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, we 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 tic-tac-toe games, etc. .. If certain members of such a class have to be identified, a safe procedure is to generate *all* members and to check, for each of them individually, whether they have the desired properties. The procedure is safe but its time complexity may grow gigantically, and in many cases there is no cure.

* * *

As a typical 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

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

The backtracking strategy boils down to the introduction of a function f on the set $prefV$ of all bitstrings of length at most N, defined by

$$f.u = \langle \downarrow x : u +\!\!+ x \in V : c.(u +\!\!+ x) \rangle \qquad , \quad u \in prefV$$

(Symbol $+\!\!+$ denotes catenation.)
Then the desired answer is the value of $f.[\,]$. (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 $\#u < N$, $\quad f.u = f.(u +\!\!+ [0]) \downarrow f.(u +\!\!+ [1])$.

($\#u$ 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 $F(u : \text{list}) : \mathsf{int}$ with specification

$$\text{precondition :} \quad u \in prefV$$

$$\text{postcondition :} \quad F = f.u \quad .$$

The code for a program to print target value $f.[\,]$ is now straightforward. It reads

$$\begin{array}{l} \mathsf{func}\ \ F(u : \text{list}) : \mathsf{int}\ ; \\ \quad \{\ \text{pre :} \quad u \in prefV\ \} \\ \quad \{\ \text{post :} \quad F = f.u\ \} \\ \quad \mathsf{if}\ \ \#u = N \rightarrow\ F\ :=\ c.u \\ \quad []\ \ \#u < N \rightarrow\ F\ :=\ F(u +\!\!+ [0]) \downarrow F(u +\!\!+ [1]) \\ \quad \mathsf{fi} \\ \mathsf{cnuf} \\ ;\ \mathsf{print}\ (F([\,])) \quad . \end{array}$$

As can be seen from the function definition, the recursion only comes to an end when $\#u = N$, i.e. when a bitstring of full length has been generated. In fact, the above program will generate all 2^N elements of V exactly once, and compute the cost of 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.

* *
*

In many practical circumstances, however, 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 jargon exploits (0) by a remark like : "there is no need to extend string u if $c.u$ is at least as big as the c-value of any bitstring of length N that the program has computed so far". We will illustrate how to exploit (0) in a non-operational manner.

We generalize function f into a function g defined by

$$g.r.u \ = \ r \downarrow f.u \qquad (\ r \in \text{Int}\ ,\ \ u \in prefV\ )$$

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

$$g.r.[\,] \qquad \text{for any } r \text{ satisfying } f.[\,] \leq r$$

The great advantage of g over f is that now the value of $g.r.u$ can sometimes be computed without recourse to a "laborious" calculation of $f.u$, viz. when $r \downarrow f.u = r$, in which case $g.r.u = r$. Let us investigate this condition :

$$\begin{array}{ll} & r \downarrow f.u = r \\ \equiv & \qquad \{ \text{ property of } f \ \} \\ & r \leq f.u \\ \equiv & \qquad \{ \text{ definition of } f \ \} \\ & r \leq \langle \downarrow x : u +\!\!+ x \in V : c.(u +\!\!+ x) \rangle \\ \equiv & \qquad \{ \text{ property of } f \ \} \\ & \langle \forall x : u +\!\!+ x \in V : r \leq c.(u +\!\!+ x) \rangle \\ \Leftarrow & \qquad \{ \ (0) \ \} \\ & \langle \forall x : u +\!\!+ x \in V : r \leq c.u \rangle \\ \Leftarrow & \qquad \{ \text{ predicate calculus } \} \\ & r \leq c.u \ . \end{array}$$

As a result we find

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

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

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

$$\begin{array}{ll} & g.r.u \\ = & \qquad \{ \text{ definition of } g \ \} \\ & r \downarrow f.u \\ = & \qquad \{ \ \#u = N \ \} \ \{ \text{ scheme for } f \ \} \\ & r \downarrow c.u \\ = & \qquad \{ \ c.u < r \ \} \\ & c.u \ . \end{array}$$

