FINAL REVIEW: CSC427-212

BURTON ROSENBERG
UNIVERSITY OF MIAMI

Contents

1. NP Completeness 1
1.1. Reductions 1
1.2. SAT to 3SAT 2
1.3. 2SAT 4
1.4. k-Clique and k-Cover 4
2. Verification algorithms 4
2.1. Non-determinism is implicated 5
2.2. Examples 5
2.3. Examples of the complement problem 5
3. Turing Machines 5

1. NP Completeness

1.1. **Reductions.** The final will have problems concerning NP Completeness. The theory uses the notation of polynomial time reductions between problems, notated as $A \leq_P B$, when problem A reduces to problem B. The final can ask about reductions.

- A P-time algorithm that solves B is a map,

$$f : B \rightarrow \{\, T, F \,\}$$

such that,

$$f(b) = \begin{cases} T & b \in \mathcal{L}(B), \\ F & \text{else} \end{cases}$$

- If $h : A \rightarrow B$ is a reduction then,

$$h(a) = \begin{cases} \in \mathcal{B} & \text{if } a \in \mathcal{L}(a), \\ \notin \mathcal{B} & \text{otherwise.} \end{cases}$$

Date: May 2, 2021.

- Therefore the composite function $f \circ h$ is a P-time solution to A,

$$(f \circ h)(a) = f(h(a)) = \begin{cases} T & a \in \mathcal{L}(A), h(a) \in \mathcal{L}(B) \\ F & \text{otherwise} \end{cases}$$

1.2. **SAT to 3SAT.** The Cook-Levin Theorem showed that SAT was NP-complete. The book goes back to modify the first presentation of the Cook-Levin Theorem to show 3SAT is also NP-complete. Because the Cook-Levin Theorem produces only certain boolean formulas, it was not necessary to prove 3SAT NP-complete by a reduction from SAT to 3SAT. However in this section I outline this reduction.

It is first noted that without loss of generality the SAT instance can be a boolean formula using only operations $\wedge, \vee$ and $\neg$. Operations such as exclusive or, equality, implication, all can be rewritten using the three logical operations mentioned.

Further, a SAT instance is a valid formula according to the syntax of a boolean formula, hence has a parse tree. We will work entirely with the parse tree for the the SAT instance.

The steps are a follows,

(1) Remove any internal $\neg$ nodes. Use DeMorgans to on any $\neg$ to move it layer by layer to the leaves. At the leaves absorb it into the polarity of the variable.
(2) Use distributivity to move towards the root any $\wedge$ node that is the child of an $\vee$ node.
(3) The above steps reduces us to CNF. Then use chaining variables, as in the textbook, to further reduce the CNF to 3SAT.

1.2.1. *From SAT to CNF.* The parse tree for the SAT instance has,

- The leaves are marked with variables, of the form either x or $\neg x$, the second meaning to insert at that node the logical not of the value of x.
- binary nodes internal to the tree are marked with $\vee$ or $\wedge$ to indicate whether to OR or AND the values calculated at the two child nodes,
- a nodes with just a single child marked with $\neg$

The process is shown in the next three diagrams.

For all the NOT nodes, use DeMorgan's to bring them to the leaves, see figures 1, 2, and 3, After a finite number of steps, the tree now has only OR and AND nodes.

The tree should have all the AND nodes appear in the tree closer to the root than any OR node. If there is an OR node which has a child that is an AND node, use the distribution law to correct, see figure 4.

