CS 710: Complexity Theory 2/09/2010

Lecture 5: Space-Bounded Nondeterminism

Instructor: Dieter van Melkebeek Scribe: Matthew Anderson & Michael Correll

Last lecture we discussed time-bounded nondeterminism. We discussed hierarchy results that showed given a little more time we could solve a strictly larger subset of problems. We then addressed the P vs. NP problem, proving that if P $\neq$ NP then there are problems in NP which are neither NP-complete nor in P and thus NP-intermediate. Finally, we explored relativization showing that there exists an oracle, A, for which $\mathrm{P}^A = \mathrm{NP}^A$ and an oracle, B, for which $\mathrm{P}^B \neq \mathrm{NP}^B$, this implies that techniques that relativize are not sufficient for resolving the P versus NP question.

1 Overview

Today we continue our discussion of nondeterminism by looking at space-bounded nondeterminism. We are able to prove stronger results in the space-bounded case than in the time-bounded case. We prove three main results relating space-bounded nondeterminism to other complexity classes.

Theorem 1. $\mathrm{NSPACE}(s(n) \subseteq \cup_{c>0} \mathrm{DTIME}(2^{cs(n)})$

Theorem 2. $\mathrm{NSPACE}(s(n)) \subseteq \mathrm{DSPACE}(s^2(n))$

Theorem 3. $co\,\mathrm{NSPACE}(s(n)) = \mathrm{NSPACE}(s(n))$

The proof of all the results assume that we have a space computable function, $s(n) : \mathbb{N} \to \mathbb{N}$ such that $s(n) \geq \log n$. The restriction that $s(n)$ must be space computable is not necessary for the proofs of these theorems, we discuss how to remove this restriction. We require $s(n) \geq \log n$, because space bounds lower than logarithmic cause strange behavior and make it difficult for the machine to act in an interesting manner on its linear input.

Theorem 1 says that everything that can be done in NL can be done in P and everything that can be done in NPSPACE can be done in EXP. Theorem 2 gives us that in only quadratically more space we can perform nondeterministic computation deterministically. No similar result is known for time-bounded computation. This result is made possible due to the fact that space is in a sense more "flexible" than time (e.g. we can reuse space on tapes where we cannot reuse time steps) it is in a qualitative sense "easier" to provide definite bounds and relationships between space complexity classes than it was with time complexity bounds. The third result, Theorem 3, follows from that fact that it is easier to compute the complement of a language in space-bounded computation because you can iterate over all of the witnesses and check that none are valid. We will actually prove that $co\,\mathrm{NSPACE}(s(n)) \subseteq \mathrm{NSPACE}(s(n))$, but by complementing twice we get the equality. Theorem 3 implies a tight space hierarchy for nondeterministic machines by using diagonalization.

Directly from these results:

Corollary 1. $\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subset \mathrm{PSPACE} = \mathrm{NPSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP}$.

Proof. L $\subseteq$ NL and P $\subseteq$ NP follow trivially from earlier results. NL $\subseteq$ P follows from Theorem 1 above, using $s(n) = \log n$. NP $\subseteq$ PSPACE follows from a consideration of a PSPACE machine that iterates through every possible witness, verifying membership and thus emulating the computation of an NP machine. PSPACE = NPSPACE follows from Theorem 2 above, since if there is only a quadratic overhead in converting from NSPACE to DSPACE, the conversion from from NPSPACE to DSPACE will introduce only a squared term, which is still polynomial and thus in PSPACE. □

Whether these inclusions are strict is an open problem, the only separations that are known come from the hierarchy results which we discussed in previous lectures. For example, we know L $\subsetneq$ PSPACE but not whether L $\subsetneq$ NP.

Corollary 2. *If $s, s' : \mathbb{N} \to \mathbb{N}$ such that $s(n)$ is space constructable and $s(n) = \omega(s'(n))$ then* NSPACE$(s'(n)) \subsetneq$ NSPACE$(s(n))$.

The proof of this corollary follows the same diagonalization strategy as we have seen before. Since complementation is easier in the space domain a constant factor is sufficient to accomplish strictly more.

Recall, from Lecture 1, that our definition of space usage of a machine is the index of highest indexed cell it accesses. If a machine runs in space s, the machine's state (tape heads, tape state, computation state) can be described using $O(s)$ bits.

