

ALGORITMI  
E STRUTTURE  
DATI : PART 2

PROGRAMMATIONE  
DINAMICA

# PROGRAMMAZIONE DINAMICA

La programmazione dinamica si basa sul principio che se non si ha la certezza di quali problemi risolvere, si può prorare a risolverli tutti e conservare i risultati ottenuti per poterli usare successivamente.

Rispetto al divide-et-impera la programmazione dinamica è:

- iterativa, non ricorsiva
- affronta i problemi bottomup e non topdown
- memorizza i risultati in una tabella
- risolve un sottoproblema una sola volta.

## ESEMPIO - [Coefficiente binomiale]

$C(n, k) = n!/(k!(n-k)!)$  rappresenta il numero di modi di scegliere  $k$  oggetti da un insieme di  $n$  oggetti, con  $0 \leq k \leq n$ , ed è definibile ricorsivamente come segue:

$$C(n, k) = \begin{cases} 1 & \text{se } k=0 \vee n=k \\ C(n-1, k-1) + C(n-1, k) & \text{altrimenti} \end{cases}$$

Un algoritmo ricorsivo *divide-et-impera* si ottiene immediatamente dalla precedente definizione ricorsiva:

### integer C(integer $n, k$ )

```
if  $n = k$  or  $k = 0$  then return 1
else return  $C(n - 1, k - 1) + C(n - 1, k)$ 
```

Purtroppo, la complessità dell'algoritmo *divide-et-impera* cresce come il numero di chiamate ricorsive, che è proprio uguale a  $C(n, k)$ . Questo è dovuto all'elevato numero di sottoproblemi identici che vengono risolti più volte. Per esempio già  $C(n-2, k-1)$  è richiamata due volte, sia per calcolare  $C(n-1, k)$  che  $C(n-1, k-1)$ , mentre al decrescere dei valori degli argomenti ciascun sottoproblema è risolto per un numero sempre maggiore di volte. Un algoritmo di programmazione dinamica, invece, risolve i sottoproblemi per valori crescenti degli argomenti secondo il ben noto schema del "triangolo di Tartaglia", memorizzando i risultati una volta per tutte in una matrice intera  $C[0 \dots n][0 \dots n]$  in  $\Theta(n^2)$  tempo.

### tartaglia(integer $n$ , integer[][] $C$ )

```
for integer  $i \leftarrow 0$  to  $n$  do
   $C[i, 0] \leftarrow 1$ 
   $C[i, i] \leftarrow 1$ 
for integer  $i \leftarrow 2$  to  $n$  do
  for integer  $j \leftarrow 1$  to  $i-1$  do
     $C[i, j] \leftarrow C[i-1, j-1] + C[i-1, j]$ 
```

In questo modo, il valore di  $C(m, k)$ , per ogni coppia di interi  $m, k$  tali che  $0 \leq k \leq m \leq n$ , può essere successivamente letto direttamente dalla matrice  $C$  in tempo  $O(1)$ .

La programmazione dinamica è stata introdotta per risolvere problemi di ottimizzazione. Perché sia applicabile occorre:

- (1) Sia possibile combinare le soluzioni dei sottoproblemi per risolvere un problema più grande
- (2) le decisioni prese per risolvere un problema rimangono tali anche per risolvere un suo sottoproblema
- (3) una tecnica divide-et-impera richiede di risolvere un sottoproblema più volte.

Le proprietà (1) e (2) prendono il nome di sottostruttura ottima. Affinché un algoritmo basato su programmazione dinamica abbia complessità polinomiale occorre:

- (4) ci siano un numero polinomiale di sottoproblemi da risolvere
- (5) per memorizzare i risultati dei sottoproblemi viere utilizzata una tabella.
- (6) il tempo per trovare una soluzione al problema e combinare le soluzioni dei sottoproblemi deve essere polinomiale

# MOLTIPLICAZIONE CATENA DI MATRICI

Date  $n$  matrici,  $A_1, A_n$ , si consideri il problema di calcolare il prodotto  $A_1 \cdot A_2 \cdots A_n$ . Questo è possibile se le matrici sono compatibili a due a due con il prodotto, ovvero il numero di colonne di  $A_i$  è uguale al numero di righe di  $A_{i+1}$ . Moltiplicare due matrici  $A_i$  di dimensione  $C_{i-1} \times C_i$  e  $A_{i+1}$  di dimensioni  $C_i \times C_{i+1}$  richiede  $C_{i-1} \cdot C_i \cdot C_{i+1}$  moltiplicazioni scalari. Essendo la moltiplicazione di matrici associativa, il numero di moltiplicazioni scalari dipende dall'ordine in cui vengono moltiplicate le matrici.

- Date  $n$  matrici  $A_1, \dots, A_n$ , trovare l'ordine di moltiplicazioni di costo minimo, ovvero che richiede il minor numero di moltiplicazioni scalari.

Sfruttando l'associatività delle matrici, si può ragionare su come parentesizzare le matrici in modo da ridurre le moltiplicazioni scalari: una parentesizzazione  $P_{i,j}$  del prodotto  $A_i, A_{i+1}, \dots, A_j$  consiste nella matrice  $A_i$  se  $i=j$ , nel prodotto di due parentesizzazioni  $(P_{i,k} \cdot P_{k+1,j})$  altrimenti. Una parentesizzazione si dice ottima se non esiste altra parentesizzazione  $P'_{i,j}$  con costo inferiore.

La definizione ricorsiva ci permette di scrivere una soluzione basata sul divide-et-impera. Il numero  $F(n)$  di possibili parentesizzazioni è definibile:

$$F(n) = \begin{cases} 1 & n=1 \\ \sum_{k=1}^{n-1} F(k)F(n-k) & \text{altrimenti} \end{cases}$$

dove una matrice  $n=1$  ha in solo modo per essere espressa in parentesi, mentre nel caso di  $n$  matrici, per ognuno dei possibili indici  $k$  ci sono  $F(k)$  modi di parentesizzare  $A_1 \dots A_k$  e  $F(n-k)$  modi di parentesizzare  $A_{k+1} \dots A_n$ .  
 $F(n)$  è  $\mathcal{O}(4^n/n^{3/2}) \Rightarrow$  superpolinomiale.

