Hybrid Constraint Solvers

- An overview
    - Why Hybrid Solvers
    - CP and SAT: Lazy Clause Generation
    - CP and LP: Reification of Linear Constraints
    - Conclusions

Why Hybrid Solvers

Over the years a number of solvers have been developed in various domains, taking advantage of specificities of these domains.

Examples include:

- Linear Programming solvers (**Simplex**)
- Mixed Integer Programming (**MIP**)
- Propositional Satisfiability solvers (**SAT**)
- Answer Set Programming Solvers (**ASP**)
- Satisfiability Modulo Theory Solvers (**SMD**)
- Constraint Programming Solvers (**CP**)
- … Specialised solvers

All of them have their strengths and weaknesses, in different aspects of their functioning.

Why Hybrid Solvers

From a **CP** perspective…

SAT Solvers:

Low expressive power, structure of problem is lost

… but

Good search features (nogoods, non-chronologic backtracking, …)

LP solvers:

Low expressive power – only linear constraints

… but

Good search features (Simplex)

CP Solvers:

Expressive power, flexible, good propagation (global constraints)

… but

not good enough (no nogoods, chronological becktracking, …)

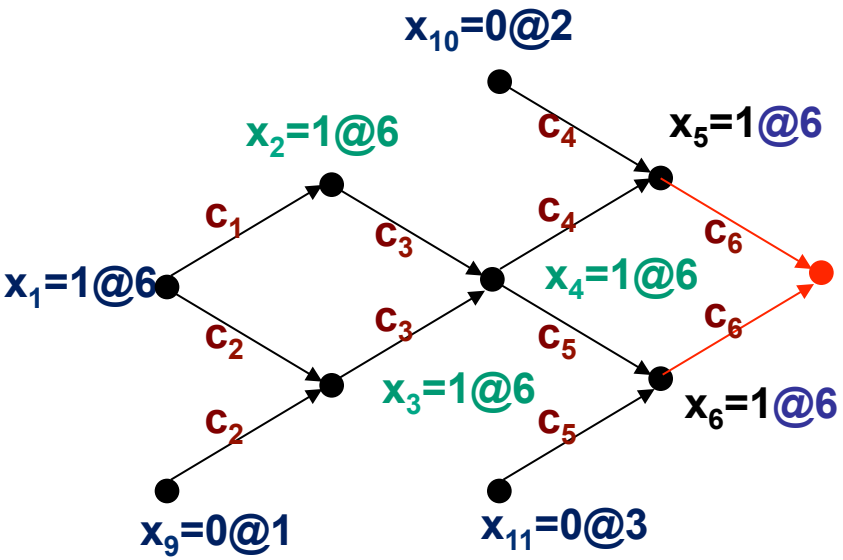
Why Hybrid Solvers

- **Hence … Why not hybridize different solvers taking advantage of their complementary nature** ???
- This is an old idea, with limited results the past (e.g. Benders decomposition). More promising approaches have been developed more recently:
- **Lazy-Clause generation**
  - General purpose hybridisation scheme, where CP propagation is encoded into SAT clauses.
  - Their solving by SAT solvers, improves CP propagation in may ways, e.g. intelligent backtracking (nogoods and non-chronologica backtracking).
- **Reification of Linear Constraints,**
  - General purpose hybridisation scheme, bridging CP variables and constraints with LP solvers;
  - Better detection of failure, and exploitation of intelligent backtracking

Lazy Clause Generation

A brief recall of nogood detection in SAT

$c_1 : (\neg x_1 \vee x_2)$
$c_2 : (\neg x_1 \vee x_3 \vee x_9)$
$c_3 : (\neg x_2 \vee \neg x_3 \vee x_4)$
$c_4 : (\neg x_4 \vee x_5 \vee x_{10})$
$c_5 : (\neg x_4 \vee x_6 \vee x_{11})$
$c_6 : (\neg x_5 \vee \neg x_6)$
$c_7 : (x_1 \vee x_7 \vee \neg x_{12})$
$c_8 : (x_1 \vee x_8)$
$c_9 : (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
...



- Conflict caused by assignments

  x_1=1 and x_9=0 and x_{10}=0 and x_{11} = 0, // x_4=1 and x_{10}=0 and x_{11} = 0,

- No-good clause to prevent repetition of such assignment

  $c_{0a} : (\neg x_1 \vee x_9 \vee x_{10} \vee x_{11})$ // $c_{0b} : (\neg x_4 \vee x_{10} \vee x_{11})$

Lazy Clause Generation

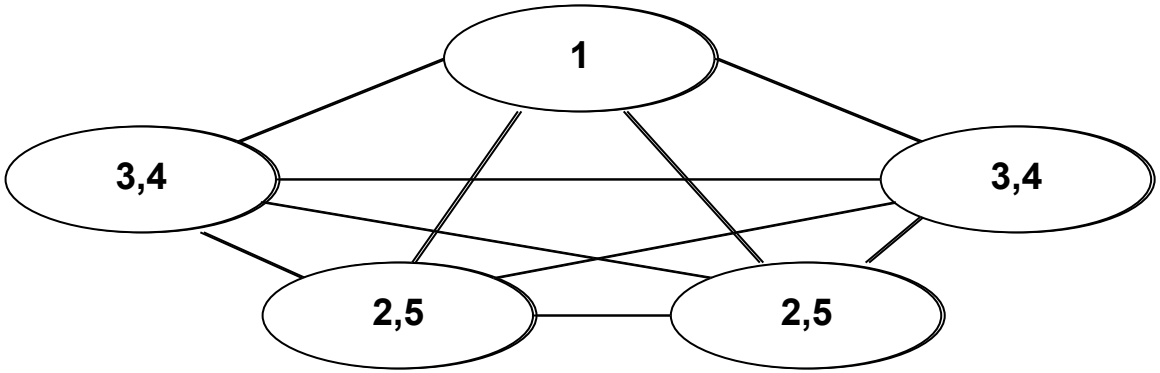
Why is the same not easy in CP?

- CP takes full advantage of specialised algorithms to reason globally with a number of “local” constraints.
- For example, rather than posting inequality constraints whose dealing separately does not propagate adequately,

```
forall(i in 1..n-1, j in i+1..n) cp.post(q[i] ≠ q[j])
```

... post a single **global** constraint

```
cp.post(all_different(q))
```



Lazy Clause Generation

Why is the same not easy in CP?

- But the algorithms that make propagation fast, are very specific and not easy to analyse
- Thus detection of the decisions that caused failure is difficult to do
- Some interesting work on nogoods for CP was done for one or two global constraints, but in general, it does not seem feasible to do so in general.

- **Lazy-clause generation – Basic idea**
  - Decompose global constraints into simple ones;
  - Encode their propagators as clauses
  - Any time a propagation is performed, activate the adequate clauses

Lazy Clause Generation

Example:

- Take constraint `x = y*z` where `x, y` and `z in [-20..20]`

- Two propagators are adopted for `x` (notice variable can be positive or negative)

```
max|min(x) <- max|min(Sx)
```

where

```
Sx = {max(y)*max(z),max(y)*min(z),min(y)*max(z),min(y)*min(z)}
```

- Two propagators are adopted for `y` (and `z` alike)

```
max|min(y) <- if min(x3)*max(x3)>0 then max|min(Sy)
```

where

```
Sy = {max(x)/max(z),max(x)/min(z),min(x)/max(z),min(y)/min(z)}
```

Lazy Clause Generation

- Now these propagators can be triggered several times.

  `x = y*z` where `x, y z in [-20..20]`

- For the above propagators many instances of the propagation rules (e.g. decisions made by the solver) are made during search. For example,

```
y ≥ 2  ∧ z ≥ 3  ->  x ≥ 6
x ≥ 6  ∧ z ≥ 0  ∧ z ≤ 3  ->  y ≥ 2
x ≤ 10 ∧ y ≥ 6  ->  z ≤ 1
x ≤ 10 ∧ y ≥ 9  ->  z ≤ 1   (redundancy!)
y ≥ -1 ∧ y ≤ 1  ∧ z ≥ -1 ∧ z ≤ 1 ->  x ≤ 1
```

- The last example may have happened when z already had its domain restricted to -1..+1, and the domain of y has changed to -1..+1.

- Notice that redundant clauses may be collected during search (see example) !.

Lazy Clause Generation

The propagators may now be represented in clausal form.

