Overview

- **Today**: How state-of-the-art SAT solvers work
- Many competitive solvers based on DPLL, but extend it in three important ways:
  1. Non-chronological backtracking
  2. Learning from past “mistakes”
  3. Heuristics for choosing variables and assignments

Non-Chronological Backtracking

- **Recall basic DPLL**: First try assigning \( p \) to \( \top \); if doesn’t work, backtrack to most recent decision level and try \( p = \bot \)
- Called chronological backtracking but often sub-optimal
- Suppose made assignments \( p_1, p_2, \ldots, p_{100} \) but discovered \( p_4 \) was a bad choice
- Backtracking to decision level associated with \( p_{100} \) is stupid...
- In non-chronological backtracking, can go back to earlier decision levels

Learning

- **Learning** = acquisition of new clauses to prevent similar bad assignments
- For instance, suppose we discover \( p_5 = \top, p_{32} = \bot, p_{100} = \top \) is inconsistent, i.e.,
  \[ \phi \Rightarrow \neg(p_5 \land \neg p_{32} \land p_{100}) \]
- Can add this clause without changing satisfiability (why?)
- Such clauses called conflict clauses \( \Rightarrow \) SAT solver has database of conflict clauses

Decision Heuristics

- Basic DPLL chooses variables in random order
- But making assignment to certain variables can make formula much easier to solve!
- Modern solvers use more sophisticated heuristics
- This is something of a black art, but one of the most important elements in SAT solving . . .
The Plan

- We will talk about BCP and AnalyzeConflict first (related)
- Then: common decision heuristics used in the Decide step
- Finally: Implementation tricks to make all this fast

BCP in SAT Solvers

- Recall: BCP is all possible applications of unit resolution
- SAT solvers remember deductions performed in the BCP process ⇒ recorded as implication graph
- First some terminology . . .

Some Terminology and Conventions

- **Decision variable**: variable assigned in the Decide step
- The decision level of a decision variable is the level (order) in which it was assigned
- The decision level of a variable assigned due to BCP is the decision level of the last assigned decision variable
- **Important note**: Think of assignments as literals: Assignment \( p = \top \) is literal \( p \); assignment \( p = \bot \) as literal \( \neg p \)
- **Also**: An assignment corresponds to a new unit clause added to our set of clauses

Decision Level Example

\((\neg x_1 \lor x_2) \land (\neg x_3 \lor \neg x_4)\)

- Decide assigns \( x_1 = \top \) ⇒ \( x_1 \) decision var at level 1
- BCP yields:
- **Decision level of \( x_2 \)?**
- Decide next assigns \( x_4 = \top \). BCP deduces:
- \( x_3 \) decision variable with decision level:
- \( x_3 \)’s decision level:

Implication Graph

- An implication graph is a labeled directed acyclic graph
- **Nodes**: literals in the current partial assignment
- **Node labels**: Indicate assignment and decision level.
- Example: Node labeled \( \neg x : 3 \) means variable \( x \) was assigned to \( \bot \) at decision level 3
- Edges from \( l_1, \ldots, l_k \) to \( l \) labeled with \( c \): Assignments \( l_1, \ldots, l_k \) caused assignment \( l \) due to clause \( c \) during BCP
- A special node \( C \) is called the conflict node.
- Edge to conflict node labeled with \( c \): current partial assignment contradicts clause \( c \).
Another Example

- Consider the following clauses:
  \[ c_1 : (\neg a \lor c) \quad c_2 : (\neg c \lor \neg a \lor b) \quad c_3 : (\neg c \lor d) \quad c_4 : (\neg d \lor \neg b) \]
- Suppose \textit{Decide} assigned \( a = \top \) at decision level 1
- Using clause \( c_3 \), BCP yields:
- Using clause \( c_2 \), BCP yields:
- Assignment \( b = \top, d = \top \) contradicts:

Example 3

- Based on this implication graph, what is \( c_4 \)?
- What is \( c_3 \)?
- What is \( c_1 \)?
- What is \( c_2 \)?

Analyzing Conflicts

- Point of implication graph: analyze conflict
- AnalyzeConflict has two goals:
  1. Learn new conflict clauses
  2. Figure out what level to backtrack to

Conflict Clauses

- A \textit{conflict clause} is a clause implied by the original formula
- Point of conflict clause: Prevent bad partial assignments by deriving contradiction as quickly as possible
- \textbf{Question}: To achieve this goal, are small or large conflict clauses better?
- \textbf{Answer}: Small ones because the smaller the clause, the quicker BCP forces variable assignments, and the quicker we derive contradictions!
- The implication graph is very useful for deriving small clauses implied by the original formula!