È possibile esprimere il costo della parentesizzazione ottima  $P_{i,j} = (P_{i,n}, P_{k+1,j})$  come somma del costo delle parentesizzazioni ottime  $P_{i,k}$  e  $P_{k+1,j}$  più  $c_{i-1}c_k c_j$  moltiplicazioni scalari per moltiplicare tra loro le matrici risultanti di dimensione  $c_{i-1} \times c_k \times c_j$ .

Sia  $\Pi[i,j]$  il minimo numero di moltiplicazioni aritmetiche necessarie per calcolare  $P_{i,j}$ ; è possibile esprimere come:

$$\Pi[i,j] = \begin{cases} \min_{i \leq k \leq j-1} \{\Pi[i,k] + \Pi[k+1,j] + c_{i-1}c_k c_j\} & \text{se } 1 \leq i < j \leq n \\ 0 & \text{se } 1 \leq i = j \leq n \end{cases}$$

La procedura parentesizzazione() calcola la matrice M. Fissato  $h = j - i + 1$ , gli estremi i e j della sequenza  $A_i \cdot A_j$  di n matrici sono tali che  $1 \leq i \leq h-1$ , mentre  $j = i + h - 1$ .

Complessità  $\mathcal{O}(n^3)$

---

```
parentesizzazione(integer[] c, integer[][] M, integer[][] S, integer n)
```

---

```
integer i, j, h, k, t
for i ← 1 to n do
    M[i, i] ← 0
for h ← 2 to n do
    for i ← 1 to n - h + 1 do
        j ← i + h - 1
        M[i, j] ← +∞
        for k ← i to j - 1 do
            t ← M[i, k] + M[k + 1, j] + c[i - 1] · c[k] · c[j]
            if t < M[i, j] then
                M[i, j] ← t
                S[i, j] ← k
```

---

Oltre a calcolare una parentesizzazione è necessario produrne anche una; questo viene fatto con l'ausilio della matrice S che contiene, per ogni parentesizzazione  $P_{i,j}$ , l'indice  $S[i,j]=k$  che rappresenta il punto di moltiplicazione fra le due sottoparentesizzazioni.

---

```
stampaPar(integer[] S, integer i, integer j)
```

---

```
if i = j then
    print "A["; print i; print "]"
else
    print "("; stampaPar(S, i, S[i, j]); print ":"; stampaPar(S, S[i, j] + 1, j); print ")"
```

---

Complessità  $\mathcal{O}(n)$ .

## STRING MATCHING APPROXIMATO

Date una stringa  $P = p_1 \dots p_m$  (Pattern) e una stringa  $T = t_1 \dots t_n$  (testo), con  $m \leq n$ , un'occorrenza  $K$ -approssimata di  $P$  in  $T$ , con  $\text{OK}, \text{cm}$ , è una copia della stringa  $P$  nella stringa  $T$  in cui sono ammessi  $K$  errori (o differenti) fra i caratteri di  $P$  e i caratteri di  $T$ , del seguente tipo:

- (1) i caratteri di  $P$  e  $T$  sono diversi (sostituzione)
- (2) un carattere in  $P$  non è in  $T$  (inserimento)
- (3) un carattere in  $T$  non è in  $P$  (cancellazione)

Il problema consiste nel trovare un'occorrenza  $K$ -approssimata di  $P$  in  $T$  per cui  $K$  sia minima.

### ESEMPIO - [Occorrenza 2-approssimata]

Siano  $T = \text{"questoèunoscempio"}$  (dove  $n = 17$ ) e  $P = \text{"unesempio"}$  (dove  $m = 9$ ). Un'occorrenza 2-approssimata di  $P$  parte dalla posizione 8 di  $T$ : questoèunoscempio. Infatti, la prima e di  $P$  corrisponde ad una o di  $T$  (errore di tipo (1)), mentre la c di  $T$  è un carattere non incluso in  $P$  (errore di tipo (3)).

Definiamo il prefisso  $X(K) = x_1 \dots x_K$  della stringa  $X = x_1 \dots x_n$  come sottostringa con i primi  $K$  caratteri.

Definiamo  $D[i,j]$  con  $i \leq m$  e  $j \leq n$ , come il minimo valore  $K$  per cui esiste un'occorrenza  $K$ -approssimata di  $P(i:j)$  in  $T(j)$ .

Per ricavare il cammino minimo si definisce  $p^k(u,v)$  come il predecessore del nodo  $v$  in un cammino  $k$ -vincolato da  $u$  a  $v$ .

$$p^{k+1}(u,v) = \begin{cases} p^k(u,v) & \text{se } d^{k+1}(u,v) = d^k(u,v) \\ p^k(k,v) & \text{se } d^{k+1}(u,v) = d^k(u,k) + d^k(k,v) \end{cases}$$

La procedura floydWarshall() computa  $d^{k+1}(u,v)$  e  $p^{k+1}(u,v)$   $\forall u,v$  in  $G.V()$ : usa iterativamente la riga  $k$  e la colonna  $K$  delle matrici  $d$  e  $p$  per aggiornare i rimanenti elementi delle matrici.

```
floydWarshall(GRAPH G, integer[][], integer[][] p)
  foreach  $u, v \in G.V()$  do
     $d[u, v] \leftarrow +\infty$ 
     $p[u, v] \leftarrow 0$ 
  foreach  $u \in G.V()$  do
    foreach  $v \in G.adj(u)$  do
       $d[u, v] \leftarrow w(u, v)$ 
       $p[u, v] \leftarrow u$ 
  for  $k \leftarrow 1$  to  $n$  do
    foreach  $u \in G.V()$  do
      foreach  $v \in G.V()$  do
        if  $d[u, k] + d[k, v] < d[u, v]$  then
           $d[u, v] \leftarrow d[u, k] + d[k, v]$ 
           $p[u, v] \leftarrow p[k, v]$ 
```

Complessità  $O(n^3)$

## ESERCIZI

Sia dato in input un vettore  $V$  che ha interi distinti e un valore intero  $w$ .

