Lecture 6

The Immerman-Szelepscényi Theorem

January 22, 1998
Lecturer: Richard Ladner
Notes: Andrew Keller

6.1. Immerman-Szelepscényi Theorem

Theorem 6.1 (Immerman-Szelepscényi): If $s(n) \geq \log n$ and tape constructible, then

$$\text{NSPACE}(s(n)) = \text{CoNSPACE}(s(n)).$$

Proof:

It suffices to show $L \in \text{NSPACE}(s(n)) \Rightarrow \overline{L} \in \text{NSPACE}(s(n))$. Let T be a $s(n)$ space bounded nondeterministic Turing Machine. Fix x to be an input of length n. Define three nondeterministic space $s(n)$ bounded partial functions:

1. Count(k) = number of configurations reachable from the initial configuration in $\leq k$ steps of T. Count is a total function.

2. Reach(C, k) = true if and only if configuration C is reachable from the initial configuration $Init(x)$ in $\leq k$ steps of T. Reach is only defined when it is true.

3. NoReach(C, k) = true if and only if configuration C is not reachable from the initial configuration $Init(x)$ in $\leq k$ steps of T. NoReach is only defined when it is true.

Definition 6.2: A partial function $f(x)$ is *non-deterministic computable* in space $s(n)$ (or time $t(n)$) if there is a nondeterministic Turing Machine with output tape such that for all x:

- all computations run in allotted space $s(n)$ (or time $t(n)$),

- if $f(x)$ is defined then there is at least one computation that leads to acceptance with $f(x)$ on the output tape,

- if $f(x)$ is defined then any output produced in an accept state is $f(x)$.

Using the above three nondeterministic functions, an algorithm for $\overline{L}$ can be written:

```
for all configurations C
    if C is accepting then NoReach(C,2**(cs(n)))
```

The algorithms for the three functions are:

- Algorithm for Reach(C, k):

```
run T nondeterministically from Init(x) for <= k steps
if C is reached, return true
```

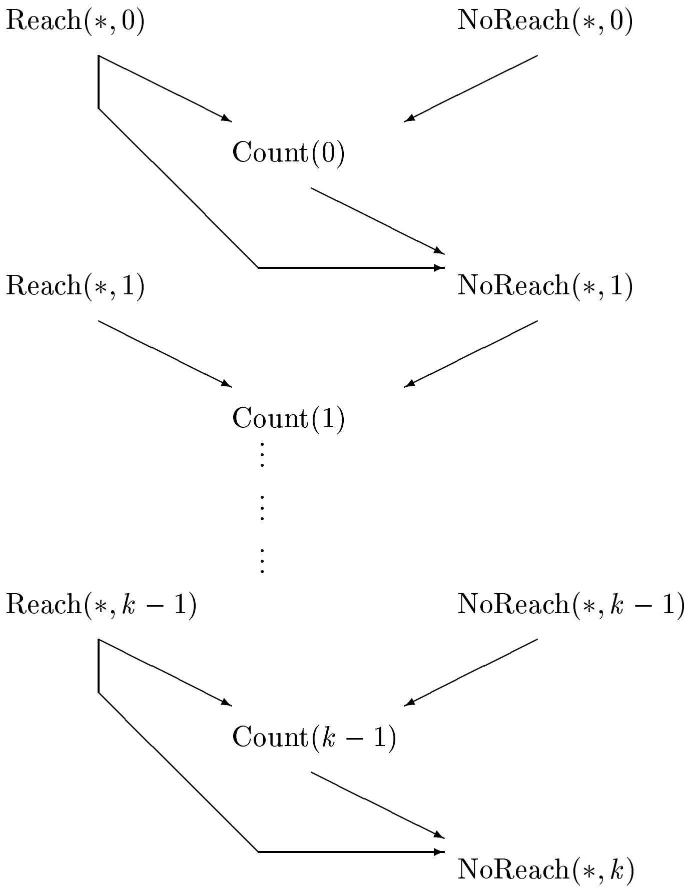
- Algorithm for NoReach(C, k):

```
if k = 0
   if C <> Init(x) return true
   else return false
if k > 0
   m := 0
   for all D
      if Reach(D, k-1) and C is not reachable from D in <= 1 step then m := m+1
   if m = Count(k-1) return true
```

- Algorithm for Count(k):

```
m := n := 0
for all C do one of:
     if Reach(C,k) then m := m+1
     if NoReach(C,k) then n := n+1
if m+n=M where M is the total number of configurations, return m
```

To compute NoReach(k), one first needs Reach($k-1$) and Count($k-1$), and to compute Count($k-1$), one first needs Reach($k-1$) and NoReach($k-1$). One can thus compute NoReach(C, k) by iteratively computing Count(0), Count(1),.....,Count($k-1$). The following figure describes the dependencies among the partial functions.



