Trade-off spazio tempo nelle strategie di ricerca

Università degli studi di Padova
Corso di Laurea in Ingegneria dell'Informazione

Laureando: Nicola Baraldo

Relatore: Prof. Andrea Pietracaprina

Anno accademico: 2010-2011
30 settembre 2011

Sommario

Questa tesina tratta il problema della selezione dell'n-esimo elemento più piccolo da un insieme infinito di valori distinti salvati in un albero binario che soddisfa la proprietà di heap. Gli argomenti sono presi da *On a Search Problem Related to Branch-and-Bound Procedures*, Karp, Saks, Wigderson, cercando di strutturare al meglio gli algoritmi presentati sia dal punto di vista della descrizione che dell'analisi delle prestazioni.

Indice

Capitolo 1

Introduzione

In questa tesina verrà trattato un problema di ricerca da un insieme di numeri interi salvati in una particolare struttura dati. I valori sono disposti in un albero binario T ipoteticamente infinito che soddisfa la proprietà di heap, cioè:

- ogni vertice $v \in T$ ha due figli: uno sinistro $L(v)$ e uno destro $R(v)$. La radice di T in particolare viene indicata con r
- ad ogni nodo v è associato un valore per mezzo della funzione $val : T \to N$ e per la proprietà di heap per ogni nodo interno $v \in T$ si ha $val(v) \leq val(L(v))$ e $val(v) \leq val(R(v))$. La radice conterrà l'elemento più piccolo, e ogni percorso dalla radice verso i propri figli attraverserà i nodi con valori in ordine crescente.

Per semplicità non è restrittivo assumere che i valori siano tutti distinti tra loro, cioè $val(v) \neq val(u) \forall u, v \in T$.
L'algoritmo è eseguito da un'agente di calcolo M che esplora l'albero. Ad ogni passo M si trova in un generico nodo $v \in T$, le operazioni che può eseguire sull'albero sono principalmente due: lettura di $val(v)$ (e di nessun altro valore), spostarsi in uno dei seguenti nodi: $L(v), R(v), F(v)$ (dove $F(v)$ denota il padre di v se esiste).

Il problema di ricerca che verrà trattato è il problema della selezione cioè: trovare l'n-esimo elemento più piccolo dall'insieme di valori $\{val(v)|v \in T\}$ salvati nella struttura dati appena descritta. L'input dell'algoritmo è costituito da due parametri: la radice r da cui inizierà l'esecuzione e un numero intero n.

Per quanto riguarda la complessità temporale ipotizziamo che il costo di ogni attraversamento di un nodo costa una unità, mentre tutte le altre operazioni sono gratis in quanto influenzano la complessità solo per un fattore

costante. Lo spazio utilizzato dall'algoritmo invece è definito come il numero totale di registri utilizzati da M. Per semplicità supponiamo che ogni registro possa contenere un singolo valore. Sia $T(n)$ la complessità temporale e $S(n)$ lo spazio utilizzato dall'algoritmo $SELECT(n)$ nel caso peggiore rispetto alla funzione $val : T \to N$.

1.1 Limiti inferiore e superiore della complessità temporale del problema

Prima di iniziare a parlare degli algoritmi troviamo un limite superiore e uno inferiore alla complessità temporale $T(n)$.

Limite inferiore Per decidere che un valore è l'n-esimo elemento più piccolo bisogna almeno aver visitato i nodi con gli $n-1$ valori più piccoli. Si ha quindi che $T(n)$ è

$$\Omega(n)$$

Limite superiore Per la proprietà di heap possiamo certamente dire che l'elemento da cercare sarà in un nodo $v \in T''$ con profondità minore di n dato che i valori sono tutti distinti tra loro, questi nodi sono al massimo $2^n - 1$ nel caso in cui ogni nodo abbia due figli, possiamo quindi concentrarci sul sotto albero T' che contiene solo tali nodi. L'estrazione dell'elemento minore da T' costa $O(\log(2^n)) = O(n)$, si può quindi trovare l'n-esimo elemento più piccolo con n estrazioni successive. Di conseguenza $T(n)$ è

$$O(n^2)$$

In questa tesi verranno presentati i vari argomenti nel seguente modo:

Capitolo 2 Verranno presentati gli algoritmi proposti per risolvere SELECT(n) con le relative analisi della complessità nel tempo e nello spazio.

Capitolo 3 Con i risultati del Capitolo 2 si cercherà di capire la relazione che lega tra loro la complessità temporale e spaziale al problema, fino al caso con spazio minimo.

Capitolo 4 Si morstrerà come sia proprio la tecnica adottata dagli algoritmi presentati a fornire i risultati ottenuti.

Capitolo 2

Algoritmi

In questo capitolo verranno trattati tre algoritmi per risolvere il problema $SELECT(n)$ che si differenziano per le complessità $T(n)$ e $S(n)$.

- Un algoritmo deterministico A con:

$$T_A(n) = n \cdot 2^{O(\sqrt{\log n})}, \qquad S_A(n) = O(n)$$

- Un algoritmo randomizzato Las Vegas (cioè che porta sempre ad un risultato corretto) B, con:

$$T_B(n) = n \cdot 2^{O(\sqrt{\log n})}, \qquad S_B(n) = O(\sqrt{\log n})$$

- Un algoritmo deterministico C con:

$$T_C(n) = n \cdot 2^{O(\sqrt{\log n})}, \qquad S_C(n) = O(\log^{2.5} n)$$

E nel Capitolo 3 verrà mostrato che per ogni ε fissato, esiste un algoritmo randomizzato LasVegas D_ε con:

$$T_D(n) = O(n^{1+\varepsilon}), \qquad S_D(n) = O\left(\frac{1}{\varepsilon}\right)$$

Il primo algoritmo non è mai utilizzato per dato che è inefficiente per quanto riguarda lo spazio utilizzato, però è la base per gli algoritmi B e C in quanto ha la stessa complessità temporale. Gli algoritmi B e C sono una variante efficiente rispetto $S(n)$ di quello A infatti, si vedrà più avanti che l'algoritmo A consuma molta memoria in una parte particolare dell'elaborazione. B risolve quel problema rendendola randomizzata, mentre C usa una variante deterministica che consuma poca memoria.

2.1 Algoritmo A

L'algoritmo applica la procedura branch-and-bound per trovare l'n-esimo elemento più piccolo, che da ora in poi verrà chiamato NTH. Un'osservazione importante è che questo elemento è unico perché abbiamo supposto all'inizio che tutti i valori siano distinti tra loro.
L'algoritmo procede per fasi, la fase i-esima eredita dalla precedente due valori interi: il limite inferiore L_{i-1} e il limite superiore U_{i-1}. Questi limiti definiscono l'intervallo all'interno del quale è sicuramente presente NTH. Al termine dell'elaborazione la fase i-esima produrrà due nuovi limiti L_i e U_i che definiranno un intervallo più piccolo rispetto quello della fase precedente, cioè:

$$L_{i-1} < L_i \leqslant NTH < U_i \leqslant U_{i-1}{}^{1}$$

Le fasi sono numerate a partire da 1, per la prima fase si assume:

$$L_0 = val(r), \qquad U_0 = +\infty$$

Il limite inferiore è $L_0 = val(r)$ perché per la proprietà di heap l'elemento più piccolo è nella radice. Mentre il limite superiore è $U_0 = +\infty$ perché non è possibile dire subito quale sia l'elemento maggiore in quanto la proprietà di heap non è di utilità.

Prima di tutto diamo due definizioni che saranno utili più avanti nella spiegazione:

Definizione 2.1. *Un valore è* ***buono*** *se è minore o uguale a NTH.*

Definizione 2.2. *T_i è il sottoalbero di T composto da tutti i nodi $v \in T$ che soddisfano $val(v) \leq L_{i-1}$.*

La fase i-esima è principalmente strutturata nel seguente modo. Parte dal sottoalbero T_i, attraverso una serie di operazioni calcola il sottoalbero T_{i+1} aggiungendo a T_i un certo numero di nodi buoni $g(n)$. Per trovare questi nodi, l'algoritmo richiama se stesso ricorsivamente più volte su alcuni nodi di T. Alla fine calcola il nuovo limite L_i per la fase successiva, che è funzione di T_{i+1}. Pure il limite superiore U_i viene aggiornato in un certo modo, però non è indispensabile per l'elaborazione, serve solamente a risparmiare di fare alcune chiamate ricorsive quando si calcola T_{i+1}.

Passiamo ora a descrivere meglio la struttura della fase i-esima. I nodi su cui l'algoritmo lavora (quelli su cui viene richiamato ricorsivamente) per aggiungere i $g(n)$ nodi buoni a T_i sono detti radici e sono salvati in una lista $R = \{r_1, r_2, \ldots, r_s\}$, $s \leq n$, le radici sono nodi di T che soddisfano certe

[1] Più avanti verrà spiegato il perchè delle disuguaglianze strette o meno

caratteristiche. Per identificare questi nodi consideriamo il sottoalbero T_i, l'insieme R sarà formato da tutti quei nodi $r_j \in T$ (non T_i) che soddisfano i seguenti requisiti:

- r_j è figlio di uno nodo foglia di T_i
- $val(r_j) \leq U_{i-1}$

Da notare che per la proprietà di heap i valori buoni non ancora trovati si troveranno solamente negli alberi radicati nei nodi che appartengono a R.

