A Unified Approach for Developing Efficient Algorithmic Programs

Xue Jinyun (薛锦云)

Computer Software Institute, Jiangxi Normal University, Nanchang 330027
Computer Engineering Department, Santa Clara University, Santa Clara, CA 95053, USA

Received March 18, 1996; revised October 11, 1996.

Abstract

A unified approach called partition-and-recur for developing efficient and correct algorithmic programs is presented. An algorithm (represented by recurrence and initiation) is separated from program, and special attention is paid to algorithm manipulation rather than program calculus. An algorithm is exactly a set of mathematical formulae. It is easier for formal derivation and proof. After getting efficient and correct algorithm, a trivial transformation is used to get a final program. The approach covers several known algorithm design techniques, e.g. dynamic programming, greedy, divide-and-conquer and enumeration, etc. The techniques of partition and recurrence are not new. Partition is a general approach for dealing with complicated objects and is typically used in divide-and-conquer approach. Recurrence is used in algorithm analysis, in developing loop invariants and dynamic programming approach. The main contribution is combining two techniques used in typical algorithm development into a unified and systematic approach to develop general efficient algorithmic programs and presenting a new representation of algorithm that is easier for understanding and demonstrating the correctness and ingenuity of algorithmic programs.

Keywords: Programming method, algorithm design method, correctness of algorithmic program, recurrence relation, loop invariant.

1 Introduction

Algorithmic program is an algorithm described with an implemented or abstract programming language. Developing correct and efficient algorithmic programs is a heart of computer science. The process of development consists of design and description of new algorithms and explanation and proof of existing algorithms. Due to the creativity involved, this remains to be one of the field's most challenging problems. A lot of researches have worked on it over the years, yielding many approaches to this area, including program calculus[1-4], program transformation[5] and program automation[6-8]. Because of the obstinacy of the problems, these approaches are far from practical use in developing algorithms[9-11]. There remain

This research was supported by the 863 Hi-Tech Programme and the National Natural Science Foundation of China.

many crucial unsolved problems. Every year, a large number of new algorithms occur in literature and many existing algorithms are described and explained in the new textbooks of algorithms and algorithm design. They still use traditional algorithm design methods, including dynamic programming, greedy, divide-and-conquer, etc. However, there are no effective standards or rules to guide algorithm designers to choose appropriate one from these methods. This has made great difficulty not only in designing algorithms manually, but also in some experiment systems of automated algorithmic program design. These systems can only generate some simple algorithmic program based on divide-and-conquer and exhaustive search[8]. The KIDS system developed by D.R. Smith got better results[6,7]. Smith investigated the properties of many algorithm design methods, e.g., dynamic programming, divide-and-conquer and global search, separately and built one subsystem for each method, then combined these subsystems into KIDS system. There is, however, no effective standard to have the system choose proper algorithm design method for given problem. A unified approach is needed for solving these problems. Manber answered this challenge[12,13]. He proposed a unified approach for algorithm design based on mathematical induction and claimed that his methodology covered many known algorithm design techniques and could explain algorithm in a way to make it easier to understand. According to Manber's approach, induction hypothesis or stronger induction is presented at the beginning of algorithm design. That is, assume $n-1$ size problem was solved, then solve n size problem based on the hypothesis. This is an effective beginning. Unfortunately, he did not formulate an easy way to solve the n size problem. He treated it as a fully creative work. Our approach given in this paper presents an easier way not only to solve the n size problem but also to give a new representation of algorithm to facilitate the derivation and proof of some algorithms. We do believe that designing algorithm is a creative activity. However, we also believe that the invention of scientific notations, methods and theory can convert *some* creative work into regular work. Our research follows this confidence. In fact, invention of some mathematical method has given us some heuristics. The creation of Analytic Geometry is a good example for converting some creative work into routine job.

Our approach for developing efficient and correct algorithm is called partition-and-recur. The key idea is partition and recurrence. Similar to other techniques of developing algorithm, our approach is also based on the powerful mathematical induction. Dijkstra[2] and Gries[14] presented a methodology to develop algorithmic program together with their proof of correctness hand in hand. We borrow their techniques of formal specification and quantifier transformation, but our emphasis is different. We separate algorithm, represented by recurrence and initiation, from program, then pay special attention to algorithm manipulation rather than program calculus. Using partition-and-recur, we begin from the formal specification of a problem, partition the problem into a couple of subproblems, then develop an efficient and correct algorithm represented by recurrence and initiation. Finally, corresponding program and its loop invariant are developed. Our approach covers several known algorithm design techniques, e.g., dynamic programming, greedy,

divide-and-conquer and enumeration. The techniques of partition and recurrence are not new. Partition is a general approach for dealing with complicated objects and is typically used in divide-and-conquer approach. Recurrence is used in algorithm analysis and dynamic programming approach. Our main contribution is combining two techniques used in typical algorithm problem into a unified and systematic approach to solve general algorithm problems and presenting a new representation of algorithm for understanding and demonstrating more easily the correctness and ingenuity of an algorithm.

In Section 2, we present the partition-and-recur approach with detailed explanation. A brief description of traditional algorithm design methods is given. In Section 3, two typical examples using partition-and-recur are developed. Originally, these problems were solved by two traditional approaches. A careful comparison with traditional algorithm design techniques is given in the end of Section 3. Comparison with other unified approaches is described in Section 4. Finally, conclusions and discussions are presented.

