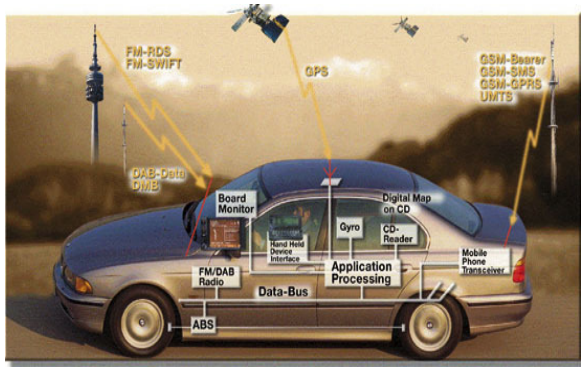
Verification using Satisfiability Checking, Predicate Abstraction, and Craig Interpolation

Himanshu Jain

THESIS ORAL TALK

Computer Systems are Pervasive



Computer Systems = Software + Hardware

Software/Hardware Bugs are common as well!

Building Reliable Computer Systems

Programming Languages

Testing

Static Analysis

Model Checking

Theorem Proving

Abstract Interpretation

Need for more automatic, scalable, precise verification tools

Decision Procedures

Reasoning engines in verification tools

Formula

Satisfiable

Unsatisfiable

Example: Boolean Satisfiability (SAT) Solvers

Decision Procedures++

Reasoning engines in verification tools

Formula

Satisfiable

Unsatisfiable
(Craig Interpolants)

Example: Boolean Satisfiability (SAT) Solvers

Contributions of this thesis

Fast Non-clausal Boolean Satisfiability (SAT) Solvers
- Based on Negation Normal Form (NNF), hpgraph, vpgraph
- DPLL based algorithm
- General Matings based algorithm

Verifying Hardware at Register Transfer Level (RTL)
- Model checking RTL Verilog using Predicate Abstraction and Counterexample Guided Abstraction Refinement Loop

Efficient Craig Interpolation for Subsets of Integer Linear Arithmetic
- Integer linear equations, congruences, disequations

Contributions of this thesis

Fast Non-clausal Boolean Satisfiability (SAT) Solvers
- Based on Negation Normal Form (NNF), hpgraph, vpgraph
- DPLL based algorithm
- General Matings based algorithm

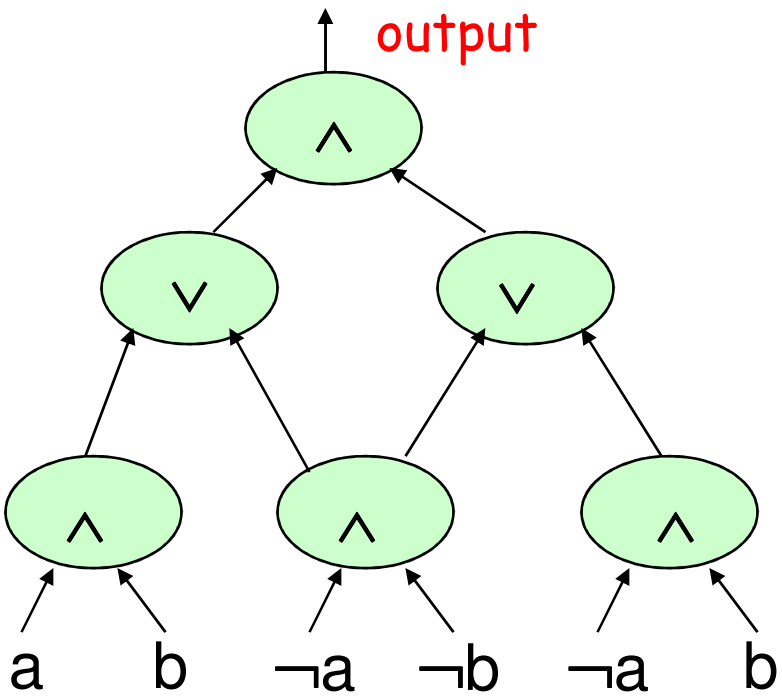
Verifying Hardware at Register Transfer Level (RTL)
- Model checking RTL Verilog using Predicate Abstraction and Counterexample Guided Abstraction Refinement Loop

Efficient Craig Interpolation for Subsets of Integer Linear Arithmetic
- Integer linear equations, congruences, disequations

Outline

- Motivation
- Fast Boolean Satisfiability (SAT) Solvers
  - DPLL Algorithm
  - Related work
  - Negation Normal Form, hpgraph, vpgraph
  - SAT solving using hpgraph, vpgraph
- Interpolation for Subsets of Integer Linear Arithmetic
  - An application
- Summary

Boolean Satisfiability

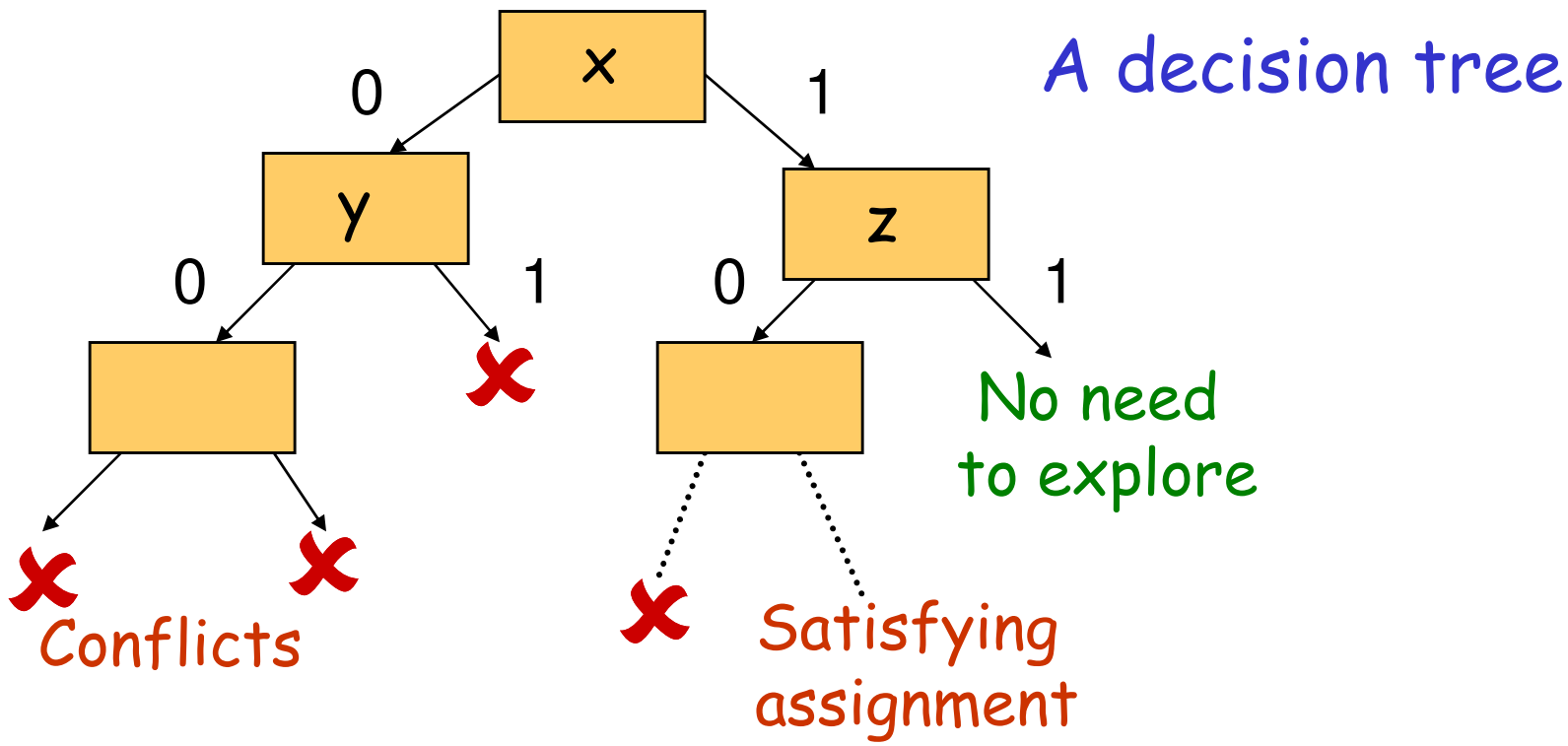


Boolean Circuit

- Applications in verification
  - Model checking
  - Equivalence checking
  - Theorem proving
  - Test generation
  - Static analysis
- Circuits in practice
  - Thousands of inputs
  - Millions of gates
  - Structure sharing

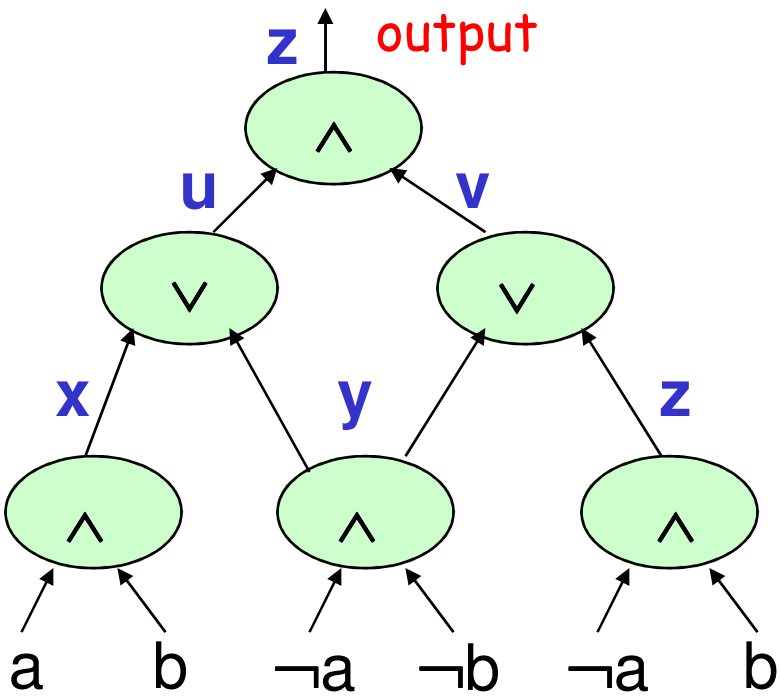
Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Input: Boolean Formula $\phi(x, y, z, \ldots)$
Find a satisfying assignment to ϕ by building a decision tree



Most solvers apply DPLL on CNF

Clausal solvers: RSAT, PicoSat, MiniSat, BerkMin, zChaff, GRASP, ...



Boolean Circuit

z

$\neg z \vee u$

$\neg z \vee v$

$z \vee \neg u \vee \neg v$

$\neg u \vee x \vee y$

$u \vee \neg x$

$u \vee \neg y$

.....

$z = u \wedge v$

$u = x \vee y$

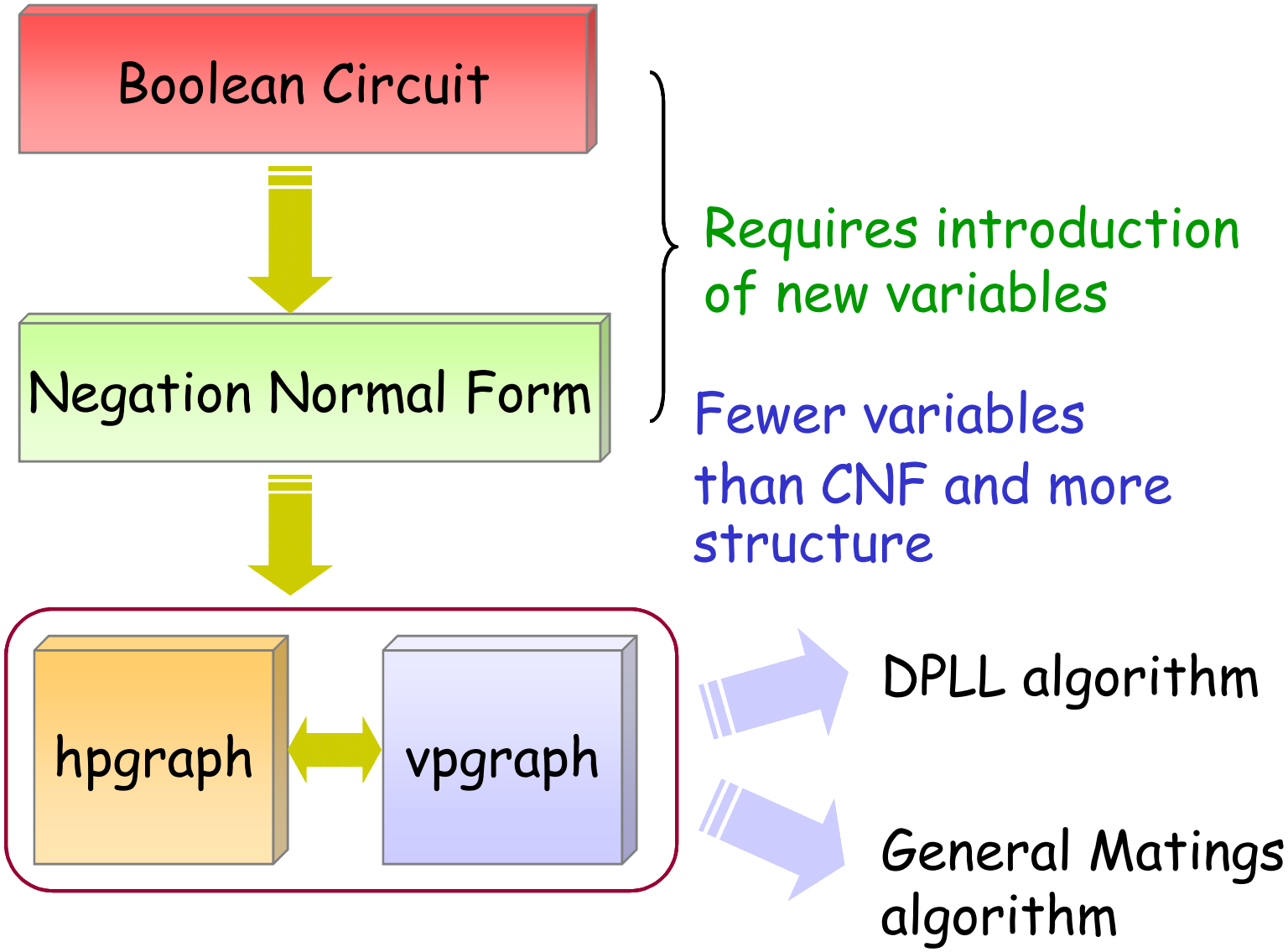
Conjunctive Normal Form (CNF)

New variables introduced is proportional to number of gates

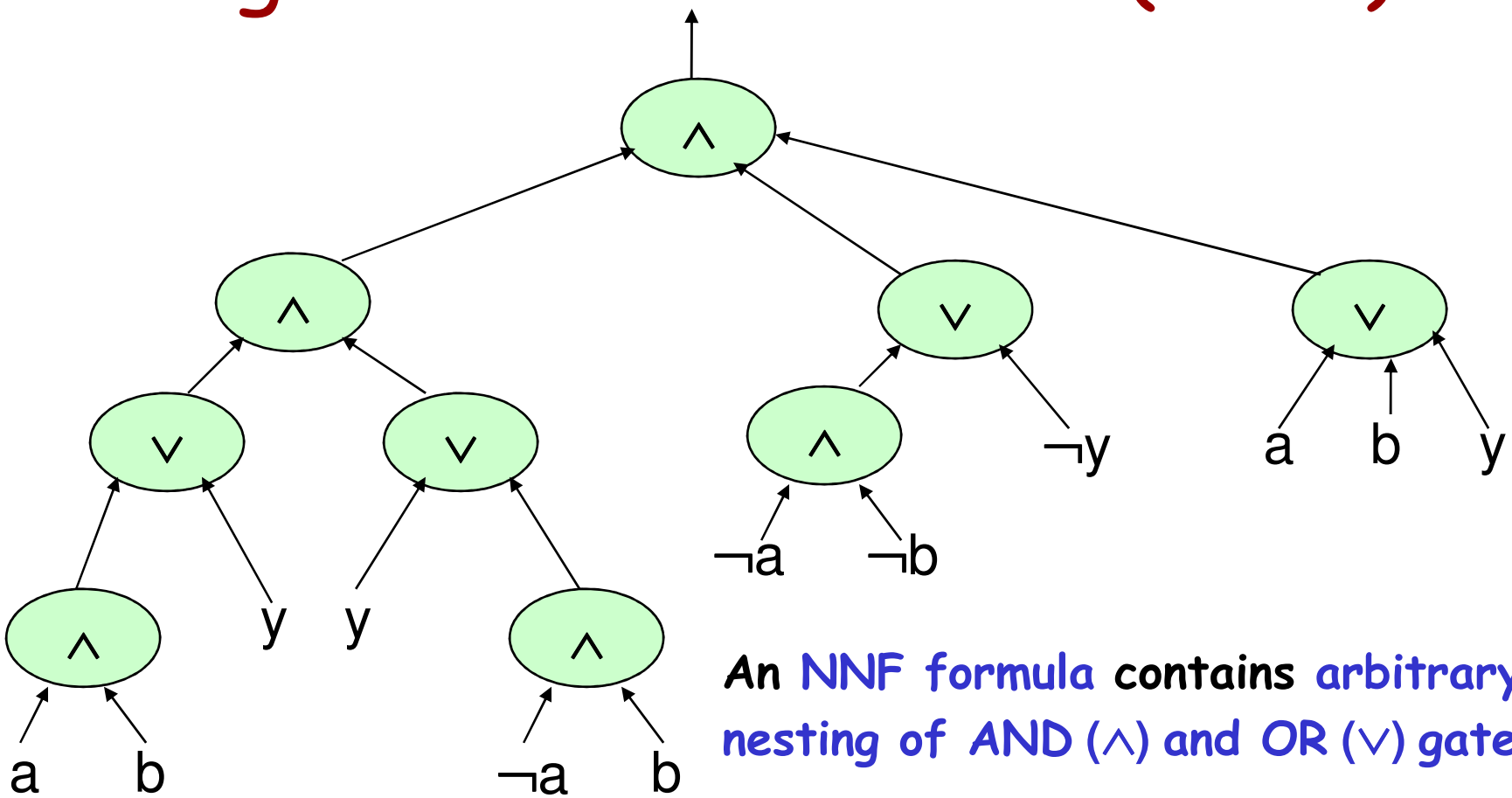
Current SAT solvers

- Davis-Putnam-Logemann-Loveland (DPLL) algorithm
  - Conversion of circuit to CNF
- Number of variables proportional to number of gates
  - Can be 10^3-10^4 times more than the number of inputs in a circuit
  - Slowdown due to large number of new variables (and clauses)
- Modern CNF solvers use pre-processing techniques
  - To reduce #variables and #clauses in CNF
  - But pre-processing has large memory requirements

