SOLVING REAL-WORLD LINEAR PROGRAMS: A DECADE AND MORE OF PROGRESS

ROBERT E. BIXBY

ILOG, Inc. and Rice University, bixby@ilog.com or bixby@rice.edu

This paper is an invited contribution to the 50th anniversary issue of the journal *Operations Research*, published by the Institute of Operations Research and Management Science (INFORMS). It describes one person's perspective on the development of computational tools for linear programming. The paper begins with a short personal history, followed by historical remarks covering the some 40 years of linear-programming developments that predate my own involvement in this subject. It concludes with a more detailed look at the evolution of computational linear programming since 1987.

1. INTRODUCTION

I am a relative newcomer to computation. For the first half of my scientific career, my research focused exclusively on the theoretical aspects of operations research and discrete mathematics. That focus began to change in the early 1980s with the appearance of personal computers.

My first PC was used primarily to implement elementary algorithms used in teaching. At first these algorithms did not include a simplex algorithm; eventually, however, I concluded that it would be useful to incorporate computation in the LP courses that I was teaching. As a result, I started writing my own code, initially a simple tableau code.

At that time, in the early 1980s, I knew nothing about the computational aspects of linear programming (LP). I knew a great deal of theory, but numerical analysis and the computational issues associated with numerical algorithms were not subjects that were part of my graduate education. I had no idea that tableaus were numerically unstable.

Fortunately for me, by the time my interests in computation had started, the Department of Industrial Engineering and Management Sciences at Northwestern University had hired Bob Fourer, one of the creators of the AMPL modeling language. Bob had worked for several years at the National Bureau of Economic Research doing practical linear programming, followed by a graduate career at Stanford. He knew a lot about the computational aspects of mathematical programming, and he passed on a great deal of that knowledge to me in informal conversations.

Linear programming become more central to what I was doing when a friend of mine, Tom Baker, founded Chesapeake Decision Sciences (now a part of Aspen Technologies). Shortly thereafter, Tom asked if I had an LP code that he could use in the LP module of the product he was building. I said yes, converted my code to C (that was one of Tom's conditions), and delivered it to him.

To this day, I'm not quite sure why Tom thought my code would eventually be reasonably good. Initially it certainly was not.

After the code was delivered to Chesapeake, there followed a period of about two years during which I received a steady stream of practical LPs from Chesapeake, LPs on which my code did not do very well. In each case, I poked around in my code and the LP itself to see what ideas I could come up with, never looking in the literature (this wasn't my area of research). Slowly the code got better, until some time around 1986, one of Tom's colleagues informed me that my code had actually gotten good enough that one of their customers was interested in obtaining it separately. I was, to say the least, surprised, and immediately set about doing my first actual comparisons to other LP codes. I chose Roy Marsten's (1981) quite successful and portable (that was key for me) XMP code. I discovered, to my amazement, that for a substantial subset of the *netlib*[1] testset my code was indeed pretty good, running on average two times faster than XMP. In addition, it appeared that my code was significantly more stable than XMP.

This comparison to XMP was an important part of what transformed LP computation into a serious part of my scientific research. Equally important was integer programming.

This was the mid-1980s, and integer-programming computational research was beginning to flower, with important contributions by people such as Martin Grötschel, Ellis Johnson, Manfred Padberg, and Laurence Wolsey. Linear programming was an essential component in that work, but the tools available at that time were proving to be inadequate. The then state-of-the-art codes, such as MPSX/370, simply were not built for this kind of application; in addition, they did not deal well with issues such as degeneracy. The situation at the time is well described by some remarks of Grötschel and Holland (1991), commenting on their use of MPSX/370 in work on the traveling salesman problem: They note that if the LP-package they were using had been

Subject classification: Professional: comments on
Area of review: Anniversary Issue (Special).

"better suited for a row generation process than MPSX is, the total speed-up obtained by faster (cut) recognition procedures might be worth the higher programming effort," (p .174) and "Some linear programs that arose were hard to solve, even for highly praised commercial codes like IBM's MPSX" (p. 142).

What was needed was a numerically robust code that was also flexible enough to be embedded in these integer-programming applications. It had to be a code that made it easy to handle the kinds of operations that arose in a context in which it was natural to begin with a model instantiated in one form followed by a sequence of problem modifications (such as row and column additions and deletions and variable fixings) interspersed with resolves. These needs were among the fundamental motivations behind the development of the callable-library version of the CPLEX[2] code.

The connection between linear and integer programming also offered me the opportunity to work on something that would not only be of potential commercial value, but would fit nicely with a research program in integer programming. This research program later grew to include one of my most fruitful research collaborations, joint work with David Applegate, Vasek Chvátal, and Bill Cook on the traveling salesman problem.

2. HOW FAR HAVE WE COME?

It was thus around 1987 that I became seriously involved in the computational aspects of linear programming. The first version of CPLEX, CPLEX 1.0, was released in 1988. A lot has happened in the field since then. The question I would like to address here—as quantitatively as possible—is, how have the developments during this period affected our ability to solve real-world LPs? The size and magnitude of the real models that are regularly solved today was, I believe, unimaginable 10 years ago.

It will come as no surprise that in most of what I present, I will make use of the CPLEX LP code. However, it is my sense that similar improvements could be demonstrated using other modern LP codes. Much of what I will discuss is based upon technological advances that have benefitted LP in general.

While the focus of this paper will be on linear programming, a few comments are in order on integer programming. Integer programming, and most particularly the mixed-integer variant, is the dominant application of linear programming in practice. Integer programming makes direct use of all the advances we will discuss in LP algorithms. In addition, there have been other major advances that are domain specific to integer programming, such as the use of cutting planes and integer-programming-specific presolve techniques. These two classes of methods alone often transform models from being unsolvable to straightforward. There is little doubt that the overall improvement in present-day integer-programming codes exceeds that for linear programming.

It will be assumed throughout this paper that the reader has a general familiarity with LP algorithms, particularly the primal and dual simplex algorithms and the primal-dual log barrier algorithms. For a general reference on simplex algorithms see Chvátal (1983), and for a more detailed discussion of some of the specific computational issues discussed here see Applegate et al. (forthcoming). For a reference on barrier algorithms see Wright (1997).

3. ADVANCES IN COMPUTING MACHINERY

We all recognize that advances in computing machinery have had an enormous effect on the practical application of linear programming. Without computing machines, linear programming as we know it would not exist.

During the period since 1987, the influence of computing advances has been particularly strong. Indeed, it is sometimes asserted that machine advances are the main reason that linear programming has become such a powerful tool. While it is my conviction that algorithmic and software improvements have been equally important, there is no doubt that hardware effects are large and pervasive. Today's desktop computers have reached the stage that their power exceeds by a considerable factor that of even the best supercomputers available just 10 years ago.

Beyond simply the issue of speed, huge increases in computer memory capacity have made it possible to handle much larger problems, and have made it possible to consider entirely different solution strategies and implementations of these strategies. Many of the fundamental algorithmic ideas that are key to modern LP codes simply could not have been implemented if memory were not so plentiful. The improvements in computer programming languages and systems have made it much simpler to build large complex systems. Improvements in computer-human interfaces have improved not only the usability of the tools that we create, but greatly facilitated the creation of the tools themselves, including the basic improvements in the underlying algorithms. The insights for many of these improvements have come from the simple ability to examine larger and more interesting real instances in real time.

