Lecture 15

October 7, 2005

1 Review

In the last lecture we discussed how we can use automata to represent formulas. Here we continue this discussion and focus on the problem of constructing the DFA that corresponds to some formula.

Definition 1 (Emptiness Problem). *Given an automaton A, $L(A) \stackrel{?}{=} \emptyset$.*

We can check whether a DFA is empty by verifying that there is no path from the start state to an accepting state. There are well-known graph-theoretical algorithms that can help us look for a path, however, we first need to be able to "convert" an automaton into a graph. The next definition does precisely this.

Definition 2 (G_A). *Let $A = (\Sigma, S, s_0, \rho, F)$ be a DFA. Define the graph G_A as (S, E_A), where*

$$E_A = \{(s,t) : t = \rho(s,a) \text{ for some } a \in \Sigma\} \ .$$

Next we prove that by studying graph reachability we can answer whether the automaton generates an non-empty language.

Lemma 1 (Emptiness as Reachability). *Let A be an automaton. Then $L(A) \neq \emptyset$ iff there is a path from s_0 to F in G_A.*

Proof. We prove the "if" and "only if" parts separately.

$\boxed{\Rightarrow}$ Suppose that $L(A) \neq \emptyset$. Then there is some word $a_1 a_2 \cdots a_n \in L(A)$ and a sequence of states $s_0 s_1 \cdots s_n$ such that $s_{i+1} = \rho(s_i, a_{i+1})$ for $0 \leq i \leq n-1$ and $s_n \in F$. By definition of G_A there is a path from s_0 to s_n in G_A, namely, the path that we would traverse we follow the nodes $s_0 s_1 \cdots s_n$ in this order.

$\boxed{\Leftarrow}$ Suppose that $s_0 s_1 \cdots s_n$ is a path in G_A from s_0 to $s_n \in F$. Since there is an edge in G_A from s_i to s_{i+1} for $0 \leq i \leq n$, there is a letter $a_i \in \Sigma$ such that $s_{i+1} = \rho(s_i, a_i)$ for $0 \leq i \leq n$. Then $s_0 s_1 \cdots s_{n-1} \in L(A)$.

Notice that the proof doesn't require A to be deterministic. The property holds for NFAs and DFAs. $\square$

By Lemma 1, an algorithm to verify emptiness can simply be a breadth-first search on the edges of the graph to see if there is a path from s_0 to a final state. Thus, the problem is reduced to a simple graph-reachability problem, which is solvable in linear time. This justifies the following corollary:

Corollary 2 (Emptiness Problem). *The emptiness problem for automata is solvable in linear time, and more precisely $O(|S| + |E_A|)$.*

2 Representation using DFA

The discussion about DFAs was motivated by our quest for a good representation of SAT questions. There is an equivalence between SAT and DFAs: each string accepted by the DFA corresponds to a truth assignment to the corresponding SAT formula. We also showed that finding such strings is a simple graph search algorithm and takes PTIME. Does this mean that now we can solve SAT in PTIME?

Before we can answer this, notice that we still haven't discussed how we can convert a SAT formula into a DFA. If we find a way to do this in PTIME, then we will be able to solve NPC problems in PTIME.

The basic case of the construction involves only the propositions. This is the case when there is only one atomic proposition, i.e., $\varphi = p_i$. We can construct the automata A_{p_i} as follows:

Definition 3. *The following DFA accepts strings of length n for which the i-th bit is 1:*

$$\begin{aligned} \Sigma &: \{0,1\} \\ S &: \{0,1,\ldots,n,n+1\} \\ s_0 &: \mathit{0} \\ F &: \{n\} \end{aligned}$$

and ρ is defined as:

$$\rho(j,a) = \begin{cases} j+1, & j \leq n, j \neq i-1 \\ i, & j = i-1 \text{ and } a = 1 \\ n+1, & \text{otherwise} \end{cases}$$

Intuitively, we construct the base case by making a transition from state j to state $j+1$ regardless of the input letter, except for the transition between state $i-1$ and i. The latter is only allowed if the input is 1, otherwise the transition goes to the "sink" state $n+1$.

Conjunction and disjunction are a bit messier. They require building $|S_1 \times S_2|$ states to represent all possible combinations of states that the two automata could be in at any point. This is because when the first machine is in s_1 the second could be in any state and this must be represented by the machine. Note that some set of states may never be reached, but the transition function handles this.

2.1 Conjunction

Consider formulas φ_1 and φ_2 represented by automata A_1 and A_2, respectively. Assume $A_1 = (\Sigma, S^1, s_0^1, \rho^1, F^1)$ and $A_2 = (\Sigma, S^2, s_0^2, \rho^2, F^2)$. We would like to build a new automaton $A_1 \cap A_2$ such that $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$.

