Lecture 18:
PSPACE-completeness, and Randomized Complexity Classes

Valentine Kabanets

November 22, 2016

1 PSPACE-completeness

Recall the NP-complete problem SAT: Is a given Boolean formula $\phi(x_1, \ldots, x_n)$ satisfiable? The same question can be stated equivalently as: Is the formula $\exists x_1 \ldots \exists x_n \phi(x_1, \ldots, x_n)$ true? Also the coNP-complete problem TAUT: Is a given Boolean formula $\psi(x_1, \ldots, x_n)$ a tautology (i.e., true on all assignments)? (*Exercise:* Show that TAUT is indeed coNP-complete.) This can be stated as: Is the formula $\forall x_1 \ldots \forall x_n \psi(x_1, \ldots, x_n)$ true?

What if we start alternating the quantifiers? Let us define the language TQBF (True Quantified Boolean Formulas) as the set of true formulas of the form

$$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \phi(x_1, x_2, \ldots, x_n),$$

where $Q_i \in \{\exists.\forall\}$ is a quantifier, ϕ is a Boolean formula in CNF (conjunctive normal form), and the sequence of quantifiers $Q_1 \ldots Q_n$ alternates between $\exists$ and $\forall$. For example, all odd Q_{2k+1} can be $\exists$, and all even Q_{2k} can be $\forall$.

We can assume, without loss of generality, that $Q_1 = \exists$: if not, then we can always add an $\exists y$ before the formula for some variable y that does not occur in the formula. In this case, one can think of a QBF as a *game* between two players: Player $\exists$ and Player $\forall$. First, Player $\exists$ sets the value of x_1. Then Player $\forall$ sets the value of x_2. Then Player $\exists$ sets the value of x_3, and so on. After n moves, the winner is declared using the following criterion: Player $\exists$ wins iff the assignments to $x_1 \ldots x_n$ constructed during the game is *satisfying* for the formula ϕ. It is easy to see that a given QBF is true iff Player $\exists$ has a *winning strategy*, i.e., for any choice of moves by Player $\forall$, Player $\exists$ can play so as to guarantee its win.

How can we decide if a given QBF is true? The following simple recursive algorithm $Truth$ does the job.

Algo $Truth(\Phi)$
if Φ is quantifier-free **then** return its value
end if
Let $\Psi = Q_1 x_1 \ldots Q_n x_n \phi(x_1, \ldots, x_n)$.
$b_0 = Truth(Q_2 x_2 \ldots x_n \phi(0, x_2, \ldots, x_n))$;
$b_1 = Truth(Q_2 x_2 \ldots x_n \phi(1, x_2, \ldots, x_n))$;% re-using space
if $Q_1 = \exists$ **then** return $b_0 \vee b_1$
  else return $b_0 \wedge b_1$

end if
end Algo

The algorithm $Truth$ runs in polynomial space: the depth of the recursion is n, and the size of each stack record is $poly(n)$.

Remark: Using some tricks, one can show that TQBF is in $SPACE(n)$ (i.e., linear space).

It turns out that PSPACE is probably the best possible we can do for TQBF, as it is PSPACE-complete.

Theorem 1. *TQBF is* PSPACE*-complete.*

Proof. We need to show (1) $TQBF \in$ PSPACE: we already showed that above.

(2) Every language in PSPACE is polytime reducible to TQBF. We'll show this next.

We need to reduce every language in PSPACE to TQBF. Let L be any language in PSPACE. Say L is decided by some TM M in $Space(n^c)$ for some constant c. To decide if $x \in L$, we consider the configuration graph of M on x. Each configuration will be of size $O(n^c)$; let us denote this size by m. There at most 2^m different configurations. So, $x \in L$ iff there is a path from the start configuration to the accept configuration of length at most 2^m. We will construct a QBF that will express the existence of such a path.

