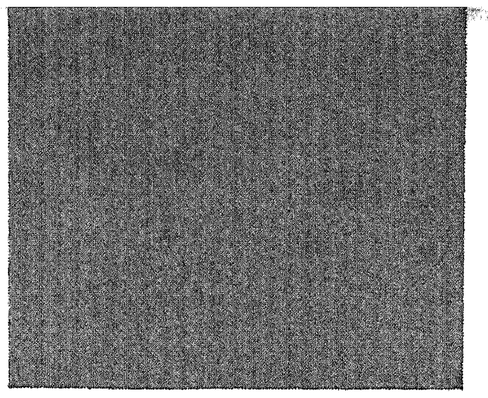
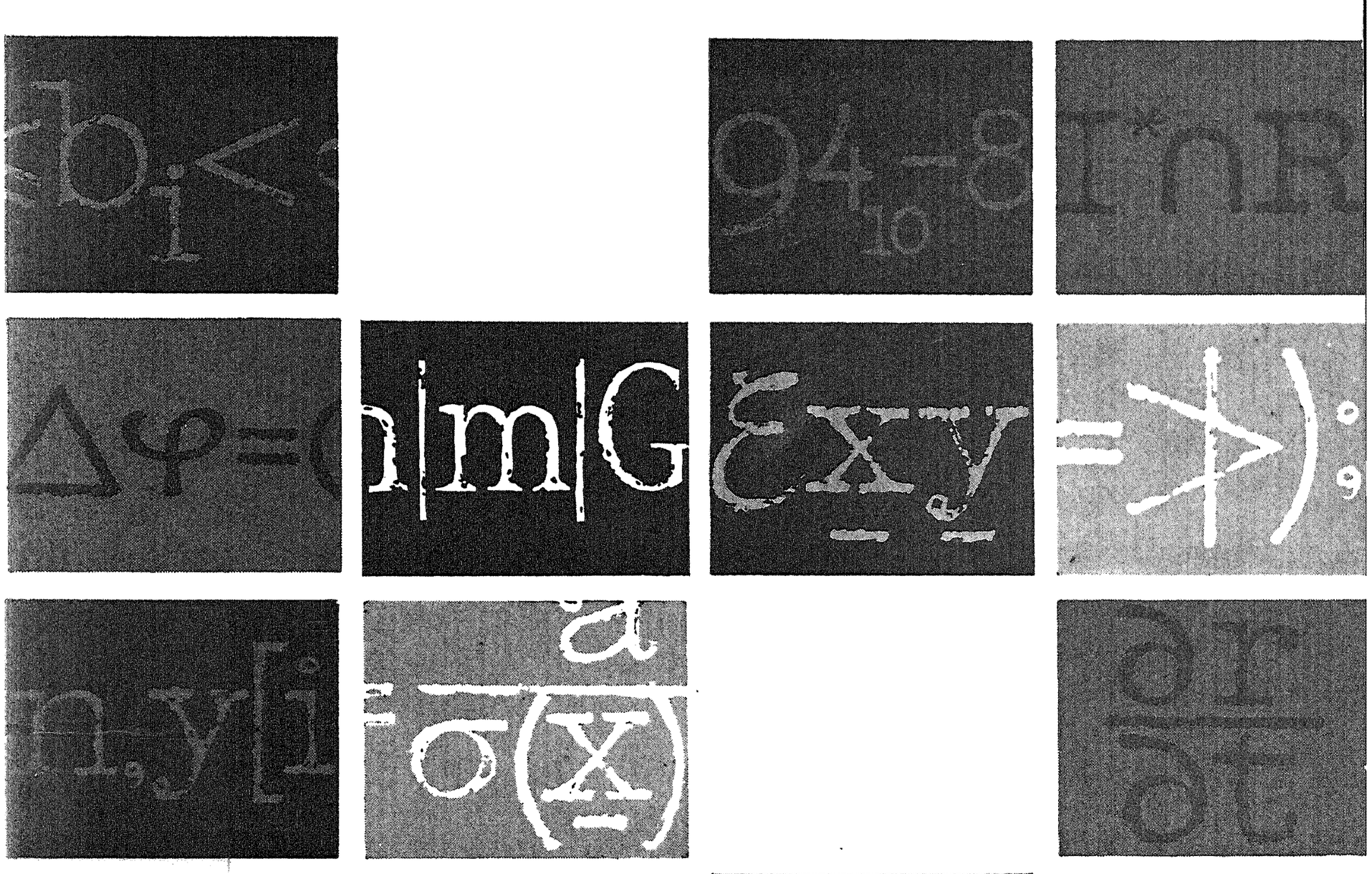


INTERFACES BETWEEN COMPUTER SCIENCE AND OPERATIONS RESEARCH

Edited by J.K. LENSTRA
A.H.G. RINNOOY KAN
P. VAN EMDE BOAS



MATHEMATICAL CENTRE TRACTS 99

MATHEMATICAL CENTRE TRACTS 99

INTERFACES BETWEEN COMPUTER SCIENCE AND OPERATIONS RESEARCH

PROCEEDINGS OF A SYMPOSIUM HELD AT THE MATHEMATISCH CENTRUM, AMSTERDAM, SEPTEMBER 7 - 10, 1976

Edited by

J.K. LENSTRA

A.H.G. RINNOOY KAN

P. VAN EMDE BOAS

MATHEMATISCH CENTRUM AMSTERDAM 1978

AMS (MOS) subject classification scheme (~~1970~~): 05Cxx, 60D05, ~~68A10~~, ~~68A20~~, 90B35, 90C10

ISBN 90 6196 170 X

PREFACE

This volume contains the proceedings of a "Symposium on Interfaces between Computer Science and Operations Research", held at the Mathematisch Centrum, Amsterdam, September 7-10, 1976.

The program of the symposium consisted of the following lectures.

1. The influence of the machine model on computational complexity, by *W.J. Savitch*.
2. Developments in data structures, by *P. Van Emde Boas*.
3. Graphical algorithms, by *E.L. Lawler*.
4. Programming for linear and integer programming, by *J.M. Anthonisse* and *B.J. Lageweg*.
5. Complexity of combinatorial problems, by *J.K. Lenstra*.
6. Worst-case analysis of algorithms, by *M.L. Fisher*.
7. Probabilistic analysis of algorithms, by *R.M. Karp*.
8. Scheduling on parallel machines, by *A.H.G. Rinnooy Kan* and *E.L. Lawler*.

The content of lectures 1, 2, 5 and 6 has been adapted to appear in this volume. Lectures 3 and 7 were of a general nature; the papers contributed by these lecturers deal with more specific problems. It has not been possible to publish a written version of lecture 4. The material presented in lecture 8 is included in the final paper in the proceedings.

The symposium grew out of the observation that the disciplines of computer science and operations research, though never far apart, seem to exhibit more and more interaction and that many interesting developments in both areas occur at the interfaces between them. On one hand, the theory of operations research has found large-scale application only through the power of modern computing devices, and its development has benefited from the study of problem complexity and the design and analysis of algorithms. On the other hand,

problems arising in the design and analysis of operating systems have led to an increased interest in queueing and scheduling theory. Thus, computer science contributes to the solution of operations research problems and asks for the solution of such problems at the same time. The proceedings demonstrate the importance of both aspects.

In order to allow mathematical analysis of computer algorithms, a formal model of a computer is required. The paper by *W.J. Savitch* provides a survey of the various models that have been proposed. Particular attention is given to the influence of the machine model chosen on standard performance measures such as running time and storage requirements.

Implementing an algorithm on a computer is an art by itself. Indeed, an appropriate data structure may yield important savings in time and storage. This is especially true for the representation and manipulation of sets, for which most programming languages hardly provide any support. The paper by *P. Van Emde Boas* demonstrates how in a few years a wide range of elegant and efficient data structures for this goal has become available, and indicates on which grounds to choose between them.

In spite of all elaborate methods, many problems currently remain essentially intractable, even for the fastest machines. The running time of any known algorithm for their solution grows superpolynomially with increasing problem size. Recent developments in computational complexity theory have provided means to show that, due to the inherent difficulty of such a problem itself, no polynomial-time algorithm is likely to exist. The resulting theory of NP-completeness is now a standard tool in the analysis of combinatorial problems. The paper by *J.K. Lenstra* and *A.H.G. Rinnooy Kan* presents an introduction to this area and illustrates various proof techniques.

NP-completeness of a problem justifies the use of a superpolynomial method to find an optimal solution, but also suggests that is may be wise to apply a fast heuristic and accept an approximate solution. Analyzing the behavior of heuristics then becomes of obvious interest. A worst-case analysis tries to establish a performance guarantee for an algorithm; a probabilistic analysis requires the specification of a density function for the problem instances and may yield results with respect to, *e.g.*, the average-case performance of an algorithm. Again, these are areas in which enormous progress has been made recently. The paper by *M.L. Fisher* reviews the techniques of worst-case analysis of heuristics and describes some examples.

This concludes the first part of the proceedings, which is devoted to general concepts, techniques and results. In the remaining papers these con-

cepts are used and the techniques and results are applied in the context of more specific problems.

The first paper in this second part has been contributed by *A. Hajnal* and *L. Lovász*. It provides and elegant solution to a well-known, if not notorious, combinatorial problem involving the spread of infectuous diseases. The authors prove a lower bound on the cheapest prevention method and present an algorithm that meets this bound within an additive constant equal to 1.

The paper by *E.L. Lawler* contains a very thorough worst-case analysis of heuristics for knapsack problems. The approximation schemes discussed produce solutions within ε percent from the optimum and run in time polynomial in the size of the problem and the inverse of ε.

The paper by *R.M. Karp* takes a probabilistic approach to the analysis of heuristics for the traveling salesman problem in the plane. In spite of many technical complications, surprising new insights into this classical problem are obtained.

The final paper by *R.L. Graham, E.L. Lawler, J.K. Lenstra* and *A.H.G. Rinnooy Kan* presents a comprehensive survey of the theory of deterministic sequencing and scheduling. In this area, many of the tools previously introduced find successful application.

The editors of these proceedings are confident that the coming years will confirm the growing importance of the interfaces discussed during the symposium. They are grateful to the guest speakers and authors for their contributions, to the audience at the symposium for their participartion, and to the Netherlands Organization for the Advancement of Pure Research (Z.W.O.) and the Graduate School of Management in Delft for their financial and organizational assistance. Finally, they thank the Mathematisch Centrum for the opportunity to publish the volume in the series Mathematical Centre Tracts and all those at the Mathematisch Centrum who have contributed to its technical realization.

J.K. Lenstra
A.H.G. Rinnooy Kan
P. Van Emde Boas

Amsterdam, April 1978

CONTENTS

THE INFLUENCE OF THE MACHINE MODEL ON COMPUTATIONAL COMPLEXITY
W.J. Savitch 1

DEVELOPMENTS IN DATA STRUCTURES
P. Van Emde Boas 33

COMPUTATIONAL COMPLEXITY OF DISCRETE OPTIMIZATION PROBLEMS
J.K. Lenstra, A.H.G. Rinnooy Kan 63

WORST-CASE ANALYSIS OF HEURISTICS
M.L. Fisher 87

AN ALGORITHM TO PREVENT THE PROPAGATION OF CERTAIN DISEASES AT MINIMUM COST
A. Hajnal, L. Lovász 105

FAST APPROXIMATION ALGORITHMS FOR KNAPSACK PROBLEMS
E.L. Lawler 109

PROBABILISTIC ANALYSIS OF PARTITIONING ALGORITHMS FOR THE TRAVELING-SALESMAN PROBLEM IN THE PLANE
R.M. Karp 141

OPTIMIZATION AND APPROXIMATION IN DETERMINISTIC SEQUENCING AND SCHEDULING: A SURVEY
R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan 169

BIBLIOGRAPHY 215

THE INFLUENCE OF THE MACHINE MODEL ON COMPUTATIONAL COMPLEXITY

W.J. SAVITCH

University of California, San Diego, U.S.A./

Mathematisch Centrum, Amsterdam, The Netherlands

ABSTRACT

The most common abstract models for digital computers are described and these models are compared with respect to time and storage efficiency.

CONTENTS

1. INTRODUCTION	3
2. THE TURING MACHINE MODEL	5
2.1. The usual model	5
2.2. Time and storage defined	7
2.3. Resolution of the model	8
2.4. The single tape model	8
3. RANDOM ACCESS STORAGE MODELS	10
3.1. The Cook-Reckhow model	10
3.2. RAM storage	12
3.3. RAM time	14
3.4. RAM-ALGOL	16
3.5. Stored program machines	17
3.6. The class P	17
4. NONDETERMINISTIC MODELS	18
4.1. Nondeterministic RAMs	18
4.2. Nondeterministic time and storage	21
4.3. Nondeterministic Turing machines	22
4.4. Relationships between different nondeterministic models	23
4.5. Deterministic simulation of nondeterministic machines	23
5. MULTIPLICATION RAMS	25
6. PARALLEL PROCESSING RAMS	27
6.1. The k-PRAM model	27
6.2. Eliminating nondeterminism	30
6.3. Counting processors	31
6.4. Time storage trade-off	31
7. SUMMARY	32
ACKNOWLEDGMENT	32

1. INTRODUCTION

In order to state with precision the amount of time or storage consumed by an algorithm, we need a well-defined formal model of a computer. Different formal models can give different values for the amount of time or storage needed for a given informally stated algorithm. In this paper we will survey the various common models for a computer and will compare the models with respect to time and storage efficiency. One of our conclusions will be that, if care is taken to exclude certain models with special characteristics, then the various models do not vary too greatly with respect to time and storage efficiency.

All of our remarks will be predicated on a number of assumptions, some important and some merely convenient. In all our models we assume that the computer has unbounded memory. When measuring time and storage, we will always measure these quantities as a function of the problem size. So time and storage measures will always be functions from and to the natural numbers. When we say that a problem is doable in time or storage g(n), we will mean that if an instance of the problem has size n, then the program uses at most g(n) units of time or storage. So, for example, to say that a graph can be checked for planarity in linear time means that there is a constant c and an algorithm which, given any graph of "size" n, will determine if the graph is planar and will do so in at most cn time units. Thus we are not discussing algorithms that depend on some upper bound on the problem size and we are not, directly, dealing with the problem of getting something to run on a particularly small machine. Also, in discussing time and storage bounds we will only consider asymptotic results. The models discussed do not seem to give very meaningful results beyond this limit of resolution. Following standard usage, we define $f(n) = O(g(n))$ to mean that there is a constant c such that $f(n) \leq cg(n)$, for all n greater than some threshold. With all our models, to say something is doable in time or storage g(n) conveys little more information than to say it is doable in time or storage O(g(n)). In some cases discussed below we will even have to settle for less resolution than this.

The remainder of our assumptions are to simplify the exposition and are not critical to the results presented. We will assume that all problems have yes/no answers. This is reflected in the formalism by the fact that we will talk about languages being accepted rather than problems being solved. The language corresponding to a problem is just the set of inputs

which yield (if the program is correct) the answer "yes". We could have stated the results for problems which admit of arbitrary answers and similar results would hold. Also, we will only be concerned with worst-case analysis. This is convenient and consistent with most of the literature on the subject. However, all our results are case-by-case simulation algorithms. So the results would apply equally well to average-case analysis.

2. THE TURING MACHINE MODEL

2.1. The usual model

The oldest and most extensively studied abstract model of a general-purpose computer is the Turing machine model. The model was first introduced by Turing in 1936 [Turing 1936, 1937]. Since that time Turing's model and variations on it have been central to much of the theoretical development in computer science. The most commonly used variant of Turing's model is the so-called "deterministic, multi-tape, off-line Turing machine". In this paper we will use the unmodified term "Turing machine" to refer to this model. Below is an informal definition of the model. The reader who finds this description too imprecise can consult any of a number of standard references ([Hopcroft & Ullman 1969] for example).

A *Turing machine* is a finite state machine attached to a read-only input tape, finitely many read/write storage tapes, and a write-only output tape. The finite state machine is referred to as the *finite state control* or sometimes just the *control*. The tapes are divided into squares. Each square of a storage tape is capable of holding any one symbol from a specified finite storage tape alphabet. The storage tapes are infinitely long in both directions. The output tape has a left-hand end but extends infinitely long to the right. There is a specified finite input alphabet and a specified finite output alphabet. The input consists of a string over the input alphabet and is placed on the input tape. The input tape is provided with two distinguished end markers, one at each end of the input string. Each tape has one *tape head* communicating with the finite state control. The situation is diagrammed in Figure 1. The machine in Figure 1 has two storage tapes, and the end markers are denoted by ¢ and $.

At any point in time, each head will be scanning one square on its tape and the finite state control will be in one state. Depending on this state and the symbols scanned by the input and storage tape heads, the machine will, in one step, do all of the following:

(1) overwrite a symbol on the scanned square of each of its storage tapes (it is, of course, permissible to overwrite a symbol by itself and so leave it unchanged);

(2) shift its input head and each storage tape head either left one square, right one square or not at all (different heads may get different instructions);

(3) possibly, write a symbol on the ouput tape; in this case the output tape head is advanced one square to the right so that it is ready to write the next output symbol;

(4) change the state of the finite state control.

The finite state control is designed so that the input head will never leave the segment of tape containing the input string and the end markers.

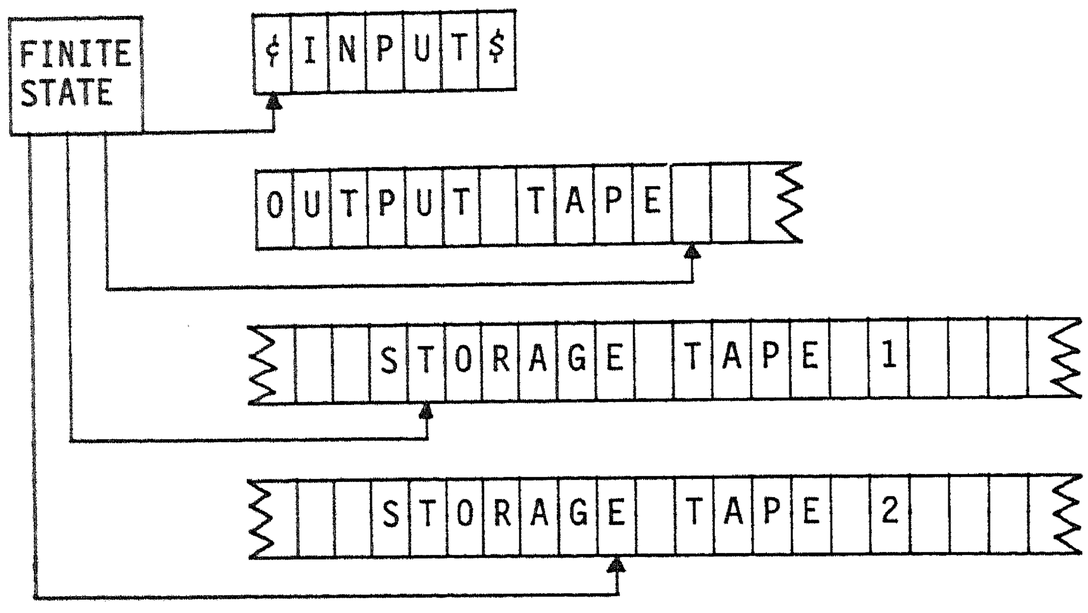


Figure 1 Turing machine with two storage tapes.

One state of the finite state control is designated as a *start state* and finitely many states are designated to be *accepting states*. One special tape symbol is designated as the *blank* symbol. At the start of a computation, the input string is placed on the input tape and delimited by the end markers, the input tape head is set scanning the left end marker, the finite state control is put into the start state, the output tape and storage tapes are blank, and the output tape head is placed at the left end of the output tape.

A Turing machine is said to *compute a function* f from input strings to output strings, provided that starting in the designated start configuration with input w, the machine eventually halts in an accepting state with f(w) written on its output tape. If f is a partial function, then it is usual to insist that the machine does not halt on any input w for which f(w) is undefined.

To simplify the discussion we will confine ourselves to situations in which the input is in some sense either accepted or rejected. In these cases there is no need for an output tape. From now on, we will assume that our Turing machines have no output tape. If the machine reaches an accepting state, that will designate acceptance. Thus, we say that the machine

accepts the input w provided that the computation of the machine on input w eventually reaches an accepting state. We say that a Turing machine M *accepts the language* L provided that L is the set of all input strings accepted by M.

2.2. Time and storage defined

We now introduce the measures of time and storage that we will use for the Turing machine model. In all cases, the time and storage will be measured as a function of the length of the input.

DEFINITION. Let M be a Turing machine, let A be a set of strings over the input alphabet of M, and let both T(n) and S(n) be functions on the natural numbers.

(1) M is said to *accept* A *within time* T(n) provided that
   (i) M accepts exactly those input strings which are in A, and
   (ii) for each string w in A, the computation of M on w takes T(n) or fewer steps, where n is the length of w.

(2) M is said to *accept* A *within storage* S(n) provided that
   (i) M accepts exactly those input strings which are in A, and
   (ii) for each string w in A, the computation of M on w uses no more than S(n) storage tape squares, where n is the length of w; that is, a total of at most S(n) storage tape squares are scanned by the storage tape heads during the computation on w.

There are a number of observations to be made about these two definitions. First note that we are only measuring time and storage on those inputs which are accepted. If an input is not accepted, then the machine may use any amount of time and storage. This may seem peculiar. However, for well-behaved time and storage bounds it can be shown that the above definition is equivalent to one that requires that the machine always operates within the bound specified. Later on we will introduce the notion of nondeterministic machines. The reason for giving the definition in this form is so that it will be consistent with the definitions given for nondeterministic machines. It should also be noted that the above definition can easily be extended to accommodate machines that have an output tape and which compute some partial function. Finally, note that when measuring storage, no charge is made for the input tape; only the storage tapes are charged for. This is

to accommodate procedures that run in very small storage.

2.3. Resolution of the model

In the introduction, we said that we would only be considering asymptotic results. If we confine ourselves to the Turing machine model, then this restriction is forced on us by the model. Let us first consider storage. It can be shown that, if a language is accepted by a Turing machine within storage S(n), then we find another machine that accepts the same language in storage cS(n), for any constant c, no matter how small. To accomplish this, all we need to do is enlarge the tape alphabet so many symbols can be coded as a single symbol. So, saying that something is doable on a Turing machine in storage S(n), conveys no more information than to say it is doable in storage O(S(n)). The following result makes this more precise.

THEOREM 1. *If* A *is accepted by a Turing machine within storage* S(n), *and if* c *is any constant greater than zero, then we can find another Turing machine that accepts* A *within storage* cS(n).

A result similar to Theorem 1 can be proven for time. So, saying that something is doable on a Turing machine in time T(n), conveys no more information than to say it is doable in time O(T(n)). (Theorems 1 and 2 are from [Hartmanis *et al.* 1965; Hartmanis & Stearns 1965], and can now be found in many introductory texts.)

THEOREM 2. *If* A *is accepted by a Turing machine within time* T(n) *and* c *is any positive constant, then we can find another Turing machine that accepts* A *within time* $T_2(n) = \max\{cT(n), n+1\}$, *provided* $\inf_{n\to\infty} T(n)/n = \infty$.

2.4. The single tape model

Another Turing machine model that is often used is the so-called "deterministic, *single tape Turing machine*". This model has only one tape and a finite state control. The tape is of the same type as the storage tapes of the usual model; the tape head on this single infinite tape can both read and write. Since the input is delimited by blanks, there is no need for special end markers. A computation begins with the finite state control in a designated start state and with the tape head scanning the left most symbol of

the input. The computation proceeds in much the same way as with the usual model. At an instant of time, the finite state control is in some state and the tape head is scanning some symbol. On the basis of this information, the machine, in one move, replaces the scanned symbol by some, possibly different, symbol, changes the state of the finite state control, and moves the tape head a maximum of one square. The input is accepted if the computation ever reaches a configuration with the finite state control in one of a designated set of accepting states. Acceptance of a language within designated time and storage bounds is defined analogously to how they are defined for the usual model.

The single tape model has the virtue of being particularly simple and, for some purposes, is equivalent to the usual model. It can be shown that for any storage bound $S(n) \geq n$, a language is accepted by some (deterministic) single tape Turing machine within storage $S(n)$ if and only if it is accepted by some usual Turing machine within storage $S(n)$. The situation for time is a little more complicated. Forcing a procedure to work with but one tape can increase the run time significantly. However, the next theorem shows that it cannot increase it above the square of the time needed by the usual model.

THEOREM 3. *If a language* L *is accepted by a usual (deterministic, off-line, multi-tape) Turing machine in time* $T(n)$, *then* L *is accepted by some (deterministic) single tape Turing machine in time* $(T(n))^2$, *provided* $\inf_{n\to\infty} T(n)/n = \infty$.

Theorem 3 is from [Hartmanis & Stearns 1965]. A discussion of this and other results about single tape Turing machines can be found in [Hopcroft & Ullman 1969].

3. RANDOM ACCESS STORAGE MODELS

Most real computers can access a large part of their memory immediately, that is, in one machine step. In this sense, the Turing machine model is somewhat unlike real computers. If a Turing machine tape head is at one square and the machine needs to access information that is k squares away from the tape head location, then it takes k steps before the machine can start to access this information. Since k may be very large, the time penalty for implementing algorithms on a Turing machine can be large. The multi-head models partially overcome this problem but even they have only a fixed number of memory locations that can be accessed in one step. In order to obtain more realistic models of computers, a number of authors have introduced what are called "random access machines" (RAMs). These include [Shepherdson & Sturgis 1963; Elgot & Robinson 1964; Hartmanis 1971; Cook & Reckhow 1973]. The model most often used is the Cook-Reckhow model, and our discussion of RAMs is based on their paper.

3.1. The Cook-Reckhow model

The Cook-Reckhow RAM is a model that uses a programming language which is very similar to the assembly language of many existing computers. The definition is as follows. A RAM consists of an infinite sequence $R_0, R_1, R_2, \ldots$ of registers together with a program. Each register is capable of holding any one integer. The program consists of a finite list of uniquely labeled instructions chosen from the instruction list given in Table 1. In this table, a and b may be any operands of the form i, indicating the integer value i, or R_i, indicating the contents of register R_i; the relation *COMP* may be any of the binary relation symbols $<, \leq, =, \geq, >, \neq$, where these symbols have their usual interpretation. The symbolism $R[R_j]$ is used to indicate indirect addressing. $R[R_j]$ stands for R_i, where i is the contents of register R_j. If $R_j < 0$ and the RAM attempts to execute an instruction involving $R[R_j]$, then the computation is aborted.

The effect of each of these instructions should be evident. For example, $R_5 \leftarrow 7$ causes register R_5 to assume the value 7; if R_3 contains 6, then $R[R_3] \leftarrow 12$ causes register R_6 to assume the value 12; $R_7 \leftarrow R_1+3$ causes register R_7 to assume the value x+3 where x is the value of the register R_1, and so forth.

A RAM recieves a single integer as input. A computation proceeds as

TABLE 1. RAM INSTRUCTIONS AND EXECUTION TIMES

function	instruction format	execution time
direct assignment	$R_i \leftarrow b$	$\ell(b)$
indirect assignment	$R[R_i] \leftarrow b$	$\ell(R_i)+\ell(b)$
indirect assignment	$R_i \leftarrow R[R_j]$	$\ell(R_j)+\ell(R[R_j])$
addition	$R_i \leftarrow a+b$	$\ell(a)+\ell(b)$
subtraction	$R_i \leftarrow a-b$	$\ell(a)+\ell(b)$
conditional branch	IF a *COMP* b THEN LABEL_1 ELSE LABEL_2	$\ell(a)+\ell(b)$
unconditional branch	GOTO LABEL	1
accept input	ACCEPT	1

follows. Initially, register R_0 contains the input and all other registers are set to zero. The RAM program is started at the first instruction. The instructions are executed in order until a conditional or unconditional branch is encountered. When an instruction of the form GOTO X is encountered, then control is transferred to the instruction with label X; the instructions are then executed in order starting with the instruction labeled X. A conditional branch is similar. If the comparison a *COMP* b is true, then the instruction is equivalent to GOTO LABEL_1; if the comparison is false, then the instruction is equivalent to GOTO LABEL_2. The program terminates whenever any of the following types of instructions are encountered: ACCEPT, a branch that transfers control to a label which is not in the program, or an instruction involving a negative indirect address. The input is said to be accepted, if the computation on the input terminates with the instruction ACCEPT. The language accepted by the RAM is the set of all nonnegative integers accepted by the RAM.

We could have defined RAMs so that they compute arbitrary partial recursive functions. For example, we could say that the RAM has output f(x) for input x provided that the computation on input x terminates with the instruction ACCEPT and that, when it does terminate, the contents of register R_0 is f(x). All our remarks would apply equally well to such RAMs with output. Also, we could have defined RAMs to have several integers as input instead of just one input; we would obtain a theory similar to the one we get for RAMs with a single input. However, as in the Turing machine case, we have, for simplicity, confined ourselves to the special case where a single input is simply accepted or not accepted.

In order to compare Turing machines and RAMs, we will need to consider

the input to a RAM as a string of symbols. To make this comparison easy, we have assumed that the input to a RAM is a nonnegative integer. So, if we identify nonnegative integers with their binary representation, then a RAM accepts strings over the alphabet {0,1}. (We always assume that leading zeros have been trimmed when we take the binary representation of an integer.) Thus we can talk of a set of integers being accepted by either a RAM or a Turing machine. Similarly we can talk of a language over the symbols {0,1} as being accepted by either a RAM or a Turing machine. Leading zeros present a minor problem. However, this problem can be accommodated in a number of ways. For example, when we wish to give the RAM an input w which is a string of zeros and ones, we can give the RAM the integer with binary representation 1w as input. By taking integers in some base other than two, we can similarly identify strings over any alphabet (with at least two symbols) with the nonnegative integers. All these details are, however, of minor importance. Any reasonable way of identifying RAM and Turing machine inputs would yield similar results.

3.2. RAM storage

Since the instruction set for a RAM resembles the assembly language of many existing computers and since the registers behave rather like the memory locations of such computers, one might be tempted to define the storage used in a RAM computation as the maximum number of registers used. The next result indicates that there would be something peculiar about such a definition. The theorem is from [Minsky 1961].

THEOREM 4. *If* A *is accepted by a* RAM, *then* A *is also accepted by some other* RAM *that uses only two registers.*

An analysis of the proof of Theorem 4 indicates how the RAM model admits of this peculiarity. The proof uses a simulation algorithm which generates extremely large numbers. This presents no problem in the RAM model, since any register can hold an arbitrarily large integer. If, on the other hand, the simulation were to be implemented on a real computer, then it would take many memory locations to hold the contents of these two registers. A RAM register corresponds to a chunk of memory which may grow arbitrarily large; it does not correspond to a single memory location of a real computer. Thus, in counting storage, we will count the number of bits used rather

than the number of registers used.

DEFINITION. Let M be a RAM, A a set of natural numbers and S(n) a function on the natural numbers. M is said to *accept* A *within storage* S(n) provided

(i) M accepts exactly those natural numbers which are in A, and

(ii) for each number w in A, the following inequality is satisfied at each step in the computation of M on w:

$$\sum_{i=0}^{m} \text{length}(R_i) \leq S(n),$$

where n is the length of w in binary, m is the largest address of any register referenced in the computation, and length(R_i) is the length of the contents of R_i. (In computing length(R_i), we set length(0) = 1; otherwise, we take the length of the binary numeral for the contents of R_i, after deleting the sign and any leading zeros.)

Formally, the RAM model gives better resolution than the Turing machine model. The analogue of Theorem 1 for a RAM is not true. It can be shown that with a slight increase in storage, a RAM can accept additional languages. However, if we implement an algorithm on a RAM as described above and also on a RAM defined in a slightly different way, then the storage needs can differ by a constant multiple. For example, if we charge for storing the sign of an integer, compute lengths of integers as their base ten representation or charge for storing register addresses, then our storage bounds can change by a constant multiple. So to say something can be done on a RAM in storage S(n), conveys little more information than to say it can be done in storage O(S(n)).

Storage is a very stable concept. If we implement an algorithm on most any two different reasonable models of a computer, we will need about the same storage on both models. If the algorithm can be done in storage O(S(n)) on one model, it can be done in storage O(S(n)) on the other. The next theorem formalizes this result when the two models are the RAM and the Turing machine. The result is implicit in [Cook & Reckhow 1973].

THEOREM 5. *Suppose* S(n) ≥ n. *A set* A *is accepted by some Turing machine within storage* S(n) *if and only if* A *is accepted by some* RAM *in storage* O(S(n)).

If we were to provide our RAMs with a separate input tape like that of a Turing machine, then the hypothesis $S(n) \geq n$ in Theorem 5 can be dropped.

3.3. RAM time

The run time of a RAM program is computed using a function which assigns a time charge (execution time) to each instruction executed. The execution time of each instruction is given in Table 1, where ℓ is a function from integers to positive integers. Different choices for ℓ can, of course, give vastly different run times.

DEFINITION. Let M be a RAM, A a set of natural numbers and T(n) a function on the natural numbers. M is said to *accept* A *within time* T(n) using the ℓ-cost criterion provided

(i) M accepts exactly those natural numbers which are in A, and

(ii) for each number w in A, the computation of M on w takes time T(n), where n is the length of w in binary; that is, if we sum up the execution cost (as given in Table 1) of all instructions executed in this computation, then this sum is at most T(n).

The most common values for the function ℓ are $\ell(x) = 1$ for all x and $\ell(x) =$ length of x in binary. If we use the first choice for ℓ, the RAM is said to operate with the *uniform cost criteria*. If we use the second choice for ℓ, the RAM is said to operate with the *logarithmic cost criteria*. The logarithmic cost criteria usually gives a more accurate measure of what the run time of an algorithm would be if implemented on a real machine. The reason for this is that registers may hold arbitrarily large integers. Suppose a RAM program were implemented on a real machine with fixed word length. The fixed word length machine would require about $c \log_2 x$ storage locations to store the number x held by a single RAM register, where c is a constant depending on the word length. So it would require about $c \log_2 x$ memory accesses on the real machine in order to simulate one memory access to a register containing x. Thus the logarithmic cost criteria gives a good approximation to the number of memory accesses needed to implement an algorithm on a real machine with fixed word length. Of course, the constant c is lost but, since this c is machine dependent, we have no hope of capturing it in a single formal model. To say something is doable on a RAM, with logarithmic cost, in time T(n) conveys little more real information than to say it is doable in

time O(T(n)).

Although the logarithmic cost criteria is a more realistic time measure, the uniform cost criteria is also widely used. When a program produces only register values which are relatively small, the uniform cost criteria is about as accurate as the logarithmic cost criteria and can safely be taken as an approximate measure of the time needed to implement the program on a real machine. Even when numbers are allowed to grow very large, the uniform cost criteria still gives a crude approximation to the cost of implementing the program on a real machine. The following theorems show the relationship between the various time measures discussed so far.

THEOREM 6. *Suppose* T(n) > n.

(1) *If* A *is accepted by some* RAM *using the logarithmic cost criteria in time* T(n), *then* A *is accepted by the same* RAM *using the uniform cost criteria within time* T(n).

(2) *If* A *is accepted by some* RAM *using the uniform cost criteria in time* T(n), *then* A *is accepted by some* RAM *using the logarithmic cost criteria in time* $O((T(n))^2)$.

THEOREM 7. *Suppose* T(n) > n.

(1) *If* A *is accepted by some* RAM *using the logarithmic cost criteria in time* T(n), *then* A *is accepted by some Turing machine within time* $(T(n))^2$.

(2) *If* A *is accepted by some Turing machine in time* T(n), *then* A *is accepted by some* RAM *using the logarithmic cost criteria within time* $O(T(n)\log_2 T(n))$.

THEOREM 8. *Suppose* T(n) > n.

(1) *If* A *is accepted by some* RAM *using the uniform cost criteria in time* T(n), *then* A *is accepted by some Turing machine in time* $(T(n))^3$.

(2) *If* A *is accepted by some Turing machine in time* T(n), *then* A *is accepted by some* RAM *using the uniform cost criteria in time* O(T(n)).

Theorem 6 is implicit in [Hartmanis 1971]. Theorems 7 and 8 are from [Cook & Reckhow 1973].

3.4. RAM-ALGOL

As with any assembly language, RAM programs are often difficult to write and even more difficult to read. So it is desirable to have a high level language to state programs in. One can safely use a small, but still rich, subset of ALGOL which is easier to read than RAM programs and which allows for easy estimations of the time and storage needed to implement programs on a RAM. Cook and Reckhow describe a language called RAM-ALGOL which is a subset of ALGOL 60. The main differences between RAM-ALGOL and ALGOL 60 are that in RAM-ALGOL:

1. declarations are superfluous;
2. real numbers are eliminated;
3. the only arithmetic operations are + and -;
4. procedures and switches are not allowed to be recursive;
5. arrays are one-dimensional and infinite.

Time and storage for RAM-ALGOL are computed in much the same way that they are computed for RAM programs. The *storage* used in a computation of a RAM-ALGOL program is $\sum_X \text{length}(X)$, where length is defined as before and X ranges over all variables and array elements referenced in the computation. RAM-ALGOL *run times* are also defined by analogy to RAM run times. For example, the execution time for the RAM-ALGOL statement A[Z] := Y+B[X] is $\ell(Z)+\ell(Y)+\ell(X)+\ell(B[X])$. As with RAM programs, ℓ is usually taken to be either identically one or defined by $\ell(x) = \text{length}(x)$. So we have both a uniform and a logarithmic cost criteria for RAM-ALGOL. RAM-ALGOL time and storage corresponds very closely to RAM time and storage. If a RAM-ALGOL program runs in storage S(n), then it can be converted to an ordinary RAM program that runs in storage O(S(n)). If a RAM-ALGOL program runs in logarithmic (respectively uniform) cost time T(n), then it can be converted to an ordinary RAM program that runs in logarithmic (respectively uniform) cost O(T(n)). So we can safely write in RAM-ALGOL knowing that all of our results about RAM programs apply equally well to our RAM-ALGOL programs.

We can make RAM-ALGOL even richer by taking a still larger subset of ALGOL 60. However, it is difficult to say exactly how much additional structure we can safely take and still accurately measure time and storage. For example, it is easy to handle simple instances of recursive procedures. However, for nested recursion and co-routines, it is frequently not clear what the time and storage requirements of the program are.

3.5. Stored program machines

The RAM model is a fixed program machine. The program is separated from memory and may not modify itself. One can define RAMs in such a way that the program is stored in memory and may modify itself just as it can modify any other memory locations. Such RAMs are usually called "random access stored program machines" or RASPs. Most existing computers do allow programs to modify themselves and this feature can sometimes save some time and memory. However, the amount of time and storage saved is not significant, within the bounds of accuracy we are using. If a set A is accepted by a RASP within resource bound R(n), then A is accepted by some RAM within resource bound O(R(n)). The resource bound may be any of the ones we have discussed: storage, logarithmic cost time or uniform cost time. For a discussion of RASPs, see [Hartmanis 1971; Cook & Reckhow 1973; Aho *et al.* 1974].

3.6. The class P

As we have seen, Turing machine and RAM storage are equivalent up to growth rate. So when discussing storage it matters little which model we use. The situation for time is different. The usual Turing machine model, the single tape Turing machine, the uniform cost RAM model and the logarithmic cost RAM model can each give a significantly different run time when we implement the same informally stated algorithm on the various models. However, all the run times will be polynomially related. In particular, if we devise an algorithm and show that it runs in polynomial time on one of these models, then we know that it can be made to run in polynomial time on any of the other models as well. Thus, if we are only interested in whether or not an algorithm runs in polynomial time, then it does not matter which of the above discussed models we use. With this in mind, we define P to be the class of all languages accepted in polynomial time by a usual Turing machine (equivalently single tape Turing machine, equivalently uniform cost RAM or RASP or RAM-ALGOL program, equivalently logarithmic cost RAM or RASP or RAM-ALGOL program). The class P and its invariance under change of machine model were first discussed in [Cobham 1964]. Lately, it has received much attention since it is considered an approximation to the class of practically doable problems.

4. NONDETERMINISTIC MODELS

All the models discussed so far are what is called "deterministic". That is, at each point in a computation there is a unique next step (or the computation has ended). All real computers are deterministic. This is, therefore, a very reasonable assumption to build into the definition of a model. There are, however, some very good reasons for considering models in which this assumption is relaxed to allow finitely many possible next steps. Models in which a machine configuration may in one step go to finitely many next configurations (as opposed to just one) are called "nondeterministic" models. (Whenever we do not explicitly state whether a model is deterministic or nondeterministic, we are assuming that it is deterministic.) The easiest nondeterministic model to describe is the nondeterministic RAM and so we will consider it first.

4.1. Nondeterministic RAMs

A *nondeterministic* RAM is defined exactly the same as a RAM except that we do not require that statements have unique labels. Two or more statements may have the same label. A computation of a nondeterministic RAM proceeds exactly as it does for an ordinary (deterministic) RAM until it encounters a transfer (conditional or unconditional branch) to a label L such that two or more statements are labeled L. Say there are $m \geq 2$ statements labeled L. At this point, the machine replicates itself and produces m identical copies of itself. Each copy transfers to a different one of the m statements labeled L and the m copies compute in parallel. Some of these m copies may later encounter a transfer to a label which labels two or more statements. Those copies then replicate themselves to the needed number and all the copies compute in parallel. This process continues with machines replicating themselves whenever they encounter a transfer to a label which labels two or more statements. The input is said to be accepted if at least one of these machines which are computing in parallel reaches an ACCEPT instruction. For example, consider the following nondeterministic RAM program.

```
L1: R1 ← 1
L2: GOTO L3
L3: R1 ← R1+1
L3: R1 ← R1+0
```

L4: GOTO L5

L5: $R_1 \leftarrow R_1+1$

L5: $R_1 \leftarrow R_1+0$

L6: IF $R_0 = R_1$ THEN L7 ELSE L8

L7: ACCEPT

Recall that the input is in R_0. The above program is a rather poor way to test if R_0 contains 1, 2 or 3. So the above nondeterministic RAM program accepts the set {1,2,3}. Say the input were 2. Initially, $R_0 = 2$ and all other registers contain zero. The instructions executed by the parallel machines are shown in Figure 2. The input is accepted since at least one

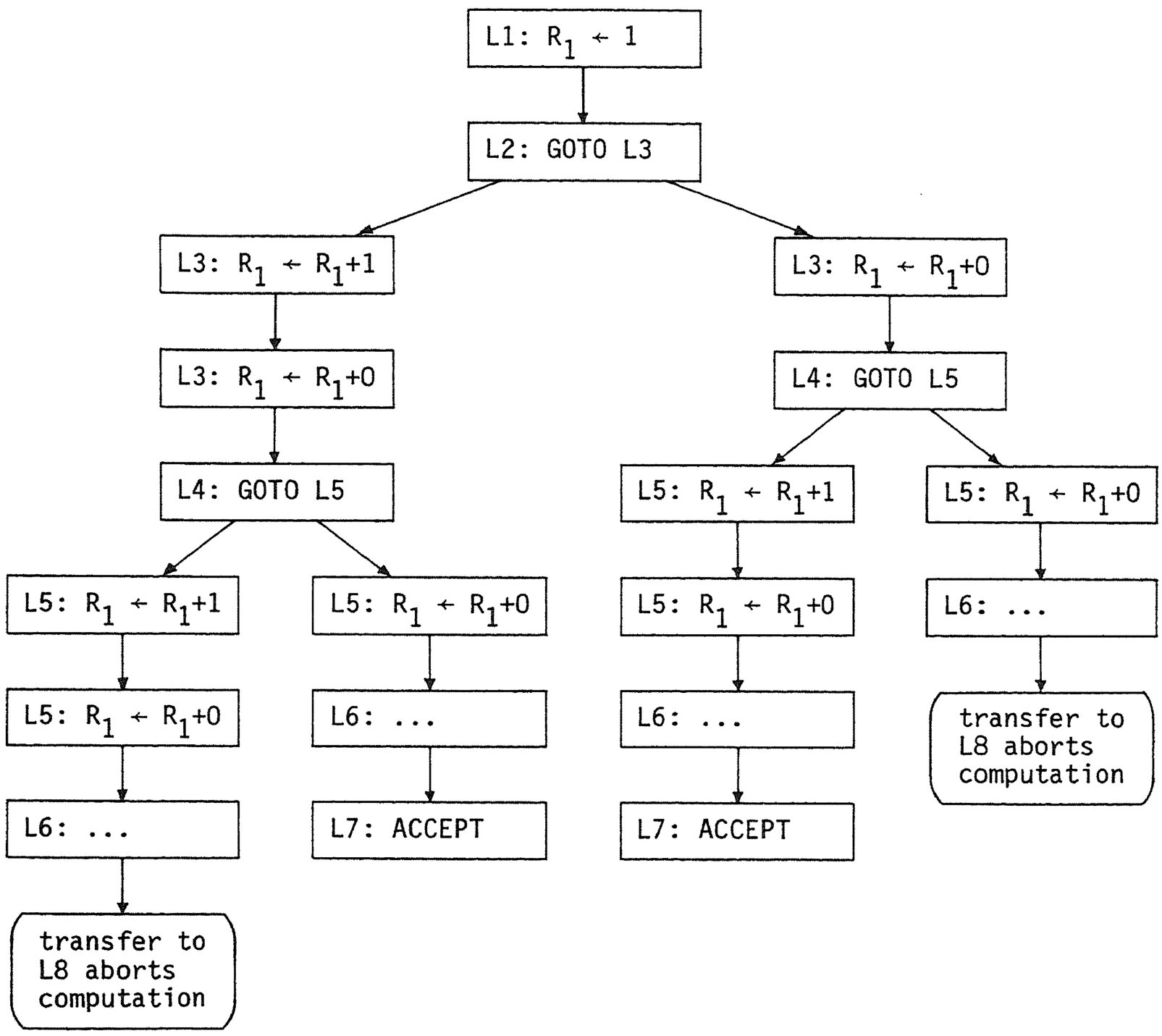


Figure 2

of the parallel computations leads to an ACCEPT instruction. The transfer to label L8 ends the computation in a nonaccepting mode.

We can think of the branching tree in Figure 2 or the replicating machines as a kind of guessing procedure. If a machine is faced with a transfer to label L and there is more than one statement labeled L, then it guesses which one to execute next. The input is said to be accepted if there is some sequence of guesses that leads to an ACCEPT instruction. The above program has four possible sequences of guesses represented by the four paths from the root to the leaves of the tree. The program can be stated informally as follows.

1. Guess at an integer R_1 such that $1 \leq R_1 \leq 3$.
2. Test: Does $R_1 = R_0$?
   Yes: ACCEPT.
   No: Abort the computation.

Notice that if a nondeterministic machine makes a series of guesses that does not lead to an ACCEPT instruction, then this does not mean that the input is in any sense rejected. In fact, no information is gained, since another series of guesses might or might not lead to an ACCEPT instruction. This is the reason we confined our attention to partial algorithms that accept the appropriate inputs but need not reject the rest of the inputs.

It is cumbersome to have to write complete nondeterministic RAM programs. However, we can often write very easily stated and understood informal, nondeterministic RAM programs. For example, consider the following informal, nondeterministic program to accept the composite numbers. As usual, the input is in R_0.

1. Guess at a number R_1 such that $1 < R_1 < R_0$.
2. Test: Does R_1 divide R_0?
   Yes: ACCEPT.
   No: Abort the computation.

If a set A is accepted by a nondeterministic RAM program then we can always produce a deterministic program to accept A. So, in some sense, nondeterministic programs give us no additional computing power. However, they are often easier to state and often run in much less time or much less storage than the best known deterministic programs to perform the same task. So one

goal of research in computer science has been to find algorithms that convert efficient nondeterministic programs into efficient deterministic programs. With such conversion algorithms at our disposal, we can write efficient nondeterministic programs and know that they can be converted to efficient deterministic programs which can then be implemented on real machines. As we shall see, sometimes there are such efficient conversion algorithms and sometimes none seem possible.

4.2. Nondeterministic time and storage

As noted above, a nondeterministic RAM can be viewed as a type of guessing machine. The input is accepted if there is a possible series of guesses that leads to an ACCEPT instruction. Each series of guesses produces a computation much like the computation of an ordinary (deterministic) RAM. Each series of guesses causes the RAM to execute a sequence of instructions that changes the contents of its registers. Just as with ordinary (deterministic) RAMs, each such computation uses some amount of storage and depending on which criteria we use, uniform or logarithmic, some amount of time.

DEFINITION. Let M be a nondeterministic RAM, let A be a set of natural numbers, and let T(n) and S(n) be functions on the natural numbers.

(1) M *accepts* A *in time* T(n) with the uniform (respectively logarithmic) cost criteria provided that
   (i) M accepts exactly those inputs which are in A, and
   (ii) for each $x \in A$, there is some computation (sequence of guesses) of M on input x that leads to an ACCEPT instruction and uses at most T(n) time units using the uniform (respectively logarithmic) cost criteria; here n is the length of x.

(2) M *accepts* A *in storage* S(n) provided that
   (i) M accepts exactly those inputs which are in A, and
   (ii) for each $x \in A$, there is some computation of M on input x that leads to an ACCEPT instruction and is such that the following inequality is satisfied at each step in the computation:

$$\sum_{i=0}^{m} \text{length}(R_i) \leq S(n),$$

where n is the length of x in binary, m is the largest address of any register referenced in the computation, and length(R_i) is the length of the contents of R_i written in binary.

Now that we have well-defined notions of nondeterministic time and storage, we can point out examples of where a nondeterministic procedure is more efficient than the obvious deterministic procedure for the same task. Consider the previously described procedure to accept the composite numbers. For a given input x it need perform only one division. The obvious deterministic algorithm would require about $\sqrt{x}$ divisions. Other papers in this tract give numerous examples of time efficient nondeterministic procedures.

4.3. Nondeterministic Turing machines

The notion of nondeterminism for Turing machines is similar to what it is for RAMs. A *nondeterministic Turing machine* is defined exactly the same way as a usual (deterministic) Turing machine except that the finite state control is allowed to be nondeterministic. That is, for any machine configuration, the Turing machine description specifies a finite number of actions each of which determines a different next configuration. Each of these finitely many actions are just like a single move of a usual Turing machine as described in Section 2.1. When a nondeterministic Turing machine has more than one possible next action, we will think of it as guessing which action to follow. Each series of guesses produces a computation which is (aside from the guessing) just like a computation of a usual Turing machine. The start configuration is defined the same as it is for usual Turing machines. A nondeterministic Turing machine is said to *accept an input* w provided there is some sequence of guesses such that the computation on input w produced by these guesses causes the finite state control to enter one of its accepting states. We can also think of a nondeterministic Turing machine as a kind of parallel machine. Instead of guessing, it replicates and produces enough copies of itself to follow all possible next steps. However, the guessing characterization will prove more useful. A more or less formal definition of time and storage measures follows. More formal definitions of these concepts can be found in [Hopcroft & Ullman 1969].

DEFINITION. Let M be a nondeterministic Turing machine, let A be a set of strings over the input alphabet of M, and let both T(n) and S(n) be functions on the natural numbers. M is said to *accept* A *within time* T(n) (respectively *storage* S(n)) provided that

(i) M accepts exactly those input strings which are in A, and

(ii) for each string w in A, there is at least one computation (sequence

of guesses) of M on input w such that this computation both accepts w and takes T(n) or fewer steps (respectively both accepts w and uses S(n) or fewer storage tape squares); here n is the length of w.

4.4. Relationships between different nondeterministic models

All of the results stated above for deterministic models carry over if we assume that the models are nondeterministic. Theorems 1 through 8 remain true if we replace every occurrence of "Turing machine" by "nondeterministic Turing machine" and replace every occurrence of "RAM" by "nondeterministic RAM". As long as all models are nondeterministic, the same simulation algorithms work. However, when one model is deterministic and one is nondeterministic, the situation appears to be quite different. A nondeterministic machine can obviously simulate a deterministic machine of the same variety with no loss of efficiency. This is trivially true, since a deterministic machine is just a special type of nondeterministic machine. It is the special case where the finite set of next moves always has cardinality at most one. How efficiently a deterministic machine can simulate a nondeterministic machine is a major area of current research in theoretical computer science.

4.5. Deterministic simulation of nondeterministic machines

As we already noted, every nondeterministic machine can be simulated by a deterministic machine. All the deterministic machine need do is to systematically try all possible sequences of guesses. However, if this is done in the most obvious way, the time and storage costs can be extremely high. The obvious algorithm can simulate nondeterministic storage S(n) in deterministic storage $k^{S(n)}$ and can simulate nondeterministic time T(n) in deterministic time $k^{T(n)}$. Here k is a constant depending on the nondeterministic procedure that is being simulated. With a more efficient simulation we can do much better than this for the case of storage. The result is from [Savitch 1970].

THEOREM 9. *If* A *is accepted by a nondeterministic Turing machine in storage* S(n), *then we can find a (deterministic) Turing machine that accepts* A *within storage* $(S(n))^2$, *provided* $S(n) \geq \log_2 n$.

The analogue of Theorem 9 holds for RAMs, or any other reasonable model,

and not just for Turing machines. Of course with RAMs, it does not make sense to talk of storage S(n) unless S(n) ≥ n. Also, since we do not have a technique to shrink RAM storage by a constant multiple, the bound of $(S(n))^2$ changes to $O((S(n))^2)$ in the case of RAMs.

The best known algorithm for simulating nondeterministic time by deterministic time is the obvious one discussed above. With that algorithm, a deterministic machine needs time exponential in T(n) in order to simulate a T(n) time bounded nondeterministic program. Some improvement on the obvious algorithm may be possible but the commonly accepted conjecture is that no major improvement can be obtained. The conjecture is usually phrased in terms of polynomial time bounded programs. By analogy to the class P, we define NP to be the class of all languages accepted in nondeterministic polynomial time. As with P, it does not matter which of the previously discussed machine models we use; they all yield the same class NP. Trivially, P is a subclass of NP. The commonly accepted conjecture is that $P \neq NP$. This conjecture has received much attention in recent years, since many important problems are known to be in the class NP, and it would be very useful to have deterministic polynomial time algorithms for these problems. Work by Cook [Cook 1971], Karp [Karp 1972, 1975A] and others has shown that if the conjecture $P \neq NP$ is true, then many practically important problems lie outside the class P and so probably do not admit of practical computer program solutions. These problems are among the so-called "NP-complete problems". A language A is said to be *NP-complete* if (i) A is in NP, and (ii) A in P implies $P = NP$. Some of the other papers in this tract [Lenstra & Rinnooy Kan 1978B; Graham *et al.* 1978] discuss NP-complete problems in more detail.

5. MULTIPLICATION RAMS

Our definition of RAMs allowed only addition and subtraction as operations but did not allow multiplication or any other arithmetic operations. Most of our analysis would remain valid if we allowed our RAMs to have multiplication and any other reasonably simple operations. All our results on storage would remain valid for a RAM with such additional operations. The situation for time is a little more complicated. If we use the logarithmic cost criteria, then the character of our results is not changed. In some cases, the run time of the simulation algorithm can increase by a square (or other relatively small power) but all models which were polynomially related using our original definition remain polynomially related if we allow our RAMs to have these additional operations. In particular, we could define P and NP using such machines and we would obtain the same two classes as we did for RAMs with only addition and subtraction. If we use the uniform cost criteria, the situation appears to be quite different. Recent work by Hartmanis and Simon has shown that, under the uniform cost criteria, a small change in the arithmetic operations allowed in a RAM model appears to change its computing efficiency significantly.

Define a *multiplication RAM* (MRAM) to be just like a RAM except that we allow the following instructions in addition to those in Table 1: $R_i \leftarrow a*b$ and $R_i \leftarrow a$ *BOOL* b. Here a*b denotes multiplication and *BOOL* may be any bitwise Boolean operation. (To perform *BOOL* we consider integers to be strings of binary bits, drop the sign and drop leading zeros.) Both instructions have execution time $\ell(a)+\ell(b)$. As with RAMs, MRAMs come in both deterministic and nondeterministic models. The next theorem is from [Hartmanis & Simon 1974].

THEOREM 10. *Assume the uniform cost criteria for* MRAMs.

(1) A *is accepted by some nondeterministic* MRAM *in polynomial time if and only if* A *is accepted by some deterministic* MRAM *in polynomial time.*

(2) A *is accepted by some Turing machine in polynomial storage if and only if* A *is accepted by some* MRAM *in polynomial time.*

Notice that in (2) we do not mention whether or not the machines are deterministic. This is because, by Theorem 9 and part (1), it does not matter: deterministic and nondeterministic polynomial storage are equivalent; deterministic and nondeterministic polynomial time are equivalent for MRAMs

with the uniform cost criteria.

As already mentioned, the common conjecture is that $P \neq NP$, where P and NP are defined using any of the models except the MRAM under the uniform cost criteria. But, by Theorem 10, MRAMs can accept all languages in NP and can do so in deterministic polynomial time. Thus it appears that, under the polynomial time restriction, determinstic MRAMs are more powerful than deterministic RAMs. Since an MRAM looks more like existing machines than a RAM does, this result needs to be reconciled with our other remarks. If MRAMs are more powerful than the other models and are also more realistic, then we should happily use the MRAM model and not the other models. However, an analysis of the proof of the Theorem 10 would show that this is not always a wise course of action. The simulation algorithm used to prove Theorem 10 generates very large numbers and, since the uniform cost criteria is assumed, these numbers are multiplied together in one machine step. If the algorithm were implemented on a real machine, these multiplications would take many machine steps, since these extremely large numbers must be multiplied with complete precision. Thus, although Theorem 10 looks like it gives a good simulation algorithm, the algorithm would not perform efficiently in practice. In order to avoid the danger of writing seemingly efficient algorithms that cannot be efficiently implemented on real machines, we should use the logarithmic cost criteria when using MRAMs, or else use one of the other models. Theorem 10 does not appear to be true for MRAMs with the logarithmic cost criteria. MRAMs with the logarithmic cost criteria have run times which are polynomially related to the run times of the other models we have discussed.

Theorem 10 does not appear to give us a practically efficient simulation algorithm. It is, however, one of the important recent results in the theory of formal models for computers. It gives us an alternate, surprising and useful characterization of the class of languages accepted in polynomial space. It also provides us with a dramatic example of the importance of using the logarithmic cost criteria. Until the result of Hartmanis and Simon, there were really no results which so dramatically separated the logarithmic and uniform cost criterias. Some very nice earlier work of Pratt, Stockmeyer and Rabin [Pratt *et al.* 1974] gave similar examples of powerful instruction sets for RAMs. Their model was not as conspicuously like existing machines as the Hartmanis-Simon MRAM model, but the two models are similar in character.

6. PARALLEL PROCESSING RAMS

With the recent rapid advance in hardware technology, multiprocessing computer systems are growing in importance and availability. Some highly parallel machines, such as the ILLIAC IV and CDC STAR-100, are already available. When very highly parallel systems become generally available, parallel processors may come to be thought of as a computing resource in much the same way that storage is now. Recently, a number of formal models for parallel processing computers have been introduced. For example, models have been proposed in [Chandra & Stockmeyer 1976; Kozen 1976; Savitch & Stimson 1976, 1978]. The proofs of all our results can be found in [Savitch & Stimson 1978].

6.1. The k-PRAM model

In order to obtain a model for parallel processing of algorithms, we extend the RAM model of Cook and Reckhow to allow parallel recursive calls. The parallel model obtained in this way is called a k-*offspring parallel random access machine* (k-PRAM). Let k be a positive integer. A k-PRAM consists of a finite program, a potentially infinite supply of processors and a positive integer u. The integer u is the number of parameters that are passed when a recursive call is made or returned. As with an ordinary RAM, each k-PRAM processor has a pointer into the program telling it which instruction to execute next; all processors use the same program. Each processor has a memory which consists of an infinite sequence of registers $R_0, R_1, R_2, \ldots$, each of which is capable of holding any integer. Each processor also has available to it k other processors (its offspring) which it can call. Each processor may have up to k of its offspring computing in parallel with it. These k processors may, in turn, call up to k offspring each, and so forth. When an offspring is done with its computation, it returns its response to its parent by way of a special bank of registers called *channels*. That offspring is then available for another recursive call. Channel ℓ, $1 \leq \ell \leq k$, is used by a processor to receive parameters passed to it by its ℓ-th offspring. Channel ℓ is a bank of u read-only registers $C_1^{\ell}, C_2^{\ell}, \ldots, C_u^{\ell}$, each of which is changed only as a result of offspring ℓ completing a computation. Therefore, each channel register is a read-only register to the parent, but the registers in channel ℓ are modified whenever offspring ℓ returns from a recursive call.

The *program* for a PRAM is a sequence of (optionally) labeled instruc-

tions. If the labels in a PRAM program are unique, the program is said to be *deterministic*; otherwise, it is said to be *nondeterministic*. The instructions for a PRAM program are drawn from the instruction set given in Table 2. In Table 2, ℓ and h are integers such that $1 \leq \ell \leq k$, $1 \leq h \leq u$; $a,b,a_1, a_2,\ldots,a_u$ are operands of the form

(1) i, indicating the integer value i, or

(2) R_i, indicating the contents of register R_i, or

(3) C_h^ℓ, indicating the contents of channel register C_h^ℓ.