3.1. Speed Comparison

A direct speed comparison is clouded by the differences between barrier algorithms and simplex algorithms. In barrier algorithms there is a single computational step that usually dominates: the computation of the Cholesky factorization. There is no such single step that dominates in simplex computations. In addition, the computation of the Cholesky factorization is very regular; we are much more able to exploit the capabilities of modern computing architectures than in simplex algorithms.

For these reasons, I give separate estimates of machine speed improvements for simplex algorithms and barrier algorithms. To estimate simplex speedups I make use of computational results from studies carried out in 1988 using CPLEX 1.0 combined with results on the same

Table 1. Machine improvements—simplex algorithms.

Old Machine/ Processor	New Machine/Processor	Estimated Speedup
Sun 3/50	Pentium 4, 1.7 GHz	800
Sun 3/50	Compaq Server ES40, 667 MHz	900
25 MHz Intel 386	Compaq Server ES40, 667 MHz	400
IBM 3090/108S	Compaq Server ES40, 667 MHz	45
Cray X-MP/416	Compaq Server ES40, 667 MHz	10

models using CPLEX 1.0 compiled and run on current machines. For barrier algorithms, the standard *Linpack* benchmarks[3] provide a good measure of the effects of machine improvement.

A Sun 3/50 was the machine on my desk in late 1987. I do have computational results for Intel processors available at that time, but the usability of machines based upon these processors was severely hampered by their 16-bit, 640K memory limit. The 25 MHz 386 listed in Table 1 was the first of the PC processors available to me where these limitations could be overcome, using so-called 32-bit "DOS extenders."

The IBM 3090 is included here because, into the mid 1980s, these machines were typical of the mainframes that dominated LP practice. It is worth noting that the simplex speedup listed is surely an overestimate of the speedup relative to a code such as MPSX. The C compilers for the IBM 3090 were not very good; moreover, the CPLEX code took no account of the special properties of the 3090 architecture. MPSX, by contrast, was written largely in machine assembly code and tuned to the specifics of the 3090 architecture.

The final machine listed, the Cray X-MP, was never in wide use as an LP computing environment. However, significant testing was carried out on these machines in the late 1980s and early 1990s, and they do illustrate the upper limit of computing power available at that time.

What I conclude from Tables 1 and 2 is that for desktop computing, machine speedups have contributed a factor between 500 and 1,000 to the speed of simplex algorithms. Barrier algorithms, on the other hand, have experienced speedups an order of magnitude greater. This difference is fundamental to the fact that barrier algorithms have emerged as a powerful computational tool in linear programming.

Table 2. Machine improvements—barrier algorithms.

Old Machine/ Processor	New Machine/Processor	Estimated Speedup
Sun 3/50	Pentium 4, 1.7 GHz	13000
Sun 3/50	Compaq Server ES40, 667 MHz	12000
25 MHz Intel 386	Compaq Server ES40, 667 MHz	4000
IBM 3090/108S	Compaq Server ES40, 667 MHz	10
Cray X-MP/416	Compaq Server ES40, 667 MHz	5

4. LP COMPUTATION: 1947–LATE 1980s

George Dantzig is widely recognized as the father of linear programming. A central part of his many contributions to this subject was the recognition that linear programming was more than simply a conceptual tool. It was important to be able to solve linear programs and compute actual answers:

> A certain wide class of practical problems appears to be just beyond the range of modern computing machinery. These problems occur in everyday life; they run the gamut from some very simple situations that confront an individual to those connected with the national economy as a whole. Typically, these problems involve a complex of different activities in which one wishes to know which activities to emphasize in order to carry out desired objectives under known limitations (Dantzig 1948).

LP computation began with Dantzig's introduction of the simplex method in 1947.[4] The above quotation is taken from the paper in which, to my knowledge, the simplex algorithm first appeared. Perhaps the first instance of a nontrivial LP solved with the simplex algorithm was Laderman's solution (see Dantzig 1963) of Stigler's (1945) diet problem. This LP had nine constraints and 77 variables. Reportedly, nine coworkers working on electronic calculators for an estimated total of 120 man-days were needed to carry out the computations.

The first computer implementation of the simplex method seems to have been developed at the National Bureau of Standards, the present-day National Institute of Standards and Technology, on the SEAC computer. Orden (1952) and Hoffman et al. (1953) report computational tests with this machine. One instance with 48 equations and 71 variables was solved in 18 hours and 73 simplex iterations.

William Orchard-Hays began his pioneering work on implementations of the simplex method in 1953–54. This work was the beginning of the development of commercially available LP codes. The computing machine used was an IBM "card programmable calculator" (CPC), hardly a real computer by today's standards. In the words of Orchard-Hays (1990) "The CPC was an ancient conglomeration of tabulating equipment, electro-mechanical storage devices, and an electronic calculator (with tubes and relays), long since forgotten. One did not program in a modern sense, but wired three patch-boards which became like masses of spaghetti." The first code implemented by Orchard-Hays used an explicit basis inverse, with the inverse freshly recomputed at each iteration. Again, in the words of Orchard-Hays, "One could have started an iteration, gone to lunch, and returned before it finished..." The initial results were not encouraging. However, in 1954 Dantzig recalled the idea of the product-form of the inverse, proposed by Alex Orden, and this device led to a second, more efficient CPC implementation. Orchard-Hays (1990) reports that the largest instance solved with this code had 26 constraints and 71 variables, and took "eight hours

of hard work feeding decks to hoppers" to complete the solution.

As computers continued to get better, so did implementations of simplex algorithms. In 1956 a code named RSLP1 was implemented on an IBM 704, a machine with 4K of core storage, miniscule by today's standards. The maximum number of constraints was limited to 255, with an essentially "unlimited" number of variables. RSLP1 was distributed through the SHARE organization and was used by larger petroleum companies. In the period 1962–1966 LP/90 was implemented for the IBM 7090, followed by LP/90/94 for the IBM 7094. The number of allowed constraints grew to 1,024. By then the use and application of LP had grown as well.

In 1966, IBM introduced a major advance in computing hardware, the family of IBM 360 computers. At about the same time, the development of an LP system designed to run on these computers was commissioned by IBM. That system became known as MPS/360. MPS/360 was followed by MPSX and later by MPSX/370. During that period, motivated by Kalan's (1971) work on supersparsity, Ketron Corporation developed MPS III with Whizard; this work was supported by Exxon Corporation through the urging of Milt Gutterman. Tuned for the new, fast IBM platforms, these codes, most particularly MPS III, represented quantum leaps both in speed and problem size, accepting models with up to 32,000 constraints. Other powerful systems were also developed during that period, including the UMPIRE system for the UNIVAC 1108, APEX-III for CDC machines, and in the mid to late 70s LAMPS, written independently by John Forrest. With modest updates and machine improvements, these systems were the dominant LP computing environments into the late 1980s.

5. LP COMPUTATION: 1987–PRESENT

I will begin with a summary of important algorithmic ideas. This summary will be followed by computational results comparing LP performance in 1987 with LP performance today, first in anecdotal form, and then through more extensive tests carried out on several different collections of LP models. The final and by far most extensive of these will involve a testset with 680 models, the largest of which has more than 6 million constraints.

