Stanford Department of Computer Science June 1080
Report No. STAN-CS-80-794

RECENT DEVELOPMENTS IN THE COMPLEXITY OF COMBINATORIAL ALGORITHMS

by

Robert Endre Tarjan

Research sponsored by

National Science Foundation
and
Office of Naval Research

COMPUTER SCIENCE DEPARTMENT
Stanford University

Recent Developments in the Complexity of Combinatorial Algorithms

Robert Endre Tarjan
Computer Science Department
Stanford University
Stanford, California 94305

June, 1980

Abstract.

The last three years have witnessed several major advances in the area of combinatorial algorithms. These include improved algorithms for matrix multiplication and maximum network flow, a polynomial-time algorithm for linear programming, and steps toward a polynomial-time algorithm for graph isomorphism. This paper surveys these results and suggests directions for future research. Included is a discussion of recent work by the author and his students on dynamic dictionaries, network flow problems, and related questions.

Presented at the Fifth IBM Symposium on Foundations of Computer Science, Hakone, Japan, May 26-28, 1980.

This research was supported in part by National Science Foundation grant MCS-7826858 and by Office of Naval Research contract N00014-76-C-0330.

Recent Developments in the Complexity of **Combinatorial Algorithms**

Robert Endre **Tarjan**
Computer Science Department
Stanford University
Stanford, California 94305

1. Introduction.

In a 1978 paper **(Tarjan [1978]),** the author surveyed the then-current state of knowledge concerning combinatorial algorithms. Since the time that paper was published, researchers in the area have obtained several major new results. These include **a** sequence of more-and-more efficient algorithms for matrix multiplication, a similar sequence of algorithms for maximum network flow, a **polynomial-**time algorithm for linear programming, and a number of new approaches to graph isomorphism. In this paper we shall examine these recent advances, attempt to access their significance, and suggest directions for future research. The paper includes an examination of recent work by the author and his students on dynamic dictionaries, maximum flow problems, and related topics. Much of the work we shall describe is still unpublished.

Before beginning this survey, it is useful to review the general framework in which we shall view combinatorial algorithms and their complexity. We are interested in sequential algorithms; that is, in algorithms that perform only one step at a time. As a computer model, we use either **a** random-access machine with uniform cost measure **(Aho,** Hopcroft, and Ullman **[1974]),** or a pointer machine (Taijan **[1979]).** With either machine model we can perform a single arithmetic or logical operation in constant time, and we can store or retrieve a single piece of information in constant time.

The difference between random access machines and pointer machines lies in their memory organization. The memory of a random access machine consists of **a** one-dimensional array of cells, each capable of holding a single piece of information (such as a real number). The memory of **a** pointer machine is a linked structure consisting of a collection of nodes linked by pointers; each node consists of a' fixed finite number of fields, some of which are designated as containing pointers to other nodes. See Figure 1. Random **access** machines seem inherently more powerful than pointer machines, since we can perform arithmetic on array addresses but not on pointers. However, this difference in power amounts to at most a factor of log n in running time, and in fact the algorithms we shall discuss have the same asymptotic running times on either random-access machines or pointer machines.

[Figure 1]

As a complexity measure we use the worst-case running time of an algorithm as a function of the sire of its input. We shall ignore constant factors. For graph problems we shall use n, the number of vertices, and m, the number of edges, to measure the input size. For algorithms that manipulate real numbers, measuring input size is more problematic. The issue is whether we should allow rational numbers of arbitrary precision (or even irrational numbers) as input, and count any arithmetic operation as a single step, or whether we should measure the sise of real numbers by the number of bits needed to represent them, and count each bit operation as a single step. The former approach is truer to the spirit of algebraic complexity, and we shall adopt it in the case of matrix multiplication and maximum network flow. The latter approach is more reasonable if we are interested in issues of NP-completeness, and we shall adopt it in the case of linear programming.

The paper comprises 7 sections. Section 2 examines the recently discovered ellipsoid method for linear programming and its implications for combinatorial complexity. Section 3 discusses graph isomorphism. Section 4 surveys new work on matrix multiplication. Section 5 examines recent results on dynamic dictionaries, and Section 6 discusses network flow. Section 7 contains conclusions and remarks about promising directions for future research.

2. Linear Programming.

Garey and Johnson's book on NP-completeness (Garey and Johnson [1979]) concludes with a list of problems whose status with respect to NP-completeness was unknown at the time the book was published. A prominent problem on this list is linear programming, which can be defined as follows: given a set of linear inequalities $a_i x \leq b_i$ for $1 \leq i \leq m$, find a vector x that satisfies these inequalities and maximizes $cx = c_1x_1 + \cdots + c_nx_n$. Here $x = (x_1, \ldots, x_n)$ is a vector of n variables, each a_i is a vector of n real numbers, and the b_i's and c_j's are real numbers.

There is a well-known and empirically efficient algorithm for linear programming, Dantzig's simplex method (Dantzig [1951]), but this algorithm runs in exponential time on certain sets of input data (Klee and Minty [1972]). However, Khachiyan [1979] managed to prove that a completely different method based upon ideas of Shor [1970, 1977] solves the linear programming problem in polynomial time. The algorithm, which we shall call the ellipsoid method, is surprisingly simple, and in retrospect it is a wonder that it was not discovered many years previously.

The key idea of the method is to use binary search, maintaining an ellipsoid that bounds the set of solutions. We shall assume that the a_i's, b_i's and c_j's consist of integers, and that the size of the input is measured by the number of bits ℓ required to write down all these integers in binary. We note first that if the function cx has a finite maximum, then the value of x achieving the maximum has a number of bits polynomial in ℓ. If we add to our set of inequalities an inequality $cx \geq \alpha$, where α is a parameter, we can use binary search on α to determine the maximum value of a for which a feasible solution exists.

Thus we have reduced our original optimization problem to the problem of solving a polynomial number of feasibility problems of the following form: given a set of inequalities $a_i x \leq b_i$ for $1 \leq i \leq m$, is there some value of x that satisfies all the inequalities? By means of a second transformation (perturbing each b_i by a sufficiently small value) we can make all the inequalities strict.

Now we come to the heart of the matter. In order to search for feasible solutions, we construct an ellipsoid guaranteed to contain at least a certain volume of solutions if there are any solutions at all. It suffices to choose a sphere centered at the origin with sufficiently large (but polynomially bounded) radius. We then repeatedly reduce the size of the bounding ellipsoid until either we find a solution or the bounding ellipsoid is so small that it can't contain any solutions, and thus there are no solutions.

To carry out a general step of this reduction process, we test the center x of the ellipsoid for feasibility. If x satisfies all the inequalities, we are done. If not, we find a violated inequality $a_i x \geq b_i$. We then construct a new ellipsoid

containing those points in the old ellipsoid that satisfy $a_i x < b_i$. See Figure 2. The crucial point is that the volume of the new ellipsoid is smaller by a constant factor (depending polynomially on n) than the volume of the old ellipsoid; thus after a polynomial number of steps either we find a feasible solution or we can terminate the process **and** declare that no solutions exist. For further details of the algorithm, see **Gacs** and **Lovász [1979]** or Aspvall and Stone **[1979]**.

