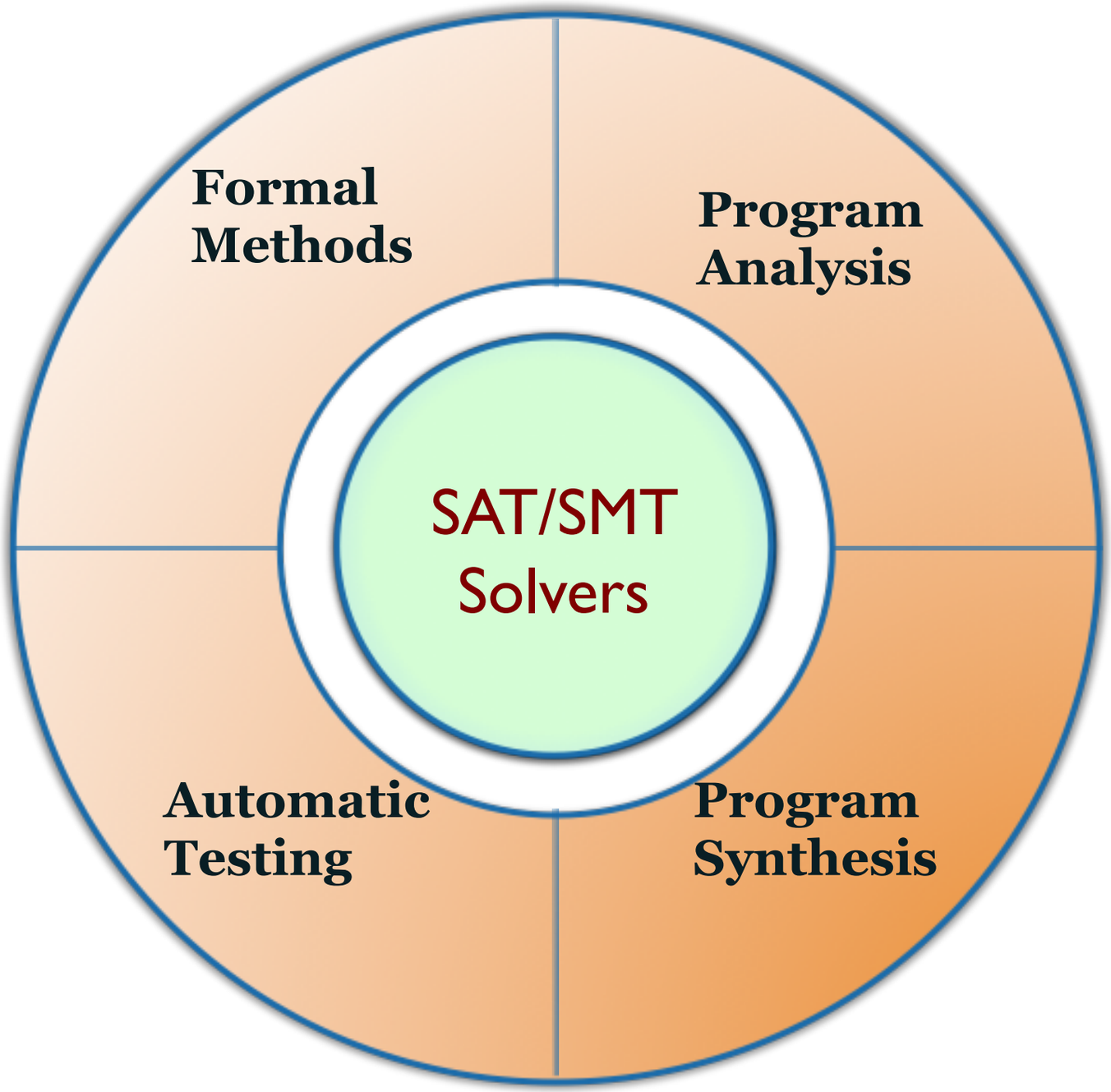
# SAT/SMT Solver Basics
# (and Impact of Community Structure on SAT Solver Performance)

Vijay Ganesh
Affiliation: University of Waterloo
October 28, 2014, Dagstuhl, Germany

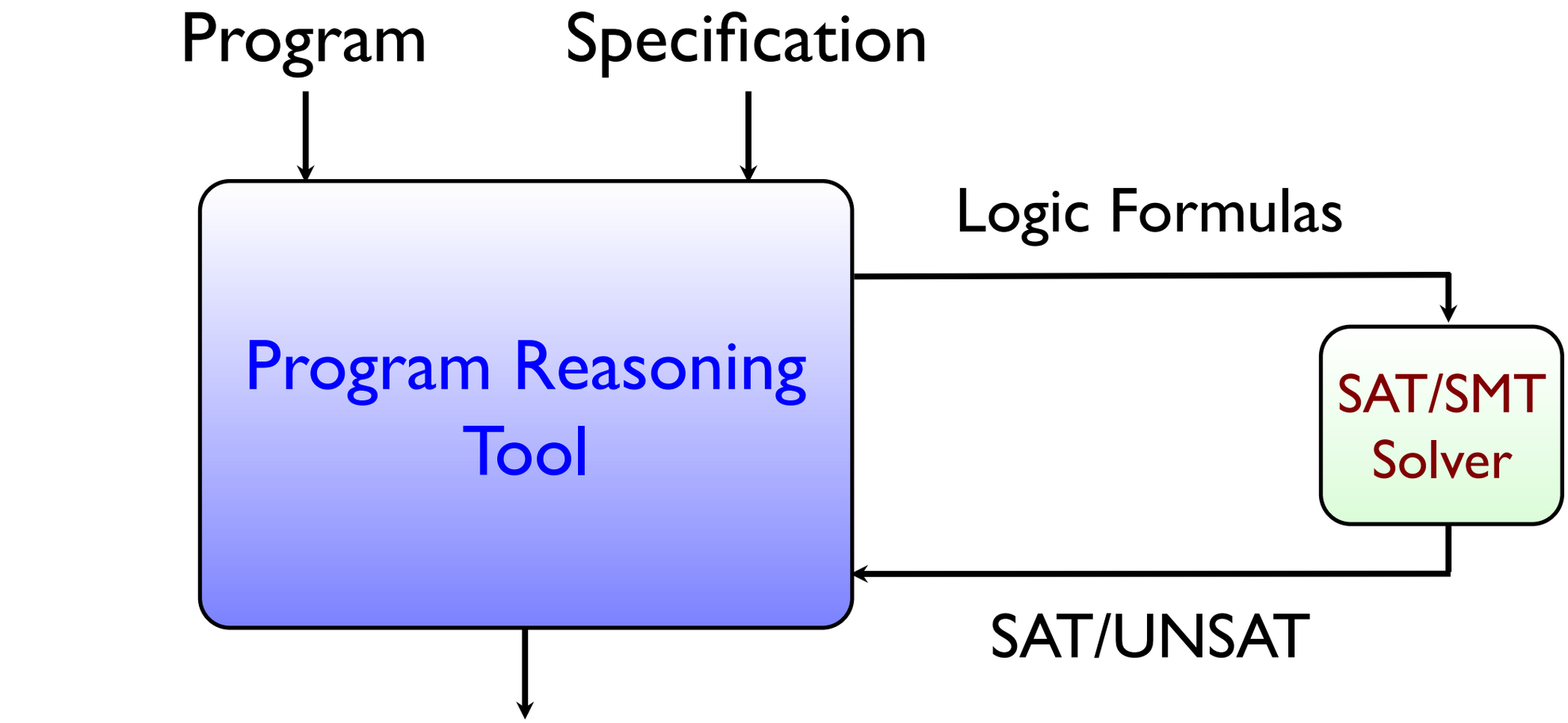
# Software Engineering & SAT/SMT Solvers

## An Indispensable Tactic for Any Strategy



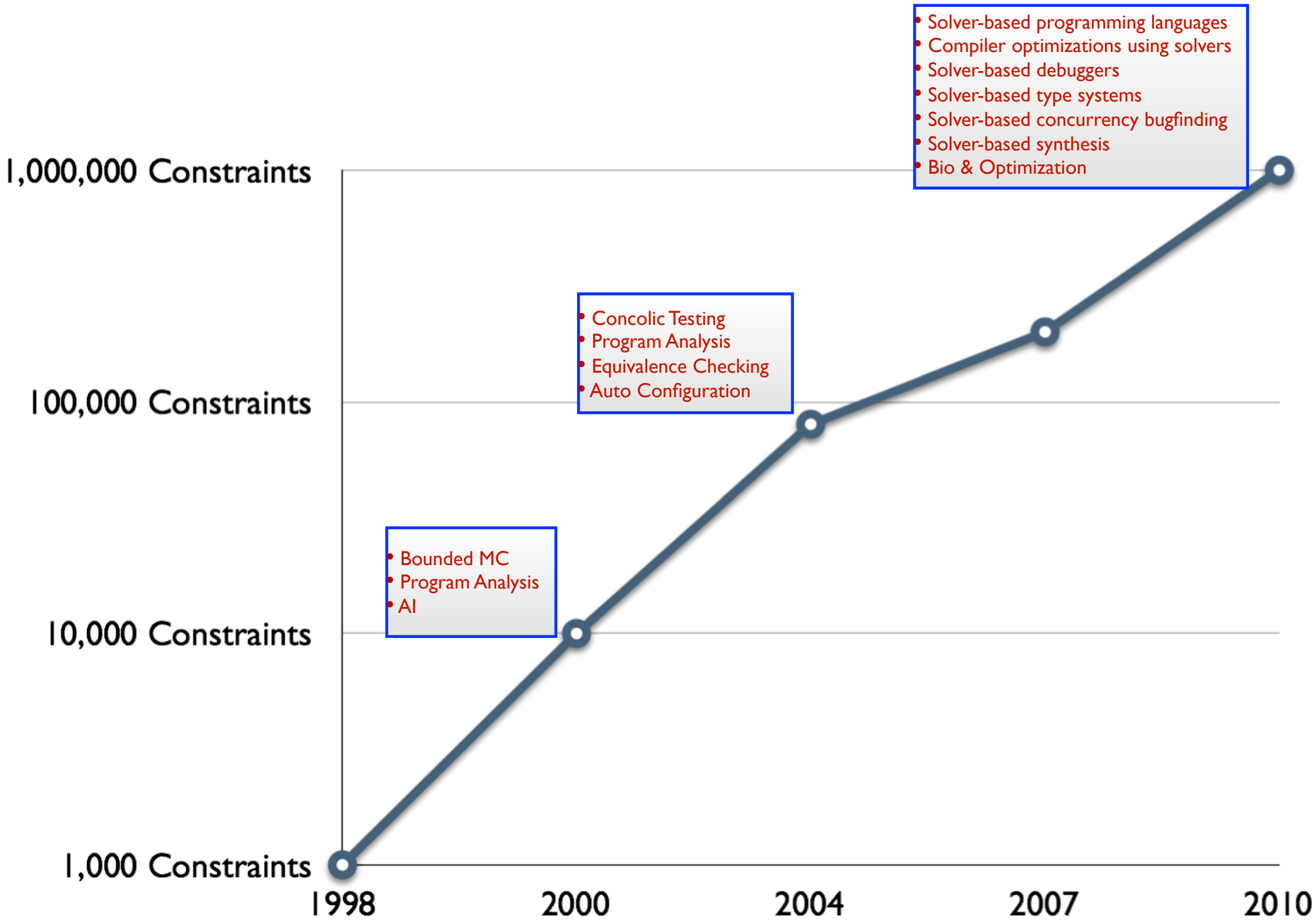
# Software Engineering using Solvers

## Engineering, Usability, Novelty



# SAT/SMT Solver Research Story

## A 1000x Improvement



# Talk Outline

## Topics covered on SAT Solvers

- Motivation for SAT/SMT solvers in software engineering
- High-level description of the SAT problem
- Key techniques used in SAT solvers
  - DPLL search and its shortcomings
  - Modern CDCL SAT solver: propagate (BCP), decide (VSIDS), conflict analysis, clause learn and backJump, clause deletion
  - Big lesson: learning from mistakes

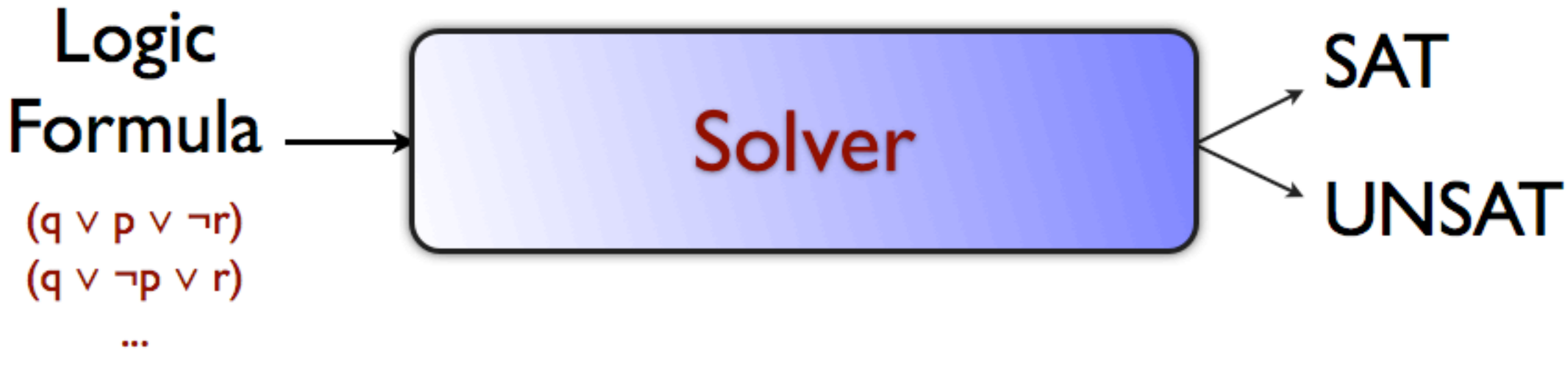
## Topics covered on SMT Solvers

- Modern SMT solvers and key techniques
  - Combination of theories
  - DPLL(T): Boolean reasoning over theories
  - Various forms of abstraction and refinement techniques
- Future directions