2 Partition-and-Recur Approach

2.1 Consideration of Efficiency

The efficiency of an algorithm is mainly influenced by the method of algorithm design and implementation, the data structures and the programming language describing the algorithm. In this paper we put emphasis on the method of algorithm design and implementation. In general, it is easier to design algorithms using enumeration or exhaustive search, but the algorithms have low efficiency; in contrast, it is rather difficult to design efficient algorithm using effective design method. Using traditional method, e.g. dynamic programming, greedy, divide-and-conquer, etc., one can get efficient algorithm, but the difficulty is in choosing the suitable one. We want to pursue a unified approach of developing algorithm that can cover these well-known algorithm techniques and is easier to use. Since implementing algorithm using iteration has higher efficiency than using recursion, we stick to iteration rather than recursion in our methodology.

2.2 Comments on Traditional Algorithm Design Methods

Although those traditional algorithm design methods often occur in various textbooks and literature of algorithm design and programming, there is no uniform and precise description. Here we cite Horowitz's descriptions in Ref.[15]. The *divide-and-conquer* strategy suggests that a problem should be split into subproblems, which are somehow solved and then the solutions are combined into a solution for the original problem. The subproblems are solved in the same way—by splitting, solving and combining—and this can often be done recursively. It is generally advisable to split the problems into subproblems with roughly equal size. *Dynamic programming* arises when the only algorithm we can think of is enumerating all possible configurations

of the given data and testing each one to see if it is a solution. An essential idea is to keep a table that contains all previously computed configurations and their results. If there are only a small number of distinct configurations, dynamic programming avoids recomputing the solution to these problems over and over. To determine if there are only a small number of distinct configurations, one needs to detect when the so-called *principle of optimality* holds. This principle asserts every decision that contributes to the final solution must be optimal with respect to the initial state. As for *greedy method*, Horowitz suggests it produces an algorithm that works in stages. At each stage, a decision is made regarding whether an input combination is feasible and is better or worse than the previous solution. At the end of final stage, the best solution is obtained. However, greedy algorithms do not always produce the best result. Horowitz's description stands for common viewpoints. Obviously, there are big differences among these methods. After carefully choosing among these methods, one can really design many efficient algorithms. However, there are no effective standards or rules to guide algorithm designers to choose appropriate one from these methods. This has made great difficulty in designing algorithm by hand or by computers. For example, students, even some computer scientists, get very confused about dynamic programming and greedy in dealing with optimal problems. A unified approach is needed for solving these problems.

2.3 Designing Efficient Algorithms

In Ref.[16], we generalize the recurrence relation concept of a sequence of numbers (difference equation) to problem solving sequence and have the following definitions.

Definition 1. *Computing step is one execution of a statement or a group of statements.*

Both an iteration of a loop and a recursive call of a recursive procedure can be considered as computing steps.

Definition 2. *Suppose that the solution of problem P can be obtained by n computing steps which produce the solution sequence: $S_1, S_2, \ldots, S_n$, where each S_i, $1 \leq i < n$, produced by one computing step is a subsolution of P, then S_n is the solution of P. Construct the equation: $S_i = F(\overline{S_j})$, which means subsolution S_i is the function of subsubsolution $\overline{S_j}$, where $1 < i \leq n$, $1 \leq j < i$, and $\overline{S_j}$ denotes several subsolutions S_j. We call the equation $S_i = F(\overline{S_j})$ as recurrence relation of problem solving sequence, or recurrence for short.*

The examples of recurrence relation of problem solving sequence are given in Section 3. Obviously, it is easy to get the final solution of a problem from its recurrence relation of problem solving sequence. Algorithm analysis theory and practical experience show that partition is a powerful strategy for dealing with complex objects. Computation by iteration can avoid recomputing the subsolution over and over. These two strategies give us some general hints to design efficient algorithms. Based on this knowledge for a given problem, the algorithm design can be broken into 3 steps:

Step 1. Construct the specification of a problem as formal as possible, that is to formulate

what the algorithm should do precisely.

Formal specification forces the designer of algorithm to think over the problem carefully and precisely, and it makes derivation and proof of recurrence possible. To construct formal specification, we can borrow some notations from Dijkstra[3], Knuth[17], Gries[18] and Backhouse[1]. However, as algorithm design is also a creative activity, it needs designer's knowledge, experience and intelligence. Some notations for special problem must be created by himself.

Step 2. Partition the problem into a number of subproblems, each of which has the same structure as the original problem but is smaller in size, then partition the subproblems into smaller subproblems until each subproblem can be solved directly. The subproblem that can be solved directly is called the *smallest subproblem.*

Obviously, the shape of recurrence or algorithm is decided by partition. Different partitions correspond to different algorithms. One can obtain some hints for partitioning a problem from the formal specification of the problem. In general, there are two strategies to partition a problem, balanceable and unbalanceable partitions. For most optimal problems, unbalanceable partition is proper.

Step 3. Construct the recurrence relation of problem solving sequence $S_i = F(\overline{S_j})$ and the initiation that initiates the value of variables and functions appearing in recurrences, then combine the initiation and all recurrences into an algorithm.

A good formal specification helps one to derive recurrences with quantifier transformation. An example presented in Section 3 demonstrates this technique.

