Appunti Algoritmi

- Insertion sort

[*Funzionalità***]**
Risolve il problema dell'ordinamento **(per confronti)** . I numeri da ordinare sono detti anche chiavi . Opera nello stesso modo in cui molte persone ordinano le carte da gioco .
[*Caratteristiche***]**
Insertion sort è un algortimo efficiente per ordinare un piccolo numero di elementi , ma è molto inefficiente per un numero grandi di elementi . Insertion è un algoritmo **IN-PLACE** e **STABILE** .

Un algoritmo si dice **in loco** oppure **in place** quando è in grado di trasformare una struttura dati utilizzando soltanto un piccolo e costante spazio dimemoria extra. I dati in ingresso sono solitamente sovrascritti con il risultato prodotto durante l'esecuzione dell'algoritmo.
Un algoritmo si dice **stable** o **stabile** quando mantiene l'ordine relativo tra chiavi equivalenti.

[*Complessità***]**
Caso migliore : O(n) Il caso migliore è quando i numeri da ordinare sono gia ordinati .
Caso peggiore : O(n^2) Il caso peggiore è quando i numeri da ordinare sono ordinati in modo decrescente .
Caso medio : risulta una funzione quadratica nella dimensione dell'input , siccome in media meta degli elementi da ordinare sono maggiore da una qualche chiave j , mentre gli altri sono più grandi , in media verifichiamo metà degli numeri .

- Merge Sort

[*Funzionalità***]**
Risolve il problema dell'ordinamento **(per confronti)** . Merge sort è un algoritmo **ricorsivo** che adotta un approccio **divide et impera** .
/***ricorsivo** : per risolvere un determinato problema , l'algoritmo chiamo se stesso in modo ricorsivo uno o piu volte . **divide et impera** : suddivide il problema in vari sottoproblemi , che sono simili al problema originale , ma di dimensioni piu piccole , risolvono i sottoproblemi in modo ricorsivo e , poi , combinano le soluzioni per creare una soluzione al problema originale .*/
L'operazione chiave dell'algoritmo è la fusione di due sottosequenze ordinate nel passo "combina" . Per effetuare la fusione si usa una procedura ausiliaria Merge(A,p,q,r) . La procedura suppone che i sottoarray sono ordinati , li fonde per formare un unico sottoarray ordinato .
[*Caratteristiche***]**
Merge sort è un algoritmo **STABILE** ma non è IN-PLACE .
[*Complessità***]**
Funzione ausiliaria Merge(A,p,q,r) : $O(n)$. (per qualsiasi input)
Algoritmo Merge sort : $\Theta(nlgn)$. (per qualsiasi input)

- Heap binario (SD)

Struttura dati composta da un array che possiamo considerare come un albero binario quasi completo (albero dove ogni nodo se ha figlio destro deve avere anche quello sinistro,oppure avere solo quello sinistro).
Ogni nodo corrisponde a un elemento dell'array che memorizza il valore del nodo . Tutti i livelli dell'albero sono completamente riempiti tranne l'ultimo, che può essere riempito a sinistra sino ad un certo punto .

ATTRIBUTI:
- lenght[A] = numero di elementi dell'array
- heapsize[A] = numero di elementi registrati nell'heap

La radice dell'albero è A[1].
Per muoversi nell'albero si usa: PARENT[i], LEFT[i], RIGHT[i]

TIPI DI HEAP-BINARI:
- Max-heap: dove A[Parent[i]] >= A[i]
- Min-heap: dove A[Parent[i] <= A[i]

CARATTERISTICHE:
- Altezza di un nodo: numero di archi nel cammino più semplice , più lungo , che dal nodo scende fino alla foglia
- Altezza di un heap: numero di archi nel cammino più semplice , più lungo , che dal nodo radice scende fino alla foglia. Ha valore O(lgn).
- Tutte le operazioni fondamentali di un heap vengono eseguite in un tempo O(lgn), ovvero proporzionale con l'altezza dell'albero.

ALGORITMI:
- max-heapify
- build-max-heap
- heapsort
- maxheap-insert, maxheap-extract-max, heap-increase-key, heap-maximum

- Max-Heapify

[*Funzionalità***]**
E' un importante sub-routine per manipolare i max-heap, e fa si che la proprietà: A[Parent[i]] >= A[i] sia sempre rispettata.
[*Caratteristiche***]**
I suoi input sono un array A e un indice dell'array.
Quando viene chiamata questa subroutine si suppone che i sotto-alberi con radici in LEFT[i] e RIGHT[i] siano max-heap, ma che A[i] possa essere più piccolo dei suoi figli,violando cosi la proprietà del max heap .
La funzione max-heapify(A,i) consente al valore A[i] di "scendere" nel max-heap in modo che il sottoalbero con radice in i diventi max-heap.
[*Complessità***]**
La funzione max-heapify(A,i) viene eseguita sempre in un tempo O(lgn), ovvero proporzionale con l'altezza dell'albero.
Il caso migliore e`quando le chiamate al nodo rispetta le proprieta`della heap. Il caso peggiore quando devo controllare tutti i nodi fino in fondo.

- Build-Max-Heap

[*Funzionalità***]**
Serve a generare un max-heap da un array non ordinato.
[*Caratteristiche*]

[*Complessità*]
L'heap viene generato in tempo lineare O(n).

- Heapsort

[*Funzionalità*]
Serve ad ordinare sul posto un array.
[*Caratteristiche*]
Riceve come input un array non ordinato.

1) Prevede prima la costruzione di un max-heap con "Build-Max-Heap".
2) Prende la radice e lo mette in fondo all'albero.
3) Cala il valore di heap-size[A] , quindi l'elemento che prima era la radice dell'albero viene estratto dell'albero .
4) Esegue "Max-Heapify" sugli elementi rimasti per ripristinare la proprietà dell'heap nel caso sia stata violata .
5) Se heapsize[A]>1 ricomincia dal punto 2.
L'array è stato ordinato .

[*Complessità*]

L'heapsort viene eseguito in O(nlgn), poichè la chiamata "Build-Max-Heap**"** impiega un tempo O(n), e ciascuna delle n-1 chiamate di "Max-Heapify" impiega O(lgn).

[*Considerazioni*]

Heapsort è un eccellente algoritmo,ma una buona implementazione di quicksort batte heapsort.
Il caso migliore e`quando il vettore in input deve essere una heap. Il caso peggiore quando il vettore e`una heap all'incontrario. Se implementato con un ciclo "for" diventa in- place.

- Code di priorità (SD)

Struttura dati che utilizza gli heap binari. Serve a gestire un insieme S di elementi, ciascuno dei quali ha un valore associato detto *chiave*.

TIPI DI CODE di PRIORITA':

- Max-priorità
- Min-priorità

OPERAZIONI su Max-priorità:

- Insert(S,x) : inserisce l'elemento x nell'insieme S.
- Maximum(S): restituisce l'elemento S con la chiave più grande.
- Extract-Max(S): elimina e restituisce l'elemento di S con la chiave più grande.
- Increase-key(S,x,k): aumenta il valore della chiave dell'elemento x al nuovo valore di k, che si suppone sia almeno pari al valore corrente della chiave dell'elemento x.

OPERAZIONI su Min-priorità:

- Insert(S,x) : inserisce l'elemento x nell'insieme S.
- Minimum(S): restituisce l'elemento S con la chiave più piccola.
- Extract-Min(S): elimina e restituisce l'elemento di S con la chiave più piccola.
- Decrease-key(S,x,k): diminuisce il valore della chiave dell'elemento x al nuovo valore di k, che si suppone sia almeno pari al valore corrente della chiave dell'elemento x.

- Quicksort

[*Funzionalità***]**

Risolve il problema dell'ordinamento **(per confronti)**. Basato sul paradigma divide et impera ed è ricorsivo . Usa un metodo ausiliario Partition(A,p,r) che seleziona un elemento x = A[r] come **pivot** intorno al quale partiziona l'array.

Divide: partitiona l'array A[p...r] in due sottoarray A[p..q-1] e A[q+1...r]
Impera: ordina i due sottoarray A[p...q-1] e A[q+1..r] chiamando ricorsivamente quicksort.
Combina: poichè i sottoarray sono ordinati sul posto, non occore alcun lavoro per combinarli: l'intero array A[p...r] è ordinato.

Inserito da: -Filippo Vivaldi 03/07/10 14.40

[Caratteristiche]
Quicksort è un algoritmo**IN-PLACE**ma non è STABILE .

[*Complessita***]**
Il tempo di esecuzione dell'algoritmo dipende dal fatto che il partizionamento sia bilanciato o sbilanciato. Se il partizionamento è bilanciato ha la stessa velocità di mergesort. Altrimenti potrebbe essere lento quanto insertion-sort.
Se il pivot viene preso come min o come max, oppure il vettore e`gia ordinato in maniera decrescente, si e`nel caso peggiore; se invece il pivot viene preso a meta`dell'array dove n e`dispari, si e`nel caso migliore

Tempo di esecuzione atteso : $\Theta(nlgn)$ quando il partizionamento è bilanciato. La ricorsione produce un'albero di ricorsione di profondità O(lgn) dove ogni livello ha complessitò O(lgn).

Caso peggiore : $\Theta(n^2)$ Il caso peggiore è quando lo sbilanciamento è massimo. Il caso pessimo si ha quando l'array in input è già ordinato.

- Quicksort randomized*

[*Funzionalita***]**
Funziona nello stesso modo come il quicksort originale solo che nella procedura Partition invece di prendere l'ultimo numero come pivot viene preso un numero **random** .

[*Caratteristiche***]**
Ha le stesse caratteristiche di quicksort . Ma il vantaggio di questo algoritmo è che il caso peggiore ha un probabilità più piccola .

[*Complessità***]**
Caso peggiore : $\Theta(n^2)$.
Tempo di esecuzione atteso : $\Theta(nlgn)$.

- Counting sort

[*Funzionalita***]**
L'algoritmo counting sort suppone che ciascuno degli n elementi di input sia un numero intero compreso nell'intervallo da 0 a k , per qualche intero k . Il concetto dietro questo algortimo è determinare per ogni elemento di input x , il numero di elementi minori di x . Questa informazione può essere utilizzata per inserire l'elemento direttamente nella sua posizione nell'array di output .

ALGORTIMO:
Counting-Sort(A,B,k) , dove A è l'array in input, B è l'array che contiene l'output, k è il valore massimo della chiave.
1) Viene salvato nell'array C le occorrenza di ogni valore i dell'array A
es. ci sono tre 7, un 5...
2) L'array C viene "aggiornato" con il numero di elementi dell'array <= i.
es. ci sono quattro elementi <= 2, ci sono otto elementi <= 5.
L'algoritmo conta il numero di occorrenze di ciascun valore presente nell'array da ordinare, memorizzando questa informazione in un array temporaneo di dimensione pari all'intervallo di valori. Il numero di ripetizioni dei valori inferiori indica la posizione del valore immediatamente successivo.
3)Vengono salvati gli output nell'array B

[*Caratteristiche***]**
Un'importante proprietà di questo algoritmo è la **stabilità** . Counting sort viene spesso utilizzato come subroutine di radix sort .

[*Complessità***]**
Caso k != n : $O(k+n)$ **.**
Caso k == n : $O(n)$ **.**

(dove k limite dell'intervallo degli possibili valori e n è il numero degli elementi)

-Radix sort

[*Funzionalità***]**
Radix sort risolve il problema dell'ordinamento delle schede perforate in contenitori in funzione della posizione della scheda perforata in una maniera contraria all'intuizione , ordinando prima le schede in base alla cifra meno significativa . Le schede vengono poi combinate in un unico mazzo dove : le schede del contenitore 0 precedono quelle del contenitore 1 e cosi via . Poi tutto il mazzo viene ordinato in base alla seconda cifra meno significativa . Il processo continua finché le schede saranno ordinate rispetto a tutte le d cifre . Occorrono solo d passaggi attraverso il mazzo per completare l'ordinamento .

[*Caratteristiche***]**
Per Radix Sort è essenziale che gli ordinamenti delle cifre siano stabili . Radix Sort può essere utilizzato per ordinare record di informazioni con più campi chiave .

[*Complessità***]**
Dati n numeri di d cifre, dove ogni cifra può avere fino a k valori possibili, la complessità è $\Theta(d(n+k))$

Complessità di Radix Sort con n numeri di b bit : $\Theta((b/r)(n+2^r))$. Con d = b/r

- Quando $r \leq b$:
  Se $b<lgn$ per qualsiasi valore di $r \leq b$ si ha : $(n+2^r) = \Theta(n)$
- Quando $r = b$
  Si ottiene un tempo di esecuzione ottimale : $(n+2^b) = \Theta(n)$
- Quando $b \geq lgn$ e $r = lgn$
  Si ottiene un tempo ottimale a meno di un fattore costante : $\Theta(bn/lgn)$
- Quando $b \geq lgn$ e $r>lgn$
  Si ottiene : $\Omega(bn/lgn)$
- Quando $b \geq lgn$ e $r<lgn$
  Si ottiene : $\Theta(n)$

(dove r un intero positivo)

-Bucket sort

[*Funzionalità***]**
Bucket sort suppone che l'input sia generato da un processo casuale che distribuisce gli elementi uniformemente nell'intervallo [0,1) . Il concetto che sta alla base di questo algoritmo è quello di dividere l'intervallo [0,1) in n sottointervalli della stessa dimensione detti **bucket**, e poi ne distribuire gli n numeri di input nei bucket . Poiché gli input sono uniformemente distribuiti non ci si aspetta che tanti numeri vadano a finire nello stesso bucket . Per produrre l'output si ordinano gli elementi dentro ogni bucket e si elencano gli elementi di ciascuno .

