GRAPH THEORY,
COMBINATORICS AND
ALGORITHMS

INTERDISCIPLINARY
APPLICATIONS
GRAPH THEORY, COMBINATORICS AND ALGORITHMS

INTERDISCIPLINARY APPLICATIONS

Edited by

Martin Charles Golumbic
Irith Ben-Arroyo Hartman
Contents

Foreword ................................................................. vii

Chapter 1 Optimization Problems Related to Internet Congestion Control
Richard Karp ......................................................... 1

Chapter 2 Problems in Data Structures and Algorithms
Robert Tarjan ......................................................... 17

Chapter 3 Algorithmic Graph Theory and its Applications
Martin Charles Golumbic ......................................... 41

Chapter 4 Decompositions and Forcing Relations in Graphs and Other Combinatorial Structures
Ross McConnell ...................................................... 63

Chapter 5 The Local Ratio Technique and its Application to Scheduling and Resource Allocation Problems
Reuven Bar-Yehuda, Keren Bendel, Ari Freund and Dror Rawitz 107

Chapter 6 Domination Analysis of Combinatorial Optimization Algorithms and Problems
Gregory Gutin and Anders Yeo .................................... 145

Chapter 7 On Multi-Object Auctions and Matching Theory: Algorithmic Aspects
Michal Penn and Moshe Tennenholtz .......................... 173

Chapter 8 Strategies for Searching Graphs
Shmuel Gal .............................................................. 189

Chapter 9 Recent Trends in Arc Routing
Alain Hertz .............................................................. 215

Chapter 10 Software and Hardware Testing Using Combinatorial Covering Suites
Alan Hartman .......................................................... 237

Chapter 11 Incidences
Janos Pach and Micha Sharir ....................................... 267
Foreword

The Haifa Workshops on Interdisciplinary Applications of Graph Theory, Combinatorics and Algorithms have been held at the Caesarea Rothschild Institute (C.R.I.), University of Haifa, every year since 2001. This volume consists of survey chapters based on presentations given at the 2001 and 2002 Workshops, as well as other colloquia given at C.R.I. The Rothschild Lectures of Richard Karp (Berkeley) and Robert Tarjan (Princeton), both Turing award winners, were the highlights of the Workshops. Two chapters based on these talks are included. Other chapters were submitted by selected authors and were peer reviewed and edited. This volume, written by various experts in the field, focuses on discrete mathematics and combinatorial algorithms and their applications to real world problems in computer science and engineering. A brief summary of each chapter is given below.

Richard Karp’s overview, *Optimization Problems Related to Internet Congestion Control*, presents some of the major challenges and new results related to controlling congestion in the Internet. Large data sets are broken down into smaller packets, all competing for communication resources on an imperfect channel. The theoretical issues addressed by Prof. Karp lead to a deeper understanding of the strategies for managing the transmission of packets and the retransmission of lost packets.

Robert Tarjan’s lecture, *Problems in Data Structures and Algorithms*, provides an overview of some data structures and algorithms discovered by Tarjan during the course of his career. Tarjan gives a clear exposition of the algorithmic applications of basic structures like search trees and self-adjusting search trees, also known as *splay trees*. Some open problems related to these structures and to the minimum spanning tree problem are also discussed.

The third chapter by Martin Charles Golumbic, *Algorithmic Graph Theory and its Applications*, is based on a survey lecture given at Clemson University. This chapter is aimed at the reader with little basic knowledge of graph theory, and it introduces the reader to the concepts of interval graphs and other families of intersection graphs. The lecture includes demonstrations of these concepts taken from real life examples.

The chapter *Decompositions and Forcing Relations in Graphs and other Combinatorial Structures* by Ross McConnell deals with problems related to classes of intersection graphs, including interval graphs, circular-arc graphs, probe interval graphs, permutation graphs, and others. McConnell points to a general structure called *modular decomposition* which helps to obtain linear bounds for recognizing some of these graphs, and solving other problems related to these special graph classes.
In their chapter *The Local Ratio Technique and its Application to Scheduling and Resource Allocation Problems*, Bar-Yehuda, Bendel, Freund and Rawitz give a survey of the local ratio technique for approximation algorithms. An approximation algorithm efficiently finds a feasible solution to an intractable problem whose value approximates the optimum. There are numerous real life intractable problems, such as the scheduling problem, which can be approached only through heuristics or approximation algorithms. This chapter contains a comprehensive survey of approximation algorithms for such problems.

*Domination Analysis of Combinatorial Optimization Algorithms and Problems* by Gutin and Yeo provides an alternative and a complement to approximation analysis. One of the goals of domination analysis is to analyze the domination ratio of various heuristic algorithms. Given a problem $P$ and a heuristic $H$, the ratio between the number of feasible solutions that are not better than a solution produced by $H$, and the total number of feasible solutions to $P$, is the domination ratio. The chapter discusses domination analyses of various heuristics for the well-known traveling salesman problem, as well as other intractable combinatorial optimization problems, such as the minimum partition problem, multiprocessor scheduling, maximum cut, k-satisfiability, and others.

Another real-life problem is the design of auctions. In their chapter *On Multi-Object Auctions and Matching Theory: Algorithmic Aspects*, Penn and Tennenholtz use b-matching techniques to construct efficient algorithms for combinatorial and constrained auction problems. The typical auction problem can be described as the problem of designing a mechanism for selling a set of objects to a set of potential buyers. In the combinatorial auction problem bids for bundles of goods are allowed, and the buyer may evaluate a bundle of goods for a different value than the sum of the values of each good. In constrained auctions some restrictions are imposed upon the set feasible solutions, such as the guarantee that a particular buyer will get at least one good from a given set. Both combinatorial and constrained auction problems are NP-complete problems, however, the authors explore special tractable instances where b-matching techniques can be used successfully.

Shmuel Gal’s chapter *Strategies for Searching Graphs* is related to the problem of detecting an object such as a person, a vehicle, or a bomb hiding in a graph (on an edge or at a vertex). It is generally assumed that there is no knowledge about the probability distribution of the target’s location and, in some cases, even the structure of the graph is not known. Gal uses probabilistic methods to find optimal search strategies that assure finding the target in minimum expected time.

The chapter *Recent Trends in Arc Routing* by Alain Hertz studies the problem of finding a least cost tour of a graph, with demands on the edges, using a fleet of identical vehicles. This problem and other related problems are intractable, and the chapter reports on recent exact and heuristic algorithms. The problem has applications in garbage collection, mail delivery, snow clearing, network maintenance, and many others.
Software and Hardware Testing Using Combinatorial Covering Suites by Alan Hartman is an example of the interplay between pure mathematics, computer science, and the applied problems generated by software and hardware engineers. The construction of efficient combinatorial covering suites has important applications in the testing of software and hardware systems. This chapter discusses the lower bounds on the size of covering suites, and gives a series of constructions that achieve these bounds asymptotically. These constructions involve the use of finite field theory, extremal set theory, group theory, coding theory, combinatorial recursive techniques, and other areas of computer science and mathematics.