Our SAT solving framework



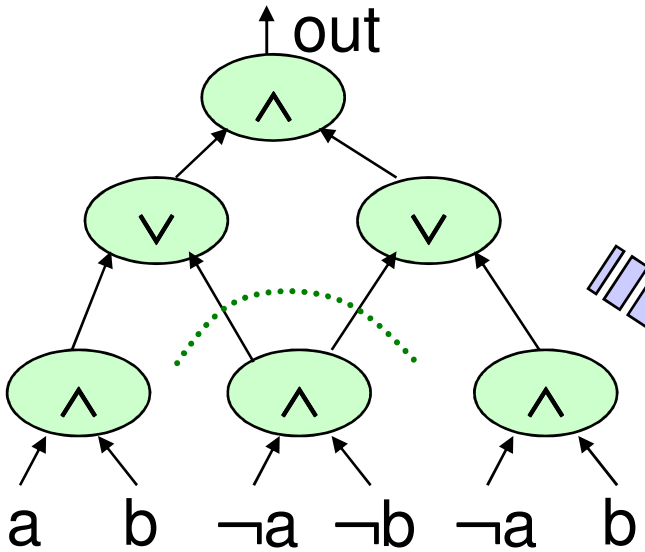
Negation Normal Form (NNF)



An NNF formula contains arbitrary nesting of AND (∧) and OR (∨) gates

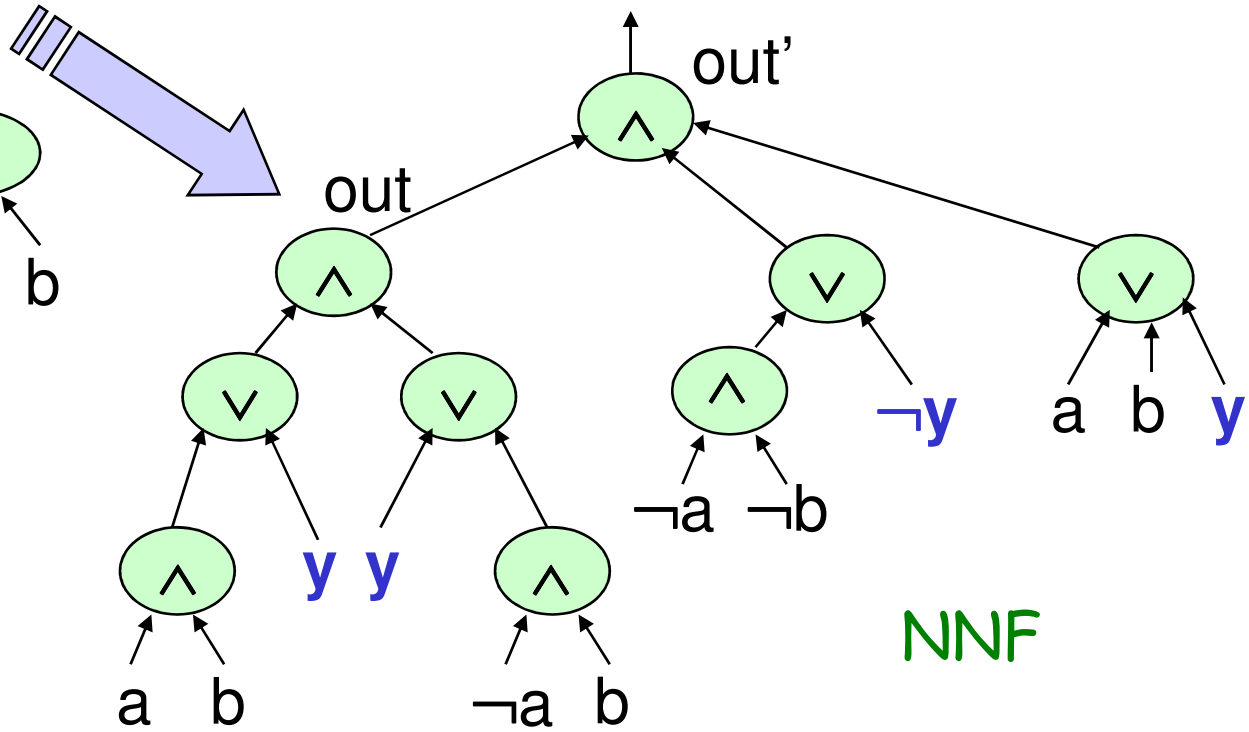
- Negations can appear at leaf level
- **No sharing** of sub formulas

How to obtain NNF from Boolean Circuits



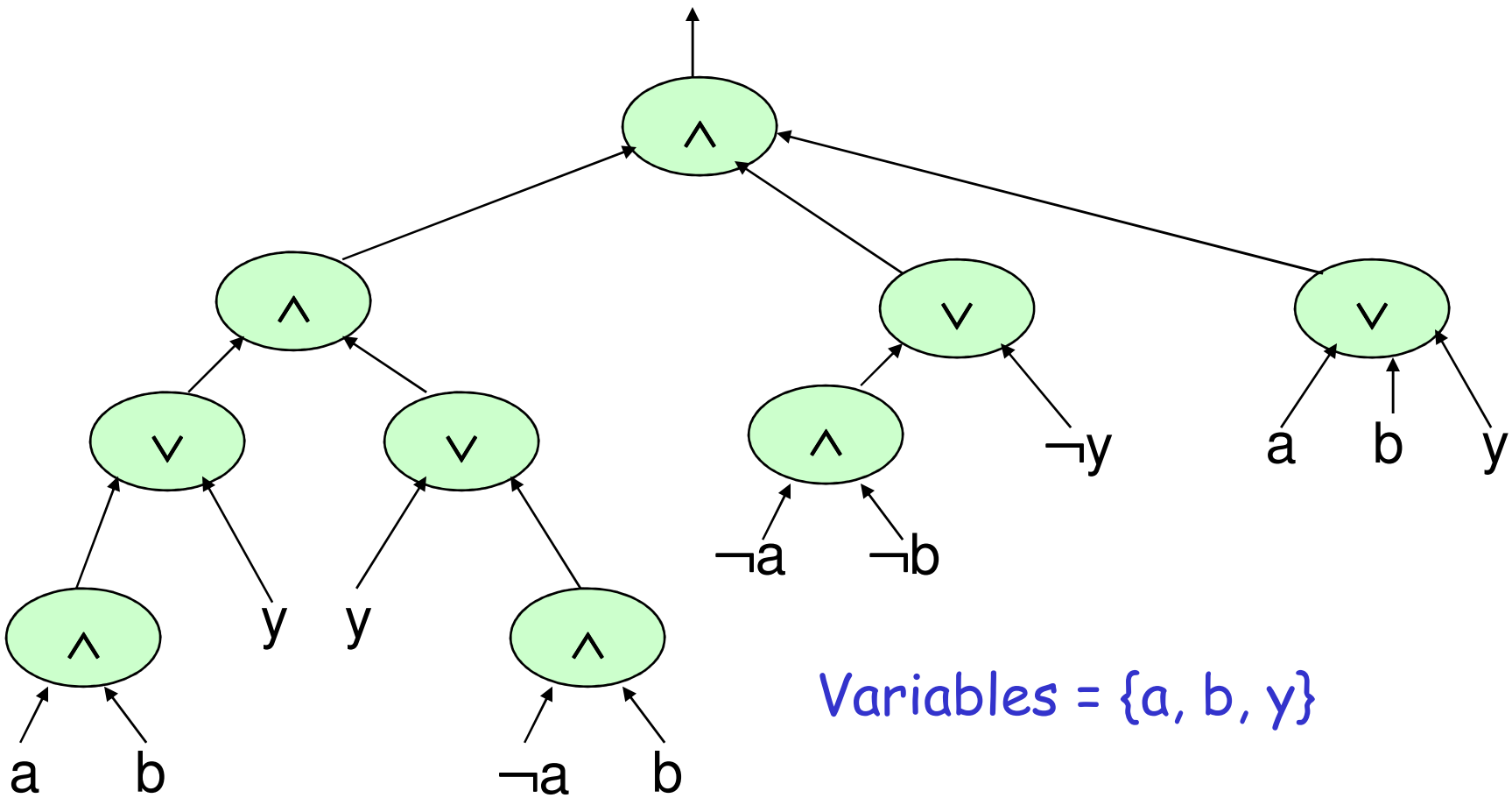
Boolean Circuit

Circuit from converted to NNF by introducing one new variable y to remove sharing



