Computational Complexity Theory **Spring 1997**

Lecture 1 : January 7, 1997

Lecturer: V. Vinay *Scribe: P. R. Subramanya*

1.1 Boolean Circuit Complexity

Definition 1 *A* **Boolean Circuit** *is a directed acyclic graph (***DAG***) whose source vertices are the inputs and the sink vertex w.l.o.g. is a single output vertex. The internal vertices are each associated with a Boolean operation from the basis* $\{\vee, \wedge, \neg\}$.

A Fixed Boolean Circuit takes a fixed input length. When we talk about a circuit for a function or a language, we essentially are referring to a **family** of circuits $\{C_n\}_{n\geq 0}$, one for each input length. That is, given an input x, we choose a circuit $C_i \in \{C_n\}_{n\geq 0}$ such that, $|x| = i$. x is applied to C_i to evaluate the function on the given input.

Henceforth, when we refer to a Boolean Circuit for a function or a language, we are actually referring to the family of circuits.

1.1.1 What are the resources in this Model of Computation?

- **Size**: Number of gates in the circuit.
- **Wires**: Number of wires used in the circuit.
- **Depth**: Depth of the circuit.
- **Fan-in** of the gates. This can classify circuits into three different types:-
  1. *Bounded* circuits. The fan-in of all gates in the family of circuits is $\leq k$ for some fixed k. That is, the fan-in is independent of n.
  2. *Unbounded* circuits. The fan-in of the gates in the circuits of the family could depend on n.
  3. *Semi-unbounded* circuits. The $\vee$-gates are unbounded while the $\wedge$-gates are bounded.
- **Monotone** circuit. $\neg$-gates are not allowed. Any circuit that does not change its output from a 1 to a 0 when an input changes from a 0 to a 1 is a *monotone* circuit. Every *monotone* function can be realized by a *monotone* circuit and vice-versa.

  Is a *non-monotone* circuit implementation of a *monotone* function smaller than its *monotone* counterpart? This is not obvious, but true.

1.2 Example - Threshold 2 Function

Let us consider a monotone function Th_2^n. Given n inputs, the function evaluates to a 1 when there are at least 2 inputs that are 1s. That is,

$$Th_2^n(x_1, x_2, \ldots, x_n) = \begin{cases} 1, & \text{if } \sum x_i \geq 2 \\ 0, & otherwise \end{cases}$$

A few realizations of Th_2^n are given below to highlight the resources required.

1.2.1 An Obvious Realization

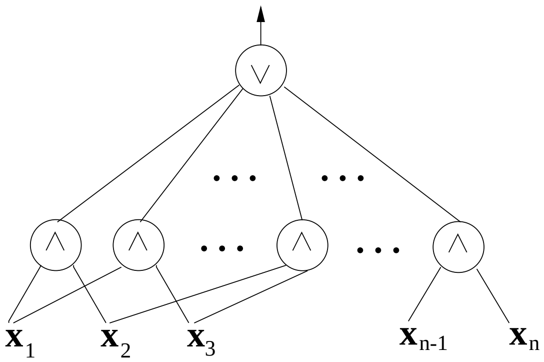

Figure 1.1: A Simple Solution

An obvious way to realize the Th_2^n function is to AND every pair of inputs. If at least two 1s are present in the input, some $\wedge$ pair will fire. $\vee$-ing all the $\wedge$ pairs will give us our realization of Th_2^n. Fig 1.1 illustrates the circuit realization.

The number of gates is $O(n^2)$, number of wires is $O(n^2)$ and depth is two, i.e. $O(1)$. This realization belongs to the class $mSAC^0$ (*monotone Semi-unbounded Alternating Class* of constant depth). **Monotone** because it does not use $\neg$-gates; **semi-unbounded** since it has unbounded $\vee$-gates and bounded $\wedge$-gates; and **alternating** since on any path from an input to the output, the $\vee$s and $\wedge$s alternate.

1.2.2 A Partitioning Approach

Suppose we partition the input into two sets. $\vee$ each set and then $\wedge$ the outputs. This gives a 1 **if and only if** there are two 1s in the input that have not been put in the same set. Now, there is no single partition that will yield the desired result. And, trying out all possible partitions is highly cumbersome (i.e. all 2^n ways of partitioning). We will split the input into $n-1$ pairs of sets. That is,

$$\{\{x_1\},\{x_2,\ldots,x_n\}\},\ldots,\{\{x_1,\ldots,x_i\},\{x_{i+1},\ldots,x_n\}\},\ldots,\{\{x_1,\ldots,x_{n-1}\},\{x_n\}\}$$

Fig 1.2 shows a circuit for the above approach. The number of gates is $O(n)$, the number of wires is $O(n^2)$ and the depth is $O(1)$. This too belongs to the class $mSAC^0$.

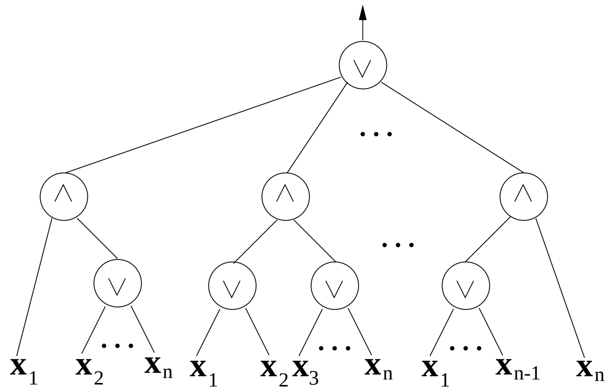

Figure 1.2: Partitioning Input Approach

1.2.3 A Recursive Approach

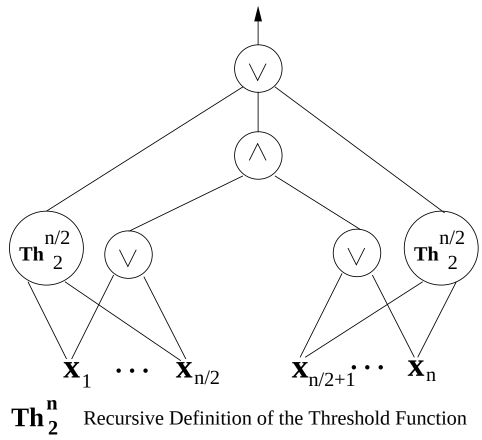

Figure 1.3: Recursive Approach

The recursive approach involves splitting the input into 2 halves and solving the $Th_2^{n/2}$ on each half. The only case for which this fails is when each half has exactly one input that is a 1. Therefore, one needs to $\vee$ each half and $\wedge$ the two. Fig 1.3 illustrates this approach.

The number of gates is $O(n)$, the number of wires is $O(n^2)$ and the depth is $O(\log n)$. This belongs to the class NC^1 since all the gates have bounded fan-in. This realization consumes too much of the depth resource.

1.2.4 A Hiding Approach

Suppose one masks off a single input and $\vee$s the remaining inputs. If the masked input is a 1 and this happens to be the sole 1 in the input, the output is a 0. If there are at least 2 1s in the input, irrespective of which input is masked, every such $\vee$-term will be a 1. Hence, the strategy is to hide one input at a time and $\vee$ the rest of the input. And then $\wedge$ all the $\vee$-terms.

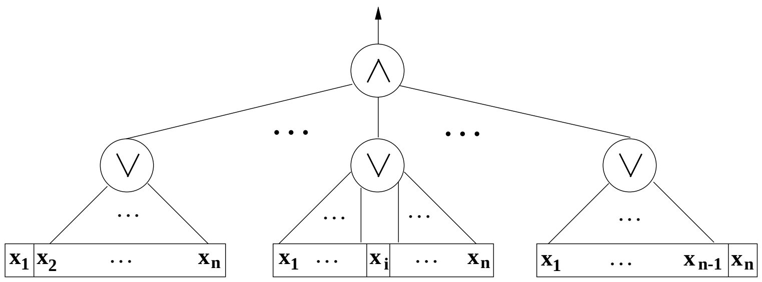

Figure 1.4: Hide 1-input method

The above scheme is shown in Fig 1.4. $O(n)$ gates with $O(n^2)$ wires and a constant depth is required to realize this approach. This realization belongs to the class AC^0. $O(n^2)$ wires is rather high and we would like an $O(n)$ wire realization.

1.2.5 A Linear Wires Realization

Suppose one arranges the inputs on a square grid of length $n^{1/2}$ by padding unoccupied grids by 0s. Next, one $\vee$s each row and column separately. If there exists two 1s in the input, either 2 rows or 2 columns will necessarily have 1s. Hence, if one does a Th_2^n on the output of the row and column $\vee$s separately, one of them will succeed. Fig 1.5 illustrates the approach.

$\vee$-ing the rows and columns requires $2n$ wires and performing $Th_2^{n^{1/2}}$ on the row and column outputs requires $O(n)$ using the Hiding Approach. That is, $O(n)$ wires in total. Number of gates is still $O(n)$ and the depth is $O(1)$.

A variation of the above scheme performs a $Th_3^{2n^{1/2}}$ on the combined row and column outputs.

Exercise: How does one achieve a $2n + 4n^{1/2} + o(n^{1/2})$ realization for Th_2^n? *Hint:* Optimize the above technique.

Exercise: How does one realize Th_3^n with linear number of wires in constant depth? Can the above method be generalized for Th_k^n? What about $O(\log n)$ gates realization for Th_3^n?

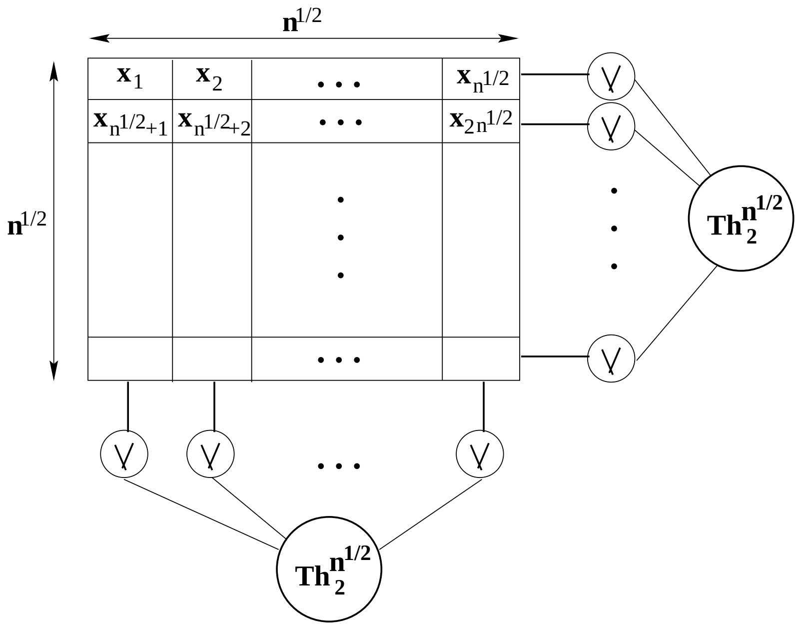

Figure 1.5: Grid realization for Linear number of wires

1.2.6 A $\log n$ gates Realization

Consider the binary representation of the numbers $\{1, 2, \ldots, n\}$. $O(\log n)$ bits are required to represent each of the numbers. Any 2 distinct numbers i and j from this set necessarily differ on some bit. Suppose we partition the set of inputs based on the value of the i^{th} bit. We will have $\log n$ sets of partitions. That is,

$$Th_2^n(x_1, x_2, \ldots, x_n) = \bigvee_{1 \leq j \leq \log n} F^j \text{ where } F^j \text{ is defined as,}$$

$$F^j = (\bigvee_{i \in Bin_0(j)} x_i) \wedge (\bigvee_{i \in Bin_1(j)} x_i)$$

$$Bin_0(j) = \{i : j^{th} \ bit \ of \ i \ is \ 0\} \qquad Bin_1(j) = \{i : j^{th} \ bit \ of \ i \ is \ 1\}$$

Each F^j requires 3 gates and there are $\log n$ of these. Number of gates is $O(\log n)$ and depth is still $O(1)$. The number of wires is $O(n \log n)$.

Note: This method cannot be extended to Th_3^n for the simple reason that in base 3, we could have a triple where none of the numbers differ on a single bit. For example, $\{122, 221, 222\}$. What is the general method for achieving $O(\log n)$ gates for Th_k^n for a fixed k?

Exercise: Give a realization for $Th_{\log n}^n$. Is this in AC^0?

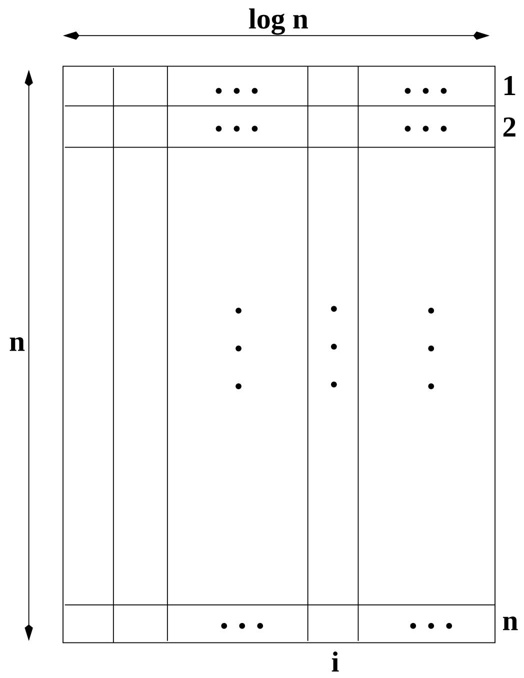

Figure 1.6: Log n gates realization for Th_2^n

Computational Complexity Theory	**Spring 1997**
Lecture 2-3 : January 9 & 16, 1997	
Lecturer: V. Vinay	*Scribe: P. R. Subramanya*

2.1 On SAC^0, AC^0, TC^0 and NC^1

In this lecture we shall try to show the following hierarchy:-

$$AC^0 \subset TC^0 \subseteq NC^1 \subseteq DLOG \subseteq SymLOG \subseteq NLOG \subseteq LOGCFL \ldots \subseteq P$$

We shall concentrate on the first 3 inclusions.

2.1.1 Some definitions

Definition 2 *AC^0 (Alternating Class) is the class of Boolean Circuits of $O(1)$ depth and polynomial size (i.e. number of gates) with unbounded fan-in gates over the basis $\{\wedge, \vee, \neg\}$.*
SAC^0 (Semi-unbounded AC^0) is the class of AC^0 circuits with the restriction that the gates are semi-unbounded.
TC^0 (Threshold Class) has the restriction that the gates are Threshold gates (i.e. Th_k^n for any k). $\neg$-gates are also allowed.

