Introduction to SMT
Solving CSP's with SMT

Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, ...

SAT and SMT for Solving CSP's - Session 2

Seminar on Constraint Programming

31 March 2011

University of Bergen

Overview of the Session

- Pros/cons of SAT & Constraint Programming
- Satisfiability Modulo Theories
- Theories for Global Constraints

Good vs. Bad in SAT Solvers

What's GOOD?

- SAT solvers outperform other tools on real-world problems
- with a single, fully automatic variable selection strategy!
- Hence problem solving is essentially declarative

What's BAD?

- very low-level language: needs modeling and encoding tools
- no good encodings for many aspects: arithmetic, ...
- Optimization not as well studied as satisfiability

Good vs. Bad in CP Solvers

What's GOOD?

- Expressive modeling constructs and languages
- Specialized algorithms for many (global) constraints
- Optimization aspects better studied

What's BAD, or, well, not so good?

- Biased by random or artificial problems (not realistic)
- Performance(?)
  (no learning, backtracking instead of backjumping, ...)
- Not quite automatic or push-button
  Heuristics tuning per problem (or even per instance)

Why Are SAT Solvers Really Good?

Three key ingredients that only work if used TOGETHER:

- Learn at each conflict the backjump clause as a lemma:
  - makes UnitPropagate more powerful
  - prevents future similar conflicts
- Decide on variable with most occurrences in recent conflicts:
  - so-called activity-based heuristics
  - idea: work off clusters of tightly related variables
- Forget from time to time low-activity lemmas:
  - crucial to keep UnitPropagate fast and afford memory usage
  - idea: lemmas from worked off clusters no longer needed!

Not the Same Success in CP...

- Not easy to get everything together right
- Heuristics make solver work simultaneously on too unrelated vars
  - would require storing too many lemmas at the same time
- No simple uniform underlying language (as SAT's clauses):
  - hard to express lemmas (in SAT, 1st-class citizens: clauses)
  - hard to understand conflict analysis
  - hard to implement things really efficiently
- Learning lemmas not found very useful...
  - misled by random/academic pbs
  - Indeed, it is useless isolatedly, and also on random pbs!
- Can we get the best of the two worlds?
  See next slides for a solution

Overview of the Session

- Pros/cons of SAT & Constraint Programming
- Satisfiability Modulo Theories
- Theories for Global Constraints

What is Satisfiability Modulo Theories (SMT)?

- Some problems are more naturally expressed in other logics than propositional logic, e.g:
  - Software verification needs reasoning about equality, arithmetic, data structures, ...
- SMT consists in deciding the satisfiability of a (ground) first-order formula with respect to a background theory
- Example (Equality with Uninterpreted Functions – EUF):

$$g(a)=c \quad \wedge \quad (\ f(g(a)) \neq f(c) \vee g(a)=d\ ) \quad \wedge \quad c \neq d$$

- SMT is widely applied in hardware/software verification

  Theories of interest here:
  EUF, arithmetic, arrays, bit vectors, combinations of these
- With other theories SMT can also be used to solve Constraint Satisfaction Problems

Lazy Approach to SMT

Methodology:

Example: consider EUF and

$$\underbrace{g(a)=c}_{1} \;\wedge\; (\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}) \;\wedge\; \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1, \overline{2}\vee 3, \overline{4}\}$ to SAT solver

  SAT solver returns model $[1, \overline{2}, \overline{4}]$
  Theory solver says T-inconsistent

Lazy Approach to SMT

Methodology:

Example: consider EUF and

$$\underbrace{g(a)=c}_{1} \;\wedge\; (\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}) \;\wedge\; \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1, \overline{2} \vee 3, \overline{4}\}$ to SAT solver

  SAT solver returns model $[1, \overline{2}, \overline{4}]$
  Theory solver says T-inconsistent

- Send $\{1, \overline{2} \vee 3, \overline{4}, \overline{1} \vee 2 \vee 4\}$ to SAT solver

  SAT solver returns model $[1, 2, 3, \overline{4}]$
  Theory solver says T-inconsistent

Lazy Approach to SMT

Methodology:

Example: consider EUF and

$$\underbrace{g(a)=c}_{1} \ \wedge \ \big(\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}\big) \ \wedge \ \underbrace{c\neq d}_{\overline{4}}$$

- Send $\{1, \overline{2}\vee 3, \overline{4}\}$ to SAT solver

  SAT solver returns model $[1, \overline{2}, \overline{4}]$
  Theory solver says T-inconsistent

- Send $\{1, \overline{2}\vee 3, \overline{4}, \overline{1}\vee 2\vee 4\}$ to SAT solver

  SAT solver returns model $[1, 2, 3, \overline{4}]$
  Theory solver says T-inconsistent

