ALGORITMO DI VISITA GENERICA

Un algoritmo di visita ci permette di visitare tutti i vertici appartenenti ad un grafo. La proprietà fondamentale di una visita è che un vertice deve essere visitato solo una volta. Durante la visita i vertici vengono divisi in tre colori:
1)Bianco: ovvero il vertice non è stato visitato.
2)Grigio: ovvero il vertice è stato visitato ma i suoi adiacenti non sono ancora stati tutti visitati quindi tra i suoi adiacenti c'è qualche vertice bianco.
3)Nero: ovvero il vertice è stato visitato e i suoi adiacenti sono stati visitati quindi sono o neri o grigi.
Durante la visita viene usata una struttura dati detta frangia usata per contenere i vertici che sono grigi.
I vertici passano dal colore bianco al grigio e dal grigio al nero e non si può saltare nessun colore.

FUNZIONAMENTO ALGORITMO
1)Per prima cosa l'algoritmo inizializza i vertici colorandoli tutti di bianco.
2)Dopodiché si visita il nodo sorgente e lo si colora di grigio e lo si inserisce nella frangia.
3)Infine si visitano tutti i nodi del grafo ed il modo in cui si visita il grafo dipende dalla struttura dati che verrà usata come frangia. Se è una coda viene fatta una visita in ampiezza altrimenti, se viene usato uno stack viene fatta una visita in profondità.

Invariante 1: Se l'arco (u, v) appartiene all'insieme E degli archi del grafo G e u è un vertice nero allora il vertice v è un vertice o nero oppure grigio.
Invariante 2: Tutti i vertici grigi o neri sono raggiungibili dalla sorgente s.
Dimostrazione per induzione
Caso Base: Se durante la nostra visita abbiamo visitato solo la sorgente allora l'invariante è vero poiché la sorgente è raggiungibile da sé stessa.
Passo induttivo: Se esiste l'arco (u, v) e se u è raggiungibile da s lo è anche v tramite il cammino che porta da s a u e da u a v.
Invariante 3: Qualunque cammino dalla sorgente a un vertice bianco deve contenere un vertice grigio.
Consideriamo due casi:
1)Se la sorgente è grigia allora l'invariante è vero.
2)Se la sorgente è nera allora è vero poiché per il primo invariante ad un vertice nero c'è adiacente o un vertice nero o uno grigio e se così non fosse si andrebbe a negare il primo invariante.

COMPLESSITA DELLA VISITA GENERICA→ O(m+n)

ALGORITMO DI VISITA IN AMPIEZZA

La visita in ampiezza BFS esamina i vertici del grafo in un ordine ben preciso costruendo un albero di visita chiamato albero BFS. La frangia è usata come una coda. Attraverso questa visita trovo i cammini minimi dalla sorgente verso tutti i vertici v raggiungibili dalla sorgente nei grafi non pesati.

CARATTERISTICHE DELLA VISITA BFS

1)La caratteristica principale è che tale visita costruisce un albero di visita BFS diviso a livelli.

2)La differenza dalla visita generica è che vi sono presenti un vettore delle distanze che mi indica il cammino minimo dalla sorgente fino a un vertice v. Ed inoltre è presente un vettore che mi indica i predecessori di tutti i vertici del grafo.

DIMOSTRAZIONE DELLE DUE PROPRIETA

1)I vertici sono ordinati per livelli nella coda.

2)La coda contiene sempre al massimo due livelli.

Dimostrazione per induzione

Caso base

Se nella frangia c'è solo la sorgente le due proprietà sono vere.

Passo induttivo

Ipotizziamo di essere ad un certo passo del nostro algoritmo e le due proprietà sono vere fino a questo passo, all'interno della coda si possono effettuare due operazioni:

1)dequeue: Quindi vado a rimuovere la testa della coda e quindi le due proprietà sono vere poiché andando a togliere un vertice l'ordine dei vertici nella coda non viene cambiato e il numero di livelli non può aumentare poiché stiamo togliendo un vertice.

2)enqueue: Con questa operazione vado ad aggiungere un nodo alla coda però prima di aggiungere un nodo che non sia più adiacente al nodo in testa viene dapprima rimosso (e quindi viene fatta una dequeue) il nodo in testa alla coda che non aveva più adiacenti, ed è per tale motivo che anche in questo caso la proprietà numero due è ancora vera.

DIMOSTRAZIONE DELLA TERZA PROPRIETA d[v] = δ (s, v)

Per dimostrare che la lunghezza del cammino dalla sorgente fino ad un vertice v e pari alla distanza si dimostra che:

1)d[v] >= δ (s, v)

Tale diseguaglianza è vera poiché una volta ottenuto l'albero di visita BFS esso non è altro che un sottografo del grafo di partenza, quindi, sarà un sottografo che contiene tutti i nodi raggiungibili dalla sorgente e il cammino per raggiungerli sarà o maggiore o uguale al cammino minimo e quindi alla distanza.

2)d[v] <= δ (s, v)

Per dimostrare ciò definiamo l'insieme V_k che è l'insieme dei vertici che hanno distanza k dalla sorgente nel grafo. **$V_k = \{v \in V \mid \delta(s, v) = k\}$**

Dimostriamo per induzione che qualsiasi vertice $v \in V_k$, **d[v] <= δ (s, v)** o **d[v]<= k**

Caso base

Con k = 0 l'unico vertice a distanza 0 dalla sorgente è la sorgente stessa. Quindi l'ipotesi induttiva pari a d[v] <= k è vera.

Passo induttivo

Con k > 0 definiamo due ulteriori insiemi oltre a **V_k**:

1)V_{k-1} ovvero l'insieme dei vertici con distanza k-1 dalla sorgente quindi

$V_{k-1} = \{u \in V \mid \delta(s, u) = k-1\}$

2)U_{k-1} è l'insieme dei vertici con distanza k-1 dalla sorgente con un arco entrante in v.

$U_{k-1} = \{u \in V_{k-1} \mid (u, v) \in E\}$

Sia u il primo vertice di U_{k-1} ad essere scoperto e inserito nella coda. Quando u viene scoperto il vertice v non è ancora stato scoperto per come si sviluppa la visita BFS. La distanza che va dalla sorgente verso il vertice u è pari d[u] = k-1. Quindi la distanza che va dalla sorgente fino al vertice v è pari alla distanza fino al vertice u più (u, v). Quindi:

1→d[u] = k – 1.

2→d[v] = d[u] + 1 (il +1 è rappresentato dall'arco (u, v)).

2→d[v] = k.

Quindi l'ipotesi induttiva pari a d[v] <= k è vera.

ALGORITMO DI VISITA IN PROFONDITA

La visita in profondità esamina i vertici del grafo usando come frangia uno stack. Grazie a questa visita attraverso un contatore scoprirò l'ordine in cui i vertici vengono scoperti (quando un vertice diventa grigio) e l'ordine in cui i vertici vengono chiusi (quando un vertice diventa nero).

INTERVALLI DI ATTIVAZIONE

I tempi in cui i vertici vengono scoperti e vengono chiusi sono detti intervalli di attivazione. Per tenere traccia dei tempi di scoperta e dei tempi di chiusura di un vertice si usano due vettori quello di scoperta indicato con la d[] mentre quello di chiusura indicato con la f[].

Questi intervalli per una coppia di vertici u e v possono essere:

1)Disgiunti

d[u]<f[u]<d[v]<f[v]

In questo caso non c'è relazione tra i due vertici u e v.

2)Uno contenuto nell'altro

d[u]<d[v]<f[v]<f[u]

In questo caso il vertice u è antenato del vertice v mentre, il vertice v è discendente del vertice u.

TEOREMA DEL CAMMINO BIANCO: In una foresta DFS un vertice v è discendente di un vertice u se e solo se al tempo d[u] v è raggiungibile attraverso un cammino composto solo da vertici bianchi.

COMPLESSITA DELLE VISITE BFS E DFS→ O(m+n)

Questo perché derivano dalla visita generica.

GRAFO CICLICO E ACICLICO

Una possibile applicazione della visita DFS è quella per verificare se un grafo contiene dei cicli oppure non ne contiene. Durante la visita DFS per trovare eventuali cicli si classificano gli archi di un grafo e se si trova un arco all'indietro (più avanti viene definito di che cosa si tratta) allora siamo sicuri che il grafo è ciclico in caso contrario se non si trova un arco all'indietro il grafo è aciclico. Questa regola è anche detta **teorema del grafo aciclico.**

CLASSIFICAZIONE ARCHI PER GRAFI ORIENTATI

1)Arco dell'albero→ arco inserito nella foresta DFS che viene percorso la prima volta. Quindi se il colore del vertice v dell'arco appena attraversato (u, v) è bianco allora si tratta di un arco dell'albero.
2)Arco all'indietro→ arco che collega un vertice ad un suo antenato. Quindi se il colore del vertice v dell'arco appena attraversato (u, v) è grigio allora si tratta di un arco all'indietro.
3)Arco in avanti→ arco che collega un vertice ad un suo discendente. Se la visita di v è terminata (quindi v è nero) e si scorre l'arco (u, v) se u è antenato di v allora l'arco è in avanti. Quindi **d[u] < d[v].**
4)Arco d'attraversamento→ arco che collega due vertici che non hanno relazione tra di loro. Se la visita di v è terminata (quindi v è nero) e si scorre l'arco (u, v) se u è antenato di v allora l'arco è in avanti.

CLASSIFICAZIONE ARCHI PER GRAFI ORIENTATI

In questo caso è importante sottolineare che si classificano gli archi percorrendoli una sola volta.

1)Arco dell'albero→ arco inserito nella foresta DFS che viene percorso la prima volta. Quindi se il colore del vertice v dell'arco appena attraversato (u, v) è bianco allora si tratta di un arco dell'albero.
2)Arco all'indietro→ arco che collega un vertice ad un suo antenato. Quindi se il colore del vertice v dell'arco appena attraversato (u, v) è grigio o nero allora si tratta di un arco all'indietro.

COMPLESSITA DELLA RICERCA DI UN CICLO→ O(m+n)
Questo perché derivano dalla visita generica.

DEFINIZIONE DI UN DAG: Un DAG è un grafo orientato e aciclico.

ORDINAMENTO TOPOLOGICO

Dato un DAG è possibile ordinare i vertici di un DAG secondo un ordinamento topologico ossia, se esiste un arco che va dal vertice u al vertice v, nell'ordinamento topologico dovrà venire prima il vertice u e poi il vertice v.

1)PROPRIETA DEI DAG: Un DAG può possedere uno o più diversi ordini topologici.

DATO UN DAG COME OTTENERE UN ORDINE TOPOLOGICO?

Esistono due metodi per trovare un ordine topologico:

1)Uno basato su un'intuizione detto: nodi sorgente e nodi pozzo.

2)Il secondo più complesso ma che risolve il problema con un'efficienza maggiore. Esso viene fatto tramite una visita DFS.

NODI SORGENTE E NODI POZZO

I nodi sorgente sono dei nodi che non hanno archi entranti mentre i nodi pozzo sono nodi che non hanno archi uscenti.

2)PROPRIETA DEI DAG: In un DAG ci sono almeno un nodo pozzo e almeno un nodo sorgente.