The relation *COMP* in Table 2 may be any of the binary relation symbols $<,\leq,=,\geq,>,\neq$, where these symbols have their usual interpretation over the integers. In what follows, we will consider each instruction to have an execution time of one time unit.

Instructions 1 through 7 are exactly the same as their counterparts for ordinary RAMs. Instruction 8 is the same as instruction 1, except that in 8 it is a channel register that is accessed. Instruction 8 changes the value of R_i to the value of channel register C_h^ℓ of the ℓ-th channel bank. Instruction 9 allows a processor (the parent) to initiate its ℓ-th offspring, $1 \leq \ell \leq k$. If that offpsring is currently active, the parent is blocked at that instruction until the offspring returns. When the call instruction is executed, the values of $a_1,a_2,\ldots,a_u$ are copied into the ℓ-th offspring's registers $R_0,R_1,\ldots,R_{u-1}$, respectively. The parent then continues with its computation and, in parallel, the offspring begins executing the program starting with the first instruction. If the parent attempts to access channel register C_h^ℓ or recall offspring ℓ while it is still active, then the parent is blocked at that point in the computation until offspring ℓ completes its computation and returns. Instruction 10 allows an offspring to return an answer to its parent. The operation of this instruction is similar to that of the call instruction, except that information is passed back up to the parent. When offspring ℓ executes RETURN $(a_1,a_2,\ldots,a_u)$, the values of $a_1,a_2,\ldots,a_u$ are loaded into the parent's channel registers $C_1^\ell,C_2^\ell,\ldots,C_u^\ell$, respectively. The offspring and all its descendants are then removed from the computation and are available for additional calls. Instruction 11 is similar to instruction 6. If offspring ℓ is not active at the time instruction 11 is executed, then control transfers to the instruction labeled by LABEL_1. If offspring ℓ is active, then the instruction labeled by LABEL_2 is executed next. An offspring is said to be *active* if it has been called but has not yet returned from that call.

TABLE 2. PRAM INSTRUCTIONS

number	function	instruction format
1	direct assignment	$R_i \leftarrow b$
2	indirect assignment	$R[R_i] \leftarrow b$
3	indirect assignment	$R_i \leftarrow R[R_j]$
4	addition	$R_i \leftarrow a+b$
5	subtraction	$R_i \leftarrow a-b$
6	conditional branch	IF a *COMP* b THEN $LABEL_1$ ELSE $LABEL_2$
7	unconditional branch	GOTO LABEL
8	channel access	$R_i \leftarrow C_h^{\ell}$
9	call to offspring ℓ	CALL $\ell(a_1,a_2,\ldots,a_u)$
10	return to parent	RETURN $(a_1,a_2,\ldots,a_u)$
11	ℓ-return-test branch	IF ℓ RETURNED THEN $LABEL_1$ ELSE $LABEL_2$

The computation of a PRAM proceeds as follows. Initially, there is a single processor active. The input is in register R_0 of this processor and all other registers are set to zero. The PRAM computes in much the same way as an ordinary RAM until a call instruction is executed. When this occurs, an offspring processor is activated. The call parameters are copied into the registers of the offspring as outlined in the description of the call instruction above. All of the offspring's other registers are set to zero. The offspring then starts computing. Both processors remain active and compute in parallel (and, in fact, each processor may make additional calls) until one executes a return instruction. When this occurs, the return parameters are loaded into the channel registers corresponding to the returning offspring and this offspring and all of its descendants are removed from the computation.

An integer x is said to be *accepted* if there is a computation of the k-PRAM on input x such that the computation terminates with the top level processor executing a return instruction. The set accepted by a k-PRAM is the set of all integers accepted by the k-PRAM. A k-PRAM P is said to *accept* the set A of integers *in time bound* T(n) provided that P accepts A and that, for every x in A, there is an accepting computation of P on x that takes at most T(n) steps, where n is the length of x as a binary numeral.

It will be convenient (and sometimes necessary) to assume that our time bounds satisfy some minimum niceness conditions. We say that a function f(n) is T(n) *time countable* provided there is a deterministic ordinary RAM which,

given any input x of length n, will construct the value f(n) in a specified one of its registers within T(n) time units. A function is O(T(n)) *time countable* if it is cT(n) time countable for some c. In what follows we will assume that every time bound function T(n) is O(T(n)) time countable. We also assume that all time bound functions T(n) are such that $T(n) \geq n$, for all n. Most all common time bounds T(n) are O(T(n)) time countable.

6.2. Eliminating nondeterminism

We have observed that a nondeterministic machine can be viewed as a kind of parallel processing machine. Therefore, it should not be suprising to find out that parallelism can be used to eliminate nondeterminism and to do it for only a small time penalty. The next result makes the point formal.

THEOREM 11. *If* A *is accepted by some nondeterministic* k-PRAM *in time* T(n), *then we can find a deterministic* k-PRAM *that accepts* A *in time* $O((T(n))^4)$, *provided* $k \geq 2$.

Theorem 11 says that if we are only interested in getting a polynomial time bounded algorithm and if we have a parallel processing computer available, then it does not matter whether our programs are deterministic or nondeterministic. A time efficient nondeterministic program can always be converted to a time efficient deterministic program with the aid of parallelism.

We have essentially used the uniform cost criteria in computing run times for k-PRAMs. Since the logarithmic cost criteria is more realistic, it is natural to ask if Theorem 11 depends heavily on the use of the uniform cost criteria. It does not. If we use a logarithmic cost criteria we do not appear to get a bound of $O((T(n))^4)$. However, we still do get a bound which is a polynomial in T(n). This should not be suprising, since we have only allowed addition and subtraction as arithmetic operations. As we already noted, if we restrict RAMs to having only these operations, then logarithmic and uniform cost time are polynomially related.

Theorem 11 might seem to hint that parallelism and nondeterminism are equivalent. They are not. Nondeterminism is a very special kind of parallelism and parallelism, in full generality, appears to be more powerful than nondeterminism. The next theorem shows that with just a small amount of additional time, a deterministic parallel machine can do more than a nondeterministic serial machine.

THEOREM 12. *There are sets that can be accepted by* $O(T_1(n))$ *time bounded deterministic* 2-PRAMs*, but cannot be accepted by any* $O(T_2(n))$ *time bounded nondeterministic* RAM*, provided* $\sup_{n\to\infty} T_2(n)/T_1(n) = 0$. *(Here we may use either the uniform of the logarithmic cost criteria for* RAMs*.)*

6.3. Counting processors

Theorem 11 indicates that if we only consider time bounds, then parallel machines are very powerful. This power does come at some cost though. In order to get very fast parallel algorithms we use a large number of processors. Our next result says that if a parallel processor is required to run in polynomial time and required to use only a polynomial number of processors, then it is no more powerful than a serial processor which runs in polynomial time. To make this more precise, let us say that a k-PRAM is U(n) *processor bounded* provided that in any computation on an input of length n, no more than U(n) processors are ever active at the same time.

THEOREM 13. *A deterministic (respectively nondeterministic)* k-PRAM *program which is simultaneously* T(n) *time bounded and* U(n) *processor bounded, can be simulated by a deterministic (respectively nondeterministic)* RAM *using the uniform cost criteria in time* O(U(n)T(n)).

6.4. Time storage trade-off

When studying MRAMs, we saw that for some powerful types of machines polynomial time and polynomial storage are equivalent. We can expand this equivalence to include k-PRAMs as well as MRAMs and obtain the following.

THEOREM 14. *The following statements are equivalent.*

(1) A *is accepted by a nondeterministic polynomial time bounded* k-PRAM.

(2) A *is accepted by a deterministic polynomial time bounded* k-PRAM.

(3) A *is accepted by a nondeterministic polynomial storage bounded* RAM.

(4) A *is accepted by a deterministic polynomial storage bounded* RAM.

(5) A *is accepted by a nondeterministic polynomial time bounded* MRAM *under the uniform cost criteria.*

(6) A *is accepted by a deterministic polynomial time bounded* MRAM *under the uniform cost criteria.*

7. SUMMARY

We have seen that if a given informally stated algorithm is implemented on two or more different formal models for a computer then the different models can yield different, and sometimes vastly different, run times and storage consumption. There are, however, a number of things that remain invariant under change of machine model. Storage is a very stable concept. If we only measure storage up to growth rate, then it does not matter which one of the various models considered we use. As long as all models are deterministic or all models are nondeterministic, they all will have the same storage consumption. Time, on the other hand, is much less stable. However, if we only measure time up to a polynomial relation, then most models are equivalent. In this case, the models seem to fall into two classes. One class contains parallel processing models and the MRAM with the uniform cost criteria. The other class contains all the remaining models. The two classes can then be subdivided into deterministic and nondeterministic models. If we consider only polynomial time bounded programs, then all models in a subclass are equivalent. The relationships between the subclasses can be summarized as follows. Let NPparallel, Pparallel, NP and P denote the class of sets accepted in polynomial time by nondeterministic parallel models, deterministic parallel models, nondeterministic serial models and deterministic serial models respectively. (Consider the MRAM with the uniform cost criteria to be a parallel model.) Then $NP\text{parallel} = P\text{parallel} \supseteq NP \supseteq P$ and the commonly accepted conjecture is that the last two inclusions are strict.

ACKNOWLEDGMENT

This work was supported in part by NSF grant MCS-74-02338.

DEVELOPMENTS IN DATA STRUCTURES

P. VAN EMDE BOAS
University of Amsterdam, Amsterdam, The Netherlands

ABSTRACT

A number of data structures for set manipulation is discussed. We illustrate how set manipulation is involved in the design of algorithms and in which manner the efficiency of an algorithm may be hurted by an ill-chosen representation for the sets dealt with. The choice of a representation mainly depends on the instructions one needs to execute and on the amount of storage available. The paper concludes with a new description of a linear space data structure, supporting the full repertoire of single-set operations with sublogarithmic processing time.

CONTENTS

1. INTRODUCTION 35

2. SETS AND INSTRUCTIONS 36

3. HOW TO REPRESENT A GRAPH - AN OMINOUS EXAMPLE 39

4. ARRAYS 42

5. BINARY SEARCH TREES 44

6. THE BINARY HEAP 48

7. BALANCED TREES 50

8. TRITER TREES 52

9. THE BINOMIAL HEAP 55

10. PRIORITY DEQUES 58

1. INTRODUCTION

Current day mathematics is founded on a set theoretical base. The language of set theory has established itself even in the curricula of primary and secondary schools. Consequently set operations such as union and intersection are household terms and generally considered trivial and noninteresting.

It is a notable fact that the set concept hardly plays any role in the practice of programming. Most programming languages (PASCAL being an exception) have no implemented set feature. The available data structures are arrays, records, files and pointers, and sets have to be simulated in software.

On closer inspection there appears to be an explanation for this phenomenon. The choice how to implement sets using hardware features such as words, bits and addresses heavily depends on the operations one likes to execute, on the presence or absence of an ordering and on whether one or more sets have to be dealt with simultaneously. Time and space restrictions further limit the possible methods of set representation.

In the present paper we demonstrate that the efficiency of algorithms may suffer from an ill-chosen representation of the data used by means of an example concerning graph representations. We discuss a number of data structures for set manipulation, indicating for which operations they are particularly appropriate.

2. SETS AND INSTRUCTIONS

Data in everyday computer practice are usually ordered collections of items such as real numbers, integers, truth values, alphanumeric strings or references to other pieces of data. These composite objects can be implemented using programming constructs like the *record type* of PASCAL or the *structured modes* of ALGOL 68. In general, a special field is used for data identification. This field, which is called the primary key, has either an integral or a string value; in both cases the primary key values are totally ordered. Depending on whether data items are of constant or variable size, the primary key can be used as the address for the data item or as the address for a pointer to the data item. In many cases data can be manipulated by manipulating their primary keys.

In turns out that, in manipulating sets of records, we can assume by way of approximation that we are dealing with sets of primary keys, *i.e.*, totally ordered sets of numbers or strings. Moreover there will be a fixed *a priori* upper bound for the integer value or string length, implying that we are dealing with subsets of a large but finite universe U. We denote the size of the universe by u; thus we may assume that U = {1,...,u} or U = {0,...,u-1}.

The choice of a data structure is not dictated by the size of the universe alone. Another important factor is the expected size of the subsets one has to deal with; this value is denoted by n. Finally there are external restrictions imposed by the limited amount of available storage. These restrictions enforce that at most N elements can be dealt with simultaneously. Clearly $n \leq u$; we also assume that $n \leq N$ since otherwise our task is hopeless right from the beginning.

TABLE 1. OPERATIONS FOR SET MANIPULATION

	no order	order	
single set	INSERT(i)	MIN	MAX
	DELETE(i)	EXTRACT MIN	EXTRACT MAX
	MEMBER(i)	ALL MIN(i)	ALL MAX(i)
	CARD	PREDECESSOR(i)	SUCCESSOR(i)
	EMPTY		
more sets	UNION(A,B,C)	SPLIT(a,S,T)	
	FIND(i)	CONCATENATE(A,B,C)	

In Table 1 we indicate the basic instructions we are going to discuss. The table is divided according to whether a single set or more sets are involved and whether the order on the universe U is involved in the operation or not.

In the above table lower case letters denote elements in U wheras upper case letters denote subsets of U. An instruction may yield a value or not. The types of the values obtained are:

- *boolean* for MEMBER and EMPTY;
- *integer* for CARD;
- elements in U for MIN, MAX, PREDECESSOR and SUCCESSOR;
- subset of U for FIND.

The remaining instructions yield no value.

For most instructions, *e.g.*, INSERT, DELETE, MEMBER, CARD, EMPTY, MIN, MAX, EXTRACT MIN and EXTRACT MAX, the meaning should be clear. The remaining ones perform the following tasks:

- ALL MIN(i) (ALL MAX(i)) removes from the single set manipulated all members $\leq$ i ($\geq$ i);
- PREDECESSOR(i) (SUCCESSOR(i)) computes the largest (smallest) member in the set $\leq$ i ($\geq$ i);
- UNION(A,B,C) unites the sets A and B creating a new set named C, or equivalently C:= A$\cup$B;
- FIND(i) yields the (name of the) subset currently containing element i.

It is always assumed that programs dealing with UNION and/or FIND deal with disjoint sets. A similar assumption holds for SPLIT and CONCATENATE where it is moreover assumed that the sets are pairwise comparable with respect to the ordering. These latter instructions have the following meanings:

- SPLIT(a,S,T) removes from S all entries > a putting them into a new set T;
- CONCATENATE(A,B,C) is equivalent to UNION(A,B,C), provided A $\leq$ B.

It is possible to call the above instructions with improper arguments, *e.g.* MIN on an empty set; we assume that in these circumstances some appropriate action will be executed which we leave unspecified for the moment.

In the sequel we will not discuss the instructions CARD and EMPTY; they are easily implemented using a single counter.

Specific names are attached to data structures supporting particular subsets of the above repertoire. They are indicated in Table 2. The question mark indicates the absence of a well established name for a structure supporting UNION and FIND, a pair which has been investigated thoroughly; see

TABLE 2. SOME TYPES OF DATA STRUCTURES

data structure	instructions
dictionary	INSERT, DELETE, MEMBER
priority queue	INSERT, DELETE, MIN
restricted priority queue	INSERT, EXTRACT MIN
priority deque	all single-set instructions
mergeable heap	INSERT, DELETE, MIN, UNION, FIND
concatenable queue	INSERT, DELETE, FIND, CONCATENATE, SPLIT
?	UNION, FIND

Section 8.

It turns out that some structures are hard to implement under the restriction N << u but are easy if N = u, whereas for other structures the problem remains hard even if N = u. If there is no limit on the amount of storage and initialization time, the complete instruction repertoire can be implemented in such a way that each instruction takes unit time, using a huge precomputed table storing the result of each legal instruction on each legal configuration. For all practical purposes, however, this trivial solution is useless.

All data structures described in this paper will be of size O(u), where storage is expressed in terms of RAM words. The time needed to perform operations is measured in terms of RAM operations with unit time cost (*cf.* [Aho *et al.* 1974]). Except possibly for the priority deques in Section 10, this will correspond to the programmer's intuition.

3. HOW TO REPRESENT A GRAPH - AN OMINOUS EXAMPLE

The question we consider in this section is: how should one store a directed graph within a computer? According to one of the available mathematical definitions, a graph is a pair (V,E) where E is a subset of V×V. Without loss of generality one takes V = {1,...,n}.

One of the representations found in text books on graph theory is the so-called *adjacency matrix*. This is an n×n matrix with entry a_{ij} equal to the number of edges from i to j. This representation allows multiple edges as well.

The adjacency matrix has some useful properties. Absence of self-loops can be detected by inspection of the diagonal entries. Undirected graphs can be represented by symmetric matrices. The matrix is {0,1}-valued if no multiple edges occur. A clique in the graph corresponds to a submatrix J-I (where J represents the all 1-matrix and I represents the identity matrix).

A more interesting connection exists between paths in the graph and algebraic operations on the matrix. If we raise the matrix to the k-th power, the entry at location i,j equals the number of paths of length k from i to j. If one is only interested in presence or absence of paths, one can perform the same matrix operations over the Boolean algebra {0,1} instead of the integers.

An alternative representation is the so-called *adjacency list structure*. For each node i we give a linear list of all nodes j such that <i,j> is an edge in E. This representation has none of the nice mathematical features mentioned above.

Assume that we want to develop an algorithm to decide if a given graph is strongly connected. Starting from the adjacency matrix A we have the following obvious method. The graph is strongly connected if each node can be reached from each other node by some path. Eliminating cycles from this path we may assume that the path length is at most n. Presence or absence of paths of length ≤ n can be detected by raising the matrix I+A to the n-th power over the Boolean algebra {0,1}. Starting from the adjacency list we can develop algorithms based on *depth-first search*. In fact, one of the first described applications of this technique of graph traversal yields an algorithm for testing strong connectedness; see [Aho *et al.* 1974].

So far both representations seem suitable. However, if we consider the complexity of the algorithms involved, we discover an impressive difference. Computation of $(I+A)^n$ requires $\lceil \log_2 n \rceil$ squarings; using Strassen's matrix

multiplication this yields an $O(n^{2.81})$ algorithm [Strassen 1969; Aho *et al.* 1974]. The depth-first search algorithm has a run time of order n+e, where e denotes the number of edges in the given graph.

The reader might ask whether the above gap between $O(n^{2.81})$ and O(n+e) can be narrowed. It might be possible to improve the efficiency of algorithms based on adjacency matrices. However, it can be shown that this improvement will never yield an algorithm which runs faster than $O(n^2)$ in the worst case. The reason is that any algorithm operating on adjacency matrices which tests for strong connectedness can be forced to probe all n^2 entries of the matrix at least once.

The above statement is formalized by investigating the so-called decision tree model for algorithms that recognize graph properties, based upon the presence or absence of edges. The model is best understood by considering the following two-person game. Player A conceives a graph G on the vertices 1,...,n; player B has to determine whether this graph has a particular property P (*e.g.*, strong connectedness) or not. In order to do so B may probe individual pairs <i,j> with $i \neq j$; A tells B whether <i,j> is an edge or not. The game is continued until B has gathered sufficient information to verify the property. B loses if he cannot decide the property before having asked for all n(n-1) feasible pairs.

The best strategy for A is not to select his graph in advance but to develop it during the game. Each question posed by B divides the set of possible graphs in two equal parts; by giving an answer A selects the part on which he wishes to play. As long as the set of possible graphs contains both graphs having the property and graphs not having it, the game is not finished.

The above reasoning indicates that A has a winning strategy in case the total number of graphs having the property is odd, since it is always possible to select the part containing an odd number of graphs having the property; note that the number of possible graphs always is a power of two. This argument can be strengthened as follows. Consider the so-called *weight enumerator* of the property P. This is the polynomial $F(X) = \sum_{e=0}^{n(n-1)} A_e X^e$, where A_e denotes the number of graphs with property P having e edges. A sufficient condition for A to have a winning strategy is that 1+X does not divide F, or equivalently $F(-1) \neq 0$. The latter expression has been subjected to study in enumerative graph theory. In particular for the property "strong connectedness" R.W. Robinson [Robinson 1973] has shown that $F(-1) = (n-1)!$. Consequently A has a winning strategy, showing that any algorithm for testing strong connectedness can be forced to probe all edges.

For which properties can one prove $O(n^2)$ lower bounds for adjacency matrix based algorithms using the tricks explained above? A. Rosenberg conjectured in 1973 that for any *nontrivial graph property* (*i.e.*, a property invariant under relabeling the nodes, which could not be decided from the number of nodes only), such a lower bound would exist. Shortly afterwards S. Aanderaa gave an example of such a property which could be decided in 3n probes. To circumvent this counterexample they added the condition of *monotonicity* (*i.e.*, the property is not disturbed by adding edges) and in this form the conjecture was proved by R.L. Rivest and J. Vuillemin. For further information the reader is referred to [Best *et al*. 1974; Rivest & Vuillemin 1976].

The result shows that for some problems there exists a "natural" lower bound for adjacency matrix based algorithms which can be circumvented by using adjacency lists. The choice of the representation matters indeed.

4. ARRAYS

The most direct way of representing subsets $S \subseteq \{1,\ldots,u\}$ uses the encoding of a set by its characteristic function which is stored in a table. The resulting type is an ALGOL Boolean array or a PASCAL type like *array*[1..u] *of* (*present*,*absent*). Using this representation we have a perfect dictionary; INSERT, DELETE and MEMBER take constant time. The snag is that in general $n \leq N << u$ (these identifiers have been introduced in Section 2); the array uses too much space and the information has to be condensed.

We may store the present elements in a table. The corresponding type then becomes *array*[1..N] *of* 0..u, where 0 denotes that no element is stored. On such a data structure the times needed for executing an INSERT, DELETE or MEMBER instruction become O(1), O(n) and O(n) respectively. If we arrange to have the elements stored in order, we can retrieve an element using binary search but insertions and deletions become or remain expensive, because of the necessity to shift the array in order to create or close gaps. On the other hand MAX and EXTRACT MAX (as well as MIN and EXTRACT MIN when we use the array as a cyclic buffer) can now be executed in constant time.

Replacing the array by a linear list or doubly linked list structure does not improve the situation very much. If no order is preserved, the run times for INSERT, DELETE and MEMBER become O(1), O(n) and O(n) respectively, but if the location of an item to be removed is known, it can be deleted in constant time if a doubly linked list is used. Since binary search cannot be used on a linear list, preserving order will not help; on the other hand MIN, MAX, EXTRACT MIN and EXTRACT MAX take constant time on a doubly linked ordered list.

An alternative approach which preserves the random access behavior of an array is the use of a *hash table*. We use a fixed mapping $g: \{1,\ldots,u\} \rightarrow \{1,\ldots,N\}$, which tells where an element of U should be stored in the *array* [1..N] *of* 0..u. However using the function g we can condense U into the array only because of the fact that g is not 1-1, leaving the possibility that two different items have to be stored at the same location. An occurrence of two items $x \neq y$ with $g(x) = g(y)$ is called a *collision*. There are several ways of dealing with collisions. One can create an external overflow area which can be organized into a family of linear lists, most of which will be empty. An alternative is to store the collided items into the hash table on locations reserved for still other items. For example, one selects a number, called *stride*, coprime to N, to be used as follows. If one detects

that the location g(x) is already used, inspect the locations g(x)+stride, g(x)+2*stride,... (all mod N) until a free location is found, and store x at this place. On a subsequent lookup for x the same path through the table is traversed. If the insertion fails, this means that the table is full and the insertion is impossible. The hashing technique described above does not allow for items to be removed.

If measured according to worst-case behavior, a hash table behaves badly (all items may collide), but its expected run time renders it a widely used data structure.

5. BINARY SEARCH TREES

A binary search tree is a labeled tree whose internal nodes are labeled by elements in the set S whereas the leaves represent intervals of elements outside S (the gaps in S). For the universe one uses an ordered set; binary search trees can also be used for storing finite sets with real valued primary keys. Figure 1 gives an example of a binary search tree, where the universe U is the set {1,...,49}.

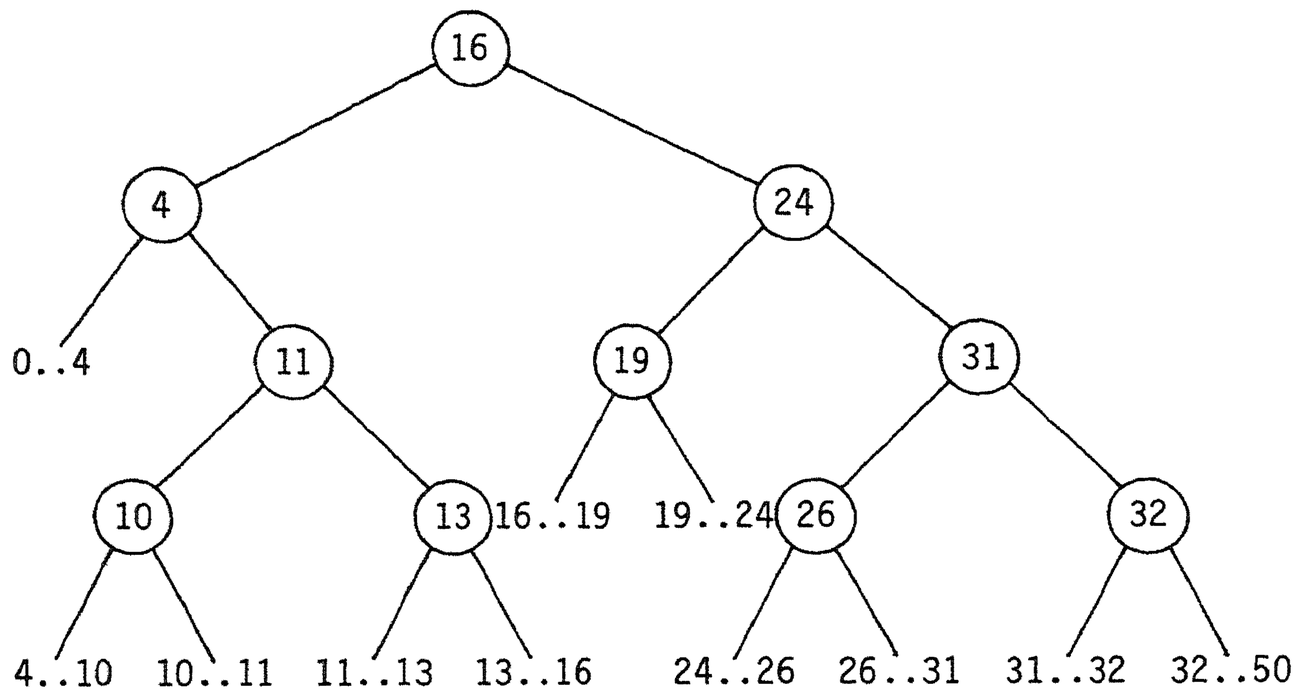


Figure 1 A binary search tree.

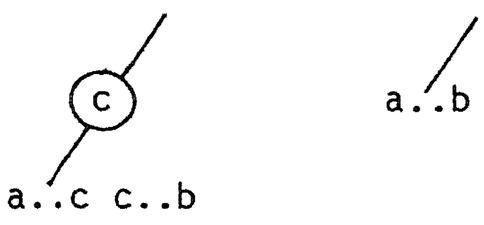
The order relation in a binary search tree is that all labels of a left-hand (right-hand) subtree are smaller (greater) than the label of a node. The intervals labeling a leaf are determined as follows: the left-hand (right-hand) endpoint of the interval labeling a right-hand (left-hand) leaf is the label of its direct father; the other label is the label of the lowest ancestor of which it is a left-hand (right-hand) descendant, or u+1 (0) if such an ancestor does not exist. Internal nodes having two, one or no leaves as direct sons are called *boundary*, *semi-boundary* and *interior* nodes respectively.

On a binary search tree the instructions INSERT, DELETE, MEMBER, MIN and MAX can be executed in time O(k), where k is the longest path length in the tree inbetween a leaf and the root. The expected run time is proportional to the expected path length; the expected time may moreover depend on probability weights given to the internal nodes and leaves.

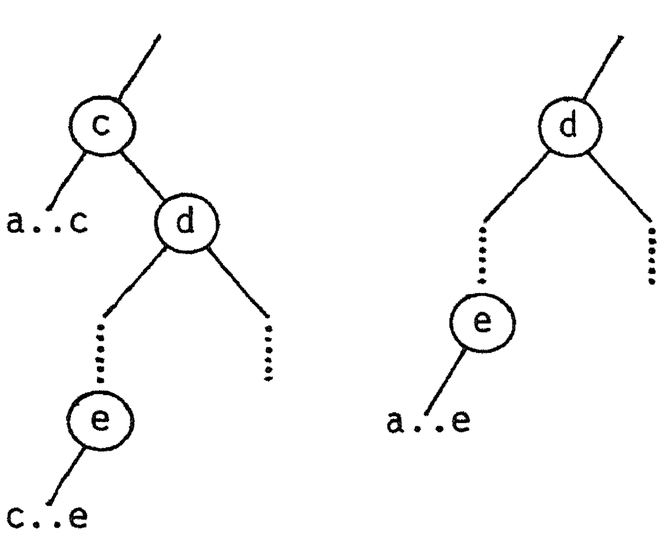
Most instructions proceed by a top to bottom search guided by the labels

of the internal nodes. If a new element is inserted, a new internal node is created at the location of the leaf labeled by the gap into which the new element is situated; its two leaf sons are labeled by the two parts into which the gap is decomposed.

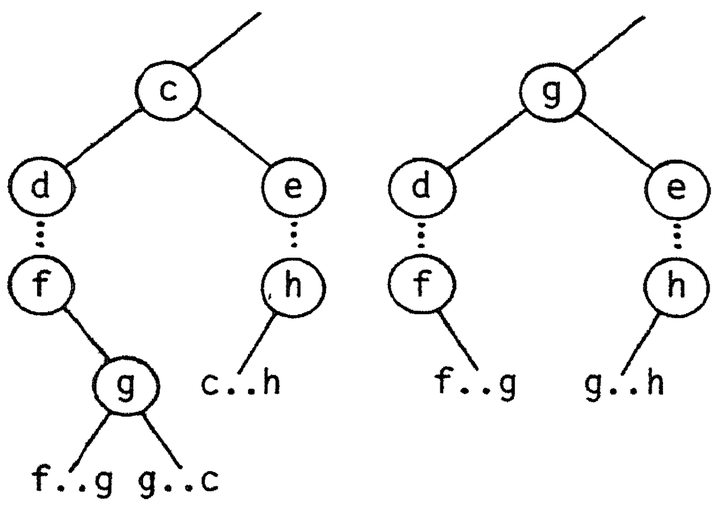
The most complex instruction is DELETE. Deleting a boundary node is an easy reverse to the insert described above. A semi-boundary node with a right-hand (left-hand) leaf is deleted by replacing it by its left-hand (right-hand) subtree, merging the interval which labeled its original leaf with the interval labeling the right most (left most) leaf of this subtree. An interior node is deleted by replacing it by the largest internal node of



(a) Deleting a boundary node.



(b) Deleting a semi-boundary node.



(c) Deleting an interior node.

Figure 2

its left-hand subtree (this is always a boundary or semi-boundary node), deleting the latter node and updating the left most leaf of its right-hand subtree. The effect of these deletes is illustrated in Figure 2.

In order to reduce the run time it is advantageous to keep the tree *balanced*, *i.e.*, the average and maximal path lengths should be as equal as possible. Note that the longest path length may vary inbetween $\lceil \log n \rceil$ and n. In case the structure of a binary search tree remains fixed (no insertions or deletions), one may ask for optimal binary search trees, which should be based on expected probabilities for elements in or outside S to be asked for.

A dynamic programming algorithm which computes an optimal binary search tree in time $O(n^2)$ is given by D.E. Knuth [Knuth 1973A]; he also describes an algorithm due to Hu and Tucker which gives for the special case that elements in S are never asked for an optimal search tree in time O(n log n) and space O(n).

Instead of asking for an optimal tree one may produce a suboptimal search tree using a linear time algorithm based upon a heuristic rule which yields a result with efficiency within a constant factor of the optimal tree. Two of these rules are:

- the *weight balancing rule* (Gottlieb and Walker): make the weights of the left- and right-hand subtree as equal as possible;
- the *min-max rule* (Bayer and Schnorr): minimize the maximal weight of the left- and right-hand subtree.

The result of these rules is illustrated in Figure 3. Consider the four element set $S = \{s_1, s_2, s_3, s_4\}$ with the following weights (all multiplied by 24):

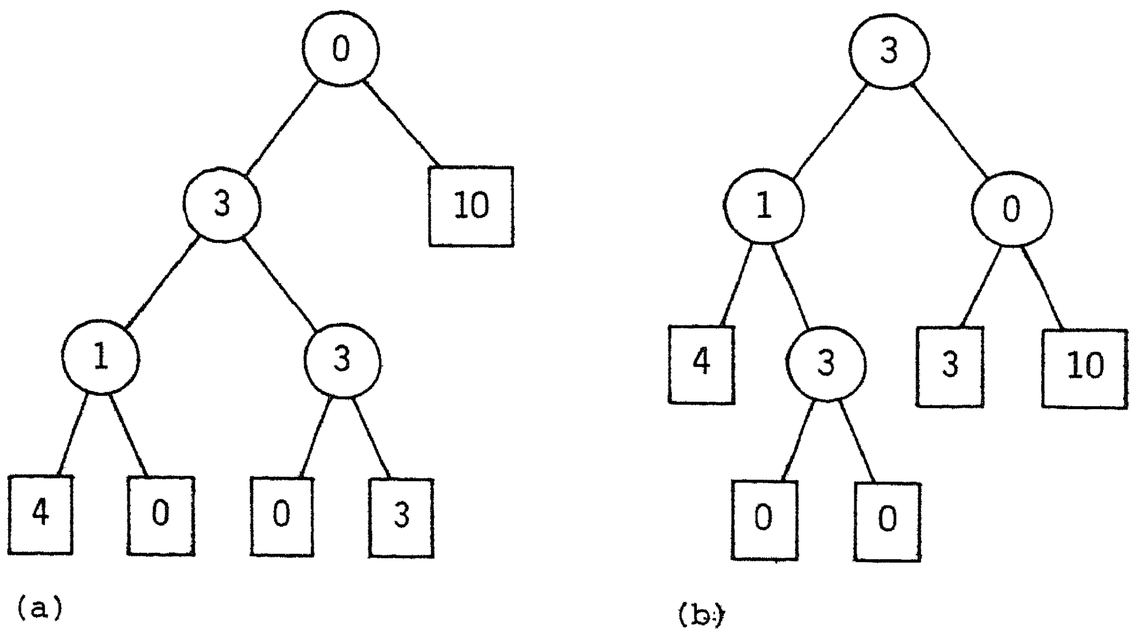


Figure 3 Binary search trees constructed using heuristic rules.

$\underline{x} < s_1$: 4, $\underline{x} = s_1$: 1, $s_1 < \underline{x} < s_2$: 0, $\underline{x} = s_2$: 3, $s_2 < \underline{x} < s_3$: 0, $\underline{x} = s_3$: 3, $s_3 < \underline{x} < s_4$: 3, $\underline{x} = s_4$: 0 and $\underline{x} > s_4$: 10. Using the weight balancing rule one obtains the tree of Figure 3(a) with average path length 49/24. The min-max rule yields the tree of Figure 3(b) with average path length 2. It turns out that the latter tree is optimal.

For further information the reader is referred to [Knuth 1973A; Güttler *et al.* 1976].

The still more difficult problem of preserving suboptimality within dynamically changing binary search trees has been investigated only recently. A solution with a search time within a constant factor of the optimum and an updating time of the same order has been given by K. Mehlhorn [Mehlhorn 1977].

6. THE BINARY HEAP

The binary heap is the data structure underlying the heapsort algorithm. It is a restricted priority queue supporting the INSERT and EXTRACT MIN instructions in time O(log n). The structure moreover is stored within an array using as much space as there are elements in the set.

Consider an array a[1..N]. This array becomes an optimally balanced binary tree by defining the element with index i to be the father of the elements with index 2*i and 2*i+1 (as long as these indices are within the range 1,...,N). Hence a[1] becomes the root of the tree. Thus the tree is numbered from top to bottom with each layer numbered from left to right, as shown in Figure 4.

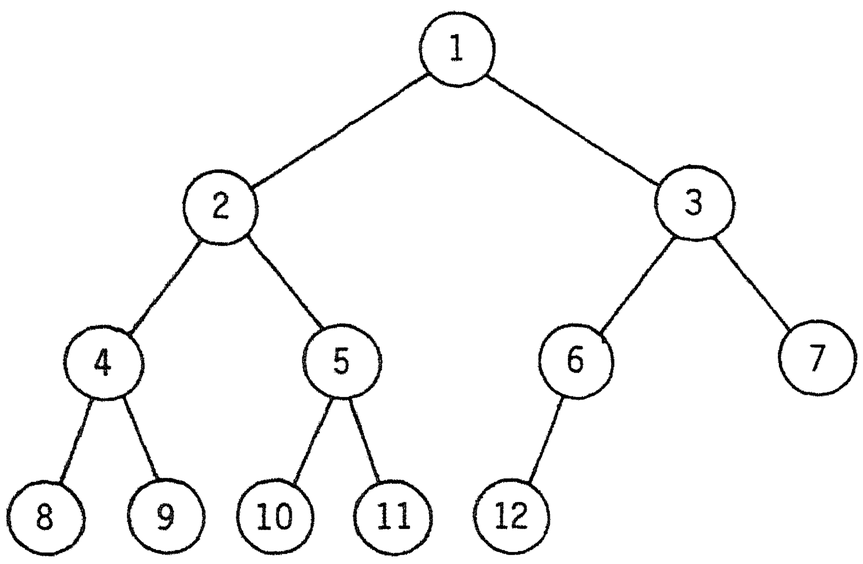


Figure 4 A twelve element array representing a binary heap.

The crucial condition turning the array into a binary heap is the ordering on the paths: the element stored at a node should be less than the elements stored at its sons (if present). Consequently the least element is stored at the root.

Manipulating a binary heap requires this ordering to be preserved. We always assume that an initial part of the array is used; the identifier n stores the number of entries in the heap, and it is increased and decreased when elements are inserted and deleted.

To insert an element x we increase n by one and store x into a[n]; next x is interchanged with the element stored at its father as long as the latter element is larger than x. When this process stops (or when the root is reached), a binary heap is obtained.

To extract the least element we remove the element a[1], replace it by

a[n] and decrease n by one. Next we have to interchange the element at the root with its smallest son, proceeding downwards as long as it is larger than this son. When this stops (or when a leaf is reached), a binary heap is obtained.

The two processes are formalized in the following ALGOL like programs. Other, less transparent, presentations can be found in [Knuth 1973A; Aho *et al.* 1974].

```
proc insert(x):
     begin n:= n+1; a[n]:= x; j:= n;
          while if j = 1 then false else a[j÷2] > a[j] fi
          do interchange(a[j],a[j÷2]); j:= j÷2 od
     end #insert#;

function extract min:
     begin extract min:= a[1]; a[1]:= a[n]; n:= n-1; j:= 1;
          while if 2*j > n then false
                else k:= if 2*j = n then n
                         elif a[2*j] ≥ a[2*j+1] then 2*j+1 else 2*j fi;
                     a[j] > a[k]
                fi
          do interchange(a[j],a[k]); j:= k od
     end #extract min#;
```

Clearly the instructions INSERT and EXTRACT MIN suffice to sort an array. It should be noted however that no additional work space is needed; the unsorted array can be reordered into a heap and be transformed into a sorted array afterwards. In this way the heapsort algorithm is obtained which is given below. Note that this algorithm is far from optimal; each call of insert(a[j]) loads the element a[j] into itself! For improvements see [Knuth 1973A].

```
proc sort(m):
     begin n:= 1;
          for j from 2 to m do insert(a[j]) od;
          for j from m by -1 to 2 do a[j]:= extract min od
     end #sort#;
```

7. BALANCED TREES

Balanced trees enforce some rules on the branching orders, in order to have path lengths approximately equal to O(log n). Several schemes for achieving this goal have been proposed; we mention the AVL trees, mentioned after their proposers Adelson, Vilenski and Landis, the 1-2 trees, proposed by Maurer, Wood, Ottman and Six, and the 2-3 trees, described in [Aho *et al.* 1974]. For further references see [Van Leeuwen 1976]. By way of example we describe the 2-3 trees.

In a 2-3 tree all leaves have the same level. Each internal node has two or three descendants. As a consequence the number of leaves of a height-k tree lies in range $2^k \leq m \leq 3^k$ and conversely a 2-3 tree with m leaves has a height in the range $\log_3 m \leq k \leq \log_2 m$.

The leaves of a 2-3 tree are used for storing elements; the internal nodes may be used to store additional information like the number of descendant leaves and the largest and smallest element stored at a descendant leaf.

Manipulating 2-3 trees requires the preservation of their structural properties. We describe the INSERT and DELETE instructions. Although these instructions deal with elements, we generalize their meaning in order to be capable of inserting and deleting subtrees at any level in the tree. The instructions all operate locally, considering a node with its father and (sometimes) its uncles. In some circumstances the operation may require a recursive call one level upwards; if the recursion goes up to the root, the height of the tree may be increased or decreased by one.

To insert a node x at father y, x is made a son of y. If y now has four sons, y is "splitted" and the newly created uncle is inserted as a brother of y. If the root is splitted, a new root is created with the two "roots" as direct sons.

To delete a node x at father y, x is removed. If y now has less than two sons, we inspect the remaining uncles of x. If one of the uncles is "rich" and has three sons, y adopts one of his nephews; otherwise y's unique son is adopted by one of his poor uncles and y is deleted. If y however is the root, then his unique son becomes the root and y is deleted.

Formal descriptions of these algorithms are given in [Aho *et al.* 1974]. Their operations are illustrated in Figures 5 and 6.

With various additional internal information the 2-3 trees have been used to obtain dictionaries, mergeable heaps and concatenable queues with O(log n) instruction time. Details can be found in [Aho *et al.* 1974].

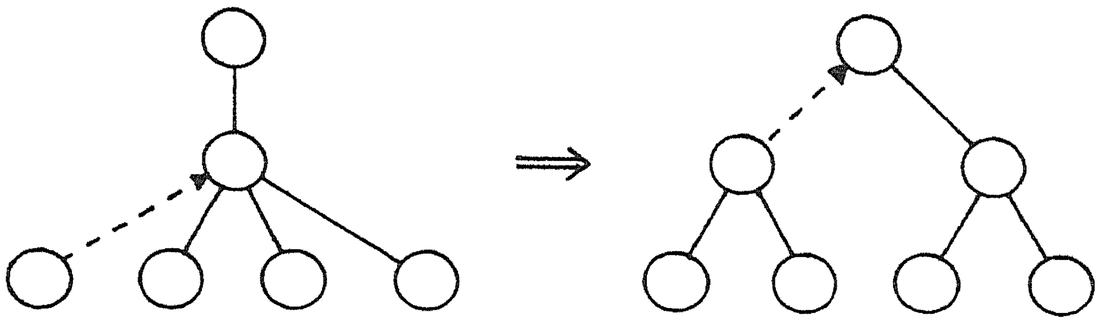
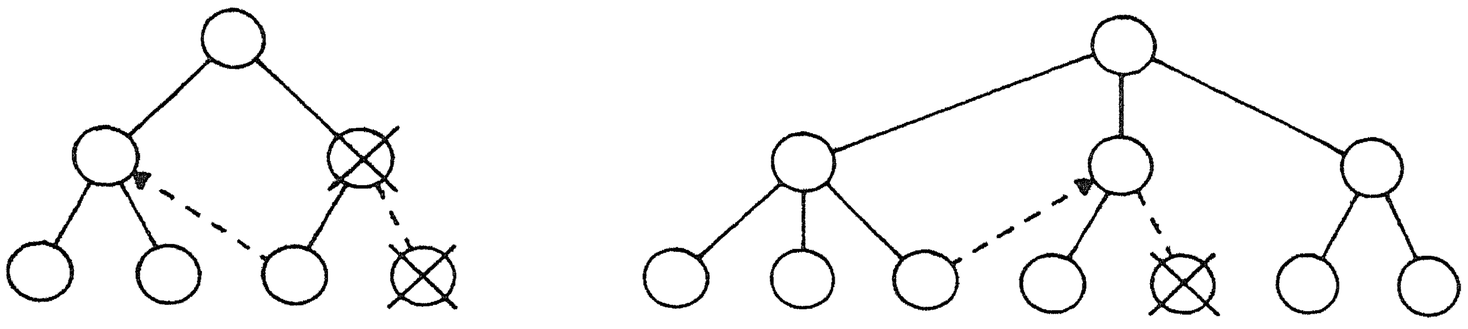


Figure 5 Inserting a fourth son.



(a) with poor uncle.

(b) with rich uncle.

Figure 6 Deleting a second son.

8. TRITER TREES

In Triter trees the edges are directed from son to father, the root being chracterized by the absence of an outgoing edge. There is no bound on the number of sons a father may have.

The most important application of Triter trees is to support the UNION-FIND repertoire. A set is represented by gathering all elements belonging to the set in a tree, using the root of the tree as a location for storing the "name" of the set. A UNION instruction can be executed by making the root of one set a son of the root of the other set. A FIND instruction is executed by traversing the path of the node looked for to the root of the tree to which it currently belongs. The time needed to execute a UNION instruction is constant; the time for executing a FIND instruction is proportional to the length of the path traversed.

Without further precautions it is clear that the above algorithms may lead to trees with path lengths of order n, leading to an O(nm) run time for programs consisting of n UNION and m FIND instructions. This time can be reduced using two different tricks, called *balancing* and *path compression*.

The balancing rule states that in performing a UNION instruction the root of the tree having the smaller number of elements is made a son of the other tree. In this way one enforces that a tree which has a descendant at the k-th level has an offspring of size at least 2^k. The maximal path length now becomes O(log n).

The rule of path compression states that during execution of a FIND instruction all nodes on the path between the node looked for and the root are made direct sons of the root. This does not reduce the time needed for executing the current FIND instruction (actually it requires the path to be traversed twice), but it reduces the time needed for future FIND instructions.

Estimating the efficiency gained using path compression is far from trivial. It has been shown by M.S. Paterson [Paterson -] that the total run time needed for execution of n UNIONs and m FINDs using path compression without balancing is of order m log n. If both path compression and balancing are used, upper and lower bounds of order $m \cdot \alpha(m,n)$ have been proved by R.E. Tarjan [Tarjan 1975A]. In this expression α denotes a functional inverse of a function A related to the Ackermann function. The function α grows unboundedly but slower than any primitive recursive function. The formal definition of A reads as follows:

$A(0,m) = 2m;$

$A(i+1,0) = 0;$

$A(i+1,m+1) = A(i,A(i+1,m))$.

Consequently $A(1,m) = 2^m$ and $A(2,m) = 2^{2^{\cdot^{\cdot^{2}}}}$ (m−1 twos). The definition of α becomes:

$$\alpha(m,n) = \min\{k \mid A(k,4\lceil m/n\rceil) \geq \log n\}.$$

An impression of the huge growth of the function A may be obtained by inspection of Table 3. The numbers in the table preceded by periods denote entries from which only the least significant digits are given. It turns out that they are easier to compute than the most significant ones; *cf.* [Labbers 1976].

TABLE 3. FRAGMENT OF AN ACKERMANN TABLE

i↓ m→	0	1	2	3	4	5
0	0	2	4	6	8	10
1	1	2	4	8	16	32
2	1	2	4	16	65536	..56736
3	1	2	4	65536	..48736	..48736
4	1	2	4	..48736	..48736	..48736

The UNION-FIND algorithm with balancing and path compression is an example of a concrete algorithm having a nonprimitive recursive run time complexity.

The same $O(m \cdot \alpha(m,n))$ time bound shows up at an increasing number of seemingly unrelated problems. This occurs because of the following generalization given in [Tarjan 1975B].

Assume that in a forest of Triter trees the edges are labeled by values from a semigroup F. For each node on which a FIND instruction is executed one would like to compute the product of the labels of the edges on the path traversed. In performing path compression one replaces the label of the edge by the product of the edge labels along the path segment of which the new edge becomes a shortcut; note that these products can easily be computed during the downward scan of path compression. See also Figure 7.

In this application it is difficult to keep the trees balanced. The balancing requires that the root of a small tree is made a son of the root of a larger one also in the case that the instruction uniting the two trees actually requires to link the two trees in the opposite direction. If each

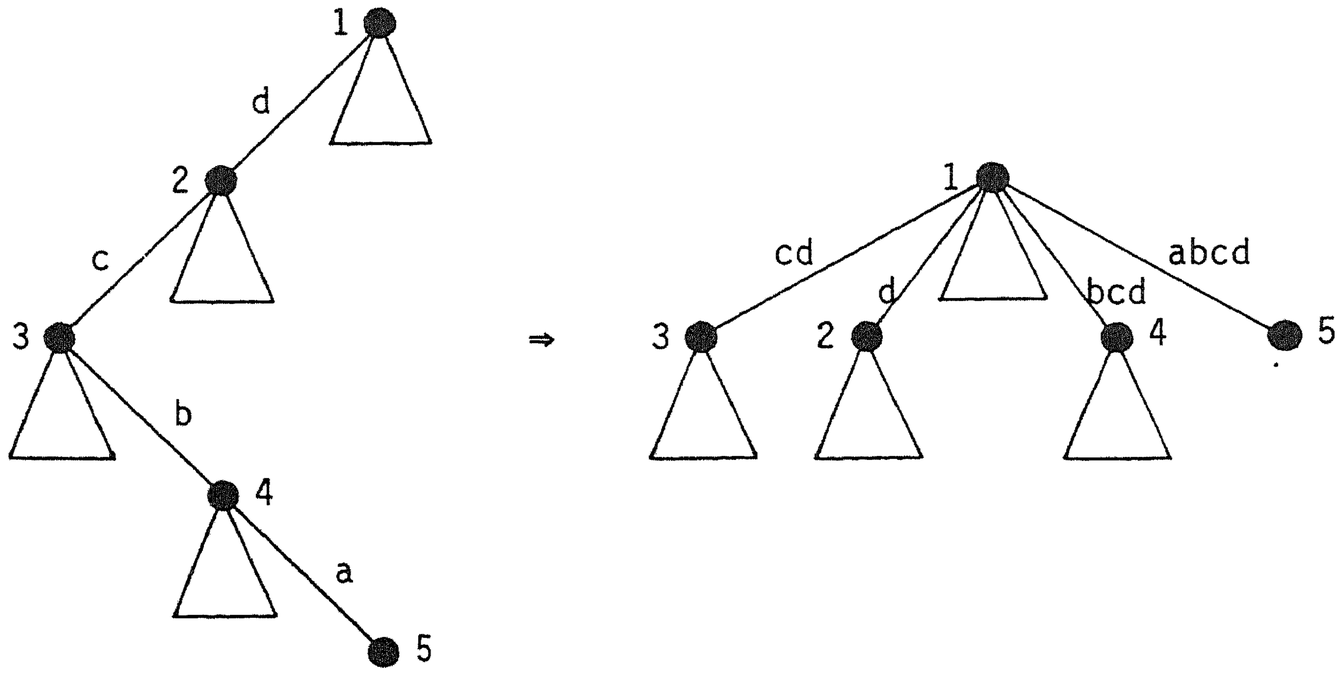


Figure 7 Path compression on a Triter tree with edge labels.

element in F has an inverse, this change of direction is resolved by replacing the edge label by its inverse. For some other concrete semigroups Tarjan has obtained $O(m \cdot \alpha(m,n))$ algorithms by rather complicated methods as well.

9. THE BINOMIAL HEAP

The binomial heap is a data structure designed by J. Vuillemin [Vuillemin 1978] for implementing a mergeable heap, *i.e.*, the repertoire MIN, INSERT, EXTRACT MIN and UNION.

The structure is based on the so-called binomial trees B_k. These are defined inductively as follows: a single node forms a B_0, and two copies of a B_k with an additional edge between the two roots form a B_{k+1}; see Figure 8. It turns out that a B_k-tree may also be described as a root with k sons being roots of a $B_{k-1}, B_{k-2}, \ldots, B_0$ respectively; see Figure 9.

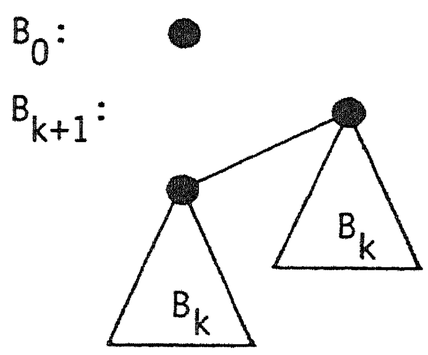


Figure 8 Definition of binomial trees.

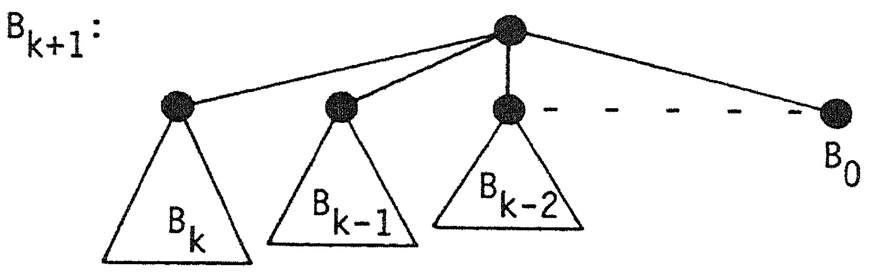


Figure 9 An alternative decomposition of a binomial tree.

Each B_k consists of 2^k nodes. In order to represent n elements one expresses n in binary and collects a B_k-tree for every k for which the k-th binary digit in the representation of n equals 1. The resulting forest is denoted by F_n.

The binomial heap is obtained from F_n by enforcing the heap ordering which we have seen before in the binary heap (Section 6): the element stored at the father is always less than all elements stored at the sons. As a consequence the least element always resides at one of the roots, and since there are at most log n binomial trees in an F_n-forest locating this element takes time O(log n).

The crucial operation for the binomial heap is UNION. First note that

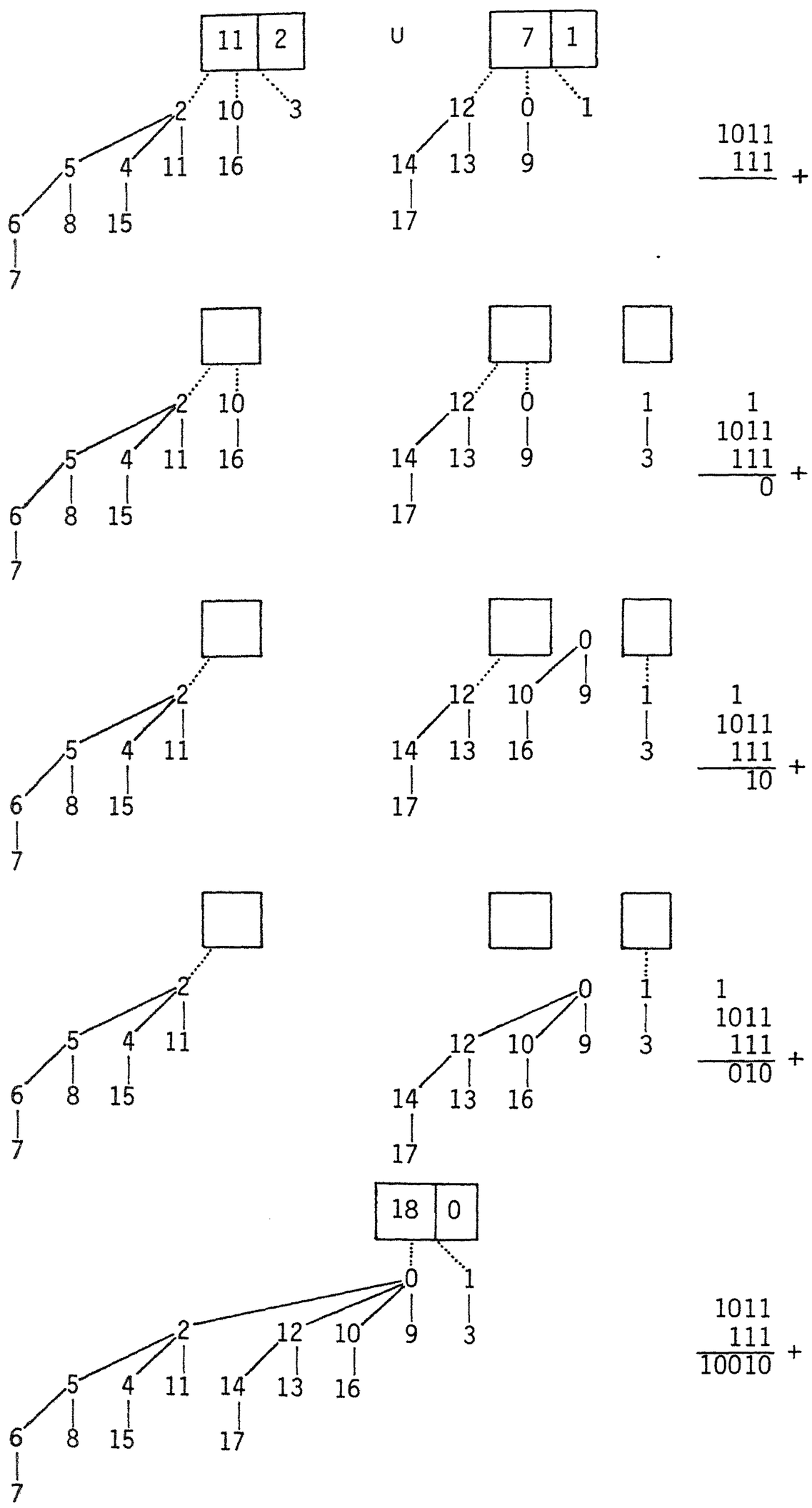


Figure 10 Union of an F_{11} and an F_7.

two heap ordered B_k-trees can be combined into a B_{k+1}-tree in constant time by making the root of the B_k-tree with the largest element stored at the root a son of the other root. Using this operation as a basic step the union of two binomial forests can be performed like a binary addition in time O(log n); see Figure 10.

INSERT is implemented by a UNION of a binomial forest F_n with a single element forest F_1 consisting of a single B_0-tree. Careful analysis shows that the actual time spent during a UNION is proportional to the number of new edges created, which again equals the number of carries processed during the corresponding binary addition. As a consequence a sequence of k INSERTs in an F_n is about as complex as a corresponding sequence of k incrementations of the binary number n; this latter time is easily seen to be about linear in n - an impressive improvement over the O(log n) estimate for a single UNION.

EXTRACT MIN is implemented by first locating the B_k-tree with the smallest root, decomposing it into an F_{2^k-1} by deleting this root and uniting this new binomial forest with the remainder of the original one. This takes again time O(log n). In fact it can be shown that an arbitrary element from a binomial heap can be deleted in time O(log n) provided its location within the forest is completely known.

On base of a practical implementation of the binomial heap it has been shown by M. Brown [Brown 1978] that the binomial heap is not only an asymptotically optimal structure but among the structures achieving the same O(log n) bound the one with the smallest constant factor.

10. PRIORITY DEQUES

In this final section we describe a rather complex data structure developed by the author. On this structure the complete instruction repertoire for single-set manipulation is supported with an O(log log u) processing time per instruction. The name "priority deque" is due to Knuth [Knuth -].

The structure is based upon a simple divide-and-conquer scheme. Assume $u = k \cdot m$. We divide the universe $\{0,\ldots,u-1\}$ into a cluster of k galaxies each of size m. So element x becomes element x mod m in the galaxy with index $\lfloor x/m \rfloor$. We denote these expressions by:

```
head x = ⌊x/m⌋;
tail x = x mod m;
y conc z = y*m+z.
```

Each galaxy in the cluster is represented by a separate priority deque of size m, whereas another priority deque represents the set of nonempty galaxies in the cluster.

It is easy to see that each instruction in the single-set manipulation repertoire can be decomposed into similar instructions operating on individual galaxies and the cluster. For example the INSERT instruction becomes:

```
proc insertun(x):
     begin y:= head x; t:= tail x;
          if emptygal(y)
          then insertgal(y,t); insertcl(y)
          else insertgal(y,t)
          fi
     end #insertun#;
```

Other instructions can be decomposed similarly:

```
function minun:
     begin y:= mincl; minun:= y conc mingal(y)
     end #minun#;

function successorun(x):
     begin y:= head x; t:= tail x;
```

```
if t > maxgal(y)
then z:= successorcl(y+1); successorun:= z conc mingal(z)
else successorun:= y conc successorgal(y,t)
fi
    end #successorun#;
```

It turns out that each operation on the universe level requires at most two calls of similar operations at galaxy or cluster level. A divide-and-conquer scheme based on a recursive decomposition with $k = m = \sqrt{u}$ would yield an O(log u) implementation. However it can be concluded from inspection of the programs that in those cases that two inner calls are required one of them is always simple, *i.e.*, inserting a first element or deleting the last one. For example, the INSERT instruction is decomposed as follows:

```
proc insertun(x):
    begin
        if emptyun then firstinsertun(x)
        else y:= head x; t:= tail x;
             if emptygal(y)
             then firstinsertgal(y,t); insertcl(y)
             else insertgal(y,t)
             fi
        fi
    end #insertun#;
```

If we can construct an implementation where "trivial" operations like first-insert and lastdelete take constant time, we can obtain a divide-and-conquer scheme yielding a recurrence relation of the type $T(u) \leq T(\sqrt{u})+C$, which has an O(log log u) solution.