## Impact of Community Structure on SAT Solver Performance (presented at SAT 2014)

# What is a SAT/SMT Solver?

## Automation of Mathematical Logic




- Rich logics (Modular arithmetic, arrays, strings, non-linear arithmetic, theories with quantifiers, ...)
- From proof procedures to validity to satisfiability
- SAT problem is NP-complete, PSPACE-complete,...
- Practical, scalable, usable, automatic
- Enable novel software reliability approaches

# DPLL SAT Solver Architecture

## The Basic Solver

```
DPLL(Θ cnf, assign) {

Propagate unit clauses;

if "conflict": return FALSE;

if "complete assign": return TRUE;

"pick decision variable x";


return
        DPLL(Θ cnf | x=0, assign[x=0])
     || DPLL(Θ cnf | x=1, assign[x=1]);

}
```

Key Steps in a DPLL SAT Solver

Propagate (Boolean Constant Propagation)

- Propagate inferences due to unit clauses
- Most of the solving "effort" goes into this step

Detect Conflict

- Conflict: partial assignment is not satisfying

Decide (Branch)

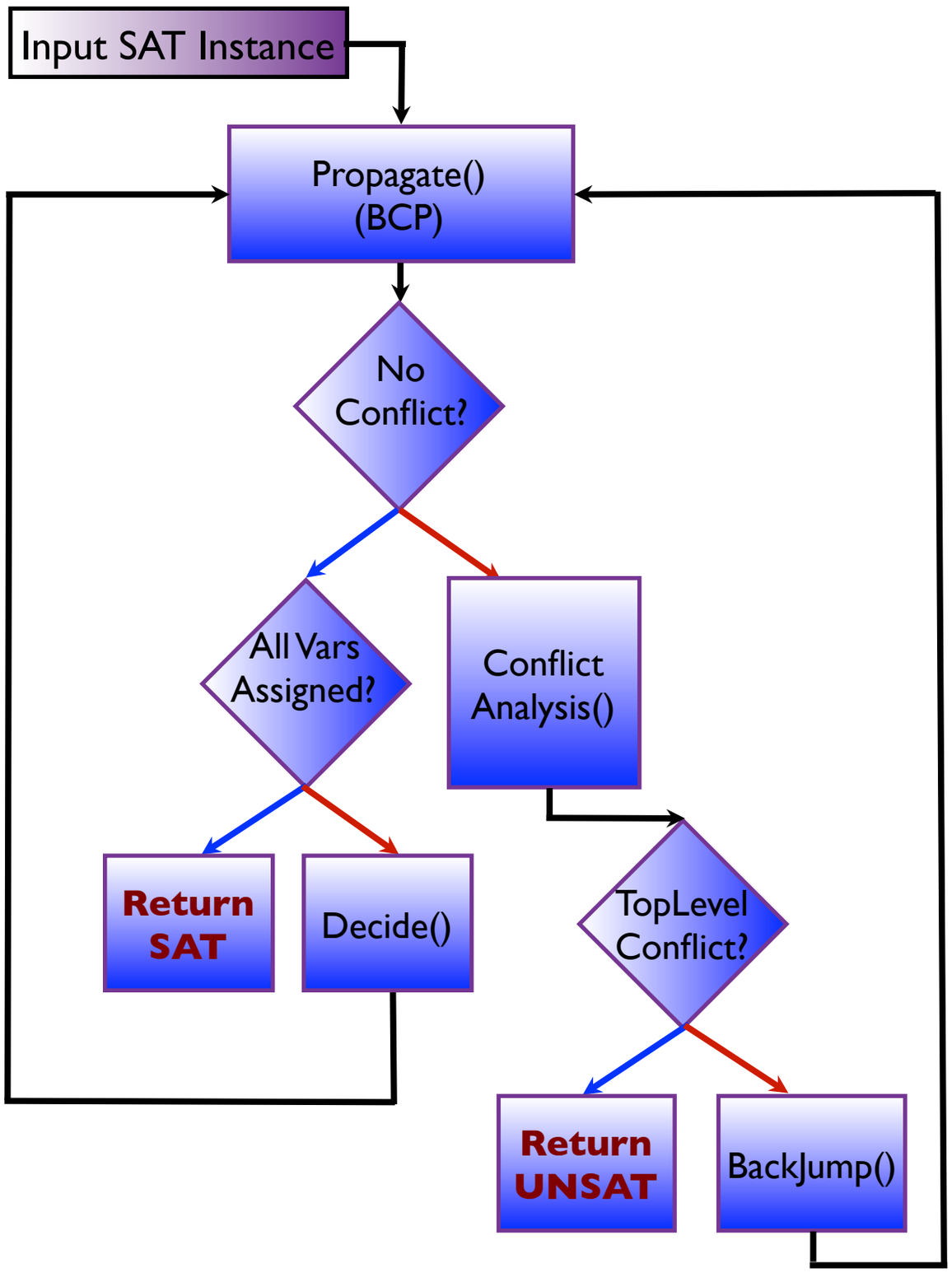
- Choose a variable & assign some value

Backtracking

- Implicitly done through the recursive call in DPLL

# Modern CDCL SAT Solver Architecture

## Key Steps and Data-structures




Key steps

- Decide()
- Propagate() (Boolean constant propagation)
- Conflict analysis and learning() (CDCL)
- Backjump()
- Forget()
- Restart()

CDCL: Conflict-Driven Clause-Learning

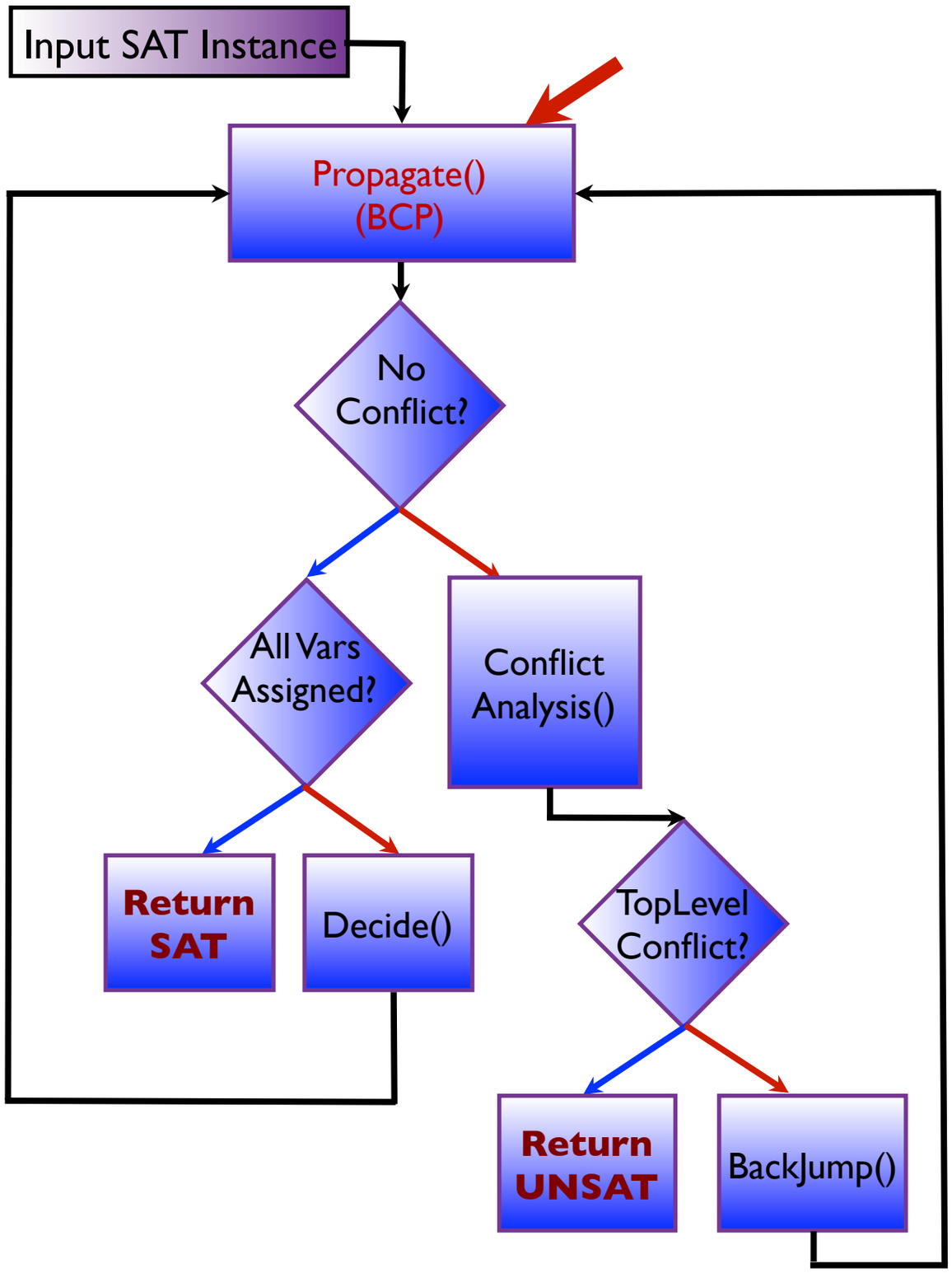
- Conflict analysis is a key step
- Results in learning a learnt clause
- Prunes the search space

Key data-structures (Solver state)

- Stack or trail of partial assignments (AT)
- Input clause database
- Conflict clause database
- Conflict graph
- Decision level (DL) of a variable

# Modern CDCL SAT Solver Architecture

## Propagate(), Decide(), Analyze/Learn(), BackJump()

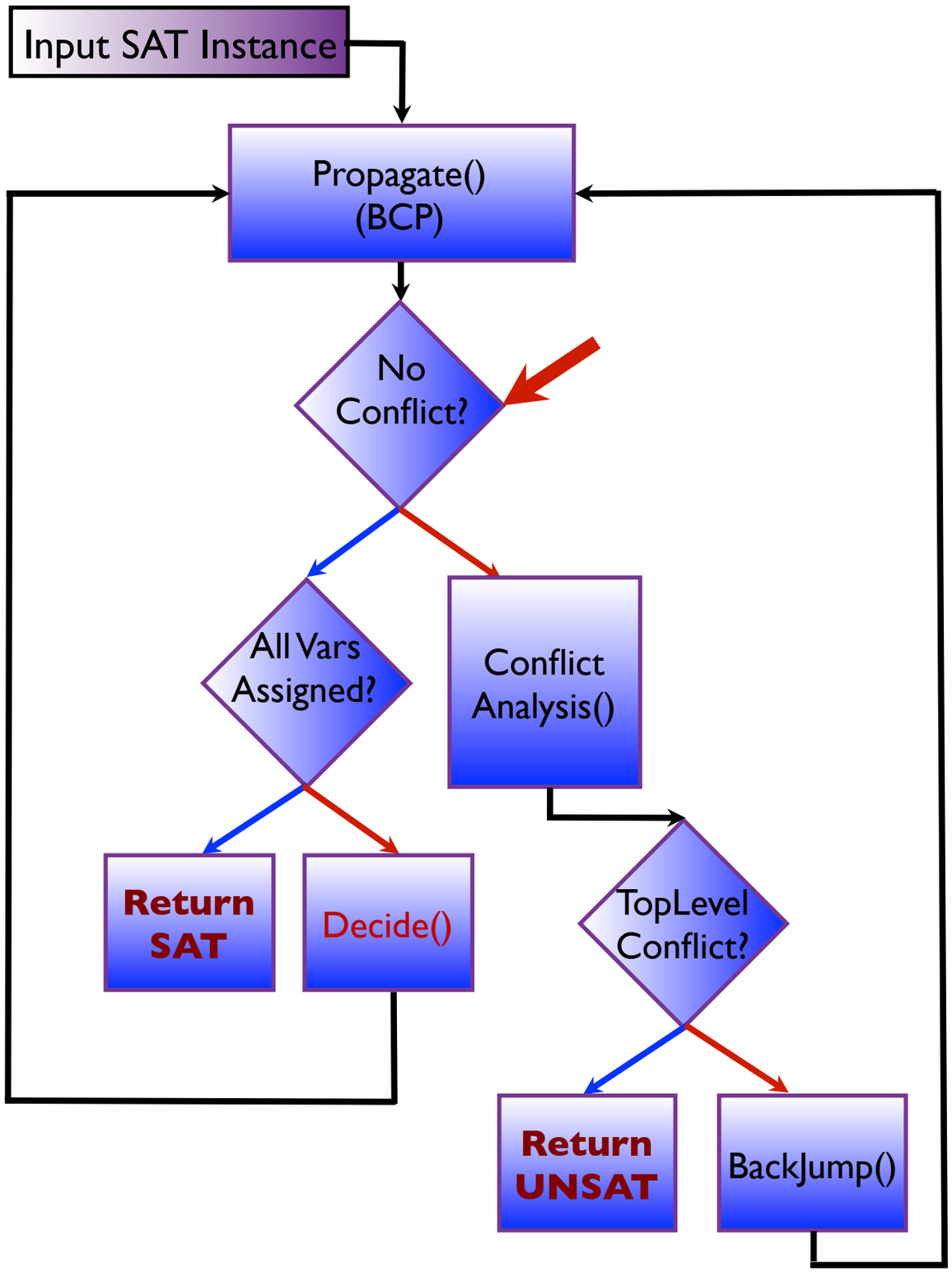



Propagate (Boolean Constant Propagation)

- Propagate inferences due to unit clauses
- Most time in solving goes into this

# Modern CDCL SAT Solver Architecture

## Propagate(), Decide(), Analyze/Learn(), BackJump()




Propagate (Boolean Constant Propagation)

