The main purpose of these notes is to help me organize the material that I used to teach today's lecture. They often contain text fragments, lots of typos, hints to myself, and imaginary questions and are often written in a style as if I were talking to someone else. They are by no means intended to be text book quality.

4.1 Review - the DPLL procedure

On Tuesday we talked about validity and satisfiability of formulas and the Davis-Putnam procedure for testing the satisfiability of a given formula in CNF. The beauty of that algorithm is that it is both simple and efficient. Let me briefly review it in a somewhat more precise formulation

Given a formula F in CNF, `DPLL(`F`)` returns a boolean value `true` or false and proceeds as follows

- Apply **Unit propagation** as long as possible:

  Identify a unit clause of the form $C_i = l$, remove all clauses containing l from F and remove the negation of l from the remaing clauses *this was inaccurate on Tuesday*
- Apply **Pure literal elimination** as long as possible:

  Identify a literal l that occurs only in one polarity and remove all clauses containing l
- If F is empty, return `true`

  If F contains the empty clause, return `false`
- **Split:** Otherwise select a literal l and set $val(l) = t$. Simplify F to F_1 by removing all clauses containing l and removing the negation of l from the remaining clauses.

  Compute `DPLL(`F_1`)`. If the result is `true`, return `true`
- Otherwise set $val(l) = f$. Simplify F to F_2 by removing all clauses containing l and removing the negation of l from the remaing clauses.

  Compute `DPLL(`F_2`)` and return the result.

Here is a somewhat longer formula to illustrate the procedure:

$(p \vee q \vee r \vee s) \wedge (\neg p \vee q \vee \neg r) \wedge (\neg q \vee \neg r \vee s) \wedge (p \vee \neg q \vee r \vee s)$
$\wedge (q \vee \neg r \vee \neg s) \wedge (\neg p \vee \neg r \vee s) \wedge (\neg p \vee \neg s) \wedge (p \vee \neg q)$

(1) No unit or pure elimination possible.

(2) We have four s-Literals, and two $\neg s$: Split using s

  Remove clauses: $(\neg p \vee q \vee \neg r) \wedge \wedge (q \vee \neg r \vee \neg s) \wedge (\neg p \vee \neg s) \wedge (p \vee \neg q)$

  Remove $\neg s$: $(\neg p \vee q \vee \neg r) \wedge \wedge (q \vee \neg r) \wedge (\neg p) \wedge (p \vee \neg q)$

(3) **Unit propagation** with $\neg p$: $(q \vee \neg r) \wedge (\neg q)$

  Unit propagation with $\neg q$: $(\neg r)$

  Unit propagation with $\neg r$ gives the empty formula, return `true`

That was pretty fast – as you can see, **Unit propagation** does most of the work for us.

But this is not always the case. We could have chosen a different branching literal, say r, first:

(1) No unit or pure elimination possible.

(2) Split using $\neg r$

Remove clauses: $(p \vee q \vee r \vee s) \wedge (p \vee \neg q \vee r \vee s) \wedge (\neg p \vee \neg s) \wedge (p \vee \neg q)$

Remove r: $(p \vee q \vee s) \wedge (p \vee \neg q \vee s) \wedge (\neg p \vee \neg s) \wedge (p \vee \neg q)$

(3) No **Unit propagation** possible

No pure literal

(4) Split using $\neg q$

Remove clauses: $(p \vee q \vee s) \wedge (\neg p \vee \neg s)$

Remove q: $(p \vee s) \wedge (\neg p \vee \neg s)$

(5) No **Unit propagation** possible

No pure literal

(6) Split using p

Remove clauses: $(\neg p \vee \neg s)$

Remove $\neg p$: $(\neg s)$

(7) No **Unit propagation** $\neg s$ gives the empty formula, return `true`

So this took quite a bit longer. Actually *The Davis-Logemann-Loveland algorithm depends on the choice of branching literal, which is the literal considered in the backtracking step. As a result, this is not exactly an algorithm, but rather a family of algorithms, one for each possible way of choosing the branching literal. Efficiency is strongly affected by the choice of the branching literal: there exist instances for which the running time is constant or exponential depending on the choice of the branching literals.*

On the average, DPLL is very fast – the cases where a wrong choice of the branching literal is the reason for exponential runtime are very rare. There are, however, formulas, where every strategy for selecting the branching variable will lead to an exponential runtime.

Example: complete formula with 3 vars - I have to go through all 8 cases to find out that the formula is not satisfiable. No matter what valuation I choos there is one clause that will become false. This means that in the worst case, the algorithm has exponential time.

Now the question is – does it have to be that way or can we avoid going through exponentially many alternatives before we figure out that the formula is not satisfiable?

Since unfortunately the answer to that question is that we cannot expect to avoid exponential runtime, the second question is what we can do about it? Is there a way to make the procedure so efficient that we can deal with significantly large formulas anyway? That is, can we push the tractability of the problem from formulas containing, say, 60 variables – where exponential runtime (that is 10^{18} = a billion billion steps) would usually become a problem – to a few hundred variables or even more? This is in fact the case an that's why SAT solvers can be used to deal with real-world problems.