In a sense, our proof is a restatement of the proof of Savitch's Theorem. Recall the function $Path(a, b, i)$ that we used in the proof of Savitch's Theorem: $Path(a, b, i)$ is True iff there is a path from node a to node b of length at most 2^i. We will define a sequence of QBFs ψ_i, $i = 0, \ldots, m$, where $\psi_i(A, B)$ is True iff the variables A and B encode two configurations a and b such that b is reachable from a in at most 2^i steps; note that A and B are groups of m variables each. The required QBF will then be $\psi_m(Start, Accept)$, where $Start$ and $Accept$ are the binary strings encoding the start configuration and accept configuration, respectively.

For $i = 0$, $\psi_0(X, Y)$ is a quantified Boolean formula on free variables X and Y that is True iff either $X = Y$ or Y is reachable from X in 1 step. The formula $\psi_0(X, Y)$ can be written as a DNF of size $poly(m)$. (*Exercise:* Explain why.)

Given $\psi_i(X, Y)$, we can try to define $\psi_{i+1}(X, Y) = \exists Z[\psi_i(X, Z) \wedge \psi_i(Z, Y)]$. However, this is a *bad* idea: the size of ψ_{i+1} doubles, and so, the size of ψ_m would be exponential. The trick is to "re-use" the formula ψ_i. Here is a correct definition of $\psi_{i+1}(X, Y)$:

$$\exists Z_{i+1} \forall X_{i+1} \forall Y_{i+1}[((X_{i+1} = X \wedge Y_{i+1} = Z_{i+1}) \vee (X_{i+1} = Z_{i+1} \wedge Y_{i+1} = Y)) \Rightarrow \psi_i(X_{i+1}, Y_{i+1})]$$

Note how the formula above is True iff there is a Z_{i+1} such that both $\psi_i(X, Z_{i+1})$ and $\psi_i(Z_{i+1}, Y)$. Also, the size of the new formula ψ_{i+1} is that of ψ_i plus a polynomial term. So, our final formula ψ_m is of polynomial size as required. It's not hard to see that all ψ_i can be constructed in polytime. □

A few other two-person games are (e.g., Go, checkers, and chess, when generalized to $n \times n$ size boards to allow asymptotics) are also known to be PSPACE-complete.

2 Nondeterministic Log Space

This entire section is optional.

2.1 NL-completeness

To talk about NL-completeness, we need to refine our notion of a reduction. We'll talk about *logspace*-computable reductions (recall our definition of a space-bounded TM that also has a special write-only output tape).

A language A is NL-*complete* if

1. $A \in \mathsf{NL}$, and
2. every language $B \in \mathsf{NL}$ is logspace-reducible to A.

Define the language $st-CONN$ as the collection of triples $\langle G, s, t\rangle$ where G is a directed graph with a path from s to t.

Theorem 2. *$st-CONN$ is NL-complete.*

Proof. (1) $st-CONN \in \mathsf{NL}$. We have a nondeterministic algorithm that tries to guess a path from s to t, storing only the current node in its memory. If a graph has n nodes, that gives us a nondeterministic $\log n$-space algorithm.

(2) Every problem in NL is logspace-reducible to $st-CONN$. Fix a NL machine M and an input w. Given M and w, we can construct a graph of configurations of M on w. Two configurations are connected by an edge iff one yields the other one in one computation step of M.

Here by a configuration we mean the following info: contents of the work-tape, the position of the tape head on the input tape, the state of the TM.

Note that for a logspace NTM, the amount of info in the configuration is $O(\log n)$. It's also not hard to see that this configuration graph can be output by a logspace TM. (The size of the output can be bigger than $\log n$, but the size of the worktape is only $O(\log n)$.) □

Finally, we observe the following:

Theorem 3. *If A is logspace reducible to B, and if B is in L, then A is also in L.*

The proof of this is not trivial: we need to deal with the fact that the output of the logspace-computable reduction f can be of polynomial size. However, the trick is that we can always *re-compute* the required position in the string $f(x)$. (See the text for more details.)

