Branch-and-Bound

A.A. 2017-2018

Nei problemi di ottimizzazione ad ogni soluzione è associato un costo e si richiede di costruire una soluzione di costo minimo o massimo. Per tali problemi spesso è possibile calcolare un limite sul costo di tutti i percorsi che iniziano con un dato cammino parziale.

La tecnica per il calcolo dei limiti per le soluzioni parziali al fine di contenere il numero di soluzioni complete da esaminare viene chiamata "***branch-and-bound***" (separa e limita) ed è un adattamento e una generalizzazione del backtracking ai problemi di ottimizzazione.

La tecnica "branch and bound" prevede di:

- esplorare l'albero delle possibili soluzioni, mantenendo sempre la migliore soluzione ammissibile tra quelle esaminate;
- evitare di generare i sottoalberi delle scelte di nodi che non possono migliorare la soluzione trovata fino a quel momento.

Potare l'albero

Quando si cerca un percorso "migliore" esiste un'altra importante tecnica di "potatura" in grado di porre fine alla ricerca non appena si scopre che questa non può avere successo.

Se si è trovato un percorso di costo c , non ha senso proseguire la ricerca lungo un percorso il cui costo supera c.
La riduzione è tanto più efficace quanto prima si trova un percorso a basso costo.

Spesso è anche possibile calcolare un limite sul costo di tutti i percorsi che iniziano con un dato cammino parziale.

Potare l’albero

La tecnica per il calcolo dei limiti per le soluzioni parziali al fine di contenere il numero di soluzioni complete da esaminare viene chiamata “***branch-and-bound***” (separa e limita) ed è un adattamento del backtracking ai problemi di ottimizzazione.

Risolvere un particolare problema richiede lo sviluppo di sofisticati criteri da usare per limitare la ricerca. Comunque, per quanto possano essere sofisticati questi criteri, generalmente *il tempo di esecuzione nel caso peggiore degli algoritmi di backtracking e branch-and-bound resta super-polinomiale*.

Poiché la riduzione dello spazio di ricerca dipende dai criteri di eliminazione e questi variano da problema a problema, non è possibile dare una valutazione complessiva del metodo che prescinda dal particolare problema in esame.

Branch & Bound

Vediamo esempi di applicazione della tecnica su problemi di *minimizzazione*: ogni soluzione "ammissibile" ha un "costo" associato, si tratta di trovarne una di costo *minimo*.

Ipotesi:

- Ogni sequenza di scelte ha un costo *non negativo*.
- Ogni scelta aggiunta alle scelte già fatte *non diminuisce* il costo della soluzione parziale.

Perchè Branch and Bound?

- Branch: criterio per determinare le scelte da fare e generare i figli di un nodo dell'albero degli stati.
- Bound: regola per calcolare il lower bound LB per ogni nodo dell'albero, cioè un minimo al di sotto del quale non scendono i costi delle soluzioni ammissibili presenti nel suo sottoalbero.

Per lo stesso problema si possono ottenere algoritmi diversi variando le regole di branch e la funzione lower bound.

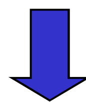
Il ***Lower Bound*** per un nodo X dell'albero in generale dipende dalla sequenza di scelte fatte S[1], …, S[i] e deve garantire che tutte le soluzioni ammissibili generabili aggiungendo nuove scelte a tale sequenza abbiano un costo maggiore o uguale al ***Lower Bound***: LB(X).

Se nel nodo X raggiunto con le scelte S[1], …, S[i], LB(X) risulta maggiore o uguale al costo della soluzione migliore trovata fino a quel momento, si evita di procedere, potando così l'albero delle scelte del sottoalbero che ha radice in quel nodo.

È importante individuare una buona funzione ***Lower Bound***

Entrambe le strategia di visita dello spazio delle soluzioni, breadth-first e depth-first, si generalizzano alle strategie branch & bound, dove sono chiamate First In First Out (FIFO) e Last In First Out (LIFO), rispettivamente.

Sia per il LIFO che per il FIFO branch and bound la regola di selezione del prossimo nodo da espandere è "**cieca**": non dà preferenza al nodo che ha una chance maggiore di arrivare "velocemente" a un "nodo risposta".



Usare un metodo di ricerca piu` intelligente che permetta di selezionare il nodo da espandere più promettente.

Il modo ideale per assegnare il rango a un nodo è sulla base dello sforzo computazionale aggiuntivo per raggiungere da quel nodo il nodo risposta. Ad esempio si potrebbe associare ad ogni nodo:

1. Il numero di nodi nel sottoalbero che devono essere generati prima che sia generato un nodo risposta
2. Il numero di livelli per cui un nodo risposta dista dal nodo

Sono funzioni costo ideali: nel tempo in cui il costo di un nodo è determinato, il suo sottoalbero è stato esplorato; per questo motivo gli algoritmi “branch & bound” di solito usano ranghi basati solo su una “stima” del costo.

