Is Disciplined Programming Transferable, and is it Insightful?

By Charles Lakos and Arthur Sale*

This paper examines the disciplined programming methodology of E.W. Dijkstra which advocates the development of correctness proofs simultaneously with writing a program, (if not before) in the context of two problems which faced the authors recently. The paper applies the thought processes advocated by Dijkstra to these problems and indicates the insights that the authors gained from this. In both cases algorithms new to the authors were derived, and the properties of these are also examined. The paper should be interesting to those concerned with increasing the effectiveness of programming, for it demonstrates that the techniques advocated by Dijkstra are indeed transferable to other programmers, and that this transfer yields better insight into the activity we call programming.

KEYWORDS AND PHRASES: Disciplined programming, programming methodology, tree traversal, text searching, algorithm analysis, correctness proofs.
CR CATEGORIES: 3.74, 4.0, 5.24.

1. INTRODUCTION

The purpose of this paper is simple: it is to demonstrate that the programming techniques advocated by E.W. Dijkstra in 'A Discipline of Programming' are

(a) transferable to other programmers,
(b) useful and insightful, and
(c) important.

Such a simple aim is however very difficult to achieve, since we have to steer a narrow course between rigorous formality and loose reportage. The first will lose us readers we wish to interest in these ideas, while the second will disgust computer scientists with a taste for formality.

We have therefore chosen to describe attacks on two quite distinct problems, illustrating how we came by them and how we tackled them, at a level of description which is intermediate between the two extremes indicated above. Our description of invariants and correctness proofs are aimed to be understandable to ordinary programmers so as to make the paper generally useful, but leaving sufficient hints and clues so that the formal gaps can be filled in without difficulty by those with the necessary expertise.

The two problems we shall illustrate are quite common programming situations, and were not contrived for the purpose. In fact they arose independently in the course of the work of each of the joint authors. Nevertheless they serve as admirable examples of the power of clear thought in programming and, we hope, illustrate the development of correctness proofs together with the act of programming. The first example calls for traversing a tree data structure (commonly found in symbol tables, file hierarchies, search situations, databases, and many other places), while the second looks at the problem of searching a piece of text for occurrences of a given word or phrase (increasingly important in information retrieval).

It is impossible to encapsulate all the ideas expressed in 'A Discipline of Programming' in this introduction for readers unfamiliar with Dijkstra's work; indeed it would be naive to think anything less than a book would do. Obviously readers who have carefully read this work will benefit from our discussion, as perhaps will those who have skimmed the ideas which are most apparent. However, so as not to neglect the rest of the computing community, we feel we have to try to express some of what it is we are promoting:

(a) the notion that developing a correctness proof *simultaneously along with the act of programming* is the best way to program;
(b) the promotion of clear thought in programming, including the expression of *exactly what conditions* (post-conditions, pre-conditions, loop-invariants, etc) *are supposed to hold* at any particular execution point of a program;
(c) the acquisition of some *useful design rules* which aid the programmer in generalizing a given post-condition (the statement of the problem) so as to develop a program that automatically particularizes the generalization; and
(d) the removal of obsessions with control flow by the introduction of guarded commands which are executed only when the corresponding (boolean) guard conditions are true, and then sometimes without strict sequencing requirements.

We shudder at the simplication of the above, but hope that it will help. In most of what follows, the point of (d) is not relevant, and the major Dijkstra construct we shall use (a simple **do-od** loop) can be given an exact equivalent in a conventional programming language. **Table 1**. The point behind the **do-od** loop in its full generalization and in these equivalents of a simple case lies in three simple facts:

(a) there is some logical condition (a loop *invariant*) which is true when the loop is entered, every time around it, and when it is left;
(b) each time around the loop some finite progress is made towards a definite goal (this ensures termination); and
(c) when the loop terminates we know the guard condition (just *b* above) must be false.

We shall also have occasion to use Dijkstra's **if-fi** construct. In this construct, the guards (boolean conditions) are evaluated and the statement corresponding to one of the true guards is executed (an error occurring if there is no

*Both with the Department of Information Science, University of Tasmania. *Manuscript received 31st January 1978. Revised version received 10th May 1978.*

TABLE 1

DO-OD LOOP
do b →   s1;   s2;   ...   **od**;
PASCAL EQUIVALENT

while b **do**
 begin
 s1;
 s2;
 ...
 end;

FORTRAN EQUIVALENT
1 IF (.NOT. b) GO TO 2   s1   s2   ...   GO TO 1   2 ...

true guard). Since all our programs use guards which are mutually exclusive, we can give another equivalence which will suffice for this paper (see table below). We should however add a warning that if we explain Dijkstra's constructs this way, this does not mean that the programs should be implemented directly by our equivalences. The notation is currently one for *thinking in,* not for coding.

TABLE 2

IF-FI SELECTION
if b1 → s1;   □b2 → s2;   □...   **fi**;
PASCAL EQUIVALENT
if b1 **then** s1   **else if** b2 **then** s2   ...   **else** aborttheprogram;
FORTRAN EQUIVALENT
IF (.NOT. b1) GO TO 1   s1   GO TO 99   1 IF (.NOT. b2) GO TO 2   s2   GO TO 99   ...   n STOP 7777   99 ...

2.1 A First Problem: Traversing a Tree

Tree structures occur quite frequently in computing problems, and the first program to which we wish to apply Dijkstra's ideas arises from a need to traverse a tree (or in other words, to apply some procedure to every node of the tree structure). Examples of practical situations where this is needed often occur in system software:

(a) if a compiler symbol table is stored as a tree structure, then printing its contents is a tree traversal problem; and

(b) if file names in an operating system are qualified by a hierarchy of directory qualifiers, then inspecting all files in an automatic archiving process is also a tree traversal problem.