Definition 3 *NC^1 (Nick's Class) is the class of Boolean Circuits of $O(\log n)$ depth and polynomial size with* bounded *fan-in gates over the basis $\{\wedge, \vee, \neg\}$.*

2.2 $AC^0 \subset TC^0$

An $\wedge$-gate corresponds to Th_n^n and an $\vee$-gate corresponds to Th_1^n. Hence, this inclusion is trivial. In order to show that TC^0 contains circuits that are not in AC^0, we require some arithmetic.

ADDITION shall be shown to belong to the class AC^0. We shall then go on to show that *MULTIPLICATION* belongs to TC^0 and not AC^0.

2.2.1 *ADDITION*

Given two n-bit numbers $x_{n-1}x_{n-2}\ldots x_1x_0$ and $y_{n-1}y_{n-2}\ldots y_1y_0$, the standard *ripple carry* method for adding them is to define the sum as,

$$\begin{aligned} z_0 &= x_0 \oplus y_0 \\ c_0 &= x_0 \wedge y_0 \\ z_i &= x_i \oplus y_i \oplus c_{i-1} \\ c_i &= (x_i \wedge y_i) \vee ((x_i \vee y_i) \wedge c_{i-1}) \end{aligned}$$

This is a rather naive method with $O(n)$ depth! We are interested in constant depth circuits.

Suppose, we were to do the following,

$$p_i = x_i \wedge y_i \qquad P_i = x_i \vee y_i$$

$$\begin{aligned} c_i &= p_i \vee (P_i \wedge c_{i-1}) = p_i \vee (P_i \wedge p_{i-1}) \vee (P_i \wedge P_{i-1} \wedge c_{i-2}) \\ &= \sum_{j=0, i \geq j}^{i} P_i \wedge P_{i-1} \wedge \ldots P_{j+1} \wedge p_j \end{aligned}$$

We compute p_i and P_i at the first level. These are then combined in the next level to compute c_i. The z_i's are computed in the 4^{th} level. Fig 2.1 shows an AC^0 implementation for $ADDITION$. This shows that $ADDITION \in AC^0$.

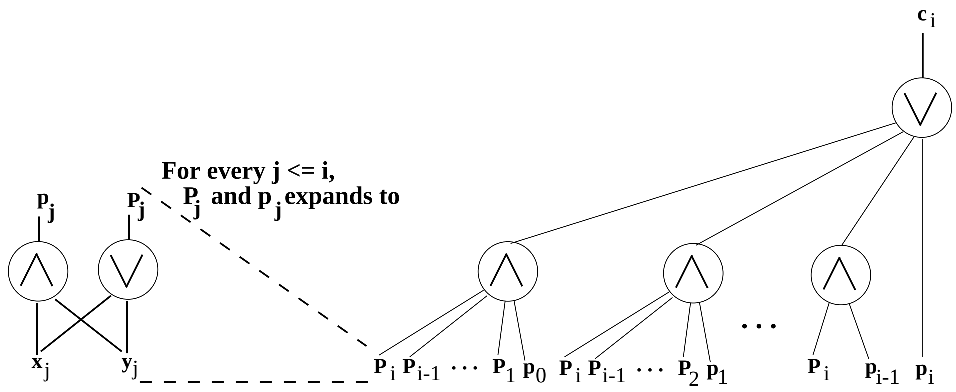

Figure 2.1: An AC^0 realization of $ADDITION$

NOTE: It has been proven that any AC^0 circuit for $ADDITION$ requires at least $O(n\log^* n)$ size.

The *carry* is the stumbling block for an efficient implementation of the $ADDITION$ circuit. Let us trace the *carry* and see how it propogates. A *carry* starts when both the bits, x_i and y_i, are 1s and continues to propogate as along as at least one of the 2 bits is 1. It dies the moment both the bits are 0s. It will restart again when both bits are 1s.

In Fig 2.2 we can see that at a given i, there will be a *carry* if at some earlier point (including i) a carry is generated. And, from that point onwards the *carry* is propogated, without getting killed, to i. This can be formally stated as,

$$
\begin{aligned}
c_i &= \exists_{j \le i}\,(x_j \ \wedge\ y_j) \ \wedge\ (\bigwedge_{k \in [i,j)} x_k \ \vee\ y_k) \\
&= \bigvee_{j \le i} (\ (x_j \ \wedge\ y_j) \ \wedge\ (\bigwedge_{k \in [i,j)} x_k \ \vee\ y_k)\ )
\end{aligned}
$$

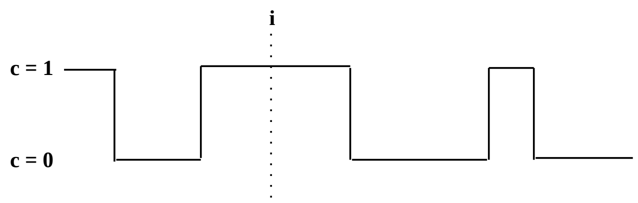

Figure 2.2: Another realization of $ADDITION$

The evaluation of c_i requires $O(n^2)$ gates and $ADDITION$ requires $O(n^3)$ gates on the whole.

Exercise: Can you reduce the size of $ADDITION$ to $O(n^2)$ or $O(n \log n)$? **Hint:** *Increase the depth.*

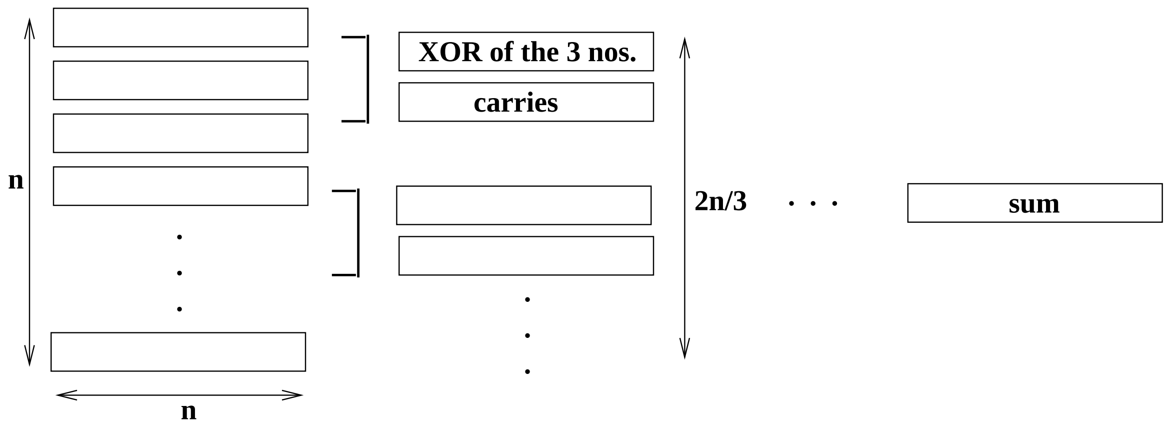

Figure 2.3: Adding n n-bit numbers

2.2.2 $MULTIPLICATION \in NC^1$

Adding n n-bit numbers is a generalization of $MULTIPLICATION$. The standard technique of forming an "addition" tree will require $\log n$ depth and $n-1$ additions. This realization would then belong to AC^0. Now, each unbounded fan-in gate expands to a $O(\log n)$ bounded fan-in realization. This would imply that adding n n-bit numbers and hence, $MULTIPLICATION$ belongs to NC^2. We want something better.

Suppose we were to partition the n numbers into groups of three. For each group, the sum is represented as two numbers. The first holds the $\oplus$ (i.e. XOR) of the three numbers and, the other holds the carries shifted by one bit to the left. This addition can be realized using NC^0 circuits. An example of this operation is shown below.

$$\left\{\begin{array}{ccccc} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{array}\right. \quad \begin{array}{cccccc} & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{array}$$

The first level of the above operation on n numbers leaves us with $2n/3$ numbers. If we keep repeating this, we get an addition tree of depth $\log_{3/2} n$ where, each level would be an NC^0 sub-circuit. Hence, we have an NC^1 realization for the addition of n n-bit numbers. This is illustrated in Fig 2.3. This technique is due to *Offman.*

We can therefore conclude that $MULTIPLICATION \in NC^1$. Can we say something more stronger that would help us to prove one of the containments stated earlier in the lecture?

2.2.3 Adding $\log n$ n-bit numbers

Suppose we are given $\log n$ n-bit numbers to add. If we are able to show that this is in AC^0, then some progress would be made towards showing that $MULTIPLICATION \in TC^0$. It is again the *carry* bits which are troublesome.

Suppose we look at the i^{th}-bits of all the $\log n$ numbers. If we add these bits, the size of the *carry* would be $\log\log n$. This would then generate n $\log\log n$-bit numbers, each shifted left by one. We can achieve our objective if there is a clever way of adding these n numbers, which possess a rather nice structure. Fig 2.4 shows the approach.

But firstly, can $\log n$ bits be added efficiently? A *truth table* for $\log n$ bits would have $2^{\log n}$ (i.e. n) rows with $\log\log n$ output bits. This *truth table* is known and, more importantly, tractable. Even if we constructed a DNF realization for each output bit, there would not be more than n minterms and hence, gates. Each of the $\log\log n$ bits can be realized by an AC^0 circuit of depth 2 with o(n) gates.

Coming to the problem of adding n $\log\log n$-bit numbers, a clever method is to "swivel" the relevant numbers about the top edge of the bounding rectangle. That is, take the diagonals and lay them out horizontally. The lowermost diagonal becomes the uppermost line and vice-versa. Note that, while laying them out the diagonals need to be shifted one place to the left. Fig 2.5 illustrates this technique.

Recursing on the resultant $\log\log n$ n-bit numbers will eventually lead to an $O(\log^* n)$ realization. But we want AC^0. The idea is to close the recursion somewhere by way of brute force.

Since there are $\log\log n$ bits in each column, the *truth table* technique can be applied successfully. In fact, we can afford to take $\frac{\log n}{\log\log n}$ columns (since, these would have $\log n$ bits and the number of rows in the *truth table* would be n).

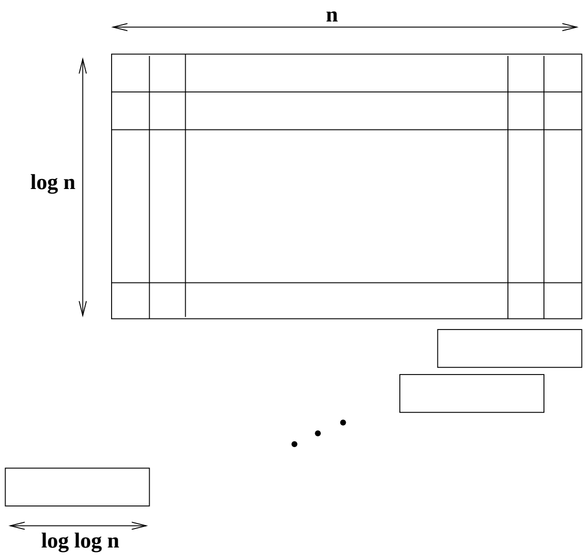

Figure 2.4: Adding $\log n$ n-bit numbers

We can see from Fig 2.6 that for each such block of $\frac{\log n}{\log\log n}$ columns there are two numbers, the *sum* and the *carry*. These are positioned such that the *carry* for a block fully overlaps with the *sum* of the next block. The next step would be to simply add a set of 2 numbers and, this we know can be done in AC^0.

Exercise: Given $\log^k n$ n-bit numbers, add them in AC^0. **Hint:** *The solution would have depth $O(k)$.*

2.2.4 *Iterated Addition*

The problem of adding $\log n$ n-bit numbers is known to be in AC^0. Now, if we can show that adding n n-bit numbers can be reduced to adding $\log n$ $O(k)$-bit numbers, we are in business. In order to reduce n n-bit numbers to $\log n$ n-bit numbers, threshold gates will be required. Assume that for every value of k, $Exact_k^n$ gates are available.

Using the $Exact_k^n$ gates, we can count the number of 1s in each column. For each value of k, the output of the $Exact_k^n$ gates are fed into a *Multiplexer* so as to generate the binary representation of the number of 1s in the column. The result is a set of n $\log n$-bit numbers. Fig 2.7 illustrates the *Iterated Addition* process.

Each of the $n \log n$ bits need to be shifted one bit to the left starting from the least significant column. But interestingly, at any bit position there are never more than $\log n$ bits to add. This addition essentially boils down to adding $\log n$ $O(n)$ numbers which we have already seen belongs to AC^0. We can therefore conclude that *Iterated Addition* $\in TC^0$.

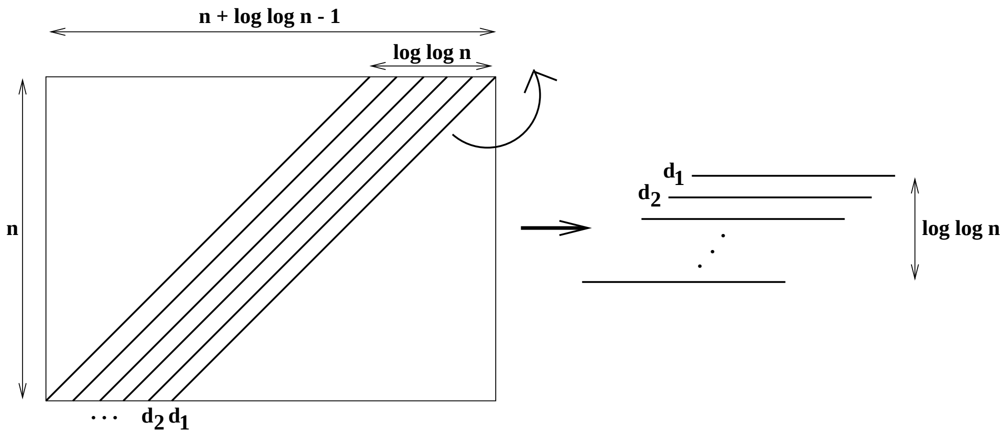

Figure 2.5: Adding $n \log\log n$-bit numbers

2.2.5 *Iterated Multiplication*

Given n n-bit numbers $a_1\ a_2\ \ldots\ a_n$, we would like to find the product of these numbers. The result could have n^2 bits. We could construct a binary tree of multiplication to compute the result. And then, using the result shown in the previous section, we can conclude that *Iterated Multiplication* $\in TC^1$. But, TC^1 is too high a class. Can we think of a better method? We shall be cover this in the next lecture.

