Certification Trails and Software Design for Testability

Gregory F. Sullivan[1]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Dwight S. Wilson[2]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Gerald M. Masson[3]
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Abstract

This paper investigates design techniques which may be applied to make program testing easier. We present methods for modifying a program to generate additional data which we refer to as a certification trail. This additional data is designed to allow the program output to be checked more quickly and effectively. Certification trails [14, 16] have heretofore been described primarily from a theoretical perspective. In this paper, we report on a comprehensive attempt to assess experimentally the performance and overall value of the certification trail method. The method has been applied to nine fundamental, well-known algorithms for the following problems: convex hull, sorting, huffman tree, shortest path, closest pair, line segment intersection, longest increasing subsequence, skyline, and voronoi diagram. Run-time performance data for each of these problems is given, and selected problems are described in more detail. Our results indicate that there are many cases in which certification trails allow for significantly faster overall program execution time than a 2-version programming approach, and also give further evidence of the breadth of applicability of this method.

Keywords: Software design for testability, software fault detection, certification trails, error monitoring, design diversity, data structures.

1 Introduction

We have examined a wide variety of fundamental algorithms to determine how they can be redesigned to allow for easier testability. To make the problem of testing the correctness of the output of a program more tractable we have found it is desirable to modify the program so that it generates additional data which we refer to as a *certification trail*. This additional data is designed to allow the program output to be checked more quickly and effectively. Our previous work on certification trails emphasized a theoretical perspective in which we proved that the asymptotic time complexity of the testing process could be reduced [14, 16]. In this paper, we report on implementations of the certification trail method so as to assess experimentally with run-time data the performance and overall value of the technique. We have implemented the certification trail method for nine fundamental and well-known algorithms of broad importance and applicability. For each algorithm, we have produced three implementations: a version which produces the output; a version which produces the output and generates a certification trail; and a version which checks the output while utilizing the certification trail. Specifically, algorithms for the following problems are analyzed: huffman tree, shortest path, sorting, closest pair, line segment intersection, convex hull, longest increasing subsequence, skyline, and voronoi diagram. The scope of the algorithms considered gives credibility to the overall applicability of the certification trail method. Furthermore, comparisons of run-time data for each of the three versions of each of the algorithms considered reveal many cases in which an approach using certification trails allows for significantly faster overall program execution time than a 2-version programming approach.

[1] Research partially supported by NSF Grants CCR-8910569 and CCR-8908092 and an IBM Technology Interchange Program Grant.

[2] Research partially supported by NSF Grant CCR-8910569 and an IBM Technology Interchange Program Grant.

[3] Research partially supported by NASA Grant NSG 1442 and an IBM Technology Interchange Program Grant.

2 Introduction to Certification Trails

First, let us consider a basic method which is used to perform testing to detect software faults called N-version programming [1, 2]. This method utilizes N teams of programmers, each independently implementing separate programs based on a problem specification. The programs are executed on the same input and the outputs are compared. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors. Thus the technique exploits design diversity. Also, note that the method can detect hardware faults which affect the separate executions in distinct ways causing distinct outputs. It is particularly valuable for detecting errors caused by transient fault phenomena. The N-version programming method can be used to detect faults af-

INTERNATIONAL TEST CONFERENCE 1993
0-7803-1429-8/93 $3.00 © 1993 IEEE

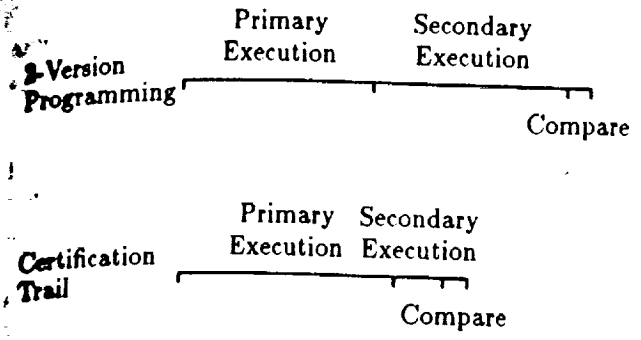

Figure 1: Timeline Comparison of the Certification Trail with 2-Version Programming

ter a system has been put into production or it can be used to detect faults in a testing phase prior to production. If two teams are used then we refer to the method as 2-version programming.

The certification-trail technique is designed to provide similar capabilities for detecting software and hardware faults as 2-version programming but expend fewer resources. As mentioned above the central idea is to modify the first algorithm so that, with modest additional overhead, it leaves behind a trail of data which we call a certification trail. This data is chosen so that it can allow the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. An illustration of typical execution times of 2-version programming versus the certification trail method is given in Figure 1. We assume that the two implementations developed for 2-version programming have approximately equal execution times. Note, however, that we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two program executions. For example, suppose the first program execution contains an error which causes an incorrect output and an incorrect trail of data to be generated. Further suppose that no error occurs during the execution of the second program. It still appears possible that the execution of the second program might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first program. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions we give below exclude this possibility. They demand that the second execution either generate a correct answer or signal that an error has been detected.

