Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).
MATHEMATICAL CENTRE TRACTS 81

FOUNDATIONS OF
COMPUTER SCIENCE II

K.R. APT (ed.)
J.W. DE BAKKER (ed.)

PART 1

MATHEMATISCHE CENTRUM AMSTERDAM 1979
Preface

E.L. LAWLER:  
Graphical algorithms and their complexity  
3

J. VAN LEEUWEN:  
The complexity of data organization  
37
PREFACE

The Second Advanced Course on the Foundations of Computer Science was organized by the Mathematical Centre as part of an international cooperation under the auspices of the European Communities, and took place at the University of Amsterdam, May 31 - June 11, 1976. The Course consisted of six series of lectures, and the notes of five of these have been collected in the present Tracts, jointly edited by K.R. Apt and the undersigned.

I am very grateful to the Netherlands Organization for the Advancement of Pure Research (Z.W.O.) and to the European Communities for generously supplying the money to organize the Course, to the lecturers for their excellent contributions, to Mrs. S.J. Kuipers-Hoekstra for her invaluable help with the organization, and to the staff of the publication services of the Mathematical Centre for their undaunted efforts in the typing and printing of these books.

J.W. de Bakker

Director of the Course
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2. Topological ordering</td>
<td>4</td>
</tr>
<tr>
<td>3. Recognition of series parallel digraphs.</td>
<td>5</td>
</tr>
<tr>
<td>4. Isomorphism of transitive series parallel digraphs</td>
<td>16</td>
</tr>
<tr>
<td>5. Subgraph isomorphism</td>
<td>19</td>
</tr>
<tr>
<td>6. Finding minimum spanning trees</td>
<td>20</td>
</tr>
<tr>
<td>7. Generating all maximal independent sets</td>
<td>25</td>
</tr>
<tr>
<td>8. Finding a maximum independent set</td>
<td>28</td>
</tr>
<tr>
<td>9. Computing the chromatic number of a graph.</td>
<td>29</td>
</tr>
<tr>
<td>References</td>
<td>31</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

Graphs have important applications in all branches of science and technology. It follows that algorithms which solve problems on graphs are important for the problems they solve. But there are additional reasons why graphs and graphical algorithms have particular fascination for computer scientists. Graphical algorithms pose a challenging array of problems in data structures, algorithmic analysis, and complexity theory.

There is now a great wealth of literature on the subject of graphical algorithms. The present notes are not intended as a survey. The author has made a rather arbitrary and personal selection of problems to discuss, with the expectation that these problems may serve as an introduction to the methodology of the subject area.

As a first, very simple, problem, we find a "topological ordering" of the nodes of an acyclic digraph. This can be done in $O(n+m)$ time, provided the digraph is specified by an "arc-l:st" data structure. (Hereafter, we consistently let $n$ denote the number of nodes and $m$ the number of arcs.)

We then consider a "recognition" problem; determining if an acyclic digraph is "series parallel". By careful tailoring of the algorithm, we are able to obtain a bound of $O(n^2)$ on the running time - a second example of an algorithm with polynomial-bounded running time.

We then take a brief excursion into the problem of graphic isomorphism. There is no known polynomial-bounded algorithm for determining whether or not two graphs are isomorphic, nor is there convincing evidence that such an algorithm does not exist. Yet we can show that for the special case of transitive series parallel digraphs there is such an algorithm.
A very basic and important problem is that of finding a minimum cost spanning tree in an undirected graph. Various algorithms have been proposed to solve this problem in $O(n^2)$ or $O(m \log n)$ time.

Recently, several algorithms have been proposed which have $O(m \log \log n)$ running time. We give a brief description of the simplest of these new algorithms. The reduced running time is obtained through the clever use of data structures for the implementation of priority queues.

Given a graph, one sometimes wants to generate all subgraphs of a particular type, e.g. trees, cycles. As typical of a problem of this type, we consider the generation of all maximal independent sets. A newly developed algorithm is described, and shown to have running time which is linear in the number of maximal independent sets.

The problem of finding a maximum (cardinality) independent set is "NP-complete", and hence it is extremely unlikely that a polynomial-bounded algorithm exists. One obvious way to solve the problem is to generate all maximal independent sets and to pick out one with a maximum number of elements. However, it is possible to do significantly better than this, and we indicate techniques one can apply to reduce the exponential running time of the search procedure.

Finally, we consider another very difficult NP-complete problem: computing the chromatic number of a graph. We describe the application of dynamic programming to this problem, and make use of the previous results on generating maximal independent sets, to obtain an algorithm whose running time is bounded by a relatively low order exponential function.

2. TOPOLOGICAL ORDERING

It is a well known result that a digraph $G = (N,A)$ is acyclic (contains no directed cycles) if and only if it is possible to "topologically" number its nodes, so that $(i,j) \in A$ only if $i < j$. (That is, all arcs are directed from a lower-numbered node to a higher-numbered node.) How efficiently can we test an arbitrary digraph $G$ to determine if it is acyclic, and if it is acyclic, to determine a topological numbering of its nodes?

The "classical" procedure for solving this problem is as follows. Find a node with in-degree zero, give that node the number 1, and delete it from $G$. Then find another node with in-degree zero, give it number 2 and delete it. Continue in this way until either all nodes have been num-
bered or until there is no node with in-degree zero in the remaining subgraph (in which case \( G \) is not acyclic).

The efficiency with which this procedure can be implemented depends critically on the way in which the digraph is specified to the computer. If \( G \) is represented by an adjacency matrix, there is no way to implement the procedure in less than \( O(n^2) \) time. In fact, it is has been proved that no algorithm can solve this problem (or any problem in a much wider class) without "looking at" at least \( O(n^2) \) entries in the adjacency matrix [12].

We can, however, do much better if \( G \) is specified by means of "arc lists". That is, for each node \( x \), there is a simple linear list of all the nodes to which there are arcs directed from \( x \). Then we can proceed as follows: First compute the in-degree of each node. This is done quite easily in \( O(n+m) \) time by simply scanning the arc lists. (When \( y \) is found in the list for \( x \), the in-degree of \( y \) is incremented by one.) Then scan through the array of in-degrees, picking out those nodes which have in-degree zero, and make a list of these nodes. This requires \( O(n) \) time. Then choose a node with in-degree zero and give it the number 1. The arc list for this node is used to decrement the in-degrees of the nodes to which it has arcs. If any of these in-degrees are reduced to zero, the node in question is placed on the list of nodes with in-degree zero. Then choose another node with in-degree zero and repeat. It is apparent that the decrementation operations require only \( O(m) \) time overall, because there are exactly \( m \) entries in the arc lists. Hence the algorithm is seen to require \( O(n+m) \) time overall.

This very simple example serves to illustrate the importance of the data structure which is used to specify a graph or digraph.

3. RECOGNITION OF SERIES PARALLEL DIGRAPHS

"Series parallel" digraphs have a number of very useful properties. In particular, a number of combinatorial problems which are NP-complete for general digraphs are computationally tractible for the special case of series parallel digraphs [5]. In the present section, we shall concern ourselves with the problem of recognizing series parallel digraphs, i.e. testing an arbitrary digraph to determine if it is series parallel. In later sections, we shall discuss some interesting properties of these digraphs.

First let us give a recursive definition of this class of digraphs.
A digraph on a single node is transitive series parallel. If \( G_1 = (N_1, A_1) \) and \( G_2 = (N_2, A_2) \), \( N_1 \cap N_2 = \emptyset \), are transitive series parallel digraphs, then \( G_1 \cup G_2 = (N_1 \cup N_2, A_1 \cup A_2) \), the parallel composition of \( G_1, G_2 \), is transitive series parallel. Also, \( G_1 \circ G_2 = (N_1 \cup N_2, A_1 \cup A_2 \cup (N_1 \times N_2)) \), the series composition of \( G_1, G_2 \), is transitive series parallel. Only those digraphs which can be obtained by a finite number of series and parallel compositions of single-node digraphs are transitive series parallel. Any digraph whose transitive closure is transitive series parallel is series parallel.

Some examples of series parallel digraphs are given in Figure 1.

![Examples of Series Parallel Digraphs](image)

Figure 1

Clearly every series parallel digraph is acyclic. Every acyclic digraph with three nodes or less is series parallel. The only 4-node acyclic digraph that is not acyclic is the one shown in Figure 2. It is not hard to show that if a digraph contains this 4-node digraph as an induced subgraph, then the digraph is not series parallel. The recognition procedure will provide a proof of the converse for transitive digraphs. Thus, a digraph is series parallel if and only if its transitive closure does not contain the digraph of Figure 2 as a "forbidden subgraph".
The structure of a series parallel digraph is represented by a "decomposition tree". Decomposition trees for the digraphs in Figure 1 are shown in Figure 3. Each leaf of the decomposition tree of G is identified with a node of G. Each internal node is designated "S" or "P", indicating serial and parallel composition of the subgraphs identified with its sons. The ordering of the sons of a P-node is unimportant, whereas the ordering of the sons of S-node is significant, indicating the order of series composition.

Let us adopt the convention that no son of an S-node is an S-node and no son of a P-node is a P-node. We assert that, subject to this assumption, the decomposition tree of a series parallel digraph is uniquely determined, up to a permutation of the sons of P-nodes.
Our recognition procedure attempts to construct the decomposition tree for a given digraph $G$. If, at any step, it is not possible to carry the construction further, then we shall be able to exhibit the forbidden subgraph as an induced subgraph of the transitive closure of $G$, proving that $G$ is not series parallel.

Let us suppose that the nodes of the given digraph $G = (N, A)$ are topologically ordered. As we have seen, such a numbering can be obtained
for an acyclic digraph in O(n+m) time.

Our strategy will be to obtain the decomposition tree T_{j+1} for the
subgraph induced on nodes 1,2,...,j+1 from the tree T_j obtained for nodes
1,2,...,j. It is, of course, a simple matter to obtain T_2, to start the
procedure.

To facilitate our task, we shall compute the values of two Boolean
variables, SER and PAR, for each node of T_j. At each leaf node i,

\[
\begin{align*}
\text{SER}(i) &= 1 \\
\text{PAR}(i) &= 0
\end{align*}
\]

if \((i,j+1) \in A\)

and

\[
\begin{align*}
\text{SER}(i) &= 0 \\
\text{PAR}(i) &= 1
\end{align*}
\]

if \((i,j+1) \notin A\)

For each internal node x of T_j, PAR(x) = 1 if and only if PAR(y) = 1 for
each son y of x. For each internal P-node x, SER(x) = 1 if and only if
SER(y) = 1 for each son y of x. For each internal S-node x, SER(x) = 1
if and only if PAR(y) = 1 for the rightmost son y of x.

It should be clear that the values of SER and PAR can be computed for
all nodes of T_j in O(n) time, proceeding from the leaf nodes toward the
root node.

In words, PAR(x) = 1, if and only if node j+1 is "in parallel" with
the subgraphs induced by the nodes which are descendent of x in T_j. (Note
that this property depends upon topological ordering.) Node j+1 is "in
series" with the subgraph induced by node x of T_j if and only if either
SER(x) = 1 or else x is a P-node and PAR(x') = 0, where x' is a brother to
the right of x. Clearly, it is not possible for both SER(x) = 1 and
PAR(x) = 1.

Now let us try to insert node j+1 into T_j, to obtain the tree T_{j+1}.
If either SER(r) = 1 or PAR(r) = 1 at the root node r, then j+1 is in
series or in parallel, respectively, with the subgraph induced on nodes
1,2,...,j. In particular, if r is a P-node and SER(r) = 1 or if r is an
S-node and PAR(r) = 1, we must create a new root node, as indicated
Creation of New Root Node

Figure 4

in Figure 4. These are special cases. Otherwise, follow the procedure for S-nodes or P-nodes, as appropriate, outlined below.

S-Node Procedure

It is assumed that \( \text{PAR}(r) = 0 \) for the S-node \( r \).

Case 1. If \( \text{SER}(r) = 1 \), then insert \( j+1 \) into \( T \) as the new rightmost son of
r. This completes the construction of $T_{j+1}$.

Case 2. Suppose Case 1 does not apply. Let the sons of $r$, indexed from right to left, be $s_1, s_2, \ldots, s_p$, $p \geq 2$. If $\text{PAR}(s_1) = 0$ or if $\text{SER}(s_2) = 1$, then node $j+1$ is in series with the subgraph induced on the descendents of $s_2, s_3, \ldots, s_p$. Consequently, if $j+1$ is to be inserted anywhere into $T_j$, $j+1$ must be inserted somewhere into the subtree rooted at $s_1$. If $s_1$ is itself a leaf node (in which case $\text{PAR}(s_1) = 1$), replace $s_1$ by a new $P$-node with $s_1$ and $j+1$ as sons, thereby completing the construction of $T_{j+1}$. If $s_1$ is a $P$-node, then apply the $P$-node procedure with $r$ set to $s_1$. (Note that in this case $\text{SER}(s_1) = 0$.)

Case 3. Suppose Cases 1 and 2 do not apply. (In this case, $\text{PAR}(s_1) = 1$ and $\text{SER}(s_2) = 0$.) If $\text{PAR}(s_1) = \text{PAR}(s_2) = \text{PAR}(s_k) = 1$, for $k \geq 2$, and $\text{SER}(s_{k+1}) = 1$, then node $j+1$ is in parallel with the subgraph induced on the descendents of $s_1, s_2, \ldots, s_k$ and in series with the subgraph induced on the descendents of $s_{k+1}, s_{k+2}, \ldots, s_p$. Consequently, $j+1$ can be inserted into $T_j$ by creating a new $S$-node and $P$-node, as shown in Figure 5. This completes the construction of $T_{j+1}$.

![Diagram](image.png)

Insertion of Node $j+1$ in Case 3 of S-node Procedure

Figure 5
Case 4. Suppose Cases 1, 2, and 3 do not apply. That is, \( \text{PAR}(s_1) = \text{PAR}(s_2) = \ldots = \text{PAR}(s_k) = 1, \ k \geq 1, \) and \( \text{PAR}(s_{k+1}) = 0, \text{SER}(s_{k+1}) = 0. \)

In this case it is possible to identify the forbidden subgraph in the transitive closure of the subgraph induced on nodes \( 1, 2, \ldots, j+1, \) as shown in Fig. 6.

\[
\begin{align*}
\text{SER}(s_{k+1}) &= 0 \\
\text{PAR}(s_{k+1}) &= 0 \\
\text{PAR}(s_1) &= 1
\end{align*}
\]

Forbidden Subgraph
Identified in Case 4 of S-Nodes Procedure
Figure 6.

P-Node Procedure

It is assumed that \( \text{SER}(r) = 0 \) for the P-node \( r. \)

Case 1. If \( \text{PAR}(r) = 1, \) then insert \( j+1 \) into \( T_j \) as a new son of \( r. \) This completes the construction of \( T_{j+1}. \)

Case 2. Suppose Case 1 does not apply. Let the sons of \( r, \) indexed arbitrarily, be \( s_1, s_2, \ldots, s_p; \ p \geq 2. \) If \( \text{PAR}(s_i) = 1 \) for all \( i \neq k, \) and \( \text{PAR}(s_k) = 0, \) then node \( j+1 \) is in parallel with the subgraph induced on the descendents of \( s_k, \ i \neq k. \) Consequently, if \( j+1 \) is to be inserted anywhere into \( T_j, \) \( j+1 \) must be inserted somewhere into the subtree rooted to \( s_k. \) If \( s_k \) is itself a leaf node (in which case \( \text{SER}(s_k) = 1), \) replace \( s_k \) by a new S-node with \( s_k \) and \( j+1 \) as sons, thereby completing the construction of \( T_{j+1}. \) If \( s_k \) is an S-node, then apply the S-node procedure with \( r \) set to \( s_k. \) (Note that in this case \( \text{PAR}(s_k) = 0. )\)
Case 3. Suppose Cases 1 and 2 do not apply. (In this case, there are at least two sons $s_i$ such that $\text{PAR}(s_i) = 0$.) If $\text{SER}(s_i) = 1$ for each son $s_i$ such that $\text{PAR}(s_i) = 0$, then node $j+1$ is in parallel with the subgraph induced on the descendents of all sons $s_i$ for which $\text{PAR}(s_i) = 1$ and in series with the subgraph induced on the descendents of all sons $s_i$ for which $\text{PAR}(s_i) = 0$. Consequently, $j+1$ can be inserted into $T_j$ by creating a new S-node and P-node, as shown in Figure 7. This completes the construction of $T_{j+1}$.

Case 4. Suppose Cases 1, 2 and 3 do not apply. That is, there are sons, say $s_1$, $s_2$, such that $\text{PAR}(s_1) = \text{PAR}(s_2) = 0$, and $\text{SER}(s_2) = 0$. In this case it is possible to identify the forbidden subgraph in the transitive closure of the subgraph induced on nodes $1,2,...,j+1$, as shown in Figure 8.

---

**Figure 7.**

Insertion of node $j+1$ in Case 3 of P-node Procedure.

**Figure 8.**

Forbidden Subgraph Identified in Case 4 of P-node Procedure
It should be apparent that by starting at the root node and proceeding toward the leaves, one either obtains \( T_{j+1} \) from \( T_j \) in \( O(n) \) running time, or shows that the digraph \( G \) is nonseries parallel. Thus, the overall running time required for recognition of series parallel digraphs is \( O(n^2) \).

"Two Dimensionality" of Transitive Series Parallel Digraphs

Every partial order on \( n \) elements can be represented as the intersection of total orders. The minimum number of total orders which are necessary to so represent a partial order is called its "dimension".

Every transitive series parallel digraph represents a partial order of dimension 2, but not conversely. To capture the notion of two-dimensionality geometrically, imagine that the \( n \) elements of the partial order " \( \leq \) " are identified with points in the plane. Let element \( i \) have coordinates \((x_i, y_i)\) and element \( j \) have coordinates \((x_j, y_j)\). Then \( i \leq j \), if and only if \( x_i \leq x_j \) and \( y_i \leq y_j \). (Note that antisymmetry requires that either \( x_i \neq y_i \) or \( x_j \neq y_j \).)

Thus an element is " \( \leq \) " a second element if and only if it is located below and to the left of the second element. Note that parallel composition can be effected by locating \( G_1, G_2 \) as shown in Figure 9 and series composition as shown in Figure 10.

![Parallel Composition](image)
One can obtain two total orders whose intersection represents the transitive closure of a given series parallel digraph by simple operations performed on the decomposition tree. For each leaf node \( j \), the two total orders are simply \( \alpha = \beta = (j) \). For an \( S \)-node whose sons are \( s_1, s_2, \ldots, s_k \), from right to left, with total orders \( \alpha_1, \beta_1; \alpha_2, \beta_2; \ldots; \alpha_k, \beta_k \), the total orders are \( \alpha = \alpha_1 \alpha_{k-1} \ldots \alpha_k \alpha_1, \beta = \beta_1 \beta_{k-1} \ldots \beta_k \). For a \( P \)-node, the total orders are \( \alpha = \alpha_1 \alpha_{k-1} \ldots \alpha_k \) and \( \beta = \beta_1 \beta_2 \ldots \beta_k \).

It should be evident that these total orders can be computed in \( O(n) \) time, given the decomposition tree of the series parallel digraph.

Note that not every two-dimensional partial order is transitive series parallel. This can be confirmed from the fact that the forbidden subgraph can be so represented, as shown in Figure 11. Please note that one representation of the forbidden subgraph is \( \alpha = (1,2,3,4), \beta = (2,4,1,3) \). One can always renumber the nodes so that one of the partial orders is \( \alpha = (1,2,\ldots,n) \). Then all information about the partial order can be obtained from \( \beta \). In particular, one can read off the complete adjacency matrix in \( O(n^2) \) time.

The preceding observations show that one can compute the transitive closure of a series
Forbidden Subgraph is Two-Dimensional
Figure 11

parallel digraph in time $O(n^2)$. This contrasts with a time of $O(n^{2.81})$ for the best available transitive closure algorithm for general digraphs.

4. ISOMORPHISM OF TRANSITIVE SERIES PARALLEL DIGRAPHS

The status of the graphic isomorphism problem is unclear. Specifically, we know that the problem is in NP, but we do not know if it is either polynomial-bounded or NP-complete. We do know that a number of problems are polynomial equivalent to the graphic isomorphism problem: isomorphism of semigroups, digraphs, bipartite graphs, chordal graphs, and series parallel digraphs.

We present a simple reduction of the (undirected) graph isomorphism problem to the problem of isomorphism for series parallel digraphs.

For a given graph $G = (N,A)$, construct a digraph $G' = (N',A')$ with $m+n+1$ nodes, as follows. $N' = N \cup A \cup \{\ast\}$, where "\ast" is a special node. There is an arc $(i,j,\ast) \in A'$ from each "arc" node of $G'$ to $\ast$, and an arc $(\ast,j)$ from $\ast$ to each "node" node of $G'$. There are also arcs $(i,j,1)$, $(i,j,2)$ from each "arc" node $(i,j)$ to the "node" nodes $i$ and $j$. The resulting
digraph $G'$, indicated schematically in Figure 12, is clearly series parallel.

We also assert, without formal proof, that two graphs $G_1, G_2$ are isomorphic if and only if $G'_1, G'_2$, constructed as above, are isomorphic. It therefore follows

![Series Parallel Digraph in Isomorphism Reduction](image)

that we cannot expect to solve the isomorphism problem for series parallel digraphs without solving the general graphic isomorphism problem as well.

What we shall do is to solve the isomorphism problem for transitive series parallel digraphs. Or to put it another way, we shall be able to test two series parallel digraphs $G_1, G_2$, for isomorphism of their transitive closures.

It is apparent that the decomposition tree for a series parallel digraph is also a decomposition tree for its transitive closure. We also noted, but shall not provide a formal proof, that the decomposition tree of a transitive series parallel digraph is uniquely determined, up to a reordering of the sons of the P-nodes. It follows that the isomorphism problem for transitive series parallel digraphs reduces to an isomorphism problem for decomposition trees.

Suppose $T_1$ and $T_2$ are the decomposition trees for the transitive series parallel digraphs $G_1, G_2$. These graphs can be isomorphic only if both $T_1, T_2$ have an S-node or a P-node as roots. It follows that at each level, $T_1, T_2$ each
contain only S-nodes and leaves or only P-nodes and leaves.

We shall test for isomorphism of \(T_1, T_2\) by working from level to level, from the leaves toward the roots. We shall speak of a node of \(T_1\) as being "isomorphic" to a node of \(T_2\) if the subgraphs induced on their descendents are isomorphic. Clearly each leaf of \(T_1\) is isomorphic to each leaf of \(T_2\), since all single-node digraphs are isomorphic.

Now suppose we have determined isomorphism of all pairs of nodes at a P-level and we wish to determine isomorphism for nodes at the S-level immediately above. Suppose node \(s_1\) of \(T_1\) has sons \(p_1, p_2, \ldots, p_k\), from right to left and node \(s_2\) of \(T_2\) has sons \(p'_1, p'_2, \ldots, p'_k\). Clearly \(s_1\) can be isomorphic to \(s_2\) only if \(k = k'\). Assuming this is the case, \(s_1\) is isomorphic to \(s_2\) if and only if \(p_1, p'_1, p_2, p'_2, \ldots, p_k, p'_k\) are pairwise isomorphic. It follows that it is easy to determine isomorphism of S-nodes from isomorphism of their sons.

