Microsoft Research

Propositional Satisfiability & Constraint Programming

Youssef Hamadi

Microsoft Research,

Cambridge, UK.

Goal of the talk

- Understand the similarities and differences of two problem solving formalisms
  - Propositional SATisfiability
  - Constraint Programming
- Know which formalism to use for a particular problem
- Draft research directions

A first reference

Propositional Satisfiability and Constraint Programming: a Comparative Survey

L. Bordeaux, Y. Hamadi and L. Zhang

ACM Computing Surveys 2006

By no means the only ref. on SAT:

- forthcoming handbooks,
- survey by David Mitchell...

Outline

- Introduction
- Algorithms for SAT & CP
- Programming SAT & CP
- Lessons learned, integration(?)
- Perspectives & Conclusion

INTRODUCTION

SAT & CP in brief

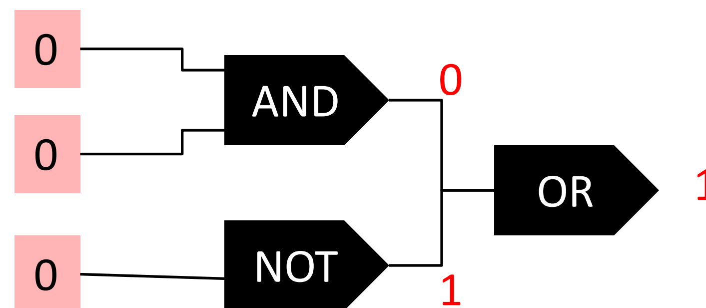
- SAT & CP, two automated reasoning technologies:
  - NP-hard problem: P
  - Language: modelling(P)
  - Solver: solve(P)

Very similar...

Very different...

A few things to know about SAT

- SAT means “propositional satisfiability”
  - Given a Boolean circuit, find an input for which the circuit evaluates to true.



A few things to know about SAT

- An interesting restriction of SAT is *Conjunctive Normal Form (CNF)*

$$(x) \vee (\neg y) \vee (z) \vee (\neg t)$$

A literal

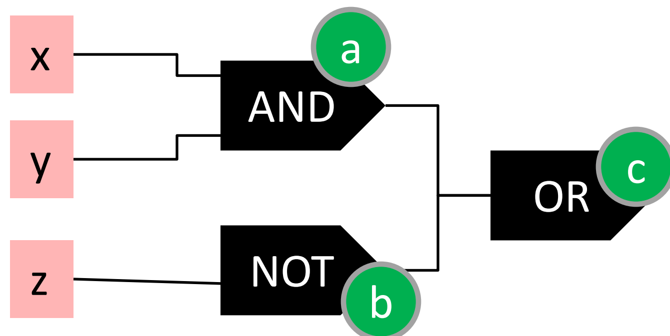
A clause

- Note that a clause is a *no-good*: it forbids a partial assignment

$$\neg [(x=0) \wedge (y=1) \wedge (z=0) \wedge (t=1)]$$

A few things to know about SAT

- SAT Instances can be put in CNF



-Add variables for each intermediate result

a=(x .y), b=(-z) c=(a+b)

-Express the relations between these vars by clauses

(!a + x), (!a + y) (!x + !y + a)

...

-Constrain output to be true (c)

- We obtain a formula that is *equi-satisfiable*

$$(x \wedge y) \vee \neg z \leftrightarrow \exists a,b,c. \begin{pmatrix} a = (x \wedge y), & b = (\neg z), \\ c = (a \vee b), & c \end{pmatrix}$$

Modeling in SAT

- Input:
  - Language: Conjunctive Normal Form formula,

$\sum = (l_1 \vee l_2 \vee \ldots \vee l_{c1}) \wedge \ldots \wedge (l_1 \vee l_2 \vee \ldots \vee l_{ck})$

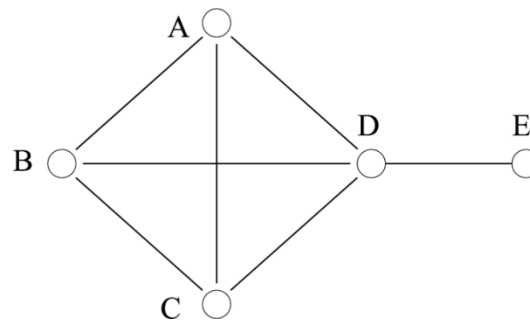
  - n Boolean variables
- Output:
  - Yes [+ [smallest] model]
  - No [+ core]
- Remark:
  - Several *encodings* bring different performances
    - Size n, k
    - # of solutions,
    - Symmetries,

Modeling in CP

- Input:
  - Variables, range of values : [lb .. ub]
  - Constraints available
    - Numerical comparisons, *e.g.* x≤y and basic arithmetic, *e.g.* 10.x=y, y+z=u
    - Logical/meta constraints, *e.g.* (AvB)↔C and *logical combinations of constraints*
    - Symbolic constraints, *e.g.* T[X] > Y indirection between variables, etc.
    - Global constraints on lists of arguments
      - allDiff(x1, …, xn)
      - Constraints on cardinality and occurrences
      - Etc.
- Output: [best-]solution

Example: graph-coloring

- SAT



Vertex A:

•Booleans: A_1, A_2, A_3,

•"each vertex has one color":$(A_1 \vee A_2 \vee A_3)$

•"one color at the time":$(\neg A_1 \vee \neg A_2) \wedge (\neg A_2 \vee \neg A_3) \wedge (\neg A_3 \vee \neg A_1)$

Etc.

Edges: different colors

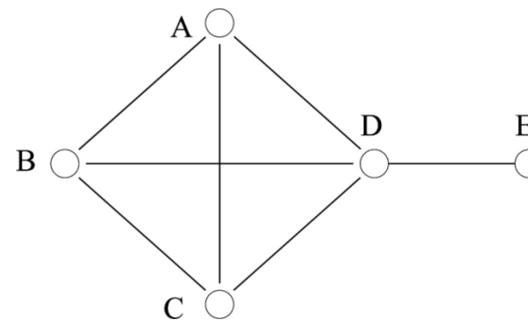
•(A,B):$(\neg A_1 \vee \neg B_1) \wedge (\neg A_2 \vee \neg B_2) \wedge (\neg A_3 \vee \neg B_3)$

•Etc.

... CNF: n=15, k=41

Example: graph-coloring

- CP



Vertex A:
•Finite Domain: A = {1,2,3}
Etc.

Edges: binary constraints
A ≠ B, B ≠ C, C ≠ D, A ≠ C, A ≠ D, B ≠ D, D ≠ E

5 variables, 3 values, 7 constraints

Alternative model: N-ary + binary constraints: allDiff(A,B,C,D), D ≠ E

Comparison

SAT

- Low-level
  - "Assembly" language for decision procedures.

CP

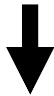
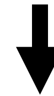
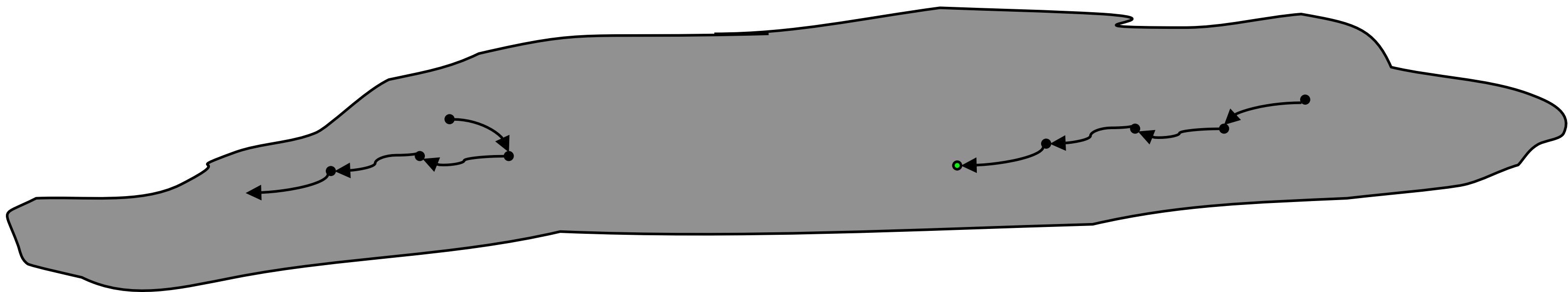
- High-level
  - Real language with rich set of constructs and constraints.

ALGORITHMS FOR SAT & CP

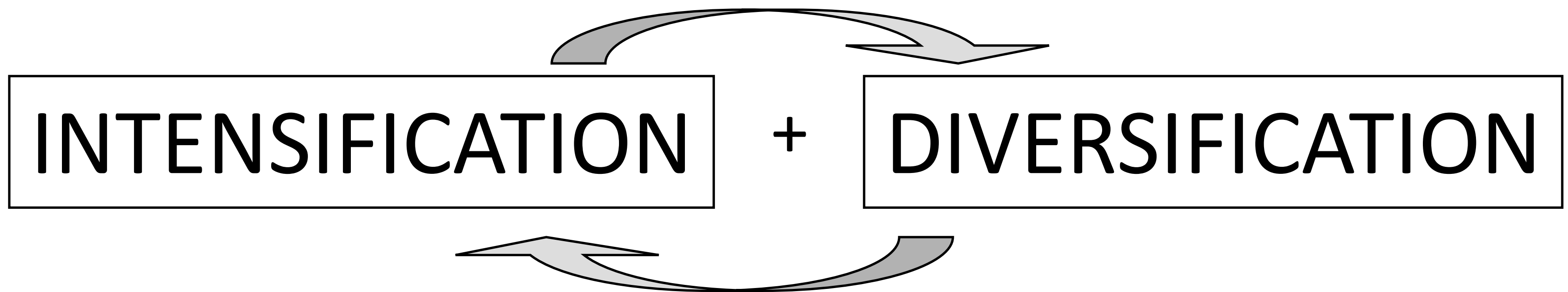
LOCAL SEARCH

Local Search: general principle

INTENSIFICATION + DIVERSIFICATION

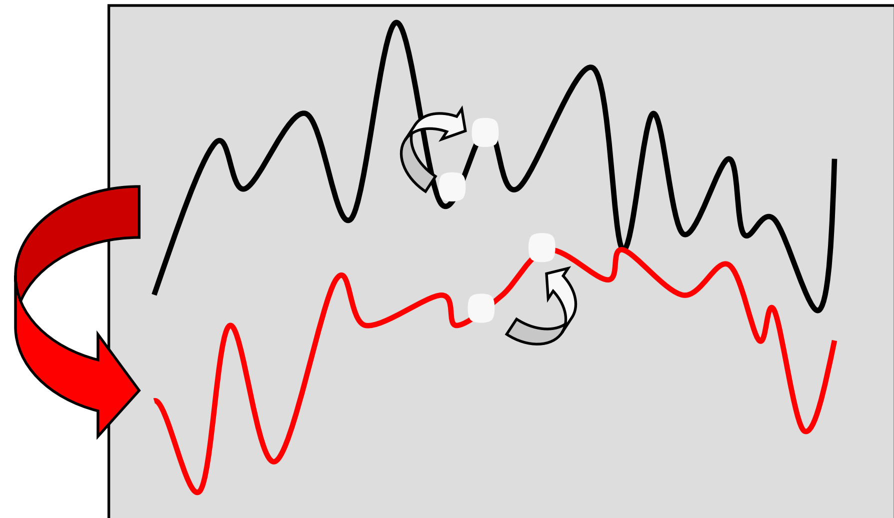




Local Search: general principle



Local Search: general principle

- Guided Local Search
  - Modify the evaluation function to escape minima
    - CP [Voudouris 96]
    - SAT [Hutter et al. 02]