Implication Graph Properties

- Root nodes in the implication graph correspond to what kind of variables?
- Edges and internal nodes arise due to BCP
- If literal \( l \) has incoming edge labeled \( c \), what do we know about \( c \)?
- If literal \( l \) has outgoing edge labeled \( c \), what do we know about \( c \)?
**Using Implication Graph to Analyze Conflicts**

- What can we say about source of conflict based on this (partial) implication graph?
- Are other decision variables relevant to conflict?

**One Strategy to Derive Conflict Clause**

- One way to derive conflict clause: Conjoin all literals associated with root nodes reaching conflict node, use negation as conflict clause
- Another way to derive conflict clause: The negation of current partial assignment
- Question: Which one is better?

**Analyzing Conflicts**

- This strategy is one of the earliest strategies proposed for inferring conflict clauses (e.g., the GRASP SAT solver)
- But people have improved upon this; possible to derive even better conflict clauses!
- A key concept is unique implication points

**Unique Implication Point**

- A node $N$ in the implication graph is a **unique implication point (UIP)** if all paths from current decision node to the conflict node must go through $N$
- Is the current decision node a UIP?
- Can there be multiple unique implication points?
- First unique implication point: UIP closest to conflict node

**UIP Example**

- Which nodes are UIP's?
- Which node is first UIP?
Using UIP and Resolution for Deriving Conflict Clause

- Inferring better conflict clauses: Start with clause labeling incoming edge to conflict node, derive new clauses via resolution until we find literal in first UIP.
- Specifically: In current clause $c$, find last assigned literal $l$ in $c$.
  - Pick any incoming edge to $l$ labeled with clause $c'$.
  - Resolve $c$ and $c'$.
  - Set current clause be resolvent of $c$ and $c'$.
  - Repeat until current clause contains negation of the first UIP literal (as the single literal at current decision level).

Analyzing Conflict via Resolution Example

- What is $c_1$?
  - Last assigned literal in $c_1$:
  - Clause $c_3$ labeling incoming edge:
  - Resolve $c_1$ and $c_3$:
  - $\neg x_4$ only literal from decision level 8 $\Rightarrow x_2 \lor \neg x_4$ conflict clause

Another Example

- What is the first UIP?
  - Start with clause $c_4$:
  - Suppose we pick $\neg x_7$:
  - Clause on incoming edge to $\neg x_7$:
    - Resolve $c_3$, $c_4$:
    - Suppose $x_6$ assigned later, pick $x_6$:
    - Clause on incoming edge:
    - Resolve current clause with $c_2$:

Another Example, cont.

- Current clause:
  - Are we done?
  - Pick last assigned literal: $x_5$:
  - Incoming edge to $x_5$:
    - Resolve with current clause:
    - Are we done?
    - New conflict clause: $x_2 \lor \neg x_4 \lor x_5$

Why is this correct?

- Why are the clauses obtained this way implied by formula?
- Unclear if there is a deep reason why this works well, but seems effective in practice ...
Backtracking and Asserting Clauses

- **A good strategy:** We want to backtrack to a level that makes conflict clause \( c \) an asserting clause in the next step
- Asserting clause is a clause with exactly one unassigned literal
- Hence, if we make \( c \) an asserting clause, BCP will force at least one assignment

Choosing Backtracking Level

- **Question:** If we want to make conflict clause \( c \) an asserting clause in the next step, what level should we backtrack to?
- **Answer:**
  - Since conflict clause contains only one literal, say \( l' \), from the first highest decision level, backtracking to \( d \) will assert \( l' \)

Going Back to Example

- Recall: We obtained the conflict clause \( x_2 \lor \neg x_4 \)
- What level do we backtrack to?
- What do we delete in the graph?
- After we add \( x_2 \lor \neg x_4 \) to clause database, BCP implies:
  - Different assignment than before!

