Lezione 2 – strutture dati elementari

PILA : principio Last In First Out (LIFO)
operazioni:
- stack empty (S) : true se la pila è vuota e viceversa
- Push (S,x) : aggiunge x come utimo elemnto di S
- Pop (S) : toglie l'ultimo elemento inserito

CODA : Principio Fist In First Out (FIFO)
operazioni:
- enqueue (Q,x) : aggiunge x come ultimo elemento di Q
- dequeue (Q) : toglie da Q il primo elemento inserito e lo restituisce

nozioni:
- Tail [Q] : posizione dell'ultimo elemento inserito (ogni volta che si aggiunge un elemento questa si aggiorna con la posizione dell'ultimo elemento aggiunto)
- Head [Q] : Inizio della coda (primo elemento disponibile da togliere dalla coda)

TECNICHE DI RAPPRESENTAZIONE
tipologie:
- rappresentazioni indicizzate : i dati sono contenuti all'interno di array
  - vantaggi : accesso diretto ai dati mediante indice (veloce)
  - svantaggi : dimensione fissa (riallocazione dell'array richiede tempo lineare)
- rappresentazioni collegate : i dati sono contenuti in record collegati tra loro mediante puntatori (liste)
  - vantaggi : dimensione variabile (aggiunta\rimozione record tempo costante)
  - svantaggi : accesso sequenziale ai dati (più lento)

LISTE
tipologia:
- lista concatenata : oggetti disposti in un ordine lineare determinato da un puntatore in ogni oggetto
  - campi : KEY, PREV
- lista doppiamente concatenata
  - campi : KEY, NEXT (puntatore al prossimo elemento) e PREV (puntatore all'elemento precedente)

dettagli aggiuntivi :
- ordinata : testa elemento più piccolo, coda elemento più grande
- non ordinata : ordine casuale
- circolare : PREV della testa punta alla coda, il NEXT della coda punta alla testa

operazioni:
- List-Search (L,k) : restituisce un puntatore al primo elemento con val. k nella lista L
- List-Insert (L,x) : inserisce un elemento x davanti alla lista concatenata
- List-delete (L,x) : rimuove un elemento x da L

sentinella:
elemento fittizio posto tra testa e coda

ALBERI
informazioni :

- ogni nodo ha un solo padre
- tutti i nodi con lo stesso padre sono fratelli
- ogni nodo può avere zero o più figli tali che (u,v) abbiano un arco
- il numero di figli di un nodo è detto grado di tale nodo
- un nodo senza figli è una foglia
- la profondità di un nodo è data dal numero di archi che deve attraversare per raggiungerlo
- l'altezza di un albero è la massima profondità a cui si trova una foglia

tipologie :

- albero radicato : Insieme vuoto di nodi / radice R e 0 o più sottoalberi con la radice collegata a R da un arco (orientato)
  - coppia T = (N,A) dove N corrispondono ai nodi e A agli archi

albero binario :
- ogni elemento ha tre campi:
p : memorizza il padre
left : puntatore al figlio sinistro
right : per memorizzare il puntatore al figlio destro
root[T] : punta alla radice dell'albero
P.S. p[x]=NIL ---> x è la radice

albero con array:
left-child[x] = punta al figlio più a sinistra di x ---> NIL se x non ha figli
right-sibiling[x] = punta al fratello di x immediatamente a destra ---> NIL se x è il nodo più a destra

visita degli alberi:
-in profondità : depth-first search (visitati i rami uno dopo l'altro)
-preordine : si visita prima la radice poi le chiamate ricorsive sul figlio sinistro e destro
-inordine (simmetrica) : si visita prima la chiamata ricorsiva sul figlio sinistro, poi si passa ala radice e infine chiamata ricorsiva sul figlio destro.
-postordine : si effettuano le chiamate ricorsive sul figlio sinistro, destro e poi si passa alla radice
-in ampiezza : breasth-first search (a livelli partendo dalla radice)
-unica modalità: si analizza tutti gli elementi a una determinata profondità k aumentando (ogni volta finiti gli elementi per ogni profondità k) la profondità di 1

LEZIONE 3 – ordinamento

INJECTIONSORT
-complessità:
-caso migliore : $bn + a$
-caso medio : $c''n^2 + b''n + a''$
-caso peggiore : $c'n^2 + b'n + a'$

- funzionamento:
injectionsort usa un approcio incrementale ovvero ordina i sottoarray uno per volta.
Prende un numero “a” e lo confronta con il successivo “b”, se “a” è minore lo lascia stare, viceversa se è superiore li inverte, quindi “b” prende il posto di “a”.
A questo punto accadono due cose:
- “a” viene confrontato con numero successivo “c” e si ripetono le stesse operazioni di prima
- “b” viene confrontato con il valore che lo precede, se è più piccolo li si inverte (e così fino a quando non si esauriscono i numeri) viceversa rimane dov'è.

Molti altri algoritmi sono divide et impera, seguono quindi un iter di tre step:
- divide : il problema si suddivide in sottoproblemi
- impera : i sottoproblemi vengono risolti in modo ricorsivo
- combina : le soluzioni vengono com binate per generare la soluzione del problema principale

MERGESORT
-complessità:
- caso migliore : *O(n log n)*
- caso medio *O(n log n)*
- caso peggiore : *O(n log n)*
- complessità dello spazio : *O(n)*

-funzionamento:
questo algoritmo prende la stringa di valori e la divide a metà, e la divide nuovamente a metà fino a quando non rimangono “n” stringe da un valore ciascuna, a questo punto si inizia a ricomporne due alla volta ma in modo ordinato fino a ricomporre un'unica stringa ordinata.

- confronto tra i due algoritmi studiati:
  - array quasi ordinati e sufficientemente grandi :
    per Injectionsort si dimostra migliore rispetto a mergesort
  - array di piccole dimensioni :
    Insertionsort è migliore rispetto a Mergesort

HEAPSORT
- complessità :
- caso migliore : *O(n log n)*
- caso medio *O(n log n)*
- caso peggiore : *Θ(n log n)*
- complessità dello spazio : *O(n)*

-funzionamento:
introduciamo il concetto di:
-Max-heap: per ogni nodo (ad esclusione della radice) il padre è sempre maggiore
-Max-heapify : usato per ripristinare lo stato di “max-heap” ogni vota che questa proprietà viene violata a causa di modifiche all'interno dell'array complessità : *O(log n)*
-Min-heap: per ogni nodo (ad esclusione della radice) il padre è sempre minore
vi sono vari passaggi:

-inizializzazione : considerare i figli come foglie
-conservazione :
-conclusione :

gli heap si possono utilizzare per implementare code con priorità (strutture dati in cui non è possibile immagazzinare gli oggetti con una chiave ed estrarli uno ala volta in ordine di priorità decrescente).
Vi sono due priorità:
-Max-priorità
operazioni:
- MaxheapInsert (A,x) : aggiunge x alla coda A
- heapMaximum (A) : ritorna x con chiave massima in A
- heap extractMax (A) : toglie e ritorna X con chiave massima in A
- heapIncreaseKey (A,x,key) : aumenta il valore della chiave di x

-Min-priorità

QUICKSORT
- complessità :
- caso migliore : $O(n \log n)$
- caso medio $O(n \log n)$
- caso peggiore : $\Theta(n^2)$
- complessità dello spazio : dipende dalle implementazioni

-funzionamento:
si basa sulla partizione dell'array rispetto ad un suo elemento scelto come pivot, l'operazione viene quindi ripetuta sule sue due parti così ottenute.
Utilizza il metodo del divide et impera
-divide: partiziona l'array in due sottoarray tali che ogni elemento del primo array è <= rispetto al pivot, e ogni elemento del secondo array sia > rispetto al pivot.
-impera: ordine i due sottoarray chiamando ricorsivamente quicksort
-combina: siccome i due array sono ordinati sul posto non occorre alcuna combinazione

confronto:
dipende dal partizionamento che a sua volta dipende da quali elementi vengono utilizzati per il partizionamento. Comunque:
-partizionamento bilanciato:
l'algoritmo viene eseguito con la stessa velocità di MergeSort
-partizionamento sbilanciato:
l'algoritmo può essere asintoticamente lento quanto l'InsertionSort.

COUNTING SORT
- complessità :
- caso migliore : $O(n+k)$
- caso medio : $O(n+k)$
- caso peggiore : $O(n+k)$
- complessità dello spazio : ?

-funzionamento:
necessita di due ulteriori array:
-B in cui mettere la sequenza ordinata

-C array ausiliario (segue un iter di due passaggi) :
-ogni casella dell'array ha un indice “i” e nella casella corrispondente all'indice “i” viene inserito un valore pari a quante volte compare “i” per ciascuna “i”
-una volta terminato il passaggio del conteggio avviene un ordinamento secondo il seguente principio: ogni indice “i” ha un valore pari alla somma di tutti i valori precedenti all'indice “i” in modo da sapere quante caselle dell'array dovrò saltare prima di inserire “i”

RADIX SORT
- complessità :
  - caso migliore : ?
  - caso medio : ?
  - caso peggiore : *O(kN)*
  - complessità dello spazio : *O(kN)*

funzionamento:
Esegue gli ordinamenti per posizione della cifra ma partendo dalla cifra meno significativa.

BUCKET SORT
- complessità :
  - caso migliore : *Θ(n)*
  - caso peggiore : *Θ(n^2)*

N.B.
(a,b) = { x | $a < x < b$ } (intervallo aperto)
[a,b] = { x | $a \leq x \leq b$ } (intervallo chiuso)

-funzionamento:
assume che i valori da ordinare siano numeri reali in un intervallo semiaperto [a,b) che per semplicità di esposizione assumiamo sia intervallo [0,1).
per ordinare un array A[1....n] divide l'intervallo in n parti uguali e usa un'altro array B[1...n] di liste (bucket) mettendo in in B[k] gli A[i] che cadono nella k-esima parte dell'intervallo.

LEZIONE 4 – hashing

Le tavole hash usano il principio delle tavole ad indirizzamento diretto:
una tavola hash richiede memoria proporzionale al numero massimo di chiavi contemporaneamente presenti nel dizionario ed indipendente dalla cardinalità di U.
In una tavola hash di m celle ogni chiave *k* viene memorizzata nella cella *h(k)* usando una funzione ---> h : U → {o....m-1}
Vi possono essere collisioni quando k1 è diversa da k2 ma h(k1) = h(k2).
La soluzione alle collisioni è l'adozione di liste concatenate e relative operazioni di insert (all'inizio della lista con l'operazione chainde-hash-insert), serach e delete (che scandiscono l'intera lista alla ricerca della chiave, con le operazioni chainde-hash-search e chained-hash-delete).

Il tempo di ricerca di un chiave non presente è pari a "*a*" lunghezza della lista nella quale si sta cercando, mentre il tempo di ricerca di una chiave presente richiede in media $\Theta(1+a)$.
La distribuzione delle chiave può seguire due criteri:

-il metodo della divisione: $h(k) = k \bmod m$

basato sul resto della divisione per m (ovvero applicazione del modulo m)

vantaggi: molto veloce

svantaggio : non funziona bene per tutti i valori, scegliere un numero primo non troppo vicino a una potenza di 2

-il metodo della moltiplicazione: $h(k) = [m(kA \bmod 1)]$

A è una costante reale compresa tra 0 e , x mod 1 =x – [x]

vantaggi : la scelta di m non è critica e nella pratica funziona bene con tutti i valori di A anche se ci sono ragioni teoriche per preferire il valore $A=(\sqrt{5}-1)/2$

svantaggi : più lento del metodo della divisone

il metodo dell'indirizzamento aperto:

tutti gli elementi sono memorizzati nella tavola, la funzione hash non individua una singola cella ma una sequenza esaustiva di celle, l'inserimento di un nuovo elemento avviene nella prima cella libera che si incontra nella sequenza, la ricerca di un elemento percorre la stessa sequenza di celle

sequenze di ispezione :
è una lista ordinata degli slot esaminati, vi sono tre tecniche:

- ispezione lineare : percorre circlolarmente tutta la tabella

  vantaggi : facile da realizzare

  svantaggio : si formano lunghe file di celle occupate che aumentano il tempo medio di ricerca
- ispezione quadratica : i valori debbono essere scelti opportunamente in modo che la sequenza di ispezione percorra tutta la tavola
- hashing doppio : il valore di hash della chiave deve essere primo con m

---

LEZIONE 5 – alberi binari di ricerca

Il valore di ogni nodo è maggiore o uguale dei valori dei nodi del suo sottoalbero sinistro e minore o uguale dei nodi del suo sottoalbero destro.
Operazioni:

-inorder-tree-walk(x) : restituisce l'elenco ordinato delle chiavi presenti nell'albero

-tree-search(x,k) : restituisce il puntatore al nodo con chiave k (se esiste) o NIL se non esiste

-tree-minimum (x) : restituisce l'elemento con chiave minima (nodo + a sinistra)

-tree-maximum (x) : restituisce l'elemento con chiave massima (nodo + a destra)

-tree-successor(x) : restituisce il successore (più piccolo nodo maggiore) del nodo x. vi sono due casi:
-il successore è il minimo del sottoalbero destro
-se x non ha un figlio destro il successore è il primo avo x' tale per cui x sta nel sotto albero sinistro di x'. x' = padre di x

-tree-predecessor(x) : restituisce il predecessore (più grande nodo minore) del

nodo x. Vi sono due casi:
-il predecessore è il massimo del sottoalbero sinistro
-nel caso in cui non vi sia un sottoalbero sinistro il predecessore è il primo avo x' tale che x sta nel sottoalbero destro di x.