- First, all conditions are represented by boolean variables of type (regular encoding – other encodings do not work so well for representing changes in bounds of integer variables)

$$[\![x \leq d]\!] \quad \text{and} \quad [\![x = d]\!].$$

- Second, all recorded propagations are transformed into clauses.

$$x \leq 10 \wedge y \geq 6 \rightarrow z \leq 1$$

  is transformed in a clause

$$x > 10 \vee y < 6 \vee z \leq 1$$

  with the appropriate literals

$$\neg [\![x \leq 10]\!] \vee [\![y \leq 5]\!] \vee [\![z \leq 1]\!]$$

Lazy Clause Generation

- Of course other clauses must be added such that contradictions can be detected, namely to model variables taking values within ranges (bounds consistency).
- For example, variable x taking values in the range `a .. b` is represented by a number of clauses, namely

$$(1)\ [\![x \le d]\!] \rightarrow [\![x \le d+1]\!]$$

$$\neg [\![x \le d]\!] \vee [\![x \le d+1]\!] \qquad a \le d \le b-1$$

$$\text{and } (2)\ [\![x = d]\!] \leftrightarrow [\![x \le d]\!] \wedge \neg[\![x \le d-1]\!]$$

$$\neg[\![x = d]\!] \vee [\![x \le d]\!] \qquad a \le d \le b$$

$$\neg[\![x = d]\!] \vee \neg[\![x \le d-1]\!] \qquad a < d \le b$$

$$[\![x = d]\!] \vee \neg[\![x \le d]\!] \vee [\![x \le d-1]\!] \qquad a < d < b$$

$$[\![x = a]\!] \vee \neg[\![x \le a]\!]$$

$$[\![x = b]\!] \vee \neg[\![x \le b-1]\!]$$

- The regular encoding requires 2n Boolean variables and 4n clauses to code a domain of size n (the direct encoding, only $[\![x = v]\!]$ variables) would require n2 clauses).
- Different clauses are used for other types of domains (e.g. with holes)!

Lazy Clause Generation

- With **all** these encodings, it may be proved that a SAT solver using Unit Propagation on these clauses would obtain at least the same pruning than the corresponding propagators in the CP solver.

- In fact, the SAT solver may even obtain stonger prunnings. For example, propagation rule

  $$y \geq -1 \wedge y \leq 1 \wedge z \geq -1 \wedge z \leq 1 \rightarrow x \leq 1$$

  is encoded as

  $$[\![y \leq -2]\!] \vee \neg[\![y \leq 1]\!] \vee [\![z \leq -2]\!] \vee \neg[\![z \leq 1]\!] \vee [\![x \leq 1]\!]$$

  However, from

  `x in 2 .. 20`, `y in -1..1` and `z in -1..20`

  the CP propagator would not infer `z` to be in range `2..20` (the propagator does not prune x when either y or z include 0 in their domain.

Lazy Clause Generation

- Hence, **why not encoding the whole process in clausal form and use a SAT solver, rather than a CP solver**?

- The problem lies in the sheer size of the resulting encodeing. In addition to the very large, yet acceptable, size of the domains, the eager encoding of all the instances of all the propagators is in general impossible.
- For example, the encoding of all the possible instances of the propagators of constraint `x = y*z` , with domains -20 .. 20 for all variables requires more than 100 000 clauses!
- This is clearly beyond even the most powreful SAT solvers. Hence, the notion of

Lazy Clause Generation:

- Rather than eagerly generate all the clauses corresponding to domains and propagation rules, only the instances of the propagators that are generated by a CP solver are generated!

Lazy Clause Generation

- The rational for this is clear – for every constraint, only a few instances of their propagators are effectively generated during search.

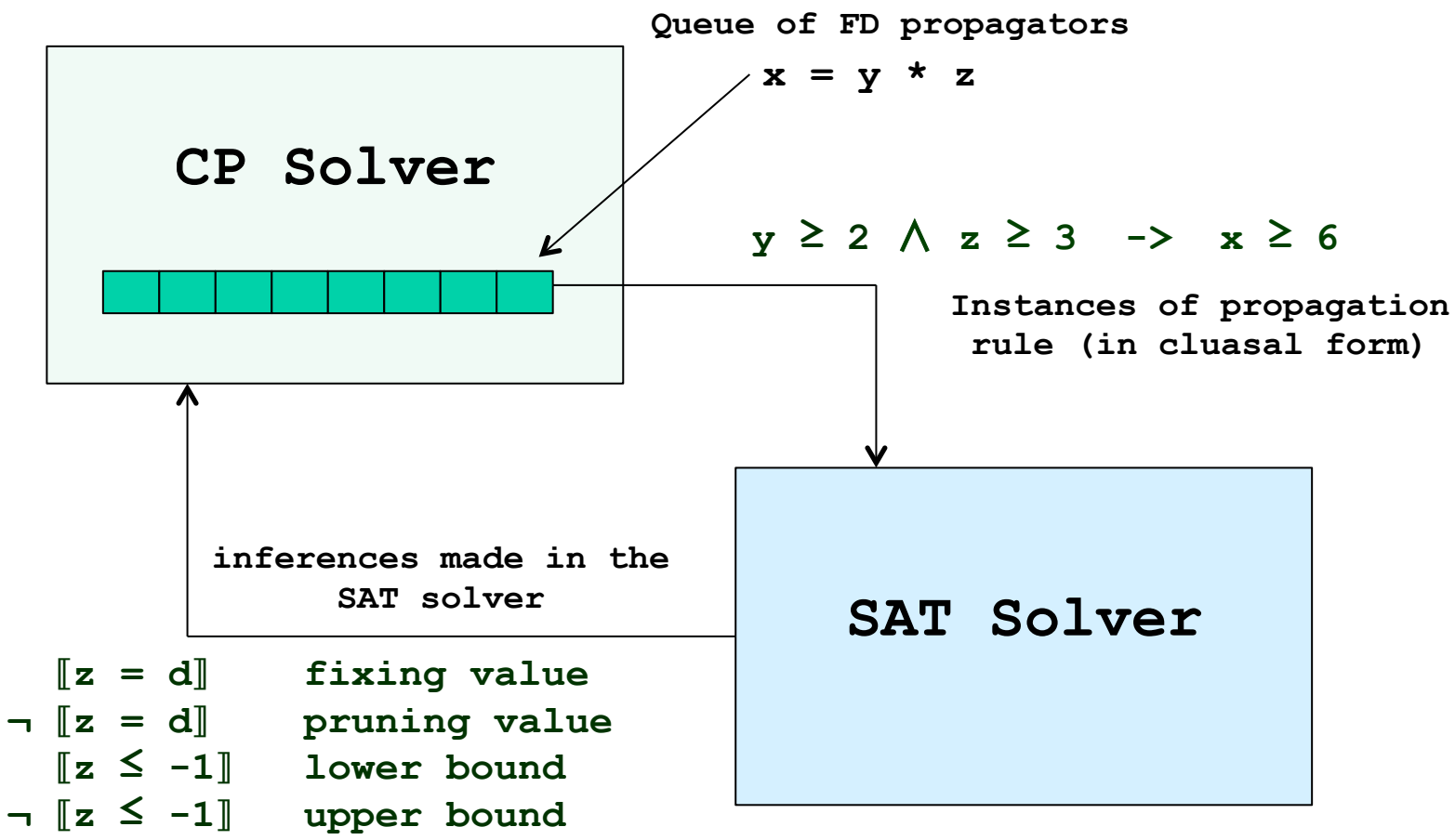
- For example for the constraint **`x = y * z`** with domain **`-20 .. 20`**, it is possible that some constraint prunes x to range **`4..20`**, and hence the values of **`x=-19, -18, ... , 3`** are never considered in the lazy clause generator.

- Nevertheles, this is not always the case. For constraints **`x > y`** and **`y < x,`** failure detection requires that the bounds of x and y to take almost every value in their domain!

- Nevertheless, a hybridisation between a CP and a SAT solver has been peoposed and proved quite competitive. The main idea is that a CP solver, in addition to its working, also passes to a Sat solver the intancs of propagation rules it iuses. In addition to its own pruning, the CP solver also receives pruning information from the SAT solver.

Lazy Clause Generation

- Simplistically,we may consider the following schema



Lazy Clause Generation

- This architecture, with many variants have been proposed and tested by Peter Stuckey group (Melbourne University) and is currently quite competitive.
- In fact, it has won CP solver competitions with top CP solvers (e.g. Gecode) in recent **ZINC competitions**.
- Unfortunately, the CP community is still divided with respect to the competition that should become a standard.
- Unlike SAT, there are many different constraints (e.g. Global) which are not implemented in most solvers, which means that competitions can only test the simpler constraints.