A nasty problem in designing this implementation is the realization of the operations head, tail and conc, which are required by most of the operations. Using the definition in terms of multiplication and division is not a legitimate solution, since these instructions are not available at unit time cost in the ordinary RAM model. Moreover the introduction of these operations with unit time cost may lead to an unrealistic machine model as has been shown in [Hartmanis & Simon 1976].

In [Van Emde Boas *et al.* 1977] we have described a concrete implementation which results from unwinding the recursion in the proposed data struc-

ture. The operations head, tail and conc are replaced by a collection of pointers which requires storage of size O(u log log u) RAM words.

Consider a universe of size $u = 2^{2^h}$. We represent it using the leaves of a huge binary tree of height 2^h. The root represents the universe. The $\sqrt{u} = 2^{2^{h-1}}$ internal nodes at height 2^{h-1} represent the galaxies; they are the root of a subtree with $\sqrt{u}$ leaves, and also the leaves of a subtree with $\sqrt{u}$ leaves and having the original root as top.

The complete tree is called a canonical tree of rank h. It is decomposed into a top subtree and $2^{2^{h-1}}$ bottom subtrees of rank h-1. These subtrees are decomposed analogously into canonical subtrees of rank h-2, etc. Canonical subtrees of rank 0 consist of three nodes: a father with two sons.

In each node we store h pointers $p_0,\ldots,p_{h-1}$, such that p_j points to the root of the unique canonical subtree of rank j which contains this node and for which this node is not the root. At the root of the complete tree these pointers are undefined. The collection of these pointers requires the use of log log u RAM words for each node, leading to the O(u log log u) storage requirements for the complete structure. These pointers are used to jump in a single step to the node at the halfway level inbetween a leaf and the root of some canonical subtree.

Aside from the pointers each node contains a fixed number of additional storage locations, used to represent subsets $S \subseteq \{1,\ldots,u\}$. The basic idea is that as long as (within a particular hierarchical level, corresponding to a canonical subtree) there is a single entry to be represented, its identity is stored at the root of this canonical subtree and all internal information is "clean". If however more entries have to be represented, the corresponding nodes at halfway level become active as well, in this way activating the next level of the hierarchical decomposition. This idea is illustrated in Figure 11.

The data structure contains next to the stratified tree a doubly linked

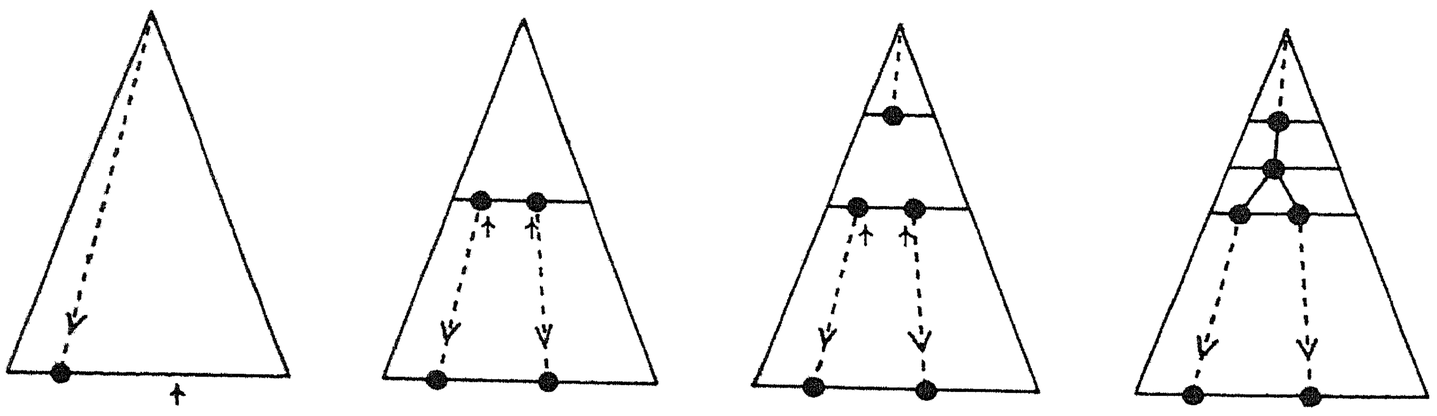


Figure 11 Inserting a second element in a stratified tree.

list of all elements in S, which is used for preserving order. For further details the reader is referred to [Van Emde Boas *et al*. 1977].

It has been shown afterwards [Van Emde Boas 1977] that the above structure can be used to obtain an implementation for a priority deque with O(log log u) processing time and O(u) space. This implementation is obtained by performing one more cluster-galaxy decomposition and using different structures for the galaxy and the cluster.

For the cluster we take the O(log log k) time O(k log log k) space implementation mentioned above. The galaxies are implemented using an O(m) time O(m) space structure, *e.g.*, an unsorted list. Taking $u = k \cdot m$ with $m = \log \log k$ we obtain the following estimates:

time: $O(\log \log k) + O(m) = O(\log \log k) = O(\log \log u)$;
space: $O(k \log \log k) + k \cdot O(m) = O(k \log \log k) = O(u)$.

Knuth [Knuth -] has proposed to use a recursive data structure instead of unwinding the recursion. The structure may be expressed by the following mode description:

```
mode deque = struct(int card,min,max,
                    ref deque cluster,
                    ref [] deque galaxies);
```

Using multiplication and addition for the realization of the head, tail and conc instructions, the resulting space requirements become O(u) bits. The forbidden instructions can be eliminated by introduction of tables for the functions head and tail and for the multiples of m needed for computing conc. Since we need these operations at each level in the recursion, we need log log u tables, but their sizes are of respective orders u, $\sqrt{u}$, $\sqrt{\sqrt{u}}$, etc., and therefore the total storage requirements are O(u) RAM words. These tables can be precomputed in time O(u) without use of forbidden instructions by simple counting.

COMPUTATIONAL COMPLEXITY OF DISCRETE OPTIMIZATION PROBLEMS

J.K. LENSTRA

Mathematisch Centrum, Amsterdam, The Netherlands

A.H.G. RINNOOY KAN

Erasmus University, Rotterdam, The Netherlands

ABSTRACT

Recent developments in the theory of computational complexity as applied to combinatorial problems have revealed the existence of a large class of so-called *NP-complete* problems, either *all* or *none* of which are solvable in polynomial time. Since many infamous combinatorial problems have been proved to be NP-complete, the latter alternative seems far more likely. In that sense, NP-completeness of a problem justifies the use of enumerative optimization methods and of approximation algorithms. In this paper we give an informal introduction to the theory of NP-completeness and derive some fundamental results, in the hope of stimulating further use of this valuable analytical tool.

CONTENTS

1. INTRODUCTION	65
2. CONCEPTS OF COMPLEXITY THEORY	67
3. NP-COMPLETENESS RESULTS	70
3.1. SATISFIABILITY	70
3.2. CLIQUE, VERTEX PACKING & VERTEX COVER	71
3.3. SET PACKING, SET COVER & SET PARTITION	73
3.4. DIRECTED & UNDIRECTED HAMILTONIAN CIRCUIT	75
3.5. 0-1 PROGRAMMING, KNAPSACK & 3-PARTITION	77
3.6. 3-MACHINE UNIT-TIME JOB SHOP	80
4. CONCLUDING REMARKS	83
ACKNOWLEDGMENTS	85

1. INTRODUCTION

After a wave of initial optimism, integer programming soon proved to be much harder than linear programming. As integer programming formulations were found for more and more discrete optimization problems, it also became obvious that such formulations yielded little computational benefit. To this day, general integer programming problems of more than miniature size remain computationally intractable.

For some specially structured problems, however, highly efficient algorithms have been developed. Network flow and matching provide well-known examples of problems that are *easy* in the sense that they are solvable by a *good* algorithm - a term coined by J. Edmonds [Edmonds 1965A] to indicate an algorithm whose running time is bounded by a polynomial function of problem size. This notion is not only theoretically convenient, but is also supported by overwhelming practical evidence that polynomial-time algorithms can indeed solve large problem instances very efficiently; the polynomial involved is usually of low degree. For example, in a network on v vertices a maximum flow can be determined in $O(v^3)$ time [Dinic 1970; Karzanov 1974; Even 1976] and a maximum weight matching can be found in $O(v^3)$ time [Gabow 1976; Lawler 1976B].

It is commonly conjectured that no good algorithm exists for the general integer programming problem. A similar conjecture holds with respect to many other combinatorial problems that are *notorious for their computational intractability* [Johnson 1973], such as graph coloring, set covering, traveling salesman and job shop scheduling problems. Typically, all optimization methods that have been proposed so far for these problems are of an enumerative nature. They involve some type of backtrack search in a tree whose depth is bounded by a polynomial function of problem size. In the worst case, those algorithms require superpolynomial *(e.g.*, exponential) time.

For the time being, we shall loosely denote the class of all problems solvable in polynomial time by P and the class of all problems solvable by polynomial-depth backtrack search by NP. It is obvious that $P \subset NP$.

The battle against hard combinatorial problems dragged on until S. Cook [Cook 1971] and R.M. Karp [Karp 1972] showed the way to *peace with honor* [Fisher 1976B]. They exhibited the existence within NP of a large class of so-called NP-complete problems [Knuth 1974] that are equivalent in the following sense:

- none of them is known to belong to P;

- if one of them belongs to P, then all problems in NP belong to P, which would imply that $P = NP$.

NP-completeness of a problem is generally accepted as strong evidence against the existence of a good algorithm and consequently as a justification for the use of enumerative optimization methods such as branch-and-bound or of approximation algorithms. By way of examples, even restricted versions of all hard problems mentioned above are NP-complete.

NP-completeness theory has proved to be an extremely fruitful research area. The computational complexity of many types of combinatorial problems has been analyzed in detail. Under the assumption that $P \neq NP$, this analysis often reveals the existence of a sharp borderline between P and the class of NP-complete problems that is expressible in terms of natural problem parameters. A truly remarkable feature of the theory is the large proportion of time in which a given problem in NP can be shown to be either in P or NP-complete. Moreover, the two types of problems really have proved to be quite different in character. As mentioned, extremely large instances of problems in P are efficiently solvable, whereas only relatively small instances of NP-complete problems admit of solution by tedious enumerative procedures. Establishing NP-completeness of a problem provides important information on the quality of the algorithm that one can hope to find, which makes it easier to accept the computational burden of enumerative methods or to face the inevitability of a heuristic approach.

In this paper we shall not attempt to present an exhaustive survey of all NP-completeness results (see [Karp 1972; Karp 1975A; Garey & Johnson 1978A]). Instead, we shall examine some typical NP-complete problems, demonstrate some typical proof techniques and discuss some typical open problems (*cf.* [Aho *et al.* 1974; Savage 1976; Reingold *et al.* 1977]). We hope that as a result the reader will be stimulated to consider the computational complexity of his of her favorite combinatorial problem and to draw the algorithmic implications.

2. CONCEPTS OF COMPLEXITY THEORY

A formal theory of NP-completeness would require the introduction of *Turing machines* [Aho *et al.* 1974] as theoretical computing devices. A *deterministic* Turing machine is a classical model for an ordinary computer, which is polynomially related to more realistic models such as the *random access machine* [Aho *et al.* 1974]. It can be designed to *recognize languages*; the input consists of a *string*, which is *accepted* by the machine if and only if it belongs to the language. A *nondeterministic* Turing machine is an artificial model, which can be thought of as a deterministic one that can create copies of itself corresponding to different state transitions whenever convenient. In this case, a string is accepted if and only if it is accepted by one of the deterministic copies. *P* and *NP* are now defined as the classes of languages recognizable in polynomial time by deterministic and nondeterministic Turing machines, respectively.

For the purposes of exposition, we will expound the theory in terms of *recognition problems*, which require a yes/no answer. A string then corresponds to a problem *instance* and a language to a problem *type* or, more exactly, to the set of all its *feasible* instances. The feasibility of an instance is usually equivalent to the existence of an associated *structure*, whose size is bounded by a polynomial in the size of the instance; for example, the instance may be a graph and the structure a Hamiltonian circuit [Karp 1975A]. A recognition problem is in *P* if, for any instance, one can determine its feasibility or infeasibility in polynomial time. It is in *NP* if, for any instance, one can determine in polynomial time whether a given structure affirms its feasibility.

Problem P' is said to be *reducible* to problem P (notation: $P' \propto P$) if for any instance of P' an instance of P can be constructed in polynomial time such that solving the instance of P will solve the instance of P' as well. Informally, the reducibility of P' to P implies that P' can be considered as a special case of P, so that P is at least as hard as P'.

P is called *NP-hard* if $P' \propto P$ for every $P' \in NP$. In that case, P is at least as hard as any problem in *NP*. P is called *NP-complete* if P is NP-hard and $P \in NP$. Thus, the NP-complete problems are the most difficult problems in *NP*.

A good algorithm for an NP-complete problem P could be used to solve all problems in *NP* in polynomial time, since for any instance of such a problem the construction of the corresponding instance of P and its solu-

tion can be both effected in polynomial time. Note the following two important observations.

- It is very unlikely that $P = NP$, since NP contains many notorious combinatorial problems, for which in spite of a considerable research effort no good algorithms have been found so far.
- It is very unlikely that $P \in P$ for any NP-complete P, since this would imply that $P = NP$ by the earlier argument.

The first NP-completeness result is due to Cook [Cook 1971]. He designed a "master reduction" to prove that every problem in NP is reducible to the SATISFIABILITY problem. This is the problem of determining whether a boolean expression in conjuctive normal form assumes the value *true* for some assignment of truth values to the variables; for instance, the expression

$$(x_1) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_3) \qquad (1)$$

is satisfied if $x_1 = x_2 = x_3 =$ *true*. Given this result, one can establish NP-completeness of some $P \in NP$ by specifying a reduction $P' \propto P$ with P' already known to be NP-complete: for every $P'' \in NP$, $P'' \propto P'$ and $P' \propto P$ then imply that $P'' \propto P$ as well. In the following section we shall present several such proofs.

As far as *optimization problems* are concerned, we shall reformulate a minimization (maximization) problem by asking for the existence of a feasible solution with value at most (at least) equal to a given *threshold*. It should be noted that membership of NP for this recognition version does not immediately imply membership of NP for the original optimization problem as well. In particular, proposing a systematic search over a polynomial number of threshold values, guided by positive and negative answers to the existence question, is not a valid argument. This is because a nondeterministic Turing machine is only required to give *positive* answers in polynomial time. Indeed, no *complement* of any NP-complete problem is known to be in NP!

As an obvious consequence of the above discussion, NP-completeness can only be proved with respect to a recognition problem. However, the corresponding optimization problem might be called NP-hard in the sense that the existence of a good algorithm for its solution would imply that $P = NP$.

So far, we have been purposefully vague about the specific encoding of problem instances. Suffice it to say that most reasonable encodings are

polynomially related. One important exception with respect to the representation of positive integers will be dealt with in Section 3.5.

The classes *P* and *NP* are certainly not the only classes of interest to complexity theorists. There is, for instance, the class PSPACE, which contains all languages recognizable in polynomial space. This class is the same for both deterministic and nondeterministic Turing machines. There is a notion of PSPACE-completeness analogous to NP-completeness. The standard PSPACE-complete problem is "quantified" SATISFIABILITY or QSATISFIABILITY [Stockmeyer & Meyer 1973; Aho *et al.* 1974]. An instance of this problem results from the quantification of a boolean expression by both existential and universal quantifiers, *e.g.*

$$\forall\, x_1\ \exists\, x_2\ \forall\, x_3\ [(x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)].$$

The QSATISFIABILITY problem can be viewed as defining a game between two players: an "existential" player who tries to select values to make the expression *true* and a "universal" player who tries to defeat him. This insight has suggested a rich lode of simply-structured combinatorial games for which the problem of determining the outcome of optimal play is PSPACE-complete [Schaefer 1976]. One example of such a game is "generalized hex" [Even & Tarjan 1976].

Clearly $NP \subset$ PSPACE. It has not been proved that $NP \neq$ PSPACE. However, it seems reasonable to conjecture that this is the case and that PSPACE-complete problems are more difficult than NP-complete ones.

We should also mention that there are problems which have been shown to be inherently more difficult than any problem in PSPACE. For example, consider the "reachability" problem for vector addition systems: given a finite set of vectors with integer components, an initial vector u and a final vector v, is it possible to add vectors from the given set to u, with repetition allowed, so as to reach v, while always staying within the positive orthant? This problem has been shown to be decidable [Sacerdote & Tenney 1977] but to require exponential space [Lipton 1976]. Some other combinatorial problems have been shown to require exponential space as well [Stockmeyer & Meyer 1973].

3. NP-COMPLETENESS RESULTS

In this section we shall establish some basic-NP-completeness results according to the scheme given in Figure 1, and we shall mention similar results for related problems. Our proofs will be sketchy; for instance, it will be left to the reader to verify the membership of *NP* for the problems considered and the polynomial-boundedness of the reductions presented.

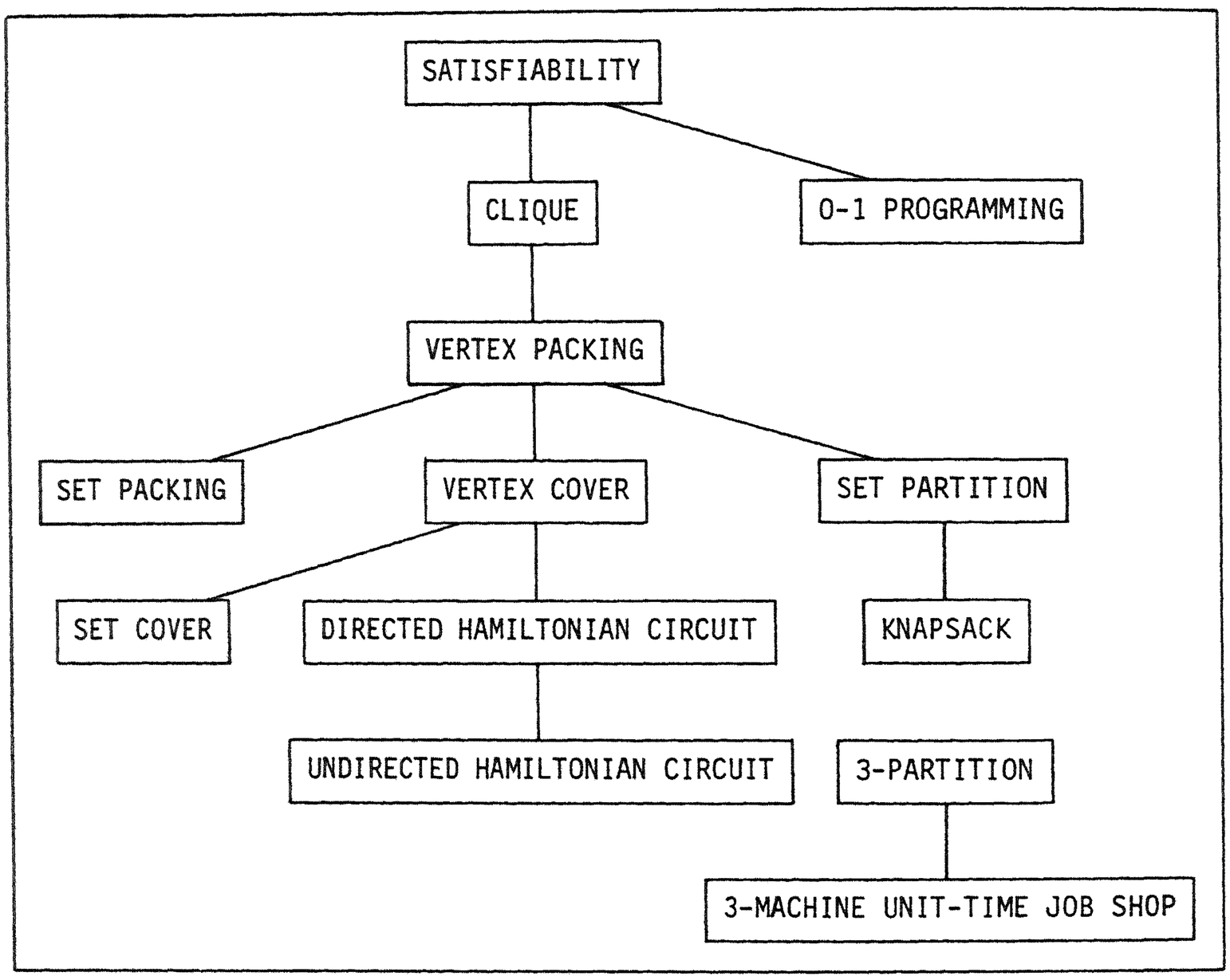


Figure 1 Scheme of reductions.

3.1. SATISFIABILITY

SATISFIABILITY: Given a *conjuctive normal form expression, i.e.* a conjunction of *clauses* $C_1,\ldots,C_s$, each of which is a disjunction of *literals* $x_1,\bar{x}_1,\ldots,x_t,\bar{x}_t$ where $x_1,\ldots,x_t$ are boolean variables and $\bar{x}_1,\ldots,\bar{x}_t$ denote their complements, is there a truth assignment to the variables such that the expression assumes the value *true*?

NP-completeness

It has already been mentioned that SATISFIABILITY was the first problem shown to be NP-complete. The proof of this key result is quite technical and beyond the scope of this paper; we refer to [Cook 1971; Aho *et al.* 1974]. We shall take (1) as an example of an instance of SATISFIABILITY to illustrate subsequent reductions.

Related results

Even the 3-SATISFIABILITY problem, *i.e.* SATISFIABILITY with at most three literals per clause, is NP-complete [Cook 1971]. The 2-SATISFIABILITY problem, however, belongs to P. Often, the borderline between easy and hard problems is crossed when a problem parameter increases from two to three. This phenomenon will be encountered on various occasions below, and is held by some to explain the division of mankind in two and not three sexes.

3.2. CLIQUE, VERTEX PACKING & VERTEX COVER

CLIQUE: Given an undirected graph G = (V,E) and an integer k, does G have a set of at least k pairwise adjacent vertices?

VERTEX PACKING (INDEPENDENT SET): Given an undirected graph G' = (V',E') and an integer k', does G' have a set of at least k' pairwise non-adjacent vertices?

VERTEX COVER: Given an undirected graph G = (V,E) and an integer k, does G have a set of at most k vertices such that every edge is incident with at least one of them?

NP-completeness

SATISFIABILITY $\propto$ CLIQUE:

$V = \{(x,i) \mid x \text{ is a literal in clause } C_i\}$;

$E = \{\{(x,i),(y,j)\} \mid x \neq \bar{y},\ i \neq j\}$;

$k = s$.

Cf. Figure 2. We have created a vertex for each occurrence of a literal in a clause and an edge for each pair of literals that can be assigned the value *true* independently of each other. A clique of size k corresponds to s literals (one in each clause) that satisfy the expression and *vice versa*

[Cook 1971]. The NP-completeness of CLIQUE now follows from (i) its membership of *NP*, (ii) the polynomial-boundedness of the reduction, and (iii) the NP-completeness of SATISFIABILITY.

CLIQUE $\propto$ VERTEX PACKING:

$V' = V;$

$E' = \{\{i,j\} \mid i \neq j, \{i,j\} \notin E\};$

$k' = k.$

Cf. Figure 3. A set of vertices is independent in G' if and only if it is a clique in the complementary graph G. This relation between the two problems belongs to folklore.

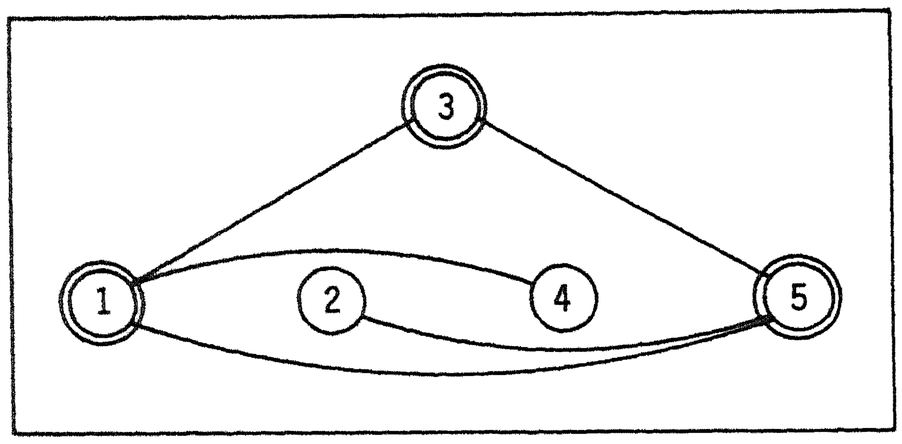


Figure 2 Instance of CLIQUE for the example.

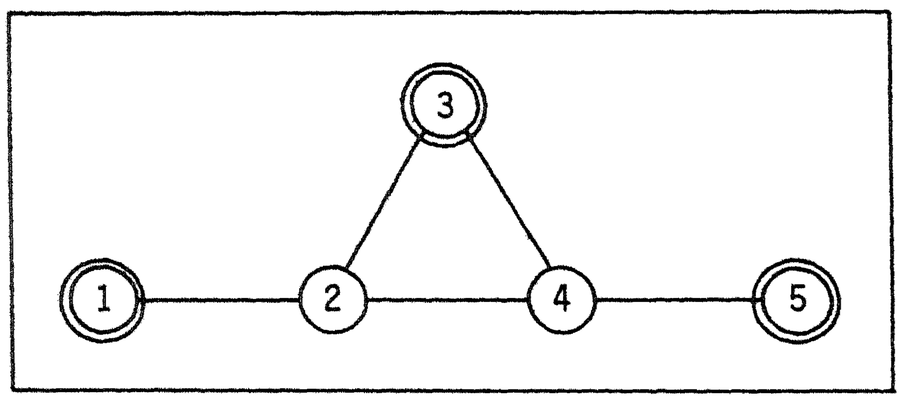


Figure 3 Instance of VERTEX PACKING for the example.

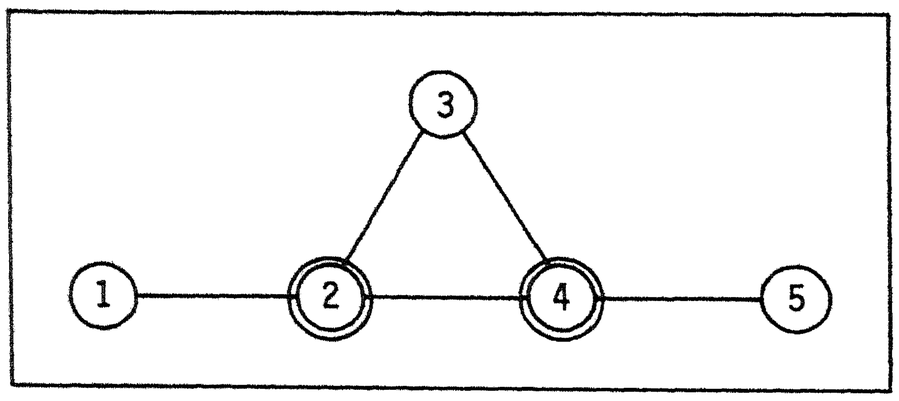


Figure 4 Instance of VERTEX COVER for the example.

VERTEX PACKING $\propto$ VERTEX COVER:

V = V';

E = E';

k = |V'|-k'.

Cf. Figure 4. It is well known that a set of vertices covers all edges if and only if its complement is independent.

Related results

Given the above results, it is not surprising (though less easy to prove) that the problems of determining whether the vertex set of a graph can be covered by at most k cliques or, after complementation, by at most k independent sets are NP-complete [Karp 1972]. These problems are known as CLIQUE COVER and GRAPH COLORABILITY respectively. In fact, it is already an NP-complete problem to determine if a planar graph with vertex degree at most 4 is 3-colorable [Garey *et al.* 1976D], whereas 2-colorability is equivalent to bipartiteness and can be checked in polynomial time.

3.3. SET PACKING, SET COVER & SET PARTITION

SET PACKING: Given a finite set S, a finite family $\mathcal{S}$ of subsets of S and an integer ℓ, does $\mathcal{S}$ include a subfamily $\mathcal{S}'$ of at least ℓ pairwise disjoint sets?

SET COVER: Given a finite set S, a finite family $\mathcal{S}$ of subsets of S and an integer ℓ, does $\mathcal{S}$ include a subfamily $\mathcal{S}'$ of at most ℓ sets such that $\bigcup_{S' \in \mathcal{S}'} S' = S$?

SET PARTITION (EXACT COVER): Given a finite set S and a finite family $\mathcal{S}$ of subsets of S, does $\mathcal{S}$ include a subfamily $\mathcal{S}'$ of pairwise disjoint sets such that $\bigcup_{S' \in \mathcal{S}'} S' = S$?

NP-completeness

VERTEX PACKING $\propto$ SET PACKING:

S = E';

$\mathcal{S}$ = {{{i,j}|{i,j} $\in$ E'}|i $\in$ V'};

ℓ = k'.

VERTEX COVER $\propto$ SET COVER:

delete the primes in the above reduction.

VERTEX PACKING and VERTEX COVER are easily recognized as special cases of SET PACKING and SET COVER respectively, and these reductions require no further comment.

VERTEX PACKING $\propto$ SET PARTITION:

$S = E' \cup \{1,\ldots,k'\}$;

$\mathcal{S} = \{S_{ih} \mid i \in V',\ h = 1,\ldots,k'\} \cup \{S_{\{i,j\}} \mid \{i,j\} \in E'\}$, where

$S_{ih} = \{\{i',j\} \mid \{i',j\} \in E,\ i' = i\} \cup \{h\}$,

$S_{\{i,j\}} = \{\{i,j\}\}$.

Cf. Figure 5. Suppose that G' has an independent set $U' \subset V'$ of size k', say, $U' = \{v_1,\ldots,v_{k'}\}$. Then the sets $S_{v_1 1},\ldots,S_{v_{k'},k'}$ are pairwise disjoint, and the elements of S not contained in any of them belong to E'. It follows that a partition of S is given by

$$\{S_{v_1 1},\ldots,S_{v_{k'},k'}\} \cup \{S_{\{i,j\}} \mid \{i,j\} \in E',\ i \notin U',\ j \notin U'\}.$$

Conversely, suppose that there exists a partition $\mathcal{S}'$ of S. Then $\mathcal{S}'$ contains k' pairwise disjoint sets $S_{v_1 1},\ldots,S_{v_{k'},k'}$, and the vertices $v_1,\ldots,v_{k'}$ clearly constitute an independent set of size k' in G'.

This reduction simplifies the NP-completeness proof given in [Karp 1972].

S	S_{11}	S_{21}	S_{31}	S_{41}	S_{51}	S_{12}	S_{22}	S_{32}	S_{42}	S_{52}	S_{13}	S_{23}	S_{33}	S_{43}	S_{53}	$S_{\{1,2\}}$	$S_{\{2,3\}}$	$S_{\{2,4\}}$	$S_{\{3,4\}}$	$S_{\{4,5\}}$
{1,2}	◉	•				•	•				•	•				•				
{2,3}		•	•				•	◉				•	•				•			
{2,4}		•		•			•		•			•		•				◉		
{3,4}			•	•				◉	•				•	•					•	
{4,5}				•	•				•	•				•	◉					•
1	◉	•	•	•	•															
2						•	•	◉	•	•										
3											•	•	•	•	◉					

Figure 5 Instance of SET PARTITION for the example.

Related results

Even the EXACT 3-COVER problem, where all subsets in $\mathcal{S}$ are constrained to be of size 3, is NP-complete, since it is an obvious generalization of the 3-DIMENSIONAL MATCHING problem, proved NP-complete in [Karp 1972]. An EXACT

2-COVER corresponds to a perfect matching in a graph, which can be found in polynomial time. The existence of good matching algorithms proves that EDGE PACKING and EDGE COVER problems are members of P.

3.4. DIRECTED & UNDIRECTED HAMILTONIAN CIRCUIT

DIRECTED HAMILTONIAN CIRCUIT: Given a directed graph H = (W,A), does H have a directed cycle passing through each vertex exactly once?

UNDIRECTED HAMILTONIAN CIRCUIT: Given an undirected graph G = (V,E), does G have a cycle passing through each vertex exactly once?

NP-completeness

VERTEX COVER $\propto$ DIRECTED HAMILTONIAN CIRCUIT:

$$W = \{(i,j),\{i,j\},(j,i) \mid \{i,j\} \in E\} \cup \{1,\ldots,k\};$$

$$A = \{((i,j),\{i,j\}),(\{i,j\},(i,j)),((j,i),\{i,j\}),(\{i,j\},(j,i)) \mid \{i,j\} \in E\}$$
$$\cup \{((h,i),(i,j)) \mid \{h,i\},\{i,j\} \in E,\ h \neq j\}$$
$$\cup \{((i,j),h),(h,(i,j)),((j,i),h),(h,(j,i)) \mid \{i,j\} \in E,\ h = 1,\ldots,k\}.$$

Cf. Figure 6. For each edge {i,j} in G we have created a configuration in H consisting of three vertices (i,j),{i,j},(j,i) and four arcs, as shown in the figure. The configurations are linked by arcs from (h,i) to (i,j) for $h \neq j$. Further, we have added k vertices 1,...,k and all arcs between them and the vertices of type (i,j).

Suppose that G has a vertex cover $U \subset V$ of size k, say, $U = \{v_1,\ldots,v_k\}$. The edge set E can then be written as

$$E = \{\{v_h,w_{h1}\},\ldots,\{v_h,w_{h\ell_h}\} \mid h = 1,\ldots,k\}$$

and it is easily checked that a hamiltonian circuit in H is given by

$$(1,(v_1,w_{11}),\{v_1,w_{11}\},(w_{11},v_1),\ldots,(v_1,w_{1\ell_1}),\{v_1,w_{1\ell_1}\},(w_{1\ell_1},v_1),$$
$$\ldots$$
$$k,(v_k,w_{k1}),\{v_k,w_{k1}\},(w_{k1},v_k),\ldots,(v_k,w_{k\ell_k}),\{v_k,w_{k\ell_k}\},(w_{k\ell_k},v_k),$$
$$1).$$

Conversely, suppose that H has a hamiltonian circuit. By deletion of all arcs incident with vertices 1,...,k, the circuit is decomposed into k paths. A path starting at (i,j) for $\{i,j\} \in E$ has to go on to visit {i,j} and (j,i); then it ends or goes on to visit (i,j'),{i,j'},(j',i) for some

{i,j'} $\in$ E, etc. Thus, this path corresponds to a specific vertex i $\in$ V, covering edges {i,j},{i,j'}, etc. Since the circuit passes through each {i,j} exactly once, each edge {i,j} $\in$ E is covered by one of k specific vertices, which therefore constitute a vertex cover of size k in G.

The above reduction is a modification of the original construction due to E.L. Lawler [Karp 1972], based on ideas of M. Fürer [Schuster 1976] and P. van Emde Boas.

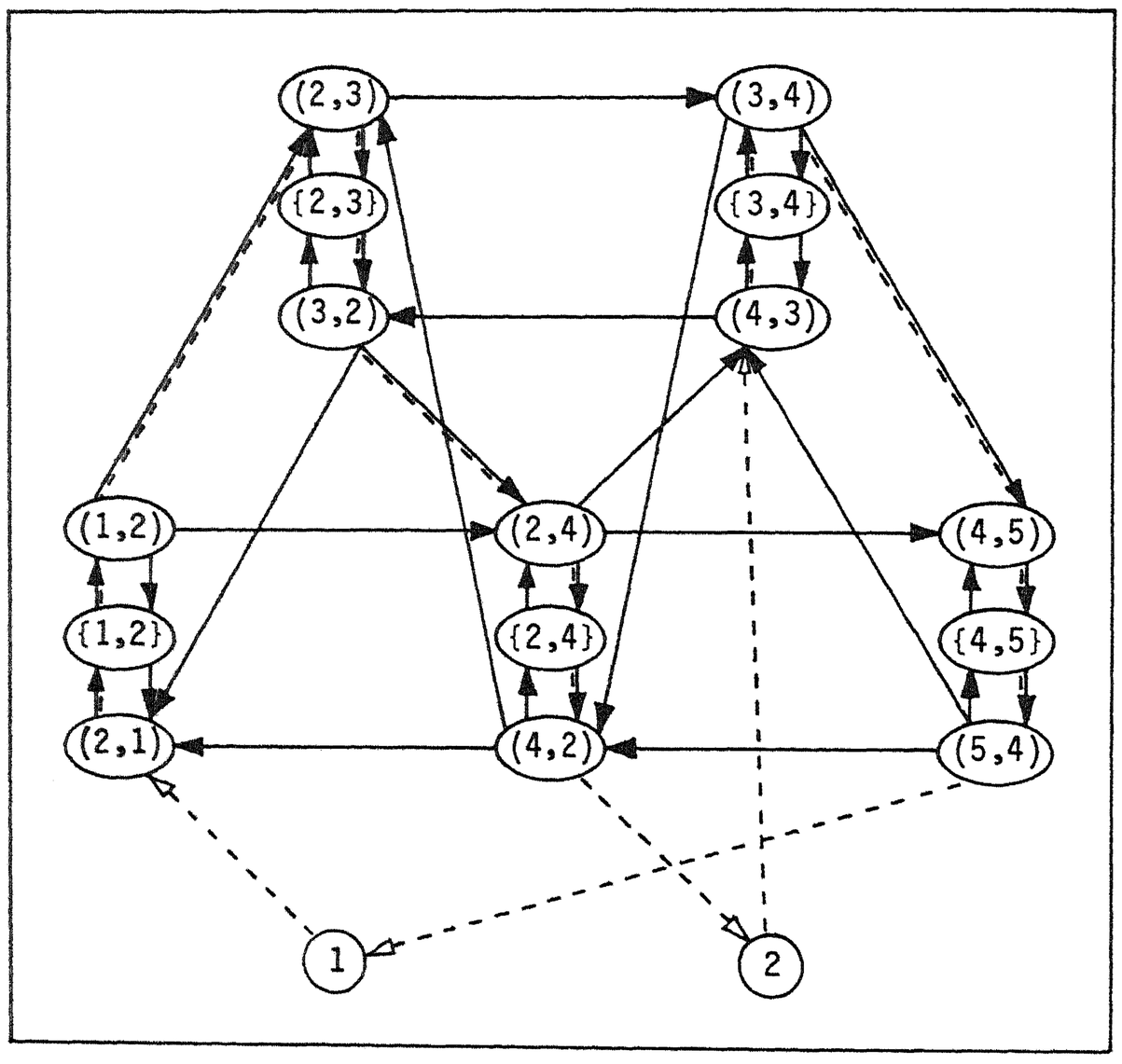


Figure 6 Instance of DIRECTED HAMILTONIAN CIRCUIT for the example. Not all arcs incident with vertices 1,...,k have been drawn.

DIRECTED HAMILTONIAN CIRCUIT $\propto$ UNDIRECTED HAMILTONIAN CIRCUIT:

$$V = \{(i,in),(i,mid),(i,out) \mid i \in W\};$$

$$E = \{\{(i,in),(i,mid)\},\{(i,mid),(i,out)\} \mid i \in W\}$$
$$\cup \{\{(i,out),(j,in)\} \mid (i,j) \in A\}.$$

The one-one correspondence between undirected hamiltonian circuits in G and directed hamiltonian circuits in H is evident. This reduction is due to R.E. Tarjan [Karp 1972].

Related results

The above results have been strengthened in various ways. For instance, the UNDIRECTED HAMILTONIAN CIRCUIT problem remains NP-complete if G is planar, triply-connected and regular of degree 3 [Garey *et al.* 1976E] or if G is bipartite [Krishnamoorthy 1975]. The latter result is a simple extension of the last reduction given above and we recommend it as an exercise.

NP-hardness of the (general) TRAVELING SALESMAN problem is another obvious consequence. Intricate NP-hardness proofs for the EUCLIDEAN TRAVELING SALESMAN problem can be found in [Garey *et al.* 1976A; Papadimitriou 1977]. It is well known that TRAVELING SALESMAN is a special case of the problem of finding a maximum weight independent set in the intersection of three matroids. Thus, the 3-MATROID INTERSECTION problem is NP-hard, whereas 2-MATROID INTERSECTION problems, such as finding an optimal linear assignment or spanning arborescence, can be solved in polynomial time [Lawler 1976B].

The TRAVELING SALESMAN problem serves as a prototype for a whole class of routing problems where, given a mixed graph consisting of a set V of vertices, a set E of (undirected) edges and a set A of (directed) arcs, a salesman has to find a minimum-weight tour passing through subsets $V' \subset V$, $E' \subset E$ and $A' \subset A$. If $V' = \emptyset$, $E' = E$ and $A' = A$, we have the CHINESE POSTMAN problem, which can be solved in polynomial time in the undirected or directed case ($A = \emptyset$ or $E = \emptyset$) [Edmonds 1965B; Edmonds & Johnson 1973], but is NP-hard in the mixed case [Papadimitriou 1976]. For the case that only $V' = \emptyset$, NP-hardness has been established for the UNDIRECTED and DIRECTED RURAL POSTMAN problems ($A = \emptyset$ and $E = \emptyset$ respectively) [Lenstra & Rinnooy Kan 1976] and for the STACKER-CRANE problem ($E' = \emptyset$, $A' = A$) [Frederickson *et al.* 1976].

3.5. 0-1 PROGRAMMING, KNAPSACK & 3-PARTITION

0-1 PROGRAMMING: Given an integer matrix A and an integer vector b, does there exist a 0-1 vector x such that $Ax \geq b$?

KNAPSACK: Given positive integers $a_1,\ldots,a_t,b$, does there exist a subset $T \subset \{1,\ldots,t\}$ such that $\sum_{j\in T} a_j = b$?

3-PARTITION: Given positive integers $a_1,\ldots,a_{3t},b$, do there exist t pairwise disjoint 3-element subsets $S_i \subset \{1,\ldots,3t\}$ such that $\sum_{j\in S_i} a_j = b$ $(i = 1,\ldots,t)$?

NP-completeness

SATISFIABILITY ∝ 0-1 PROGRAMMING:

$$a_{ij} = \left\{ \begin{array}{l} 1 \text{ if } x_j \text{ is a literal in clause } C_i, \\ -1 \text{ if } \bar{x}_j \text{ is a literal in clause } C_i, \\ 0 \text{ otherwise} \end{array} \right\} \quad (i = 1,\ldots,s,\ j = 1,\ldots,t);$$

$$b_i = 1 - |\{j | \bar{x}_j \text{ is a literal in clause } C_i\}| \quad (i = 1,\ldots,s).$$

Cf. Figure 7 and [Karp 1972].

$$\begin{aligned} x_1 &\geq 1 \\ -x_1 + x_2 - x_3 &\geq -1 \\ x_3 &\geq 1 \end{aligned}$$

Figure 7 Instance of 0-1 PROGRAMMING for the example.

SET PARTITION ∝ KNAPSACK:

Given $S = \{e_1,\ldots,e_s\}$ and $\mathcal{S} = \{S_1,\ldots,S_t\}$, we define

$$\varepsilon_{ij} = \left\{ \begin{array}{l} 1 \text{ if } e_i \in S_j \\ 0 \text{ if } e_i \notin S_j \end{array} \right\} \quad (i = 1,\ldots,s,\ j = 1,\ldots,t),$$

$$u = t+1,$$

and specify the reduction by

$$a_j = \sum_{i=1}^{s} \varepsilon_{ij} u^{i-1} \quad (j = 1,\ldots,t);$$

$$b = (u^s - 1)/t.$$

Cf. Figure 8. The one-one correspondence between solutions to KNAPSACK and SET PARTITION is easily verified [Karp 1972].

Given this result, the reader should have little difficulty in establishing NP-completeness for the PARTITION problem, *i.e.* KNAPSACK with

$$\sum_{j=1}^{t} a_j = 2b.$$

3-PARTITION has been proved NP-complete through a complicated sequence of reductions, which can be found in [Garey & Johnson 1975].

ε_{ij} ↓→	①	2	3	4	5	6	7	⑧	9	10	11	12	13	14	⑮	16	17	⑱	19	20	a_j	b
1	①	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	0	0	0	0	$\varepsilon_{1j}\cdot u^0+$	u^0+
2	0	1	1	0	0	0	1	①	0	0	0	1	1	0	0	0	1	0	0	0	$\varepsilon_{2j}\cdot u^1+$	u^1+
3	0	1	0	1	0	0	1	0	1	0	0	1	0	1	0	0	0	①	0	0	$\varepsilon_{3j}\cdot u^2+$	u^2+
4	0	0	1	1	0	0	0	①	1	0	0	0	1	1	0	0	0	0	1	0	$\varepsilon_{4j}\cdot u^3+$	u^3+
5	0	0	0	1	1	0	0	0	1	1	0	0	0	1	①	0	0	0	0	1	$\varepsilon_{5j}\cdot u^4+$	u^4+
6	①	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\varepsilon_{6j}\cdot u^5+$	u^5+
7	0	0	0	0	0	1	1	①	1	1	0	0	0	0	0	0	0	0	0	0	$\varepsilon_{7j}\cdot u^6+$	u^6+
8	0	0	0	0	0	0	0	0	0	0	1	1	1	1	①	0	0	0	0	0	$\varepsilon_{8j}\cdot u^7$	u^7

Figure 8 Instance of KNAPSACK for the example, where s = 8, t = 20, u = 21.

Binary vs. unary encoding

KNAPSACK was the first example of an NP-complete problem involving numerical data. The size of a problem instance is O(t log b) in the standard *binary* encoding and O(tb) if a *unary* encoding is allowed. Readers will have noticed that the reduction SET PARTITION $\propto$ KNAPSACK is polynomial-bounded only with respect to a binary encoding. Indeed, KNAPSACK can be solved by dynamic programming in O(tb) time [Bellmore & Dreyfus 1962], which might be called a *pseudopolynomial* algorithm in the sense that it is polynomial-bounded only with respect to a unary encoding. Thus, the *binary NP-completeness* of KNAPSACK and its *unary membership of P* are perfectly compatible results, although it tends to make us think of KNAPSACK as less hard than other NP-complete problems.

3-PARTITION was the first example of a problem involving numerical data that remains NP-complete even if we measure the problem size by using the actual numbers involved instead of their logarithms. This *strong* or *unary NP-completeness* of 3-PARTITION indicates that already the existence of a pseudopolynomial algorithm for its solution would imply that $P = NP$ [Garey & Johnson 1978B].

Quite often, a binary NP-completeness proof involving KNAPSACK or PARTITION can be converted to a unary NP-completeness proof involving 3-PARTITION in a straightforward manner. Occasionally, however, the polynomial-boundedness of a reduction depends essentially on allowing a unary encoding for 3-PARTITION. An example of such a reduction is given in the next section.

3.6. 3-MACHINE UNIT-TIME JOB SHOP

3-MACHINE UNIT-TIME JOB SHOP: Given 3 machines M_1, M_2, M_3 each of which can process at most one job at a time, n jobs $J_1, \ldots, J_n$ where J_j $(j = 1, \ldots, n)$ consists of a chain of unit-time operations, the h-th of which has to be processed on machine μ_{jh} with $\mu_{jh} \neq \mu_{j,h-1}$ for $h > 1$, and an integer k, does there exist a schedule with length at most k?

NP-completeness

3-PARTITION $\propto$ 3-MACHINE UNIT-TIME JOB SHOP:

$n = 3t+2;$

$\mu_j = (M_1, M_3, [M_1, M_2]^{a_j}, M_3) \quad (j = 1, \ldots, 3t);$

$\mu_{n-1} = ([M_2, M_3, M_2, M_1, M_2, M_1, [M_2, M_3]^b, M_1, M_2, M_1]^t);$

$\mu_n = ([M_3, M_2, M_3, M_2, M_1, M_2, [M_3, M_1]^b, M_2, M_1, M_2]^t);$

$k = (2b+9)t;$

where $[s]^h = s, [s]^{h-1}$ for $h > 1$ and $[s]^1 = s$.

Note that both J_{n-1} and J_n consist of a chain of operations of length equal to the threshold k. We may assume the h-th operations of these chains to be completed at time h, since otherwise the schedule length would exceed k. This leaves a pattern of idle machines for the other jobs that can be described as

$$([[M_1]^3, [M_3]^3, [M_1, M_2]^b, [M_3]^3]^t)$$

(*cf.* Figure 9). We will show that this pattern can be filled properly if and only if 3-PARTITION has a solution.

Suppose that 3-PARTITION has a solution $(S_1, \ldots, S_t)$. In this case, processing J_j with $j \in S_i$ entirely within the interval $[(2b+9)(i-1), (2b+9)i]$ $(j = 1, \ldots, 3t,\ i = 1, \ldots, t)$ yields a schedule with length k.

Conversely, suppose that there exists a schedule with length k. We will prove that in such a schedule exactly three jobs are started in $[0, 2b+9]$ and that they are completed in this interval as well; clearly, these jobs indicate a 3-element subset S_1 with $\sum_{j \in S_1} a_j = b$. One easily proves by induction that S_i is similarly defined by the jobs started and completed in $[(2b+9)(i-1), (2b+9)i]$ $(1 < i \leq t)$.

If J_j starts in $[0, 2b+9]$, its subchain of operations completed in that interval is of one of four types:

type 1: (M_1);
type 2: $(M_1, M_3, [M_1, M_2]^h)$ $(0 \leq h \leq a_j)$;
type 3: $(M_1, M_3, [M_1, M_2]^h, M_1)$ $(0 \leq h < a_j)$;
type 4: $(M_1, M_3, [M_1, M_2]^{a_j}, M_3)$.

Let x_i denote the number of subchains of type i and y_i the number of operations on M_2 in subchains of type i. We have to prove that $x_1 = x_2 = x_3 = 0$, $x_4 = 3$. Observing that a schedule of length k contains no idle unit-time periods, we have

(1) $x_1 + x_2 + y_2 + 2x_3 + y_3 + x_4 + y_4 = b+3$;
(2) $y_2 + y_3 + y_4 = b$;
(3) $x_2 + x_3 + 2x_4 = 6$.

Subtracting (1) from the sum of (2) and (3), we obtain $-x_1-x_3+x_4 = 3$ and therefore $x_4 \geq 3$. Also, (3) implies that $x_4 \leq 3$. It follows that $x_4 = 3$, and $x_1 = x_2 = x_3 = 0$.

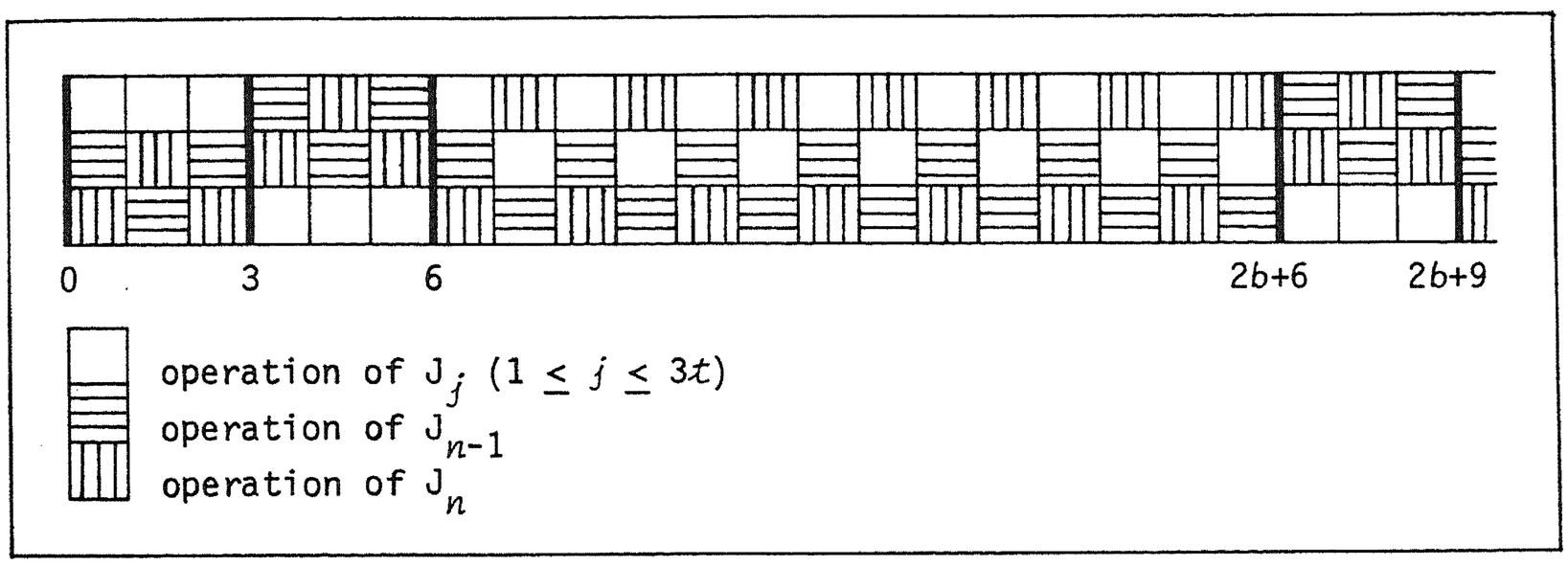


Figure 9 First part of 3-MACHINE UNIT-TIME JOB SHOP schedule corresponding to an instance of 3-PARTITION with b = 7.

Related results

The complexity of the 2-MACHINE UNIT-TIME JOB SHOP problem is unknown; to introduce a competitive element we shall be happy to award a chocolate windmill to the first person establishing membership of P or NP-completeness for this problem. If the processing times of the operations are allowed to be equal to 1 or 2, the 2-machine problem can be proved NP-complete by a reduction similar to (but simpler than) the above one; this improves upon related results given in [Garey *et al.* 1976C; Lenstra *et al.* 1977]. If each job has at most two operations, the 2-machine problem belongs to

P even for arbitrary processing times [Jackson 1956].

These results form but a small fraction of the extensive complexity analysis carried out for scheduling problems. We refer to [Ullman 1975; Garey & Johnson 1975; Coffman 1976; Garey *et al.* 1976C; Lenstra *et al.* 1977; Lenstra & Rinnooy Kan 1978A] for further details and to [Graham *et al.* 1978] for a concise survey of the field.

4. CONCLUDING REMARKS

We hope that the preceding section has conveyed some of the flavor and elegance of NP-completeness results. In only a few years an impressive amount of results has been obtained. Nevertheless, there are still plenty of *open problems,* for which neither a polynomial algorithm nor an NP-completeness proof is available. We shall mention four famous ones, on whose complexity status little or no progress has been made so far.

(a) GRAPH ISOMORPHISM

This is the problem of determining whether there exists a one-one mapping between the vertex sets of two graphs which preserves the adjacency relation. The essential nature of the problem does not change if we restrict our attention to graphs of certain types such as bipartite or regular ones; these problems are polynomially equivalent to the general case [Booth 1976]. The status of the problem is totally unknown and we do not dare to guess the final outcome.

(b) MATROID PARITY

This problem is interesting because it generalizes both the matroid intersection problem and the nonbipartite matching problem [Lawler 1976B]. Despite serious investigation, its status is far from clear. A special case of the matroid parity problem is as follows. Given a connected graph G with an even number of edges, arbitrarily paired (*i.e.*, each edge e has a uniquely defined *mate* $\bar{e}$), does G have a spanning tree T with the property that if an edge is contained in T, then its mate is in T as well? An NP-completeness proof for this special case would, of course, resolve the question for the general problem. On the other hand, a polynomial-time algorithm for this special case would probably suggest a similar procedure for the general problem.

(c) 3-MACHINE UNIT-TIME PARALLEL SHOP

This problem involves the scheduling of unit-time jobs on three identical parallel machines subject to precedence constraints between the jobs, so as to meet a common deadline of the jobs. For a variable number of machines,, the problem is NP-complete [Ullman 1975; Lenstra & Rinnooy Kan 1978A]; the special case of tree-type precedence constraints can be solved in polynomial time [Hu 1961]. The 2-machine problem belongs to P [Coffman & Graham

1972], even if for each job a time-interval is specified in which it has to be processed [Garey & Johnson 1977]. The 3-machine problem has remained open in spite of vigorous attacks. In this case we would be willing to extrapolate on the magic quality of three-ness and conjecture NP-completeness.

(d) LINEAR PROGRAMMING

This is perhaps the most vexing open problem. The simplex method performs very well in practice and usually requires time linear in the number of constraints. On certain weird polytopes, however, it takes exponential time [Klee & Minty 1972]. Fortunately, in this case there is circumstantial evidence against NP-completeness. Thanks to duality theory, determining the existence or nonexistence of a feasible solution are equally hard problems, and NP-completeness of LINEAR PROGRAMMING would therefore imply NP-completeness for the complements of all other NP-complete problems as well. However, as mentioned, it is not even known whether the complement of any NP-complete problem belongs to *NP*. In addition to the above rather technical argument, it seems highly unlikely that all NP-complete problems would allow a polynomial-bounded linear programming formulation.

Interpretation of NP-completeness results as more or less definite proofs of computational intractability has stimulated the design and analysis of fast *approximation algorithms*.

With respect to the *worst-case analysis* of such algorithms, a wide variety of outcomes is possible. We give the following examples.

(1) For the optimization version of the KNAPSACK problem, a solution within an arbitrary percentage ε from the optimum can be found in time polynomial in t and $1/\varepsilon$ [Ibarra & Kim 1975A; Lawler 1978B].

(2) For the EUCILDEAN TRAVELING SALESMAN problem, a solution within 50% from the optimum can be found in polynomial time [Christofides 1978; Cornuejols & Nemhauser 1978].

(3) For the GRAPH COLORABILITY problem, a solution within 100% from the optimum cannot be found in polynomial time unless $P = NP$ [Garey & Johnson 1976A].

(4) For the general TRAVELING SALESMAN problem, a solution within any fixed percentage from the optimum cannot be found in polynomial time unless $P = NP$ [Sahni & Gonzalez 1976].

We refer to [Garey & Johnson 1976C] for a survey of this area. Impressive

advances have been made and more can be expected in the near future.

The above approach to *performance guarantees* may be accused of being overly pessimistic - *cf.* the simplex method with its exponential worst-case behavior! The *probabilistic analysis* of average or "almost everywhere" behavior, however, requires the specification of a probability distribution over the set of all problem instances. For some problems, a natural distribution function is available and some intriguing results have been derived [Karp 1976], although technically this approach seems to be very demanding.

The worst-case analysis of approximation algorithms shows that there are significant differences in complexity within the class of NP-complete problems. These problems might be classifiable according to the best possible polynomial-time performance guarantee that one can get (*cf.* [Ausiello & D'Atri 1977; Paz & Moran 1977]). Another refinement of the complexity measure may be based on the way in which numerical problem data are encoded, *i.e.* on the distinction between binary and unary encoding mentioned in Section 3.5. Several other ways of measuring problem size could be devised and each of them could be subjected to a complexity analysis, producing new information on the best type of algorithm that is likely to exist.

The concluding remarks above were intended to confirm to the reader that the field of computational complexity is still very much alive. In the first place, however, the theory of NP-completeness has yielded highly useful tools for the analysis of combinatorial problems that deserve to find acceptance in a wide circle of researchers and practitioners.

ACKNOWLEDGMENTS

We have received valuable comments from several participants to "Discrete Optimization 1977" in Vancouver and are particularly grateful for various constructive suggestions by E.L. Lawler.

WORST-CASE ANALYSIS OF HEURISTICS

M.L. FISHER

The Wharton School, University of Pennsylvania, Philadelphia, U.S.A.

ABSTRACT

The basic ground rules of worst-case analysis of heuristics are reviewed and a large variety of the existing types of worst-case results are described in terms of the knapsack problem. A selected sample of results for other problems is also given.

CONTENTS

1. INTRODUCTION 89

2. FUNDAMENTALS OF WORST-CASE ANALYSIS 92

3. SELECTIVE SURVEY OF RESULTS FOR OTHER PROBLEMS 99

3.1. Parallel machine scheduling 99

3.2. Uncapacitated location 100

3.3. The traveling-salesman problem 103

ACKNOWLEDGMENTS 104

1. INTRODUCTION

One of the significant developments in the field of combinatorial optimization during this decade was the identification of a large class of important problems that are probably computationally intractable. To be more precise, it is conjectured that any algorithm for these problems must, in the worst case, require an amount of computation time that grows exponentially in the size of the problem input. Two pieces of evidence support this somber prediction. First, all known algorithms for these problems have exponential worst-case running time. Second, Cook [Cook 1971] and Karp [Karp 1972, 1975A] have used the concept of NP-completeness to show that these problems are equivalent in the sense that if any of them can be solved in less than exponential time, they all can. The list of NP-complete problems is long and includes many important examples in scheduling, location, routing and graphical optimization.