-tree-insert(T,z): inserisce il nuovo elemento z nell'albero binario T
-tree-delete(T,z) : cancella l'elemento z da T

LEZIONE 6 – alberi bilanciati

Un albero binario perfettamente bilanciato di n nodi ha altezza h pari a [log n] + 1,
le foglie sono circa il 50% dei nodi totali.
Il costo di una delle tre operazioni (insert delete search) è pari a 0(log n)
Il fattore di bilanciamento *B*(v) di un nodo v è la massima differenza tra i sottoalberi di v (bilanciamento in altezza) ed è pari alla differenza tra:
"altezza del sottoalbero sinistro di v – altezza sottoalbero destro di v".
Vi sono varie tipologie di alberi bilanciati:

- -alberi AVL
  - -*B*(v) <= 1 per ogni nodo
  - -bilanciamento ottenuto tramite rotazioni
- -alberi rosso neri
- -alberi 2-3
  - -*B*(v) = 0 per ogni nodo v
  - -bilanciamento ottenuto tramite merge/split, grado variabile
- -B-alberi
  - -*B*(v) = 0 per ogni nodo v
  - -usati per memorie di strutture secondarie

ALBERI AVL
alberi binari di ricerca bilanciati in altezza
B(v) = "altezza del sottoalbero sinistro di v – altezza sottoalbero destro di v".
p.s. Se l'albero è vuoto pesa -1.

la ricerca avviene con in un normale albero binario, mentre gli inserimenti e le cancellazioni possono provocare degli sbilanciamenti (nel caso in cui *B*(v) >=2 o <=2) che vanno opportunamente riparati tramite delle rotazioni.

Per quanto riguarda gli inserimenti vi sono diversi casi:

- -DD inserimento nel sottoalbero destro di un figlio destro
- -SD inserimento nel sottoalbero destro di un figlio sinistro
- -DS inserimento nel sottoalbero sinistro di un figlio destro
- -SS inserimento nel sottoalbero sinistro di un figlio sinistro

Ora le soluzioni sono due:

- -il fattore di bilanciamento di v è +1 (-1) ed il nodo è stato inserito nel sottoalbero destro (sinistro), e quindi il sottoalbero con radice v viene automaticamente bilanciato.

-oppure l'albero risulta sbilanciato e necessita di una rotazione

Mentre per quanto riguarda la cancellazione, se dopo il ricalcolo dei fattori di bilanciamento notiamo che qualcuno risulta essere +-2, allora vanno eseguite le opportune rotazioni per effettuare il ribilanciamento.

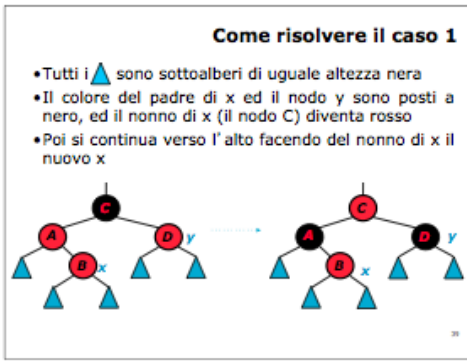
ALBERI ROSSO NERI
Alberi binari di ricerca con ulteriori proprietà, viene aggiunto un bit ad ogni nodo allo scopo di indicare il colore (rosso/nero).

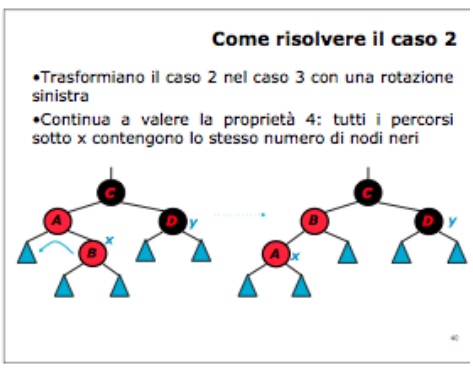
Vi sono 4 regole principali:
-1 la radice è sempre nera
-2 i nodi nil sono sempre neri
-3 se un nodo è rosso entrambi i figli sono neri
-4 ogni percorso da un nodo interno ad una foglia ha lo stesso numeri di nodi neri
altezza nera = (bh)

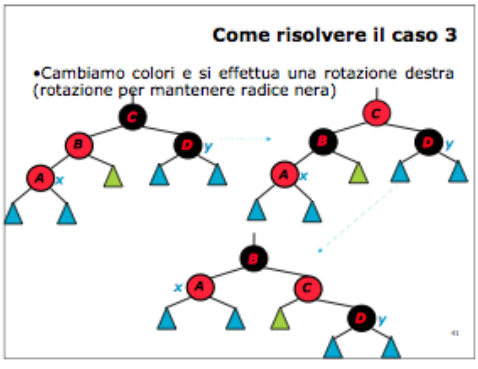
ogni nodo ha quindi le seguenti informazioni:
-parent: puntatore al genitore
-left, right : puntatore al figlio sinistro / destro
-clor : colore del nodo
-jey, data : chiave (indice) e dati.

-operazioni:
Su di un albero rosso-nero con n nodi interni le operazioni Search, Minimum, Maximum, Successor e Predecessor richiedono tempo O(log n). Le operazioni Insert e Delete su di un albero rosso- nero richiedono tempo O(log n) ma siccome esse modificano l'albero possono violare le proprietà degli alberi rosso-neri ed in tal caso occorre ripristinare tali proprietà.
Per ripristinare le proprietà degli alberi rosso neri occorre utilizzare cambi di colore e rotazioni (RB-insert-fixup negli inserimenti / RB-delete-fixup nelle cancellazioni)

Negli inserimenti:
-la proprietà 2 rimane sempre soddisfatta (sentinelle NIL sempre nere)
-la proprietà 4 rimane soddisfatta in quanto il figloi inserito sarà rosso con relativi NIL neri
-la proprietà 1 può essere violata se il nodo è inserito nella radice
-la proprietà 3 viene violata se il padre del nodo inserito è rosso







Nelle cancelazioni:

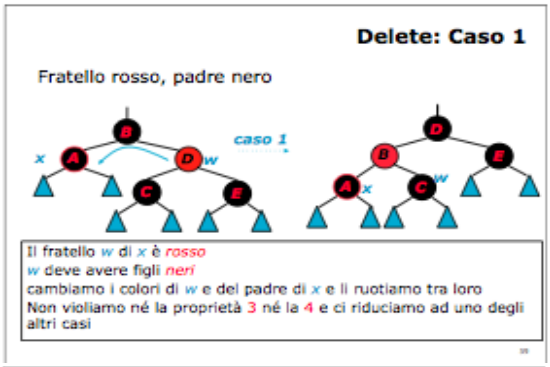
-se il nodo cancellato è rosso non sono necessari cambiamenti in quanto le proprietà non vengono violate.

-s il nodo cancellato è nero:

-violazione proprietà 1 : la radice può diventare rossa

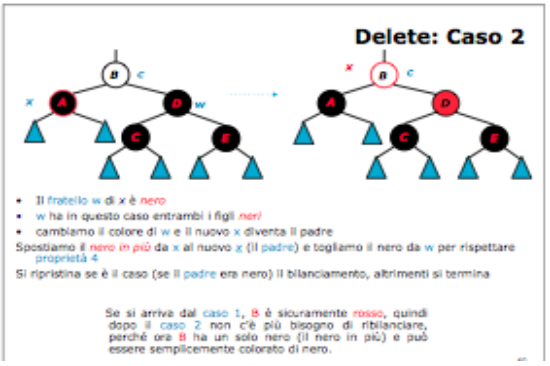
-violazione proprietà 3 : il genitore e uno dei figli del nodo cancellato erano rossi

-violazione proprietà 4 : altezza nera cambiata

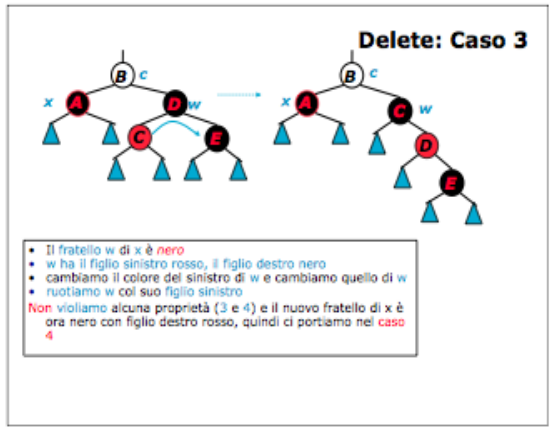


si effettua una rotazione e si passa in un altro caso: - 3 / 4 continua

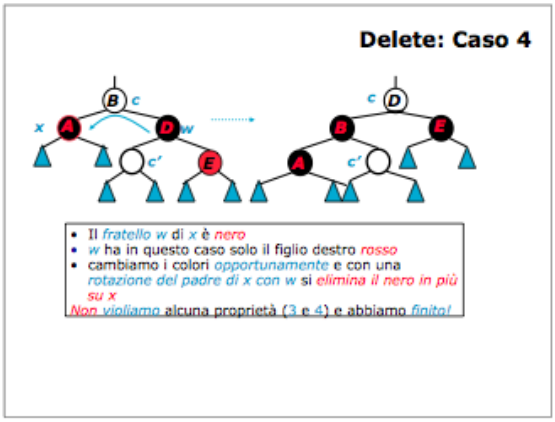
- 2 termina



Non fa rotazioni e passa in uno qualsiasi dei casi ma salendo lungo il percorso di cancellazione (verso la radice)



applica una rotazione e passa nel caso 4



è risolutivo e applica una sola rotazione

ALBERI 2-3
Praticamente è un insieme di nodi 2-3 che soddisfa la definizione di un albero e che verificano le seguenti proprietà:

- -le foglie sono tutte sullo stesso livello
- -ogni nodo ha al massimo 3 figli

-ricerca:
Un nodo 2-3 è un insieme ordinato contenente due chiavi k1 e k2 (in ordine crescente) e tre puntatori p0, p1 e p2 (sfruttati per la ricerca) tali che:

- -p0 punta al sottoalbero contenente chiavi k tali che $k < k1$
- -p1 punta al sottoalbero contenente chiavi k tali che $k1 < k < k2$
- -p2 punta al sottoalbero contenente chiavi k tali che $k > k2$

-inserimento:
l'inserimento di un nuovo elemento avviene solo nel momento in cui al termine di una ricerca non è stato trovato il valore ricercato.
Vi sono due possibilità:

- -se il vertice v contiene una sola chiave la nuova chiave viene inserita nel nodo stesso ordinatamente con quelle presente
- -se v contiene già due chiavi si considera l'insieme ordinato (a1,a2,a3) costituito dalle due chiavi del nodo e dalla nuova chiave
  - -v viene suddiviso (split) in due nodi contenenti rispettivamente a1 e a3 come figli e a2 come padre, e l'operazione procede verso l'alto nel caso in cui servissero ulteriori aggiustamenti

-cancellazione:
è l'operazione inversa dell'inserimento, opera compattando (merge) i nodi interessati.

-complessità:
O(log n) siccome l'altezza equivale a O(log n)

LEZIONE 7 – B-alberi

B-ALBERI
operazioni:

- -insert
- -delete
- -search
- -split
- -join

I nodi dei B-alberi possono contenere un numero n di chiavi ed avere n+1 figli con $n \geq 1$.
Un elevato grado di diramazione riduce in modo drastico sia l'altezza dell'albero sia il numero di letture da disco necessarie per cercare una chiave.

