Lecture 9: The Splitting Method for SAT

1 Importance of SAT

Cook-Levin Theorem: SAT is NP-complete.

The reason why SAT is an important problem can be summarized as below:

1. A natural NP-Complete problem.
2. We can prove NP hardness of other problems by reducing SAT to them.
3. We can solve other problems by reducing them to SAT and then solving SAT.

The focus of this lecture is on SAT solving.

Complexity of NP-Complete problems is usually defined in terms of worst-case. SAT best known worst-case complexity is around 1.3^n. Does this mean all other NP-complete problems have worst-casecomplexity of 1.3^n? No, because it also depends on the complexity of the reduction. Now, the reduction to is polynomial, but how large is the the polynomial? For example, 1.3^n is very different from 1.3^{n^2}.

Our approach here would be to develop heuristics that would work well in practice.

2 Developing an algorithm for solving SAT

Since SAT is a very important problem, we would like to develop efficient algorithms for its solution. SAT being NP-Complete implies that SAT solvers are general problem solving engines. The algorithm to solve it is going to be exponential (to the best of our knowledge so far). A naive SAT algorithm would look like one shown below:

$SAT(\varphi)$
  $B := FALSE$
  for all $\tau \in 2^{AP(\varphi)}$
    $B := B \vee \tau(\varphi)$
  return B

Yet, given two exponential algorithms, one can be exponentially better than the other! This means that we should look for possible optimizations. Some of the obvious improvements to the above problem are:

1. Return early as soon as B is true.
2. Look for sub-formulas to prune the search space.

In particular, below we give a few examples where the problem could be solved really easily because the formula has some property that we can detect.

- Consider $\varphi = (p \vee q \vee \neg r) \wedge (q \vee \neg r \vee s) \wedge \cdots \wedge s \wedge (\neg s)$. The formula can be arbitrarily long, but still, because it contains a contradiction $s \wedge (\neg s)$ it is unsatisfiable no matter what the truth assignment is to the other variables!
- Consider the formula $\varphi = (\psi \vee p)$, where ψ is a complicated formula. We can satisfy φ easily by taking p to be true.
- Another example is a formula like $\varphi = p \wedge \psi$, where ψ is also a formula. Although we cannot immediatly say anything about the final result, we can reduce the problem in size. Specifically, p **must** be true, otherwise the formula is obviously not satisfiable. So we can assign to proposition p its only possible truth value, and propagate this assignment to ψ. Thus, p becomes a constant and the original formula that we had to check is transformed to an equivalent formula which is simpler since it has one less variable.

The approach we will take is to search for a satisfying truth assignment incrementally, using the principle of "*Divide and Conquer*"[1]. Here we present a simplified version, and the course project will be to build a very fast version.

To deal with partial truth assignments we need to propagate truth values into formulas. We start by defining *extended formulas*:

$$FORM' = FORM \cup \{0, 1\}.$$

This practically means that we now consider the constants 0 and 1 as extended formulas. We now show that extended formulas satisfy the closure condition, that is, they are closed under the propositional connectives.

- $\neg : FORM' \to FORM'$.
- $\circ : FORM' \times FORM' \to FORM'$

are defined as follows:

$$\neg(\varphi) = \begin{cases} 1, & \varphi = 0 \\ 0, & \varphi = 1 \\ (\neg\varphi), & \varphi \in FORM \end{cases},$$

[1] *Divide et impera*

$$\vee(\varphi,\psi)=\begin{cases}1 & \varphi=1\\ 1 & \psi=1\\ 0 & \varphi=0 \text{ and } \psi=0\\ (\varphi\vee\psi) & \varphi,\psi\in FORM\\ \psi & \varphi=0 \text{ and } \psi\in FORM\\ \varphi & \psi=0 \text{ and } \varphi\in FORM\end{cases},$$

$$\wedge(\varphi,\psi)=\begin{cases}0 & \varphi=0\\ 0 & \psi=0\\ 1 & \varphi=1 \text{ and } \psi=1\\ (\varphi\wedge\psi) & \varphi,\psi\in FORM\\ \psi & \varphi=1 \text{ and } \psi\in FORM\\ \varphi & \psi=1 \text{ and } \varphi\in FORM\end{cases},$$