Google "**Lazy Clause Generation**" and find several papers in this area with more details, and experimental results, as well as extensions (e.g. symmetry breaking).

Reification of Linear Constraints

- This different type of Hybridization, that we are currently working on, was motivated by model-checking applications (namely software verification).
- First we recall a major advantage of CP – the handling of **reification**. Basically this means that you may not only **post** constraints, but also check whether they have already been entailed.
- This property only works for a limited set of constraints, named reified constraints, since the entailment of many constraints is impossible to detect efficiently.
- For example, the entailment of a Fermat constraint (given variables withing a certain range, `a,b,c in 1..100`, and `n in 3..5` check whether $a^n+b^n = c^n$.) can only be done by eumeration or very specialised number theory results.
- However, this is not the case for many simpler constraints, namely linear (or even quadratic constraints), at least in an approximate form.
- For example, the entailment of a linear constraint such as `a + b > c` is detected as soon as the maximum value of the domain of `c` is gretaer than the sum of the maximum values of the domains of `a` and `b` (as well as other conditions).

Reification of Linear Constraints

- These reified constraints are important in a number of applications, namely in scheduling.
- If two tasks should not overlap, we can state this disjunctive constraint as

$$s1+d1 \leq s2 \quad \vee \quad s2 + d2 \leq s1$$

- which are implemented as posting constraints

$$i) \quad b1 \Leftrightarrow s1+d1 \leq s2$$

$$ii) \quad b1 \Leftrightarrow s1+d1 \leq s2,$$

$$iii) \quad b1+b2 \geq 1$$

  the first two associating Boolean (0/1) variables to the disjuncts and the thrird effectively impose disjunction.

- As such we should not commit earlier to an ordering of the tasks, before checking whether these orderings are acceptble, since this would require, for n non-overlapping tasks to check n! potential orderings.

Reification of Linear Constraints

- These reified constraints are important in a number of applications, namely in scheduling.
- If two tasks should not overlap, we can state this disjunctive constraint as

$$s1+d1 \leq s2 \quad \vee \quad s2 + d2 \leq s1$$

- which are implemented as posting constraints

$$i) \quad b1 \Leftrightarrow s1+d1 \leq s2$$

$$ii) \quad b1 \Leftrightarrow s1+d1 \leq s2,$$

$$iii) \quad b1+b2 \geq 1$$

  the first two associating Boolean (0/1) variables to the disjuncts and the thrird effectively impose disjunction.

- As such we should not commit earlier to an ordering of the tasks, before checking whether these orderings are acceptble, since this would require, for n non-overlapping tasks to check n! potential orderings.

Reification of Linear Constraints

- Of course, this reification requires that entailment is efficient. For example, let us consider the following two constraints

$$a+b \geq c$$

$$a+b \leq d$$

to be satisfied simultaneously. If at a certain point we know that

$$c > k \quad \text{and} \quad d < k,$$

the contradition would be found, although in time quadratic to the domains of the variables a, b and c, which is not convenient.

-However, this does not need to be so. In elementary algebra we would fisrt deduce

$$a + b > k$$

$$-a - b > -k.$$

-Now adding both inequations we get $0 < 0$, which proves inconsistency.

- Therefore it would be convenient to obtain reified constraint with linear expressions, namely in **(bounded) model checking** software verification.

Reification of Linear Constraints

- In (bounded) model checking software verification, programs and pre- and (negation of) post conditions are modeled as constraints. The existance of a bug implied that the set of costraints are satisfiable.

$$\texttt{Pre} \cup \texttt{Code} \models \texttt{Post} \quad \text{or} \quad \texttt{Pre} \cup \texttt{Code} \cup \neg\texttt{Post} \models \bot$$

- In this technique, an imperative if statement such as

```
if (A > B)
   {X = Y+1;}
else
   {X = Y-1;}
```

would be modeled as two implication constraints

$$(A > B) \Rightarrow (X = Y+1)$$
$$(A \leq B) \Rightarrow (X = Y-1)$$

Reification of Linear Constraints

- And yet, these two constraints propagate very slowly if we know later that X ≥ Y. Why?

- In fact, many constraint solvers (e.g. SICStus) do not reify linear constraints such as **`X-Y #> Z`**, and thus are not able to detect, **before enumeration**, inconsistency in constraints

  `X-Y #> Z` and **`X-Z #< Y`**

- (although they detect it in simpler constraints **`X #> Y`** and **`Y #< Z)`**.

- Hence a first step towards implementing the intended model was to explicitely associate a Boolean with each linear constraint, and make enumeration be made primarily at the booleam variables.
- Hopefully, some of the reified expressions (the simpler) would eventually deduce relevant information able to trigger the Booleans.
- Otherwise, the checking would require the enumeration of integer variables with very large domains (at least 2^{16}).

Reification of Linear Constraints

- The process can be significantly speeded up, if rather than considering integer values, we consider real domains for the integer variables.

- More specifically, we consider redundant constraints, by bridging integer variables into real variables and duplicating the linear constraints over the integer/linear variables.

- Still we rely on the enumeration of the Boolean variables, but now propagation (and detection of inconsistencies) can be performed much faster.

- This is because we can take advantage of a Simplex-like algorithm to transform the set of constraints, by algebraic operations alone, into a **Solved Form**. If the set of constraints cannot be placed in such a solved form, then they are inconsistent!

Reification of Linear Constraints

- We can illustrate this algorithm with the set of constraints (where X1 and X2 should be non-negative)

$$b1 \Leftrightarrow -x_1 + 3x_2 \leq 9$$
$$b2 \Leftrightarrow x_1 + x_2 \leq 11$$
$$b3 \Leftrightarrow 2x_1 + x_2 \leq 18$$
$$b4 \Leftrightarrow x_1 + 2x_2 \geq 12$$

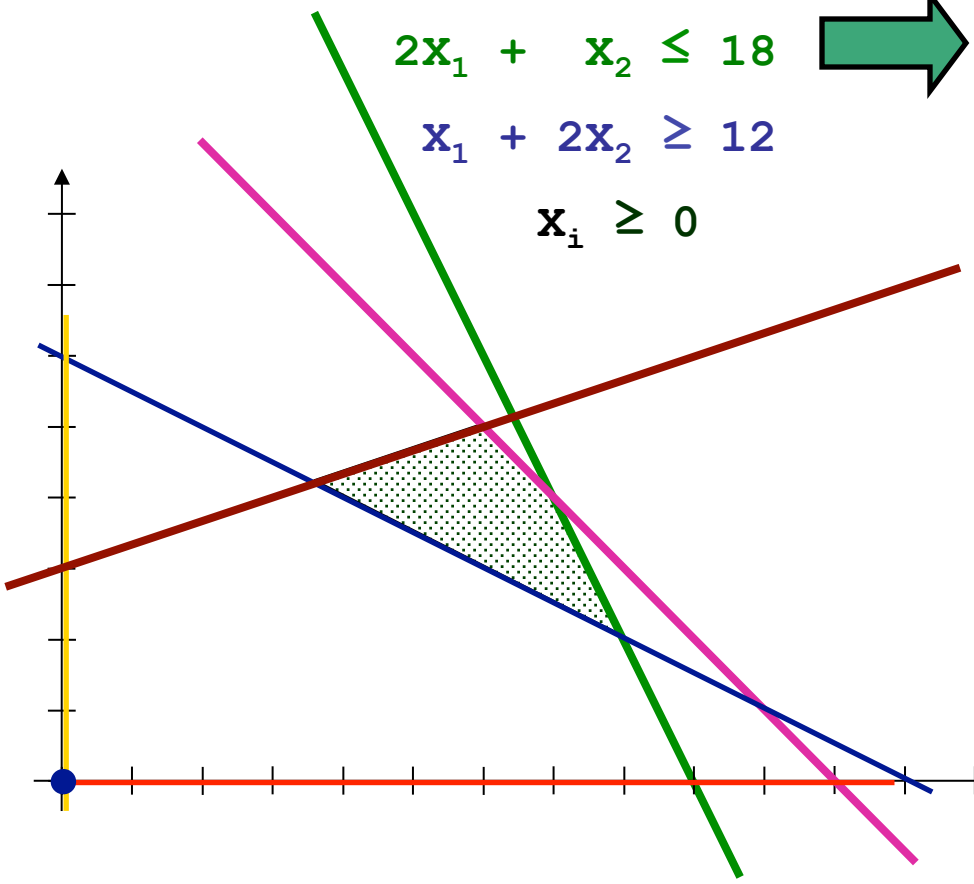
- First we convert them into equality constraints by addition of slack, non-negative variables, obtaining

$$b3 \Leftrightarrow -X1 + 3X2 + X3 = 9$$
$$b4 \Leftrightarrow X1 + X2 + X4 = 11$$
$$b5 \Leftrightarrow 2X1 + X2 + X5 = 18$$
$$b6 \Leftrightarrow X1 + 2X2 + X6 = 12$$

- Then we proceed with a number of “pivotal” transformations, rewriting the equations upon switching basic and non-basic variables.