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

$$g.r.u$$

$$
\begin{array}{ll}
= & \{ \text{ definition of } g \ \} \\
& r \downarrow f.u \\
= & \{ \ \#u < N \ \} \ \{ \text{ scheme for } \ f \ \} \\
& r \downarrow (\ f.(u +\!\!+ [0]) \ \downarrow f.(u +\!\!+ [1])) \\
= & \{ \ \downarrow \text{ is associative } \} \\
& (r \downarrow f.(u +\!\!+ [0])) \downarrow f.(u +\!\!+ [1]) \\
= & \{ \text{ definition of } g \ \} \\
& g \,.\, r \,.\, (u +\!\!+ [0]) \downarrow f.(u +\!\!+ [1]) \\
= & \{ \text{ definition of } g \ \} \\
& g \ . \ (g \,.\, r \,.\, (u +\!\!+ [0])) \,.\, (u +\!\!+ [1])
\end{array}
$$

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

$$g.r.u \ = \ t \quad .$$

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

- for $\quad r \leq c.u \ , \qquad\qquad g.r.u \ = \ r$
- for $\quad c.u < r \ \wedge \ \#u = N \ , \qquad g.r.u \ = \ c.u$
- for $\quad c.u < r \ \wedge \ \#u < N \ , \qquad g.r.u \ = \ t$

where $\quad t \ = \ g \,.\, s \,.\, (u +\!\!+ [1])$
and $\quad s \ = \ g \,.\, r \,.\, (u +\!\!+ [0]) \quad .$

The corresponding program text that prints the target value reads

```
func G(r:int, u:string) : int ;
    { pre :   u ∈ prefV }
    { 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 $f.[\,] \leq r$, print(G(r ,[ ])) " .

Remark The difference between F and G is that the recursion in G also comes to an end when $r \leq c.u$, i.e. when $c.u$ has become too large or r has become small enough. Therefore, it is beneficial —even quite beneficial— for the efficiency of the program, if the initial value for r in the main call can be chosen as small as possible. And in many applications one can indeed without too much effort find an initial estimate for r that is quite close to target value $f.[\,]$.
End Remark .

* * *

In practice, one usually wishes to compute a witness for the cheapest solution as well. We shall now show how to do this for our program above.

In order to compute a cheapest bitstring in V , we introduce a global 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 how the three alternatives of function G can establish this new postcondition

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

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

  $\equiv$ { 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 $u \in \mathit{pref}V$, we decide that

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

• $c.u<r \ \wedge \ \#u=N \rightarrow G := c.u$:

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

$\equiv$ { substitution }

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

$\Leftarrow$ { Leibniz }

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

Because $u \in prefV$ —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 $z := u$.

• $c.u<r \ \wedge \ \#u<N \rightarrow$ 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 .

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

End • .

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

func G(r : int , u : string) : int ;
{ pre : $u \in prefV \wedge z \in V \ \wedge \ c.z=r$ }
{ post : $G = g.r.u \ \wedge \ z \in V \ \wedge \ c.z=G$ }
if $r \leq c.u \rightarrow G := r$
[] $c.u<r \ \wedge \ \#u=N \rightarrow z := u \ ; G := c.u$

```
[]  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 and z such that z∈V , c.z=r , and f.[ ]≤r ,
  print(G(r,[ ])) ; print(z) "  .
```

* * *

Branch and Bound, and Backtracking, belong to what the field has called "Combinatorial Algorithms. Those algorithms are usually presented in a very operational manner. Backtracking is explained as tree traversal and bounding as some sort of pruning. And the algorithms are presented in terms of pictures rather than with formulae. We do not really like such explanations, in particular if one realizes that the mathematical crux of bounding, viz. the transition from function

$f.u$

to $r \downarrow f.u$

is so beautifully simple.

We learned this technique from our colleagues Anne Kaldewaij and Rob Hoogerwoord. Kaldewaij in his book [?] has considerably raised the standards for dealing with Combinatorial Algorithms, and it should be pointed out —for methodological reasons— that this rise of standard has become possible by the emergence of nice calculational styles of functional programming as, for instance, the one laid down in Hoogerwoord's [?].