How to build & benchmark a solver

PD Dr. Timo Berthold
Private Lecturer @ TU Berlin
Director, Mixed-Integer Optimization @ FICO

Note: Slides are heavily inspired by Philipp Christophel's talk at MIP Workshop 2025 and re-used with his approval

Motivation

Your Project

Introduction

- Let's assume you want to implement a numerical optimization algorithm!
- Examples:
  - A new algorithm (maybe a heuristic) for a well-known problem
  - A new algorithm to solve a very special problem
  - A new algorithm for a new problem
  - An old algorithm under new circumstances (parallelism)
  - An extension of a given solver as part of a summer block course
  - ...
- This talk should give you some ideas what to look out for

Why Implement an Algorithm at All?

Introduction

The proof is in the pudding!

Probieren geht über Studieren!

Implementation Matters

Introduction

- Better implementation = better perception of the algorithm
  - Example: smart linear algebra vs. tableau in Simplex
- Computational complexity vs. empirical research
  - Example: ellipsoid method vs Simplex
- Usage scenarios
  - Example: Warm starting, special data, …
- Practical considerations
  - Example: Memory requirements, how to parallelize
- What are the bottlenecks on practical data?
- Research on implementation is research!
  - Example: Simplex papers from the 1980s, cut papers from the 1990s, heuristic papers from the 2000s, MINLP papers from the 2010s, GPU papers from the 2020s,…

First computational study of optimization algorithms in 1953

COMPUTATIONAL EXPERIENCE IN SOLVING LINEAR PROGRAMS*

A. HOFFMAN, M. MANNOS, D. SOKOLOWSKY
and N. WIEGMANN

1. Introduction. This paper is a discussion of three methods which have been employed to solve problems in linear programming, and a comparison of results which have been yielded by their use on the Standards Eastern Automatic Computer (SEAC) at the National Bureau of Standards.

- Many considerations in that paper still hold today!

Getting Started

The Algorithm (Design)

Requirements

- Before you start coding: **Try the algorithm!**
  - Create a small example and do it on paper
- Write it down in pseudocode
- Think about the theoretical effort each step should take
  - Where are the bottlenecks?
  - What doesn't matter that much?
- Which options do you want to have?
- Think about what to output into the log and where to do that
- Identify sub-algorithms and third-party components
- The algorithm will evolve as you implement it

The Data

Requirements

- What input does your algorithm take?
  - For some problems, various input formats exist
- Implement your own reader or find open-source ones to use
- Identify public libraries of test problems
- Think about how to create your own test problems
- More on test problems when we talk about benchmarking …

The Development Environment

Requirements

- Programming language
  - Clearly best choice: C (/C++)
  - Possible: FORTRAN, Rust
  - Maybe: Go, Julia
  - Python only useful as “glue” or for prototyping
- Tools
  - Integrated development environment (IDE): VS Code, Visual Studio
  - Compiler: gcc, Intel® compilers
  - Source code management: Git, GitHub
  - Debugger: gdb, UDB
  - Performance profiler: perf, Intel® VTune Amplifier
  - Memory profiler: valgrind, PurifyPlus

The Experimentation Environment

Requirements

- Machine(s) to run experiments on
  - Typically: The more the better for quicker turnaround
- Equal hardware necessary for reliable results
- Even better: Full access to configure things like power saving, turbo, …
- Cloud computing probably not a good option (shared environment)
- Calibrate your environment and measure its accuracy!
  - Run unchanged code twice, measure the difference
- Important: Noise level vs expected measurement level
  - Considerations differ when you want to measure improvements of $\pm 1\%$ vs an order of magnitude

Some More Processes

People testing their own algorithms often stop here →

A QA Tester walks into a bar:

He orders a beer.
He orders 3 beers.
He orders 2976412836 beers.
He orders 0 beers.
He orders -1 beer.
He orders q beers.
He orders nothing.
Él ordena una cerveza.
He orders a deer.
He tries to leave without paying.
He starts ordering a beer, then throws himself through the window half way through.
He orders a beer, gets his receipt, then tries to go back.
He orders a beer, goes to the door of the bar, throws a handful of cookies into the street, then goes back to the bar to see if the barmaid still recognises him.
He orders a beer, and watches very carefully while the barmaid puts his order into the till to make sure nothing in his request got lost along the way.
He starts ordering a beer, and tries to talk the barmaid into handing over her personal details.
He orders a beer, sneaks into the back, turns off the power to the till, and waits to see how the barmaid reacts, and what she says to him.
He orders a beer while calling in thousands of robots to order a beer at exactly the same time.

