Appunti di Algoritmi e strutture dati

Filippo Bisconcin

2018

Corso

Libri:

- [DFI] C. Demetrescu, I. Finocchi, G.F. Italiano. Algoritmi e strutture dati. Seconda Edizione. McGraw-Hill, 2008.
- [CLRS] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduzione agli algoritmi e strutture dati 3/ed. McGraw-Hill, 2010.

Lezioni:

1. Lezione (20/09/2017): MP

- Introduzione al corso

2. Lezione (21/09/2017): MP

- Algoritmi e loro complessità: i numeri di Fibonacci ([DFI] cap. 1)

3. Lezione (27/09/2017): MP

- Notazione asintotica: le classi O, Omega, Theta ([DFI] 2.2; [CLRS] 3.1)

4. Lezione (28/09/2017): MP

- Esercizi sulla notazione asintotica. Le classi o, omega ([CLRS] 3.1)

5. Lezione (04/10/2017): MP

- Ulteriori esercizi sulla notazione asintotica.
- Delimitazioni inferiori e superiori. Metodi di analisi ([DFI] 2.3, 2.4)

6. Lezione (05/10/2017): MP

- Ricorrenze. Metodi di iterazione e sostituzione ([DFI] 2.5; [CLRS] 4.1)

7. Lezione (11/10/2017): MP

- Divide et impera. Il teorema fondamentale delle ricorrenze (o "Teorema master"). Esercizi. ([DFI] 2.5; [CLRS] 4.3)

8. Lezione (12/10/2017): MP

- Dimostrazione del Teorema master

9. Lezione (18/10/2017): AR

- I tipi di dato e le strutture dati. ([DFI] pp. 63-65)
- Tecniche per rappresentare collezioni di oggetti: strutture indicizzate. ([DFI] pp. 65-68)

10. Lezione (19/10/2017): AR

- Tecniche per rappresentare collezioni di oggetti: strutture collegate. ([DFI] pp. 68-70)

11. Lezione (25/10/2017): AR

- Esercizi su calcolo della complessità
- Esercizi su array

12. Lezione (26/10/2017): AR

- Esercitazione in laboratorio: esercizi su array

13. Lezione (2/11/2017): AR

- Le pile e le code. ([CLRS] pp. 192-195)

14. Lezione (8/11/2017): AR

- Implementazione in C dei tipi di dato.
- Esercizi su pile e code.

15. Lezione (9/11/2017): AR

- Il tipo di dato Lista. Realizzazione con strutture concatenate ([CLRS] pp. 195-199)
- Le invarianti di ciclo e la correttezza ([CLRS] p. 16)

16. Lezione (15/11/2017): AR

- Realizzazione delle liste con array. ([CLRS] pp. 199-203)
- Esercizi su liste

17. Lezione (16/11/2017): AR

- Gli alberi: definizioni. ([CLRS] pp. 977-979)
- Dimostrazioni per induzione di proprietà degli alberi
- Realizzazione degli alberi con vettori

18. Lezione (22/11/2017): AR

- Realizzazione degli alberi con strutture concatenate. ([CLRS] pp. 203-205)
- Visite di alberi. ([DFI] pp. 77-80)

19. Lezione (23/11/2017): AR

- Esercitazione in laboratorio: esercizi su liste e alberi generali

20. Lezione (29/11/2017): AR

- Esercizi su alberi binari

21. Lezione (30/11/2017): AR

- Alberi Binari di ricerca. (Prima parte [CLRS] pp. 237-244)

22. Lezione (13/12/2017): AR

- Alberi Binari di ricerca: cancellazione ([CLRS] pp. 244-247)
- Cenni su alberi AVL, B-alberi e alberi rosso-neri.

23. Lezione (14/12/2017): AR

- Esercizi su alberi binari di ricerca e alberi generali

24. Lezione (20/12/2017): AR

- Il problema dell'ordinamento. ([CLRS] pp. 123-125)
- Ordinare in tempo quadratico: Insertion sort. ([CLRS] pp. 14-17 e pp. 21-25)
- L'algoritmo mergesort. ([CLRS] pp. 25-33

25. Lezione (21/12/2017): AR

- Esercitazione in laboratorio: esercizi su alberi binari e alberi binari di ricerca.

26. Lezione (7/02/2018): AR

- L'algoritmo quicksort. ([CLRS] pp. 141-147)

27. Lezione (8/02/2018): AR

- Versione randomizzata di quicksort ([CLRS] pp. 148)
- Quicksort con elementi di valore uguale
- Definizione e proprietà dell'heap ([CLRS] pp. 127-129)

28. Lezione (14/02/2018): AR

- L'algoritmo heapsort. ([CLRS] pp. 129-135)

29. Lezione (15/02/2018): AR

- Code di priorità. ([CLRS] pp. 135-140)
- Esercizi

30. Lezione (21/02/2018): AR

- Limite inferiore per l'ordinamento per confronti. ([CLRS] pp. 157-159)
- Counting sort. ([CLRS] pp. 159-161)

31. Lezione (22/02/2018): AR

- Radix sort ([CLRS] pp. 162-164)
- Le tabelle ad indirizzamento diretto. ([CLRS] pp. 209-211)

32. Lezione (28/02/2018): MP

- Grafi: definizione e prime proprietà ([CLRS] 22.1, B.4)

33. Lezione (01/03/2018): MP

- Proprietà dei grafi connessi e aciclici ([CLRS] B.4)

34. Lezione (07/03/2018): AR

- Le tabelle Hash. ([CLRS] pp. 211-213)
- Analisi dell'hashing con concatenamento. ([CLRS] pp. 213-216, teoremi senza dimostrazione)

35. Lezione (08/03/2018): AR

- Le funzioni Hash. ([CLRS] pp. 216-219)
- Tabelle Hash con indirizzamento aperto. ([CLRS] pp. 223-225)

Algoritmi e loro complessità: i numeri di Fibonacci

[DFI] cap. 1

$$f(n) = \begin{cases} 1 & \text{se } n = 1, n = 2 \\ (n-1) + f(n-2) & \text{se } n \geq 3 \end{cases} \tag{1}$$

Formula di Binet

$$\forall n \in N \tag{2}$$

$$F_n = \frac{1}{\sqrt{5}}\left(\Phi^n - \overline{\Phi}^n\right)$$

Dove con Φ si indica la sezione aurea $\frac{1+\sqrt{5}}{2} \simeq -0,618$ e con $\overline{\Phi}$ si indica $\frac{1-\sqrt{5}}{2} \simeq -0,618$

Dimostrazione

Dimostriamo per Induzione la formula di Binet:

Base #1

$n = 1$

$\frac{1}{\sqrt{5}}(\Phi^1 - \overline{\Phi}^1)$

sostistuisco con le definizioni di Φ e semplifico

$= 1 = F_1$

Base #2

$n = 2$

$\frac{1}{\sqrt{5}}(\Phi^2 - \overline{\Phi}^2)$

sostistuisco con le definizioni di Φ e semplifico

$= 1 = F_2$

Passo induttivo

$$n \geq 3$$

$$F_n = F_{n-1} + F_{n-2}$$

$$F_{n-1} = \frac{1}{\sqrt{5}}\left(\Phi^{n-1} - \overline{\Phi}^{n-1}\right) \text{ e } F_{n-2} = \frac{1}{\sqrt{5}}\left(\Phi^{n-2} - \overline{\Phi}^{n-2}\right)$$

assomiglia alla forma $F_n = \frac{1}{\sqrt{5}}\left(\Phi^n - \overline{\Phi}^n\right)$

ci chiediamo se $\Phi^n = \Phi^{n-1} + \Phi^{n-2}$ e se $\overline{\Phi}^n = \overline{\Phi}^{n-1} + \overline{\Phi}^{n-2}$

divido da entrambe le parti per Φ^{n-2} e $\overline{\Phi}^{n-2}$

Definizione #1

Utilizzeremo la notazione T_n per indicare la velocità/complessità di una particolare funzione (numero di righe di codice eseguite)

Fib1 (int n) $\rightarrow$ int

return $\frac{1}{\sqrt{5}}\left(\Phi^n - \overline{\Phi}^n\right)$

$$T(Fib1)_n = 1 \; \forall \, n$$

Problema: con l'aumentare di n, la funzione Fib1(n) diventa sempre più imprecisa

$$Fib1_3 = 1{,}9999\ldots \simeq 2$$

$$Fib1_{16} = 986{,}699\ldots \simeq 987$$

$$Fib1_{18} = 2583{,}1\ldots \neq Fib_{18} = 2584$$

Definizione #2

```
Fib2 (int n) → int
    if n <= 2 then return 1
    else return Fib2(n-1) + Fib2(n-2)
```

$$T(Fib2)_n = \; 1 \qquad n = 1[?][?][?]\, n = 2$$

$$2 + T_{n-1} + T_{n-2} \qquad \forall \, n \geq 3$$

$T(Fib2)_n$ è una formula ricorsiva o ricorrenza

n	T_n
1	1
2	1
3	2 + 1 + 1 = 4
4	2 + 4 + 1 = 7
5	2 + 7 + 4 = 13

Risolviamo la ricorrenza per determinare la complessità / bontà della funzione esaminata.

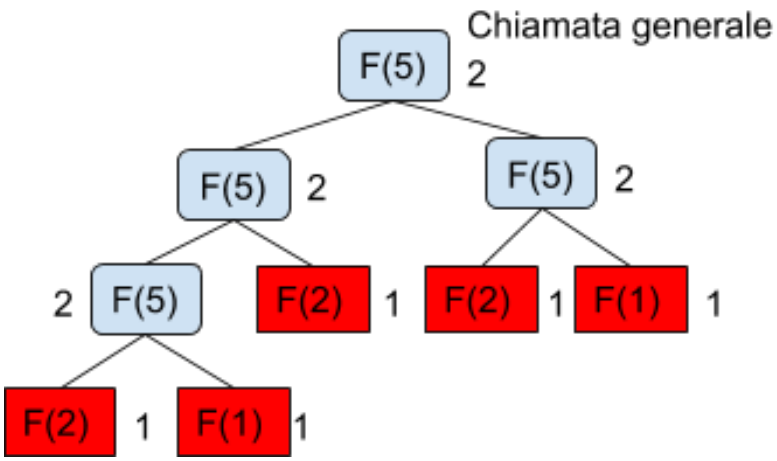
- andamento esponenziale rispetto ad n → funzione NON efficiente
- andamento logaritmico / proporzionale rispetto ad n → funzione efficiente

Strumento #1 - albero di ricorsione

Notiamo che $T(Fib2)_5 = (1 \times 5) + (2 \times 4) = 13$

dove (1 * 5) è la complessità delle foglie (5 foglie con peso 1)

e (2 * 4) è la complessità dei nodi interni (4 nodi con peso 2)



Proprietà #1

Se T_n rappresenta l'albero di ricorsione relativo a $Fib2_n$ allora il numero di foglie di T_n è uguale a F_n (con F_n indichiamo l'n-esimo numero della successione di Fibonacci)

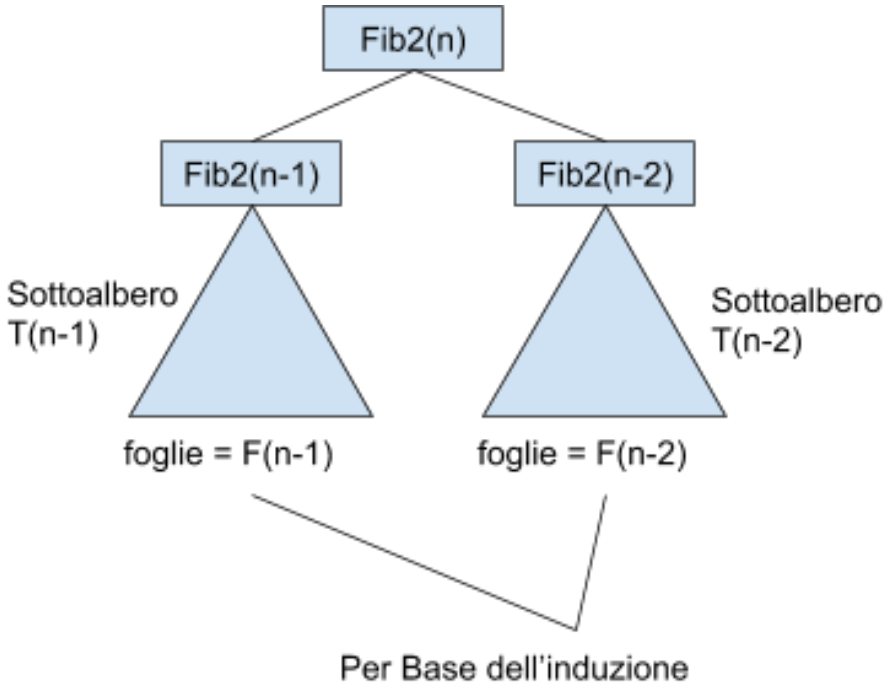
Dimostrazione per induzione

Base:

se n = 1 → numero di foglie di $T_1 = 1 = F_1$

se n = 2 → numero di foglie di $T_2 = 1 = F_2$

Passo induttivo:



Proprietà #2

Sia$T$un albero binario in cui ogni nodo interno ha esattamente 2 figli

Allora $i_T = f_T - 1$

dove con $i_T$indichiamo il numero di nodi interni dell'albero T

e con $f_T$indichiamo il numero di foglie dell'albero T

Dimostrazione per induzione

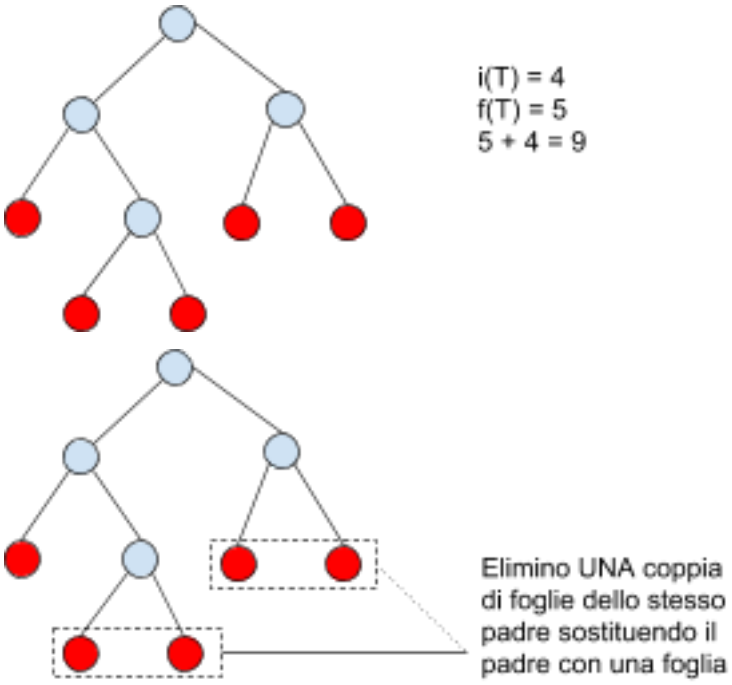
dimostrazione su n, dove n è il numero di nodi di T