Recursive algorithms to solve this traversal problem are well-known, and easy to derive, since a tree structure is a recursively defined data structure. To make the problem more interesting, and the solution more useful, we shall look for non-recursive programs to solve this problem. This will make it usable in FORTRAN and assembly code environments.

To make the succeeding development simpler, we need to define some notation. Let us agree to describe our programs in a PASCAL-like notation, so that we can describe the procedure heading of the program we are trying to write as:

procedure traverse (T:tree ; **procedure** P);

and it is supposed to apply the procedure *P* to every node of the tree pointed at by *T* in turn. Since the representation of the tree is of minor importance for the algorithm, we shall simply use the following notation to access parts of the tree T (derived from a notation of J.B. Hext, 1972).

NOTATION	MEANING
∝T	the value of the root node of T
δT	the degree of the root node of T (i.e. the number of subtrees)
T[1]	the first subtree of T
T[2]	the second subtree of T
T[n]	the n-th subtree of T

2.2 Massaging a Recursive Algorithm

To contrast approaches, we shall first employ a classical approach. Since the problem is trivial if recursion is allowed, this immediately suggests starting with a recursive program and manipulating ("massaging") it into a non-recursive form. Accordingly, let us start with the following recursive solution to the problem, which you should understand before we go further:

```
procedure traverse(T:tree; procedure P);
    {recursive version R1}
    var i : integer;
    begin {process the root node}
        P(∝T);
        {process the subtrees}
        for i:=1 to δT do
            traverse (T[i], P);
    end; {of procedure traverse}
```

This simple algorithm effects a traversal by subtrees. In other words, for every node in the tree, the first subtree of that node is completely traversed before the second subtree and so on. Thus with the tree of Fig. 1 the order of

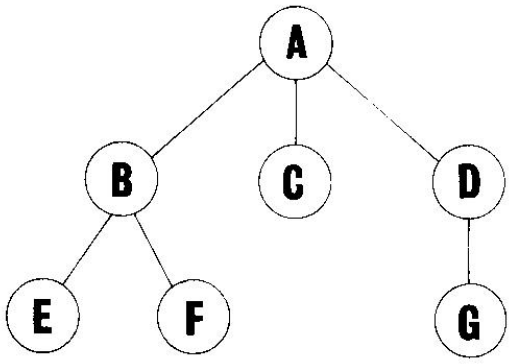

Figure 1 A tree example

traversal of nodes is: A,B,E,F,C,D,G.

In contrast, a traversal by levels would traverse all the nodes at level 0 before the nodes at level 1, etc. Then the node order for the above example would be: A,B,C,D,E,F,G. To produce this latter ordering is generally regarded as more involved since the processing cuts across the natural definition of a tree in terms of subtrees. One normally expects the tree to be differently represented if traversing by levels is to be an important operation (Hext, 1972; Knuth, 1968). We shall not consider this point further until we reach section 2.4.

Returning to the recursive algorithm, if we are to make it non-recursive then the normal approach (cf Bird, 1974) is to make explicit what has so far been implicit – the stack which has held the activation records. To simplify the discussion we define the following stack primitive operations:

NOTATION	MEANING
clear(S);	stack S is initialized to be empty
load(S,a_1,a_2,...)	loads a_1,a_2, etc. onto stack S in a left- to right order.
unload(S,a_1,a_2,...)	unloads a_1,a_2, etc. from stack S in a right-to-left order.
empty(S)	a boolean function that returns true if stack S is empty, and false otherwise.

The stack operations which have so far been implicit in the recursion must now be made explicit. They consisted of saving parameters and local variables on the stack prior to the procedure call and restoring them at procedure exit. The tree parameter (T) and local variable (i) are the only items which change with each procedure invocation and therefore they are the only ones that need to be saved on our stack.

Thus, in eliminating the recursive procedure call, the call itself will be replaced by code to load the current values of T and i onto the stack, reset T (as the new "parameter") and jump to the start of the procedure (the new "recursive call"). Then at "procedure exit", values of T and i must be unloaded and the loop continued. However, since it is not normally possible to re-enter a loop construct once it has been exitted, the loop is rewritten using **if** and **goto** statements. In other words, the loop

```
for i := 1 to δT do S;
```

is rewritten as

```
     i := 1;
loop: if i ⩽ δT then
           begin
           S;
           i := i+1; goto loop;
           end;
```

These transformations result in our first non-recursive solution (N1).

```
01 procedure traverse (T:tree; procedure P);
02       {non-recursive version N1}
03       label start, subtree, exit;
04       var   i : integer;
05             S : stack;
06       begin
07             clear(S);
08 start: {process root and then initialize for subtrees}
09             P(αT);
10             i:=1;
11 subtree: {try next subtree unless all have been processed}
12             if (i ⩽ δT) then
13             begin .
14                   {save current position and try next level}
15                   load(S,T,i);
16                   T:=T[i];
17                   goto start;
18             end;
19             { no more subtrees – go back a level }
20             if empty(S) then goto exit;
21             unload(S,T,i);
22             i:=i+1;
23             goto subtree;
24 exit:
25       end; {of procedure traverse}
```