- Send $\{1, \overline{2}\vee 3, \overline{4}, \overline{1}\vee 2\vee 4, \overline{1}\vee\overline{2}\vee\overline{3}\vee 4\}$ to SAT solver

  SAT solver says UNSATISFIABLE

Lazy Approach to SMT (2)

- Why "lazy"?
  Theory information used lazily when checking T-consistency of propositional models

- Characteristics:
  - \+ Modular and flexible
  - \- Theory information does not guide the search

- Tools:
  - Barcelogic (UPC)
  - CVC3 (Univ. New York + Iowa)
  - DPT (Intel)
  - MathSAT (Univ. Trento)
  - Yices (SRI)
  - Z3 (Microsoft)
  - ...

Lazy Approach to SMT - Optimizations

Several optimizations for enhancing efficiency:

- Check T-consistency only of full propositional models

Lazy Approach to SMT - Optimizations

Several optimizations for enhancing efficiency:

- ~~Check T-consistency only of full propositional models~~
- Check T-consistency of partial assignment while being built

Lazy Approach to SMT - Optimizations

Several optimizations for enhancing efficiency:

- ~~Check T-consistency only of full propositional models~~
- Check T-consistency of partial assignment while being built

- Given a T-inconsistent assignment M, add $\neg M$ as a clause

Lazy Approach to SMT - Optimizations

Several optimizations for enhancing efficiency:

- ~~Check T-consistency only of full propositional models~~
- Check T-consistency of partial assignment while being built

- ~~Given a T-inconsistent assignment M, add $\neg M$ as a clause~~
- Given a T-inconsistent assignment M, identify a T-inconsistent subset $M_0 \subseteq M$ and add $\neg M_0$ as a clause

Lazy Approach to SMT - Optimizations

Several optimizations for enhancing efficiency:

- ~~Check T-consistency only of full propositional models~~
- Check T-consistency of partial assignment while being built

- ~~Given a T-inconsistent assignment M, add $\neg M$ as a clause~~
- Given a T-inconsistent assignment M, identify a T-inconsistent subset $M_0 \subseteq M$ and add $\neg M_0$ as a clause

- Upon a T-inconsistency, add clause and restart

Lazy Approach to SMT - Optimizations

Several optimizations for enhancing efficiency:

- ~~Check T-consistency only of full propositional models~~
- Check T-consistency of partial assignment while being built

- ~~Given a T-inconsistent assignment M, add $\neg M$ as a clause~~
- Given a T-inconsistent assignment M, identify a T-inconsistent subset $M_0 \subseteq M$ and add $\neg M_0$ as a clause

- ~~Upon a T-inconsistency, add clause and restart~~
- Upon a T-inconsistency, do conflict analysis and backjump

Lazy Approach to SMT - Important Points

Advantages of the lazy approach:

- Everyone does what it is good at:
  - SAT solver takes care of Boolean information
  - Theory solver takes care of theory information
- Theory solver only receives conjunctions of literals
- Modular approach:
  - SAT solver and T-solver communicate via a simple API
  - SMT for a new theory only requires new T-solver
  - SAT solver can be extended to a lazy SMT system with very few new lines of code (40?)

Lazy Approach to SMT - Theory propagation

- As pointed out the lazy approach has one drawback:
  - Theory information does not guide the search
- How can we improve that? Theory propagation

T-Propagate

$$M \parallel F \quad \Rightarrow \quad M\,l \parallel F \quad \textbf{if} \begin{cases} M \models_T l \\ l \text{ or } \neg l \text{ occurs in } F \text{ and not in } M \end{cases}$$

- Search guided by T-Solver by finding T-consequences, instead of only validating it as in basic lazy approach.
- Naive implementation: Add $\neg l$. If T-inconsistent then infer l.

  But for efficient T-Propagate we need specialized T-Solvers
- This approach has been named DPLL(T)

DPLL(T) - Example

Consider again EUF and the formula:

$$\underbrace{g(a)=c}_{1} \;\wedge\; (\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}) \;\wedge\; \underbrace{c\neq d}_{\overline{4}}$$

$$\emptyset \parallel 1,\; \overline{2}\vee 3,\; \overline{4} \quad\Rightarrow\quad (\text{UnitPropagate})$$

DPLL(T) - Example

Consider again EUF and the formula:

$$\underbrace{g(a)=c}_{1} \;\wedge\; (\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}) \;\wedge\; \underbrace{c\neq d}_{\overline{4}}$$

$$\emptyset \parallel 1,\; \overline{2}\vee 3,\; \overline{4} \quad\Rightarrow\quad (\text{UnitPropagate})$$

$$1 \parallel 1,\; \overline{2}\vee 3,\; \overline{4} \quad\Rightarrow\quad (\text{T-Propagate})$$

