Progress in Academic Computational Integer Programming*

Thorsten Koch[1], Alexander Martin[2], and Marc E. Pfetsch[3]

[1]Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany, koch@zib.de
[2]Friedrich-Alexander Universität Erlangen-Nürnberg,, Cauerstr. 11, 91058 Erlangen, Germany, alexander.martin@math.uni-erlangen.de
[3]TU Darmstadt, Research Group Optimization, Dolivostr. 15, 64293 Darmstadt, Germany, pfetsch@opt.tu-darmstadt.de

September 17, 2016

Abstract

This paper discusses issues related to the progress in computational integer programming. The first part deals with the question to what extent computational experiments can be reproduced at all. Afterwards the performance measurement of solvers and their comparison are investigated. Then academic progress in solving mixed-integer programming at the examples of the solver SIP and its successor SCIP is demonstrated. All arguments are supported by computational results. Finally, we discuss the pros and cons of developing academic software for solving mixed-integer programs.

1 Introduction

The field computational integer programming deals with the computational aspects of integer and combinatorial optimization. Ever since the paper of Dantzig, Fulkerson, and Johnson [33] it has been clear that one main goal would be to actually compute (optimal) solutions and that practical considerations would have a major influence on the evolution of this field.

This article discusses the developments that have been made in the last 20 years with and through academic research and software. The main point is to highlight important issues that arise when implementing, testing, and benchmarking mixed-integer programming (MIP) software. For concreteness, we use the MIP-solvers SIP and SCIP as examples, see [67] and [3, 71]. Consequently, this article is written from the personal perspective and experience of the authors, which allows us to present more details compared to just a general overview. We believe that many parts will find their analogies with other solvers (both commercial and academic). All three authors were involved in the development of SCIP or its ancestor SIP and were or are still working at the group of Martin Grötschel at the University of Augsburg and at the Zuse Institute Berlin (ZIB). In fact, the authors are lucky to have lead the integer programming group for some time.

One main motivation for this article is the growing importance of computer driven experiments and how to draw scientifically meaningful conclusions from them in the

*Published in “Facets of Combinatorial Optimization”, pp. 483–506, Michael Jünger, Gerhard Reinelt (Eds.), Springer, 2013

area of computational integer programming in particular. Indeed, it seems that the publication standards of papers involving computations have not yet been fully fixed – in contrast to other physical sciences that are based on experiments. Actually, Hooker [49] called for a new paradigm to evaluate experimental results obtained on the computer, and Greenberg [42] already discussed standards of publication – both, from our point of view, without much effect. Our article tries to bring (back) into focus several issues that one has to be aware of in the context of computational integer programming.

It seems to be important to note that most issues that we discuss are not primarily of a mathematical nature, but they are necessary to come to scientifically sound conclusions when evaluating mathematical ideas. Clearly, these topics are at the intersection of different fields like computer science, operations research, engineering, and mathematics. Consequently, researchers from all these fields – mathematicians, in particular – have to be aware of the loopholes, traps, and organizational issues that are related to such computations. We see our article as one contribution in this direction.

We explicitly mention mathematicians here, because the authors have heard several times throughout their careers that implementation would be an issue for engineers/computer scientists and should not be performed by mathematicians. We do not agree and believe in the interdisciplinary viewpoint stated above.

We begin with a brief historical review of the developments during the last two decades related to SIP and SCIP in order to put things into a perspective and – in a sense – to document its achievements. Then we discuss the question whether the computational experiments that have been performed in the past can be reproduced at all. This is a topic that is often neglected in the literature and that, we think, deserves more attention. The next section deals with our experiences of benchmarking MIP-solvers. In the final section, we discuss issues related to the development of academic software in this context.

2 Historical Overview

This section briefly reviews the last 20 to 30 years in computational integer programming; it focuses on academic developments, and, in particular, those at ZIB.

As mentioned above, we decided to largely neglect other (academic) software developments in this article. The following references give more balanced surveys on the available software: Atamtürk and Savelsbergh [17], as well as Linderoth and Lodi [61], give an overview of MIP-solvers, while Linderoth and Ralphs [62] focus on non-commercial MIP-solvers. For an overview of linear programming (LP) solvers, see Fourer [39]. More details and a discussion of possible future research topics can be found in the survey of Lodi [64].

2.1 General Developments starting in the 1980s

Possibly the first major step for the development of *integer programming* solvers was the seminal paper by Crowder, Johnson, and Padberg [31], which introduces many concepts that are still part of modern MIP-solvers. In the context of combinatorial optimization, the development of *branch-and-cut* (B&C) algorithms turned out to be very influential, based, e.g., on the article by Grötschel, Jünger, and Reinelt [43] on the linear ordering problem. From a computational perspective, the paper of Padberg and Rinaldi [70] on the solution of the traveling salesman problem (TSP) was a major step. Apart from the introduction of many important components like the cut pool, it also coined the name "branch-and-cut". At that time it became common to look for an $\mathcal{NP}$-hard combinatorial problem, investigate its facets, and then implement a specialized B&C algorithm to solve it. The algorithms consisted of a separation procedure for the identified problem-specific facets and valid inequalities plus some

branching scheme.

At ZIB, a large number of specialized B&C solvers were implemented in the late 1990s, e.g., [28, 29, 35, 44, 45, 57, 75], to mention just a few.

People generally implemented their own branch-and-bound framework using one of the available out-of-the-box LP-solvers like OSL, MINOS, CPLEX, or XPRESS. General cutting planes like Gomory cuts were not considered to be useful in practice. For MIP-solving this only changed with the rediscovery of Gomory's mixed integer cuts by Balas et al. [18], which made B&C approaches dominant also for MIP-solving.

The main focus of the B&C algorithms for combinatorial optimization was on separating, and the number of branch-and-bound nodes that had to be enumerated was relatively small (or the instance could not be solved anyway due to the computer standards of the time). This fits well with the statement that *branching is a sign of mathematical defeat*, which is attributed to Manfred Padberg.

Several general out-of-the-box MIP-solvers like OSL, CPLEX, or XPRESS existed, but apart from their versatility, these were inferior to the special implemented algorithms.

2.2 MIP-Solving at ZIB

The work of the integer programming group at ZIB started in the early 90s, after Martin Grötschel had moved to Berlin. Beginning in 1992, Cray Research had funded a project to develop a general parallel B&C framework to solve large combinatorial optimization problems. This framework was targeted at the then state-of-the art distributed memory Cray T3D computer. The project was lead by Christian Hege and conducted by Roland Wunderling and Martin Grammel. Their assumption was that the central part of any B&C solver is the simplex algorithm, and therefore the first goal of the project was to develop a fast parallel distributed memory simplex algorithm.

When in 1996 Roland Wunderling finished his PhD thesis [76] on the LP-solver library SoPlex (sequential object-oriented simplex), two things could be concluded:

1. It was possible to write a simplex-based LP-solver that matched the performance of the commercial implementations at that time.
2. Developing a (distributed memory) parallel simplex-based LP-solver is not an especially promising idea (see also [26]). The work on the distributed memory B&C framework ceased even some time before.

One of the main features of SoPlex was the exploitation of the sparsity of the constraint matrix. In most linear and integer programming problems, the matrix A is very sparse, see [65]. There are some exceptions in which specialized algorithms for dense linear algebra computations are needed, see, e.g., [34]; remarkably, this area never received much attention. Despite the enormous speed-up that linear programming achieved in practice as reported by Bixby [22], there is only a small number of articles that together seem to contain everything important on how to implement a simplex based LP-solver: [32, 38, 37, 47, 58, 60, 72, 73].

At the end of the 1990s, the performance of general out-of-the-box MIP-solvers tremendously improved. One main influencing action was *mining the literature*, as Bob Bixby, the founder of CPLEX and Gurobi, called it. At this point a significant amount of theoretical results, in particular, on cutting planes, had been published that were not utilized in the out-of-the box (commercial) MIP-solvers. By incorporating these insights, it was possible to enormously improve the performance of general MIP-solvers, and for the first time it became hard to beat them by special implementations. More and more (practical) problems could be modeled and solved directly without further programming. As the solvers evolved, it became common to use them as frameworks for specialized algorithms by only extending the basic solver, instead of implementing a complete B&C algorithm from scratch.