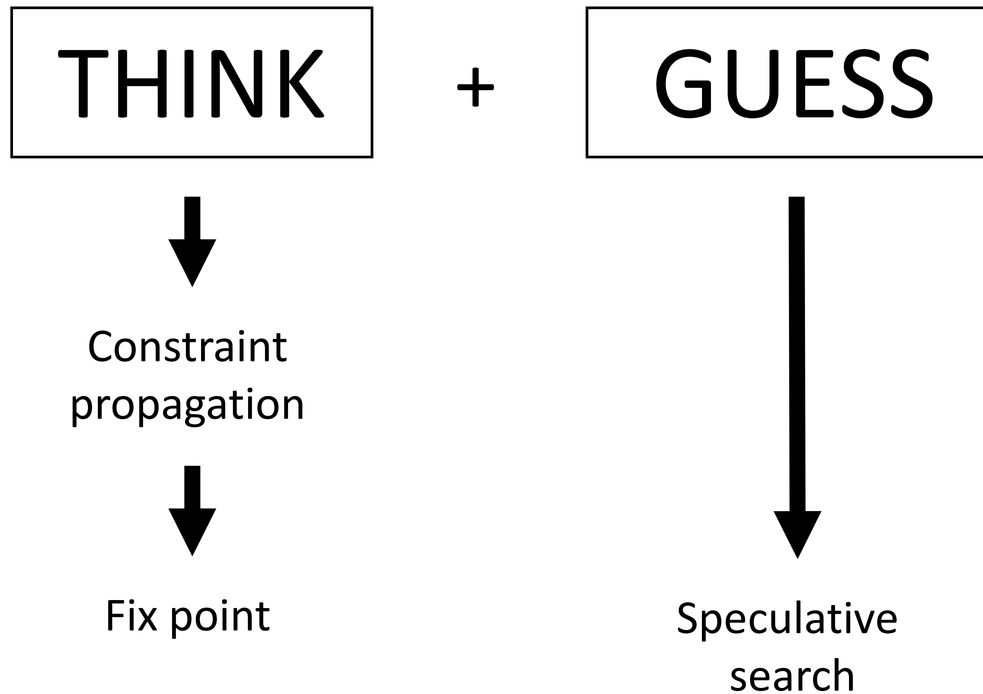
Local Search in SAT & CP

- In SAT
  - Flip(x), maximizes the improvement in quality
  - Quality = # satisfied clauses

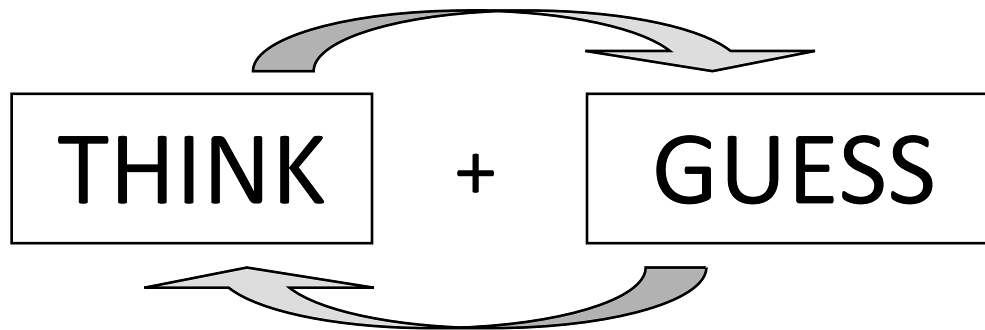
- In CP
  - Arithmetic, x != y: reports |x-y|
  - Global constraints: reports their insatisfaction
    - E.g. AllDifferent, number of pairs of variables with equal values
      - Minimal number of move to reach satisfaction

COMPLETE SEARCH

Complete Search: general principle



Complete Search: general principle



COMPLETE SEARCH IN SAT

Branching

- MOMS (M), introduced in [Pretolani, 1993], which prefers variables that occur frequently in the shortest clauses;
- – Jeroslow-Wang (JW), see [Jeroslow and Wang, 1990], where the occurrences of variables in short clauses are exponentially better than those in long clauses;
- – Bohm (B), discussed in [Buro and Buning, 1992], which considers occurrences in clauses of any length and, in case of ties, prefers variables occurring frequently in short clauses;
- – SATZ (S), as explained in [Li and Anbulagan, 1997], which features a complex scoring mechanism based on BCP and a modified version of JW;
- – Unitie0 (U0), introduced by [Copty et al., 2001] under the name “Unit”, which prefers variables producing the highest simplification with BCP.

Branching

- Select a variable
  - **VSID heuristic**: rank the variables according to their number of "occurrences" in the clause database.
  - Pick up the variable with highest score.
  - **Regular decay of the scores**
  - Rem: dynamic ordering since new clauses are learnt through backtracking…

Propagation

- DPLL makes particular choices of the form x=0, x=1. In other words, a choice is a literal.
- Propagating literals is a particular form of resolution called unit resolution:

$$\frac{A \vee x \qquad \neg x \vee B}{A \vee B}$$

resolution

$$\frac{x \qquad \neg x \vee B}{B}$$

unit resolution

Propagation

The problem:

- Determine as fast as possible the clauses that become unit (i.e. all but one literals false → we must impose the remaining one)

!x1	x2	x3	x4	!x5	x6	x7	x8	!x9	x10

Propagation

- The old-fashioned way

!x1	x2	x3	x4	!x5	x6	x7	x8	!x9	x10

We subscribe to the events related to all 10 literals.

We want to detect the case where one single literal remains possible.

QUIZZ

How can we react to each event in constant time?

Propagation

- The old-fashioned way

Possible answer: we attach a counter to each clause

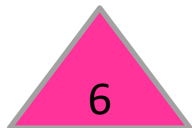
Propagation

- The old-fashioned way

Each time we are notified for an event (falsified literal) we simply decrease the counter

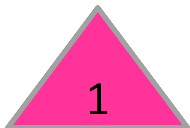
Propagation

- The old-fashioned way



Propagation

- The old-fashioned way



A counter at 1 means that we have a unit clause

(we can find the literal to propagate by iteration or maintain backtrackable stuff to locate it in constant time)

Propagation

- The old-fashioned way is in a sense optimal (linear time, and even amortized linear time)
- BUT each variable maintains two static lists (clauses in which appears positively/negatively).

  More constraints means slower propagation.
- Solution:
  – Use dynamic lists
  – Rely on backtrackable data-structures

Propagation

- Head/Tail approach

Microsoft Research

Propagation

- Head/Tail approach

Microsoft Research

Propagation

- Head/Tail approach

We just subscribe to 2 literals

Propagation

- Head/Tail approach

When a literal becomes false we move the pointer

We stop at first *free or satisfied* literal and (if free) change dynamically subscribe to this literal

Propagation

- Head/Tail approach

When the two pointers are equal, the clause is unit

(if the 2 pointers meet at literal false then failure)

Propagation

- 2-watch approach

By the way, why did we impose this invariant that the pointers be specifically at head and tail?

Propagation

- 2-watch approach

Propagation

- 2-watch approach

Microsoft Research

Propagation

- 2-watch approach

When a watched literal gets violated: we search for a free literal at a position different from the other pointer

Propagation

- 2-watch approach

When the only free position is the other Pointer, the corresponding literal is unit (when all positions incorrect, fail)

QUIZZ

Can you see the advantage over Head/tail?

Propagation

- 2-watch approach

Answer: the pointers don't even need to be backtrackable

Propagation

- 2-watch approach

Answer: the pointers don't even need to be backtrackable

Backtracking

- Failure detected during the propagation process

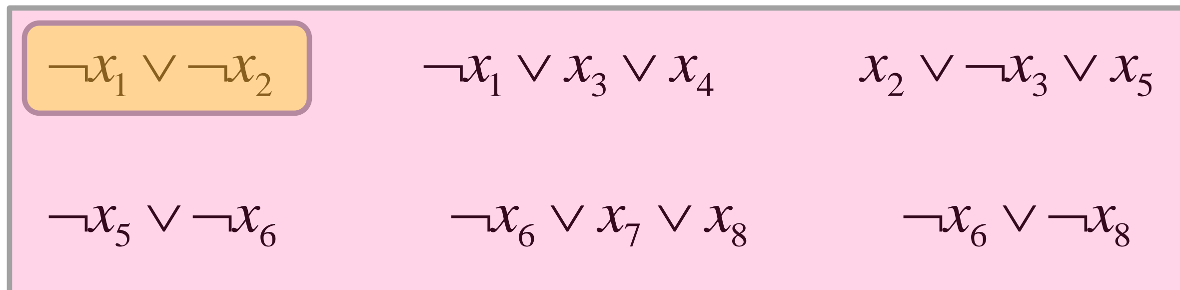
1. Conflict Analysis
2. Backtracking

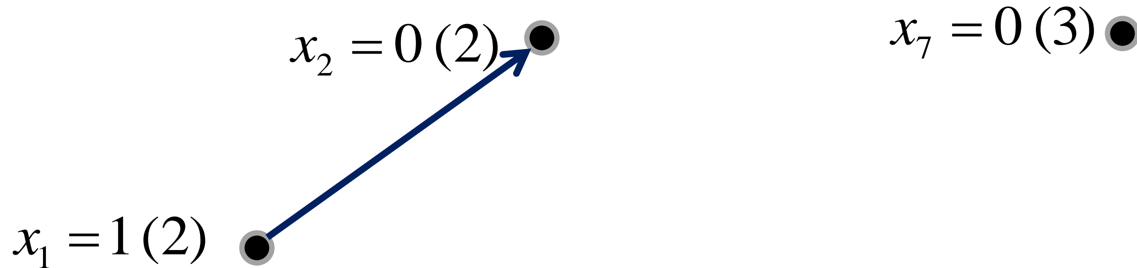
Conflict Analysis

- First thing is to trace the reasoning: notion of implication graph

- Note that cheaply computed in SAT context
- Note that each clause used exactly once