While this algorithm is certainly effective, it may offend those of us with an aesthetic sense with its liberal sprinklings of **goto** statements. Can these be eliminated? If we look at the control flow of the program we have written (Fig. 2), it breaks up into two overlapping loops. We can achieve a tidier solution either by nesting *loop2* within *loop1*, or vice versa. The results of these two approaches are given below; the latter solution requires the loading of a dummy value.

```
procedure traverse(T:tree; procedure P);
    {non-recursive solution N2}
    label exit;
    var  i : integer;
         S : stack;
    begin
         clear(S);
         while true do
         begin
              {process root and initialize for subtrees}
              P(αT);
              i:=1;
              {find next subtree to be processed}
              while (i > δT) do
              begin
                   if empty(S) then goto exit;
                   unload (S,T,i);
                   i:=i+1;
              end;
              {save position and try deeper level}
              load(S,T,i);
              T:=T[i];
         end;
exit:
    end; {of procedure traverse}

procedure traverse(T:tree; procedure P);
    {non-recursive solution N3}
    var  i : integer;
         S : stack;
    begin
         {initialize stack and process the root}
         clear(S);
         load(S,nil,0);
         P(αT);
         i:=1;
         {continue while subtrees remain}
         while not empty(S) do
         begin
              follow subtree
              while (i ≤ δT) do
              begin
                   {save position and go a level deeper}
                   load(S,T,i);
                   T:=T[i];
                   P(αT);
                   i:=1;
              end;
              {no more subtrees – go back a level}
              unload(S,T,i);
              i:=i+1;
         end;
    end; {of procedure traverse}
```

As this now stands, N2 appears to be the less attractive solution: it contains an extra **goto** statement together with the unattractive construct "**while** true **do** . . ." (or **forever do** . . .). Yet if the tree traversal is to be a search for a particular node ("nut" say) then the outer loop of N2 becomes

"**while** (αT ≠ nut) **do** . . ."

This modified form of N2 is the one derived by J.B. Hext (1972). It is also the one which can be modified so that the search can always be guaranteed to terminate by

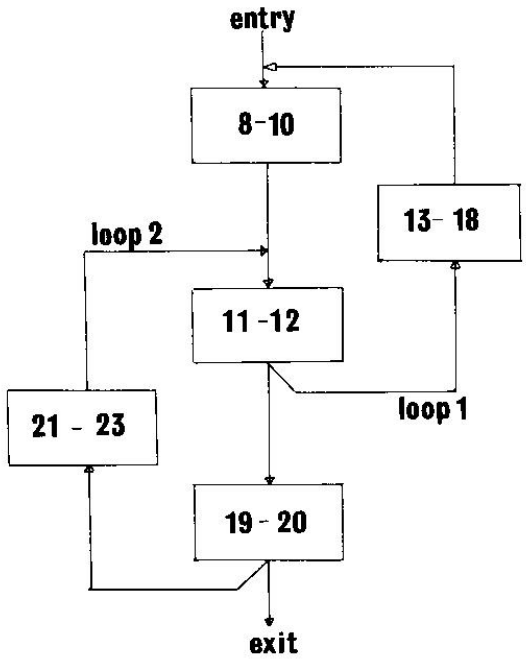

Figure 2 Control flow of program N1

encountering a sentinel placed in the tree thus making redundant the test for an empty stack. (This is the sort of approach taken by Wirth, 1976).

Program N3 has its flaws too. For example it contains a dummy load onto the stack simply to satisfy the boundary conditions.

So, we have derived two non-recursive tree traversal algorithms by transforming a recursive solution. The principal advantage of such an approach is the simplicity of the original recursive solution, and our confidence in its correctness. If all the transformation steps can be proven to be allowable, then the original together with the steps can be considered to be a correctness proof for the end-program. As such this is a respectable and very useful technique. Unfortunately, the transformation process can be long and involved if an attractive end-product is desired. This massaging serves also to decrease our confidence in the correctness of the final solutions, for each manipulation must preserve the essential nature of the process and its correctness. Can you be as confident that N_2 and N_3 are correct as you can be about R_1?

This is a very serious objection to the transformation process, since it will manifest itself in a debugging problem if we cannot be certain that our programs are indeed correct. This is exacerbated in the example by the modifications making the transformed algorithms look little like the original R1; this introduces uncertainty as to the precise significance of each statement.

2.3 Starting from Scratch

We therefore propose to attempt to derive a tree-traversal algorithm using Dijkstra's notation and thought patterns (as far as we can reproduce them). After this has been done, we can compare the results. Any readers who are familiar enough with Dijkstra's ideas are urged to attempt their own solutions to the problem before reading further since this is sure to increase awareness of the tasks involved.

Following Dijkstra's methodology, we should look at what is to be achieved, and formulate an appropriate post-condition which is capable of being generalized in moving back further from the end-result. Suppose we take this simply as:

R_1 = *all the tree nodes have been processed*

To achieve this post-condition, it is not difficult to deduce that one or more loops are going to be required, but the choice of loop invariants is not quite as obvious. From our experience or intuition, let us suggest that a stack is going to play some part in this algorithm. (For consistency we shall retain our earlier programming style and stack primitive operations; for although it is trivial to turn these into Dijkstra's vector operations, this will destroy some of the structure of what we are doing.) A stack is normally used to record partially processed items, and so it appears natural to suggest that an appropriate invariant for an outer loop of the program would be:

P_1 = *the stack holds the trees still to be processed (with the understanding that if a tree is recorded as unprocessed, all its subtrees are unprocessed too)*

Clearly then, the loop will continue while the stack is not empty. Hence we arrive at the program fragment following, which is written in a blend of Dijkstra's notation and PASCAL:

```
{put the tree on the stack – establishes P1}
clear(S);
load(S,T);
do (not empty(S)) →
      {remove an unprocessed tree}
      unload(S,T);
      {process the root node}
      P(αT);
      {re-establish P1}
      <stack the subtrees>;
od;
```

The initial load directive establishes the invariant to begin with. Then, each time around the loop, the next subtree to be processed is removed from the stack, its root is processed, and its subtrees are stacked for later processing, thus re-establishing the invariant P_1.