In 1994, Alexander Martin started to develop the general MIP-solver SIP (Solving Integer Programs). When he finished his habilitation treatise in 1998, two things could be concluded:

1. At that time it was possible to write a B&C based MIP-solver that matched the performance of commercial implementations: SIP was comparable in performance to CPLEX at the time, see [67], although a notable disadvantage was that it used CPLEX as embedded LP-solver.
2. It became clear that implementing (shared memory) parallel MIP-solvers had some potential, but proved to be difficult. The reason for this is that the backbone of MIP-solvers was (and still is) the (dual) simplex algorithm. Moreover, one particular problem was the insufficient memory bandwidth of the available machines.

When Alexander Martin moved to TU Darmstadt in 2000, Tobias Achterberg and Thorsten Koch continued the work on SIP at ZIB. In particular, it was interfaced with SoPlex, making SIP completely available in the source code for the first time.

A main component of MIP-solvers are rules to choose the branching variable in each node of the tree. One key idea is strong branching, which was invented in the 1990s, see [14, 15, 51]. Before actually branching on some variable, its value is tested by temporarily fixing the variable to its up and down value and comparing the resulting LP relaxations. This is obviously time-consuming, but currently the best choice in terms of the number of branch-and-bound nodes. In 2004, the state-of-the-art w.r.t. solving time was the so-called *pseudo cost branching*, an idea that was already developed in the 70s, see [19, 67, 63], which tries to "learn" from previous branching decisions and constructs artificial costs of the variables that hopefully reflect their merit for branching. In 2005, a dynamic combination of both methods, the so-called *reliability branching*, was developed, see [8]. A variation of it is still state-of-the-art, see Achterberg and Berthold [4]. This was a major step forward, since the main handicap of *pseudo cost branching* is that, especially in the beginning when the branching decision is most influential, no additional information is available, and *most infeasible branching* was used; this missing information at the beginning is dynamically supplied by strong branching. As experiments revealed, most infeasible branching does not perform better than branching randomly.

Around 2003, MIPs were generalized by incorporating concepts from constraint programming leading to so-called *Constraint Integer Programming*, see [2] for an exact definition. This was motivated by an industry project with Infineon on the verification in chip design. Since it became clear that the basic infrastructure of SIP was very much tailored towards solving MIPs, Tobias Achterberg began to develop SCIP (Solving Constraint Integer Programs) as part of his PhD thesis, see [2, 5]. He also extended SCIP by using SAT solving techniques, e.g., restarts and conflict analysis.

Since then, SCIP has been continuously developed and improved. Apart from the added functionality with respect to constraint programming, SCIP has been one of the fastest non-commercial MIP-solvers, see, e.g., Mittelmann [69], and used in numerous areas – often beyond classical MIP-solving; examples are:

- column generation and decomposition of MIPs [40]
- constraint programming and conflict analysis [1, 3, 6],
- counting solutions [7, 48],
- MIP-solver technology [4, 10, 21],
- mixed-integer nonlinear programming [27, 74],
- pseudo-Boolean optimization [20],
- semidefinite programming [16, 66],
- symmetries in integer programs [52, 53].

In 2007, the first version of the ZIB Optimization Suite was released, which integrated SCIP as CIP-solver, SoPlex as solver for the LP-relaxations, and Zimpl [55, 54] as modeling environment. The source codes are freely available for academic usage.

Thus, a complete state-of-the-art solver environment is available in the source code for usage, improvement, and teaching.

In 2012, the third major version of the now called SCIP Optimization Suite has been released. It integrates, for example, a framework for distributed parallel computations and substantially extended the functionality to solve mixed-integer nonlinear programs.

However, the size of the suite (altogether about 800,000 lines of code) also demonstrates that computational integer programming has clearly evolved beyond the point where an individual can just sit down and implement a state-of-the-art solver as a PhD thesis. A large number of complex and involved components are needed, and their integration is a major issue. We will discuss this further in Section 6.

When looking back over the last 25 years of work on the topic, the question that comes up is *How much progress has been made?* There is the well-known article by Bixby [22] that determines at least a million times speed-up for linear programming during 15 years and an article by Achterberg and Wunderling [11] investigating the improvements in MIP solving. Can we make similar conclusions regarding (academic) integer programming? To answer this question we have to compare the performance of different MIP solvers over time. But in order to compare results, they have to be available or at least reproducible, an issue addressed in the next section. The question on how to compare results is then discussed in Section 4.

3 Reproducibility of Computational Results

One of the foundations of scientific research is that experiments should be reproducible. The key question is what kind of reproducibility one actually requires. This generally varies over different natural sciences. The results of an experiment reproduced by independent researchers with the same or similar material and experimental set-up are generally accepted, where the degree of agreement of the results depends on the field. However, this is complicated in such cases when the investigated material is destroyed during measurements as it may happen in biology, for instance. In areas that use computers to perform experiments reproducibility has been discussed for some time, see, e.g., [77, 68]. Clearly, this is highly relevant for computational integer programming. It seems, however, that we have not yet reached a generally accepted agreement on the kind of reproducibility that is needed or wanted. One guideline would be that at least the same/similar conclusions could be drawn from similar experiments - a statement that must be made more precise. In fact, one goal of this section is to provide an example from computational integer programming that illustrates the difficulties of reproducibility in this area.

There are four basic options for reproducing results of an algorithmic idea published in the literature:

- use the published results;
- run an old code on old machines;
- recompile and run an old code on new machines, possibly using new libraries;
- reimplement the published method.

Clearly, every option has severe drawbacks: Assuming that the new code is run on a new computer it is obviously very difficult to compare running times across different machines. (We do not know of any computational experiment with a new algorithm that has been run on purpose on a much older machine.) While this might suffice to get a very crude estimation, there is one major obstacle to this approach: The sample sizes especially of the older articles are too small for today's standards. As it can be seen in [11], one needs several hundred and more instances to actually be able to measure smaller performance changes. Thus, we actually need to run the published algorithm on more instances than have been published in order to derive a sound comparison. Running an old code on old or new machines comes with complicated technical problems, as we shall illustrate below. This leaves us to reimplement the

published method. But this last option is often practically too time consuming. One issue is that a reimplementation does not provide any scientific merit. Moreover, it moves the burden of supplying a good implementation of an algorithmic idea to the one that is performing the comparison. If the old idea performs badly, it might be due to a bad implementation.

We do not have good solutions for these issues, but in the following we will investigate how precisely we can reproduce old results and use the particular example of reproducing results from SIP to highlight the problems involved. Clearly, to provide a historic perspective like in this paper, only the first three options above are significant.

As one goal of this article we would like to show the progress achieved in the field and thus compare results of current solvers with previously published ones. As reference we will take Table 5.1 given on page 75 of [67], where the results of solving a set of instances using SIP 1.1 are listed and copied in Table 1. Is it possible to reproduce those results? Our observations concerning this question are:

- The test instances used in the article are still openly available, though there are no checksums (or similar) available to ensure that they are identical with the ones used in [67].
- The SIP source code is available. While not mentioned in [67], it seems clear which precise version of the code was used.
- The program was run on a SUN Ultra Enterprise 3000 with 4 UltraSparc processors with 167 MHz, 1 GB RAM using Solaris 7. Incidentally, this machine or a quite similar one is still available at ZIB, because it is planned to be exhibited in a museum. In a few years it will be very unlikely to find such a machine without major effort.
- As it turned out, this machine has no compiler installed. Furthermore, it is neither clear from the description in the article nor the source code which compiler was used. It seems that some version of the SUN SUNWspro C/C++ compiler had been applied. Probably it would be possible to locate an old CD with this software and try to install it. Nevertheless, it is unlikely that we would be able to reproduce the exact binary.
- SIP used CPLEX 5.0 as its LP-solver, i.e., it needs the CPLEX callable library. Finding an old CD will not help, since any available license would have expired for years. In the meantime ILOG, the owner of CPLEX at that time, has been acquired by IBM. IBM is still in the possession of the source code, but is not willing to release the code, even in binary form, due to the legal effort required.

