198571

P-27

Certification of Computational Results

Gregory F. Sullivan[1]

Dwight S. Wilson[2]

Gerald M. Masson[3]

Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

We describe a conceptually novel and powerful technique to achieve fault detection and fault tolerance in hardware and software systems. When used for software fault detection, this new technique uses time and software redundancy and can be outlined as follows. In the initial phase, a program is run to solve a problem and store the result. In addition, this program leaves behind a trail of data which we call a *certification trail.* In the second phase, another program is run which solves the original problem again. This program, however, has access to the certification trail left by the first program. Because of the availability of the certification trail, the second phase can be performed by a less complex program and can execute more quickly. In the final phase, the two results are compared and if they agree the results are accepted as correct; otherwise an error is indicated. An essential aspect of this approach is that the second program must always generate either an error indication or a correct output even when the certification trail it receives from the first program is incorrect. We formalize the certification trail approach to fault tolerance and illustrate realizations of it by considering algorithms for the following problems: convex hull, sorting, and shortest path, We discuss cases in which the second phase can be run concurrently with the first and act as a monitor. We compare the certification trail approach to other approaches to fault tolerance.

Keywords: Software fault tolerance, error monitoring, design diversity, data structures.

1 Introduction

In this paper we describe a novel and powerful technique for achieving fault tolerance in systems. Although applicable to both hardware and software implementation, we restrict our discussion of this technique to implementation in software. To explain our technique, we will first discuss a simpler method. In this method the specification of a problem is given and an algorithm to solve it is constructed. This algorithm is executed on a particular input and the output is stored. Next, the same algorithm is executed again on the same input and the output is compared to the earlier output. If the outputs differ then an error is indicated, otherwise the output is accepted as correct. This method requires additional time, so called time redundancy [16, 22]; however, it requires no additional software. It is particularly valuable for detecting errors caused by transient fault phenomena. If such faults cause an error during only one of the executions then either the error will be detected or the output will be correct.

A variation of the above method uses two separate algorithms, one for each execution, which have been written independently based on the problem specification. This technique, called N-version programming [9, 3] (in this case N=2), allows for the detection of errors caused by some faults in

[1] Research partially supported by NSF Grants CCR-8910569 and CCR-8908092.

[2] Research partially supported by NSF Grant CDA-9015667.

[3] Research partially supported by NASA Grant NSG 1442.

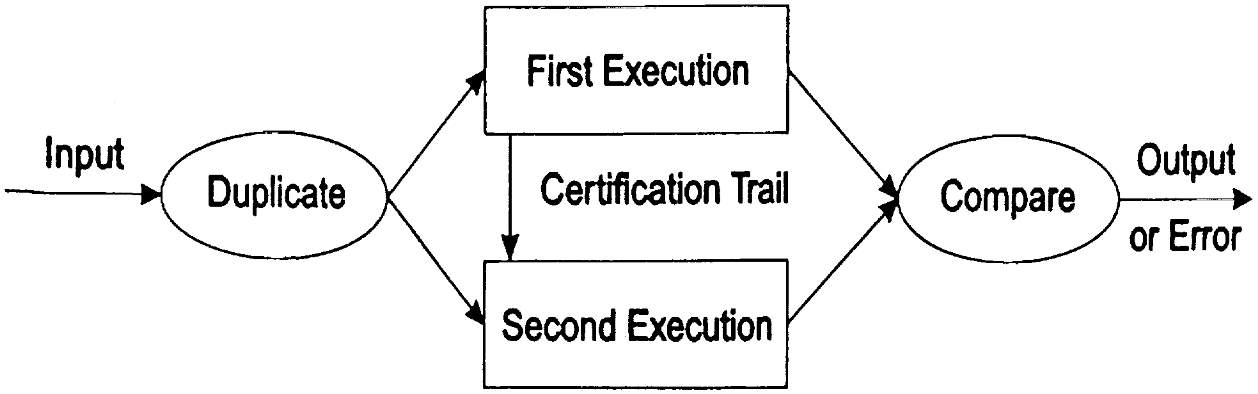

Figure 1: Certification trail method.

the software in addition to those caused by transient hardware faults and utilizes both time and software redundancy. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors.

A significant drawback to the above approaches is the overhead required. Either extra time is required to run the algorithms serially on a single processor or extra hardware is required to run them in parallel. The technique we will describe is designed to achieve similar types of error detection capabilities while reducing the required resource overhead. The central idea, as illustrated in Figure 1, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail*. This data is chosen to 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. 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 algorithm executions. For example, suppose the first algorithm 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 algorithm. It appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output produced by the first algorithm. Intuitively, we can regard the two executions as "adversaries." The second execution must guard against an incorrect certification trail "fooling" it into producing an incorrect output. The definitions we give below exclude this possibility. They demand that the second execution either generates a correct answer or signals the fact that an error has been detected in the certification trail.

2 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 2.1 A problem **P** is formalized as a relation, i.e., a set of ordered pairs. Let **D** be the domain (that is, the set of inputs) of the relation **P** and let **S** be the range (that is, the set of solutions) for the problem. We say an algorithm **A** solves a problem **P** iff for all $d \in \mathbf{D}$ when d is input to **A** then an $s \in \mathbf{S}$ is output such that $(d, s) \in \mathbf{P}$.

Definition 2.2 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 there exists $t \in \mathbf{T}$ 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 for 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 not in their respective domains to the error symbol. The definitions above assure that the error detection capability of the certification trail approach is comparable to that obtained with the simple time redundancy approach discussed earlier. (That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct.) It should be further noted, however, that the examples to be considered will indicate that this approach can also save overall execution time.

The certification trail approach also allows for the detection of faults in software. As in 2-version programming, separate teams can write the algorithms for the first and second executions. Note that the specification now must include precise information describing the generation and use of the certification trail. Because of the additional data available to the second execution, the specifications of the two phases can be very different; similarly, the two algorithms used to implement the phases can be very different. (This will be illustrated in the convex hull example to be considered later.) Alternatively, the two algorithms can be very similar, differing only in data structure manipulations. (This will be illustrated in the shortest path example to be considered later.) When significantly different algorithms are used, the probability that both algorithms will contain or be affected by faults which generate matching errors should be reduced. When very similar algorithms are used it is sometimes possible to save programming effort by sharing program code. For example, the code implementing any data structures needed by the program might be different, while the code that uses the data structure operations would be the same. This approach is well suited for the creation of libraries of fault-tolerant data structures. While this reduces the ability to detect errors in the software it does not change the ability to detect transient hardware errors as discussed earlier. Furthermore, in situations like the above example, it is possible (perhaps even probable) that the majority of software errors will be in the data structure implementation. Thus the ability to detect software errors may not be reduced as much as first imagined.

Throughout this section we have assumed that our method is implemented with software, however, it is clearly possible to implement the method with assistance from dedicated hardware. It is also possible to generalize the basic idea we have suggested. We discuss some of these generalizations in a later section. Finally, we note that a wide variety of approaches to software fault tolerance have been proposed and we contrast our method to the most closely related ideas in a later section.

In the following two sections we illustrate the application of certification trails to three well-known and significant problems in computer science: the convex hull problem, sorting, and the shortest path problem. It should be stressed that the certification trail is not limited to these problems. Rather, these algorithms have been selected for illustrative purposes.

3 Certification Trails for Convex Hulls

The convex hull problem is a fundamental one in computational geometry. Our certification trail solution is based on a solution due to Graham [13] called Graham's Scan. For basic definitions in computational geometry see the text of Preparata and Shamos [20]. This text also illustrates some statistical applications of convex hull computations. For simplicity in the following discussion we will assume the points are in so called general position, i.e., no three points are co-linear. It is not difficult to remove this restriction.

Definition 3.1 The *convex hull* of a set of N points, S, 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 S. It is specified by a counterclockwise sequence of its vertices.

The algorithm given below constructs the convex hull incrementally in a counterclockwise fashion. Sometimes it is necessary for the algorithm to "backup" the construction by throwing some vertices out and then continuing. The first step of the algorithm selects the point with minimum x-coordinate (using minimum y-coordinate to break ties), and calls it p_1. For each other point q in S we compute the slope of the line p_1q. Sort the points of S (except for p_1) by this slope (since the points are in general position, the slopes are distinct). Number these vertices $p_2, p_3, \ldots, p_N$. 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 might not be convex. It is possible to think of the algorithm as removing points from this simple polygon until it becomes convex. This code below performs this by "walking" through the vertices in order. The main FOR loop iteration adds points to the polygon under construction. After a point is added, the inner WHILE loop checks the angle formed by the addition of this point. (Note: We measure angles as follows: Given the three points q_{m-1}, q_m, p_k we measure the angle from $q_{m-1}q_m$ to q_mp_k in the clockwise direction.) If the angle is not acute (i.e., it makes the the polygon non-convex), then the angle vertex (i.e., the preceding point on the polygon) is removed. Note that this will change the preceding angle, which may now be obtuse and should be eliminated. The WHILE loop terminates when an acute angle is encountered. Figure 2 illustrates the construction of a convex hull using this algorithm. from the hull.

When the main FOR loop is complete the convex hull has been constructed.