It is not difficult to see that the loop will terminate since the number of nodes to be processed decreases by one each time around the loop. Therefore provided the tree is a finite one, termination is guaranteed. Since we are then assured that the guard condition is false, it follows that the invariant taken together with:

not empty(S) = false

i.e. empty(S) = true

implies the desired postcondition R_1. It only remains to fill in the details of 'stack the subtrees', and this is simply done:

```
{Dijkstra style version D1}
{initialize, establish P1}
clear(S);
load(S,T);
do (not empty(S)) →
      unload(S,T);
      P(αT);
      i:=δT;
      {stack the subtrees}
      do (i ≠ 0) →
         load(S,T[i]);
         i:=i-1;
      od;
od;
```

Readers will have noted that the subtrees are stacked in reverse order, so that they will be unloaded, and therefore processed by P, in left-to-right order. (In other words we are preserving the behaviour of our earlier algorithms in that the first subtree is processed before the second, etc.). You may be struck, as we were, by the simplicity of the solution. Furthermore, the derivation was supported by theoretical considerations, and we are guaranteed of its correctness by our reasoning. Our level of confidence that this solution is correct is much higher than with the version N_2 or N_3.

Having created a solution, we might still try to improve on it. We observe, for example, that we could improve the storage efficiency of the algorithm. Each time around the outer loop we save all the subtrees (as pointers perhaps) on the stack. Instead we could record a pointer to the root node, and the number of the next subtree to be processed: this is similar to our approach in converting algorithm R_1 to N_1. Then if we assume a pointer and an index take the same storage space, we shall use less stack space if the average node branching ratio is less than two. (In fact, with a homogeneous tree of degree d and height h, this approach uses (2xh) stack elements and $4\sum_{i=0}^{h-1} d^{i+1}$ stack accesses whereas the earlier solution used a maximum of (dxh – h+1) stack elements and $2\sum_{i=0}^{h} d^i$ stack accesses.)

The outer invariant will need to be reformulated to become:

P2 = *the stack contains an indication of the subtrees to be processed, such that a pair, (T, i) on the stack implies that subtrees T[i], T[i+1], ... T[δT] are still to be processed.*

The results in the program:

```
{Dijkstra style version D2}
{initialize – establish P2}
clear(S);
P(αT);
{save indication of subtrees}
if (δT = 0) → skip;
□ (δT > 0) → load(S,T,1);
fi;
do (not empty(S)) →
        unload (S,T,i);
        {save remaining subtrees, if any}
        if (i = δT) → skip;
        □(i < δT) → load(S,T,i+1);
        fi;
        {follow this subtree}
        T:=T[i];
        {process its root}
        P(αT);
        {and save its subtrees}
        if (δT = 0) → skip;
        □(δT > 0) → load(S,T,1);
        fi;
od;
```

We make one final observation. The algorithm D_2 is formulated in terms of *subtrees*. Since the entire tree is not itself a subtree, the root must be processed outside the loop. To eliminate this the invariant must again be altered giving P_3 (below) with the resulting solution D_3.

> P_3 = *the tree T (provided it is not nil) is still to be processed, so are the subtrees indicated on the stack, where (T,i) on the stack implies that subtrees T[i+1], T[i+2], . . . T[δT] are still to be processed.*

```
{Dijkstra style version D3}
clear(S);
do (T ≠ nil) →
        {process a node}
        P(αT);
        {and its subtrees}
        load(S,T,0);
        T:=nil;
□       (T=nil) and not empty (S) →
        {process a subtree}
        unload (S,T,i);
        i:=i+1;
        if      (i ≤ δT) →
                load(S,T,i);
                T:=T[i];
        □       (i > δT) →
                {subtree completed}
                T:=nil;
        fi;
od;
```

2.4 Using a Queue

When algorithm D_1 was originally derived, the use of a stack seemed 'natural', preconditioned as we were by the existence of other known solutions. However, the form of derivation of this version highlighted an obscure possibility: what other structures could be used to hold subtree information as yet unprocessed? Clearly stacks, queues, and even trees! Let us explore the possibility of replacing the stack in algorithm D_1 by a queue. To do this we shall need some new primitive queue operations:

NOTATION	MEANING
open(Q)	queue Q is initialized to be empty.
join(Q,a_1,a_2,. . .)	add a_1, a_2, etc to queue Q in left-to-right order.
leave(Q,a_1,a_2,. . .)	remove a_1, a_2, etc from queue Q in left-to-right order.
empty(Q)	a boolean function that returns true if queue Q is empty, and false otherwise.

We can now derive program version D_4:

```
{Dijkstra style version D4}
open(Q);
join(Q,T);
{initialized, established P1}
do (not empty (Q)) →
        leave(Q,T);
        P(αT);
        i:=0;
        {enqueue the subtrees}
        do (i ≠ δT) →
                i:=i+1;
                join(Q,T[i]);
        od;
od;
```

To illustrate the conversion of this Dijkstra-like notation into a practical program, a PASCAL program equivalent to it is given in the Appendix.

Astonishingly perhaps, this solution turns out to be a significant one – it results in the tree being traversed by levels, instead of subtrees. The result is simple, beautiful, and apparently not widely known. We referred to traversal by levels in section 2.1, and indicated that the common view was that this required a special data-structuring, so that the structure should reflect the operations commonly required. Here however the transition has been achieved almost painlessly: apparently the only price to pay is the size of queue required whose maximum size is proportional to the maximum width of the tree.