[*Caratteristiche***]**
Questo algoritmo è veloce perché fa un'ipotesi sull'input . La sua stabilità dipende dall'algoritmo che si usa per ordinare i bucket .

[*Complessità***]**
Il tempo di esecuzione atteso di bucket sort è lineare quando l'input è estratto da una distribuzione uniforme .Tempo atteso : $O(n)$.

Tale tempo si ha anche se i dati non sono distribuiti uniformemente, ma la somma dei quadrati delle dimensioni dei bucket è lineare rispetto al numero totale degli elementi.
Il caso peggiore tutti i numeri cadono in un solo bucket, allora si avrà $O(n^2)$

L'i-esima statistica d'ordine di un insieme di n elementi è l'i-esimo elemento più piccolo. Per esempio, il **minimo** di un insieme di elementi è la prima statistica d'ordine (i=1) e il massimo è l' n-esima statistica d'ordine (i=n). La mediana è l'n-esima statistica che sta in mezzo.
Il **problema della selezione** può essere definito formalmente in questo modo:

- Input: un insieme A di n numeri(distinti) e un numero i, con 1<=i<=n
- Output: l'elemento $x \in A$ che è maggiore esattamente di altri i-1 elementi di A.

MINIMO E MASSIMO
Per determinare il minimo o il massimo in una statistica d'ordine sono necessari n-1 confronti.
Se si ricercano simultaneamente sono sufficienti: *3 * intero inferiore di (n/2)* confronti, $O(n)$.

SELEZIONE ELEMENTO
In generale richiede un tempo lineare, $O(n)$.

-Randomized Select

[*Funzionalità***]**
Randomized select è modellato sull'algoritmo quicksort . L'idea è di partizionare ricorsivamente l'array di input , la differenza è che Randomized select opera soltanto su un lato della partizione. Randomized select usa la procedura Randomized partition .
[*Caratteristiche***]**
Il funzionamento di Randomized-Select(A,p,r,i) è determinato in parte dall'output di un generatore di numeri casuali .

ATTRIBUTI:

- A è l'array da scorrere
- p indice più a sinistra dell'array
- r indice più a destra dell'array
- i è l'i-esimo elemento da ricercare

ALGORITMO:
Randomized-Select(A,p,r,i)

1. Se p=r allora ritorna A[p]
2. q<- RP(A,p,r) , esegue la partizione dell'array in base ad un pivot scelto a caso
3. k <- q-p+1 (numero di elementi nella parte bassa della partizione + il pivot)
4. Se k=i, ritorno A[k]
5. Se k<i, allora rieseguo l'algoritmo su Randomized-Select(A,p,q-1,i)
6. Se k>i, allora rieseguo l'algoritmo su Randomized-Select(A,q+1,r,i-k)

NB: si cerca l'i-k esimo elemento perchè i k elementi più piccoli di i sono nell'altro array

[*Complessità***]**
Caso peggiore : $O(n^2)$. Poiché potremmo essere estremamente sfortunati, effettuando la partizione attorno all'elemento più grande rimasto. Tuttavia l'algoritmo funziona bene perchè la scelta dell'input è randomizzata, quindi non si può verificare il caso peggiore.
Caso medio : $O(n)$. Nell'ipotesi che tutti gli elementi siano distinti .

- Select
[Funzionalità]
L'algoritmo select trova l'elemento desiderato partizionando ricorsivamente l'array di input . L'idea che sta la base di questo algoritmo è quella di una buona ripartizione dell'array. Come Quick sort l'algoritmo usa la procedura deterministica Partition con una modifica che consente di accettare come parametro d'input l'elemento attorno al quale effettuare il partizionamento .
Questo algoritmo:
1) Divide gli n elementi dell'array di input in **n/5 gruppi di 5 elementi ciascuno**.
2) Trova la mediana di ciascuno degli n/5 gruppi effettuando prima un ordinamento per inserimento degli elementi di ogni gruppo e poi scegliendo la mediana dalla lista ordinata degli elementi di ogni gruppo .
3) Usa select ricorsivamente per trovare la mediana x delle n/5 mediane trovate nel passo 2
4) Partiziona l'array intorno alla mediana delle mediane, utilizzando la versione modifacata di Partition . Se la posizione della mediana è l'elemento cercato l'algoritmo ritorna il valore.
Se non è cosi cerca a destra se l'elemento è più grande della mediana e a sinistra altrimenti .
[*Caratteristiche***]**
Select è un algoritmo deterministico .
[*Complessità***]**
Nel **caso peggiore** la complessità del algoritmo è $O(n)$; tuttavia le costanti nascoste nella notazione sono grandi.

- Hashing (SD)
L'accesso algli elementi nella tabella avviene tramite operazioni aritmetiche che trasformano le chiavi in indirizzi della tabella.

a)Tavola hash a indirizzamento diretto
[*Funzionalità****]***
Utilizzato per rappresentare un insieme dinamico dove ogni posizione o **cella** corrisponde a una chiave dell'universo U .
[*Caratteristiche****]***
L'indirizzamento diretto è una tecnica che funziona bene quando l'universo U delle chiavi è ragionevolmente piccolo .
[*Complessità****]***
Direct-Adress-Search , Direct-Adress-Insert e Direct-Adress-Delete sono operazioni che possono essere eseguiti in tempo

b)Tavola hash
[*Funzionalità***]**
Vengono usate quando l'universo delle chiavi U è esteso , cosi che non è possibile creare una tavola hash di quella dimensione . Inoltre l'insieme delle chiavi memorizzate potrebbe essere cosi piccolo rispetto all'universo U che la maggior parte della tavola sarebbe sprecata . Con il hashing ogni elemento è memorizzato nella cella h(k) dove h è una funzione detta **funzione hash** , usata per calcolare la cella della chiave k . Qui h associa l'universo U delle chiavi alle celle di una tavola hash T[0...m-1] . Il compito della funzione è ridurre l'intervallo degli indici dell'array da gestire . Pur troppo due chiavi possono essere associati alla stessa cella , questo evento si chiama **collisione** .
Risoluzione delle collisione mediante **concatenamento** . Questa tecnica pone tutti gli elementi che sono associati alla stessa cella in una lista concatenata .
[*Caratteristiche***]**
Si definisce **fattore di carico** α della tavola T il rapporto n/m , ossia il numero medio di elementi memorizzati in una lista . Questa misura viene utilizzata per l'analisi del comportamento delle tavole hash .
[*Complessità***]**
Il tempo di esecuzione nel caso peggiore per l'**inserimento** (Chained-Hash-Insert) è : $O(1)$.
La **ricerca** (Chained-Hash-Search) nel caso peggiore è proporzionale alla lunghezza della lista .

La **cancellazione** (Chained-Hash-Delete) ha un tempo di esecuzione : $O(1)$ (quando le liste sono doppiamente concatenate) più il tempo per la ricerca .

Caso peggiore per il concatenamento: Tutte le n chiavi sono memorizzate nella stessa cella. Il tempo di esecuzione è quindi $\Theta(n)$ più il tempo di calcolare la funzione hash.
Supponiamo che tutte le chiavi abbiano la stessa probabilità di cadere in una qualsiasi delle m celle. Questa ipotesi è detta **hashing uniforme semplice,** con quale una ricerca con successo richiede un tempo atteso di $\Theta(1+\alpha)$.

c)Funzioni hash
Una buona funzione hash soddisfa (approssimativamente) l'ipotesi dell'hashing uniforme semplice : ogni chiave ha la stessa probabilità di essere mandata in una qualsiasi delle m celle, indipendentemente dalla cella cui viene mandata qualsiasi altra chiave .

- *Il metodo della divisione :* una chiave k viene associata a una delle m celle prendendo il resto della divisione fra k e m , la funzione è : $h(k)=k\%m$. Questo metodo è molto veloce perché usa una sola operazione . Con questo metodo è molto importante scegliere bene il numero m . Una buona soluzione potrebbe essere un numero primo non troppo vicino a una potenza di 2 .
- *Il metodo della moltiplicazione :* si svolge in due passaggi . Prima la chiave k viene moltiplicata per una costante A nell'intervallo 0<A<1 ed estraiamo la parte frazionaria di kA. Poi si moltiplica questo valore per m e prendiamo la parte intera inferiore del risultato . La funzione è : $h(k)=m(kA\%1)$. Un vantaggio di questo metodo è che il numero m non è più critico . A invece viene preso come una frazione della forma $s/2^w$ dove w è la dimensione della parola macchina e s è un intero nell'intervallo 0 < s < 2^w.

d)Indirizzamento aperto

Nell'indirizzamento aperto tutti gli elementi sono memorizzati nella stessa tavola hash; ovvero ogni voce della tavola contiene un elemento dell'insieme dinamico o la costante NIL (oppure Deleted) .
Quando cerchiamo un elemento , esaminiamo le celle della tavola finché non troviamo l'elemento desiderato o finché non ci accorgiamo che l'elemento non si trova nella tavola . Non ci sono liste né elementi memorizzati all'esterno della tavola . Il vantaggio dell'indirizzamento aperto sta nel fatto che esclude completamente i puntatori consentendo di avere un maggiore numero di celle che riduce il numero di collisioni , ma la tavola può riempirsi (teoricamente) . La sequenza delle celle esaminate durante una ispezione dipende dalla chiave da inserire.

La funzione hash viene estesa in modo da includere l'ordine di ispezione . Ogni posizione della tavola hash può essere considerata come possibile cella in cui inserire una nuova chiave mentre la tavola si riempie .

- *Ispezione lineare* : h' : U -> {0,1,...,m-1} funzione hash ausiliaria . Il metodo dell'inspezione lineare usa la funzione hash : $h(k,i)=(h'(k)+i)\%m$. l'ispezione lineare è facile da implementare ma presenta un problema noto come *"addensamento primario" :* lunghe file di celle occupate che aumentano il tempo di ricerca a causa del fato che la probabilità che una cella vuota preceduta da i celle occupate è $(i+1)/m$.
- *Ispezione quadratica* : $h(k,i)=(h'(k)+c_1i+c_2i^2)\%m$. Dove i valori c_1 e c_2 non si possono scegliere arbitrariamente . Porta a una forma più lieve di addensamento , che viene chiamato *"addensamento secondario"* .
- *Doppio hashing* : è il metodo migliore disponibile per l'indirizzamento aperto e usa una funzione hash di forma : $h(k,i)=(h_1(k)+ih_2(k))\%m$ dove h_1 e h_2 sono due funzioni ausiliarie . Il doppio hashing è migliore delle ispezioni lineari e quadratiche in quanto usa $\Theta(m^2)$ sequenze di ispezione .

[Complessità**]**
Nell'ipotesi di hashing uniforme e sia α il fattore di carico , $\alpha<1$ il numero atteso di ispezioni in una ricerca senza successo è al massimo : $1/(1-\alpha)$
Quando il fattore di carico $\alpha<1$ e nell'ipotesi di hashing uniforme il numero atteso di ispezioni in una ricerca con successo è al massimo : $(1/\alpha)ln(1/(1-\alpha))$.

- Programmazione dinamica

La programmazione dinamica risolve i problemi combinando le soluzioni dei sottoproblemi quando i sottoproblemi non sono indipendenti , ovvero quando i sottoproblemi hanno in comune dei sottoproblemi . Un algoritmo di programmazione dinamica risolve ciascun sottoproblema una sola volta e salva la soluzione in una tabella evitando il ricalcolo .

La programmazione dinamica viene di solito applicata ai problemi di ottimizzazione . Per questo tipo di problemi di solito ci possono essere molte soluzioni possibili . Si vuole trovare la soluzione con il valore ottimo .

Il processo di sviluppo di un algoritmo di programmazione dinamica può essere suddiviso in quattro fasi :

1) Caratterizzare la struttura di una soluzione ottima .
2) Definire in modo ricorsivo il valore di una soluzione ottima .
3) Calcolare il valore di una soluzione ottima secondo uno schema bottom-up .
4) Costruire una soluzione ottima dalle informazioni calcolate .

Le prime tre fasi formano la base per risolvere un problema applicando la programmazione dinamica oppure solo per calcolare il valore di una soluzione ottima . I due principali ingredienti di un problema di ottimizzazione affinché possa essere applicata la programmazione dinamica sono : la **sottostruttura ottima** e i **sottoproblemi ripetuti** .

