Riassunto Algoritmi 2

# I grafi

Un grafo è una struttura dati, solitamente indicata come la generalizzazione di un albero, che consiste in una coppia G=(V,E) in cui V è l’insieme dei nodi (detti anche vertici) e E è un insieme di coppie di nodi VxV, detti anche archi. Un grafo può essere:

* Orientato: gli archi dell’insieme E sono unidirezionali;
* Non orientato: gli archi dell’insieme E sono bidirezionali.

La relazione che lega archi e vertici è detta incidenza, un arco (u,v) infatti è incidente sui vertici da cui è composto (ovvero u e v). Inoltre, se il grafo è orientato, l’arco (u,v) è detto uscente per u ed entrante per v. L’ordine di un grafo è dato dal numero di vertici contenuti in esso, il grado di un nodo è dato dal numero di archi incidenti a esso, nei grafi orientati si può distinguere il grado entrante e quello uscente, ovvero il numero di archi incidenti rispettivamente in entrata e in uscita. Dato un arco (u,v), il vertice v è adiacente a u se:

* nei grafi non orientati, u è a sua volta adiacente a v, può contenere al massimo n(n-1)/2 archi;
* nei grafi orientati, se l’arco (u,v) è incidente, allora u è adiacente a v ma non viceversa, può contenere al massimo n(n-1) archi.

NOTA: n indica il numero di nodi appartenenti al grafo.

## Rappresentazione di un grafo

Un grafo può essere rappresentato in due modi:

* La matrice di adiacenza n\*n in cui si segnano le coppie di vertici connesse tramite un arco ( e il relativo peso se il grafo è pesato);
* L’array di liste di adiacenza in cui per ogni nodo (corrispondente all'indice dell’array e quindi a una lista) vengono memorizzati tutti i suoi vicini.

NOTA: non esiste una rappresentazione migliore dell’altra, dato che la prima è più complessa in spazio mentre l’altra lo è in tempo.

## Componenti connesse

L’insieme dei vertici adiacenti a un nodo forma i cosiddetti vicini.

Una componente connesse è un sottografo del grafo originale in cui ogni nodo può essere raggiunto da tutti gli altri, l’insieme dei vertici e degli archi del grafo sono sottoinsiemi di quelli del grafo stesso. Un grafo non orientato è detto connesso quando formato da un’unica componente connessa (per essere tale ogni vertice deve essere connesso con almeno un arco), in tutti gli altri casi è detto sparso. Un grafo è completo quando tutti i vertici raggiungono tutti gli altri.

## Cammino

Un cammino è una sequenza ordinata di vertici che viene percorsa a partire da un nodo con l’obiettivo di arrivare a un nodo d’arrivo, esso contiene tutti i vertici attraversati e i relativi archi percorsi, la lunghezza è data dal numero totale di archi che connettono i vertici della sequenza. Un cammino può essere:

* Semplice: la sequenza di vertici è formata da nodi distinti;
* Ciclo (detto anche cammino chiuso): particolare cammino in cui si ritorna a un nodo già visitato passando per un arco differente da quello iniziale;
* Minimo: percorso in cui il numero degli archi attraversati è minimo;
* Minimo pesato: percorso in cui la somma dei pesi degli archi è minima;

Un nodo è detto raggiungibile da un’ulteriore nodo se esiste un cammino che li connette.

# Visite di grafi

Una visita di un grafo permette di esaminare i nodi e gli archi in modo sistematico e la ricerca di particolari informazioni su di esso, essa può essere:

* in ampiezza (BFS): vengono prima esaminati tutti i vicini di un dato nodo e in seguito si passa ai successivi;
* in profondità (DFS): vengono prima scoperti tutti i nodi e in seguito si visitano “risalendo”.

La visita di un grafo ha come obiettivo quello di scoprire e visitare tutti i nodi appartenenti a esso, di seguito vi è un algoritmo di vista generica:

| genericVisit()  S=vuoto  A=vuoto  finchè è possibile  scelgo un arco (u,v)  if(u scoperto e v non scoperto)  aggiungo v a S  aggiungo (u,v) a A |
| --- |

La visita termina quando finisce il numero di iterazioni, quindi se il grafo è connesso, tutti i nodi vengono visitati, se il grafo è orientato, tutti i nodi raggiungibili dalla sorgente vengono visitati. L’insieme di nodi e archi percorsi forma il cosiddetto albero di visita, un insieme di nodi connessi e senza cicli dimostrabile nel seguente modo:

### Dimostrazione

BASE: |S|=1 → (S,A) è un albero

IPOTESI: |S(k)|=k → (S(k),A(k)) è un albero

TESI: |S(k+1)|=k+1 → (S(k+1),A(k+1) è un albero

per ipotesi induttiva → S(k+1)=S(k) U {v} e A(k+1)=A(k) U {(u,v)}

In ogni passo induttivo vengono controllati cicli e connessioni.

I nodi di frontiera sono vertici già visitati con almeno un arco uscente verso un nodo non visitato. L’algoritmo di visita generica deve visitare tutti i nodi ma non tutti gli archi, di conseguenza la sua complessità è O(n+m) in cui n è il numero di nodi visitati e m il numero di archi percorsi.

## Visita in ampiezza

La visita in ampiezza esamina i nodi estraendoli da una coda e inserendo in essa tutti i suoi vicini non scoperti.Osservando il contenuto della coda dopo l’inserimento dei vicini e l’albero di visita ottenuto, è possibile notare che i nodi inseriti sono al livello successivo rispetto al “padre” nell’albero. La complessità dell’algoritmo è O(n+m).

| BfsVisit(int source)  S=S U {source}  q← source  finchè q non vuoto  node=estraggo da q  per ogni vicino di node  if(vicino non scoperto)  q← vicino  S=S U {vicino} |
| --- |

l’albero di visita ottenuto con la visita in ampiezza è l’albero di cammini minimi a partire dalla sorgente. Ogni nodo dell’albero è a livello k se e solo se la distanza dalla radice è k, per dimostrarlo si utilizza il teorema di sottostruttura ottima: il cammino tra un nodo e un’altro è minimo, allora lo sono anche tutti i sottocammini.

### Dimostrazione

Lemma--> d\_grafo(source,v) <= liv\_albero(v)

dist(source,v)=k → liv(v)=k

BASE: dist(source,v)=0 → source=v → liv(v)=liv(source)=0

IPOTESI: dist(source,v)=h → liv(v)=h

TESI: dist(source,v)=h+1 → liv(v)=h+1

source---nodo1---nodo2---u---v---w

Visto che c’è per forza un nodo u prima di v, allora utilizziamo l’ipotesi induttiva per dire che:

dist(source,u)=h ,

quindi questo cammino è minimo sennò per la sottostruttura ottima ci sarebbe un cammino più breve, quindi per ipotesi induttiva u è a livello h.

Supponiamo esiste un nodo w più in basso di v.

Utilizziamo ora l’algoritmo BFS e vediamo che u esce dalla coda prima di v perchè è ad un livello inferiore quindi v può essere figlio solo di u e non di w che è ad un livello più basso ed esce dopo dalla coda.

Quindi la tesi è verificata.

BASE: dist(source,v)=0 ← liv(v)=0

di conseguenza: source=v

IPOTESI: dist(source,v)=k ← liv(v)=k

TESI: dist(source,v)=k+1 ← liv(v)=k+1

source---t---w---v

dist(source,v)<=k+1

supponiamo che dist(source,v)<h+1:

si ha che liv(v)=dist(source,v)<h+1

dato che dist(source,w)=liv(v)-1 → liv(v)-1=h+1

La distanza non può essere minore di h+1 perchè il nodo sarebbe già uscito dalla coda.

## Visita in profondità

La visita in profondità esamina i nodi di un grafo partendo dall’ultimo vertice incontrato per poi “risalire” visitando tutti gli altri. Essa richiama ricorsivamente ogni nodo ( la cui visita, però, non è ancora terminata) fino a quando tutti i suoi vicini non sono stati scoperti. La visita di un nodo termina quando tutti i suoi vicini discendenti terminano, questa proprietà è ben visibile nell'albero di visita. Grazie alla sua implementazione, la visita in profondità permette di rilevare e ricostruire cicli all'interno del grafo. La complessità dell’algoritmo è O(n+m), anche s in realtà è O(n\*m) dato che ogni nodo scorre tutti i suoi archi ogni volta. Con la matrice di adiacenza si raggiunge una complessità di O(n), tuttavia si preferisce la lista di incidenza perchè le complessità delle prime due implementazioni si equivalgono nei grafi densi.

| DfsVisit(int source)  S=S U {source}  per ogni vicino di source  if(vicino non scoperto)  DfsVisit(vicino) |
| --- |

## Ricostruzione di cammini minimi e cicli

La ricostruzione di cammini minimi e cicli è possibile attraverso la visita in ampiezza e in profondità di un grafo, essi utilizzano un array che memorizza il padre di ogni nodo durante la visita per poi risalirlo una volta conclusa, di seguito vi sono i rispettivi algoritmi:

| BuildPath(int source, int dest)  queue ← source  S=S U {source}  father[source]=-1  while(queue non vuota)  node ← estrai da queue  if(node non scoperto)  S=S U {node}  per ogni vicino di node  queue ← vicino  father[vicino]=node  int node=dest  while(node!=-1)  path ← node  node=father[node] | BuildCycle(int source)  S=S U {source}  per ogni vicino di source  if(vicino non scoperto)  father[vicino]=source  BuildCycle(vicino)  else(vicino non terminato)  int node=source  while(node!=vicino)  cycle← node  node=father[node] |
| --- | --- |

## Visite nei grafi orientati

Applicando gli algoritmi di visita citati in precedenza su un grafo orientato, si scopre che alcuni archi non vengono considerati dalla visita, essi possono essere classificati nel seguente modo:

* archi trasversali: archi che connettono due nodi sullo stesso livello nell’albero di visita;
* archi all’indietro: archi che connettono un nodo a un suo antenato nell'albero di visita, la sua presenza determina se in un grafo sono presenti dei cicli;
* archi in avanti: archi che connettono un nodo a un suo discendente nell’albero di visita.

NOTA: questi tipi di archi sono presenti anche nei grafi non orientati, con la differenza che nelle visite in ampiezza sono presenti solo gli archi trasversali dato che gli altri vengono considerati dall’albero di visita. Nella visita in profondità, invece, sono presenti tutti i tipi di archi. La presenza di questi archi nei grafi non orientati permette la rilevazione di cicli.