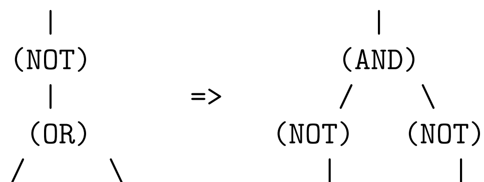

FIGURE 1. Moving NOT over an OR

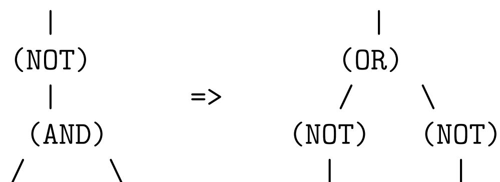

FIGURE 2. Moving NOT over an AND

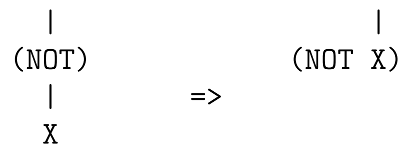

FIGURE 3. Moving NOT into a variable

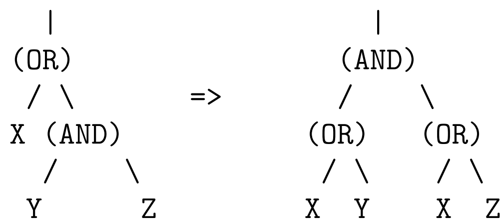

FIGURE 4. Moving AND up past an OR

1.2.2. *CNF to 3SAT.* The argument is laid out in the book how to reduce CNF to 3SAT. As an example,

$$x \vee y \vee z \vee t = (x \vee y \vee s) \wedge (\neg s \vee z \vee t)$$

You should check this works. The variables s and $\neg s$ are called chaining variables.

To generalize the example, if there are k variables in the clause, $k > 4$, this construction is used to move the two of the k variables to a clause, plus a newly introduced chaining variable, and a $k-1$ variable clause consisting of the remaining $k-2$ variables of the clause, plus the chaining variable in negation.

1.3. **2SAT.** It is not a coincidence that this chaining construction yields 3 variable clauses. The 2SAT problem is a CNF where each clause has two variables, and it is solvable in P-time. You should think about 2SAT and why with only two variables per clause, the problem becomes "easy".

Note that 3SAT does apply when some or all of the clauses have only one or two variables. Formally, turn a clause of one or two variables into one with three variables by repeating one of the variables,

$$x \vee y \Longrightarrow x \vee x \vee y$$

This points out an import fact. That repeating of variables shows

$$\text{2SAT} \leq_P \text{3SAT}$$

because 3SAT is harder than 2SAT. That is the reason for the inequality sign — it points upwards in a difficulty hierarchy.

Although there are problems harder than any in NP, the NP-complete problems can be considered the "top" problems in the $\leq_P$ ordering restricted to NP problems. The easiest problems are the two trivial languages of the empty set and the universal set. In the first case, an algorithm solving the language ignores the input and answers F, in the second the algorithm ignores the input and answers T.

1.4. **k-Clique and k-Cover.** These two reductions, in the textbook and covered in class, should be studied, and perhaps memorized.

2. Verification algorithms

We studied two forms language recognition, given a language $\mathcal{L} \subseteq \Sigma^*$. Decision calculates the decision without help in the form of a witness, or certificate or hints. Verification permits an additional string, really "pulled out of the air", that can help compute the solution.

- **Decision:** A Turing Machine M_D such that

$$M_D(x) = T \text{ if and only if } x \in \mathcal{L}.$$

- **Verification:** A Turing Machine M_V such that if

$$M_V(x, y) = T \text{ if and only if } x \in \mathcal{L}$$

  for some y in a witness set W.

2.1. **Non-determinism is implicated.** There is a connection with non-determinism. A verification machine can be a non-deterministic machine on just the problem instance x. The witnesss y is written down non-deterministically by the non-deterministic TM and then begins the deterministic verification TM.

2.2. **Examples.** We should be familiar with particular verification instances. Here are some examples to think about,

- The problem of whether an integer n is a composite. What is the witness?
- The problem of whether, given a set of integers, S, and an integer, s, there is a subset $K \subseteq S$ such that the sum of all integers in K is s.

2.3. **Examples of the complement problem.** Note that co-situation. This is important for clarification. Here are the co-verification problems connected with the above two problems,

- The problem of whether an integer n is a prime. What is the witness?
- The problem of whether, given a set of integers, S, and an integer, s, for no subset $K \subseteq S$ the sum of all integers in K is s.

In addition, remember that verification is wider than decision. A decision is a verification that does not need to use the witness.

3. Turing Machines

Programs (expressed in the TM language of the problem sets, although you are free to use the notation in the book) that recognize regular languages are something that are easy to write.

Writing TM programs for CFL's are a bit harder, and generally require non-determinism.

We have several techniques for TM programs, such as zig-zagging to match pairs of symbols.