Guida alla realizzazione di algoritmi "branch-and-bound"

Giovanni Righini

9 Marzo 2000

Questa guida è una traccia per aiutare lo studente a realizzare algoritmi di branch-and-bound. Mi servo a mo' di esempio del problema del commesso viaggiatore asimmetrico (ATSP) in Pascal. Le indicazioni fornite qui hanno valore puramente didattico, non affrontano il problema di come massimizzare l'efficienza del codice e delle strutture-dati. Sono possibili realizzazioni più sofisticate e più efficienti (e naturalmente più complesse).

L'esposizione è divisa in due parti:

Parte I: Algoritmo di ricerca su albero

Parte II: Bounding

Parte I: Algoritmo di ricerca su albero

Esistono almeno tre modi di realizzare un algoritmo di ricerca su albero:
a) ricorsivo
b) iterativo con backtracking
c) iterativo senza backtracking

Nei primi due casi la ricerca avviene con politica depth-first. Nel terzo caso è possibile modificare la politica di esplorazione dell'albero, modificando il criterio di ordinamento di una lista a puntatori.

Nel seguito viene illustrata quest'ultima tecnica. Ogni nodo dell'albero ha diversi nodi successori; nella terminologia convenzionale si dice anche che ogni nodo è "padre" di diversi nodi "figli", poiché i nodi di livello *k+1* vengono generati in seguito all'esame di un nodo di livello *k*. L'unico nodo che non ha un "padre" è la radice dell'albero, che viene generata esplicitamente; i nodi che non hanno "figli" sono invece le "foglie" dell'albero. L'idea è di non rappresentare tutto l'albero, ma soltanto i nodi che non hanno ancora generato i loro successori. Ogni volta che un nodo "padre" viene esaminato e vengono generati i suoi nodi "figli", esso viene cancellato poiché non verrà più usato nella ricerca. I nodi che non hanno ancora generato i "figli" sono detti anche "nodi aperti". Mantenendo in una lista a puntatori l'elenco di tutti i nodi aperti è possibile scegliere iterativamente uno di essi, toglierlo dalla lista, esaminarlo, generarne i "figli", inserire i "figli" nella lista dei nodi aperti e infine cancellarlo. Quando viene generato un nodo "foglia" dell'albero, esso non viene inserito nella lista dei nodi aperti, poiché non deve generare successori. I nodi "foglia" corrispondono alle soluzioni complete.

Per accompagnare la spiegazione con un esempio, ho scelto di illustrare passo dopo passo un algoritmo per il problema del commesso viaggiatore asimmetrico (ATSP). Sono date le distanze tra *N* città e si vuole trovare il circuito Hamiltoniano di lunghezza minima, cioè il percorso di lunghezza minima che parte da una città, passa una volta per ogni città e torna alla città di partenza. L'esempio è illustrato in Pascal.

I.1 Dati del problema

Supponiamo di aver memorizzato le distanze tra le *N* città in una matrice *D* di dimensioni *N*x*N* di numeri reali, dichiarata così:

```
const N = ...;
var D: array [1..N,1..N] of real;
```

I.2 Parte dichiarativa

Occorre innanzitutto scegliere la struttura-dati sufficiente a rappresentare ogni nodo dell'albero, cioè ogni sottoproblema.

Per farlo bisogna definire le variabili del problema. Una scelta naturale nel caso dell'ATSP è di usare un array di *N* elementi, ciascuno dei quali indica una tappa del percorso. Poichè ogni soluzione (circuito) può essere rappresentata in *N* modi diversi, si può decidere di fissare come città di partenza e arrivo la città numero 1. La città 1 viene quindi scritta nella prima posizione dell'array.

Una soluzione parziale perciò è rappresentata da un array parzialmente riempito, da un indice che ne rappresenta il livello di riempimento, cioè il numero di elementi significativi, e da un valore che rappresenta il costo parziale. L'array rappresenta le tappe già scelte; il livello di riempimento indica il

numero di tappe già scelte e rappresenta il livello dell'albero cui il nodo appartiene (l'albero ha un nodo di livello 1, che è la radice, e *N!* nodi di livello *N* che sono le soluzioni complete); il costo parziale è la somma delle lunghezze dei tratti da percorrere corrispondenti alle tappe già scelte e viene incrementato ogni volta che si sceglie una tappa in più. Al completamento di tutta la sequenza bisogna ricordarsi di aggiungere la lunghezza dell'ultimo tratto, quello che dall'ultima città visitata torna alla città 1.