5.1. Algorithmic Improvements

Some of the topics discussed here, such as primal-dual log barrier algorithms, will be known to all readers. Others, such as "bound shifting," a device used in primal simplex algorithms, will be less known. For a detailed presentation of the simplex-specific parts of this discussion, see Applegate et al. (forthcoming).

The Dual Simplex Algorithm with Steepest Edge. The dual simplex algorithm is not new. It was introduced by Lemke (1954). However, to my knowledge, commercial implementations of this algorithm were not available in 1987 as full-fledged alternatives to the primal simplex algorithm. As examples, MPSX/370 and MPS III[5] as well as APEX-III[6] did include rudimentary implementations of the dual algorithm, but these implementations included no Phase I facility for dealing with infeasibilities.

All that has changed. The dual simplex algorithm is now a standard alternative in modern codes. Indeed, computational tests, some of which will be presented later in this paper, indicate that the overall performance of the dual algorithm may be superior to that of the primal algorithm.

There are a number of reasons why implementations of the dual simplex algorithm have become so powerful. The most important is an idea introduced by Goldfarb (1976), a so-called "steepest-edge" rule for selecting the "leaving variable" at each dual simplex iteration. Dual steepest-edge "version 1" as described in Forrest and Goldfarb (1992)[7] requires relatively little additional computational effort per iteration and is far superior to "standard" dual methods, in which the selection of the leaving variable is based only upon selecting a basic variable with large primal infeasibility.

Linear Algebra. Linear algebra improvements touch all the parts of simplex algorithms and are crucial to good barrier implementations as well. Enumerating all such improvements is beyond the scope of this paper. I will mention only a few.

For simplex algorithms, two improvements stand out among the rest. The first of these is dynamic LU-factorization using Markowitz threshold pivoting. This approach was perfected by Suhl and Suhl (1990), and has become a standard part of modern codes. In previous-generation codes, "preassigned pivot" sequences were used in the numerical factorization (see Hellerman and Rarick 1971). These methods were very effective when no numerical difficulties occurred, but encountered serious difficulties in the alternative case.

The second major linear-algebra improvement is that LP codes now take advantage of certain ideas for solving large, sparse linear systems, ideas that have been known in the linear-algebra community for several years (see Gilbert and Peierls 1988). At each major iteration of a simplex algorithm, several sizeable linear systems must be solved. The order of these systems is equal to the number of constraints in the given LP. Typically these systems take as input a vector with a very small number of nonzero entries, say between one and 10—independent of overall model size—and output a vector with only a few additional nonzeros. Since it is unlikely that the sparsity of the output is due to cancellation during the solve, it must be that only a small number of nonzeros in the LU-factorization (and update) of the basis were touched during the solve. The trick then is to carry out the solve so that the work is linear in this number of entries, and hence, in total, essentially a constant time operation, even as problem size grows. The effect on large linear programs can be enormous.

For primal-dual log barrier algorithms, as previously noted, one computation typically dominates the time per

iteration, the computation of the Cholesky factorization of AA^T, where A denotes a matrix obtained by a column scaling of the constraint matrix of the given LP. A crucial part of this computation is a preprocessing step, carried out in advance of the actual barrier solve, in which the rows of A are ordered so that an associated symbolic factorization of AA^T will realize as little numerical "fill" as possible, where fill refers to the additional nonzeros created by the factorization process.

While ordering techniques for minimizing fill in Cholesky factorizations have been under study for years, long before barrier algorithms became important for linear programming, the kinds of matrices that arise in linear programming are quite different from those that had previously been studied. Subsequent advances in this domain have been crucial to the present-day performance of barrier algorithms. See Rothberg and Hendrickson (1998) for a description of the state of the art.

Primal-Dual Log Barrier Algorithms. Karmarkar's (1984) paper started a virtual revolution in the theory of linear programming. It also played a major role in prompting the computational developments (see Lustig et al. 1994) that led to present-day primal-dual log barrier algorithms for LP.

As we shall see, for the CPLEX implementation, barrier algorithms have emerged as overall the most powerful single algorithm for solving LPs. Moreover, for a wide range of problem dimensions, these algorithms can be very effectively parallelized, further increasing their advantage. Simplex algorithms have not been successfully parallelized in this sense. However, in spite of these facts, one property continues to severely limit the importance of barrier algorithms in practice: their inability to replicate the performance of simplex algorithms when resolving an LP from an advanced starting point.

The limited resolve capabilities of barrier algorithms effectively limit their application in the domain of integer programming to solving the initial LP relaxation, thus leaving the majority of the work to simplex algorithms. Since the solution of integer programs is the dominant application of linear programming in practice, simplex algorithms remain the dominant algorithms in practice. In addition, even for very large models, where the advantages of barrier algorithms tend to grow, simplex algorithms are still the winning approach in a significant number of cases.

Other Ideas. *Presolve.* This idea is made up of a set of problem reductions: removal of redundant constraints, fixed variables, and other extraneous model elements. The seminal reference on presolve is Brearley et al. (1975).

Presolve was available in MPS III, but modern implementations include a much more extensive set of reductions, including so-called aggregation (substituting out variables, such as free variables, the satisfaction of the bounds of which are guaranteed by the satisfaction of the bounds on the variables that remain in the model). The effects on problem size can be very significant, in some cases yielding reductions by factors exceeding an order of magnitude. Modern presolve implementations are also seamless in the sense that problem input and output occur in terms of the original model, with exactly the same solution information being provided as if the full model had been solved.

Perturbation, Shifting, and the Harris Ratio Test. The previous generation of LP developers were fully aware of the problems posed by degeneracy, stalling (long sequences of degenerate pivots), and cycling, at least in theory, but, by and large, chose not to implement explicit procedures to deal with these phenomena. The effects of degeneracy did not seem to pose a serious threat to performance for the size and difficulty of models being solved. However, as difficulty has increased, such measures have become essential. Research in integer programming accelerated this process by often focusing on structures with combinatorial origins in which degeneracy was particularly pervasive.

In the CPLEX primal simplex implementations, the key ideas for dealing with degeneracy are an expanded two-pass ratio test introduced by Paula Harris (1974), a bound-shifting idea (which temporarily expands bounds that become violated during the application of a simplex algorithm), and an associated notion of bound perturbation that is applied simultaneously to all bounds of nonbasic variables, if it is determined that the first two measures (the expanded ratio test and shifting) have not led to sufficient progress during optimization.

Corresponding ideas are used in dual simplex algorithms, but applied to objective-function coefficients rather than problem bounds.

Hybrid Pricing. Hybrid pricing, or variable selection, is a technique used by CPLEX primarily in the primal simplex algorithm. Two alternative pricing techniques that work well in practice are some form of *partial pricing*, in which one attempts to examine only a small subset of potential entering variables at each iteration, and *devex* pricing, introduced by Harris (1974), a relatively inexpensive form of approximate steepest-edge pricing.

Partial pricing typically does well on easier LPs or at the beginning of an optimization when good, potential entering variables are plentiful. Devex pricing does better on more difficult models and near the end of an optimization, but incurs more cost at each iteration. Hybrid pricing is any scheme that begins an optimization using partial pricing (or some other inexpensive scheme), and switches to devex (or some other more powerful, more expensive scheme) later in the optimization. The result is a more robust version of the primal simplex algorithm.

5.2. Performance Improvements: Examples

Unless otherwise stated, in this and subsequent sections the computing platform used was a 667 MHz Compaq Server ES40. All reported solution times are in CPU seconds.