2.5 An Examination of this Problem

So far we have been considering the production of a non-recursive algorithm for tree traversal. We found that if we modified a simple recursive solution, the indirect nature of the approach meant that we did not have a very good grasp of the end-product, and consequently our confidence level in our ability to avoid mistakes (in proof or in coding) suffered.

In contrast, the use of Dijkstra's approach to programming, once we had mastered the change in thought-pattern required, yielded better results – the resulting programs were neater and the accompanying development of a proof *while* we were programming meant that we could be very confident of the result. As an added bonus, we found that the same basic solution could yield either a traversal by subtree, or a traversal by level, depending on whether a stack or a queue was used to store

the unprocessed subtrees. The discovery of this new (to us at least) solution is directly traceable to our better understanding of what we were doing, and of the decision freedom we had at each point.

It is interesting to go back and ask ourselves where we went wrong in our traditional derivation of tree traversal algorithms, for the recursive algorithm R_1 exhibits the same compelling elegance as the Dijkstra-style algorithm D_1. Indeed, this is the clue, for the two algorithms are *very* similar and have the same invariant and proof structure. The only essential differences are that R_1 processes one node and initiates the processing of the associated subtrees each recursive activation whereas D_1 does this each time round the outer loop. Other than that, they are identical. They both reflect the basic unit of the data structure, viz. a node with a set of subtrees. The more we depart from this ideal and the more we become obsessed with control flow and the details of the algorithm, the messier our programs become and the weaker is our grasp of what the algorithm was trying to do. We think that such insights must be chalked up as a plus for the techniques we used because the discipline of identifying and writing down the invariants and post-conditions allowed us to keep the basic tree structure in mind during the whole process, and consequently the derived programs reflected the structure of the data. By contrast, though it is not impossible to go from program R_1 to our D_1, it is very easy to be side-tracked by some detail of the transformation.

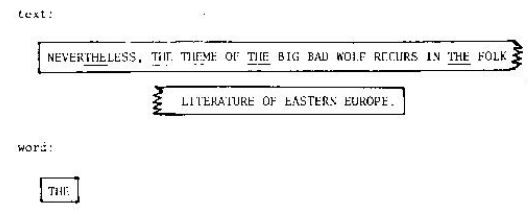

Figure 3 Illustrating the text-search or pattern-match problem.

3.1 The Text-Searching Problem

As a second example, we shall look at the problem of searching some text for occurrences of a particular (smaller) piece of text. Suppose that we have two objects declared:

```
var
    text : array[1..M] of char;
    word : array[1..N] of char;
```

Fig. 3 illustrates one possible situation. To make the problem formulation general and similar to the previous one, assume that we are asked to apply a procedure P to every occurrence of *word* in *text*. Thus if we want simply to count the occurrences, or print them in context, or take some other action, we can supply particular versions of P for our needs. Thus we can define a PASCAL-like procedure heading for the program we are going to write:

```
procedure search (text,word : array of char;
                  M,N : integer;
                  procedure P);
```

This is a famous problem and many algorithms have been devised to tackle it. We can first exhibit the most obvious solution, which is one which appears to be derived by most beginning programmers faced with this problem.

```
{simple-minded search: version S1}
for i:=1 to (M-N+1) do
begin
    j:=0;
    while (j ≠ N) cand (text[i+j] = word[j+1]) do
    begin
        j:=j+1;
    end;
    if (j = N) then P(i);
end;
```

It should not need much explanation, except perhaps for the connective **cand** (conditional-and). This connective is similar to the familiar **and** for logical algebra, except that the right term of a **cand** is evaluated only if the left term proves to be true. Assuming that **if-then-else** is evaluated straightforwardly, the following equivalence holds:

(b1 **cand** b2) ≡ (**if** b1 **then** b2 **else** false)

Program S_1 is however a very poor algorithm. Its best performance requires it to inspect (M-N+1) characters, while its worst performance requires the inspection of approximately (M x N) characters. We have given this algorithm for comparison purposes, and because most people seem to think of it first. It is instructive for the reader to prove it correct, and to deduce what properties of *text* and *word* evoke the best- and worst-case performances.

Dijkstra, too, has tackled this problem in 'A Discipline of Programming', and a short but very perceptive chapter (No 18) is devoted to it. The solution naturally proceeds by specifying a post-condition R_1 from the statement of the problem, intuiting that a loop is needed, and generalizing R_1 to give Dijkstra's outer invariant R_2:

R_1 = *all occurrences of word in text have been processed by P.*

R_2 = *all occurrences of word which have its first character in any of text positions 1 to i have been processed by P.*

It is interesting to note in passing that this is exactly the same invariant needed to prove S_1 correct! Thereafter, however, the two solutions part company. The simple-minded search simply advances i by one (cautiously) each iteration, and uses none of the information which may have been obtained from previous match attempts. Dijkstra's program is however based on advancing i by as far as is safe to go based on the available evidence, and using that evidence also to begin matching as far advanced as is warranted. Necessarily this involves pre-analysis of the word to set up a table of safe increments *d* (indexed by the last depth of successful matching), and Dijkstra gives also an algorithm to create this table. Since Dijkstra's treatment is already available, we exhibit the algorithm for searching without comment, except to say that it inspects of the order of M characters in all cases, and that we have slightly modified the solution to accord with our conventions.

```
{Dijkstra-search : version S2}
<initialize table d>;
i,j := 1,0;
do (i ≤ (M-N+1)) →
      do (j ≠ N) cand (text[i+j] = word [j+1]) →
         j:=j+1;
      od;
      if (j = N) →        P(i);
                          i,j := i+d[j], j-d[j];
      □(0 < j < N) →      i,j := i+d[j], j-d[j];
      □(j = 0) →          i:=i+1;
      fi;
od;
```

3.2 The Skip Algorithm

The initial impetus to tackle this problem came from a paper written for publication by Geoff Dromey (University of Wollongong, 1977). The algorithm proposed therein (which we propose to call the skip algorithm) was developed on quite different premises from the two programs we have considered so far. Dromey argues that matches with typical text could be expected to be rare, and that the algorithm should minimize the number of characters inspected on average. From this base it is quite easy to see that the absolute minimum number of characters that might need to be inspected is (M/N), and this is achieved if characters at text positions N, 2 x N, 3 x N, etc are inspected and none of them actually occur in the word searched for..

