How to bound in Branch and Bound

"Branch and Bound", in particular "Bound", refers to a technique to remove some of the inefficiency that almost inevitably attends most "Backtracking Algorithms". Before dealing with that technique, let us first spend a few words on backtracking.

Backtracking Algorithms usually enter the picture if a huge class of things has to be explored in order to identify a subclass with certain properties. Typical examples of such huge classes are all subsets of a given set, all permutations of a number of objects, all partitions of a natural number, all walks through a graph, all chess games, etc.. If certain members of such a class have to be identified, it is a safe procedure to generate (to "visit") all of its members and check, for each of them individually, whether it has the desired properties. The procedure is safe but its time complexity may grow gigantically, and in many cases there is no cure.

By way of example let us investigate the set V of all bitstrings of length N, and let us assume that each bitstring x — of whatever length — has a cost $c.x$ associated with it. The problem we consider is to compute the minimal cost of a bitstring in V, i.e. the value of

$$\langle \downarrow x : x \in V : c.x \rangle .$$

A typical way of tackling this problem is to introduce a function f on the set pref$V$

of all bitstrings of length $\leq N$, as follows:

$$f.u \;=\; \langle \downarrow x{:}\; u \mathbin{+\!\!+} x \in V{:}\; c.(u \mathbin{+\!\!+} x) \rangle \qquad , \quad u \in \mathrm{pref}V .$$

(Symbol $\mathbin{+\!\!+}$ denotes catenation.)
Then the desired answer is the value of

$$f.[\,]\quad .$$

(Symbol $[\,]$ denotes the empty string.)

The recursive scheme governing the computation of f is — proof omitted —

- for $\#u = N$, $\quad f.u = c.u$
- for $\#u < N$, $\quad f.u = f.(u \mathbin{+\!\!+} [0]) \downarrow f.(u \mathbin{+\!\!+} [1])$

($\#u$ stands for the length of u, and $[b]$ is the singleton list containing element b.)

For the actual computation of f we introduce a function $F(u{:}\,list){:}\ int$ with specification

precondition: $\quad u \in \mathrm{pref}V$
postcondition: $\quad F = f.u \quad .$

The code for a program may now read

```
func  F(u: list) : int ;
   {pre:  u ∈ prefV}
   {post:  F = f.u}
   if  #u = N   →   F := c.u
   ▯  #u < N   →   F :=  F(u ++ [0]) ↓ F(u ++ [1])
   fi
cnuf    ;    print( F([]) )  .
```

As can be seen from this program text, the recursion does not come to an end before $\#u = N$, i.e. before a bit string of full length is generated. In fact, the program will generate all of the 2^N elements of $\check{V}$ exactly once, and compute the cost for each of them. And if function c is of a whimsical nature, there is nothing we can do to improve on the program's efficiency.

* * *

However, in many practical circumstances cost function c is not whimsical at all. A very frequent situation is that we have

$$(0) \qquad c.u \leq c.(u \# x) ,$$

a case that we shall now further investigate. The traditional operational jargon exploits (0) by the remark that "there is no need to extend string u if c.u is at least as big as any bitstring of length N that the program has visited or seen thus far". We will first illustrate how to exploit (0) in a nonoperational manner and then relate it to the above operational interpretation.

We generalize function f into a function g defined by

$$g.r.u \;=\; r \downarrow f.u \qquad\qquad (r \in Int,\; u \in pref V) .$$

Here, parameter r is called "the bound". In terms of g, the desired answer — which was f.[] — is

$g.r.[\,]$ for any r: $f.[\,] \leq r$.

The great advantage of g over f is that now we can sometimes compute the value of $g.r.u$ without resorting to a "laborious" calculation of $f.u$, viz. we can compute $g.r.u$ at a bargain whenever $r \downarrow f.u = r$, in which case $g.r.u = r$. Let us investigate this condition:

$r \downarrow f.u = r$

$\equiv$ { definition of $\downarrow$ }

$r \leq f.u$

$\equiv$ { definition of f }

$r \leq \langle \downarrow x : u +\!\!+ x \in \bar{V} : c.(u +\!\!+ x) \rangle$

$\equiv$ { definition of $\downarrow$ }

$\langle \forall x : u +\!\!+ x \in \bar{V} : r \leq c.(u +\!\!+ x) \rangle$

$\Leftarrow$ { (0) }

$\langle \forall x : u +\!\!+ x \in \bar{V} : r \leq c.u \rangle$

$\Leftarrow$ { predicate calculus }

$r \leq c.u$

As a result we find

- for $r \leq c.u$, $g.r.u = r$.

For the remaining case, viz. $c.u < r$ we resort to the recursive scheme for f:

- for $c.u < r \wedge \#u = N$,

$g.r.u$

$=$ { definition of g }

$r \downarrow f.u$

$=$ { $\#u = N$ }{ scheme for f }

$r \downarrow c.u$

$=$ $\{c.u < r\}$

$c.u$

- for $c.u < r$ and $\#u < N$,

$g.r.u$

$=$ $\{$ definition of g $\}$

$r \downarrow f.u$

$=$ $\{\#u < N\}\{$ scheme for $f\}$

$r \downarrow (\, f.(u \mathbin{+\!\!+} [0]) \downarrow f.(u \mathbin{+\!\!+} [1])\,)$

$=$ $\{ \downarrow$ is associative $\}$

$(r \downarrow f.(u \mathbin{+\!\!+} [0])) \downarrow f(u \mathbin{+\!\!+} [1])$

$=$ $\{$ definition of g $\}$

$g.r.(u \mathbin{+\!\!+} [0]) \downarrow f.(u \mathbin{+\!\!+} [1])$

$=$ $\{$ definition of g $\}$

$g.\,(g.r.(u \mathbin{+\!\!+} [0])).\,(u \mathbin{+\!\!+} [1])$

or — equivalently —
with $s = g.r.(u \mathbin{+\!\!+} [0])$
and $t = g.s.(u \mathbin{+\!\!+} [1])$

$g.r.u = t$.