Base:

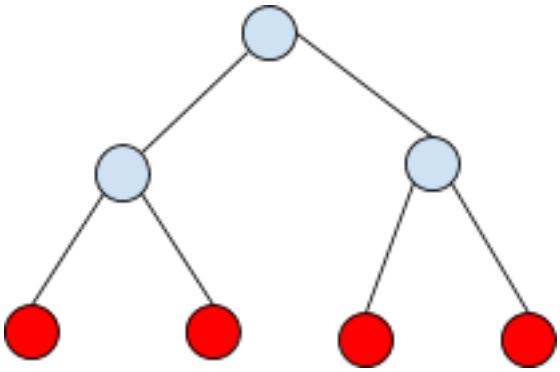
$n = 1 \rightarrow$ OVVIO

Passo induttivo:

$n \geq 2$



L'albero risultante lo chiamiamo $\overline{T}$:



Notiamo che

$$i_T = f_{\overline{T}} - 1_*$$

$$f_{\overline{T}} = f_T - 1_{**}$$

$$i_{\overline{T}} = i_T - 1$$

→ (inverto) $i_T = i_{\overline{T}} + 1$

→ (sostituisco *) $i_T = f_{\overline{T}} - 1 + 1$

→ (semplifico) $i_T = f_{\overline{T}}$

→ (sostituisco **) $i_T = f_T - 1$

Studio complessità di $Fib2_n$ con l'utilizzo del metodo dell'albero di ricorsione

$$i_{T_n} = F_n - 1$$
$$f_{T_n} = F_n$$
$$T_n = 2 \times i_{T_n} + 1 \times f_{T_n}$$
$$\rightarrow 2 \times F_n - 1 + F_n$$
$$\rightarrow 3 \times F_n - 2$$

Proprietà #3

$$\forall\, n \geq 6 \rightarrow F_n \geq 2^{\frac{n}{2}}$$

Dimostrazione per induzione

Base:

$$n = 6$$
$$F_6 = 8$$
$$2^{\frac{6}{2}} = 2^3 = 8$$

Passo induttivo:

$$n \geq 7$$

$$F(n) \geq 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}}$$

$$F(n) \geq 2^{\frac{n}{2}} * 2^{-\frac{1}{2}} + 2^{\frac{n}{2}} * 2^{-1}$$

$$F(n) \geq 2^{\frac{n}{2}} * (2^{-\frac{1}{2}} + 2^{-1})$$

$2^{-\frac{1}{2}} + 2^{-1}$ è sempre maggiore o uguale ad 1

Fib2(n) risulta essere troppo poco efficiente

$$T(8) = 61$$

$$T(45) = 3{,}404{,}709{,}508$$

Definiamo allora Fib3(n) utilizzando l'iterazione al posto della ricorsione

```
Fib3 (int n) –> int
    //Allocazione di un array lungo n;
    F(1) = 1; F(2) = 1;
    For i = 3 to n
        F(i) = F(i-1) + F(i-2);
    Return F(n)
```

$$T(fib3,n) = 3 + (n-2) + (n-1) = 2n$$

Ci chiediamo se fib3 sia efficiente dal punto di vista della memoria...

Notazione asintotica: le classi O, Omega, Theta

[DFI] 2.2; [CLRS] 3.1

Esercizi sulla notazione asintotica. Le classi o, omega

[CLRS] 3.1

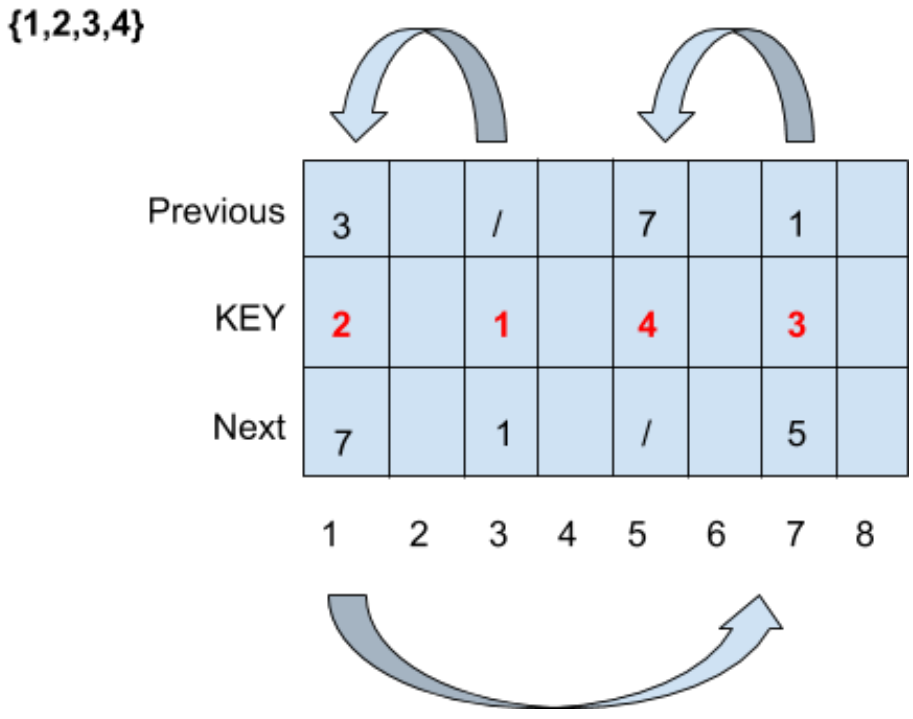
Divide et impera. Il teorema fondamentale delle ricorrenze (o "Teorema master"). Esercizi.

[DFI] 2.5; [CLRS] 4.3

Rappresentazione di una lista attraverso un vettore (matrice)

La lista è doppiamente concatenata, ovvero contiene riferimento sia all'elemento precedente che a quello successivo.

La costante NULL viene rappresentata da un indice che non appartiene all'insieme degli indici del vettore (0 in pseudocodice, -1 in C)



Analogamente si può creare una lista singolarmente concatenata chiamata FreeList contenente le celle libere (i campi Key e Previous si possono ignorare). Essa verrà utilizzata per l'allocazione di una nuova lista. FreeList è una variabile globale.

All'inizio [...] la FreeList contiene TUTTI gli oggetti non allocati.

Allocate_Object() Complessità $\Theta(1)$

```
if(free == null)
        errore "spazio esaurito"
    else
        x = free
        free = nexy[free]
        return x
```

#Inserisce la cella da liberare in testa alla FreeList

Free_Object(x) Complessità $\Theta(1)$

```
next[x] = free
    free = x
```

Alberi

[CLRS] pp. 977-979

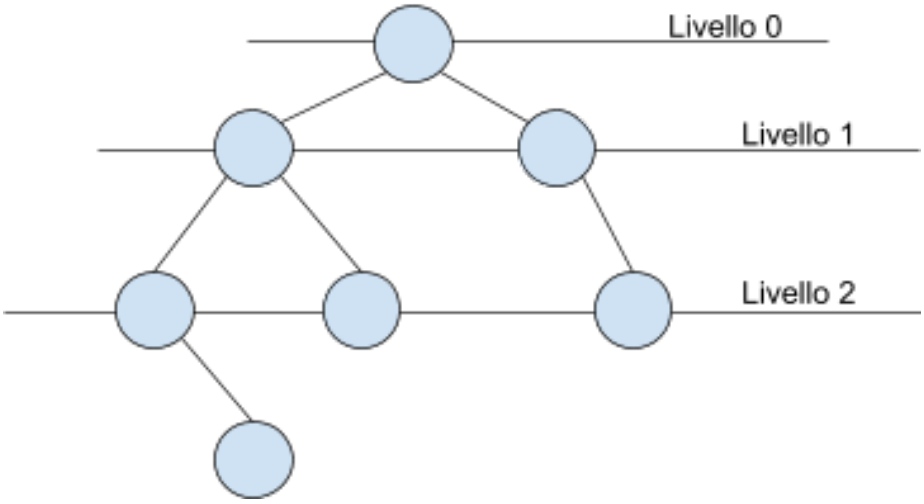
L'albero è un tipo particolare di grafo connesso, aciclico e non orientato.

Un albero radicato è una coppia $T = (N,A)$

N è un insieme finito di nodi fra cui si distingue un nodo R detto 'Radice'.

A è un sottoinsieme del prodotto cartesiano $(N\ x\ N)$,è un insieme di coppie di nodi che chiamiamo archi.

In un albero, ogni nodo v (eccetto la radice) ha esattamente un genitore che viene chiamato padre u, tale che $(v,u) \in A$



Un nodo può avere 0 o più figli, un figlio è tale se esiste un arco $(v,u) \in A$

Il numero dei figli di un nodo si chiama GRADO del nodo.

Un nodo senza figli è detto FOGLIA o NODO ESTERNO.

Un nodo non foglia è un NODO INTERNO.

Se due nodi hanno lo stesso padre, allora sono fratelli.

Il cammino da v ad v' nell'albero T è una sequenza di nodi $n_0, n_1, n_2, \dots, n_k$ tale che soddisfa queste condizioni:

$$v = n_0$$

$$v' = n_k$$

$$(n_{i-1}, n_i) \in A \ \forall 1 < i < k$$

La lunghezza di un cammino è il numero di archi nel cammino oppure il numero di nodi - 1.

Sia x un nodo dell'albero radicato T con radice r.

Qualsiasi nodo y in un cammino che parte dalla radice r ed arriva ad x è detto ANTENATO di x (x compreso, è antenato di se stesso).

Se y è un antenato di x, allora x è DISCENDENTE di y

NB: Ogni nodo è DISCENDENTE ed ANTENATO di sè stesso.

Se y è un antenato di x ed x è diverso da y, allora y è un ANTENATO PROPRIO di x ed x è DISCENDENTE PROPRIO di y.

Il sottoalbero con radice in x è l'albero indotto dai discendenti di x

La profondità di un nodo x è la lunghezza del cammino dalla radice ad x.

Un livello di un albero è costituito da tutti i nodi che stanno alla stessa profondità.

L'altezza di un nodo x è la lunghezza del più lungo cammino che scende da x alle foglie (qualsiasi foglia) di profondità massima.

L'altezza di un albero è l'altezza del nodo radice.

L'altezza è la massima profondità di un qualsiasi nodo dell'albero.

Alberi binari (Albero k-ario con K = 2)

Gli alberi binari sono definiti in modo ricorsivo:

- Un albero vuoto è binario
- Un albero costituito da un nodo (radice) ,

  da un albero binario detto sottoalbero sinistro della radice,

  e da un'altro albero binario detto sottoalbero destro della radice

  è detto un albero binario.

Albero k-ario

E' un albero in cui i figli si un nodo sono etichettati con interi positivi distinti e le etichette maggiori di K sono assenti = ogni nodo può avere al più K figli.

Un albero K-ario completo è tale quando tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno grado K

ALGORITMO

Trovare l'algoritmo per determinare la completezza di un albero

Dimostro per induzione che le foglie sono K^h

$h = 0$ (caso base) $K^0 = 1$ OK

assumiamo che per un albero di altezza h sia vero che $n = K^h$

lo dimostro per $h+1$

il numero di nodi di profondità h è K^h per ipotesi

il numero di nodi di profondità $h+1$ è $K^{h} * K = K^{h+1}$ OK

ALGORITMO

Trovare il numero di foglie e il numero di nodi interni di un albero k-ario completo di altezza h

$\sum_{\varphi=0}^{h-1} K^{\varphi}$ è una sommatoria GEOMETRICA, si semplifica in $\frac{K^{h-1+1}-1}{K-1} con\ K \neq 1$, quindi $\frac{K^h-1}{K-1}$

Altezza di un albero k-ario completo con n nodi:

$$n = K^h \rightarrow log_k(n) = log_k(K^h) \rightarrow log_k(n) = h$$

Proprietà

Dimostrare per induzione che in un albero BINARIO completo non nullo avente $n$nodi, il numero di foglie è $\frac{n+1}{2}$

Tipo di dato ALBERO

Struttura:

Insieme di nodi

Insieme di archi

Le operazioni del tipo ALBERO:

newTree() → T	Nuovo albero T
numNodi(T) → int	Numero di nodi presenti in T
grado(T,N) → int	Numero di figli di N, N appartenente a T
padre(T,N) → int	Nodo padre, null se N è radice, N appartenente a T

figli(T,N) → int[] Lista contenente i figli del nodo N, N appartenente a T

Rappresentazione di alberi tramite array

1.

Utilizzo del vettore padri

sia $T = (N,A), N = \{n_1, n_2, n_3, n_4, n_5, \dots, n_n\}$

Costruisco un vettore di dimensione n, le cui celle contengono copie di (i,u). Con i indichiamo l'informazione del nodo e con u indichiamo il nodo padre (indice).

$\forall v \in [1,n]$

p[v] → info = contenuto

p[v] → padre = indice del padre, $(u,v) \in A(archi)$, se v è radice il padre è NULL/-1/0

Spazio per memorizzare $n = \Theta(n)$

Implementazioni:

Padre - Complessità $\Theta(1)$

```
padre(Tree P, Node v)
        if ( p[v] == 0 )
                return null
    else
            return p[v] → parent;
```

Figli - Complessità $\Theta(n)$

```
figli(Tree P, Node v)
    l = crealista()
    for i = 1 to n
        if ( p[i] → parent == v )
            inserisci(i,l)
    return l
```

---

2) Utilizzo del vettore posizionale

L'albero deve essere completo e con $K \geq 2$. La ricerca è ottimizzata rispetto all'albero dei padri. Ogni nodo ha una posizione prestabilita.

