Appunti di Algoritmica

Marco Finocchi, Marco Mulas

A.A. 2013/2014

Introduzione

Questo pdf rappresenta gli appunti, presi durante l'anno accademico 2013/14, del corso di *Algoritmica* (ex Algoritmi e strutture dati) della laurea magistrale in *Informatica* presso l'*Università degli studi di Roma La Sapienza* tenuto dalla professoressa *Rossella Petreschi*.
Essi non vogliono rappresentare una sostituzione al corso o alle slide dello stesso, ma solo un appoggio con il quale poter studiare la materia e superare (ci auguriamo) le varie difficoltà incontrate.
Detto ciò, per segnalazioni o refusi vi preghiamo contattare uno dei due autori via mail.

Buono studio a tutti.

`mrcfinocchi@gmail.com`
`mlsmrc@gmail.com`

Indice

Capitolo 1

Analisi ammortizzata

A differenza della notazione asintotica in cui veniva sempre calcolato il caso peggiore sulla singola operazione, nell'analisi ammortizzata viene calcolato il caso peggiore sulla *sequenza di operazioni.*

Facciamo l'esempio con la pila con operazioni *push*, *pop* e *multipop*:

- *push*{P,x}: inserimento dell'elemento x nella pila P. Tempo O(1)
- *pop*{P}: eliminazione dell'elemento di testa della pila P. Tempo O(1)
- *multipop*{P,t}: eliminazione di t elementi dalla pila. Tempo O(min{n,t})

Se si considera una sequenza di K operazioni su una pila di dimensione n il tutto mi costa O($k \cdot min${t,n}) e considerando che t al più può essere n si ha

$$O(n^2)$$

Per un analisi migliore vengono messi a disposizione tre metodi per l'analisi ammortizzata:

- *Aggregati*: misura il **costo M della sequenza delle K operazioni**, da cui una singola operazione costa $\frac{M}{K}$
- *Accantonamenti*: **alcune operazioni pagano in anticipo** il costo delle altre
- *Potenziale*: **prepagato sulla struttura dati**, in cui il costo dell'i-esima operazione viene calcolato mediante

  $$\chi_i = c_i + \Phi(d_i) - \Phi(d_{i-1})$$

  dove $\Phi(d_i) - \Phi(d_{i-1})$ rappresenta la **differenza di potenziale** ovvero il variare della struttura dati d

1.1 Pila

1.1.1 Aggregati

I costi delle operazioni singole rimangono asintoticamente le stesse, ma rispetto a prima bisogna fare una valutazione: una volta riempita la pila ed effettuata una *multipop* di t elementi rimarranno nella pila n-t elementi. Successivamente si può quindi effettuare una multipop al massimo di n-t e non da t come era stato detto nell'analisi semplice.

Da quanto detto il **costo della sequenza** è O(n), mentre il **costo della singola operazione** è pari a $\frac{O(n)}{K} = \frac{O(n)}{O(n)}$ =O(1)

1.1.2 Accantonamenti

Si va ad anticipare il costo della *pop* di un elemento al momento della push dello stesso. Quindi ora la *push* viene a costare 2, mentre la *pop* e la *multipop* conseguentemente verranno a costare 0, poichè il costo per togliere un determinato elemento dalla lista è stato anticipato dalla *pop*.

Il **costo** totale **della sequenza** per il caso degli accantonamenti è pari a $2n$, quindi O(n), e anche qui il **costo della singola operazione** è pari a O(1).

1.1.3 Potenziale

Sia d_0 la struttura dati iniziale, d_i la struttura dati che si ottiene a partire dalla struttura dati d_{i-1}

Il costo ammortizzato per l'i-esima operazione è per definizione:

$$\chi_i = c_i + \Phi(d_i) - \Phi(d_{i-1}) \tag{1.1}$$

in cui con $\Phi(d_i)$ identifichiamo il numero di elementi nella pila dopo i operazioni.

Il costo della sequenza di operazioni è:

$$\sum_{i=1}^{n}(c_i + \Phi(d_i) - \Phi(d_{i-1})) =^1$$

$$\sum_{i=0}^{n} c_i + \Phi(d_n) - \Phi(d_0) \tag{1.2}$$

Vediamo in base all'Equazione 1.1 i **costi ammortizzati delle tre operazioni**, considerando |P|=p:

- *push*{P,x} = $c_i + \Phi(d_i) - \Phi(d_{i-1})$ = 1+p-(p-1) = 2
- *pop*{P} = $c_i + \Phi(d_i) - \Phi(d_{i-1})$ = 1+(p-1)-p = 0
- *multipop*{P,t} = $c_i + \Phi(d_i) - \Phi(d_{i-1})$ = t+(p-t)-t = 0

Riprendendo l'Equazione 1.2 il **costo della sequenza** è pari a:

$$\sum_{i=1}^{n} \chi_i = \Phi(d_n) - \Phi(d_0) + \sum_{i=0}^{n} c_i \leq n - 0 + n = 2n$$

questo poichè i costi singoli sono pari a O(1) e in $\Phi(d_o)$ ci sono 0 elementi per definizione, mentre in $\Phi(d_n)$ al più ci saranno n elementi.

[1] Serie telescopica.

1.2 Contatore binario

Prendiamo ora in considerazione un contatore binario[2]:

$A(2)$	$A(1)$	$A(0)$
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Per incrementare il contatore binario e necessario muoversi da *dx* a *sx* del numero espresso in binario e si commutano gli 1 in 0 e ci si ferma al primo 0 che si commuta in 1.

Con l'**analisi non ammortizzata** nel caso peggiore il costo per passare da un numero n_i al successivo n_{i+1} sarebbe stato di $\mathrm{O}(k)$ (causato dalla commutazione di tutti i k bit del numero), per un totale di $\mathrm{O}(n \cdot k)$.

Vediamo ora con l'**analisi ammortizzata**:

1.2.1 Aggregati

Si può notare che ogni bit $\mathrm{A}(i)$ viene complementato precisamente $\left\lfloor \frac{n}{2^i} \right\rfloor$ volte. Allora il numero totale della modifica dei bit è dato da:

$$\sum_{i=0}^{logn} \left\lfloor \frac{n}{2^i} \right\rfloor < n \cdot \sum_{i=0}^{\infty} (\frac{1}{2})^i =^3 2n$$

Quindi il **costo della singola operazione** è pari a 2, mentre il **costo della sequenza** è pari a $2n$.

1.2.2 Accantonamento

Mediante accantonamento si fa pagare 2 il cambiamento dei bit da 0 ad 1, poichè viene commutato un solo bit ad 1 in 0 al passaggio da un numero n_i al suo successivo n_{i+1}, e 0 il cambiamento dei bit da 1 a 0. Si nota che quindi, assumendo che il contatore parta da 0, si ha che:

$$\mathrm{c}(\textit{sequenza di n incrementi}) \leq \mathrm{O}(n)$$

e che conseguentemente il **costo della sequenza** è lineare.

1.2.3 Potenziale

Nell'analisi ammortizzata con il metodo del potenziale definiamo come c_i il numero dei bit modificati e t_i il numero di bit che si commutano a 0 con la codifica del numero i-esimo, allora si ha che:

$$c_i = (t_i + 1) \tag{1.3}$$

ovvero il numero di bit che si commutano da 1 a 0 più l'unico bit che si commuta da 0 a 1.

Si definisce b_i il numero di bit posti ad 1 nella codifica del numero nel passo i-esimo e la differenza di potenziale $\Delta\Phi$ come la differenza di bit posti a 1 fra l'i-esima codifica e l'(i-1)-esima codifica:

[2]Per passare in base 10 basta calcolare $x = \sum_{i=0}^{k} A[i] \cdot 2^i$.

[3]Serie di potenze: $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} se |x| < 1$.

$$\Delta\Phi = b_i - b_{i-1} = (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i \tag{1.4}$$

in cui:

- b_{i-1}sono i bit ad 1 nella codifica del numero nel passo i-esimo
- $-t_i$poichè t_i bit vengono commutati da 1 a 0, conseguentemente vengono sottratti
- $+1$ poichè creando la codifica del numero successivo

Ora per 1.3 e per 1.4 si ha che:

$$\chi_i = c_i + \Delta\Phi = (t_1 + 1) + (1 - t_i) = 2$$

quindi il costo **ammortizzato** della **singola operazione** è pari a 2, mentre il **costo della sequenza** è $2n$, quindi lineare.

1.3 Gestione di tabelle dinamiche

Definiamo il fattore di carico $\alpha(T)$ come

$$\alpha(T) = \frac{num(T)}{size(T)}$$

in cui $num(T)$ sono gli elementi presenti nella tabella T.

Quando la tabella è piena si ha che $\alpha(T) = 1$ e, per "imposizione", quando la tabella è vuota si ha che $\alpha(T) = 1$.

Se l'inserimento è possibile il costo è pari ad 1

x	x
x	

nel caso in cui la tabella fosse piena bisogna creare un'altra tabella con uno spazio necessario per inserire le vecchie informazioni più le nuove. Quindi:

$c_i = 1$ se T non piena

$c_i = i = (i-1) + 1$ se T piena

Tutto ciò si può **ottimizzare**: se, nel caso in cui la tabella T fosse piena, anzichè creare una tabella con uno spazio in più per l'elemento da inserire, si andasse a creare una tabella avente il doppio dello spazio si otterrebbe metà dello spazio libero dalla nuova tabella si avrebbe che

$c_i = 1$ se T non piena

$c_i = i$ se $size(T) = i$-1 è una potenza di 2

Analizziamo in un primo momento **solo gli inserimenti** con i tre metodi e poi **l'inserimento e la cancellazione**.

1.3.1 Inserimenti

1.3.1.1 Aggregati

Consideriamo una sequenza di n inserimenti. Per ciò che è stato detto in precedenza:

$$\sum_{i=1}^{n} c(i) = n + \sum_{j=0}^{\lfloor logn \rfloor} 2^j$$

in cui n è dovuto agli inserimenti e $\sum_{j=0}^{\lfloor logn \rfloor} 2^j$ è dovuto all'espansione della tabella.

$$\sum_{i=1}^{n} c(i) = n + \sum_{j=0}^{\lfloor logn \rfloor} 2^j \leq^{4} n + \frac{2^{\lfloor log_2 n \rfloor + 1} - 1}{2 - 1} < n + 2^{log_2 n} \cdot 2 = n + 2n = 3n$$

da cui si ha che il **costo della sequenza** è 3n, mentre il **costo della singola operazione** è pari a $\chi_i = \frac{3n}{n} = 3$.

1.3.1.2 Accantonamenti

Ovviamente l'operazione che si fa pagare di più è l'inserimento che costerà 3:

- 1 per l'inserimento nella tabella corrente
- 1 per l'eventuale spostamento nella tabella di dimensione raddoppiata
- 1 per lo spostamento di uno degli elementi della vecchia tabella al momento dell'espansione della stessa.

Quindi, banalmente, il **costo della sequenza** è pari a $3n$ mentre quello della **singola operazione** è pari a 3, per quanto già detto.

1	2
3	4

$\Rightarrow$

1	2	5	
3	4		

1.3.1.3 Potenziale

Ci mettiamo nel caso in cui la tabella è piena per metà e viene inserito un elemento, ovvero $\alpha = \frac{num(T)}{size(T)} \geq \frac{1}{2}$, da cui si ha che

$$2num(T) - size(T) \geq 0$$

e definiamo

$$\Phi(T) = 2num(T) - size(T).$$

Si nota che *subito dopo una espansione* si ha che

$$\Phi(T) = 2num(T) - size(T) = 2 \cdot \frac{size(T)}{2} - size(T) = 0 \tag{1.5}$$

poichè num(T) rispetto a $size$(T) è proprio $\frac{size(T)}{2}$.
Mentre *subito prima di un'espansione* si ha che

$$\Phi(T) = 2num(T) - num(T) = num(T) \tag{1.6}$$

Ora si va a verificare se un i-esimo inserimento attiva o no una espansione.

[4] Serie geometrica.

1. non attiva espansione $\left(se \frac{num(T)+1}{size(T)} < 1\right)$

   Se non attiva un'espansione si ha che in T_i e T_{i-1} non si ha una espansione

   $\chi_i = c_i + \Delta\Phi = 1 + (2num(T_i) - size(T_i)) - (2num(T_{i-1}) - size(T_{i-1}))=$

   $= 1 - 2num(T_i) - size(T_i) - 2num(T_i) + 2 + size(T_{i-1}) = 3$

2. attiva espansione $\left(se \frac{num(T)+1}{size(T)} = 1\right)$

   Se attiva si ha che il costo della singola operazione è dovuto dallo spostamento delle $num(T_i)$ nella tabella nuova.

   Si ha banalmente che:

   - $num(T_{i-1}) = num(T_i) - 1$, poichè gli elementi all'istante (i-1)-esimo sono gli elementi all'istante i-esimo - 1
   - $size(T_{i-1}) = num(T_i) - 1$, poichè il numero degli elementi all'instante i-esimo è pari alla dimensione della tabella prima dell'espansione (istante (i-1)-esimo), quando la tabella stava per diventare piena, $+1$.
   - $c_i = num(T_i) = 1 + num(T_{i-1}) + 1$, poichè bisogna spostare gli elementi della vecchia tabella ($num(T_{i-1})$) sulla nuova tabella e in più aggiungerlo 1.

   $\chi_i = c_i + \Delta\Phi = num(T_i) + (2num(T_i) - size(T_i)) - (2num(T_{i-1}) - size(T_{i-1})) =$

   $= num(T_I) + (2num(T_i) - 2(num(T_i) - 1)) - (2(num(T_i) - 1) - (num(T_i) - 1)) =$

   $= num(T_i) + 2num(T_i) - 2num(T_i) + 2 - 2num(T_i) + 2 + num(T_i) - 1 = 3$

Quindi il **costo di una singola operazione** è 3 mentre il **costo della sequenza** è pari a $3n$

1.3.2 Inserimenti e cancellazioni

1.3.2.1 Potenziale per gli inserimenti

Avendo già considerato $\alpha(T_{i-1}) \geq \frac{1}{2}$ consideriamo ora i casi in cui

1. $\alpha(T_{i-1}) < \frac{1}{2}$, $\alpha(T_i) < \frac{1}{2}$
2. $\alpha(T_{i-1}) < \frac{1}{2}$, $\alpha(T_i) = \frac{1}{2}$, caso in cui viene provocata un'espansione

e nel caso in cui $\alpha(T_i) < \frac{1}{2}$ diciamo che

$$\Phi(T) = \frac{size(T)}{2} - num(T)$$

poichè $\alpha(T_i) = \frac{num(T_i)}{size(T_i)} < \frac{1}{2}$, da cui $\Phi(T) = \frac{size(T)}{2} - num(T) > 0$.

Ricordando che $\chi_i = c_i + \Delta\Phi$ si ha che:

1. $\chi_i = 1 + (\frac{size(T_i)}{2} - num(T_i)) - (\frac{size(T_{i-1})}{2} - num(T_{i-1})) =$
   $= 1 + \frac{size(T_i)}{2} - num(T_i) - \frac{size(T_i)}{2} - (num(T_i) - 1) = 0$
2. $\chi_i = 1 + (2num(T_i) - size(T_i)) - \left(\frac{size(T_{i-1})}{2} - num(T_{i-1})\right) =$
   $= 1 + (2(num(T_i - 1) + 1)) - size(T_{i-1})) - \left(\frac{size(T_{i-1})}{2} - num(T_{i-1})\right) =$
   $= 3num(T_{i-1}) - \frac{3}{2}size(T_{i-1}) + 3 =$
   in cui per ipotesi si ha che $\frac{num(T_{i-1})}{size(T_{i-1})} < \frac{1}{2}$, quindi $num(T_{i-1}) < \frac{size(T_{i-1})}{2}$, da cui
   $< \frac{3}{2}size(T_{i-1}) - \frac{3}{2}size(T_{i-1}) + 3 = 3$

Quindi anche in questi casi una **sequenza di operazioni** può costare al più n a partir da una tabella vuota.

1.3.2.2 Potenziale per le cancellazioni

L'operazione di **contrazione** (quando il fattore di carico è troppo piccolo), che si verifica dopo le cancellazioni, consiste nel:

- creare una nuova tabella con un numero di posizioni minore di quella vecchia;
- copiare tutti gli elementi della vecchia tabella nella nuova (idealmente si vorrebbe che: il fattore di carico sia sempre limitato inferiormente da una costante);
- il costo ammortizzato di una operazione rimanga limitato superiormente da una costante.

Consideriamo i casi in cui

1. $\alpha(T_{i-1}) < \frac{1}{2}$, $\frac{1}{4} < \alpha(T_i) < \frac{1}{2}$
2. $\alpha(T_{i-1}) < \frac{1}{2}$ e $\alpha(T_i) = \frac{1}{4}$, quindi l'$i$-esima operazione provoca una contrazione, conseguentemente $\alpha(T_i) = \frac{1}{2}$
3. $\alpha(T_{i-1}) > \frac{1}{2}$, $\alpha(T_i) > \frac{1}{2}$
4. $\alpha(T_{i-1}) > \frac{1}{2}$, $\alpha(T_i) = \frac{1}{2}$
5. $\alpha(T_{i-1}) = \frac{1}{2}$, $\alpha(T_i) < \frac{1}{2}$

Si ha che:

1. $\chi_i = 1 + (\frac{size(T_i)}{2} - num(T_i)) - (\frac{size(T_{i-1})}{2} - num(T_{i-1})) =$
   $= 1 + (\frac{size(T_i)}{2} - num(T_i)) - (\frac{size(T_i)}{2} - (num(T_i) + 1)) = 2$
2. $\chi_i = (num(Ti) + 1) + (2num(Ti) - size(Ti)) - (\frac{size(T_{i-1})}{2} - num(T_{i-1})) =$
   $= (num(T_i) + 1) + (2num(T_i) - 2num(T_i))$–$(2num(T_i) - (num(T_i) + 1) = 2$
3. $\chi_i = 1 + (2num(T_i) - size(T_i) - (2num(T_{i-1}) - size(T_{i-1})) =$
   $= 1 + ((2(num(T_{i-1}) - 1) - size(T_{i-1})) - (2num(T_{i-1}) - size(T_{i-1})) = -1$
4. CASO IDENTICO A 3
5. $\chi_i = 1 + (\frac{size(T_i)}{2} - num(T_i)) - (2num(T_{i-1}) - size(T_{i-1})) =$
   $= 1 + (\frac{size(T_i)}{2} - (\frac{size(T_i)}{2} - 1)) - (2num(T_{i-1}) - 2num(T_{i-1})) = 2$

Abbiamo mostrato come tutte le operazioni di inserimento e di cancellazione su una tabella abbiano costo ammortizzato limitato superiormente da una costante. Quindi una **qualunque sequenza di n operazioni** su una tabella dinamica richiede tempo ammortizzato O(n).

Capitolo 2

Alberi binari di ricerca

Un albero binario è un albero in cui un nodo ha al più due figli. La caratteristica dell'ABR è che dato un nodo il sottoalbero sinistro conterrà elementi minori di quel nodo e il sottoalbero destro elementi maggiori o uguali.

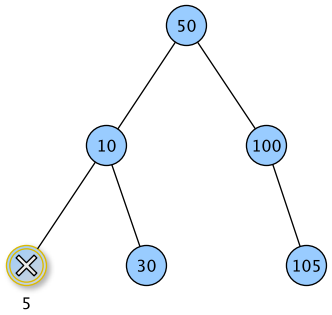
L'*inserimento* nel caso peggiore è l'altezza dell'albero. Nel caso migliore, quando si tratta di un albero binario completo, l'altezza dell'albero sarà pari a O($log\ n$), mentre nel caso peggiore, quando l'albero è totalmente sbilanciato, l'altezza sarà pari a O(n).

$$\mathrm{O}(log\ n) \leq \mathrm{O}(h(\mathrm{A})) \leq \mathrm{O}(n)$$

La *cancellazione* è un po' più complessa.

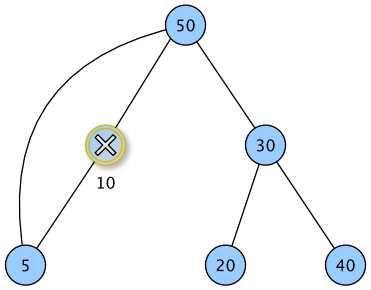
Se dovessi **eliminare una foglia** l'operazione è semplice e costa O(h).

Figura 2.1: 1° caso di cancellazione



Nel **secondo caso** se dovessi eliminare un nodo che ha un foglio mi basta ricollegare il "nonno" con il "nipote" e anche in questo caso il costo è O(h).

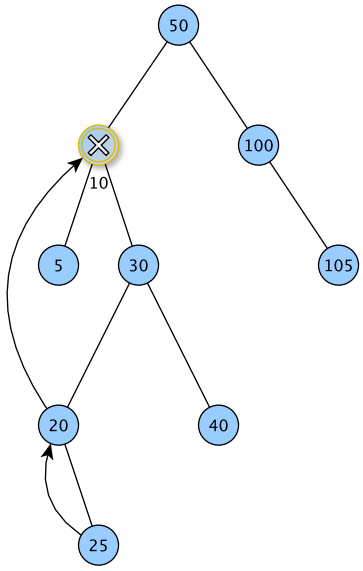
Figura 2.2: 2° caso di cancellazione



Il **terzo caso** è più complicato ed è quello in cui il nodo ha due figli. In questo caso al posto del nodo da cancellare metto il più grande del sottoalbero sinistro (il più grande dei più piccoli) oppure il più piccolo del sottoalbero destro (il più piccolo dei più grandi). Nel caso del più piccolo del sottoalbero destro andiamo nel sottoalbero destro e prendiamo il nodo più a sinistra, il quale si tratta di un nodo con figlio destro o di una foglia (in maniera speculare per il più piccolo dei più grandi).

Anche in questo caso la complessità è $O(h)$.

Figura 2.3: 3° caso di cancellazione



2.1 Costruzione di un albero ABR nel caso medio

Proposizione 1 *La complessità sulla sequenza di n inserimenti ha costo O(n log n) nel caso medio.*

Dimostrazione. Si vuol dimostrare che:
A(n)=2(n+1)H_n-4n dove H_n=$\sum_{i=1}^{n} \frac{1}{i} \leq log_2 n + \gamma$ e γ=0.5572 ottenuta dalla seguente equazione di ricorrenza:

$$\mathrm{A}(n)=\begin{cases} 0 & se \quad n = 0 \\ (n+1) + \frac{2}{n} \sum_{j=1}^{n} A(j-1) & se \quad n > 0 \end{cases}$$

Per HP si ha che le $n!$ permutazioni sono ugualmente distribuite e si ha che ogni permutazione corrisponde agli n inserimenti in sequenza.
Nel caso medio si ha che:

$$A(n) = \frac{\sum_{a \in S_n} Bliv(a)}{n!} \tag{2.1}$$

dove

- *Bliv(a)* =$\sum liv(x)$, per tutti gli x nodi di B(a)
- S_n sono le permutazioni sui primi n elementi
- $B(a)$ è l'ABR relativo alla permutazione a
- liv(x) è il numero di spigoli dalla radice ad x in $B(a)$ (diciamo "altezza di x")

Definiamo ora $\mathcal{L}(a)$ e $\mathcal{R}(a)$ i sottoalberi rispettivamente sinistri e destri dell'ABR costruito sulla permutazione a e definiamo come $S_{n,j}$ il sottoinsieme dell'insieme delle permutazioni S_n che hanno l'elemento j come primo elemento.
Si nota quindi che:

$$|S_n| = n!$$

e che

$$|S_{n,j}| = (n-1)! \tag{2.2}$$

poichè j è fissato e permuto i rimanenti $n-1$ elementi
Sappiamo che il numero di volte che ogni permutazione $\rho \in S_{j-1}$ (sui primi $j-1$ elementi) appare è pari a

$$\binom{n-1}{j-1}(n-1-(j-1))! \tag{2.3}$$

poichè fisso $j-1$ elementi da $n-1$ elementi e faccio permutare i rimanenti $(n-1-(j-1))$ ovvero $\binom{n-1}{j-1} \cdot (n-1-(j-1))!$
in cui $\binom{n-1}{j-1}$ sono gli insiemi da j-1 elementi su un insieme da $n-1$ ($n-1$ poichè è stato fissato j come primo elemento) e $(n-1-(j-1))!$ poichè fissando $j-1$ elementi permuto i rimanenti, da cui

$$\binom{n-1}{j-1} \cdot (n-1-(j-1))! = \frac{(n-1)!}{(j-1)! \cdot (n-1-(j-1))!} \cdot (n-1-(j-1))! = \frac{(n-1)!}{(j-1)!}$$

Risolviamo ora l'equazione

$$\mathrm{A}(n)=\frac{1}{n!}\sum_{a\in S_n} Bliv(a)$$

Andiamo a considerare i pesi sul sottoalbero sinistro e destro che definiremo rispettivamente con $\mathcal{L}liv(x)$ e $\mathcal{R}liv(x)$. Poichè andiamo quindi a non considerare la testa si ha che su ogni livello su $\mathcal{L}liv$(x) e $\mathcal{R}liv$(x) diminuisce di 1, quindi per ogni nodo su $\mathcal{L}liv(x)$ e su $\mathcal{R}liv(x)$ si perde di 1 sul peso. In totale sono n-1 nodi che perdono di 1 sul loro peso, conseguentemente bisognerà restituire $(n-1)$ alla sommatoria totale.

$$=\frac{1}{n!}\sum_{1}^{n!}[(n-1)+\mathcal{L}liv(a)+\mathcal{R}Liv(a)]=$$

$$=\frac{1}{n!}\sum_{1}^{n!}(n-1)+\frac{1}{n!}\sum_{1}^{n!}\mathcal{L}liv(a)+\frac{1}{n!}\sum_{1}^{n!}\mathcal{R}Liv(a)$$

Gli alberi a coppie sono speculari, quindi nelle coppie si ha che

$$\mathcal{L}liv(x)=\mathcal{R}liv(y) \text{ e } \mathcal{L}liv(y)=\mathcal{R}liv(x)$$

e che quindi presi due alberi

$$\mathcal{L}liv(x)+\mathcal{R}liv(y)+\mathcal{L}liv(y)+\mathcal{R}liv(x)=2\mathcal{L}liv(x)+2\mathcal{L}liv(y)$$

da cui, proseguendo la risoluzione della funzione di ricorrenza,

$$(n-1)+\frac{2}{n!}\sum_{1}^{n!}Lliv(a)$$

Pensiamo ora a tutte le permutazioni nella sommatoria come le permutazioni che hanno l'elemento j come primo elemento, quindi all'insieme $S_{n,j}$, iterando j da 1 a n, quindi

$$(n-1)+\frac{2}{n!}\sum_{j=1}^{n}(\sum_{a\in S_{n,j}} Lliv(a))$$

Ricordando che $\mathcal{L}liv(x)$ fa riferimento al sottoalbero sinistro della testa, si ha che gli elementi del sottoalbero sinistro sono minori di j poichè si sta prendendo in considerazione l'insieme $S_{n,j}$ in cui $j \in [n]$ è il primo elemento della permutazione. Essendo minori di j e avendo la necessità di permutarli possiamo identificarli come S_{j-1}.