[Figure **2]**

Khachiyan's discovery has led to an explosion of research on related issues. Of primary interest are the questions, **"Is** the algorithm **efficient** (or can it be made so)?" and "What implications does the algorithm have for other combinatorial problems?" As a tentative answer to the first question, it seems that the ellipsoid method is not competitive in practice with the simplex method, although this matter deserves further study.

The theoretical implications of the ellipsoid method are more interesting. The algorithm does not depend heavily on linearity, only on the convexity of the solution space. Indeed, **Kozlov,** Tarasov, and Khaciyan **[1979]** have extended the algorithm to convex quadratic programming. On the other hand, the linear case is the most interesting from the point of view of combinatorics. A standard technique of operations researchers is to use linear programming to attack integer programming problems. The idea is to **define** an integer programming problem as a linear programming problem with a large (possibly exponential) number of inequalities, in such a way the inequalities defining the problem are easy to generate. Karp and Papadimitriou **[1979]** have proved a negative result for this approach: any NP-complete combinatorial optimization problem cannot be **poly-**nomially characterized by a set of linear inequalities, unless NP = co-NP. By **"polynomially** characterized" we mean that the set of inequalities is in NP; that is, given an inequality that. is in the set we can construct a polynomial-length proof of this fact. Since it is unlikely that NP = co-NP, this means that any combinatorial optimization problem that can be polynomially characterized by inequalities is unlikely to be NP-complete.

In order to run the ellipsoid method, we do not need an explicit listing of all the inequalities defining the problem but only a **way** to test whether a point is feasible and to generate a violated inequality if it is not. Suppose we have a combinatorial opimization problem for which it is possible to test feasibility and generate a violated inequality in polynomial time. Then by means of Khachiyan's algorithm we can solve the combinatorial optimization problem in polynomial time (if certain other weak conditions are satisfied). Karp and Papadimitriou and independently **Grötschel, Lovász,** and Schrijver **[1980]** made this **observation,**

which implies that NP-complete problems are unlikely to have polynomial time generators of violated inequalities. The result also means that the ellipsoid method is in some sense a universal method for combinatorial optimization problems, and it can be used to generate new fast algorithms. For instance, Grötschel, Lovász, and Schrijver have derived a polynomial-time algorithm for vertex packing in perfect graphs. This area appears to be ripe for further work. For an excellent non-technical discussion of Khachiyan's algorithm and its implication for combinatorial optimization, see Lovász [1980].

3. Graph Isomorphism.

Another of Garey and Johnson's open problems is graph isomorphism: given two undirected graphs G_1 and G_2, determine whether there is a one-to-one map ping of the vertices of G_1 onto the vertices of G_2 that preserves adjacency. This problem has a number of applications, especially in the **cataloguing** of chemical molecules. Although no one has yet discovered a polynomial-time algorithm for graph isomorphism, much progress has been made recently.

Most of the early algorithms for graph isomorphism (see for instance Corneil and Gotlieb [1970], Read and Corneil [1977]) combine backtrack search with a partition refinement method to reduce the size of the search space. Such a method is very efficient in practice but requires exponential time on some highly symmetric graphs, because the refinement scheme becomes useless. For various **special** classes of graphs, efficient algorithms are known. These classes include trees (Aho, Hopcroft, and Ullman [1974]), planar graphs (Hopcroft and Tarjan [1972], Hopcroft and Wong [1974]), series-parallel graphs (Valdes, Tarjan, and Lawler [1979]), and interval graphs (Colbourn and Booth [1979]), for which **linear-time** algorithms exist. The tree isomorphism algorithm depends upon clever use of lexicographic sorting. The algorithms for series-parallel graphs and **interval** graphs are straightforward extensions of the algorithm for trees. The algorithm for planar graphs combines the tree isomorphism algorithm with a linear-time decomposition into triconnected components (Hopcroft and Tarjan [1973]), and a reduction method for triconnected embedded planar graphs. The method is fast for three reasons: the relationship among triconnected components can be represented by a decomposition tree, a triconnected planar graph has only two planar embeddings, and there is a linear-time algorithm for embedding a planar graph (Hopcroft and Tarjan [1974]). Recently Lichtenstein [1980] has **discovered** a polynomial-time isomorphism algorithm for graphs embeddable in a projective plane, and Miller [1980] and Filotti and Mayer [1980] have found a **polynomial-time** isomorphism algorithm for graphs of any fixed genus. These methods combine fast embedding algorithms (Filotti, Miller, and Reif [1979]) with a careful analysis of the ways a graph can be embedded in a surface of the appropriate genus.

Other researchers have attempted to use degree constraints or symmetry properties to aid in testing isomorphism. Miller [1977] showed that isomorphism is in **co-NP** for arc-transitive trivalent graphs; that is, if two such graphs are non-isomorphic, thre is a polynomial-length proof of this fact. Lipton [1980] discovered an $n^{O(\log n)}$ -time isomorphism algorithm for arc-transitive **trivalent** graphs, and Babai [1980] found an $n^{O(\sqrt{n} \log n)}$ -time algorithm for strongly regular graphs. The major breakthrough was Babai's discovery (Babai [1979]) of **a** random

polynomial-time algorithm for vertex-colored graphs with bounded color multiplicities. Babai's work is important much more for his techniques than for the specific results to be obtained. He was able to use properties of the automorphism group of a graph, describing this group by means of a tower of subgroups, in such a way that one could quickly construct the tower. This work was extended by Hoffman [1980], who found a random $n^{O(\log n)}$-time algorithm for a class of graphs called "cone graphs". Furst, Hopcroft, and Luks (private communication) made Babai's methods deterministic and generalized Hoffman's algorithm to test isomorphism of trivalent graphs deterministically in $n^{O(\log n)}$ time. Luks [1980] reduced the running time of the algorithm for trivalent graphs to polynomial and found an $n^{O(\log n)}$-time algorithm for graphs of any fixed valence.

Extrapolating this work, it seems likely that a polynomial-time isomorphism algorithm for graphs of any fixed degree will soon be discovered. Whether this will lead to a polynomial-time algorithm for the general problem is less clear. Much of the recent work on isomorphism depends heavily on properties of finite groups, and a complete solution may require new results in group theory or at least ingenious use of old ones.

4. **Matrix Multiplication.**

Not all research in combinatorial complexity has been devoted to finding polynomial-time algorithms for problems not known to have them. Much effort has also been devoted to finding faster algorithms for problems already known to be solvable in polynomial time. For example, let us examine recent progress on matrix multiplication. See Table 1. The classical algorithm for this problem multiplies two $n \times n$ matrices in $O(n^3)$ time. Strassen [1969] discovered a way to multiply two 2×2 matrices with only seven multiplications, and using this observation constructed a recursive algorithm to multiply $n \times n$ matrices in $O(n^{\log_2 7})$ time.