The construction is straightforward: $A_1 \cap A_2 = (\Sigma, S^1 \times S^2, (s_0^1, s_0^2), \rho, F^1 \times F^2)$, and let $\rho((s,t),a) = (\rho^1(s,a), \rho^2(t,a))$. The time complexity is quadratic. The diligent student will prove the correctness of this construction as an exercise.

2.2 Disjunction

Given two DFAs A_1 and A_2, it is easy to construct an NFA that accepts the language $L(A_1) \cup L(A_2)$. Such a construction takes polynomial time. However, we would then need to determinize the NFA and that can lead to an exponential blow-up. This can be avoided by using a clever algorithm which exploits the fact that the two automata are deterministic and have the same order of variables. We will not present the construction here, but it is important to note that this can be accomplished in PTIME.

2.3 Complement

Finally, we would like to build the automaton whose language corresponds to $\Sigma^* - L(A)$. The construction is quite intuitive: $\overline{A} = (\Sigma, S, s_0, \rho, S - F)$. The complexity here is linear in the number of states.

Thus, we have established a good representation that has easy SAT test, base case, and inductive case. In the next section we will see whether it is useful in solving the satisfiability problem.

3 Using Automata for SAT

We wish to create a function $MAKE - DFA() : \varphi \rightarrowtail A_\varphi$, such that $L(A_\varphi) = models(\varphi) \subseteq 0, 1^{|AP(\varphi)|}$.

MAKE-DFA(φ) /* $\varphi \in$ Form, $n = |AP(\varphi)|$ */
φ *is* p_i *return* Ap_i
φ *is* $(\neg\theta)$ *return* $\overline{A}_\theta$

φ *is* $(\theta \circ \psi)$ *return* $\circ(DFA(\theta), DFA(\psi))$

We can now use MAKE-DFA to solve SAT problems. Given a formula φ, we build the corresponding automaton and check if it is empty.

Lemma 3 (SAT via DFA). *Let φ be a formula. Then φ is satisfiable iff $L(\mathit{MAKE\text{-}DFA}(\varphi)) \neq \emptyset$. We call this algorithm* DFA-SAT.

Every operation on a DFA can be implemented polynomially, so it might appear that DFA-SAT runs in polynomial time. To see if this is indeed the case, let's try to build a DFA using our algorithm on the following problem: We are given two binary numbers $p_1 \ldots p_n q_1 \ldots q_n$ and we want to check whether $\overline{p} = \overline{q}$. In propositional logic, we can express the above problem as $\varphi = \bigwedge_{i=1}^{n}(p_i \leftrightarrow q_i)$.

Therefore, using our DFA construction algorithm we will end up with $\bigcap_{i=1}^{n}(A_{p_i} \leftrightarrow A_{q_i})$. Note that since $|A_{p_i}| = |A_{q_i}| = O(n)$, $A_{p_i} \leftrightarrow A_{q_i}$ is of size $O(n^2)$. Therefore, $|\bigcap_{i=1}^{n}(A_{p_i} \leftrightarrow A_{q_i})| = O((n^2)^n) = O(n^{2n}) = O(n^n)$. This suggests that using the automata-theoretic approach is worse than using the splitting method.

We can, however, improve our algorithm to construct the smallest possible DFAs making the search process, post conversion, even faster. We rely on canonicity, a concept discussed at length in Comp 481:

Lemma 4 (Canonical DFA). *Given a regular language $L \subseteq \Sigma^*$ there is a unique (up to isomorphism) DFA such that $L(A) = L$.*

Moreover, canonical DFAs are minimal. That is, there is a minimization algorithm (invented by Hopcroft) such that given a DFA A it finds a minimal DFA B such that $L(A) = L(B)$. The time complexity of Hopcroft's algorithm is $O(n \log(n))$.

To improve our DFA construction algorithm we will add the minimization rule after every case. This will guarantee that that the running time will be $2^{O(n)}$.

Lemma 5 (Complexity of SAT via DFA). *Solving SAT using DFAs requires $EXPTIME$ and $EXPSPACE$.*

Another name for 0/1 decision DFAs is BDDs (Binary Decision Diagrams). Note that the performance of BDDs depends on the order of the input. Considering the above example (checking wither $\overline{p} = \overline{q}$), it would be much easier if the bits were interleaved; that is, if the input were of the form $p_1 q_1 \ldots p_n q_n$. In this case, the size of the automaton would be $O(n)$.

