Cutting planes:
don’t back yourself into the corner!

Matteo Fischetti and Domenico Salvagnin
University of Padova

Bertinoro, 30/11-2/12/2009

Cutting plane methods

- **Cutting plane** methods widely used in convex optimization and to provide bounds for Mixed-Integer Programs (MIPs)

- Made by two equally important components:
  - (i) the **separation procedure** (oracle) that produces the cut(s) used to tighten the current relaxation, and
  - (ii) the overall **search framework** that actually uses the generated cuts and determines the next point to cut

- In the last 50 years, considerable research effort devoted to the study of (i) → families of cuts, cut selection criteria, etc.

- Search component (ii) much less studied by the MIP community → the standard approach is to always cut an optimal LP vertex

The problem

- Let's focus on a generic MIP: $z(PLI) := min \{ c^T x : x \varepsilon conv(X) \}$

- We are given an LP relaxation

$$z := min \{c^T x: x \varepsilon P \}, \qquad P := \{ x : A x \leq b \}$$

- We are also given a set P_1 with $conv(X) \leq P_1 \leq P,$
  described only implicitly through a separation function:

  oracle(y) returns a valid inequality for P_1 violated by y (if any)

- We want to compute $z_1 := min \{c^T x: x \varepsilon P_1 \}$

Kelley's cutting plane method

- A classical search scheme

  J. E. Kelley. The cutting plane method for solving convex programs, Journal of the SIAM, 8:703-712, 1960.

  – Let *P' := { x ε P : x satisfies all cuts generated so far}*

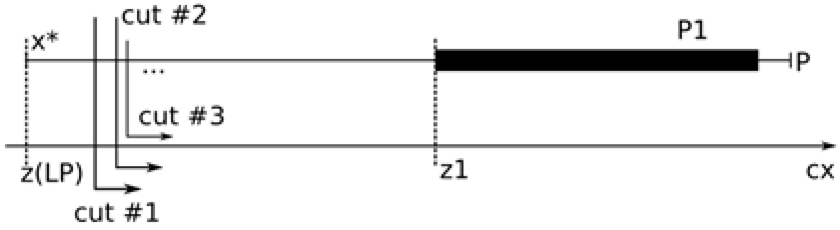
  – Find an optimal **vertex** x^* of the current LP: $\min \{c^T x: x \in P'\}$,

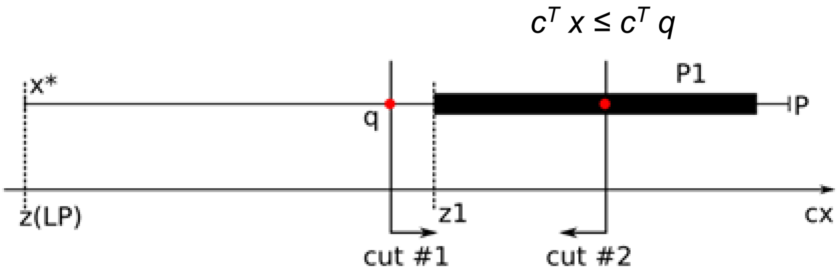
  – Invoke *oracle(x*)* and repeat (if a violated cut is found)

- Practically satisfactory only in case the oracle is able to find "**deep**" cuts (e.g., defining facets of P_1 or, al least, supporting it).
- Very ineffective in case shallow cuts are generated
- May induce a dangerous correlation between x^* and the returned cut (e.g. when the cuts are read from the LP tableau)

1-dimensional problems: binary search

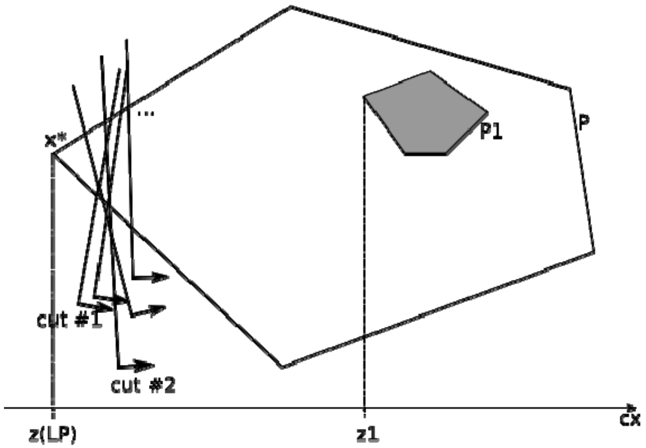
- Kelley's method very unnatural (and inefficient) for 1-dim. problems
- The most effective search scheme available for 1D is **binary search**, invoking *oracle(**q**)* for the middle point ***q*** of *P'*
- Its convergence does not depend on the cut quality (the cut needs not be deep—a cut just tight at *q* suffices!!)