Proprietà:
Un B-Albero di ordine *m* è quindi un albero bilanciato che soddisfa le seguenti proprietà:
-ogni nodo contiene al più *m* - 1 elementi

-ogni nodo contiene almeno $\lceil m \rceil - 1$ elementi, la radice può contenere anche un solo elemento
-ogni nodo non foglia contenente j elementi ha $j + 1$ figli
-ogni nodo ha una struttura del tipo: $p0(k1,r1)p1(k2,r2)p2 \ldots pj-1(kj,rj)pj$ dove j è il numero degli elementi del nodo
-$k1 \ldots kj$ sono chiavi ordinate, $k1 < \ldots < kj$
-nel nodo sono presenti $j + 1$ riferimenti ai nodi figli $p0 \ldots pj$ e j riferimenti ai file dati $r1 \ldots rj$
-per ogni nodo non foglia $k(pi)$ $(i = 1,\ldots j)$ è l'insieme delle chiavi memorizzate nel sottoalbero di radice pi le quali soddisfano le seguenti relazioni:
- $\forall y \in k(p0), y < k1$ (minore del primo valore della radice)
- $\forall y \in k(pi), ki < y < ki+1, i = 1,\ldots j-1$ (compreso tra il primo e il secondo valre della radice)
- $\forall y \in k(pj), y > kj$ (maggiore dell'ultimo valore della radice)

numero di nodi n di un B-albero di altezza h:

-n minimo: $1 + 2\frac{\lceil \frac{m}{2} \rceil^{h-1} - 1}{\lceil \frac{m}{2} \rceil - 1}$

-n massimo: $\frac{m^h - 1}{m - 1}$

numero di chiavi k di un B-albero di altezza h:

-k minimo: $2\lceil \frac{m}{2} \rceil^{h-1} - 1$

-k massimo: $m^h - 1$

funzionamento:
il funzionamento è identico a quello degli alberi 2-3 tranne per il fatto che ogni nodo può avere più dei 3 figli che possono al massimo essere assegnati a un albero 2-3.

?
?
?
?

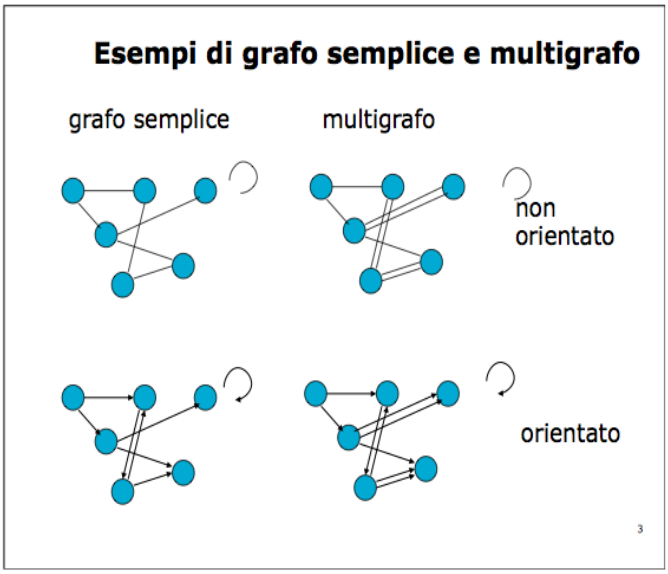
costo di cancellazione :
-caso migliore: $h+1$
-caso peggiore : $3h$ (si leggono 2h-1 nodi e se ne scrivono h+1)
un b-albero non è adatto alle elaborazioni di tipo seuqenziale, motivo per cui viene introdotto il B+ALBERO

B+ALBERI
A differenza del B-Tree, nel B+Tree tutti i dati sono salvati nelle foglie. I nodi interni contengono solamente chiavi e puntatori. Tutte le foglie sono allo stesso livello. I nodi foglia sono inoltre collegati assieme come una lista per rendere il recupero di informazioni più semplici. Tale collegamento consente di svolgere in maniera efficiente anche

interrogazioni su un intervallo di valori ammissibili. Il numero massimo di chiavi in un record è detto ordine R del B+Tree. Il numero minimo di chiavi per record è R/2. Il numero di chiavi che può essere indicizzato utilizzando un B+Tree è in funzione di R e dell'altezza dell'albero. Per un B+Tree di ordine n-esimo e di altezza h:
Il massimo numero di chiavi è : n^h
Il minimo numero di chiavi è : 2(n/2)^(h-1)
Di tutte le varianti del B-Tree, questa è la più usata, perché tutti i primi nodi interni che la memoria centrale può contenere vengono mantenuti su di essa mentre il resto dei nodi e le foglie vengono lasciate su memoria di massa. Ciò permette una maggior velocità di ricerca.

LEZIONE 8 – Algoritmi elementari per grafi

Un grafo G è costituito da un insieme (v1,v2,v3...vn) di vertici V e un insieme (e1,e2,e3....en) di archi E che collega due vertici V (estremi dell'arco).
Nei grafi (orientati) vi è un ordine tra i due estremi degli archi: il primo si chiama coda, mentre il secondo testa. (se testa e coda coincidono il cammino è un cappio).



Non orientato: semplice quando non ha cappi e non ci sono due archi con gli stessi stremi

Orientato : semplice quanto on ci sono due archi con gli stessi estremi (in caso contrario si tratta di multigrafo)

e = uv appartenenti a E ---> arco e incidente in u e v (se orientato: esace da u e entra in v)

il grado ∂(v) del vertice v è il numero di archi incidenti in v, se il grafo è orientato è bene identificare gli archi uscenti con $\partial^+(v)$ e gli archi entranti con $\partial^-(v)$

caratteristiche:

- -raggiungibilità di v : quando esiste almeno un cammino da un vertice u al vertice v
- -connesso (grafo non orientato) : se esiste almeno un cammino tra ogni coppia di vertici
- -componenti connesse : classi di equivalenza dei suoi vertici rispetto alla relazione di raggiungibilità
- -fortemente connesso (grafo orientato): se esiste almeno un cammino da ogni vertice u ad ogni altro vertice v (sono le classi di equivalenza dei suoi vertici rispettto alla relazionei di mututa accessibilità)

Un sottografo è un grafo appartenente a un grafo più grande.

Rappresentazione:

-liste di adiacenza : viene costruita una lista per ogni vertice v contenente i vertici adiacenti al vertice v

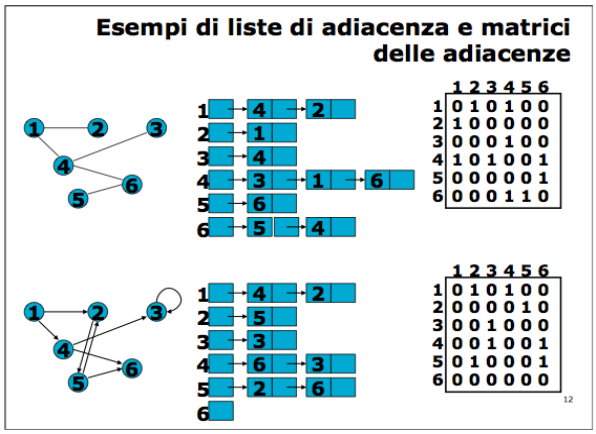
necessità:

-non necessari |V| puntatori alle cime delle liste Adj[u]

-|E| elementi delle liste se il grafo è orientato

-2|E| elementi delle liste se il grafo non è orientato

-matrice delle adiacenze: i vertici vengono numerati 1,2,....|V| in modo arbitrario, la rappresentazione è quindi costruita da un matrice booleana A = $(a_{i,j})$ tale che il valore è 1 se vi è un arco (e quindi i vertici sono adiacenti) e 0 viceversa.



-confronto:

lista di adiacenza:

-la quantità di memoria richiesta è O(|V| + |E|)

-non c'è nessun modo per verificare se un arco è presente nel grafo.

Matrice di adiacenza:

-la quantità di memoria richiesta è $O(|V|^2)$

-per determinare se un arco è presente si richiede tempo costante

modalità di visita di un grafo:

-visita in ampiezza (BFS)

-funzionamento :si effettua la ricerca del valore desiderato tra tutti i nodi aventi distanza dalla radice pari a 1, se non lo si trova si passa tra i nodi aventi distanza 2, 3 e via dicendo.

N.B. (Assume che il grafo sia rappresentato con liste di adiacenzaQuesta tipologia di ricerca espande uniformemente la frontiera tra i vertici scoperti e quelli non ancora scoperti).

-principio dei colori: i vertici possono esser di colore:

-bianco : non ancora visitati

-grigio : vertici raggiunti che stanno alla frontiera

-nero: vertici raggiunti che non stanno alla frontiera

-regole:

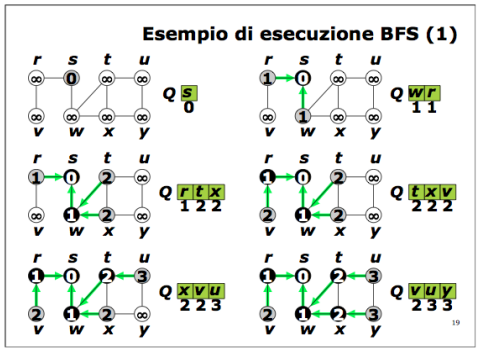
-i vertici adiacenti a quelli neri possono essere neri o grigi (non bianchi)

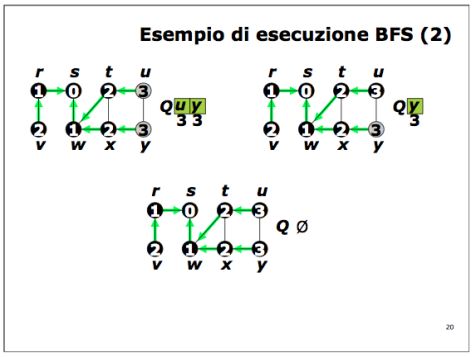
-i vertici adiacenti ad un vertice grigio possono esser solo bianchi

-l'albero iniziale contiene solo la radice s

-quando viene scoperto un vertice bianco v a causa si un arco uv che lo connette ad un vertice u scoperto precedentemente, il vertice v e l'arco uv vengono aggiunti all'albero.

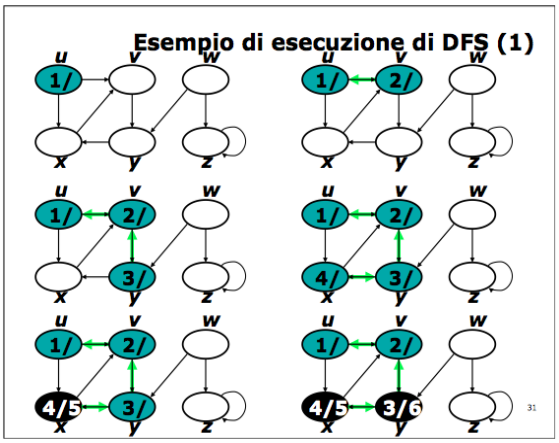
-il vertice u viene detto padre di v

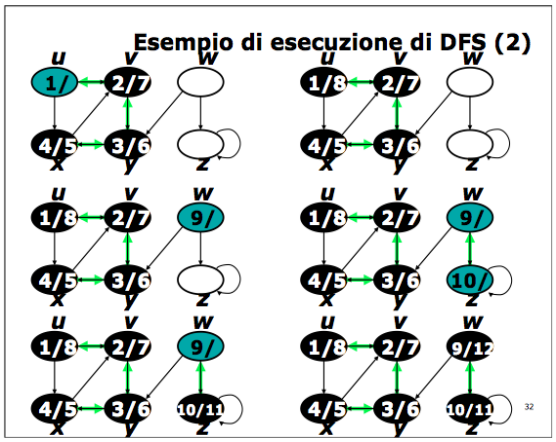




-complessità: O(n+m) ---> m = lunghezza delle liste
-proprietà:
-proprietà delle distanze: d(u,v) = cammino minimo tra u e v, se u e v non sono collegati d(u,v)=∞
-proprietà del limite superiore e della coda: $d[v_i] <= d[v_{i+1}]$

-visita in profondità (DFS)
-funzionamento: avanzare nella ricerca finche è possibile, vanno quindi esplorati i vertici uscenti dall'ultimo vertice esplorati, se tutti i vertici uscenti dall'ultimo vertice esplorato portano ad un altro vertice già esplorato si torna indietro lungo il cammino percorso.
Il procedimento continua fino a quando non vengono scoperti tutti i vertici, nel caso in cui non venissero scoperti tutti, in tal caso si ripete il procedimento partendo da uno dei vertici non ancora raggiunti.
A differenza del BFS non viene costituito un albero ma una foresta, ovvero un insieme di alberi.
La ricerca pone due marcatempo su ogni vertice u:
-d[u]: registra quando il vertice viene scoperto e colorato di grigio.
-f[u]:registra quando il vertice è stato completato e viene colorato di nero
-principio dei colori: i vertici possono esser di colore:
-bianco : non ancora raggiunti
-grigio : vertici scoperti
-nero: vertici raggiunti la cui lista delle adiacenze è stata completamente visitata





-complessità: O(|V| + |E|)
-proprietà delle parentesi: se si rappresenta la scoperta di ogni vertice u con una parentesi aperta ($_u$ ed il completamento con una parentesi chiusa $_u$) si ottiene una sequenza bilanciata di parentesi
-proprietà delle discendenti e del camino bianco: Il vertice ***v*** è discendente del vertice ***u*** in un albero della foresta di visita in profondità se e solo se quando ***u*** viene scoperto esiste un cammino da ***u*** a ***v*** i cui vertici sono tutti bianchi (cammino bianco).