[Table 1]

After a hiatus of nine years, Pan [1978] found a slightly faster ($O(n^{2.795})$ vs. $O(n^{2.807})$) method. Pan used an ingenious and complicated technique called "aggregating, uniting, and cancelling." A key point in Pan's approach was to use fairly large size matrices ($n = 70$) as the basis for the recursion, rathr than very small matrices ($n = 2$ or 3 or 4). Shortly after Pan's result appeared, Bini, Capovani, Romani, and Lotti [1979] introduced a notion of *approximate" matrix multiplication, and produced an $O(n^{2.78})$ -time algorithm to multiply matrices in this approximate sense. Schönhage [1979] showed that any approximate matrix multiplication algorithm can be converted into an exact one, and he further showed that any method for multiplying sparse matrices can be converted into a method to multiply dense ones. The culmination of Schönhage's advances was an $O(n^{2.6088})$ -time algorithm. Subsequent work by Pan, Schönhage, and Winograd, using a combination of these techniques, has produced a sequence of faster-and-faster algorithms; the best bound currently claimed is $O(n^{2.49+})$, due to Pan (private communication). Pan has recently written a long paper (Pan [1980]) surveying these developments. It may be possible to multiply matrices in $O(n^{2+\epsilon})$ time for any positive ϵ; certainly the recent results lead in this direction.

5. Dynamic Dictionaries.

Fast algorithms require the use of appropriate data structures, some of which are quite complicated. In this section we shall examine efficient ways to represent one important type of data structure, called a dynamic dictionary. A dynamic dictionary consists of a collection items, each with an associated key. We assume that the keys are totally ordered and can be compared; we further assume that no two keys are the same. We are interested in performing the following kinds of operations on dynamic dictionaries:

(a) Given a key, access the item (if any) with this key.
(b) Insert a new item in the dictionary.
(c) Delete a given item from the dictionary.
(d) Merge two dictionaries into a single dictionary.
(e) Concatenate two dictionaries, such that all keys in one dictionary are smaller than all keys in the other.
(f) Split a dictionary on a given key into a dictionary containing all keys no larger than the given key and a dictionary containing all keys larger than the given key.

Dynamic dictionaries have widespread uses in computer science; see Knuth [1973]. There are a number of ways to represent such dictionaries. If only accesses and insertions are to be performed, a hash table (Knuth [1973]) can be used. Hash tables allow accesses and insertions in O(1) time on the average, although the worst-case time is O(n), where n is the number of items in the dictionary. Hash tables can be adapted to allow efficient deletion, but not merging, concatenation, or splitting; such tables do not maintain the ordering information needed to efficiently merge, concatenate, or split. A balanced tree structure, such as a 2 — 3 tree (Aho, Hopcroft, and Ullman [1974]) or more generally a B-tree (Bayer and McCreight [1972]), a height-balanced tree (Knuth [1973]), or a weight-balanced tree (Reingold, Nievergelt, and Deo [1974]), is appropriate if such operations are to be performed. Such a structure allows access, insertion, deletion, concatenation; and splitting in $O(\log n)$ time (see Aho, Hopcroft, and Ullman [1974] for instance).

There are a number of recent results on dynamic dictionaries. Brown and Tarjan [1979] showed how to merge two dictionaries represented as height-balanced trees in $O(m \log \frac{n}{m})$ time, if the smaller dictionary has m items and th larger one has n items. This result also holds for B-trees and for weight-balanced trees. Brown and Tarjan [1980] showed how to maintain 'fingers" into 2-3 trees so that access is very fast on the vicinity of a finger. Their proposed structure supports fast access, finger creation, insertion, and deletion, as long as the insertions and

deletions occur in separate parts of the tree. Huddleston (private communication), Maier and Salveter [1979], and Mehlhorn [1979b] independently used less balanced versions of B-trees to extend Brown and Tarjan's results so that arbitrary insertions and deletions are fast.

An interesting question is what happens if we want to bias the dictionary so that certain items are easier to access than others. This is desirable, for instance, in keyword tables for compilers and in language dictionaries, where some words are accessed much more often than others. To study this question we assume that each item i has a *weight* w_i, and that we wish to minimize the sum of the weighted access times $\sum_{i=1}^{n} w_i t_i$, where t_i is the access time of item i. Knuth [1971] and Hu and Tucker [1971] have proposed efficient algorithms to construct *optimum* binary search trees; Knuth's algorithm requires $O(n^2)$ time but allows items to be stored in internal nodes of the tree; Hu and Tucker's algorithm uses $O(n \log n)$ time but requires that all the items be in external tree nodes. Garsia and Wachs [1977] have given an interesting variant of the Hu-Tucker algorithm.

Optimum binary search trees are not suitable if insertions and deletions are to be performed, because they require too much time to update. Mehlhorn [1978, 1979a] has investigated the question of dynamically maintaining an almost optimum tree. An entropy argument shows that the sum of the weighted access times in an optimum tree is bounded below by a constant times $\sum_{i=1}^{n} w_i \log \frac{w}{w_i}$, where $w = \sum_{i=1}^{n} w_i$; thus the goal is to maintain a tree in which each item i has $0(1 + \log \frac{w}{w_i})$ access time. Mehlhorn has described a complicated version of weight-balanced trees with the following properties:

(a) $0(1 + \log \frac{w}{w_i})$ time to access, insert, or delete item i;

(b) $O(1 + \log \frac{\max(w, w')}{\min(w_i, w_i')})$ time to change the weight of item i from w_i to w_i', where w is the total weight before the change and w' is the total weight after the change.

Bent, Sleator, and Tarjan [1980] have found a way to implement dynamic dictionaries that is not only much simpler than Mehlhorn's but allows fast concatenation and splitting; specifically, $0(1 + \log \frac{w}{w'})$ time to concatenate dictionaries of total weights w and w' with $w \geq w'$, and $0(1 + \log \frac{w}{w_i})$ time to split a dictionary at item i. The data structure resembles a $2 - 3$ tree. Each node in the tree has between zero and three children. Certain nodes contain items; others are non-item nodes (an item node contains exactly one item). A symmetric-order traversal of the tree visits the items in order by key. An item node has a left son and a right son, either or both of which can be missing; a non-item node has either two or three children. In addition, each node has a level, defined as follows: the level of a node containing item i is $\lfloor \log w_i \rfloor$; the level of a non-item node is one

greater than the minimum of the levels of its children. We impose the additional requirement that the level of a node be strictly greater than the levels of all its children; thus all children of a non-item node have the same level. Figure 3 gives an example of such a tree.

[Figure 3]

It is not hard to implement access, insertion, deletion, concatenation, splitting, and weight change on such trees. Analyzing the efficiency of these operations, however, requires a clever accounting argment. The data structure has not only the obvious applications but also a number of not-so-obvious ones, as we shall see in the next section.

