On the role of randomness in exact tree search methods

Matteo Fischetti, University of Padova
(based on joint work with Michele Monaci)

EURO XXV, Vilnius, July 2012

Part I: cutting planes

THE POWER OF 10

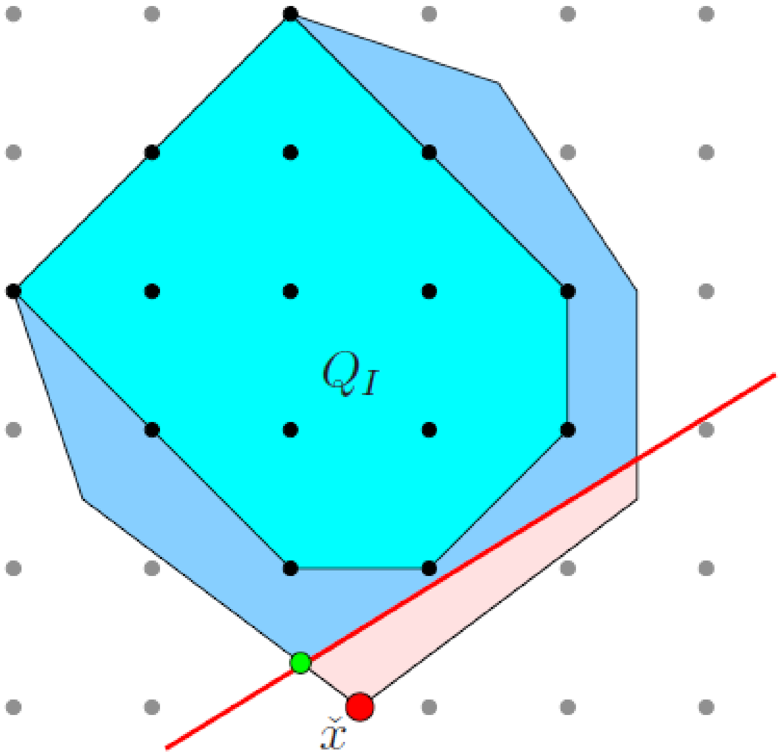
Cutting planes for MIPs

- Cutting planes are crucial for solving hard MIPs

- **Useful** to tighten bounds but…
- … also potentially **dangerous** (heavier LPs, numerical troubles, etc.)

- Every solver has its own recipe to handle them

- **Conservative** policies are typically implemented (at least, in the default)



Measuring the power of a single cut

- Too many cuts might hurt …

  … what about a **single** cut?

- The added single cut can be beneficial because of
  - root-node bound improvement
  - better pruning along the enumeration tree
  - **but also**: improved preprocessing and variable fixing, etc.

- **Try to measure what can be achieved by a single cut to be added to the given initial MIP formulation**
- … thus allowing the black-box MIP solver to take full advantage of it

Rules of the game

- We are given a MIP described through an input .LP file

  (MIP) $\min \{ z : \quad z = c\,x, \quad Ax \sim b, \quad x_j \textit{ integer } \; j \in J \}$

- We are allowed to generate a **single** valid cut $\boldsymbol{\alpha\, x \geq \alpha_0}$

- … and to append it to the given formulation to obtain

  (MIP++) $\min \{ z : \; \boldsymbol{\alpha\, x \geq \alpha_0}, \; z = c\,x, \quad Ax \sim b, \quad x_j \textit{ integer } \; j \in J \}$

- Don't cheat: CPU time needed to generate the cut must be **comparable** with CPU time to solve the root-node LP

- Apply a same black-box MIP solver to both MIP and MIP++
- … and compare computing times to solve both to proven optimality

Testbed

- ✓ We took all the instances in the MIPLIB 2003 and COR@L libraries and solved them through IBM ILOG Cplex 12.2 (default setting, no upper cutoff, single-thread mode) on an Intel i5-750 CPU running at 2.67GHz.