Prima di descrivere le operazioni da fare sui nodi $r_i \in R$ facciamo subito vedere come si capisce se un valore è buono o meno. Per verificare se un dato valore Z è buono è sufficiente contare i valori minori di Z nell'albero, nel caso in cui durante la conta si superi n allora il valore non è buono altrimenti si. Ecco un esempio in pseudo codice per contare il numero di nodi minori di Z fino ad un massimo di n:

Algorithm 2.1.1: GOOD(r, n, K)

if $val(r) \leq K$

then
- $y \leftarrow 1 + good(L(r), n-1, K)$
- **if** $y \leq n$
  - **then return** $(y) + good(R(r), n-y, K)$
  - **else return** (y)

else return (0)

Algorithm 2.1.2: isGOOD(v, n, K)

return $(\text{GOOD}(v, n, K) \leq n)$

L'algoritmo parte dalla radice r di T e legge $val(r)$, se $val(r) > Z$ allora per la proprietà di heap non serve controllare i figli di r e non abbiamo trovato nessun valore maggiore di Z.
Se $val(r) < Z$ abbiamo trovato un valore minore di Z e per trovare gli altri è sufficiente richiamare l'algoritmo ricorsivamente su uno dei figli per cercare altri $n-1$ valori minori di Z.
Se il totale di nodi trovati è maggiore a n allora l'elemento non è buono e l'algoritmo ritornerà un valore maggiore a n altrimenti se non ho ancora raggiunto n continua la ricerca degli altri $n-y$ elementi rimanenti nell'altro figlio. Dato che appena si contano n valori minori di Z l'algoritmo si arresta, la complessità temporale è $O(n)$.

Supponiamo di sapere che per ogni sotto albero radicato in r_j ci siano n_j valori siano buoni. Questi nodi andranno aggiunti a T_i, si avrà quindi:

$$\sum_{j=1}^{s} n_j = g(n) \leq n$$

Per capire meglio quali sono questi n_j nodi da trovare basta ripensare alla definizione di valore buono per capire che questi nodi non sono altro che gli n_j nodi più piccoli nel sottoalbero radicato in r_j. Se si conoscesse n_j si potrebbe richiamare l'algoritmo ricorsivamente su r_j per trovare l'n_j-esimo elemento più piccolo partendo da r_j, a questo punto gli altri valori saranno tutti quelli minori del valore trovato dall'algoritmo.
Tuttavia questo non è possibile perché n_j non è noto a priori, quindi è necessario procedere per tentativi incrementando ogni volta il valore candidato n_j testando ogni volta se il valore ritornato è buono o meno, nel caso non lo fosse allora si considera il valore precedente che si è usato. Usando una variabile intera k come candidato n_j da testare il procedimento può essere riassunto nei seguenti passi:

- si chiama l'algoritmo ricorsivamente per cercare il k-esimo elemento più piccolo (la prima volta $k = 1$)
- si testa se il valore ritornato dal passo precedente è buono o meno, nel caso lo fosse si incrementa k e si riparte dal passo precedente, nel caso in cui non lo sia ci si ferma.

Il modo con cui l'algoritmo incrementa k è quello di raddoppiarlo ad ogni iterazione, questo modo di procedere consente di ammortizzare il costo delle chiamate ricorsive precedenti. Si nota che non è detto che k raggiunga n_j ma è sicuro che il suo valore è circa uguale (si approfondiranno meglio questi aspetti nel paragrafo seguente).

Quello appena detto però è solamente l'idea generale su come aggiungere i nuovi valori, l'algoritmo infatti deve eseguire questa operazione non in un nodo solo ma in s nodi.
Prima di tutto bisogna ricavare la lista di nodi radice R, per farlo basta costruire un algoritmo che parte dalla radice r di T come il seguente:

Algorithm 2.1.3: EXPAND(v, LOW, UP)

if $val(v) \leq LOW$
  then return (EXPAND$(L(v), LOW, UP) \cup$ EXPAND$(R(v), LOW, UP)$)
  else if $val(v) \leq UP$
  then return (v)
  else return $(\emptyset)$ **comment:** non ritorna alcun nodo

L'algoritmo si richiama ricorsivamente sui nodi interni di T_i (blocco-if con la condizione $val(v) \leq LOW$, passo ricorsivo) finché non raggiunge i figli delle foglie di T_i (caso base), in quel caso se il valore presente nel nodo è minore di U_i allora ritorna quel nodo. L'algoritmo è chiamato ricorsivamente sul figlio sinistro $L(v)$ che sul quello destro $R(v)$ perché i nodi radice nel sottoalbero radicato in v sono in entrambi i figli. L'insieme di nodi radice di v sarà l'unione dei nodi radice dei suoi figli $L(v)$ e $R(v)$.

Una volta trovata la lista R ad ogni nodo radice è associata una variabile booleana che indica se quel nodo è vivo o meno, all'inizio ogni radice è marcata come viva e man mano che l'elaborazione procede alcune di esse vengono marcate morte quando l'elaborazione su quel nodo si è conclusa. La prima cosa da fare è cercare tutti i nodi a cui è associato un valore buono, perché in caso contrario si sa già che quel nodo non poterà nessun nuovo nodo buono, i nodi a cui non è associato un valore buono vengono marcati come morti. Bisogna quindi trovare tutti i valori buoni dalla lista S così definita:

$$S = \{Z_1, Z_2, \ldots, Z_s\}, \qquad Z_i = val(r_i) \qquad \forall i \quad 1 \leq i \leq s$$

Si nota che esiste un relazione importante tra R e S, infatti l'algoritmo aggiornerà i valori di S più volte durante l'esecuzione tuttavia si rimanda la spiegazione in seguito. Una proprietà utile per riconoscere i valori buoni da un insieme di valori è la seguente:

Proprietà 2.1. *Se Z è un valore buono anche tutti i valori minori di Z lo sono.*

Possiamo quindi sfruttarla per evitare di fare il controllo sul valore di ogni radice, basta trovare il più grande valore buono nell'insieme S. Per fare ciò si ordina S, una volta ordinato si cerca il massimo valore buono usando un algoritmo di ricerca binaria sull'insieme ordinato.

Algorithm 2.1.4: MAXGOOD(r, n, S)

$S' \leftarrow$ SORT(S) **comment:** ordina S e lo inserisce in S'

comment: ricerca binaria su S'

$min \leftarrow 1$
$max \leftarrow s$ **comment:** dimensione di S

repeat
$\begin{cases} mid \leftarrow (min + max)/2 \\ \textbf{if } \text{ISGOOD}(r, n, S'[mid]) \\ \quad \textbf{then } min \leftarrow mid + 1 \\ \quad \textbf{else } max \leftarrow mid - 1 \end{cases}$
until $min \geq max$
$iMaxGood \leftarrow max$ **comment:** indice di S' del massimo valore buono

if $(min == max)$ **and** (!ISGOOD$(S'[min])$)
then $iMaxGood \leftarrow min - 1$
return $(S'[iMaxGood])$

Una volta trovato il valore buono massimo basta marcare come morti tutti i nodi radice che contengono un valore maggiore.

A questo punto, dopo aver escluso alcuni nodi radice marcandoli come morti si procede a cercare i nodi buoni nei sottoalberi radicati nei nodi radice rimanenti (quelli che sono ancora marcati come vivi), il modo di procedere è quello descritto per il singolo nodo. Si potrebbero quindi eseguire le operazioni che faccio sul singolo nodo su tutte le radici vive in modo sequenziale, cioè lavorare solamente su un nodo alla volta e passare al nodo successivo solamente quando si sono concluse tutte le operazioni sul nodo corrente. Questo modo di procedere però ha un difetto: tratta le operazioni sui singoli nodi in modo indipendente, il che non è vero perché non è stata sfruttata la proprietà che lega i valori buoni (Proprietà 2.1), infatti ora mostreremo come usare questa proprietà per evitare di fare il controllo del valore buono su ogni nodo dopo la ricorsione.
L'algoritmo sfrutta questa proprietà nel seguente modo: come prima cosa si effettua la chiamata ricorsiva su ogni radice viva, come input si usa sempre una variabile intera k comune a tutte le radici in questo caso. La variabile k si usa per individuare i valori candidati n_j di ogni sottoalbero radicato r_j. Al termine delle chiamate ricorsive per ogni nodo radice abbiamo un valore, su questo bisogna verificare se è buono o meno. Per fare la verifica possiamo usare l'algoritmo spiegato nel paragrafo precedente che permette di evitare di fare il controllo su ogni radice (algoritmo 2.1.4). Una volta trovate, le radici non buone vengono marcate come morte e su di esse non verrà più eseguita alcuna operazione, per quanto riguarda le radici vive r_j rimanenti si aggiorna il valore corrispondente a S, Z_j, con l'ultimo ritornato

dalla chiamata ricorsiva. Poi si ripetono le operazioni appena descritte con k raddoppiato. La relazione che lega le liste R e S è la seguente: $Z_j \in S$ è il massimo valore buono trovato nel sottoalbero radicato nel nodo radice $r_j \in R$.
Le operazioni appena descritte possono essere schematizzate per chiarezza in più punti in questo modo:

1. trovare l'insieme di radici con valori non buoni e marcarle morte;

2. chiamare l'algoritmo ricorsivamente su ogni radice r_j marcata viva con input k;

3. si prende la lista di valori ritornati dal punto precedente e si trovano quelli buoni;

4. le radici a cui corrispondono dei valori non buoni vengono marcate morte, per le altre invece si sovrascrivono i valori buoni trovati nel punto precedente nella lista S (il valore buono prodotto dalla chiamata ricorsiva sul nodo r_j va in Z_j);

5. se ci sono ancora radici vive si ricomincia dal punto 2, altrimenti si considera conclusa la ricerca di nuovi valori buoni e si procede con l'individuazione dei limiti L_i e U_i.

A questo punto rimane da spiegare come vanno calcolati i nuovi limiti inferiore e superiore: L_i e U_i.

