COM S 6810 Theory of Computing February 5, 2009

Lecture 6: PSPACE Completeness and Circuits

Instructor: Rafael Pass *Scribe: Hu Fu*

1 PSPACE Completeness

Recall that we introduced the polynomial hierarchy (PH) in the last lecture. A language is in Σ_i if $x \in L \Leftrightarrow \exists y_1 \forall y_2 \cdots Q_i y_i, R(x, y_1, y_2, \cdots y_i)$, for some poly-time (in terms of $|x|$) relation R, where Q_i is $\exists/\forall$ depending on whether i is even or odd. Related to Σ_i, we also defined

$$\Pi_i = \text{co-}\Sigma_i, \quad PH = \cup_{i\geq 1}\Sigma_i = \cup_{i\geq 1}\Pi_i.$$

We showed that Σ_i-SAT is Σ_i-complete, and proved two theorems on PH:

Theorem 1 $\Sigma_i = \mathsf{NP}^{\Sigma_{i-1}}$.

Theorem 2 *If* $\Sigma_i = \Pi_i$*, then* $\mathsf{PH} = \Sigma_i$.

We also defined the language TBQF:

$$x \in \text{TBQF} \Leftrightarrow \exists y_1 \forall y_2, \cdots Q_n y_n,\ \varphi(x, y_1, y_2, \cdots y_n),$$

where n is the input length, φ is a Boolean formula, and Q_i is $\exists/\forall$ depending on whether n is odd or even. Note the difference between TBQF and Σ_i-SAT: in the definition of Σ_i-SAT the number i of alternating quantifiers is fixed, whereas in TBQF, the number of alternating variables goes up with the length of the input.

Now we show that TBQF is PSPACE-complete, where completeness is defined in terms of polynomial time reduction. This result further illustrates the relation between PH and PSPACE (recall that PH $\subseteq$ PSPACE, where the inclusion is proper unless PH collapses).

Theorem 3 *TBQF is PSPACE-complete.*

Proof. It is easy to see that TBQF is in PSPACE: by trying out all possible choices of $y_1, y_2, \cdots, y_n$ and reusing the space, we can decide an instance within linear space.

The proof that any problem in PSPACE can be reduced to TBQF is similar to the proof of Savitch's theorem. Given any language $L \in$ PSPACE, it is decided by a Turing machine M that uses polynomial space: $\forall x \in \{0,1\}^*$, $x \in L \Leftrightarrow M(x) = 1$.

Recall that we defined the configuration graph for Turing machines. For a machine M that uses at most $p(n)$ space, where $p(n)$ is a polynomial of n, the number of nodes in the configuration graph has $2^{O(p(n))}$ nodes. Node i is connected to node j by an arch from i to j if, according to the transition function, M goes from the configuration corresponding to i to the one corresponding to j. The problem thus boils down to deciding if there exists a path from an initial configuration to the "accepting" configuration. If we can reduce this problem in polynomial time to an equivalent TBQF instance, then we have shown TBQF is PSPACE-complete.

For two nodes c_i and c_j in the configuration graph, we define $\delta_n(c_i, c_j)$ to be one if one can reach c_j from c_i within 2^n steps, and zero otherwise. For our purpose, it is sufficient to represent δ_n as a TBQF instance in polynomial time.

Following the proof for Savitch's theorem, it is tempting the write the recursion as $\delta_n(c_i, c_j) = 1 \Leftrightarrow \exists c', \delta_{n-1}(c_i, c') \wedge \delta_{n-1}(c', c_j)$. However, since each step calls for two incursions of δ of lower order, it is clear that continuing with this recursion down to the base case would incur exponentially many quantifiers, while we need a polynomial time reduction.

Now "$\forall$" quantifier comes to rescue. If we write the recursion as

$$\delta_n(c_i, c_j) = 1 \quad \Leftrightarrow \quad \exists c' \forall D_1, D_2, (D_1 = c_i \wedge D_2 = c') \vee (D_1 = c' \wedge D_2 = c_j) \Rightarrow \delta_{n-1}(D_1, D_2),$$

then the reduction takes only polynomial time, and hence we have shown that TBQF is indeed PSPACE-complete. (Note that each "$=$" and "$\Leftrightarrow$)" can be represented by polynomial number of AND, OR and NOT gates.

2 Circuits

We start a new topic by defining the notion of circuit.

Definition 1 *A* circuit *is a directed acyclic graph where nodes are* gates *and edges are wires; the* input gates *have in-degree zero, and the* output gates *have out-degree zero. Each input gate is labeled with a value from* $\{0, 1\}$*, and each non-input gate is labeled with an operation from* $\{AND, OR, NOT\}$ *(in this case the circuit is called a* Boolean circuit*), or* $\{\{AND\}_{n \in \mathbb{N}}, \{OR\}_{n \in \mathbb{N}}, NOT\}$.