Si usa qui l'equazione 2.3 e si è ragionato "bloccando" i primi $j-1$ elementi e facendo permutare i rimanenti, quindi per riottenere tutte le permutazioni rimane da permutare i primi $j-1$ elementi identificati da S_{j-1} come detto in precedenza, quindi

$$(n-1)+\frac{2}{n!}\sum_{j=1}^{n}\frac{(n-1)!}{(j-1)!}(\sum_{a\in S_{j-1}} Bliv(a))$$

Per 2.1 si sostituisce $\frac{1}{(j-1)!}\sum_{a\in S_{nj-1}} Bliv(a)$ con $\mathrm{A}(j-1)$, quindi

$$(n-1)+\frac{2}{n!}\sum_{j=1}^{n}(n-1)!A(j-1)=$$

$$=(n-1)+\frac{2}{n}\sum_{j=1}^{A}(j-1)$$

Risolviamo ora l'equazione:

$$\mathrm{A}(n)=\begin{cases} 0 & se \quad n=0 \\ (n-1)+\frac{2}{n}\sum_{j=1}^{n} A(j-1) & se \quad n>0 \end{cases}$$

Moltiplico per n entrambi i membri

$$nA(n) = n(n-1) + 2\sum_{j=1}^{n} A(j-1) \tag{2.4}$$

Viene calcolata l'equazione (2.4) per n-1

$$(n-1)A(n-1) = (n-1)(n-2) + \sum_{j=1}^{n-1} A(j-1) \tag{2.5}$$

Si sottrae l'equazione (2.4) all'equazione (2.5) e si divide per $2n(n+1)$

$$\frac{nA(n)}{2n(n+1)} = \frac{2(n-1)+(n+1)A(n-1)}{2n(n+1)} =$$

$$\frac{(n-1)}{n(n-1)} + \frac{A(n-1)}{2n} =$$

$$\frac{2n-n-1}{n(n+1)} + \frac{A(n-1)}{2n} =$$

$$\frac{2n}{n(n+1)} - \frac{n+1}{n(n+1)} + \frac{A(n-1)}{2n}$$

Si applica ora il metodo di sostituzione in maniera ripetuta

$$\frac{A(n)}{2(n+1)} = \frac{2}{n+1} - \frac{1}{n} + \frac{A(n-1)}{2n} =$$

$$= \frac{2}{n+1} - \frac{1}{n} + \left[\frac{2}{n} - \frac{1}{n-1} + \frac{A(n-2)}{2(n-1)}\right] =$$

$$= \frac{2}{n+1} - \frac{1}{n} + \left[\frac{2}{n} - \frac{1}{n-1} + \left[\frac{2}{n-1} - \frac{1}{n-2} + \frac{A(n-3)}{2(n-2)}\right]\right] =$$

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots + \frac{2}{n-(i-2)} - \frac{1}{n} - \frac{1}{n-1} - \frac{1}{n-2} + \ldots - \frac{1}{n-(i-1)} + \frac{A(n-i)}{2(n-(i-1))} =$$

Il metodo di sostituzione viene ripetuto fino a quando n-i=0 (passo base), ovvero quando n=i

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots + \frac{2}{2} - \frac{1}{n} - \frac{1}{n-1} - \frac{1}{n-2} + \ldots - 1 + A(0) =$$

$$= \frac{2}{n+1} + 2\sum_{i=2}^{n} \frac{1}{i} - \sum_{i=1}^{n} \frac{1}{i} + 0 =$$

La prima sommatoria parte da i=2, ma scrivendo 0 come $^2/_1$-$^2/_1$ possiamo portare $^2/_1$ dentro la prima sommatoria facendola così partire da i=1, lasciando -$^2/_1$

$$= \frac{2}{n+1} + 2\sum_{i=1}^{n} \frac{1}{i} - \sum_{i=1}^{n} \frac{1}{i} - 2 =$$

$$= \frac{2}{n+1} - 2 + \sum_{i=1}^{n+1} \frac{1}{i} =$$

$$= - \frac{2n}{n+1} + \sum_{i=1}^{n} \frac{1}{i}$$

Quindi

$$\frac{A(n)}{2(n+1)} = -\frac{2n}{n+1} + \sum_{i=1}^{n} \frac{1}{i} \tag{2.6}$$

L'equazione 2.6, moltiplicando entrambi i membri per $2(n+1)$, si risolve in

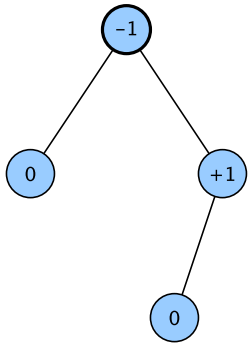
$$\mathrm{A}(n) = 2(n+1)\sum_{i=1}^{n} \frac{1}{i} - 4n \ \square$$

2.2 Alberi binari di ricerca bilanciati

Gli alberi di ricerca bilanciati affrontati nel corso di Introduzione degli Algoritmi sono gli alberi AVL. Sono caraterizzati da un fattore di bilanciamento su ogni nodo fra -1 e 1 che ci garantisce l'altezza logaritmica

$$-1 \leq f(b) \leq 1$$

Figura 2.4: Esempio di AVL



L'altezza dell'albero AVL è *logaritmica*.

Definizione 1 *Tra tutti gli alberi di altezza h bilanciati in altezza, un albero di Fibonacci ha il minimo numero di nodi. Un albero di Fibonacci di altezza h può essere costruito unendo mediante l'aggiunta di una radice, un albero di Fibonacci di altezza h-1 ed un albero di Fibonacci di altezza h-2*

Proposizione 2 *Sia T_h un albero di Fibonacci di altezza h e sia n_h il numero dei suoi nodi. Risulta $h=\Theta(log n_h)$*

Dimostrazione. Per costruzione di T_h risulta che $n_h = 1 + n_{h-1} + n_{h-2}$. Questo ricorda molto da vicino l'equazione dei numeri di Fibonacci: $F_i = F_{i-1} + F_{i-2}$. Per induzione verrà dimostrato che

$$n_h = F_{n+3} - 1$$

Il passo ase, per h=0, è banalmente verificato essendo $n_0 = 1$ ($= F_3$ - 1 = 2 - 1). Assumendo per ipotesi induttiva che $n_k = F_{k+3} - 1 \; \forall k < h$, ed usando le ricorrenze relativa ad $n_h$ed F_i, si ha:

$$n_h = 1 + n_{h-1} + n_{h-2} = 1 + (F_{h+2} - 1) + (F_{h+1} - 1) = F_{h+3} - 1$$

Ricordando ora che $F_h = \Theta(\phi^h)$ dove $\phi \approx 1.618$ è la sezione aurea.

L'altezza e il numero di nodi di $T_h$sono quindi esponenzialmente correlate, e pertanto $h=\Theta(log\ n_h)$.

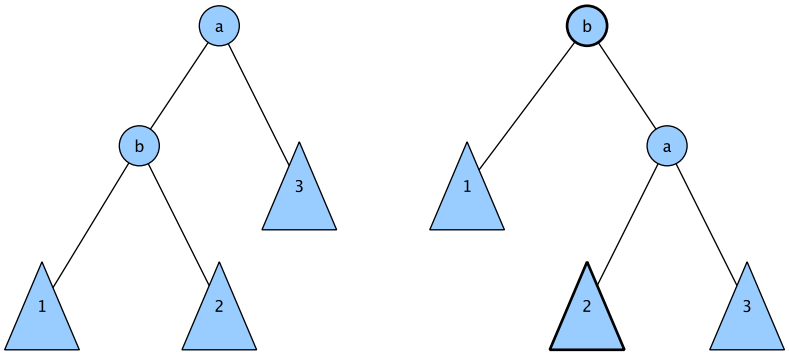
□

2.2.1 Bilanciamento AVL

Per mantenere l'altezza logaritmica all'interno di un AVL si utilizzano due tipi di rotazione:

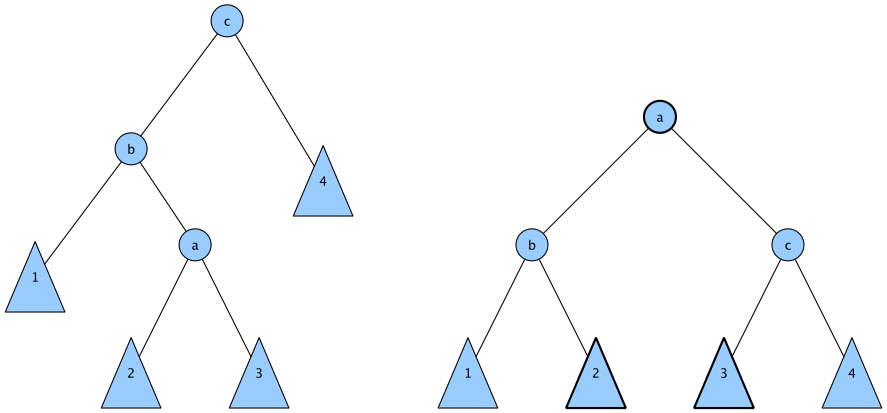
- la rotazione a destra di a e b (pensiamo nel caso in cui b è figlio sinistro di a) è fatta facendo diventare:
  - a come figlio dx di b
  - il sottoalbero dx di b diventa il sottoalbero sx di a

Figura 2.5: Rotazione a destra



- la rotazione a sinistra-destra di a, b e c (pensiamo al caso in cui b è il figlio sinistro di a e c è figlio destro di b) è fatta facendo diventare:
  - b come figlio sx di a ($rot.sx$)
  - il sottoalbero sx di a diventa il sottoalbero dx di b ($rot.sx$)
  - c come figlio dx di a ($rot.dx$)
  - il sottoalbero dx di a diventa il sottoalbero sx di c ($rot.dx$)

Figura 2.6: Rotazione a sinistra-destra



2.3 B-Alberi (generalizzazione ABR)

I B-Alberi sono alberi con altezza logaritmica implementati su memoria esterna.

Definizione 2 *Un B-Albero è un albero di ricerca 2t-ario (al più 2t figli, con t arbitrario e t≥2)*

Si hanno nei B-Alberi le seguenti proprietà:

1. ogni nodo interno x ha $n(x)$ figli dove:
$$t \leq n(x) \leq 2t$$
2. la radice ha almeno 2 figli, altrimenti verrebbe rappresentata solo da un unico puntatore
3. gli $n(x)$ figli sono radici dei sottoalberi $A_1(x)$, ..., $A_{n(x)}(x)$ raggiungibili tramite puntatori
4. $\forall x$ con $n(x)$ figli sono associate ($n(x)$-1) chiavi t.c.
$$K_1 \leq K_2 \leq K_3 \leq \ldots \leq K_{n(x)-1}$$
(PR.ORDINAMENTO)
5. tutte le chiavi del sottoalbero A(i) sono maggiori delle chiavi del sottoalbero A(i-1) e minori[1] delle chiavi del sottoalbero A(i+1) (PR.ABR)
6. tutte le foglie hanno la stessa profondità (PR.RAFFORZATA)

Definizione 3 *Diremo che un nodo è pieno se ha 2t-1 chiavi, quindi 2t figli*

Proposizione 3 *L'altezza di un B-Albero da K chiavi, con K≥1 e t≥2 è logaritmica*

Dimostrazione. Osserviamo subito che per il numero di chiavi vale sempre la seguente relazione:

$$\mathrm{K} \geq 1 + (t\text{-}1) \sum_{i=1}^{h} 2t^{i-1}$$

questo è possibile riassumerlo sotto altra forma:

$$K = \sum_{i=0}^{h} \textit{chiavi su meganodi a livello } h$$

in cui il meganodo rappresenta il nodo contenente le chiavi e si può affermare che per la proprietà 2 e 4 la radice ha almeno 2 figli e quindi almeno 1 chiave quindi

$$K = 1 + 2 \cdot \sum_{i=1}^{h} \textit{chiavi su meganodi a livello } h$$

in cui, per la proprietà 4, ogni meganodo ha almeno $t-1$ chiavi , da cui

$$K \geq 1 + 2(t-1) \cdot \sum_{i=1}^{h} \textit{nodi a livello } h$$

e facilmente si sa che i *nodi a livello h* sono t^{i-1}a livello i, da cui

[1] l'uguale lo assumiamo legato con il segno minore

$$\mathrm{K} \geq 1 + (t\text{-}1) \sum_{i=1}^{h} 2t^{i-1}$$

Si risolve ora

$$\mathrm{K} \geq 1 + 2(t\text{-}1) \sum_{i=1}^{h} t^{i-1}$$

$$\mathrm{K} \geq 1 + 2(t\text{-}1) \frac{t^h - 1}{t - 1}$$

$$\mathrm{K} \geq 1 + 2(t^h\text{-}1)$$

$$\mathrm{K} \geq 1 + 2t^h\text{-}2$$

da cui

$$\mathrm{K} \geq 2t^h\text{-}1$$

$$\frac{K+1}{2} \geq t^h$$

dove

$$h \leq log_t \frac{K+1}{2}$$

segue quindi che asintoticamente l'altezza dell'albero è O($log\ K$). □

2.3.1 Inserimento

L'operazione di inserimento in un B-Albero è analoga a quella già vista per un albero binario di ricerca. Vengono sempre effettuati i seguenti passi:

- Si controlla x (elemento da inserire) con i valori $k_1(\mathrm{y}) \leq k_2(\mathrm{y}) \leq \ldots \leq k_{n(y)-1}(\mathrm{y})$ del nodo che si sta analizzando(inizialmente $y = r$)
- Se $k_1(y) > x$, si confronta x con il primo figlio di y
- Se $k_i(y) < x < k_{i+1}(y)$ si confronta x con il figlio $(i+1)$-esimo di y
- Se $k_{n(y)-1}(\mathrm{y}) < \mathrm{x}$, si confronta x con l'ultimo figlio di y

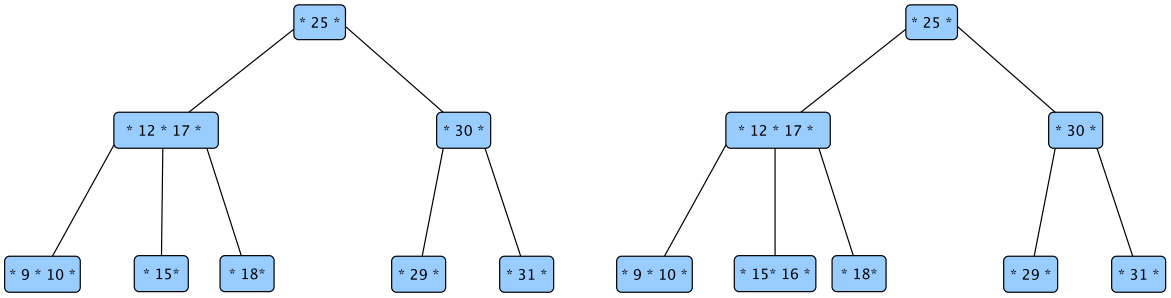
Si prosegue ricorsivamente sul cammino fintanto che non si trova libera la posizione appropriata sull'albero.

Per inserire mantenendo invariate le proprietà del B-Albero (nei seguenti esempi $t = 2$) verranno analizzati i seguenti casi e per ognuno di essi sarà presente una sequenza di operazioni ben definita:

1. *Blocco-foglia con meno di 2t-1 chiavi:*

   L'inserimento avviene in maniera banale cercando la posizione opportuna nel blocco-foglia.

Figura 2.7: Esempio, inserimento della chiave 16

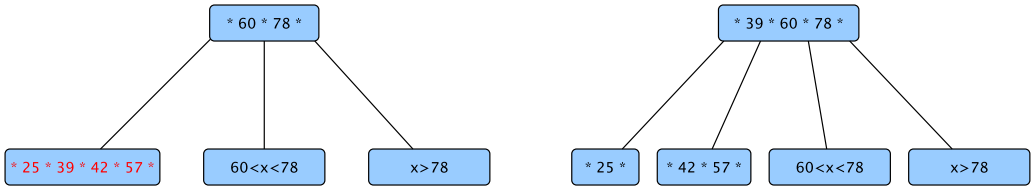


2. *Blocco-foglia con 2t-1 chiavi e blocco-padre con meno di 2t-1 chiavi:*

   Una volta inserito l'elemento nel blocco-foglia viene estratto l'elemento in posizione $\left\lfloor \frac{2t-1}{2} \right\rfloor$ e inserito nel blocco-padre.

   Le chiavi nel blocco-foglia nelle posizioni inferiori a $\left\lfloor \frac{2t-1}{2} \right\rfloor$ verranno raggruppate per formare il blocco-figlio associato al puntatore subito a sinistra dell'elemento appena inserito, mentre le chiavi nel blocco-foglia nelle posizioni superiori a $\left\lfloor \frac{2t-1}{2} \right\rfloor$ verranno associate al puntatore subito a destra.

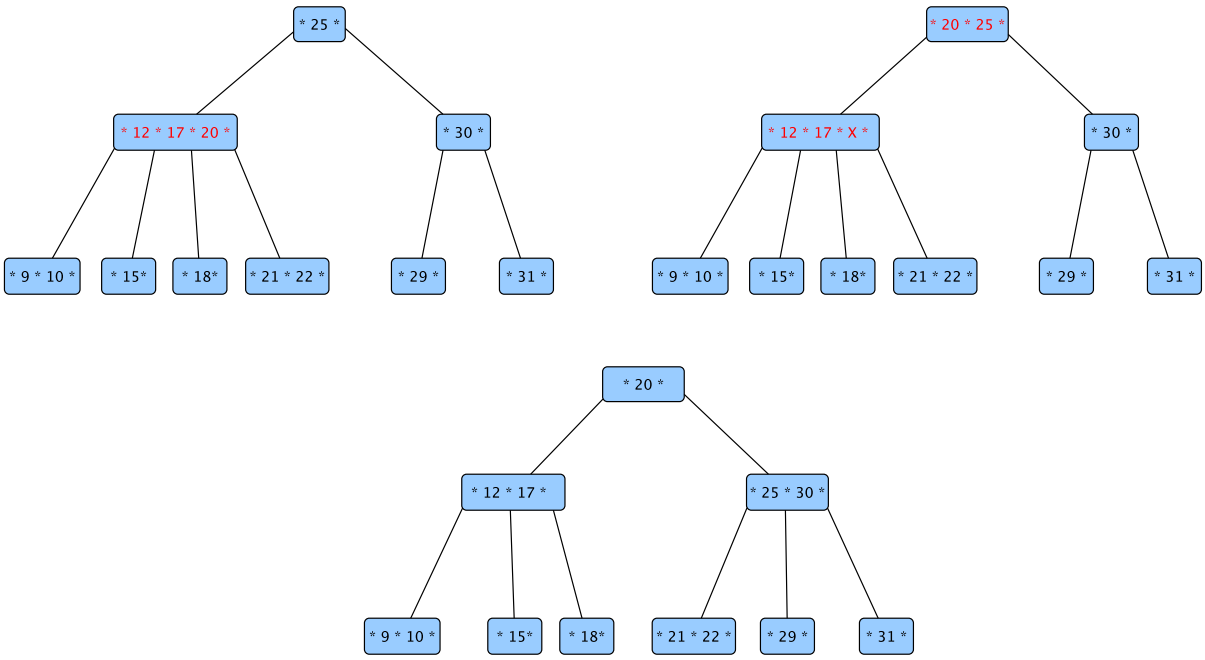
Figura 2.8: Esempio



3. *Blocco-foglia con 2t-1 chiavi, blocco-padre con 2t-1 chiavi e fratello del padre* [2] *con meno di 2t-1 chiavi:*

   Una volta inserito l'elemento nel blocco-foglia e applicato il secondo caso, nel caso in cui il fratello destro del padre(o sinistro) abbia meno di $2t$-1 chiavi viene scelto l'elemento destro (o sinistro) del blocco padre e salvato il relativo blocco destro (o sinistro) puntato (Figura 2.9.a). L'elemento estratto dal blocco-padre va a finire nel blocco-nonno nella posizione più a sinistra (o destra) (Figura 2.9.b) e viene fatto successivamente scendere l'elemento più a destra (o sinistra) nel blocco del fratello del padre avente meno di $2t$-1 chiavi nella posizione più a sinistra (o destra) e come puntatore più a sinistra rispetto al blocco viene inserito il puntatore in precedenza salvato (Figura 2.9.c)

Figura 2.9: Esempio nel caso di fratello del padre destro con meno di $2t$-1 chiavi



4. *Blocco-foglia con 2t-1 chiavi, blocco-padre con 2t-1 chiavi e fratello destro e sinistro del padre con 2t-1 chiavi:*

   Una volta inserito l'elemento nel blocco-foglia, non potendo applicare il caso 3 visto che i blocchi-zio destro e sinistro sono pieni, si procede come nel caso 2 splittando il blocco foglia in due parti e portando l'elemento mediano al blocco padre che a sua volta non potendo accogliere tale elemento verrà splittato e ne verrà mandato il suo elemento mediano al blocco-nonno. In questo caso possiamo trovarci in una delle tre seguenti situazioni mutuamente esclusive:

   - il blocco-nonno può mantenere l'elemento mediano del blocco-padre;
   - viene effettuata una ridistribuzione delle chiavi se il fratello del blocco-nonno può accogliere nuovi elementi;

[2] *per non avere troppa complessità si vedono sono il fratello destro e il fratello sinistro più vicini*

- viene a sua volta splittato il blocco-nonno e si ripetono le operazioni ricorsivamente. Se esso era la radice allora l'elemento mediano di tale blocco diventerà la nuova radice dell'albero *andando così ad incrementarne l'altezza.*

2.3.2 Cancellazione

La cancellazione di una chiave, a differenza dell'inserimento, può avvenire in un qualsiasi nodo del B-Albero.
Pertanto, ci sarà utile la seguente definizione:

Definizione 4 *Un nodo interno u con esattamente $t-1$ chiavi è detto* ***nodo interno critico****. Se il nodo u è invece una foglia, è detto* ***foglia critica****.*

Prima di cancellare un elemento bisogna individuarne la posizione all'interno dell'albero, sono quindi necessari i seguenti passi:

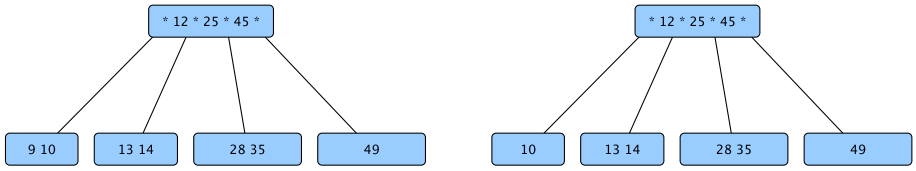
1. Si controlla x (elemento da eliminare) con i valori $k_1(y) \leq k_2(y) \leq \ldots \leq k_{n(y)-1}(y)$ del nodo che si sta analizzando (inizialmente y è la radice).
2. Se $k_1(y) > x$, si confronta x con il primo nodo figlio di y.
3. Se $k_i(y) < x < k_{i+1}(y)$ si confronta x con il nodo figlio $(i+1) - esimo$ di y.
4. Se $k_{n(y)-1}(y) < x$, si confronta x con l'ultimo nodo figlio di y.

Si procede così finchè non si individua la chiave con valore x all'interno dell'albero, se tale chiave non esiste la procedura di eliminazione termina senza effettuare nessun'altra operazione.

Una volta individuata la chiave da eliminare si opera a seconda dei casi:

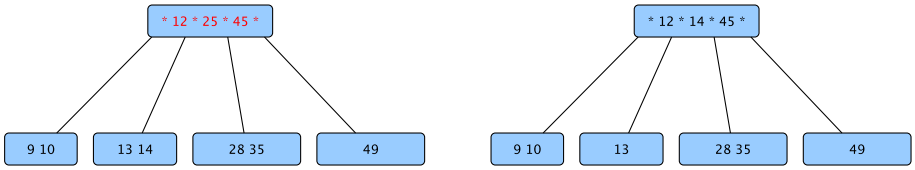
1. La chiave x da eliminare si trova in una **foglia**:
   - Eliminazione diretta della chiave x e terminazione.

Figura 2.10: Esempio di eliminazione della chiave 9 con $t = 2$.



2. La chiave x da eliminare si trova in un **nodo interno**:
   - Ricerca del $pred(x)$ o $succ(x)$ il quale sarà sicuramente in una foglia (il più grande dei più piccoli o il più piccolo dei più grandi), se essa non è una foglia critica allora si sostituisce con tale valore la chiave x, si rimuove fisicamente il valore dalla foglia e si termina.

Figura 2.11: Esempio di eliminazione della chiave 25 con $t = 2$.

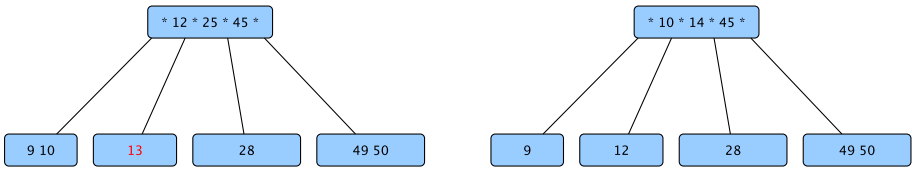


- Nel caso invece in cui entrambi (sia $pred(x)$ che $succ(x)$) sono su di una foglia critica:
  (a) Effettuare comunque la sostituzione della chiave x con il suo predecessore (o successore).
  (b) Procedere seguendo il caso 3 assumendo come chiave da eliminare il predecessore (o successore) della chiave x originale ma nella sua posizione originale.

3. La chiave x da eliminare si trova in una **foglia critica**:

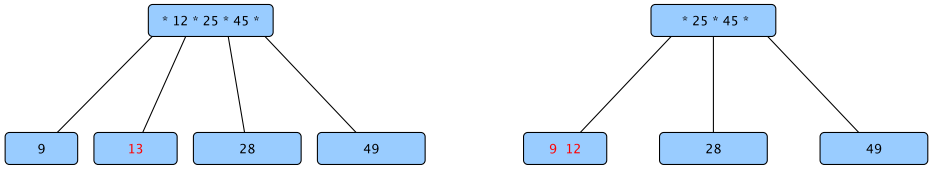
- Controllo: il fratello del nodo in cui si trova x è anch'esso una foglia critica?
  – NO: Il $pred(x)$[3] sostituisce x ed il $pred(pred(x))$ sostituisce $pred(x)$, eliminazione fisica del $pred(pred(x))$ e terminazione.

