non è possibile risolverlo in tempo polinomiale. Si può dire che, anche se in realtà non sappiamo niente a riguardo, i problemi NP completi non hanno soluzione ricavabili in tempo polinomiale.

Cosa si fa quando un problema è difficile?

Tra le possibili soluzioni vi è:

* la ricerca esaustiva, però ha complessità esponenziale (anche se migliorabile con euristiche);
* la ricerca d’ulteriori sottoproblemi più facili;
* la ricerca locale, ovvero cerco una soluzione un po’ approssimata/random e cerco di migliorarla, la soluzione ottenuta potrebbe non essere ottimale ma molto vicina a esso.
* Approssimazione: Trovo una soluzione non ottimale ma abbastanza buona.

APPROSSIMAZIONE

Quanto deve essere buona l’approssimazione per ottenere una soluzione buona?

Esempio: calcolo di un minimo generico

S\* è la soluzione ottimale del problema ed è il minimo, ogni altra soluzione S sarà maggiore o uguale a essa

Per capire quanto è complessa la soluzione, la limitiamo superiormente supponendo che val(S\*)<=p\*val(S) in cui:

* se p=1, allora la soluzione S è identica a S\*;
* se p>1, allora S è maggiore di S\*

Il valore di p si trova rapportando i valori delle due soluzioni:

p>=val(S)/val(S\*)

Esempio: problema di massimizzazione

S\* è la soluzione ottimale e il massimo, tutte le altre soluzione S sono minori o uguali

Limitando inferiormente la soluzione, si ha che p<= (1/p)\*val(S\*)<=val(S)

Se riusciamo a limitare la soluzione S all’interno del “range”, si può ricavare che:

val(S\*)/val(S)<=p

Combinando questa soluzione con quella dell’esempio precedente, si ha che:

p>=max(val(S)/val(S\*),val(S\*)/val(S)) DEFINIZIONE DI RHO RISPETTO A PROBLEMI DI MINIMIZZAZIONE/MASSIMIZZAZIONE

ovvero p è maggiore o uguale del massimo dei due rapporti.

ALGORITMO D’APPROSSIMAZIONE

approssimazione vertex cover:

R=0, A=0

sceglie (u,v) appartenente a E

R= R U {u} U {v}

A= A U{u,v}

elimina tutti gli archi uscenti da u

elimina tutti gli archi uscenti da v

Vengono tolti gli perchè hanno almeno uno dei due estremi nel ricoprimento

Per ricoprire il grafo iniziale, devo anche ricoprire il grafo RA

quindi:

R\*g(ottimale) >= R\*a(stimata) = R/2(approssimata)

Questo vuol dire che R\*a<= R <=2R\*g

2 + il fattore d’approssimazione del problema

PROGRAMMAZIONE DINAMICA

La programmazione dinamica è una tecnica algoritmica in cui si risolve un determinato problema dividendolo in sottoproblemi più piccoli basandosi sull’utilizzo della sottostruttura ottima, ovvero che, data una struttura dati con determinate qualità, quelle stesse qualità valgono per una qualsiasi sottostruttura della struttura stessa.

Un esempio di algoritmo che utilizza la programmazione dinamica in cui il numero totale di conigli in un determinato anno è dato dalla somma tra le coppie di conigli nuove più quelle che hanno figliato.

Fibonacci(n):

if(n<=0) return 1;

else return Fibonacci(n-1) + Fibonacci(n-2)

Questa tecnica però, se implementata come nell’esempio, ricalcola valori già calcolati in precedenza (che è uno spreco di tempo), esiste una soluzione a questo? Si, la Memoization, ovvero si utilizza una struttura dati (solitamente un array) in cui memorizzare i valori calcolati a una determinata iterazione, in questo modo l’algoritmo calcolerà i valori solo una volta e andrà a prenderseli quando necessario.

Fibonacci(n):

a[0]=1

a[1]=1

for i=2 to n-1

a[i]=a[i-1] + a[i-2]

MASSIMO SOTTOINSIEME INDIPENDENTE

Il massimo sottoinsieme indipendente è un problema in cui, dato un grafo lineare con nodi pesati non negativamente, si deve prendere l’insieme di nodi con somma dei pesi massima evitando di considerare quelli adiacenti.