Finally, observe that $\mathsf{NL} \subseteq \mathsf{P}$. This is because $st-CONN$ is in P (by a BFS algorithm), and every language in NL is logspace (and hence polytime) reducible to $st-CONN$.

2.2 NL = coNL

Next we turn to another amazing result in complexity which proves the closure of NL under complementation. (This is like $NP = coNP$ for space-bounded machines!) The question whether nondeterministic space is closed under complementation was open for 23 years; it was first stated in 1964 by Kuroda in relation to the class of context sensitive languages, which Kuroda proved to be exactly the class $NSpace(n)$. In 1987, Neil Immerman, an American researcher, and Robert Szelepcsényi, a Slovakian undergraduate student, independently proved that $\mathsf{NSPACE}(s(n)) = \mathsf{coNSPACE}(s(n))$, for any proper complexity function $s(n) \geq \log n$. This was a big shock to the CS community for two reasons: (1) it was widely believed that $\mathsf{NL} \neq \mathsf{coNL}$, and (2) the proofs by Immerman and Szelepcsényi were quite simple.

Since $ST - CON$ is NL-complete, in order to prove NL = coNL, it suffices to prove that $ST - CON \in$ coNL, i.e., that it can be checked in nondeterministic logspace whether t is *not* reachable from s.

Theorem 4 (Immerman-Szelepcsényi)**.** $ST - CON \in$ coNL

Proof. Idea: To check if t is not reachable from s, enumerate all nodes that *are* reachable from s and check that t is *not* among them.

This sounds too easy. The trick is to do this enumeration of *all* nodes in logspace, and ensuring that indeed *all* nodes reachable from s were enumerated. We need some clever idea to do this. The clever idea is to *count*.

Let us imagine for a moment that we are given a number $N = \#$ of nodes reachable from node s. (Later we'll show how to compute this N in NL.) The following NL algorithm checks if t is *not* reachable from s in a given directed graph $G = (V, E)$, where $|V| = n$.

Algo $Unreach(G, s, t)$
% given $N = \#$ nodes reachable from s
    $count = 0$;
    for every node v
        "make a nondeterministic guess whether v is reachable from s"
        if guess is Yes **then**
            "nondeterministically try to guess a path from s to v of length at most n";
            if "guessed path does not lead to v" **then** Reject **end if**
            if $v = t$ **then** Reject
            else $count = count + 1$;
            end if
        end if
    end for
    if $count < N$ **then** Reject
    else Accept % if $count = N$
    end if
end Algo

Clearly the algorithm $Unreach$ runs in nondeterministic logspace; observe that N and *count* can be at most n, and so they can be written as binary numbers of length at most $\log n$.

Claim 1. *Algorithm $Unreach(G, s, t)$ has an accepting computation iff t is* not *reachable from s.*

Proof of Claim. The algorithm makes sure that it enumerates all nodes reachable from s, by comparing *count* with N. The algorithm accepts iff node t was not one of these N nodes reachable from s. □

To compute $N = \#$ nodes reachable from s, we will iteratively compute (re-using space) the values $R(i) = \#$ nodes reachable from s in at most i steps. Then we obtain $N = R(n)$.

Algo $\#Reach(G, s, t)$
    $R(0) = 1$ % s is reachable from s in 0 steps
    for $i = 1..n$

```
        R(i) = 0 % initialize R(i)
        for every node v
            % try all nodes u reachable from s in ≤ (i − 1) steps, and
            % check if v is reachable in ≤ 1 steps from any such u
            count = 0;
            for every node u
                "make nondeterministic guess whether u is reachable from s in ≤ (i − 1) steps";
                if "guess is Yes" then
                  "nondeterministically try to guess a path from s to u of length ≤ (i − 1)";
                  if "guessed path does not lead to u" then Reject end if
                  count = count + 1; % if u is reachable, count it in
                  if u = v OR (u, v) ∈ E
                    then R(i) = R(i) + 1;
                         break; % go to next iteration of "for v" loop
                  end if
                end if
            end for
            if count < R(i − 1) then Reject
            end if
        end for
    end for
    return R(n);
end Algo
```

Remark: The algorithm *#Reach* needs to remember only two successive values $R(i)$ and $R(i+1)$ at any point in time. So, it re-uses space when computing $R(1), \ldots, R(n)$. Thus, the algorithm can be made to run in NL.