Rejoinder: The only known strict inclusion is $AC^0 \subset TC^0$ in the inclusions sequence

$$AC^0 \subset TC^0 \subseteq NC^1 \subseteq DLOG \subseteq SymLOG \subseteq NLOG \subseteq LOGCFL \subseteq AC^1 \subseteq TC^1$$

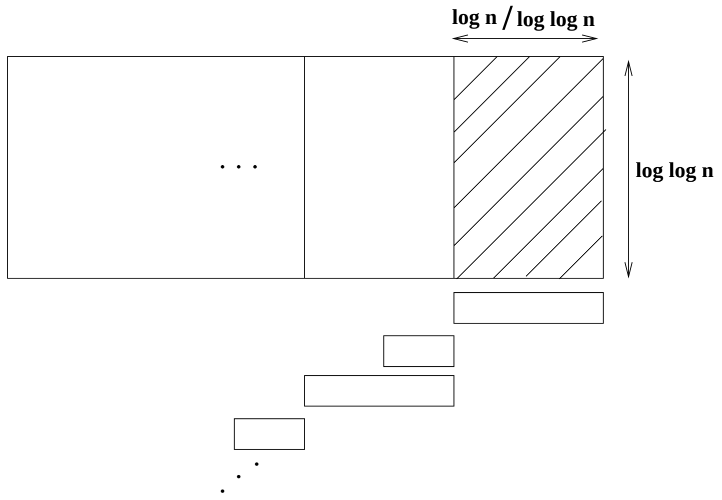

Figure 2.6: Adding $\log\log n$ n-bit numbers

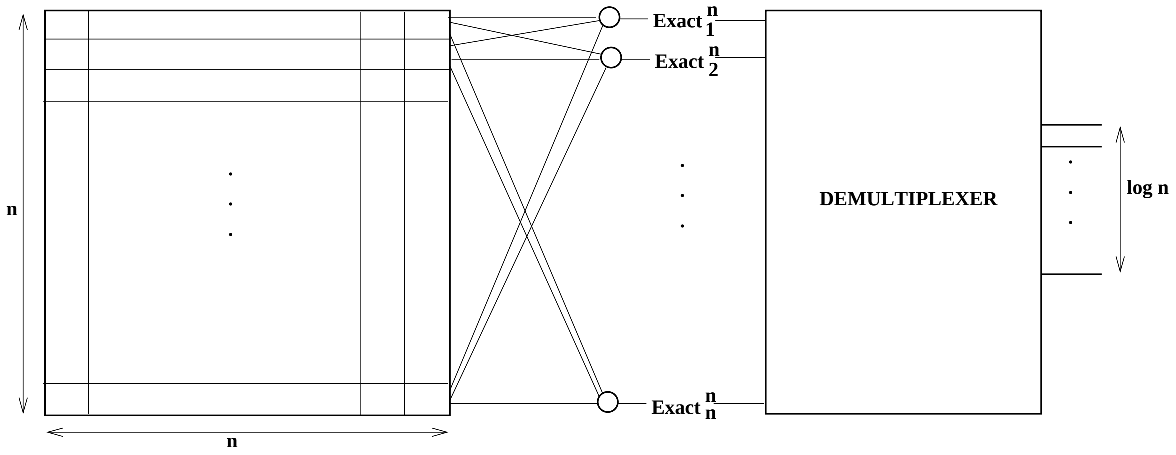

Figure 2.7: Reducing n n-bit numbers to $\log n$ n-bit numbers

Computational Complexity Theory	**Spring 1997**
Lecture 4-5 : January 28 & 30, 1997	
Lecturer: V. Vinay	*Scribe: P. R. Subramanya*

3.1 *Iterated Multiplication* $\in TC^0$

Given n n-bit numbers, $b_1, b_2, \ldots, b_n$, we are interested in computing the product of these numbers, $\prod_{i=1}^{n} b_i$. We want to show that *Iterated Multiplication* can be done in TC^0.

Chinese Remaindering is used to show this result. The proof consists of picking the first n^2 primes $\{p_1, p_2, \ldots, p_{n^2}\}$ and showing that, for each prime p :-

- Computing $b_i\ mod(p) \in TC^0$.
- Computing $\prod_{i=1}^{n} b_i\ mod(p) \in TC^0$.
- Combining the results of the previous steps and taking $mod(p_1 \cdot p_2 \cdot \ldots \cdot p_{n^2})$, is in TC^0.

The proof relies heavily on the fact that the n^{th} prime, p_n, is small, i.e. a polynomial in n.

3.2 Chinese Remaindering

Theorem 1 *Given $m_1, m_2, \ldots, m_k$ such that $gcd(m_i, m_j) = 1, \forall\ i \neq j$, and a set of residues $a_1, a_2, \ldots, a_k$, i.e. for some x,*

$$x \equiv a_1(m_1), x \equiv a_2(m_2), \ldots, x \equiv a_k(m_k),$$

there is a **unique** *solution to* $x \equiv y\ mod(\prod_i m_i)$.

Example: Let us consider the two primes 2 and 9 and generate the residues,

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
mod 2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
mod 9	0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8

An interesting thing to note is that each column of residues is *unique*. The *Chinese Remaindering Theorem* is based on this observation. What is the general technique to go from the residues to the unique number y?

Define,

$$M = \prod_{i=1}^{k} m_i \quad ; \quad M_i = \frac{M}{m_i} \quad ; \quad \hat{M}_i * M_i = 1\ mod(m_i) \quad and,$$
$$Y = \sum_{i=1}^{k} \hat{M}_i \cdot M_i \cdot a_i. \quad Then, \quad y = Y\ mod(M)$$

3.2.1 General Technique

Given n n-bit numbers, $b_1, b_2, \ldots, b_n$, the product, $x = \prod_{i=1}^{n} b_i$, can have at most n^2 bits. The idea is to select the first n^2 primes $\{p_1, p_2, \ldots, p_{n^2}\}$, so that $\prod_i p_i \geq 2^{n^2} > \prod_i b_i$. {**Note:** Why is this necessary?}

The procedure is to take a $p \in \{p_1, p_2, \ldots, p_{n^2}\}$ and, do the following for each p.

1. Evaluate $(\prod_i b_i)\ mod(p)$. This requires finding $b_1\ mod(p)$, $b_2\ mod(p)$, ... ,$b_n\ mod(p)$.
2. Since, $b_i = b_i^p\ mod(p)$, set $B'_p = \prod_i b_i^p$.
3. Set $B_p = B'_p\ mod(p)$.

We next apply *Chinese Remaindering* where, the set of co-primes is $\{p_1, p_2, \ldots, p_{n^2}\}$ and, the set of residues is $\{B_1, B_2, \ldots, B_{n^2}\}$. Now, since $x < \prod_i p_i$, the y that we get using *Chinese Remaindering* will necessarily be the x that we started off with.

For all this to work, we require each of the primes p to be small. This will ensure that the product terms evaluated above are small. Now, we know from the *Prime Number Theorem* that,

Theorem 2 [Prime Number Theorem] *Let $\pi(x)$ denote the number of primes less than or equal to x. As x increases without bound, the ratio of $\pi(x)$ to $\frac{x}{\log x}$ approaches 1.*

That is, the n^{th} prime, $p_n \leq cn \log n$ and, $p_{n^2} \leq n^3$. This means that each of the n^2 primes can be represented using $O(\log n)$ bits.

What remains to be shown is, each of the above steps can be done efficiently (i.e. $\in TC^0$).

3.2.2 $mod(p) \in TC^0$

Let us take an n-bit number A. This can be represented as, $A = \sum_{i=0}^{n-1} A_i 2^i$. That is, $A\ mod(p) = \sum_{i=0}^{n-1} A_i(2^i\ mod(p))$. This clearly can be done in TC^0 since,

- The computation of $2^i\ mod(p)$ can be *hardwired* because, p is $O(\log n)$ bits and for every i, we can directly maintain the residue in a table without too much expense. For example, if $p = 3$, the table looks like,

i	0	1	2	3	4	5	6	7	8	9	...
2^i	1	2	4	8	16	32	64	128	256	512	...
$mod(3)$	1	2	1	2	1	2	1	2	1	2	...

  The residues are $\log p$ bits.

- $A_i \in \{0, 1\}$ and, finding the n product terms in the summation is a *depth* 1 circuit. Each product term is necessarily $O(\log n)$ bits.

- Lastly, adding n $\log n$-bit numbers is known to $\in$ TC^0. This was shown in the previous lecture.

This implies that computing $b_i\ mod(p)\ \ \forall i, p$ can be done is TC^0.

3.2.3 Computing $B_p \in TC^0$

It is known that for any prime, p, the prime field $\mathbb{Z}_p$ gives a cyclic group, $\mathbb{Z}_p^*$, under multiplication. That is, there exists a unique g such that $1, g, g^2, g^3, \ldots, g^{p-2}$ generates all of $\{1, 2, \ldots, p-1\}$. This means that we can maintain an *element-exponent* table of elements and the corresponding exponents that generate it. That is, for each $i \in \{1, 2, \ldots, p-1\}$ we store the exponent of g that generates it. As there are a polynomial number of exponents in the table, this can be hardwired.

Using the table, each product term can now be equivalently transformed to contain factors of the form g^j, for some $0 \leq j \leq (p-i)$, instead of b_i^ps. Since, all the factors in a product term are powers of the same g, multiplication now becomes an addition of the exponents of g. And since, each exponent is $\log n$ bits, the addition can be done in TC^0.

Note that, the sum of the exponents could exceed p. But, it is known that $a = a^p mod(p)$, and therefore, we need to do $mod(p)$ on the sum of the exponents. This again can be done in TC^0.

At this stage, we have computed $B_p' = \prod_i b_i^p$ in the form of g^j, where $0 \leq j \leq (p-1)$. To revert back to the element $a \in \{1, 2, \ldots, p-1\}$, we use the *element-exponent* table, as above, except that we now go from the exponent to the residue. This again can be done in TC^0.

3.2.4 Winding up *Iterated Multiplication* $\in TC^0$

The next step is to compute $Y = \sum_{p \in \{p_1, p_2, \ldots, p_{n^2}\}} P_p \cdot \hat{P}_p \cdot B_p$. For a given n, P_p and $\hat{P}_p$ are known beforehand and, independent of the inputs. These can therefore be hardwired. Each term in the summation (i.e. $P_p * \hat{P}_p * B_p$) requires two levels of multiplication and hence, computing these terms is in TC^0.

The summation of the terms to get Y can be done through *Iterated Addition*, which we already know is in TC^0. The problem is that Y could be *greater than* $\prod_i p_i$. This could happen because each term in the summation can be at most $\prod_i p_i$. Therefore, $0 < Y < n^2(\prod_i p_i)$.

Using *Iterated Subtraction*, we can get $y = Y mod(\prod_i p_i)$. Basically, in parallel try out all possible j for which $Y - j\prod_i p_i$ yields a number between 0 and $\prod_i p_i$. The number of js that we need to try is small (at most n^2). Hence, this too can be realized in TC^0.

Exercise: Show that the number of js to be tried for the *Iterated Subtraction* above is indeed *small.*

3.3 A bit of Number Theory

$\pi(n)$ denotes the number of *prime numbers* less than n. p_n denotes the n^{th} prime. In this section, we are interested in the following questions,

- What can we say about $\pi(n)$? Can we give a lower bound for $\pi(n)$?
- What can we say about p_n, the n^{th} prime?
- Does there necessarily exist a prime between n and $2n$?

The *Prime Number Theorem* can be restated as

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\log n} = 1$$

In 1792, *Gauss* first stated the theorem in a slightly different form. *Chebyschev* showed that

$$\frac{c_1 x}{\log x} \leq \pi(x) \leq \frac{c_2 x}{\log x}$$

where $c_1 \approx 0.95$ and $c_2 \approx 1.2$. *Hadamard* and *de la Vallée Poussin* independently furnished a proof in 1896.

Both used the *Riemann-Zeta function* to prove the *Prime Number Theorem*. The *Riemann-Zeta function* is,

$$\zeta(s) = \sum_{n\geq 1} \frac{1}{n^s}$$

where $s = \sigma + it$. *Riemann* conjectured that all the *zero*s of the function lie on the line $\sigma = 1/2$. *Hadamard*'s and *de la Vallée Poussin*'s proofs required to show that there are **no zeros** in a small region near $\sigma = 1$. Fig 3.1 illustrates the *Riemann-Zeta function*.

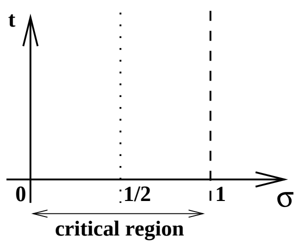

Figure 3.1: Riemann-Zeta function $\zeta(s)$

Selberg and *Erdós* independently proved the *Prime Number Theorem* using elementary techniques.

3.3.1 The Number of Primes is Infinite

Proof by Contradiction is the standard high school way of proving that the number of primes is infinite. Suppose there are a finite number of primes, $\mathbb{P} = \{p_1, p_2, \ldots, p_k\}$, for some fixed k. Every number should be divisible by at least one of the prime numbers from $\mathbb{P}$. Consider, the number $(\prod_{j=1}^{k} p_j) + 1$, which is not in $\mathbb{P}$. This is not divisible by any of the prime numbers in $\mathbb{P}$. Therefore, it too must be prime. But then, this is a contradiction.

Euler presented a neat proof for this. Consider,

$$(\sum_k \frac{1}{2^k})(\sum_k \frac{1}{3^k})(\sum_k \frac{1}{5^k})\cdots = \sum_{n\geq 1} \frac{1}{n}$$

This is true because every number n is factorizable into a set of prime factors. The product of the summations in the *lhs* generates every possible prime factorization.

The sum of the infinite geometric progression is given by, $\sum_k \frac{1}{p^k} = \frac{1}{1-1/p}$. Rewriting this, we have

$$\sum_{n\geq 1} \frac{1}{n} = \prod_{p\ is\ prime} (\frac{1}{1-1/p})$$

Now, if there are a finite number of primes, then the *rhs* is finite. But, the *lhs* definitely diverges. Hence, the number of primes must be infinite.