3 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

Definition 3.1 A problem $\mathbf{P}$ is formalized as a relation, i.e., a set of ordered pairs. Let $\mathbf{D}$ be the domain (that is, the set of inputs) of the relation $\mathbf{P}$ and let $\mathbf{S}$ be the range (that is, the set of possible solutions). We say an algorithm $\mathbf{A}$ solves a problem $\mathbf{P}$ iff for all $d \in \mathbf{D}$ when d is input to $\mathbf{A}$ then an $s \in \mathbf{S}$ is output such that $(d, s) \in \mathbf{P}$.

Definition 3.2 Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be a problem. A solution to this problem using a *certification trail* consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ and $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$. $\mathbf{T}$ is the set of *certification trails*. The functions must satisfy the following two properties:

(1) for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and $t \in \mathbf{T}$ such that $F_1(d) = (s, t)$ and $F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$
(2) for all $d \in \mathbf{D}$ and all $t \in \mathbf{T}$ either $(F_2(d, t) = s$ and $(d, s) \in \mathbf{P})$ or $F_2(d, t) = \text{error}$.

We also require that F_1 and F_2 be implemented so that they map elements which are not in their respective domains to the error symbol. Intuitively, the first condition states that if both parts of our solution execute correctly, then their answers agree and are correct. The second condition states that a correct secondary execution will never produce an incorrect output, i.e., one that is not a solution to the problem.

The definitions above assure that the testing capability of the certification-trail approach is similar to that obtained with a 2-version programming approach discussed earlier. That is, if a software or hardware fault occurs during only one of the executions then either the fault will be detected or the output will be a correct solution to the problem. The examples in this paper will indicate that this new approach can save overall execution time.

4 Certification Trail Examples

In the remainder of this paper we evaluate the use of certification trails for nine classic problems in computer science. We have implemented algorithms for these problems together with other algorithms which generate and use certification trails. In addition, we

discuss a general technique for construction of certification trails for algorithms using a wide range of data structures. This technique is used to implement the certification trails for several of our examples.

We provide a full description of the algorithm for the convex hull problem which generates a certification trail and a full description of the algorithm which uses that trail. Because of space considerations the discussion of the other algorithms is abbreviated. In some cases references to previous publications or technical reports which describe the algorithms more fully are given.

The algorithms we have chosen to implement are not always the algorithms which have the smallest asymptotic time complexity. Often the asymptotically fastest algorithms have large constants of proportionality which make them slower on the data sizes we examined. We modified and used some programs from major software distributions such as quicker-sort from a Berkeley Unix distribution. Fortune's algorithm for computing the Voronoi diagram was obtained from an Internet site at AT&T Bell Labs. Other algorithms were based on textbook discussions. It should be stressed here that this research is continuing as we further increase our corpus of algorithm and data-structure implementations.

4.1 Explanation of timing data

We have collected timing data for the algorithms on a Sun SPARCstation ELC with 16MB of RAM. The system was run as a standalone machine in single user mode during the timing experiments. Timing data was obtained through the getrusage() system call. The user times are reported in the data.

Much of the data presented in the timing table is essentially self-explanatory relative to the certification trail technique and algorithms considered. However, a brief discussion of the table entries is appropriate.

The column labelled *Basic* contains timing data which gives the execution time of the algorithm in producing the output without the generation of the certification trail. All timing data is listed in seconds.

The *Primary Execution (Prim. Exec.)* column gives the execution time of the algorithm in producing the output with the additional overhead of generating the certification trail.

The *Secondary Execution (Sec. Exec.)* column gives the execution time of the algorithm in producing the output while using the certification trail.

The *Percent Savings (% Sav.)* column records the percentage of the execution time savings which is gained by using the certification trail method as compared to 2-version programming approach. This assumes that both versions take approximately the same amount of time to execute.

The *Speedup* column is the ratio of the run times of the Basic Algorithm and the Secondary Execution.

For the Huffman tree data, the input size for the Huffman tree program is the number of nodes. Each node is given a frequency, chosen uniformly from the integers $\{1, 2, \ldots, n\}$. n was also selected to be the number of nodes.

For the shortest path table, there are two numbers associated with the input size, the first is the number of vertices in the graph, the second the number of edges. A graph with the required edges is selected uniformly from the set of all such graphs, then tested for connectedness in order to assure that paths exist to all vertices.

