Efficient Algorithms for the Temporal Precedence Problem

D. Ranjan, E. Pontelli, G. Gupta
Dept. of Computer Science
New Mexico State University
{dranjan,epontell,gupta}@cs.nmsu.edu

Keywords: Algorithms, Dynamic Data Structures, Complexity, Pointer Machines.

1 Introduction

In this paper we study the complexity of what we call the *Temporal Precedence ($\mathcal{TP}$) Problem* on pointer machines. Intuitively, the problem is to manage the dynamic *insertion* of elements, with the ability of determining, given two elements, which one was inserted first. We are not aware of any study regarding the complexity of this problem on pointer machines.

The $\mathcal{TP}$ problem is related to a very concrete problem that arises in parallel implementation of logic programming languages [9]. In such parallel implementation, given a dynamic tree, capable of growing and shrinking at the leaves, and given two nodes, we need to detect whether the first node is the leftmost leaf in the subtree rooted at the second node. This operation is fundamental to allow correct ordering of execution of order-sensitive operations (e.g., side-effects). We have observed that this problem is equivalent to the $\mathcal{TP}$ problem [9]. We will consider the *Pure Pointer Machines (PPM)* as model of computation [5, 14, 3, 6, 13]. Formally, the $\mathcal{TP}$ problem can be defined as follows: given a collection of distinct elements $a_1, a_2, \ldots, a_n$, we want to carry out a sequence of operations of the following two types:
insert(a_i) which marks the element a_i as inserted
precede(a_i, a_j) returns true if a_i was inserted before a_j

The operations are to be carried out *on-line*, i.e., each instruction has to be completed before the next one is considered. For the sake of simplicity, we assume that each sequence of operations is correct (e.g., a **precede** is applied to elements that have already been inserted), and contains exactly n **insert** operations—thus, all the elements in the sequence are inserted at some point of time. The problem is very basic, as it captures the fundamental concept of *ordering* elements, and is closely related to the problem of *counting*. Note that in the $\mathcal{TP}$ problem we do not assume anything about the structure of the input data a_i and for all practical purposes the data are abstract entities.

2 Pointer Machines

A *Pure Pointer Machine (PPM)* consists of a finite but expandable collection R of records, called *global memory*, and a finite collection of registers. Each record is uniquely identified by an *address*, and let $\mathcal{N}$ be the set of all addresses. *nil* is a special element of $\mathcal{N}$ used to denote an invalid address. Each record is a collection of c fields; for simplicity we assume that each record can contain only an address[1]. Each PPM is supplied with a finite collection of registers, each capable of storing one address. PPM programs are finite numbered sequences of instructions. The instructions allow transfer of addresses between registers and/or record fields and selection of unused records. The only control instruction allowed is the conditional jump—if $r_i = r_j$ goto k where k identifies one instruction in the program and r_i, r_j are two of the machine registers.

An overview of the concept of Pointer Machines and references to relevant results are presented in [1].

3 Complexity of $\mathcal{TP}$ on PPMs

It is easily observed that the $\mathcal{TP}$ problem can be solved in $O(1)$ time on the RAM. The essential idea is to "time-stamp" the elements during insertion. The **precede** operation then is just the comparison of the time-stamps of the two elements.

Our focus here is to study the problem in the context of pointer machines. In this section we present upper bounds for the $\mathcal{TP}$ problem on pointer machines. We are not aware of previous results for this particular problem. To provide an intuition as to how we

[1]The more general notion of PPM allows also for data fields, whose content can only be copied and compared for equality. Such extension does not add anything to the considerations in this paper.

achieved our results, we present several successively better schemes, finally ending with the best result.

A naive approach to the problem easily leads to a complexity of $O(1)$ for the **insert** operation and $O(n)$ time for the **precede** operation. Essentially, one can maintain the inserted elements as a simple linked list. The **insert** operation requires simply the creation of a new record in the list, which is appended to the tail. The **precede** operation requires traversing the list, in order to detect the relative position of the two elements. This has a worst–case time complexity $O(n)$.