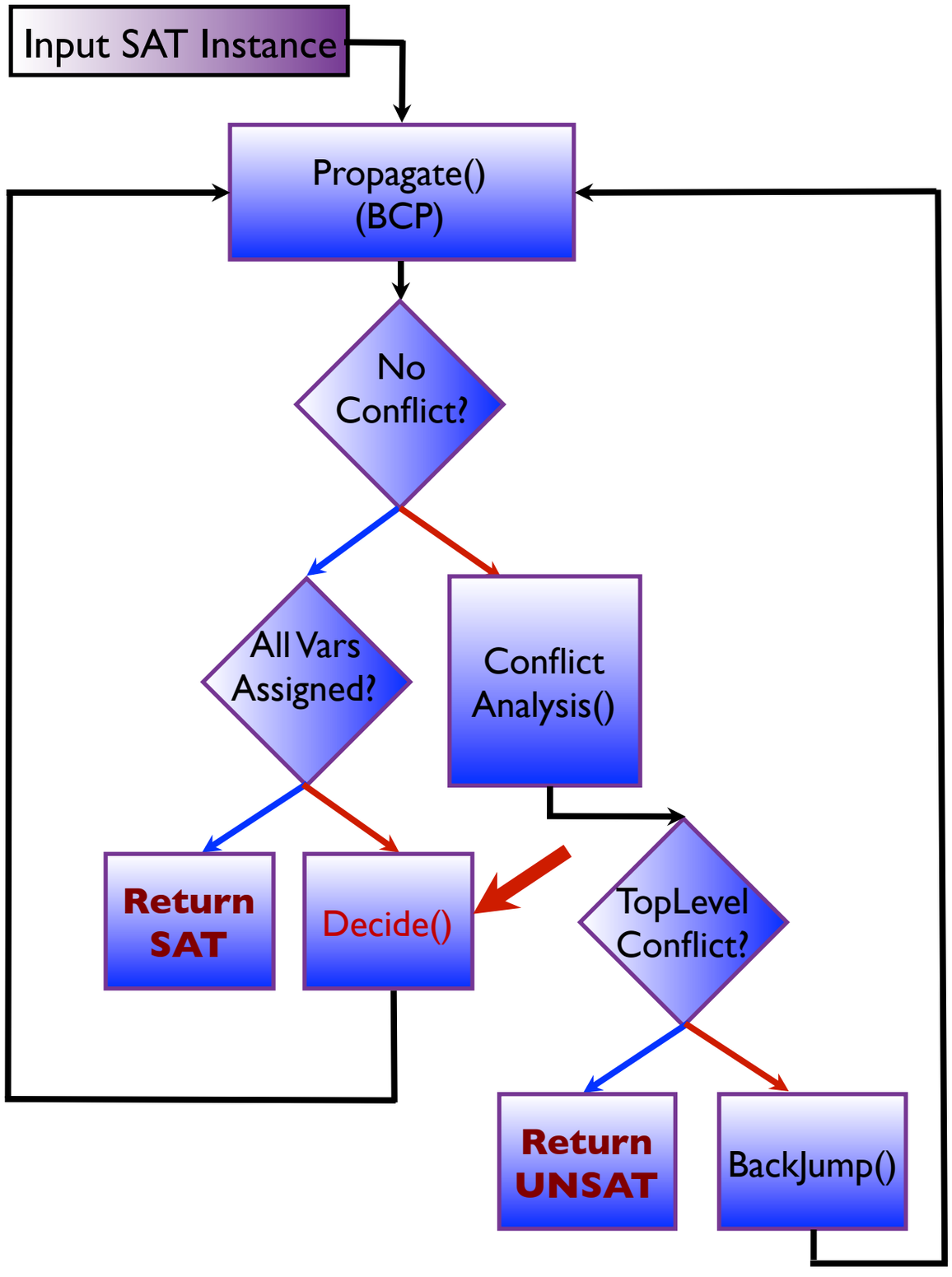
- Propagate inferences due to unit clauses
- Most time in solving goes into this

Detect Conflict?

- Conflict: partial assignment is not satisfying

# Modern CDCL SAT Solver Architecture

## Propagate(), Decide(), Analyze/Learn(), BackJump()




**Propagate (Boolean Constant Propagation)**

- Propagate inferences due to unit clauses
- Most time in solving goes into this

**Detect Conflict?**

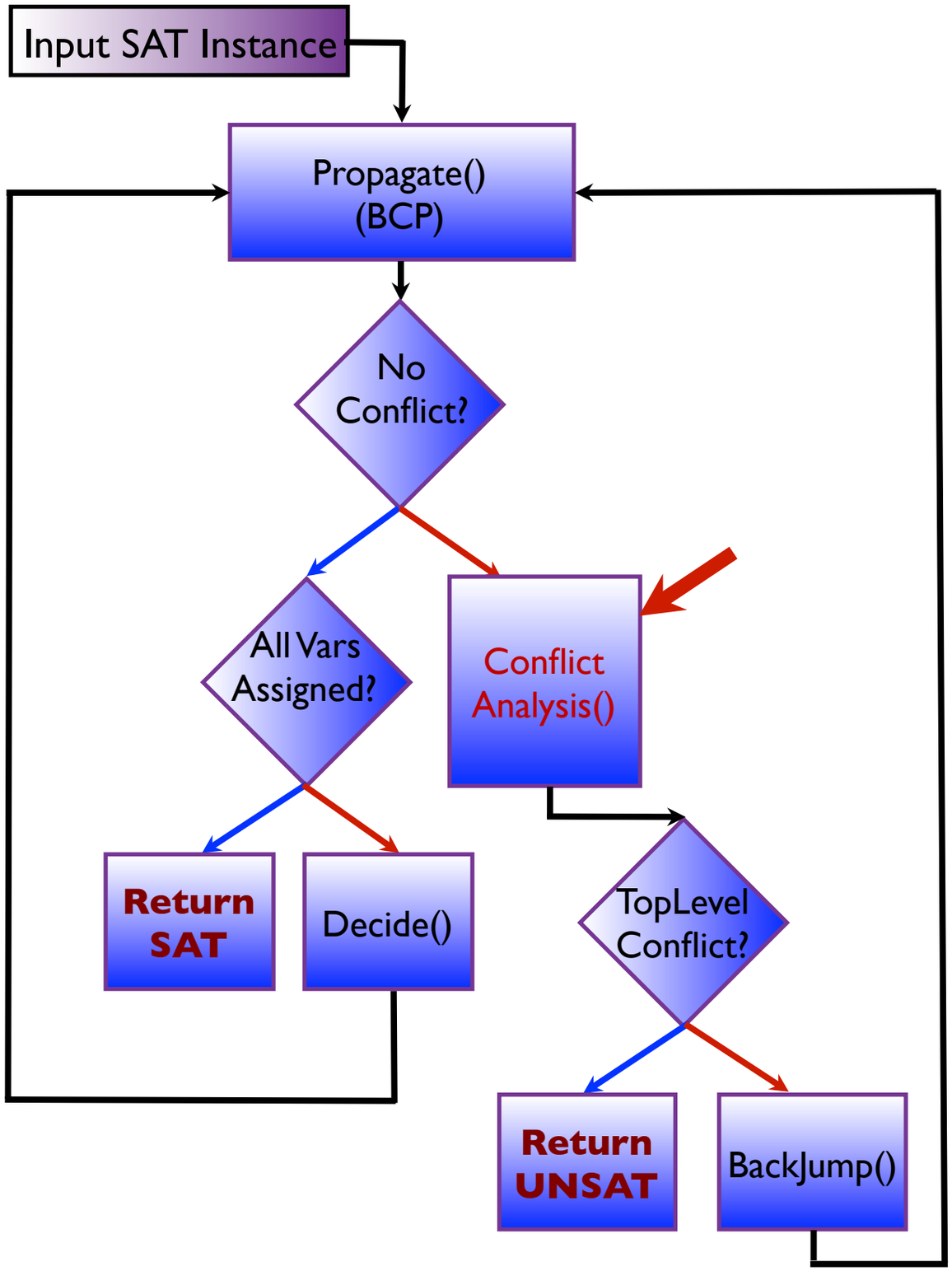
- Conflict: partial assignment is not satisfying

**Decide (Branch)**

- Choose a variable & assign some value (decision)
- Basic mechanism to do search
- Imposes dynamic variable order
- Least understood aspect of modern SAT solvers

# Modern CDCL SAT Solver Architecture

## Propagate(), Decide(), Analyze/Learn(), BackJump()




**Propagate**

- Propagate inferences due to unit clauses
- Most time in solving goes into this

**Detect Conflict?**

- Conflict: partial assignment is not satisfying

**Decide (Branch)**

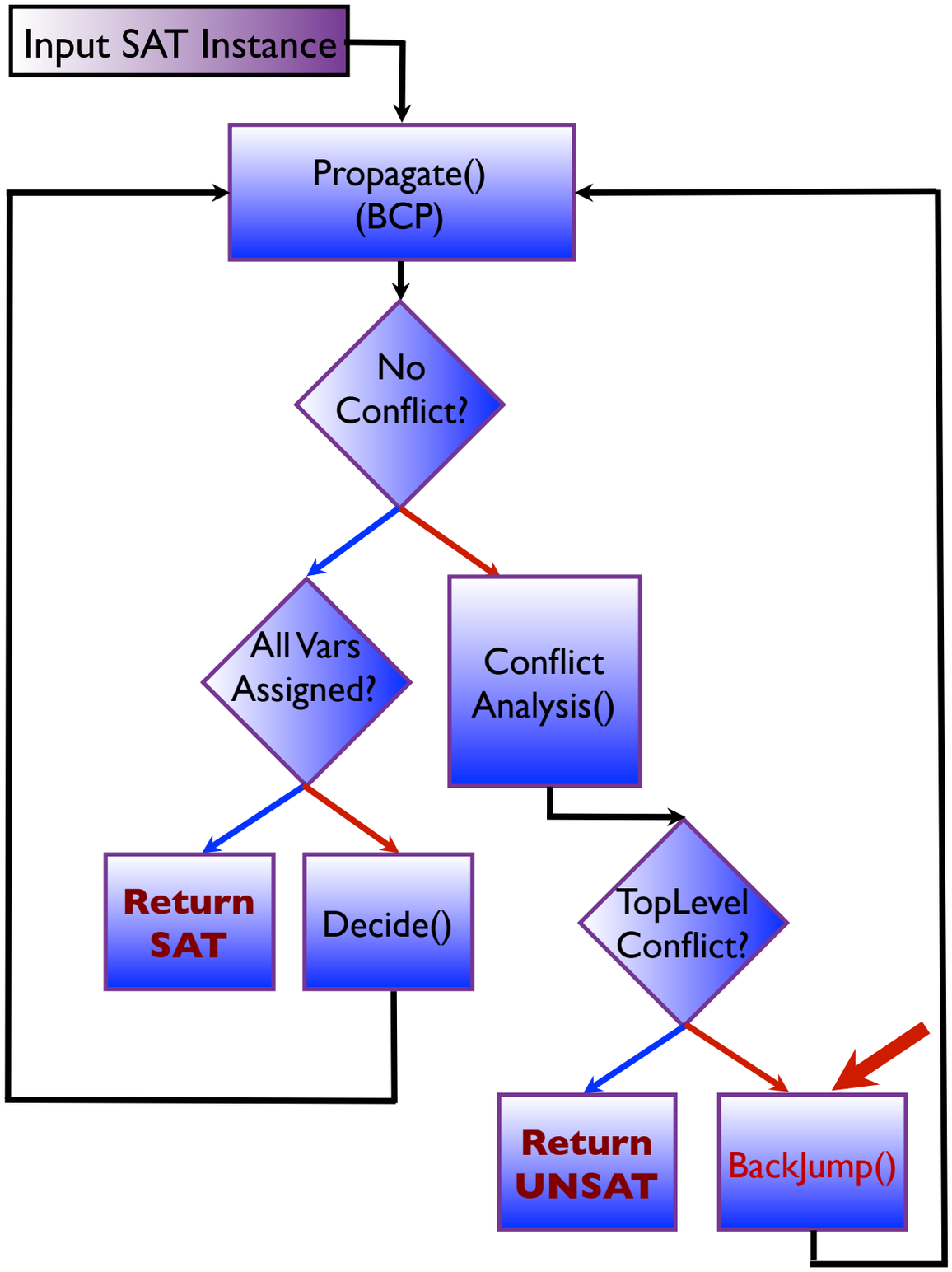
- Choose a variable & assign some value (decision)
- Each decision is a decision level
- Imposes dynamic variable order
- Decision Level (DL): variable ⇒ natural number

**Conflict analysis and clause learning**

- Compute implications that lead to conflict (analysis)
- Construct learnt clause that blocks the non-satisfying and a set of other 'no-good' assignments (learning)
- Originally developed by Marques-Silva & Sakallah(1996)

# Modern CDCL SAT Solver Architecture

## Propagate(), Decide(), Analyze/Learn(), BackJump()




**Propagate**

- Propagate inferences due to unit clauses
- Most time in solving goes into this

**Detect Conflict?**

- Conflict: partial assignment is not satisfying

**Decide**

- Choose a variable & assign some value (decision)
- Each decision is a decision level
- Imposes dynamic variable order
- Decision Level (DL): variable ⇒ natural number