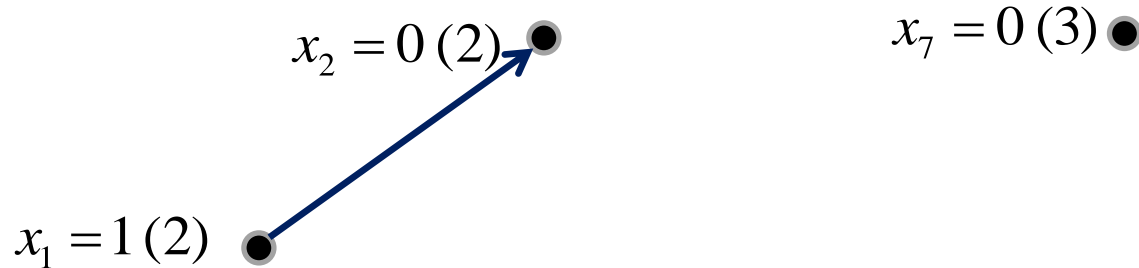
Implication Graphs



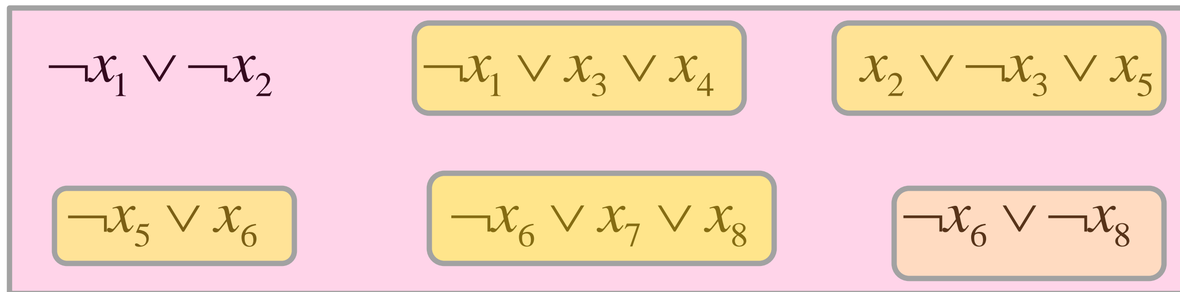


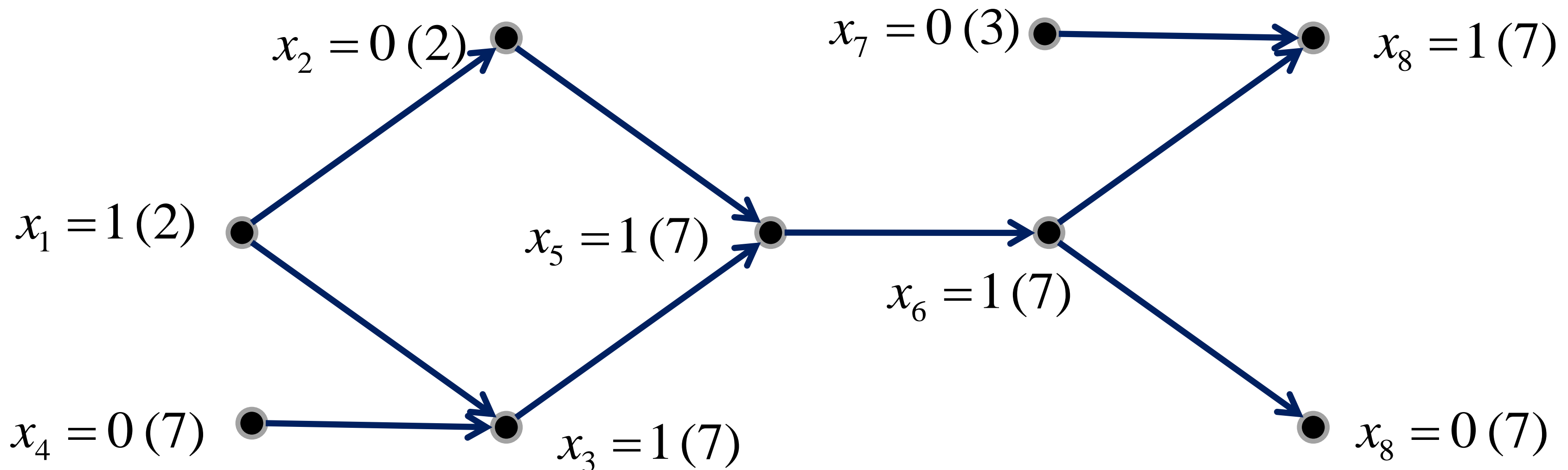
Implication Graphs

$\neg x_1 \vee \neg x_2$	$\neg x_1 \vee x_3 \vee x_4$	$x_2 \vee \neg x_3 \vee x_5$
$\neg x_5 \vee x_6$	$\neg x_6 \vee x_7 \vee x_8$	$\neg x_6 \vee \neg x_8$



Implication Graphs



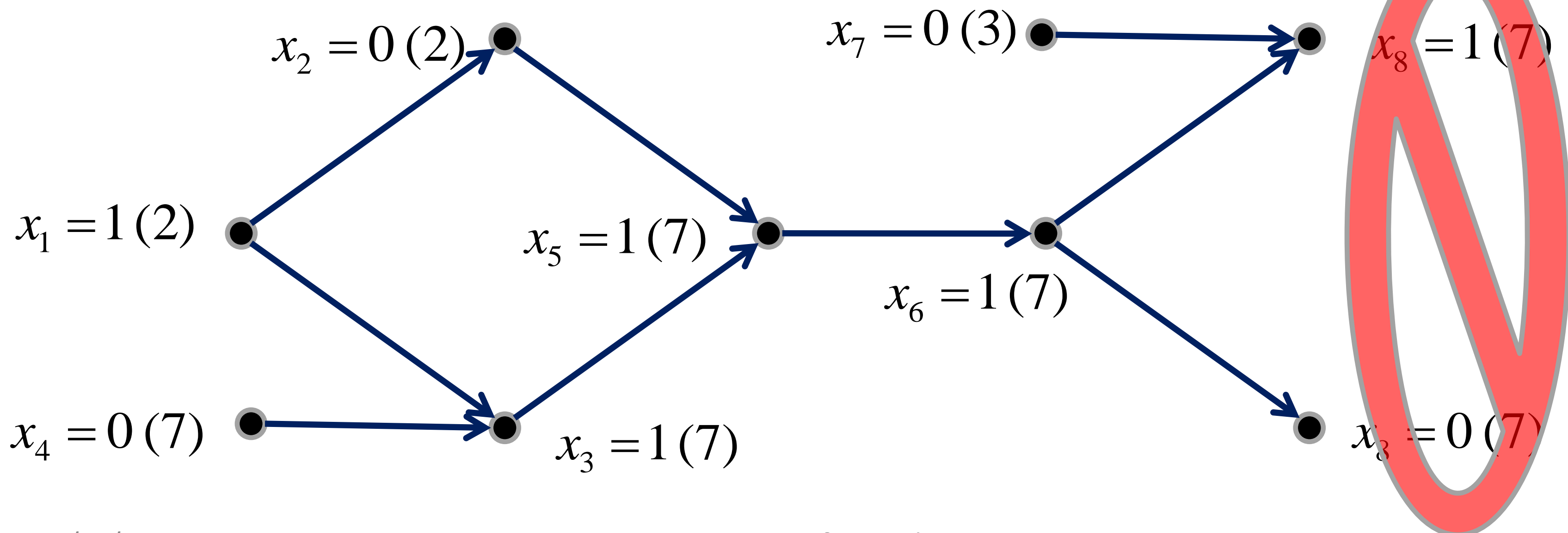


Microsoft Research

Implication Graphs

$\neg x_1 \vee \neg x_2$ $\quad$ $\neg x_1 \vee x_3 \vee x_4$ $\quad$ $x_2 \vee \neg x_3 \vee x_5$

$\neg x_5 \vee \neg x_6$ $\quad$ $\neg x_6 \vee x_7 \vee x_8$ $\quad$ $\neg x_6 \vee \neg x_8$



Clause Learning

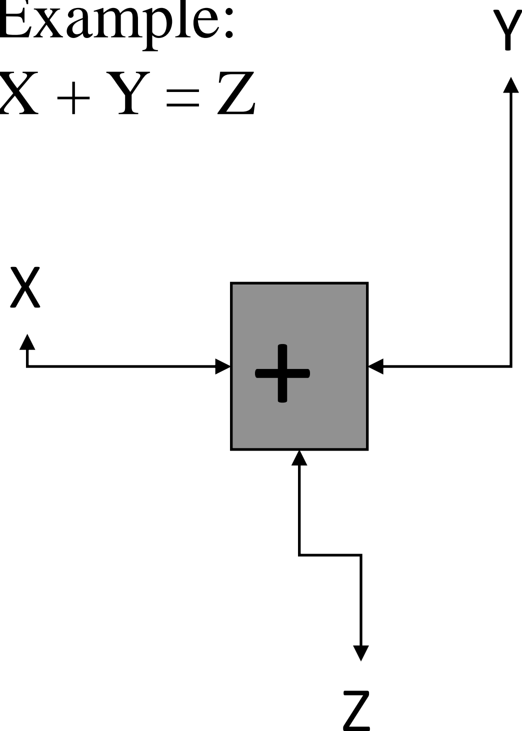
- Creating no-good -> (ordered) resolution
  - "Assertive", allows the refutation of one past hypothesis.
- Possibility to clean memory by forgetting some of them
  - Here again, activity-based: score increased when involved in propagation
  - Clauses with the lowest scores can be forgotten
  - Solvers that are "memory-opportunistic": if some memory is free let's use it, otherwise we are still complete.

COMPLETE SEARCH IN CP

Constraint Propagation

Example:

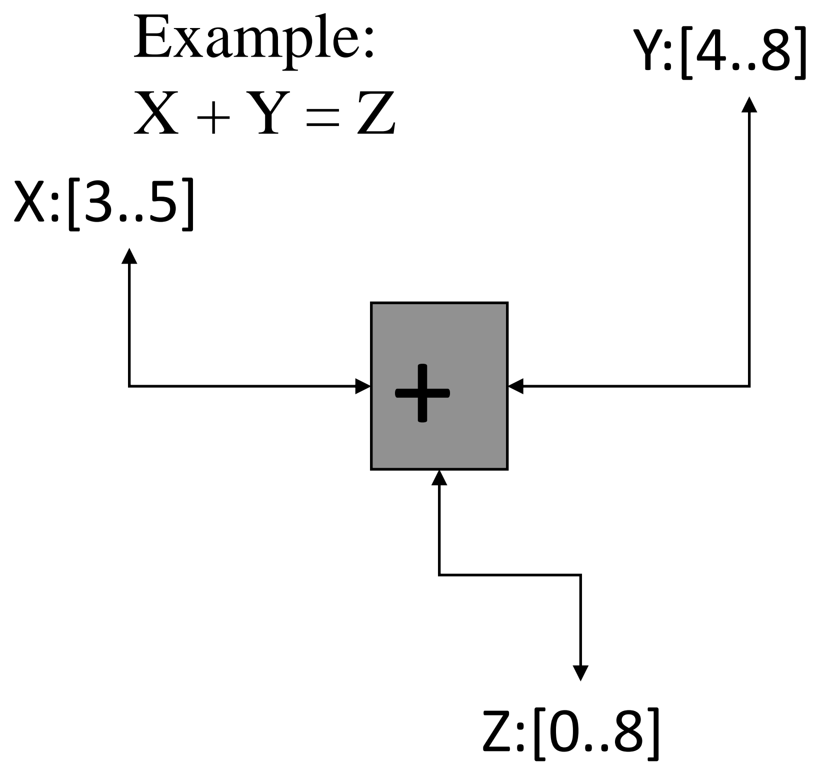
$X + Y = Z$



X=[lb .. ub]
Y=[lb .. ub]
Z=[lb .. ub]

Z <= (X.ub + Y.ub)
Z >= (X.lb + Y.lb)
X <= (Z.ub - Y.lb)
X >= (Z.lb - Y.ub)
Y >= (Z.lb - X.ub)
Y <= (Z.ub - X.lb)