Trovare il più grande valore ottenuto dalla somma (con ripetizione) di qualunque elemento in  $V$  che sia minore o uguale a  $w$ .

$$V = \{2, 3, 7\}, w = 15 \Rightarrow \text{out} = 15 \text{ perché } 3 \cdot 5 = 15$$

Questo problema è molto simile al problema dello zaino con  $w$  = alla capacità residua. Ragionando in modo analogo, la decisione da massimizzare sta nel decidere di prendere un determinato valore  $V[i]$  (quindi ridurre  $w$  di  $V[i]$ ) oppure non prendere in considerazione proprio quel valore. Supponendo che l'indice  $i$  scorre il vettore da  $n$  fino a zero (caso base) si ottiene la tabella:

$$F[i, w] = \begin{cases} 0 & i=0 \text{ OR } w=0 \\ -\infty & w < 0 \\ \max \left\{ F[i-1, w], F[i, w-V[i]] + V[i] \right\} & \text{altrimenti.} \end{cases}$$

skizzo l'elemento  $i$ -esimo,  
 $w$  rimane inalterato

considero l'elemento  $i$ -esimo,  
sottraggo il valore  $V[i]$  dalla capacità  
 $w$ , e non altero  $i$ ; perciò posso  
ancora considerare l'elemento  $i$ -esimo.

È possibile calcolare  $D[i,j]$  come segue

- $D[i-1, j-1]$  se  $p_i = t_j$
- $D[i-1, j-1] + 1$  se  $p_i \neq t_j$
- $D[i-1, j] + 1$  se  $p_i$  non compare in  $T$
- $D[i, j-1] + 1$  se  $t_j$  non compare in  $P$

La ricorsione termina quando una delle due stringhe è ruota.

$$D[i, j] = \begin{cases} 0 & \text{se } i=0 \\ i & \text{se } j=0 \\ \min\{D[i-1, j-1] + 1, D[i-1, j] + 1, D[i, j-1] + 1\} & \text{altrimenti} \end{cases}$$

dove  $\Delta = 0$  se  $p_i = t_j$  OR  $\Delta = 1$  se  $p_i \neq t_j$ .

Se il pattern è ruoto,  $K$  è per forza uguale a zero, perché la stringa ruota  $P(0)$  occorre in ogni posizione di  $T$ . Se il testo è ruoto, il numero  $K$  di errori è pari alla lunghezza del pattern.

---

```

integer stringMatching(ITEM[] P, ITEM[] T, integer m, integer n)
integer[][] D ← new integer[0 ... m][0 ... n]
for integer i ← 1 to m do D[i, 0] ← i
for integer j ← 0 to n do D[0, j] ← 0
for i ← 1 to m do
    for j ← 1 to n do
        integer t ← D[i - 1, j - 1] + if(P[i] = T[j], 0, 1)
        t ← min(t, D[i - 1, j] + 1)
        t ← min(t, D[i, j - 1] + 1)
        D[i, j] ← t
    integer min ← D[m, 0]
    integer pos ← 0
    for j ← 1 to n do
        if D[m, j] < min then
            min ← D[m, j]
            pos ← j
    return pos

```

---

$p_{i-1}$	$t_{j-1}$	$t_j$
$D[i-1, j-1]$	$D[i-1, j]$	$D[i, j]$

$+0X+1$

$\downarrow$

$+1$

## CAMMINI MINIMI TRA TUTTE LE COPPIE DI NODI

Per trovare i cammini minimi tra tutte le coppie di nodi è possibile applicare in volte uno degli algoritmi visti in precedenza. Una svantaggio è che questo approccio risolve più volte lo stesso problema: ovvero trovare il cammino minimo tra  $(u, v)$  e tra  $(v, u)$ .

La programmazione dinamica permette di progettare un algoritmo di complessità inferiore e utilizzando i problemi già risolti.

Dato  $G = (V, E)$ , un cammino  $K$ -vincolato da  $u \in V$  è un cammino semplice da  $u \rightarrow v$  che non attraversa i nodi  $\{k, k+1, \dots, h\} - \{u, v\}$ .

Per i cammini  $K$ -vincolati vale la seguente proprietà di sottostruttura ottima: ogni sottocammino  $c_{ij}$  di un cammino  $K$ -vincolato  $c_{uv}$  in  $G$ , è esso stesso un cammino  $K$ -vincolato in  $G$ . Sia  $d^K(u, v)$  il costo di un cammino  $K$ -vincolato fra  $u \in V$ . Per la proprietà di sottostruttura ottima, per un cammino  $(K+1)$ -vincolato sono date le due possibilità:

$$d^{K+1}(u, v) = \min \left\{ \underline{d^K(u, v)}, d^K(u, x) + d^K(x, v) \right\}$$

Non passa per  $K$ , quindi il suo costo è dato dal cammino minimo  $K$ -vincolato.

Passa per il nodo  $K$ , allora è uguale al costo del cammino minimo  $K$ -vincolato tra  $u \in K$  e  $K \in V$ .

Si immagini di avere un filo di case. Dovete raccogliere fondi passando casa per casa sapendo che: la casa  $i$ -esima odia i propri vicini  $i+1$  e  $i-1$ , quindi non dovendo sapendo che ha donato uno di loro, e la casa  $i$ -esima può donare al max  $D[i]$ , calcolare la quantità massima di fondi che può essere raccolto.

$$C[i] = \begin{cases} 0 \\ \max \left\{ C[i-2] + D[i], C[i-1] \right\} \end{cases}$$

caso in cui prendo la donazione dalla casa  $i$ -esima, salto il suo vicino e vado alla casa dopo.

## INTERVALLI INDEPENDENTI: Dimostrazione GREEDY

In questo problema la scelta greedy consiste nell'ordinare gli intervalli per tempo di fine, e ad ogni iterazione aggiungere sempre l'intervalle con tempo di fine minore, indipendente con il precedente.