Now suppose we have determined isomorphism of all pairs of nodes at an S-level and we wish to determine isomorphism for nodes at the P-level immediately above. Let node \(p_1\) of \(T_1\) have sons \(s_1, s_2, \ldots, s_k\) and node \(p_2\) of \(T_2\) have sons \(s'_1, s'_2, \ldots, s'_k\). Again, clearly \(p_1\) can be isomorphic to \(p_2\) only if \(k = k'\). Assuming this is true, \(p_1\) is isomorphic to \(p_2\) if and only if it is possible to establish a one-to-one correspondence (a complete matching) between isomorphic sons of \(p_1\) and \(p_2\). Note that isomorphism is an equivalence relation. Hence if \(s_1 = s'_1\) and \(s_2 = s'_2\), then \(s_1 = s_2\). A necessary and sufficient condition for a complete matching to exist is that for each \(s_1\), \(s_1\) is isomorphic to \(q\) sons \(s'_j\), \(q \geq 1\) and that each of these sons be isomorphic to exactly \(q\) sons \(s_1\). It follows that if a complete matching exists, it can be found by a simple algorithm, as follows:

**Step 0** Set \(i = 0\). No nodes are matched.

**Step 1** If \(i = k\), then \(p_1\) and \(p_2\) are isomorphic. Otherwise, set \(i = i + 1\).

**Step 2** Try to find an unmatched node \(s'_j\), isomorphic to \(s_1\). If no such node exists, stop; \(p_1\) and \(p_2\) are not isomorphic. If such a node \(s'_j\) can be found, match \(s_1\) with \(s'_j\), and return to step 1.

The two transitive series parallel digraphs are isomorphic if and only if their root nodes are isomorphic. It is evident that this can be determined
in $O(n^2)$ time.

5. SUBGRAPH ISOMORPHISM

The subgraph isomorphism problem is to determine if a given graph $G_1$ is isomorphic to a subgraph of a second graph $G_2$. It is well-known that the subgraph isomorphism problem is NP-complete. This result follows immediately from the fact that the maximum independent set problem is NP-complete, since this problem is equivalent to determining if a given complete graph is isomorphic to a subgraph of a second graph.

The subgraph isomorphism problem can, however, be solved efficiently for transitive series parallel digraphs.

For simplicity, let us suppose that the decomposition trees for $G_1$, $G_2$ have the same type of root node ($S$ or $P$). If this is not the case, then the problem becomes that of determining if the root for $G_1$ is isomorphic to a son of the root node for $G_2$. (Note that we shall continue speaking of "isomorphism" of nodes, when what we mean is "isomorphism of the subgraph induced by the descendents of the node in $T_1$ to a subgraph of the subgraph induced by the descendents of the node in $T_2".)

We must now make some modifications of the rules stated in the previous section.

As before, suppose we have determined isomorphism for all pairs of nodes at a $P$-level and we wish to determine isomorphism for nodes at the $S$-level immediately above. Suppose node $s_1$ of $T_1$ has sons $p_1, p_2, \ldots, p_k$ from right to left, and node $s_2$ of $T_2$ has sons $p'_1, p'_2, \ldots, p'_k$, where $k' \geq k$. We find the rightmost son, $p'_1$ of $T_1$ to which $p_1$ is isomorphic, then the rightmost son, $p'_2$ of $T_2$ to which $p_2$ is isomorphic. The nodes $s_1$, $s_2$ are isomorphic if and only if all the nodes $p_1, p_2, \ldots, p_k$ can be matched up in this way.

Now suppose we have determined isomorphism of all pairs of nodes at an $S$-level and we wish to determine isomorphism for nodes at the $P$-level immediately above.

Suppose node $p_1$ of $T_1$ has sons $s_1, s_2, \ldots, s_k$ and node $p_2$ of $T_2$ has sons $s'_1, s'_2, \ldots, s'_k$, where $k' \geq k$. Node $s_1$ is isomorphic to $s_2$ if and only if the nodes $s_1, s'_2, \ldots, s'_k$ can be matched to distinct nodes from among $s'_1, s'_2, \ldots, s'_k$. 
This is a bipartite matching problem which can be solved in $O(k')^{5/2}$ time [2]. Let $k_j$ denote the number of sons of the $j$th $P$-node at level $i$ of the $T_2$. Then

$$
\sum_j k_j^{5/2} \leq \left( \sum_j k_j \right)^{5/2} = n_{i+1}^{5/2},
$$

where $n_{i+1}$ is the number of nodes at level $i+1$. But

$$
\sum_i n_i \leq 2n,
$$

$$
\sum_i n_i^{5/2} \leq \left( \sum_i n_i \right)^{5/2} < (2n)^{5/2}.
$$

It follows that the overall running time required for solving all matching problems is $O(n^{5/2})$. All other operations are dominated by matching, hence subgraph isomorphism can be tested in $O(n^{5/2})$ time.

Essentially the same approach was used by Edmonds and by Matula [8] for the problem of determining whether one tree is isomorphic to a subtree of another. The present procedure can be viewed as an extension of their techniques to a broader class of isomorphism problems.

6. FINDING MINIMUM SPANNING TREES

Let $G = (N,A)$ be a connected undirected graph, with a value $c(v,w)$ specified for each arc $(v,w) \in A$. We wish to find a spanning tree $T = (N,A')$, such that

$$
\sum_{(v,w) \in A'} c(v,w).
$$

This problem has been solved in time $O(n^2)$ or $O(m \log n)$. Recently, a variety of algorithms have been suggested, all with $O(m \log \log n)$ running time [2, 14]. In the present section we describe what appears to be the simplest of all known algorithms with $O(m \log \log n)$ running time. The presentation is adapted from Cheriton and Tarjan [2].

**Lemma 1.** Let $X \subseteq N$. Let $(v,w) \in A$ be such that $v \in X$, $w \in N - X$, and $c(v,w) = \min\{c(x,y) \mid (x,y) \in A, x \in X, y \in N - X\}$. Then some minimum spanning tree contains $(v,w)$. 
This lemma justifies the following very general procedure:

**First step:** Pick some node. Choose the smallest arc incident to the node. This is the first arc of the spanning tree.

**General step:** The arcs so far selected define a forest (a graph consisting of a set of trees) which is a subgraph of G. (Isolated nodes, not incident to any arcs selected so far, are regarded as trees with one node.) Pick a tree in the forest. Select the smallest unselected arc \((v,w)\) incident to a node in \(T\) and to a node in a different tree \(T'\). Delete all arcs smaller than \((v,w)\) which are incident to \(T\) (these arcs form cycles in \(T\)). Add \((v,w)\) to the minimum spanning tree (updating the forest by combining \(T\) and \(T'\)). Repeat the general step until all nodes are connected.

This general method requires a mechanism for keeping track of the subtrees in the forest. To keep track of the nodes in each tree of the forest, we propose to use the well-known "disjoint set union" algorithm [13]. Given a collection of disjoint sets (in this case sets of the nodes in each tree) the set union algorithm implements three operations:

- **FIND** \((x)\) : returns the name of the set containing node \(x\).
- **UNION** \((i,j)\) : combines sets \(i\) and \(j\), naming the new set \(i\).
- **INIT** \((i,L)\) : initializes set \(i\) to contain all nodes in list \(L\).

The worst-case time required for \(O(m)\) FINDs and \(O(n)\) UNIONs is \(O(n \alpha (m,n))\), where \(\alpha\) is a functional inverse of Ackermann's function [13]. Initialization requires \(O(n)\) time for \(n\) trees, each consisting of a single node.

To keep track of the arcs incident to each tree of the forest, we use mechanisms of priority queues. We maintain queues of all arcs incident to each tree of the forest (except, of course, those arcs which have been deleted). We need three operations on these queues:

- **\(\text{QUNION} (i,j)\)** : combines queues \(i\) and \(j\), naming the new queue \(i\).
- **MIN** \((i)\) : returns the smallest arc in queue \(i\) which has an endpoint outside tree \(i\). MIN \((i)\) also deletes this arc and all smaller arcs from queue \(i\). (Note: MIN \((i)\) must use the FIND operation to test whether a given arc has an endpoint outside tree \(i\).)
- **\(\text{QINIT} (i,L)\)** : initializes queue \(i\) to contain all arcs in list \(L\).

An implementation of the general algorithm using these operations as primitives appears below. We assume the graph has node set \(N = \{1, 2, \ldots, n\}\) and
that for each node \( v \), \( I(v) = \{(v,w) \mid (v,w) \in A\} \) is a list of the arcs incident to \( v \). (Note that \((v,w)\) is considered an ordered pair \((v,w) \in I(v), (w,v) \in I(w)\).) Each tree \( i \) of the forest is represented by a set \( i \) of its nodes and by a queue \( i \) of its incident arcs. Initially each node \( v \) is represented by the set \( \{v\} \) and by a queue consisting of the arcs in \( I(v) \).

At all times during execution of the algorithm, every arc \((v,w)\) in queue \( k \) satisfies \( \text{FIND}(v) = k \).

**General Algorithm**

```
For i = 1 until n do
  begin
    INIT(i,\{i\});
    QINIT(i,I(i));
  end
While more than one tree do
  begin
    pick some tree K;
    (i,j) = MIN(K);
    add arc \((i,j)\) to spanning tree;
    x = FIND(j);
    UNION(x,k);
    UNION(x,k)
  end;
```

We shall consider two methods for implementing priority queues:

(a) unordered lists and (b) ordered lists of size \( k \).

(a) Unordered Lists

\( \text{QINIT} (i,L) \) requires \( O(m(i)) \) time where \( m(i) \) is the size of queue \( i \) (and of \( L \)).

\( \text{MIN} (i,L) \) requires \( O(m(i)) \) time plus time for \( m(i) \) FINDs.

\( \text{UNION} (i,j) \) requires \( O(1) \) time.

(b) Ordered Lists of Size \( k \)

Another possibility is to represent a queue as a list of sets, each set initially of size \( k \), plus possibly one set of size less than \( k \). Each set is in sorted order, but there is no ordering relationship among the sets. We
carry out QINIT \((i,L)\) by dividing \(L\) into \(\lfloor |L|/k\rfloor\) sets of size \(k\) plus at most one set of size less than \(k\). We then sort these sets. This process requires \(O(|L| \log k)\) time. MIN \((i)\) is carried out by inspecting the arcs in each set of queue \(i\) in order, discarding those that don't connect two different trees. We then compare the smallest arc from each set and select the overall minimum arc. MIN \((i)\) requires \(O(s(i) + d(i))\) time, plus time for \(s(i) + d(i) - 1\) FINDs, where \(s(i)\) is the number of sorted sets in queue \(i\) and \(d(i)\) is the number of arcs deleted from queue \(i\) by MIN \((i)\).

(Note: \(d(i) \geq 1\), always, because the arc returned by MIN \((i)\) is deleted.) QUNION \((i,j)\) is accomplished by simply merging the list of sets in queue \(i\) and the list of sets in queue \(j\). This requires \(O(1)\) time.

We shall use the "uniform" selection rule to determine the order in which trees are processed. This works as follows. Initially all of the \(n\) trees (each a single node) arc placed on a queue. At the general step, the first tree \(T\) at the front of the queue is examined. When this tree \(T\) is connected to another tree \(T'\), both \(T\) and \(T'\) are deleted from the queue and the new combined tree is placed at the rear of the queue. It is easy to implement this selection rule so that the total overhead for selection is \(O(n)\).

Without loss of generality, assume that the nodes of the graph are numbered, so that the \(i^{th}\) execution of MIN is MIN \((i)\). Let \(m(i)\) be the number arcs in queue \(i\) then MIN \((i)\) is executed. For each \(i\), \(1 \leq i \leq n-1\), let \(T_i\) be the tree in \(F\) containing node \(i\) when MIN \((i)\) is executed. \(T_n\) is the minimum spanning tree constructed.

We define a stage number, \(st(T)\) for \(T\), by \(st(T) = 0\), if \(T\) contains a single node, and \(st(T) = \min(st(T'), st(T'')) + 1\), if \(T\) is formed by connecting trees \(T'\) and \(T''\) by an arc. It is easy to prove by induction that if \(st(T) = j\), then \(T\) has at least \(2^j\) vertices, so there are at most \(\lceil \log n \rceil\) stages at which trees are connected.

**Lemma 2.** Two different trees \(T_i\) and \(T_j\) with the same stage number are node disjoint.

**Proof.** Suppose \(T_i\) and \(T_j\) share a node. Then either \(T_i \subseteq T_j\) or \(T_j \subseteq T_i\).

Without loss of generality, assume \(T_i \subseteq T_j\). The trees initially on the queue have nondecreasing stage numbers (all zero), and the property is preserved as the algorithm proceeds. Just before MIN \((i)\) is executed, \(T_i\) has a stage number as small as any tree on the queue. Thus, if \(T_i \neq T_j\), all
trees $T$ on the queue which are subtrees of $T_j$ have $st(T) \geq st(T_j)$, which
means $st(T_j) \geq st(T_j) + 1$. □

**COROLLARY 3.** For any stage $s$, $0 \leq s < \log n$,

$$\sum_{st(T_i) = s} m(i) \leq 2m.$$  

Moreover,

$$\sum_{i=1}^{n-1} m(i) \leq 2m \log n.$$  

**Proof.** Each arc is represented at most twice in the queues, and
$0 \leq st(T_i) \leq \log n-1$, if $1 \leq i \leq n-1$, by the proof of Lemma 2 and the
fact that $|T_i| \geq 2^{st(T_i)}$. □

We now propose to implement the algorithm as follows:

**Phase 1:** Initialize all queues as unordered lists, and execute the algorithm
until stage $\log n$.

**Phase 2:** Re-initialize all queues as ordered lists of size $k = \log n$, and
execute the algorithm until completion.

The time required for the operations QINIT and MIN in the table below.
Note that the total number is UNIONS performed is exactly $n - 1$, with
$O(1)$ time for each UNION. The time required for FINDs and UNION is domi-
nated by $O(m \log \log n)$, which is seen to be the running time for the over-
all algorithm.

<table>
<thead>
<tr>
<th></th>
<th>QINIT</th>
<th>MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>$O(m)$</td>
<td>$O(m \log \log n)$</td>
</tr>
<tr>
<td>Phase 2</td>
<td>$O(m \log \log n)$</td>
<td>$O(m)$</td>
</tr>
</tbody>
</table>

Note that if unordered lists were used for the entire computation, the time
for MIN would be $O(m \log n)$. If ordered lists of size $\log n$ were used for
the entire computation something like $m \log n$ sets could be created at
initialization, and the time for MIN would then contain a term of the form
$n \log n$. Thus two phases are necessary to attain the desired running time of
$O(m \log \log n)$.
7. GENERATING ALL MAXIMAL INDEPENDENT SETS

An independent set of nodes of a graph \( G = (N,A) \) is a subset \( I \subseteq N \) such that no two nodes in \( I \) are adjacent. There are situations, e.g. in graph coloring, in which one may want to generate all the maximal independent sets of a given graph. This can be accomplished very efficiently by an algorithm due to Tsukiyama, et al [16].

Note that an independent set in \( G \) can be identified with a complete subgraph in the complement of \( G \). Also, the complement of an independent set is a covering of arcs by nodes. Thus any algorithm for generating maximal independent sets can be used to generate maximal cliques or minimal covers as well.

Before describing the algorithm of the four Japanese, let us review an older algorithm due to Paul and Unger [10].

Let \( I_j \) denote the family of all independent sets which are maximal in the subgraph induced on nodes 1,2,...,j. Clearly \( I_1 = \{\{1\}\} \). We then propose to find \( I_{j+1} \) from \( I_j \), finally obtaining \( I_n \), the family of maximal independent sets of \( G \) itself.

Let \( I \in I_j \), and let \( A(j+1) \) denote the set of nodes adjacent to node \( j+1 \). If \( I \cap A(j+1) = \emptyset \), it is clear that \( I \cup \{j+1\} \in I_{j+1} \). And if \( I \cap A(j+1) \neq \emptyset \), then \( I \in I_{j+1} \). We thus deduce that

\[
|I_1| \leq \ldots \leq |I_j| \leq |I_{j+1}| \leq \ldots \leq |I_n| = K.
\]

Now consider a typical set \( I' \in I_{j+1} \). If \( j+1 \not\in I' \), then \( I' \in I_j \). If \( j+1 \in I' \), then \( I' - \{j+1\} \subseteq I_j \), for some \( I \in I_j \) and \( I' = (I - A(j+1)) \cup \{j+1\} \).

It thus follows that if we form the multiset

\[
I_{j+1}' = (I' \mid I' = (I - A(j+1)) \cup \{j+1\}, I \in I_j),
\]

then

\[
I_{j+1} \subseteq I_j \cup I_{j+1}'.
\]

It is very important to note that we refer to \( I_{j+1}' \) as a multiset. That is, \( I_{j+1}' \) may contain duplicate entries. A given set \( I' \in I_{j+1}' \) may result from two distinct sets \( I_1, I_2 \in I_j \). That is,

\[
(6.1) \quad I' - \{j+1\} = I_1 - A(j+1) = I_2 - A(j+1),
\]
even though $I_1 \neq I_2$.

In order to obtain $I_{j+1}$ from $I_j \cup I'_{j+1}$, it is necessary to purge from $I_j \cup I'_{j+1}$ those sets which are either nonmaximal or duplicates of other entries. Since $|I_j| = |I'_{j+1}| \leq k$, this can be done with at most $O(K^2)$ pairwise comparisons of sets.

Now suppose $A(j+1)$ is given by a list and each $I \in I_j$ is specified by an $n$-vector. Then $O(|A(j+1)| K)$ time suffices to form the multiset $I'_{j+1}$. Each set comparison requires $O(n)$ time, so purging $I_j \cup I'_{j+1}$ of nonmaximal and duplicate entries can be done in $O(nK^2)$ time. It is clear that the overall running time of the procedure is bounded by $O(n^2 K^2)$ and storage $O(mnK)$.

What the four Japanese have done is to reduce the running time to $O(nmK)$ and space requirements to $O(m+n)$. Their key contribution is to show how to form $I'_{j+1}$ without introducing either nonmaximal or duplicate sets.

The avoidance of nonmaximal sets is very simple. Let $I \in I_j$. Then

$$I' = (I - A(j+1)) \cup \{j+1\}$$

is a maximal independent set on the subgraph induced on $1, 2, \ldots, j+1$, if and only if, for all $k < j+1, k \notin I$,

$$A(k) \cap I' \neq \emptyset.$$  

(6.2)

This test can be performed for all $I' \in I_{j+1}$ in $O((n+m)K)$ time.

Now let us consider the matter of duplication. Suppose a situation were to exist, as in (6.1). The rule we shall use is the following. For a given set $I \in I_j$, we shall introduce $I' = (I - A(j+1)) \cup \{j+1\}$ into $I_{j+1}$ only if $I \cap A(j+1)$ is lexicographically smallest, with respect to all sets $I \in I_j$ for which $I - A(j+1)$ is the same.

Recall that lexicography induces a total order. Let $I_1 \neq I_2$, where

$$I_1 = \{e_{j(1)}, e_{j(2)}, \ldots, e_{j(p)}\},$$

$$I_2 = \{e_{j'(1)}, e_{j'(2)}, \ldots, e_{j'(q)}\},$$

$j(1) < j(2) < \ldots < j(p); j'(1) < j'(2) < \ldots < j'(q)$, and $p \leq q$. Suppose there is an $k \leq p$, such that $j(i) = j'(i)$, for $i = 1, 2, \ldots, k-1$, and $j'(k) < j(k)$. Then we say that $I_2$ is lexicographically smaller than $I_1$; otherwise $I_2$ is lexicographically larger.
In words, a set $I \in I_j$ satisfies the lexicography condition if and only if there does not exist a node $k < j+1$, $k \notin I$ which is nonadjacent to both $I - A(j+1)$ and to all lower-numbered nodes in $I \cap A(j+1)$. In other words, if and only if, for all $k < j+1$, $k \notin I$,

$$A(k) \cap (I-(A(j+1) \cap \{k+1,k+2,...,j\})) \neq \emptyset.$$ 

This test can also be performed for all $I \in I_j$ in $O((n+m)K)$ time.

It is now apparent that $I_{j+1}$ can be formed from $I_j$ in $O((n+m)K)$ running time. Hence $I_n$ can be obtained in $O(n(n+m)K)$ time overall. Assume $m \geq n$, and let us call this time $O(mnK)$. (If $m < n-1$, then there is at least one node with degree zero, and any such node is contained in every maximal independent set.)

Let us now see how storage requirements can be reduced. The process of generating the maximal independent sets can be visualized by means of a tree, as shown in Figure 14, for the graph in Figure 13.

![Graph for Example](Figure 13)

![Generation of Maximal Independent Sets for Example](Figure 14)
The nodes at level \( j \) of the tree are identified with sets in \( I_j \), with the tree rooted to \( \{1\} \), the unique set in \( I_1 \). Each node has at least one son. (Recall that for each \( I \in I \), either \( I \) or \( I \cup \{j+1\} \) is in \( I_{j+1} \).) The left son, if it exists, is identified with \( (\mathcal{I} - \mathcal{A}(j+1)) \cup \{j+1\} \). This son is provided only if the maximality and lexicography conditions are satisfied. The right son, if it exists, is identified with \( I \), but this son exists only if \( I \cap \mathcal{A}(j+1) \neq \emptyset \). (otherwise \( I \) is not maximal in \( I_{j+1} \), being a subset of \( I \cup \{j+1\} \in I_{j+1} \).

We now carry out our computation in the form of a depth-first search of the tree, always traveling downward to the left-most son of a node that we have not yet visited. When we reach a node in \( I_n \), we simply output the maximal set in question (rather than maintaining it in storage). We then backtrack. The backtracking procedure is facilitated by maintaining a stack. We leave it as an exercise for the reader to show that the stack can be implemented in \( O(n) \) space.

8. FINDING A MAXIMUM INDEPENDENT SET

Let us now consider the problem of finding a maximum (i.e., maximum cardinality) independent set in a graph. As we know, this is an NP-complete problem, so we can hardly hope to find such a set in running time which is polynomial in \( m \) and \( n \).

It is a well-known result that there are at most \( 3^{n/3} \) maximal independent sets in a graph [10]. It thus follows that the algorithm of the previous section can be applied to solve the problem in \( O(mn(3^{1/3})^n) \) time. Note: \( 3^{1/3} \approx 1.445 \).

Tarjan and Trojanowski [14] have shown that the problem can be solved in \( O(2^{n/2}) \) time. We shall not go into the complete elaboration of their procedure, but only indicate the central idea behind their procedure.

First note that the problem is trivial if the maximum degree of any node in \( G \) does not exceed 2. In this case, each component consists of a chain or a cycle, and it is easy to pick out a maximum number of nodes from each component to be part of an independent set.

Hence let us assume that there is at least one node \( j \) of degree three or more. Obviously a maximum independent set either (a) contains \( j \) or (b) it does not. If we make the former assumption, then none of the nodes in \( A(j) \) can be contained in the solution. It then suffices to find an inde-
pendent set in the graph from which \( j \) and the nodes in \( A(j) \) are deleted. In the latter case, we are left with the problem of finding the best solution we can in the subgraph obtained by deleting node \( j \) only. The first subgraph has at most \( n-4 \) nodes, and the latter \( n-1 \).

Let \( c(n) \) be the running time required to solve the problem for an \( n \)-node graph, by our procedure. Then we have

\[
c(n) \leq c(n-1) + c(n-4) + p(n),
\]

where \( p(n) \) is a polynomial function required for housekeeping, comparing solutions for the two subproblems, etc.

Assuming \( c(n) = a^n \), for some base \( a \), an upper bound on the order of \( c(n) \) is given by solving for \( a \) such that

\[
a^n = a^{n-1} + a^{n-4}
\]

or

\[
a^4 = a^3 + 1.
\]

This indicates \( a \approx 1.38 \), which, of course, is better than \( 3^{1/3} \approx 1.445 \).

Tarjan and Trojanowski succeeded in reducing the base \( a \) to less than \( 2^{1/3} \) by a much more involved case analysis.

9. Computing the Chromatic Number of a Graph

A \( k \)-coloring of a graph \( G = (N, A) \) is a partition of the node set \( N \) into \( k \) independent sets. If there exists such a partition, \( G \) is said to be \( k \)-colorable. The chromatic number of \( G \) is the least value of \( k \) such that \( G \) is \( k \)-colorable.

It is well known that the chromatic number problem is \( \text{NP} \)-complete. In fact, even the problem of determining the chromatic number of a graph to within a given factor \( r < 2 \) has been shown to be \( \text{NP} \)-complete [3].