4.2 Why is SAT hard?

So let's come back to the first of the two questions – why is testing for satisfiability so difficult? Why is it that all known algorithms – not just the Davis-Putnam procedure and truth tables – need exponential time to solve the SAT problem for some formulas?

Well one of the indicators is that one can encode all kinds of really difficult problems as SAT problems – finding cliques in graphs, coloring graphs, finding complete routes in a network (TSP), integer linear programming, subgraph isomorphism, graph partitioning, binpacking, processor scheduling, factorizing numbers, cracking encryptions, ... all problems for which there is no known efficient algorithm to solve them.

In fact, every problem that can be solved in polynomial time by a computer with an unlimited number of parallel threads can also be encoded as SAT problem. So if you solve SAT efficiently, you know how to solve all of these problems efficiently on a standard computer. Theoreticians call this property of SAT *NP-completeness* because they use a nondeterministic machine model to describe unlimited parallellism – we can't build such machines yet as long as quantum computers don't become reality – and have shown that SAT is the most difficult problem that can be solved with such a machine. So this gives us an indicator why solving SAT is hard – although one cannot prove that all SAT solvers *must* use exponential time in the worst case one can point to the fact that thousands of problems for which the best known algorithm is exponential are "easier" than SAT and this although tens of thousands of researchers all over the world have unsuccessfully tried to solve these problems for decades.

Before I go on giving you a rough idea how to encode nondeterministic computations as SAT problem, let me ask how many of you have heard of Turing machines, nondeterministic machines and the concept of NP-completeness *(I assume less than half)*.

A TM is one of the simplest models for computation. It can simulate every computer architecture that we use today with just a few concepts: a program with internal states and an external memory consisting of an unlimited tape and a head that can read and write single bits (or characters if you want) and move the head left or right on the tape. The basic mechanism is deterministic: for each state and symbol under the head the TM program determines a new state, a bit to write and a direction to move the head on the tape. If the machine is nondeterministic that means that the program allows it to choose arbitrarily from a given set of new states, bits, and directions. In both cases the machine starts with the input on the tape and then it starts reading bits and writing others, moving the head and switching to other internal states according to what the program tells it to do – a modern computer does essentially the same.

So how can we encode every computation problem that can be solved by a nondeterministic machine in polynomial time as a SAT problem? For simplicity we look at problems that test some behavior – like that there is a clique, a routing, a schedule, or even a valuation that makes a formula true – so the machine takes an encoding of the problem as input string and writes the answer – say 0 or 1 – onto the tape as the first bit. There are many other ways to describe this, but they're essentially all the same. Now we know that for a given input string of length n the machine runs in polynomial time $p(n)$ and will have seen at most $p(n)$ cells of the tape during that time. So we can safely ignore the rest of the tape and that is the key to encoding the whole computation as a finite formula.

The variables we need will express that cell number i contains at time t a symbol A. We call them $y_{t,i,A}$. For simplicity we consider the state of the program as the symbol of the cell where the head is at. There are $p(n)$ cells, time is limited to $p(n)$ as well and A we can be a tape symbol or a state. So a series of cells $a_0 a_1 ... a_{j-1}\, q\, a_{j+1} ... a_{p(n)}$ says that the head is at cell j, the program is in state q and the tape content is described by the a_i. Altogether we have in the order of $p(n)^2$ variables in our formula, which will express that there is a successful computation from the initial configuration to one that says "yes".

We need 4 types of CNF formulas to encode this:

(1) A CNF formula describing the **initial configuration:** Input string $s_1..s_n$ is on the tape, the rest is blank, we start in the initial state q_0 with the head at the beginning of the tape:

$$y_{0,0,q_0} \wedge y_{0,1,w_1} \wedge .. \wedge y_{0,n,w_n} \wedge y_{0,n+1,B} \wedge .. \wedge y_{0,p(n),B}$$

(2) Formulas stating the **frame conditions**, saying that at each time the head is at exactly one position (that is $y_{t,i,q}$ holds for each t and one i and q and $\neg y_{t,i',q'}$ holds for all others)

(3) Formulas describing the **transition** from time t to $t+1$. They encode that if the head was at position i (that is if $y_{t,i,q}$ is true for some state q) then the cells $i-1, i, i+1$ can change according to program table while the rest is unchanged at time $t+1$.

That leads to a very big formula for each line of the program and each time t and cell i but altogether we don't have more than $p(n)^2$ clauses.

(4) Formulas stating the **final configuration**: at time $p(n)$ the head is at the beginning in a "final" state, there is a 1 in cell 1 and the rest is blank.

