Algorithmic Modeling and Complexity **Fall 2003**

Lecture 11: 7 October

Lecturer: J. van Leeuwen *Scribe: Munish Mahabiersing*

11.1 Overview

Today's lecture was about aspects of working with formulae. Formulae are expressions in some formal framework like propositional or predicate logic. A formula $\phi(x_1, \ldots, x_n)$ describes a 'statement' or 'desired fact' by means of relation and function symbols, with $x_1, \ldots, x_n$ the 'free' variables. Formulae only have meaning when interpreted in a *model.* One is typically interested in questions like: given a formula ϕ, is there a model and are there 'values' for the variables in the model such that $\phi(x_1, \ldots, x_n)$ holds. This is known as the *satisfiability problem.* The problem is also interesting when a model is given. The lecture considered the satisfiability problem for propositional formulae, i.e. *boolean formulae* where the variables can only be true (1) or false (0). Determining whether a set of boolean formulae is satisfiable or not is a central problem in algorithmic modeling and complexity.

11.2 The satisfiability problem

The satisfiability problem is of long-standing interest in logic. For example, a classical result in both propositional or predicate logic links the notion of satisfiability with the formal notion of consistency ('no contradiction can be derived from the formulae').

Theorem 11.1 *A set of formulae is satisfiable if and only if it is consistent.*

In first order (predicate) logic, the satisfiability problem is *not* algorithmically solvable ('undecidable') but in propositional logic it is: the question whether there exist $x_i \in \{0, 1\}$ such that $\phi(x_1, \ldots, x_n)$ holds is easily decidable by checking all 2^n possible valuations. Can one do better?

11.2.1 Classes of formulae

In order to study the satisfiability problem for boolean formulae further, we restrict ourselves to formulae in a suitable standard form. Define a *literal* to be any variable or its negation: x_i or $\overline{x}_i$ for any i.

Definition 11.2 *A boolean formula is in so-called conjunctive normal form or 'clause form' if it is the conjunction (the 'and') of a number of clauses, with every clause being the disjunction (the 'or') of a set of literals.*

Fact 11.3 *Every boolean formula ϕ is equivalent to a formula in clause form, which can be computed from ϕ in easy polynomial time.*

We will henceforth restrict ourselves to the satisfiability problem for sets of clauses, denoted by SAT. There were several special further cases:

> 2SAT: the satisfiability problem for sets of clauses with at most 2 literals per clause.
>
> 3SAT: the satisfiability problem for sets of clauses with at most 3 literals per clause.
>
> kSAT: the satisfiability problem for sets of clauses with at most k literals per clause.

In 2SAT, 3SAT and kSAT we usually assume that all clauses have exactly 2, 3 or k literals, respectively.

11.2.2 Boolean formulae as decision problems

Propositional formulae are interesting because they can be used in 'encoding' quite a few (instances of) decision problems that arise e.g. in Operations Research. Even 2SAT can arise. Consider the *M-respecting Vertex Cover problem* or simply: M-Vertex Cover problem, defined as follows:

> Let $G =< V, E >$ be a network, and M a matching in G. Determine whether there exists a vertex cover $C \subseteq V$ with the property that for every edge (u, v), if $(u, v) \in M$ then EITHER u OR v belongs to the cover.

In an M-vertex cover the matching M forces that the 'streets' belonging to M are watched by precisely *one* guard. Note that M can be arbitrary (e.g. non-maximal) and that every vertex cover of G must contain at least one node of every edge in M.

Lemma 11.4 *2SAT is 'equivalent' to the M-Vertex Cover problem.*

Proof: ($\Rightarrow$) Take an instance ϕ of 2SAT. Reduce ϕ so it contains no clauses with 1 literal (clauses with 1 literal force truth values trivially). Thus assume w.l.o.g. that ϕ has 2 literals per clause. Create a network $G(\phi)$ with $2n$ nodes, one for every literal x_i and $\overline{x}_i$ occurring in ϕ, and edges $(x_i, \overline{x}_i)$. Add edges (l_1, l_2) for literals $l_1 \neq l_2$ if $(l_1 \vee l_2)$ is a clause in ϕ. Take $M = \{(x_i, \overline{x}_i) | 1 \leq i \leq n\}$, which is obviously a (maximum) matching. If ϕ is satisfiable, then the 'true' literals form an M-vertex cover. Conversely every M-vertex cover of $G(\phi)$ corresponds to a truth value assignment (verify) that satisfies ϕ (as every clause is 'covered' by at least one node of the M-respecting cover).