Le strutture-dati necessarie a rappresentare la soluzione parziale devono essere inserite in un record, che a sua volta fa parte di una lista a puntatori, la lista dei nodi aperti. Perciò una possibile dichiarazione nel nostro esempio è la seguente:

```
type punt = ^rec;
     rec = record
             seq: array [1..N] of integer;  /*soluzione parziale*/
             liv: integer;                  /*livello*/
             val: real;                     /*valore parziale (costo)*/
             next: punt;
           end;
```

Capita spesso che successivamente sia necessario tornare alla dichiarazione ed aggiungere altre strutture-dati, cioè altri campi alla definizione del tipo del record (capiterà anche in questo esempio).

Bisogna poi dichiarare la variabile che funge da testa della lista a puntatori.

```
var tree: punt;      /* lista dei nodi aperti */
```

I.3 Parte esecutiva

Si divide in due parti, a parte le procedure di I/O: l'inizializzazione e la ricerca.

```
Begin
/* procedure di input */
Inizializza;
Ricerca;
/* procedure di output */
End;
```

I.3.1 Inizializzazione

Nella procedura di inizializzazione bisogna generare esplicitamente il primo nodo dell'albero, cioè la radice. Nel nostro caso:

```
procedure Inizializza;
begin
/* inizializzazione radice */
New (tree);
tree^.liv := 1;
tree^.seq[1] := 1;
tree^.val := 0.0;
tree^.next := nil;
end;
```

Dopo questa inizializzazione esiste un solo nodo aperto ed è la radice dell'albero.

Ad ogni eventuale successiva modifica della definizione del tipo `rec` deve seguire ovviamente la corrispondente modifica della procedura di inizializzazione.

I.3.2 Ricerca

La ricerca consiste nella ripetizione ciclica delle operazioni indicate in precedenza: estrazione del "padre", generazione dei "figli", inserimento in lista dei "figli" (se non sono nodi "foglia"), cancellazione del "padre". La ricerca termina quando tutti i nodi dell'albero sono stati esplorati e hanno generato i loro successori. In tal caso la lista dei nodi aperti risulta vuota. Nell'esempio si ha perciò il ciclo:

```
Procedure Ricerca;

begin
while (tree <> nil) do
  begin
  /* estrazione del nodo padre */
  Estrai_padre (tree,padre);
  /* generazione dei nodi figli */
  Genera_figli (padre,tree);
  /* cancellazione del nodo padre */
  Cancella_padre (padre);
  end;
end;
```

La variabile `padre` rappresenta una soluzione parziale e deve essere stata dichiarata (all'interno della procedura `Ricerca`):

```
var padre: punt;
```

L'estrazione del nodo padre dalla lista dei nodi aperti è una banale operazione di estrazione di un elemento in testa ad una lista a puntatori e si realizza indipendentemente dal problema con le stesse tre istruzioni.

```
procedure Estrai_padre (var tree, padre: punt);
begin
padre := tree;
tree := padre ^.next;
padre ^.next := nil;
end;
```

Ancora più banale è la cancellazione del nodo padre, che si riduce ad una sola istruzione.

```
procedure Cancella_padre (var padre: punt);
begin
Dispose (padre);
end;
```

Per semplicità nel seguito queste ultime due procedure saranno sostituite da semplici blocchi di istruzioni.

Più complessa è la seconda delle tre operazioni: il numero di nodi "figli" da generare e il modo in cui vengono generati è fortemente dipendente dal problema in esame. La tecnica consigliata è di eseguire tante iterazioni di un ciclo quanti sono i nodi "figli" da generare in modo da dover considerare solo un nodo "figlio" in ogni iterazione. A questo proposito val la pena ricordare che è sconsigliabile

per motivi di efficienza ricondurre alberi n-ari ad alberi binari; se si possono usare variabili intere anziché 0-1 è meglio. Bisogna anche sottolineare che i metodi per realizzare il branching, cioè la scomposizione di un problema in sottoproblemi sono tanti: alcuni fissano una o più variabili, altri introducono vincoli mutuamente esclusivi. Nel nostro esempio il branching viene eseguito scegliendo la prossima tappa in tutti i modi possibili. Si tratta quindi di un branching su una variabile intera che può assumere *N* possibili valori diversi (non tutti ammissibili, perché non si può tornare in una città già visitata).

