CSE491/596 Lecture Wed. 11/08: PSPACE and Oracle Turing Machines

First, to recap the PSPACE-completeness of TQBF from a bottom-up rather than top-down view, we begin with the relation

$$\Phi_0(I,J) \equiv (I = J) \vee I \vdash_M J$$

Recall that the latter notation means that ID I can go to ID J by one step of the machine M (which the proof doesn't care whether it is deterministic or nondeterministic). Thus $\Phi_0(I,J)$ means that I can go to J by *at most one step*. The predicate $I = J$ actually stands for the bitwise equivalence of variables $i_1, i_2, \dots, i_s$ standing for the binary encoding of an ID I and $j_1, j_2, \dots, j_s$ standing for J:

$$(i_1 \vee \bar{j}_1) \wedge (\bar{i}_1 \vee j_1) \wedge (i_2 \vee \bar{j}_2) \wedge (\bar{i}_2 \vee j_2) \wedge \cdots \wedge (i_s \vee \bar{j}_s) \wedge (\bar{i}_s \vee j_s).$$

Here s is basically $s(n)$ times $\log_2|\Gamma|$. The predicate $I \vdash_M J$ is coded in much the same way as the conversion from Turing machines to Boolean circuits employed in the Cook-Levin proof; again the size of the formula needed is $O(s) = O(s(n))$.

Now to express that an ID I can go to an ID K in *at most* **2** steps, it would be natural to write

$$\Psi_1(I,K) \equiv (\exists J) : \Phi_0(I,J) \wedge \Phi_0(J,K).$$

(The colon $:$, by the way, means that the quantifier grabs everything to the end of the formula.) Then to express "at most **4** steps" we would recurse:

$$\Psi_2(I,K) = (\exists J) : \Psi_1(I,J) \wedge \Psi_1(J,K).$$

But when we expand this out, it starts getting very bushy:

$$\Psi_2(I,K) = (\exists I', J, K') : \Phi_0(I,I') \wedge \Phi_0(I',J) \wedge \Phi_0(J,K') \wedge \Phi_0(K',K).$$

For $\mathbf{2^r}$ steps, we would have 2^r terms---exponentially many. By using an alternation rather than keep everything existential, we cut down the expansion *syntactically*, even though it initially looks bushier:

$$\Phi_1(I,K) \equiv (\exists J)(\forall I', J') : [(I' = I \wedge J' = J) \vee (I' = J \wedge J' = K)] \longrightarrow \Phi_0(I',J').$$

For the unrolling part it is better to change J' to read K' here:

$$\Phi_1(I,K) \equiv (\exists J)(\forall I', K') : [(I' = I \wedge K' = J) \vee (I' = J \wedge K' = K)] \longrightarrow \Phi_0(I',K').$$

When we expand the implication ($a \longrightarrow b \equiv \neg a \vee b$) and use De Morgan's Laws, the Boolean part does become more complicated than CNF or DNF:

$$\Phi_1(I,K) \equiv (\exists J)(\forall I',K'): [(I' \neq I \vee K' \neq J) \wedge (I' \neq J \vee K' \neq K)] \vee \Phi_0(I',K').$$

The point is what happens when we step up to 4 (and then 8,16,...):

$$\Phi_2(I,K) \equiv (\exists J)(\forall I',K'): [(I' \neq I \vee K' \neq J) \wedge (I' \neq J \vee K' \neq K)] \vee \Phi_1(I',K')$$

$$\equiv \begin{array}{l} (\exists J)(\forall I',K')(\exists J')(\forall I'',K''): \\ \qquad [(I' \neq I \vee K' \neq J) \wedge (I' \neq J \vee K' \neq K)] \vee \\ \qquad [(I'' \neq I' \vee K'' \neq J') \wedge (I'' \neq J' \vee K'' \neq K')] \vee \Phi_0(I'',K''). \end{array}$$

It's weird that the only predicates in the body are inequalities. An analogy for what this is saying is:

IF I' and K' are set to the claimed halfway point and either the begin or end point, THEN (it works)

which is the same thing as

EITHER I' and K' are NOT set up correctly to the halfway point and either endpoint, OR they are set up correctly and (...it works).

There is no need for an extra predicate because there is only one final ID I_f and we initialize K to I_f and I to $I_0(x)$. Maybe we can get a feel from the unrolling for 8 machine moves:

$$\begin{array}{c} \Phi_3(I,K) \equiv (\exists J)(\forall I',K')(\exists J')(\forall I'',K'')(\exists J'')(\forall I''',K'''): \\ [(I' \neq I \vee K' \neq J) \wedge (I' \neq J \vee K' \neq K)] \vee \\ [(I'' \neq I' \vee K'' \neq J') \wedge (I'' \neq J' \vee K'' \neq K')] \vee \\ [(I''' \neq I'' \vee K''' \neq J'') \wedge (I''' \neq J'' \vee K''' \neq K'')] \vee \Phi_0(I''',K'''). \end{array}$$

Where this gets dizzying is how the logic for (not) setting up pairs of IDs correctly all becomes hierarchical. But once you get the pattern, the final formula Φ_r is easy to stream out. Here is 16:

$$\begin{array}{c} \Phi_4(I,K) \equiv (\exists J)(\forall I',K')(\exists J')(\forall I'',K'')(\exists J'')(\forall I''',K''')(\exists J''')(\forall I'''',K''''): \\ [(I' \neq I \vee K' \neq J) \wedge (I' \neq J \vee K' \neq K)] \vee \\ [(I'' \neq I' \vee K'' \neq J') \wedge (I'' \neq J' \vee K'' \neq K')] \vee \\ [(I''' \neq I'' \vee K''' \neq J'') \wedge (I''' \neq J'' \vee K''' \neq K'')] \vee \\ [(I'''' \neq I''' \vee K'''' \neq J''') \wedge (I'''' \neq J''' \vee K'''' \neq K''')] \vee \Phi_0(I'''',K''''). \end{array}$$