A number of algorithms for this problem have been proposed [9], yet there appears to have been very little discussion in the literature concerning nontrivial upper bounds on the complexity of the problem. We propose to make use of the results obtained above concerning maximal in-
dependent sets to obtain an algorithm with worst case running time of \( O(mn(1+3/\sqrt{3})^n) \). Note: \( 1 + \frac{3}{\sqrt{3}} \approx 2.445 \).

It is easy to see that if a graph is \( k \)-colorable, at least one of the \( k \) colors may be assumed to be a maximal independent set. For a given graph \( G = (N,A) \), let \( K(N') \) denote the chromatic number of the subgraph induced on \( N' \subseteq N \). It follows that if \( N' \) is nonempty there exists a maximal independent set \( I \) of the subgraph induced on \( N' \), such that

\[
K(N') = 1 + K(N'-I).
\]

There are a finite number of maximal independent sets \( I \) of the induced subgraph. By minimizing over them we obtain the "dynamic programming" equations

\[
K(N') = 1 + \min_{I \subseteq N'} \{K(N'-I)\}, \; N' \neq \emptyset
\]

\[
K(\emptyset) = 0.
\]

We now estimate the running time required to solve these equations for all \( N' \subseteq N \), in the worst case.

Suppose, for fixed \( N' \), we have already found \( K(N'') \), for all proper subsets \( N'' \subseteq N' \). The time required to compute \( K(N') \) is then proportional to the number of maximal independent sets of the subgraph induced on \( N' \), plus the time required to generate them. But, as we have seen, this time is no worse that \( O(mr^{r/3}) \), where \( |N'| = r \). Summing over all possible \( N' \subseteq N \), and invoking the binomial theorem, we obtain

\[
\sum_{r=0}^{n} \binom{n}{r} mx^{r/3} < mn \sum_{r=0}^{n} \binom{n}{r} x^{r/3} = mn(1+3/\sqrt{3})^n.
\]

This yields the desired result.
REFERENCES


[8] D.W. MATULA, Subtree isomorphism in \( O(n^{5/2}) \), to appear.


Note added in proof.

It appears that it is possible to recognize series parallel digraphs in $O(m)$ time. This newer procedure will be communicated in [7].
THE COMPLEXITY OF DATA ORGANIZATION

J. VAN LEEUWEN *
State University of New York, Buffalo, USA

0. INTRODUCTION

0.1. The way data is presented to the computer and represented in the system can be a dominating factor in the performance of the computer as a data-processing tool.

By its very nature the area of data-organisation extends from the study of combinatorial algorithms to the design of information management systems, and it is too broad to be conveyed in a series of six lectures only.

We decided to emphasize a few topics in concrete complexity theory which have recently received much attention and which have lead to a number of fundamental techniques and new results which may be applicable to many problems which computer-programmers encounter.

0.2. The present lectures neither emphasize (say) a specific domain in computational complexity nor give you the design-philosophy of relational database, but instead we will concentrate on useful techniques (tricks?) in data-organisation which can bring improvements in many computer-programs performing data-manipulations of some sort.

We have specifically avoided to present results which are already adequately treated in such excellent texts as Knuth [43],[44] or Aho, Hopcroft, and Ullman [3].

The reader is assumed to have some familiarity with computer-programming and the fundamental information-structures as discussed in Knuth [43] or Wirth [66]. Roberts [54] gives a useful survey of the basic file-organisation techniques.

*) Present address: Dept. of Computer Science, Whitmore Laboratory, the Pennsylvania State University, University Park, Pennsylvania 16802.
fig.1. Some terminology

* Sometimes called "file-organisation" and "data-organisation" respectively.
1. EFFICIENCY VERSUS DATA-REPRESENTATION

1.1. Let us start from an intuitive concept of data. The user is largely responsible for data-collection, and must make sure that he gathers the information needed for a successful data-processing system.

1.2. To facilitate retrieval and efficient manipulation information should not be stored randomly, but is preferably organised in structures which allow for easy access and (say) deletion and addition of data at all times.

1.3. Computer storage is divided into directly addressable core-storage ("main memory") and "supplementary" or secondary storage ("auxiliary memory"). Data is usually stored in auxiliary memory on mass-storage devices like magnetic tapes, magnetic drums, (magnetic) cards, data-cells, or disc-packs, and only small parts will be held for processing in main memory at a time. In time-shared systems the data will necessarily be segmented, with due restrictions on size per segment.

1.4. The user is not likely to be concerned with the "hardware" of a data management system, but is using an intermediate language instead. This can be a "host"-language (i.e. some general purpose programming language) providing helpful primitive data-structures and a flexible mode-definition mechanism for creating more complex structures, or a special data-language.

Data-structures are merely the model of data-storage as observed by the user. The "system" presumably interprets his model in "real" storage.

1.5. Information is usually provided in small packages of logically connected data called records. The fields of a record must be specified, and may consist of other records. The information in a record can be accessed by using the field names as selectors.

1.6. Some fields may be used (or added) for uniquely identifying a record. The contents of such fields together form the primary key of a record.

Later we shall simply identify a record and its primary key (as it is the only part our algorithms will be using), and define it to be an ordered k-tuple \((b_1, b_2, \ldots, b_k)\) of values taken from some ordered set \(R\).
1.7. Consider an arbitrary, unstructured collection of (perhaps) randomly stored records

\[ V = \{ v_1, v_2, \ldots, v_N \} \subseteq \mathbb{R}^k. \]

Numerous address-calculation schemes exist for "finding" a record when its primary key is presented (Knuth [44]).

1.8. The simplest method would be to interpret keys as indices, and to store the addresses in a table (fig. 2). Sequential search is easy to program, but will require an average of \( N/2 \) probes. If keys were sorted in lexicographic order, then presumably some kind of binary search over "outgoing" edges for each successive component of the key would reduce the search-time to \( \sim \log N \) (at worst) per component. Fredman [31] indicated how an information-theoretic argument can help to improve it further. (The proof we develop here seems new.)

1.9. Consider an arbitrary node reached in progressing down the tree

\[ a_1 < a_2 < \ldots < a_M \]
Fig. 2. A table and a (lexicographic) tree
Suppose "son_1" has $p_1$ descendant leaves, $p_1 < N$. In searching for an $i$ such that $a_i = b$ (sending us off to son_1), it is useful in probing to emphasize more frequently occurring $a$'s.

The proper formulation makes use of a binary search-tree: each node in such a tree carries a query

\[
\text{if } b \leq X \quad \text{then } Y \quad \text{else } Z
\]

......go left,

......go right, or

"is son_1" (found it!)

**Lemma.** There exists a search-tree for finding $b$ in

\[
\leq \lceil \log(p_1 + \ldots + p_M) \rceil - \lfloor \log(p_1 + 1) \rfloor + 1 \text{ steps},
\]

where $i$ is such that $b = a_i$.

**Proof.** Make $p_j$ copies of $a_j$ (for each $j = 1, \ldots, M$), and build a binary tree of minimum path-length

```
          O
         /|
        / |  
       /  |   
      /   |    
     /    |     
    /     |      
   /      |       
  /       |        
 a_j     a_j     a_j
   .... p_j copies    ....
```

Call a node $j$-critical if and only if all its descendant leaves carry $a_j$ and there is no node less deep in the tree with that property. For each $j$ these are one or two $j$-critical nodes.

Make it a search-tree by assigning appropriate queries topdown such that the search for some $b$ (presumably $a_i$) is always directed to an
i-critical node. All nodes that are irrelevant now (thus, not on a search-path to some critical node) may be purged, after which the tree may have to be condensed back to binary form (thus reducing the search-length of some paths even further).

The identity of b is uncovered at the father of a critical node (and it obviously is the fastest way of identifying it).

The height of the "father" is bounded by the smallest t such that \( d_t = 0 \) in the following recurrence

\[
\begin{align*}
  d_0 &= p_1 \\
  d_s &= \left\lfloor \frac{s-1}{2} \right\rfloor, \quad \text{for } s = 1, 2, \ldots
\end{align*}
\]

which is \( t = \lceil \log(p_1+1) \rceil \). Thus a decision is found after \( \lceil \log(p_1+\ldots+p_k) \rceil - 1 \) \( \rightarrow \lceil \log(p_1+1) \rceil + 1 \) steps. \( \square \)

1.10. The resulting search-tree (after all unnecessary information has been purged) takes only little more space than was needed at the node of 1.9 anyway, and should be substituted. The original lexicographic tree becomes a cascade of local search-trees, and it enables one to find the address of a record \((b_1, b_2, \ldots, b_k)\) quickly.

**Theorem.** A record can be identified within \( \sim \log N + 2k \) queries.

**Proof.** Follow the trail of identifying \((b_1, \ldots, b_k)\) (fig. 3).

By lemma 1.9 the total number of queries needed is bounded by

\[
\begin{align*}
  &\sum_{0}^{k-1} \left( \lceil \log N_1 \rceil - \lfloor \log(N_1+1) \rfloor + 1 \right) \\
  = \lceil \log N_0 \rceil + \sum_{1}^{k-1} (\lceil \log N_1 \rceil - \lfloor \log(N_1+1) \rfloor) - 1 + k \\
  \leq \log N + 2k. \quad \square
\end{align*}
\]

Recall that the original, unbalanced search-strategy did cost us about \( \sim \log N \) queries per component.

1.11. Large collections of logically related records are hardly ever stored
randomly, and are normally arranged in one or more perhaps specifically structured units called *files*. The reason for *file-design* (or *file-organisation*) is to store data appropriately on available storage-media to allow for efficient handling of the operations which the user requests.

1.12. In modern terminology a collection of stored operational files used by the application systems of some particular enterprise is called a *database* (Engles [25]).

1.13. Files can be distinguished by the supporting retrieval mechanism. The basic types are

(1) *sequential files*
- A record can be accessed only by scanning the entire file for the point where it is stored. It is common to *batch* successive record-
requests, and to avoid the need for rewinding the file for each
individual request.

(ii) random access files
- The address of a record is determined from its primary key with
  a simple (perhaps tabulated) function, and access to its location
  follows (almost) immediately.

(iii) indexed (sequential) files
- A derived key is used as index in a directory structure to yield
  the bucket where a record must be, and ordinary retrieval from
  the bucket follows.

1.14. An important task in programming consists of finding the best data-
organisation for an application in order that the data can be processed
most efficiently.

However, the data may not be available in most desirable form or may
be of much larger volume than can fit in core, and it can be equally impor-
tant to modify algorithms for an application to work better and better
under given constraints on the data-organisation.

These two directions of research together form the domain of the
complexity-theory of data organisation.

1.15. An interesting example to show how a given data organisation can
affect the choice of an algorithm is Warren's transitive closure algorithm
for 0-1 matrices (Warren [63]).

We shall first consider the more traditional algorithm of Marshall
(Warshall [64]) for this task.

1.16. An n× n 0-1 matrix \( M = (m_{ij}) \) can always be interpreted as the
adjacency-matrix of a directed graph \( G_M \) on \( \{1, \ldots, n\} \) where

\[
m_{ij} = 1 \quad \text{if } i \rightarrow j
\]

The 0-1 matrix \( M^* = (m^*_{ij}) \) is called the transitive closure of \( M \) if and
only if its coefficients satisfy

\[
m^*_{ij} = 1 \quad \text{if } \text{there is a path } i \rightarrow \ldots \rightarrow j \text{ (of length } \geq 0) \text{ in } G_M.
\]
1.17. Most transitive closure algorithms can be distinguished by the stepwise manner in which all paths \( i \rightarrow \ldots \rightarrow j \) are built up. One only needs to consider paths free of repeating nodes.

1.18. In Warshall’s algorithm one builds \( M^* \) in stages \( M^*(s) \) where

\[
M^*_i_j(s) = 1 \iff \text{there is a path } i \rightarrow \ldots \rightarrow j \in \{1, \ldots, s\} \text{ of length } \geq s \text{ in } G^*_M.
\]

1.19. Obviously \( M^*(0) = M \) and \( M^*(n) = M^* \).

1.20. For \( s = 1, \ldots, n \) one can construct \( M^*(s) \) from the previous stage by observing that

\[
m^*_i_j(s) = m^*_i_j(s-1) \lor m^*_i_s(s-1) \land m^*_s_j(s-1)
\]

fixed factor for row \( i \).

Denoting the \( i \)-th row of \( M^*(s) \) by \( M^*_i(s) \) we obtain the rule

\[
M^*_i(s) := \text{if } m^*_i_s(s-1) \text{ then } M^*_i(s-1) \lor M^*_s(s-1) \text{ else } M^*_i(s-1).
\]

The algorithm can now be formulated as

"let \( M^* \) be \( M \);
for \( s := 1 \) to \( n \) do
  for \( i := 1 \) to \( n \) do
    \[
    M^*_i := \text{if } m^*_i_s \text{ then } M^*_i \lor M^*_s
    \]
  od
od;

1.21. Warshall’s algorithm works very well if each row of the matrix can
be packed in a single word and words can be "or"-ed directly.

As a contrast it is of interest to evaluate the algorithm for (very) large matrices which are stored externally in a sequential file. (It also serves as an adequate model for a paging environment).

Let consecutive records correspond to consecutive rows of the matrix, and assume that one can store $\sim 2$ records (at least) in memory at a time.

We shall use the number of records "paged in" by the algorithm as a measure for its complexity.

1.22. The algorithm must now be formulated as

```
rewind;
for s := 1 to n do
    scan for record s;
    $M^*_s$ := get record;
    rewind;
for i := 1 to n do
    $M^*_i$ := get record
    if is then reset record;
    $M^* := M^*_i \vee M^*_s$;
    put $M^*_i$
```

and it follows that $\sim n^2 + n$ records must be read into memory.

1.23. With the given (sequential) organisation of the data Warshall's algorithm immediately becomes less attractive because the need for the coefficient $M^*_{1s}$ forces one to read in every record even if there is no subsequent action on the record.

In an algorithm recently proposed by Warren ([63]) this has been eliminated, and records are paged in only when necessary. (Our proof seems to be new.)

1.24. In Warren's algorithm $M^*$ is built up in two passes.

Pass I yields an intermediate matrix $M^\theta$ in stages $M^\theta(s)$ where
\[ m_{ij}^s(s) = 1 \quad \text{there is a path} \]
\[ i \rightarrow \ldots \rightarrow j \]
\[ \epsilon \{1, \ldots, i-1\} \]
\[ \text{for all } i \leq s \]

and pass II subsequently yields \( M^*(s) \) in stages \( M^*(s) \) where

\[ m_{ij}^*(s) = 1 \quad \text{there is a path} \]
\[ i \rightarrow \ldots \rightarrow j \]
\[ \epsilon \{i+1, \ldots, n\} \]
\[ \text{for all } i \leq s. \]

1.25. Obviously \( M^*(1) = M^* \), \( M^*(0) = M^*(n) \), and \( M^*(n-1) = M^* \).

For \( s = 2, \ldots, n \) one can construct \( M^*(s) \) from the previous stage by updating row \( s \). One can similarly construct \( M^*(s) \) from the previous stage for \( s = 1, \ldots, n-1 \).

1.26. To compute \( m_{i,j}^s(s) \) it is helpful to conceive of intermediate coefficients \( m_{i,j}^s(i,s) \) where

\[ m_{i,j}^s(i,s) = 1 \quad \text{there is a path} \]
\[ s \rightarrow \ldots \rightarrow j \]
\[ \epsilon \{1, \ldots, i\} \]

\((0 \leq i < s)\).

Obviously \( m_{i,j}^s(0,s) = m_{i,j}^s(s-1) \), and the values of \( m_{i,j}^s(i,s) \) for \( i = 1, \ldots, s-1 \) can be computed using the rule

\[ m_{i,j}^s(i,s) = m_{i,j}^s(i-1,s) \lor m_{i,j}^s(i-1,s) \cdot m_{i,j}^s(s-1) \]

fixed factor for the row.

Denoting the \( s \)-th row of \( M^*(s) \) by \( M_{i}^*(s) \) we obtain the iteration
for $i := 1$ to $s-1$ do
    if $m_{si}^\theta(s) \text{ then } M_{si}^\theta(s) := M_{si}^\theta(s) \land M_i^{\theta(s-1)}$
    fi
od

(where $M_i^{\theta(s-1)} \equiv M_i^{\theta(s)}$).

1.27. To compute $m_{sj}^*(s)$ it is similarly helpful to conceive of intermediate coefficients $m_{sj}^*(i,s)$, where

$$m_{sj}^*(i,s) = 1 \iff \text{there is a path}$$

$$s \rightarrow \ldots \rightarrow j$$

$$\in \{s+1, \ldots, i\}$$

($s \leq i \leq n$).

Obviously $m_{sj}^*(s,s) = m_{sj}^\theta(n)$, and the values of $m_{sj}^*(i,s)$ for $i = s+1, \ldots, n$ can be computed using the rule

$$m_{sj}^*(i,s) = m_{sj}^*(i-1,s) \lor m_{si}^*(i-1,s) \cdot m_{ij}^*(s-1)$$

fixed factor for the row.

We can therefore obtain $m_{sj}^*(s)$ from the iteration

for $i := s+1$ to $n$ do
    if $m_{si}^*(s) = 1$ then $M_{si}^*(s) := M_{si}^*(s) \lor M_i^{*(s-1)}$
    fi
od

(where $M_i^{*(s-1)} \equiv M_i^{*(s)} \equiv M_i^{*(0)}$).

1.28. The entire algorithm can now be formulated as

rewind;
scan for record 2;
for $s := 2$ to $n$ do
    $M_s^* := \text{get record;}
    \text{rewind;}
    \text{for } i := 1 \text{ to } s-1 \text{ do}$
if \( m_s^* \) then \( m_{i1}^* := \text{get record;} \)
\[
M^* := M^* \lor M_{11}^*
\]
else scan for next record
fi
od;
put \( M_s^* \)
od;
rewind;
for \( s := 1 \) to \( n-1 \) do
\( M_s^* := \text{get record} \)
for \( i := s+1 \) to \( n \) do
if \( m_{si}^* \) then \( m_{i1}^* := \text{get record;} \)
\[
M^* := M^* \lor M_{11}^*
\]
else scan for next record
fi
od;
rewind;
scan for record \( s \);
put \( M_s^* \)
od;

1.29. The number of records "paged in" by Warren's algorithm is bounded by

\[
2(n-1) + \#\text{off-diagonal ones in } M^*
\]
\[
\sim n^2 + n - \#\text{zero's in } M^*
\]

which shows that it behaves better than Marshall's the more sparse \( M^* \) is.

1.30. It follows that the design of good algorithms which use a pre-determined data-organisation can lead to non-trivial questions which the theory must consider just as well, and in the complexity-theory of data organisation one has to pay attention to the trade-offs between new data-structures and new algorithms.
2. FILE-MERGING

2.1. Sequential files are tape-like.

We shall assume that all records have equal size and that records are stored with small, detectable intermissions

\[ \cdots \quad | \quad | \quad \quad \cdots \]
\[ \uparrow \]
\[ \text{inter record gap} \]

2.2. There is a file-pointer, normally positioned at the beginning of a record

\[ \downarrow \quad | \quad | \quad \quad \cdots \]

2.3. We shall later consider essentially random access files which allow an arbitrary (but per application bounded) number of pointers.

2.4. Any general purpose programming language provides some or all of the following operations on a sequential file:

\textbf{rewind}

position the file-pointer back to the beginning of the tape

\[ \downarrow \quad | \quad | \quad \quad \cdots \]

\textbf{x := get record}

the contents of the record currently pointed at is read into record-variable \( x \) and the pointer is advanced to the beginning of the next record

\[ \cdots \quad \begin{array}{c} A \end{array} \quad \begin{array}{c} B \end{array} \quad \cdots \]
\[ \downarrow \]
\[ \Rightarrow \quad \cdots \quad \begin{array}{c} A \end{array} \quad \begin{array}{c} B \end{array} \quad \cdots \]
\[ \&\]
\[ x \equiv A \]
put x

the contents of record-variable x is stored into the
current record-position on the tape. The operation is
usually allowed only if the pointer is located at the
end of the file, otherwise we assume it simply overwrites
the previous record contents

\[
\begin{array}{c}
\rightarrow \\
\downarrow \\
\cdot \cdot \cdot \ | \ A \ | \ B \ | \cdot \cdot \cdot \\
& \cdot \cdot \cdot \ | \ C \ | \ B \ | \cdot \cdot \cdot \\
\rightarrow \\
\downarrow \\
\cdot \cdot \cdot \ | \ A \ | \ B \ | \cdot \cdot \cdot
d\end{array}
\]

reset record

positions the pointer at the beginning of the immediately
preceeding record (and void at the beginning of the tape)

\[
\begin{array}{c}
\rightarrow \\
\downarrow \\
\cdot \cdot \cdot \ | \ A \ | \ B \ | \cdot \cdot \cdot \\
\rightarrow \\
\downarrow \\
\cdot \cdot \cdot \ | \ A \ | \ B \ | \cdot \cdot \cdot
d\end{array}
\]

and occasionally one may want to use derived operations like

scan for b

\[
\begin{array}{c}
\text{repeat} \\
\text{x := get record} \\
\text{until eof \lor x.key = b;}
d\end{array}
\]

(or an equivalent interpretation of it).

2.5. On files with more than one pointer we shall in fact allow primitive
operations such as

\[
\begin{array}{c}
\text{swap \{P_1, P_2\}} \\
\text{P_1} \downarrow \\
\cdot \cdot \cdot \ | \ A \ | \cdot \cdot \cdot \\
\text{P_2} \downarrow \\
\cdot \cdot \cdot \ | \ B \ | \cdot \cdot \cdot \\
\rightarrow \\
\text{P_1} \downarrow \\
\cdot \cdot \cdot \ | \ B \ | \cdot \cdot \cdot \\
\text{P_2} \downarrow \\
\cdot \cdot \cdot \ | \ A \ | \cdot \cdot \cdot
d\end{array}
\]

and perhaps a boolean-valued instruction
ordered \( \langle p_1, p_2 \rangle \)
yielding true iff \( p_1 \) is before \( p_2 \) and \( p_1.\text{key} \leq p_2.\text{key} \).

EXAMPLE. A random access file

\[
\begin{array}{cc}
\text{P}_1 & \text{P}_2 \\
\uparrow & \uparrow \\
\vdots & \vdots
\end{array}
\]

with \( n \) records can be inverted in linear time by

\[
\text{while } p_2 > p_1 \text{ do swap } (p_1, p_2); \\
p_1 := p_1 + s; \\
p_2 := p_2 - s \\
\text{od;}
\]

(where \( s \) is the record size).

2.6. A (sequential) file is said to be ordered if and only if the keys of consecutive records form a nondecreasing chain.

2.7. The process of combining two ordered (sequential) files into a single ordered (sequential) file is called merging.

A merging procedure is said to be stable if and only if it leaves the relative order of records with the same key in the original files unchanged. (See Knuth [44].)

2.8. It is now reasonably well-understood how to merge two files in case some auxiliary tapes are available (Wirth [66]). It is less obvious how to proceed with minimal storage-requirements, and in particular how to do a linear time merge without the use of an unbounded number of (perhaps hidden) pointers or links.

This problem was first cited in Knuth ([44], p.388) and solved in 1974 by Horvath [40] and by Luis Trabb Pardo [58].

To acquire a better appreciation for the specific constraints of the problem we shall first consider a linear time two tape merge-procedure of Floyd & Smith [28].

2.9. Suppose we are given two ordered sequential files of records of a certain size \( s \)
N records, but perhaps only \( n \leq N \) with a distinct key.

\[ B: \begin{array}{cccccccc}
\vdots \\
\end{array} \]

M records, but perhaps only \( m \leq M \) with a distinct key.

We shall assume very little about \( s \), but only require that it provides enough bits for the distinct keys:

\[ s \geq \log(n+m). \]

2.10. The idea is to copy file \( A \) (at the end of \( B \)) and file \( B \) (at the end of \( A \)) and insert the proper sequence numbers in intermediate records first, showing in what order the file-members must be assembled.

