Notes for CS538 Lecture on 2/10/00

Detlef Ronneburger *

June 8, 2000

Closure of Nondeterministic Space under Complement

As a result of Savitch's theorem [Sav69], it is known that the deterministic and nondeterministic versions of PSPACE coincide. Thus, $\mathsf{NPSPACE}$ (and many other "large" nondeterministic space-bounded classes) are trivially closed under complement. For small space bounds, and in particular for logarithmic space, Savitch's theorem does not imply closure under complement.

However Immerman and Szelepcsényi independently [Imm88, Sze88] introduced the technique of inductive counting which they used to prove the following statement.

Theorem 1 ([Imm88, Sze88]). $\mathsf{NL} = \mathsf{coNL}$.

With a simple padding argument it is possible to translate this result to bigger space bounds.

Corollary 2. *Let $s \in \Omega(\log n)$ be a space constructible function, then*

$$\mathsf{NSPACE}[s] = \mathsf{co\text{-}NSPACE}[s] \quad .$$

Proof of Corollary 2. Let $s \in \Omega(\log n)$ be a space constructible function, and $A \in \mathsf{NSPACE}[s]$. Define

$$A' := \left\{ x10^i \mid \left|x10^i\right| = m,\ \log m = s(|x|) \text{ and } x \in A \right\} \quad .$$

First let us note that $A' \in \mathsf{NL}$. Since s is space constructible, the check whether $\log m = s(|x|)$ can be performed in logarithmic space in terms of m by marking $\log m$ tape cells and then constructing $s(|x|)$ on those marked cells. If the marked space is not sufficient the machine stops and rejects. Further we can decide $x \in A$ in nondeterministic space $s(|x|) = \log m$. Thus there is an NL–algorithm computing A'.

*detlef@paul.rutgers.edu

Using the closure of NL under complement we also know that $A' \in \mathsf{coNL}$ and thus $\overline{A'} \in \mathsf{NL}$. With a similar argument as before we can see that that the set $\overline{A''}$ is also contained in NL, where

$$\overline{A''} := \left\{ x10^i \mid \left|x10^i\right| = m,\ \log m = s(|x|) \text{ and } x \notin A \right\} \quad .$$

Now note that $x \in A$ if and only if $\exists i : \ x10^i \in A'$ and thus equivalently $x \in \overline{A}$ if and only if $\exists i : \ x10^i \in \overline{A''}$.

Combining this gives a nondeterministic Turing machine running in space $\mathcal{O}(s)$ which recognizes $\overline{A}$. The algorithm just has to nondeterministically guess a value for i (in space $\log i \leq s(|x|)$) and then simulate the NL–machine for $\overline{A''}$ to decide whether $x10^i \in \overline{A''}$. □

Now let us proceed with the proof of the theorem. We will show that closure of nondeterministic logarithmic space under complement, by proving the there is a NL–algorithm deciding the complement of the graph accessibility problem (GAP). Since GAP is complete for NL this yields the desired result.

Proof of Theorem 1. Let $G = (V, E)$ be a directed graph with source and target nodes $s, t \in V$. Further let $d(u, v)$ denote the distance from a node v to another node u. In other words $d(u, v)$ is the number of edges of the shortest path from u to v. Now define the value c_k as

$$c_k := |\{ v \in V \mid d(s, v) \leq k \}| \quad .$$

In other words c_k determines how many nodes have at most distance k from the source node s.

If we assume that we can compute c_n where $n = |V|$ in nondeterministic logarithmic space, then the following algorithm decides $\overline{\text{GAP}}$.

Algorithm 3 (deciding $\overline{GAP}$).

```
input:  <G = (V,E), s,t ∈ V>
    compute c_n;           /* c_n is the total number of vertices reachable from s
*/
    a := 0; found(t) := FALSE;
    foreach v ∈ V
        nondeterministically guess if v is reachable from s ;
        if (guess = YES) then
            a := a + 1;
            nondeterministically guess a path from s to v;
            if (guess is not possible) then
                reject;
            else
                if (v = t) then found(t) := TRUE;
    next v;
    if (a = c_n and not found(t)) then
        accept;
    else
        reject;
```

The key idea in this algorithm is that for each vertex we nondeterministically guess whether it is connected to s or not. We then can verify whether our guess was good or not. In particular we keep track of whether we find a path from s to t. At the end we only accept if we guessed the correct number of vertices and we did not find a path from s to t.