An individual who must face one of these problems as part of his daily life is frequently bewildered when told that his problem is extremely difficult. While he may regard problem definition, collection of relevant data or implementation of results as challenging tasks, the computation of a solution is rarely difficult for him because he is willing to resort to simple heuristic methods. These methods require little computation and usually produce solutions that are acceptably close to optimal. Computational economy is one obvious reason for employing heuristics. Their use is sometimes also dictated by a problem setting that requires that a solution be computed dynamically before all problem data are known. For example, this is often the case with scheduling problems.

Until recently, formal research on heuristic methods contained large doses of art and empiricism and there was a tendency to regard heuristics as somehow less elegant than optimization methods. Of course, there is no reason why a rigorous treatment of approximation methods need be inelegant. Recently a number of analytic results have appeared that might be regarded as first steps toward a theory of heuristics. These results provide various kinds of information about a heuristic to assist a problem solver who must select among alternative heuristics or a designer of heuristics who wishes to evaluate his latest creation. Two distinct approaches have been taken to heuristic analysis: worst-case and probabilistic. Worst-case results establish the maximum deviation from optimality that can occur when a specified heuristic is applied within a given problem class. In a probabilistic

approach, one assumes a density function for the problem data and tries to establish probabilistic properties of a heuristic, such as a bound on the probability that the heuristic finds a solution within a prespecified percentage of optimality. A general description of the probabilistic approach is available in [Karp 1976] and a specific application to the traveling-salesman problem is discussed in this volume [Karp 1977].

It should be clear that worst-case and probabilistic analyses offer different advantages and limitations in achieving the basic objective of predicting the performance of a heuristic. Therefore, these methods, together with computational testing, should be viewed as complementary rather than competitive. In assessing the limitations of either approach it is also important to realize just how new this field is. A recent bibliography [Garey & Johnson 1976C] provides a thorough compilation of references through mid-1976. Of the 48 papers listed in this bibliography, only 3 have publication dates prior to 1972. An optimist would expect that there is substantial opportunity for improving our ability to predict heuristic performance.

These methods can also be employed for broader purposes than the study of a specific heuristic.

The effect of approximations in models can be studied using either worst-case or probabilistic analysis in a manner that parallels heuristic analysis both philosophically and technically. For example, the effect of various levels of market aggregation in a location model might be studied. Studies of this type have been conducted by Geoffrion [Geoffrion 1977A, 1977B] using a worst-case approach.

Another natural extension is to use these methods to study the performance of relaxations designed for obtaining lower bounds in a branch-and-bound algorithm. For example, a worst-case result for a linear programming relaxation of the uncapacitated location problem has been reported in [Cornuejols *et al.* 1977].

These methods can also be of value in understanding the performance of optimizing algorithms, such as branch-and-bound, that employ heuristics and relaxations. Also, while it is important to have the option of pursuing the optimum, one must also be prepared for the possibility that this pursuit will be uneconomical for some large problems. If the algorithm is terminated early, worst-case or probabilistic analysis can provide useful predictions about the quality of the suboptimal solutions obtained.

Finally, research on heuristics might further refine the problem taxon-

omy offered by complexity theory. For example, NP-complete problems might be classified by how amenable they are to heuristic solution. Some preliminary results along these lines are given in [Garey & Johnson 1978B].

This paper provides an introduction to the worst-case analysis of heuristics. In Section 2 the basic ground rules of worst-case analysis are reviewed and a large variety of the existing types of worst-case results are described in terms of the knapsack problem. Section 3 gives a selected sample of results for other problems. Although the list of references for this paper is long, it is not intended to be a comprehensive survey. For a comprehensive listing of references through mid-1976, consult the bibliography in [Garey & Johnson 1976C].

2. FUNDAMENTALS OF WORST-CASE ANALYSIS

In this section the knapsack problem is used to introduce the fundamental concepts of worst-case analysis of heuristics and to illustrate the type of results that have been obtained. The standard knapsack problem is concerned with optimally filling a knapsack of capacity b using n items. Item j has value p_j and weight a_j. The problem is formulated as follows.

$$Z = \max \sum_{j=1}^{n} p_j x_j \tag{1}$$

$$\text{s.t.} \sum_{j=1}^{n} a_j x_j \leq b, \tag{2}$$

$$x_j \geq 0 \text{ and integral, } j = 1,\ldots,n. \tag{3}$$

We assume that n, b, p_j and a_j are positive integers satisfying $a_j \leq b$ for all j. The 0-1 knapsack problem is a variation in which (3) is replaced by the requirement that $x_j = 0$ or 1.

Despite the simplicity of this model, a number of heuristic solution methods have been proposed and studied analytically. These knapsack heuristics and worst-case analyses are representative of much of the general research on worst-case performance of heuristics for combinatorial problems.

The ground rules of a worst-case analysis are simple. The analysis is always conducted with respect to a well-defined heuristic and set of problem instances. Let P denote the set of problem instances, $I \in P$ a particular problem instance, Z(I) the optimal value of problem I, and $Z_H(I)$ the value obtained when heuristic H is applied to problem I. We assume that heuristic H is suitably specified to uniquely define $Z_H(I)$. In terms of the standard knapsack problem, I is defined by a vector of integers consisting of a positive integer value n denoting the number of items and a (2n+1)-tuple of positive integers $p_1,\ldots,p_n,a_1,\ldots,a_n,b$ that satisfy $a_j \leq b$ for all j. P is the set of all such n and (2n+1)-tuples.

We assume throughout this section that we are dealing with maximization problems for which $Z(I) \geq 0$. Given a real number $r \leq 1$, heuristic H is said to solve problem class P within r if

$$Z_H(I) \geq rZ(I) \quad \text{for all } I \in P. \tag{4}$$

In analyzing a heuristic we would like to find the largest r for which (4) holds. This value is called the *worst-case performance ratio*. We know that a given r is the largest possible if we can find a single problem instance

I for which $Z_H(I) = rZ(I)$ or an infinite sequence of instances for which $Z_H(I)-rZ(I)$ approaches zero. For notational simplicity we will drop the argument I on $Z_H(I)$ and $Z(I)$ throughout the remainder of the paper with the understanding that Z_H and Z are always defined for a particular problem instance.

The first heuristic we will consider for the knapsack problem is called a *greedy heuristic*. It is convenient to assume that items are indexed so that $p_1/a_1 \geq p_2/a_2 \geq \ldots \geq p_n/a_n$. A greedy heuristic can be given for either the standard or 0-1 knapsack problems. For the standard problem items are considered in index order (decreasing p_i/a_i). We place as many integral units of each new item into the knapsack as will fit in the remaining capacity. In the case of the 0-1 problem, a single unit of each new item is placed into the knapsack if it fits.

A formal description of the greedy heuristic that applies to both versions of the knapsack problem can be given if we introduce an upper bound u_j on x_j. For the 0-1 knapsack problem $u_j = 1$, while for the standard problem $u_j = [b/a_j]$ where $[y]$ denotes the largest integer less than or equal to y.

Greedy heuristic

1. Set $j = 1$ and $\bar{b} = b$.
2. Set $x_j = \min\{u_j, [\bar{b}/a_j]\}$ and $\bar{b} = \bar{b}-a_jx_j$.
3. Stop if $j = n$. Otherwise set $j = j+1$ and go to 2.

If the items have been presorted so that $p_1/a_1 \geq p_2/a_2 \geq \ldots \geq p_n/a_n$, greedy runs in O(n) time. Sorting the items requires O(n log n) time.

It is easy to see that for the standard knapsack problem greedy has a worst-case performance ratio of $r = \frac{1}{2}$. Let Z_G denote the value of a greedy knapsack packing and Z the value of an optimal packing. Clearly $Z_G \geq p_1[b/a_1]$. Solution of (1)-(3) with the integrality requirement on x_j relaxed provides the upper bound $p_1(b/a_1) \geq Z$. Together these results imply

$$\frac{Z_G}{Z} \geq \frac{[b/a_1]}{b/a_1} = \frac{[b/a_1]}{[b/a_1]+(b/a_1-[b/a_1])} \geq \frac{1}{2}.$$

The second inequality follows from $[b/a_1] \geq 1 \geq (b/a_1-[b/a_1])$. To show that the bound of $\frac{1}{2}$ is tight, consider the series of problems with $n = 2$, $p_1 = a_1 = k+1$, $p_2 = a_2 = k$, and $b = 2k$ for $k = 1,2,\ldots$. For this series $Z = 2k$ and $Z_G = k+1$, so Z_G/Z can be arbitrarily close to $\frac{1}{2}$.

Obviously, for any heuristic and problem class the worst-case performance ratio must be at least 0. The greedy heuristic applied to the 0-1 knapsack problem is a case which shows that the worst-case performance can be as bad as 0. The series of examples $n = 2$, $p_1 = a_1 = 1$, $p_2 = a_2 = k$, $b = k$, $k = 1,2,\ldots$ has $Z_G = 1$ and $Z = k$ so Z_G/Z can be arbitrarily close to zero. Such a heuristic is said to be *arbitrarily bad* for the given problem class.

Since greedy is arbitrarily bad for the 0-1 knapsack problem, one might ask whether there is any polynomial-time heuristic for this problem with $r > 0$. The answer is yes; in fact there are heuristics with r arbitrarily close to 1. The first method of this type to be suggested involves partial enumeration of subsets of items. The heuristic is parameterized by an integer k, has a worst-case performance ratio of $r = \frac{k}{k+1}$ and requires $O(kn^{k+1})$ computation. For $S \subseteq \{1,\ldots,n\}$ let $Z_G(S)$ denote the value of a greedy packing of items from $\{1,\ldots,n\}$-S into a knapsack of capacity $b - \sum_{j\in S} a_j$. The k-th level partial enumeration algorithm obtains a solution by solving

$$Z_{E(k)} = \max_S \{\sum_{j\in S} p_j + Z_G(S)\}$$

$$\text{s.t. } \sum_{j\in S} a_j \leq b,$$

$$|S| \leq k.$$

This problem is solved by enumeration of all sets S of cardinality k or less. There are $\sum_{i=1}^{k} \binom{n}{i} < kn^k$ such sets and computation of $Z_G(S)$ for each set requires at most $O(n)$. Thus the time to compute $Z_{E(k)}$ is bounded by $O(kn^{k+1})$.

Partial enumeration was first proposed and studied by Johnson [Johnson 1974B] for the case $p_j = a_j$ for all j. Sahni [Sahni 1975] extended this work to the general case. It is shown in [Sahni 1975] that $Z_{E(k)} \geq \frac{k}{k+1} Z$. The series of problem instances $n = k+2$, $p_1 = 2$, $a_1 = 1$, $p_i = a_i = j$, $b = (k+1)j$ for $j = 3,4,\ldots$ has $Z = (k+1)j$ and $Z_{E(k)} = kj+2$ so this bound is tight for all k.

A heuristic of this type is called a *polynomial approximation scheme*. Any performance ratio can be obtained and for a fixed performance ratio, the computational requirements are polynomial in problem size. The quantity 1-r might be called the worst-case percentage error of a heuristic. As a function of 1-r the computation required by the partial enumeration scheme is $O(\frac{r}{1-r} n^{1/(1-r)})$. Thus, although the scheme can achieve r arbitrarily close

to 1, the computation grows exponentially in the inverse of 1-r.

This disadvantage is overcome in the following scheme for the 0-1 knapsack problem in which r can be arbitrarily close to 1 and computational requirements grow polynomially in the inverse of 1-r. A method with this property is called a *fully polynomial approximation scheme*.

There is a well-known dynamic programming algorithm in which the 0-1 knapsack problem is solved by recursively solving a family of knapsack problems that use some of the items and a portion of the knapsack. A variation on this approach will be given that forms the basis of the fully polynomial approximation scheme. Given integers y and k with $1 \le k \le n$, let

$$f_k(y) = \min \sum_{j=1}^{k} a_j x_j$$

$$\text{s.t.} \sum_{j=1}^{k} p_j x_j \ge y,$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,k.$$

By convention set $f_k(y) = \infty$ if $y > \sum_{j=1}^{k} p_j$. We also know by definition that $f_k(y) = 0$ if $y \le 0$. Then $f_k(y)$ can be computed for $k = 1,\ldots,n$ and $y = 1,2,\ldots$ by

$$f_1(y) = \begin{cases} a_1, & 1 \le y \le p_1, \\ \infty, & y > p_1, \end{cases}$$

$$f_k(y) = \min\{f_{k-1}(y), f_{k-1}(y-p_k)+a_k\}, \quad k = 2,\ldots,n.$$

The 0-1 knapsack problem can be solved by computing $f_n(y)$ for $y = 1,2,\ldots$. Let K+1 be the least integer for which $f_n(K+1) > b$. Then the optimal value is given by $Z = K$. The optimal solution can also be obtained if a 0-1 index $x_k(y)$ is maintained as $f_k(y)$ is computed. The index is given by

$$x_1(y) = 1, \quad 1 \le y \le p_1$$

and

$$x_k(y) = \begin{cases} 1, & f_{k-1}(y) > f_{k-1}(y-p_k)+a_k, \\ 0, & \text{otherwise} \end{cases}$$

for $k = 2,\ldots,n$ and $y \ge 1$. Then it is easy to verify that the solution $\bar{x}_1,\ldots,\bar{x}_n$ computed recursively by $\bar{x}_n = x_n(K)$ and $\bar{x}_k = x_k(K - \sum_{j=k+1}^{n} p_j \bar{x}_j)$, $k = n-1,n-2,\ldots,1$ satisfies $\sum_{j=1}^{n} p_j \bar{x}_j = K$ and $\sum_{j=1}^{n} a_j \bar{x}_j = f_n(K) \le b$.

Let $d = \sum_{j=1}^{n} p_j$. Since $f_n(d+1) > b$ this dynamic programming scheme for solving the 0-1 knapsack problem requires at most O(nd) evaluations of $f_k(y)$. Since the computational requirements are proportional to $\sum_{j=1}^{n} p_j$, one might consider the possibility of economizing on computation by using a coarser scale for measuring profits. Specifically, for some $\lambda > 1$ we might replace each p_j by $\langle p_j/\lambda \rangle$ where $\langle a \rangle$ denotes the smallest integer not less than a. With this substitution, computational requirements become $O(\frac{nd}{\lambda})$. For example, if p_j is measured in dollars, then using $\lambda = 100$ is equivalent to measuring profit in hundreds of dollars. If it is really impossible to measure item profits more accurately than to the nearest hundred dollars, introducing such a scale change makes perfect sense; we save two orders of magnitude in computation with no real loss in accuracy. Moreover, even if item profits can be determined precisely to the nearest dollar, using scaled profits may be a desirable heuristic if the loss in optimality due to scaling is outweighed by the savings in computation.

To assess the loss in optimality due to scaling, let $\bar{x}$ denote an optimal knapsack solution for profits $\langle p_j/\lambda \rangle$, define $Z_{DP(\lambda)} = \sum_{j=1}^{n} p_j \bar{x}_j$, and consider the ratio $Z_{DP(\lambda)}/Z$. Clearly $Z \geq \frac{d}{n}$ since $\max_j\{p_j\} \geq \frac{d}{n}$ and $a_j \leq b$ for all j. Also, since $\lambda\langle p_j/\lambda \rangle \geq p_j$ and $\bar{x}$ is optimal for profits $\lambda\langle p_j/\lambda \rangle$, $Z \leq \sum_{j=1}^{n} \lambda\langle p_j/\lambda \rangle \bar{x}_j \leq \sum_{j=1}^{n} (p_j+\lambda)\bar{x}_j \leq Z_{DP(\lambda)} + n\lambda$. Together these results give

$$Z_{DP(\lambda)} \geq (1 - \frac{n\lambda}{Z})Z \geq (1 - \frac{n^2\lambda}{d})Z.$$

Thus to guarantee a specific performance ratio r we would set $\lambda = (1-r)d/n^2$ which implies computational requirements $O(\frac{nd}{\lambda}) = O(n^3 \frac{1}{1-r})$. Since these requirements are polynomial in n and the inverse of 1-r, we have obtained a fully polynomial approximation scheme.

The version of the fully polynomial approximation scheme presented here is simple and easy to explain. For operational purposes, there are more complicated schemes based on similar principles that have running time less than $O(n^3 \frac{1}{1-r})$. The first of these was given in [Ibarra & Kim 1975A]. Their scheme is improved upon in [Lawler 1978B], appearing in this volume.

Fully polynomial approximation schemes exist for some other problem types. On the other hand, Garey and Johnson have shown that if a problem has a certain property defined in [Garey & Johnson 1978B], then no fully polynomial approximation scheme is possible. A number of important problems have this property.

Dynamic programming with data scaling is clearly more attractive than

partial enumeration for the 0-1 knapsack problem, at least for $r \geq \frac{2}{3}$. Partial enumeration is still of interest, however, because it is applicable to other problems (specifically the uncapacitated location problem as described in [Nemhauser *et al.* 1978; Fisher *et al.* 1978B]) for which a *fully* polynomial approximation scheme is precluded by the Garey-Johnson result.

Simple though they are, the foregoing analyses of knapsack problem heuristics illustrate a number of features that are typical of many worst-case analyses. Specifically:

1. For a given heuristic H, a bound on Z_H/Z is actually established by bounding the ratio of Z_H to an upper bound on Z. This is not surprising since the whole reason for resorting to heuristics is that it is hard to get information about Z.
2. There is no single problem instance for which Z_H/H achieves its minimum value. Rather, an infinite series of instances are required.
3. Several types of worst-case performance are possible. At one extreme, a perfectly plausible heuristic has arbitrarily bad performance ($r = 0$). Another heuristic has a fixed performance ratio r between 0 and 1. Other heuristics are parameterized and can have any performance ratio r satisfying $0 < r < 1$.
4. For fixed r, all heuristics have a running time that is polynomial in problem input. Most heuristic either run in polynomial time or are iterative improvement methods (see Section 3.2) that can be made to run in polynomial time by early termination. For heuristics where r may be varied, the running time can vary either exponentially in r (polynomial approximation scheme) or polynomially in r (fully polynomial approximation scheme).
5. The examples that achieve the worst-case performance are quite specialized and one might suspect that the worst-case performance is not predictive of performance for a "random" problem. It is possible to compute more predictive performance bounds that are functions of one or more summary parameters of the problem inputs. For example, as a function of $m = [b/\max_j\{a_j\}]$, the bound for greedy for the standard knapsack problem can be sharpened to

$$\frac{Z_G}{Z} \geq \frac{[b/a_1]}{b/a_1} \geq \frac{[b/a_1]}{[b/a_1]+1} \geq \frac{m}{m+1}.$$

This bound could provide useful information to someone contemplating the use of the greedy heuristic for the repetitive solution of knapsack

problems for which it is known that the weight of all items are small relative to knapsack size. Despite their apparent usefulness, past research has not emphasized input dependent bounds.

6. After executing the greedy heuristic one can compute what might be called an "a posteriori" bound. For the standard knapsack problem and the greedy heuristic this bound is equal to $Z_G/(b/a_1)$ and is the tightest of the possible bounds we have discussed. It would be useful in deciding whether additional computational effort is warranted before using more elaborate heuristics or optimizing procedures.

3. SELECTIVE SURVEY OF RESULTS FOR OTHER PROBLEMS

This section will give a brief description of some results for three other problems: (1) parallel machine scheduling, (2) uncapacitated location, and (3) the traveling-salesman problem. These particular results have been selected to illustrate the diversity of worst-case research on heuristics.

3.1. Parallel machine scheduling

In pioneering papers published in 1966 and 1969, Graham describes what is probably the first worst-case analysis of a heuristic. Graham was concerned with the following parallel machine scheduling problem. Given m identical machines and n jobs, where each job j is to be processed on one of the machines for an uninterrupted interval of length p_j, determine an assignment of jobs to machines that minimizes the earliest time at which all jobs are complete.

For simplicity, assume that jobs are indexed so that $p_1 \geq p_2 \geq \ldots \geq p_n$. The longest processing time (LPT) rule is a heuristic for this problem that assigns jobs sequentially in increasing index order. Each job is assigned to the machine with the least processing already assigned. Assume ties in the choice of a machine are resolved in favor of the machine with smallest index. Figure 1 depicts the LPT and optimal assignments for an example with m = 3, n = 7, and processing times 5, 5, 4, 4, 3, 3 and 3.

In this example the completion time of the LPT assignment is $\frac{11}{9}$ times the minimum completion time. In [Graham 1969] it is shown that this is the worst that the LPT rule can do when m = 3. More generally, if Z_{LPT} is the completion time of the LPT assignment and Z the minimum completion time, then

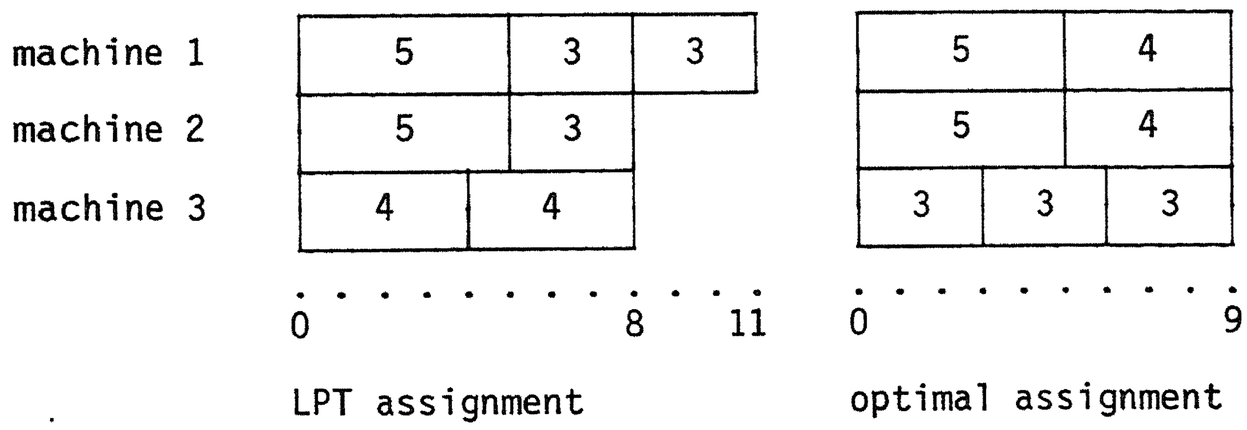


Figure 1 Assignments for the example.

$$Z_{LPT} \leq (\frac{4}{3} - \frac{1}{3m})Z.$$

There is a natural minimization analogue for the definition of worst-case performance ratio stated for maximization problems in Section 2. Given a real number $r \geq 1$, a heuristic solves a class of minimization problems within r if for every problem instance it finds a solution with cost no greater than r times the optimal cost. Graham's result is an example with $r = \frac{4}{3} - \frac{1}{3m}$.

Graham's work has spawned a variety of results giving improved heuristics (such as [Coffman *et al.* 1978]) or dealing with variations of the basic model. This research is surveyed in [Garey *et al.* 1978]. In addition, a polynomial approximation scheme for this problem based on partial enumeration is given in [Graham 1966]. Fully polynomial approximation schemes based on dynamic programming and data scaling have been developed for various scheduling problems in [Sahni 1976]. The comprehensive survey of scheduling theory that appears in this volume [Graham *et al.* 1978] also contains a thorough discussion of approximation results.

3.2. Uncapacitated location

Given a set $N = \{1,\ldots,n\}$ of possible facility locations, a set $M = \{1,\ldots,m\}$ of markets, and a nonnegative value c_{ij} for serving market i from a facility at location j, where should K facilities be located to maximize total value? Assuming that each facility can handle any set of markets, each market should be served from the highest value available facility. Then the value of placing a facility at every location in the set $S \subseteq N$ is given by $f(S) = \sum_{i=1}^{m} \max_{j \in S}\{c_{ij}\}$ and the uncapacitated location problem can be stated as

$$Z = \max_{S \subseteq N, |S| \leq K}\{f(S)\}. \qquad \text{(UL)}$$

Examples of this problem include the location of disbursement accounts to maximize payment float and the location of firehouses to maximize the number of neighborhoods reachable within a specified time interval.

Feasible solutions for (UL) can be obtained with a greedy heuristic similar to the one stated in Section 2 for the knapsack problem. This heuristic forms a solution in a single pass by selecting locations sequentially in an order that maximizes the increase in the objective at each step. The first location j_1 is selected to maximize $f(\{j_1\})$. If loca-

tions $j_1,\ldots,j_k$, $k < K$ have been selected, location j_{k+1} is selected to maximize $f(\{j_1,\ldots,j_k,j_{k+1}\})$. This process stops when K locations have been selected or when additional locations do not increase f. Assuming ties in selection are resolved in a well specified way, the value Z_G of a greedy solution is well defined.

The analysis conducted in [Cornuejols *et al.* 1977] shows that the greedy heuristic satisfies

$$Z_G \geq (1 - (\frac{K-1}{K})^K)Z \geq (1 - \frac{1}{e})Z$$

and that this bound is tight for all K. The bound $1 - \frac{1}{e}$ is also tight as K approaches ∞.

[Cornuejols *et al.* 1977] also considered a local improvement or interchange heuristic that begins with some solution set S and attempts to improve it by replacing some element of S by an element of N-S. This process continues as long as improving interchanges can be made. Let Z_I denote the value of a solution set S that cannot be improved by the interchange of an element in S with an element in N-S. Then

$$Z_I \geq \frac{K}{2K-1} Z$$

and this bound is tight for all K.

All heuristics previously considered in this paper have required an amount of computation that is bounded by a low order polynomial of the problem size and certainly computational economy is a principal motivation for using a heuristic. For this reason, it is important to note that the interchange heuristic may require an exponential number of iterations. Consider the sequence of examples (given in [Nemhauser *et al.* 1978]) defined for K = 2,3... by m = K, n = 2K and

$$c_{ij} = \begin{cases} 2^i-1, & j = 2i-1, \\ 2(2^i-1), & j = 2i, \\ 0, & \text{otherwise.} \end{cases}$$

For K = 3 we have

$$C = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 14 \end{bmatrix}$$

and beginning at $S_0 = (1,3,5)$ the interchange heuristic can require $2^{3+1}-(3+2)$ iterations and generate the sequence of solutions (1,3,5), (2,3,5), (3,4,5), (1,4,5), (2,4,5), (2,5,6), (3,5,6), (1,3,6), (2,3,6), (3,4,6), (1,4,6), and (2,4,6). It is easy to verify that a similar sequence exists for general K, beginning at (1,3,5,...,K-3,K-1), ending at (2,4,...,K-2,K) and requiring $2^{K+1}-(K+2)$ iterations.

Interchange heuristics have been given for a variety of other problems, the best known being the heuristic from [Lin & Kernighan 1973] for the traveling-salesman problem. For most of these, including [Lin & Kernighan 1973], it is currently unknown whether or not an exponential number of interchanges can occur.

The uncapacitated location problem also has a minimization version in which c_{ij} is the nonnegative cost of serving market i from location j and one must determine a set S that minimizes $f(S) = \sum_{i=1}^{m} \min_{j \in S}\{c_{ij}\}$ subject to $|S| \leq K$. At first glance the distinction between maximization and minimization might seem to be inconsequential. Both the greedy and interchange heuristics can be modified for the minimization problem and for most optimizing algorithms, the problem of finding a minimizing location set is mathematically equivalent to the maximization problem. Unfortunately, this equivalence does not extend to the analysis of heuristics. Not only are the bounds for the greedy and interchange heuristics invalid for the minimization problem, but it is unlikely that any polynomial-time heuristic can be found that is not arbitrarily bad in the worst-case. More precisely, letting Z denote the minimum value for a location problem and given *any* fixed $r > 1$, the problem of finding a location set with value Z_A satisfying $Z_A \leq rZ$ is NP-hard. This follows because such a heuristic could solve the NP-complete *vertex covering* problem. Given a graph $G = (V,E)$ with vertices $V = \{1,...,n\}$ and edges E, a vertex i is said to cover vertex j if $i = j$ or if $\{i,j\} \in E$. The vertex covering problem asks whether there is a set of K vertices that covers V. We can answer this question with a heuristic that guarantees $Z_A \leq rZ$ for some fixed r by applying the heuristic to the minimization location problem defined by $M = N = \{1,...,n\}$ and

$$c_{ij} = \begin{cases} \frac{1}{n} & \text{if } i = j \text{ or } \{i,j\} \in E, \\ r+1 & \text{otherwise.} \end{cases}$$

It is easy to verify that the answer is yes if and only if $Z_A = 1$.

Similar results have been obtained for other problems. In the case of

the traveling-salesman problem, although there is a polynomial-time heuristic given in [Fisher *et al.* 1978A] for finding a maximum length tour with a worst-case ratio of $\frac{2}{3}$, it is shown in [Sahni & Gonzalez 1976] that approximating the minimum length tour is NP-hard for finite r. For the problem of coloring the vertices of a graph with a minimum number of colors so that no two adjacent vertices receive the same colors, [Garey & Johnson 1976A] gives an intricate construction showing that approximation is NP-hard for any $r < 2$.

All of the results given in this section have been generalized in [Nemhauser *et al.* 1978; Fisher *et al.* 1978B] for a version of problem (UL) that allows a broader class of objective functions. A number of new results are also given in [Nemhauser *et al.* 1978; Fisher *et al.* 1978B], including a polynomial approximation scheme based on partial enumeration and results for a version of (UL) with more complicated constraints. [Nemhauser & Wolsey 1976] shows that the greedy heuristic for the general problem is "best" in a well-defined sense.

3.3. The traveling-salesman problem

Let $G = (V,E)$ be a complete graph with vertices $V = \{1,\ldots,n\}$ and edges $E = \{\{i,j\} \mid 1 \le i < j \le n\}$. Let c_{ij} be the length of the edge $\{i,j\}$. The well-known traveling-salesman problem requires an optimal tour, a minimum length cycle passing through each vertex exactly once.

Many heuristics have been proposed for this model. When the edge lengths satisfy the triangle inequality ($c_{ij} \le c_{ik}+c_{kj}$ for all i, j, and k) the best worst-case performance is achieved by the following heuristic recently given in [Christofides 1978]. Let $E_{ST} \subseteq E$ be the edges of a minimum spanning tree and $T \subseteq E$ the edges of a minimum length tour. For any $S \subseteq E$ let $Z(S) = \sum_{\{i,j\}\in S} c_{ij}$. It is clear that $Z(E_{ST}) \le Z(T)$ since the deletion of any edge from T will produce a spanning tree.

Let $V_0 \subseteq V$ be the vertices which have odd degree in the tree E_{ST} and let $E_0 = \{\{i,j\} \in E \mid i \in V_0,\ j \in V_0\}$. Since $k = |V_0|$ is even, the graph $G_0 = (V_0,E_0)$ has a perfect matching (an edge set for which each vertex has degree 1). Let $E_{PM} \subseteq E$ be the edges of a minimum length perfect matching of G_0 and without loss of generality, assume that $T_0 = \{e_{12},e_{23},\ldots,e_{k-1,k},e_{k1}\}$ are the edges of a minimum length tour of G_0. Then

$$Z(E_{PM}) \le \frac{1}{2} Z(T_0) \le \frac{1}{2} Z(T).$$

The first inequality follows because $\{e_{12},e_{34},e_{56},\ldots,e_{k-1,k}\}$ and $\{e_{23},e_{45},\ldots,e_{k-2,k-1},e_{k1}\}$ are perfect matchings. The second inequality follows because the edge lengths satisfy the triangle inequality.

Define the multigraph $G' = (V,E_{PM} \cup E_{ST})$. Since every vertex of G' has even degree there is a closed walk (a cycle passing through each vertex at least once) of length $Z_{ST}+Z_{PM} \leq \frac{3}{2} Z$ consisting of exactly the edges in $E_{PM} \cup E_{ST}$. By the triangle inequality the walk may be converted to a tour without increasing its length.

In [Cornuejols & Nemhauser 1978] it is shown that the performance bound of $\frac{3}{2}$ is tight and as a function of n may be sharpened to $\frac{3\lfloor n/2 \rfloor - 1}{2\lfloor n/2 \rfloor}$. Other analyses of traveling-salesman problem heuristics have been conducted in [Rosenkrantz *et al.* 1977; Fisher *et al.* 1978A].

ACKNOWLEDGMENTS

The author is indebted to Abba Krieger, Alexander Rinnooy Kan and Dorit Hochbaum for their helpful comments on an early draft of this paper. This research was supported in part by NSF grant ENG76-20274 to the University of Pennsylvania.

AN ALGORITHM TO PREVENT THE PROPAGATION OF CERTAIN DISEASES AT MINIMUM COST

A. HAJNAL
Hungarian Academy of Sciences, Budapest, Hungary

L. LOVÁSZ
Jőzsef Attila University, Szeged, Hungary

ABSTRACT

Given n rabbits $R_1,\ldots,R_n$, each having a different disease, and m plates $P_1,\ldots,P_m$, smeared with different radioactive isotopes, we want to carry out nm experiments of placing each rabbit on each plate. Protective membranes have to prevent the rabbits from infecting each other or a plate and the isotopes from getting on another plate or on an animal. In each experiment, we may put an arbitrary number of membranes on the plate under the rabbit. The same membrane may be used more than once, but if an infected surface touches another surface, an animal, or a plate, then the infection carries over. The problem is to design the order of the experiments and the use of the membranes so as to minimize the total number of membranes used. Restricting ourselves to the case $n = m = 6k$, we present an algorithm using $7k+1$ membranes and prove that every algorithm needs at least $7k$ membranes.

We have n experimental rabbits $R_1,\ldots,R_n$, each having a different disease. We have m plates $P_1,\ldots,P_m$, smeared with different radioactive isotopes. We want to carry out nm experiments of placing each rabbit on each plate; the order in which this is done is arbitrary. We use protective membranes whose purpose is multitude: they have to prevent the rabbits from infecting each other or a plate, and the isotopes from getting on another plate or on an animal. In each experiment, we may put an arbitrary number of protecting membranes on the plate under the rabbit. The same membrane may be used more than once, but one has to be careful because if a surface of a membrane is infected by the disease of an animal or by an isotope and it touches another surface, an animal, or a plate, the infection carries over. There is no way to clean the membranes.

The problem is to design the order of the experiments and the use of the membranes so as to minimize the total number of membranes used. (This problem has a well-known more practical interpretation, which the authors are too shy to formulate.)

We shall restrict ourselves to the case when $n = m = 6k$. Distinguishing 36 possibilities modulo 6, one could get a full analysis along the same lines, but this is left to the applier of the theory. Also note that there is a difference of 1 between the upper and lower bounds we shall prove on the minimum number of membranes. To fill in this gap needs further work in this fertile area.

THEOREM. *If* $n = m = 6k$, *then there is an algorithm using* $7k+1$ *membranes. On the other hand, every algorithm needs at least* $7k$ *membranes.*

(The general result would be $\frac{2}{3}\min\{m,n\} + \frac{1}{2}\max\{m,n\} + O(1)$.)

Proof. I. First we present the algorithm. Let us relabel the rabbits by $R_1,\ldots,R_{3k},R'_1,\ldots,R'_{3k}$, and the plates by $P_1,\ldots,P_{2k},P'_1,\ldots,P'_{2k},P''_1,\ldots,P''_{2k}$. Label the membranes by $M_1,\ldots,M_{3k},N_1,\ldots,N_{2k},N'_1,\ldots,N'_{2k}$ and J (J for Jolly Joker). Let us distinguish an "upper" and a "lower" surface of each membrane (although there is no difference in use).

During the run of the experiments, a surface can be in three different states: it can be *clean*, *infected* (by a disease or isotope, but only one), or *dirty* (infected by at least two diseases and/or isotopes). Clearly we may assume that the lower surface of J is dirty right at the beginning; the other side will remain clean throughout.

Let us denote the placing of rabbit R on plate P by (PR). If we want a full description of the experiment we write $(PM_1\bar{M}_2\bar{M}_3M_4R)$, meaning that the membranes M_1, M_2, M_3, M_4 are used, in this order, with M_2 and M_3 upside down.

The experiments are carried out in six turns.

Turn 1. All experiments $(P_iN_iM_jR_j)$ $(1 \le i \le 2k,\ 1 \le j \le 3k)$.

After turn 1, the lower surface of N_i gets infected by P_i, and the upper surface of M_j gets infected by R_j, for all i and j.

Turn 2. All experiments $(P'_iN'_iM_jR_j)$.

After turn 2, the lower surface of N'_i gets infected by P'_i.

Turn 3. All experiments $(P''_i\bar{N}'_iJM_jR_j)$.

The upper surface of N'_i gets infected by P''_i, its lower surface gets dirty.

Turn 4. All experiments $(P_iN_i\bar{J}\bar{M}_jR'_j)$.

The upper surface of M_j gets dirty, its lower surface gets infected by R'_j.

Turn 5. All experiments $(P'_i\bar{N}_i\bar{M}_jR'_j)$.

The upper surface of N_i gets infected by P'_i, its lower surface gets dirty.

Turn 6. All experiments $(P''_i\bar{N}'_i\bar{M}_jR'_j)$.

It is straightforward to check that all experiments are feasible in the sense that no infection and no isotopes can be carried over to any other animal or plate.

II. Consider now an arbitrary algorithm solving the problem and consider a side of a membrane. It may happen that (a) it remains clean throughout, or (b) it gets first infected by a disease or isotope, or (c) it gets dirty without being first infected. In case (b), let us assign to the given side of the membrane the rabbit or plate which infects it first. In cases (a) and (c) we assign an arbitrary rabbit to the given side of the membrane.

Also let us call the side which gets infected first the *first side*. Again, in case of ambiguity we call either one of the sides first.

Now we represent the situation by a graph G. The vertices of G are the rabbits and the plates. For each membrane M, we draw a directed line $\vec{M} = \overrightarrow{XY}$ from the vertex X assigned to its first side to the vertex Y assigned to its other side.

Let us make some observations.

(1) G has no isolated vertices.

In fact, the first experiment in which a given vertex V is involved yields a membrane with a side infected by V, *i.e.* an edge adjacent to V.

(2) There cannot exist two edges $\overrightarrow{R_1R_2}$ and $\overrightarrow{P_1P_2}$ such that R_1, R_2 are rabbits, P_1, P_2 are plates and these edges form a single-edge component of G.

For suppose that there are such edges. Consider the experiments (P_2R_1) and

(P_1R_2). Without loss of generality we may assume that (P_2R_1) is carried out earlier. Since $\overrightarrow{P_1P_2}$ is a component of G, in this experiment the membrane on the bottom must be $\overrightarrow{P_1P_2}$. But by the definition of the orientation of G, by this time the other side of this membrane must be infected by isotope P_1. So the rabbit R_1 cannot have been placed directly on this membrane. Consider the second membrane from below. The bottom side of this could not be clean or P_1-infected before the experiment, because then it would yield a second edge incident with P_1, in contradiction with the assumption that $\overrightarrow{P_1P_2}$ forms a component of G just by itself. So this bottom side was infected by something different from P_1 or was dirty. Hence after the experiment the P_1-side of $\overrightarrow{P_1P_2}$ gets dirty. But then there is no way to carry out (P_1R_2).

(3) Let $\overrightarrow{P_1R_1}$ and $\overrightarrow{P_2R_2}$ be edges such that P_1,P_2 are plates, R_1,R_2 are rabbits and these edges form single-edge components of G. Then one of them must be oriented from the rabbit to the plate and the other one conversely.

This follows by the same argument as (2).

We are in the position now to make a computation. Let $G_1,\ldots,G_t$ be the connected components of G, and let p_i be the number of vertices in G_i. Let, say, $G_1,\ldots,G_s$ be those components having two points and one edge. Then G_j $(s < j \leq t)$ has at least $\frac{2}{3}\,p_j$ edges. So the number of edges is at least

$$s + \frac{2}{3}(p_{s+1}+\ldots+p_t) = s + \frac{2}{3}(12k-2s) = 8k - \frac{s}{3}.$$

But from (2) and (3) we know that there are at most two single-edge components connecting rabbits to plates, and that either the rabbits or the plates span no such components. Hence $s \leq 3k+1$, and so the number of edges is at least

$$8k - \frac{1}{3}(3k+1) = 7k - \frac{1}{3}.$$

This completes the proof. □

FAST APPROXIMATION ALGORITHMS FOR KNAPSACK PROBLEMS

E.L. LAWLER
University of California, Berkeley, U.S.A.

ABSTRACT

Fully polynomial approximation algorithms for knapsack problems are presented. These algorithms are based on ideas of Ibarra and Kim, with modifications which yield better time and space bounds, and also tend to improve the practicality of the procedures. Among the principal improvements are the introduction of a more efficient method of scaling and the use of a median-finding routine to eliminate sorting. The 0-1 knapsack problem, for n items and accuracy $\varepsilon > 0$, is solved in $O(n \log(1/\varepsilon) + 1/\varepsilon^4)$ time and $O(n + 1/\varepsilon^3)$ space. The *unbounded* knapsack problem is solved in $O(n + 1/\varepsilon^3)$ time and $O(n + 1/\varepsilon^2)$ space. For the *subset-sum* problem, $O(n + 1/\varepsilon^3)$ time and $O(n + 1/\varepsilon^2)$ space, or $O(n + (1/\varepsilon^2)\log(1/\varepsilon))$ time and space, are achieved. The *multiple-choice* problem, with m equivalence classes, is solved in $O(nm^2/\varepsilon)$ time and space.

CONTENTS

1. INTRODUCTION 111

2. A BASIC OPTIMIZATION PROCEDURE 114

3. SCALING OF PROFIT SPACE 117

4. A LOWER BOUND ON P^* 118

5. SEPARATION OF ITEMS 120

6. COMPUTATION OF ϕ-VALUES 123

7. DISCARDING SUPERFLUOUS ITEMS 124

8. AN IMPROVED METHOD OF SCALING 125

9. MODIFICATION OF THE LARGE-ITEM COMPUTATION 127

10. SUMMARY OF ALGORITHM 129

11. "BOOTSTRAPPING" THE ALGORITHM 130

12. THE UNBOUNDED PROBLEM 132

13. SUBSET-SUM PROBLEM 133

14. MULTIPLE-CHOICE PROBLEMS 135

15. SEPARABLE NONLINEAR FUNCTIONS 137

16. FURTHER EXTENSIONS 138

17. CONCLUDING REMARKS 139

ACKNOWLEDGMENTS 139

1. INTRODUCTION

The 0-1 knapsack problem is as follows: Given n pairs of positive integers (p_j,a_j) and a positive integer b, find $x_1,x_2,\ldots,x_n$ so as to

$$\text{maximize} \quad P = \sum_j p_j x_j$$

$$\text{subject to} \quad A = \sum_j a_j x_j \le b,$$

$$x_j \in \{0,1\}.$$

We may think of j as indexing *items*, with associated *profits* p_j and *weights* a_j. The object is to find the most profitable possible selection of items which can be made to fit into a knapsack with capacity b. One variation of the problem permits items to be chosen with repetition. That is, x_j is permitted to be any nonnegative integer. This is sometimes called the *unbounded* knapsack problem.

There are well-known methods for solving the 0-1 knapsack problem which have worst-case running time of O(nb) [Bellman & Dreyfus 1962]. However, these do not qualify as "polynomial-bounded" algorithms, because the running time is not bounded by a polynomial in the length of the encoding of input data. Thus O(n log b) is a polynomial bound, but not O(nb). (Unless data are encoded in unary notation, a possibility we disregard here, *cf.* [Garey & Johnson 1978B]). In fact, the problem is known to be NP-complete, even in the case of the *subset-sum* problem, where $p_j = a_j$, for all items. Hence it is very unlikely that any polynomial-bounded algorithm exists.

However, it is possible, within polynomial time, to find a solution which is arbitrarily close to optimum. An approximation algorithm for this purpose receives two inputs: one is the encoded set of data for a problem instance, *i.e.* pairs (p_j,a_j), and the other is a number ε, $0 < \varepsilon \le 1$, which prescribes the degree of accuracy required. The algorithm then produces a solution with profit P, such that if P^* is the value of an optimal solution,

$$P^* - P \le \varepsilon P^*.$$

If for every fixed ε, the algorithm operates in time bounded by the length of the encoded input, the algorithm is a "polynomial time approximation scheme". If the algorithm operates in time bounded by a polynomial in the length of the encoded input and in $1/\varepsilon$, it is said to be "fully polynomial" [Garey & Johnson 1976C, 1978B].

Ibarra and Kim [Ibarra & Kim 1975A] have presented fully polynomial approximation algorithms for the 0-1 knapsack problem, and variations. Their most efficient algorithm utilizes a lower bound P_0, $P_0 \leq P^* \leq 2P_0$, to separate items according to profits into a class of "small" items and a class of "large" items. The problem is then solved for the large items only, with profits scaled by a suitably chosen scale factor. Feasible solutions of large items are then joined with sets of small items, and the feasible solution with the largest total profit is selected.

By noting that it is possible to obtain a bound, independent of n, on the number of large items which need to be considered for their computation, Ibarra and Kim claim a bound of

$$O(n \log n + \frac{1}{\varepsilon^4} \log(\frac{1}{\varepsilon})) \tag{1.1}$$

on running time and

$$O(n + \frac{1}{\varepsilon^3}) \tag{1.2}$$

on space requirements.

In this paper we elaborate on the Ibarra-Kim approach, introducing a number of improvements which yield better time and space bounds and also tend to enhance the practicality of the procedures. Among the modifications proposed are the use of a median-finding routine to eliminate sorting and a more efficient scaling technique. A number of other modifications are proposed, including alternative data structures and a different method for carrying out the large-item computation.

As a result of these changes, we are able to obtain the following bounds. For the 0-1 problem: $O(n \log(1/\varepsilon) + 1/\varepsilon^4)$ time and $O(n + 1/\varepsilon^3)$ space. For the *unbounded* problem: $O(n + 1/\varepsilon^3)$ time and $O(n + 1/\varepsilon^2)$ space. For the *subset-sum* problem: $O(n + 1/\varepsilon^3)$ time and $O(n + 1/\varepsilon^2)$ space, or $O(n + (1/\varepsilon^2)\log(1/\varepsilon))$ time and space. The *multiple-choice* problem, with m equivalence classes, is solved in $O(nm^2/\varepsilon)$ time and space.

The organization of this paper is as follows. In Section 2 a basic optimization procedure is described. Modifications of this procedure are presented in Sections 3-11. These yield various approximation algorithms for the 0-1 knapsack problem. In Sections 12 through 15, algorithms are outlined for related problems, including the *unbounded* problem, the *subset-sum* problem, and *multiple-choice* knapsack problems. Concluding sections, 16 and 17, indicate possible further extensions and directions for future

research.

Comment. Complexity estimates are based on the assumption that computer word length is sufficient to accommodate numbers as large as P^*, b, and n. Arithmetic operations on numbers as large as these are assumed to require constant time. Thus, factors such as log b do not appear in bounds such as (1.1) and (1.2). However, we shall *not* assume word length on the order of $1/\varepsilon$ or n bits (numbers as large as $2^{1/\varepsilon}$ or 2^n). □

2. A BASIC OPTIMIZATION PROCEDURE

We begin with the description of an optimization algorithm for the 0-1 knapsack problem. All of the approximation algorithms presented in this paper are modifications or adaptations of this basic procedure.

One way to solve the 0-1 knapsack problem is to generate a list of all feasible combinations of profit and weight. Each such combination is represented by a pair (P,A), for which there is a subset of items S with

$$\sum_{j\in S} p_j = P,$$

$$\sum_{j\in S} a_j = A \leq b.$$

The value of P^* is then given by a pair (P,A) for which P is maximum.

The list can be generated in n iterations as follows. Initially place only the pair (0,0) in the list. Then, at iteration j, form from each pair (P,A) a "candidate" pair $(P+p_j,A+a_j)$, if $A+a_j \leq b$, and place the new pair in the list if it does not duplicate an existing one. The result is that at the end of iteration j, each pair in the list represents a feasible profit-weight combination for some subset of items $S \subseteq \{1,2,\ldots,j\}$, and each subset is represented by a pair.

This procedure is unnecessarily inefficient, because it generates many pairs which are not needed to determine an optimal solution. It clearly does not affect the computed value of P^* if "dominated" pairs are discarded. That is, if (P,A) is in the list, one may eliminate any pair (P',A'), where $P \geq P'$, and $A \leq A'$. After dominated pairs are eliminated, each remaining pair (P,A) satisfies the following conditions at the end of iteration j: P is the largest attainable profit for a subset of items $S \subseteq \{1,2,\ldots,j\}$, with weight at most equal to A, and A is the smallest attainable weight for a subset of items with profit at least equal to P.

The procedure is now revised as follows. At the end of each iteration, the pairs (P,A) are in strictly increasing order of P and of A. To perform iteration j, produce a candidate pair $(P+p_j,A+a_j)$ for each pair (P,A), provided $A+a_j \leq b$. Since the list is in increasing order of A, the production of candidate pairs can be terminated whenever a pair (P,A) is reached for which $A+a_j > b$. Then merge the existing list and the list of candidates, discarding dominated pairs in the process. This is easily accomplished, because of the ordering of the P's and A's. At the end of iteration n, the last pair in the list is (P^*,A^*), where A^* is the minimum attainable weight

for an optimal solution.

There are various data structures and list merging techniques which are suitable for processing the list of pairs. In any case, it is clear that at each iteration the running time and space requirements are linearly proportional to the number of pairs existing in the list at the beginning of the iteration. An upper bound on the number of pairs in the list is $\min\{P^*,b\}$. Hence an upper bound on the running time for the overall procedure, in the worst case, is $O(n \min\{P^*,b\})$, and an upper bound on space requirements is $O(n + \min\{P^*,b\})$. (The term n in the space bound accounts for the storage of input data.)

Up to this point, we have ignored the problem of constructing an optimal solution, *i.e.* determining a set S^* corresponding to the last pair (P^*,A^*) in the list at the end of iteration n. There are at least two methods for doing this.

A very straightforward method is to convert each pair (P,A) to a triple (P,A,S), where S is a list of indices of items such that $\Sigma_{j\in S} p_j = P$, $\Sigma_{j\in S} a_j = A$. Initially, only the triple $(0,0,\emptyset)$ is placed in the list. Thereafter, at iteration j, each candidate triple is of the form $(P+p_j, A+a_j, S \cup \{j\})$. This has the effect of increasing the space bound to $O(n \min\{P^*,b\})$, Since S may be O(n) in size. Forming the set $S \cup \{j\}$ may be assumed to require O(n) time, thereby also adding an extra factor of n to the time bound, running it up to $O(n^2 \min\{P^*,b\})$.

We choose not to provide an explicit representation of the set S corresponding to a pair (P,A). Instead, we propose to construct S by "back-tracing" through secondary data structure in the form of a rooted tree.

The necessary data structures are indicated in Figure 1. Each entry in the list of (P,A) pairs has four components: a P value, an A value, a pointer to the next entry in the list, and a pointer to a node in the rooted tree. Each tree node has two components: an item index j and a pointer to its parent node.

To find the set S for an entry (P,A) in the list, one goes to the tree node indicated by its pointer and then reads off the item indices associated with nodes on the path to the root. Thus, for the entry at the head of the list in Figure 1, $S = \{6,4,3,1\}$. It is easily seen that S can be constructed in O(n) time.

Initially the list contains only the pair (0,0) and the tree pointer for this pair is directed to the root of the tree. Thereafter, whenever a pair $(P+p_j, A+a_j)$ is added to the list of (P,A) pairs, the tree pointer for

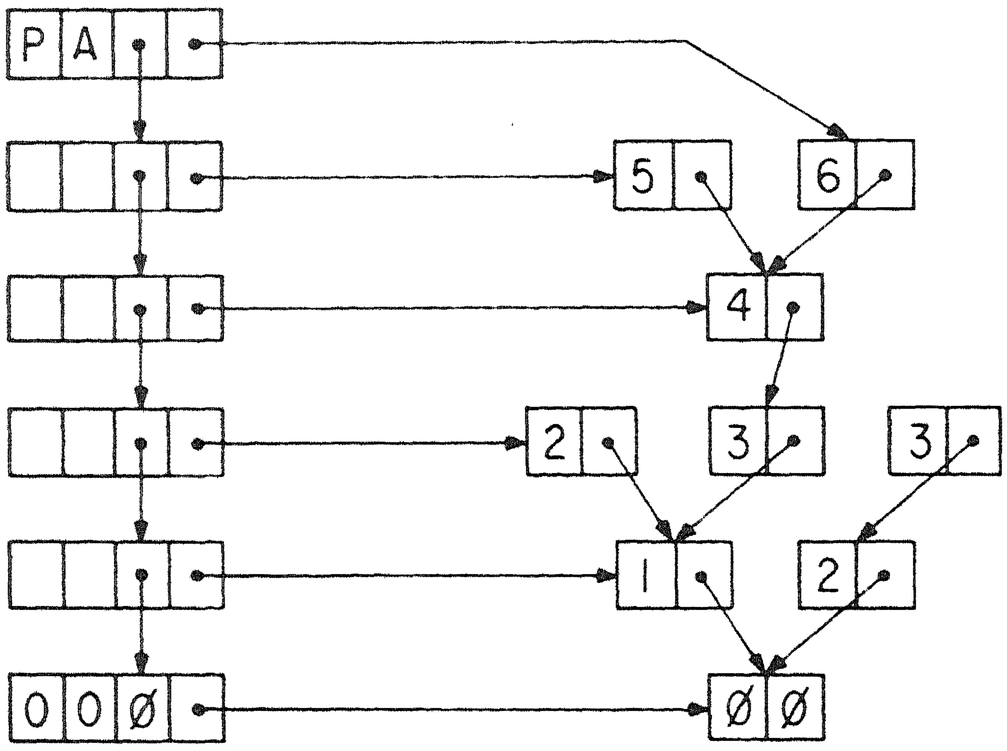


Figure 1 Data structures for list of (P,A) pairs and tree for backtracing.

$(P+p_j, A+a_j)$ is directed to a new tree node with associated item index j. The pointer, for this new node is directed to the node pointed to by (P,A). It is clear that these operations can be implemented in constant time for each new pair $(P+p_j, A+a_j)$, or in $O(n \min\{P^*, b\})$ time overall. Moreover, the tree requires $O(n \min\{P^*, b\})$ space.

3. SCALING OF PROFIT SPACE

One way to make the computation more efficient is to reduce the number of distinct P (or A) values which may occur in the pairs (P,A). The simplest method to accomplish this is to replace each p_j coefficient by

$$q_j = \left\lfloor \frac{p_j}{K} \right\rfloor,$$

where K is a suitably chosen scale factor. Then P^*/K replaces P^* in the time and space bounds given above.

How large can we make K, and be assured that the solution we obtain differs from the optimum by no more than εP^*? We note the inequality,

$$Kq_j \leq p_j < K(q_j+1). \tag{3.1}$$

It follows that for any set S,

$$\sum_{j\in S} p_j - K \sum_{j\in S} q_j < K|S|.$$

Hence if we can insure that $K|S^*| \leq \varepsilon P^*$, where S^* is an optimal solution, then K will be a valid scale factor: the solution found by the computation outlined in the previous section will be within the prescribed accuracy $\varepsilon > 0$. But surely $|S^*| \leq n$ and $P^* \geq p_{max}$, where

$$p_{max} = \max_j\{p_j\}.$$

(We can assume $a_j \leq b$ for all items.) Hence we may choose

$$K = \frac{1}{n} \varepsilon p_{max}. \tag{3.2}$$

Now $P^* \leq n\, p_{max}$, so

$$\frac{P^*}{K} \leq \frac{n^2}{\varepsilon}.$$

Substituting n^2/ε for P^* in the bounds obtained in the previous section, we obtain time and space bounds of $O(n^3/\varepsilon)$, with assurance that the relative error of the solution we obtain does not exceed ε, as specified. (Hereafter we ingnore the role of b in time and space bounds.) We have thus obtained a fully polynomial approximation algorithm.

4. A LOWER BOUND ON P^*

It is evident that a lower bound on P^* better than p_{max} will enable us to increase the size of the scale factor K and thereby improve the efficiency of the computation.

If we relax the conditions $x_j \in \{0,1\}$ to $0 \le x_j \le 1$, a linear programming problem results. It can be solved quite simply: First sort the items in nonincreasing p_j/a_j ratio order, so that, without loss of generality, $p_1/a_1 \ge p_2/a_2 \ge \ldots \ge p_n/a_n$. Then place the items in the knapsack in order until either (a) the items are exhausted or (b) the capacity is exactly used up or (c) it is necessary to fractionalize one item to use up the capacity exactly. In cases (a) and (b), an optimal solution is obtained, and it is unnecessary to proceed further. So suppose case (c) occurs and $a_1+a_2+\ldots+a_j < b$ but $a_1+a_2+\ldots+a_j+a_{j+1} > b$. We assert that

$$P_0 \le P^* < 2P_0, \tag{4.1}$$

where

$$P_0 = \max\{p_1+p_2+\ldots+p_j, p_{max}\}.$$

This is because $p_1+p_2+\ldots+p_j \le P^*$, $p_{j+1} \le p_{max} \le P^*$, but $p_1+p_2+\ldots+p_{j+1} > P^*$. Replacing p_{max} by P_0 in (3.2), we obtain

$$K = \frac{1}{n}\,\varepsilon P_0 \tag{4.2}$$

and find that

$$\frac{P^*}{K} \le \frac{2n}{\varepsilon}.$$

The computation can now be carried out in $O(n^2/\varepsilon)$ time and space, exclusive of the time required to sort the items in p_j/a_j order.

But sorting is not necessary to compute P_0. This can be done in $O(n)$ time by employing a median-finding algorithm as follows: First compute the ratio p_j/a_j for all items. Then find the median of these ratios. (All ratios are considered to be distinct; if ties occur, the item with the smaller index is considered to have the smaller ratio.) Suppose the median ratio is r and let

$$J = \{j \mid p_j/a_j \ge r\}. \tag{4.3}$$

If $\sum_{j\in J} a_j >(<) b$, find the median ratio in (the complement of) J until the largest set J is found such that $\sum_{j\in J} a_j \leq b$.

Case (a) above occurs if J contains all items. Case (b) occurs if $\sum_{j\in J} a_j = b$. Otherwise, case (c) occurs and $P_0 = \max\{\sum_{j\in J} p_j, p_{max}\}$.

There are median-finding routines which require only O(n) time [Blum *et al.* 1973]. This procedure requires O(log n) applications of such a routine, but these are carried out over sets which contain n,n/2,n/4,... elements. It is thus evident that the computation of P_0 requires only O(n) time and space.

In the next section, we shall have need for a procedure which will fill the knapsack to any desired capacity b', $0 \leq b' \leq b$, just as we filled it to capacity b in computing P_0. This means finding the smallest ratio r such that $\sum_{j\in J} a_j \leq b'$, where J is defined as in (4.3). We shall let $\phi(b')$ denote the total profit of the items so placed in the knapsack: $\phi(b') = \sum_{j\in J} p_j$.

5. SEPARATION OF ITEMS

The existence of the lower bound P_0 enabled us to reduce the time bound from $O(n^3/\varepsilon)$ to $O(n^2/\varepsilon)$. A technique due to Sahni [Sahni 1975] and employed by Ibarra and Kim enables us to reduce the bound still further, to $O(n/\varepsilon^2)$.

Comment. The reader may question whether n/ε^2 is "better" than n^2/ε. The bounds stated are intended to emphasize asymptotic behavior in n, rather than ε. The algorithm we are about to describe will, in fact, be bounded by $O(n^2/\varepsilon)$, as well as $O(n/\varepsilon^2)$. □

The approach is as follows: First compute P_0, as described in the previous section. Use P_0 to determine a threshold value T. Items with $p_j \leq T$ are considered "small", and those with $p_j > T$ "large". Solve the problem, using the large items only, with some appropriately chosen scale factor K. This yields a final list of (Q,A) pairs, where Q denotes total scaled profit.

For each pair (Q,A) in the final list, fill in the remaining knapsack capacity b-A with small items, as indicated in the previous section. These small items yield total profit $\phi(b-A)$. The approximate solution, a combination of large and small items, is chosen to yield profit P, where

$$P = \max_{(Q,A)}\{KQ + \phi(b-A)\}. \tag{5.1}$$

Intuitively, it seems reasonable to divide the permissible error equally between the small items and the large items. This suggests setting $T = \frac{1}{2}\varepsilon P_0$, which has the following result: When space b-A is filled with small items, at most one item with profit no greater than $\frac{1}{2}\varepsilon P_0$ (half the estimated permissible error) will be omitted. The scale factor K should then be chosen so that the total error contributed by the large items is also no more than one half the permissible amount. Below we justify this intuitive argument, and prove that this choice of T also enables us to maximize the scale factor K.

