

ottiene una funzione di incertezza che rappresenta la
probabilità di avere un dato valore.

STRUTTURE DATI E PROGETTAZIONE

6.2 Strutture di ALGORITMI

Ogni algoritmo è composto da un insieme di operazioni.

Le operazioni sono: inserimento, eliminazione e ricerca.

6.1 Nella 6.1: si definisce un algoritmo per la ricerca binaria di un elemento in un insieme ordinato.

6.2 Nella 6.2: si definisce un algoritmo per la ricerca binaria di un elemento in un insieme ordinato.

6.3 Nella 6.3: si definisce un algoritmo per la ricerca binaria di un elemento in un insieme ordinato.

6.4 Nella 6.4: si definisce un algoritmo per la ricerca binaria di un elemento in un insieme ordinato.

6.5 Nella 6.5: si definisce un algoritmo per la ricerca binaria di un elemento in un insieme ordinato.

Nel progettare algoritmi si evidenziano quattro fasi nelle quali occorre passare:

(1) Classificazione del problema: se il problema ha caratteristiche comuni ad altri problemi:

- problemi decisionali: la cui risposta sia sì/no a seconda che il dato in ingresso soddisfi una certa proprietà

- problemi di ricerca: dove si vuole trovare una soluzione ammessa

- problemi di ottimizzazione: nei quali la soluzione ammessa è associata ad una misura e si vuole trovare la soluzione ottima.

(2) Caratterizzazione della soluzione: si prova a caratterizzare matematicamente il problema.

(3) Tecniche di progettazione: come chiavi-etc., imperi, programmazione dinamica, greedy..

(4) Strutture dati: si cerca di trovare un modo conveniente per strutturare i dati che permetta di migliorare l'esecuzione dell'algoritmo.

CAMMINI MINIMI

Dato un grafo orientato $G = (V, E)$, assumiamo che ogni arco $(u, v) \in E$ sia associato un costo $w(u, v)$. Dato un cammino $C = v_1, \dots, v_k$ ($k > 1$), il costo del cammino è definito da

$$w(C) = \sum_{i=2}^k w(v_{i-1}, v_i)$$

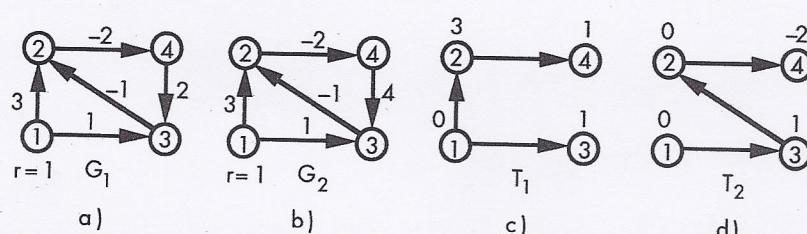
Dato il seguente problema: [CAMMINI MINIMI]

Dato un grafo $G = (V, E)$, una funzione di costo $w: E \rightarrow \mathbb{R}$ e un nodo r , trovare un cammino da r a v , $\forall v \in V$, il cui costo sia minimo, ovvero più piccolo o uguale del costo di qualunque altro cammino da r a v .

Una soluzione ammissibile è l'albero di copertura radicato in r , che include un cammino da r ad ogni altro nodo.

ESEMPIO – [Albero di cammini]

La Fig. 11.1 mostra un grafo G_1 che contiene un ciclo negativo ed un grafo G_2 privo di cicli negativi. Se $r = 1$, gli alberi T_1 e T_2 sono, rispettivamente, una soluzione ammissibile ed una ottima per G_2 .



11.1 a) Un grafo G_1 con un ciclo negativo; b) un grafo G_2 privo di cicli negativi; c) una soluzione ammissibile per G_2 ; d) una soluzione ottima per G_2 .

Sia T una soluzione ammissibile. Ogni nodo v del grafo è caratterizzato da un valore reale d_v , che indica la distanza di v da r , e che è uguale al costo del cammino tra r e v in T .