1. ***Sottostruttura ottima :*** Un problema presenta una sottostruttura ottima se una soluzione ottima del problema contiene al suo interno le soluzioni ottime dei sottoproblemi . Quando un problema presenta una sottostruttura ottima è un buon indizio dell'applicabilità della programmazione dinamica. Una soluzione ottima si costruisce dalle soluzioni ottime dei sottoproblemi, quindi dobbiamo essere sicuri che l'insieme dei sottoproblemi considerati includa quelli utilizzati in una soluzione ottima. Quando si cerca di scoprire una sottostruttura ottima si segue uno schema comune :

1). Dimostrare che una soluzione del problema consiste nel fare una scelta. La scelta lascia uno o più problemi da risolvere.
2). Per un dato probl ema, supponete di conoscere la scelta che porta a una soluzione ottima. Non interessa sapere come sia stata determinata, supponete di conoscere tale scelta.
3). Fatta la scelta determinate quali sottoproblemi considerare e quale sia il modo migliore per rappresentare lo spazio di sottoproblemi risultante
4). Dimost rare che le soluzioni dei sottoproblemi che avete utilizzato all'interno della soluzione ottima del problema devono essere necessariamente ottime, adottando una tecnica "taglia e incolla". Prima supponete che ciascuna delle soluzioni dei sottoproblemi non sia ottima e , poi arrivate a una contraddizione. "Tagliando" la soluzione non ottima di un sottoproblema e "incollando" quella ottima dimostrate che potete ottenere una soluzione migliore del problema originale,contraddicendo l'ipotesi di avere già una soluzione ottima. La sottostruttura ottima varia in funzione del tipo di problema in due modi: per il numero di sottoproblemi che sono utilizzati in una soluzione ottima del problema originale,

per il numero di scelte che possiamo fare per determinare quale sottoproblema utilizzare in una soluzione ottima . Il tempo di esecuzione di un algoritmo di programmazione dinamica dipende dal prodotto di due fattori : il numero di sottoproblemi da risolvere complessivamente e il numero di scelte da considerare per ogni sottoproblema . La programmazione dinamica usa un approccio **bottom-up** , prima vengono trovate le soluzioni ottime dei sottoproblemi e poi viene trovata la soluzione ottima del problema .

2. ***Ripetizione dei sottoproblemi:*** Lo spazio dei sottoproblemi deve essere "piccolo", un algoritmo ricorsivo per il problema risolve ripetutamente gli stessi sottoproblemi, anziché generare sempre nuovi sottoproblemi. Tipicamente il numero totale di sottoproblemi distinti è un polinomio nella dimensione dell'input . Quando un algoritmo ricorsivo rivisita più volte lo stesso problema diciamo che il problema di ottimizzazione ha dei **sottoproblemi ripetuti** . Gli algoritmi di programmazione dinamica sfruttano i sottoproblemi ripetuti risolvendo ciascun sottoproblema una sola volta e poi memmorizzando la soluzione in una tabella da cui può essere recuperata quando serve, impiegando un tempo costante .

La tecnica di **annotazione** è una variante della programmazione dinamica che, pur conservando la strategia top down, spesso offre la stessa efficienza dell'uso di un algoritmo di programmazione dinamica. Il concetto che sta alla base di questa tecnica consiste nel dotare di un blocco per appunti il naturale, ma inefficiente algoritmo ricorsivo, viene utilizzata una tabella con le soluzioni dei sottoproblemi, ma la struttura di controllo per riempire la tabella è più simile a quella ricorsiva. Ogni posizione della tabella inizialmente contiene un valore speciale per indicare che la posizione non è stata ancora riempita . Questo approccio presuppone che sia noto l'insieme di tutti i parametri dei sottoproblemi e che sia definita la relazione fra le posizioni della tabella e i sottoproblemi. In generale se tutti i sottoproblemi devono essere risolti almeno una volta. Un algoritmo di programmazione dinamica di solito supera le prestazioni di un algoritmo con annotazione per un fattore costante, dovuto ai costi per la ricorsione e i costi per la gestione della tabella che sono minori nella programmazione dinamica.

Programmazione delle catene di montaggio :

Dati del problema: Un'azienda produce automobili in uno stabilimento con due catene di montaggio. Ogni catena ha n stazioni numerate con j = 1,2,...,n . Indichiamo con S_{ij} la j-esima stazione della catena i . Il tempo richiesto in ogni stazione varia anche fra stazioni che occupano la stessa posizione nelle due catene . Indichiamo con a_{ij} il tempo di montaggio richiesto nella stazione S_{ij} , indichiamo inoltre con t_{ij} il tempo per trasferire un telaio da una catena di monteggio i. Il problema è determinare quali stazioni scegliere dalla catena 1 e dalla catena 2 per minimizzare il tempo totale per completare l'assemblaggio di un'auto .

- *la struttura del percorso più rapido :* Consideriamo il percorso più veloce possibile che puo seguire un telaio dal punto iniziale fino alla stazione S_{1j} . Per j = 1 c'è solo un percorso che il telaio può seguire . Per j = 2,3,...n ci sono due scelte : il telaio può provenire da S_{1j-1} oppure S_{2j-1} e in questo caso si somma anche il tempo di trasferimento t_{2j-1} . In primo luogo si suppone che il percorso più rapido fino a S_{1j} passi per la stazione S_{1j-1} (analogamente S_{2j-1}) , se no c'è una **contraddizione** . Possiamo dire che per la programmazione delle catene di montaggio, una soluzione ottima di un problema contiene al suo interno una soluzione ottima di sottoproblemi, indicato col nome **sottostruttura ottima** . Il percorso più rapido per arrivare alla stazione S_{1j} può essere : A)il percorso più rapido per raggiungere S_{1j-1} e andare direttamente a S_{1j} e B)il percorso più rapido per raggiungere S_{2j-1} fare il trasferimento e arrivare a S_{1j} . Analogamente per S_{2j} . Possiamo costruire una soluzione ottima sfruttando le soluzioni ottime dei sottoproblemi .
- *una soluzione ricorsiva :* Scegliamo come sottoproblemi i problemi di trovare il percorso più rapido fino alla stazione j in entrambe le catene di montaggio , con j = 1,2,...,n. Indichiamo con $f_i[j]$ il tempo più piccolo possiblie che impiega un

telaio dal punto iniziale fino all'uscita della stazione S_{ij} . Indichiamo con f* il tempo minimo che impiega un telaio dal punto iniziale fino all'uscita . Il più rapido di questi percorsi è il percorso più rapido per l'intero stabilimento abbiamo : $f^* = min(f_1[n]+x1, f_2[n]+x2)$. Le equazioni ricorsive sono :

$f_1[j] = min(f_1[j-1]+a_{1,j}, f_2[j-1]+t_{2,j-1}+a_{1,j})$ se $j \geq 2$

$f_2[j] = min(f_2[j-1]+a_{2,j}, f_1[j-1]+t_{1,j-1}+a_{2,j})$ se $j \geq 2$

Definiamo inoltre $l_i[j]$ come il numero della catena di montaggio , 1 o 2 , la cui stazione j-1 è utilizzata nel percorso più rapido per arrivare alla stazione S_{ij} , e l* come la catena la cui stazione n è utilizzata nel percorso più rapido dell'intero stabilimento .

- *calcolo dei tempi minimi :* Siccome c'è un problema con l'algoritmo ricorsivo : il suo tempo di esecuzione è esponenziale in n ; calcoliamo i valori di $f_i[j]$ in un ordine diverso da quello ricorsivo . Calcolando i valori di $f_i[j]$ per valori crescenti dell'indice j delle stazioni , da sinistra a destra possiamo calcolare nel tempo $\Theta(n)$ il percorso più rapido che attraversa lo stabilimento e il tempo richiesto per percorrerlo .
- *costruzione del percorso più rapido :* Avendo già calcolato i valori di $f_i[j]$,f*, $l_i[j]$,l* dobbiamo solo creare una procedura di stampa dei valori di $l_i[j]$.

- Algoritmi golosi

Un algoritmo goloso e caratterizzato dal fatto che fa sempre la scelta che sembra ottima in un determinato momento, ovvero fa una scelta *localmente ottima*, nella speranza che tale scelta porterà a una soluzione globalmente ottima. Gli algoritmi golosi non sempre riescono a trovare le soluzioni ottime, ma per molti problemi sono in grado di farlo . Un algoritmo goloso produce una soluzione "ottima" di un problema effettuando una sequenza di scelte . Un algoritmo goloso si basa su alcuni elementi della programmazione dinamica (sottostruttura ottima) . Per sviluppare un algoritmo goloso si possono compiere i seguenti passaggi :

1. Definire il problema di ottimizzazione come quello in cui , fatta una scelta resta un solo sottoproblema da risolvere .
2. Dimostrare che esiste sempre una soluzione ottima del problema originale che fa la scelta golosa , quindi la scelta golosa è sempre sicura .
3. Dimostrare che avendo fatta la scelta golosa , ciò che resta è un sottoproblema con la proprietà che , se combiniamo una soluzione ottima del sottoproblema con la scelta golosa che abbiamo fatto , arriviamo a una soluzione ottima del problema originale .

Ogni algoritmo goloso contiene due proprietà chiave :

- ***Scelta golosa :*** una soluzione globalmente ottima può essere ottenuta facendo una scelta localmente ottima . Quando valutiamo la scelta da fare , facciamo la scelta che sembra migliore per il problema corrente , *senza considerare le soluzioni dei sottoproblemi* . La scelta fatta da un algoritmo goloso può dipendere dalle scelte precedenti , ma non può dipendere dalle scelte future o dalle soluzioni dei sottoproblemi . Una strategia golosa di solido procede dall'alto vero il basso (top-down) , facendo una scelta golosa dopo l'altra , riducendo ogni istanza di un determinato problema a un problema più piccolo . Dobbiamo dimostrare che una scelta golosa in ogni passaggio genera una soluzione globalmente ottima . Grazie a un'elaborazione preliminare dell'input o all'impiego di una struttura dati appropriata , riusciamo a fare rapidamente delle scelte golose , ottenendo cosi un agloritmo efficiente .
- ***Sottostruttura ottima :*** un problema ha una sottostruttura ottima se una soluzione ottima del problema contiene al suo interno soluzione ottime dei sottoproblemi . Si concede il lusso di assumere di arrivare a un sottoproblema facendo la scelta golosa nel problema originale . Tutto ciò che si deve fare è

dedurre che una soluzione ottima del sottoproblema combinata con la scelta golosa già fatta , genera una soluzione ottima del problema originale .

Confronta fra strategia golosa e la programmazione dinamica : Poiché la sottostruttura ottima viene sfruttata da entrambe le strategie è possibile sbagliare nella scelta . Un buon esempio è il classico problema dello zaino .

- ***Problema dello zaino 0-1*** : Un ladro entra in un magazzino e trova n oggetti ; l'i-esimo oggetto vale v_i e pesa w_i . Il ladro vuole realizzare il furto di maggior valore , ma il suo zaino può supportare un peso W e il ladro non può prendere una frazione dell'oggetto . Quali oggetti dovrà prendere ?
- **Problema dello zaino frazionario** : le condizioni sono le stesse , con la differenza che il ladro può prendere frazioni di oggetti .

Entrambi i problemi presentano la proprietà della sottostruttura ottima . Il problema dello zaino frazionario può essere risolto con un algoritmo goloso , il problema dello zaino 0-1 no .

Problema della selezione di attività :

Supponiamo di avere un insieme S = $\{a_1,a_2,a_3,...,a_n\}$ di attività che devono utilizzare la stessa risorsa . Ogni attivita a_i ha un tempo di inizio s_i e un tempo di fine f_i . Le attivita a_i e a_j sono **compatibili** se gli intervalli $[s_i,f_i]$ e $[s_j,f_j]$ non si sovrappongono . Il *problema della selezione di attività* consiste nel selezionare il sottoinsieme che contiene il maggior numero di attività mutuamente compatibili .

Iniziamo definendo gli insiemi $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$ in modo che S_{ij} sia il sottoinsieme delle attività in S che possono iniziare dopo che è finita l'attività a_i e con tutte le attività che finiscono prima di iniziare a_j . Per rappresentare il problema aggiungiamo delle attività fittizie a_0 con $f_0 = 0$ e a_{n+1} con $s_{n+1} = \infty$.

Teorema

Consideriamo un sottoproblema non vuoto S_{ij} ; indichiamo con a_m l'attività in S_{ij} che ha il primo tempo di fine : $f_m = min\{f_k : a_k \in S_{ij}\}$. Allora :

1. *L'attività a_m è utilizzata in qualche sottoinsieme massimo di attività mutuamente compatibili S_{ij} .*
2. *Il sottoproblema S_{im} è vuoto , quindi la scelta di a_m fa si che S_{mj} sia l'unico problema non vuoto .*

Il teorema riduce significativamente il problema , perché utilizziamo un solo sottoproblema in una soluzione ottima e dobbiamo considerare una sola scelta : quella con il primo tempo di fine in S_{ij}.