2 Proof of NSPACE$(s(n)) \subseteq \cup_{c>0}$ DTIME$(2^{cs(n)})$

Proof. Consider a NTM M running in space $s(n)$. Fix an input $x \in \Sigma^n$. Consider the configuration graph, $G = (V, E)$, of M on input x. Each vertex represents a possible configuration of M that does not use more that $s(n)$ space. It follows that:

From this definition we can compute a bound for the number of vertices in this graph, viz.:

$$|V| \leq |Q| \cdot n \cdot |\Gamma|^{s(n)\cdot k}] \cdot s(n)^k = 2^{O(s(n))} \tag{1}$$

By construction, an input is accepted $\iff$ there exists a path from our initial state Q_o to an accepting state Q_f (WLOG we allow for one unique accepting state. This can be done by requiring that M clear the work tapes and move the tape heads to the beginning before accepting an input in a final state). Thus the problem of "is this input accepted on M" becomes a new problem, "is there a valid path from one vertex to another in a digraph G," or the ST-CONNECTIVITY problem. This problem can be solved in polynomial time. Thus a problem in NSPACE$(s(n))$ can be encoded as a problem in DTIME$(2^{O(s(n))})$. □

This proof works nicely if $s(n)$ is space constructable. If this is not the case or we do not know the value of $s(n)$, we can run the procedure several times while increasing a fixed space-bound until the computation has enough space to complete. More precisely, iterate over space bounds $s = 1, 2, 3, ...$ If the computation reaches an accepting state, then accept. If the computation does not accept but tries to exceed the space bound, repeat this procedure with a larger space bound. If the computation does not on any path try to exceed the space bound, then accept if there is a path to an accepting state, otherwise reject.

Corollary 3. *ST-CONNECTIVITY for directed graphs is complete for* NL *under* $\leq_m^{\log}$.

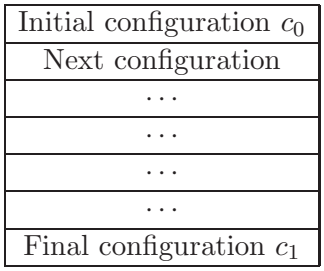

Figure 1: An illustration of the configuration tableau used in the proof of Theorem 2. The tableau has width $O(s(n)) \cdot 2^{O(s(n))}$.

Proof. ST-CONNECTIVITY $\in$ NL since we can use non-determinism to guess a valid path to take from S to T. This problem takes log time since we need to track current location, a counter to determine whether we are within our space bound n, and a comparison to the accepting vertex. Since ST-CONNECTIVITY has been shown in the proof above to correspond to an encoding of a NTM, it follows that it is also hard for NL. □

This corollary implies that if ST-CONNECTIVITY (STCON) $\in$ L we have L = NL. The STCON problem is also known as the Direction Path Problem (DPP). It is not known if DPP $\in$ L, however, the undirected version of the problem is.

3 Proof of NSPACE$(s(n)) \subseteq$ DSPACE$(s^2(n))$

Proof. Consider a NTM M running in NSPACE$(s(n))$. Fix input $x \in \Sigma^n$. Consider a computation tableau for a M: each row corresponds to a description of a step of computation, including inputs tape, work tape(s), tape head position, internal state, &c. From a similar argument used above, the width of one row of the tableau is $O(s(n)) \cdot 2^{O(s(n))}$. Assuming a unique final state (allowed WLOG by using a similar tape clearing scheme employed above, combined with the mandate that the most parsimonious computation route be used) there are $t(n)$ such rows in this tableau.

We will label the first row c_0 (encoding the initial configuration) and the final row c_1 (representing the accepting state's configuration). A row can be placed in the tableau $\iff$ it is a configuration representing a valid transition from the previous row.

Again, we would like to determine if there are a sequence of states that take us from c_0 to c_1 in t time and using no more than $s(n)$ space. A naive approach will take exponential space. To achieve the quadratic space blow up we use a divide and conquer strategy. Instead of going directly from c_0 to c_1 we guess an intermediate state $c_{\frac{1}{2}}$ and verify independently that we can go from c_0 to $c_{\frac{1}{2}}$ and $c_{\frac{1}{2}}$ to c_1. The benefit of doing it this way is that we can reuse the space used in the first part of the computation when we do the second part.