For the geometric algorithms, the input size is the number of points (or lines) in the original data set. Point set input was generated by choosing points with integer coordinates uniformly over a large square (typically 1,000,000 by 1,000,000 or larger square). For the Line Segment Intersection problem, lines were generated by picking a line segment start point uniformly from a large square and picking offsets for x and y-coordinates from a smaller range to give the end point of the line segment. This was done to bound the line length and avoid data sets resulting in a quadratic number of intersections.

Data for the longest increasing subsequence problem was produced by generating a random permutation of $[1..N]$ for input size N.

Sorting was performed on an array of pointers to structures. It was assumed that each structure contains an extra integer field for use in generating the certification trail. Sorting was performed on integer keys, though the technique can be used with a more complex key (in fact, using complex keys is very likely to increase the speedup achieved). Integers were chosen uniformly from interval $[1..1,000,000,000]$.

4.2 Convex Hull Example

The convex hull problem is fundamental in the field of computational geometry. Our certification trail solution is based on a convex hull algorithm due to Graham [6] called Graham's Scan. For basic definitions in computational geometry see the text of Preparata and Shamos[11]. For simplicity in the discussion which follows we will assume the points are in general position, e.g., no three points are collinear. It is not hard to remove this restriction.

Definition 4.1 The *convex hull* of a set of points, T, in the Euclidean plane is defined as the smallest convex polygon enclosing all the points. This polygon is unique

and its vertices are a subset of the points in T. It is specified by a counterclockwise sequence of its vertices.

The algorithm given below constructs the convex hull incrementally in a counterclockwise fashion. The first step of the algorithm selects an "extreme" point and calls it p_1. The next two steps sort the remaining points. The order of the points is determined by the slopes of the line segments formed by joining each point to p_1. It is not hard to show that after these three steps the points when taken in order, $p_1, p_2, \ldots, p_n$, form a simple polygon; although this polygon may not be convex. The Graham Scan algorithm traverses this polygon, removing points until the resulting polygon is convex. The main FOR loop iteration adds vertices to the polygon under construction and the inner WHILE loop removes vertices from the construction. A point is removed when the angle test performed at line 6 reveals that the angle at that vertex is obtuse. It is easy to demonstrate that when a point is removed, it must fall within the triangle defined by three other points, p_1 and the two points that were adjacent to the point removed. When the main FOR loop is complete the convex hull has been constructed. The execution of this algorithm is demonstrated in Figure 2. For each removed point, the associated triangle is indicated in bold lines, and in the text below the diagram. Our certification trail relies on the fact that that these triangles can be determined quickly.

Algorithm CONVEXHULL(T)
Input: Set of points, T, in R^2
Output: Counterclockwise sequence of points in R^2 which define the convex hull of T
1 Let p_1 be the point with the largest x coordinate (and smallest y to break ties)
2 For each point p (except p_1) calculate the slope of the line through p_1 and p
3 Sort the points (except p_1) from smallest slope to largest. Call them $p_2, \ldots, p_n$
4 $q_1 := p_1$; $q_2 := p_2$; $q_3 := p_3$; $m = 3$
5 FOR $k = 4$ to n DO
6 WHILE the angle formed by q_{m-1}, q_m, p_k is ≥ 180 degrees DO $m := m - 1$ END
7 $m := m + 1$
8 $q_m := p_k$
9 END FOR
10 FOR $i = 1$ to m DO, OUTPUT(q_i) END FOR
END CONVEXHULL

First execution: In this execution the code CONVEXHULL is used. The certification trial is generated

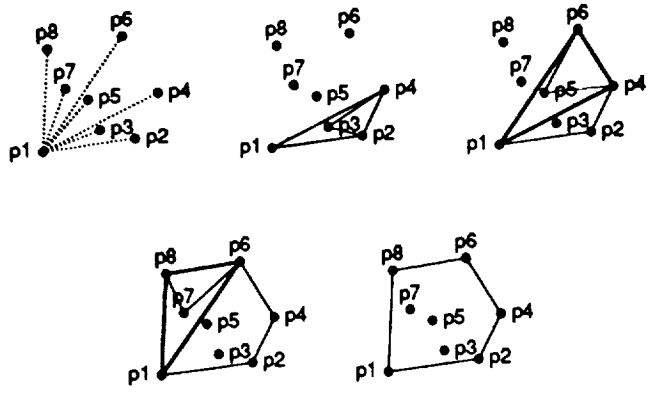

Figure 2: Convex hull example.