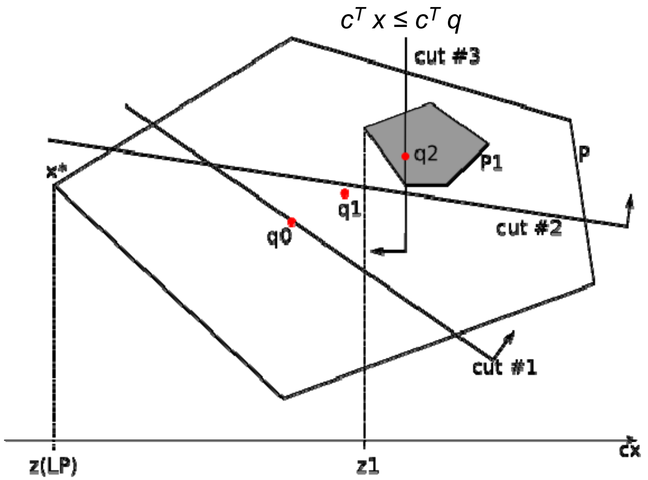




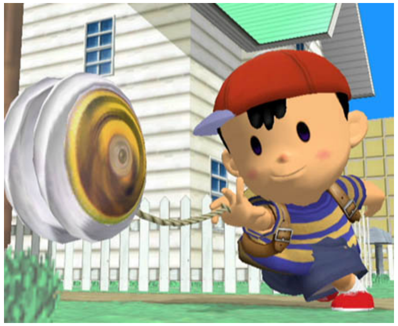
Ellipsoid & analytic center methods

- Generalize binary search to the multi-dimensional case: at each iteration, a **corepoint** q in the **relative interior** of P' is computed and passed to the oracle
- If no cut is generated, then q ε P_1 and the ***neutral cut*** $c^T x \leq c^T q$ (tight at q) is added—in this context, even a tight cut works!
- The overall convergence does not depend (too much) on the quality of the oracle's cut, but the computation of corepoint q can be heavy





A hybrid method: yoyo search



- Two is better than one: maintain <u>two</u> points (x^*,q)
  - x^* in an **optimal vertex** of *P'*, as in the Kelleys' method
  - q is an **internal point** of P_1, in the spirit of corepoint methods

  where the uncertainty interval $[c^T x^*, c^T q]$ contains the unknown z_1

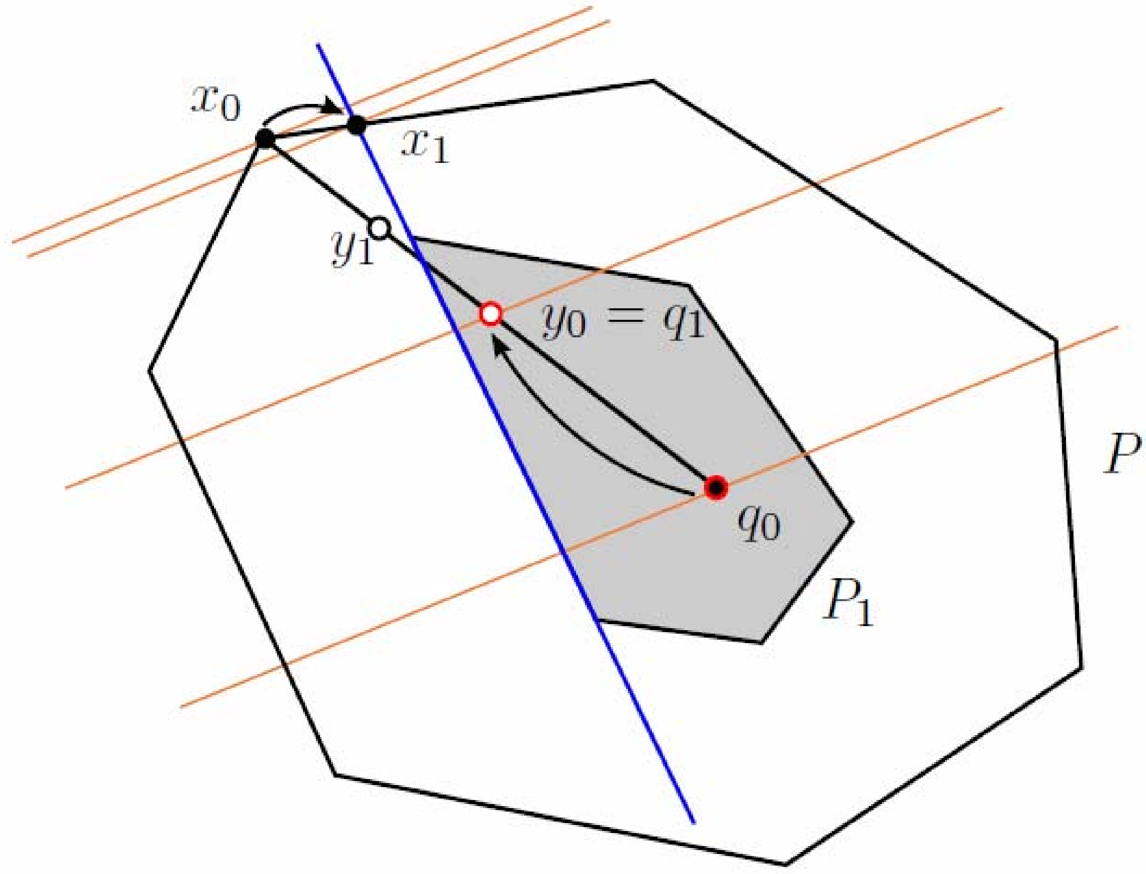
- Think of the line segment $[x^*,q]$ as in the 1-dimensional binary search, and invoke *oracle(y)* for its middle point $y := (x^* + q)/2$