Per provare la correttezza si deve dimostrare che esiste una soluzione ottima che include il primo intervallo dopo l'ordinamento.

Sia  $S$  una soluzione ottima.

- se  $I_1 \in S$ : dimostrazione banale, non fare nullo.
- se  $I_1 \notin S$ :

Sia  $I_s$  l'intervalle con minor estremo finale in  $S$ .  
Sia  $S' = S - \{I_s\} \cup \{I_1\}$ ; poiché il tempo di fine di  $I_1$  è minore del tempo di fine di  $I_s$ , la sostituzione di  $I_s$  con  $I_1$  in  $S$  non modifica l'indipendenza dell'insieme  $S$ .

Quindi  $S'$  è una soluzione indipendente massimale con la scelta greedy.

## GREEDY

Il metodo greedy si può applicare a quei problemi di ottimizzazione che richiedono di selezionare un sottoinsieme ottimo di oggetti.

Un algoritmo greedy ordina dapprima gli oggetti in base ad un criterio di appetibilità, poi considera gli oggetti uno alla volta aggiungendo ogni volta il più appetibile. In altri termini effettua sempre la scelta che fornisce immediatamente il miglior risultato.

Affinché un algoritmo greedy fornisca la soluzione ottima, occorre siano verificate due proprietà:

(1) Sottostruttura ottima: occorre dimostrare che una soluzione ottima contiene al suo interno le soluzioni ottime dei sottoproblemi

(2) Scelta greedy: occorre dimostrare che la prima scelta greedy riduce il problema in un sotto problema più piccolo dello stesso tipo, e poi mostrare che il procedimento può essere esteso ad una sequenza di scelte greedy.

## INSIEME INDEPENDENTE DI INTERVALLI

- Dati  $n$  intervalli distinti  $I_1 = [a_1, b_1]$ , ...,  $I_n = [a_n, b_n]$  della retta reale, trovare un insieme indipendente massimo (ovvero un insieme di massima cardinalità formato da intervalli tutti disgiunti fra loro).

Si ordinano gli intervalli per estremi finali non decrescenti. Si inserisce il primo intervallo  $I_1$  e ci riconduciamo al sottoproblema dato da tutti gli intervalli che non intersecano  $I_1$ .

Questa scelta greedy è sempre corretta perché si riesce a dimostrare che esiste una soluzione ottima che include il primo intervallo.

L'algoritmo `independentSet()` ordina gli intervalli, fa la scelta greedy e poi scorre il vettore da sinistra a destra aggiungendo sempre l'intervallo compatibile. Complessità  $O(n \log n)$ .

```
SET independentSet(integer[] a, integer[] b)
```

```
{ordina a e b in modo che  $b[1] \leq b[2] \leq \dots \leq b[n]$ }
```

```
SET S  $\leftarrow$  Set()
```

```
S.insert(1)
```

```
integer ultimo  $\leftarrow$  1
```

% Ultimo intervallo inserito

```
for integer i  $\leftarrow$  2 to n do
```

```
    if  $a[i] \geq b[\text{ultimo}]$  then
```

% Controllo indipendenza

```
        S.insert(i)
```

```
        ultimo  $\leftarrow$  i
```

```
return S
```

Dimostrazione della scelta greedy nella pag precedente.

# UNIONE DI INSIEMI DISGIUNTI (MERGE FIND)

La struttura dati MFSET (Merge Find Set) opera su un insieme generativo  $S$ , il quale viene partitionato in una collezione di sotto insiemi in cui sono ammesse due operazioni:

- ricerca (find): permette di capire a quale componente appartiene un generico elemento in  $S$
- unione (merge): unisce due componenti distinte,  $x \in S, y \in S \rightarrow$  nuova componente  $x \cup y$ .

Non prevede operazioni di inserimenti o di cancellazione, ma tutti gli elementi vengono inseriti all'inizio, suddivisi in una componente per elemento e le operazioni di unione mantengono le componenti disgiunte.

## MFSET

% Crea  $n$  componenti  $\{1\}, \dots, \{n\}$   
MFSET Mfset(integer  $n$ )

% Restituisce il rappresentante della componente contenente  $x$   
integer find(integer  $x$ )

% Unisce le componenti che contengono  $x$  e  $y$   
merge(integer  $x$ , integer  $y$ )

- Mfset( $n$ ): crea un insieme di componenti disgiunte da  $1-n$ .

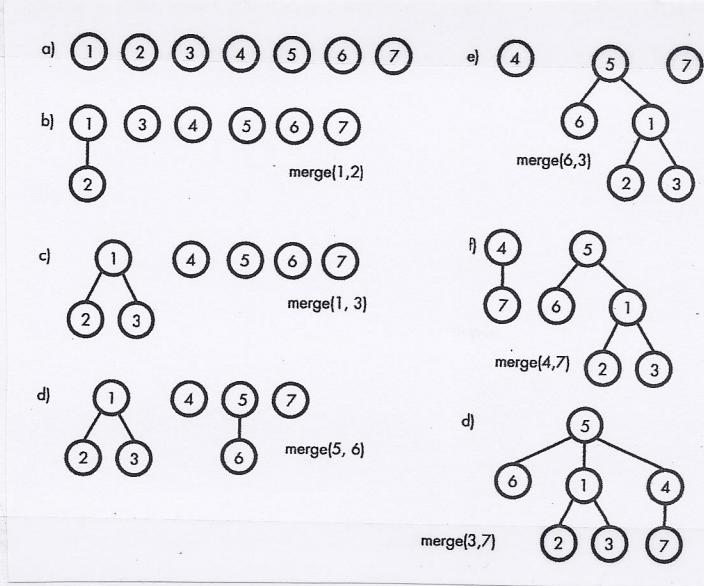
- Find( $x$ ): ritorna l'identificatore della componente di  $x$ . L'assegnamento dell'identificatore della componente avviene arbitrariamente tra gli elementi della componente.