Per generare il nodo "figlio" conviene fare una copia del record del nodo "padre" e poi apportare le dovute modifiche. Nel nostro caso le modifiche consistono nell'aggiunta di una tappa alla sequenza di tappe del padre (array `seq`), nell'incremento di 1 del livello (variabile `liv`) e nell'incremento del valore della soluzione (variabile `val`).

```
procedure Genera_figli (padre: punt; /* in ingresso */
                        var tree: punt  /* in uscita */
                          );
  var figlio: punt;
      i: integer;
begin
for i:=2 to N do
  if (/*città i non ancora visitata*/)     /* test (*) */
    then begin
         New (figlio);
         /* copia nodo padre in nodo figlio */
         figlio^ := padre^;
         /* modifica il figlio rispetto al padre */
         figlio^.liv := figlio^.liv + 1;
         figlio^.seq[figlio^.liv] := i;
         figlio^.val := figlio^.val + D[padre^.seq[padre^.liv],
                                        figlio^.seq[figlio^.liv]];
         /* inserisci figlio nella lista dei nodi aperti */
         ...
         end;
end;
```

Come preannunciato, si scopre a questo punto che per generare i nodi "figli" è utile una struttura-dati in più: occorre infatti sapere quali città sono già state visitate in una soluzione parziale e quali non ancora. E' vero che ciò potrebbe essere desunto dall'esame dell'array `seq` dall'elemento 1 fino all'elemento `liv`, ma ciò implicherebbe una scansione dello stesso e quindi un rallentamento dell'algoritmo. Sembra più ragionevole invece associare ad ogni città una variabile booleana il cui valore indica se la città è già stata visitata o no nella soluzione parziale in esame. Bisogna quindi aggiungere la dichiarazione, l'inizializzazione e l'aggiornamento della variabile nei tre corrsipondenti punti del programma. La dichiarazione del tipo `rec` viene modificata con l'aggiunta di un altro campo:

```
type punt = ^rec;
     rec = record
             seq: array [1..N] of integer;  /*soluzione parziale*/
             liv: integer;                  /*livello*/
             val: real;                     /*valore parziale (costo)*/
             vis: array [1..N] of boolean;  /*città già visitate*/
             next: punt;
           end;
```

L'inizializzazione avviene nell'apposita procedura `Inizializza`:

```
procedure Inizializza;
  var i: integer;
begin
/* inizializzazione radice */
New (tree);
tree^.liv := 0;
tree^.val := 0.0;
tree^.next := nil;
tree^.vis[1] := true;
for i:=2 to N do tree^.vis[i] := false;
end;
```

Infine bisogna aggiungere la modifica della nuova struttura-dati nella procedura `Genera_figli`:

```
/* modifica il figlio rispetto al padre */
figlio^.liv := figlio^.liv + 1;
figlio^.seq[figlio^.liv] := i;
figlio^.val := figlio^.val + D[padre^.seq[padre^.liv],
                               figlio^.seq[figlio^.liv]];
figlio^.vis[i] := true;
```

L'array `vis` può quindi essere usato per il test indicato con (*), che era rimasto in sospeso e che diventa:

```
if (not padre^.vis[i])
```

Resta da realizzare l'inserimento del nodo "figlio" nella lista dei nodi aperti. Tale inserimento va fatto solo se il nodo figlio è destinato a generare altri nodi. I casi in cui ciò *non* avviene sono due: (1) il nodo "figlio" è una foglia dell'albero e quindi rappresenta una soluzione completa; (2) il nodo "figlio" rappresenta una soluzione parziale non conveniente. Il secondo di questi due casi viene esaminato nella parte II, quando viene introdotta la tecnica di bounding. Consideriamo per ora solo il primo dei due casi. Per sapere se il nodo "figlio" è una foglia dell'albero o no, basta controllare il valore del campo `liv`: se esso è pari a *N*, la soluzione è completa. In tal caso il suo valore deve essere incrementato con la lunghezza dell'ultimo tratto. Perciò:

```
/* inserisci figlio nella lista dei nodi aperti */
if (figlio^.liv = N)
  then begin /* nodo foglia */
       figlio^.val := figlio^.val +
                      D[figlio^.seq[figlio^.liv],1];
       Elabora_foglia (figlio);
       end
  else Inserisci (figlio, tree);
```

Se il nodo in esame è un nodo "foglia", esso viene elaborato da un'apposita procedura, che dipende dal tipo di problema che si vuole risolvere. Se si tratta di un problema di enumerazione, cioè si vogliono elencare tutte le soluzioni ammissibili o tutte quelle che soddisfano certi criteri, allora la procedura `Elabora_foglia` aggiunge la soluzione ad un apposito elenco di soluzioni trovate; se invece si tratta, come nel nostro esempio, di un problema di ottimizzazione, cioè si cerca la soluzione migliore tra tutte quelle ammissibili, allora la procedura esegue un confronto tra la soluzione trovata e la soluzione ottima corrente (cioè la migliore trovata in precedenza) e memorizza la migliore tra le due.