Possiamo risolvere ciascuno sottoproblema secondo uno schema top-down . Per risolvere S_{ij} scegliamo a_m in S_{ij} con il primo tempo di fine e aggiungiamo a questa l'insieme delle attività utilizzate in una soluzione ottima del sottoproblema S_{mj} prima di risolvere S_{ij} . Nella nostra soluzione S_{ij} , non dobbiamo risolvere S_{mj} prima di risolvere S_{ij} . Per risolvere S_{ij} , possiamo prima scegliere a_m come l'attivita in S_{ij} con il primo tempo di fine e *poi* risolvere S_{mj} .

L'attività a_m che scegliamo quando risolviamo un sottoproblema è sempre quella con il tempo di fine che può essere legittimamente scelta . L'attività selezionata è quindi una scelta golosa nel senso che , intuitivamente , lascia il maggior numero possibile di opportunità per la scelta delle restanti attività . Ovvero la scelta golosa è quella che massimizza la quantita di tempo residuo inutilizzato .

- Albero binario di ricerca (SD)

Struttura dati dove le operazzioni base richiedono un tempo proporzionale con l'altezza dell'albero, O(lgn) nel caso peggiore. E' organizzato come un albero binario, ovvero contiene un campo *key.* Ma contiene altri attributi.

ATTRIBUTI:

- key(x): chiave di x.
- left(x): chiave del figlio sinistro di x
- right(x): chiave del figlio destro di x
- p(x): chiave del padre di x

CARATTERISTICHE:

- Sia x un nodo dell'albero. Se y è un nodo del sottoalbero sinistro di x, allora key(y)<=key(x). Se y è un nodo del sottoalbero destro di x, allora key(y)>=key(x) .
- Se manca un filglio si mette NIL.
- Il nodo radice è l'unico il cui padre è NIL.
- L'altezza dell'albero è **O(lgn) = O(h)**.

ALGORITMI

- Inorder-tree-walk: Grazie alle proprietà degli alberi binari di ricerca, consente di elencare ordinatamente tutte le chiavi con un semplice algoritmo ricorsivo. Richiede tempo O(n).
- Tree-Search: Serve a cercare un nodo con una data chiave in un albero binario di ricerca . Restituisce un puntatore al nodo cercato, se esiste, altrimenti restituisce NIL. Tempo di esecuzione O(h), dove h è l'altezza dell'albero.
- Tree-Successor: Dato un nodo x, ne restituisce il sucessore. Tempo di esecuzione O(h).
- Tree-Insert: Inserisce un elemento nell'albero. Tempo di esecuzione O(h).
- Tree-Delete: Elimina un elemento dell'albero. Tempo di esecuzione O(h).
- Tree-Maximum e Tree-Minimum: ricercano rispettivamente l'elemento con chiave più grande e più piccola nel sottoalbero di radice x. Tempo di esecuzione O(h).

- Albero RossoNeri (SD)

E' un albero binario di ricerca con un attributo in più: *colore del nodo*.

ATTRIBUTI:

- key(x): chiave di x.
- left(x): chiave del figlio sinistro di x
- rignt(x): chiave del figlio destro di x
- p(x): chiave del padre di x
- color(x): può essere rosso o nero

CARATTERISTICHE:

- Gli RB alberi garantiscono che nessuno dei percorsi dalla radice alla foglia sia più di due volte più lungo di qualsiasi altro. Quindi l'albero è approssimativamente bilanciato.
- Altezza nera bh(x): è il numero di nodi neri lungo il percorso che inizia dal nodo x (ma non lo include) e finisce con una foglia.
- L'altezza massima di un RB albero con "n" nodi interni è 2lg(n+1).

PROPRIETA':

1. Ogni nodo è rosso o nero
2. La radice è nera
3. Ogni foglia (NIL) è nera
4. Se un nodo è rosso, allora entambi i suoi figli sono neri
5. Per ogni nodo, tutti i percorsi che vanno dal nodo alle foglie (sue discendenti) contegono lo stesso numero di nodi neri.

ALGORITMI:

- Tree-Search, Tree-Successor, Tree-Predecessor, Tree-Maximum, Tree-Minimum possono essere eseguiti in tempo O(h), ovvero O(lgn).

- Tree-Insert e Tree-Delete sono eseguiti in tempo O(lgn). Poichè queste operazioni modificano l'albero, il risultato potrebbe violare le proprietà degli alberi rosso-neri. Per ripristinare queste proprietà, dobbiamo modificare i colori di qualche nodo dell'albero e anche la struttura dei puntatori. Per fare ciò si compie una *rotazione* sugli elementi dell'albero.

ROTAZIONE:
- Operazione locale in un albero di ricerca che preserva la proprietà degli alberi binari di ricerca.
- Sono di due tipi: Left-Rotate(T,x) e Right(T,x).
- La procedura viene eseguita in tempo costante O(1).

INSERIMENTO:
- L'inserimento di un elemento in un RB albero può essere effettuato nel tempo O(lgn), tramite l'algoritmo "RB-Insert(T,Z)".
- Per garantire che le proprietà degli alberi rosso-neri siano preservate,al termine di "RB-Insert" si usa una procedura ausiliaria "RB-Insert-Fixup" che ricolora i nodi ed effettua delle rotazioni.

AGGIUNTA DI UN CAMPO:

- Sia *f un campo* che aumenta un RB albero di n nodi, supponiamo che il valore di f per un nodo x possa essere calcolato utilizzando le info di: x, left(x), right(x), f(left(x)) ovvero il campo f del figlio sinistro di x, f(right(x)) ovvero il campo f del figlio destro di x.
- La modifica di un campo f in un nodo x si propaga solo negli antenati di x nell'albero. Quindi aggiornamneto di f(x) significa aggiornamento di f(p(x)) e di f(p(p(x))) e cosi via...
- Poichè l'altezza di un RB albero è O(lgn) la modifica di un campo f in un nodo richiede un tempo O(lgn)

- Statistiche d'ordine dinamiche(SD)

L' i-esima "statistica d'ordine" di un insieme di n elementi, è semplicemente l'i-esima chiave più piccola dell'insieme.
Qualsiasi statistica d' ordine può essere ottenuta in un tempo O(n) da un insieme non ordinato.

ATTRIBUTI:
- Rango: posizione occupata da un elemento in una determinata sequenza.
- Size[x]: numero di nodi interni nel sottoalbero di radice x, compreso x. Size(x) = size(left) + size(right) + 1

RICERCA di un ELEMENTO con un DATO RANGO:
- Si usa l'algoritmo OS-Select(x,i), che restituisce un puntatore al nodo che contiene l' i-esima chiave più piccola nel sottoalbero con radice in x.
- Il tempo di esecuzione è O(lgn)

DETERMINARE il RANGO di un ELEMENTO
- Si usa la procedura OS-Rank(T,x), che restituisce la posizione x nell'ordinamento lineare determinato da un attraversamento simmetrico dell'albero T.
- Il tempo di esecuzione è O(lgn)

- A lberi di intervalli (SD)
Un albero di intervalli è un RB albero che gestisce un insieme dinamico di elementi, in cui ogni elemento x contiene un intervallo int[x] . Sono utilizzati per rappresentare eventi che si svolgono in un periodo continuo nel tempo.

- Intervallo Chiuso: coppia ordinata di numeri reali [t1,t2] con t1<=t2.
- Intervallo Aperto: omettono uno o entrambi gli estremi.

ATTRIBUTI:

- Oggetto "i" =intervallo(t1,t2)
- low[i] = t1 (estremo inferiore)
- high[i] = t1 (estremo superiore)

CARATTERISTICHE:

- Due oggetti i e i' si sovrappongono se $i \cap i' \neq \phi$, ovvero se low[i]<=high[i'] e low[i']<=high[i]
- Due intervalli i e i' soddisfano la tricotomia degli intervalli ; ovvero solo se una delle seguenti proprietà può essere vera:
  a. i e i' si sovrappongono
  b. i è a sinistra di i' (ovvero high[i] < low[i'])
  c. i' è a sinistra di i (ovvero high[i'] < low[i])

ALGORITMI
Interval-Search(T,i): questo algoritmo scende nell'albero fino a quando non trova un intervallo che sovrappone i; se non lo trova significa che non esiste. L'algoritmo continua a "stare" nella parte dove potrebbe esserci un intervallo che sovrappone i.

- B-Alberi (SD) - *Un accenno...*

- I B-Alberi (comunemente noti come B-Tree cioè Balanced-Tree) sono strutture dati ad albero, vengono comunemente utilizzati nell'ambito di database e dispositivi di memoria secondaria e ad accesso diretto.
- Essi derivano dagli alberi di ricerca, in quanto ogni chiave appartenente al sottoalbero sinistro di un nodo è di valore inferiore rispetto a ogni chiave appartenente ai sottoalberi alla sua destra.
- Derivano anche dagli alberi bilanciati perché tutte le foglie si trovano alla stessa distanza rispetto alla radice.
- Il vantaggio principale dei B-Tree è che essi mantengono automaticamente i nodi bilanciati permettendo operazioni di inserimento, cancellazione e ricerca in tempi ammortizzati logaritmicamente.

- Insiemi Disgiunti(SD)

- Una struttura dati per insiemi disgiunti mantine una collezione $S = \{S1, S2, ..., Sk\}$ di insiemi dinamici disgiunti, cioè che non hanno elementi in comune.
- Ciascun insieme è identificato da un rappresentante, che è un elemento dell'insieme (per esempio si potrebbe scegliere l'elemento più piccolo di un insieme).

OPERAZIONI:

- Makeset(x): crea un nuovo insieme il cui unico elemento(e rappresentante) è x.
- Union(x,y): unisce gli insiemi dinamici, per esempio Sx e Sy che contengono x e y come rappresentanti. Il rapresentante dell'insieme risultante è un elemento qualsiasi di $Sx \cup Sy$. Poichè è richiesto che gli insiemi della collezione siano disgiunti (ovvero non abbiano elementi in comune), Sx e Sy vengono distrutti, eliminandoli dalla collezione.
- Findset(x): restituisce un puntatore al rappresentante dell'insieme che contiene x.

- Heap riunibili (SD)

- Ovvero heap binari, heap binomiali, heap di Fibonacci(ammortizzato)

OPERAZIONI:

- **Make-heap()**: crea e restituisce un nuovo heap(albero binario quasi completo) senza elementi. Tempi esecuzione: O(1) per tutti e tre gli heap .
- **Insert(H,x)**: inserisce nell'heap H un nodo x, il cui campo key è già riempito. Tempi: O(lgn)(binari) ,O(lgn)(binomiali), O(1)(Fibonacci) .
- **Minimum(H)**: restituisce un puntatore al nodo dell' heap la cui chiave è minima. Tempi: O(1)(binari), O(lgn)(binomiali), O(1)(Fibonacci) .
- **Extract-Min(H)**: toglie dall'heap H il nodo con chiave minima. restituendo un puntatore al nodo. Tempi: O(lgn)(binari), O(lgn)(binomiali), O(lgn)(Fibonacci) .
- **Union(H1, H2)**: crea e restituisce un nuovo heap che contiene tutti i nodi degli heap H1 e H2. Gli heap H1 e H2 vengono "distrutti".Tempi: O(n)(binari), O(lgn)(binomiali), O(1)(Fibonacci) .
- **Decrease-Key(H,x,k)**: assegna al nodo x all'interno dell' haep H il nuovo valore della chiave k, che supponiamo non sia maggiore del suo valore corrente.Tempi: O(lgn)(binari), O(lgn)(binomiali), O(1)(Fibonacci) .
- **Delete(H,x)**:cancella il nodo x dall'heap H. O(lgn)(binari), O(lgn)(binomiali), O(lgn)(Fibonacci) .