## ESEMPIO - [MFSET]

Sia  $A = \{1, 2, 3, 4\}$ . MFSET restituisce  $S = \{\{1\}, \{2\}, \{3\}, \{4\}\}$ . Poiché  $\text{find}(1) \neq \text{find}(3)$ , eseguendo  $\text{merge}(1, 3)$  si ottiene  $S = \{\{1, 3\}, \{2\}, \{4\}\}$ . Adesso,  $\text{find}(4) \neq \text{find}(2)$  ed eseguendo  $\text{merge}(4, 2)$ , si ha  $S = \{\{1, 3\}, \{2, 4\}\}$ . Essendo  $\text{find}(3) \neq \text{find}(4)$ , l'esecuzione di  $\text{merge}(3, 4)$  dà per risultato  $S = \{\{1, 3, 2, 4\}\}$ . D'ora in poi, non sono più possibili ulteriori fusioni.

## REALIZZAZIONE BASATA SU FORESTA

Questa realizzazione prevede che gli elementi di ciascuna componente siano ospitati nei nodi di un albero; quindi inizialmente (senza nessuna operazione di merge o find) si avrà una foresta di alberi. Con questa rappresentazione, l'identificatore della componente è la radice dell'albero. La funzione  $\text{find}(x)$  non fa altro che risalire l'albero in cui  $x$  si trova con un percorso nodo-radice e restituisce la radice. La funzione  $\text{merge}(x, y)$  lavora in modo analogo, trovando la radice dell'albero a cui appartiene  $x$ , e aggancia un albero all'altro.



### ESEMPIO – [Realizzazione basata su foresta]

Sia dato un MSET  $S$  di 7 componenti, ciascuna contenente il solo intero  $i$ ,  $1 \leq i \leq 7$ .  $S$  è rappresentato con la foresta di Fig. 10.5(a). L'effetto su  $S$  dell'esecuzione della sequenza di operazioni  $\text{merge}(1, 2)$ ,  $\text{merge}(1, 3)$ ,  $\text{merge}(5, 6)$ ,  $\text{merge}(6, 3)$ ,  $\text{merge}(4, 7)$  e  $\text{merge}(3, 7)$  è mostrato nelle Figg. 10.5(b)-(g).

# EURISTICA SUL RANGO

La creazione dell'MFSET costa  $O(n)$ , mentre la funzione find e merge dipendono dall'altezza dell'albero che devono visitare; quindi se l'albero è tutto sbilanciato e contiene  $n$  nodi, avranno costo  $O(n^2)$ .

La complessità può essere abbassata scegliendo opportunamente quali tra i due alberi unire all'altro durante l'operazione di merge; generalmente si unisce il più "corto" a quello più radicato.

Questa informazione viene mantenuta nel vettore rango; se  $r_x$  è la radice dell'albero contenente  $x$ ,  $rango[r_x]$  contiene l'altezza di tale albero.

## MFSET

```
integer[] p
integer[] rango

Mfset(integer n)
    MFSET t ← new MFSET
    t.p ← integer[1 ... n]
    t.rango ← integer[1 ... n]
    for integer i ← 1 to n do
        t.p[i] ← i
        t.rango[i] ← 0
    return t
```

## merge(integer x, integer y)

```
rx ← find(x)
ry ← find(y)
if rx ≠ ry then
    if rango[rx] > rango[ry] then
        p[ry] ← rx
    else if rango[ry] > rango[rx] then
        p[rx] ← ry
    else
        p[rx] ← ry
        rango[ry] ← rango[ry] + 1
```

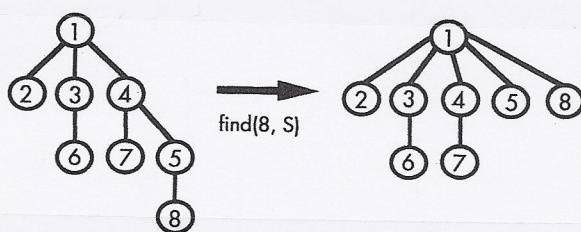
Complessità:  $O(n)$

Complessità  $O(\log n)$

## COMPRESSEIONE DEI CATTINI

Questa heuristica permette di rendere l'albero più compresso ad ogni operazione di `find()` per renderlo più accessibile successivamente.

L'idea è di modificare la `find()` in modo che ogni nodo che viene incontrato nel percorso di risalita alla radice, viene reso figlio della radice.



### ESEMPIO – [Compressione dei percorsi]

L'effetto dell'esecuzione di `find()` con la compressione dei percorsi è mostrato nella Fig. 10.6.

Questa heuristica rende la complessità amortizzata della `merge()` e della `find()` costante  $O(1)$

```
integer find(integer x)
  if p[x] ≠ x then
    | p[x] ← find(p[x])
  return p[x]
```

## COMPONENTI CONNESSE INCREMENTALI

L'applicazione dell'MFSET più naturale è quando si devono calcolare le componenti connesse di un grafo  $G$  composto da nodi non connessi fra loro, al quale si vuole aggiungere un arco.

Per mantenere le componenti connesse sempre aggiornate dovremo fare una visita in profondità per ogni nuovo nodo arco inserito:  $O(m(n+m))$ .

Se si usa invece un MFSET con l'euristica sul vampo e compressione dei cammini, l'aggiunta di un nuovo arco si riduce alla ricreazione dell'MFSET sul nuovo grafo aggiornato ( $O(n)$ ) e di  $m$  operazioni di merge di costo costante ( $O(m)$ ), quindi in totale otterremo una complessità di  $O(m+n)$  come una visita in profondità.