Limite inferiore Per quanto riguarda il limite inferiore, che è quello più importante, si ragiona sui valori di S ottenuti alla fine dell'elaborazione precedentemente spiegata. La lista S all'inizio contiene i valori delle radici, con l'elaborazione successiva (punti 2, 3, 4, 5) vengono aggiornati solamente quei valori di S che in precedenza contenevano già valori buoni. Ad ogni iterazione del ciclo l'algoritmo aggiorna alcuni valori buoni di S (dipende da quanti valori buoni saranno trovati nel punto 3) con altri valori buoni che sono maggiori rispetto a quelli precedenti perché provengono dalle chiamate ricorsive del punto 2 effettuate con un k maggiore. Una volta che tutte le radici saranno marcate morte (al termine del punto 5), la lista S conterrà valori buoni e non buoni, quelli non buoni derivano da quelle radici che sono state marcate morte al punto 1 (perché non vengono più modificate nei punti successivi) mentre gli altri valori (quelli buoni) sono il risultato dell'elaborazione precedente e sono i massimi valori buoni che sono stati trovati nelle chiamate ricorsive alle radici ad essi associate. Quindi il nuovo limite inferiore L_i sarà il più grande valore buono di S.

Limite superiore In questo caso il discorso è diverso perché l'algoritmo è impostato per aggiornare il limite inferiore. Il limite superiore serve solo ad evitare il lavoro su nodi di cui è noto a priori che non porteranno all'aggiunta di alcun valore buono. Per la proprietà di heap e per la proprietà di cui godono i valori buoni; se al nodo v è associato il valore $val(v)$ non buono anche tutti i nodi nel sottoalbero radicato in v non lo sono, quindi nessuno di essi verrà aggiunto da alcuna fase. Quindi basta porre U_i come il più piccolo valore non buono trovato fino ad ora. Per fare ciò basta definire una variabile $UPPER$ che rappresenta il limite superiore calcolato fino ad ora, ogni volta che si trova un valore non buono, se questo è minore si aggiorna $UPPER$ con quel valore.

Con la definizione dei nuovi limiti si conclude la fase i-esima. Quindi per chiarire meglio la sua struttura, si può costruire un algoritmo che ha come input: la radice r di T e i limiti inferiore e superiore LOW e UP della fase precedente, in uscita ci sarà la coppia di nuovi limiti.

Algorithm 2.1.5: COMPUTEBOUNDS(r, n, LOW, UP)

$R \leftarrow$ EXPAND(r, LOW, UP) **comment:** costruiamo la lista R, di dimensione s

$S \leftarrow$ lista di dimensione s inizializzata con i valori dei nodi di R
comment: punto 1 dell'elenco precedente

$Z_{max} \leftarrow$ MAXGOOD(r, n, S)
for each $r_j \in R \quad | \quad S[j] > Z_{max}$
  do marca r_j come radice morta
comment: aggiorna il valore del limite superiore

$UPPER \leftarrow$ valore minimo in S maggiore di Z_{max}
$UPPER \leftarrow$ MIN$(UP, UPPER)$
$k \leftarrow 1$
$S_{temp} \leftarrow$ lista copia di S
comment: ripete i passi finché tutte le radici sono morte

while c'è qualche radice marcata viva
do
- **comment:** punti 2,3,4 dell'elenco precedente
- **for each** radice r_j marcata viva
  do $S_{temp}[j] \leftarrow$ SELECT(r_j, k)
- $Z_{max} \leftarrow$ MAXGOOD(r, n, S_{temp})
- **for each** $r_j \in R \quad | \quad S_{temp}[j] > Z_{max}$
  do marca r_j come radice morta
- **comment:** aggiorna il valore del limite superiore
- $U \leftarrow$ valore minimo in S_{temp} maggiore di Z_{max}
- $UPPER \leftarrow$ MIN$(U, UPPER)$
- **comment:** sostituisce in S i nuovi valori buoni
- **for each** radice r_j marcata viva
  do $S[j] \leftarrow S_{temp}[j]$
- $k \leftarrow 2 \cdot k$

comment: calcolo dei nuovo limite L_i da S

$LOWER \leftarrow$ il più grande valore buono di S
return $(LOWER, UPPER)$ **comment:** la coppia di nuovi limiti inferiore e superiore

Per quanto riguarda il susseguirsi delle fasi, per capire quando fermarsi, basta notare che al termine di ogni fase $L_{i-1} < L_i$, dato che i valori sono tutti distinti tra loro, perché verrà aggiunto almeno un nuovo valore buono in ogni fase. Per la definizione di valore buono è facilmente deducibile che finché i valori sono tutti distinti tra loro si ha che il numero massimo di valori buoni è n. Di conseguenza al termine di ogni fase basta controllare se

$$good(r, n, L_i) < n$$

se è vera significa che si possono ancora aggiungere dei valori buoni quindi si procede con la fase $i+1$ esima. Altrimenti, se la condizione non è vera, sarà per forza:

$$good(r, n, L_i) = n$$

perché non è possibile avere $good(r, n, L_i) > n$. Al termine quindi il valore da ritornare sarà proprio L_i. Con la definizione della condizione di uscita si può costruire un pseudocodice per l'algoritmo A:

Algorithm 2.1.6: SELECT(r, n)

$LOW \leftarrow$ VAL(r)
$UP \leftarrow +\infty$
while GOOD$(r, n, LOW) < n$
  do $(LOW, UP) \leftarrow$ COMPUTEBOUNDS(r, n, LOW, UP)
return (LOW)

A questo punto la spiegazione dell'algoritmo è conclusa, ora è possibile specificare meglio la relazione lasciata in sospeso all'inizio, riguardante la relazione che lega tra loro i limiti calcolati nelle varie fasi:

$$L_{i-1} < L_i \leqslant NTH < U_i \leqslant U_{i-1}$$

La relazione $L_{i-1} < L_i$ è appena stata trattata nel paragrafo precedente. Per spiegare la relazione $L_i \leqslant NTH < U_i$ basta guardare la definizione di valore buono, infatti L_i può essere uguale a NTH, cosa che succede al termine dell'ultima fase. Per quanto riguarda la seconda parte ($NTH < U_i$), dalla definizione di limite superiore, sappiamo che è sempre un valore non buono quindi non può mai essere uguale a NTH.
L'ultima relazione ($U_i \leqslant U_{i-1}$) è meno ovvia delle precedenti, per dimostrarla consideriamo un albero T dove i figli sinistro e destro della radice r soddisfano le seguenti caratteristiche: il figlio sinistro $L(r)$ (consideriamo il figlio sinistro per semplicità) contiene il più basso valore non buono, l'altro figlio $R(r)$ deve essere per forza essere buono (perché altrimenti non ci sarebbe alcun valore buono in T). Per la proprietà di heap e di valore buono tutti nodi buoni di T devono essere contenuti nel sottoalbero radicato in $R(r)$. La prima fase dell'algoritmo è caratterizzata dall'avere T_i composto solamente dalla radice r ($L_0 = val(r)$ e $U_0 = +\infty$), l'insieme delle radici per la prima fase è quindi $R = \{L(r), R(r)\}$. All'inizio S conterrà $val(L(r))$ che è il più piccolo valore non buono e $val(R(r))$ che sarà un valore buono. Con la prima scansione di S, fatta nella prima fase, l'algoritmo trova subito che $val(L(r))$ non è buono e lo fa diventare subito limite superiore temporaneo $UPPER = val(L(r))$ (perché $U_0 = +\infty$). Durante l'elaborazione verranno trovati altri valori non buoni, ma saranno tutti maggiori di $val(L(r))$ perché abbiamo ipotizzato all'inizio che fosse il minimo. Di conseguenza al termine

della prima fase $val(L(r))$ sarà il limite superiore U_1 trovato. A questo punto U_1 coincide già con il valore minimo non buono, quindi nelle fasi successive i valori non buoni trovati saranno tutti maggiori. Questo comporta che i limiti superiori calcolati nella fasi successive U_i saranno tutti uguali a quello della prima U_1.

2.2 Calcolo della complessità dell'algoritmo A

Prima di tutto sia $f(n)$ la complessità temporale dell'algoritmo A.

Nella generica fase i-esima avremo il contributo principalmente di due componenti:

- il costo delle chiamate ricorsive all'algoritmo A e
- il costo per capire ogni volta quali sono le radici buone e quali no

Primo punto Se consideriamo una delle radici iniziali per la fase, $r_j \in R$, sappiamo che produrrà eventualmente n_j nuovi valori buoni. Questi valori verranno calcolati richiamando ricorsivamente l'algoritmo A raddoppiando k ad ogni iterazione. Si avrà quindi:

$$f(1) + f(2) + f(4) + \ldots + f(2^{t-1}) + f(2^t), \qquad 2^{t-1} \leq n_j < 2^t \tag{2.1}$$

Dato che il problema $SELECT(n)$ ha complessità temporale $\Omega(n)$ possiamo approssimare la sommatoria con l'ultimo termine $f(2^t)$. Per la seconda disuguaglianza si vede che n_j è $\Theta(2^t)$. Per cui per qualche costante c_1 questa sommatoria è limitata da

$$c_1 f(n_j)$$

Questa sommatoria vale per ogni nodo $r_j \in R$, quindi il primo punto è minore di:

$$\sum_{j=1}^{s} c_1 f(n_j) = c_1 \sum_{j=1}^{s} f(n_j)$$

Secondo punto L'algoritmo per calcolare se un valore è buono o meno (algoritmo 2.1.2)ha complessità temporale $O(n)$ come già detto e ci sono s valori da elaborare.
Il costo per ordinare gli s valori è $O(s \log(s))$ e, una volta ordinati, per trovare l'elemento massimo buono grazie alla ricerca binaria si ha $O(n \log(s))$ perché ad ogni passo della ricerca è necessario testare se il valore corrente è buono o meno. Per $s \leq n$ questo passo è $O(n \log(n))$.
Il tutto è ripetuto finché c'è qualche radice viva in R e dato che ogni radice r_j produrrà n_j valori buoni, si continuerà finché la radice con l'n_j massimo non porterà ad un k-esimo elemento minore non buono.