Il funzionamento di tale metodo è il seguente: nel DAG viene scelto un nodo sorgente e viene messo come primo elemento dell'ordinamento topologico. Tale nodo viene poi rimosso dal grafo, così facendo vengono eliminati i suoi archi uscenti che sono archi che entrano in altri nodi e di conseguenza vengono creati dei nuovi nodi sorgenti che verranno trattati come il primo nodo sorgente. Se alla fine dell'algoritmo ci sono ancora dei nodi nel grafo allora il grafo non era un DAG.

Nomenclatura

G': copia del grafo di partenza. Sarà il grafo in cui eliminare i nodi sorgenti.
ord: lista in cui sarà contenuto l'ordinamento topologico.

DIMOSTRAZIONE METODO NODI SORGENTE E NODI POZZO

Si dimostra per induzione che al passo i dell'algoritmo non esiste nessun cammino nel grafo G che porta da un vertice in G_i' ad uno in ord_i.

Caso base i=0

In questo caso ord_0 è pari all'insieme vuoto quindi non essendoci nessun vertice da raggiungere l'ipotesi induttiva è vera.

Passo induttivo i=k

Al passo k-esimo scegliamo un nodo sorgente u in G'_k. Per come è stato scelto u non ci sono cammini in G che raggiungono u passando solo per vertici in G'_k. L'unico modo per raggiungere u in ord sarebbe quello di passare per un nodo v già presente in ord però tale metodo non attuabile poiché il nodo v non è raggiungibile poiché anche lui dovrebbe essere raggiunto tramite un nodo in G'_k e ciò non è possibile.

ORDINAMENTO TOPOLOGICO TRAMITE VISITA DFS

Eseguendo una visita DFS su DAG se si riordinano i vertici secondo il loro tempo di fine visita in ordine decrescente si ottiene un ordinamento topologico.

Questo per ogni arco (u, v) presente nel grafo con f[u]>f[v].

DIMOSTRAZIONE PER ASSURDO

Supponiamo per assurdo che per almeno un arco (u, v) f[u]<f[v]. Ovviamente f[u]=f[v] è impossibile. Esistono due possibilità:

1)d[u]<f[u]<d[v]<f[v]

Ma ciò non possibile poiché non si può chiudere il vertice u prima che abbia visitato tutti i suoi adiacenti e in questo caso non avrebbe visitato il vertice v ma sarebbe stato chiuso prima di poterlo fare.

2)d[v]<d[u]<f[u]<f[v]

In questo caso vorrebbe dire che oltre all'arco (u, v) sarebbe presente nel grafo anche l'arco (v, u) e ciò porterebbe ad avere un ciclo e quindi non sarebbe possibile ottenere un ordinamento topologico.

Per tali ragioni eseguendo una visita DFS su un DAG e riordinando i vertici in ordine decrescente di tempi di fine visita l'ordinamento topologico ottenuto è corretto.

RELAZIONE DI EQUIVALENZA: Una relazione di equivalenza è una relazione che include tre relazioni: la relazione riflessiva, la relazione simmetrica e la relazione transitiva.

GRAFI NON ORIENTATI: COMPONENTI CONNESSE

In un grafo non orientato **la relazione di raggiungibilità** ovvero che esiste un cammino tra due vertici u e v tale che u è raggiungibile da v e viceversa è una relazione di equivalenza.

1)È riflessiva poiché ogni vertice è raggiungibile da sé stesso.

2)È simmetrica poiché se u è raggiungibile da v allora per lo stesso cammino anche v è raggiungibile da u.

3)È transitiva poiché se v è raggiungibile da u e w è raggiungibile da v allora anche w è raggiungibile da u per concatenazione tramite il cammino u->v e il cammino v->w.

GRAFI ORIENTATI: COMPONENTI FORTEMENTE CONNESSE (cfc)

In un grafo orientato **due vertici u e v sono fortemente connessi** quando esiste un cammino che permette di raggiungere da u il vertice v e viceversa. Tale relazione è una relazione di equivalenza. Il simbolo di cfc è il seguente <->.

GRAFO FORTEMENTE CONNESSO: Tale grafo è fortemente connesso se tutti i suoi vertici sono fra loro fortemente connessi.

1)È riflessiva poiché ogni vertice è fortemente connesso con sé stesso.

2)È simmetrica poiché se u<->v.

3)È transitiva poiché se u<->v e v<->z allora u<->z.

Per concludere il discorso delle cfc esse sono per un grafo le classi di equivalenza della relazione connessione forte. Una cfc di un grafo orientato G è un sottografo di G fortemente connesso e massimale.

COME CALCOLARE LE COMPONENTI FORTEMENTE CONNESSE DI UN GRAFO?
Esistono due metodi per calcolare le cfc di un vertice u di un grafo:
1)Il primo metodo è attraverso una semplice operazione di intersezione tra l'insieme dei vertici discendenti del vertice u e l'insieme dei vertici antenati del vertice u.
2)Il secondo metodo è attraverso l'algoritmo di kojasaru.

FUNZIONAMENTO PRIMO METODO
Per calcolare la cfc di un vertice u in questo caso si calcola l'insieme formato dai vertici discendenti di u che sono i vertici raggiungibili da u, poi si calcola l'insieme formato dai vertici antenati di u quindi i vertici che possono raggiungere u, infine si fa l'intersezione tra l'insieme dei vertici discendenti e l'insieme dei vertici antenati e il risultato sarà l'insieme dei vertici che appartengono alla stessa cfc.

LEMMA DEL CAMMINO FORTEMENTE CONNESSO: Se due vertici x e y di un grafo appartengono alla stessa cfc, allora nessun cammino tra di essi può abbandonare tale cfc.
Dimostrazione: Siano x e y due vertici appartenenti alla stessa cfc. Sia z un vertice. Nel grafo esistono gli archi <x, z> e <z, y>.
Dimostrare che z appartiene alla stessa cfc di x e y.
Per dimostrarlo basta verificare che z sia raggiungibile da x e che z possa raggiungere x. Per dimostrare che x raggiunge z è banale poiché esiste l'arco <x, z> mentre per dimostrare che z raggiunge x basta pensare che x e y appartengono alla stessa cfc quindi esistono anche gli archi <x, y> e <y, x>. Quindi per una concatenazione di cammini tramite gli archi <z, y> e <y, x> vediamo che z può raggiungere x.
Per tale motivo z appartiene alla stessa cfc di x.

TEOREMA DEL SOTTOALBERO FORTEMENTE CONNESSO
In una qualunque visita DFS di un grafo G orientato tutti i vertici di una cfc vengono inseriti in uno stesso sottoalbero.

Due proprietà:
1)Proprietà: Esiste sempre per ogni grafo diretto, almeno un ordine di visita DFS dei suoi nodi tale per cui le cfc sono già separate nella foresta di visita.
2)Proprietà: Un grafo G ed il suo trasposto G' hanno le stesse cfc.

Seguendo queste due proprietà si può fare una visita DFS per trovare le cfc di un grafo. In particolare, si potrebbe trovare un albero DFS contenente tutte le cfc e tagliare tale albero nei punti opportuni in modo da ottenere le cfc tra loro separate.
Il problema è trovare i punti in cui tagliare l'albero.

Per far ciò basta eseguire **l'algoritmo di kosajaru.** L'algoritmo si divide in tre fasi:
1)Si effettua una visita DFS sul grafo e si riordinano i vertici in ordine decrescente di tempo di fine visita.
2)Si crea il grafo trasposto del grafo visitato al passo 1.
3)Si visita il grafo trasposto considerando l'ordine dei vertici trovato al passo 1.

Il motivo per cui si devono considerare i vertici in ordine decrescente di tempo di fine visita è il seguente: siccome nell'albero di visita DFS ci possono essere più cfc dati due vertici u e v si possono presentare due casi:
1)I vertici u e v appartengono alla stessa cfc.
2)I vertici u e v non appartengono alla stessa cfc. In questo caso, per esempio, può esistere un cammino che va dal vertice u al vertice v nel grafo e siccome non sono nella stessa cfc non esiste un cammino che va da v a u. Dunque, per separare i vertici u e v nelle loro cfc e quindi in sottoalberi di visita DFS diversi, con la visita DFS nel grafo trasposto tale visita dovrà partire da u se u antenato di v. In questo modo, se nel grafo trasposto non esiste un altro cammino da u a v come nel grafo di partenza i due vertici non sono nella stessa cfc e verranno collocati in due sottoalberi di visita DFS diversi.

TECNICA ALGORITMICA: LA TECNICA GREEDY

La tecnica greedy è utilizzata per risolvere i problemi di ottimizzazione ovvero problemi per cui andiamo a cercare una soluzione ottima. Inoltre, i problemi per essere risolti devono godere di due proprietà:

1)Proprietà della sottostruttura ottima: Un problema gode della proprietà della sottostruttura ottima se una soluzione ottima del problema include le soluzioni ottime dei suoi sottoproblemi.

2)Proprietà della scelta greedy: Un problema gode di tale proprietà se è sempre possibile attribuire alle variabili un valore detto appetibilità e che ci permetta poi di scegliere la variabile con la migliore appetibilità.

Esistono due tipi di **appetibilità fissa e modificabile**. L'appetibilità **fissa** significa che una volta che sono stati definiti i valori di appetibilità non cambiano durante l'esecuzione dell'algoritmo. L'appetibilità **modificabile** significa che i valori di appetibilità cambiano durante l'esecuzione dell'algoritmo.

DIFFERENZA TRA PROBLEMI DI OTTIMIZZAZIONE E PROBLEMI DI RICERCA

La differenza tra questi due tipi di problemi è che una soluzione di un problema di ottimizzazione è anche una soluzione di un equivalente problema di ricerca mentre non è vero il contrario. Inoltre, nei problemi di ottimizzazione c'è una funzione obiettivo che da un punteggio da attribuire a ciascuna soluzione del problema e in base a tale punteggio si capisce se una soluzione è ottima o meno.

TECNICA GREEDY – ZAINO FRAZIONARIO

Un ladro entra in un magazzino e trova $\boldsymbol{n}$ oggetti. L'i-esimo oggetto $\boldsymbol{oi}$ ($\boldsymbol{oi}$ è come vengono identificati gli oggetti) ha un valore di $\boldsymbol{ci}$ euro e pesa $\boldsymbol{pi}$ chilogrammi.

Il valore dell'oggetto $\boldsymbol{vi=ci/pi}$ è il suo valore per unità di peso. Gli oggetti sono frazionabili (ad esempio sono delle polveri), quindi il ladro ne può prendere anche solo una frazione $\boldsymbol{xi}$, il cui valore può essere $0 \leq \boldsymbol{xi} \leq 1$. In tal caso il valore della parte presa sarà $\boldsymbol{cixi}$. Il ladro ha solo uno zaino, che può contenere oggetti per un massimo di $\boldsymbol{P}$ chilogrammi.

<u>Quali oggetti e in quale quantità dovrà prendere il ladro per ottenere il massimo guadagno dal furto?</u>

Per ottenere il massimo del guadagno dal furto il ladro dovrà prendere prima l'oggetto (o gli oggetti) di maggior valore e prenderlo (o prenderli) in tutta la sua quantità o meglio prenderlo finché lo zaino non è pieno o finché l'oggetto non è finito.

TECNICA GREEDY: MASSIMO NUMERO DI INTERVALLI DISGIUNTI

Dato un insieme di intervalli, in generale non tutti disgiunti fra di loro, trovarne un sottoinsieme costituito da intervalli tutti disgiunti e tale che il numero di intervalli sia il massimo.
Un intervallo è formato da un tempo di inizio intervallo e un tempo di fine intervallo.
Per risolvere tale problema scegliamo ogni volta l'intervallo che finisce prima. Quindi tale problema ha una appetibilità fissa.