Figura 2.12: Esempio di eliminazione della chiave 13 con $t = 2$.



  – SI: Eliminazione fisica di x. Fusione delle $t-1$ chiavi del fratello del nodo di x, delle $t-2$ chiavi rimanenti del nodo che contiene x e della chiave del padre separatrice. Si avrà così un nodo con esattamente $2t-2$ chiavi, il quale non viola le proprietà sul numero massimo di chiavi per nodo, ovvero $2t-1$. Potrebbe essere necessario dover ripetere queste stesse operazioni di ridistribuzione delle chiavi, in maniera ricorsiva (dal penultimo livello questa volta), nel caso in cui il padre era anch'esso un nodo critico. Il caso peggiore può arrivare fino alla radice, causando così un *decremento dell'altezza dell'albero.*

Figura 2.13: Esempio di eliminazione della chiave 13 con $t = 2$.

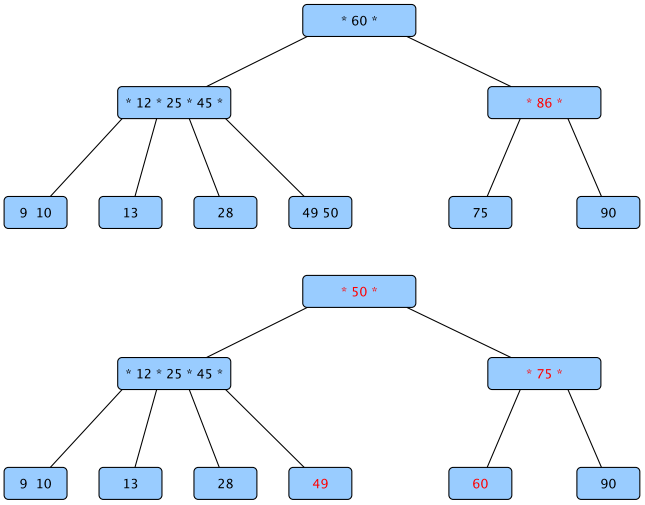


[3]Non sempre si opera con i predecessori. È soltanto un esempio, dipende dalla posizione del nodo.

4. La chiave x da eliminare si trova in un **nodo interno critico**:

   - Se il predecessore di x non si trova su di una foglia critica si può procedere come nel caso 2a. Altrimenti vengono effettuate le seguenti operazioni:
     (a) Il $pred(x)$ sostituisce x.
     (b) Il $pred(pred(x))$ sostituisce il $pred(x)$ nella sua posizione originale.
     (c) Controllo: Il $pred(pred(pred(x)))$ è su una foglia critica?
     - NO: $pred(pred(pred(x)))$ sostituisce $pred(pred(x))$ nella sua posizione originale, eliminazione fisica di $pred(pred(pred(x)))$ e terminazione.

Figura 2.14: Esempio di eliminazione della chiave 86 con $t = 2$.



   - SI: $pred(pred(pred(x)))$ sostituisce $pred(pred(x))$ nella sua posizione originale e la procedura continua dal punto 3, la quale assume come chiave da eliminare $pred(pred(pred(x)))$ ma nella sua posizione originale.

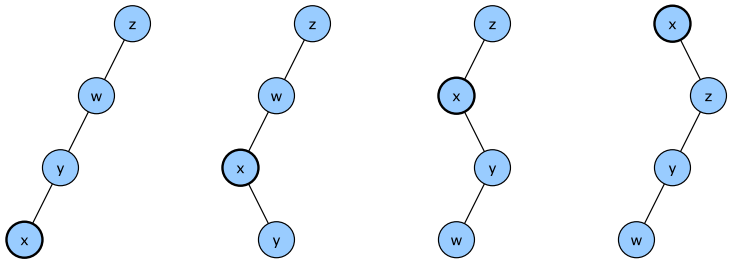
2.4 Alberi binari di ricerca autoaggiustanti

Sono stati introdotti per l'organizzazione dei dati in memoria cache, infatti i dati usati più di recenti li troviamo ai livelli alti dell'albero.

L'operazione che consente di riportare i dati verso i livelli alti dell'albero viene denominata *movetoroot.* Essenzialmente si tratta di rotazioni semplici che permettono di muovere un generico elemento x verso la radice.

Mostriamo ora come opera la *movetoroot* su un generico elemento dell'albero:

Figura 2.15: Esempio di *movetoroot* del vertice x



Ovviamente, come si può ben vedere dalla Figura 2.15, l'altezza dell'albero rimane invariata, come il suo bilanciamento.

Vediamo ora una variante dovuta a Robert Tarjan, il quale propone più varianti di rotazioni che permettono al nodo di risalire di due livelli ad ogni passo, anzichè uno (come nel caso della Figura 2.15) :

Figura 2.16: Variante di Tarjan CASO 1

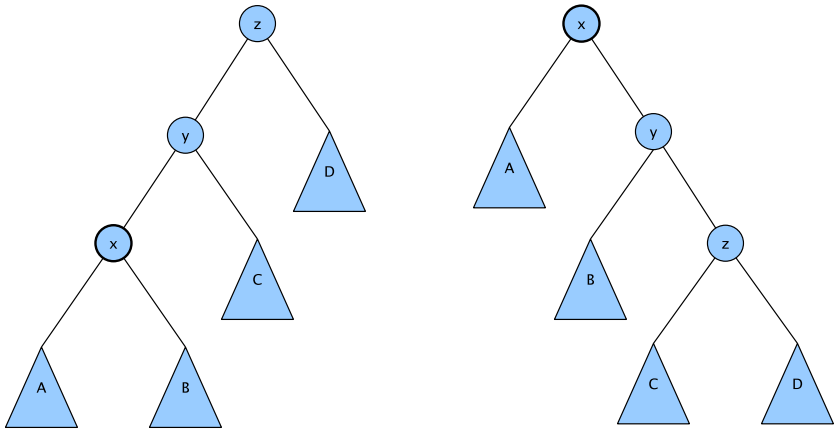
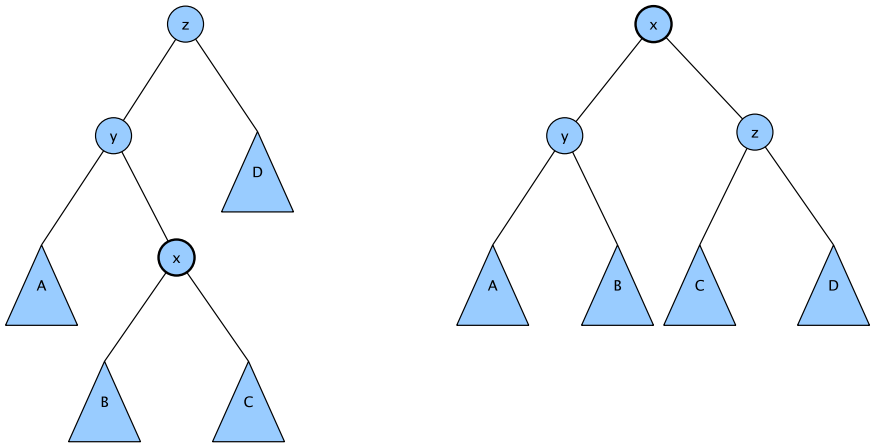
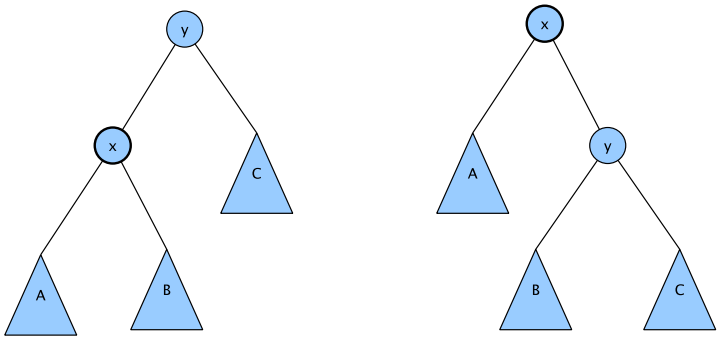


Figura 2.17: Variante di Tarjan CASO 2



Ovviamente non sempre i livelli saranno divisibili per due, quindi si valuta un terzo caso di incremento unitario di livello della x.

Figura 2.18: Variante di Tarjan CASO 3



Con questa modifica si garantisce che un albero autoaggiustante abbia tempo di esecuzione ammortizzato *logaritmico* su di una sequenza di operazioni, senza mantenere alcuna condizione esplicita di bilanciamento.

Dimostriamo ora mediante il metodo del potenziale quanto appena affermato:

Definiamo il potenziale come

$$\Phi = \sum_{x \in T} r(x)$$

dove

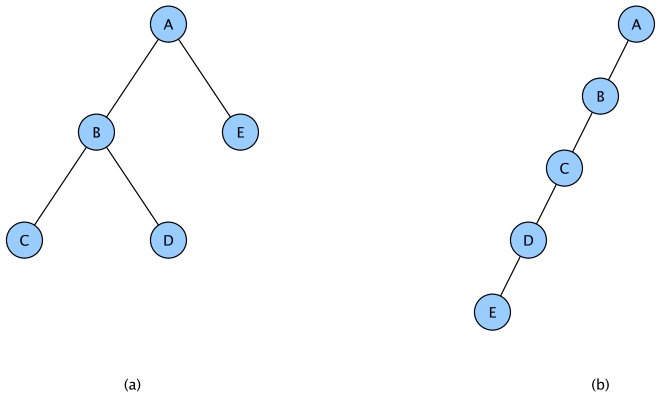
$$r(x) = log(elementi\ del\ sottoalbero\ radicato\ in\ x) \tag{2.7}$$

da cui

$$\Phi = \sum_{x \in T} log(size(x))$$

Vediamo come si comporta il potenziale su di un esempio:

Figura 2.19: Comportamento potenziale



$\Phi_a = log5 + log3 + 3log1 \approx 3{,}9$
$\Phi_b = log5 + log4 + log3 + log2 + log1 \approx 5{,}9$

Si nota, da questa definizione di potenziale, che esso è inversamente proporzionale al bilanciamento dell'albero.

Per la seguente dimostrazione ci sarà utile la seguente definizione:

Definizione 5 *Si definisce operazione di splay l'insieme delle operazioni utili a riportare un elemento generico x alla radice dell'albero.*

Proposizione 4 *Il costo ammortizzato di un'operazione di splay è pari a O(log m), dove m è pari al numero di nodi sull'albero su cui si sta lavorando.*

Dimostrazione. Ci limiteremo ad analizzare soltanto le rotazioni del CASO 1 e del CASO 3, poichè il CASO 2 può essere ricondotto facilmente a quanto affermeremo per il CASO 1.

- CASO 1:

  Introduciamo prima un Lemma che sara' utile successivamente:

  Lemma 1 *Dati α e β reali e positivi (>0) si ha che se $\alpha + \beta \leq 1$ allora $log\alpha + log\beta \leq 2$*

  Dimostrazione. Per le proprietà sui logaritmi si sa che

  $$log\alpha + log\beta = log(\alpha \cdot \beta)$$

  La funzione $log(\alpha \cdot \beta)$ raggiunge il massimo nella condizione $\alpha + \beta \leq 1$ quando $\alpha = \beta = \frac{1}{2}$ e quindi banalmente si avrà che $log(\frac{1}{4}) = -2$ e quindi che $log(\alpha \cdot \beta)$ è limitata superiormente sotto le ipotesi fatte da -2. □

  Scegliamo ora

  $$\alpha = \frac{size_{i-1}(x)}{size_i(x)} > 0$$

$$\beta = \frac{size_i(z)}{size_i(x)} > 0$$

quindi banalmente

$$\alpha + \beta = \frac{size_{i-1}(x) + size_i(z)}{size_i(x)}$$

in cui $\alpha + \beta \leq 1$ poichè, come si può vedere dalla Figura 2.16, si ha che $size_{i-1}(x) + size_i(z) \leq size_i(x)$ poichè rispetto all'albero radicato in x manca il nodo y.

Applicando il Lemma 1 otteniamo che

$$log\left(\frac{size_{i-1}(x)}{size_i(x)}\right) + log\left(\frac{size_i(z)}{size_i(x)}\right) \leq -2$$

da cui

$$r_{i-1}(x) - r_i(x) + r_i(z) - r_i(x) \leq -2$$

cambiando verso

$$2 \cdot r_i(x) \text{ - } r_{i-1}(x) \text{ - } r_i(z) \text{ -2} \geq 0$$

Secondo il metodo del potenziale ogni rotazione χ_i del CASO 1 ha costo

$$\chi_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$$

Applicando le dovute sostituzioni e ricordando che i ranghi dei nodi che subiscono variazioni rimangono inalterati, da cui

$$\Phi(T_i) - \Phi(T_{i-1}) = r_i(x) + r_i(y) + r_i(z) - (r_{i-1}(x) + r_{i-1}(y) + r_{i-1}(z))$$

otteniamo:

$$\chi_i = 2 + r_i(x) + r_i(y) + r_i(z) \text{ - } r_{i-1}(x) \text{ - } r_{i-1}(y) \text{ - } r_{i-1}(z)$$

poichè, come si può vedere dalla Figura 2.16, i ranghi dei nodi nei sottoalberi A, B, C e D non cambiano dopo la rotazione e quindi ininfluenti ai fini del costo.

Si nota inoltre che, per definizione di rango,

$$r_{i-1}(z) = r_i(x)$$

segue quindi che

$$\chi_i = 2 + r_i(y) + r_i(z) \text{ - } r_{i-1}(x) \text{ - } r_{i-1}(y)$$

la quale possiamo maggiorare con

$$\leq 2 + r_i(x) + r_i(z) \text{ - } 2{\cdot}r_{i-1}(x) + [\ 2{\cdot}r_i(x) \text{ - } r_{i-1}(x) \text{ - } r_i(z) \text{ - } 2\ ] = 3[\ r_i(x) \text{ - } r_{i-1}(x)\ ]$$

poichè sappiamo, sempre per la Figura 2.16, che

$$r_i(y) < r_i(x),\ r_{i-1}(x) < r_{i-1}(y)$$

da cui

$$-r_{i-1}(x) > -r_{i-1}(y)$$

ed inoltre, per il Lemma 1, che

$$2{\cdot}r_i(x) \text{ - } r_{i-1}(x) \text{ - } r_i(x) \text{ - } 2 \geq 0$$

- CASO 2
  si riconduce al CASO 1

- CASO 3
  $\chi_i = 1 + r_i(x) + r_i(y) \text{ - } r_{i-1}(x) \text{ - } r_{i-1}(y)$
  Per i sottoalberi A, B, C e per $r_i(x)$ e $r_{i-1}(y)$ vale quanto detto per il CASO 1, quindi

  $$\chi_i = 1 + r_i(y) \text{ - } r_{i-1}(x)$$

  maggiorato per la Figura 2.18 da

  $$\chi_i \leq 1 + r_i(x) \text{ - } r_{i-1}(x)$$

  il quale è a sua volta maggiorabile con

  $$\chi_i \leq 1 + 3 \cdot [r_i(x) \text{ - } r_{i-1}(x)]$$

Il **costo** quindi di **una operazione di splay** è dato da

$$\sum_{i=1}^{k} \chi_i$$

dove k sono le operazioni per lo *splay*

$$\sum_{i=1}^{k-1} \chi_i + \chi_k \tag{2.8}$$

dove $\sum_{i=1}^{k-1} \chi_i$ sono le rotazioni doppie (CASO 1 e 2) in cui

$$\chi_i = 3[r_i(x) - r_{i-1}(x)]$$

e χ_k è l'eventuale rotazione singola (CASO 3) se l'altezza è dispari, in cui

$$\chi_k = 1 + 3[r_k(x) - r_{k-1}(x)]$$

Ritornando all'equazione 2.8, si può calcolare il costo dello *splay*, pari, per quanto detto, a

$$\sum_{i=1}^{k-1} 3[r_i(x) - r_{i-1}(x)] + 1 + 3[r_i(x) - r_{i-1}(x)] =^4$$

$$1 + 3r_m \text{ - } 3r_0$$

in cui per 2.7 si ha che

$$r_0 = log(size_0(x)) = log(1) = 0$$

poichè il sottoalbero radicato in x ha inizialmente solo x come elemento e quindi $size_0(x)$
e quindi

$$r_k(x) = log_k(x)$$

si può ora facilmente capire il costo dell'operazione di splay che è pari a

$$1 + 3 \cdot log_k(m) \ = O(log\, m)$$

Una sequenza di δ operazioni (inserimento, cancellazione, ricerca) costa quindi $O(\delta \cdot log\, n)$ dove per n si intende il massimo valore tra gli $m_1, ..., m_\delta$ sapendo che m_i indica il valore di m dell'albero ottenuto dopo l'$i - esima$ operazione. □

[4]Serie telescopica.

2.5 Alberi di intervalli

Un albero di intervalli, che chiameremo AI, è un albero bilanciato (ad esempio un AVL) dove ciascun nodo memorizza un intervallo i rappresentato come coppia ordinata $(sx(i), dx(i))$.
Su un AI si esegue:

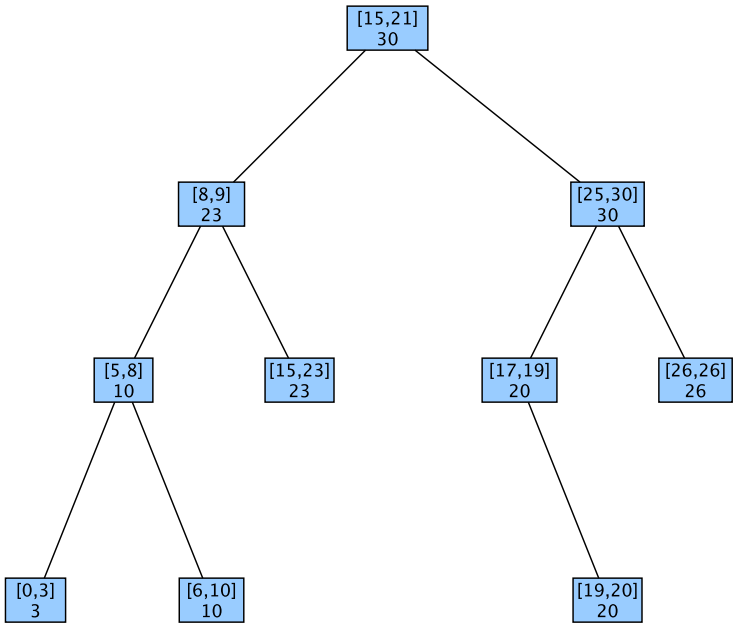
- Inserimento di un intervallo
- Eliminazione di un intervallo
- Ricerca, se esiste, di un intervallo i nell'albero che si sovrapponga ad un dato intervallo x

A differenza dell'inserimento e della cancellazione, lavorando con gli AI, bisognerà inserire informazioni aggiuntive per riuscire a mantenere un tempo $O(h)$ anche per la ricerca.
La procedura di costruzione degli alberi di intervalli è la seguente:

1. Inserisci considerando l'ordine sull'estremo sinistro $sx(i)$
2. Per ogni nodo si inserisce, come **informazione aggiuntiva**, il nodo di valore massimo del suo sottoalbero destro

   Questa procedura è abbastanza semplice: si procede ricorsivamente verso le foglie dell'albero in cui banalmente il massimo è $dx(i)$ e si ritorna questo valore verso la chiamata del padre, il quale sceglierà il proprio massimo fra i massimi riportati dai propri figli e il proprio $dx(i)$.

Figura 2.20: Esempio di AI



La procedura di ricerca è la seguente:

Algoritmo 2.1 Ricerca(AI, intervallo x)

i = radice
while $i \neq$ null e x non si sovrappone a AI(i) **do**
  if *figlio_sx*(i)≠null e max[*figlio_sx*(i)] $\geq sx(x)$ **then**
    i = *figlio_sx*(i)
  else
    i = *figlio_dx*(i)
return i

Correttezza della procedura Ricerca. Vogliamo dimostrare che se ad un certo passo abbiamo fatto una scelta di uno dei due sottoalberi, sicuramente nel sottoalbero non scelto non potrà esserci alcun intervallo che si potrebbe intersecare con x.

Dividiamo i due casi:

- Se **la ricerca procede a destra** necessariamente si avrà che $max[figlio\_sx(i)] < sx(x)$ allora **tutti** *gli estremi destri degli intervalli* i' *nel sottoalbero* **sinistro** *sono tali che* $max[figlio\_sx(i)] \geq dx(i')$ da cui si evince:

$$\forall i' \quad dx(i') < sx(x)$$

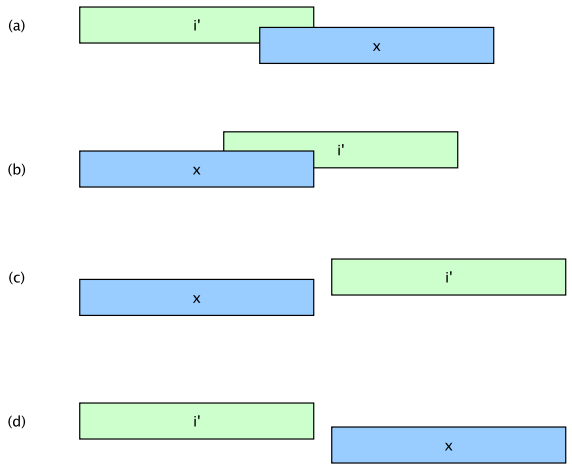
  segue quindi che ogni intervallo nel sottoalbero sinistro non può intersecarsi con x.

- Se **la ricerca procede a sinistra** necessariamente si avrà che

$$max[figlio\_sx(i)] \geq sx(x) \tag{2.9}$$

  ed inoltre che nel sottoalbero sinistro di AI(i) esisterà sicuramente un intervallo i' tale che $dx(i') = max[figlio\_sx(i)]$ per definizione di max.

Figura 2.21: Casi possibili



Osserviamo che per (2.9) non è possibile il caso 2.21d della Figura 2.21, mentre possono risultare possibili i casi 2.21a,b,c.
Supponiamo ora che nessun intervallo nel sottoalbero sinistro (quello scelto) si inteseca con x e mostriamo che anche tutti gli intervalli presenti nel sottoalbero **destro** (non scelto) non possono intersecarsi con x. Quindi l'unico caso che descrive la situazione è il caso 2.21c, da cui si ricava che:

$$sx(i') > dx(x)$$

quindi per costruzione dell'albero ogni $sx(i'')$ dove i'' appartiene al sottoalbero **destro** (non scelto) verifica la condizione che

$$sx(i'') \geq sx(i') > dx(x)$$

da cui segue immediatamente la tesi. □

Capitolo 3

Union&Find

Nel corso di Progettazione di Algoritmi della triennale, più precisamente, durante la spiegazione dell'algoritmo di Kruskal per il calcolo dell'MST, è stata introdotta una struttura dati per insiemi disgiunti, la *Union&Find*. Riassumiamo brevemente il suo funzionamento tramite l'algoritmo di Kruskal (per E^* si intende archi pesati non orientati).

Algoritmo 3.1 MSTbyKruskal(V,E^*)

Ordina E^* in base al peso di ogni arco in maniera crescente
$Comp = \emptyset$

per ogni $v \in V$
    $Comp = Comp \cup$ **MakeSet**(v)

per ogni $\{u, v\} \in E^*$ and $|Comp| > 1$
    if (**Find**(u)=**Find**(v)) scarta $\{u, v\}$
    else **Union**(**Find**(u),**Find**(v))

if ($|Comp| = 1$) return $Comp$
else No MST

In questo algoritmo: l'operazione di **MakeSet** crea per ogni nodo del grafo una componente connessa, l'operazione di **Find** cerca la componente connessa associata ai due nodi estremi di un arco ed infine l'operazione di **Union** unisce (con un arco) le componenti connesse di appartenenza dei due nodi coinvolti. Si nota facilmente che se due nodi sono nella stessa componente connessa allora l'operazione di Union non sarà possibile poichè essa comporterebbe un ciclo.
Vediamo ora come diverse implementazioni di Union&Find modificano la complessità dell'algoritmo.

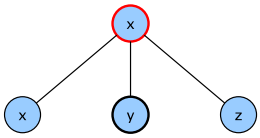
3.1 Quick-Find

Singola operazione di MakeSet in $O(1)$: riceve un nodo x e lo pone come foglia di un albero radicato in x stesso.

Figura 3.1: MakeSet del nodo x

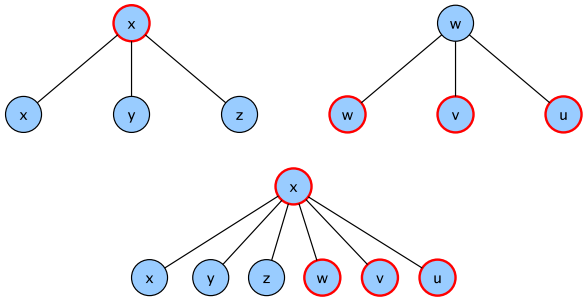
Singola operazione di Find in $O(1)$: per ogni nodo trovare la componente si riduce a trovare il padre nell'albero della componente.

Figura 3.2: Find del nodo y



Singola operazione di Union in $O(n)$: quando vengono unite due componenti si puo' arrivare a dover effettuare n cambiamenti di padre (radice), uno per ogni foglia.

Figura 3.3: Union delle componenti x e w



Nel caso di Quick-Find, la complessità dell'algoritmo di Kruskal è quindi data da:

$$O(n \cdot \log(n)) + O(n) + O(m) + O(n^2) = O(n^2).$$

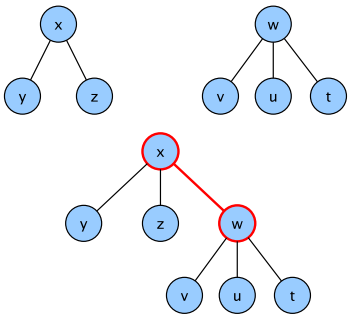
3.2 Quick-Union

Singola operazione di MakeSet in $O(1)$: riceve un nodo x e lo pone come radice di un albero appena creato.

Figura 3.4: MakeSet del nodo x

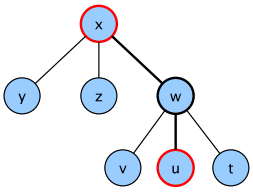
Singola operazione di Union in $O(1)$: uno dei due alberi delle componenti connesse diventa sotto-albero della radice dell'altro.

Figura 3.5: Union delle componenti x e w



Singola operazione di Find in $O(n)$: si tratta di ricercare la radice a cui è associato un nodo, puo' costare fino ad $O(n)$ nel caso di uno sbilanciamento totale dell'albero.