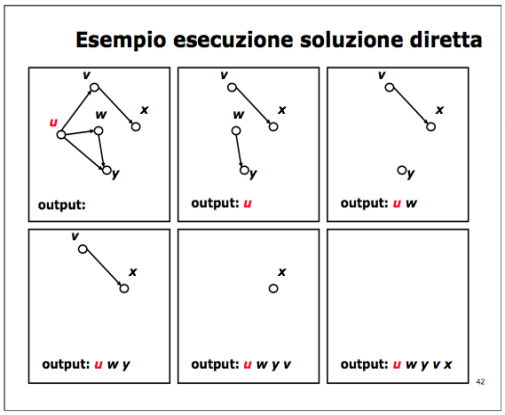
CLASSIFICAZIONE DEGLI ALBERI
-archi d'albero: archi ***uv*** con ***v*** scoperto visitando le adiacenze di ***u***
-archi all'indietro: archi ***uv*** con ***u = v*** oppure ***v*** ascendente di ***u*** in un albero della foresta di ricerca in profondità

-archi in avanti: archi ***uv*** con ***v*** discendente di ***u*** in un albero della foresta
-archi trasversali: archi ***uv*** in cui ***v*** ed ***u*** appartengono a rami o alberi distinti della foresta

N.B. Se un arco soddisfa le condizioni per appartenere a più di una categoria, viene classificato in quella che compare per prima nell'ordine in cui le abbiamo elencate
Esempio: ogni arco d'albero soddisfa anche le condizioni per essere un arco in avanti; se il grafo non è orientato ogni arco all'indietro è anche un arco in avanti

Visitando un grafo tramite DFS bisogna porsi due problemi:

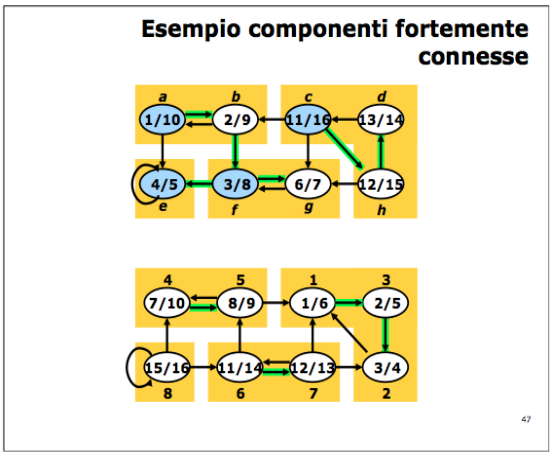
-calcolare l'ordinamento topologico indotto da un grafo aciclico (se e solo se nella visita in profondità non si trova nessun arco all'indietro) per ogni arco uv appartenente ad E, u precede v, l'ordinamento topologico si usa per determinare un ordine in cui eseguire un insieme di attività in presenza di vincoli di precedenze



-soluzione:
-trovare ogni vertice che non ha alcun arco incidente in ingresso
-stampare tale vertice e rimuoverlo insieme ai suoi archi
-Ripetere la procedura finché tutti i vertici risultano rimossi

-complessità: O(|V| + |E|)

- individuare le componenti connesse / fortemente connesse di un grafo non orientato: una componente fortemente connessa di un grafo orientato è un insieme massimale di vertici tale per cui ogni vertice ha un cammino verso ogni altro vertice



-calcolo delle componenti connesse:
- si usa la visita in profondità in G per ordinare i vertici in ordine di tempo di completamento f decrescente (come per l'ordinamento topologico)
- si calcola il grafo trasposto G^T del grafo G
- si esegue una visita in profondità in GT usando l'ordine dei vertici calcolato nella prima fase nel ciclo principale

-proprietà grafi (non orientati):
ciclo di Eulero : ciclo in un grafo G che visita ogni arco di G una volta (possibile solo se ogni vertice di G è pari, vedi sotto)
ciclo di Hamilton: Ciclo in un grafo G che visita ogni nodo di G una volta
vertice v dispari : grado di v dispari
vertice v pari : grado di v pari

LEZIONE 9 – Alberi di connessione minima

Determinare come interconnettere diversi elementi fra loro minimizzando certi vincoli è un problema di notevole importanza nella quale rientrano:

INPUT

-G=(V,E)

-w = costo di connessione

-se u,v sono collegati w indica il peso della loro connessione, viceversa è infinito

-w(u,v) = w (v,u) se G non è orientato

OUTPUT

-albero di copertura (spanning tree) è un sottografo $T=(V,E_T)$ tale per cui:

-T è un albero

-E_T è compreso in E

-t contiene tutti i vertici di G

informazioni:

-Se si considera il peso degli archi, allora gli alberi di connessione possono avere costi diversi.

-L'albero di copertura minimo è l'albero con il più basso costo di interconnessione (potrebbe non essere unico).

-Un arco (u,v) è detto sicuro per un sottoinsieme si qualche albero di connessione minima A se A U {(u,v)} è ancora un sottoinsieme di qualche albero di connessione minimo.

Un taglio.

Definizioni:

-un taglio (S,V/S) di un grafo non orientato G=(V,E) è una partizione di V in due sottoinsiemi disgiunti.

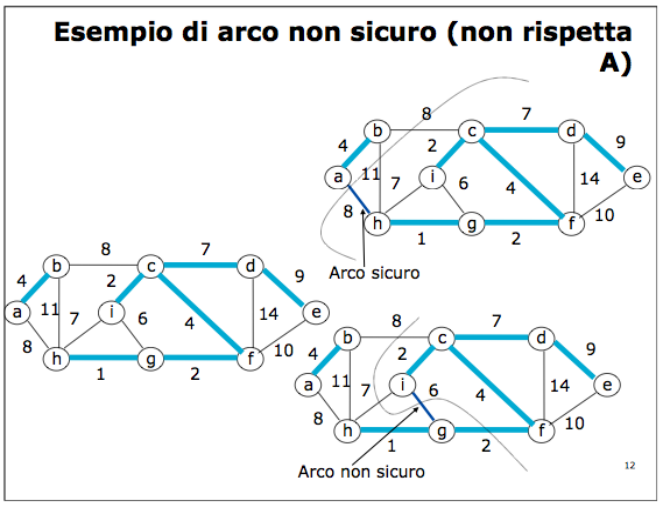
-Un arco (u,v) attraversa il taglio se u appartiene a S e se v appartiene a V/S

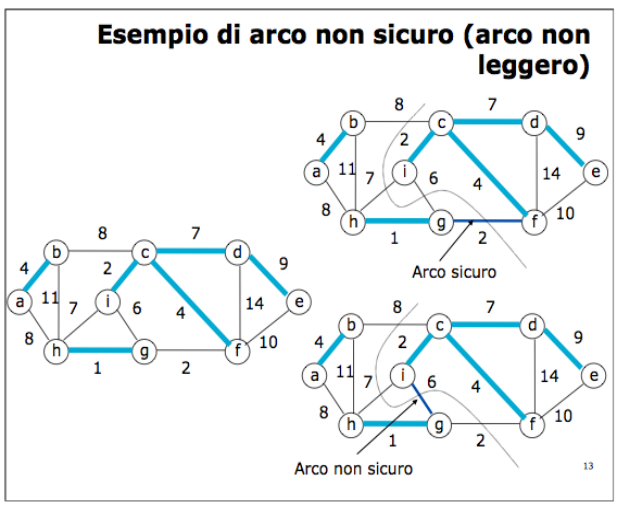
-Un taglio rispetta un insieme di archi si A se nessun arco attraversa il taglio

un arco che attraversa un taglio è leggero nel taglio se il suo peso è minimo fra i pesi degli archi che attraversano un taglio.

Caratteristiche di un arco sicuro:

-sia A un insieme di archi contenuto in qualche albero di connessione minimo, sia (S,V/S) un taglio che rispetta A e sia a=uv un arco leggero per il taglio (S,V/S)



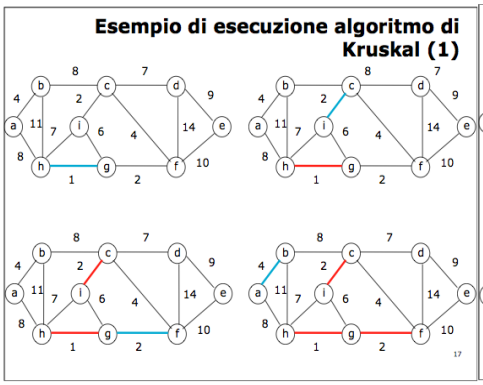


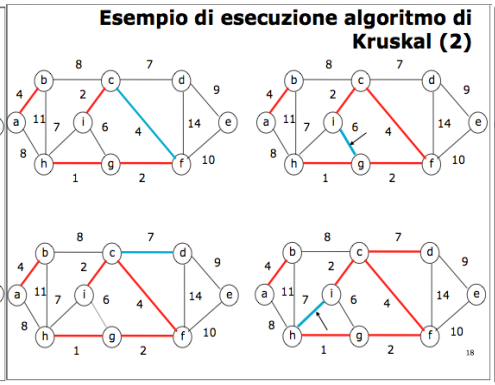
Algoritmo di Kruskal

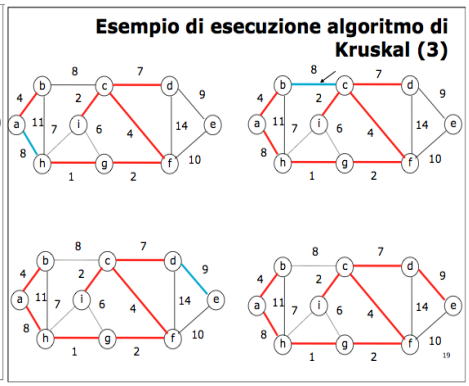
-funzionamento: Ingrandire sottoinsiemi disgiunti di un albero di copertura minimo connettendoli fra di loro fino ad avere l'albero complessivo.

Si individua un arco sicuro scegliendo un arco (u,v) di peso minimo tra tutti gli archi che connettono due alberi distinti (componenti connesse) della foresta

Nell'algoritmo ad ogni passo si aggiunge alla foresta un arco con il peso minore.







-Complessità: O(|E|log|E|)

Algoritmo di Prim

-funzionamento: L'algoritmo di Prim procede mantenendo in *A* un singolo albero

L'albero parte da un vertice arbitrario *r* (la *radice*) e cresce fino a quando non ricopre tutti i vertici.Ad ogni passo viene aggiunto un arco leggero che collega un vertice in V*A* con un vertice in *V\VA (*insieme di nodi raggiunti da archi in *A* Correttezza).

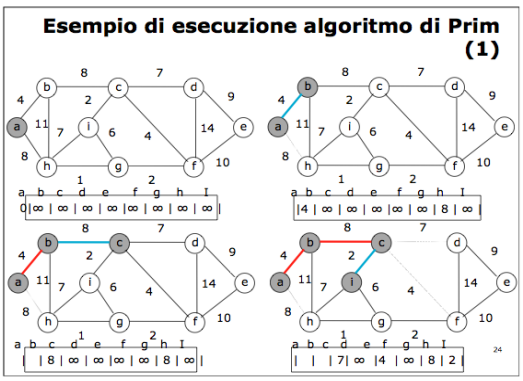
(VA,V\VA) è un taglio che rispetta *A* in quanto gli archi leggeri che attraversano il taglio sono sicuri.