Utilizzo un vettore posizionale P di dimensione n tale che

1. 0 è la posizione della radice
2. l'i-esimo figlio di un certo nodo v è in posizione $kv + i + 1$, con $0 \leq i < K - 1$

un nodo f è foglia se non ha figli, quindi se $kv + 1 > n$

Il padre del nodo f è in posizione PII$\left\lfloor \frac{f-1}{k} \right\rfloor$ (parte intera inferiore) ed i figli si trovano tra $kv + 1$ e $kv + 1 + i - 1$

Implementazioni:

Padre - Complessità $\Theta(1)$

```
padre(Tree P, Node v)
        if ( n == 0 )
                return null
    else
        return PII[a][(v-1) / K]
```

Figli - Complessità $\Theta(k)$, con k = grado di v

```
figli(Tree P, Node v)
    l = crealista()
    if( kv + 1 n)
        return l
    else
        for i = 0 to k-1
            inserisci(kv + 1 + i ,l)
          return l
```

3) Utilizzo di strutture collegate

3.a) Parent + Childs

Ogni nodo è un record con i seguenti campi:

k : informazione

p : puntatore al padre

Se il numero di figli è noto

left : puntatore al figlio sinistro

right : puntatore al figlio destro

Altrimenti si utilizza una lista di puntatori ai propri figli

c [] : lista di puntatori ai figli

Se ogni nodo ha grado al più k, è possibile mantenere in ogni nodo un puntatore a ciascuno dei possibili k figli

Spazio necessario: $\Theta(nk)$, se k costante allora $\Theta(n)$

Implementazioni:

Padre - Complessità $O(1)$

```
padre(Tree P, Node v)
      return v → p
```

Figli - Complessità $O(k)$, con k = grado di v

Con k non noto si utilizza un ciclo che scorre c[]

```
figli(Tree P, Node v)
    l = crealista()
    if( v → left <>null )
          inserisci(v → left ,l)
    if( v → right<>null )
          inserisci(v → right,l)
    return l
```

3.b) Parent + Left child + Right sibling

Ogni nodo è un record con i seguenti campi:

k : informazione

p : puntatore al padre

left_child : puntatore al figlio sinistro

right_sibling : puntatore al fratello immediatamente a destra

Implementazioni:

Padre - Complessità $\Theta(1)$

```
padre(Tree P, Node v)
       return v → p
```

Figli - Complessità $\Theta(k)$ con k = grado di v

```
figli(Tree P, Node v)
    l = crealista()
    iter = v → left_child
    while ( iter <> null)
            inserisci(iter,l)
          iter = iter → right_sibling
    return l
```

Algoritmi Di Visita Degli Alberi

[DFI] pp. 77-80

Visita generica

```
VisitaGenerica(Node v)
        s = { v }
        while ( s <> {} )
                u = pop(s)
                figli = VisitaGenerica(u)
                s = s U figli
        return s
```

Dimostrare che ha costo LINEARE[b]

Teorema

L'algoritmo di visita, applicato alla radice di un albero con m nodi, termina in O(m) iterazioni. Lo spazio usato è O (n)

Dimostrazione

Hp: L'inserimento e la cancellazione da S sono effettuati in tempo costante

Ogni nodo verrà inserito ed estratto dall'insieme s una sola volta, perchè in un albero non si può tornare ad un nodo a partire dai suoi figli.

Quindi le iterazioni del ciclo while saranno al più O(n)

Poiché ogno nodo compare al più una volta in S, lo spazio richiesto non è più alto di O(n)

Visita DFS - Depth first search - Ricerca in profondità

Seguiamo tutti i figli sinistri, andando in profondità fino a che non si raggiunge la prima foglia sinistra. Solo quando il sottoalbero sx è stato completamente visitato, si passa a visitare il sottoalbero dx

```
Dfs_Visit(Node r)
        s = stack()
        push (r,s)
        while(not stackempty(s))
                u = pop(s)
                if(u <> null)
                        Dfs_Visit(u)
                        push(u → right, s)
                        push(u → left , s)
        return s
```

Visita BFS - Breadth first search - Ricerca in ampiezza

Heap

Heap: albero quasi completo con tutti i nodi dell'ultimo livello a sinistra

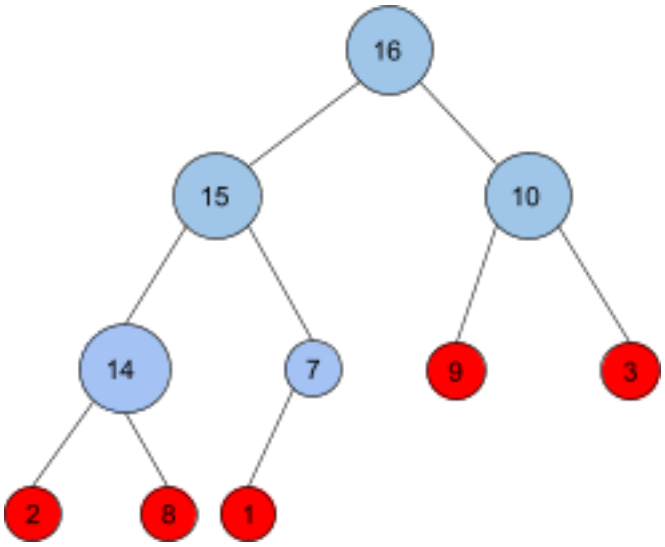
MaxHeap: La radice ha valore maggiore o uguale a quello dei figli

MinHeap: La radice ha valore minore o uguale a quello dei figli

Lemma 2:

Nell'array che rappresenta un heap di n elementi, le foglie sono i nodi con indici che vanno alle posizioni

$$\frac{n}{2}+1, \frac{n}{2}+2, \ldots, n$$



Elemento

16

15

10

14

7

9

3

2

8

1

Posizione

1°

2°

3°

4°

5°

6°

7°

8°

9°

10°

$\frac{n}{2}$

$\frac{n}{2}+1$

$\frac{n}{2}+2$

$\frac{n}{2}+3$

$\frac{n}{2}+4$

n

Lemma 3:

Il teorema definisce la numerosità di nodi con una certa altezza:

Ci sono al massimo $\frac{n}{2^{h+1}}$ nodi di altezza[c] h in un qualsiasi heap di n elementi

Max_heapify

L'operazione max_heapify permette di mantenere le proprietà di maxheap

Precondizioni:

Gli alberi binari con radice in left(i) e right(i) sono maxheap

Postcondizioni:

L'albero radicato in i è un maxheap

```
Max_heapify( Heap A, Node i)
        l = left(i)
        r = right(i)
        if ( l <= A.heapsize AND A[l] > A[i] )
                massimo = l
        else
                massimo = i


        if ( r <= A.heapsize AND A[r] > A[massimo] )
                massimo = r


        if ( massimo != i )
                scambia ( A[i] , A[massimo] )
                max_heapify[d](A, massimo)
```

Tempo di esecuzione : O(h) dove h è l'altezza del nodi i poichè l'heap ha altezza log(n) (LEMMA 2)

Dato un vettore disordinato, costruire un heap

```
Build_maxheap ( Array A )
        A.heapsize = A.length
        for i in PII[e] ( A.length / 2 ) downto 1
                max_heapify ( A , i )
```

Invariante: ogni nodo in posizione $i+1, \dots, n$ è radice di un maxheap, con n =

Sembrerebbe complessa $O\left(\frac{n}{2} log(n)\right) = O\left(n \, log(n)\right)$ ma max_heapify lavora principalmente su foglie, quindi la complessità è lineare $O\left(n\right)$.

$$\sum_{h=0}^{log(n)} PIS(\frac{h}{2^{h-1}}) * O(h) = O(\, n * \sum_{h=0}^{log(n)} PIS(\frac{h}{2^h})\,)$$

L'ultima sommatoria è la serie nota $\sum_{h=0}^{+inf} h * x^h = \frac{x}{(1-x)^2}$

Con $x = \frac{1}{2}$, $\sum_{h=0}^{+inf} h * x^h = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$

Quindi $O(\, n * \sum_{h=0}^{log(n)} PIS(\frac{h}{2^h})\,) = O(2n) = O(n)$

Heapsort

```
HeapSort ( Array A )
        build_maxheap ( A )
        for i = A.length downto 2
                scambia A[ 1 ] e A[ i ]
                A.heapsize -= 1
                max_heapify ( A , 1 )
```

INV = Il sottoarray che va dalla posizione 1 alla posizione i è un maxheap che contiene gli elementi più piccoli del intero vettore di partenza, mentre $A[\, a+1, [?][?][?]., n\,]$ contiene gli $n-1$ elementi più grandi di $A[\, 1, \ldots, n\,]$ ordinati.

Teorema

L'algoritmo HeapSort ordina in loco n elementi eseguendo nel peggiore dei casi $O(\, n\, log(n)\,)$ confronti in quanto algoritmo basato sui confronti.

Code di priorità

[CLRS] pp. 135-140

Struttura dati che serve a mantenere un insieme dinamico i cui elementi (aggiungibili e rimovibili) hanno un valore associato detto chiave o peso.

Esistono due tipi di code di priorità:

- MaxPriorità (Si utilizza la struttura MaxHeap)
- MinPriorità (Si utilizza la struttura MinHeap)

Le operazioni delle code di priorità massima/minima sono:

Insert(Coda s , Elemento X)

Inserisce l'elemento X in S

Maximum(Coda s)

Restituisce l'elemento di S con la chiave maggiore senza rimuoverlo

Minimum(Coda s)

Restituisce l'elemento di S con la chiave minore senza rimuoverlo

Extract_max(Coda s)

Elimina e restituisce l'elemento di S con la chiave maggiore

Extract_min(Coda s)

Elimina e restituisce l'elemento di S con la chiave minore

Increase_key(Coda s, Elemento x, Chiave k)

Incrementa il valore della chiave di X al nuovo valore K. Si suppone che K sia maggiore o uguale al valore corrente della chiave di X

$$K \geq chiave(X)$$

Decrease_key(Coda s, Elemento x, Chiave k)

Decrementa il valore della chiave di X al nuovo valore K. Si suppone che K sia minore o uguale al valore corrente della chiave di X

$$K \leq chiave(X)$$

Implementazione di code di massima priorità con le strutture Heap

---

```
Heap_maximum(Heap A)
    if ( A.heapSize < 1 )
        error “heap underflow”
    else
        max = A[1]
        return max
```

Complessità di Heap Maximum : $O(1)$

```
Heap_extract_max(Heap A)
    if ( A.heapSize < 1 )
        error “heap underflow”
    else
        max = A[1]
        A[1] = A[ A.heapSize ]
        A.heapSize -= 1
        max_heapify[f] ( A , 1[g] )
      return max
```

Complessità di Heap Extract Max :$O(log(n))$

```
Heap_Increase_Key(Heap A, Nodo i, Key K)
    if ( K < A[i] )
        error “la nuova chiave è più piccola di quella esistente”
    else
        A[i] = K
        while[h][i] ( i > 1 AND A[i] > A[parent(i)] )
            scambia ( A[i] , A[parent(i)] )
            i = parent(i)
```

Complessità di Heap Increase Key : $O(log(n))$

Invariante del while: L'array $A[i, ... , A.heapSize\ ]$ soddisfa le proprietà di maxHeap tranne una possibile violazione: A[i] potrebbe essere più grande di A[parent(i)]

Con un Heap di n elementi, la complessità è $O(log(n))$ in quanto il cammino dal nodo fino alla radice ha lunghezza $O(log(n))$

```
1 Heap_Insert(Heap A, Element K)
2   A.heapSize += 1
3   A[A.heapSize] = -inf
4   Heap_increase_key( A, A.heapSize , K)
```

```
Heap_Insert(Heap A, Element K)
    A.heapSize += 1
    A[A.heapSize] = -inf[j]
  Heap_increase_key[k]( A, A.heapSize, K)
```

Complessità di Heap_Insert : $O(log(n))$

La ricerca su una cosa di priorità, nel caso peggiore, ha costo $O(n)$

Premessa : $1 \leq i \leq A.heapSize$

```
Heap_Delete( Heap A, Node i)
    if A.heapSize == 1
        A.heapSize = 0
    else
        val = A[i]
        A[i] = A[ A.heapSize ]
        A.heapSize -= 1
        if ( val > A[i] )[l]
            max_heapify ( A , i )
        else[m]
            while ( i > 1 AND A[i] > A[parent(i)] )
                scambia ( A[i] , A[parent(i)] )
                i = parent(i)
```

Complessità di Heap Delete : $O(log(n))$

Esercizi

Esercizio 1 :

Scrivere una funzione che determini se un albero è quasi completo. Gli output devono essere 1 se l'albero è quasi completo o 0 se non lo è. L'albero è rappresentato con notazione left e right.

Note: notiamo che non è sufficiente sapere se un albero è quasi completo o meno, necessitiamo anche di un valore per rappresentare l'albero completo.

Gli output saranno quindi: 0 - albero completo, 1 - albero quasi completo, 2 - albero non quasi completo.

```
is_quasi_completo( Node u )
    int h
    return is_quasi_completo_aux(u,&h) == 1[n]
```

```
is_quasi_completo_aux( Node u, int *h)
        int risSx, risDx, hSx, hDx
        if ( u == NULL )
                *h = -1
                return 0 //albero completo
        risSx = is_quasi_completo_aux(u->left, &hSx) <= 1 //se completo o
quasi completo
        risDx = is_quasi_completo_aux(u->right, &hDx) <= 1 //se completo
o quasi completo
        *h = ( hSx < hDx ? hDx : hSx ) + 1 //l'altezza massima + 1
        if(
          ( risSx == 0 AND risDx == 0 AND hDx <= hSd AND hSx <=
(hDdx + 1) )
                || ( risSx == 0 AND risDx == 1 AND hSx == hDx)
                || ( risSx == 1 AND risDx == 0 AND hSx == (hDx + 1) )
        )
                return (hDx < hSx ? 1 : risDx)
        return 2 //non è quasi completo
```

Complessità di is_quasi_completo : $O(n)$

$$T(n) = T(k) + T(n - k - 1) + c = O(n)$$

Esercizio 2 :

Siano dati due alberi binari completi di radice r ed s aventi la stessa altezza h e dimensione totale (somma dei nodi dei due alberi) n. Le chiavi memorizzate nei nodi soddisfano la proprietà di maxheap. Si vuole creare un unico albero quasi completo maxheap, fusione dei due alberi, con altezza h+1 e dimensione n.

Le seguenti soluzioni vengono accettate:

1. Soluzione di costo e tempo $[?][?](n)$ e in spazio aggiuntivo $[?][?](n)$
2. Soluzione (più elegante) di costo e tempo $O(log(n))$ e spazio aggiuntivo costante

Soluzione:

1. Soluzione di costo e tempo $[?][?](n)$ e in spazio aggiuntivo $[?][?](n)$

   Vettore di n elementi v. (spazio aggiuntivo $[?][?](n)$)

   Carichiamo il vettore con i valori di r ed s. $[?][?](n)$

Applichiamo build_maxheap all'intero vettore v.$[?][?](n)$

Creo l'albero corrispondente. $[?][?](n)$

2. Soluzione di costo e tempo $O(log(n))$ e spazio aggiuntivo costante

Non possiamo fare altro che modificare le strutture a nostra disposizione, non usiamo strutture ausiliarie.

Prendo come radice il nodo foglia x più a destra dell'albero s. $[?][?](log\ n)$

Assegno x.left ad s e x.right as r.

Applico la max_heapify al nuovo nodo radice x. $[?][?](log\ n)$

Limite inferiore per l'ordinamento per confronti

[CLRS] pp. 157-159

Fino ad ora abbiamo visto i seguenti algoritmi di ordinamento, tutti basati sul confronto.

MergeSort	$[?][?]\,(\,n\,log(n)\,)$
QuickSort	Caso medio : $[?][?]\,(\,n\,log(n)\,)$, caso pessimo : $[?][?]\,(\,n^2\,)$
HeapSort	$[?][?]\,(\,n\,log(n)\,)$
InsertionSort	$[?][?]\,(\,n^2\,)$

Ci domandiamo, è possibile infrangere il limite inferiore di $[?][?]\,(\,n\,log(n)\,)$? Dimostreremo che NON è possibile farlo con algoritmi basati sul confronto.