La ricerca di questo valore viene fatta raddoppiando k ad ogni iterazione quindi, se t è il numero di iterazioni da fare, ci si ferma quando:

$$2^{t-1} \leq \max_j\{n_j\} < 2^t$$

cioè 2^t è $\Theta(\max_j\{n_j\})$.
Dato però che non conosciamo $\max_j\{n_j\}$ possiamo trovare un limite superiore ad esso:

$$\max_j\{n_j\} \leq \sum_{j=1}^{s} n_j = g(n) \leq n$$

Possiamo dire che 2^t è $O(n)$ e quindi t è $O(\log(n))$. Quindi per qualche costante c_2 il secondo punto è limitato da:

$$c_2 n \log^2(n)$$

Complessità totale Unendo i risultati dei due punti è possibile dire che per una costante $c = \max\{c_1, c_2\}$ la fase i-esima è limitata da:

$$c\left[\sum_{j=1}^{s} f(n_j) + n\log^2(n)\right]$$

Per quanto detto in precedenza verrebbe da pensare che il numero di nodi aggiunti nella fase i-esima sia esattamente $g(n)$, però in realtà ne potrebbero essere aggiunti di più. Per capire questa cosa consideriamo l'algoritmo 2.1.5 che calcola i nuovi limiti della fase. Al termine del ciclo while la lista S contiene tutti i valori buoni massimi associati ad ogni nodo radice. Supponendo che la lista S sia composta dei seguenti valori buoni $\{Z_1, Z_2, Z_3\}$ dove il generico Z_j porta all'individuazione di n_j valori buoni nel sottoalbero radicato nella radice r_j, quindi $g(n) = n_1 + n_2 + n_3$. Ipotizziamo che il valore massimo dei tre sia Z_2, secondo l'algoritmo, il limite inferiore L_i coincide con Z_2 quindi in realtà verranno aggiunti tutti i valori minori di esso collocati in ogni sottoalbero radicato nelle radici r_1, r_2, r_3. Nel caso in cui ci siano dei valori buoni nei sottoalberi radicati in r_1 e r_3 rispettivamente compresi negli intervalli $(Z_1, Z_2]$ e $(Z_3, Z_2]$, saranno anch'essi aggiunti nella fase i-esima oltre ai $g(n)$ valori già previsti.
Come detto in precedenza ci sono al massimo n nodi buoni e una volta trovati tutti l'algoritmo si arresterà, dato che in ogni fase vengono aggiunti almeno $g(n)$ nodi, il numero delle fasi può essere al massimo:

$$\frac{n}{g(n)}$$

In totale la complessità dell'algoritmo A sarà limitata da:

$$f(n) \leq \frac{n \cdot c}{g(n)}\left[\sum_{j=1}^{s} f(n_j) + n\log^2(n)\right]$$

Ora l'obbiettivo è quello di semplificare questa espressione cercando di eliminare la sommatoria interna alle parentesi quadre.

Prima di far vedere come eliminare la sommatoria dimostriamo una relazione che sarà utile per fare ciò:

Teorema 2.2.1. *Sia $f(n)$ una funzione convessa, x_1 e x_2 due numeri interi appartenenti al dominio di $f(n)$ e $\lambda \in [0;1]$ un numero reale. Allora vale:*

$$f\left(\lambda x_1 + (1-\lambda)\, x_2\right) \leq \lambda f(x_1) + (1-\lambda) f(x_2) \tag{2.2}$$

Dimostrazione. Questa relazione deriva da una disequazione più generale detta disequazione di Jensen la quale afferma che per una funzione convessa $f(n)$ una successione finita di dimesione k $\{x_1, \cdots, x_k\}$ e una successione di numeri non negativi $\{a_1, \cdots, a_k\}$ si ha che:

$$f\left(\frac{\sum_i x_i a_i}{\sum_j a_j}\right) \leq \frac{\sum_i f(x_i) a_i}{\sum_j a_j}$$

Per $k = 2$ si ha

$$f\left(\frac{a_1 x_1 + a_2 x_2}{a_1 + a_2}\right) \leq \frac{a_1 f(x_1) + a_2 f(x_2)}{a_1 + a_2} \tag{2.3}$$

Ponendo $a_1 = \lambda$ e $a_2 = 1 - \lambda$, dato che $\lambda \in [0;1]$ si ha che:

$$a_1 = \lambda \geq 0 \qquad \text{e che} \qquad a_2 = 1 - \lambda \geq 0$$

inoltre vale $a_1 + a_2 = 1$, si può quindi riscrivere la disequazione 2.2 nel seguente modo:

$$f\left(\frac{a_1 x_1 + a_2 x_2}{a_1 + a_2}\right) \leq \frac{a_1 f(x_1) + a_2 f(x_2)}{a_1 + a_2}$$

A questo la disequazione 2.2 è stata ricondotta all'equazione 2.3 e valgono le ipotesi della disequazione di Jensen. Di conseguenza la 2.2 vale. □

Teorema 2.2.2. *Sia $f(n)$ una funzione convessa con $f(0) \leq 0$ e $\{n_j\}$ una successione finita di numeri interi. Allora vale:*

$$\sum_j f(n_j) \leq f\left(\sum_j n_j\right)$$

Dimostrazione. Sia

$$N = \sum_{j=1}^{s} n_j$$

possiamo quindi scrivere la prima parte della disequazione nel seguente modo:

$$\sum_j f(n_j) = \sum_j f(N\frac{n_j}{N})$$

A questo punto sia $\lambda \in [0;1]$, per il Teorema 2.2.1 possiamo scrivere:

$$f(\lambda x) = f(\lambda x + (1-\lambda)0) \leq \lambda f(x) + (1-\lambda)f(0)$$

Dato che $f(0) \leq 0$ si ha che:

$$f(\lambda x) = f(\lambda x + (1-\lambda)0) \leq \lambda f(x) + (1-\lambda)f(0) \leq \lambda f(x)$$

Quindi:

$$f(\lambda x) \leq \lambda f(x) \tag{2.4}$$

Dato che $0 \leq \frac{n_j}{N} \leq 1$ (si vede dalla definizione di N) si può porre $\lambda = \frac{n_j}{N}$ quindi:

$$\sum_j f(N\frac{n_j}{N}) = \sum_j f(N \cdot \lambda)$$

Usato la 2.4 nell'espressione appena trovata:

$$\sum_j f(N \cdot \lambda) \leq \sum_j \lambda f(N)$$

dato che N non dipende da j si ottiene:

$$\sum_j f(N \cdot \lambda) \leq \sum_j \lambda f(N) = \frac{f(N)}{N} \sum_j n_j = \frac{f(N)}{N} N = f(N)$$

ricordando le definizioni di N e λ si ottiene infine:

$$\sum_j f(n_j) \leq f\left(\sum_j n_j\right)$$

□

Se si sceglie $f(n)$ convessa tale che $f(0) \leq 0$ è possibile usare questo teorema su

$$f(n) \leq \frac{n \cdot c}{g(n)} \left[\sum_{j=1}^{s} f(n_j) + nlog^2(n)\right] \qquad \text{dove} \qquad \sum_{j=1}^{s} n_j = g(n)$$

ottenendo

$$f(n) \leq \frac{n \cdot c}{g(n)} \left[f(g(n)) + nlog^2(n) \right]$$

Sia ora $f(n) = nlog^2(n)h(n)$. Allora:

$$nlog^2(n)h(n) \leq \frac{n \cdot c}{g(n)} \left[g(n)log^2[g(n)] \cdot h[g(n)] + nlog^2(n) \right]$$

$$h(n) \leq \frac{c}{g(n)log^2 n} \left[g(n)log^2[g(n)] \cdot h[g(n)] + nlog^2(n) \right] = c \left[\left(\frac{log[g(n)]}{log(n)} \right)^2 h[g(n)] + \frac{n}{g(n)} \right]$$

Sfruttando $g(n) \leq n$:

$$h(n) \leq c \left[h[g(n)] + \frac{n}{g(n)} \right]$$

Ora si può applicare questa disequazione più volte su $h(n)$ ottenendo:

$$h(n) \leq c \left[h[g(n)] + \frac{n}{g(n)} \right] \leq c \cdot h[g(n)] + \frac{c \cdot n}{g(n)} \leq$$

$$\leq c \left(c \cdot h[g(g(n))] + \frac{c \cdot g(n)}{g(g(n))} \right) + \frac{c \cdot n}{g(n)} \leq$$

$$\leq \frac{c \cdot n}{g(n)} + \frac{c^2 \cdot g(n)}{g(g(n))} + c^2 h[g(g(n))] \leq$$

$$\leq \frac{c \cdot n}{g(n)} + \frac{c^2 \cdot g(n)}{g(g(n))} + \frac{c^3 \cdot g(g(n))}{g(g(g(n)))} + c^3 h[g(g(g(n)))] \leq$$

$$\leq \frac{c \cdot n}{g(n)} + \frac{c^2 \cdot g(n)}{g(g(n))} + \frac{c^3 \cdot g(g(n))}{g(g(g(n)))} + \cdots + \frac{c^t \cdot g^{[t-1]}(n)}{g^{[t]}(n))} + c^t h[g^{[t]}(n)]$$

Nel prossimo paragrafo vedremo che una condizione fondamentale che deve soddisfare $g(n)$ è che $g^{[t]}(n) = 1$. Quindi l'ultimo termine della sommatoria appena scritta è $c^t h(1)$ e per la scelta finale di $h(\cdot)$ a cui si arriverà si avrà $h(1) = 0$, di conseguenza il termine $c^t h(1)$ si può eliminare. Quindi l'espressione su cui ora si lavorerà ora è la seguente:

$$h(n) \leq \frac{c \cdot n}{g(n)} + \frac{c^2 \cdot g(n)}{g(g(n))} + \frac{c^3 \cdot g(g(n))}{g(g(g(n)))} + \cdots + \frac{c^t \cdot g^{[t-1]}(n)}{g^{[t]}(n))} \tag{2.5}$$

A questo punto è necessario trovare $g(n)$ e t tali che soddisfano la seguente proposizione:

Proposizione 2.1. *Le due condizioni che devono soddisfare $g(n)$ e t sono:*

1. *la sommatoria a destra della disequazione 2.5 è minima rispetto a $g(n)$ e t;*

2. $g^{[t]}(n) = 1$.