TEORIA DI BELLMAN

Una soluzione ammissibile T è ottima se volgono le seguenti condizioni:

- $d_v = d_u + w(u, v) \forall (u, v) \in T$
- $d_u + w(u, v) \geq d_v \forall (u, v) \in E$

Questo permette di formulare un algoritmo che verifica arco-per-arco, il teorema di Bellman.

camminiMinimi(GRAPH G, NODE r, integer[] T)

```
integer[] d ← new integer [1 ... G.n]
boolean[] b ← new boolean [1 ... G.n]
foreach  $u \in G.V() - \{r\}$  do
     $T[u] \leftarrow \text{nil}$ 
     $d[u] \leftarrow +\infty$ 
     $b[u] \leftarrow \text{false}$ 
 $T[r] \leftarrow \text{nil}$ 
 $d[r] \leftarrow 0$ 
 $b[r] \leftarrow \text{true}$ 
① STRUTTURADATI  $S \leftarrow \text{StrutturaDati}(); S.\text{aggiungi}(r)$ 
while not  $S.\text{isEmpty}()$  do
    ②  $u \leftarrow S.\text{estrai}()$ 
     $b[u] \leftarrow \text{false}$ 
    foreach  $v \in G.\text{adj}(u)$  do
        if  $d[u] + w(u, v) < d[v]$  then
            if not  $b[v]$  then
                ③  $S.\text{aggiungi}(v)$ 
                 $b[v] \leftarrow \text{true}$ 
            else
                ④ % Azione da intraprendere nel caso  $v$  sia già presente in  $S$ 
                 $T[v] \leftarrow u$ 
                 $d[v] \leftarrow d[u] + w(u, v)$ 
```

Il vettore d contiene le distanze di tutti i nodi dall'origine r , T è un vettore di padri, che viene modificato in modo da ottenere l'albero dei cammini minimi, infine b è un vettore di booleans per segnare se un nodo v è nella struttura dati S .

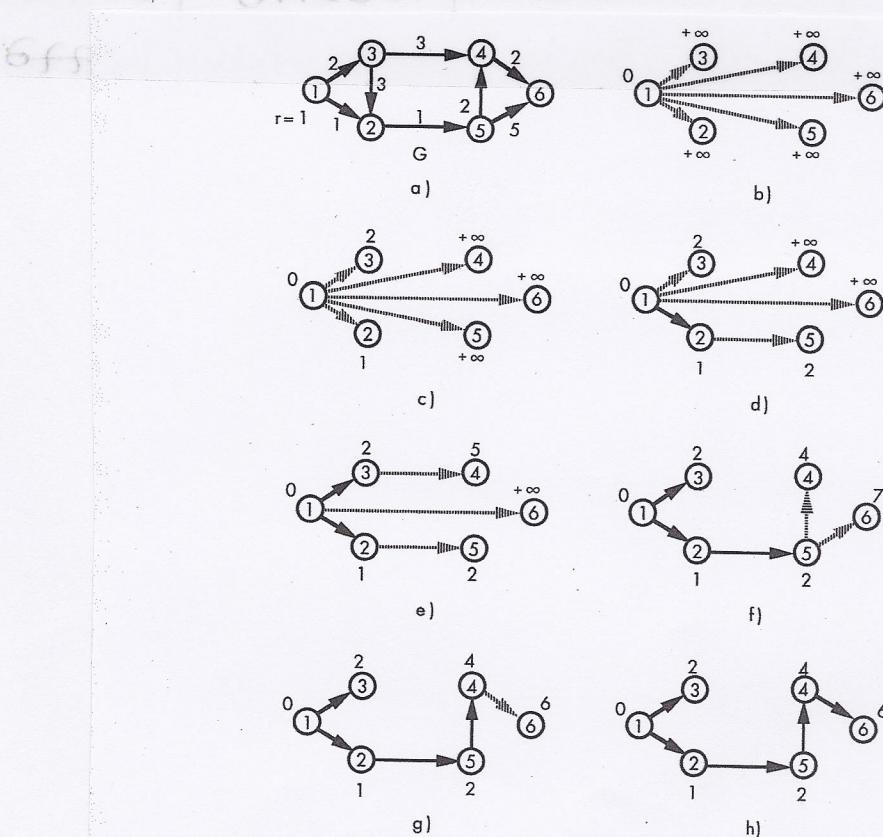
Questa procedura non è ottima: è superpolinomiale.

ALGORITMO DI DIJKSTRA

Se S è una coda a priorità si ottiene l'algoritmo di DIJKSTRA. Gli elementi della coda sono nodi del grafo e le loro priorità corrispondono alla distanza da r . Essendo che l'operazione $\text{extract}()$ si riduce ad una $\text{deleteMin}()$, ad ogni iterazione è estratto il nodo con $d[u]$ minima.

