THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 60, No. 9, November 1981
Printed in U.S.A.

McDonald's Problem—An Example of Using Dijkstra's Programming Method

By D. K. SHARMA

(Manuscript received April 20, 1981)

We use Dijkstra's programming method to solve the so-called McDonald's problem and show how to rigorously introduce file input/output operations in the program. The steps involved are quite simple and the paradigm suggested is applicable to the wider class of problems that involve sequentially processing file records. For these problems, the programs developed using the data structure design methodology are generally considered to be the most desirable. We show that Dijkstra's method can yield the same program. Unlike other methodologies, it also yields a correctness proof, which is extremely valuable in understanding the program and in modifying it.

I. INTRODUCTION

In this paper, we use Dijkstra's method of simultaneously developing a program and a proof of its correctness to solve the so-called McDonald's warehouse problem. The problem briefly is to read a card file and print an inventory report. Our interest in it stems from the fact that it requires sequentially processing the records of a file—a task common to a large class of problems. As we solve the problem, we illustrate how to rigorously introduce file input/output operations within the framework of Dijkstra's method. This aspect of the solution is intended to be a paradigm that is applicable to the above-mentioned class of problems.

Our solution serves one other purpose. The McDonald's warehouse problem is often used to compare the effectiveness of different programming methodologies in developing programs that sequentially process files. As discussed in Ref. 1, the programs developed using the data structure design methodology are generally considered to be the most desirable. This is primarily because the structure of the resulting

program closely reflects the structure of the input data. The program developed in this paper is identical to that developed by the data structure design methodology, except that the latter has no correctness proof associated with it. The proof-related assertions in a program are not only helpful in understanding it, but also in systematically modifying it.

The solution discussed below is quite simple and regular—as a paradigm it can be systematically applied to other similar problems. It is obtained in three steps. In Step 1, we develop the program and its proof assuming that the records of the file are available in an array. This version of the program also uses a few symbols that are related to its proof, assuming that their values are readily available. These assumptions are made purely for the sake of convenience in developing the program and are removed in the next two steps. In Step 2, we (*i*) introduce additional program variables, (*ii*) modify the assertions to reflect the introduction of the new program variables, and (*iii*) add appropriate statements that make the assertions hold. Thus, we are guaranteed that the program remains correct through all the modifications. This is done to eliminate from the program text the symbols whose values are not readily available. In Step 3, we introduce the file operations. This is done by replacing suitably chosen initialization statements by *openfile* and by replacing certain other groups of assignment statements, by *readfile* or *writefile*.

Section II presents the problem, first informally and then formally. This is followed by the three steps of the solution in Section III and by a summary in Section IV.

II. PROBLEM SPECIFICATION

McDonald's food warehouse receives and distributes food items. Each shipment received or distributed is recorded on a punched card that contains the name of the item and the change in the quantity of the item due to that shipment. The change is recorded as a positive integer when items are received and negative otherwise. These cards are alphabetically sorted according to item names by another program.

The problem is to write a program to read the sorted card file and print an inventory report. The report should show the net change in the inventory of each item transacted and the number of distinct food items transacted. At this point, the reader may wish to devise his or her own solution and later compare it with the solution derived below.

The following is a more formal statement of the problem, which is used in developing the solution in Section III.

Assume that the transaction file contains $M - 1$ cards, and the ith card contains two fields, $f(i)$ and $q(i)$, where $1 \leq i \leq M - 1$, $f(i)$ is a positive integer representing the name of the food item on the ith card,

and $q(i)$ is an integer representing the change in the quantity of $f(i)$. Note that, without any loss of generality, we have assumed $f(i)$'s to be integers instead of identifiers.

The file has been sorted in the nondescending order of $f(i)$'s. For the sake of discussion, we augment the file with an extra card signifying "end of file" (EOF) condition for which $f(M) = \text{EOF}$ and $q(M) = 0$. The value EOF is assumed to be greater than all the other $f(i)$'s to maintain the sorted nature of the file.

Clearly, all the cards of a particular food item are grouped together in the file. Let N be the number of distinct food items, that is, the number of groups in the augmented file. Let m_n be the index of the first card of the nth group, where $1 \le n \le N$. That is,

$$m_1 = 1$$

and $m_n = \min\{i : m_{n-1} < i \le M \quad \text{and } f(i-1) < f(i)\} \quad \text{for } 1 < n \le N.$

The m_i's have the following property

$$1 = m_1 < m_2 \cdots < m_{N-1} < m_N = M.$$

Define p_n to be the net change in the quantity of the nth food item, where $1 \le n < N$. We do not define p_N, which corresponds to the fictitious card used to augment the file. We can express p_n as

$$p_n = \sum_{m_n \le i < m_{n+1}} q(i) \quad \text{for } 1 \le n < N.$$

Note that p_n's are the values to be printed in the report.