Janos Pach and Micha Sharir’s chapter, Incidences, relates to the following general problem in combinatorial geometry: What is the maximum number of incidences between m points and n members of a family of curves or surfaces in d-space? Results of this kind have numerous applications to geometric problems related to the distribution of distances among points, to questions in additive number theory, in analysis, and in computational geometry.

We would like to thank the authors for their enthusiastic response to the challenge of writing a chapter in this book. We also thank the referees for their comments and suggestions. Finally, this book, and many workshops, international visits, courses and projects at CRI, are the results of a generous grant from the Caesarea Edmond Benjamin de Rothschild Foundation. We are greatly indebted for their support throughout the last four years.

Martin Charles Golumbic
Irith Ben-Arroyo Hartman
Caesarea Edmond Benjamin de Rothschild Foundation Institute for Interdisciplinary Applications of Computer Science
University of Haifa, Israel
Optimization Problems Related to Internet Congestion Control

Richard Karp
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Introduction

I’m going to be talking about a paper by Elias Koutsoupias, Christos Papadimitriou, Scott Shenker and myself, that was presented at the 2000 FOCS Conference [1] related to Internet-congestion control. Some people during the coffee break expressed surprise that I’m working in this area, because over the last several years, I have been concentrating more on computational biology, the area on which Ron Shamir reported so eloquently in the last lecture. I was having trouble explaining, even to myself, how it is that I’ve been working in these two very separate fields, until Ron Pinter just explained it to me, a few minutes ago. He pointed out to me that improving the performance of the web is crucially important for bioinformatics, because after all, people spend most of their time consulting distributed data bases. So this is my explanation, after the fact, for working in these two fields.

The Model

In order to set the stage for the problems I’m going to discuss, let’s talk in slightly oversimplified terms about how information is transmitted over the Internet. We’ll consider the simplest case of what’s called unicast—the transmission of message or file $D$ from one Internet host, or node, $A$ to another node $B$. The data $D$, that host $A$ wishes to send to host $B$ is broken up into packets of equal size which are assigned consecutive serial numbers. These packets form a flow passing through a series of links and routers on the Internet. As the packets flow through some path of links and routers, they pass through queues. Each link has one or more queues of finite capacity in which packets are buffered as they pass through the routers. Because these buffers have a finite capacity, the queues may sometimes overflow. In that case, a choice has to be
made as to which packets shall be dropped. There are various queue disciplines. The one most commonly used, because it is the simplest, is a simple first-in-first-out (FIFO) discipline. In that case, when packets have to be dropped, the last packet to arrive will be the first to be dropped. The others will pass through the queue in first-in-first-out order.

First-in-first-out disciplines, as we will see, have certain disadvantages. Therefore, people talk about *fair queuing* where several, more complicated data structures are used in order to treat all of the data flows more fairly, and in order to transmit approximately the same number of packets from each flow. But in practice, the overhead of fair queuing is too large, although some approximations to it have been contemplated. And so, this first-in-first-out queuing is the most common queuing discipline in practical use.

Now, since not all packets reach their destination, there has to be a mechanism for the receiver to let the sender know whether packets have been received, and which packets have been received, so that the sender can retransmit dropped packets. Thus, when the receiver \( B \) receives the packets, it sends back an *acknowledgement* to \( A \). There are various conventions about sending acknowledgements. The simplest one is when \( B \) simply lets \( A \) know the serial number of the first packet not yet received. In that case \( A \) will know that consecutive packets up to some point have been received, but won’t know about the packets after that point which may have been received sporadically. Depending on this flow of acknowledgements back to \( A \), \( A \) will detect that some packets have been dropped because an acknowledgement hasn’t been received within a reasonable time, and will retransmit certain of these packets.

The most undesirable situation is when the various flows are transmitting too rapidly. In that case, the disaster of *congestion collapse* may occur, in which so many packets are being sent that most of them never get through—they get dropped. The acknowledgement tells the sender that the packet has been dropped. The sender sends
the dropped packet again and again, and eventually, the queues fill up with packets that are retransmissions of previous packets. These will eventually be dropped and never get to their destinations. The most important single goal of congestion control on the Internet is to avoid congestion collapse.

There are other goals as well. One goal is to give different kinds of service to different kinds of messages. For example, there are simple messages that have no particular time urgency, email messages, file transfers and the like, but then there are other kinds of flows, like streaming media etc. which have real-time requirements. I won’t be getting into quality-of-service issues in this particular talk to any depth. Another goal is to allocate bandwidth fairly, so that no flow can hog the bandwidth and freeze out other flows. There is the goal of utilizing the available bandwidth. We want to avoid congestion collapse, but also it is desirable not to be too conservative in sending packets and slow down the flow unnecessarily.

The congestion control algorithm which is standard on the Internet is one that the various flows are intended to follow voluntarily. Each flow under this congestion control algorithm has a number of parameters. The most important one is the window size $W$—the maximum number of packets that can be in process; more precisely, $W$ is the maximum number of packets that the sender has sent but for which an acknowledgement has not yet been received. The second parameter of importance is the roundtrip time (RTT). This parameter is a conservative upper estimate on the time it should take for a packet to reach its destination and for the acknowledgement to come back. The significance of this parameter is that if the acknowledgement is not received within RTT time units after transmission, then the sender will assume that the packet was dropped. Consequently, it will engage in retransmission of that particular packet and of all the subsequent packets that were sent up to that point, since packet drops often occur in bursts.

In the ideal case, things flow smoothly, the window size is not excessive and not too small, no packet is dropped, and a receives an acknowledgement and sends a packet every $RTT/W$ time steps. But in a bad case, the packet “times out”, and then all packets sent in the last interval of time $RTT$ must be retransmitted. The crucial question is, therefore, how to modify, how to adjust this window. The window size should continually increase as long as drops are not experienced, but when drops are experienced, in order to avoid repetition of those drops, the sender should decrease its window size.

The Jacobson algorithm, given below, is the standard algorithm for adjusting the window size. All Internet service providers are supposed to adhere to it.

**Jacobson’s Algorithm for adjusting $W$**

**start-up:**

\[
W \leftarrow 1
\]

when acknowledgement received

\[
W \leftarrow W + 1
\]
when timeout occurs
\[ W \leftarrow \lfloor W/2 \rfloor \]
go to main

\textbf{main:}
\begin{verbatim}
if W acknowledgements received before timeout occurs then
    \[ W \leftarrow W + 1 \]
else
    \[ W \leftarrow \lfloor W/2 \rfloor \]
\end{verbatim}

Jacobson’s algorithm gives a rather jagged behavior over time. The window size \( W \) is linearly increased, but from time to time it is punctuated by a sudden decrease by a factor of two. This algorithm is also called the \textit{additive increase/multiplicative decrease (AIMD) scheme}. There are a number of variations and refinements to this algorithm. The first variation is called \textit{selective acknowledgement}. The acknowledgement is made more informative so that it indicates not only the serial number of the first packet not yet received, but also some information about the additional packets that have been received out of order.