An alternative solution can be obtained by using *time-stamps*. Each element can be labeled with the time of insertion. The **precede** operation is then reduced to the comparison of the labels of the two elements in question. Since pointer machines are not supplied with any form of arithmetic, time stamps have to be implemented using records. One possible way is to code the numbers in binary format, representing each digit using a record, and each digit's value as the value of the first field of the record. Using this approach, a **precede** operation requires comparison of two labels, whose length is $O(\lg n)$. A non-constant cost is also incurred during the **insert** phase. The cost of creating a label is proportional to the length of the label—as we need to either traverse or copy an arbitrarily large part of the previously created label. Thus the insertion operation has itself an individual cost of $O(\lg n)$.

3.1 A Simple Tree-Based Scheme

An improvement over the $O(\lg n)$ time cost per operation presented in the previous section can be obtained by switching from a linear list structure (explicitly represented or implicit in the growing time labels) to a tree structure. We name this approach the *Binary-Tree scheme*. Let us start from the simple case in which a binary tree structure is employed. The elements which are inserted are placed in a binary tree which is expanded level by level from left to right. Each node has a pointer to the node immediately to its right and to its parent (see Figure 1). The insertion of elements can be done in $O(1)$ time easily by maintaining a pointer to the last inserted node, a pointer to the last node of the last level that is full and a pointer to the first node in the current level. The operation **precede**(u, v) can now be implemented using the following idea – **precede**(u, v) should return true iff one of the following three conditions holds:

(i) u is the root and v is not;

(ii) v is immediately to the right of u;

(iii) u'=parent(u) and v'=parent(v) exist and **precede**(u', v') is true.

It is clear that a straightforward implementation of **precede** gives us an $O(\lg n)$ scheme because the depth of the binary tree is bounded by $\lceil \lg n \rceil$. Figure 2 gives the pseudo-code for the **insert** operation; for the sake of simplicity we are also assuming a model with at least four pointers per record (without loss of generality). The code assumes[2] that (see Figure 2)

- register r_0 points to the element to be inserted;
- each record n uses n^1 to point to the parent (*nil* for the root), n^2 to point to *nil* iff this record is the left child of its parent, n^3 to point to the node on its right, and n^4 to point to the object record.
- register r_1 points to the rightmost node in the last full layer of the tree;
- r_2 points to the leftmost node in the current layer of the tree;
- register r_3 points to the last node created;

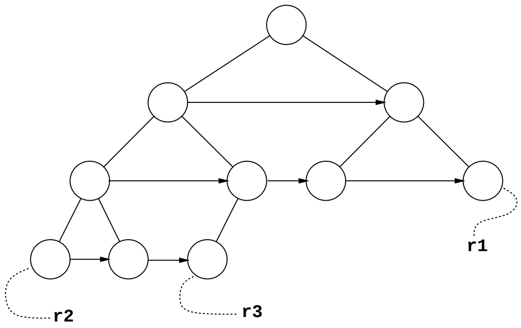

Figure 1: Binary Tree Representation

insert:
  temp := new;
  $\text{temp}^4 := r_0$;
  if $(r_3^1 = r_1$ and $r_1^2 \neq nil)$ then
    $\text{temp}^1 := r_2$; $\text{temp}^2 := nil$;
    $r_1 := r_3$; $r_2 :=$ temp;
  else if $(r_3^2 = nil)$ then
      $\text{temp}^1 := r_3^1$; $\text{temp}^2 :=$ temp;
      $r_3^3 :=$ temp;
    else
      $\text{temp}^1 := (r_3^1)^3$;
      $\text{temp}^2 := nil$; $r_3^3 :=$ temp;
    endif
  endif
  $r_3 =$ temp;

Figure 2: Pseudo-code for **insert**

[2] r_i denotes the i^{th} register of the pointer machine, while n^j indicates the j^{th} field of record n.

```
precede:
    found := false;
    while (not found)
        if (r_0 = r_1^3) then
            output "FALSE"
            found := true;
        elseif (r_0^3 = r_1) then
            output "TRUE"
            found := true;
        elseif (r_0 = root) then
            output "TRUE"
            found := true;
        elseif (r_1 = root) then
            output "FALSE"
            found := true;
        else r_0 := r_0^1; r_1 := r_1^1;
        endif
    endwhile
```

Figure 3: Operation **precede**

As can be seen from the code in figure 2, the cost of each insertion operation is $O(1)$. The code in figure 3 describes the operation **precede** (r_0 and r_1 contain the pointers to the two arguments of **precede**).

