A NOTE ON THE STANDARD STRATEGY FOR DEVELOPING LOOP INVARIANTS AND LOOPS

David Gries

TR 82-531
October 1982

Department of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853

A Note on the Standard Strategy for Developing Loop Invariants and Loops

David Gries
Computer Science
Cornell University

October 1982

Abstract

The by-now-standard strategy for developing a loop invariant and loop was developed in [1] and explained [2]. Nevertheless, its use still poses problems for some. The purpose of this note is to provide further explanation. Two problems are solved that, without this further explanation, seem difficult.

Introduction

The standard strategy for developing a loop invariant and loop can indeed be powerful in the hands of one who understands it fully, but others with less experience and skill find it difficult to use in the seemingly complex situations where it is really needed. This is due, at least partially, to the way the strategy is presented. The purpose of this note is to present the strategy in a slightly different and less formal manner.

We begin by discussing the proof of correctness of a loop. We then develop a rather trivial algorithm in order to describe the strategy in its simplest form. Finally, we proceed to illustrate a more difficult part of the strategy using two examples.

Proving a Loop Correct

E.W. Dijkstra's guarded command loop with precondition Q and postcondition R, $\{Q\}$ **do** $B \rightarrow S$ **od** $\{R\}$, is proved correct using an ***invariant relation*** P and a ***bound function*** t, which gives an upper bound on the number of iterations still to perform (see e.g. [1], [2]):

(1) P is true initially: $Q \Rightarrow P$;

(2) P is a loop invariant: $P \wedge B \Rightarrow wp(S, P)$;

(3) Upon termination R is true: $P \wedge \neg B \Rightarrow R$;

(4) If another iteration can be performed, then $t > 0$: $P \wedge B \Rightarrow t > 0$;

(5) t is decreased by at least one with each iteration: using a fresh variable $t1$, we have
$\{P \wedge B\}\ t1 := t;\ S\ \{t < t1\}$

Developing a Loop Using the Standard Strategy

To illustrate the standard strategy for developing a loop, let us develop an algorithm for storing in variable p the sum of the elements of array $b[0{:}n{-}1]$, where $0 \leq n$. The desired postcondition of the algorithm is

$$R: p = (\sum k: 0 \leq k < n: b[k])$$

Recognizing that iteration (or recursion) is needed, we try to develop the loop invariant P first —by weakening R to include a state that can be easily established. In this case, weakening R can be done by replacing constant n of R by a fresh variable i and placing suitable bounds on i:

$$P: 0 \leq i \leq n \ \wedge \ p = (\sum k: 0 \leq k < i: b[k])$$

P is easily established using $i, p := 0, 0$. Further, $P \wedge i = n \Rightarrow R$, so we can take $i \neq n$ as the guard of the loop.

The bound function t, an upper bound on the number of iterations still to perform, is $n - i$. To reduce the bound function at each iteration choose the command $i := i + 1$, yielding, thus far,

$i, p := 0, 0;$
do $i \neq n \rightarrow i := i + 1$ **od**

The last step is to ensure that P is indeed a loop invariant. To do this, we first calculate

$$Q1: wp(i := i + 1, P)$$
$$= 0 \leq i + 1 \leq n \ \wedge \ p = (\sum k: 0 \leq k < i + 1: b[k])$$

Now, for P to be a loop invariant we must have $P \wedge B \Rightarrow Q1$. But this is not the case, so the loop body must be modified. Comparing $Q1$ and P, we see that adding $b[i]$ to p within the loop body will allow P to remain invariantly true, and we end up with the algorithm

$i, p := 0, 0;$
do $i \neq n \rightarrow p, i := p + b[i], i + 1$ **od**

The steps in this development are fairly standard, and we discuss only those pertinent to this paper, those dealing with the development of invariant P. The first step was the following:

(6) • Find the (first approximation to the) invariant by weakening the postcondition R.

General techniques for weakening a predicate are discussed in [1] and [2].

The second point is that, once a way is found to reduce the bound function —in our case $i := i + 1$— the invariance of P must be shown. That is, if S is the loop body determined so far, it must be shown that $P \wedge B \Rightarrow wp(S, P)$ holds. If not, S must be modified so that it does, and to determine the modification one investigates the difference between $P \wedge B$ and $wp(S, P)$.

This part of the strategy can be described as follows:

(7) • Determine the conditions under which decreasing the bound function will falsify the invariant, and modify the loop body S accordingly.

Steps (6) and (7) are often easy to apply. However, in some cases applying (7) may seem difficult or may lead to inefficient and clumsy algorithms. At this point, more direction is needed. We summarize the strategy to be followed as follows:

(8) • Determine what information is needed in order to make application of (7) more effective, and represent that information in fresh variables, thus modifying P.

Conscious application of (8) can indeed be useful. To illustrate this, let us turn to two "difficult" problems. For purposes of comparison, the reader may want to try developing the algorithms using her own methods before reading the idealized developments.