Analizziamo il limite inferiore degli algoritmi basati sul confronto:

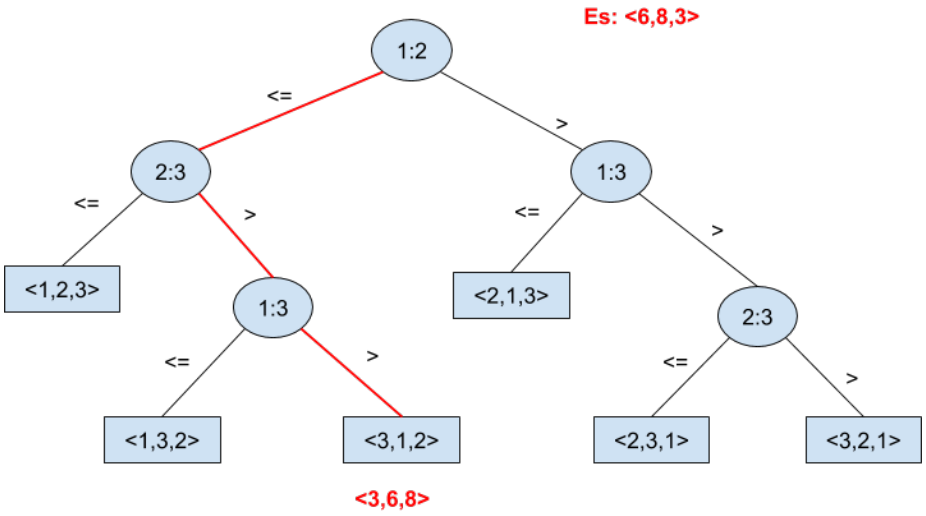
$[?][?](n)$ è il limite banale, in quanto devo analizzare n elementi. Questo limite è tuttavia irrealistico e insensato.

Tutti gli algoritmi basati sul confronto hanno come limite inferiore

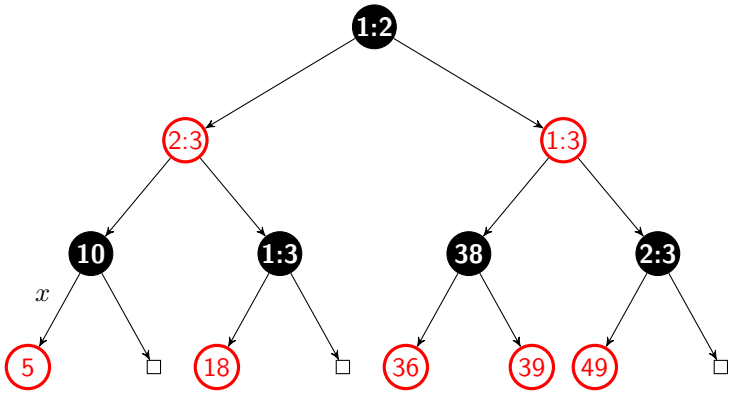
$[?][?](n\,log(n))$. Per dimostrarlo facciamo uso degli alberi di decisione.

Un albero di decisione è un'astrazione di un qualsiasi algoritmo di ordinamento basato sui confronti.

Esempio: Ordina 3 elementi



Esempio: $< 6, 8, 3 >$



Questa struttura assomiglia all'InsertionSort[o]

Per un input A di dimensione n ($A = <A_1, \ldots, A_n>$) ogno nodo

interno è etichettato da una coppia $i{:}j$ dove i e j sono indici dell'insieme da ordinare.

- $i{:}j$ significa confrontare A_i con A_j
- il sottoalbero sinistro dà i successivi confronti se $A_i \leq A_j$
- il sottoalbero destro dà i successivi confronti se $A_i > A_j$
- ogni foglia fornisce una permutazione dell'input tale che se io vado a prendere gli elementi dell'input, ordinati a seconda della permutazione, ottengo l'insieme ordinato in ordine crescente

Dato un qualsiasi algoritmo di ordinamento basato sul confronto, è possibile costruirmi gli alberi di decisione.

- Ogni valore di n è associato ad un proprio albero di decisione.
- L'albero modella tutte le traccie possibile di esecuzione.
- Il tempo di esecuzione (cioè il numero di confronti necessari) è la lunghezza di un cammino sull'albero.
- Il tempo di esecuzione nel caso peggiore è il più lungo cammino dell'albero, ovvero l'altezza dell'albero.

Mi basta quindi trovare la dimensione dell'albero di decisione per determinare il tempo di esecuzione.

Quanto è grande un albero di decisione?

Quante foglie contiene?

L'albero ha come minimo $n!$ foglie poichè, per essere corretto, ogni permutazione deve comparire almeno una volta.

Lemma 2:

Un albero binario di altezza h ha al più 2^h foglie.

Dimostrazione per induzione:

- Se $h = 0$ abbiamo un albero costituito da un solo nodo radice che è l'unica foglia, quindi le foglie sono 1. Ovviamente con $h = 0$, $1 \leq 2^h$ è verificata, infatti $1 \leq 2^0$.
- Assumiamo vera la proprietà per alberi binari di altezza $k < h$ e lo dimostro per h. Sia r la radice dell'albero t.

- Se t ha un solo figlio allora il numero di foglie di t è uguale a quello del figlio che ha altezza $h-1$. Per ipotesi induttiva: Il numero delle foglie del sottoalbero figlio è minore di 2^{h-1}, che è minore di 2^h.
- Se sono presenti entrambi il figlio sx e il figlio dx, allora il numero delle foglie è dato dalla somma delle foglie dei due sottoalberi. Siano h_L, h_R le altezze dei due figli. Entrambe sono minori di h.

$$f = f_L + f_R = 2^{h_L} + 2^{h_R} \leq 2 * 2^{max(h_L, h_R)}$$

$f \leq 2^{1 + max(h_L, h_R)}$ ma $1 + max(h_L, h_R) = h$, quindi $f \leq 2^h$

Quindi il numero delle foglie è compreso tra $n!$ e 2^h.

Teorema:

Qualsiasi algoritmo di ordinamento per confronti richiede almeno $[?][?](n\,log(n))$ confronti nel caso peggiore.

Dimostrazione:

Bisogna determinare l'altezza di un albero di decisione, dove ogni permutazione appare come foglia. Si consideri un albero di decisione di altezza h con l foglie che corrisponde ad un ordinamento per confronti di n elementi.

Allora $n! \leq l \leq 2^h$ (per Lemma 2)

Passando al logaritmo, $h \geq log\,(\,n!\,)$

Utilizziamo l'approssimazione di Stirling per approssimare $n!$: $n! \simeq \sqrt{2\,\pi n}\left(\frac{n}{e}\right)^n$

Per n sufficientemente grande, considero solo il termine più grande

$$h \geq log\,(\left(\frac{n}{e}\right)^n)$$

Per proprietà dei logaritmi

$$h \geq n * log\,(\frac{n}{e})$$

$h \geq n * (\log(n) - \log(e))$, il secondo algoritmo è costante

$h \geq n * \log(n)$

Non ci può essere quindi un algoritmo basato sui confronti minore di $n * \log(n)$

Corollario:

Gli algoritmi HeapSort e MergeSort sono algoritmi di ordinamento per confronti asintoticamente ottimali.

Dimostrazione:

I limiti superiori dei sue algoritmi sono $O(n \log n)$ nei tempi di esecuzione corrispondono al limite inferiore $[?][?](n \log(n))$ nel caso peggiore dato dal teorema.

Se elimino il modello basato sui confronti, posso battere il limite inferiore di $[?][?](n \log(n))$? SI.

Algoritmi di ordinamento privi di confronti

CountingSort

[CLRS] pp. 159-161

Assunzione:

I numeri da ordinare sono interi in un intervallo che va da 0 a k, per qualche k prefissato

Input:

$A = [1..n]$ dove $A[j] \in [0..k]$, n e k sono parametri

Output:

$B = [1..n]$ ordinato, permutazione di A

Utilizzo una memoria ausiliaria C , costituita da $k+1$elementi, $C = [0..n]$

Codice:

CountingSort(Array A, Array B, int k, int k)
for i = 0 to k
C[i] = 0 //inizializzo a 0 i contatori$[?][?](k)$

for j = 1 to n //conto le occorrenze, bound garantito da ipotesi di k $[?][?](n)$
C[A[j]] ++

for i = 1 to k //somme prefisse$[?][?](k)$
C[i] = C[i] + C[i-1]
for j = n downto 1 //ciclo di distribuzione$[?][?](n)$
B[C[A[j]]] = A[j]
C[A[j]] –

Complessità di CountingSort : $[?][?]\,(\,n+k\,)$

Di solito il CountingSort quando k è limitato superiormente da n, $k = O(n)$. Allora il tempo di esecuzione risulta $[?][?]\,(n)$. Il tempo di esecuzione di questo algoritmo è dato da $[?][?]\,(\,n+k\,)$. Di solito è utilizzato (Il CountingSort). Quando k è limitato superiormente da n in $k=O(n)$. Allora, il tempo di esecuzione risulta essere $O(n)$.

Il CountingSort è un algoritmo stabile, caratteristica molto importante.

Analisi:

All'uscita dal secondo ciclo: $C[i] = \left|\{x \in \{1..n\} \mid A[x] = i\}\right|$

All'uscita dal terzo giro: $C[i] = \left|\{x \in \{1..n\} \mid A[x] \leq i\}\right|$

RadixSort

[CLRS] pp. 162-164

1° PASSO		2° PASSO		3° PASSO		4° PASSO		
253		10		5		5		5
346		253		10		10		10
1034	→	1034	→	127	→	1034	→	127
10		5		1034		127		253
5		346		346		253		346
127		127		253		346		1034

L'algoritmo di ordinamento scelto deve essere stabili e rispettare l'ordine.

RadixSort(Array A, int d) // A contiene interi di d cifre
for i = 1 to d // 1: cifra meno significativa, d: cifra più significativa
//uso un ordinamento stabile per ordinare l'array A sulla crifra i-esima

---

Dimostriamo la correttezza dell'algoritmo tramite induzione sulla colonna da ordinare:

Caso base (i=1)

Ordino l'unica colonna

Assumo

che le cifre delle colonne che vanno da 1 a $i-1$siano ordinate

Dimostro

che un algoritmo stabile sulla colonna i, lascia le colonne da 1 a i ordinate:

Se due cifre in posizione i

- sono uguali:

per stabilità, rimangono nello stesso ordine e, per ipotesi induttiva, sono ordinate.

- sono diverse:

l'algoritmo di ordinamento sulle colonna i le ordina e le mette in posizione corretta.

Complessità: $[?][?](d * (k+n))$

Teorema:

Dati n numeri di d cifre, dove ogni cifra può avere fino a $k$valori possibili[p], la procedura ordina correttamente i numeri nel tempo di $[?][?](d * (k+n))$ se l'algoritmo stabile utilizzato dalla procedura impiega un tempo $[?][?](k+n)$

Dimostrazione:

Per ogni iterazione, il costo risulta essere$[?][?](k+n)$

Le iterazioni sono d, per un totale di $[?][?](d * (k+n))$

Osservazioni:

Se $k = O(n)$ il tempo di esecuzione è $[?][?](d * n)$

Inoltre, se d è costante, la complessità è $[?][?](n)$

Come ripartire le chiavi in cifre?

- Usiamo il CountingSort su ciascuna cifra $[?][?](k+n)$
- Siano n interi, ognuno costituito da b bits
- Divido ogni intero in $PIS(\frac{b}{r})$ "cifre", ognuna di r bits. La cifra appartiene a $[0, \dots, 2^r - 1]$, $k = 2^r$

Esempio: Parola di 32 bits

La suddivido in cifre, ciascuna di 8 bits

$b = 32, r = 8, d = 4\,(cifre)$

$[?][?](\frac{b}{r} * (n+k))$

$$[?][?](\frac{b}{r} * (n+2^r)) = [?][?](\frac{b}{r} * n + \frac{b}{r} * 2^r)$$

Cerco di minimizzare la complessità ponendo r grande, $\frac{b}{r} * n$ risulta diminuito, ma $\frac{b}{r} * 2^r$ è cresciuto esponenzialmente. Scelgo r piccolo, altrimenti 2^r domina su n.

Scegliamo r essere il massimo valore tale che n risulti essere $n \geq 2^r$, quindi $r = log\, n$.

Sostituendo:

$[?][?](\frac{b}{log\, n} * (n+2^{log\, n}))$, essendo $2^{log\, n} = n$

abbiamo $[?][?](\frac{b}{log\, n} * n)$

I numeri variano nell'intervallo $[0, \dots, 2^b - 1]$. Se fisso $b = c * log\, n$, allora l'intervallo diventa $[0, \dots, n^c - 1]$.

Allora il tempo è uguale a $[?][?](c * n)$, se c è costante, allora il tempo è $[?][?](n)$.

Ho ampliato la grandezza dell'intervallo su cui posso applicare l'algoritmo.

Tabelle hash

[CLRS] pp. 209-211

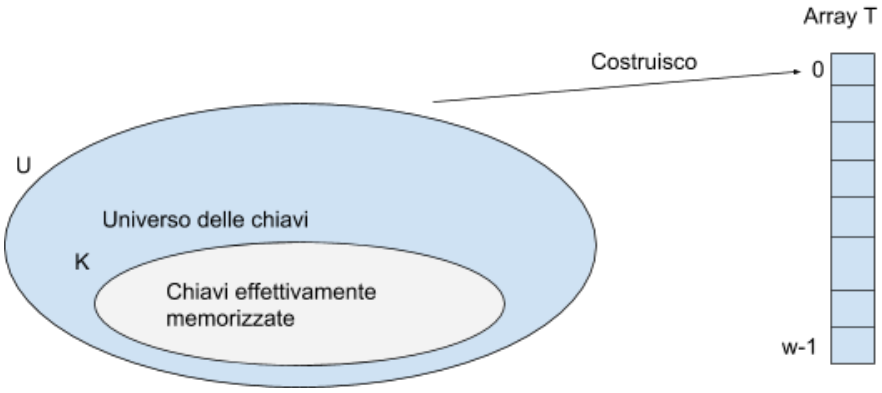
Le tabelle hash sono una possibile implementazione dei dizionari, insiemi dinamici con inserimento, cancellazione e ricerca dove ogni elemento è associato ad una

chiave.

I dizionari possono essere implementati nei seguenti modi:

- Con le liste: $O(n)$
- Con gli alberi binari di ricerca
- non bilanciati $O(n)$
- bilanciati $O(log\ n)$
- Tabelle hash
- Tempo medio delle tabelle hash $[?][?](1)$ Motivo principale per l'utilizzo
- Caso peggiore $[?][?](n)$

Vogliamo fornire un'applicazione che ha bisogno di un insieme dinamico (insert/delete/search). Ogni elemento ha una chiave estratta da un universo U con $|U| = \{0,\dots, w-1\}$, dove w non è troppo grande. Nessun elemento ha la stessa chiave (elementi distinti).