3.3.2 Erdós' Proof

The innocuous looking $\binom{2n}{n}$ has some interesting properties.

$$\binom{2n}{n} = \frac{2n!}{(n!)^2} = \frac{(n+1)\cdot(n+2)\cdot\ldots\cdot 2n}{1\cdot 2\cdot\ldots\cdot n} = \frac{n+1}{1}\cdot\frac{n+2}{2}\cdot\ldots\cdot\frac{2n}{n} \geq 2^n$$

As $\binom{2n}{n}$ is the largest term of all terms $\binom{2n}{k}$, we can write,

$$(1+1)^{2n} = \sum_k \binom{2n}{k} \Rightarrow 4^n \leq (2n+1)\binom{2n}{n} \Rightarrow \frac{4^n}{2n+1} \leq \binom{2n}{n} \leq 4^n$$

The primes in the range $[n, 2n]$ will survive in $\binom{2n}{n}$ since these cannot be cancelled. This leads to,

$$\Rightarrow \prod_{n<p\leq 2n} p \leq \binom{2n}{n} \leq 4^n$$

Changing the range to $[n/2, n]$, $[n/4, n/2]$ and so on, we have,

$$\Rightarrow \prod_{n/2<p\leq n} p \leq 4^{n/2} \quad , \qquad \prod_{n/4<p\leq n/2} p \leq 4^{n/4} \quad , \qquad \ldots$$

$$\Rightarrow \prod_{1<p\leq n} p \leq 4^{n/2+n/4+\ldots} \quad \leq 4^n$$

Taking log ,

$$\sum_{1\leq p\leq n} \log p \leq n \log 4 \Rightarrow \sum_{\sqrt{n}\leq p\leq n} \log p \leq n \log 4$$

$$\Rightarrow \frac{1}{2} \cdot \log n \cdot \sum_{\sqrt{n}\leq p\leq n} 1 = \frac{1}{2} \cdot \log n \cdot (\pi(n) - \pi(\sqrt{n})) \leq n \log 4$$

Adjusting the terms and, because $\pi(\sqrt{n})$ is at most $\sqrt{n}$, we get

$$\Rightarrow \pi(n) - \pi(\sqrt{n}) \leq \frac{2n\log 4}{\log n} \Rightarrow \pi(n) \leq \frac{2n\log 4}{\log n} + \pi(\sqrt{n}) \leq \frac{2n\log 4}{\log n} + \sqrt{n}$$

Now, $\sqrt{n}$ is very small when compared to $\frac{2n\log 4}{\log n}$. That is,

$$\pi(n) \leq \frac{3n\log 4}{\log n} \leq \frac{6n}{\log n}$$

This gives an upper bound for $\pi(n)$. We have been rather sloppy with the approximations.

3.3.3 A Lower Bound for $\pi(n)$

This lower bound proof is due to [13]. Consider the integral, I, defined as,

$$I = \int_0^1 x^n \cdot (1-x)^n dx = \int_0^1 x^n \cdot \sum_{k=0}^{n} (-1)^k \cdot \binom{n}{k} \cdot x^k dx = \sum_{k=0}^{n} (-1)^k \cdot \binom{n}{k} \cdot \int_0^1 x^{n+k} dx$$

$$= \sum_{k=0}^{n} (-1)^k \cdot \binom{n}{k} \cdot \frac{1}{n+k+1} = \frac{\cdots}{n+1} + \frac{\cdots}{n+2} + \ldots + \frac{\cdots}{2n+1} = \frac{P}{d_{2n+1}} \quad \textit{for some } P$$

d_n is the *LCM* of $\{1, 2, \ldots, n\}$. It follows that, $d_{2n+1} \cdot I \geq 1$. Since $x \cdot (1-x) \leq 1/4$, it implies,

$$\Rightarrow |I| \leq \frac{1}{4^n} \Rightarrow d_{2n+1} \geq 4^n$$

If some prime power divides d_{2n+1}, then there must exist a number m between 1 and $2n+1$ which is also divided by the prime power. That is,

$$If \quad p^\alpha \| d_{2n+1} \quad \Rightarrow \exists m \ \ (1 \leq m \leq 2n+1) \quad s.t.\ p^\alpha \| m \quad \Rightarrow p^\alpha \leq m \leq 2n+1$$

This essentially means that *no prime power can contribute too much.*

$$\Rightarrow 4^n \leq d_{2n+1} = \prod_p p^\alpha \leq (2n+1)^{\pi(2n+1)} \Rightarrow n\log 4 \leq \pi(2n+1)\cdot \log 2n+1$$

$$\Rightarrow \pi(2n+1) \geq \frac{n\log 4}{\log 2n+1}. \quad Simplifying,\ \ \pi(n) \geq \frac{1}{3}\cdot\frac{n}{\log n}$$

Considering a slightly different integral, $\hat{I}$, and doing *integration by parts*, we get

$$\hat{I} = \int_0^1 x^{n-1}\cdot(1-x)^n dx = \left[x^{n-1}\cdot(-\frac{(1-x)^{n+1}}{n+1})\right]_0^1 - \int_0^1 (n-1)\cdot x^{n-2}\cdot(-\frac{(1-x)^{n+1}}{n+1})$$

$$= 0 + \left[\frac{n-1}{n+1}\cdot x^{n-2}\cdot(-\frac{(1-x)^{n+2}}{n+2})\right]_0^1 - \frac{n-1}{n+1}\cdot\int_0^1 (n-2)\cdot x^{n-3}\cdot(-\frac{(1-x)^{n+2}}{n+2}) = \ldots$$

$$= \frac{n-1}{n+1}\cdot\frac{n-2}{n+2}\cdot\ldots\cdot\frac{1}{2n-1}\cdot\frac{1}{2n} = \frac{1}{n\begin{pmatrix}2n\\n\end{pmatrix}}$$

$\hat{I}$ can be evaluated in the same way as I to get,

$$\hat{I} = \int_0^1 x^{n-1}\cdot(1-x)^n dx = \sum_{k=0}^{n}(-1)^k\cdot\begin{pmatrix}n\\k\end{pmatrix}\cdot\frac{1}{n+k} \quad \Rightarrow \hat{I}\cdot d_{2n}\in\mathbb{N}$$

$$\Rightarrow n\cdot\begin{pmatrix}2n\\n\end{pmatrix}\| d_{2n} \quad \Rightarrow n\cdot\begin{pmatrix}2n\\n\end{pmatrix}\| d_{2n+1}$$

If I were to be *integrated by parts* as was done for $\hat{I}$, we would be led to the fact that $(2n+1)\cdot \begin{pmatrix}2n\\n\end{pmatrix}$ also divides d_{2n+1}. But $(2n+1)$ and n are co-primes. Hence,

$$n\cdot(2n+1)\cdot\begin{pmatrix}2n\\n\end{pmatrix}\| d_{2n+1} \quad \Rightarrow d_{2n+1} \geq n\cdot(2n+1)\cdot\begin{pmatrix}2n\\n\end{pmatrix} \geq n\cdot 4^n \geq 2^{2n+1}$$

That is, $d_n \geq 2^n$ and therefore, $2^n \leq d_n = \prod_p p^\alpha \leq n^{\pi(n)}$. Taking log s, we can conclude that, $\boxed{\pi(n) \geq \frac{n\log 2}{\log n}}$

Let us look at the primes till $4n\log n$.

$$\pi(4n\log n) \geq \frac{4n\cdot\log n\cdot\log 2}{\log 4+\log n+\log\log n} \geq \frac{4n\cdot\log n\cdot\log 2}{2\log n} \geq 2n\log 2 \geq n$$

That is, the n^{th} prime, p_n is as high as $\boxed{p_n \leq 4n\log n}$.

3.3.4 Is there a prime in $[n, 2n]$

A proof for the existence of a prime in the range $[n, 2n]$ using *Erdós*' method above is discussed below. Let us consider $\binom{2n}{n} = \frac{2n!}{(n!)^2}$ again. $n!$ can be represented as a product of prime factors. That is, $n! = \prod_p p^{\alpha}$. Let us consider a single prime and see what its exponent is likely to be.

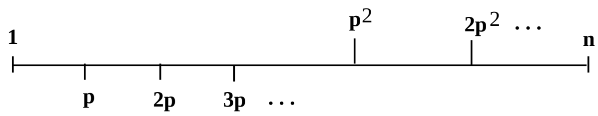

Figure 3.2: The $[1, n]$ line

As is evident from Fig 3.2, the exponent of p is given by $\alpha = \lfloor \frac{n}{p} \rfloor + \lfloor \frac{n}{p^2} \rfloor + \ldots$. The contributions to α comes from $\log_p n$ terms.

For a prime p that occurs in $\binom{2n}{n}$, its exponent is

$$\lfloor \frac{2n}{p} \rfloor + \lfloor \frac{2n}{p^2} \rfloor + \ldots - 2 \cdot \lfloor \frac{n}{p} \rfloor - 2 \cdot \lfloor \frac{n}{p^2} \rfloor - \ldots$$

Since $0 \leq \lfloor 2x \rfloor - 2 \cdot \lfloor x \rfloor \leq 1$, the contribution to the exponent of p by each pair will be at most 1. This means that the exponent α for a prime p can be at most $\log_p 2n$. That is, for any prime p in $\binom{2n}{n}$, $p^{\alpha} \leq 2n$. In other words, *the contribution of each prime is small.*

Consider primes in the range $(2n/3, n)$. These will not contribute to $\prod_p p^{\alpha}$ since $\lfloor \frac{2n}{p} \rfloor$ is 2 and $2 \cdot \lfloor \frac{n}{p} \rfloor$ is also 2. Fig 3.3 shows the contributions of the primes.

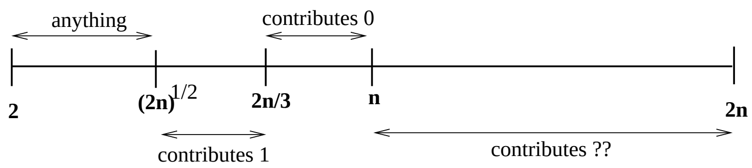

Figure 3.3: The contributions by primes

Suppose there are no primes in the range $[n, 2n]$. The contribution from the range $[\sqrt{2n}, 2n/3]$

will be at most $4^{2n/3}$ and, since p^α can be at most $2n$, we have

$$\begin{aligned} \binom{2n}{n} &= \prod_p p^\alpha \leq 4^{2n/3} \cdot (2n)^{\sqrt{2n}} \\ \Rightarrow \binom{2n}{n} &\leq 4^{2n/3} \cdot (2n)^{\sqrt{2n}} < \tfrac{4^n}{4n} < \binom{2n}{n} \end{aligned}$$

This is a contradiction. Hence, there must be prime in the range $[n, 2n]$.

Computational Complexity Theory	**Spring 1997**
Lecture 6-8 : February 11, 13 & 14, 1997	
Lecturer: V. Vinay	*Scribe: P. R. Subramanya*

In the next few lectures a broad overview of *Complexity Theory* shall be presented.

4.1 Broad Overview of Complexity Theory

Given an alphabet Σ, a language L is a subset of the set of strings over Σ^*, i.e. $L \subseteq \Sigma^*$. A *decision* problem is one where one asks the question 'Does $x \in L$?'. Similarly, an *existence* problem is one where one asks 'Is there an $x \in L$?'. These are problems that have an answer *Yes/No*. We shall see some other problems that have a different type of answer.

4.1.1 Non-determinism

Let us explain the concept through a game. A professor in a respected university has a number of students whom he/she advises. To each student the professor gives a problem P. The professor can afford to give each student 1 hour/week. That is, the student has 1 hour/week to convince the professor about a *decision* or *existence* proof. The amount of time spent in arriving at the proof is not of relevance. What is important is whether the professor can verify the correctness of the proof presented by the student in a fixed amount of time.

Suppose the professor asks the student the following question. Is the number N a compositie number? The student might give a proof for compositeness of N as, $N = p \cdot q$. The professor needs to verify that N indeed is the product of the two numbers p and q. This is rather easy. $N = p \cdot q$ that the student furnishes is a *witness/certificate* for the problem instance. The length of the certificate should be polynomial in the length of the input since the validity of the certificate should be checkable in polynomial time.

In order to show that N is not composite (i.e. N is *prime*), the student needs to provide a *witness/certificate*. What would be a good witness? The naive technique of showing that none of the numbers $2, 3, \ldots, \sqrt{N}$ divides N, would be exponential in the length of the input (viz $\log N$).

Let us consider another example.
Instance: G is a graph (encoded in a *reasonable* manner).
Question: G has a *Hamiltonian Cycle* (i.e. every vertex of G is visited exactly once)?

The *witness* here is a cycle C. One needs to verify whether

- C is indeed a cycle.
- all edges in C do indeed exist in G.
- all the vertices in G are included in C.

Each of these 3 steps can be done in polynomial time. If the input is provided as an *adjacency list*, the input length is n^2. If the witness is encoded as the end-points of the edges in the cycle, the witness length is $2n \log n$.

4.1.2 $\mathcal{NP}$: Non-deterministic Polynomial Time

$\mathcal{P}$ is the class of problems that can be solved by a deterministic Turing machine in polynomial time.

Definition 4 *Let $L \subseteq \Sigma^*$. $L \in \mathcal{NP}$ if $\exists B \in \mathcal{P}$, $\exists p \cdot p$ is a polynomial, and $x \in L \Rightarrow \exists y \cdot |y| \leq p(x)$ and $< x, y > \in B$. x here is the* specific problem, *y is the* witness/certificate *and the* existence *problem $< x, y > \in B$ is the* verification/validation *process.*

There is an assymmetry in the notion of $\mathcal{NP}$. If $x \notin L$ there should be no witness to the contrary. This is identical to what in legal parlance one calls *innocent unless proven guilty.* That is, proving a person's innocence is too difficult since all witnesses should fail.

This assymmetry leads one to the complementary class, *co*$\mathcal{NP}$, for $\mathcal{NP}$. That is, if $L \in \mathcal{NP}$ then $\bar{L} \in co\mathcal{NP}$ where $\bar{L} = \{x : x \notin L\}$. Note that, $\mathcal{P} = co\mathcal{P}$ since one gets an answer *Yes/No* in polynomial time. The relation between the classes $\mathcal{P}$, $\mathcal{NP}$ and *co*$\mathcal{NP}$ is shown in Fig 4.1.

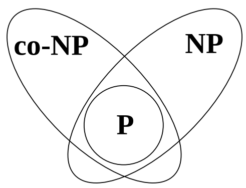

Figure 4.1: Relation among $\mathcal{P}$, $\mathcal{NP}$ and $co\mathcal{NP}$

4.1.3 $\sharp P$ and $\mathcal{NLOG}$

We have indicated that in *existence* problems one is interested in the question, *are there* ≥ 1 *solutions?* One could ask the question, *how many solutions are there?* Note that, this question expects a numerical answer, i.e., given an input one has to output the number of solutions. Such problems fall in the *function classes* called *counting classes.*

Let us consider the class $\sharp\mathcal{P}$.