NNF

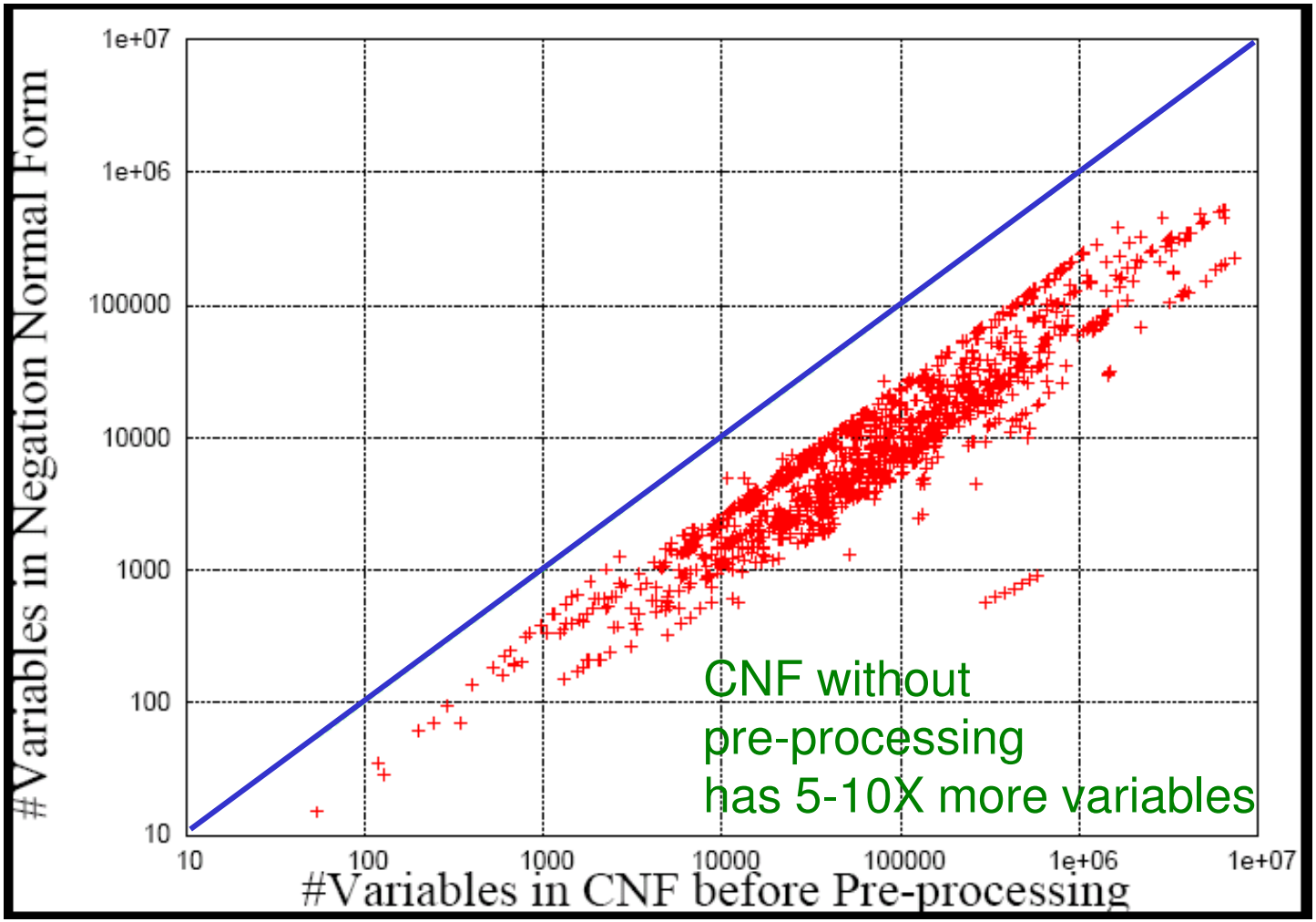
Satisfiability Checking of NNF Formulas



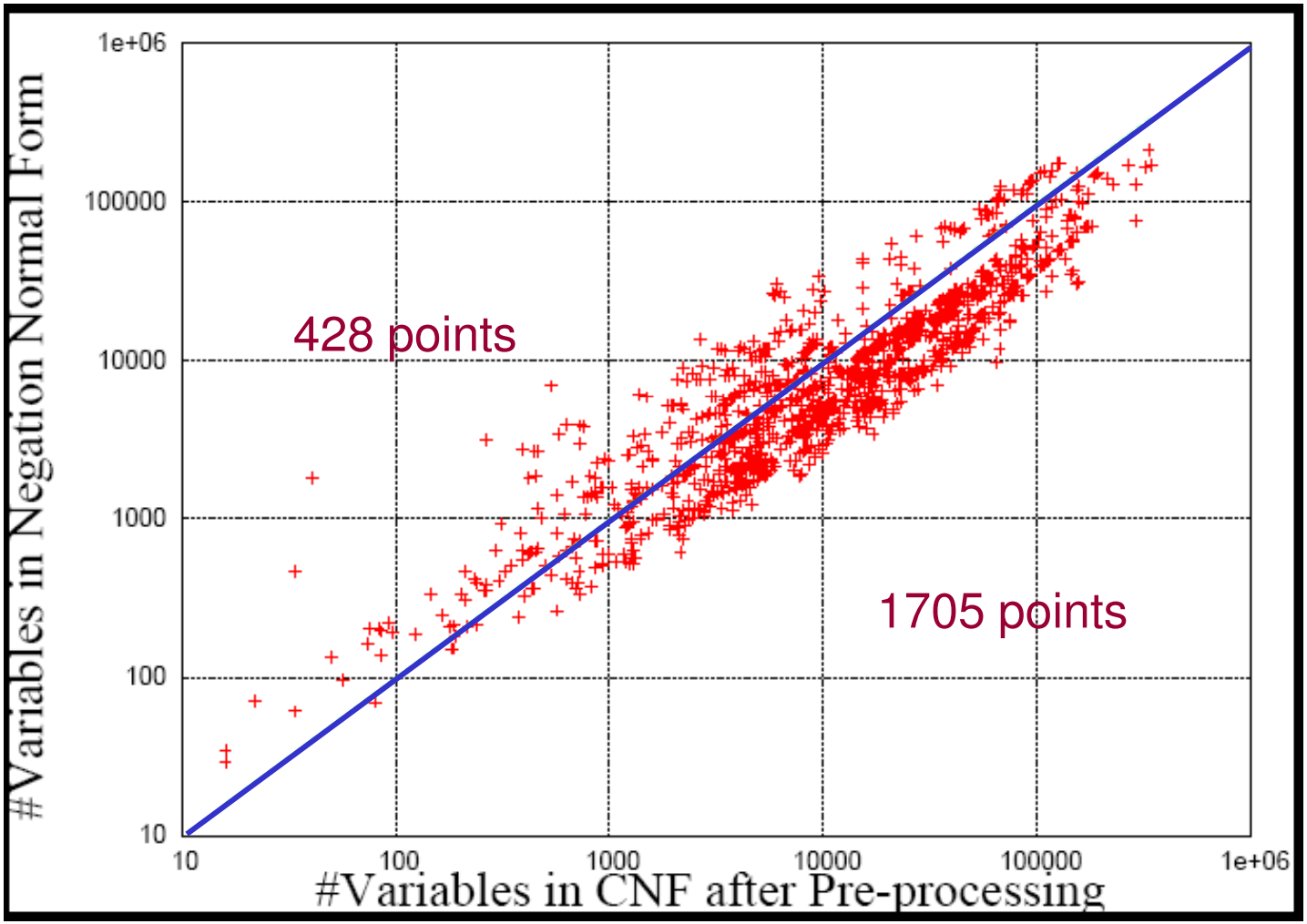
Variables = {a, b, y}

Our SAT algorithms will not add any more new variables

Why NNF: Number of variables

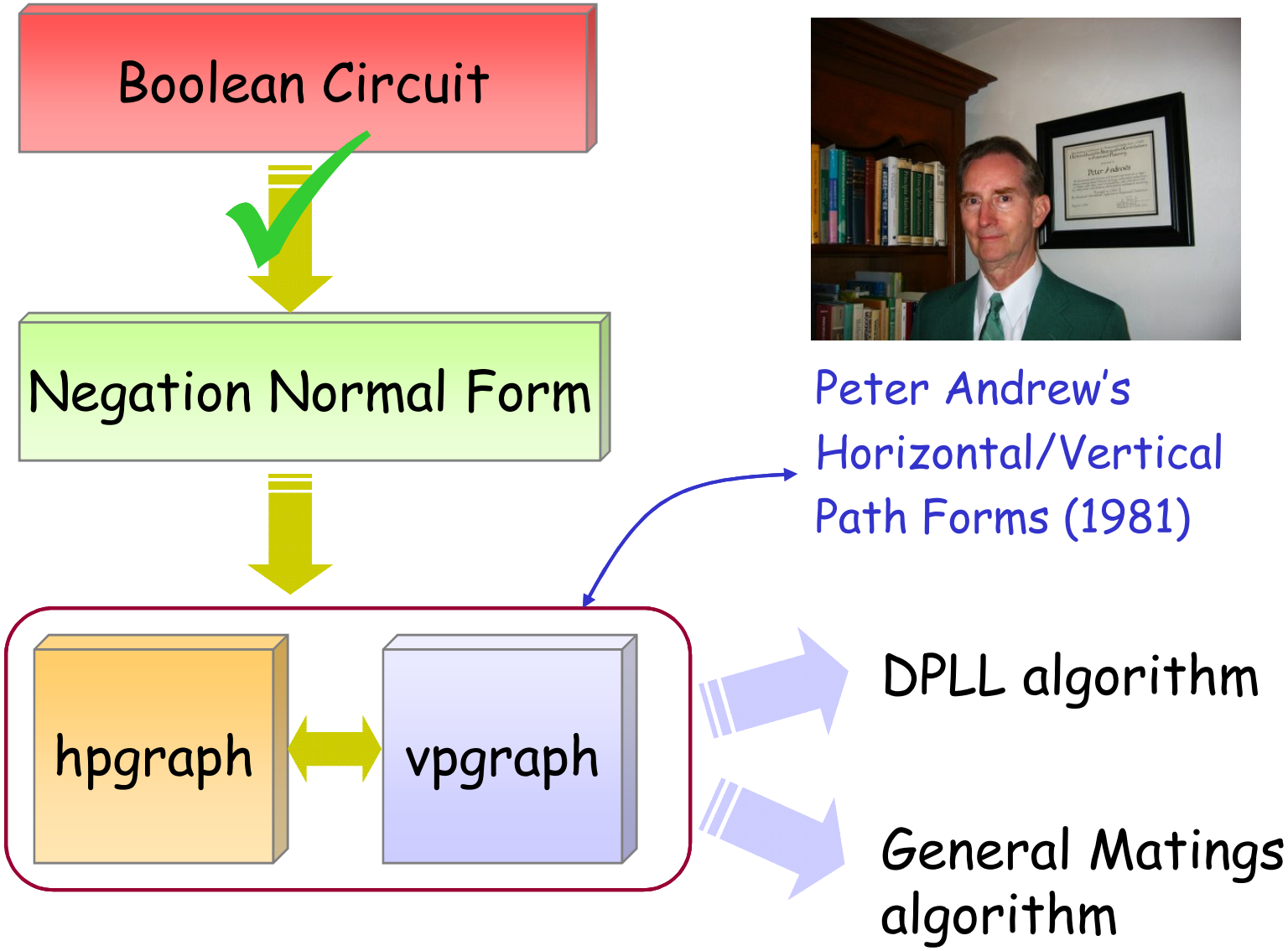


Why NNF: Number of variables



Note no pre-processing for NNF

Our SAT solving framework

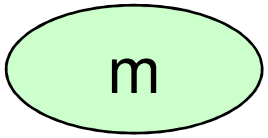


Peter Andrew's Horizontal/Vertical Path Forms (1981)

Inductively creating hpgraph(ϕ) from NNF formula ϕ

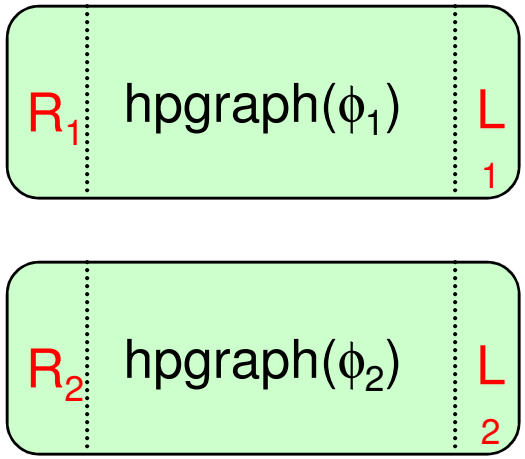
ϕ is a literal *m*

Create new node

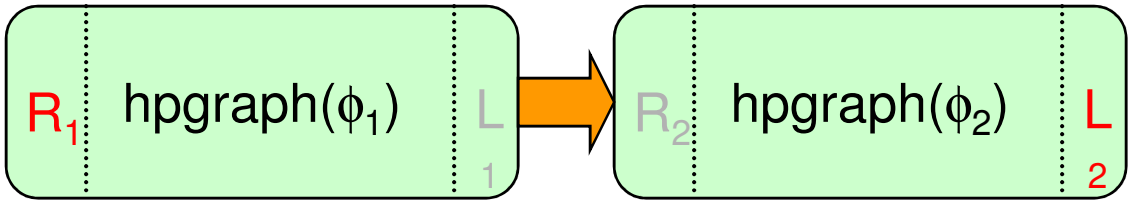


$\phi = \phi_1 \wedge \phi_2$

Take graph union



$\phi = \phi_1 \vee \phi_2$



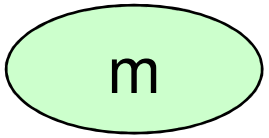
Add hyperedge from L_1 to R_2

Inductively creating vpgraph(ϕ) from NNF formula ϕ

Just interchange the handling of $\wedge$ and $\vee$

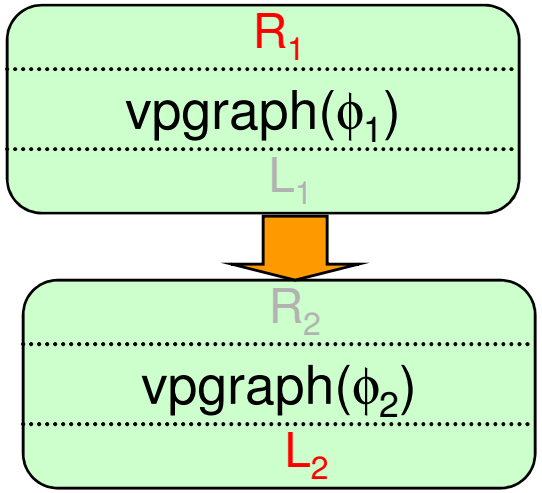
ϕ is a literal *m*

Create new node



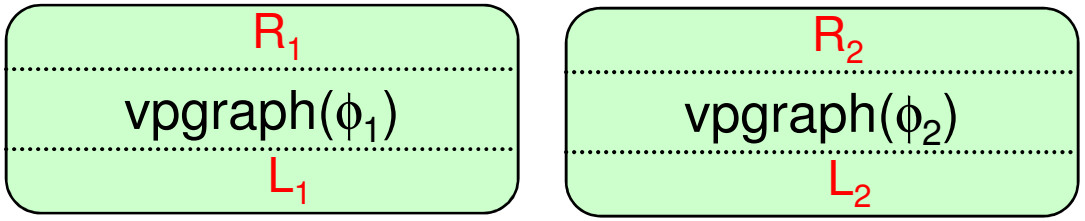
$\phi = \phi_1 \wedge \phi_2$

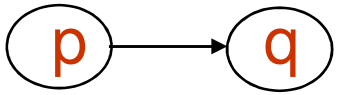
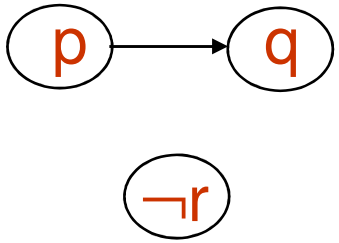
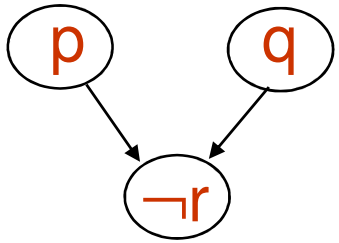
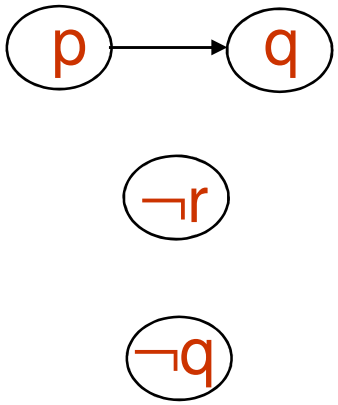
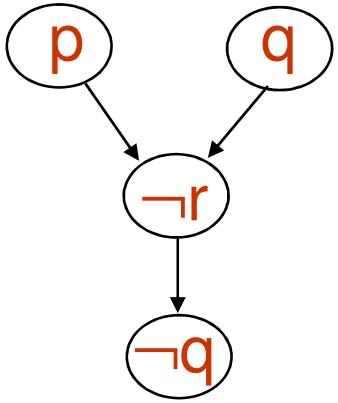
Add hyperedge from L_1 to R_2



$\phi = \phi_1 \vee \phi_2$

Take graph union

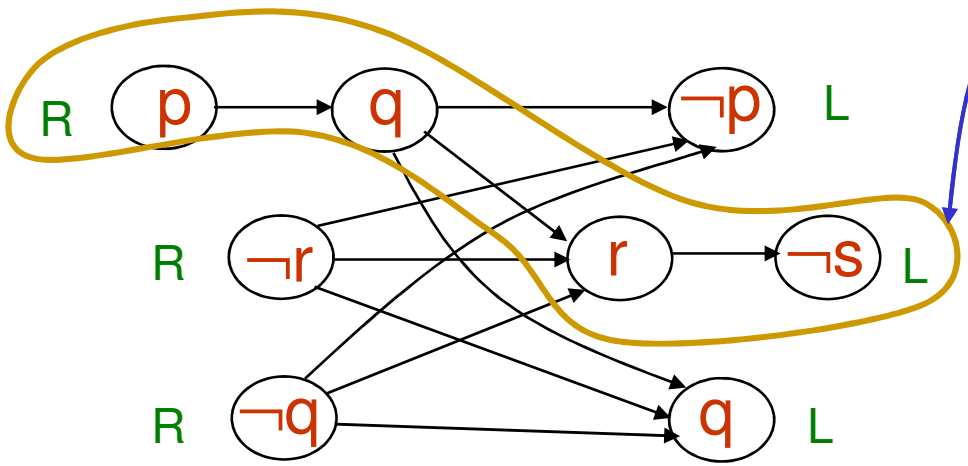


Formula	Hpgraph	Vpgraph
p		
$p \vee q$		
$(p \vee q) \wedge \neg r$		
$(p \vee q) \wedge \neg r \wedge \neg q$		

Example

Formula F: $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$

horizontal path



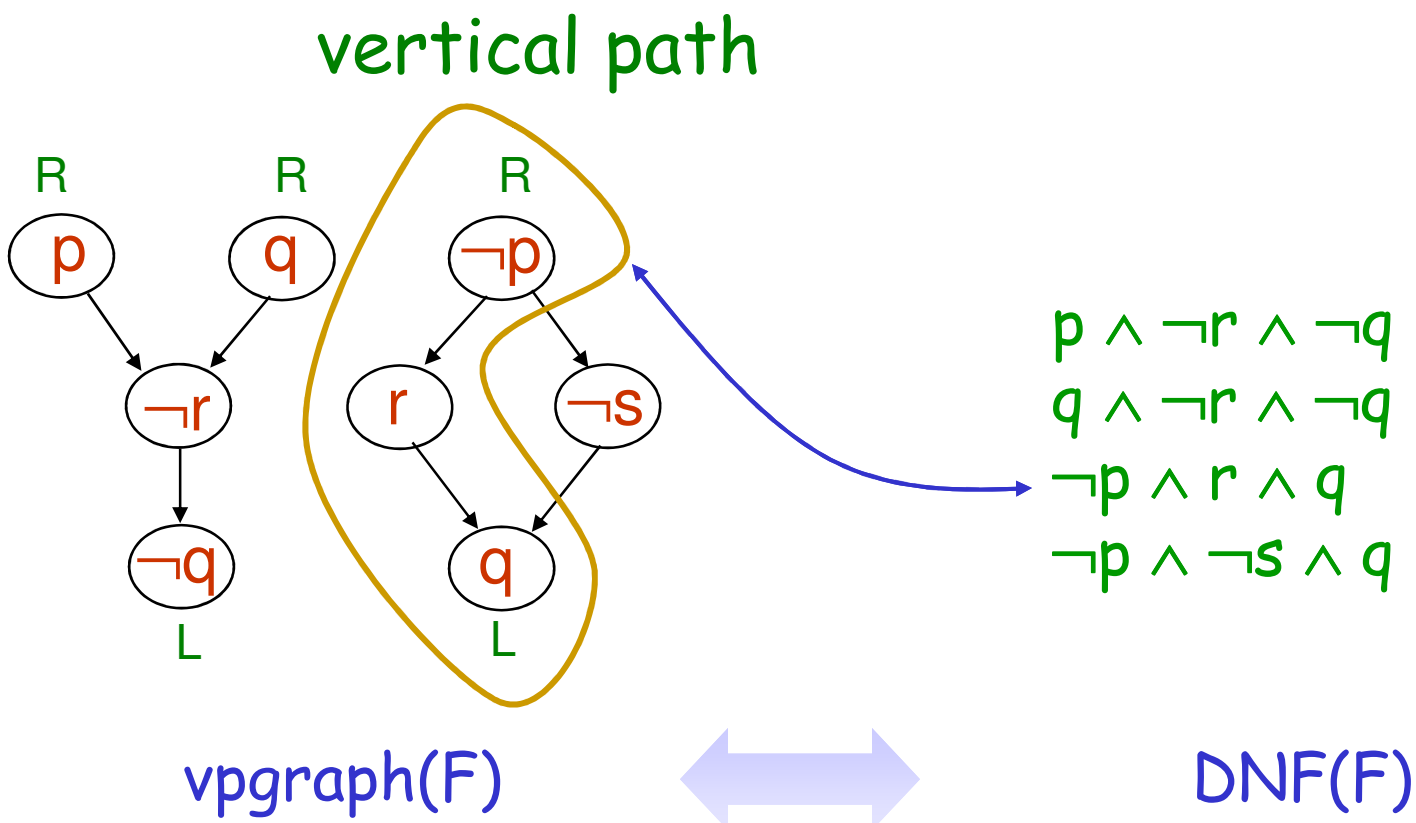
$p \vee q \vee \neg p$
$p \vee q \vee r \vee \neg s$
$p \vee q \vee q$
$\neg r \vee \neg p$
$\neg r \vee r \vee \neg s$
$\neg r \vee q$
$\neg q \vee \neg p$
$\neg q \vee r \vee \neg s$
$\neg q \vee q$

hpgraph(F) ⟷ CNF(F)