Point not on convex hull	Three surrounding points
p_3	p_1, p_2, p_4
p_5	p_1, p_4, p_6
p_7	p_1, p_6, p_8

by adding an output statement within the WHILE loop. Specifically, if an angle of less than 180 degrees is found in the WHILE loop test then the four tuple consisting of q_m, q_{m-1}, p_1, p_k is output to the certification trail. The final convex hull points $q_1, \ldots, q_m$ are also output to the certification trail. Strictly speaking the trail output does not consist of the actual points in R^2. Instead, it consists of indices to the original input data. This means if the original data consists of $p_1, p_2, \ldots, p_n$ then rather than output the element in R^2 corresponding to p_i the number i is output.

Second execution: Let the certification trail consist of a set of four tuples, $(x_1, a_1, b_1, c_1), (x_2, a_2, b_2, c_2), \ldots, (x_r, a_r, b_r, c_r)$ followed by the supposed convex hull, $q_1, q_2, \ldots, q_m$. The code for CONVEXHULL is not used in this execution. Indeed, the algorithm is dramatically different than CONVEXHULL.

It consists of five checks on the trail data.

- First, it checks that there is a one to one correspondence between the input points and the points in $\{x_1, \ldots, x_r\} \cup \{q_1, \ldots, q_m\}$.
- Second, it checks that for each $i \in \{1, \ldots, r\}$, a_i, b_i, and c_i are among the input points.
- Third, the algorithm checks that for each $i \in \{1, \ldots, r\}$, x_i lies within the triangle defined by a_i,b_i, and c_i.

- Fourth, the algorithm checks that for each triple of counterclockwise consecutive points on the supposed convex hull, the angle formed by the points is less than or equal to 180 degrees.

- Fifth, it checks that there is a unique point among the points on the supposed convex hull which is a local maxima. We say a point q on the hull is a *local maxima* if its predecessor in the counterclockwise ordering has a strictly smaller y coordinate and its successor in the ordering has a smaller or equal y coordinate.

If any of these checks fail then execution halts and "error" is output. Otherwise the convex hull read from the trail is output. As mentioned above, the trail data actually consists of indices into the input data. This does not unduly complicate the checks above; instead it makes them easier. The correctness and adequacy of these checks must be proven. A complete formal proof is beyond the scope of this paper, instead a brief outline of the proof will be given.

Using our formal definition of certification trails, let $\mathbf{D}$ be the set of all finite planar point sets T. Let $\mathbf{S}$ be the set of convex polygons, with vertices in counterclockwise order (the restriction to counterclockwise ordering makes the convex hull unique). Then the problem we are considering is $HULL : \mathbf{D} \rightarrow \mathbf{S}$ where $HULL(T)$ is the polygon in $\mathbf{S}$ that forms the convex hull of T.

The description of the algorithms above defines functions F_1 and F_2. We must show that both conditions of Definition 3.2 hold. The following two lemmas, which we state without proof, are required.

Lemma 4.2 *Let P be a polygon on n points $p_1, p_2, \ldots, p_n$. P is a convex polygon iff P is simple and each angle $p_i p_j p_k$ is less than or equal to 180 degrees, where i is in $1, 2, \ldots n$, $j = (i+1) \bmod n$, and $k = (i+2) \bmod n$.*

Lemma 4.3 *If P is a non-simple polygon, then either P has more than one local maxima, or the interior angle at some vertex is greater than 180 degrees.*

These are deceptively simple statements. Though they are intuitively obvious, a formal proof is difficult. It is interesting to note that some computer graphics texts give an incorrect test for determing convexity of a polygon by omitting the check for simplicity required by Lemma 4.2.

Recall that the first condition is:

For all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and $t \in \mathbf{T}$ such that $F_1(d) = (s, t)$ and $F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$.

Intuitively, this means that if both executions perform correctly then they will both output the convex hull of the input, which is unique. Note that generation of the certification trail does not affect the output of the Graham Scan algorithm. Thus the condition on $F_1(d)$ is satisfied by the correctness of the Graham Scan algorithm, the proof of which is well known [11]. To show that $F_2(d, t) = s$, note that a copy of s is contained on the trail t. Our description of $F_2(d, t)$ states that s is output unless one of the five checks above fails. It is trivial to verify that the first three of these checks must be satisfied. The fourth check cannot fail, since the polygon described by s is convex (because $(d, s) \in \mathbf{P}$). Similarly, if the fifth check fails, then the polygon described by s has two local maxima, and this is not possible for a convex polygon.

The second condition is:

For all $d \in \mathbf{D}$ all $t \in \mathbf{T}$ either ($F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$) or $F_2(d, t) =$ error.

Intuitively, this means that given an input and arbitrary trail, $F_2(d, t)$ produces a solution to the problem or flags an error.