Theorem 3.1 (Complexity of Binary-Tree) *In the Binary-Tree,* **insert** *has a worst–case time complexity of* $O(1)$ *while* **precede** *has a worst–case time of* $O(\lg n)$*, where* n *elements are inserted.*

3.2 Nested Trees

In the scheme presented above we use a binary tree for organizing the elements that are progressively inserted. Clearly, one can think of generalizing this to schemes using d-ary tree structure for any d. The order of complexity will not change as long as d is a constant; d steps will be required if the compared nodes have the same ancestor, while $\lg_d n$ will be the maximum number of levels that can be traversed before getting to the root of the tree. The **insert** operation is still $O(1)$, whereas the **precede** operation will take time $O(d + \lg_d n)$ (in the worst case). Intuitively, this scheme partitions the set of n inserted elements into groups of size d; these groups are themselves organized into groups of size d and so on. Identifying the ordering between two elements reduces to identifying the "smallest" class which contains both the element.

The next step in the generalization is to allow the degree of the nodes in the tree to vary. We can think of generalizing in two directions: keeping the degree at each node the same but making it a function of n or letting the degree vary in some systematic way. For the first type of generalization, an *a priori* knowledge of the number of elements to be inserted seems necessary. We discuss some schemes with this type of generalization in subsection 3.5. The following subsections will discuss schemes where no *a priori* knowledge of n is assumed.

Let us start by considering trees in which the degree of the nodes at each level is twice the degree of the nodes at the previous level. Thus, if we indicate with $d(i)$ the degree of the nodes at depth i, then we have the recurrence

$$d(i + i) = 2d(i) \qquad d(0) = 2$$

Figure 4 shows the first levels of this tree. We can easily

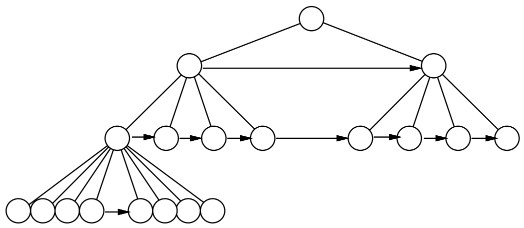

Figure 4: Tree for $d(i + 1) = 2d(i)$

compute the number of nodes $t(i)$ that are present in the i^{th} level of the tree: $t(i+1) = t(i)d(i)$ and $t(0) = 1$. By solving the recurrences we have $d(i) = 2^{i+1}$ and

$$t(i + 1) = t(i)d(i) = t(i)2^{i+1} = 2^{\frac{(i+1)(i+2)}{2}}$$

The increasing number of nodes present at each level requires a smarter approach in handling the **precede** operation in the case where the two nodes have the same direct ancestor. In fact, if the two nodes considered are at level i and they have the same direct ancestor, then they belong to a collection of $d(i-1) = 2^i$ elements. Thus, a linear scan of the siblings as done in the previous case (where only two siblings were present) becomes now inefficient.

Let us start by estimating the number of levels (i.e., the depth) that this sort of tree is going to have for a sequence of n **insert** operations. From the fact that $t(i) = 2^{\frac{i(i+1)}{2}}$ it follows that the depth of the tree is no more than $\lceil\sqrt{2\lg n}\rceil$ which is $O(\sqrt{\lg n})$.

Now, we can derive what the worst case situation will be for the **precede** operation. This will arise when scanning linearly the largest group of nodes having the same direct ancestor. Since at level l a group of 2^l nodes will share the same direct ancestor, the worst–case linear scan will require $O(2^{\lceil\sqrt{2\lg n}\rceil})$ steps. Thus the cost of **precede** in case the two nodes being compared were from one of these large groups would be excessive. Note that in the completely dual case, where the two nodes being compared never have the same ancestor, the cost is considerably lower, being bounded by the depth $O(\sqrt{\lg n})$ of the tree.

The cost of **precede** can be reduced by repeating the same kind of tree-structuring in each group of nodes having the same ancestor, and doing this recursively. This structure may be imposed through another set of pointers (it helps to think of these as different color pointers). The innermost level of nesting will be characterized by having a tree-structure, in which no more than two nodes share the same direct ancestor[3]. Figure 5 illustrates a simple example (for the sake of clarity we have omitted from the picture the links between nodes with the same ancestor).