Source: reddit.com/r/ProgrammerHumor

Development Process

- Make it work, then make it fast
  - 1. Handle one specific test problem
  - 2. Handle normal execution cases
  - 3. Handle special cases
  - 4. Improve hotspots one at a time
- Test-driven development
  - Unit tests, asserts, debug code is what you write first
  - Sometimes, a bug can make an algorithm very fast …
    - E.g., stress test for new dual methods in MIP solvers: run without primal heuristics
- Continuous integration / continuous deployment (CI/CD)
  - Even if this just means running unit tests once a day
  - Always have running code

Benchmarking

Comparison

Benchmarking

- Comparing to a previous version
  - Changes might be very small
  - Deterministic implementation needed
- Comparing to another algorithm/implementation
  - Implement yourself or download
  - Make sure published results are replicated!
- What is the right competitor?
  - E.g., for a new open-source MIP solver, that would be SCIP or HiGHS
  - For a new general-purpose cutting plane algorithm, that could be SCIP/HiGHS/Xpress/Gurobi default vs a version that separates your cuts in a callback
  - Note: A heuristic or specialized algorithm that beats a general-purpose solver is the expected result!
  - If you implement a new heuristic for VRP → your competitor should be a state-of-the-art VRP heuristics

A major mistake (imho)

Don’t oversell!

- A new method where you can only demonstrate effectiveness on special cases, I would not sell as revolutionary for generic problems
- Compare the following abstracts for a new MIP heuristic that is tested only on a set of Multi-Knapsack Problems

We introduce a new heuristic for mixed-integer programs and demonstrate its effectiveness.

We introduce a new heuristic for mixed-integer programs and test it on multi-knapsack problems.

We introduce a new heuristic for multi-knapsack problems. We demonstrate its effectiveness and argue how it can be extended to general mixed-integer programming.

Evaluation

Benchmarking

- Evaluation methods: use more than one!
  - Means of runtime (to first solution, to optimal solution, to optimality proof)
  - Other metrics like iterations, nodes, function evaluations
  - Number of solved instances
  - Primal(-Dual) Integral
  - Subsets/brackets
  - Performance profiles (Dolan and Moré 2002)
  - Statistical tests
- Don't shy away from inventing your own evaluation method
- Meaningful graphics are great!
- Like your algorithm, your evaluation methods might evolve
- In the end, you might need to dig into the full results table and logs

It's easy to fool yourself!

What does X times faster mean?

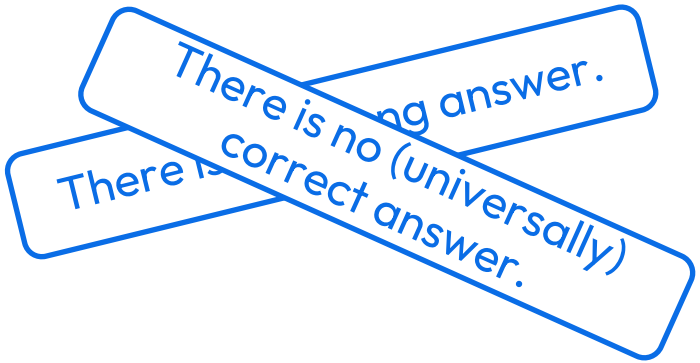
(by established community standards)

Example – How to compute a speedup (and how not to)

Benchmarking

problem	old (sec)	new (sec)
01_bench	0.00	0.00
02_marking	0.00	0.00
03_is	0.00	0.00
04_not	0.00	0.00
05_so	0.00	0.00
06_easy	0.00	0.00

On the next slide, I will fill in some runtimes.
Your task: Tell me which algorithm is faster.
And how much.