Each branch of computation checks of all the possible combinations of vertices which could be reachable from s and accepts if and only if that particular combination covered exactly all the vertices reachable from s and t was not one of them.

Note that no branch accepts if t is reachable from s since either a path from s to t was found or the number of vertices reachable from s was guessed incorrectly.

Therefore the algorithm accepts an instance of $\overline{\text{GAP}}$ if and only if there is not a path from s to t in G.

Now we still need to prove that we can compute c_n in nondeterministic logarithmic space. Here we will say that a nondeterministic machine computes a value x if each branch of computation that does not reject outputs the same result x.

We will compute the different values c_i in an inductive fashion. Clearly $c_0 = 1$ and thus it is trivial to compute in logarithmic space. So let us assume c_k is computable within our resource bounds and give an algorithm that computes c_{k+1}.

Algorithm 4 (computing c_k).

```
input:  <G = (V,E), s ∈ V, k>
    compute c_k;
    result := 0;
    foreach (v ∈ V)        /* want to check if d(s,v) ≤ k+1 */
        foreach (w ∈ V with (w,v) ∈ E )        /* want to check if d(s,w) ≤ k */
            l := 0; found(w) := FALSE;
            foreach (x ∈ V)
                nondeterministically guess if d(s,x) ≤ k ;
                if (guess = YES) then
                    l := l+1;
                    nondeterministically guess a path of length ≤ k from s to x;
                    if (guess is not possible) then
                        reject;
                    else
                        if (x = w) then found(w) := TRUE;
            next x;
            if (l = c_k) then
                    if (found(w)) then
                        result := result + 1;
                        next v;
                    else
                        next w;
            else    /* l ≠ c_k */
                reject;
output:  <result>
```

In order to compute how many vertices have distance at most $k+1$ from s, check for every possible vertex v whether it is adjacent to some other vertex w which has distance less than or equal to k.

The check for the distance of w is very similar to Algorithm 3. The only difference is that instead of just focusing on the notion of whether there is a path at all, now we are only interested in paths of length at most k.

If at the end we did not find all the vertices within distance k (i.e. $l \neq c_k$) the branch of computation just rejects. Otherwise we check whether we found a path of length $\leq k$ for w or not. In the first case $d(s,v) \leq k+1$ and we can consider the next v, in the second case w was not a good choice as an "intermediate" step and thus we consider the next w adjacent to v.

The entire computation only needs to store a constant number of nodes and counters each having logarithmic length description. Also the recursive call to compute c_k occurs only one at the very beginning and thus we can reuse the space for the computation for each c_i. So there is an NL–machine that can compute c_k.

Combining this with Algorithm 3 yields that $\overline{\mathrm{GAP}} \in \mathsf{NL}$ and therefore $\mathsf{NL} = \mathsf{coNL}$. □

If we have a nondeterministic Turing machine M deciding $A \in \mathsf{NL}$ in logarithmic space, the previous theorem gives another nondeterministic Turing machine M' deciding $\overline{A}$ within the same space bounds. Running those two machines in parallel enables each single path of computation to decide whether $q \in A$ or $q \in \overline{A}$. Therefore a deterministic logspace machine can answer queries to any set $A \in \mathsf{NL}$ by itself. Therefore the following corollary holds.

Corollary 5. $\mathsf{NL} = \mathsf{L}^{(\mathsf{NL})}$.

Using this corollary it becomes clear that we can decide problems like Shortest Path or Transitive Closure in nondeterministic logarithmic space.

NL–Computations can be Unambiguous

To show the power of inductive counting let show a result from Allender and Reinhardt ([RA97], [ARZ99]) in which they consider that in certain settings the restricted version of nondeterministic logspace with only a single accepting path is equally powerful as the unrestricted case.

For that purpose let us define the class UL. We call a Turing machine unambiguous, if for every input it has at most one accepting path of computation. Using this notion we define

$$\mathsf{UL} := \{\, A \in \mathsf{NL} \mid A \text{ can be accepted by an unambiguous } \mathsf{NL}\text{–machine} \,\} \quad .$$

Clearly $\mathsf{L} \subseteq \mathsf{UL} \subseteq \mathsf{NL}$.

In [RA97] Allender and Reinhard show that $\mathsf{NL} = \mathsf{UL}$ if the UL–machine has access to a "good choice of random bits". This means that this result only holds in the nonuniform case. In [ARZ99] they derandomize their argument under a likely complexity theoretic assumption. We will focus on the nonuniform result.