We will see later that the latter set of operations does not differ much from the former one. In general, the fan-outs in the circuits can be unbounded. If each gate in the circuit has fan-out one, then we call the circuit a *formula.*

The *size* $|C|$ of a circuit C is defined to be the number of non-input gates in C, and the depth of C is the length of the longest path in C. Each circuit C represents a function f_C, which is a function that, when given the inputs in C, computes the same outputs as C.

Proposition 1 *$\{NOT, AND, OR\}$ is a universal basis, that is, every function $f : \{0,1\}^n \longrightarrow \{0,1\}$ can be computed by a boolean circuit.*

Proof. Let $X = \{x \in \{0,1\}^n, f(x) = 1\}$, then $f(x) = \underset{x' \in X}{OR}\ (x = x')$. The expression $x = x'$ can be expressed as a boolean formula as $\underset{i\in[n]}{AND}(x_i \wedge x'_i) \vee (\neg x_i \wedge \neg x'_i)$, where x_i and x'_i denote the i-th coordinate of x and x', respectively. Now we have expressed $f(x)$ with a circuit; what remains to show is that we can further represent the OR and AND having multiple fin-ins with simple boolean gates OR and AND. This, however, is simple: we can evaluate a "big" OR with a binary tree, where the inputs are the leaves of the tree, and each internal node is an "OR" gate with fan-in two. Obviously the output at the root of the tree is the same as that of the "big OR" gate. The same argument applies for the AND gate.

Remark: 1. Note that the restriction on the dimension of the output of f is without loss of generality. For higher dimensions, we can simply compute the outputs bit by bit.

2. When representing a "big OR (AND)" gate that has n fan-ins with $OR(AND)$ gates that have two fan-ins, we introduced at most n more gates.

Computing the functions more cleverly, we can do the same thing using boolean circuits of smaller sizes:

Theorem 4 *Any function $f : \{0,1\}^n \longrightarrow \{0,1\}$ can be computed by a boolean circuit of size $O(2^n)$ and depth $O(n)$.*

Proof. Making use of the fact that

$$f(x_1, \cdots x_n) = (x_1 \wedge f(1, x_2, \cdots, x_n) \vee (\neg x_1 \wedge f(0, x_2, \cdots, x_n)),$$

we can wire the circuit recursively to compute f. To compute the size of the circuit, we have $c_n = O(1) + 2c_{n-1}$, where c_n denotes the size of the circuit needed to compute a function with n arguments. This gives $c_n = O(2^n)$. Moreover, each recursion incurs only $O(1)$ depth; since the two subproblems can be computed in a parallel fashion, the depth of the whole circuit is within $O(n)$.

Here is another theorem that further illustrates the relation between boolean circuits and boolean functions:

Theorem 5 *Almost all boolean functions on n variables require circuits of sizes $O(\frac{2^n}{n})$.*

Proof. Observing that each distinct circuit can compute only one function, we will apply the pigeon-hole principle to show this fact. The number of all possible boolean functions defined on $\{0,1\}^n$ is 2^{2^n}. Now we count the number of possible boolean circuits of size s: if we specify for each gate the type of boolean operation it does and where its two input wires connect to, then the whole circuit is determined. This gives a count of the number of circuits $O(s^{2s})$. Now when $s = \frac{2^n}{4n}$, the number of circuits is bounded by

$$O\left(\left[\frac{2^n}{4n}\right]^{\frac{2^n}{4n}\cdot 2}\right) \ll 2^{2^{n-1}} \ll 2^{2^n},$$

therefore most boolean functions on $\{0,1\}^n$ cannot be computed by circuits of sizes $O(\frac{2^n}{n})$.

Definition 2 *Let $T : N \longrightarrow N$ be a function. A $T(n)$-sized* circuit family *is a sequence $\{c_n\}_{n\in\mathbb{N}}$ of boolean circuits, where c_n has n inputs and one output such that $|c_n| \leq T(n)$.*

Definition 3 *We write* $\mathrm{size}(T(n))$ *as the class of languages that can be decided by $O(T(n))$ sized circuits. Furthermore,*

$$\mathsf{P/poly} \triangleq \cup_{c\geq 1}\mathrm{size}(n^c).$$

The following theorem is easily seen by combining the previous two theorems:

Theorem 6 $\mathrm{size}\ (s^2) \subsetneq \mathrm{size}(s)$.

Proof. For a boolean function f, define a language f such that $x \in L$ iff $f(x) = 1$. Then using cs^2 space (for a constant c), all functions on inputs $\{0,1\}^{c' \log s}$ can be computed (and hence the corresponding languages), while circuits of size $O(\frac{s^2}{\log s})$ cannot compute most of such functions. The theorem is hence implied.