Algorithm CONVEXHULL(S)
Input: Set of points, S, in R^2
Output: Counterclockwise sequence of points in R^2 which define convex hull of S

```
Let $p_1$ be the point with the smallest $x$ coordinate (and smallest $y$ to break ties)
For each point $p$ (except $p_1$) calculate the slope of the line through $p_1$ and $p$
Sort the points (except $p_1$) from the smallest slope to the largest.
Call them $p_2, \ldots, p_n$
$q_1 := p_1$; $q_2 := p_2$; $q_3 := p_3$; $m = 3$
FOR $k = 4$ to $n$ DO
  WHILE the angle formed by $q_{m-1}, q_m, p_k$ is $\geq 180$ degrees DO
    $m := m - 1$
  END WHILE
  $m := m + 1$
  $q_m := p_k$
END FOR
FOR $i = 1$ to $m$ DO, OUTPUT($q_i$) END FOR
```

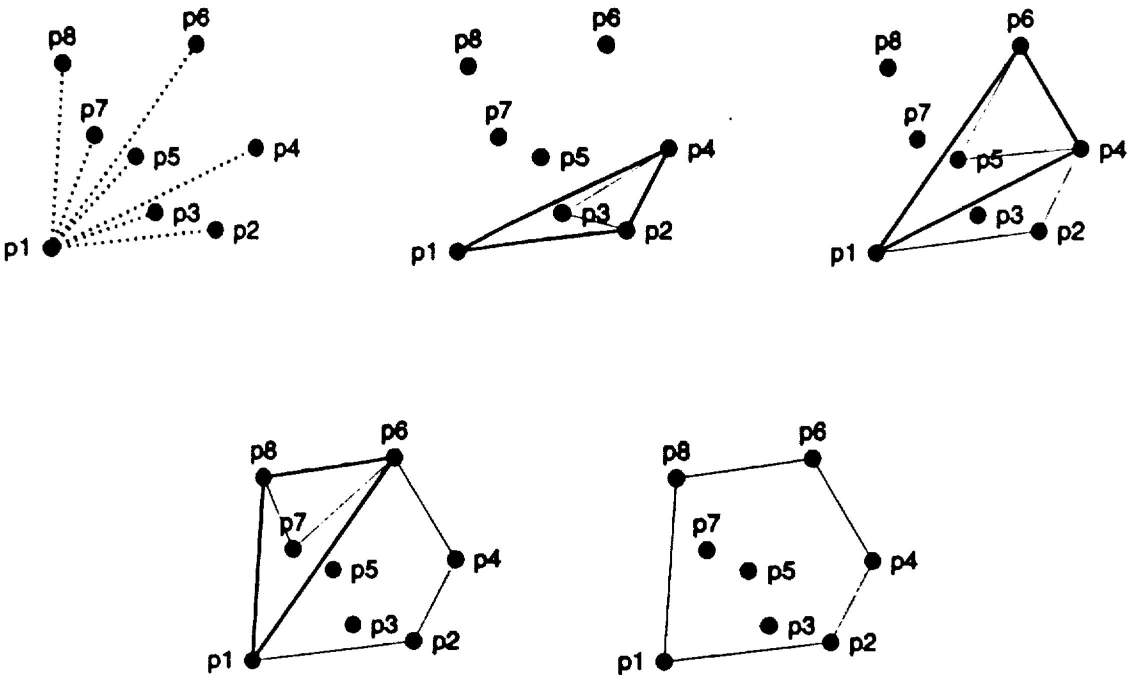

Figure 2: Convex hull example.

END CONVEXHULL

First execution: To generate a certification trail for this algorithm, we rely on the property that for each point eliminated by the WHILE loop in the code above, we can produce a triangle of points in S containing the eliminated point.

Theorem 3.2 *Let p, a, b, and c, be points in the plane such that no three are co-linear, p has the smallest x-coordinate of the four points (and the smaller y-coordinate if another other point has the same x-coordinate) $slope(\overline{pa}) < slope(\overline{pb}) < slope(\overline{pc})$. If the angle abc is obtuse (measured in the clockwise direction), then b is inside the triangle pac.*

Proof: By the ordering of the slopes, b is inside the triangular wedge determined by the rays $\vec{pa}$ and $\vec{pc}$. Note that the line segments pa and pc are in the half plain $x \geq p_x$, and in at least one case the inequality is strict, since no three points are co-linear. This implies that the angle apc (in the clockwise direction) must be greater than 180 degrees. Since the angle abc is also obtuse, both p and b must be on the same side of line $\overline{ac}$. Therefore, b is inside the triangle pac. ▮

Corollary 3.3 *During execution of CONVEXHULL, if, after adding p_k, the angle formed by q_{m-1}, q_m, p_k is obtuse (measured in the clockwise direction), then q_m is contained in the triangle p_1, q_{m-1}, p_k.*

Proof: $slope(\overline{p_1 q_{m-1}}) < slope(\overline{p_1 q_m}) < slope(\overline{p_1 p_k})$. ▮

In the first execution the code CONVEXHULL is used. The certification trail is generated by adding an output statement within the WHILE loop. Specifically, if an angle greater than 180 degrees is found in the WHILE loop test then the 4-tuple consisting of q_m, q_{m-1}, p_1, p_k is output to the certification trail. The table below shows the 4-tuples of points that would be output by the algorithm when run on the example in Figure 2. The points in the table are given the same names as in Figure 2. The final convex hull points $q_1, \ldots, q_m$ are also output to the certification trail. Finally, 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 $s_1, s_2, \ldots, s_n$ then rather than output the element in R^2 corresponding to s_i the number i is output. If point coordinates were output instead of these indices, the second execution would have to verify that the points on the trail are members of S.

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

Second execution: Let the certification trail consist of a set of 4-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 performed is dramatically different than CONVEXHULL.

It consists of five checks on the trail data.

i. That there is a one to one correspondence between the input points and the points in $\{x_1, \ldots, x_r\} \cup \{q_1, \ldots, q_m\}$.

ii. That for $i \in \{1, \ldots, r\}$, a_i, b_i, and c_i are among the input points.

iii. For $i \in \{1, \ldots, r\}$ that x_i lies within the triangle defined by a_i,b_i, and c_i.

iv. That for each triple of counterclockwise consecutive points on the supposed convex hull the angle formed by the points is acute.

v. That there is a unique point among the points on the supposed convex hull which is a locally maximal point. We say a point q on the hull is a *local maximum* point 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. As mentioned above, the trail data actually consists of indices into the input data. This does not unduly complicate the checks above; in fact it makes it easier to verify the first and second conditions.

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. Because indices into the input data are used, the first condition can be checked by verifying that each index is used exactly once, and that all indices are between 1 and N. The second condition may checked simply by verifying that each index is between 1 and N. Checking that a point lies within a triangle is a geometric calculation that can be done in constant time. Checking that the angle formed by three points is acute requires only constant time. The third and fourth checks can be done in $O(n)$ because the certification trail contains indices into the input data as described above. The uniqueness of the "local maximum" requires only a constant time calculation at each point, so it may checked in linear time.

Experimental timing data for this method may be found in Section 6.

3.1 Proof of correctness

We wish to prove that the algorithms above constitute a certification trail solution for the convex hull problem. Although the definition is phrased in terms of functions, not algorithms, we can simply define the functions $F_1(d)$ and $F_2(d,t)$ on particular arguments as the values computed by the associated algorithms.

Using our formal definition of certification trails, let **D** be the set of all finite planar point sets T. Let **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 **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 2.2 hold. The following two lemmas, which we state without proof, are required.

Lemma 3.4 *Let P be a polygon on n points $p_1, p_2, \ldots, p_n$. P is a convex polygon iff P is simple nd each angle $p_ip_jp_k$ is less than or equal to 180 degrees, where i is in $1, 2, \ldots n$, $j = (i+1) \bmod n$, nd $k = (i+2) \bmod n$.*

emma 3.5 *If P is a non-simple polygon, then either P has more than one local maxima, or the terior angle at some vertex is greater than 180 degrees.*

heorem 3.6 *$F_1(d)$ and $F_2(d,t)$, as defined above, constitute a certification trail solution for the oblem HULL.*

oof: We must prove that both conditions of Definition 2.2 are satisfied by these functions.

Part 1: Recall that the first condition is: for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and $t \in \mathbf{T}$ such $F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in \mathbf{P}$. Intuitively, this means that if both executions orm correctly, then they will both output the convex hull of the input, which is unique. Note generation of the certification trail does not affect the output of the Graham Scan algorithm. s the condition on $F_1(d)$ is satisfied by the correctness of the Graham Scan algorithm, the proof hich is well known [20]. To show that $F_2(d,t) = s$, note that a copy of s is contained on the t. Our description of $F_2(d,t)$ states that s is output unless one of the five checks above fails. rivial to verify that the first three of these checks must be satisfied. The fourth check cannot ince the polygon described by s is convex (because $(d,s) \in \mathbf{P}$). Similarly, if the fifth check then the polygon described by s has two local maxima, and this is not possible for a convex on.

t 2: 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 $)$ = error. Intuitively, this means that given an input and arbitrary trail, $F_2(d,t)$ produces a n to the problem or flags an error. Our definition of $F_2(d,t)$ states that the polygon Q stored trail is output unless one of the five checks fails. We must therefore demonstrate that if all ecks succeed, then Q is the convex hull of the input points d. Let H be the convex hull of nts 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 on the trail 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 3.5 states that Q is simple, and therefore it must be convex by Lemma 3.4.