Definition 5 *A function $f : \Sigma^* \to \mathbb{N}$, $f \in \sharp\mathcal{P}$ if $\exists B \in \mathcal{P}$ such that $f(x) = |\{y :< x, y >\in B\}|$.*

The *permanent* of a matrix A is defined as $per(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$. Note that, the *determinant* of a matrix is defined very similarly as $\det(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} (-1)^{sgn(\sigma)} \cdot a_{i,\sigma(i)}$. Though these two problems are very closely related, they are of contrasting difficulty. The $det(A)$ problem is known to be in $\mathcal{NLOG}$. The problem, *is* $per(A) > 0$*?* is known to be in $\mathcal{P}$. *Valiant* in a seminal work showed that $per(A)$ is $\sharp\mathcal{P}$-Complete. The notion of completeness is discussed in a later section.

Let us consider another problem, $st - connectivity$. One is given a DAG with a source s and a sink t. The question asked is, *is there a path from s to t*. In other words, are s and t connected. How would one solve this?

One stores the current node v and the sink node t at any instant. At the start, $v = s$. One asks the question, *are v and t connected?* From the current node, one non-deterministically goes to a neighbour of v. If the neighbour is not the sink node t, set v to this neighbour. And ask again whether v is connected to t.

If s and t are connected, one would find it since some computation would have ended and selected a path from s to t. The resources used by the algorithm presented above requires one to store information about 2 nodes only. That is, $2\log n$ bits would be sufficient for the computation to be completed. One puts no constraint on the amount of time that the algorithm should take. The $st - connectivity$ problem is in the class $\mathcal{NLOG}$ since a non-deterministic Turing machine with $O(\log n)$ space can decide whether s and t are connected.

Similarly, one has the class $\mathcal{DLOG}$ and $\sharp\mathcal{LOG}$. These are deterministic and counting versions of $LOGSPACE$ problems. It is known that evaluating $det(A)$ is exactly equivalent to counting the number of paths for s to t in some graph G.

4.2 Primality Testing

In this section we shall slightly digress and talk about *certificates* that are not necessarrily solutions to the problem in hand. One saw the following question being posed:

Question:	Given N, is it a composite number?
Certificate:	a, b such that $N = a \cdot b$.

Can one think of a certificate that does not tell us how to factorize N?

One shall make use of *Fermat's Little Theorem* which states that if $(a, p) = 1 \Rightarrow a^{p-1} \equiv 1\ mod(p)$. *Fermat's Little Theorem* follows from the fact that $(\mathbb{Z}_p, *)$ is a cyclic group of order $p - 1$. Another way to prove the theorem is to look at $(x + y)^p \equiv (x^p + y^p) mod(p)$ and use induction. Now, $1^{p-1} \equiv 1 mod(p)$.

$$\text{Assume, } a^{p-1} \equiv 1 \ mod(p). \Rightarrow a^p \equiv a \ mod(p)$$
$$\Rightarrow (a+1)^p \equiv (a^p + 1^p) mod(p) = (a+1) mod(p) \Rightarrow (a+1)^{p-1} \equiv 1 mod(p)$$

Consider the field $\mathbb{Z}_p$. There exists an element g in $\mathbb{Z}_p$ which is the *generator*. g has an order of $p-1$. Now, given a number N, one defines $\mathbb{Z}_N = \{x : (x, N) = 1\} \Rightarrow |\mathbb{Z}_N| = \phi(N)$ and $\phi(N) < (N-1)$. If N is a composite number then $\forall a$ where $(a, N) = 1$, $order(a) < (N-1)$.

Now, if one were told that $\exists a \ (a, p) = 1$ and if one had to show that a is a *generator* one would do the following:

(1) $a^{p-1} \equiv 1 \ mod(p)$

(2) $a^{\frac{p-1}{q}} \not\equiv 1 \ mod(p) \ \forall q \in \{q_1, q_2, \ldots, q_k\}$ and $(p-1) = \prod q_i^{\alpha_i}$.

Note that this works **if and only if** p is a prime.

Let us look at a non-deterministic algorithm to detect whether a number is a prime or not. Given the number p, the algorithm is

1) Guess a factorization $\{q_1, q_2, \ldots, q_k\}$ for $(p-1)$.
2) Guess an a and show that it is a generator.
3) Recurse and show that the q_is are also primes.

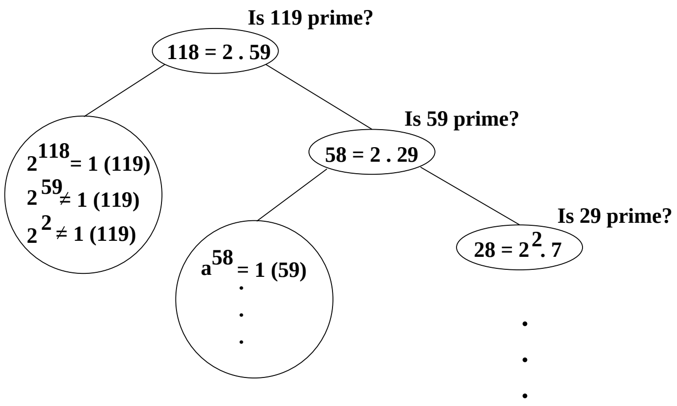

Figure 4.2: Example for Primality Testing

Consider as an example, $N = 113$. Fig 4.2 illustrates the primality testing algorithm.

4.2.1 Analysis of Primality Testing

The recursion depth of the algorithm described above is bounded by $\log N$, where N is the given number. This is so because at each level of recursion, one recurses to the next level for a number that is at most 1/2 the current number. Furthermore, at any level of recursion, the primality question is asked for the prime factors of the parent number. The product of these can never be greater than N. And since each of the numbers is at least 2, the number of nodes at any level of recursion is bounded by $\log N$.

The size of the computation tree for primality testing is bounded by $\log^2 N$. The size of the circuit would be $\log^3 N$ and the verification time is $\log^4 N$.

The above scheme is known as *Pratt*'s certificates. The standard measure for primality testing schemes is number of multiplications. *Pratt*'s scheme requires $\log^2 N$ multiplications. *Pomerance* reduced the number of multiplications to $\frac{5}{2}\log N$ using *elliptic curves.* And there is scope for improvement.

4.3 $\mathcal{RP}, \mathcal{PP}$ and $\mathcal{BPP}$

Suppose the size of *certificates* for a problem is bounded by a polynomial p, i.e. $|certificate| \leq p(n)$. The total number of possible *certificates* is $2^{p(n)}$. If the number of *certificates* that lead to acceptance is $\geq 2^{p(n)-1}$, then if a *certificate* is picked at random, with probability 1/2 it will indeed be a *certificate.* The class of problems for which this scheme holds is called $\mathcal{RP}$.

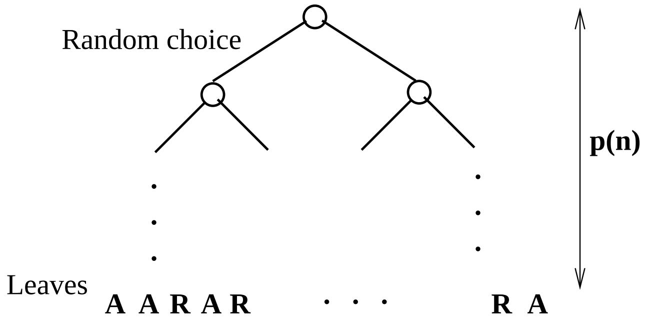

Figure 4.3: Computation Tree for Randomized Classes

Definition 6 *A language $L \in \mathcal{RP}$, Randomized Polynomial Time, if there exists a $B \in \mathcal{P}$ and a polynomial p such that,*
$x \in L \Rightarrow |\{y :< x, y >\in B\}| \geq 2^{p(n)-1}$ *and*
$x \notin L \Rightarrow |\{y :< x, y >\in B\}| = 0$

The difference between $\mathcal{NP}$ and $\mathcal{RP}$ lies in the fact that the latter requires at least 1/2 the leaves in the computation tree to be accepting. In fact, if the density of certifcates is an inverse polynomial $\frac{1}{n^\alpha}$, one can use *amplification* to get any desired ratio. For example, if we repeat the experiment $n^{2\alpha}$ times, the probability of error is $(1 - \frac{1}{n^\alpha})^{2\alpha}$ which is e^{-2}.

Definition 7 *A language $L \in \mathcal{BPP}$, Bounded-error Probabilistic Polynomial Time, if there exists a $B \in \mathcal{P}$ and a polynomial p, such that,*

$x \in L \Rightarrow \frac{|\{y:<x,y>\in B\}|}{2^{p(n)}} \geq 3/4$ *and*
$x \notin L \Rightarrow \frac{|\{y:<x,y>\in B\}|}{2^{p(n)}} \leq 1/4$

$\mathcal{BPP}$ is more of a *democratic* class. The probabilities are bounded away from 1/2 and hence the name. Furthermore, this a symmetric class.

Definition 8 *A language $L \in \mathcal{PP}$, Probabilistic Polynomial Time, if there exists a $B \in \mathcal{P}$ and a polynomial p, such that,*
$x \in L \Rightarrow \frac{|\{y:<x,y>\in B\}|}{2^{p(n)}} \geq 1/2$ *and*
$x \notin L \Rightarrow \frac{|\{y:<x,y>\in B\}|}{2^{p(n)}} < 1/2$

$\mathcal{PP}$ is not a natural class essentially because it is not amplifiable.

As long as there is a meaningful notion of *certificate/proof* in a class $\mathcal{C}$, one can define the classes $\mathcal{NC}, \mathcal{RC}, \mathcal{BPC}$ and $\mathcal{PC}$. Fig 4.4 shows the complexity hierarchy for the variants of the class $\mathcal{P}$.

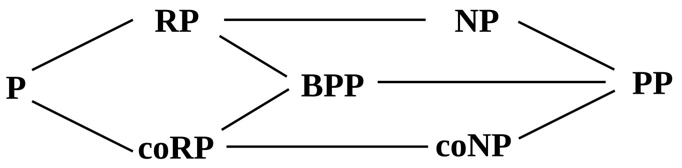

Figure 4.4: Relation among $\mathcal{P}, \mathcal{RP}, \mathcal{NP}, \mathcal{BPP}$ and $\mathcal{PP}$.

4.4 Notion of Completeness

Consider a complexity class $\mathcal{C}$. It encompasses a number of problems of varying difficulty or hardness. The *hardest* problem in the class stretches the resources available to $\mathcal{C}$ to the limit. If one could solve such problems *easily* then the entire class would be easy. Hence, it makes sense to identify the most difficult problems in a class.

The most difficult problems in a class are called the **complete** problems for the class. These have the property that all other problems in the class can be *reduced* to the *complete* problem. By reduction, one means a mapping from instances of a problem to instances of some other problem.

Definition 9 *Let f be a polynomial time computable many-one function. f reduces L_1 to L_2* **iff** $x \in L_1 \iff f(x) \in L_2$. *This is written as $L_1 \leq_m^P L_2$.*

Assume a language $L \in \mathcal{NP}$ such that $\forall \hat{L} \in \mathcal{NP}$, $\hat{L} \leq_m L$. Now if $L \in \mathcal{P}$ then $\mathcal{NP} \subseteq \mathcal{P}$. This would mean that $\mathcal{P} = \mathcal{NP}$ settling a long standing open problem.

Now, let us look at $\mathcal{P}$ and $\mathcal{NP}$ with the view of trying to separate them. One would like to show that some problem from $\mathcal{NP}$ is unlikely to be in $\mathcal{P}$. And there is the other contrasting goal of showing that a *complete* problem can be solved *easily*. Both these goals require that the resources used in the reduction of problems be small, i.e. in $\mathcal{P}$.

4.4.1 Showing something is $\mathcal{NP}$-Complete

An $\mathcal{NP}$-Complete problem by definition is a problem in $\mathcal{NP}$ to which all other problems in the class can be reduced to. Now, suppose one wants to show that a problem G is $\mathcal{NP}$-Complete. Suppose L is a known $\mathcal{NP}$-Complete problem. What one does is
(a) Show that $G \in \mathcal{NP}$. This automatically means that $G \leq_m^P L$.
(b) Show that $L \leq_m^P G$.

An important point to note is the notion of *transitivity/composability* of reductions (i.e. functions). $\forall \hat{L}\ \ \hat{L} \xrightarrow{f_1} L \xrightarrow{f_2} G$. Does it necessarily follow that $\forall \hat{L}\ \ \hat{L} \xrightarrow{f_1 \circ f_2} G$? This is true in the case of polynomial time computable reductions. We shall see later where this is not so obvious.

4.4.2 $\mathcal{P}$-Completeness and $\mathcal{LOGSPACE}$

The notion of completeness can be defined for most classes except the unnatural ones like $\mathcal{RP}$, $\mathcal{BPP}$, etc. Let us consider $\mathcal{P}$ and $\mathcal{DLOG}$. Analogous to $\mathcal{NP}$-Complete problems, questions pertaining to $\mathcal{P}$-Complete problems and $\mathcal{DLOG}$ are very relevant.

Languages in the class $\mathcal{DLOG}$ use $O(\log N)$ space for deciding whether a string is in the language or not. This means that the number of different configurations (essentially, states) that a $\mathcal{DLOG}$ machine has is at most $n^{O(k)}$, i.e. a polynomial. If one were to simulate a $\mathcal{DLOG}$ machine using a deterministic Turing machine, one can run through all the states and decide whether a string is in the language or not, in $n^k + 1$ steps at most. Hence, $\mathcal{DLOG} \subseteq \mathcal{P}$.

Once it is clear that $\mathcal{DLOG}$ is contained in $\mathcal{P}$, one would like to know categorically whether $\mathcal{P} \subseteq \mathcal{DLOG}$ or not. Hence, one looks at the most difficult problems in $\mathcal{P}$ and attempts to look for an easy solution within the resource constraints of the lower class $\mathcal{DLOG}$.

There are two classes of problems that could arise.
(1) $\forall \hat{L} \in \mathcal{P}\ \ \hat{L} \leq_m^{\log} L$.
(2) $L \in \mathcal{P}$.

If (1) alone holds, then L is $\mathcal{P}$-Hard. If (1) and (2) hold, L is $\mathcal{P}$-Complete. Showing a problem to be $\mathcal{P}$-Complete proceeds in exactly the same way as showing that a prpblem is $\mathcal{NP}$-Complete. What is not obvious here is the question of *composability* of reductions. That is, are $LOGSPACE$ reductions preserved?

One could have the following scenario

$$|x| = n \quad \xrightarrow{f_1} \quad |f_1(x)| = n^2 \quad \xrightarrow{f_2} \quad |f_2(x)| = n^4$$

Since one is talking about $LOGSPACE$ reductions, the naive scheme of composing f_1 and f_2 does not work. What one needs to do is the following:

(0) f_2 starts off first and asks f_1 for a bit whenever it wants one.
(1) f_2 writes the required bit's position on the tape and hands over control to f_1.
(2) f_1 starts its computation keeping a count of the bits that it generates.
(3) f_1 discards its output till the desired output bit, and hands over control to f_2.
(4) f_2 continues its computation till it requires another bit. Then the process repeats.