- Two possible outcomes:

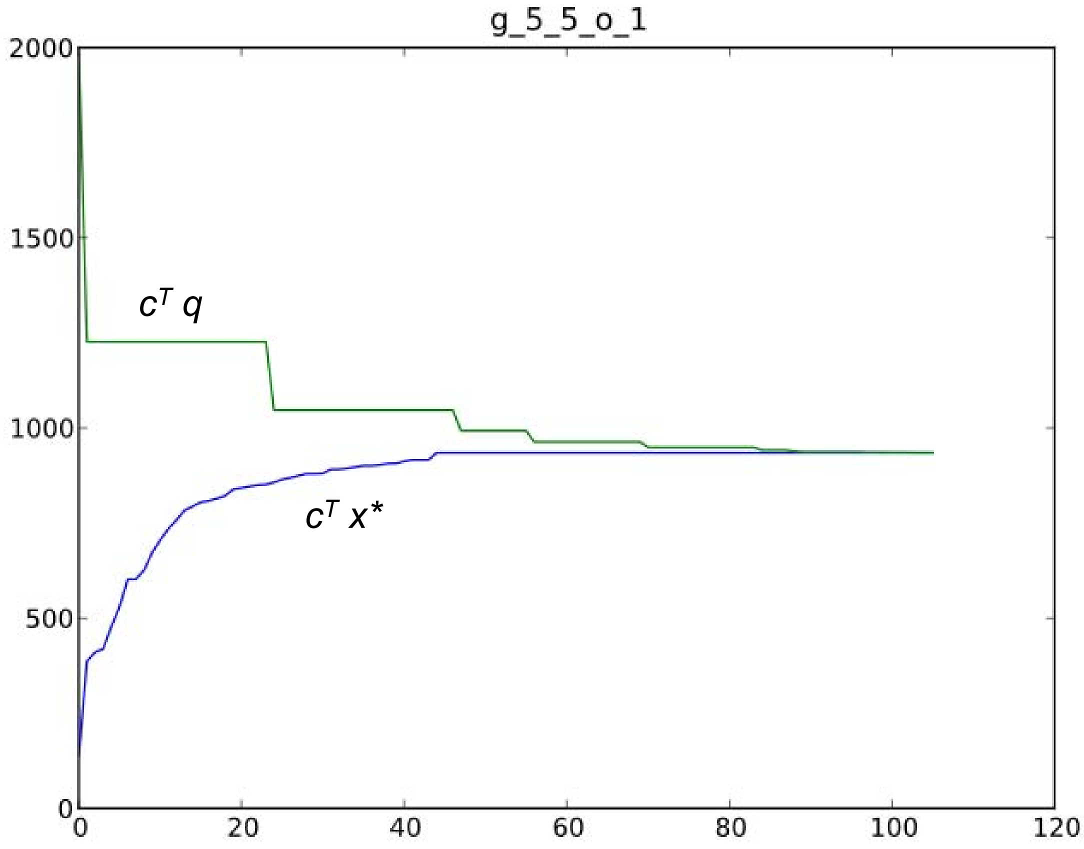
  (1) If a cut is returned, add it to P' (the cut is likely to be **deep!**)

  (2) **otherwise update $q := y$ (this <u>halves</u> the uncertainty interval!)**

An example of yoyo search



The two bound trajectories



yoyo search: pros and cons

PROS

Wrt Kelley's method

→ much **deeper** cuts are typically generated

Wrt corepoint methods

→ **no extra-time** to update the internal point q

→ no neutral cuts generated, hence non-corepoint q allowed

POTENTIAL CONS

Wrt Kelley's method

→ **denser** point to be separated (more time can be needed)

→ **heuristic** separation oracles can lead to weak cuts or loops

→ **fewer** cuts can be generated at the beginning (half of y is q)

→ q needs to be **initialized** (it can be easy for many problems)

Preliminary computational tests

- We wanted to evaluate yoyo search in a **controlled** setting first
- For a given LP problem (e.g. root node relaxation of a MIP)

$$min \{c^T x: A' x \leq b', A'' x = b'', l \leq x \leq u \}$$

  - $P := \{ x: A'' x = b'', l \leq x \leq u \}$
  - *oracle()* stores the list of the constraints in $A' x \leq b'$