Il primo punto è dovuto al fatto che stiamo cercando la migliore complessità temporale (a discapito di quella nello spazio) per l'algoritmo A. Mentre la seconda condizione è necessaria per il funzionamento dell'algoritmo, per capire meglio il perché di questa cosa bisogna ricordarsi che $g(n)$ è il numero di valori buoni che vengono aggiunti nella fase i-esima. Consideriamo l'istanza X dell'algoritmo che viene eseguito con input n, nella fase i-esima dell'istanza X vengono aggiunti $g(n)$ nodi. L'istanza X chiama ricorsivamente l'algoritmo con diversi input e per la relazione 2.1 e il Teorema 2.2.2 possiamo approssimare questa chiamata ricorsiva con un istanza Y eseguita con input $g(n)$. Quindi $g(g(n))$ è il numero di nodi aggiunti nella fase i-esima dall'istanza Y. L'istanza Y dell'algoritmo chiamerà a sua volta l'algoritmo ricorsivamente, generando un'altra istanza con input $g(g(n))$ la cui fase i-esima aggiungerà $g(g(g(n)))$ nodi. Se supponiamo che $g(n)$ sia una funzione strettamente crescente, dato che $g(n) < n$ si che: $g(g(n)) < g(n)$ e più in generale per una generico intero j, $g^{[j]} < g^{[j-1]}$. Quindi una generica istanza dell'algoritmo genererà con una chiamata ricorsiva un altra istanza dell'algoritmo con un input di dimensione minore. Questo processo tuttavia non si può ripetere all'infinito perché prima o poi si giungerà ad un punto in cui ci sarà una generica variabile t tale che $g^{[t]} = 1$ (questo è il caso base dell'algoritmo A). Infatti per come è costruito l'algoritmo, quando viene eseguito con un input di dimensione 1 non vengono effettuate ulteriori chiamate ricorsive.

A questo punto cercheremo di capire qual'è la condizione per cui la sommatoria 2.5 è minima. Fissato t si può notare che i termini della sommatoria hanno prodotto costante infatti:

$$\frac{c \cdot n}{g(n)} \cdot \frac{c^2 \cdot g(n)}{g(g(n))} \cdot \frac{c^3 \cdot g(g(n))}{g(g(g(n)))} \cdot \dots \cdot \frac{c^t \cdot g^{[t-1]}(n)}{g^{[t]}(n))} = c^{1+2+3+\dots+t} n = c^{\frac{t(t+1)}{2}} n$$

Dato che il prodotto dei termini è costante, la loro sommatoria è minimizzata quando tutti loro sono uguali. Infatti vale il seguente teorema.

Teorema 2.2.3. *Sia $X_1, ..., X_t$ un insieme di variabili, $X_i \in R$, dove il loro prodotto è uguale a $K \neq 0$ costante. Allora la loro è sommatoria quando sono tutti uguali tra loro, cioè:*

$$X_i = X_j = \sqrt[t]{K} \qquad \forall i, j \tag{2.6}$$

Dimostrazione. Dato che il prodotto delle variabili è costante si può scrivere:

$$\prod_{i=1}^{t} X_i = X_1 \cdot \prod_{i=2}^{t} X_i = K \qquad X_1 = \frac{K}{\prod_{i=2}^{t} X_i}$$

Sia $f(X_1, ..., X_t) = \sum_{i=1}^{t} X_i$, l'obbiettivo è calcolare il minimo di questa funzione a più variabili, per fare ciò basta calcolare ogni derivata parziale di $f(X_1, ..., X_t)$ e annullarla, cioè $\nabla f(X_1, ..., X_t) = 0$.
Dato che X_1 è stata espressa in funzione delle altre:

$$f(X_1, ..., X_t) = \frac{K}{\prod_{i=2}^{t} X_i} + X_2 + X_3 + \cdots + X_t$$

La derivata rispetto alla generica variabile X_j, $j \neq 1$ è:

$$\frac{\partial}{\partial X_j} f(X_1, ..., X_t) = \frac{-K}{X_j^2 \prod_{i=2, i\neq j}^{t} X_i} + 1 = 0$$

$$\frac{-K}{X_j^2 \prod_{i=2, i\neq j}^{t} X_i} + 1 = 0 \iff X_j^2 \prod_{i=2, i\neq j}^{t} X_i - K = 0 \iff X_j^2 \prod_{i=2, i\neq j}^{t} X_i = K$$

Deve valere per ogni variabile X_j, $j \neq 1$ quindi prendendo una generica coppia di variabili X_i e X_j dato che K è sempre lo stesso:

$$X_i^2 \prod_{l=2, l\neq i}^{t} X_l = K = X_j^2 \prod_{l=2, l\neq j}^{t} X_l$$

$$X_1 \cdot X_2 \cdot \cdots \cdot X_i^2 \cdot \cdots \cdot X_t = X_1 \cdot X_2 \cdot \cdots \cdot X_j^2 \cdot \cdots \cdot X_t$$

Dividendo a sinistra e a destra per $\prod_{l=1}^{t} X_l$ diverso da 0 perché $K \neq 0$ si ottiene:

$$X_i = X_j \qquad \forall i, j \neq 1$$

Lo stesso procedimento può essere fatto esplicitando all'inizio X_2 per esempio ottenendo:

$$X_i = X_j \qquad \forall i, j \neq 2$$

Dato che devono valere entrambe, si ha che:

$$X_1 = X_2 = \cdots = X_t = \sqrt[t]{K}$$

□

In base al lavoro di Karp,Saks e Wigderson [1] $g(n)$ e t sono scelte in questo modo:

$$g(n) = \frac{n}{c^{\sqrt{2\log_c n} - \frac{1}{2}}} \qquad t = \sqrt{2\log_c n} \tag{2.7}$$

Verifichiamo quindi che siano soddisfatte la condizione 1 e 2 della Prosizione 2.1.
Per il Teorema 2.2.3 la sommatoria è minimizzata se vale la 2.6, quindi deve essere:

$$\frac{c \cdot n}{g(n)} = \frac{c^2 \cdot g(n)}{g(g(n))} = \frac{c^3 \cdot g(g(n))}{g(g(g(n)))} = \cdots = \frac{c^t \cdot g^{[t-1]}(n)}{g^{[t]}(n))} \tag{2.8}$$

Usando la 2.7 si può scrivere $g(n)$ in funzione di t in questo modo:

$$g(n) = \frac{n}{c^{t-\frac{1}{2}}}$$

Il primo termine della 2.8 può essere riscritto come:

$$\frac{c \cdot n}{g(n)} = \frac{c \cdot n}{\frac{n}{c^{t-\frac{1}{2}}}} = c^{t+\frac{1}{2}}$$

A questo punto non resta che verificare che le due condizioni della proposizione 2.1 siano soddisfatte.

Teorema 2.2.4. *Sia* $t = \sqrt{2\log_c n}$ *e* $g(n) = \frac{n}{c^{\sqrt{2\log_c n}-\frac{1}{2}}}$. *Allora valgono:*

(a)

$$\frac{c^i \cdot g^{[i-1]}(n)}{g^{[i]}(n)} = c^{t+\frac{1}{2}} \qquad \forall\, 1 \leq i \leq t$$

(b)

$$g^{[t]}(n) = 1$$

Dimostrazione. Ipotizzando che $g^{[0]}(n) = n$.

Dimostrazione (a) Dimostriamo il punto (a) per induzione su i.

Caso base Per $i = 1$:

$$\frac{c \cdot g^{[0]}(n)}{g(n)} = \frac{c \cdot n}{g(n)} = c^{1+\sqrt{2\log_c n}-\frac{1}{2}} = c^{\sqrt{2\log_c n}+\frac{1}{2}} = c^{t+\frac{1}{2}}$$

Passo induttivo Supponiamo che valga la (a) fino ad i:

$$\frac{c^k \cdot g^{[k-1]}(n)}{g^{[k]}(n)} = c^{t+\frac{1}{2}} \qquad \forall\, 1 \leq k \leq i$$

Verifichiamo la (a) per $i+1$:

$$c^{i+1}\frac{g^{[i]}(n)}{g^{[i+1]}(n)} = c \cdot c^i \frac{g^{[i-1]}(g(n))}{g^{[i]}(g(n))} \overset{ind.}{=} c \cdot c^{\frac{1}{2}+\sqrt{2\log_c g(n)}} =$$

$$= c^{1+\frac{1}{2}+\sqrt{2\log_c g(n)}}$$

Perché valga deve essere:

$$c^{1+\frac{1}{2}+\sqrt{2\log_c g(n)}} = c^{\frac{1}{2}+\sqrt{2\log_c n}} \iff 1 + \sqrt{2\log_c g(n)} = \sqrt{2\log_c n} \iff$$

$$\iff 2\log_c n + 1 - 2\sqrt{2\log_c n} = 2\log_c g(n) = 2\log_c n - 2\sqrt{2\log_c n} + 1$$

Dimostrazione (b) Per quanto riguarda il punto (b), facendo il prodotto di tutti i termini della 2.8:

$$\prod_{i=1}^{t} \frac{c^i \cdot g^{[i-1]}(n)}{g^{[i]}(n)} = c^{\frac{t(t+1)}{2}} \frac{n}{g^{[t]}(n)}$$