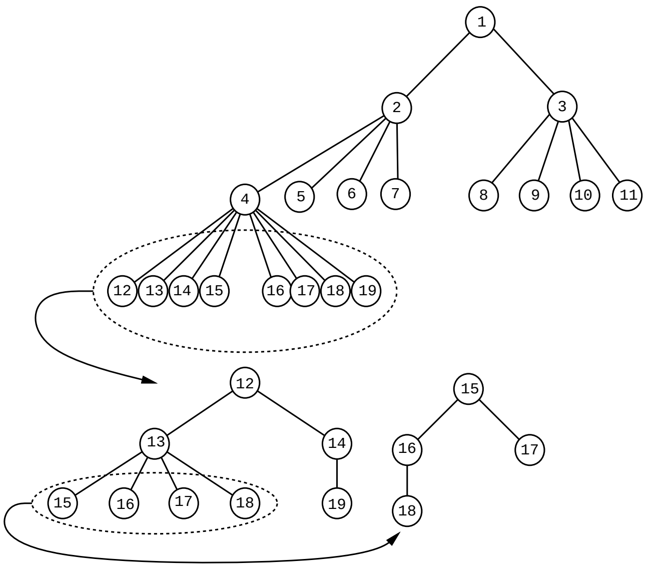

Figure 5: Nesting of Trees

The next observation that we need to make regards the maximum level of nesting of trees (or the number of different colored pointers needed) in this scheme. From previous considerations, the maximum number of nodes present in a group with the same ancestor is $2^{\lceil\sqrt{2\lg n}\rceil}$. Furthermore, for a tree with m nodes, the depth of the tree is $\sqrt{2\lg m}$. Thus, the first time we create a nesting we are going to obtain a tree with a depth of no more than

$$(\lceil\sqrt{\lg 2^{\lceil\sqrt{2\lg n}\rceil}}\rceil) \leq 2(\lg n)^{\frac{1}{4}}$$

In fact we can show that at the kth level of nesting of trees, we are going to have trees with depth at most $2(\lg n)^{\frac{1}{2^k}}$. Since we can stop further nesting once there are no more than $O(\sqrt{\lg n})$ nodes with the same ancestor, then we can estimate the maximum depth of nesting k by solving the inequality

$$2^{2(\lg n)^{\frac{1}{2^k}}} < (\lg n)^{\frac{1}{2}}$$

[3]Actually we could stop the nesting earlier, as soon as there are at most $O(\sqrt{\lg n})$ nodes with the same ancestor.

which leads to the result $k = O(\lg\lg\lg n)$. A similar order of magnitude is achieved even if we impose the simpler constraint $2^{(\lg n)^{\frac{1}{2^k}}} < H$ for a constant H.

One important thing to note here is that we have a non-constant (k) number for nesting levels; this implies that, in order to maintain the nesting information for each node, our records must have a non-constant number of fields. This is not allowed in the PPM. But we can overcome this by maintaining the k fields as a linked list, and paying a penalty of $O(k)$ for each field access. In the specific case of nested-trees the penalty is $O(\lg\lg\lg n)$. The implementation of the scheme is based on the following steps:

insert: whenever a node is inserted in the tree structure, it needs to be enriched with the appropriate pointers to allow identifying its position within each nested tree. The intuition is that each node u is supplied with a list of pointers, $n_1, \ldots, n_s$, where n_i is supposed to indicate the ancestor of u with respect to the ith level of nesting (i^{th} colored pointer). The value of s can reach, in the worst case, a value equivalent to the maximum number of nesting of tree structures, i.e., $O(\lg\lg\lg n)$. The node has to be inserted at each level of nesting (i.e., for each color a pointer may have to be set appropriately). Hence, in the worst case it may require modifying $O(\lg\lg\lg n)$ pointers. Insertion in each level of nesting requires modification of a constant number of pointers in a constant number of records (a generalization of the $O(1)$ insertion presented in the Binary-tree case)—with a worst–case time complexity $O(\lg\lg\lg n)$ (due to the penalty for having records with a non-constant number of pointers). Detection of completion of a group, level, or tree (e.g., determining that $2d(i)$ children have been added to a node) can be realized by moving pointers in the trees (e.g., scanning twice the previous level of the tree).