**Conflict analysis and clause learning**

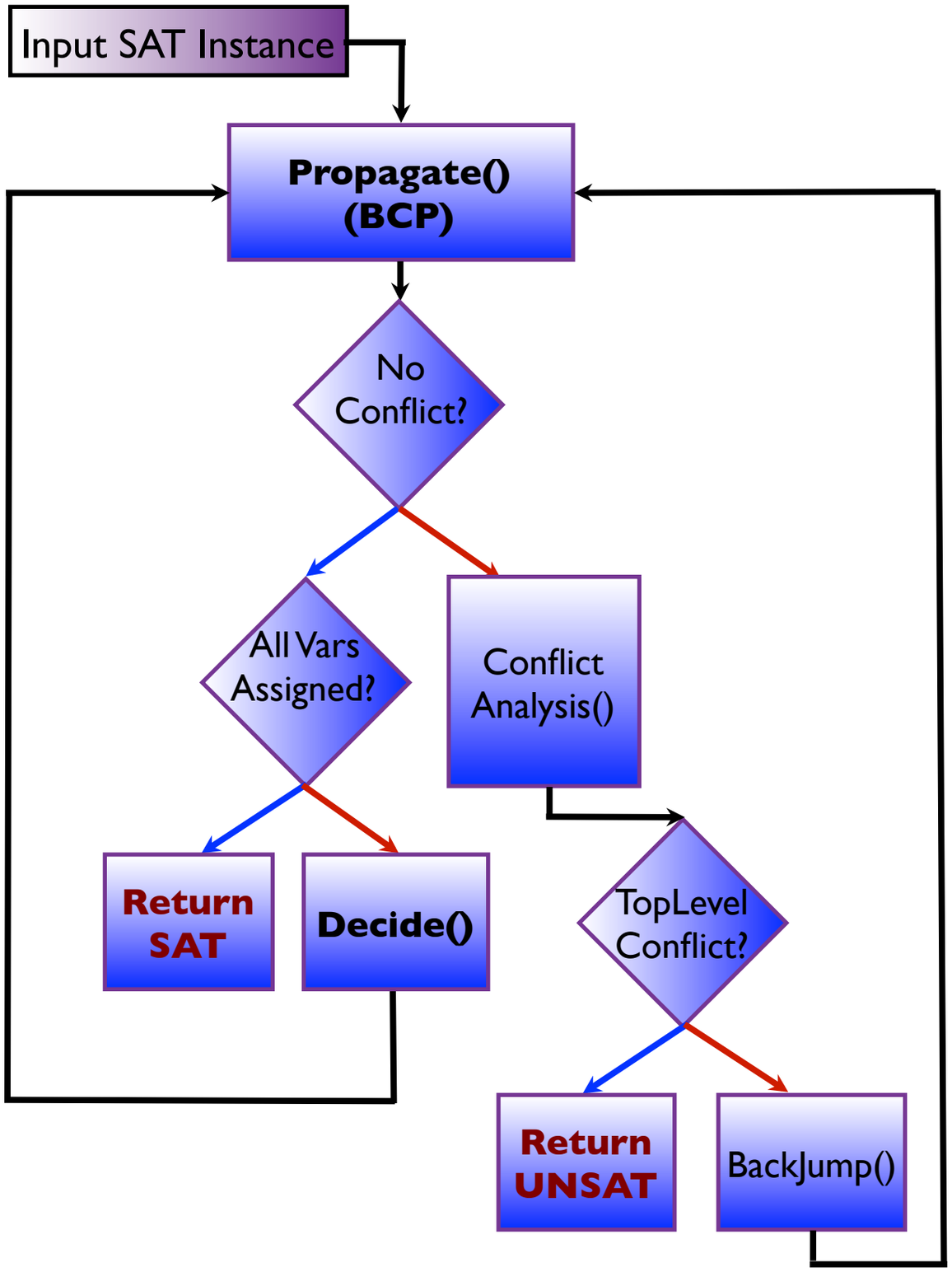
- Compute implications that lead to conflict (analysis)
- Construct learnt clause that blocks the non-satisfying and a set of other 'no-good' assignments (learning)

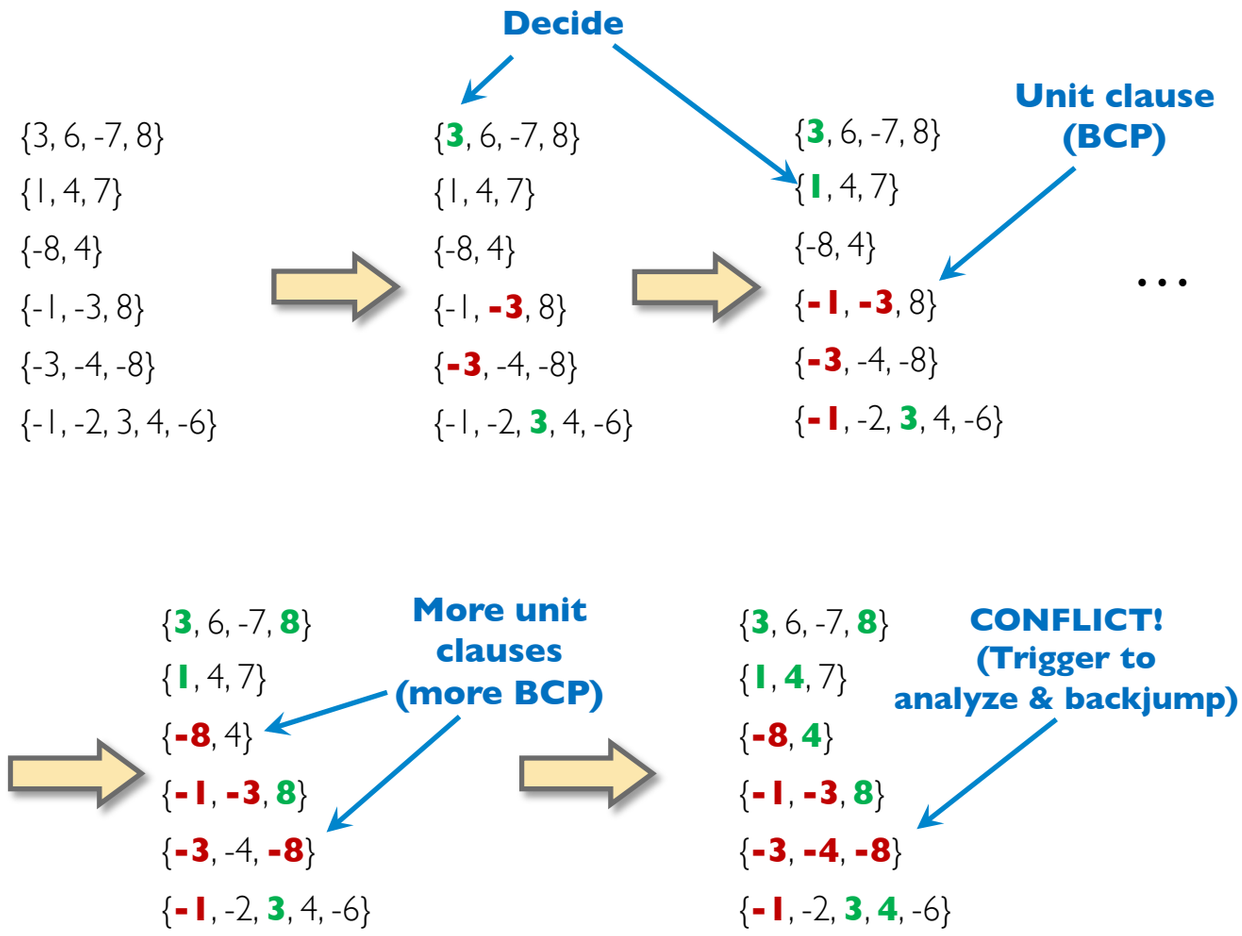
**Conflict-driven BackJump**

- Undo some decision(s) that caused no-good assignment
- Assign undone variable different value
- Go back several decision levels
- The technique is sound, i.e., no solutions are missed
- Backjump: Marques-Silva, Sakallah (1999)
- Backtrack: Davis, Putnam, Loveland, Logemann (1962)

# Modern CDCL SAT Solver Architecture

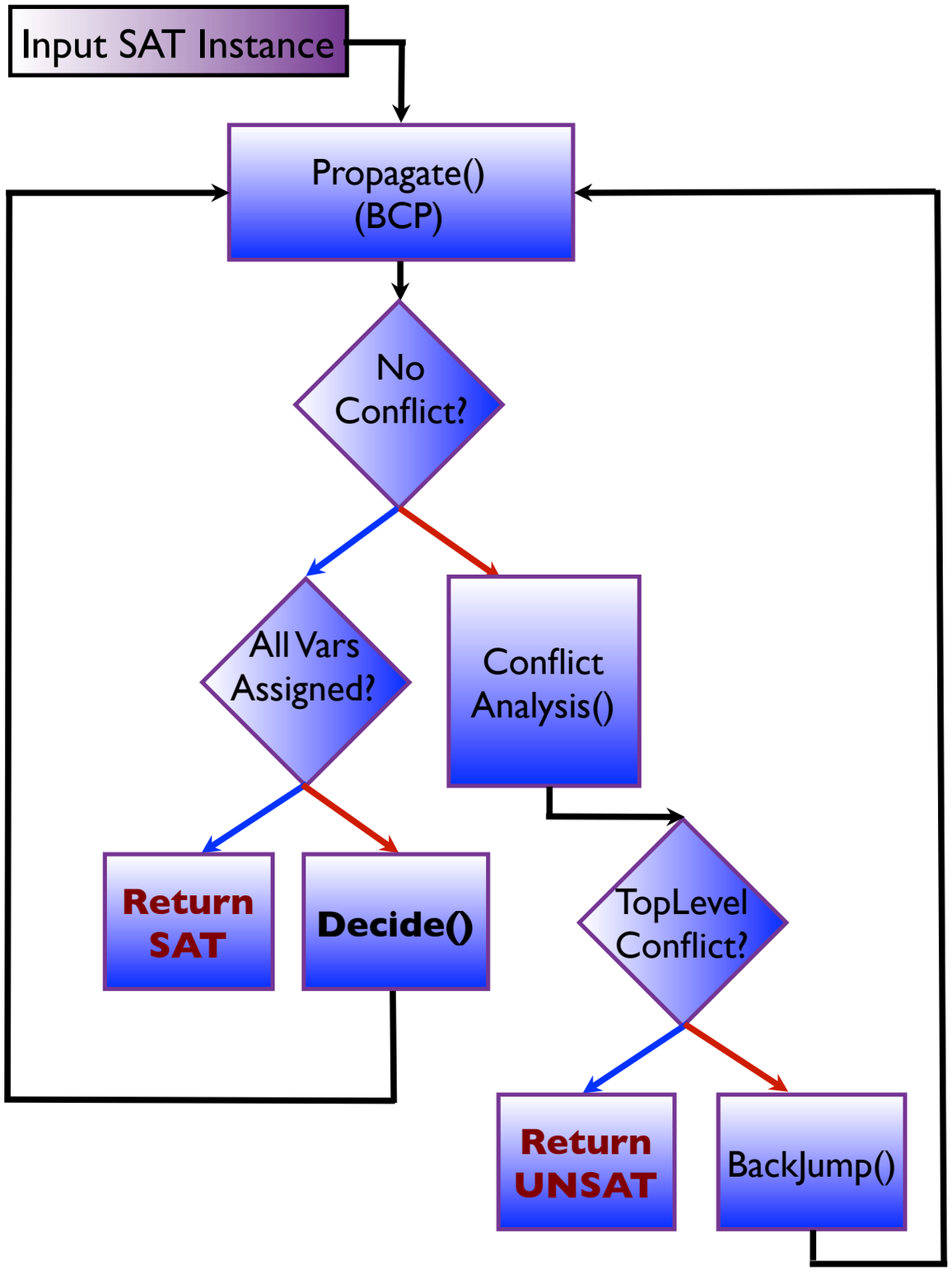
## Propagate(), Decide(), Analyze/Learn(), BackJump()



# Modern CDCL SAT Solver Architecture

## Decide() Details: VSIDS Heuristic




Decide() or Branching()

- Choose a variable & assign some value (decision)
- Imposes dynamic variable order

How to choose a variable

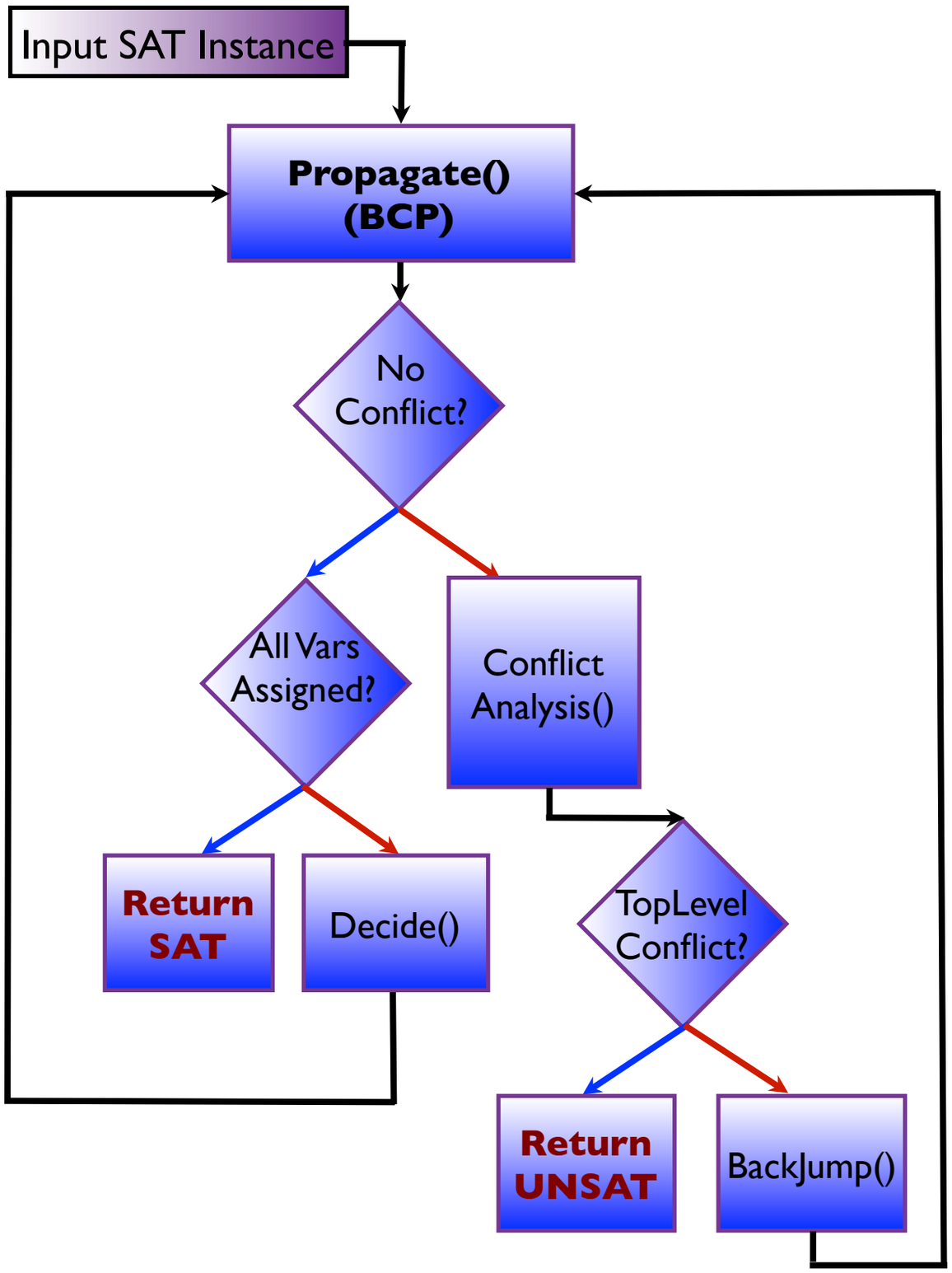
- VSIDS heuristics (zChaff team, DAC 2001)
- Each variable is assigned an activity value (a real number)
- Activity is bumped additively, if variable occurs in the "most recent" learnt clause
- At regular intervals, the activity of all variables is decayed by multiplying by a constant less than 1
- The "next decision variable" is the variable with highest activity
- One of the least understood aspects of SAT solving