## Ordine Topologico

L’ordine topologico è un tipo di ordinamento del grafo in cui tutti gli archi uscenti partono da un nodo di sinistra e arrivano a uno di destra, questo è possibile solo se il grafo è DAG, ovvero che è orientato e aciclico. Il calcolo dell’ordine topologico è possibile attraverso la visita in profondità, in cui ogni nodo termina la sua visita quando terminano quelle dei suoi discendenti in assenza di cicli. Quando un nodo non presenta archi uscenti, viene in automatico messo alla fine dell’ordine.

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

## Componenti fortemente connesse

Una componente fortemente connessa (SCC) è un sottografo massimale di un grafo orientato in cui, per ogni coppia di nodi, esiste un cammino orientato. Ogni nodo può raggiungere ed è raggiungibile da altri vertici della stessa SCC attraverso un cammino fortemente connesso, tuttavia nessun cammino può abbandonare la propria componente, dato che gli ulteriori nodi percorsi sarebbero raggiungibili e quindi appartenenti a essa. Un grafo orientato può avere diverse SCC connesse tra loro, da esse è possibile ricavare il grafo delle SCC, una rappresentazione del grafo stesso in cui le SCC vengono utilizzate come nodi, esso è DAG perchè i cammini non possono uscire dalla propria componente, di conseguenza è possibile calcolarne l’ordine topologico.

### Proprietà delle SCC

Date due componenti connesse c1 e c2 collegate nel seguente modo: c1→ c2, la visita in profondità del primo nodo di di c1 termina dopo la visita di tutti i nodi di c2 perché essi vengono scoperti prima di questi ultimi. Grazie a questa proprietà, i nodi di un grafo che non hanno archi entranti sono quelli che terminano per ultimi la visita in ogni caso. Questa regola definisce un ordine di terminazione dei nodi nella visita in profondità che, guardato dal fondo, corrisponde all’ordine topologico del grafo delle SCC, esso è detto ordine di fine visita.

### Grafo trasposto

Il grafo trasposto è una versione del grafo orientato in cui tutti gli archi sono invertiti, le componenti fortemente connesse non cambiano e l’ordine di fine visita è inverso. Visitando il grafo trasposto utilizzando l’ordine di fine visita, si può notare che l’ordine di fine visita equivale all’ordine topologico del grafo trasposto delle SCC e, quando si visita un nodo di una determinata SCC, esso non può uscire da essa.

### Algoritmo di Kosarasu

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

* Calcolo dell’ordine di fine visita visitando il grafo in profondità;
* Calcolo del grafo trasposto;
* Visita del grafo trasposto seguendo l’ordine di fine visita, essa può essere sia in ampiezza che in profondità;

#### Dimostrazione di correttezzak

Date due pile P e R in cui inserire rispettivamente i nodi “parziali” e quelli rappresentanti e il +-primo nodo visitato v, si considera un grafo delle SCC, per esso vale la seguente proprietà: dopo il termine della visita del primo nodo di una SCC, sulle pile R e P non ci saranno più i relativi nodi.

**Lemma 1**: Se un grafo ha una sola SCC, l’algoritmo è corretto.

**Dimostrazione**: Inizialmente le pile R e P sono vuote:

* tutti i nodi termineranno la visita prima di v per via delle proprietà della visita: vengono quindi messi nella pila P dopo il primo nodo.
* Da ogni nodo u della SCC è possibile tornare a v per via delle proprietà delle SCC, essi avranno ordine(u)>ordine(v): quindi ogni nodo diverso da v verrà tolto dalla pila R prima o poi.
* Dal momento che non esistono nodi di altre SCC che possono raggiungere v, dopo esso possono esserci solamente nodi della SCC a cui appartiene all’interno delle pile.
* Al termine della visita in profondità di un nodo u in cima alla pila R, esso viene tolto e dalla pila P e tutti i nodi fino a quest’ultimo verranno tolti e assegnati alla rispettiva SCC.

**Lemma 2**: Se una SCC non ha archi uscenti, l’algoritmo è corretto per essa.

**Dimostrazione**: I primi tre punti valgono anche per questo lemma, in più i nodi precedenti a v nelle pile sono stati visitati prima di esso e di conseguenza non appartengono alla stessa SCC. Dato che la SCC in questione non ha archi uscenti, non esiste nessun cammino da v coi precedenti u nelle pile dato che ordine(u)<ordine(v) di conseguenza v non verrà mai tolto fino a quando non termina la sua visita. Alla termine della visita di v, nella pila P ci sono tutti e solo i nodi della stessa SCC e in cima c’è proprio v stesso (lo stesso vale per la pila R), quindi l’algoritmo è corretto anche in questo caso.

**Lemma 3**: Date due SCC C e C’ connesse tramite un cammino (C→ C’), il primo nodo della prima SCC termina la visita dopo tutti i nodi della seconda SCC.

**Dimostrazione**: Dato che l’ordine di esecuzione dipende da un ciclo esterno alla visita stessa, se comincia prima C’, essa non avrà modo di raggiungere i nodi appartenenti a C, altrimenti vi è un nodo v2 di C avente un arco uscente che incide su un nodo w1 di C’, da quest’ultimo la visita continuerà finché non verranno scoperti tutti i nodi raggiungibili. Al termine di w1, tutti i nodi di C’ hanno già terminato la visita mentre quelli di C, inclusi v2 e v, non l’hanno ancora conclusa.

### Dimostrazione della proprietà

Base: Data una SCC C, essa è l’ultima nell’ordinamento topologico e quindi non ha archi uscenti.

Ipotesi: se tutte le altre SCC hanno una posizione maggiore di k nell’ordine topologico, allora C citata è k-esima nell’ordine. Questo è dimostrabile dato che, seC non ha archi uscenti, la tesi segue il lemma 2, altrimenti la visita delle successive SCC C’, basandosi sul lemma 3, tali per cui esiste un cammino tra C e C’ termina la visita di v, tutti gli altri nodi di C’ non saranno di conseguenza più presenti all’interno delle due pile. Tutte le SCC C’ che seguono C nell’ordine topologico non possono essere scoperte attraverso la visita di v, di consenguenza verranno aggiunti prima di v nelle due pile.

## Algoritmo di Dijkstra

L'algoritmo di Dijkstra è un algoritmo che, dato un grafo pesato con pesi non negativi, permette di calcolare tutti i cammini minimi a partire da una sorgente. Inizialmente l’algoritmo mette in un insieme tutti gli archi che partono dalla sorgente e, finchè è possibile, estrae l’arco con la distanza minima che porta a un vertice non ancora scoperto, aggiungendo di conseguenza i suoi archi. Ogni arco appartenente all’insieme ha associato un numero, esso è la distanza del nodo a cui porta a partire dalla sorgente.

| Dijkstra(int source)  S={source}  dist[source]=0  finchè è possibile  scelgo un arco (u,v) con u scoperto e v non scoperto  tale che dist[u]+peso(u,v) sia minimo  S=S U {v}  dist[v]=dist[u]+peso(u,v) |
| --- |

### Dimostrazione

BASE: |S|=1 → dist[source]=d(source,source)=0

IPOTESI: |S|=k → dist[v]=d(source,v)

TESI: |S|=k+1 → per ogni v in S, dist[v]=d(source,v)

supponiamo che il cammino passi per un arco (x,y)

distance=source →(x,y) → v

len(distance)=len(source→ x)+peso(x,y)+len(y,v)

Con pesi degli archi maggiori o uguali a zero, si ha che:

len(distance)>=len(source→ x)+peso(x,y)

len(distance)>=d(source,x)+peso(x,y)

d(source,x)+peso(x,y)>=d(source,u)+peso(u,v)

dato che d(sorg,u)+peso(u,v)=len(other),

si ha che len(distance)>=len(other)

La complessità di questo algoritmo è O(n\*m), dato il costo esponenziale in termini di tempo, è possibile implementare una soluzione migliore utilizzando le code con priorità (un heap precisamente).

| Dijkstra(int source)  S={source}  dist[source]=0  per ogni arco (source,v)  heap ← ((source,v), dist[source]+peso(source,v))  while(heap non vuoto)  (u,v) ← estraggo da heap  if(v non scoperto)  S=S U {v}  dist[v]=dist[u]+peso(u,v)  per ogni nodo (v,w) tale che w non scoperto  heap ← ((v,w),dist[v]+peso(v,w)) |
| --- |

La complessità con questa implementazione diminuisce a O(mlogn). Questo algoritmo utilizza la cosiddetta tecnica greedy, ogni scelta calcolata non viene mai ripetuta per cambiarla.

# Sequenziamento dei processi

Consideriamo n processi p con lunghezze positive l, se i processi vengono eseguiti nell’ordine p1, p2,...,pn, il tempo d’attesa di un processo pk è:

Tk=somma(l1...lk)

ovvero il tempo d’attesa fino al completamento di pk.

Il tempo d’attesa medio si calcola allo stesso modo:

A=somma(T1...Tn)/n

L’obiettivo è minimizzare il tempo d’attesa medio.

### Soluzione greedy

Esempio: dati due processi p1 e p2 con lunghezza 4 e 7, il tempo d’attesa medio è:

A=(4+11)/2=7,5

se invertiamo l’ordine si ha che

A’=(7+11)/2=9

dato che A>A’, è opportuno ordinare i processi in base alla lunghezza prima di sceglierli

La soluzione greedy sceglie il processo con lunghezza minima non ancora eseguito, rendendola in questo modo ottimale.

#### Dimostrazione

Supponiamo che i processi abbiano lunghezze differenti e che per assurdo la soluzione greedy g non sia ottimale, quindi esiste una soluzione ottimale s il cui tempo d’attesa medio è minore di quello di g. La soluzione greedy ordina i processi in ordine crescente, quindi sarà nel seguente modo:

g={p1,p2,p3,...,pn}

La soluzione s è diversa da g perché vi sono due processi ph e pk in cui la lunghezza del primo e maggiore di quella del secondo:

s={p1,p2,...,ph,...,pk,...,pn}

Scambiando ph e pk otteniamo s’, una soluzione uguale a s se non per i due processi scambiati:

s’={p1,p2,...,pk,...,ph,...,pn}

I tempi d’attesa medi di s e s’ saranno di conseguenza:

As=somma(T1,...Th,...Tk,...Tn)/n

As’=somma(T1,...T’k,...T’h,...Tn)/n

in cui T’h e T’k sono i tempi di attesa dei processi ph e pk in s’, quindi:

As-As’=(Th+Tk-T’h-T’k)/n=(lh-lk)/n>0

La formula sopra contraddice l’ipotesi lh>lk, di conseguenza la soluzione greedy è quella ottimale.