8. Maximum Network Flow.

An important special case of linear programming is the maximum network flow ***problem:*** given a directed graph G = ***(V,*** E), two distinguished vertices, **a** ***source*** s and a sink t, and a non-negative capacity c(e) on each edge e, find **a** flow of maximum value from s to t. A flow is defined by a value f(e) on each edge e, such that $0 \leq f(e) \leq c(e)$; for every vertex u except the source and sink, the total flow on edges entering u must equal the total flow on edges leaving v. The ***value*** of the flow is the total flow on edges leaving the source (or equivalently on edges entering the sink).

Ford and Fulkerson [1962] were the first to study this problem. They proved the famous ***max-flow*** min-cut theorem, which states that the value of a maximum flow equals the capacity of **a** minimum cut. (A cut is a partition $X, \overline{X}$ of the vertices such that $s \in X$ and $t \in \overline{X}$; the capacity of the cut is the total capacity of edges leaving X and entering $\overline{X}$.) They proved this theorem by means of an ***augmenting path method*** which, given **a** flow, attempts to **find** a path from s to t along which the flow can be increased. If the algorithm finds such a path, the flow value is increased appropriately. If not, the method locates a cut whose capacity is equal to the value of the current flow.

Ford and Fulkerson's method does not automatically give a fast algorithm for maximum network flow. If the capacities are large integers, the method can require enormous amounts of time; if the capacities are irrational, the method need not terminate. However, if the search for augmenting paths is systematic, the method leads to a fast algorithm. Table 2 shows the running times of various maximum network flow algorithms based on this idea.

[Table 2]

Karp and Edmonds [1972] were the first to give a polynomial-time algorithm for maximum flow. They showed that if a shortest augmenting path is always selected, then no more than $O(nm)$ augmentations take place. From this they obtained an $O(nm^2)$-time algorithm. Independently Dinic [1970] made the same observation, and further noted that all the augmenting paths of a given length can be found at once, in $O(nm)$ time, giving an overall bound of $O(n^2m)$. All the recent progress on maximum flow is based on Dinic's work.

Karzanov [1974] improved Dinic's running time to $O(n^3)$ by discovering how to find all augmenting paths of a given length in $O(n^2)$ time. Karzanov's algorithm is quite complicated, but Malhotra, Kumar, and Maheshwari [1978] obtained **a** very simple algorithm that achieves the same time bound. Cherkasky [1977] discovered an $O(n^2m^{1/2})$ algorithm, improved by Galil [1978] to $O(n^{5/3}m^{2/3})$. Galil and Naamad [1979] and independently Shiloach [1978] found an $O(nm(\log n)^2)$

algorithm, which Sleator and Tarjan [1980] improved to $O(nm \log n)$.

The Sleator-Tarjan algorithm obtains its speed by using sophisticated data structures that maintain the flow information implicitly; thus it is not necessary to perform an augmentation by changing the flow in every edge of the augmenting path. The general method is illustrated in Figure 4. The algorithm maintains a tree of edges with residual capacity whose root is the sink. The path from the source to the sink in this tree defines an augmenting path. An augmentation is performed on this path, saturating at least one edge and causing the tree to break into at least two edges. The tree is reassembled by adding new edges with residual capacity, and the process is repeated.

[Figure 4]

[Figure 5]

To represent the tree, the algorithm decomposes it into paths, as in Figure 5. Before an augmentation is performed, the paths representing the tree are rearranged, by splitting and concatenation, so that the source and sink are on the same path. Then the augmentation proceeds. Galil and Naamad [1979] and Shiloach [1979] showed that only O(logn) splits and concatenations occur per augmentation. By representing each path of the tree by a data structure consisting of a balanced binary tree, they obtained an O(log n) bound per split or concatenation, an $O((\log n)^2)$ bound per augmentation, and an $O(nm(\log n)^2)$ bound overall. By representing each path by a dynamic dictionary implemented as described in Section 5, Sleator and Tarjan were able to reduce the time per augmentation to $O(\log n)$, saving a factor of logn in the overall running time. This algorithm seems hard to beat; further improvements in maximum flow may require a basic approach different from and more powerful than Dinic's.

7. Remarks.

What are we to conclude from all these new results in combinatorial complexity? First, it is clear that NP-completeness is **a** very powerful and precise tool for classifying combinatorial problems; it seems that any natural problem is either NP-hard or has a polynomial-time algorithm. The candidates for **counterexamples**, such as linear programming and graph isomorphism, are yielding to diligent attack. Thus the $P = NP$ question becomes, if anything, even more important. In general, it seems that the lack of a non-trivial lower bound for a problem is a good reason to believe that faster algorithms exist for it.

Second, some polynomial-time algorithms, such as the sophisticated algorithms for network flow, show promise of being quite practical. Others, such as the fastest methods for matrix multiplication, are only asymptotic results and seem to hold no implications for practice. In order to detect such differences, much more study is needed of algorithmic overhead, the associated constant factors, and the practical trade-offs between algorithms.

Third, the careful and systematic study of data structures is extremely important in the design of algorithms that are fast both in theory and in practice. In particular, there is much to be learned about the properties of various kinds of trees and their use **as** data structures. We still lack an adequate theory that will fit the appropriate data structure to each problem we wish to solve.

Fourth, the advent of very-large-scale integrated circuits has raised entirely new questions for combinatorial complexity. We are faced with the problems of designing new models of complexity matched to the new hardware, and of discovering what part of the knowledge obtained for sequential algorithms will translate into the new framework, which will incorporate large-scale concurrency. Thus parallel algorithms and spatial layout problems are important topics for future research.

References

A. V. Aho, J. E. Hopcroft, and J. D. Ullman [1974]. *The* Design *and* Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass.

B. Aspvall and R. E. Stone [1979]. "Khachiyan's linear programming algorithm," Report STAN-CS-79-726, Computer Science Department, Stanford University, Stanford, California.

L. Babai [1979]. *Monte-Carlo algorithms in graph isomorphism testing," unpublished manuscript.

L. Babai [1980]. "On the complexity of canonical labeling of strongly regular graphs," SIAM Journal on Computing, to appear.

R. Bayer and E. McCreight [1972]. 'Organization and maintenance of large ordered indexes," *Acta Informatica,* 173-189.

S. Bent, D. Sleator, and R. E. Tarjan, "Biased 2—3 trees," submitted to Twenty-first Annual Symposium on Foundations of Computer Science.

D. Bini, M. Capovani, F. Romani, and G. Lotti [1979]. "$O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication," Information Processing Letters 8,234–235.

M. R. Brown and R. E. Tarjan [1979]. "A fast merging algorithm," Journal ACM 26, 211-226.

M. R. Brown and R. E. Tarjan [1980]. "Design and analysis of a data structure for representing sorted lists," SIAM Journal *on* Computing, to appear.

B. V. Cherkasky [1977]. "Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations," Mathematical Methods of Solution of *Economical Problems 7,* 117-125.