We conclude that while it might be technically possible to reproduce the binary (or something very similar) and run it on a machine similar to the original one, it would require a lot of effort to do so. Furthermore, there is no way to check whether we actually succeeded in producing the precise environment under which the tests where performed in the paper. It seems to be the only possibility to compare the new results with the old log files – which are still available – and those printed in the article. If there is any mismatch, this could be due to a hundred different reasons, like a different compiler switch for some subsystem, a different version of a system library, etc.

Another point is determinism and reproducibility regarding computer hardware. Computer hardware is not free of errors, e.g., Intel is listing several hundred errors for their CPUs in so-called *Specification Updates* on `http://arki.intel.com`. It should be noted though that most of these errors are very rare and need extremely complicated conditions to appear. Computers and in particular RAM are also prone to errors due to cosmic rays. While all this might be an issue on large scale super-computers, we can neglect it on workstation level. Another issue is that modern systems have features like NUMA, Hyperthreading, Turbo-Boost, etc. that require careful attention to ensure reproducible results.

Since we are interested in the algorithmic advances and less in the question whether the computing machinery got faster, it should suffice to compile the code on a modern machine, using a modern compiler and get comparable results apart from the timings.

Table 1: SIP with default settings; taken from Table 5.1 given on page 75 of [67].

Example	B&B	Cuts	Dual Bound	Primal Bound	Time	Gap %
lOteams	10370	0	922	924	3600.0	0.217
air03	8	0	340160	340160	6.7	0.000
air04	1220	0	56137	56137	1532.5	0.000
air05	3588	0	26374	26374	1696.8	0.000
arkiOOl	100776	4	7579808.299	7646059.57	3600.1	0.874
bell3a	25146	0	878430.316	878430.316	45.3	0.000
bell5	337394	1	8966406.491	8966406.491	536.7	0.000
blend2	15055	5	7.598985	7.598985	122.4	0.000
cap6000	4323	2578	-2451418.742	-1236924	3604.0	49.543
dano3mip	1	0	576.2316203	–	3710.3	–
danoint	12655	0	62.94058146	70	3600.3	11.216
dcmulti	2637	0	188182	188182	14.6	0.000
dsbmip	867	0	-305.198175	-305.198175	42.7	0.000
egout	222	0	568.1007	568.1007	0.2	0.000
enigma	8002	524	0	0	24.2	0.000
fast0507	234	0	172.2530211	177	3604.8	2.756
fiber	783	372	405935.18	405935.18	16.9	0.000
fixnet6	1669	0	3983	3983	14.6	0.000
flugpl	7976	25	1201500	1201500	4.4	0.000
gen	11	20	112313.3627	112313.3627	0.3	0.000
gesa2	209525	33	25771445.96	25783761.56	3600.0	0.048
gesa2_o	264243	0	25711931.57	25823063.47	3600.0	0.432
gesa3	5297	0	27991042.65	27991042.65	97.1	0.000
gesa3_o	74472	0	27991042.65	27991042.65	1144.7	0.000
gt2	2215	5	21166	21166	3.2	0.000
harp2	23990	15966	-73944202.17	-70801289	3600.1	4.250
khb05250	2637	0	106940226	106940226	16.3	0.000
l1521av	3209	269	4722	4722	93.8	0.000
lseu	303	164	1120	1120	1.1	0.000
misc03	699	14	3360	3360	4.1	0.000
misc06	308	0	12850.86074	12850.86074	4.2	0.000
misc07	35585	0	2810	2810	378.8	0.000
mitre	1286	3865	115155	115155	1125.8	0.000
mod008	884	371	307	307	9.8	0.000
modO1O	237	3	6548	6548	5.6	0.000
mod011	6108	0	-54558535.01	-54558535.01	2791.4	0.000
modglob	1000000	0	20652263.27	20763655.71	3495.8	0.539
noswot	1000000	179	-43	-41	2270.7	4.651
nw04	1827	0	16862	16862	732.9	0.000
p0033	77	53	3089	3089	0.1	0.000
p0201	507	136	7615	7615	5.0	0.000
p0282	1345	2308	258411	258411	38.3	0.000
p0548	1610	902	8691	8691	25.3	0.000
p2756	23151	6923	3113.257351	3141	3600.2	0.891
pk1	501934	0	11	11	1581.8	0.000
pp08a	1000000	0	5446.190476	8620	2092.7	58.276
pp08aCUTS	624198	0	6970.027419	7650	3600.0	9.756
qiu	17378	0	-132.873137	-132.873137	2326.5	0.000
qnet1	17694	12	16029.69268	16029.69268	1229.9	0.000
qnet1_o	3806	0	16029.69268	16029.69268	158.6	0.000
rentacar	105	0	30356760.98	30356760.98	53.2	0.000
rgn	2505	315	82.19999924	82.19999924	9.6	0.000
rout	200371	316	1048.991823	1079.19	3600.0	2.879
set1ch	841033	0	39920.71098	67819.5	3600.0	69.886
seymour	1947	0	406.4218572	438	3601.8	7.770
stein27	4666	0	18	18	8.0	0.000
stein45	54077	0	30	30	277.7	0.000
vpm1	1000000	0	19.5	20	1892.6	2.564
vpm2	555712	0	13.75	13.75	1368.7	0.000
Total (59)	8017878	35366			77823.6	226.547

Tobias Achterberg, now at IBM, kindly agreed to do this. He compiled the original SIP code together with CPLEX 5.0.1 using gcc 4.0.1 on a modern Intel Xenon X5260 powered Linux computer. As it turned out, we achieved identical results regarding the number of branch-and-bound nodes for all but three instances. Trying to solve the instance *mitre* the program crashed due to a numerical error from CPLEX. The runs for the instances *lseu* and *p0033* differ. The most likely explanation is that the LP-solver returned a different optimal basis. Indeed, Intel x86 architecture CPUs do double precision (64 bit) floating point calculations internally with 80 bit precision. This can lead to different results compared to SPARC CPUs which use 64 bits throughout. Even though both conform with the IEEE 754 standard. There are further reasons which might lead to different results. These differences can be large enough to let the simplex algorithm terminate at a different vertex of the optimal face. As a consequence, SIP then generated different cuts, which in turn led to a different number of nodes. The returned objective function value was the same in all solved cases. Since the original solutions are not available anymore, we cannot check whether the solutions returned are also the same.

Different to the original runs, the time limit given to SIP was in wall-clock time. Due to changes in the operating system API, CPU time measurement seemed not to work correctly (anymore). We also conducted limited experiments concerning the CPLEX 5.0.1 MIP-solver. Here we experienced that any change of the compiler options would also change the number of branch-and-bound nodes. And again, there were problems regarding the time measurement.

Moreover, we tried to link the newly compiled SIP code to more recent versions of CPLEX. This succeeded only half-way because of changes in the CPLEX API and behaviors that could not easily be translated back to match the functionality expected by the SIP code. We found the Version 10.1 (current is 12.5) to be the best suitable one (later versions had some used functions removed). But also in 10.1 the behavior of some functions has changed. Between Versions 7 and 8, CPLEX changed the return codes given for the optimization functions. We tried to fix this in the old code, but there seems to be at least one issue for which we do not know a solution.

SIP could also run using multiple threads in a non-deterministic parallel mode. Again, we were not able to reliably reproduce the old results. The major reason is that especially in the area of multi-threaded computations the underlying computational environment changed significantly. Furthermore, in the case of non-deterministic codes where decisions are taken depending on timings, the reproduction of results on different hardware is, by construction, hardly possible at all.

One main observation is that the use of closed-source (commercial) libraries is a major roadblock for reproducibility. If a company goes out of business, the source code or binary might be lost forever. In any case it is usually difficult to get the old code. Moreover, licensing problems are critical.

We observe that it is important to save all relevant information of the experiments. The source code used for experiments should be available including all needed libraries, all necessary data and log files. It seems to be useful to also save the binaries. Table 3 below, for example, was produced using a binary made in 2003. The current trend to use dynamically linked libraries is disastrous for reproducibility. While a statically linked program might run still decades later, trying to run a dynamically linked program later on is often hopeless and even might not produce the same result due to possibly changed system libraries.