DPLL(T) - Example

Consider again EUF and the formula:

$$\underbrace{g(a)=c}_{1} \;\wedge\; (\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}) \;\wedge\; \underbrace{c\neq d}_{\overline{4}}$$

$$\begin{array}{rclcl}
\emptyset & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow & (\text{UnitPropagate}) \\
1 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow & (\text{T-Propagate}) \\
1\ 2 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow & (\text{UnitPropagate})
\end{array}$$

DPLL(T) - Example

Consider again EUF and the formula:

$$\underbrace{g(a)=c}_{1} \;\wedge\; (\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}) \;\wedge\; \underbrace{c\neq d}_{\overline{4}}$$

$$\begin{array}{rcl}
\emptyset \;\|\; 1,\; \overline{2}\vee 3,\; \overline{4} & \Rightarrow & (\text{UnitPropagate}) \\
1 \;\|\; 1,\; \overline{2}\vee 3,\; \overline{4} & \Rightarrow & (\text{T-Propagate}) \\
1\,2 \;\|\; 1,\; \overline{2}\vee 3,\; \overline{4} & \Rightarrow & (\text{UnitPropagate}) \\
1\,2\,3 \;\|\; 1,\; \overline{2}\vee 3,\; \overline{4} & \Rightarrow & (\text{T-Propagate})
\end{array}$$

DPLL(T) - Example

Consider again EUF and the formula:

$$\underbrace{g(a)=c}_{1} \wedge \left(\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}\right) \wedge \underbrace{c\neq d}_{\overline{4}}$$

$$\begin{array}{rcll}
\emptyset & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow \quad (\text{UnitPropagate}) \\
1 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow \quad (\text{T-Propagate}) \\
1\ \overline{2} & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow \quad (\text{UnitPropagate}) \\
1\ \overline{2}\ 3 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow \quad (\text{T-Propagate}) \\
1\ \overline{2}\ 3\ 4 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow \quad (\text{Fail})
\end{array}$$

DPLL(T) - Example

Consider again EUF and the formula:

$$\underbrace{g(a)=c}_{1} \;\wedge\; (\,\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}\,) \;\wedge\; \underbrace{c\neq d}_{\overline{4}}$$

$$\begin{array}{rcllcl}
\emptyset & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow & (\text{UnitPropagate}) \\
1 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow & (\text{T-Propagate}) \\
1\,2 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow & (\text{UnitPropagate}) \\
1\,2\,3 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow & (\text{T-Propagate}) \\
1\,2\,3\,4 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow & (\text{Fail}) \\
\mathit{fail} & & & &
\end{array}$$

DPLL(T) - Example

Consider again EUF and the formula:

$$\underbrace{g(a)=c}_{1} \;\wedge\; (\underbrace{f(g(a))\neq f(c)}_{\overline{2}} \vee \underbrace{g(a)=d}_{3}) \;\wedge\; \underbrace{c\neq d}_{\overline{4}}$$

$$\begin{array}{rcll}
\emptyset & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow\ (\text{UnitPropagate}) \\
1 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow\ (\text{T-Propagate}) \\
1\,2 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow\ (\text{UnitPropagate}) \\
1\,2\,3 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow\ (\text{T-Propagate}) \\
1\,2\,3\,4 & \| & 1,\ \overline{2}\vee 3,\ \overline{4} & \Rightarrow\ (\text{Fail}) \\
\textit{fail} & & &
\end{array}$$

No search!

DPLL(T) - Overall algorithm

High-level view gives the same algorithm as a CDCL SAT solver:

```
while(true){

    while (propagate_gives_conflict()){
        if (decision_level==0) return UNSAT;
        else analyze_conflict();
    }

    restart_if_applicable();
    remove_lemmas_if_applicable();

    if (!decide()) returns SAT; // All vars assigned
}
```

Differences are in:

- `propagate_gives_conflict`
- `analyze_conflict`

DPLL(T) - Propagation

```
propagate_gives_conflict( ) returns Bool

    // unit propagate
    if ( unit_prop_gives_conflict() ) then return true

  return false
```

DPLL(T) - Propagation

```
propagate_gives_conflict( ) returns Bool

    do {

      // unit propagate
      if ( unit_prop_gives_conflict() ) then return  true

      // check T-consistency of the model
      if ( solver.is_model_inconsistent() ) then return  true

      // theory propagate
      solver.theory_propagate()

    } while  (doneSomeTheoryPropagation)

    return false
```

DPLL(T) - Propagation (2)