FUNZIONAMENTO: L'algoritmo si divide in tre fasi:
1)Si ordina l'insieme degli intervalli in una sequenza di intervalli ordinati in ordine non decrescente per tempo di fine intervallo.
2)Si crea una lista vuota, che sarà la soluzione, in cui saranno contenuti gli intervalli disgiunti.
3)Si scorre la sequenza di intervalli ordinati prima. Se l'intervallo A inizia dopo la fine dell'ultimo intervallo presente nella soluzione allora si può aggiungere l'intervallo A alla soluzione. In caso contrario, non si può aggiungere l'intervallo A nella soluzione.

DIMOSTRAZIONE DI CORRETTEZZA DELL'ALGORITMO

Si vuole dimostrare che all'uscita dell'algoritmo la sequenza A_1, A_2, … e A_k di intervalli disgiunti scelti è una sequenza massimale.

Si definiscono i seguenti invarianti:
S: è l'insieme di intervalli già esaminati.
MAX: è la sequenza massimale di intervalli disgiunti già selezionati e appartenenti all'intervallo S.
PRIMAMAX: è fra le sequenze massimali di intervalli disgiunti quella che finisce prima.
PRIMAVISTI: è la sequenza di intervalli disgiunti ancora da esaminare. Ogni intervallo che appartiene a questo insieme termina dopo ogni intervallo che appartiene a S.

Caso base
All'inizio dell'algoritmo l'insieme S è vuoto, perciò, anche MAX sarà vuoto il che è corretto poiché l'insieme massimale di un insieme vuoto di intervalli analizzati e appunto un insieme vuoto. Per lo stesso motivo anche PRIMAMAX è corretto.
Inoltre, anche PRIMAVISTI è corretto poiché contiene tutti intervalli che finiscono dopo la sequenza vuota mantenendo corretto l'invariante.

Passo induttivo
Consideriamo gli invarianti corretti fino a un certo passo dell'algoritmo e analizziamo questi due casi. Sia A l'intervallo che termina prima fra quelli ancora da esaminare che verrà preso dall'insieme PRIMAVISTI. Esistono due possibilità:

1)caso: *L'intervallo A inizia prima della fine dell'ultimo intervallo A_k che si trova nella sequenza massimale.* In tal caso non si aggiunge A alla soluzione e quindi siccome gli invarianti fino a quel momento erano corretti non aggiungendo A lo rimangono. Anche PRIMAVISTI è ancora corretto per via del metodo in cui andiamo a selezionare l'intervallo A.
2)caso: *L'intervallo A inizia dopo la fine dell'ultimo intervallo A_k che si trova nella sequenza massimale.* In tal caso si aggiunge A alla soluzione e quindi il numero massimale di intervalli che prima aveva cardinalità pari a k adesso diventa pari a k+1 e quindi l'invariante MAX è corretto. Inoltre, anche PRIMAMAX è corretto poiché la nuova sequenza massimale o meglio tutte le sequenze massimali includono l'intervallo A e non esistono sequenze massimali che non includono A che terminano prima. Anche PRIMAVISTI è ancora corretto per via del metodo in cui andiamo a selezionare l'intervallo A.

TECNICA GREEDY: ALGORITMO MOORE
Dato un insieme di JOBs caratterizzati da una durata e da una scadenza, trovare il massimo numero di lavori che possono essere eseguiti entro la loro scadenza. Questa sequenza da trovare si chiama scheduling ed esso è una sequenza di lavori che devono essere eseguiti uno dopo l'altro consecutivamente. Questo algoritmo ha un'appetibilità modificabile.

FUNZIONAMENTO: L'algoritmo si divide in tre fasi:
1)Si ordina l'insieme dei JOBs in una sequenza di JOBs ordinati in ordine non decrescente di tempo di scadenza.
2)Si crea una lista vuota che è la soluzione in cui saranno contenuti i JOBs.
3)Si scorre la sequenza di JOBs ordinati prima e si prende il JOB L non ancora analizzato. Ci sono due possibilità:
Se il tempo di durata del JOB L sommata il tempo di esecuzione della sequenza di scheduling trovata finora è minore uguale del tempo della scadenza del JOB L allora si può aggiungere L nella soluzione.
In caso contrario se il tempo di durata del JOB L sommata il tempo di esecuzione della sequenza di scheduling trovata finora è maggiore del tempo della scadenza del JOB L allora si valuta se scartare il JOB L oppure un JOB presente nella sequenza di scheduling. Il JOB che viene scartato è quello che ha durata più alta e se ne viene tolto uno dalla sequenza allora si aggiunge L alla sequenza altrimenti se quello con durata maggiore è il JOB L si elimina proprio quest'ultimo.

DIMOSTRAZIONE DI CORRETTEZA DELL'ALGORITMO

<u>Dato un insieme di JOBs dire se dopo l'algoritmo la soluzione ottenuta è una sequenza di scheduling massimale di JOBs che ha una durata totale minima.</u>

Si definiscono i seguenti invarianti:
S: è l'insieme di tutti i JOBs finora esaminati.
SOL: è una sequenza massimale di JOBs già esaminati che rispetta le scadenze. Inoltre, fra tutte le sequenze di scheduling massimali è quella che ha durata minima. Infine, SOL è ordinato in ordine non decrescente per tempi di scadenza.
S': è l'insieme di tutti i JOBs che non sono ancora stati esaminati che hanno una scadenza maggiore o uguale rispetto a tutti i JOBs che appartengono a S.

Caso base: L'invariante è vero quando sia S e sia SOL sono vuoti.

Passo induttivo:
Sia T_k l'istante di fine scheduling con SOL = L_1, L_2, …, L_k.
Sia L il primo JOB non ancora esaminato che ha scadenza prima di tutti gli altri JOBs.
Siano S la scadenza di L e D la durata di L.
Consideriamo ora che SOL sia una sequenza di scheduling massimale di k JOBs e che venga estratto L dalla sequenza di JOBs da esaminare, esistono due casi:
1)caso: il primo in cui $T_k + D > S$. In questo caso si verifica quale JOBs ha durata massima tra i JOBs presenti nella soluzione SOL e il JOB L. Se L è quello di durata più alta si elimina L se invece è un JOBs Y presente in SOL ad avere la durata più alta si toglie Y dalla soluzione SOL e viene aggiunto al suo posto il JOB L. L'invariante SOL rimarrebbe corretto poiché il numero totale di JOBs rimane k (poiché ne abbiamo tolto uno per metterne un altro) inoltre migliorerebbe anche la durata della sequenza di scheduling poiché mettiamo un JOB con una durata minore.
2)caso: il primo in cui $T_k + D <= S$. In questo caso il JOB L viene aggiunto alla soluzione e quindi SOL sarebbe ancora corretto poiché il numero di JOB presenti passa da k a k+1.

TECNICA GREEDY: ALGORITMO DEL CODICE DI HUFFMAN

Dato un testo scritto secondo un certo alfabeto C, trovare una codifica che sia minimale, cioè che renda minima la lunghezza del testo codificato.

FUNZIONAMENTO: L'algoritmo esegue i seguenti passi:
1)Per ciascun carattere crea un albero formato solo da una foglia contenente il carattere e la frequenza del carattere.
2)Fonde i due alberi che hanno le due frequenze minime e costruisce un nuovo albero che ha come frequenza la somma delle frequenze degli alberi fusi.
3)Ripete la fusione finché si ottiene un unico albero.
Questo algoritmo ha un'appetibilità modificabile.

DIMOSTRAZIONE DI CORRETTEZA DELL'ALGORITMO

Dimostriamo che l'algoritmo restituisca veramente un albero che renda minima la codifica di un testo.
Dato un alfabeto con funzione di frequenza esiste un albero di Huffman che noi non conosciamo, costituito da vari sottoalberi e in cui le foglie rappresentano i caratteri dell'alfabeto.

Caso base
All'inizio dell'esecuzione dell'algoritmo, quando vengono creati alberi di un solo nodo corrispondenti ai vari caratteri, è ovvio che essi appartengano a questo ipotetico albero di huffman.

Passo induttivo
Ora dimostriamo che al k-esimo passo dell'algoritmo, quando vengono uniti i due alberi Ta e Tb di frequenza minima in un unico albero Tab, la foresta risultante sia ancora formata da sottoalberi (e nodi) tutti appartenenti all'albero di huffman.
Consideriamo tra i nodi interni all'albero di huffman che l'algoritmo non ha ancora creato quello di profondità massima. Ovviamente, questo nodo interno deve avere due sottoalberi non nulli, e non può esistere scelta migliore che collocarci i due sottoalberi di frequenza minore Ta e Tb, perché saranno quelli che corrisponderanno a una codifica più lunga in tutto l'albero di huffman non ancora scoperto. Quindi Ta e Tb saranno figli dello stesso nodo più profondo z, che corrisponderà al sottoalbero Tab. Il sottoalbero Tab è quindi un sottoalbero dell'albero di huffman.

TECNICA GREEDY: ALGORITMO DI DIJKSTRA

Il cammino minimo tra due nodi v e w in cui w è raggiungibile da v si dice distanza. Se w non è raggiungibile da v si dice che la loro distanza è infinita. Questo algoritmo serve appunto a trovare il cammino minimo tra due nodi in un grafo pesato.

APPROCCIO CORRETTO MA MOLTO INEFFICIENTE: per trovare un cammino minimo si potrebbero esaminare tutti i cammini fra due nodi calcolandone le rispettive lunghezze e scegliere poi il cammino con lunghezza minima. Tale metodo però è inefficiente poiché, ha una complessità esponenziale. L'obiettivo è quello di trovare i cammini minimi senza dover analizzare tutti i possibili cammini esistenti.

IDEA DELL'ALGORITMO: Costruiamo un albero di visita che contenga tutti i vertici raggiungibili da un nodo di partenza s. Partendo da un albero che contiene solo s, ad ogni iterazione della visita scegliamo un vertice u e lo aggiungiamo all'albero, aggiungendo anche un arco (del grafo) che lo collega all'albero. Alla fine della visita, l'albero conterrà tutti i vertici raggiungibili da s ed i cammini minimi tra s ed i vertici.
Come scegliamo u e l'arco da aggiungere? Che tipo di visita adottiamo?
L'algoritmo di Dijkstra è una visita BFS in cui la **frangia è una coda con priorità** ed è gestita come in un algoritmo greedy con **appetibilità modificabili.** Il vertice u viene scelto in base alla sua appetibilità greedy e ogni volta che aggiungo u all'albero aggiorno l'appetibilità dei vertici v ad esso adiacenti.
L'appetibilità di un vertice u è data da una stima della distanza tra s ed u che l'algoritmo ha in un determinato istante (che indicheremo con d[u]).
Inizialmente, tutti i vertici sono stimati a distanza infinito da s, tranne s stesso che (banalmente) ha distanza d[s] = 0 da sé stesso.
Ad ogni ciclo di una visita, scelgo un vertice u da aggiungere all'albero, tra quelli non ancora inseriti ma raggiunti dalla ricerca (cioè grigi), e scelgo quello con distanza stimata da s minima.
Come aggiorno le distanze stimate/l'appetibilità dei vertici non neri?
Se il cammino tra s e v (un nodo adiacente ad u) che passa da u (il nodo appena aggiunto all'albero) è di lunghezza minore a quello finora stimato (ovvero se questa disequazione è vera **d[v] > d[u] + W(u, v)**) ho trovato una nuova distanza stimata quindi un nuovo cammino minimo tra s e v migliore di quello precedente. Perciò la nuova distanza per raggiungere d[v] sarà pari a **d[v]=d[u] + W(u, v)**.