We can now define the goal of the program as: Print $N - 1$ lines such that the nth line contains the name of the nth food item [(i.e., $f(m_n)$)] and the net change in the quantity of the food item, i.e., p_n. Then print a line containing $N - 1$, the number of food items transacted. Note that N is the number of groups in the augmented file.

III. SOLUTION

In the following, we first develop a program to print the first $N - 1$ lines of the report and add the last line later.

We assume that the reader has at least a cursory knowledge of the methodology described in Refs. 1 and 2. See Ref. 3 for a brief tutorial.

We develop the iterative solution in three steps. In Section 3.1, we assume that N is known, and the following are available as arrays:

$$f(i) \quad \text{and} \quad q(i) \quad \text{for} \quad 1 \le i \le M, \text{ and}$$

$$m_i \quad \text{for} \quad 1 \le i \le N.$$

In Section 3.2, we remove the above assumptions one by one, as

described in Section I. This is done by introducing new program variables, etc., while still maintaining program correctness. This culminates in a program in which N need not be known, and only one element each from the arrays $f(i)$ and $q(i)$ appears in the loop.

In Section 3.3, we identify statements that can be replaced by file operations. This substitution is quite mechanical and yields a program that sequentially reads the card file and prints the first $N-1$ lines of the report. The post-assertion of this program is then used to print the last line of the report.

3.1 Step 1

In the notation of Ref. 1, the result assertion is given by

$$\text{R1: } (\mathbf{A}\ i : 1 \le i < N : p_i \text{ has been printed}).$$

This should be read as follows: for all i such that $1 \le i < N$, p_i has been printed.

The iterative statement will be the main part of the program. Its loop invariant $P1$ is obtained by weakening the result assertion $R1$, that is, replacing the constant N by a variable n. Thus, we have

$$P1\text{: } (\mathbf{A}\ i : 1 \le i < n \le N : p_i \text{ has been printed})$$

$$\text{and} \quad (P1 \wedge n = N) \equiv R1.$$

The first version of the program is as below.

Solution 1.1

$n := 1$; $\{P1\}$
do $n \neq N \rightarrow$
  Increase n under the invariance of $P1$.
od $\{P1 \wedge n = N\}$.

This program begins with an initialization step that trivially establishes $P1$. The loop increases n and keeps $P1$ invariant; therefore, at the end we can assert $P1 \wedge n = N$, which is equivalent to $R1$.

To show termination, choose the termination function

$$t = N - n.$$

The value of t is initially $N-1$, each iteration of the loop reduces it, and $t \ge 0$. The loop, therefore, must terminate.

We now add a statement to increment n:

$n := 1$; $\{P1\}$
do $n \neq N \rightarrow$
  $S1$; $\{Q\}$
  $n := n + 1$ $\{P1\}$
od $\{P1 \wedge n = N\}$.

Here, Q is the "weakest precondition such that the execution of '$n := n + 1$' will establish $P1$." It is obtained by replacing all occurrances of n in $P1$ by $n + 1$, and the resulting expression is denoted by $P1|_n^{n+1}$. Thus,

$$Q = wp(\text{“}n := n + 1\text{”}, P1) = P1|_n^{n+1},$$

where $wp(S,P)$ is the weakest precondition in which execution of S will establish P.

The statement $S1$, starting execution in state $P1$, must establish $P1|_n^{n+1}$. This can be simply done by computing the value of p_n and printing it. Thus, $S1$ can be refined as:

$$\begin{aligned} &S1\text{: } \{P1\} \\ &\quad S2; \qquad \{sum = p_n \wedge P1\} \\ &\quad print\ (sum)\ \{P1|_n^{n+1}\}, \end{aligned}$$

where the program variable *sum* has been introduced.