0--1--2--3--4 La soluzione ottimale S per questa istanza del problema é 03 perché, oltre a non essere 4 3 2 9 4 adiacenti, la somma dei loro pesi è il massimo possibile, ovvero 13

POSSIBILI SOLUZIONI:

Greedy: la soluzione greedy viene ricavata prendendo tutti i nodi non adiacenti di grado massimo (024 in questo caso) però non è ottimale;

Divide et Impera: questa soluzione viene ricavata dividendo sempre a metà il grafo lineare cercando il nodo massimo del sottoproblema, tuttavia questa tecnica può portare a soluzioni non combinabili dato che i nodi con i pesi più grandi potrebbero essere adiacenti.

SOLUZIONE: Programmazione Dinamica

MSI è formato da due sottoproblemi:

1. Il nodo appartiene a S: dato che non posso prendere il nodo adiacente, lo salto e considero il sottoproblema partendo dal successivo di quest’ultimo, ovvero il terzo nodo. Quindi esiste una soluzione S’ = S- {nodo i} ottimale per il sottoproblema citato e dimostrabile per assurdo: data una soluzione S’ non ottimale, si ha che esiste una soluzione S’’ tale che val(S’’)>=val(S’), quindi la S’’ + {i} è ottimale per il sottoproblema 1 dato che val(S’’+{i}) >val(S) i n modo assoluto.
2. Il nodo non appartiene a S: S è una soluzione ottimale anche per il sottoproblema che parte dal nodo adiacente a quello tutt’ora considerato. Dimostrandola per assurdo: si ha che S non è ottimale se e solo se esiste S’ tale che val(S’) > val(S). Questa implicazione però non può andare bene perchè si contraddice, quindi si conclude che S è soluzione ottimale per il sottoproblema 2.

Il calcolo della soluzione massima, dato il testo e i due punti descritti in precedenza, è dato dal massimo delle due sottosoluzioni:

val(i) = max( val(i-2) + w(i), val(i-1))

val(i), val(i-2) e val(i-1) sono rispettivamente le soluzioni ottimali del sottoproblema odierno, del punto 1 e del punto 2.

PROBLEMA DELLO ZAINO NON FRAZIONARIO

Lo zaino non frazionario è un problema in cui, dati n oggetti non divisibili con valori e volumi e uno zaino di capacità C, si deve creare un insieme di oggetti la cui somma dei valori è massima è la somma dei volumi è minore della capacità. Il problema però è scegliere quali oggetti prendere, proviamo a vedere con delle soluzioni:

DATI: 3 oggetti e capacità 10

vol1=6 val1=9 OTTIMALE: 10 (oggetti 2 e 3)

vol2=5 val2=5

vol3=5 val3=5

Greedy: la tecnica greedy per lo zaino prende, tra gli n oggetti, quelli con il rapporto valore/volume più alti diminuendo ogni volta lo spazio. In questo esempio, la soluzione é 9 che non è quella ottimale, dato che non è possibile aggiungere altri oggetti all’interno. Tuttavia la soluzione greedy non sarebbe comunque ottimale anche se guardassimo il valore specifico d’ogni oggetto. Il problema viene dal fatto che, riempiendo lo zaino col primo oggetto, ci resta dello spazio che non viene utilizzato perchè non si può riempire con altri oggetti.

Per risolvere questo problema in modo efficiente, si utilizza la programmazione dinamica, per farlo si introduce il concetto di sottostruttura ottima: Data una soluzione S ottimale per il problema P(C,i) si hanno i seguenti casi:

1. se S non contiene i allora S è ammissibile e ottimale anche per il problema P(c, i-1);
2. se invece S non contiene i, allora esiste una soluzione S’=S-i ammissibile e ottimale per il problema P(C-vol(i), i-1);

Entrambi i punti sono dimostrabili per assurdi:

1- Se S non é ottimale per il sottoproblema, allora esiste una soluzione S’ per il problema P(C,i-1) tale che val(S’)>=val(S). In questo modo si dimostra che S’ è una soluzione ammissibile e ottimale anche per il problema P(C,i).

2- Se S non è ottimale per il sottoproblema, allora esiste una soluzione S’’ tale che il val(S’’)>=val(S’) per il problema P(C-vol(i),i-1). Di conseguenza, la soluzione S’’ U i é ottimale per il problema P(C,i).