C.-J. Colbourn and K. S. Booth [1979]. 'Linear time automorphism algorithms for trees, interval graphs, and planar graphs," unpublished manuscript.

D. G. Corneil and C. C. Gotlieb [1970]. "An efficient algorithm for graph isomorphism," Journal ACM 17, 51-64.

G. B. Dantzig [1951]. "Maximization of a linear function of variables subject to linear inequalities," Activity Analysis of Production and *Allocation,* T. C. Koopmans, ed., Wiley, 339-347.

E. A. Dinic [1970]. "Algorithm for solution of a problem of maximal flow in a network with power estimation," Soviet Math. *Dokl.* 11, 1277–1280.

J. Edmonds and R. M. Karp [1972]. 'Theoretical improvements in algorithmic efficiency for network flow problems," *Journal* ACM 19, 248-264.

I. S. Filotti, G. L. Miller, and J. H. Reif [1979]. "On determining the genus of a graph in $O(v^{O(g)})$ steps," *Proc. Eleventh* Annual ACM Symposium on *Theory* of Computing, 27-37.

I. S. Filotti and J. N. Mayer [1980]. "A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus," *Proc. Twelfth Annual ACM* Symposium on Theory of Computing, 236-243.

I. S. Filotti, G. L. Miller, and J. H. Reif [1979]. "On determining the genus of a graph in $O(v^{O(g)})$ steps," *Proc. Eleventh* Annual ACM *Symposium on Theory of* Computing, 27-37.

L. R. Ford and D. R. Fulkerson [1962]. *Flows* in Networks, Princeton University Press, Princeton, New Jersey.

P. Gács and L. Lovász [1979]. "Khachian's algorithm for linear programming," Report STAN-CS-79-750, Computer Science Department, Stanford University, Stanford, California.

Z. Galil [1978]. "A new algorithm for the maximal flow problem," *Proc.* Nineteenth Annual Symposium on Foundations of Computer Science, 231-245.

Z. Galil and A. Naamad [1979]. 'Network flow and generalized path compression," *Proc.* Eleventh Annual ACM *Symposium* on *Theory* of Computing, 13–*26.*

M. R. Garey and D. S. Johnson [1979]. Computers and *Intractibility: A Guide to* the *Theory* of *NP-Completeness,* W. H. Freeman, San Francisco.

A. M. Garsia and M. L. Wachs [1977]. "A new algorithm for minimum cost binary trees," SIAM Journal *on* Computing 6, 622-642.

M. Grötschel, L. Lovász, and A, Schrijver [1980]. "The ellipsoid method and its consequences in combinatorial optimization," Report No. 80151-OR, Institut für Ökonometrie und Operations Research, Universität Bonn.

C. M. Hoffman [1980]. "A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus," *Proc.* Twelfth Annual *ACM Symposium on* Theory of Computing, 244-251.

J. E. Hopcroft and R. E. Tarjan [1972]. "Isomorphism of planar graphs," *Complexity of* Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 131-152.

J. E. Hopcroft and R. E. Tarjan [1973]. "Dividing a graph into triconnected components," SIAM Journal *on* Computing 2, 135-158.

J. E. Hopcroft and J. K. Wong [1974]. 'Linear time algorithm for isomorphism of planar graphs," *Proc.* Sixth Annual ACM Symposium *on Theory of Computing,* 172–184.

T. C. Hu and C. Tucker [1971]. "Optimum computer search trees," *SIAM* Journal *on* Applied Mathematics 21, 514-532.

R. M. Karp and C. H. Papadimitriou [1980]. "On linear characterizations of combinatorial optimization problems," TM-154, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass.

A. V. Karzanov [1974]. *Determining the maximal flow in a network by the method of preflows," Soviet Math. *Dokl.* 15, 434-437.

L. G. Khachiyan [1979]. "A polynomial algorithm for linear programming," Doklady Akademia Nauk *USSR 244,* 191-194.

V. Klee and G. Minty [1972]. "How good is the simplex algorithm?", *Inequalities III,* 0. Shisha, ed., Academic Press, 159-175.

D. E. Knuth [1971]. "Optimum binary search trees," Acts *Informatica* 1, 14-25.

D. E. Knuth [1973]. *The Art of Computer Programming,* Volume *3: Sorting and Searching,* Addison-Wesley, Reading, Mass.

M. Z. Kozlov, S. P. Tarasov, and L. G. Hacijan [1979]. 'Polynomial solvability of convex quadratic programming," Soviet *Math. Dokl. 20.*

D. Lichtenstein [1980]. "Isomorphism for graphs embeddable on the projective plane," *Proc. Twelfth Annual ACM Symposium on Theory of Computing,* 218-224,

, R. J. Lipton [1980]. "The beacon set approach to graph isomorphism," SLAM Journal *on Computing,* to appear.

L. Lovász [1980]. "A new linear programming algorithm – better or worse than the simplex method?", *Math. Intelligencer, to* appear.

E. M. Luks [1980]. "A polynomial-time algorithm for testing isomorphism of trivalent graphs," submitted to Twenty-first Annual ACM Symposium on Foundations of Computer Science.

D. Maier and S. C. Salveter [1979]. *Hysterical B-trees," Technical Report No. 79/007, Department of Computer Science, State University of New York at Stony Brook.

V. M. Malhotra, M. P. Kumar, and S. N. Maheshwari [1978]. "An $O(V^3)$ algorithm for finding the maximum flows in networks," Information Processing Letters 7, 277-278.

K. Mehlhorn [1978]. "Arbitrary weight changes in dynamic trees," Bericht 78/04, Fachbereich 10 – Angewandte Mathematik und Informatik, Universität des Saarlandes, Saarbriicken.

K. Mehlhorn [1979a]. 'Dynamic binary search," SIAM Journal on Computing 8, 175-198.

K. Mehlhorn [1979b]. "A new data structure for representing sorted lists," Fachbereich 10 – Informatik, Universiät des Saarlandes, Saarbriicken.

G. L. Miller [1977]. "Graph isomorphism, general remarks," Proc. Ninth Annual ACM Symposium on Theory of Computing, 143–150.

G. L. Miller [1980]. "Isomorphism testing for graphs of bounded genus," Proc. Twelfth Annual ACM Symposium on Theory of Computing, 225-235.

V. Y. Pan [1978]. "Strassen's algorithm is not optimal," Proc. Nineteenth Annual Symposium on Foundations of Computer Science, 166476.

V. Y. Pan [1980]. "New combinations of methods for the acceleration of matrix multiplication," unpublished manuscript.

R. E. Read and D. G. Corneil [1977]. "The graph isomorphism disease,'@ Journal of Graph Theory 1, 339-363.

E. M. Reingold, J. Nievergelt, and N. Deo [1974]. Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Inc., Englewood Cliffs, N.J.

A. Schönhage [1979]. "Partial and total matrix multiplication," Technical Report, Mathematisches Institut, Universität Tübingen.