L'algoritmo di DIJKSTRA non funziona con archi che hanno peso negativo.

FUNZIONAMENTO:

1)Si setta la distanza dei vertici a infinito tranne la sorgente che si setta a 0.

2)Viene scelto il nodo u con distanza minore. Al primo ciclo verrà scelta la sorgente.

3)Per ogni suo vertice adiacente v, se d[v]>d[u]+W(u, v), la sua distanza viene aggiornata e si setta d[v]=d[u]+W(u, v). Si ripetono gli ultimi due punti fino ad arrivare al vertice desiderato (o dopo aver visitato tutti i vertici del grafo).

PROPRIETA DEI SOTTOCAMMINI MINIMI:

Un sottocammino minimo di un cammino minimo è un cammino minimo.

Sia u ⇝v un sottocammino di un cammino minimo da s a t:

s⇝u⇝v ⇝t

Se u⇝v non fosse minimo, ci sarebbe un altro cammino da u a v di costo inferiore. Ma allora sostituendo tale sottocammino nel cammino da s a t, si otterrebbe un cammino da s a t di costo inferiore rispetto al cammino minimo considerato prima. Per tale motivo u ⇝v è un cammino minimo.

Dimostriamo la correttezza dell'algoritmo: *ossia che alla fine della sua esecuzione il cammino dalla sorgente a qualunque vertice raggiungibile da essa è pari alla distanza.*

Caso base: All'inizio consideriamo la distanza tra la sorgente e sé stessa. In questo caso il teorema è vero poiché il cammino dalla sorgente a sé stessa è pari a 0.

Passo induttivo:
Abbiamo due insiemi di nodi, S che contiene quelli già visitati, e D, che contiene quelli ancora da visitare.
Quando estraiamo un vertice u durante la visita (quindi dall'insieme D) possono verificarsi due casi:
1)caso: d[u]=∞. Ciò vuol dire che non esiste nessun cammino che porta dalla sorgente fino al vertice u.
2)caso: d[u]!=∞. In questo caso il vertice u è raggiungibile quindi u avrà anche un predecessore, per esempio, il vertice r che apparterà all'albero della visita anche detto albero dei cammini minimi. (π[u] = r).

Supponiamo per assurdo che tra la sorgente e il vertice u esista un cammino di peso minore di d[u]. Questo cammino deve contenere un arco tra un vertice x che appartiene a S e un vertice y che appartiene a D. Questo cammino tra s e u può essere visto come la concatenazione di tre cammini.
s⇝x⇝y⇝u, se tale cammino è un cammino minimo anche s⇝x⇝y lo è per la proprietà del sottocammino minimo. Quindi d[y] sarà pari a W(s⇝x⇝y).
Quindi W(s⇝x⇝y⇝u) sarà pari a:
W(s⇝x⇝y⇝u) = W(s⇝x⇝y) + W(y⇝u).
W(s⇝x⇝y⇝u) = d[y] + W(y⇝u). (Quindi d[u]>=d[y]).
Ma siccome u è stato estratto prima di y d[u] non può essere maggiore o uguale rispetto a d[y] ma dovrà essere minore. Questo vuol dire che il cammino per raggiungere y rispetto a u è maggiore di quello per raggiungere u. Quindi per come sono stati estratti i vertici i valori delle distanze sono d[y]>d[u] e quindi, per tale motivo questa supposizione per assurdo è errata.

Minimo Albero Ricoprente
Un minimo albero ricoprente dato un grafo G connesso, non orientato e pesato, è un sottografo di G tale che:
• È un albero libero (ovvero un grafo connesso, aciclico e non orientato);
• Contiene tutti i nodi di G;
• **(ammissibilità)** e fra tutti i sottografi di G soddisfacenti le due condizioni precedenti, è quello (o uno di quelli) in cui **(criterio di ottimalità)** la somma dei pesi degli archi è minima.

Taglio

Un taglio è una partizione dell'insieme V di un grafo in due parti non vuote, S e V-S. Si dice che un arco (u, v) attraversa il taglio se i due vertici u e v appartengono uno a un insieme (S) e l'altro al secondo insieme (V-S). In genere si scrive che il taglio (S, V-S) taglia l'arco (u, v).

Il lemma del taglio
Sia A un insieme di archi appartenenti a un MAR di un grafo G. Consideriamo un taglio non attraversato da alcun arco di A; siano S e V-S le sue due parti.
Sia (u, v) l'arco (o un arco) di peso minimo fra tutti gli archi del grafo che attraversa il taglio (tale arco viene detto arco leggero): allora (u, v) appartiene a un MAR (di G) che estende A, cioè l'insieme A ∪ {(u, v)} è anch'esso un sottoinsieme di un MAR del grafo G.

Proprietà importanti per dimostrare il lemma del taglio

PROPRIETÀ P1: Se in un albero libero, cioè un grafo non orientato, connesso ed aciclico, si elimina un arco, il grafo (albero) si scinde in due sottografi (alberi) distinti non connessi fra di loro (e diventa quindi un grafo non connesso).
Questo succede perché fra due nodi di un albero vi è uno e un solo cammino (perché altrimenti nel grafo non orientato connesso ci sarebbe un ciclo). Allora se si elimina un arco appartenente a tale cammino non ci sono altri cammini fra i suoi due vertici quindi non ci sono cammini che permettono di far raggiungere un vertice con un altro, e quindi nemmeno fra i due sottoalberi esistono cammini che li uniscono.

PROPRIETÀ P2: Se si connettono con un arco due nodi appartenenti rispettivamente a due alberi fra loro non connessi, si ottiene un albero.

DIMOSTRAZIONE DEL LEMMA DEL TAGLIO

Dato un grafo G, siano dunque:
1) A un insieme di archi di G che supponiamo appartenenti ad un MAR di G; cioè A può essere esteso a un MAR;
2) (S, V-S) un taglio che non taglia alcun arco di A;
3) (u, v) è l'arco (o un arco) di peso minimo fra quelli di G tagliati dal taglio (S, V-S); (tale arco viene detto anche arco leggero).

Un minimo albero ricoprente di G che estende l'insieme A di archi è per definizione un albero di peso minimo fra tutti gli alberi ricoprenti di G che estendono (cioè contengono) A.
Proviamo quindi a trovare un albero ricoprente (e poi un minimo albero ricoprente) che estenda A.

Un albero ricoprente AR contenente A deve connettere tutti i nodi di G quindi deve contenere un cammino da u a v. Poiché u e v si trovano da parti opposte del taglio, un tale cammino deve contenere almeno un arco (x, y) che attraversa il taglio.

Se nell'albero AR sostituiamo l'arco (x, y) con l'arco (u, v), per la concatenazione delle proprietà 1 e 2 otteniamo ancora un albero, AR_{uv}, ancora ricoprente (perché i nodi sono gli stessi, cioè ci sono tutti i nodi del grafo), e il cui peso totale è minore o uguale al peso di AR, perché (u, v) è, fra gli archi che attraversano il taglio, uno di peso minimo. Infatti, $W(u, v) < W(x, y)$.

Teorema dell'unicità del MAR: ***Se i pesi degli archi sono tutti distinti, il MAR è unico.***
DIMOSTRAZIONE.
Per assurdo, **supponiamo per ipotesi** che G abbia due minimi alberi ricoprenti distinti M1 e M2. **Poiché sono distinti, esiste almeno un arco in uno dei due alberi che non appartiene anche all'altro.** Sia e l'arco di peso minimo che appartiene solo ad uno dei due MAR ma non all'altro. Supponiamo che **l'arco e** appartenga a M1.
Allora, se si aggiunge **l'arco e** a M2, si genera un ciclo C. Questo perché M2 è un albero ricoprente, quindi è connesso e contiene tutti i nodi del grafo.
Consideriamo il ciclo C. Poiché M1 non contiene cicli, nel ciclo c'è almeno **un arco e'** che non appartiene a M1. Tale arco ha peso maggiore **dell'arco e**, perché abbiamo scelto **l'arco e** come arco di peso minimo che appartiene ad uno dei due alberi ma non all'altro. Togliendo **l'arco e'** dal ciclo (e quindi da M2 modificato) si torna ad avere un albero M2' diverso da M2 perché contiene **l'arco e, e non l'arco e'.** Si tratta di un albero ricoprente di G perché il ciclo non esiste più, M2' ricopre il grafo ed è connesso (perché l'arco e' faceva parte di un ciclo). **Ma M2' ha peso minore di M2 (poiché l'arco e' è stato sostituito con un arco di peso minore), fatto che contraddice l'ipotesi che M2 sia un albero ricoprente minimo di G.**

ALGORITMO DI PRIM

Un albero ricoprente è un possibile albero di visita, quindi per trovare un minimo albero ricoprente si può adattare un algoritmo di visita.
Questo problema è un problema di minimo (vogliamo trovare un algoritmo di visita tale che sia minima una caratteristica dell'albero di visita).

IDEA: anche Dijkstra affronta un problema di minimo, possiamo usarlo?
La risposta è no. L'albero dei cammini minimi che viene generato dall'algoritmo di Dijkstra in generale non è un MAR perché nell'albero dei cammini minimi sono minimi i cammini dal nodo di partenza a ciascun nodo, mentre nel MAR deve essere minima la somma di tutti gli archi dell'albero. Tuttavia, possiamo partire da Dijkstra per costruire un algoritmo.

Nel caso dei MAR, poiché si cerca la somma minima dei cammini e non i cammini minimi ai singoli nodi, si prende ogni volta il cammino minimo non dalla sorgente ma da un qualsiasi nodo nero. **Cioè in prim si cerca ogni volta non il nodo più vicino alla radice, ma il nodo più vicino all'albero già costruito.** Quindi l'appetibilità (di un nodo grigio) è rappresentata dalla stima della distanza del nodo dall'albero di visita. Una volta aggiunto un vertice, però, (come in Dijkstra) l'appetibilità dei nodi grigi va aggiornata.

DIFFERENZA TRA DIJKSTRA E PRIM

In Dijkstra, possiamo sfruttare la proprietà per cui d[v], con v nero, è la distanza di un vertice v dalla sorgente. Essendo la distanza, non esisterà nessun cammino il cui peso sarà minore di d[v], e quindi d[v] non verrà mai più modificato.
FUNZIONAMENTO PRIM: In Prim, questa proprietà non vale: d[v] è il peso dell'arco minimo tra v e l'albero di visita già costruito nel momento in cui il vertice v viene estratto, ma questo non vieta che in un secondo momento (aggiungendo un vertice u all'albero) possa esistere un arco da un vertice nero (u) al vertice v di peso minore di d[v].