The algorithm for $\overline{L}$ executes the computation of NoReach$(C, 2^{cs(n)})$ in such a manner. Computing Reach(C, k) requires $O(s(n) + \log n + \log k) = O(s(n) + logk)$ space. Furthermore, if both NoReach$(C, k-1)$ and Count$(k-1)$ also require only $O(s(n) + \log k)$ space, then so will NoReach(C, k). Since computing NoReach$(C, 0)$ and Count(0) satisfy that space requirement, one may conclude by induction that computing NoReach(C, k) requires $O(s(n) + \log k)$ space, and that $\overline{L}$ can be computed in space $O(s(n) + \log 2^{cs(n)}) = O(s(n))$ space.

$\square$

6.2. Propositional Logic

Boolean values T,F (constants)

Boolean variables X_1, X_2, ...

Operators $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, \oplus, \equiv$

Syntax for Boolean Formulas e.g. $x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$

Operator Precedence $\neg, \wedge, \Rightarrow, \equiv, \vee, \oplus$

The semantics of a formula is an interpretation of its variables. To be specific, an assignment σ maps variables to Boolean values: $\sigma(X_i) \in \{T, F\}$. For the constants T and F, $\sigma(T) =$ T, and $\sigma(F) =$ F σ can be extended to all formulas in the natural way where each operation symbol is interpreted by the correct operator, e.g.

$$\sigma(\alpha \wedge \beta) = \sigma(\alpha) \wedge \sigma(\beta)$$

Define $\sigma \models \alpha$ if and only if $\sigma(\alpha) =$T. Further, α is *valid* if $\sigma(\alpha) =$ T for all σ, or $\sigma \models \alpha$ for all σ σ is *satisfiable* if $\sigma(\alpha) =$ T for some σ

We have the following logic relationship between satisfiability and validity.

$$\models \alpha \Leftrightarrow (\neg\alpha \text{ is } \textit{not} \text{ satisfiable})$$

Naturally there are many laws about logical identities. Here are a few.

$$\models (\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$$

$$\models \neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

$$\models \alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\beta \vee \gamma)$$

A statement is in *Disjunctive Normal Form* (DNF) if it is expressed as:

$$\bigvee_{i=1}^{m} (\bigwedge_{j=1}^{m_i} z_{ij})$$

where z_{ij} are literals (T,F,X,$\neg X$, etc). For example: $(x \wedge y) \vee (x \wedge \neg y)$.

A statement is in *Conjunctive Normal Form* (CNF) if it is expressed as:

$$\bigwedge_{i=1}^{m} (\bigvee_{j=1}^{m_i} z_{ij}).$$

Theorem 6.3 (Basic Theorem): For all α there is a β in CNF and γ in DNF and

$$\models \alpha \equiv \beta \equiv \gamma$$

Proof: Let α be a statement with variables X_1, X_2, ..., X_k. Then one can construct a truth table in which the value of $\sigma(\alpha)$ is tabulated for all 2^k possible combinations of $\sigma(X_1)$, $\sigma(X_2)$, ..., $\sigma(X_k)$. The truth table for $\sigma(\alpha)$ is:

assignment	*variable*				
	X_1	X_2	...	X_k	$\sigma(\alpha)$
σ_1	T	T	...	T	T
σ_2	T	T	...	F	F
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
σ_{2^k}	F	F	...	F	T

Define a conjunction C related to σ,

$$C(\sigma) = z_1 \wedge z_2 \wedge ... z_k$$

where $z_i = \{\ x_i$ if $\sigma(x_i) = \text{T}$, $\neg x_i$ if $\sigma(x_i) = \text{F}\}$.

Similarly, define a disjunction D related to σ,

$$D(\sigma) = \neg z_1 \vee \neg z_2 \vee ... \neg z_k$$

where double negatives are removed.

Then,

$$\gamma = \bigvee_{\sigma:\sigma(\alpha)=T} C(\sigma),$$

$$\beta = \bigwedge_{\sigma:\sigma(\alpha)=F} D(\sigma).$$

□