Figura 3.6: Find del nodo u



Nel caso di Quick-Union, la complessità dell'algoritmo di Kruskal è quindi data da:

$$O(n \cdot \log(n)) + O(n) + O(m \cdot n) + O(n) = O(m \cdot n).$$

3.3 Quick-Find bilanciato

Cambiamo il padre (radice) della componente che ha meno nodi (foglie). Notiamo subito che con questo accorgimento se due componenti A e B hanno la stessa cardinalità allora la componente risultante avrà una cardinalità pari a $2 \cdot |A|$ oppure $2 \cdot |B|$. In generale, prese due componenti qualsiasi A e B con $|A| \geq |B|$ si ha che la cardinalità della componente risultante C soddisfa la seguente relazione:

$$|C| \geq 2 \cdot |B|$$

Se durante tutto l'algoritmo una generica foglia effettua k cambiamenti di radice la relazione precedente ci permette di scrivere che:

$$n \geq 2^k$$

e passando ai logaritmi si ottiene che:

$$k \leq \log(n)$$

Nel caso di Quick-Find bilanciato, la complessità dell'algoritmo di Kruskal è quindi data da:

$$O(n \cdot \log(n)) + O(n) + O(m) + O(n \cdot \log(n)) = O(m + n \cdot \log(n)).$$

3.4 Quick-Union bilanciato

L'albero che diventa sotto-albero della radice dell'altro è quello che ha altezza minore.
Indicheremo con $size(x)$ il numero di nodi presenti nell'albero radicato in x e con $rank(x)$ un *limite superiore* alla sua altezza.
Dimostriamo per induzione (forte) che:

$$size(x) \geq 2^{rank(x)} \tag{3.1}$$

Passo base:
Albero da un solo nodo quindi: $size(x) = 1$ e $rank(x) = 0 \Rightarrow size(x) = 2^{rank(x)}$.

Passo induttivo:
Ipotesi induttiva: se $size(x) \in \{0, ..., k-1\}$ si ha che $size(x) \geq 2^{rank(x)}$.
Tesi induttiva: per $size(x) = k$ vale che $size(x) \geq 2^{rank(x)}$.

1. Caso $rank(x) < rank(y)$ e tali che $size(x) + size(y) = k$

   Per ipotesi sia ha che $rank(x) < rank(y)$, quindi x sarà figlio di y. Ci si chiede ora quale sia la nuova altezza di y, quindi il $rank(y)$, dopo che y ha acquisito x, e il suo sottoalbero radicato in esso, come suo figlio. Come conseguenza dell'ipotesi si ha che

   $$rank(x) + 1 \leq rank(y)$$

   la quale ci assicura che nonostante l'acquisizione di x come figlio da parte di y, andando ad aumentare l'altezza di 1 ($rank(x) + 1$), la nuova altezza di y rimane $rank(y)$.

   Per ipotesi induttiva sappiamo che:

   $$size(x) \geq 2^{rank(x)} \text{ e } size(y) \geq 2^{rank(y)}$$

   segue quindi che:
   $k \geq 2^{rank(x)} + 2^{rank(y)} > 2^{rank(y)} = 2^{size(x \bigcup y)}$

2. Caso $rank(y) < rank(x)$ e tali che $size(x) + size(y) = k$

   Analogo al caso precedente.

3. Caso $rank(x) = rank(y)$ e tali che $size(x) + size(y) = k$

   Notiamo subito che *il rank dell'albero risultante sarà pari a* $rank(x) + 1$ oppure $rank(y) + 1$ e per ipotesi induttiva sappiamo che:

   $$size(x) \geq 2^{rank(x)} \text{ e } size(y) \geq 2^{rank(y)}$$

   segue quindi che:

   $k \geq 2^{rank(x)} + 2^{rank(y)} = 2^{rank(x)+1}$ □

Sapendo che $size(x) \leq n$ osserviamo che per quanto appena dimostrato:

$$size(x) \geq 2^{rank(x)}$$

e passando ai logaritmi si ottiene:

$$rank(x) \leq \log(size(x))$$

e quindi che

$$rank(x) \leq log(n) \tag{3.2}$$

Notiamo ora alcune **proprietà del rank**:

1. Quando un nodo x viene creato $rank(x) = 0$
2. Quando facciamo la *union* fra due alberi con testa x e y con stesso rank r ($rank(x) = rank(y) = r$) l'albero risultante con testa x (o con testa y) sarà tale che $rank(x) = r + 1$ (o $rank(y) = r + 1$) e l'albero che è diventato figlio di x (o di y) non cambierà più il suo *rank* che rimarrà r (vedere proprietà 3).
3. Quando un nodo x non è più radice allora rank(x) rimarrà invariato.

Nel caso di Quick-Union bilanciato, la complessità dell'algoritmo di Kruskal è quindi data da:

$$O(n \cdot \log(n)) + O(n) + O(m \cdot \log(n)) + O(n) = O((m + n) \cdot \log(n)).$$

Vediamo adesso come, utilizzando una tecnica diversa durante l'operazione di Find nella procedura Quick-Union, si riesce ad ottenere una complessità *ammortizzata* pari a $O((m + n) \cdot \log^*(n))$.

3.5 Quick-Union con Path-Compression

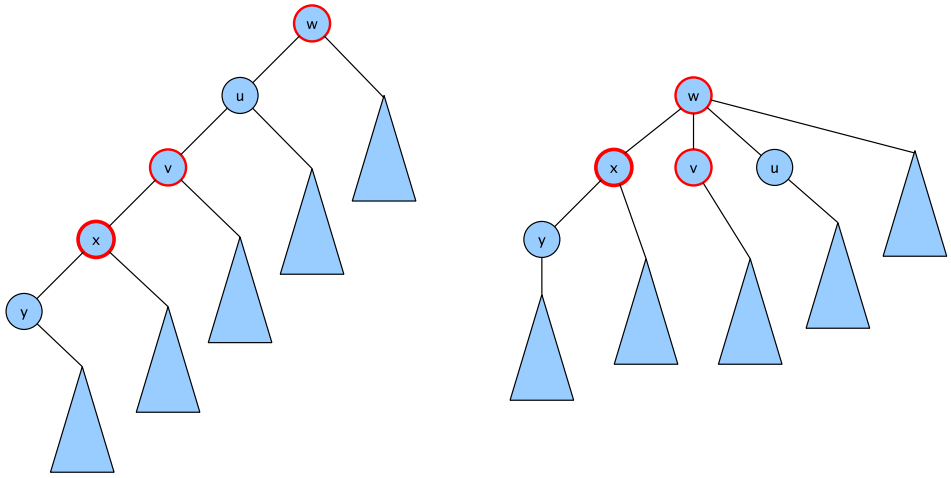
3.5.1 Euristica

Sempre nell'ambito della rappresentazione Quick-Union, è possibile introdurre ulteriori accorgimenti volti a migliorare la complessità dell'operazione di find. Vedremo in questo paragrafo l'euristica di Path Compression avente come obiettivo la riduzione dell'altezza di un albero. L'euristica di Path Compression si serve dell'algoritmo di find facendo leva sul movimento che esso esegue nella ricerca dell'etichetta posta alla radice: risalita lungo un intero verso dell'albero. L'idea alla base di tale euristica è quella di assegnare un ulteriore compito all'algoritmo di find: ristrutturare l'albero ponendo il puntatore al padre di ogni nodo incontrato lungo il cammino uguale alla radice dell'albero. Eseguiamo in tal modo una compressione dell'altezza dell'albero lungo tutto il cammino che dal nodo contenente l'elemento argomento della find termina nella radice.

3.5.2 Path-Compression

La tecnica del Path-Compression funziona nel seguente modo:
Siano $u_0, u_1, ..., u_h$ i nodi incontrati dall'operazione Find(x) durante il cammino verso la radice con $x = u_0$ e $root = u_h$.
Ogni nodo u_i con $i \in \{0, ..., h-2\}$ viene reso figlio della radice. Si osservi la figura sottostante:

Figura 3.7: Path-Compression



Attraverso il metodo degli accantonamenti (o dei crediti) dimostriamo la complessità ammortizzata di $O(m + n \cdot \log(n))$.
Assegnamo i seguenti crediti:

- MakeSet $\rightarrow 1 + \log(n)$
- Union $\rightarrow 1$
- Find $\rightarrow 2$

Notiamo dalla figura precedente che, con questa tecnica, l'altezza dell'albero radicato in w potrebbe variare (in negativo) a seguito di una compressione. Per 3.2 sappiamo che $rank(x) \leq \log(n)$ (caso della radice finale).
Osserviamo quindi che, per quanto appena detto, i crediti assegnati sono ben distribuiti:

- MakeSet $\rightarrow 1+\log(n) \rightarrow$ costo dell'operazione di MakeSet addizionato al costo del Path-Compression nel caso peggiore, per ogni nodo[1]
- Union $\rightarrow 1 \rightarrow$ costo dell'operazione di Union
- Find $\rightarrow 2 \rightarrow$ costo per la radice ed il figlio della radice poichè essi saranno comunque visitati, il costo del Path-Compression è pagato dal MakeSet

In conclusione:

- n operazioni di MakeSet
- m operazioni di Find
- al più $n-1$ operazioni di Union

hanno un costo *ammortizzato* pari a:

$$O(n \cdot \log(n)) + O(m) + O(n) = O(m + n \cdot \log(n)).$$

Quanto appena visto è frutto di un'analisi superficiale. Vediamo adesso come sia possibile ottenere la complessità *ammortizzata* promessa nel paragrafo precedente (cioè $O((m+n) \cdot \log^*(n))$) attraverso un'analisi più raffinata.
Iniziamo introducendo la funzione $\log^*(n)$:

$$\log^*(n) = \begin{cases} 0 & n \leq 1 \\ min\{i : \log^i(n) \leq 1\} & n > 1 \end{cases}$$

dove $\log^i(n) = \log(\log(\log ... \log(n)))$ per i volte.
Si noti che la funzione $\log^*(n)$ cresce molto più lentamente della funzione $\log(n)$.
Si consideri anche la seguente funzione:

Definizione 6 *Si definisce la funzione F sui naturali secondo la seguente ricorrenza:*

$$F(i) = \begin{cases} 1 & i = 0 \\ 2^{F(i-1)} & i > 0 \end{cases}$$

Di seguito alcuni esempi:

$F(0) = 1$
$F(1) = 2$
$F(2) = 4$
$F(3) = 16$
$F(4) = 65536$
$F(5) = 2^{65536}$
...
$F(i) = 2^{F(i-1)}$

Osserviamo che la funzione F è legata alla funzione $\log^*(n)$:

$\log^*(x) = 0 \quad x \leq 1$
$\log^*(x) = 1 \quad 1 < x \leq 2$
$\log^*(x) = 2 \quad 2 < x \leq 4$
$\log^*(x) = 3 \quad 4 < x \leq 16$
$\log^*(x) = 4 \quad 16 < x \leq 65536$

[1] Ogni nodo subisce un Path-Compression (una risalita) di al massimo $\log(n)$ passi.

$\log^*(x) = 5 \qquad 65536 < x \leq 2^{65536}$
...
$\log^*(x) = i \qquad 2^{F(i-2)} < x \leq 2^{F(i-1)}$

Partizioniamo ora i nodi in blocchi nel seguente modo: se un nodo v è tale che $rank(v) = r$ allora esso verrà inserito nel blocco $B(\log^*(r))$. Si avrà necessariamente che il numero di blocchi è esattamente pari a $\log^*(n)$ poichè si parte dal blocco etichettato con 0 e si arriva a $\log^*(\log(n)) = \log^*(n) - 1$, poichè $r \leq \log(n)$. Elenchiamo ora i blocchi:

$B(0)$ contiene nodi con rango in $[0, F(0)]$
$B(1)$ contiene nodi con rango in $[F(0)+1, F(1)]$
$B(2)$ contiene nodi con rango in $[F(1)+1, F(2)]$
...
$B(i)$ contiene nodi con rango in $[F(i-1)+1, F(i)]$
...
$B(\log^*(n)-1)$ contiene nodi con rango in $[F(\log^*(n)-2)+1, F(\log^*(n)-1)]$

Ogni blocco $B(i)$ contiene nodi, i quali hanno un numero di ranghi diversi tra loro ($\#r_{i\neq}$) che è al più $F(i) - F(i-1) - 1$ che è a sua volta limitato superiormente da $F(i)$. Possiamo quindi concludere che:

$$\#r_{i\neq} \leq F(i) - F(i-1) - 1 \leq F(i)$$

da cui si ottiene:

$$\#r_{i\neq} \leq F(i) \tag{3.3}$$

Assegniamo i crediti in maniera differente a quanto visto precedentemente:

- Find $\rightarrow 1 + \log^*(n)$
- MakeSet $\rightarrow 1 + \log^*(n)$
- Union $\rightarrow 1$

Dimostriamo che essi sono ben distribuiti.
Ci saranno utili le seguenti premesse:

1. I crediti dell'operazione di Find servono per i nodi che non sono nel blocco del proprio padre (chiamiamoli crediti di tipo A), per il figlio della radice e la radice
2. I crediti dell'operazione di MakeSet servono per i nodi che sono nel blocco del proprio padre (chiamiamoli crediti di tipo B) e per l'operazione MakeSet stessa
3. Il credito dell'operazione di Union è per il costo della stessa

Come nella prova precedente, ogni volta che viene applicata una *Find* su un nodo x, essa opera su di un cammino π che va da x alla radice (Figura 3.7). Tutti i nodi presenti su questo cammino (eccetto il figlio della radice) subiranno una compressione la quale cambierà il loro padre. Ogni volta che questo accade, un generico nodo v sul cammino π sarà tale che il $rank(newParent(v)) > rank(oldParent(v))$ cioè il "nuovo" padre avrà *rank* maggiore del "vecchio" padre. Questo porta a concludere che tale nodo v, inizialmente richiederà crediti di tipo B, ma quando inizierà a richiedere crediti di tipo A tale richiesta sarà permanente.
Iniziamo osservando che su un cammino π, durante un'operazione di *Find*, il massimo numero di nodi che possono richiedere crediti di tipo A sono al più $\log^*(n) - 1$. Infatti supponendo che la radice sia in un blocco $B(i)$, tutti gli altri nodi potranno necessariamente occupare i blocchi da $B(0)$ a $B(i-1)$ se sono in un blocco diverso dal padre, sapendo che i blocchi sono esattamente $\log^*(n)$ la tesi segue immediatamente.
I crediti dell'operazione di Find sono quindi sufficienti perchè aggiungendo anche il costo della visita della radice e suo figlio otteniamo che:

$$\log^*(n) - 1 + 1 + 1 = 1 + \log^*(n)$$

Per 3.3 si ha che in un blocco $B(i)$ vale che $\#r_{i\neq} \leq F(i)$, quindi un nodo presente in un blocco $B(i)$ potrà richiedere (durante tutta la sequenza di operazioni) al più $F(i)$ volte crediti di tipo B. Tutto ciò è dovuto anche al fatto che se un generico nodo x inizia a richiedere crediti di tipo A, non richiederà mai più crediti di tipo B.
Arrivati a questo punto occorre fare un piccolo inciso prima di concludere la dimostrazione.

Lemma 2 *Al più $n/2^r$ nodi possono avere rank uguale ad r durante l'esecuzione dell'algoritmo.*

Dimostrazione. Quando un nodo x viene ad avere $rank(x) = r$, questo vuol dire che è stata appena effettuata una union ed è diventato radice e si ha, per 3.1, che $size(x) \geq 2^{rank(x)} = 2^r$ ed etichettiamo questo albero con $x(r)$.
Diciamo ora che ci sono j alberi etichettati con $x(r)$, aventi quindi almeno 2^r nodi, quindi ci saranno

$$j \cdot 2^r nodi$$

che fanno riferimento ad una etichetta $x(r)$.
Sapendo che ci sono n nodi nella foresta di alberi allora si può dire che

$$n \geq j \cdot 2^r$$

da cui

$$j \leq \frac{n}{2^r}$$

Per avere quindi una stima di quanti nodi si trovano in un generico blocco $B(i)$ possiamo osservare che:

$$\sum_{r=F(i-1)+1}^{F(i)} \frac{n}{2^r} \leq n \cdot \left(\sum_{r=0}^{F(i)} \frac{1}{2^r} - \sum_{r=0}^{F(i-1)} \frac{1}{2^r} \right) = n \cdot \left(\frac{\left(\frac{1}{2}\right)^{F(i-1)+1} - \left(\frac{1}{2}\right)^{F(i)+1}}{\frac{1}{2}} \right) \leq \frac{n \cdot \left(\frac{1}{2}\right)^{F(i-1)+1}}{\frac{1}{2}}$$

ed infine concludere che:

$$\frac{n \cdot \left(\frac{1}{2}\right)^{F(i-1)+1}}{\frac{1}{2}} = \frac{n}{2^{F(i-1)}} = \frac{n}{F(i)}$$

Questo ci permette di concludere che i crediti dell'operazione di MakeSet sono sufficienti poichè i crediti di tipo B richiesti da ogni blocco $B(i)$ sono limitati superiormente da:

$$\frac{n}{F(i)} \cdot F(i) = n$$

Essendoci esattamente $\log^*(n)$ blocchi differenti ed esattamente n operazioni di MakeSet, la dimostrazione è completata.

Affermiamo quindi che:

- n operazioni di MakeSet
- m operazioni di Find
- al più $n - 1$ operazioni di Union

hanno un costo *ammortizzato* pari a:

$$O(n) + O(n \cdot \log^*(n)) + O(m) + O(m \cdot \log^*(n)) + O(n) = O((m+n) \cdot \log^*(n)).$$

Capitolo 4

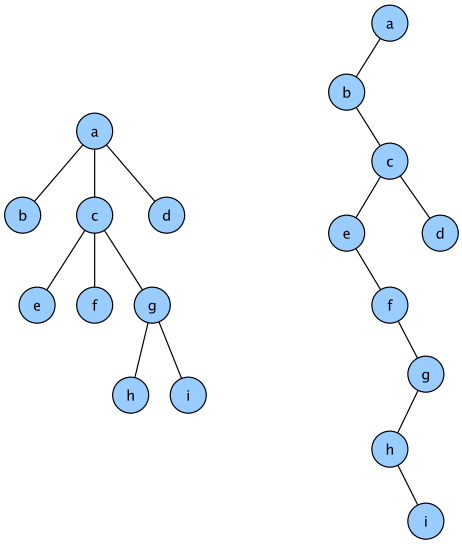
Rappresentazione succinta di alberi

In questo capitolo mostreremo come è possibile rappresentare, in maniera efficiente, alberi *statici* ordinali.

Definizione 7 *Un albero statico ordinale è un albero che non varia la sua struttura e che non ha alcun vincolo sul numero dei figli di ogni nodo.*

Iniziamo facendo notare che ogni albero ordinale da n nodi è ***binarizzabile***, ovvero riconducibile ad un albero binario da n nodi[1].
La seguente figura mostra tale binarizzazione.

Figura 4.1: Esempio di binarizzazione di un albero ordinale



La procedura è molto semplice: si uniscono tutti i figli di ogni nodo, a partire dalla radice, scollegandone i relativi collegamenti ai loro padri tranne quello del figlio estremo a sinistra.
Questo piccolo inciso serve a farci capire che possiamo concentrarci soltanto sulla rappresentazione succinta degli alberi binari statici, poichè ogni qualsiasi albero è binarizzabile.

[1]Attenzione: non è vero il contrario. Per avere infatti una funzione biiettiva tra i due insiemi è necessario considerare alberi ordinali da $n+1$ nodi ed alberi binari da n nodi. Tuttavia, per i nostri scopi, ci serviremo della sola funzione di mappatura tra gli alberi ordinali da n nodi e gli alberi binari da n nodi.

Iniziamo dicendo che il numero di alberi binari costruibili su n nodi è esattamente l'n-esimo numero di Catalano[2], ovvero:

$$C_n = \sum_{s=0}^{n-1} C_s \cdot C_{n-s-1} = \frac{\binom{2n}{n}}{n+1}$$

La domanda seguente è quindi: "Quanti bit occorrono per rappresentare la struttura di un albero binario statico da n nodi ?"
La risposta è ovviamente:

$$\lceil \log_2 C_n \rceil$$

Mostriamo che:

$$\lceil \log_2 C_n \rceil \geq 2n - O(\log n)$$

Per fare ciò dimostriamo prima che:

$$\binom{2n}{n} \geq \frac{2^{2n}}{2n}$$

$$\binom{2n}{n} = \frac{2n!}{n! \cdot n!} = \frac{2n!}{n! \cdot n!} \cdot \frac{2n}{2n} = \frac{2n \cdot (2n-1) \cdot (2n-2) \cdot ... \cdot 1}{n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1 \cdot n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1} \cdot \frac{2n}{2n} =$$

$$= \frac{1}{2n} \cdot \frac{2n \cdot 2n}{n \cdot n} \cdot \frac{(2n-1) \cdot (2n-2)}{(n-1) \cdot (n-1)} \cdot ... \cdot \frac{3 \cdot 2}{1 \cdot 1} \geq \frac{1}{2n} \cdot \frac{2n \cdot 2n}{n \cdot n} \cdot \frac{(2n-2) \cdot (2n-2)}{(n-1) \cdot (n-1)} \cdot ... \cdot \frac{2 \cdot 2}{1 \cdot 1} =$$

$$= \frac{1}{2n} \cdot 2^2 \cdot 2^2 \cdot ... \cdot 2^2 = \frac{2^{2n}}{2n}$$

Possiamo ora affermare che:

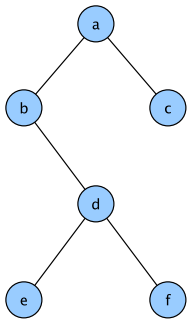
$$\lceil \log_2 C_n \rceil = \left\lceil \log_2 \frac{\binom{2n}{n}}{n+1} \right\rceil \geq \left\lceil \log_2 \frac{2^{2n}}{2n(n+1)} \right\rceil \geq \log_2(2^{2n}) - \log_2(2n \cdot (n+1)) = 2n - O(\log n) \qquad \square$$

Questa dimostrazione ci fornisce un lower-bound alla complessità spaziale di tale rappresentazione.
Le implementazioni che conosciamo dai corsi di algoritmi della triennale sono due: esplicita ed implicita. Entrambe hanno dei difetti. La prima è onerosa in termini di spazio poichè vengono mantenuti i dati così come sono e quindi per ogni nodo abbiamo bisogno di allocare memoria per mantere padre ed eventuali figli, la seconda invece è applicabile solo ad una ristretta categoria di alberi (Heap e Alberi Binari Completi) poichè utilizza memoria aggiuntiva (inizializzata a 0) per occupare lo spazio dovuto alla mancanza dei figli di un generico nodo.
Vedremo adesso come rappresentare in maniera succinta[3] *un qualsiasi albero binario statico* da n nodi.

[2] Affermazione non dimostrata a lezione e fuori programma.
[3] Rappresentazione che ottimizza lo spazio utilizzato e garantisce le operazioni più frequenti in tempi efficienti.

Consideriamo l'albero nella figura sottostante.

Figura 4.2: Albero binario



Rappresentiamo tale albero con due array come segue:

NODE:	a	b	c	d	e	f

CHILD:	1	1	1	0	1	0	0	1	1	0	0	0	0

Come si può intuire, il primo array rappresenta i nodi $\{a, b, c, d, e, f\}$ nelle posizioni da 0 a 5, mentre il secondo array rappresenta un valore 1 in prima posizione che serve ad indicare la presenza di una radice e nelle posizioni successive, a due a due, indica la presenza o meno del figlio (sinistro o destro) di ogni nodo. Ad esempio se consideriamo il nodo b, osserviamo che in concomitanza con le posizioni 3 e 4 dell'array CHILD sono presenti rispettivamente uno 0 ed un 1 i quali rappresentano il fatto che il nodo b non ha il figlio sinistro (infatti 0) ma ha solo il figlio destro (infatti 1).
Non è difficile capire quale sia la regola che dato un generico nodo x ci permette di risalire alle posizioni che occupano i suoi figli nell'array CHILD:

Se un generico nodo x occupa la posizione i in NODE $\Rightarrow$ l'informazione sul figlio sinistro è in CHILD[$2i$+1] e quella sul figlio destro è in CHILD[$2i + 2$].

Definiamo ora due procedure:

1. *Rank*(CHILD,i) $\Rightarrow$ restituisce il numero di 1 presenti in CHILD nell'intervallo $[0, i]$
2. *Select*(CHILD,i) $\Rightarrow$ restituisce la posizione dell'$(i + 1)$-esimo 1 in CHILD

Date queste due procedure si può dire che per un nodo x=NODE[i] valgono le seguenti **proprietà**:

- Il figlio sinistro è dato da NODE[*Rank*(CHILD,$2i + 1$)-1] se CHILD[$2i + 1$]$\neq 0$
- Il figlio destro è dato da NODE[*Rank*(CHILD,$2i + 2$)-1] se CHILD[$2i + 2$]$\neq 0$
- Il padre è dato da NODE[(*Select*(CHILD,i)-1)$\cdot\frac{1}{2}$] se $i \neq 0$

Mostriamo ora come è possibile ottenere i risultati delle due procedure[4] in tempo costante, rimanendo in limiti di spazio ragionevoli.

Il metodo più efficace è quello di pre-calcore il *Rank* a blocchi di grandezza k, creando un nuovo array R_1, al posto dell'array CHILD, composto da esattamente $\frac{m}{k}$ ranghi, dove $m = 2n + 1$. Si noti che se $k = \frac{\log(m)}{2}$ lo spazio occupato dal nuovo array R_1 sarebbe di $2m$ bit.

[4]Vedremo soltanto la procedura *Rank*, la *Select* non è in programma.

In aggiunta viene mantenuta una tabella PREFIX formata da k righe e 2^k colonne dove vengono pre-calcolate (in colonna) le somme prefisse di tutte le possibili combinazioni di stringhe binarie di lunghezza k per poter risalire in tempo costante agli altri ranghi mancanti.

Ad esempio se $k = 3$ si ha:

0	0	0	0	1	1	1	1
0	0	1	1	1	1	2	2
0	1	1	2	1	2	2	3

Si noti che lo spazio occupato da PREFIX è $o(m)$ bit.

Un generico rango è quindi dato dalla relazione:

$$Rank(\text{CHILD},i)=R_1[q]+\text{PREFIX}[r,s]$$

dove $q = \left\lfloor \frac{i+1}{k} \right\rfloor$, $r = (i+1) - qk$ ed infine s che corrisponde alla codifica decimale (anch'essa pre-calcolata) del numero binario espresso dai k elementi nel sotto-array dell'elemento in posizione i. Si noti che il pre-calcolo di s comporta un ulteriore array da m bit.
Si ottiene quindi una rappresentazione che riesce ad effettuare le operazioni più frequenti (ricerca di figli e padre per un generico nodo) sempre in tempo costante ma che occupa uno spazio aggiuntivo, oltre ai bit dell'array NODE, di $3m + o(n)$ bit. Vediamo adesso come questo spazio aggiuntivo puo' essere portato a $m + o(m)$ bit quindi a $2n + o(n)$ bit.