Il costo di un nodo X può essere visto come somma del costo stimato e di una funzione del costo speso per raggiunge X dalla radice:

$$c^s(X) = f(h(X)) + g^s(X)$$

Funzione non decrescente (f)

Costo per raggiungere X dalla radice ($h(X)$)

Stima del costo per raggiungere un nodo risposta ($g^s(X)$)

Se il prossimo nodo da espandere viene scelto come il nodo “vivo” con c^s minore, la strategia di ricerca si chiama ***Least Cost Search*** (***LC-Search***)

- FIFO è una LC-S in cui ogni vertice X ha
  $g^s(X) = 0$ e $f(h(X)) = livello(X)$
- LIFO è una LC-S in cui ogni vertice X ha
  $f(h(X)) = 0$ e se Y è figlio di X, $g^s(X) >= g^s(Y)$

A volte è anche possibile definire una funzione u(X) che fornisce un *upper-bound* al costo di un nodo X. Questo può aiutare a guidare la ricerca.

Branch & Bound

Ogni nodo risposta (ogni soluzione ammissibile) X ha un costo $c(X)$ e, poiché consideriamo problemi di minimizzazione, si deve trovare un nodo risposta di costo minimo. Sia ora $c^s(X)$ una stima per difetto di $c(X)$ e $u(X)$ una stima per eccesso ($c^s(X) \leq c(X) \leq u(X)$)

- $c^s(X) \leq c(X)$ è usata per fornire un lower bound sulle soluzioni ottenibili da un nodo X.

- Se *u è un upper bound* di una soluzione di costo minimo, tutti i nodi vivi con $c^s(X) > u(X)$ possono essere uccisi perché hanno costo $c(X) \geq c^s(X) > u(X)$

- Un valore iniziale per u può essere ∞ o trovato con qualche euristica; u deve essere aggiornato quando si trova una soluzione di costo inferiore.

Branch & Bound: commesso viaggiatore

Un algoritmo “branch and bound” per il problema del commesso viaggiatore.

Commesso Viaggiatore - TSP: *Dato un grafo $G = (V,E)$ non orientato, completo ($\forall u,v \in V$, $u \neq v$, $(u, v) \in E$) e pesato con la funzione $d\colon E \to N^+$, trovare un ciclo semplice, che contenga tutti i vertici di G, di costo minimo (Il costo di un ciclo è la somma dei pesi degli archi che lo compongono).*

Branch & Bound: commesso viaggiatore

Possibili applicazioni:

- un furgone postale deve raccogliere la posta dalle cassette postali in n località diverse.

  Si può rappresentare questo problema con un grafo di n vertici. Ogni vertice rappresenta un ufficio postale; il furgone dovrà partire dall'ufficio centrale e farvi ritorno. Agli archi *<i, j>* è associato come costo la distanza tra *i* e *j*. Il percorso circolare minimo sarà il tragitto migliore per il furgone

- un braccio robotico avvita i bulloni di un macchinario su una linea di produzione. Il braccio ha una posizione iniziale, si muove su ogni bullone e ritorna nella posizione iniziale.

  Il cammino del braccio è chiaramente un tour su un grafo in cui i vertici rappresentano i bulloni. Un tour di costo minimo minimizzerà il tempo neccessario al braccio per completare il task.

- un ambiente di produzione in cui diversi prodotti vengono assemblati dallo stesso insieme di macchinari. La produzione procede per cicli. Ad ogni ciclo si producono n differenti prodotti. Il passaggio della macchina dall'assemblaggio del prodotto i a quella del prodotto j, ha un costo c(i,j). Trovare una sequenza per l'assemblaggio dei prodotti che minimizzi i costi di spostamento equivale a risolvere il problema *TSP* su un grafo con n vertici i cui pesi sono i costi di spostamento.

Branch & Bound: commesso viaggiatore

- la scelta avviene tra le città (nodi del grafo)
- definiamo un lower bound da associare ad ogni nodo

Sia $|V| = n$; assumiamo $n > 1$ e i vertici numerati 1, 2, …, n.
Supponiamo inoltre che 1 sia il vertice iniziale.

Usiamo il vettore S come vettore che ricorda i vertici sul cammino che si sta esplorando, *minsol* e *mincost* sono rispettivamente un cammino minimo e il suo costo.

Possiamo definire il costo dei nodi nel modo seguente:

$$c(X) = \begin{cases} \text{costo del cammino ciclico individuato, se X è una foglia} \\ \text{costo della foglia di costo minimo nel sottoalbero di X, se X non è una foglia} \end{cases}$$

Cerchiamo due funzioni c^s e u tali che per ogni nodo X:

$$c^s(X) \leq c(X) \leq u(X).$$

Il nodo soluzione con costo minimo corrisponde al percorso più breve.