- (1) $\text{PRIORITYQUEUE } S \leftarrow \text{PriorityQueue}(); S.\text{insert}(r, 0)$
- (2) $u \leftarrow S.\text{deleteMin}()$
- (3) $S.\text{insert}(v, d[u] + w(u, v))$
- (4) $S.\text{decrease}(v, d[u] + w(u, v))$

Essendo che la $\text{deleteMin}()$ viene eseguita su tutti i nodi del grafo, la complessità è $\mathcal{O}(n^2)$



11.2 Algoritmo di Dijkstra. a) Un grafo G con lunghezze non negative; b) inizializzazione ad un albero fittizio per $r = 1$; c) - h) aggiornamento dell'albero e delle distanze, estraendo i nodi nell'ordine: 1, 2, 3, 5, 4, 6.

ALGORITMO DI JOHNSON

Qualora la struttura S sia una coda a priorità realizzata con heap binario, si ottiene l'algoritmo proposto da D.B. Johnson (1977). L'algoritmo ha complessità $\mathcal{O}(m \log n)$ dove m è il numero di archi presenti nel grafo. I grafi sparsi, dove $m = O(n)$ l'algoritmo di Johnson risulta più efficiente di quello di Dijkstra; per grafi densi, dove $m = \Omega(n^2)$, l'algoritmo di Johnson è più lento.

ALGORITMO DI BELLMAN-FORD-FOORSE

Se la struttura dati S è una coda si ottiene un algoritmo che ha complessità polinomiale, anche in presenza di archi con lunghezza negativa.

- (1) QUEUE $S \leftarrow \text{Queue}(); S.\text{enqueue}(r)$
- (2) $u \leftarrow S.\text{dequeue}()$
- (3) $S.\text{enqueue}(v)$
- (4) Sezione non necessaria

Ha complessità $\mathcal{O}(n(m+n)) = \mathcal{O}(nm)$ qualsiasi sia il segno degli archi.

11.4 ESEMPIO – [Algoritmo di Bellman-Ford-Moore]

La Fig. 11.3 mostra l'esecuzione dell'algoritmo di Bellman-Ford-Moore su un grafo G per $r = 1$. L'ordine di estrazione dei nodi da S è: 1 (passata 0); 2, 3 (passata 1); 4, 2 (passata 2) e 4 (passata 3).

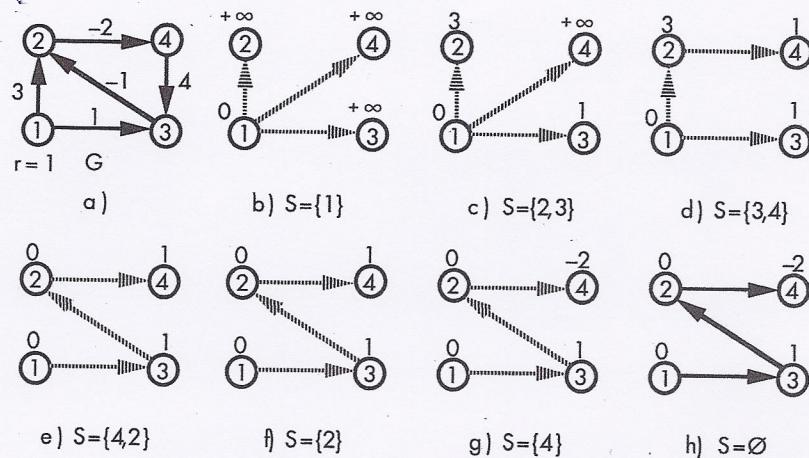


Fig. 11.3 Algoritmo di Bellman-Ford-Moore. a) Un grafo G ($r = 1$); b)-g) estrazione dalla coda dei nodi nell'ordine: 1, 2, 3, 4, 2, 4; h) la soluzione ottima.

DIVIDE - ET - IMPERA

Questa tecnica consente di partizionare il problema in sottoproblemi più piccoli dello stesso tipo, risolverli e ricombinarli con poco sforzo. Questa tecnica permette di impostare facilmente l'equazione di ricorrenza:

$$T(n) = \begin{cases} C & n \leq K \\ D(n) + C(n) + \sum_{i=1}^n T(n_i) & n > K \end{cases}$$

dove C è una costante, mentre $D(n)$ e $C(n)$ sono il numero di operazioni per dividere il problema e ricombinarlo rispettivamente.