MFSET CC(GRAPH G)
MFSET $M \leftarrow \text{Mfset}(\{G, V\})$
foreach $u \in G.V$ do
foreach $v \in G.\text{adj}(u)$ do
$M.\text{merge}(u, v)$
return $M$

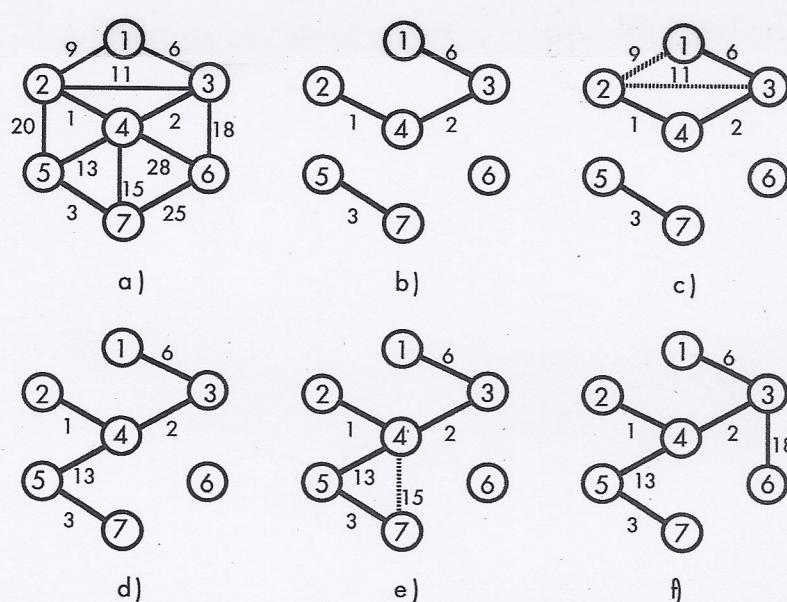
## MINIMO ALBERO DI COPERTURA

- Minimum Spanning Tree: Dato un grafo non orientato e connesso  $G = (V, E)$ , con pesi  $w(u, v)$  non negativi. Scegli archi, trovare un albero di copertura  $G' = (V, T)$  per  $G$ , cioè un albero avente tutti i nodi di  $V$ , ma solo  $n-1$  archi degli  $m$  archi in  $E$ ; t.c. la  $\sum_{(u,v) \in T} w(u,v)$  sia minima.

Può essere risolto con gli algoritmi di Prim e Kruskal utilizzando tecniche greedy.

## ALGORITMO DI KRUSKAL

Questo algoritmo ~~seleziona~~ ~~abilitizza~~ gli archi in ordine crescente di peso, e viene aggiunto all'albero  $T$  se non forma circuiti con gli archi già inseriti. Lo sforzo computazionale sta nell'ordinare gli archi, in quanto l'aggiunta di un nuovo arco all'albero richiede costo  $O(n)$  se si utilizza un TIFSET.



Algoritmo di Kruskal; a) Il grafo di ingresso. b) la foresta di copertura dopo l'inclusione di [2,4], [3,4], [5,7] e [1,3]; c) [1,2] e [2,3] non sono inclusi; d) inclusione di [4,5]; e) [4,7] non è incluso; f) inclusione di [3,6] e soluzione ottima.

Per risparmiare iterazioni, la variabile  $c$  conta il numero di archi inseriti nell'albero, che possono essere al massimo  $n-1$ .

### SET kruskal(ARCO[] A, integer n, integer m)

```

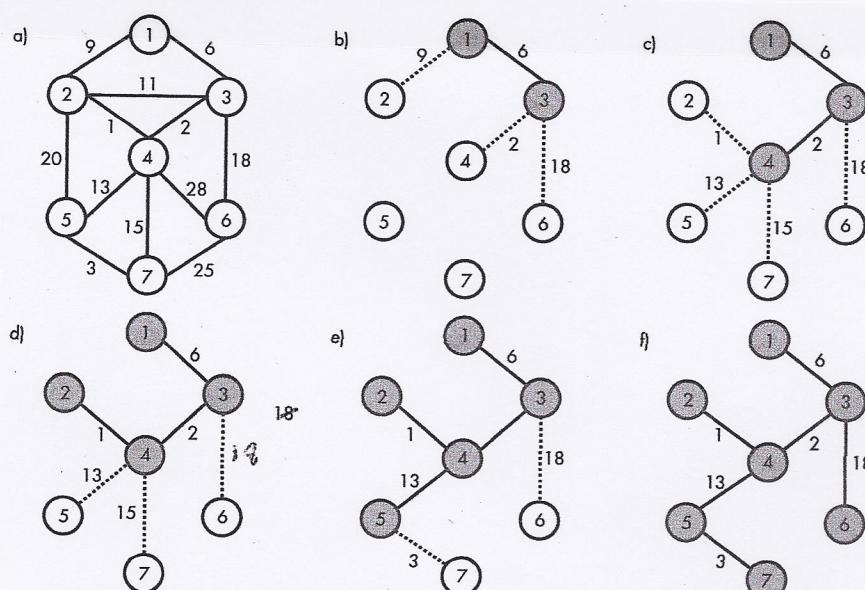
SET T ← Set()
MFSET M ← Mfset(n)
{ordina A[1, ..., m] in modo che A[1].peso ≤ ... ≤ A[m].peso}
integer c ← 0
integer i ← 1
while c < n - 1 and i ≤ m do
    if M.find(A[i].u) ≠ M.find(A[i].v) then
        M.merge(A[i].u, A[i].v)
        T.insert(A[i])
        c ← c + 1
    i ← i + 1
    % Termina quando l'albero è costruito

```

L'albero  $T$  è restituito come insieme di archi.  
Complessità  $O(m \log n)$ .

## ALGORITMO DI PRIM

L'algoritmo di Prim differisce da quello di Kruskal in quanto fa crescere un albero radicato da un nodo arbitrario del grafo, aggiungendo all'albero sempre l'arco con peso minimo che collega un nodo che non è in  $T$  (evitando cicli).



- .3 Algoritmo di Prim. I nodi in grigio sono quelli estratti dalla coda. Gli archi tratteggiati sono quelli che "escono" da  $T$ ; gli altri archi rappresentano  $T$ . a) Il grafo di ingresso; b) l'albero dopo aver estratto 1 e 3; c), d), e) l'albero dopo aver estratto 4, 2, 5, rispettivamente; f) soluzione ottima.