- ✓ We disregarded the instances that turned out to be too “easy" → can be solved within just 10,000 nodes or 100 CPU seconds on our PC

- ✓ Final testbed containing 38 hard instances

Computational setting

- MIP black-box solver: IBM ILOG Cplex 12.2 (single thread) with default parameters; 3,600 CPU sec.s time limit on a PC.

- To reduce side-effects due to heuristics:
  - Optimal solution value as input cutoff
  - No internal heuristics (useless because of the above)

- Comparison among **10** different methods:
  - Method **#0**: Cplex default (no cut added)
  - Methods **#1-9**: nine variants to generate a single cut

Computational results

	Avg. sec.s	Avg. nodes	Time ratio	Node ratio
Default (no cut)	**533,00**	**64499,09**	**1,00**	**1,00**
Method #1	**397,50**	**37194,89**	**0,75**	**0,58**
Method #2	419,22	44399,47	0,79	0,69
Method #3	468,87	48971,72	0,88	0,76
Method #4	491,77	46348,39	0,92	0,72
Method #5	582,42	58223,10	**1,09**	0,90
Method #6	425,38	43492,35	0,80	0,67
Method #7	457,95	46067,74	0,86	0,71
Method #8	446,89	44481,75	0,84	0,69
Method #9	419,57	41549,07	0,79	0,64

Cases with large speedup

	NO CUT		METHOD #1		
	Time	Nodes	Time	Nodes	Time Speedup
glass4	43,08	118.151	12,95	17.725	3,33
neos-1451294	3.590,27	20.258	102,94	521	34,88
neos-1593097	149,94	10.879	16,12	508	9,30
neos-1595230	1.855,69	152.951	770,6	89.671	2,41
neos-603073	452,4	36.530	130,75	10.017	3,46
neos-911970	3.588,54	5.099.389	3,29	1.767	1.090,74
ran14x18_1	3.287,59	1.480.624	2.066,70	759.265	1,59

Conclusions

1. We have proposed a new cut-generation procedure
2. … to generate **just one cut** to be appended to the initial formulation

3. Computational results on a testbed of 38 hard MIPs from the literature have been presented
4. … showing that an **average speedup of 25%** can be achieved w.r.t. Cplex

5. A key ingredient of our method is not to overload the LP by adding too many cuts → single cut mode

Can you just describe the 10 methods?

- Method # 0 is the default (no cut added)
- All other methods add a single cut obtained as follows (assume *x* ≥ *0*)
  - Step 1. Choose a variable permutation

$$x_{\pi(1)}, \cdots, x_{\pi(n)}$$

  - Step 2. Obtain a single valid inequality through lifting as

$$\sum_{i=1}^{n} \alpha_{\pi(i)} x_{\pi(i)} \geq \alpha_0$$

How about variable permutations?

- Nine different policies for the nine methods:

Seed = 1. Pseudo random sequence
2. Pseudo random sequence
3. Pseudo random sequence
4. Pseudo random sequence
5. Pseudo random sequence
6. Pseudo random sequence
7. Pseudo random sequence
8. Pseudo random sequence
9. Pseudo random sequence

How about lifting?

- To have a fast lifting, we specialize

$$\sum_{i=1}^{n} \alpha_{\pi(i)} x_{\pi(i)} \geq \alpha_0$$

- to

$$\sum_{i=1}^{n} 1 \cdot x_{\pi(i)} \geq \alpha_0$$

- and finally to

$$\sum_{i=1}^{n} 1 \cdot x_{\pi(i)} \geq -1$$

Where is the trick?