Per prima cosa utilizziamo un intervallo di campionamento maggiore rispetto a k, usiamo $h = \frac{\log^2(m)}{8}$. D'ora in avanti i campionamenti avverranno ogni per blocchi più grandi ma saranno un numero minore rispetto a prima, infatti lo spazio occupato dall'array che li contiene, che chiameremo R_2, è pari a $o(m)$ bit.
Arrivati a questo punto, cambiamo il ruolo dell'array R_1. Tale array, d'ora in avanti, conterrà i campionamenti dei ranghi per blocchi di lunghezza k (proprio come prima) ma questa volta lo farà per porzioni dell'array CHILD di lunghezza h in maniera separata. In altre parole, è come se avessimo un piccolo array per ogni blocco di lunghezza h dell'array CHILD e ciascuno di questi piccoli array campiona separatamente un blocco di lunghezza h ogni k volte proprio come faceva R_1 nell'altra implementazione. Si noti che due diversi blocchi h sono da considerarsi separati e quindi il conteggio è **indipendente**. Vale a dire che il blocco successivo non si carica il conto del blocco precedente, il conteggio ripartirà da zero poichè ogni piccolo array è fine a se stesso.
Lo spazio occupato dal "nuovo" array R_1 (contentente tutti i piccoli array per intenderci) è pari a $\frac{m}{h} \cdot \frac{h}{k} \cdot log(h) = \frac{m}{k} \cdot \log(h)$ il quale è pari a $o(m)$.
Un generico rango è quindi dato da una relazione simile a quella precedente:

$$Rank(\text{CHILD},i)=R_1[q]+\text{PREFIX}[r,s]+R_2[t] \qquad \text{dove } t = \left\lfloor \frac{i+1}{h} \right\rfloor$$

Otteniamo così una rappresentazione che utilizza $m + o(m)$ bit aggiuntivi, per la precisione $2n + o(n)$ bit. Notando che il lower bound alla struttura è di $2n - O(\log n)$ bit si evince che il risultato raggiunto è pressoché ragionevole.

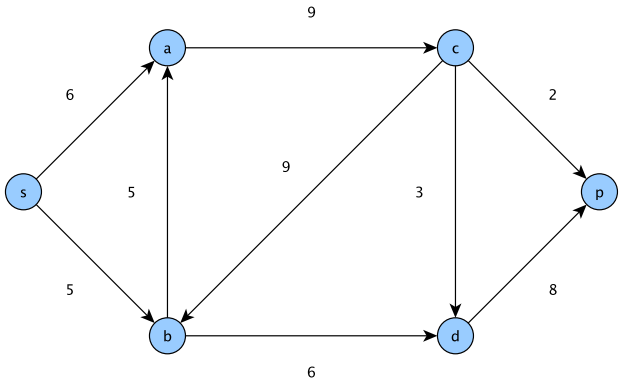
Capitolo 5

Reti di flusso

Una rete di flusso è un grafo orientato e pesato R=(V,E,s,p,C) in cui

- V è l'insieme dei vertici
- E è l'insieme degli archi
- s è un nodo che chiamiamo *sorgente,* da cui escono archi, ma non entrano
- p è un nodo che chiamiamo *pozzo,* da cui entrano archi, ma non escono
- C è una funzione che rappresenta i costi sugli archi C: $V^2 \rightarrow \mathbb{R}^+$
- $\forall v \in V$ $\exists$ un cammino dalla sorgente al pozzo che passa per v

Figura 5.1: Esempio di rete di flusso R

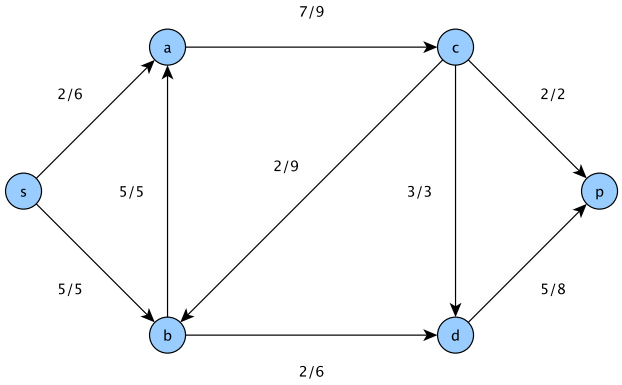


Un assegnamento di flusso è una funzione f:$V^2 \rightarrow \mathbb{R}^+ \cup \{0\}$ nella quale valgono i seguenti tre vincoli:

1. Vincolo di capacità: $f(u,v) \leq c(u,v)$ $\forall u, v \in V^2$
2. Vincolo antisimmetria: $f(u,v) = -f(v,u)$
3. Vincolo di conservazione del flusso: $\forall u \in V - \{s,p\}$, $\sum f(u,v) = 0$

Un assegnamento di flusso nella rete della Figura 5.1 è il seguente:

Figura 5.2: Assegnamento di flusso F su R



Nel caso in cui la rete di flusso abbia più sorgenti o più pozzi basta creare una sorgente fittizia S collegata alle sorgenti al quale assegno capacità maggiore di qualunque capacità della rete effettiva e un pozzo fittizio P a cui vengono collegati tutti i pozzi. Si vedrà che qualunque risultato per la rete globale sarà valido per la sottorete.

5.1 Rete residua

La rete residua è data dalla capacità residua di un singolo arco (rispetto ad un flusso), ovvero

$$r(\text{u,v}) = \text{C(u,v)-}f(u,v)$$

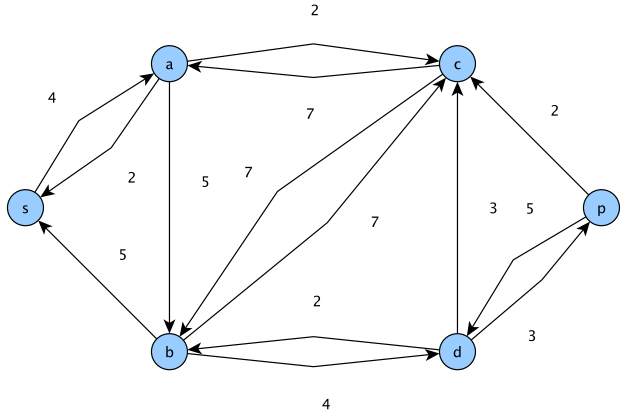
da cui per la simmetria si ottenere che

$$r(\text{v,u}) = \text{C(v,u)-}f(\text{v,u})=\text{C(v,u)}+f(\text{u,v})$$

La rete residua ha quindi lo stesso numero di vertici della rete normale, ma il doppio degli archi su cui, come peso, andremmo a mettere la capacità residua dell'arco su cui è passato un flusso. Gli archi saturati non verranno riportati nella rete.

Si nota che, quando il flusso è nullo, la rete residua è uguale alla rete di flusso di partenza, ma con gli archi direzionati nei versi opposti.

Figura 5.3: Rete residua su F



5.2 Cammino aumentante

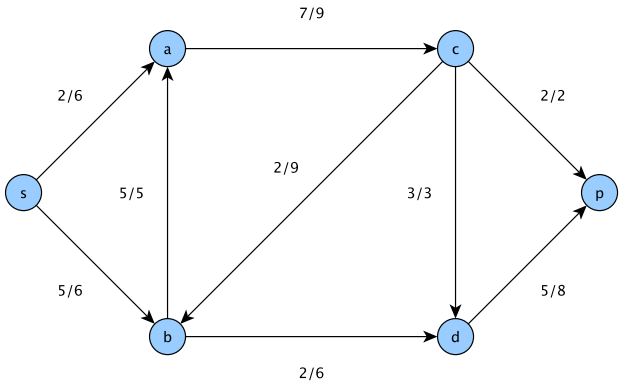
Si sa che $\forall v \in V$ $\exists$ un cammino dalla sorgente al pozzo che passa per v, allora prendendo un qualunque cammino dalla sorgente al pozzo si prende in considerazione la minima capacità degli archi di tutto il cammino.

Se il flusso nella rete residua che passa sul cammino è uguale alla minima capacità degli archi, allora il cammino non è aumentante.

Sulla Figura 5.2 si potrebbe scegliere il cammino *s-a-c-p* e la minima capacità è 2. Il flusso che passa su questo cammino è 2, quindi **non è un cammino aumentante.**

Se avessimo preso in considerazione la rete di flusso F' della Figura 5.4, allora *s-a-d-p* avrebbe avuto minima capacità 6 e il flusso che passa su questo cammino è 5 e quindi **è un cammino aumentante.**

Figura 5.4: Assegnamento di flusso F' rete di flusso R'



Quindi si può dire che, chiamando c_m la capacità minima sul cammino C, il flusso su quel cammino è tale che

$f(u, v) \geq c_m$ se $(u, v) \in C$ (se (u,v) è un arco del cammino) e se C è un cammino aumentante

$f(u, v) \geq -c_m$ se $(v, u) \in C$ e se C è un cammino aumentante

5.3 Taglio della rete

Il taglio della rete è una partizione dei vertici in due classi tale che in una classe ci sia la sorgente e nell'altra il pozzo.

Definiamo la *capacità del taglio {S,T}* come

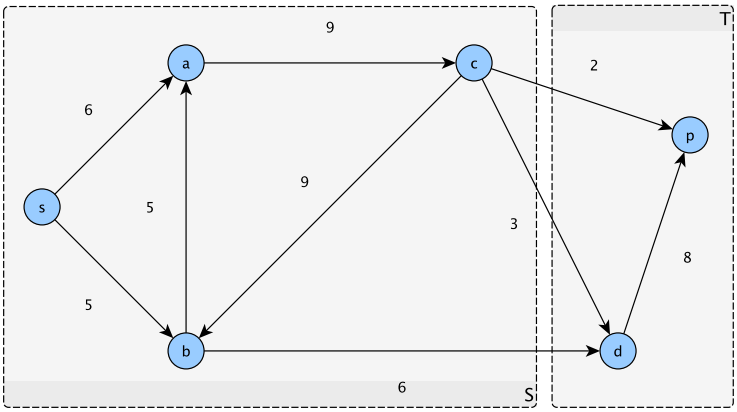
$$\sum_{u \in S, v \in T} c(u,v) = C(S,T)$$

e in maniera analoga il *flusso del taglio {S,T}* è pari a

$$F(S,T) = \sum u \in S, v \in T f(u,v)$$

Se prendessimo come esempio S={s,a,b,c} e T={d,p}, allora la capacità e il flusso di questo taglio sarebbero dati dagli archi {c,d},{b,d},{c,p}.

Figura 5.5: Esempio di taglio {S,T} su R



Teorema 1 *(Max.flow-min.cut)*

1. $\exists(S,P)$: C(S,P) = f
2. f è **flusso massimo** in R
3. $\nexists$ cammini aumentanti in G (o in R_f)

Dimostrazione:

1.→2.

$|f| \leq$C(S,P) $\forall$S,P, quindi se $|f|$=C(S,P) allora vorrà dire che f è massimo

2.→3.

banale. Se ci fosse un cammino aumentante in R per f, allora f non sarebbe massimo

3.→1.

Sia S l'insieme dei vertici raggiungibili da s nella rete residua Rf e P = V-S. Poiché non vi sono cammini aumentanti in R per f, in Rf non vi sono spigoli da S a P, ovvero in R tutti gli spigoli da S a P sono saturati, ovvero C(S,P) = f.

5.4 Metodo delle reti residue

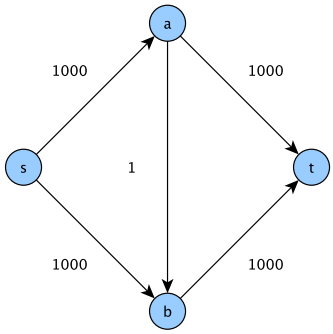
Inizialmente si considera associato alla rete R un flusso nullo, $f = f_0$

Data una rete R con assegnato un flusso f (non massimo), viene costruita una rete residua R_f, sulla quale deve poi essere trovato un flusso f'. Il flusso finale assegnato ad R sarà il flusso $f+f'$. Si procede così per iterazione finchè a R non è assegnato un flusso massimo, ovvero finchè esiste nel grafo della rete residua R_f un cammino $s \rightarrow t$.

Se le *capacità degli archi sono intere*, l'algoritmo trova sempre un flusso massimo in tempo finito e la velocità di convergenza del metodo dipende da come viene calcolato f'; invece se le *capacità degli archi sono reali*, l'algoritmo potrebbe avvicinarsi asintoticamente all'ottimo senza raggiungerlo mai.

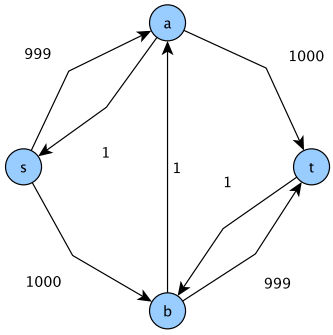
Prendiamo la rete di flusso R della Figura 5.6 e consideriamo il flusso iniziale pari a 0.

Figura 5.6: Rete di flusso R



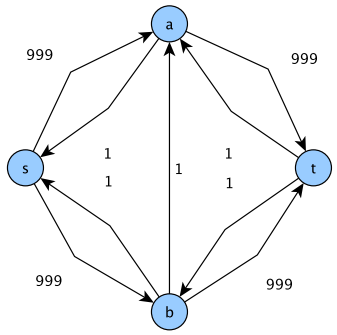
Se si andasse a scegliere il cammino aumentante *s-a-b-t* (dal valore 1) in R si otterrebbe la rete residua $R_f^{'}$ nella figura seguente

Figura 5.7: Rete residua $R_f^{'}$



su cui si può scegliere il cammino aumentante *s-b-a-t* (dal valore 1) producendo quindi la seguente rete residua $R_f^{''}$

Figura 5.8: Rete residua $R_f^{''}$



Iterando questo procedimento arriveremo a scegliere 1000 cammini aumentanti *s-a-b-t* (da 1) e 1000 cammini aumentanti *s-b-a-t* (da 1), quindi la ricerca del flusso massimo in questo caso non dipenderebbe nè da $|V|$ nè da $|E|$ e il tempo risulterebbe quindi esponenziale, in quanto si baserebbe solo sulle capacità degli archi.

Proposizione 5 *Se f su R e $f^{'}$ su R residuo sono flussi allora $f+f^{'}$ è un flusso su R e $|f+f^{'}| = |f|+|f^{'}|$.*

Per essere ancora un flusso devono valere i tre vincoli:

- CAPACITÀ: *(f+f')(xy) = f(xy)+f'(xy) ≤ f(xy)+((c(xy)-f(xy)) = c(xy)*
- ANTISIMMETRIA: *(f+f')(xy) = f(xy)+f'(xy)* $=^1$ *-f(xy)-f'(xy)*
- CONSERVAZIONE FLUSSO: $\sum$(f+f')(xy) = $\sum$(f(xy)+f'(xy)) = $\sum$f(xy)+$\sum$f'(x,y) $=^2$ 0+0 = 0

[1] poichè vale l'antisimmetria su f e f' per *hp*.
[2] poichè vale la conservazione del flusso su f e f' per *hp*.

5.5 Algoritmo di Ford-Fulkerson (1956)

L'algoritmo di Ford-Fulkerson è il primo algoritmo che permette di trovare il flusso massimo. Di seguito lo pseudocodice:

Algoritmo 5.1 Ford-Fulkerson

for each arco $(u,v) \in E$ *do*
f(u,v)=0
f(v,u)=0
while $\exists$ un cammino π da s a p in R_f *do*
cf (π) = $min.c_f(u,v)$ con $(u,v) \in \pi$ in Rf
for each arco (u,v) $\in \pi$ *do*
f(u,v) = f(u,v) + c_f (π)
f(v,u) = - f(u,v)

Il tempo di esecuzione dell'algoritmo è pari a O((|V|+|E|)·|f*|) poichè nel caso peggiore il flusso f è aumentato di un'unità e l'aumento di questa, sempre nel caso peggiore, può prevedere la visita dell'intera rete.

5.6 Algoritmo di Edmonds-Karp (1972)

Algoritmo 5.2 Edmonds-Karp

for each arco $(u,v) \in E$ *do*
f(u,v)=0
f(v,u)=0
while $\exists$ un cammino **minimo** π da s a p in R_f *do*
$c_f\ (\pi) = min.c_f(u,v)$ con $(u,v) \in \pi$ in R_f
for each arco $(u,v) \in \pi$ *do*
f(u,v) = f(u,v) + $c_f\ (\pi)$
f(v,u) = - f(u,v)

Per cammino minimo si intende *minimo rispetto alla distanza e non rispetto al costo del cammino.* Ad esempio nella Figura 5.2 il cammino minimo è *s-a-c-p* ed è di lunghezza 3, ma esistono altri cammini (non minimi) che portano da s a p, come ad esempio *s-a-c-b-d-p* di lunghezza 5.

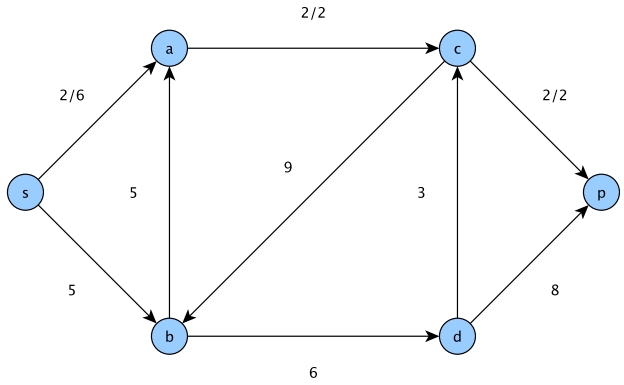
Usando ogni volta il cammino più breve per andare da s a p Edmonds e Karp limitano la complessita dell'algoritmo.

Lemma 3 *(delle distanze) Eseguendo l'algoritmo Edmonds-Karp su una rete di flusso di R (V,E,s,p,C) si ha che* $\forall v \in V - s, p$*, la distanza* $\delta_f(s,v)$ *in* R_f *cresce monotònamente ad ogni aumento di flusso.*

($\delta_f(u,v)$ è la lunghezza del cammino u$\rightsquigarrow$v in R_f)

Osservazione. Supponiamo di avere la rete di flusso F" in Figura 5.9 in cui è stato scelto il cammino aumentante *s-a-c-p* dal valore 2. La distanza da s a c è pari a 2. In questo caso si è saturato l'arco $\{a,c\}$ quando si passa la rete residua per andare da s a c ho un cammino di costo maggiore di 2 (*s-b-d-c*, di costo 3).

Figura 5.9: Esempio di rete di flusso F"



In generale se si ha un cammino $s \rightsquigarrow c$ **minimo**, del tipo $s \rightsquigarrow a\text{-}b \rightsquigarrow c$, chiamiamolo $\delta_f(s,c)$, dove si è saturato l'arco $\{a,b\}$, necessariamente bisognerà trovare un percorso alternativo $s \rightsquigarrow c$, chiamiamolo $\delta_{f'}(s,c)$, del tipo $s \rightsquigarrow b \rightsquigarrow c$ oppure $s \rightsquigarrow a \rightsquigarrow c$, il cui costo sarà necessariamente maggiore del cammino $s \rightsquigarrow a\text{-}b \rightsquigarrow c$ poichè era stato scelto in quanto minimo, quindi si ha che $\delta_f(s,c) \leq \delta_{f'}(s,c)$ e che quindi cresce monotònamente.

Dimostrazione. Siano $R_f = (V_f, E_f)$ e $R_{f'} = (V_{f'}, E_{f'})$ le reti residue ottenute rispettivamente prima e dopo l'aumento del flusso da f a f'. La dimostrazione viene fatta per assurdo supponendo che esista un insieme di vertici A nella rete di flusso tale che $\forall a \in A$ vale che $\delta_{f'}(s,a) < \delta_f(s,a)$.

Sia $v \in A$ il vertice che tra questi è il più vicino alla sorgente nella rete residua $R_{f'}$, quindi banalmente avremo che:

$$\delta_{f'}(s,v) = min\{\delta_{f'}(s,a)\}$$

Sia u il vertice che precede v nel cammino $s \rightsquigarrow v$ all'interno di $R_{f'}$. Osserviamo che $(u,v) \in E_{f'}$ e che $\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1$. Sappiamo inoltre, per come abbiamo scelto v, che $u \notin A$ e quindi vale che $\delta_f(s,u) \leq \delta_{f'}(s,u)$.
Mostriamo subito come, sotto l'ipotesi assurda, deve valere che $(u,v) \notin E_f$. Se infatti $(u,v) \in E_f$ si avrebbe necessariamente:

$$\delta_f(s,v) \leq \delta_f(s,u) + 1 \leq \delta_{f'}(s,u) + 1 = \delta_{f'}(s,v) \Rightarrow \delta_f(s,v) \leq \delta_{f'}(s,v)$$

Arrivando così ad una contraddizione.
Ma allora in che caso possiamo avere che $(u,v) \in E_{f'}$ ed $(u,v) \notin E_f$? Solo quando viene aumentato il flusso sull'arco $(v,u) \in E_f$. Questo accade quando il cammino minimo scelto dell'algoritmo Edmonds-Karp, prima dell'aumento del flusso, contiene l'arco (v,u).
Segue quindi che:

$$\delta_f(s,v) = \delta_f(s,u) - 1 \leq \delta_{f'}(s,u) - 1 = \delta_{f'}(s,v) - 2 \Rightarrow \delta_f(s,v) < \delta_{f'}(s,v)$$

Anche in questo caso si arriva a contraddire l'ipotesi, la quale, infatti, non è corretta. □

5.6.1 Complessità dell'algoritmo di Edmonds-Karp

Il tempo d'esecuzione è pari a $O(|V| \cdot |E|^2)$ poichè

1. Ogni iterazione viene fatta con una visita (Edmonds e Karp usano la visita in ampiezza), quindi $O(|V| + |E|)$ che nel caso pessimo equivale a $O(|E|)$.
2. L'algoritmo esegue $O(|V| \cdot |E|)$ iterazioni

La seconda affermazione è dovuta al fatto che su ogni cammino aumentante c'è un arco saturato o critico che sparisce dalla rete residua dopo che è stato aumentato il flusso nella rete. Un arco critico (u,v) può ritornare critico solo quando l'arco (v,u) appare su un cammino aumentante e quindi quando il flusso su (u,v) decresce.

La prima volta che diventa critico si ha che $\delta_f(s,v) = \delta_f(s,u) + 1$ e l'arco (u,v), mentre la seconda volta si ha che $\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1$.

Per il Lemma 3 (*delle distanze*) si ha che $\delta_f(s,v) \leq \delta_{f'}(s,v)$ da cui

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 \geq \delta_f(s,v) + 1 = \delta_f(s,u) + 2$$

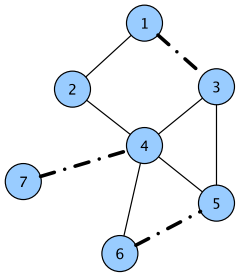
Quindi fra i due momenti di criticità la distanza di u dalla sorgente aumenta di almeno 2.
Tale distanza può variare da 0 a $|V|$-2 e quindi O(|V|) volte per ogni nodo (u,v), da cui $O(|V| \cdot |E|)$.□

Capitolo 6

Il problema degli abbinamenti

Consideriamo un grafo $G = (V, E)$ non orientato, connesso e senza cappi. Un'**abbinamento** (o **matching**) su $G = (V, E)$ è un'insieme di archi $M \subseteq E$ a due a due non incidenti. Ricordiamo che un arco $e = \{x, y\}$ è incidente ad un'altro arco $e' = \{x', y'\}$ se e solo se vale che $x = x' \vee y = y'$.
La figura sottostante mostra un esempio di matching M su un grafo G.

Figura 6.1: Esempio di matching



Definizioni utili con esempi:

- *Matching Massimale*: un matching che non è contenuto propriamente in nessun altro matching del grafo G, il matching dell'esempio è massimale.
- *Matching Massimo*: L'accoppiamento del grafo G che ha la cardinalità massima rispetto a tutti gli altri accopiamenti costruibili in G. Il matching dell'esempio è anche massimo.
- *Vertice saturo*: Un vertice che fa parte di un'arco del matching. Esempio: $\{1, 3, 4, 5, 6, 7\}$.
- *Vertice esposto*: Un vertice che non fa parte di un'arco del matching. Esempio: $\{2\}$.
- *Matching perfetto*: Accoppiamento che non lascia vertici esposti. Il matching dell'esempio non è perfetto poichè lascia un vertice esposto, il vertice 2.

Proprietà:

- Se un matching è massimo allora è massimale ma non è vero il viceversa. (Controesempio banale)
- Un matching perfetto è massimo ma non è vero il viceversa. Un controesempio è fornito nella figura precedente.
- Matching massimi o matching perfetti non è detto che siano unici.

- $|M_{max}| \leq \left\lfloor \frac{|V|}{2} \right\rfloor$

- Se M_{max} è perfetto allora G ha un numero pari di nodi poichè $|M_{max}| = \frac{|V|}{2}$.

Definiamo adesso il concetto di **cammino alternante** e **cammino aumentante** per un matching M:

- Un cammino *alternante* per un matching M in un grafo G è un cammino che alterna archi di M e archi non di M.

- Un cammino *aumentante* per un matching M in un grafo G è un cammino alternante per M che inizia e termina con un nodo esposto.

Definiamo inoltre l'**operazione di XOR** tra due insiemi di archi X e Y come $X \oplus Y = (X \cup Y) - (X \cap Y)$. Quello che si riesce a dimostrare è:

1. Se M è un matching su G e p è un cammino aumentante per M allora vale che $M' = M \oplus p$ è un matching su G e $|M'| = |M| + 1$.

2. M è un matching massimo su G se e solo se in G non sono presenti cammini aumentanti per M.

3. Se M ed M' sono due matching su G tali che $|M| < |M'|$ allora vale che in G vi sono almeno $|M'| - |M|$ cammini aumentanti per M.