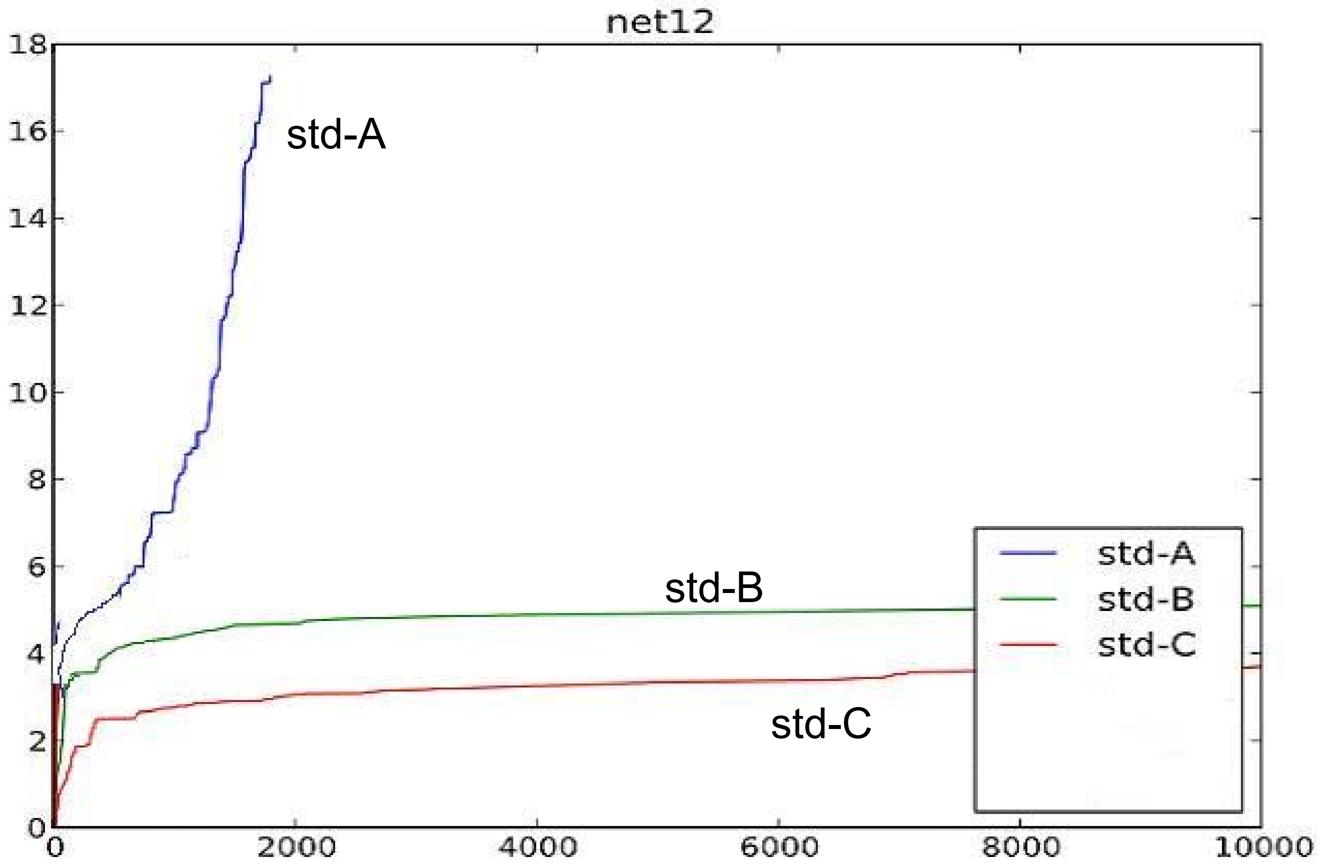
- 3 cut **selection criteria** implemented for the oracle → return:

  A) the **deepest** violated cut in the list (Euclidean distance)

  B) a convex combination of the deepest one and of the (at most) first 10 **violated** or **tight** cuts encountered when scanning the list