EXAMPLE 1: **A Fleet-Assignment Model.** The size statistics for this model, `degen4`, are displayed below, followed

by a set of solution times in Table 3. The order of the results in Table 3 roughly represents the chronology of events that eventually led to the effective solution of degen4.

Model:	degen4
Rows	4420
Columns	6711
Nonzeros	101377

The degen4 model is a larger version of the *netlib* models degen2 and degen3, and is much more difficult. It is an early instance of an airline fleet-assignment model. In late 1989, degen4 was presented as a challenge problem to optimizers and computer vendors.

In my first attempt at solving degen4, working together with John Gregory, then at Cray Research, I tried using CPLEX 1.0 on a Cray Y-MP. This attempt failed miserably. After seven hours of computing, the solution was still infeasible, and had been stuck on the same objective value for several hours. Largely because of the effects of degeneracy, it appeared very unlikely that one could solve this model with CPLEX 1.0. As it turned out, this conclusion was probably incorrect. If running on a Cray hadn't been so expensive, and there hadn't been so many competing users, it now seems likely that degen4 would eventually have solved.

Running CPLEX 1.0 on a current, fast workstation (a 667 MHz Compaq Server ES40), with no competing users, degen4 solved after about 1.5 days of computing and over 26 million iterations. This run did illustrate clearly the potential effects of stalling. For one period in the computation, lasting some 4,897,095 iterations, the objective appeared to remain identically constant. As the results in Table 3 indicate, by simply introducing a perturbation (in this case a pseudorandom perturbation reducing each lower bound by 1.0E-5 multiplied by a uniform random $[0, 1]$ variable), CPLEX 1.0 was able to solve degen4, taking about one-half hour on a present-day machine.

Following the above attempt, degen4 was successfully solved using the OB1 barrier code of Lustig et al. (1994). The solution time was about 20 minutes on a Y-MP, corresponding roughly to the CPLEX 2.2 barrier time in Table 3. The conclusion of some at the time was that barrier was the right way to solve this model, and that there was little point in further investigating the use of simplex algorithms. That conclusion was wrong.

CPLEX 2.2 included the measures described in the previous section for dealing with degeneracy—shifting and perturbation—and also included a hybrid primal pricing algorithm using devex. The result was a greatly reduced number of iterations (about 50,000 total) and a running time that was at least competitive with the barrier approach. We then finally recognized what turned out to be the key idea behind solving this model correctly: Look at the dual. By explicitly constructing the dual problem and applying the default primal algorithm, all traces of degeneracy disappeared, and the solution time was reduced to a time much smaller than that for the barrier algorithm. This observation was among the important motivations for implementing dual simplex algorithms in CPLEX.

However, simply dualizing degen4 was not the end of the story. As one can see from Table 3, using the dual alone, with "textbook" variable selection, choosing as the leaving variable the basic variable with the largest primal infeasibility, resulted in performance significantly worse than simply applying the primal to the explicit dual. The reason was that maximum-infeasibility variable selection just didn't work very well, and doesn't work very well in general. The primal was using devex in this case, a version of steepest edge. What was missing was steepest edge for the dual. That final piece of the puzzle was provided by Forrest and Goldfarb (1992), who introduced a particularly effective approach to steepest-edge pricing for the dual. This modification not only works well on degen4, but in general. It is one of the key reasons why the dual simplex algorithm has emerged as a powerful all-purpose algorithm for linear programming.

EXAMPLE 2: **The PDS Models.** These models are described in Carolan et al. (1990). They are military logistics models; "PDS" stands for patient-distribution system. The smaller instances, from pds02 through pds20, are now part of the standard *netlib* testset.

The PDS models have an underlying multicommodity flow structure, often a source of difficult LP problems. When first introduced, they were considered very difficult indeed. Taking CPLEX as a measure, solving pds20 on a 1990's vintage workstation would have taken an estimated 40 days, and solving pds70 would have been practically impossible (see Table 5).

Table 3. Solution times—degen4.

Version	Remark	Seconds
CPLEX 1.0	primal (default)	119364.0
CPLEX 1.0	perturbation	1545.7
CPLEX 2.2	barrier	125.8
CPLEX 2.2	primal (default)	170.2
CPLEX 2.2	primal on explicit dual	33.4
CPLEX 2.2	dual standard pricing	102.3
CPLEX 2.2	dual (default)	12.0

Table 4. PDS models.

Instance	Rows	Columns	Nonzeros
pds100	156171	546469	1193533
pds90	142823	507771	1112089
pds80	129181	467192	1025706
pds70	114944	422356	929346
pds60	99431	367268	809094
pds50	83060	304348	671605
pds40	66844	242649	536690
pds30	49944	177628	393657
pds20	33874	119438	265793
pds10	16558	52712	118283
pds06	9881	28655	62524
pds02	2953	7535	16390

Because of the difficulty of these models, they have received considerable attention in the LP literature, and several special-purpose algorithms have been developed. To my knowledge, the most recent and best of these algorithms is described in Castro (2000). The largest model solved by Castro was pds90, with a solution time of 21,781 seconds on a 200 MHz UltraSparc. As we shall see, Castro's algorithms are now dominated by current general-purpose implementations of the dual simplex algorithm.

In Table 5, run times are given for three versions of CPLEX, starting with CPLEX 1.0. All runs were made on a 300 MHz Sun UltraSparc. Note that CPLEX 1.0 times are missing for the largest of the models. These runs were omitted because of their anticipated length.

Significant progress did occur between CPLEX 1.0 and CPLEX 5.0. I have included solution times only for the dual, but solution times for the primal are similar. This progress was due largely to the effects of presolve (CPLEX 1.0 had no presolve) and the use of strong pricing algorithms (steepest-edge in the dual and hybrid pricing in the primal).

The most recent improvements, between Versions 5.0 and 7.1, dwarf earlier improvements and transform the PDS models, once considered difficult, into easy LPs. CPLEX 7.1 performance on these models is due to the fact that CPLEX 7.1 fully exploits sparsity, uses more aggressive perturbation in the dual (treating this idea as an algorithmic technique rather than simply a remedy for degeneracy), and also benefits from an idea called "bound flipping." See Applegate et al. (forthcoming) for a description. Note that the primal simplex method has also benefited from properly exploiting sparsity.

While not illustrated in the above table, standard barrier algorithms have also become a viable approach for the largest PDS instances. Current ordering algorithms are able to directly exploit multicommodity-like structures in constraint matrices: CPLEX 5.0 barrier solves pds20 in 880.8 seconds, while CPLEX 7.1 takes only 69.3 seconds, an improvement of over a factor of 10.

I would also like to note the effect of problem size on the relative improvements from CPLEX 1.0 to 7.1, measured as the ratio of the CPLEX 7.1 dual solve time to the CPLEX 1.0 (primal) solve time. Results are given in Table 6.

Table 5. PDS models—solution times.

