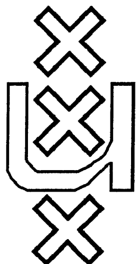
Institute for Language, Logic and Information

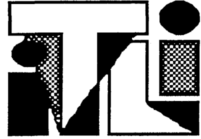
A DATA STRUCTURE FOR THE UNION-FIND PROBLEM HAVING GOOD SINGLE-OPERATION COMPLEXITY

Michiel H.M.Smid

ITLI Prepublication Series
for Computation and Complexity Theory CT-88-06



University of Amsterdam


Institute for Language, Logic and Information
Instituut voor Taal, Logica en Informatie

A DATA STRUCTURE FOR THE UNION-FIND PROBLEM HAVING GOOD SINGLE-OPERATION COMPLEXITY

Michiel H.M.Smid
Centre for Mathematics and Computer Science
Amsterdam

Received September 1988

Correspondence to:

Faculteit der Wiskunde en Informatica
(Department of Mathematics and Computer Science)
Roetersstraat 15
1018WB Amsterdam

or

Faculteit der Wijsbegeerte
(Department of Philosophy)
Grimburgwal 10
1012GA Amsterdam

A Data Structure for the Union-Find Problem Having Good Single-Operation Complexity

Michiel H.M. Smid*

September 1988

Abstract

We give a data structure for the union-find problem, in which each *UNION* resp. *FIND* operation takes $O(k)$ resp. $O(\log_k n)$ time, where k is a parameter, such that $2 \leq k \leq n$. This data structure is in the class of structures defined by Blum, and for $k = \Omega((\log n)^\epsilon)$ for some $\epsilon > 0$, it is optimal in this class. We also show how a copy of the data structure can be maintained efficiently in secondary memory, which makes it possible to reconstruct the original structure in case of a system crash.

Keywords: Analysis of algorithms, data structures, reconstruction problem, shadow administration, union-find problem.

1 Introduction

One of the basic problems in the theory of algorithms and data structures is the *Union-Find Problem.* In this problem we are given a collection of n disjoint sets $S_1, S_2, \ldots, S_n$, each containing one single element, and we have to carry out a sequence of operations of the following two types:

1. *UNION*(A, B, C): combine the two disjoint sets A and B into a new set named C.
2. *FIND*(x): compute the name of the (unique) set that contains x.

These operations have to be carried out on-line, i.e. each operation has to be completed before the next one is known. The union-find problem has received considerable attention. Tarjan showed in [3] that a sequence of m *UNION* and *FIND* operations can be carried out in total time $O(m\,\alpha(m+n, n) + n)$ using $O(n)$

*Bureau SION, Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. This work was supported by the Netherlands Organization for Scientific Research (NWO).

space, where α is a functional inverse of Ackermann's function. Furthermore, he introduced in [4] a machine model—the pointer machine—on which all known algorithms solving the union-find problem can be implemented. Tarjan showed that on such a pointer machine, any algorithm for the union-find problem needs $\Omega(m\,\alpha(m+n,n)+n)$ time for performing m *UNION* and *FIND* operations. See also [5].

In this paper we are interested in the single-operation time complexity of the union-find problem. Until recently, only algorithms were known having single-operation complexity $\Omega(\log n)$. (In this paper all logarithms without a subscript are to the base 2.) That is, there is always either a *UNION* or a *FIND* operation that needs $\Omega(\log n)$ time. In [1], Blum gives a data structure of size $O(n)$, in which each *UNION* operation can be performed in $O(k + \log_k n)$ time, and each *FIND* operation in $O(\log_k n)$ time. Here k is a parameter, possibly depending on n. Blum also gives a very general class $\mathcal{B}$ of data structures—containing all implementations on Tarjan's pointer machine—for which he proved the following theorem.

Theorem 1 (Blum) *Let DS be any data structure from the class $\mathcal{B}$. Suppose that every UNION operation can be performed in $O(k)$ time. Then there is a FIND operation that needs time*

$$\Omega\left(\frac{\log n}{\log k + \log\log n}\right).$$

As a corollary of this theorem we see that for each data structure in class $\mathcal{B}$, there is always either a *UNION* or a *FIND* operation that takes $\Omega(\log n/\log\log n)$ time.