Mi costruisco una tabella ad accesso diretto. Si può utilizzare un array $[0,\dots, w-1]$ dove

- ogni posizione (o cella) corrisponde ad una chiave di U
- Se c'è un elemento x con chiave k, allora nella posizione $T(k)$ è contenuto un puntatore a x
- Altrimenti, se l'insieme non contiene l'elemento, $T(k) = NULL$

$T(k) = x$ se $x.key = k$ e $x \in K$

$T(k) = NULL$ altrimenti

Direct_Access_Search : $O(1)$

```
Direct_Access_Search(Array T, Chiave K)
    return T[K]
```

Direct_Access_Insert : $O(1)$

```
Direct_Access_Insert(Array T, Elem X)
    T[X.key] = X
```

Direct_Access_Delete : $O(1)$

```
Direct_Access_Delete(Array T, Elem X)
    T[X.key] = NULL
```

Potrebbe essere uno spreco se $|K| \ll |U|$ (Molto minore)

Pregi:

Operazioni eseguibili in tempo costante

Difetti:

Lo spazio utilizzato è proporzionale a w e non al numero n di elementi. Di conseguenza si può avere un importante spreco di memoria.

Per ovviare al problema dello spreco di memoria si utilizzano le tabelle hash.

Richieste:

1. Vogliamo ridurre lo spazio per la tabella ad $\Theta(|K|)$, ovvero il numero di chiavi effettivamente utilizzate
2. Le operazioni siano con costo medio $\Theta(1)$, ma non nel caso pessimo

Idea:

Invece di memorizzazione un elemento con chiave o nella cella k, si usa una funzione h, detta funzione hash, e si memorizza l'elemento nella cella $h(k)$

$h: U \rightarrow \{0,1,\ldots, m-1\}$, dove m della tabella hash è generalmente molto più piccola della dimensione dello spazio di tutte le chiavi.

Problema:

Se $m < |U|$, due valori possono essere mappati nella stessa cella, ciò è chiamato collisione.

Le tabelle hash possono soffrire del problema delle collisioni: quando un elemento da inserire è mappato, tramite h, in una cella già occupata, si verifica una collisione.

$$k_1 \neq k_2 \in U, h(k_1) = h(k_2)$$

Se $|K| > m$, ho la certezza di imbattermi in collisioni.

Cerchiamo quindi strategie per gestire le collisioni, ne analizzeremo due:

1. Metodo di concatenamento (liste di collisione (o di trabocco))
2. Indirizzamento aperto

Risoluzione delle collisioni tramite metodo di concatenamento

Si mettono tutti gli elementi che sono associati alla stessa cella in una lista concatenata.

La cella j contiene un puntatore alla testa di una lista di tutti gli elementi memorizzati che sono mappati in j. Se non ci sono elementi la cella j conterrà $NULL$.

Implementazioni

Andiamo ad analizzare l'hashing con concatenamento.

```
Chained_Hash_Insert( Array A, Elem x)
    //inserisci x in testa alla lista T[h(x.key)]
```

Tempo di esecuzione: $\Theta(1)$ se h impiega tempo costante $\Theta(1)$ e l'elemento non è presente nella lista

Chained_Hash_Search(Array A, Key k)
//ricerca un elemento con chiave k nella lista T[h(k)]

Tempo di esecuzione:

Caso peggiore:

Proporzionale alla lunghezza della lista nella cella $h(x)$

Chained_Hash_Delete(Array T, Elem x)
//cancella x dalla lista T[h(k)]

Tempo di esecuzione:

Caso peggiore:

$\Theta(1)$ se la lista doppiamente concatenata (ci serve un puntatore al predecessore)

Senza una lista doppiamente concatenata servirebbe ricercare la chiave del predecessore come ulteriore step.

Sia T una tabella hash con n celle dove sono stati memorizzati n elementi.

Caso peggiore:

Tutti gli elementi sono mappati nella stessa cella. Abbiamo quindi un'unica lista di lunghezza n.

Tempo di esecuzione della ricerca: $\Theta(n)$

Caso medio:

Analisi di h

Deve soddisfare la proprietà di Hashing Uniforme Semplice:

Ogni elemento ha la stessa probabilità di essere mandato in una qualsiasi delle m celle, indipendentemente dalle celle in cui sono mancati gli altri elementi.

$$\forall\, i \in \{0, ..., n-1\}, Q(i) = \frac{1}{n}$$

Assumo che h soddisfi la proprietà di Hashing Uniforme Semplice: indico con n_j la lunghezza della lista $T[j]$

Il valore medio di n_j è $\alpha = \frac{n_0 + n_1 + ... + n_{m-1}}{m} = \frac{n}{m}$

Fattore di carico:

In una tabella hash con n chiavi ed m celle, il fattore di carico è $\alpha = \frac{n}{m}$. Esso è il numero medio di elementi memorizzati in ogni lista.

Teorema 1

In una tabella hash in cui le collisioni sono risolte con il concatenamento, una ricerca senza successo richiede, nel caso medio, un tempo $\Theta(1+\alpha)$, nell'ipotesi di Hashing Uniforme Semplice.

Intuizione:

- Calcolo $i = h(k)$: $\Theta(1)$
- Accedo a $T[j]$: $\Theta(1)$
- Scorro la lista $T[j]$ (fino alla fine) : $\Theta(\alpha)$ in media

Teorema 2

In una tabella hash in cui le collisioni sono risolte con il concatenamento, una ricerca con successo richiede, nel caso medio, un tempo $\Theta(1+\frac{\alpha}{2}) = \Theta(1+\alpha)$, nell'ipotesi di Hashing Uniforme Semplice.

Se il numero di celle della tabella hash è almeno proporzionale al numero di elementi da memorizzare, cioè abbiamo $n = O(m)$, di conseguenza $\alpha = \frac{n}{m} = \frac{O(m)}{m} = \Theta(1)$. (Se alpha è costante)

Tutte le operazioni quindi, mediamente, sono svolte in tempo $\Theta(1)$.

Come si costruiscono le funzioni hash?

Significato di hash: polpetta, tritare.

Una buona funzione hash dovrebbe essere in grado di distribuire in modo uniforme le chiavi nello spazio degli indici della tabella. Deve quindi rispettare l'ipotesi di HUS.

Esempio: la distribuzione delle chiavi è nota.

Le chiavi sono numeri reali k casuali e distribuiti in modo indipendente e uniforme nell'intervallo $0 \leq k < 1$.

Per distribuirle su m celle possiamo moltiplicarle per il valore di quest'ultimo.

$$h(k) = PII(k * m)$$

Questa funzione soddisfa l'ipotesi HUS.

Le distribuzioni delle chiavi difficilmente risultano note a priori.

Metodi di costruzione delle funzioni hash.

1. Metodo della divisione

$$h(k) = k \, mod \, m$$

Esempio:

$$h = 10, k = 91, h(k) = 91 \, mod \, 19 = 15$$

Svantaggio:

Scegliere il valore di m diventa critico.

Vantaggio:

Semplice da implementare

Come scegliere m:

Si evitano le potenze di due per non ottenere sempre e solo i p bit meno significativi ed evitare problemi (carattere uguale di fine stringa). Scegliamo un numero primo non troppo vicino ad una potenza esatta di 2 o 10.

Es: 3 collisioni accettabili, $n = 2000, n/3 = 666, perci[?][?] \, m = 701$

2. Metodo della moltiplicazione

Se $x \in [0 \ldots 1]$ uniformemente distribuiti

$$h(x) = PII(m * x)$$

Data una chiave naturale, la trasformo in un numero nell'intervallo $[0 \ldots 1]$ per poi moltiplicarlo per m.

Fisso cuna costante $A, 0 \leq A \leq 1$

Calcolo $k * A$ ed estraggo la parte frazionaria

$$k * A \, mod 1 = k * A - PII(k * A) P$$
$$h(x) = PII\,(m * (K * A)\, mod 1)$$

Vantaggio:

Il valore di m non è più critico. Funziona bene con tutti i valori di A. Knuth suggerì il valore $\frac{\sqrt{5}-1}{2}$.

Per semplificare i conti possiamo scegliere $m = 2^p$

$A = \frac{q}{2w}, w =$ lunghezza di una parola in memoria, $0<q<2^w$ intero.

Ipotesi:

k entra in una sola parola: $k * a = mod 1 = \frac{k*q}{2^w} mod 1$

$h(k) = p$ bit più significativi della parola meno significativa di $k * q$.

3. Hashing universale

Se un avversario conosce la funzione hash, qualunque essa sia, potrebbe inserire nella tabella elementi che finiscono tutti nella stessa cella e ciò porterebbe a pessime prestazioni.

La soluzione è costruire un insieme di funzioni hash da pescare casualmente.

1. Costruisco un insieme h di funzioni hash, l'insieme deve essere opportunamente costruito.
2. Il programma sceglie casualmente h dall' insieme

Risoluzione delle collisioni tramite indirizzamento aperto

Ipotesi:

Non ho alcuna struttura ausiliaria esterna.

Idea:

Gli elementi sono tutti memorizzati nella tabella.

Non c'è memoria esterna

Ogni cella contiene un elemento dell'insieme dinamico oppure NULL

Per cercare un elemento di chiave k:

1. Calcoliamo h ed esaminiamo la cella con indice (ispezione)
2. Se la cella contiene la chiave k, la ricerca ha successo.
3. Se invece contiene NULL, la ricerca termina senza successo.
4. Se la cella contiene una chiave che non è k calcoliamo l'indice di un'altra cella in base a k e all'ordine di ispezione. Si continua la scansione della tabella finché non si trova k (successo), una cella contenente NULL oppure dopo m ispezioni senza successo.

La funzione hash per l'indirizzamento aperto è la seguente:

$$h : U \; x \; \{0,1,...,m-1\} \rightarrow \{0,1,...,m-1\}$$

$h(k,i)$ rappresenta la posizione della chiave k dopo i ispezioni fallite

per ogni chiave, la sequenza di ispezioni data da $<h(k,0),h(k,1),h(k,2),..., h(k,m-1)>$ deve essere una permutazione di $<0,1,...,m-1>$ in modo che ogni posizione della tabella hash possa essere considerata come possibile cella in cui inserire una nuova chiave.

Assunzioni:

gli elementi della tabella hash sono senza dati satellite

Hash insert: restituisce l'indice della cella dove ha memorizzato la chiave k, oppure segnala un errore se la tabella è piena

```
Hash_Insert(Array T, Elem k)
      i = 0 //ispezioni
      trovata  =  false //inserita
      repeat
            j = h(k,i)
            if T[j] == NULL or T[j] == "deleted"
[q]            T[j] == k
               trovata = true
        else
               i++
    until trovata or i == m
    if trovata
        return j
    else
        return error "overflow, tabella piena"
```

Hash search: restituisce j se $T[j]$ contiene la chiave k oppure $NULL$ se essa non esiste in T.

```
Hash_Search(Array T, Key K)
        i=0
        trovata = false
        repeat
                j=h(k,i)
                if T[j] == k
                        trovata = true
                else
                        i++
        until trovata or i==m or T[j] == NULL
        if trovata
                return j
        else
                return NULL
```

Problema:

La cancellazione da una tabella hash con indirizzamento aperto è problematica.

Soluzione:

Non si può cancellare ponendo NULL al posto della chiave. Usiamo quindi un marcatore (un valore speciale, chiamato "deleted") al posto di NULL per marcare una cella come vuota a causa di un'eliminazione.

Svantaggio:

Il tempo di ricerca non dipende più dal fattore di carico $\frac{m}{n}$
Non si usa l'indirizzamento aperto quando le chiavi vanno cancellate. Si utilizza invece il concatenamento.

Hash_Delete(Array T, Key K)[r]

La posizione viene determinata dalla funzione $h: U x U \rightarrow \{0,1,\dots,m-1\}$ che restituisce permutazioni

$<h(k,0),h(k,1),\dots,h(k,m-1)>$ permutazioni di $\{0,1,\dots,m-1\}$.

Le possibili permutazioni sono $m!$ ma ottenere un sufficiente numero di permutazioni distinte non è banale.

Estensione dell'hashing uniforme semplice:

Situazione ideale : h deve rispettare la proprietà di hashing uniforme, ovvero ogni chiave deve avere la stessa rpobavilità di avere la stessa probabilità di avere come senzienza di ispezione, una delle $n!$permutazioni di $\{0,1,\dots,m-1\}$

Per far ciò:

$h(k,0)$deve distribuire in modo uniforme nelle m celle.

$h(k,1)$deve distribuire in modo uniforme nelle $m-1$ celle. (Nel caso la cella fosse occupata)

...

$h(k,m-1)$deve distribuire in modo uniforme nelle $m-1$ celle.

Ovvero, h deve rispettare la proprietà di Hashing Uniforme Semplice per ogni ispezione (o "iterazione")

Analizziamo quindi tre metodi di scansione

1. Ispezione (o "scansione") lineare
2. Ispezione (o "scansione") quadratica
3. Hashing doppio

1. Ispezione (o "scansione") lineare

Data una funzione hash ordinaria $h':U \to \{0,1,\dots,m-1\}$chiamata funzione ausiliaria, il metodo dell'ispezione lineare usa la seguente funzione:

$$h(k,i) = (h'(k) + i) \bmod m$$

con $i \in \{0,1,\dots,m-1\}$

Nota: la prima cella ispezionata determina l'intera sequenza di ispezioni, quindi ci sono soltanto m sequenze di ispezioni distinte.

Vantaggi:

Facilità di calcolo

Svantaggi:

Dopo i celle occupate la proprietà che venga estratta la cella immediatamente successiva è $\frac{i+1}{m}$. Abbiamo quindi un problema di addensamento o aglomerazione primaria. Si possono formare lunghe file di celle occupate che aumentano il tempo di ricerca.

Per superare il limite delle m ispezioni distinte e dell'addensamento, proviamo a cambiare il passo con una funzione quadratica.

2. Ispezione (o "scansione") quadratica

Utilizziamo una funzione di hashing quadratica.

$$h(k,i) = (h'(k) + C_1 * i + C_2 * i^2) \, mod \, m$$

con C_1, C_2 costanti non nulle ed $i \in \{0,1,\dots,m-1\}$

La scansione quadratica funziona meglio[s] ma particolare attenzione va posta nella ricerca dei valori di C_1, C_2 in modo che vengano generati tutti gli indici. Non possono essere scegli in modo arbitrario.

Esempio: $C_1 = C_2 = \frac{1}{2}$, $m = 2^p$

Ho un massimo di m sequenze di ispezione distinte.

Svantaggio: "Addensamento secondario"

Se due chiavi distinte $k_1 \neq k_2$ hanno valore hash ausiliario $h'(k_1) = h'(k_2)$ allora hanno la stessa sequenza di ispezione.

Nonostante tutto, continuo ad avere lo stesso passo tra un'iterazione e la successiva. Procedo quindo con una seconda funzione hash.

3. Hashing doppio

$$h(k,i) = (\, h_1(k) + i * h_2(k) \,) \, mod \, m$$

Con h_1, h_2 funzioni hash ausiliarie e $i \in \{0,1,\dots,m-1\}$

Vantaggio:

La posizione finale viene data dai valori combinati della coppia $(h_1(k), h_2(k))$, i cui elementi producono m combinazioni distinte ciascuno.

Ho quindi $\theta(m^2)$ sequenze di ispezione perchè ogni possibile coppia $(h_1(k), h_2(k))$ produce una sequenza distinta di ispezione.

Vogliamo porre $h_2(k)$ ed m (dimensione della tabella hash) coprimi (relativamente primi). Ciò mi assicura che l'intera tabella venga ispezionata.