The highest level of the computation will look something like this:

$$x \in L(M) \Leftrightarrow (\exists c_{1/2})(\forall b \in \{0,1\}) c_{\frac{b}{2}} \vdash_M^{\frac{t}{2}} c_{\frac{b+1}{2}} \tag{2}$$

This reduces the original problem into two subproblems of half the size. We can apply this procedure recursively reusing the space that was taken up in the computation of the first part of the problem. Because we are halving the size of the problem at each level after $\log t$ recursions we reach the base

case where we have to verify that one state transitions to another in one step. We can phrase this as a full quantified boolean formula:

$$x \in L(M) \Leftrightarrow (\exists c_{\frac{1}{2}})(\forall b_1 \in \{0,1\})(\exists ...)(\forall ...)...(\exists c_{\frac{x}{t}})(\forall b_y \in \{0,1\}) c_{\frac{x+b_y-1}{t}} \vdash_M^1 c_{\frac{x+b_y}{t}} \tag{3}$$

The index x in the last part of Equation 4 is dependent on the previous choices for the b variables. The predicate checks whether the configurations guessed to come immediately before and after $c_{\frac{x}{t}}$ have valid one step transitions to and from $c_{\frac{x}{t}}$.

Consider how much space is required to evaluate this boolean formula. Each existential quantifier requires $s(n)$ space to describe the configuration. Each universal quantifier requires one bit of information to store its value. There are $\log t$ pairs of quantifiers so the total amount of space required is $O(s \cdot (\log t + 1)) = O(s^2)$.

□

Again we are implicitly assuming that s is space constructable, however we can apply the same technique as before to remove this requirement.

3.1 TQBF

The boolean formula which was introduced in the previous proof is called a quantified boolean formula. These types of formulas are part of an interesting language called true quantified boolean formulas. TQBF can be interpreted as a language that contains first order sentential logical formulae that happen to be true.

Definition 1 (True Quantified Boolean Formula (TQBF))**.** *The language of all quantified boolean formulas which have a number of quantifier alternations, contain no free variables and are true.*

Corollary 4. TQBF *is complete for* PSPACE *under* $\leq_m^{\mathrm{P}}$.

Proof. The proof of Theorem 2 gives a polynomial mapping reduction for a general PSPACE problem transforming it into a TQBF in polynomial time making TQBF hard for PSPACE. TQBF is in PSPACE because a PSPACE machine can iterate over all possibilities. □

Note that PSPACE $\subseteq \mathrm{P}^{\mathrm{TQBF}}$ by the corollary and $\mathrm{P}^{\mathrm{TQBF}} \subseteq$ PSPACE because all queries to the oracle can be constructed in polynomial time and we have closure under polynomial composition. Therefore $\mathrm{P}^{\mathrm{TQBF}} =$ PSPACE. Taking this idea further we get the following containment:

$$\mathrm{P}^{\mathrm{TQBF}} = \mathrm{PSPACE} = \mathrm{NP}^{\mathrm{TQBF}} \subseteq (\mathrm{PSPACE}^{\mathrm{TQBF}} = \mathrm{PSPACE}). \tag{4}$$

The last equality is model dependent because it matters whether you count the cells used on the oracle tape or not (if the oracle tape is not counted, a $\mathrm{PSPACE}^{\mathrm{TQBF}}$ machine could abuse the oracle tape to compute more than an unassisted PSPACE machine could). This also gives another example of an oracle (this time a natural one) for which $\mathrm{P}^A = \mathrm{NP}^A$. As before, note that techniques that relativize cannot be the sole means to solve the P versus NP question.

Many PSPACE complete problems can be interpreted as adversarial game theory calculations. TQBF can be thought of as a game between two players, $\forall$ and $\exists$, who place quantifiers in a boolean formula. $\exists$ wins if, after no player can move, the formula is true. Determining if there is a dominating strategy for $\exists$ given a formula is a PSPACE complete problem.

4 Proof of $co\,\mathrm{NSPACE}(s(n)) = \mathrm{NSPACE}(s(n))$

Proof. Consider a NTM M using space $s(n)$ on a fixed input $x \in \Sigma^n$. We saw before that given a graph and two vertices that reachability can be guessed reachability in $\log(|V|)$ space. This theorem shows that the complement can also be computed in log space, by showing that a path *does not* exist.