In Section 2 we shall give a variant of Blum's structure, having the same complexity. That is, we give a structure of size $O(n)$, in which each *UNION* resp. *FIND* operation takes $O(k + \log_k n)$ resp. $O(\log_k n)$ time. Next, in Section 3, we adapt this structure such that each *UNION* operation can be carried out in $O(k)$ time, whereas the size of the structure and the time for a *FIND* operation remains the same. This structure is in Blum's class $\mathcal{B}$. Hence it follows from Theorem 1 that this structure is optimal—in class $\mathcal{B}$— if $k = \Omega((\log n)^\epsilon)$ for some $\epsilon > 0$.

The improved data structure consists of a number of trees—each set is stored in one such tree—and has the property that for a *UNION* operation we only have to visit the roots of two trees, together with their—at most k— direct descendants. Furthermore, a *FIND* operation does not change the structure. This property leads to an efficient solution to the *reconstruction problem*: Suppose that the data structure is stored in main memory. Then, after a system crash or as a result of errors in software, all information will get lost. Therefore, we store in the safe secondary memory a copy of the data structure, the so-called*shadow administration.* Due to the just mentioned property, the copy in secondary memory can be maintained efficiently after a *UNION* operation.

The reconstruction problem first appeared in Torenvliet and van Emde Boas [6], where the reconstruction of trie hashing functions is investigated. For a more thorough introduction to the reconstruction problem the reader is referred to Smid et al. [2], where it is shown that for several classes of data structures efficient shadow administrations exist.

In order to be able to analyze the efficiency of a shadow administration, we have to model the structure of secondary memory. In this paper we shall use the *Indexed Sequential Model* of [2]. In this model, the file in secondary memory is divided into blocks. There is the ability of replacing a block by another one, and to add a new block at the end of the file. Note that this model is realistic in practice; it corresponds to the notion of indexed sequential files. The model will be described more precisely in Section 4. In that section, we will show how a copy of the improved union-find structure can be maintained efficiently in secondary memory.

2 A variant of Blum's structure

Before we give the data structure that solves the union-find problem, let us first specify the model of computation. As usual, we take the random access machine (RAM), the memory of which is modeled as an array. A data structure is composed of 'indivisible pieces of information' of constant size, such as pointers, integers, etc. Each such indivisible piece will be stored in one array location. We consider a pointer to be an absolute address of an array location.

Let S be a set of n elements for which we want to solve the union-find problem. That is we want to maintain a partition of S under a sequence of *UNION* and *FIND* operations, where initially each set in the partition contains exactly one element. We store each set in the partition in a $UF(k)$-tree, defined as follows.

Definition 1 *Let k be an integer, $2 \leq k \leq n$. A tree T is called a* $UF(k)$-tree, *if*

1. *the root of T has at most k sons,*
2. *each node in T has either 0 or more than k grandsons (a grandson of a node v is a son of the son of v),*
3. *all leaves of T have the same depth.*

As mentioned already, we store each set A in the partition of S in a separate $UF(k)$-tree. The elements of A are stored in the leaves of the tree. In the root, we store the name of the set, the height of the tree, and the number of its sons. Each non-root node contains a pointer to its father. Finally, the root of the tree contains a pointer to each of its sons, and a pointer to an (arbitrary) leaf. Note that the root contains at most $k+1$ pointers. A $UF(k)$-tree storing a set of cardinality one, has two nodes, a root and one leaf.

To perform an operation $FIND(x)$, we get the leaf containing element x. Then we follow father-pointers until we reach the root of the tree, where we read the name of the set containing x.

It remains to give the $UNION$-algorithm. In order to perform the operation $UNION(A, B, C)$, we get the roots r resp. s of the trees containing the sets A resp. B. We distinguish three cases.

Case 1. The trees containing A and B have equal height, and the total number of sons of r and s is $\leq k$.

Assume w.l.o.g. that the number of sons of s is less than or equal to the number of sons of r. We change the father-pointers from all sons of s into pointers to r, and we store in r pointers to its new sons. Next we discard the root s, together with all its information. Finally, we adapt in r the number of its sons and the name of the set.

Case 2. The trees containing A and B have equal height, and the total number of sons of r and s is $> k$.

In this case we create a new root t. In this new root, we store two pointers to r and s; a pointer to a leaf of the new tree (we can take the corresponding pointer stored in r); the name of the new set C; the height of the new tree, which is one more than the corresponding value stored in r; and the number of sons, which is 2. In the old roots r and s we discard all information, and we add pointers to their new father t.