Esercizio:

Date due ispezioni $i,i' < m, h(k,i) = h(k,i')$ con $h_2(k),m$ coprimi, allora dimostrare che $i = i'$

Dimostrazione A:

Scelgo $m = 2^p$ potenza di due e definisco $h_2(k)$ in modo che produca sempre un numero dispari: $h_2(k) = 2 * h_1(k) + 1$

Dimostrazione B:

Scelgo m primo, definisco $h_2(k)$ in modo che generi sempre un numero intero positivo minore di m: $h_1(k) = k \bmod m, h_2(k) = 1 + (k \bmod m'), m' < m$

Esempio:

$$m = 13$$
$$h_1(k) = k \bmod 13$$
$$h_2(k) = 1 + (k \bmod 11)$$
$$h(k,i) = (h_1(k) + i * h_2(k)) \bmod m$$

Input: $<79,50,69,72,98,14>$

$$h(79,0) = 1, h(50,0) = 11, h(69,0) = 4, h(72,0) = 7$$
$$h(98,0) = 7$$

Collisione: $h(98,1) = 5$

$h(14,0) = 5$

Collisione: $h(14,1) = 9$

Analisi dell'hashing a indirizzamento aperto

Teorema:

Nell'ipotesi di

- Tabella hash priva di cancellazioni
- Funzione hash che rispetta l'ipotesi di hashing uniforme
- $\alpha = \frac{m}{n}$ fattore di carico. Essendo $n \leq m, 0 \leq \alpha \leq 1$

il numero atteso (medio) di ispezioni in una ricerca senza successo è al massimo $\frac{1}{1-\alpha}$.

Dimostrazione:

Essendo $\alpha \leq 1$ per ipotesi, sono presenti delle celle vuote.

La prima scansione avviene con probabilità 1.

La seconda scansione avviene con probabilità $\frac{n}{m} = \alpha$

La terza scansione avviene con probabilità $\frac{n}{m} * \frac{n-1}{m-1} \simeq \alpha^2$ [t]

$$1 + \alpha + \alpha^2 + ... \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-}$$

Il valore atteso (medio) del numero di ispezioni sarà quindi

con $\alpha^i < 1$

Se α è costante, una ricerca senza sucesso viene eseguita in tempo medio $\theta(1)$ e in caso pessimo $O(n)$

Analisi del valore di α:

Se $\alpha = 0.5$ (tabella riempita a metà), allora il numero medio di ispezioni è

$\frac{1}{1-0.5} = 2$.

Se $\alpha = 0.9$ (tabella riempita al 90%), allora il numero medio di ispezioni è $\frac{1}{1-0.9} = 10$.

Corollario:

L'inserimento di un elemento in una tabella hash a indirizzamento aperto, con fattore di carico α, richiede in media non più di $\frac{1}{1-\alpha}$ispezioni (tempo richiesto da una ricerca senza successo) nell'ipotesi di Hashing Uniforme.

Nota: l'elemento viene inserito solamente se c'è almeno una cella vuota, quindi $\alpha<1$

L'inserimento richiede una ricerca senza successo, seguita dalla sistemazione della chiave nella prima cella vuota. Quindi, dal teorema, il numero massimo di ispezioni sarà al massimo $\frac{1}{1-\alpha}$.

Teorema:

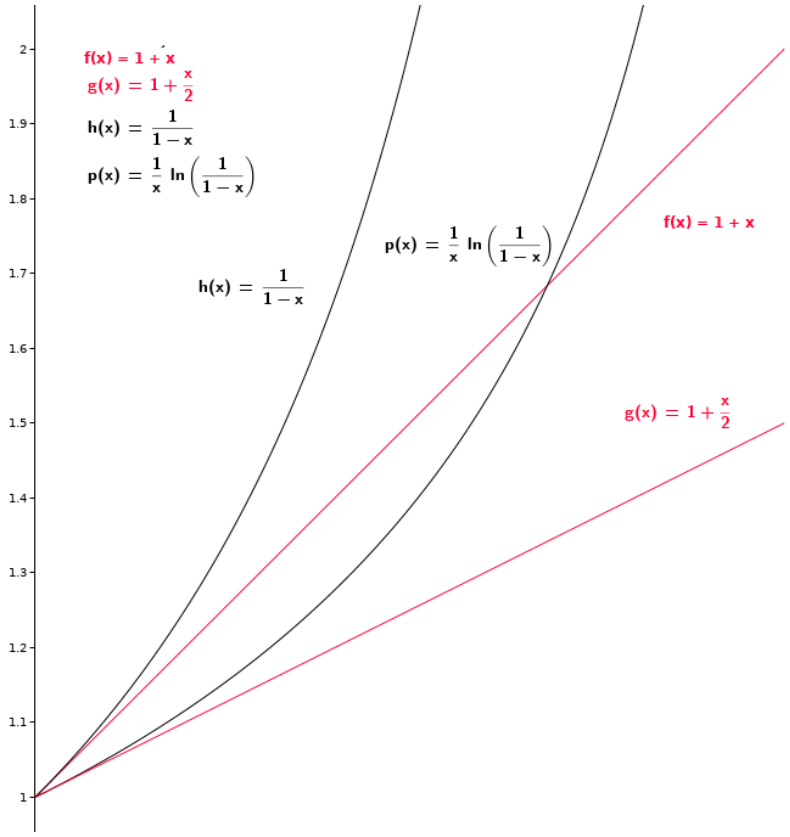
Data una tabella hash ad indirizzamento aperto con $\alpha \leq 1$ e funzione hash che rispetti l'ipotesi di Hash Uniforme, in una ricerca con successo in una tabella le cui chiavi hanno uguale probabilità di essere scelte, il numero atteso di ispezioni è al massimo $\frac{1}{\alpha} * log(\frac{1}{1-\alpha})$.

Analisi del valore di α:

Se $\alpha = 0.5$ (tabella riempita a metà), allora il numero massimo di ispezioni è 1.387., ovvero con meno di 2 accessi riesco a trovare l'elemento cercato.

Se $\alpha = 0.9$ (tabella riempita al 90%), allora il numero massimo di ispezioni è 2.255, ovvero con meno di 3 accessi riesco a trovare l'elemento cercato.

Confrontro tra metodi di risoluzione delle collisioni



Notiamo che i costi del concatenamento in nero (indirizzamento aperto) crescono molto più velocemente delle altre due rosse (liste di collisione).

Mod 2 - Grafi

Coppia ordinata $G = (V, E)$ formata da due insiemi V (vertici) ed E (archi)

$V = \{1,2,3,...,n\}$

$E \subseteq V x V$, ovvero E è un sottoinsieme dell'insieme delle parti (prodotto cartesiano) dell'insieme V

I grafi possono essere

- Orientati
- Non orientati , se le relazioni sono simmetriche

se $\forall (u,v) \in E <=> (v,u) \in E$

Non si ammettono cappi nei grafi non orientati.

$\forall u \in E <=> (u,u) \notin E$

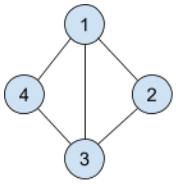
Sottografi

Sottografo di G

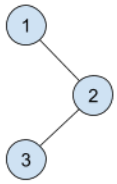
$G' = (V', E')$ è sottografo di G se $V' \subseteq V$ e $E' \subseteq E$

Sottografo indotto

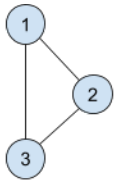
Dato $V' \subseteq V$, il sottografo indotto da V' di G è $G[V'] = (V', E')$ con $E' = E \cap VxV'$



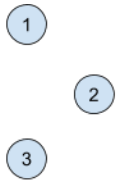
Grafo G



Sottografo di G



Sottografo indotto da {1,2,3} di G



Sottografo di G

Cammini

Un cammino di G è una sequenza $<X_0, X_1, X_2, \dots, X_k>$ dove i vertici appartengono al grafo di partenza.

$\forall i, 0 \leq i < k, (X_i, X_{i+1}) \in E$

Cammini semplici e cammini non semplici

$< 1, 2, 3, 4, 5 >$ è un cammino semplice
$< 1, 2, 3, 1, 4, 5 >$ non è un cammino semplice

Un cammino è semplice se tutti i vertici sono distinti

Lunghezza di un cammino

La lunghezza di un cammino è il numero dei suoi archi

Raggiungibilità dei vertici

u è raggiungibile da v se esiste un cammino $< X_0, X_1, X_2, ..., X_k >$ tale che $X_0 = u$ e $X_k = v$

Ciclo

Un ciclo è un cammino dove $X_0 = X_k$

Un grafo senza cicli si dice aciclico.

[NO] Grafo connesso

Un grado non orientato si dice connesso se $\forall (u,v) \mid u,v \in V$, esiste un cammino da u a v con $u \neq v$

[NO] Componente connessa

Dato $G = (V, E)$ e $V' \subseteq V$

V' si dirà componente connessa se

1. $G[V']$ è connesso
2. V' non è sottoinsieme stretto di un sottografo connesso

Se il numero di componenti connesse di un grafo è uguale a 1, il grafo è connesso.

[NO] Vertici adiacenti

Dato $G = (V, E)$ e $u \in V$

Il grado di $u = deg(u)$ è il numero di vertici adiacenti a u

Arco incidente

Un arco che è collegato ad v

Vertici isolati e terminali

Se $deg(u) = 0$ allora u è isolato

Se $deg(u) = 1$ allora u è terminale

Teorema della stretta di mano (HandShaking Lemma)

[NO]

$\sum_{u \in V} degree(u) = 2 * m$, dove m è il numero di archi, $m = |E|$

[O]

$outDegree(u) =$ numero degli archi uscenti da u

$inDegree(u) =$ numero degli archi entranti in u

$$\sum_{u \in V} outDegree(u) = \sum_{u \in V} inDegree(u) = m$$

[NO] Proprietà

Il numero di vertici che hanno grado dispari è sempre pari

Dim:

$$V = P \cup D$$
$$P = \{u \in V \mid deg(u)\,[?][?]\,pari\}$$
$$D = \{u \in V \mid deg(u)\,[?][?]\,dispari\}$$
$$2m = \sum_{u \in V} deg(u)$$

$$= \sum_{u \in P} deg(u) + \sum_{u \in D} deg(u)$$

$$= \sum_{u \in P} (2 * h(u)) + \sum_{u \in D} (2 * h(u) + 1)$$

$$= \sum_{u \in P} (2 * h(u)) + \sum_{u \in D} (2 * h(u)) + |D|$$

$$|D| = 2m - 2 \sum_{u \in P} h(u) - 2 \sum_{u \in D} h(u)$$

$$= 2 * (m - \sum_{u \in V} h(u))$$

Il numero di vertici con grado dispari è quindi pari

Esercizio:

Dimostrare che, dato $G = (V, E)$ non orientato senza vertici isolati (nessuno vertice ha grado 0),

con $|E| = |V| - 1$

Allora, esistono almeno due vertici terminali

Dimostrazione per assurdo:

$n = |V|, m = |E|$

Per il lemma della stretta di mano:

$$\sum_{u \in V} deg(u) = 2|E| = 2m$$

$$2n - 2 = 2m = \sum_{u \in V} deg(u)$$

Chiamo V_1 un sottoinsieme di V di vertici terminali

$$V_1 = \{u \in V \mid deg(u) = 1\}$$

Il problema diventa: dimostrare che $|V_1| \geq 2$

$$2m = \sum_{u \in V_1} deg(u) + \sum_{u \in V \setminus V_1} deg(u)$$

$$2n - 2 \geq 2n - |V_1|$$????[u]

$$|V_1| \geq 2$$ OK

Matrice di adiacenza

Dato $G = (V, E)$

$n = |V|, m = |E|$

A Matrice quadrata nxn

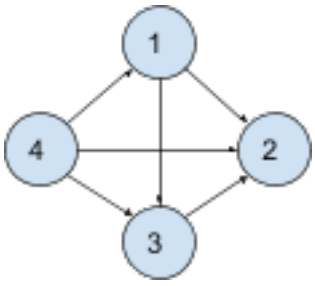
- Per trovare il grado di un vertice basta calcolare la somma degli elementi sulla corrispondente riga
- [NO]n archi in totale = doppia somma di tutti gli elementi della matrice
- [O] n archi in totale = doppia somma di tutti gli elementi della matrice

Usare le matrici di adiacenza solitamente conviene se ci sono molti archi, altrimenti è meglio utilizzare le liste.

Matrice di adiacenza per grafi orientati

$$A_{i,j} = 0 \, se \, (i,j) \notin E$$

$$A_{i,j} = 1 \, se \, (i,j) \in E$$



1

2

3

4

1

0

1

1

0

2

0

0

0

0

3

0

1

0

0

4

1

1

1

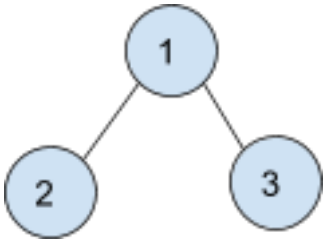
0

Matrice di adiacenza per grafi non orientati

$$A_{i,j} = 0 \, se \, \{i,j\} \notin E$$

$$A_{i,j} = 1 \, se \, \{i,j\} \in E$$

Essendo la relazione tra vertici simmetrica, $A = A^T\,(trasposta)$



1

2

3

1

0

1

1

2

1

0

0

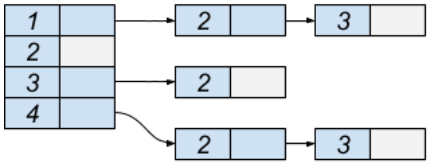
3

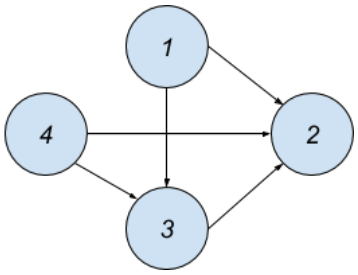
1

0

0

Liste concatenate





Usare le liste di adiacenza solitamente conviene se ci sono pochi archi, altrimenti è meglio utilizzare le matrici.