Therefore, we conclude that to be efficient, we need the right ordering. However, finding the best ordering is NP-hard. To this purpose, current methods rely heavily on heuristics. First we need a good *initial order*. Second, we can use *Dynamic Reordering*. The idea behind Dynamic Reordering is that we start with a good initial order. If this order blows up the size of the BDD, we try to improve it.

BDDs are not only used for satisfiability. Given $\varphi, \psi \in$ Form we may want to prove $\models (\varphi \leftrightarrow \psi)$. This problem may be of interest to us since we may want to find out if two given circuits φ and ψ compute the same boolean function.

$DFA{-}SAT(\varphi)$ is useful in comparing for logical equivalence between circuits or formulas: $\varphi \models\!\!=\!\!| \psi$ iff $L(DFA(\varphi)) = L(DFA(\psi))$. Further, if in $DFA(\varphi)$ we minimize after every step, we can say that $\varphi \models\!\!=\!\!| \psi$ iff $DFA(\varphi) = DFA(\psi)$. We make sure that we avoid isomorphism when we construct A_φ and A_ψ since checking for isomorphism is NP-hard.

We saw in the previous section that automata may be used to solve SAT problems. The input to an automata for φ, call it A_φ, would be from the alphabet $0, 1$ and would be of the form $p_0, p_1, p_2, ...p_n$ where $p_i \in$ AP(φ). The automata to check if $p_i = 1$ consists of $n + 2$ states. Moreover, we may inductively define DFA's for negation (using DFA complementation), disjunction (using DFA union), and conjunction (using DFA intersection). Using a DFA representation for SAT is useful because we can quickly check to see if a formula φ is satisfiable by checking to see if $L(A_\varphi \neq \emptyset$. However, DFA intersection and (sometimes union) is still exponential in time and space in the worst case. So, we've seen that SAT is NP-complete and therefore there's no easy way to solve it. We've seen a representation that is good for testing satisfiability, but that is still exponential the worst case.

4 Living with NP-Completeness[1]

How does one cope with having an NP-Complete problem?

1. Find practical algorithms for the problem.
2. Find how simple you can make the problem and have it remain NP-Complete, i.e. explore the boundary between tractability and intractability.

To explore the boundary, we need to have a good definition of what a boundary is. Let's see this for the satisfiability problem:

SAT: NP-complete

CNF-SAT: NP-complete

$k{-}CNF$-SAT: depends on the value of k, where a k-clause is a disjunction of at most k literals.

[1]This is bonus material that we did not discuss in class in 2005.

Let's investigate $k-CNF$-SAT for different values of k:

$3-CNF$:

$$\bigwedge_{i=1}^{m} c_i,$$

where $c_i = \pm p_{i_1} \pm p_{i_2} \pm p_{i_3}$
$3-CNF$ is NP-Complete, but $1-CNF$ and $2-CNF$ are both in PTIME.

Lemma 6. *A $1-CNF$ formula is satisfiable iff it does not contain conflicting literals. (conflicting literals: $p \wedge \neg p$)*

It is less obvious to see why $2-CNF$ is in PTIME. We will see why 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.

Another interesting idea is to count the polarity of clauses. We define a k^+-clause as having at most k positive literals. 2^+-CNF is NP-Complete, while 0^+-CNF and 1^+-CNF are both in PTIME. In fact, 1^+-CNF are the Horn formulas.

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

5 Revisiting the Splitting Method

DFA-SAT is worse than Splitting in the worst case. They are orthogonal methods, in that the situation producing the best running time in one produces the worst in the other. The intuition is that having lots of unit clauses is good for DPLL, but is bad for DFA-SAT because we have to take the intersection of a larger number of clauses. Unit clauses are discussed more below.
First, recall the splitting algorithm:

$SAT(\varphi)$
  if φ is easily satisfiable, return 1

if φ is easily not satisfiable, return 0
choose $p \in AP(\varphi)$ and $c \in 0, 1$
if $SAT(\varphi[p \mapsto c])$ return true
else return $SAT(\varphi[p \mapsto 1 - c])$

We now introduce some nomenclature.

- *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 ignore $p \mapsto 0$). If φ has a unit clause $\neg p$, we choose p and assign it 0.

Using the splitting method with unit preference gives us the DLL(*Davis-Longemann-Loveland*) algorithm.

$SAT(\varphi)$
if φ is easily satisfiable, return 1
if φ is easily not satisfiable, return 0
choose $p \in AP(\varphi)$ and $c \in 0, 1$ [***Unit Preference***]
if $SAT(\varphi[p \mapsto c])$ return true
else return $SAT(\varphi[p \mapsto 1 - c])$