- Three operations:
  - Unit propagation (SAT solver)
  - Consistency checks (T-solver)
  - Theory propagation (T-solver)
- Cheap operations are computed first
- If theory is expensive, calls to T-solver are sometimes skipped
  - Only strictly necessary to call T-consistency at the leaves (i.e. when we have a full propositional model)
  - T-propagation is not necessary for correctness

DPLL(T) - Conflict Analysis

Remember conflict analysis in SAT solvers:

C`:= conflicting clause`

while C `contains more than one lit of last DL`

    l`:=last literal assigned in` C
    C`:=Resolution(`C`,reason(`l`))`

end while

`// let` $C = C' \vee l$ `where` l `is the only lit of last DL`
`backjump(maxDL(`C'`))`
`add` l `to the model with reason` C
`learn(`C`)`

DPLL(T) - Conflict Analysis

Conflict analysis in DPLL(T):

```
if boolean conflict then C:= conflicting clause
else  C:=¬( solver.explain_inconsistency() )

while C contains more than one lit of last DL

    l:=last literal assigned in C
    C:=Resolution(C,reason(l))

end while

// let C = C′ ∨ l where l is the only lit of last DL
backjump(maxDL(C′))
add l to the model with reason C
learn(C)
```

DPLL(T) - Conflict Analysis (2)

What does `explain_inconsistency` return?

- An explanation of the inconsistency:
  A (small) conjuntion of literals $l_1 \wedge \ldots \wedge l_n$ such that:
  - It is T-inconsistent
  - Lits were in the model when T-inconsistency was detected

What is now $reason(l)$?

- If l was unit propagated: clause that propagated it
- If l was T-propagated:
  - An explanation of the propagation:
    A (small) clause $\neg l_1 \vee \ldots \vee \neg l_n \vee l$ such that:
    - $l_1 \wedge \ldots \wedge l_n \models_T l$
    - $l_1, \ldots, l_n$ were in the model when l was T-propagated
  - Pre-compute explanations at each T-Propagate?
    Better only on demand, during conflict analysis

DPLL(T) - Conflict Analysis (3)

Let M be $c=b$ and let F contain

$$a=b \vee g(a)\neq g(b), \qquad h(a)=h(c) \vee p, \qquad g(a)=g(b) \vee \neg p$$

Take the following sequence:

1. Decide $h(a)\neq h(c)$
2. T-Propagate $a\neq b$ (due to $h(a)\neq h(c)$ and $c=b$)
3. UnitPropagate $g(a)\neq g(b)$
4. UnitPropagate p
5. Conflicting clause $g(a)=g(b) \vee \neg p$

Explain($a\neq b$) is $\{h(a)\neq h(c), c=b\}$

$$\dfrac{h(a)=h(c)\vee c\neq b\vee a\neq b \qquad \dfrac{a=b\vee g(a)\neq g(b) \qquad \dfrac{h(a)=h(c)\vee p \qquad g(a)=g(b)\vee\neg p}{h(a)=h(c)\vee g(a)=g(b)}}{h(a)=h(c)\vee a=b}}{h(a)=h(c)\ \vee\ c\neq b}$$

DPLL(T) – T-Solver API in a Nutshell

What does DPLL(T) need from T-Solver?

- T-consistency check of a set of literals M, with:
  - Explain of T-inconsistency:
    find small T-inconsistent subset of M
  - Incrementality: if l is added to M,
    check for $M\,l$ faster than reprocessing $M\,l$ from scratch.
- Theory propagation: find input T-consequences of M, with:
  - Explain T-Propagate of l:
    find (small) subset of M that T-entails l.
- Backtrack n: undo last n literals added

Overview of the Session

- Pros/cons of SAT & Constraint Programming
- Satisfiability Modulo Theories
- Theories for Global Constraints

SMT(all_different)

- $\texttt{all\_different}(x_1, \ldots, x_n)$ if $x_1, \ldots, x_n$ take different values
- Global constraint appearing in many CSP's

  Example 1: Round-Robin Sports Scheduling

  Example 2: Quasi-Group Completion (QGC)
  Each row, column in a part. filled grid $n \times n$ must contain $1, \ldots n$

  Vars x_{ij} standing for value at row i, column j

$$\text{no repetitions in rows} \quad \begin{cases} \texttt{all\_different}(x_{1\,1}, x_{1\,2}, \ldots, x_{1\,n-1}, x_{1\,n}) \\ \ldots \\ \texttt{all\_different}(x_{n\,1}, x_{n\,2}, \ldots, x_{n\,n-1}, x_{n\,n}) \end{cases}$$

$$\text{no repetitions in cols} \quad \begin{cases} \texttt{all\_different}(x_{1\,1}, x_{2\,1}, \ldots, x_{n-1\,1}, x_{n\,1}) \\ \ldots \\ \texttt{all\_different}(x_{1\,n}, x_{2\,n}, \ldots, x_{n-1\,n}, x_{n\,n}) \end{cases}$$