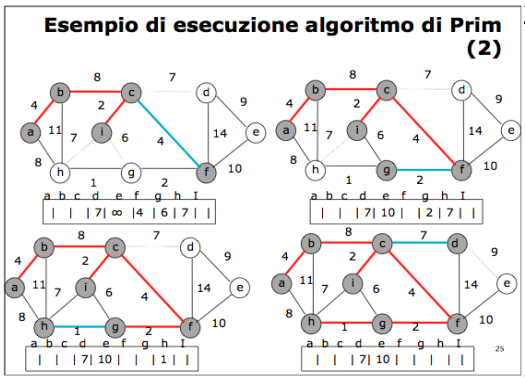
I vertici vengono aumentati con:

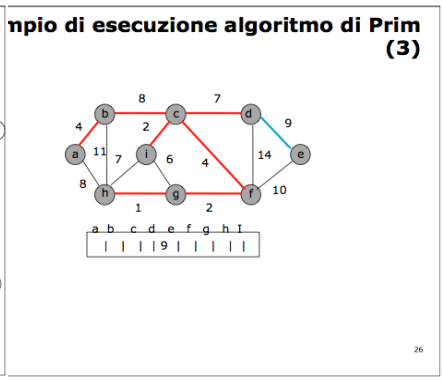
-Un puntatore p al padre nell'albero in costruzione

-Un campo colore che è bianco inizialmente e diventa nero quando il vertice viene aggiunto all'albero in costruzione

-Un campo key che contiene il costo minimo di un arco che connette il vertice ad uno dei vertici già raggiunti dall'albero in costruzione







-Complessità : O(|V|lg|V| + |E|lg|V|)

LEZIONE 10 – {1} Cammini minimi da sorgente unica

un grafo orientato, i cui archi (u,v) hanno un relativo peso w(u,v), ha un cammino minimo dove la somma dei pesi degli archi w(u,v) da un vertice u a un vertice v è più bassa.
Vi sono quattro possibili casi:

- -{1} cammini minimi da sorgente unica
- -{2} cammini minimi da ogni vertice ad un'unica destinazione
  - -si risolve simmetricamente
- -{3} cammini minimi da un'unica sorgente ad un'unica destinazione
  - -si risolve come unica sorgente (istanza 1)
- -{4} cammini minimi da ogni vertice ad ogni altro vertice
  - -si risolve con istanza 1 per ogni vertice

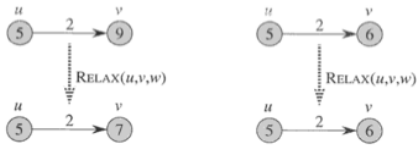
N.B. Un cammino può contenere cicli a patto che questi non siano negativi, difatti se vi sono archi con peso negativo all'interno di un ciclo (che può diventare negativo), per convenzione tale ciclo pesa -∞. Ma attenzione, un cammino minimo non può contenere nemmeno cicli positivi in quanto il peso non sarebbe minimo (w ciclo infinito = +∞)

Nella maggiora parte dei casi è utile tenere traccia anche dei vertici all'interno di un cammino, per convenzione quindi si tiene traccia del nodo predecessore, sia che sia un vertice che NIL, tale sottografo si chiama sottografo dei predecessori.

Se un cammino è minimo lo sono anche tutti i suoi sottocammini.

N.B. ∂ = somma del peso di più archi
Introduciamo il concetto di tecnica di rilassamento, ovvero per ogni vertice si mantiene un ulteriore campo d[v] che rappresenta la stima del cammino minimo (durante l'esecuzione dell'algoritmo è un limite superiore per ∂(s,v) e alla fine equivale a ∂(s,v))
funzionamento: verificare se passando per u è possibile migliorare la stime del cammino minimo per v e in caso affermativo nell'aggiornare i valori sia di d[v] si di π[v].

Esempio di rilassamento

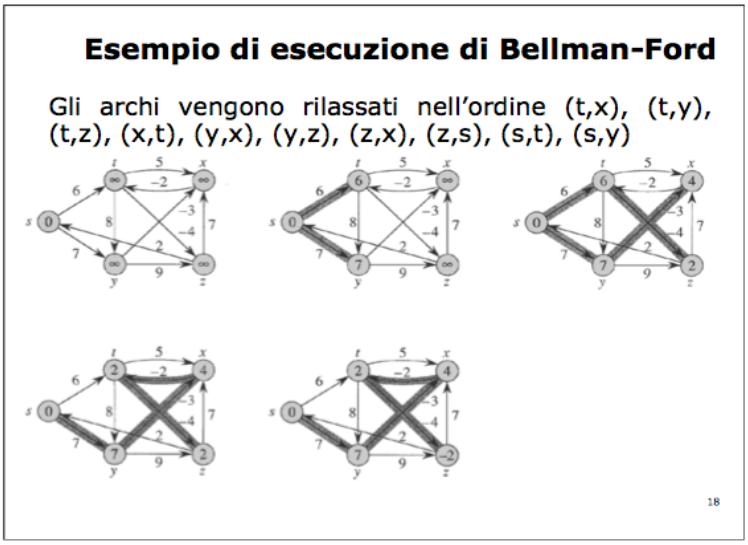


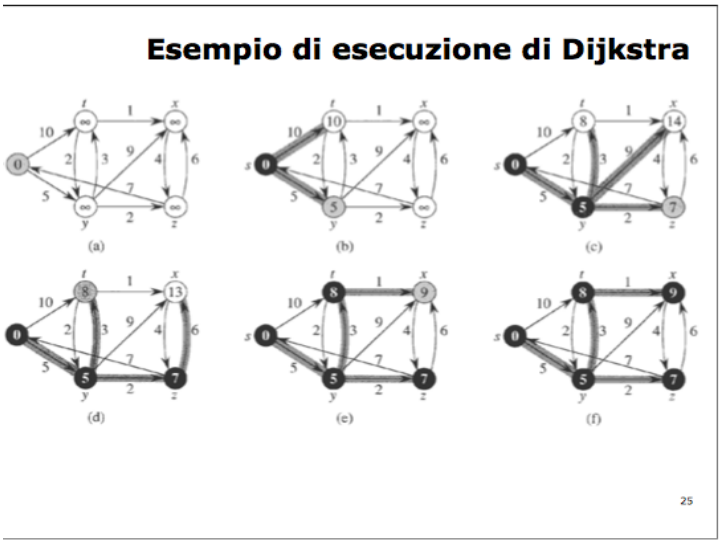
-Algoritmo di Bellman-Ford
1. $d[v_1] \leftarrow d[s] + w(s,v_1)$
2. $d[v_2] \leftarrow d[v_1] + w(v_1,v_2)$
3. $d[v_3] \leftarrow d[v_2] + w(v_2,v_3)$
4. …
5. $d[vk] \leftarrow d[vk-1] + w(vk-1,vk)$

-Dopo k passi d[vk]=δ(s,vk)
-Il problema è che non si conoscono gli archi del cammino minimo né l'ordine
-Se rilassiamo tutti gli archi, sicuramente si effettuerà anche il rilassamento al passo 1
-Se si ripete il processo, sicuramente si effettuerà anche il rilassamento al passo 2 e così via ⇒sono necessari |V|-1 passate nella quale rilassa tutti gli archi (esegue molti rilassamenti inutili.
Complessità: O(|V||E|)

-Algoritmo di dijkstra
risolve il problema sei cammini minimi a sorgente unica in un grafo senza archi con peso

negativo mantenendo un insieme di nodi S i cui pesi dei cammini minimi da s sono già stati determinati selezionando ripetutamente il nodo in u appartenente a V-S con la minima stima del cammino minimo e rilassa tutti gli archi che escono da u.





-complessità: O(n2 + n) = O(n2)

LEZIONE 11 – {2} Cammini minimi fra tutte le coppie

Dato un grafo vogliamo trovare i cammini minimi tra tutte le coppie di vertici in G, tale problema può essere risolto tramite l'algoritmo per il problema a sorgente unica applicato ad ogni vertice:

- -Dijkstra
- -Bellman Ford

gli algoritmi per determinare i cammini minimi tra tutte le coppie dono basati su una rappresentazione del grafo tramite matrice di adiacenza.
L'output tabellare è una matrice D0(d_{ij}) di dimensione *nxn* dove d_{ij} è il peso di un cammino minimo dal vertice i al vertice j, inoltre va calcolata anche la matrice dei predecessori.

Algoritmo di Floyd-warshall
-funzionamento:
p=cammino minimo all'interno di un sottogruppo di vertici {1,2,3...k}
L'algoritmo di floyd-warshall sfrutta una relazione tra il cammino p ed i cammini minimi da i a j i cui vertici intermedi sono in {1,2,3...k-1}
Vi sono due diversi casi:

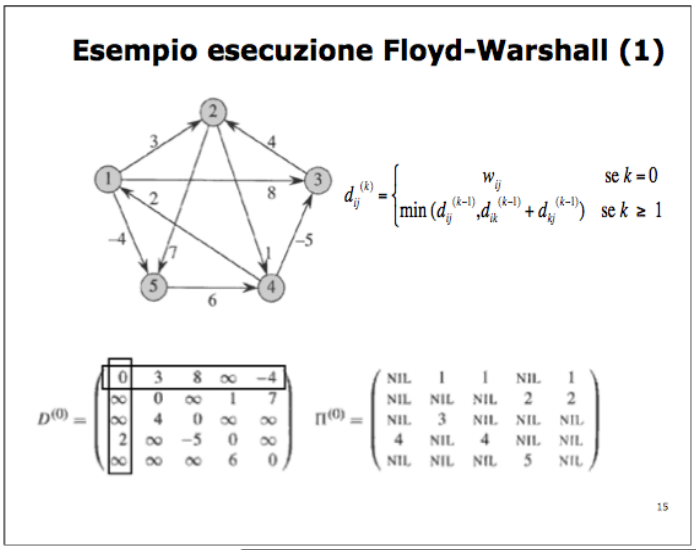
- -k non è un vertice intermedio:
  - i vertici intermedi sono tutti in {1,2,3....k-1}, e un cammino minimo da i a j con vertici intermedi in {1,2,3....k-1} è anche un cammino minimo con vertici intermedi {1,k-1}
- -k è un vertice intermedio.

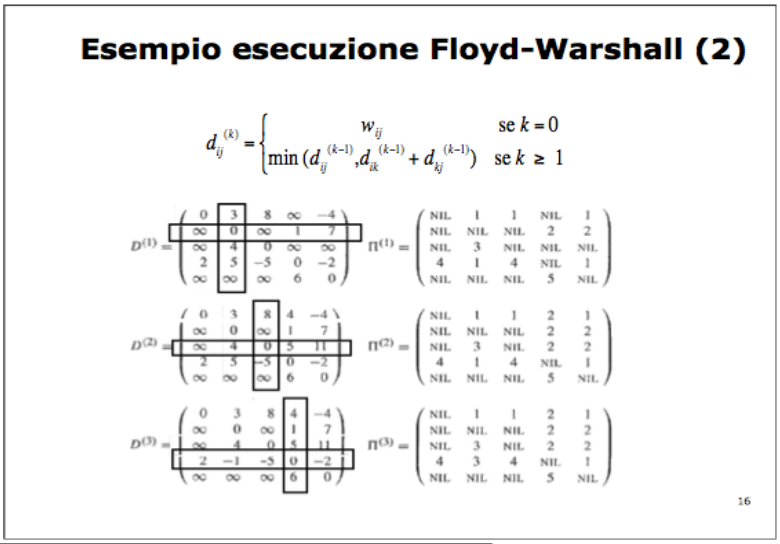
La definizione ricorsiva prevede che quindi il peso totale sia la somma del minimo cammino tra ogni vertice.
-complessità: Θ(n3)