- The additional cut is of course **redundant** and hence removed

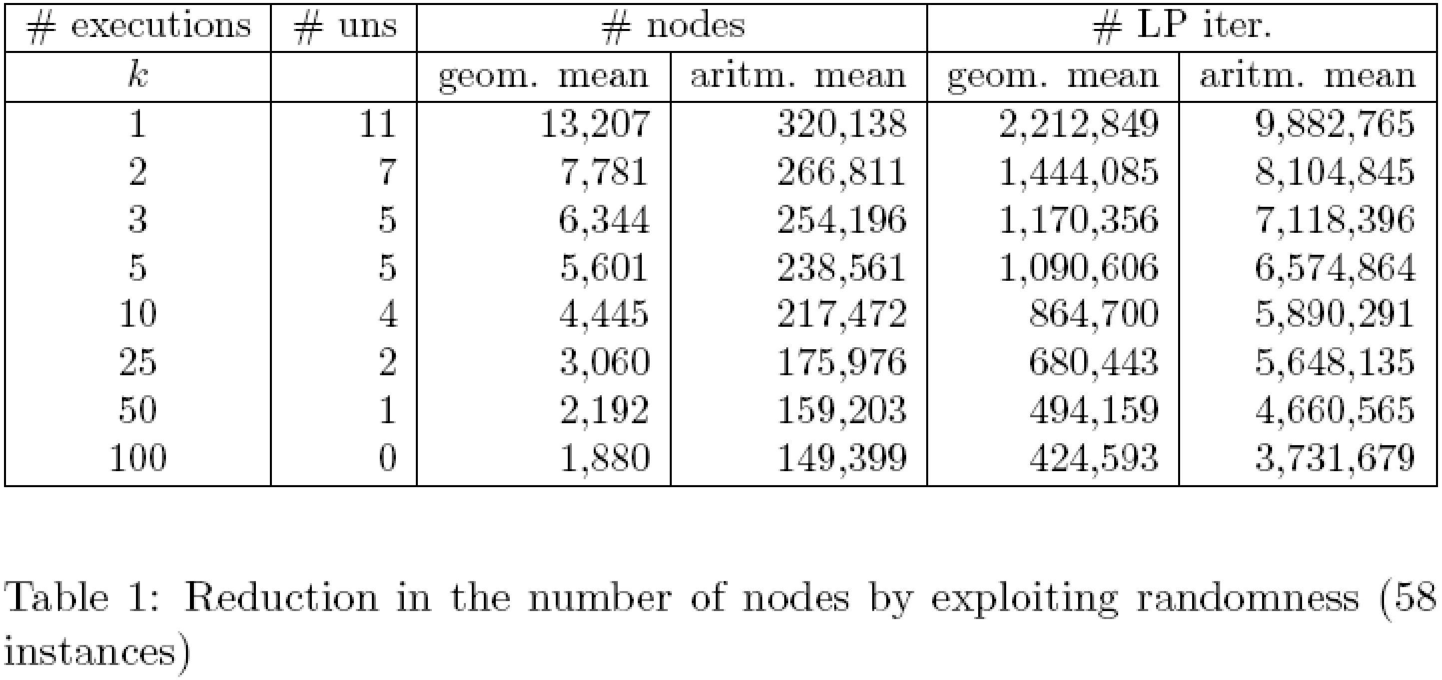
- Minor changes (including var. order in the LP file) …change **initial conditions** (col. sequence etc.)

- Tree search is very sensitive to initial conditions …as branching acts as a **chaotic amplifier** → **the pinball effect**

- (Some degree of) erraticism is intrinsic in tree-search nature …
- … you cannot avoid it (important for experiment design)
- … and you better try to turn it to your advantage
- … though you will never have a complete control of it

Parallel independent runs

- Experiments with *k* independent runs with randomly-perturbed initial conditions (Cplex 12.2 default, single thread)



# executions	# uns	# nodes		# LP iter.	
k		geom. mean	aritm. mean	geom. mean	aritm. mean
1	11	13,207	320,138	2,212,849	9,882,765
2	7	7,781	266,811	1,444,085	8,104,845
3	5	6,344	254,196	1,170,356	7,118,396
5	5	5,601	238,561	1,090,606	6,574,864
10	4	4,445	217,472	864,700	5,890,291
25	2	3,060	175,976	680,443	5,648,135
50	1	2,192	159,203	494,159	4,660,565
100	0	1,880	149,399	424,593	3,731,679

Table 1: Reduction in the number of nodes by exploiting randomness (58 instances)