Example – How to compute a speedup (and how not to)

Benchmarking

problem	old (sec)	new (sec)
01_bench	**3.90**	**2.10**
02_marking	**4.10**	**4.10**
03_is	**3600.00**	**112.00**
04_not	**34.00**	**7.00**
05_so	**0.20**	**0.30**
06_easy	**0.001**	**10.00**

Example – How to compute a speedup (and how not to)

Benchmarking

problem	old (sec)	new (sec)	ratio
01_bench	3.90	2.10	
02_marking	4.10	4.10	
03_is	3600.00	112.00	
04_not	34.00	7.00	
05_so	0.20	0.30	
06_easy	0.001	10.00	
average	607.03	22.58	**0.04**

- Easy answer: Just take the average!
  - What's the issue here?

Example – How to compute a speedup (and how not to)

Benchmarking

problem	old (sec)	new (sec)	ratio
01_bench	3.90	2.10	
02_marking	4.10	4.10	
03_is	**3600.00**	112.00	
04_not	34.00	7.00	
05_so	0.20	0.30	
06_easy	0.001	10.00	
average	**607.03**	22.58	0.04

- Issue: Averages of multi-order numbers are dominated by the largest number(s)
  - Never use averages when your data spans multiple orders of magnitude!

Example – How to compute a speedup (and how not to)

Benchmarking

problem	old (sec)	new (sec)	ratio
01_bench	3.90	2.10	
02_marking	4.10	4.10	
03_is	3600.00	112.00	
04_not	34.00	7.00	
05_so	0.20	0.30	
06_easy	0.001	10.00	
average	607.03	22.58	0.04
geom. mean	2.71	5.22	**1.92**

- More educated answer: Take a geometric mean! It considers factors.
  - What's the issue here?

Example – How to compute a speedup (and how not to)

Benchmarking

problem	old (sec)	new (sec)	ratio
01_bench	3.90	2.10	0.54
02_marking	4.10	4.10	1.00
03_is	3600.00	112.00	0.03
04_not	34.00	7.00	0.21
05_so	0.20	0.30	1.5
06_easy	0.001	10.00	**10000**
average	607.03	22.58	0.04
geom. mean	2.71	5.22	1.92

- Issue: Geometric means are ill-defined at 0.0
  - Close to 0.0, measuring inaccuracies dominate
  - And arguably, 0.1 vs 0.001 does likely not make a huge different in practice, both is lightning-fast

Example – How to compute a speedup (and how not to)

Benchmarking

problem	old (sec)	new (sec)	ratio	shifted old	shifted new	shifted ratio
01_bench	3.90	2.10	0.54	13.90	12.10	0.87
02_marking	4.10	4.10	1.00	14.10	14.10	1.00
03_is	3600.00	112.00	0.03	3610.00	122.00	0.03
04_not	34.00	7.00	0.21	44.00	17.00	0.39
05_so	0.20	0.30	1.5	10.20	10.30	1.01
06_easy	0.001	10.00	10000	10.001	20.00	2
average	607.03	22.58	0.04			
geom. mean	2.71	5.22	1.92	38.34	20.44	

- Solution (and widely accepted standard): Shifted geometric mean
  - Shift all numbers by an offset s (here: 1sec) and take the geometric mean of the shifted values

Example – How to compute a speedup (and how not to)

Benchmarking

problem	old (sec)	new (sec)	ratio	shifted old	shifted new	shifted ratio
01_bench	3.90	2.10	0.54	13.90	12.10	0.87
02_marking	4.10	4.10	1.00	14.10	14.10	1.0
03_is	3600.00	112.00	0.03	3610.00	122.00	0.03
04_not	34.00	7.00	0.21	44.00	17.00	0.39
05_so	0.20	0.30	1.5	10.20	10.30	1.01
06_easy	0.001	10.00	10000	10.001	20.00	2
average	607.03	22.58	0.04			
geom. mean	2.71	5.22	1.92	38.34	20.44	
shifted mean				28.34	10.44	**0.37**