$$\rightarrow(\varphi,\psi)=\begin{cases}0 & \varphi=1 \text{ and } \psi=0\\ 1 & \varphi=0 \text{ or } \psi=1\\ (\varphi\rightarrow\psi) & \varphi,\psi\in FORM\\ \psi & \varphi=1 \text{ and } \psi\in FORM\\ \neg\varphi & \psi=0 \text{ and } \varphi\in FORM\end{cases},$$

$$\leftrightarrow(\varphi,\psi)=\begin{cases}0 & \varphi=1 \text{ and } \psi=0\\ 0 & \varphi=0 \text{ and } \psi=1\\ 1 & \varphi=1 \text{ and } \psi=1\\ 1 & \varphi=0 \text{ and } \psi=0\\ (\varphi\leftrightarrow\psi) & \varphi,\psi\in FORM\\ \psi & \varphi=1 \text{ and } \psi\in FORM\\ \varphi & \psi=1 \text{ and } \varphi\in FORM\\ \neg\varphi & \psi=0 \text{ and } \varphi\in FORM\\ \neg\psi & \varphi=0 \text{ and } \psi\in FORM\end{cases}.$$

Now we can define *simplification* by substituting a logical value into an atomic proposition in an extended formula:
Let $\varphi\in Form$. Let $c\in\{0,1\}$. Let $p\in AP$.

$$\varphi[p\mapsto c]=\begin{cases}c & \text{if} \quad \varphi=p\\ q & \text{if} \quad \text{for some } q\neq p,\ \varphi=q\\ \neg(\varphi'[p\mapsto c]) & \text{if} \quad \varphi=(\neg\varphi')\\ \circ(\varphi_1[p\mapsto c],\varphi_2[p\mapsto c]) & \text{if} \quad \varphi=(\varphi_1\circ\varphi_2)\end{cases}$$

Now we build on the naive SAT solver algorithm to create a general algorithm that will allow us to determine whether a formula is satisfiable. In order to justify the correctness of the algorithm, we need several lemmas. The following lemma, states that by substituting a proposition with its truth assignment, we do not change the satisfiability of the formula.

Lemma 1 (Substitution)**.** *Let* $\varphi\in$ Form$,\tau\in 2^{AP(\varphi)}$*, and* $p\in AP(\varphi)$*. Then* $\tau\models\varphi \iff \tau_{_{AP(\varphi)-\{p\}}}\models\varphi[p\mapsto\tau(p)]$.

Note: $AP(\varphi[p \mapsto c]) = AP(\varphi) - \{p\}$

Reminder: For a function f, $f : X \to Y$, given $X' \subseteq X$, then $f\,|_{X'}$: $X' \mapsto Y$, defined by: if $a \in X'$ then $f\,|_{X'}\,(a) = f(a)$

Essentially, this means that if we keep simplifying, eventually we will just get 1 or 0. The proof of Lemma 1 is one of the exercises on Homework 3.

Now we can go back to satisfiability:

Corollary 2. *Let $\varphi \in$ FORM, and $p \in AP(\varphi)$. Then φ is satisfiable $\iff$ $\varphi[p \mapsto 0]$ is satisfiable or $\varphi[p \mapsto 1]$ is satisfiable.*

Proof.

$\boxed{\Rightarrow}$ Suppose that φ is satisfiable. Then $\tau \models \varphi$ for some $\tau \in 2^{AP(\varphi)}$. By the lemma above, $\tau_{_{AP(\varphi)-\{p\}}} \models \varphi[p \mapsto \tau(p)]$. But either $\tau(p) = 0$ or $\tau(p) = 1$.
So either $\varphi[p \mapsto 0]$ or $\varphi[p \mapsto 1]$ is satisfiable.

$\boxed{\Leftarrow}$

- Suppose $\varphi[p \mapsto 0]$ is satisfiable. Then $\tau \models \varphi[p \mapsto 0]$ for some $\tau \in 2^{AP(\varphi)-\{p\}}$. Define $\tau' \in 2^{AP(\varphi)}$ as follows:

$$\tau'(q) = \begin{cases} 0, & \text{if } q \text{ is } p \\ \tau(q), & \text{otherwise} \end{cases}$$

  By the Substitution Lemma (Lemma 1), $\tau' \models \varphi$. So φ is satisfiable.

- Suppose $\varphi[p \mapsto 1]$ is satisfiable.Then $\tau \models \varphi[p \mapsto 1]$ for some $\tau \in 2^{AP(\varphi)-\{p\}}$ Define $\tau' \in 2^{AP(\varphi)}$ as follows:

$$\tau'(q) = \begin{cases} 1, & \text{if } q \text{ is } p \\ \tau(q), & \text{otherwise} \end{cases}$$

  By the Substitution Lemma, $\tau' \models \varphi$. So φ is satisfiable.