- Specialized filtering algorithms exist in CP

SMT(`all_different`) (2)

- 3-D SAT encoding infers no value here by unit propagation
- `all_different` filtering infers $z = 3$
  Why?

x	y	z		
	3	4		
3	4	5		
4	5			
5				

SMT(all_different) (2)

- 3-D SAT encoding infers no value here by unit propagation
- `all_different` filtering infers $z = 3$
  Why? Because $\{x, y\} = \{1, 2\}$

x	y	z		
	3	4		
3	4	5		
4	5			
5				

SMT(`all_different`) (2)

- 3-D SAT encoding infers no value here by unit propagation
- `all_different` filtering infers $z = 3$
  Why? Because $\{x,y\} = \{1,2\}$

x	y	z		
	3	4		
3	4	5		
4	5			
5				

Idea:

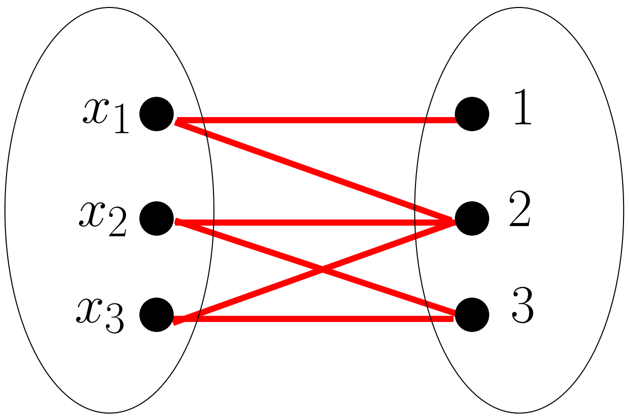
- Use 3-D encoding + SMT where T is `all_different`
- T-solver is incremental CP filtering but with explain:
  in our example, the literal p_{133} (meaning $z = 3$) is entailed by $\{\overline{p_{113}}, \overline{p_{114}}, \ldots, \overline{p_{135}}\}$ (meaning $x \neq 3, x \neq 4, \ldots, z \neq 5$)
- From time to time invoke T-solver before Decide, but do always cheap SAT stuff first: Backjump, UnitPropagate, etc.

Value Graph of `all_different`

- A graph $G = (V, E)$ is bipartite
  if V can be partitioned into two disjoint sets U and V such that
  all edges have one endpoint in U and the other in V
- Given variables $X = \{x_1, \ldots, x_n\}$ with domains $D_1, \ldots, D_n$,
  $(x_1 = \alpha_1, \ldots, x_n = \alpha_n)$ is a solution to `all_different`$(x_1, \ldots, x_n)$
  iff $\alpha_i \in D_i$, and $i \neq j$ implies $\alpha_i \neq \alpha_j$
- The value graph of `all_different`$(x_1, \ldots, x_n)$ is the bipartite
  graph $G = (X \cup \bigcup_{i=1}^{n} D_i, E)$ where $(x_i, d) \in E$ iff $d \in D_i$
- For simplicity, we will assume that $|X| = |\bigcup_{i=1}^{n} D_i|$

`all_different`(x_1, x_2, x_3)

$$\begin{aligned} D_1 &= \{1, 2\} \\ D_2 &= \{2, 3\} \\ D_3 &= \{2, 3\} \end{aligned}$$

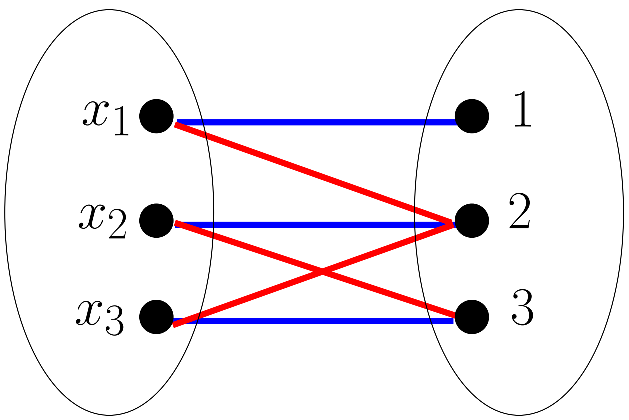


Matching Theory

- A matching M in a graph $G = (V, E)$ is a subset of edges in E without common vertices
- A maximum matching is a matching of maximum size
- A matching M covers a set X if every vertex in X is an endpoint of an edge in M
- Solutions to `all_different`(X) = matchings covering X