Case 3. The tree of B has smaller height than the tree of A.

We find in the tree of A a node v such that the subtree of v has equal height as the tree of B. This node v can be found by following the pointer from r to a leaf, and then by walking up in the tree. Then we change the father-pointers from all sons of s into pointers to v. (This makes sure that all leaves remain on the same level.) We discard the root s, together with all its information. Finally, we adapt the name of the set stored in r. Note that the height of the tree and the number of sons of r does not change.

The following theorem gives the result.

Theorem 2 *Let k and n be integers, such that $2 \leq k \leq n$. Using $UF(k)$-trees, the union-find problem on n elements can be solved, such that*

1. *each UNION takes $O(k + \log_k n)$ time,*
2. *each FIND takes $O(\log_k n)$ time,*
3. *the data structure has size $O(n)$.*

Proof. The time needed to perform a $FIND$ operation is bounded above by the height of a $UF(k)$-tree. It follows from Definition 1 that if level i in a $UF(k)$-tree contains any nodes, it contains at least $k^{\lfloor i/2 \rfloor}$ of them. (Here the root is at level 0.)

Since such a tree has at most n leaves, its height is at most $1 + 2\lceil \log_k n \rceil$. Hence a *FIND* operation takes $O(\log_k n)$ time in the worst case.

It is easy to see that the given *UNION*-algorithm correctly maintains $UF(k)$-trees. Note that we can determine in constant time in which of the three cases we are, since all relevant information for deciding this is stored in the roots. It is clear that Case 1 and 2 of the *UNION*-algorithm take $O(k)$ time in the worst case. In Case 3, it takes $O(\log_k n)$ time to find the node v, whereas the rest of this case can be carried out in $O(k)$ time.

Clearly, the size of a $UF(k)$-tree is linear in the number of its leaves, which shows that the entire data structure has size $O(n)$. □

3 An improved data structure

We saw in Section 2, that it takes $O(k+\log_k n)$ time to perform a *UNION* operation on $UF(k)$-trees. In this time bound, the $O(\log_k n)$ term is due to the fact that in Case 3, we have to search the node v, the subtree of which has equal height as the tree containing B. Clearly, if we take for v an arbitrary son of the root of the tree of A, the time for each *UNION* operation will be bounded by $O(k)$. It remains to prove then, however, that the height of the trees does not increase.

Definition 2 *Let k be an integer, $2 \leq k \leq n$. A tree T is called an $IUF(k)$*-tree, *where the I stands for* improved, *if*

1. *the root of T has at most k sons,*
2. *each node in T has either 0 or more than k grandsons.*

Again, we store each set A in the partition of S in a separate $IUF(k)$-tree. As before, the elements of A are stored in the leaves of the tree; in the root, we store the name of the set, the height of the tree, and the number of its sons. Also each non-root node contains a pointer to its father, and the root contains pointers to all its sons. (Now we do not need a pointer from the root to a leaf.)

Remark that Definition 2 does not imply anymore that the heights of these trees are bounded above by $1 + 2\lceil \log_k n \rceil$, since the leaves do not have to be positioned at the same level. However, the trees that are made by the *UNION*-algorithm to be described below have heights bounded by $1 + 2\lceil \log_k n \rceil$.

The *FIND*-algorithm for $IUF(k)$-trees is the same as for $UF(k)$-trees.

Suppose we have to carry out the operation $UNION(A, B, C)$. Let r resp. s be the roots of the trees containing the sets A resp. B. As before, we distinguish three cases.

Case 1′. The trees containing A and B have equal height, and the total number of sons of r and s is $\leq k$. This case is handled in the same way as Case 1 of Section 2.

Case 2'. The trees containing A and B have equal height, and the total number of sons of r and s is $> k$. Also this case is handled as Case 2 of Section 2.

Case 3'. The tree of B has smaller height than the tree of A.

Let v be an arbitrary son of r. Then we change the father-pointers from all sons of s into pointers to v. The root s, together with all its information, is discarded. Also, we adapt the name of the set stored in r.