- Solution (and widely accepted standard): Shifted geometric mean
  - Shift all numbers by an offset s (here: 1sec) and take the geometric mean of the shifted values
  - And undo the shift again

Example – How to compute a speedup (and how not to)

Benchmarking

problem	old (sec)	new (sec)	ratio	shifted old	shifted new	shifted ratio	shift. ratio inv.
01_bench	3.90	2.10	0.54	13.90	12.10	0.87	1.15
02_marking	4.10	4.10	1.00	14.10	14.10	1.0	1.00
03_is	3600.00	112.00	0.03	3610.00	122.00	0.03	29.6
04_not	34.00	7.00	0.21	44.00	17.00	0.39	2.59
05_so	0.20	0.30	1.5	10.20	10.30	1.01	0.99
06_easy	0.001	10.00	10000	10.001	20.00	2	0.51
average	607.03	22.58	0.04				
geom. mean	2.71	5.22	1.92	38.34	20.44		
shifted mean				28.34	10.44	0.37	**2.71**

- Marketing trick: Rather report old/new than new/old
  - Factor 2.7 faster sounds better than “Takes only 37% of the runtime” or “63% faster”

Example - Sub-setting: The Wrong Way

Benchmarking

Problems where old took more than 10sec:

problem	old (sec)	new (sec)	ratio	shifted old	shifted new	shifted ratio
hard	3600.00	8.40	428.57	3610.00	18.40	196.20
easy	2800.00	3600.00	0.78	2810.00	3610.00	0.78
jim	300.00	112.00	2.68	310.00	122.00	2.54
bob	34.00	7.00	4.86	44.00	17.00	2.59
shifted				599.90	98.34	6.10

Example – Reporting on subsets

Benchmarking

Problem	old (sec)	new (sec)	ratio	shifted old	shifted new	shifted ratio
00_gosh	0.10	10.00	0.01	10.10	20.00	0.51
01_bench	3600.00	8.40	428.57	3610.00	18.40	196.20
02_marking	2800.00	3600.00	0.78	2810.00	3610.00	0.78
03_is	5.00	75.00	15.0	15.00	85.00	0.18
04_really	300.00	112.00	2.68	310.00	122.00	2.54
05_not	34.00	7.00	4.86	44.00	17.00	2.59
06_so	2.10	4.20	2.00	12.10	14.20	0.85
07_easy	5.00	4.00	0.80	15.00	14.00	1.07
shifted mean				258.58	67.27	3.84

- Typical trap: But my algorithm performs good on hard instances...

Example – Reporting on subsets

Benchmarking

Problem	old (sec)	new (sec)	shifted ratio
00_gosh	0.10	10.00	0.51
01_bench	3600.00	8.40	196.20
02_marking	2800.00	3600.00	0.78
03_is	5.00	75.00	0.18
04_really	300.00	112.00	2.54
05_not	34.00	7.00	2.59
06_so	2.10	4.20	0.85
07_easy	5.00	4.00	1.07
shifted mean			1.73

- Typical trap: My algorithm performs best on hard instances…

Example – Reporting on subsets – the wrong way

Benchmarking

Problem	old (sec)	new (sec)	shifted ratio
00_gosh	0.10	10.00	
01_bench	3600.00	8.40	196.20
02_marking	2800.00	3600.00	0.78
03_is	5.00	75.00	
04_really	300.00	112.00	2.54
05_not	34.00	7.00	2.59
06_so	2.10	4.20	
07_easy	5.00	4.00	
shifted mean			5.63

- Typical trap: My algorithm performs **extra-ordinarily well** on hard instances…
  - since it is 5.6x times faster on instances that were hard (>10sec) for the old algorithm
    - This is a one-sided, biased comparison!

