Satisfiability of Propositional Formulas
Scribe for lecture given on March 23, 2006 by Mooly Sagiv

Problem Definition
Given a propositional formula (Boolean function), e.g., \( \varphi = (a \lor b) \land (\neg a \lor \neg b \lor c) \),
determine whether \( \varphi \) is satisfiable: find a satisfying assignment or report that no
satisfying assignment exists.

The obvious solution – enumerate all assignments and check each one – is too
expensive as there are \( 2^n \) possible truth assignments for \( n \) variables.

A propositional formula can be also described by a decision tree:

\[
\begin{array}{c}
\text{a} \\
\downarrow \\
\text{b} \\
\downarrow \\
\text{c} \\
\downarrow \\
\text{c} \\
\downarrow \\
\text{c} \\
\downarrow \\
\text{c} \\
\end{array}
\]

Figure 1 A decision tree for the formula given above

The tree is determined by the ordering on the variables (a,b,c in this case).

Problem Importance
SAT is a core computational engine for major applications in various CS and EE
(Electronic Engineering) fields:

- **Artificial Intelligence.** SAT is used for knowledge base deduction, planning,
  and reasoning.
- **Program Analysis.** SAT is used by automatic theorem provers.
- **Electronic Design Automation:**
  - Testing and verification
  - Logic synthesis
  - FPGA routing
  - Path delay analysis

Many problems that are not originally specified as propositional formulae are
translated to propositional formulae form and then passed to SAT solvers. SAT also
captures some of the essence of problems that manifest in first-order logic (more on
that in the next lecture).
**Problem Representation**

In order to conceptually simplify the problem, formulas are usually represented in **CNF** (Conjunctive Normal Form) form.

Propositional formulae can be transformed into CNF by:

1. Applying equivalence-preserving Boolean algebra rules:
   \[
   a \lor (b \land \neg (c \lor \neg d)) \equiv (a \lor (b \land \neg c \land \neg d)) \equiv (a \lor (b \land \neg c) \land (a \lor d)) \equiv (a \lor b) \land (a \lor \neg c) \land (a \lor d)
   \]
   The size of the output formula can be exponential in the size of the input formula in the worst case.

2. Adding auxiliary variables in order to produce an equisatisfiable formula. The size of the output formula is polynomial in the size of the input formula, but number of variables grows, which might have a negative impact on the performance of SAT solving algorithms.
   (More details on conversion algorithms is available in "The Optimality of a Fast CNF Conversion and its use with SAT" / Daniel Sheridan. Appeared in SAT 2004).

The advantage of the CNF representation is that the data structure used to encode the formula in the computer is rather simple. The problem becomes conceptually simple as well – the algorithm needs to satisfy all clauses.

Notations and terms: A **literal** is a (propositional) variable or its negation. A **clause** is a disjunction of literals.

The following notations are equivalent:

\[
\phi = (a \lor b) \land (\neg a \lor \neg b \lor c) \equiv (a + b)(a' + b' + c)
\]

The one on the right is usually encountered in EE.

**Complexity Results**

SAT is the first established NP-Complete problem. The problem is NP-Complete even when there are at most 3 literals per clause (3-SAT):


This means that there is no polynomial algorithm for all instances unless P = NP.

There are a few instances where the problem becomes polynomial:

- When the formula contains at most two literals per clause (2-SAT) there are several linear time algorithms, e.g., a recent one given by **Alvaro del Val**.

- The formula contains at most one positive literal in every clause (Horn form). This problem is sometimes called **HORNSAT**.

The problem is that these syntactic restrictions do not allow many **useful** formulae to be specified. One interesting question is whether better complexities can be achieved if we extract additional parameters from formulae (similar to what is done for problems on graphs).
Goals
We wish to develop algorithms to solve SAT instances. Of course, we cannot hope to have sub-exponential time algorithms (unless we want to show that P=NP) so we aim to have algorithms that work well for many cases by using different heuristics.

Until today different SAT solvers have been developed. State-of-the-art solvers can handle very large instances (tens of millions of variables). The CS/EE communities hold SAT conferences annually, e.g., International Conference on Theory and Applications of Satisfiability Testing.

SAT competitions are held, where different tools are run against various SAT instances and compared in different categories: randomly-generated formulae, hand-made (crafted) formulae, industrial formulae, and formulae from AI.

The Resolution Principle
The resolution operation takes two clauses that disagree only on a single variable, that is, the variable appears in different polarities and all other variables have the same polarity in both clauses. The output is a single clause. For example, the resolution of \( (a + b + c' + f) \) and \( (g + h' + c + f) \) is the clause \( (a + b + g + h + f) \). The final clause is obtained by taking all literals from the clauses, except for the one that contains the incompatible variable (c in this case). The resulting clause is equisatisfiable to the conjunction of the two clauses, that is, the output formula is satisfiable if and only if the input formula is satisfiable. However, the output formula is "simpler" then the original conjunction, in the sense that every assignment that satisfies the conjunction satisfies their resolution.

If two clauses have more than one incompatible variable then their resolution is a tautology (a valid formula), which does not contribute to simplifying the formula.