Finally, one might also question whether the originally published results were correct to begin with. In the current academic system, the review process conducted by journals usually does not include looking at the code or the experimental data. Questions similar to the ones above led to the founding of the journal *Mathematical Programming Computation* in 2008 published by the Mathematical Optimization Society, see [30]. A notable feature of the journal is that authors are encouraged to submit the code and all the data necessary to reproduce the results of the article. A special group of *Technical Editors* then recompiles the code and reproduces the

results. Furthermore, the code is reviewed to assess whether it resembles the description in the paper. Initially, questions were raised whether this will always be possible. From the experience made in the last years the answer is yes. It was even possible to rerun and review a code that was designed for a BlueGene supercomputer. And in this case, as in many others, the review process helped to substantially improve the quality and usability of the code submitted. Also the review process forces the authors to explicitly state the details of all third party codes needed.

The solving of publicly available (benchmark) instances can lead to a certain competitiveness. As announced on June 28th, 2011 on the website `http://ilk.uvt.nl/icga` of the International Computer Games Association (ICGA), the four times world chess championship winner program Rybka was banned and disqualified because an investigation by [59] found the programmer guilty of plagiarizing two other programs. This also can only happen if the code is not available for reviewing.

Assume you have the source code for the program in question. Furthermore, there is a script that runs the test and so documents the settings used and the complete log files for the runs used for the publication. They have to be detailed enough to decide whether another run was similar/identical. The best situation is if the environment that has been used for the publication is still available, i.e., the computer/operating system and the binary. Given that a static binary was used, it should be possible to reproduce the experiments easily. Therefore, a static linked binary of the program should be kept. Once the computer is not available anymore, but at least the architecture/operating system is still available, chances are relatively good that the binary will run on a more modern environment. If the architecture/operating system is no longer available because it is too old, there is a high probability that emulators are available.

If the binary is not available anymore, it gets more complicated. If the source code to all used third party libraries is available, it is possible to completely compile the program again, although maybe not by the same compiler. If the library source code is not available, there might be (new) versions of the library for the new target architecture. But APIs change over time and it may be necessary to adjust the source code of the program. Here it is important that the log files are available and detailed enough to allow a check whether the program produces similar results.

To summarize here is what you want to keep:

1. source code and makefiles,
2. run scripts and log files,
3. source code for all used non-system libraries, and
4. a static linked binary of the program or as close as you can get to it.

4 How to Measure Performance of Integer Programming Solvers

An obvious question that arises when trying to find better ways to solve integer programs is how to measure progress. The common solution is to have a publicly available and accessible library of test instances to compare algorithms and implementations against. This was started for LPs with the NETLIB by Gay [41] and then for integer programming with the MIPLIB by Bixby, Boyd, and Indovina [24]. Due to the permanent advances in algorithms and the increase in computer speed, instances that are difficult in the beginning become easy to be solved over time. Consequently, the test instance libraries have to evolve, dropping "too easy" instances and adding harder ones. For the MIPLIB this has been done five times by now, as shown in Table 2.

The first two authors have been involved in these updates: first in 2003, see [9], and in the current version in 2010, see [56]. For the first time a consensus of all major solver developers in industry and academia on the selection of the instances

Table 2: List of MIPLIB versions

Version	Name	Date	Who	Reference
1	MIPLIB	1991	Bixby, Boyd, Indovina	[24]
2	MIPLIB 2.0	1992	Bixby, Boyd, Indovina	[24]
3	MIPLIB 3.0	1996	Bixby, Ceria, McZeal, Savelsbergh	[25]
4	MIPLIB 2003	2003	Achterberg, Koch, Martin	[9]
5	MIPLIB 2010	2010	Many	[56]

could be achieved. It was agreed to substantially extend the test set. The effort and investigations to produce the 5th incarnation of the MIPLIB were substantial. The reward was a comprehensive instance library that is accepted by researchers worldwide. Furthermore, it was possible to give a true snapshot of the state-of-the art in MIP-solvers.

We report on some of the issues related to benchmarking, based on the experiences from the current MIPLIB. A key issue is that people tend to like condensed information. For measuring performances, be it the speed of a computer or the publication performance of a researcher, one would like to have a single number to describe it, since single numbers are easy to compare. Sometimes this is impossible. Regarding the computation of citation scores for researchers, we refer to [12, 13] for a discussion why this is not catching the truth. For integer programs it is slightly easier, both to compute a single number and to show that this number has to be interpreted very carefully.

It is the central problem to decide which measures to compare and on which instances. One way is to select a large number of instances by some high-level argument, for instance, they should be real-world instances or from a mix of applications. A more detailed description of this can be found in [56]. The selection of the instances seemed to work quite well, since incidentally the geometric mean performance of the three top solvers on the benchmark set were nearly equal, while the maximum difference on a particular instance was a factor greater than 1,000. This answers the question what such a mean number tells you in case that you have to solve a particular class of instances: nothing. It reveals that by careful (or sloppy) selection of the instances it is quite easy to come up with a test set where one solver is 1,000 times faster than the other. Note the above characteristic makes it easy for the marketing departments to produce funny comparison numbers. Basically, all three top commercial solver vendors claim to be faster than the competition, see [36], [46], [50].

It is even more complicated to compare solvers on instances that one solver can solve in a given time and the other solver cannot. In many cases it is not possible to wait until both solvers have solved the instance, e.g., if we have a one-hour time limit and solver B would need 10,000 times as long, we would have to wait more than a year. How should these instances be counted? One possibility is to only take those instances that both solvers can solve. This is quite biased towards the weaker solver: Imagine a solver that only checks whether the zero solution is feasible on instances that have no objective function. It will be at least as fast as any other solver and there is no way to beat it. Another option is to use a time limit and then report this limit as the solution time. Due to the large differences in solution time the ratio between two solvers then becomes just a lower bound in favor of the weaker solver. By increasing the time limit, the ratio will also continue to increase, as long as one solver solves one instance more. One could also simply not use averaging times and just count the number of instances that can be solved by a particular solver in a given amount of time. The obvious problem is that again the result depends heavily on the particular time limit and very much on the selected test set.

Figure 1 shows the distribution of solution times for the 89 instances of the MIPLIB 2010 *benchmark* test set. For each solver the times have been sorted in ascending

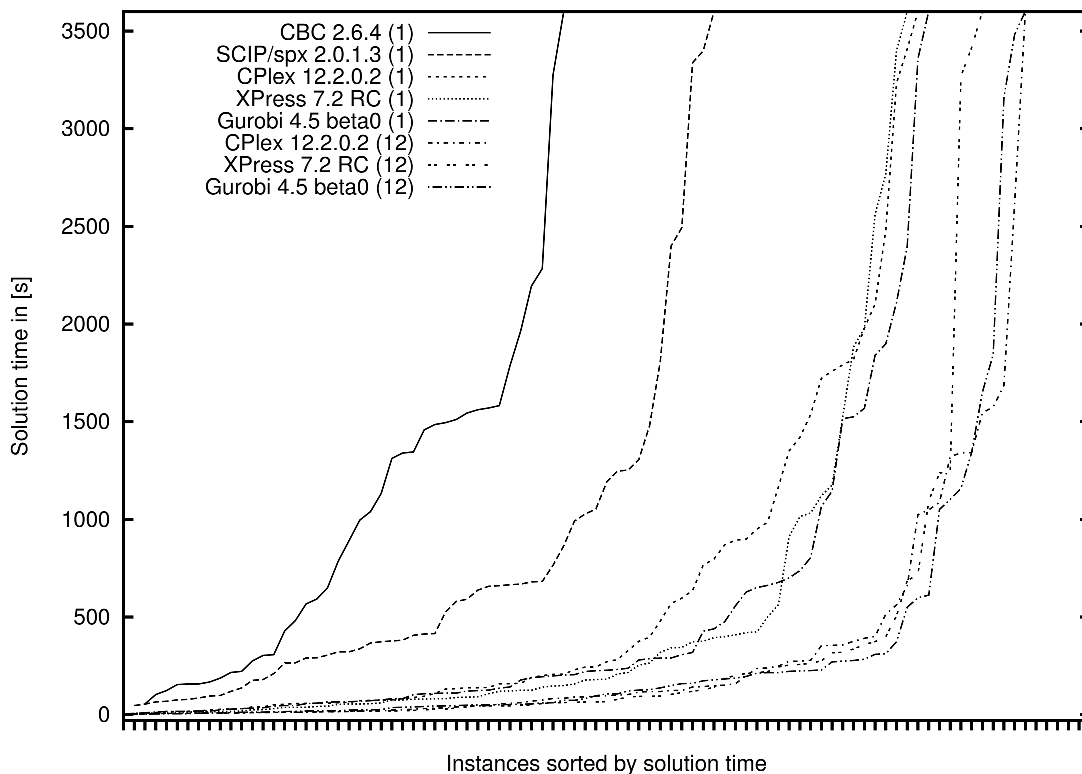

Figure 1: MIPLIB 2010 benchmark set, distribution of solving time for 1 and 12 threads