`all_different`(x_1, x_2, x_3)

$$
\begin{array}{lcl}
D_1 & = & \{1,2\} \\
D_2 & = & \{2,3\} \\
D_3 & = & \{2,3\}
\end{array}
\qquad
\begin{array}{l}
x_1 = 1 \\
x_2 = 2 \\
x_3 = 3
\end{array}
$$



Matching Theory

- A matching M in a graph $G = (V, E)$ is
  a subset of edges in E without common vertices
- A maximum matching is a matching of maximum size
- A matching M covers a set X
  if every vertex in X is an endpoint of an edge in M
- Solutions to `all_different`(X) = matchings covering X
- Algorithm for checking satisfiability of `all_different`(X):

```
// Returns true if there is a solution, otherwise false
M = Compute_maximum_matching(G)
if ( |M| < |X| ) return false

return true
```

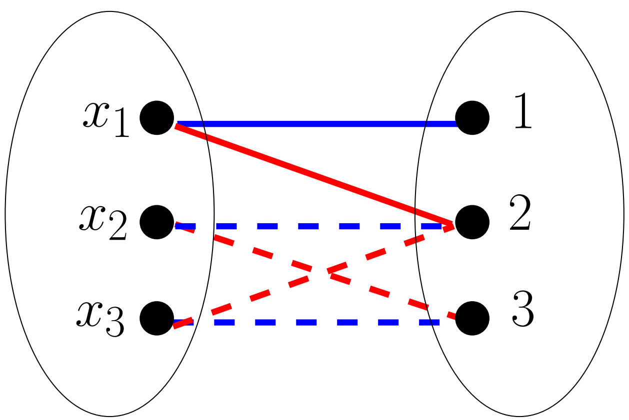
Matching Theory

- A matching M in a graph $G = (V, E)$ is
  a subset of edges in E without common vertices
- A maximum matching is a matching of maximum size
- A matching M covers a set X
  if every vertex in X is an endpoint of an edge in M
- Solutions to `all_different`(X) = matchings covering X
- Algorithm for checking satisfiability of `all_different`(X):
- Can be extended to filter out arc-inconsistent edges

```
// Returns true if there is a solution, otherwise false
M = Compute_maximum_matching(G)
if ( |M| < |X| ) return false
Remove_edges_from_graph(G, M)
return true
```

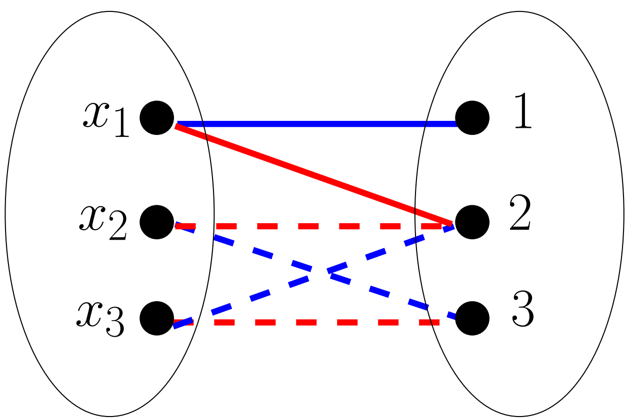
Matching Theory (2)

- **Theorem.** `all_different`(X) is arc-consistent iff every edge of the graph belongs to a matching covering X
- A matching edge belongs to the matching , else it is free
- An alternating cycle is a simple cycle whose edges are alternately matching and free
- A vital edge belongs to any maximum matching
- **Theorem.** A non-vital edge belongs to a maximum matching iff for an arbitrary maximum matching M it belongs to an even-length alternating cycle wrt. M



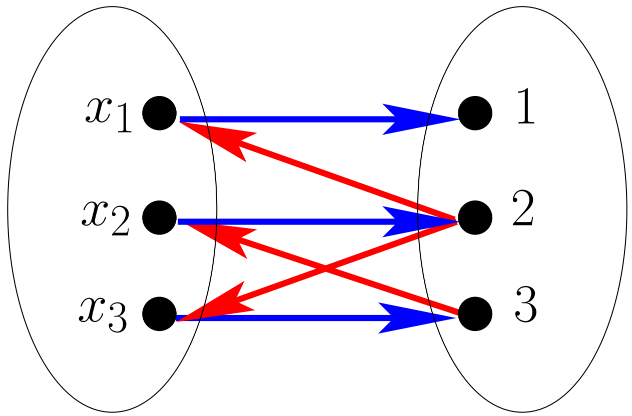
Matching Theory (2)

- **Theorem.** `all_different`(X) is arc-consistent iff every edge of the graph belongs to a matching covering X
- A matching edge belongs to the matching , else it is free
- An alternating cycle is a simple cycle whose edges are alternately matching and free
- A vital edge belongs to any maximum matching
- **Theorem.** A non-vital edge belongs to a maximum matching iff for an arbitrary maximum matching M it belongs to an even-length alternating cycle wrt. M