RICERCA BINARIA E RICERCA PER INTERPOLAZIONE

La funzione `binarySearch()` è un esempio di divide-et-impera, con la particolarità che la chiamata ricorsiva arriverà in uno solo dei due sottoproblemi in cui viene diviso il problema originario. Una regola importante in questi casi è: se non tutti i sottoproblemi devono essere analizzati, è buona norma affrontare ricorsivamente i problemi più piccoli possibili.

Si può affinare ancora di più la ricerca, tenendo conto dei valori delle chiavi e della loro distribuzione di probabilità.

Dovendo cercare la chiave K nel vettore $A[1..n]$, anziché provare nella posizione centrale è ragionevole pensare in quella più vicina a

$$\frac{n(K - K_{\min})}{(K_{\max} - K_{\min})}$$

dove le chiavi sono distribuite nell'intervallo $[K_{\min}, K_{\max}]$. Infatti il loro rapporto indica di quanto il valore di K si discosta dai valori estremi.

Se $K = K_{\min}$, allora il rapporto è zero e quindi conviene cercare nella prima posizione di A , mentre se $K = K_{\max}$, il rapporto è 1, quindi conviene cercare nell'ultima posizione di A .

Il metodo di ricerca risultante è detto di interpolazione.

Sotto l'ipotesi di distribuzione delle chiavi, la complessità di `ricercaInterpolata()` è $O(\log \log n)$.

QUICKSORT

È basato su divide-et-impara ma differisce dal mergesort nel modo in cui divide e ricombina i risultati.

La strategia consiste nel selezionare un elemento detto perno attorno al quale riarrangiare gli elementi.

Tutti gli elementi più piccoli del perno vengono spostati prima di lui nel vettore, mentre gli elementi più grandi sono spostati in posizioni che lo seguono.

Lo stesso algoritmo viene poi applicato alle due porzioni di elementi minori o maggiori del perno.

Sia $A[\text{primo-ultimo}]$ la porzione di A che l'algoritmo deve ordinare:

QuickSort(ITEM[] A, integer primo, integer ultimo)

```
if primo < ultimo then
    integer j ← perno(A, primo, ultimo)
    QuickSort(A, primo, j - 1)
    QuickSort(A, j + 1, ultimo)
```

La procedura perno() utilizza due cursori: i punta al primo elemento y non ancora esaminato e j da la posizione finale in cui va spostato il perno. Ad ogni iterazione se $y \geq x$, allora y non viene spostato, mentre se $y < x$, allora y viene scambiata con il primo elemento fra quelli maggiori o uguali a x. Al termine x è scambiato con $A[j]$.

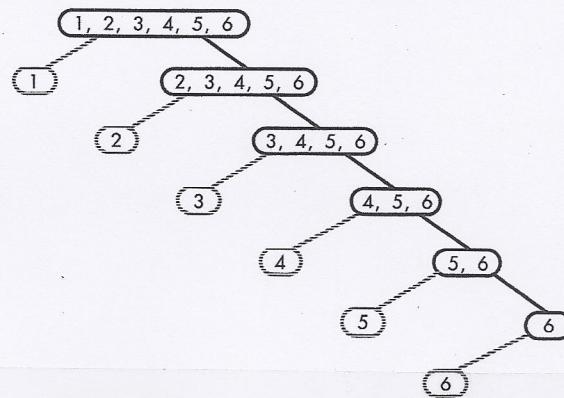
integer perno(ITEM[] A, integer primo, integer ultimo)

```
ITEM x ← A[primo]
integer j ← primo
for integer i ← primo to ultimo do
    if A[i] < x then
        j ← j + 1
        A[i] ↔ A[j]
    A[primo] ← A[j]
    A[j] ← x
return j
```

1	2	3	4	5	6	7	8	9	10
9	12	8	18	6	13	11	3	5	10
i		i							
9	8	12	18	6	13	11	3	5	10
i			i						
9	8	6	18	12	13	11	3	5	10
i					i				
9	8	6	3	12	13	11	18	5	10
i						i			
9	8	6	3	5	13	11	18	12	10
i							i		
5	8	6	3	9	13	11	18	12	10

12.5 ESEMPIO – [Caso pessimo QuickSort()]

La Fig. 12.3 mostra la partizione dei dati fortemente sbilanciata indotta eseguendo QuickSort() su un vettore già ordinato.



Nel caso pessimo è lento come l'insertionSort ma avviene solo se il vettore è ordinato. Questo è dovuto al forte sbilanciamento dei due insiemi su cui vengono fatte le chiamate ricorsive.
Nel caso medio la complessità è $O(n \log n)$.