Branch & Bound: commesso viaggiatore

Osservazioni:

S = S[1], S[2], …, S[i] sia un cammino parziale.

1. Bisogna raggiungere un altro vertice da S[i], cioè ci deve essere un arco da S[i] a un vertice h non in S:

$$A = \min_{h \notin S} \{d[S[i], h]\}$$

2. Bisogna chiudere il ciclo, cioè ci deve essere un arco da un vertice h non ancora in S a S[1]:

$$B = \min_{h \notin S} \{d[S[1], h]\}$$

3. Ogni vertice h non in S dovrà essere attraversato, cioè essere estremo di due archi: (h, p) e (h, q)

$$D[h] = \min_{\substack{h \neq p \neq q \\ h \notin S}} \{d[h,p]+d[h,q]\}$$

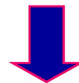
N.B. p e q possono anche appartenere a S

Branch & Bound: commesso viaggiatore

Se il numero di vertici in V – S è k,

$$A + B + \Sigma_{h \notin S} D[h]$$

è il peso di 2 + 2k = 2(k+1) archi; il numero di archi da aggiungere per ottenere un ciclo è k+1 e il loro peso non può essere inferiore a $(A + B + \sum_{h \notin S} D[h])/2$, in quanto abbiamo preso gli archi di peso minimo



Ogni cammino ciclico che inizia con S[1], S[2], …, S[i] non può avere peso minore di

$$\sum_{j=1}^{i-1} d(S[j],S[j+1]) + \lfloor (A + B + S_{h \notin S} D[h])/2 \rfloor$$

Abbiamo trovato una buona definizione per $c^s(X)$!:

sia S[1], S[2], …, S[i] la soluzione parziale individuata dal cammino dalla radice al vertice X, definiamo $C[X] = \sum_{j=1}^{i-1} d(S[j],S[j+1])$

$c^s(X) = C[X] + \lfloor (A + B + \Sigma_{h \notin S} D[h])/2 \rfloor$ se X non e` una foglia

$c^s(X) = C[X] + d[S[i], S[1]]$ se X e` una foglia

Branch & Bound: commesso viaggiatore

Si mantiene in una variabile globale *mincost* (*minsol*) il costo (cammino) della soluzione minima finora trovata.

```
TSP_B&B (i,…)
   if (i ≤ n)
      for ogni j ∉ S = S[1]…S[i-1]
         S[i] ← j
         C[i] ← C[i-1] + d[S[i-1],S[i]]
         if (i < n)
            {calcola A, B e D[h] per ogni h ∉ S}
            LB ← C[i] + ⌊(A+B+∑_{h ∉ S} D[h])/2 ⌋
         else LB ← C[i] + d[S[i],S[1]]
         if (LB < mincost)      \\altrimenti si abbandona questa strada
            if (i = n) mincost ← LB
                       minsol ← S
            else TSP_B&B (i +1)
      return
   return
```