Our definition of $F_2(d, t)$ states that the polygon Q stored on the trail is output unless one of the five checks fails. We must therefore demonstrate that if all five checks succeed, then Q is the convex hull of the input points d. Let H be the convex hull of the points d. The first condition guarantees that every point in d is classified as a hull point or an interior point. The second condition guarantees that the triangles used to identify interior points are formed from input points, and the third check verifies that the interior points are indeed inside their respective triangles. Note that we do not attempt to verify that the triangles used are the ones that would be produced by $F_1(d)$. In general, for a given interior point, there may be several triangles of input points in which it is contained. Together, the first three conditions imply that all points in H are also in Q, since it is impossible for a hull point to be contained in a triangle. Note that these three checks do not exclude the possibility that interior points are present in Q, nor do they guarantee that the ordering of the hull points in Q is correct. The final two checks will accomplish this. If the last two checks are satisfied, Lemma 4.3 states that Q is simple, and therefore it must be convex by Lemma 4.2.

Thus, Q is a convex polygon whose vertex set is a superset of the vertices of H, i.e., H is contained in T. This implies that no other point from the input set may be a vertex of Q, since any input point that is not a hull point is interior to H and therefore interior to Q. Finally, it is clear that the ordering of the vertices of Q and H must be the same (although there

might appear to be two possible orderings, clockwise and counterclockwise, a clockwise ordering will fail the fourth check). Therefore if all five checks succeed, then the output of $F_2(d,t)$ will be the convex hull of d.

This demonstrates that the algorithms described meet the conditions of Definition 3.2, and are therefore a certification trail solution to the convex hull problem.

Time complexity: In the first execution the sorting of the input points takes $O(n \log(n))$ time where n is the number of input points. One can show that this cost dominates and the overall complexity is $O(n \log(n))$.

It is possible to implement the second execution so that all five checks are done in $O(n)$ time. The first two checks may be done in linear time since the certification trail contains indices into the input data. The third and fourth checks require a constant time calculation at each point. Finally, the uniqueness of the local maxima is clearly checkable in linear time.

Order-of-Magnitude Testing Speedup: It should be noted that for the convex hull problem, we are seeing an order of magnitude speedup for reasonable sized problems. We believe this offers a dramatic demonstration of the efficiency of our proposed software testing technique using certification trails in comparison with the 2-version programming technique.

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
5000	0.64	0.67	0.08	41.41	8.00
10000	1.38	1.40	0.17	43.12	8.12
25000	3.89	3.84	0.46	44.73	8.46
50000	8.44	8.50	0.85	44.61	9.93
100000	17.36	17.68	1.65	44.33	10.52

Table 1: Convex Hull

4.3 Sorting Example

This important problem has a massive literature. In this section we will discuss how to apply the certification trail approach to the sorting problem. Let us assume that the sorting algorithm takes as input an array of n elements and outputs an array of n elements. The algorithm is supposed to place the data in non-decreasing order.

To design a certification trail algorithm we must discover the nature of the data that should be included in the certification trail to allow quick computation of the final output sorted array. Suppose that we decide to use the output array itself as the certification trail. We note that it is easy to check that this array is in non-decreasing order by simply performing a single pass over the array. Unfortunately, it is considerably more difficult to make sure that this array contains exactly the same elements as the original input array. Indeed, this problem has a lower bound time complexity of $\Omega(n \log(n))$ in a comparison based model.

Because of this difficulty we use the permutation of the elements defined by the input and output data arrays as the certification trail. This permutation is computed by attaching an Item Number field to the data elements before sorting. The i-th item receives item number i. After the elements are sorted, the permutation from input to output is obtained by reading the Item Numbers from the elements in their new order.

The second execution reads the permutation from the trail and verifies that it is a permutation on n elements, i.e., that no numbers are repeated or omitted. This permutation is used to rearrange the input elements in linear time. Finally the algorithm checks that these elements are now in non-decreasing order.

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
10000	0.28	0.30	0.04	39.29	7.00
50000	1.80	1.90	0.19	41.94	9.47
100000	3.96	4.08	0.41	43.31	9.66
500000	23.95	24.69	2.14	43.99	11.19
1000000	50.23	51.57	4.38	44.31	11.47

Table 2: Sort

4.4 Certification Trails For Abstract Data Types

Before we present the rest of our example algorithms we discuss a general technique applicable to many algorithms and data structures.

An *abstract data type* is a data object or set of data objects together with a group of operations for manipulating the object(s). Each operation takes a (possibly empty) set of arguments, and some, but not necessarily all, operations return answers. Many algorithms make extensive use of abstract data types.

We describe a method for automatically generating a certification trail for an algorithm which uses an abstract data type. This is done by modifying the abstract data type operations, so that during the first execution they generate a certification trail, and during the second execution they use the certification trail. Otherwise, these operations are identical to the original abstract data type operations, i.e., they take the same type of arguments and have the same return types. The object of creating and using the certification trail is to

allow a more efficient implementation of the abstract data type during the second execution.