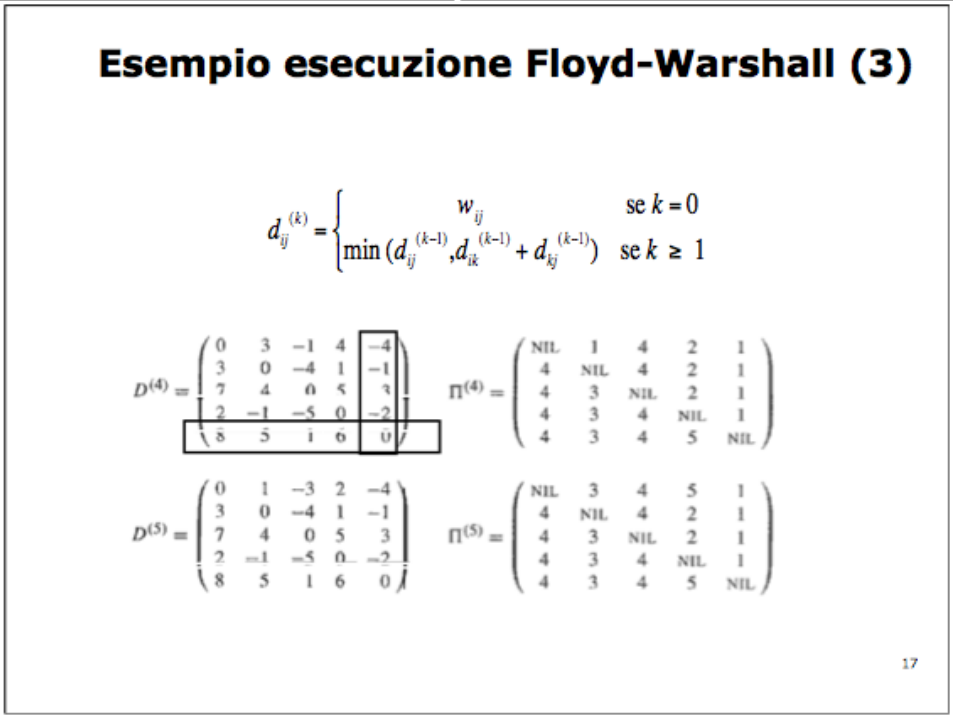
Costruzione del cammino minimo
Vi sono vari modi:

-si costruisce la matrice π della matrice D

-si costruisce la matrice π online nello stesso modo con cui si costruisce la matrice D (sequenza di matrici)





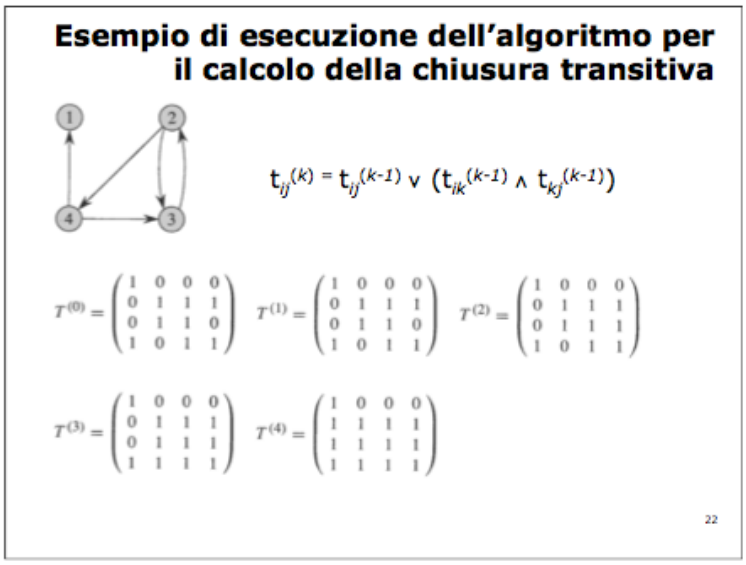


-chiusura transitiva di un grafo orientato: è il grafo G*=(V,E*) dove E*={(i,j) : esiste un cammino da i a j in G}

Vi sono due soluzioni applicabili:

-assegno un peso pari a 1 ad ogni arco in E ed applico Floyd Warshall e se esiste un cammino da i a j allora $D_{ij} < n$; altrimenti $d_{ij}=\infty$ ---> complessità $\Theta(n^3)$

-sostituisco le operazioni di min e + dell'algoritmo di Floyd Warshall con le operazioni logiche v (or) e (and)



complessità: Θ(n3)

LEZIONE 12 – Problema del flusso

Una rete di flusso è un grafo orientato G=(V,E) ai cui vertici è associata una capacità c(u,v)>=0 (se non appartiene all'insieme degli archi E allora c(u,v) = 0)
N.B. È bene distinguere il vertice s (sorgente) e il vertice t (pozzo), nel seguente modo:

- -sorgente: ogni vertice è raggiungibile dalla sorgente
- -pozzo: il pozzo è raggiungibile da ogni vertice

Il flusso viene definito come il flusso totale che esce dalla sorgente s diretta al vertice v.
Il problema principale è quello di trovare in una rete di flusso il flusso con valore massimo.

proprietà dei flussi:
-limite di capacità : il flusso non può superare la capacità
-antisimmetria : dice che il flusso f(u,v) è l'esatto opposto del flusso f(v,u) → f(u,v) = -f(v,u)
N.B. f(u,u) = -f(u,u) = 0
-conservazione: è nulla la somma dei flussi in ogni nodo diverso da s e da t

vi sono varie tipologie di flusso:
-Il flusso entrante totale positivo in un nodo u è : sommatoria di f(v,u)
-il flusso totale uscente positivo in un nodo u è: sommatoria di f(u,v)
-il flusso totale netto è il flusso totale uscente positivo meno il flusso totale entrante positivo.

Algoritmo di Ford-Fulkerson
principi:

- -cammini aumentati
- -reti residue
- -tagli

funzionamento:

- -f(u,v)=0 per ogni u e v e quindi si aumenta iterativamente il valore del flusso cercando un cammino dalla sorgente s al pozzo t lungo il quale sia impossibile

inviare ulteriore flusso. Il processo termina quando non ci sono più cammini aumentanti.

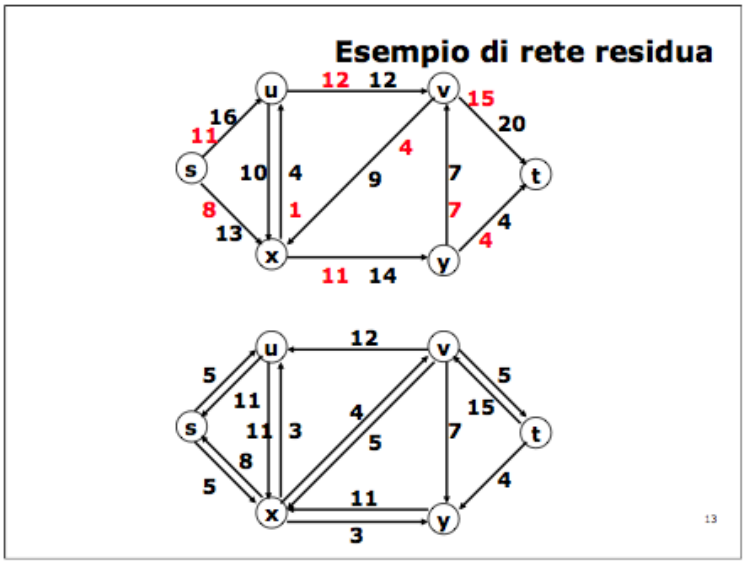
Rete residua
dato un flusso f in una rete di flusso G = (V,E) la capacità residua di un arco uv è la quantità $c_f(u,v) = c(u,v) – f(u,v)$ va da se che ila rete residua p la rete con archi :

-E_f={(u.v) appartenenti a VxV tali che $c_f(u,v)>0$}

se un arco (u,v) in G ha un flusso tale che $0<f(u,v)<c(u,v)$ allora l'arco (u,v) si sdoppia in G_f:

-arco in avanti con c_f (u,v) = c(u,v) – f(u,v)

-arco all'indietro con $c_f(v,u) = c(v,u) – f(v,u) = c(v,u) + f(u,v)$



Proprietà additiva dei flussi:

f = flusso in una rete di flusso

g = flusso in una rete residua di flusso

|f+g| = |f| + |g|

un cammino aumentante è semplicemente un cammino p da s a t nella rete residua. La relativa capacità è la minima capacità degli archi di p nella rete residua.

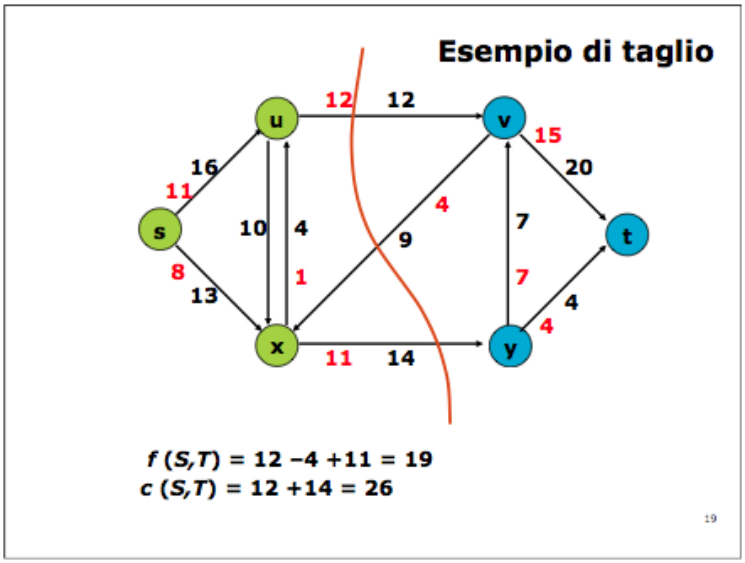
Se p è un cammino aumentante allora $f_p(u,v)$ è un flusso di G_f di valore $|f_p|$ 0 $c_f(p)>0$

Metodo del cammino aumentante:
aumenta ripetutamento il flusso lungo dei cammini aumentanti finchè non si arrivi ad un flusso di valore massimo (massimo se e solo se non ci sono cammini aumentanti da s a t nella rete residua --> abbiamo bisogno di definire il concetto di taglio in una rete di flusso)

se f è un flusso in G = (V,E) allora:
-il flusso che attraversa il taglio (S,T) è: la somma dei flussi tra x e y
-la capacità del taglio è la somma delle capacità di x e y

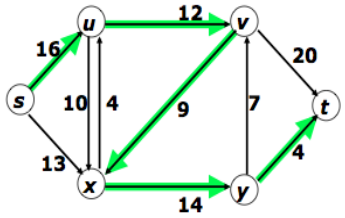
il flusso che attraversa il taglio S,T è pari a |*f*| = c(S,T)

Sia f un flusso in una rete di flusso G=(V,E). Allora le seguenti affermazioni sono equivalenti:
-f è un flusso massimo di G
-la rete residua Gf non ha cammini aumentanti
-|f| = c(S,T) per qualche taglio (S,T)

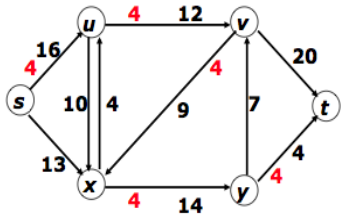


Esempio (1)

Rete



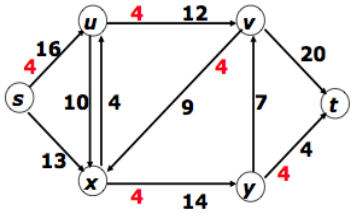
Cammino aumentante



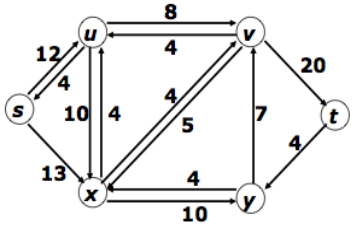
Nuovo flusso

Esempio (2)

Nuovo flusso

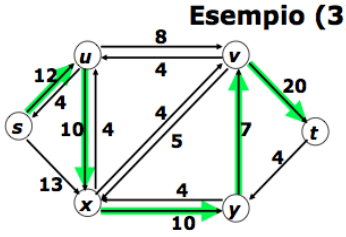


Rete residua

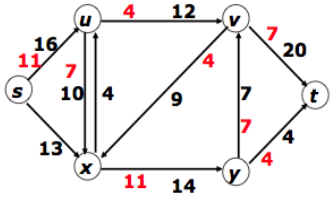


Esempio (3)