Decision Heuristics

- Important part of SAT solvers, but something of a black art
- Can come up with hundreds of heuristics with varying tradeoffs
- We’ll only talk about two:
  1. dynamic largest individual sum (DLIS)
  2. variable state independent decaying sum (VSIDS)

Dynamic Largest Individual Sum (DLIS)

- This heuristic chooses the literal that satisfies the largest number of currently unsatisfied clauses.
- **Example:** \( (x_1 \lor \neg x_2) \land (\neg x_2 \lor x_3) \land (\neg x_3 \lor x_2 \lor \neg x_3) \)
- What assignment would DLIS pick for this formula? (assuming no assignments so far)
- How is this heuristic is dynamic?
- Thus, overhead can be high and must be implemented carefully to minimize bookkeeping
Variable State Independent Decaying Sum (VSIDS)

- Similar to DLIS, but tries to reduce overhead and favor literals involved in conflicts (i.e. conflict-driven)
- Count number of clauses in which the literal appears, but disregard if the clause it appears in is satisfied or not
- Specifically, initialize the score of each literal to the number of clauses in which literal appears
- Every time we add a conflict clause involving literal $l$, increase the score of that literal by 1
- Periodically divide scores of all literals by 2 ⇒ decaying sum

Favors literals involved in conflicts
- If a literal doesn’t appear in recent conflict, its score will decay over time
- On the other hand, if literal appears in recent conflict, its score will be increased, so its score won’t decay as much
- Much cheaper compared to DLIS because we don’t need to scan all clauses to figure out which ones are satisfied
- Introduced in the CHAFF SAT solver from Princeton, written by undergrads!

Implementation Tricks
- To build competitive SAT solvers, it is important to minimize overhead of implementing Decide, BCP, and Analyze Conflict
- Very important because SAT solver might be searching through hundreds of thousands of assignments!
- We’ll talk about two issues:
  1. number of conflict clauses
  2. trick to perform BCP fast: watch literals

Conflict Clauses
- Recall: After analyzing conflict, we add new conflict clause to our clause database
- Pro: Conflict clauses quickly block bad assignments and prevent future mistakes
- Con: More clauses = more overhead
- Tradeoff between conflict prevention and minimizing overhead

Conflict Clauses, cont.
- For this reason, many SAT solvers do not keep all the conflict clauses they derive
- For example, they put a limit on the number of conflict clauses they derive
- Typically, keep most recent conflict clauses since they are most relevant to current part of search space

Implementing BCP
- Implementing BCP efficiently is very important because SAT solvers spend a lot of time doing BCP
- Naive implementation of BCP: Requires scanning all currently unsatisfied clauses
- But industrial SAT contain hundreds of thousands of clauses, so scanning all unsatisfied clauses too expensive!
- A more intelligent implementation: Keep mapping from each literal to all clauses in which each literal appears (because we perform unit resolution after each variable assignment)
- But this is still very expensive because typically each literals appears in many clauses
The Trick: Watch Literals

- Modern SAT solvers use a much more clever trick to perform BCP fast: watch literals
- Observe: Ultimate purpose of BCP is to figure out which variable assignments imply which others
- Question: If we are performing unit resolution between \( l \) and clause \( c = (\neg l \vee l_1 \ldots \vee l_k) \), under what condition will a new assignment be implied?
- Answer:
  - Idea: Since a clause will not imply new variable assignment unless it has only two literals left, we only need to look at clauses that have at most two unassigned literals!

Watch Literals

- To efficiently detect clauses with at most two unassigned literals, select two unassigned literals in each unsatisfied clause as watch literals
- Invariant: Watch literals are always unassigned!
- To maintain invariant: If a watch literal is assigned a truth value and clause has other unassigned literals, choose any unassigned literal in clause to be new watch literal
- If a watch literal is assigned a truth value and there are no other unassigned non-watch literals left, BCP implies an assignment to the only remaining watch literal!

Watch Literals, cont.

- Question: Given this invariant, if we make assignment \( l \), which clauses can imply new variable assignments?
- Answer:
  - If \( \neg l \) does not appear, we can’t perform unit resolution
  - If \( \neg l \) appears but is not a watch literal, then clause has more than two unassigned literals \( \Rightarrow \) won’t imply new assignment!

- Watch literal trick makes BCP much faster because much fewer clauses contain negation of current literal as a watch literal!
- Yielded huge improvement in SAT solver performance!

Practical SAT Solving Summary

- Most competitive solvers today are based on DPLL
- But they extend DPLL in three ways: non-chronological backtracking, conflict clause learning, decision heuristics, engineering tricks (watch literals)
- Referred to as CDCL: conflict-driven clause learning
- Many competitive SAT solvers based on CDCL
- There are also other kinds of SAT solvers not based on CDCL, for instance, perform stochastic search (e.g., WalkSAT)
- Stochastic SAT solvers perform well on randomly-generated SAT instances, but not so well on industrial ones