Instance	CPLEX 1.0	CPLEX 5.0 Dual	CPLEX 7.1 Primal	CPLEX 7.1 Dual
pds100	—	50413.1	2414.8	256.3
pds90	—	59981.0	2452.2	320.3
pds80	—	42055.4	2201.5	304.4
pds70	335292.1	21120.4	1504.1	197.8
pds60	205798.3	7442.6	852.4	160.5
pds50	122195.9	8509.9	493.2	114.6
pds40	58920.3	2816.8	188.3	79.3
pds30	15891.9	1154.9	74.8	39.1
pds20	5168.8	232.6	27.9	20.9
pds10	208.9	13.0	3.7	2.6
pds06	26.4	2.4	1.4	0.9
pds02	0.4	0.1	0.1	0.1

For the largest models in this set, the performance improvements well exceed machine improvements for this time period, while for smaller models this is not the case. Ratios for the very largest PDS instances would likely be larger yet. Indeed, I performed a related test on one additional model studied by Castro (2000), mnetgen24. This model has 66,641 constraints, 370,739 variables, and 1,039,461 nonzeros and a multicommodity structure. CPLEX 7.1 dual solved this model in 114.5 seconds, while CPLEX 1.0 took 1,221,920.3 seconds (a little over two weeks) on the same machine, a ratio of over 10,672.

Finally, I would like to make an observation about the rate of growth of the solution times as the size of the PDS model increases.[8] Consider Figure 1. In this figure the solution times for CPLEX 1.0 and CPLEX 7.1 are plotted as a function of the "number" of the corresponding PDS model, where the number for pds02 is '2', the number for pds06 is '6', and so on. These numbers are a measure of the number of distinct time periods, or "blocks" in the underlying model.

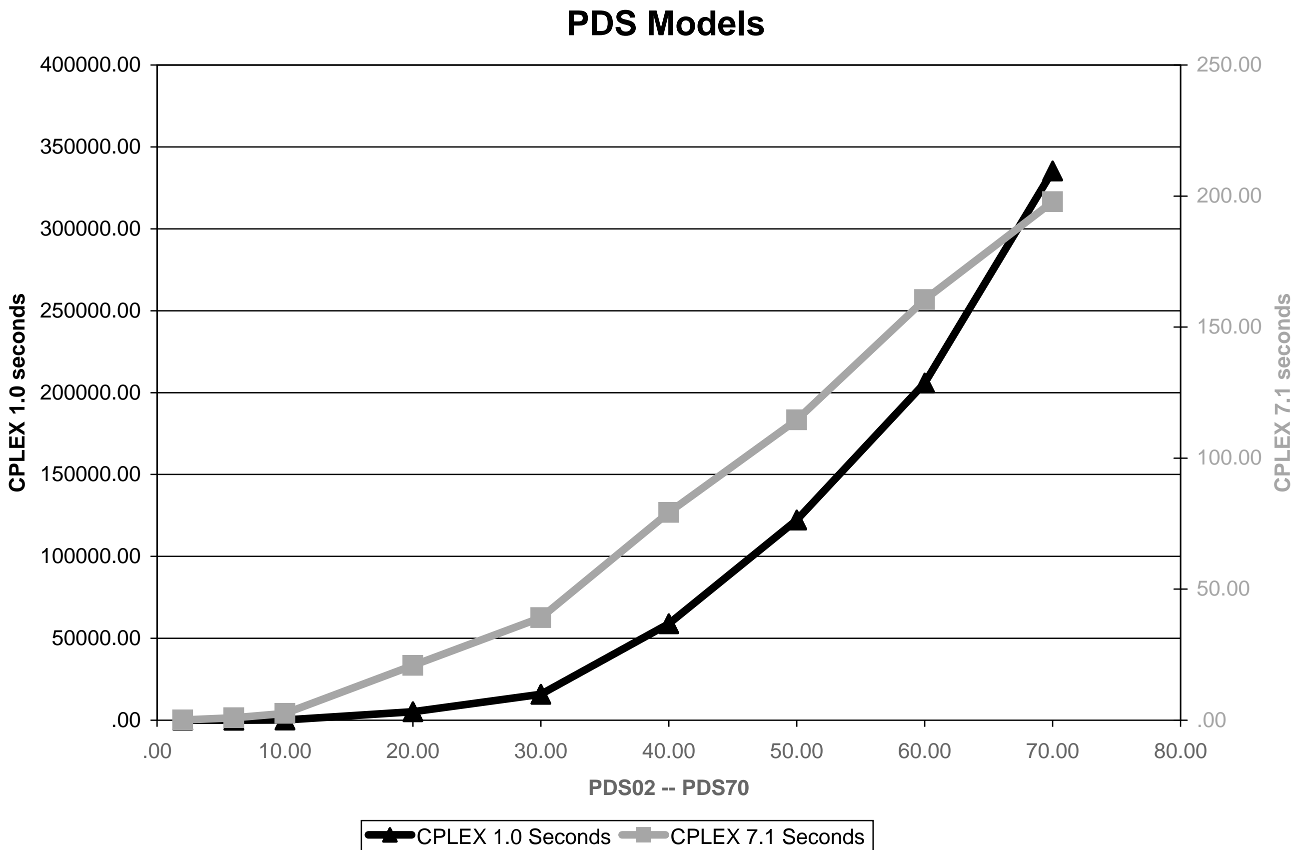
It is clear from the shapes of the graphs in Figure 1 that the solution times grow at different rates for the different CPLEX versions. This point can be made more precise by applying straight line and parabolic least-squares fits to the data. The adjusted R^2 for the CPLEX 1.0 data when a straight line is used is 0.796, but for a parabola the adjusted R^2 is 0.994. Thus, the parabolic fit is noticeably better, explaining 99.4% of the variance in solution times, while the straight-line fit only explains 79.6% of the variance. In the case of CPLEX 7.1, however, the straight-line fit adjusted R^2 is 0.954, indicating that the simple linear fit alone explains over 95% of the variance in the solution times.

Based upon the above analysis, the growth rates of solution times for CPLEX 1.0 and CPLEX 7.1 are qualitatively different: One appears to be quadratic and the other, CPLEX 7.1, nearly linear. This phenomenon may be explained as follows. Note first that a solution time may be viewed as the product of the number of iterations and the time per iteration. Beginning with iteration counts, it is quite reasonable to expect a growth rate that is linear with the number of blocks, or at least approximately so. What about the time per iteration? In older versions of CPLEX

Table 6. PDS models—relative improvements.

Model	CPLEX 1.0	CPLEX 7.1–Dual	Ratio
pds70	335292.1	197.8	1695.1
pds60	205798.3	160.5	1282.2
pds50	122195.9	114.6	1066.3
pds40	58920.3	79.3	743.0
pds30	15891.9	39.1	406.4
pds20	5168.8	20.9	247.3
pds10	208.9	2.6	80.3
pds06	26.4	0.9	29.3
pds02	0.4	0.1	4.0

Figure 1. PDS models.



this growth rate was at best linear. Thus, the overall growth rate was quadratic, at best. By contrast, the newer linear-algebra routines in CPLEX 7.1, exploiting the ideas in Gilbert and Peierls (1988), can avoid unnecessarily examining zeros during solves. This change implies that one might expect the time per iteration for a one-block model to be about the same as when many blocks are present, yielding times that are essentially constant as the number of blocks grows. The result is overall solution times that grow nearly linearly.