Dimostrazione 1: Effettuiamo lo XOR tra gli insiemi M e p e sia M' l'insieme così ottenuto. Osserviamo che: gli archi del matching M che non facevano parte di p rimangono inalterati, dunque si ritrovano in M'; invece gli archi di M che fanno parte anche di p vengono esclusi da M' ma vengono però inseriti in M' gli archi di p che non facevano parte di M. L'insieme M' è ancora un matching su G, in quanto: per i nodi che non sono in p non è cambiato nulla, su quelli intermedi di p prima incideva un arco di M e adesso un arco di M', i nodi estremi di p erano esposti e ora sono invece accoppiati con archi di M'. Poichè p è un cammino aumentante rispetto a M esso avrà lunghezza pari a $2k + 1$ e conterrà necessariamente k archi del matching M e $k + 1$ archi liberi. Facendone lo XOR con M, gli archi liberi entrano a far parte del matching M' potendo quindi affermare che M ha un'arco in meno rispetto ad M'. □

Dimostrazione 2: Il "solo se" è ovvio per quanto appena dimostrato per 1. Vediamo allora di dimostrare che se non esistono cammini aumentanti rispetto a un certo matching M, allora quel matching M è massimo. Supponiamo per assurdo che pur non esistendo cammini aumentanti per M, M non sia massimo e che quindi esista un matching M' in G tale che $|M'| > |M|$. Consideriamo ora il grafo $G' = (V, E')$ con $E' = M' \oplus M$. Andiamo ad analizzare i gradi dei nodi di G', considerando tutti i casi possibili.

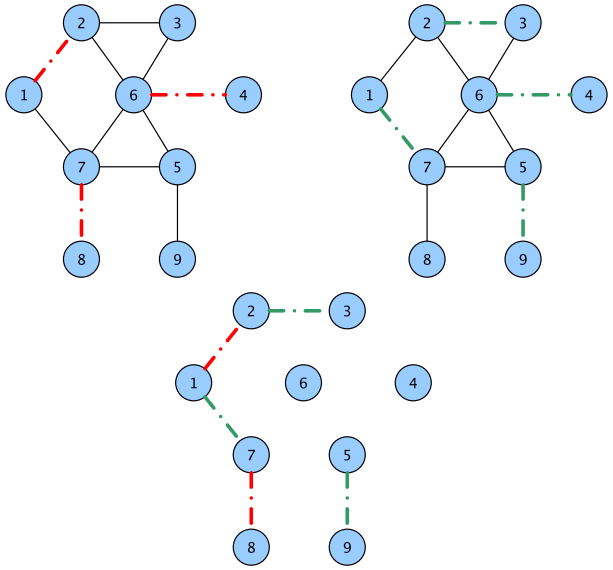
1. Un nodo che in ambedue i matching M e M' è accoppiato con lo stesso vicino è un nodo isolato in G'.

2. Un nodo che in ambedue i matching M e M' è accoppiato, ma con vicini diversi, ha grado 2 in G'.

3. Un nodo accoppiato in M ed esposto in M' o viceversa, in G' ha grado 1.

4. Un nodo esposto in ambedue i matching, in G' è un nodo isolato.

Figura 6.2: Esempio: in alto i due matching M ed M' su G, in basso il grafo G'



Dunque in G' nessun nodo ha grado superiore a 2 (come in Figura 6.2). Ecco che allora le componenti connesse di G' possono essere solo di tre tipi:

1. Nodi isolati
2. Cammini, costituiti alternatamente da archi di M e archi di M'
3. Cicli pari, in quanto se un ciclo avesse un numero dispari di archi ci sarebbero due archi dello stesso matching incidenti uno stesso nodo, il che è impossibile.

Ora, ricordando che stiamo supponendo $|M'| > |M|$, deve risultare che in E' vi siano più archi di M' che archi di M. Ma tali archi non possono trovarsi ovviamente nei cicli pari, dal momento che questi sono costituiti dallo stesso numero di archi di M' e di M. Quindi, deve esistere almeno un cammino costituito da un numero di archi di M' superiore al numero di archi di M. Ma affinchè questo accada tale cammino deve necessariamente iniziare e terminare con archi di M' e quindi essere un cammino aumentante per M in G. Avendo così contraddetto l'ipotesi secondo la quale non esistono cammini aumentanti in G, segue la tesi. □

Dimostrazione 3: Segue da quanto dimostrato per 1 e 2. □

6.1 Abbinamenti su grafi bipartiti

Un grafo $G = (V, E)$ si dice bipartito se e solo se valgono le seguenti proprietà:

1. $V = X \cup Y$ con $X \cap Y = \emptyset$ (Unione disgiunta)
2. Per ogni $e = \{x, y\} \in E$ si ha che $x \in X$ e $y \in Y$

Si può usare più comodamente la notazione $G = (X \sqcup Y, E)$ dove il simbolo "$\sqcup$" indica l'unione disgiunta. Notiamo subito che:

1. Se $G = (X \sqcup Y, E)$ allora $|M_{max}| \leq min\{|X|, |Y|\}$
2. Se $|X| \neq |Y|$ allora G non ha un matching perfetto

Nel 1890 Jacobi dimostrò[1], attraverso il teorema dei matrimoni, che se $G = (X \sqcup Y, E)$ (è bipartito) allora esso ha matching perfetto se e solo se valgono le seguenti:

1. $|X| = |Y|$
2. Per ogni $S \subseteq X$ si deve avere che $|S| \leq |\Gamma(S)|$ dove con il simbolo "Γ" si intende il vicinato dell'insieme S.

Successivamente (1930-1931) Konig ed Egervary dimostrarono[2] che la cardinalità massima di un abbinamento è pari alla cardinalità minima di un ricoprimento (vertex-cover).
Dopo questa breve digressione storica, definiamo il concetto di *albero ungherese* introdotto da Kuhn nel 1955.
Sia $G = (X \sqcup Y, E)$ un grafo bipartito, M un matching arbitrario in G, r un vertice esposto in G. Costruiamo l'albero ungherese T radicato in r (ovvero un albero in cui ogni cammino dalla radice ad una foglia è alternante) nella seguente maniera:

- Partendo da r, si aggiunga a T ogni arco incidente ad r, siano essi $\{r, x_i\}, ..., \{r, x_n\}$. Se c'è anche solo un nodo x_i esposto ci si ferma (cammino aumentante). Altrimenti si aggiunga a T l'unico arco di M incidente ad ogni x_i.
- Si ripeta la procedura, in maniera iterativa, a partire da ogni adiacente di ogni nodo x_i finchè: o si trova un adiacente esposto per uno di questi ultimi vertici oppure l'albero non puo' più crescere (albero ungherese).

Siamo ora pronti ad illustrare l'algoritmo per la ricerca del matching massimo in grafi bipartiti.

Algoritmo 6.1 MaxMatchingBipartiteGraph($G = (X \sqcup Y, E), M$)

while esistono due vertici esposti in G (uno in X ed uno in Y) **do**
  Costruisci T radicato $r \in X$ (con r nodo esposto) tramite una BFS
  if T è ungherese elimina T da G (sia archi che nodi)
  else esegui $M = M \oplus p$ dove p è il cammino aumentante trovato durante la costruzione di T ed aggiorna oppurtunamente i vertici saturi in G

Si noti che questa procedura funziona solo per grafi bipartiti poichè tali grafi non contengono cicli dispari. Durante la costruzione di T infatti, archi che collegano nodi allo stesso livello dell'albero non potranno esistere, poichè T è alternante e quindi un arco che collega due cammini alternanti di stessa lunghezza comporta l'esistenza di un ciclo dispari nel grafo.
La sua complessità è data, nel caso peggiore, da $O(|V|)$ BFS quindi $O(|E| \cdot |V|)$.
Nella prossima sezione vedremo la relazione che c'è tra il matching massimo nei grafi bipartiti ed il flusso massimo in una rete di flusso.

[1] Dimostrazione non in programma
[2] Dimostrazione non in programma

6.2 Abbinamento massimo in un grafo bipartito e reti di flusso

A partire dal grafo bipartito B=($L \cup R$, E), si costruisca la rete di flusso G=(V, E', s, p, C) così definita:

- V=L$\cup R \cup \{s,p\}$, con s e p che rappresentano la sorgente e il pozzo da aggiungere al bipartito B per creare la rete di flusso
- E' = E $\cup \{(s,u),\ u \in$ L$\} \cup \{(v,p),\ v \in$ R$\}$
- C è una funzione che associa ad ogni arco $(u, v) \in E'$ una capacità pari ad 1.[3]

Figura 6.3: Esempio di B

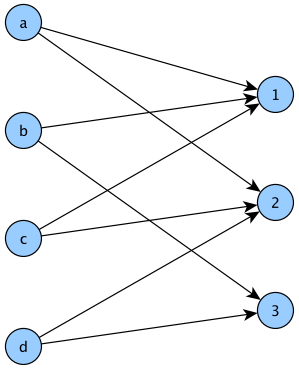
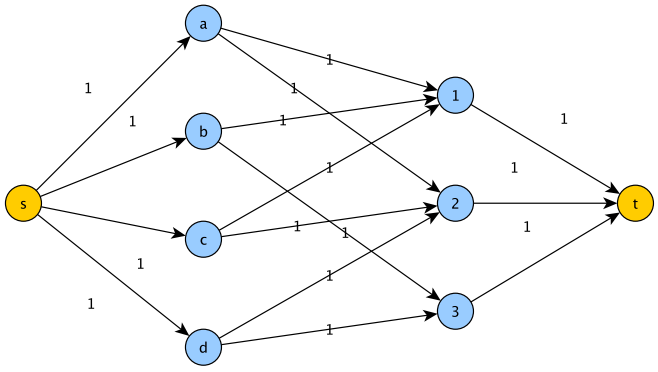


Figura 6.4: Costruzione di G a partir da B



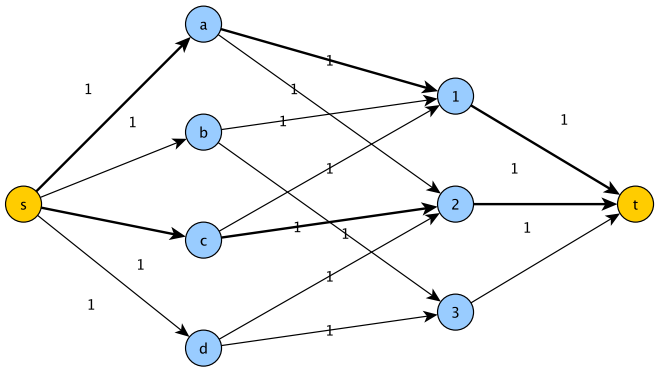
Lemma 4 *Siano dati B e G come sopra definiti. Per ogni M, abbinamento in B, esiste un flusso f a valori interi in G con valore $|f| = |M|$. In modo analogo vale che per ogni flusso a valori interi in G esiste un abbinamento M in B con cardinalità $|M| = |f|$.*

Dimostrazione (*Un flusso a valori interi in G corrisponde ad un abbinamento M in B*). Dato un flusso f a valori interi in G vale la conservazione del flusso sul nodo e visto che, per costruzione, si ha che tutti gli archi a capacità 1 non ci può essere più di un arco del flusso incidente a quel nodo. Preso dunque un qualsiasi arco che appartiene al flusso esso sicuramente appartiene all'abbinamento per quanto detto.

Inoltre si ha che $|M|=|f|$ poichè $\forall\ u \in$ L, $f(s,u)$=1 e $\forall\ (u,v) \in E \setminus M, f(u,v)$=0, da cui: $|M|=f(L,R)=|f|$.

[3]ricordiamo che il bipartito B non è pesato.

Figura 6.5: Esempio di assegnamento di flusso f su G



Dimostrazione (Un abbinamento M in B corrisponde ad un flusso a valori interi in G). Dato un abbinamento M in B se un arco $(u, v) \notin M$ allora $f(u,v)=0$ e se $(u, v) \in M$ allora $f(s,u)=f(u,v)=f(v,p)=1$ per costruzione, ovvero qualunque arco dell'abbinamento corrisponde ad un cammino dalla sorgente s al pozzo p che passa per (u,v) che è un non cammino aumentante della rete che stiamo prendendo in considerazione. Si ha inoltre che i cammini indotti dagli archi in M hanno vertici disgiunti, a meno di s e p e il valore del flusso attraverso il taglio $\{(L \cup \{s\}, R \cup \{p\})$ coincide con il valore dell'abbinamento sul grafo bipartito ed è uguale a $|M|$.

Rimane da vedere che il flusso f associato alla rete sia effettivamente un flusso. Basta vedere che si rispettano le tre proprietà:

- CAPACITÀ: sugli archi viene assegnato o 0 o 1 per quanto già detto e quindi banalmente è verificato
- CONSERVAZIONE DEL FLUSSO: per ogni vertice entra ed esce un solo arco di valore 1, quindi banalmente il flusso entrante è pari al flusso uscende ed è uguale ad 1
- ANTISIMMETRIA: banalmente verificato visto che l'aumento del flusso è dovuto da singoli cammini disgiunti

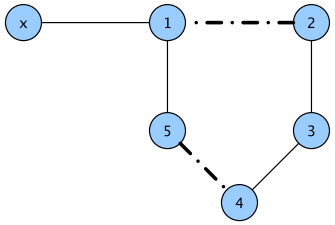
6.3 Abbinamento in un grafo generico

L'algoritmo per trovare il matching massimo in un grafo generico è dovuto ad Edmonds ed è stato ribattezzato *Paths,trees and flowers* (1965).

Definiamo ***germoglio*** un ciclo di lunghezza dispari costituito da un alternanza di archi appartenenti e non appartenenti al matching. In un germoglio si individua la *base* ed il suo *stelo*, ovvero l'arco collegato alla base del germoglio.

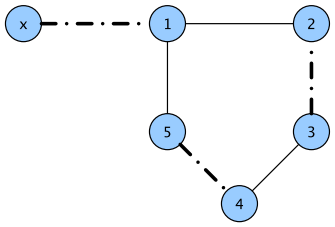
Nella Figura 6.6 lo stelo è rappresentato dall'arco $\{x, 1\}$, la base è rappresentata dal nodo 1, mentre gli archi del matching sono individuati da archi tratteggiati.

Figura 6.6: Esempio di germoglio H



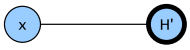
Si nota dalla Figura 6.6 che nel caso in cui venga "ruotata" la scelta degli archi del matching all'interno del germoglio è possibile aumentare il matching. Nel caso del germoglio mostrato nell'esempio precedente, il matching su di esso puo' essere aumentato da 2 a 3, come mostrato dalla Figura 6.7.

Figura 6.7: Aumento di matching nel germoglio H



Alla luce di ciò che è stato appena detto l'idea di Edmonds è quella di contrarre il germoglio H in un unico supernodo H' (Figura 6.8) da cui ne deriviamo il Teorema 2.

Figura 6.8: Aumento di matching nel germoglio H



Teorema 2 *Sia G(V,E) un grafo non orientato e sia G' il grafo ottenuto da G comprimendo un germoglio in un supernodo. G' contiene un cammino aumentante se e solo se G lo contiene.*

Algoritmo 6.2 *Paths,trees and flowers* (1965)

INPUT: $G = (V, E)$, M abbinamento arbitrario in G (M può essere l'insieme vuoto)
OUTPUT: abbinamento massimo su G ovvero M_{max}

finchè esistono almeno due vertici esposti, si costruisca un cammino alternante T partendo da un vertice esposto
  se si trova un *germoglio*, ovvero se il cammino alternante diviene un ciclo dispari che ritorna sul vertice iniziale, questo va *contratto* in un meganodo H e l'algoritmo prosegue sul nuovo grafo G' considerando H un vertice esposto a meno di una rotazione in G'
  se si trova p, *cammino aumentante*, si esegua $M = M \oplus p$ per incrementare la cardinalità di M
si *decontraggono* i nodi per produrre M_{max} su G

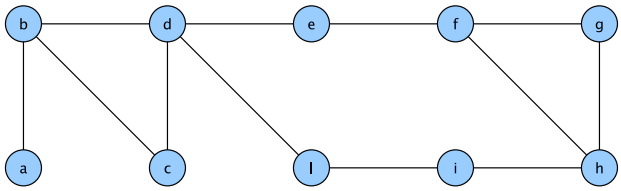
Il tempo d'esecuzione dell'algoritmo *Paths, trees and flowers* è pari a $O(|V| \cdot |E|)$ poichè:

1. Il ciclo *while* viene al più eseguito $|V|$ volte poichè almeno 1 vertice diventa non esposto ad ogni iterazione.

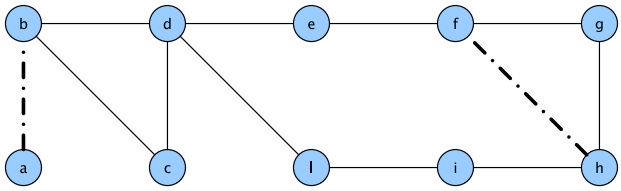
2. La ricerca di un germoglio o di un cammino aumentante costa il tempo di una visita, ovvero $O(|E|)$.

Vediamo ora un esempio di come opera l'algoritmo *Paths, trees and flowers* sul seguente grafo G:

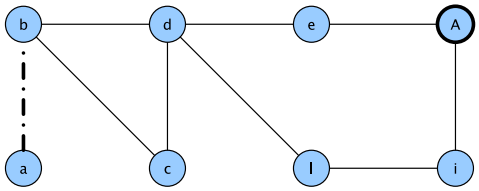
Figura 6.9: Esempio di esecuzione dell'algoritmo *Paths, trees and flowers* su un grafo G



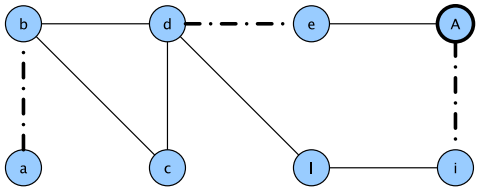
Cammini aumentanti $a \rightsquigarrow b$ e $f \rightsquigarrow h$



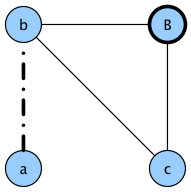
Germoglio $g \rightsquigarrow h \rightsquigarrow f \rightsquigarrow g$



Cammini aumentanti $d \rightsquigarrow e$ ed $a \rightsquigarrow i$



Germoglio $l \rightsquigarrow i \rightsquigarrow A \rightsquigarrow e \rightsquigarrow d \rightsquigarrow l$



Cammino aumentante in $c \rightsquigarrow B$

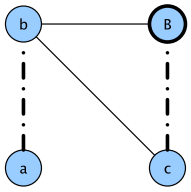
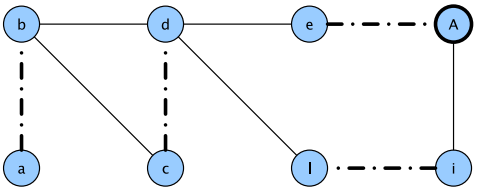
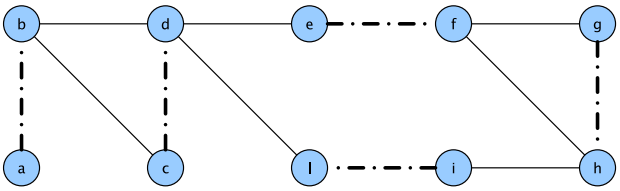


Figura 6.10: Esplosione dei germogli



Esplosione germoglio B ed aggiustamento degli archi del matching in B



Esplosione germoglio A ed aggiustamento degli archi del matching in A

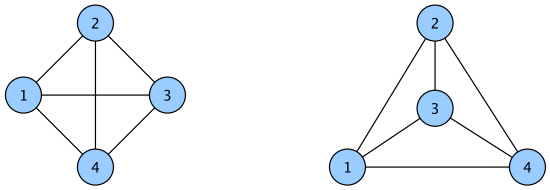
Dagli esempi nelle figure 6.9 e 6.10 si nota che $|M_{max}| = 5$.

Capitolo 7

Grafi planari

Dicesi grafo planare, un grafo che ammette un disegno senza intersezioni di archi. Si osservi la seguente figura.

Figura 7.1: Esempio di planarizzazione di un grafo



Dimostriamo subito una proprietà dei grafi planari ricavata da Eulero nel 1750.

Proposizione 6 *Sia un grafo planare connesso, non orientato e non pesato con n vertici, m archi ed f facce allora vale che:*

$$n - m + f = 2$$

Dove per faccia interna di un grafo planare si intende la regione del piano minimale interamente contornata dagli archi, inoltre è presente una faccia esterna la quale è la regione del piano esterna al grafo planare.

Dimostrazione. Dimostriamolo per induzione su m.

Passo base per $m = 0$: Il grafo è composto da un solo nodo percui vale che $n = 1$ ed $f = 1$ essendoci solo la faccia esterna. Quindi vale che $1 - 0 + 1 = 2$.

Passo base per $m = 1$: Il grafo è composto da soltanto due vertici collegati tra loro quindi si ha che $n = 2$ ed $f = 1$. Ovviamente $2 - 1 + 1 = 2$.

Passo induttivo:

Ipotesi induttiva: la proprietà è vera per $m - 1$.
Tesi induttiva: dimostriamolo anche per m.

Consideriamo dapprima il caso in cui il grafo G sia un albero. Rimuoviamo una foglia da tale albero ed otteniamo che: $n-1$ vertici, $m-1$ archi ed 1 faccia. Per ipotesi induttiva sappiamo che: $n-1-m+1+1 = 2$ ma allora vale anche che $n - m + 1 = 2$ (ricordando che $f = 1$ sempre in un albero).

Se G non è un albero allora ha necessariamente un ciclo (poichè connesso) e quindi togliamo un arco da tale ciclo. Quello che otteniamo è n vertici, $m-1$ archi ed $f-1$ facce. Per ipotesi induttiva si ha che $n-m+1+f-1=2$ e quindi vale anche che $n-m+f=2$. □

Osserviamo adesso che nei grafi planari è possibile porre delle **limitazioni superiori sia ad m che ad f** ovvero:

Sia G un grafo planare connesso, non orientato e non pesato con $n \geq 3$ vertici allora vale che

$$m \leq 3n-6$$

ed

$$f \leq 2n-4.$$

In un grafo planare massimale, ovvero in un grafo tale che l'aggiunta di un arco renderebbe il grafo non planare (è facile pensare che un grafo planare massimale è formato da tutti triangoli), si ha che $3f=2m$, quindi vale in generale per grafi planari anche non massimali che $3f \leq 2m$. Dalla formula di Eulero abbiamo che:

$$2 = n-m+f \leq n-m+\frac{2m}{3} = n-\frac{m}{3}$$

$$2 \leq n-\frac{m}{3}$$

$$m \leq 3n-6$$

Analogamente:

$$2 = n-m+f \leq n-\frac{3f}{2}+f = n-\frac{f}{2}$$

$$2 \leq n-\frac{f}{2}$$

$$f \leq 2n-4$$

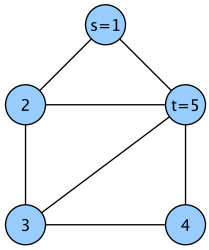
Prima di passare all'algoritmo che stabilisce se un grafo è planare o meno bisogna introdurre il concetto di $s-t$ numerazione (Even-Tarjan 1976).

Definizione 8 *Chiamiamo (s-t)-numerazione una numerazione da 1 ad n dei vertici di un grafo che tale che rispetti le seguenti proprietà:*

1. I nodi con numerazione 1 ed n (rispettivamente chiamati nodi s,t) devono essere adiacenti
2. Ogni nodo numerato $j \in [n]$ con $j \neq 1,n$ deve essere adiacente ad almeno un nodo numerato i con $i<j$ ed almeno un nodo numerato k con $k>j$

Si veda l'esempio della figura:

Figura 7.2: Grafo s-t numerato



7.1 Algoritmo Even-Tarjan

Even e Tarjan dimostrano in maniera costruttiva (fornendo un algoritmo) la seguente affermazione:

Teorema 3 *Un grafo che non ha punti di articolazione, ovvero è 2-connesso (per brevità G_{2C}), ammette sempre una numerazione dei suoi vertici.*

Even e Tarjan si limitano a farlo solo per questa tipologia di grafi poichè era stato già dimostrato che solo tali grafi ammettono una $s-t$ numerazione dei propri vertici[1].

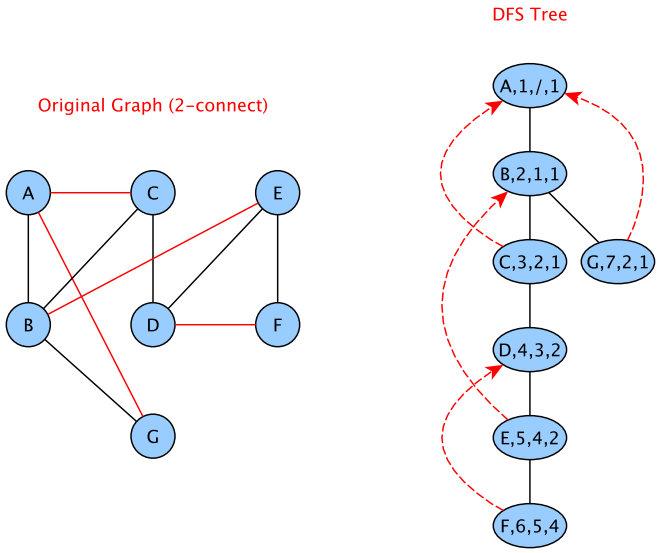
7.1.1 Inizializzazione

Prima di fornire l'algoritmo dobbiamo definire alcune funzioni utili per la sua comprensione.

1. DFN(v) è il numero dell'ordine in cui viene effettuata la DFS sul vertice v del grafo
2. FATH(v) è il padre del vertice v
3. LOW(v) = min(DFN(v),DFN(w)) per ogni w estremo dello spigolo di riporto $\{u, w\}$ con u discendente di v e w ascendente v nell'albero T_{DFS}[2].

La figura sottostante raffigura i tre concetti espressi precedentemente, indicando rispettivamente per ogni vertice v dell'albero della visita in profondità DFN(v), FATH(v) e LOW(v).

Figura 7.3: Esempio di inizializzazione dell'algoritmo Even-Tarjan



[1] Dimostrazione non in programma.
[2] Si noti che v è sia discendente che ascendente di se stesso.

Di seguito la procedura di inizializzazione in termini di pseudo-codice.