[O] Densità

$$\delta = \frac{numero\,di\,archi}{numero\,di\,possibili\,archi = n^2}$$

$n = |V|, m = |E|$

$$\delta = \frac{m}{n^2}$$

[NO] Densità

$$\delta = \frac{numero\,di\,archi}{numero\,di\,possibili\,archi}$$

$$\delta = \frac{m}{k}, k = \frac{n(n-1)}{2}$$ coefficiente binomiale

Grafi sparsi: $n \simeq m \rightarrow$ lista

Grafi densi: $n^2 \simeq m \rightarrow$ matrice

Proprietà:

Sia G un grafo NO. Se G è aciclico, allora la cardinalità $|E| \leq |V| - 1$

Dimostrazione induttiva su $n = |V|$:

Caso base:

se $n = 1$allora$m = 0$

se $n = 2$allora$m \leq 1$

Passo induttivo $n \geq 3$:

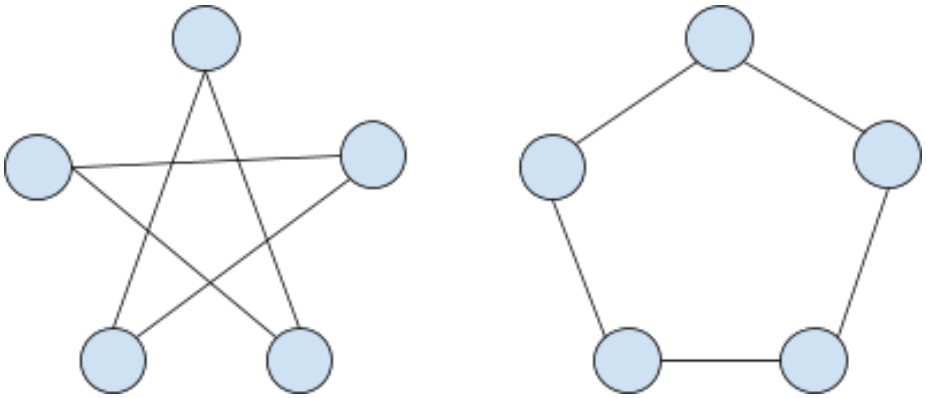
[NO] Complemento di un grafo

Il complemento di un grafo è un grafo con gli stessi vertici, i cui archi sono complementari:

$G = (V, E), \overline{G} = (\overline{V}, \overline{E})$

Grafo autocomplementare

Un grafo è detto autocomplementare se $\overline{G} = G$, ovvero se $\forall\,(i,j) \in E\; sse\; (i,j) \in \overline{E}$



Prodotto tra matrici di adiacenza

Prodotto di matrici

Due matrici nxm ed p*n (il numero di righe della prima equivale al numero di colonne dela seconda) possono essere moltiplicate tramite il prodotto "righe per colonne".

$$C = A * B$$

$$C_{i,j} = \sum_{k=1}^{n} a_{i,k} * b_{k,j}$$

Matrice al quadrato

Moltiplicando una matrice di adiacenza A per se stessa, tramite il prodotto tra righe e colonne, ottengo una matrice delle stesse dimensioni con le seguenti caratteristiche:

- Sulla diagonale principale ho i gradi dei vertici
- Nelle altre posizioni ho il numero di cammini tra i e j di lunghezza 2

$$A^2 = A * A = (a^{(2)}_{i,j})$$

$$a^{(2)}_{i,j} = \sum_{k=1}^{n} a_{i,k} * a_{k,j}$$

Dimostrazione:

"Sulla diagonale principale ho i gradi dei vertici" ($i = j$)

$$a^{(2)}_{i,i} = \sum_{k=1}^{n} a_{i,k} * a_{k,i}$$

Essendo G non orientato, la matrice è simmetrica e $a_{i,k} * a_{k,i} = a^2_{i,k}$

$$a^{(2)}_{i,i} = \sum_{k=1}^{n} a^2_{i,k}$$

, notiamo che il termine della sommatoria contiene solo valori binari che elevati al quadrato restituiscono il medesimo valore. Perciò

$$a^{(2)}_{i,i} = \sum_{k=1}^{n} a^2_{i,k} = \sum_{k=1}^{n} a_{i,k} = deg(i)$$

"Nelle altre posizioni ho il numero di cammini tra i e j di lunghezza" ($i \neq j$)

$$a^2_{i,j} = \sum_{k=1}^{n} a_{i,k} * a_{k,j}$$

, essendo la matrice di adiacenza composta da valori binari, il prodotto risulta valere 1 solo se entrambi i fattori valgono 1, ovvero se esiste un cammino di lunghezza 2

Matrice con esponente maggiore di 2

$A^n, n>2$ conterrà

- Sulla diagonale: il numero di cicli di lunghezza n che partono da i
- Fuori dalla diagonale: il numero di cammini di lunghezza n

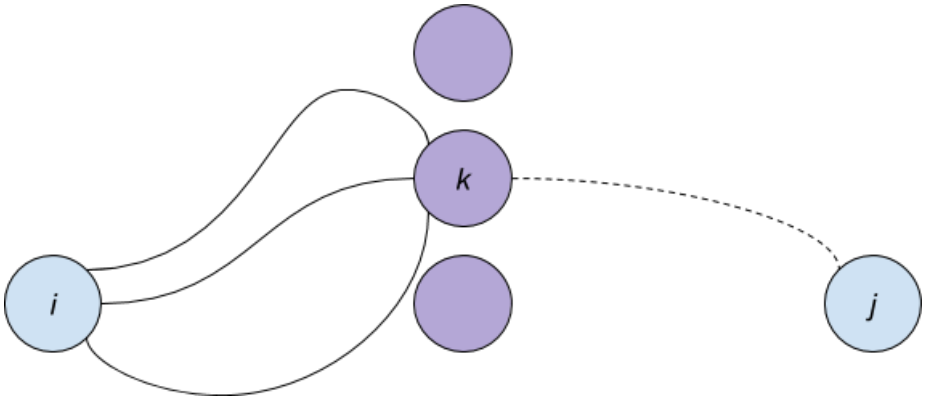
Dimostrazione per induzione:[v][w]

Ipotesi:

Passo induttivo:

$$A^n = A*A*\_* A\,(n\ volte) = A^{n-1}*A$$

$a^{(n)}_{i,j} = \sum_{k=1}^{n} a^{(n-1)}_{i,k} * a_{k,j}$, ovvero il numero di cammini da i a j passanti per k.

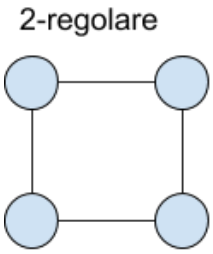


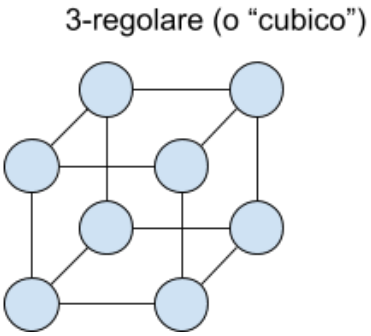
Esercizio

Mostrare che $G = (V, E)[NO]$ contiene un triangolo, ovvero un ciclo di lunghezza 3, se e solo se esistono due indici i e j tali che sia la matrice di adiacenza A e la matrice al quadrato A^2 hanno un elemento non nullo in posizione (i,j)

[NO] Grafi regolari

$G = (V, E)$ si dice k-regolare se tutti i vertici hanno grado k. Con $n = |V|, m = |E|$





Proprietà:

Se g è 2-regolare allora il numero di vertici coincide necessariamente con il numero degli archi. $n = m$

Dimostrazione:

Utilizziamo il teorema della stretta di mano

$$2m = \sum_{u \in V} deg(u)$$ [x][y]

Esercizio:

Dimostrare che se il grafo G è 3-regolare, allora n è pari. Analogamente, analizzare G 4-regolare.

Isomorfismi di grafi

Un isomorfismo di grafi è un funzione che mappa un grafo in un secondo dalla stessa forma.

Viene fornita la definizione per i grafi non orientati, essa può però essere adattata a grafi orientati.

[NO] Definizione:

$G_1 = (V_1, E_1), G_2 = (V_2, E_2)$

$\Phi : V_1 \to V_2$ si dice isomorfismo se valgono le due proprietà:

1. Φ deve essere una funzione biettiva (è necessaria una corrispondenza 1-1).
2. La relazione di adiacenza deve essere preservata
   $\forall u, v \in V_1, (u,v) \in E_1 \Leftrightarrow (\Phi(u), \Phi(v)) \in E_2$

Determinare se due grafi sono isomorfi

$G_1 = (V_1,E_1), G_2 = (V_2,E_2)$ sono isomorfi se esiste un isomorfismo da G_1 a G_2 e si indica con $G_1 \simeq G_2$

Condizioni necessarie perché $G_1 \simeq G_2$:

1. $|V_1| = |V_2|$
2. $|E_1| = |E_2|$
3. Stessa degree sequence $degseq(G_1) = degseq(G_2)$
4. $\#ComponentiConnesse(G_1) = \#ComponentiConnesse(G_2)$

G	$\overline{G}$	Può verificarsi
Connesso	Connesso	FALSO
Connesso	Disconnesso	FALSO (Es: autocomplementare)
Disconnesso	Connesso	VERO
Disconnesso	Disconnesso	FALSO

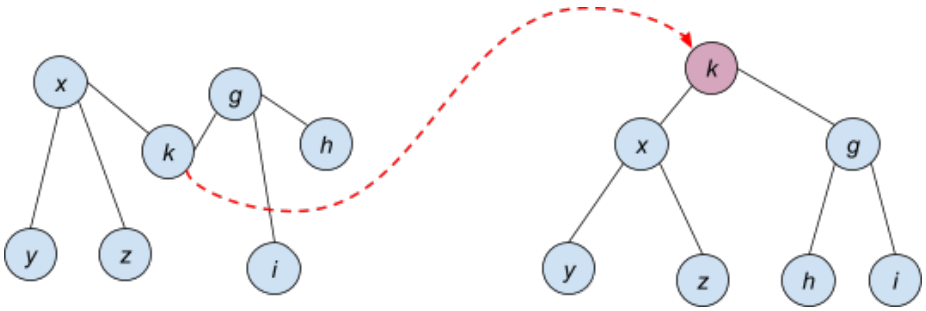
Il problema di determinare l'esistenza di un'isomorfismo tra due grafi è esoso in termini di tempo. Per evitare il bruteforce di tutte le combinazioni si procede a mappare gli archi con grado analogo.

Alberi

Un albero è una particolare categoria di grafo

[NO] Definizione

$G = (V, E)$ si dice albero [libero] se è aciclico e connesso.
Un albero libero in cui si seleziona un particolare vertice, chiamato "radice", si dice radicato.



Proprietà 1:

Se G è un albero, allora $|E| = |V| - 1$ poichè è, per definizione, connesso e aciclico:

- G è connesso se $|E| \geq |V| - 1$: si deve rispettare un numero minimo di archi
- G è aciclico se $|E| \leq |V| - 1$: si deve rispettare un numero massimo di archi

L'albero risulta essere quindi una struttura "fragile" per quanto riguarda il numero di archi.

Proprietà degli alberi [liberi]

Sia $G = (V, E)[NO]$, le seguenti affermazioni sono equivalenti:

1. G è un albero
2. Due vertici qualsiasi di G sono connessi da un unico cammino semplice, ovvero senza vertici ripetuti

Dimostrazione:

Sia $G = (V, E)[NO]$, per assurdo assumo ci siano almeno due vertici u, v, tali che esistono almeno due cammini che li collegano. Facendo ciò si forma un ciclo.

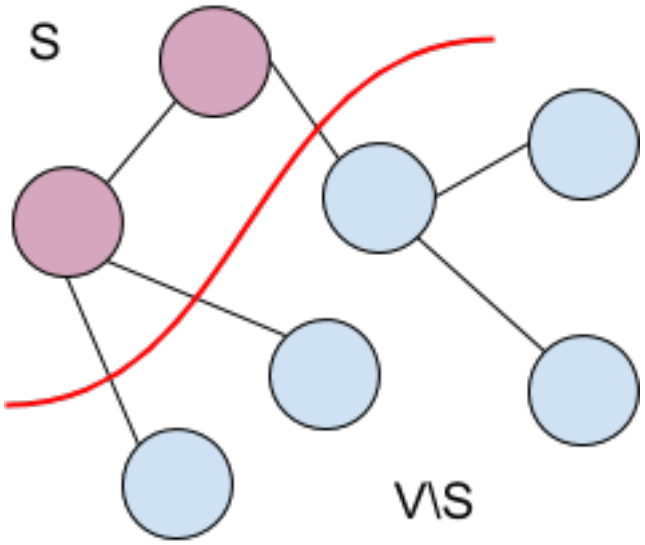
3. G è connesso, ma se un qualche arco viene rimosso da G, il grafo risultante è disconnesso
4. G è connesso e $|E| = |V| - 1$
5. G è aciclico e $|E| = |V| - 1$
6. G è aciclico, ma se un qualunque arco viene aggiunto, allora il grafo risultante è ciclico.

Alberi di copertura

Un albero $G = (V, E)[NO]$ connesso si dice di copertura se tutti gli archi in $T \subseteq E$ toccano tutti i vertici di G.

Taglio di un albero

Un taglio di un albero è una divisione in due parti $(S, V\S)$ dell'insieme dei vertici.



Un taglio rispetta $A \subseteq E$ se non esistono archi di A tagliati. Essi possono essere in una delle due partizioni ma non possono attraversare il taglio.

Arco leggero

Un arco è leggero se ha il peso minore tra gli altri archi attraversanti un taglio.

Albero di copertura minimo (MST)

Sia T un albero di copertura, definisco

$w(T) = \sum_{(u,v)\in T} w(u, v)$

dove $w : E \to \mathbb{R}$ è detta "funzione peso"

Definizione:

Un albero di copertura T si dice minimo o "di peso minimo" (minimum spanning tree) se $w(T)$ è il minimo rispetto a tutti gli alberi di copertura

Teorema fondamentale degli MST:

> Sia $G = (V, E)[NO]$ connesso con funzione peso w.
>
> Se le tre condizioni

1. $A \subseteq E$è contenuto in qualche MST
2. $(S, V \setminus S)$ è un taglio che rispetta$A$
3. Sia (u, v) un arco leggero che attraversa il taglio $(S, V \setminus S)$

sono rispettate, allora l'arco (u, v) è sicuro per A, ovvero $A \subseteq \{(u, v)\}$ è contenuto in qualche MST.

Dimostrazione

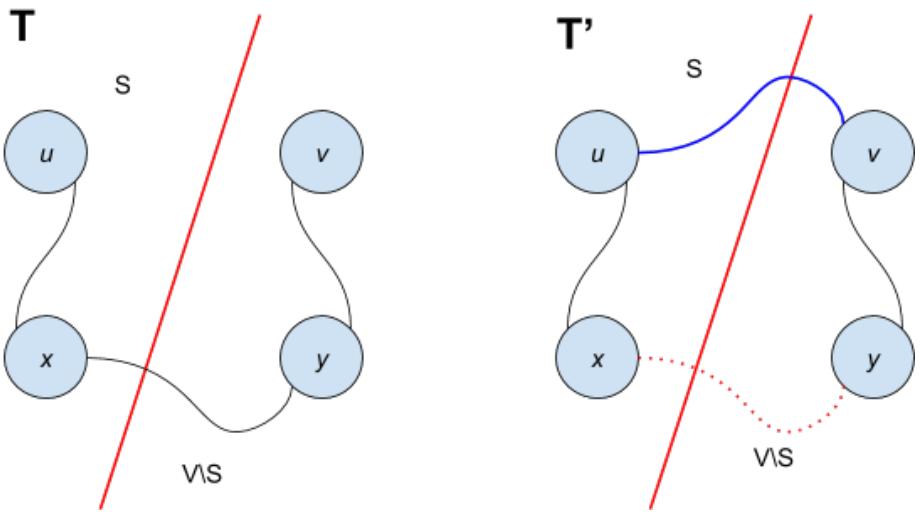
$A \subseteq T$ con T MST

Due casi:

1. $(u, v) \in T$, caso banale in quanto $A \cup \{(u, v)\} \subseteq T$ e T è l'MST che cerchiamo.
2. $(u, v) \notin T$, quindi T non è l'MST che cerchiamo, procediamo quindi a costruirne uno:

Per le sei proprietà fornite, se unisco (u, v) a T formo un ciclo, è quindi necessario rimuovere l'altro cammino (x, y) che lo forma tra gli archi che attraversano il taglio.