Rete residua



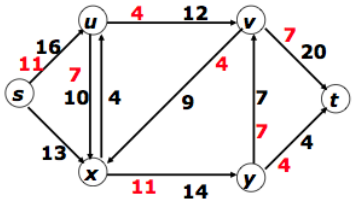
Cammino aumentante



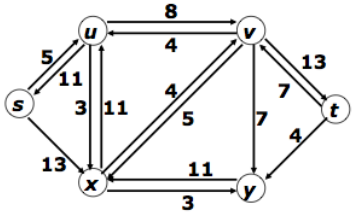
Nuovo flusso

Esempio (4)

Nuovo flusso

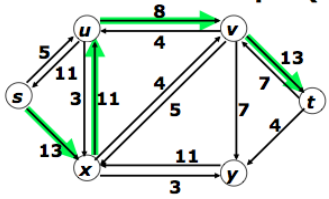


Rete residua

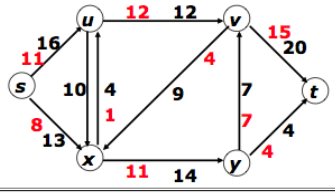


Esempio (5)

Rete residua



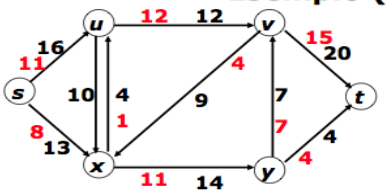
Cammino aumentante



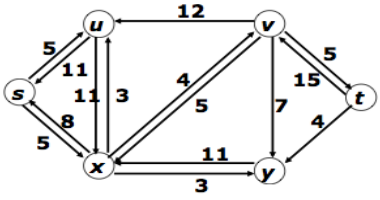
Nuovo flusso

Esempio (6)

Nuovo flusso

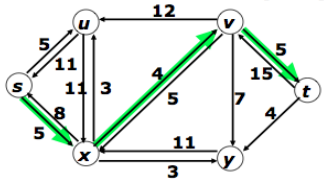


Rete residua

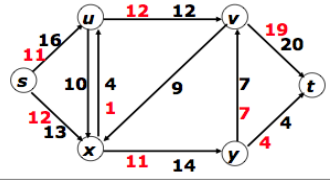


Esempio (7)

Rete residua



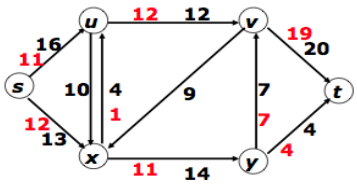
Cammino aumentante



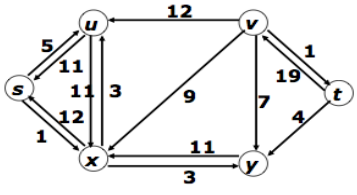
Nuovo flusso

Esempio (8)

Nuovo flusso



Rete residua



-complessità : $O(|E|^2 |V|)$

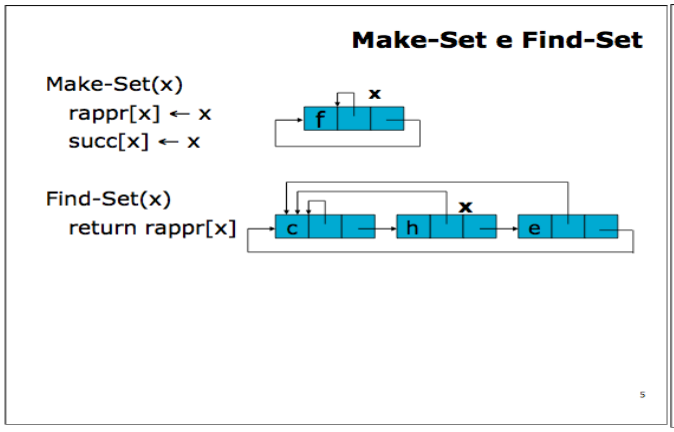
Algoritmo di Edmonds-Karp
utilizza la ricerca in ampiezza per trovare il cammino aumentante (cammino minimo da s a t nella rete residua dove ogni arco ha distanza di valore unitario)

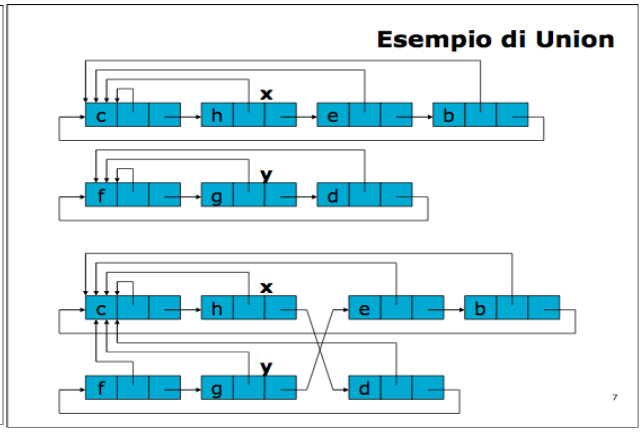
grafi bipartiti
un grafo non orientato si dice bipartito ei i suoi vertici si possono ripartire in due sottosistemi S e D tali che ogni arco abbia estremi appartenenti a sottoinsiemi distinti (E compreso in SxD). Se xy è un arco diciamo che x e y si possono accoppiare.
Il problema principale chiede di trovare un insieme massimo di coppie distinte.

LEZIONE 13 – Strutture dati per insiemi disgiunti

Queste strutture servono a mantenere una collezione $S = \{S_1, S_2, ...S_k\}$ di insiemi disgiunti. Ogni insieme della collezione è individuato da un rappresentante che è un particolare elemento dell'insieme.

-operazioni sugli insiemi disgiunti:
- -Make-set(x): agigunge alla struttura dati un nuovo insieme contenente solo l'elemento x (si richiede che non compaia in nessun altro insieme della struttura)
- -Find-set(x) . ritorna il rappresentante dell'insieme contenente x
- -Union(x,y) : riunisce i due insiemi contenti x e y in un unico insieme





-rappresentazione:
- il modo più sempiceper rappresentare una famiglia di insiemi disgiunti è usare una lista circolare per ogni insieme dove ogni nodo ha un puntatore al nodo successivo ed un puntatore al nodo della lista che viene assunto come rappresentante.

-campi:
- -info: l'informazione contenuta nel nodo
- -rappr: il puntatore al rappresentante
- -succ: il puntatore al nodo successivo

-complessità Union(x,y) : $O(n_2)$ con n_2 = lunghezza della seconda lista

Euristica dell'unione pesata
consiste nello scegliere sempre la lista più corta per aggiornare i puntatorei al rappresentante, occorre memorizzare la lunghezza di ciascuna lista in un opportuno campo “lung” del rappresentante, si aggiunge ad ogni nodo un campo booleano rp per distinguere il rappresentante dagli altri nodi (posiamo mettere il campo lung al posto del campo rappr che risulta inutile)

Una rappresentazione efficiente la si ottiene usando foreste di insiemi disgiunti, dove ogni insieme è rappresentato con un albero i cui nodi, oltre al campo info, hanno soltanto un campo parent che punta al padre.

-complessità find-set(x):
    -proporzionale alla lunghezza del cammino dal nodo alla radice dell'albero
-complessità Union :
    è quella delle due chiamate find-set(x) e find-set(y)

Euristica dell'unione per rango
Per ogni nodo x manteniamo un campo rank che è un limite superiore all'altezza del sottoalbero di radice x (numero di archi nel cmamino più lungo fra x e foglia discendente),.
Union mette la radice con rango minore come figlia di quella di rango maggiore.

Euristica della compressione dei cammini
Quando effettuiamo una find-set(x) dobbiamo attraversare il cammino dalla x alla radice.
Possiamo approfittarne per far puntare alla radice dell'albero i puntatori al padre di tutti i nodi incontrati lungo il cammino. Le eventuali operazioni findset successive sui nodi di tale cammino risulteranno molto meno onerose.

Considerazioni del paragone:
entrambe le euristiche migliorano le prestazioni quando sono usate separatamente.
-euristica dell'unione per rango : una sequenza di m operazioni delle quali n sono make-set richiede tempo O(m log n)
-euristica della comprensione dei cammini: una sequenza di m operazioni delle quali n sono make-set ed f sono findset richiede tempo:

$$\Theta(f \log_{(1+f/n)} n) \qquad \text{se } f \geq n$$
$$\Theta(n + f \log n) \qquad \text{se } f < n$$

-la combinazione delle due euristiche ha tempi ancora migliori

---

LEZIONE 14 – Ricorsione

E' ricorsivo quello che può essere definito in termini di se stesso (es. fattoriale, fibonacci), ovvero definire un insieme di oggetti infinito mediante una descrizione finita..

tipologie:

-diretta: quando una procedura è espressa in termini di se stessa

-indiretta: quando una procedura “p” è espressa in termini di una procedura a sua volta definita direttamente o indirettamente in termini di “p”

Una routine ricorsiva è costituita da:

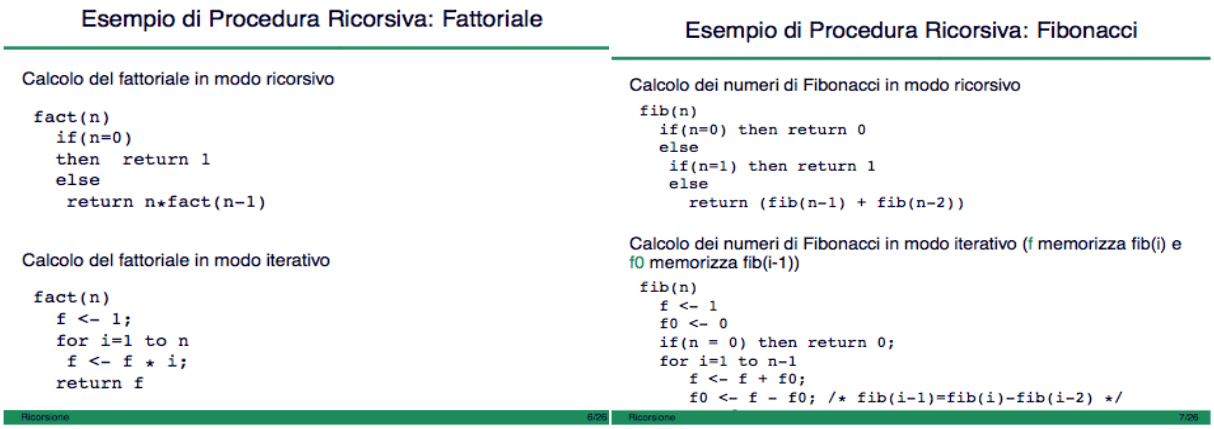
-base della ricorsione

-passo ricorsivo

funziona solo se le sue variabili interne sono stanziate nuovamente per ogni attivazione.
L'introduzione della ricorsione introduce il problema potenziale di ricorsioni infinite, ovvero mentre un loop infinito effettivamente non ha mai termine, la ricorsione “infinita” prima o poi deterina l'esaurimento della memoria disponibile per lo stack del processo che la esegue e la distruzione di quest'ultimo, in pratica è necessario che la ricorsione termini e che il numero di passi della ricorsione (uso dello stack) sia modesto.
Viene utilizzata per semplificare la struttura del programma ma:

- in alcuni linguaggi le chiamate di procedura sono più costose delle operazioni di assegnamento
- la ricorsione implica molto utilizzo di memoria

Ogni procedura ricorsiva può essere tradotta in una procedura equivalente iterativa, se è necessario utilizzare esplicitamente una pila per realizzare in modo iterativo un algoritmo ricorsivo allora si dice che l'algoritmo è intrinsecamente ricorsivo.