L'algoritmo utilizza una coda con priorità (min-heap) per estrarre sempre l'arco con peso minore e un settore pos per indicare se un arco deve ancora essere estratto dalla coda o meno. Ad ogni nodo  $v$  è associata la priorità per raggiungere quel nodo; in questo modo se si trova un arco con minore peso  $w(u, v)$  per raggiungere  $v$ , si aggiorna la priorità con la funzione decrease sul nodo  $u$ , in modo da farlo estrarre al prossimo giro. Complessità  $\mathcal{O}(m \log n)$ .

```

prim(GRAPH G, integer r, integer[] p)
  PRIORITYQUEUE Q ← MinPriorityQueue()
  PRIORITYITEM[] pos ← new PRIORITYITEM[1 ... G.n]
  foreach  $u \in G.V() - \{r\}$  do
    pos[u] ← Q.insert( $u, +\infty$ )
  pos[r] ← Q.insert( $r, 0$ )
  p[r] ← 0
  while not Q.isEmpty() do
     $u \leftarrow Q.deleteMin()$ 
    pos[u] ← nil
    foreach  $v \in G.adj(u)$  do
      if  $pos[v] \neq \text{nil}$  and  $w(u, v) < pos[v].priorità$  then
        Q.decrease(pos[v],  $w(u, v)$ )
        p[v] ← u
  
```

## RICERCA LOCALE

Un modo per cercare di risolvere problemi di ottimizzazione può essere quello di partire da una soluzione ammessa e cercarne un'altra nell'intorno della precedente.

DATA UNA SOLUZIONE  $SOL$  DEL PROBLEMA, SI DEFINISCE UN INTORNO  $I(SOL)$  DI QUESTA SOLUZIONE COME UN INSIEME DI SOLUZIONI AMMESSIBILI CHE SONO OTTENIBILI DA  $SOL$  IN BASE A CERTI CRITERI DI TRASFORMAZIONE. SE GUARDANDO IN  $I(SOL)$  SI TROVA UNA SOLUZIONE MIGLIORE DI  $SOL$ , ALLORA QUELLA SOVRÀ LA NUOVA SOLUZIONE.

IL PROCESSO DI RICERCA SI ARRESTA SE NON È POSSIBILE MIGLIORARE ULTERIORMENTE.

ricercaLocale()

$SOL \leftarrow$  UNA SOLUZIONE AMMESSIBILE  
while  $\exists S \in I(SOL)$  che è migliore  
 $SOL \leftarrow S$

return  $SOL$

In generale si arresta su una soluzione che è un ottimo locale, ma che può non essere l'ottimo globale. Partendo da soluzioni iniziali diverse si possono raggiungere ottimi locali diversi.

$$o = (x_1, z, \{x_2\} - \{x_3\})$$

no stoy di tutti smos obrov ozzut ib smoldong imp  
ozut ib smolot di Tchuo smos smotir s ozut

## PROBLEMI DI FLUSSO - FLUSSO MASSIMO

Per questo tipo di problema si utilizza un grafo  $G = (V, E)$  orientato con una capacità associata agli archi. L'obiettivo è descrivere un problema che massimizzi l'uso della capacità.

### DEFINIZIONI

- Rete di Flusso:  $G = (V, E, s, p, c)$

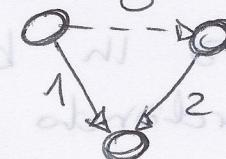
$V$ : insieme dei vertici

$E$ : insieme degli archi

$s$ : nodo sorgente

$p$ : nodo pozzo

$c$ : funzione di capacità  $V \times V \rightarrow \mathbb{R}^{>0}$



- Flusso in  $G$ :  $f: V \times V \rightarrow \mathbb{R}$  che soddisfa:

- vincolo di capacità: per ogni arco il flusso passante non può essere maggiore o uguale alla sua capacità.

$$\forall (u,v) \in V, f(u,v) \leq c(u,v)$$

- antisimmetria: per ogni flusso di ogni arco  $(u,v)$  esiste un flusso di uguale capacità ma di direzione opposta.

$$\forall (u,v) \in V: f(u,v) = -f(v,u)$$

- conservazione del flusso: in un nodo la somma del flusso entrante deve essere uguale alla somma del flusso uscente.

$$\forall (u,v) - \{s,p\}: \sum_{v \in V} f(u,v) = 0$$

Ogni problema di flusso prende come INPUT la rete di flusso e ritorna come OUTPUT la funzione di flusso.

Dalle proprietà della funzione di flusso si definiscono:

- valore del flusso: somma di tutti i flussi degli archi uscenti dalla sorgente.

$$|f| = \sum_{(s,v) \in E} f(s,v)$$

- valore del flusso massimo: massimo fra tutti i valori del flusso

$$|f^*| = \max\{|f|\}$$

- Somma dei flussi:

$$(fg)(u,v) = f(u,v) + g(u,v)$$

- capacità residua: ogni arco ha una capacità, e il flusso "consuma" tutta, o una parte, di questa capacità; quindi la capacità rimanente dopo il passaggio del flusso è definita come una funzione:

$$cr: V \times V \rightarrow \mathbb{R}^{>0} \text{ t.c. } cr(u,v) = c(u,v) - f(u,v)$$

- rete di flusso residuo: è un grafo che ha lo stesso insieme di nodi (stessa sorgente e pozzo) di  $G$  e l'insieme degli archi  $E_r$  contiene un arco  $(u,v)$  con capacità  $cr(u,v) \neq 0$   $\forall (u,v) \in V$  t.c.  $cr(u,v) > 0$ .

$$G_r = (V, E_r, s, p, cr) \quad (u,v) \in E_r \Leftrightarrow cr(u,v) > 0$$

$$(u,v) \in E_r \Leftrightarrow cr(u,v) > 0$$

È possibile sintetizzare la ricerca di flusso come segue:

$f \leftarrow f_0$  // situazione iniziale con flusso nullo

$c_r \leftarrow c$  // se  $f=f_0$  allora la capacità residua = capacità

REPEAT

$\leftarrow g$  un flusso  $(V, E_r, s, p, c_r)$  per la rete residua corrente

/\*

questa situazione si tradurrà in una visita nella rete di flusso residua per trovare l'arco con capacità minore; quindi sarà quella quantità di flusso massima che potrà passare per quel determinato cammino.