In summary, the recursive scheme governing the computation of g is

- for $r \le c.u$, $\quad g.r.u = r$
- for $c.u < r \wedge \#u = N$, $\quad g.r.u = c.u$
- for $c.u < r \wedge \#u < N$, $\quad g.r.u = t$
  where $\quad t = g.s.(u \mathbin{+\!\!+} [1])$
  and $\quad s = g.r.(u \mathbin{+\!\!+} [0])$.

The corresponding program text may read

```
func G (r: int, u: string): int;
  {pre:  $u \in$ pref V }
  {post:  G = g.r.u }
  if r ≤ c.u → G := r
  ▯ c.u < r ∧ #u = N → G := c.u
  ▯ c.u < r ∧ #u < N →
        |[ s, t : int;
           s := G(r, u ++ [0])
         ; t := G(s, u ++ [1])
         ; G := t
        ]|
  fi
cnuf
; "for some r such that f.[] ≤ r , print ( G(r, [])) " .
```

* * *

Next, we extend the program so as to produce a witness of a cheapest bitstring in V as well. Not only is such an extension a quite practical one, but also will it create a better opportunity to discuss the aforementioned operationalist's view of "bounding".

In order to compute a witness we introduce a variable z of type string, and strengthen the postcondition of G with the conjunct

$$z \in V \;\wedge\; c.z = G \; .$$

Because G(r, []) as it occurs in the main call, is the desired minimal value, the string z satisfying this additional postcondition is an appropriate witness indeed.

Now we investigate under what conditions the three alternatives of function G establish this new postcondition

- $\underline{r \leq c.u \rightarrow G := r}$:

$(z \in V \wedge c.z = G)(G := r)$

$\equiv$ {substitution}

$z \in V \wedge c.z = r$,

and because this does not follow from the guard $r \leq c.u$ nor from G's precondition $u \in \text{pref}\,V$, we decide that

(*) $z \in V \wedge c.z = r$ be a precondition of G

as well.

- $\underline{c.u < r \wedge \#u = N \rightarrow G := c.u}$:

$(z \in V \wedge c.z = G)(G := c.u)$

$\equiv$ {substitution}

$z \in V \wedge c.z = c.u$

$\Leftarrow$ {Leibniz}

$z \in V \wedge z = u$

Because $u \in \text{pref}\,V$ — from pre of G — and because $\#u = N$ — from the guard — we conclude that $u \in V$ so that the above calculated precondition $z \in V \wedge z = u$ of $G := c.u$ is readily established by prefixing $G := c.u$ with statement $z := u$.

- $\underline{c.u < r \wedge \#u < N \rightarrow \text{block}}$:

For the block in the third alternative we need no further adjustments of G, as may follow from the annotation given below. Please observe that the preassertion of each call of G exactly matches the required additional precondition (*) of G.

```
    { $z \in V \wedge c.z = r$ , from precondition (*) }
     $s := G.r.(u ++ [0])$
  ; { $z \in V \wedge c.z = s$ , from added postcondition
                             of G }
     $t := G.s.(u ++ [1])$
  ; { $z \in V \wedge c.z = t$ }
     $G := t$
    { $z \in V \wedge c.z = G$, as required } .
```

Thus, our final program which computes a witness as well, has become

```
func G(r: int, u: string) : int;
  {pre: $u \in \mathrm{pref}V \wedge z \in V \wedge c.z = r$ }
  {post: $G = g.r.u \wedge z \in V \wedge c.z = G$ }
   if $r \le c.u \rightarrow G := r$
   [] $c.u < r \wedge \#u = N \rightarrow z := u ;\ G := c.u$
   [] $c.u < r \wedge \#u < N \rightarrow$
               |[ s, t: int ;
                  $s := G(r, u ++ [0])$
               ; $t := G(s, u ++ [1])$
               ; $G := t$
               ]|
   fi
cnuf
; "for some r, z such that $z \in V$, $c.z = r$, and $f[\,] \le r$,
  print (G (r, [])) ; print (z) " .
```

* * *

Here we have reached a point where we can can comply with the operationalist's remark that indeed "string u is not extended if $c.u$ is at least as big as any bitstring of length N that the program has seen thus far". From the first alternative we conclude that the recursion comes to an end whenever $r \leq c.u$, and from the precondition $z \in \check{V} \wedge c.z = r$ we see that the program has "seen" a bitstring z of length N, with $c.z \leq c.u$. And from the postcondition of G we see that z is the best, the cheapest, bitstring that the program has "seen".

We very much dislike such an operational parlance, especially if we confront it with what is the mathematical crux of bounding, viz. the transition from function

$$f.u$$

to function

$$r \downarrow f.u \ .$$

Isn't this beautifully simple? We learned this from Anne Kaldewaij, who learned it from Rob Hoogerwoord. We are grateful to them, because this is how one bounds in "Branch and Bound".

WHJ Feijen
12 October 1995
Sterksel