![The sawtooth behavior of Jacobson's standard algorithm.](image)

The second variation is “\textit{random early drop}.” The idea is that instead of dropping packets only when catastrophe threatens and the buffers start getting full, the packets get dropped randomly as the buffers approach saturation, thus giving an early warning that the situation of packet dropping is approaching. Another variation is \textit{explicit congestion notification}, where, instead of dropping packets prematurely at random, warnings are issued in advance. The packets go through, but in the acknowledgement there is a field that indicates “you were close to being dropped; maybe you’d better slow down your rate.” There are other schemes that try to send at the same long-term average rate as Jacobson’s algorithm, but try to smooth out the flow so that you don’t get those jagged changes, the abrupt decreases by a factor of two.

The basic philosophy behind all the schemes that I’ve described so far is voluntary compliance. In the early days, the Internet was a friendly club, and so you could just ask people to make sure that their flows adhere to this standard additive increase/multiplicative decrease (AIMD) scheme. Now, it is really social pressure that holds things together. Most people use congestion control algorithms that they didn’t implement themselves but are implemented by their service provider and if their service provider doesn’t adhere to the AIMD protocol, then the provider gets a bad reputation. So they tend to adhere to this protocol, although a recent survey of the actual algorithms provided by the various Internet service providers indicates a considerable amount of
deviation from the standard, some of this due to inadvertent program bugs. Some of this may be more nefarious—I don’t know.

In the long run, it seems that the best way to ensure good congestion control is not to depend on some voluntary behavior, but to induce the individual senders to moderate their flows out of self-interest. If no reward for adhering, or punishment for violation existed, then any sender who is motivated by self-interest could reason as follows: what I do has a tiny effect on packet drops because I am just one of many who are sharing these links, so I should just send as fast as I want. But if each individual party follows this theme of optimizing for itself, you get the “tragedy of the commons”, and the total effect is a catastrophe. Therefore, various mechanisms have been suggested such as: monitoring individual flow rates, or giving flows different priority levels based on pricing.

The work that we undertook is intended to provide a foundation for studying how senders should behave, or could be induced to behave, if their goal is self-interest and they cannot be relied on to follow a prescribed protocol. There are a couple of ways to study this. We have work in progress which considers the situation as an \( n \)-person non-cooperative game. In the simplest case, you have \( n \) flows competing for a link. As long as some of their flow rates are below a certain threshold, everything will get through. However, as soon as the sum of their flow rates crosses the threshold, some of them will start experiencing packet drops. One can study the Nash equilibrium of this game and try to figure out different kinds of feedback and different kinds of packet drop policies which might influence the players to behave in a responsible way.

The Rate Selection Problem

In the work that I am describing today, I am not going to go into this game theoretic approach, which is in its preliminary stages. I would like to talk about a slightly different situation. The most basic question one could perhaps ask is the following: suppose you had a single flow which over time is transmitting packets, and the flow observes that if it sends at a particular rate it starts experiencing packet drops; if it sends at another rate everything gets through. It gets this feedback in the form of acknowledgements, and if it’s just trying to optimize for itself, and is getting some partial information about its environment and how much flow it can get away with, how should it behave?

The formal problem that we will be discussing today is called the Rate Selection Problem. The problem is: how does a single, self-interested host \( A \), observing the limits on what it can send over successive periods of time, choose to moderate its flow. In the formal model, time will be divided into intervals of fixed length. You can think of the length of the interval as perhaps the roundtrip time. For each time interval \( t \) there is a parameter \( u_t \), defined as the maximum number of packets that \( A \) can send \( B \) without experiencing packet drops. The parameter \( u_t \) is a function of all the other flows in the system, of the queue disciplines that are used, the topology of the Internet, and other factors. Host \( A \) has no direct information about \( u_t \). In each time interval \( t \), the parameter \( x_t \) denotes the number of packets sent by the sender \( A \). If \( x_t \leq u_t \), then
all the packets will be received, none of them will time out and everything goes well. If \( x_t > u_t \), then at least one packet will be dropped, and the sender will suffer some penalty that we will have to model. We emphasize that the sender does not have direct information about the successive thresholds. The sender only gets partial feedback, i.e. whether \( x_t \leq u_t \) or not, because all that the sender can observe about the channel is whether or not drops occurred.

In order to formulate an optimization problem we need to set up a cost function \( c(x, u) \). The function represents the cost of transmitting \( x \) packets in a time period with threshold \( u \). In our models, the cost reflects two major components: \textit{opportunity cost} due to sending of less than the available bandwidth, i.e. when \( x_t < u_t \), and \textit{retransmission delay and overhead} due to dropped packets when \( x_t > u_t \).

We will consider here two classes of cost functions.

The \textbf{severe cost function} is defined as follows:

\[
c(x_t, u_t) = \begin{cases} 
  u_t - x_t & \text{if } x_t \leq u_t \\
  u_t & \text{otherwise}
\end{cases}
\]

The intuition behind this definition is the following: When \( x_t \leq u_t \), the user pays the difference between the amount it could have sent and the actual amount sent. When \( x_t > u_t \), we’ll assume the sender has to resend all the packets that it transmitted in that period. In that case it has no payoff for that period and its cost is \( u_t \), because if it had known the threshold, it could have got \( u_t \) packets through, but in fact, it gets zero.

The \textbf{gentle cost function} will be defined as:

\[
c(x_t, u_t) = \begin{cases} 
  u_t - x_t & \text{if } x_t \leq u_t \\
  \alpha(x_t - u_t) & \text{otherwise}
\end{cases}
\]

where \( \alpha \) is a fixed proportionality factor. Under this function, the sender is punished less for slightly exceeding the threshold. There are various interpretations of this. In certain situations it is not strictly necessary for all the packets to get through. Only the quality of information received will deteriorate. Therefore, if we assume that the packets are not retransmitted, then the penalty simply relates to the overhead of handling the extra packets plus the degradation of the quality at the receiver. There are other scenarios when certain erasure codes are used, where it is not a catastrophe not to receive certain packets, but you still pay an overhead for sending too many packets. Other cost functions could be formulated but we will consider only the above two classes of cost functions.