($\Leftarrow$) Given a network G and a matching M in G, construct a formula ϕ as follows. Let $|M| = k$. Label the endpoints of the ith edge of M by x_i and $\overline{x}_i$ $(1 \leq i \leq k)$ and give labels $x_{k+1}, \ldots$ to the remaining $n - 2k$ nodes of G. Let ϕ consist of the clauses $(l_1 \vee l_2)$ for which (l_1, l_2) is an edge in G. One easily argues that G has an M-respecting vertex cover if and only if ϕ is satisfiable. ■

In the next section we will see a method for solving the M-Vertex Cover problem.

11.3 Solving the 2SAT problem

While SAT is computationally hard, 2SAT actually turns out to be efficiently solvable!

Lemma 11.5 (Cook, 1971) *The 2SAT problem can be decided in polynomial time.*

Proof: Consider any formula ϕ in clause form with ≤ 2 literals per clause. The following procedure will decide whether ϕ is satisfiable or not.

Choose any literal α that occurs in ϕ and that hasn't yet received a truth value. Set $\alpha := true$ (and consequently $\overline{\alpha} := false$). Substituting $\alpha = true$ in ϕ eliminates clauses and forces several further truth value assignments in a 'clean-up round' as follows. Every clause of the form (α), $(\alpha \vee \beta)$ or $(\beta \vee \alpha)$ becomes satisfied and can be omitted. Every clause of the form $(\overline{\alpha} \vee \beta)$ or $(\beta \vee \overline{\alpha})$ can be reduced and thus replaced by (β). If a clause $\overline{\alpha}$ occurs, the assignment is declared a failure. As long as the assignments are successful, continu to reduce the formula by (necessarily!) setting every β that occurs as a clause by itself to true, and repeating the clean-up rounds of eliminating and replacing clauses. The end result is either that the assignment is a failure or that we obtain a subset ϕ' of the clauses of ϕ with the property that ϕ is satisfiable if and only if ϕ' is.

In case the assignment was a failure, try again with $\alpha := false$. If this assignment is also declared a failure, then ϕ is unsatisfiable. Thus, assume (at least) one of the assignments to α was successful and ended with a subset ϕ' of ϕ. Note that ϕ' is a set of clauses over variables that were not yet assigned to, i.e. is a subset of the clauses in ϕ that was not 'touched' at all. Continue with the same procedure applied to ϕ'. If ϕ' proves satisfiable then so is ϕ, and vice versa.

Observe that every clean-up round takes linear time per substituted truth value, thus the clean-up rounds together take quadratic time in the worst case. The whole algorithm takes at most $2 \cdot n$ clean-up rounds, where n is the number of different variables in ϕ. ■

The procedure underlying the lemma can be cast in a much neater, and more efficient, form.

Theorem 11.6 (Aspvall, Plass, Tarjan, 1979) *The 2SAT problem can be decided in linear time.*

Proof: Let ϕ be an arbitrary boolean formula with ≤ 2 literals per clause. Let the variables used in ϕ be $x_1, \ldots, x_n$ and assume w.l.o.g. that ϕ has exactly 2 literals per clause. Create a directed network $G(\phi) = \langle V, E \rangle$ as follows:

> V: the set of vertices $x_1, ..., x_n$ and $\overline{x}_1, \ldots, \overline{x}_n$ corresponding to the literals in ϕ.
>
> E: the set of edges with edges $(\bar{l}_1, l_2)$ and $(\bar{l}_2, l_1)$ whenever $(l_1 \vee l_2)$ is a clause in ϕ.