\*/

$f \leftarrow f + g$  // aggiorno il mio flusso

$c_r \leftarrow$  calcola la nuova rete residua

UNTIL  $g = f_0$  //  $g = f_0$  quando non si riesce a trovare un cammino  $(s, p)$ .

Per esempio nel grafo  $G = (V, E)$  con  $V = \{v_1, v_2, v_3, v_4, v_5\}$  e  $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_3, v_5), (v_4, v_5)\}$  e capacità  $c_e$  per ogni arco  $e \in E$ , si vuole trovare un cammino da  $v_1$  a  $v_5$ . Si parte da  $v_1$  e si considera il cammino  $v_1 - v_2 - v_4 - v_5$ , che è un cammino dalla sorgente a  $v_5$  ma non è un cammino massimale perché ci sono archi da  $v_1$  a  $v_3$  e da  $v_3$  a  $v_5$  che non sono stati utilizzati.

Si considera allora il cammino  $v_1 - v_3 - v_5$ , che è un cammino massimale dalla sorgente a  $v_5$  perché non ci sono archi da  $v_1$  a  $v_3$  e da  $v_3$  a  $v_5$  che non sono stati utilizzati.

## DIMOSTRAZIONE CHE IL FLUSSO È MASSIMO

Lo schema mostrato prima restituisce sempre il flusso massimo perché questo tipo di problemi è strettamente correlato a quello di trovare un taglio minimo sullo stesso grafo.

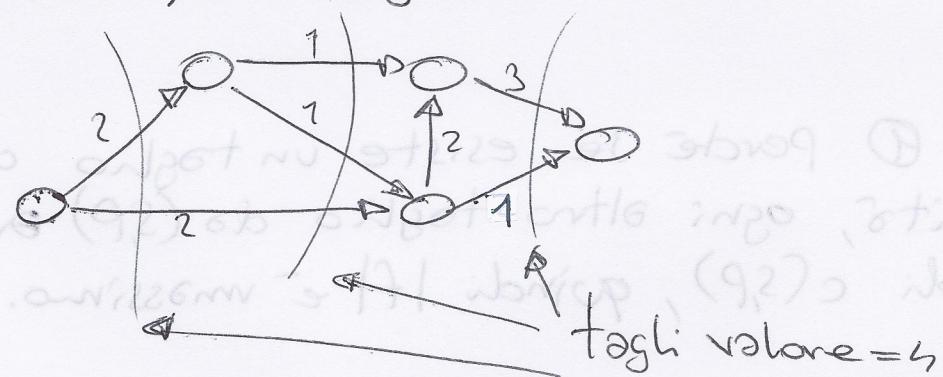
Si definisce un taglio  $(S, P)$  come una partizione dell'insieme dei nodi t.c.  $s \in S$  e  $p \in P$ ,  $S \cup P = V$  e  $S \cap P = \emptyset$ . La capacità del taglio  $c(S, P)$  è

$$c(S, P) = \sum_{\substack{u \in S \\ v \in P}} c(u, v)$$

Un taglio di capacità minima è detto minimo. Se  $f$  è un flusso ed  $(S, P)$  è un taglio allora il flusso che attraversa il taglio è

$$f(S, P) = \sum_{\substack{u \in S \\ v \in P}} f(u, v)$$

Il valore di un flusso è uguale al flusso che attraversa qualunque taglio:



Se vale che  $f(S, P) = c(S, P)$  allora  $f$  è il flusso massimo e  $(S, P)$  è un taglio minimo.

## TEOREMA DEL FIUSO MASSIMO - TAGLIO MINIMO

Le condizioni seguenti sono equivalenti:

①  $f$  è un flusso massimo

①  $\Rightarrow$  ②

② non esiste un cammino avventante per  $G_f$

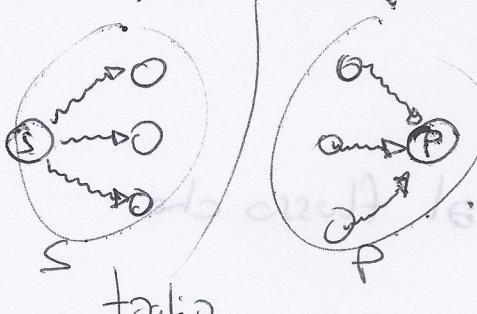
②  $\Rightarrow$  ③

③  $\exists$  taglio  $(S, P)$  con  $c(S, P) = |f|$

③  $\Rightarrow$  ①

① implica ② perché se esiste un cammino avventante, posso aggiornare il mio flusso con il valore del cammino avventante, quindi quello che ottengo è un nuovo flusso massimo, il che contraddice l'ipotesi che  $f$  sia massimo (un assurdo).

② implica ③ perché, assumendo l'ipotesi ② si vengono a creare due componenti del grafo  $G_f$  che comprendono  $s$  e  $p$  rispettivamente, quindi esiste un taglio  $(S, P)$  che le divide con  $|f| = f(S, P) = c(S, P)$



③ implica ① perché se esiste un taglio con flusso = alla capacità, ogni altro taglio da  $(S, P)$  avrà come flusso < di  $c(S, P)$ , quindi  $|f|$  è massimo.

## ALGORITMO DI FORD-FULKERSON

Questo tipo di algoritmo cerca un cammino aumentante usando qualsiasi tipo di visita, assumendo che costi  $O(n+m)$  e che le capacità siano intere.

Essendo che ogni cammino aumentante aumenta il valore del flusso di una quantità  $\geq 1$ , possono essere effettuati al massimo  $|f^*|$  incrementi; quindi si ottiene  $O(|f^*|(m+n))$

## ALGORITMO DI EDMOND-KARP

Molto simile a Ford-Fulkerson ma utilizza una BFS.  
Complessità  $O(mn^2)$