Suppose there exists an optimal solution in which the large items contribute profit P_L and weight A_L and the small items P_S and A_S. Let Q_L be the sum of the scaled profit coefficients contributing to P_L, using scale factor K. It should be apparent, without need of proof, that the final list for the large-item computation must contain a pair (Q,A), dominating (Q_L,A_L), *i.e.*, $Q \geq Q_L$, $A \leq A_L$. Thus P, determined by (5.1), must be such that

$$P \geq KQ + \phi(b-A) \geq KQ_L + \phi(b-A) \tag{5.2}$$

The number of large items contributing to the optimal solution cannot exceed P_L/T, where

$$\frac{P_L}{T} \leq \frac{P^*}{T}. \tag{5.3}$$

Employing the inequality (3.1),

$$Kq_j \leq p_j \leq K(q_j+1),$$

with (5.3), we obtain

$$P^* = P_L + P_S < K(Q_L + \frac{P^*}{T}) + P_S. \tag{5.4}$$

From (5.2) and (5.4) it follows that

$$P^* - P < \frac{K}{T}P^* + P_S - \phi(b-A). \tag{5.5}$$

But $b-A > b-A_L$. If our procedure exactly filled the capacity b-A with small items, then $\phi(b-A) \geq P_S$. If it left some unused capacity, the profit of the item which could not be fit in was no greater than T, and

$$\phi(b-A) + T \geq P_S. \tag{5.6}$$

From inequalities (5.5) and (5.6), we obtain

$$P^* - P < \frac{K}{T}P^* + T. \tag{5.7}$$

It follows that T and K should be chosen to insure that $(K/T)P^* + T \leq \varepsilon P^*$, which can be done by letting $K/T = \lambda\varepsilon$ and $T = (1-\lambda)\varepsilon P_0 \leq (1-\lambda)\varepsilon P^*$ for any λ, $0 \leq \lambda < 1$. Assuming $T \neq 0$, this means $K = \lambda(1-\lambda)\varepsilon^2 P_0$. Since we wish to maximize K, we choose $\lambda = \frac{1}{2}$, yielding

$$K = \frac{1}{4}\varepsilon^2 P_0, \quad T = \frac{1}{2}\varepsilon P_0. \tag{5.8}$$

This confirms our intuition as to the correct choice of T and K.

Comment. In [Ibarra & Kim 1975A], the choice of K and T was, in effect:

$$K = \frac{2}{9}\varepsilon^2 P_0, \quad T = \frac{2}{3}\varepsilon P_0,$$

corresponding to a choice of $\lambda = \frac{1}{3}$. □

Observe that

$$\left\lceil \frac{P^*}{K} \right\rceil \leq \frac{2P_0}{\frac{1}{4}\varepsilon^2 P_0} = \frac{8}{\varepsilon^2},$$

so the size of scaled profit space is now $O(1/\varepsilon^2)$, instead of $O(n/\varepsilon)$. Time and space for the large-item computation are bounded by $O(n/\varepsilon^2)$, since there are n iterations.

We have yet to discuss the computation of the ϕ-values at the end of the large-item computation. This can be accomplished in $O(n \log(1/\varepsilon))$ time, as we shall show in the next section. Hence the overall time and space bounds for the procedure are $O(n/\varepsilon^2)$. Now notice that if $T < 1$ (all items "large") as determined by (5.8), then certainly $K < 1$ in (4.2). Hence there is no instance in which the method of the previous section provides a useful scale factor (greater than unity) and the present one does not. Moreover, if $T = \frac{1}{2}\varepsilon P_0 < 1$, then one should simply solve the problem optimally, which can be done in $O(n/\varepsilon)$ time, since $P^* < 4/\varepsilon$.

Next notice that P^*/T can be replaced in (5.3) by n, yielding

$$P^* - P < Kn + T.$$

Assuming $\lambda = \frac{1}{2}$, this implies that K should be of the form

$$K = \frac{1}{2n}\varepsilon P_0,$$

which results in

$$\frac{P^*}{K} \leq \frac{4n}{\varepsilon}$$

and an $O(n^2/\varepsilon)$ computation, as in Section 4.

These observations suggest that equations (5.8) should be modified to become

$$K = \max\{\tfrac{1}{4}\varepsilon^2 P_0,\ \tfrac{1}{2n}\varepsilon P_0,\ 1\}, \quad T = \begin{cases} \varepsilon P_0 & \text{if } K = 1, \\ \frac{1}{2}\varepsilon P_0 & \text{if } K > 1. \end{cases} \tag{5.9}$$

Equations (5.9) assure a computation which is time and space bounded by both $O(n^2/\varepsilon)$ and $O(n/\varepsilon^2)$. Moreover, whenever $T < 1$ in (5.9), an *optimal* solution is obtained in $O(n/\varepsilon)$ time. Although in succeeding sections we shall present ideas leading to substantially better asymptotic performance with respect to n, we shall not be able to improve on $O(n^2/\varepsilon)$, for asymptotic performance with respect to ε.

6. COMPUTATION OF ϕ-VALUES

It is easy to compute the ϕ-values in $O(n + 1/\varepsilon^2)$ time, as in [Ibarra & Kim 1975A], once the small items have been placed in p_j/a_j ratio order. However, we eliminated sorting in the computation of P_0, so we do not have an ordered set of small items available as a byproduct. The procedure we propose is as follows.

First find the median p_j/a_j ratio for the set of all small items. Let this ratio be r and let $J = \{j | p_j/a_j \geq r\}$, as before. If $\sum_{j\in J} a_j > b$, throw away the complement of J and repeat, continuing to throw away half-sets until a ratio r is found, such that $\sum_{j\in J} a_j \leq b$. Then search the list of pairs until a pair $(\bar{Q},\bar{A})$ is found, such that $\bar{A} = \max_{(Q,A)}\{A | \sum_{j\in J} a_j \leq b-A\}$ and $\phi(b-\bar{A}) = \sum_{j\in J} p_j$.

At this point one ϕ-value has been found in $O(n)$ time, exclusive of the time required to find the pair $(\bar{Q},\bar{A})$. A set J, $|J| \leq n/2$, has been determined, as has a complementary set $\bar{J}$, $|\bar{J}| \leq n/2$, within the set of remaining small items. Proper subsets of J determine $\phi(b-A)$ for $A > \bar{A}$. Proper subsets of $\bar{J}$, joined with all the items in J, determine $\phi(b-A)$ for $A < \bar{A}$. Thus there are now two subproblems of the same character as the original one.

The difficulty in estimating the remaining time required for the procedure is that we cannot be sure how many pairs are involved for each subproblem: about $4/\varepsilon^2$ for each or $8/\varepsilon^2$ for one and none for the other? We can confirm that time is bounded by $O(n \log(1/\varepsilon))$ by the following analysis.

Let $T(m,n)$ = the time required for median finding, given m pairs and n small items. Then

$$T(m,n) = cn + \max_{1\leq q\leq m}\{T(m-q,\tfrac{n}{2}) + T(q-1,\tfrac{n}{2})\},$$

where c is a constant. Assume $T(m,n) \leq cn \log_2 m$. Then

$$T(m,n) \leq cn + \max_q\{c(\tfrac{n}{2})[\log_2(m-q) + \log_2(q-1)]\}$$

$$\leq cn + cn \log_2(\tfrac{m-1}{2})$$

$$\leq cn \log_2 m,$$

as required.

Thus, median-finding requires at most $O(n \log(1/\varepsilon))$ time, since m is $O(1/\varepsilon^2)$. Other operations required in determining the ϕ-values (such as locating pairs $(\bar{Q},\bar{A})$) are bounded by $O((1/\varepsilon^2)\log(1/\varepsilon))$.

7. DISCARDING SUPERFLUOUS ITEMS

The time and space bounds can be further reduced by the following observations.

The number of distinct q_j values that can exist for the large items is bounded by

$$\left\lceil \frac{P^*}{K} \right\rceil - \left\lceil \frac{T}{K} \right\rceil \leq \frac{8}{\varepsilon^2} - \frac{2}{\varepsilon}.$$

The number of large items with scaled profit q_j which can be contained in any feasible solution is at most

$$n_j = \left\lfloor \frac{P^*}{Kq_j} \right\rfloor.$$

Hence for any scaled profit q_j, one need only consider the n_j items with smallest weight for use in the large-item computation.

For q_j values in the interval $(2/\varepsilon, 4/\varepsilon]$, the average value of n_j is about $3/\varepsilon$. For the interval $(4/\varepsilon, 8/\varepsilon]$, the average value of n_j is about $3/2\varepsilon$. For each successive interval, the average value of n_j is half as large, but the interval contains twice as many q_j values. There are at most $\log_2(P^*/T) \leq \log_2(4/\varepsilon)$ intervals. Hence the number of items which must be considered for the large-item computation is bounded by

$$\frac{6}{\varepsilon^2} \log_2\left(\frac{4}{\varepsilon}\right). \tag{7.1}$$

Identifying the items which need be considered for the large-item computation is simple: Place the large items in at most $\max\{n, 8/\varepsilon^2\}$ buckets, each bucket containing items with the same q_j value. Then apply a median-finding routine to each bucket q_j, to identify the n_j items with smallest weight. This can be done in $O(n)$ time.

We can now substitute $(1/\varepsilon^2)\log(1/\varepsilon)$ for n in the time and space bounds for the large-item computation obtained in Section 5. This yields a time bound of $O(n \log(1/\varepsilon) + (1/\varepsilon^4)\log(1/\varepsilon))$ and a space bound of $O(n + (1/\varepsilon^4)\log(1/\varepsilon))$ for the overall computation. (Note that $O(n)$ space is required to store the input data.)

8. AN IMPROVED METHOD OF SCALING

The method of scaling we have employed up to this point is unnecessarily conservative. The same fixed error, K, has been permitted each scaled profit coefficient q_j. It is more effective to produce coefficients for which the permissible error is somewhat in proportion to the size of the coefficient. This can be done as follows.

Let

$$T = \frac{1}{2}\varepsilon P_0, \quad K = \frac{1}{2}\varepsilon T = \frac{1}{4}\varepsilon^2 P_0, \tag{8.1}$$

as in Section 5. If p_j lies in the interval $(2^k T, 2^{k+1} T]$, $k = 0,1,2,\ldots, \lfloor \log_2(P^*/T) \rfloor$, then let

$$q_j = \left\lfloor \frac{p_j}{2^k K} \right\rfloor 2^k. \tag{8.2}$$

Now notice that

$$Kq_j \le p_j < Kq_j + 2^k K \le Kq_j + \frac{1}{2}\varepsilon 2^k T \le Kq_j + \frac{1}{2}\varepsilon p_j.$$

The above inequality is parallel to (3.1). It follows that we have, in place of inequality (5.4),

$$P^* = P_L + P_S < KQ_L + \frac{1}{2}\varepsilon P_L + P_S.$$

By reasoning similar to that used before, we obtain

$$P^* - P < \frac{1}{2}\varepsilon P^* + T = \frac{1}{2}\varepsilon P^* + \frac{1}{2}\varepsilon P_0.$$

Let $m = \lfloor \log_2(P^*/T) \rfloor$. The size of scaled profit space is bounded by

$$\left\lfloor \frac{P^*}{2^m K} \right\rfloor 2^m \le \frac{P^*}{K} \le \frac{8}{\varepsilon^2}.$$

as before. However, the number of distinct q_j values obtained from each p_j-interval $(2^k T, 2^{k+1} T]$ is at most $(2/\varepsilon)-1$, independent of k. Thus for the scaled profit-space interval $(2/\varepsilon, 4/\varepsilon]$, the number of q_j values is about $2/\varepsilon$, and the average value of n_j is $3/\varepsilon$. For the interval $(4/\varepsilon, 8/\varepsilon]$, the number of q_j values is still $2/\varepsilon$, and the average value of n_j is $3/2\varepsilon$, and so on. It follows that the number of items which must be considered for the large-item computation is bounded by

$$\frac{6}{\varepsilon^2}\left(1 + \frac{1}{2} + \ldots + \frac{T}{P^*}\right) \le \frac{12}{\varepsilon^2}.$$

We have thus eliminated the log factor in (7.1).

Comment. We have dispensed with a more general argument, in which $T = (1-\lambda)\varepsilon P_0$, $K = \lambda\varepsilon T$, as this carries through exactly as in Section 5. Nor shall we confirm that there is no advantage in choosing an integer $a > 2$, and letting

$$q_j = \left\lfloor \frac{p_j}{a^k K} \right\rfloor a^k,$$

if p_j lies in the interval $(a^k T, a^{k+1} T]$. Note that the previous scaling technique is the case in which $a \geq P^*/T$. □

Since there are now $O(1/\varepsilon^2)$ items for the large-item computation, the time bound can be reduced to $O(n \log(1/\varepsilon) + 1/\varepsilon^4)$ and the space bound to $O(n + 1/\varepsilon^4)$. In the next section we shall show how the space bound can be further reduced to $O(n + 1/\varepsilon^3)$, while maintaining the time bound.

9. MODIFICATION OF THE LARGE-ITEM COMPUTATION

We propose to modify the large-item computation to reduce the space required for backtracing. Our plan is as follows.

First sort at most n_j items with a given q_j value into nondecreasing order of weight. This can be done, for all q_j values, in $O((1/\varepsilon^2)\log(1/\varepsilon))$ time. Now notice that there are at most n_j+1 possibilities for each q_j value: Either no items are chosen, the first (smallest-weight) item is chosen, the first two items are chosen, ..., or all (at most n_j) items are chosen. The large-item computation is now carried out in iterations over distinct q_j values (instead of individual items) in strictly decreasing order of q_j.

Initially the list contains only the pair (0,0). At the end of iteration i, each pair (Q,A) indicates the smallest weight A attainable for a subset of items chosen from the i largest q_j values, with total profit at least Q. It also indicates the largest profit Q attainable for a subset of items chosen from the i largest q_j values, with total weight not exceeding A.

Suppose iteration i is for scaled profit q_j and there are n_j items with this scaled profit. For each pair (Q,A) in the list, n_j candidate pairs are formed, corresponding to the choice of 1 item, 2 items, ..., n_j items. These pairs are placed in n_j separate candidate lists. The n_j+1 lists are then merged by means of n_j pairwise merges. Each list entering into a pairwise merge is $O(1/\varepsilon^2)$ in length and the resulting list is also $O(1/\varepsilon^2)$ in length, dominated entries being eliminated in the course of the merge. It follows that the running time for each iteration is bounded by $O(n_j/\varepsilon^2)$. The running time for the large-item computation is thus bounded by $O(1/\varepsilon^4)$, since Σn_j is $O(1/\varepsilon^2)$.

Now let us consider the space bound. At each iteration, the space required by candidate lists is at most $O(n_jN_j)$, where N_j is the length of the list of pairs produced by the previous iteration. This space is bounded by $O(1/\varepsilon^3)$. The number of new entries added to the list of pairs, and hence the number of nodes added to the tree used for backtracing is $O(N_j)$. But N_j is at most 1 for the largest interval, 2 for the second-largest, ..., $2/\varepsilon^2$ for the second-smallest, and $4/\varepsilon^2$ for the smallest. This is because the q_j values in each interval are a power of 2, which reduces the effective size of the scaled profit space from $P^*/K \le 8/\varepsilon^2$ to $2^{-k}P^*/K \le 2^{-k}(8/\varepsilon^2)$, for iterations over q_j values in interval k. The number of q_j values (iterations)

for each interval is at most $T \leq 2/\varepsilon$. It follows that the total number of nodes in the tree is bounded by a number proportional to

$$\frac{2}{\varepsilon}(1 + 2 + \ldots + \frac{4}{\varepsilon^2}) \simeq \frac{16}{\varepsilon^3} . \tag{9.1}$$

Hence total space is bounded by $O(n + 1/\varepsilon^3)$.

Comment. Each of the n_j candidate pairs obtained from a pair (Q,A) can be viewed as corresponding to the choice of a "multiple" item. The indexing scheme and backtracing procedure must be slightly modified. It is not difficult to do this, while staying within the time and space bounds. Hereafter, we shall not mention necessary modifications of the indexing scheme and backtracing procedure, assuming the reader can supply details. □

10. SUMMARY OF ALGORITHM

We now summarize the steps of the approximation algorithm for the 0-1 knapsack problem:

1. Compute p_j/a_j ratios for all items. Compute P_0, using a median-finding routine as indicated in Section 4: O(n) time and space.
2. Determine the threshold T and scale factor K by (8.1). Compute q_j for all large items, using equation (8.2): O(n) time and space.
3. Determine the $O(1/\varepsilon^2)$ items to enter into the large-item computation, by applying a median-finding routine to find the at most n_j items with smallest weight, for each q_j value: O(n) time and space.
4. Sort the at most n_j items for each q_j value in order of nonincreasing weight: $O((1/\varepsilon^2)\log(1/\varepsilon))$ time and $O(1/\varepsilon^2)$ space.
5. Carry out the modified version of the large-item computation described in the previous section: $O(1/\varepsilon^4)$ time and $O(1/\varepsilon^3)$ space.
6. Compute $\phi(b-A)$ fro each pair (Q,A) in the final list, employing a median-finding procedure, as described in Section 6: $O(n \log(1/\varepsilon) + (1/\varepsilon^2)\log(1/\varepsilon))$ time and $O(n + 1/\varepsilon^2)$ space.
7. Find a pair (Q,A) for which $KQ + \phi(b-A)$ is maximum: $O(1/\varepsilon^2)$ time and space.
8. Find the set of large items in the approximate solution by backtracing: $O(1/\varepsilon)$ time and space. The set of small items in the approximate solution is readily available as a byproduct of Step 6.

In succeeding sections we shall indicate how the steps of the algorithm should be modified, for other versions of the knapsack problem.

11. "BOOTSTRAPPING" THE ALGORITHM

A further analysis of the space bound shows that space is bounded by a constant times

$$\frac{P^*}{\varepsilon^3 P_0}.$$

Similarly, time is bounded by a constant times

$$\frac{(P^*)^2}{\varepsilon^4 P_0^2}.$$

It follows that an improvement in the quality of the lower bound P_0 will yield a reduction in the bounds by at least a linear scale factor.

One way to obtain a better bound is to use the algorithm itself to produce approximate solutions which provide better lower bounds. A "bootstrapping" procedure is as follows. Begin with the lower bound P_0 with accuracy $\varepsilon_0 \leq \frac{1}{2}$. Use P_0 to obtain a threshold T_1 and scale factor K_1 for accuracy $\varepsilon_1 < \varepsilon_0$. Apply the approximation algorithm to obtain an approximate solution with profit P_1 and relative error ε_1. Proceed through successive iterations, with accuracies $\varepsilon_0 > \varepsilon_1 > \varepsilon_2 > \ldots > \varepsilon_N = \varepsilon$.

At succesive iterations, the lower bounds P_i, thresholds T_i and scale factors K_i are determined by the relations:

$$P_i \geq (1-\varepsilon_i)P^*,$$

$$T_i = \frac{1}{2}\varepsilon_i P_{i-1} \geq \frac{1}{2}\varepsilon_i(1-\varepsilon_{i-1})P^*,$$

$$K_i = \frac{1}{4}\varepsilon_i^2 T_i \geq \frac{1}{4}\varepsilon_i^2(1-\varepsilon_{i-1})P^*.$$

The running time at iteration i is then bounded by a constant times

$$\frac{(P^*)^2}{\varepsilon_i^4 P_{i-1}^2} \leq \frac{1}{\varepsilon_i^4(1-\varepsilon_{i-1})^2}. \qquad (11.1)$$

If we perform one iteration, as in the preceding, with $\varepsilon_0 = \frac{1}{2}$, $\varepsilon_1 = \varepsilon$, then the quantity (11.1) becomes

$$\frac{4}{\varepsilon^4}.$$

If we perform N iterations, a lower bound on the quantity (11.1) is

$$\frac{1}{\varepsilon^4}.$$

It follows that no matter how such a bootstrapping scheme is arranged, we can expect to improve the time bound by no more than a linear scale factor less than four.

A practical advantage of bootstrapping is that early iterations are likely, in practice, to be quite short, lengthening considerably with each successive iteration. This enables one to halt the procedure, when desired, with an approximate solution known to have accuracy ε_i, where i was the last iteration.

12. THE UNBOUNDED PROBLEM

Recall the *unbounded* knapsack problem is the case in which the variables x_j are not restricted to 0,1 values, but may not be nonnegative integers. A number of simplifications can be made in this case.

First, it is evident that the computation of P_0 and of the ϕ-values is now much more straightforward: One need only identify a single item with maximum p_j/a_j ratio, which can be done with a single scan through the items. A small item with maximum p_j/a_j ratio is all that is needed to compute all the ϕ-values. This can be done in $O(n + 1/\varepsilon^2)$ time.

It is evident we need retain only one large item for each q_j value for the large-item computation, *i.e.* one with minimum weight. However, we must provide for all possible n_j multiplicities of each such item, where

$$n_j = \left\lfloor \frac{P^*}{Kq_j} \right\rfloor \leq \left\lfloor \frac{4}{\varepsilon} \right\rfloor .$$

Ibarra and Kim propose doing this by providing n_j identical copies of the item. A more sensible procedure is to provide only $\lfloor \log_2 n_j \rfloor$ additional copies by "doubling". That is, let the i-th copy of item j be such that

$$p_j^{(i)} = 2^i p_j, \quad a_j^{(i)} = 2^i a_j .$$

All necessary copies of items can be found in $O((1/\varepsilon^2)\log(1/\varepsilon))$ time. It is now necessary to retain only the smallest-weight items, or copy of an item, for each q_j value. This leaves $O((1/\varepsilon)\log(1/\varepsilon))$ items for the large-item computation. The large-item computation now proceeds by iteration over items or q_j values (there is no difference), from largest to smallest. It is evident that this can be carried out in $O(1/\varepsilon^3)$ time. Finally, we note that the secondary data structure used for backtracing in the 0-1 problem can be eliminated. List entries can be of the form (Q,A,j), where j is the index of the item identified with the iteration at which the entry is formed, *i.e.* candidate entries are of the form $(P+p_j, A+a_j, j)$. Solutions can be constructed by simple backtracing using the item indices. Hence we conclude that the unbounded knapsack problem can be solved in $O(n + 1/\varepsilon^3)$ time and $O(n + 1/\varepsilon^2)$ space.

13. SUBSET-SUM PROBLEM

The *subset-sum* problem is as follows: Given n positive integers $p_1, p_2, \ldots, p_n$, and an integer b, does there exist a subset S, such that $\sum_{j \in S} p_j = b$? This can obviously be reduced to a 0-1 knapsack problem of the form

$$\text{maximize} \quad \sum_j p_j x_j$$

$$\text{subject to} \quad \sum_j p_j x_j \leq b,$$

$$x_j \in \{0,1\}.$$

This is the type of problem for which we propose to find an approximate solution with accuracy $\varepsilon > 0$.

We first observe that it is a simple matter to compute P_0. The knapsack may be filled with items in arbitrary order. This clearly requires only O(n) time.

We next observe that it is possible to carry out the large-item computation without consideration of item weights. That is, it is possible to process only lists of scaled profits Q, $Q \leq b/K$, instead of pairs (Q,A). However, we must insure that each entry in our lists is feasible. That is, if S is the set corresponding to Q, where $KQ \leq b$, then $\sum_{j \in S} p_j \leq b$. In order to guarantee this, we propose to round-up in scaling, rather than rounding down, *i.e.* replace (8.2) by

$$q_j = \left\lceil \frac{p_j}{2^k K} \right\rceil 2^k.$$

When this is done, all of the preceding error analysis goes through as before.

Notice that no existing list entry need ever be eliminated by dominance. Hence the data structure used for backtracing requires only $O(1/\varepsilon^2)$ space. (In fact, the secondary data structure can be dispensed with entirely, and the pointers replaced by pointers to other list entries.)

Let us now make a selection of items for the large-time computation. First place the large items in buckets, according to their scaled profits q_j and eliminating any surplus items (more than n_j) in any bucket. This leaves at most $O(1/\varepsilon^2)$ items. The situation now differs from the ordinary 0-1 problem in that the items in each bucket are indistinguishable: they can all be considered to have the same weight. We now want to provide for the choice of any possible number of the items in any bucket. The procedure

is as follows.

Start with the smallest and work upward. If bucket q_j contains an odd number of items, say 2k+1, place k "multiple" items, each with scaled profit $2q_j$, in bucket $2q_j$, discarding any extra items if the capacity of bucket $2q_j$ is exceeded. This leaves one item in bucket q_j. If bucket q is nonempty and contains 2k+2 items, do the same thing, leaving two items. This process requires $O(1/\varepsilon^2)$ time, for all buckets.

We are now left with at most two items with any given scaled profit q_j. The large-item computation can be carried out in a straightforward fashion, in $O(1/\varepsilon^3)$ time and $O(1/\varepsilon^2)$ space. This yields overall time and space bounds of $O(n + 1/\varepsilon^3)$ and $O(n + 1/\varepsilon^2)$.

It is also possible to solve the subset-sum problem in $O(n + (1/\varepsilon^2)\log(1/\varepsilon))$ time and space by applying the computation proposed in [Karp 1975B] to the $O((1/\varepsilon)\log(1/\varepsilon))$ large items. This computation involves the consideration of "intervals" of attainable P-values. We shall not give details of this computation, referring the reader to the reference.

14. MULTIPLE-CHOICE PROBLEMS

Suppose the n items are partitioned into m equivalence classes and it is stipulated that no more than one item (or multiples of one item) may be chosen from each equivalence class. Such a problem is sometimes called a *multiple-choice* knapsack problem.

The author has developed an approximation algorithm for the unbounded multiple-choice problem with time and space bounds of $O(n + (1/\varepsilon^6)\log(1/\varepsilon))$ and $O(n + (1/\varepsilon^4)\log(1/\varepsilon))$. However, this algorithm is rather complicated and very likely can be improved upon. Hence we shall limit our discussion to the 0-1 multiple-choice problem.

Our first observation is that there seems to be no feasible method to compute the lower bound P_0 for the 0-1 multiple-choice problem. One can easily find an item with maximum p_j/a_j ratio in each of the m equivalence classes. But what if these m items do not fill the knapsack to capacity? There are similar, but even more severe difficulties in computing ϕ-values. There seems to be no alternative to returning to the approach of Section 3.

In order to find some sort of lower bound, find an item with maximum profit in each equivalence class. Then fill the knapsack with these m items, in nondecreasing order of profit, until either the items are exhausted or it is not possible to insert another item. (This can be done in O(m) time, using a median-finding routine.)

In the former case, an optimal solution has been found, and there is nothing more to do. In the latter case, m' items have been used, where $1 \le m' < m$. Let the total profit of these m' items be P'. Then

$$P' \le P^* \le \frac{m}{m'} P' \le mP'.$$

We now propose to let P' play the same role as p_{max} in the approach of Section 3. Thus, we let

$$K = \frac{m'}{m} \varepsilon P' \ge \frac{1}{m} \varepsilon P'.$$

Note that the size of scaled profit space is given by

$$\frac{P^*}{K} \le \frac{m^2}{\varepsilon}.$$

The list of pairs (Q,A) is processed with iteration over equivalence classes, rather than single items. The procedure is very similar to that proposed for the large-item computation in Section 9.

Initially the list contains only the pair (0,0). At the end of iteration i, each pair (Q,A) is identified with a feasible solution containing items chosen from equivalence classes 1 through i. Suppose there are n_i items in equivalence class i. To perform iteration i, form n_i candidate items for each pair (Q,A) existing in the list at the end of iteration i-1. These candidate pairs are placed in n_i separate candidate lists. The n_i+1 lists are then merged, eliminating dominated entries.

Iteration i requires $O(n_i m^2/\varepsilon)$ time, $O(m^2/\varepsilon)$ space, and at most $O(m^2/\varepsilon)$ nodes are added to the tree used for backtracing. Hence overall time and space requirements are bounded by $O(nm^2/\varepsilon)$ and $O(n + m^3/\varepsilon)$, respectively.

Comment. The number of items which need be considered from each equivalence class is bounded by the number of distinct q_j values, which is $O(m^2/\varepsilon)$. Hence a time bound of $O(n + m^5/\varepsilon^2)$ can also be obtained. □

15. SEPARABLE NONLINEAR FUNCTIONS

One considerable generalization of the knapsack problem is:

$$\text{maximize} \quad \sum_{j=1}^{m} p_j(x_j)$$

$$\text{subject to} \quad \sum_{j=1}^{m} a_j(x_j) \le b,$$

$$x_j \text{ nonnegative integer},$$

$$x_j \le n_j.$$

Here p_j and a_j are arbitrary real-valued functions, $j = 1,2,\ldots,m$.

By evaluating each function at each feasible integer point, one obtains $n = \sum n_j$ items for a 0-1 multiple-choice knapsack problem, with m equivalence classes. This can be solved, for any prescribed relative error $\varepsilon > 0$, in $O(nm^2/\varepsilon)$ time and space, using the procedure of the previous section.

Note that there is nothing in our theory which cannot accommodate real-valued profits and weights. (Of course, items with negative profits or weights may be neglected, or may cause a knapsack problem to become unbounded, under certain obvious conditions.)

16. FURTHER EXTENSIONS

The knapsack problem arises in many applications. However, it is a greatly specialized version of more general integer programming models for which there is a real need for approximation algorithms.

Certainly the techniques discussed in this paper fall far short of providing a fully polynomial approximation algorithm for multi-constraint problems. The principal reason is that they involve rescaling profit (objective) space instead of weight (constraint) space. Until a new technique is devised, there seems to be no reasonable way to apply the present approach to multi-constraint problems.

That is, unless we are willing to modify our views of optimization and approximation. For example, it is clearly possible to scale weight space to obtain an approximate knapsack solution of the following type: Given $\delta > 0$, find a subset of items S, such that

$$\sum_{j \in S} p_j \geq p^*$$

and

$$\sum_{j \in S} a_j \leq (1+\delta)b.$$

At first glance, the above may seem like an unnatural form of approximation. Possibly this is because we have been taught that constraints are inviolate. But suppose the knapsack problem is being used as a simplified model for, say, project scheduling. A manager seeks to choose projects for a certain period, subject to certain resource constraints (knapsack capacity). The profits associated with the items are real and hard. The constraints are soft and flexible. He certainly wants to earn P^* dollars, if possible. Which type of approximation is more reasonable?

If the notion of constraint approximation is accepted, then it seems feasible to move ahead with the application of known techniques to multi-constraint problems.

17. CONCLUDING REMARKS

It is certainly possible that the time and space bounds presented here can be improved upon. Aside from improvements in factors of $1/\varepsilon$, or $\log(1/\varepsilon)$, there are a number of open questions and directions for future research which suggest themselves.

We have made a few simple assumptions in making time and space bounds. Among these are that arithmetic operations can be performed in constant time on integer operands as large as P^* and b. In the case of the 0-1 knapsack problem, operations on integers as large as n are assumed, for the purpose of finding medians. Can this assumption be removed?

Perhaps a more interesting question is: Can a 0-1 approximation algorithm be found which is "strictly linear" in n, instead of order n $\log(1/\varepsilon)$? More generally, is it possible to establish that the 0-1 problem in inherently more complex than the unbounded problem?

ACKNOWLEDGMENTS

The author wishes to thank Alexander Rinnooy Kan and David Lichtenstein for reading preliminary drafts of this paper, and for their helpful comments and suggestions. The research reported in this paper was supported in part by NSF grant MCS76-17605, and in part by the Mathematisch Centrum, Amsterdam, The Netherlands, where the author was a visitor, January-February, 1977.

PROBABILISTIC ANALYSIS OF PARTITIONING ALGORITHMS FOR THE TRAVELING-SALESMAN PROBLEM IN THE PLANE

R.M. KARP

University of California, Berkeley, U.S.A.

ABSTRACT

We consider partitioning algorithms for the approximate solution of large instances of the traveling-salesman problem in the plane. These algorithms subdivide the set of cities into small groups, construct an optimum tour through each group, and then patch the subtours together to form a tour through all the cities. If the number of cities in the problem is n, and the number of cities in each group is t, then the worst-case error is $O(\sqrt{n/t})$. If the cities are randomly distributed, then the relative error is $O(t^{-1/2})$ (with probability one). Hybrid schemes are suggested, in which partitioning is used in conjunction with existing heuristic algorithms. These hybrid schemes may be expected to give near-optimum solutions to problems with thousands of cities.

CONTENTS

1. INTRODUCTION 143

2. TOURS AND SPANNING WALKS 145

3. A PARTITIONING ALGORITHM 147
Specification of Algorithm 1 147
Correctness of Algorithm 1 149
Analysis of the execution time of Algorithm 1 149
A cutting game 150
Error analysis of Algorithm 1 152

4. RANDOM TRAVELING-SALESMAN PROBLEMS IN THE PLANE 155

5. EXPECTED PERFORMANCE OF A PARTITIONING ALGORITHM 158
Specification of Algorithm 2 158
The expected execution time of Algorithm 2 159
The expected error of Algorithm 2 161

6. EXPERIMENTAL RESULTS 165

7. CONCLUSION 167

ACKNOWLEDGMENTS 168

1. INTRODUCTION

By the traveling-salesman problem in the plane we mean the problem of constructing a polygon of minimum perimeter through a given set of points (cities) in the plane. There has been considerable investigation of heuristic methods for the solution of this problem. Computer programs based on local improvement techniques [Lin & Kernighan 1973] or other heuristic principles [Krolak *et al.* 1970] appear to give near-optimal solutions to problem instances with two or three hundred cities, without using excessive amounts of computer time. Good results have also been obtained using man-machine systems, in which a person, communicating with a computer through a display terminal, controls the search for a solution [Barbosa -; Krolak *et al.* 1971; Mitchie *et al.* 1968]. Success on problems of modest size has also been achieved by persons armed with pegs to mark the cities and string to measure distances [Dantzig -].

On the other hand, at the present state of the art it is quite impossible to find, and prove that one has found, the strictly optimal solution to a large problem. The most effective exact solution methods are based on branch-and-bound techniques [Helbig Hansen & Krarup 1974; Held & Karp 1970, 1971; Smith & Thompson 1977; Smith *et al.* 1977]; they solve sixty-city problems routinely, but use excessive amounts of computer time on problems with one hundred cities. The fact that the traveling-salesman problem in the plane is NP-hard [Garey *et al.* 1976A; Papadimitriou 1977] provides convincing evidence that there does not exist a polynomial-time algorithm capable of solving the problem exactly.

Recently attention has turned to the construction of polynomial-time algorithms guaranteed to solve the problem within a specified approximation [Christofides 1978; Rosenkrantz *et al.* 1977]. The best result along these lines is due to Christofides, who has given an algorithm that runs in time $O(n^3)$, and always yields a tour less than 50% longer than the optimum tour.

The present paper takes a probabilistic approach. We assume that the cities are scattered at random in a rectangular region X of the plane. We exhibit a family of algorithms with the following property: for every $\varepsilon > 0$ there is an algorithm $A(\varepsilon)$ in the family such that

(a) $A(\varepsilon)$ runs in time $C(\varepsilon)n + O(n \log n)$;

(b) with probability 1, $A(\varepsilon)$ produces a tour costing not more than $(1+\varepsilon)$ times the cost of an optimal tour.

The algorithms are based on partitioning the region X into "small" subre-

gions, each of which contains about t cities. An optimum tour is constructed within each subregion, and these subtours are then combined to yield a tour through all the cities. Of course, standard heuristic methods may be used instead of exact solution methods to find the tours through the subregions. Such a combination of partitioning with existing heuristic methods should make it feasible to find near-optimal solutions to problems with many thousands of cities.

2. TOURS AND SPANNING WALKS

The traveling-salesman problem in the plane asks for a polygon of minimum length through a given set of points. Such a polygon corresponds to a closed tour in which each city is visited exactly once. In designing algorithms for the problem it is convenient to allow a larger set of feasible solutions, corresponding to tours which visit some cities repeatedly. This short section is devoted to showing that such a change in the problem statement makes no real difference.

Let V be a set of points in the plane. For $u \in V$ and $v \in V$, let $d(u,v)$ be the Euclidean distance between u and v. Given any multigraph (possibly with loops or multiple edges) $G = (V,E)$, with vertex set V and edge set E, define $w(G)$, the *weight* of G, as $\sum_{\{u,v\}\in E} d(u,v)$; here $d(u,v)$ is counted multiply if $\{u,v\}$ is a multiple edge. The graph $G = (V,E)$ is a *tour* if G is connected and every vertex has degree 2; G is a *spanning walk* if G is connected and all vertices are of even degree (a loop at v contributes 2 to the degree).

LEMMA 1. *Let G be a spanning walk. Then there is a tour H such that* $w(H) \le w(G)$.

Proof. We define two operations on a multigraph $G = (V,E)$ at a vertex v.

(a) If there is a loop v, then the operation LOOP(v) is applicable; it deletes the loop.

(b) If $\{u,v\} \in E$, $\{w,v\} \in E$ and the pair of arcs $\{\{u,v\},\{w,v\}\}$ is not a cut set of G, then the operation PASS(u,v,w) is applicable; it deletes the arcs $\{u,v\}$ and $\{w,v\}$, and adds the arc $\{u,w\}$.

We claim (omitting the easy proof) that

(1) the application of any operation transforms G to another spanning walk G',

(2) $w(G') \le w(G)$ (this follows from the triangle inequality $d(u,v)+d(v,w) \ge d(u,w)$), and

(3) if v is of degree > 2 in G, then some operation at v is applicable.

Repeated application of operations yields the desired tour H. □

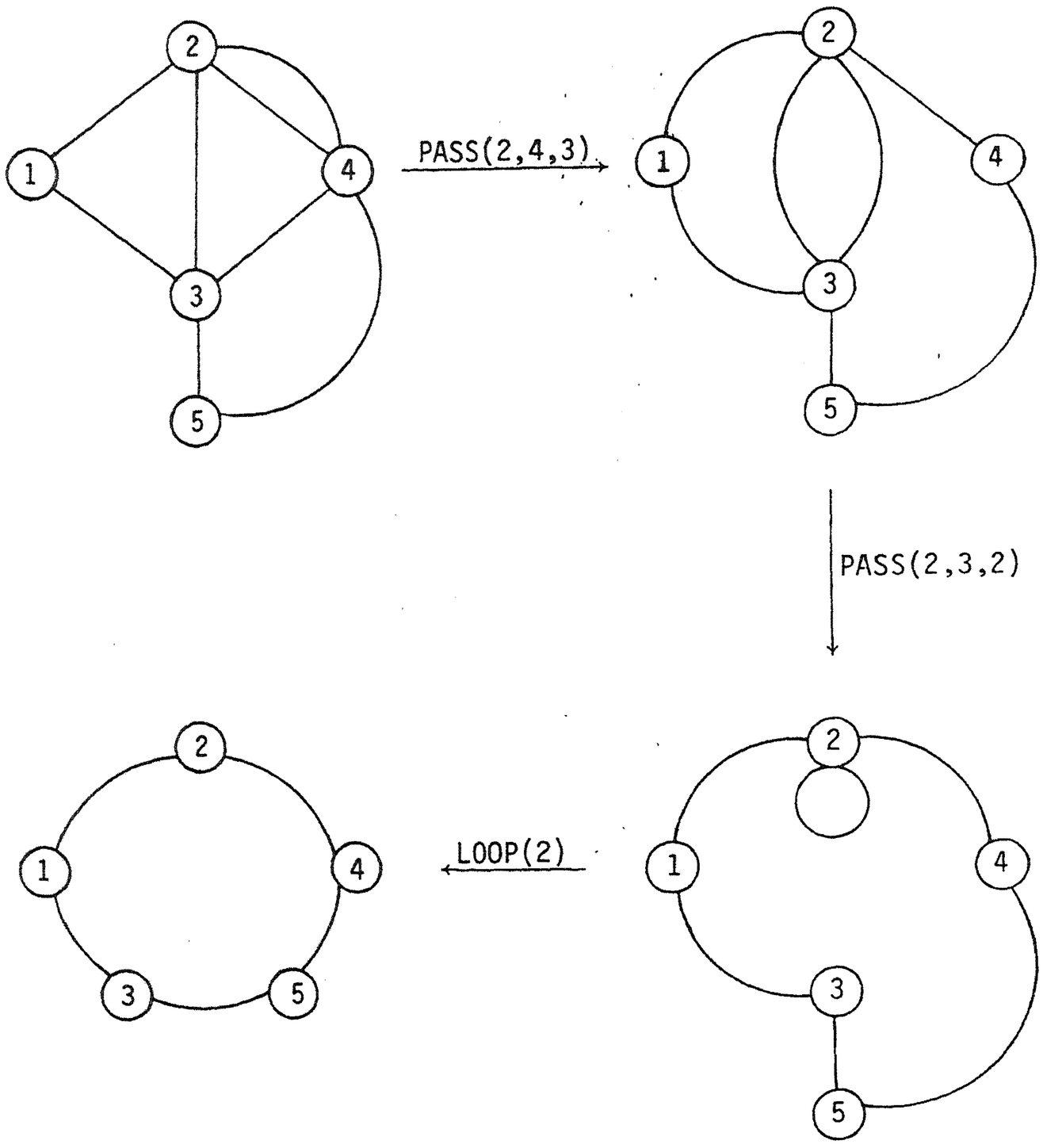


Figure 1 Transforming a spanning walk to a tour.

3. A PARTITIONING ALGORITHM

In this section we present a partitioning algorithm (called Algorithm 1) for the construction of a spanning walk through n given points (*cities*) in a rectangular region of the plane. The algorithm uses a subroutine TOUR capable of the exact solution of t-city traveling-salesman problems, where t is specified by the user of the algorithm. We show that the execution time of Algorithm 1 is O(n log n) plus the time for $\frac{n-1}{t-1}$ calls on TOUR, and that $|W_1|-|T^*| = O(\sqrt{n/t})$, where $|W_1|$ is the length of the spanning walk W_1 produced by the algorithm, and $|T^*|$ is the length of an optimum tour T^*. It follows that, if the cities are distributed at random, then, with probability 1, $|W_1|/|T^*| = 1+O(t^{-1/2})$.

Specification of Algorithm 1

Let n be the number of cities, let t be a parameter which will serve as an upper bound on the sizes of subproblems solved exactly by the subroutine TOUR, and let $k(n) = \lceil \log_2 \frac{n-1}{t-1} \rceil$; when n is clear from context we write k instead of k(n).

Algorithm 1 proceeds by subdividing the original rectangle into 2^k subrectangles, each containing at most t of the cities. Subroutine TOUR is then called to construct an optimum tour through the cities in each subrectangle. The subdivision is such that the union of the 2^k subtours forms a spanning walk through all n cities. The operations LOOP and PASS introduced in Lemma 1 may then be used to transform this walk to a tour.

We assume for convenience that no two cities are at exactly the same distance from any side of the original rectangle.

Let Y be a rectangle containing m cities. Assume Y is placed so that its longer side is horizontal. Let x be the $\lceil \frac{m}{2} \rceil$th closest city to the left edge of Y. A vertical cut through x subdivides Y into a "left rectangle" $\ell(Y)$ and a "right rectangle" r(Y), having x on their common boundary. The

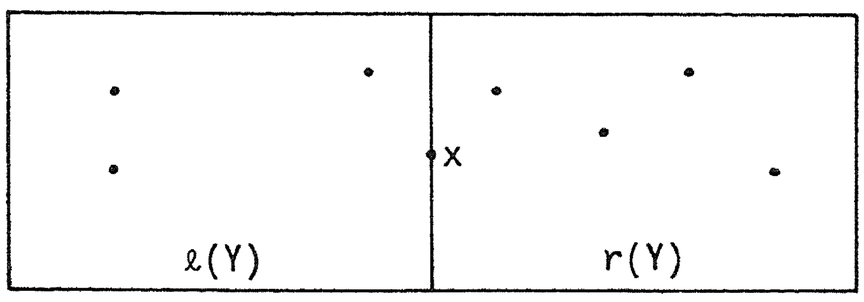


Figure 2 Partitioning a rectangle by a cut in the shorter direction.

construction is indicated in Figure 2.

Note that a spanning walk through the cities in $\ell(Y)$, plus a spanning walk through the cities in $r(Y)$, constitutes a spanning walk through the cities in Y.

Now we are ready to define our algorithm. Procedure A1 takes a rectangle X as input and produces as output a spanning walk through the cities in X. The quantity n(X) denotes the number of cities in X. In the course of the definition a recursive procedure WALK occurs. This procedure takes as inputs a rectangle Y and a nonnegative integer j. The output of WALK is a spanning walk through the cities in Y. The argument j determines the depth of the recursion used in constructing this walk. At the base of the recursion (j = 0), WALK calls on a subroutine TOUR(Y) that constructs an optimum tour through the cities in Y.

```
PROCEDURE A1
    A1(X) = WALK(X,k(n(X)))
    WALK(Y,j) = if j = 0
    then TOUR(Y)
    else WALK(ℓ(Y),j-1) ∪ WALK(r(Y),j-1)
```

Figure 3 shows the result of applying Algorithm 1 to an example with n = 25, t = 4 and k = 3. The walk is the union of 8 quadrilaterals. Figure 4 gives a tour obtained from this walk by the technique of Lemma 1.

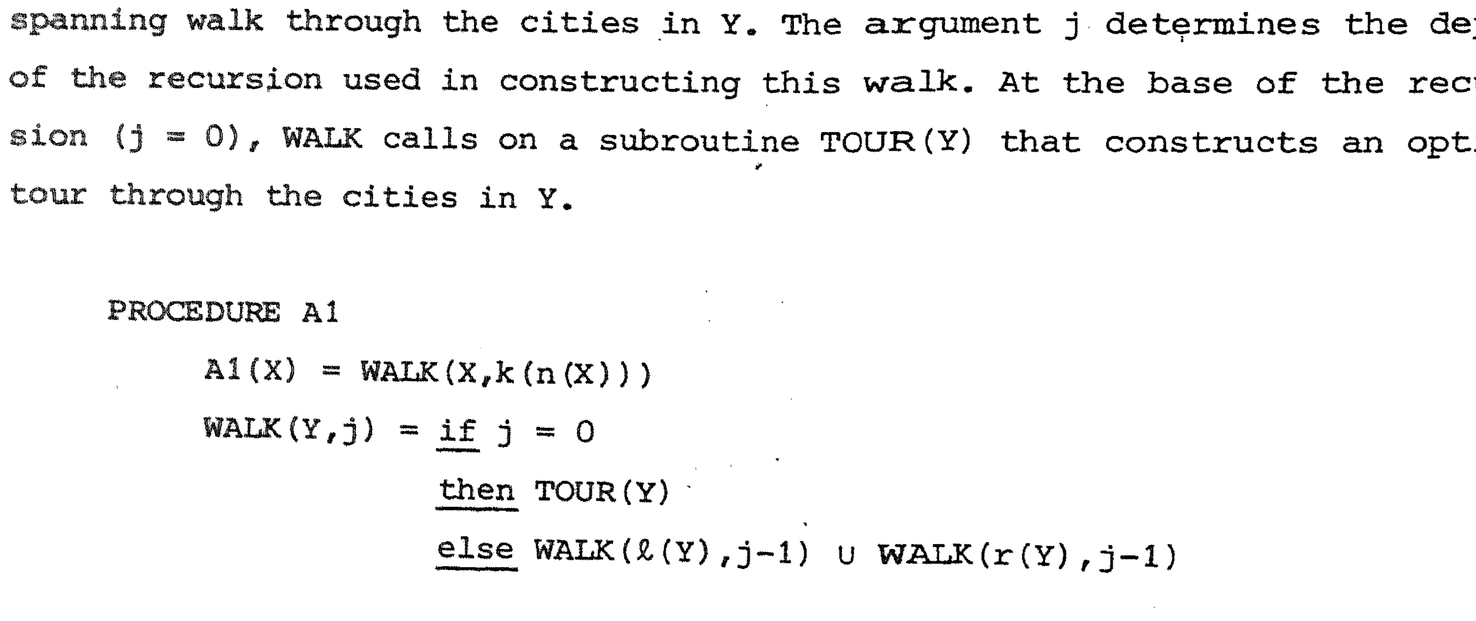


Figure 3 Walk created by Algorithm 1.

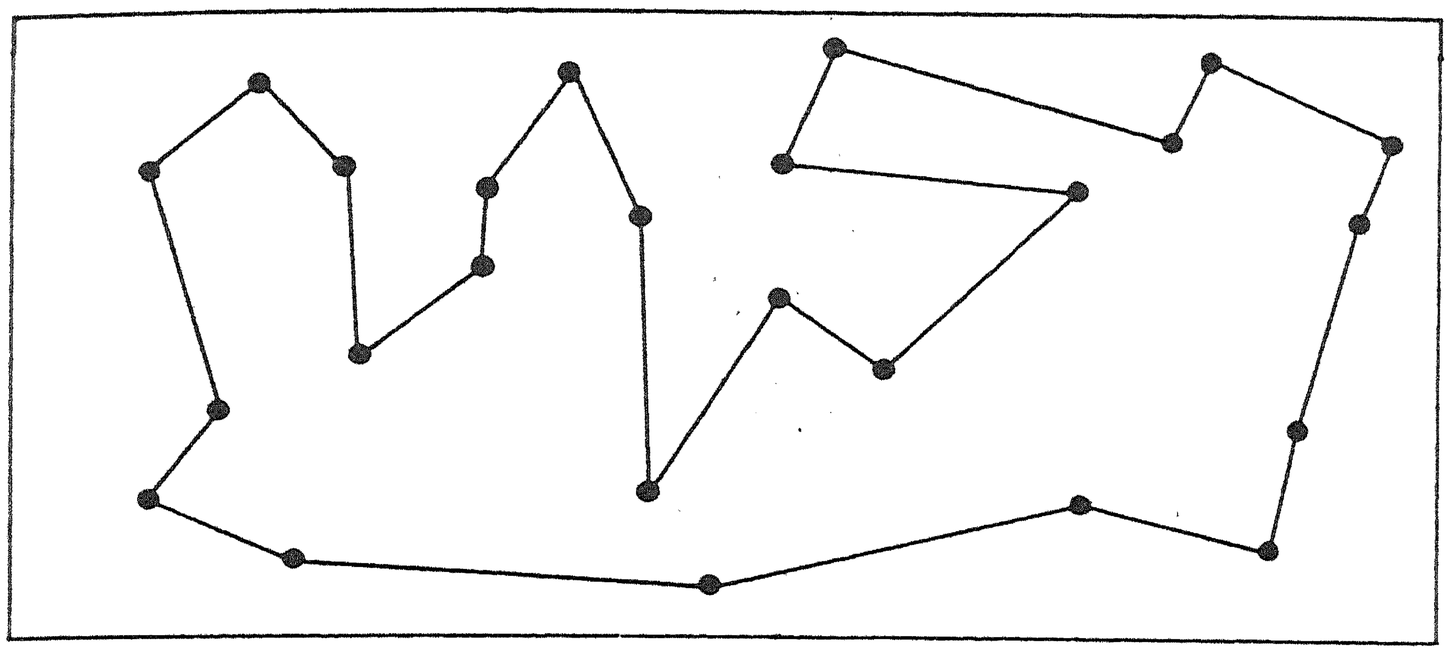


Figure 4 Tour obtained using the LOOP and PASS operations.

Correctness of Algorithm 1

LEMMA 2. *The result of applying* $A_1(X)$ *is a spanning walk through the cities in* X. *Each time* TOUR(Y) *is called,* Y *contains at most* t *cities.*

Proof. Induction on k shows that WALK(Y,j) delivers a spanning walk through the cities in X; the first statement follows. A second induction, through decreasing values of j, shows that, whenever WALK(Y,j) is called, the number of cities in Y is $\leq 2^j(t-1)+1$; since TOUR(Y) is called by WALK(Y,j) only when $j = 0$, the second result follows. □

Analysis of the execution time of Algorithm 1

In the following analysis we assume that Algorithm 1 is to be implemented on a random-access computer that requires one unit of time to compare or add real numbers (such as the x- or y-coördinates of two cities). We assume there are constants D and d such that TOUR( ) requires time $\leq Dd^t$ to solve a t-city problem. For example, if the standard dynamic programming algorithm with execution time $t^2 \cdot 2^t$ is used [Bellman 1962; Held & Karp 1962], then any value > 2 can be used for d.

THEOREM 1. *With suitable implementation, Algorithm* 1 *operates within the time bound* $2^{k(n)}Dd^t + O(n \log n) < 2\,\frac{n-1}{t-1}\,Dd^t + O(n \log n)$.

Proof. The term $2^{k(n)}Dd^t$ bounds the total time spent in executing procedure TOUR (*i.e.*, solving small traveling-salesman problems).

The remaining work is dominated by the computations of $\ell(Y)$ and $r(Y)$. Assume inductively that when we are ready to compute $\ell(Y)$ and $r(Y)$, we have available $n(Y)$, the number of cities in Y, as well as linked lists H(Y) and V(Y); H(Y) contains the cities in Y listed in left-to-right order, and V(Y) contains these cities listed in bottom-to-top order. Setting up these lists initially requires sorting the n cities on their horizontal and vertical coordinates, which can be done in O(n log n) steps. Thereafter we can process each Y in time proportional to n(Y), producing $\ell(Y)$, $r(Y)$, $n(\ell(Y))$, $H(\ell(Y))$, $V(\ell(Y))$, $n(r(Y))$, $H(r(Y))$ and $V(r(Y))$ as output. The total work for these computations is O(n log n) for the initial sorting, and O(nk) = O(n log n) for the subsequent processing. □

A cutting game

Our next objective is to derive an upper bound on the difference between the cost of the walk produced by our algorithm and the cost of an optimum tour.

In preparation for this analysis we introduce a game involving the subdivision of a rectangle X into subrectangles. There are two players, called Min and Max. The play requires k rounds. Each round consists of a move by Min, and then a move by Max. At the beginning of round ℓ, X has been subdivided into $2^{\ell-1}$ subrectangles $\{X_i\}$. During round ℓ, each of the X_i is cut in two, by either a vertical or a horizontal cut. Min's move consists of deciding, independently for each X_i, whether the cut dividing X_i will be vertical or horizontal. Max then chooses the location of the cut. At the end of the k rounds of play, Min pays Max an amount equal to the sum of the perimeters of the 2^k rectangles produced in round k.

Figure 5 shows a play of the 3-round game. Each cut is labelled with the number of the round in which it is played.

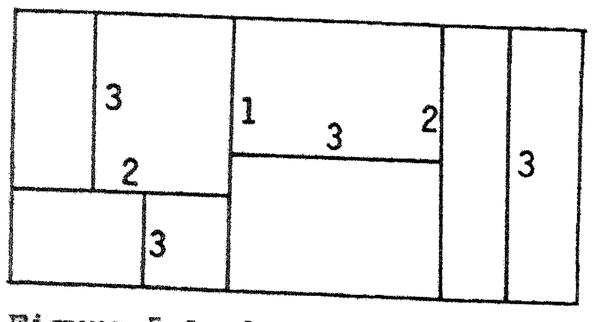


Figure 5 A play of the cutting game.

By the *short strategy* for Min we mean the policy of choosing, for each rectangle which occurs, the direction parallel to the shorter side. By the *bisection strategy* for Max we mean the policy of placing each cut so as to divide a rectangle into equal halves.

THEOREM 2. *The short strategy is optimal for* Min *and the bisection strategy is optimal for* Max.

Proof. First we show that the short strategy is best against the bisection strategy. However Min plays against the bisection strategy, the result of the play will be a subdivision of X into 2^k rectangles of equal area. If X is a×b, then each of these rectangles will be $2^{-\ell}a \times 2^{-m}b$, where ℓ and m are nonnegative integers adding to k. Since the perimeter of a rectangle of fixed area is an increasing function of the longer side, the most favorable choice of ℓ and m for Min is the one that minimizes $\text{Max}\{2^{-\ell}a, 2^{-m}b\}$. The short strategy achieves this optimal choice simultaneously for all 2^k rectangles occurring in the subdivision.

Next we show by induction on k that the bisection strategy is best for Max against the short strategy. This is certainly true for k = 1, where any strategy for Max is best against the short strategy. Assume it as an induction hypothesis for $k = \ell$. Since we know already that the short strategy is best against the bisection strategy, we can conclude that the short strategy and the bisection strategy form an optimal strategy pair for any ℓ-round game. Now consider an $(\ell+1)$-round game on an a×b rectangle, with a < b. Suppose Min, following the short strategy, specifies a cut parallel to the short side. If Max bisects we get two $a \times \frac{b}{2}$ rectangles, and optimal play (with Min using the short strategy and Max using the bisection strategy) for the ℓ ensuing rounds yields $2^{\ell+1}$ congruent rectangles of size, say, $\alpha a \times \beta\frac{b}{2}$, where $\alpha\beta = 2^{-\ell}$. The value of the game to Max is

$$2^{\ell+1}(2\alpha a + 2\beta\tfrac{b}{2}) = 2^{\ell+2}\alpha a + 2^{\ell+1}\beta b.$$

On the other hand, if Max does not bisect we get an $a \times b_1$ rectangle and an $a \times (b-b_1)$ rectangle, with $b_1 \neq \frac{b}{2}$. Suppose that, in the ensuing play, Max uses the bisection strategy (which is known to be optimal), but Min, possibly deviating from optimal play, chooses the same directions he would have chosen if Max had bisected in the first round. Then we get 2^{ℓ} $\alpha a \times \beta b_1$ rectangles, and 2^{ℓ} $\alpha a \times \beta(b-b_1)$ rectangles, for a total payoff of

$$2^{\ell}(2\alpha a + 2\beta b_1) + 2^{\ell}(2\alpha a + 2\beta(b-b_1)) = 2^{\ell+2}\alpha a + 2^{\ell+1}\beta b.$$

Thus, if Max fails to bisect in the first round, Min can achieve at least as much as he could have achieved if Max had bisected. It follows that bisection is best for Max against the short strategy in the $(\ell+1)$-round game, and the induction step is complete.

Finally, since the short strategy and the bisection strategy are best against each other, they form a saddle point, or optimal pair of pure strategies, for the cutting game. □

Let $F_k(a,b)$ denote the value (to Max) of a k-round cutting game on an $a \times b$ rectangle.

COROLLARY 1.

(*a*) $F_k(a,b) = \text{Min}_{s \text{ integer}, s+t=k}\ 2(2^t a + 2^s b)$.

(*b*) *If* a *and* b *are held fixed, then* $\sup_k F_k(a,b)/2^{k/2}$ *exists.*

Error analysis of Algorithm 1

We are now ready to apply our results about the cutting game in an error analysis of Algorithm 1. First we establish notation. Let per(Y) denote the perimeter of rectangle Y, and let |W| denote the length of the walk W (*i.e.*, the sum of the lengths of the occurrences of line segments in W). Let X be an $a \times b$ rectangle containing n cities. Let T^* denote an optimum tour through the n cities, let W_1 denote the walk produced by Algorithm 1, and let $k = k(n) = \lceil \log_2 \frac{n-1}{t-1} \rceil$.

THEOREM 3. *Let* Y *be a rectangle within* X. *Let* T(Y) *be an optimum tour through the cities in* Y. *Then* $|T(Y)| - |T^* \cap Y| \le \frac{3}{2}\,\text{per}(Y)$.

Proof. Let $T^* \cap Y$ consist of k continuous curves $C_1, C_2, \ldots, C_k$. Let the 2k end points of these curves, in clockwise order around the boundary of Y, be $Y_1, Y_2, \ldots, Y_{2k}$. Assume without loss of generality that $\overline{Y_1Y_2} + \overline{Y_3Y_4} + \cdots + \overline{Y_{2k-1}Y_{2k}} \le \overline{Y_2Y_3} + \overline{Y_4Y_5} + \cdots + \overline{Y_{2k}Y_1}$, where $\overline{Y_iY_j}$ denotes the distance from Y_i to Y_j along the perimeter of Y. Consider the walk W(Y) consisting of the following three parts:

- the curves $C_1, C_2, \ldots, C_k$,
- two copies of each of the segments $Y_1Y_2, Y_3Y_4, \ldots, Y_{2k-1}Y_{2k}$, plus

- one copy of each of the segments $y_2y_3, y_4y_5, \ldots, y_{2k}y_1$.