Observation 1: *If there is an arc from α to β in $G(\phi)$, there is a clause $\overline{\alpha} \vee \beta$ in ϕ.*

Observation 2 (symmetry): *If there is an arc from α to β in $G(\phi)$, then there is an arc from $\overline{\beta}$ to $\overline{\alpha}$. By transitivity, this also holds for directed paths.*

Claim 11.7 *ϕ is unsatisfiable if and only if there is an x such that there are directed paths from x to $\overline{x}$ and vice versa.*

Proof: ($\Leftarrow$) Suppose nevertheless that ϕ is satisfiable. Consider the satisfying truth value assignment. Then x is, for example, set to true. Consider the directed path from x ('true') to $\overline{x}$ ('false'). Along the path one must pass an edge (α, β) such that α is true and β is false. By Observation 1, $\overline{\alpha} \vee \beta$ is a clause in ϕ and it would be false. This is a contradiction. When x were set to false, consider the path from $\overline{x}$ to x.

($\Rightarrow$) Conversely, suppose no x exists for which these directed paths exist. (Note that by Observation 2 neither path exists if at least one of them does not exist.) We claim now that formula ϕ is satisfiable! To prove this we will show that all variables can be given a truth value that satisfies ϕ. We do this by induction.

Pick a literal α that didn't get a truth value assigned yet. Set α to true. Set all literals to true that can be reached by a directed path from α. Also, set the negations of all these literals to false: these are precisely all the literals that are reachable 'backwards' from $\overline{\alpha}$ (i.e. from which $\overline{\alpha}$ can reached by a directed path). No conflicts arise in the assignment:

a: *the literals that receive a value 'true' did not receive a value 'false' earlier, the literals that receive a value 'false' did not receive a value 'true' earlier.*

Say γ received a truth value true. Then γ is reachable by a directed path from α. If γ had received a value false earlier then so would α, because all nodes 'backward reachable' from γ get the same value. Contradiction. A similar argument holds in case γ received a truth value false.

b: *no literals β and $\overline{\beta}$ can get the same value assigned.*

Suppose both β and $\overline{\beta}$ are assigned the value true. Then there is a directed path from α to β and a directed path from α to $\overline{\beta}$. By Observation 2, the latter implies a path from $\overline{\overline{\beta}} = \beta$ to $\overline{\alpha}$. Composing the two paths gives a path from α to $\overline{\alpha}$. Contradiction. A similar argument applies if β and $\overline{\beta}$ were both assigned false.

Repeat until all literals get a suitable truth value assigned. This proves that ϕ is satisfiable. Contradiction! ■

The claim holds the key to an efficient procedure for deciding satisfiability, for it shows that ϕ is satisfiable if and only if for no literal x both x and $\overline{x}$ belong to the same strongly connected component of $G(\phi)$. The strongly connected components can be computed in $O(|G(\phi)|) = O(|\phi|)$ time, and the necessary check merely takes a traversal of every strongly connected component and marking in a boolean array which literals are encountered. This is a linear-time procedure. ■

Exercise. Show that the proof of Theorem 11.6 implies a method for finding a satisfying assignment, if one exists.

Recall the M-Vertex Cover problem from the previous section.

Corollary 11.8 *The M-Vertex Cover problem is solvable in linear time.*

11.4 Solving the 3SAT problem

The 3SAT problem is the version of the SAT in which every clause can have at most 3 literals. This version of SAT appears to be much harder to solve: *no polynomial-time algorithm is presently known for it*, despite many attempts to find one over the past thirty years. Perhaps this shouldn't be surprising, as the following lemma shows that 3SAT carries 'all the complexity' of the fully unrestricted satisfiability problem.

Lemma 11.9 *SAT can be 'reduced' to 3SAT.*

Proof: Consider any instance ϕ of SAT. Consider any clause C of ϕ with more than 3 literals, say $C = (l_1 \vee l_2 \vee l_3 \vee \ldots \vee l_{k-1} \vee l_k)$. Allocate $k-2$ new variables $y_1, \ldots, y_{k-2}$ and consider the clause

$$C' = (l_1 \vee l_2 \vee y_1)(\overline{y}_1 \vee l_3 \vee y_2)(\overline{y}_2 \vee l_4 \vee y_3)\ldots(\overline{y}_{k-2} \vee l_{k-1} \vee l_k)$$

One easily verifies: C is satisfiable if and only if C' is satisfiable. Replacing every clause with > 3 literals by its equivalent form with 3 literals per clause, reduces ϕ to an equivalent instance of 3SAT. ■

11.4.1 Reducing 3SAT to other models

SAT and 3SAT are implicitly 'present' in many other frameworks of algorithmic modeling, which is often seen as an explanation of their computational intractability. We give two examples, one in networks and one in Integer Linear Programming.