Y. Shiloach [1978]. "An $O(nI \log^2 I)$ maximum-flow algorithm," Technical Report STAN-CS-78-702, Computer Science Department, Stanford University, Stanford, California.

N. Z. Shor [1970]. *Convergence rate of the gradient descent method with dilation of the space," *Kibernetika* 2, 80–85.

N. Z. Shor [1977]. "Cut-off method with space extension in convex programming problems," *Kibernetika* 13, 94-95.

D. Sleator and R. E. Tarjan [1980]. "An $O(nm \log n)$ algorithm for maximum network flows," submitted to Twenty-first Annual Symposium *on* Foundations **of** *Computer Science.*

V. Strassen [1969]. "Gaussian elimination is not optimal," Num. **Math. 13, 154-356.**

R. E. Tarjan [1978]. *Complexity of combinatorial algorithms," SIAM Review **20,** 457-491.

R. E. Tarjan [1979]. "A class of algorithms which require nonlinear time to **main**tain disjoint sets," *J. Computer* and *System Sciences* 18, 110427.

J. Valdes, R. E. Tarjan, and E. L. Lawler [1979]. "The recognition of **series** parallel digraphs," *Proc. Eleventh* **ACM** *Symposium on Theory of* **Computing,** 1-12.

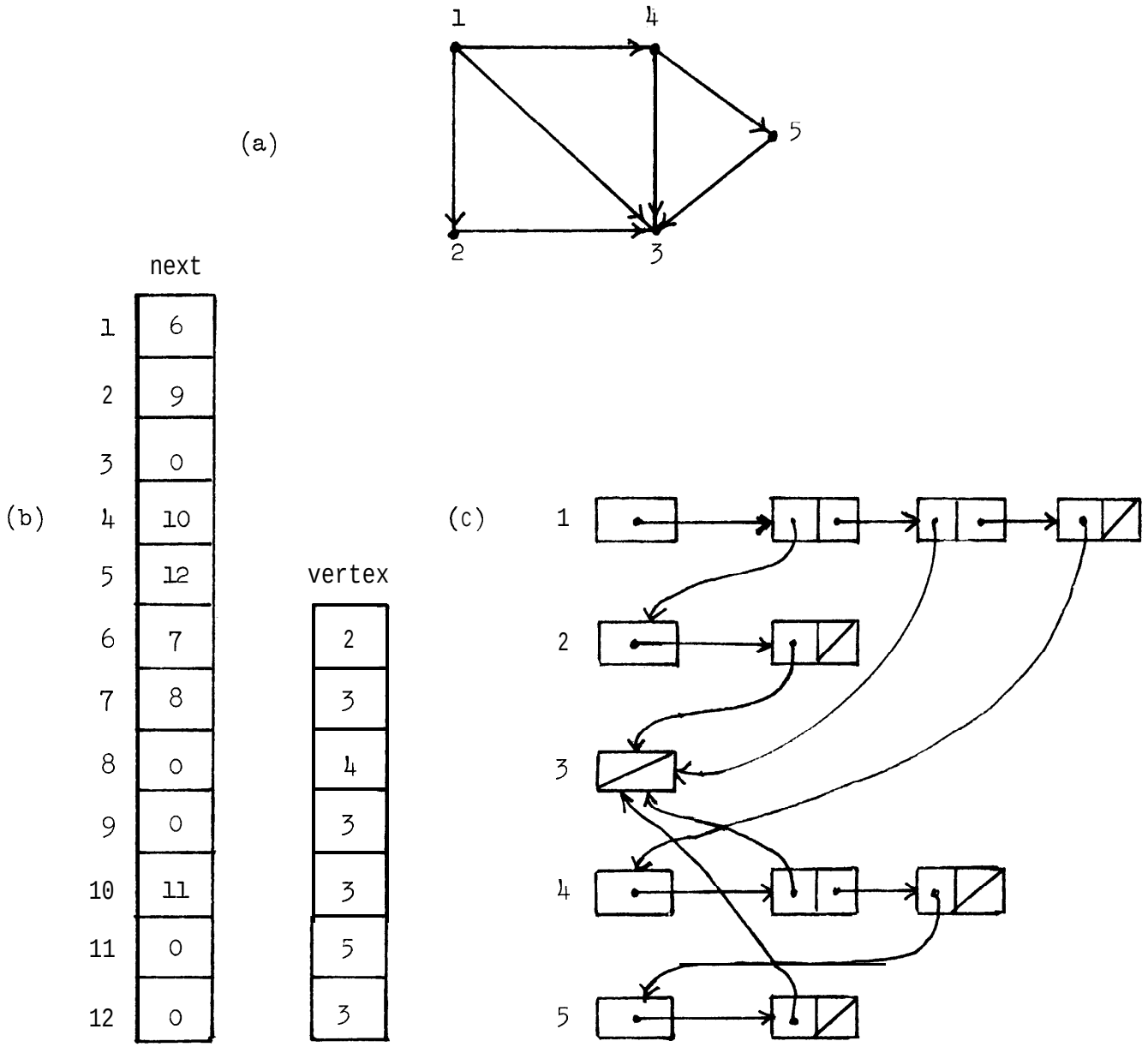

Figure 1. A directed graph and its computer representation.

(a) Graph.

(b) Representation by two arrays.

(c) Representation by a linked structure.

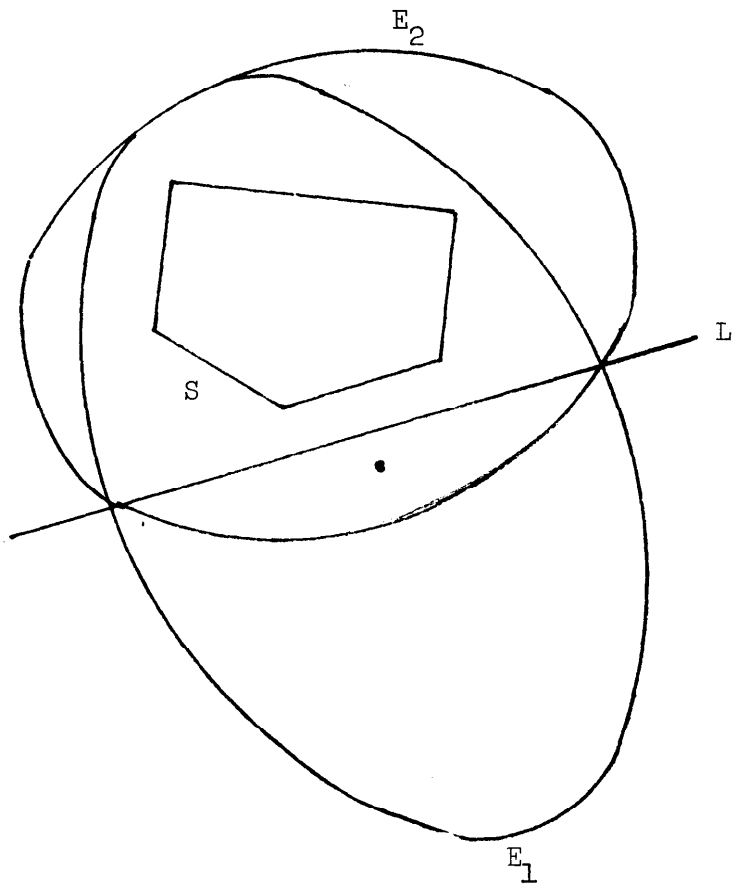

Figure 2. One step of the ellipsoid method,

S = solution space.