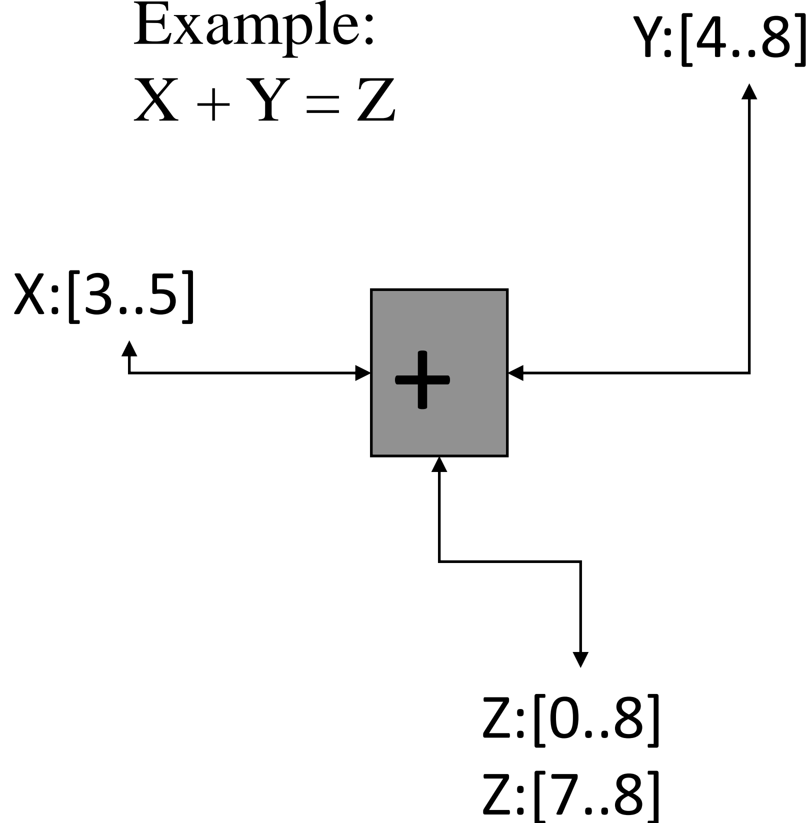
Constraint Propagation



X=[lb .. ub]
Y=[lb .. ub]
Z=[lb .. ub]

Z <= (X.ub + Y.ub)
Z >= (X.lb + Y.lb)
X <= (Z.ub - Y.lb)
X >= (Z.lb - Y.ub)
Y >= (Z.lb - X.ub)
Y <= (Z.ub - X.lb)

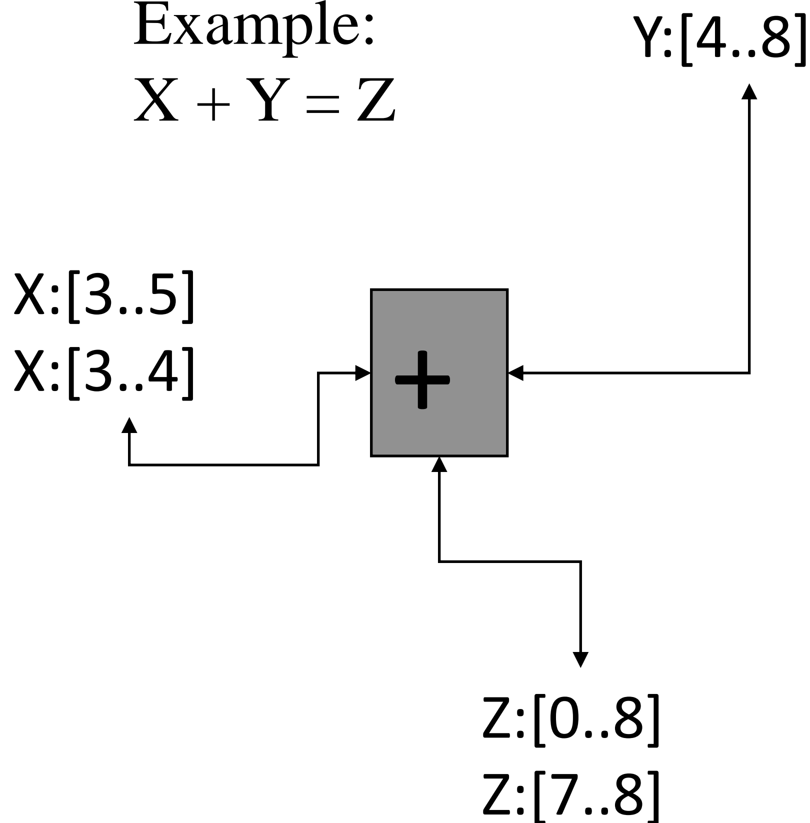
Constraint Propagation



X=[lb .. ub]
Y=[lb .. ub]
Z=[lb .. ub]

Z <= (X.ub + Y.ub)
Z >= (X.lb + Y.lb)
X <= (Z.ub - Y.lb)
X >= (Z.lb - Y.ub)
Y >= (Z.lb - X.ub)
Y <= (Z.ub - X.lb)

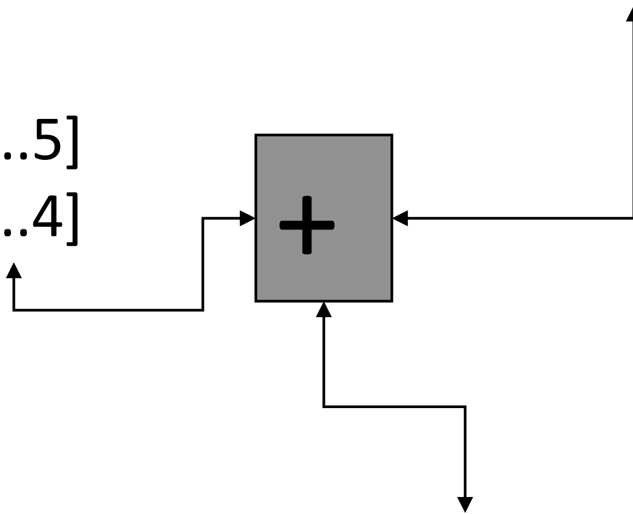
Constraint Propagation



X=[lb .. ub]
Y=[lb .. ub]
Z=[lb .. ub]

Z <= (X.ub + Y.ub)
Z >= (X.lb + Y.lb)
X <= (Z.ub - Y.lb)
X >= (Z.lb - Y.ub)
Y >= (Z.lb - X.ub)
Y <= (Z.ub - X.lb)

Constraint Propagation

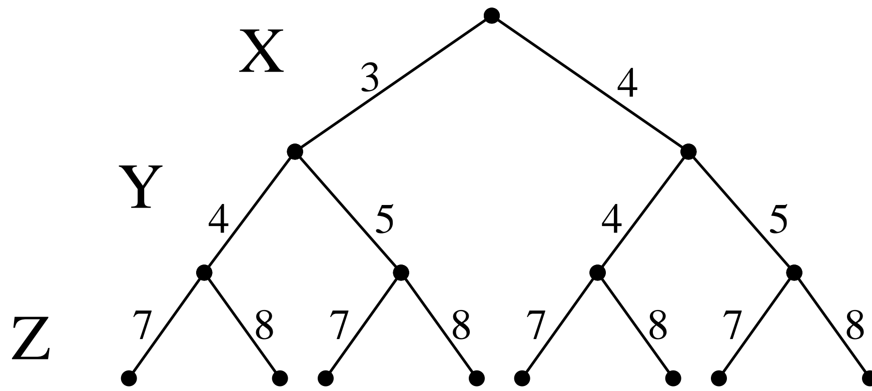


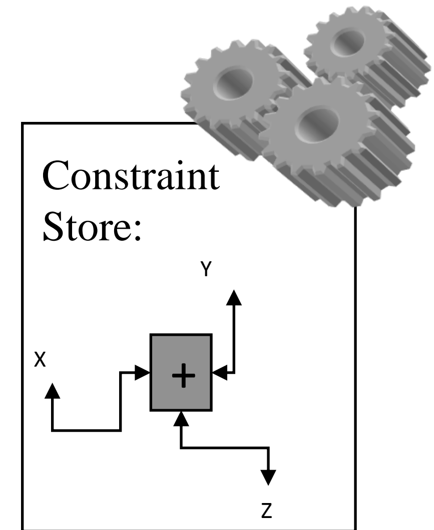
X=[lb .. ub]
Y=[lb .. ub]
Z=[lb .. ub]

Z <= (X.ub + Y.ub)
Z >= (X.lb + Y.lb)
X <= (Z.ub - Y.lb)
X >= (Z.lb - Y.ub)
Y >= (Z.lb - X.ub)
Y <= (Z.ub - X.lb)