E’ possibile dimostrare che la soluzione greedy è ottimale anche nel caso in cui i processi abbiano lunghezze uguali, ma non per assurdo. Supponiamo l’esistenza di una soluzione ottimale s diversa da quella greedy, se si scambiano due processi ph e pk mettendoli in un ordine diverso da quello greedy, si ottiene una soluzione s’ non peggiore della precedente ma più simile a greedy. Ripetendo questi scambi un numero sufficiente di volte, si ottiene la soluzione greedy, è comunque possibile notare che, con lunghezze non tutte distinte, vi possono essere più soluzioni ottimali.

### Processi con pesi

Consideriamo ora il caso in cui gli n processi, oltre ad avere delle lunghezza positive, abbiano anche dei pesi w, è quindi possibile definire il tempo d’attesa pesato:

W=somma(w1\*T1,w2\*T2,...,wn\*Tn)

L’obiettivo è minimizzare il tempo d’attesa pesato.

Esistono due casi particolari di questo problema:

* Nel caso in cui tutte le lunghezza siano uguali, si ordinano i processi in ordine decrescente in base al peso;
* Nel caso in cui i pesi siano tutti uguali, si ordinano i processi in ordine crescente in base alla lunghezza.

Nel caso generale è possibile combinare i primi due: si calcola un punteggio che cresce all’aumentare del peso e al diminuire della lunghezza e si ordinano i processi in base a quello. I processi possono quindi essere ordinati in vari modi, ad esempio:

* Ordinarli in modo decrescente in base al rapporto peso/lunghezza;
* Ordinarli in modo crescente in base al rapporto lunghezza/peso;

#### Dimostrazione

Supponiamo che la soluzione greedy g di questo problema non sia ottimale, di conseguenza esiste una soluzione s ottimale. La soluzione greedy ordina i processi in modo decrescente in base al rapporto peso/lunghezza, quindi:

g={p1,p2,...,pn}

s è la soluzione ottimale ed è diversa da g, quindi vi sono due processi ph e pk in cui il rapporto del primo è minore di quello del secondo:

s={p1,p2,...,ph,...,pk,...,pn}

se scambiamo i due processi, otteniamo s’, una soluzione uguale a s ma coi due processi scambiati:

s’={p1,p2,...,pk,...ph,...pn}

I tempi d’attesa pesati delle soluzioni s e s’ saranno quindi:

Ws=somma(w1\*T1,...,wh\*Th,...,wk\*Tk,...wn\*Tn)

Ws’=somma(w1\*T1,...,wk\*T’k,...,wh\*T’h,...wn\*Tn)

In cui T’k e T’h sono i tempi d’attesa dei processi pk e ph nella soluzione s’, quindi:

Ws-Ws’=wh\*Th + wk\*Tk + wh\*T’h + wk\*T’k=wk\*lh - wh\*lk>0;

la formula sopra contraddice l’ipotesi wh\*lh<wk\*lk, di conseguenza la soluzione greedy è quella ottimale.

E’ possibile dimostrare che la soluzione greedy è ottimale anche quando vi sono processi il cui rapporto è uguale, la dimostrazione è analoga ma non per assurdo. Supponiamo l’esistenza di una soluzione s diversa da quella greedy, se scambiamo due processi ph e pk si ottiene una soluzione s’ non peggiore della precedente e più simile a quella greedy. Ripetendo questi scambi un numero sufficiente di volte, si ottiene la soluzione greedy, è comunque possibile notare che, nel caso in cui i rapporto dei processi non siano tutti distinti, vi possono essere più soluzioni ottimali.

# Minimo albero ricoprente

Un albero ricoprente è un sottografo di un grafo che comprende tutti i nodi ed è senza cicli, un grafo può avere molteplici alberi ricoprenti al suo interno e, se esso ha dei pesi sugli archi, è possibile calcolare il costo di ognuno di essi.

costo(A)=somma(peso(u,v)...)

Il costo di un albero ricoprente è quindi la somma dei pesi degli archi di cui è composto, possiamo quindi dire che un albero ricoprente è minimo quando il suo costo è minimo. Nel caso in cui i pesi degli archi in un grafo siano tutti distinti, esso ammette solamente un minimo albero ricoprente, altrimenti ve ne possono essere molteplici.

### Dimostrazione

Supponiamo che, per assurdo, un grafo ammetta due minimi alberi ricoprenti differenti, ovvero che esiste almeno un arco presente in un albero ma non nell’altro e viceversa. Prendiamo ora l’arco di peso minimo presente in un albero ma non nell’altro e lo inseriamo in quest’ultimo, esso genererà un ciclo. Considerando questo ciclo, dal momento che nel primo albero nno se sono presenti, allora esiste un arco che non appartiene a quest’ultimo, esso avrà un peso maggiore di quello inserito precedentemente dato che quest’ultimo è minimo. Togliendo quest’arco si andrà a creare un albero ricoprente diverso dal secondo ma con peso minore, contraddicendo l’ipotesi che il secondo albero ricoprente sia minimo.

Nel caso in cui esistessero archi con lo stesso peso nel grafo, non si potrebbe affermare che l’arco da togliere sia strettamente maggiore di quello aggiunto, esso infatti può essere maggiore o uguale, di conseguenza un grafo può ammettere più minimi alberi ricoprenti se i pesi non sono tutti distinti.

### Teorema del taglio

Per costruire un minimo albero ricoprente si utilizza la tecnica greedy prendendo gli archi un alla volta, essi devono essere scelti secondo un determinato criterio, quindi ci serve la definizione di taglio: un taglio di un grafo è una partizione di vertici in due insiemi, quello dei nodi scoperti e il suo complementare (ovvero quelli ancora da scoprire), un arco (u,v) attraversa il taglio quando u è scoperto e v non lo è. A questo punto si può definire la cosiddetta regola del taglio: di tutti gli archi del grafo che attraversano il taglio, viene scelto quello con peso minimo. Un minimo albero ricoprente deve contenere almeno un arco che attraversa il taglio perchè altrimenti non comprenderebbe tutti i vertici.

#### Dimostrazione

Dato un grafo non orientato, connesso e pesato, ogni passo della tecnica greedy mantiene la seguente invariante: Esiste sempre un minimo albero ricoprente che comprende tutti gli archi appartenenti alla soluzione e non contiene tutti gli altri. Inizialmente l’invariante verrà verificata da qualsiasi albero ricoprente dato che non sono ancora presenti archi nella soluzione. Nel passo induttivo viene utilizzata la regola del taglio: dato un taglio, un arco e appartenente alla soluzione e un minimo albero ricoprente T che ha verificato l’invariante prima che l’arco e entrasse nella soluzione, vi possono essere due casi:

* Se l’arco e è nell’albero T, allora l’invariante è verificata anche dopo che e é entrato nella soluzione;
* se invece non lo è, allora esiste un cammino (u,v) in T tra gli estremi di e, ovvero u e v.

Grazie all’invariante nessun arco di T è escluso dalla soluzione e quelli che attraversano il taglio non appartengono (ancora) a essa. Dato che l’arco e è stato inserito nella soluzione per via della regola del taglio, allora il suo peso è minore o uguale a qualsiasi altro arco f che attraversa il taglio. In conclusione f non appartiene alla soluzione e di conseguenza l’albero T’=(T-e’) U e verifica l’invariante dopo l’applicazione della regola del taglio.

La tecnica greedy termina una volta che tutti gli archi sono stati inseriti/esclusi dalla soluzione. Nel caso in cui la tecnica greedy non riesca ad applicare la regola del taglioe che esista un arco e non colorato, a causa dell’invariante, tutti gli archi della soluzione formano una foresta. Se gli estremi dell’arco incidono su due alberi differenti, si può applicare la regola del taglio.

### Algoritmo di Prim

Un metodo per ricavare il minimo albero ricoprente è l’algoritmo di Prim, un algoritmo simile a quello di Dijkstra con la differenza che il primo tiene conto solo del peso dell’arco considerato e non la distanza dalla sorgente.

| PrimVisit(int source)  S={source}  A=vuoto  per ogni arco (source,v)  heap ← ((source,v),peso(source,v))  while(heap non vuoto)  (u,v) ← estraggo da heap  if(v non scoperto)  S=S U {v}  A=A U {(u,v)}  per ogni arco (v,w)  if(w non scoperto)  heap ← ((v,w),peso(v,w)) |
| --- |

### Algoritmo di Kruskal

Un’altro algoritmo che permette il calcolo del minimo albero ricoprente è Kruskal: Inizialmente crea una foresta in cui gli alberi sono formati da un solo vertice e ogni volta considera un arco a cui applica il seguente passo: se l’arco ha entrambi gli estremi sullo stesso arco, esso viene escluso, altrimenti viene incluso nella soluzione. Una volta esaminati tutti gli archi, l’algoritmo termina e restituisce l’albero.

#### Dimostrazione

In base al teorema del taglio è sufficiente dimostrare che l’algoritmo di Kruskal utilizza la tecnica greedy. Gli archi vengono esaminati in ordine non decrescente, quando Kruskal esamina un arco e, esso non è ancora stato inserito/escluso dalla soluzione mentre tutti quelli analizzati prima lo sono perchè hanno un peso minore di e. Se l’arco e ha entrambi gli estremi su un albero all’interno del grafo, esso viene escluso, viene aggiunto alla soluzione se invece connette due alberi differenti, di conseguenza e è il primo albero che attraversa il taglio.

| Kruskal()  finchè è possibile  (u,v) ← arco con peso minimo disponibile  if(non forma ciclo) A=A U {(u,v)}  else scarto(u,v) |
| --- |

Per verificare che gli estremi dell’arco scelto appartengano allo stesso albero o meno, l’algoritmo deve effettuare una verifica che richiede un tempo O(n) nel caso peggiore e, di conseguenza, il tempo totale è O(nm) dato che viene effettuata su ogni arco.

### Implementazione con union-find

Per effettuare la verifica dell’arco in modo più efficiente, si utilizza una struttura dati detta union-find, essa considera gli alberi del grafo come insiemi disgiunti su cui effettuare le operazioni di unione di due alberi (union) o se gli estremi di un arco appartengono o meno a uno stesso albero (find). In questo modo l’algoritmo di Kruskal calcola il minimo albero ricoprente in tempo O(mlogn).

#### Dimostrazione

Si ordinano gli archi in modo non decrescente in base al peso utilizzando un apposito algoritmo, una volta ordinati, un minimo albero ricoprente può essere ottenuto con n makeSet(), 2m find e n-1 union, il tempo totale sarà quindi O(mlogn) nel caso peggiore.

## Union-find

Dato un’insieme di oggetti U, si vuole implementare una struttura che permette di raggrupparli in sottoinsiemi disgiunti a partire da quelli unitari, ognuno è denotato da un elemento detto rappresentante. Le strutture vengono create attraverso l’operazione makeset(), la quella crea n sottoinsiemi unitare (quindi la complessità è O(n)), su ogni sottoinsieme vengono applicate le seguenti operazioni:

* Union(a,b) sostituisce gli insiemi rappresentati da a e b con la loro unione;
* Find(e) restituisce il rappresentante dell’insieme a cui e appartiene.