AGGIORNAMENTO DELL'APPETTIBILITA DI PRIM → d[v] = W(u, v)

DIMOSTRAZIONE DI PRIM

Sia S l'albero di visita costruito. S conterrà i nodi definitivi e V-S contiene i nodi non definitivi.
Definiamo gli invarianti:
IS) Tutti gli archi dell'albero S appartengono ad un qualche minimo albero ricoprente dell'intero grafo G.
ID) Per ogni nodo x non definitivo (in D), d[x] è il peso dell'arco (più) leggero che collega x a un nodo nero. Se x non è adiacente ad un nodo nero, d[x] = ∞.

Caso base
Dopo la prima iterazione c'è un solo nodo nero/definitivo, ovvero la sorgente e l'albero S non contiene nessun arco.
Ogni nodo x adiacente alla sorgente ha distanza d[x] uguale al peso dell'arco (s, x) ogni altro nodo ha distanza uguale a ∞.
Allora:
L'invariante IS è soddisfatto (in S non ci sono archi)
L'invariante ID è soddisfatto, perché essendoci un solo nodo nero, l'arco che connette s ad un nodo adiacente x è l'unico che unisce x a un nodo nero (quindi è il minimo), e gli altri nodi, non adiacenti ad s, hanno correttamente d = ∞.

Passo induttivo
Supponiamo che IS e ID siano veri per ipotesi Induttiva. Scegliamo (opportunamente) un taglio che separi i nodi neri dagli altri. Sicuramente non taglia nessun arco di S. Dopodiché scegliamo un arco di peso minimo che colleghi un nodo in S (quindi definitivo) con un nodo non appartenente in S. Tale arco attraversa il taglio e per il lemma del taglio tale arco appartiene ad un MAR del grafo G che estende S. Quindi aggiungendo l'arco i due invarianti IS e ID continuano ad essere veri.

Dalla dimostrazione, abbiamo che IS è un invariante di ciclo. Siccome G è connesso, alla fine della visita S conterrà tutti i nodi (che saranno neri). Quindi, alla fine della visita S è un Minimo Albero Ricoprente di G.

ALGORITMO DI KRUSCAL

Idea: Un MAR è un albero libero, quindi un grafo aciclico e connesso, che contiene tutti i vertici del grafo di partenza G e con somma dei pesi degli archi minima.
Considerando la definizione di MAR e la struttura generica di un algoritmo greedy (con appetibilità fissa), possiamo definire un algoritmo che costruisce iterativamente un MAR, a partire da una foresta vuota alla quale si aggiungono via via degli archi con i seguenti criteri:
Appetibilità: l'appetibilità di un arco è inversamente proporzionale al suo peso (il MAR ha peso totale minimo).
Criterio di ammissibilità: posso aggiungere un arco alla soluzione provvisoria solo se questo non crea cicli (il MAR è aciclico).
Criterio per riconoscere una soluzione: l'albero finora costruito è una soluzione se e solo se è connesso (il MAR è connesso).

CONTROLLO DELLA PRESENZA DI CICLI DURANTE L'ESECUZIONE DELL'ALGORITMO

Controllare sempre l'esistenza di cicli nella soluzione è fortemente inefficiente.
IDEA: usiamo una UnionFind per identificare gli insiemi di vertici che già appartengono ad uno stesso albero A.
Inizialmente, la UnionFind conterrà degli insiemi unitari (a volte chiamati singoletti) contenenti i vertici di G.
Ogni volta che sto per aggiungere un arco (u, v), considerò find(u) e find(v). Se sono uguali, u e v appartengono allo stesso albero e (u, v) causerebbe un ciclo in A e quindi questo arco (u, v) non va aggiunto ad A.
Altrimenti, posso aggiungere (u, v) ad A e fare la union dell'insieme contenente u e quello contenente v.

FUNZIONAMENTO ALGORITMO DI KRUSCAL

1) L'algoritmo ordina gli archi del grafo G in ordine non decrescente di peso.
2) Scorrendo gli archi, li aggiunge al MAR solo se essi non creano cicli. Tale operazione viene ripetuta fino a che non si ottenuto un grafo connesso e aciclico.

DIMOSTRAZIONE ALGORITMO DI KRUSCAL

Definiamo un invariante: ***Gli archi in un albero libero A (che rappresenta un certo MAR) definiscono una foresta che è un sottoinsieme di un certo MAR.***

CASO BASE
L'invariante è vero all'istante iniziale, quando A non contiene nessun arco.

PASSO INDUTTIVO
Si sceglie (u, v) come l'arco più leggero non ancora considerato.
Dobbiamo considerare due casi:

Caso 1: u e v appartengono allo stesso albero. L'aggiunta di (u, v) ad A creerebbe un ciclo, quindi l'arco (u, v) viene correttamente scartato poiché u e v sono già connessi.

Caso 2: u e v appartengono a due alberi distinti T1 e T2. Consideriamo un taglio che non tagli nessun arco di A, e che abbia T1 e T2 da parti opposte del taglio. Quindi l'arco (u, v) attraversa tale taglio. L'arco (u, v), essendo quello di peso minimo fra tutti gli archi del grafo che connettono due alberi distinti, è anche di peso minimo fra tutti gli archi che attraversano il taglio. Quindi, per il lemma del taglio, sappiamo che (u, v) ∪ A è un sottoinsieme di un MAR. Quindi aggiungendo (u, v) ad A manteniamo l'invariante.

Abbiamo quindi dimostrato che il corpo del ciclo mantiene l'invariante.
Quindi all'uscita del ciclo:
- A è un sottoinsieme di un MAR.
- È connesso.
- Contiene tutti i nodi del grafo G.

Al termine dell'esecuzione, A è un MAR del grafo G.

TECNICHE ALGORITMICHE: PROGRAMMAZIONE DINAMICA

La programmazione dinamica si applica in problemi in cui è possibile dividere il problema iniziale in sottoproblemi (come per la tecnica Divide et Impera e la tecnica Greedy). È necessario che **il problema goda della proprietà della sottostruttura ottima** (come per Greedy). In particolare, è utile applicare la programmazione dinamica quando un sottoproblema viene risolto più volte (rendendo inefficiente l'approccio Divide et Impera). Per farlo dobbiamo riscrivere la funzione in una forma ricorsiva (permesso dalla sottostruttura ottimale), individuare i sottoproblemi ed una **struttura di memoizzazione** per salvare le loro soluzioni.
In più, possiamo e dobbiamo adottare **un approccio di tipo Bottom-up** ovvero si risolve il più piccolo sottoproblema fino a risolvere l'ennesimo sottoproblema.

PROGRAMMAZIONE DINAMICA: LONGEST COMMON SUBSEQUENCE (LCS)

Data una sequenza S: a_1, ..., a_m
Una sottosequenza di S è una qualsiasi sequenza ottenuta da S togliendo alcuni (o nessun) elementi.
La sottosequenza deve rispettare l'ordine degli elementi della sequenza originale. Quindi a_1, a_2, a_{10}, a_m è una sottosequenza di S mentre a_2, a_1, a_5 non lo è, perché a_1 e a_2 non sono nel giusto ordine. S stessa e la sequenza vuota sono sottosequenze di S.

PROBLEMA

Date due sequenze S1 ed S2, trovare la più lunga sequenza S3 che è sottosequenza sia di S1 sia di S2, cioè trovare una sottosequenza comune di lunghezza massima.
Notazione: S3 = lcs(S1, S2)
Se vi sono più sottosequenze comuni di lunghezza massima, trovarne una, non importa quale.
Esempi di applicazione del problema:
1)Biologia: trovare la più lunga sottosequenza comune a due sequenze di DNA
2)È alla base di **diff**, il software (Unix) per trovare le differenze tra file

Esempio: Date le due sequenze di DNA:

AGCC**G**GA**T**CG**A**GT
TC**AG**TA**CGT**T**A**

una sottosequenza comune di lunghezza massima è:

AGCGTA

Un'altra sottosequenza comune di lunghezza massima, per le stesse due sequenze, è:

AGTCGA

Infatti:

AGCCGGA**TCGA**GT
TC**AGT**A**CG**TT**A**

SOTTOSTRUTTTURA OTTIMA DEL PROBLEMA
Ragioniamo sul problema. Siano
1)S1: a_1, ..., a_m
2)S2: b_1, ..., b_n
3)S3: c_1, ..., c_k sia **S3 = lcs(S1, S2)** (S3 è la sottosequenza risultante)
E ragioniamo sugli ultimi elementi di ciascuna sequenza. Abbiamo **2 casi** possibili:
Caso 1: $\mathbf{a_m = b_n}$
Caso 2: $\mathbf{a_m \neq b_n}$
Analizziamo i 2 casi.

Caso 1: $a_m == b_n$
Ad esempio, S1=BAC**A** e S2=ATCB**A**
I confronti si iniziano dal fondo. Quindi siccome A e contenuta sia in S1 sia in S2 sarà contenuta in S3.
In questo caso, è abbastanza facile capire che **a_m** (o l'equivalente **b_n**) **sarà contenuto in S3**, e **occuperà l'ultima posizione** di S3 (non ci possono essere elementi comuni dopo l'ultimo elemento di ciascuna sequenza). Quindi: **$c_k = a_m = b_n$**
In più, poiché gli ultimi elementi delle sequenze iniziali già appartengono alla lcs(cioè li abbiamo già «accoppiati»), possiamo **non considerarli per il confronto con il resto delle sequenze**. Quindi possiamo dire che: ***S3 = lcs(S1-{a_m}, S2-{b_n}) + {a_m}***
o meglio (usando i prefissi dove S_{l-1} sono i primi l-1 elementi di S)
S3 = lcs($S1_{m-1}$, $S2_{n-1}$) + {a_m} oppure **S3 = lcs($S1_{m-1}$, $S2_{n-1}$) + {b_n}** poiché **a_m==b_n**

Caso 2: a_m != b_n
Ad esempio, S1=BAC**A** e S2=ATCB**AB**
In questo caso, possiamo dire che a_m, o b_n (o entrambi) non saranno contenuti in S3.
Consideriamo **il caso in cui sia a_m a non essere contenuta in S3**. Possiamo «toglierla» dal confronto, e dire che: ***S3 = lcs(S1-{a_m}, S2)***
Il caso in cui sia b_n a non essere incluso è (speculare e) dato da: ***S3 = lcs(S1, S2-{b_n})***
(il caso in cui entrambi non siano compresi può essere ottenuto da un passaggio ulteriore di questo ragionamento, quindi non lo consideriamo per ora).
Ma come faccio a sapere quale delle due versioni di S3 usare?
Semplice. Sto cercando la LCS, quindi S3 è la più lunga tra
lcs(S1-{a_m}, S2) e lcs(S1, S2-{b_n}) quindi è ***MAX(lcs(S1-{a_m}, S2), lcs(S1, S2-{b_n}))***

Quindi, il problema della LCS gode della proprietà della **sottostruttura ottima**. Abbiamo visto come, **per calcolare lcs(S1, S2)**, ci basti conoscere le soluzioni dei sottoproblemi **lcs($S1_{m-1}$, $S2_{n-1}$), lcs($S1_{m-1}$, S2), lcs(S1, $S2_{n-1}$)** e a_m e b_n.
Possiamo quindi applicare un algoritmo di **programmazione dinamica** per risolvere il problema della LCS.

Definiamo la struttura per la memoizzazione

Usiamo una **matrice**, chiamandola **LCS**, di **m+1 righe e n+1 colonne**. La casella **LCS[i, j] contiene la più lunga sottosequenza comune dei due segmenti iniziali** di lunghezze rispettive i e j, cioè **a[0 .. i-1]** e **b[0 .. j-1]**, cioè: **LCS[i, j] = lcs(a[0 .. i-1], b[0 .. j-1]).**
Perché il +1 in righe e colonne?
La **casella 0-esima** contiene la **più lunga sottosequenza dei prefissi vuoti**(con i e/o j= 0).