order, therefore a specific position of the x-axis does not necessarily correspond to a particular instance. It can be observed that the instances fall into basically three categories: easy instances that can be solved within a few minutes, those instances that cannot be solved at all, and those in between. The criteria for the benchmark set were, among others, that at least two solvers where able to solve a particular instance within two hours and that the instance was not too easy. Therefore, compared to a larger more random set of instances, the amount of easy and unsolved instances is reduced. But we still can see that the *in-between* category is small. By speeding up the solvers using 12 threads this phenomenon becomes more pronounced, i.e., the in-between group actually shrinks. This is a phenomenon that can be observed in general. Making the computer faster will just solve those problems faster that could be solved before, but the number of instances that could not be solved at all will stay mostly the same.

5 Measuring Advances in Computational Integer Programming

As described in the introduction, the work on the MIP-solvers SIP and SCIP has now spanned more than 16 years from 1996 to 2012. In the following we give an impression on the advances of the field during this time.

First, Figure 2 shows the time distribution for the test set used in [67] in 1998. The results on the UltraSparc CPU were taken from [67] (see Table 1). The results on the 3333 MHz Intel Xenon X5260 CPU were obtained by recompiling the original SIP code and linking to CPLEX 5.0.1 using gcc 4.0.1. The picture shows the comparison between the old and the new codes. In 1998, SIP could solve 41 out of 59 instances. Instance *mitre* that could be solved on the UltraSparc now stopped with a numerical

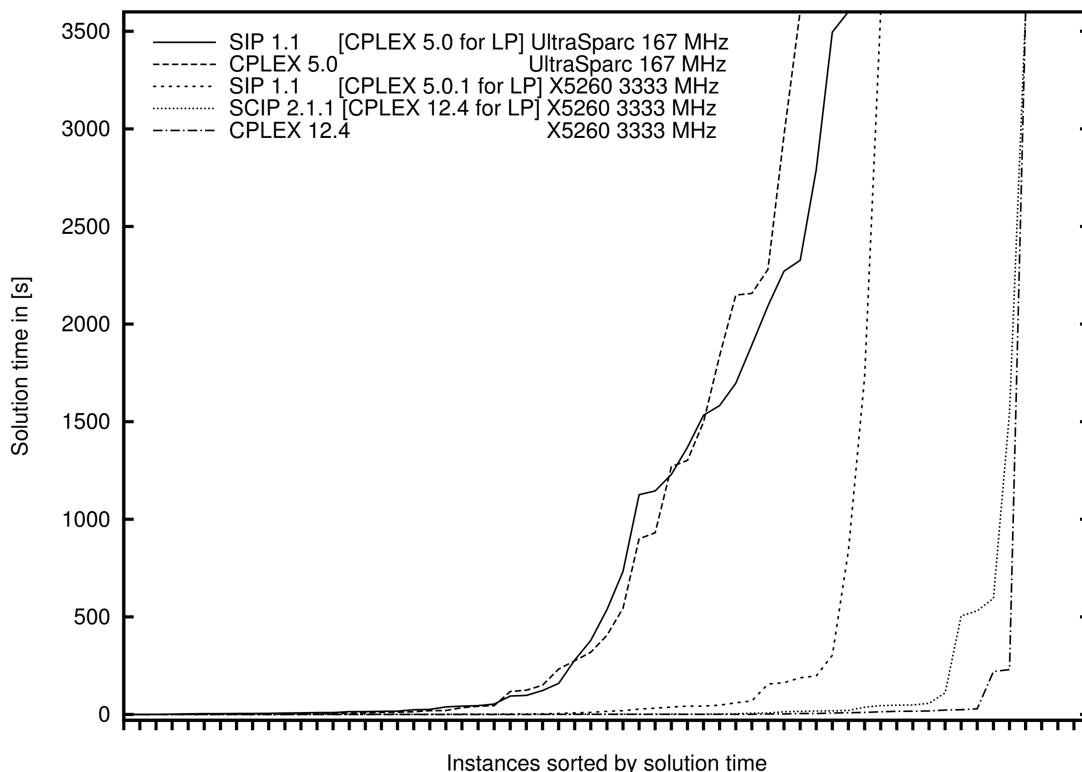

Figure 2: 1998 benchmark set, distribution of solving time for different computers and codes

error and was counted as a timeout with 3,600 seconds. For the X5260 we did not impose the 1,000,000 node limit used in the paper (this limit was probably set in order to control memory consumption). This affects the results for instances *10teams*, *2756*, *rout*, *pp08aCUTS*, *vmp1*, *gesa2_o* – compare Table 1.

The geometric mean of the running time using SIP on the UltraSparc was 40.2 seconds. This is now down to about 1.2 seconds, which gives us a speed-up factor of roughly 30 between the 167 MHz and the 3333 MHz computer. The clock speed ratio is about 20, but one should keep in mind that these are two totally different architectures: UltraSparc is a pure RISC architecture with in-order execution, while the Xenon is a much more complex out-of-order execution CISC CPU. Moreover, note that since reading times are included in the time measurement and due to the slight variations that are always present, fractions of seconds are not measured accurately enough to draw conclusions from it.

It should be noted that in case of instance *dano3mip*, the optimum is still unknown, and in case of instance *seymour*, while it has been solved, none of the commercial solvers is able to solve it within one hour even on 12 threads.

What can be concluded from the comparison of the curves of SIP 1.1 on the UltraSparc and the X5260 is that basically the same number of instances can be solved, but it now only takes 1/30th of the time. This speed-up results in a nearly rectangular shaped curve for the faster computer for the MIPLIB 2010 case. SIP 1.1 either solves an instance in two minutes, or not at all. The same is essentially true for the new codes: They are able to solve about ten more instances, but do this very fast, while a few instances remain which have a solving time close to the time limit.

Figure 3 shows the performance over time relative to the current version of SCIP. All results were computed on an Intel Xeon X5672 CPU at 3200 MHz. Note that due to the time limit of one hour, the maximum slow-down factor compared to the latest

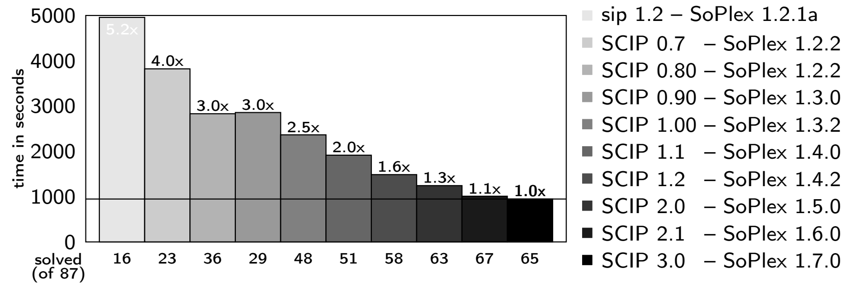

Figure 3: Performance of different SIP/SCIP versions on the same computer

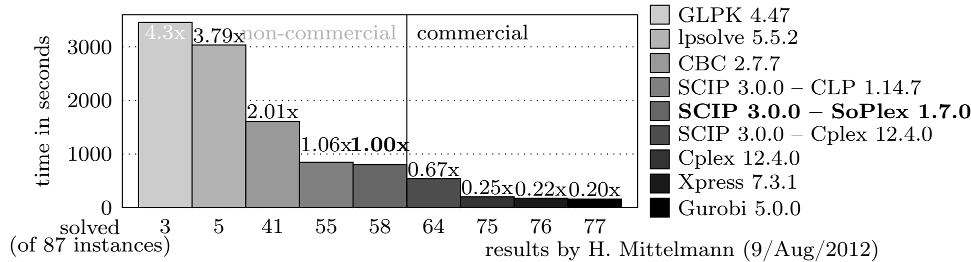

Figure 4: Comparison of MIP-solver performance relative to SCIP/SoPlex

version is five. The time for SIP 1.2 was computed using an old binary. It was able to solve only 13 out of 87 instances within the time limit and consequently was rated 4.9 times slower than the current version.