The above simulation can be done in $LOGSPACE$ provided f_1 and f_2 can. If the lengths of the outputs of reductions increases quadratically, one can show that any number of $LOGSPACE$ reductions can be combined.

4.4.3 Some other types of reductions

It has been proven that AC^0 reductions are sufficient for $\mathcal{NP}$-Complete problems. Another type of reduction is *projection.* $L_1 \leq_m^{proj} L_2$ if the following holds. Given

$$\begin{array}{l} x_1 x_2 \ldots x_n \in L_1 \\ y_1 y_2 \ldots y_m \in L_2 \end{array} \quad \text{each } y_i = \begin{cases} x_j \\ \neg x_j \\ 0 \\ 1 \end{cases}$$

It is conjectured that every known $\mathcal{NP}$-Complete is complete under projections. The *isomorphism* conjecture by *Berman & Hartmanis* states that an *one-to-one onto polynomial time invertible polynomial time computable map* exists between any two $\mathcal{NP}$-Complete problems.

Computational Complexity Theory **Spring 1997**

Lecture 9 : February 18, 1997

Lecturer: V. Vinay *Scribe: P. R. Subramanya*

In this lecture, we shall firstly prepare the necessary background for *Allender*'s talk. The talk itself shall be summarized subsequently.

5.1 Arithmetic Circuits

Let us look at the circuit characterization of $\mathcal{NP}$. Suppose one transforms the circuit in the following natural way,

$$\bigwedge \longrightarrow *,\ \bigvee \longrightarrow +,\ x_i \longrightarrow x_i \text{ and } \bar{x}_i \longrightarrow 1 - x_i.$$

This transformed circuit evaluates the *number of solutions* i.e. *accepting subtrees* in the $\mathcal{NP}$ computation. This *arithmetization* of a Boolean Circuit also works for $\mathcal{NLOG}$ and $\mathcal{LOGCFL}$, though it is rather difficult to show for the latter.

Circuits that compute functions of the form $f:\ \Sigma^* \longrightarrow \mathbb{N}$ are termed *arithmetic circuits.* Any function over the ring $(\mathbb{N}, +, *)$ can be computed by arithmetic circuits. Now, if one wants to compute functions that yield *integer* values, i.e. $f:\ \Sigma^* \longrightarrow \mathbb{Z}$, one can work with the constants $\{0, 1, -1\}$. Another way is to define f, as

$$f = f^+ - f^- \text{ where } f^+ :\ \Sigma^* \longrightarrow \mathbb{N} \text{ and } f^- :\ \Sigma^* \longrightarrow \mathbb{N}.$$

That is, one is realizing the function by working over the ring $\{\mathbb{N}, +, *, 0, 1\}$.

The first scheme of generating functions over $\mathbb{Z}$ yields the $\mathcal{GAP}$ classes, while the second yields the $\mathcal{DIFF}$ classes. That is, if the decision version of the function f is in the class $\mathcal{C}$, then the two arithmetic characterizations stated above yield the $\mathcal{GAPC}$ and $\mathcal{DIFFC}$ classes.

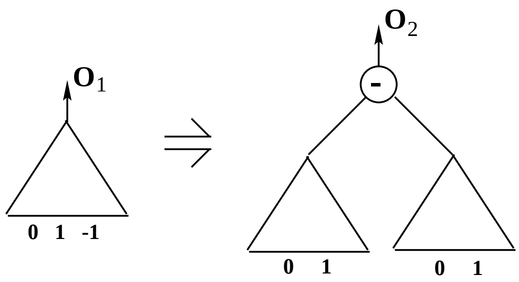

Figure 5.1: Transformation of a $\mathcal{GAP}$ circuit to a $\mathcal{DIFF}$ circuit.

Exercise: Given a $\mathcal{GAP}$ circuit, prove that it can be transformed into an equivalent $\mathcal{DIFF}$ circuit as shown in Fig 5.1.

Hint: Assume the circuit is *semi-unbounded* and use induction.

It is generally true that for the higher classes, every $\mathcal{GAP}$ circuit can be equivalently transformed into a $\mathcal{DIFF}$ circuit.

In this lecture, we shall show that $PARITY \in \mathcal{GAP}AC^0$. In a later lecture we shall see that $PARITY \notin AC^0$. This essentially reaffirms our intuition that *arithmetic circuits* are strictly more powerful than their *Boolean* counterparts.

5.2 $PARITY$, $\mathcal{GAP}AC^0$ and $\mathcal{DIFF}AC^0$

$PARITY$ essentially requires a $mod(2)$ operation on the sum of the inputs, i.e. $PARITY = \sum x_i(mod(2))$. The $mod(2)$ operation proves to be a stumbling block for a $\mathcal{GAP}AC^0$ realization. Equivalently, one can convert each input x_i to now be $(1-2x_i)$. This would transform an input 1 or 0 to a -1 or 1 respectively. Fig 5.2 shows how this can be achieved.

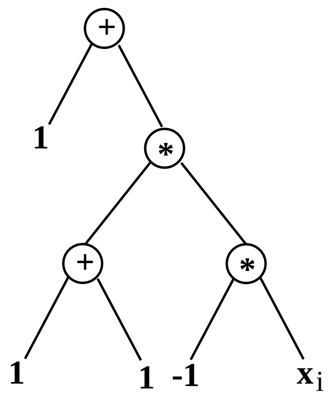

Figure 5.2: Transforming input x_i to $(1-2x_i)$.

Now, if one were to take a product of all the modified inputs, the output would be -1 if there were an odd number of 1s in the input, else 1. That is, if the $PARITY$ is even the output is 1, else it is -1. Suppose one were to add 1 to the output, the output would be $\in \{0,2\}$. All these steps can be realized in a circuit using a constant number of levels.

How does one get an output of $\{0,1\}$ instead of $\{0,2\}$? Dividing by 2 would seem to be the answer, but remember that division is beyond AC^0, and therefore, is ruled out. An interesting way out is to use $\begin{pmatrix} f \\ 2 \end{pmatrix}$, where f is some function value. In our case, the answers would be as we want, i.e. $\begin{pmatrix} 0 \\ 2 \end{pmatrix} = 0$ and $\begin{pmatrix} 2 \\ 2 \end{pmatrix} = 1$. But, why resort to this?

It is known that, if $f \in \sharp AC^0$, then $g = \begin{pmatrix} f \\ 2 \end{pmatrix} \in \sharp AC^0$. In fact, $\begin{pmatrix} f \\ k \end{pmatrix} \in \sharp AC^0$ for any constant k. The proof for this is rather ugly and a better proof is much awaited.

This would mean that $PARITY \in \mathcal{GAP}AC^0$. The question whether $PARITY \in \mathcal{DIFF}AC^0$, is an open one. The other open question is, does $\mathcal{DIFF}AC^0 \subseteq \mathcal{GAP}AC^0$? And, is the inclusion *strict*?

5.3 Main Result

The *probabilistic language class* $\mathcal{PP}$ is defined as the class of languages for which the ratio of the accepting paths to the total number of paths in the computation tree is greater than $1/2$, i.e. $\frac{\#\ accepting\ paths}{total\ \#\ paths} > \frac{1}{2}$. Contrast this with $\sharp P$ which is a *counting class*. These are functions that count the number of accepting paths in the computation tree. Note that, $\sharp P$ is closed under complementation, i.e. the negated machine is also $\sharp P$.

Let us now consider $\mathcal{GAP}P$ and look at a function $f \in \sharp\mathcal{P}$. $\bar{f}$ is the function corresponding to f negated. And $F = f - \bar{f}$ is the $\mathcal{GAP}P$ function that currently interests us. Fig 5.3 illustrates the $\mathcal{GAP}P$ function one is referring.

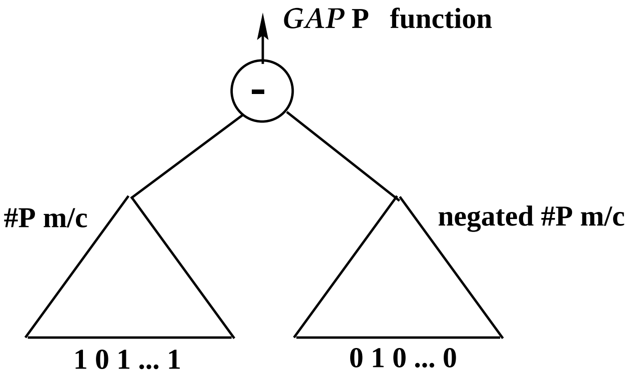

Figure 5.3: Connection between $\mathcal{PP}$ and $\mathcal{GAP}P$.

If the value of F on an input is greater than 0, it amounts to saying that the number of accepting paths (i.e. value of f) is greater than the number of rejecting paths (i.e. value of $\bar{f}$). That is, the majority of the paths in computation tree of f are accepting. The reverse holds if $F \leq 0$. This is a new characterization for the *probabilistic classes*.

$\mathcal{P}AC^0_{circ}$ (*probabilistic* AC^0) can be characterized in terms of $\mathcal{GAP}AC^0$. That is, $L \in \mathcal{P}AC^0$ if $\exists f \in \mathcal{GAP}AC^0$ such that $x \in L \equiv f(x) > 0$.

Once back in the languages domain, one would like to exactly pin down the position of $\mathcal{P}AC^0$.

This way one would have characterized $\mathcal{GAP}AC^0$ in terms of the language classes. The main result of *Allender*'s paper is

Theorem 3 $\mathcal{P}AC^0 \quad = \quad TC^0$.

The motivation for such an endeavour is to extend whatever little is known about AC^0 and TC^0. That is, it is known that $AC^0 \subset TC^0$. By way of the intimate relationship between $\mathcal{GAP}AC^0$ and TC^0, one maybe able to extend the results from AC^0 onto $\mathcal{GAP}AC^0$ and show that $TC^0 \subset NC^1$. That is, the proofs (or their techniques) from $AC^0 \subset TC^0$ might carry over to proving $TC^0 \subset NC^1$.

5.4 Uniformity

Uniformity essentially addresses the issue of, how easy is it to construct $\{C_n\}$, the family of circuits, for a language. That is, given 1^n, can one build the circuit C_n. The types of uniformity that interest us are:-

* $non-uniform$
* $\mathcal{P}$-uniform
* $\mathcal{LOG}$space-uniform
* $\mathcal{DLOG}$time-uniform

$Non-uniformity$ refers to those languages that may have small circuits, but it is very difficult to construct them. *Tally* languages are an example. These are languages of the form $\{111, 11111, \ldots\}$. The circuit C_n for a particular value of n is either the constant 1 or 0. Since there are an *uncountable* number of such languages but only a *countable* number of possible circuits, some languages cannot be expressed easily.

$\mathcal{P}$-uniformity refers to those languages whose circuit families can be constructed in polynomial time. Similarly, one defines $\mathcal{LOG}$space-uniform and $\mathcal{DLOG}$time-uniform classes of languages.

It is an known that $Iterated\ Multiplication \in \mathcal{P} - uniform\ TC^0$. It is open whether it belongs to any of the lower uniformity classes.

Computational Complexity Theory **Spring 1997**

Lecture 10 : February 20, 1997

Lecturer: V. Vinay *Scribe: P. R. Subramanya*

In this lecture we shall look at *Brent*'s result on formulas. This will be continued on to an introduction to *Branching Programs* which precludes *Barrington's* seminal paper on NC^1 and *Bounded Width Permutation Branching Programs*.

6.1 Brent's result

Brent showed in the mid-70s that any polynomial-sized binary tree can be converted into an equivalent tree of depth $O(\log n)$. In the next few sections, *Boolean formulae* will be defined followed by *Brent*'s result.

6.1.1 Formulae

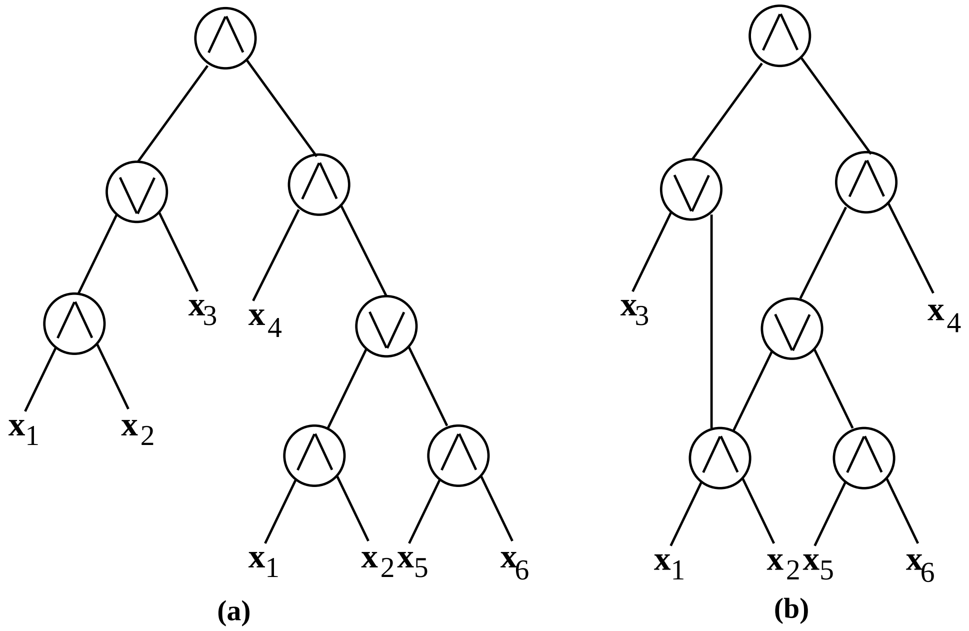

Figure 6.1: An example of a formula and its circuit

A *Boolean formula* is a tree, i.e. a DAG where every node except the root has a unique parent, the inputs occur as leaves and each internal node is a Boolean binary operation. Contrast this with a *Boolean circuit* where an internal node can have a fan-out greater than 1. Fig 6.1 gives a sample circuit and its representation as a formula.

NC^1 is the class of bounded fan-in polynomial sized circuits of depth $O(\log n)$. The standard way of converting a *circuit* into a *formula* is by replicating every internal node as many times as its fan-out. One can show that the conversion of an NC^1 circuit to its corresponding formula causes a polynomial increase in the size.

Exercise: Show that the conversion of an NC^1 circuit to a *Boolean formula* causes a polynomial increase in its size.