We will call a Turing machine nonuniform if additionally to the input it has access to polynomially long extra information (the *advice*) which can arbitrarily depend on the length of the input. Given that poly is the class of all functions $f : \mathbb{N} \to \Sigma^*$ such that $|h(n)| \leq p(n)$ for some polynomial p, we can formally define the two nonuniform complexity classes $\mathsf{NL}/_{\mathsf{poly}}$ and $\mathsf{UL}/_{\mathsf{poly}}$ as follows.

$$\mathsf{NL}/_{\mathsf{poly}} := \{\, A \mid \exists B \in \mathsf{NL} \text{ and } \exists h \in \mathsf{poly} \text{ such that } (x, h(|x|)) \in A \iff x \in B \,\} \quad ,$$

$$\mathsf{UL}/_{\mathsf{poly}} := \{\, A \mid \exists B \in \mathsf{UL} \text{ and } \exists h \in \mathsf{poly} \text{ such that } (x, h(|x|)) \in A \iff x \in B \,\} \quad .$$

Using those definitions we can state the results of Allender and Reinhardt.

Theorem 6 ([RA97]). $\mathsf{NL}/_{\mathsf{poly}} = \mathsf{UL}/_{\mathsf{poly}}$.

Theorem 7 ([ARZ99]). *If there is a set $A \in \mathsf{DSPACE}[n]$ and a constant $\varepsilon > 0$ such that A does not have circuits of size $< 2^{\varepsilon n}$, then $\mathsf{NL} = \mathsf{UL}$*

In order to prove Theorem 6 we will need the notion of a min-unique graph.

Definition 8. A directed graph $G = (V, E)$ is *min-unique* with respect to a source node s, if for any node $v \in V$ with distance $d(s, v) = k$ there is at most one (and thus exactly one) path of length k from s to v.

So a graph is min-unique, if every node v that can be reached from s has a unique shortest path from s to v.

Proof of Theorem 6. We will generalize the inductive counting technique from before slightly to show that the following language $A \in \mathsf{UL}$.

$$A := \{ (G_1, \ldots, G_k, s, t) \mid \exists i\, G_i \text{ is min-unique}, \forall j < i : G_j \text{ is not min-unique}, \\ s, t \in \bigcap G_i \text{ and } (G_i, s, t) \in \text{GAP} \} \quad .$$

Let us define c_k as before, namely

$$c_k := |\{ v \in V \mid d(s, v) \leq k \}| \quad .$$

Additionally we will also keep track of the cumulative distance of all vertices close to s which we will denote by Σ_k.

$$\Sigma_k := \sum_{\{ v \mid d(s,v) \leq k \}} d(s, v) \quad .$$

Now we can give an easy UL–algorithm deciding A under the assumption that we can compute c_n and Σ_n (or detect that the given graph is not min-unique) unambiguously within our resource bounds. We also require that given c_n and Σ_n we can compute the predicate $d(s, t) \leq n$ unambiguously in logspace.

Algorithm 9 (deciding A).

```
input:  <G_1, ..., G_r,  s,t ∈ ⋂ G_i>
    GoodGraph := TRUE;
    foreach (G_i)
        try to compute c_n and Σ_n;
        if (GoodGraph = FALSE) then
            next G_i;
        else
            if (d(s,t) ≤ n) then
                accept;
            else
                reject;
```

Essentially the algorithm tries to find the first graph in the sequence that is min-unique, by trying to compute c_n and Σ_n. With the help of those two values it then determines if t can be reached from s or not by checking whether $d(s,t) \leq n$.

Except for the computation of c_n, Σ_n and $d(s,t) \leq n$ the algorithm works in deterministic in logarithmic space. Since we assumed that we can perform those computations in unambiguous logspace, we showed $A \in \mathsf{UL}$.

The values c_0, Σ_0 and $d(s,w) \leq 0$ are trivial to compute. Given c_k and Σ_k as well as knowing whether $d(s,w) \leq k$, we will show how to compute c_{k+1} and Σ_{k+1}. Further we will also compute $d(s,w) \leq k+1$ based on c_{k+1} and Σ_{k+1}.

Let $G|_i$ denote the subgraph of all vertices of G with distance at most i from s.