E_1 = old bounding elipsoid.

L = inequality violated by center of E_1 .

E_2 = new bounding ellipsoid.

Word	Relative frequency	Relative level
a	5074	2
and	7638	2
as	1853	0
for	1869	0
I	2272	1
in	4312	2
is	2509	1
it	2255	1
of	9767	3
that	3017	1
the	15568	3
to	5739	2

Figure 3(a). Relative frequencies of the twelve most common words. Relative level is level minus ten.

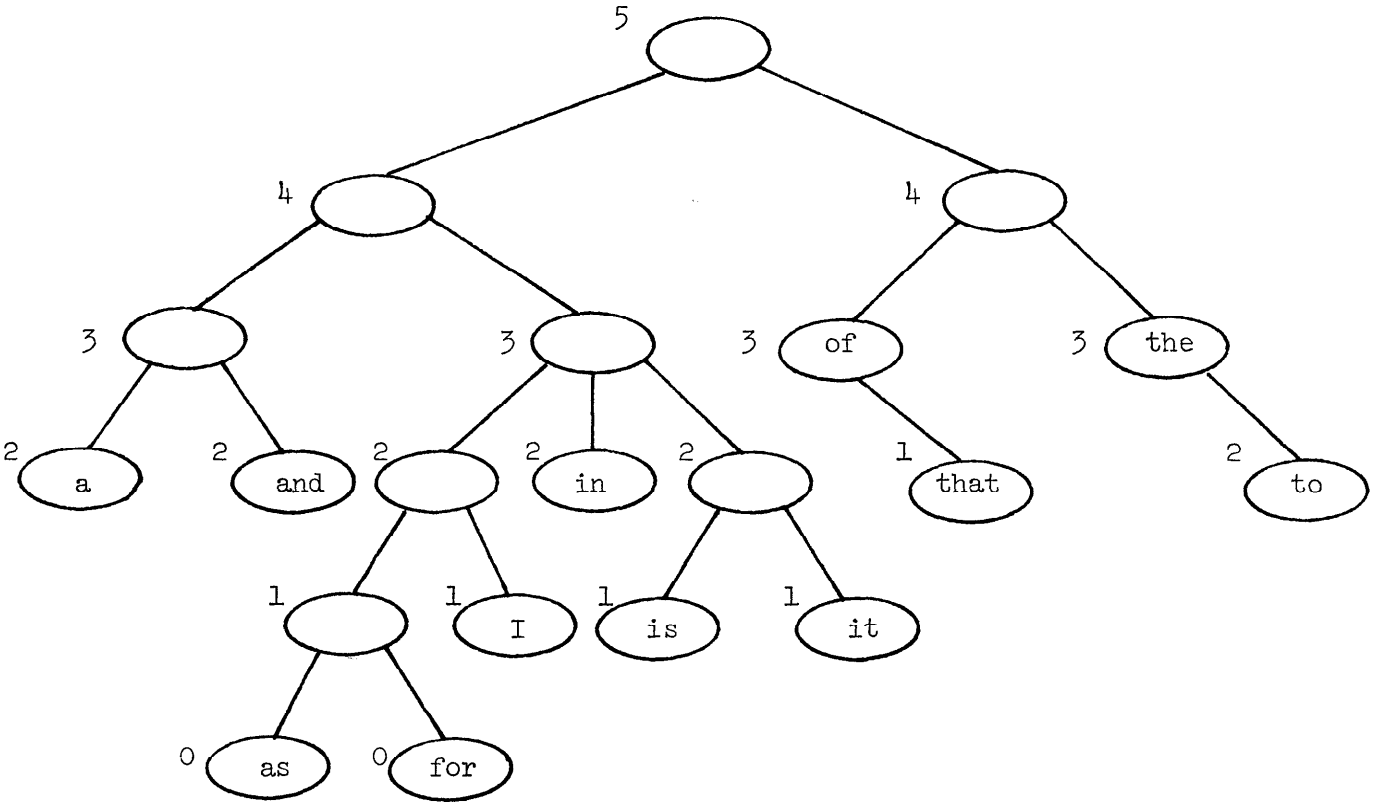

Figure 3(b). Biased 2-3 tree containing the words in Figure 3(a).