R denotes a root node and L denotes a leaf node

Example

Formula F: $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$

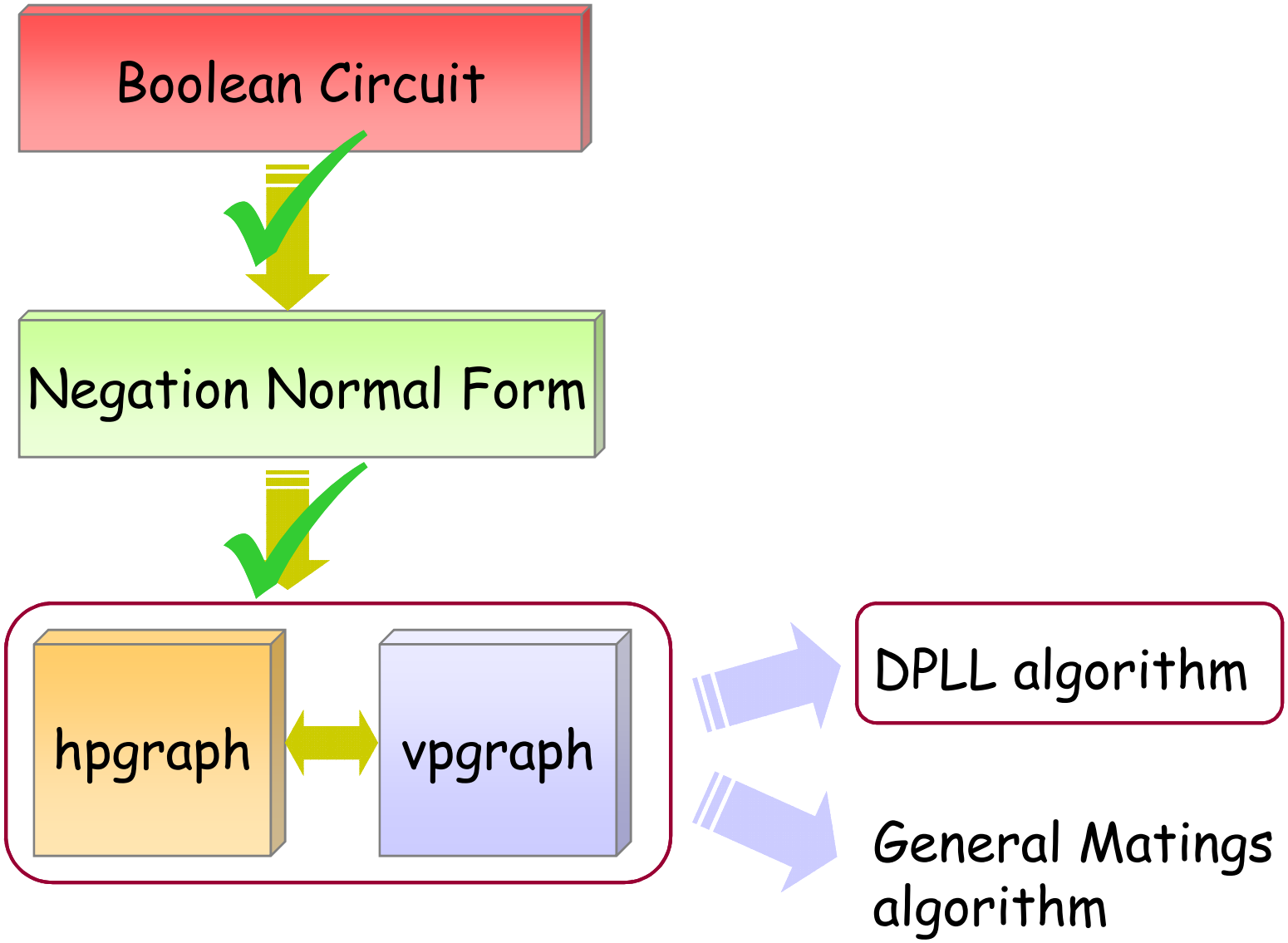


R denotes a root node and L denotes a leaf node

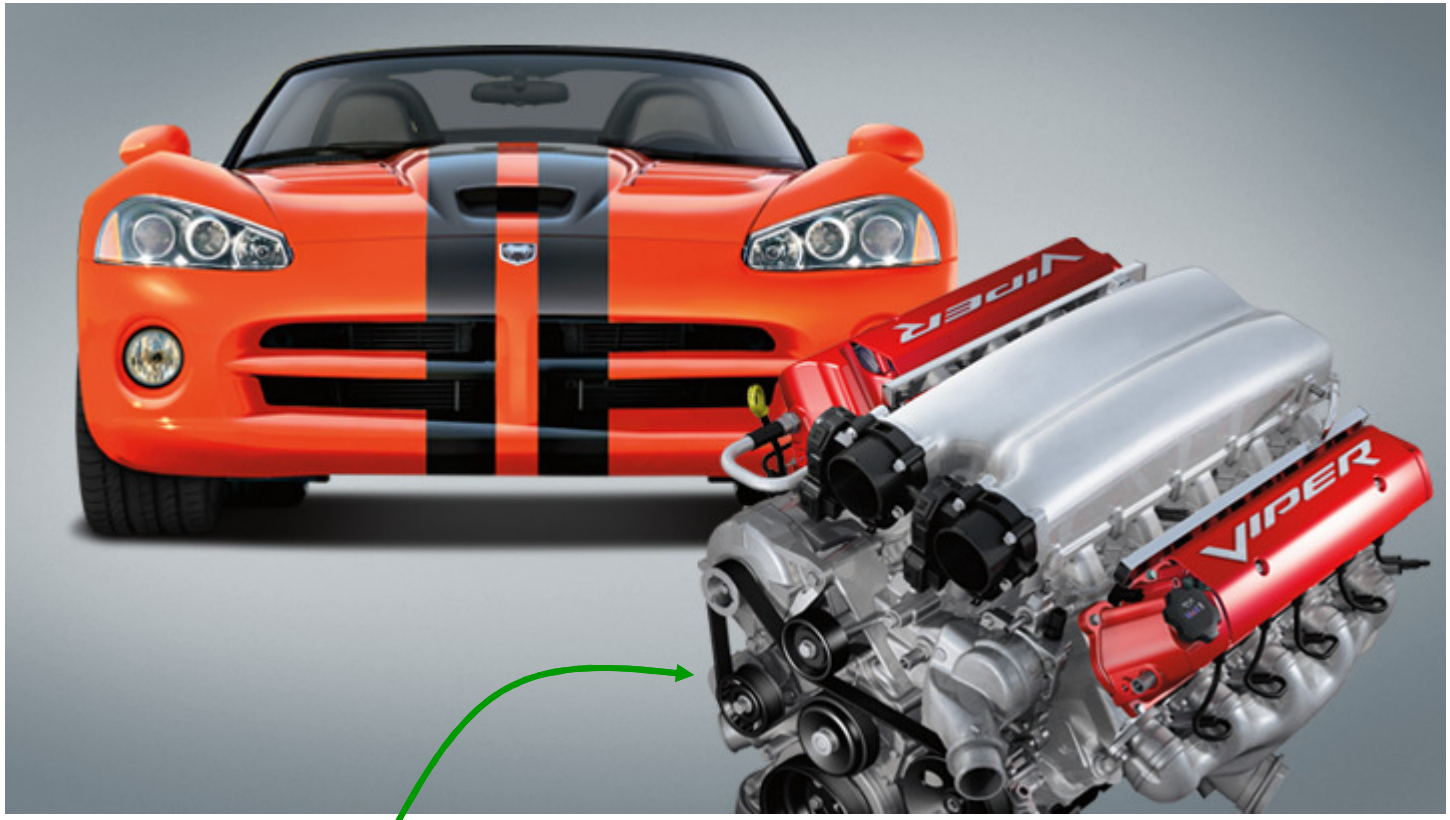
Representing NNF using Graphs

- Represent NNF formula ϕ in form of two graphs
- **hpgraph(ϕ)**
  - Implicitly encodes all clauses present in CNF(ϕ)
- **vpgraph(ϕ)**
  - Implicitly encodes all cubes/terms present in DNF(ϕ)
- Directed acyclic graphs, linear in the size of ϕ
- **Horizontal/Vertical Path Forms [P. Andrews 1981]**
  - Used in higher order theorem proving (General Matings)
  - Not been explored for building fast SAT solvers

Our SAT solving framework



DPLL SAT solver



Boolean Constraint Propagation (BCP)

Boolean Constraint Propagation (BCP)

- Given a partial assignment σ to variables in a formula ϕ
- BCP determines if σ falsifies ϕ (termed as conflict)
- Otherwise, BCP provides implied assignments (literals) due to σ

CNF example:
$(a \vee b \vee c) \wedge$
$(\neg a \vee \neg b) \wedge$
$(\neg c \vee \neg a)$

BCP detects a conflict when σ :={a=1, c=1).

For σ := {a=1} BCP detects {c=0,b=0} as implied literals

Efficient BCP in CNF SAT solvers

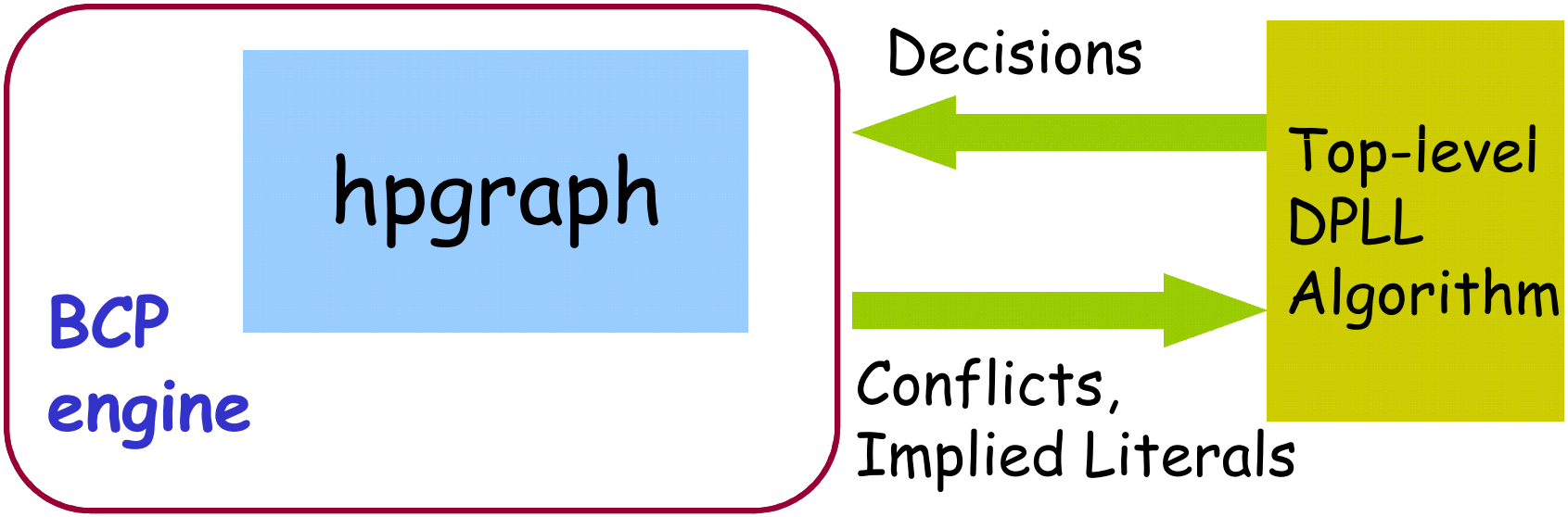
Two watched literal scheme [zChaff 2001]

Clause C: $l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5 \vee \ldots \vee l_{100} \vee \ldots$

* watch 1 * watch 2

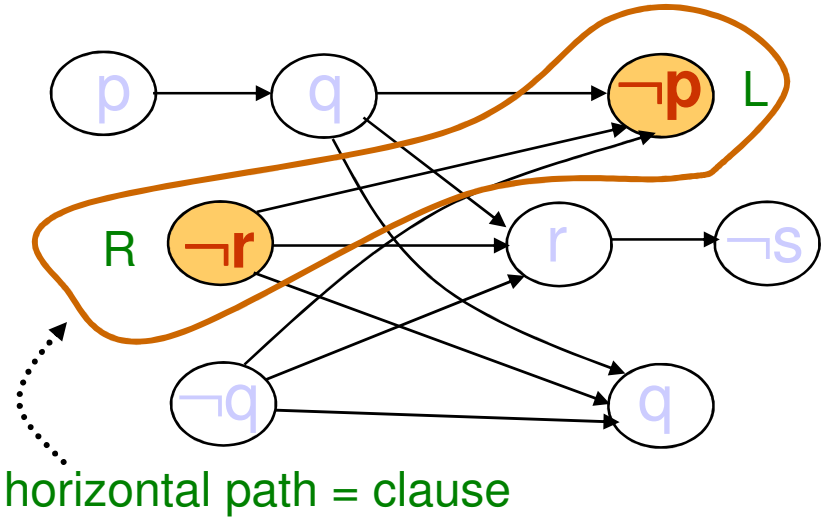
C is examined during BCP only when one of watches (l_2 or l_5) becomes false

DPLL on hpgraph



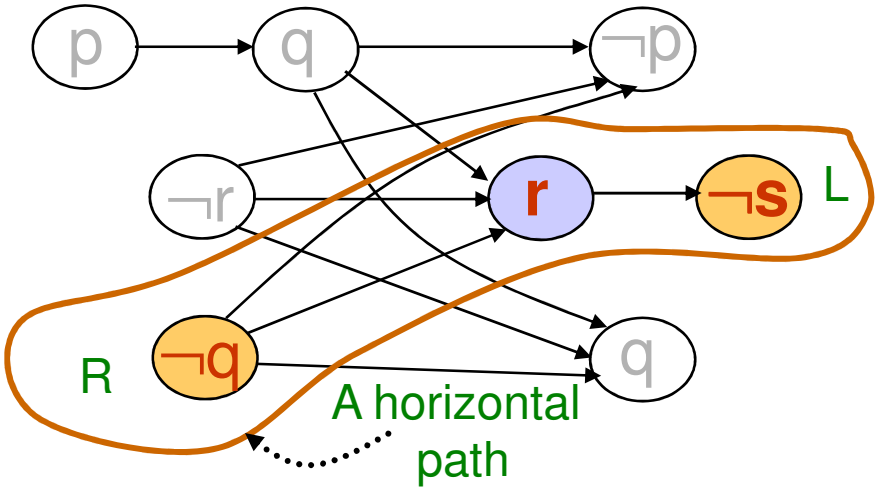
Meaning of BCP on hpgraph

Assignment σ_1 :={r=1,p=1}



Conflict clause: ¬r ∨ ¬p

Assignment σ_2 :={q=1,s=1}

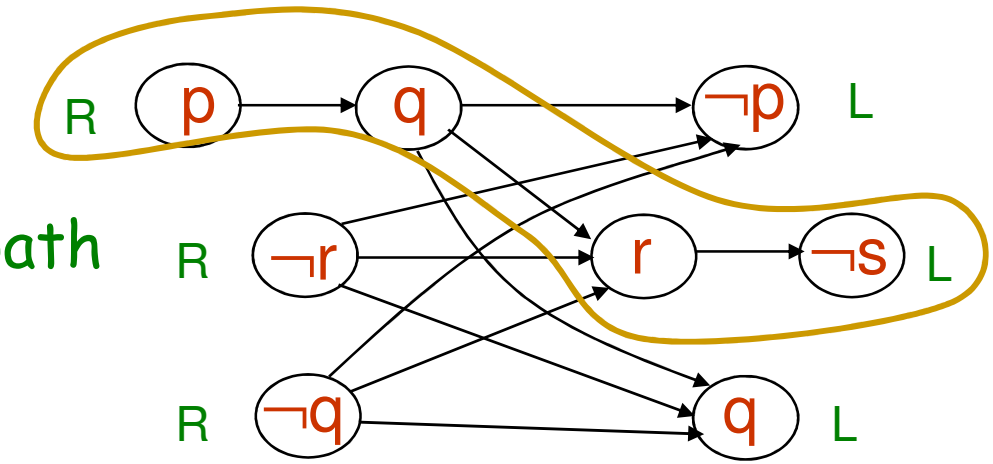


Unit clause: ¬q ∨ r ∨ ¬s
Implied literal: r

Can we generalize the CNF watched literal scheme?

Hpgraph

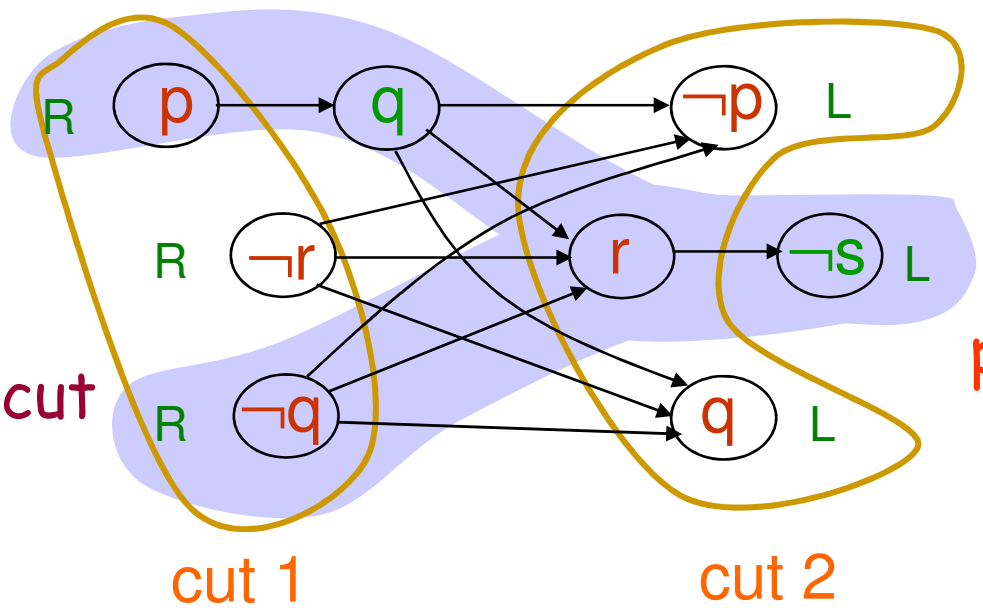
Horizontal path
=
clause



- Watch two literals (nodes) on each clause (path)
- But exponential number of clauses (paths)!