A nice surprise

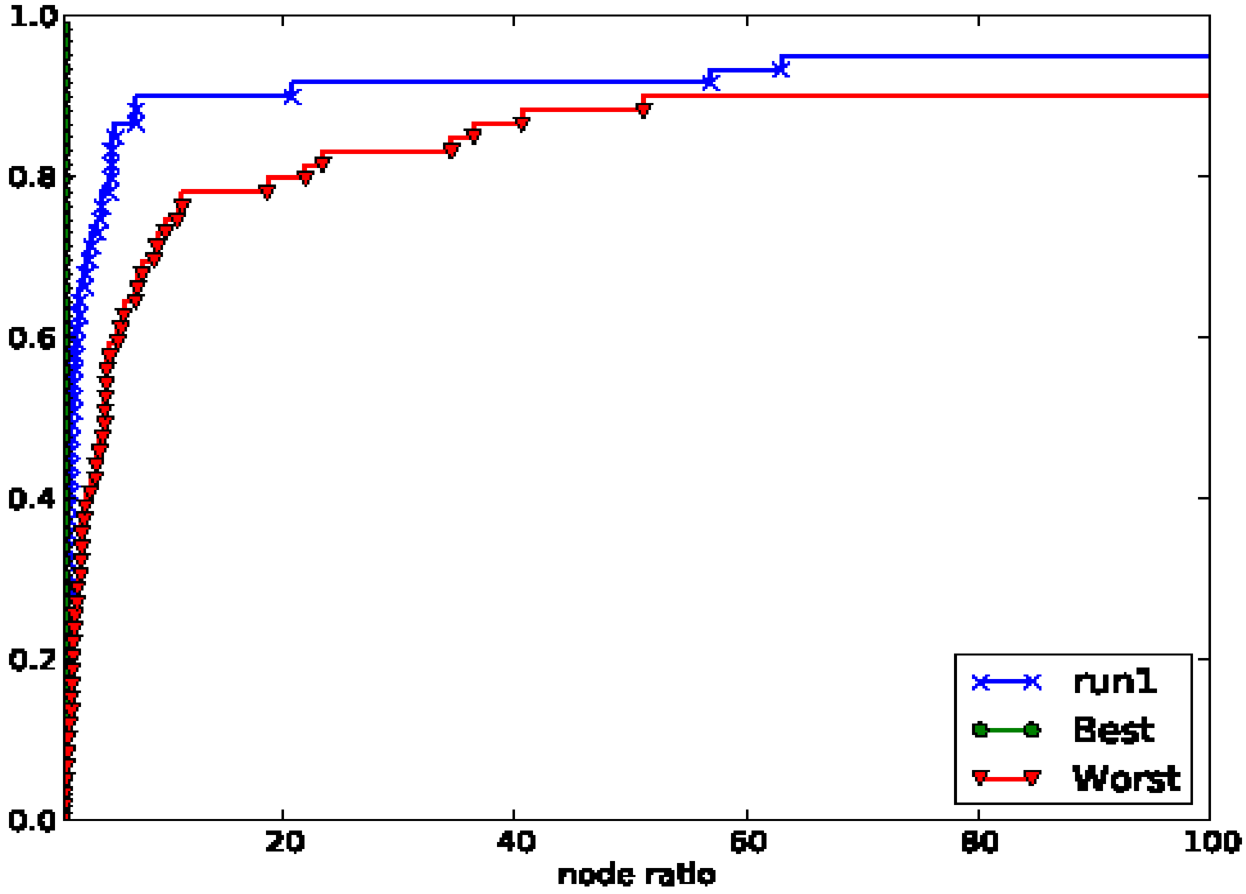
- Incidentally, during these experiments we were able to solve to proven optimality, for the first time, the very hard MIPLIB 2010 instance **buildingenergy**

- One of our parallel runs (k=2) converged after 10,899 nodes and 2,839 CPU seconds of a IBM power7 workstation → integer solution of value 33,285.4433 → optimal within default tolerances

- We then reran Cpx12.2 (now with 8 threads) with optimality tolerance zero and initial upper bound of 33,285.4433 → 0-tolerance optimal solution of value 33,283.8532 found after 623,861 additional nodes and 7,817 CPU sec.s