The \textbf{optimization problem} then is the following: Choose over successive periods the amounts \( x_t \) of packets to send, so as to minimize the total cost incurred over all periods. The amount \( x_{t+1} \) is chosen knowing the sequence \( x_1, x_2, \ldots, x_t \) and whether \( x_i \leq u_i \) or not, for each \( i = 1, 2, \ldots, t \).
The Static Case

We begin by investigating what we call the static case, where the conditions are unchanging. In the static case we assume that the threshold is fixed and is a positive integer less than or equal to a known upper bound \( n \), that is, \( u_t = u \) for all \( t \), where \( u \in \{1, 2, \ldots, n\} \). At step \( t \), \( A \) sends \( x_t \) packets and learns whether \( x_t \leq u_t \). The problem can be viewed as a Twenty Questions game in which the goal is to determine the threshold \( u \) at minimum cost by queries of the form, “Is \( x_t > u \)?” We remark that the static case is not very realistic. We thought that we would dispose of it in a few days, and move on to the more interesting dynamic case. However, it turned out that there was a lot of mathematical content even to the static case, and the problem is rather nice. We give below an outline of some of the results.

At step \( t \) of the algorithm, the sender sends an amount \( x_t \), pays a penalty \( c(x_t, u_t) \) according to whether \( x_t \) is above or below the threshold, and gets feedback telling it whether \( x_t \leq u_t \) or not. At a general step, there is an interval of pinning containing the threshold. The initial interval of pinning is the interval from 1 to \( n \). We can think of an algorithm for determining the threshold as a function from intervals of pinning to integers. In other words, for every interval of pinning \( [i, j] \), the algorithm chooses a flow \( k, (i \leq k \leq j) \) for the next interval. The feedback to this flow will tell the sender whether \( k \) was above the threshold or not. In the first case, there will be packet drops and the next interval of pinning will be the interval \( [i, k - 1] \). In the second case, the sender will succeed in sending the flow through, there will be no packet drops, and the interval of pinning at the next time interval will be the interval \( [k, j] \). We can thus think of the execution of the algorithm as a decision tree related to a twenty questions game attempting to identify the actual threshold. If the algorithm were a simple binary search, where one always picks the middle of the interval of pinning, then the tree of Figure 1 would represent the possible runs of the algorithm. Each leaf of the tree corresponds to a possible value of the threshold. Let \( A(u) \) denote the cost of the algorithm \( A \), when the

![Figure 1](image-url)
threshold is \( u \). We could be interested in the expected cost which is the average cost over all possible values of the threshold, i.e. \( 1/n \sum_{u=1}^{n} A(u) \). We could also be interested in the worst-case costs, i.e. \( \max_{1 \leq u \leq n} A(u) \). For the different cost functions defined above, ("gentle" and "severe") we will be interested in algorithms that are optimal either with respect to the expected cost or with respect to the worst-case cost.

It turns out that for an arbitrary cost function \( c(x, u) \), there is a rather simple dynamic programming algorithm with running time \( O(n^3) \), which minimizes expected cost. In some cases, an extension of dynamic programming allows one to compute policies that are optimal in the worst-case sense. So the problem is not so much computing the optimal policy for a particular value of the upper limit and of the threshold, but rather of giving a nice characterization of the policy. It turns out that for the gentle cost function family, for large \( n \), there is a very simple characterization of the optimal policies. And this rule is essentially optimal in an asymptotic sense with respect to both the expected cost and the worst-case cost.

The basic question is: Given an interval of pinning \([i, j]\), where should you put your next question, your next transmission range. Clearly, the bigger \( \alpha \) is, the higher the penalty for sending too much, and the more cautious one should be. For large \( \alpha \) we should put our trial value close to the beginning of the interval of pinning in order to avoid sending too much. It turns out that the optimal thing to do asymptotically is always to divide the interval of pinning into two parts in the proportions \( 1: \sqrt{\alpha} \). The expected cost of this policy is \( \sqrt{\alpha} n/2 + O(\log n) \) and the worst-case cost is \( \sqrt{\alpha} n + O(\log n) \). Outlined proofs of these results can be found in [1].

These results can be compared to binary search, which has expected cost \( (1 + \alpha)n/2 \). Binary search does not do as well, except in the special case where \( \alpha = 1 \), in which case the policy is just to cut the interval in the middle.

So that’s the complete story, more or less, of the gentle-cost function in the static case. For the severe-cost function, things turn out to be more challenging.

Consider the binary search tree as in Figure 2, and assume that \( n = 8 \) and the threshold is \( u = 6 \). We would start by trying to send 5 units. We would get everything through but we would pay an opportunity cost of 1. That would take us to the right child of the root. Now we would try to send 7 units. Seven is above the threshold 6, so we would overshoot and lose 6, and our total cost thus far would be 1 + 6. Then we would try 6, which is the precise threshold. The information that we succeeded would be enough to tell us that the threshold was exactly 6, and thereafter we would incur no further costs. So we see that in this particular case the cost is 7. Figure 2 below demonstrates the costs for each threshold \( u \) (denoted by the leaves of the tree). The total cost in this case is 48, the expected cost is 48/8, the worst-case cost is 10. It turns out that for binary search both the expected cost and the worst-case cost are \( O(n \log n) \).

The question is, then, can we do much better than \( O(n \log n) \)? It turns out that we can. Here is an algorithm that achieves \( O(n \log \log n) \). The idea of this algorithm is as
follows: The algorithm runs in successive phases. Each phase will have a target—to reduce the interval of pinning to a certain size. These sizes will be, respectively, \( n/2 \) after the first phase, \( n/2^2 \) after the second phase, \( n/2^4 \) after the third phase, \( n/2^8 \) after the 4\(^{th} \) phase, and \( n/2^{2^{k-1}} \) after the \( k \)-th phase. It’s immediate then that the number of phases will be \( 1 + \log \log n \), or \( O(\log \log n) \). We remark that we are dealing with the severe-cost function where there is a severe penalty for overshooting, for sending too much. Therefore, the phases will be designed in such a way that we overshoot at most once per phase.

We shall demonstrate the algorithm by a numerical example. Assume \( n = 256 \) and the threshold is \( u = 164 \). In each of the first two phases, it is just like binary search. We try to send 128 units. We succeed because 128 \( \leq \) 164. Now we know that the interval of pinning is \([128, 256]\). We try the midpoint of the interval, 192. We overshoot. Now the interval of pinning is of length 64. At the next step we are trying to reduce the interval of pinning down to 16, which is 256 over \( 2^4 \). We want to be sure of overshooting only once, so we creep up from 128 by increments of 16. We try 144; we succeed. We try 160; we succeed. We try 176; we fail. Now we know that the interval of pinning is \([160, 175]\). It contains 16 integers. At the next stage we try to get an interval of pinning of size 1. We do so by creeping up one at a time, 161, 162, etc. until we reach the correct threshold \( u = 164 \). A simple analysis shows that the cost of each phase is \( O(n) \), and since the number of phases is \( O(\log \log n) \), the cost of the algorithm is \( O(n \log \log n) \).