A First Example: The Minimum-Sum Section

Given is an array $b[0{:}n-1]$, where $n \geq 1$. A *minimum-sum section* of b is a non-empty sequence of adjacent elements whose sum is a minimum. For example, the minimum-sum section of array $b[0{:}4] = (5,-3,2,-4,1)$ is $b[1{:}3] = (-3,2,-4)$; its sum is -5. The minimum-sum section of array $b = (5,-3,5,-4,1)$ is $b[3{:}3] = (-4)$; its sum is -4. The two minimum-sum sections of $b = (5,2,5,4,2)$ $b[1{:}1]$ and $b[4{:}4]$.

Desired is a program that stores in a variable s the sum of a minimum-sum section of b. Let $S_{i,j}$ denote the sum of section $b[i{:}j]$:

$$S_{i,j} = (\sum k: i \leq k < j+1: b[k]).$$

Then the postcondition R is

$$R: s = (\mathbf{min}\, i,j: 0 \leq i \leq j < n: S_{i,j})$$

It is obvious that each element of b must be referenced to determine a minimum-sum section and its sum, so our first thought is to reference them in increasing order of subscript value. Using strategy (6) as our guide, we find a first approximation to the invariant P by replacing constant n by a fresh variable k (and placing bounds on k):

$$P: 1 \leq k \leq n \ \wedge\ s = (\mathbf{min}\, i,j: 0 \leq i \leq j < k: S_{i,j})$$

The initialization is obvious. Also, the bound function is obviously $n-k$. And we write

$k, s := 1, b[0];$
do $k \neq n \rightarrow k := k+1$ **od**

Next, applying (7), we determine the conditions under which P may be falsified by execution of the loop body. We calculate $Q1 = wp(k := k+1, P)$:

$$Q1: 1 \leq k+1 \leq n \ \wedge\ s = (\mathbf{min}\, i,j: 0 \leq i \leq j < k+1: S_{i,j})$$

The first conjunct of $Q1$ is implied by $P \wedge k \neq n$, but the second is not. Comparing the second conjuncts of P and $Q1$, we see that $\neg(P \Rightarrow Q1)$ holds precisely when there is a section of b ending in $b[k]$ whose sum is less than s.

At this point, it looks like determining whether a section $b[i{:}k]$ with $S_{i,k} < s$ exists may take time at least proportional to k. What can we do to make it more efficient? Turning to strategy (8), we ask what additional information can help. Suppose we know $(\mathbf{min}\, i: 0 \leq i < k: S_{i,k-1})$. Then $(\mathbf{min}\, i: 0 \leq i < k+1: S_{i,k})$ can be calculated in constant time. Let us introduce a variable c and change P accordingly:

$$P: 1 \leq k \leq n \ \wedge$$
$$\quad s = (\mathbf{min}\, i,j: 0 \leq i \leq j < k: S_{i,j}) \ \wedge$$
$$\quad c = (\mathbf{min}\, i: 0 \leq i < k: S_{i,k-1})$$

Note that P implies that $s \leq c$. It is then a simple task to modify and complete the algorithm:

$k, s, c := 1, b[0], b[0];$
do $k \neq n \rightarrow c := min(c + b[k], b[k]);$
$\qquad\qquad s := min(s, c);$
$\qquad\qquad k := k+1$
od

This algorithm has the following history. Jon Bentley of Carnegie-Mellon saw a statistician using an $O(n^3)$ algorithm that determined the bounds of a minimum-sum section as well as its sum several years ago. An $O(n^2)$ algorithm was found to replace it, then an $O(n \times log(n))$ algorithm, and there the matter rested for several years. Someone then found a linear algorithm, and Bentley, while discussing programming methodology at Cornell, challenged us with this problem. The linear algorithm was developed basically as shown above —the usual starts and restarts to get familiar with the problem are not shown, and a less formal notation was used— and, in fact, it was the only algorithm considered.

The determination of a minimum-sum section has been omitted simply to omit detail not germane to the topic at hand. The reader may find it useful to solve the same problem with a slight change: empty sections should also be considered. Thus, if the array contains only positive values, the minimum-sum section is the empty section and its sum is 0.

A Second Example: Finding the Largest Square

We proceed with this problem in the same as with the last. We will, however, use a two-dimensional "picture" notation for part of the invariant, which may make it easier to understand the development. This notation can be translated easily into the predicate calculus.

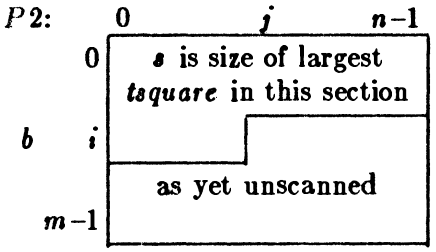
Given is a rectangular, Boolean array $b[0{:}m-1, 0{:}n-1]$, where $m, n \geq 0$. Desired is the size —i.e. length of a side— of a largest square of adjacent, true elements. In what follows, we abbreviate "square of adjacent, true elements" by "*tsquare*". Thus, we write the postcondition R as

R: s is the size of the largest *tsquare* in $b[0{:}m-1, n-1]$.