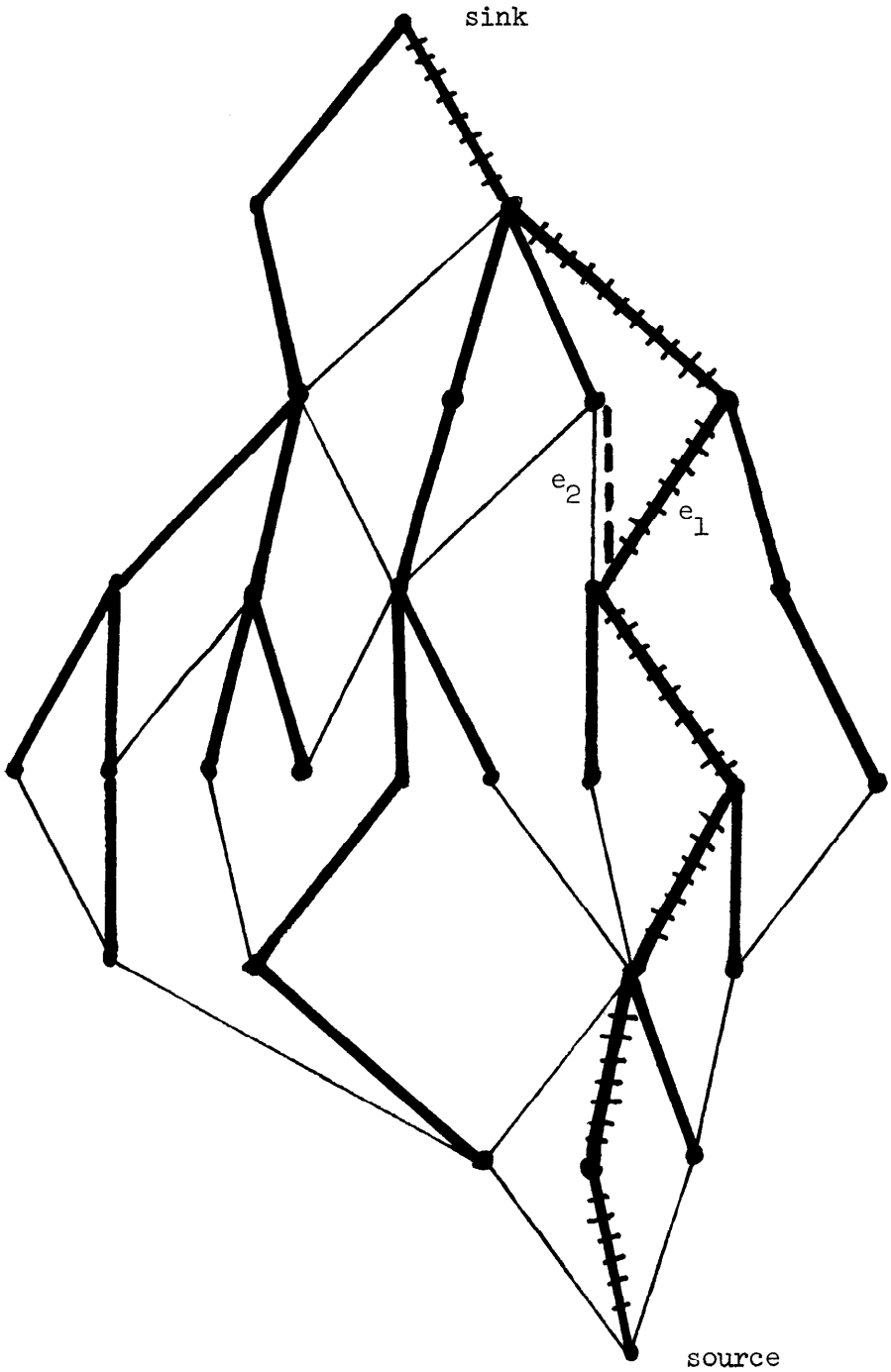

Figure 4. Candidate edges for augmenting paths. Bold edges denote spanning tree. Sending flow along path from source to sink saturates edge e_1, which is replaced by e_2 in the tree.

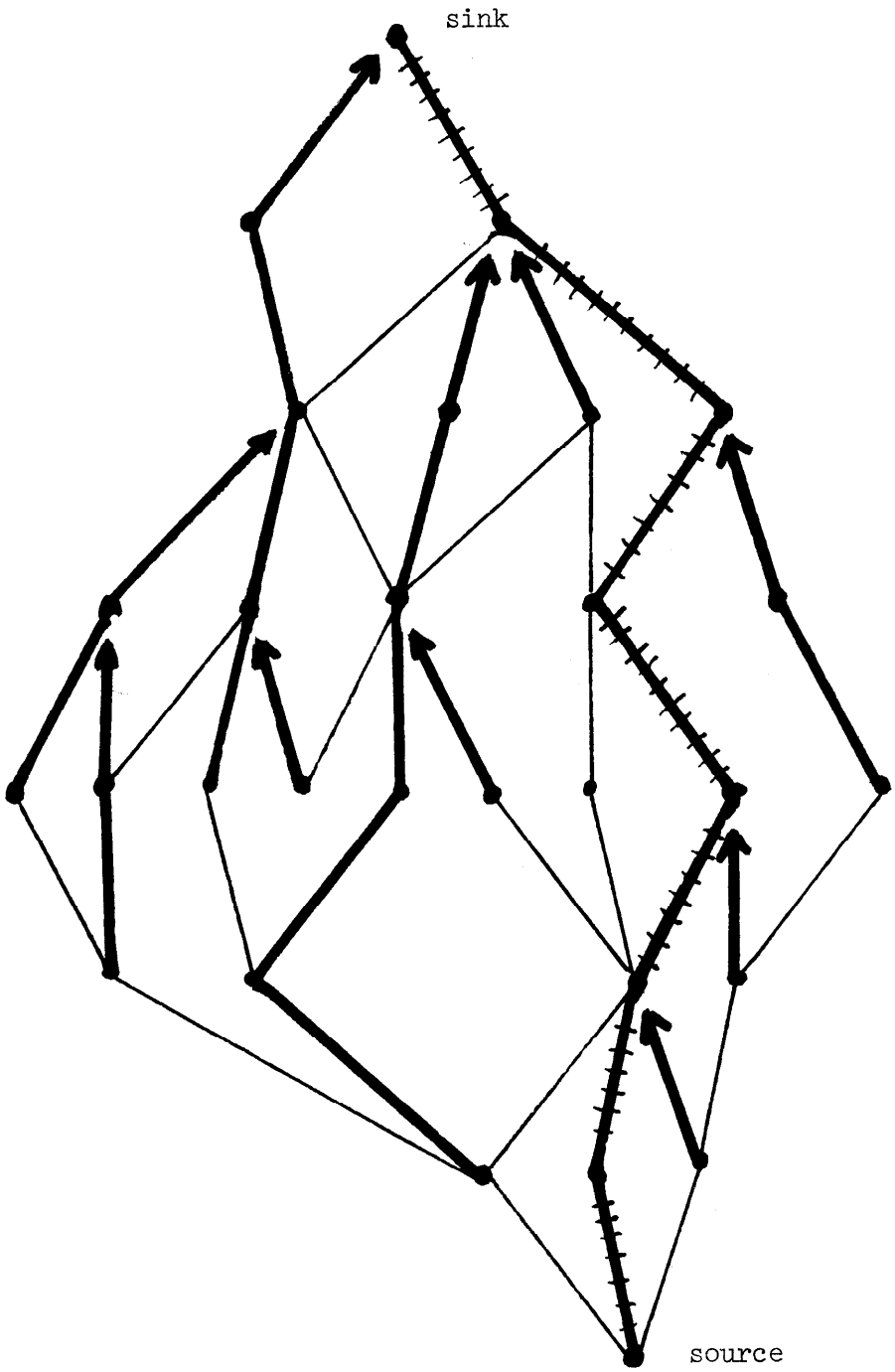

Figure 5. Decomposition of the original tree of Figure 4 into paths, one of which leads from source to sink.

Date	Discoverer	Exponent of n
1969	Strassen	2.807
October 1978	Pan	2.795
November 1978	Bini, et.al.	2.78
June 1979	Schönhage	2.609
October 1979	Pan	2.605
October 1979	Schönhage	2.548
October 1979	Pan and Winograd	2.522
March 1980	Pan	2.49+

Table 1. Improvements in matrix multiplication.

Date	Discoverer	Running Time
1956	Ford and Fulkerson	--
1969	Edmonds and Karp	$O(nm^2)$
1970	Dinic	$O(n^2m)$
1974	Karzanov	$O(n^3)$
1978	Malhotra, et, al.	$O(n^3)$
1977	Cherkasky	$O(n^2 m^{1/2})$
1978	Galil	$O(n^{5/3} m^{2/3})$
1979	Galil and Naamad, Shiloach	$O(nm(\log n)^2)$
1980	Sleator and Tarjan	O(nmlogn)

Table 2. Improvements in maximum network flow.