We illustrate this technique for the following abstract data type which we call *Ordered Collection*. An Ordered Collection will contain a set of pairs (i, x) where i is an item number, and x is a real number value. (This selection is made for simplicity of description, the elements being stored could be more complex). No two elements of the set may have the same item number, though several items may have a common value. We define a total ordering on pairs by $(i, x) < (i', x')$ iff $x < x'$ or $x = x'$ and $i < i'$

The following operations are defined on an Ordered Collection:

INSERT(i,x) Add the element (i, x) to the set.

DELETE(i) Delete the element with item number i from the set.

PREDECESSOR(i) Let (i, x) be the element in the set with item number i. This operation returns its predecessor, that is, the largest pair less than (i, x). A special value SMALLEST is returned if (i, x) is the smallest element in the set.

MIN Return the smallest element in set.

NEAREST(x) Return the element from the set with value closest to x. If there is a tie, return the element with the smallest item number.

This small set of operations is being chosen for concreteness, several additional operations could be easily defined. If an error occurs during any of these operations, for example, inserting pairs with duplicate item numbers or attempting to delete a non-existent item, then the program terminates indicating an error.

These operations may be modified to produce a certification trail during the first execution by modifying the INSERT(i,x) and NEAREST(x) operations to do the following (in addition to their normal function):

INSERT(i,x) After adding this element to the set, perform a PREDECESSOR(i) operation and write the item number of the answer to the certification trail.

NEAREST(x) Write the item number of the answer to the certification trail.

A typical implementation of an abstract data type supporting the above operations would require $\Omega(n \log(n))$ time to process a sequence of n operations. By using the certification trail, we can achieve linear time for n operations during the second execution. This includes the time necessary to check the trail for correctness as well as use it.

The implementation of the Ordered Collection for the second execution will be a structure called an indexed linked list. This is a doubly linked list, along with an array *Items* of pointers, indexed by item number. The i-th element in this array points to the list node for the element with item number i (or is NULL if no element in the list has item number i). This allows us to find an element in constant time given its item number. The elements themselves are maintained in ascending order (according to the pair ordering given above) on a doubly linked list, i.e., each element has pointers to its successor and predecessor. In addition to the array, we maintain a variable *Start*, which stores the item number of the first element in the list.

The abstract data type operations for the second execution are defined as follows:

INSERT(i,x) Read the item number p from the trail. p is the item number that would be the predecessor of (i, x) if it were in the set. *Items*[p] points to the list node for the element with index p, call this element (p, x_p). We can insert (i, x) after this node using ordinary list operations. Before doing so, however, we make three checks:

i. Check that *Items*[i] is currently NULL, i.e., there is not currently an element with item number i in the set.

ii. Check that (i, x) is greater than (p, x_p).

iii. Check that (i, x) is less than the successor of (p, x_p)

If these checks are satisfied, then (i, x) may be inserted after (p, x_p). Set *Items*[i] pointing to the list node for (i, x).

Note that special cases occur at the beginning and end of the list. We omit the specifics of these cases, mentioning only that *Start* must be updated for insertions at the front of the list.

DELETE(i) Check that *Items*[i] is not NULL, i.e., there is an element with item number i currently in the set. If so, remove it from the linked list, and set *Items*[i] to NULL. If we remove the first element of the list we must also update *Start*.

PREDECESSOR(i) *Items*[i] points to the element with item number i, and its predecessor may be found by following the appropriate pointer.

MIN The variable *Start* indicates the item number of the first element on the list, i.e., the minimum element. *Items*[*Start*] therefore points to this element.

NEAREST(x) Read the index i from the trail. *Items*[i] points to the element having this item number, call it (i, v). To verify that this is the correct answer we will have to check one of its neighbors. If $v < x$, then only the successor of (i, x) could have a value closer to v. Otherwise, only the predecessor is a candidate. Check the appropriate neighbor.

Although our example uses elements that contain item numbers, it is not necessary that the abstract data type be defined in this way. The insert operation of an abstract data type may be modified to tag elements with item numbers as they are inserted.

Variations on this scheme are possible. For example, by modifying DELETE(i) and NEAREST(x) operations so that they also write the item numbers of predecessors to the trail, it is possible to use a singly linked list during the second execution. More sophisticated schemes, involving marking list nodes for deletion and delayed checks, allow the use of singly linked lists without requiring DELETE(i) and NEAREST(x) to produce predecessor information.

The technique in this example generalizes to other abstract data types supporting a predecessor operation. In fact, a somewhat weaker condition often suffices; it is sufficient that the specific implementation of the abstract data type allow the predecessor of an element to be found at the time the element is inserted. The abstract data type itself need not support a predecessor operation. This technique is used in four of our example algorithms.