Cplex vs Cplex

- 20 runs of Cplex 12.2 (default, 1 thread) with scrambled rows&col.s
- 99 instances from MIPLIB 2010 (Primal and Benchmark)



Part II: Exploiting erraticism

- A simple **bet-and-run** scheme
  - Make KTOT independent short runs with randomized initial conditions, and abort them after MAX_NODES nodes
  - Take statistics at the end of each short run (total depth and n. of open nodes, best bound, remaining gap, etc.)
  - Based on the above statistics, choose the most promising run (say the k-th one)
  - “Bet on” run k, i.e., restore exactly the initial conditions of the k-th run and reapply the solver from scratch (without node limit)

Discussion

- Similar approaches already used for solving very hard problems (notably, QAPs etc.), by trying different parameter configurations and estimating the final tree size in a clever way

- The underlying “philosophy” is that a **BEST parameter configuration** exists somewhere and could be found if we were clever enough

- Instead, we do not pretend to find a best-possible tuning of solver’s param.s (whatever this means)
- … our order of business here is to play with **randomness** only

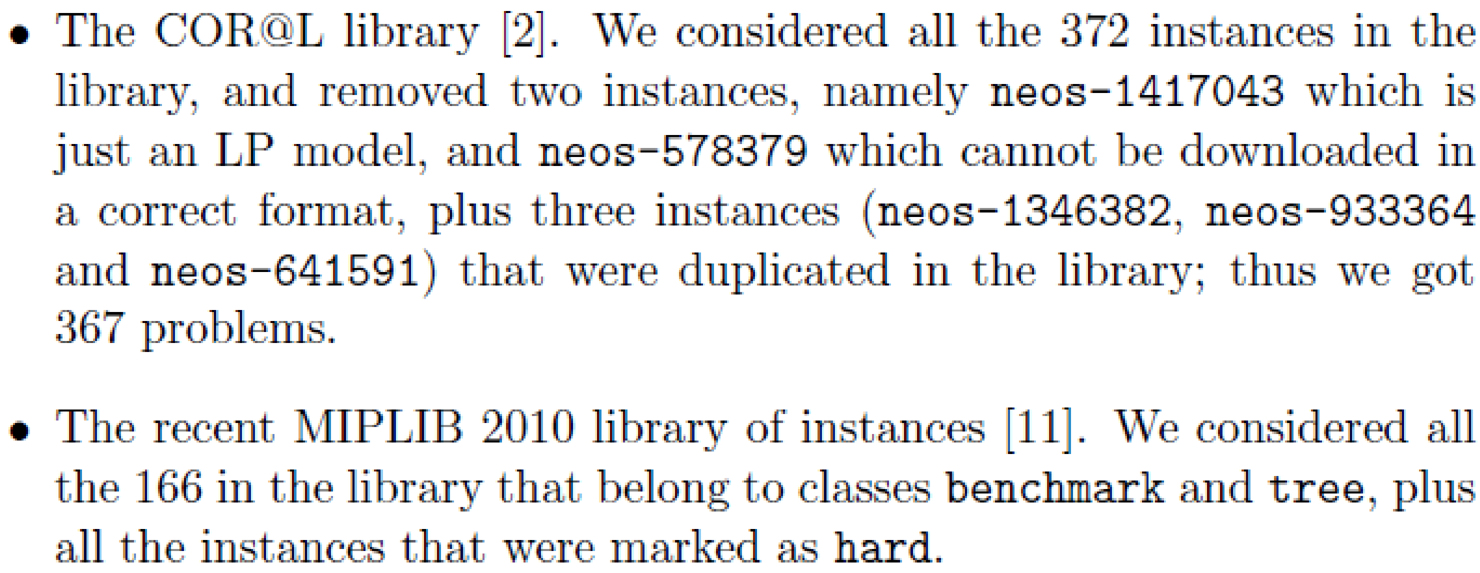
- We apply a **very quick-and-dirty** selection criterion for the run to bet on
- … as we know that no criterion can be perfect → what we are looking for is just a **positive correlation** with the a-posteriori best run