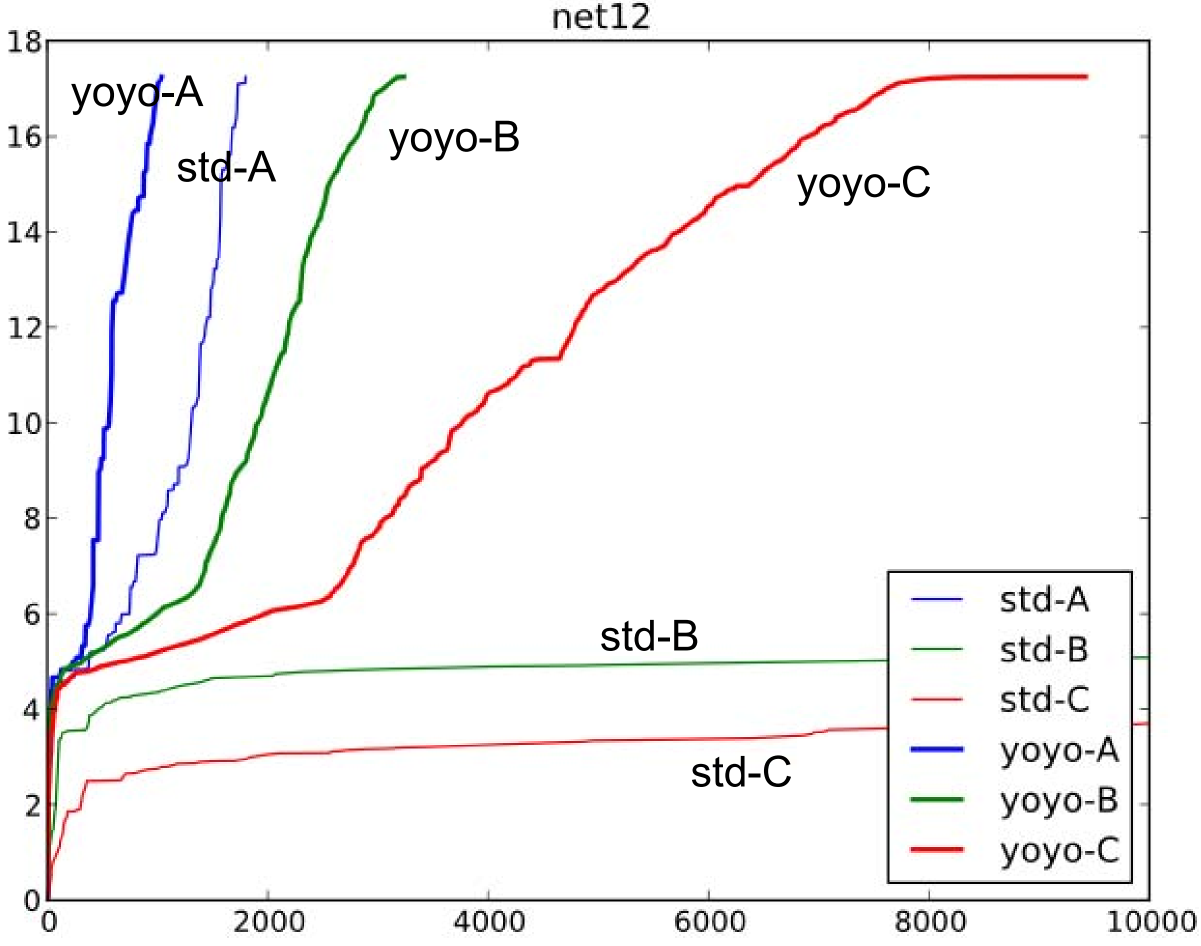
  C) the cut first defined as in case B, and then its rhs is **weakened** so as to **half** the degree of violation