Using this technique, it is possible to reuse the first execution code, except for the code implementing the abstract data type operations. One advantage of this is that it may be possible to add extra checking to such code, such as bounds checking and checks on pointer references, that may be too expensive to include in the first execution. Of course, the two programs may be developed separately as long as the specifications agree on the use of the abstract data type.

Space does not permit a full proof of correctness of this scheme. A proof proceeds by establishing the following invariants on the indexed linked list used in the second execution.

i. The pairs in the linked list are in order from smallest to largest.

ii. Each element of the *Items* array is either NULL or points to one of the nodes in the linked list.

iii. If *Items*[i] is not NULL, then the list node pointed to by it stores an element with item number i. (Note that this implies that each list node is pointed to at most once).

iv. Every node in the list is pointed to by some item in *Items*[i].

v. *Start* is the item of the first element in the list.

These conditions are clearly satisfied by an indexed linked list containing no elements (i.e., before any operations have been performed). Inspection of operations that query the list (MIN and NEAREST for example) shows that they function correctly if the above conditions are met. It is easy to prove correctness of the certification trail by demonstrating that the operations maintain a one to one corresponce between the pairs in the linked list and the elements in the abstract data type and that the above invariants are preserved.

4.5 Shortest Path Example

This is another classic problem which has been examined extensively in the literature. Our approach is applied to a variant of the Dijkstra algorithm [3] as explicated in [17]. We are concerned with the single source problem, i.e., given a graph and a vertex s, find the shortest path from s to v for every vertex v.

The algorithm for this problem which has the fastest asymptotic time complexity uses fusion trees and is given in [5]. This algorithm however appears to have a large constant of proportionality and therefore we do not use it.

We use the techniques just discussed to implement the certification trail for this problem. A full description may be found in a technical report [15].

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
100,1000	0.04	0.05	0.02	12.50	2.00
250,2500	0.15	0.16	0.06	26.67	2.50
500,5000	0.31	0.33	0.11	29.03	2.82
1000,10000	0.70	0.76	0.23	29.29	3.04
2000,20000	1.58	1.67	0.45	32.91	3.51
2500,25000	2.06	2.15	0.55	34.47	3.75

Table 3: Shortest Path

4.6 Huffman Tree Example

This is another classic algorithmic problem and one of the original solutions was found by Huffman[7]. It has been used extensively to perform data compression through the design and use of so called Huffman codes. These codes are prefix codes which are based on the

Huffman tree and which yield excellent data compression ratios. The tree structure and the code design are based on the frequencies of individual characters in the data to be compressed. Here we are concerned exclusively with the Huffman tree. See [7] for information about the coding application.

Definition 4.4 The Huffman tree problem is the following: Given a sequence of frequencies (positive integers) $f[1], f[2], \ldots, f[n]$, construct a tree with n leaves and with one frequency value assigned to each leaf so that the weighted path length is minimized. Specifically, the tree should minimize the following sum: $\sum_{l_i \in \text{LEAF}} \text{len}(i) f[i]$ where LEAF is the set of leaves, $\text{len}(i)$ is the length of the path from the root of the tree to the leaf l_i, $f[i]$ is the frequency assigned to the leaf l_i.

A full description of the method we employ to generate and use a certification trail is detailed in a technical report [15].

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
5000	0.81	0.87	0.16	36.42	5.06
10000	1.76	1.86	0.33	37.78	5.33
25000	6.01	6.30	1.02	39.10	5.89
50000	10.62	11.14	1.70	39.55	6.25

Table 4: Huffman tree

4.7 Other problems

We report timing data for five other problems, the "Manhattan skyline" problem, computation of Voronoi diagrams, longest increasing subsequence, the closest pair problem, and line segment intersection. Space permits only a brief description of these problems, rather than a full exposition of the certification trail techniques used.

The "Manhattan skyline" problem is: Given a set of rectangles with collinear bottom edges, compute the polygonal outline of the union of the rectangles [9].

The Voronoi diagram is a fundamental concept in computational geometry [11]. Given a set of points P in the plane, the Voronoi diagram is a partition of the plane into regions such that each region consists of all points closer to a given $p \in P$ than to any other other point in P. Computation of the Voronoi diagram is an important step in many problems involving point location.

The next problem we consider is, given a sequence of integers, find the longest (not necessarily unique) strictly increasing subsequence.