Two Watched Cut Scheme

Hpgraph



$\neg q \vee r \vee \neg s$

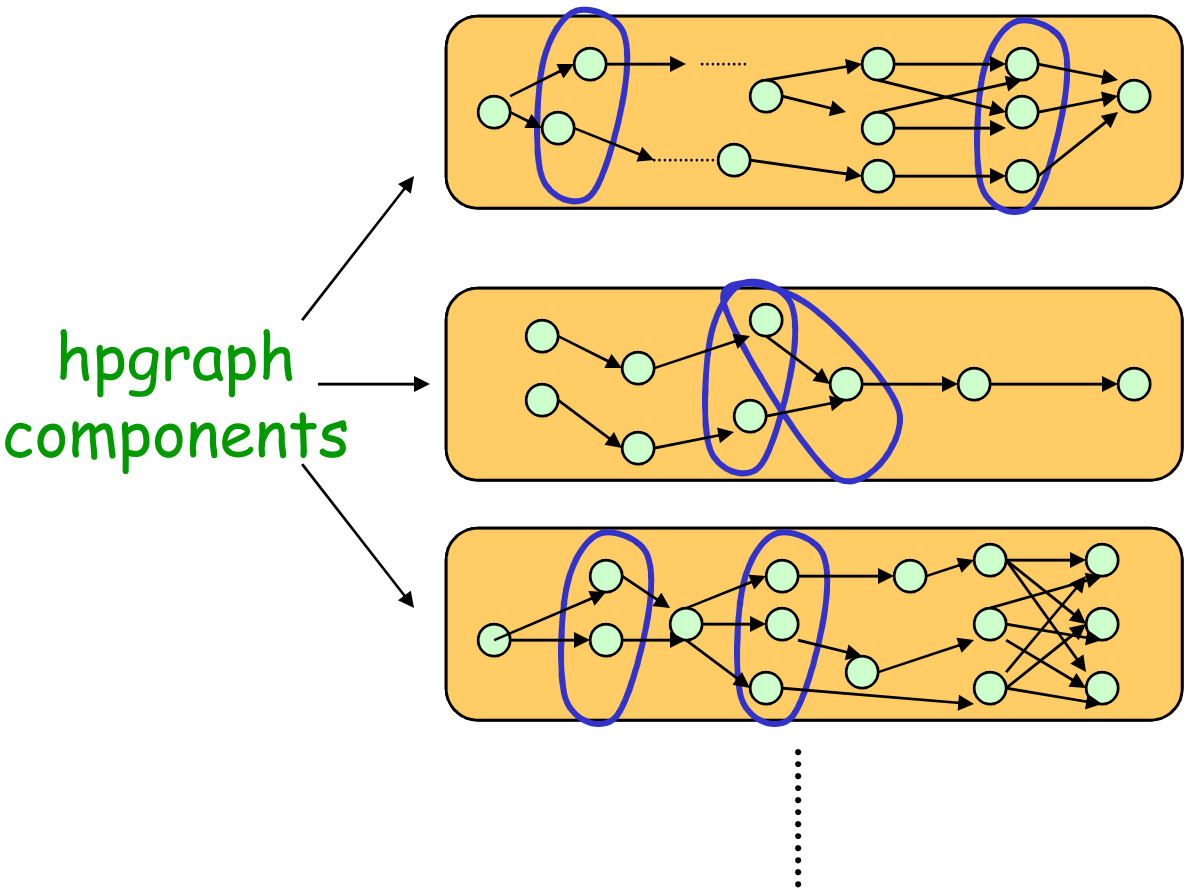
$p \vee q \vee r \vee \neg s$

watch a node cut

1. A node cut disconnects all horizontal paths
2. Watch two node cuts (allows observing two literals on each clause)
3. Minimal cuts (covered later)

Actual picture....

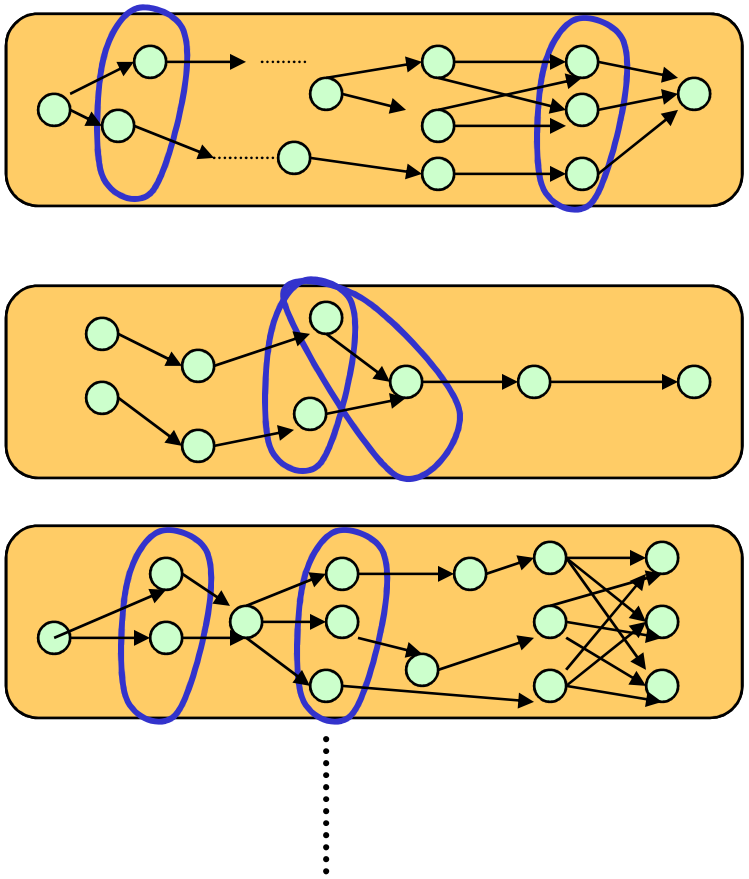
An hpgraph



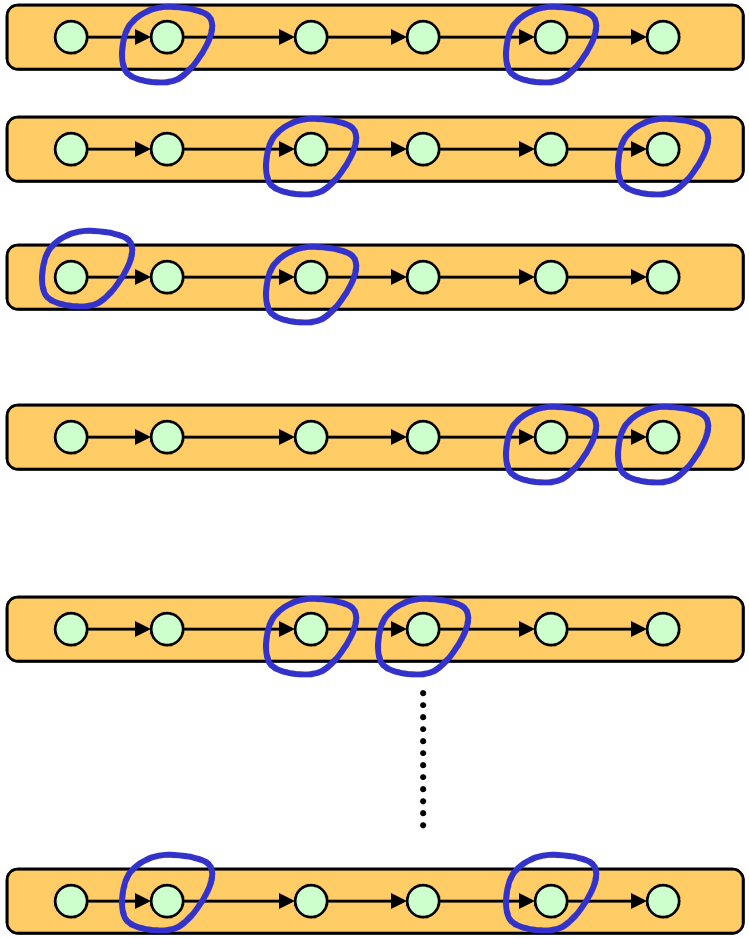
1. Two cuts for each hpgraph component
2. Can be updated locally during BCP
3. Non-chronological backtracking is cheap

Our algorithm generalizes CNF watched literal scheme

An hpgraph



Special hpgraph (CNF)

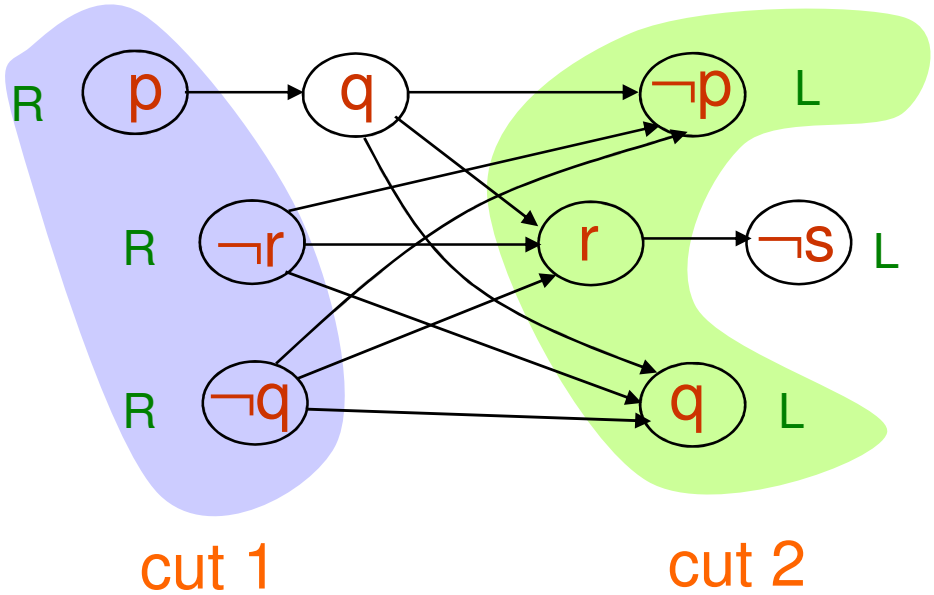


Acceptable cut

A cut is acceptable if no literal appearing in it is false

Cut 1 is:

acceptable for
σ:= {p=1}



Cut 2 is:

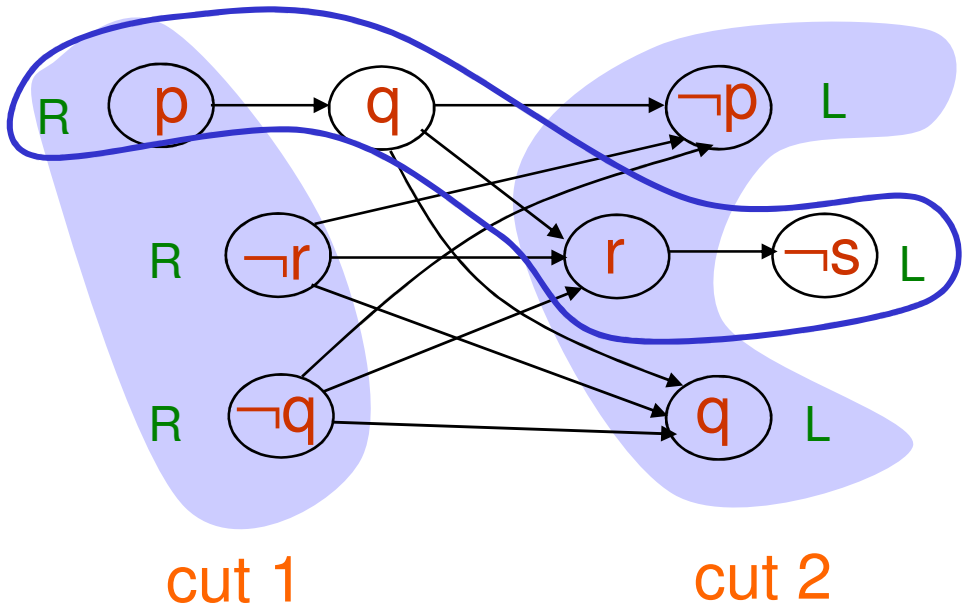
not acceptable for
σ:= {p=1}

Now let us see the use of cuts during BCP

BCP Case 1: Both cuts are acceptable and disjoint

Hpgraph component

$\sigma := \{s=1\}$



No need to examine the hpgraph component

Intuitively, there will be no conflicts or implied literals

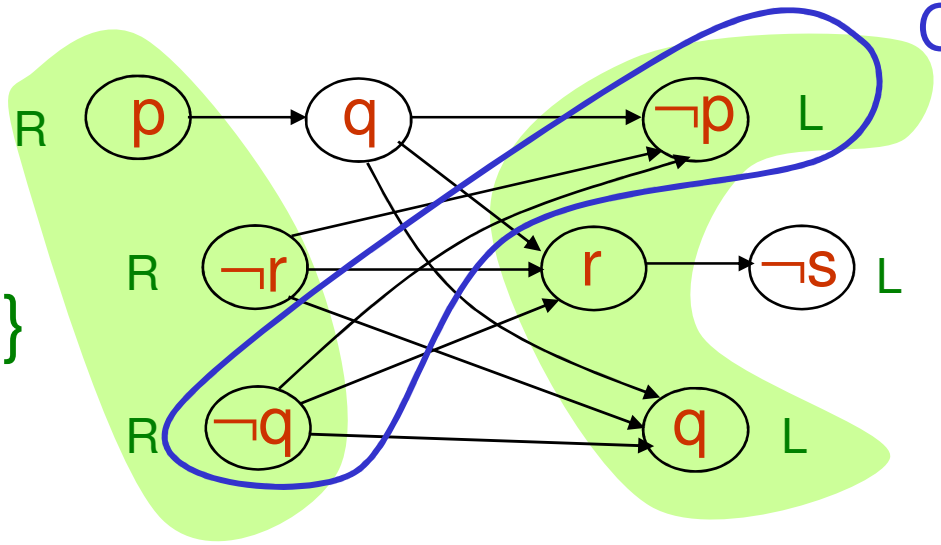
BCP Case 2 (conflict): No acceptable cut

Hpgraph component

Conflict clause:

$\neg q \vee \neg p$

$\sigma := \{p=1,q=1\}$

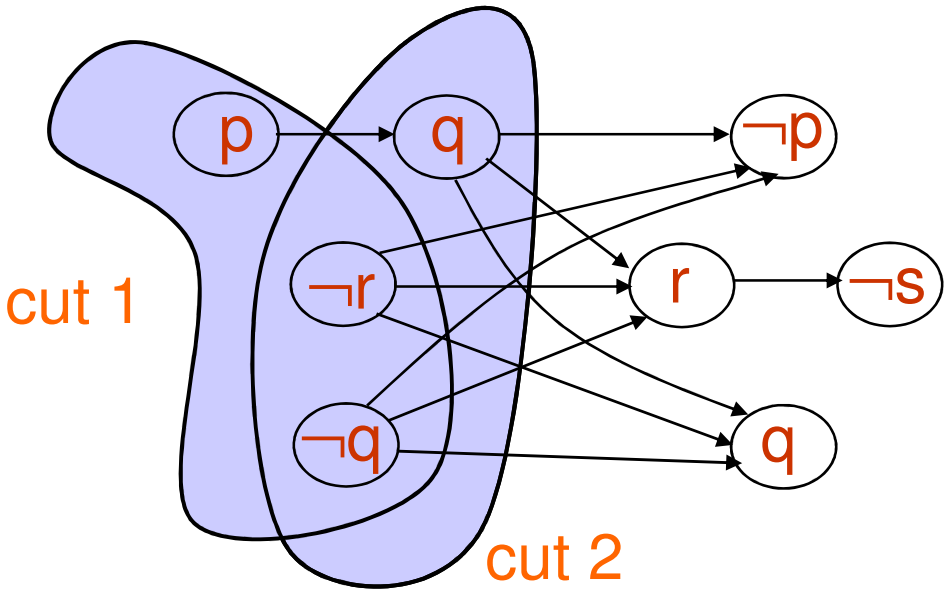


cut 1 cut 2

An hpgraph has no acceptable cut if and only if current assignment falsifies the formula

BCP Case 3 (Implications): Acceptable cuts but not disjoint

Hpgraph component



For σ := {p=1} we can find two acceptable cuts.

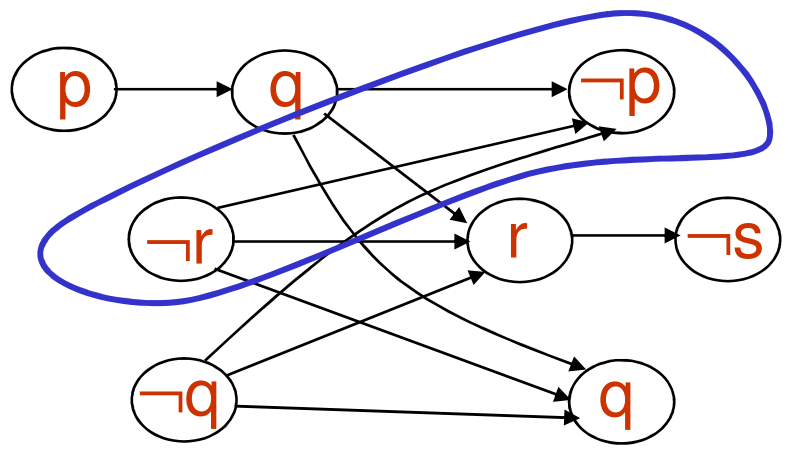
We cannot find two completely node-disjoint cuts

Intuitively nodes common to both cuts contain implied literals

In our example ¬**r** and ¬**q** are implied literals.