It is intuitively clear that the cost of an insertion will be $O((\lg\lg\lg n)^2)$ in the worst case ($O(\lg\lg\lg n)$ nodes to modify, with a cost of $O(\lg\lg\lg n)$ for each modification).

precede: the search procedes very similarly to the scheme described in the previous section. Given the two nodes u and v to be compared, we first check if one is immediately to the right of the other in which case the answer is obvious. Otherwise, the parents of the two nodes at the first level of nesting (i.e., their parents in the outermost tree) are compared. If they are different then **precede** is called recursively on the parents. Otherwise, the attention is switched to the next level of nesting (the next set of colored pointers) and the same process is repeated. One can easily argue that in the worst case **precede** performs at most

$O(\sqrt{\lg n}+\lg\lg\lg n) = O(\sqrt{\lg n})$ steps "walking-up" the trees at any level of nesting. Hence the complexity of **precede** in the worst case is $O(\sqrt{\lg n}\lg\lg\lg n)$—again, due to the $O(\lg\lg\lg n)$ penalty for managing records with non-constant number of pointers.

Theorem 3.2 (Complexity for $d(i) = 2d(i-1)$) *The Nested-tree algorithm implements* **precede** *with a worst-case time complexity of* $O(\sqrt{\lg n}\lg\lg\lg n)$ *and* **insert** *with a complexity of* $O((\lg\lg\lg n)^2)$ *per operation, given that n elements are inserted.*

3.3 Balanced Distribution Trees

One of the problems in the previous schema is that the tree generated is too deep, and solving the case where two nodes have different ancestors may require more time than the other cases. The solution that we now introduce reduces the depth of the tree by increasing the degree of the nodes.

We consider the case where the degree of nodes at level i is defined to be $d(i) = t(i)$ for $i \geq 1$ (and $d(0) = 2$ and $t(0) = 1$), where, as before, $t(i)$ indicates the number of nodes present at level i. $t(i)$ can be defined again as $t(i) = t(i-1)d(i-1)$. Putting together the various components, we can solve the recurrence relation and obtain ($i \geq 1$):

$$d(i) = 2^{2^{i-1}} \qquad t(i) = 2^{2^{i-1}}$$

It is quite interesting to see that $d(i) > \sum_{j=0}^{i-1} d(j)$. Let us assume that n nodes are inserted in the tree, and let us indicate with h the depth of the tree. Clearly, n should satisfy the inequalities $t(h-1) < n \leq t(h)$ By solving this we obtain $h = \Theta(\lg\lg n)$. This shows that we have actually managed to reduce the height of the tree with respect to the nested-tree case.

We need now to determine what will be the maximum number of nestings of trees that can occur for n elements. Observe that the maximum number of nodes with the same direct ancestor is $d(h-1)$. Is it possible for a second level tree to have the same (or bigger) depth than the outermost one? We can show that the depth of the second level tree is strictly smaller than the depth of the outermost tree. If, on the contrary, the depth of the second level tree is the same or larger than the depth of the outermost tree, then

$$\sum_{j=0}^{h-1} t(j) < d(h-1)$$

as otherwise we would be able to fit all the $d(h-1)$ elements within the first $h-1$ levels. But we need to remember that $d(i) = t(i)$, thus

$$d(h-1) + \sum_{j=0}^{h-2} d(j) < d(h-1)$$

which would imply that the number of nodes in the first $h-2$ levels is negative, clearly impossible. Thus, the second level tree will have depth strictly lower than the depth of the outermost tree. As a consequence, the maximum number of nestings of trees k that we can have is itself $O(\lg\lg n)$.

Considering the two operations, **insert** and **precede**, we can observe that their implementations can be obtained as a straightforward modification of the algorithms used for the nested-tree in the previous section. To perform an insertion, **insert** updates a number of pointers proportional to the number of nesting of trees, which is $O(\lg\lg n)$. Since we pay a penalty of $O(k) = O(\lg\lg n)$ (as described in the previous subsection) for each access, we get that **insert** has a worst–case time complexity of $O((\lg\lg n)^2)$. The **precede** operation is implemented exactly as in the nested-tree case. It is easy to show that it has worst–case time complexity $O((\lg\lg n)^2)$.

Theorem 3.3 (Complexity for $d(i) = t(i)$) *The Balanced Distribution tree algorithm implements both* **insert** *and* **precede** *with a worst-case time complexity of* $O((\lg\lg n)^2)$ *per operation, where n is the number of elements inserted in the structure.*

3.4 Other Tree–Based Schemes

The above schemes are based on idea of using a tree organization and increasing the degree of the tree nodes as a function of their depth. We noticed that faster the rate at which these degrees grew the better performance we had in terms of the worst–case complexity for **insert** and **precede**. A natural extension of the above schemes would be to let the degrees of the nodes grow at an even faster rate than prescribed by the last scheme in the above section. In this section, we discuss schemes based on this idea and show that none of the "natural extensions" yields a better performance than the scheme discussed above.

