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.

3.1 Review

Last week Mark Bickford introduced formulas, valuations, validity, and satisfiability. Satisfiability and validity are the key notions because there are many applications from mathematics, physiscs and computer science that can be formulated as satisfiability problem or as validity problem. Mark gave you one example about finding triangles in graphs. A more general question is finding cliques, that is a group of totally connected nodes in a graph (or a network). Once the network gets too big, you need a computerized satisfiability solver to answer this question because you cannot solve it by hand anymore. I believe Mark showed you that these SAT solvers have become very efficient.

Validity is equally important. If you want to guarantee that a program has a given property, then this is essentially a validity problem - for all possible values of the input variables, the output must have that property. Proving this with a computer is called program verification and this is particularly important when we're dealing with safety and security issues, like controlling an airplane or encrypting messages to protect them from being read by unauthorized people.

I spent quite a bit of time just talking about this

Today we want to take a deeper look at satisfiability and validity, and at procedures that can be used to solve problems.

3.2 The truth table method

The simplest – but probably not the most efficient – way to check whether a formula `X` is valid or satisfiably, is building a truth table. That is, we have to compute the value of `X` for all possible interpretations of the formula's variables. To do this systematically, we build a table where we write down all possible combinations of values `t` and `f` for the variables of the formula and then compute the value for all subformulas `Y` of `X`, including `X` itself.

You have seen an example of that method last week and I'd like to review this method by considering another small example formula $(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$.

P	Q	$P \Rightarrow Q$	$\neg Q$	$\neg P$	$\neg Q \Rightarrow \neg P$	$(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$
t	t	t	f	f	t	t
t	f	f	f	t	t	t
f	t	t	t	f	f	t
f	f	t	t	t	t	t

The formula is a tautology because every row ends with t. If only a single row would result in `f`, it would not be a tautology anymore. However, it could still be *satisfiable*.

The truth table method is simple, but has one severe drawback. The tables grow incredibly fast even for relatively small formulas. The number of rows grows *exponentially in the number of variables* and additionally the number of columns is linear in the number of subformulas, which is roughly the number of connectives plus the number of variables. The example of the formula $(P \vee (Q \wedge R)) \Rightarrow ((P \vee Q) \wedge (P \vee R))$ shows how quickly a truth table may grow.

P	Q	R	$Q \wedge R$	$P \vee (Q \wedge R)$	$P \vee Q$	$P \vee R$	$(P \vee Q) \wedge (P \vee R)$	$(P \vee (Q \wedge R)) \Rightarrow ((P \vee Q) \wedge (P \vee R))$
t	t	t	t	t	t	t	t	t
t	t	f	f	t	t	t	t	t
t	f	t	f	t	t	t	t	t
t	f	f	f	t	t	t	t	t
f	t	t	t	t	t	t	t	t
f	t	f	f	f	t	f	f	t
f	f	t	f	f	f	t	f	t
f	f	f	f	f	f	f	f	t

Truth tables were based on the definition that *a formula is a tautology if it is true under every interpretation*. However, using truth tables for evaluating the truth of a formula quickly becomes infeasible. For larger formulas, we should therefore look for a better method – something that is fairly schematic, but doesn't rely checking individual valuations anymore.

Before we go into that I would like to investigate a few theoretical issues

3.3 How many truth tables are there?

2^{2^n} - way to many to store them for table lookup

example n=2 - write down all 4 options, then assign truth values systematically to get 16 entries, identify them by name. *I went through half of them, asking them to identify the rest*

Sheffer Stroke $X|Y$ is true iff one of X and Y is false.

3.4 How many boolean operations do we need?

We have identified 16 different boolean operations on 2 variables, there will be 256 on 3 variables and even more on 4, 5 etc. It will obviously be impossible to check the satisfiability of a formula with hundreds of different operations. So how many do we actually need?

We call a boolean operation op ***definable*** from $op_1, ..op_k$ if it can be expessed in terms of $op_1, ..op_k$, that is if $op(X_1, ..X_n)$ is functionally equivalent to a formula that only uses $op_1, ..op_k$.

When we look at truth tables it becomes obvious that all n-ary boolean operations can be expressed by $\neg$, $\vee$, and $\wedge$.