Algoritmo 7.1 INITIALIZE($G_{2c} = (V, E)$)

$T_{DFS} = \emptyset$; $count = 1$
for $i = 1$ to $|V|$ **do**
  mark v_i as not visited
Search(v_1)

Procedure Search(v)
  mark v as visited
  DFN(v)= $count$; $count = count + 1$; LOW(v)=DFN(v)
  for each $w \in Adj(v)$ **do**
    if w is not visited **then**
      $T_{DFS=}T_{DFS} \cup \{v, w\}$; FATH($w$)= v; **Search(**w**)**
      LOW(v)=min(LOW(v),LOW(w))
    else if FATH(v)$\neq w$ **then**
      LOW(v)= min(LOW(v),DFN(w))

7.1.2 Funzione *PATH*

Una volta inizializzato l'algoritmo, mediante FATH, DFN e LOW, conosciamo l'albero T_{DFS} costruito dalla visita in profondità. Per assicurarci la proprietà 1 dell'$s-t$ numerazione identifichiamo come t la radice di T_{DFS} e come s uno qualsiasi dei suoi figli il cui arco che li collega è presente in tale albero.
La funzione PATH associa l'etichetta *visitato* all'arco $\{s,t\}$ ed ai nodi s e t, mentre tutti gli altri nodi ed archi risultano inizialmente essere etichettati come *nuovi*.
A questo punto possiamo distinguere 4 diversi casi della funzione PATH(v), la quale prende come argomento un nodo v marcato come *visitato*:

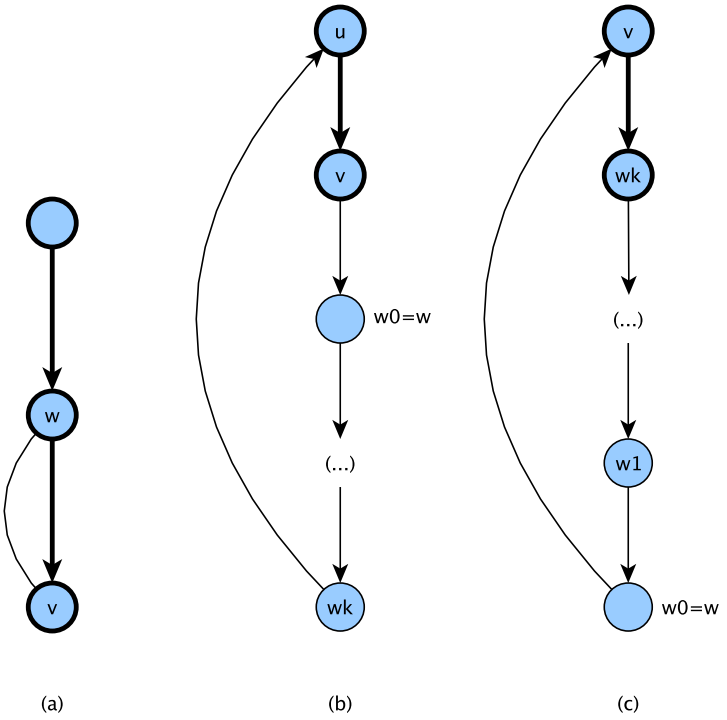
1. Esiste un arco di riporto non *visitato* $\{v,w\}$ (Figura 7.4a) e che esista l'arco $\{w,v\}$ su T_{DFS}; in questo caso viene marcato come *visitato* l'arco *nuovo* $\{v,w\}$ (poichè v e w già erano *visitati*) e PATH(v) restituisce v,w.

2. Esiste un arco dell'albero non *visitato* $\{v,w\}$ (Figura 7.4b); in questo caso PATH(v) si basa sulla funzione LOW: sia $w_0 = w, w_1, ..., w_k, u$ il cammino che percorre l'albero e termina con un arco all'indietro in un nodo u tale che LOW(w_k) = DFN(u) (il nodo più profondo che punta più in alto di tutti). Allora PATH(v) restituisce $v, w = w_0, w_1, ..., w_k, u$ e vengono marcati come *visitati* tutti i nodi e gli archi *nuovi* in tale cammino.

3. Esiste un arco di riporto non *visitato* $\{v,w\}$ per cui DFN(w) > DFN(v) (Figura 7.4c), che non esista l'arco $\{w,v\}$ su T_{DFS} e supponiamo che $w_0 = w, w_1, ..., w_k$ sia il cammino all'indietro che risale sugli archi dell'albero seguendo la funzione FATH, da w ad un nodo marcato. In questo caso PATH(v) restituisce $v, w_0 = w, w_1, ..., w_k$ e vengono marcati come *visitati* tutti i nodi e gli archi *nuovi* in tale cammino.

4. Se tutti gli archi (sia dell'albero che di riporto) incidenti a v sono *visitati* allora PATH(v) restituisce $\emptyset$.

Figura 7.4: Casi della funzione PATH



Una volta definiti i casi della funzione PATH possiamo passare all'algoritmo per l'$s-t$ numerazione che opera in tempo $O(|E|)$:

Algoritmo 7.2 ST-NUMBERED($G_{2c} = (V, E)$)

INITIALIZE(G_{2c})
si marcano i nodi s, t e l'arco $\{s, t\}$ come *visitati* e tutti gli altri vertici ed archi come *nuovi*
si inseriscono t ed s nella pila P, prima t poi s
count =1

Pop v da P
while $v \neq t$ **do**
  if PATH(v) = Ø then
    STNUM[v]=*count*
    count++
  else
    Push in P tutti i vertici di PATH in ordine inverso a v (v deve rimanere in testa)
  Pop v da P
STNUM[t]=*count*

7.2 Rappresentazione a Cespuglio (Bush form)

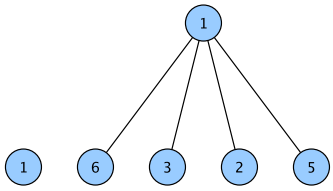
Dato un grafo G su cui è stata calcolata l'$s-t$ numerazione dei vertici, mediante l'Algoritmo 7.2 viene definito il grafo $G_k = (V_k, E_k)$, il quale è un sottografo di G indotto dai primi k vertici dell'$s-t$ numerazione.
Se $k < n$ esiste, per la Definizione 8, almeno un arco $\{x, y\}$ con $x \in V_k$ e $y \in V - V_k$.
Grazie a questa proprietà sarà possibile costruire un grafo G'_k, a partire dal grafo G_k, incrementandone ognuno dei k vertici con gli archi che rispettano la condizione sopracitata creando una rappresentazione detta "*a cespuglio*".

Definizione 9 *Una rappresentazione a cespuglio di G'_k è un embedding di G'_k con tutti i vertici virtuali posizionati sulla faccia esterna (usualmente sulla stessa orizzontale).*

Prendiamo in considerazione il grafo G $s-t$ numerato della Figura 7.10 e costruiamo $\forall k$ i grafi G'_k a partire dal grafo indotto G_k:

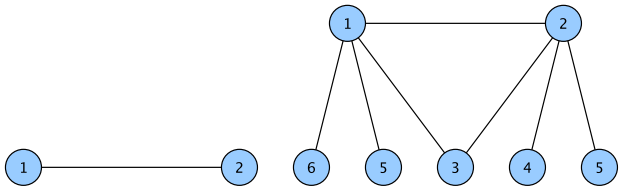
k=1 Il grafo G_1 è banalmente il solo nodo s e il grafo G'_1 è costruito aumentando il solo nodo s dai nodi adiacenti ad s (in questo caso specifico i nodi 2,3,5,6). In questo caso semplice i nodi aggiunti fanno parte della faccia esterna poichè c'è solo la faccia esterna in G'_1.

Figura 7.5: grafi G_1 e G'_1



k=2 Il grafo G_2 è fatto a partire dall'indotto su V_2 e alla stessa maniera vengono aggiunti gli archi che collegano un nodo in V_2 e un nodo in $V - V_2$, quindi:

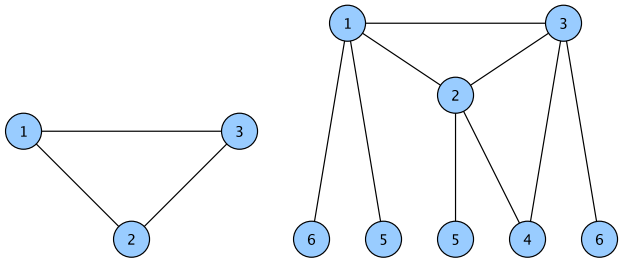
Figura 7.6: grafi G_2 e G'_2



il nodo 3, essendo adiacente sia ad 1 che a 2, si riunisce in un solo nodo, cosa che non avviene per 5, poichè la successiva costruzione sarà del grafo G_3.

k=3 il grafo G_3 è fatto a partire dall'indotto su V_3, sottografo di G'_2, e alla stessa maniera vengono aggiunti gli archi che collegato un nodo in V_3 e un nodo in $V - V_3$, quindi:

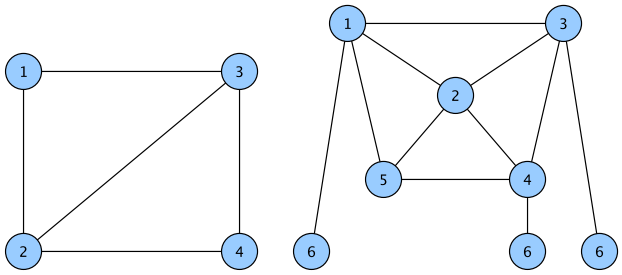
Figura 7.7: grafi G_3 e G'_3



il nodo 4, essendo adiacente sia ad 2 che a 3, si riunisce in un solo nodo poichè la successiva costruzione sarà del grafo G_4.

k=4 il grafo G_4 è fatto a partire dall'indotto su V_4, sottografo di G'_3, e alla stessa maniera vengono aggiunti gli archi che collegato un nodo in V_3 e un nodo in $V - V_4$, quindi:

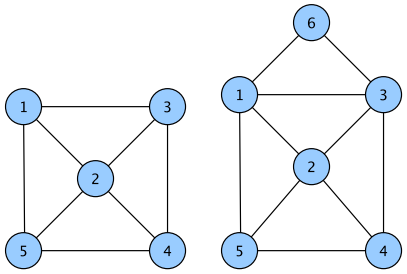
Figura 7.8: grafi G_4 e G'_4



il nodo 5, essendo adiacente a 1, 2 e 4, si riunisce in un solo nodo poichè la successiva costruzione sarà del grafo G_5.

k=5 il grafo G_5 è fatto a partire dall'indotto su V_5, sottografo di G'_4, e alla stessa maniera vengono aggiunti gli archi che collegato un nodo in V_5 e un nodo in $V - V_5$, quindi:

Figura 7.9: grafi G_4 e G'_4



In questo ed ultimo caso il grafo G'_5 è uguale al grafo G originale e planare.

Partendo dal presupposto che un qualsiasi grafo è planare se e solo se tutte le sue componenti 2-Connesse sono planari e che una componente 2-Connessa è planare se tutte le sue B_k (Bush form) con $1 \leq k < n$ sono planari, si ottiene un algoritmo per il test di planarità[3].
Questo algoritmo utilizza strutture dati complesse note come *PQ-tree* di cui non entreremo nel dettaglio per gli scopi di questo corso. Tuttavia il paragrafo 7.4 ne illustra una brevissima panoramica.

[3]Dimostrazioni non in programma.

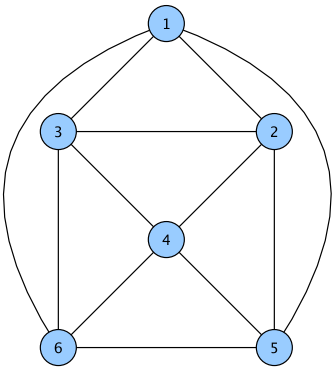
7.3 Embedding

Dato un grafo planare G è possibile salvare in memoria il grafo con liste di adiacenza, in maniera tale che la rappresentazione non venga alterata. Si ricorda che qualunque permutazione degli elementi di una singola lista degli adiacenti potrebbe variare la *rappresentazione grafica* del grafo stesso, ma non il grafo e le sue proprietà.
Per mantenere la sua rappresentazione grafica invariata, per ogni vertice del grafo planare vengono selezionati gli *archi incidenti in senso orario*[4] e viene salvata nella lista degli adiacenti il vertice associato all'arco.
Prendendo in considerazione la Figura 7.10 otteniamo la seguente lista di adiacenza:

1. 5236
2. 5431
3. 2461
4. 2563
5. 6421
6. 1345

dalla quale, riseguendo la regola con cui si sono salvati i nodi adiacenti, è possibile ridisegnare il grafo con la stessa sua precedente rappresentazione.

Figura 7.10: Grafo planare G



[4] Anche antiorario, non c'è differenza.

7.4 PQ-tree

Una forma a cespuglio B_k è normalmente rappresentata tramite una struttura dati chiamata PQ-tree. I nodi di un PQ-tree sono divisi in tre classi: P-nodi, Q-nodi e foglie, dove:

- I P-nodi rappresentano punti di articolazione di G_k e i figli di un P-nodo possono essere permutati arbitrariamente
- I Q-nodi rappresentano le componenti biconnesse di G_k e i figli di un Q-nodo possono essere solo scambiati
- Le foglie sono i nodi aggiunti in B_k e si muovono in accordo ai movimenti dei P-nodi (o dei Q-nodi)

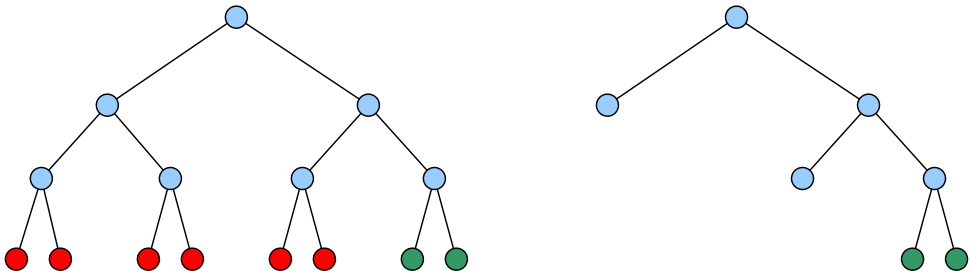
A scopo puramente informativo: Booth e Lueker hanno provato che le permutazioni e gli scambi nominati sopra possono essere trovati applicando ripetutamente 9 trasformazioni base.

Capitolo 8

Backtracking

La tecnica del backtracking è un miglioramento della ricerca esaustiva ed è spesso utile per problemi di natura esponenziale. Tale miglioramento si ottiene con l'inserimento di opportune funzioni, cosiddette di "taglio", le quali vengono scelte attraverso delle euristiche ed evitano alla macchina di proseguire su cammini inutili (ai fini del problema) all'interno dell'albero della sua computazione.
La seguente figura mostra quanto appena detto. L'albero di sinistra è un esempio di albero di computazione ottenuto generando tutti i possibili output (foglie) di un dato problema. Successivamente, su tale insieme, viene effettuata una scelta delle sole soluzioni. Con la tecnica del backtracking invece viene effettuato un controllo (funzione di taglio) per ogni possibile ramo uscente da ogni nodo. Se tale controllo non risulterà positivo la computazione non potrà più proseguire lungo tale ramo. Si noti che l'insieme di output *ottimo* ottenuto utilizzando la tecnica del backtracking è quello che coincide con l'insieme delle soluzioni al problema dato.

Figura 8.1: Ricerca esaustiva vs Backtracking



Nelle due seguenti sezioni vedremo come effettivamente utilizzare la tecnica del backtracking per risolvere due famosi problemi di natura esponenziale.

8.1 I sottoinsiemi di somma m

Il problema è quello di voler generare, a partire da un insieme di n interi positivi, **tutti i sottoinsiemi** di tale insieme la cui somma degli elementi è m. Asintoticamente il guadagno nella complessità temporale è nullo, ma vedremo che, in pratica, la macchina ha un enorme risparmio di tempo computazionale.
Iniziamo fornendo lo pseudo-codice dell'algoritmo di ricerca esaustiva e poi ci preoccuperemo di inserire le funzioni di taglio che individueranno le soluzioni del problema senza operazioni aggiuntive. Utilizziamo il vettore "*Sol*", di dimensione n, per la creazione del sottoinsieme[1] ed il vettore globale "*Elem*" per poter risalire al valore originale di un determinato elemento.

Algoritmo 8.1 Ricerca esaustiva

```
SOTTOINSIEMI(Sol, n, m, i)
  if (i>n) then
    if(CHECK(Sol, m)) then
      PRINT(Sol) and break
    else break
  Sol[i]=0; SOTTOINSIEMI(Sol, n, m, i + 1)
  Sol[i]=1; SOTTOINSIEMI(Sol, n, m, i + 1)

Procedure CHECK(Sol, m)
  sum=0
  for i=1 to n do
    if Sol[i]=1 then
      sum=sum+Elem[i]
  if (sum=m) then return true
  else return false
```

Vediamo adesso quali sono alcune euristiche che portano a delle notevoli funzioni di potatura:

1. Se un elemento i-esimo del vettore "Sol" permette alla somma totale dei suoi elementi di superare la soglia m, allora sarebbe inutile proseguire a generare sottoinsiemi che contengono tali elementi.

2. Se ad un certo passo i-esimo la somma degli elementi del vettore "Sol" non arriva ad m nemmeno se venissero inclusi tutti gli altri elementi rimanenti non ancora presi, allora sarebbe inutile continuare a generare sottoinsiemi che contengono tali elementi.

[1] I sottoinsiemi vengono generati mediante la rappresentazione caratteristica, ovvero 1 se l'elemento è presente 0 altrimenti.

In termini di pseudo-codice:

Algoritmo 8.2 Backtracking

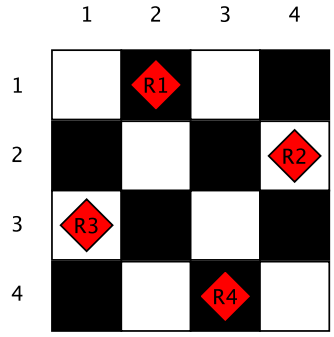
```
SOTTOINSIEMI(Sol, n, m, i, CurrentSum, RemainingSum)
  if (i>n) then
    PRINT(Sol) and break
  if (CurrentSum+RemainingSum-Elem[i]≥ m) then
    Sol[i]=0; SOTTOINSIEMI(Sol, n, m, i + 1, CurrentSum, RemainingSum-Elem[i])
  else break
  if (CurrentSum+Elem[i]≤ m) then
    Sol[i]=1; SOTTOINSIEMI(Sol, n, m, i + 1, CurrentSum+Elem[i], RemainingSum-Elem[i])
  else break
```

Come si può facilmente notare, oltre ad un'enorme riduzione del tempo computazionale, non vi è nemmeno più il bisogno della funzione di CHECK poichè ogni stampa del vettore "Sol" produce un sottoinsieme di somma m.

8.2 Il problema delle n regine

In questo problema abbiamo una scacchiera $n \times n$ ed n regine da posizionarvi sopra senza che nessuna di esse viene "mangiata" da nessun altra secondo le regole degli scacchi.
La figura mostra un esempio di possibile soluzione per $n = 4$:

Figura 8.2: Possibile soluzione per il problema delle 4 regine



Prima di dare lo pseudo-codice dell'algoritmo che trova tutte le possibili configurazioni corrette mediante backtracking, osserviamo che per le diagonali valgono le seguenti regole:

1. Sia d_1 una diagonale a partire da alto-sinistra verso basso-destra. Per ogni elemento (i, j) su d_1 vale che: $i - j = k_{d_1}$ (con i righe, j colonne e k_{d_1} costante).

2. Sia d_2 una diagonale a partire da alto-destra verso basso-sinistra. Per ogni elemento (i, j) su d_2 vale che: $i + j = k_{d_2}$ (con i righe, j colonne e k_{d_2} costante).

Se una regina R_1 è in posizione (i, j) ed una regina R_2 è in posizione (m, l) della scacchiera possiamo concludere che esse sono sulla stessa diagonale se e solo se vale che:

$$j - l = |i - m|$$

Questo perchè se fossero sulla stessa diagonale varrebbe una di queste due implicazioni:

1. Se $i - j = m - l \Rightarrow j - l = i - m$

2. Se $i + j = m + l \Rightarrow j - l = m - i$

Per essere sicuri che esse non occupano una stessa diagonale basta controllare che $j - l \neq |i - m|$.
Siamo ora pronti per fornire lo pseudo-codice. Il vettore "Sol", di lunghezza n, indica che la regina i-esima deve occupare la colonna Sol[i]-esima della scacchiera. Nell'esempio della figura 8.2 la soluzione sarebbe data dalla quadrupla $(2, 4, 1, 3)$ poichè (senza perdere in generalità) la numerazione delle regine inizia dalla riga più alta e finisce nella riga più bassa della scacchiera.

Algoritmo 8.3 Backtracking

```
REGINE(Sol, r, n)
    if (r > n) then
    PRINT(Sol) and exit
  for i=r to n do
    if (CHECK(Sol, r, i)) then
      Sol[r]=i
      REGINE(Sol, r + 1, n)

Procedure CHECK(Sol, r, i)
  if (r = 0) then true
  for j = 0 to r − 1 do
    if (i-Sol[j]=|r-j|) then false
  return true
```

8.3 Cicli Hamiltoniani

A differenza dei cicli Euleriani, in cui si ha che $\forall\{x, y\} \in E(G)$, $\{x, y\}$ viene attraversato una sola volta da cui deriva la proprietà strutturale che ogni vertice ha grado pari, con i cicli Hamiltoniani **non si ha nessuna proprietà strutturale** e quindi bisogna procedere per backtracking per poterli enumerare tutti.
I vincoli che impone il problema sono i seguenti:

- Rappresentazione dell'*output* in una permutazione (x_1, ...,x_n) dove $x_i \in V$
- $(x_i, x_{i+1}) \in E$

Sull'albero d'esecuzione possiamo distinguere diverse due tipologie di nodi e tre diverse tipologie di stati:

- Nodo vivo, il quale può portare ad una soluzione
- Nodo morto, il quale, non avendo passato i controlli, non porta ad una soluzione e viene rappresentato come una foglia senza soluzione
- Stato soluzione, il quale è la foglia che porta alla soluzione
- Stato del problema, il quale, come il nodo vivo, può portare ad una soluzione successiva
- Stato risposta: il quale porta ad una soluzione "SI" o "NO" del problema ed è rappresentato come una foglia dell'albero

Ogni volta quindi l'algoritmo, descritto in maniera estesa in Algoritmo 8.4, dovrà controllare se esiste l'arco tra l'i-esimo e l'$(i+1)$-esimo vertice evitando però di ricadere in vertici già visitati, ovvero in cicli che non sono Hamiltoniani.
Last but not least, bisognerà verificare che esista anche l'arco $\{x_n, x_1\}$, prima di stampare la l'eventuale soluzione, al fine di completare il ciclo Hamiltoniano.

Algoritmo 8.4 Backtracking

```
HAMCYCLES(Sol,k,Visitati,n,G)
  if (k = n and {Sol[n], Sol[1]}∈ E) then
    PRINT(Sol)
    break
  for i = 1 to n do
    if k = 0 then
      Sol[k + 1]=i
      Visitati[i]=1
      HAMCYCLES(Sol,k + 1,Visitati,n,G)
    elif (Visitati[i]=0 and {Sol[k], i}∈ E) then
      Sol[k + 1]=i
      Visitati[i]=1
      HAMCYCLES(Sol,k + 1,Visitati,n,G)
    Visitati[i]=0
```

La Figura 8.4 mostra parte dell'esecuzione dell'algorimo sul grafo G in Figura 8.3.

Figura 8.3: Grafo G usato per l'esecuzione di HAMCYCLES

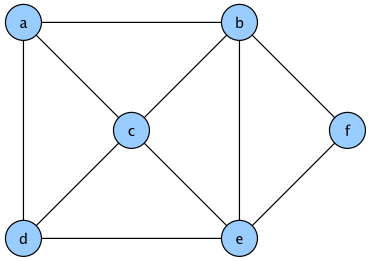
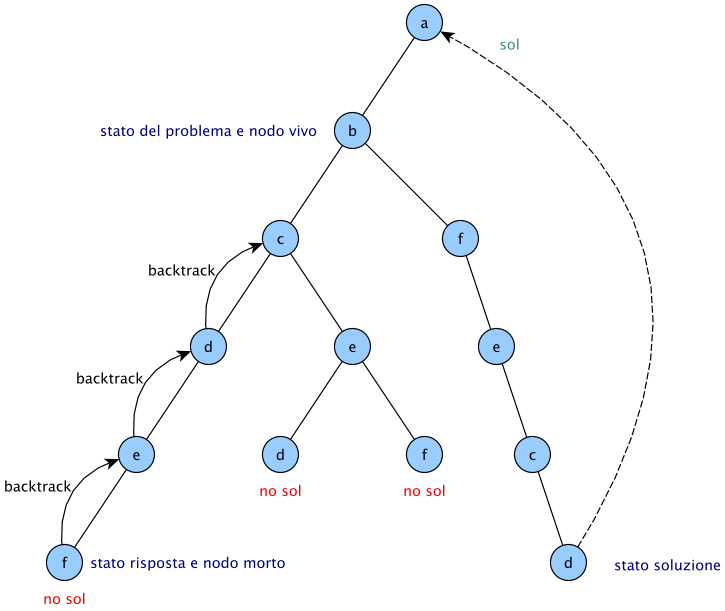


Figura 8.4: Esempio parziale di HAMCYCLES sul grafo G



Capitolo 9

Branch & Bound

La tecnica *Branch & Bound* viene usata per problemi di ottimizzazione.

Rispetto al Backtracking richiede in più per ogni nodo:

- di poter **calcolare un limite superiore** (o inferiore) rispetto al valore delle soluzioni ammissibili raggiungibili da quel nodo, il quale solitamente non è una soluzione ammissibile
- di **conoscere il valore della migliore soluzione** calcolata fino a quel momento.

Grazie alla conoscenza del valore della migliore soluzione calcolata fino a quel momento è possibile scartare le altre costruite. Ogni volta che viene assegnato un valore parziale per la costruzione della soluzione, bisogna riassegnare i valori non calcolati, ma "pronosticati" quando si era calcolato il lower bound, per fare sì che la soluzione pronosticata sia una soluzione ammissibile che rispetta i vincoli del problema.

Una soluzione ottima è una soluzione ammissibile che raggiunge il miglior valore per la funzione obiettivo.

9.1 Problema dell'assegnamento

Si vogliono assegnare n persone ad n lavori t.c. ogni persona ad esattamente un lavoro, ogni lavoro ad esattamente una persona, in modo da rendere il costo totale dell'assegnamento il minore possibile.

Oppure in altra formulazione bisogna selezionare un elemento in ogni riga della matrice in modo tale che la somma degli elementi selezionati sia la più piccola possibile e nessuna coppia di elementi selezionati sia sulla stessa colonna.

Come **input** si ha:

- n persone, n differenti lavori.
- Matrice dei costi C(n·n): il costo dell'assegnamento della persona i al lavoro j è la quantità C(i,j).

e come **lower bound** viene scelta la somma dei valori minimi sulle singole righe, che in generale non è una soluzione ammissibile.

Si prenda in considerazione la Tabella 9.1 per il problema dell'assegnamento.