If we further increase the nesting of trees and degree of nodes, by using the degree function

$$d(i) = \sum_{j=0}^{i} t(j)$$

(*Total Tree*), we can show (with similar reasoning as before) the following result [10].

Theorem 3.4 (Complexity for $d(i) = \sum t(i)$) *The Total Tree algorithm implements both* **insert** *and* **precede** *with a worst-case time complexity of* $\Theta((\lg\lg n)^2)$ *per operation, where n is the number of elements inserted in the structure.*

With the previous scheme we noticed that the degrees were not being increased at a fast enough rate to gain any asymptotic advantage in the performance. Hence, to get better algorithms we have to increase the degrees at an even faster rate. Doing this however may have the consequence that nested trees may not have strictly monotonically decreasing depth. *Quadratic Trees* are defined using the following degree function: $d(i) = d(i-1)^2$. In this case we cannot guarantee that nested trees have depth smaller than the outermost ones. Nevertheless, we can still show that the nesting of trees is proportional to the depth of the outermost tree. It is possible to show the following result [10].

Theorem 3.5 (Complexity for $d(i) = d(i-1)^2$) *The Quadratic Tree algorithm implements both* **insert** *and* **precede** *with a worst–case complexity of $O((\lg\lg n)^2)$ per operation, where n is the number of elements inserted.*

Thus, as we have seen in the last three schemes, even by progressively increasing the degree of the nodes we did not achieve any asymptotic improvement in the worst–case complexity of the two operations. On the other hand, faster increases in the degree of nodes may lead to unbalanced situations, where the **insert** operation may become considerably more expensive, as seen in the next scheme.

Let us consider the model where the degree function is $d(i) = 2^{d(i-1)}$ (*Exponential Trees*). In this case the degree of nodes at the lower levels of the tree becomes very large, and the nested trees do not decrease in depth with each additional level of nesting. This leads to the following result [10].

Theorem 3.6 (Complexity for $d(i) = 2^{d(i-1)}$) *The Exponential Tree algorithm has a worst–case time complexity of $\Omega(n)$ per insertion and $\Omega(n)$ per comparison, assuming that n is the number of elements inserted.*

Thus, indiscriminate increase of the rate at which the degrees grow can degrade the quality of the solution.

3.5 Does a priori knowledge of n help?

A natural question that emerges from the previous discussion is the following: would an *a priori* knowledge of n allow a better solution? In particular, can one use a tree–based scheme where the degrees of the tree nodes are a function of n that will yield a better solution than any of the previous scheme? This is similar in spirit to the analysis of union-find algorithms where the number of elements to be inserted is known in advance [2]. In this section, we briefly analyze a scheme that makes use of this information. For the sake of simplicity, we first ignore the additional penalty incurred for manipulating records with variable number of fields, and we ignore ceilings and floors in our calculations. Complete details of the analysis of this case can be found in [10].

Let us consider a scheme where we use the nested tree idea together with constant arity d at each level. d is assumed to be a function $t(n)$ of the number n of nodes in the tree (known in advance). We will choose $d = t(n)$ for some monotonically growing function t. A second-level tree will contain $t(n)$ nodes, and it will have degree $t(t(n))$; each third-level tree will contain $t(t(n))$ nodes and will have degree $t(t(t(n)))$ and so on.

Let us focus on the cost of performing a **precede** operation in this structure; we indicate with $c(n)$ the cost paid for executing a **precede** operation on a tree containing n nodes. The **precede** operation requires walking up the tree from the two nodes until a common ancestor is found, and repeating the same search in the nested tree localized as the set of successors of the previously found common ancestor. Hence, a crude overestimate gives us that $c(n) \le 1 + \lg_{t(n)} n + c(t(n))$ where the 1 originates from the comparison required to verify if the two nodes have the same ancestor, $\lg_{t(n)} n$ is the worst case number of steps required to walk up the tree in search of a common ancestor, and $c(t(n))$ is the cost for repeating the activity in the tree at the next level of nesting. The problem is now to find the best t which allows to minimize $c(n)$. Assuming that the recurrence will stop as soon as $t(n)$ is a constant (e.g., 2), and assuming that this happens in i steps of recursion, we obtain

$$c(n) \le i + \sum_{j=0}^{i} \frac{\lg t^i(n)}{\lg t^{i+1}(n)}$$

It can be shown [10] that the optimum chioce for $t(n)$ is $\sqrt{n}$, producing an upper bound of $\Theta(\lg\lg n)$ for $c(n)$. This yields an upper bound of $\Theta((\lg\lg n)^2)$ once the missing penalty for the non-constant number of fields in the records is reintroduced.