An immediate problem shows up if we want to perform this in linear time: the sequence-numbers for file \( A \) (and similarly for file \( B \)) will occupy

\[ \sim N \cdot \log(N+M) \] in tape-length

and this may not at all be bounded in terms of the original file-length

\[ \sim N \cdot s \]

which could be as little as \( N \cdot \log(n+m) \).

2.11. Instead of complete sequence-numbers we shall therefore insert intermediate records once for each distinct key only (a common trick in data-reduction), but augment the information per record to

\[ \begin{array}{cccccccc}
\vdots \\
\vdots \\
\vdots \\
\end{array} \]

key \quad \log(N+M) \text{ bits for a position-number} \quad \log(N+M) \text{ bits counting how many records with the present key occur} \]
This will require an extra space of only

\[ \sim n \left( s + 2 \log(N + M) \right) \]

\[ \leq Ns + 2n \log(N + M) \]

\[ \leq Ns + \gamma(Ns + Ms) \log n \]

\[ \leq \gamma(Ns + Ms) \]

for tape A (and similarly for tape B), which is now bounded by a constant factor in the length of the original files.

2.12. The first step of the algorithm will copy A and insert occurrence-numbers for all distinct keys.

After a similar procedure is applied to copy tape B into B' at the end of tape A we have

\[ \begin{array}{c}
A: \square \square \ldots \ldots \square \downarrow \\
A \\
\text{containing } M + m \text{ records.}
\end{array} \]

\[ \begin{array}{c}
B: \square \ldots \ldots \ldots \ldots \square \downarrow \\
B \quad A' \\
\text{containing } N + n \text{ records}
\end{array} \]

and the time required so far is

\[ \sim Ns + Ms + Ms + 3Ns + Ns + \gamma(Ns + Ms) + Ns + \]

\[ \text{rewind \hspace{2cm} scan B \hspace{2cm} scan + copy \hspace{2cm} copy onto } A' \]

\[ A \]

\[ + \gamma(Ns + Ms) + Ms + 3Ms + Ms + \gamma(Ns + Ms) \]

\[ \text{rewind over \hspace{2cm} scan + copy \hspace{2cm} copy onto } B' \]

\[ A' \text{ and } B \]

\[ B \]

\[ \sim (6 + 3\gamma) Ns + (7 + 3\gamma)Ms. \]
2.13. Rewind over B'

A: ⬆

B: ⬆

and perform a "dummy" merge of A and B by looking at the intermediate,
accounting records only and inserting the appropriate position-number for
the first record of each distinct key in the proper field.

A: ⬆

B: ⬆

Rewind over A'

A: ⬆

B: ⬆

and copy the records of A (taken from A') at the end of tape A in the
proper relative final position (which we can determine from the information
in the accounting records which should not be copied over)

A: ⬆

B: ⬆

Rewind tape A to the beginning of B'

A: ⬆

B: ⬆

and copy all of B' onto tape B:
Finally, rewind over $B''$ and copy the records of $B$ (taken from $B''$) into their final positions on tape $A$ (again using the information from the accounting records)

$A$: $\square \ldots \square | \square \ldots \square | \square \ldots \square | \square \ldots \square \uparrow$

$B$: $\square \ldots \square | \square \ldots \square | \square \ldots \square | \square \ldots \square \uparrow$

$B'' \equiv B'$

The merged version of the original files appears as the last block on tape $A$.

2.14. We have now obtained the essential result of Floyd & Smith.

**Theorem.** The two-tape merge-procedure is stable and requires time linear in the length of the merged files.

**Proof.** Stability follows from the construction. The time required for all successive stages adds up to

$$\sim (B + 13y)(Ns + Ms).$$

2.15. Note that the algorithm requires no third tape, and we could have used the given space even more economically had we overwritten parts which were at some stage in the algorithm not needed anymore. We seem to be close to an answer of Knuth's problem, but there is apparently no way to eliminate the need for all accounting records without thereby causing a non-linear increase of time.

It doesn't seem to be easier for random access files either, and therefore the following result of Luis Trabb Pardo [58] is of interest.

**Theorem.** There is an algorithm to merge two "rigid" random access files in a stable way in linear time and no auxiliary storage using only a bounded number of pointers.
2.16. Trabb Pardo's algorithm contains a number of interesting contributions which we shall present in a simplified form to better demonstrate the ideas.

Assume that the files are stored as one chunk

\[
\begin{array}{cc}
\text{\textbackslash{}} & \text{\textbackslash{}} \\
\text{\textbackslash{}} & \text{\textbackslash{}} \\
\text{\textbackslash{}} & \text{\textbackslash{}} \\
\text{file A} & \text{file B} \\
\text{N records} & \text{M records}
\end{array}
\]

**EXAMPLE.**

\[
\begin{array}{cccccccccccc}
\text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} \\
1a & 1b & 2c & 2d & 2e & 1A & 2B & 2C & 3E & 3F & 3g & 4I & 5j & 5k
\end{array}
\]

2.17. A block of consecutive records is called an *internal buffer* if and only if it is fully ordered and contains only records with a distinct key.

**EXAMPLE.**

\[
\begin{array}{cccc}
\text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} \\
1A & 2C & 3G & 4I
\end{array}
\]

2.18. The first phase consists of extracting an internal buffer of size \(\sqrt{N+M}\) from the given files (which we always can if there are enough distinct keys).

The buffer-elements can be collected by scanning the files from left to right, picking a new larger record only if it is the last sample with the current key in the row (to ensure stability later).

**EXAMPLE**

\[
\begin{array}{cccccccccccc}
\text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} & \text{\textbackslash{}} \\
1a & 1b & 2c & 2d & 2e & 2B & 3E & 3F & 4H & 5J & 5K & 1A & 2C & 3G & 4I
\end{array}
\]

The buffer-elements are kept together in a "growing" block sliding steadily to the right, while a pointer advances to search for a next candidate.
Then:

reverse X

\[ \delta \quad \gamma \quad \beta \quad \alpha \quad \gamma \quad \delta \quad \ldots \]

and reverse the marked parts

\[ \gamma \quad \delta \quad \alpha \quad \beta \quad \gamma \quad \ldots \]

new B

and we can proceed until B is full size. B is then exchanged in the same way to the end of the file.

The time required for this phase is bounded by

\[ \sim N + M + \frac{(\sqrt{N + M})^2}{\sqrt{N + M}} \]

each stretch of elements can be involved in reversion once

but the elements of the buffer are involved in it perhaps as many times as its number of elements.

2.19. It could very well happen that file A is "small" with respect to

\[ s = \sqrt{N + M} \]

EXAMPLE.

\[ \delta \quad \gamma \quad \beta \quad \alpha \quad \gamma \quad \delta \quad \ldots \]

\[ \leq s \quad s \]

(If it is so for B we do the next procedure "backwards").
Exchange A with the first part of the buffer (in linear time using the reversion-trick):

EXAMPLE.

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 \\
\text{A} & \text{B} & \text{C} & \text{D} & \text{E} \\
1a & 2b & 3y & 1a & 1b & 2c & 3d & 3e & 1a & 1b & 2c & 4d
\end{array}
\]

and merge B and A into place in a stable manner by moving pointers 2 and 3 steadily rightwards, each time exchanging the smallest with the (buffer-) record under 1.

EXAMPLE.

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 \\
\text{A} & \text{B} & \text{C} & \text{D} & \text{E} \\
1a & 2b & 3y & 1a & 1b & 2c & 3d & 3e & 1a & 1b & 2c & 4d
\end{array}
\]

The buffer-records act as "dummies" which keep making place for records merged into position, and in the end they recollect in the buffer-zone.

Sort the buffer back into order, and complete the procedure (in linear time) by merging the buffer back into the file (how to do it will
follow later).

It is now clear why an internal buffer is essential: it is a group of
records which we can move around to provide a space where needed and which
we can bring back into stable order by just sorting!
(Another application of the buffer will follow next.)

2.20. Let us now assume that both A and B are "long". Think of A and B as
divided into blocks of size s.

EXAMPLE.

```
A   ----- S ---------------- S ---------------- S ---- B
```

Buffer

It is convenient to have a notation for the first and last key of a block
of known size:

```
+---------------------+-----
|                      | X   |
\                   /   |
|                  s  |
\                /    |
|               t_{a} |
\            /      |
|          \      |
\         +-----+---
```

\[
\text{first}(X) = t_1 \\
\text{last}(X) = t_2
\]

The values can be picked up using only two probes into the block.

2.21. If the size of A is no integer multiple of s

EXAMPLE.

```
A'       A''
/          /
+----------+---
A
```

then get rid off the left-over part A" by merge-exchanging it (as a block)
into B as far right as possible, i.e., after the last block X in B for which

\[
\text{last}(X) < \text{first}(A")
\]
EXAMPLE.

\[ \begin{array}{cccc}
  & & X & Y \\
  A' & B' & & Buffer \\
  & & \vdots \\
  & & \vdots \\
  & & last(X) < first(A") \\
  & & first(A") \leq last(Y). \\
\end{array} \]

X can be found by a simple scan, and using the reversion-trick the exchange follows in linear time.

The task is now reduced to

- merge A' and B'
- merge A" and B"

(where the last part follows as in 2.19). Note that the records will appear in final position if we do the merge in place.

2.22. Without loss of generality we can therefore assume that A has a neat block-arrangement.

EXAMPLE.

\[ \begin{array}{ccc}
  & A & B \\
  & & Buffer \\
\end{array} \]

Let the blocks of A and B be \( A_1, \ldots, A_k \) and \( B_1, \ldots, B_\ell \). Since \( s \sim \sqrt{NW} \), \( k + \ell \leq s \).

2.23. The idea is to first "merge" the A-blocks (as a block) in between the B-blocks as far to the right as possible while preserving stable order.
EXAMPLE.

\[ \begin{array}{c|c|c|c|c|c|c}
\vdots & B_j & A_{i1} & \cdots & A_{i2} & B_{j+1} & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\text{last}(B_j) & \text{first}(A_{i1}) & \text{first}(A_{i2}) & \leq & \text{last}(B_{j+1}) \\
\end{array} \]

In order to find the proper order of all blocks we shall first do a "dummy" pass and "assign position-numbers" (as in the Floyd & Smith algorithm).

The trick that makes it work in no extra space is to use the increasingly ordered buffer-records as "position-numbers" which we simply exchange into the first position of a block while temporarily storing the original first record of the block into the buffer-position corresponding to its position in the final order.

2.24. In this way we reduced "merging" blocks to a sort on the first element of each block. Use a simple straight insertion method.

Swapping each next "minimal" block into place (by a record-wise exchange) takes \( s \) steps, and searching for the next, \( i \)-th minimal block may take up to \( k + \ell - i \) probes. The total time is bounded by

\[ \sim (k + \ell)s + \sum_{i=1}^{k+\ell} (k + \ell - i) \sim s^2 \sim N + M \]

and is therefore linear.

Finally, in a left-to-right scan we exchange the original first records back into the blocks while the buffer is automatically restored at the same time.

2.25. It seems that we only need to re-hash neighbouring blocks somewhat to get records in the proper place (finally). We shall do so from right-to-left.

We didn't "mark" what the A- and B-blocks were (it doesn't matter as long as we merge correctly), but one can recover the partition by inspecting the order-relation between the first and last record of blocks while scanning to the left.
EXAMPLE.

\[ \text{... Z Y ... X W ... l ... l ... l ... l ...} \]
\[ \text{... ... ... \ldots buffer} \]
\[ \text{\text{last}(Z) < \text{first}(Y) \quad \text{first}(W) \leq \text{last}(X) \quad \text{merged}} \]
\[ \text{last}(W) > \text{first}(X) \]

"A-blocks" "B-blocks"

The situation can be recovered in time linear in the length of the stretch of elements involved. Note the L-block that may drag along. A graph helps to show the precise order-relation between the parts.

EXAMPLE.


EXAMPLE.

and use the reversion-trick to exchange $X$ (as a block) as far to the right into $W$ as possible.
EXAMPLE.

\[ W_1 \quad X \quad W_2 \]

yielding

EXAMPLE.

\[ Z \quad Y' \quad X \quad Y' \quad W_1 \quad W_2 \]

2.27. Now merge $X$ into $W_2 Y'$ in a stable manner using the buffer-trick from 2.19 (which we can because $X$ is just $s$ records long).

EXAMPLE.

\[ Z \quad Y' \quad W_1 \]

and we finished off "half" of the zone in time just linear in the number of elements involved.

*Don't resort the buffer again* each time we do such a phase.

With the same technique one can merge $Y'$ and $W_1$ in a stable manner and "finish" the entire zone.
EXAMPLE.

Proceed left and locate the next zone, thus continuing and completing the merge in linear time and a bounded number of pointers.

2.28. In the last phase of the algorithm one should resort the buffer (in time $\sim s^2$).

EXAMPLE.

merged file | buffer

and insert the buffer back into the file again, leaving each buffer-record at the far right of the stretch of file-records with the same key to ensure final stability.

With the reversion-trick one can steadily slide the buffer to the left and absorb it in linear time and no extra space.

EXAMPLE.

merged file

2.29. This completes the essential part of Trabb Pardo's minimal requirement stable merge procedure. (What should happen if no full size buffer can be extracted is left to the reader, but all necessary techniques to use have been presented.) Several steps were due to Kronrod and to Horvath.

2.30. Note that an "in-place" linear time merge procedure immediately yields
an "in-place" and minimal requirement $O(n \log n)$ sorting method by merging
the data into adjacent blocks of a size which is an increasing power of
two.

3. TABLES AND BALANCED TREES

3.1. The choice of a proper data-organisation for a task depends on how
the user wants to operate on the data.

In the design-philosophy of programming languages there is a tendency
to make data-structuring more and more automatic. Ultimately the user
should be able to specify the axioms and rules for the operations on the
data in some formal language, and the system should determine a consistent
(and preferably efficient) internal representation. This has been studied
theoretically (see Spitzen & Wegbreit [55]), and it is anticipated for
PASCAL in terms of graph-specification primitives (Gehani [33]).

3.2. Perhaps the simplest requirement on a (random access) file or table
is that one can perform

\begin{verbatim}
find (k)
    - locate the record with key k -
insert (k)
    - put the record with key k into the table -
\end{verbatim}

for all keys quickly.

Since the universe of key-values may be much larger than the set of
keys actually occurring in an application, an efficient and easy computable
key-to-address transformation should presumably "chop down" a key into an
index in a small hash-table where the record-address is (or will be)
listed.

Such hash-functions $h(k)$ are almost by definition many-to-one, and it
may happen that different keys are mapped onto the same hash-address.

The anomaly in insertion occurring when a hash-address is already
occupied is called collision.

3.3. Collisions can be resolved by entering records in an appropriate
overflow-area.
Another method (called open addressing) is to search the table for an opening "nearby" by systematically probing locations at a distance of
\[ \text{incr}(k) \]

further and further down the table. The hash-increment function is assumed to be positive, and is presumably chosen to lead to an opening fast.

The case when
\[ \text{incr}(k) = F(h(k)) \]

for all keys and some easy function \( F \) is sometimes called double-hashing.

3.4. Suppose we maintain a table
\[
\text{item}[0 : M-1]
\]
in this manner, where \( \text{item}[i] \) contains a key \( k \) such that \( h(k) = i \) if such a key was ever added and probably 0 otherwise. (It is strictly true only until collisions have occurred.)

One can search for \( k \) with

\[
\begin{align*}
\text{visits} & := 0; \\
\text{probe} & := h(k); \\
\text{while} \ \text{visits} < M \ \text{do} \\
\text{x} & := \text{item (probe)}; \\
\text{if} \ x = 0 \ \text{then} \ \text{not found \& exit for L fi;} \\
\text{if} \ x = k \ \text{then} \ \text{found \& exit for L fi;} \\
\text{probe} & := \text{probe} - \text{incr}(k) \ (\text{mod} \ M); \\
\text{visits} & := \text{visits} + 1 \\
\text{od}; \\
\text{table full};
\end{align*}
\]

L:

and insert \( k \) with essentially the same routine after replacing "not found" by

\[
\text{item (probe)} := k.
\]

In order that all locations of the table be probed in searching we demand
that incr(k) is relatively prime to M.

3.5. A hashing-method should avoid clustering of records (eliminating collisions from occurring too often), and is therefore usually dependent on an estimated probability distribution of the keys.


It does not mean that the table itself is entirely ordered, but that large parts of the chains are.

One can search for k with

\[
\begin{align*}
\text{visits} & := 0; \\
\text{probe} & := h(k); \\
\text{step} & := \text{incr}(k); \\
\text{while } \text{visits} < M \text{ do} & \\
\text{x} & := \text{item } (\text{probe}); \\
\text{if } x < k \text{ then not found & exit for L fi; } & \\
\text{if } x = k \text{ then found & exit for L fi; } & \\
\text{probe} & := \text{probe} - \text{step } (\text{mod } M); \\
\text{visits} & := \text{visits} + 1 \\
\text{od;} & \\
\text{table full;} & \\
\text{L:} & \\
\end{align*}
\]

The kind of arrangement does not specifically make successful searches faster, but it tends to detect unsuccessful searches much earlier.

3.7. If a set of keys were inserted in an empty table from largest to smallest (in that order) with the algorithm of 3.4, then an ordered hash-table results.

The insertion of randomly presented keys into an ordered hash-table is somewhat trickier.

EXAMPLE. Suppose we want to move 15 into position i now occupied by 10.

\[
\begin{array}{c}
\ldots \ldots \\
15 \\
\ldots \ldots \\
\end{array}
\]

\[
\begin{array}{c}
\ldots \ldots \\
10 \\
\ldots \ldots \\
\end{array}
\]

\[
\begin{array}{c}
\ldots \ldots \\
i \\
\end{array}
\]

\[
\begin{array}{c}
\ldots \ldots \\
\end{array}
\]

\[
\begin{array}{c}
\ldots \ldots \\
\end{array}
\]
Then we simply remove 10 and replace it by 15.

and try to re-insert 10. (Note that the replacement of an item by a larger item leaves all existing chains through that location consistent.)

Re-inserting 10 by back-chaining from h(10) on will bring us back to i again, so we better start the re-insertion procedure from position i on right-away. Note that for consistency the stepsize must become incr(10) now.

One can insert k in an ordered hash-table with

\[
\begin{align*}
\text{visits} & := 0; \\
\text{probe} & := h(k); \\
\text{step} & := \text{incr}(k); \\
\text{key} & := k; \\
\text{while} \text{ visits} < M \text{ do} \\
\text{x} & := \text{item (probe)}; \\
\text{if} \ x = 0 \text{ then} \text{item (probe)} & := \text{key} \ & \text{exit for L fi;} \\
\text{if} \ x < \text{key} \text{ then} \text{item (probe)} & := \text{key}; \\
\text{step} & := \text{incr}(x) \\
\text{fi;} \\
\text{if} \ x = \text{key} \text{ then} \text{found & exit for L fi;} \\
\text{key} & := x; \\
\text{probe} & := \text{probe} - \text{step (mod M)}; \\
\text{visits} & := \text{visits} + 1 \\
\text{od;} \\
\text{overflow;} \\
\text{L:}
\end{align*}
\]

Since the x-value keeps descending M is still a valid bound on the search-length.

Verify that insertion in an ordered hash-table indeed leaves another ordered hash-table.

3.7. The arrangement of N keys (N < M) in an ordered hash-table is unique.
3.8. Entirely different requirements of a data-organisation occur in ordered files or in structuring the directory of indexed files where one would like to perform operations

\[ \text{find} (k) \]
\[ \text{insert} (k) \]

but also

\[ \text{delete} (k) \]

- remove the item with key \( k \)

and perhaps

\[ \text{split file at} \; k \]

- separate the file in a part with items \( < k \)
and a part with items \( \geq k \)

as efficiently as possible.

3.9. Deleting elements in a hash-table could break up existing chains and make items further down the chain inaccessible unless a cumbersome reorganisation procedure is applied. Most implementations therefore avoid direct addressing schemes and use binary trees instead.

3.10. An interesting intermediate file-organisation suggested by Coffman & Eve [19] results if we choose a hash-function which maps keys onto bit-strings.

One can interpret bitstrings as coding paths down a tree and maintain the table as a hash-tree. To insert \( k \) we follow \( h(k) \) bit-after-bit and enter it into the first open node encountered, unless the key was found on the way.

**EXAMPLE.**

\[
\begin{array}{c}
0_1^1100... \quad \circ \quad h(k_1) = 1_1^11100...\\
\quad \circ \quad h(k_2) = 11_1^1100...\\
110_1^1100... \quad \circ \quad h(k_3) = 111_1^1100...
\end{array}
\]

Note the part of the hash-code used to enter \( k_1 \), \( k_2 \) and \( k_3 \) (0 \( \equiv \) "left", 1 \( \equiv \) "right").
Collisions are automatically avoided except when \( h(k) \) is "too short" to reach a distinguishing position (in which case an overflow-area must be entered).

3.11. Deleting \( k \) from its position \( p \) in the tree is easy when it is a leaf, but otherwise

\[ \begin{align*}
  &h(k) = a_A \\
  &h(k') = a_B
\end{align*} \]

one can delete an arbitrary leaf \( k' \) from its subtree and move it up into \( p \) to keep the hash-tree consistent, which is very easy also (provided \( k' \) is not too far away).

3.12. The average search-time in a hash-tree with \( N \) items will be \( \sim \log N \) or less (assuming a uniform distribution), but series of inserts and deletes can seriously impair it and cause long and inefficient paths in the tree.

3.13. Various kinds of binary search tree organisations have been proposed which remain balanced throughout.

In such trees the file-elements are normally arranged in left-to-right order at the leaves, and the internal nodes contain appropriate queries

\[ \text{if key} < k \text{ then left else right} \]

which enable one to retrieve stored items in a top-down search (see also 1.9). To maintain a balance in a tree with \( N \) items one must design insert-and delete-procedures which keep the worst case distance from the root to any leaf \( \sim \log N \).

Known balanced tree-models include AVL-trees (Aedelsson-Velskii & Landis [1]), various kinds of B-trees (Bayer [8]), 2-3 trees (Hopcroft, see [3]), and 1-2 trees (Maurer & Wood [45]).
We shall consider a simplified kind of 1-2 tree recently proposed by Ottmann & Six [49].

3.14. A binary (search) tree is called an HB-tree if and only if
(i) all leaves have equal depth,
(ii) each node with only one son has a brother with two sons.

EXAMPLE.

The root of an HB-tree must have two sons (unless it is a leaf). If a node has only one son, say p, then p must have two sons or else be a leaf. It follows that there can be

\[ \ldots \]
\[ \text{no} \]
\[ \ldots \]

\text{chains}

and HB-trees must always be quite "dense".

THEOREM. The depth \( d \) of any HB-tree with \( N \) leaves satisfies

\[ \lceil \log N \rceil \leq d < 1.44 \ldots \log(N + 1) - 0.32 \]
Proof. (Our argument differs from Ottman & Six [49].) In a binary tree of depth \( d \) we always have \( N \leq 2^d \implies d \geq \lceil \log N \rceil \).

Let \( T_e \) (\( e \geq 0 \)) be an HB-tree of depth \( e \) with the smallest possible number of leaves. It follows that (up to symmetry) \( T_e \) must be of the form

\[
\begin{array}{c}
O \\
| \\
O & O \\
| & |
\end{array}
\begin{array}{c}
T_{e-1} \\
| \\
T_{e-2}
\end{array}
\]

for \( e \geq 2 \), and \( T_e \) must have \( F_{e+2} \) leaves by induction (where \( F_i \) denotes the \( i \)-th Fibonacci-number). The depth \( d \) of a tree with \( N \) leaves satisfies therefore

\[
\frac{\varphi^{d+2}}{\sqrt{5}} - 1 < F_{d+2} \leq N.
\]

3.15. Observe that HB-trees are AVL-trees in which all paths from the root down the tree are made equally long by inserting extra nodes of degree one at the appropriate places. The effect, however, is that the operations for inserting or deleting items are much easier to explain.