Theorem 11.10 *The 3SAT problem 'reduces' to (the decision version of) the Vertex Cover problem.*

Proof: Let ϕ be a formula in clause form with ≤ 3 literals per clause. We assume w.l.o.g. that every clause contains exactly 3 literals. Let ϕ have m clauses and use n different variables, in positive or negated form. We construct an instance G, K of the Vertex Cover problem as follows:

A. for each variable x such that x or $\overline{x}$ occurs in ϕ, create two corresponding nodes and an edge $(x, \overline{x})$.

B. for every clause $(\alpha \vee \beta \vee \gamma)$ in ϕ, create a 'triangle' with nodes corresponding to α, β, γ and edges $(\alpha, \beta), (\beta, \gamma), (\gamma, \alpha)$. Furthermore, connect every node ('literal') of a triangle to the corresponding literal in part A by a direct edge.

Observe that a vertex cover for part A requires at least n nodes, for part B at least $2m$ nodes. Take $K = n + 2m$. We claim that ϕ is satisfiable if and only if G has a vertex cover of size $\leq K$.

Suppose ϕ is satisfiable. Construct a vertex cover as follows. First, include the n nodes in part A that correspond to a true literal. Every clause must contain a true literal, thus the edge between this node and the corresponding literal in part A is covered. Include the remaining two nodes of every triangle in the cover. This covers all the remaining edges. The cover has size K.

Conversely, suppose G has a vertex cover with $\leq n + 2m$ nodes. Then the cover necessarily has n nodes in part A, one node of each pair x and $\overline{x}$, and 2 nodes per triangle. Consider the truth value assignment that gives value true to each literal in the vertex cover in part A. Consider any triangle in part B. As two nodes of the triangle belong to the vertex cover, the edge connecting the third node to a part A node must be covered by the node in part A, i.e. by a true literal. Thus every clause contains a true literal and ϕ is satisfiable. ■

Theorem 11.11 *The 3SAT problem 'reduces' to (the decision version of) 0-1 Linear Programming.*

Proof: Let ϕ be a formula consisting of clauses $C_1, \ldots, C_m$ and let its constituent variables be $x_1, \ldots, x_n$. (We will not specifically use that the clauses have ≤ 3 literals per clause.) Let C_j^+ be the set of indices of positive literals (x) in C_j, and C_j^- the set of indices of negative literals ($\overline{x}$) in C_j.

Introduce an indicator variable $y_i \in \{0, 1\}$ that expresses whether x_i is set to false (0) or true (1), and introduce an indicator variable $z_j \in \{0, 1\}$ that expresses whether C_j is satisfied or not. Consider the 0-1 linear program P defined as follows:

$$\max \quad z = \sum_{j=1}^{m} z_j$$

subject to

$$\begin{array}{ll} \sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) \geq z_j & \text{for every } 1 \leq j \leq m \\ y_i \in \{0, 1\} & \text{for every } 1 \leq i \leq n \\ z_j \in \{0, 1\} & \text{for every } 1 \leq j \leq m \end{array}$$

The 0-1 LP model P actually 'counts' the largest number of clauses in ϕ that can be simultaneously satisfied. One verifies: ϕ is satisfiable if and only if P has a solution $\geq m$. ■

In a later lecture we will argue that the converse of Theorems 11.10 and 11.11 holds as well: there are polynomial time computable 'reductions' from (the decision version of) the Vertex Cover problem and of 0-1 Linear Programming to 3SAT also.

11.5 The Davis-Putnam procedure

In the meantime, SAT or 3SAT instances ϕ must be solved. A practical method was proposed already in 1960 by Davis and Putnam. It is a branching algorithm: it grows a branching tree with nodes labeled by ‘reduced’ instances of ϕ. The aim is to arrive at an ‘empty’ node, which signals that ϕ is satisfiable ‘along the chosen path in the branching tree’.