We now refine $S2$. It has the property

$$\{P1\}\ S2\ \{R2 \text{ and } P1\},$$

where

$$R2\text{: } sum = p_n = \sum_{m_n \le i < m_{n+1}} q(i).$$

Statement $S2$ will be an iterative program, and to get its loop invariant $P2$, replace the constant m_{n+1} by a variable m. Thus,

$$P2\text{: } sum = \sum_{m_n \le i < m \le m_{n+1}} q(i)$$

$$\text{and} \quad (P2 \wedge m = m_{n+1}) \equiv R2.$$

Using the same technique as before, $S2$ is refined as

$$\begin{aligned} &S2\text{: } m := m_n;\ sum := 0; \{P2\} \\ &\quad \textbf{do } m \neq m_{n+1} \rightarrow \\ &\qquad S3;\ \{P2|_m^{m+1}\} \\ &\qquad m := m + 1 \{P2\} \\ &\quad \textbf{od } \{P2 \wedge m = m_{n+1}\}. \end{aligned}$$

Notice that the initializations establish $P2$ in the beginning and the post-assertion is equivalent to $R2$. Statement $S3$ must have the property

$$\{P2\}\ S3\ \{P2|_m^{m+1}\}$$

and can be easily shown to be $sum := sum + q(m)$.

This gives us Solution 1.2, after assembling all the pieces.

Solution 1.2

$n := 1; \{P1\}$
do $n \neq N \rightarrow$
  $m := m_n$; $sum := 0$; $\{P2 \wedge \mathbf{m} = \mathbf{m_n}\}$
  do $m \neq m_{n+1} \rightarrow$
    $sum := sum + q(m)$;
    $m := m + 1\{P2\}$
  od; $\{P2 \wedge \mathbf{m} = \mathbf{m_{n+1}}\}$
  print (*sum*);
  $n := n + 1\{P1 \wedge \mathbf{m} = \mathbf{m_n}\}$
od $\{P1 \wedge \mathbf{m} = \mathbf{m_n} \wedge n = N\}$.

We have added "$m = m_n$" to the assertions where it happens to hold. This is indicated in bold and is used in the next section.

3.2 Step 2

Solution 1.2 is unsatisfactory since it explicitly uses N, m_n, and m_{n+1}, which are not available *a priori*. We modify it in the following to eliminate this deficiency.

These modifications are done by (*i*) introducing additional program variables, (*ii*) slightly modifying the assertions to reflect the introduction of new variables, and (*iii*) adding appropriate statements to make the assertions hold. Thus, the modified program is guaranteed to be correct. We believe that this technique is applicable not just to this problem but also to the wider class of problems wherein files are processed sequentially or where the initial versions of the programs refer to quantities not readily available.

We make the above-mentioned three changes as follows. In the first change, we eliminate the assignment statement $m := m_n$. Note that the rest of the outer loop maintains $m = m_n$ invariant; so we could add this term to the loop invariant $P1$ and eliminate the assignment statement. An extra initialization statement, $m := 1$, would then become necessary to establish the new loop-invariant in the beginning. The program is still correct. See Solution 2.1 below.

In the second change, we modify the outer loop guard $n \neq N$. Since

$$(n \neq N) \equiv [f(m_n) \neq f(m_N)],$$

$$m_N = M, \text{ and}$$

$$m = m_n$$

hold before the outerloop, its guard can be replaced by $f(m) \neq f(M)$. This change does not affect any of the assertions.

In the third change, we modify the inner loop guard, $m \neq m_{n+1}$. For $m_n \leq m < m_{n+1}$,

$$(m \neq m_{n+1}) \equiv [f(m) = f(m_n)] .$$

Therefore, we can use $f(m) = f(m_n)$ as the guard in place of $m \neq m_{n+1}$, without disturbing anything else. We now replace $f(m_n)$ by a program variable F; the guard becomes $f(m) = F$. The meaning of the inner loop is kept unchanged by requiring $F = f(m_n)$ to be true before the inner loop. This requirement is trivially met by adding the assignment $F := f(m)$ at that point; notice that $m = m_n$ is true before the loop, as discussed above.

The above additions appear in bold print in the following program.

Solution 2.1

$n := 1$; $\mathbf{m} := 1$; $\{P1 \wedge \mathbf{m} = \mathbf{m_n}\}$
do f(m) $\neq$ **f(M)**$\rightarrow$
  sum $:= 0$; $\mathbf{F} :=$ **f(m)**; $\{P2 \wedge m = m_n \wedge \mathbf{F} = \mathbf{f(m_n)}\}$
  do f(m) $=$ **F**$\rightarrow$
    sum $:=$ *sum* $+ q(m)$; $m := m + 1\{P2\}$
  od; $\{P2 \wedge m = m_{n+1}\}$
  print (*sum*);
  $n := n + 1\{P1 \wedge \mathbf{m} = \mathbf{m_n}\}$
od $\{P1 \wedge \mathbf{m} = \mathbf{m_n} \wedge n = N\}$.