Fix point
Initial space: 3*5*9
Final space: 2*2*2

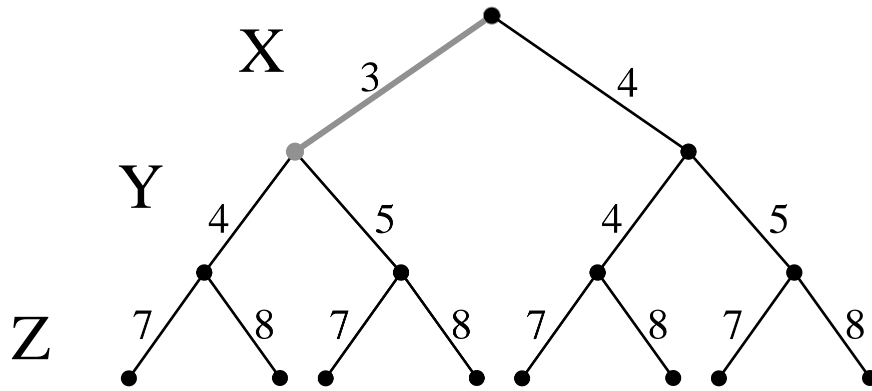
Constrained Search



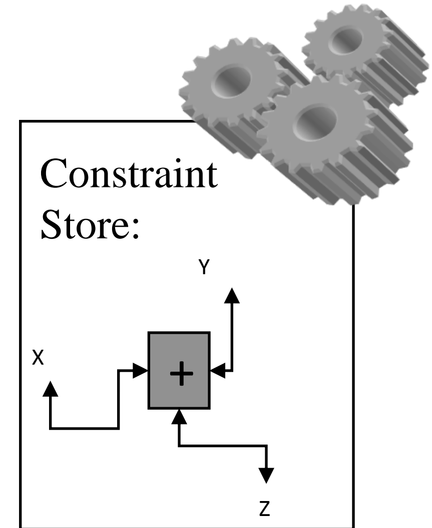


Constrained Search

Example: $X + Y = Z$



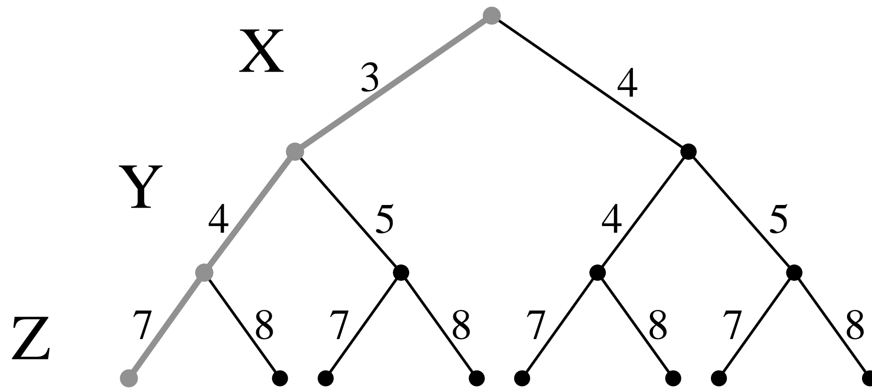
X = 3

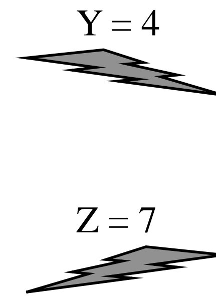


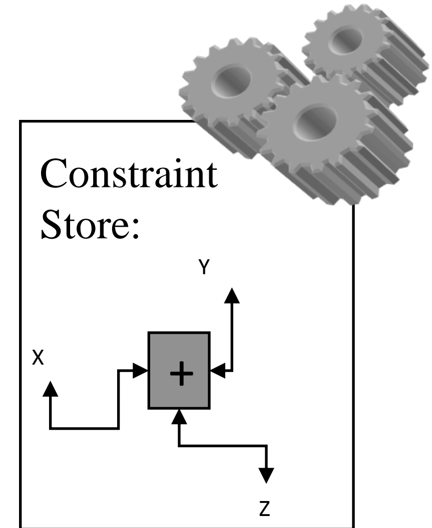
Microsoft Research

Constrained Search

Example: X + Y = Z

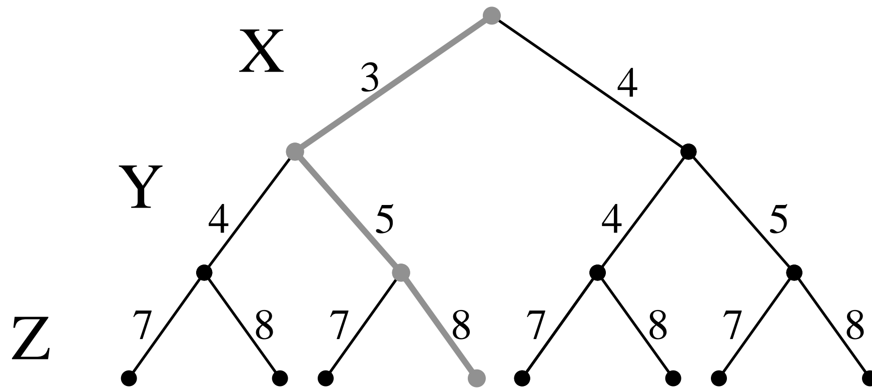


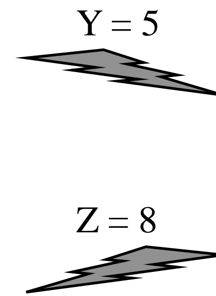


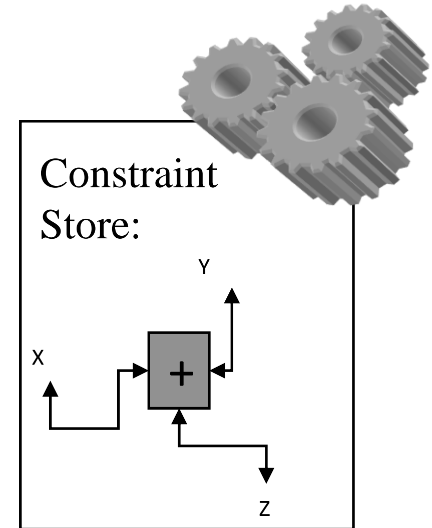


Constrained Search

Example: X + Y = Z

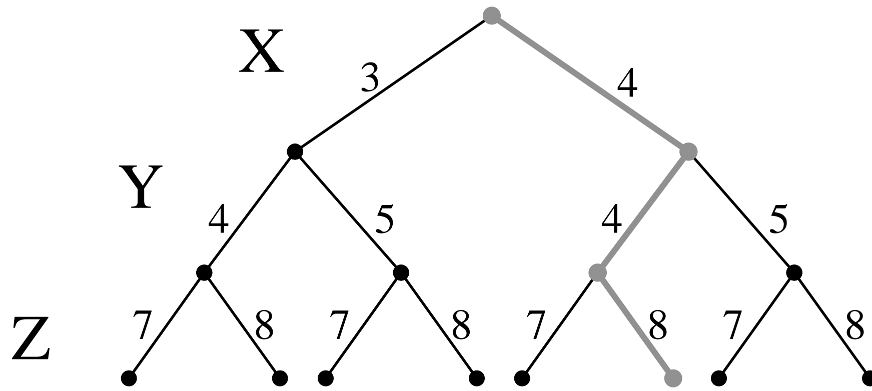






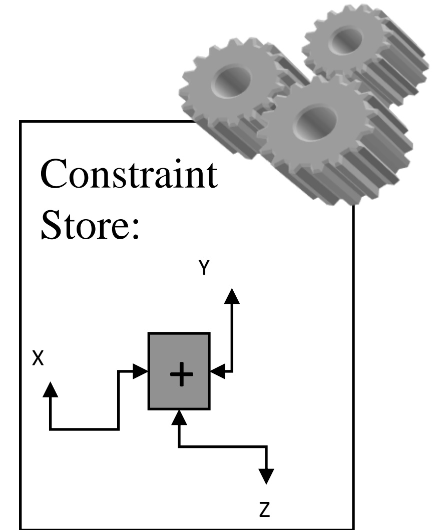
Constrained Search

Example: $X + Y = Z$



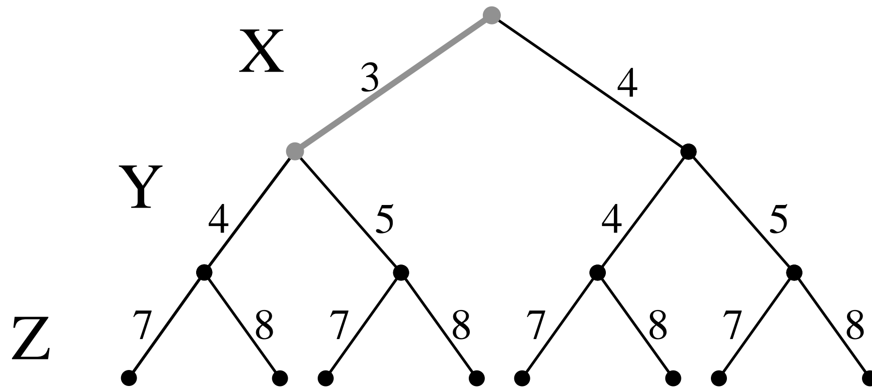
$X = 4$

$Y = 4$
$Z = 8$

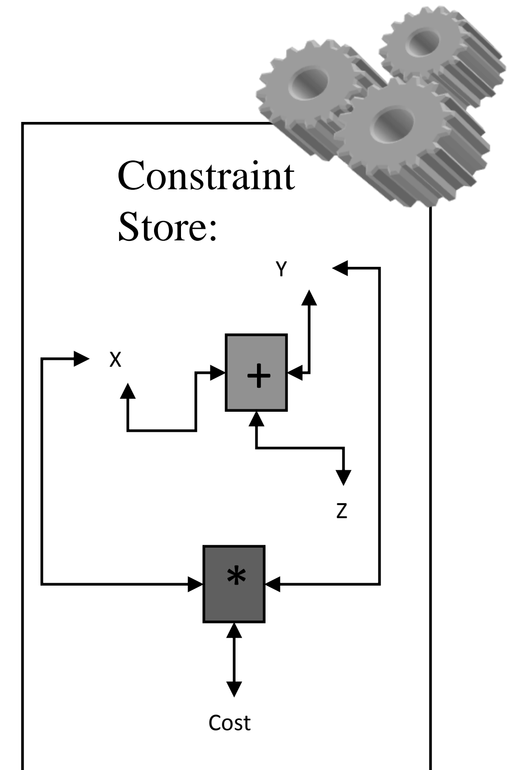


Constrained Optimization

Example: X + Y = Z, Min(X*Y)



X = 3

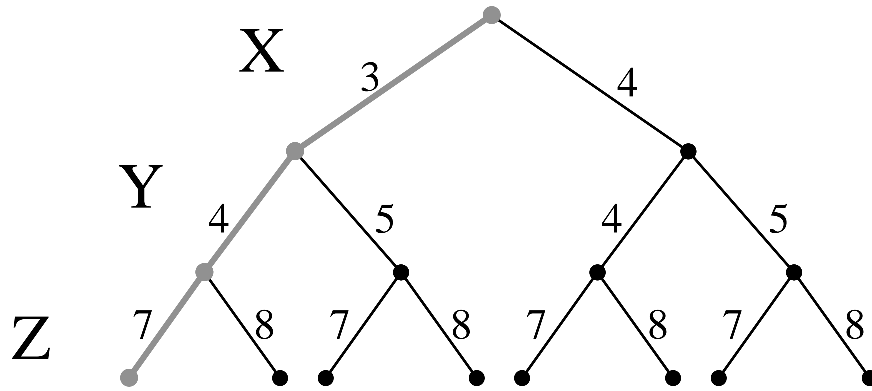


Goal: min(x*y)

Cost = [12 .. 15]

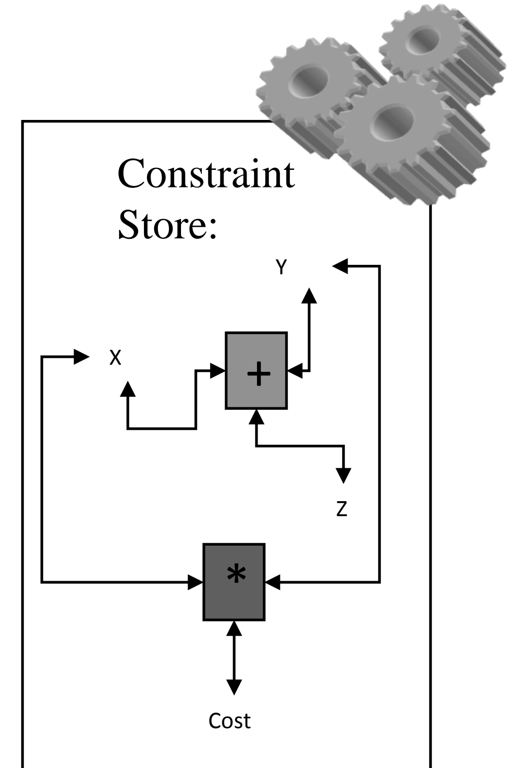
Constrained Optimization

Example: X + Y = Z, Min(X*Y)



Y = 4

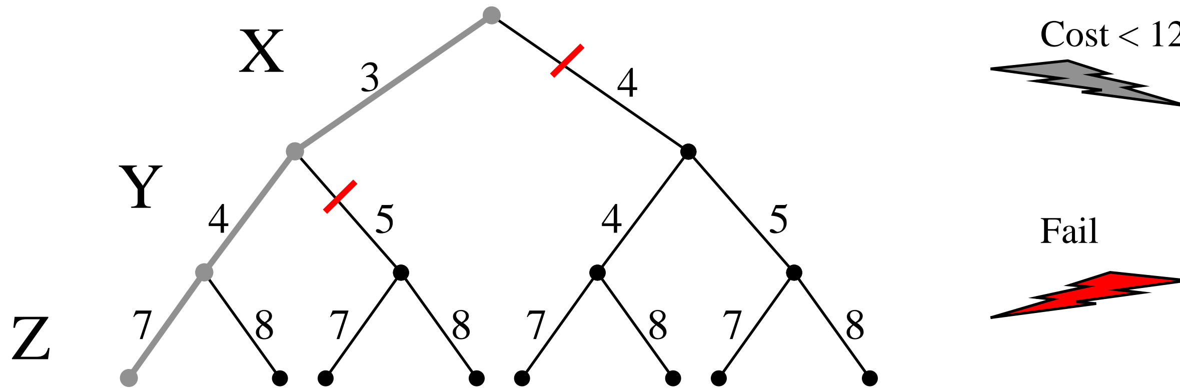
Z = 7
Cost = 12

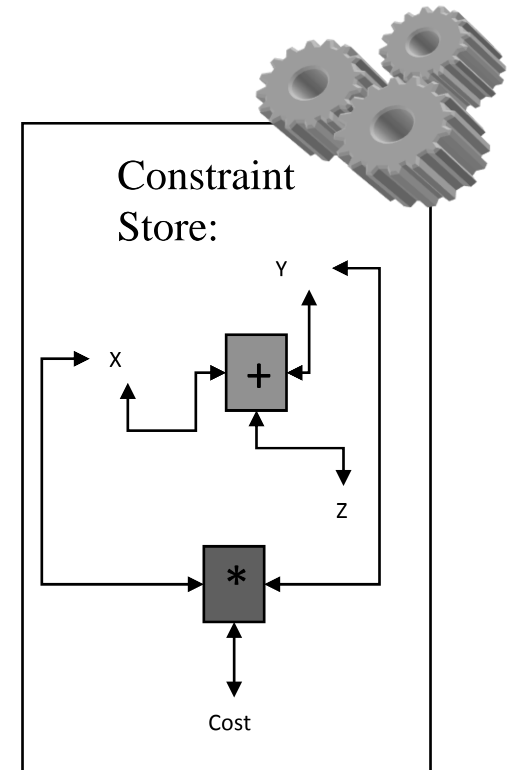


Goal: min(x*y)