Considering the initiation, there are two methods to produce the final solution to the problem. One is *recursion* which produces computing sequence according to *top-down* fashion, denoted by $S_i =\downarrow F(\overline{S_j})$. The other is *iteration* which produces computing sequence according to *bottom-up* fashion, denoted by $S_i =\uparrow F(\overline{S_j})$. Here, we get two algorithms represented by recurrence relation for solving the problem: recursive algorithm and iterative algorithm. The iterative algorithm has higher efficiency than recursive one since there is no overlapping computation in its computing sequence.

2.4 Developing Correct Program

The recursive program corresponding to recursive algorithm can be developed directly based on the recurrence relation. We just pay main attention to developing the program corresponding to iterative algorithm.

The key for developing correct iterative program is loop invariant. This is recognized by not only the advocators of formal method of designing algorithms and programs but also some specialists of algorithm design, for example, Kingston[19], Baase[9], and others. However, the existing standard strategies for developing loop invariants are only suitable for some simple problems. There are many complicated algorithms and programs that cannot get satisfying loop invariants using these techniques. This leads to that many computer scientists doubt the possibility of deriving or proving algorithms and programs using loop invariant. In Ref.[20], we exposed new properties of loop invariant and presented two new strategies for developing it. Following is one of the two strategies.

Strategy of developing loop invariants for new algorithmic programs: Based on

the recurrence relation of problem solving sequence, determine all needed variables in the program and describe the variation laws or the functions of each variable, the laws or the functions are required by loop invariants; if a sequence variable (sometimes a set variable) which will be used as a stack is needed, the content of the sequence variable is defined recursively.

This strategy is quite powerful, especially in using its recursive definition technique to develop the loop invariant of an iteration program with inherent recursive property. In Ref.[20] one can find an example using this technique. The development of a program can be broken into two steps:

Step 1. Develop loop invariant based on the new strategy.
Step 2. Transform the algorithm to program.

Remark. The algorithms and programs developed by above approach are abstract ones described in an abstract language that is similar to Dijkstra's guarded command language. To obtain executable program code, one must apply the techniques of functional refinement and data refinement to abstract programs. The techniques presented in Refs.[16,18,21] are especially recommended. For saving space, we do not state these popular techniques here.

3 Examples of Using Partition-and-Recur Approach

The following two classic examples are elaborately chosen from many algorithms we developed using partition-and-recur[22]. They respectively show that the problems that were originally solved by dynamic programming and greedy can be solved using partition-and-recur approach. Since divide-and-conquer approach is a special case of our approach, we do not include a divide-and-conquer example here. In the development of these algorithms of the problems, we concentrate on developing algorithms represented by recurrence relation. Based on this representation of algorithm, we can write loop invariants and program code of the algorithms in a fairly straightforward manner. We put special emphasis on first example that solves the problem of maximum product section. We will describe the details of each step of algorithm design according to our methodology. However for saving space, in Example 2, we keep the explicit steps and more details of algorithm design in mind and just give an outline of algorithm development using partition-and-recur.

Remark About Symbols and Notations. We use Dijkstra's guarded command language to describe programs. The variables are integer type without special declaration. $b(i : j-1)$ denotes all adjacent elements $b(k)$ in array $b(0 : n-1)$ and is called a *section* where $0 \le i \le k \le j \le n-1$; if $i = j$, $b(i : j-1)$ denotes *empty section*. We use a unified format $Q(i : r(i) : f(i))$ given by Dijkstra to denote quantifiers including extended ones, where Q can be $\forall$, $\exists$, $\sum$, $\prod$, MIN (minimum value of elements of a set) and MAX (maximum value of elements of a set), i is a bounded variable, $r(i)$ is the variant range of i and $f(i)$ is a function. $\max(a, b)$ and $\min(a, b)$ are two functions whose values are maximum and minimum of a and b respectively.

3.1 Maximum Product Section

3.1.1 Problem and Its Specification

Given an array $a(0:n-1)$ containing n integers, try to store the maximum product of all sections of a in p. That is:

$$p = \text{maxprod}(a(0:n-1))$$

where, $\text{maxprod}(a(0:n-1)) = \text{MAX}(r,t: 0 \le r \le t \le n : \text{prod}(r,t))$

$$\text{prod}(r,t) = \begin{cases} 1 & \text{if } r = t \\ \prod(i: r \le i \le t-1 : a(i)) & \text{if } r < t \end{cases}$$

According to this definition, $\text{prod}(r,t)$ is the product of array section $a(r:t-1)$; if $r = t$, $a(r:t-1)$ denotes an *empty section* and its product is 1. An obvious algorithm to solve this problem is enumerating the product of all sections of array a, then choosing the maximum, but the computation complexity is $O(n^3)$. We hope to develop a more efficient algorithm using partition-and-recur approach.

3.1.2 Partition

There are two methods to partition array a: balanceable and unbalanceable partitions. For this problem, since the maximum product of each section obtained by using balanceable partition cannot be combined into the solution of the original section, we choose unbalanceable partition. Therefore, we partition computing $\text{maxprod}(a(0:n-1))$ into computing $\text{maxprod}(a(0:n-2))$ with $a(n-1)$, then partition computing $\text{maxprod}(a(0:n-2))$ into computing $\text{maxprod}(a(0:n-3))$ with $a(n-2)$, ..., until computing $\text{maxprod}(a(0:0))$. Let F be the function to be determined, we have

$$\text{maxprod}(a(0:n-1)) = F(\text{maxprod}(a(0:n-2)), a(n-1))$$
$$\text{maxprod}(a(0:i-1)) = F(\text{maxprod}(a(0:i-2), a(i-1)) \qquad (1 \le i \le n)$$

So, the key of constructing recurrence relation is to determine function F.