CONSIDERAZIONI:

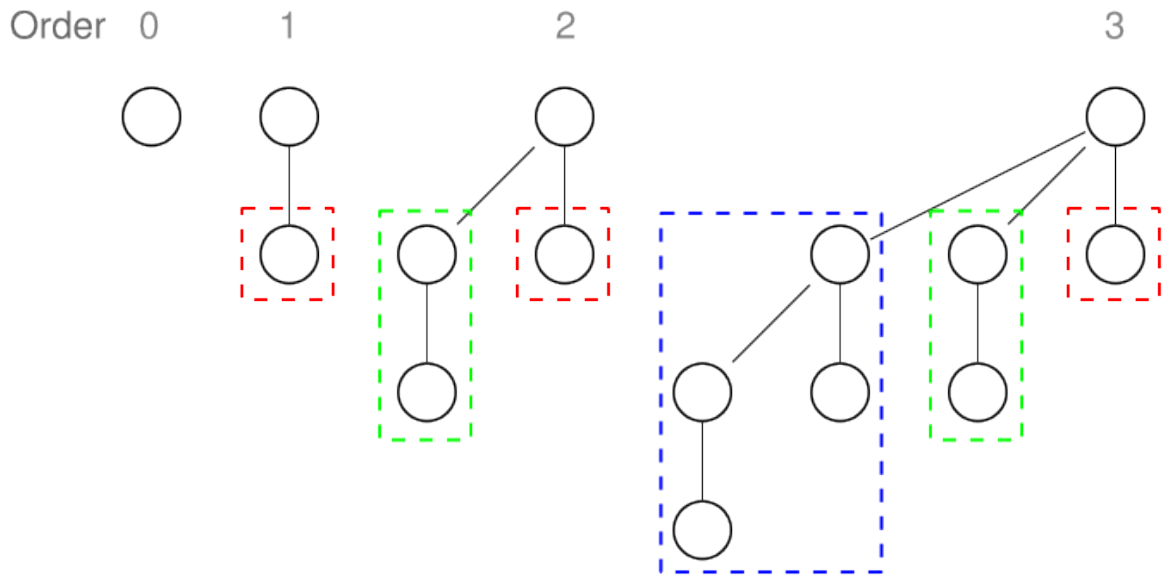
- Se NON si usa la UNION conviene usare gli heap binari.
- Se si usa la UNION, gli heap binari la eseguono in tempo O(n), quindi meglio usare gli heap binomiali che la eseguono in tempo O(lgn).
- Tutti e tre i tipi di heap eseguono l'operazione SEARCH in modo poco efficente, poichè scandire le chiavi di tutti i nodi potrebbe richiedere un tempo molto lungo. Per questo motivo a Decrease-Key(H,x,k) e Delete(H,x) gli viene passato un puntatore ad un nodo.

- Alberi Binomiali (SD)

Un *albero binomiale* Bk, è un albero ordinato definito in maniera ricorsiva.

NB: Un *albero ordinato* è un albero radicato (albero con nodo radice) in cui i figli di ogni nodo sono ordinati. Ovvero, se un nodo k ha figli, c'è un primo filgio, un secondo figlio, un k-esimo figlio.

ESEMPIO CONCRETO:



STRUTTURA RICORSIVA:

- L'albero B0 ha un 1 nodo. Ovvero la radice.
- L'albero B1 ha 2 nodi. Ovvero la radice + B0
- L'albero B2 ha 4 nodi. Ovvero la radice + B0 + B1
- L'albero B3 ha 8 nodi. Ovvero la radice + B0 + B1 + B2

PROPRIETA', per un albero binomiale Bk:

1. I nodi dell'albero sono 2^k
2. L'altezza dell'albero è k
3. Ci sono esattamente $\binom{k}{i}$ nodi alla profondità i per i=0,1,...,k
4. La radice ha grado k (ovvero ha k filgli), che è maggiore del grado di qualsiasi altro nodo; inoltre, se i figli della radice sono numerati da sinistra a destra con k-1, k-2,...,0 il figlio i è la radice di un sottoalbero Bi.

- Heap Binomiali (SD)

Un *heap binomiale* H è un insieme di alberi binomiali che soddisfano le seguenti proprietà degli heap binomiali:

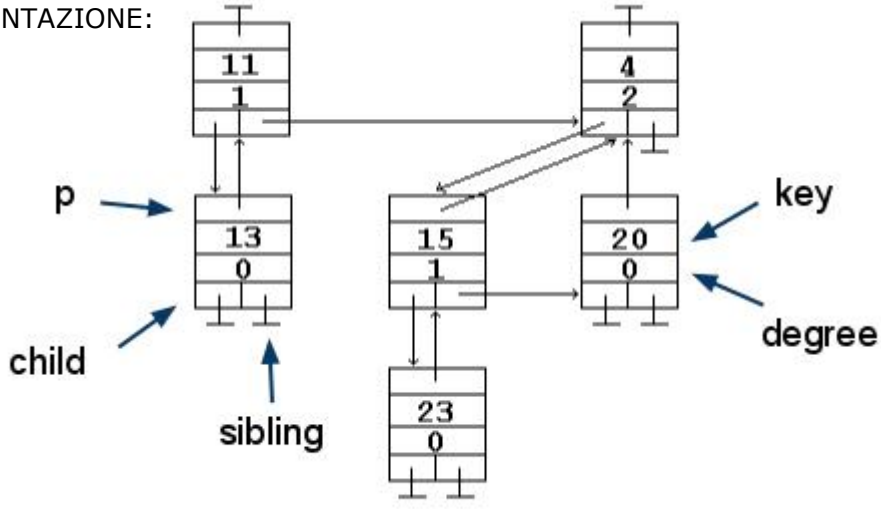
PROPRIETA':

1. Ogni albero binomiale in H soddisfa la proprietà del *min-heap:* la chiave di un nodo è maggiore o uguale alla chiave di suo padre.
2. Per qualsiasi intero k non negativo, c'è al più di un albero binomiale in H la cui radice ha grado k.

CONSIDERAZIONI:

1. La prima proprietà indica che la radice di un albero ordinato come min-heap contiene la chiave più piccola dell'albero.
2. La seconda proprietà implica che un heap binomiale H di n nodi è formato al massimo da: (lgn arrotondato all'intero inferiore) + 1 alberi binomiali.

RAPPRESENTAZIONE:



ATTRIBUTI:

- key(x) : contiene la chiave del nodo
- p(x): puntatore al padre del nodo
- child(x): puntatore al figlio più a sinistra
- sibling(x): puntatore al figlio immediatamente più a destra(o fratello destro)
- degree(x): numero di figli del nodo

CONSIDERAZIONI:

- Se il nodo x è una radice, allora p(x) = NIL.
- Se il nodo non ha figli, allora child(x) = NIL.
- Se x è il figlio più a destra di suo padre, allora sibling(x)=NIL
- Se x è una radice, allora sibling(x) punta alla sucessiva radice nella lista delle radici (notate che sibling[x] = NIL se x è l'ultima radice nella lista delle radici).

OPERAZIONI:

- Make-Binomial-Heap: alloca e restituisce un oggetto H, dove head[H] = NIL. Tempo di esecuzione è O(1).
- Binomial-Heap-Minimum: restituisce un puntatore al nodo con chiave minima in un heap binomiale H di n nodi. Poichè un heap binomiale è ordinato come un min-heap, la chiave minima deve risiedere in un nodo radice. La procedura esamina tutte le radici, salvando man mano il valore min e il puntatore al nodo corrispondente. Tempo di esecuzione: O(lgn).
- Binomial-Heap-Union(H1,H2): serve ad unire due heap-binomiali. La procedura collega ripetutamente gli alberi binomiali le cui radici hanno lo stesso grado.
- Binomial-Link(y,z): aggiunge il nodo y in testa alla lista concatenata dei figli del nodo z nel tempo O(1).
- Binomial-Heap-Insert(H,x): La procedura crea nel tempo O(1) un heap binomiale H' di un nodo e lo unisce nel tempo O(lgn) all'heap binomiale H di n nodi.

UNIRE DUE HEAP BINOMIALI:

La procedura Binomial-Heap-Union(H1,H2) unisce due heap binomiali H1 e H2 restituendo l'heap risultante (H1 e H2 sono distrutti).
La procedura si divide in due fasi:

1. Viene invocata la "Binomial-Heap-Merge" che fonde le due liste H1 e H2 in un'unica lista concatenata H che è ordinata per gradi di nodi monotonicamente crescenti.
2. Vengono unite le radici di pari grado (non più di due per grado). Risultera cosi una lista formata da radici di diverso grado. Poichè la lista è ordinata per grado, possiamo eseguire rapidamente tutte le operazioni di collegamento.

Il tempo di esecuzione O(lgn).

- Grafi (SD)

G=(E,V); E = insieme degli archi; V = insieme dei nodi

RAPPRESENTAZIONE:
1. Liste di adiacenza (se |E| é << di $|V|^2$)
2. Matrici di adiacenza (se |E| é prossimo a $|V|^2$)

1) Liste di adiacenza
Consiste in un array Adj di |V| liste, una per ogni vertice in V. Per ogni $u \in V$ la lista Adj[u] include tutti i vertici v tali che esiste in arco $(u,v) \in E$. Ovvero Adj[u] include tutti i vertici adiacenti a u in G.
Se G è un grafo orientato la somma di tutte le liste di adiacenza è |E|.
Se G è un grafo non orientato la somma di tutte le liste di adiacenza è 2|E|.
La quantità di memoria richiesta è $\Theta(V+E)$.

SVANTAGGIO:
Non c'è un modo più veloce per determinare se un particolare arco(u,v) è presente nel grafo che cercare V nella lista di adiacenza di Adj[u]. La soluzione sarebbe quella di usare matrici di adiacenza, ma al costo di usare una maggiore quantità di memoria.

2)Matrice di adiacenza
I vertici devono essere numerati 1,2...|V| in modo arbitrario.

$A_{ij} = a_{ij}$ di dimensioni |V| x |V| tali che : a_{ij} =
- 1 se $(i,j) \in E$
- 0 negli altri casi

In un grafo non orientato $A = A^T$
In alcuni casi conviene memorizzare la matrice triangolare superiore o inferiore della matrice.
Le liste di adiacenza possono essere facilmente adattate per rappresentare i grafi pesati, cioè i grafi per i quali ogni arco ha un peso associato. La funzione peso è $w: E \to R$

BFS - VISITA IN AMPIEZZA DEI GRAFI (sia grafi orientati che non)

S sorgente, la visita in ampiezza ispeziona siatematicamente gli archi di G per "scoprire" tutti i vertci che sono raggiungibili da S. Calcola la distanza(il minimo n° di archi) da S a ciascun vertice raggiungibile.
Per ogni vertice V raggiungibile da S, il cammino nell'albero BF che va da S a V corrisponde a un "cammino minimo" da S a V in G cioè un percorso che contiene il minor numero di archi.

Perchè visita in ampiezza?
Perché espande la frontiera fra i vertici scoperti e quelli da scoprire in maniera uniforme lungo l'ampiezza della linea di frontiera. L'algoritmo scopre tutti i vertici che si trovano a distanza k da S, poi quelli a distanza k+1, poi k+2....

Come fa?
Con dei marcatori: BIANCO (nodo non scoperto), GRIGIO (nodo scoperto), NERO (nodo scoperto con nodi adiacenti scoperti).
- Inizialmente tutti i vertici sono BIANCHI, dopo possono diventare grigi e poi neri.
- Un vertice si dice **scoperto** quando viene incontrato per la prima volta durante la visita.

- Se $(u,v) \in E$ e il vertice u è nero, allora il vertice v è grigio oppure nero. Tutti i vertici adiacenti a un vertice nero sono stati scoperti.
- I vertici grigi possono avere qualche vertice bianco adiacente, essi rappresentano la frontiera tra i vertici scoperti e quelli da scoprire.
- Quando un vertice bianco v viene scoperto durante l'ispezione della lista di adiacenza di un vertice u già scoperto , il vertice v e l'arco (u,v) vengono aggiunti all'albero.
- Il vertice u è detto **precedessore o padre** di v , e si indica con $\pi(v)$. Poichè ogni vertice può essere scoperto una sola volta, ogni vertice ha al più un padre.
- Se u è lungo il cammino cha va dalla sorgente al vertice v , allora u è un **antenato** di v e v è un **discendente** di u.

Analisi costi
Le operazioni di inserimento e cancellazione della coda richiedono un tempo **O(1)**, quindi il tempo dedicato alle operazioni con la coda Q(che contiene la lista dei nodi grigi) è **O(V)**.
Ogni lista di adiacenza viene ispezionata al più una volta. Poichè la somma delle lunghezze di tutte le liste di adiacenza è $\Theta(E)$, il tempo totale impiegato per ispezionare le liste di adiacenza è O(E).
Quindi il tempo di esecuzione di BFS è lineare con O(V) + O(E) $\Rightarrow$ **O(V+E)** ovvero *il tempo per gestire la coda ed ispezionare le liste*.

Alberi di visita in ampiezza
La procedura BFS costruisce un albero BF mentre visita il grafo. L'albero è rappresentato dal campo π in ciascun vertice .
Formalmente il **sottografo dei precedessori** di G è
$G_\pi = (V_\pi, E_\pi)$
dove $V_\pi = \{ v \in V : \pi[v] \neq NIL \} \cup \{s\}$ e $E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$
Il sottografo dei precedessori G_π è un albero BF se V_π è formato dai vertici raggiungibili da s e, per ogni $v \in V_\pi$, c'è un solo cammino semplice da s a v in G_π che è anche un cammino minimo da s a v in G.

DFS - VISITA IN PROFONDITA' DEI GRAFI

L'obbiettivo dell'algoritmo è quello di visitare il grafo sempre più in "profindità" possibile.
- Gli archi vengono ispezionati a partire dall'ultimo vertice scoperto che ha ancora archi non ispezionati.
- Quando tutti gli archi v sono stati ispezionati, la visita fa "marcia indietro" per ispezionare gli archi che escono dal vertice dal quale v era stato scoperto. Tutto ciò continua finchè non saranno scoperti tutti i vertici che sono raggiungibili dal vertice sorgente originale.
- Se restano dei vertici non scoperti, allora uno di essi viene selezionato come nuovo vertice sorgente e la visita riparte da questa nuova sorgente. L'intero processo viene ripetuto finchè non saranno scoperti tutti i vertici del grafo.
- Quando un vertice v viene scoperto durante un'ispezione della lista di adiacenza di un vertice u già scopeto, la visita in profondità registra questo evento assegnando u al campo $\pi(v)$ del predecessore di v .
- A differenza della visita in ampiezza, il cui sottografo dei precedessori prodotto forma un albero, la visita in profondità genera sottografo dei precedessori che può essere formato da più alberi, perché la visita può essere ripetuta da più sorgenti.