Some experiments

IBM ILOG Cplex 12.2 (single thread, default without dynamic search)

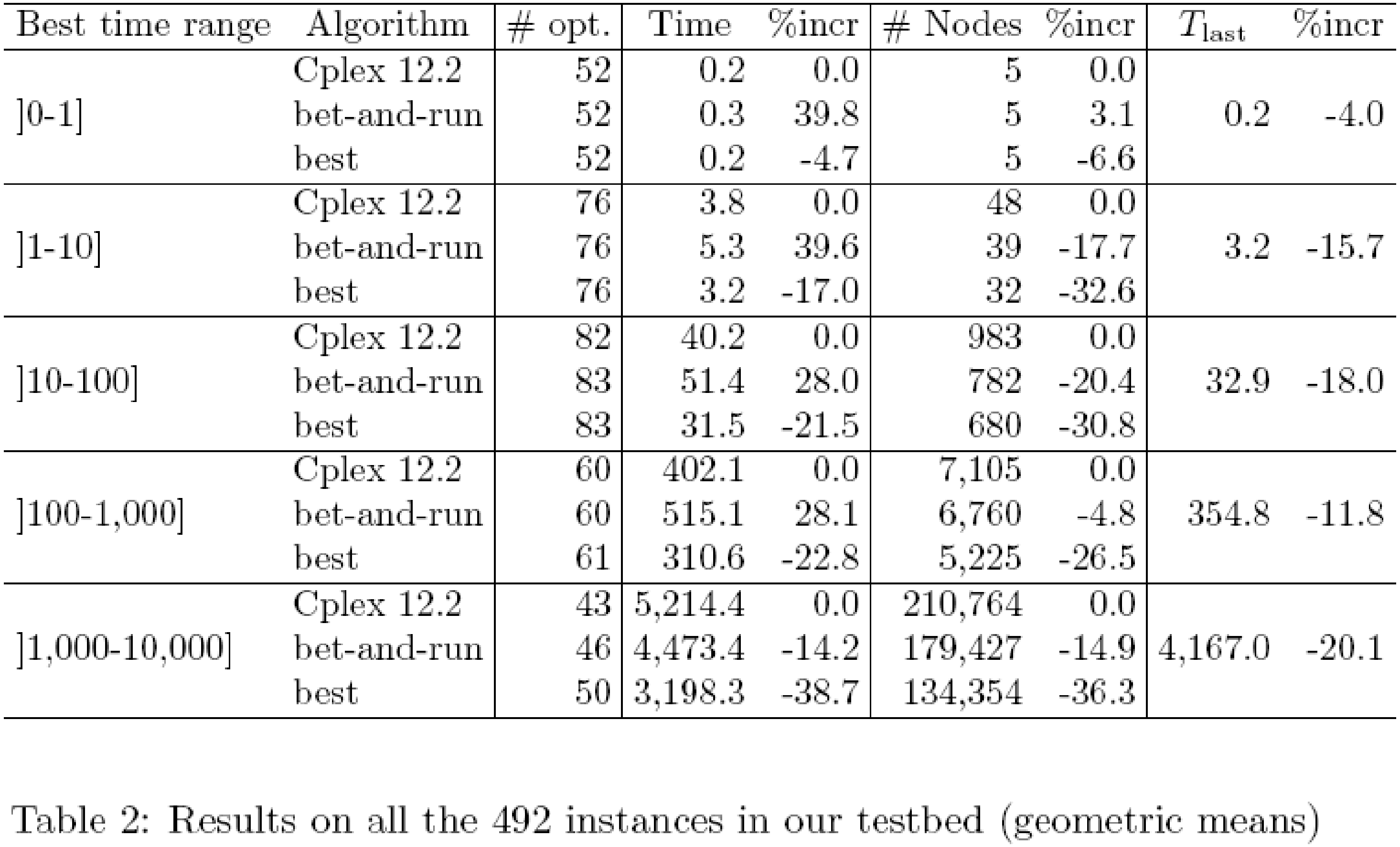
Time limit: 10,000 CPU sec.s on a PC i5-750@2.67GHz

Large testbed with 492 instances taken from:



- The COR@L library [2]. We considered all the 372 instances in the library, and removed two instances, namely `neos-1417043` which is just an LP model, and `neos-578379` which cannot be downloaded in a correct format, plus three instances (`neos-1346382`, `neos-933364` and `neos-641591`) that were duplicated in the library; thus we got 367 problems.
- The recent MIPLIB 2010 library of instances [11]. We considered all the 166 in the library that belong to classes `benchmark` and `tree`, plus all the instances that were marked as `hard`.