Example – Reporting on subsets – the right way

Benchmarking

Problem	old (sec)	new (sec)	shifted ratio
00_gosh	0.10	10.00	0.51
01_bench	3600.00	8.40	196.20
02_marking	2800.00	3600.00	0.78
03_is	5.00	75.00	0.18
04_really	300.00	112.00	2.54
05_not	34.00	7.00	2.59
06_so	2.10	4.20	
07_easy	5.00	4.00	
shifted mean			2.12

- Correct statement: My algorithm is **twice as fast** (in sgm) on hard instances!
  - since it is 2.12x times faster on instances that were hard (>10sec) for either of the two algorithms
    - This is unbiased: The same criteria is applied to both sides ☺

Performance Variability

Performance variability

- Performance variability is a change in performance by seemingly performance-neutral changes, like changing the
  - Input format
    - Essentially, this leads to a permutation of the problem
  - Operating system
  - Different number of threads
- This occurs even though the underlying software is deterministic
- Reasons are imperfect tie-breaking, numerical round-off differences on different platforms…
- Related: Adding redundant information (E.g., a constraint $\sum_{i=1}^{n} x_i \leq n, x_i \in \{0,1\}$)

Impact of performance variability

- Performance variability
  - Can indicate improvements where there are none
  - Can shadow improvements
  - Can lead to wrong conclusions (“on some instances our method was bad, but on some, it was good”)
- Running Xpress with ten different random permutations:
  - 118/240 instances have a variability of more than a factor of 2
  - For 30, it is more than a factor of 10
  - Most extreme case: 0.02 seconds versus timeout
  - For large test sets, the effect averages out
  - Still 4% difference between „best“ and „worst“ permutation on MIPLIB2017

Exploiting Performance variability

- Performance variability can be utilized to
  - Distinguish between structural improvements and random noise
    - The more permutations/seeds you run, the clearer the picture for each single instance
  - Mimic performance on unknown instances from same application
    - Very, VERY often, customers give you one single example
- Performance variability gives you a poor man's parallelization:
  - Fire off X permutations of the problem, stop when the first one solves
  - Doesn't scale very well ;-)

Test Problems

Benchmarking

- The problems you test with matter (a LOT!)
- Use public test problems like MIPLIB, QPLIB, MINLPLIB …
  - What is the established standard in the field?
- You want reasonably-sized benchmarks, but not at each price!
  - Avoid overrepresentations of one problem-type, data-set,…
  - Requires benchmark size, besides many others, depends on the order of magnitude of change you want to measure as compared to the order of magnitude of noise you have to work against!
- Last resort: Creating problems yourself
  - Random data is usually not good
  - Better: Random perturbation of real problems
  - Also good: Real data for fake problems (e.g., real geographic data for some new VRP variant)

Some more best practices

Assorted list of advice

- State the question(s) that you want to answer and the means of doing so
  - @Reviewers: This might not be the same question and the same methodology that you would have chosen....
- Never trust your own results
  - Investigate outliers, negative and positive ones
  - Use methods like permutations, statistical tests etc to double-check your results
- Use diverse measures

A practical example

- Six test sets
  - Two categories (academic/industrial)
  - Three problem classes (MIP,MIQCP,MINLP)
- Nine performance measures in three groups
  - #Feasible solutions, #Optimality proven, #better objective, time sepnt in heuristics, primal integral, time to first, #solutions, #branch&bound nodes, solve time
- Statistical tests where suitable, same experiment repeated on five permutations

Table 11.1.: Impact of primal heuristics (academic benchmarks)

	all					all feas		all opt	
	feas	opt	obj	t_{heur}	$\frac{P(t_{max})}{t_{max}}$	$time_1$	sols	nodes	time
MMM			(167 instances)			(149 instances)		(116 instances)	
default	160	121	24	11.0 %	8.9 %	9.3	32.2	2 348	78.9
no heur	149	116	3	–	17.2 %	35.3	5.7	3 538	90.9
GLOMIQO			(167 instances)			(137 instances)		(118 instances)	
default	156	118	25	10.2 %	9.0 %	3.4	11.2	559	9.8
no heur	138	119	3	–	19.7 %	7.8	4.1	689	10.4
MINLPLIB			(240 instances)			(196 instances)		(160 instances)	
default	218	162	47	11.6 %	15.5 %	1.9	8.7	780	8.9
no heur	196	162	0	–	23.5 %	6.3	3.8	1 005	9.1