Sottografo dei predecessori $G_\pi = (V_\pi, E_\pi)$ dove $E_\pi = \{(\pi[v], v) : v \in V \wedge \pi[v] \neq NIL\}$

Il sottografo dei predecessori di una visita forma un a **foresta DF** composta da vari **alberi DF**. Gli archi in E_π sono detti **archi d'albero.**

- Inizialmente tutti i vertici sono bianchi
- Un vertice diventa grigio quando viene scoperto durante la visita, diventa nero quando completato (lista adiacenza completamente ispezionata).
- Questa tecnica garantisce che ogni vertice vada a finire in un solo albero DF, in modo che questi alberi siano disgiunti.

ATTRIBUTI:

- d[v] registra il momento in cui il vertice v viene scoperto
- f[v] registra il momento in cui la visita completa l'ispezione della lista di adiacenza del vertice v
- d[v]<f[v]

ALGORITMO:

1. DFS(G):Colora tutti i nodi del grafo di bianco e inizializza il predecessore $\pi(v)$ a NIL. Per ogni nodo bianco viene effettuata una DFS-Visit (si determineranno qui le varie sorgenti che creeranno i vari alberi DF).
2. DFS-Visit(u): visita il nodo u che viene colorato di grigio e registra in d[u] il momento in cui il nodo è scoperto . Scorre l'array Adj[u] e per ogni nodo non bianco si applica la DFS-Visit. Dopo che l'array è stato scandito il nodo u che viene colorato di nero e si registra in f[u] il momento in cui è stata completata l'ispezione del nodo.
3. La DFS, se trova ancora nodi bianchi nel grafo li userà come nuova sorgente ed eseguirà la DFS-Visit su di essa (ripartendo dal punto 1).

PROPRIETA'

1. Il sottografo dei precedessori G_π forma una foresta di alberi che rispecchia esattamente la struttura delle chiamate ricorsive di DFS-Visit. Ovvero $u = \pi(v)$ se e solo se la DFS-Visit è stata chiamata durante una visita della lista di adiacenza di u . In aggiunta v è un discendente del vertice u nella foresta DF se e soltando se v viene scoperto durante il periodo in cui u è grigio.
2. Tempi di scoperta. Si può rappresentare la scoperta di un vertice con una parentesi aperta "(u" e il suo completamento con una parentesi chiusa "u)". La storia delle scoperte e dei completamenti produce un'espressione ben formata, nel senso che le parentesi sono oppurtunamente annidate.

CLASSIFICAZIONE DEGLI ARCHI:

1. **Archi d'albero**: sono gli archi nella foresta DF G_π. L'arco (u,v) è un arco d'albero se v viene scoperto la prima volta durante l'esplorazione di (u,v).
2. **Archi all'indietro**: sono quegli archi(u,v) che collegano un vertice u a un antenato v in un albero DF. I cappi, che possono presentarsi nei grafi orientati, sono considerati archi all'indietro.
3. **Archi in avanti**: sono gli archi(u,v)(diversi dagli archi d'albero) che collegano un vertice u a un discendente v in un albero DF.
4. **Archi trasversali**: tutti gli altri archi. Possono contenere i vertici nello stesso albero DF, purchè un vertice non sia antenato dell'altro, oppure possono connettere vertici di alberi DF differenti.

ORDINAMENTO TOPOLOGICO

- Un ordinamento topologico di un dag (direct acyclic graph) è un ordianmento linerare di tutti i suoi vertici t.c., se G contiene un arco(u,v) allora u appare prima di v nell'ordinamento.
- I grafi orientati aciclici sono utilizzati per indicare la precedenza tra eventi. Un arco orientato (u,v) indica che l'indumento u deve essere indossato prima dell'indumento v.
- Un grafo orientato G è aciclico se e solo se una DFS non genera archi all'indietro.

Topological-Sort: usa la DFS per calcolare i tempi di completamento f[v] per ogni vertice;
una volta completata l'ispezione di un vertice, lo inserisce in una lista concatenata che viene ritornata. Complessità: $\Theta(V+E)$.

COMPONENTI FORTEMENTE CONNESSE

Una componente fortemente connessa di un grafo orientato G=(V,E) è un insieme massimale di vertici $C \subseteq V$ tale che per ogni coppia di vertici u e v in C , si ha u->v e v->u ovvero i vertici u e v sono raggiungibili l'uno dall'altro .

Un algoritmo per trovare le componenti fortemente connesse di un grafo G=(V,E) utilizza il grafo trasposto di G , $G^T(V,E^T)$ dove $E^T = \{(u,v):(v,u) \in E\}$.

[*Funzionamento***]**
Algoritmo SCC:

1. Per ciascun vertice u di G chiama DFS per calcolare f[u]
2. Calcola G^T e
3. Chiama DFS(G^T), ma nel ciclo principale di DFS, considera i vertici in ordine decrescente rispetto ai tempi f[u].
4. Genera l'output dei vertici di ciascun albero della foresta DF che è stata prodotta nel punto 3 come componente fortemente connessa e distinta.

[*Complessità***]**
Tempo di esecuzione : $\Theta(V+E)$.

- Alberi di connessione minimi

In un grafo non orientato G=(V,E) dove a ogni arco (u,v)∈E è dato un **peso w(u,v)** che specifica il costo per collegare u e v , vogliamo trovare un sottoinsieme aciclico T⊆E che collega tutti i vertici il cui peso totale $w(T) = \sum_{(u,v)\in T} w(u,v)$ sia minimo.
Poiché T è aciclico e collega tutti i vertici , deve anche formare un albero , che è detto **albero di connessione** perché "connette" il grafo G . Il problema di trovare l'albero T è detto **problema dell'albero di connessione minimo** .

L'algoritmo generico che crea un albero di connessione minimo usa una strategia golosa che fa crescere l'albero di connessione minimo di un arco alla volta . L'algoritmo gestisce un insieme di archi A , conservando la seguente invariante di ciclo : "Prima di ogni iterazione , A è un sottoinsieme di qualche albero di connessione minimo" . A ogni passaggio si determina un arco (u,v) che può essere aggiunto ad A senza violare l'invariante . Tale arco è detto *arco sicuro* per A .

La parte difficile di quest'algoritmo e di trovare l'arco sicuro . Un ***taglio*** (S,V-S) di un grafo orientato G = (V,E) è una partizione di V . Si dice che un arco (u,v) ∈ E ***attraversa*** il taglio (S,V-S) se una delle sue estremità si trova in S e l'altra in V-S . Si dice che un taglio ***rispetta*** un insieme A di archi se nessun arco di A attraversa il taglio . Un arco che attraversa un taglio è un ***arco leggero*** se il suo peso è il minimo fra i pesi degli altri archi che attraversano il taglio.

Teorema
Sia G = (V,E) un grafo connesso non orientato con una funzione peso w a valori reali definita in E . Sia A un sottoinsieme di E che è contenuto in qualche albero di connessione minimo per G , sia (S,V-S) un taglio qualsiasi di G che rispetta A e sia (u,v) un arco leggero che attraversa (S,V-S) . Allora , l'arco (u,v) è sicuro per A .

Corollario
Sia G=(V,E) un grafo connesso non orientato con una funzione peso w a valori reali definita in E. Sia A sottoinsieme di E che è contenuto in qualche albero di connessione minimo per G sia C =(V_C,E_C) una componente connessa nella foresta $G_A = (V,A)$.

Se (u,v) è un arco leggero che collega C a qualche altra componente in G_A , allora (u,v) è sicuro per A .
Durante l'esecuzione dell'algoritmo , l'insieme A è sempre aciclico , altrimenti un albero di connessione minimo che include A conterrebbe un ciclo , e ciò sarebbe una contraddizione . I V-1 archi di un albero di connessione minimo vengono determinati una dopo l'altro . Inizialmente A = $\varnothing$ ci sono V alberi in G_A e ogni iterazione riduce questo numero di 1 . Quando la foresta contiene un albero soltanto l'algoritmo termina .

Algoritmo di Kruskal : L'algoritmo di Kruskal si basa direttamente sull'algoritmo generico . Trova un arco sicuro da aggiungere alla foresta in costruzione scegliendo , fra tutti gli archi che collegano due alberi qualsiasi nella foresta , un arco (u,v) di peso minimo . Se indichiamo con C_1 e C_2 i due alberi che sono collegati da (u,v) , l'arco deve essere un arco leggero che collega C_1 a qualche altro albero , implica che (u,v) è un arco sicuro per C_1 .

L'algoritmo di Kruskal è un algoritmo goloso perché a ogni passaggio aggiunge alla foresta un arco con il minor peso possibile .

L'algoritmo di Kruskal è simile all'algoritmo che calcola le componenti connesse . Si usa una struttura dati per insiemi disgiunti per mantenere vari insiemi disgiunti di elementi . Ogni insieme contiene i vertici di un albero della foresta corrente . L'operazione Find-Set(u) restituisce un rappresentante dell'insieme che contiene u . Quindi possiamo determinare se i due vertici u e v appartengono allo stesso albero . L'unione degli alberi è effettuata dalla procedura Union .
Gli archi in E vengono ordinati in senso decrescente rispetto al peso . Se le estremità u e v appartengono allo stesso albero , l'arco (u,v) non può essere aggiunto alla foresta senza generare un ciclo , quindi l'arco viene scartato , altrimenti i due vertici appartengono ad alberi differenti e l'arco (u,v) viene aggiunto ad A e vengono fusi dalla procedura Union , questo per ogni u e v .
Il tempo di esecuzione dell'algoritmo di Kruskal per un grafo G=(V,E) dipende dall'implementazione della struttura dati per gli insieme disgiunti:
V Make-Set + ordinamento degli archi + E Find-Set + V Union
Usando le euristiche di unione per rango e osservando che $E<V^2$ si ha $lgE = O(lgV)$ il costo è: $O(ElgE)+O((V+E)\alpha(V))=O(ElgV)$.

Algoritmo di Prim : L'algoritmo di Prim è un caso speciale dell'algoritmo generico per gli alberi di connessione minimi . L'algoritmo di Prim opera in modo simile all'algoritmo di Dijkstra per trovare cammini minimi in un grafo . L'algoritmo di Prim ha la proprietà che gli archi dell'insieme A formano sempre un albero singolo . L'albero inizia da un arbitrario vertice radice r e si sviluppa fino a coprire tutti i vertici in V . A ogni passaggio viene aggiunto all'albero A un arco leggero che collega A con un vertice isolato di $G_a=(V,A)$. Questa regola aggiunge soltanto archi sicuri per A cosicché che quando l'algoritmo termina , gli archi di A formano un albero di connessione minimo .

Questa strategia è golosa perché l'albero cresce includendo a ogni passaggio un arco che contribuisce con la quantità più piccola *possibile a formare il peso dell'albero .*

La chiave per implementare con efficienza l'algoritmo di Prim consiste nel semplificare la scelta di un nuovo arco da aggiungere . Durante l'esecuzione dell'algoritmo tutti i vertici che non si trovano nell'albero risiedono in una coda di min-priorita Q basata su un campo key . Per ogni vertice v , key[v] p il peso minimo di un arco qualsiasi che collega v a un vertice nell'albero . Il campo π(v) indica il padre di v nell'albero .
Durante l'esecuzione dell'algoritmo l'insieme A è mentenuto implicitamente come :
A={(v,π(v):v$\in$V-{r}-Q)}
Quando l'algoritmo termina , la coda di min-priorita Q è vuota , l'albero di connessione minimo A per G è quindi : A={(v,(v):v$\in$V-{r})}

L'algoritmo conserva la seguente invariante di ciclo :

1. A={(v,π(v):v$\in$V-{r}-Q)}
2. I vertici già inseriti nell'albero di connessione minimo sono quelli che appartengano a V-Q
3. Per ogni vertice v$\in$Q, se π(v)$\neq$NIL , allora key[v] < ∞ e key[v] è il peso di un arco leggero (v,π(v)) che collega v a qualche vertice che si trova già nell'albero di connessione minimo .

Costi: V(Extract-Key) + E(Reduce-Key)

Dipendono dal modo in cui viene implementata la coda di min-priorita Q . Se Q è implementata come un min-heap binario il tempo totale dell'algoritmo di Prim è $O(VlgV+ElgV)=O(ElgV)$ che è asintoticamente uguale a quello di Kruskal.

Tale implementazione va bene se il grafo è sparso, nel caso di un grafo completo sono più vantaggiose le liste non ordinate: $O(V^2+E)$.