Inizializziamo la matrice LCS

Partiamo dai casi base. Se una delle due (sotto)sequenze è **vuota**, la **lcs** delle due **è** ovviamente **vuota**. Quindi
LCS[0, j] = [] per 0<=j<=n
LCS[i, 0] = [] per 0<=i<=m

Popolamento della matrice LCS

Supponendo di aver valorizzato **LCS[i, j-1], LCS[i-1, j] e LCS[i-1, j-1]** dobbiamo **valorizzare LCS[i, j]**.
Dalla definizione della sottostruttura ottimale, abbiamo che
$a_m = b_n \Rightarrow lcs(S1, S2) = lcs(S1_{m-1}, S2_{n-1}) + a_m$
che diventa
if a[i-1] == b[j-1] then
LCS[i, j] <-LCS[i-1, j-1] + a[i-1]
e
$a_m != b_n \Rightarrow lcs(S1, S2) = max\_lunghezza(lcs(S1_{m-1}, S2), lcs(S1, S2_{n-1}))$
che diventa
if a[i-1] != b[j-1]then
LCS[i, j] <-max_lunghezza(LCS[i-1, j], LCS[i, j-1])

Algoritmo lcs–programmazione dinamica

```
lcs(a, b, m, n){
        LCS <-nuova matrice di dimensioni m+1 x n+1
        for i = 0..m LCS[i, 0] <-[] //inizializzazione casi base
        for j = 0..n  LCS[0, j] <-[] //inizializzazione casi base
        for i = 1..m
                for j = 1..n
                        If(a[i-1] == b[j-1]) then
                                LCS[i, j] <-LCS[i-1, j-1] + a[i-1]
                        else
                                LCS[i, j] <-max_lunghezza(LCS[i-1, j], LCS[i, j-1])
        return LCS[m, n]
}
```

PROGRAMMAZIONE DINAMICA:
CAMMINI MINIMI E ALGORITMO DI BELLMAN FORD

DUE PROPRIETA IMPORTANTI

1)Disuguaglianza Triangolare delle Distanze
Le distanze in un grafo rispettano la disuguaglianza triangolare.
Per ogni tripla di vertici s, u, v **$\delta(s, v) \leq \delta(s, u) + \delta(u, v)$**
DIMOSTRAZIONE
Un cammino minimo s $\rightsquigarrow$v ha peso (**$\delta(s, v)$**)**minore o uguale di qualsiasi altro cammino tra s e v**, quindi anche della concatenazione dei cammini s $\rightsquigarrow$u e u $\rightsquigarrow$v.

2)Condizione di Bellman: Per ogni arco (u, v) e per ogni vertice s
$\delta(s, v) \leq \delta(s, u) + W(u, v)$
DIMOSTRAZIONE: Per definizione **$\delta(u, v) \leq W(u, v)$**, quindi sostituendo δ(u, v) con W(u,v) nella disuguaglianza triangolare, la disuguaglianza continua a valere.

APPARTENENZA AD UN CAMMINO MINIMO

Dalla condizione di Bellman possiamo derivare il seguente lemma.
Lemma 1: Un arco **(u, v) appartiene ad un cammino minimo** da s a v **se e solo se u è raggiungibile da s** e la **condizione di Bellman** è soddisfatta con **l'uguaglianza** per l'arco (u, v), cioè se vale: **$\delta(s, v) = \delta(s, u) + W(u,v)$**

TECNICA DEL RILASSAMENTO DELL'ARCO
Partiamo assegnando ad ogni vertice una sovrastima D della sua distanza dalla radice, di modo da ottenere D[s, v] >= δ (s, v). Supponiamo ora che esista un arco <u, v> tale per cui D[s, v] > D[s, u] + W(u, v). D[s, u] e W(u, v) non sono minori di δ(s, u) e δ(u, v). Per questo D[s, v] non può essere certamente uguale a δ(s, v). Per avvicinarla di valore, però, possiamo assegnarle il valore di D[s, u] + W(u, v).
D[s, v] = D[s, u] + W(u, v)
Questa tecnica si chiama rilassamento dell'arco.
Eseguendo più volte questa operazione si arriva al punto in cui D per ogni vertice corrisponde alla sua distanza. Se noi partiamo assegnando a D per ogni vertice il valore infinito, sapremo per certo che sicuramente è maggiore o uguale alla sua distanza dalla radice.

A differenza dell'algoritmo di Dijkstra, con questo algoritmo si può ottenere il cammino minimo anche in grafi con archi negativi.

PROPRIETÀ: Un sottocammino di un cammino minimo è un cammino minimo.
Proviamo ad applicare la tecnica della Programmazione Dinamica.

COSTRUIAMO UN ALGORITMO DI PROGRAMMAZIONE DINAMICA

Sia $\langle s, v_1, v_2, \dots v_k \rangle$ un cammino minimo (di k elementi) da s a v_k. Grazie alla condizione di Bellman, possiamo scrivere $\delta(s, v_k) = \delta(s, v_{k-1}) + W(v_{k-1}, v_k)$ (Lemma1)

Ma allora possiamo ricondurre il problema di trovare $\delta(s, v_k)$ a quello di trovare $\delta(s, v_{k-1})$. Il problema però, è che noi non sappiamo quale vertice sarà v_{k-1} nel nostro cammino minimo. Come fare?

Sappiamo però che il cammino (minimo) $\langle s, v_1, v_2, \dots v_k \rangle$ avrà lunghezza k-1, quindi il sottoproblema di trovare $\langle s, v_1, v_2, \dots v_{k-1} \rangle$ dovrà considerare solo cammini di lunghezza k-1. A sua volta un sottoproblema da trovare $\langle s, v_1, v_2, \dots v_{k-2} \rangle$ considererà cammini di lunghezza k-2 e così via fino a k = 0.
Ora ragioniamo Bottom-up.

Con **k = 0**, il problema è banale, si tratta di trovare il cammino minimo da s a s con 0 archi. La soluzione è banalmente $\langle s \rangle$.
Usiamo le stime **D(s, v)** come struttura di memoizzazione, ed inizializziamo **D(s, s) = 0** e **D(s, v) = ∞** per gli altri vertici v nel grafo, rispettando la condizione per cui **D(s, v) ≥ δ(s, v)**. Banalmente, a questo punto **D(s, s) = 0 = δ(s, s)**.

Ora, per **k = 1**, vogliamo trovare il cammino minimo tra s e v_1, che contenga un solo arco (ricordate che però **non conosciamo v_1**).
Applicando il **rilassamento su tutti gli archi una volta sola**, tra i vari rilassamenti fatti, sicuramente troveremo ***∞ = D(s, v_1) > D(s,s) + W(s,v_1).*** Poiché $W(s, v_1) < \infty$.
Il rilassamento allora porrà **D(s, v1) = D(s,s) + W(s,v1)**.
Ma **D(s,s) = δ(s, s)** e **⟨s, v_1⟩ è un cammino minimo** (per la proprietà dei sottocammini minimi), quindi **D(s, v_1) = δ(s, s) + W(s, v_1) =δ(s, v_1).**
Notate che, essendo la lunghezza di un cammino minimo **nessun rilassamento successivo potrà aggiornarla**.

Al passo successivo, cerchiamo **$\delta(s, v_2)$**.Applichiamo di nuovo il rilassamento su tutti gli archi. Tra i vari rilassamenti, troveremo **$D(s, v_2) > D(s, v_1) + W(v_1, v_2)$**
Poiché **$D(s, v_1) = \delta(s, v_1)$** e $\langle s, v_1, v_2 \rangle$ è un cammino minimo.
Il rilassamento allora porrà **$D(s, v_2) = D(s, v_1) + W(v_1, v_2)$**. Anche qui possiamo dire che **$D(s, v_2) = \delta(s, v_2)$** e $D(s, v_2)$ non sarà più aggiornato.
Iterando il processo, al k-esimo passo avremo **$D(s, v_k) = \delta(s, v_k)$**. Ciò sarà vero per ogni vertice v_k raggiungibile da un cammino minimo di al più k archi (dopo lo dimostreremo, per ora vi basta capire il funzionamento).

Iterando il processo, al k-esimo passo avremo **$D(s, v_k) = \delta(s, v_k)$**. Ciò sarà vero per ogni vertice v_k raggiungibile da un cammino minimo di al più k archi. Ma noi non sappiamo neanche qual è la lunghezza k del cammino minimo! Tuttavia, un cammino minimo semplice (cioè senza vertici duplicati) può contenere al massimo tutti i vertici del grafo (n), quindi n-1 archi. Di conseguenza, bastano n-1 iterazioni per trovare i cammini minimi da s a tutti i vertici del grafo.

DIMOSTRAZIONE BELLMAN-FORD

Sia **$\delta(s, v_k)$**la distanza (peso del cammino minimo) da s a v_k.
Definiamo l'invariante **KPATH**:
Dopo la k-esima iterazione del ciclo esterno (for i = 1..n-1) si ha **d[vk] = δ(s, vk)**per ogni nodo v_k per cui il **cammino minimo da s a v_k** è composto al più da **k archi**.

DIMOSTRAZIONE. (per induzione)
BASE: Per **k = 0 (prima del ciclo)**, per il cammino da s ad s (l'unico cammino composto da 0 archi) **d[s] = 0 e d[u] = ∞** per tutti gli altri nodi u.
PASSO: Supponendo che KPATH sia vera dopo k iterazioni, dimostriamo che rimane vera anche dopo k+1 iterazioni.

Sia **v_{k+1}** un nodo il cui cammino minimo **$s \rightsquigarrow v_k \rightsquigarrow v_{k+1}$ è composto da k+1 archi**. Sappiamo che **$\delta(s, v_{k+1}) = \delta(s, v_k) + W(v_k, v_{k+1})$** per il Lemma 1 e **$d[v_k] = \delta(s, v_k)$** per ipotesi induttiva. Quindi possiamo dire che **$\delta(s, v_{k+1}) = d[v_k] + W(v_k, v_{k+1})$**
Al passo k+1, ci sono 2 casi:
Caso 1: **$d[v_{k+1}]$ viene aggiornata**,
Allora ***$d[v_{k+1}] = d[v_k] + W(v_k, v_{k+1}) = \delta(s, v_k) + W(v_k, v_{k+1}) = \delta(s, v_{k+1})$***

Caso 2: **d[v_{k+1}] non viene aggiornata**, poiché d[v_{k+1}] è il peso di un cammino da s a vk+1in G, **d[v_{k+1}] ≥ δ(s, v_{k+1})** poiché non è stata aggiornata,
d[v_{k+1}] ≤ d[v_k] + W(v_k,v_{k+1}) = **δ(s, v_{k+1})**, possiamo avere solo **d[v_{k+1}] = δ(s, v_{k+1})**quindi **il cammino trovato è comunque minimo**.
Quindi, dopo la k-esima iterazione, **KPATH vale** e **d[v_k] = δ(s, v_k)** per ogni nodo v_k per cui il **cammino minimo da s a v_k, che è sicuramente un cammino semplice, è composto al più da k archi**.
Ma in un grafo, un cammino minimo può contenere al più tutti i nodi appartenenti a V, quindi un cammino minimo non può contenere più di **n-1 archi**.
Allora, dopo n-1 iterazioni d[u] = δ(s, u) per ogni nodo u. Quindi i cammini restituiti dall'algoritmo sono effettivamente quelli minimi.