3.16. One can insert \( k \) in the following manner.

Find the position in the tree where \( k \) must be inserted among the leaves.

\[
\begin{array}{c}
O \\
| \\
O & O \\
| & |
\end{array}
\begin{array}{c}
\vdots \\
p \\
| \\
O
\end{array}
\]

If \( p \) had only one son then it now has two and the structure remains an HB-tree. If \( p \) had two sons already then it now has three, and we must
enact a procedure \textit{split} \( p \).

It is now advantageous to pretend that all leaves have two sons (for reasons of consistency only).

The invariant maintained throughout the algorithm is that \( \text{split} \ p \) is called if and only if \( p \) has three sons at least two of which have degree 2.

\textbf{EXAMPLE.}

\begin{center}
\begin{tikzpicture}


\node (p) at (0,0) {p};
\node (q) at (-1.5,-1) {q};
\node (r) at (1.5,-1) {r};
\node (s) at (0,-2) {s};
\node (t) at (0,-3) {t};
\node (u) at (0,-4) {u};
\node (v) at (0,-5) {v};
\node (w) at (0,-6) {w};

\draw (p) -- (q);
\draw (p) -- (r);
\draw (q) -- (s);
\draw (r) -- (t);
\draw (s) -- (u);
\draw (t) -- (v);
\draw (u) -- (w);
\end{tikzpicture}
\end{center}

Thus at least one of the outermost sons of \( p \) must have degree 2 and we shall designate one as the \textit{active} son (on the \textit{active} side of \( p \)).

3.17. If \( p \) is the only son of \( q \) (say)

\textbf{EXAMPLE}

\begin{center}
\begin{tikzpicture}

\node (q) at (0,0) {q};
\node (p) at (1.5,-1) {p};
\node (a) at (0,-2) {a};
\node (b) at (0,-3) {b};
\node (c) at (0,-4) {c};

\draw (q) -- (p);
\draw (a) -- (p);
\draw (b) -- (p);
\draw (c) -- (p);
\end{tikzpicture}
\end{center}

then create a new brother \( p' \) on the \textit{active} side of \( p \), connect the \textit{active} son of \( p \) to \( p' \) and finish.

\textbf{EXAMPLE.}

\begin{center}
\begin{tikzpicture}

\node (p) at (0,0) {p'};
\node (q) at (1.5,-1) {q};
\node (a) at (-1.5,-2) {a};
\node (b) at (0,-2) {b};
\node (c) at (1.5,-2) {c};

\draw (p) -- (a);
\draw (p) -- (b);
\draw (p) -- (c);
\draw (q) -- (b);
\end{tikzpicture}
\end{center}
If \( p \) has a brother \( q \) and father \( r \)

**EXAMPLE**

```
        o
       /|
      /  \
     q   p
      /|
     A  B  C
    1 2 2
```

then make the son of \( p \) on the \( q \)-side a son of \( q \) if \( q \) has degree one (which makes an HB-tree even if \( A \) had degree one) and finish

**EXAMPLE**

```
        o
       /|
      /  \
     q   p
      /|
     D  A  B  C
    2 1 2  2
```

(as originally only son of \( q \) \( D \) must have degree 2, see 3.14)

and otherwise (when \( q \) has degree 2) create a new brother \( p' \) directly on the active side of \( p \), make the active son of \( p \) the son of \( p' \), and continue with **split** \( r \) (observing that the invariant is preserved!).

**EXAMPLE**

```
        o
       /|
      /  \
     q  p'  p
      /|
     A  B  C
    2 2 1
```

If \( p \) has no father we must have reached the root of the tree and must create a new root one level higher.
3.18. One can delete $k$ in the following manner.

Find the position among the leaves where $k$ is located.

If $p$ is the only son of $q$

**EXAMPLE**

then delete $p$

**EXAMPLE**

(note that $q$ must have a brother of degree 2)

and continue to delete $q$.

After this initial phase the invariant maintained by the algorithm is that

*delete p* is called if and only if $p$ must be deleted and it has a brother of degree 2.

3.19. (Our deletion-procedure differs slightly from Ottman & Sifn.)

In a *delete p* the order between $p$ and his brother is irrelevant, so we shall always assume the brother to be on the "convenient" side.
If the father of $p$ is the only son of $q$

**EXAMPLE**

\[
\begin{array}{c}
q' \\
\quad \quad q \\
\quad \quad \quad r \\
\quad \quad \quad \quad p'' \\
\quad \quad \quad \quad \quad p \\
\quad \quad \quad \quad \quad \quad \quad \quad p' \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 2 \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 2
\end{array}
\]

(this is the implied global structure)

then make $p'$ a son of $r$ in case $r$ has degree one, omit $p$ and $p''$, and continue with delete $q$ (observing that the invariant is preserved).

**EXAMPLE.**

\[
\begin{array}{c}
q' \\
\quad \quad q \\
\quad \quad \quad r \\
\quad \quad \quad \quad p' \\
\quad \quad \quad \quad \quad \quad \quad \quad 2 \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 2
\end{array}
\]

and otherwise (when $r$ has degree 2) we make $p'$ a son of $r$ also while dropping $p$ and $p''$ but observe that now the invariant is satisfied for split $r$. 
EXAMPLE

If split r (et cetera) runs out before q' is reached we continue with delete q (observing that the invariant will hold), otherwise we omit q and finish.

If the father of p has a brother

EXAMPLE

then make p' a son of r, omit p, and continue with delete q in case r had degree one, and otherwise (when r has degree 2) construct a new brother r' between r and q, make p' the son of r' and omit p and q, and finish.

EXAMPLE.
If the father of \( p \) is no son of any node we must have reached the top of the tree, and we can simply omit \( p \) and make his brother the new root.

3.20. The procedures show

**Theorem.** One can execute find, insert, and delete-instructions on an arbitrary HB-tree with \( N \) leaves in \( \sim \log N \) steps per instruction.

3.21. Given two HB-trees \( T_1 \) and \( T_2 \) (in that order) one can merge the corresponding sets in the following manner.

Assume that \( T_1 \) is smaller than \( T_2 \) (and modify the algorithm accordingly otherwise). Find the element on the left-side of \( T_2 \) equally high as root(\( T_1 \)).

**Example**

![Diagram of HB-trees branching](image)

Make \( r \) a son of \( p \). If \( p \) had one son before it now has two and we can finish, otherwise the invariant for split \( p \) is enacted and an HB-tree results only after some additional moves.

Merging two trees is thus also bounded by \( \sim \log N \) (or, more precisely, by depth(\( T_2 \)) - depth(\( T_1 \))).

HB-trees can now also support instructions

split at \( k \)

in \( \sim \log N \) steps as follows. Cut the tree along the path down to \( k \) in two parts, each part consisting of correctly ordered but loose hanging HB-trees. Merge the smaller trees into the larger continuously.

3.21. The idea of HB-trees can be generalised following similar steps to \( k \)-ary search trees (Maurer, Ottmann & Six [45]).

3.22. All previous tree-structures could be used also for implementing
disjoint unordered files or sets.

The simplest operations data-structures for sets should support efficiently include

union (A,B,C)
   - merge the (disjoint) sets named
     A and B into a new set named C -

find (k)
   - determine the name of the set to
     which k currently belongs -

A variety of new, helpful programming techniques for set-manipulation programs developed during and after a systematic study of Aho, Hopcroft, and Ullman [3] (chapter 4, see also Knuth [44]).

Since we shall hardly ever need a search-facility in such applications we shall abandon the strict binary trees in favor of more general and perhaps more compact trees in which for each node only a father-link (if any) is listed.

EXAMPLE.

```
..... o o o o o c ..... 
 ..... 
 ..... o o o o ..... 
```

3.23. One can search the root of such a tree from any interior node on using

```pascal
visitednode := startnode;
while (p := visitednode.father; p ≠ nil) do
  visitednode := p
od;
root := visitednode;
```

but it is impossible to start at the root and search the tree in opposite direction.

It is therefore convenient to keep a fixed directory where set-elements are and store the set-name at the root of the tree (which also makes updating set-names easy).

We shall maintain a directory which yields for any set-name the
address of the corresponding root.

3.24. Union-find programs obviously benefit from balancing the trees. Let the weight of a node be equal to the number of leaves in its subtree.

Build the trees from items of

\[
\text{type } \text{node} = \text{record} \\
\quad \text{name: a character-string;} \\
\quad \text{weight: an integer;} \\
\quad \text{father: } \uparrow \text{node;} \\
\text{end;}
\]

The idea of balancing in performing a "union" is to never merge a tree of larger weight into a tree of smaller weight.

EXAMPLE. Note the difference in merging.

\[
\begin{align*}
\text{A} & \ 2 \\
\text{B} & \ 4 \\
\end{align*}
\]

and

\[
\begin{align*}
\text{A} & \ 2 \\
\text{B} & \ 4 \\
\end{align*}
\]

into

\[
\begin{align*}
\text{C} & \ 6 \\
\text{B} & \ 4 \ 117 \\
\text{A} & \ 2 \ 117 \\
\end{align*}
\]

(Note also that in merging some of the original set-names are automatically "lost").
The objective of this rule is to keep the path-length in the merged trees down, thus permitting efficient find's at any moment.

3.25. A union \((A,B,C)\) instruction can be executed by

\[
\begin{align*}
\text{set } a &:= \text{root } A; \\
\text{set } b &:= \text{root } B; \\
\text{if } \text{set } a.\text{weight} > \text{set } b.\text{weight} \text{ then } p &:= \text{set } a; \\
&\quad \text{set } a := \text{set } b; \\
&\quad \text{set } b := p \\
\text{fi;}
\end{align*}
\]

\[
\text{set } b.\text{name} := C; \\
\text{set } b.\text{weight} := \text{set } b.\text{weight} + \text{set } a.\text{weight}; \\
\text{set } a.\text{father} := \text{set } b;
\]

followed by an update of the set-name directory.

3.26. Given \(n\) one-element sets initially, it follows that

**THEOREM.** The given organisation can support \(m\) (less than \(n\)) union-find instructions in \(\sim \log m\) steps per instruction.

**Proof.** Unions take \(\sim\) constant time. The time required by find's is bounded by \(\sim\) the maximal path-length in any tree built during the algorithm.

We observe that any tree of depth \(d\) in the forest under construction must have weight \(\geq 2^d\). This is true initially (for \(d = 0\)), and whenever two trees are merged into a tree of depth \(d\) later

![Diagram of trees](image)

either \(T_2\) already had depth \(d\) and weight \(\geq 2^d\) by induction or \(T_1\) has depth \(d - 1\) and

\[
\text{weight}(T_2) \geq \text{weight}(T_1) \geq 2^{d-1}
\]

because of balancing by induction
which means that the resulting tree has

$$\text{weight} \geq 2^{d-1} + 2^{d-1} = 2^d$$

also.

The maximal path-length \(d\) therefore satisfies \(2^d \leq m \implies d \sim \log m\). \(\square\)

3.27. In merging, one set will always lose its original name (see 3.24) and because the result of balancing is rather unpredictable we may not know in advance which one. Note also that under circumstances the same root may be renamed more than once.

It is therefore impossible in the given implementation to perform

\[
lca(k_1, k_2)
\]

- find the node (or name) of the least recent
  ("smallest") set to which \(k_1\) and \(k_2\) were made
  to belong together, and undefined if no such set exists -

on-line.

Aho, Hopcroft, and Ullman [4] proved that one may execute \(\sim m\) union-find-lca instructions online in an average of \(\sim \log m\) steps per instruction.

Using a different data-structure for forests we have been able to reduce this to a less than logarithmic bound (Van Leeuwen [60]).

THEOREM. One may execute \(\sim m\) union-find-lca instructions in \(\sim \log \log m\) steps per instruction on the average.

(We note that Aho, Hopcroft, and Ullman [4] considered a slightly more general type of merge-instruction to occur in conjunction with lca's than we permitted here.)

4. PATH COMPRESSION

4.1. The logical development of set-manipulation programs often benefits from representing sets as trees, even though an actual implementation may
use different structures like arrays. It relates to a similar aspect of information management systems where the data-model of the user may be very different from the organisation of the physical database.

Trees were already used as data-structures in the early fifties when "sorting and searching" emerged as an area of intensive study.

In 1964 the idea was used by Galler & Fischer [32] in improving earlier work on Arden, Galler, and Graham [6] on MAD and aspects of FORTRAN-implementation. They considered how one can efficiently assign addresses to EQUIVALENCEd variables in FORTRAN-programs at compile-time, and used trees which actually carried names of variables in all nodes (including array-identifiers).

**EXAMPLE.** EQUIVALENCE (U[3],M[10],N) is represented, for example, among

```
......       .....       
     X       U       
      O       O       
     -2       -7       +3
      0       0       0
     Y       Z       
      0       0       V
     J       K       M
      0       0       N
```

4.2. Trees have been successfully applied in programs for manipulating disjoint sets, which occur for instance when one is building classes of an equivalence-relation or in a variety of graph-algorithms.

4.3. There are only a few programming tricks for making ordinary set-manipulation programs more efficient, but some of these techniques are very powerful and fundamental.

Following an approach of Aho, Hopcroft, and Ullman [3] we shall study union-find programs for more abstractly presenting an analysis of balancing and path-compression.

4.4. A collection of disjoint trees (representing disjoint sets) is called a forest. We shall assume that set-elements are attached to leaves only.
EXAMPLE.

The weight gives the number of set-elements stored in a tree. Directories tell where elements are stored, and where for each set-name the corresponding root is, but to execute a "find" one must explicitly follow father-links as in 3.23.

4.5. Assume (for simplicity) that we start with \( n \) singleton sets

and execute \( m \) finds and (at most) \( n-1 \) unions.

Unions take constant time each, but the cost for finds is to be the number of edges traversed in each find-path.

Let \( P \) be the set of edges which will ever appear in the finds. (Note that these edges may not all be present in the current forest at some moment.)

4.6. A direct implementation could cost time \( \sim mn \) in worst case.

Balancing helps, and a rule as in 3.24 (the weighted union rule) for resolving unions reduces execution-time to \( \sim m \log n \) in worst case.

Tritter (according to Knuth [43]) and also McIlroy & Morris (according to Hopcroft) apparently first realized how one can reduce the cost of (future) finds. In executing a find\((k)\) one should save all nodes encountered on the find-path in a stack and attach all directly to the root once it is located (see fig.7). It is sometimes called the collapsing rule, but nowadays better known as path-compression.

4.7. It is clear that balancing and path-compression (or even path-compression alone) should eventually lead to very low trees and thus reduce the
cost for later finds substantially, but the precise analysis requires a new and important accounting-technique.

Paterson proved in 1972 that when only path-compression (and no balancing) is used and \( m \sim n \), the execution-time is still bounded by \( \sim n \log n \). Hopcroft & Ullman [39] found an elegant level-crossing argument soon after and proved that balancing and path-compression together reduce the execution-time to \( \sim m \log^* n \), which is almost linear. The ideas were carefully analysed further by Tarjan [56] who finally succeeded showing that

The cost for \( m \) finds(\( m \geq n \)) and \( n-1 \) unions with balancing and path-compression is practically linear in \( m \).
Our presentation of the key-ideas in the analysis will differ somewhat from Tarjan's.

4.8. Consider arbitrary programs of m finds and n - 1 unions which use some strategy for each instruction first.

Nodes can gradually climb to higher and higher levels.

A "find" (even when path-compression is used) does not change the current location of a root, but it can make that some interior nodes are pulled up to a shorter distance of the root.

It is therefore important to know where the roots will be during the algorithm, which is entirely determined by the unions!

4.9. Pretend that we do all unions first (obeying whatever strategy there was), and see where all nodes will eventually finish.

Leave all nodes where they are (hanging up in the air at their respective levels), but remove all connecting edges and start the program again.

Unions are now easy (just insert the connecting edge again), and we can concentrate entirely on what finds do in the model.

4.10. The level-number of a node v is called its rank r(v).

Let the maximum rank be M.

EXAMPLE.

```
M | - - - - O - - - - - -
   |   o   o
   | ..... o o ..... 
   |   o   o
   | - - - - O  O - - - -
```

4.11. In a find with path-compression interior nodes get connected to nodes higher up, and their (new) father always remains of higher rank.

It means that at any moment the ranks along an upward path must form an increasing sequence.

4.12. We can estimate #f by counting how many times each node v ever occurring on a find-path can stretch its father-link to a higher node.
EXAMPLE.

We shall do so by counting how many times eventually occurring edges

can cross distinguished levels of a grid.

4.13. The crucial step in Tarjan's analysis is to work with not just one
grid, but to consider \( k + 1 \) grids

\[
A(i,*) \quad (0 \leq i \leq k)
\]

instead, where \( A(i-1,*) \) is a refinement of \( A(i,*) \). Thus we require the
existence of a monotone function \( B(*,*) \) such that

\[
A(i,j) = A(i-1,B(i,j-1)) \quad (i \geq 1),
\]

where we define \( B(i,-1) = 0 \) for consistency.

Let \( a_B(i,x) = \min\{j \mid A(i,j) > x\} \). Thus \( a_B(i,M) \) gives how many levels
in each grid there are.
EXAMPLE.

\[
A(i-1,B(i,j)) \quad \ldots \quad A(i,j+1) \\
\vdots \\
A(i-1,B(i,j-1)+3) \quad \ldots \\
A(i-1,B(i,j-1)+2) \quad \ldots \\
A(i-1,B(i,j-1)+1) \quad \ldots \\
A(i-1,B(i,j-1)) \quad \ldots \quad A(i,j)
\]

4.14. The finest grid is \(A(0,*\)) defined by

\[A(0,x) = cx \quad (c \geq 2).\]

Note that any point ever occurring on a find-path must cross this grid after being pulled up \(c-1\) times.

The next grids all begin with

\[A(i,0) = 0.\]

Let the number of nodes \(v\) with \(A(i,j) \leq r(v) < A(i,j+1)\) be \(s(i,j)\), and assume that

\[s(i,j) \leq \frac{f(n)}{g(i,j)} \quad (i \geq 1).\]

4.15. For each node \(v\) we should estimate how many edges

\[
\begin{array}{c}
Q \\
\downarrow \\
v
\end{array}
\]

there can be on later find-paths.

EXAMPLE.
Follow \( v \) as it "bumps" into lower grids while rising first until it crosses a next grid \( A(i-1,*) \) for the first time.

**EXAMPLE.**

\[
\begin{array}{c}
A(i,j+1) \\
\vdots \\
A(i,j) \\
\end{array}
\]

Watch \( v \) to see when it gets attached to a node across the next level of \( A(i,*) \), *in the meantime* "charging the \( i \)-th grid" one edge whenever \( v \) gets attached to a node across yet another level of \( A(i-1,*) \)

- thus the \( i \)-th grid can get charged at most
  \( B(i,j) - B(i,j-1) \) times —

and "charging the find" whenever \( v \) is in between but on its way

**EXAMPLE**

\[
\begin{array}{c}
A(i,j+1) \\
\vdots \\
A(i,j) \\
\end{array}
\]

It means that we partition \( F \) into sets

\[
F_0 = \{(v,w) \in F \mid \exists_j A(0,j) \leq r(v) < r(w) < A(0,j+1)\}
\]

\[
F_i = \{(v,w) \in F \mid \exists_j A(i,j) \leq r(v) < r(w) < A(i,j+1) \}
\]

and \( \exists_j^j r(v) < A(i-1,j) \leq r(w) \)

(for \( 1 \leq i \leq k \))
\[ F_{k+1} = \{(v, w) \in F \mid \exists \ell \ r(v) < A(k, \ell) \leq r(w)\} \]

and divide the charges (i.e. the edge-count) between

\[ F_i - L_i \quad \text{and} \quad L_i \quad (0 \leq i \leq k+1), \]

where

\[ L_i = \{(v, w) \in F_i \mid \text{on its find-path (v,w) is the last edge in } F_i\} \]

(thus all edges in \( F_i - L_i \) are not last and must contribute a new crossing across the level crossed by the last).

4.16. Clearly \( \#L_i \leq m \), because each find can get charged at most one last edge.

4.17. An edge \( Q_0 \_ v \) can occur in \( F_0 - L_0 \) at most \( c-1 \) times, because after that many times \( v \) must be attached to a node across the first level of \( A(0,*) \) (and the edges are charged to \( F_1 \) or \( F_2 \) etc.). Thus

\[ \#(F_0 - L_0) \leq (c-1)n. \]

4.18. The points \( v \) on edges \( Q \_ v \) with \( A(i, j) \leq r(v) < r(w) < A(i, j+1) \) in \( F_i - L_i \) (see 4.15) can contribute at most \( B(i, j) - B(i, j-1) \) each, and thus for each \( 1 \leq i \leq k \)

\[ \#(F_i - L_i) \leq \sum_{j=0}^{*} (B(i, j) - B(i, j-1))s(i, j) \leq \sum_{j=0}^{*} \frac{B(i, j) - B(i, j-1)}{g(i, j)} f(n) \leq h(i) \cdot f(n) \]

for some function \( h \).

4.19. For each node \( v \) there can be at most as many edges \( D \_ v \) in \( F_{k+1} - L_{k+1} \) as there are lines to cross in the \( k \)-th grid. Thus

\[ \#(F_{k+1} - L_{k+1}) \leq a_k(k, M) \cdot n. \]

4.20. Adding up all estimates the total charge for \( m \) finds and \( n-1 \) unions
is bounded by

\[(k+2)m + cn + \sum_{1}^{k} h(i) \cdot f(n) + a_B(k,M \cdot n) ,\]

and all we have to do is choose the grids such that the expression is as small as possible.

4.21. It is important to note that the bound was derived independent of the strategy for unions. We can immediately obtain Paterson's result as follows.

THEOREM. The cost for \( n \) finds and \( n-1 \) unions using only path-compression (and no balancing) is \( \sim n \log n \).

Proof. Let \( B(i,j) = c \cdot (j+1) \) such that by induction we have \( A(i,j) = c \cdot j \).

Since no balancing is used, the "best" estimate for \( s(i,j) \) is just some \( f(n) \frac{g(l,j)}{g(l,j)} \) with \( f(n) = n \) and \( \sum_{j=0}^{k} \frac{1}{g(l,j)} \sim 1 \).

It follows that \( h(i) = c \) for \( 1 \leq i \leq k \), and also \( a_B(k,M) \sim M/c^k \).

By 4.20 the total cost for \( n \) finds and \( n-1 \) unions is bounded by \( \sim ck n + Mn/c^k \). Let \( c = 2 \) and choose \( k \sim \log M \). It follows that the cost is bounded by \( \sim n \log M \), which could be as worse as \( \sim n \log n \) for an unbalanced tree. \( \square \)

Fischer [27] proved in fact that \( n \log n \) is tight to within a constant factor.

4.22. When a proper form of balancing is used we get \( M \sim \log n \), and it immediately follows from 4.21 that the cost for \( n \) finds and \( n-1 \) unions is at least reduced to \( \sim n \log \log n \).

Tarjan showed that one can get much better bounds from 4.20 in this case.

Remember from 3.26 that when the weighted union rule is used (or indeed any other rule for proper balancing) a node of rank \( r \) must lead a tree with at least \( 2^r \) leaves. The number of rank \( r \) nodes is therefore bounded by \( n/2^r \), and it follows that

\[ s(i,j) \leq A(i,j+1) \frac{n}{2^x} \leq \frac{2n}{2^{A(i,j)}} \left( = \frac{f(n)}{g(l,j)} \right). \]
Thus we must choose B such that we can somehow reasonably bound
\[
\sum_{j=0}^{\infty} \frac{B(i,j) - B(i,j-1)}{A(i,j)}
\]

4.23. Big trick: one can choose \( B(i,j) = A(i,j) \) for \( j \geq 1 \) and \( B(i,0) = 1. \) (We could have chosen even larger B's but this will do to get extremely sharp results already).

4.24. The grids we now have are defined by

\[
\begin{align*}
A(0,x) &= cx \\
A(i,0) &= 0 \\
A(i,1) &= c \\
A(i,j+1) &= A(i-1,A(i,j)) \quad (i,j \geq 1)
\end{align*}
\]

which yields an analog of Ackerman's function with an equally super-fast, not primitive recursively bounded growth-behaviour.