Thus, Q is a convex polygon whose vertex set is a superset of the vertices of H, i.e., H is contained in Q. 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 2.2, and are therefore a certification trail solution to the convex hull problem. ▮

3.2 Other convex hull algorithms

It is possible to use this technique to provide certification trails for other convex hull algorithms. The key is that for each non-hull point p we must find a triangle of input points (not necessarily hull points), containing p. For some convex hull algorithms, a containing triangle is available directly or can be easily computed when it is determined that a particular point is not on the hull. However, this is not true of all convex hull algorithms. If, however, we allow extra overhead during the first execution we may apply this technique to any planar convex hull algorithm, provided that the output is a polygon and not merely an unordered list of hull vertices.

Let $H = q_1, q_2, q_3 \ldots, q_h$ be the convex hull of a set of n points. We label the points so that q_1 is the point with smallest abscissae (and smallest ordinate in case of a tie). Since H is convex, the remaining points occur in sorted angular order around q_1. Now for each non-hull point p, we may determine which triangle $p_1 p_i p_{i+1}$ it lies in with a binary search. Thus we may determine containing triangles for the non-hull points in $O(nlogh)$ time. Under several distributions the number of hull points is much smaller than the number of input points [20] so this overhead will often be quite small.

4 Sorting

Sorting is one of the most important basic problems in computer science. There is a massive body of literature discussing sorting and a significant fraction of computer time is spent performing sort operations. We will see how the certification trail approach may be applied to this problem. Assume that a particular 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 into non-decreasing order.

Note that it may not appear necessary to use a certification trail for this problem. It might seem that all that is required is to verify that the output is in non-decreasing order. Unfortunately, this is not sufficient and we must also verify that the output consists of the same elements as the input. A certification trail is required to perform this check efficiently.

The information placed on the trail is a permutation relating the input and output arrays. This permutation is created by adding an Item Number field to the elements being sorted, such that the i-th element is labelled with item number i. After sorting, the permutation is obtained by reading the Item Numbers from the elements in their new order.

The second algorithm reads the permutation from the trail, uses it to rearrange the input elements in linear time, and checks that they are now in sorted order. Additionally, it is necessary to check that the the information on the certification trail actually is a permutation of n elements, i.e., each number from 1 to n occurs exactly once. Should any of these checks fail, the second algorithm outputs "error", otherwise it outputs the sorted elements.

Note that the certification trail given for sorting is quite different than that given for the convex hull problem. In the latter case, the certification trail was constructed for a particular algorithm, and the code executing that algorithm modified to produce the trail. In this case, the sorting algorithm is not changed. Instead the data being sorted is modified by a preprocessing step, and the necessary information extracted by a postprocessing step. Thus this technique may be implemented as a "wrapper" around existing sort routines, no matter which algorithm is implemented.

Experimental data is presented in Section 6.

4.1 Proof of correctness

For concreteness we consider only the sorting of integers, though the proof does not depend on this condition.

Definition 4.1 Let **D** consist of all finite sequences of integers. Let **S** consist of all finite non-decreasing sequences of integers. Let $\mathbf{P} : \mathbf{D} \to \mathbf{S}$ be the sorting problem, i.e., $(d, s) \in \mathbf{P}$ iff s is a permutation of d (by definition of **S**, s is a non-decreasing sequence). Note that for every $d \in \mathbf{D}$, there is a unique $s \in \mathbf{S}$ such that $(d, s) \in \mathbf{P}$. Let **T** consist of finite sequences of integers. For x a member of any of the sets $\mathbf{D}, \mathbf{S}$, or $\mathbf{T}$, we will also denote the sequence of integers by $x_1, x_2, ..., x_N$.

Definition 4.2 The function $F_1 : \mathbf{D} \to \mathbf{S} \times \mathbf{T}$ is defined as follows. Given an input sequence d of N integers, $F_1(d) = (s,t)$ where s is the unique element of **S** such that, $(d, s) \in \mathbf{P}$ and t is a permutation of 1,2,3,...,N s.t., $s_i = d_{t_i}$ for all $i = 1, 2, ...N$. Note that unless d consists of N distinct integers, there will be more than one possible t. The t produced by $F_1(d)$ may be chosen arbitrarily. Since for every $d \in \mathbf{D}$, there exists a unique $s \in \mathbf{S}$ with $(d, s) \in \mathbf{P}$, the function F_1 is well defined.

Definition 4.3 The function $F_2 : \mathrm{D} \times \mathrm{T} \to \mathrm{S} \cup \{\text{error}\}$ is defined as follows. $F_2(d,t) = d_{t_1}, d_{t_2}, ..., d_{t_N}$ (where d consists of N integers) iff

i. t contains at least N integers.

ii. The first N integers of t are a permutation of $\{1, 2, ...N\}$.

iii. $d_{t_i} \le d_{t_{i+1}}$ for $i = 1, 2, ..., N - 1$.

Otherwise, $F_2(d,t)$ = error. Note that though t may contain more than N integers, $F_2(d,t)$ depends only on the first N.

The definitions of the functions F_1 and and F_2 correspond to the informal descriptions of the sorting algorithms given in the text above.

Theorem 4.4 *F_1 and F_2 are a certification trail solution to the sorting problem* **P**.

Proof: We must prove that both conditions of Definition 2.2 are satisfied by these functions.

Part 1: We must prove that 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 $(d, s) \in \mathbf{P}$. If $F_1(d) = (s, t)$, then by definition $(d, s) \in \mathbf{P}$. We must show that $F_2(d, t) = s$. t is a permutation of $\{1, 2, ..., N\}$, so the first two conditions of Definition 4.3 are satisfied. Furthermore, by Definition 4.2, $d_{t_i} = s_i$ for $i = 1, 2, ...N$. Since $s \in \mathbf{S}$, it is a nondecreasing sequence, and thus the third condition of Definition 4.3 is satisfied. Therefore $F_2(d, t) = s$.

Part 2: We must show that 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 $F_2(d, t) =$ error. Pick $d \in \mathbf{D}$ with length N. Pick $t \in \mathbf{T}$. The interesting case is when t is a permutation of $\{1, 2, ..., N\}$. If not, then either the first N integers of t are not such a permutation, in which case $F_2(d, t) =$ error. We may ignore the possibility that t consists of such a permutation followed by more integers, since F_2 depends only on the first N integers of t.

Examine the sequence $d_{t_1}, d_{t_2}, , , d_{t_N}$. If there is an i such that $d_{t_i} > d_{t_{i+1}}$ then the third condition of Definition 4.3 is violated so $F_2(d, t) =$ error. Otherwise $F_2(d, t) = d_{t_1}, d_{t_2}, ..., d_{t_N}$. Furthermore, this is a non-decreasing sequence, so it must be in $\mathbf{S}$. Finally, since this sequence is a permutation of d, $(d, F_2(d, t)) \in \mathbf{P}$.

Therefore, both conditions of Definition 2.2 are satisfied, so F_1 and F_2 constitute a certification trail solution to sorting. ∎

Note that we defined $\mathbf{T}$ as the set of all finite sequences of integers. We could have instead defined $\mathbf{T}$ as the set of permutations of $\{1, 2, ...N\}$ for all positive N. This would make the function F_2 "simpler", in that it doesn't have to verify that that certification trail consists of a permutation (it would, however, have to verify that it consists of a permutation of the correct size). In this case, checking that the trail t is indeed a permuation (i.e., actually in its domain) would be left to the implementation of the function.

5 Certification Trails for Shortest Paths

This classic problem has been examined extensively in the literature. Our approach is applied to a variant of the Dijkstra algorithm [11] as explicated in [10]. First we require some preliminary definitions.

Definition 5.1 A *graph* $G = (V, E)$ consists of a *vertex* set V and an *edge* set E. An edge is an unordered pair of distinct vertices which we notate with the following style: $[v, w]$ and we say v is adjacent to w. A *path* in a graph from v_1 to v_k is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $[v_i, v_{i+1}]$ is an edge for $i \in \{1, \ldots, k-1\}$. Let w be a real function defined on E. The *length* of a path from v_1 to v_k is the sum of $w([v_i, v_{i+1}])$ for each edge $[v_i, v_{i+1}]$ in the path.

Let $G = (V, E)$ be a graph and let w be a positive rational valued function defined on E. Given a vertex v_1 in V, find a set of shortest paths from v_1 to each other vertex in V. Note that since w is positive on all edges, a shortest path must exist between any two vertices, though it need not be unique.