Different scenarios: bound vs iter.s

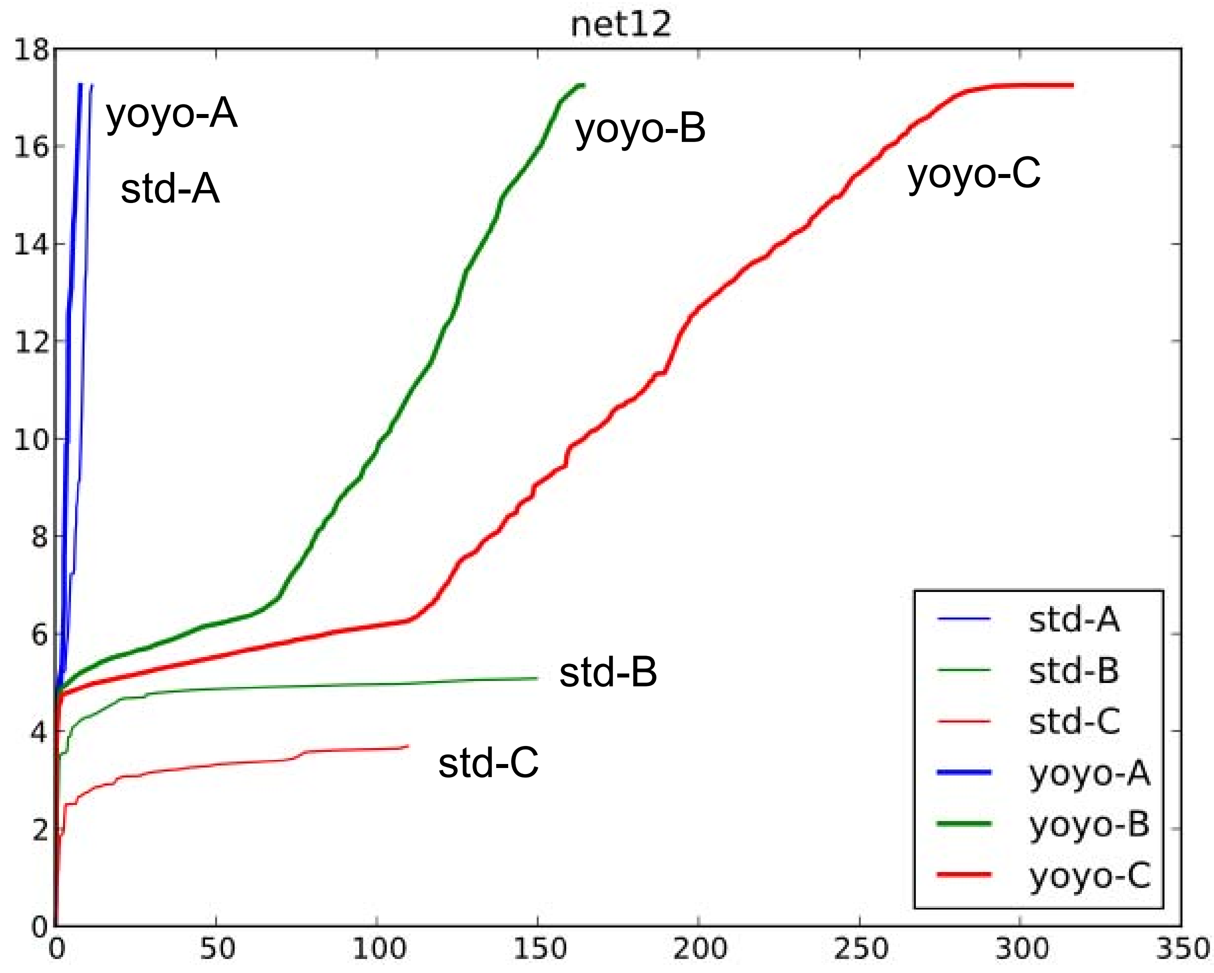


and the winner is …

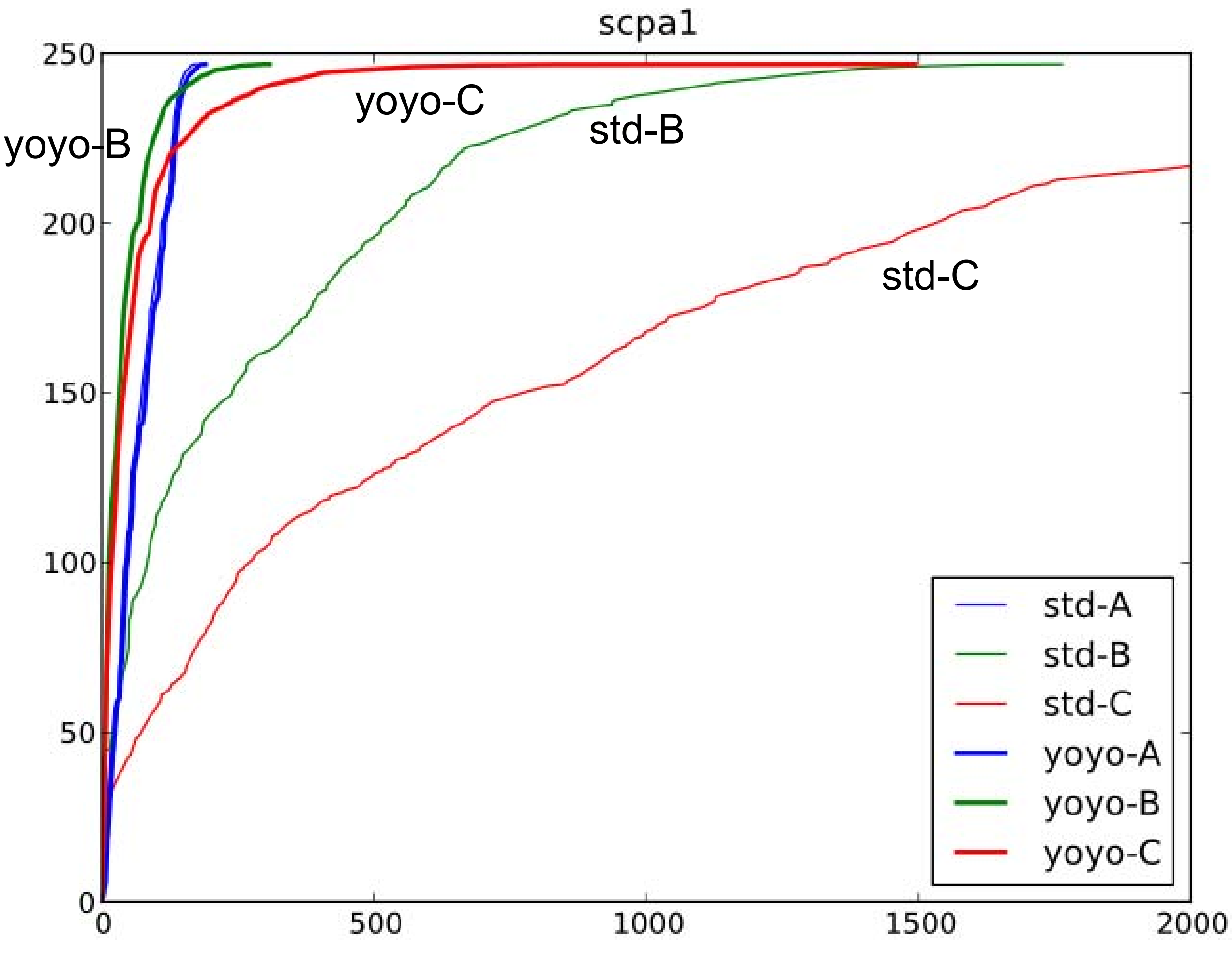
Some plots: bound vs iter.s



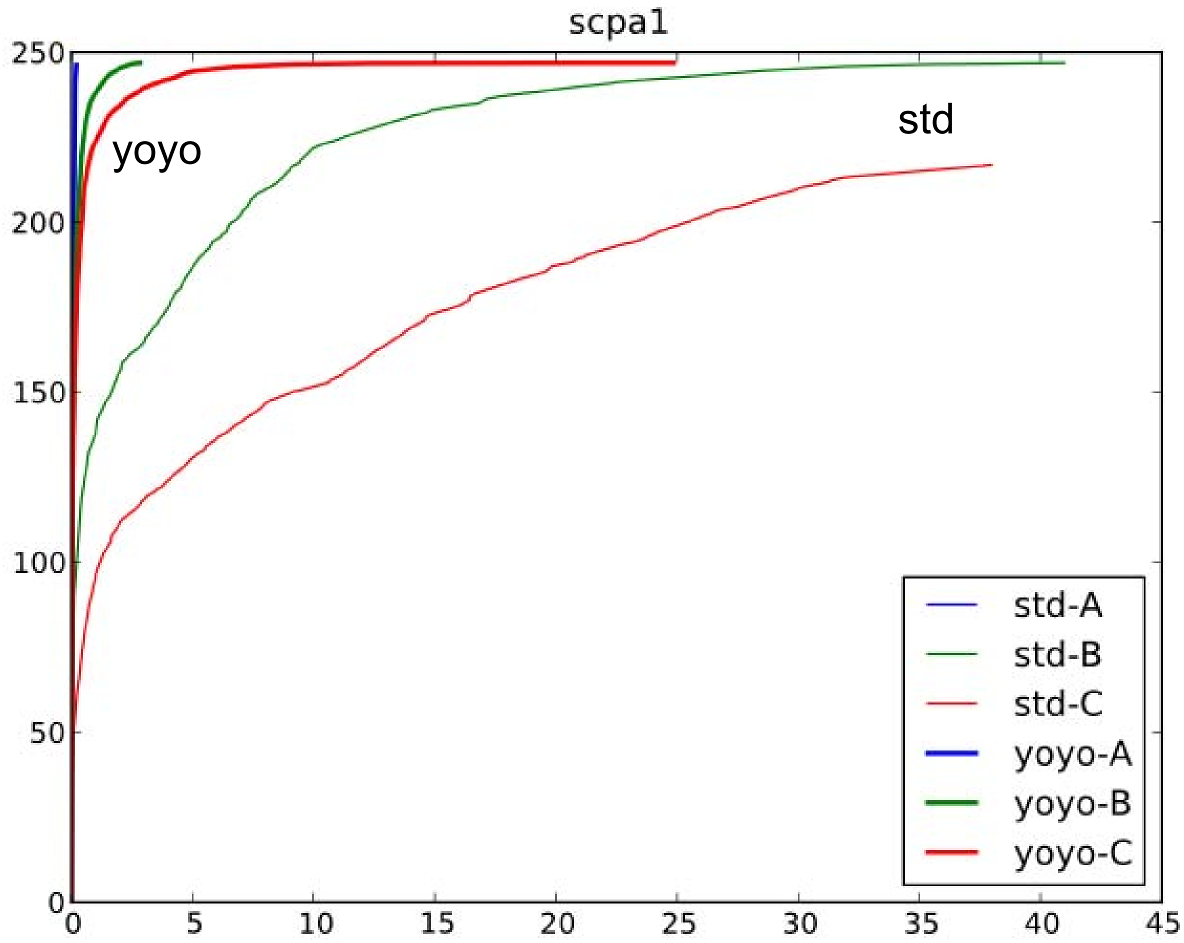
Some plots: bound vs CPU time



Some plots: bound vs iter.s



Some plots: bound vs CPU time



Results on set-covering and MIPLIB

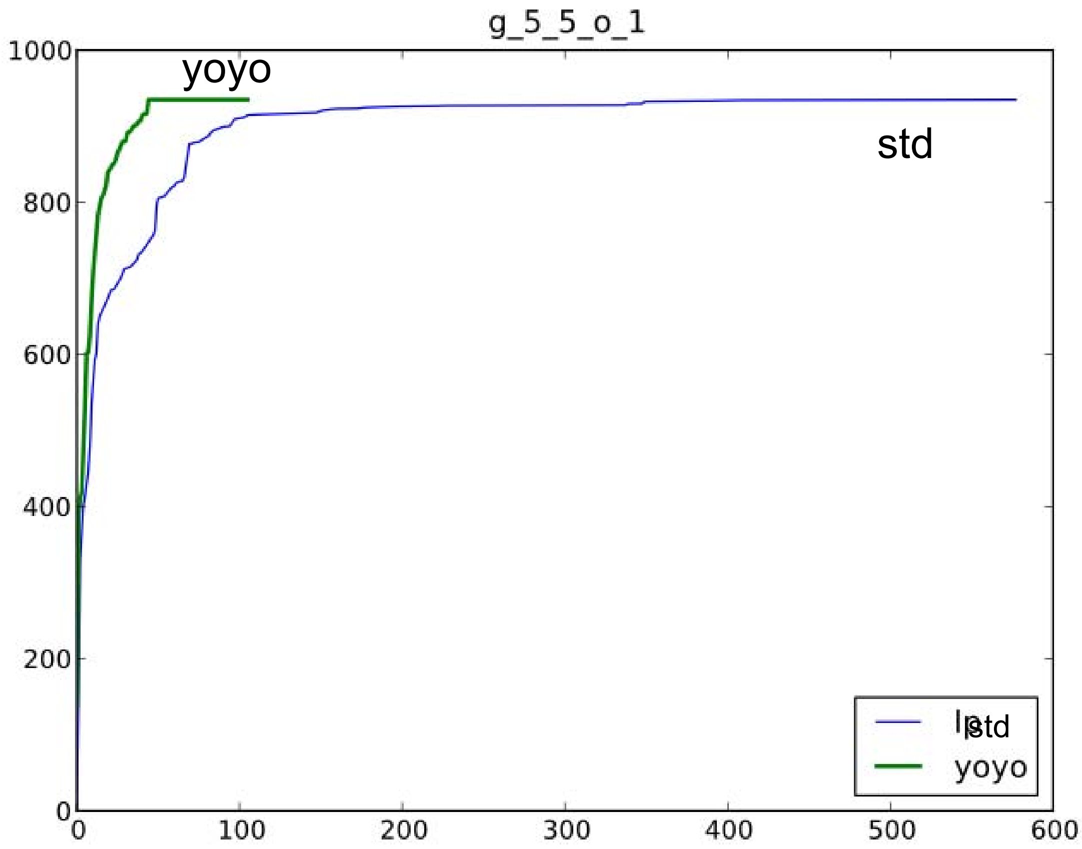
		itr		time		%Cl.Gap	
testbed	scenario	std	yoyo	std	yoyo	std	yoyo
scp	A	360.2	381.8	7.04	7.60	100.0	100.0
	B	6,248.4	871.5	1,558.59	157.85	100.0	100.0
	C	10,000.0	3,471.0	1,936.75	903.18	92.2	100.0
miplib	A	761.2	738.2	5.37	4.76	100.0	100.0
	B	8,720.0	3,442.1	267.59	144.95	46.0	77.7
	C	10,000.0	6,207.2	171.96	221.87	34.7	68.1

More details for shallow cuts

		itr					
problem	method	90%	95%	99%	%Cl.Gap	totTime	totItr
scpclr11	std	3,470	5,473	-	98.8	807.25	10,000
	yoyo	249	316	757	100.0	706.28	4,087
scpclr12	std	7,723	-	-	93.6	946.48	10,000
	yoyo	73	171	740	100.0	1,160.06	3,885
scpclr13	std	7,136	-	-	93.8	2,399.71	10,000