Observe that

\[
A(1,x) = c^x
\]

\[
A(2,x) = c \underbrace{c \cdots c}_{|x|}, \text{ etc.}
\]

4.25. Since we can now choose \( h(i) = 2 \) the total cost for \( m \) finds and \( n-1 \) unions is bounded by

\[
\sim k(m+n) + cn + a_A(k,\log n)n
\]

where one should notice that already for \( k = 2 \) the function \( a_A(k,\log n) \) is very slowly growing.

For \( k = 2 \) and \( c = 2 \) we get Hopcroft & Ullman's result that \( m \) finds and \( n-1 \) unions take at most time \( \sim m + n \log^* n. \)

Define a "functional inverse" \( a(m,n) \) of the A-function in 4.24 by

\[
a(m,n) = \min\{j \mid A(j,\frac{m}{n}) > \log n\}.
\]
We obtain a form of Tarjan's result

**THEOREM.** The cost for \( m \) finds and \( n - 1 \) unions using balancing and path-compression is bounded by \((m+n)\alpha(m,n)\).

**Proof.** Choose \( k \sim \alpha(m,n) \), and observe that \( s_A(k, \log n) \leq \frac{m}{n} \alpha(m,n) \).

\( \rightarrow \) bound \( \sim (m+n)\alpha(m,n) \). \( \square \)

Assuming that in practical situations usually \( m \geq n \) it follows that the time-bound is

\( \sim m \alpha(m,n) \)

and that is practically linear. Tarjan [56] proved that the algorithm is not completely linear.

4.26. There are numerous applications of Tarjan's theorem in data-organisation, showing that all kinds of algorithms can be made practically linear with balancing and path-compression.

Hopcroft & Karp [38] proved in 1971 that the equivalence of two finite automata with at most \( n \) states can be determined in \( \sim n \log n \) steps, but one may now show

**THEOREM.** The equivalence of \( n \)-state finite automata can be determined in \( \sim n \alpha(n,n) \) steps.

**Proof.** Let \( M_1 \oplus M_2 = \langle S_1 \oplus S_2, I, \delta, \{s_1, s_2\}, F \rangle \) and consider the equivalence-relation \( E \) determined by

\[ p \in E q \iff \forall x (\delta(p, x) \in F \iff \delta(q, x) \in F). \]

\( M_1 \) and \( M_2 \) are equivalent if and only if \( s_1 \in E s_2 \). It means that the smallest relation \( R \) satisfying

(i) \( s_1 \in R s_2 \)
(ii) \( p \in R q \iff \forall c \in \Sigma \delta(p, c) R \delta(q, c) \)

must be contained in \( E \).

Thus \( M_1 \) and \( M_2 \) are equivalent if and only if we can build \( R \)-classes as if it were equivalence-classes without ever getting a final and a non-final state together in the same class.
Prepare classes with names $\epsilon \{1, \ldots, 2n\}$ and attributes $\epsilon \{\text{FINAL, NON-FINAL}\}$ with the following algorithm

$$\begin{array}{l}
\text{classes} \leftarrow \text{a family of 2n disjoint sets of one element each:}
\text{set } i \leftarrow \text{state i attr.}
\hline
\text{queue} \leftarrow (s_1, s_2);
\text{while queue} \neq \emptyset \text{ do}
\text{detach (p,q) from queue;}
\text{name}_1 := \text{find(p)};
\text{name}_2 := \text{find(q)};
\text{if name}_1 \neq \text{name}_2 \text{ then}
\quad \text{if name}_1 \cdot \text{attrib} \neq \text{name}_2 \cdot \text{attrib then}
\quad \quad M_1 \text{ and } M_2 \text{ are not equivalent & signal fi}
\quad \quad \text{else union(name}_1, \text{name}_2, \text{name}_1);\]
\quad \quad \text{attach } (\delta(p,\sigma), \delta(q,\sigma)) \text{ to queue for each } \sigma \in \Sigma
\quad \text{fi}
\od;
\end{array}$$

When $M_1$ and $M_2$ are equivalent the algorithm will produce exactly all equivalence-classes of admissible states, but if they aren't it will find the clash.

There can be at most $\sim n$ unions, and thus at most $\sim n \cdot \#\Sigma$ pairs of states will ever get onto the queue. We need to execute at most $n$ unions and $O(n)$ finds, assuming that $\#\Sigma$ is fixed. [1]

4.27. An interesting and instructive application of path-compression was found by Aho, Hopcroft, and Ullman [4] in studying programs which maintain a forest with the following instructions

$$\text{merge}(u,v)$$

- to be executed only if $u$ is a root and $v$ is not in the tree with root $u$. The instruction makes $u$ a son of $v$. -
depth(v)
- determine the distance of v to the root of the
tree to which it currently belongs -

EXAMPLE. A merge(u, v) applied to

will yield

where as a result depth(u) := depth(v) + 1.

Even balancing may not seem straightforward here as it might destroy
the ability to determine depth correctly if it was applied directly to the
real forest.

In addition to the real forest we shall therefore maintain an auxiliary shadow forest which does support depth-instructions quickly, and we
shall give a proof of

THEOREM. One can execute n-1 merge- and m depth-instructions (m ≥ n) in ~ m α(m,n) steps.

4.28. A depth-instruction looks like a find, but if you want to move an
internal node v

EXAMPLE

and make it a direct son of the root (as we would in path-compression),
then one must attach a relative displacement ("disp") to it which gets
added to the depth-count when we pass v from x on our (shortened) way
up to the root.
4.29. Suppose we start with a forest of \( n \) singletons, and maintain the structure with nodes of type

\[
\text{type node} = \text{record} \quad \begin{align*}
    w & : \text{number of descendant leaves;} \\
    \text{disp} & : \text{current displacement;} \\
    \text{father} & : \uparrow \text{node;}
\end{align*} \\
\text{end};
\]

In executing a "\( \text{depth}(v) \)" we can do path-compression as in a find, provided we adjust the displacement of all nodes pulled up to the root appropriately.

EXAMPLE. A \( \text{depth}(v_3) \) on \( \begin{array}{c}
    v_0 \\
    \downarrow \\
    v_1 \\
    \downarrow \\
    \ldots \\
    \downarrow \\
    v_2 \\
    \downarrow \\
    v_3\end{array} \)

should result in

\[
\begin{array}{c}
    v_0 \\
    \downarrow \\
    v_1 \\
    \downarrow \\
    \ldots \\
    \downarrow \\
    v_2 \\
    \downarrow \\
    v_3 \quad \text{disp}'(v_2) := \text{disp}(v_2) + \text{disp}(v_1) + 1 \quad \text{disp}'(v_3) = \text{disp}(v_3) + \text{disp}'(v_2) + 1 \\
\end{array}
\]

with \( \text{depth}(v_3) = \text{disp}'(v_3) + 1 + \text{disp}(v_0) \).
The following iterative algorithm will work

\[
\begin{align*}
\text{stack } S & \leftarrow \emptyset; \\
p & := +v; \\
\text{if } p = \text{root then } depth & := p.\text{disp} & \text{HALT fi};& \text{while } p \neq \text{root do} \\
\text{push } p; & \text{do } & \text{od}; \\
root & := p; & \text{while stack } \neq \emptyset \text{ do } \\
p & := \text{pop } S; & q & := \text{pop } S; \\
q.\text{father} & := \text{root}; & q.\text{disp} & := q.\text{disp} + p.\text{disp} + 1; \\
p & := q; & \text{od}; \\
\text{depth} & := p.\text{disp} + 1 + \text{root.disp} \\
\end{align*}
\]

(Note that the w.field of internal nodes is not up-dated and becomes unreliable, but fortunately we only need its integrity for the root.)

4.30. To execute a merge(u,v) we first check in the shadow forest that the instruction is permissible by executing

\[
\begin{align*}
\text{depth}(u) & \\
\text{depth}(v) & \\
\end{align*}
\]

(with path-compression).

We should find that depth(u) = 0, and also get as a side-result the roots x and y of the trees where u and v currently belong. Check x \neq y.

EXAMPLE.

\[
\begin{align*}
\text{(Diagram showing trees with nodes labeled x, y, u, v)}
\end{align*}
\]
(Note that the roots in the shadow-forest could differ from the roots in the real forest.)

With some care the weighted union rule can be applied to the merge as reflected in the shadow forest.

EXAMPLE. The merge(u,v) should come up with a correctly displaced structure equivalent to

![Diagram]

note that u is the real root of this tree.

4.31. If \( y\) weight \( \leq x\) weight then we may just as well make \( y\) a direct son of \( x\), provided we adjust the displacement of \( y\) to correct for all nodes in that tree.

EXAMPLE.

![Diagram]

where \( \text{disp}'(y) = \text{disp}(y) + 1 + \text{disp}(v)\).

Note for instance that

\[
\text{depth}(u) := \text{disp}(u) + 1 + \text{disp}'(y) + 1 + \text{disp}(x) = \\
= \text{disp}(u) + 1 + \text{disp}(y) + 1 + \text{disp}(v) + 1 + \text{disp}(x) = \\
= 1 + \text{disp}(v) + 1 + \text{disp}(x)
\]

which is what is should be.
One can implement this case simply with

\[ \begin{align*}
x.w &:= x.w + y.w; \\
y.disp &:= y.disp + 1 + v.disp;
\end{align*} \]

4.32. If \( y.weight > x.weight \) then "balancing" requires that we attach \( x \) to \( y \) instead! Adjusting the displacements is slightly trickier now.

**EXAMPLE.**

where now \( \text{disp}'(y) = \text{disp}(y) + 1 + \text{disp}(v) + 1 + \text{disp}(x) \) and for proper correction \( \text{disp}'(x) = \text{disp}(x) - \text{disp}'(y) - 1 \).

We should execute

\[ \begin{align*}
y.w &:= y.w + x.w; \\
y.disp &:= y.disp + 1 + v.disp + 1 + x.disp; \\
x.disp &:= x.disp - y.disp - 1;
\end{align*} \]

(and the order of the instructions is important!).

4.33. The given implementation of \( m \) depths and \( n - 1 \) merges actually requires \( \sim 2n \) extra auxiliary depth-instructions, but since \( m \geq n \) the time needed is the same as for the underlying finds and unions on the shadow-forest. Apply Tarjan's theorem.

4.34. A generalisation of the techniques for merge-depth programs for other functions over trees was formulated recently by Tarjan [57]. He showed for instance that one can verify a minimal spanning tree on an \( n \)-node graph with \( m \) edges in time \( \sim m \alpha(m,n) \).
5. ASSOCIATIVE SEARCH STRUCTURES

5.1. The known techniques for data-structuring can usually be combined somehow by a creative programmer into an acceptable organisation for efficiently handling records on an identifying primary key, but additional, non-trivial considerations may be needed to work it out for accessing records on secondary keys.

In such applications a user can issue data-requests (or queries) asking the system to produce all data-items from a file or data-base which possess a desired combination of attributes.

Queries are typically of the form

- list all records with property P in workspace W -

where P may be specified by some formula in relational algebra or, say, as a predicate in a calculus-based language like Codd's data-sublanguage ALPHA [10].

The process of answering data-requests is called data-retrieval. (One may speak of re-three-val after probing the data-base twice.)

5.2. An important task in data-base management consists of organising information such that all queries in a known language-system can be handled quickly. One may also require that in realistic applications this property must be maintained under a choice of admissible operations on the data-set.

The task usually reduces to

- resolving the nearly always present time/
  storage trade-offs favourably -

and in close connection to it

- finding the optimal degree of redundancy
  needed in the data-base -

5.3. There may be an advantage in keeping lists with instant answers to basic queries, and it is conceivable that in such situations one record has to occur on many lists.

Note that redundancies in the data-model can possibly be avoided in the physical data-base.
5.4. A common principle learns not to store a data-item where it is not needed, and not to duplicate a data-item when you can avoid it.

The use of pointers tends to be successful exactly for this reason, but one should not forget that storing many pointers may require much space also.

5.5. One may wish to avoid storing explicit pointers and try to "hack" a pointer (as a record key) in little pieces which one can search in a "small" auxiliary table first.

If at least one piece is not found then there is no point in searching the original, much larger file.

5.6. A technique of Bloom [10] suggests to maintain a long bit-string

\[ a_0 \ldots a_M \quad (\text{some } M) \]

and to use hashing functions \( h_1, \ldots, h_s \) as follows. Mark that record \( k \) is stored by setting bits \( a_{h_1(k)}, \ldots, a_{h_s(k)} \) to 1, as if it were unique coordinates for \( k \).

Provided \( \binom{M+1}{s} > n \), there are good chances that this method can distinguish many records.

5.7. In cases where pointers are used to link elements together in a (linear) list, the need for extra space seems unavoidable.

EXAMPLE. Doubly linked linear lists are usually built from elements of

\begin{verbatim}
type block = record
  llink : block;
  info : ...;
  rlink : block;
end;
\end{verbatim}

Nevertheless (depending on the application) it may take little effort occasionally to eliminate much of the pointers even here.
EXAMPLE. Suppose a doubly linked list was chosen in order to traverse a file from left-to-right and back. Then one can save 50% in pointer-space by using a linked structure

One can move one block to the right with

\[
\begin{align*}
\text{if } & (\text{next} := \text{pos.link}; \text{next} = \text{nil}) \text{ then } \text{exit } \text{fi;} \\
\text{pos.link} & := \text{prevlink}; \\
\text{prevlink} & := \text{pos}; \\
\text{pos} & := \text{next};
\end{align*}
\]

and one can move one block back (to the left) with

\[
\begin{align*}
\text{if } & (\text{next} := \text{prevlink}; \text{next} = \text{nil}) \text{ then } \text{exit } \text{fi;} \\
\text{prevlink} & := \text{next.link}; \\
\text{next.link} & := \text{pos}; \\
\text{pos} & := \text{next};
\end{align*}
\]

5.8. It is common programming practice also sometimes to add a pointer into records as a means of referring to a part in memory where auxiliary information relevant to each individual record is stored.

It is probably much better to do so than to list the auxiliary information explicitly in each record in detail, especially in cases where many records share the same data.

Thus pointers are used here to eliminate redundancy, and we pay only one additional field per record (instead of many).

There are many variants of this idea; one may want to eliminate the pointers altogether and use a table at a "known" location instead in which is listed where the auxiliary data for each record can be found (but then again, at the cost of a table-search).

5.9. In an interesting paper Bobrow [11] pointed out very recently that the problem of locating "associated information" may be attacked by making better use of the one and only attribute of a record that comes for free: its address!
It is there anyway, and it occupies no space within the record.

The idea is that one can simply hash the address of a record (serving as its primary key) into a unique position of a table, and immediately find what we need.

The technique of hash-linking can be applied in various forms.

EXAMPLE. A company may keep records

<table>
<thead>
<tr>
<th>STATUS</th>
<th>ACCOUNTNO</th>
<th>AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20 bits</td>
<td>20 bits</td>
</tr>
<tr>
<td></td>
<td>20 bits</td>
<td>20 bits</td>
</tr>
</tbody>
</table>

for its customers which exactly fit in one word for amounts not exceeding 20 bits of storage.

The range of AMOUNT is conceivably larger, and in programming there is always the tendency to allocate enough space to handle "the worst case". Assuming that amounts usually fit in 20 bits one should not allocate more words for each record, but instead hash-link a cell to a separate, much smaller table with "special purpose" records

<table>
<thead>
<tr>
<th>ACCOUNTNO</th>
<th>AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 bits</td>
<td>40 bits</td>
</tr>
</tbody>
</table>

whenever its amount overflows (which presumably doesn't happen very often). An "all ones"-field in the original record can signal that it is hash-linked.

As long as there are no collisions one may compress the hash-table even further because it isn't necessary to keep keys in the table (a hash-link will immediately lead to the right spot). Otherwise one might mark a record to indicate that it collided, and that one must follow a hash-link into an overflow-area where this time identifying keys are available.

5.10. The methods for locating "secondary information" in a data-file get more involved when we consider querying a data-base.

One might distinguish the following types of queries (following Rivest [53])
- exact match queries
  locate the record with key k

- component match queries
  locate all records in which the i-th
  attribute of the key is j

- partial match queries
  locate all records for which s specified
  attributes match given data (and the
  remaining attributes can be arbitrary)

- range match queries
  locate all records for which specified
  attributes are within a given range

- good match queries
  locate all records which have a key
  differing from k by at most a given tolerance

- best match queries
  locate all records which have a key with
  least distance to k

In more general, boolean queries one can specify records by means of an
arbitrary propositional formula over the attributes.

5.11. Most queries seem to force a search through the entire data-base,
and there is great need for a specific data-organisation (perhaps one for
each single type of queries) to support faster retrieval. It is the object
of current research in software and in hardware systems to implement associative search in a best possible way.

There are probably natural limits to how efficient data-retrieval can
be. If one specifies more and more conditions on the attributes, then the
amount of work to test each record increases while the actual number of
records satisfying the query probably decreases.

5.12. Consider the following, typical library search problem.

Suppose records are classified on k out of many possible keywords,
and suppose we want to store records in such a way that we can later find all records with \( s \leq k \) attributed keywords quickly without having to inspect each element of the data-base individually.

How can such a goal be achieved?

Common solutions are based on an appropriate partitioning of the data-base in various buckets.

5.13. Apparently Gustafson [35] systematically investigated such problems first, and in 1969 he suggested the following variant of hashing for obtaining an acceptable solution.

Choose an "ordinary" hash-function

\[ h : \text{(keywords)} \rightarrow \{0, \ldots, b-1\} \]

for some \( b > k \), and build a "super" hash-function

\[ H : \text{unordered } k\text{-tuples} \rightarrow b\text{-bit numbers} \]

classifying each record with attribute-set \( \{\beta_1, \ldots, \beta_k\} \) in the bucket with address \( a_0 \ldots a_{b-1} \) where

\[ a_i = 1 \iff \exists j \ h(\beta_j) = i. \]

We thus hash each key into a bit-position, and sometimes speak of hash-coding. Assuming a more or less uniform distribution, buckets will have \( \sim \frac{N}{b} \) elements each.

5.14. To search for all records which contain keywords \( \gamma_1, \ldots, \gamma_s \) (some \( s \leq k \)) one should first determine the partial address \( a_0 \ldots a_{b-1} \) with

\[ a_i = \begin{cases} 1, & \text{if } \exists j \ h(\gamma_j) = i \\ * \end{cases} \]

and then fill in the *'s with \( k-s \) ones and \( b-k \) zeros in all possible ways to obtain the names of all buckets which one would have to inspect only.

On the average one would have to scan through no more than

\[ \sim \binom{b-s}{k-s} \cdot \frac{N}{\binom{b}{k}} = \frac{\binom{k}{s}}{\binom{b}{s}} \cdot N = \]
\[ = (k-s+1) \ldots (b-s) \frac{k!}{N} \]

which indeed tends to be a very small fraction of the entire data-base for \( s \sim o(k) \).

The method is promising for small \( k \).

5.15. We note in passing that methods for partial match retrieval based on intersecting inverted lists will not be discussed here (see Engles [25] or Roberts [54]).

5.16. In 1971 Rivest (see Knuth [44], Rivest [52]) suggested another, simple hash-coding algorithm for partial match retrieval in a data-base.

Suppose now that keys are ordered \( k \)-tuples, and assume it simply are length \( k \) strings over some alphabet \( \Sigma \) with \( \# \Sigma = c \).

Determine an "ordinary" hash-function

\[ h: \Sigma \rightarrow b \text{-bit codes} \]

(whence perhaps \( 2^b < c \)), and build a super hash-code

\[ H: \Sigma^k \rightarrow c \text{-bit numbers} \]

with \( c = kb \) by defining

\[ H(\beta_1, \ldots, \beta_k) = h(\beta_1) \ldots h(\beta_k). \]

There will be \( \sim N/2^c \) elements per bucket on the average.

5.17. Let a partial match query be represented as a length \( k \) string

\[ \gamma = \gamma_1 \ldots \gamma_k \]

where for each \( i \)

\[ \gamma_i \in \Sigma \ ("specified") \ or \ \gamma_i = * \ ("unspecified"), \]

and suppose there are \( s \) stars.

As in Gustafson's method, again, we only have to search in the \((2^b)^s\) buckets with a name matching \( \gamma \) in the specified positions.

It means that on the average we must inspect
\[
\sim \frac{N}{2^b} \cdot 2^{bs} = 2^{b(s-k)}_N = o^{s-k}_N
\]

elements, which is
\[
\sim 2^{s \cdot \log o} = o^s \sim N \frac{s}{k}
\]

for \( b \sim \log o \) and a full data-base with \( N \sim o^k \).

5.18. Provided enough extra storage is available there is a simple variant of Rivest's algorithm from which one may expect an even better behaviour.

Partition each length \( k \) key in \( m \) fields of base-\( o \) numbers of \( \frac{k}{m} \) digits

\[
\begin{array}{llllll}
A_1 & A_2 & \ldots & \ldots & \ldots & A_{\frac{k}{m}} \\
\end{array}
\]

and store each record with such a key in bucket \( A_j \) of the \( j \)-th data-base for each \( 1 \leq j \leq m \).

We thus list each record many times in our data-model, but it will appear that the redundancy will pay off in greater retrieval-efficiency.

5.19. In a partial match query with a total of \( s \) stars there must be at least one field with \( \leq s/m \) stars. Among these choose the field with the fewest number of stars (say field \( j \)).

One may expect that the \( j \)-th data-base will have the elements we search for concentrated in the smallest number of buckets.

Roughly, there are \( \sim N/o^{k/m} \) elements per bucket and we have to scan through
\[
\sim \frac{N}{o^{k/m}} \cdot o^{s/m} = o^{s-k/m} \cdot N
\]

elements. This compares favourably with the bound \( \sim o^{s-k} \cdot N \) in 5.17 (although a more precise analysis would be needed to support the apparent conclusion).

5.20. One may similarly develop other methods, and it becomes of interest to study how the various hash-coding algorithms for partial-match retrieval compare in efficiency.

Rivest [52] (also [53]) answered this question for a specific class of hashing algorithms, and we shall give a simplified approach to make his result understandable.
5.21. Each hash-code storage schema consists of a super hash-function

\[ H: \mathbb{L}^k \rightarrow \text{a finite set of buckets} \]

(the algorithm of 5.18 is not in this category).

The relation "having the same H-value" gives a partitioning of \( \mathbb{L}^k \),
and the various equivalence-classes may be called \( H\)-blocks.

Now consider how we answer partial-match queries \( \gamma \) with \( s \) stars. The
algorithm must return the names of all \( H\)-blocks which contain at least
one record matching \( \gamma \) in the specified positions.

Rivest suggested to "measure" \( H \) by the average number of \( H\)-blocks
needed to answer \( s\)-star queries.

5.22. Suppose there are \( Q \) \( s\)-star queries.

On the average we need to inspect a number of blocks equal to

\[
\frac{1}{Q} \sum_{\gamma} \left( \# \text{\( H\)-blocks with a \( \gamma \)-match} \right)
\]

\[ = \frac{1}{Q} \sum_{B} \left( \# \text{\( s\)-star \( \gamma \)'s touching \( B \)} \right), \]

where \( B \) ranges over all \( H\)-blocks.

Each \( s\)-star query occurs at least once in this summation, and it is
counted more than once if and only if it hits more than one block.

It is conceivable (and Rivest proved it for "balanced" hash-coding
algorithms) that the average behaviour will come out best once the "answer"
to as many partial-match queries as is practical is contained in one block.

It means that preferably \( H\)-blocks must allow a decomposition into

\[ \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_k \]

where for each \( i \): \( \Sigma_i = \Sigma \) or \( \# \Sigma_i = 1 \).

It holds for the algorithms of Gustafson and Rivest, and will hold
for later algorithms also.

5.23. Based on these heuristics it seems promising to consider a general
type of (hash-coded) partial-match file design which consists of a table
linking independent key-patterns and bucket-addresses
<table>
<thead>
<tr>
<th>PATTERN</th>
<th>BUCKET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><em>1**001</em>1*</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where

(i) each row contains s stars and k-s symbols from Σ;

(ii) any two key-patterns (rows) differ in at least one specified position

(which means that buckets will be disjoint).