Ma questa produttoria può essere risolta sfruttando il punto (a) appena dimostrato e l'ipotesi che $g^{[0]}(n) = n$, si ottiene:

$$\prod_{i=1}^{t} \frac{c^i \cdot g^{[i-1]}(n)}{g^{[i]}(n)} = \left(c^{t+\frac{1}{2}}\right)^t = c^{\frac{t(t+1)}{2}+\frac{t^2}{2}} = c^{\frac{t(t+1)}{2}} n$$

Quindi:

$$c^{\frac{t(t+1)}{2}} \frac{n}{g^{[t]}(n)} = c^{\frac{t(t+1)}{2}} n \Longrightarrow g^{[t]}(n) = 1$$

□

A questo punto si può calcolare la complessità dell'algirtimo A. In base alla 2.5, sfruttando il fatto che

$$\frac{c \cdot n}{g(n)} = \sqrt{c} \cdot c^{\sqrt{2 \log_c(n)}}$$

Si ha:

$$h(n) \leq \sqrt{2 \log_c(n)} \cdot \sqrt{c} \cdot c^{\sqrt{2 \log_c(n)}} = 2^{O(\sqrt{\log(n)})}$$

La complessità temporale dell'algoritmo A è quindi:

$$f(n) = n \cdot \log^2(n) 2^{O(\sqrt{\log(n)})} = n \cdot 2^{O(\sqrt{\log(n)})}$$

2.2.1 Spazio occupato

In base al lavoro fatto da Karp, Saks, e Wigderson [1] lo spazio occupato dall'algoritmo A è:

$$S_A(n) = O(n)$$

2.3 Algoritmo B

In questa sezione verrà presentato l'algoritmo B come variante dell'algoritmo A. L'algoritmo A usa un numero costante di variabili per livello di ricorsione, tranne nel momento in cui si trovano i valori buoni della lista $S = \{Z_1, Z_2, \cdots, Z_s\}$. Per fare ciò l'algoritmo A ordina i valori di S e poi applica la ricerca binaria sulla lista ordinata (algoritmo 2.1.4). L'ordinamento porta ad un consumo di spazio proporzionale al numero di valori da ordinare che in questo caso è s, il quale è un valore che cambia ad ogni fase. L'algoritmo B si differenzia da quello A perché ricava i valori buoni di S senza ordinarne gli elementi, con un consumo di spazio costante e mantenendo invariata la complessità temporale totale.

Illustriamo ora la variante dell'algoritmo 2.1.4 adottata dall'algoritmo B. Sia $R = \{r_1, r_2, \cdots, r_s\}$ la lista di radici della fase i-esima ordinate secondo l'ordine in cui sarebbero visitate in una ricerca in profondità dell'albero di T_i.

L'algoritmo che verrà ora presentato si differenzia leggermente da quello presentato da Karp, Saks e Wigderson [1] in quanto il loro è un algoritmo randomizzato Montecarlo, quello che verrà presentato ora è sempre randomizzato però LasVegas, l'idea di base è sempre la stessa e pure l'analisi.
L'idea è quella fare una specie di ricerca binaria in un insieme non ordinato per trovare il valore massimo buono Z_{max}, usando una coppia di variabili Z_L e Z_U che compiono salti in modo aleatorio ad ogni passo della ricerca. Queste due variabili rappresentano rispettivamente: Z_L il più grande valore buono e Z_U il più piccolo valore non buono trovati fino a quel momento nella ricerca. Quindi si avrà che Z_L e Z_U definiscono un intervallo $I = [Z_L, Z_U)$ all'interno del quale è presente il valore da cercare, $Z_{max} \in I$.

Inizialmente

$$Z_L = -\infty \qquad \text{e} \qquad Z_U = +\infty$$

Sia l il numero di Z_p di S tali che $Z_p \in I$. Il corpo dell'algoritmo è il seguente, e viene ripetutamente eseguito finché $l > 1$ (quando I contiene più di un valore di S):

1. Sia h il numero di valori di S nell'intervallo $I' = (Z_L, Z_U) \subset I$;
2. Sia j un valore casuale in $[1; h]$;
3. Trova il j-esimo valore Z_m di S che appartiene all'insieme I', i valori di S si considerano ordinati secondo l'ordine in cui si presentano nella lista;
4. Se Z_m è buona allora $Z_L \leftarrow Z_m$, altrimenti $Z_U \leftarrow Z_m$.
5. Aggiorna l

Questo è un algoritmo LasVegas infatti siamo sicuri che prima o poi terminerà. Infatti ad ogni passo l'algoritmo esclude sempre qualche valore candidato perché sceglie un valore a caso di S nell'intervallo I', che è l'insieme I privato del limite inferiore Z_L. Questo è fondamentale perché se nella scelta casuale di Z_m fosse scelto quello i limiti Z_L e Z_U non subirebbero modifiche (potrebbe ripetersi questa cosa all'infinito), negli altri casi invece viene sempre escluso almeno un valore candidato. Quindi prima o poi I individuerà un solo elemento di S.
Dato che tutti i valori sono tra loro distinti il numero di valori h di S in I' è $l - 1$ perché I e I' si distinguono tra loro solamente per il valore X_L.

Possiamo costruire un pseudocodice per quello appena detto, l'input è lo stesso dell'algoritmo 2.1.4: la radice r di T, l'intero n e l'insieme S, l'output è il massimo valore buono di S.

Algorithm 2.3.1: MAXGOODB(r, n, S)

$Z_L \leftarrow -\infty$
$Z_U \leftarrow +\infty$
$l \leftarrow s$ **comment:** dimensione di S

repeat
$h \leftarrow l - 1$
$j \leftarrow$ RANDOM$(1, h)$ **comment:** valore casuale in $[1; h]$
$k \leftarrow 0$
$i \leftarrow 0$
repeat
comment: cerca il j-esima valore di S che sta in I'
$i \leftarrow i + 1$
if $Z_L < Z_i < Z_U$
then $k \leftarrow k + 1$
until $k \geq j$
$Z_m \leftarrow S[i]$
if ISGOOD(r, n, Z_m) **comment:** aggiorna il valore di uno dei limiti
then $Z_L \leftarrow Z_m$
else $Z_U \leftarrow Z_m$
$l \leftarrow 0$ **comment:** aggiorna il valore di l
for $i \leftarrow 1$ **to** s
do **if** $Z_L \leq Z_i < Z_U$
then $l \leftarrow l + 1$
until $l \leq 1$
return (l'unico valore di S nell'intervallo I)

Il funzionamento del resto dell'algoritmo B è identico a quello A, quindi il pseudocodice per l'algoritmo B è lo stesso dell'A con la sostituzione del 2.1.4 con il 2.3.1.

2.4 Calcolo della complessità dell'algoritmo B

Come prima cosa bisogna calcolare la complessità dell'algoritmo 2.3.1 appena spiegato. Dato che questo algoritmo è randomizzato bisogna valutare la sua complessità temporale $T(s)$ nei vari casi.

Caso peggiore Ad ogni iterazione l'algoritmo prende la j-esima radice r_m con il valore massimo o minimo dell'insieme, quindi l diminuisce solamente

di un unità per la prossima iterazione.
Se $T(l)$ è la la sua complessità temporale sarà:

$$T(l) = n + T(l-1)$$

all'inizio il numero di valori è s, quindi la complessità nel caso peggiore è:

$$\Theta(n \cdot s)$$

Caso medio Se j è preso casualmente nell'insieme $[1, h]$ la probabilità che j sia uguale ad un particolare valore J è $P(j = J) = \frac{1}{h} = \frac{1}{l-1}$.
Visto che j è presa a caso non è nemmeno possibile fare delle ipotesi se Z_m sarà buono o no, però se supponiamo che ad ogni iterazione la proporzione tra valori buoni e non sia più o meno sempre la stessa è possibile definire: p la probabilità che un dato valore Z_m sia buono.
Se $T(l)$ è la la sua complessità temporale sarà:

$$T(l) = n + \frac{1}{l-1} \sum_{i=1}^{l-1} [p \cdot T(l-i) + (1-p) \cdot T(i)] =$$

$$= n + \frac{p}{l-1} \sum_{i=1}^{l-1} [T(l-i)] + \frac{1-p}{l-1} \sum_{i=1}^{l-1} [T(i)]$$

Però si vede subito che le due sommatorie sono identiche: vengono sommati gli stessi elementi solamente nell'ordine opposto, è possibile quindi raccoglierle assieme

$$T(l) = n + \frac{1}{l-1} \sum_{i=1}^{l-1} T(i)$$

moltiplicando per $l-1$ a sinistra e a destra

$$(l-1) \cdot T(l) = n \cdot (l-1) + \sum_{i=1}^{l-1} T(i)$$

questa espressione vale anche calcolata in $l-1$ quindi

$$(l-2) \cdot T(l-1) = n \cdot (l-2) + \sum_{i=1}^{l-2} T(i)$$

sottraendo la seconda nella prima si ottiene:

$$(l-1) \cdot T(l) - (l-2) \cdot T(l-1) = n \cdot (l-1-l+2) + T(l-1)$$

portando a sinistra $T(l)$ e a destra $T(l-1)$ e dividendo per $l-1$:

$$T(l) = \frac{n}{l-1} + T(l-1)$$

La prima iterazione inizia con s valori e si conclude quando ne resta solo 1 quindi:

$$\begin{cases} T(l) = \frac{n}{l-1} + T(l-1) & l > 1 \\ T(l) = n & l = 1 \end{cases}$$

$T(s)$ può essere espressa in forma chiusa come:

$$T(s) = n \sum_{l=1}^{s} \frac{1}{l} + n$$

La sommatoria può essere limitata superiormente con l'integrale associato

$$\sum_{l=1}^{s} \frac{1}{l} \leq \int_{1}^{s} \frac{1}{l} dl = \ln(s)$$

La complessità nel caso medio è quindi:

$$O(n \log(s) + n) = O(n \log(s))$$

Caso migliore Si verifica quando sia nella prima iterazione che nella seconda, la radice r_m in esame è proprio quella con Z_{max}, in questo modo ho solo due iterazioni, per cui la complessità è:

$$\Theta(n)$$

Complessità temporale totale Nel caso medio la complessità temporale per trovare l'insieme di valori buoni dell'algoritmo 2.3.1 è la stessa dell'algoritmo 2.1.4, quindi anche per l'algoritmo B si ha:

$$T_B(n) = n \cdot 2^{O(\sqrt{\log(n)})}$$

2.4.1 Spazio occupato

Per quanto riguarda lo spazio occupato ad ogni livello di ricorsione è usato un numero fisso di variabili, $O(1)$, e ci sono $t = O(\sqrt{\log(n)})$ livelli di ricorsione quindi, se $S_B(n)$ è lo spazio occupato:

$$S_B(n) = O(\sqrt{\log(n)})$$

2.5 Algoritmo C

Questo algoritmo è sempre una variante di quello A e anche questo, come il B, si differenzia perché ricava l'elenco di valori buoni (algoritmo 2.1.4) nello stesso tempo però con un minor consumo di memoria.
La differenza rispetto all'algoritmo B sta nel fatto che non è randomizzato ma deterministico.