□

Lemma 1 tells us that SAT is *self-reducible*, that is, if we have an algorithm for SAT, then we can turn it into a search for a satisfying truth assignment.

Lemma 1 also suggests a strategy for solving SAT. Here we present a simplified version, and the course project will be to build a very fast version.

3 Splitting Method

The algorithms works as follows:

$SAT(Form\ \varphi)$
if φ is 1 **then** return **true**
if φ is 0 **then** fail (backtrack or return **false**)
choose $p \in AP(\varphi)$, choose $c \in \{0, 1\}$
if $SAT(\varphi[p \mapsto c])$ **then** return **true**
if $SAT(\varphi[p \mapsto 1 - c])$ **then** return **true**
return **false**

The running time of this procedure is still exponential, $O(2^{|\varphi|})$, but it will usually be much more efficient than the truth-table algorithm. Note, however, that it is important how we choose p and c in this algorithm. If they are chosen poorly, the running time for this procedure will be no better than that of the simple truth table algorithm.

The space requirement is equal to the size of the call stack. The worst case stack depth is $O(|\varphi|)$, and each stack frame is of linear size. This makes the space requirements polynomial with the size of the input formula.

If we do not want to do the same work twice, we can use "memoing". That is, once we determine whether a formula is satisfiable, we record this. When we reach a formula that has already been examined, we look up the saved result. This technique saves time, but could use exponential space. In effect, we are trading space for time.

The Splitting Method is the dominant method of solving large SAT problems right now. It can be very fast in practice.

3.1 CNF

There's another approach to handling NP-complete problems, which is to look at subproblems that might be easier. A natural class of formulas is a conjunction of requirements. If we restrict these requirements to implications and disjunctions, we end up in *conjunctive normal form* (CNF). It will be shown later than every formula has an equivalent in CNF.

Definition 1. *A* literal *is a proposition or its negation. A* disjunctive clause *is a disjunction of literals. A* CNF *formula is a conjunction of disjunctive clauses. If every clause has at most k literals, the formula is in k-CNF.*

To explore the boundary of computability, we gradually increase the k in k-CNF and see how far we get. A 0-CNF formula is the set of empty constraints, and is satisfied in all worlds.

Lemma 3. *A 1-CNF formula is satisfiable iff it does not contain conflicting literals. (conflicting literals: $p \wedge \neg p$). 1-CNF-SAT is in PTIME.*

That $2-CNF$ is in PTIME, while 3-CNF is NP-complete, will be proven later in the course. Therefore, summarizing our results:

$0-CNF$-SAT: PTIME.

$1-CNF$-SAT: PTIME.

$2-CNF$-SAT: PTIME.

$3-CNF$-SAT: NP-complete.

Therefore, for $k-CNF$-SAT the boundary is between $2-CNF$-SAT and $3-CNF$-SAT.

Now let's revisit the splitting method and see how we can speed up the performance.

4 Revisiting the Splitting Method

When employing the splitting method on CNF formulas, we can devise a number of useful heuristics.

- *Unit clauses* are formulas of the form

$$p \text{ or } \neg p.$$

- The *unit preference rule* : If φ has a unit clause p, then we choose p and assign it 1 (and we can ignore $p \mapsto 0$). If φ has a unit clause $\neg p$, we choose p and assign it 0.

- *Polarity*: if a proposition p only occurs as a positive or negative literal, select p in this step and assign appropriately. This requires we track the polarity of each proposition.

- *Subsumption*: If there are two clauses c and c' where $c \subseteq c'$, then we can eliminate c'. If c is satisfied, then c' is satisfied, and if c is not satisfied then we are already have a counterexample. This potentially requires a quadratic number of comparisons at each step, and its value is not clear.

Of the above heuristics, the most useful is the unit preference rule. Using the splitting method on CNF formulas with unit preference gives us the DPLL(*Davis-Putnam-Longemann-Loveland*) algorithm.

$SAT(\varphi)$
  if φ is 1, return 1
  if φ i 0, fail and backtrack or 0
  choose $p \in AP(\varphi)$ and $c \in 0, 1$ [***Unit Preference***]
  if $SAT(\varphi[p \mapsto c])$ return true
  else return $SAT(\varphi[p \mapsto 1 - c])$