Nel caso dovessi riempire uno zaino che ha una capacità diversa da C, la soluzione ottenuta contiene anche la soluzione relativo agli zaini più piccoli, in questo modo posso riempire meglio lo zaino e quindi trovare la soluzione ottimale. Per capire come costruire la soluzione, supponiamo di avere una soluzione S ottimale per il problema P(C,n), ovvero la capacità C da riempire. Quindi ci si domanda:

L’oggetto n appartiene alla soluzione? se S è ottimale, deve per forza contenere n? Metto l’oggetto nello zaino oppure no?

Da queste tre domande, possiamo ricavare due situazioni:

* CASO n appartenente a S: L’oggetto n può stare nello zaino e quindi lo prendo, dopo considererò i restanti oggetti per riempire la capacità rimanente;
* CASO n non appartenete a S: Dato che lo spazio a disposizione non basta, scarto l’oggetto e vado a vedere e gli altri possono starci.

Lo spazio rimanente per i restanti n-1 oggetti è quindi deterministico: sarà C-vol(i) nel caso n venga messo nello zaino, altrimenti sarà C, dato che n è troppo voluminoso per essere inserito.

ALGORITMO

PASSO BASE: Con 0 oggetti a disposizione, l’algoritmo non fa niente qualunque sia la capacità dello zaino:

for c=0 to C

val(c,0)=0;

dato che 0 oggetti, l’algoritmo non inserisce niente nello zaino e quindi il valore della soluzione sarà 0 di conseguenza. (in poche parole se non ho niente da prendere, non porto via niente).

PASSO INDUTTIVO: Vi sono più capacità da guardare per ogni oggetto:

for obj=1 to n

\*for c=0 to C

if(vol(obj)>cap) val(c,obj)=val(c,obj-1);

else

value1=val(c-vol(obj), obj-1)

value2=val(c,obj-1)

val(c,obj) = max(value1 ,value2 )

\* Prendiamo l’oggetto obj o no? Se obj è troppo grande per lo zaino, non lo prendiamo, quindi:

* se obj é più grande dello spazio che resta, la soluzione attuale equivale alla soluzione calcolata in precedenza ( quella con lo stesso spazio ma con obj-1 oggetti).
* se invece obj è più piccolo, la soluzione sarà data dal massimo dei valori delle soluzioni dei due sottoproblemi (vedi paragrafo precedente per spiegazione)

Esempio: 5 oggetti e capacità 10

| Oggetto | Volume | Valore |
| --- | --- | --- |
| 1 | 6 | 9 |
| 2 | 5 | 5 |
| 3 | 5 | 5 |
| 4 | 7 | 15 |

**GRASSETTO**: Caso Base

| obj\cap | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 9 | 9 | 9 | 9 |
| 2 | 0 | 0 | 0 | 0 | 0 | 5 | 9 | 9 | 9 | 9 | 9 |
| 3 | 0 | 0 | 0 | 0 | 0 | 5 | 9 | 9 | 9 | 9 | 10 |
| 4 | 0 | 0 | 0 | 0 | 0 | 5 | 9 | 15 | 15 | 15 | 15 |

Il problema dello zaino è un problema di massimizzazione, cioè che la soluzione ottimale è data dall’insieme di valori la cui somma è la massima possibile per quel determinato input, di conseguenza:

* il valore (i,j) d’un problema è dato dal massimo tra i valori dei suoi sottoproblemi;
* Prenderò sempre la strada che mi porterà ad avere il valore più alto.

La complessità dell’algoritmo è O(nC) in cui n è il numero di oggetti e C è la capacità dello zaino. Questa è una situazione in cui l’algoritmo dipende da un valore dell’input e questo è un problema perchè è proporzionale al numero di iterazioni dell’algoritmo.

Quanto spazio occupiamo per scrivere la capacità C? Lo spazio occupato è logaritmico, ovvero logC mentre, per esprimere gli oggetti, lo spazio è almeno lineare.

La complessità è quindi esponenziale nell’input, nella sua lunghezza e nei bit che lo rappresentano, viene così detta pseudo-polinomiale perchè non risolvibile in tempi accettabili dato che è sempre legato all’input n in ogni istanza del problema.