Since the base predicate allows equality, this says "I can go in *at most* 16 steps to K" and so covers the case of 13 steps. The ultimate point is that we can get up to 2^r steps with just r rows. Since each row has 8 occurrences of IDs and hence requires $8s$ variables to code, the whole formula size is about $3rs$ for the quantifiers (which also tells that $3rs$ is the number of different variables, not counting those in I and K which get initialized to the binary encoding of $I_0(x)$ and I_f), plus $8rs$ for the rows, plus $2s$ for the final invocation of Φ_0. Since $s = O(s(n))$ and $r = O(s(n))$, we get size $O\big(s(n)^2\big)$.

- In the reduction to **TQBF**, $s(n)$ is $O\left(n^k\right)$ for some k, being the space used for M accepting the given language $A \in$ PSPACE, so $s(n)^2 = O\left(n^{2k}\right)$, which is still polynomial. There are $O\left(n^{2k}\right)$ binary variables, but each has a numerical index of size $\log_2\left(n^{2k}\right) = 2k\log n$ which is $O(\log n)$, so a log-space reduction can manage the indices while streaming. Thus TQBF is PSPACE-complete under $\leq_m^{\log}$.
- For Savitch's Theorem, what happens is that whether the resulting QBF instance Φ_r is true can be decided **deterministically** in space linear in its size, so space $O\left(s(n)^2\right)$. Since A can come from an arbitrary **N**TM running in $s(n)$ space, this proves that (provided $s(n) \geq \log n$) NSPACE$[s(n)] \subseteq$ DSPACE$\left[s(n)^2\right]$. [Dropping the O-notation in DSPACE$\left[s(n)^2\right]$ is a permissible fib.]

Some words to the wise:

1. If we have an opponent making moves too, then the space savings does not work over 2^r steps.
2. But if our alternating-move games are limited to r moves, say where r is polynomial in n, then we can get an $O(r)$-size formula directly just by the direct formulation shown at the outset in the previous lecture.

Theorem. Chess extended to $n \times n$ boards (with more of the same kinds of pieces, still just one king) is PSPACE-hard under $\leq_m^{\log}$. It is PSPACE-complete under $\leq_m^{\log}$ in the presence of a "generalized 50-move rule" limiting games to length $n^{O(1)}$. (But with no such limit it is---amazingly, IMHO---complete for EXP, i.e., exponential time $2^{n^{O(1)}}$. This is thought to imply that $n \times n$ chess without a polynomial length limit on games requires exponential space too.)

The further development is that many two- or multi-player games of strategy, not just chess, are PSPACE-complete to solve (presuming a reasonable limit on the length of games). Whereas NP-completeness characterizes myriad solitary (or co-operative) optimization problems, PSPACE-completeness characterizes optimization in the face of competition.

Oracle Turing Machines

Another facet of PSPACE as a "catchall" for polynomial-based complexity comes from **oracle Turing machines**. An **OTM** M is allowed instantaneous access to an external database in the form of an **oracle function** $f : \Sigma^* \to \Sigma^*$. At any point in its computation, M can write a **query string** y on a special oracle tape and enter a special **query state** $q_?$. In that state, the query tape is magically erased and replaced by the value $f(y)$ with the head reading the first bit of it. This idea was in Turing's original paper, and anything M does is said to **Turing-reduce** to f. In particular, we write $L\left(M^f\right)$ for the language of M *with oracle f* or *relative to f*.

Here is my favorite small-scale example using a function. Whether the product xy of two n-bit integers can be computed in $O(n)$ time is a major open problem. The algorithm learned in school takes time order-n^2. *Karatsuba's method* runs in time $O\left(n^{1+\epsilon}\right)$ for any prescribed $\epsilon > 0$ but the lower ϵ, the higher the constant in the O, and this tradeoff issue affects the known $\widetilde{O}(n)$ time methods even more. But if there were magically a linear-time algorithm for *squaring* a number, then we would get such an algorithm for multiplication in general.

Theorem: Except over fields of characteristic 2, integer multiplication Turing-reduces to the squaring function in linear time via the formula $xy = \frac{1}{2}\left[(x + y)^2 - x^2 - y^2\right]$.

If f is the 0-1 valued characteristic function of a language A, then we just write M^A. After any query y is submitted, M is left reading either 0 for "no, $y \notin A$" or 1 for "yes" on its tape. Many sources have M go to one of two special answer states q_y or q_n instead---this difference is immaterial. If C is a class of *machines*, then we write C^A to be the class of languages $L\left(M^A\right)$ over all machines M:

- $\mathsf{P}^A = \left\{L\left(M^A\right) : \textit{The DOTM M runs in polynomial time (with oracle } A)\right\}$
- $\mathsf{NP}^A = \left\{L\left(N^A\right) : \textit{The NOTM N runs in polynomial time (with oracle } A)\right\}$
- $\mathsf{PSPACE}^A = \{L(M^A) : \textit{The DOTM M runs in polynomial space (with oracle } A)\}$.