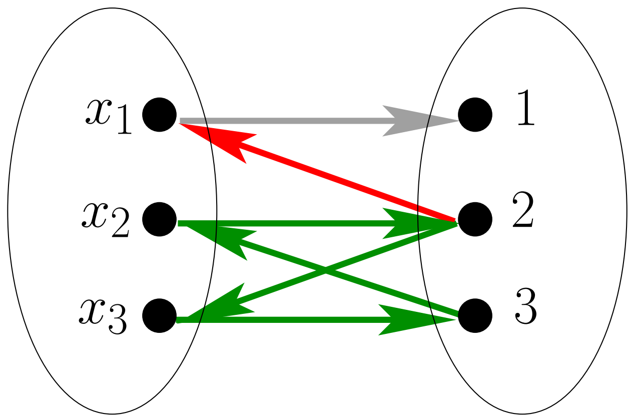


Matching Theory (3)

- It simplifies things to orient edges:
  - Matching edges are oriented from left to right
  - Free edges are oriented from right to left



- **Theorem.** A non-vital edge belongs to a max matching iff for any max matching *M* it belongs to a cycle in oriented graph

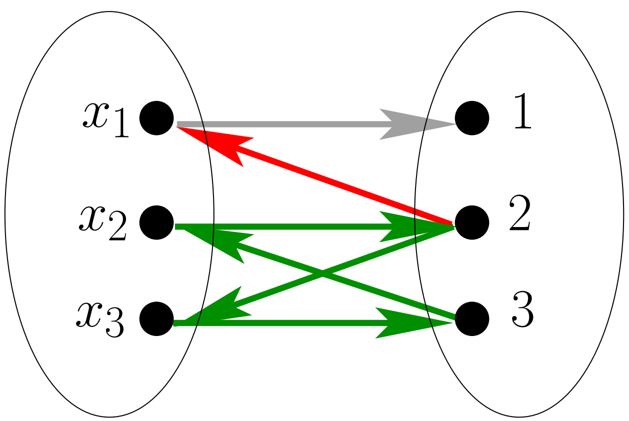


Removing Arc-Inconsistent Edges

```
Remove_edges_from_graph(G)
    mark all edges in G as UNUSED
    compute SCCs, mark as USED edges with vertexs in same SCC
    mark matching UNUSED edges as vital
    remove remaining UNUSED edges
```

- Removed edges are free edges whose endpoints belong to different SCCs
- Explanation of removed edge (x, d) requires expressing x and d do not belong to the same SCC

$\overline{(x_1, 2)}$ since $\{\overline{(x_2, 1)}, \overline{(x_3, 1)}\}$
since x_2, x_3 consume 2, 3



SMT(PB Constraints)

- A pseudo-boolean (PB) constraint is of the form
  $a_1x_1 + \ldots + a_nx_n \leq k$ where $x_i \in \{0,1\}$, $a_i, k \in \mathbb{Z}$
- PB constraints appear in many contexts
  (e.g. weighted Max-SAT, `cumulative`: see later)
- SAT encodings not appropriate if there are many PB cons:
  too big formulas!

Idea:

- Use T-solver for each PB constraint:
  T-solver enforces arc-consistency of its PB constraint
- Alternatively, a single T-solver can take care of all PB cons and share information for better filtering

SMT(PB Constraints) (2)

Example of filtering by arc-consistency:

- Assume: $a_1x_1 + \ldots + a_nx_n \leq k$ with $a_i \geq 0$
- Let $I_0 = \{i \mid x_i = 0\},\ I_1 = \{i \mid x_i = 1\},\ I_\perp = \{i \mid x_i = \perp\}$
- Then $a_1x_1 + \ldots + a_nx_n \leq k$ becomes

$$
\begin{array}{ccccccl}
\underbrace{\Sigma_{i\in I_0} a_i \cdot 0}_{0} & + & \Sigma_{i\in I_1} a_i \cdot 1 & + & \Sigma_{i\in I_\perp} a_i x_i & \leq & k \\
 & & \Sigma_{i\in I_1} a_i & + & \Sigma_{i\in I_\perp} a_i x_i & \leq & k \\
 & & & & \Sigma_{i\in I_\perp} a_i x_i & \leq & k - \Sigma_{i\in I_1} a_i
\end{array}
$$

- If $j \in I_\perp$ is such that $a_j > k - \Sigma_{i\in I_1} a_i$, then it must be $x_j = 0$
- Explanation?