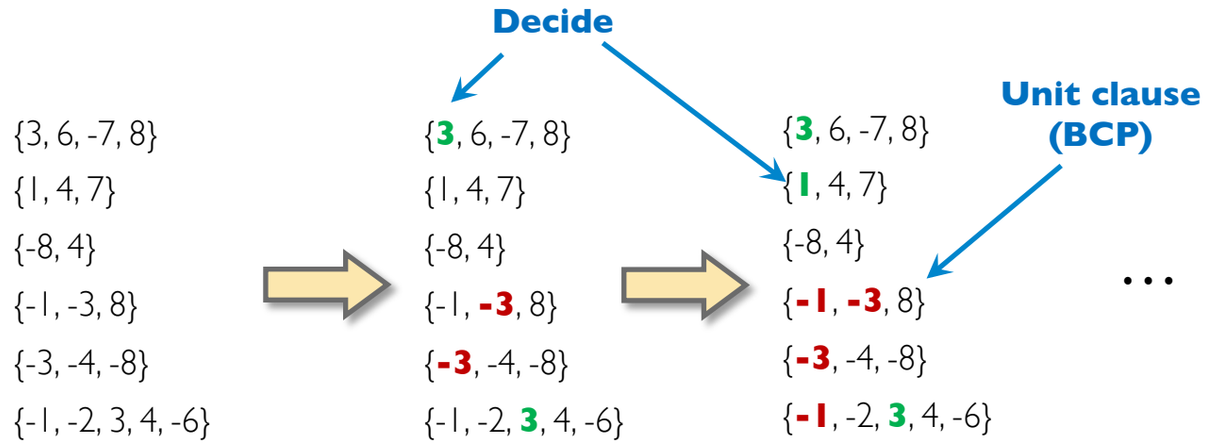
Why does VSIDS work so well (ongoing work)

- Imposes dynamic variable order
- Behaves like a low-pass filter or an Exponential Moving Average
- Favors variables that are recently persistently in "conflict"
- More work required to understand the power of VSIDS

# Modern CDCL SAT Solver Architecture

## Propagate() Details: Two-watched Literal Scheme




| Watched Literal | Watcher List |
|---|---|
| -1 | {-1, -3, 8},... |
| -3 | {-1, -3, 8},... |
| ... | ... |

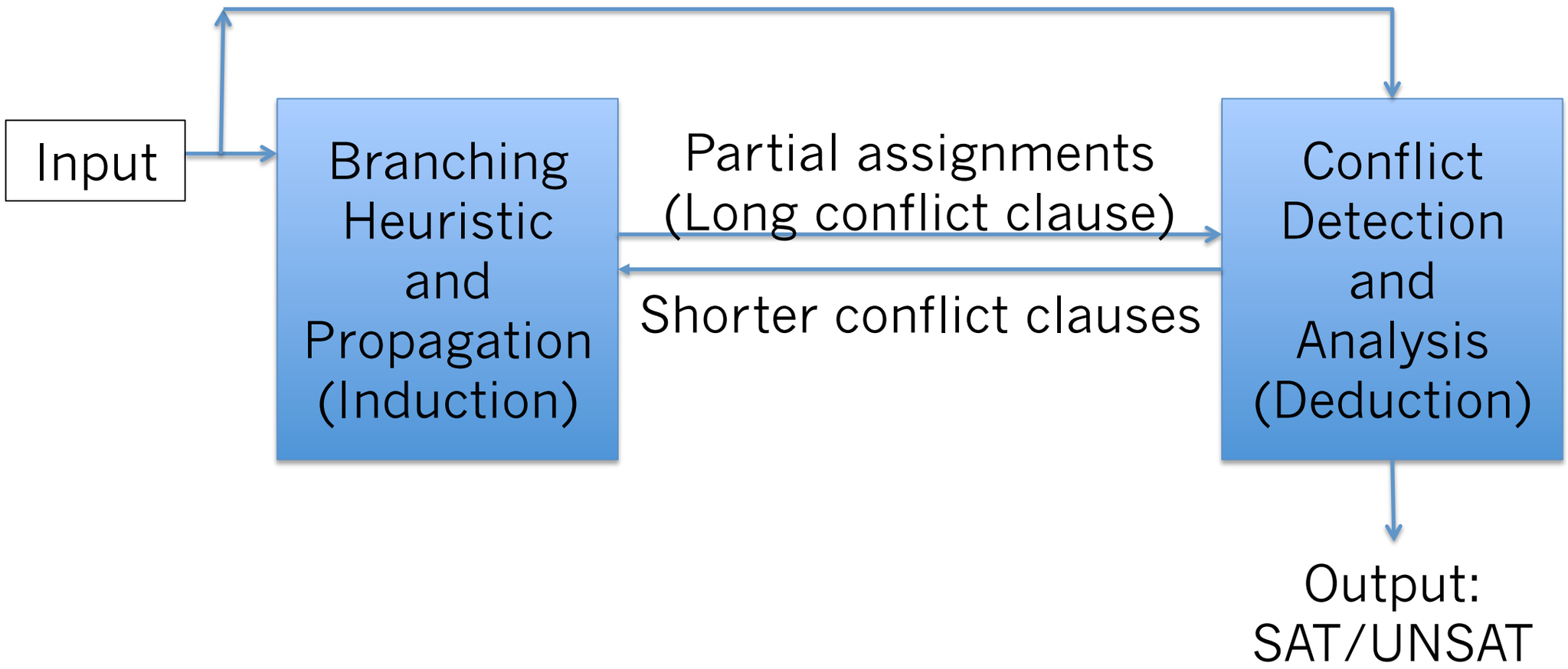
| Watched Literal | Watcher List |
|---|---|
| -1 | {-1, -3, 8},... |
| **-3** | ... |
| 8 | {-1, -3, 8},... |
| ... | ... |

| Watched Literal | Watcher List |
|---|---|
| **-1** | ... |
| **-3** | ... |
| 8 | {-1, -3, 8},... |
| ... | ... |

The constraint propagates 8

# A Model for CDCL Solvers

## CDCL Solvers: Induction and Deduction with Feedback



# Rest of the Talk

## Topics covered on SAT Solvers

- Motivation for SAT/SMT solvers in software engineering
- High-level description of the SAT problem
- Key techniques used in SAT solvers
  - DPLL search and its shortcomings
  - Modern CDCL SAT solver: propagate (BCP), decide (VSIDS), conflict analysis, clause learn and backJump, clause deletion
  - Big lesson: learning from mistakes

## Topics covered on SMT Solvers

- Modern SMT solvers and key techniques
  - Combination of theories
  - DPLL(T): Boolean reasoning over theories
  - Various forms of abstraction and refinement techniques
- Future directions

## Impact of Community Structure on SAT Solver Performance (presented at SAT 2014)

# The SMT Problem

## Satisfiability over Boolean Combination of First-order Theories

- SMT stands for Satisfiability Modulo Theories

- An SMT formula is a Boolean combination of formulas over first-order theories

- Example of SMT theories include bit-vectors, arrays, integer and real arithmetic, strings,…

- The satisfiability problem for these theories is typically hard in general (NP-complete, PSPACE-complete,…)

# Motivations for SMT Solvers
## Aren't SAT Solvers Enough?

- Program semantics are easily expressed over these theories

- Many software engineering problems can be easily reduced to the SAT problem over first-order theories

- Reduction to Boolean SAT not always a good option because of "blowup" during reduction

- SMT expressions have structure that can be more easily detected and solved using powerful heuristics, than by analyzing the equivalent Boolean representation

# Modern SMT Solvers
## Key Design Concepts

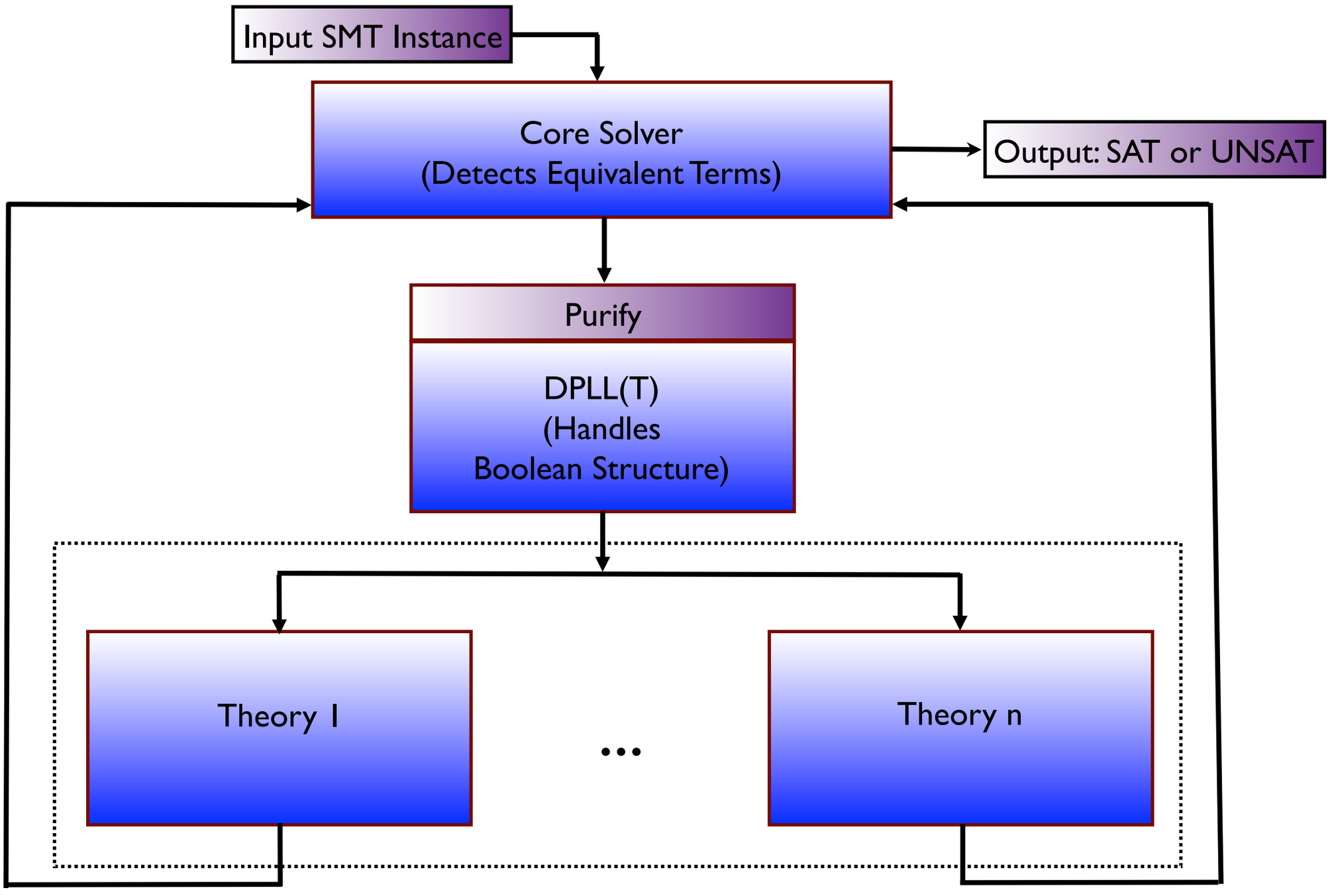
- Combination of theories (Nelson-Oppen, Shostak, Ghilardi,…)

- DPLL(T): lazily solve Boolean combination of literals from "different" theories

- Combining "lazy (abstraction and refinement)" and "eager" techniques to solve SAT problem for individual theories

# Standard-issue SMT Solver Architecture

## Combination of theories & DPLL(T)



# Standard-issue SMT Solver Architecture

# Combination of theories: Nelson-Oppen

## Problem Statement

- Combine theory solvers to obtain a solver for a union theory

## Motivation

- Software engineering constraints span many "natural" first-order theories
- Some first-order theories do have "practically" efficient solvers
- Modular composition of such decision procedures is often deemed more efficient than a monolithic solver (Caution: not always true)

## How

- Setup communication between individual theory solvers
- Communication over shared predicates (equality in the case of Nelson-Oppen)
- Soundness, completeness and termination

# Standard-issue SMT Solver Architecture

## DPLL(T)

### Problem Statement

- Efficiently handle the Boolean structure of the input formula

### Basic Idea

- Use a SAT solver for the Boolean structure & check assignment consistency against a T-solver
- T-solver only supports conjunction of T-literals

### Improvements (Nuiwenheus, Oliveras, and Tinelli 2006)

- Check partial assignments against T-solver
- Do theory propagation (similar to SAT solvers)
- Conflict analysis guided by T-solver & generate conflict clauses (similar to SAT solvers)
- BackJump (similar to SAT solvers)