Nel nostro caso supponiamo quindi di mantenere una variabile globale `best`, che rappresenta la soluzione ottima corrente. Essa va dichiarata:

```
var tree, best: punt;
```

e inizializzata in `Inizializza`:

```
best := nil;
```

La procedura `Elabora_foglia` è quindi nel nostro esempio:

```
Procedure Elabora_foglia (var figlio: punt);
begin
if (best = nil)
  then begin /* prima soluzione completa trovata => ottima corrente*/
       best := figlio;
       figlio := nil;
       end
  else begin /* confronta con l'ottima corrente */
       if (figlio^.val < best^.val)
         then begin /* è migliore */
              Dispose (best);
              best := figlio;
              figlio = nil;
              end
         else begin /* non è migliore */
              Dispose (figlio);
              end;
end;
```

L'altra procedura indicata in `Genera_figli` è la procedura `Inserisci`, che ha il compito di inserire il nodo "figlio" appena generato nella lista dei nodi aperti. Se l'estrazione del nodo "padre" dalla lista ad ogni iterazione viene fatta per comodità sempre dalla testa della lista, la politica di esplorazione dell'albero dipende dalla politica di inserimento. In particolare vale la seguente corrispondenza:

Politica di inserimento in lista	Politica di esplorazione dell'albero
in testa	depth-first-search
in coda	breadth-first-search
ordinato	best-first-search

Il principale vantaggio della tecnica illustrata rispetto a quella ricorsiva e a quella iterativa con backtracking è proprio quello di poter scegliere la politica di esplorazione dell'albero più conveniente o addirittura di poterla cambiare a run-time, senza essere vincolati all'esplorazione depth-first. E' quindi ragionevole scegliere un criterio di ordinamento dei nodi aperti in base ad una cifra di merito che indica quanto un nodo è "promettente" in modo che la ricerca esplori prima i nodi più "promettenti", a qualunque livello essi siano nell'albero. La cifra di merito da usare deve essere una stima di quanto buona può essere una soluzione completa ottenibile completando la soluzione parziale in esame. Un criterio ragionevole ad esempio è quello di adottare come cifra di merito il valore del lower bound (in caso di problema di minimizzazione). Si veda in proposito la parte II. Non riporto di seguito il codice

della procedura `Inserisci`, dal momento che essa esegue solo una banale operazione di inserimento di un elemento in una lista a puntatori ordinata.

Parte II: Bounding

L'algoritmo di ricerca su albero sviluppato finora ha un tempo di esecuzione che tende ad aumentare molto rapidamente al crescere di *N*. Per limitare il numero di nodi da esaminare si associa ad ogni nodo un "lower bound", ossia un limite inferiore ("upper bound" se problema è di massimizzazione), che indica una stima per difetto del valore di tutte le soluzioni complete ottenibili come discendenti di quel nodo. Anche un "upper bound" ("lower bound" se il problema è di massimizzazione) deve essere noto e viene usato come termine di confronto per i lower bounds dei nodi. L'upper bound è il valore della soluzione ottima corrente. Se un nodo dell'albero ha un lower bound maggiore o uguale all'upper bound corrente, esso viene chiuso, cioè non viene più inserito nella lista dei nodi aperti. Infatti tutte le soluzioni complete che si otterrebbero da esso avrebbero un valore non migliore del valore della soluzione ottima corrente.

Per arricchire il programma con la parte relativa al bounding è quindi necessario disporre di strutture-dati da cui desumere i valori dell'upper bound e dei lower bounds.