Example implication $p \Rightarrow q \equiv \neg p \vee q$

Show everythis is definable via xor – what do we have to do? *I forgot to do this*

Homework: all via Sheffer Stroke

3.5 Generating Normal forms

TT yields DNF (complexity?): Build whole table - space is 2^n rows, $n+1$ entries

Conversion via De Morgan?: provide 3 laws to move $\neg$ inside and $\vee$ to the outside

Conversion to CNF is exponential in the wost case

SAT is easy with DNF (make one of the rows true, find the valuation from the clause)

SAT procedures usually do NOT start with DNF but often with CNF (a Matrix, usually shorter). That makes it difficult, because we can't look at just a single clause anymore. While it is true, taht the formula is unsatisfiable, if one clause is this is not the real issue. You will hardly have a problem formulation where a single clause is already unsat (when would that be the case??). It is the interaction between the clauses that makes it difficult. We have to find a valuation that makes all clauses true. If for each valuation one of these CNF clauses evaluates to false, then the formula is unsatisfiable.

3.6 Why can we often find efficient solutions anyway?

3.7 Davis Putnam

The DPLL/Davis-Putnam-Logemann-Loveland algorithm is a complete, backtracking-based algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form, i.e. for solving the CNF-SAT problem.

It was introduced in 1962 by Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland, and is a refinement of the earlier Davis-Putnam algorithm, which is a resolution-based procedure developed by Davis and Putnam in 1960. Especially in older publications, the Davis-Logemann-Loveland algorithm is often referred to as the Davis-Putnam method or the DP algorithm. Other common names that maintain the distinction are DLL and DPLL.

DPLL is a highly efficient procedure, and after more than 40 years still forms the basis for most efficient complete SAT solvers, as well as for many theorem provers for fragments of first-order logic.

The basic backtracking algorithm runs by choosing a literal, assigning a truth value to it, simplifying the formula and then recursively checking if the simplified formula is satisfiable; if this is the case, the original formula is satisfiable; otherwise, the same recursive check is done assuming the opposite truth value. This is known as the splitting rule, as it splits the problem into two simpler sub-problems. The simplification step essentially removes all clauses which become true under the assignment from the formula, and all literals that become false from the remaining clauses.

Example: $C_1 \wedge C_2 \wedge C_3$ ... (make one up with 4 variables p, q, r, s)

- if X is empty return satisfiable (why?: no requirements to satisfy)
- if X contains an empty clause then return unsatisfiable
- Simplify using the techniques below

- Set p to be true (split)
- Drop every clause containing p (why?: C_i evals to true, we don't need it anymore)
- Drop every $\neg p$ from the remaining clauses (why?: if C_j evals to true, then it doesn't have to do with $\neg p$, which is false)
- The formula doesn't contain p anymore

  Repeat splitting the remaining variables in the remaining formula X'

  If the result is that formula X is unsatisfiable set p to be false and do the same

The DPLL algorithm enhances over the backtracking algorithm by the eager use of the following rules at each step:

Unit propagation If a clause is a unit clause, i.e. it contains only a single unassigned literal, this clause can only be satisfied by assigning the necessary value to make this literal true. Thus, no choi ce is necessary. In practice, this often leads to deterministic cascades of units, thus avoiding a large part of the naive search space.

Drop $C_j = q$ and every clause containing q then drop every $\neg p$ from the remaining clauses

Pure literal elimination : If a propositional variable occurs with only one polarity in the formula, it is called pure. Pure literals can always be assigned in a way that makes all clauses containing them true. Thus, these clauses do not constrain the search anymore and can be deleted. While this optimization is part of the original DPLL algorithm, most current implementations omit it, as the effect for efficient implementations now is negligible or, due to the overhead for detecting purity, even negative.

Drop all clauses containing pure literals.

Both improvements are obvious, when one looks at the DPLL procedure from the perspective: what can I do before I split? Both look at situations where the splitting rule runs into a special case where we don't need to backtrack. Unit propagation means the literal in the clause MUST be made true (we don't have a choice) , pure literal elimination means we have a literal that SHOULD be made true because making it false will not contribute to the solution.

Unsatisfiability of a given partial assignment is detected if one clause becomes empty, i.e. if all its variables have been assigned in a way that makes the corresponding literals false. Satisfiability of the formula is detected either when all variables are assigned without generating the empty clause, or, in modern implementations, if all clauses are satisfied. Unsatisfiability of the complete formula can only be detected after exhaustive search.