Constrained Optimization

Example: $X + Y = Z$, $Min(X^*Y)$





Goal: min(x*y)

PROGRAMMING IN SAT & CP

Programming in SAT

- Tuning of SAT solvers
  - Not essential: learning and VSID are usually able to grasp and solve the problem after a few restarts

Programming in CP

- Search strategy
  - Variable ordering
  - Value ordering

Use the understanding of the domain to encode an efficient strategy…

… expertise required…

…time consuming…

(Real) Example: Microsoft Consulting Services

- Capacity planning problem, (simplified) definition
  - 850 IT consultants,
    - All over the U.S.
    - Different skills, availability, etc.
  - Incoming demands
    - 200 /month
    - Requiring different set of skills, at specific dates etc.
  - Allocate {consultants} to each demand while,
    - Minimizing travelling cost,
    - Respecting annual utilization target per/consultant,
    - Providing a fair repartition of travel between consultants,
    - Taking into account training (skill refreshing),
    - Taking into account consultants/customers preferences,
    - Taking into account expected demands…
    - Etc.

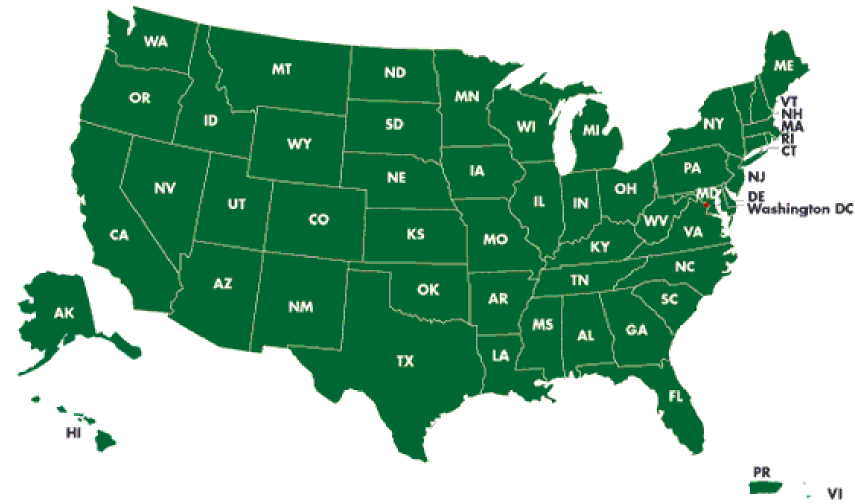
(Real) Example: Microsoft Consulting Services

Experiment:

- Batch allocation of 50 demands over 850 possible consulting resources
- Out-of-the-box Disolver: 1.9hours
  - Default “fail-first principle” heuristic

(Real) Example: Microsoft Consulting Services

- Disolver provided with problem structure: 17.58s
  - Problem structure:
    - Hvar: demand
    - Hval: “closest consultant first”

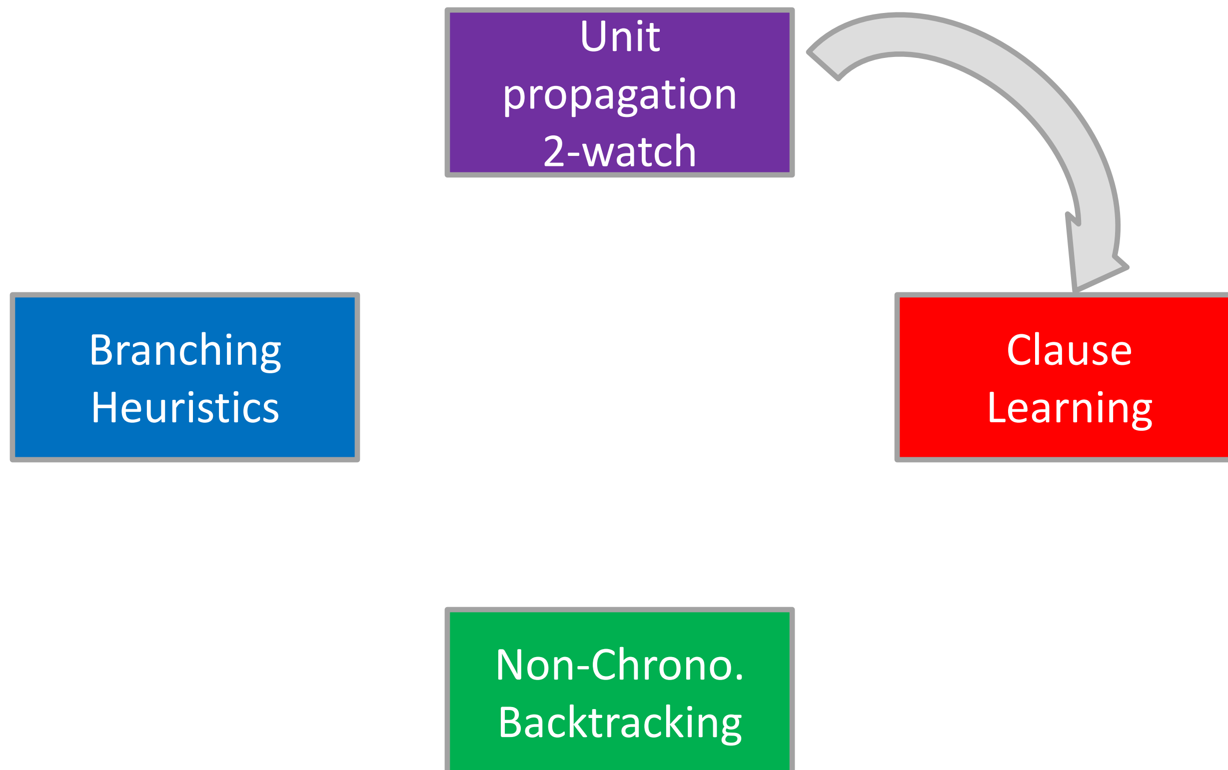


LESSONS LEARNED, INTEGRATION?