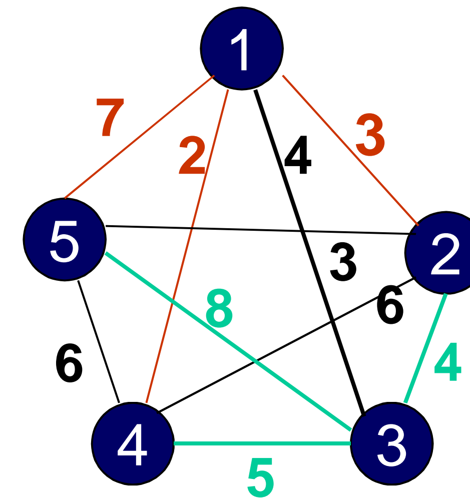
Branch & Bound: commesso viaggiatore

Siano i = 2, S[1]=1 e S[2]= 3

C[2] = C[1] + d(1,3) = 4

A = min{d[3,2], d[3,4], d[3,5]} =
= min{4,5,8} = 4

B = min{d[1,2], d[1,4],d[1,5]} =
= min{3,2,7} = 2



D[2] = 3+3 = 6

D[4] = 2+5 = 7

D[5] = 3+6 = 9

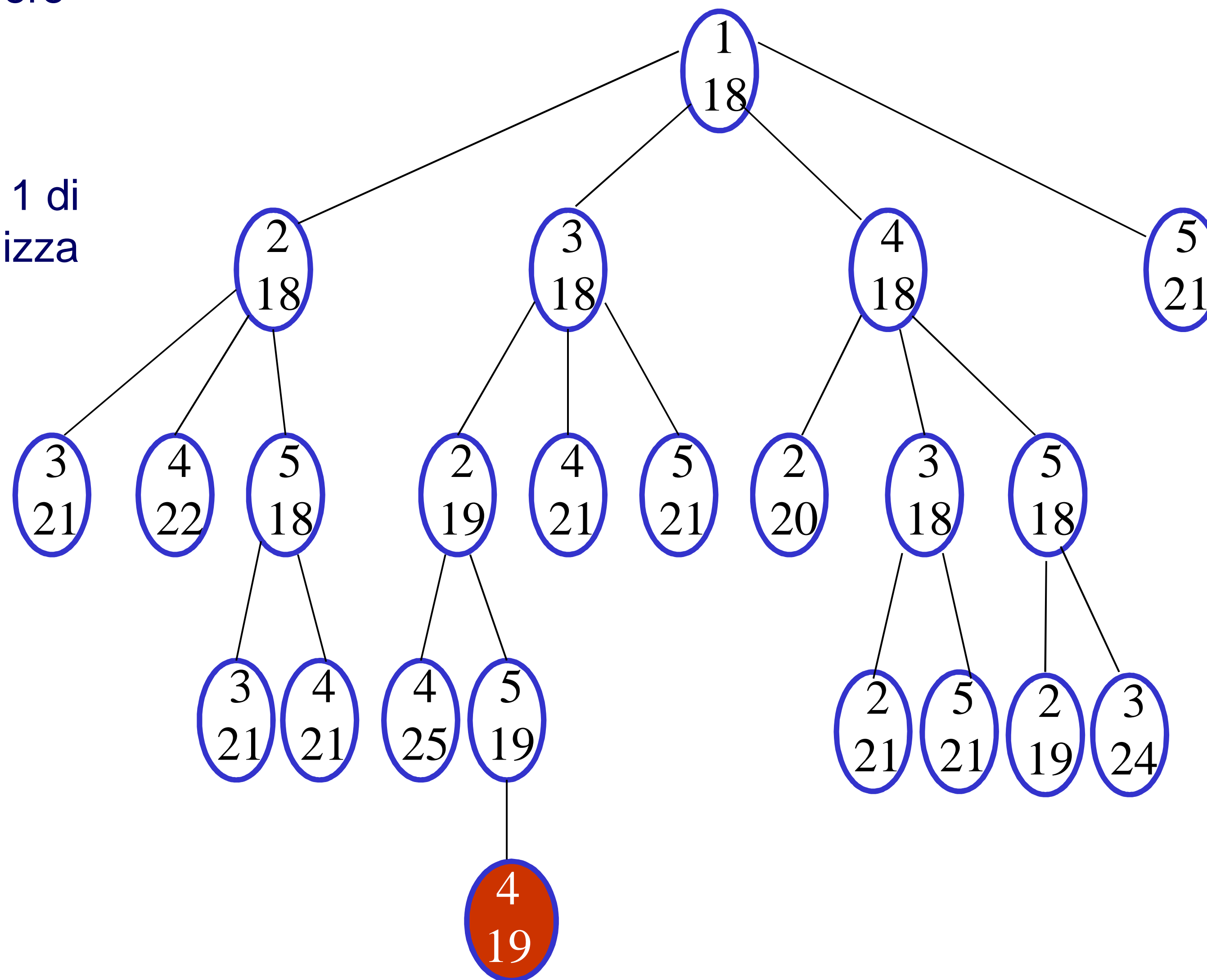
$$LB(S) = C[i] + \lfloor (A+B+\Sigma_{h \notin S} D[h])/2 \rfloor$$

$$4 + \lfloor (4+2+6+7+9)/2 \rceil = 18$$

Branch & Bound: commesso viaggiatore

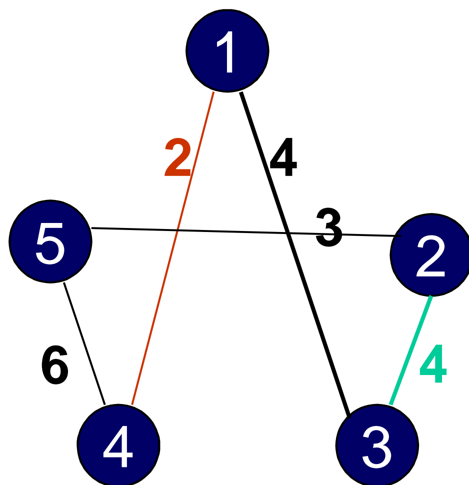
Supponiamo di avere una soluzione ottenuta con una permutazione casuale: 1 4 3 2 5 1 di costo 21. Si inizializza *mincost* a 21