Before we discuss the algorithm we must describe the properties of the principal data structure that are required. Since many different data structures can be used to implement the algorithm, we initially describe abstractly the data that can be stored by the data structure and the operations that can be used to manipulate this data. The data consists of a set of ordered pairs. The first element in these ordered pairs is referred to as the *item number* and the second element is called the *item value* or just *value*. Ordered pairs may be added and removed from the set, however, at all times the item numbers of distinct ordered pairs must be distinct. It is possible, though, for multiple ordered pairs to have the same item value. In this paper the item numbers are integers between 1 and n, inclusive. Our default convention is that i is an item number, x is a value and h is a set of ordered pairs. A total ordering on the pairs of a set can be defined lexicographically as follows: $(i, x) < (i', x')$ iff $x < x'$ or ($x = x'$ and $i < i'$). Our data structure should support a subset of the following operations.

member(i, h) returns a boolean value of true if h contains an ordered pair with item number i, otherwise returns false.

insert(i, x, h) adds the ordered pair (i, x) to the set h.

delete(i, h) deletes the unique ordered pair with item number i from h.

changekey(i, x, h) is executed only when there is an ordered pair with item number i in h. This pair is replaced by (i, x).

deletemin(h) returns the ordered pair which is smallest according to the total order defined above and deletes this pair. If h is the empty set then the token "empty" is returned.

predecessor(i, h) returns the item number of the ordered pair which immediately precedes the pair with item number i in the total order. If there is no predecessor then the token "smallest" is returned.

A description such as the one above describes an *abstract data type*. There may be several possible implementations for a particular ADT. In our solution, different ADT implementations will be used for the two executions. The first implementation will produce a certification trail allowing the second implementation to be simpler and to perform ADT operations more quickly.

Aside from the implementation of the abstract data type, both of our algorithms are the same. Pidgin code for this algorithm appears below. Figure 3 illustrates the execution of the algorithm on a sample graph. Table 1 records the data structure operations performed when the algorithm is run on the sample graph. The first column gives the operations, with the parameter h omitted to reduce clutter. Member operations are also omitted from the table. The second column gives contents of h after the execution of each instruction. The third column records the order pair deleted by deletemin operations. The fourth column records the information (if any) output to the certification trail by this operation.

This certification trail is created by modifying the insert(i, x, h) and changekey(i, x, h) operations performed during the first execution. The modified instructions perform the same operations described above and in addition output the following information to the certification trail.

insert(i, x, h) Output the item number of the predecessor of (i, x) (as defined above) to the trail. If there is no predecessor, output the token "smallest". Note that depending on the data structure implementation, the predecessor may already be computed during insertion or may require a separate call to the predecessor(i, h) operation.

changekey(i, x, h) Output the predecessor of the ordered pair (i, x) (i.e., pair resulting from the change) to the trail. If there is no predecessor, output the token "smallest" to the trail.

We shall see that this information allows a faster and simpler data structure implementation to be used for our second algorithm.

The algorithm proceeds by maintaining a set S of vertices for which shortest path lengths are known, and a "frontier" set F of vertices adjacent to members of S along with the best known path length from v_1. At each step, we find the vertex v in F with smallest known path length and place it in S, F is then updated by examining the neighbors of v. New vertices may be added to F or a shorter path (passing through v) may be found to existing vertices in F.

To efficiently find the vertex to add to S, the algorithm uses the data structure operations described above. As soon as a vertex v is adjacent to some vertex u in S, it is inserted in the set F. The value for v is the shortest known path to v, which is the value of u (shortest path to u) plus the weight of edge vw. The array element prefer(v) is used to keep track of this "best" edge connecting v to S. As the tree grows, information is updated by operations such as insert(i, x, h) and changekey(i, x, h). The deletemin(h) operation is used to select the next vertex to add to the span of the current tree. Note, the algorithm does not explicitly store paths. Implicitly, however, if (v, x) is returned by deletemin, then prefer(v) indicates the predecessor of v on the shortest path from v_1.

```
Algorithm SHORTEST-PATH($G$,$v_1$,weight)
Input: Connected graph $G = (V, E)$ where $V = \{1, \ldots, n\}$ with edge weights.
Output: Lengths of shortest paths from $v_1$ to all other vertices.
FOR ALL $u \in V$, $u) := \infty$ END FOR
$v_1) := 0$
$F := v_1$;
WHILE $F \neq \emptyset$ DO
 $(v, k) :=$ deletemin($F$)
 FOR EACH $[v, w] \in E$ DO
  IF $v) +$ weight($[v, w]$) $< w)$ THEN
   $w) := v) +$ weight($[v, w]$); prefer($w$) $:= v$
   IF member($w, F$) THEN changekey($w, w), F$)
    ELSE insert($w, w), F$) END IF
  END IF
 END FOR
END WHILE
FOR ALL $u \in V - \{v_1\}$, OUTPUT($u$)) END FOR
END SHORTEST-PATH
```

Note that this code may be easily modified to output the shortest paths as well as their lengths.

First execution: In this execution the SHORTEST-PATH code is used and the abstract data type is implemented with a balanced search tree such as an AVL tree [1], a red-black tree [14], or a b-tree [5]. In addition, an array indexed from 1 to n is used. Each element of this array contains two fields, *InSet*, a boolean, and *Value*, storing the same type as the value used in the ordered pairs. Initially, *InSet* is false for all array elements. The balanced search tree stores the ordered pairs in h and is based on the total order described earlier. For each item number i, the *InSet* field of the i-th array element is true if and only if there is a pair with item number i in the set. The *Value* field of the i-th array element stores the value of the pair with item number i, if there is one in the set. It is undefined if there is no such pair in the set. This array allows rapid execution of operations such as member(i, h) and delete(i, h).

Second execution: This execution also uses the SHORTEST-PATH code, however, a different data structure is used to implement the ADT. We call this data structure an *indexed linked list* and it is depicted in Figure 5. It consists of an array and a doubly linked list. The array is indexed from 0 to n and contains pointers to the elements of the linked list. Except for the first element,

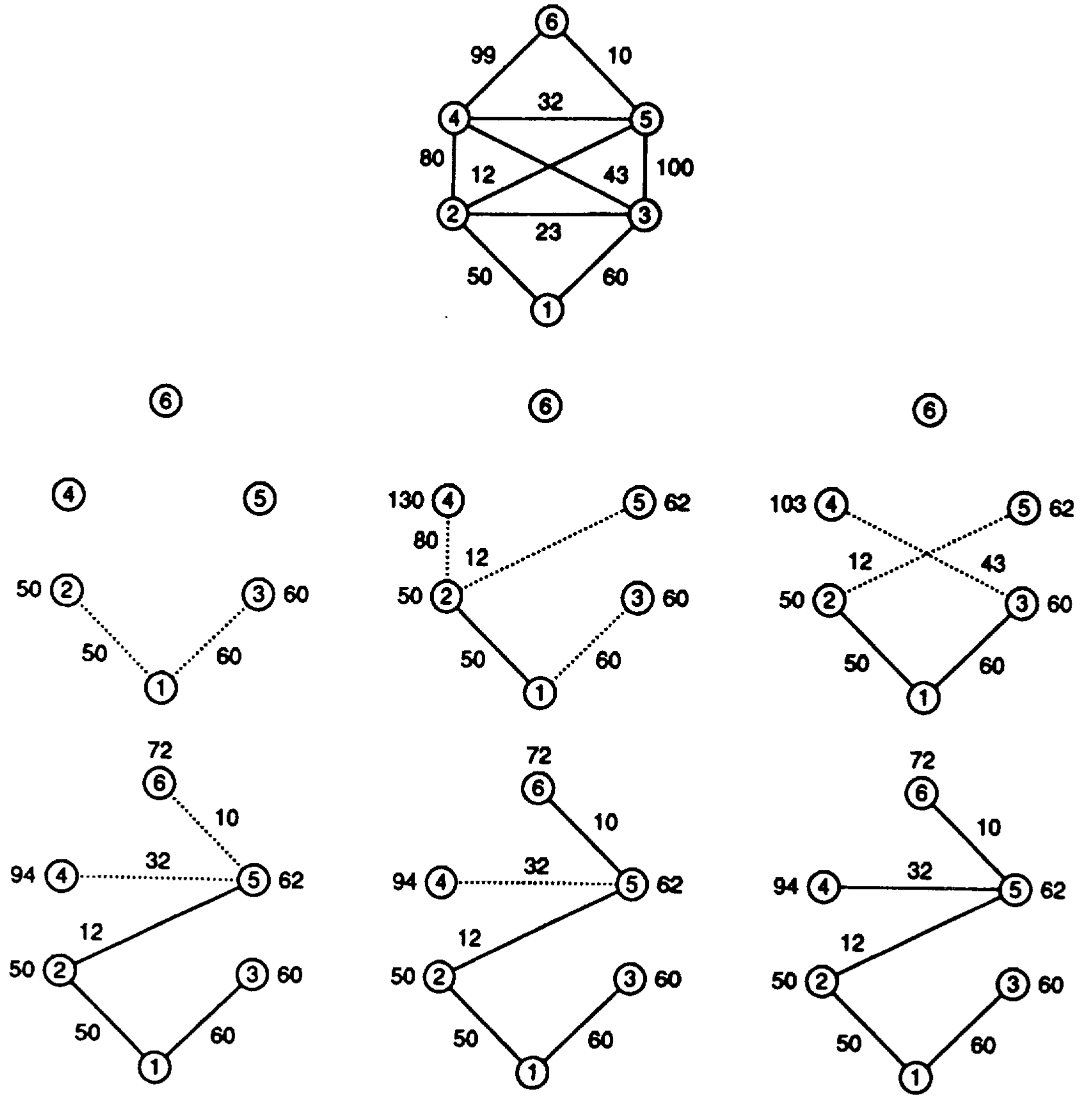

Figure 3: Shortest path example.