BCP Case 3 (Implications): Acceptable cuts but not disjoint

Hpgraph component



$\sigma := \{p=1\}$

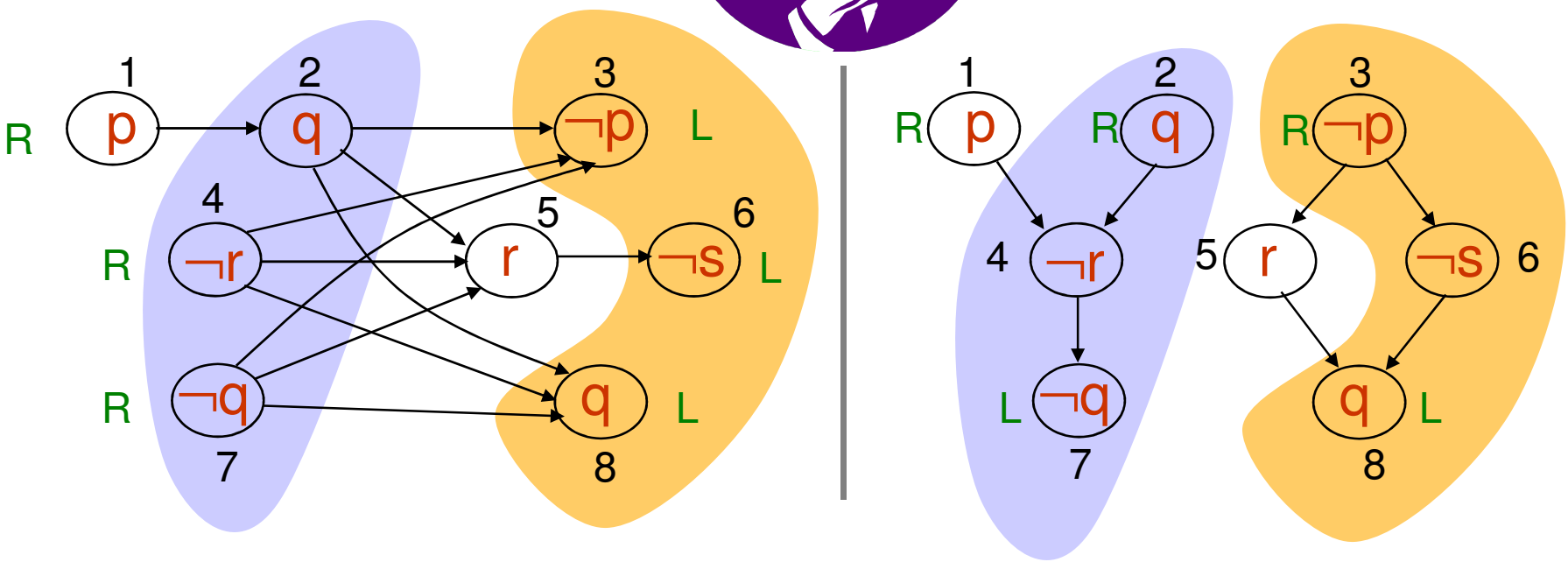
¬r is implied due to unit clause ¬r ∨ ¬p

¬q is implied due to unit clause ¬q ∨ ¬p

Finding and Maintaining Minimal Cuts

hpgraph component

vpgraph component

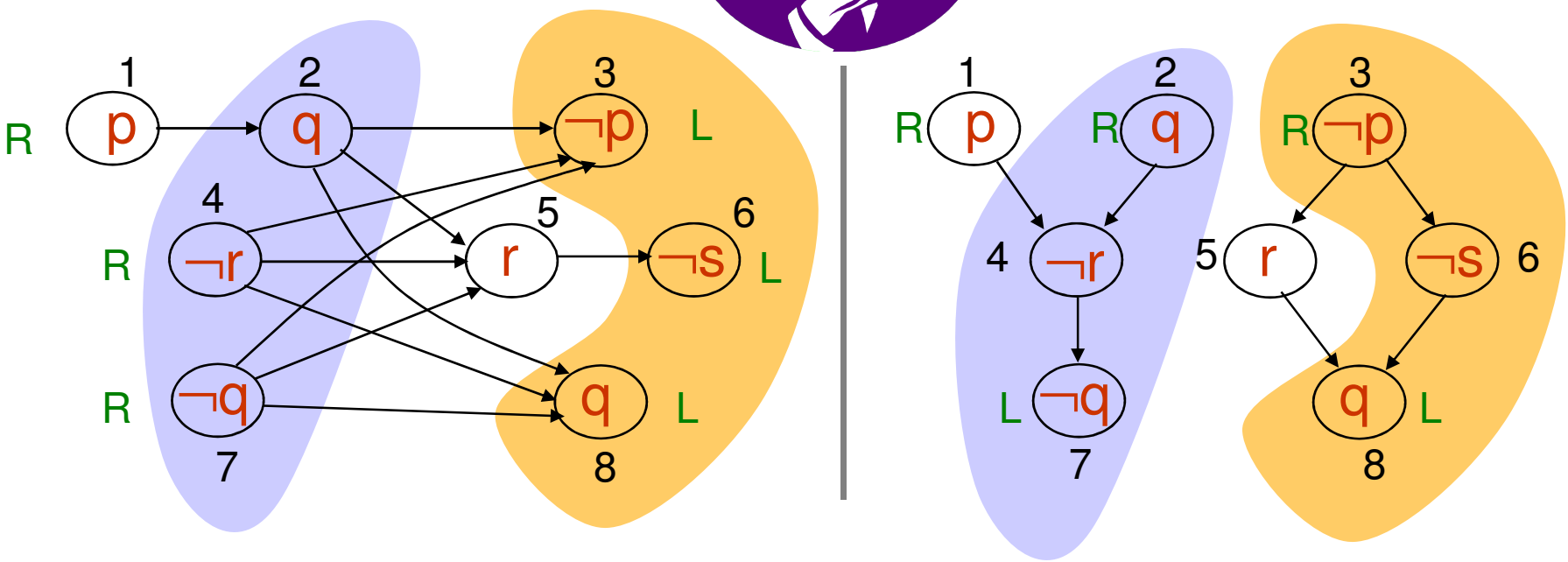


minimal cut in hpgraph component =
a path in vpgraph component

Finding and Maintaining Minimal Cuts

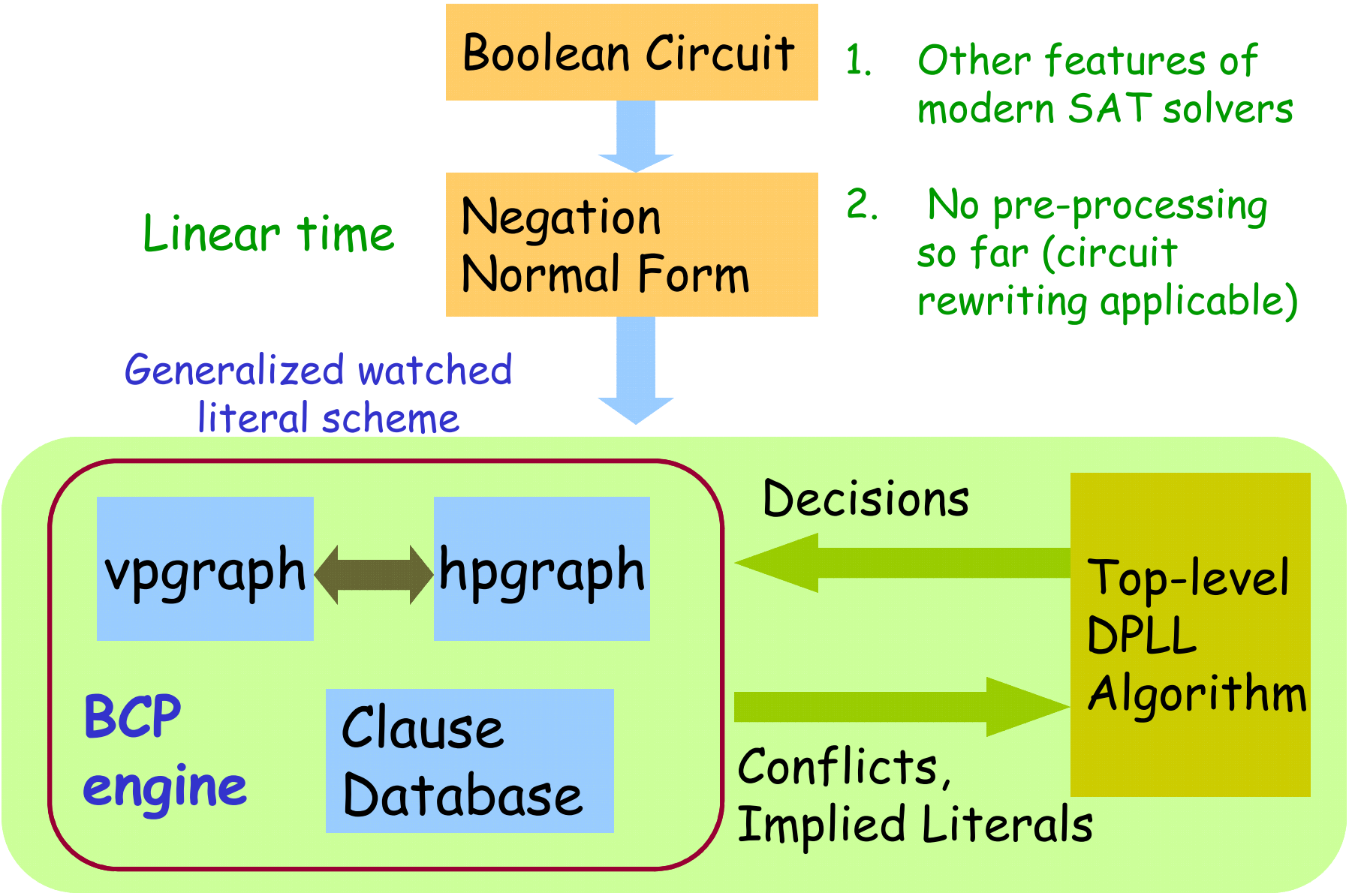
hpgraph component

vpgraph component



Depth first search for an acceptable path in vpgraph gives us an acceptable cut in hpgraph…

Overall Solver



1. Other features of modern SAT solvers
2. No pre-processing so far (circuit rewriting applicable)

Experimental snapshot

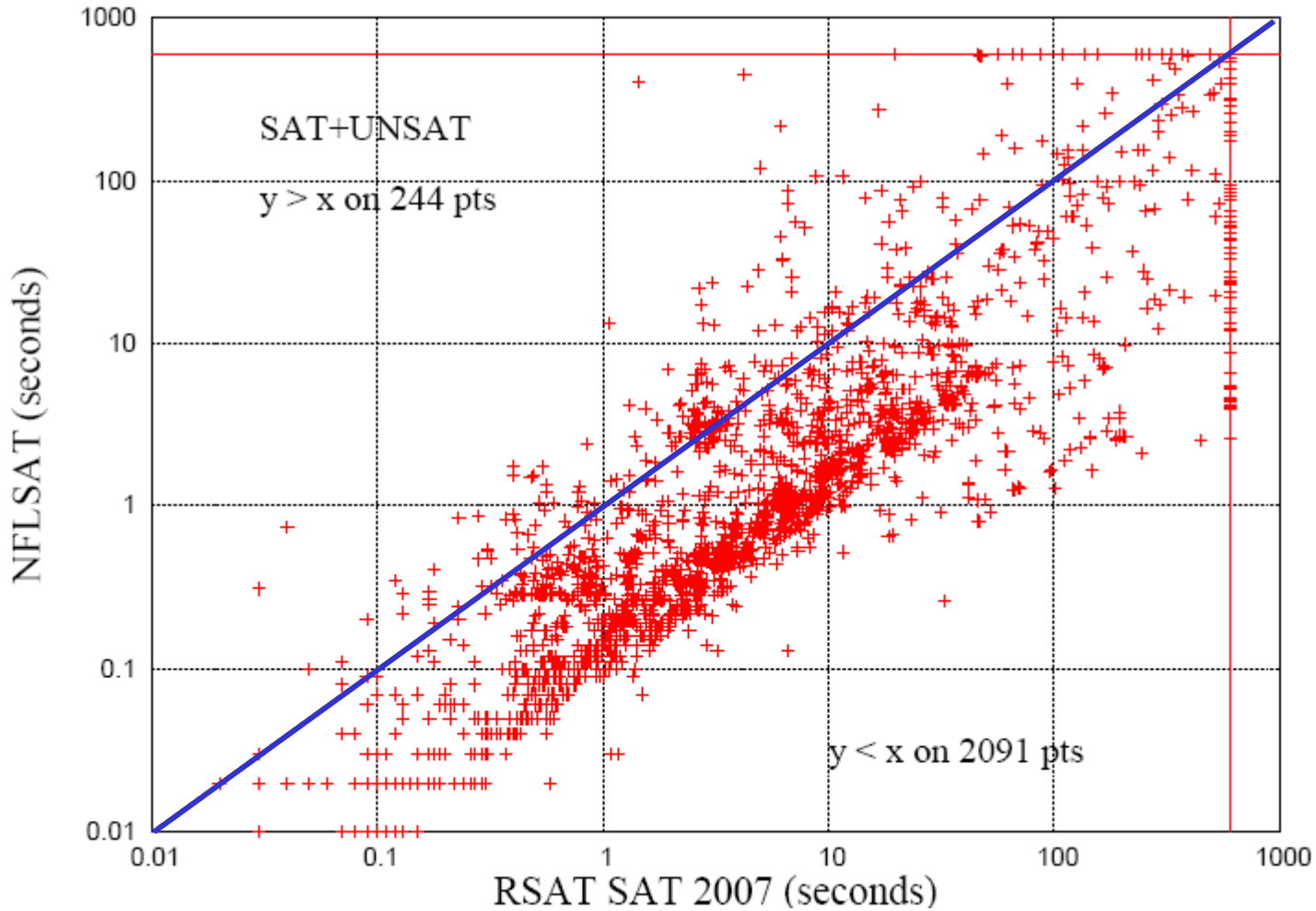
- These techniques implemented in:
  - **NFLSAT** (Non-clausal FormuLas SATisfiability checker)
- ~2500 non-clausal circuit formulas (industrial category)
  - Bounded model checking, k-induction, SW/HW verification
  - CNF obtained by adding new variables
- Comparing with state-of-the-art solvers
  - SAT-Race 2008 top two winners: MiniSAT++, Picoaigersat
  - Top three winners of SAT 2007 comp: RSAT, MiniSAT, PicoSAT
  - Using the competition binaries or latest source code

Comparison with SAT 2007 winners

Solver	Solved	Failed	Solved time	Total time
NFLSAT	2364	178	29753	131153
RSAT	2310	231	45794	184394
PicoSAT	2281	260	43298	199298
MiniSAT2	2272	269	50958	212358
MiniSAT2 070721	2270	271	39490	202089

Timeout 600 seconds per problem. Total 2541 problems.