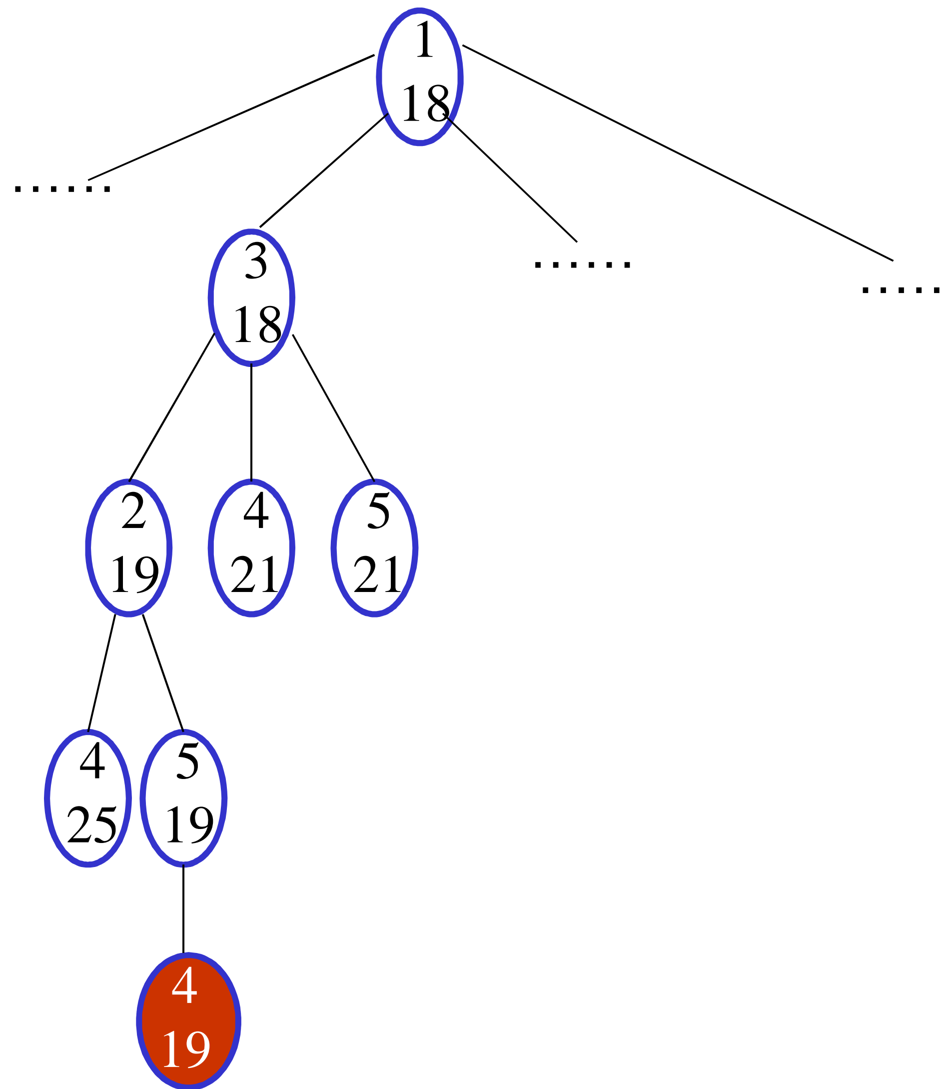
Si espandono prima i nodi di costo minimo.



Branch & Bound: commesso viaggiatore

Una soluzione di minimo costo:





Un algoritmo "branch and bound" per il problema "zaino 0-1".

Ricordiamo il problema:

Zaino 0-1 : *Dato uno zaino di "capacità" C ed n articoli* $O_1, \ldots, O_n$, *ciascuno con un "peso"* p_i *ed un "valore"* v_i, *riempire lo zaino in modo da massimizzare il valore degli oggetti inseriti*.

Vogliamo ottenere un algoritmo branch & bound, e confrontarlo con un algoritmo di backtracking, nell'ipotesi che i vettori peso (P) e valore (V) siano ordinati in modo non crescente rispetto a v_i/p_i e che la somma dei pesi degli oggetti sia maggiore della capacità dello zaino, $\sum_{i=1}^{n} p_i > C$.

Branch & Bound: zaino

Per usare la tecnica branch & bound dobbiamo stabilire il problema come problema di minimo:

funzione obiettivo da minimizzare: $-\sum_{i=1}^{n} V[i] \cdot S[i]$

soggetta al vincolo: $\sum_{i=1}^{n} P[i] \cdot S[i] \leq C$

Possiamo definire per ogni foglia X il costo nel modo seguente:

$c(X) = -\sum_{i=1}^{n} V[i] \cdot S[i]$ foglia per cui la somma dei pesi è $\leq C$

$c(X) = \infty$ per le altre foglie.