From this idea, the algorithm develops. If the character inspected does not occur in the word, make the next inspection N characters further on; if it does occur in the word but not as the last character, make the next inspection where the end of the word may be expected to be; and if it is the same as the last character of the word, attempt a retrospective match, after which re-apply the two previous rules. Care is needed to generalize the above to cases where characters occur multiply in the word, and possibly also as the last character. Dromey also suggests that the most effective retrospective search is one which inspects the character positions relative to the presumed end-of-word in increasing occurrence probability order of the actual characters in the word. This maximizes the probability of not finding a match, and therefore of being able to leave off matching and resume forward stepping.

The key point about the Dromey algorithm is that it inspects comparatively few characters if matches are in fact rare. The best-case performance, as indicated above, calls for (M/N) character inspections, and the average performance on English text approaches this limit for short words (say N ≤ 10). As N is increased, the performance relative to the (M/N) limit degrades, and for very long words the algorithm tends to take an asymptotic stride through the text which is determined by the probability distributions of characters in the text and in the word. This behaviour should be well-developed in English text for N=100, but nevertheless despite the degradation from the limit, the number of characters inspected is much smaller than in S_1 or S_2. The worst-case performance for Dromey's original version is poor (MxN inspections), but the case for the algorithm rests on this being very rare.

After preparation of the manuscript for this paper, we noted that the essentials of the algorithm proposed by Dromey were published by Boyer and Moore (1977) in a recent issue of the Communications of the ACM, and had been in draft form since 1975. The two algorithms differ in some details which affect the average and worst-case performances. Our concern is with program design, not primarily with particular forms of the algorithm, and interested readers are referred to this article for a version which has a worst-case performance which is of order (M + N) while preserving the low number of probes for typical text searches. Our future discussion will refer to the basic approach as the 'skip algorithm' since many characters in the text are skipped.

The description of the skip algorithm given above is sketchy, and deliberately so. The algorithm appears to be difficult to describe in its original form: Dromey's description is quite long, and Boyer and Moore's description runs to about 2000 words. One of the authors was dissatisfied with this complexity, and with the difficulty of proving it formally correct, together with some niggling concerns about its performance for large N. (Neither paper attempted a formal correctness proof; the suggested programs are structurally messy.) This motivated us to undertake a proof of the algorithm which we shall shortly develop. However, it is interesting to note that we found this process much harder than any of the other programs in this paper; indeed the polished version presented here was arrived at only after this section had been re-drafted twice. The moral is, of course, that it is much more difficult to prove someone else's program correct than it is to develop correctness proofs of your own programs as you design them. Naturally we expected this, but it does not seem to have percolated through to the industry as yet.

The key that we need to develop the skip algorithm cleanly is the generalization of the post-condition R_1 into the following invariant which we propose for the outer loop of the program:

> *R_3 = all occurrences of word which have its last character in any of text positions 1 to (i-1) have been processed by P.*

Clearly, no such occurrences occur in the first (N-1) character positions of the text, and we can establish R_3 trivially by setting i to N. Inside the loop, we need to increase i under invariance of R_3, and two cases immediately present themselves. If the character inspected matches the last character of the word, then we need to carry out a retrospective search to establish if a match of word actually occurs and if so to process it with P, and then we need to increase i as far as we can keeping R_3 invariant.

The retrospective search will at least have warranted the increase of i by 1, so the loop will terminate.

If on the other hand, the character inspected is not the last one in the word, we simply advance i by the maximum permitted keeping R_3 invariant. If the character does not appear in positions 1 to (N-1) of the word we can advance by N positions, while if it does we can only advance by the distance from the end of the word to the nearest occurrence of the character in the word. Fig. 4 illustrates this process. This calls for pre-processing of the word to set up a table of safe increments D indexed for each character possible. This gives us the following skeleton program:

```
{skeleton of skip family of programs : S3}
{establish R3 trivially}
i:=N;
do    (i ≤ M) →
      ch:=text[i];
      if    (ch = word[N]) →
            <attempt match based on i as last char>;
            <increase i under invariance of R3>;
            {having re-established R3}
      □     (ch ≠ word[N]) →
            i:=i+D[ch];
      fi;
      {R3 is still true}
od;
{(i > M) and R3 imply R1}
```

So far we have not specified precisely the actions to be taken when the last character is found, for there is a whole family of skip algorithms depending on how this part is filled out. The average performance of the algorithm for normal text is dominated by the alternative guarded command which strides through the text. The last-character part mainly determines the worst-case performance, and has minor effects on average performance. It is however interesting to consider the possibilities.