Reification of Linear Constraints

$$-x_1 + 3x_2 \leq 9$$
$$x_1 + x_2 \leq 11$$
$$2x_1 + x_2 \leq 18$$
$$x_1 + 2x_2 \geq 12$$
$$x_i \geq 0$$

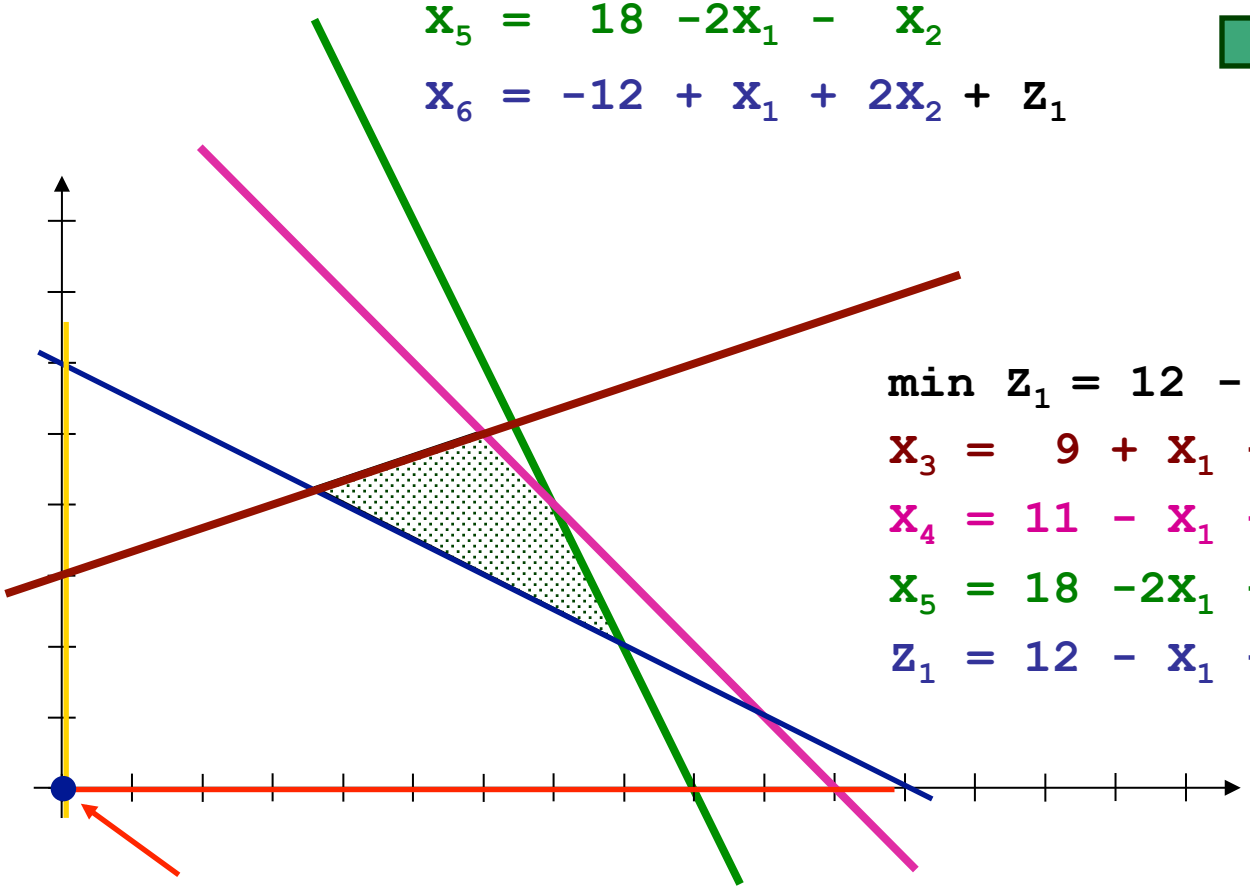


$$-x_1 + 3x_2 + x_3 = 9$$
$$x_1 + x_2 + x_4 = 11$$
$$2x_1 + x_2 + x_5 = 18$$
$$x_1 + 2x_2 - x_6 = 12$$

$$x_3 = 9 + x_1 - 3x_2$$
$$x_4 = 11 - x_1 - x_2$$
$$x_5 = 18 - 2x_1 - x_2$$
$$x_6 = -12 + x_1 + 2x_2$$

Reification of Linear Constraints

$$x_3 = 9 + x_1 - 3x_2$$
$$x_4 = 11 - x_1 - x_2$$
$$x_5 = 18 - 2x_1 - x_2$$
$$x_6 = -12 + x_1 + 2x_2 + z_1$$



$$\min z_1 = 12 - x_1 - 2x_2 + x_6$$
$$x_3 = 9 + x_1 - 3x_2$$
$$x_4 = 11 - x_1 - x_2$$
$$x_5 = 18 - 2x_1 - x_2$$
$$z_1 = 12 - x_1 - 2x_2 + x_6$$