Technically each entry in the table is an s-star partial-match query, but obviously other queries can be answered from it also. If no entry matches a record-query then no records of that type are in the data-base.

The way stars are distributed in the key-patterns can strongly influence the average performance of the file-design. Rivest [53] and later Burkhard [14] (see also Bentley & Burkhard [9]) studied several special designs with an alleged approximately optimal behaviour.

5.24. One may assume that entries in the table (as strings over Σ u {*}) are listed in radix-sorted order (a general storage method suggested by Hildebrand & Isbitz [36] as early as 1959). It is likely that the best results will be implied when the entire data-base is stored in a compatible manner.

The idea is that in such a scheme it becomes unnecessary to store the table separately, and its structure (or rather, the structure of its buckets) is implied by a straightforward digit-wise search where each "*" is interpreted as a "branch in both directions"-instruction.

5.25. Assume that Σ = {0,1} (for simplicity).

When length-k strings of 0's and 1's are entered into a tree where at each node "0" is interpreted as "left" and "1" as "right", each record-key becomes a code for the path to a unique leaf where the corresponding record-address can be stored.

The idea is independently due to de la Briandais [23] and Fredkin [30], and the resulting storage structure is known as "trie memory" (pronounced usually as "try-memory", although the editor of the section "Techniques" in the 1960-issue of the Communications pointed out in a footnote to Fredkin's paper that "trie" apparently was derived from the word re-TRIE-val).
EXAMPLE. The trie corresponding to \( \{001000, 001011, 101000, 101001, 111011\} \) is

Tries are often used when there are records with variable-length keys also.

5.26. Despite the clarity of the file-structure, many programmers observed that tries can be somewhat inefficient in their use of storage.

One reason is that in entering keys we may not have made sufficient use of the observation that some strings could have had identical segments of bits which we should have followed first.

DEFINITION. A \( \pi \)-trie (where \( \pi \in S_k \)) is the storage-structure resulting after entering keys \( \gamma \) with bits ordered as \( \gamma_{\pi(1)}\gamma_{\pi(2)}\cdots\gamma_{\pi(k)} \).

EXAMPLE. The 341256-trie for \( \{001000, 001011, 101000, 101001, 111011\} \) (see 5.25) is

It has only 12 internal nodes, as compared to the 123456-trie in 5.25 which
has 16.

It is unlikely that there will be a polynomial-time bounded algorithm to find an optimal \(\pi\)-trie for a file, since Comer & Sethi [20] recently proved the following interesting result.

**Theorem.** The problem to determine a \(\pi\)-trie with the smallest number of internal nodes for a file is NP-complete.

5.27. Given tries (or \(\pi\)-tries) can be optimized in various, simple ways.

Observe, for instance, that one could eliminate chains of the form

\[
\begin{array}{c}
A \\
\vdots \\
\vdots \\
\vdots \\
r_j
\end{array}
\]

(which Comer & Sethi call leaf-chains), and prune the tree in \(A\) at the earliest moment that \(r_j\) is uniquely identified.

The idea is that the remaining bits of the key need not take up trie-storage and can be checked elsewhere.

**Example.** The pruned 341256-trie for \{001000,001011,101000,101001,111011\} is

\[
\begin{array}{c}
1 \\
0 \\
| \\
0 | 1 \\
| \\
0 | 1 \\
r_1 r_2 \\
r_3 r_4 r_5
\end{array}
\]
It seems to require an entirely different strategy to determine the best possible pruned π-trie for a file.

EXAMPLE. The 651234-trie for \{001000, 001011, 101000, 101001, 111011\} is

![Diagram of a trie]

It has as many as 21 internal nodes, but the pruned version has only 5.

Comer & Sethi [20] proved also

**THEOREM.** The problem to determine a pruned π-trie with the smallest number of internal nodes for a file is NP-complete.

5.28. A next step would to collapse long chains

![Diagram showing collapse]

in the interior of the trie also, by adding an instruction at node \(A\) to skip over bits \(γ_π(i+1), \ldots, γ_π(j)\) and to continue immediately with bit \(γ_π(j+1)\)'

The idea was apparently first proposed by Morrison [48] in 1968, and
implemented in his system PATRICIA. Testing the skipped bits of the key afterwards avoids the need for ineffective trie-storage, and the technique leads to the most compact \( \tau \)-trie.

The problem to determine an "optimal" compacted \( \tau \)-trie is meaningless if the number of internal nodes is the only measure, and one should now also take the degree of balancing into account.

5.29. In a \( \tau \)-trie we could have inscribed the tested bit-positions in the corresponding nodes.

**EXAMPLE.**

![Diagram]

Clearly in partial-match retrieval some bits in the input will be unspecified, and when a * is found one must descend down two branches (if there are two) simultaneously (and so on ..), thus implicitly defining what bucket is searched.

5.30. Carrying this idea one step further, Birkhard [15] recently formulated a type of generalised trie which seems an appropriate structure for partial-match retrieval from large indexed files.

A \((k,s)\) PM-trie is a complete binary tree with \(2^{k-s}\) leaves in which

(i) each leaf corresponds to a bucket;

(ii) each internal node is labeled by one out of \(k\) possible bit-positions;

(iii) on no path from the root to a leaf there is a bit-position which is tested more than once.

(Thus each such path is like an \(s\)-star query.)

Technically each \((k,s)\) PM-trie directly implies a \((k,s)\) partial-match file-design in the sense of 5.23, but not conversely.
EXAMPLE.

\[
\begin{array}{c}
1 \\
2 & 3 \\
4 & 5 & 6 \\
B_1 & B_2 & B_3 & B_4 & B_5 & B_6 & B_7 & B_8
\end{array}
\]

is a (6,3) FM-trie.

To test how good a FM-trie is we shall measure how many buckets must be searched in worst case to answer a t-star query \( t \leq k \).

The result is likely to depend on the way tests are distributed over the tree, and the task is to find "good" \( (k,s) \) FM-tries for use on files with \( k \)-bit entries.

EXAMPLE. The query \( ***010 \) in the given \( (6,3) \) trie leads into 5 buckets.

The query \( *1**** \) leads into 5 buckets also, despite the fact that it is "almost unspecified".

5.31. In early 1975 Burkhard [14] first formulated and analysed a type of "good" \( (2n+1,n) \) FM-tries. We shall present a modified version which was shortly later suggested by Dubost & Trousselle [24].

DEFINITION. A \( (2n+1,n) \) FM-trie is called a BDT-trie of order \( n \) if and only if

(i) the root (at level 1) has label 1;

(ii) for each node in level \( i \) \( (1 \leq i \leq n) \) the left-son is labeled \( i+1 \) and the right-son is labeled \( 2n+2-i \).

Observe that the tested bit-positions at the nodes may be computed while descending down the tree very easily.
EXAMPLE. The BDT-trie of order 3 is

\[
\begin{array}{c}
\begin{tikzpicture}
  \node {1} [grow之上={2, 7}]
  \node {2} [grow之上={3, 6}]
  \node {3} [grow之下={4, 5}]
  \node {4} [grow之下={B_1, B_2}]
  \node {5} [grow之下={B_3, B_4}]
  \node {6} [grow之下={B_5, B_6}]
  \node {7} [grow之下={B_7, B_8}]
  \node {8} [grow之下={B_9, B_{10}}]
  \node {9} [grow之下={B_{11}, B_{12}}]
  \node {10} [grow之下={B_{13}, B_{14}}]
  \node {11} [grow之下={B_{15}, B_{16}}]
  \end{tikzpicture}
\end{array}
\]

Observe that in a BDT-trie of order \( n \) all paths have length \( n+1 \), and there is an obvious regularity in the assignment of tests.

The labels of two brothers always add to \( 2n+3 \).

Let \( T_n \) denote the BDT-trie of order \( n \).

LEMMA. For all \( n \geq 1 \)

\[
T_n = \begin{array}{c}
\begin{tikzpicture}
  \node {1} [grow之上={2, 2n+1}]
  \node {2} [grow之下={1}]
  \node {3} [grow之下={2n+1}]
  \end{tikzpicture}
\end{array}
\]

where each marked sub-tree is \( T_{n-1} \) with 1 added to all labels (and 2n to the root of the right copy).

5.32. Formulas for the worst case performance of BDT-tries were first given by Burkhard [14]. We shall outline an interesting proof-technique due to Dubost and Trouve.

The problem is how to visualize what buckets can be entered in a t-star query.

Suppose we first follow the trie-path until we hit the first bit-test which probes a star.
It means we narrowed the search for buckets to a small sub-trie.

However, each sub-trie (essentially) is a BDT-trie for sub-keys of the original file.

Thus we reduced the task to finding the maximal number of buckets reached in a t-star query with a * in position 1 (it will turn out to be an increasing function in n).

5.33. A consistent assignment with t stars in a BDT-trie is a complete marking of all (internal) nodes with "stars" and "dots" such that
(i) the root is marked with *

(ii) If a left-son in level i is marked with a *, then so are all left-sons in level i (which is reasonable since they probe the same bit-position).

If it happens, then the left-sons in level i are said to contribute one star.

(iii) If a right-son in level i is marked with a * then all are, and the right-sons are said to contribute one star also.

(iv) The number of contributed stars adds up to t.

EXAMPLE.

The search-structure in a BDT-trie of order 3 for a query ******* (where
dots mean defined bits). No matter what, 9 buckets will be visited on any 5-star query of this pattern.

The reason for introducing this concept is that each t-star query implies a consistent assignment with t stars and conversely.

5.34. For symmetry-reasons we may assume that if a level \( i > 1 \) contributes one star then it is contributed by the left-sons in that level.

Observe further that if two brothers are identically marked

```
      O
     /|
    / \|
   /   V
  S_1  S_2
```

```
      O
     /|
    / \|
   /   V
  S_3  S_4
```

or

then the entire, corresponding sub-trees are identical in marking (i.e., \( S_1 \equiv S_2 \) and \( S_3 \equiv S_4 \) respectively).

5.35. Given a consistent assignment one can easily determine the largest possible number of buckets generated by a corresponding query.

Collapse and prune the tree according to the following rules.

- for each dotted node with not identically marked sons,
  purge the sub-tree of the dotted son

Because the bit at A is specified we would enter either \( S_1 \) or \( S_2 \). The sub-trees are identically marked except at their root, so entering \( S_1 \) will definitely lead into a larger number of buckets.
- for each dotted node with two identically marked sons, purge one of its subtrees also.

The left- and right-subtrees are identical, and since we enter only one on the specified bit we need only count buckets in one sub-tree.

One could go one step further and collapse all chains of dotted nodes left. The number of leaves in the resulting tree is exactly the number of buckets to be visited.

5.36. It follows that the number of buckets visited by t-star queries of the same *-pattern is the same.

5.37. Consider first how many buckets result when in the original, consistent assignment each level contributes one star.

**Lemma.** For BDT-tries of order n this number is $F_{n+3}$.

**Proof.** By induction.

For $n = 0$ we have

$$
\begin{array}{c}
\ast \\
\times
\end{array} 
\rightarrow 2 \equiv F_3
$$

and for $n = 1$ we have (remember: only one star per level)

$$
\begin{array}{c}
\ast \\
\times \\
\ast \\
\times \\
\ast
\end{array} 
\rightarrow 3 \equiv F_4
$$

For $n + 1$ the marked trie-structure $S_{n+1}$ of this form decomposes as

$$
\begin{array}{c}
\ast \\
\times \\
S_n
\end{array} 
\quad \ast \\
\times \\
S_{n-1}
$$
and the number of buckets adds up to $F_{n+3} + F_{n+2} = F_{n+4} \equiv F_{(n+1)+3}$.

5.38. Next we consider consistent assignments in which each level contributes at least one star, and the number of levels indeed contributing two stars is assumed "fixed".

Examine the "lowest" layer of two-star levels in such an assignment

The number of buckets visited is

$$s \cdot 2^b \cdot F_{a+3} \cdot F_{c+3}$$

(using the previous lemma). Assuming b fixed, and observing that

$$s \cdot 2^b \cdot F_{a+3} \cdot F_{c+3} \leq s \cdot 2^b \cdot F_{3} \cdot F_{a+c+3}$$

it follows that the number of buckets is maximal if $a = 0$. We immediately obtain

**Lemma.** In a consistent assignment with one- and two-star levels the number of buckets is largest if all two-star levels occur together and precede all one-star levels.

5.39. In a general consistent assignment there can be dotted levels also, and we shall now analyse where they have to occur for obtaining the largest number of buckets.

Our argument here differs from the proof of Dubost and Trousse who
tried to "collapse" non-contributing levels in the "worst possible" way.

Let

\[ w_n(t) = \text{the maximum number of buckets visited with a} \]
\[ t\text{-star query on a BDT-trie of order } n. \]

5.40. Observe first that the number of buckets for an assignment does not change if we add non-contributing levels at the bottom of the trie.

The idea is to make a given assignment "worse" by an exchange operation on the levels without changing the order of the trie and without changing how many contributed stars there are. (Thus the assignment remains a t-star query).

Do the following reductions whenever levels of the indicated form occur.

(i)

(one-star and

two-star levels)

\[ \rightarrow \]

\[ \text{two-star levels} \]

\[ \text{one-star levels} \]

This is the simple transformation we proved in 5.38. It can be applied to subtrees also, provided the same transformation brings profit in all subtrees at this level (viz. when it is a two-star level).

(ii)

(one-star and

two star levels)

\[ \text{.................................} \]

It is easy to see that the worst case occurs if from the dotted level we
always proceed to the starred son. Thus we can apply (i) in the subtrees and should have all two-star levels immediately follow the initial one-star level

\( \rightarrow \)

two-star levels

\[
\begin{array}{c}
\text{one-star levels} \\
\end{array}
\]

\[(iii)\]

\[
\begin{array}{c}
\text{From each node in the dotted level we reach two stars two levels lower. Now borrow a star from the two-star level and make the dotted level into a one-star level:}
\end{array}
\]

Considering how one can enter the formerly dotted level to get a worst case now, it follows that at least as many buckets are entered now as we entered before.
A two-star level and a dotted level can always be interchanged. (With the previous transformations it now follows that for the worst case to occur all two-star levels must occur at the top of the tree).

(v) Anywhere in the tree the number of consecutive dotted levels can be reduced to 1, provided we add the deleted levels at the bottom (to preserve the order of the tree).

(vi) It is always "better" to have the dotted level occur in between the two one-star levels.
Note that this transformation together with the other tries to sift the dotted levels down and use them to separate as many one-star levels as it can.

5.14. If we perform these operations then we reduce a given assignment to a kind of normal form for the worst case:

```
  x two-star levels
  
  y one-star levels
  
  z one-star levels separated by dotted levels
  
  .... dotted levels
```

where \( 1 + x + y + 2z \leq n + 1 \) (the number of levels) and \( 1 + 2x + y + z = t \) (the number of stars).

5.42. We can now combine all information and prove Burkhard's theorem.

**THEOREM.** For t-star queries on a BDT-trie of order \( n \)

\[
W_n(t) = \begin{cases} 
2^t & \text{for } 0 \leq t \leq n+1 - \left\lceil \frac{n}{2} \right\rceil \\
2^{n+1-t} F_{2t-n+1} & \text{for } n+1 - \left\lceil \frac{n}{2} \right\rceil < t \leq n+1 \\
2^{t-n-1} F_{2n+4-t} & \text{for } n+1 < t \leq 2n+1.
\end{cases}
\]

**Proof.** For the worst case stars must be distributed as in the tree of 5.41. We reach \( 2^x \) nodes in the last all-star level (which could be the root if \( x = 0 \)), \( F_{y+1} \) stars and \( F_{y} \) dots from each of these in the last of the consecutive one-star levels, and in the last part of the tree we get into \( 2^{x+1} \) buckets from a star and \( 2^x \) buckets from a dot.

It follows that we must maximize the total of

\[
2^X \cdot F_{y+1} \cdot 2^{x+1} + 2^X \cdot F_y \cdot 2^x = 2^{x+2} (2F_{y+1} + F_y) = 2^{x+2} \cdot F_{y+3} \text{ buckets},
\]

under the condition that
\[ x + y + 2z \leq n \]
\[ 2x + y + z = t - 1. \]

5.43. In a subsequent generalization Burkhard [14] formulated another, more flexible type of FM-trie in which he abstracts the essential features that make BDT-tries work.

**DEFINITION.** A \((2n+1,n)\) FM-trie is called a G-trie of order \(n\) if and only if
(i) any label \(j\) \((1 \leq j \leq 2n+1)\) occurs in a unique level;
(ii) brothers carry different labels.

Burkhard proves that the expressions of 5.42 are upperbounds for order \(n\) G-tries in general.

Burkhard [16] also develops similar kinds of tries for files where keys are over an arbitrary alphabet \(I\).

6. PATTERN-MATCHING

6.1. One may describe pattern-matching as the task of finding one or more sub-structures which meet a certain specification in given larger structures of some kind (usually in on-line manner).

Pattern-matching operations occur in lexical analysis, in searching library-files, in text-editing and in symbol-manipulation, and although it may seem easy we often need non-trivial data-organisations to perform the operations quickly without undesirable overload.

6.2. Pattern-matching is undoubtedly the most powerful feature of the SNOBOL 4 programming language, and it often occurs in conjunction with replacement (see Griswold [34]).

Although we shall restrict the algorithms to strings and one-dimensional patterns, one could also consider pattern-matching problems for arrays and trees (see Karp, Miller, and Rosenberg [41]).

6.3. Let all strings considered be over the alphabet \(I\) with \(#I = \sigma\).

The simplest algorithm to find all length \(d\) substrings of

\[ x_1x_2\ldots x_n \]

\((n \geq d)\)
makes use of a *shift-register* containing

\[
\text{state} = |x_{i+1}|\sigma^{d-1} + |x_{i+2}|\sigma^{d-2} + \ldots + |x_{i+d}|
\]

\[
\vdots
\]

\[
\ldots x_{i+1}x_{i+2} \cdots x_{i+d} \ldots
\]

current position of scanner

Suppose that \(\sigma^d\) buckets \(B_0, B_1, \ldots, B_{d-1}\) are allocated, one for each possible string of length \(d\). Then one can collect all information from \(x\) in one scan as follows

\[
\begin{align*}
\text{state} &:= |x_1|\sigma^{d-1} + |x_2|\sigma^{d-2} + \ldots + |x_d|; \\
\text{enter } 1 \text{ in } B_{\text{state}}; \\
\text{position} &:= d; \\
\text{while } \text{position} < n \text{ do} \\
\text{position} &:= \text{position} + 1; \\
\text{state} &:= \text{state} \cdot \sigma + |x_{\text{position}}| \text{ (mod } \sigma^d); \\
\text{enter } \text{position} - d+1 \text{ in } B_{\text{state}}; \\
\text{od}; \\
\end{align*}
\]

In the end \(B_{|y_1|\sigma^{d-1} + \ldots + |y_d|}\)

\[
\begin{align*}
x_{i+1} \ldots x_{i+d} &= y_1 \ldots y_d. \\
\end{align*}
\]

The algorithm essentially traverses a finite state automaton with a very special transition-structure (known as a de Bruijn graph in combinatorics) which could have been computed and stored in advance. The approach is of reasonable complexity only if arithmetic modulo \(\sigma^d\) can be performed quickly.

6.4. More common pattern-matching algorithms of low complexity develop and use *positional information* of non-numeric nature.

A typical procedure for classifying and locating all length \(d\) sub-strings of \(x\) makes use of the following equivalence-relation
positions i and j of x are s-equivalent if and only if
\[ x_i \ldots x_{i+s-1} = x_j \ldots x_{j+s-1}. \]

Our representation of equivalence-classes will differ from Karp, Miller, and Rosenberg [41], and simplify the final algorithm somewhat.

6.5. Let the name of an equivalence-class be the smallest element it contains. \( \text{CLASS}[i] = j \) means that position i belongs to the s-equivalence class named j.

The elements of an equivalence-class will be linked together in ascending order using NEXT. The end of a NEXT-chain will be marked by a negative integer which yields the beginning address of another class-chain, and 0 if all classes have been linked together.

One could enumerate all positions in equivalence classes with

\[
\begin{align*}
\text{position} & := 1; \\
\text{repeat} & \\
\alpha & := \text{CLASS}[\text{position}]; \\
\text{print position} & \\
\text{while NEXT}[\text{position}] > 0 & \text{ do} \\
\text{position} & := \text{NEXT}[\text{position}]; \\
\text{print position} & \\
\text{od} & \\
\text{end of class } \alpha & \\
\text{position} & := -\text{NEXT}[\text{position}]
\end{align*}
\]

6.6. The problem is how one can construct the s-equivalence classes in this representation quickly. The idea is an inductive approach based on the following result.

**Lemma.** For all \( r \leq s \) and i and j, \( i \sim_{r+s} j \iff i \sim_s j \& i+r \sim_s j+r \).

**Proof.** By overlap. \[\square\]
We can now show

**Theorem.** Given the representation of $s$-equivalence classes one can construct the $r+s$-equivalence classes for any $r \leq s$ in linear time.

**Proof.** Assume that we have $n$ queues $Q_1, \ldots, Q_n$ available (one for each equivalence class).

Read out all $s$-equivalence classes (which yields the elements per class in ascending order) and attach each position $i$ so encountered at the rear of $Q_j$, where $j = \text{CLASS}[i+r]$ (provided $i+r+s-1 \leq n$).

In the end each $Q_j$ is either empty or made up of blocks

```
    .......   *.*   .......   .......   .......
    \      \     \       \    \      \    \
  front          rear
```

where each block simply consists of the items attached to $Q_j$ while reading out an entire $s$-class. Note that each block can be identified because it is the longest stretch of elements with the same particular \text{CLASS}-value.

Observe now that $i$ and $j$ are in the same block if and only if $i \sim_s j$ and $\text{CLASS}[i+r] = \text{CLASS}[j+r]$, thus by the lemma if and only if $i \sim_{r+s} j$. Thus the blocks exactly are the $r$-$s$-equivalence classes.

One can now obtain the representation of $r+s$-classes by reading out the queues block after block. The first element of a block by definition is its name (since it is smallest), and it can be retained and put into \text{CLASS} while passing through the block. The classes can be linked in the order in which the queues are emptied. (Start with the block containing $1$, which must occur up-front somewhere.) \hfill \square

6.7. It follows that

**Theorem.** One can construct the representation of $s$-equivalence classes for $x$ in $\sim \sigma \cdot n + n \log s$ steps.

**Proof.** The $1$-equivalence classes follow by a simple scan of $x$ in $\sim \sigma \cdot n$ steps. (There is one chain for each distinct symbol of $x$.) If the representation of the $r$-equivalence classes is known, then we can construct the $2r$-equivalence classes in linear time. Thus the following algorithm will do
\begin{verbatim}
r := 1;
record the r-classes;
while 2r < s do
  r := 2r;
record the r-equivalence classes
end;
record the (s-r)\n classes from the r-classes;
\end{verbatim}

6.8. To find all substrings of length \(d\) in \(x\) with the given algorithm works pleasantly in \(\sim n \log d\) steps (assuming a fixed alphabet). A direct search for the classes with more than one element gives all repeated patterns of length \(d\) in about the same number of steps.

The same principle as in 6.7 can be used to show

THEOREM. One can find all longest repeated substrings of \(x\) in 
\(\sim \sigma \cdot n + n \log n\) steps.

Proof. Karp, Miller, and Rosenberg [41] observed that the problem is equivalent to finding the largest \(r\) such that there is at least one \(r\)-class with more than one element.

Determine the \(1\)-classes first in \(\sim \sigma \cdot n\) steps, and then find \(r\) with binary search after a usual "doubling" procedure has located a lower bound \(d\) and upper bound \(2d\) for it. \(\Box\)

One can develop similar algorithms based on \(s\)-equivalence to determine for instance whether \(x\) is of the form \(y^k\) for some \(k \geq 2\) and \(y\).