Per i nodi che non sono foglie definiamo:

$c(X) = \min \{c(\textbf{LCHILD}(X)), c(\textbf{RCHILD}(X))\}$.

Per ogni nodo X, - **BOUND** è $\leq c(X)$ per cui può essere usata come funzione $c^s(X)$ di lower bound.

Branch & Bound: zaino

valori attuali

Parte dal valore *attuale* e lo massimizza con la tecnica greedy

Riprendiamo:

```
BOUND (vv, pp, k, C)
    val ← vv
    peso ← pp
    for i ← k+1 to n
        if (peso + P[i] < C)
                peso ← peso + P[i]
                val ← val + V[i]
        else return (val + (|C – peso|)/P[i] ·V[i])
    return (val)
```

Per ogni nodo X, - **BOUND** è $\leq c(X)$ per cui può essere usata come funzione $c^s(X)$ di lower bound.

Branch & Bound: zaino

Per un nodo a livello j, $-\sum_{i=1}^{j-1} V[i] \cdot S[i]$ è sicuramente >= $-\sum_{i=1}^{n} V[i] \cdot S[i]$, pertanto possiamo scegliere tale somma come funzione upper bound per il nodo.

Una scelta migliore di u(X) è fornita dalla seguente funzione:

UBOUND $\left(-\sum_{i=1}^{j-1} V[i] \cdot S[i] \,,\, \sum_{i=1}^{j-1} P[i] \cdot S[i] \,,\, j-1, C\right)$

```
UBOUND (vv, pp, k, C)
    val ← vv
    peso ← pp
    for i ← k+1 to n do
        if (peso + P[i] ≤ C) peso ← peso + P[i]
                             val ← val - V[i]
    return (val)
```

Riempie "alla cieca" lo zaino fin quando possibile (nota: non si può fare peggio)

Per i nodi terminali (foglie dell'albero di ricerca) ovviamente si ha $u(X) = c^s(X)$