This implies that NC^1 can be viewed as a Boolean Tree evaluation where the depth is $O(\log n)$. Every binary tree of depth $O(\log n)$ is necessarily convertible to a NC^1 circuit. But then, this would be restricting the type of trees that one is looking at. It would be better if one can talk about trees that can be transformed into NC^1 circuits. So, the question is, what are the types of trees that can be transformed into NC^1 circuits?

Brent showed in the mid-70s that any polynomial-sized tree can be converted into an equivalent one of $O(\log n)$ depth.

6.1.2 $\frac{1}{3}$-$\frac{2}{3}$ separator

Given a binary tree with m nodes, Brent claimed that there always exists a node such that the subtree below it will have size in the range $[\frac{m}{3}, \frac{2m}{3}]$. How does one show the existence of such a node?

Let us define the weight of a node to be the size of the subtree below it (i.e. the number of descendants). Suppose one starts from the root and goes to the heavier child. One follows this strategy till one reaches a leaf and then, one would have defined a path in the tree. Along this path, starting from a weight of m, one has dropped to a weight of 1. One can now apply the discrete version of *Rolle's* Mean Value Theorem.

There necessarily has to be some node with a weight just greater than $\frac{2m}{3}$. And, the next node would have a weight less than $\frac{2m}{3}$. But, this node cannot have a weight less than $\frac{m}{3}$ since the tree is a binary tree and the traversal strategy is that of moving to the heavier child.

6.1.3 Joining the two pieces

Once the tree has been separated by the $\frac{1}{3}$-$\frac{2}{3}$ separator, as shown in Fig 6.2, how does one join the two together. Note that the objective behind the whole exercise is to reduce the depth of the tree to $O(\log n)$. This too remains to be shown.

Y_2, the separated sub-tree, is now treated as a variable in Y_1, the dismembered tree. By making *two* copies, Y_1' and Y_1'', of Y_1 and assuming Y_2 to be 0 and 1 resp., one would have captured the vagaries of Y_2. The selection of Y_1' or Y_1'' is done through a *selector* gate with the control input being Y_2's output. This is illustrated in Fig 6.3. Note that the selector gate is the function $\theta = \bar{c} \cdot x_0 + c \cdot x_1$. This can be realized by a depth 2 circuit.

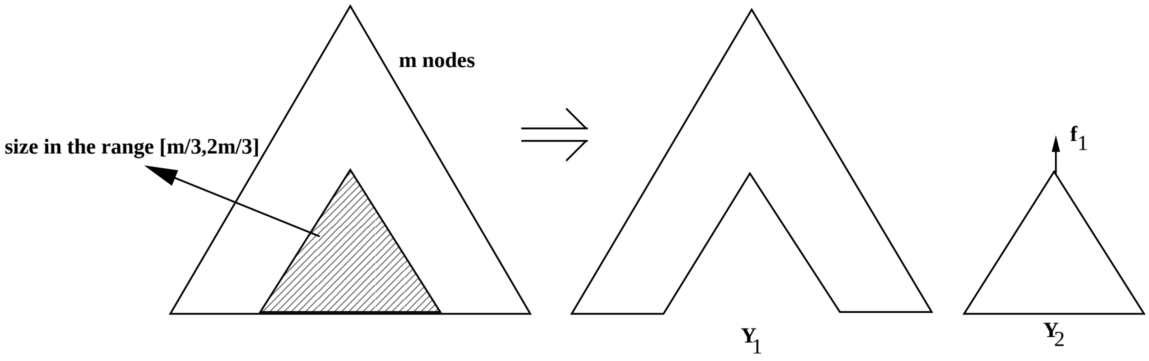

Figure 6.2: Effect of $\frac{1}{3}$-$\frac{2}{3}$ separator.

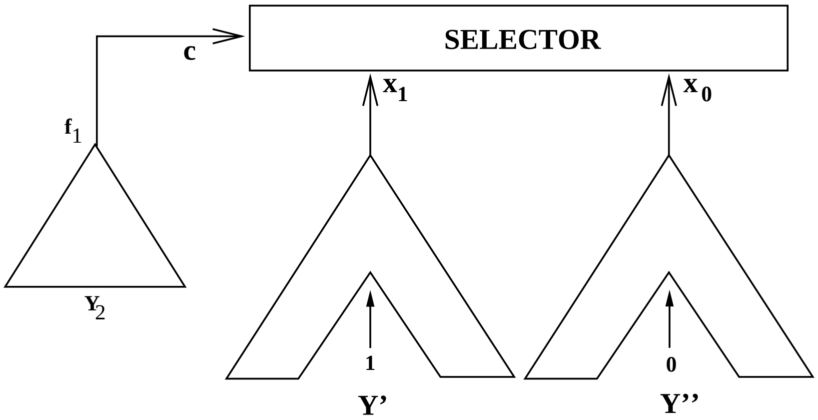

Figure 6.3: Joining the two sub-trees.

Let us consider the recurrence relation for the depth. $D(x)$ is the depth of the resultant tree with x number of nodes.

$$D(m) \leq D(\frac{2m}{3}) + 2$$

If one expands out the recurrence relation, $D(m)$ comes out to be $O(\log m)$. A similar recurrence relation for the size shows that there is $2m + O(\log m)$ increase in the size.

$$S(m) \leq m + 3 + S(\frac{2m}{3})$$

Any *arithmetic tree* can be converted into an NC^1 circuit. The $\frac{1}{3}$-$\frac{2}{3}$ separator trick does not work in this general case. It is also known that if *division* is allowed, the tree can be converted to an NC^1 circuit with a *single* division.

6.1.4 Evaluating and expression tree

While we are on trees, there is an interesting problem of determining how many registers would be required to evaluate an expression tree (*arithmetic* or *boolean*). The naive technique of computing the operands and then performing the operation, when at an internal node, requires $O(\log n)$ registers. In fact, any of the standard tree traversal schemes (*depth-first traversal* and its variants) require $O(\log n)$ registers.

Ben-Or and *Cleave* have shown that this evaluation can be done with just 3 registers! We shall look at this result in a later lecture.

6.2 Branching Programs

A *Deterministic Branching Program(DBP)* is a *DAG* with a special designated vertex s, called the *source* and two *sink* vertices 0 and 1. Every node v has an associated input x_i and two out-edges corresponding to the 0/1 value of x_i. Given an input $x_1 x_2 \ldots x_n$, a path is defined from s to either of the sink vertices. If there is a path from s to the sink vertex 1, then the input is accepted, else rejected. Note that, a *DBP* can be converted into a layered *DAG* and the *width* of the DBP is the maximum number of nodes in any particular layer.

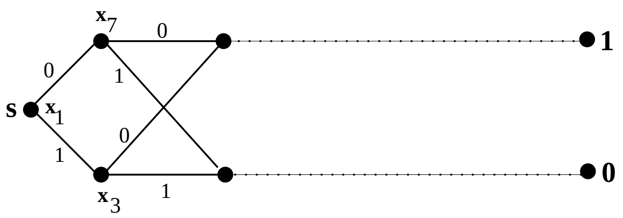

Figure 6.4: Deterministic Branching Programs.

*DBP*s are a generalization of *DFA*s. *DBP*s can be viewed as a *non- uniform FA*. *DFA*s are online processes while *DBP*s are off-line in the sense that, in a *DFA* once an input is given, it has to be necessarily consumed while in a *DBP*, an input can be looked at any time, any number of times.

The *width* of a *DBP* can be viewed as the number of registers required for the computation to complete. When one goes from one layer to the next, the register values can be updated and the computation can proceed with these updated registers. So, a lot of people looked at the *width* as a resource and tried to identify the power of *DBP*s. Fig 6.5 illustrates how $PARITY$ can be accomplished as a *width-2 DBP*.

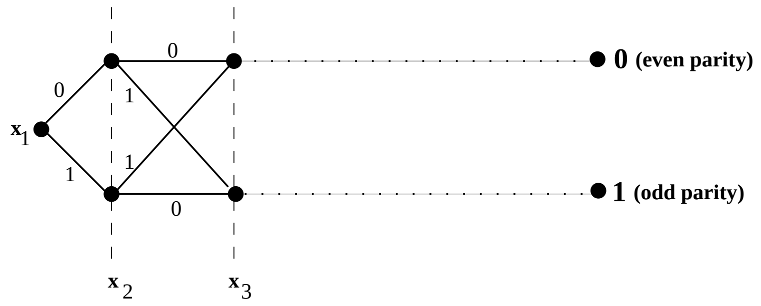

Figure 6.5: $PARITY$ is in *width-2 DBP*s.

The major question in the 80s was whether $MAJORITY$ can be computed using *bounded width DBP*s. It was strongly suspected that $MAJORITY$ cannot be realized by *bounded-width DBP*s, until *Barrington* showed that NC^1 is *exactly width-5 DBP*s, also termed as *bounded width-5 branching programs* (*5-BWBP*).

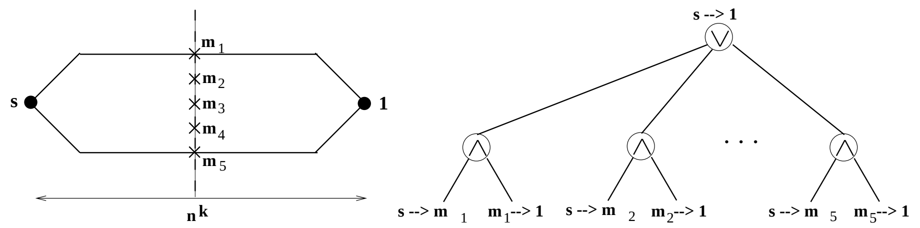

Figure 6.6: Showing 5-BWBP $\subseteq NC^1$.

Showing *5-BWBP*s $\subseteq NC^1$ is rather simple. Take the mid-points m_1, m_2, m_3, m_4, m_5. If there is a path from $s \longrightarrow 1$, then it must pass through at least one of the mid-points. Hence, to verify that on an input there is a path from s to 1, one checks if at least one of the following is satisfied,

$$(s \longrightarrow m_i) \bigwedge (m_i \longrightarrow 1) \quad where \;\; 1 \leq i \leq 5$$

We can then proceed recursively with the verification. This scheme can be converted into a circuit as shown in Fig 6.6. Since the size (i.e. length) of the *5-BWBP* is a polynomial, the depth of the circuit is $O(\log n)$. That is, an NC^1 circuit. More of *Barrington*'s result in the next lecture.

Computational Complexity Theory	**Spring 1997**
Lecture 11-12 : February 25 & March 6, 1997	
Lecturer: V. Vinay	*Scribe: P. R. Subramanya*

Barrington's seminal paper on *Bounded-Width Branching Programs* (BWBP) shall be discussed. We shall also look at *Ben-Or & Cleve*'s result showing that any polynomial size expression tree can be evaluated using 3 registers. This result was inspired by *Barrington*'s paper. But first, we shall clarify a few things from the previous lectures.

7.1 $PARITY$ revisited

We had looked at $PARITY$ in *lecture 9*. There we had shown that by changing each input $x_i \longrightarrow (1-2x_i)$ and taking a product, we could figure out $PARITY$. In fact, there exists a rather simple function for $PARITY$, a $\mathcal{GAPAC}^0$ function.

$$PARITY(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} \left(\prod_{j<i} (1-2x_j) \right) x_i$$

Let us consider an example. The input 0001011010 results in the following intermediate results

Input	0	0	0	1	0	1	1	0	1	0
Product term	1	1	1	1	-1	-1	1	-1	-1	1
Summation terms	0	0	0	1	0	-1	1	0	-1	0

$\Rightarrow$ 0, i.e. *even parity.*

Exercise: The above characterization of $PARITY$ requires $(n-1)$ subtractions. Can this be reduced to anything below $(n-1)$? In fact, it is an open problem as to whether $PARITY$ necessarily requires greater than one -1s. This would then mean that $PARITY \notin \mathcal{DIFFAC}^0$.

Exercise: Show the following identities,

$$(a) \quad \binom{\sum_{i=1}^{n} x_i}{2} = \sum_{i=1}^{n} \binom{x_i}{2} + \sum_{i<j} x_i x_j$$

$$(b) \quad \binom{\prod_{i=1}^{n} x_i}{2} = \sum_{i=1}^{n} \left[\prod_{j<i} x_j \binom{x_i}{2} \prod_{j>i} x_j^2 \right]$$

7.2 Brent's result revisited

If one recalls, *Brent*'s result was shown for Boolean binary trees. Here we shall briefly outline two schemes to show that the result indeed does hold for general arithmetic circuits.

The first scheme is shown in Fig 7.1. If one considers the separated sub-tree as a variable x, then the evaluation of the whole tree can be expressed in terms of x as, $Px+Q$ where P and Q are constants. By setting $x=0$, one would get Q and, by setting $x=1$ one would get $P+Q$. If one subtracts these two quantities, one would be able to get P and Q separately. Then, x as evaluated by the separated sub-tree can be plugged in at the appropriate place to get our rejoined function.

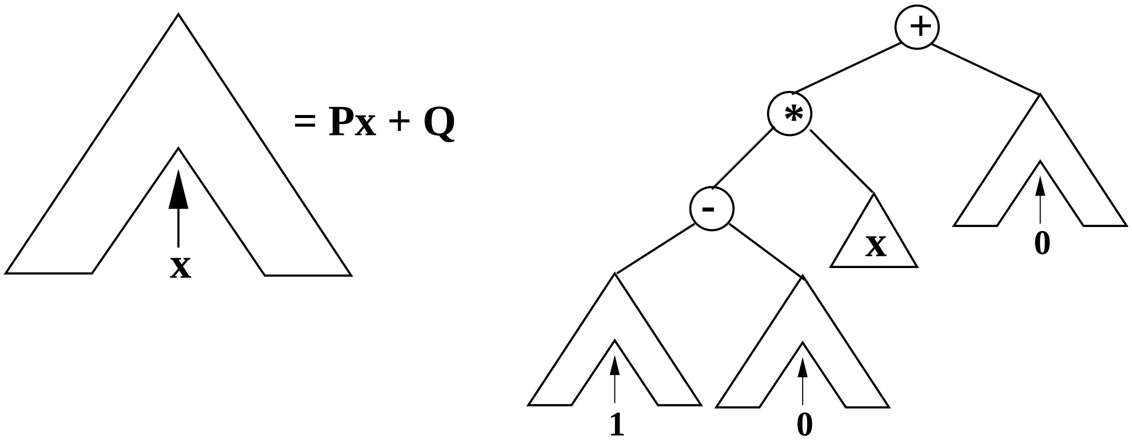

Figure 7.1: A scheme for joining the separated sub-trees.

The problem with this scheme is that a circuit which originally contained just $+$ and $*$, now contains $-$. The $-$ operation might not be meaningful over certain rings.