La complessità di queste ultime operazione dipende dall’implementazione.

### Quick Find

E’ possibile rappresentare la struttura utilizzando un’array di lunghezza n in cui ogni cella memorizza il rappresentante del proprio insieme, ad esempio: se l’insieme U = {0, 1, 2, 3} e vi sono all’interno i sottoinsiemi {0, 1, 3} e {2}, se 3 è rappresentante del primo insieme allora l’array sarà [3, 3, 2, 3]. L’implementazione può essere vista logicamente come un albero in cui ogni elemento punta al suo rappresentante, esso avrà altezza massima 1 dato che è necessario attraversare un solo arco per ottenere il rappresentante. Di conseguenza, l’operazione find() avrà complessità O(1) appunto perché effettua un unico accesso all’array. L’operazione di union(a,b), invece, deve rinominare tutte le celle di b assegnandogli a come rappresentanto (la scelta opposta è comunque legittima), ad esempio: se si effettua una union(2,3) sull’array [3, 3, 2, 3], si otterrà come risultato [2, 2, 2, 2]. Di conseguenza, l’operazione union(a,b) ha complessità O(n).

### Quick Union

Una strategia differente è quella di rendere più efficiente l’operazione union(a,b), essa è possibile aggiornando solamente la cella di b, ponendolo uguale ad a, lasciando inalterati gli altri elementi. In questo modo l’operazione union(a,b) avrà complessità costante. L’operazione find(e), invece, ha un costo che dipende dalla profondità dell’albero logico (che non è più di altezza 1), dato che deve risalire l’albero a partire da e fino ad arrivare alla radice. Infatti, se le unioni vengono fatte in ordine crescente, si avrà un albero logico con un unico ramo lungo n-1 e, di conseguenza, la find(e) del nodo a livello più alto avrà complessità O(n).

### Union By Rank

E’ possibile ottenere un risultato intermedio realizzando una union(a,b) costante e una find(e) logaritmica al costo di mantenere un’ulteriore array grande n in cui mantenere la profondità dell’albero logico. A partire da due alberi A e B, si appende B come sottoalbero di A in modo la radice del primo punti a quella del secondo, tutti gli altri nodi rimangono invariati. La radice di A diventa quindi la radice dell’albero risultante C, quindi: Se l’altezza di A è maggiore di quella di B, l’altezza di C è uguale a quella di A, altrimenti è uguale a quella di B più 1.

#### Dimostrazione

I rami di C sono:

* Tutti i rami di A invariati;
* Tutti i rami di B più un’ulteriore arco che connette le due radici (questo spiega perché l’altezza è incrementata di 1).

Di conseguenza, il ramo più lungo di C il massimo tra le altezze dei due sottoalberi, infatti se il k>h, allora h+1<=k e quindi il ramo più lungo è lungo k, altrimenti è lungo h+1, confermando la tesi.

Se si appende l’albero meno profondo a quello più profondo, infatti, si mantiene sotto controllo la profondità dell’insieme, aumentando solo quando le altezze sono uguali. E’ sufficiente mantenere un array grande in cui ogni cella memorizza la profondità del proprio albero, la sua inizializzazione avviene tramite il makeset(), il quale mantiene complessità O(n). La profondità dell’albero prende il nome di rank, da cui deriva il nome dell’implementazione. L’implementazione della union(a,b) deve determinare il rank dell’albero A e quello di B, di conseguenza si procede nel seguente modo:

* se rank(A)>rank(B), il rappresentante dell’insieme è a e quindi rank(A U B)=rank(A), in più non è necessario aggiornare alcun valore dato che il rappresentante è a;
* Lo stesso discorso vale anche per rank(A)<rank(B), l’unica differenza è che il rappresentante è b;
* Se rank(A)=rank(B), il rappresentante varia a seconda dell’implementazione e rank(A U B)=rank(A)+1=rank(B)+1 , in più bisogna aggiornare la cella corrispondente a quella del rappresentante;

La complessità di union(a,b) rimane costante, dato che deve leggere due valori, aggiornare il puntatore e al massimo una profondità, per quanto riguarda la find(e), invece, bisogna determinare un limite superiore degli alberi.

#### Dimostrazione

La profondità dell’albero è legata al numero di elementi dell’insieme dalla relazione:

Il caso base consiste in una collezione di insiemi unitari, ognuno con cardinalità 1 e profondità 0, confermando quindi la relazione. Dopo k union(a,b), la relazione è valida per tutti gli insiemi della collezione, essa rimane tale anche dopo la k+1esima union(a,b).

Considerando la k+1esima Union, dato che per ipotesi induttiva e , è sufficiente dimostrare se la relazione é ancora valida per l’insieme unione (dato che gli altri rimangono invariati), vi sono te casi possibili:

* Se rank(A)>rank(B), |A U B|=|A|+|B| perchè A e B sono disgiunti,in più |A|+|B|>=”^rank(A) per ipotesi induttiva. Dato che il rappresentante è a, si può dire che e quindi la relazione vale anche è è per A U B.
* Il passo induttivo si dimostra come sopra con rank(A)<rank(B), prendendo b come rappresentante;
* se rank(A)=rank(B) significa che |A U B|=|A|+|B|>=2^rank(A) + 2^rank(B)=2\*2^rank(A)=2^(rank(A)+1), quindi in base al punto 2 riguardante la profondità dell’albero, quando rank(A)=rank(B), risulta che rank(A u B)=rank(A)+1, confermando che la relazione vale anche per A U B per questo terzo caso.

E’ anche possibile dimostrare che la complessità della find(e) è O(logn) in cui n è la dimensione della collezione U, infatti per ogni insieme si ha che , dunque rank(A)<=log2|A|. Dato che |A|<n per ogni albero, tutti gli insiemi hanno sempre rank al massimo log2n.

## Clustering

Dati n oggetti e le distanze tra ogni coppia, determinare il raggruppamento di oggetti in h insiemi che abbia spaziamento massimo, esso è la distanza minima tra oggetti non appartenenti allo stesso cluster. Gli oggetti e le relative distanze possono essere rappresentati come un grafo non orientato, pesato e completo in cui i primi rappresentano i vertici mentre i secondi gli archi. Questo problema è un’applicazione dell’algoritmo di Kruskal in cui la terminazione avviene quando vengono raggiunti h insiemi. Utilizzando la tecnica greedy, lo spaziamento è dato dal peso dell’arco che Kruskal avrebbe selezionato dopo l’interruzione, l’arco infatti è di peso minimo e unisce due cluster differenti.

### Dimostrazione

Si considera un clustering K ottenuto con la tecnica greedy e un’altro A, se essi sono distinti,allora esistono due oggetti i e j che in K sono nello stesso cluster ma in A sono in due cluster differenti. Quindi se i e j sono nello stesso cluster di K, allora esiste un cammino tra i e j i cui archi sono stati aggiunti da Kruskal durante le interazioni, essi avranno peso minore a uguale a quello dello spaziamento di K:

w(u,v)<sp(K) per ogni vertice u e v tra i e j.

Dato che i e j sono in due cluster differenti di A, allora esiste un arco (u,v) nel cammino tra i e j i cui estremi si trovano in cluster differenti, il suo peso è la distanza tra due oggetti in due cluster diversi ed è maggiore o uguale allo spazioamento:

w(u,v)>=sp(A)

Dato che u e v sono due nodi del cammino tra i e j, le due relazioni implicano che sp(A)<=sp(K), cioè che lo spaziamento di K non è peggiore di qualunque altro clustering diverso da quello greedy.

## Problema dello zaino frazionario

Il problema dello zaino è un problema in cui, dati n materiali, ciascuno con un volume e un valore, e una capacità C, si deve riempire quest’ultima ottenendo come valore totale il massimo possibile. Per ogni materiale si definisce una dose, una frazione compresa tra 0 e 1 che indica la quantità di materiale da mettere nello zaino (questo limite è detto rilassamento del problema). Questo problema è risolvibile attraverso una soluzione greedy in cui si prende un’intera dose del materiale il cui rapporto valore/volume è il più alto possibile, di conseguenza si può migliorare l’algoritmo ordinando i materiali in modo decrescente secondo questo rapporto. L’algoritmo prenderà un’intera dose di un materiale fino a quando la capacità lo permette, se non vi è più spazio di un’intera dose per un dato materiale, si riempie lo spazio restante con una frazione di esso, l’algoritmo termina fino a quando lo zaino non è totalmente pieno. La complessità è data in gran parte dall’algoritmo d’ordinamento utilizzato, di conseguente è O(nlogn).

| FracKnapsac(int[] mat, double capacity)  mat.sort()  per ogni i: dose[i]=0  space=capacity  int j=1  while(j<=n and space>0)  if(mat[j].volume<=space)  dose[j]=1  space-=mat[j].volume  tot+=mat[j].value  else  dose[j]=space/mat[j].volume  space=0  tot+=(dose[j]\*mat[j].value)  return tot |
| --- |

### Dimostrazione

Se la somma dei pesi di tutti i materiale è minore o uguale alla capacità dello zaino, essi verrebbero tutti presi per intero e di conseguenza si ottiene una soluzione ottimale. Se così non fosse, vi è un materiale j che non verrà preso per intero, di conseguenza si riempirà la parte restante dello zaino con esso:

dose[j]=space/mat[j].volume

Quindi dose(i)\*volume(i)+space=capacity

Supponiamo che per assurdo la soluzione S trovata non sia ottimale, di conseguenza esiste una soluzione S’ tale che il suo valore totale è maggiore di S:

S’=somma(dose’(i)\*value(i)

S=somma(dose(i)\*value(i))

S’>S

Anche per dose’ deve risultare che dose’(i)\*volume(i)+space=capacity e, dato che gli elementi sono ordinati in modo decrescente, vi sono due materiali j e k tale che dose’(j)<1 e dose’(k)>0. Se i materiale j e k non esistessero, allora j=k e quindi dose’=dose, di conseguenza esiste e>0 tale che dose’(j)+e<=1 e dose(k)-e<=0. Si può quindi definire una soluzione S’’=somma(dose’’(i)\*value(i)) in cui:

* se i=j, dose’’(i)=dose’(i)+e;
* se i=k, dose’’(i)=dose’(i)-e;
* dose’’(i)=dose’(i) altrimenti.

la funzione dose’’ vale quindi e\*(value(j)-value(k)), di conseguenza non può essere ottimale.

