Algoritmi e Strutture Dati (II anno) - a.a. 1996-97

Ultima modifica: 20 Giugno 1997. Commenti a Eugenio Moggi.

Altre tecniche di programmazione

Finora abbiamo identificato un problema con una relazione P tra I e O, dove I e' l'insieme degli input ed O quello degli output (o soluzioni possibili). Dirremo che o e' una soluzione ammissibile per i se P(i,o). Dal punto di vista di cosa e' richiesto da un algoritmo introdurremo la seguente terminologia:

- problema della ricerca di una soluzione ammissibile. output: (un o t.c. P(i,o)): O
- problema dell'esistenza di una soluzione ammissibile. output: (esiste un o t.c. P(i,o)): boolean
- problema della enumerazione delle soluzioni ammissibili. output: {o | P(i,o)}: sottoinsiemi di O

In certi casi si ha anche una misura di bonta' sulle soluzioni possibili v:O -> V, dove V e' un insieme ordinato (p.e. l'insieme R dei reali). Dirremo che o e' una soluzione ottima se e' ammissibile e per ogni o' ammissibile v(o') <= v(o).

- problema della ricerca di una soluzione ottima. output: (un o t.c. o ottima): O
- problema del valore ottimo output: max {v(o) | P(i,o)): V
- problema della enumerazione delle soluzioni ottime. output: {o | o ottima}: sottoinsiemi di O

Enumerazioni e visite

Molto spesso possiamo strutturare l'insieme delle soluzioni possibili come le foglie di un albero. Se l'albero e' finito un modo semplice di enumerare le foglie e' di effettuare una visita DFS. Lo schema generale di enumerazione consiste nel fornire le seguenti procedure

```
var x: stato {della enumerazione, usato come variabile globale}
procedure init; {inizializza lo stato}
procedure next; {tenta di generare il prossimo elemento}
function fine: boolean; {indica se abbiamo terminato}
function elem: O: {elemento corrente nella enumerazione}
```

Facendo uso di esse e' immediato risolvere i problemi introdotti precedentemente (purche' si disponga di procedure P e v per verificare l'ammissibilita' e calcolare il valore di una soluzione possibile), p.e.

```
** ricerca di una soluzione ammissibile per l'input i
init;
while not fine do
  o <- elem
  if P(i,o)
  then break
  else next
if fine
then "non esistono soluzioni ammissibili"
else return(o)
```

```
** ricerca di una soluzione ottimale per l'input i
var val: V
    opt: O
init; val <- -infinito;
while not fine do
  o <- elem
  if P(i,o) and val < v(o)
  then opt <- o; val <- v(o)
  else next
if val = -inifinito
then "non esistono soluzioni ammissibili"
else return(opt)
```

Diamo alcuni esempi di alberi associati ad insiemi di soluzioni possibili:

- i sottoinsiemi di un insieme con n elementi sono 2^n e si possono identificare con le foglie dell'albero S_n t.c. S_0 e' una foglia e S_{n+1} e' un albero con due sottoalberi della forma S_n
- le permutazioni di una sequenza di lunghezza n sono n! e si possono identificare con le foglie dell'albero P_n t.c. P_0 e' una foglia e P_{n+1} e' un albero con n+1 sottoalberi della forma P_n.

Una possibile implementazione della enumerazione dei sottoinsiemi (visti come bitvectors) e'

```
var x: array [1..n] of boolean
    fine: boolean
procedure init;
  for i <-1 to n do x[i] <- false
  fine <- false
procedure next;
  i <- n
  while (i > 0) and (x[i]=true) do
    x[i] <- false
    i <- i-1
  if i=0
  then fine <- true
  else x[i] <- true
```

le procedure fine ed elem sono banali. Una possibile implementazione della enumerazione delle permutazione di una sequenza x e'

```
input: x array [1..n] of X
var p: array [1..n] of integer {i <= p[i] <= n}
    fine: boolean
procedure init;
  for i <-1 to n do p[i] <- i
  fine <- false
procedure next;
  i <- n
  while (i > 0) and (p[i]=n) do
    p[i] <- i
    i <- i-1
  if i=0
  then fine <- true
  else p[i] <- p[i]+1
procedure elem;
  y <- x;
  for i <-1 to n do exchange(y[i],y[p[i]])
  return(y)
```

La procedure fine e' banale. La permutazione di x corrispondente ad un certo p corrisponde ad effettuare una sequenza di n scambi, alcuni dei quali banali (quando p[i]=i). Una variante del precedente algoritmo consiste nel mantere y nello stato, evitando il ricalcolo a partire da x ad ogni chiamata di elem.

```
input: x array [1..n] of X
var p: array [1..n] of integer {i <= p[i] <= n}
    y: array [1..n] of X
    fine: boolean
procedure init;
  for i <-1 to n do p[i] <- i
  y <- x
  fine <- false
procedure next;
  i <- n
  while (i > 0) and (p[i]=n) do
    p[i] <- i
    exchange(y[i],y[n])
    i <- i-1
  if i=0
  then fine <- true
  else exchange(y[i],y[p[i]])
       p[i] <- p[i]+1
       exchange(y[i],y[p[i]])
```

In questa variante si mantiene il valore di y consistente con quello di p, quando quest'ultimo viene modificato in init e next.