First note that the given algorithm correctly maintains $IUF(k)$-trees. Furthermore, it is clear that each *UNION* operation takes $O(k)$ time, since in Case 3', the node v can be found in constant time. Also, the size of the data structure is still bounded by $O(n)$. It remains to show that the heights of the trees are bounded by $O(\log_k n)$.

Suppose we are given a collection of n disjoint sets $S_1, S_2, \ldots, S_n$, each containing one single element, and consider a sequence of *UNION* and *FIND* operations. Let UF be the data structure, consisting of $UF(k)$-trees, if we perform these operations according to the algorithm of Section 2. Furthermore, let IUF be the data structure consisting of $IUF(k)$-trees, where the operations are carried out as described in the present section.

So if $\bigcup_{i \in I} A_i$ is a partition of the n elements at some moment in the sequence of operations, there are two data structures. First, there is a structure $UF = \{T_i : i \in I\}$, where each T_i is a $UF(k)$-tree storing the set A_i. Also, there is a structure $IUF = \{T_i' : i \in I\}$, where each T_i' is an $IUF(k)$-tree storing A_i. Now each *UNION* operation is performed—in parallel—on both UF and IUF.

Lemma 1 *At each moment, the trees T_i and T_i' have the same height, and the roots of these trees have the same number of sons. To perform a UNION operation, if we are in Case j for the structure UF, we are in Case j' for IUF, for $j = 1, 2, 3$.*

Proof. Initially, each tree T_i and T_i' contains two nodes, one root and one leaf. Hence at the beginning the statement is true.

Then the lemma can easily be proved by induction on the number of performed *UNION* operations, using the following observation. Whether we are in Case 1, 2 or 3 of the *UNION*-algorithm of the structure UF, depends only on the height of the trees and on the number of sons of the corresponding roots. The same holds for the structure IUF. Then by checking the *UNION*-algorithms, the lemma follows easily. □

Theorem 3 *Let k and n be integers, such that $2 \leq k \leq n$. Using $IUF(k)$-trees, the union-find problem on n elements can be solved, such that*

1. *each UNION takes $O(k)$ time,*
2. *each FIND takes $O(\log_k n)$ time,*

3. the data structure has size $O(n)$.

Proof. We have seen already that a *UNION* operation takes $O(k)$ time, and that the size of the data structure is bounded by $O(n)$. It follows from Lemma 1 and the proof of Theorem 2, that the height of an *IUF*(k)-tree, made by the given *UNION*-algorithm, is at most $1 + 2\lceil \log_k n \rceil$. Hence each *FIND* operation takes $O(\log_k n)$ time. □

Since the given data structure is contained in Blum's class $\mathcal{B}$, introduced in [1], Theorems 1 and 3 yield

Corollary 1 *The data structure of Theorem 3 is optimal in Blum's class $\mathcal{B}$ of structures for the union-find problem, for all values of k satisfying $k = \Omega((\log n)^\epsilon)$ for some $\epsilon > 0$.*

4 An efficient shadow administration

In this section we shall show how we can maintain efficiently a copy of the data structure of Theorem 3 in secondary memory. In this way we are able to reconstruct the original data structure—that is stored in main memory—after e.g. a system crash. The structure stored in secondary memory is called the *shadow administration.*

First we introduce our model of secondary memory, the so-called *Indexed Sequential Model* of [2]. In this model, a file in secondary memory is divided into *blocks* of some fixed size. There is the ability of *direct block access*: it is possible to access a block directly provided its physical address is known. To update a file, the following operations are allowed:

1. We can replace a block by another block, or a number of (physically) consecutive blocks by at most the same number of blocks.

2. We can add a new block, or a number of new blocks, at the end of the file.

Now the file in secondary memory is maintained by replacing all blocks that have to be changed by the corresponding updated parts of the structure in main memory. The complexity of this operation is expressed by the number of *disk accesses* that has to be done (for each segment of consecutive blocks we have to replace, we have to do one disk access) and by the total amount of space that has to be transported from main memory to secondary memory.

As mentioned already in Section 2, the data structure consists of 'indivisible pieces of information', each of which is stored in main memory in one array location. Also recall that we consider a pointer as an absolute address of an array location in main memory. We store a copy of the data structure in secondary

memory as follows. We reserve a number of blocks of some predetermined size (see below), and we distribute the structure over these blocks. Together with each indivisible piece of information we store its absolute address in main memory. Hence the size of the structure in secondary memory is at most twice as large as the structure in main memory. In order to reconstruct the original data structure—e.g. after a system crash—we transport the entire file to main memory, and each indivisible piece of information is stored in the array location where it was before the crash. This guarantees that each pointer 'points' to the correct position in main memory. It is clear that reconstruction takes an amount of data transport and an amount of computing time which are linear in the size of the data structure.