## Zaino non frazionario

Il problema dello zaino non frazionario è un caso particolare in cui tutti gli i materiali possono essere presi solo nella loro interezza, quindi dati n oggetti con valore e volume e una capacità C, bisogna riempire lo zaino in modo che il valore totale sia il massimo possibile. Utilizzando una soluzione greedy in cui, dopo aver ordinato gli oggetti in modo decrescente secondo il rapporto valore/volume, si inseriscono gli oggetti finché è possibile.

Esempio: Dati 3 oggetti e una capacità di 10

| Oggetto | volume | valore |
| --- | --- | --- |
| 1 | 6 | 9 |
| 2 | 5 | 5 |
| 3 | 5 | 5 |

Com’è possibile notare, la soluzione ottimale è data dagli oggetti 2 e 3 in quanto il loro valore è il massimo possibile, la soluzione greedy, invece, dà come risultato 9 dato che prende solamente l’oggetto 1 e quindi non è ottimale. Il problema della soluzione greedy è che, mettendo l’oggetto 1 nello zaino, non si ha più capacità sufficiente per contenere gli altri. Per risolvere questo problema in modo efficiente, si utilizza la programmazione dinamica: di definisce una soluzione ottimale come sottoproblema generico, si suppone che essa avrà una capacità minore e un insieme di oggetti più ristretto. Quindi per i e j tali che 0<=i<=n e 0<=j<=capacity, si denota un problema P(i,j) relativo al caso in cui si ha un insieme formato da i oggetti e una capacità dello zaino pari a j, indicando con V(i,j) il valore di tale situazione. Si può quindi osservare che per ogni i e j, V(i,0)=0 e V(0,j)=0 dato che non è possibile riempire lo zaino quando non è più spazio o quando non vi sono oggetti a disposizione. Da queste osservazioni, è possibile definire il valore massimo V(i,j) per il problema p(i,j), esso può essere ottenuto a partire dalle soluzioni ottimali che utilizzano un sottoinsieme degli oggetti (da 0 a i-2), vi possono essere due differenti situazioni:

* se l’oggetto i-1 non è incluso nella soluzione, allora la soluzione di P(i,j) è ammissibile e ottimale anche per P(i-1,j) dato che V(i,j)=V(i-1,j);
* se invece l’oggetto i-1 è incluso nella soluzione, il valore sarà la somma del problema con i-1 oggetti e capacità j-peso(i-1), dato che l’oggetto è all’interno dello zaino e quindi si considera la capacità restante, più il valore dell’oggetto i-1, quindi V(i,j)=V(i-1,j-peso(i-1)+value(i-1);

La soluzione ottimale per il problema P(i,j), dato che lo zaino è un problema di massimizzazione, sarà data dal massimo tra le due soluzioni ottimali ottenute dai sottoproblemi da cui dipende: V(i,j)=max(V(i-1,j),V(i-1,j-peso(i-1))+value(i-1). Con un insieme di i oggetti e una capacità j, se non si prende l’oggetto i-1, la soluzione ottimale sarà data dagli altri i-1 oggetti, altrimenti si deve anche considerare una capacità minore dato che l’oggetto è stato inserito nello zaino. La complessità dell’algoritmo è O(ij), dove i è il numero di oggetti e j è la capacità, dato che vi sono O(ij) sottoproblemi e ognuno comporta l’accesso ai relativi sottoproblemi in tempo costante.

| Knapsac(int[] mat,double capacity)  for c=0 to capacity  val(c,0)=0  for obj=1 to n  for c=0 to capacity  if(mat[obj].volume>c)  val(c,obj)=val(c,obj-1)  else  value1=val(c-mat[obj].volume,obj-1)  value2=val(c,obj-1)  val(c,obj)=max(value1,value2) |
| --- |

### Sottostruttura ottima per lo zaino

Data una soluzione S per il problema P(i,j) in cui i è il numero di oggetti e j la capacità, si ha due differenti situazioni:

* se S non contiene i, allora è anche una soluzione ottimale per il problema P(i-1,j);
* se S contiene i, allora S’=S\{i}, una soluzione derivata da S senza considerare l’oggetto, è ammissibile e ottimale per il problema P(i-1,j-volume(i)).

#### Dimostrazione

L’ipotesi del primo punto è che, se S non contiene i, allora S è ammissibile e ottimale per P(i-1,j). Se S non fosse ottimale, allora esiste una soluzione S’ migliore di S per il problema P(i-1,j), essa però sarebbe ottimale anche per P(i,j) e questo contraddice l’ipotesi.

L’ipotesi del secondo punto è che, se S contiene i, allora S’=S/{i} è una soluzione ammissibile e ottimale per il problema P(i-1, j-volume(i)). se S’ non fosse ottimale, allora esiste una soluzione S’’ in cui migliore di S’ nel problema P(i-1, j-volume(i)), di conseguenza S’’’=S’’ U {i} sarebbe migliore di S e quindi ottimale per P(i,j), cosa che contraddice la tesi.

## Tipi e classi di problemi

I problemi finora trattati ammettono algoritmi aventi complessità polinomiale, ovvero accettabili, nella dimensione dell’input, sfortunatamente esistono anche problemi in cui o esiste un algoritmo la cui complessità richiede un tempo esponenziale oppure non vi è affatto, non permettendo l’utilizzo degli stessi su istanze reali. Uno dei problemi tuttora aperti è rispondere a una domanda: P=NP? ovvero, L’insieme dei problemi polinomiale è uguale a quello dei non polinomiali?

Un problema P si può rappresentare con la relazione P contiene I x S, in cui I è l’insieme delle istanze di ingresso mentre S quello delle soluzioni. Si può quindi immaginiamo un predicato che, dato un input i e una soluzione s, restituisce 1 se (i,s) appartiene a P e 0 altrimenti. I problemi possono essere di tre tipi:

* Decisione: problemi che richiedono una risposta binaria, quindi S={0,1}, permettono data un’istanza i di verificare certe proprietà. Essi possono essere partizionati in istanze positive ((i,1) appartiene a P) e negative ((i,0) appartiene a P);
* Ricerca: data un’istanza i, è richiesta la restituzione di una soluzione s tale che (i,s) appartiene a P;
* Ottimizzazione: data un’istanza i, si deve trovare la soluzione s migliore tra tutte le soluzioni di P, essa è valutata dai criteri specificati dal problema stesso.

I problemi di ottimizzazione possono essere ulteriormente distinti in massimizzazione e minimizzazioni, ovvero che la soluzione deve essere la più grande/piccola possibile.

Ogni problema di ottimizzazione può essere espresso in forma decisionale, la difficoltà del problema rimane invariata, in più se riusciamo a risolvere un problema di ottimizzazione, riusciamo a risolvere anche il rispettivo problema decisionale.

### Classi di complessità

Dato un problema decisionale P e un algoritmo A, si può dire che A risolve P se restituisce 1 sull’istanza i se e solo se (i,1) appartiene a P, se esso viene risolto in tempo t(n) e in spazio s(n) allora sono rispettivamente il tempo d’esecuzione e lo spazio occupato in memoria da A. Quindi data una funzione f(n), si definiscono Time(f(n)) e Space(f(n)) gli insiemi di problemi decisionali risolvibili in tempo e spazio O(f(n)). La classe P è la classe di problemi risolvibile in tempo polinomiale nella dimensione n dell’input: P= U(infinito,c=0,Time(n^c)).

La classe PSpace è la classe dei problemi risolvibile in spazio polinomiale nella dimensione n dell’input: PSpace= U(infinito,c=0,Space(n^c)). La classe ExpTime è la classe dei problemi risolvibili in tempo esponenziale nella dimensione n dell’input: ExpTime=U(infinito,c=0,Time(2^(n^c))). Si può osservare che P è contenuto in PSpace dato che un algoritmo che richiede tempo polinomiale può avere accesso a un numero polinomiale di locazioni di memoria. In più risulta che PSpace è contenuto in ExpTime dato che se assumiamo le locazioni di memoria come binarie, si hanno n^c locazioni che possono trovarsi 2^(n^c) stati differenti. Tuttavia non si sa ancora se si tratta di inclusioni strette o meno, l’unico finora confermato è quello tra PSpace e ExpTime, di conseguenza esiste un problema risolvibile in tempo esponenziale in cui il tempo polinomiale è insufficiente. La classe P è quindi considerata come la soglia tra i problemi trattabili e quelli che non lo sono.

### La classe NP

In caso di risposta affermativa di un problema di decisione data un’istanza i, è anche richiesto un’ulteriore oggetto y detto certificato, esso certifica il fatto che i soddisfa la proprietà, quindi si può utilizzare come soluzione costruttiva per il problema. Il costo per verificare che un oggetto sia la soluzione di un dato problema può essere usato per caratterizzare la complessità del problema stesso. I problemi decisionali verificabili in tempo polinomiale sono contenuti nella classe NP. La classe NP si basa su algoritmi non deterministici, essi possono eseguire oltre alle normali istruzioni delle funzioni con cui “indovinano” un possibile percorso e da lì proseguono la computazione. le computazioni non deterministiche sono rappresentabili tramite un arco in cui ogni ramo rappresenta un possibile percorso da seguire quando viene “indovinato”. Per determinati algoritmi non è possibile incanalare la computazione nel giusto ramo perché occorrerebbe verificare che tutte le foglie di un sottoalbero portino al successo. Data una funzione f(n), si definisce NTime(f(n)) come l’insieme dei problemi risolvibili in modo non deterministico in tempo O(f(n)), la classe NP è la classe dei problemi risolvibili polinomialmente in modo non deterministico nella dimensione n dell’input: NP= U(infinito,c=0,NTime(N^c)). Dividere un algoritmo non deterministico in due fasi è possibile e si fa in modo tale che la funzione indovina venga fatta nella prima:

* Fase non deterministica di costruzione: si sfrutta la funzione indovina per costruire un certificato che descrive una soluzione data un’istanza;
* Fase deterministica di verifica: l’algoritmo verifica che il certificato sia quello dell’istanza data restituendo 1 se è vero e 0 altrimenti, viene fatta in modo deterministico.

All’interno della gerarchia, la classe NP contiene (per ora) la classe P dato che gli algoritmi deterministici sono casi particolari di quelli che non lo sono, in più la fase di verifica può essere condotta in tempo polinomiale se il certificato ha dimensione polinomiale, questo suggerisce che NP è contenuto in PSpace. La gerarchia però è più complessa, infatti esistono problemi indecidibili in cui nessun algoritmo è in grado di risolvere indipendentemente dal tempo e dallo spazio a disposizione. Anche se sono solo definite in teoria le classi di complessità hanno forti implicazioni pratiche, i tempi di esecuzione polinomiali, tipici della classe P, sono solitamente polinomi di grado basso e quindi utilizzabili.