**A Lower Bound**

The question then is, is it possible to improve the bound \( O(n \log \log n) \)? The answer is negative as is seen in the next theorem.
**Theorem 1** \( \min_A \max_{1 \leq u \leq n} A(u) = \Theta(n \log \log n). \)

Theorem 1 claims that the best complexity of an algorithm, with a given a priori bound on the threshold \( u \leq O(n) \), is \( \Theta(n \log \log n) \). This is achievable by the algorithm described above.

There is also another result that deals with the case where no upper bound is given on the threshold. In this case, as well, a bound of \( \Theta(u \log \log u) \) is achieved for every threshold \( u \).

We shall demonstrate the idea behind the proof of the lower bound in Theorem 1. Any run of an algorithm corresponds to some path from the root to a leaf in the binary decision tree. The path contains right and left turns. A right turn means that the amount we send is less than or equal to the threshold; a left turn means that we overshoot, and the amount that we send is greater than the threshold. The left turns are very undesirable because we lose an amount equal to the threshold whenever we take a left turn. However, we also accumulate costs associated with the right turns, because we are not sending as much as we could have. We therefore have a trade-off between the number of left turns, and the cost of right turns. For threshold \( u \) denote the number of left turns in the path from root to leaf \( u \) by \( \text{leftheight}(u) \). Let \( \text{rightcost}(u) \) denote the sum of the costs accumulated in the right turns. Thus, the cost of an algorithm is given by

\[
A(u) = u \cdot \text{leftheight}(u) + \text{rightcost}(u)
\]

For example, for the path given in Figure 3 we have \( \text{leftheight}(15) = 2 \) and \( \text{rightcost}(15) = (15 - 7) + (15 - 13) + (15 - 14) = 11 \).

We define two more parameters related to the binary tree \( T \). Let \( \text{leftheight}(T) = \max_u \text{leftheight}(u) \), and \( \text{rightcost}(T) = \sum_u \text{rightcost}(u) \).

The following key lemma states that there is an inherent antagonism between minimizing the left height and the goal of minimizing the right cost.

**Lemma 1** There exists a constant \( a > 0 \) such that every \( n \)-leaf binary tree \( T \) with \( \text{leftheight}(T) \leq \log \log n \) has \( \text{rightcost}(T) \geq an^2 \log \log n \).

The proof of Theorem 1 now follows easily from Lemma 1. For details see [1].

**The Dynamic Case**

So far we have discussed the static problem, which is not entirely realistic. The static problem means that the sender is operating under constant conditions, but we don’t expect that to be the case. We expect some fluctuation in the rate available to the sender from period to period.
In the dynamic case, you can think of an adversary who is changing the threshold in such a way as to fool the sender. The problem has different forms depending on the restrictions we assume on the adversary. If the adversary can just do anything it would like in any period, then clearly the sender doesn’t have a clue what to do. So we may have various assumptions on the adversary. We can assume that the threshold $u_t$, chosen by the adversary, is simply an integer satisfying $u_t \in [a, b]$ where $a$ and $b$ are any two integers. Or we can assume that the variation of the threshold is more restricted. One such assumption that we investigated is that the adversary can drop the threshold as rapidly as it likes but can only increase the threshold from one period to the next by at most a factor, $\theta > 1$, i.e. $u_{t+1} \in [0, \theta u_t]$. Another possible assumption is that the threshold is bounded below by a positive constant $\beta$ and the adversary is additively constrained so that it can only increase the threshold by some fixed amount, $\alpha$, at most in any period, i.e. $u_{t+1} \in [\beta, u_t + \alpha]$.

As in the static case, the game is played in rounds, where in each round the algorithm sends $x_t$ packets. Unlike the static case, here we assume that the adversary chooses a sequence $\{u_t\}$ of thresholds by knowing the algorithm for choosing the sequence $\{x_t\}$ of probes. Up to this point, we have considered the cost or the loss that the sender has. Now we are going to consider the gain that the player achieves. The gain is defined as $g(x_t, u_t) = u_t - c(x_t, u_t)$, where $c(x_t, u_t)$ is the severe cost function. It is essentially the number of packets that the player gets through. The player receives feedback $f(x_t, u_t)$ which is a single bit stating whether or not the amount sent is less than or equal to the threshold for the current period.

Why are we suddenly switching from lose to gain? This is, after all, an online problem. The sender is making adaptive choices from period to period, making each
choice on the basis of partial information from the previous period. The traditional approach for analyzing online problems is of competitive analysis [2], in which the performance of an on-line algorithm for choosing \( \{x_t\} \) is compared with the best among some family of off-line algorithms for choosing \( \{x_t\} \). An off-line algorithm knows the entire sequence of thresholds \( \{u_t\} \) beforehand. An unrestricted off-line algorithm could simply choose \( x_t = u_t \) for all \( t \), incurring a total cost of zero. The ratio between the on-line algorithm’s cost and that of the off-line algorithm would then be infinite, and could not be used as a basis for choosing among on-line algorithms. For this reason it is more fruitful to study the gain rather than the loss.

The algorithm’s gain (ALG) is defined as the sum of the gains over the successive periods, and the adversary’s gain (OPT) is the sum of the thresholds because the omniscient adversary would send the threshold amount at every step.

We adopt the usual definition of a randomized algorithm. We say that a randomized algorithm achieves competitive ratio \( r \) if for every sequence of thresholds.

\[
r \cdot ALG \geq OPT + const,
\]

where \( const \) depends only on the initial conditions.

This means that, for every oblivious adversary, its payoff is a fraction \( 1/r \) of the amount that the adversary could have gotten. By an oblivious adversary we mean an adversary which knows the general policy of the algorithm, but not the specific random bits that the algorithm may generate from step to step. It has to choose the successive thresholds in advance, just knowing the text of the algorithm, but not the random bits generated by the algorithm. If the algorithm is deterministic, then the distinction between oblivious adversaries and general adversaries disappears.

We have a sequence of theorems about the optimal competitive ratio. We will mention them briefly without proofs. The proofs are actually, as is often the case with competitive algorithms, trivial to write down once you have guessed the answer and come up with the right potential function. For those who work with competitive algorithms this is quite standard.

**Adversary Restricted to a Fixed Interval**

The first case we consider is when the adversary can be quite wild. It can choose any threshold \( u_t \) from a fixed interval \([a, b]\). The deterministic case is trivial: An optimal on-line algorithm would never select a rate \( x_t > a \) because of the adversary’s threat to select \( u_t = a \). But if the algorithm transmits at the minimum rate \( x_t = a \), the adversary will select the maximum possible bandwidth \( u_t = b \), yielding a competitive ratio of \( b/a \). If randomization is allowed then the competitive ratio improves, as is seen in the following theorem:

**Theorem 2** The optimal randomized competitive ratio against an adversary that is constrained to select \( u_t \in [a, b] \) is \( 1 + \ln(b/a) \).
The analysis of this case is proved by just considering it as a two-person game between the algorithm and the adversary and giving optimal mixed strategies for the two players. The details are given in [1].