Note that it is unlikely that SIP will solve much more instances when increasing the time limit. This can be seen by Figure 2, where SIP on a roughtly 30 times faster machine can only solve 5 instances more.

As mentioned before, the above factor is a lower bound on the real slow-down, which is likely to be arbitrarily large, because there will be instances which can be solved with SCIP 2.1 and which might take practically forever using SIP 1.2. In this sense, the number of solved instances is much more important than the slow-down factor.

Figure 4 depicts a comparison between contemporary solvers conducted by Hans Mittelmann. (See `http://plato.asu.edu/ftp/milpc.html` for the latest results.) There are several interesting facts to note:

- Because of the one-hour time limit compared to the two-hours in Figure 3, SCIP with SoPlex solves 10 instances less. As Figure 1 shows, the bend in the curve is more pronounced in the commercial solvers, i.e., the number of *in-between* instances is bigger for SCIP. Therefore SCIP would benefit more from an increased time limit as compared to the commercial solvers.
- The speed-up from SCIP using CPLEX as LP-solver instead of SoPlex is just about 30%. Given that CPLEX on pure LP benchmarks is much faster than SoPlex and that solving the LPs takes a considerable amount of the total running time of a MIP-solver, this is a surprisingly small difference. The reasons for this are manifold, and we are currently investigating this phenomenon which was also observed by others, e.g., Bixby [23].
- If two solvers are compared that solve a substantially different number of instances to optimality, the speed-factor is underestimated in favor of the code that solves less instances. For LPSOLVE and GLPK the factor given in the picture is meaningless as they solve only 3 or 5 instances, respectively.

We can now make an estimation on the progress in MIP-solving by SIP/SCIP since 1998. As computed above, the hardware speed-up factor is about 30. The latest

Table 3: Number of instances in each class of MIPLIB 2003/2010 over time

	Date	Easy	Hard	Unsolved
		MIPLIB 2003		
Start	2003	22	3	35
	2004	27	12	21
	2005	28	13	19
	2006	28	13	19
	2007	31	22	7
	2008	34	20	6
	2009	35	19	6
	2010	35	19	6
	2011	41	15	4
	2012	44	12	4
		MIPLIB 2010		
	05.2011	185	42	134
	07.2011	196	33	132
	08.2011	202	29	130
	01.2012	202	30	129
	02.2012	203	40	118
	03.2012	204	41	116
	04.2012	204	43	114
	05.2012	206	42	113
	06.2012	208	45	108
	07.2012	208	47	106
	08.2012	208	50	103

version of SCIP/SoPlex is at least five times faster than SIP 1.2/SoPlex. We assume SIP 1.2 was at least as fast as SIP 1.1. Since SIP 1.1 was run with CPLEX as LP-solver, we now must also compare relatively to SCIP/CPLEX, which is about 3.5 times slower than Gurobi. This gives us an estimated lower bound on the speed-up from SIP 1.1/CPLEX on an UltraSparc to Gurobi on a modern PC of about $30 \times 5 \times 3.5 \approx 525$. Multiplying this number with the average speed-up from multithreading (approximately a factor of 3) gives an average speed-up of 1.63 times per year over 15 years or a practical doubling of the MIP-solver performance every 18 months. Remember that this is only a lower bound and that the speed-up is distributed extremely unevenly on the instances.

A more general picture can be drawn from the MIPLIB. In MIPLIB we classify an instance as *easy*, if a commercial solver on a high-end PC can solve the instance within an hour. It is classified as *hard* if it can be solved by some solver, but not by every solver, and as *unsolved* otherwise. People often report or publish if they are able to solve an instance for the first time. Figure 3 lists the number of *easy*, *hard*, and *unsolved* instances in MIPLIB 2003 and 2010 over time. Note that this includes also the progress through faster computers.

One has to be cautious regarding the interpretation of these numbers, because part of the progress results from the library instances used to tune the solver algorithms. Therefore the progress is probably overstated.

6 Developing Academic Integer Programming Codes

Last but not least, we want to discuss issues related to the development of academic integer programming codes. We think that such a discussion is important, since we have the impression that currently researchers may not be aware of several of these issues or might even disagree over the consequences. Our main question is

"Does it still make sense to develop integer programming codes in academia?"

Before addressing this question, we point to several organizational obstacles that have to be dealt with. We dispense with licensing issues here, because this is a longer topic of its own, and rather focus on code development and publications.

Concerning *code development* we mentioned above that solver development has more and more become a team effort. As an example, a new release of SCIP requires a tremendous amount of work that we briefly mention in the following, as it might give an example for other projects. SCIP alone contains more than 400,000 lines of code and surely contains (possibly many) errors. Thus, a big part of the work concerns debugging. Bugs are either reported by the users through a web interface or are found by significant amount of tests. These bugs might become visible, because the computed result differs from the known optimum or from a previously computed value. Bugs can also be found because one of the checkpoints (asserts) or unit-tests in SCIP is triggered. Some bugs may be due to numerical issues and thus require longer time to debug. The infrastructure of SCIP helps debugging, but in total the preparation of a release is spread over the time span of three months and the work of about four full-time developers.

There is one further aspect that seems to be relevant in this context. It is indispensable to further advance the field teamwork, and this has repercussions on how research is conducted. Like in physics and other areas, research in computational integer programming is more and more becoming team-work. Moreover, students are able to learn how a solver works and are possibly able to join commercial MIP-solver teams. (This seems to be the employee recruitment strategy in the field.) We think that all this is only possible through academic research.

With respect to *publications*, we all know that it is still hard to publish papers on computational optimization in first class journals. Basically, there is no credit for the code development work that is involved. It is a fact that still almost all codes used for publications are not publicly available. This is, actually, the biggest obstacle for reproducibility. The introduction of the journal *Mathematical Programming Computation* is a little step in the right direction, but a different way of handling papers by editors of journals is needed. It is clear that the goal should be that every code used for computations in a paper should be available for possible reproduction or even improvement. Moreover, the effort needed for code development should be taken into account.

We now come back to our initial question: Is there (still) room for academic solvers in times when the commercial solvers seem to be computationally ahead?

There are several commercial codes available that either serve their purpose as a stand-alone program or can be used as a B&C framework for individual applications through callbacks. Clearly, for many applications it suffices to just use a stand-alone MIP-solver, since this already "finishes the job" in many cases.

It is also a fact that many publications in the field use these commercial solvers as a B&C framework through callbacks. To quantify this claim, we conducted the following literature investigation. We checked all articles in the journals Mathematical Programming A and B (MPA and MPB, resp.) and Mathematical Computation (MPC) in the years from 2003 (for MPA/MPB) and 2009 (for MPC) to 2012 that perform computations and use integer programming techniques. Table 4 shows the results.

Here, we have not counted articles on mixed integer nonlinear programming. Several articles simply apply a MIP-solver, but have been counted if there is additional

Table 4: Statistics on articles in Mathematical Programming A and B (MPA/MPB) and Mathematical Programming Computation (MPC) that use MIP-solvers/frameworks (2003–2012)

Journal	# articles
MPA	61
MPB	33
MPC	12
Total	106

Framework	# articles
CPLEX	51
XPRESS	9
COIN-OR	14
SCIP	6
ABACUS	4
MINTO	2
CONCORDE	2
unkown	2
own	27

coding involved. Furthermore, note that double counts in the number of frameworks are possible, e.g., article [56] on MIPLIB 2010, which compares several MIP-solvers. Articles that use own implementations to handle subproblems, bounds, primal heuristics, etc. are listed in "own". These articles often also use MIP-solvers, e.g., for solving MIP-subproblems; such articles are counted twice.