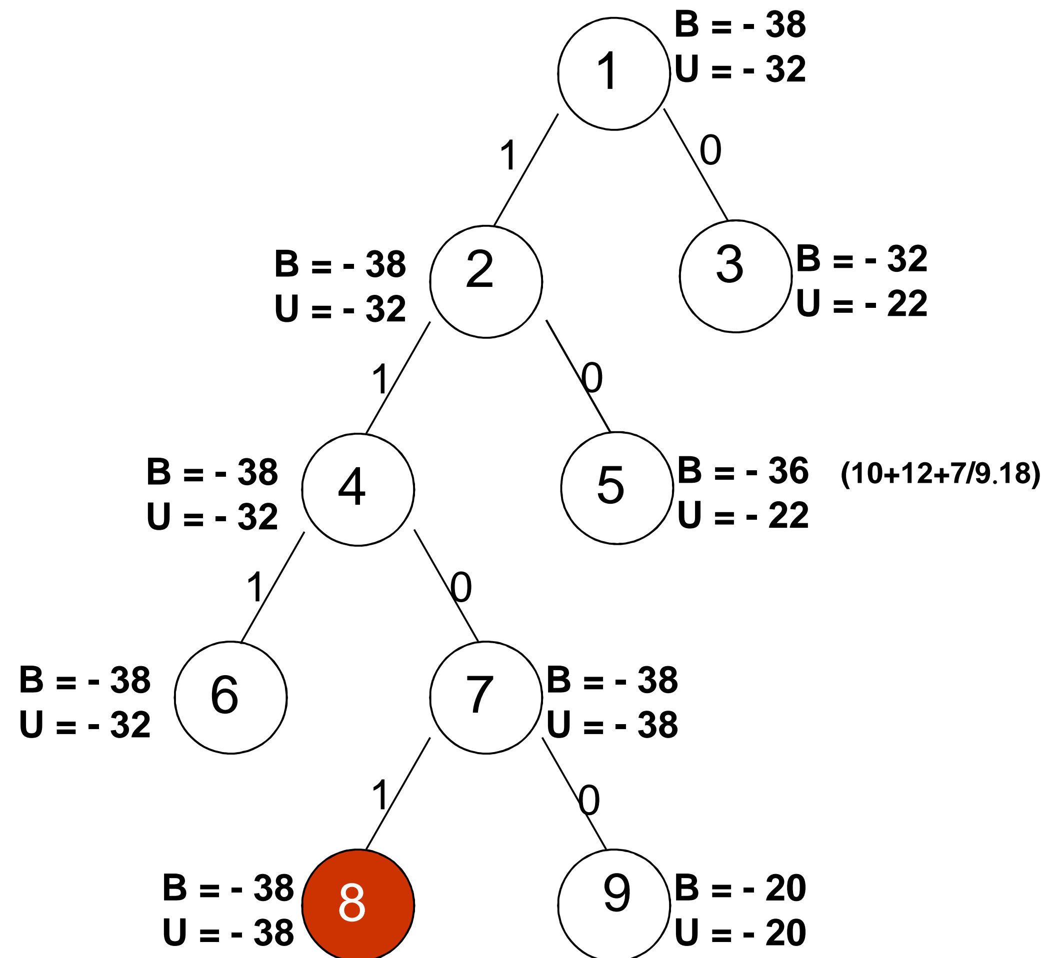
Branch & Bound: zaino

V = < 10, 10, 12, 18 >
P = < 2, 4, 6, 9 > C = 15

La numerazione dei vertici indica l'ordine in cui sono generati da una visita Least Cost, supponendo che il vertice 7 sia estratto dalla coda con priorità prima del vertice 6.

I nodi ancora vivi nel momento in cui la soluzione nel nodo 8 viene trovata sono: 6, 5, 3, 9. Tutti vengono uccisi perché un valore di $c^s(X) > U(8) = -38$
Quindi da essi non può essere trovata una soluzione migliore.

B rappresenta $c^s(X)$



Branch & Bound e zaino 0/1

Un algoritmo "backtracking" e un algoritmo "branch and bound" per il problema "zaino 0-1".

Ricordiamo il problema:

Zaino 0-1 : *Dato uno zaino di "capacità" C ed n articoli* $O_1, \ldots, O_n$, *ciascuno con un "peso"* p_i *ed un "valore"* v_i, *riempire lo zaino in modo da massimizzare il valore degli oggetti inseriti.*

Vogliamo ottenere un algoritmo branch & bound, e confrontarlo con un algoritmo di backtracking, nell'ipotesi che i vettori peso (P) e valore (V) siano ordinati in modo non crescente rispetto a v_i/p_i e che la somma dei pesi degli oggetti sia maggiore della capacità dello zaino, $\sum_{i=1}^{n} p_i > C$.

Backtracking: zaino

Usiamo un vettore S come vettore caratteristico della soluzione, Sol e Vmax come la soluzione migliore e il suo valore in ogni momento durante la visita. Vmax e` una variabile globale alla quale possiamo dare come valore iniziale - ∞.

Come funzione bounding

- per inserire un oggetto nello zaino (potare i sottoalberi sinistri) si puo` richiedere che il peso degli oggetti gia` nello zaino, sommato al peso dell'oggetto considerato, non superi la capacita` dello zaino;
- per scartare un oggetto (potare i sottoalberi destri) si puo` cercare un limite superiore al valore raggiungibile non considerando l'oggetto stesso.

Backtracking: zaino

```
Zaino (P, V, C, n, k, S, valore, peso, Sol)
    if (k > n and valore > Vmax)
        for i ← 1 to n
            Sol[i] ← S[i]
        Vmax ← valore
    else
        if (peso + P[k]) <= C)
            S[k] ← 1
            Zaino (P, V, C, n, k+1, S, valore+V[k], peso+P[k], Sol)
        y ← limite superiore al valore senza il k-esimo oggetto
        if (y > Vmax)
            S[k] ← 0
            Zaino(P, V, C, n, k+1, S, valore, peso, Sol)
    return
```

Chiamata esterna: *Zaino* (P, V, C, n, 1, S, 0, 0, Sol)

Backtracking: zaino

Per assegnare un valore a y si puo` preparare un vettore Resto tale che $Resto[k] = \sum_{i=k}^{n} V[i]$

$\underline{\text{if}}$ (k < n) y ← valore + Resto[k+1]
$\underline{\text{else}}$ y ← valore

Per definire la funzione bounding possiamo sfruttare l'ipotesi che i vettori peso e valore siano ordinati in modo non crescente rispetto a v_i/p_i ?

Se nel nodo X di livello k+1 i valori della soluzione S[i] con $1 \leq i \leq k$, sono già stati determinati, un upper bound per X può essere ottenuto rilasciando la condizione 0/1, cioe` considerando gli oggetti come continui (di cui puo` essere prelevata anche solo una parte.)
Si rilascia pertanto la richiesta S[i] = 0 o S[i] = 1 $(k+1 \leq i \leq n)$

Backtracking: zaino

L’algoritmo **BOUND** determina un upper bound sulla migliore soluzione ottenibile espandendo il nodo X nell’albero degli stati, trattando gli oggetti come “continui”.

```
BOUND (vv, pp, k, C)
    val ← vv
    peso ← pp
    for i ← k+1 to n
        if (peso + P[i] < C)
                        peso ← peso + P[i]
                        val ← val + V[i]
        else return (val + (|C – peso|)/P[i] ·V[i])
    return (val)
```