3.1.3 Constructing Recurrence Relation

Suppose $\text{maxprod}(a(0:i-2))$ has been solved. We can derive the function F by using the properties of quantifiers[1,2,17]. We have

$$\begin{aligned}
&\text{maxprod}(a(0:i-1)) \\
&\quad = \text{MAX}(r,t: 0 \le r \le t \le i : \text{prod}(r,t)) \\
&\quad = \{\text{Cartesian Product}\} \\
&\qquad \text{MAX}(t: 0 \le t \le i : \text{MAX}(r: 0 \le r \le t : \text{prod}(r,t))) \\
&\quad = \{\text{Range Splitting and Singleton Range with } t = i\ \} \\
&\qquad \max(\text{MAX}(t: 0 \le t \le i-1 : \text{MAX}(r: 0 \le r \le t : \text{prod}(r,t))), \\
&\qquad\quad \text{MAX}(r: 0 \le r \le i : \text{prod}(r,i))) \\
&\quad = \{\text{Cartesian Product}\} \\
&\qquad \max(\text{MAX}(r,t: 0 \le r \le t \le i-1 : \text{prod}(r,t)), \text{MAX}(r: 0 \le r \le i : \text{prod}(r,i))) \\
&\quad = \max(\text{maxprod}(a(0:i-2)), \text{MAX}(r: 0 \le r \le i : \text{prod}(r,i)))
\end{aligned}$$

Let $ma(i) = \mathrm{MAX}(r : 0 \le r \le i : \mathrm{prod}(r, i))$, which denotes the maximum product of the section ending at $a(i-1)$ of array a. We have the following recurrence.

Recurrence 1.

$\mathrm{maxprod}(a(0 : i-1)) = \max(\mathrm{maxprod}(a(0 : i-2)), ma(i)) \qquad 1 \le i \le n.$

To compute $ma(i)$, one can, based on the definition of $ma(i)$, enumerate the product of all sections in $a(0 : i-1)$ ending at $a(i-1)$ and get an $O(i^2)$ algorithm. Using partition-and-recur, we hope to find a more efficient algorithm. Suppose $ma(i-1)$ has been computed. We try to find the recurrence relation for computing $ma(i)$. Based on the properties of quantifiers, we have

$$\begin{aligned} ma(i) &= \mathrm{MAX}(r : 0 \le r \le i : \mathrm{prod}(r, i)) \\ &= \max(\mathrm{MAX}(r : 0 \le r \le i-1 : \mathrm{prod}(r, i)), \mathrm{prod}(i, i)) \\ &= \max(\mathrm{MAX}(r : 0 \le r \le i-1 : \mathrm{prod}(r, i-1) * a(i-1)), 1) \\ &= \begin{cases} ma(i-1) * a(i-1) & \text{if } a(i-1) > 0 \\ 1 & \text{if } a(i-1) = 0 \\ -ma(i-1) * a(i-1) & \text{if } a(i-1) < 0 \end{cases} \end{aligned}$$

where, $-ma(i-1)$ is the minimum product of all sections ending at $a(i-2)$ of array $a(0 : i-2)$. The following is its formal definition.

$$mi(i) = \mathrm{MIN}(r : 0 \le r \le i : \mathrm{prod}(r, i))$$

Similarly, we have

$$\begin{aligned} mi(i) &= \min(\mathrm{MIN}(r : 0 \le r \le i-1 : \mathrm{prod}(r, i-1) * a(i-1)), 1) \\ &= \begin{cases} \min(mi(i-1) * a(i-1), 1) & \text{if } a(i-1) > 0 \\ 0 & \text{if } a(i-1) = 0 \\ ma(i-1) * a(i-1) & \text{if } a(i-1) < 0 \end{cases} \end{aligned}$$

Based on the above derivation, we have the following recurrence.

Recurrence 2.

$$ma(i) = \begin{cases} ma(i-1) * a(i-1) & \text{if } a(i-1) > 0 \\ 1 & \text{if } a(i-1) = 0 \\ \max(mi(i-1) * a(i-1), 1) & \text{if } a(i-1) < 0 \end{cases} \qquad 0 < i \le n$$

$$mi(i) = \begin{cases} \min(mi(i-1) * a(i-1), 1) & \text{if } a(i-1) > 0 \\ 0 & \text{if } a(i-1) = 0 \\ a(i-1) * a(i-1) & \text{if } a(i-1) < 0 \end{cases} \qquad 0 < i \le n$$

Obviously, according to the definition of $\mathrm{maxprod}(a(0 : i-1))$ and $\mathrm{prod}(r, t)$, if $i = 0$, then $r = t = i = 0$, $\mathrm{prod}(r, t) = 1$, $\mathrm{maxprod}(a(0 : i-1)) = 1$. Similarly, $ma(i) = mi(i) = 1$. Therefore we have the following.