**Adversary Restricted by a Multiplicative Factor**

It is more reasonable to suppose that the adversary is multiplicatively constrained. In particular, we assume that the adversary can select any threshold $u_{t+1} \in [0, \theta u_t]$ for some constant $\theta \geq 1$. The adversary can only increase the threshold by, at most, some factor $\theta$, from one period to the next. You might imagine that we would also place a limit on how much the adversary could reduce the threshold but it turns out we can achieve just as good a competitive ratio without this restriction. It would be nice if it turned out that an optimal competitive algorithm was additive-increase/multiplicative-decrease. That this would give a kind of theoretical justification for the Jacobson algorithm, the standard algorithm that is actually used. But we haven’t been quite so lucky. It turns out that if you are playing against the multiplicatively constrained adversary, then there’s a nearly optimal competitive algorithm which is of the form multiplicative-increase/multiplicative-decrease. The result is stated below:

**Theorem 3** There is a deterministic online algorithm with competitive ratio $(\sqrt{\theta} + \sqrt{\theta - 1})^2$ against an adversary who is constrained to select any threshold $u_{t+1}$ in the range $[0, \theta u_t]$ for some constant $\theta \geq 1$. On the other hand, no deterministic online algorithm can achieve a competitive ratio better than $\theta$.

In the proof, the following multiplicative-increase/multiplicative-decrease algorithm is considered: If you undershoot, i.e. if $x_t \leq u_t$

\[
\text{then } x_{t+1} = \theta x_t,
\]

\[
\text{else } x_{t+1} = \lambda x_t, \text{ where } \lambda = \frac{\sqrt{\theta}}{\sqrt{\theta} + \sqrt{\theta - 1}}\]

It is argued in [1] that the following two invariants are maintained:

- $u_t \leq \frac{\theta}{\lambda} x_t$, and
- $rgain_t \geq opt_t + \Phi(x_{t+1}) - \Phi(x_1)$, where $\Phi(x) = \frac{1}{1 - \lambda}x$ is an appropriate potential function.

Once the right policy, the right bounds, and the right potential function are guessed, then the theorem follows from the second invariant using induction. I should say that most of this work on the competitive side was done by Elias Koutsoupias.
Adversary Restricted by an Additive Term

We consider the case where the adversary is bounded below by a positive constant and constrained by an additive term, i.e., \( u_{t+1} \in [\beta, u_t + \alpha] \). For a multiplicatively constrained adversary you get a multiplicative-increase/multiplicative-decrease algorithm. You might guess that for an additively constrained adversary you get additive-increase/additive-decrease algorithm. That’s in fact what happens:

**Theorem 4** The optimal deterministic competitive ratio against an adversary constrained to select threshold \( u_{t+1} \) in the interval \([\beta, u_t + \alpha]\) is at most \( 4 + \alpha/\beta \). On the other hand, no deterministic online algorithm has competitive ratio better than 1 + \( \alpha/\beta \).

The algorithm is a simple additive-increase/additive-decrease algorithm and again the proof involves certain inductive claims that, in turn, involve a potential function that has to be chosen in exactly the right way. For more details, consult the paper [1].

There is a very nice development that came out this somewhat unexpectedly and may be of considerable importance, not only for this problem, but also for others. I went down to Hewlett-Packard and gave a talk very much like this one. Marcello Weinberger at Hewlett-Packard asked, “Why don’t you formulate the problem in a different way, taking a cue from work that has been done in information theory and economics on various kinds of prediction problems? Why don’t you allow the adversary to be very free to choose the successive thresholds any way it likes, from period to period, as long as the thresholds remain in the interval \([a, b]\)? But don’t expect your algorithm to do well compared to arbitrary algorithms. Compare it to a reasonable class of algorithms.” For example, let’s consider those algorithms which always send at the same value, but do have the benefit of hindsight. So the setting was that we will allow the adversary to make these wild changes, anything in the interval \([a, b]\) at every step, but the algorithm only has to compete with algorithms that send the same amount in every period.

This sounded like a good idea. In fact, this idea has been used in a number of interesting studies. For example, there is some work from the 70’s about the following problem: Suppose your adversary is choosing a sequence of heads and tails and you are trying to guess the next coin toss. Of course, it’s hopeless because if the adversary knows your policy, it can just do the opposite. Yet, suppose you are only trying to compete against algorithms which know the whole sequence of heads and tails chosen by the adversary but either have to choose heads all the time or have to choose tails all the time. Then it turns out you can do very well even though the adversary is free to guess what you are going to do and do the opposite; nevertheless you can do very well against those two extremes, always guessing heads and always guessing tails.

There is another development in economics, some beautiful work by Tom Cover, about an idealized market where there is no friction, no transaction costs. He shows
that there is a way of changing your portfolio from step to step, which of course cannot
do well against an optimal adaptive portfolio but can do well against the best possible
fixed market basket of stocks even if that market basket is chosen knowing the future
course of the market.

There are these precedents for comparing your algorithm against a restricted fam-
ily of algorithms, even with a very wild adversary. I carried this work back to ICSI
where I work and showed it Antonio Piccolboni and Christian Schindelhauer. They got
interested in it. Of course, the hallmark of our particular problem is that unlike these
other examples of coin tossing and the economic market basket, in our case, we don’t
really find out what the adversary is playing. We only get limited feedback about the
adversary, namely, whether the adversary’s threshold was above or below the amount
we sent. Piccolboni and Schindelhauer undertook to extend some previous results in
the field by considering the situation of limited feedback. They considered a very gen-
eral problem, where in every step the algorithm has a set of moves, and the adversary
has a set of moves. There is a loss matrix indicating how much we lose if we play \( i \)
and the adversary plays \( j \). There is a feedback matrix which indicates how much we
find out about what the adversary actually played, if we play \( i \) and if the adversary
plays \( j \).

Clearly, our original problem can be cast in this framework. The adversary chooses
a threshold. The algorithm chooses a rate. The loss is according to whether we overshoot
or undershoot and the feedback is either 0 or 1, according to whether we overshoot or
undershoot. This is the difference from the classical results of the 1970’s. We don’t
really find out what the adversary actually played. We only find out partial information
about what the adversary played.