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
1000	0.27	0.26	0.12	29.63	2.25
5000	1.69	1.65	0.57	34.32	2.96
10000	3.91	3.72	1.14	37.85	3.43
15000	6.08	5.78	1.77	37.91	3.44
20000	8.53	8.27	2.33	37.87	3.66

Table 5: Skyline

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
100	0.04	0.04	0.03	12.50	1.33
500	0.24	0.26	0.19	6.25	1.26
1000	0.51	0.51	0.39	11.76	1.31
5000	2.75	2.82	2.03	11.82	1.35
10000	5.79	5.89	4.06	14.08	1.43
50000	40.15	40.63	22.00	22.00	1.83

Table 6: Voronoi Diagram

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
10000	0.13	0.14	0.04	30.77	3.25
50000	0.78	0.81	0.22	33.97	3.55
100000	1.61	1.70	0.44	33.54	3.66
500000	9.17	9.32	2.22	37.08	4.13
1000000	18.66	19.58	4.46	35.58	4.18

Table 7: Longest Increasing Subsequence

Given a set of points P in the plane, the Closest Pair problem is that of finding the pair of points with minimum distance over all pairs in the set.

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
10000	0.26	0.27	0.07	34.62	3.71
50000	1.45	1.55	0.36	34.14	4.03
100000	3.06	3.26	0.72	34.97	4.25
500000	16.84	18.02	3.62	35.75	4.65

Table 8: Closest Pair

Given a set of line segments in the plane, the line intersection problem is the problem of determining all intersections of line segments in this set.

For the first four problems, algorithms running in $O(n \log(n))$ time were implemented for the first execution. The second execution, using certification trails, runs in linear time. The first execution algorithm used for line intersection runs in $(O((k+n)\log(n))$ time where k is the number of intersections and n the number of points. The second execution runs in $O(k+n)$ time. Note that k may be quadratic in n.

Size	Basic	Prim. Exec. (Also Gen. Trail)	Sec. Exec.	% Sav.	Speedup
1000	0.47	0.49	0.04	43.62	11.75
2500	1.45	1.53	0.12	43.10	12.08
5000	3.33	3.47	0.26	43.99	12.81
10000	7.72	7.88	0.60	45.08	12.87
25000	24.00	24.12	1.75	46.10	13.71

Table 9: Line Segment Intersection

5 Concluding Discussion

Certification trails have heretofore been discussed principally from a theoretical perspective. In this paper we have presented experimental timing data which illustrates the advantages of the certification trail technique for software testing over the 2-version programming technique. We have further presented techniques and analytical results for several new algorithms which further support the significance of the certification trail technique by demonstrating its broadening applicability. It should be appreciated that the scope of our experimental investigation is not limited to the algorithms considered here; numerous other algorithms we have considered could have been discussed, and we continue to work on new applications. It should also be pointed out that in addition to the timing experiments reported here, software fault injection experiments have also been conducted which verify the detection capabilities of the certification trail method. The breadth of applicability of the certification trail technique continues to expand along with the credibility of its advantages. Increasingly, the certification trail method can be viewed as a competitive software testing alternative.

References

[1] Avizienis, A., "The N-version approach to fault tolerant software," *IEEE Trans. on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.

[2] Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *1978 Fault Tol. Comp. Symp.*, pp. 3-9, IEEE Computer Society Press, 1978.

[3] Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math. 1*, pp. 269-271, 1959.

[4] Fortune, S. "A Sweepline Algorithm for Voronoi Diagrams," *Algorithmica*, pp. 153-174, 2, 1987.

[5] Fredman, M. L., and Willard, D. E., "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Proc. 31st IEEE Foundations of Computer Science*, pp. 719-725,1990.

[6] Graham, R. L., "An efficient algorithm for determining the convex hull of a planar set", *Information Processing Letters*, pp. 132-133, 1, 1972.

[7] Huffman, D., "A method for the construction of minimum redundancy codes", *Proc. IRE*, pp 1098-1101, 40, 1952.

[8] Johnson, B., *Design and analysis of fault tolerant digital systems* Addison-Wesley, Reading, MA, 1989.

[9] Manber U., *Introduction to Algorithms* Addison-Wesley, Reading, MA, 1989.

[10] Nievergelt, J., and Hinrichs, K. H., *Algorithms and Data Structures With Applications to Graphics and Geometry*, Prentice Hall, NJ 1993

[11] Preparata F. P., and Shamos M. I., *Computational geometry*, Springer-Verlag, New York, NY, 1985.

[12] Sedgewick, R., "Implementing quicksort programs," *Comm. of the ACM*, pp. 847-857, 21(10), 1978.

[13] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[14] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Digest of the 1990 Fault Tolerant Computing Symposium*, pp. 423-431, IEEE Computer Society Press, 1990.

[15] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Department of Computer Science Technical Report JHU 89/26*, Johns Hopkins University, Baltimore, Maryland, 1989.

[16] Sullivan, G.F., and Masson, G.M., "Certification trails for data structures," *Digest of the 1991 Fault Tolerant Computing Symposium*, pp. 240-247, IEEE Computer Society Press, 1991.

[17] Tarjan, R. E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.