### NP-completezza

NP-completezza permette di caratterizzare i problemi più difficili all’interno della classe NP, L’espressione “più difficile” va interpretata nel seguente modo: se esistesse un algoritmo capace di risolvere uno solo di questi problemi in tempo polinomiale, allora è possibile farlo con tutti gli altri. La riconducibilità consente di stabilire formalmente quanto un problema è più difficile di un altro: dati due problemi P1 e P2 tali che P1=I1 x {0,1} e p2=I2 x {0,1}, si può dire che P1 è riconducibile polinomialmente a P2 (cond(P1,P2)) se esiste una funzione

f:I1→ I2 avente le seguenti proprietà:

* f è calcolabile in tempo polinomiale;
* per ogni istanza i di P1 e ogni soluzione s appartenente a {0,1}, (i,s) appartiene a P1 se e solo se (f(i),s) appartiene a P2;

Quindi la funzione f trasforma un’istanza di P1 in una di P2 in modo che le istanze positive in P1 risultino tali anche per P2 ( lo stesso vale anche per quelle negative). Di conseguenza, se esiste un algoritmo per risolvere P2, lo si può utilizzare anche per P1, questo significa che P2 è almeno tanto difficile quanto P1. Se cond(P1,P2) e P2 appartiene all’insieme P, allora anche P1 appartiene a quell’insieme. Un problema P1 è più facile di un problema P2 se è possibile trovare un algoritmo che risolve P1 richiamando “poche” volte P2. Un problema decisionale P è detto NP-Hard se ogni problema p appartenente a NP è riducibile polinomialmente a P, esso non deve per forza appartenente all’insieme NP. Un problema è detto NP-Completo quando é NP-Hard e appartiene a NP, esso contiene tutta la difficoltà della classe NP e una sua risoluzione in tempo polinomiale indicherebbe che tutti i problemi in NP sono risolvibili polinomialmente e quindi P=NP. Per dimostrare che P è diverso da NP, bisognerebbe prendere un problema in NP e dimostrare che non è risolvibile polinomialmente. Anche se non si sa niente a riguardo, i problemi NP-Completi non hanno soluzioni ricavabili in tempo polinomiale. Di fronte a un problema NP si possono utilizzare i seguenti approcci:

* Ricerca esaustiva: permette di ricavare una soluzione però ha complessità esponenziale (anche se migliorabile con delle euristiche);
* Ricerca locale: si cerca una soluzione random e si prova a migliorare, la soluzione ottenuta potrebbe non essere ottimale ma molto vicina a esso;
* Approssimazione: si cerca una soluzione non ottimale ma abbastanza buona;

## Approssimazione

L’approssimazione una tecnica che permette di ricavare una soluzione non ottimale ma molto vicina a essa quando si è di fronte a un problema in NP, infatti dato un problema di ottimizzazione, esistono numerose soluzioni ammissibili ma non tutte sono ottime. Il problema della colorazione di un grafo, ad esempio, assegna a ogni vertice un colore, una soluzione ammissibile sarebbe utilizzare un colore per ogni nodo per un totale di n colori. Questa soluzione è ottima nel caso in cui il grafo è completo, nel caso mancasse un arco (u,v), i vertici u e v potrebbero avere lo stesso colore. Dato un problema di ottimizzazione P, un algoritmo di approssimazione per P ha un fattore di approssimazione p(n) sel il costo C della soluzione prodotta differisce di un fattore moltiplicativo <=p(n) dal costo C\* di una soluzione ottimale: max{C/C\*,C\*/C}<=p(n). Dal momento che non esistono soluzioni ammissibili con un valore migliore di quelle ottimali, si può dire che p(n)>=1. p(n) è generalmente una funzione della dimensione n dell’input. In alcuni casi però sono dimostrabili fattori costanti ed esistono problemi approssimabili con un fattore e>0 al costo di una esecuzione più lunga.

## Vertex cover

Il vertex cover è un problema in cui è richiesta la copertura di un grafo tramite i vertici, ovvero che per ogni arco, uno degli estremi deve appartenere alla copertura. Questo problema è risolvibile prendendo un arco e aggiungendo i suoi estremi alla copertura, in questo modo tutti gli archi incidenti agli estremi vengono coperti e quindi possono essere esclusi dal grafo, si itera in questo modo finchè il grafo non è vuoto.

| VertexCover(graph g)  while(g.edges non vuoto)  (u,v)=arco di g  cover=cover U {u,v}  per ogni arco (u,w)  g.edges=g.edges\{(u,w)}  per ogni arco (v,w)  g.edges=g.edges\{(v,w)} |
| --- |

La complessità dell’algoritmo sopra è O(n+m) in n è il numero di nodi e m quello degli archi. La copertura è garantita dal fatto che l’algoritmo non termina finchè sono presenti archi non coperti nel grafo, dato che tutti quelli che lo sono vengono esclusi. L’algoritmo VertexCover garantisce sempre una soluzione che contiene al più il doppio dei vertici di una soluzione ottima, di conseguenza ha un fattore di approssimazione uguale a 2.

### Dimostrazione

Dato un’insieme R contenente tutti gli archi scelti dall’algoritmo: la scelta di un arco (u,v) implica che tutti quelli incidenti agli estremi vengono coperti e quindi esclusi, di conseguenza non vi è nessuna coppia di archi con un estremo in comune. Per coprire gli archi in R sono necessari |R| vertici distinti e, dato che tutti gli archi selezionati sono aggiunti a C, si può dire che |C|=2|R| e quindi il fattore di approssimazione è uguale a 2.

## Commesso viaggiatore

Il problema del commesso viaggiatore è un problema in cui, dato un grafo g completo e con pesi non negativi, si deve trovare un percorso che passa per tutti i nodi che abbiamo il costo minimo. Questo problema nel complesso non ammette approssimazioni per nessuna R, tuttavia esiste un caso in cui si può ottenere una soluzione vicina a quella ottimale: si applica agli archi la disuguaglianza triangolare, ovvero che dati tre archi, quello con peso maggiore è al più uguale alla somma degli altri due. Questa disuguaglianza risulta soddisfatta in molte applicazioni, infatti se i vertici sono punti del piano del piano e i pesi rappresentano la distanza euclidea tra essi, la disuguaglianza è soddisfatta. La disuguaglianza però non altera la NP-Completezza del problema, quindi improbabile che vi sia un algoritmo che lo risolva in tempo polinomiale in modo esatto, però ci si può andare vicino.

### Problema con disuguaglianza triangolare

Supponiamo che S sia la soluzione ottimale del problema con disuguaglianza triangolare, dato che S è un cammino chiuso,se togliamo un qualsiasi arco, otteniamo un albero ricoprente che chiameremo T, il costo di T è al più uguale a quello di S dato che l’arco tolto potrebbe avere peso uguale a 0.

c(S)>=c(T)

Una possibile soluzione per trovare il cammino di costo minimo è utilizzare l’algoritmo di Prim per ricavare il minimo albero ricoprente di quel grafo, il cui peso è c(A). Questo percorso però ci costringe a percorrere due volte ogni arco (una volta per l’andata, l’altra per il ritorno). Dal momento che A è il minimo albero ricoprente, il suo costo sarà minore o uguale a quello di T.

c(T)>=c(A)

Prendiamo ora l’albero A e facciamo una visita completa su di esso, il percorso W ricavato percorre due volte tutti i nodi di A come detto in precedenza, di conseguenza c(W)=2c(A) e questo implica che c(W)<=2c(S).

Per evitare di percorrere più volte gli stessi archi, è possibile prendere queste scorciatoie, chiamiamo questo nuovo percorso W’ e per disuguaglianza triangolare c(W’)<=c(W). Dato che prendere scorciatoie conviene, le prendiamo sempre visitando in profondità il grafo utilizzando l’ordine in cui vengono scoperti i nodi durante la costruzione del minimo albero ricoprente, chiamiamo questo percorso S’ e si ha che:

c(S’)<=c(W)=c(A)

Dato che c(S)>=c(T)>=c(A), si può dire che:

c(S’)<=c(W)=2c(A)<=2c(S)

e quindi si può concludere che il percorso di S’ non può essere peggio del doppio di quello ottimale e quindi il fattore di approssimazione è uguale a 2:

c(S’)<=2c(S)

Nonostante il limite favorevole, esistono algoritmi migliori per questo problema.

### Problema generale

Togliendo la disuguaglianza triangolare non è possibile trovare un algoritmo approssimato in tempo polinomiale con limite p per questo problema a meno che P=NP.

#### Dimostrazione

Si supponga che per p>=1 ci sia un algoritmo approssimato A polinomiale e con limite p, un’eventuale risoluzione del problema problema in tempo polinomiale implica che P=NP. Dato un grafo g=(V,E) come istanza del problema, si vuole determinare un ciclo hamiltoniano efficiente con l’algoritmo approssimato A. Si converte il grafo g in un’istanza del commesso viaggiatore:

g=(V,E) → g’=(V,E’) in cui E’={(u,v): u,v appartengono a V e u!=v}

i pesi di ogni arco saranno 1 se l’arco appartiene a E e p|V|+1 altrimenti.

g’ è quindi una rappresentazione in tempo polinomiale di g rispetto a |V| e |E|.

Se il grafo g ha un ciclo hamiltoniano H, la funzione assegna a ogni arco di H il costo 1, altrimenti ogni percorso in g’ dovrà utilizzare archi non presenti in E. Ogni arco non presente in E ha un costo (p|V|+1)+(|V|-1)>p|V|, dato che sono costosi, vi è differenza tra il costo di un ciclo hamiltoniano e qualsiasi altro ciclo. Applicando quindi l’algoritmo A a g’, è garantito che esso restituisce un giro al più uguale a p volte il giro ottimo, quindi restituisce un ciclo hamiltoniano qualora sia presente in g. Se così non fosse, A restituisce un giro di costo maggiore rispetto a p|V|, di conseguenza si può utilizzare l’algoritmo A per risolvere il ciclo hamiltoniano in tempo polinomiale.

## Programmazione dinamica

La programmazione dinamica è una tecnica algoritmica che permette la risoluzione delle istanze di un problema dividendolo in sottoproblemi più piccoli, esso si basa sul teorema di sottostruttura ottima, ovvero che se la soluzione di un dato problema è ottimale, lo sono anche quelle dei sottoproblemi da cui dipende.

### I sottoproblemi

La scelta dei sottoproblemi a di pari passo con l'identificazione di un modo semplice per trovare la soluzione di un sottoproblema più complesso. Non vi sono regole generali per individuare i sottoproblemi, servono esperienza e intuito, esistono però scelte che si rivelano utili per molti problemi. Il numero complessivo di sottoproblemi deve essere limitato polinomialmente dalla dimensione n dell’input, spesso infatti è necessaria per risolvere un problema la risoluzione dei sottoproblemi da cui dipende. Queste dipendenze sono rappresentabili attraverso un “grafo orientato dei sottoproblemi”, in esso ogni nodo rappresenta un sottoproblema e i suoi archi uscenti puntano ai sottoproblemi da cui dipende.