It should be clear that these numbers have to be treated with care. However, it seems to be clear from these numbers that CPLEX is clearly the framework that has been used the most. Moreover, most of the articles use closed-code frameworks; for instance, we have: CPLEX + XPRESS = 60 vs. COIN-OR + SCIP + ABACUS = 24 (MINTO/CONCORDE should probably be counted as a closed code, and ABACUS was a closed code at the time of the publication of some of the articles).

Of course, this dominance of commercial codes has reasons. Some of the arguments for using commercial MIP-solvers are the following:

1. Commercial solvers are highly tuned and, thus, also promise the best performance for individual applications.
2. These solvers have to be used as a black-box. Thus, there is no possibility to tune the implementation of the framework. Consequently, researchers can concentrate on their own code, which reduces the amount of work needed.
3. Many researchers in the field have developed a code over the years, which is often based on a particular solver. Thus, changing the framework would require significant reimplementation effort.
4. One main argument against using such solvers was that licensing was problematic. However, most commercial solvers offer academic licenses today, so this is currently not an issue.

Ironically, all of these arguments can be turned around and used against using commercial MIP-solvers:

1. The promise of best performance might be wrong, and it is (almost) impossible to check whether small changes to the system or implementation might lead to a still better performance. Moreover, using a black-box solver does not help to understand why a code is fast.
2. We do not know (completely) what happens in a black-box solver. Thus, it is scientifically questionable to have significant parts of the code in which we are not able to determine exactly what happens. More severely, the resulting code might produce wrong results, since some effects inside the black-box could not be taken into account.

3. The code basis is not really a scientific issue. Possibly, researchers would be willing to switch their code bias, if this would promise a significantly improved performance.
4. Licensing might change (see the comments in Section 3).

An additional argument for academic MIP-solvers is that some functionality of a black-box solver might not be available through its API. Moreover, the usage of API functions might incur unintended effects – sometimes even if no action should actually be taken; examples are refactorizations of the basis in the LP-solver or even removing old basis information or the automatic deactivation of certain algorithmic components when using callbacks.

All these arguments support the development of academic MIP-solvers or, more generally, B&C frameworks. However, it is unclear whether academic implementations for MIP-solving will be able to keep up with the performance of commercial solvers. Currently, the difference seems to be still acceptable as we have seen in this paper. Of course, this might change in the future, but predictions are always difficult. It is our belief that new ideas in the field have to be developed both in academia and industry – this worked very well in the past. Otherwise, the performance of MIP-solvers, academic and commercial, will stall.

7 Acknowledgments

We thank all developers of SCIP/SoPlex (for a complete list, see [71]); in particular, we thank Timo Berthold and Stefan Heinz for preparing the Figures 3 and 4. Thanks to Madeline Lips for her help with the literature investigation in Section 6. We also thank a referee for helpful and insightful comments that improved this paper. Last but not least, we are most thankful to Martin Grötschel. Investing in a long-term project as the development of a general MIP solver is only possible if you get the time and the trust to do it. Both has been given to all three of us by Martin Grötschel in an excellent environment at ZIB.

References

[1] Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Opt. **4**(1), 4–20 (2007)

[2] Achterberg, T.: Constraint integer programming. Ph.D. thesis, Technische Universität Berlin (2007)

[3] Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation **1**(1), 1–41 (2009)

[4] Achterberg, T., Berthold, T.: Hybrid branching. In: W.J. van Hoeve, J.N. Hooker (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, *Lecture Notes in Computer Science*, vol. 5547, pp. 309–311. Springer, Berlin (2009)

[5] Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: A new approach to integrate CP and MIP. In: L. Perron, M.A. Trick (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, CPAIOR 2008, *Lecture Notes in Computer Science*, vol. 5015, pp. 6–20. Springer (2008)

[6] Achterberg, T., Brinkmann, R., Wedler, M.: Property checking with constraint integer programming. Tech. Rep. 07-37, ZIB, Takustr.7, 14195 Berlin (2007)

[7] Achterberg, T., Heinz, S., Koch, T.: Counting solutions of integer programs using unrestricted subtree detection. In: L. Perron, M.A. Trick (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, CPAIOR 2008, *Lecture Notes in Computer Science*, vol. 5015, pp. 278–282. Springer (2008)

[8] Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Operations Research Letters **33**, 42–54 (2005)

[9] Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Operations Research Letters **34**(4), 361–372 (2006)

[10] Achterberg, T., Raack, C.: The MCF-separator – detecting and exploiting multi-commodity flows in MIPs. Mathematical Programming Computation **2**(2), 125–165 (2010)

[11] Achterberg, T., Wunderling, R.: Mixed integer programming: Analyzing 12 years of progress (2013). To appear

[12] Adler, R., Ewing, J., Taylor, P.: Citation statistics. Tech. rep., IMU, ICIAM, IMS (2008). Http://www.mathunion.org/fileadmin/IMU/Report/CitationStatistics.pdf

[13] Adler, R., Ewing, J., Taylor, P.: Citation statistics. Notices of the AMS **55**(8) (2008)

[14] Applegate, D., Bixby, R.E., Chvátal, V., Cook, W.: Finding cuts in the TSP. Tech. Rep. 95-05, DIMACS (1995)

[15] Applegate, D.L., Bixby, R.E., Chvatal, V., Cook, W.J.: The Traveling Salesman Problem: A Computational Study. Princeton University Press (2006)

[16] Armbruster, M., Fügenschuh, M., Helmberg, C., Martin, A.: Lp and sdp branch-and-cut algorithms for the minimum graph bisection problem: A computational comparison. Mathematical Programming Computation **4**(3) (2012)

[17] Atamtürk, A., Savelsbergh, M.: Integer-programming software systems. Annals of Operations Research **140**, 67–124 (2005)

[18] Balas, E., Ceria, S., Cornuéjols, G., Natraj, N.: Gomory cuts revisited. Operations Research Letters **19**, 1–9 (1996)

[19] Benichou, M., Gauthier, J.M., Girodet, P., Hentges, G., Ribiere, G., Vincent, O.: Experiments in mixed-integer programming. Math. Prog. **1**, 76–94 (1971)

[20] Berthold, T., Heinz, S., Pfetsch, M.E.: Nonlinear pseudo-boolean optimization: relaxation or propagation? In: O. Kullmann (ed.) Theory and Applications of Satisfiability Testing – SAT 2009, *Lecture Notes in Computer Science*, vol. 5584, pp. 441–446. Springer (2009)

[21] Berthold, T., Heinz, S., Pfetsch, M.E., Vigerske, S.: Large neighborhood search beyond mip. In: L.D. Gaspero, A. Schaerf, T. Stützle (eds.) Proceedings of the 9th Metaheuristics International Conference (MIC 2011), pp. 51–60 (2011)

[22] Bixby, R.E.: Solving real-world linear programs: A decade and more of progress. Operations Research **50**(1), 3–15 (2002)

[23] Bixby, R.E.: Personal communication (2012)

[24] Bixby, R.E., Boyd, E.A., Indovina, R.R.: MIPLIB: A test set of mixed integer programming problems. SIAM News **25**, 16 (1992)

[25] Bixby, R.E., Ceria, S., McZeal, C., Savelsbergh, M.: An updated mixed integer programming library: MIPLIB 3.0. Optima **58**, 12–15 (1998)

[26] Bixby, R.E., Martin, A.: Parallelizing the dual simplex method. INFORMS Journal on Computing **12**, 45–56 (2000)

[27] Bley, A., Gleixner, A., Koch, T., Vigerske, S.: Comparing MIQCP solvers to a specialised algorithm for mine production scheduling. In: H.G. Bock, H.X. Phu, R. Rannacher, J.P. Schlöder (eds.) Modeling, Simulation and Optimization of Complex Processes: Proceedings of the Fourth International Conference on High Performance Scientific Computing, March 2–6, 2009, Hanoi, Vietnam, pp. 25–40. Springer (2012)

[28] Borndörfer, R.: Aspects of set packing, partitioning, and covering. Ph.D. thesis, Technische Universität Berlin (1998)