COMPLESSITA

L'algoritmo fa **n-1 iterazioni del ciclo esterno**. Per ogni iterazione, è facile notare che considera **tutti gli archi del grafo** con una serie di operazioni di costo O(1). Quindi la complessità dell'algoritmo di Bellman-Ford è **O((n-1)*m) = *O(nm)***
(la complessità è maggiore di Dijkstra, quindi nel caso di grafi con archi di peso solo positivo Dijkstra è più efficiente).
Domanda: perché Dijkstra «batte» Bellman-Ford? In questo caso, perciò, con solo archi positivi Dijkstra batte Bellman-Ford poiché esegue meno operazioni di rilassamento. Questo perché l'algoritmo di Dijkstra sceglie un arco garantendo che quell'arco sia tra tutti gli archi nei quali un vertice è stato raggiunto quello che minimizza un cammino per raggiungere gli altri vertici del grafo.

OTTIMIZZAZIONE DI BELLMAN-FORD PER I DAG

Dato un ordinamento topologico, in cui un vertice u precede un vertice v (**u < v**), allora **in qualsiasi cammino che contenga sia u che v (compresi i cammini minimi)** su quel grafo, **u precederà v**.
Questa proprietà ci suggerisce che se rilassiamo gli archi in maniera che **D(s, u) sia uguale a δ(s, u) prima che l'arco (u, v) sia considerato** per il rilassamento, non effettueremo rilassamenti inutili (e quindi da ripetere).
Allora, **se un grafo è un DAG**, possiamo prima calcolare un **ordinamento topologico**, e poi **rilassare** tutti **gli archi** una **sola volta seguendo l'ordine topologico**.

TEORIA DELLA COMPLESSITA

La **Teoria della Complessità** è la branca dell'informatica che studia la complessità dei **problemi**.
La **complessità** di un **problema** (non di un algoritmo!) è la complessità del «**migliore**» **algoritmo che lo risolve**, sia che questo algoritmo sia noto, sia che esso sia ipotizzato.

TIPI DI PROBLEMI

Problemi di decisione: problemi che richiedono di verificare una certa proprietà sull'input. Sono problemi per cui S = {0, 1}. Ad esempio, dato un grafo G dire se è connesso.

Nota bene→ Si vogliono caratterizzare i problemi in base alle risorse di calcolo richieste per risolverli. Una classe di complessità è un insieme di problemi che possono essere risolti usando le stesse risorse di calcolo.

CLASSI DI COMPLESSITA

CLASSE P: È la classe dei problemi risolvibili in tempo polinomiale rispetto a n.

CLASSE PSPACE: È la classe dei problemi risolvibili in spazio polinomiale rispetto a n.

Notate che potendo fare al più un numero polinomiale di operazioni, e quindi anche di accessi in memoria, $\boldsymbol{P \subseteq PSPACE}$.

CLASSE EXPTIME: È la classe dei problemi risolvibili in tempo esponenziale rispetto a n. $\boldsymbol{PSPACE \subseteq EXPTIME}$.

GERARCHIA
L'unica inclusione propria dimostrata è $\boldsymbol{P \subset EXPTIME}$ (cioè esiste almeno un problema che non può essere risolto in tempo polinomiale).
Capire se $P \subset PSPACE$ e $PSPACE \subset EXPTIME$ sono problemi aperti
(ma si pensa che sia così).

Certificati: Un **certificato** è un oggetto y, dipendente dall'istanza x e dal problema P, che possa giustificare $x,1 \in P$ (per problemi **decisionali**).
Ad esempio, nel problema della soddisfacibilità, l'assegnazione $\boldsymbol{z1}$, $\boldsymbol{z2}$, …, $\boldsymbol{zn}$ è un certificato. Per il problema di dire se un grafo è connesso, un **albero ricoprente** è un certificato.

Possiamo ora riscrivere i nostri algoritmi in due fasi: una di **costruzione** ed una di **verifica.**

NON DETERMINISTICO

Definiamo una funzione **indovina z** $\in \{\mathbf{0},\mathbf{1}\}$ che può indovinare (in tempo costante) un **giusto** valore booleano. (a volte la funzione indovina è chiamata **oracolo**).

La funzione **indovina** ovviamente **non esiste**. Tuttavia, ci permette di introdurre una nuova classe di algoritmi (e di problemi)

Chiamiamo gli algoritmi sviluppati con il suo uso **non deterministici**, perché la loro computazione non è deterministica (cioè **il passo successivo della computazione non è determinato solo dallo stato della computazione stessa**).

Solo la fase di costruzione fa uso della funzione indovina, mentre **la fase di verifica è deterministica.**

CLASSE NP: È la classe di **problemi risolvibili** in **tempo polinomiale** da un algoritmo **non deterministico.** O, equivalentemente, NP è la **classe** di **problemi** per cui la fase di **verifica** è fatta (**deterministicamente**) in **tempo polinomiale**.

Ovviamente, un algoritmo deterministico è un caso particolare di uno non deterministico, quindi $P \subseteq NP$. Inoltre, affinché la verifica sia fatta (deterministicamente) in tempo polinomiale, il certificato deve occupare spazio polinomiale, quindi $NP \subseteq PSPACE$.

GERARCHIA

Come per la gerarchia vista precedentemente, non è noto, ma si sospetta, che $\boldsymbol{P \subset NP}$ e $\boldsymbol{NP \subset PSPACE}$**.**

$NP \subset PSPACE$**?**

Introduciamo un problema appartenente a **PSPACE** ma che si sospetta non appartenere a NP.

Formula Booleana Quantificata: è un'espressione in forma normale congiuntiva preceduta da una serie di quantificatori universali ∀ed esistenziali ∃che legano tutte le variabili.

Data tale formula, il problema delle formule booleane quantificate richiede di verificare se essa è **vera**.

$P \subset NP$ **o** $P = NP$**?**

Il confine tra P ed NP è quello più importante in teoria della complessità. Infatti, è quello che separa i problemi «trattabili» da quelli «non trattabili» (almeno con facilità).

Cerchiamo di capire come arrivare ad una conclusione per questa domanda.

Prima di tutto cerchiamo un modo per **confrontare i problemi** tra di loro.

RIDUCIBILITA POLINOMIALE

Diciamo che un problema (decisionale) $\boldsymbol{P1} \subseteq \boldsymbol{I1} \times \{\boldsymbol{0},\boldsymbol{1}\}$ **è riconducibile polinomialmente ad un problema**(decisionale) $\boldsymbol{P2} \subseteq \boldsymbol{I2} \times \{\boldsymbol{0},\boldsymbol{1}\}$ se e solo se:

•Esiste una funzione che, data un'istanza $x1 \in I1$di P1, la **trasforma** in un'istanza $x2 \in I2$di P2 usando **tempo("abbastanza") polinomiale**

•Per ogni coppia di istanze x1 e x2 così costruite, **la soluzione di x1 è la stessa soluzione di x2**

PROPRIETA': Se P2 appartiene alla classe P, anche P1 appartiene a P.

DIMOSTRAZIONE: la definizione di riducibilità polinomiale suggerisce un algoritmo polinomiale per P1. Basta risolvere ogni istanza x1 di P1 trasformandola (polinomialmente) in x2 e risolvendo x2 con il rispettivo algoritmo per P2 di complessità polinomiale.

Riducibilità Polinomiale - esempio

Problema del Cammino (lungo) semplice (P2): Dato un grafo G, due vertici s e t ed una lunghezza k, verificare se esiste un cammino semplice in G da s a t di lunghezza almeno k.

Problema del Ciclo Hamiltoniano (P1): Dato un grafo G, verificare se esiste un **ciclo che visita ciascun vertice una volta sola**.

È facile notare che per risolvere il Problema del Ciclo Hamiltoniano basta trovare un arco (u,v) tale che esista tra i due vertici un cammino semplice di lunghezza k = n-1.

Banalmente, dobbiamo fare questa ricerca per ogni arco (u,v), quindi la complessità della trasformazione è O(m), che è polinomiale.

Allora il Problema del Ciclo Hamiltoniano (P1) è riconducibile polinomialmente al Problema del Cammino semplice (P2).

$P \subset NP$ o $P=NP$?

Possiamo definire le seguenti classi.

Problemi NP-hard: un problema decisionale D si definisce NP-hard (o NP-arduo) se **ogni problema**$\boldsymbol{Q} \in \boldsymbol{NP}$è **riconducibile** a D in **tempo polinomiale**.

Esistono problemi NP-hard sia appartenenti ad NP che a classi superiori.

Problemi NP-completi: un problema decisionale D si definisce NP-completo se è **NP-hard ed appartiene alla classe NP**.

I problemi NP-completi sono quelli di maggiore importanza: **basterebbe dimostrare che uno di essi appartiene a P per poter dire che** $\boldsymbol{P}=\boldsymbol{NP}$. Tuttavia, ciò non è mai stato dimostrato e si sospetta che ciò non sia nemmeno possibile.

PROBLEMI NP-COMPLETI?
Si esistono ma non sono argomenti principali del corso.
HALT LIMITATO: dato un **programma X** ed un **intero k**, verificare se **esiste un input per cui X termina al massimo in k passi**.
Appartiene ad **NP** perché la fase di verifica (usando come certificato i) può essere effettuata in tempo **polinomiale** simulando X(i) per k passi.
Ora dimostriamo che **ogni problema $Prob \in NP$ è riducibile polinomialmente ad halt limitato (cioè L-HALT è NP-hard)**.
Sia $Prob \in NP$, allora **esiste un programma C** che verifica un certificato in tempo polinomiale **p(n)**.
Allora è possibile costruire un **programma C'** che va in **loop** ogni qualvolta la risposta di **C** sia **negativa**, **termina altrimenti**.

Ma allora possiamo riscrivere Prob come il problema dell'halt limitato, con C' come programma X e p(n) come k (il numero di passi). L'halt limitato restituirà **1 se esistono dei dati (un certificato) per cui C' termina in p(n) passi**, **0 altrimenti** (quindi, quando non esiste una soluzione).
Ma se $L{-}HALT(C', p(n)) = 1$ vuol dire che Prob ha soluzione (s = 1), e se $L{-}HALT(C', p(n)) = 0$ Prob non ha soluzione (s = 0)!
Quindi **Prob è riconducibile polinomialmente al problema dell'halt limitato**. Quindi il problema dell'halt limitato è **NP-hard.** Essendo sia NP che NP-hard, è **NP-completo.**

PROBLEMI INDECIDIBILI
Concludiamo questo discorso con un'osservazione. EXPTIME non è la classe più «difficile». Esistono infatti una serie di problemi definiti **Indecidibili**: non solo **non si conoscono algoritmi per risolverli**, ma è stato dimostrato che tali algoritmi **non possono esistere**, nemmeno utilizzando tempo e spazio infiniti.