Before explaining the branching rule, we design a procedure to ‘reduce’ a formula in clause form by some self-evident clean-up steps.

```
Clean-up

will be called on some formula $\phi$ in clause form while some literals have already received a truth value

while at least one rule still applies do

    iterate

    one-literal clause rule: if $(\alpha)$ is a clause, set $\alpha := true$, remove all clauses containing $\alpha$, and delete all occurrences of $\overline{\alpha}$ from other clauses.

    affirmative negative rule: if literal $\alpha$ occurs in some clauses but $\overline{\alpha}$ does not, set $\alpha := true$, and delete all clauses containing $\alpha$.

    note that a contradiction can arise in the preceding rules, if any of the assignments sets a truth value that contradicts with an earlier assignment to a literal

    subsumption rule: remove every clause $C$ for which there is another clause $D$ that contains all literals of $C$.

if any contradiction has arisen then declare failure and exit

if $\phi = \emptyset$ then declare ‘satisfiable’ else return the reduced set of clauses.
```

Note that the clean-up procedure is very similar to, and even more effective than the clean-up rounds designed in Lemma 11.3: the clean-up is completely forced c.q. justified by the intention to look for a satisfying assignment for ϕ.

The branching rule will repeatedly pick a node of the branching tree and carry out the following rule:

Branching rule. Given (non-empty) formula ϕ at the node, do: Clean-up. If Clean-up results in failure, then label the node as ‘failure’. If Clean-up results in ‘satisfiable’, then label the node by $\emptyset$ (declare ‘satisfiable’).

If Clean-up did not result in either conclusion, let ϕ denote the reduced formula resulting from it. Choose a variable x such that *both* x and $\overline{x}$ are present in ϕ (note: after Clean-up, all variables that still occur have this property). Split the formula into two reduced formulae:

- ϕ': ϕ with x set to true, i.e. with all clauses containing x removed, and all occurrences of $\overline{x}$ deleted from other clauses.
- ϕ'': ϕ with x set to false: i.e. with all clauses containing $\overline{x}$ removed, and all occurrences of x deleted from other clauses.

The branching algorithm essentially tries all possible truth values but does so while cleaning up a formula as rigorously as possible after every step. It is clear that every path down the branching tree will ultimately end, after at most n branchings, where n is the number of variables in ϕ.

Theorem 11.12 (Davis and Putnam, 1960) *ϕ is satisfiable if and only if the branching algorithm declares it satisfiable.*

It is known that despite its efficient clean-ups, the Davis-Putnam procedure still requires exponential time on infinitely many formulae. On the other hand, when applied to 2SAT the Davis-Putnam procedure essentially does what we did in the proof of Lemma 11.5 and perhaps even more efficiently so.

11.6 Further remarks

The complexity of SAT, 3SAT and kSAT continues to receive much attention because of the crucial role in algorithmic modelling. Competitions are held between the best 3SAT-solvers, and far more powerful techniques have been applied to it than discussed in this lecture. Among the best worst-case results is O. Kullmann's: 3-SAT instances over n variables can be decided within $O(1.5045^n)$ time.

References

[1] B. Aspvall, M.T. Plass, R.E. Tarjan. A linear-time algorithm for testing the truth odf certain quantified Boolean formulas, *Information Processing Letters* 8 (1979) 121-123.

[2] S.A. Cook. The complexity of theorem proving procedures. In: *Proc. 3rd Annual ACM Symposium on Theory of Computing*, ACM Press, New York, 1971, pp 151-156.

[3] M. Davis, H. Putnam. A computing procedure for quantification theory, *Journal of the ACM* 7 (1960) 201- 215.

[4] M.R. Garey, D.S. Johnson. *Computers and intractability - A guide to the theory of NP-completeness*, W.H. Freeman and Company, San Francisco, 1979.

[5] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theor. Comput. Sci.* 223 (1999) 1-72.

[6] D.W. Loveland. *Automated theorem proving: A logical basis.* North-Holland, Amsterdam, 1978.

[7] SatLive. *Website at* http://www.satlive.org/index.jsp