6.9. The idea of positional equivalences is clearly of great help in solving pattern-matching problems, but it seems to be efficient only when we search all patterns of some kind. Note that the algorithm in 6.8 does not simplify (for instance to complexity \(\sim \sigma n\)) in case we just want any longest repeated substring.

Another draw-back is that the algorithms become more inefficient (in storage and search-time) in case we must consider and "remember" several \(s\)-equivalences at the same time.
6.10. Weiner [65] proposed a very interesting idea for extracting exactly all the information you need to know from the various s-equivalences.

Let us assume that x always has a fixed marker at the end which is not occurring elsewhere

\[ x_1x_2\ldots x_n\$. 

DEFINITION. y is called the substring-identifer for position i if and only if it is the smallest u with the property that u occurs nowhere else in x except at position i.

EXAMPLE. The substring-identifiers for the various positions in

\[ bbababba\$

are

1  bbab
2  baba
3  aba
4  babb
5  abb
6  bba$
7  ba$
8  a$
9  $

The trick of adding $ at the end of x guarantees that each position has a substring-identifer, and one can observe that for each i the substring-identifer is precisely equal to the length-\( r \) substring at i for the smallest \( r \) such that there are no other positions in the \( r \)-equivalence class containing i.

6.11. Clearly no substring-identifer can be a proper prefix of another, and one can enter all substring-identifiers consistently in a 3+1-ary position-tree.
EXAMPLE. The position-tree for \texttt{bababa$} (the previous example) is

\begin{itemize}
  \item $a$
  \item $b$
  \item $\$$
  \item $a$
  \item $b$
  \item $\$\
  \item $a$
  \item $b$
  \item $\$$
  \item $a$
  \item $b$
  \item $\$
  \item $a$
  \item $b$
  \item $\$
\end{itemize}

The path with $y$ ends at the leaf with label $i$ if and only if $y$ is the substring-identifier of position $i$.

6.12. With the algorithm of Karp, Miller, and Rosenberg it follows that

THEOREM. The position-tree for $x$ can be constructed in $\sim s \cdot n \cdot L$ steps, where $L$ is the length of the longest substring identifier.

Proof. Construct the $s$-equivalence classes for $s = 1, 2, \ldots, L$. Each time a new relation is constructed, record the current $s$-value for all positions which now form a singleton equivalence class. Use the table afterwards to enter the corresponding substring identifiers in the tree in $\sim s \cdot L$ steps per entry. \hfill $\Box$

The theorem is not the best possible result though. A position-tree with many internal nodes (as many as $\sim n^2$ are possible even though it can only have $n$ leaves) must contain long straight chains.
which one can compress, resulting in a compacted position-tree of only
~ o(n) nodes.

Weiner [65] proved that one may construct the compacted tree directly
in only linear time. At the SWAT-meeting in Iowa City it was promptly
called "the algorithm of 73", because it took that many pages to explain.
(Easier descriptions are now available, see Aho, Hopcroft, and Ullman
[3].)

6.13. Once the position-tree is known all kind of pattern-matching questions
can be answered quickly.

Consider the problem of determining a longest repeated substring y of
x. Any such y must be the proper prefix of some substring-identifier.
Thus one must choose a y which leads to a lowest interior node in the
position-tree (and any such y qualifies).

EXAMPLE. A longest repeated substring of bbaaabba$ (see 6.11) is bab.

To find a longest common (contiguous) substring of x and y one can
construct the position-tree for

\[ x \neq y \$

and solve the task by searching for a lowest interior node with at least
one son representing a position in x and one son representing a position
in y.

6.14. The position-tree can be used also to solve the authentic pattern-
matching problem to determine whether given string y is a substring of x.

Let

\[ y = y[1]y[2] ... y[m] \]
\[ x = x[1]x[2] ........x[n] \]

with \( m \leq n \).

Before we see a direct algorithm for this problem it is of interest
to look at the method SNOBOL 4 uses

```
position := 0;
check := 1;
while(position < n - m+1 do
```
while x[position+check] = y[check] do
  check := check + 1
  until check > m;
  if check > m then match found & exit fi;
  position := position + 1;
  check := 1
od;

The method may require in worst case \( n \cdot m \) steps. Morris & Pratt [47] apparently first realized that these is a linear time pattern-matching algorithm. At the same time \( \sim 1970 \) the result followed from a general, linear time algorithm for simulating arbitrary 2-way deterministic pushdown-automata found by Cook [21], which prompted Knuth & Pratt to write a report entitled "automata-theory can be useful" [42].

6.15. Consider how one might improve the previous algorithm.

Suppose the algorithm has been in action for a while

\[
\begin{align*}
\cdots & \cdots x_{k-j+1} \cdots x_k & \text{(match)} \\
y_1 & \cdots y_j
\end{align*}
\]

and now finds that \( x_{k+1} \neq y_{j+1} \)

\[
\begin{align*}
\cdots & \cdots x_k & x_{k+1} \\
y_1 & \cdots y_j & y_{j+1}
\end{align*}
\]

Instead of trying again from the very beginning one should continue at the largest \( i \) such that

\[
\begin{align*}
\cdots & \cdots x_k & x_{k+1} \\
y_1 & \cdots y_i & y_{i+1} \quad \text{with} \quad x_{k+1} = y_{i+1}
\end{align*}
\]

which is exactly the largest \( i \) such that

\[
\begin{align*}
y_1 & \cdots y_j \\
y_1 & \cdots y_{i+1} \quad \text{with} \quad x_{k+1} = y_{i+1}
\end{align*}
\]
Candidate i's can therefore be determined based solely on y.

6.16. The following concept now appears relevant.

**DEFINITION.** The *overlap-identifier* \( h(j) \) of position \( j \) in \( y \) is the largest \( i < j \) such that

\[
\begin{array}{c}
\ldots \ldots \quad y_j \\
\ldots \ldots \quad y_i \\
\end{array}
\]

Observe that always \( h(1) = 0 \). Let \( h(0) = 0 \) for consistency.

**EXAMPLE.** The overlap-identifiers of babaaba are

\[
\begin{array}{cccccccc}
 b & a & b & a & a & b & a \\
 0 & 0 & 1 & 2 & 0 & 1 & 2 \\
\end{array}
\]

**LEMMA.** The overlap-identifiers can be determined in \( \sim m \) steps.

**Proof.** Suppose \( h(1), \ldots, h(j) \) have been determined and we want \( h(j+1) \) next. We know

\[
\begin{array}{c}
\ldots \ldots \quad y_j \\
\ldots \ldots \quad y_{j+1} \\
\ldots \ldots \quad y_{h(j)} \\
\end{array}
\]

and can put \( h(j+1) = h(j) + 1 \) in case \( y_{j+1} = y_{h(j) + 1} \), but otherwise we must back up further

\[
\begin{array}{c}
\ldots \ldots \quad y_j \\
\ldots \ldots \quad y_{j+1} \\
\ldots \ldots \quad y_{h(j)} \\
\ldots \ldots \quad y_{h(h(j))} \\
\ldots \\
\end{array}
\]

and further until such a match is found. The algorithm can be formulated as

\[
\begin{align*}
h(1) &:= 0; \\
j &:= 1; \\
\text{while } j < m \text{ do}
\end{align*}
\]
\[ i := h(j); \\
\textbf{while} \ y[j+1] \neq y[i+1] \ \\ \textbf{&} \ i > 0 \ \textbf{do} \\
\textbf{od}; \\
h(j+1) := \begin{cases} 
0 & \text{if } y[j+1] \neq y[i+1] \\
i + 1 & \text{else}
\end{cases} \; \\
j := j + 1 \\
\textbf{od};
\]

Consider the total cost of this algorithm as we keep sliding a copy of \( y \) across along \( y \) itself. As long as \( y[j+1] \) and \( y[i+1] \) do not match we can charge the cost for a step to position \( y[j-i+1] \) (and note here that \( i \) steadily decreases). If there is a match we charge the cost to \( y[j+1] \) and continue with an increased value of \( j \). No position can get charged more than twice. Thus the algorithm takes only a linear number of steps.

6.17. Once the overlap-identifiers are determined we can enact the on-line pattern-match across \( x \). The algorithm (see 6.15) becomes

\[
\begin{align*}
\text{match} & := 0; \\
\text{scan} & := 1; \\
\textbf{while} \ \text{scan} \leq n \ \textbf{do} \\
& \text{if } x[\text{scan}] = y[\text{match}+1] \ \text{then goto } L \ \text{fi;} \\
& \textbf{while} \ x[\text{scan}] \neq y[\text{match}+1] \ \textbf{do} \\
& \quad \text{match} := h(\text{match}) \\
& \quad \textbf{until} \ \text{match} = 0; \\
L: & \quad \text{match} := \begin{cases} 
0 & \text{if } x[\text{scan}] = y[\text{match}+1] \\
\text{match} + 1 & \text{else}
\end{cases} \\
& \quad \text{if } \text{match} = m \ \text{then pattern found & exit fi} \\
& \quad \text{scan} := \text{scan} + 1 \\
& \quad \textbf{od}; \\
\end{align*}
\]

By changing the "exit" into

\[
\begin{align*}
& \text{record the match at scan;} \\
& \quad \text{match} := h(\text{match}); \\
\end{align*}
\]

one can use the algorithm to locate all (possibly overlapping) occurrences of \( y \). Observe that each time around the loop we either match a next posi-
tion of $x$ or move the beginning of $y$ up, thus "pointing" at a position of $x$ for the second time (but we will never point at it later after we passed it). Thus the algorithm requires time bounded by $\sim 2n$.

Adding the work we obtain

**Theorem.** One can determine whether $y$ is a substring of $x$ in $\sim n+m$ steps.

6.18. Hirschberg [37] considered an interesting variant of the longest common substring problem, now known as the maximal common subsequence problem.

**Definition.** $z$ is a maximal common subsequence of $x$ and $y$ if and only if

(i) $z \leq x$ and $z \leq y$;
(ii) there is no $w$ with $|w| > |z|$ such that $w \leq x$ and $w \leq y$.

The problem typically occurs when a maximal correspondence between two data-arrays must be found, and is encountered in comparing molecular structures and for instance in problem-solving. Learning systems classify problems by a series of features in order of decreasing importance, and to find out to what extent two problems are similar (or "analogical") one tries to establish the best possible sequence of common features

```
  __________
 |         |
 |         |
 |         |
```

**Example.** A longest common subsequence of babaaba and bbaabba is bbaaba

```
   b a b a a b a
   b b a b a b a
```

The problem poses some interesting questions of time- and storage-utilization (the "simplicity" of the task is again deceptive). We shall prove Hirschberg's result (in a somewhat different presentation) that there is a data-organisation such that

**Theorem.** The maximal common subsequence of $x$ and $y$ can be found in $\sim n+m$ steps and $\sim n+m$ space.
6.19. If $z$ is to be a longest common sequence then for each position $i$ in $x$

\[ \begin{array}{cccccc}
\cdots & \ast & \ast & \ast & \ast & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
& x_i & x_{i+1} & \cdots & \cdots & x_{i+k} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
& z_1 & & & & z_2
\end{array} \]

where $z_1 \ast z_2 = z$

there must be a corresponding "split"-position $j$ in $y$

\[ \begin{array}{cccccc}
\cdots & \ast & \ast & \ast & \ast & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
& y_j & y_{j+1} & \cdots & \cdots & y_{j+k} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
& z_1 & & & & z_2
\end{array} \]

such that there is no position $k$ in $y$ for which the maximal common subsequence of

\[ x[1] \ldots x[i] \quad \text{and} \quad y[1] \ldots y[k] \]

concatenated with the maximal common subsequence of

\[ x[i+1] \ldots x[n] \quad \text{and} \quad y[k+1] \ldots y[m] \]

is longer than $z$.

This criterion immediately suggests an algorithm to find $z$. Split $x$

in two parts

\[ x[1] \ldots x[\lfloor \frac{n}{2} \rfloor] \quad \text{and} \quad x[\lfloor \frac{n}{2} \rfloor +1] \ldots x[n] \]

and find a position $j$ such that the longest common subsequences of

\[ x[1] \ldots x[\lfloor \frac{n}{2} \rfloor] \quad \text{and} \quad y[1] \ldots y[j] \]

and

\[ x[\lfloor \frac{n}{2} \rfloor +1] \ldots x[n] \quad \text{and} \quad y[j+1] \ldots y[m] \]

determined by a recursive application of the algorithm concatenate to a

longest possible string.

6.20. The common trick in dynamic programming learns to compute the value

of $j$ quickly first (without actually computing what the common subsequences
are), and then do a specific recursive call to do the hard work of finding
the subsequences.

The following concept is now needed.

**DEFINITION.** For $i \leq j$ and $k \leq \ell$ let $M_{i,k}^{[j,\ell]}$ be the length of the maximal
common subsequence of

$$x[i].....x[j] \; \text{and} \; y[k].....y[\ell].$$

**LEMMA.** One can compute $M_{i,k}^{[j,\ell]}$ in $\sim (j-i)(\ell-k)$ time and only $\sim (\ell-k)$
space.

**Proof.** One can determine $M_{i,k}^{[i,k]}, M_{i,k}^{[i,k+1]}, \ldots, M_{i,k}^{[i,\ell]}$ by comparing
symbol after symbol with $x[i]$ in $\sim \ell-k$ steps.

Assuming that $M_{i,k}^{[s-1,k]}, M_{i,k}^{[s-1,k+1]}, \ldots, M_{i,k}^{[s-1,\ell]}$ have been
determined, one can compute a next row with

$$
M_{i,k}^{[s,k]} := \begin{cases} 
1 & \text{if } y[k] \text{ occurs in } x[i] \ldots x[s] \text{ then } \text{else } 0; \\
& \text{for } t := k+1 \text{ to } \ell \text{ do} \\
& M_{i,k}^{[s,t]} := \begin{cases} 
1 & \text{if } x[s] = y[t] \text{ then } M_{i,k}^{[s-1,t-1]}+1 \\
& \text{else } \text{MAX}(M_{i,k}^{[s-1,t]}, M_{i,k}^{[s,t-1]}); \\
& \text{od};
\end{cases}
\end{cases}
$$

One must compute $j-i+1$ rows. \square

6.21. By building $M_{i,k}^{[j,\ell]}$ from "right-to-left" one can similarly compute
rows

$$M_{i,k}^{[j,\ell]}, M_{i,k+1}^{[j,\ell]}, \ldots, M_{i,\ell}^{[j,\ell]}$$
in the same time and space bounds.

6.22. We can now develop the recursive procedure (due to Hirschberg)
**FIND** $[i,j,k,\ell,a]$ determining $a$ as a longest common subsequence of

$$x[i].....x[j] \; \text{and} \; y[k].....y[\ell]$$
procedure FIND[i,j,k,ℓ,α]
begin
if i = j or k = ℓ then determine α directly fi;
    half := i + (j-i)/2;
compute row M_{1k}[half,k], M_{1k}[half,k+1], ..., M_{1k}[half,ℓ];
compute row M_{half+1,k}[j,ℓ], M_{half+1,k+1}[j,ℓ], ..., M_{half+1,ℓ}[j,ℓ];
split := an index k ≤ t ≤ ℓ such that
    M_{1k}[half,t] + M_{half+1,t+1}[j,ℓ]
    is maximal (over all choices of t);
    FIND[i,half,k,split,α₁];
    FIND[half+1,j,split+1,ℓ,α₂];
return α := α₁, α₂
end;

It is easy to make the procedure iterative and to see that only linear
space is needed.

In the time-analysis we shall use that for integers x,y ≥ 1 and
A ≥ 2B+C (A,B, and C positive)

A xy ≥ B(x+y) + C.

Assume as an induction-hypothesis that FIND[i,j,k,ℓ,α] requires only

α₁((j-1)(ℓ-k) + α₂[(j-i)+(ℓ-k)] + α₃

time.

Choosing α₂ > 1 and α₃ ≥ 1 appropriately it holds when i = j and/or
k = ℓ.

For the induction-step we shall assume (by 6.20 and 6.21) that the
computation of a row M_{1k}[j,ℓ] or M_{1k}[j,ℓ] takes

b₁((j-1)(ℓ-k) + b₂[(j-i)+(ℓ-k)] + b₃

time.

Then the time for FIND[i,j,k,ℓ,α] is bounded by

const₁ + 2b₁((j-1)(ℓ-k) + 2b₂[(j-i)+(ℓ-k)] + 2b₃ +
+ const₂(ℓ-k) + α₁[(j-i)+(ℓ-k)] + α₂[(j-i)+(ℓ-k)] + α₃ +
+ const₃ =
\[
\left(\frac{a_1}{2} + 2b_1\right)(j-i)(\ell-k) + (a_2 + 2b_2 + \text{const}_2)[(j-i) + (\ell-k)] + \\
+ (a_3 + 2b_3 + \text{const}_1 + \text{const}_3) \leq \\
\leq \left(\frac{a_1}{2} + 2b_1 + 2a_2 + 4b_2 + 2\text{const}_2 + a_3 + 2b_3 + \text{const}_1\right)(j-i)(\ell-k).
\]

If we choose \(a_1\) such that

\[a_1 \geq \frac{a_1}{2} + 2b_1 + 2a_2 + 4b_2 + a_3 + 2b_3 + \text{const}\]

(which we can) then the assumption is consistent. □

6.23. One may now ask if there is perhaps a faster than quadratic algorithm to find maximal common subsequences. Aho, Hirschberg, and Ullman [2] proved under weak assumptions that in general isn't.

Questions of optimality require that we somehow delimit the class of algorithms under consideration.

Consider straight-line programs which extract information from the input only by cross-comparisons and count how many queries

"is \(x[p] = y[q]\)"

are needed in worst case to find a longest common subsequence.

Note that if \(x^{(1)}\) and \(y^{(1)}\) and \(x^{(2)}\) and \(y^{(2)}\) yield the same answer query after query such algorithms generate the same computation in both instances and locate the longest common subsequence in identical manner. We shall exploit this fact to prove that if such algorithms query the input only a "few" times they cannot distinguish different inputs correctly.

6.24. We can immediately show

**Lemma.** Any algorithm for solving the longest common subsequence problem needs \(\geq n + m - 1\) queries in worst case.

**Proof.** Consider the sequence of queries (all yielding no) generated on

\[
\begin{align*}
x &= 0 0 \ldots 0 \\
n
\end{align*}
\quad\text{and}\quad
\begin{align*}
y &= 1 1 \ldots 1 \\
m
\end{align*}
\]

which have an "empty" solution.
Construct a bipartite graph $G$ connecting the positions queried in each step.

If $G$ has at least two connected components, then change the symbols of the positions of $x$ and $y$ in one component to 1 and 0 (respectively). The queries don't change and the algorithm still yields "empty" despite that there now is a non-trivial common subsequence.

It follows that $G$ must be a connected graph with $n + m$ vertices. □

6.25. For $c = 2$ (say alphabet $\Sigma = \{0,1\}$) the given bound cannot be improved. The positions of $x$ and $y$ can be identified in only $n + m - 1$ comparisons as follows.

Think of $x[1]$ as " 0 " and ask "is $x[1] = y[1]" to determine if we can think of $y[1]$ as " 0 " also or should consider it as " 1 ". Identify $x[2], \ldots, x[n]$ by asking "is $x[i] = y[i]", and $y[2], \ldots, y[m]$ by asking "is $x[1] = y[\ast]"

For $c > 2$ it doesn't work, and we can indeed improve the lemma to a quadratic bound.

**Theorem.** For $c > 2$ any algorithm for solving the longest common subsequence problem requires $\geq nm$ queries in worst case.

**Proof.** Construct the same graph $G$ as in 6.24, and suppose it has $< nm$ lines. Then there must be a pair $p,q$ not connected and thus not queried in the algorithm.

Change $x[p]$ and $y[q]$ into " 2 " (which we can because a third symbol is available). It does not affect the answer of any query asked and the algorithm still comes up with "empty" despite that the correct result must be a non-trivial string. □

**References** (some references are not explicitly cited in the text).


[16] BURKHARD, W.A., Associative retrieval trie hash-coding, Dept. of Applied Physics and Information Sc., Univ. of California/
San Diego, La Jolla (1975) TR-6, also: Proc. 8th Annual ACM

[17] CODD, E.F., A Relational model of data for large shared data banks,

[18] CODD, E.F., A data-base sublanguage founded on the relational cal-
culus, Proc. ACM SIGFIDET Workshop on data-description, access,
and control (1971).

[19] COFFMAN, E.G., and J. EVE, File-structures using hashing functions,

minimization, Dept. of Computer Sc., Pennsylvania State Univ.,

[21] COOK, S.A., Linear time simulation of deterministic two-way pushdown


[23] DE LA BRIANDAIS, R., File searching using variable length keys,

of combinatorial hashing, Stanford University, Stanford (1975)
STAN-CS-75-511.

in Automatic Progr. 7 (1972) 1-64.

[26] EVE, J., On computing the transitive closure of a relation, Stanford
University, Stanford (1975) STAN-CS-75-508.

[27] FISCHER, M.J., Efficiency of equivalence algorithms, in: R.E. MILLER,
and J.W. THATCHER (ed.), Complexity of Computer Computations,


[57] TARJAN, R.E., Applications of path-compression on balanced trees, Stanford University, Stanford (1975) STAN-CS-75-512.


OTHER TITLES IN THE SERIES MATHEMATICAL CENTRE TRACTS

A leaflet containing an order-form and abstracts of all publications mentioned below is available at the Mathematisch Centrum, Tweede Boerhaavestraat 49, Amsterdam-1005, The Netherlands. Orders should be sent to the same address.


MCT 7 W.R. VAN ZWET, Convex transformations of random variables, 1964. ISBN 90 6196 007 X.


MCT 13 H.A. LAUWERIER, Asymptotic expansions, 1966, out of print; replaced by MCT 54 and 67.


MCT 22 T.J. DEKKER, ALGOL 60 procedures in numerical algebra, part 1, 1968. ISBN 90 6196 029 0.


MCT 28 J. OOSTERHOF, Combination of one-sided statistical tests, 1969. ISBN 90 6196 041 X.


MCT 42 W. VERVAAT, Success epochs in Bernoulli trials (with applications in number theory), 1972. ISBN 90 6196 077 0.


MCT 65 J. DE VRIES, Topological transformation groups I: A categorical approach, 1975. ISBN 90 6196 113 0.

MCT 67  H.A. LAUWERIER, Asymptotic analysis, part 2.
ISBN 90 6196 119 X.

MCT 68  P.P.N. DE GROEN, Singularly perturbed differential operators of


MCT 70  W.P. DE ROEVER JR., Recursive program schemes: semantics and proof


MCT 72  J.K.M. JANSEN, Simple periodic and nonperiodic Lamb functions
ISBN 90 6196 130 0.

ISBN 90 6196 122 x.

MCT 74  H.J.J. TE RIELE, A theoretical and computational study of general-

MCT 75  A.E. BROUWER, Treelike spaces and related connected topological


MCT 77  W.C.M. KALLENBERG, Asymptotic optimality of likelihood ratio tests in

ISBN 90 6196 133 5.


MCT 80  P.W. HEMKER, A numerical study of stiff two-point boundary problems,

MCT 81  K.R. APT & J.W. DE BAKKER (eds), Foundations of computer science II,

MCT 82  K.R. APT & J.W. DE BAKKER (eds), Foundations of computer science II,

MCT 83  L.S. VAN BENTEM JUTTING, Checking Landau's "Grundlagen" in the
AUTOMATH system, ISBN 90 6196 147 5.

MCT 84  H.L.L. BUSARD, The translation of the elements of Euclid from the


MCT 87  S.G. VAN DER MEULEN & M. VELDHORST, Torrix II,
ISBN 90 6196 153 x.


* MCT 92 P.J. WEEDA, Finite generalized Markov programming, ISBN 90 6196 158 0.


* MCT 97 A. FEDERGHEM, Markovian control problems; functional equations and algorithms, . ISBN 90 6196 165 3.


AN ASTERISK BEFORE THE NUMBER MEANS "TO APPEAR"