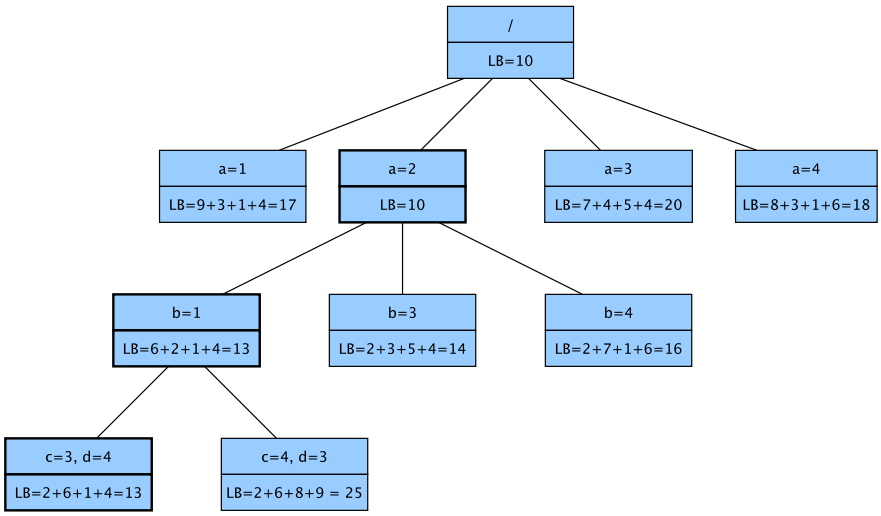
Tabella 9.1: Esempio del problema dell'assegnamento

		lavoro			
		1	2	3	4
persona	*a*	9	**2**	7	8
	b	6	4	**3**	7
	c	5	8	**1**	8
	d	7	6	9	**4**

Il lower bound è dato dalla somma dei valori minimi di ogni riga, quindi:

$$LB_0 = \sum_{i=0}^{n} minA[i,j] \ \forall j \in V = 2+3+1+4 = 10$$

Figura 9.1: Albero per il problema dell'assegnamento



Dalla Figura 9.1 si nota che il valore ottimo per il problema dell'assegnamento è la permutazione 2,1,3,4 che porta ad un costo di 13.

9.2 Problema dello zaino

Il problema dello zaino è abbastanza famoso in letteratura: si ha a disposizione uno zaino con capacità C ed n elementi di peso c_i e valori associati v_i e si vuol trovare il miglior riempimento dello zaino ottimizzandone il valore dello stesso, dove questo è dato dalla somma di tutti i valori degli elementi inseriti nello zaino.

Per risolvere questo problema con *Branch & Bound* inizialmente vengono ordinati tutti i rapporti $\frac{v_i}{w_i}$ per ogni oggetto in ordine decrescente:

$$\frac{v_n}{w_n} \geq \frac{v_{n-1}}{w_{n-1}} \geq \ldots \geq \frac{v_1}{w_1} \tag{9.1}$$

Sapendo (9.1) si può ipotizzare che la capienza rimanente dello zaino possa essere riempita da oggetti con rapporto $\frac{valore}{peso}$ relativamente all'oggetto preso in considerazione nell'istante i-esimo. Da qui possiamo prendere in considerazione il **upper bound** dato dalla seguente formula:

$$UB_i = V_i + (W - w_i) \cdot \frac{v_{i+1}}{w_{i+1}} \tag{9.2}$$

dove:

- W indica la capacità dello zaino
- w_i indica il riempimento dello zaino nell'istante i-esimo
- V indica il valore dello zaino in quel momento

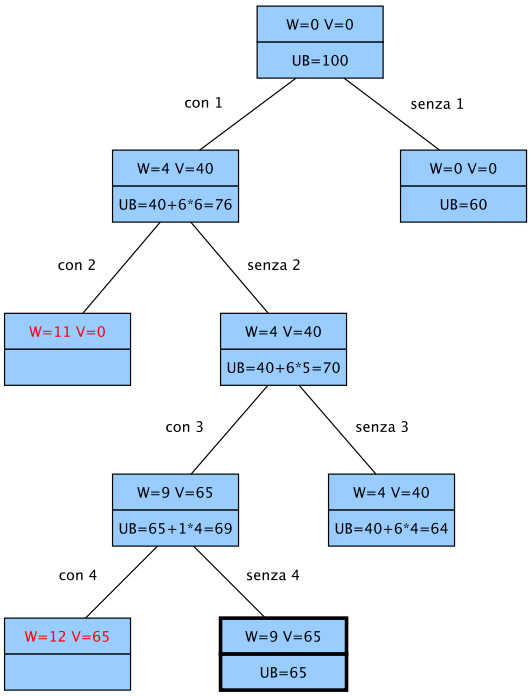
Si prenda ora in considerazione la Tabella 9.2 e la capacità dello zaino W=10.

Tabella 9.2: Esempio del problema dello zaino

	peso	valore	$valore/peso$
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

Il lower bound, secondo (9.2), è pari a 100.

Figura 9.2: Albero per il problema dello zaino



9.3 Problema del commesso viaggiatore

Il problema del commesso viaggiatore è la versione pesata del problema del ciclo hamiltoniano in un grafo completo, già descritto nella Sezione 8.3.

Come nei casi precedenti viene definito un lower o upper bound. In questo caso un **upper bound banale** è dato da

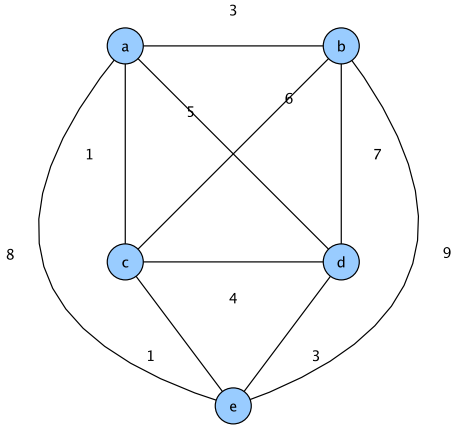
$$\text{UP} = max(\text{c}\{\text{x,y}\})\cdot m$$

Una maniera più raffinata ed **intelligente** di considerate un **upper bound** è invece considerare per ogni vertice due archi di costo minore

$$\text{UP}^1 = \frac{\sum s_i}{2} \ \forall i \in [n]$$

in cui s_i è, per quanto già detto, la somma delle distanze minime da x_i a due città ad esso vicine.
Si consideri il grafo nella Figura 9.3,

Figura 9.3: Esempio del problema del commesso viaggiatore



da cui, per quanto detto sull'upper bound, si ricava la Tabella 9.3

Tabella 9.3: Tabella dei costi iniziali

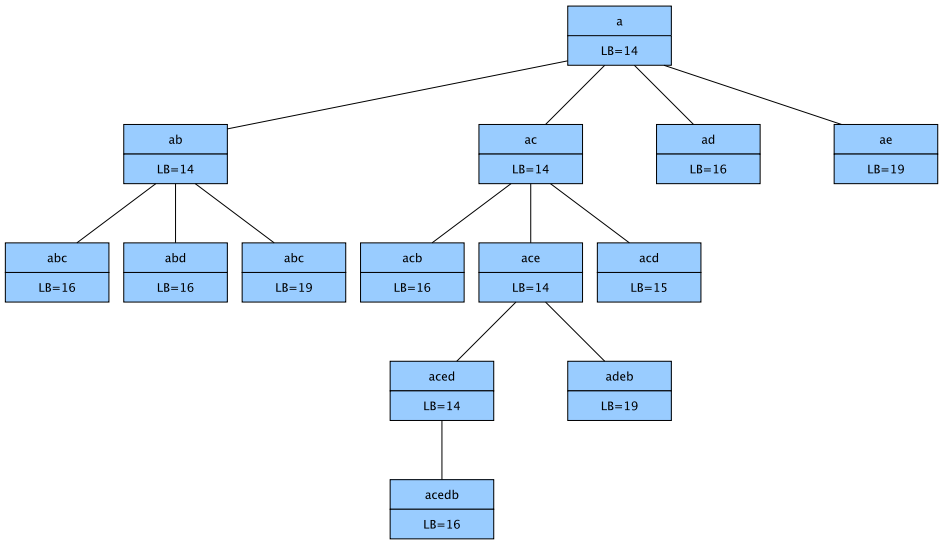
Nodo	*Somma costi*	*Città più vicine*	S_i
a	3+1	c e b	4
b	3+5	a e c	9
c	1+2	a e d	3
d	4+3	e e a	7
e	2+3	d e c	5

28/2=14

[1] si divide per 2 perchè per ogni nodo si prende una coppia di città e alla fine dell'iterazione ogni nodo farà parte del ciclo hamiltoniano pesato e apparirà 2 volte

Una volta che viene scelto un arco $\{x,y\}$ appartenente al ciclo hamiltoniano pesato esso viene fissato all'interno della tabella e da ciò viene ricalcolato l'upperbound. Di seguito l'albero generato

Figura 9.4: Albero per il problema del commesso viaggiatore



Si evince dall'albero generato che il ciclo hamiltoniano pesato esiste ed è *a, c, e, d, b* di costo 16.

Capitolo 10

Algoritmi Approssimanti

Un algoritmo approssimante è un algoritmo che ci permette, *in tempo polinomiale*, di ottenere una soluzione per un problema di ottimizzazione di natura esponenziale. Ovviamente, tale soluzione, non è detto che sia sempre ottima, tuttavia vedremo che esistono algoritmi approssimanti con un ottimo livello di approssimazione i quali sono preferibili ad algoritmi di tipo backtracking o branch&bound.
Iniziamo introducendo un po' di notazione che ci sarà utile nel corso del capitolo:

- S_a soluzione approssimata
- S^* soluzione ottima
- $f(S_a)$ costo della soluzione approssimata
- $f(S^*)$ costo della soluzione ottima
- $re(S_a) = \frac{f(S_a) - f(S^*)}{f(S^*)}$ errore relativo della soluzione approssimante
- $r(S_a)$ rapporto di accuratezza (≥ 1)
- $r(S_a) = \frac{f(S_a)}{f(S^*)}$ per i problemi di minimizzazione
- $r(S_a) = \frac{f(S^*)}{f(S_a)}$ per i problemi di massimizzazione
- $\alpha = max\{r(S_a)\}$ tanto più è vicino ad 1 tanto più l'algoritmo è buono

Solitamente, in letteratura, tali algoritmi vengono anche chiamati algoritmi $\alpha-$approssimanti se α stabilisce un upper bound (non infinito) ad ogni istanza possibile del problema.

10.1 Il problema dello zaino

Vediamo un algoritmo approssimante per il problema dello zaino. Per tale algoritmo è stato dimostrato[1] che:

$$\alpha = 1 + \frac{1}{k} \qquad \text{con } 0 < k \leq |oggetti|$$

Questo algoritmo approssimante funziona nel seguente modo:

1. Crea tutti i sottoinsiemi di cardinalità minore od uguale a k basandoti sugli oggetti da inserire nello zaino
2. In maniera *greedy* rispetto al rapporto valore/peso, inserisci gli oggetti rimanenti finchè puoi e stando attento a non eccedere nella capacità dello zaino
3. Scegli il riempimento che ti porta al guadagno maggiore

Il tempo di questa procedura è polinomiale, per la precisione è pari a $O(k \cdot |oggetti|^{k+1})$ e fornisce una soluzione approssimata a questo problema di natura esponenziale.
Per il punto 1 (con $n = |oggetti|$) si ha che:

$$\sum_{i=1}^{k} \binom{n}{i} = \sum_{i=1}^{k} \frac{n \cdot (n-1) \cdot ... \cdot (n-j+1)}{j!}$$

Tale somma è maggiorabile con:

$$\sum_{i=1}^{k} \binom{n}{i} \leq \sum_{i=1}^{k} n^j \leq \sum_{i=1}^{k} n^k = n^k \cdot (k+1)$$

Per il punto 2 bisogna moltiplicare tale quantità per n quindi otteniamo:

$$n^{k+1} \cdot (k+1) = O(k \cdot |oggetti|^{k+1})$$

Esempio di esecuzione:

Oggetto	Peso	Valore	Valore/Peso
1	4	40	10
2	7	42	6
3	5	25	5
4	1	4	4

Supponiamo che la capienza massima dello zaino sia pari a 10 e che $k = 2$.
Quindi eseguiamo i primi due passi dell'algoritmo:

Sottoinsiemi degli oggetti	Aggiunta (greedy)	Valore dello zaino
$\emptyset$	$\{1, 3, 4\}$	69
$\{1\}$	$\{3, 4\}$	69
$\{2\}$	$\{4\}$	46
$\{3\}$	$\{1, 4\}$	69
$\{4\}$	$\{1, 3\}$	69
$\{1, 2\}$	$\emptyset$	eccede nel peso
$\{1, 3\}$	$\{4\}$	69
$\{1, 4\}$	$\{3\}$	69
$\{2, 3\}$	$\emptyset$	eccede nel peso
$\{2, 4\}$	$\emptyset$	46
$\{3, 4\}$	$\{1\}$	69

Ovviamente una delle qualsiasi soluzioni con valore 69 è quella migliore.

[1] Dimostrazione non in programma.

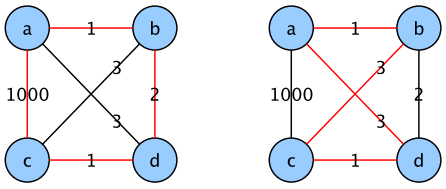
10.2 Il problema del commesso viaggiatore (TSP)

In questa sezione vedremo inizialmente alcuni algoritmi riguardanti il problema del commesso viaggiatore. Successivamente ci concentreremo su di una variante di tale problema chiamata: commesso viaggiatore **euclideo**. Tale variante ammette algoritmi di approssimazione ma è ovviamente più restrittiva rispetto al problema originale.

10.2.1 Il vicino più vicino

Come descrive il nome del paragrafo, questo algoritmo sceglie per ogni nodo il vicino che può raggiungere con minor costo. Tale algoritmo non è poi così buono poichè il suo errore cresce con la distanza dell'arco che chiude il ciclo Hamiltoniano pesato, poichè esso è obbligato.
Si consideri infatti il seguente esempio (partendo dal nodo a) nella figura sottostante:

Figura 10.1: Approssimante vs Ottimo



Come possiamo osservare, l'arco $\{c, a\}$ è quello che potrebbe creare un costo elevatissimo al ciclo Hamiltoniano rispetto alla soluzione ottima. Il parametro α non avrebbe quindi nessun limite, potrebbe crescere in maniera smisurata, pertanto tale algoritmo non è α-approssimante.

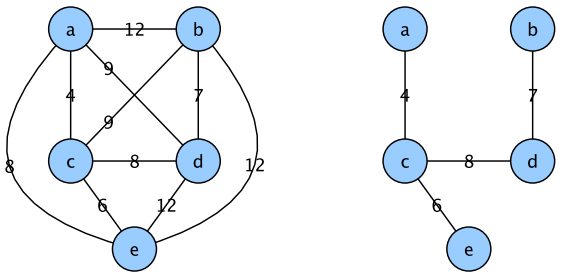
10.2.2 Due giri intorno all'albero

Questo algoritmo approssimante si compone delle seguenti fasi:

1. Costruisce l'MST per il grafo completo pesato dato in input
2. Esegue una DFS a partire da un qualsiasi nodo sorgente dell'MST e crea un circuito pesato considerando tutte le ripetizioni di archi e nodi (due giri intorno all'albero)
3. Elimina le ripetizioni di archi e nodi intermedie e fornisce il ciclo Hamiltoniano pesato

Si consideri la seguente figura come esempio d'esecuzione:

Figura 10.2: Esempio di esecuzione



Un possibile circuito ottenibile partendo dal nodo a dopo i due giri dell'MST è $a, c, e, c, d, b, d, c, a$ da cui si ricava poi il ciclo Hamiltoniano pesato a, c, e, d, b, a di costo 41.
Purtroppo, per il problema del commesso viaggiatore originale, della bontà di questo algoritmo di approssimazione non sappiamo dire nulla.

10.2.3 La variante Euclidea (TSPE)

Come già accennato, questa variante è più restrittiva del problema originale ed infatti impone un vincolo in più sulle distanze:

$$d(i,j) \leq d(i,k) + d(k,j) \qquad \text{per ogni } i,j,k \in V$$

Nel 1977, è stato dimostrato[2] da Reingold, Nievergled e Deo che l'algoritmo del "vicino più vicino" su tali istanze del problema risulta essere $\alpha-$approssimante con un parametro $\alpha = \frac{\lceil \log n \rceil + 1}{2}$.
Quello che invece dimostreremo adesso è che l'algoritmo "due giri intorno all'albero", su istanze di TSP Euclidee, risulta essere un algoritmo $2-$approssimante.
Dato un generico grafo G completo, pesato e che rispetta i vincoli sopracitati, consideriamo la sua soluzione ottima S^* (per TSPE), la quale esiste sempre. Essendo essa un ciclo, eliminando un qualsiasi arco da tale soluzione otterremo uno Spanning Tree (ST).
Possiamo quindi osservare che

$$f(S^*) \geq Costo(ST) \geq Costo(MST)$$

da cui segue che

$$2 \cdot f(S^*) \geq 2 \cdot Costo(MST)$$

Ma $2 \cdot Costo(MST)$ è esattamente il costo del circuito costruito dopo i "due giri intorno all'albero".
Sotto l'ipotesi Euclidea, possiamo banalmente affermare che

$$f(S_a) \leq 2 \cdot Costo(MST)$$

da cui segue la tesi

$$f(S_a) \leq 2 \cdot f(S^*) \Rightarrow \frac{f(S_a)}{f(S^*)} \leq 2 = \alpha$$

10.2.4 Un risultato importante

Sotto l'ipotesi che P≠NP è possibile affermare che non esistono algoritmi $\alpha-$approssimanti per il problema del commesso viaggiatore **originale**.
Supponiamo che tale algoritmo esista, consideriamo un generico grafo $G = (V, E)$ e costruiamo $G' = (V', E')$ il grafo completo pesato associato a G nel seguente modo:

1. $V = V'$
2. Per ogni arco in E, riportalo in E' e dagli peso 1
3. Inserisci gli archi che mancano ad E' per rendere G' completo dandogli peso pari ad $\alpha n + 1$.

In questo modo eseguendo l'algoritmo (supposto per assurdo) su G', si ottiene che:

- se G ha un ciclo Hamiltoniano $\Rightarrow f(S^*) = n \Rightarrow \frac{f(S_a)}{n} \leq \alpha \Rightarrow f(S_a) \leq \alpha n$
- se G non ha un ciclo Hamiltoniano $\Rightarrow f(S_a) > f(S^*) > \alpha n$

Questo ci permetterebbe di decidere in tempo polinomiale HAMCYCLE che sappiamo essere NP-Completo e quindi contraddire l'ipotesi che P≠NP. □

[2] Dimostrazione non in programma.

Capitolo 11

Stringhe

Trovare tutte le occorrenze di una stringa all'interno di un testo è un problema che si presenta frequentemente nei programmi di scrittura dei testi. Il *testo* tipicamente è un documento e la stringa cercata, chiamata anche *pattern*, è una particolare parola fornita dall'utente.
Iniziamo con un po' di notazione e definizioni:

- Σ^* insieme di tutte le stringhe di lunghezza arbitraria formate da simboli appartenenti all'alfabeto Σ
- Testo $T[1 \dots n] \in \Sigma^*$
- Pattern $P[1 \dots m] \in \Sigma^*$
- $|x|$ dimensione della stringa $x \in \Sigma^*$
- $\epsilon \in \Sigma$ stringa vuota
- xy concatenazione delle stringhe $x, y \in \Sigma^*$

Definizione 10 *Una stringa w si dice prefisso di una stringa x se e solo se x=wy*

Definizione 11 *Una stringa w si dice suffisso di una stringa x se e solo se x=yw*

Da notare che ϵ è suffisso e prefisso di una qualsiasi stringa x ed inoltre come le relazioni di prefisso e suffisso siano *relazioni transitive.*

Definizione 12 *Uno spostamento $s \in \mathbb{N}$ è uno spostamento valido per P in T se $T[s+1,\dots,s+m] = P[1,\dots,m]$*

Dalla Definizione 12 si può trarre una *riformulazione del problema dello string-matching*: dato un pattern P trovare tutti gli spostamenti validi s per il pattern P in un testo T.

11.1 Algoritmo Naif

L'algoritmo *naif* trova tutti gli spostamenti validi controllando l'uguaglianza fra $P[1,\ldots,m]$ e $T[s+1,\ldots,s+m]$ per ognuno degli $n-m+1$ possibili valori di s.

Algoritmo 11.1 Algoritmo Naif

for $s = 0$ *to* $n-m$ **do**
  if $P[1,\ldots,m] = T[s+1,\ldots,s+m]$ **then stampa** s

Esempio:

Siano T= *acaabc* e P=*aab*
Per s= 0 P è confrontato con T[1,. . . ,3]
Per s= 1 P è confrontato con T[2,. . . ,4]
Per s= 2 P è confrontato con T[3,. . . ,5] (unico caso in cui vale l'uguaglianza)
Per s= 3 P è confrontato con T[4,. . . ,6]

Il tempo d'esecuzione è ovviamente $\Theta((n-m+1)\cdot m)$.

11.2 Regola di Horner e classi resto

Prima di vedere l'*algoritmo Rabin-Karp* che migliora l'algoritmo Naif è doveroso fare un'introduzione sulla *regola di Horner* e sulle *classi resto.*
La **regola di Horner**, o più correttemente l'algoritmo di Horner, permette di valutare un polinomio $P_N(x) = x^N + a_1x^{N-1} + \ldots + a_{N-1}x + a_N$ in questo modo:

$$P_N(x) = a_N + x(a_{N-1} + x(a_{N-2} + \ldots + xa_1)\ldots) \tag{11.1}$$

La regola di Horner in 11.1 può essere facilmente estesa per rappresentare numeri decimali:

$$1251 = \sum_{j=0}^{3} a^j x^j = a_0 + 10(a_1 + 10(a_2 + 10a_3)) = 1 + 10(5 + 10(2 + 10 \cdot 1)) = 1251$$

Per semplificare l'esposizione d'ora in poi supponiamo che $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, in modo che ciascun carattere sia una cifra decimale. Notiamo però che nel caso generale si puo' sempre adattare questa soluzione apportando le dovute modifiche alle procedure di conversione e di confronto.

Una **classe di equivalenza** a modulo n è un insieme di numeri tali che la divisione per n dia resto a.
In maniera più rigorosa:

$$[a]_n = \{a + kn : z \in \mathbb{Z}\} \tag{11.2}$$

11.3 Algoritmo Rabin-Karp

Una generica sottostringa di T definita tramite uno spostamento s, $T[s+1,\ldots,s+m]$, sfruttando la regola di Horner (Definizione 11.1), è possibile convertirla quindi in un numero decimale t_s, lo stesso vale per il pattern P.
L'algoritmo ha i seguenti costi:

- t_0 e P convertiti a numeri decimali in tempo $O(m)$ mediante la regola di Horner (calcolo preliminare)
- Potenze di 10 calcolate in tempo $O(m)$ (calcolo preliminare)
- Il passaggio da t_s a t_{s+1} può avvenire in maniera costante mediante la seguente formula:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \tag{11.3}$$

Esempio: Da $s_1 = 1251$ a $s_2 = 2518$ sulla stringa 12518...

Si sottrae a s_1 il valore $s_1[1] \cdot 10^m = 1 \cdot 10^4$ ottenendo 251. Esso viene traslato di una posizione a sinistra (moltiplicando quindi per 10) ottenendo 2510. In ultimo si aggiunge la cifra successiva, da cui 2518.

L'algoritmo Rabin-Karp è identico come procedura all'algoritmo naif presentato nella Sezione 11.1, ma per pattern non troppo grandi si nota che il tempo delle operazioni aritmetiche è costante e ovviamente la complessita è pari a $O(m) + O(n-m+1)$.

Si nota che se le dimensioni di P sono grandi, è irragionevole pensare che ogni operazione aritmetica sia costante.

A tal proposito si utilizzano le *classi resto*, viste precedentemente, per diminuire il numero dei confronti che potrebbero non portare ad un matching del pattern con una sottostringa del testo.
Per ogni t_s e per il pattern P, viene calcolata la classe resto modulo un valore q. Notiamo adesso che se un generico t_s è congruo a $P \, mod \, q$ non possiamo necessariamente affermare che le due stringhe P e t_s siano uguali e dovremo procedere con il confonto. Se però t_s non è congruo a $P \, mod \, q$ allora possiamo concludere che le due stringhe P e t_s sono differenti.
La complessità nel caso medio è $O(n) + O(m(v + n/q))$ con v numero degli spostamenti validi.

11.4 Stringhe ed automi a stati finiti

Definizione 13 *Un automa a stati finiti è una quintupla (Q,q_0,A,Σ,δ) dove:*

- Q è un insieme finito di stati
- q_0 in Q è lo stato iniziale
- A in Q è un insieme di stati accettanti
- Σ è un alfabeto di input finito
- δ è una funzione da $Q \times \Sigma$ in Q, chiamata funzione di transizione, ovvero $\delta(q, a)$ è il nuovo stato in cui giunge l'automa che era nello stato q dopo aver letto il carattere a

E' possibile trasformare il problema dello string matching visto precedentemente al problema di riconoscimento di una sequenza di caratteri con un automa a stati finiti.
L'automa che ci servirà è definito nel seguente modo:

Definizione 14 *Un automa a stati finiti per la corrispondenza fra stringhe relativo ad un dato pattern P[1,. . . ,m] è siffatto:*

- $Q = \{0, 1, \ldots, m\}$ è l'insieme finito di stati
- $q_0 = 0$ in Q è lo stato iniziale
- $A = m$ in Q
- Σ è un alfabeto di input finito
- $\delta : Q \times \Sigma \to Q$ dove $\delta(q, a)$ restituisce lo stato q' che riconosce il più lungo prefisso del pattern P che è anche suffisso del testo T letto fino al simbolo a

Per essere più chiari vediamo un esempio.
Dato il pattern $P = ababaca$ con caratteri sull'alfabeto (a, b, c), l'automa corrispondente è definito dalla seguente tabella:

Q	*Riconoscimento*	*Funzione δ rispetto all'input*		
		$q' = \delta(q, a)$	$q' = \delta(q, b)$	$q' = \delta(q, c)$
0	$\emptyset$	1	0	0
1	a	1	2	0
2	ab	3	0	0
3	aba	1	4	0
4	abab	5	0	0
5	ababa	1	4	6
6	ababac	7	0	0
7	P	1	2	0

Infine vediamo lo pseudocodice:

Algoritmo 11.2 Stringhe ed automi(T, δ, m)

```
n=lunghezza(T)
q = 0
for i = 1 to n do
  q = δ(q, T(i))
    if q = m then
      s = i − m
      stampa "il pattern appare con spostamento s"
```

il quale ha tempo d'esecuzione pari a $O(n) + calcolo\,di\,\delta$ che è ovviamente pre-elaborato.