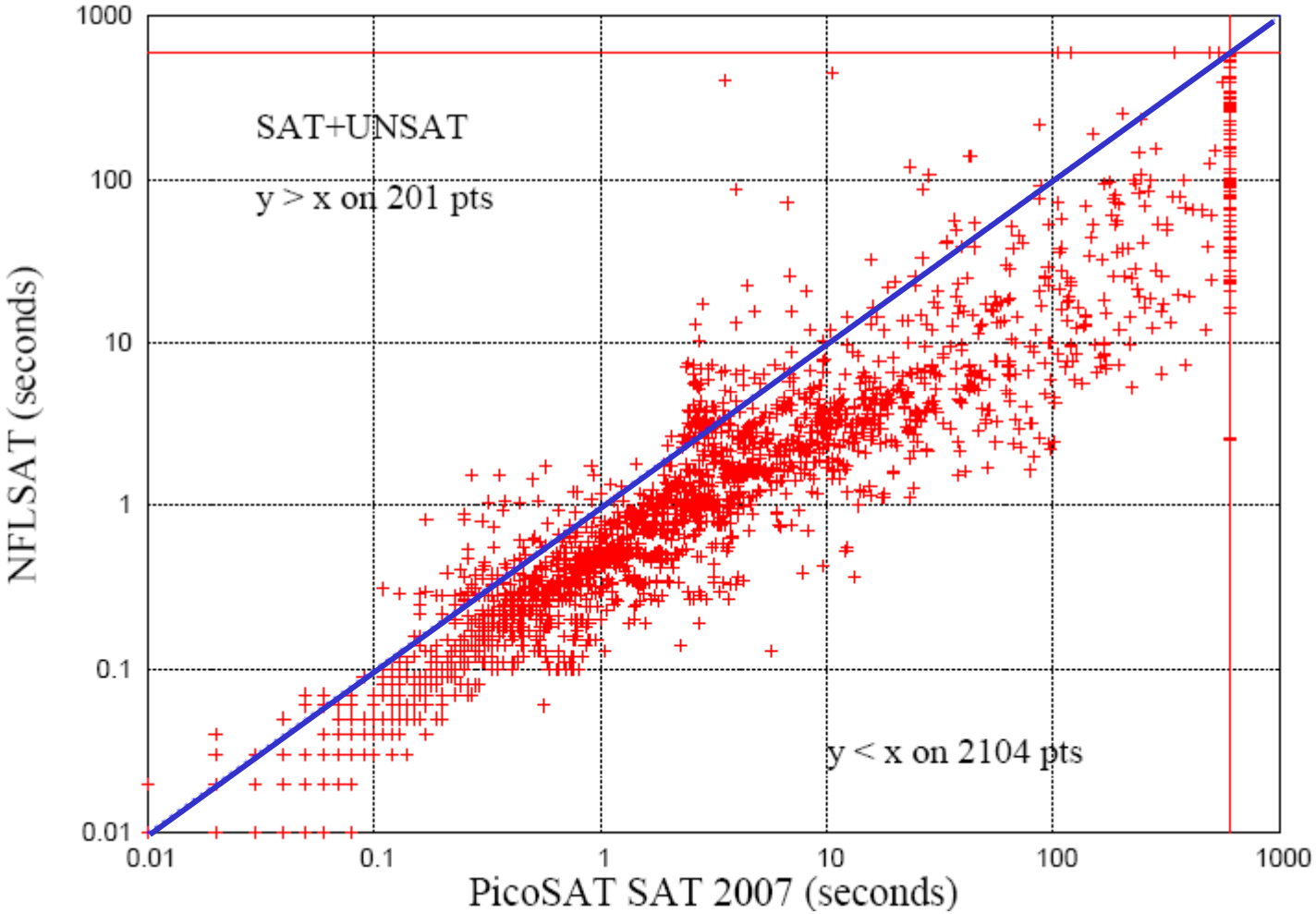
NFLSAT vs. RSAT



NFLSAT solves 54 more problems

RSAT uses pre-processing

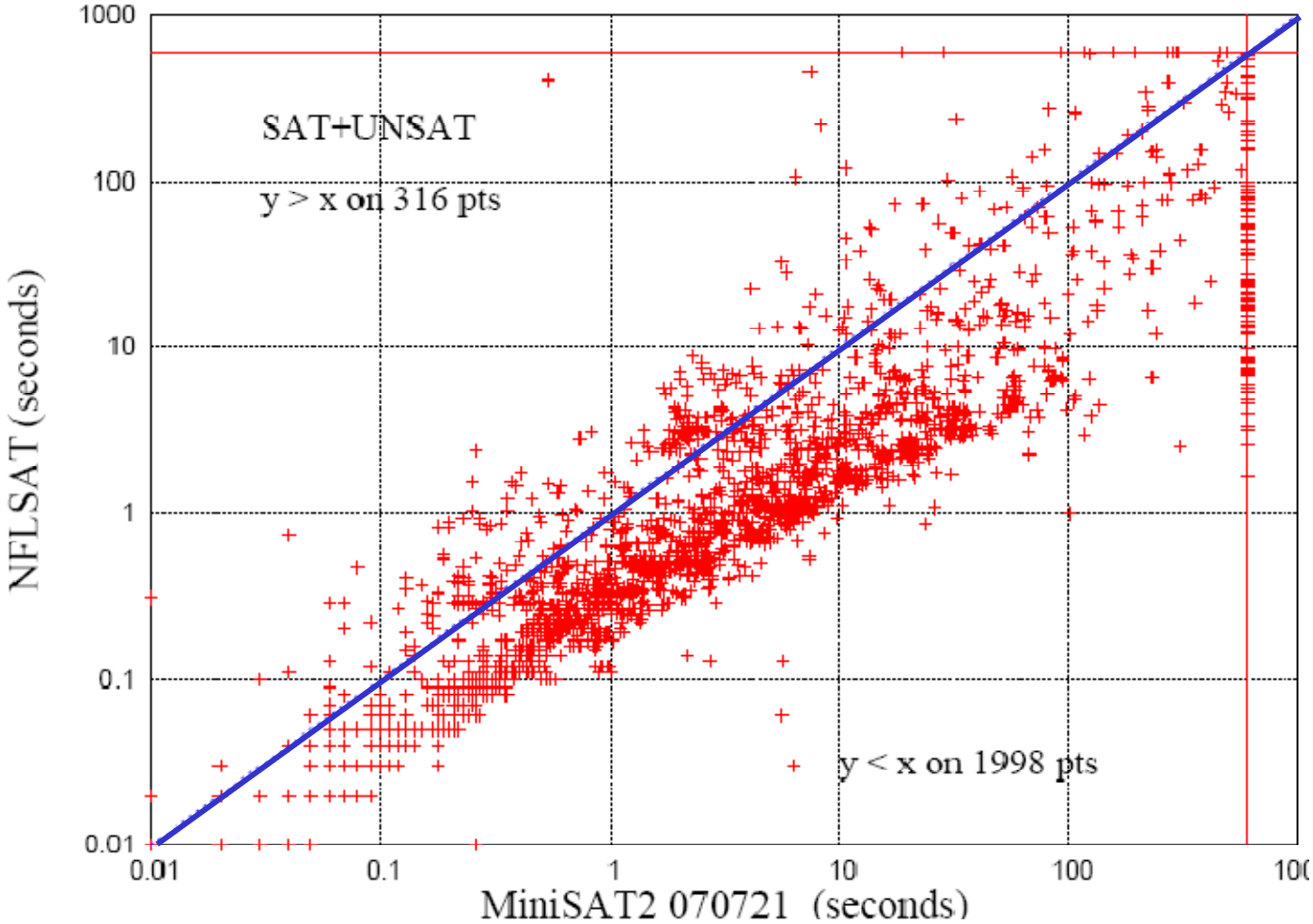
NFLSAT vs. PicoSAT



NFLSAT solves 83 more problems

PicoSAT does not use pre-processing

NFLSAT vs. MiniSAT2 070721



NFLSAT solves 94 more problems

MiniSAT2 070721 uses pre-processing

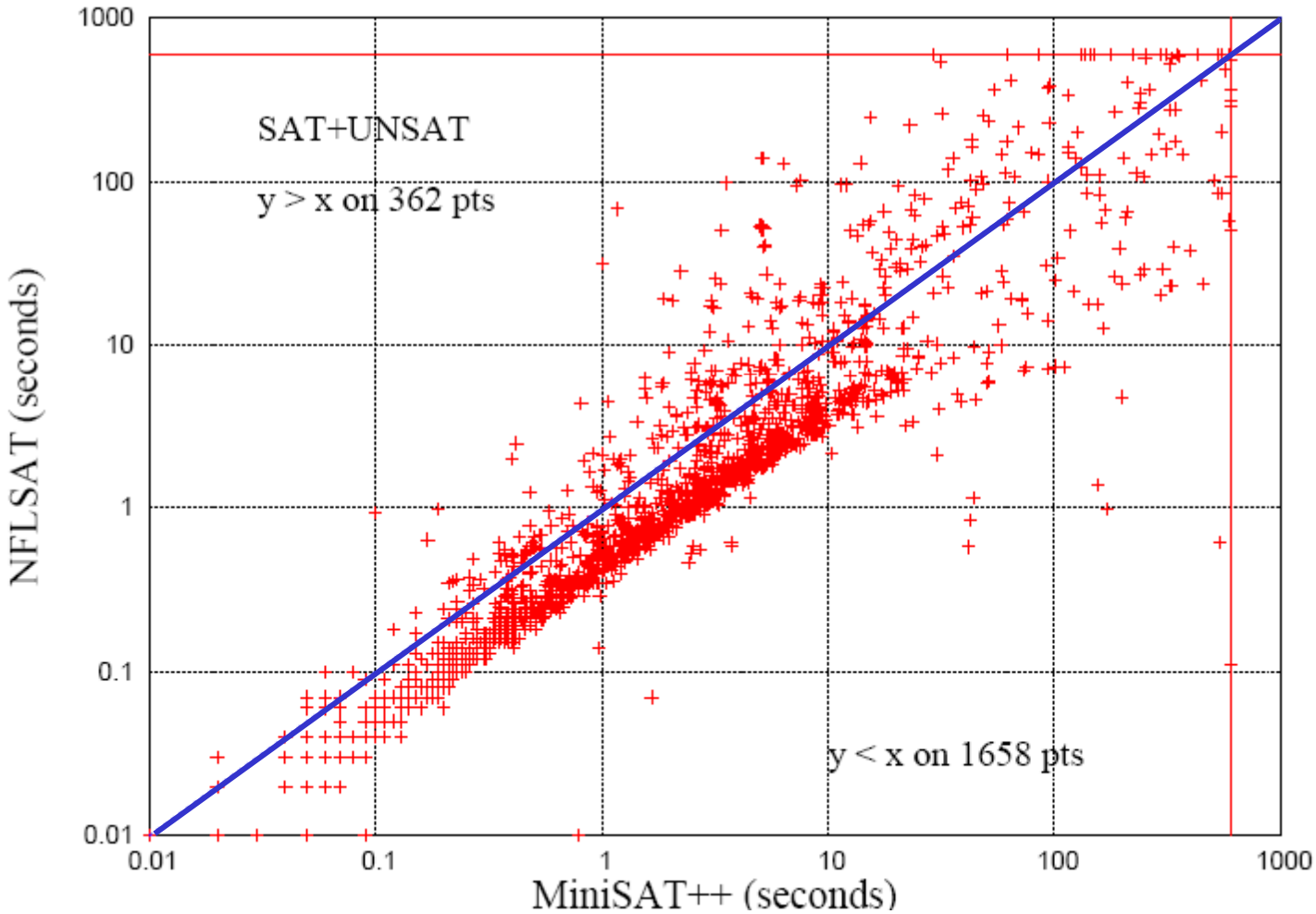
Comparison SAT-Race 2008 winners And Inverter Graph (AIG) Track

Solver	Solved	Failed	Solved time	Total time
NFLSAT	2060	132	26585	105785
MiniSAT++	2074	118	32457	103257
PicoaigerSAT	2033	159	35892	131292

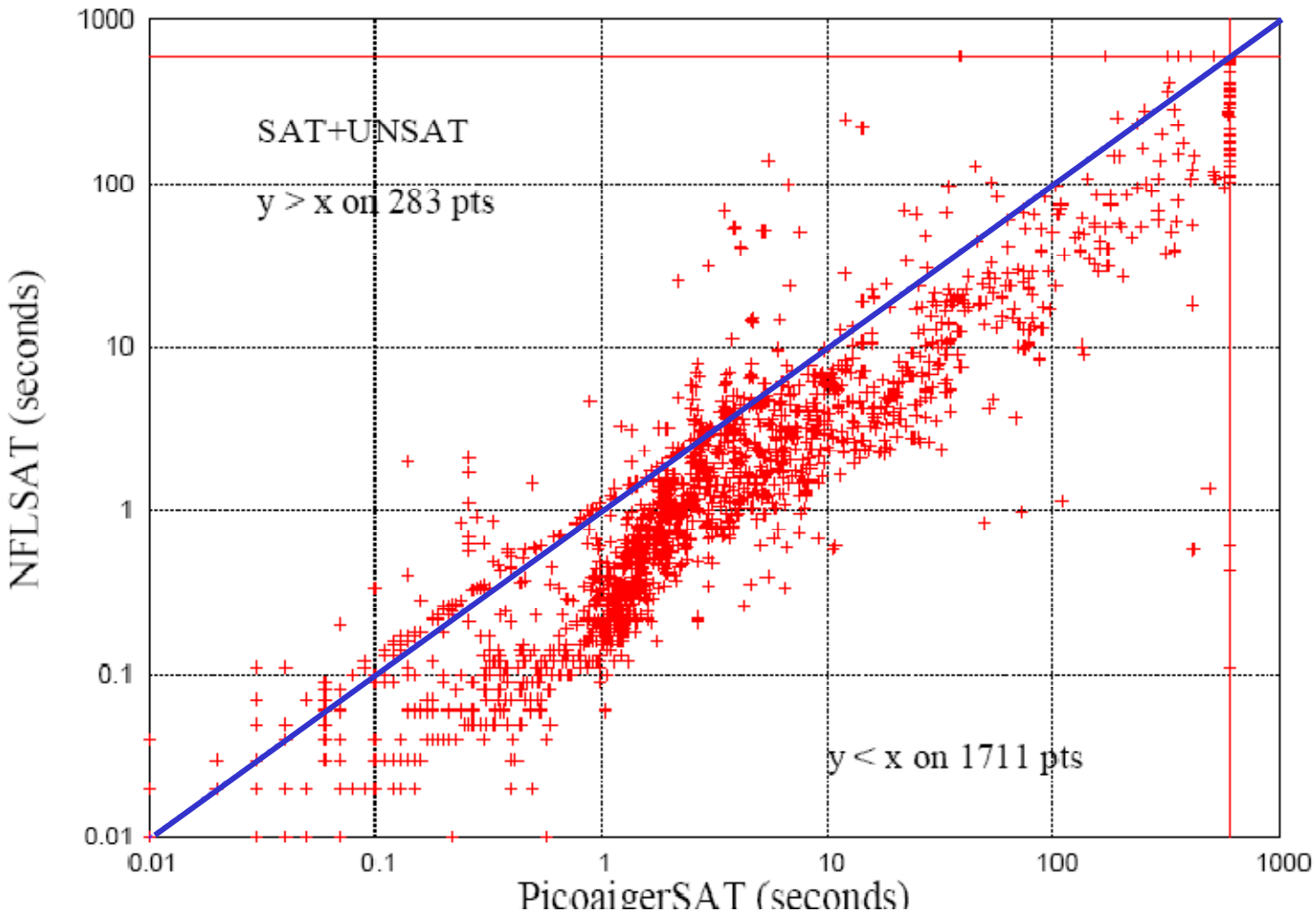
Timeout 600 seconds per problem. Total 2192 problems.

If we remove 350 SAT 2007 benchmarks, NFLSAT (8504 sec) is twice as faster than MiniSAT++ (18119 sec)

NFLSAT vs. MiniSAT++



NFLSAT vs. PicoaigerSAT



Outline

- Motivation
- Fast Boolean Satisfiability (SAT) Solvers
  - DPLL Algorithm
  - Related work
  - Negation Normal Form, hpgraph, vpgraph
  - SAT solving using hpgraph, vpgraph
- Interpolation for Subsets of Integer Linear Arithmetic
  - An application
- Summary

Decision Procedures++

Reasoning engines in verification tools

Formula

Satisfiable

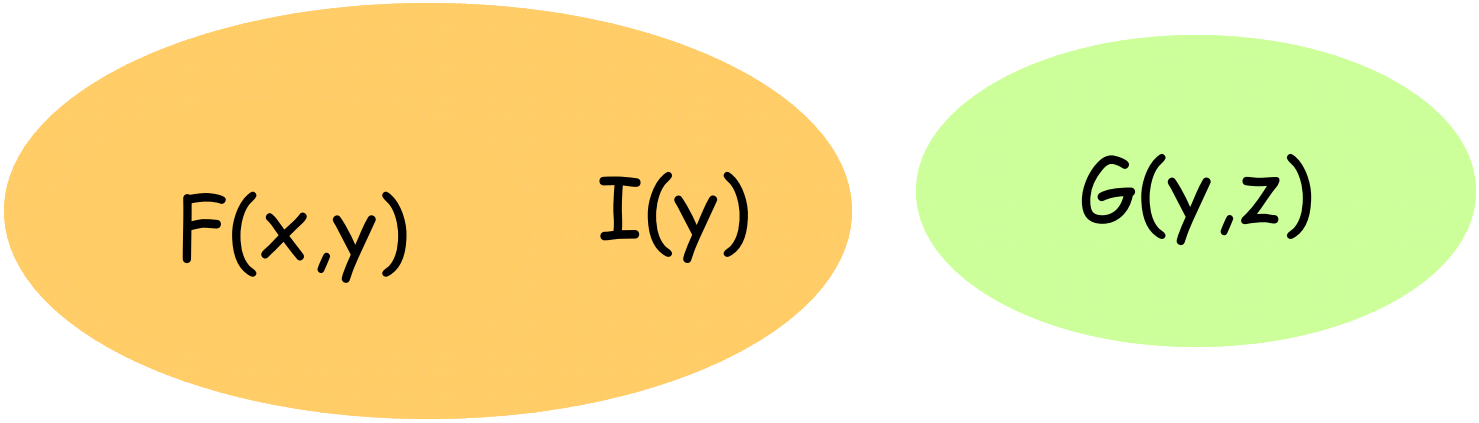
Unsatisfiable
(Craig Interpolants)

Interpolants [Craig 1957]

Given formulas F, G such that $F \wedge G$ is unsatisfiable

An interpolant for (F,G) is a formula I:

1. $F \Rightarrow I$
2. $I \wedge G$ is unsatisfiable
3. I contains only common variables of F and G



Interpolants Example

Example 1 (propositional logic):

$F := p \wedge q$ $\quad$ $G := \neg q \wedge r \wedge s$

$I := q$

Example 2 (linear arithmetic):