Claim 2. *Algorithm #Reach computes the number of nodes reachable from s.*

Proof of Claim. The proof is by induction on i. For $i = 0$, $R(0) = 1$ is obviously correct.

For the induction step, assume that $R(i)$ is equal to the number of nodes reachable from s in at most i steps. We need to prove that $R(i+1)$ is equal to the number of nodes reachable from s in at most $(i+1)$ steps. To prove this, notice that the algorithm increments $R(i+1)$ on a node v iff v is reachable from s in at most $(i+1)$ steps. This is because $R(i+1)$ is *not* incremented only if all nodes at distance $\leq i$ from s were tried, and v is *not* reachable in ≤ 1 steps from any one of them. □

Thus, to check if t is not reachable from s, we first run the algorithm *#Reach* to compute N, then run the algorithm *Unreach* with that N. The total space this nondeterministic procedure takes is $O(\log n)$, because each of the two algorithms is logspace. □

3 Randomized complexity classes

A language $L \in \mathsf{RP}$ if there is a deterministic polytime TM $M(x, r)$ such that

1. for all $x \in L$, $\mathbf{Pr}_r[M(x, r) \text{ accepts}] \geq 1/2$, and

2. for all $x \notin L$, $\mathbf{Pr}_r[M(x,r) \text{ accepts}] = 0$.

Here, we assume some polynomial bound on $|r|$ in terms of $|x|$, i.e., for some constant c, we have $|r| \leq |x|^c$.

Now, to decide L, on input x, a randomized algorithm may first flip $|r|$ random coins to compute a random string r, and then simulate $M(x,r)$. This randomized algorithm will run in polytime, and will be correct for all $x \notin L$ with probability 1, and will be correct for all $x \in L$ with probability at least $1/2$.

Note that if $L \in \mathsf{RP}$, then $L \in \mathsf{NP}$. So, we get that $\mathsf{RP} \subseteq \mathsf{NP}$. The class RP contains those languages that can be decided probabilistically with *one-sided* error: an algorithm may err on positive instances, but never on negative instances. Next, we define the class of languages decidable with two-sided error.

A language $L \in \mathsf{BPP}$ if there is a polytime DTM $M(x,r)$ such that

1. for all $x \in L$, $\mathbf{Pr}_r[M(x,r) \text{ accepts}] \geq 3/4$, and
2. for all $x \notin L$, $\mathbf{Pr}_r[M(x,r) \text{ accepts}] \leq 1/4$.

Note that now we allow a "small" probability of error even on inputs not in the language.

4 Error reduction

For one-sided error algorithms, we have that they are always correct on no-instances, and may be wrong with probability at most $1/2$ on yes-instances. For two-sided error algorithms, we have that they may wrong on both yes- and no-instances, but only with probability at most $1/4$.

To decrease the error probability of our one-sided error algorithm on a given input x, we run the algorithm k times, independently, and accept iff at least one run was accepting. Observe that the error probability of this new algorithm is at most 2^{-k}.

For a two-sided algorithm, to decrease the error probability, we also run the algorithm k times, and then output the *majority answer*. Observe that in k runs, we expect to see at least $(3/4)k$ correct answers. By the Law of Large Numbers (its quantitative version), the probability the actual number of correct answers will be much smaller, say below $k/2$, is exponentially small in k (is $2^{-\Omega(k)}$).

The upshot is: by paying a polynomial-factor overhead, we can reduce the error probability of our randomized polytime algorithm on inputs of size n to below 2^{-n^c}, for any constant $c > 0$.