# Modern SMT Solvers
## Key Design Concepts

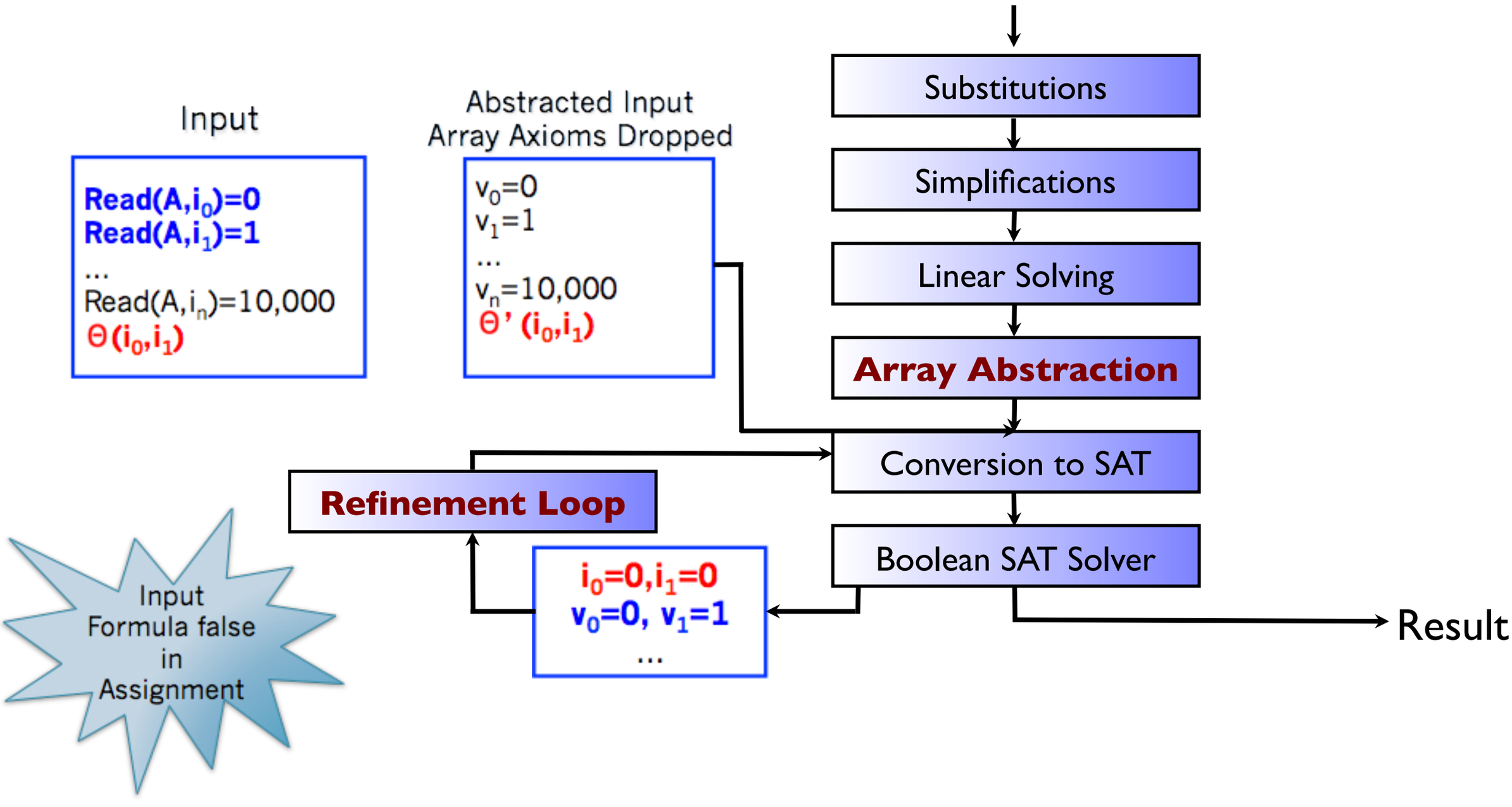
- Combination of theories (Nelson-Oppen, Shostak, Ghilardi,…)

- DPLL(T): lazily solve Boolean combination of literals from "different" theories

- Combining "lazy (abstraction and refinement)" and "eager" techniques to solve SAT problem for individual theories

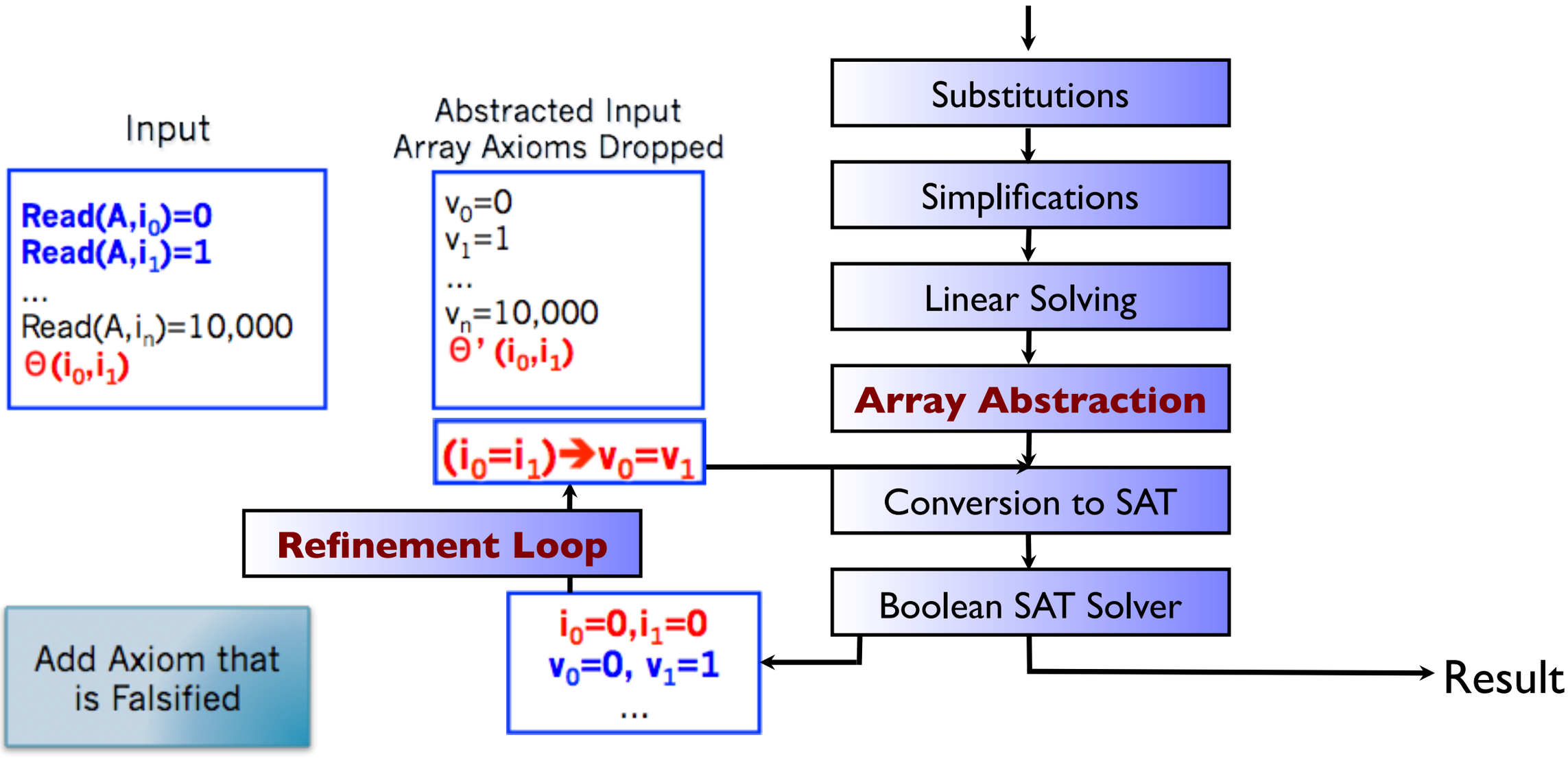
# How STP Works

## Eager for Bit-vectors, Lazy for Arrays



# How STP Works

## Abstraction-refinement for Array-reads



# An Incomplete History of SAT and SMT Solvers

| Category | Research Project | Researcher/Institution/Time Period |
|---|---|---|
| Theorem Proving (very early roots of decision procedures) | NuPRL<br>Boyer-Moore Theorem Prover<br>ACL2<br>PVS Proof Checker | Robert Constable / Cornell / 1970's-present<br>Boyer & Moore / UT Austin / 1970's-present<br>Moore, Kauffmann et al. / UT Austin / 1980's - present<br>Natarajan Shankar / SRI International / 1990's-present |
| SAT Solvers | DPLL<br>GRASP (Clause learning and backjumping)<br>Chaff & zChaff<br>MiniSAT | Davis, Putnam, Logemann & Loveland / 1962<br>Marques-Silva & Sakallah / U. Michigan / 1996-2000<br>Zhang, Malik et al. / Princeton / 1997-2002<br>Een & Sorensson / 2005 - present |
| Combinations | Simplify<br>Shostak<br>ICS<br>SVC, CVC, CVC-Lite, CVC3 ...<br>Non-disjoint theories | Nelson & Oppen / DEC and Compaq / late 1980s<br>Shostak / SRI International / late 1980's<br>Ruess & Shankar / SRI International / late 1990's<br>Barrett & Dill / Stanford U. / late 1990's<br>Tinelli, Ghilardi, G.,.. / 2000 - 2008 |
| DPLL(T) | Barcelogic and Tinelli group | Oliveras, Nieuwenhuis & Tinelli / UPC and Iowa / 2006 |
| Under/Over Approximations | UCLID<br>STP | Seshia & Bryant / CMU / 2004 - present<br>Ganesh & Dill / Stanford / 2005 - present |
| Widely-used SMT Solvers | Z3<br>CVC4<br>OpenSMT<br>Yices<br>MathSAT<br>STP<br>UCLID | DeMoura & Bjorner / Microsoft / 2006 - present<br>Barrett & Tinelli / NYU and Iowa / early 2000's - present<br>Bruttomesso / USI Lugano / 2008 - present<br>Deuterre / SRI International / 2005 - present<br>Cimatti et al. / Trento / 2005 - present<br>Ganesh / Stanford & MIT / 2005 - present<br>Seshia / CMU & Berkeley / 2004 - present |

# Rest of the Talk

## Topics covered on SAT Solvers

- Motivation for SAT/SMT solvers in software engineering
- High-level description of the SAT problem
- Key techniques used in SAT solvers
  - DPLL search and its shortcomings
  - Modern CDCL SAT solver: propagate (BCP), decide (VSIDS), conflict analysis, clause learn and backJump, clause deletion
  - Big lesson: learning from mistakes

## Topics covered on SMT Solvers

- Modern SMT solvers and key techniques
  - Combination of theories
  - DPLL(T): Boolean reasoning over theories
  - Various forms of abstraction and refinement techniques
- Future directions

**Impact of Community Structure on SAT Solver Performance (presented at SAT 2014)**

# Problem Statement

## Why are SAT Solvers efficient for Industrial Instances

- Conflict-driven clause learning (CDCL) Boolean SAT solvers are remarkably efficient for large industrial instances

- This is true for industrial instances from a diverse set of applications

- These instances may have tens of millions of variables and clauses

# Motivation

## Why are SAT Solvers efficient for Industrial Instances

- This question has stumped both theoreticians and practitioners alike

- May lead to better solvers and predictors

- In turn may lead to better solver applications

# Previous Work

## Why are SAT Solvers efficient for Industrial Instances

- Results on the practical front to-date
  - Phase-change at clause-var ratio of 4.25 for randomly-generated instances [GW00]
  - Empirically observed *only* for randomly-generated instances
  - Doesn't explain the power of CDCL solvers for industrial instances
  - Industrial SAT instances community structure by Ansotegui, Levy et al.[AL12, AL13]
  - Large successful predictive model by Xu, Hoos et al. Basis for a machine learning based predictor [XHHL08]

# Structure in Industrial Instances

## Why are SAT Solvers efficient for Industrial Instances

- SAT solvers exploit structure Inherent to SAT instances

- Relevant questions

  1. What structure of industrial instances do solvers exploit?
  2. What evidence connects so-called structure and solver performance?
  3. How does the solver go about exploiting this structure?

- In this talk we discuss answers to Questions 1 and 2

# Community Structure and SAT Solver Performance

## Take Home Message

- Take-home Message
  - Community structure (the quality of which is measured using a metric called Q) of SAT instances strongly affect solver performance
- Working towards
  - Small predictive set of features that forms the basis for a complete explanation (we don't have it yet)

# Community Structure and SAT Solver Performance

## Empirical Results

- Result #1: Hard random instances have low Q ($0.05 \leq Q \leq 0.13$)

- Result #2: Number of communities and Q of SAT instances are more predictive of CDCL solver performance than other measures we considered

- Result #3: Strong correlation between community structure and LBD (Literal Block Distance) in Glucose solver

# Community Structure and SAT Solver Performance

## Graphs and their Community Structure

- Informal definition of community in a graph
  - View SAT instances as variable-incidence graphs
  - A community is sub-graph that has more internal edges than outgoing ones
  - A community structure characterizes how "well clustered" a graph is

# A note on Community Structure in Graphs

## Applications in Various Domains

- Community structure [GN03,CNM04,OL13] is used to study all kinds of complex networks.
  - Social Networks, e.g. Facebook
  - Internet
  - Protein networks
  - Neural network of the human brain
  - Citation graphs
  - Business networks
  - Populations
  - And more recently, the graph of logical formulas