II.1 Upper bound

Può fungere da upper bound il valore di qualsiasi soluzione ammissibile già nota. Si può ricalcolare spesso l'upper bound usando algoritmi approssimanti costruttivi a partire dalle soluzioni parziali rappresentate da ogni nodo, oppure limitarsi a considerare come upper bound il valore della soluzione ottima corrente. Nel primo caso si impiega più tempo di calcolo e si possono ottenere upper bounds più bassi e quindi più utili per chiudere alcuni nodi dell'albero; nel secondo caso invece non si usa tempo di calcolo aggiuntivo. Tra questi due estremi sono possibili innumerevoli variazioni. Per semplicità, illustro di seguito il programma nel secondo caso, assumendo come upper bound (banale) il valore della soluzione ottima corrente. Inizialmente, finché nessuna soluzione completa viene calcolata, il valore dell'upper bound viene posto uguale ad un numero molto grande. Un'alternativa, non considerata in questo esempio, è quella di eseguire un algoritmo di approssimazione prima di quello di branch-and-bound, in modo da aver già a disposizione una soluzione ammissibile fin dall'inizio. Una soluzione più sofisticata è di far partire l'algoritmo di branch-and-bound con politica depth-first in modo giungere velocemente ad una soluzione completa e di modificare poi la politica in best-first (riordinando di conseguenza la lista dei nodi aperti) dopo aver raggiunto la prima foglia. Nel nostro esempio useremo comunque una variabile, dichiarata, inizializzata e modificata come segue. Nella parte dichiarativa del programma principale bisogna aggiungere:

```
var ub: real;
```

Nella procedura di inizializzazione bisogna aggiungere:

```
ub := infinito;
```

essendo `infinito` il valore di una costante abbastanza grande. Infine, al termine della procedura `Elabora_foglia` bisogna aggiungere:

```
begin
...
ub := best^.val;
end;
```

II.2 Lower bound

Diversamente dall'upper bound che è valido globalmente, il valore del lower bound è specifico di ogni nodo. Pertanto esso deve essere memorizzato in una variabile nel record del nodo. Bisogna quindi aggiungere al codice la dichiarazione,

```
type rec = record
             ...
             lb: real;                        /*lower bound */
             ...
           end;
```

eventualmente l'inizializzazione (in `Inizializza`), che però è ininfluente dal momento che la radice è unica, e la modifica (in `Genera_figli`).

```
procedure Genera_figli (padre: punt; /* in ingresso */
                     var tree: punt  /* in uscita */
                       );
  var figlio: punt;
      i: integer;
begin
for i:=2 to N do
  if (not padre^.vis[i])
    then begin
         New (figlio);
         /* copia nodo padre in nodo figlio */
         figlio^ := padre^;
         /* modifica il figlio rispetto al padre */
         figlio^.liv := figlio^.liv + 1;
         figlio^.seq[figlio^.liv] := i;
         figlio^.val := figlio^.val + D[padre^.seq[padre^.liv],
                                        figlio^.seq[figlio^.liv]];
         figlio^.vis[i] := true;
         /* inserisci figlio nella lista dei nodi aperti */
         if (figlio^.liv = N)
           then begin /* nodo foglia */
                figlio^.val := figlio^.val +
                               D[figlio^.seq[figlio^.liv,1];
                Elabora_foglia (figlio);
                end
           else begin
                Calcola_lower_bound (figlio);
                Inserisci (figlio, tree);
                end;
end;
```

Il calcolo del lower bound dipende fortemente dal problema in esame. E' quindi opportuno che sia compiuto da un'apposita procedura o funzione, che ha in ingresso i dati del nodo da valutare e calcola in uscita il valore del lower bound di quel nodo. La sua dichiarazione potrebbe essere:

```
procedure Calcola_lower_bound (figlio: punt);
begin
...
figlio^.lb := ...;
end;
```

Il suo effetto è di assegnare il valore calcolato al campo `lb` del record del nodo. Algoritmi di lower bounding per il problema del TSP asimmetrico sono ad esempio quelli basati sul calcolo del doppio spanning tree e sul calcolo del matching di costo minimo su un grafo bipartito con *N* nodi per partizione. Non vengono presentati qui perchè sono specifici per l'esempio.