Initiation 1. $\quad i = 1 \wedge \mathrm{maxprod}(a(0 : -1)) = 1 \wedge ma(0) = 1 \wedge mi(0) = 1$

Combining Initiation 1, Recurrences 1 and 2, we have:

ALGORITHM: maximum product section
BEGIN: $i = 1 \wedge \mathrm{maxprod}(a(0 : -1)) = 1 \wedge ma(0) = 1 \wedge mi(0) = 1$
RANGE: $1 \le i \le n$
RECUR: $\mathrm{maxprod}(a(0 : i-1)) = \max(\mathrm{maxprod}(a(0 : i-2)), ma(i))$;
ALGORITHM: $ma(i)$ and $mi(i)$
RECUR:

$$ma(i) = \begin{cases} ma(i-1) * a(i-1) & \text{if } a(i-1) > 0 \\ 1 & \text{if } a(i-1) = 0 \\ \max(mi(i-1) * a(i-1), 1) & \text{if } a(i-1) < 0 \end{cases}$$

$$mi(i) = \begin{cases} \min(mi(i-1) * a(i-1), 1) & \text{if } a(i-1) > 0 \\ 0 & \text{if } a(i-1) = 0 \\ ma(i-1) * a(i-1) & \text{if } a(i-1) < 0 \end{cases}$$

END

END

Based on this algorithm, let initial value of i be 1, one can compute maxprod($a(0 : i-1)$) step by step; finally when $i = n$, we get the result of maxprod($a(0 : n-1)$).

Remark on algorithm representation. The algorithm representation consists of five components headed by five key words: ALGORITHM, BEGIN, RANGE, RECUR, END. ALGORITHM component gives the name of the algorithm or the name of the function that will be computed by the algorithm. The recurrences are given between RECUR and END. BEGIN component gives the initiation of the recurrences, that is the initial values of the variables and functions contained in the recurrences. RANGE component gives the range of the variables appearing in the recurrences. An algorithm may contain subalgorithms. In above example, algorithms $ma(i)$ and $mi(i)$ are subalgorithms of the maximum product section algorithm. An algorithm and its subalgorithm may share the same BEGIN and RANGE components.

3.1.4 Developing Loop Invariant and Program

Observing the algorithm of maximum product problem, besides variable i, additional three variables p, x, y are needed to store the value of maxprod($a(0 : i-1)$), $ma(i)$, $mi(i)$ respectively. Now according to our strategies given in Ref.[20], we can write loop invariant I as follows in a fairly straightforward manner.

$$I: \quad p = \text{maxprod}(a(0 : i-1)) \wedge x = ma(i) \wedge y = mi(i) \wedge \quad 0 \leq i \leq n$$

Based on Initiation 1, I can be established by $p, x, y := 1, 1, 1$. The body of loop can be constructed mechanically based on Recurrences 1 and 2. Following is the program described in guarded command programming language for solving the maximum product section problem.

$i, p, x, y := 0, 1, 1, 1;$
do $i \neq n \rightarrow$ if $a(i) > 0 \rightarrow x, y := x * a(i), \min(y * a(i), 1);$
$\quad \square a(i) = 0 \rightarrow x, y := 1, 0;$
$\quad \square a(i) < 0 \rightarrow x, y := \max(y * a(i), 1), x * a(i)$
$\quad$ fi
$p := \max(p, x);$
$i := i + 1$
od

Readers can verify the correctness of this program based on Recurrences 1, 2, Initiation, loop invariant I and standard proof techniques in Ref.[2,4]. It is a simple and trivial task in this context.

Remark. This is not an easy problem. The problem is taken from A. Kaldewaij and its solution is given by M. Rem using the standard techniques of developing program and proof hand in hand. Our development of the algorithm has some similarity with theirs in using the techniques of quantifier transformation. However our emphasis is different. We distinguish an algorithm, represented by recurrence and initiation, from program, and pay special attention to algorithm manipulation rather than program calculus. The algorithm represented by recurrence relation is exactly a set of mathematical formulae. It is easier for formal proof and derivation. After getting correct algorithm, transforming it into correct program is a trivial work. Compared with Manber's methodology[12,13], ours is also based on powerful mathematical induction. So are the existing techniques of program proof and derivation. We do believe that designing algorithm is a creative activity. However, we also believe that the invention of scientific notations, methods and theory can convert *some* creative work into a routine job. In this example, we stick to partition-and-recur as well as formal specification of the problem, and the derivation of Recurrence 1 and Recurrence 2 is mainly done mechanically.

3.2 Minimum Spanning Tree Problem

Let $G = (V, E, W)$ be a connected, undirected graph with a real-valued weight function, where V is the set of nodes, E is the set of edges and W is the set of weights $W(u, v)$ of each edge (u, v) in E. A *spanning tree* of graph G is a tree that connects all nodes of the graph G, denoted by $ST(G)$. There is more than one $SP(G)$. Let $\{SP(G)\}$ denote the set of $SP(G)$, where there is at least one $SP(G)$ whose sum of weights of all edges is as small as possible. This $SP(G)$ is called the *minimum spanning tree of graph* G, denoted by $MST(G)$. The problem is how to construct an $MST(G)$ for a given graph G.