Reification of Linear Constraints

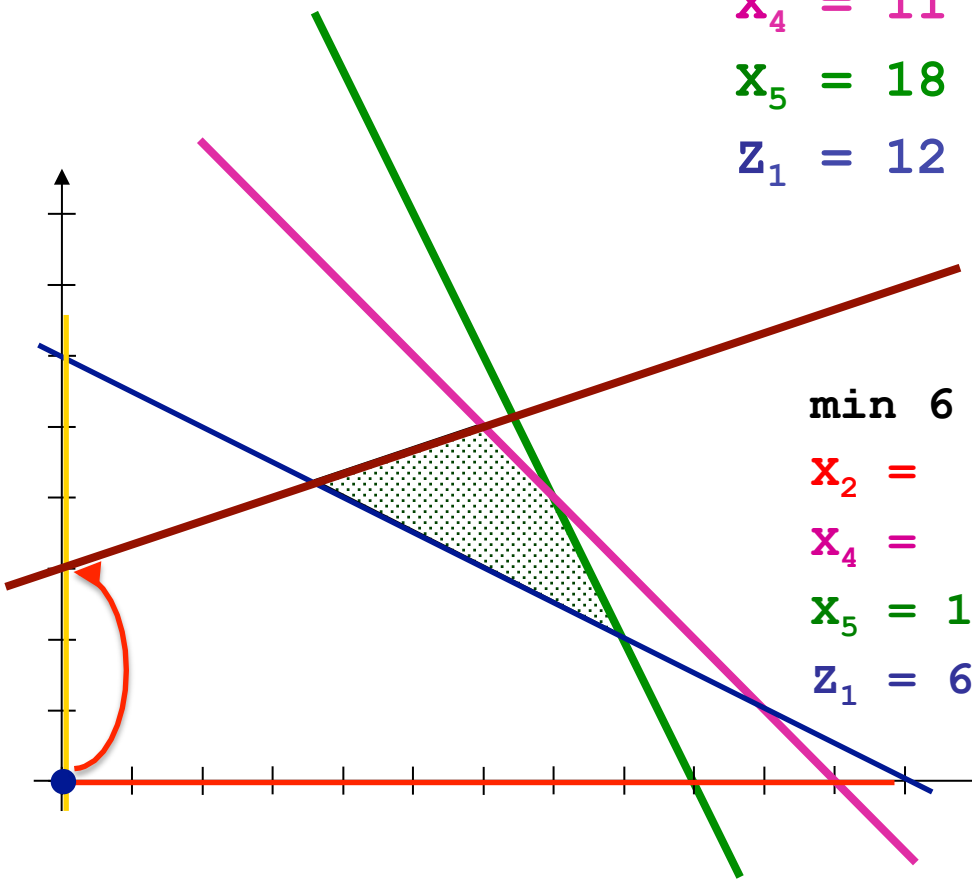
$$\min Z_1 = 12 - X_1 - 2X_2 + X_6$$

$$X_3 = 9 + X_1 - 3X_2 \quad (9/3)$$

$$X_4 = 11 - X_1 - X_2 \quad (11/1)$$

$$X_5 = 18 - 2X_1 - X_2 \quad (18/1)$$

$$Z_1 = 12 - X_1 - 2X_2 + X_6 \quad (12/2)$$



$$\min 6 - 7X_1/3 - 2X_3/3 + X_6$$

$$X_2 = 3 + X_1/3 - X_3/3$$

$$X_4 = 8 - 4X_1/3 + X_3/3$$

$$X_5 = 15 - 7X_1/3 + X_3/3$$

$$Z_1 = 6 - 5X_1/3 + 2X_3/3 + X_6$$

Reification of Linear Constraints

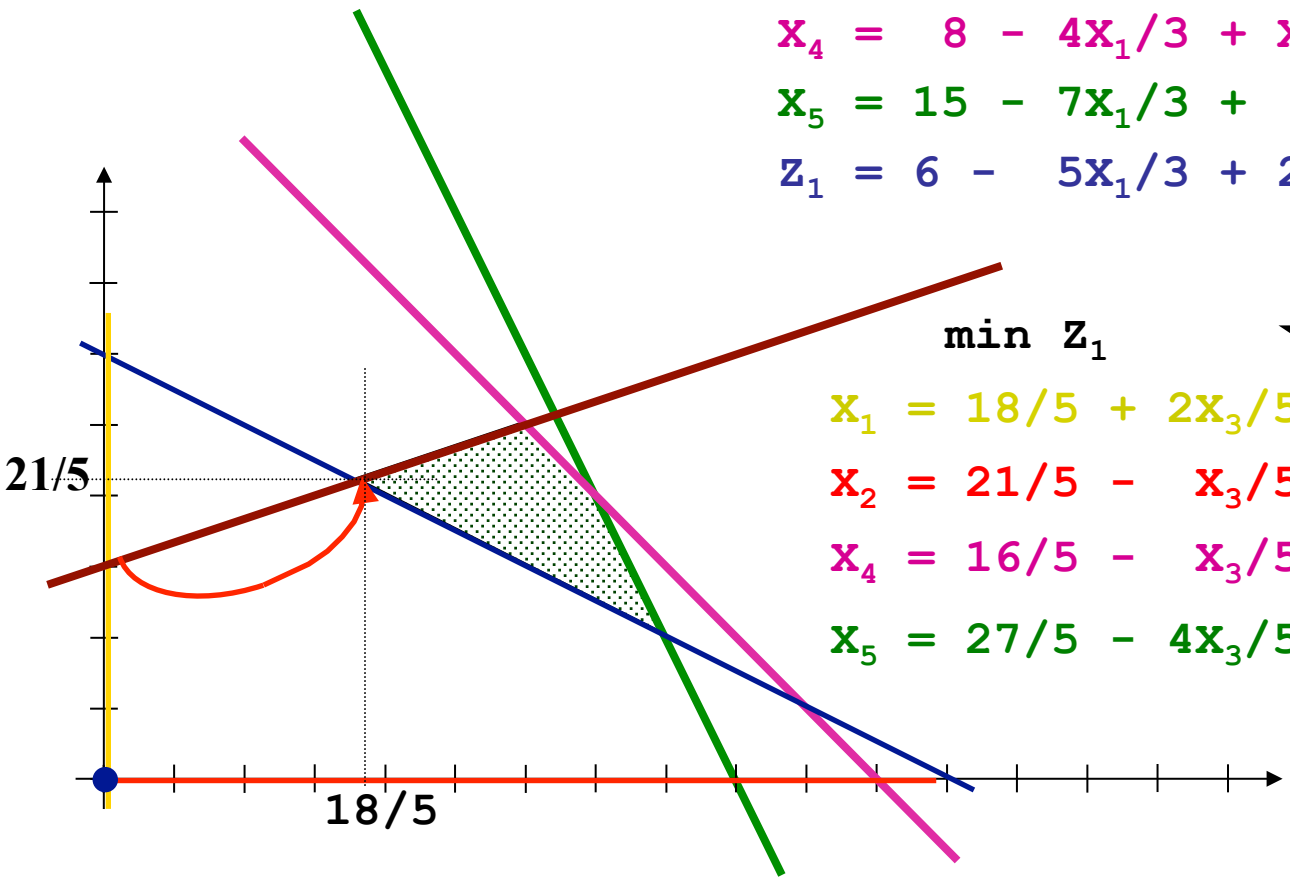
$$\min 6 - 7X_1/3 - 2X_3/3 + X_6$$

$$X_2 = 3 + X_1/3 - X_3/3$$

$$X_4 = 8 - 4X_1/3 + X_3/3 \qquad (6)$$

$$X_5 = 15 - 7X_1/3 + X_3/3 \qquad (45/7)$$

$$Z_1 = 6 - 5X_1/3 + 2X_3/3 + X_6 \qquad (18/5)$$



$$\min Z_1$$

$$X_1 = 18/5 + 2X_3/5 + 3X_6/5 - 3Z_1/5$$

$$X_2 = 21/5 - X_3/5 + X_6/5 - Z_1/5$$

$$X_4 = 16/5 - X_3/5 - 4X_6/5 + 4Z_1/5$$

$$X_5 = 27/5 - 4X_3/5 - 6X_6/5 + 7Z_1/5$$

Reification of Linear Constraints

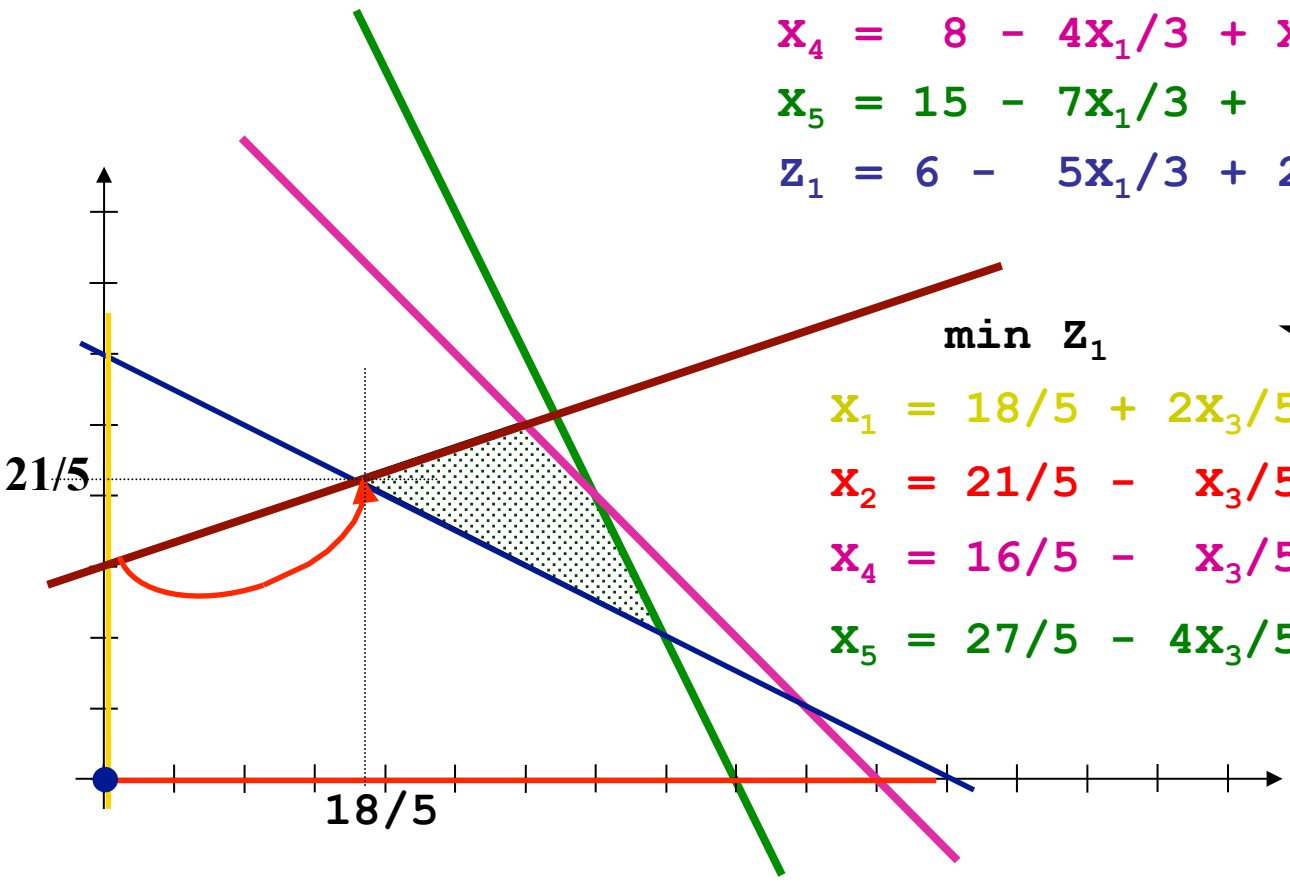
$$\min\ 6 - 7X_1/3 - 2X_3/3 + X_6$$

$$X_2 = 3 + X_1/3 - X_3/3$$

$$X_4 = 8 - 4X_1/3 + X_3/3 \qquad (6)$$

$$X_5 = 15 - 7X_1/3 + X_3/3 \qquad (45/7)$$

$$Z_1 = 6 - 5X_1/3 + 2X_3/3 + X_6 \qquad (18/5)$$



$$\min\ Z_1$$

$$X_1 = 18/5 + 2X_3/5 + 3X_6/5 - 3Z_1/5$$

$$X_2 = 21/5 - X_3/5 + X_6/5 - Z_1/5$$

$$X_4 = 16/5 - X_3/5 - 4X_6/5 + 4Z_1/5$$

$$X_5 = 27/5 - 4X_3/5 - 6X_6/5 + 7Z_1/5$$

Reification of Linear Constraints

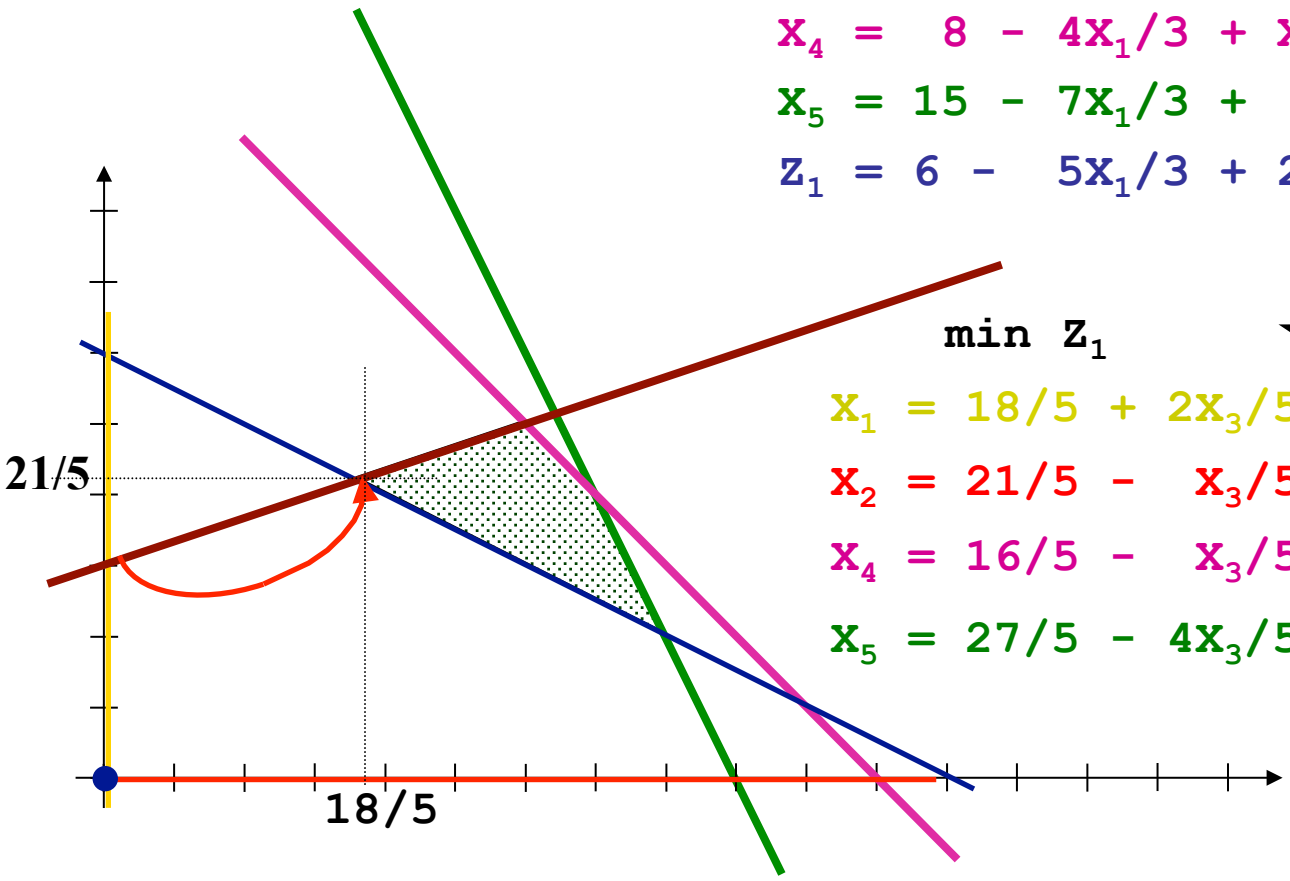
$$\min\ 6 - 7X_1/3 - 2X_3/3 + X_6$$