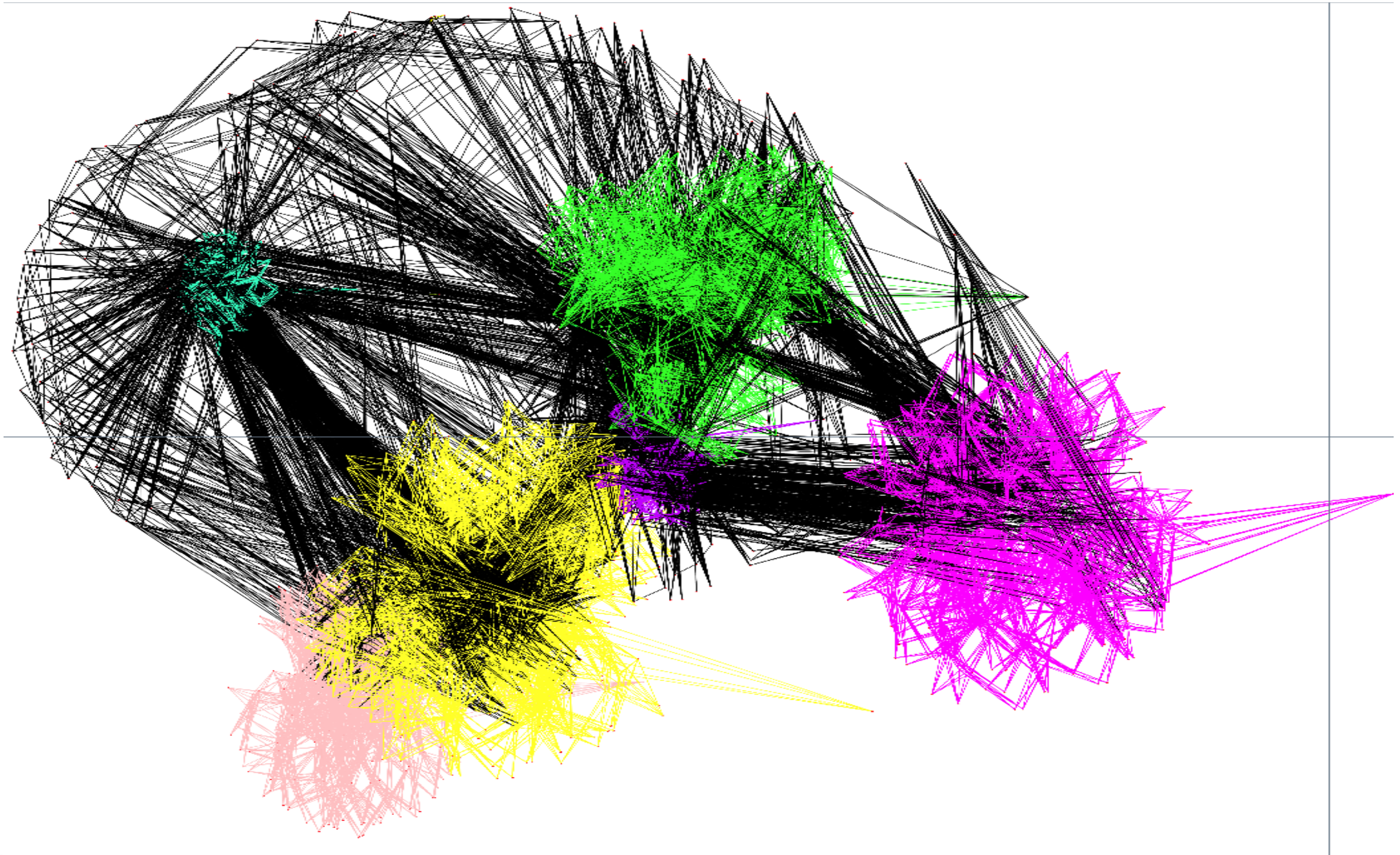
# Modularity and Communities in Graphs

## Community Structure in Graphs

- Modularity (Q) of graph lies between 0 and 1
- Q measures quality
  - How separable are the communities in the graph
  - Higher Q implies "good community structure", i.e. *highly separable communities*
  - Lower Q implies "bad community structure", i.e. *one giant hairy ball*

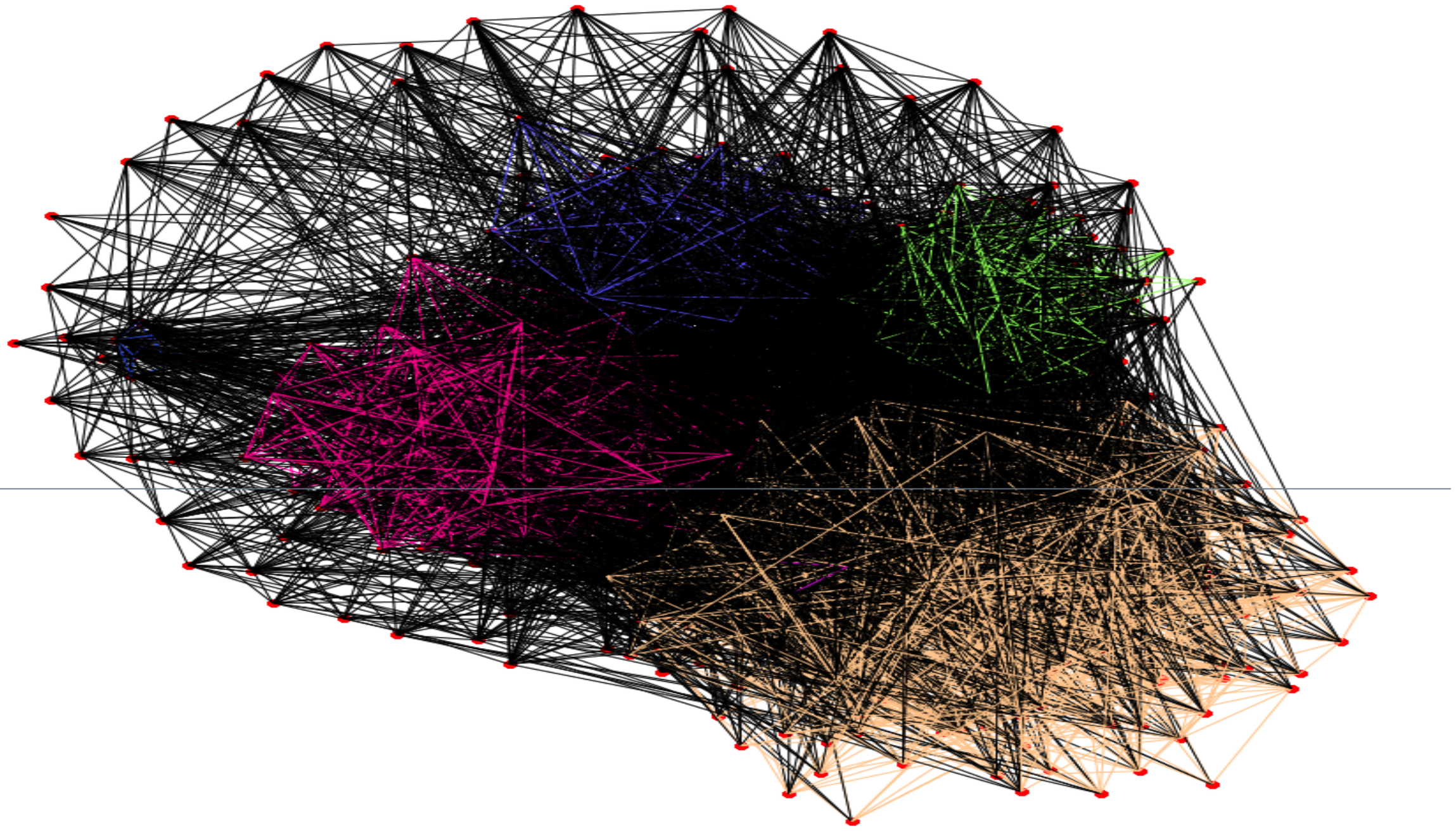
# Community Structure in Graphs

## Variable-incidence Graph of Non-random Formula



# Community Structure in Graphs

## Variable-incidence Graph of Random Formula



Vijay Ganesh

# Modularity and Communities in Graphs

## Community Structure in Graphs

- How to compute community structure?
- The decision version of the Q maximisation problem is NP-complete [Brandes et al., 2006]
- Many efficient *approximate* algorithms proposed, e.g. [CNM04] and [OL13]
- We use the above two algorithms for our experiments
- We get similar results with both algorithms, increasing our confidence in our results

# Community Structure and Random Instances

## Experiments #1: Hypothesis and Definitions

Hypothesis tested:

- Is there a range of Q values for randomly generated instances, that are hard for CDCL solvers; regardless of the number of clauses/ variables

- Are randomly generated instances outside this range uniformly easy

# Community Structure and Random Instances

## Experiments #1: Setup

- Randomly generated 550,000 SAT instances for the experiment
  - Varied $N_V$ between 500 and 2000 in increments of 100
  - Varied $N_{cl}$ between 2000 and 10000 in increments of 1000
  - Varied target Q between 0 and 1 in increments of 0.01
  - Varied "Number of communities" between 20 and 400 in increments of 20
- Experiments using MiniSAT
  - Timeout of 900 seconds per run
  - Run solver on inputs in a random order
  - Average the running time over several runs

# Community Structure and Random Instances

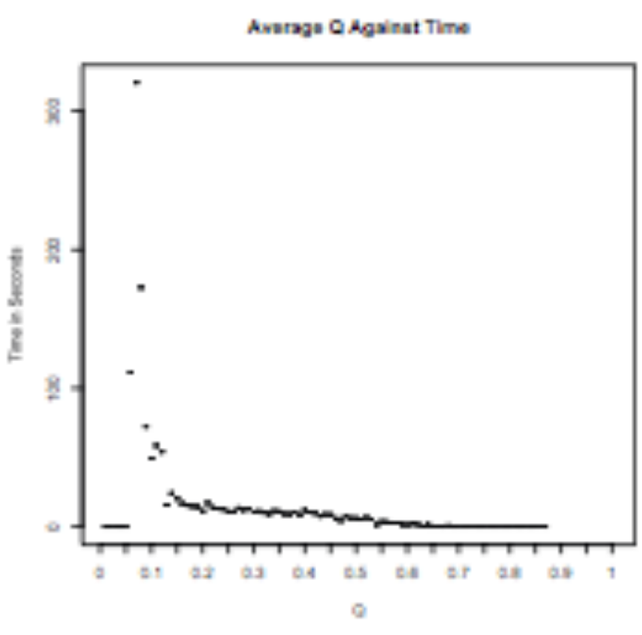
## Experiments Performed (#1)

- Plotted Q against time
- Noticed significant increase in execution time when $0.05 \leq Q \leq 0.13$
- Also recomputed the results using a stratified sample
  - Used due to high number of instances within target range
  - Randomly sample the data taking 250 results from each 0.1 range of Q between 0 and 0.9
  - Almost the same result: $0.05 \leq Q \leq 0.12$

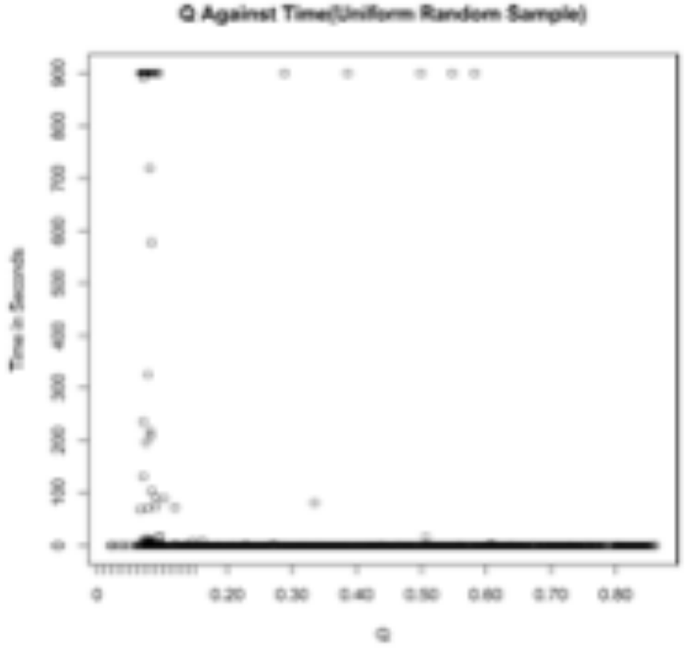
# Community Structure and Random Instances

## Experiments Performed (#1)

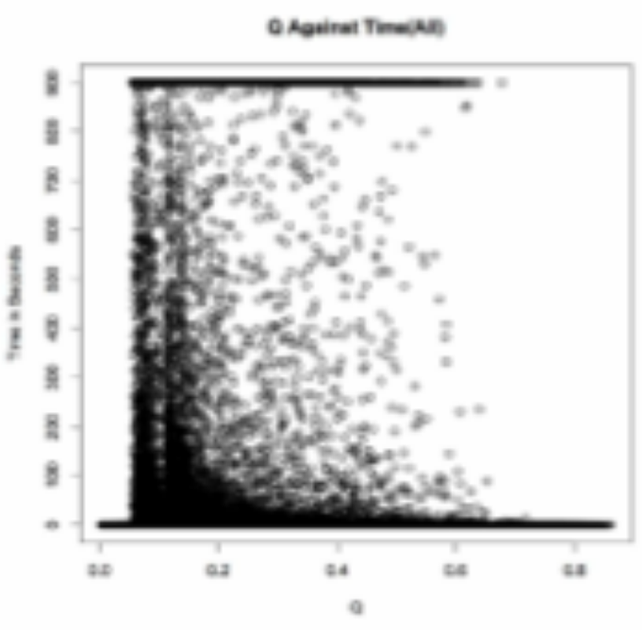
- Huge increase in running time of randomly generated instances when $0.05 \leq Q \leq 0.13$




(a) Average Time




(b) Stratified Sample




(c) All instances

# Community Structure and Random Instances

## Experiments #2: Hypothesis and Definitions

Hypothesis tested:

- Are the community modularity and number of communities better correlated with the running time of CDCL solvers than traditional metrics

- Is the correlation better for industrial instances than randomly generated or hand crafted ones

# Community Structure and Random Instances

## Experiments #2: Hypothesis and Definitions

- Instances used
  - Approximately 800 instances from the SAT 2013 competition. For the remaining we couldn't compute community structure due to resource constraints
  - Using OL algorithm to compute community structure for the 800 instances
    - Much faster and more scalable
- All experimental results are for Minipure
  - Obtained from the SAT 2013 competition website
- Used statistical tool R to perform standard linear regression

# Community Structure and Random Instances

## Experiments Performed (#2)

- Performed linear regression on the solver running time twice
  - Once with community structure metrics (and variables/clauses)
  - Once without
- Compared the adjusted $R^2$ (variability) from both experiments
  - Variability measures how good the models predicted results are, compared with the actual results.
  - Varies from 0 to 1
- The lower the variability (higher the $R^2$) the more predictive the model

# Community Structure and Random Instances

## Experiments Performed (#2)

- Timeouts included
  - A large portion (Approximately 60%) of the instances timedout
  - Not ideal, but without them there isn't enough data
- log(time) used
  - Timeouts
  - Wide distribution between instances that finished and timedout
- Data standardized to have mean = 0 and standard deviation = 1
  - Standard practice when regressors are in different scales.