Then the length of the first part is $|T^* \cap Y|$, and the sum of the lengths of the second and third parts is less than or equal to $\frac{3}{2}$ the perimeter of Y. Thus $|T(Y)| \leq |W(Y)| \leq |T^* \cap Y| + \frac{3}{2}\,\mathrm{per}(Y)$. □

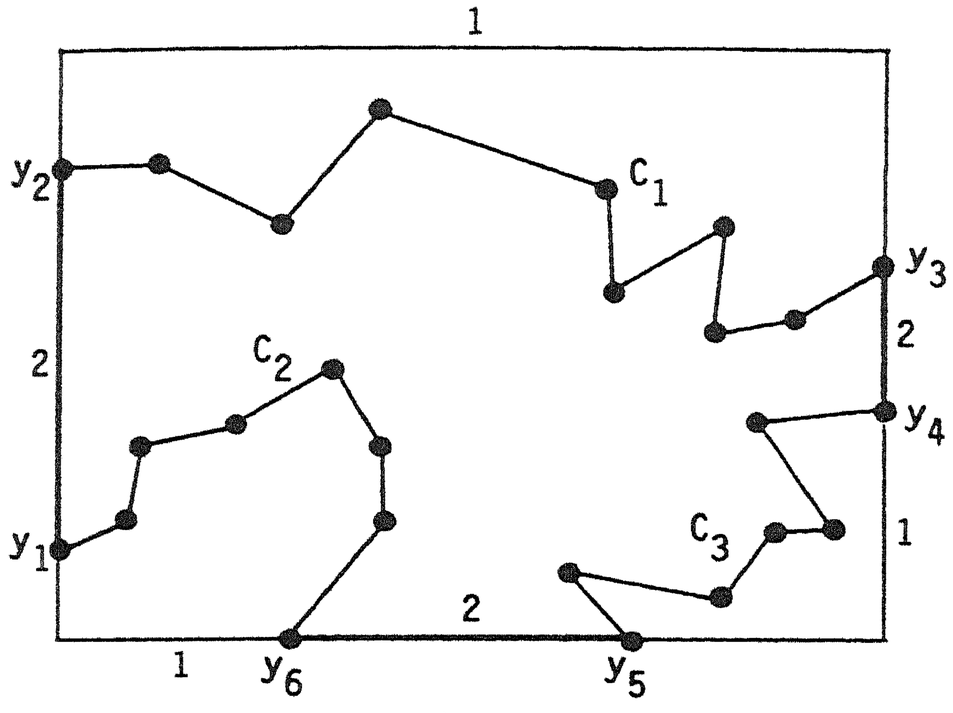
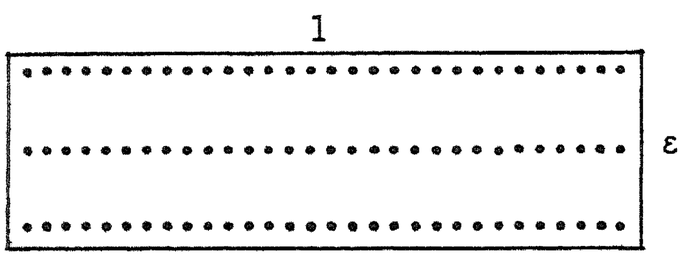
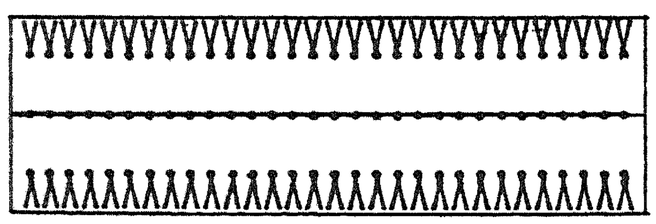


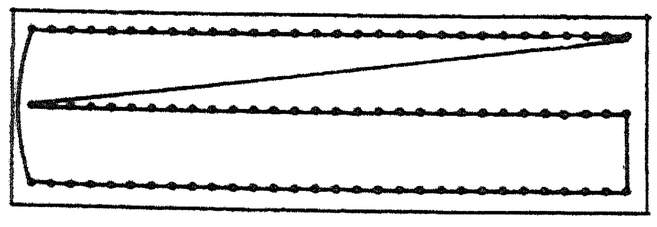
Figure 6 Converting $T^* \cap Y$ to a walk W(Y).



(a) points in a rectangle



(b) $T^* \cap Y$



(c) T(Y)

Figure 7 An adverse distribution of points.

Figure 7 indicates a family of examples for which $|T(Y)|-|T^*\cap Y|$ approaches $\frac{3}{2}$ per(Y). Let Y be an $\varepsilon\times 1$ rectangle, where ε is small, let the cities occur more and more densely on the dotted line segments, and let $T^*\cap Y$ be as indicated in Figure 7(b). The $|T^*\cap Y| \to 1$, $|T(Y)| \to 4+2\varepsilon$, and $|T(Y)|-|T^*\cap Y| \to 3+\varepsilon \to \frac{3+\varepsilon}{1+\varepsilon}\cdot\frac{1}{2}$ per(Y).

THEOREM 4. $|W_1|-|T^*| \le \frac{3}{2} F_k(a,b)$.

Proof. The execution of Algorithm 1 subdivides X into 2^k subrectangles, $\{Y_i\}$, $i = 1,2,\ldots,2^k$, and may be regarded as a play of a k-round cutting game on X. Since every cut is parallel to the short side of its rectangle, the play may regarded as one in which Min uses the short strategy, which is optimal. Thus $\sum_{i=1}^{2^k} \text{per}(Y_i) \le F_k(a,b)$. But $|W_1| = \sum_{i=1}^{2^k} |T(Y_i)|$ and, applying Theorem 3,

$$\sum_{i=1}^{2^k} |T(Y_i)| \le \sum_{i=1}^{2^k} (|T^*\cap Y_i| + \tfrac{3}{2}\,\text{per}(Y_i))$$

$$\le |T^*| + \tfrac{3}{2} \sum_{i=1}^{2^k} \text{per}(Y_i) \le |T^*| + \tfrac{3}{2} F_k(a,b). \quad \square$$

Now regard a and b as fixed and n and t as variable. Then the error bound $\frac{3}{2} F_k(a,b) \sim 2^{k/2} \sim \sqrt{n/t}$. Thus, we have

COROLLARY 2. $|W_1|-|T^*| = O(\sqrt{n/t})$.

The following construction, which we sketch informally, shows that the growth rate of our error estimate cannot be improved. Let X be the unit square, let k(n) be even and let t be a multiple of 4. Subdivide X into 2^k congruent subsquares Y_i, $i = 1,2,\ldots,2^k$, each of side $2^{-k/2}$. Then it is possible to place the cities such that Algorithm 1 produces the subdivision $\{Y_i\}$, and such that the t cities in each subsquare Y_i fall into four clusters of size t/4, with one cluster infinitesimally close to each corner of Y_i. Then

$$F_k(1,1) = 4\cdot 2^{k/2},$$

$$\sum_{i=1}^{2^k} |T(Y_i)| \sim 2^k(4\cdot 2^{-k/2}) = 4\cdot 2^{k/2},$$

$$|T^*| \sim (2^{k/2}+1)^2\cdot 2^{-k/2} \sim 2^{k/2},$$

and $$\sum_{i=1}^{2^k} |T(Y_i)|-|T^*| \sim 3\cdot 2^{k/2} = \frac{3}{4} F_k(1,1).$$

4. RANDOM TRAVELING-SALESMAN PROBLEMS IN THE PLANE

In this section we discuss a theorem from [Beardwood *et al.* 1959], showing that, if the cities are randomly distributed in a region of the plane, then the length of the shortest tour tends to grow as the square root of the number of cities. Since Algorithm 1 produces a spanning walk whose cost differs from the cost of an optimum tour by $O(\sqrt{n/t})$, it follows that the ratio of the cost of this walk to the cost of an optimum tour tends to vary with the parameter t as $1+O(t^{-1/2})$.

We model a random distribution of points in a region X of the plane by a two-dimensional Poisson distribution $\Pi_n(X)$. The distribution $\Pi_n(X)$ is determined by the following assumptions:

(1) the number of cities occurring in two or more disjoint subregions are distributed independently of each other;

(2) the expected number of cities in a region A is nv(A), where v(A) is the area of A; and

(3) as v(A) tends to zero, the probability of more than one city occurring in A tends to zero faster than v(A).

From these assumptions it follows that

(4) $\Pr\{\text{A contains exactly m cities}\} = e^{-\lambda}\frac{\lambda^m}{m!}$, where $\lambda = nv(A)$.

We study the random variable $T_n(X)$, which denotes the length of a shortest tour through the cities in X, assuming that the set of cities is distributed according to $\Pi_n(X)$.

THEOREM 5 [Beardwood *et al.* 1959]. *There exists a positive constant* β *(independent of* X*) such that* $T_n/\sqrt{nv(X)} \to \beta$ *with probability* 1.

The technical meaning of this statement is as follows. Suppose we form an infinite sequence $Z_1, Z_2, \ldots, Z_n, \ldots$ of independent samples, where Z_n is drawn from the distribution of $T_n/\sqrt{nv(X)}$. Then, for every $\varepsilon > 0$, $\Pr\{\lim|Z_n-\beta| > \varepsilon\} = 0$.

Thus, when n is sufficiently large, one can predict the value of $T_n/\sqrt{n}$ closely with a high probability of being correct. The result of Beardwood, Halton and Hammersley applies not only to rectangles, but to all Lebesgue measurable regions; replacing $\sqrt{n}$ by $n^{(d-1)/d}$, their result also applies to traveling-salesman problems in Euclidean d-space.

Combining Theorem 5 with our analysis of Algorithm 1, we can state the following result, which establishes that Algorithm 1 yields a "probabilistic

ε-approximation scheme" for the solution of the traveling-salesman problem in the plane.

THEOREM 6. *There are constants* D_1 *and* d_1 *such that, for every* $\varepsilon > 0$, *we can construct an algorithm* $A_1(\varepsilon)$ *with the following properties:*

(1) $A_1(\varepsilon)$ *runs within time* $D_1\varepsilon^{-2}d_1^{1/\varepsilon^2}n + O(n \log n)$, *and*

(2) *with probability* 1, $A(\varepsilon)$ *constructs a tour of length* $< (1+\varepsilon)$ *times the cost of an optimum tour.*

Proof. Corollary 2 tells us that $|W_1|-|T^*| < C\sqrt{n/t}$. Thus, the relative error is $< C't^{-1/2}/\beta$, with probability 1, for any $C' > C$; $A(\varepsilon)$ is simply Algorithm 1, with $t > C^2/\beta^2\varepsilon^2$. By Theorem 1, the running time is $< 2\frac{n-1}{t-1}Dd^t + O(n \log n)$. The result follows if we set $D_1 = 2\beta^2/C^2$, and $d_1 = d^{C^2/\beta^2}$. □

We shall be interested in the expected performance of a partitioning algorithm for the traveling-salesman problem in the plane.

This leads us to investigate the quantity $\beta_X(t) = E(T_t(X))/\sqrt{t}$. By a construction given in [Beardwood *et al.* 1959] there exists a constant C depending on X such that the length of a shortest tour through any n points in X is $\leq C\sqrt{n}$. From this it follows that $\beta_X(t)$ is uniformly bounded. Hence, using Theorem 5 and the dominated convergence theorem [Moran 1968,p.206], it follows that $\beta_X(t) \xrightarrow[t\to\infty]{} \beta\sqrt{v(X)}$. Here we study the rate of convergence.

Let the a×b rectangle X be fixed throughout the following discussion. Assume that dimensions are scaled so that ab = 1.

THEOREM 7. *For all* t, $\beta_X(t)-\beta \leq 6(a+b)/\sqrt{t}$.

Proof. Consider a problem instance drawn for $\Pi_{4t}(X)$. Let T^* be an optimum tour. Subdivide X into four $\frac{a}{2}\times\frac{b}{2}$ rectangles Y_1, Y_2, Y_3, Y_4. Let $T(Y_i)$ denote the length of a shortest tour through the cities in Y_i. By Theorem 3

$$|T(Y_i)| \leq |T^*\cap Y_i| + \frac{3}{2}\,\mathrm{per}(Y_i).$$

Hence,

$$\sum_{i=1}^{4}|T(Y_i)| \leq |T^*| + 6(a+b).$$

But

$$E(|T^*|) = \beta_X(4t)\sqrt{4t}$$

and

$$E(|T(Y_i)|) = \frac{1}{2}\,\beta_X(t)\sqrt{t}$$

since the set of cities in Y_i is distributed as if it were drawn from $\Pi_t(X)$, and then had all dimensions scaled down by a factor of $\frac{1}{2}$. Hence,

$$\beta_X(t) \le \beta_X(4t) + 3(a+b)/\sqrt{t}.$$

By induction on k,

$$\beta_X(t) \le \beta_X(4^k t) + 3(a+b)\,(1+\tfrac{1}{2}+\ldots+\frac{1}{2^{k-1}})/\sqrt{t}$$

$$\le \beta_X(4^k t) + 6(a+b)/\sqrt{t}.$$

Since $\beta_X(4^k t) \xrightarrow[k\to\infty]{} \beta$, the theorem follows. □

5. EXPECTED PERFORMANCE OF A PARTITIONING ALGORITHM

In this section we assume that the set of cities is drawn from $\Pi_n(X)$. We consider a variant of Algorithm 1 in which the rectangle X is partitioned into subrectangles, in each of which the expected number of cities is t. An optimum tour is constructed in each subrectangle, and these tours are then patched together to form a walk W_2 through all the cities. Our main result is that $E(|W_2|)/\sqrt{n} \leq \beta_X(t) + O(t^{-1/6})$.

Specification of Algorithm 2

Throughout the following discussion X is a fixed rectangle of area 1, and t is a fixed positive real number. Choose $k = k(n)$ as the least positive integer such that $t \cdot 2^{k(n)} \geq n$. Given any rectangle Y, define $\ell'(Y)$ and $r'(Y)$ as the two subrectangles determined by a bisecting cut parallel to the short sides of Y. Define e(Y) as the shortest line segment joining a city in $\ell'(Y)$ with a city in $r'(Y)$; if either $\ell'(Y)$ or $r'(Y)$ contains no city, then e(Y) is undefined.

In the following procedure definition, procedure A2 takes a rectangle as input and produces as output a spanning walk through the cities in X. The recursive procedure WALK2(Y,j) accepts as input a rectangle Y and a positive integer j, and produces a spanning walk through the cities in Y. The input j controls the depth of recursion. The procedure TOUR(Y) is a subroutine that constructs an optimum tour through the cities in Y.

```
PROCEDURE A2
    A2(X) = WALK2(X,k)
    WALK2(Y,j) = if j = 0
                 then TOUR(Y)
                 else WALK2(ℓ'(Y),j-1) ∪ WALK2(r'(Y),j-1) ∪ 2e(Y)
```

Alternately, Algorithm 2 may be viewed as follows. The rectangle X is subdivided into 2^k subrectangles according to a play of the cutting game in which the two players use the short strategy and the bisection strategy, respectively. The spanning walk W_2 consists of shortest tours through these subrectangles, together with additional line segments linking these tours together into a connected structure; each of these connecting segments occurs twice.

Figure 8 shows the result of applying Algorithm 2 to the example of Figure 3. The parameters are $n = 25$, $t = \frac{25}{8}$, $k = 3$. Each cross-hatched segment in Figure 8 occurs twice in W_2.

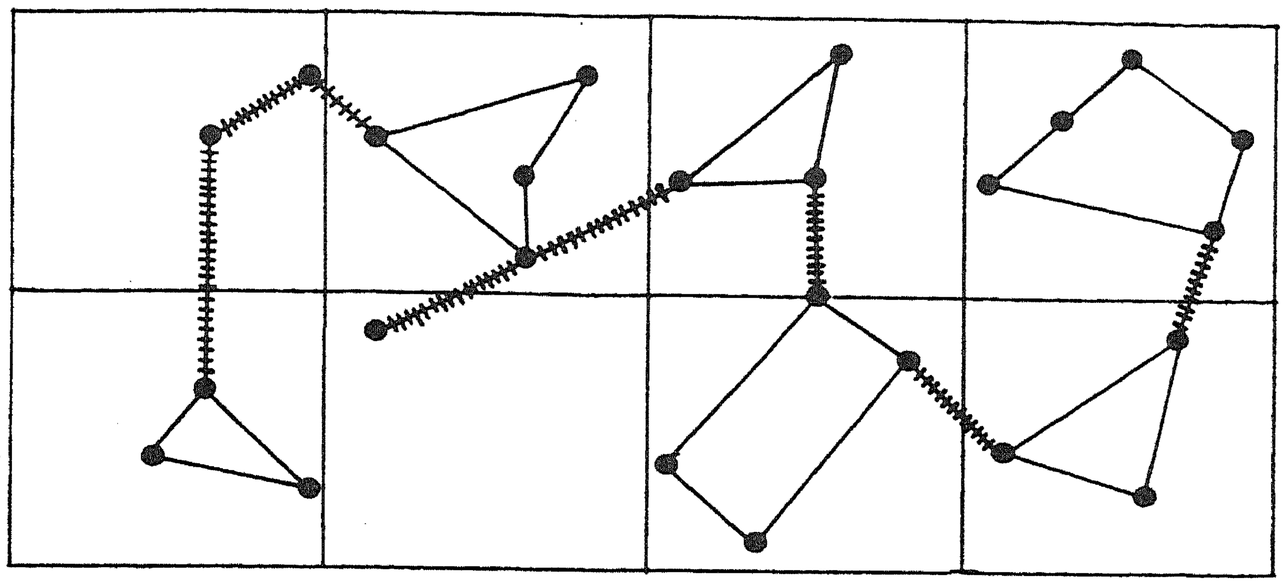


Figure 8 Example of the construction of W_2.

LEMMA 3. *The result of applying Algorithm* 2 *is an Eulerian walk through the cities in* X. *Each time* TOUR(Y) *is called, the expected number of cities in* Y *is* $n/2^k \leq t$.

The expected execution time of Algorithm 2

In analyzing the performance of Algorithm 2, we make the following assumption about the subroutine TOUR.

ASSUMPTION. *Let* E(s,Y) *denote the expected time for* TOUR *to compute a shortest tour through* s *points randomly distributed in the rectangle* Y. *Then there are absolute constants* c *and* C *such that, for all* s *and* Y, $E(s,Y) \leq Cc^s$.

If TOUR is the standard dynamic programming algorithm [Bellman 1962; Held & Karp 1962] with execution time $O(s^2 \cdot 2^s)$ then any constant $c > 2$ will work. Certain branch-and-bound methods [Helbig Hansen & Krarup 1974; Held & Karp 1970, 1971; Smith & Thompson 1977; Smith *et al.* 1977] seem to achieve substantially smaller values of c, but no rigorous analyses exist.

THEOREM 8. *Let* c *and* C *be as in the Assumption. Then, with suitable implementation, the expected execution time of Algorithm* 2 *on problems drawn from* $\Pi_n(X)$ *is less than or equal to* $\frac{2n}{t} Ce^{(c-1)t} + O(n \log^2 n)$.

Proof. The execution time is the sum of three contributions:

(1) the time spent solving "small" traveling-salesman problems using the subroutine TOUR;

(2) the time spent determining the line segments e(Y); and

(3) the time spent on other operations.

Contribution (3) can be bounded by O(n log n), exactly as in the analysis of Algorithm 1.

We estimate contribution (1) as follows. The subroutine TOUR is invoked $2^{k(n)}$ times. Each time the number of cities has a Poisson distribution with mean t. Thus the expected execution time of each invocation is $\leq C \sum_{x=0}^{\infty} e^{-t} \frac{t^x}{x!} c^x = Ce^{(c-1)t}$. The expected total time spent in TOUR is thus $\leq 2^{k(n)} Ce^{(c-1)t} < \frac{2n}{t} Ce^{(c-1)t}$.

Finally, we show that contribution (2) is $O(n \log^2 n)$. As a first step, we show that the time to compute e(Y), the shortest segment joining a city in $\ell'(Y)$ with a city in r'(Y), is O(n(Y) log n(Y)). Each candidate for e(Y) must cross B, the cut separating $\ell'(Y)$ from r'(Y). For each city $a \in \ell'(Y)$, define the interval I(a) by

$$I(a) = \{x \in B \mid a \text{ is the closest city in } \ell'(Y) \text{ to } x\};$$

similarly, for any city $b \in r'(Y)$,

$$J(b) = \{x \in B \mid b \text{ is the closest city in } r'(Y) \text{ to } x\}.$$

Using techniques from [Shamos 1975; Shamos & Hoey 1975] these intervals can be determined in O(n(Y) log n(Y)) steps. Then, in linear time, one can list all pairs a,b such that $I(a) \cap J(b)$ has positive measure. There are at most n(Y) such pairs, and they are the only candidates for the segment e(Y). Thus e(Y) can be determined in O(n(Y) log n(Y)) steps. The expected time spent in computing the segments e(Y) thus grows as

$$E(\sum_{\{Y \mid Y \text{ is subdivided}\}} n(Y) \log n(Y)).$$

Application of Chebyshev's inequality yields the result that, if n(Y) is Poisson distributed with mean λ, then

$$E(n(Y) \log_2 n(Y)) \leq \lambda \log_2 \lambda + \frac{4}{3}.$$

Hence,

$$E(\sum_{\{Y \mid Y \text{ is subdivided}\}} n(Y) \log n(Y)) \leq \sum_{j=0}^{k-1} 2^j (\frac{n}{2^j} \log \frac{n}{2^j} + \frac{4}{3})$$

$$< kn \log n + \frac{4}{3}(2^k - 1)$$

$$= O(n \log^2 n).$$

This completes the proof. □

The expected error of Algorithm 2

We continue to assume that X is a fixed a×b rectangle and the parameter t is fixed. Assuming that problem instances are drawn from $\Pi_n(X)$, we analyze the expected value of $|W_2| - |T^*|$ as a function of n. Here W_2 denotes the spanning walk generated by Algorithm 2, and T^* denotes a fixed tour.

To avoid purely technical complications we assume that $a \le b < 2a$, $n = 2^{k(n)} \cdot t$, and k(n) is even.

THEOREM 9. $E(|W_2|) = \sqrt{n}(\beta_X(t) + O(t^{-7/6}))$.

Proof. In estimating $|W_2|$ we note that, because $a \le b < 2a$, the k-stage subdivision process alternates between stages of vertical cutting and stages of horizontal cutting. Because k is even, the 2^k resulting rectangles are similar to X, but with both their dimensions scaled down by the factor $2^{-k/2}$.

The length $|W_2|$ is the sum of two contributions:

(1) the sum of the lengths of the shortest tours within the rectangles, and

(2) twice the sum of the lengths of the arcs e(Y).

The first contribution may be estimated as follows. The distribution of cities in any one of the 2^k rectangles is the same as if the cities had been drawn from $\Pi_t(X)$, and then all directions had been scaled down by the factor $2^{-k/2}$. Thus the expected value of the first contribution is

$$2^k(\beta_X(t)\sqrt{t})2^{-k/2} = \beta_X(t)\sqrt{n}.$$

We estimate the second contribution as follows. By Lemma 5, the expected length of e(Y) is $n^{-2/3}\ell^{-1/3} + o(n^{-2/3}\ell^{-1/3})$, where ℓ is the length of the shorter side of Y (the fact that $\ell'(Y)$ or $r'(Y)$ may fail to contain a city, in which case we take $|e(Y)| = 0$, only helps us). Summing over all the rectangles Y that get subdivided we have

$$\sum_{j=0}^{\frac{1}{2}k-1} n^{-2/3}(a\cdot 2^{-j})^{-1/3}\cdot 2^{2j} + \sum_{j=0}^{\frac{1}{2}k-1} n^{-2/3}(b\cdot 2^{-(j+1)})^{-1/3}\cdot 2^{2j+1}$$

$$= n^{-2/3}a^{-1/3}\sum_{j=0}^{\frac{1}{2}k-1} 2^{7j/3} + 2n^{-2/3}b^{-1/3}\sum_{j=0}^{\frac{1}{2}k-1} 2^{7(j+1)/3}$$

$$= O(n^{-2/3}2^{7k/6}) = O(n^{-2/3}(\frac{n}{t})^{7/6}) = O(n^{1/2}t^{-7/6}). \square$$

It only remains to give the Lemma used in the proof of Theorem 9. This requires a preliminary Lemma.

LEMMA 4. *Let* h *points be placed at random on a unit interval. Then the expected value of the minimum distance between a pair of these points is* $\leq \frac{2}{(h-2)(h-1)}$.

Proof. Let A be a constant. We derive an upper bound on Q_A, the probability that the minimum distance is $\geq$ A. Regard the points as being placed successively at random locations on the unit interval. Assuming that no two of the first k points are within A of each other, the probability that the (k+1)st point is within A of some previously placed point is $\geq 2\frac{A}{2} + (k-2)A = (k-1)A$. Hence

$$Q_A \leq \prod_{k=2}^{h-1}(1-(k-1)A) \leq \prod_{k=2}^{h-1} e^{-A(k-1)} = e^{-\frac{A}{2}(h-2)(h-1)}.$$

The expected value of the shortest distance is

$$\int_{A=0}^{\infty} Q_A dA \leq \int_{A=0}^{\infty} e^{-\frac{A}{2}(h-2)(h-1)} dA = \frac{2}{(h-2)(h-1)}. \square$$

LEMMA 5. *Suppose cities are distributed according to a 2-dimensional Poisson distribution with density* n *in the infinite strip between the two parallel lines* y = 0 *and* y = ℓ. *The expected value of the minimum distance between a city in the right half-plane and a city in the left half-plane is* $\leq 2n^{-2/3}\ell^{-1/3} + o(n^{-2/3}\ell^{-1/3})$.

Proof. Let h be a positive integer to be specified later. Order the cities in increasing order of their distance from the line x = 0; call this total ordering "$\prec$". Select an increasing sequence of cities $a_1, a_2, \ldots, a_h$ as follows. For a_1, we select the first city in the ordering. Given $a_1, \ldots, a_u$, $a_{u+1} = \min\{a \mid a_u \prec a \text{ and } P_u(a) \text{ holds}\}$, where $P_u(a)$ is a property which holds if the point in $\{a_1, a_2, \ldots, a_u\}$ at the least vertical displacement from a is in the opposite half-plane from a. Among the points $\{a_1, \ldots, a_h\}$, let a_{i_1} and

a_{i_2} be the two at minimum vertical distance from each other. By the way the points were selected, a_{i_1} and a_{i_2} are in opposite half-planes. We compute the expected value of the distance between a_{i_1} and a_{i_2}.

Let a_i have the coordinates (x_i,y_i). Then $|x_1|$ has an exponential distribution with mean $\frac{1}{2n\ell}$, and $|x_{i+1}|-|x_i|$ is exponential with mean $\frac{1}{n\ell}$. Thus $E(|x_i|) = \frac{i-\frac{1}{2}}{n\ell}$. Since a_{i_1} is equally likely to be any of the a_i, $E(|x_{i_1}|) = \frac{1}{h}\sum_{i=1}^{h} \frac{i-\frac{1}{2}}{n\ell} = \frac{h}{2n\ell}$. Similarly, $E(|x_{i_2}|) = \frac{h}{2n\ell}$.

The random variables $\{a_1,\ldots,a_h\}$ are highly dependent, but their vertical coordinates $\{y_u\}$ are independent and uniformly distributed over $[0,\ell]$. For, given $a_1,a_2,\ldots,a_u$, note that a_{u+1} is the city closest to the y-axis in a region R consisting of horizontal strips in both the right half-plane and the left half-plane, such that $R\cap\{(x,y)|y = y_0\}$ is

$$\text{either} \quad \{(x,y_0)|x > a_u\}$$
$$\text{or} \quad \{(x,y_0)|x < -a_u\}.$$

Hence, every y_0 is equally likely to be the y-coordinate of the next city chosen. By Lemma 4, $E(|y_{i_2}-y_{i_1}|) \le \frac{2\ell}{(h-2)(h-1)}$. Hence, the expected value of the distance between a_{i_1} and a_{i_2} is $\le \frac{2\ell}{(h-2)(h-1)} + \frac{h}{n\ell}$. Setting $h = [n^{1/3}\ell^{2/3}]$, the result follows. □

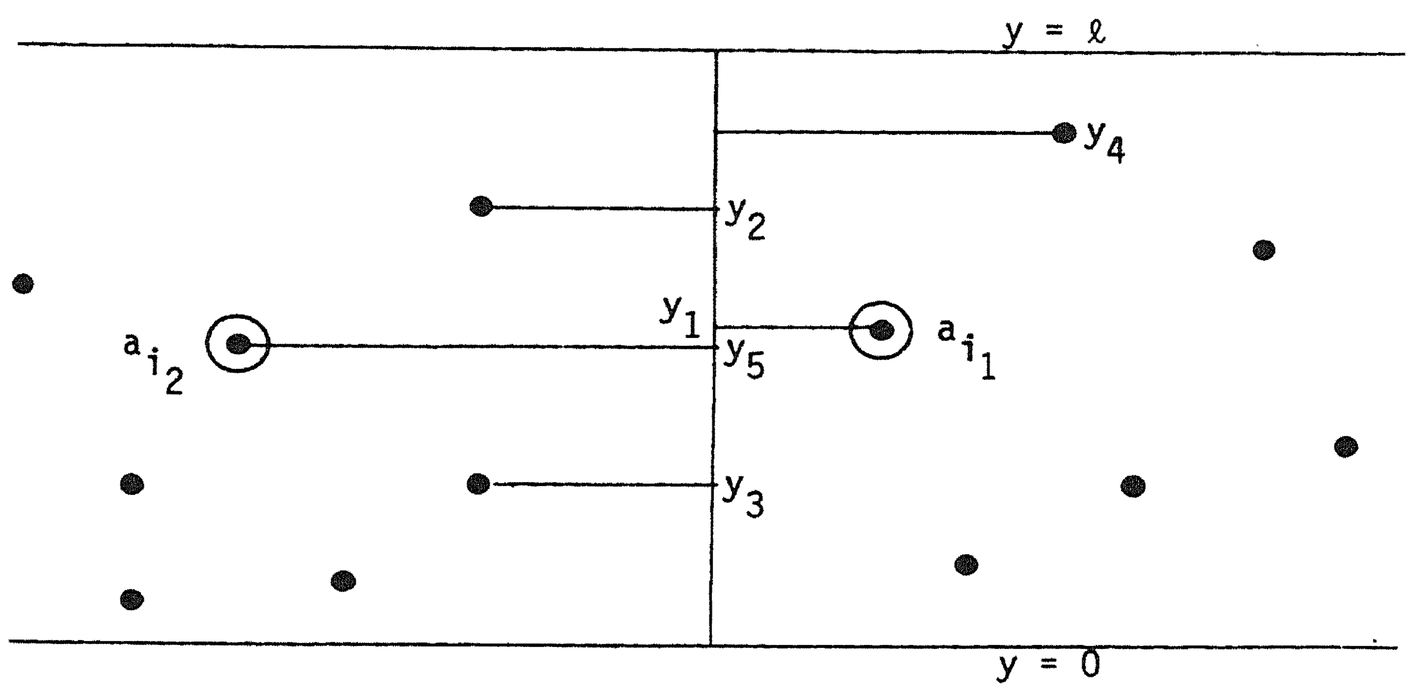


Figure 9 Construction in the proof of Lemma 5 (h = 5).

It is natural to conjecture that $\beta_X(t) \ge \beta$ for all t. We cannot prove this, but Theorem 9 does yield the following corollary.

COROLLARY 3. $\beta_X(t) + O(t^{-7/6}) \ge \beta$.

Proof. Theorem 9 shows that, for infinitely many n, $\beta_X(t) + O(t^{-7/6})$ $\geq \beta_X(n)$. Since $\beta_X(n) \to \beta$, the result follows. □

COROLLARY 4. *The expected value of* $|W_2|-|T^*| \leq \sqrt{n}(\beta_X(t) - \beta + O(t^{-7/6}))$.

Proof. By Theorem 9, $E(|W_2|) = \sqrt{n}(\beta_X(t) + O(t^{-7/6}))$. By definition, $E(|T^*|) = \sqrt{n}\beta_X(n)$. By Corollary 3, $\beta_X(n) + O(n^{-7/6}) \geq \beta$. Combining these results,

$$E(|W_2|-|T^*|) \leq \sqrt{n}(\beta_X(t) - \beta_X + O(t^{-7/6}) + O(n^{-7/6})).$$

Since $t < n$, this simplifies to $\sqrt{n}(\beta_X(t) - \beta + O(t^{-7/6}))$. □

COROLLARY 5. *For every* $\alpha > 1$, *the ratio* $|W_2|/|T^*|$ *is* $\leq 1 + \alpha(\beta_X(t) + O(t^{-7/6}))/\beta$, *with probability* 1.

Since $\beta_X(t)-\beta = O(t^{-1/2})$, the expected relative error for Algorithm 2 is $O(t^{-1/2^X})$. So far as growth rate is concerned, this is no improvement over Algorithm 1. The advantage of Algorithm 2, and our reason for presenting it in detail, is that the expected error is given explicitly in terms of $\beta_X(t)$. Thus, any information gained about $\beta_X(t)$, with respect to either its growth rate or its values for specific t, will be directly applicable.

6. EXPERIMENTAL RESULTS

To determine experimentally the quality of solutions produced by Algorithm 1 or Algorithm 2 would have required a supply of randomly generated problems for which optimal (or nearly optimal) solutions are known, as well as considerable investment in computer programming and data preparation. We decided instead to test the effectiveness of partitioning schemes using the minimum spanning tree problem in the plane as a substitute for the traveling-salesman problem. This permitted the experiments to be done by hand.

The problem of constructing a minimum-length spanning tree through a set of n points in the plane can be solved in O(n log n) steps [Shamos 1975; Shamos & Hoey 1975]. To test our partitioning ideas, we ignored the existence of this efficient exact solution algorithm, and instead used the following partitioning scheme, which combines features of our two partitioning algorithms for the traveling-salesman problem.

Let Y be a rectangle, oriented so that its longer sides are horizontal, and containing m cities. Then Y can be subdivided into two rectangles, $\ell^*(Y)$ and $r^*(Y)$, by a vertical cut equidistant between the $\lfloor \frac{m}{2} \rfloor$th city from the left and the $(\lfloor \frac{m}{2} \rfloor + 1)$st city from the left. Let $e^*(Y)$ denote the shortest segment joining a city in $\ell^*(Y)$ with a city in $r^*(Y)$. Let TREE(Y) be a subroutine capable of finding a minimum spanning tree through all the cities in Y; this subroutine will be called only when Y contains t or fewer cities. Let k(n) be the least integer such that $2^{k(n)} \geq \frac{n}{t}$.

The following is a recursive presentation of the partitioning scheme, similar to our earlier presentations of procedures A1 and A2.

```
PROCEDURE A3
     A3(X) = APPROXTREE(X,k)
     APPROXTREE(Y,j) = if j = 0
                       then TREE(Y)
                       else APPROXTREE(ℓ*(Y),j-1)
                            ∪ APPROXTREE(r*(Y),j-1) ∪ e*(Y)
```

Applying ideas quite similar to those that occur in the analysis of Algorithm 1, we can show that the difference between $|T_{approx}|$, the length of the spanning tree produced by $A_3(X)$, and $|T_{opt}|$, the length of the minimum spanning tree, is less than

$$(\sum_Y \text{per}(Y))-\text{per}(X),$$

where the summation is over all rectangles Y to which the procedure TREE is applied.

Five 128-city problems were generated. In each, the cities were randomly distributed in the unit square. Each problem was solved with t = 4, 8, 16, 32 and 64. Table 1 reports the observed percentage error (defined as $100(|T_{approx}|/|T_{opt}|-1)$) for each run of Algorithm 3.

TABLE 1. PERCENTAGE ERROR EXPERIENCED BY ALGORITHM 3

problem	1	2	3	4	5
t 64	.6	3.2	3.0	.5	1.6
32	2.4	4.7	4.3	1.7	4.9
16	5.3	10.7	7.0	3.7	9.7
8	7.9	14.1	8.0	6.2	12.9
4	10.5	16.3	10.6	10.4	16.8

A good empirical formula for the error is

$$|T_{approx}|-|T_{min}| \approx .13((\sum_Y \text{per}(Y))-\text{per}(X)).$$

Thus, the error is typically proportional to $(\sum_Y \text{per}(Y))-\text{per}(X)$, but with a much smaller constant of proportionality than the one arising from a worst-case analysis.

We speculate that a similar relationship exists between the average and worst-case error of our traveling-salesman algorithms. In particular, we conjecture that $\beta_X(t)-\beta$ is proportional to $t^{-1/2}$, but with a much smaller constant of proportionality than the one given in the upper bound of Theorem 7.

7. CONCLUSION

We believe that the partitioning schemes presented here have both theoretical and practical interest. Using Algorithm 1 we have available a "probabilistic ε-approximation scheme" for the traveling-salesman problem in the plane. That is, for every $\varepsilon > 0$, we can construct an algorithm $A_1(\varepsilon)$ that runs within time $D_1\varepsilon^2 d_1^{1/\varepsilon^2} n + O(n \log n)$ and, with probability 1, constructs a tour of length $< (1+\varepsilon)$ times the cost of an optimum tour. This is done simply by running Algorithm 1, with t depending suitably on ε. A similar scheme can be constructed using Algorithm 2. The choice of t necessary to achieve a specified relative error in this case will depend on the rate at which $\beta_X(t)$ converges to β; our present estimates of this convergence rate (Theorem 7) are undoubtedly far too pessimistic. We conjecture that $\beta_X(t)-\beta$ is indeed proportional to $t^{-1/2}$, but with a much smaller constant of proportionality than is given by our upper estimate. Our experimental results with a partitioning algorithm for the minimum spanning tree problem lend support to this conjecture.

In practice Algorithm 1 is probably to be preferred over Algorithm 2, since its absolute error bound is independent of any assumptions about the distribution of the cities.

Some of the technical results used in analyzing the algorithms may be of independent interest. Among these results are the characterization of optimal strategies for the cutting game, and the lemma bounding the expected shortest distance between cities in adjacent rectangles.

Our partitioning schemes should prove to be of practical use in the solution of extremely large traveling-salesman problems in the plane. The heuristic methods from [Krolak *et al.* 1970; Lin & Kernighan 1973] yield good approximate solutions in reasonable time to problems with two or three hundred cities, but they become unwieldy for larger problems. When the number of cities is in the thousands one can use a hybrid scheme with combines partitioning with one of these heuristics. In such a scheme the exact solution procedure TOUR(Y) in Algorithm 2 would be replaced by a heuristic method. It would then be feasible to run the algorithm with t = 200 (say), and the expected relative error would be

$$\Delta(t) + \beta_X(t)/\beta + O(t^{-7/6})$$

where $\Delta(t)$ is the expected relative error for the underlying heuristic

algorithm.

All the results generalize easily if we allow the locations of the cities to be determined according to an arbitrary 2-dimensional probability density function or if we let the domain X be any connected Lebesgue measurable set. Also, we may use the rectilinear or L_∞ metric instead of the Euclidean metric. The algorithms may also be generalized to d dimensions; the worst-case error in Algorithm 1 becomes $O((\frac{n}{t})^{(d-1)/d})$, and the relative error (with probability 1) is $O(t^{-(d-1)/d})$.

Finally, the partitioning methods and their analyses may be modified in a straightforward way to apply to many other optimization problems of a geometric nature. Among these are the Steiner tree problem in the plane, the problem of constructing a minimum-weight perfect matching when the weights are Euclidean distances, the simple plant location problem in the plane, and various multi-vehicle delivery problems in the plane.

ACKNOWLEDGMENTS

This research was supported by NSF grant MCS74-17680-A02. This paper has been reprinted from *Mathematics of Operations Research*, Vol. 2, No. 3, August 1977, pp. 209-224.

OPTIMIZATION AND APPROXIMATION IN DETERMINISTIC SEQUENCING AND SCHEDULING: A SURVEY

R.L. GRAHAM

Bell Laboratories, Murray Hill, N.J., U.S.A.

E.L. LAWLER

University of California, Berkeley, U.S.A.

J.K. LENSTRA

Mathematisch Centrum, Amsterdam, The Netherlands

A.H.G. RINNOOY KAN

Erasmus University, Rotterdam, The Netherlands

ABSTRACT

The theory of deterministic sequencing and scheduling has expanded rapidly during the past years. In this paper we survey the state of the art with respect to optimization and approximation algorithms and interpret these in terms of computational complexity theory. Special cases considered are single machine scheduling, identical, uniform and unrelated parallel machine scheduling, and open shop, flow shop and job shop scheduling. We indicate some problems for future research and include a selective bibliography.

CONTENTS

1. INTRODUCTION 172

2. PROBLEM CLASSIFICATION 173
2.1. Introduction 173
2.2. Job data 173
2.3. Machine environment 173
2.4. Job characteristics 174
2.5. Optimality criteria 175
2.6. Examples 175
2.7. Reducibility among scheduling problems 177

3. SINGLE MACHINE PROBLEMS 178
3.1. Introduction 178
3.2. Minimizing maximum cost 178
3.3. Minimizing total cost 179
3.3.1. $1|\beta|\sum w_j C_j$ 179
3.3.2. $1|\beta|\sum w_j T_j$ 179
3.3.3. $1|\beta|\sum w_j U_j$ 180

4. PARALLEL MACHINE PROBLEMS 182
4.1. Introduction 182
4.2. Nonpreemptive scheduling: unit processing times 182
4.2.1. $Q|p_j=1|\sum f_j$, $Q|p_j=1|f_{max}$ 182
4.2.2. $P|prec,p_j=1|C_{max}$ 183
4.2.3. $P|res,\beta,p_j=1|C_{max}$ 185
4.3. Nonpreemptive scheduling: general processing times 188
4.3.1. $P||\sum w_j C_j$ 188
4.3.2. $Q||\sum C_j$ 189
4.3.3. $R||\sum C_j$ 189
4.3.4. *Other cases: enumerative optimization methods* 190
4.3.5. *Other cases: approximation algorithms* 191
4.3.5.1. $P||C_{max}$ 191
4.3.5.2. $Q||C_{max}$ 193
4.3.5.3. $R||C_{max}$ 194
4.3.5.4. $P|prec|C_{max}$ 195
4.3.5.5. $Q|prec|C_{max}$ 196

4.3.5.6. P|*res,prec*|C_{max} 197
4.4. Preemptive scheduling 198
4.4.1. P|*pmtn*|$\sum C_j$ 198
4.4.2. Q|*pmtn*|$\sum C_j$ 198
4.4.3. R|*pmtn*|$\sum C_j$ 198
4.4.4. P|*pmtn,prec*|C_{max} 199
4.4.5. Q|*pmtn,prec*|C_{max} 200
4.4.6. R|*pmtn*|C_{max} 200
4.4.7. P|*pmtn*,r_j|L_{max} 201
4.4.8. Q|*pmtn*,r_j|L_{max} 201

5. OPEN SHOP, FLOW SHOP AND JOB SHOP PROBLEMS 203
5.1. Introduction 203
5.2. Open shop scheduling 203
5.2.1. *Nonpreemptive case* 203
5.2.2. *Preemptive case* 205
5.3. Flow shop scheduling 206
5.3.1. F2|β|C_{max}, F3|β|C_{max} 206
5.3.2. F||C_{max} 207
5.3.3. *No wait in process* 209
5.4. Job shop scheduling 210
5.4.1. J2|β|C_{max}, J3|β|C_{max} 210
5.4.2. J||C_{max} 210

6. CONCLUDING REMARKS 214

ACKNOWLEDGMENTS 214

1. INTRODUCTION

In this paper we attempt to survey the rapidly expanding area of deterministic scheduling theory. Although the field only dates back to the early fifties, an impressive amount of literature has been created and the remaining open problems are currently under heavy attack. An exhaustive discussion of all available material would be impossible - we will have to restrict ourselves to the most significant results, omitting detailed theorems and proofs. For further information the reader is referred to the classic book by Conway, Maxwell and Miller [Conway *et al.* 1967], the more recent introductory textbook by Baker [Baker 1974], the advanced expository articles collected by Coffman [Coffman 1976] and a few survey papers and theses [Bakshi & Arora 1969; Lenstra 1977; Liu 1976; Rinnooy Kan 1976].

The outline of the paper is as follows. Section 2 introduces the essential notation and presents a detailed problem classification. Sections 3, 4 and 5 deal with single machine, parallel machine, and open shop, flow shop and job shop problems, respectively. In each section we briefly outline the relevant complexity results and optimization and approximation algorithms. Section 6 contains some concluding remarks.

We shall be making extensive use of concepts from the theory of computational complexity [Karp 1972, 1975A]. An introductory survey of this area appears elsewhere in this volume [Lenstra & Rinnooy Kan 1978B] and hence terms like (*pseudo*)*polynomial-time algorithm* and (*binary* and *unary*) *NP-hardness* will be used without further explanation.

2. PROBLEM CLASSIFICATION

2.1. Introduction

Suppose that n *jobs* J_j (j = 1,...,n) have to be processed on m *machines* M_i (i = 1,...,m). Throughout, we assume that each machine can process at most one job at a time and that each job can be processed on at most one machine at a time. Various job, machine and scheduling characteristics are reflected by a 3-field problem classification $\alpha|\beta|\gamma$, to be introduced in this section.

2.2. Job data

In the first place, the following data can be specified for each J_j:

- a *number of operations* m_j;
- one or more *processing times* p_j or p_{ij}, that J_j has to spend on the various machines on which it requires processing;
- a *release date* r_j, on which J_j becomes available for processing;
- a *due date* d_j, by which J_j should ideally be completed;
- a *weight* w_j, indicating the relative importance of J_j;
- a nondecreasing real *cost function* f_j, measuring the cost $f_j(t)$ incurred if J_j is completed at time t.

In general, m_j, p_j, p_{ij}, r_j, d_j and w_j are integer variables.

2.3. Machine environment

We shall now describe the first field $\alpha = \alpha_1\alpha_2$ specifying the machine environment. Let $\circ$ denote the empty symbol.

If $\alpha_1 \in \{\circ,P,Q,R\}$, each J_j consists of a single operation that can be processed on any M_i; the processing time of J_j on M_i is p_{ij}. The four values are characterized as follows:

- $\alpha_1 = \circ$: *single machine*; $p_{1j} = p_j$;
- $\alpha_1 = P$: *identical parallel machines*; $p_{ij} = p_j$ (i = 1,...,m);
- $\alpha_1 = Q$: *uniform parallel machines*; $p_{ij} = q_i p_j$ for a given *speed factor* q_i of M_i (i = 1,...,m);
- $\alpha_1 = R$: *unrelated parallel machines*.

If $\alpha_1 = O$, we have an *open shop*, in which each J_j consists of a set of operations $\{O_{1j},...,O_{mj}\}$. O_{ij} has to be processed on M_i during p_{ij} time units, but the order in which the operations are executed is immaterial. If

$\alpha_1 \in \{F,J\}$, an ordering is imposed on the set of operations corresponding to each job. If $\alpha_1 = F$, we have a *flow shop*, in which each J_j consists of a chain $(O_{1j},\ldots,O_{mj})$. O_{ij} has to be processed on M_i during p_{ij} time units. If $\alpha_1 = J$, we have a *job shop*, in which each J_j consists of a chain $(O_{1j},\ldots,O_{m_jj})$. O_{ij} has to be processed on a given machine μ_{ij} during p_{ij} time units, with $\mu_{i-1,j} \neq \mu_{ij}$ for $i = 2,\ldots,m_j$.

If α_2 is a positive integer, then m is *constant* and equal to α_2. If $\alpha_2 = \circ$, then m is assumed to be *variable*. Obviously, $\alpha_1 = \circ$ if and only if $\alpha_2 = 1$.

2.4. Job characteristics

The second field $\beta \subset \{\beta_1,\ldots,\beta_6\}$ indicates a number of job characteristics, which are defined as follows.

1. $\beta_1 \in \{pmtn,\circ\}$
   - $\beta_1 = pmtn$: *Preemption* (job splitting) is allowed; the processing of any operation may be interrupted and resumed at a later time.
   - $\beta_1 = \circ$: No preemption is allowed.
2. $\beta_2 \in \{res,res1,\circ\}$
   - $\beta_2 = res$: The presence of s limited *resources* R_h $(h = 1,\ldots,s)$ is assumed, with the property that each J_j requires the use of r_{hj} units of R_h at all times during its execution. Of course, at no time may more than 100% of any resource be in use.
   - $\beta_2 = res1$: The presence of only a *single resource* is assumed.
   - $\beta_2 = \circ$: No resource constraints are specified.
3. $\beta_3 \in \{prec,tree,\circ\}$
   - $\beta_3 = prec$: A *precedence relation* $<$ between the jobs is specified. It is derived from a directed acyclic graph G with vertex set $\{1,\ldots,n\}$. If G contains a directed path from j to k, we write $J_j < J_k$ and require that J_j is completed before J_k can start.
   - $\beta_3 = tree$: G is a *rooted tree* with either outdegree at most one for each vertex or indegree at most one for each vertex.
   - $\beta_3 = \circ$: No precedence relation is specified.
4. $\beta_4 \in \{r_j,\circ\}$
   - $\beta_4 = r_j$: *Release dates* that may differ per job are specified.

$\beta_4 = \circ$: We assume that $r_j = 0$.

5. $\beta_5 \in \{m_j \leq \bar{m}, \circ\}$

$\beta_5 = m_j \leq \bar{m}$: A *constant upper bound* on m_j is specified (only if $\alpha_1 = J$).

$\beta_5 = \circ$: No such bound is specified.

6. $\beta_6 \in \{p_{ij}=1, \underline{p} \leq p_{ij} \leq \bar{p}, \circ\}$

$\beta_6 = p_{ij}=1$: Each operation has *unit processing time*.

$\beta_6 = \underline{p} \leq p_{ij} \leq \bar{p}$: *Constant lower and upper bounds* on p_{ij} are specified.

$\beta_6 = \circ$: No such bounds are specified.

2.5. Optimality criteria

The third field $\gamma \in \{f_{max}, \sum f_j\}$ refers to the optimality criterion chosen. Given a schedule, we can compute for each J_j:

- the *completion time* C_j;
- the *lateness* $L_j = C_j - d_j$;
- the *tardiness* $T_j = \max\{0, C_j - d_j\}$;
- the *unit penalty* U_j = if $C_j \leq d_j$ then 0 else 1.

The optimality criteria most commonly chosen involve the minimization of

$$f_{max} \in \{C_{max}, L_{max}\}$$

where $f_{max} = \max_j\{f_j(C_j)\}$ with $f_j(C_j) = C_j, L_j$, respectively, or

$$\sum f_j \in \{\sum C_j, \sum T_j, \sum U_j, \sum w_j C_j, \sum w_j T_j, \sum w_j U_j\}$$

where $\sum f_j = \sum_{j=1}^{n} f_j(C_j)$ with $f_j(C_j) = C_j, T_j, U_j, w_j C_j, w_j T_j, w_j U_j$, respectively.

It should be noted that $\sum w_j C_j$ and $\sum w_j L_j$ differ by a constant $\sum w_j d_j$ and hence are *equivalent*. Furthermore, any schedule minimizing L_{max} also minimizes T_{max} and U_{max}, but not *vice versa*.

The optimal value of γ will be denoted by γ^*, the value produced by an (approximation) algorithm A by $\gamma(A)$. If a known upper bound ρ on $\gamma(A)/\gamma^*$ is best possible in the sense that examples exist for which $\gamma(A)/\gamma^*$ equals or asymptotically approaches ρ, this will be denoted by a dagger ($\dagger$).

2.6. Examples

$1|prec|L_{max}$: minimize maximum lateness on a single machine subject to general precedence constraints. This problem can be solved in

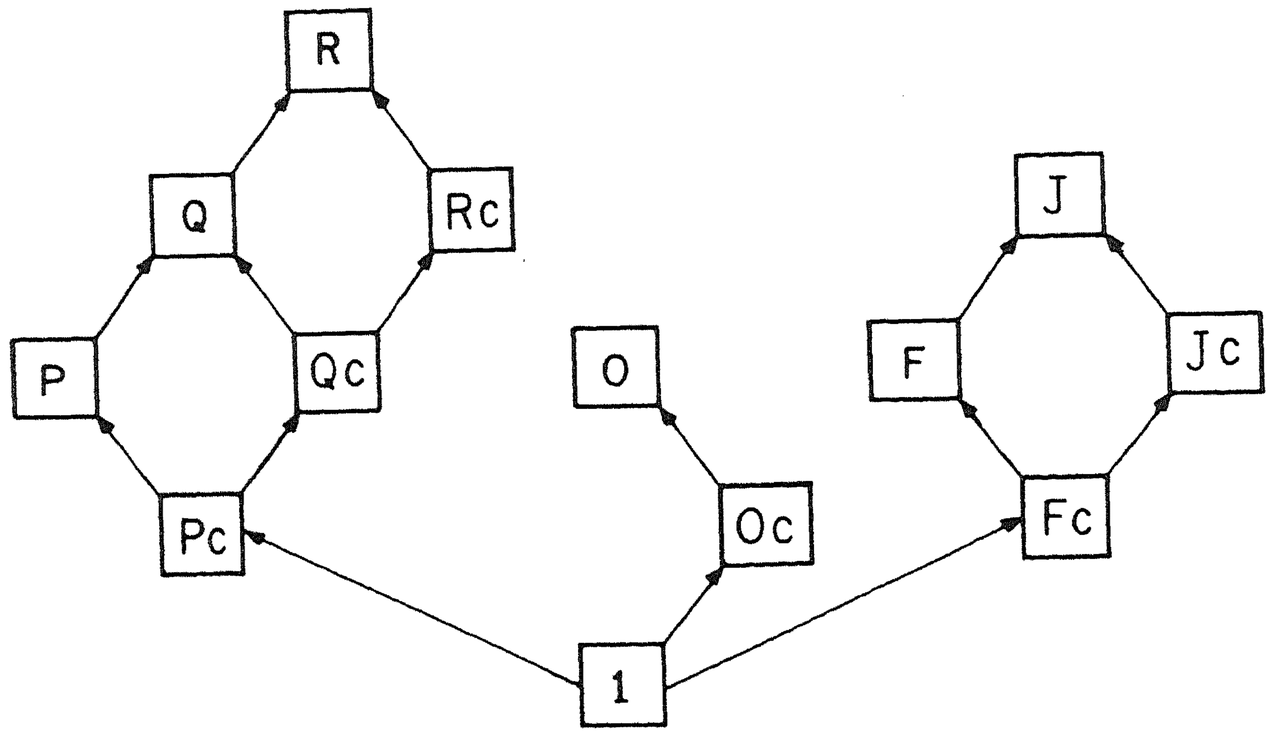


Figure 2.1 G_1; c denotes an integer constant.

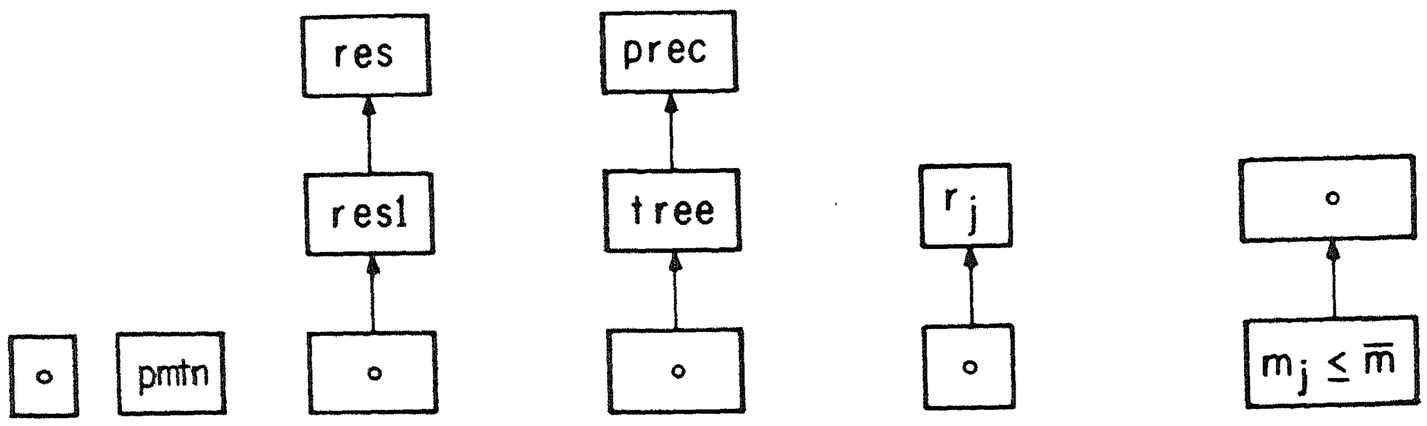


Figure 2.2 G_2. Figure 2.3 G_3. Figure 2.4 G_4. Figure 2.5 G_5. Figure 2.6 G_6.

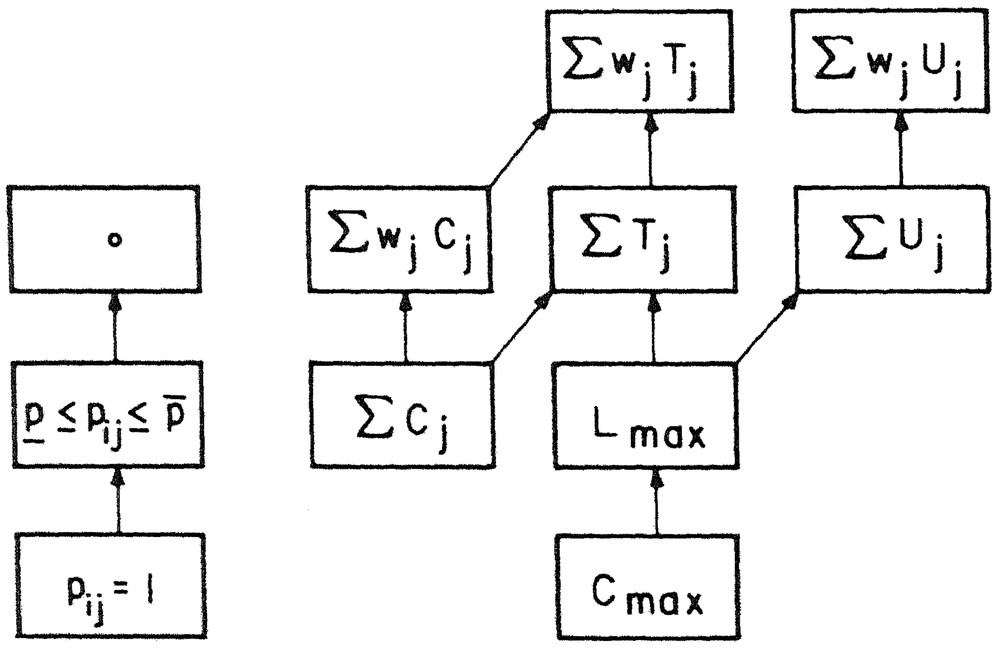


Figure 2.7 G_7. Figure 2.8 G_8.

polynomial time (Section 3.2).

$R|pmtn|\sum C_j$: minimize total completion time on a variable number of unrelated parallel machines, allowing preemption. The complexity of this problem is unknown (Section 4.4.3).

$J3|p_{ij}=1|C_{max}$: minimize maximum completion time in a 3-machine job shop with unit processing times. This problem is NP-hard (Section 5.4.1).

2.7. Reducibility among scheduling problems

Each scheduling problem in the class outlined above corresponds to a 8-tuple $(v_i)_{i=1}^{8}$, where v_i is a vertex of graph G_i, drawn in Figure 2.i ($i = 1,\ldots,8$). For two problems $P' = (v_i')_{i=1}^{8}$ and $P = (v_i)_{i=1}^{8}$, we write $P' \rightarrow P$ if either $v_i' = v_i$ or G_i contains a directed path from v_i' to v_i, for $i = 1,\ldots,8$. The reader should verify that $P' \rightarrow P$ implies $P' \propto P$. The graphs thus define elementary reductions among scheduling problems. It follows that

- if $P' \rightarrow P$ and $P \in \mathcal{P}$, then $P' \in \mathcal{P}$;
- if $P' \rightarrow P$ and P' is NP-hard, then P is NP-hard.

3. SINGLE MACHINE PROBLEMS

3.1. Introduction

The single machine case has been the object of extensive research ever since the seminal work by Jackson [Jackson 1955] and Smith [Smith 1956]. We will give a brief survey of the principal results, classifying them according to the optimality criterion chosen. As a general result, we note that if all $r_j = 0$ we need only consider schedules without preemption and without machine idle time [Conway *et al.* 1967].

3.2. Minimizing maximum cost

The most general result in this section is an $O(n^2)$ algorithm to solve $1|prec|f_{max}$ for arbitrary nondecreasing cost functions [Lawler 1973]. At each step of the algorithm, let S denote the index set of unscheduled jobs, let $p(S) = \sum_{j \in S} p_j$, and let $S' \subset S$ indicate the jobs all whose successors have been scheduled. One selects J_k for the last position among $\{J_j | j \in S\}$ by requiring that $f_k(p(S)) \leq f_j(p(S))$ for all $j \in S'$.

For $1||L_{max}$, this procedure specializes to *Jackson's rule*: schedule the jobs according to nondecreasing due dates [Jackson 1955]. Introduction of release dates turns this problem into a unary NP-hard one [Lenstra *et al.* 1977].

$1|prec,r_j,p_j=1|L_{max}$ and $1|pmtn,prec,r_j|L_{max}$ can still be solved in polynomial time: first update release and due dates so that they suitably reflect the precedence constraints and then apply Jackson's rule continually to the set of available jobs [Lageweg *et al.* 1976].