II.3 Chiusura dei nodi

La chiusura dei nodi "non abbastanza promettenti" avviene in seguito ad un confronto che va fatto dopo aver generato ogni nodo "figlio" e averne calcolato il lower bound e prima di inserirlo nella lista dei nodi aperti. Se il nodo generato è un nodo "foglia" ovviamente è inutile calcolarne il lower bound. La procedura `Genera_figli` risulta perciò modificata come segue:

```
procedure Genera_figli (padre: punt; /* in ingresso */
                        var tree: punt  /* in uscita */
                          );
  var figlio: punt;
      i: integer;
begin
for i:=2 to N do
  if (not padre^.vis[i])
    then begin
         New (figlio);
         /* copia nodo padre in nodo figlio */
         figlio^ := padre^;
         /* modifica il figlio rispetto al padre */
         figlio^.liv := figlio^.liv + 1;
         figlio^.seq[figlio^.liv] := i;
         figlio^.val := figlio^.val + D[padre^.seq[padre^.liv],
                                        figlio^.seq[figlio^.liv]];
         figlio^.vis[i] := true;
         /* inserisci figlio nella lista dei nodi aperti */
         if (figlio^.liv = N)
           then begin /* nodo foglia */
                figlio^.val := figlio^.val +
                               D[figlio^.seq[figlio^.liv,1];
                Elabora_foglia (figlio);
                end
           else begin
                Calcola_lower_bound (figlio);
                if (figlio^.lb < ub)
                  then Inserisci (figlio, tree)
                  else Dispose (figlio);
```

```
                end;
end;
```

Per concludere riporto il programma completo, tranne che per le procedure di I/O, di inserimento ordinato e di calcolo del lower bound.

```
Program Branch_and_bound (input, output);

Const N = ...;
      Infinito = ...;

type punt = ^rec;
     rec = record
             seq: array [1..N] of integer;  /*soluzione parziale*/
             liv: integer;                  /*livello*/
             val: real;                     /*valore parziale (costo)*/
             vis: array [1..N] of boolean;  /*città già visitate*/
             lb: real;                      /*lower bound */
             next: punt;
           end;

var tree, best: punt;
    ub: real;
    D: array [1..N,1..N] of real;

procedure Inizializza;
  var i: integer;
begin
/* inizializzazione radice */
New (tree);
tree^.liv := 0;
tree^.val := 0.0;
tree^.next := nil;
tree^.vis[1] := true;
for i:=2 to N do tree^.vis[i] := false;
best := nil;
ub := infinito;
end;

Procedure Ricerca;
  var padre, figlio: punt;
      i: integer;

  Procedure Elabora_foglia (var figlio: punt);
  begin
  if (best = nil)
    then begin /* prima soluzione completa trovata => ottima corrente*/
         best := figlio;
         figlio := nil;
         end
    else begin /* confronta con l'ottima corrente */
         if (figlio^.val < best^.val)
           then begin /* è migliore */
                Dispose (best);
                best := figlio;
                figlio = nil;
                end
           else begin /* non è migliore */
                Dispose (figlio);
                end;
  ub := best^.val;
  end;

  procedure Inserisci (var figlio, tree: punt);
```

```
  begin
  /* inserimento in lista ordinata */
  end;

  procedure Calcola_lower_bound (figlio: punt);
  begin
  /* calcolo del lower bound del nodo */
  figlio^.lb := ...;
  end;

begin
while (tree <> nil) do
  begin
  /* estrazione del nodo padre */
  padre := tree;
  tree := padre ^.next;
  padre ^.next := nil;
  /* generazione dei nodi figli */
  for i:=2 to N do
    if (not padre^.vis[i])
      then begin
           New (figlio);
           /* copia nodo padre in nodo figlio */
           figlio^ := padre^;
           /* modifica il figlio rispetto al padre */
           figlio^.liv := figlio^.liv + 1;
           figlio^.seq[figlio^.liv] := i;
           figlio^.val := figlio^.val + D[padre^.seq[padre^.liv],
                                          figlio^.seq[figlio^.liv]];
           figlio^.vis[i] := true;
           /* inserisci figlio nella lista dei nodi aperti */
           if (figlio^.liv = N)
             then begin /* nodo foglia */
                  figlio^.val := figlio^.val +
                                 D[figlio^.seq[figlio^.liv,1]];
                  Elabora_foglia (figlio);
                  end
             else begin
                  Calcola_lower_bound (figlio);
                  if (figlio^.lb < ub)
                    then Inserisci (figlio, tree)
                    else Dispose (figlio);
                  end;
  /* cancellazione del nodo padre */
  Dispose (padre);
  end;
end;

/* main */

Begin
/* procedure di input: lettura della matrice D */
Inizializza;
Ricerca;
/* procedure di output: scrittura della soluzione best^ */
End;
```