All 4 types of formulas are CNF formulas that are joined together by a conjunction into one big formula of size $p(n)^3$ which is satisfiable exactly if the TM's program on input $s_1...s_n$ allows for a series of $p(n)$ computation steps which lead to a final configuration that says "yes". In fact we can determine the valuation that makes the formula true from the series of computation steps of the machine and vice versa.

The technical details of that argument are tricky - you have to describe the 4 types of formulas in detail, prove that they are in CNF and don't exceed the magnitude of $p(n)^3$, and of course that the above correspondence between valuations and computation steps is in fact correct. Because of that the full proof of "Cook's theorem" takes up quite a few pages – I want to spare you the details but if you're interested, you should take a course on complexity theory or read books about NP-completeness.

4.3 How to make DPLL efficient

The fact that you can encode *every* problem that can be solved with a solved with a NTM in polynomial time as CNF formula which is satisfiable if and only if the NTM "accepts" the given input string indicates that SAT is a really difficult problem and that we cannot expect to be able to build a SAT solver that is not exponential in the worst case.

So what can we do? After all the SAT problem is essential for many applications, so there is a strong need for efficient SAT solvers that provide computer support for these applications. Because

of that many people have worked on improving the basic DPLL algorithm and got astonishing results without changing the essential idea.

Current work on improving the algorithm has been done on four directions:

(1) Avoid copying the formula
(2) Defining new data structures to make the algorithm faster, especially the part on unit propagation.
(3) defining different policies for choosing the branching literals
(4) defining variants of the basic backtracking algorithm.

In all three areas all kinds of optimizations are possible if one utilizes the experience about efficient algorithms in the respective areas.

4.3.1 Partial valuations

Several steps of the DPLL procedure require literals to be removed from a formula. While this appears natural it becomes infeasible once we deal with millions of clauses. Instead we use *partial valuations* that assign truth values to some but not all of the variables in a formula.

Thus instead of removing a literal from a clause we simply set its value to f. Actually since we only need that in the split rule and in unit propagation, we will already have set the negation of l to t so nothing will have to be done at all.

This saves a lot of unnecessary steps.

As a result however, we need to redefine the notion of a unit clause. Instead of saying that it consists of only one literal a *unit clause is a clause where all but one of its literals have been assigned the value* f. This is equivalent to taking these literals out, but we don't have to actually modify the formula itself.

4.3.2 Specialized data structures

For applying the unit propagation we need to know which clauses have exactly one literal *that is not assigned* f

Obviously it doesn't make sense to go through the whole formula each time and count the number of these literals in each clause. But given the fact that we will have millions of clauses, it is also infeasible to maintain a list that gives us the number of literals in each clause that aren't false.

A better approach is the so-called *watched literals* method, which proceeds as follows

> For each clause C in F we select *two* yet unassigned literals, that we *watch*
>
> Then for each variable x in F we maintain two lists
> – one list of clauses (indices) where x is watched
> – one list of clauses (indices) where $\neg x$ is watched
>
> Then we can act quickly as soon as x is assigned a value
> When x is assigned the value t then check all clauses in the watch list for $\neg x$.

– Find a different variable y in that clause that can be watched (so we have two watched literals again) and move the clause to the watch list of y
– If all but one literal l in the clause are assigned f we have a unit clause and can assign t to l and recur
– Just continue if all any literal is already assigned true
– If (through another op) all literals in that clause have been assigned f stop - F is unsatisfiable

Although this seems a bit more complicated at first it reduces the number of clauses that need to inspected each time significantly.

When the algorithm needs to backtrack, watch lists need not be restored. The details for this are a bit tricky but this means we don't need to store previous versions of watch lists.

4.3.3 Policies for choosing the branching literals

Choosing the right variable for splitting has a strong effect on the runtime of the algorithm. people come up with heuristics but one has to make sure that computing the heuristic itself isn't too expensive.

In general one chooses variables that occur frequently.

4.3.4 conflict analysis: variants of the basic backtracking algorithm

Quite often the naive algorithm splits and goes into great depth to find out that the formula cannot be satisfied. then it backtracks to the previous variable and again cannot satisfy the formula. As it turns out the reason was the same as befor but the algorithm doesn't know that.

To improve backtracking one needs to be able to reuse information that is obtained in another branch.

The basic method is *clause learning*: if a conflicting clause is found generate a *conflict clause* containing all literals that are assigned false at this point. Then backtrack to the earliest decision level where one of these variables was yet unassigned. I'll leave it with that because the details again are a bit tricky but again the results are tremendous if you do it right, because we avoid a lot of unnecessary search for the satisfying valuation.

The latter direction include non-chronological backtracking and clause learning. These refinements describe a method of backtracking after reaching a conflict clause which "learns" the root causes (assignments to variables) of the conflict in order to avoid reaching the same conflict again.

4.3.5 Significant progress could be achieved

There are more low-leve improvements that one might add and that reusults in the fact that SAT solvers can now deal with tens of thousands of variables, sometimes even millions of them and up to a billion clauses. Obviously there are situations here even these procedures behave poorly even on small formulas but in general they do extremely well.