The natural measure of performance in this setting is worst-case regret. What it
is saying is that we are going to compare, in the worst-case over all choices of the
successive thresholds by the adversary, our expected loss against the minimum loss of
an omniscient player who, however, always has to play the same value at every step.
The beautiful result is that, subject to a certain technical condition which is usually
satisfied, there will be a randomized algorithm even in the case of limited feedback
which can keep up with this class of algorithms, algorithms that play a constant value,
make the same play at every step. This is very illuminating for our problem, but we
think that it also belongs in the general literature of results about prediction problems
and should have further applications to statistical and economic games. This is a nice
side effect to what was originally a very specialized problem.

**Acknowledgment**

The author wishes to express his admiration and sincere gratitude to Dr. Irith
Hartman for her excellent work in transcribing and editing the lecture on which this
paper is based.
References


Problems in Data Structures and Algorithms

Robert E. Tarjan
Princeton University and Hewlett Packard

1. Introduction

I would like to talk about various problems I have worked on over the course of my career. In this lecture I’ll review simple problems with interesting applications, and problems that have rich, sometimes surprising, structure.

Let me start by saying a few words about how I view the process of research, discovery and development. (See Figure 1.)

My view is based on my experience with data structures and algorithms in computer science, but I think it applies more generally. There is an interesting interplay between theory and practice. The way I like to work is to start out with some application from the real world. The real world, of course, is very messy and the application gets modeled or abstracted away into some problem or some setting that someone with a theoretical background can actually deal with. Given the abstraction, I then try to develop a solution which is usually, in the case of computer science, an algorithm, a computational method to perform some task. We may be able to prove things about the algorithm, its running time, its efficiency, and so on. And then, if it’s at all useful, we want to apply the algorithm back to the application and see if it actually solves the real problem. There is an interplay in the experimental domain between the algorithm developed, based on the abstraction, and the application; perhaps we discover that the abstraction does not capture the right parts of the problem; we have solved an interesting mathematical problem but it doesn’t solve the real-world application. Then we need to go back and change the abstraction and solve the new abstract problem and then try to apply that in practice. In this entire process we are developing a body of new theory and practice which can then be used in other settings.

A very interesting and important aspect of computation is that often the key to performing computations efficiently is to understand the problem, to represent the
problem data appropriately, and to look at the operations that need to be performed on the data. In this way many algorithmic problems turn into data manipulation problems, and the key issue is to develop the right kind of data structure to solve the problem. I would like to talk about several such problems. The real question is to devise a data structure, or to analyze a data structure which is a concrete representation of some kind of algorithmic process.

2. **Optimum Stack Generation Problem**

Let’s take a look at the following simple problem. I’ve chosen this problem because it’s an abstraction which is, on the one hand, very easy to state, but on the other hand, captures a number of ideas. We are given a finite alphabet $\Sigma$, and a stack $S$. We would like to generate strings of letters over the alphabet using the stack. There are three stack operations we can perform.

push ($A$)—push the letter $A$ from the alphabet onto the stack,
emit—output the top letter from the stack,
pop—pop the top letter from the stack.

We can perform any sequence of these operations subject to the following well-formedness constraints: we begin with an empty stack, we perform an arbitrary series of push, emit and pop operations, we never perform pop from an empty stack, and we
end up with an empty stack. These operations generate some sequence of letters over
the alphabet.

**Problem 2.1** Given some string $\sigma$ over the alphabet, find a minimum length sequence of
stack operations to generate $\sigma$.

We would like to find a fast algorithm to find the minimum length sequence of
stack operations for generating any particular string.

For example, consider the string $ABCACBA$. We could generate it by per-
forming: $\text{push (A), emit A, pop A, push (B), emit B, pop B, push (C), emit C, pop C}$ etc., but since we have repeated letters in the string we can use the same item on
the stack to generate repeats. A shorter sequence of operations is: $\text{push (A), emit A, push (B), emit B, push (C), emit C, push (A), emit A, pop A}$; now we can emit $C$ (we
don’t have to put a new $C$ on the stack), $\text{pop C, emit B, pop B, emit A}$. We got the
‘CBA’ string without having to do additional push-pops. This problem is a simplifica-
tion of the programming problem which appeared in “The International Conference on
Functional Programming” in 2001 [46] which calls for optimum parsing of HTML-like
expressions.

What can we say about this problem? There is an obvious $O(n^3)$ dynamic pro-
gramming algorithm. This is really a special case of optimum context-free language
 parsing, in which there is a cost associated with each rule, and the goal is to find a
minimum-cost parse. For an alphabet of size three there is an $O(n)$ algorithm (Y. Zhou,
private communication, 2002). For an alphabet of size four, there is an $O(n^2)$ algo-
rithm. That is all I know about this problem. I suspect this problem can be solved by
matrix multiplication, which would give a time complexity of $O(n^\alpha)$, where $\alpha$ is the
best exponent for matrix multiplication, currently 2.376 [8]. I have no idea whether
the problem can be solved in $O(n^2)$ or in $O(n \log n)$ time. Solving this problem, or
getting a better upper bound, or a better lower bound, would reveal more information
about context-free parsing than what we currently know. I think this kind of question
actually arises in practice. There are also string questions in biology that are related to
this problem.

### 3. Path Compression

Let me turn to an old, seemingly simple problem with a surprising solution. The
answer to this problem has already come up several times in some of the talks in

---

1 *Sketch of the algorithm*: Let $S[1 \ldots n]$ denote the sequence of characters. Note that there must be exactly $n$
emits and that the number of pushes must equal the number of pops. Thus we may assume that the cost is simply
the number of pushes. The dynamic programming algorithm is based on the observation that if the same
stack item is used to produce, say, $S[i_1]$ and $S[i_2]$, where $i_2 > i_1$ and $S[i_1] = S[i_2]$, then the state of the stack
at the time of emit $S[i_1]$ must be restored for emit $S[i_2]$. Thus the cost $C[i, j]$ of producing the subsequence
$S[i, j]$ is the minimum of $C[i, j-1] + 1$ and $\min \{C[i, t] + C[t+1, j-1] : S[t] = S[j], i \leq t < j\}$. 

the conference “Second Haifa Workshop on Interdisciplinary Applications of Graph Theory, Combinatorics and Algorithms.” The goal is to maintain a collection of $n$ elements that are partitioned into sets, i.e., the sets are always disjoint and each element is in a unique set. Initially each element is in a singleton set. Each set is named by some arbitrary element in it. We would like to perform the following two operations:

- **find($x$)**—for a given arbitrary element $x$, we want to return the name of the set containing it.
- **unite($x,y$)**—combine the two sets named by $x$ and $y$. The new set gets the name of one of the old sets.

Let’s assume that the number of elements is $n$. Initially, each element is in a singleton set, and after $n - 1$ unite operations all the elements are combined into a single set.

**Problem 3.1** Find a data structure that minimizes the worst-case total cost of $m$ find operations intermingled with $n - 1$ unite operations.