SMT(PB Constraints) (2)

Example of filtering by arc-consistency:

- Assume: $a_1x_1 + \ldots + a_nx_n \leq k$ with $a_i \geq 0$
- Let $I_0 = \{i \mid x_i = 0\}$, $I_1 = \{i \mid x_i = 1\}$, $I_\perp = \{i \mid x_i = \perp\}$
- Then $a_1x_1 + \ldots + a_nx_n \leq k$ becomes

$$\begin{array}{ccccll}
\underbrace{\Sigma_{i \in I_0} a_i \cdot 0}_{0} & + & \Sigma_{i \in I_1} a_i \cdot 1 & + & \Sigma_{i \in I_\perp} a_i x_i & \leq k \\
 & & \Sigma_{i \in I_1} a_i & + & \Sigma_{i \in I_\perp} a_i x_i & \leq k \\
 & & & & \Sigma_{i \in I_\perp} a_i x_i & \leq k - \Sigma_{i \in I_1} a_i
\end{array}$$

- If $j \in I_\perp$ is such that $a_j > k - \Sigma_{i \in I_1} a_i$, then it must be $x_j = 0$
- Explanation?
- A set $\{x_i = 1 \mid i \in J\}$ where $J \subseteq I_1$ is such that $a_j > k - \Sigma_{i \in J} a_i$

SMT(`cumulative`)

- n tasks share common resource with capacity c. Each task:
  - has a duration d_i
  - consumes r_i units of resource per hour
  - must start not before est_i (earliest starting time)
  - must end not after let_i (latest ending time)
  - once started, cannot be interrupted
- horizon $h_{\max}$ = latest time any task can end = $max_{i \in \{1...n\}} let_i$
- `cumulative`$(s_1, \ldots, s_n)$ is satisfied by starting times $s_1, \ldots, s_n$ if:
  - at all times used resources do not exceed capacity:

$$\forall h \in \{0, \ldots, h_{\max} - 1\} : \qquad \sum_{\substack{i \in \{1...n\}: \\ s_i \le h \le s_i + d_i}} r_i \le c$$

  - starting times respect feasible window:

$$\forall i \in \{1 \ldots n\} : \qquad est_i \le s_i, \quad s_i + d_i \le let_i$$

SMT(`cumulative`) (2)

Pure SMT approach, modeling with variables $s_{i,h}$:

- $s_{i,h}$ means $s_i \leq h$ (so $\overline{s_{i,h-1}} \wedge s_{i,h}$ means $s_i = h$)
- T-solver propagates using CP filtering algs. with explanations

Better "decomposition" approach, adding variables $a_{i,h}$:

- $a_{i,h}$ means task i is active at hour h
- Time-resource decomposition:
  quadratic no. of clauses like
  - $\overline{s_{i,h-d_i}} \wedge s_{i,h} \longrightarrow a_{i,h}$
  - $a_{i,h} \longrightarrow \overline{s_{i,h-d_i}}$
  - $a_{i,h} \longrightarrow s_{i,h}$
- T-solver handles, for each hour h and each resource r,
  PB constraints like $3a_{i,h} + 4a_{i',h} + \ldots \leq capacity(r)$

Comparison with Lazy Clause Generation

Lazy Clause Generation (LCG) was the instance of SMT where:

- each time the T-solver detects that lit can be propagated, it generates and adds (forever) the explanation clause so the SAT-solver can UnitPropagate lit with it.

But as we have seen in this seminar, it is usually better to:

- Generate explanations only when needed: at conflict analysis time
- Never add explanations as clauses. Otherwise: die keeping too many explanations (or the whole SAT encoding).

  Remember: Forget of the usual lemmas is already crucial to keep UnitPropagate fast and memory affordable!

Since recently, with these improvements, LCG = SMT.

Bibliography - Some further reading

- A. Aggoun, N. Beldiceanu. *Extending CHIP in Order to Solve Complex Scheduling and Placement Problems*. Mathematical and Computer Modelling 17(7), 57-73 (1993)
- A. Schutt, T. Feydy, P. Stuckey, M. Wallace. *Why `Cumulative` Decomposition Is Not as Bad as It Sounds*. CP 2009.
- J-C. Régin. *A Filtering Algorithm for Constraints of Difference in CSPs*. AAAI (1994).
- R. Nieuwenhuis, A. Oliveras, C. Tinelli. *Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)*. J. ACM 53(6): 937-977 (2006)
- C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli. *Satisfiability Modulo Theories*. Handbook of Satisfiability 2009: 825-885
- O. Ohrimenko, P. Stuckey, M. Codish. *Propagation = Lazy Clause Generation*. CP 2007.
- R. Sebastiani. *Lazy Satisfiability Modulo Theories*. JSAT 3(3-4): 141-224 (2007).