	yoyo	123	187	1,064	100.0	8,872.70	4,804
scpnrg1	std	-	-	-	86.7	2,179.87	10,000
	yoyo	773	1,007	1,613	100.0	1,027.77	4,255
scpnrg2	std	-	-	-	88.7	1,913.55	10,000
	yoyo	749	993	1,511	100.0	766.45	4,138
scpnrg3	std	9,773	-	-	90.4	1,863.02	10,000
	yoyo	631	925	1,536	100.0	983.29	3,931

Table 3: Set covering results under scenario C.

Benders' decomposition



Benders' decomposition

	itr		time		#cuts	
problem	std	yoyo	std	yoyo	std	yoyo
g_5_5_f_1	155	83	24.76	11.01	154	71
g_5_5_f_2	137	75	25.85	14.91	136	63
g_5_5_f_3	131	81	15.97	9.80	130	68
g_5_5_f_4	129	80	15.16	11.21	128	68
.....						
r_25_5_o_3	2041	241	441.66	111.01	4080	447
r_25_5_o_4	1728	196	469.09	77.34	3450	359
r_25_5_o_5	851	124	162.98	50.40	1675	222
geom.mean	314	105	50.74	22.86	441	129

Multicommodity-flow network design problem

Work in progress

- Evaluation of **disjunctive cut** separation based on different cut generation LPs
- Modification of yoyo search for **specific classes of oracles** (including again disjunctive cut separation)
- Integration with **feasibility-pump** like heuristics
- Use of **analytic-center** fast codes (do you have one to lend?)

Lessons learned (to be discussed…)

Separating a **vertex** is often an over-simplified task, that hides the real difficulty of the problem at hand.

Complexity theory implies the following dichotomy for NP-hard problems:

(i) either one cuts only LP vertices and uses the LP tableau to **simplify separation** (thus accepting the unavoidable cut saturation issues),

(ii) or else one uses a more **sophisticated search scheme** with a polynomial number of steps (thus accepting an increased complexity inside the separation oracle).

E.g., for MIPs one can easily read violated intersection cuts from the optimal LP tableau, but these cut cannot be embedded into an efficient search scheme (unless P=NP)

If an **exact black-box separation** procedure is available that works with non-extreme points, the standard search method can be much less efficient than those working with internal points.