$$X_2 = 3 + X_1/3 - X_3/3$$

$$X_4 = 8 - 4X_1/3 + X_3/3 \qquad (6)$$

$$X_5 = 15 - 7X_1/3 + X_3/3 \qquad (45/7)$$

$$Z_1 = 6 - 5X_1/3 + 2X_3/3 + X_6 \qquad (18/5)$$



$$\min\ Z_1$$

$$X_1 = 18/5 + 2X_3/5 + 3X_6/5 - 3Z_1/5$$

$$X_2 = 21/5 - X_3/5 + X_6/5 - Z_1/5$$

$$X_4 = 16/5 - X_3/5 - 4X_6/5 + 4Z_1/5$$

$$X_5 = 27/5 - 4X_3/5 - 6X_6/5 + 7Z_1/5$$

Reification of Linear Constraints

- Since it was possible to transform this system into solved form SF0 (a set of equations with non-negative free coeficients), then the system is feasible.
- A possible solution is obtained by zero-ing the non-basic variables X3 and X6, thus obtaining solution

  $X_1 = 18/5$ $X_2 = 21/5$ $X_4 = 16/5$ $X_5 = 27/5$

  and, as mentioned, $X_4 = X_6 = 0$
- The artificial variable, once reaching the value 0, can simply be ignored, as the system with $Z_1 = 0$ is equivalent to the initial system (where Z_1 did not occur).

$$x_1 = 18/5 + 2x_3/5 + 3x_6/5 - 3z_1/5$$
$$x_2 = 21/5 - x_3/5 + x_6/5 - z_1/5$$
$$x_4 = 16/5 - x_3/5 - 4x_6/5 + 4z_1/5$$
$$x_5 = 27/5 - 4x_3/5 - 6x_6/5 + 7z_1/5$$

Reification of Linear Constraints

- This scheme not only allows a quick detection of satisfiability, but also allows the detection of nogoods, which are a usually a small subset of the constraints.

- In theory, finding the set of all minimum IIS (inconsistent Irreducible Sets) is an NP problem.

- However, finding one such set is trivial (in this context), if we assign to the slack variables the role of witnesses.