### Teorema di sottostruttura ottima

Il rapporto tra un soluzione ottimale di un problema e quella dei relativi sottoproblemi è descritto nel problema di sottostruttura ottima: esso indica che una soluzione ottimale di un problema contiene in sè una soluzione ottimale di uno dei sottoproblemi da cui dipende. Dal momento che abbiamo sia i sottoproblemi che le dipendenze, bisogna solo dimostrare per assurdo che una soluzione S sia ottimale per un problema P. Infatti, per assurdo, se la soluzione S di un sottoproblema non fosse ottimale, allora esiste una soluzione S’ migliore di S per tale sottoproblema e di conseguenza ottenere una soluzione ottimale migliore di quella di partenza, contraddicendo così l’ipotesi iniziale. Spesso questa proprietà indica lo spazio dei sottoproblemi, ovvero l’insieme dei sottoproblemi a cui applicare la tecnica di programmazione dinamica. In generale per un certo sottoproblema esistono delle classi di sottoproblemi che possono essere considerate naturali. Un buon metodo per definire lo spazio dei sottoproblemi è analizzare la sottostruttura ottima del problema e di ogni sottoproblema.

### Funzionamento della programmazione dinamica

Un proprietà che un problema di ottimizzazione deve soddisfare è che lo spazio delle soluzioni sia “piccolo”, ovvero che ogni sottoproblema risolva più volte gli stessi sottoproblemi anziché generarne di nuovi, essi sono detti sottoproblemi comuni. La differenza principale con la tecnica divide et impera è che questa genera nuovi sottoproblemi a ogni chiamata ricorsiva, nella programmazione dinamica, invece, ogni sottoproblema viene risolto ricalcolando gli stessi sottoproblemi comuni ogni volta. Questo è un problema perchè la complessità totale di un algoritmo che sfrutta questa tecnica diventa esponenziale in n. Per risolvere questo problema, si utilizza la memoizzazione, una variante della programmazione dinamica in cui si memorizzano i risultati di ogni sottoproblema in una tabella, quando necessario verranno presi direttamente da lì senza ricalcolarli. In questo modo, al costo di tenere occupata della memoria, la complessità degli algoritmi diventa accettabile. Se tutti i problemi devono essere risolti almeno una volta, allora la strategia bottom.up è più efficiente rispetto a quella con memoizzazione perchè non vi è alcun costo per la ricorsione e si ha un costo minore per il mantenimento della tabella. E? comunque possibile che vi siano problemi che non necessitano di essere risolti, in questo caso la memoizzazione porta notevoli vantaggi perchè risolve solamente i problemi necessari.

## Massimo sottoinsieme Indipendente

Il problema del massimo sottoinsieme indipendente è un problema in cui, dato un grafo lineare con nodi pesati, si deve costruire un sottoinsieme indipendente il cui costo sia massimo. Un sottoinsieme di un grafo è detto indipendente quando per ogni coppia di nodi u e v, essi non sono adiacenti, ovvero non esiste un arco che li connette. Per risolvere questo problema, è possibile utilizzare una ricerca esaustiva della soluzione massima su tutte quelle possibili, la complessità però è esponenziale. un algoritmo greedy che sceglie a ogni iterazione un nodo di peso massimo non adiacente a quelli scelti in precedenza permette di ottenere una soluzione ammissibile al problema dato, tuttavia non è sempre ottimale e di conseguenza non è corretto. Con la tecnica divide et impera divido il problema principale in sottoproblemi di dimensioni uguali, restituendo il nodo massimo d’ogni sottoproblema, l’algoritmo però non è comunque corretto perchè i sottoproblemi potrebbero restituire nodi adiacenti rendendo il sottoinsieme non più indipendente. L’unica tecnica possibile è dunque la programmazione dinamica.

### Sottostruttura ottima del massimo sottoinsieme indipendente

Il problema ha sottostruttura ottima, si definisce il problema P(i) riferito ai nodi da 0 a i, se la soluzione S(i) è una soluzione ammissibile e ottimale di P(i), allora valgono le seguenti affermazioni:

* se il nodo i non appartiene a S(i), allora la soluzione è ottimale anche per p(i-1)
* se il nodo i appartiene a S(i), allora S’=S(i)/{i}, una soluzione derivata da S(i) senza considerare il nodo i, è ammissibile e ottimale anche per il problema P(i-2);

#### Dimostrazione

L’ipotesi del primo punto è che, se S(i) non contiene i, allora è ammissibile e ottimale per il problema P(i-1), se non fosse ottimale, allora esiste una soluzione S’ migliore di S(i) per il problema P(i-1), essa però sarebbe una soluzione migliore di S(i) anche per il problema P(i) e questo contraddice l’ipotesi.

L’ipotesi del secondo punto é che, se S(i) contiene i, allora la soluzione S’=S(i)\{i} è ammissibile e ottimale per P(i-2), se non fosse ottimale, allora esiste una soluzione S’’ migliore di S’ per il problema P(i-2), di conseguenza la soluzione S’’’=S’’ U{i} sarebbe una soluzione migliore di S(i) per il problema P(i), contraddicendo così l’ipotesi.

Dato che il massimo sottoinsieme indipendente è un problema di massimizzazione, se S(i-1) è una soluzione ottimale di P(i-1) e S(i-2)U{i} é una soluzione ottimale per P(i-2), allora la soluzione ottimale di P(i) è data dalla soluzione con peso massimo.

#### Dimostrazione

Data una soluzione S di peso massimo tra S(i-1) e S(i-2)U{i}, questo implica che w(S(i-1))<=w(S) e w(S(i-2)<=w(S)- w({i}). Per assurdo supponiamo che S non sia ottimale per P(i), di conseguenza esiste S’ migliore di S. Per sottostruttura ottima, se i appertiene a S’, la soluzione S’’=S’\{i} è una soluzione ottimale per P(i-2), però S’’’=S’’U{i} sarebbe una soluzione con peso maggiore rispetto a S, contraddicendo l’ipotesi. Se invece i non appartiene a S’, per sottostruttura ottima S’ è una soluzione ottimale per P(i-1), di conseguenza, dato che w(S(i-1))<=w(S) e w(S)<W(S’), si ha che W(S(i-1))<W(S’), cosa che contraddice l’ipotesi.

Dato che il problema è risolvibile con la programmazione dinamica, a partire dal problema P(i) e i suoi sottoproblemi P(i-1) e P(i-2), si ha che:

* La soluzione al problema P(0) è 0 dato che si sta considerando un insieme di nodi vuoto;
* La soluzione al problema P(1) è il peso del nodo 1;
* La soluzione ottimale del caso generico è data dalla migliore tra le due sottosoluzione ottimali: w(S(i))=max(w(S(i-1)),w(S(i-2)+peso(i))

#### Dimostrazione

La base dell’induzione è data dal fatto che le soluzioni S0 e S1 sono le uniche possibili per i sottoproblemi P(0) e P(1), il loro peso infatti corrispondono a 0 (dato che non sono presenti nodi nel’insieme) e al peso dell’unico nodo nell’insieme in quel momento. L’ipotesi induttiva è dimostrata nella precedente dimostrazione.

### Algoritmi

| RicMsi(int n)  if(n==0) w=0  else(n==1) w=peso(1)  else  v1=RicMsi(n-1)  v2=RicMsi(n-2)+peso(n)  w=max(v1,v2)  return w | MemMsi(int n)  if(A(n)>=0)  w=A(n)  else(n==0)w=0  else(n==1)w=peso(1)  else  v1=MemMsi(n-1)  v2=MemMsi(n-2)+peso(n)  w=max(v1,v2)  return w | IterMsi(int n)  A(0)=0  A(1)=peso(1)  for i=2 to n  v1=A(i-1)  v2=A(i-2)+peso(i)  A[i]=max(v1,v2) |
| --- | --- | --- |

L’esecuzione della procedura genera due chiamate ricorsive, di conseguenza il costo di RicMSi è esponenziale, tuttavia il numero di sottoproblemi che esso deve risolvere è lineare, il costo infatti è dovuto al fatto che i sottoproblemi comuni vengono svolti più volte. QUindi si utilizza la memoizzazione, esistono due approcci differenti di questa tecnica:

* l’approccio ricorsivo consiste nel risolvere ricorsivamente un sottoproblema i e memorizzarlo nella specifica cella dell’array;
* l’approccio iterativo risolve attraverso un ciclo tutti i sottoproblemi memorizzando la loro soluzione nella rispettiva cella.

Sia la versione ricorsi che quella iterativa devono risolvere tutti i sottoproblemi, esse infatti non presentano nessun vantaggio l’una rispetto all’altra. Vi sono però situazioni in cui la soluzione ricorsiva è preferibile perchè permette di risolvere solamente i sottoproblemi necessari, dato che la versione iterativa è bottom-up e quindi li risolve tutti. L’implementazione ricorsiva comunque non diminuisce la complessità dell’algoritmo nel caso peggiore, però lo si può rendere più veloce con alcune istanze.