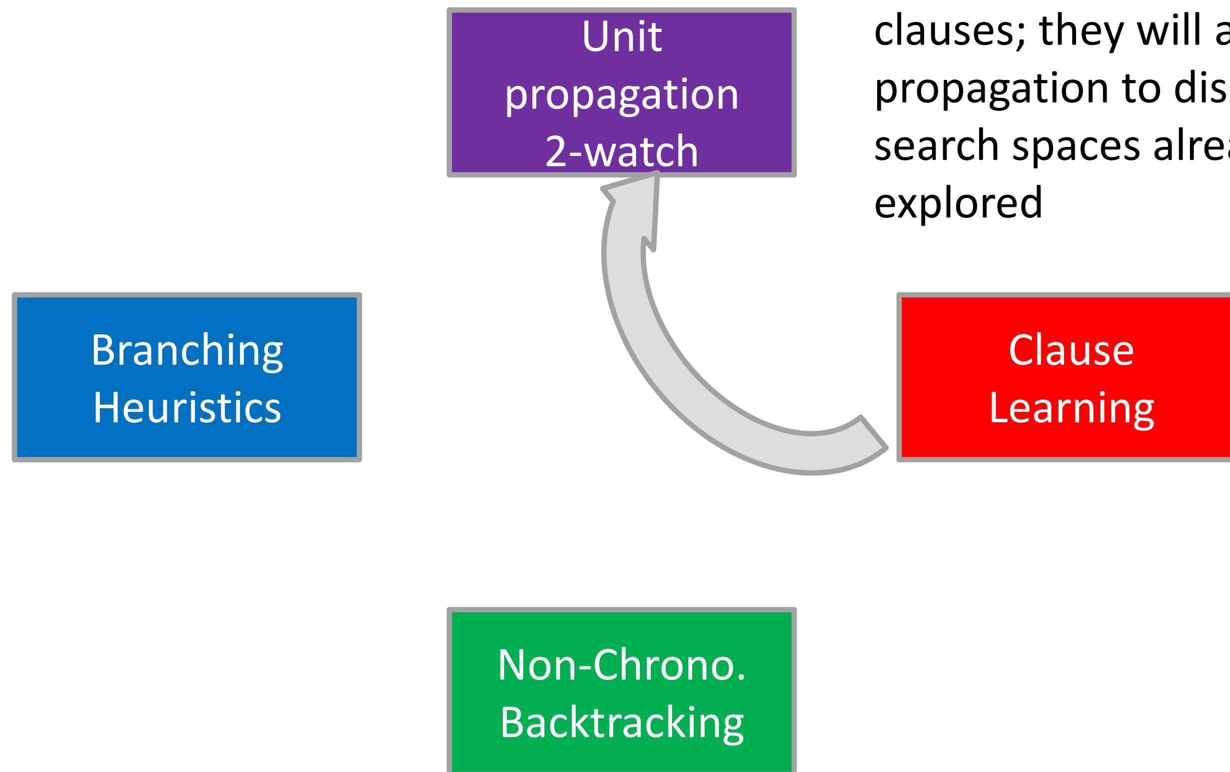
The magic of DPLL / zChaff –

- or *how to get a combination right*

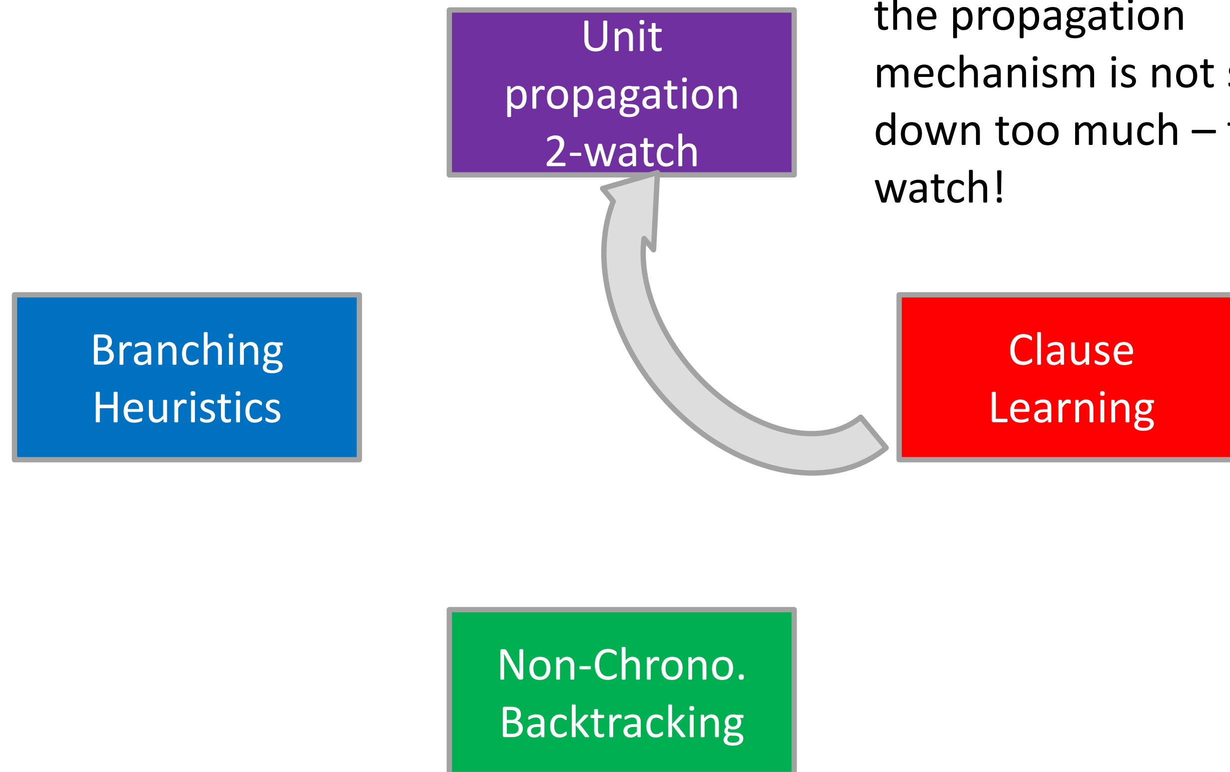
The implication graph, obtained cheaply from the clause data-structures, captures the reasoning done by BCP and allows clause-learning



Learnt no-goods are naturally encoded as clauses; they will allow propagation to discard search spaces already explored



Note that learning extra clauses is only feasible if the propagation mechanism is not slowed-down too much – thanks, 2-watch!



Unit propagation 2-watch

Branching Heuristics

Clause Learning

Non-Chrono. Backtracking

How to undo a conflict is determined by the learnt clause

Unit propagation 2-watch

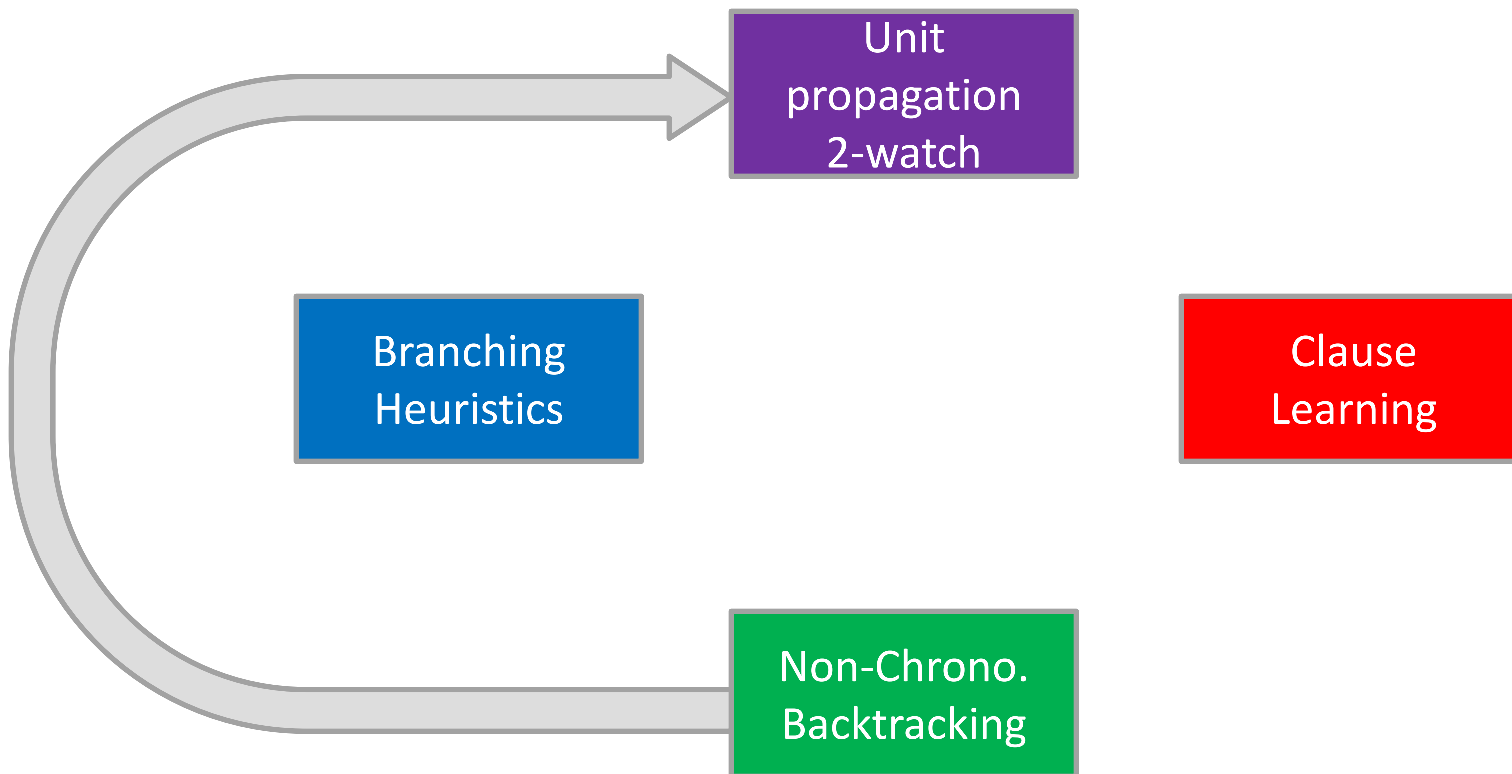
Branching Heuristics

Clause Learning

Non-Chrono. Backtracking

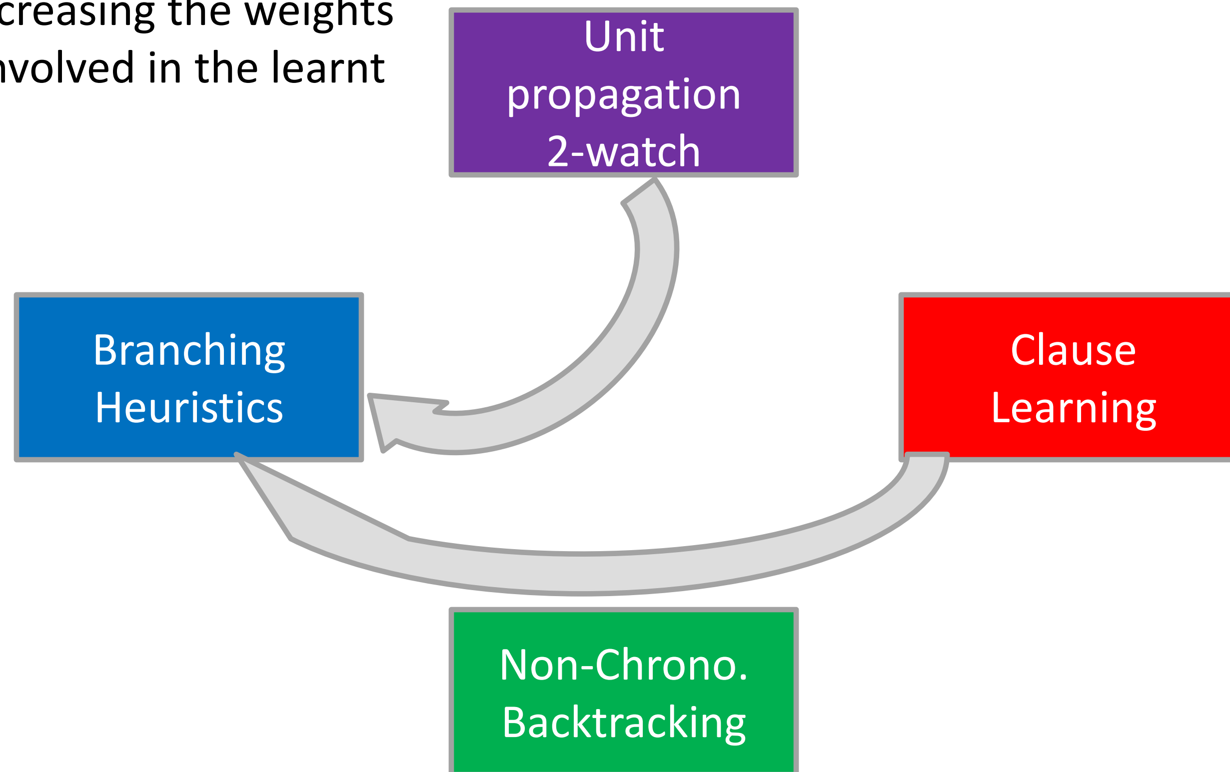
The level at which to backtrack is directly deduced from learnt clause. Plus we can pick an *asserting* clause

Note that the data-structures used for propagation are backtrack-friendly: other than undoing some decision we don't need to re-compute anything



The branching heuristics is based on observing the activity of the deduction mechanisms (propagation)

This is done by increasing the weights of the variables involved in the learnt clauses



Microsoft Research

Now if we add restarts to the picture, they also fit pretty well:

Learning helps reducing the time lost when “restarting from scratch” as we may keep deductions from previous runs

Unit propagation 2-watch

Branching Heuristics

Clause Learning

Non-Chrono. Backtracking

Restarts

Microsoft Research

Now if we add restarts to the picture, they also fit pretty well:

Plus when we restart we can benefit from the scoring

Unit propagation 2-watch

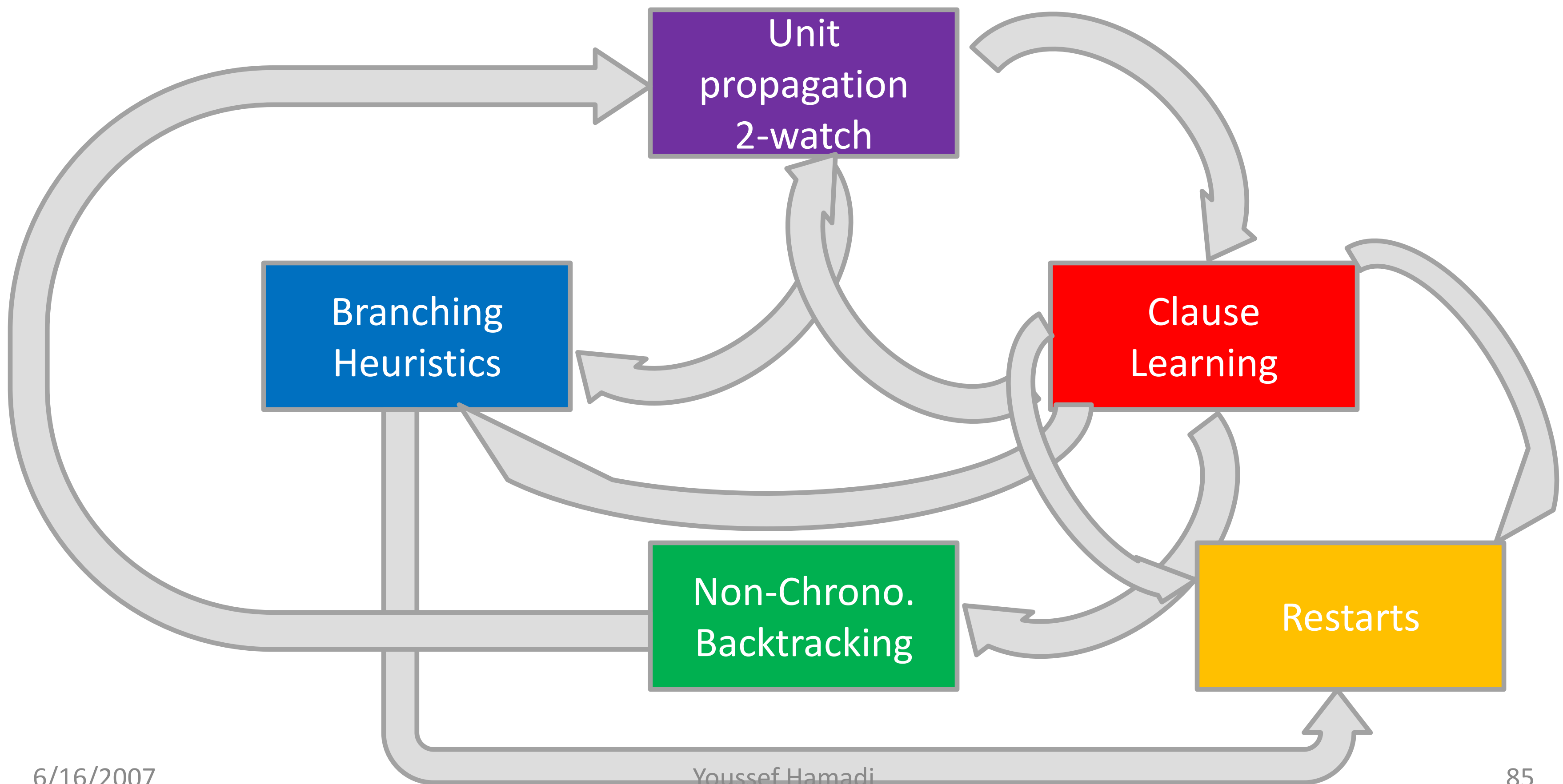
Branching Heuristics

Clause Learning

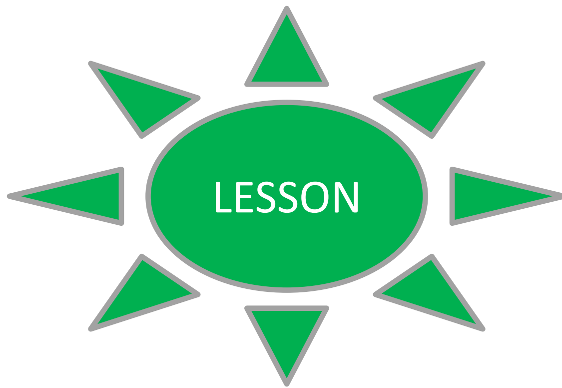
Non-Chrono. Backtracking

Restarts

A modern DPLL solver is not exactly a *loosely coupled* piece of software, is it? ☺



Lessons to learn from the architecture of SAT solvers



Research Problem

- Implementers of CP solvers like to think in terms of “open system”, “decoupling of components”, “user-extendibility”
- The interaction between branching, propagation and other components is weak
- Couldn’t CP solvers go the SAT way?

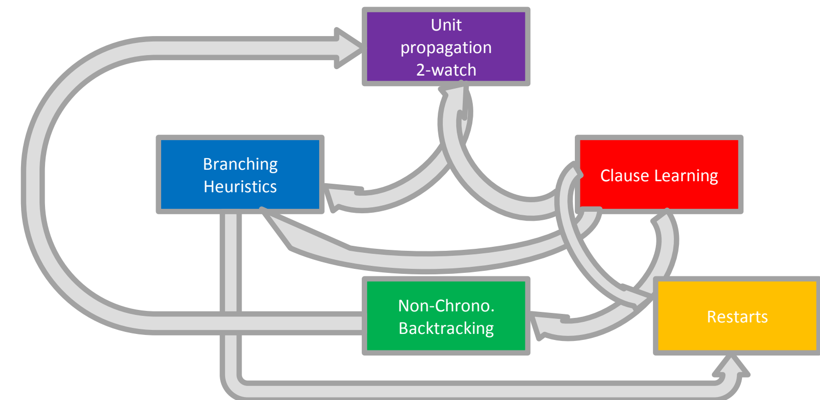
PERSPECTIVES

Problems of SAT & CP

- SAT
  - Expressivity.. CNF encoding too low level for some new domains
- CP
  - Required expertise, tuning…

SAT

- Satisfiability Modulo Theory
  - Modern DPLL solver



  - Extended with new theories, e.g., linear arithmetic

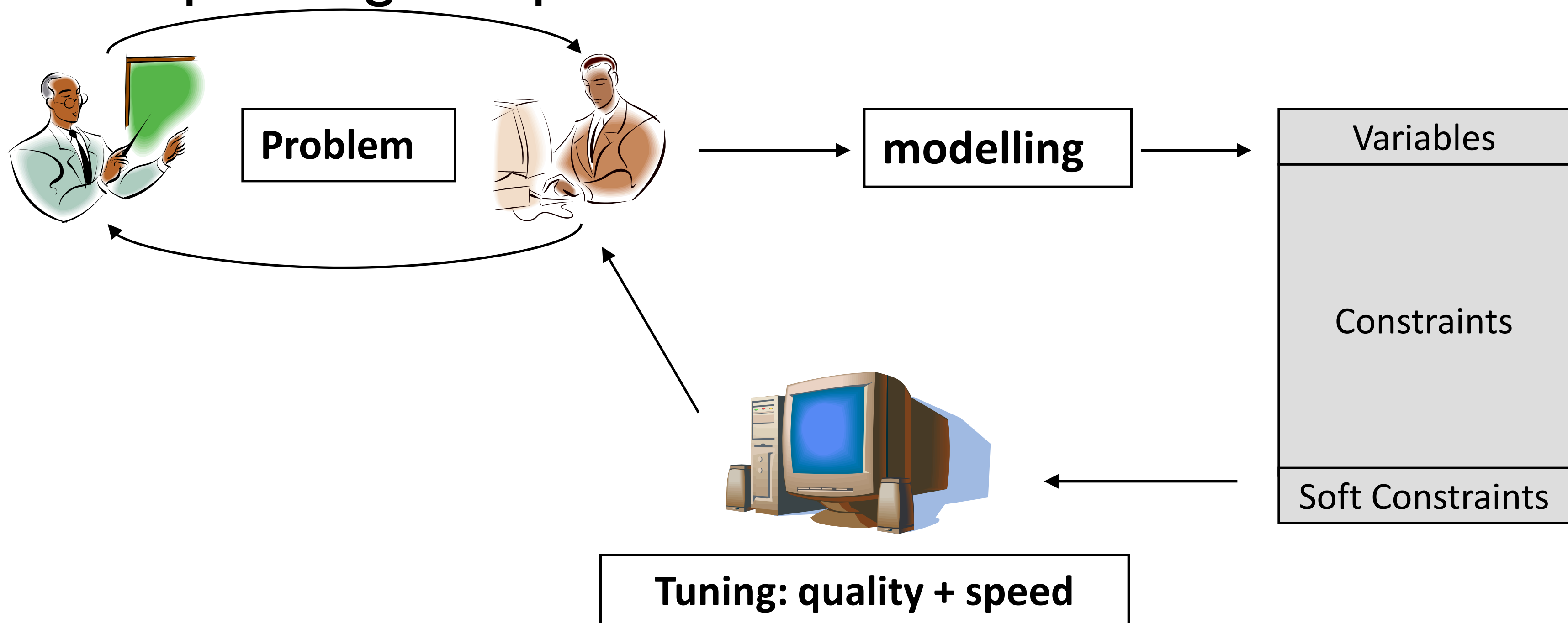
➔Compact representation,

➔DPLL efficiency

The Problem of CP Usability

Usability Problem of Constraint Programming

- Exploiting the problem structure:



Microsoft Research

Performance Prediction and Automated Tuning

Classical approach for tuning

Tuning

1. Find a large set of representative instances I
2. Test various parameter settings in order to find p^* the best setting
3. Use p^* on all instances with the real application

Problems:

- Must be able to characterize the problem to correctly generate I
- Time consuming

Instance aware Problem Solver

1. Learn a single function that maps instance features and parameter settings to runtime
   - Instance features:
     - Quickly computed
       - Basic statistics – size, etc.
       - Graph-based features – degree, etc.
       - Local search probes - #local minima, etc.
       - DPLL-based measures – Knuth estimator, etc.
       - Etc.
2. Given a new instance
   1. Compute its features
   2. Search for the parameter setting p* that minimizes predicted runtime for these features

Evaluation

- Prediction of two local search for SAT
  - Novelty+ (winner SAT 04 competition)
  - SAPS
- Problems
  - Hard random
  - Quasi-group completion problem
  - Mixed

Experimental results

Instance-aware solver, average speed-ups:

Set	Algo	Gross corr	RMSE	Corr per inst.	best fixed params	s_{bpi}	s_{wpi}	s_{def}	s_{fixed}
SAT04	Nov	0.90	0.78	0.86	50	0.62	275.42	0.89	0.89
QWH	Nov	0.98	0.58	0.69	10	0.81	457.09	**177.83**	0.91
Mixed	Nov	0.95	0.8	0.81	40	0.74	363.08	**13.18**	**10.72**
SAT04	SAPS	0.95	0.67	0.52	$\langle 1.3, 0 \rangle$	0.56	10.72	**2.00**	**1.07**
QWH	SAPS	0.98	0.40	0.39	$\langle 1.2, .1 \rangle$	0.65	6.03	**2.00**	0.93
Mixed	SAPS	0.91	0.60	0.65	$\langle 1.2, 0.2 \rangle$	0.46	17.78	**1.91**	0.93

Experimental results

- Two orders of magnitude better than the default for Novelty+ on Quasi-groups
- One order of magnitude better than the best-fixed for Novelty+ on Mixed
- Factor 2 better than the best-fixed for SAPS on Mixed
- Never very far than optimal settings for the two algorithms

CONCLUSION

SAT vs CP

SAT solving

- Modelling
  - Conjunctive Normal form formula.
- Relaxation
  - Fast unit clause propagation.
- Search
  - Built-in branch-and-prune.

Constraint Programming

- Modelling
  - Arithmetic, logical, symbolic, global constraints.
- Relaxation
  - Constraint propagation.
- Search
  - Built-in branch-and-prune / bound, support for easy integration of user supplied heuristics.

SAT vs. CP

	SAT	CP
Methodology	Low-level Black box Automatic	High-level Glass box Parameterised
Applications	Focus on decision High specialisation	Focus on optimisation High versatility
Solver Design	Small Homogeneous	Large Open
Evolution	Bottom-up evolution Progress measurable	Top-down evolution Poor measurability

Conclusion

- Propositional Satisfiability,
  → If your problem is easy to "compile" in conjunctive normal form
  + efficient out-of-the-box-free DPLL solvers
  - encoding too complex for large instances
- Constraint Programming
  → If your problem has many facets, is big and can accommodate good sub-optimal solutions
  + optimisation
  - Heavy tuning nearly always required!

A first reference

Propositional Satisfiability and Constraint Programming: a Comparative Survey

L. Bordeaux, Y. Hamadi and L. Zhang

ACM Computing Surveys 2006

By no means the only ref. on SAT:

- forthcoming handbooks,
- survey by David Mitchell...