- Still with an example, similar to the previous one

$$b3 \Leftrightarrow -x_1 + 3x_2 + x_3 = 9 \quad \ldots -x_1 + 3x_2 \leq 9 \quad \ldots$$

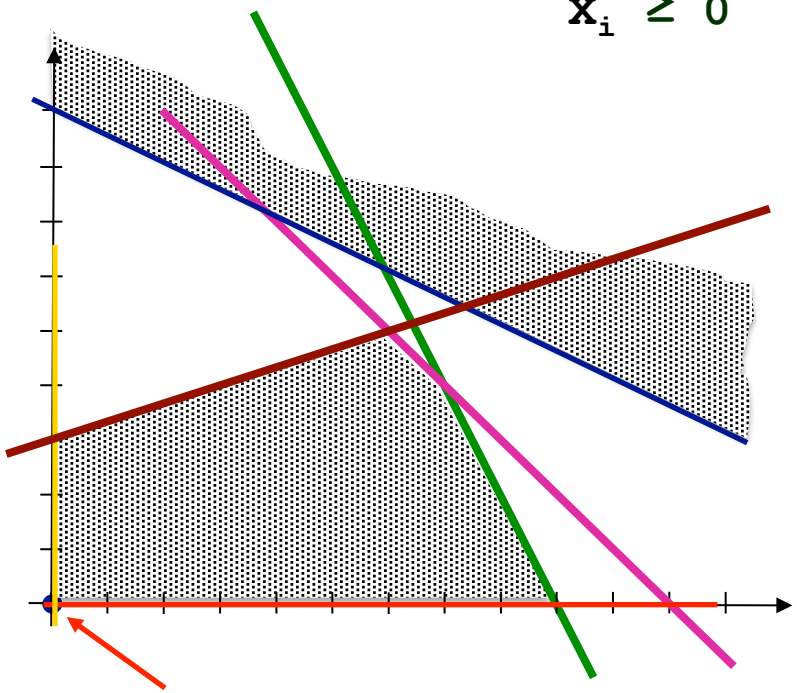
$$b4 \Leftrightarrow x_1 + x_2 + x_4 = 11 \quad \ldots x_1 + x_2 \leq 11$$

$$b5 \Leftrightarrow 2x_1 + x_2 + x_5 = 18 \quad \ldots 2x_1 + x_2 \leq 18$$

$$b6 \Leftrightarrow x_1 + 2x_2 + x_6 = 18 \quad \ldots x_1 + 2x_2 \geq 18$$

Reification of Linear Constraints

$$-x_1 + 3x_2 \leq 9$$
$$x_1 + x_2 \leq 11$$
$$2x_1 + x_2 \leq 18$$
$$x_1 + 2x_2 \geq 18$$
$$x_i \geq 0$$

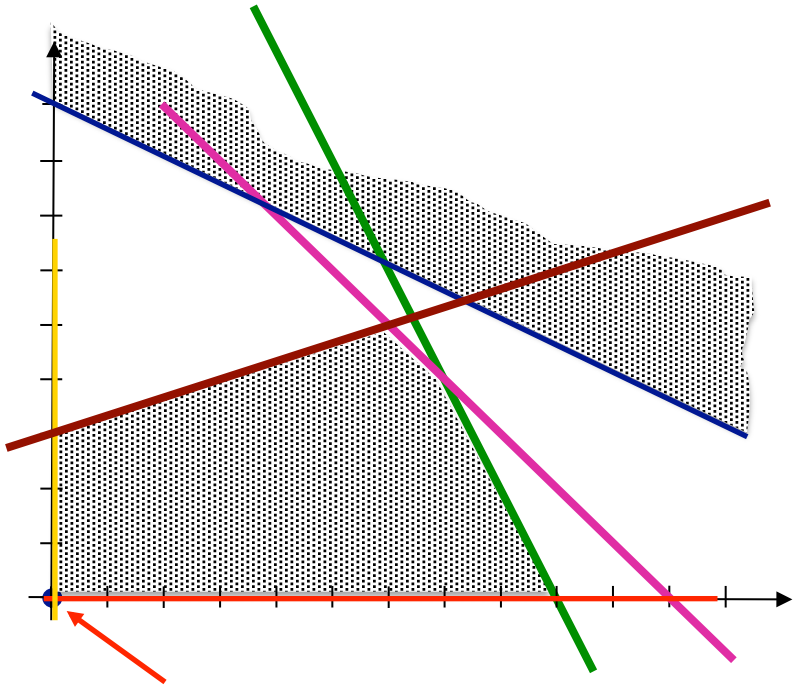


$$-x_1 + 3x_2 + x_3 = 9$$
$$x_1 + x_2 + x_4 = 11$$
$$2x_1 + x_2 + x_5 = 18$$
$$x_1 + 2x_2 - x_6 = 18$$

$$x_3 = 9 + x_1 - 3x_2$$
$$x_4 = 11 - x_1 - x_2$$
$$x_5 = 18 - 2x_1 - x_2$$
$$x_6 = -18 + x_1 + 2x_2$$

Reification of Linear Constraints

$$x_3 = 9 + x_1 - 3x_2$$
$$x_4 = 11 - x_1 - x_2$$
$$x_5 = 18 - 2x_1 - x_2$$
$$x_6 = -18 + x_1 + 2x_2 - z_1$$



$$\min z_1 = 18 - x_1 - 2x_2 + x_6$$
$$x_3 = 9 + x_1 - 3x_2$$
$$x_4 = 11 - x_1 - x_2$$
$$x_5 = 18 - 2x_1 - x_2$$
$$z_1 = 18 - x_1 - 2x_2 + x_6$$

Reification of Linear Constraints

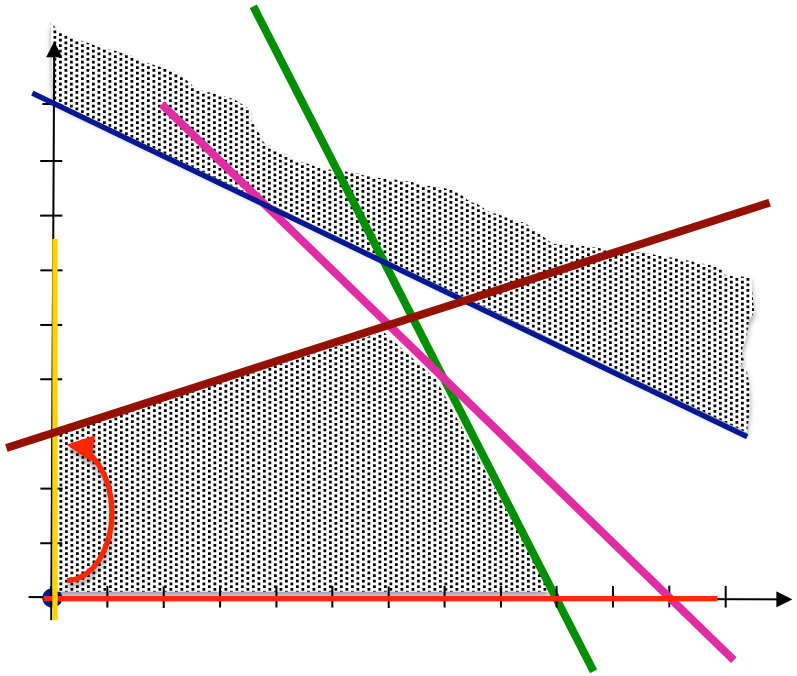
$$\min Z_1 = 18 - X_1 - 2X_2 + X_6$$

$$X_3 = 9 + X_1 - 3X_2 \quad (9/3)$$

$$X_4 = 11 - X_1 - X_2 \quad (11/1)$$

$$X_5 = 18 - 2X_1 - X_2 \quad (18/1)$$

$$Z_1 = 18 - X_1 - 2X_2 + X_6 \quad (18/2)$$



$$\min Z_1 = 9 - 5X_1/3 + 2X_3/3 + X_6$$

$$X_2 = 3 + X_1/3 - X_3/3$$

$$X_4 = 8 - 4X_1/3 + X_3/3$$

$$X_5 = 15 - 7X_1/3 + X_3/3$$

$$Z_1 = 12 - 5X_1/3 + 2X_3/3 + X_6$$

Reification of Linear Constraints

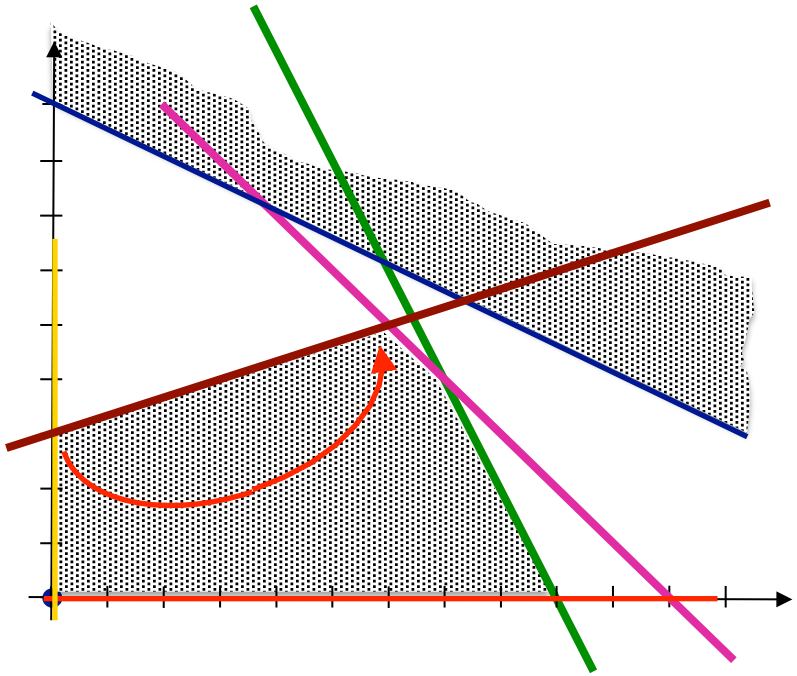
$$\min Z_1 = 9 - 5X_1/3 + 2X_3/3 + X_6$$