Vi sono casi in cui è conveniente sostituire la ricorsione con l'iterazione (sempre possibile), nei casi in cui l'algoritmo è intrinsecamente ricorsivo la maggior efficienza viene ottenuta al prezzo della manipolazione esplicita di un pila.
Differenza tra algoritmo intrinsecamente ricorsivo e uno no:

-un algoritmo ricorsivo deve sempre tenere traccia dei valori delle sue variabili locali

-uno non intrinsecamente ricorsivo non ha bisogno di tener traccia

Per eliminare la ricorsione possiamo avvalerci della tecnica di “tail recursion” (quando una procedura p):

-ha un parametro x passato per valore

-la sua ultima istruzione è chiamata di p ricorsiva con valore y dal parametro di chiamata

funzionamento eliminazione:

si scrive la procedura/funzione in modo che il ramo then sia quello che contirnr la

chiamata ricorsiva di cosa ed il ramo else quello che contiene la base della ricorsione.
La chiamata ricorsiva si può trasformare in un ciclo come segue:

-l'istruzione if\then\else diventa un ciclo di while
-il test di if\then\else diventa il test del ciclo while
-la sequenza di istruzioni del ramo then che precedono la chiamata ricorsiva diventa la prima parte del corpo del ciclo while
-la chiamata ricorsiva si trasforma in un insieme di assegnamenti che in genere faranno uso di variabili temporanee, questi assegnamenti sono la parte finale del corpo del while
-le istruzioni del caso base diventano le istruzioni da eseguire all'uscita del ciclo while

Ricorsione e liste
Molte funzioni su liste possono essere scritte in modo naturale usando la ricorsione.
Considerando una lista di interi, supponiamo di voler scrivere una funzione print-rev(*p) che stampa gli elementi delle lista puntata p in ordine inverso:
-1) il problema ha una semplice soluzione ricorsiva, infatti ragionando per induzione sulla lunghezza della lista p:

-se p è la lista vuota, non occorre fare nulla
-altrimenti si stampa in ordine inverso la lista p->next chiamando ricorsivamente preint-rev(), quindi si stamp p-> dato

-2) Se non si vuole usare la ricorsione per ottenere una soluzione efficiente in cui la lista è attraversata una sola volta occorre usare delle strutture ausiliarie, quando un elemento della lista è stato visitato, occorre salvare il valore dell'elemento e poi stampare i valori salvati in ordine inverso, dato che il primo elemento salvato è l'ultimo che verrà usato, la struttura più adatta in cui salvare gli elementi è uno stack, occorre quindi usare le funzioni per gestire la lista ed anche le funzioni per gestire lo stack.

Confronto:
la soluzione 1 è più elegante e non richiede strutture ausiliarie, nella soluzione 2 occorre definire esplicitamene uno stack il cui ruolo è quello di simulare lo stack di sistema.
-se si hanno liste grosse la soluzione 1 può portare ad un errore se viene esaurita la memoria disponibile.

LEZIONE 15 – Programmazione dinamica e Grady

Dato un problema non ci sono regole generali per risolverlo in modo efficiente, ma è comunque possibile proporre 4 fasi standard per affrontarlo:

-{1} classificazione del problema
problemi decisionali : il dato soddisfa delle determinate proprietà?
-problemi di ricerca :
-spazio di ricerca: ovvero insieme di soluzioni possibili
-soluzione ammissibile: soluzione che rispetta certi vincoli
-{2} caratterizzazione della soluzione
-definire la soluzione dal punto di vista matematico
-formulare una prima die di soluzione
-scegliere una possibile tecnica

-{3} tecnica di progetto
    -divide-et-impera : un problema viene suddiviso in sotto problemi indipendenti che vengono risolti ricorsivamente
    -programmazione dinamica*: la soluzione viene costruita a partire da un insieme di sotto problemi potenzialmente ripetuti (problemi non indipendenti)
    -algoritmo Greedy: ad ogni passo si sceglie la soluzione che è localmente ottima
-{4} utilizzo di strutture dati

Programmazione dinamica*
Utilizza una tecnica iterativa sfruttando un approccio top-down.
Risulta più efficiente della divide-et-impera nel caso in cui i problemi non sono indipendenti (il divide-et-impera li risolve più volte i sotto problemi uguali nello stesso modo).
Il divide-et-impera utilizza una tecnica ricorsiva con l'approccio top-down (problemi divisi in sotto problemi) e quindi risulta vantaggioso quando i sotto problemi sono del tutto indipendenti altrimenti converrebbe utilizzare la programmazione dinamica.

Quando utilizzare la programmazione dinamica?
-Sotto struttura ottimale
E' possibile combinare le soluzioni dei sotto problemi per trovare la soluzione di un problema più grande. Le decisioni vanno prese per risolvere un problema rimangono valide quando esso diviene un sotto problema di un problema più grande

-Sotto problema
un sotto problema può occorrere più volte

-spazio dei sotto problemi
deve esser polinomiale

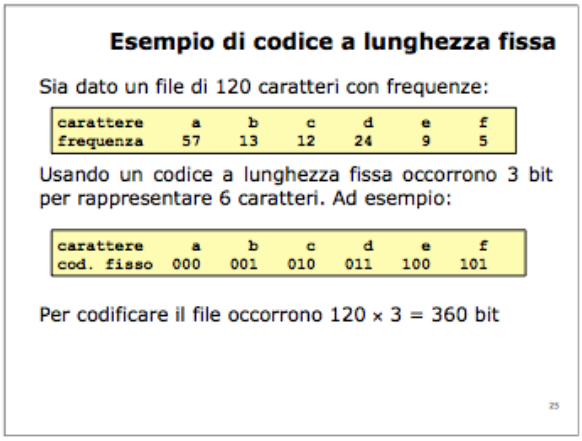
Fasi della programmazione dinamica:
-caratterizzare la struttura si una soluzione ottima
    mostrare che una soluzione ottima si ottiene da soluzioni ottime di sottoproblemi.
-definire ricorsivamente il valore di una soluzione ottima
    la soluzione ottima ad un problema contiene le soluzioni ottime ai sotto problemi
    quindi, esprimere ricorsivamente la soluzione ottima in termini di soluzioni di sottoproblemi.
-calcolare il valutare di una soluzione ottima
    si usa una tabella per memorizzare le soluzioni dei sotto problemi
    evitare di ripetere il lavoro più volte utilizzando la tabella
-costruire una soluzione ottima

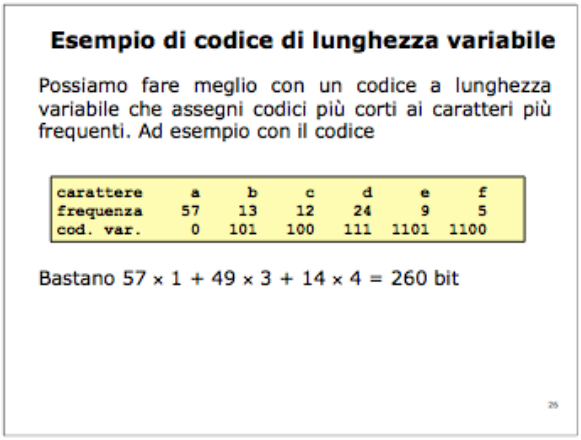
Algoritmo Greedy
ogni volta viene effettuata la scelta migliore localmente, in questo modo per alcuni problemi si ottiene una soluzione globalmente ottima!

Elementi:
    -sotto struttura ottima : ogni soluzioni ottima non elementare si compone di soluzioni ottime di sotto problemi.

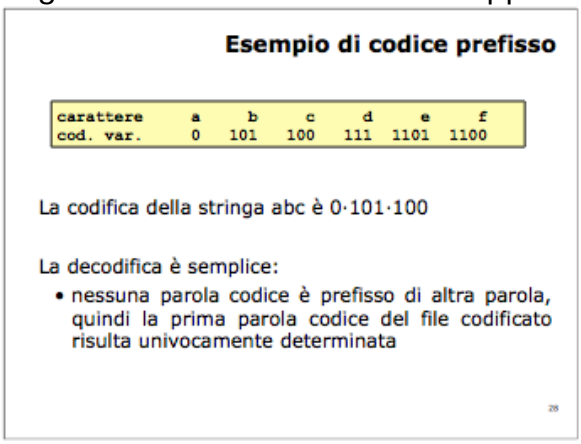
-proprietà della scelta Greedy : la scelta ottima localmente non pregiudica a possibilità di arrivare ad una soluzione globalmente ottima.

Codici di Huffman
vengono usati nella compressioe dei dati, permettono un risparmio compreso tra il 20 ed il 90% a seconda del tipo di file.
Sulla base delle frequenza con cui ogni carattere appare nel file, l'algoritmo Greedy di Huffman trova un modo ottimale di associare ad ogni carattere una sequenza di bit detta parola codice.





Codici prefissi
un codice prefisso è un codice in cui nessuna parola codice è prefisso di una ulteriore parola, ogni codice a lunghezza fissa è ovviamente prefisso ma anche il codice a lunghezza variabile che abbiamo appena visto è un codice prefisso.

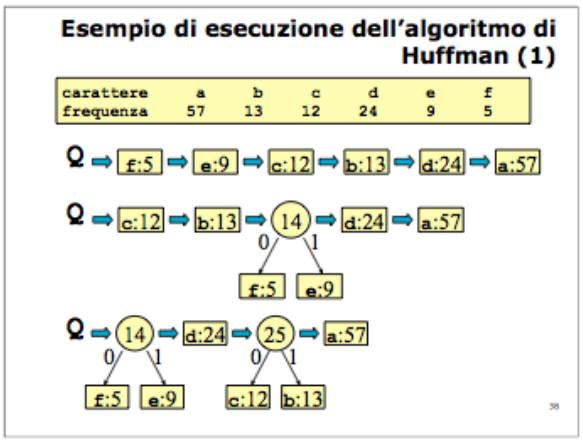


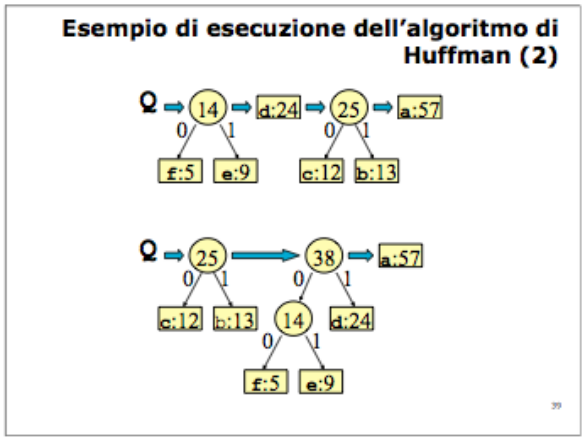
Step:
-individuare la prima parola codice del file codificato
-tradurla nel carattere originale e aggiungere tale carattere al file decodificato
-rimuovere la parola codice dal file codificato
-ripetere l'operazione per i successivi caratteri

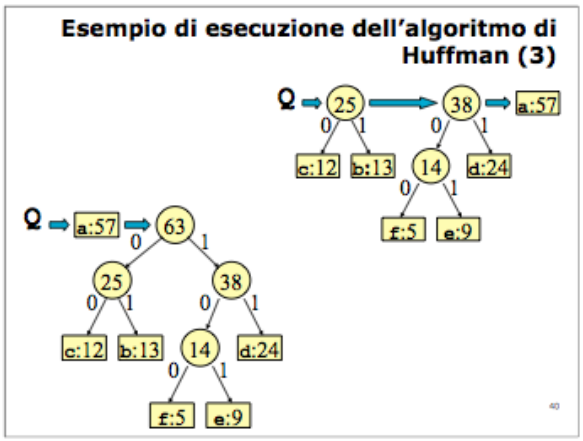
codice ottimo: dato un file F un codice C è ottimo per F se non esiste un altro codice tramite il quale F possa essere compresso impiegando un numero inferiore di bit (i codici a prefisso ottimi sono rappresentati da un albero in cui tutti i nodi interni hanno due figli

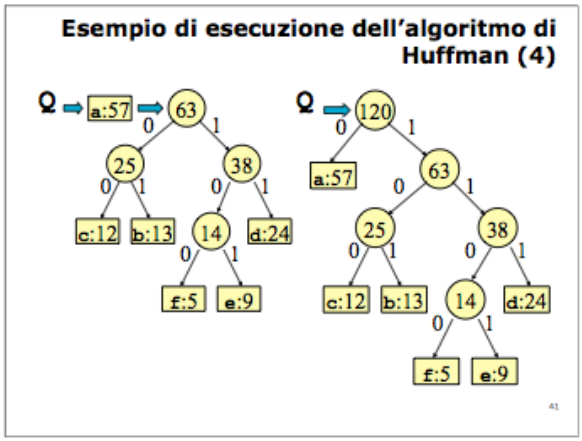
Principio del codice di Huffman:
-minimizzare la lunghezza dei caratteri che compaiono più frequentemente
-assegnare ai caratteri con la frequenza minore i codici corrispondenti ai percorsi più lunghi all'interno dell'albero

un codice è progettato per un file specifico:
-si ottiene la frequenza di tutti i caratteristiche-si costituisce il codice
-si rappresenta il file tramite il codice
-si aggiunge al file una rappresentazione del codice









-Complessità: O(n log n) con n=numero di caratteri dell'alfabeto

Questo algoritmo è Greedy perchè ad ogni passo costruisce il nodo interno avente frequenza minima possibile.