Example 3: **Primal**. We recently received a customer LP generated to test the viability of a new modeling approach. Size statistics before and after presolve are as follows:

Very large model

	Rows	Columns	Nonzeros
Original size	5034171	7365337	25596099
After presolve	1296075	2910559	10339042

Solution times were as follows:

Very large model—solution times

	Algorithm		
Version	Barrier	Dual	Primal
CPLEX 5.0	8642.6	350000.0	71039.7
CPLEX 7.1	5642.6	6413.1	1880.0

The barrier algorithm did not see a substantial improvement version to version, but the dual algorithm improved by a factor of 54.6 and the primal by a factor of 37.8. The effect of presolve on the solution times for this problem is also substantial. While the CPLEX 7.1 primal run took only 131,016 iterations using presolve (the default), running on the unpresolved model the same code had completed over 875,000 iterations in slightly over 15,000 seconds, and the solution was far from even being feasible. A corresponding run with CPLEX 1.0 completed only 2,500 iterations in a similar time period, a factor of about 350 times slower per iteration running on the unpresolved model. Clearly, while hard to estimate with any accuracy, the ratio of the best CPLEX 7.1 solve time to the potential CPLEX 1.0 solve time is easily in the thousands.

Example 4: **Barrier**. It is not difficult to find examples where barrier algorithms dominate the best of the available simplex algorithms. These instances obviously represent models where algorithmic improvements have been the difference between solving and not solving. One particularly striking example is the following car manufacturing model:

Model:	`CARS`
Rows	196400
Columns	205040
Nonzeros	604060

CARS solved in 583 seconds using barrier followed by crossover to a basic solution, where the crossover step consumed 481 of these seconds. Neither CPLEX 7.1 primal nor CPLEX 7.1 dual finished solving this model in 350,000 seconds.

5.3. Performance Improvements: Larger Testsets

How should one carry out a systematic comparison of LP technology in 1987 with that of today? Perhaps the ideal approach would be to put together a large, representative testset of models, run these models with some appropriate code, vintage 1987, run the same models with a present-day code, and compare the results. Indeed, I do have access to an excellent testbed of models, and these could be the basis for such a test.

However, as we saw in the case of the PDS models, speedups can be substantially greater for larger models, and many of these models, even with the best codes now available, take several thousands of seconds to solve. If one is to solve the same models with older codes and really expect to see the full effect of the speedups—some of which would likely exceed four orders of magnitude—then run times could easily range to significant fractions of a year. Carrying out such a program is clearly impractical. As an alternative, I will present a sequence of results making comparisons that move forward in time. These results will, I believe, make a convincing case when taken as a whole.

Where should the comparison begin? With what code? I could choose to use as a baseline a state-of-the-art code from the late 1980s, MPSX/370 or MPX III. However, there are several reasons why I believe that would be the wrong approach. Leaving aside software design issues that are critical to the current state of linear-programming practice (such as portability and embeddability), the simple speed comparison is of limited interest. These solvers were tied to mainframe computing, and those mainframes were no more than 50 times slower than current workstations. The results presented below will easily demonstrate that the algorithmic improvements between CPLEX 1.0 and CPLEX 7.1 exceed a factor of 50. Given that test results from 1989 (not presented here) comparing MPSX/370 and CPLEX 1.0 both running on an IBM 3090 showed comparable running times on several more difficult models, the conclusion is clear.

The code that I have chosen for the starting point of my comparisons is the XMP code of Roy Marsten (1981). Because it was portable and embeddable, XMP was heavily used in the late 1980s in integer-programming research. While certainly not comparable in speed to MPSX or MPS III, it was in some ways more advanced algorithmically, including state-of-the-art factorization and factorization-update routines. MINOS, developed by Murtagh and Saunders (1998), was another portable code available in the same period (and still available, in much improved form). It was significantly more stable than XMP, and somewhat faster (based upon comparisons I did at that time with CPLEX 1.0), but not as easy to embed.

Besides XMP and MINOS, there were already several PC codes on the market in the late 1980s; however, I have seen no evidence to suggest that the performance of any of these codes significantly exceeded that of XMP.

Table 7 is taken from a talk I gave in 1988 at Columbia University. It compares CPLEX 1.1[9] to XMP on a subset of the *netlib* testset. Runs were made on a Sun 3/50. Two models that were included in the original test runs do not appear in the table, grow22 and pilotnov, both of which prematurely terminated due to numerical singularities when running XMP.

One possible comparison from this table is the ratio of the total solution times. These yield a speedup of approximately 4.3 (= 15979.2/3733.3). Another possible comparison is to compute the ratio of each individual CPLEX time divided by the corresponding XMP time and compute the arithmetic mean of these ratios. The result is an average ratio of 9.6. However, both of these measures are overly sensitive to single, large entries, either in the total solve time or in the ratios. A much better, more robust, and more conservative measure is the geometric mean of the individual ratios. The geometric mean in this case is 4.70.

Table 7. A 1988 comparison.

	CPLEX 1.1		XMP	
Model	Iterations	Seconds	Iterations	Seconds
share2b	103	3.0	138	6.6
bore3d	115	6.3	5801	736.6
standat	179	9.0	75	8.2
seba	187	11.9	1433	244.4
sc205	191	11.1	273	23.0
share1b	197	9.9	411	28.8
scorpion	227	13.8	429	44.9
brandy	241	20.8	8521	1144.0
forplan	249	17.9	2313	244.5
israel	300	25.6	242	27.0
capri	329	15.8	550	48.0
bandm	374	41.1	1679	250.9
e226	421	26.5	653	71.3
stair	446	134.6	1667	530.9
sierra	459	49.7	950	270.0
scagr25	508	46.9	1470	248.9
shell	530	32.2	830	123.3
gfrd-pnc	613	43.9	983	144.1
ganges	701	108.8	1769	587.6
sctap3	795	92.8	3404	1293.6
scrs8	812	81.1	1271	182.7
fffff800	834	66.4	1611	297.2
etamacro	883	55.8	1140	144.6
scfxm3	1018	133.7	2860	864.8
ship12l	1035	144.2	1510	515.7
czprob	1138	137.8	3014	936.3
scsd8	1890	230.7	1818	346.7
nesm	3810	413.8	7255	1844.4
25fv47	4559	1742.3	10859	4753.2

Table 8. Model sizes—original.

Model	Rows	Columns	Nonzeros
car	43387	107164	189864
continent	10377	57253	198214
energy1	16223	28568	88340
energy2	8258	21200	145329
energy3	27145	31053	268153
fuel	18800	38540	219880
initial	27441	15128	96118
schedule	23259	29342	75520

It is obvious, but worth noting, that the models used in comparing XMP and CPLEX are trivial in size by today's standards. It is my view that they likely lead to an underestimate of the difference between XMP and CPLEX 1.0, and they are certainly too small to yield any useful information about current codes: The total running time for the entire set on a 667 MHz Compaq Server ES40 is 3.5 seconds with CPLEX 1.0 and 2.7 seconds with CPLEX 7.1.

For a second comparison, I will use a testset from a study that was published in Bixby (1994), motivated by Lustig et al. (1994), comparing CPLEX 1.0 to CPLEX 2.2 on a set of models that at that time were considered quite difficult and quite large. The original sizes and names of these models are given in Table 8. The sizes after application of CPLEX 7.1 presolve are given in Table 9.

Tables 10 and 11 give solution times for the models in Table 8 using CPLEX 1.0, 2.2, 5.0, and 7.1. Runs were made on a 300 MHz UltraSparc. In the first table I have tabulated the best of the primal and dual solution times for each of the eight models and for each of the CPLEX versions. The final column specifies which algorithm was the winner for each of the eight models running CPLEX 7.1. The second table records the best of all three algorithms —barrier, primal, and dual—with the final column again recording the winners for Version 7.1.