According to partition-and-recur approach, we assume $T.n$, $0 \leq n \leq \mid V \mid -1$, is a partial $MST(G)$. It means $T.n$ is a set of n edges and all edges of $T.n$ belong to an $MST(G)$. If $n = \mid V \mid -1$, then $T.n$ is a full $MST(G)$. We need to extend $T.n$ with more edges one by one. The problem is what kind of edge can be added to $T.n$, i.e., finding a function F satisfying

$$T.n + 1 = F(T.n)$$

Since the final product is a tree with minimum weight, the edge we should find must satisfy two constraints:

1. two nodes of the edge are not in the same tree;
2. the cost of the edge is as small as possible.

These give us some heuristic to produce a partition for graph G. Let P be a set of nodes in G and divide nodes of G into two parts: P and $V - P$, with no edge in

$T.n$ whose one node is in P and the other in $V-P$. We call this division *a partition*, denoted by $(P, V-P)$.

An edge (u,v) satisfying $u \in P$ and $v \in V-P$ is called a *crossing edge* and a crossing edge (u,v) is called *minedge* if its weight is the minimum among all crossing edges. Let cross$(P, V-P)$ denote all crossing edges on partition $(P, V-P)$. Suppose M is an $MST(G)$ that includes $T.n$, then there is at least one edge in M that is a minedge in cross$(P, V-P)$. This edge is what we are looking for. Based on this observation, we have the following recurrence:

Recurrence 3. $T.n+1 = T.n \cup \{\text{minedge}(P, V-P)\} \quad 0 \leq n \leq \mid V \mid -2$

The function minedge$(P, V-P)$ produces a minedge on partition $(P, V-P)$. The reader can find a proof of Recurrence 3 in many textbooks, such as in Ref.[23].

Based on Recurrence 3, we can derive two well-known algorithms. Let $T.n$ be the connected component of partial $MST(G)$ and $D.n$ be a set of all nodes in $T.n$, then we can make partition $(D.n, V-D.n)$. Therefore we have another recurrence:

Recurrence 4. $T.n+1 = T.n(\{\text{minedge}(D.n, V-D.n)\} \quad 0 \leq n \leq \mid V \mid -2$

This recurrence gives us an easy way to construct an $MST(G)$, specially in choosing minedge. Since $D.n$ contains all nodes in $T.n$, therefore cross$(D.n, V-D.n)$ includes all edges whose one node is in $D.n$ and the other is in $V-D.n$. Minedge can be chosen in this domain. The key is how to make the partition $(D.n, V-D.n)$. Assume edge (u,v) is the minedge on partition $(D.n-1, V-D.n-1)$, where $u \in D.n-1$ and $v \in V-D.n-1$. The partition $(D.n, V-D.n)$ can be derived from the following recurrence:

Recurrence 5.

$(D.n, V-D.n) = (D.n-1 \cup \{v\}, V-D.n-1-\{v\}) \quad 0 \leq n \leq \mid V \mid -2$

We make the following initiation:

Initiation 2. $n = 0 \wedge T.0 = \emptyset \wedge D.0 =$ any node x in V.

Beginning at Initiation 2, using Recurrences 4 and 5, we can compute $T.n$ step by step, finally get $T. \mid V \mid -1$ that contains $MST(G)$. Obviously, Initiation 2, Recurrence 4 and Recurrence 5 form an efficient algorithm. That is the well-known PRIM algorithm.

```
ALGORITHM: PRIM
BEGIN: n = 0 ∧ T.0 = ∅ ∧ D.0 = any node x in V
RANGE: 0 ≤ n ≤| V | −2
RECUR: T.n + 1 = T.n ∪ {minedge(D.n, V − D.n)}
      ALGORITHM: minedge(D.n, V − D.n)
      RECUR: choose((D.n, V − D.n), (u, v));
            (D.n + 1, V − D.n + 1) = (D.n ∪ {v}, V − D.n − {v});
      END
END
```

where Procedure choose$((D.n, V-D.n), (u,v))$ has the following meaning: (u,v) contains the edge that has the minimum weight in cross$(D.n, V-D.n)$.

Suppose all nodes of graph G are numbered by 1, 2, ..., $\mid V \mid -1$. Considering Recurrences 4 and 5, let $T.n$ be stored in set variable T. We need another variable

to store the weight of each edge in cross$(D.n, V - D.n)$. However, a wise idea is that for each node v in $V - D.n$, let element $d(v)$ of array $a(1 : |v| - 1)$ contain the minimum weight of any edge connecting v to a node in $T.n$. Based on PRIM algorithm described above, we can write the precondition Q, postcondition R, loop invariant and its program as follows:

Program of PRIM Algorithm
$\{Q : G = (V, E, W) \wedge A = \{adj(v) \mid v \in V \wedge adj(v) \text{ is a set of nodes adjacent to } v\}\}$
$\{R : T = MST(G)\}$
$\{I : T = T.n \wedge \forall(j : j \in V - D : d(j) = \text{MIN}(k : k \in D : w(j,k))) \wedge 1 \leq n \leq | V | -1\}$

for each $v \in V$ **do** $d(v) := \infty$ **od**;
$T, u := \emptyset$, any node in V;
  $D := \{u\}$;
do $D \neq V \rightarrow$ **for** each v in $V - D$ **do** **if** $v \in adj(u) \wedge w(u,v) < d(v)$
  then $d(v) := w(u,v); p(v) := u$ **od**;
  choosemin$(u, V - D, d)$;
  $D, T := D \cup \{u\}, T \cup \{(u, p(u))\}$
od

where Procedure choosemin$(u, V - D, d)$ has the following meaning: choose node x, $x \in V - D$, make $d(x)$ as small as possible, then store x in u. The first two lines in the loop body implement Procedure choose$((D.n, V - D.n), (u, v))$.