Table 11.3.: Impact of primal heuristics (real-world instances)

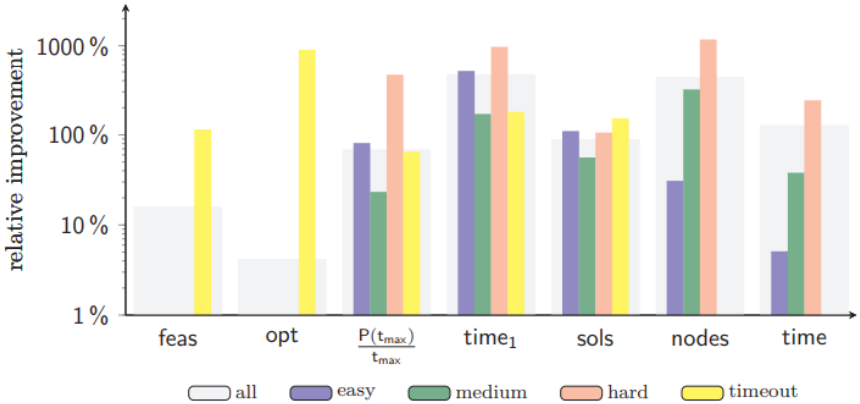
	all					all feas		all opt	
	feas	opt	obj	t_{heur}	$\frac{P(t_{max})}{t_{max}}$	$time_1$	sols	nodes	time
SAP			(40 instances)			(17 instances)		(16 instances)	
default	36	16	19	13.1 %	22.2 %	3.9	4.7	137	4.4
no heur	17	16	0	–	57.6 %	4.7	2.0	259	5.6
SIEMENS			(27 instances)			(0 instances)		(0 instances)	
default	19	0	19	7.4 %	37.1 %	–	–	–	–
no heur	0	0	0	–	100.0 %	–	–	–	–
FORNE			(430 instances)			(198 instances)		(293 instances)	
default	221	315	23	6.8 %	25.8 %	3.0	1.0	84	3.2
no heur	198	293	0	–	36.0 %	30.7	1.0	1 684	18.7

A practical example

- A completely different split of the same instances
  - 4 disjoint subsets that form the „all“ set
- Nine performance measures in three groups
  - #Feasible solutions, #Optimality proven, #better objective, time sepnt in heuristics, primal integral, time to first, #solutions, #branch&bound nodes, solve time
- Additionally, a visualization, normalized s.t. positive values correspond to improvements
  - Logarithmic scale

Table 11.5.: Impact of primal heuristics (all instances, grouped by the number of branch-and-bound nodes of the worse setting).

	all					all feas		all opt	
	feas	opt	obj	t_{heur}	$\frac{P(t_{max})}{t_{max}}$	$time_1$	sols	nodes	time
all			(983 instances)			(630 instances)		(639 instances)	
default	732	669	144	9.2 %	17.8 %	4.3	4.3	322	9.4
no heur	631	642	6	–	30.1 %	24.2	2.3	1 739	21.6
easy			(195 instances)			(136 instances)		(195 instances)	
default	136	195	0	7.1 %	0.2 %	0.8	3.1	9	1.6
no heur	136	195	0	–	0.3 %	5.1	1.5	13	1.6
medium			(264 instances)			(232 instances)		(264 instances)	
default	232	264	0	19.3 %	1.7 %	3.2	3.5	197	7.0
no heur	232	264	0	–	2.1 %	8.5	2.2	828	9.6
hard			(180 instances)			(175 instances)		(180 instances)	
default	175	180	0	13.5 %	1.1 %	5.1	9.1	7 554	57.2
no heur	175	180	0	–	6.5 %	54.4	4.4	96 135	195.4
timeout			(344 instances)			(87 instances)		(0 instances)	
default	189	30	144	4.6 %	47.2 %	3.7	2.1	–	–
no heur	88	3	6	–	77.9 %	10.4	0.8	–	–