# Community Structure and Random Instances

## Experiments Performed (#2)

- Model #1 - $R^2 \sim 0.5$
  - log(time) ~ |CL| * |V| * Q * |CO| * QCOR * CLVR
  - * denotes interaction terms between factors
  - |CL| = number of clauses
  - |V| = number of variables
  - |CO| = number of communities
  - QCOR = ratio of Q to communities
  - CLVR = ratio of clauses to variables
- Model #2 - $R^2 \sim 0.33$
  - log(time) ~ |CL| * |V| * CLVR

# Community Structure and Random Instances

## Experiments #2: Results and Interpretation

- The regressions show us that the model with the community structure metrics is a better predictor of running time than traditional metrics, i.e. number of clauses/variables.

| Factor | Estimate | Std. Error | t value | $Pr(>\|t\|)$ | Sig |
|---|---|---|---|---|---|
| $\|CO\|$ | -1.237e+00 | 3.202e-01 | -3.864 | 0.000121 | *** |
| $\|CL\| \odot Q \odot QCOR$ | -4.226e+02 | 1.207e+02 | -3.500 | 0.000492 | *** |
| $\|CL\| \odot Q$ | -2.137e+02 | 6.136e+01 | -3.483 | 0.000523 | *** |
| $\|CL\| \odot Q \odot \|CO\| \odot QCOR \odot VCLR$ | -1.177e+03 | 3.461e+02 | -3.402 | 0.000702 | *** |
| $\|CL\| \odot Q \odot \|CO\|$ | -6.024e+02 | 1.774e+02 | -3.396 | 0.000719 | *** |
| $Q \odot QCOR$ | 3.415e+02 | 1.023e+02 | 3.339 | 0.000881 | *** |
| $Q$ | 1.726e+02 | 5.200e+01 | 3.318 | 0.000947 | *** |
| $Q \odot \|CO\| \odot QCOR$ | 9.451e+02 | 2.927e+02 | 3.229 | 0.001292 | ** |

# Literal Block Distance (LBD) and Communities

## Experiment #3: Hypothesis and Definitions

### Hypothesis tested

- The number of communities in a conflict clause correlates strongly with its LBD measure

### What is LBD? (First introduced in Glucose solver [AS09])

- LBD measure M of a learnt clause C is a rank based on the number N of distinct decision levels the vars in C belong to
- The lower the value of N the better the rank M
- LBD is a powerful measure of the utility of a conflict clause

# Literal Block Distance (LBD) and Communities

## Experiment #3: Hypothesis and Definitions

- Clause deletion
  - Integral to the efficiency of modern solvers
  - Without clause deletion, conflict clause production quickly consumes available memory
- Which clauses to delete? LBD to the rescue
  - Periodically delete conflict clauses with bad LBD rank
  - As we will see, clauses with bad LBD rank are shared by many communities

# Literal Block Distance (LBD) and Communities

## Experiment #3: Intuition

- The number of communities in a conflict clause
  - The number of communities N in a conflict clause C is the number of distinct communities the variables in C belong to
- Intuition behind the hypothesis
  - High quality conflict clauses tend to span very few communities, i.e. N is small
  - High quality conflict clauses are likely to cause more propagation per decision variable, and hence are likely to have low LBD
  - LBD picks out high quality conflict clauses

# Literal Block Distance (LBD) and Communities

## Experiment #3: Setup

- Instances considered
  - 189 SAT 2013 industrial category instances out of 300
  - We were only able to compute communities for these 189
  - The rest caused memory-out errors

- Step 1
  - For each of the 189 instances, compute:
    - Community structure
    - The number of communities a learnt clause spans
    - LBD of every learnt clause (only for the first 20,000 due to resource constraints)

# Literal Block Distance (LBD) and Communities

## Experiments Performed (#3)

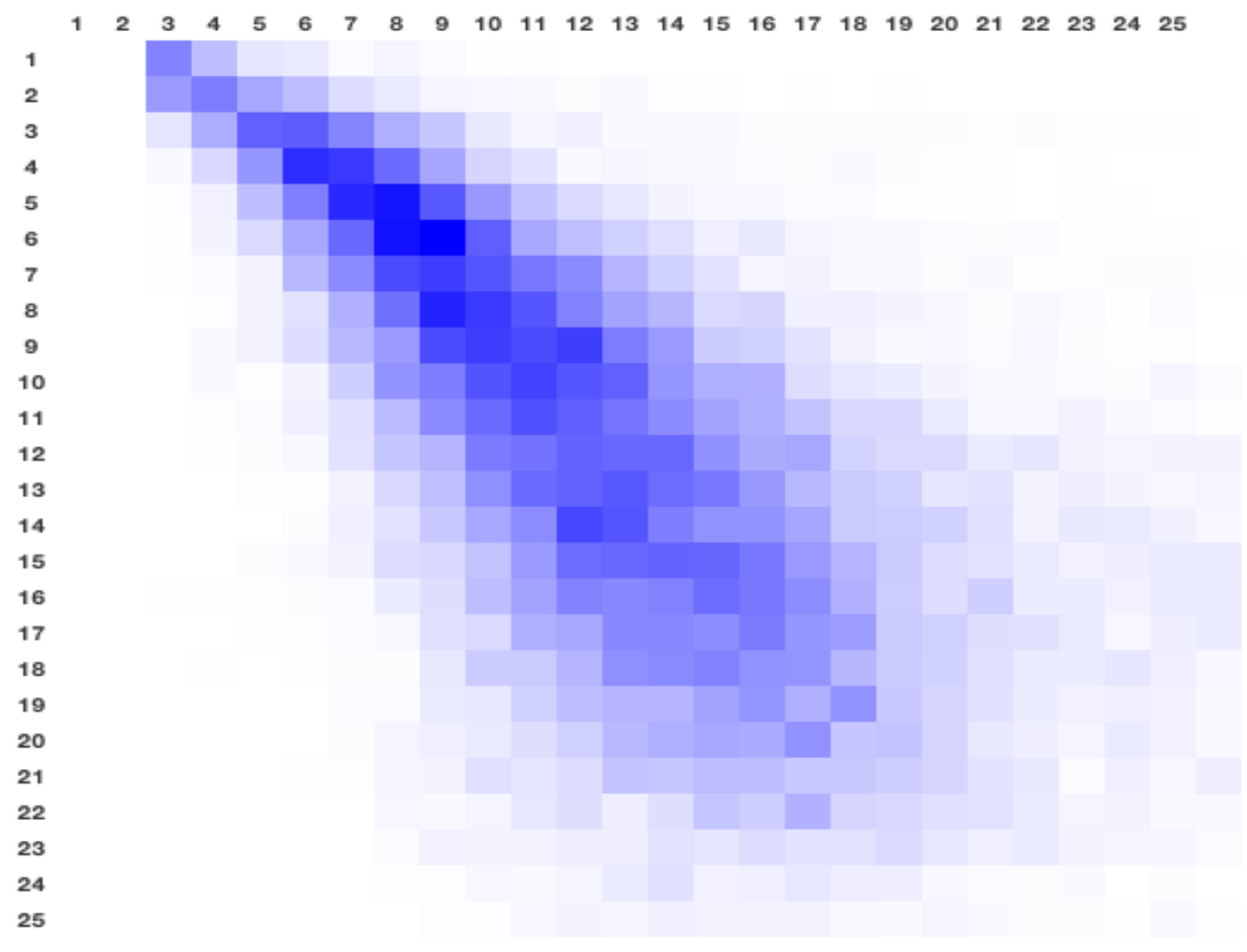
- Step 2
  - LBD of every learnt clause considered, was correlated with the number of communities it spans
  - Thousands of data points over the 189 instances

- Correlate LBD and number of communities using heatmaps
  - Heatmap of LBD and communities of learnt clauses
  - Otherwise difficult to correlate thousands of data points over hundreds of instances
  - One heatmap per SAT instance

# Literal Block Distance (LBD) and Communities

## Experiments #3: Results and Interpretation

- Result #1
  - Most industrial instances have a very strong correlation between LBD and communities



# Impact of Community Structure and Solver Running Time

## Scope for Improvement

- Use larger set of data
  - Enabling us to exclude timeouts
- Focus on individual categories
  - In some cases the $R^2$ for the individual categories is higher than the complete set
- Consider different regression techniques
  - The non-normality of the data stops us from estimating confidence intervals
- Try experiments on more solvers
  - Glucose result is similarly strong
- Compare different random generation techniques

# Impact of Community Structure and Solver Running Time

## Future Work

- Consider different graph representations for community detection
- Is it possible to determine a highly predictive model?
- Compare community structure model with graph-width based models
- Other solver measures, e.g. memory usage, rate of conflict generation
- Build visualisation tool to confirm/refute hypothesis
  - SATGraf
- Dynamically track how community structure of learnt clauses evolve

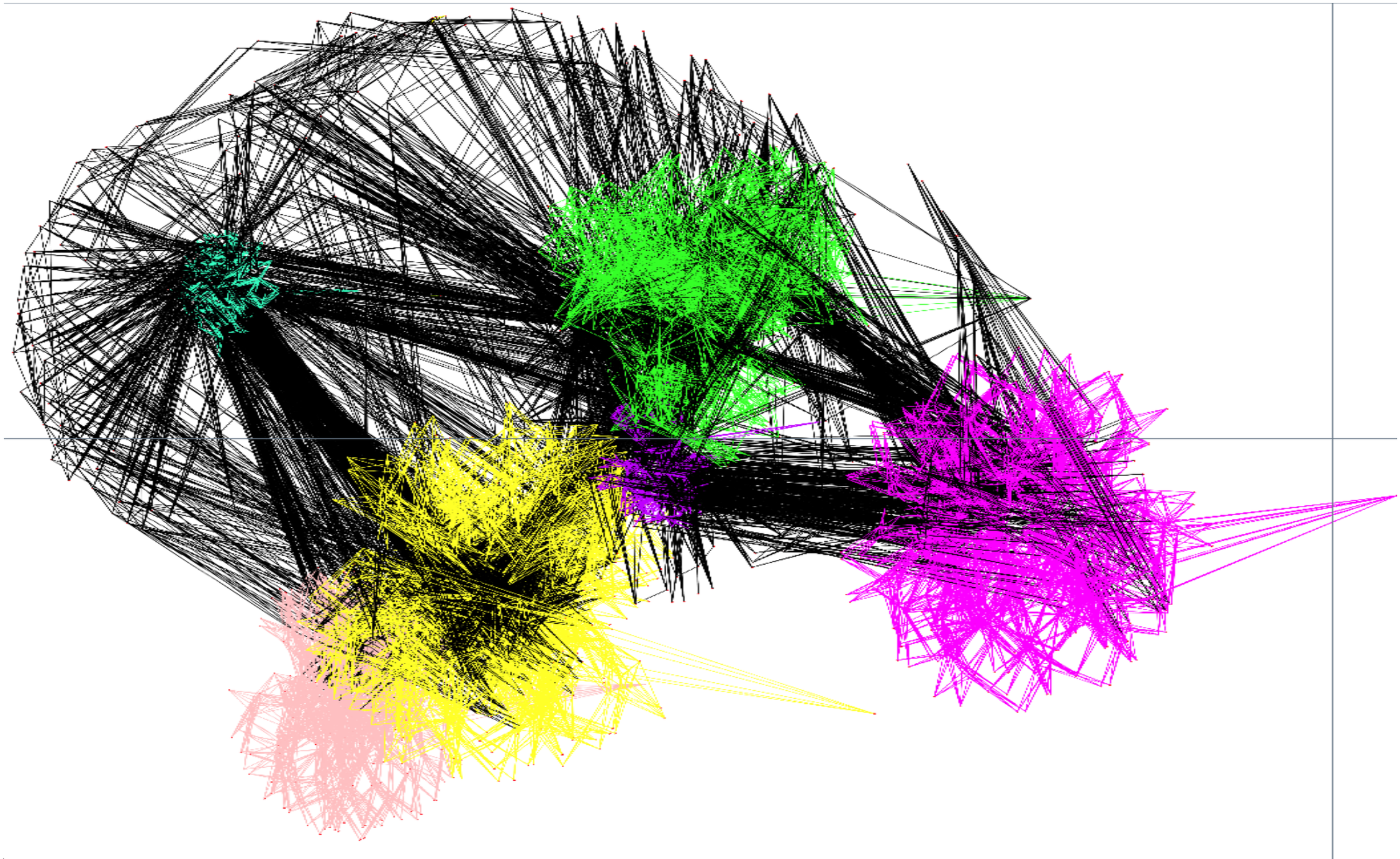
# Community Structure and SAT Solver Performance

## Conclusions

- Take home message
  - Community structure of SAT instances strongly affect solver performance
- Result #1
  - Hard random instances have low Q ($0.05 \leq Q \leq 0.13$)
- Result #2
  - Number of communities and Q of SAT instances are more predictive of Minipure performance than other measures
- Result #1
  - Strong correlation between community structure and LBD in Glucose

# Community Structure and SAT Solver Performance

## Questions



Vijay Ganes

# Key Contributions

## http://ece.uwaterloo.ca/~vganesh

| **Name** | **Key Concept** | **Impact** | **Pubs** |
|---|---|---|---|
| **STP**<br>Bit-vector & Array Solver[1,2] | Abstraction-refinement for Solving | Concolic Testing | CAV 2007<br>CCS 2006<br>TISSEC 2008 |
| **HAMPI**<br>String Solver[1] | App-driven Bounding for Solving | Analysis of Web Apps | ISSTA 2009[3]<br>TOSEM 2012<br>CAV 2011 |
| **Z3-str**<br>String and Numeric Solver[1] | Solver combination | Analysis of Web Apps | FSE 2013 |
| **Taint-based Fuzzing** | Information flow is cheaper than concolic | Scales better than concolic | ICSE 2009 |
| **Automatic Input Rectification** | Acceptability Envelope:<br>Fix the input, not the program | New way of approaching SE | ICSE 2012 |
| **Undecidability of forall-exists fragment over word equations** | Reduction from halting problem for two-counter machines | | HVC 2012 |

1. 100+ research projects use STP and HAMPI
2. STP won the SMTCOMP 2006/2010 and second in 2011/2014 competitions for bit-vector solvers
3. HAMPI: ACM Best Paper Award 2009
4. Retargetable Compiler (DATE 1999, 2008). Proof-producing decision procedures (TACAS 2003)