Hence, this scheme that makes use of the optimum choice of the degree function does not improve the upper bound, even with an *a priori* knowledge.

4 Discussion

In this paper we have described and studied the Temporal Precedence Problem, i.e., the problem of dynamically supporting the execution of the operations **insert** and **precede**, on a pointer machine. We have proposed various algorithms to implement these operations, studied their worst–case time complexity, and proved an upper bound of $O((\lg\lg n)^2)$ per operation for the problem.

By carefully looking at these results it is possible to observe that the square factor in this complexity formula arises from maintaining the non-constant number of fields for each record as a simple list of pointers. By using a more efficient organization it is possible to reduce the complexity to a simple $O(\lg\lg n)$ [10]. This upper bound, to our knowledge, is the best known for the $\mathcal{TP}$ problem. These results can be applied directly to state upper bounds for some problems related to parallel execution in non-deterministic programming frameworks (e.g., Prolog)—we have developed [9] and implemented [8] more efficient execution schemes for and-parallel Prolog using some of the ideas presented.

5 Epilogue

The main question that we have not dealt with in this paper is the determination of a lower bound for the $\mathcal{TP}$ problem. Techniques for proving lower bounds on pointer machines used by other researchers (e.g., [14, 11]) and results achieved by other researchers on similar problems [4, 15, 16, 7] do not seem to be suitable for the $\mathcal{TP}$ problem.

Since this paper was submitted, using a different technique, we have been able to prove that the $\mathcal{TP}$ problem has a lower bound time complexity of $\Omega(\lg\lg n)$ [12]. This result proves that the data structures proposed in this paper are indeed *optimal*.

Acknowledgements

We would like to thank L. Longpre, K. Melhorn and the anonymous referees for their comments and help in the development of this work.

References

[1] A.M. Ben-Amram. What is a Pointer Machine? Technical report, DIKU, University of Copenhagen, 1995.

[2] N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing*, 15(4), 1986.

[3] B. Chazelle. A Functional Approach to Data Structures and its Use in Multidimensional Searching. *SIAM Journal of Computing*, 17(3), 1988.

[4] P.F. Dietz and D.D. Sleator. Two Algorithms for Maintaining Order in a List. In *Symp. on Theory of Computing*. ACM Press, 1987.

[5] D.E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

[6] A.N. Kolmogorov and V.A. Uspenskii. On the Definition of an Algorithm. *Uspehi Mat. Nauk*, 13, 1958.

[7] K. Mehlhorn and S. Naher. Dynamic Fractional Cascading. *Algorithmica*, 5, 1990.

[8] E. Pontelli and G. Gupta. Implementation mechanisms for dependent and-parallelism. In *International Conference on Logic Programming*. MIT Press, 1997.

[9] E. Pontelli, D. Ranjan, and G. Gupta. On the Complexity of Parallel Implementation of Logic Programs. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 123–137. Springer Verlag, 1997.

[10] E. Pontelli, D. Ranjan, and G. Gupta. On the complexity of the insertion/precedes problem. Technical report, New Mexico State University, 1997.

[11] H. La Poutré. Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines. *Journal of Computer and System Sciences*, 52:87–99, 1996.

[12] D. Ranjan, E. Pontelli, L. Longpre, and G. Gupta. The Temporal Precedence Problem. Technical Report NMSU-CS-TR-9807, New Mexico State University, Computer Science, 1998.

[13] A. Schönhage. Storage Modification Machines. *SIAM Journal of Computing*, 9(3):490–508, August 1980.

[14] R.E. Tarjan. A Class of Algorithms which Require Nonlinear Time to Mantain Disjoint Sets. *Journal of Computer and System Sciences*, 2(18), 1979.

[15] A. Tsakalidis. Maintining Order in a Generalized Linked List. *ACTA Informatica*, 21, 1984.

[16] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory*, 10, 1977.