Based directly on Recurrence 3, we can derive Kruskal algorithm using similar approach.

Final Remark. Only using partition-and-recur, two typical problems are solved. On cursory examination, one may think partition-and-recur approach is the same as the well-known divide-and-conquer. We believe that there is some similarity between two approaches, however the differences are crucial. As described by Horowitz in Academic Encyclopedia Computer Science(1993)[15], divide-and-conquer approach always suggests splitting the problems into roughly equal size problems. Therefore only a small number of problems could be solved by this approach. In literature and textbooks of algorithm design, these problems were arranged in a small chapter. However partition-and-recur can be used to solve general problems which originally were solved by dynamic programming, greedy, enumeration, as well as divide-and-conquer. The examples presented in this section are good evidence. The maximum product section originally was solved by dynamic programming, Minimum spanning tree is a typical example solved by greedy approach. Readers can find an example[20], preorder binary tree problem, of using partition-and-recur to solve divide-and-conquer problem. Fairly to say, divide-and-conquer is a special case of partition-and-recur.

4 Comparison with Related Work

The partition-and-recur approach described above is a unified and systematic approach for design algorithmic programs. There are some approaches similar with ours.

Manber's approach. We have given our comments on Manber's approach in Section 1. The maximum product section algorithm is an example that follows our viewpoint. This shows some promise to develop a software system that can finish this routine work automatically. The similarity with Manber's approach is that our approach is also based on powerful mathematical induction. In fact, so are the existing techniques of proof and derivation of algorithm and program.

Program calculus. This approach was created by Dijkstra[2,3], and developed into practical techniques by Gries[4], Backhouse[1], and others. The approach put special emphasis on developing program rather than designing algorithm. One must develop a program hand-in-hand with its loop invariant and proof of correctness. Our development of algorithms has some similarity with theirs. We borrow some of their techniques of formal specification and quantifier transformation. However our emphasis is different: we separate algorithm, represented by recurrence, from program, then pay special attention to algorithm manipulation rather than program calculus. The algorithm represented by recurrence relation is exactly a set of mathematical formulae and has mathematical transparency. It is easy for formal proof and derivation on existing theoretical foundation. We believe that, after getting correct algorithm represented by recurrence, transforming it into correct program is a trivial work. Most of the work can be done mechanically. In contrast with program calculus, one cannot derive program statements directly. One can only manipulate logic formulae, i.e., semantics of program, extracted from the program. There are some complicated transformation processes from logic to program or from program to logic during derivation and proof of program.

Program transformation. The approach is founded by Strong, Burstall, Darlington, Bird, Cohen, *et al.* Paull's book[5] summarized the main results in this area. To develop an algorithm, one must define a recursive function by enumerating all possible cases. This recursive definition does not imply an efficient algorithm. For increasing the efficiency, one must remove the recursion by some transformation rules. In many cases, this is a difficult task. In contrast with our approach, the algorithm represented by recurrences and initiation has the same shape as their recursive definition, but the difference is crucial. Our recurrence relation represents an efficient algorithm rather than a general recursive definition. Our development of minimum spanning tree algorithm is simpler and more convincing than Paull's.

5 Conclusions and Discussions

In summary, developing efficient and correct algorithm using partition-and-recur can be broken into a sequence of 5 steps:

1. Develop the formal specification of the problem;

2. Partition the problem into a number of subproblems each of which has the same structure as the original problem, but is smaller in size; continue this procedure until reaching the smallest problem;

3. Develop an algorithm represented by recurrences and initiation;

4. Write loop invariant directly based on our new strategy in Ref.[20];
5. Transform the algorithm to program.

As claimed by Manber (Ref.[13], p.4), *a common criticism of almost any methodology is that, although it may present an interesting way to explain things that were already created, it is of no help in creating them. This is a valid criticism, since only the future will tell how effective a certain methodology is and how widely used it becomes.* Currently, our methodology is also mainly used in explaining, deriving and proving existing algorithms[22,24,25], including path algorithms, travel tree algorithms, Knapsack algorithm, schedule algorithms, sorting algorithms, array section algorithms and some numeric algorithms. A convincing example is the development of Knuth's challenging program, using partition-and-recur, that converts a binary fraction to decimal fraction with certain conditions[26,27]. We have also taught this approach in graduate level courses[22]. The benefit of having such a general methodology is twofold:

- Partition-and-recur is a unified and systematic approach for designing algorithms and programs. It covers several existing algorithm design techniques, including divide-and-conquer, dynamic programming, greedy, enumeration and some nameless methods. It can partly avoid the difficulty in making choice among various existing algorithm design methods. Algorithm design is a creative activity. It needs designer's knowledge, experience and intelligence. So does our approach. Partition-and-recur guides algorithm designer to follow an effective way to find an efficient solution.
- We get a new representation of algorithm, mainly a set of recurrences and initiations. That is exactly a set of mathematical formulae and is easy for formal proof and derivation. It characterizes the main idea of an efficient algorithm and is more precise and simple than the representation of algorithm in natural language, flowchart and program. Its particular merit is easiness of understanding and demonstrating the ingenuity and correctness of an algorithm. This also shows us some promise to use partition-and-recur in some automatic or semiautomatic algorithm and program development system.