Il problema dell'halt è un problema indecidibile.
Definire un programma **halt(P, i)**che restituisca 1 se il programma Prog(con un certo input i) termina in un numero finito di passi, 0 altrimenti.
DIMOSTRAZIONE: Se potessimo scrivere halt, potremmo scrivere il programma g come

```
g(Prog)
  if (halt(Prog, Prog))
    loop
  else
    return 0
```

(è possibile dare in input ad un algoritmo se stesso)
A questo punto, quale sarebbe l'output di g(g)? **g(g) termina se e solo se g(g) non termina**, e questa è una **contraddizione**.

ALGORITMI DI APPROSSIMAZIONE

È sempre possibile **associare** ad un **problema** di **ottimizzazione** un **problema decisionale**, che ci dica se una determinata soluzione per un problema è ottima.

È facile capire che **il relativo problema decisionale** è **più facile** del **problema di ottimizzazione**.

Ma se già il problema decisionale è NP-completo, sembra davvero difficile trovare algoritmi efficienti per risolvere il problema di ottimizzazione.

A volte, **privilegiando l'efficienza a costo dell'ottimalità**, si utilizzano **algoritmi di approssimazione** tali per cui la soluzione restituita non è per forza ottima, ma ha costo C che si discosta dal costo C* di quella ottima di al massimo un **fattore di approssimazione** moltiplicativo $\boldsymbol{\rho(n)}$.

Copertura (minima) di Vertici

Vediamo un esempio.

Dato un grafo non orientato G = (V, E), una **copertura di vertici** è un sottoinsieme $\mathbf{V' \subseteq V}$ **tale che, se (u, v)** $\in \mathbf{E}$**, allora** $\mathbf{u \in V' \vee v \in V'}$.

La **dimensione** di una **copertura** V' è il **numero** di **vertici** che contiene.

Il problema della copertura di vertici consiste nel trovare **V' ottima**, cioè tale che la sua **dimensione** sia **minima**.

Decidere se una certa copertura V'' è ottima è un problema **NP-completo**; quindi, difficilmente possiamo risolvere il nostro problema «efficientemente».

Possiamo però trovare facilmente una soluzione «**approssimativamente** ottima».

Fattore di Approssimazione

La copertura minima di vertici ha un **fattore di approssimazione di 2**. Si dice anche che è **2-approssimato.**

DIMOSTRAZIONE: Sia A l'insieme di archi (u, v) scelti all'inizio di ogni ciclo.

Per coprire gli archi in A, una **copertura ottima** deve contenere **almeno un vertice di ciascun arco in A**.

Due archi in A non possono avere un vertice in comune, perché una volta scelto (u,v) tutti gli archi incidenti ad u o v sono rimossi da E'.

Pertanto, non ci sono due archi in A coperti dallo stesso vertice e **|A| è un limite inferiore** per la dimensione di una **copertura ottima C*. C*≥$\boldsymbol{|A|}$**

Ma $\boldsymbol{C=2|A|}$, poiché nel risultato metto sia u che v, quindi **|$\boldsymbol{C}$ |=2|$\boldsymbol{A}$|≤2|$\boldsymbol{C^*}$|**

Problema del Commesso Viaggiatore (TSP)

Il **Travel Salesman Problem** (da cui TSP) è uno dei problemi di **ottimizzazione** più famosi, con molte applicazioni pratiche.

Data una mappa (modellata con un grafo), il commesso viaggiatore deve passare per **tutte le città** in essa contenute **una sola volta**, e **tornare** al **punto di partenza** facendo il percorso **migliore**.

Il problema è uno dei più conosciuti e studiati in informatica. Ha moltissime applicazioni pratiche (es. macchina foratrice per i circuiti elettronici).

Più formalmente, dato un grafo **pesato**, **completo** e **non orientato** G = (V, E), bisogna trovare un **ciclo Hamiltoniano** (un ciclo semplice che contenga tutti i vertici) di **costo minimo**.

Noi consideriamo la versione «semplice» (ma realistica, ad esempio in una mappa), in cui i pesi degli archi rispettano la **disuguaglianza triangolare**

$W\ (u, w) \leq W(u, v)+W(v, w)$

Il problema di capire se una soluzione è ottima è esso stesso NP-completo. Di conseguenza, cerchiamo soluzioni **approssimate**.

IDEA: un **minimo albero ricoprente (MAR)** è un insieme di archi di peso minimo che tocca tutti i vertici, ma **non è un ciclo**. Tuttavia, il **peso di un MAR** è un **limite inferiore** per il **peso di un circuito Hamiltoniano**. Circumnavigando il MAR otteniamo un ciclo (**con peso doppio rispetto al MAR**), che però non è Hamiltoniano perché passa sullo stesso nodo più volte. Per trasformarlo in Hamiltoniano, possiamo «tagliare» le ripetizioni dei nodi, sfruttando la **disuguaglianza triangolare** che ci assicura che se sostituiamo (v, w), (w, z) (dove **w** è una **ripetizione** nel ciclo) con (v, z) otteniamo un ciclo di peso non maggiore.

IDEA: la circumnavigazione + il taglio delle ripetizioni, altro non sono che una visita in profondità del MAR con traccia di inizio visita! Possiamo usare una **visita in profondità dell'albero** (restituendo i vertici in **ordine crescente di inizio visita**) per passare direttamente dall'albero al circuito.

APPROX-TSP (G, W, r)

```
scegli un vertice r casualmente
A <-Prim(G, W, r)
ord<-DFS-ric-INIZIO (A, r)
return ord
```

Sia ord= [V1, V2, … Vn], il ciclo Hamiltoniano è **V1, V2, … Vn, V1**

DFS-ric-INIZIO restituisce un vettore contenente i vertici in ordine di inizio visita di una visita in profondità. L'algoritmo è **2-approssimato.**

UNION FIND

Una unionFind è una collezione di insiemi disgiunti sulla quale sono possibili le operazioni di union(a, b) ,find(x) e makeset(x) .

Costo ammortizzato con crediti

Serve ad analizzare il costo di operazioni in sequenza. Per ottenere il costo delle singole operazioni si divide il costo della sequenza per il numero di operazioni.

Con questa tecnica le operazioni meno costose depositano sullo stack dei crediti (svolgendo lavoro extra) che potranno essere sfruttati dalle operazioni più costose di modo da alleggerirle.

Un esempio è il costo di push(costo O(1)) e multipop di k elementi (costo O(n)).

Se carichiamo il costo di push otteniamo due crediti, uno consumato da lui ed uno depositato sullo stack. In questo modo con ogni push verrà depositato uno stack. Al multipop assegniamo invece il costo di zero crediti. Ciò vuol dire che ad ogni multipop verrà consumato un credito. In questo modo anche multipop avrà costo costante, la stess sequenza avrà costo costante anch'essa.

Un altro esempio è la sequenza di N makeset ed N-1 union. Ogni volta che si esegue una union, per il bilanciamento, le foglie che appartenevano all'albero più piccolo (e che quindi cambiano padre) si ritroveranno in un nuovo albero che è grande almeno il doppio. Quindi, dopo k cambi di padre, una foglia apparterrà ad un insieme di 2^k elementi. Dato n il numero totale di elementi, 2^k <= n. Ciò vuol dire che k <= $\log_2$ di n. Assumiamo che ad ogni makeset depositiamo $\log_2$ di n crediti. Ogni volta che una foglia cambia padre consuma un credito. Ma potendo ogni foglia cambiare padre al più $\log_2$ n volte, allora potranno essere consumati al più n$\log_2$ di n crediti. In questo modo il costo della sequenza diventa O(n$\log_2$n).

PROG DIN – ALGORITMO DI FLOYD WARSHALL PER I CAMMINI MINIMI

Questo algoritmo serve a trovare il cammino minimo tra tutte le coppie di vertici.

17 – Cammino minimo k – vincolato

E' il cammino minimo tra due vertici x e y tra tutti i cammini che non contengono i vertici che vanno da k a n

E' il cammino minimo in un grafo in cui non esistono i vertici da k a n e tutti gli archi che li collegano tra di loro o agli altri vertici. Anche per questo grafo risultante si potranno avere dei cammini k1 vincolati, con k1 compreso tra 1 e k. Si deduce quindi che vale la proprietà della sottostruttura ottima.

18 – Distanza k – vincolata

E' il peso del cammino minimo se esso esiste. Altrimenti è infinito.

Per ogni k ed ogni coppia di vertici x e y, definiamo la distanza k – vincolata come:

- $k\boldsymbol{d}_{xyk} = \mathbf{min}(k\boldsymbol{d}_{xy-1}, k\text{-}1\boldsymbol{d}_{xk} + k\text{-}1\boldsymbol{d}_{ky})$

Quello che si fa è vedere ogni volta se D (x, y) > D (x, k) + D (k, y)

Se D (x, y) è maggiore, si applica un rilassamento assegnandogli il valore di D (x, k) + D (k, y).

Elenco delle dimostrazioni da sapere per l'esame orale (a.a. 2020/2021)

NOTA BENE: questo **non è un elenco esaustivo** degli argomenti richiesti all'esame orale. Tuttavia, per rendere lo studio più facile, ho fatto un elenco delle possibili dimostrazioni di correttezza viste a lezione e richieste all'esame orale. Non ho elencato solo le dimostrazioni ma anche altri argomenti, per sottolinearne l'importanza. In questa lista, sono indicate in verde (carattere normale) le dimostrazioni facili, da capire e sapere per la sufficienza, in blu (carattere corsivo) le dimostrazioni di difficoltà intermedia (comunque da sapere, ma che da sole non pregiudicano la sufficienza) ed in **rosso (carattere grassetto)** le dimostrazioni più difficili (da sapere per la lode, ma comunque da aver capito).

0)Le prime due proprietà della visita in ampiezza sui livelli dei nodi in frangia e sull'ordine dei nodi in frangia

1)Visita in Ampiezza Grafi: Dimostrazione $d[v] = \delta(s,v)$

2)Ordinamento Topologico: Correttezza algoritmo astratto

3)Ordinamento Topologico: Teorema dell'ordinamento topologico

4)Componenti Fortemente Connesse: Lemma del cammino fortemente connesso

5)Componenti Fortemente Connesse: Teorema del sottoalbero fortemente connesso

6)Greedy Intervalli Disgiunti: Dimostrazione di Correttezza

7)Greedy Moore: Dimostrazione di Correttezza

8)Greedy Huffman: Dimostrazione di Correttezza

9)Dijkstra: Dimostrazione di Correttezza

10)Minimo Albero Ricoprente: Lemma del Taglio

11)Uso del Lemma del Taglio in Prim e Kruskal

12)Minimo Albero Ricoprente: Teorema dell'unicità del MAR

13)Prim: Dimostrazione di Correttezza

14)Union Find: Analisi ammortizzata con il metodo dei crediti

15)Kruskal: Dimostrazione di Correttezza

16)LCS: saper spiegare come si arriva all'algoritmo, partendo dalla sottostruttura ottima, casi e sottoproblemi

17)Bellman-Ford: Dimostrazione di Correttezza

18)Floyd-Warshall: saper spiegare come si arriva all'algoritmo, partendo dalla definizione di distanze k-vincolate

19)Perché si sospetta che $P \subset PSPACE$ ***e*** $NP \subset PSPACE$

20)Dimostrazione che il problema dell'Halt limitato è NP completo

21)Dimostrazione che il problema dell'Halt è indecidibile

22)Algoritmo Approssimato per Copertura Vertici: saper spiegare perché è un algoritmo 2-approssimato

23)Algoritmo Approssimato per TSP: saper spiegare perché è un algoritmo 2-approssimato