5 Known containments

It's obvious that $\mathsf{P} \subseteq \mathsf{RP} \subseteq \mathsf{NP}$. It's also clear that $\mathsf{P} \subseteq \mathsf{BPP}$. With error reduction, it's easy to argue also that $\mathsf{RP} \subseteq \mathsf{BPP}$. It is an open question whether BPP is contained in NP.

A popular conjecture is: $\mathsf{BPP} = \mathsf{P}$. That is, it is believed that every efficient randomized algorithm has an equivalent efficient deterministic algorithm. The proof of this conjecture, however, seems to be beyond our reach at the moment.

6 Polynomial Identity Testing

One of the early problems for which a probabilistic polytime algorithm was developed is Primality Testing: Given a number n in binary, decide if n is prime. (Even though in 2002, a deterministic polytime algorithm was discovered for Primality Testing, the old randomized algorithms will continue to be used in practice since they are significantly faster.)

Instead of Primality Testing, we'll consider another important problem for which no deterministic polytime algorithm is known yet. It's *Polynomial Identity Testing:* Given two polynomials p_1 and p_2 in variables $x_1, \ldots, x_n$, decide if p_1 is identically equal to p_2. (We assume that the polynomials have integer coefficients.)

If the two polynomial are given explicitly as a sum of monomials, then the identity testing problem is trivial: we just have to check that the corresponding coefficients by same monomials are equal. The interesting case of the problem is when the polynomials p_1 and p_2 are given implicitly, e.g., as determinants of symbolic matrices.

Consider the case of single-variable polynomials: $p_1(x)$ and $p_2(x)$. Define $h(x) = p_1(x) - p_2(x)$. We have $p_1(x) = p_2(x)$ iff $h(x) \equiv 0$.

To check if $h(x) \equiv 0$, we can take a random integer r from a set $S = \{1, 2, 3, \ldots, N\}$, and evaluate $h(r)$. If $h(r) \neq 0$, we output "Nonzero"; if $h(r) = 0$, we output "Probably zero".

Now, suppose that the degree of h is at most d. Suppose also that $N = 10 \cdot d$. Then, if $h(x) \equiv 0$, our algorithm above will be always correct. If $h(x) \neq 0$, then the probability of error is at most $d/N = 1/10$, since $h(x)$ may have at most d roots, and the probability of randomly picking a number that happens to be one of these d roots is at most d/N.

The interesting thing is that the same kind of argument is also applicable to the case of *multivariate* polynomial of total degree at most d. Namely, suppose that we have a nonzero polynomial $p(x_1, ..., x_n)$ of total degree at most d (i.e., the highest degree monomial of p is of degree at most d). Consider the set of integers $S = \{1, 2, 3, \ldots, N\}$, for $N = 10 \cdot d$. Our randomized algorithm will be:

> Pick uniformly at random $r_1, \ldots, r_n \in S$, and evaluate $p(r_1, \ldots, r_n)$. If the result is non-zero, then output "Non-Zero"; otherwise, output "Zero".

Note that if $p \equiv 0$, then our algorithm will always say "Zero", which is correct! On the other hand, if $p \not\equiv 0$, then it's possible for the algorithm to say "Zero" incorrectly, when it happens to choose $r_1, \ldots, r_n \in S$ such that $p(r_1, \ldots, r_n) = 0$. Fortunately, the probability of such an error is small, as we show next.

This is known in Computer Science under the name of

Theorem 5 (Schwartz-Zippel Lemma)**.** *Let $p(x_1, ..., x_n)$ be any non-zero polynomial of total degree at most d. Let S be any finite set (of integers). Then*

$$\mathbf{Pr}[p(r_1, ..., r_n) = 0] \leq d/|S|,$$

where the probability is over the independently randomly chosen values $r_1 \in S$, $r_2 \in S$,..., $r_n \in S$.

We'll give the proof next time. Note that in our case, $|S| = 10 \cdot d$, and so the error probability of our algorithm is at most $d/(10d) = 1/10$. We can reduce the error even further as we discussed above (by repetition), or by making the size of our set S larger.