In questo caso l'obbiettivo è realizzare una ricerca binaria in una lista non ordinata $S = \{Z_1, Z_2, \cdots, Z_s\}$ dove l'accesso agli elementi della lista non è casuale ma sequenziale da sinistra a destra, anche in più volte se necessario. Per fare questo è possibile usare un algoritmo sviluppato da Munro e Paterson [2] che hanno dimostrato che, nelle condizioni descritte sopra, è possibile trovare l'i-esimo elemento più piccolo da una lista s valori in $O(\log(s))$ passate su tutta la lista con l'uso di $O(\log^2(s))$ variabili. Possiamo quindi supporre l'esistenza di un algoritmo GETIEST(S, i) che ritorna l'i-esimo elemento più piccolo della lista S.

Usando questo risultato è possibile fare una ricerca binaria. Dove nel generico passo della ricerca ci saranno le seguenti azioni da fare:

- calcolo per capire se il valore Z_m in esame è buono o no
- spostarsi nel valore successivo per la ricerca, per fare questo non serve avere la lista di valori ordinata perché è possibile usare l'algoritmo GETIEST(S, i) sopra citato.

Detto questo possiamo costruire anche per l'algoritmo C una versione dell'algoritmo 2.1.4 dell'algoritmo A:

Algorithm 2.5.1: MAXGOODC(r, n, S)

$min \leftarrow 1$
$max \leftarrow s$ **comment:** dimensione di S
repeat
  $mid \leftarrow (min + max)/2$
  $Z_m \leftarrow$ GETIEST(S, mid)
  if ISGOOD(r, n, Z_m)
    then $min \leftarrow mid + 1$
    else $max \leftarrow mid - 1$
until $min \geq max$
$iMaxGood \leftarrow max$ **comment:** indice di S del massimo valore buono
$Z_m \leftarrow$ GETIEST$(S, iMaxGood)$
if $(min == max)$ **and** (!ISGOOD(Z_m))
  then $iMaxGood \leftarrow min - 1$
return (GETIEST$(S, iMaxGood)$)

2.6 Calcolo della complessità dell'algoritmo C

Come per l'analisi dell'algoritmo B calcoliamo la complessità temporale per l'algoritmo 2.5.1 appena spiegato.
La complessità del generico passo della ricerca è la somma delle complessità

dei due punti descritti sopra che sono rispettivamente $O(n)$ e $O(\log(s))$. Finché $s \leq n$, $O(n) + O(\log(s))$ è $O(n)$ e ci sono $O(\log(s))$ passate quindi, come per l'algoritmo A, il calcolo dei valori buoni ha complessità temporale:

$$O(n \log(s))$$

Quindi nel complesso, anche per l'algoritmo C si ha:

$$T_C(n) = n \cdot 2^{O(\sqrt{\log(n)})}$$

2.6.1 Spazio occupato

Ad ogni livello di ricorsione vengono usate $O(\log^2(s))$ variabili, e ci sono $t = O(\sqrt{\log(n)})$ livelli di ricorsione, quindi in totale:

$$S_C(n) = O(\log^2(s) \cdot \sqrt{\log(n)}) = O(\log^{2.5}(n))$$

Capitolo 3

Diminuire lo spazio a spese del tempo

In questo capitolo vedremo che relazione intercorre tra la complessità temporale e quella spaziale dell'algoritmo A. In particolare si cercherà di capire la relazione con $g(n)$ e t.

Nella sezione 2.2 era stata mostrata la 2.7 che riportiamo ora:

$$g(n) = \frac{n}{c^{\sqrt{2\log_c n}-\frac{1}{2}}} \qquad t = \sqrt{2\log_c n} \tag{3.1}$$

La scelta $g(n)$ nel calcolo della complessità dell'algoritmo A è stata fatta per minimizzare il tempo di esecuzione. Però $g(n)$ influenza anche la profondità della ricorsione e quindi anche lo spazio totale usato.

Possiamo riscrivere $g(n)$ in un altra forma:

$$g(n) = n \cdot c^{\frac{1}{2}-\sqrt{2\log_c n}} = \sqrt{c} \cdot n \cdot c^{-\sqrt{2\log_c n}} = \sqrt{c} \cdot n \cdot c^{-\frac{2\log_c n}{\sqrt{2\log_c n}}} =$$

$$= \sqrt{c} \cdot n \cdot c^{-\frac{\log_c n}{\sqrt{2\log_c n}}} c^{-\frac{2\log_c n}{2\sqrt{2\log_c n}}} = \sqrt{c} \cdot n \cdot n^{-\frac{1}{\sqrt{2\log_c n}}} c^{-\frac{\sqrt{2\log_c n}}{2}} =$$

$$= \sqrt{c} \cdot n \cdot n^{\frac{1}{t}} \cdot c^{-\frac{t}{2}} = c^{\frac{1-t}{2}} n^{\frac{t-1}{t}}$$

Se ci soffermiamo su

$$c^{\frac{1-t}{2}}$$

possiamo vedere che per $t \geq 1$ si ha:

$$\frac{1-t}{2} \leq 0 \Longrightarrow c^{\frac{1-t}{2}} \leq 1$$

Di conseguenza vale:

$$g(n) \leq n^{1-\frac{1}{t}}$$

Secondo il lavoro di Karp, Saks e Wigderson [1] scegliamo:

$$g(n) = n^{1-\frac{1}{t}}, \qquad 2 \leq t \leq \sqrt{\log(n)}$$

Si possono ripetere gli stessi passaggi per il calcolo della complessità dell'algoritmo A fino a

$$h(n) \leq \frac{c \cdot n}{g(n)} + \frac{c^2 \cdot g(n)}{g(g(n))} + \frac{c^3 \cdot g(g(n))}{g(g(g(n)))} + \cdots + \frac{c^t \cdot g^{[t-1]}(n)}{g^{[t]}(n)} \tag{3.2}$$

In questo caso però i termini della sommatoria non sono tutti uguali tra loro, infatti si può facilmente vedere che

$$g^{[k]}(n) = n^{\left(1-\frac{1}{t}\right)^k}$$

e quindi

$$\frac{c^k \cdot g^{[k-1]}(n)}{g^{[k]}(n)} = c^k \left(\frac{n}{g(n)}\right)^{\left(1-\frac{1}{t}\right)^k} = c^k \left(n^{\frac{1}{t}}\right)^{\left(1-\frac{1}{t}\right)^k}$$

L'espressione 3.2 può essere quindi riscritta in questo modo:

$$h(n) \leq \sum_{k=1}^{t} \left[c^k \left(n^{\frac{1}{t}}\right)^{\left(1-\frac{1}{t}\right)^k} \right]$$

Visto che $t > 0$ si può dire che

$$\left(1 - \frac{1}{t}\right)^k \text{ è } O(1)$$

quindi

$$h(n) \leq \sum_{k=1}^{t} \left[c^k n^{\frac{1}{t}} \right] = n^{\frac{1}{t}} \sum_{k=1}^{t} c^k$$

Il termine maggiore della sommatoria è c^t e ci sono t termini, quindi:

$$h(n) \leq n^{\frac{1}{t}} t c^t$$

Si ha quindi:

$$f(n) = n \log^2(n) h(n) \leq n \log^2(n) n^{\frac{1}{t}} t c^t \leq n^{1+\frac{1}{t}} t c^t$$

La complessità temporale finale è quindi:

$$T_D(n) = O\left(n^{1+\frac{1}{t}} t c^t\right) \tag{3.3}$$

Per quanto riguarda la memoria occupata, la profondità massima della ricorsione è $O(t)$, quindi riusando i risultati ottenuti per gli algoritmi B e C si ottiene:

- $$S_D(n) = O(t) \quad \text{con un algoritmo randomizzato} \tag{3.4}$$

- $$S_D(n) = O(t \log^2(n)) \quad \text{con un algoritmo deterministico} \tag{3.5}$$

Consideriamo una costante ε e consideriamo fissiamo

$$t = \frac{1}{\varepsilon}$$

Dalla 3.3 si ha che:

$$T_D(n) = O\left(n^{1+\varepsilon} \cdot \frac{1}{\varepsilon} \cdot c^{\frac{1}{\varepsilon}}\right) = O\left(n^{1+\varepsilon}\right)$$

Mentre per la 3.4 si può avere:

$$S_D(n) = O\left(\frac{1}{\varepsilon}\right) = O(1)$$

Capitolo 4

Relazione con le procedure branch-and-bound

In questo capitolo verrà data la spiegazione ai risultati ottenuti nei capitoli precedenti in particolare quelli del Capitolo 3. Analizzando questi algoritmi si è visto come sia possibile diminuire notevolmente lo spazio richiesto per l'esecuzione $S(n)$ a fronte di un piccolo aumento del tempo di esecuzione $T(n)$. Verrà mostrato ora come questa cosa sia conseguenza del fatto che gli algoritmi adottano la procedura branch-and-bound.