Se utilizziamo gli heap di Fibonacci il tempo di esecuzione migliora diventando $O(E+VlgV)$.

- Cammini minimi da sorgente unica

In un problema dei cammini minimi è dato un grafo orientato pesato G=(V,E) , con una funzione peso w:E->R che associa gli archi a pesi di valore reale . Il peso del cammino p=<v_0,v_1,...,v_k>
è la somma dei pesi degli archi che lo compongono :
$w(p)=\sum_{i=1}^{k} w(v_{i-1},v_i)$.

Il **peso di un cammino minimo** da u a v è definito in questo modo :
$\delta(u,v)=min\{w(p):u\rightarrow v\}$ *se esiste un cammino da u a v* oppure ∞ *negli altri casi .*

Un **cammino minimo** dal vertice u al vertice v è definito come un cammino qualsiasi p con peso w(p)=$\delta(u,v)$.

Molti problemi possono essere risolti con l'algoritmo per i cammini minimi da sorgente unica :
problema dei cammini minimi con destinazione unica , *problema del cammino minimo per una coppia di vertici* , *problema dei cammini minimi fra tutte le coppie di vertici* .

Sottostruttura ottima di un camino minimo : Gli algoritmi per i cammini minimi si basano sulla proprietà che un cammino minimo fra due vertici contiene altri cammini minimi al suo interno.

Lemma : *Dato un grafo orientato pesato G = (V,E) con la funzione w:E->R sia* p=<v_1,v_2,...,v_k> un cammino minimo dal vertice v_1 al vertice v_k e per qualsiasi i e j tali che $1\leq i\leq j\leq k$, sia p_{ij}=<v_i,v_{i+1},...,v_j> il sottocammino di p dal vertice v_i al vertice v_j . Allora p_{ij} è un cammino minimo da v_i al vertice v_j .

Archi di peso negativo e cicli : Se il grafo G=(V,E) non contiene cicli di peso negativo che sono raggiungibili dalla sorgente s , allora per ogni $v\in V$ il peso del cammino minimo $\delta(s,v)$ resta ben definito , se invece esiste un ciclo di peso negativo che è raggiungibile da s , i pesi dei cammini minimi non sono ben definiti . Se èsiste un ciclo di peso negativo in qualche cammino da s a v definiamo $\delta(s,v)=-\infty$. Un cammino minimo non può contenere un ciclo di peso positivo o negativo . Se un cammino minimo contiene cicli di peso 0 è possibile eliminare ripetutamente questi cicli dar cammino fino a ottenere un cammino minimo senza questi cicli .

Dato un grafo G=(V,E) manteniamo per ogni vertice $v\in V$ un predecessore $\pi[v]$ che può essere un altro vertice o NIL . Gli algoritmi per i cammini minimi impostano gli attributi π in modo che la catena dei predecessori che inizia da un vertice v percorra all'indietro il cammino minimo da s a v . $G_\pi=(V_\pi,E_\pi)$ è il **sottografo dei predecessori** indotto dai valori di π . Definiamo l'insieme dei vertici V_π come l'insieme dei vertici di G con predecessori non NIL più la sorgente . L'insieme degli archi orientati E_π è l'insieme degli archi indotto dai valori π per i vertici in V_π . Si può dimostrare nei

algoritmi di Bellman-Ford e Dijkstra alla fine della esecuzione G_π è **un albero di cammini minimi** con la radice in s .

Rilassamento *:* Il processo di rilassamento di un arco (u,v) consiste nel verificare se , passando per u, è possibile migliorare il cammino minimo per v precedentemente trovato e , in caso affermativo nell'aggiornare i valori d[v] e $\pi[v]$. Il rilassamento è l'unico modo per modificare i predecessori e le stime dei cammini minimi . L'attributo d[v] è detto stima del cammino minimo , che è un limite superiore per il peso di un cammino minimo dalla sorgente s a v .

Proprietà dei cammini minimi e del rilassamento :

- Disuguaglianza triangolare : Per qualsiasi arco $(u,v) \in E$ si ha $\delta(s,v) \leq \delta(s,u) + w(u,v)$
- Limite superiore : Per tutti i vertici $v \in V$, si ha sempre $\delta(s,v) \leq d[v]$ e una volta che d[v] assume il valore di $\delta(s,v)$ esso non cambia più .
- Assenza del cammino : Se non c'è un cammino da s a v , allora si ha sempre $d[v] = \delta(s,v) = \infty$
- Convergenza : Se s -> u -> v è un cammino minimo in G per qualche $u,v \in V$ e se $d[u] = \delta(s,u)$ in un istante qualsiasi prima del rilassamento dell'arco (u,v) allora$d[u] = \delta(s,u)$.
- Rilassamento del cammino : Se p=<v_0,v_1,...,v_k> è un cammino minimo da $s = v_0$ a v_k e gli archi di p vengono rilassati nell'ordine (v_0,v_1),(v_1,v_2),...,(v_{k-1},v_k) allora $d[v_k] = \delta(s,v)$. Questa proprietà è soddisfatta indipendentemente da altri passi di rilassamento che vengono effettuati , anche se sono interposti fra i rilassamenti degli archi di p .
- Sottografo dei predecessori : Una volta che $d[v] = \delta(s,v)$ per ogni $v \in V$, il sottografo dei predecessori è un albero di camini minimi radicato in s .

- L'Algoritmo di Bellman-Ford : Dato un grafo orientato G=(V,E) con sorgente s e una funzione w:E->R l'algoritmo di Bellman-Ford restituisce un booleano che indica se esiste o no un ciclo di peso negativo che è raggiungibile dalla sorgente , se esiste l'algoritmo indica che il problema non ha soluzioni , altrimenti l'algoritmo fornisce i cammini minimi e i loro pesi .

L'algoritmo usa il rilassamento , riducendo progressivamente il valore stimato d[v] , per il peso di un cammino minimo dalla sorgente s a ciascun vertice $v \in V$, fino a raggiungere il peso effettivo del cammino minimo $\delta(s,v)$.

[Funzionamento**]**

Dopo l'inizializzazione dei valori dei valroi d π di tutti i vertici , l'algoritmo effettua V-1 passagi sugli archi del grafo . Ogni passaggio consiste nell'effettuare un rilassamento per ciascun arco del grafo . Dopo avere effettuato V-1 passaggi si controlla se esiste un ciclo di peso negativo e si restituisce il valore booleano appropriato .

[Complessità**]**

Tempo di esecuzione : $O(VE)$, dovuto a l'inizializzazione che richiede un tempo $\Theta(V)$ e ciascuno dei V-1 passaggi sugli archi richiede un tempo $\Theta(E)$ più il controllo $O(E)$.

-L'algoritmo di Dijkstra : L'algoritmo di Dijkstra risolve il problema dei cammini minimi dalla sorgente unica s in un grafo orientato pesato G=(V,E) *nel caso in cui tutti i pesi degli archi non siano negativi* . L'algoritmo di Dijkstra mantiene un insieme S di vertici i cui pesi finali dei cammini minimi dalla sorgente s sono stati già determinati . L'algoritmo seleziona ripetutamente il vertice $u \in V - S$ con la stima minima del cammino minimo , aggiunge u a S e rilassa tutti gli archi che escono da u . Nella implementazione di Dijkstra si mantiene una coda di min-priorita Q di vertici utilizzando come chiavi i loro valori d .

[Funzionamento**]**

Dopo l'inizializzazione dei valori d e π , si inizializza l'insieme S come l'insieme vuoto e la coda di min-priorita Q in modo che questa contenga tutti i vertici V . L'algoritmo

mantiene l'invariante Q = V-S all'inizio di ogni iterazione . A ogni iterazione un vertice u viene estratto da Q = V-S e aggiunto all'insieme S conservando l'invariante . Il vertice u ha la stima minima del cammino minimo di tutti i vertici V-S . Dopodiché si rilassa ogni i arco (u,v) che esce da u e aggiornano la stima d[v] e il predecessore $\pi[v]$, se il cammino minimo che arriva a può essere migliorato passando per u .

[Caratteristiche**]**

Siccome l'algoritmo di Dijkstra sceglie sempre il vertice più leggero o più vicino in V-S da aggiungere all'insieme S , *diciamo che applica una strategia golosa* . Le strategie golose non forniscono sempre i risultati ottimali , l'algoritmo di Dijkstra calcola davvero i cammini minimi . Il punto chiave consiste nel dimostrare che ogni volta che un vertice u viene aggiunto all'insieme S , si ha $d[u]=\delta(s,u)$.

[Complessità**]**

Il tempo di esecuzione dell'algoritmo di Dijkstra dipende dal modo in cui viene implementata la coda di min-priorita .

a) Manteniamo la coda di min-priorità sfruttando il fatto che i vertici sono numerati da 1 a V . Memorizziamo semplicemente d[v] nell'elemento v-esimo di un array . Ciascuna operazione Insert , Decrese-Key impiega un tempo pari a O(1) e ciascuna operazione Extract-Min impiega un tempo O(V) . Tempo di esecuzione : $O(V^2+E)=O(V^2)$.

b) Se il grafo è sufficientemente sparso in particolare $E=O(V^2/lgV)$ conviene implementare la coda di min-priorita con un min-heap binario . Adesso ogni operazione di Extract-Min e Decrease-Key richiede un $O(lgV)$ e il tempo per costruire un min-heap binario è $O(V)$. Tempo di esecuzione : $O((V+E)lgV)=O(ElgV)$ (se tutti i vertici sono raggiungibili dalla sorgente)

c) Se invece si implementa la coda di min-priorita con un heap di Fibonacci . Il costo ammortizzato per ogni Extract-Min è $O(lgV)$ e ogni chiamata di Decrease-Key richiede un tempo costante . Tempo di esecuzione : $O(VlgV+E)$.

-Cammini minimi fra tutte le coppie :

Dato un grafo orientato pesato G=(V,E) con una funzione peso w:E->R che associa agli archi dei pesi di valore reale , vogliamo trovare , per ogni coppia di vertici $u,v\in V$, un cammino minimo da u a v , dove il peso di un cammino è la somma dei pesi degli archi che lo compongono . Vogliamo ottenere l'output in forma tabulare : l'elemento della riga u e nella colonna v rappresenta il peso di un cammino minimo dal vertice u al vertice v .

Per questa problema si potrebbe eseguire un algoritmo per cammini minimi da sorgente unica V volte una volta per ogni vertice che è utilizzato come sorgente . Se tutti i pesi sono positivi possiamo utilizzare l'algoritmo Dijkstra . Se implementiamo la coda di min priorita come :

a) un array lineare il tempo di esecuzione è : $O(V^3+EV)=O(V^3)$

b) un min-heap binario il tempo di esecuzione è : $O(VElgV)$

c) un heap di Fibonacci il tempo di esecuzione è : $O(V^2lgV+VE)$

Quando invece i pesi negativi sono ammessi bisogna eseguire l'algoritmo di Bellman-Ford una volta per ogni vertice .Il tempo di esecuzione risulta $O(V^2E)$ che per un grafo denso è $O(V^4)$.

L'input degli algoritmi che risolvano il problema dei cammini minimi fra tutte le coppie è una matrice $W_{n\times n}$, che rappresenta i pesi degli archi di un grafo orientato G=(V,E) di n vertici .

L'output di forma tabulare è una matrice $D_{n\times n}=d_{ij}$ dove l'elemento d_{ij} contiene il peso di un cammino minimo dal vertice i al vertice j . Bisogna calcolare anche la matrice dei predecessori $\prod=\pi_{ij}$ dove π_{ij} è NIL se i=j o se non esiste un cammino da i a j , altrimenti π_{ij} è il predecessore di j in qualche cammino minimo da i .

Cammini minimi e moltiplicazione di matrici : Un algoritmo di programmazione dinamica per il problema dei cammini minimi fra tutte le coppie di vertici in un grafo orientato che usa un operazione molto simile alla moltiplicazione delle matrici .

La struttura di un cammino minimo : Supponiamo che il grafo sia rappresentato da una matrice di adiacenza $W=(w_{ij})$. Consideriamo un cammino minimo p dal vertice i al vertice j e supponiamo che p contenga m archi (non ci sono cicli di peso negativo) . Se i = j allora p ha peso 0 e non ha archi altrimenti scomponiamo il cammino p in i->k->j , dove il cammino p' rappresenta il cammino da i a k e contiene m-1 archi . p' è un cammino minimo da i a k quindi $\delta(i,j)=\delta(i,k)+w_{kj}$.

Una soluzione ricorsiva : Sia l_{ij}^{m} il peso minimo di un cammino qualsiasi dal vertice i al vertice che contiene al più m archi . Quando m=0 , esiste un cammino minimo da i a j senza archi se e soltanto se i=j altrimenti $l_{ij}^{0}=\infty$. Per $m\geq 1$ calcoliamo l_{ij}^{m} come il minimo tra l_{ij}^{m-1} e il peso minimo di un cammino qualsiasi da i a j che è formato al più da m archi , che si ottiene considerando tutti i possibili predecessori k di j . Quindi la definizione ricorsiva è : $l_{ij}^{(m)}=min_{1\leq k\leq n}\left\{l_{ik}^{(m-1)}+w_{kj}\right\}$.