Algorithm 10 (computing c_{k+1} and Σ_{k+1}, rejecting if $G|_{k+1}$ not min-unique).

```
input:  <G = (V, E), s ∈ V, k>
    compute c_k and Σ_k;
    if (G|_k is not min-unique) then
        return <G|_{k+1} not min-unique>;
    result_c := c_k; result_Σ := Σ_k
    foreach (v ∈ V) with (d(s,v) ≰ k )    /* check d(s,v) = k+1, add length to Σ_k */
        foreach (w ∈ V)
            if (d(s,w) ≰ k−1 or (w ↛ v)) then
                reject;    /* w not neighbor of v with distance k from s */
            else
                foreach (w' ∈ V)
                    if (d(s,w') ≤ k−1 and (w → v)) then
                        return <G|_{k+1} not min-unique>;
                next w';
                /* found w such that s -k-> w → v is unique shortest path to v */
                result_c := result_c + 1;
                result_Σ := result_Σ + k + 1;
        next w;
    next v;
output:  <result_c, result_Σ>;
```

The `foreach`–loop counts all the vertices which have distance exactly $k+1$ from s by finding all the nodes $v \notin G|_k$ that are adjacent to a node $w \in G|_k$. Provided that $G|_{k+1}$ is min-unique, the algorithm counts all those nodes and sums up the length of all their shortest paths from s to v.

Since $G|_k$ is min-unique, the shortest path from s to v is unique unless there are two different nodes $w, w' \in G|_k$ adjacent to v. If the shortest path from s to v is unique for all $v \in G|_{k+1} \cap G|_k$ it follows that $G|_{k+1}$ is also min-unique.

The algorithm does not perform any potentially ambiguous computation unless before checking whether $G|_k$ is min-unique. Furthermore it runs unambiguously provided that $G|_k$ is min-unique.

The previous algorithm relies on the fact that we are able decide whether $d(s, v) \le k$ or not. The following algorithm uses inductive counting for exactly that purpose.

Algorithm 11 (deciding $d(s, v) \le k + 1$).

```
input:  <G = (V, E), s, v ∈ V, k>
    compute c_{k+1} and Σ_{k+1};
    if (G|_{k+1} is not min-unique) then
        return <G|_{k+1} not min-unique>;
    else
        a := 0; b := 0; found(v) := FALSE;
        foreach (x ∈ V)
            nondeterministically guess if d(s, x) = k ;
            if (guess = YES) then
                nondeterministically guess an l ≤ k and path of length l from s to x;
                if (guess is not possible) then
                    reject;
                else
                    if (x = v) then found(v) = TRUE;
                a := a + 1;
                b := b + l;
        next x;
        if (a = c_k and b = Σ_k and found(v)) then
            accept;
        else
            reject;
```

The algorithm modifies the standard inductive counting argument by additionally keeping track of the sum of all shortest paths from s in $G|_{k+1}$. This is necessary to ensure that we do not accidentally count a node twice for different distances.

If we incorrectly guess $d(s, x) = m$ for some $m \le k$ we will not be able to find a path of that length. If we incorrectly guess $d(s, x) = m$ for some $m \ge k$ the number of c_{k+1} of found vertices might be correct, but the sum of the path lengths will not be right.

The above procedure works unambiguously provided that $G|_{k+1}$ is min-unique. In this case there will always be one unique shortest path for each vertex and only one combinations of correct guesses for $d(s, x) = k$. If the graph is not min-unique we will detect it in the beginning.

Now we inductively constructed algorithms to compute c_k, Σ_k and $d(s, v) \le k$ for any k. Together with Algorithm 9 this yields $A \in \mathsf{UL}$.

To finish this proof we will show next lecture that we can nonuniformly reduce GAP $\le_m^{\mathsf{AC}^0} A$. In other words we will show how we can take a graph G and two nodes $s, t \in G$ and generate a sequence of graphs $G_1, \ldots G_r$ such that for some i the graph G_i is min-unique, every graph

G_j for $j \leq i$ is not min-unique and G_i contains a path from s to t if and only if the original graph does. $\square$

References

[ARZ99] E. Allender, K. Reinhardt, and S. Zhou. Isolation, Matching, and Counting: Uniform and Nonuniform Upper Bounds. *Journal of Computer and System Science*, 1999.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, October 1988.

[RA97] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. In *38th Annual Symposium on Foundations of Computer Science*, pages 244–253, Miami Beach, Florida, 20–22 October 1997. IEEE.

[Sav69] W. J. Savitch. Deterministic simulation of nondeterministic turing machines. In *ACM Symposium on Theory of Computing (STOC '69)*, pages 247–248, New York, May 1969. ACM Press.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, November 1988.