4.1 Caratteristiche delle procedure branch-and-bound

Mostriamo ora le caratteristiche comuni a tutti gli algoritmi che utilizzano la procedura branch-and-bound.
Consideriamo un problema di trovare un elemento ottimo da un insieme di oggetti. Sia $g(x)$ una funzione dove $x \in \{0,1\}^m$, l'ottimizzazione va fatta in modo da minimizzare il valore di $g(x)$.
La variabile x può essere vista come un insieme di m variabili booleane

$$x = x_1 x_2 \cdots x_m, \qquad x_i \in \{0,1\}$$

L'approccio branch-and-bound si basa sul considerare certe restrizioni del problema originale, ottenute fissando alcune variabili x_i a 0 o 1.

Definizione 4.1. *Una **restrizione** è definita come un elemento $s \in \{0,1,*\}^m$. Le coordinate contenenti uno 0 o un 1 sono dette **fisse**, mentre quelle contenenti un $*$ sono dette **libere**.*

Ad ogni restrizione s è associato un insieme $Dom(s)$ contenente tutte le soluzioni x che soddisfano le regole definite da s, cioè:

$$Dom(s) = \{x \in \{0,1\}^m \,|\forall s_i \neq * \Rightarrow x_i = s_i\}$$

Se nella posizione i-esima $s_i = *$ allora x_i può assumere qualsiasi valore, altrimenti deve avere il valore di s_i.
Il problema associato alla restrizione s è di minimizzare $g(x)$ dove $x \in Dom(s)$, ed è un caso particolare del problema generale che è quello associato alla restrizione $*^m$ (non ci sono variabili booleane fissate).

Ogni algoritmo basato su branch-and-bound si differenza per la procedura usata per il calcolo dei limiti inferiori, è presente in ogni algoritmo ed è quella che lo caratterizza.
La procedura ha come input una restrizione s e ritorna un valore $A(s)$ tale che

$$A(s) \leq \min_{x \in Dom(s)} g(x)$$

molte volte la procedura risolve una versione semplificata del problema di partenza.

Un'assunzione molto ragionevole da fare è la seguente:

Teorema 4.1.1. *Se s' è una restrizione è ottenuta da una restrizione s fissando alcune delle sue coordinate libere, allora $A(s') \geq A(s)$.*

Dato che s' è più restrittiva di s identifica un sottoproblema quindi $A(s')$ è un limite globale per s' ma non è detto che lo sia anche per s.

Definizione 4.2. *Una **restrizione** s è **terminale** se il limite inferiore ad essa associato è uguale al valore minimo del problema iniziale ristretto ad s, cioè:*

$$A(s) = \min_{x \in Dom(s)} g(x)$$

In particolare per ogni restrizione s dove tutte le coordinate sono fisse, cioè $s = \{0,1\}^m$. L'insieme $Dom(s)$ è costituito della sola restrizione s, $Dom(s) = \{s\}$, e per quella s si ha $A(s) = g(s)$. Quindi s è una restrizione terminale.
Per quanto riguarda le altre restrizioni s ipotizziamo l'esistenza di una procedura efficiente per capire se s è terminale o meno.

L'elemento finale di una procedura branch-and-bound è la regola con la quale vengono richiamati i problemi ristretti (branching rule).
Questa regola associa ad ogni restrizione s non terminale una coppia di restrizioni figlie $R(s)$ e $L(s)$, ottenute da s fissando il valore di una particolare coordinata i che per s è libera, $s_i = *$.
Più precisamente:

$$R(s)_i = 0, \quad L(s)_i = 1 \qquad \text{mentre} \qquad \forall j \neq i \quad R(s)_j = L(s)_j = s_j$$

$R(s)$ e $L(s)$ considerano i due possibili valori della generica variabile s_i.

La procedura branch-and-bound determina $\min_x g(x)$ trovando una restrizione terminale s per la quale $A(s)$ è minima.
Un algoritmo che sfrutta questa procedura utilizza una lista L contenente tutte le restrizioni attive, cioè quelle per cui non è ancora stata calcolata $A(\cdot)$ e non sono ancora state espanse con i rispettivi figli. Inoltre per ogni restrizione $s \in L$ tiene traccia del limite $A(s)$ e l'indicazione se s è terminale o meno. All'inizio la lista L contiene solamente la restrizione che descrive il problema generale, $*^m$.
Ad ogni passo la procedura seleziona una certa restrizione attiva s dalla lista e la sostituisce con due restrizioni $L(s)$ e $R(s)$.
L'esecuzione termina quando il limite inferiore minore tra quelli delle restrizioni attive è uguale a quello associato ad una restrizione terminale s.
Si può facilmente far vedere che quel valore è quello minimo. Infatti sia L la lista di nodi attivi e $p \in L$ la restrizione descritta sopra, quindi:

$$A(p) = \min_{r \in L} A(r) = A(s) = \min_{x \in Dom(s)} g(x) \tag{4.1}$$

se si considera ogni altra restrizione $p' \neq p$ non ancora esaminata; questa può essere in L ma non con $A(p')$ minimo, oppure figlia di una delle restrizioni di L.
In ogni caso per il Teorema 4.1.1 e la relazione 4.1 sarà $A(p') \geq A(p)$ per quanto detto prima.

Questo metodo branch-and-bound appena descritto è detto best-first-search, perché espande la restrizione con il limite inferiore minore.
Nel caso in cui $A(\cdot)$ sia biunivoca, questo modo di procedere è ottimale.
Il metodo di scelta della restrizione da espandere secondo best-first-search, è quello da usare nel caso in cui ci sia abbastanza spazio a disposizione per l'algoritmo per memorizzare la lista di restrizioni attive, perché tipicamente la dimensione della lista è esponenziale in m, dove m è la taglia del problema.
Nei casi in cui best-first-search occupi più memoria di quella a disposizione è necessario cambiare il metodo con uno che sacrifica il tempo di esecuzione.

Per quanto riguarda gli algoritmi per $SELECT(n)$ descritti precedentemente, si vede subito che branch-and-bound consente di sviluppare algoritmi che utilizzano poco spazio ad un costo relativamente modesto di tempo di esecuzione.

4.2 Relazione tra i risultati ottenuti e branch-and-bound

Per evidenziare la relazione tra i risultati ottenuti e le procedure branch-and-bound basta pensare alla regola con la quale vengono richiamati i sot-

toproblemi (branching rule).
Consideriamo una procedura branch-and-bound la cui branching rule è quella descritta in precedenza, da essa possiamo determinare un albero binario U che soddisfa le seguenti caratteristiche:

- alla radice dell'albero corrisponde la restrizione $*^m$;
- ad ogni nodo interno è associata una restrizione non terminale s, e ai suoi due figli sono associate le restrizioni $L(s)$ e $R(s)$;
- ad ogni nodo foglia è associata una restrizione terminale.

Se consideriamo i valori $A(s)$ corrispondenti, dato che per il Teorema 4.1.1 $A(s) \leq A(R(s))$ e $A(s) \leq A(L(s))$. Si vede subito che se associamo ad ogni nodo a cui corrisponde la restrizione s il valore $A(s)$, l'albero appena costruito soddisfa la proprietà di heap.
In questo modo abbiamo così costruito un albero che soddisfa quasi le stesse ipotesi dell'albero T considerato fino ad ora. Le uniche differenze tra l'albero U appena costruito e l'albero T sono:

- l'albero è finito
- i valori non sono tutti distinti

Però queste differenze possono essere facilmente risolte.

A questo punto possiamo applicare l'algoritmo $SELECT(i)$ all'albero U per cercare $\min_x g(x)$ partendo con $i = 1$ e raddoppiando i ad ogni iterazione fermandosi quando il nodo terminale del valore più piccolo è trovato.

Dato che ad ogni iterazione i viene raddoppiata, se supponiamo che il valore cercato sia l'n-esimo elemento più piccolo. Dal punto di vista della complessità temporale possiamo approssimare la somma delle chiamate a $SELECT(i)$ con l'ultima che è quella con input n. Anche per quanto riguarda la complessità spaziale si avranno le stesse prestazioni di $SELECT(n)$. Quindi il problema branch-and-bound può essere risolto con le stesse prestazioni con cui è risolto il problema $SELECT(n)$.

Conclusioni

In questa tesi sono stati presentati tre algoritmi per trovare l'n-esimo elemento più piccolo dalla struttura di dati inizialmente descritta. Dopo la presentazione e l'analisi dell'algoritmo A si è visto che il punto dove l'algoritmo consuma più memoria è quando deve trovare i valori buoni dall'insieme S (algoritmo 2.1.4). Con gli algoritmi B e C si è cercato di risolvere il problema proponendo prima una variante randomizzata (algoritmo 2.3.1) e poi una non (algoritmo 2.5.1), mantenendo la stessa complessità temporale dell'algoritmo A. Nel capitolo 3 si è visto come le complessità temporale e spaziale siano legate tra loro. In particolare come sia possibile ottenere ottime prestazioni dal punto dello spazio occupato partendo dall'algoritmo B, a fronte di un piccolo costo aggiuntino nella complessità temporale. Infine nel capitolo 4 si è visto come questi risultati siano dovuti al fatto che gli algoritmi utilizzano la procedura branch-and-bound, infatti esiste una corrispondenza tra la classe di problemi che risolve questa procedura e il problema affrontato.

Bibliografia

[1] R. M. Karp, M. Saks, A. Wigderson, *On a Search Problem Related to Branch-and Bound Procedures*

[2] J. I. Munro e M. S. Paterson, *Selection and Sortin with Limited Storage*, Proceedings of the 19th FOCS Conference, pp. 253-258, 1978