So in order to store a copy of the data structure *IUF*—consisting of $IUF(k)$-trees—we first have to choose a blocksize. Since the root of an $IUF(k)$-tree has at most k sons, the total size of this root together with all its sons and all the information stored in these nodes (i.e. pointers, name of the set, height of the tree and the number of sons), and all their addresses in main memory, is bounded above by ck for some constant c. Also, there is a constant c' such that the size of the entire data structure *IUF*, together with all their addresses, is at most $c'n$.

Now we reserve in secondary memory $\lceil \frac{c'n}{ck} \rceil$ blocks of size $2ck$. The copy of the data structure *IUF* will be stored in these blocks. We call a block *free* if at least half of the block is empty. The following lemma can easily be proved.

Lemma 2 *There is always at least one free block.*

Initially we have n trees, each of them having two nodes, a root and a leaf. We store these trees in main memory. Also copies of the trees are distributed over the reserved blocks. (For each tree the root and its son, together of course with their array locations in main memory, are stored in the same block.) In each tree in main memory, we store in its root the address of its copy in secondary memory. Finally, we maintain in main memory a stack containing the addresses of all free blocks. By Lemma 2, this stack is never empty. Note that the amount of space in main memory remains bounded by $O(n)$. Also, the stack will only be used for updating the structure in secondary memory efficiently; it is not used for reconstructing the original data structure. Therefore this stack may get lost after a crash.

Since a *FIND* operation does not change the data structure, such an operation does not affect the shadow administration.

A *UNION* operation is first performed on the structure in main memory according to the algorithm of Section 3. Then the shadow administration in secondary memory is adapted. We take care that at each moment the following holds:

Invariant: For each $IUF(k)$-tree in the structure, the root and all its sons, together with all the information stored in these nodes, and all their array locations in main memory, are stored in the same block in secondary memory.

Clearly, this invariant holds initially. In the sequel we shall not state each time explicitly that if we put information in a block, we also store with it its array location in main memory. It is clear how this can be done.

Suppose we have to carry out the operation $UNION(A, B, C)$. Let r resp. s be the roots of the trees containing the sets A resp. B.

Case 1′. The trees containing A and B have equal height, and the total number of sons of r and s is $\leq k$.

Assume w.l.o.g. that the number of sons of s is less than or equal to the number of sons of r. In the block containing r we remove this root and all its sons. (Remark that we can read the address of this block in the root r that is stored in main memory.) If this block becomes free, we put its address on the stack. In the block containing s we do the same. Next we take the address of a free block from the stack, and in that block we store the root, together with its sons, of the new tree. If this block remains free we put it back on the stack. In main memory, we store in the root of the new tree, the address of the block containing its copy.

Case 2′. The trees containing A and B have equal height, and the total number of sons of r and s is $> k$.

In the block containing r we remove this root, together with all the information stored in it. If the block becomes free, we put its address on the stack. In the block containing s we do the same. Then we store the new root, together with its sons r and s and all the information that these three nodes contain, in a free block, that we take from the stack. If this block remains free it is put back on the stack. In main memory we store in the new root the address of the block containing its copy.

Case 3′. The tree of B has smaller height than the tree of A.

In the block containing r we change the name of the set from A into C. In the block containing s, we change the pointers of the sons of s, and we remove the root s together with all its information. If this block becomes free we put its address on the stack.

The following theorem summarizes the result.

Theorem 4 *Let k and n be integers, such that $2 \leq k \leq n$. For the data structure of Theorem 3, solving the union-find problem on n elements, there is a shadow administration*

1. *of size $O(n)$,*

2. *that can be maintained after a UNION operation at the cost of at most three disk accesses and the transport of $O(k)$ data.*

Furthermore, the original data structure can be reconstructed at the cost of one disk access, the transport of $O(n)$ data, and $O(n)$ computing time.

Proof. The proof follows immediately from the above discussion. □

Acknowledgement

I would like to thank Mark Overmars, Leen Torenvliet and Peter van Emde Boas for giving their comments on an earlier version of this paper.