Outcome (5 short runs, 5 nodes each)



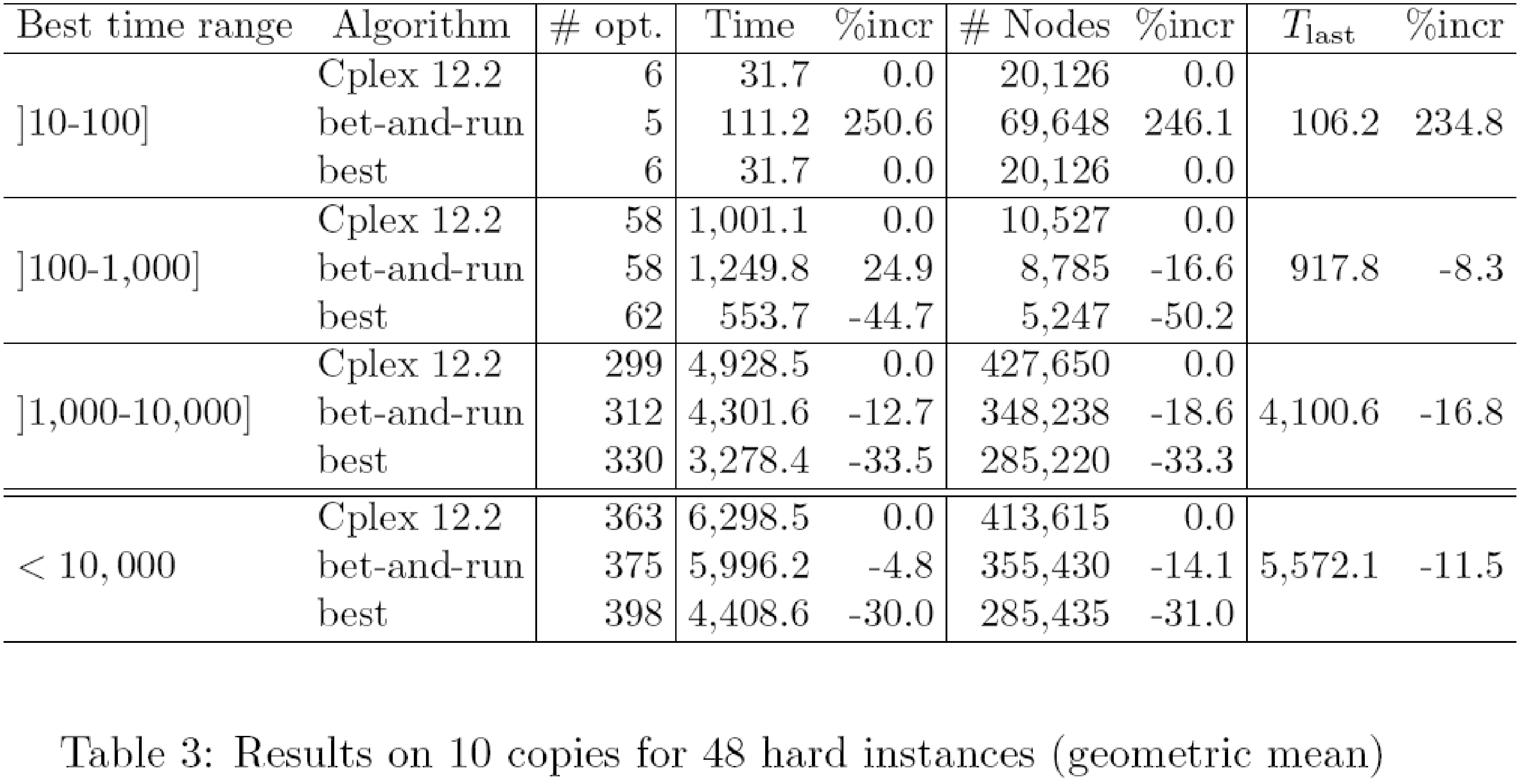
Best time range	Algorithm	# opt.	Time	%incr	# Nodes	%incr	T_{last}	%incr
]0-1]	Cplex 12.2	52	0.2	0.0	5	0.0	0.2	-4.0
	bet-and-run	52	0.3	39.8	5	3.1		
	best	52	0.2	-4.7	5	-6.6		
]1-10]	Cplex 12.2	76	3.8	0.0	48	0.0	3.2	-15.7
	bet-and-run	76	5.3	39.6	39	-17.7		
	best	76	3.2	-17.0	32	-32.6		
]10-100]	Cplex 12.2	82	40.2	0.0	983	0.0	32.9	-18.0
	bet-and-run	83	51.4	28.0	782	-20.4		
	best	83	31.5	-21.5	680	-30.8		
]100-1,000]	Cplex 12.2	60	402.1	0.0	7,105	0.0	354.8	-11.8
	bet-and-run	60	515.1	28.1	6,760	-4.8		
	best	61	310.6	-22.8	5,225	-26.5		
]1,000-10,000]	Cplex 12.2	43	5,214.4	0.0	210,764	0.0	4,167.0	-20.1
	bet-and-run	46	4,473.4	-14.2	179,427	-14.9		
	best	50	3,198.3	-38.7	134,354	-36.3		