Diamo lo schema generale di un algoritmo di enumerazione come visita DFS di un albero con radice r le cui foglie corrispondono a soluzioni possibili, cioe' elementi di O.

```
var S: stack of nodi {per la visita DFS}
procedure init;
  S <- (r)
  while (S non vuota) and (top(S) nodo interno) do
    pop(S,u)
    for u' figlio di u do push(S,u')
procedure next;
  pop(S,u)
  while (S non vuota) and (top(S) nodo interno) do
    pop(S,u)
    for u' figlio di u do push(S,u')
function fine;
  return(S e' vuota)
function elem;
  return(elem(top(S)))
```

Soluzioni ammissibili e pruning

Una enumerazione di tutte le soluzioni possibili tende ad essere molto costosa. Una tecnica generale per cercare di ridurre i costi consiste nel *potare* l'albero su cui effettuare la visita. Per esempio, se siamo interessati solo alle soluzioni ammissibili, durante la visita di un nodo interno dell'albero si potrebbe gia' avere abbastanza informazioni per concludere che e' inutile esplorare il sottoalbero corrispondente, perche' non contiene soluzioni ammissibili.

Supponiamo di essere interessati solo alle foglie dell'albero S_n dei sottoinsiemi che sono soluzioni al problema dello zaino (input: K, $c_1,...,c_n$; vincolo: $\text{Somma}_i\ c_i*x_i <= K$). Possiamo usare il seguente predicato per potare l'albero

- prune(m,x) sse (Somma_{i in 1..m} c_i * x_i) > K, dove (m,x) identifica un nodo interno dell’albero, ovvero 0 <= m < n e x e’ un array [1..m] of {0,1}

Quindi se durante la visita raggiungiamo un nodo (m,x) che soddisfa prune(m,x) torniamo indietro (backtracking).

Diamo lo schema generale di un algoritmo di ricerca di una soluzione ammissibile che usa il pruning. Essenzialmente si tratta di una variante della visita DFS. Tale variante usa una funzione booleana che per ogni nodo u (dell’albero usato per l’enumerazione delle soluzioni possibili) ci dice se e’ il caso di visitare il sottoalbero radicato in u:

- prune(u) = true implica che non esistono soluzioni ammissibili nel sottoalbero radicato in u,
- mentre prune(u) = false non ci garantisce niente.

```
var S: stack of nodi {per la visita DFS}
S <-
while S non vuota do
  pop(S,u)
  if u foglia
  then if P(i,elem(u)) {se l’elemento di O corrispondente ad u e’ ammissibile}
       then return(elem(u))
  else for u’ figlio di u do
         if not prune(u’)
         then push(S,u’)
"non esistono soluzioni ammissibili"
```

Soluzioni ottimali e metodo branch and bound

Se siamo interessati alle soluzioni ottimali possiamo effettuare un pruning dinamico dell’albero, che decide se potare o meno un sottoalbero anche sulla base della migliore tra le soluzioni ammissibili sinora esaminate. In questo caso, l’informazione da associare ad un nodo interno e’ piu’ complessa:

- v_inf(u) = valore di una soluzione ammissibile nel sottoalbero radicato in u
- v_sup(u) = limite superiore ai valori delle soluzioni ammissibili nel sottoalbero
- si conviene che v_inf(u) = v_sup(u) = -infinito se non esistono soluzioni ammissibili nel sottoalbero

```
var val: V {valore ottimo temporaneo}
    sol: O {soluzione ottima temporanea}
    Q: sottoinsiemi of nodi {sottoalberi che potrebbero avere sol. ottime}
val <- -infinito
Q <- {r}
while Q non vuoto do
  estrai un nodo u da Q
  if u foglia
  then if v_inf(u) >= val
       then sol <- elem(u)
            val <- v_inf(u)
  else if v_inf(u) >= val
       then val <- v_inf(u)
       for u’ figlio di u do
         if v_sup(u’) >= val
         then inserisci u’ in Q
if val = -infinito
then "non esistono soluzioni ammissibili"
else return(sol)
```

Supponiamo di essere interessati a trovare una soluzione ottimale al problema dello zaino con costi (input: K, c_1,...,c_n, p_1,...,p_n; vincolo: Somma_i c_i*x_i <= K; valore: Somma_i p_i*x_i). Possiamo usare le seguenti funzioni di stima:

- v_inf(m,x) = Somma_{i in 1..m} p_i * x_i se Somma_{i in 1..m} c_i * x_i) <= K, -infinito altrimenti.
- v_sup(m,x) = v_inf(m,x) + (Somma_{i in m+1..n} p_i)

Il metodo branch and bound presenta delle analogie con le visite BFS, nel senso che Q e’ come l’insieme dei nodi frontiera. Vi sono vari criteri per scegliere un elemento di Q, p.e. si potrebbe scegliere quello con v_inf(u) piu’ elevato.