For simplicity in stating time bounds, I assume that $m \geq n$, although this assumption is not very important. This problem originally arose in the processing of COMMON and EQUIVALENCE statements in the ancient programming language FORTRAN. A solution is also needed to implement Kruskal’s [31] minimum spanning tree algorithm. (See Section 7.)

There is a beautiful and very simple algorithm for solving Problem 3.1, developed in the ‘60s. I’m sure that many of you are familiar with it. We use a forest data structure, with essentially the simplest possible representation of each tree (see Figure 2). We use rooted trees, in which each node has one pointer, to its parent. Each set is represented by a tree, whose nodes represent the elements, one element per node. The root element is the set name. To answer a find($x$) operation, we start at the given node $x$ and follow

![Figure 2](image_url)
the pointers to the root node, which names the set. The time of the find operation is proportional to the length of the path. The tree structure is important here because it affects the length of the find path. To perform a unite(x,y) operation, we access the two corresponding tree roots x and y, and make one of the roots point to the other root. The unite operation takes constant time.

The question is, how long can find paths be? Well, if this is all there is to it, we can get bad examples. In particular, we can construct the example in Figure 3: a tree which is just a long path. If we do lots of finds, each of linear cost, then the total cost is proportional to the number of finds times the number of elements, \(O(m \cdot n)\), which is not a happy situation.

As we know, there are a couple of heuristics we can add to this method to substantially improve the running time. We use the fact that the structure of each tree is completely arbitrary. The best structure for the finds would be if each tree has all its nodes just one step away from the root. Then find operations would all be at constant cost. But as we do the unite operations, depths of nodes grow. If we perform the unites intelligently, however, we can ensure that depths do not become too big. I shall give two methods for doing this.

**Unite by size** (Galler and Fischer [16]): This method combines two trees into one by making the root of the smaller tree point to the root of the larger tree (breaking a tie arbitrarily). The method is described in the pseudo-code below. We maintain with each root \(x\) the tree size, \(size(x)\) (the number of nodes in the tree).
\textbf{unite}(x,y): If size \((x) \geq size\ (y)\) make \(x\) the parent of \(y\) and set 
\[ \text{size}\ (x) \leftarrow \text{size}\ (x) + \text{size}\ (y) \]
Otherwise make \(y\) the parent of \(x\) and set
\[ \text{size}\ (y) \leftarrow \text{size}\ (x) + \text{size}\ (y) \]

\textbf{Unite by rank} (Tarjan and van Leeuwen [41]): In this method each root contains a \textit{rank}, which is an estimate of the depth of the tree. To combine two trees with roots of different rank, we attach the tree whose root has smaller rank to the other tree, without changing any ranks. To combine two trees with roots of the same rank, we attach either tree to the other, and increase the rank of the new root by one. The pseudo code is below. We maintain with each root \(x\) its rank, \textit{rank}(x). Initially, the rank of each node is zero.

\textbf{unite}(x, y): if rank \((x) > rank\ (y)\) make \(x\) the parent of \(y\) else
\begin{align*}
& \text{if rank } (x) < rank\ (y) \text{ make } y \text{ the parent of } x \text{ else} \\
& \text{if rank } (x) = rank\ (y) \text{ make } x \text{ the parent of } y \text{ and increase the rank of } x \\
& \quad \text{by 1.}
\end{align*}

Use of either of the rules above improves the complexity drastically. In particular, the worst-case find time decreases from linear to logarithmic. Now the total cost for a sequence of \(m\) find operations and \(n-1\) intermixed unite operations is \((m \log n)\), because with either rule the depth of a tree is logarithmic in its size. This result (for union by size) is in [16].

There is one more thing we can do to improve the complexity of the solution to Problem 3.1. It is an idea that Knuth [29] attributes to Alan Tritter, and Hopcroft and Ullman [20] attribute to McIlroy and Morris. The idea is to modify the trees not only when we do \textit{unite} operations, but also when we do \textit{find} operations: when doing a \textit{find}, we “squash” the tree along the \textit{find} path. (See Figure 4.) When we perform a \textit{find} on an element, say \(E\), we walk up the path to the root, \(A\), which is the name of the set represented by this tree. We now know not only the answer for \(E\), but also the answer for every node along the path from \(E\) to the root. We take advantage of this fact by compressing this path, making all nodes on it point directly to the root. The tree is modified as depicted in Figure 4. Thus, if later we do a \textit{find} on say, \(D\), this node is now one step away from it the root, instead of three steps away.

The question is, by how much does path compression improve the speed of the algorithm? Analyzing this algorithm, especially if both path compression and one of the \textit{unite} rules is used, is complicated, and Knuth proposed it as a challenge. Note that if both path compression and union by rank are used, then the rank of tree root is not necessarily the tree height, but it is always an upper bound on the tree height. Let me remind you of the history of the bounds on this problem from the early 1970’s.

There was an early incorrect “proof” of an \(O(m)\) time-bound; that is, constant time per \textit{find}. Shortly thereafter, Mike Fischer [11] obtained a correct bound of
Figure 4.

$O(m \log \log n)$. Later, Hopcroft and Ullman [20] obtained the bound $O(m \log^* n)$. Here $\log^* n$ denotes the number of times one must apply the $\log$ function to $n$ to get down to a constant. After this result had already appeared, there was yet another incorrect result, giving a lower bound of $\Omega(n \log \log n)$. Then I was able to obtain a lower bound which shows that this algorithm does not in fact perform in constant time per $\text{find}$. Rather, its time per $\text{find}$ is slightly worse than constant. Specifically, I showed a lower bound of $\Omega(n \alpha(n))$, where $\alpha(n)$ is the inverse of Ackermann’s function, an incredibly slowly growing function that cannot possibly be measured in practice. It will be defined below. After obtaining the lower bound, I was able to get a matching upper bound of $O(m \cdot \alpha(n))$. (For both results, and some extensions, see [37,42].) So the correct answer for the complexity of the algorithm using both path compression and one of the $\text{unite}$ rules is almost constant time per $\text{find}$, where almost constant is the inverse of Ackermann’s function.

Ackermann’s function was originally constructed to be so rapidly growing that it is not in the class of primitively recursive functions, those definable by a single-variable recurrence. Here is a definition of the inverse of Ackermann’s function. We define a sequence of functions:

For $j \geq 1$, $k \geq 0$, $A_0(j) = j + 1$, $A_k(j) = A_{k-1}^{(j+1)}(j)$ for $k \geq 1$,

where $A^{(i+1)}(x) = A(A^{(i)}(x))$ denotes function composition.

Note that $A_0$ is just the successor function; $A_1$ is essentially multiplication by two, $A_2$ is exponentiation; $A_3$ is iterated exponentiation, the inverse of $\log^*(n)$; after that the functions grow very fast.