Calcolo dei pesi dei cammini minimi secondo lo schema bottom-up : Il cuore dell'algoritmo è la procedura che : date le matrici $L^{(m-1)}$ e W , restituisce la matrice $L^{(m)}$. La procedura aggiunge un arco ai cammini minimi calcolati fino a quel momento calcolando una matrice $L'=(l'_{ij})$ e la restituisce alla fine , ciò viene fatto calcolando : $l_{ij}^{(m)}=min_{1\leq k\leq n}\left\{l_{ik}^{(m-1)}+w_{kj}\right\}$.

[Complessità**]**
Tempo di esecuzione : $\Theta(n^4)$. Che può essere migliorato sapendo che siamo interessati soltanto alla matrice L^{n-1} e il fatto che anche la moltiplicazione di questo algoritmo è associativa . Possiamo calcolare L^{n-1} in $lg(n-1)$ prodotti di matrici . In questa maniera il tempo di esecuzione diventa : $\Theta(n^3 lgn)$.

Algoritmo di Floyd-Warshall : L'algoritmo di Floyd-Warshall usa il processo della programmazione dinamica . In questo algoritmo si suppone che non ci siano cicli di peso negativo , ma possono essere presenti archi di peso negativo .

La struttura di un cammino minimo : L'algoritmo considera i vertici "intermedi" di un cammino minimo , dove un **vertice intermedio** di un cammino semplice $p=<v_1,v_2,...,v_l>$ è un vertice qualsiasi di p diverso da v_1 e v_l . Supponendo che i vertici di G siano V={1,2,...,n} , consideriamo un sottoinsieme {1,2,...,k} di vertici per un generico k . Per una coppia qualsiasi di vertici $i,j\in V$, consideriamo tutti i cammini da i a j i cui vertici intermedi sono tutti in {1,2,...,k} e sia p un cammino di peso minimo fra di essi . L'algoritmo di Floyd-Warshall sfrutta una relazione fra il cammino p e i cammini minimi da i a j i cui vertici intermedi appartengono tutti all'insieme {1,2...,k-1} . Tale relazione dipende dal fato che k sia un vertice intermedio del cammino p oppure no .

- Se k non è un vertice intermedio del cammino p , allora tutti i vertici intermedi del cammino p sono nell'insieme {1,2,...,k-1} . Quindi un cammino minimo dal vertice i al vertice j con tutti i vertici intermedi nell'insieme {1,2,...,k-1} è anche un cammino minimo da i a j con tutti i vertici intermedi nell'insieme {1,2,...,k } .
- Se k è un vertice intermedio del cammino p , allora spezziamo p in i->k->j dove p_1 è il cammino da i a k e p_2 è il cammino da k a j . p_1 è un cammino minimo da i a k con tutti i vertici intermedi nell'insieme {1,2,...,k} . Poiché k non è un vertice intermedio del cammino p_1 , allora p_1 è un cammino minimo da i a k con tutti i vertici intermedi nell'insieme {1,2,...,k-1} . Analogamente per p_2 .

Una soluzione ricorsiva : Sia $d_{ij}^{(k)}$ il peso di un cammino minimo dal vertice i al vertice j , i cui vertici intermedi si trovano tutti nell'insieme {1,2,...,k}. Se k=0 , un cammino dal vertice i al vertice j , che non include vertici intermedi con numerazione maggiore di 0 , non ha alcun vertice intermedio . Un tale cammino ha al massimo un arco , e quindi : $d_{ij}^{(0)} = w_{ij}$. Per k > 0 si ha :

$d_{ij}^{(k)} = min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$.

Calcolo secondo lo schema bottom-up : L'input è $W_{n\times n}$ e l'output è $D^{(n)}$ dei pesi dei cammini minimi .

[Complessità**]**

Tempo di esecuzione : $\Theta(n^3)$ (e la costante nascosta e molto piccola dato che l'algoritmo non usa strutture dati complesse) . L'algoritmo di Floyd-Warshall è molto utile anche per grafi di input di discrete dimensioni .

Algoritmo di Johnson per i grafi sparsi : L'algoritmo di Johnson è molto utile per i grafi sparsi , è asintoticamente migliore di Floyd-Warshall per questo tipo di grafi . L'algoritmo di Johnson usa come subroutine sia l'algoritmo di Djkstra sia l'algoritmo di Bellman-Ford , e assume che gli archi siano memorizzati in una lista di adiacenza . L'algoritmo di Johnson usa la tecnica del ricalcolo dei pesi . Se tutti i pesi degli archi di w in un grafo G=(V,E) non sono negativi , possiamo trovare i cammini minimi fra tutte le coppie di vertici usando l'algoritmo di Dijkstra per ogni vertice e un heap di Fibonacci . Se G ha archi di peso negativo , ma è privo di cicli di peso negativo , calcoliamo semplicemente un nuovo insieme di pesi di archi non negativi che ci consente di utilizzare lo stesso metodo . Questo nuovo insieme di pesi $\widetilde{w}$ deve soddisfare due proprietà :

1. Per ogni coppia di vertici $u,v \in V$, un cammino p è un cammino minimo da u a v con la funzione di peso w, se e soltanto se p è anche un cammino minimo da u a v con la funzione di peso $\widetilde{w}$.
2. Per ogni arco (u,v) il nuovo peso $\widetilde{w}(u,v)$ non è negativo .

Dato un grafo orientato G=(V,E) con funzione peso w:E->R sia h:V->R una funzione qualsiasi che associa i vertici a numeri reali . Per ogni arco $(u,v) \in V$ definiamo :

$\widetilde{w}(u,v) = w(u,v) + h(u) - h(v)$.

Questa funzione $\widetilde{w}$ soddisfa entrambe le proprietà .

L'algoritmo di Johnson restituisce la solita matrice $D = d_{ij}$ oppure segnala che il grafo di input contiene un ciclo di peso negativo . Johnson usa Bellman-Ford per vedere se esistono cicli di peso negativo se no calcola i nuovi pesi $\widetilde{w}$ e per ogni vertice chiama Dijkstra .

[Complessità**]**

Tempo di esecuzione :

a) con heap di Fibonacci : $O(V^2 lgV + VE)$.

b) con un min-heap binario : $VElgV$.

Entrambe le complessità sono migliori di Floyd-Warshall per i grafi sparsi .

- Flusso Massimo

Una rete di flusso può essere vista come un condotto per far "scorrere" materiale dalla sorgente alla destinazione ad una certa velocità.

Arco: condotto(che non ha una capacità prefissata) dove scorre materiale.

Vertci: sono i giunti dei condotti.

La velocità con la quale il materiale entra in un vertice deve essere uguale alla velocità con la quale esce dal vertice.

Una rete di flusso G=(V,E) è un grafo orientato dove ogni arco (u,v) ha capacità non negativa $c(u,v) >= 0$

Se $(u,v) \notin E$ allora $c(u,v) = 0$

Ogni vertice giace in qualche cammino dalla sorgente al pozzo. Ovvero esiste un qualche cammino dalla sorgente al pozzo. Il grafo è quindi connesso con $|E| \succeq |V| - 1$.
Il valore di flusso f si indica con $f(u,v)$, è definito come $|f| = \Sigma_{v \in V} f(s,v)$ ovvero il flusso totale che esce dalla sorgente.

PROPRIETA':
Un flusso in G è una funzione $f: V \times V \to R$ che soddisfa le seguenti proprietà:

1. Vicolo sulla capacità: per ogni $u,v \in V$ si richiede che $f(u,v) \preceq c(u,v)$.
2. Antisimmetria: per ogni $u,v \in V$ si richiede che $f(u,v) = -f(v,u)$.
3. Conservazione del flusso: per ogni $u \in V - \{s,t\}$ si richiede che $\Sigma_{v \in V} f(u,v) = 0$.

CONSIDERAZIONI:

- L'antisimmetria è una notazione comoda che dice che il flusso da un vertice u ad un vertice v è l'opposto del flusso nella direzione inversa.
- La proprietà di conservazione del flusso dice che il flusso totale che esce da un vertice diverso dalla sorgente o dal pozzo è 0.
- Se nè (u,v) nè (v,u) si trovano in E non può esserci un flusso tra u e v quindi f(u,v)=f(v,u)=0
- Flusso totale positivo che entra in un vertice v è definito da $\Sigma_{u \in V, f(u,v) > 0} f(u,v)$

Reti con più sorgenti e pozzi : Quando abbiamo più sorgenti e pozzi il problema non è più difficile di un normale problema di flusso massimo . In questo caso si aggiunge una supersorgente s e un arco orientato (s,s_i) con capacita $c(s,s_i) = \infty$ per ogni i = 1,2,...,n . Dodo si può aggiungere un superpozzo t e un arco orientato (t_i,t) con capacita $c(t,t_i) = \infty$ per ogni i = 1,2,...,n .

Il metodo di Ford-Fulkerson : Include varie implementazioni con tempi di esecuzione differenti . Il metodo di Ford-Fulkerson dipende da tre importanti idee che trascendono il metodo stesso e sono rilevanti per molti algoritmi e problemi di flusso : reti residue , cammini aumentanti e tagli . Il metodo di Ford-Fulkerson è iterativo .

- **Reti residue :** Data una rete di flusso con un certo flusso , la rete residua è composta dagli archi che possono accettare altro flusso . Sia f un flusso in G e consideriamo una coppia di vertici $u,v \in V$. La quantità di flusso addizionale che possiamo inviare da u a v prima di superare la capacità c(u,v) è la **capacità residua** di (u,v) data da : $c_f(u,v) = c(u,v) - f(u,v)$. Data una rete di flusso G=(V,E) con un flusso f , la rete residua di G indotta da f è $G_f = (V, E_f)$ dove $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$.
- **Cammini aumentanti :** Data un rete di flusso G=(V,E) con un flusso f , un cammino aumentante p è un cammino semplice da s a t nella rete residua G . Per la definizione di rete residua , ogni arco (u,v) in un cammino aumentante può accettare un flusso positivo addizionale da u a v senza violare il vincolo sulla capacità dell'arco . La quantità massima di cui può crescere il flusso in ogni arco di un cammino aumentante p è detta **capacità residua** di p ed è espressa da : $c_f = min\{c_f(u,v) : (u,v) \in p\}$. Definiamo inoltre la funzione $f_p : V \times V -> R$ come : $f_p(u,v) = c_f(p) : (u,v) \in p$

  $f_p(u,v) = -c_f(p) : (v,u) \in p$
  $f_p(u,v) = 0$ altrimenti

- **Tagli delle reti di flusso :** Un taglio (S,T) della rete di flusso G=(V,E) è una partizione di V in S e T = V-S tale che $s \in S$ e $t \in T$. Se f è un flusso allora il flusso netto attraverso un taglio (S,T) è definito come f(S,T) . La capacità del

taglio (S,T) è c(S,T) . Un taglio minimo di una rete è un taglio la cui capacità è minima rispetto a tutti i tagli della rete . *Il valore di un flusso qualsiasi f in una rete di flusso G è limitato superiormente dalla capacità di un taglio qualsiasi di G .*

Teorema (Teorema del flusso massimo / taglio minimo)

Se f è un flusso in una rete di flusso G=(V,E) con sorgente s e pozzo t , allora le seguenti tre condizioni sono equivalenti .

1. *f è un flusso massimo di G .*
2. *la rete residua G_f non contiene cammini aumentanti .*
3. *|f|=c(S,T) per qualche taglio (S,T) di G .*

[*Funzionamento***]**

Ogni iterazione del metodo di Ford-Fulkerson cerca un cammino aumentante p , se il cammino viene trovato , viene incrementato il flusso f in ogni arco di p della capacità residua $c_f(p)$. Il flusso massimo in un grafo G=(V,E) viene calcolato aggiornando il flusso f[u,v] fra ogni coppia u,v di vertici che sono connessi da un arco . Se u e v non sono connessi da un arco in nessuna direzione , assumeremo implicitamente che f[u,v] = 0 . Si suppone che le capacità c(u,v) vengano fornite con il grafo e che c(u,v) = 0 se $(u,v) \notin E$. La capacità residua $c_f(u,v)$ è calcolata secondo la formula : $c_f(u,v) = c(u,v) - f(u,v)$.

[*Complessità***]**

Il tempo di esecuzione di Ford-Fulkerson dipende dal modo in cui viene trovato il cammino aumentante p . Se il cammino non viene scelto bene , l'algoritmo potrebbe anche non terminare . Se il cammino aumentante viene scelto utilizzando una visita in ampiezza l'algoritmo viene eseguito in tempo polinomiale .

Tempo di esecuzione : $O(E|f^*|)$ dove f* è il flusso massimo trovato dall'algoritmo . Per assicurare questo tempo dobbiamo gestire con efficienza la struttura dati utilizzata per implementare la rete G=(V,E) .