Table 2: Results on all the 492 instances in our testbed (geometric means)

Validation

The previous table shows a 15% speedup for hard cases in class]1,000-10,000]

Validation on 10 copies of each hard instance (random rows&col.s scrambling)



Best time range	Algorithm	# opt.	Time	%incr	# Nodes	%incr	T_{last}	%incr
]10-100]	Cplex 12.2	6	31.7	0.0	20,126	0.0	106.2	234.8
	bet-and-run	5	111.2	250.6	69,648	246.1		
	best	6	31.7	0.0	20,126	0.0		
]100-1,000]	Cplex 12.2	58	1,001.1	0.0	10,527	0.0	917.8	-8.3
	bet-and-run	58	1,249.8	24.9	8,785	-16.6		
	best	62	553.7	-44.7	5,247	-50.2		
]1,000-10,000]	Cplex 12.2	299	4,928.5	0.0	427,650	0.0	4,100.6	-16.8
	bet-and-run	312	4,301.6	-12.7	348,238	-18.6		
	best	330	3,278.4	-33.5	285,220	-33.3		
$< 10,000$	Cplex 12.2	363	6,298.5	0.0	413,615	0.0	5,572.1	-11.5
	bet-and-run	375	5,996.2	-4.8	355,430	-14.1		
	best	398	4,408.6	-30.0	285,435	-31.0		

Table 3: Results on 10 copies for 48 hard instances (geometric mean)

Conclusions

- Erraticism is just a consequence of the exponential nature of tree search, that acts as a **chaotic amplifier**, so it is (to some extent) unavoidable → you have to cope with it somehow!

- Tests are biased if "we test our method on the training set"

- The more parameters, the easier to make overtuning → power-of-ten effect

- Removing "instances that are easy for our competitor" is not fair

- When comparing methods A and B, the instance classification must be the same if A and B swap → blind wrt the name of the method

Conclusions

- High-sensitivity to initial conditions is generally viewed as a drawback of tree search methods, but it can be a **plus**

- We have proposed a **bet-and-run** approach to actually turn erraticism to one's advantage

- Computational results on a large MIP testbed show the potential of this simple approach… though more extensive tests are needed

- More clever selection on the run to bet on → possible with a better classification method (**support vector machine & alike**)?

- **Hot topic**: exploiting randomness in a massive **parallel** setting…

Thanks for your attention