References

[1] N. Blum. *On the single-operation worst-case time complexity of the disjoint set union problem.* SIAM J. Comput. **15** (1986), pp. 1021-1024.

[2] M.H.M. Smid, L. Torenvliet, P. van Emde Boas and M.H. Overmars. *Two models for the reconstruction problem for dynamic data structures.* Report FVI-87-13, University of Amsterdam, 1987.

[3] R.E. Tarjan. *Efficiency of a good but not linear set union algorithm.* J. ACM **22** (1975), pp. 215-225.

[4] R.E. Tarjan. *A class of algorithms which require nonlinear time to maintain disjoint sets.* J. Comput. System Sci. **18** (1979), pp. 110-127.

[5] R.E. Tarjan and J. van Leeuwen. *Worst-case analysis of set union algorithms.* J. ACM **31** (1984), pp. 245-281.

[6] L. Torenvliet and P. van Emde Boas. *The reconstruction and optimization of trie hashing functions.* Proc. 9th International Conf. on Very Large Databases (1983), pp. 142-156.

The ITLI Prepublication Series

1986

86-01	The Institute of Language, Logic and Information
86-02 Peter van Emde Boas	A Semantical Model for Integration and Modularization of Rules
86-03 Johan van Benthem	Categorial Grammar and Lambda Calculus
86-04 Reinhard Muskens	A Relational Formulation of the Theory of Types
86-05 Kenneth A. Bowen, Dick de Jongh	Some Complete Logics for Branched Time, Part I Well-founded Time, Forward looking Operators
86-06 Johan van Benthem	Logical Syntax

1987

87-01 Jeroen Groenendijk, Martin Stokhof	Type shifting Rules and the Semantics of Interrogatives
87-02 Renate Bartsch	Frame Representations and Discourse Representations
87-03 Jan Willem Klop, Roel de Vrijer	Unique Normal Forms for Lambda Calculus with Surjective Pairing
87-04 Johan van Benthem	Polyadic quantifiers
87-05 Víctor Sánchez Valencia	Traditional Logicians and de Morgan's Example
87-06 Eleonore Oversteegen	Temporal Adverbials in the Two Track Theory of Time
87-07 Johan van Benthem	Categorial Grammar and Type Theory
87-08 Renate Bartsch	The Construction of Properties under Perspectives
87-09 Herman Hendriks	Type Change in Semantics: The Scope of Quantification and Coordination

1988

Logic, Semantics and Philosophy of Language:

LP-88-01 Michiel van Lambalgen	Algorithmic Information Theory
LP-88-02 Yde Venema	Expressiveness and Completeness of an Interval Tense Logic
LP-88-03	Year Report 1987
LP-88-04 Reinhard Muskens	Going partial in Montague Grammar
LP-88-05 Johan van Benthem	Logical Constants across Varying Types
LP-88-06 Johan van Benthem	Semantic Parallels in Natural Language and Computation
LP-88-07 Renate Bartsch	Tenses, Aspects, and their Scopes in Discourse
LP-88-08 Jeroen Groenendijk, Martin Stokhof	Context and Information in Dynamic Semantics
LP-88-09 Theo M.V. Janssen	A mathematical model for the CAT framework of Eurotra

Mathematical Logic and Foundations:

ML-88-01 Jaap van Oosten	Lifschitz' Realizabiility
ML-88-02 M.D.G. Swaen	The Arithmetical Fragment of Martin Löf's Type Theories with weak Σ-elimination
ML-88-03 Dick de Jongh, Frank Veltman	Provability Logics for Relative Interpretability

Computation and Complexity Theory:

CT-88-01 Ming Li, Paul M.B.Vitanyi	Two Decades of Applied Kolmogorov Complexity
CT-88-02 Michiel H.M. Smid	General Lower Bounds for the Partitioning of Range Trees
CT-88-03 Michiel H.M. Smid, Mark H. Overmars Leen Torenvliet, Peter van Emde Boas	Maintaining Multiple Representations of Dynamic Data Structures
CT-88-04 Dick de Jongh, Lex Hendriks Gerard R. Renardel de Lavalette	Computations in Fragments of Intuitionistic Propositional Logic
CT-88-05 Peter van Emde Boas	Machine Models and Simulations (revised version)