It is sufficient for this proof to show that $\overline{\mathrm{ST-CONNECTIVITY}} \in \mathrm{NL}$, since we have already shown that ST-CONNECTIVITY is complete for NL. By increasing the allowable size of the graph we use to represent the TM, this result will hold for higher spaces as well.

We can attempt this problem using a method called "inductive counting." For the purpose of this proof we will denote c_k as "the number of vertices reachable from S in at most k steps." We are interested in finding c_n, since this will solve the problem for us in the way listed above. We can find c_0 a priori (only one vertex, S, is reachable in 0 steps from S). If, given c_k, we can generate c_{k+1}, then by the mathematical principle of induction we can generate c_n.

The key idea behind this proof is that if it is known how many vertices are reachable from the start vertex u, then it can be verified that the destination vertex v is not one of those vertices by looking at all possible vertices and guessing and verifying whether they are reachable. In order to count the number of reachable vertices we use a procedure call inductive counting.

Suppose you have a subroutine that behaves as follows. The subroutine takes as input the graph G, the start vertex u, the destination vertex v, the number of steps allowed t and the number of vertices c_t within t steps of u. The subroutine has three possible return values: YES, NO and ?. A YES answer means that it is possible to reach v from u in $t+1$ steps. A NO answer means that it is not possible to reach v from u in a most $t+1$ steps. A ? return value means that the procedure was unable to determine the answer. Consider the ? return value a value that indicates that the subroutine either made bad internal guesses or got bad input, in either case it's output could not be trusted and that computation path is compromised. This subroutine will run in NL, is always correct if it answers YES or NO, and at least one computation path returns YES or NO. If value passed as c_t is not correct the subroutine will return ?.

We will describe the subroutine more specifically later.

Consider how we can use the subroutine to solve the original problem. Start with $c_0 = 1$ and try to compute c_1. Set $c_1 = 0$. Run the subroutine for all possible v. If the subroutine says YES add one to c_1 and continue, if the subroutine says NO continue, if the subroutine says ? halt and reject. The final value of c_1 after looping through all possible v's is the number of vertices reachable from u in one step. Iterate this procedure to compute c_{n-2}. Only the current and next values of c_i need to be stored at any point in the computation. Finally run the subroutine one last time with parameters (G,u,$n-1$,c_{n-2},v), if the answer is NO, accept if the answer is ? or YES reject.

The entire computation only accepts if there exists a computation path that correctly tracks the counts and v is not reachable from u. If the last step says YES there was a path from u to v. If the last step says ? the computation was inconsistent and it bails out by rejecting.

4.1 Writing the Subroutine

The idea for the subroutine is simple: cycle through all the vertices and guess if they are reachable. If the subroutines guess they are reachable it tries to verify they are reachable. If a vertex is reachable, the subrountine modifies the count of reachable vertices seen so far then checks if v is

reachable in one additional step. The count must match the count passed in as a parameter to verify that v was not reached.

SUBROUTINE(G,u,t,c_t,v)
Input: A graph G, a starting vertex u, a number of steps t, the number of vertices within t steps of u c_t, the destination vertex v
Output: YES if v is reachable from u in $\leq t+1$ steps, NO if v is not reachable from u in $let+1$ steps, ? if the computation is not able to determine. If c_t is correct at least one computation path will return the correct answer (not ?).

```
(1)   c ← ∅
(2)   foreach w ∈ V(G)
(3)     Guess whether w is reachable from u in ≤ t steps.
(4)     if Guessed yes
(5)       Verify guess by nondeterministically guessing a path from u to w in
          ≤ t steps.
(6)       if Successfully verified
(7)         if v can be reached from w in one step
(8)           return YES
(9)         else
(10)          c ← c + 1
(11)      else
(12)        return ?
(13)  if c = c_t
(14)    return NO
(15)  else
(16)    return ?
```

□

The main point of this technique is that if the number of positive instances (reachable vertices) is known guessing and verifying positive instances will verify whether the count is correct. Then the count can be used to check whether the instance in question is a member of the positive set.

5 Next Time

Next lecture we will discuss alternation as a generalization of the hypothetical model of nondeterministic computation. In particular alternation captures all languages that can be specified with a fixed number of quantifier alternations as a TQBF.