Table 12 compares CPLEX 1.0 to the various other versions using geometric means of individual ratios of solve times. According to this table, the best simplex algorithm in CPLEX 7.1 is almost 52 times faster than CPLEX 1.0 on these models, and the best of three is 114 times faster.

A shortcoming of the testset in Table 8 is that these models no longer are "large." In addition, a single algorithm, barrier, is dominant.[10] To construct a more comprehensive, less biased measure of recent improvements, I will use a larger, more comprehensive testset, and focus on comparing only two CPLEX versions, CPLEX 5.0 and 7.1. CPLEX 5.0 was the last release prior to introducing a number of improvements for exploiting sparsity in large models.

For my final testset I have made use of the CPLEX library of LPs, a library that has been collected over the last 13 years from industry and academia. In total it contains approximately 2,000 distinct models. Since many of these models represent multiple instances with differing sizes but identical structure, I began by screening the set to remove some of these multiple instances, generally keeping only the largest two or three from a given set. In addition, all models were removed from the set that solved in under 0.25 seconds with both CPLEX 5.0 and CPLEX 7.1 and all three default algorithms, barrier, primal, and dual. The set that remained contained 680 models.

Rather than presenting an entire table of model statistics, I offer the summary statistics in Table 13, indicating the numbers of models in various ranges of row counts. Row count is the best simple predictor of model difficulty that I have found. I ran the six different default algorithms on all 680 models using a 350,000-second time limit (about four days): barrier (with crossover), dual, and primal for both CPLEX 5.0 and CPLEX 7.1. All runs were made on 667 MHz Compaq Server ES40s.

There were three models, one with approximately 20,000, one with 50,000, and one with 1,300,000 rows, that did not solve with any of the algorithms inside the time limit. These models are omitted in further comparisons. It

Table 9. Model sizes–after presolve.

Model	Rows	Columns	Nonzeros
car	32194	73512	145019
continent	6808	45728	157812
energy1	10470	19262	68799
energy2	6553	17899	126438
energy3	9464	28649	185988
fuel	8732	21313	149129
initial	18913	10788	78567
schedule	5044	12176	37828

Table 10. Solution times—best simplex.

Model	CPLEX 1.0	CPLEX 2.2	CPLEX 5.0	CPLEX 7.1	Algorithm
car	1555.0	701.1	275.8	120.6	primal
continent	364.7	110.5	104.4	46.7	primal
energy1	1217.4	275.0	260.5	22.6	dual
energy2	10130.1	736.0	664.0	693.9	dual
energy3	21797.1	271.9	229.1	161.7	dual
fuel	5619.5	1123.2	698.6	675.0	primal
initial	3832.2	102.2	51.3	15.5	dual
schedule	152404.0	252.3	220.8	64.6	dual

Table 11. Solution times—best of three.

Model	CPLEX 1.0	CPLEX 2.2	CPLEX 5.0	CPLEX 7.1	Algorithm
car	1555.0	203.0	117.1	67.3	barrier
continent	364.7	110.5	99.5	46.7	primal
energy1	1217.4	46.5	31.5	22.4	barrier
energy2	10130.1	171.4	71.7	32.4	barrier
energy3	21797.1	152.6	113.4	82.2	barrier
fuel	5619.5	999.1	340.5	124.7	barrier
initial	3832.2	102.2	51.3	15.5	dual
schedule	152404.0	252.3	132.0	47.9	barrier

Table 12. Ratios—geometric means.

Version	Best Simplex	Best of Three
CPLEX 1.0	1.0	1.0
CPLEX 2.2	15.8	30.3
CPLEX 5.0	22.0	54.0
CPLEX 7.1	51.8	114.1

should be noted, however, that each of these models did solve with barrier alone, omitting crossover; moreover, the solutions appeared to be of high quality. However, omitting crossover is not the default when using barrier in CPLEX. Indeed, crossover is considered an integral part of this algorithm, being invoked in a significant number of cases to complete the optimization of nonoptimal barrier solutions, including models that are declared infeasible.

The results of the indicated runs, excluding the three models mentioned above, are summarized in Table 14. To see what these numbers mean, consider the "Best simplex" column. To compute it, the best of the primal and dual simplex running times were extracted for CPLEX 5.0 and CPLEX 7.1, respectively, producing two lists of 677 running times each. From these two lists, 677 ratios were computed by dividing the best simplex time for each model using CPLEX 5.0 by the best simplex time for that model using CPLEX 7.1. To then compute an individual entry such as that for "⩾25,000 ," the geometric mean was computed of all ratios for the models with at least 25,000 rows, of which there were 182. The result was 3.7, indicating an average of almost a four-fold speedup.

What can one conclude from the results of this section? I have claimed an improvement of 4.7 for CPLEX 1.0 relative to XMP, both codes using exclusively the primal simplex algorithm. Using the results of the eight problems taken from Bixby (1994), one can conclude an improvement in simplex algorithms from CPLEX 1.0 to CPLEX 5.0 of 22.0, yielding a total of approximately 103 from XMP to CPLEX 5.0, for problems of moderate size. Now using the comparison of CPLEX 5.0 to CPLEX 7.1 for models with, for example, 50,000 rows and more, one obtains a total speedup that can be estimated at 960, roughly the same as the magnitude of machine improvements for simplex algorithms.

Including barrier algorithms in this analysis, one obtains an estimated improvement of 250 from XMP to CPLEX 5.0, and for models with 50,000 rows and more a total improvement of approximately 2,400. Really quite remarkable. However, the proper way to compare this improvement with machine improvements is far less clear, given the huge machine effect enjoyed by barrier algorithms.

I would like to close with a brief comparison among the three core algorithms, using CPLEX 7.1 alone. Table 15 was extracted from the same set of test runs as Table 14.

The results in Table 15 indicate that dual simplex is about twice as fast as primal simplex, on average, and that barrier is, overall, the fastest algorithm by a narrow but growing margin as problem size increases. Interestingly, taking the best of primal and dual simplex performance yields an algorithm with average performance roughly the same as barrier. However, it is important to note that the numbers in Table 15 do measure only average performance. A detailed examination of the data indicates that each of primal, dual, and barrier wins in a significant number of cases, an observation confirmed by the fact that the best of three outperforms each individual algorithm by a significant margin.

Table 13. Big testset—summary statistics.

Row-count Range	Number
0–999	93
1000–2499	124
2500–4999	86
5000–9999	88
10000–24999	105
25000–49999	70
50000–99999	40
100000–249999	39
250000–499999	15
500000–6662791	20

Table 14. CPLEX 5.0 vs. CPLEX 7.1.

Row Range	Best Simplex	Best of Three	Primal	Dual	Barrier
⩾0 rows	2.0	2.3	2.0	2.7	2.1
⩾1000 rows	2.3	2.5	2.2	3.0	2.2
⩾2500 rows	2.7	3.1	2.6	3.5	2.5
⩾5000 rows	3.1	3.7	3.1	4.2	4.0
⩾10000 rows	3.7	4.8	4.0	5.5	3.6
⩾25000 rows	5.0	6.6	5.7	7.4	4.7
⩾50000 rows	6.7	9.3	8.3	9.9	6.0
⩾100000 rows	7.0	9.4	9.5	9.6	7.2
⩾250000 rows	10.6	15.6	19.1	14.9	8.4
⩾500000 rows	10.6	20.4	24.8	23.0	7.2

Table 15. Algorithm comparison.