It should be pointed out that this is not a finished research project. As described above, a number of algorithmic programs were developed by only using partition-and-recur. Originally, several existing design methods were needed to explain them. The obtained result is encouraging. However it is still not easy to determine the exact range to which our approach can be applied. This situation is the same as several existing approaches, e.g., program calculus, program transformation, Manber's approach and even some well-known algorithm design methods. We need to develop more algorithmic programs for accumulating experiences in using our approach, then to determine how widely it can be used. In constructing problem specification, more suitable notations are needed. To pursue higher efficiency of an algorithm, we need more heuristics for partitioning a problem. For constructing and manipulating recurrences, we need more mathematical tools. We are studying these problems continually.

Acknowledgment This paper is a long term research result. I should give my thanks to many friends. My warm thanks go firstly to Prof. David Gries who encouraged and guided my interests in this field during my visiting in Cornell several years ago. I would also like to thank the 863 Hi-Tech Programme and NSF of China for their continuing support

on this research. I have benefitted from many comments of my colleagues and students. My special thanks go to Dr. Ruth E. Davis, Dr. Fuyau Lin and Dr. Daniel W. Lewis at Santa Clara University where I visited for providing me with helpful comments and good working environment that improved significantly the early version of this paper.

References

[1] Backhouse R C. Program Construction and Verification. Prentice-Hall, London, 1986.

[2] Dijkstra E W. A Discipline of Programming. Prentice-Hall, New Jersey, 1976.

[3] Dijkstra E W, Scholten C S. Predicate Calculus and Program Semantics. Springer-Verlag, New York, 1990.

[4] Gries D. The Science of Programming. Springer-Verlag, New York, 1981.

[5] Paull M C. Algorithm Design—A Recursion Transformation Framework. John Wiley & Sons, 1988.

[6] Smith D R. KIDS: A semiautomatic program development system. *IEEE Trans. on Software Engineering*, 1990, 16(9): 1024–1043.

[7] Smith D R (ed). Structure and Design of Problem Reduction Generators, Constructioning Program from Specification. IFIP, 1991, pp.1320–1328.

[8] Xu Jiafu. The Automation of Software. Qinghua University Press, 1993.

[9] Baase S. Computer Algorithms, Introduction to Design and Analysis (2nd edition). Addison-Wesley, 1991, pp.15–17.

[10] Beery D M. Whether formal methods? Some thoughts on the application of formal methods. In *Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Software Evolution*, Sept. 1994, pp. 22–37.

[11] Luqi. Monterey workshop 94: Software evolution. In *Proc. of Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Software Evolution*, Sept. 1994, pp. 1–9.

[12] Manber U. Using induction to design algorithms. *CACM*, 1988, 31(11): 1300–1313.

[13] Manber U. Introduction to Algorithms—A Creative Approach. Addison-Wesley, 1989.

[14] Gries D. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 1982, 2: 207–214.

[15] Horowitz E. Design and classification of algorithms. In *Academic Encyclopedia Computer Science*, Third Edition, Ralston A, Reilly E D (eds.), Van Nostrand, Reinhold, New York, 1993, pp.33–37.

[16] Xue Jinyun, David G. Developing a linear algorithm for cubing a cycle permutation. *Science of Computer Programming*, 1988, 11: 161–165.

[17] Knuth D. The Art of Computer Programming, Vol.1. Addison-Wesley, 1973.

[18] Gries D, Volpano D. The transform—A new language construct. *Structure Programming*, 1991, 11: 1–10.

[19] Kingston J H. Algorithms and Data Structures, Design, Correctness, Analysis. Addison-Wesley, 1990.

[20] Xue Jinyun. Two new strategies for developing loop invariants and their applications. *Journal of Computer Science and Technology*, 1993, 8(3).

[21] Gries D, Xue Jinyun. The Hopcroft-Tarjan Planarity Algorithm: Presentations and Improvements. TR88-906, CS Dept. of Cornell University.

[22] Xue Jinyun. Design and Proof of Algorithm and Programs. Textbook used in Jiangxi Normal University, 1994.

[23] Aho A, Hopcroft J, Ullman J. Data Structures and Algorithms. Addison-Wesley, 1983.

[24] Xue Jinyun. A systematic method for designing and proving algorithmic programs. In *Advances of Theoretical Computer Science, Proc. of National Theoretical Computer Science Conf. of China*, 1994.

[25] Xue Jinyun. The standard proof and formal derivation of complex algorithmic programs. In *The 8th Academic Conf. of China Computer Federation*, Beijing, 1992, pp. 61–68.

[26] Knuth D. A Simple Program Whose Proof Isn't, Beauty is Our Business, A Birthday Salute to E.W. Dijkstra. Feijen W H J *et al* (eds), 1990.

[27] Xue Jinyun, Davis R, Lin F. A simple program whose derivation and proof are also. (to appear in *Software Concepts and Tools*)

Xue Jinyun graduated from Department of Mathematics of Nanjing University in 1970. From Dec. 1985 to April 1988, he worked as a visiting scholar at Cornell University. After June 1995, he spent 10 months as a visiting scholar at Santa Clara University. He is a Professor and the Director of Computer Software Institute at Jiangxi Normal University. His research interests include Cover's theory of algorithms and programs, software engineering and computer-aided instruction.