Various elegant enumerative methods exist for solving $1|prec,r_j|L_{max}$. Baker and Su [Baker & Su 1974] obtain a lower bound by allowing preemption; their enumeration scheme simply generates all *active schedules*, *i.e.* schedules in which one cannot decrease the starting time of an operation without increasing the starting time of another one. McMahon and Florian [McMahon & Florian 1975] propose a more ingenious approach; a slight modification of their algorithm allows very fast solution of problems with up to 80 jobs [Lageweg *et al.* 1976].

3.3. Minimizing total cost

3.3.1. $1|\beta|\sum w_j C_j$

The case $1||\sum w_j C_j$ can be solved in O(n log n) time by *Smith's rule*: schedule the jobs according to nonincreasing rations w_j/p_j [Smith 1956]. If all weights are equal, this amounts to the SPT rule of executing the jobs on the basis of shortest processing time first, a rule that is often used in more complicated situations without much empirical, let alone theoretical, support for its superior quality (*cf.* Section 5.4.2).

This result has been extended to O(n log n) algorithms that deal with *tree*-like [Horn 1972; Adolphson & Hu 1973; Sidney 1975] and even *series-parallel* [Knuth 1973B; Lawler 1978A] precedence constraints; see [Adolphson 1977] for an $O(n^3)$ algorithm covering a slightly more general case. The crucial observation to make here is that, if $J_j < J_k$ with $w_j/p_j < w_k/p_k$ and if all other jobs either have to precede J_j, succeed J_k, or are incomparable with both, then J_j and J_k are adjacent in at least one optimal schedule and can effectively be treated as one job with processing time p_j+p_k and weight w_j+w_k. By successive application of this device, starting at the bottom of the precedence tree, one will eventually obtain an optimal schedule. Addition of general precedence constraints results in NP-hardness, even if all $p_j = 1$ or all $w_j = 1$ [Lawler 1978A; Lenstra & Rinnooy Kan 1978A].

If release dates are introduced, $1|r_j|\sum C_j$ is already unary NP-hard [Lenstra *et al.* 1977]. In the preemptive case, $1|pmtn,r_j|\sum C_j$ can be solved by an obvious extension of Smith's rule, but, surprisingly, $1|pmtn,r_j|\sum w_j C_j$ is unary NP-hard [Labetoulle *et al.* 1978].

3.3.2. $1|\beta|\sum w_j T_j$

$1||\sum w_j T_j$ is a unary NP-hard problem [Lawler 1977; Lenstra *et al.* 1977], for which various enumerative solution methods have been proposed, some of which can be extended to cover arbitrary nondecreasing cost functions. Lower bounds developed for the problem involve a linear assignment relaxation using an underestimate of the cost of assigning J_j to position k [Rinnooy Kan *et al.* 1975], a fairly similar relaxation to a transportation problem [Gelders & Kleindorfer 1974, 1975], and relaxation of the requirement that the machine can process at most one job at a time [Fisher 1976A]. In the latter approach,

one attaches "prices" (*i.e.*, Lagrangean multipliers) to each unit-time interval. Multiplier values are sought for which a cheapest schedule does not violate the capacity constraint. The resulting algorithm is quite successful on problems with up to 50 jobs, although a straightforward but cleverly implemented dynamic programming approach [Baker & Schrage 1978] offers a surprisingly good alternative.

If all $p_j = 1$, we have a simple linear assignment problem, the cost of assigning J_j to position k being given by $f_j(k)$. If all $w_j = 1$, the problem can be solved by a pseudopolynomial algorithm in $O(n^4 \sum p_j)$ time [Lawler 1977]; the computational complexity of $1||\sum T_j$ with respect to a binary encoding remains an open question.

Addition of precedence constraints yields NP-hardness, even for $1|prec,p_j=1|\sum T_j$ [Lenstra & Rinnooy Kan 1978A].

If we introduce release dates, $1|r_j,p_j=1|\sum w_j T_j$ can again be solved as a linear assignment problem, whereas $1|r_j|\sum T_j$ is obviously unary NP-hard (*cf.* Section 2.7).

3.3.3. $1|\beta|\sum w_j U_j$

An algorithm due to Moore [Moore 1968] allows solution of $1||\sum U_j$ in $O(n \log n)$ time: jobs are added to the schedule in order of nondecreasing due dates, and if addition of J_j results in this job being completed after d_j, the scheduled job with the largest processing time is marked to be late and removed. This procedure can be extended to cover the case in which certain specified jobs have to be on time [Sidney 1973]. The problem also remains solvable in polynomial time if we add *agreeable weights* (*i.e.*, $p_j < p_k \Rightarrow w_j \geq w_k$) [Lawler 1976A] or *agreeable release dates* (*i.e.*, $d_j < d_k \Rightarrow r_j \leq r_k$) [Kise *et al.* 1978]. $1||\sum w_j U_j$ is binary NP-hard [Karp 1972], but can be solved by dynamic programming in $O(n\sum p_j)$ time [Lawler & Moore 1969].

Again, $1|prec,p_j=1|\sum U_j$ is NP-hard [Garey & Johnson 1976B], even for *chain*-like precedence constraints [Lenstra -].

Of course, $1|r_j|\sum U_j$ is unary NP-hard. The preemptive case $1|pmtn,r_j|\sum U_j$ is an intriguing open problem.

Very little work has been done on worst-case analysis of approximation algorithms for single machine problems. For $1||\sum w_j U_j$, Sahni [Sahni 1976]

presents algorithms A_k with $O(n^3k)$ running time such that

$$\sum w_j \bar{U}_j(A_k) / \sum w_j \bar{U}_j^* \geq 1 - \frac{1}{k},$$

where $\bar{U}_j = 1-U_j$. For $1|tree|\sum w_j U_j$, Ibarra and Kim [Ibarra & Kim 1975B] give algorithms B_k of order $O(kn^{k+2})$ with the same worst-case error bound.

4. PARALLEL MACHINE PROBLEMS

4.1. Introduction

Recall from Section 2.3 the definitions of *identical*, *uniform* and *unrelated* machines, denoted by P, Q and R, respectively.

Nonpreemptive parallel scheduling problems tend to be difficult. This can be inferred immediately from the fact that $P2||C_{max}$ and $P2||\sum w_j C_j$ are binary NP-hard [Bruno *et al.* 1974; Lenstra *et al.* 1977]. If we are to look for polynomial algorithms, it follows that we should either restrict attention to the special case $p_j = 1$, as we do in Section 4.2, or concern ourselves with the $\sum C_j$ criterion, as we do in the first three subsections of Section 4.3. The remaining part of Section 4.3 is entirely devoted to enumerative optimization methods and approximation algorithms for various NP-hard problems.

The situation is much brighter with respect to *preemptive* parallel scheduling. For example, $P|pmtn|C_{max}$ has long been known to admit a simple O(n) algorithm [McNaughton 1959]. Many new results for the $\sum C_j$, C_{max} and L_{max} criteria have been obtained quite recently. These are summarized in Section 4.4. With respect to other criteria, $P2|pmtn|\sum w_j C_j$ turns out to be NP-hard (see Section 4.4.1). Little is known about $P|pmtn|\sum T_j$ and $P|pmtn|\sum U_j$; these problems remain open. However, we know from Section 3 that $1|pmtn|\sum w_j T_j$ and $1|pmtn|\sum w_j U_j$ are already NP-hard.

4.2. Nonpreemptive scheduling: unit processing times

4.2.1. $Q|p_j=1|\sum f_j$, $Q|p_j=1|f_{max}$

A simple transportation network model provides an efficient solution method for $Q|p_j=1|\sum f_j$ and $Q|p_j=1|f_{max}$.

Let there be n sources j (j = 1,...,n) and mn sinks (i,k) (i = 1,...,m, k = 1,...,n). Set the cost of arc (j,(i,k)) equal to $c_{ijk} = f_j(kq_i)$. The arc flow x_{ijk} is to have the interpretation:

$$x_{ijk} = \begin{cases} 1 & \text{if } J_j \text{ is executed on } M_i \text{ in the k-th position,} \\ 0 & \text{otherwise.} \end{cases}$$

Then the problem is to minimize

$$\sum_{i,j,k} c_{ijk}x_{ijk} \quad \text{or} \quad \max_{i,j,k}\{c_{ijk}x_{ijk}\}$$

subject to

$$\sum_{i,k} x_{ijk} = 1 \qquad \text{for all } j,$$

$$\sum_{j} x_{ijk} \leq 1 \qquad \text{for all } i,k,$$

$$x_{ijk} \geq 0 \qquad \text{for all } i,j,k.$$

The time required to prepare the data for this transportation problem is $O(mn^2)$. A careful analysis reveals that the problem can be solved (in integers) in $O(n^3)$ time. Since we may assume that $m \leq n$, the overall running time is $O(n^3)$.

It may be noted that some special cases can be solved more efficiently. For instance, $P|p_j=1|\sum U_j$ can be solved in $O(n \log n)$ time [Lawler 1976A].

4.2.2. $P|prec,p_j=1|C_{max}$

$P|prec,p_j=1|C_{max}$ is known to be NP-hard [Ullman 1975; Lenstra & Rinnooy Kan 1978A]. It is an open question whether this remains true for any constant value of $m \geq 3$. The problem is in $\mathcal{P}$, however, if the precedence relation is of the *tree*-type or if $m = 2$.

$P|tree,p_j=1|C_{max}$ can be solved in $O(n)$ time by Hu's algorithm [Hu 1961; Hsu 1966; Sethi 1976A]. The *level* of a job is defined as the number of jobs in the unique path to the root of the precedence tree. At the beginning of each time unit, as many available jobs as possible are scheduled on the m machines, where highest priority is granted to the jobs with the largest levels. Thus, Hu's algorithm is a nonpreemptive *list scheduling* algorithm, whereby at each step the available job with the highest ranking on a priority list is assigned to the first machine that becomes available. It can also be viewed as a *critical path* scheduling algorithm: the next job chosen is the one which heads the longest current chain of unexecuted jobs.

If the precedence constraints are in the form of an *intree* (each job has at most one successor), then Hu's algorithm can be adapted to minimize L_{max}; in the case of an *outtree* (each job has at most one predecessor), the L_{max} problem turns out to be NP-hard [Brucker *et al.* 1977].

$P2|prec,p_j=1|C_{max}$ can be solved in $O(n^2)$ time [Coffman & Graham 1972]. Previous polynomial-time algorithms for this problem are given in [Fujii

et al. 1969, 1971; Muraoka 1971].

In the approach due to Fujii *et al*., an undirected graph is constructed with vertices corresponding to jobs and edges {j,k} whenever J_j and J_k can be executed simultaneously, *i.e.*, $J_j \not< J_k$ and $J_k \not< J_j$. An optimal schedule is then derived from a maximum cardinality matching in the graph. Such a matching can be found in $O(n^3)$ time [Lawler 1976B].

The Coffman-Graham approach leads to a list algorithm. First the jobs are labelled in the following way. Suppose labels 1,...,k have been applied and S is the subset of unlabelled jobs all of whose successors have been labelled. Then a job in S is given the label k+1 if the labels of its immediate successors are *lexicographically minimal* with respect to all jobs in S. The priority list is given by ordering the jobs according to decreasing labels. It is possible to execute this algorithm in time almost linear in n+a, where a is the number of arcs in the *transitive reduction* of the precedence graph (all arcs implied by transitivity removed) [Sethi 1976B]. Note, however, that construction of such a representation requires $O(n^{2.8})$ time [Aho *et al*. 1972].

Garey and Johnson present polynomial algorithms for $P2|prec,p_j=1|C_{max}$ where, in addition, each job becomes available at its *release date* and has to meet a given *deadline*. In this approach, one obtains an optimal schedule by processing the jobs in order of increasing modified deadlines. This modification requires $O(n^2)$ time if all $r_j = 0$ [Garey & Johnson 1976B] and $O(n^3)$ time in the general case [Garey & Johnson 1977].

We note that $P|prec,p_j=1|\sum C_j$ is NP-hard [Lenstra & Rinnooy Kan 1978A]. Hu's algorithm does not yield an optimal $\sum C_j$ schedule in the case of intrees, but in the case of outtrees critical path scheduling minimizes both C_{max} and $\sum C_j$ [Rosenfeld -]. The Coffman-Graham algorithm also minimizes $\sum C_j$ [Garey -].

As far as approximation algorithms for $P|prec,p_j=1|C_{max}$ are concerned, the NP-hardness proof given in [Lenstra & Rinnooy Kan 1978A] implies that, unless $P = NP$, the best possible worst-case bound for a polynomial-time algorithm would be $\frac{4}{3}$. The performance of both Hu's algorithm and the Coffman-Graham algorithm has been analyzed.

When critical path (CP) scheduling is used, Chen and Liu [Chen 1975; Chen & Liu 1975] and Kunde [Kunde 1976] show that

$$C_{max}(CP)/C^*_{max} \leq \begin{cases} \frac{4}{3} & \text{for } m = 2, \\ 2 - \frac{1}{m-1} & \text{for } m \geq 3. \end{cases} \qquad (\dagger)$$

In [Kaufman 1972] an example is constructed for which *no* CP schedule is optimal.

Lam and Sethi [Lam & Sethi 1977] use the Coffman-Graham (CG) algorithm to generate lists and show that

$$C_{max}(CG)/C^*_{max} \leq 2 - \frac{2}{m} \quad (m \geq 2). \qquad (\dagger)$$

If SS denotes the algorithm which schedules as the next job the one having the greatest number of successors then it can be shown [Ibarra & Kim 1976] that

$$C_{max}(SS)/C^*_{max} \leq \frac{4}{3} \quad \text{for } m = 2. \qquad (\dagger)$$

Examples show that this bound does not hold for $m \geq 3$.

Finally, we mention some results for the more general case in which $p_j \in \{1,k\}$. For $k = 2$, both $P2|prec,1\leq p_j\leq 2|C_{max}$ and $P2|prec,1\leq p_j\leq 2|\sum C_j$ are NP-hard [Ullman 1975; Lenstra & Rinnooy Kan 1978A]. For $P2|prec,p_j\in\{1,k\}|C_{max}$, Goyal [Goyal 1977B] proposes a generalized version of the Coffman-Graham algorithm (GCG) and shows that

$$C_{max}(GCG)/C^*_{max} \leq \begin{cases} \frac{4}{3} & \text{for } k = 2, \\ \frac{3}{2} - \frac{1}{2k} & \text{for } k \geq 3. \end{cases} \qquad (\dagger)$$

4.2.3. $P|res,\beta,p_j=1|C_{max}$

We now take up the variation in which *resource* constraints enter the model. $P2|res,p_j=1|C_{max}$ can be formulated and solved as a maximum cardinality matching problem in an obvious way. However, $P2|res1,tree,p_j=1|C_{max}$ and $P3|res1,p_j=1|C_{max}$ are unary NP-hard [Garey & Johnson 1975].

For the case $P|res,prec,p_j=1,m\geq n|C_{max}$, the following results for list scheduling (LS) using an arbitrary priority list are known [Garey *et al.* 1976B]:

$$C_{max}(LS)/C^*_{max} \leq \frac{1}{2}\, sC^*_{max} + \frac{1}{2}\, s + 1$$

and examples exist with

$$C_{max}(LS)/C^*_{max} \geq \frac{1}{2}\, sC^*_{max} + \frac{1}{2}\, s + 1 - 2s/C^*_{max}.$$

For the CP scheduling algorithm, the bound improves considerably:

$$C_{max}(CP)/C^*_{max} \leq \frac{17}{10}\, s + 1 \quad (s \geq 0). \qquad (\dagger)$$

Let DMR denote the algorithm which schedules jobs according to decreasing maximum resource requirement. Then

$$C_{max}(DMR)/C^*_{max} \leq \frac{17}{10}\, s + 1.$$

In the other direction, examples are given in [Garey *et al.* 1976B] for any $\varepsilon > 0$ with

$$C_{max}(DMR)/C^*_{max} > \sum_{i=1}^{\infty} \frac{1}{a_i} - \varepsilon = 1.69\ldots - \varepsilon$$

where $a_1 = 1$ and $a_{i+1} = a_i(a_i+1)$ for $i \geq 1$.

An even better bound applies to the case of independent jobs, *i.e.*, $P|res,p_j=1,m\geq n|C_{max}$:

$$C_{max}(LS) \leq (s + \frac{7}{10})C^*_{max} + \frac{7}{2} \quad (s \geq 1),$$

where the coefficient of C^*_{max} is best possible.

The case $P|res1,p_j=1,m\geq n|C_{max}$ has been the subject of intensive study (under the name of *bin packing*) during the past few years. The problem can be viewed as one of placing a number of items with weights r_{1j} into a minimum number of bins of capacity 1. It is also known as the *one-dimensional cutting stock* problem. It is for this scheduling model that some of the deepest results have been obtained. Rather than giving a complete survey of what is known for this model, we shall instead give a sample of typical results and refer the reader to the literature for details [Johnson 1973, 1974A; Johnson *et al.* 1974; Graham 1976; Garey & Johnson 1976C].

Given a list L of items, the *first-fit* (FF) algorithm packs the items successively in the order in which they occur in L, always placing each item into the first bin into it will validly fit (*i.e.*, so that the sum of the weights in the bin does not exceed its capacity 1). The number of bins required by the packing is just the time required to execute the jobs using L as a priority list. If instead of choosing the first bin into which an item will fit, we always choose the bin for which the unused capacity is minimized, then the resulting procedure is called the *best-fit* (BF) algorithm. Finally, when L is first ordered by decreasing weights and then first-fit or best-fit packed, the resulting algorithm is called *first-fit*

decreasing (FFD) or *best-fit decreasing* (BFD), respectively.

The basic results which apply to these algorithms are the following [Johnson *et al.* 1974; Garey *et al.* 1976B]:

$$C_{max}(FF) \leq \lceil \frac{17}{10} C^*_{max} \rceil;$$

$$C_{max}(BF) \leq \lceil \frac{17}{10} C^*_{max} \rceil;$$

$$C_{max}(FFD) \leq \frac{11}{9} C^*_{max} + 4;$$

$$C_{max}(BFD) \leq \frac{11}{9} C^*_{max} + 4.$$

The only known proofs of the last two inequalities are extremely lengthy. Examples can be given which show that the coefficients $\frac{17}{10}$ and $\frac{11}{9}$ are best possible.

If constraints are made on the resource requirements, *i.e.*, $\underline{r} \leq r_{1j} \leq \bar{r}$ for all j, then the following results hold:

$$\text{if } \underline{r} \geq \frac{1}{6} \text{ then } C_{max}(BFD) \leq C_{max}(FFD);$$

$$\text{if } \underline{r} \geq \frac{1}{5} \text{ then } C_{max}(BFD) = C_{max}(FFD);$$

$$\text{if } \bar{r} \leq \frac{1}{2} \text{ then } C_{max}(FF)/C^*_{max} \leq 1 + \lfloor \bar{r}^{-1} \rfloor^{-1};$$

$$\text{if } \bar{r} \in (\frac{8}{29}, \frac{1}{2}] \text{ then } C_{max}(FFD) \leq \frac{71}{60} C^*_{max} + c \quad \text{for some constant } c.$$

For these and a number of similar results, the reader is referred to [Graham 1976].

Krause [Krause 1973] (see also [Krause *et al.* 1975, 1977]) considers the case $P|res1,p_j=1|C_{max}$. He proves that

$$(C_{max}(LS)-2)/C^*_{max} < \frac{27}{10} - \frac{24}{10m},$$

$$(C_{max}(DMR)-1)/C^*_{max} \leq 2 - \frac{2}{m} \quad (m \geq 2),$$

and he gives examples for which

$$C_{max}(LS)/C^*_{max} \geq \frac{27}{10} - \frac{37}{10m}.$$

Krause also proves several bounds for the *preemptive* case $P|pmtn,res1|C_{max}$, one of which is

$$C_{max}(DMR)/C^*_{max} < 3 - \frac{3}{m} \quad (m \geq 2).$$

Goyal [Goyal 1977A] studies the case $P|res1,prec,p_j=1|C_{max}$ with the restriction that each resource requirement is either zero or 100%. Thus, two jobs both requiring the use of the resource can never be executed simultaneously. This problem is already NP-hard for m = 2 [Coffman 1976]. Goyal proves that

$$C_{max}(LS)/C^*_{max} \leq 3 - \frac{2}{m}, \qquad (\dagger)$$

$$C_{max}(CG)/C^*_{max} \leq \frac{3}{2} \quad \text{for } m = 2,$$

where in the latter case a priority list is formed according to the CG labelling algorithm described earlier.

4.3. Nonpreemptive scheduling: general processing times

4.3.1. $P||\sum w_jC_j$

The following generalization of the SPT rule for $1||\sum C_j$ (see Section 3.3.1) solves $P||\sum C_j$ in O(n log n) time [Conway *et al.* 1967]. Assume n = km (dummy jobs with zero processing times can be added if not) and suppose $p_1 \leq \ldots \leq p_n$. Assign the m jobs $J_{(j-1)m+1}, J_{(j-1)m+2}, \ldots, J_{jm}$ to m different machines (j = 1,...,k) and execute the k jobs assigned to each machine in SPT order.

Bruno, Coffman and Sethi [Bruno *et al.* 1974] consider the algorithm RPT: first apply list scheduling on the basis of largest processing time first (LPT), then *reverse* the order of jobs on each machine, and finally left justify the schedule. RPT has the same behavior as LPT with respect to the C_{max} criterion (see Section 4.3.5.1); however, it only yields

$$\sum C_j(RPT)/\sum C^*_j \leq m. \qquad (\dagger)$$

With respect to $P||\sum w_jC_j$, similar heuristics are described and tested empirically by Baker and Merten [Baker & Merten 1973].

Eastman, Even and Isaacs [Eastman *et al.* 1964] show that after renumbering the jobs according to nonincreasing ratios w_j/p_j

$$\sum w_jC_j(LS) - \frac{1}{2}\sum_{j=1}^{n} w_jp_j \geq \frac{1}{m}\left(\sum_{j=1}^{n}\sum_{k=1}^{j} w_jp_k - \frac{1}{2}\sum_{j=1}^{n} w_jp_j\right). \qquad (\dagger)$$

It follows from this inequality that

$$\sum w_j C_j^* \geq \frac{m+n}{m(n+1)} \sum_{j=1}^{n} \sum_{k=1}^{j} w_j p_k .$$

In [Elmaghraby & Park 1974; Barnes & Brennan 1977] branch-and-bound algorithms based on this lower bound are developed.

Sahni [Sahni 1976] constructs algorithms A_k (in the same spirit as his approach for $1||\sum w_j U_j$ mentioned in Section 3.3.3) with $O(n(n^2k)^{m-1})$ running time for which

$$\sum w_j C_j(A_k) / \sum w_j C_j^* \leq 1 + \frac{1}{k}.$$

For $m = 2$, the running time of A_2 can be improved to $O(n^2k)$.

4.3.2. $Q||\sum C_j$

The algorithm for solving $P||\sum C_j$ given in the previous section can be generalized to the case of uniform machines [Conway *et al.* 1967]. If J_j is the k-th last job executed on M_i, a cost contribution $kp_{ij} = kq_i p_j$ is incurred. $\sum C_j$ is a weighted sum of the p_j and is minimized by matching the n smallest weights kq_i in nondecreasing order with the p_j in nonincreasing order. The procedure can be implemented to run in $O(n \log n)$ time [Horowitz & Sahni 1976].

4.3.3. $R||\sum C_j$

$R||\sum C_j$ can be formulated and solved as an $m \times n$ transportation problem [Horn 1973; Bruno *et al.* 1974). Let

$$x_{ijk} = \begin{cases} 1 & \text{if } J_j \text{ is the k-th last job executed on } M_i, \\ 0 & \text{otherwise.} \end{cases}$$

Then the problem is to minimize

$$\sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{n} kp_{ij} x_{ijk}$$

subject to

$$\sum_{i=1}^{m} \sum_{k=1}^{n} x_{ijk} = 1 \quad \text{for all } j,$$

$$\sum_{j=1}^{n} x_{ijk} \leq 1 \quad \text{for all } i,k,$$

$$x_{ijk} \geq 0 \quad \text{for all } i,j,k.$$

This problem, like the similar one in Section 4.2.1, can be solved in $O(n^3)$ time.

4.3.4. *Other cases: enumerative optimization methods*

As we noted in Section 4.1, $P2||C_{max}$ and $P2||\sum w_jC_j$ are NP-hard. Hence it seems fruitless to attempt to find polynomial-time optimization algorithms for criteria other than $\sum C_j$. Moreover, $P2|tree|\sum C_j$ is known to be NP-hard, both for intrees and outtrees [Sethi 1977]. It follows that it is also not possible to extend the above algorithms to problems with precedence constraints. The only remaining possibility for optimization methods seems to be implicit enumeration.

$R||C_{max}$ can be solved by a branch-and-bound procedure described in [Stern 1976]. The enumerative approach for identical machines in [Bratley *et al.* 1975] allows inclusion of release dates and deadlines as well.

A general dynamic programming technique [Rothkopf 1966; Lawler & Moore 1969] is applicable to parallel machine problems with the C_{max}, L_{max}, $\sum w_jC_j$ and $\sum w_jU_j$ optimality criteria, and even to problems with the $\sum w_jT_j$ criterion in the special case of a common due date.

Let us define $F_j(t_1,\ldots,t_m)$ as the minimum cost of a schedule without idle time for $J_1,\ldots,J_j$ subject to the constraint that the last job on M_i is completed at time t_i, for $i = 1,\ldots,m$. Then, in the case of f_{max} criteria,

$$F_j(t_1,\ldots,t_m) = \min_{1\le i\le m}\{\max\{f_j(t_i),F_{j-1}(t_1,\ldots,t_i-p_{ij},\ldots,t_m)\}\},$$

and in the case of $\sum f_j$ criteria,

$$F_j(t_1,\ldots,t_m) = \min_{1\le i\le m}\{f_j(t_i) + F_{j-1}(t_1,\ldots,t_i-p_{ij},\ldots,t_m)\}.$$

In both cases, the initial conditions are

$$F_0(t_1,\ldots,t_m) = \begin{cases} 0 & \text{if } t_i = 0 \text{ for } i = 1,\ldots,m, \\ \infty & \text{otherwise.} \end{cases}$$

Appropriate implementation of these equations yields $O(mnC^{m-1})$ computations for a variety of problems, where C is an upper bound on the completion time of any job in an optimal schedule. Among these problems are $P|r_j|C_{max}$, $Q||L_{max}$ and $Q||\sum w_jC_j$. $P||\sum w_jU_j$ can be solved in $O(mn(\max_j\{d_j\})^m)$ time.

Still other dynamic programming approaches can be used to solve $P||\sum f_j$ and $P||\sum f_{max}$ in $O(m \min\{3^n, n2^n C\})$ time, but these are probably of little practical importance.

4.3.5. *Other cases: approximation algorithms*

4.3.5.1. $P||C_{max}$

By far the most studied scheduling model from the viewpoint of approximation algorithms is $P||C_{max}$. We refer to [Garey *et al.* 1978] for an easily readable introduction into the techniques involved in many of the "performance guarantees" mentioned below.

Perhaps the earliest and simplest result on the worst-case performance of list scheduling is given in [Graham 1966]:

$$C_{max}(LS)/C^*_{max} \leq 2 - \frac{1}{m}. \qquad (\dagger)$$

If the jobs are selected in LPT order, then the bound can be considerably improved, as is shown in [Graham 1969]:

$$C_{max}(LPT)/C^*_{max} \leq \frac{4}{3} - \frac{1}{3m}. \qquad (\dagger)$$

A somewhat better algorithm, called *multifit* (MF) and based on a completely different principle, is given in [Coffman *et al.* 1978]. The idea behind MF is to find (by binary search) the smallest "capacity" a set of m "bins" can have and still accommodate all jobs when the jobs are taken in order of non-increasing p_j and each job is placed into the first bin into which it will fit. The set of jobs in the i-th bin will be processed by M_i. If k packing attempts are made, the algorithm (denoted by MF_k) runs in time $O(n \log n + knm)$ and satisfies

$$C_{max}(MF_k)/C^*_{max} \leq 1.22 + \frac{1}{2^k}.$$

We note that if the jobs are not ordered by decreasing p_j then all that can be guaranteed by this method is

$$C_{max}(MF)/C^*_{max} \leq 2 - \frac{2}{m+1}. \qquad (\dagger)$$

The following algorithm Z_k was introduced in [Graham 1969]: schedule the k largest jobs optimally, then list schedule the remaining jobs arbitrarily.

It is shown in [Graham 1969] that

$$C_{max}(Z_k)/C^*_{max} \leq 1 + (1 - \frac{1}{m})/(1 + \left[\frac{k}{m}\right])$$

and that when m divides k, this is best possible. Thus, we can make the bound as close to 1 as desired by taking k sufficiently large. Unfortunately, the best bound on the running time is $O(n^{km})$.

A very interesting algorithm for $P||C_{max}$ is given by Sahni [Sahni 1976]. He presents algorithms A_k with $O(n(n^2k)^{m-1})$ running time which satisfy

$$C_{max}(A_k)/C^*_{max} \leq 1 + \frac{1}{k}.$$

For m = 2, algorithm A_2 can be improved to run in time $O(n^2k)$. As in the cases of $1||\sum w_jU_j$ (Section 3.3.3) and $P||\sum w_jC_j$ (Section 4.3.1), the algorithms A_k are based on a clever combination of dynamic programming and "rounding" and are beyond the scope of the present discussion.

Several bounds are available which take into account the processing times of the jobs. In [Graham 1969] it is shown that

$$C_{max}(LS)/C^*_{max} \leq 1 + (m-1)\max_j\{p_j\}/\sum_j p_j.$$

For the case of LPT, Ibarra and Kim [Ibarra & Kim 1977] prove that

$$C_{max}(LPT)/C^*_{max} \leq 1 + \frac{2(m-1)}{n} \quad \text{for } n \geq 2(m-1)\max_j\{p_j\}/\min_j\{p_j\}.$$

The following *local interchange* (LI) algorithm gives a slight improvement over the original $2 - \frac{1}{m}$ bound: assign jobs to machines arbitrarily, then move individual jobs and interchange pairs of jobs as long as C_{max} can be decreased by any such change. It then follows [Graham -] that

$$C_{max}(LI)/C^*_{max} \leq 2 - \frac{2}{m+1}. \qquad (\dagger)$$

In [Bruno *et al.* 1974] the Conway-Maxwell-Miller (CMM) algorithm for solving $P||\sum C_j$ (see Section 4.3.1) is considered. Let $C^*_{max}(CMM)$ be the minimum completion time among all schedules that can be generated by CMM. Then

$$C^*_{max}(CMM)/C_{max}(LPT) \leq 2 - \frac{1}{m}, \qquad (\dagger)$$

$$C^*_{max}(CMM)/C^*_{max} \leq 2 - \frac{1}{m}. \qquad (\dagger)$$

An interesting variation on the C_{max} criterion arises in the work of Chandra and Wong [Chandra & Wong 1975]. They consider the case $P||\sum B_i^2$, where B_i

denotes the completion time of the job executed last on M_i, and establish the surprisingly good behavior of LPT:

$$\sum B_i^2(\text{LPT})/\sum B_i^{2*} \leq \frac{25}{24}.$$

They also construct examples for which

$$\sum B_i^2(\text{LPT})/\sum B_i^{2*} \geq \frac{37}{36} - \frac{1}{36m}.$$

Finally, we mention the following result [Garey *et al.* -]. For any LPT schedule, let t_{max} denote the latest possible time at which a machine can become idle and let t_{min} denote the earliest time a machine can be idle. Then

$$t_{max}/t_{min} \leq \frac{4m-2}{3m-1}$$

and this bound is best possible.

4.3.5.2. $Q||C_{max}$

In the literature on approximation algorithms for scheduling problems, it is usually assumed that *unforced idleness* (UI) of machines is *not* allowed, *i.e.*, a machine cannot be idle when jobs are available. In the case of identical machines, UI need not occur in an optimal schedule if there are no precedence constraints or if all $p_j = 1$. Allowing UI may yield better solutions, however, in the cases which are to be discussed in Sections 4.3.5.2-6. The optimal value of C_{max} under the restriction of *no* UI will be denoted by C^*_{max}, the optimum if UI *is* allowed by $C^*_{max}(\text{UI})$.

Liu and Liu [Liu & Liu 1974A, 1974B, 1974C] study numerous questions dealing with uniform machines. We outline some of their results.

For the case that $q_1 = \ldots = q_{m-1} = 1$, $q_m = q \geq 1$, they prove

$$C_{max}(\text{LPT})/C^*_{max}(\text{UI}) \leq \begin{cases} \frac{2(m-1+q)}{q+2} & \text{for } q \leq 2, \\ \frac{m-1+q}{2} & \text{for } q > 2. \end{cases}$$

For the general case, they define the algorithm A_k as follows: schedule the k longest jobs first, resulting in a completion time of $C_k(A_k)$, and schedule the remaining tasks for a total completion time of $C_{max}(A_k)$. If $C_{max}(A_k) > C_k(A_k)$, then

$$C_{max}(A_k)/C^*_{max}(\text{UI}) \leq 1 + \frac{1}{Q} - \frac{1}{Q\sum_i q_i}$$

where all $q_i \geq 1$ and

$$Q = \max\left\{\min_j\left\{\left\lceil\frac{k+1}{\sum_i\lceil q_i\rceil}\right\rceil\cdot\frac{\lceil q_j\rceil}{q_j} - \frac{\lceil q_j\rceil^{-1}}{q_j}\right\}, \frac{k+1}{\sum_i q_i}\right\}.$$

This is best possible when the q_i are integers and $\sum_i q_i$ divides k.

Gonzalez, Ibarra and Sahni [Gonzalez *et al.* 1977] consider the following generalization LPT' of LPT: assign each job, in order of nonincreasing processing time, to the machine on which it will be completed soonest. Thus, unforced idleness may occur in the schedule.

For the case that $q_1 = \ldots = q_{m-1} = 1$, $q_m \geq 1$, they show that

$$C_{max}(LPT')/C^*_{max} \geq \begin{cases} \frac{1+\sqrt{17}}{4} & \text{for } m = 2, \\ \frac{2}{3} - \frac{1}{2m} & \text{for } m > 2. \end{cases} \qquad (\dagger)$$

For the general case, they show

$$C_{max}(LPT')/C^*_{max} \leq 2 - \frac{2}{m+1}.$$

Also, examples are given for which $C_{max}(LPT')/C^*_{max}$ approaches $\frac{3}{2}$ as m tends to infinity.

4.3.5.3. $R||C_{max}$

Very little is known about approximation algorithms for this model. Ibarra and Kim [Ibarra & Kim 1977] consider six algorithms, typical of which is to schedule J_j on the machine that executes it fastest, *i.e.*, on an M_i with minimum p_{ij}. For all six algorithms A they prove

$$C_{max}(A)/C^*_{max} \leq m$$

with equality possible for four of the six. For the other two, they conjecture

$$C_{max}(A)/C^*_{max} \overset{?}{\leq} 2.$$

For the special case $R2||C_{max}$, they give a complicated algorithm G (however, with O(n log n) running time) such that

$$C_{max}(G)/C^*_{max} \leq \frac{1+\sqrt{5}}{2}. \qquad (\dagger)$$

In a variation on $R||C_{max}$, we assume that each J_j has a processing time p_j

and a fixed *memory requirement* $|J_j|$ and that each M_i has a *memory capacity* $|M_i|$. We require that $|M_i| \geq |J_j|$ in order for M_i to be able to execute J_j, *i.e.*,

$$p_{ij} = \begin{cases} p_j & \text{if } |M_i| \geq |J_j|, \\ \infty & \text{otherwise.} \end{cases}$$

Kafura and Shen [Kafura & Shen 1977] show

$$C_{max}(LS)/C^*_{max} \leq 1 + \log m.$$

They also note that when m is a power of 2, the bound can be achieved.

Suppose a list is formed in order of decreasing $|J_j|$; this algorithm is denoted by LMF (largest memory first). It can be shown [Kafura & Shen 1977] that

$$C_{max}(LMF)/C^*_{max} \leq 2 - \frac{1}{m}. \qquad (\dagger)$$

A refinement of LMF is LMTF where ties in $|J_j|$ are broken by decreasing order of p_j. In this case,

$$C_{max}(LMTF)/C^*_{max} \leq \begin{cases} \frac{5}{4} & \text{for } m = 2, \\ 2 - \frac{1}{m-1} & \text{for } m \geq 3. \end{cases} \qquad (\dagger)$$

Kafura and Shen also give a complicated (but polynomial-time) algorithm 2D for which

$$C_{max}(2D)/C^*_{max} \leq 2 - \frac{2}{m+1}. \qquad (\dagger)$$

Other results for this model may be found in [Kafura & Shen 1976].

4.3.5.4. $P|prec|C_{max}$

In the presence of precedence constraints it is somewhat unexpected [Graham 1966] that the $2 - \frac{1}{m}$ bound still holds, *i.e.*,

$$C_{max}(LS)/C^*_{max} \leq 2 - \frac{1}{m}.$$

Now, consider executing the set of jobs *twice*: the first time using processing times p_j, precedence constraints, m machines and an arbitrary priority list, the second time using processing times $p'_j \leq p_j$, weakened precedence

constraints, m' machines and a (possibly different) priority list. Then [Graham 1966]

$$C'_{max}(LS)/C_{max}(LS) \leq 1 + \frac{m-1}{m'}. \qquad (\dagger)$$

Even when critical path (CP) scheduling is used, examples exist [Graham -] for which

$$C_{max}(CP)/C^*_{max} = 2 - \frac{1}{m}.$$

It is known [Graham -] that unforced idleness (UI) has the following behavior:

$$C_{max}(LS)/C^*_{max}(UI) \leq 2 - \frac{1}{m}. \qquad (\dagger)$$

Let $C^*_{max}(pmtn)$ denote the optimal value of C_{max} if preemption is allowed. As in the case of UI, it is known [Graham -] that

$$C_{max}(LS)/C^*_{max}(pmtn) \leq 2 - \frac{1}{m}. \qquad (\dagger)$$

Liu [Liu 1972] shows that

$$C^*_{max}(UI)/C^*_{max}(pmtn) \leq 2 - \frac{2}{m+1}. \qquad (\dagger)$$

Relatively little is known in the way of approximation algorithms for the more special case $P|tree|C_{max}$. It is conjectured in [Denning & Scott Graham 1973] that

$$C_{max}(CP)/C^*_{max} \stackrel{?}{\leq} 2 - \frac{2}{m+1}.$$

If true this would be best possible as examples show. For the special case that the precedence constraints form an *intree*, Kaufman [Kaufman 1974] shows that

$$C_{max}(CP) \leq C^*_{max}(pmtn) + \max_j\{p_j\} - \left\lceil \frac{1}{m} \max_j\{p_j\} \right\rceil.$$

4.3.5.5. $Q|prec|C_{max}$

Liu and Liu [Liu & Liu 1974B] also consider the presence of precedence constraints in the case of uniform machines. They show that, when unforced idleness or preemption is allowed,

$$C_{max}(LS)/C^{*}_{max}(UI) \leq 1 + \max_i\{q_i\}/\min_i\{q_i\} - \max_i\{q_i\}/\sum_i q_i, \qquad (\dagger)$$

$$C_{max}(LS)/C^{*}_{max}(pmtn) \leq 1 + \max_i\{q_i\}/\min_i\{q_i\} - \max_i\{q_i\}/\sum_i q_i. \qquad (\dagger)$$

When all $q_i = 1$ this reduces to the earlier $2 - \frac{1}{m}$ bounds for these questions on identical machines.

Suppose that the jobs are executed twice: the first time using m machines of speeds $q_1,\ldots,q_m$, the second time using m' machines of speeds $q'_1,\ldots,q'_{m'}$. Then

$$C'_{max}(LS)/C_{max}(LS) \leq \max_i\{q_i\}/\min_i\{q'_i\} + \sum_i q_i/\sum_i q'_i - \max_i\{q_i\}/\sum_i q'_i. \qquad (\dagger)$$

Note that when all $q_i = 1$, this reduces to the previously mentioned bound of $1 + \frac{m-1}{m'}$.

We mention here two rather special results of Baer [Baer 1974]. He constructs an algorithm B based on the CG labelling algorithm which has the following behavior. For $Q2|tree|C_{max}$ with $q_2/q_1 = 3$,

$$C_{max}(B) \leq C^{*}_{max} + 1;$$

for $Q2|prec|C_{max}$ with $q_2/q_1 = 2$,

$$C_{max}(B)/C^{*}_{max} \leq \frac{6}{5}.$$

4.3.5.6. $P|res,prec|C_{max}$

The most general bound for $P|res,prec|C_{max}$ is given in [Garey & Graham 1975]. It states

$$C_{max}(LS)/C^{*}_{max} \leq m \qquad (\dagger)$$

and, in fact, examples with $s = 1$ are given which achieve this bound. Thus, the addition of even a single resource in the presence of precedence constraints can have a drastic effect on the worst-case behavior of an arbitrary priority list.

For $P|res|C_{max}$, it is shown in [Garey & Graham 1975] that for $m \geq 2$

$$C_{max}(LS)/C^{*}_{max} \leq \min\{\frac{m+1}{2},\ s + 2 - \frac{2s+1}{m}\}. \qquad (\dagger)$$

With the restriction that $m \geq n$, $s \geq 1$, this can be improved to

$$C_{max}(LS)/C^{*}_{max} \leq s + 1. \qquad (\dagger)$$

The techniques used to prove this inequality involve an interesting application of Ramsey theory, a branch of combinatorics.

4.4. Preemptive scheduling

4.4.1. $P|pmtn|\sum C_j$

A theorem of McNaughton [McNaughton 1959] states that for $P|pmtn|\sum w_jC_j$ there is no schedule with a finite number of preemptions which yields a smaller criterion value than an optimal nonpreemptive schedule. The finiteness restriction can be removed by appropriate application of results from open shop theory. It therefore follows that the procedure of Section 4.3.1 can be applied to solve $P|pmtn|\sum C_j$. It also follows that $P2|pmtn|\sum w_jC_j$ is NP-hard, since $P2||\sum w_jC_j$ is known to be NP-hard.

4.4.2. $Q|pmtn|\sum C_j$

McNaughton's theorem does not apply to uniform machines, as can be demonstrated by a simple counterexample. There is, however, a polynomial algorithm for $Q|pmtn|\sum C_j$.

One can show that there exists an optimal preemptive schedule in which $C_j \leq C_k$ if $p_j < p_k$ [Lawler & Labetoulle 1978]. Accordingly, first place the jobs in SPT order. Then obtain an optimal schedule by preemptively scheduling each successive job in the available time on the m machines so as to minimize its completion time [Gonzalez 1977]. This procedure can be implemented in $O(n \log n + mn)$ time and yields an optimal schedule with no more than $(m-1)(n-\frac{m}{2})$ preemptions. It has been extended to cover the case in which $\sum C_j$ is minimized subject to a common deadline for all jobs [Gonzalez 1977].

4.4.3. $R|pmtn|\sum C_j$

Very little is known about $R|pmtn|\sum C_j$. We conjecture that the problem is NP-hard. However, this remains one of the more vexing questions in the area of preemptive scheduling.

4.4.4. $P|pmtn,prec|C_{max}$

An obvious lower bound on the value of an optimal $P|pmtn|C_{max}$ schedule is given by

$$\max\{\max_j\{p_j\}, \frac{1}{m}\sum_j p_j\}.$$

A schedule meeting this bound can be constructed in O(n) time [McNaughton 1959]: just fill the machines successively, scheduling the jobs in any order and splitting a job whenever the above time bound is met. The number of preemptions occurring in this schedule is at most m-1. It is possible to design a class of problems for which this number is minimal, but the general problem of minimizing the number of preemptions is easily seen to be NP-hard.

In the case of precedence constraints, $P|pmtn,prec,p_j=1|C_{max}$ turns out to be NP-hard [Ullman 1976], but $P|pmtn,tree|C_{max}$ and $P2|pmtn,prec|C_{max}$ can be solved by a polynomial-time algorithm due to Muntz and Coffman [Muntz & Coffman 1969, 1970]. This is as follows.

Define $\ell_j(t)$ to be the level of a J_j wholly or partly unexecuted at time t. Suppose that at time t m' machines are available and that n' jobs are currently maximizing $\ell_j(t)$. If $m' < n'$, we assign m'/n' machines to each of the n' jobs, which implies that each of these jobs will be executed at speed m'/n'. If $m' \geq n'$, we assign one machine to each job, consider the jobs at the next highest level, and repeat. The machines are reassigned whenever a job is completed or threatens to be processed at a higher speed than another one at a currently higher level. Between each pair of successive reassignment points, jobs are finally rescheduled by means of McNaughton's algorithm for $P|pmtn|C_{max}$. The algorithm requires $O(n^2)$ time [Gonzalez & Johnson 1977].

Recently, Gonzalez and Johnson [Gonzalez & Johnson 1977] have developed a totally different algorithm that solves $P|pmtn,tree|C_{max}$ by starting at the roots rather than the leaves of the tree and determines priority by considering the total remaining processing time in subtrees rather than by looking at critical paths. The algorithm runs in O(n log m) time and introduces at most n-2 preemptions into the resulting optimal schedule.

Lam and Sethi [Lam & Sethi 1977], much in the same spirit as their work mentioned in Section 4.2.2, analyze the performance of the Muntz-Coffman (MC) algorithm for $P|pmtn,prec|C_{max}$. They show

$$C_{max}(MC)/C^*_{max} \leq 2 - \frac{2}{m} \quad (m \geq 2). \qquad (\dagger)$$

4.4.5. $Q|pmtn,prec|C_{max}$

Horvath, Lam and Sethi [Horvath *et al.* 1977] adapt the Muntz-Coffman algorithm to solve $Q|pmtn|C_{max}$ and $Q2|pmtn,prec|C_{max}$ in $O(mn^2)$ time. This results in an optimal schedule with no more than $(m-1)n^2$ preemptions.

A complicated, but computationally efficient, algorithm due to Gonzalez and Sahni [Gonzalez & Sahni 1978B] solves $Q|pmtn|C_{max}$ in $O(n)$ time, if the jobs are given in order of nonincreasing p_j and the machines in order of nondecreasing q_i. This procedure yields an optimal schedule with no more than $2(m-1)$ preemptions, which can be shown to be a tight bound.

The optimal value of C_{max} is given by

$$\max\{\max_{1\le k\le m-1}\{\sum_{j=1}^{k} p_j/\sum_{i=1}^{k} \frac{1}{q_i}\},\ \sum_{j=1}^{n} p_j/\sum_{i=1}^{m} \frac{1}{q_i}\},$$

where $p_1 \ge \ldots \ge p_n$ and $q_1 \le \ldots \le q_m$. This result generalizes the one given in Section 4.4.4.

The Gonzalez-Johnson algorithm for $P|pmtn,tree|C_{max}$ mentioned in the previous section can be adapted to the case $Q2|pmtn,tree|C_{max}$.

In [Horvath *et al.* 1977] it is shown that for $Q|pmtn,prec|C_{max}$, critical path scheduling has the bound

$$C_{max}(CP)/C^*_{max} \le (\frac{3m}{2})^{\frac{1}{2}}$$

and examples are given for which the bound $(\frac{m}{8})^{\frac{1}{2}}$ is approached arbitrarily closely.

4.4.6. $R|pmtn|C_{max}$

Many preemptive scheduling problems involving independent jobs on unrelated machines can be formulated as linear programming problems [Lawler & Labetoulle 1978]. For instance, solving $R|pmtn|C_{max}$ is equivalent to minimizing

$$C_{max}$$

subject to

$$\sum_{i=1}^{m} x_{ij}/p_{ij} = 1 \quad (j = 1,\ldots,n),$$

$$\sum_{i=1}^{m} x_{ij} \le C_{max} \quad (j = 1,\ldots,n),$$

$$\sum_{j=1}^{n} x_{ij} \leq C_{max} \quad (i = 1,\ldots,m),$$

$$x_{ij} \geq 0 \quad (i = 1,\ldots,m,\ j = 1,\ldots,n).$$

In this formulation x_{ij} represents the total time spent by J_j on M_i. Given a solution to the linear program, a feasible schedule can be constructed in polynomial time by applying the algorithm for $O|pmtn|C_{max}$, discussed in Section 5.2.2.

This procedure can be modified to yield an optimal schedule with no more than about $\frac{7}{2} m^2$ preemptions. It remains an open question as to whether $O(m^2)$ preemptions are necessary for an optimal preemptive schedule.

For fixed m, it seems to be possible to solve the linear program in linear time. Certainly, the special case $R2|pmtn|C_{max}$ can be solved in O(n) time [Gonzalez *et al.* 1978].

We note that a similar linear programming formulation can be given for the minimization of L_{max} [Lawler & Labetoulle 1978].

4.4.7. $P|pmtn,r_j|L_{max}$

$P|pmtn|L_{max}$ and $P|pmtn,r_j|C_{max}$ can be solved by a procedure due to Horn [Horn 1974]. The $O(n^2)$ running time has been reduced to O(mn) [Gonzalez & Johnson 1977].

More generally, the existence of a feasible preemptive schedule with given release dates and deadlines can be tested by means of a network flow model in $O(n^3)$ time [Horn 1974]. A binary search can then be conducted on the optimal value of L_{max}, with each trial value of L_{max} inducing deadlines which are checked for feasibility by means of the network computation. It can be shown that this yields an $O(n^3 \min\{n^2, \log n + \log \max_j\{p_j\}\})$ algorithm [Labetoulle *et al.* 1978].

4.4.8. $Q|pmtn,r_j|L_{max}$

In the case of uniform machines, the existence of a feasible preemptive schedule with given release dates and a common deadline can be tested in O(n log n + mn) time; the algorithm generates O(mn) preemptions in the worst case [Sahni & Cho 1977A]. More generally, $Q|pmtn,r_j|C_{max}$ and, by symmetry, $Q|pmtn|L_{max}$ are solvable in $O(m^2n+n^2)$ time; the number of preemptions generated is $O(n^2)$ [Sahni & Cho 1977B; Labetoulle *et al.* 1978].

The feasibility test mentioned in the previous section has been adapted to the case of two uniform machines [Bruno & Gonzalez 1976] and extended to a polynomial-time algorithm for $Q2|pmtn,r_j|L_{max}$ [Labetoulle *et al.* 1978].

It appears not unlikely that the Gonzalez-Johnson algorithm for $P|pmtn,tree|C_{max}$ and the above mentioned algorithm for $Q|pmtn,r_j|C_{max}$ allow a common generalization that will make $Q|pmtn,tree|C_{max}$ solvable in polynomial time.

5. OPEN SHOP, FLOW SHOP AND JOB SHOP PROBLEMS

5.1. Introduction

We now pass on to problems in which each job requires execution on more than one machine. Recall from Section 2.3 that in an *open shop* (denoted by O) the order in which a job passes through the machines is immaterial, whereas in a *flow shop* (F) each job has the same machine ordering $(M_1,\ldots,M_m)$ and in a *job shop* (J) possibly different machine orderings are specified for the jobs. We survey these problem classes in Sections 5.2, 5.3 and 5.4, respectively.

An obvious extension of this type of problem involves machines which can process more than one job at the same time. The resulting *resource constrained project scheduling* problems are extremely hard to solve. We refer to surveys by Davis [Davis 1966, 1973] that contain an extensive bibliography.

We shall be dealing exclusively with the C_{max} criterion. Other optimality criteria lead usually to NP-hard problems, even for m = 2 [Garey *et al.* 1976C; Lenstra *et al.* 1977]; a notable exception is $O2||\sum C_j$, which is open. Only a few enumerative algorithms for problems involving criteria other than C_{max} have been developed, *e.g.*, for $F2||\sum C_j$ [Ignall & Schrage 1965], $F||\sum w_j C_j$ [Townsend 1977A], $F||L_{max}$ [Townsend 1977B], and $J||\sum w_j T_j$ [Fisher 1973].

5.2. Open shop scheduling

5.2.1. *Nonpreemptive case*

The case $O2||C_{max}$ admits of an O(n) algorithm [Gonzalez & Sahni 1976]. A simplified exposition is given below.

For convenience, let $a_j = p_{1j}$, $b_j = p_{2j}$. Let $A = \{J_j | a_j \geq b_j\}$, $B = \{J_j | a_j < b_j\}$. Now choose J_r and J_ℓ to be any two distinct jobs (whether in A or B) such that

$$a_r \geq \max_{J_j \in A}\{b_j\},$$

$$b_\ell \geq \max_{J_j \in B}\{a_j\}.$$

Let $A' = A-\{J_r,J_\ell\}$, $B' = B-\{J_r,J_\ell\}$. We assert that it is possible to form feasible schedules for $B'\cup\{J_\ell\}$ and for $A'\cup\{J_r\}$ as indicated in Figure 5.1, the jobs in A' and B' being ordered arbitrarily. In each of these separate

schedules, there is no idle time on either machine, from the start of the first job on that machine to the completion of the last job on that machine.

Let $T_1 = \sum_j a_j$, $T_2 = \sum_j b_j$. Suppose $T_1 - a_\ell \geq T_2 - b_r$ (the case $T_1 - a_\ell < T_2 - b_r$ being symmetric). We then combine the two schedules as shown in Figure 5.2, pushing the jobs in $B' \cup \{J_\ell\}$ on M_2 to the right. Again, there is no idle time on either machine, from the start of the first job to the completion of the last job.

We finally propose to move the processing of J_r on M_2 to the first position on that machine. There are two cases to consider.

(1) $a_r \leq T_2 - b_r$. The resulting schedule is as in Figure 5.3. The length of the schedule is $\max\{T_1, T_2\}$.

(2) $a_r > T_2 - b_r$. The resulting schedule is as in Figure 5.4. The length of the schedule is $\max\{T_1, a_r + b_r\}$.

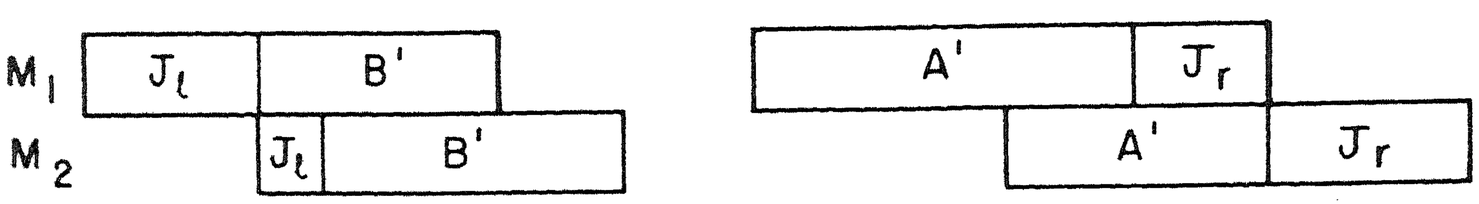


Figure 5.1

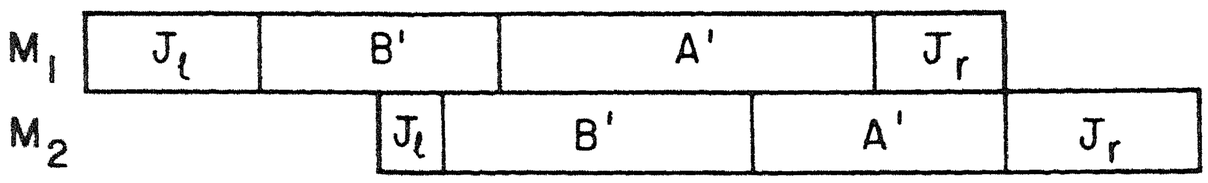


Figure 5.2

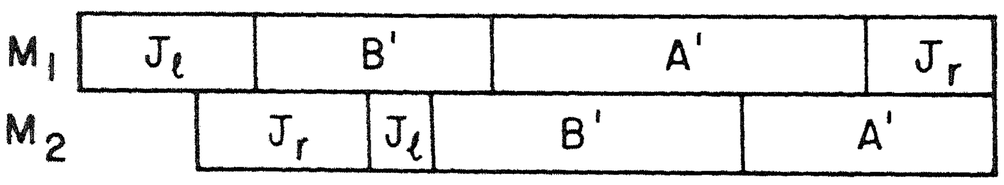


Figure 5.3

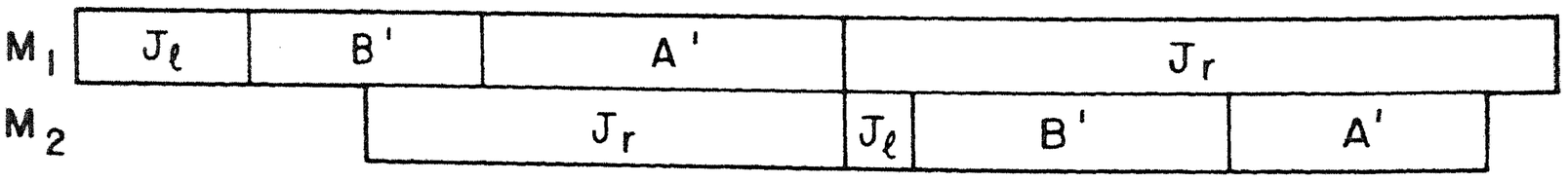


Figure 5.4

For any feasible schedule we obviously have that

$$C_{max} \geq \max\{T_1, T_2, \max_j\{a_j+b_j\}\}.$$

Since, in all cases, we have met this lower bound, it follows that the schedules constructed are optimal.

There is a little hope of finding polynomial-time algorithms for non-preemptive open shop problems more complicated than $O2||C_{max}$. The case $O3||C_{max}$ is binary NP-hard [Gonzalez & Sahni 1976] and $O2|r_j|C_{max}$, $O2|tree|C_{max}$ and $O||C_{max}$ are unary NP-hard [Lenstra -].

5.2.2. *Preemptive case*

The result on $O2||C_{max}$ presented in the previous section shows that there is no advantage to preemption for m = 2, and hence $O2|pmtn|C_{max}$ can be solved in O(n) time. More generally, $O|pmtn|C_{max}$ is solvable in polynomial time as well [Gonzalez & Sahni 1976]. We already had occasion to refer to this result in Section 4.4.6. An outline of the algorithm, adapted from [Lawler & Labetoulle 1978], follows below.

Let $P = (p_{ij})$ be the matrix of processing times and

$$C = \max\{\max_j\{\sum_i p_{ij}\}, \max_i\{\sum_j p_{ij}\}\}.$$

Call row i (column j) of P *tight* if $\sum_j p_{ij} = C$ ($\sum_i p_{ij} = C$), *slack* otherwise.

We clearly have $C^*_{max} \geq C$. It is possible to construct a feasible schedule for which $C_{max} = C$. Hence this schedule will be optimal.

Suppose we can find a subset S of strictly positive elements of P, with exactly one element of S in each tight row and in each tight column, and at most one element of S in each slack row and in each slack column. We shall call such a subset a *decrementing set*, and use it to construct a *partial schedule* of length δ, for some $\delta > 0$. The constraints on the choice of δ are as follows.

(1) If $p_{ij} \in S$ and either row i or column j is tight, then $\delta \leq p_{ij}$.

(2) If $p_{ij} \in S$ and row i (column j) is slack, then $\delta \leq p_{ij} + C - \sum_k p_{ik}$ ($\delta \leq p_{ij} + C - \sum_k p_{kj}$).

(3) If row i (column j) contains no element in S (and is therefore necessarily slack), then $\delta \leq C - \sum_k p_{ik}$ ($\delta \leq C - \sum_k p_{kj}$).

For a given decrementing set S, let δ be the maximum subject to (1),(2),(3).

Then the partial schedule constructed is such that for each $p_{ij} \in S$, M_i processes J_j for $\min\{p_{ij},\delta\}$ units of time.

We then obtain the matrix P' from P be replacing each $p_{ij} \in S$ by $\max\{0,p_{ij}-\delta\}$, and repeat the procedure until after a finite number of times P' = (0). Joining together the partial schedules obtained for successive decrementing sets then yields an optimal preemptive schedule for P.

By suitably embedding P in a doubly stochastic matrix and appealing to the Birkhoff-Von Neumann theorem, it can be shown that a decrementing set can be found by solving a linear assignment problem; see [Lawler & Labetoulle 1978] for details.

Other network formulations of the problem are possible. An analysis of various possible computations reveals that $O|pmtn|C_{max}$ can be solved in $O(r + \min\{m^4,n^4,r^2\})$ time, where r is the number of nonzero elements in P [Gonzalez 1976].

5.3. Flow shop scheduling

5.3.1. $F2|\beta|C_{max}$, $F3|\beta|C_{max}$

A fundamental algorithm for solving $F2||C_{max}$ is due to Johnson [Johnson 1954]. He shows that there exists an optimal schedule in which J_j precedes J_k if $\min\{p_{1j},p_{2k}\} \leq \min\{p_{2j},p_{1k}\}$. It follows that the problem can be solved in O(n log n) time: arrange first the jobs with $p_{1j} \leq p_{2j}$ in order of nondecreasing p_{1j} and subsequently the remaining jobs in order of nonincreasing p_{2j}.

