Announcements

- Posted practice problems on Piazza
- Relevant (optional) reading from textbooks/papers posted on course webpage

Overview

- **Last lecture:** Basic DPLL algorithm for deciding satisfiability in boolean logic
- **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
- In addition, some implementation tricks to perform BCP fast

Non-Chronological Backtracking

- **Recall basic DPLL:** First try assigning $p$ to $\top$; if doesn’t work, backtrack to most recent decision level and try $p = \bot$
- This is called chronological backtracking because we backtrack to the most recent branching point
- But in some cases this is sub-optimal!
- Suppose we assigned to variables $p_1, p_2, \ldots, p_{100}$ and discovered that assignment to $p_4$ was a bad choice
- Backtracking to decision level associated with $p_{100}$ is stupid because we were already doomed after assigning to $p_4$
- In non-chronological backtracking, we don’t have to go back to most recent decision level

Learning

- **Learning** = acquisition of new clauses that prevent bad assignments similar to those already explored
- For instance, suppose SAT solver makes an assignment and discovers that $p_5 = T, p_{32} = \bot, p_{100} = T$ is inconsistent, i.e.,
  $$\phi \Rightarrow \neg(p_5 \land \neg p_{32} \land p_{100})$$
- Thus, we can add this clause without changing $\phi$’s satisfiability (why?)
- Such clauses “learned” by SAT solvers called **conflict clauses**
- SAT solvers maintain a database of conflict clauses to prevent bad future assignments

Decision Heuristics

- In the basic DPLL algorithm, we chose variables in a random order, and always tried $T$ first before $\bot$
- But we can do better!
- Making assignment to certain variables can make formula much easier to solve!
- Practical DPLL-based solvers use more sophisticated heuristics to choose variable order and truth assignments
- 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

Some Terminology and Conventions

- **Decision variable**: variable assigned in the Decide step
- Variables assigned due to BCP are not decision variables
- 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

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 $\neg z : 3$ means variable $z$ 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 $c$. 
Implication Graph Example

- Consider the following set of clauses:
  \[ c_1 : (\neg a \lor c) \quad c_2 : (\neg a \lor \neg b) \quad c_3 : (\neg c \lor b) \]
- Assume Decide assigned \( a = \top \) at decision level 2
- BCP yields:
  - Assignment contradicts \( c_3 \)!
  - \( a:2 \quad c:2 \quad \neg b:2 \)

Root node

\[ \begin{array}{c}
  a:2 \\
  c1 \\
  c2 \\
  b:2 \\
  c3 \\
  c \\
\end{array} \]

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 Decide assigned \( a = \top \) at decision level 1
- Using clause \( c_1 \), BCP yields:
- Using clause \( c_2 \), BCP yields:
- Using clause \( c_3 \), BCP yields:
  - Assignment \( b = \top \), \( d = \top \) contradicts:

Example cont.

- 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 Decide assigned \( a = \top \) at decision level 1
- Resulting implication graph:

Example 3

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

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 \( c \)?

Analyzing Conflicts

- So far: Implication graph used to record history of choices and subsequent BCP
- But whole point of recording this history is to analyze conflict
- AnalyzeConflict has two goals:
  1. Learn new conflict clauses
  2. Figure out what level to backtrack to
- Next: How to use the implication graph to derive conflict clauses and choose backtracking level
**Conflict Clauses**

- A **conflict clause** is a clause (disjunct) implied by the original formula.

- **Point of conflict clause**: Prevent bad partial assignments by deriving contradiction as quickly as possible.

- **Question**: To achieve this goal, are small or large conflict clauses better?

- **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!

**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.

- **Question**: Ok, \( \neg A \) is valid conflict clause, but why is it better than taking the negation of the whole partial assignment?

- **Answer**: 

**Analyzing Conflicts**

- This strategy is one of the earliest strategies proposed for inferring conflict clauses.

- Original GRASP SAT solver derived conflict clauses this way.

- But people have improved upon this; possible to derive even better conflict clauses!

- A key concept is unique implication points.

**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?

**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 \( N \).

- Same concept as dominator.

- 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: $x_2 \lor \neg x_4 \lor x_{10}$

Why is this correct?

- Initially, we resolve two clauses from the original formula
- Hence, initial "current clause" is implied by the formula
- At every step, we resolve "current clause" with another clause in the formula
- Hence, new "current clause" is also implied by the formula
- Thus, we can add conflict clause without changing satisfiability
- Unclear if there is a deep reason why this works well, but seems effective in practice . . .
Backtracking

- Recall: AnalyzeConflict has two goals.
- First goal: Deriving conflict clauses ✓
- Second goal: Figure out what level to backtrack to
- Backtrack to level \(d\) means delete all variable assignments made after level \(d\) (but assignments at level \(d\) not deleted)
- Next: Talk about how to infer a good level to backtrack to

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 \(l'\)!

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)

Recall: SAT Solver Architecture

- Decision heuristics for choosing variable order and truth assignment
Dynamic Largest Individual Sum (DLIS)

- This heuristic chooses the literal that satisfies the largest number of currently unsatisfied clauses.
- A clause is unsatisfied if the clause does not evaluate to true under the current partial assignment.
- Example: \((x_1 \lor \neg x_2) \land (\neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor \neg x_3)\)
- What assignment would DLIS pick for this formula? (assuming no assignments so far)
- How is this heuristic 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.
- Much cheaper compared to DLIS because we don’t need to scan all clauses to figure out which ones are satisfied.

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 \(\Rightarrow\) 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.
- Can guarantee termination of algorithm even if we do not keep all conflict clauses.
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 \( c = (\neg l \lor l_1 \lor \ldots \lor l_k) \), 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.