Dromey's original choice can be constructed by setting up a table F which holds records containing each of the characters in the word (except the last) and their position indices. This might be declared as:

```
F :   array [1..(N-1)] of
      record
            c : char;
            index : 1 .. (N-1)
      end;
```

The initialization of this table consists of setting up all the leading characters in it and sorting it on ascending order of a virtual key of the occurrence probability of c in the text. We can then fill out the first **if** guarded command to give Dromey's original algorithm:

```
...
if    (ch = word[N]) →
      {identify and process any match}
      j:=1;
      do    (j ≠ N) cand
            (text[i-N+F[j].index] = F[j].c) →
            j:=j+1;
      od;
      if    (j = N) →          P(i-N+1);
      □     (1 ≤ j < N) →      skip;
      fi;
      {increase i under invariance of R3}
      i:=i+D[ch];
□     (ch ≠ word [N])
      ...
```

It will be noted that once having attempted to identify possible matches based on the last character of the word at text[i], this version simply bases its increase of i on the same criterion as the alternative command; i is increased as much as possible based simply on the character at text[i].

Let us look at alternative expansion possibilities. One is simply to try to match the word backwards from text[i] and word [N]. We will have increased the probability that the match proceeds past the first character inspected, but it is unlikely this will make much difference on average, and we will have eliminated table F. On the other hand, looking deeper, we see that a pattern analysis of the kind used in Dijkstra's program S_2 could be carried out for backwards matches, and we could construct a new table (indexed by greatest depth of match) giving the safe increment based on all the information we have, after which we ignore it.

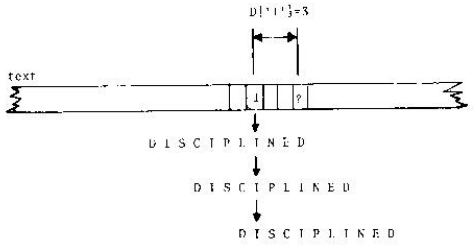

Figure 4 Possible alignments of the word 'DISCIPLINED' when an 'I' is found in the text, and the safe displacement of 3.

Such increments would be at least equal to that for D[word[N]], and usually larger. This corresponds to Boyer and Moore's algorithm. Of course, such pattern analysis is not restricted to linear scanning, but could be applied to the least-frequent-first search too, but the complexity is not warranted. There are lots of possibilities . . .

Since the skip algorithm deserves to be better known, we exhibit a simple and efficient version of it in an appendix. This example uses a PASCAL-like notation and utilizes a linear backward match scan together with use of the D-table.

3.3 Starting from Scratch Again

Again, we determined to tackle the problem with fresh eyes, and to try to use some of the insights we have been given by our previous attempts. One possibility that occurred to us from consideration of Dromey's algorithm was to rewrite our generalized invariant so that it was based on specific character positions:

> *R_4 = all occurrences of word which include the characters at text positions N, 2 x N, 3 x N, . . . , i have been processed by P, where i is an integer multiple of N.*

What we are aiming at, of course, is an algorithm whose performance is better than S_3, and hopefully achieves (M/N) performance even for large N. What we will achieve (looking ahead) is a new algorithm with very similar performance to S_3, but a quite different process. Using the invariant above, this leads directly to our first-level program fragment:

```
i:=0;
{R4 has been established trivially}
do    (i ≤ (M-N)) →
      i:=i+N;
      <process all occurrences that include text [i]>;
      {R4 has been re-established}
od;
{R4 together with (i > (M-N) implies R1}
```

To fill out the 'process all occurrences that include text [i]' part, we can again use the insights provided by Dromey's algorithm by passing straight on if the character is not represented in the word. If it does occur in the word, however, we need to carry out searches to see if there are any complete matches here that must be processed by P. There are many ways of handling this, and the one we choose to illustrate the algorithm is to create a table Z which for every possible character holds a queue of records which detail the actual position indices of that character in the word. If the character does not occur in the word, the corresponding queue is empty. For notational convenience, we shall also define the following basic operation on this table:

NOTATION	MEANING
select(Z,ch,Q)	selects the queue for character ch in table Z and makes a local copy in queue Q.

We again suspect that a loop will be useful in processing the possible occurrences, and so we need to generalize our invariant still more, suggesting that it ought to be:

> R_5 = *all occurrences of word which include the characters at text positions N, 2xN, 3xN, . . . , i have been processed except for those which correspond to having the character text[i] in the word positions noted in the queue Q.*

This now allows us to write the following program fragment to replace our outlined section:

```
{process all occurrences that include text[i]:}
select(Z,text[i],Q);
{R5 is now established trivially}
do   (not empty(Q)) →
     leave(Q,j);
     <attempt match of text[i-j+1|
       with word and process>;
     {re-establishing R5}
od;
{R5 and empty(Q) implies R4}
```

The final expansion calls for some simple match procedure to examine definite word positions in the text. Though the match procedure involves characters behind and ahead of text[i], it is possible to carry out the same types of search as were possible for the skip algorithm, for the presumed location of the last character of the word (if the match exists) is located at text[i-j+N]. If we assemble the lowest frequency search into here, we can pull all these pieces together into a whole program:

```
{new search program : version S4}
<initialize table Z>;
<initialize table F>;
{remember F must now include the last char}
i:=0;
do   (i ≤ (M-N)) →
     i:=i+N;
     select(Z,text[i],Q);
     do   (not empty(Q)) →
          leave(Q,j);
          base,count:=(i-j),0;
          do   (count ≠ N) cand ((base+N) ≤ M) cand
               (text[base+F[count+1].index]
               = F[count+1].c) →
               count:=count+1;
          od;
          if (count = N) →        P(i-j+1);
          □(0 ≤ count < N) →      skip;
          fi;
     od;
od;
```

3.4 Efficiency and Other Reflections

Clearly the new algorithm S4 we have derived is not a skip algorithm. It is easy to prove correct, but it is more efficient? There is not room here to reproduce an analysis of either of the skip algorithms contained in the two previous papers, nor to analyse this new one, but it can be said that they all perform similarly. As N becomes very large, they all reach a situation where the average 'stride' through the text (measured as M/(no of inspections)) tends to the same limit. This limit is determined by the character probability distributions in the two objects, which may not be the same. We have not therefore been able to find a *better* algorithm, though we have found a *different* one. It is also interesting to note that the additional space requirements of all algorithms grow proportionally to N; we have omitted to discuss algorithms which have space requirements which grow faster or which require other expensive space requirements.