$T' = T \cup (u, v) \setminus \{(x, y)\}$



$w(u, v) \leq w(x, y)$: Il peso dell'arco aggiunto è minore del peso dell'arco rimosso, in quanto esso è leggero per ipotesi.

Quanto vale allora $w(T')$?

$w(T') \leq w(T)$ ed essendo $w(u, v) \leq w(x, y)$ allora $w(T') \leq w(T)$

Da $w(T) \leq w(T') \leq w(T)$ [z]risulta che $w(T') = w(T)$, e T' è quindi un MST.

Nota[aa]: se $w(u,v) = w(x,y)$ allora $(u,v) = (x,y)$?

Dobbiamo però dimostrare che T' sia "sicuro", e perciò dimostriamo che $A \cup \{(u,v)\} \subseteq T'$

Corollario

Sia $G = (V,E)[NO]$ connesso con funzione peso w. Se le tre condizioni

1. $A \subseteq E$è contenuto in qualche MST
2. C è una componente connessa della foresta (V,A)
3. Sia (u,v) un arco leggero che collega la componente C con il resto del grafo

sono rispettate, allora l'arco (u,v) è sicuro per A, ovvero $A \cup \{(u,v)\}$ è contenuto in qualche MST.

Dimostrazione

Se impongo che

- Il taglio dell'ipotesi 2 sia $(C, V \setminus C)$, in modo da isolare la componente connessa
- MANCA[ab]

posso utilizzare il teorema fondamentale per la dimostrazione

Corollario

Sia (u,v) un arco di peso minimo in G. Allora (u,v) appartiene a qualche MST.

Dimostrazione tramite la tecnica "cuci e taglia"

Sia T un MST di G.

Due casi:

1. $(u,v) \in T$, ovvio
2. $(u,v) \notin T$

Corollario

Sia (u,v) un arco di peso minimo in G e supponiamo che sia unico. Allora (u,v) appartiene a tutti gli MST[ac].

Dimostrazione per assurdo

Nonostante le ipotesi, esiste [almeno] un MST senza l'arco in questione.

Non esistendo, provvediamo ad aggiungerlo a T. $T' = T \cup \{(u, v)\}$

Facendo ciò creo sicuramente un ciclo, in quanto T è albero. Scelto un arco (x, y) e lo rimuovo. Ho quindi costruito un albero di copertura con peso
$W(T[?][?][?]) = W(T) + w(u,v) - w(x,y) < W(T)$

Con $w(u, v) < w(x, y)$ (minore stretto!)

Contraddizione: essendo T MST, non può esistere un altro albero con peso inferiore.

Esercizio

$G = (V, E)[NO]$ connesso con $w : E \to \mathbb{R}$

Sia T_{min} un MST di G.

Sia T' un albero di copertura (ST) non necessariamente minimo (M).

Siano $(u, v), (x, y)$ rispettivamente gli archi di $T, T[?][?][?]$ di peso massimo. Ordino gli archi e considero i pesi maggiori.

$T_{min}: <e_1, e_2, ..., e_{n-1}>$ con $e_{n-1} = (u,v)$

$T': <e'_1, e'_2, ..., e'_{n-1}>$ con $e'_{n-1} = (x,y)$

Congettura: $w(u, v) \leq w(x, y)$

Si può procedere per confutazione con controesempio oppure per dimostrazione tramite metodo "cuci e taglia".

Soluzione[ad]:

Esercizio

Dimostrare che, se tutti i pesi del grafo sono distinti, allora esiste un solo MST.

Soluzione[ae]:

Generazione degli alberi di copertura minima

```
Generic_MST(G, w)
  A = []
  while (A non forma un MST) o analogamente (|A| ≤ |V| − 1)
    //Trova un arco sicuro per A (u,v)
    A=A U {(u,v)}
  Return A
```

Verranno analizzati gli algoritmi di Kruskal e Prim, i quali differiscono per l'implementazione della ricerca dell'arco sicuro.

Kruskall utilizza le strutture dati Set per la gestione di insiemi disgiunti. Esse dispongono di tre operatori:

1. Make_set(X)
2. Union(x,y) o Merge_set(x,y)
3. Find_set(x)

Esempio di utilizzo: determinare le componenti connesse di un grafo

```
CC(G)
   Foreach u in V[G] do
      Make_set(u)
   Foreach (u,v) in E[G] do
      If find_set(u) != find_set(v) then
            Union(u,v)
```

Generazione di MST : Kruskal

$G = (V, E)$ connesso, $n = |V|, m = |E|$

```
1  Kruskal(G,w)
2       A = []
3       Foreach u in V[G] do                                    O(n)
4            Make_set(u)
5       Ordina gli archi di E[G] in ordine crescente            Θ(mlogm)
6       Foreach (u,v) in E[G] do                                O(mlogm)
7            If find_set(u) != find_set(v) then                 O(logm)*
8             Union(u,v)                                   O(logm)*
9                  A = A U {(u,v)}
10      Return A
```

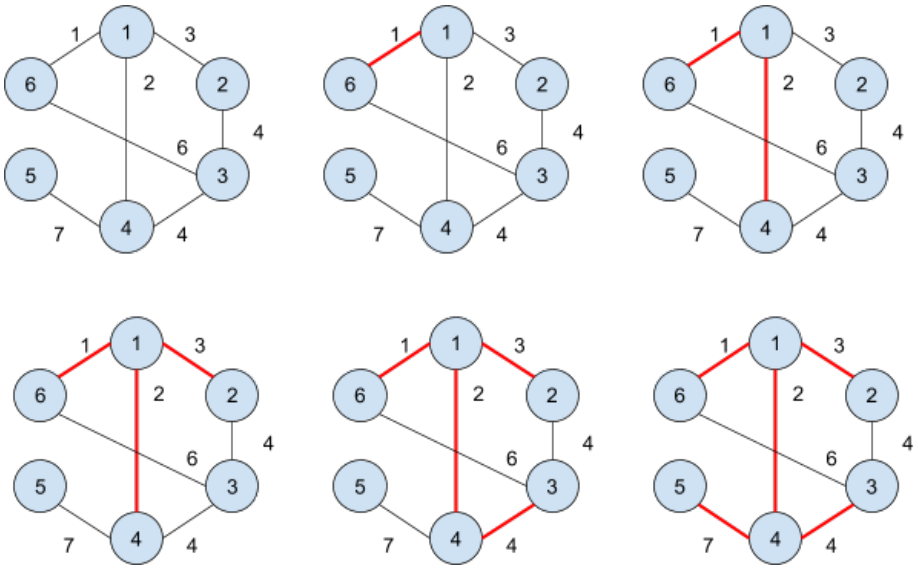
*Per l'analisi delle due find_set e della union ci viene formita la complessità ottenuta tramite Heichmann.
Complessità:

$O(m\,logm + n + m\,logm)$, avendo $m \geq n-1$ archi

poichè l'arco è connesso, la complessità è $O(m\,logm)$

Simulazione di esecuzione

Tramite grafo



Tramite tabella

Passo	A	Insiemi disgiunti
1	{}	{1},{2},{3},{4},{5},{6}
2	{(1,6)}	{1,6},{2},{3},{4},{5}
3	{(1,6),(1,4)}	{1,4,6},{2},{3},{5}
4	{(1,6),(1,4),(1,2)}	{1,2,4,6},{3},{5}
5	{(1,6),(1,4),(1,2),(4,3)}	{1,2,3,4,6},{5}
6	{(1,6),(1,4),(1,2),(4,3),(4,5)}	{1,2,3,4,5,6}

Generazione di MST : Prim

A differenza di Kruskal, Prim richiede un vertice di partenza detto "radice". Con l'insieme Q si indica l'insieme dei vertici da estrarre. V
Q indica quindi l'insieme dei vertici già estratti. Q è una coda di priorità, implementata tramite Heap Binario, i cui elementi hanno le seguenti caratteristiche:

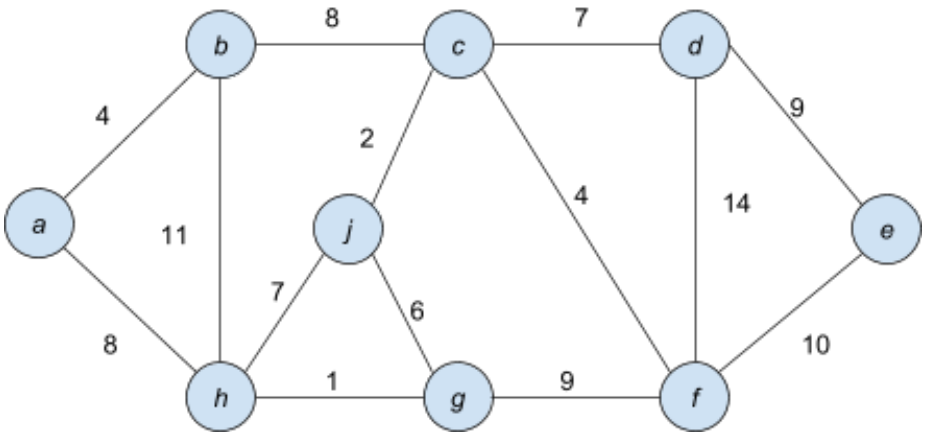
- Predecessore: $\pi[u]$
- Chiave: $Key[u]$ valore dell'arco incidente che attraversa il taglio $(Q, V$ $Q)$ con peso minore. Si utilizza il valore infinito positivo per indicare l'eventuale assenza di archi.

```
Prim(G,w,r):
  Q = V[G]
  for each v in Q
    key[v] = inf
  key[r] = 0
  π[r] = NIL
  while Q != [] do
    u = extractMin(Q)
    for each v in Adj[u]
      if v in Q and w(u,v) < key[v] then
        key[v] = w(u,v)
        π[v] = u
  return T = {(v,π[v]} | v in V \ {r}}
```

La convergenza risulta essere finita

La complessità della funzione Prim è $O(mlogn)$

Simulazione



Cammini minimi

Solitamente si sottointende l'utilizzo su grafi orientati, in caso di grafi non orientati, ogni arco può essere sostituito da una coppia di archi inversamente orientati.

$G = (V, E, w)$ orientato, $w : E \to \mathbb{R}$

Un cammino p è una sequenza $< x_0, x_1, ..., x_q >$, $\forall i, i \leq i \leq q, (x_{i-1}, x_i) \in E$.

$w(p) = \sum_{i=1}^{q} w(x_{i-1}, x_i)$ è la funzione "peso" di un cammino p.

La distanza $\delta(u,v)$ tra due vertici $u,v \in V$ è definita nel seguente modo:

$\delta(u,v) = +inf$ se non esiste un cammino orientato da u a v

$\delta(u,v) = min(w(p))$ minimo dei pesi dove p è il cammino tra u e v

Varianti

Esistono quattro diverse varianti di ricerva dei cammini minimi, combinazioni dei due casi si numerosità dei vertici di partenza e di destinazione.

Destinazione

Singola

Multipla

(tutti i vertici)

Sorgente

Singola

In: $G(V,E,w), u,v \in V$

Out: $\delta(u,v)+cm$

cm: cammino minimo

In: $G(V,E,w), s \in V$

Out: $\forall\, u \in V, \delta(s,u)$

Multipla

(tutti i vertici)

In: $G(V,E,w), d \in V$

Out: $\forall\, u \in V, \delta(u,d)$

In: $G(V,E,w)$

Out: $\forall\, u,v \in V, \delta(u,v)$

Solo i casi evidenziati verranno presi in analisi, in quanto gli altri due sono sottoproblemi di essi.

Archi con pesi negativi

Ci domandiamo se sia possibile risolvere il problema degli archi con peso negativo sommando ai pesi di tutti gli archi del grafo G la costante k capace di renderli tutti positivi. Es: $k = -min(w(u,v)) \ \forall \ u,v \in E$

Strutture dati per la rappresentazione dei cammini minimi

Per ogni vertice $u \in V$ necessitiamo di due campi:

1. $d[u]$: stima della distanza tra s ed u
2. $\pi[u]$: predecessore

Sottografo dei predecessori

Dato$G = (V, E, w)$, il sottografo dei predecessori è $G_\pi = (V_\pi, E_\pi)$ dove

$$V_\pi = \{u \in V \mid \pi[u] \neq NIL\} \cup \{s\}$$

$$E_\pi = \{(\pi[u], u) \in E \mid u \in V_\pi \setminus \{s\}\}$$

Albero dei cammini minimi

Dato$G = (V, E, w)$, l'albero dei cammini minimi $G' = (V', E')$ è un sottografo di G dove

V' : tutti gli archi ragiungibili dal vertice sorgente

G' : forma un albero radicato in s

$\forall \ v \in V'$, l'unico cammino tra s e v in G' è un cammino minimo in$G$

Procedure di modifica dei due campi

```
1 InitSingleSource(G,s)
2     foreach u in V[G] do
3         d[u] <- + inf
4         π[u] <- NIL
5     d[s] <- 0
```

```
Relax(u,v,w)
    if d[v] > d[u] + w(u,v) then
        d[v] <- d[u] + w(u,v)
        π[v] <- u
```

Dijkstra

```
Dijkstra(G,w,s)
  InitSingleSource(G,s)
  Q <- V[G]
  S <- []
  while Q != [] do
    u <- extractMin(Q)
    S <- S U {u}
    foreach v in Adj[u] do
      relax(u,v,w)
  return d,π
```

La cosa di priorità Q può essere implementata tramite

1. Array lineare
2. Heap binario

Studiamo la complessità di Dijkstra con coda di priorità implementata con array lineare:

[a]Parte Intera Inferiore

[b]?????

[c]cammino più lungo verso una foglia

[d]Complessità O(log(n))

[e]Parte Intera Inferiore

[f]Complessità O(log(n))

[g]parto da 1 (radice)

[h]invariante

[i]Complessità O(log(n))

[j]Nell'implementazione reale ci si comporta in maniera diversa, usando la parte finale della HeapIncreaseKey

[k]Complessità O(log(n))

[l]Anomalia verso il basso

Complessità: O(log(n))

[m]anomali verso il basso

Complessità : O(log(n))

[n]Aggiunta mia

[o]Che vuol dire???

[p]per il countingSort

[q]per i problemi della deleted

[r]manca

[s]???

[t]Controllare

[u]Capire la logica

[v]INCOMPLETA

[w]Pag 2- 14/3

[x]MANCA ROBA

[y]Pag 3 - 14/3

[z]Perchè il primo termine??

[aa]Eeeh...

[ab]MANCA

[ac]Solitamente quando formulata così, si procede per assurdo negando la tesi

[ad]22/3/2018

[ae]22/3/2018