A practical example: Xpress 9.6 vs Xpress 9.9 (spring 2026 vs spring 2025)

			143693				148277				
class	#D	#I	#S	time	nodes	PDI	#S	tQ	nQ	pdiQ	W/L
all	6171	993.00	899.31	149.76	2442	38.54	908.35***	0.91***	0.82***	0.91***	51/18
[0,3600}	5748	926.18	899.31	117.58	2084	31.33	908.35	0.91	0.81	0.90	51/18
[1,3600}	5745	925.80	898.93	117.71	2089	31.35	907.96	0.91	0.81	0.90	51/18
[10,3600}	5242	835.35	808.47	149.30	2797	37.15	817.51	0.90	0.81	0.89	47/18
[100,3600}	3174	469.85	442.98	469.77	10537	78.69	452.02	0.85	0.78	0.86	32/6
[1000,3600}	1086	155.89	129.02	1707.54	53756	184.86	138.05	0.74	0.68	0.75	12/2
all-opt	5393	881.47	881.47	99.87	1744	27.97	881.47	0.92	0.82	0.92	42/17
[0,1)	3	0.38	0.38	0.38	1	0.38	0.38	1.53	1.00	1.52	0/0
[1,10)	503	90.45	90.45	6.59	133	2.30	90.45	0.94	0.81	0.96	1/0
[10,100)	2068	365.49	365.49	28.61	502	10.93	365.49	0.97	0.86	0.95	11/9
[100,1000)	2088	313.96	313.96	244.69	4688	50.00	313.96	0.91	0.83	0.92	16/4
[1000,3600)	731	111.18	111.18	1483.54	48575	175.89	111.18	0.79	0.68	0.82	2/1

- Five measures: Time, Nodes, PID, solved instances, consistent wins (on all permutations)
  - 993 instances, 6171 data points (5 or 11 permutations, depending on variability)
- Shifted geometric means, speedup factors (smaller is better)
- Statistical tests (McNemar and Wilcoxon signed rank)
- Unbiased brackets by complexity, all vs all-opt vs path-changes (not depicted here)
- Diverse test set (at most X per customer)
  - Few trivial problems, few „unsolvable“
  - There are many additional test sets: HEUREMPHASIS for hard, sub 1s for trivial problems, LP, MINLP,...

Final considerations

Can I ever achieve all of this?

No.

No software out there is bug-free, perfectly designed, considers all edge-cases and has 100% test coverage

~~Truisms~~ wise words

Conclusions

Perfect is the enemy of good

Das Bessere ist der Feind des Guten

Better a diamond with a flaw than a pebble without one

Some Things to Keep in Mind

Conclusions

- This sounds daunting!
  - Algorithms evolve from failure
  - Programming is learned by doing it
- To justify research, small improvements are often good enough
- There are few things as rewarding as knowing that the code you wrote is used in practice
  - If you work on an MIP solver: tens of thousands of times, by the biggest companies in the world

Publish your computational studies!

Mathematical Programming Computation

A Publication of the Mathematical Optimization Society

Publishing model

Hybrid

Springer

Mathematical Optimization Society

Submit your manuscript →

Save journal

Explore open access funding | Select institution

About this journal | Articles | For authors | Journal updates

Overview

Mathematical Programming Computation (MPC) publishes original research articles advancing the state of the art of practical computation in Mathematical Optimization and closely related fields. Authors are required to submit software source code and data along with their manuscripts (while open-source software is encouraged, it is not required). Where applicable, the review process will aim for verification of reported computational results. Topics of articles include:

- New algorithmic techniques, with substantial computational testing
- New applications, with substantial computational testing
- Innovative software
- Comparative tests of algorithms
- Modeling environments
- Libraries of problem instances