$$X_2 = 3 + X_1/3 - X_3/3 \quad (9)$$

$$X_4 = 8 - 4X_1/3 + X_3/3 \quad (2)$$

$$X_5 = 15 - 7X_1/3 + X_3/3 \quad (45/7)$$

$$Z_1 = 12 - 5X_1/3 + 2X_3/3 + X_6 \quad (36/5)$$



$$\min Z_1 = 28 - X_3 + 5X_4 + X_6$$

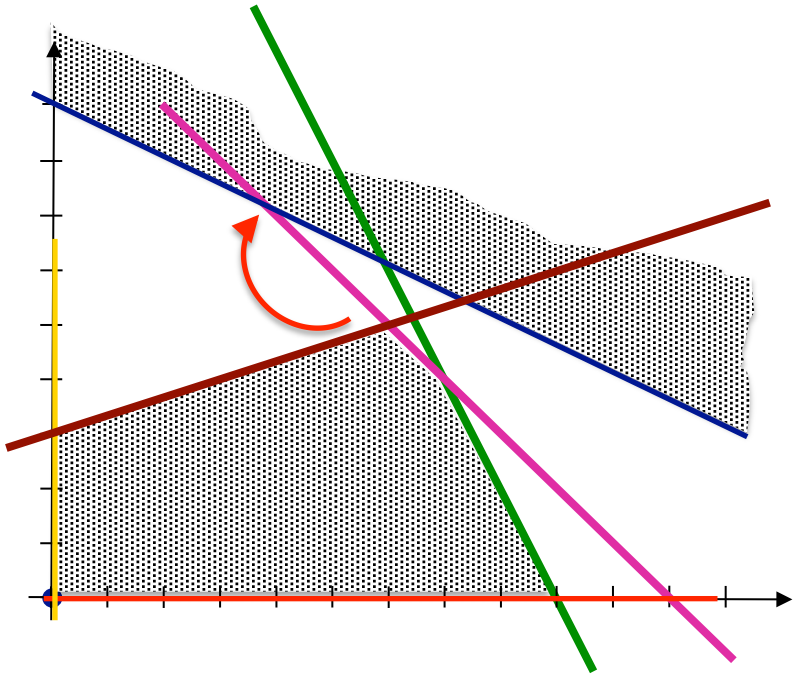
$$X_1 = 24 + X_3 - 3X_4$$

$$X_2 = 11 - X_4$$

$$X_5 = -41 - 2X_3 - 7X_4$$

$$Z_1 = 28 - X_3 + 5X_4 + X_6$$

Reification of Linear Constraints



$$\min Z_1 = 28 - X_3 + 5X_4 + X_6$$

$$X_1 = 24 + X_3 - 3X_4$$

$$X_2 = 11 - X_4$$

$$X_5 = -41 - 2X_3 - 7X_4$$

$$Z_1 = 28 - X_3 + 5X_4 + X_6$$

$$\min Z1$$

$$X_1 = 52 + 2X_4 + X_6 + Z1$$

$$X_2 = 11 - X_4$$

$$X_3 = 28 + 5X_4 + X_6 + Z_1$$

$$X_5 = -97 - 10X_4 - 2X_6 - 2Z_1$$

Reification of Linear Constraints

- At this point we detect failure, since making Z1 non-basic (i.e. Making it zero) turns one of the free coeficients of the equations negative.
- Hence, the system cannot be transformed into the solved form SF0 and is thus inconsistent.
- We may also pay special attention to the equation with the negative coefficient (agter removing the artificial variable,

$$x_5 = -97 - 10x_4 - 2x_6$$

$$\begin{aligned}
&\textbf{min} \quad z1 \\
&x_1 = 52 + 2x_4 + x_6 + z1 \\
&x_2 = 11 - x_4 \\
&x_3 = 28 + 5x_4 + x_6 + z_1 \\
&x_5 = -97 - 10x_4 - 2x_6 - 2z_1
\end{aligned}$$

Reification of Linear Constraints

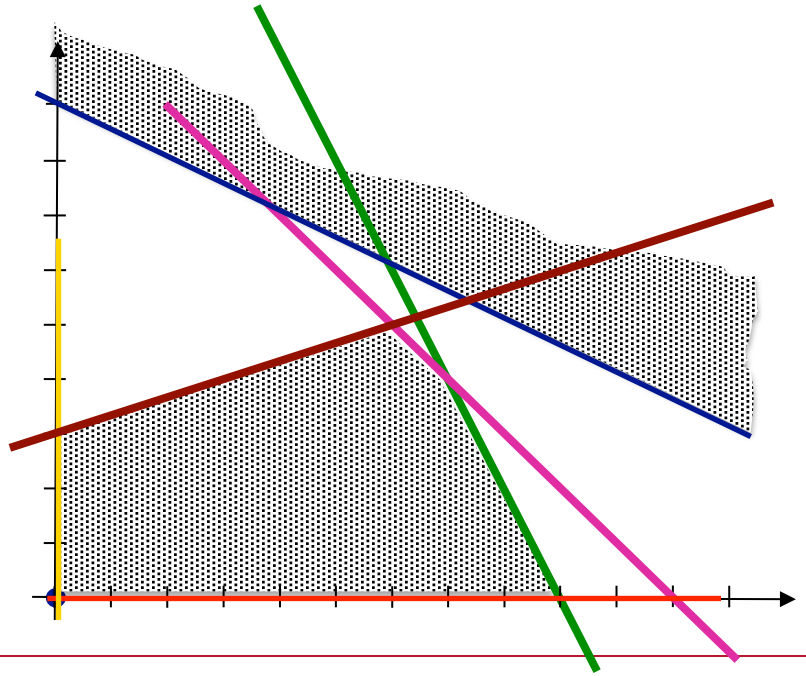
- In fact , we can rewrite this equation

$$x_5 = -97 - 10x_4 - 2x_6$$

into

$$10x_4 + x_5 + 2x_6 = -97$$

which shows that the inconsistency was caused by the constraints witnessed by variables X_4, X_5 and X_6 (i.e. X_3 played no role in it).



Reification of Linear Constraints

- If we return to the initial set of reified constraints

$$b3 \Leftrightarrow -x_1 + 3x_2 + x_3 = 9 \quad \ldots -x_1 + 3x_2 \leq 9$$
$$b4 \Leftrightarrow x_1 + x_2 + x_4 = 11 \quad \ldots x_1 + x_2 \leq 11$$
$$b5 \Leftrightarrow 2x_1 + x_2 + x_5 = 18 \quad \ldots 2x_1 + x_2 \leq 18$$
$$b6 \Leftrightarrow x_1 + 2x_2 + x_6 = 18 \quad \ldots x_1 + 2x_2 \geq 18$$

- What the equation

$$-10x_4 + x_5 + 2x_6 = -97$$

means is that there is a nogood on boolean variables **b4**, b5 and **b6**, and we can subsequently avoid its repetition by imposing the newly deduced nogood constraint

$$b4 + b5 + b6 \leq 2$$

Reification of Linear Constraints

- At the moment we have already integrated a linear constraint solver (GPLK) into the CP constraint solver we have developed Casper.

  available at http://proteina.di.fct.unl.pt/casper

- Initial tests with a poor integration (basically the communication between boolean variables and expressions was only made during enumeration, allowed us to develop a bounded model-checker that is slightly more efficient than state-of-the-art model checkers based on SAT and SMD solvers.

- But we need to do further testing

- Integrate Casper with a better solver that returns no goods (we are checking SCIP, developed at the Zuse Institute in Berlin).

Conclusion

- The variety of techniques that have been used to implement different solvers have all their strengths and weaknesses.

- Rather than finding the appropriate solver to solve each problem (how?) an alternative approach consists of hybridizing difefernt solvers to get the best of each.

- The lazy clause generation approach has already produced results at least as good as the best CP solvers in a varieaty of benchmarks.

- The reification of linear constraints is still quite experimental, but preliminary implementations are producing competitive results in specific applications.

- I hope this talk has attracted your attention to the hybridization of solvers, or at least, to the features of the different types of solvers that may help you with the modelling of applications.

Reification of Linear Expressions

Thank you for your attention