Some special cases involve *start lags* ℓ_{1j} and *stop lags* ℓ_{2j} for J_j, that represent minimum time intervals between starting times on M_1 and M_2 and between completion times on M_1 and M_2, respectively [Mitten 1958; Johnson 1958; Nabeshima 1963; Szwarc 1968]. Defining $\ell_j = \min\{\ell_{1j}-p_{1j},\ell_{2j}-p_{2j}\}$ and applying Johnson's algorithm to processing times $(p_{1j}+\ell_j,p_{2j}+\ell_j)$ will produce an optimal *permutation schedule*, *i.e.*, one with identical processing orders on all machines [Rinnooy Kan 1976]. If we drop the latter restriction, the problem is unary NP-hard [Lenstra -].

Similarly, some $F3||C_{max}$ problems can be solved by applying Johnson's algorithm to processing times $(p_{1j}+p_{2j},p_{2j}+p_{3j})$, *e.g.*, if there exists a $\theta \in [0,1]$ such that $\theta p_{1j}+(1-\theta)p_{3j} \geq p_{2k}$ for all (j,k) [Johnson 1954; Burns & Rooker 1976] or if M_2 can process any number of jobs at the same time [Conway *et al.* 1967]. We refer to [Monma 1977] for further generalizations.

The general $F3||C_{max}$ problem, however, is unary NP-hard, and the same applies to $F2|r_j|C_{max}$ and $F2|tree|C_{max}$ [Garey *et al.* 1976C; Lenstra *et al.* 1977].

It should be noted that an interpretation of precedence constraints which differs from our definition is possible. If $J_j <' J_k$ only means that O_{ij} should precede O_{ik} for i = 1,2, then $F2|tree'|C_{max}$ can be solved in O(n log n) time [Sidney 1977]. In fact, Sidney's algorithm applies even to series-parallel precedence constraints. The arguments used to establish this result are very similar to those referred to in Section 3.3.1 and apply to a larger class of scheduling problems [Monma & Sidney 1977]. The general case $F2|prec'|C_{max}$ is unary NP-hard [Monma 1978].

Gonzalez and Sahni [Gonzalez & Sahni 1978A] consider the case of preemptive flow shop scheduling. They show that preemptions on M_1 and M_m can be removed without increasing C_{max}. Hence, Johnson's algorithm solves $F2|pmtn|C_{max}$ as well. $F3|pmtn|C_{max}$ turns out to be unary NP-hard.

5.3.2. $F||C_{max}$

As a general result, we note that there exists an optimal flow shop schedule with the same processing order on M_1 and M_2 and the same processing order on M_{m-1} and M_m [Conway *et al.* 1967]. It is, however, not difficult to construct a 4-machine example in which a job "passes" another one between M_2 and M_3 in the optimal schedule. Nevertheless, it has become tradition in the literature to assume identical processing orders on all machines, so that in effect only the best permutation schedule has to be determined.

Except for some rather simple worst-case results for heuristics, obtained by Gonzalez and Sahni [Gonzalez & Sahni 1978A], that are to be mentioned in Section 5.4.2, all research in this area has focused on enumerative methods.

The usual enumeration scheme is to assign jobs to the ℓ-th position in the schedule at the ℓ-th level of the search tree. Thus, at a node at that level a partial schedule $(J_{\sigma(1)},\ldots,J_{\sigma(\ell)})$ has been formed and the jobs with index set $S = \{1,\ldots,n\} - \{\sigma(1),\ldots,\sigma(\ell)\}$ are candidates for the $(\ell+1)$-st position. One then needs to find a lower bound on the value of all possible completions of the partial schedule. It turns out that almost all lower bounds developed so far are generated by the following bounding scheme [Lageweg *et al.* 1978B].

Let us relax the capacity constraint that each machine can process at

most one job at a time, for all machines but at most two, say, M_u and M_v $(1 \le u \le v \le m)$. We then obtain a problem of scheduling $\{J_j | j \in S\}$ on five machines $N_{\cdot u}, M_u, N_{uv}, M_v, N_{v\cdot}$ in that order, which is specified as follows. Let $C(\sigma,i)$ denote the completion time of $J_{\sigma(\ell)}$ on M_i. $N_{\cdot u}$, N_{uv} and $N_{v\cdot}$ have infinite capacity; the processing times on these machines are defined by

$$q_{\cdot uj} = \max_{1 \le i \le u}\{C(\sigma,i) + \sum_{h=i}^{u-1} p_{hj}\},$$

$$q_{uvj} = \sum_{h=u+1}^{v-1} p_{hj},$$

$$q_{v\cdot j} = \sum_{h=v+1}^{m} p_{hj}.$$

M_u and M_v have capacity 1 and processing times p_{uj} and p_{vj}, respectively. Note that we can interpret $N_{\cdot u}$ as yielding release dates $q_{\cdot uj}$ on M_u and $N_{v\cdot}$ as setting due dates $-q_{v\cdot j}$ on M_v, with respect to which L_{max} is to be minimized.

Any of the machines $N_{\cdot u}, N_{uv}, N_{v\cdot}$ can be removed from this problem by underestimating its contribution to the lower bound to be the minimum processing time on that machine. Valid lower bounds are obtained by adding these contributions to the optimal solution value of the remaining problem.

For the case that $u = v$, removing $N_{\cdot u}$ and $N_{u\cdot}$ from the problem produces the *machine-based bound* used in [Ignall & Schrage 1965; McMahon 1971]:

$$\max_{1 \le u \le m}\{\min_{j \in S}\{q_{\cdot uj}\} + \sum_{j \in S} p_{uj} + \min_{j \in S}\{q_{u\cdot j}\}\}.$$

Removing only $N_{\cdot u}$ results in a $1||L_{max}$ problem on M_u, which can be solved by Jackson's rule (Section 3.2) and provides a slightly stronger bound.

If $u \neq v$, removal of $N_{\cdot u}$, N_{uv} and $N_{v\cdot}$ yields an $F2||C_{max}$ problem, to be solved by Johnson's algorithm (Section 5.3.1). As pointed out in that section, solution in polynomial time remains possible if N_{uv} is taken fully into account; the resulting bound dominates the *job-based bound* proposed in [McMahon 1971] and is the best one currently available.

All other variations on this theme (*e.g.*, taking $u = v$ and considering the resulting $1|r_j|L_{max}$ problem) would involve the solution of NP-hard problems. The development of fast algorithms or strong lower bounds for these problems thus emerges as a possibly fruitful research area.

The computational performance of branch-and-bound algorithms for $F||C_{max}$ might be improved by the use of *elimination criteria*. Particular attention has been paid to conditions under which all completions of $(J_{\sigma(1)},\ldots,J_{\sigma(\ell)},J_j)$ can be eliminated because a schedule at least as good

exists among the completions of $(J_{\sigma(1)},\ldots,J_{\sigma(\ell)},J_k,J_j)$. If all information obtainable from the processing times of the other jobs is disregarded, the strongest condition under which this is allowed is as follows. Defining $\Delta_i = C(\sigma kj,i) - C(\sigma j,i)$, we can exclude J_j for the ℓ-th position if

$$\max\{\Delta_{i-1},\Delta_i\} \leq p_{ij} \quad (i = 2,\ldots,m)$$

[McMahon 1969; Szwarc 1971, 1973]. Inclusion of these and similar dominance rules can be very helpful from a computational point of view, depending on the lower bound used [Lageweg *et al.* 1978B]. It may be worthwhile to consider further extensions that, for instance, involve the processing times of the unscheduled jobs [Gupta & Reddi 1978; Szwarc 1978].

5.3.3. *No wait in process*

In a variation on the flow shop problem, each job, once started, has to be processed without interruption until it is completed. This *no wait* constraint may arise out of certain job characteristics (*e.g.*, the "hot ingot" problem in which metal has to be processed at continuously high temperature) or out of the unavailability of intermediate storage in between machines.

The resulting $F|$*no wait*$|C_{max}$ problem can be formulated as a *traveling salesman* problem with cities $0,1,\ldots,n$ and intercity distances

$$c_{jk} = \max_{1\leq i\leq m}\{\textstyle\sum_{h=1}^{i} p_{hj} - \sum_{h=1}^{i-1} p_{hk}\} \quad (j,k = 0,1,\ldots,n),$$

where $p_{i0} = 0$ $(i = 1,\ldots,m)$ [Piehler 1960; Reddi & Ramamoorthy 1972; Wismer 1972]. We refer to [Lenstra & Rinnooy Kan 1975] for an extension of this formulation to certain job shop systems and to [Van Deman & Baker 1974] for a branch-and-bound approach to $F|$*no wait*$|\sum C_j$.

For the case $F2|$*no wait*$|C_{max}$, the traveling salesman problem assumes a special structure and the results from [Gilmore & Gomory 1964] can be applied to yield an $O(n^2)$ algorithm [Reddi & Ramamoorthy 1972]. Both $F|$*no wait*$|C_{max}$ and $F|$*no wait*$|\sum C_j$ are unary NP-hard [Lenstra *et al.* 1977], and the same is true for $O2|$*no wait*$|C_{max}$ and $J2|$*no wait*$|C_{max}$ [Sahni & Cho 1977C]. In spite of challenging prizes awarded for their solution [Lenstra *et al.* 1977], $F3|$*no wait*$|C_{max}$ and $F2|$*no wait*$|\sum C_j$ are still open.

The *no wait* constraint may lengthen the optimal flow shop schedule considerably. It can be shown [Lenstra -] that

$$C^*_{max}(\textit{no wait})/C^*_{max} < m \quad \text{for } m \geq 2. \qquad (\dagger)$$

5.4. Job shop scheduling

5.4.1. $J2|\beta|C_{max}$, $J3|\beta|C_{max}$

A simple extension of Johnson's algorithm for $F2||C_{max}$ allows solution of $J2|m_j \leq 2|C_{max}$ in O(n log n) time [Jackson 1956]. Let J_i be the set of jobs with operations on M_i only (i = 1,2) and J_{hi} the set of jobs that go from M_h to M_i (hi = 12,21). Order the latter two sets by means of Johnson's algorithm and the former two sets arbitrarily. One then obtains an optimal schedule by executing the jobs on M_1 in the order (J_{12},J_1,J_{21}) and on M_2 in the order (J_{21},J_2,J_{12}).

This, however, is probably as far as we can get. Unary NP-hardness of $J2||C_{max}$ results as soon as we allow one job to have more than two operations [Garey *et al.* 1976C; Lenstra *et al.* 1977]. In fact, $J2|1 \leq p_{ij} \leq 2|C_{max}$ and $J3|p_{ij}=1|C_{max}$ are already NP-hard [Lenstra & Rinnooy Kan 1978B].

5.4.2. $J||C_{max}$

The general job shop problem is extremely hard to solve optimally. An indication of this is given by the fact that a 10-job 10-machine problem, formulated in 1963 [Muth & Thompson 1963], still has not been solved.

A convenient problem representation is provided by the *disjunctive graph* model, introduced by Roy and Sussmann [Roy & Sussmann 1964]. Assume each operation O_{ij} being renumbered as O_u with $u = \sum_{k=1}^{j-1} m_k + i$ and add two fictitious initial and final operations O_0 and O_* with $p_0 = p_* = 0$. The disjunctive graph is then defined as follows. There is a vertex u with weight p_u corresponding to each operation O_u. The directed *conjective arcs* link the consecutive operations of each job, and link O_0 to all first operations and all last operations to O_*. A pair of directed *disjunctive arcs* connects every two operations that have to be executed on the same machine. A feasible schedule corresponds to the selection of one disjunctive arc of every such pair, granting precedence of one operation over the other on their common machine, in such a way that the resulting directed graph is acyclic. The value of the schedule is given by the weight of the maximum weight path from 0 to *. We refer to Figures 5.5 and 5.6 for examples.

At a typical stage of any enumerative algorithm, a certain subset D of disjunctive arcs will have been selected. We consider the directed graph obtained by removing all other disjunctive arcs. Let the maximum weights of

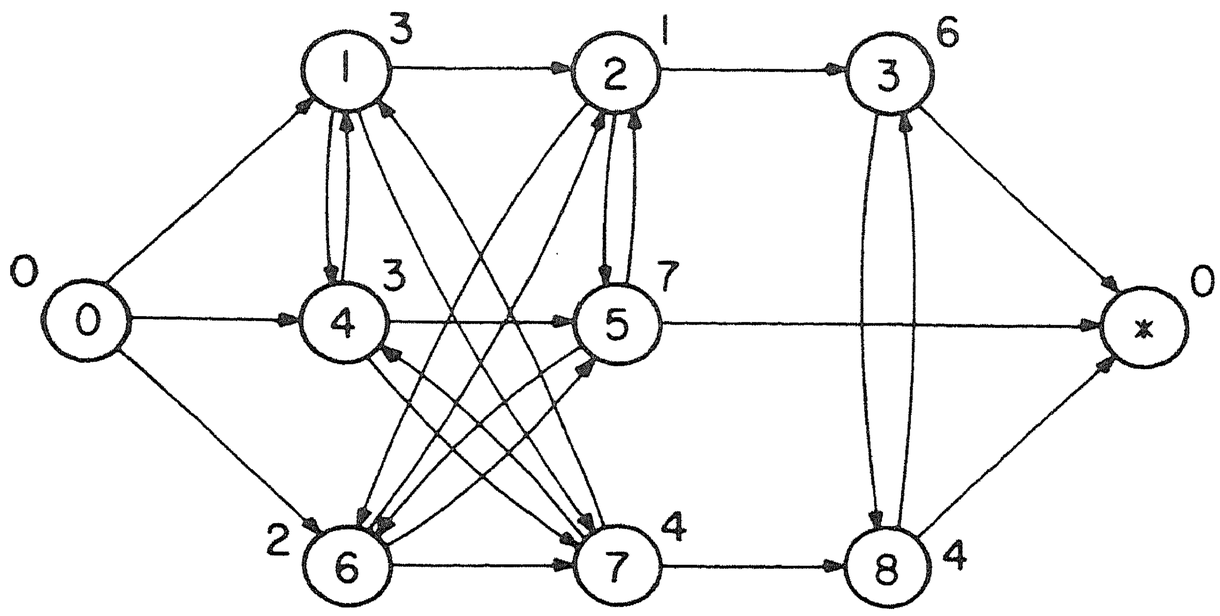


Figure 5.5 Job shop problem, represented as a disjunctive graph.

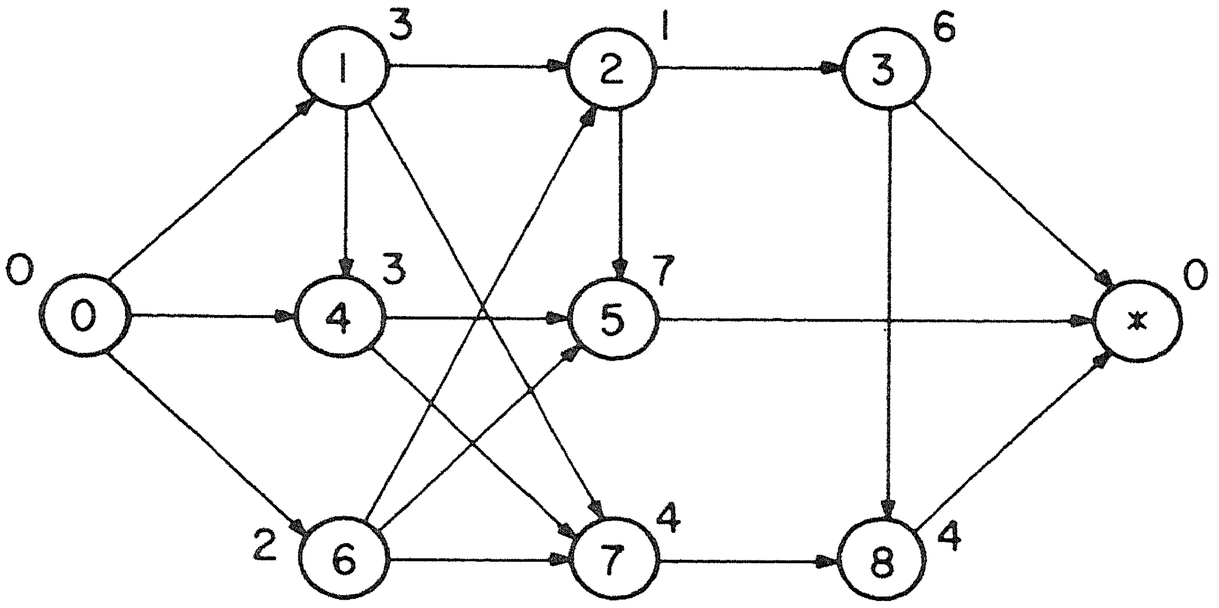


Figure 5.6 Job shop problem, represented as an acyclic directed graph.

paths from 0 to u and from u to *, excluding p_u, be denoted by r_u and q_u, respectively. In particular, r_* is an obvious lower bound on the value of any feasible schedule obtainable from the current graph [Charlton & Death 1970]. We can get a far better bound in a manner very similar to the development of flow shop bounds in Section 5.3.2 [Lageweg *et al.* 1977].

Let us relax the capacity constraints for all machines except M_i. We then obtain a problem of scheduling the operations O_u on M_i with release dates r_u, processing times p_u, due dates $-q_u$ and precedence constraints defined by the directed graph, so as to minimize maximum lateness. As pointed out in Section 3.2, this $1|prec,r_j|L_{max}$ problem is NP-hard, but there exist fast enumerative methods for its solution on each M_i. Again, all lower bounds proposed in the literature appear as special cases of the above one

by underestimating the contribution of r_u, q_u or both, by ignoring the precedence constraints, or by restricting the set of machines over which maximization is to take place.

The currently best job shop algorithm [McMahon & Florian 1975] involves the $1|r_j|L_{max}$ bound combined with the enumeration of *active schedules*. Starting from O_0, we consider at each stage the subset S of operations all of whose predecessors have been scheduled and calculate their earliest possible completion times r_u+p_u. It can be shown [Giffler & Thompson 1960] that it is sufficient to consider only a machine on which the minimum value of r_u+p_u is achieved and to branch by successively scheduling next on that machine all O_v for which $r_v < \min_{O_u \in S}\{r_u+p_u\}$. In this scheme, several disjunctive arcs are added to D at each stage. An alternative approach whereby at each stage one disjunctive arc of some *crucial* pair is selected leads to a computationally inferior approach [Lageweg *et al.* 1977].

The applicability of Lagrangean techniques to obtain stronger lower bounds is the subject of ongoing research. Either the precedence constraints fixing the machine orders for the jobs or the capacity constraints of the machines can be multiplied by a Lagrangean variable and added to the objective function. For fixed values of the multipliers, the resulting problems can be solved in (pseudo)polynomial time. Computational experiments will have to reveal if this approach, combined with subgradient optimization or another suitable technique, will lead to any substantially better job shop algorithm.

As far as approximation algorithms are concerned, a considerable effort has been invested in the empirical testing of various priority rules [Gere 1966; Conway *et al.* 1967; Day & Hottenstein 1970; Panwalkar & Iskander 1977]. No rule appears to be consistently better than any other and in practical situations one would be well advised to exploit any special structure that the problem at hand has to offer.

Not much has been done in the way of worst-case analysis of approximation algorithms for flow shop and job shop problems. Gonzalez and Sahni [Gonzalez & Sahni 1978A] show that for any active flow shop or job shop schedule (AS)

$$C_{max}(AS)/C^*_{max} \leq m. \qquad (\dagger)$$

This bound is tight even for LPT schedules, in which the jobs are ordered

according to nonincreasing sums of processing times. They give an O(mn log n) algorithm H for $F||C_{max}$ based on Johnson's algorithm for $F2||C_{max}$ with

$$C_{max}(H)/C^*_{max} \leq \left\lceil \frac{m}{2} \right\rceil.$$

With SPT defined similarly as LPT, it is also shown that for $F||\sum C_j$ and $J||\sum C_j$

$$\sum C_j(AS)/\sum C^*_j \leq n, \qquad (\dagger)$$

$$\sum C_j(SPT)/\sum C^*_j \leq m. \qquad (\dagger)$$

It thus appears that, in general, the obvious algorithms can deviate quite substantially from the optimum for this class of problems.

6. CONCLUDING REMARKS

If one thing emerges from the preceding survey, it is the amazing success of complexity theory as a means of differentiating between easy and hard problems. Within the very detailed problem classification developed especially for this purpose, surprisingly few open problems remain. For an extensive class of scheduling problems, a computer program has been developed that classifies these problems according to their computational complexity [Lageweg *et al.* 1978A]. It employs elementary reductions such as those defined in Section 2.7 in order to deduce the consequences of the development of a new polynomial-time algorithm or a new NP-hardness proof.

As far as polynomial-time algorithms are concerned, the most impressive recent advances have occurred in the area of parallel machine scheduling and are due to researchers with a computer science background, recognizable as such by their use of terms like *tasks* and *processors* rather than *jobs* and *machines*. Single machine, flow shop and job shop scheduling has been traditionally the domain of operations researchers. Here, an analytical approach to the performance of approximation algorithms is badly needed, although for any practical problem it probably will remain true that a successful heuristic will have to exploit whatever special structure and properties the problem at hand may have.

Thus, the area of deterministic scheduling theory appears as one of the more fruitful interfaces between computer science and operations research. Much progress has been made and more can be expected in the near future.

Note. The last three authors are currently engaged in writing a book on scheduling problems and would very much appreciate being informed about new algorithmic and complexity results in this area.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the useful comments from M.L. Fisher. The research by the last three authors was supported by NSF grant MCS76-17605.

BIBLIOGRAPHY

D. ADOLPHSON (1977) Single machine job sequencing with precedence constraints. *SIAM J. Comput.* 6,40-54.

D. ADOLPHSON, T.C. HU (1973) Optimal linear ordering. *SIAM J. Appl. Math.* 25,403-423.

A.V. AHO, M.R. GAREY, J.D. ULLMAN (1972) The transitive reduction of a directed graph. *SIAM J. Comput.* 1,131-137.

A.V. AHO, J.E. HOPCROFT, J.D. ULLMAN (1974) *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

G. AUSIELLO, A. D'ATRI (1977) On the structure of combinatorial problems and structure preserving reductions. In: A. SALOMAA, M. STEINBY (eds.) (1977) *Automata, Languages and Programming*. Lecture Notes in Computer Science 52, Springer, Berlin, 45-60.

J.L. BAER (1974) Optimal scheduling on two processors with different speeds. In: E. GELENBE, R. MAHL (eds.) (1974) *Computer Architectures and Networks*. North-Holland, Amsterdam, 27-45.

K.R. BAKER (1974) *Introduction to Sequencing and Scheduling*. Wiley, New York.

K.R. BAKER, A.G. MERTEN (1973) Scheduling with parallel processors and linear delay costs. *Naval Res. Logist. Quart.* 20,793-804.

K.R. BAKER, L.E. SCHRAGE (1978) Finding an optimal sequence by dynamic programming: an extension to precedence-related tasks. *Operations Res.* 26,111-120.

K.R. BAKER, Z.-S. SU (1974) Sequencing with due-dates and early start times to minimize maximum tardiness. *Naval Res. Logist. Quart.* 21,171-176.

M.S. BAKSHI, S.R. ARORA (1969) The sequencing problem. *Management Sci.* 16,B247-263.

L. BARBOSA (-) Personal communication.

J.W. BARNES, J.J. BRENNAN (1977) An improved algorithm for scheduling jobs on identical machines. *AIIE Trans.* 9,25-31.

J. BEARDWOOD, J.H. HALTON, J.M. HAMMERSLEY (1959) The shortest path through many points. *Proc. Cambridge Philos. Soc.* 55,299-327.

R.E. BELLMAN (1962) Dynamic programming treatment of the travelling salesman problem. *J. Assoc. Comput. Mach.* 9,61-63.

R.E. BELLMAN, S.E. DREYFUS (1962) *Applied Dynamic Programming*. Princeton University Press, Princeton, N.J.

M. BEST, P. VAN EMDE BOAS, H.W. LENSTRA JR. (1974) A sharpened version of the Aanderaa-Rosenberg conjecture. Report ZW 30, Mathematisch Centrum, Amsterdam.

M. BLUM, R.W. FLOYD, V. PRATT, R.L. RIVEST, R.E. TARJAN (1973) Time bounds for selection. *J. Comput. System Sci.* 7,448-461.

K.S. BOOTH (1976) Problems polynomially equivalent to graph isomorphism. In: J.F. TRAUB (ed.) (1976) *Algorithms and Complexity: New Directions and Recent Results*. Academic Press, New York, 435.

P. BRATLEY, M. FLORIAN, P. ROBILLARD (1975) Scheduling with earliest start and due date constraints on multiple machines. *Naval Res. Logist. Quart.* 22,165-173.

M.R. BROWN (1978) Implementation and analysis of binomial queue algorithms. *SIAM J. Comput.*, to appear.

P. BRUCKER, M.R. GAREY, D.S. JOHNSON (1977) Scheduling equal-length tasks under tree-like precedence constraints to minimize maximum lateness. *Math. Operations Res.* 2,275-284.

J. BRUNO, E.G. COFFMAN, JR., R. SETHI (1974) Scheduling independent tasks to reduce mean finishing time. *Comm. ACM* 17,382-387.

J. BRUNO, T. GONZALEZ (1976) Scheduling independent tasks with release dates and due dates on parallel machines. Technical Report 213, Computer Science Department, Pennsylvania State University.

F. BURNS, J. ROOKER (1976) 3×n Flow-shops with convex external stage time dominance. Unpublished manuscript.

A.K. CHANDRA, L.J. STOCKMEYER (1976) Alternation. *Proc. 17th Annual IEEE Symp. Foundations of Computer Science,* 98-108.

A.K. CHANDRA, C.K. WONG (1975) Worst-case analysis of a placement algorithm related to storage allocation. *SIAM J. Comput.* 4,249-263.

J.M. CHARLTON, C.C. DEATH (1970) A generalized machine scheduling algorithm. *Operational Res. Quart.* 21,127-134.

N.-F. CHEN (1975) An analysis of scheduling algorithms in multiprocessing computing systems. Technical Report UIUCDCS-R-75-724, Department of Computer Science, University of Illinois at Urbana-Champaign.

N.-F. CHEN, C.L. LIU (1975) On a class of scheduling algorithms for multiprocessors computing systems. In: T.-Y. FENG (ed.) (1975) *Parallel Processing*. Lecture Notes in Computer Science 24, Springer, Berlin, 1-16.

N. CHRISTOFIDES (1978) Worst-case analysis of a new heuristic for the traveling salesman problem. *Math. Programming*, to appear.

A. COBHAM (1964) The intrinsic computational difficulty of functions. *Proc. 1964 Internat. Congress Logic, Methodology and Philosophy of Science*, North-Holland, Amsterdam, 24-30.

E.G. COFFMAN, JR. (ed.) (1976) *Computer and Job-Shop Scheduling Theory*. Wiley, New York.

E.G. COFFMAN, JR., M.R. GAREY, D.S. JOHNSON (1978) An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.*, to appear.

E.G. COFFMAN, JR., R.L. GRAHAM (1972) Optimal scheduling for two-processor systems. *Acta Informat.* 1,200-213.

R.W. CONWAY, W.L. MAXWELL, L.W. MILLER (1967) *Theory of Scheduling*. Addison-Wesley, Reading, Mass.

S.A. COOK (1971) The complexity of theorem-proving procedures. *Proc. 3rd Annual ACM Symp. Theory of Computing*, 151-158.

S.A. COOK, R.A. RECKHOW (1973) Time bounded random access machines. *J. Comput. System Sci.* 7,354-375.

G. CORNUEJOLS, M.L. FISHER, G.L. NEMHAUSER (1977) Location of bank accounts to optimize float: an analytic study of exact and approximate algorithms. *Management Sci.* 23,789-810.

G. CORNUEJOLS, G.L. NEMHAUSER (1978) Tight bounds for Christofides' traveling salesman heuristic. *Math. Programming* 14,116-121.

G.B. DANTZIG (-) Personal communication.

E.W. DAVIS (1966) Resource allocation in project network models - a survey. *J. Indust. Engrg.* 17,177-188.

E.W. DAVIS (1973) Project scheduling under resource constraints - historical review and categorization of procedures. *AIIE Trans.* 5,297-313.

J. DAY, M.P. HOTTENSTEIN (1970) Review of scheduling research. *Naval Res. Logist. Quart.* 17,11-39.

P.J. DENNING, G. SCOTT GRAHAM (1973) A note on subexpression ordering in the execution of arithmetic expressions. *Comm. ACM* 16,700-702.

E.A. DINIC (1970) Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.* 11,1277-1280.

W.L. EASTMAN, S. EVEN, I.M. ISAACS (1964) Bounds for the optimal scheduling of *n* jobs on *m* processors. *Management Sci.* 11,268-279.

J. EDMONDS (1965A) Paths, trees, and flowers. *Canad. J. Math.* 17,449-467.

J. EDMONDS (1965B) The Chinese postman's problem. *Operations Res.* 13 Suppl. 1,B73.

J. EDMONDS, E.L. JOHNSON (1973) Matching, Euler tours and the Chinese postman. *Math. Programming* 5,88-124.

C.C. ELGOT, A. ROBINSON (1964) Random access stored program machines. *J. Assoc. Comput. Mach.* 11,365-399.

S.E. ELMAGHRABY, S.H. PARK (1974) Scheduling jobs on a number of identical machines. *AIIE Trans.* 6,1-12.

S. EVEN (1976) The max flow algorithm of Dinic and Karzanov: an exposition. Department of Computer Science, Technion, Haifa.

S. EVEN, R.E. TARJAN (1976) A combinatorial problem which is complete in polynomial space. *J. Assoc. Comput. Mach.* 23,710-719.

M.L. FISHER (1973) Optimal solution of scheduling problems using Lagrange multipliers, part I. *Operations Res.* 21,1114-1127.

M.L. FISHER (1976A) A dual algorithm for the one-machine scheduling problem. *Math. Programming* 11,229-251.

M.L. FISHER (1976B) Personal communication.

M.L. FISHER, G.L. NEMHAUSER, L.A. WOLSEY (1978A) An analysis of approximations for finding a maximum weight Hamiltonian circuit. *Operations Res.*, to appear.

M.L. FISHER, G.L. NEMHAUSER, L.A. WOLSEY (1978B) An analysis of approximations for maximizing submodular set functions - II. *Math. Programming*, to appear.

G.N. FREDERICKSON, M.S. HECHT, C.E. KIM (1976) Approximation algorithms for some routing problems. *Proc. 17th Annual IEEE Symp. Foundations of Computer Science*, 216-227.

M. FUJII, T. KASAMI, K. NINOMIYA (1969,1971) Optimal sequencing of two equivalent processors. *SIAM J. Appl. Math.* 17,784-789; Erratum. 20,141.

H.N. GABOW (1976) An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *J. Assoc. Comput. Mach.* 23,221-234.

M.R. GAREY (-) Unpublished.

M.R. GAREY, R.L. GRAHAM (1975) Bounds for multiprocessor scheduling with resource constraints. *SIAM J. Comput.* 4,187-200.

M.R. GAREY, R.L. GRAHAM, D.S. JOHNSON (1976A) Some NP-complete geometric problems. *Proc. 8th Annual ACM Symp. Theory of Computing*, 10-22.

M.R. GAREY, R.L. GRAHAM, D.S. JOHNSON (1978) Performance guarantees for

scheduling algorithms. *Operations Res.* 26,3-21.

M.R. GAREY, R.L. GRAHAM, D.S. JOHNSON (-) Unpublished.

M.R. GAREY, R.L. GRAHAM, D.S. JOHNSON, A.C.-C. YAO (1976B) Resource constrained scheduling as generalized bin packing. *J. Combinatorial Theory Ser. A* 21,257-298.

M.R. GAREY, D.S. JOHNSON (1975) Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. Computing* 4,397-411.

M.R. GAREY, D.S. JOHNSON (1976A) The complexity of near-optimal graph coloring. *J. Assoc. Comput. Mach.* 23,43-49.

M.R. GAREY, D.S. JOHNSON (1976B) Scheduling tasks with nonuniform deadlines on two processors. *J. Assoc. Comput. Mach.* 23,461-467.

M.R. GAREY, D.S. JOHNSON (1976C) Approximation algorithms for combinatorial problems: an annoted bibliography. In: J.F. TRAUB (ed.) (1976) *Algorithms and Complexity: New Directions and Recent Results.* Academic Press, New York, 41-52.

M.R. GAREY, D.S. JOHNSON (1977) Two-processor scheduling with start-times and deadlines. *SIAM J. Comput.* 6,416-426.

M.R. GAREY, D.S. JOHNSON (1978A) *Computers and Intractability: a Guide to the Theory of NP-completeness.* Freeman, San Francisco, to appear.

M.R. GAREY, D.S. JOHNSON (1978B) "Strong" NP-completeness results: motivation, examples and implications. *J. Assoc. Comput. Mach.*, to appear.

M.R. GAREY, D.S. JOHNSON, R. SETHI (1976C) The complexity of flowshop and jobshop scheduling. *Math. Operations Res.* 1,117-129.

M.R. GAREY, D.S. JOHNSON, L. STOCKMEYER (1976D) Some simplified *NP*-complete graph problems. *Theoret. Comput. Sci.* 1,237-267.

M.R. GAREY, D.S. JOHNSON, R.E. TARJAN (1976E) The planar Hamiltonian circuit problem is NP-complete. *SIAM J. Comput.* 5,704-714.

L. GELDERS, P.R. KLEINDORFER (1974) Coordinating aggregate and detailed scheduling decisions in the one-machine job shop: part I. Theory. *Operations Res.* 22,46-60.

L. GELDERS, P.R. KLEINDORFER (1975) Coordinating aggregate and detailed scheduling in the one-machine job shop: II - computation and structure. *Operations Res.* 23,312-324.

A.M. GEOFFRION (1977A) Objective function approximations in mathematical programming. *Math. Programming* 13,23-37.

A.M. GEOFFRION (1977B) A priori error bounds for procurement commodity aggregation in logistics planning models. *Naval Res. Logist. Quart.* 24,201-212.

W.S. GERE (1966) Heuristics in job shop scheduling. *Management Sci.* 13,167-190.

B. GIFFLER, G.L. THOMPSON (1960) Algorithms for solving production-scheduling problems. *Operations Res.* 8,487-503.

P.C. GILMORE, R.E. GOMORY (1964) Sequencing a one-state variable machine: a solvable case of the traveling salesman problem. *Operations Res.* 12,655-679.

T. GONZALEZ (1976) A note on open shop preemptive schedules. Technical Report 214, Computer Science Department, Pennsylvania State University.

T. GONZALEZ (1977) Optimal mean finish time preemptive schedules. Technical Report 220, Computer Science Department, Pennsylvania State University.

T. GONZALEZ, O.H. IBARRA, S. SAHNI (1977) Bounds for LPT schedules on uniform processors. *SIAM J. Comput.* 6,155-166.

T. GONZALEZ, D.B. JOHNSON (1977) A new algorithm for preemptive scheduling of trees. Technical Report 222, Computer Science Department, Pennsylvania State University.

T. GONZALEZ, E.L. LAWLER, S. SAHNI (1978) Optimal preemptive scheduling of a fixed number of unrelated processors in linear time. To appear.

T. GONZALEZ, S. SAHNI (1976) Open shop scheduling to minimize finish time. *J. Assoc. Comput. Mach.* 23,665-679.

T. GONZALEZ, S. SAHNI (1978A) Flowshop and jobshop schedules: complexity and approximation. *Operations Res.* 26,36-52.

T. GONZALEZ, S. SAHNI (1978B) Preemptive scheduling of uniform processor systems. *J. Assoc. Comput. Mach.* 25,91-101.

D.K. GOYAL (1977A) Scheduling equal execution time tasks under unit resource restriction. To appear.

D.K. GOYAL (1977B) Non-preemptive scheduling of unequal execution time tasks on two identical processors. Technical Report CS-77-039, Computer Science Department, Washington State University, Pullman.

R.L. GRAHAM (1966) Bounds for certain multiprocessing anomalies. *Bell System Tech. J.* 45,1563-1581.

R.L. GRAHAM (1969) Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.* 17,263-269.

R.L. GRAHAM (1976) Bounds on the performance of scheduling algorithms. In: [Coffman 1976], 165-227.

R.L. GRAHAM (-) Unpublished.

R.L. GRAHAM, E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN (1978) Optimization and approximation in deterministic sequencing and scheduling: a

survey. This volume, 169-214.

J.N.D. GUPTA, S.S. REDDI (1978) Improved dominance conditions for the three-machine flowshop scheduling problem. *Operations Res.* 26,200-203.

G. GÜTTLER, K. MEHLHORN, W. SCHNEIDER, N. WERNET (1976) Binary search trees: average and worst case behavior. In: E.J. NEUHOLD (ed.) (1976) *GI - 6. Jahrestagung*. Informatik-Fachberichte 5, Springer, Berlin, 301-313.

J. HARTMANIS (1971) Computational complexity of random access stored program machines. *Math. Systems Theory* 5,232-245.

J. HARTMANIS, P.M. LEWIS II, R.E. STEARNS (1965) Hierarchies of memory limited computations. *IEEE Confer. Record Switching Circuit Theory and Logical Design*, 179-190.

J. HARTMANIS, J. SIMON (1974) On the power of multiplication in random access machines. *Proc. 15th Annual IEEE Symp. Switching and Automata Theory*, 13-23.

J. HARTMANIS, J. SIMON (1976) On the structure of feasible computations. In: M. RUBINOFF, M.C. YOVITS (eds.) (1976) *Advances in Computers, Volume 14*. Academic Press, New York, 1-43.

J. HARTMANIS, R.E. STEARNS (1965) On the computational complexity of algorithms. *Trans. Amer. Math. Soc.* 117,285-306.

K. HELBIG HANSEN, J. KRARUP (1974) Improvements of the Held-Karp algorithm for the symmetric traveling-salesman problem. *Math. Programming* 7,87-96.

M. HELD, R.M. KARP (1962) A dynamic programming approach to sequencing problems. *J. SIAM* 10,196-210.

M. HELD, R.M. KARP (1970) The traveling-salesman problem and minimum spanning trees. *Operations Res.* 18,1138-1162.

M. HELD, R.M. KARP (1971) The traveling-salesman problem and minimum spanning trees: part II. *Math Programming* 1,6-25.

J.E. HOPCROFT, J.D. ULLMAN (1969) *Formal Languages and Their Relations to Automata*. Addison-Wesley, Reading, Mass.

W.A. HORN (1972) Single-machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAM J. Appl. Math.* 23,189-202.

W.A. HORN (1973) Minimizing average flow time with parallel machines. *Operations Res.* 21,846-847.

W.A. HORN (1974) Some simple scheduling algorithms. *Naval Res. Logist. Quart.* 21,177-185.

E. HOROWITZ, S. SAHNI (1976) Exact and approximate algorithms for scheduling nonidentical processors. *J. Assoc. Comput. Mach.* 23,317-327.

E.C. HORVATH, S. LAM, R. SETHI (1977) A level algorithm for preemptive scheduling. *J. Assoc. Comput. Mach.* 24,32-43.

N.C. HSU (1966) Elementary proof of Hu's theorem on isotone mappings. *Proc. Amer. Math. Soc.* 17,111-114.

T.C. HU (1961) Parallel sequencing and assembly line problems. *Operations Res.* 9,841-848.

O.H. IBARRA, C.E. KIM (1975A) Fast approximation algorithms for the knapsack and sum of subset problems. *J. Assoc. Comput. Mach.* 22,463-468.

O.H. IBARRA, C.E. KIM (1975B) Scheduling for maximum profit. Technical Report, Computer Science Department, University of Minnesota, Minneapolis.

O.H. IBARRA, C.E. KIM (1976) On two-processor scheduling of one- or two-unit time tasks with precedence constraints, *J. Cybernet.* 5,87-109.

O.H. IBARRA, C.E. KIM (1977) Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. Assoc. Comput. Mach.* 24,280-289.

E. IGNALL, L. SCHRAGE (1965) Application of the branch-and-bound technique to some flow-shop scheduling problems. *Operations Res.* 13,400-412.

J.R. JACKSON (1955) Scheduling a production line to minimize maximum tardiness. Research Report 43, Management Science Research Project, University of California, Los Angeles.

J.R. JACKSON (1956) An extension of Johnson's results on job lot scheduling. *Naval Res. Logist. Quart.* 3,201-203.

D.S. JOHNSON (1973) Near-optimal bin packing algorithms. Report MAC TR-109, Massachusetts Institute of Technology, Cambridge, Mass.

D.S. JOHNSON (1974A) Fast algorithms for bin packing. *J. Comput. System Sci.* 8,272-314.

D.S. JOHNSON (1974B) Approximation algorithms for combinatorial problems. *J. Comput. System Sci.* 9,256-278.

D.S. JOHNSON, A. DEMERS, J.D. ULLMAN, M.R. GAREY, R.L. GRAHAM (1974) Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.* 3,299-325.

S.M. JOHNSON (1954) Optimal two- and three-stage production schedules with setup times included. *Naval. Res. Logist. Quart.* 1,61-68.

S.M. JOHNSON (1958) Discussion: sequencing *n* jobs on two machines with arbitrary time lags. *Management Sci.* 5,299-303.

D.G. KAFURA, V.Y. SHEN (1976) An algorithm to design the memory configuration of a computer network. Technical Report 76-6, Computer Science Department, Iowa State University, Ames.

D.G. KAFURA, V.Y. SHEN (1977) Task scheduling on a multiprocessor system with independent memories. *SIAM J. Comput.* 6,167-187.

R.M. KARP (1972) Reducibility among combinatorial problems. In: R.E. MILLER, J.W. THATCHER (eds.) (1972) *Complexity of Computer Computations*. Plenum Press, New York, 85-103.

R.M. KARP (1975A) On the computational complexity of combinatorial problems. *Networks* 5,45-68.

R.M. KARP (1975B) The fast approximate solution of hard combinatorial problems. *Proc. 6th Southeastern Conf. Combinatorics, Graph Theory, and Computing,* Utilitas Mathematica Publishing, Winnipeg, 15-31.

R.M. KARP (1976) The probabilistic analysis of some combinatorial search algorithms. In: J.F. TRAUB (ed.) (1976) *Algorithms and Complexity: New Directions and Recent Results*. Academic Press, New York, 1-19.

R.M. KARP (1977) Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Math. Operations Res.* 2, 209-224; this volume, 141-168.

A.V. KARZANOV (1974) Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dokl.* 15,434-437.

M.T. KAUFMAN (1972) Anomalies in scheduling unit-time tasks. Technical Report 34, Stanford Electronic Laboratory.

M.T. KAUFMAN (1974) An almost-optimal algorithm for the assembly line scheduling problem. *IEEE Trans. Computers* C-23,1169-1174.

H. KISE, T. IBARAKI, H. MINE (1978) A solvable case of the one-machine scheduling problem with ready and due times. *Operations Res.* 26,121-126.

V. KLEE, G.J. MINTY (1972) How good is the simplex algorithm? In: O. SHISHA (ed.) (1972) *Inequalities III*. Academic Press, New York, 159-175.

D.E. KNUTH (1973A) *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Mass.

D.E. KNUTH (1973B) Personal communication to T.C. Hu, July 23, 1973.

D.E. KNUTH (1974) A terminological proposal. *SIGACT News* 6.1,12-18.

D.E. KNUTH (-) Personal communications.

D. KOZEN (1976) On parallelism in Turing machines. *Proc. 17th Annual IEEE Symp. Foundations of Computer Science,* 89-97.

K.L. KRAUSE (1973) *Analysis of Computer Scheduling with Memory Constraints.*

Doctoral Disseration, Computer Science Department, Purdue University, West Lafayette.

K.L. KRAUSE, V.Y. SHEN, H.D. SCHWETMAN (1975,1977) Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *J. Assoc. Comput. Mach.* 22,522-550;24,527.

M.S. KRISHNAMOORTHY (1975) An NP-hard problem in bipartite graphs. *SIGACT News* 7.1,26.

P.D. KROLAK, W. FELTS, G. MARBLE (1970) Efficient heuristics for solving large traveling-salesman problems. Presented at 7th Internat. Symp. Mathematical Programming, The Hague.

P.D. KROLAK, W. FELTS, G. MARBLE (1971) A man-machine approach toward solving the traveling salesman problem. *Comm. ACM* 14,327-334.

M. KUNDE (1976) Beste Schranken beim LP-Scheduling. Bericht 7603, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel.

H.W. LABBERS, JR. (1976) Labbers' zehnstellige Tafeln für die 10-adischen Werte der Ackermannschen Funktion. Facsimile-Edition, Mathematisch Instituut, Universiteit van Amsterdam; reprinted in: *Nieuw Arch. Wisk.* 24,209.

J. LABETOULLE, E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN (1978) Preemptive scheduling of uniform processors subject to release dates. To appear.

B.J. LAGEWEG, E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN (1978A) Computer aided complexity classification of deterministic scheduling problems. To appear.

B.J. LAGEWEG, J.K. LENSTRA, A.H.G. RINNOOY KAN (1976) Minimizing maximum lateness on one machine: computational experience and some applications. *Statistica Neerlandica* 30,25-41.

B.J. LAGEWEG, J.K. LENSTRA, A.H.G. RINNOOY KAN (1977) Job-shop scheduling by implicit enumeration. *Management Sci.* 24,441-450.

B.J. LAGEWEG, J.K. LENSTRA, A.H.G. RINNOOY KAN (1978B) A general bounding scheme for the permutation flow-shop problem. *Operations Res.* 26,53-67.

S. LAM, R. SETHI (1977) Worst case analysis of two scheduling algorithms. *SIAM J. Comput.* 6,518-536.

E.L. LAWLER (1973) Optimal sequencing of a single machine subject to precedence constraints. *Management Sci.* 19,544-546.

E.L. LAWLER (1976A) Sequencing to minimize the weighted number of tardy

jobs. *Rev. Française Automat. Informat. Recherche Opĕrationnelle* 10.5 Suppl.27-33.

E.L. LAWLER (1976B) *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York.

E.L. LAWLER (1977) A "pseudopolynomial" algorithm for sequencing jobs to minimize total tardiness. *Ann. Discrete Math.* 1,331-342.

E.L. LAWLER (1978A) Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Ann. Discrete Math.* 2,75-90.

E.L. LAWLER (1978B) Fast approximation algorithms for knapsack problems. This volume, 109-139.

E.L. LAWLER, J. LABETOULLE (1978) Preemptive scheduling of unrelated parallel processors. *J. Assoc. Comput. Mach.*, to appear.

E.L. LAWLER, J.M. MOORE (1969) A functional equation and its application to resource allocation and sequencing problems. *Management Sci.* 16,77-84.

J.K. LENSTRA (1977) *Sequencing by Enumerative Methods*. Mathematical Centre Tracts 69, Mathematisch Centrum, Amsterdam.

J.K. LENSTRA (-) Unpublished.

J.K. LENSTRA, A.H.G. RINNOOY KAN (1975) Some simple applications of the travelling salesman problem. *Operational Res. Quart.* 26,717-733.

J.K. LENSTRA, A.H.G. RINNOOY KAN (1976) On general routing problems. *Networks* 6,273-280.

J.K. LENSTRA, A.H.G. RINNOOY KAN (1978A) Complexity of scheduling under precedence constraints. *Operations Res.* 26,22-35.

J.K. LENSTRA, A.H.G. RINNOOY KAN (1978B) Computational complexity of discrete optimization problems. This volume, 63-85.

J.K. LENSTRA, A.H.G. RINNOOY KAN, P. BRUCKER (1977) Complexity of machine scheduling problems. *Ann. Discrete Math.* 1,343-362.

S. LIN, B.W. KERNIGHAN (1973) An effective heuristic algorithm for the traveling-salesman problem. *Operations Res.* 21,498-516.

R. LIPTON (1976) The reachability problem requires exponential space. *Theoret. Comput. Sci.*, to appear.

C.L. LIU (1972) Optimal scheduling on multi-processor computing systems. *Proc. 13th Annual IEEE Symp. Switching and Automata Theory*, 155-160.

C.L. LIU (1976) Deterministic job scheduling in computing systems. Department of Computer Science, University of Illinois at Urbana-Champaign.

J.W.S. LIU, C.L. LIU (1974A) Bounds on scheduling algorithms for heterogeneous computing systems. In: J.L. ROSENFELD (ed.) (1974) *Information*

Processing 74. North-Holland, Amsterdam, 349-353.

J.W.S. LIU, C.L. LIU (1974B) Bounds on scheduling algorithms for heterogeneous computing systems. Technical Report UIUCDCS-R-74-632, Department of Computer Science, University of Illinois at Urbana-Champaign, 68 pp.

J.W.S. LIU, C.L. LIU (1974C) Performance analysis of heterogeneous multiprocessor computing systems. In: E. GELENBE, R. MAHL (eds.) (1974) *Computer Architectures and Networks*. North-Holland, Amsterdam, 331-343.

G.B. McMAHON (1969) Optimal production schedules for flow shops. *Canad. Operational Res. Soc. J.* 7,141-151.

G.B. McMAHON (1971) *A Study of Algorithms for Industrial Scheduling Problems*. Thesis, University of New South Wales, Kensington.

G.B. McMAHON, M. FLORIAN (1975) On scheduling with ready times and due dates to minimize maximum lateness. *Operations Res.* 23,475-482.

R. McNAUGHTON (1959) Scheduling with deadlines and loss functions. *Management Sci.* 6,1-12.

K. MEHLHORN (1977) Dynamic binary search. In: A. SALOMAA, M. STEINBY (eds.) (1977) *Automata, Languages and Programming*. Lecture Notes in Computer Science 52, Springer, Berlin, 323-336.

M.L. MINSKY (1961) Recursive unsolvability of Post's problem of "Tag" and other topics in the theory of Turing machines. *Ann. of Math.* 74,437-455.

D. MITCHIE, J.G. FLEMING, J.V. OLDFIELD (1968) A comparison of heuristic, interactive, and unaided methods of solving a shortest-route problem. In: D. MITCHIE (ed.) (1968) *Machine Intelligence 3*, Edinburgh University Press, 245-255.

L.G. MITTEN (1958) Sequencing *n* jobs on two machines with arbitrary time lags. *Management Sci.* 5,293-298.

C.L. MONMA (1977) Optimal m×n flow shop sequencing with precedence constraints and lag times. School of Operations Research, Cornell University, Ithaca, N.Y.

C.L. MONMA (1978) Personal communication.

C.L. MONMA, J.B. SIDNEY (1977) A general algorithm for optimal job sequencing with series-parallel precedence constraints. Technical Report 347, School of Operations Research, Cornell University, Ithaca, N.Y.

J.M. MOORE (1968) An *n* job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Sci.* 15,102-109.

P.A.P. MORAN (1968) *Introduction to Probability Theory*. Clarendon Press, Oxford.

R.R. MUNTZ, E.G. COFFMAN, JR. (1969) Optimal preemptive scheduling on two-processor systems. *IEEE Trans. Computers* C-18,1014-1020.

R.R. MUNTZ, E.G. COFFMAN, JR. (1970) Preemptive scheduling of real time tasks on multiprocessor systems. *J. Assoc. Comput. Mach.* 17,324-338.

Y. MURAOKA (1971) *Parallelism, Exposure and Exploitation in Programs*. Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

J.F. MUTH, G.L. THOMPSON (eds.) (1963) *Industrial Scheduling*. Prentice-Hall, Englewood Cliffs, N.J., 236.

I. NABESHIMA (1963) Sequencing on two machines with start lag and stop lag. *J. Operations Res. Soc. Japan* 5,97-101.

P. NAUR (ed.) (1963) Revised report on the algorithmic language ALGOL 60. *Comm. ACM* 6,1-17.

G.L. NEMHAUSER, L.A. WOLSEY (1976) Best algorithms for approximating the maximum of a submodular set function. Discussion Paper 7636, CORE, Louvain.

G.L. NEMHAUSER, L.A. WOLSEY, M.L. FISHER (1978) An analysis of approximations for maximizing submodular set functions - I. *Math. Programming*, to appear.

S.S. PANWALKAR, W. ISKANDER (1977) A survey of scheduling rules. *Operations Res.* 25,45-61.

C.H. PAPADIMITRIOU (1976) On the complexity of edge traversing. *J. Assoc. Comput. Mach.* 23,544-554.

C.H. PAPADIMITRIOU (1977) The Euclidean traveling salesman problem is NP-complete. *Theoret. Comput. Sci.* 4,237-244.

M.S. PATERSON (-) Unpublished.

A. PAZ, S. MORAN (1977) Non-deterministic polynomial optimization problems and their approximation: abridged version. In: A. SALOMAA, M. STEINBY (eds.) (1977) *Automata, Languages and Programming*. Lecture Notes in Computer Science 52, Springer, Berlin, 370-379.

J. PIEHLER (1960) Ein Beitrag zum Reihenfolgeproblem. *Unternehmensforschung* 4,138-142.

V.R. PRATT, L.J. STOCKMEYER, M. RABIN (1974) A characterization of the power of vector machines. *Proc. 6th Annual ACM Symp. Theory of Computing*, 122-134.

S.S. REDDI, C.V. RAMAMOORTHY (1972) On the flow-shop sequencing problem with no wait in process. *Operations Res. Quart.* 23,323-331.

E.M. REINGOLD, J. NIEVERGELT, N. DEO (1977) *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Englewood Cliffs, N.J.

A.H.G. RINNOOY KAN (1976) *Machine Scheduling Problems: Classification, Complexity and Computations*. Nijhoff, The Hague.

A.H.G. RINNOOY KAN, B.J. LAGEWEG, J.K. LENSTRA (1975) Minimizing total costs in one-machine scheduling. *Operations Res.* 23,908-927.

R.L. RIVEST, J. VUILLEMIN (1976) On recognizing graph properties from adjacency matrices. *Theoret. Comput. Sci.* 3,371-384.

R.W. ROBINSON (1973) Counting labeled acyclic digraphs. In: F. HARARY (ed.) (1973) *New Directions in the Theory of Graphs*. Academic Press, New York, 239-271.

P. ROSENFELD (-) Unpublished.

D.J. ROSENKRANTZ, R.E. STEARNS, P.M. LEWIS II (1977) An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.* 6,563-581.

M.H. ROTHKOPF (1966) Scheduling independent tasks on parallel processors. *Management Sci.* 12,437-447.

B. ROY, B. SUSSMANN (1964) Les problèmes d'ordonnancement avec contraintes disjonctives. Note DS no.9 bis, SEMA, Montrouge.

G.S. SACERDOTE, R.L. TENNEY (1977) The decidability of the reachability problem for vector addition systems. *Proc. 9th Annual ACM Symp. Theory of Computing*, 61-76.

S. SAHNI (1975) Approximate algorithms for the 0-1 knapsack problem. *J. Assoc. Comput. Mach.* 22,115-124.

S. SAHNI (1976) Algorithms for scheduling independent tasks. *J. Assoc. Comput. Mach.* 23,116-127.

S. SAHNI, Y. CHO (1977A) Scheduling independent tasks with due times on a uniform processor system. Computer Science Department, University of Minnesota, Minneapolis.

S. SAHNI, Y. CHO (1977B) On line scheduling of a uniform processor system with release times. Computer Science Department, University of Minnesota, Minneapolis.

S. SAHNI, Y. CHO (1977C) Complexity of scheduling jobs with no wait in process. Technical Report 77-20, Computer Science Department, University of Minnesota, Minneapolis.

S. SAHNI, T. GONZALEZ (1976) P-complete approximation problems. *J. Assoc. Comput. Mach.* 23,555-565.

J.E. SAVAGE (1976) *The Complexity of Computing*. Wiley, New York.

W.J. SAVITCH (1970) Relationships between nondeterministic and deterministic tape complexities. *J. Comput. System Sci.* 4,177-192.

W.J. SAVITCH, M.J. STIMSON (1976) The complexity of time bounded recursive computations. *Proc. 1976 Confer. Information Sciences and Systems*, The John Hopkins University, 42-46.

W.J. SAVITCH, M.J. STIMSON (1978) Time bounded random access machines with parallel processing. *J. Assoc. Comput. Mach.*, to appear.

T.J. SCHAEFER (1976) Complexity of decision problems based on finite two-person perfect-information games. *Proc. 8th Annual ACM Symp. Theory of Computing*, 41-59.

P. SCHUSTER (1976) Probleme, die zum Erfüllungsproblem der Aussagenlogik polynomial äquivalent sind. In: E. SPECKER, V. STRASSEN (eds.) (1976) *Komplexität von Entscheidungsproblemen: ein Seminar*. Lecture Notes in Computer Science 43, Springer, Berlin, 36-48.

R. SETHI (1976A) Algorithms for minimal-length schedules. In: [Coffman 1976], 51-99.

R. SETHI (1976B) Scheduling graphs on two processors. *SIAM J. Comput.* 5,73-82.

R. SETHI (1977) On the complexity of mean flow time scheduling. *Math. Operations Res.* 2,320-330.

M.I. SHAMOS (1975) Geometric complexity. *Proc. 7th Annual ACM Symp. Theory of Computing*, 224-233.

M.I. SHAMOS, D. HOEY (1975) Closest-point problems. *Proc. 16th Annual IEEE Symp. Foundations of Computer Science*, 151-162.

J.C. SHEPHERDSON, H.E. STURGIS (1963) Computability of recursive functions. *J. Assoc. Comput. Mach.* 10,217-255.

J.B. SIDNEY (1973) An extension of Moore's due date algorithm. In: S.E. ELMAGHRABY (ed.) (1973) *Symposium on the Theory of Scheduling and its Applications*. Lecture Notes in Economics and Mathematical Systems 86, Springer, Berlin, 393-398.

J.B. SIDNEY (1975) Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs. *Operations Res.* 23,283-298.

J.B. SIDNEY (1977) The two-machine maximum flow time problem with series-parallel precedence constraints. Faculty of Management Sciences,

University of Ottawa.

T.H.C. SMITH, V. SRINIVASAN, G.L. THOMPSON (1977) Computational performance of three subtour elimination algorithms for solving asymmetric traveling salesman problems. *Ann. Discrete Math.* 1,495-506.

T.H.C. SMITH, G.L. THOMPSON (1977) A lifo implicit enumeration search algorithm for the symmetric traveling salesman problem using Held and Karp's 1-tree relaxation. *Ann. Discrete Math.* 1,479-493.

W.E. SMITH (1956) Various optimizers for single-stage production. *Naval Res. Logist. Quart.* 3,59-66.

H.I. STERN (1976) Minimizing makespan for independent jobs on nonidentical parallel machines - an optimal procedure. Working Paper 2/75, Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer-Sheva.

L.J. STOCKMEYER, A.R. MEYER (1973) Word problems requiring exponential time: preliminary report. *Proc. 5th Annual ACM Symp. Theory of Computing*, 1-9.

V. STRASSEN (1969) Gaussian elimination is not optimal. *Numer. Math.* 13,354-356.

W. SZWARC (1968) On some sequencing problems. *Naval Res. Logist. Quart.* 15,127-155.

W. SZWARC (1971) Elimination methods in the $m \times n$ sequencing problem. *Naval Res. Logist. Quart.* 18,295-305.

W. SZWARC (1973) Optimal elimination methods in the $m \times n$ sequencing problem. *Operations Res.* 21,1250-1259.

W. SZWARC (1978) Dominance conditions for the three-machine flow-shop problem. *Operations Res.* 26,203-206.

R.E. TARJAN (1975A) Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.* 22,215-225.

R.E. TARJAN (1975B) Applications of path compression on balanced trees. Report CS 75-512, Department of Computer Science, Stanford University.

W. TOWNSEND (1977A) A branch-and-bound method for sequencing problems with linear and exponential penalty functions. *Operational Res. Quart.* 28,191-200.

W. TOWNSEND (1977B) Sequencing n jobs on m machines to minimize maximum tardiness: a branch-and-bound solution. *Management Sci.* 23,1016-1019.

A.M. TURING (1936) On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.* 2-42,230-265.

A.M. TURING (1937) On computable numbers, with an application to the Entscheidungsproblem. A correction. *Proc. London Math. Soc.* 2-43,544-546.

J.D. ULLMAN (1975) *NP*-complete scheduling problems. *J. Comput. System Sci.* 10,384-393.

J.D. ULLMAN (1976) Complexity of sequencing problems. In: [Coffman 1976], 139-164.

J.M. VAN DEMAN, K.R. BAKER (1974) Minimizing mean flowtime in the flow shop with no intermediate queues. *AIIE Trans.* 6,28-34.

P. VAN EMDE BOAS (1977) Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Lett.* 6,80-82.

P. VAN EMDE BOAS, R. KAAS, E. ZIJLSTRA (1977) Design and implementation of an efficient priority queue. *Math. Systems Theory* 10,99-127.

J. VAN LEEUWEN (1976) The complexity of data organization. In: K.R. APT, J.W. DE BAKKER (eds.) (1976) *Foundations of Computer Science II, Part I.* Mathematical Centre Tracts 81, Mathematisch Centrum, Amsterdam, 37-147.

J. VUILLEMIN (1978) A data structure for manipulating priority queues. *Comm. ACM* 21,309-315.

D.A. WISMER (1972) Solution of the flowshop-scheduling problem with no intermediate queues. *Operations Res.* 20,689-697.