Operation	Set of Ordered Pairs	Delete	Trail
insert(2,50)	(2,50)		smallest
insert(3,60)	(2,50),(3,60)		2
deletemin	(3,60)	(2,50)	
insert(4,130)	(3,60),(4,130)		3
insert(5,62)	(3,60),(5,62),(4,130)		3
deletemin	(5,62),(4,130)	(3,60)	
changekey(4,103)	(5,62),(4,103)		5
deletemin	(4,130)	(5,62)	
changekey(4,94)	(4,94)		smallest
insert(6,72)	(6,72),(4,94)		smallest
deletemin	(4,94)	(6,72)	
deletemin		(4,94)	
deletemin		empty	

Table 1: Example of operations and trail.

each element in the list contains a data field storing an ordered pair. The first element stores a special ordered pair (0, “*smallest*″) which is guaranteed to compare less than any other ordered pair. The list is maintained in sorted order based on the total ordering defined above for ordered pairs. This list represents the contents of the set h. The i-th element of the array points to the node containing the ordered pair with item number i, if such an element is present in h. Otherwise the pointer is nil. The 0-th element of the array points to the node containing (0, “*smallest*″) Initially, all pointers are nil except for the 0-th one. Using an ordered list allows us to perform deletemin(h) operations quickly. The array provides rapid random access to the elements. We now describe the implementation of the data structure operations.

insert(i, x, h) Read the next value from the certification trail. This value, call it j, is the item number of the ordered pair that will be the predecessor of (i, x) after it is inserted. To insert this element, we follow the j-th array pointer to the list node containing the pair (j, y). There is one special case, if “smallest” is read from the trail rather than an item number, we follow the 0-th pointer. A new node is allocated and inserted into the list just after the node containing (j, y). The data field of this node is set to (i, x). Finally, the i-th pointer is set to point to the new node. Figure 5 shows the insertion of (5,62) into the data structure, given that the next item on the certification trail is 3. When the insert(i, x, h) operation is performed, some checks must be conducted:

i. The i-th array element must be nil before the operation is performed.

ii. The value j read from the trail must either be “smallest” or be between 1 and n, i.e., it must be a valid item number.

iii. The j-th array element must not be nil before the operation is performed.

iv. The sorted order of the pairs stored in the linked list must be maintained. That is, if the j-th pointer points to (j, y) and its successor before the insertion (ignoring the special case when (j, y) is the last element of the list) is (j', y'), then we must have $(j, y) < (i, x) < (j', y')$.

If any of these checks fails, then the execution halts and "error" is output.

delete(i, h) If the i-th pointer is nil, halt execution and output "error". Otherwise follow the i-th pointer to find the list node containing (i, x). This node is removed from the list. Note that since the list is doubly linked, this is a constant time operation. The i-th pointer is then set to nil. The only condition that must be checked is that the i-th pointer is not nil before the deletion

changekey(i, x, h) To perform this operation, it suffices to perform delete(i, h) followed by insert(i, x, h). The next item for the certification is read when the insert(i, x, h) operation is performed. If any of the conditions required by either of these operations fails, then execution halts and "error" is output.

deletemin(h) The 0-th array pointer is traversed to the list head (which contains (0, "*smallest*")). The pointer to the next node in the list is followed. If there is no next node then "empty" is returned. Otherwise, let (i, x) be the pair stored in that node. We remove the node from the list, set the i-th array element to nil, and return (i, x).

member(i, h) The i-th array pointer is examined. "False" is returned if it is nil, otherwise "true" is returned.

predecessor(i, h) This operation is not used during the second execution of SHORTEST-PATH, but is described for completeness. Follow the i-th pointer to the node containing the pair (i, x). Follow the pointer from that node to the node preceding it on the list (note that this node will always exist). If this is the special node (0, "*smallest*"), return "smallest", otherwise return the item number of the pair stored in this list.

There are two variations to this scheme that are worth noting. First, we could implement a singly linked list rather than a doubly linked list. This eliminates the overhead of maintaining the extra pointer. Note, however, that operations such as delete(i, h) require access to predecessors in order to update the list quickly. This can be provided by modifying the operations delete(i, h), changekey(i, x, h), and predecessor(i, h) so that they output predecessor information to the trail.

The other variation also uses a singly linked list but removes the need for extra certification trail information for delete(i, h) and changekey(i, x, h) operations. It uses the technique of marking a list node for deletion rather than removing them from the list node immediately (the appropriate pointer in the array is still set to nil immediately). When performing other operations, we check for and remove any marked nodes immediately following nodes visited. The total running time is still linear, though insert operations are no longer constant time operations.

Time complexity: In the first execution each data structure operation can be performed in $O(\log(n))$ time where $|V| = n$. There are at most $O(m)$ such operations and $O(m)$ additional time overhead where $|E| = m$. Thus, the first execution can be performed in $O(m \log(n))$ In addition, it provides us with a relatively simple and illustrative example of the use of a certification trail.

In the second execution each data structure operation can be performed in $O(1)$. There are still at most $O(m)$ such operations and $O(m)$ additional time overhead. Hence, the second execution can be performed in $O(m)$ time, i.e., linear time.

Section 6 contains results of timing experiments with this technique.

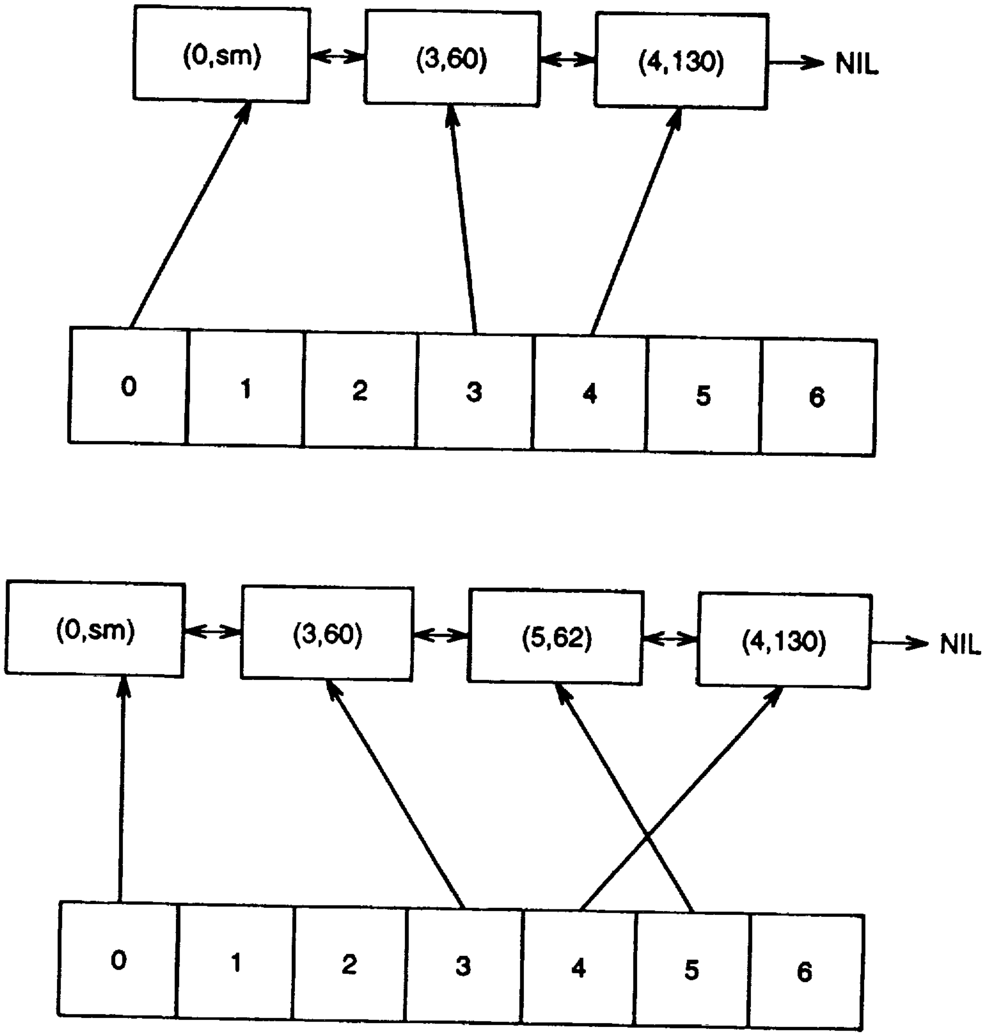

Figure 4: Example of the indexed linked list before and after inserting (5,62)

5.1 Proof of correctness

We wish to prove that the two algorithms given above constitute a certification trail solution to the SHORTEST-PATH problem, i.e., that the functions $F_1(d)$ and $F_2(d,t)$ defined by these algorithms satisfy Definition 2.2. First, we consider the problem of evaluating a sequence of the above data structure operations.

Definition 5.2 Let D be the set of finite sequences of the data structure operations defined above. Let S be the set of finite sequences of answers to data structure operations. Let P be the relation (d,s) where $d \in$ D and $s \in$ S, and s is the sequence of answers resulting from executing the operations d starting with the empty set.