Journal metrics

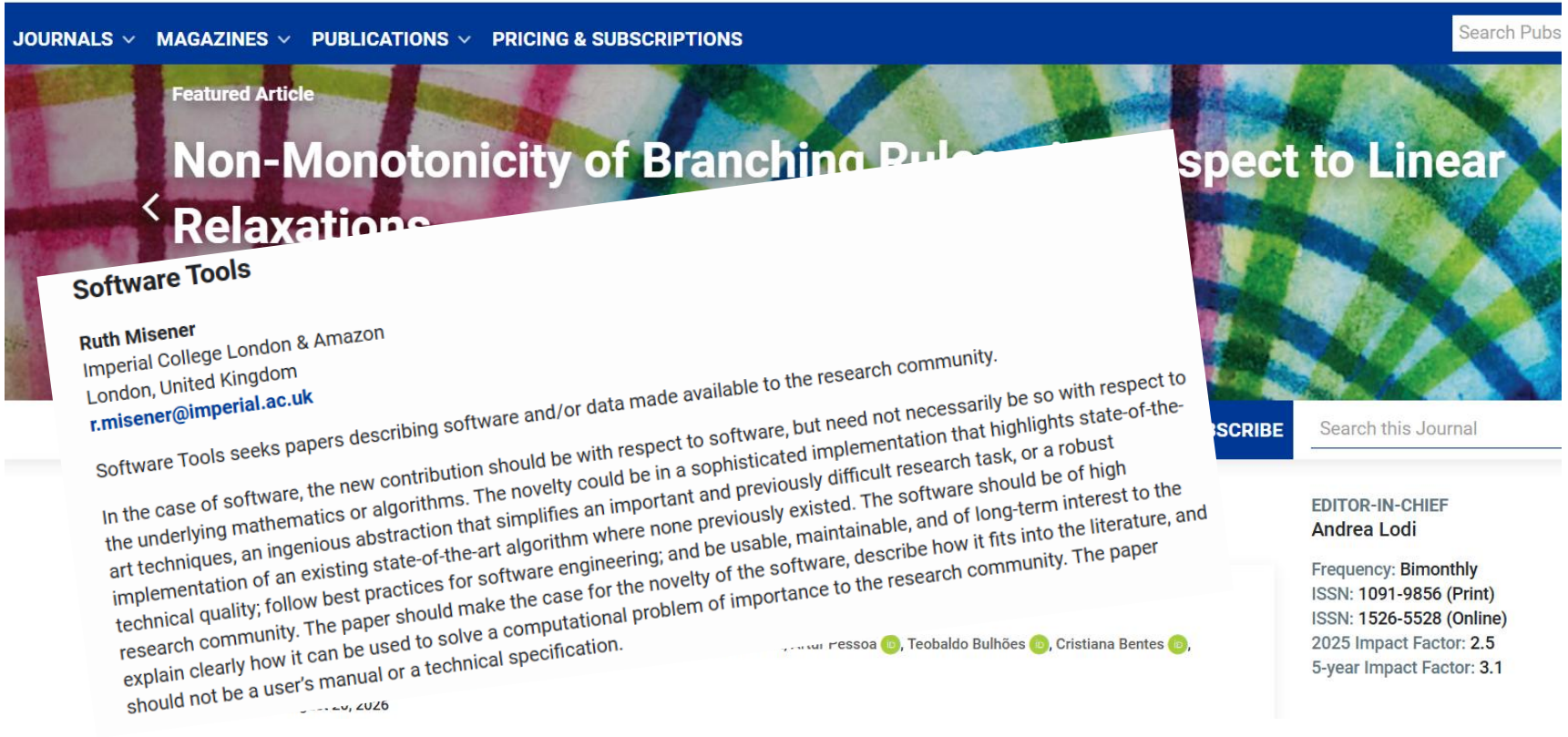
Journal Impact Factor
4.9 (2025)

5-year Journal Impact Factor
6.8 (2025)

Downloads
103.3k (2025)

Publish your computational studies!

informs PubsOnLine



Thank You!

PD Dr. Timo Berthold
Private Lecturer @ TU Berlin
Director, Mixed-Integer Optimization @ FICO