Row Range	Primal/ Dual	Dual/ Barrier	Best Simplex/ Barrier	Primal/ Best of Three
⩾0 rows	1.5	1.1	0.9	3.3
⩾1000 rows	1.6	1.1	0.9	3.5
⩾2500 rows	1.7	1.0	0.8	3.7
⩾5000 rows	1.8	1.1	0.8	4.1
⩾10000 rows	2.0	1.0	0.8	4.4
⩾25000 rows	2.0	1.2	0.9	5.1
⩾50000 rows	2.0	1.4	1.0	6.8
⩾100000 rows	2.1	1.6	1.1	8.5
⩾250000 rows	1.6	1.7	1.1	7.0
⩾500000 rows	2.5	1.5	0.9	8.9

6. CONCLUSION

In this paper I have focused primarily on one issue, solving larger, more difficult linear programs faster. The numbers presented speak for themselves. Three orders of magnitude in machine speed and three orders of magnitude in algorithmic speed add up to six orders of magnitude in solving power: A model that might have taken a year to solve 10 years ago can now solve in less than 30 seconds. Of course, no one waits one year to solve a model, at least no one I know. The real meaning of such an advance is much harder to measure in practice, but it is real nevertheless. There is no doubt that we now have optimization engines at our disposal that dwarf what was available only a few years ago, making possible the solution of real-world models once considered intractable, and opening up whole new domains of application.

How do these speed improvements fit into the overall picture of linear-programming practice? They are only a part of that picture, though an essential, enabling part. The pervasive availability of powerful, usable desktop computing, the availability of data to feed our models, and the emergence of algebraic modeling languages to represent our models have all combined with the underlying engines to make operations research and linear programming the powerful tools they are today. However, there are still important issues to be solved. In spite of all the advances, the application of linear programming remains primarily the domain of experts. The need for abstraction still stands as a hurdle between technology and solutions. While the existence of this hurdle is disconcerting, it is at least gratifying to know that the benefits from overcoming it are now greater than ever.

ENDNOTES

[1] See `http://netlib.lucent.com/netlib/lp/data/index.html`.

[2] CPLEX is a trademark of ILOG, Inc.

[3] See ⟨`http://www.netlib.org/performance/html/PDSbrowse.html`⟩.

[4] See Gass (2001) for a further discussion of early LP computational work.

[5] John Tomlin, private communication.

[6] John Gregory, private communication.

[7] The dual steepest-edge algorithm in Goldfarb (1976) is "version 3" in Forrest and Goldfarb (1992).

[8] This observation was suggested to the author by Randy Batsell, a Rice University professor in the Jesse H. Jones Graduate School of Management.

[9] The simplex implementations in CPLEX 1.1 were essentially identical to those in CPLEX 1.0.

[10] Lustig et al. (1994) point out that seven of these models were primarily of interest because they appeared difficult to solve with simplex algorithms.

ACKNOWLEDGMENTS

It would be remiss of me not to mention here the other people who have played a significant role in the development of CPLEX. Zonghao Gu, Irv Lustig, Ed Rothberg, and Roland Wunderling have all made fundamental contributions to the CPLEX linear-programming algorithms. Mary Fenelon, John Gregory, and Ed Klotz, while not contributing directly to these algorithms, deserve considerable credit for persistent and significant insights. I would also like to acknowledge Janet and Todd Lowe for transforming a collection of algorithms into a successful commercial product.

REFERENCES

Applegate, D., R. Bixby, V. Chvátal, W. Cook. *Solving Traveling Salesman Problems*. Forthcoming.

Bixby, R. E. 1994. Commentary: Progress in linear programming. *ORSA J. Comput.* **6** 15–22.

Brearley, A. L., G. Mitra, H. P. Williams. 1975. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Math. Programming* **8** 54–83.

Carolan, W. J., J. E. Hill, J. L. Kennington, S. Niemi, S. J. Wichmann. 1990. An empirical evaluation of the KORBX algorithms for military airlift applications. *Oper. Res.* **38**(2) 240–248.

Castro, J. 2000. A specialized interior-point algorithm for multicommodity network flows. *SIAM J. Optim.* **10**(3) 852–877.

Chvátal, V. 1983. *Linear Programming*. Freeman, New York.

Dantzig, G. 1948. Programming in a linear structure. U.S. Air Force Comptroller, USAF, Washington, D.C.

——. 1963. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ.

Forrest, J. J., D. Goldfarb. 1992. Steepest-edge simplex algorithms for linear programming. *Math. Programming* **57** 341–374.

Gass, S. I. 2002. The first linear-programming shoppe. *Oper. Res.* **50** 61–68.

Gilbert, J. R., T. Peierls. 1988. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.* **9** 862–874.

Goldfarb, D. 1976. Using the steepest-edge simplex algorithm to solve sparse linear programs. *Sparse Matrix Computations*. Academic Press, 227–240.

Grötschel, M., O. Holland. 1991. Solution of large-scale symmetric travelling salesman problems. *Math. Programming* **51** 141–202.

Harris, P. M. J. 1974. Pivot selection methods of the devex LP code. *Math. Programming* **5** 1–28.

Hellerman, E., D. Rarick. 1971. Reinversion with the preassigned pivot procedure. *Math. Programming* **1** 195–216.

Hoffman, A., M. Mannos, D. Sokolowsky, D. Wiegmann. 1953. Computational experience in solving linear programs. *SIAM J.* **1** 1–33.

Kalan, J. E. 1971. Aspects of large-scale in-core linear programming. *Proc. ACM Conf.* Chicago, IL 304–313.

Karmarkar, N. 1984. A new polynomial-time algorithm for linear programming. *Combinatorica* **4** 373–395.

Lemke, C. E. 1954. The dual method of solving the linear programming problem. *Naval Res. Logist. Quart.* **1** 36–47.

Lustig, I. J., R. Marsten, D. F. Shanno. 1994. Interior point methods for linear programming: Computational state of the art. *ORSA J. Comput.* **6**(1) 1–14.

Marsten, R. E. 1981. XMP: A structured library of subroutines for experimental mathematical programming. *ACM Trans. Math. Software* **7** 481–497.

Murtagh, B. A., M. A. Saunders. 1998. MINOS 5.5 User's Guide. Report SOL 83-20R, Dept of Operations Research, Stanford University, Stanford, CA.

Orchard-Hays, W. 1990. History of the development of LP solvers. *Interfaces* **20**(4) 61–73.

Orden, A. 1952. Solution of systems of linear inequalities on a digital computer. *Proc. ACM*.

Rothberg, E., B. Hendrickson. 1998. Sparse matrix ordering methods for interior point linear programming. *INFORMS J. Comput*. **10**(1) 107–113.

Stigler, G. J. 1945. The cost of subsistence. *J. Farm Econom.* **27**(2) 303–314.

Suhl, U. H., L. M. Suhl. 1990. Computing sparse LU factorizations for large-scale linear programming bases. *ORSA J. Comput.* **2** 325–335.

Wright, S. J. 1997. *Primal-Dual Interior-Point Methods*. SIAM, Philadelphia, PA.