[29] Borndörfer, R., Ferreira, C.E., Martin, A.: Decomposing matrices into blocks. SIAM Journal on Optimization **9**, 236–269 (1998)

[30] Cook, W., Koch, T.: Mathematical programming computation: A new MPS journal. Optima **78**, 1,7–8,11 (2008)

[31] Crowder, H., Johnson, E.L., Padberg, M.W.: Solving large-scale zero-one linear programming problems. Operations Research **31**(5), 803–834 (1983)

[32] Dantzig, G.: Linear Programming and Extensions. Princeton University Press (1963)

[33] Dantzig, G., Fulkerson, R., Johnson, S.: Solution of a large-scale traveling-salesman problem. Operations Research **2**, 393–410 (1954)

[34] Eckstein, J., Boduroğlu, v.v., Polymenakos, L.C., Goldfarb, D.: Data-parallel implementations of dense simplex methods on the connection machine CM-2. ORSA Journal on Computing **7**(4), 402–416 (1995)

[35] Ferreira, C.E., Martin, A., Weismantel, R.: Solving multiple knapsack problems by cutting planes. SIAM J. on Optimization **6**(3), 858–877 (1996)

[36] FICO: `http://www.fico.com/en/FIResourcesLibrary/Xpress_7.2_Benchmarking_2773FS.pdf` (2012). Visted May 2012

[37] Forrest, J., Tomlin, J.: Updated triangular factors of the basis of maintain sparsity in the product form simplex method. Mathematical Programming **2**, 263–278 (1972)

[38] Forrest, J.J., Goldfarb, D.: Steepest-edge simplex algorithms for linear programming. Mathematical Programming **57**, 341–374 (1992)

[39] Fourer, R.: Linear programming – software survey. OR/MS Today **38**(3) (2011)

[40] Gamrath, G., Lübbecke, M.: Experiments with a generic Dantzig-Wolfe decomposition for integer programs. In: P. Festa (ed.) Experimental Algorithms, *Lecture Notes in Computer Science*, vol. 6049, pp. 239–252. Springer (2010)

[41] Gay, M.: Electronic mail distribution of linear programming test problems. Mathematical Programming Society COAL Bulletin no. 13 pp. 10–12 (1985). Data available at `http://www.netlib.org/netlib/lp`

[42] Greenberg, H.J.: Computational testing: Why, how and how much. ORSA Journal on Computing **2**(1), 94–96 (1990)

[43] Grötschel, M., Jünger, M., Reinelt, G.: A Cutting Plane Algorithm for the Linear Ordering Problem. Operations Research **32**(6), 1195–1220 (1984)

[44] Grötschel, M., Martin, A., Weismantel, R.: Packing Steiner trees: a cutting plane algorithm and computational results. Mathematical Programming, Series A **72**(2), 125–145 (1996)

[45] Grötschel, M., Monma, C.L., Stoer, M.: Polyhedral and Computational Investigations for Designing Communication Networks with High Survivability Requirements. Operations Research **43**(6), 1012–1024 (1995)

[46] Gurobi Inc.: `http://www.gurobi.com/products/gurobi-optimizer/prior-versions` (2012). Visted May 2012

[47] Harris, P.M.J.: Pivot selection methods of the DEVEX LP code. Mathematical Programming **5**, 1–28 (1973)

[48] Heinz, S., Sachenbacher, M.: Using model counting to find optimal distinguishing tests. In: W.J. van Hoeve, J.N. Hooker (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems : 6th International Conference, CPAIOR 2009, no. 5547 in Lecture Notes in Computer Science, pp. 117–131 (2009)

[49] Hooker, J.N.: Needed: An empirical science of algorithms. Operations Research **42**, 201–212 (1994)

[50] IBM – Cplex: `http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/cplex-optimizer/cplex-performance/` (2012). Visted May 2012

[51] ILOG CPLEX: Reference Manual (1997). `http://www.cplex.com`

[52] Januschowski, T., Pfetsch, M.E.: Branch-cut-and-propagate for the maximum k-colorable subgraph problem with symmetry. In: T. Achterberg, J.C. Beck (eds.) Proc. 8th International Conference, CPAIOR 2011, Berlin, *Lecture Notes in Computer Science*, vol. 6697, pp. 99–116. Springer (2011)

[53] Kaibel, V., Peinhardt, M., Pfetsch, M.E.: Orbitopal fixing. Discrete Optimization **8**(4), 595–610 (2011)

[54] Koch, T.: Zimpl user guide. Tech. Rep. ZIB-Report 01-20, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr. 7, Berlin (2001)

[55] Koch, T.: Rapid mathematical programming. Ph.D. thesis, Technische Universität Berlin (2004)

[56] Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R.E., Danna, E., Gamrath, G., Gleixner, A.M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D.E., Wolter, K.: MIPLIB 2010. Mathematical Programming Computation **3**, 103–163 (2011)

[57] Koch, T., Martin, A.: Solving Steiner tree problems in graphs to optimality. Networks **32**, 207–232 (1998)

[58] Kostina, E.: The long step rule in the bounded-variable dual simplex method: Numerical experiments. Mathematical Methods of Operations Research **55**, 413–429 (2002)

[59] Lefler, M., Hyatt, R., Williamson, H., ICGA panel members: Rybka investigation and summary of findings for the ICGA. Tech. rep., International Computer Games Assosiation (2011). http://ilk.uvt.nl/icga/investigation/Rybka_disqualified_and_banned_by_ICGA.rar

[60] Lembke, C.E.: The dual method of solving the linear programming problem. Naval Research Logistics Quarterly **1**, 36–47 (1954)

[61] Linderoth, J.T., Lodi, A.: MILP software. In: J. Cochran (ed.) Wiley Encyclopedia of Operations Research and Management Science, vol. 5, pp. 3239–3248. Wiley (2011)

[62] Linderoth, J.T., Ralphs, T.K.: Noncommercial software for mixed-integer linear programming. In: J. Karlof (ed.) Integer Programming: Theory and Practice, Operations Research Series, pp. 253–303. CRC Press (2005)

[63] Linderoth, J.T., Savelsbergh, M.W.P.: A computational study of search strategies for mixed integer programming. Informs Journal on Computing **11**, 173–187 (1999)

[64] Lodi, A.: MIP computation. In: M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, L. Wolsey (eds.) 50 Years of Integer Programming 1958-2008, pp. 619–645. Springer (2009)

[65] Luce, R., Tebbens, J.D., Liesen, J., Nabben, R., Grötschel, M., Koch, T., Schenk, O.: On the factorization of simplex basis matrices. Tech. Rep. 09-24, Zuse Institute Berlin, Takustr. 7, Berlin (2009)

[66] Mars, S., Schewe, L.: SDP-package for SCIP. Tech. rep., TU Darmstadt (2012)

[67] Martin, A.: Integer programs with block structure. Habilitation thesis, Technische Universität Berlin (1998)

[68] McGeoch, C.C.: A Guide to Experimental Algorithmics. Cambridge University Press (2012)

[69] Mittelmann, H.: Decision tree for optimization software: Benchmarks for optimization software (2003). http://plato.asu.edu/bench.html

[70] Padberg, M., Rinaldi, G.: A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Rev. **33**, 60–100 (1991)

[71] SCIP: Solving constraint integer programs. http://scip.zib.de

[72] Suhl, L.M., Suhl, U.H.: Computing sparse lu factorizations for large-scale linear programming bases. ORSA Journal on Computing **2**(4), 325–335 (1990)

[73] Suhl, L.M., Suhl, U.H.: A fast lu update for linear programming. Annals of Operations Research **43**(1–4), 33–47 (1993)

[74] Vigerske, S.: Decomposition of multistage stochastic programs and a constraint integer programming approach to mixed-integer nonlinear programming. Ph.D. thesis, Humboldt-Universität zu Berlin (2012)

[75] Wessäly, R.: Dimensioning survivable capacitated networks. Ph.D. thesis, Technische Universität Berlin (2000)

[76] Wunderling, R.: Paralleler und objektorientierter Simplex-Algorithmus. Ph.D. thesis, Technische Universität Berlin (1996)

[77] Yale Law School Roundtable on Data and Code Sharing: Reproducible research. Computing in Science and Engineering **12**, 8–13 (2010)