Note that we are examining all finite sequences of data structure operations, not just "legal" ones. That is, may attempt to perform an insertion with an item number already in use, attempt to perform deletion on an item number not being used, etc. We assume that if one of these "illegal" operations is attempted, the operation will output "error" and terminate processing. Thus, we can define the answer sequences for these "illegal" sequences.

Definition 5.3 Let $F_1(d)$ be defined by the result of executing the operations on any of the standard data structures described above, with the insert(i,x,h) and changekey(i,x,h) operations modified to output trail information. Let $F_2(d,t)$ be defined by the result of executing the operations using the indexed linked list implementation described above.

Theorem 5.4 *$F_1(d)$ and $F_2(d,t)$ meet the conditions of Definition 2.2 (that is, $F_1(d)$ and $F_2(d,t)$ constitute a certification trail solution for* P*).*

Proof: We must prove that both conditions of Definition 2.2 are satisfied by these functions.

Part 1: The first condition we must verify is that for all $d \in$ D there exists $s \in$ S and there exists $t \in$ T such that $F_1(d) = (s,t)$ and $F_2(d,t) = s$ and $(d,s) \in$ P. Let $(s,t) = F_1(d)$. The modifications of the data structure operations that produce trail output do not affect how the data structure is maintained. Proofs of correctness for the standard data structures are well known, so we may assume $(d,s) \in$ P. We must demonstrate that $F_2(d,t) = s$.

This may be proven by showing that after each operation that modifies the set h, the elements stored in the indexed linked list (our implementation) correspond to the elements in the set h (the abstract definition). We must also demonstrate that if this relationship is maintained, then correct output is generated by operations that generate output.

To demonstrate this, we show that each operation maintains the following invariants.

i. If the pair (i,x) is in $h \cup (0, \text{"}smallest\text{"})$, then the i-th pointer in the array of pointers points to the list node containing (i,x).

ii. If, for some i, there is no pair in h with item number i then the i-th pointer is nil.

iii. The list nodes are in ascending order.

iv. Every list node is pointed to by some pointer in the array. (Together with the first condition, this implies that it is pointed to by exactly one pointer from the array).

The first two conditions assert that the indexed linked list and the set h contain the same elements (ignoring the special list head element in the linked list). The last two invariants allow us to demonstrate that the linked list operations function correctly.

Clearly each of these conditions is true before the first operation is performed (the set of pairs is empty, all pointers except the 0-th are nil, and (0, "*smallest*") is the only list node).

Assume that the above conditions are satisfied after the first k operations, and that the output generated by any of the first k operations is correct. We claim that the invariants will will remain satisfied after the $(k+1)$-st operation, and that if the $(k+1)$-st operation generates output, it will be correct. Let $s(k+1)$ denote the output produced by the $(k+1)$-st operation (where $F_1(d) = (s, t)$).

Consider each possible operation. For brevity, we omit details for "illegal" operations, i.e., those that violate the precondition of the operation. Similarly, we omit details of the special case of "smallest" being read from the trail.

insert(i, x, h) The trail t contains the item number j of the predecessor of (i, x). Call the predecessor (j, y). By assumption, the i-th pointer is nil before the insert. If not, this operation outputs "error" and execution halts. Since the indexed linked list correctly represents h at this point, this agrees with the result returned by $F_1(d)$, i.e., $s(k+1) =$ "*error*". After the insertion is performed, the i-th pointer is set to the new node containing (i, x), so the first condition is satisfied. No other nodes are added to the list, so the second condition will remain true. The third condition is satisfied since (j, y) is now the immediate predecessor of (i, x). Since no other pointer in the array has been changed, the fourth condition is still true.

delete(i, h) This operation sets the i-th pointer to nil, and removes the node containing (i, x) from the list. This satisfies the second invariant. Deleting a node cannot violate the third invariant. Since no other nodes are removed and no other pointers are changed, the first and fourth invariants remain satisfied.

deletemin(h) By assumption, the nodes are currently in ascending order. Thus, the minimum element in h must correspond to the node following the special list head node, call the pair it contains (i, x). This pair is the correct output for this operation. As with delete, the above four conditions remain true after this node is removed and the i-th pointer set to nil.

changekey(i, x, h) We have implemented changekey(i, x, h) as an insertion followed by a deletion. Since both of those preserve the invariants, changekey(i, x, h) must do so as well.

member(i, h) By assumption, the indexed linked list correctly represents h before this operation, so the output of this operation will be correct. Since this operation does not change the set or the indexed linked list, the invariants remain satisfied.

predecessor(i, h) By assumption, the indexed link list correctly represents h, and furthermore it is currently in sorted order. Thus, the list element preceding the node containing (i, x) is the predecessor. Since this operation changes neither h nor the indexed linked list, the invariants remain satisfied.

This demonstrates that the first condition of Definition 2.2 is satisfied.

Part 2: The second condition is for all $d \in \mathbf{D}$ and for 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 states that if $F_2(d, t)$ is passed an arbitrary trail, it either outputs a correct answer, or it outputs "error". We prove an even stronger condition. Let $t_{correct}$ be the trail returned by $F_1(d)$, i.e., $F_1(d) = (s, t_{correct})$. Then either $t_{correct}$ is a prefix of t, or $F_2(d, t) =$ error.

If $t_{correct}$ is a prefix of t, then we are done. The algorithm describing $F_2(d, t)$ does not examine any part of the trail after $t_{correct}$, so $F_2(d, t) = s$.

If $t_{correct}$ is not a prefix of t, let p be the position at which they first differ. Let O be the number of the operation that uses the trail data at p. Then operation O is either an insert(i, x, h) or changekey(i, x, h) operation. If it is an insert operation, then $t_{correct}$ contains the item number of the predecessor of (i, x). Since t contains a different value, call it j, at this location, the insert(i, x, h) operation will fail one of it's three checks. Either j will not be valid item number, or the j-th pointer will be nil, or the pair (j, y) will not be the predecessor of (i, x). The argument for the changekey(i, x, h) operation is essentially the same.

Thus, the second condition is satisfied.

Therefore, $F_1(d)$ and $F_2(d, t)$ are a certification trail solution to **P**, the problem of evaluating data structure operations. ∎

Definition 5.5 Let **D** be the set of finite graphs $G = (V, E)$ with edge weights consisting of positive integers. Assume the indices are numbered 1 through n. Let **S** be the set of finite ordered tuples of positive integers. Let **P** be the relation that associates each graph with the tuple consisting of the minimum path lengths to each vertex. Let $SP_1(d)$ be the function defined by the SHORTEST-PATH algorithm with the data structure defined for the first execution. Let $SP_2(d, t)$ be the function defined by the SHORTEST-PATH algorithm using the indexed linked list implementation.

Corollary 5.6 *$SP_1(d)$ and $SP_2(d, t)$ constitute a certification trail solution for* **P**.

Proof: If $SP_1(d) = (s, t)$, then the correctness of Dijkstra's algorithm implies that $(d, s) \in$ **P**. The algorithms that compute $SP_1(d)$ and $SP_2(d, t)$ are the same except for data structure implementation. Theorem 5.4 implies that if these algorithms generate the same data structure operations, then the same sequence of answers will be generated. Thus, to demonstrate that $SP_2(d, t) = s$, it must be shown that the same sequence of data structure operations is generated by both algorithms. Examination of SHORTEST-PATH indicates that the k-th data structure operation to be performed is dependent only on the input and the result of previous data structure operations. For example, at line 9, either an insert(i, x, h) or a changekey(i, x, h) is performed, depending on the result of a member(i, h) operation. The input graph d is identical for both algorithms, thus the first data structure operation performed must be the same. Assume that the first k operations performed by both algorithms are identical. Then, by Theorem 5.4, the answers to those operation will be the same. Since the $(k + 1)$-st operation depends only on the input and the results of the previous k operations, it must also be the same for both algorithms. Therefore the same sequence of data operations is performed in both algorithms, so $SP_2(d, t) = s$.

The proof that the second condition holds is the same as for Theorem 5.4. Either the input trail t contains the "correct" trail as a prefix, or one of the data structure operations will fail, resulting in an "error" output. ∎

One point has been glossed over in the above proof. In the SHORTEST-PATH algorithm results of deletemin(h) are not output nor are they stored in the certification trail. It might be possible for incorrect answers to be returned by deletemin(h) operations while still producing correct shortest paths and lengths. The second execution of the SHORTEST-PATH algorithm will not detect this since the correct output is produced. By proving that the answers to deletemin(h) operations are the same, we have proven more than strictly required.

6 Experimental Data on Certification Trails

We have performed extensive timing experiments on several basic and well-known problems, including the ones described in this paper. Algorithms for solving these problems were implemented, both with and without the use of certification trails. Timing data was collected on both the certification trail solutions and the basic solutions. The following tables summarize these results.