Backtracking: zaino

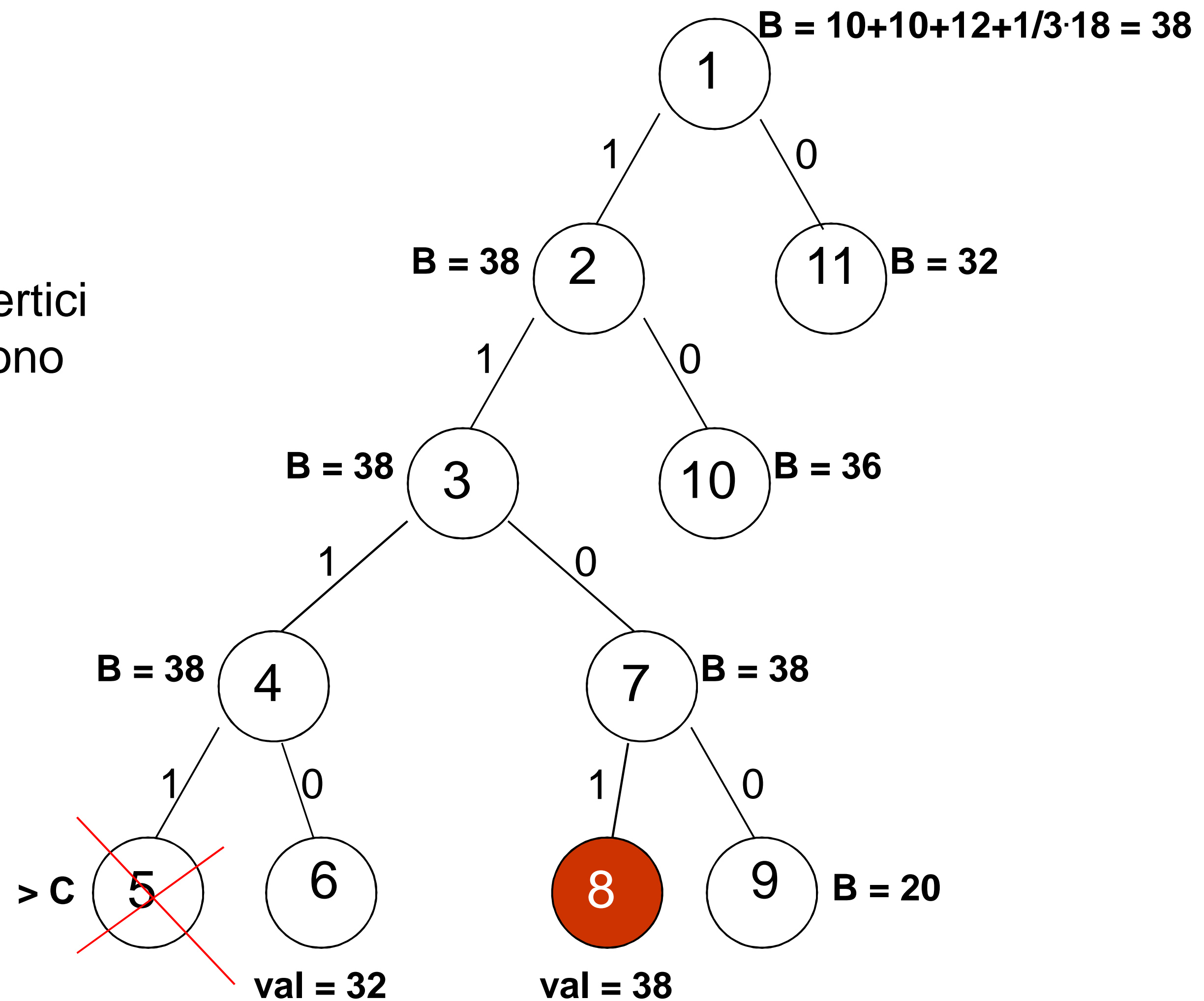
L'algoritmo **BOUND** determina un upper bound sulla migliore soluzione ottenibile espandendo il nodo X nell'albero degli stati, trattando gli oggetti come "continui".

```
BOUND (vv, pp, k, C)
    val ← vv
    peso ← pp
    i ← k+1
        while  (i <= n  and  peso + P[i] < C)
                        peso ← peso + P[i]
                        val ← val + V[i]
        return (val + (|C – peso|)/P[i] ·V[i])
    return (val)
```

Backtracking: zaino

V = < 10, 10, 12, 18 >
P = < 2, 4, 6, 9 >
C = 15

La numerazione dei vertici indica l'ordine in cui sono generati.



Backtracking: zaino

V = 11, 21, 31, 33, 43, 53, 55, 65
P = 1, 11, 21, 23, 33, 43, 45, 55
M = 110 n=8