Again, we reflect, the necessity of having to write down and formalize our invariants has led to clarity of thought. While we were looking for algorithmic twists to improve performance, the clarity with which we could view the choices provided us with a wide range of options. With the experience we gained, we found a clean and simple way to prove Dromey's algorithm correct which hinged on a de-emphasized facet of the original description. In Dromey's original statement, the last character of the word appeared to play a minor role in the algorithm; it turns out to be crucial to the proof.

4. SUMMARY

We have intended to show with this paper that the techniques advocated by Dijkstra are transferable to other programmers. Indeed, our own experiences have been that the transfer is easy and profitable once the mental switch in thought patterns has been made. We have enjoyed the experience, and have been surprised at the insights we have found into even some rather simple problems. In fact, in preparing this paper for publication, we have had to rather ruthlessly prune many byways which we found interesting for fear of complicating our exposition. We conclude therefore that the techniques are indeed *transferable* and *insightful.*

That we produce algorithms which are novel to us is not important; for all we know they are already buried somewhere in the literature. What is important is that we were led to discover them in natural ways, and that in each case we could express a high degree of confidence in the correctness of the resulting algorithm. Debugging programs is not an activity either of us pursues very often, and we perceive that we can reduce its frequency even further.

We had, too, an ulterior motive. All too many industry programmers are likely to be put off by the first few chapters of Dijkstra's book. We hope that by illustrating what can be done in a less formal way, some programmers may be encouraged to probe deeper into what we consider to be very important ideas. Though there remains much work to be done (when is there not?), we think the orientation important enough that we are teaching it as part of our undergraduate coursework. The ideas will be in the marketplace quite soon . . .

5. REFERENCES

BIRD, R.S. (1977): "Notes on Recursion Elimination", *CACM*, Vol 20, No. 6 pp434-439.

BOYER, R.S. and MOORE, J.S. (1977): "A Fast String Searching Algorithm", *CACM*, Vol 20, No. 10 (October 1977), pp762-772.

DIJKSTRA, E.W. (1976): "A Discipline of Programming", Printice-Hall.

DIJKSTRA, E.W. (1976): "Formal Techniques and Sizeable Programs", *Lecture Notes in Computer Science*. 44 EC1 Conference, 1976.

DIJKSTRA, E.W. (1975): "Guarded Commands, Non-determinancy and Formal Derivation of Programs", *CACM* Vol 18, No. 8, pp453-457.

DROMEY, R.G. (1977): "Quicksearch – An algorithm for fast keyword searching in text", *Paper submitted to CACM*, obtainable from the author, University of Wollongong.

GRIES, D. (1977): "An illustration of current ideas on the Derivation of Correctness Proofs and Correct Programs." *IEEE, Trans. Soft Eng. SE2* 4, 238-244, 1976 + correction in p.262 May 1977.

HEXT, J.B. (1972): "Data Structures", unpublished lecture notes circulated to students at the University of Sydney.

KNUTH, D.E. (1968): "The Art of Computer Programming – Volume 1 – Fundamental Algorithms", Addison-Wesley.

WIRTH, N. (1976): "Algorithms + Data Structures = Programs", Prentice-Hall.

6. APPENDIX: PASCAL EQUIVALENTS OF FINAL SOLUTIONS

In this appendix, we give equivalents of some of the Dijkstra-style programs we derived in the body of this paper for two pragmatic purposes:

(1) to illustrate the process and make it apparent that useful programs can really be produced this way, and

(2) to make the algorithms and our arguments more accessible to readers who may not be familiar with Dijkstra's notation.

The first example is an equivalent of the D_4 version of tree traversal; it is presented as a procedure with some surrounding type declarations in a slightly extended PASCAL to supply a context:

```
type
    treepointer = ↑ treenode;
    treenode = record
                   value : valuetype;
                   degree: integer;
                   subtree: array [1 .. degree] of treepointer
               end;

procedure traverse(T:treepointer; procedure P);
    {this is version D4 converted to a particular PASCAL purpose}
    var
        i : integer;
        Q ; queue;
    begin
        open(Q);
        join(Q,T);
        while not empty (Q) do
        begin
            leave(Q,T);
            P(T↑.value);
            for i:=1 to T↑.degree do join(T,T↑.subtree[i]);
        end;
    end; {of procedure traverse}
```

The second example is a procedure to implement a version of the skip search algorithm. It is written in a slightly extended PASCAL to permit notational flexibility.

```
procedure search (text,word: array of char;
                  M,N: integer;
                  procedure P);

label 1;
var
    i,j : integer;
    ch,lastch : char;
    D : array[char] of integer;
begin
    {set up table D}
    for ch:=chr(0) to maxchar do D[ch]:= N;
    for i:=1 to (N-1) do D[word[i]]:=(N-i);
    {set up lastch for efficiency}
    lastch:=word[N];
    {establish R3 trivially}
    i:=N;
    while (i ≤ M) do
    begin
        ch:=text[i];
        if (ch = lastch) then
        begin {this is an occurrence of the last character}
            j:=(N-1);
            while (j ≠ 0) do
            begin
                if (text[i-N+j] ≠ word[j]) then goto 1;
                j:=j-1;
            end;
            P(i-N+1);
    1:
        end;
        {advance i under invariance of R3}
        i:=i+D[ch];
    end; {of while}
end; {of procedure}
```