This program, still correct, does not explicitely use N or m_n; notice that $f(M)$ is the special value EOF. They are, however, an integral part of the proof and the assertions. The two assignment statements involving n could be removed from the program without affecting it. But we retain them, as the final value of n is of interest in printing the Nth line of the report.

3.3 Step 3

Solution 2.1 uses $f(m)$ and $q(m)$ as if they are available as arrays, but they are not. We eliminate their use in two steps and introduce file operations instead. This technique is useful not only in solving McDonald's problem, but in a wider class of problems that involve sequential file processing.

In Step 1, we replace $f(m)$ and $q(m)$ by the variables f and q, respectively. This necessitates asserting $f = f(m)$ and $q = q(m)$ before the statements where the substitution is made. Just as in the previous section, the assertions are made to hold by introducing appropriate assignment statements.

The two above assertions can become false only after a statement that modifies m. Therefore, we introduce the assignment statements $f := f(m)$ and $q := q(m)$ after each statement that modifies m. Solution 2.1 has only two such instances. The program after these modifications is as follows.

Solution 3.1

$n := 1; m := 0;$
$m := m + 1; f := f(m); q := q(m); \{P2 \wedge m = m_n \wedge A\}$
do $f \neq f(M) \rightarrow$
  $sum := 0; F := f; \{P2 \wedge m = m_n \wedge F = f(m_n) \wedge A\}$
  do $f = F \rightarrow$
    $sum := sum + q;$
    $m := m + 1;$
    $f := f(m);$
    $q := q(m)\{P2 \wedge A\}$
  od; $\{P2 \wedge m = m_n \wedge A\}$
  print (*sum*);
  $n := n + 1\{P1 \wedge m = m_n \wedge A\}$
od$\{P1 \wedge m = m_n \wedge n = N\}$,

where A is $f = f(m) \wedge q = q(m)$, and the initialization of m has been broken up into two statements in the beginning of the program.

We now introduce the file operations in this program: replace $m :=$ 0 by *openfile*; $m := n + 1$, $f := f(m)$; $q := q(m)$ by *read*(f, q); and $f(M)$ by EOF. Also, n equals N at the end of the program, so we can add the statement to print $n - 1$, the number of items transacted. The resulting program, without the assertions, is as follows.

Solution 3.2

$n := 1$; *openfile*; *read*(f, q);
do $f \neq$ EOF$\rightarrow$
  $sum := 0; F := f;$
do $f = F \rightarrow$
    $sum := sum + q;$
    read(f, q)
  od;
  print (*sum*);
  $n := n + 1$
od;
print $(n - 1)$.

This solution is identical to that obtained by the data structure design technique. It is considered to be a desired solution: its structure reflects the problem closely, it does not treat any card specially, it neatly handles all the groups one after the other, and it can be modified to add special processing at the beginning or at the end of a group.[3]

IV. SUMMARY

In this paper, we solved the McDonald's warehouse problem to show

how to effectively use Dijkstra's method to develop programs that process records in a file sequentially.

The solution was developed in three steps. We first assumed that the records of the file were available in an array, and developed the program disregarding the file operations. This program also used certain symbols whose values are not readily available. These symbols were removed in the next step by introducing additional program variables and modifying the program under correctness assertions so that the next step could be carried out. Finally, we replaced one initialization statement by *openfile*, and selected groups of statements by *readfile* or *writefile*. The example discussed in this paper involved only reading a file, but the same techniques apply when a file is written, too.

With a proper choice of invariants, the programs thus developed are comparable to those obtained by the data structure design, a technique that is considered to yield good programs for such problems.

V. ACKNOWLEDGMENTS

Thanks are due W. L. Bain, Jr. and N. Gehani, for many useful comments on an earlier draft of this paper, and G. D. Bergland, for bringing this problem to the author's attention.

REFERENCES

1. Dijkstra, E. W., *A Discipline of Programming*, Englewood Cliffs: Prentice-Hall, 1976.
2. Gries, D., "An Illustration of Current Ideas on the Derivation of Correct Proofs and Correct Programs," IEEE Trans. Software Engineering, *SE-2*, No. 4 (December 1976), pp. 238–44.
3. Bergland, G. D., "Structural Design Methodologies," 15th Annual Design Automation Conf., June 21, 1978, Las Vegas, Nevada, Design Automation Conf. Proc., June 1978.