The David Putnam (DP) Algorithm
This algorithm, which appeared in a paper in 1960, is based on the resolution principle. The algorithm iteratively selects a variable for resolution and applies resolution to all of the clauses including the variable, and then removes the original clauses. Note that the original clauses can be removed only after resolution has been applied to all of the clauses that contain the incompatible variable (more details in the paper). There are two special cases: 1. When there are no variables available for resolution, the formula is satisfiable; and 2. When the empty clause appears, the formula is unsatisfiable.

It is obvious that the number of iterations is bounded by the number of variables. However, every iteration can increase the number of clauses quadratically. Therefore, the space complexity of the Davis Putnam algorithm is exponential (even worse than the naïve assignment enumeration algorithm).
The DLL/DPLL Algorithm

This algorithm, which appeared in a paper in 1962, requires only a polynomial amount of space. It is used by many modern SAT solvers as the underlying scheme.

The basic idea is to enumerate the satisfying assignments in a sophisticated way, trying to prune large groups of assignments as early as possible. The algorithm uses the concept of the decision tree and enumerates the assignments in a DFS manner. The search depends on a variable ordering that is chosen by some heuristic. In every step, the status of the search can be described by a node in the tree, such that the path to the node from the root describes variables for which the assignment has already fixed some values.

In order to accelerate the search on the tree, the algorithm maintains an auxiliary data structure called the implication graph, which is updated dynamically on every decision. The vertices of the graph are pairs made of a variable and a truth value, and the edges are labeled by clauses and represent implications. More specifically, when there exists a clause containing \( n \) literals such that \( n-1 \) literals have been fixed with a 0 value then, in order for the clause to be satisfied, the remaining variable has to be assigned such that the literal has the value 1. The algorithm adds edges from the nodes included in the clause to a new node where the last variable is set with the appropriate value. When a variable occurs in two nodes with different values then a conflict has been found. This means that the algorithm does not have search further down the sub-tree. Instead, it discards the implications graph and backtracks.

The advantage of the DLL algorithm over the DP algorithm is that it requires only a polynomial amount of space. However, maintaining the implication graph can be very costly. With this basic scheme, the algorithm is still applicable for formulae with relatively small number of clauses (in the order of 1,300 clauses).
**Implications and Boolean Constraint Propagation (BCP)**

An important improvement to the basic DPLL algorithm involves propagation of constraints.

We say that a clause is a unit clause if exactly one literal in it is unassigned and all other literals have a 0 value. The unit clause rule applies when a unit clauses exists by determining the value of the variable of the unassigned literal. After applying the rule, possibly new unit clauses may appear and the rule can be applied again.

Improvements suggested in later works include conflict-driven learning and non-chronological backtracking:


These two techniques enable practical SAT instances to be solved in reasonable time.

**Conflict-Driven Learning and Non-chronological Backtracking**

Recall that when the basic DPLL encountered a conflict it discarded the implication graph and used backtracking – in essence it did not learn anything that can be used to avoid the same conflict again. A better approach is to analyze the conflict, learn the "reason" to why it appeared and add a clause to the formula that will make it possible to avoid arriving to the same conflict again.

A conflict clause is generated by a bipartition of the implication graph. The partition has all the decision variables on one side (called *reason side*), and the conflicting variable in the other side (the *conflict side*). All the vertices on the reason side that have at least one edge to the conflict side comprise the reason for the conflict.
With the conflict-learning technique, the algorithm can backtrack more than one-level by going to the highest (i.e., closest to the root) variable in the conflict clause.

Conflict learning significantly prunes the search space, since a learned clause is useful for the remainder of the algorithm. It is also useful in generating future conflict clauses.

**Restart**

Another useful technique is to abandon the current search tree and reconstruct a new one when the search is "stuck". The clauses learned prior to the restart are still there after the restart and can help pruning the search space. This technique adds to robustness in the solver.
In order to ensure termination of the algorithm with the restart heuristics, existing algorithms use a period counter. The period counter is a measure on the "time" spent by the algorithm with a given order, e.g., number of conflicts discovered. Upon every restart the period is doubled, so restarts occur less and less often. Since the algorithm without restarts terminates, the decaying restart rate ensures termination.

**Chaff: A Modern SAT Solver**


Uses all the essential ingredients of DPLL (i.e. conflict resolution and learning) to prune search space. But also emphasizes coding efficiency and optimizing data cache behavior.

"in most cases) of the solvers’ run time is spent in the BCP process. Therefore, an efficient BCP engine is key to any SAT solver."

To optimize BCP, the algorithm chooses two "guard" variables from each clause to watch. Only when these two variables become 0 the algorithm checks the clause for possible implications. The algorithm always maintains the invariant that any changes to the assignment that can impact a clause have to go through the guard variables. This saves the algorithm from unnecessarily checking clauses in many cases. A nice property of this technique is that no operation has to be taken upon backtracking since the invariant is maintained. As a result of this technique the algorithm achieves good memory locality and better cache behavior.

**Other Approaches to SAT**

The algorithms described so far are complete – they always terminate and give a correct answer. Some algorithms are based on techniques such as local search, they give up on completeness and attempt to address only satisfiable instances.