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
5000	0.61	0.62	0.07	8.73	43.62
10000	1.33	1.34	0.14	9.56	44.54
25000	3.68	3.68	0.36	10.22	45.12
50000	7.68	7.74	0.71	10.75	44.94
100000	16.23	16.30	1.43	11.35	45.39
200000	33.93	34.37	2.84	11.94	45.16

Table 2: Convex Hull

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
10000	0.28	0.30	0.04	7.00	39.29
50000	1.80	1.90	0.19	9.47	41.94
100000	3.96	4.08	0.41	9.66	43.31
500000	23.95	24.69	2.14	11.19	43.99
1000000	50.23	51.57	4.38	11.47	44.31

Table 3: Sort

Size	Basic Algorithm	First Execution (Also Generates Trail)	Second Execution (Uses Trail)	Speedup	Percent Savings
100,1000	0.04	0.05	0.02	2.00	12.50
250,2500	0.15	0.16	0.06	2.50	26.67
500,5000	0.31	0.33	0.11	2.82	29.03
1000,10000	0.70	0.76	0.23	3.04	29.29
2000,20000	1.58	1.67	0.45	3.51	32.91
2500,25000	2.06	2.15	0.55	3.75	34.47

Table 4: Shortest Path

The timing information was gathered on Sun SPARCstation ELC with 16MB of RAM. The system was run as a standalone machine in single user mode during timing experiments.

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 Algorithm* 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 *First Execution* column gives the execution time of the algorithm in producing the output with the additional overhead of generating the certification trail.

The *Second Execution* column gives the execution time of the algorithm in producing the output while using the certification trail.

The *Speedup* column is the ratio of the run times of the Basic Algorithm and the Secondary Execution. One reason this figure is important is that it is possible for the two algorithms to run in different environments (different hardware, programming language, etc). A high speedup indicates that less powerful hardware or a higher level language (with associated overhead) may be sufficient for the second execution.

The *Percent Savings* 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. The time required for a 2-version programming approach was estimated by doubling the time reported in the Basic algorithm. This assumes that both versions take approximately the same amount of time to execute.

In addition to the tables, the timing information for the convex hull algorithm is plotted in Figure 5. Plots for the other two examples are similar.

Examination of the data collected for the convex hull algorithm indicates that:

- The overhead in generating the certification trail is very small, less than 2% of the running time of the basic (no certification trail) algorithm.

- The second execution is very fast, achieving an order of magnitude speedup for larger input sizes. This suggests that a single "second algorithm" process could easily handle the output generated by several "first algorithm" processes running in parallel. Alternately, the high speedup would allow the second execution to be run on lower performance (and hence less expensive) hardward. Finally, the large speedup and reduced code complexity may make it possible to take advantage of a formally verifiable language (which may require significant overhead) in implementing the second algorithm.

The data for sorting indicates that the certification trail also requires very low overhead and results in a large speedup. For the shortest path problem the overhead is still very low, and the speedup, while not as dramatic as for the first two problems, is still quite respectable.

7 Comparison With Other Techniques

The certification trail approach shares similarities with other valuable fault tolerance and fault detection techniques that have been previously proposed and examined, but in each case there are significant and fundamental distinctions. These distinctions are primarily related to the generation and character of the certification trail and the manner in which the secondary algorithm uses the certification trail.

First consider the important and useful technique called N-version programming [9, 3]. When using this technique N different implementations of an algorithm are independently executed with subsequent comparison of the resulting N outputs. There is no relationship among the executions of the different versions of the algorithms other than that they all use the same input; each algorithm is executed independently without any information about the execution of the other algorithms. In marked contrast, the certification trail approach allows the primary algorithm to generate a trail of information which can be read by the secondary algorithm. The advantages of utilizing this additional information are shown in the body of this paper. In effect, N-version programming can be thought of relative to the certification trail approach as the employment of a *null trail.*

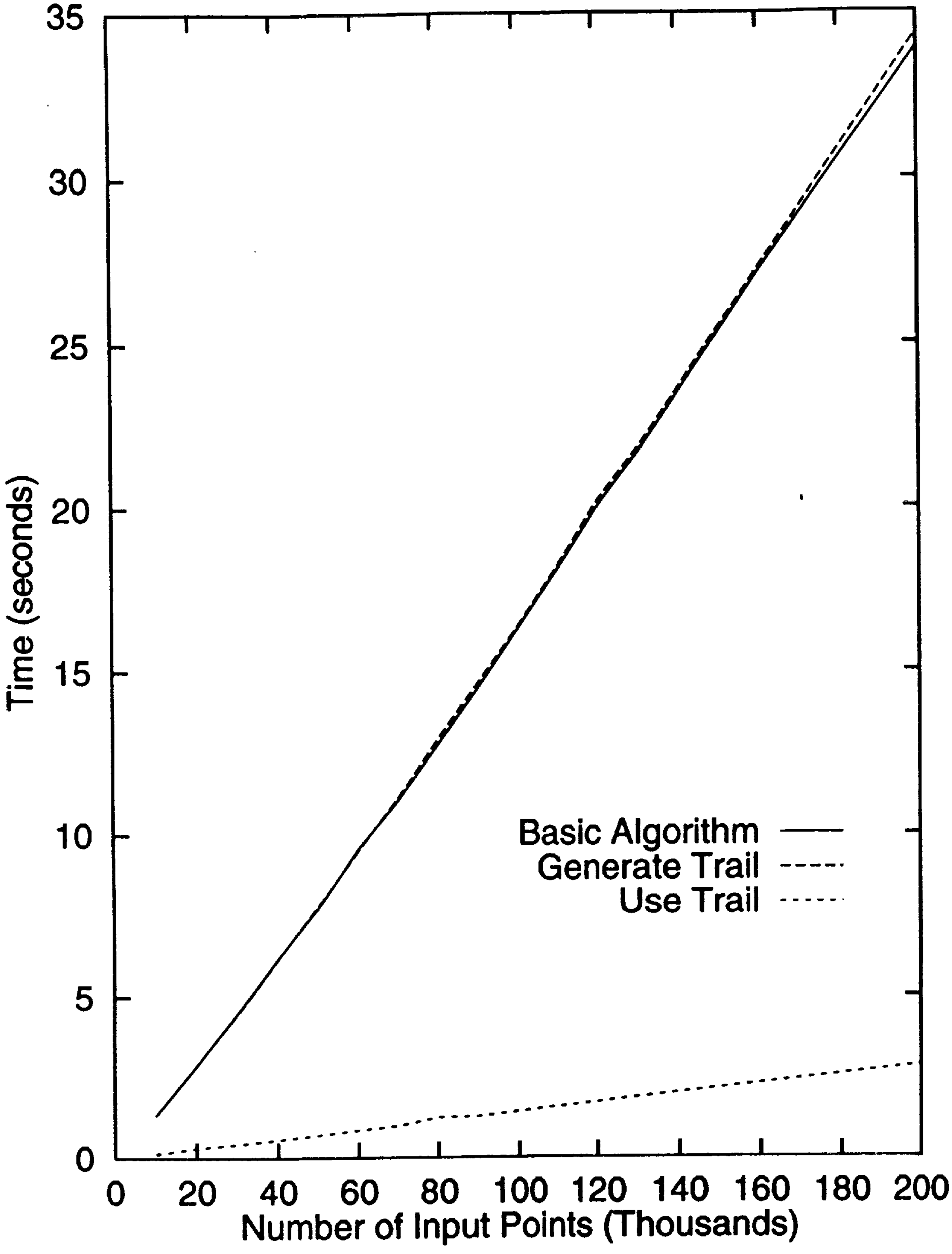

Figure 5: Convex Hull Run Times.

Another valuable technique, known as the recovery block approach [2, 18, 21], was proposed by Randell. It uses acceptance tests and alternative procedures to produce what is to be regarded as a correct output from a program. When using recovery blocks, a program is viewed as a being structured into blocks of operations, which after execution yield outputs which can be tested in some informal sense for correctness. The rigor, completeness, and nature of the acceptance test is left to the program designer, and many of the acceptance tests that have been proposed for use tend to be somewhat straightforward [2]. When using certification trails it is clearly possible to combine the second execution and the comparison test to yield a program which certifies the correctness of the output of the first execution. Unlike an acceptance test this certifier must satisfy strict formal properties of correctness. Also note that the certification trail technique emphasizes the capability of generating additional data to ease the certifying process and does not rely solely on data which would normally be computed. It should be possible to fruitfully combine the ideas of recovery blocks and certification trails.

Algorithm-based fault tolerance [15, 17, 19] uses error detecting and correcting codes for performing reliable computations with specific algorithms. This technique encodes data at a high level and algorithms are specifically designed or modified to operate on encoded data and produce encoded output data. Algorithm-based fault tolerance is distinguished from other fault tolerance techniques by three characteristics: the encoding of the data used by the algorithm; the modification of the algorithm to operate on the encoded data; and the distribution of the computation steps in the algorithm among computational units. The error detection capabilities of the algorithm-based fault tolerance approach are directly related to that of the error correction encoding utilized. The certification trail approach does not require that the data to be executed be modified nor that the fundamental operations of the algorithm be changed to account for these modifications. Instead, only a trail indicative of aspects of the algorithm's operations must be generated by the algorithm. As seen in Section 6, the production of this trail does not add significant overhead. Moreover, any combination of computational errors can be handled.

Recently, Blum and Kannan [6] have defined what they call a *program checker*. This interesting work has been followed by a burst of activity in this general area [12, 7, 25, 8, 4]. Each of these papers, however, describes work which differs significantly from the work we present. A program checker is an algorithm which checks the output of another algorithm for correctness. An early example of a program checker is the algorithm developed by Tarjan [23] which takes as input a graph and a supposed minimum spanning tree and indicates whether or not the tree actually is a minimum spanning tree.