### 

### 

### Ricostruzione della soluzione ottimale

E’ possibile ricostruire la soluzione ottimale del massimo sottoinsieme indipendente confrontando i valori delle soluzioni ottimali dei sottoproblemi e da lì stabilire quali nodi appartengono alla soluzione.

| SolutionMsi()  S=0  i=n  while(i>=2)  if(A(i-1)>=A(i-2)+peso(i)) i--  else  S=S U {i}  i-=2  if(i==1) S= S U{1} |
| --- |

## Algoritmo di Bellman-Ford

L’algoritmo di Bellman-Ford è un algoritmo che calcola i cammini minimi di un grafo a partire da una sorgente s. Considerando un cammino c(s,v(k)), bisogna trovare il costo d(s,v(k)), esso si può scrivere come d(s,v(k))=d(s,v(k-1))+w(v(k-1),v(k)), di conseguenza è possibile ricondurre il calcolo di d(s,v(k)) a quello di d(s,v(k-1)). Sia D la distanza tra due nodi, è possibile inizializzare D(s,s)=d(s,s)=0 e per ogni v!=s D(s,v)=d(s,v)=infinito. Se fossimo in grado di effettuare i passi di rilassamento nell’ordine:

* D(s,v(1))=D(s,s)+w(s,v1)
* D(s,v(2))=D(s,v(1))+w(v(1),v(2))
* fino a v(k-1)

arriveremo a k passi per ottenere D(s,v(k))=d(s,v(k)), purtroppo non conosciamo nè gli archi del cammino nè il loro ordine. L’idea di Bellman-Ford è rilassare tutti gli archi (u,v) in E in modo che, se D(s,u)+w(u,v) è minore di D(s,v), essa diventa la nuova distanza D(s,v). Basteranno n^2 passate per calcolare il cammino minimao tra s e v(k) e, dato che ogni passata costa O(m), il tempo totale sarà quindi O(m(n^2)).

| BellManFord(int s)  D(0,s)=0  per ogni v!=s  D(0,v)=infinito  for i=0 to n-1  for v=0 to n-1  v1=D(i-1,v)  v2=infinito  per ogni arco (u,v) in E  if(D(i-1,u)+w(u,v)<v2)  v2=D(i-1,u)+w(u,v)  D(i,v)=min(v1,v2)  return D(n-1) |
| --- |

Nel caso in cui un vertice x non sia raggiungibile da s, allora la distanza tra s e x rimane infinito. E’ possibile notare che l’algoritmo fa molti rilassamenti inutili, infatti se D(s,v(i-1)) è maggiore di d(s,v(i-1)), il nuovo valore di D(s,v(i)) non contribuisce in alcun modo ad aumentare la conoscenza dei cammini minimi: infatti se D(s,v(i-1)) è una stima errata, lo sarà anche D(s,v(i)), il ricalcolo di quest’ultimo sarà quindi risvolto più avanti, quando D(s,v(i-1))=d(s,v(i-1)). Esiste una variante dell’algoritmo di Bellman-Ford per grafi aciclici in cui si effettua il rilassamento utilizzando l’ordine topologico, infatti vengono prima rilassati tutti gli archi l(u)=1, poi quelli con l(u)=2 e così via. Considerando un cammino minimo c(s,v(k)) si che l(s)<l(v(1))<...<l(v(k)), di conseguenza i rilassamenti permettono di ricavare il costo in k passaggi. Ogni arco viene rilassato solamente una volta, di conseguenza la complessità è O(m+n).

| DagBellmanFord(int s)  D(0,s)=0  per ogni v!=s  D(0,v)=infinito  ot ← ordine topologico  for i=0 to n-1  u=ot(i)  per ogni arco (u,v) in E  if(D(s,u)+w(u,v)<D(s,v))  D(s,v)=D(s,u)+w(u,w)  return D(n-1) |
| --- |

### Sottostruttura ottima di Bellman-Ford

Data una soluzione S per il problema P(k,i), si ha che:

* se S è lungo meno di k archi, allora S è ottimale anche per P(k-1,i);
* se S ha esattamente k archi, S’=S\{v(i)}, una soluzione ricavata da S non tenendo in considerazione il nodo v(i), è una soluzione ottimale per P(k-1,j) con j!=i.

#### Dimostrazione

L’ipotesi del primo punto è che, se S è lungo meno di k archi, allora S è ottimale anche per P(k-1,i), se non fosse ottimale, esiste una soluzione S’ migliore di S per P(k-1,i) e di conseguenza lo è anche di P(k,i), dato che S’ è un cammino minimo dalla sorgente a i lungo al massimo k, contraddicendo così l’ipotesi.

L’ipotesi del secondo punto è che,se S ha esattamente k archi, S’=S\{v(i)}, una soluzione ricavata da S non tenendo in considerazione il nodo v(i), è una soluzione ottimale per P(k-1,j) con j!=i, se non fosse ottimale, allora esiste una soluzione S’’ migliore di S’ per P(k-1,j), di conseguenza S’’U{v(i) è una soluzione migliore di S per P(k,i), contraddicendo così l’ipotesi.

## Algoritmo di Floyd-Warshall

L’algoritmo di Floyd-Warshall permette di calcolare le distanze per ogni coppia di nodi nel grafo, per funzionare però non devono essere presenti cicli negativi. Dato un grafo con n vertici, per ogni k si indica d(x,y,k) la distanza più corta tra x e y senza passare nei nodi superiori a k (detto anche cammino k-vincolato). Dato un grafo g orientato e pesato, si può dire che c(x,y,k) è un cammino minimo k vincolato se ha costo minino fra quelli che non utilizzano vertici superiori a k (x e y esclusi). La distanza è data dal costo del cammino c. Si può osservare che d(x,y,0) é uguale al peso dell’arco (x,y) se esso appartiene a E, d(x,y,0)=0 se x=y e d(x,y,n) corrisponde al normale cammino minimo non vincolato. Dato n grafo g orientato e pesato, due vertici x e y e un valore k, il grafo k-vincolato gk=(Vk,Ek) è un grafo ottenuto da g togliendo tutti i vertici superiori a k (a eccezione di x e y) e tutti gli archi a cui incidono. Il cammino c(x,y) è quindi un cammino k-vincolato in g solo se è minimo in gk.

Nel caso non sono prensneti cicli negativi, per ogi coppia di nodi in g esiste sempre un cammino minimo k-vincolato. Ogni sottocammino di un cammino minimo k-vincolato in g è esso stesso un cammino k-vincolato. E’ facile ricavare la distanza d(x,y,k) se si conosce d(x,y,k-1), infatti: per ogni k e per ogni coppia di nodi x,y in V, si ha che:

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

### Dimostrazione

Dato un cammino minimo k-vincolato c(x,y,k), vi possono essere due casi differenti:

* se k non è compreso nel cammino c, allora c(x,y,l) è anche un cammino k-1-vincolato, quindi c(x,y,k)=c(x,y,k-1);
* se k è compreso nel cammino c, dato che i sottocammini di un cammino k-vincolato sono essi stessi cammini k-vincolati e che il vertice k non è compreso in essi, si che i sottocammini sono anche cammini k-1-vincolati e quindi d(x,y,k)=d(x,k,k-1) + d(k,y,k-1) per sottostruttura ottima.

Dato che il problema di minimizzazione il costo d(x,y,k) è dato dalla soluzione con valore minore tra le due precedenti.

### Algoritmo

| FloydWarshall()  for i=0 to n-1  for j=0 to n-1  if(i==j) d(i,j)=0  else((i,j) in E) d(i,j,0)=w(i,j,0)  else d(i,j,0)=infinito  for k=0 to n-1  for i=0 to n-1  for j=0 to n-1  v1=d(i,j,k-1)  v2=d(i,k,k-1) + d(k,j,k-1)  d(i,j,k)=min(v1,v2) |
| --- |

L’algoritmo utilizza n+1 matrici d’appoggio di dimensione n\*n in cui applicare la programmazione dinamica, è possibile procedere dal basso verso l’alto calcolando le distanze con k=0 per ogni x e u e via via incrementando k. La complessità dell’algoritmo è O(n^3) dato che per calcolare tutti i cammini si ha bisogno di 3 cicli. E’ anche possibile eseguire le operazione utilizzando una sola matrice in cui le distanze vengono sovrascritte ogni volta utilizzando lo stesso procedimento mostrato prima.

## Massima sottosequenza comune

La massima sottosequenza comune tra due stringhe X e Y è una stringa di lunghezza massima tra le possibili sottosequenze comuni di X e Y. Date due stringhe X e Y (lunghe rispettivamente n e m) e la massima sottosequenza comune W di lunghezza k, si ha che:

* se X(i)=Y(i) allora W(i)=X(i)=Y(i), W(i+1...k) è la massima sottosequenza comune di X(i+1...n) e Y(i+1...m);
* se X(i) !=Y(i) allora W(i)!=X(i) e W(i+1...k) è la massima sottosequenza comune di X(i+1...n) e Y(i...m);
* Discorso analogo al punto precedente ma con X e Y invertite.

### Dimostrazione

* Supponiamo che X(i)=Y(i) ma W(i)!=X(i), allora la stringa X(i)+W(i...k) è una stringa di lunghezza k+1, questo contraddice l’ipotesi dato che |W|=k. Supponiamo che, per assurdo, W(i+1...k) non sia una sottosequenza massima, di conseguenza esista una sottosequenza Z comune a X(i+1...n) e Y(i+1...m) con |Z|>(k-1), in tal caso la stringa W(i)+Z avrebbe una lunghezza maggiore di K e questo contraddice l’ipotesi.
* Se W(i)!=X(i) allora W è una massima sottosequenza comune tra X(i+1...n) e Y(i...m). Supponiamo che, per assurdo, esista una sottosequenza Z migliore di W comune alle due stringhe precedenti. La lunghezza di Z sarebbe maggiore di k, contraddicendo così l’ipotesi.
* Discorso analogo al precedente ma con X e Y scambiati.

All’inizio delle stringhe X e Y vi è un carattere speciale che indica la stringa vuota, esso non deve essere conteggiato come carattere comune alle stringhe.

### Algoritmo

| RicMcs(string x, string y,int i,int j)  if(i==n or j==m) return 0  else(x(i)==y(i))  return 1+RicMcs(x,y,i+1,j+1)  else  v1=RicMcs(x,y,i+1,j)  v2=RicMcs(x,y,i,j+1)  return max(v1,v2) | IterMcs(string x, string y)  for i=0 to n  len(i,0)=0  for j=0 to m  len(0,j)=0  for i=1 to n  for j=1 to m  if(x(i)==y(j))  len(i,j)=len(i-1,j-1)+1  else  v1=len(i-1,j)  v2=len(i,j-1)  len(i,j)=max(v1,v2) |
| --- | --- |

L’algoritmo ricorsivo ragiona nel seguente modo:

* se una delle due stringhe è finita (i==n oppure j==m), l’algoritmo restituisce 0 dato che non ha trovato caratteri comuni;
* Se l’algoritmo trova due caratteri uguali, allora si calcola ricorsivamente la massima sottosequenza comunque per il resto della stringa e si incrementa di uno il valore ottenuto, dato che abbiamo trovato un carattere comune;
* In tutti gli altri casi, vengono trovate le sottosequenza comuni tra i due casi descritti nella sottostruttura ottima, di queste si sceglie quello di lunghezza massima.

La complessità dell’algoritmo ricorsivo è O(2^n) dato che ricalcola più volte molti degli m\*n sottoproblemi nel caso peggiore, ovvero quando le due stringhe non hanno sottosequenze comuni. Utilizzando una memoizzazione iterativa è possibile ridurre la complessità dell’algoritmo, in pratica vengono memorizzate in una tabella tutte le soluzioni a tutti i sottoproblemi ed esse possono essere riprese in tempo costante senza ricalcolarle. In questo modo la complessità dell’algoritmo scende a O(nm). Una terza possibilità è l’utilizzo di un algoritmo che utilizza la memoizzazione ricorsiva, la complessità nel caso peggiore rimane comunque O(nm) ma essa permette di velocizzare l’esecuzione con alcune istanze.