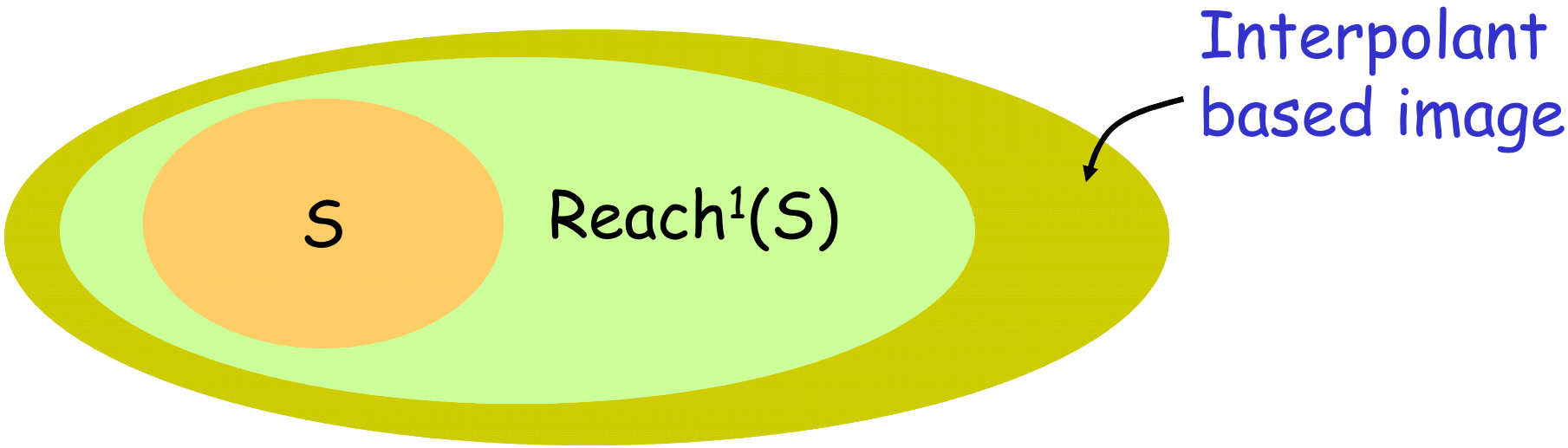
$F := x + 2y \leq 3 \wedge -x - y \leq -1$ $\quad$ $F \Rightarrow y \leq 2$

$G := y \geq 3$

$I := y \leq 2$

Interpolants in Verification [McMillan 2003]

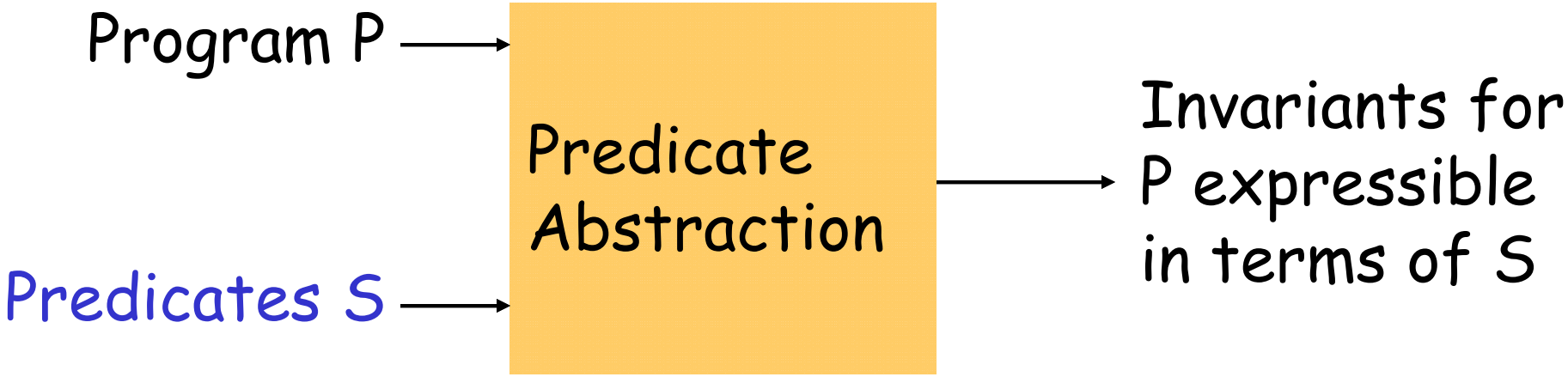
Useful in symbolic model checking



Computing Reach^1(S) requires existential quantification (costly using BDDs or SAT)

Interpolants in Verification [Jhala et al. 2004]

Useful for Property Directed Invariant Generation



Interpolants help in finding right set of predicates

How are Interpolants Obtained

$F \wedge G$

proof of unsatisfiability of $F \wedge G$

F, G

Interpolant for (F, G)

Existing Work on Computing Interpolants

Pudlak, McMillan, Jhala et al., Yorsh et al., Kapur et al., Rybalchenko et al., Kroening et al., Cimatti et al., Beyer et al.

- Can efficiently compute interpolants
  - For **rational/real linear** arithmetic
  - For equality with uninterpreted function symbols
  - Propositional logic (using SAT solvers)

- No efficient interpolation algorithms for
  - **Integer linear arithmetic**
  - **Bit-vector arithmetic**
  - Decision problem for conjunctions is itself NP-hard

We make progress in this direction.

Our results

- Polynomial time interpolation algorithms
  - For useful subsets of integer linear arithmetic
- Integer (Diophantine) linear equations
  - E.g. $x = 3y \wedge 5x = 3z+u+2 \wedge \ldots$
- Integer linear congruences (modular equations)
  - E.g. $4x = 2y + 9 \pmod 3 \wedge 2z + 5x - y = 7 \pmod 4 \wedge \ldots$
- Integer linear equations and disequations
  - E.g. $\neg(4x + 5y = 8) \wedge x = 3y \wedge \ldots$

Interpolation for Integer Linear Equations

- F, G be conjuctions of integer linear equations
- We show that interpolant for (F,G) is always:
  - An integer linear equation or
  - An integer linear congruence
- F := (x = 2y) and G := (x=2z+1)
  - An interpolant is x = 0 (mod 2)

Interpolation Algorithm Step 1

Obtain a proof of unsatisfiability of $F \wedge G$

(How to get a contradiction from $F \wedge G$)

$$F := (30x + 4y = 2)$$
$$G := (y + 5z = 2)$$

$\Downarrow$ [1/5, 1/5]

1/5 F + 1/5 G is equal to

6x+y+z=4/5 (Contradiction)

Interpolation Algorithm Step 2

Sum the equations from F according to the proof of unsatisfiability

F G

$1/5\ (30x + 4y = 2) + 1/5\ (y + 5z = 2)$

$6x + 4/5\ y = 2/5$

Partial interpolant

We do not want x

Interpolation Algorithm Step 3

Remove variables not common to F and G

$6x + 4/5\, y = 2/5$

$4/5\, y - 2/5 = -6x$

$\Rightarrow$ $4/5\, y - 2/5$ is divisible by 6

$\Rightarrow$ $4/5\, y - 2/5 = 0 \pmod 6$

$\Rightarrow$ $4y - 2 = 0 \pmod{30}$

$4y - 2 = 0 \pmod{30}$ is an interpolant for (F, G)

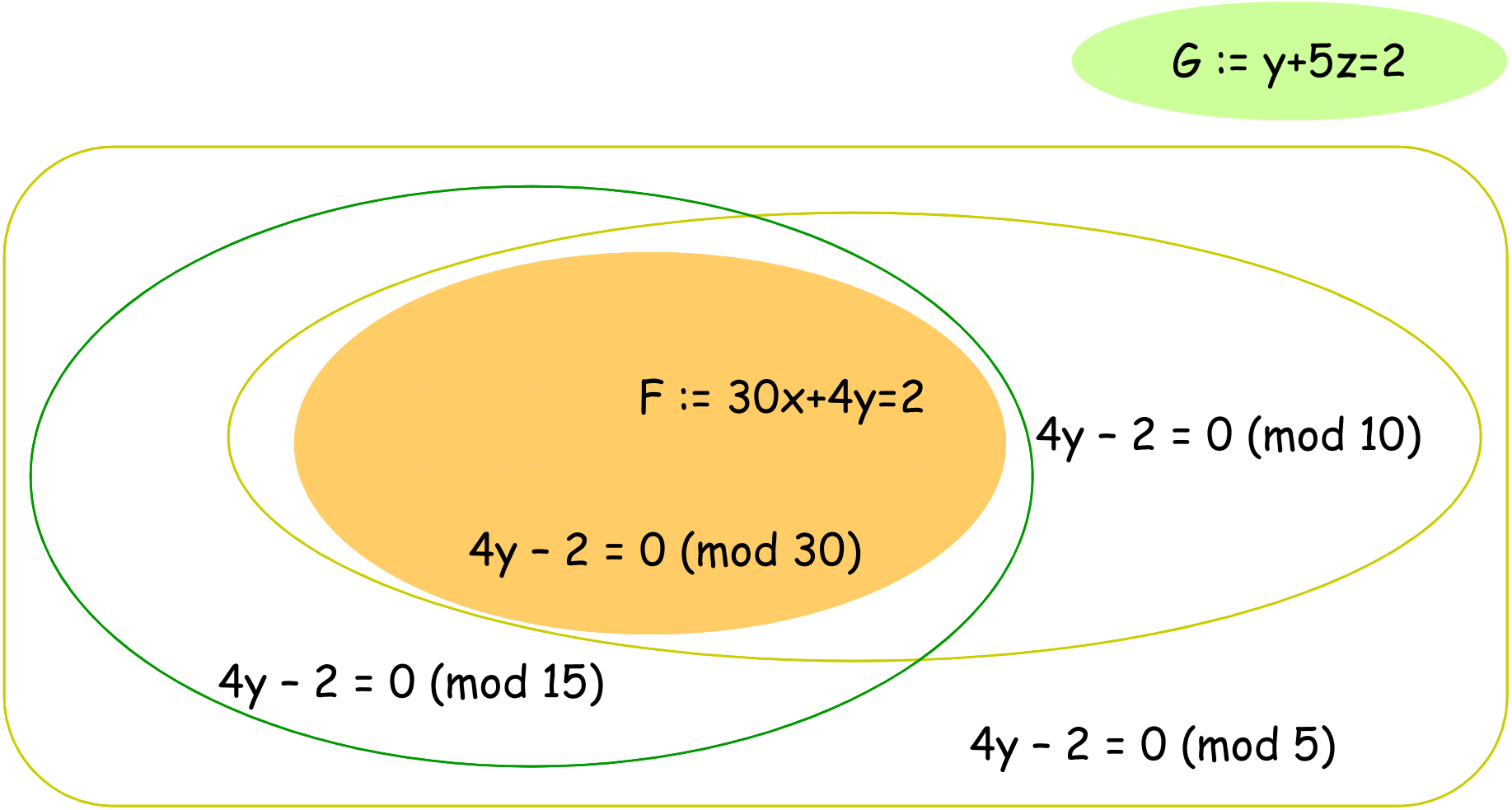
We have proved the correctness of above algorithm

Complexity of the Algorithm

- Obtain proof of unsatisfiability (step 1)
  - Polynomial time using Hermite Normal Form
  - Overall algorithm is polynomial time

- Multiple interpolants can be obtained

Multiple Interpolants



Outline

- Motivation
- Fast Boolean Satisfiability (SAT) Solvers
  - DPLL Algorithm
  - Related work
  - Negation Normal Form, hpgraph, vpgraph
  - SAT solving using hpgraph, vpgraph
- Interpolation for Subsets of Integer Linear Arithmetic
  - An application
- Summary

Predicate Discovery

```
void main()
{
  int x=0, y=0;

  while(*)
  {
    x = x + 4*nondet();
    y = y + 8*nondet();

    assert(x+y != 1);
    assert(x+y != 2);
    assert(x+y != 3);
  }
}
```

C program

Loop invariant:

x+y is divisible by 4
That is, x+y=0 (mod 4)

Such predicates can be found using our interpolation algorithms

Predica s

Existing state-of-the-art tools such as BLAST, SATABS, VCEGAR cannot verify these programs.

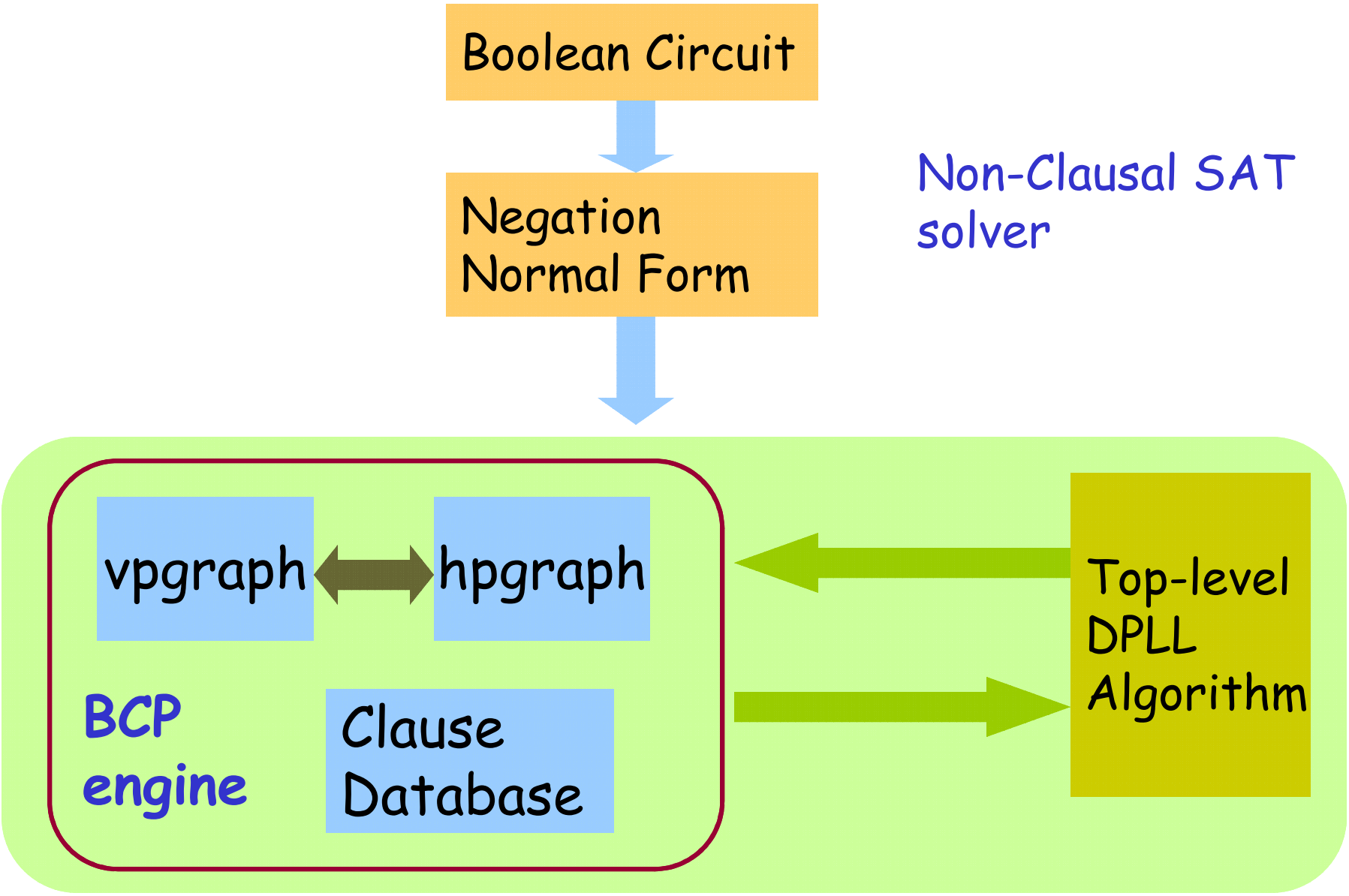
With the help of predicates found by our algorithms they can (VCEGAR).

Example		(secs)
ex1	$y = 1 \pmod 2$	2.72
ex2	$x + y = 0 \pmod 2$	0.83
ex4	$x + y + z = 0 \pmod 4$	0.95
ex5	$x=0 \pmod 4$, $y=0 \pmod 4$	1.1
ex6	$4x+2y+z = 0 \pmod 8$	0.93
ex7	$4x-2y+z = 0 \pmod{2^{22}}$	0.54
forb1	$x + y = 0 \pmod 3$	0.1

Outline

- Motivation
- Fast Boolean Satisfiability (SAT) Solvers
  - DPLL Algorithm
  - Related work
  - Negation Normal Form, hpgraph, vpgraph
  - SAT solving using hpgraph, vpgraph
- Interpolation for Subsets of Integer Linear Arithmetic
  - An application
- Summary

Summary


Boolean Circuit
Negation Normal Form
Non-Clausal SAT solver
vpgraph
hpgraph
BCP engine
Clause Database
Top-level DPLL Algorithm

Summary contd.

- Efficient Interpolation Algorithms
  - Integer linear equations
  - Integer linear congruences
  - Integer linear equations and disequations
- Proofs of unsatisfiability
  - Integer linear congruences
  - Integer linear equations and disequations

Thank you!

- Edmund Clarke
  - Exemplary advisor
- Randy Bryant, Aarti Gupta, Ken McMillan, Frank Pfenning
  - Excellent thesis committee members
- Peter Andrews
  - Math Logic classes and advice
- Daniel Kroening, Orna Grumberg, Constantinos Bartzis
  - Awesome collaborators
- NEC verification group for great summer internships

Questions