The Blum-Kannan program checking method differs from the certification trail method in two important ways. First, the checker is designed to work for a problem and not a specific algorithm. That is, the checker design is based on the input/output specification of a problem and no assumptions are made about the method being used to solve the problem. Because of this the algorithm which is being checked is treated as a black box. It can not be altered nor can its internal status be examined and exploited. In the certification trail approach the algorithm being checked is not treated as a black box. Instead, the algorithm can be modified to generate additional information (i.e., the certification trail) which is considered to be useful in the checking/verification process. By exploiting this capability it is sometimes possible to design certification trail solutions which allow faster checking than Blum-Kannan program checkers. Of course, these faster solutions are more specialized than the Blum-Kannan checkers which are guaranteed to work for any algorithm which solves the original problem. We believe that the added speed often outweighs the disadvantage of specialization.

The second important difference concerns the number of times that the program which is being checked is executed. In the Blum-Kannan approach the program may be invoked a polynomial number of times. In the certification trail approach the program is run only once. Thus, the overall time complexity of the checking process can be significantly larger for Blum-Kannan checkers.

A third less important difference stems from the fact that Blum-Kannan checkers are defined in a more general probabilistic context. Certification trails are currently defined only for deterministic programs and checkers. However, it is clearly possible to define them in the more general probabilistic context.

Other work has been done to extend the ideas of Blum-Kannan to give methods which allow the conversion of some programs into new programs which are self-testing and self-correcting [12, 7]. However, these methods are also based on treating programs as black boxes and thus have limitations similar to Blum-Kannan program checkers. A recent paper by Blum et al. [8] concerns checking the correctness of memories and data structures. The results described in that paper differ from our work using abstract data types in one central way. The checkers that they design are tightly constrained in memory usage. Typically, they use only $O(\log(n))$ storage to check data structures of size $O(n)$. Our results do not place space constraints on the algorithm used to certify the data structure. Without a space constraint we are able to certify abstract data types such as priority queues which are more complex than the data structures that they check, i.e., stacks and queues. Also, we are able to achieve a speed up in the checking process and they are not.

Babai, Fortnow, Levin and Szegedy [4] present methods which appear to allow remarkably fast checking, i.e., in polylogarithmic time. Their approach has some similarities to the methods we propose. Both methods modify original algorithms to yield new algorithms which output additional information. We refer to this additional information as a certification trail and they refer to this information as a *witness*. In our case we are interested in modified algorithms which have the same asymptotic time complexity as the original algorithm. Indeed, the modified algorithm should be slowed down by at most a factor of two. In [4] the modified algorithm is slowed down by more than any fixed multiplicative factor. Specifically, if the original algorithm has a time complexity of $O(T)$ then the modified algorithm has a time complexity of $O(T^{1+\epsilon})$. Note that in practice the ϵ cannot be too small because its inverse appears in the exponent of the checker time complexity. Another difference between our methods is the fact that their method requires that the input and output be encoded using an error-correcting code. The encoding process takes $O(N^{1+\epsilon})$ time for strings of length N. However, many of the checkers we have developed take only linear time so the cost of simply preparing to use their method appears to be too great in some cases. It is also necessary to decode the output after the check. Lastly, we note that Fortnow has stated that their result is currently not practical [24].

8 Generalization and Future Research Areas

The experimental timing data on certification trails indicates that this technique is of great practical value as well as of theoretical interest. Furthermore, the techniques illustrated are applicable to a wide range of problems, especially the certification of Abstract Data Types described in the shortest path example. There are many areas of interest for future exploration, a few of which are described below.

8.1 Certified Data Structure Libraries

It is apparent that the certification trail technique described for the SHORTEST-PATH program may be used for a variety of problems. Since the certification trail is produced and used by abstract data type operations, the technique may be used with any algorithm that can be implemented in terms of those abstract data types. Creating a library of such "certified data types" enables programmers to create fault tolerant programs without having to be familiar with the certification trail technique. Object oriented programming appears to be well suited to this task.

A possible objection to this is that it provides fault detection only for the data structure implementation, since the surrounding code is simply reused. Furthermore, the data structure implementation is likely to come from library code, and hence be highly reliable. In answer to this note that:

- In many algorithms, the code using the data structure is much simpler than the code implementing the data structure.
- Although the example above illustrated reuse of using the data structures, it is certainly possible for this code to be developed separately for the first and second execution programs.
- Errors are often found even in code that has been in use for a long period of time. The added confidence of using this technique may be desirable even for library code.
- Even if the library code is highly reliable, the certification trail can be helpful in detecting errors caused by hardware problems.
- Library code may have to be tuned or even rewritten to meet for a particular application or environment, partially negating the claim of using well-tested code.

Even if fault detection is not an issue, the certification trail technique is useful during program testing and debugging. Input may be automatically generated and processed. If the output of the first and second executions differ or an error is otherwise flagged, the input set is flagged. This reduces the need to otherwise compute output for selected input and enables both more and larger sets of input to be processed. 2-version programming may be used during debugging in a similar manner, however certification trails have the advantage of reduced overhead, allowing more test cases to be run, a reduction in the hardware required for testing, or both.

8.2 Almost-concurrent execution of the certification trail

In the above discussion and examples, the certification trail programs have been executed serially, i.e., we do not run the second execution until after first execution completed. Actually, except for sorting, the two executions in the examples above can be run almost-concurrently. The "second" execution simply reads the information from the certification trail as it becomes available. The two programs will finish nearly simultaneously, the difference being in the time after the last element is read from or written to the certification trail.

8.3 Continuing after an error

A possible extension to the use of certification trails is to attempt to continue the second execution after an error is detected. Consider the shortest path example using abstract data types. In that example, the second execution used an indexed linked list that performed each operation in constant time by using the certification trail from the first execution. Suppose that an error had been detected during the second execution. Rather than simply aborting, it may be possible to continue execution. This could be done by

- Reorganizing the existing set into some other data structure (such an AVL tree, red-black tree, etc.) that allows efficient operation without a certification trail.

- Continuing to use the indexed linked list and ignoring the rest of the certification trail. Note that this would result in some operations requiring more time.
- Continuing to use the indexed linked list and attempting to use the certification trail for future operations. This may be possible if the error that occurred has sufficiently "local" effect. For example, if part of a tree structure is corrupted during the first execution, it is still possible that operations involving other parts of the tree will be performed correctly.

On a related topic, research has been done on "self-correcting" data structures in which enough redundancy is built into a data structure so that it may be reconstructed if part of it is corrupted. Using certification trails with such structures could provide an efficient detector for corruption of the data structure.

References

[1] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[2] Anderson, T., and Lee, P., *Fault tolerance: principles and practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

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

[4] Babai, L., Fortnow, L., Levin, L., and Szegedy, M., "Checking computations in polylogarithmic time, " *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pp. 21-31, 1991.

[5] Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[6] Blum, M., and Kannan, S., "Designing programs that check their work", *Proceedings of the 1989 ACM Symposium on Theory of Computing*, pp. 86-97, ACM Press, 1989.

[7] Blum, M., Luby, M., and Rubinfeld, R., "Self-testing/correcting with applications to numerical problems," *Proceedings of the 22nd ACM Symposium on Theory of Computing*, pp. 73-83, 1990.

[8] Blum, M., Evans, W., Gemmell P., Kannan, S., and Naor, M., "Checking the correctness of memories," *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science* pp. 90-99, 1991

[9] Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *Digest of the 1978 Fault Tolerant Computing Symposium*, pp. 3-9, IEEE Computer Society Press, 1978.

[10] Cormen, T. H., and Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms* McGraw-Hill, New York, NY, 1990.

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

[12] Gemmell, R., Lipton, R., Rubinfeld, R., Sudan, M., and Wigderson, A., "Self-testing/correcting for polynomials and for approximate functions," *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pp. 32-42, 1991.

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

[14] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

[15] Huang, K.-H., and Abraham, J., "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. on Computers*, pp. 518-529, vol. C-33, June, 1984.

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

[17] Jou, J.-Y. and Abraham, J. "Fault tolerant FFT networks," *Dig. of the 1985 Fault Tolerant Computing Symposium*, pp. 338-343, IEEE Computer Society Press, June, 1985.

[18] Lee, Y.H. and Shin, K.G., "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," *IEEE Trans. Comput.*, vol. C-33, pp. 113-124, Feb. 1984.

[19] Nair, V., and Abraham, J., "General linear codes for fault-tolerant matrix operations on processor arrays," *Dig. of the 1988 Fault Tolerant Computing Symposium*, pp. 180-185, June, 1988.

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

[21] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

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

[23] Tarjan, R. E., "Applications of path compression on balanced trees", *J. ACM*, pp. 690-715, Oct., 1979.

[24] Paul Wallich, "Crunching Epsilon," *Scientific American*, pp. 22-24, Jan., 1993

[25] Andrew Chi-Chih Yao, "Coherent Functions and Program Checkers," *Proc. 22 ACM Symp. of Theory of Computing*, pp. 84-94.