A moment's thought indicates that, in the worst case, it will be necessary to reference each element of the array, so we consider ways of referencing each element. Row-major order is simple, so we weaken R to the invariant $P = P1 \wedge P2$, where

$P1$: $0 \leq i \leq m \wedge 0 \leq j < n$ and

$P2$:



The bound function t is the number of elements in the lower, unscanned, section. The obvious way to proceed towards termination is to scan element $b[i,j]$, changing j (and perhaps i) accordingly. According to strategy (7), we now have the problem of maintaining P when $b[i,j]$ is scanned. We first determine the conditions under which P is falsified:

P is falsified if there is a *tsquare* with lower right corner $b[i,j]$ whose side is longer than s.

Calculating the size of the largest *tsquare* with lower right corner $b[i,j]$ seems to be a fairly complicated task, so we turn to strategy (8) for some inspiration. What useful information can we save in fresh variables? There are several alternatives. Among them are

1. Maintain the sizes of the largest *tsquares* with lower right corners $b[i,j-1]$ and $b[i-1,j]$ —from them it is easy to calculate the size of the largest *tsquare* with lower right corner $b[i,j]$.

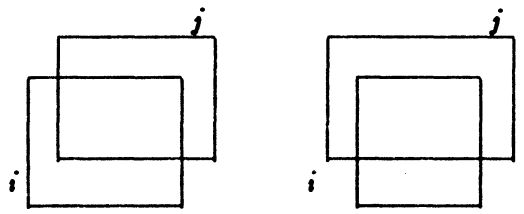
2. Maintain the size of the largest *tsquare* with lower right hand corner $b[i-1,j-1]$, the length of the column of true values ending in $b[i-1,j]$, and the length of the row of true values ending in $b[i,j-1]$.

In any case, since the problem will occur with each new column, the information will be needed for each column. Let us implement the first alternative. Add a conjunct $P3$ that describes a one-dimensional array $c[-1{:}n-1]$ to P, where element $c[-1]$ is introduced to remove a case analysis:

P: $P1 \wedge P2 \wedge P3$, where

$P3$: $c[-1] = 0 \wedge$
$(\forall k: 0 \leq k < j: c[k] =$ size of largest *tsquare* with lower right corner $b[i,k]) \wedge$
$(\forall k: j \leq k < n: c[k] =$ size of largest *tsquare* with lower right corner $b[i-1,k])$

$P3$ must be left invariantly true as $b[i,j]$ is scanned, and thus we must determine how to change $c[j]$. Suppose $b[i,j]$ is true. Consider the three cases $c[j-1] < c[j]$, $c[j-1] = c[j]$ and $c[j-1] > c[j]$, the first two of which are drawn below. (The last case is similar to the first.) Letting $p = min(c[j-1], c[j])$, we see in each case that the largest *tsquare* with lower right corner $b[i,j]$ is either $p+1$ or p, depending on whether $b[i-p, j-p]$ is true.



$c[j] = c[j-1]$ $\qquad$ $c[j] > c[j-1]$

With this information, the algorithm is written as follows:

```
i, j, s := 0, 0, 0;
(∀k: -1 ≤ k < n: c[k] := 0);
do i ≠ n →
   if ¬b[i,j] → c[j] := 0
    [] b[i,j] → var p: integer;
                p := min(c[j-1], c[j]);
                if ¬b[i-p, j-p] → c[j] := p
                 [] b[i-p, j-p] → c[j] := p+1
                fi
   fi;
   {c[j] is size of largest square with lower right corner b[i,j]}
   s := max(s, c[j]);
   if j = n-1 → i, j := i+1, 0
    [] j < n-1 → j := j+1
   fi
od
```

The execution time of this algorithm is $O(m*n)$. Again, this was essentially the only algorithm thought of during the development, although, there several ways of maintaining the necessary information were thought of. I learned of the problem from Ed Cohen of Prime, but discovered that it, and its appearance in [4], can be traced to Wim Feijen.

Discussion

This strategy for developing loop invariants and loops has been used often in the past; in fact, at times it seems the only reasonable way to proceed. A good example of its use is in the development of an algorithm for the longest upsequence problem, due originally to Dijkstra, which can be found [2]. There, the strategy had to be applied a number of times until it became obvious how the invariant had to be generalized. Reference [3] discusses the same strategy but in a more limited context.

However, many experienced and unexperienced programmers have been unable to solve effectively either of the two problems shown above, partially because they were unable to apply these problem-solving techniques in a conscious manner. Hence, this paper.

Acknowledgements

I am grateful to Fred Schneider for criticizing a draft of this paper and to J. Misra for pointing out reference [3] —a paper I had refereed but forgotten! Thanks also to members of IFIP WG2.3 for comments on a presentation of this material at our Mohonk meeting, to members of class CS600 at Cornell, many of whom were able to solve the two problems in a manner similar to the one shown after having been taught the strategy, and to the Tuesday Afternoon Club for their comments on the problems and the standard technique.

References

[1] Dijkstra, E.W. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, 1976.
[2] Gries, D. *The Science of Programming.* Springer Verlag, New York, 1981.
[3] Misra, J. A technique of algorithm construction on sequences. *IEEE Trans. on Soft. Eng. Se-4*, 1 (January 1978), 65-69.
[4] Reynolds, John. *The Craft of Programming.* Prentice Hall International, 1981.