The second method is due to *Brent*([8]). In an arithmetic tree with m nodes, one can always find a node v, whose left and right children are l_v and r_v, with the following property,

$$\sharp(v) \geq \lceil \frac{m}{2} + 1 \rceil \;\; ; \;\; \sharp(l_v) < \frac{m}{2} \;\; ; \;\; \sharp(r_v) < \frac{m}{2}$$

Exercise: Show that the above property holds for any tree.
{ *Hint:* A partial condition holds for a path from the root to a terminal node. Extend this to show that of all these paths, one satisfies the above property.}

Exercise: This problem is about inverting a sorted input.

Input	$x_1x_2\ldots x_n$	$\longrightarrow$	1^k0^{n-k}
Output	$y_1y_2\ldots y_n$	$\longrightarrow$	0^k1^{n-k}

A naive realization for the above would be $y_i = \neg x_i$, that requires n $\neg$-gates. One should invert the input using at most $O(\log n)$ $\neg$-gates. This is part of a result shown by *Markov* that in any circuit, at most $\log n$ negations are sufficient.
Hint: Use binary search and generate an NC^1 circuit.

7.3 Bounded-Width Branching Programs

In branching programs we have the scenario shown in Fig 7.2. Given an input, certain paths only would be active. In this restricted graph, if there is a path $s \longrightarrow t_+$ then the input is accepted and if there is a path $s \longrightarrow t_-$, then the input is rejected.

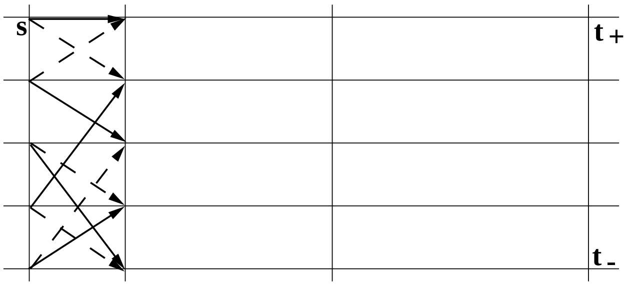

Figure 7.2: Branching Program.

Width-k Branching Program would be a BP where the number of rows is bounded by a constant k. We already saw that *width-5 BP* $\subseteq NC^1$.

BPs have too much flexibility and it is difficult to pin-down their power. *Barrington* used a restricted version of BPs called *Permutation Branching Programs (PBPs)* under the permutation group S_5.

Each instruction in a $5 - PBP$ is of the form $(x_i, \sigma_i, \gamma_i)$. This is to be interpreted as **if** x_i **then** σ_i **else** γ_i. x_i is an input symbol and each σ_i and γ_i is a permutation in S_5. A *5-PBP* would be a sequence of such straight-line instructions. On an input, we would get an evaluation of the form $\alpha_1\alpha_2\ldots\alpha_m$ where each α_i is either σ_i or γ_i depending on the input x_i. This is a composition of permutations whose result is another permutation from S_5. The acceptance condition for a *5-PBP* is defined as

$$x_1x_2\ldots x_n \in L \Rightarrow \alpha_1\alpha_2\ldots\alpha_m = \theta$$

$$x_1x_2\ldots x_n \notin L \Rightarrow \alpha_1\alpha_2\ldots\alpha_m = e$$

The acceptance condition for a *BP* was defined based on whether there is a path $s \longrightarrow t_+$ while for a *PBP* it is dependent on the composition of the permutations being equal to a specific one, θ. Does this mean a *PBP* is not a *BP*? No, one can translate the *PBP* acceptance criterion into the *BP* framework.

Let us take an example where the accepting permutation θ is given by $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 1 & 3 & 2 \end{pmatrix}$. This can also be written in the *cycle notation* as $(1\ 4\ 3)(2\ 5)$. In the *BP* framework, this amounts to saying that, starting from 1, one should end up in 4, and starting from 4, one should end at

3, and starting from 3 one should end in 1, and so on. Fig 7.3 shows the equivalent acceptance condition of a *PBP* in the *BP* framework. Note that, concatenating *PBP*s amounts to *AND*ing.

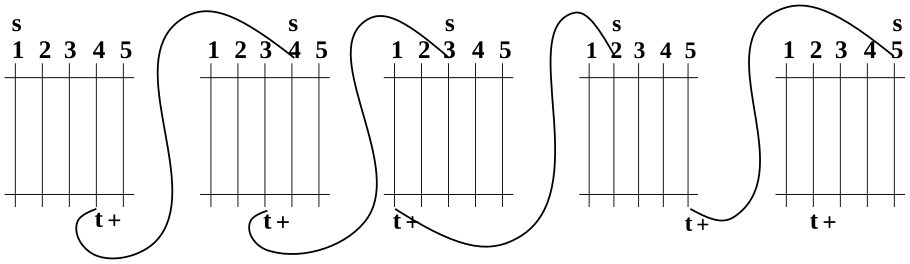

Figure 7.3: PBPs in BP framework.

Barrington put a further restriction on the permutations and asserted that $5 - cycles$ were sufficient. That is, a $5 - PBP$ B *five-cycle recognizes* a set $A \subseteq [2]^n$ if there exists a five-cycle σ (called the *output*) in the permutation group S_5 such that $B(x) = \sigma \; if \; x \in A$ and $B(x) = e \; if \; x \notin A$. ([3])

We shall state a few properties of five-cycles from ([3]).

Lemma 1 *There are five-cycles θ_1 and θ_2 in S_5 with the property that their* commutator *is also a five-cycle. That is,*

$$\exists 5 - cycles \;\; \theta_1, \theta_2, \theta_3 \quad s.t. \;\; [\theta_1, \theta_2] \neq e, \quad i.e. \quad \theta_1\theta_2\theta_1^{-1}\theta_2^{-1} = \theta_3$$

Proof: The θ_1,θ_2 and θ_3 for which this works is

$$(12345)(13542)(54321)(24531) = (13254)$$

■

Lemma 2 *If B five-cycle recognizes A with output σ and τ is any five-cycle, then there exists a $5 - PBP$ B', of the same length as B, which five-cycle recognizes A with output τ.*

Proof: Since σ and τ are five-cycles, there exists some permutation θ with $\tau = \theta\sigma\theta^{-1}$. If one changes the σ_i and τ_i in each instruction to $\theta\sigma_i\theta^{-1}$ and $\theta\tau_i\theta^{-1}$, one would get the required program B' that accepts on τ and is of the same length as B. ■

Lemma 3 *If A can be five-cycle recognized in length l, so can its complement.*

Proof: Suppose B is the $5-PBP$ that five-cycle recognizes A with output σ. Let the last instruction be $< i, \mu, \rho >$. If the last instruction were changed to $< i, \mu\sigma^{-1}, \rho\sigma^{-1} >$, then the program would accept on inputs that are rejected by B and reject on inputs accepted by B. ■

We need to show that $NC^1 \subseteq 5-BWBP$. Consider a $\wedge$-gate. $WLOG$ the left and right operands are $l_\wedge$ and $r_\wedge$ taking values,

$$l_\wedge = \begin{cases} \theta_1, & if\ 1 \\ e, & otherwise \end{cases} \quad r_\wedge = \begin{cases} \theta_2, & if\ 1 \\ e, & otherwise \end{cases}$$

The table below shows the effect of a *commutator* node on the operands of the $\wedge$-gate.

$r_\wedge \longrightarrow$ $l_\wedge \downarrow$	e	θ_2
e	$e \cdot e \cdot e^{-1} \cdot e^{-1} = e$	$e \cdot \theta_2 \cdot e^{-1} \cdot \theta_2^{-1} = e$
θ_1	$\theta_1 \cdot e \cdot \theta_1^{-1} \cdot e^{-1} = e$	$\theta_1 \cdot \theta_2 \cdot \theta_1^{-1} \cdot \theta_2^{-1} = \theta_3$

Only when both the operands of the $\wedge$-gate are 1s, will we get another five-cycle as the output. This indicates that the *commutator* acts as a $\wedge$-gate. We earlier saw that the complement of a language can be recognized by a $5-PBP$ of the same length. That is, the $\neg$-gate can also be cast in the $5-PBP$ framework. This effectively means that one can convert a Boolean circuit into a $5-PBP$.

What one needs to show now is that the size of a $5-PBP$ program corresponding to a NC^1 circuit is polynomial. In ([3]), *Barrington* stated the following theorem:-

Theorem 4 *Let a set $A \subseteq [2]^n$ be recognized by a depth d fan-in 2 Boolean circuit. Then A can be five-cycle recognized by a $5-PBP$ B of length at most 4^d.*

Proof: We shall use induction on the depth d to prove the result. The base case, $d = 0$, where the circuit has no gates can be recognized by a single instruction $5-PBP$. Assume that the theorem is true for all circuits of depth $\leq (d-1)$. Consider a depth d circuit. Its two operands can be of depth at most $(d-1)$ and the $5-PBP$s that recognize them are of length at most 4^{d-1}. WLOG one can assume that the output gate is a $\wedge$-gate (a $\vee$-gate can be expressed using $\wedge$ and $\neg$-gates).

Let the $5-PBP$s for the left and right operands be B_l and B_r of length at most 4^{d-1} with their output permutations being forced to θ_1 and θ_2, as in lemma 1. We construct B_l' and B_r' such that their output permutations are θ_1^{-1} and θ_2^{-1} resp. From *lemma 2* we know that the length of the programs B_l' and B_r' will not increase. If one concatenates the programs B_l, B_r, B_l' and B_r', one will have the desired program B which will output θ_3 **iff** the left and right operands of the $\wedge$-gate are 1s. And the length of B is at most 4^d. ■

If one sets d to be $O(\log n)$, then one has a NC^1 circuit with the corresponding $5-PBP$ being polynomial in size. This means that $NC^1 \subseteq 5-BWBP$. And, we already saw that $5-BWBP \subseteq NC^1$. This means that *polynomial-size Bounded-Width Branching Programs* correspond exactly to NC^1.

7.3.1 Outfall of Barrington's result

It is known that $LOGSPACE \subset PSPACE$, and the inclusion is strict. With *Barrington*'s result, it seemed surprisingly possible to get a $PSPACE$ machine from a $LOGSPACE$ machine with some additional "information". Suppose one has a notion of an *epoch* and is allowed to carry along just *5 bits* of information across epochs. One runs a $LOGSPACE$ machine for an epoch, which is a polynomial amount of time, and at the end retains just 5 bits of information about the epoch. In the next epoch, one runs the $LOGSPACE$ machine with this 5 bits of information. Continuing this way, it would be possible to reach $PSPACE$.

The whole idea seems very philosophically appealing.

7.4 Ben-Or & Cleve's Result

Ben-Or & Cleve in ([6]) showed the following result. Let $f(x_1, x_2, \ldots, x_n)$ be an algebraic formula of size s over an arbitrary ring $(\mathcal{R}, +, \cdot, 0, 1)$. One can construct a *straight-line program* of length polynomial in s that computes $f(x_1, x_2, \ldots, x_n)$ and uses 3 registers.

The straight-line programs that arise in the constructions have a special form. These consist of statements that apply special invertible linear operations to the registers. One can consider each possible configuration of values of the 3 registers as a vector in $\mathcal{R}^3$. The effect of executing a statement of these straight-line programs is equivalent to multiplying this vector by a $3X3$ matrix with determinant 1. That is, the statements can be treated as elements from the *special linear group* $SL_3(\mathcal{R}) = \{A : det(A) = 1 \;\; and \;\; A \; has \; dimension \; 3\}$.

Suppose, the 3 registers are R_1, R_2, R_3. The effect of multiplying the vector $(r_1 \; r_2 \; r_3)$, formed by the values of the 3 registers, with an element from $SL_3(\mathcal{R})$ is,

$$(r_1 \; r_2 \; r_3) \cdot \begin{pmatrix} 1 & 0 & 0 \\ f & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = ([r_1 + r_2 \cdot f] \; r_2 \; r_3)$$

That is, register R_1s value is offset by f times the value of register R_2. Let us consider the composition of matrix multiplications,

$$(r_1 \; r_2 \; r_3) \cdot \begin{pmatrix} 1 & 0 & 0 \\ f & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = ([r_1 + r_2 \cdot f + r_2 \cdot g] \; r_2 \; r_3)$$

That is, if the left and right operands of a $+$ node are f and g and $r_2 = 1$, the effect of the above composition of matrices is to offset the value of R_1 by $f + g$. Note that, the values of the other two registers are unchanged during the operation.

Let us consider the $\cdot$ node in the expression tree whose left and right operands are f and g. The following composition of matrices achieves the effect of offsetting the value of the R_1 register by $f \cdot g$ if R_3 is initialized to 1.

$$(r_1\ r_2\ r_3) \cdot \begin{pmatrix} 1 & 0 & 0 \\ f & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -g & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ -f & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & g & 1 \end{pmatrix} = ([r_1 + r_3 \cdot f \cdot g]\ r_2\ r_3)$$

Again note that the values of registers R_2 and R_3 are left unchanged at the end of the operation.

Theorem 5 (*Ben-Or & Cleve) ([6])*
Over an arbitrary ring $(\mathcal{R}, +, \cdot, 0, 1)$, any formula $f(x_1, x_2, \ldots, x_n)$ of depth d is computed by a straight-line program that uses 3 registers and has length at most 4^d.

Proof: One shall prove the result recursively on the depth d. When $d = 0$, a single statement offsets register R_1 by $\pm r_2 \cdot c$ or $+ - r_2 \cdot x_i$ for some $i \in \{1, 2, \ldots, n\}$. For any other depth $d > 0$, if the left and right operands are f and g, then we saw how to get the $+$ and $\cdot$ operations through straight-line programs.

Now, since the maximal recursive factor per level of the depth is at most 4, if the depth of the formula is d then the length of the straight-line program is 4^d. ■

Using *Brent*'s result ([8]) that any polynomial size expression tree can be converted into an equivalent one of depth $O(\log n)$, one can state the following corollary,

Corollary 1 *Over an arbitrary ring $(\mathcal{R}, +, \cdot, 0, 1)$, a formula of size s is computed by a straight-line program, of the form stated above, that uses 3 registers and has length polynomial in s.*

Computational Complexity Theory	**Spring 1997**
Lecture 13 : March 11, 1997	
Lecturer: V. Vinay	*Scribe: P. R. Subramanya*

This lecture focuses on the fundamental result by *Immerman* ([12]) and *Szelepcsényi* ([20]). They independently showed that $\mathcal{NL}$ (*Non-deterministic Logspace*) is closed under complementation. In fact, *Immerman* ([12]) extends the result to any nondeterministic space class higher than and including $\mathcal{NL}$. *Savitch*'s theorem ([18]) shall also be discussed.

8.1 s-t connectivity (STCONN)

Suppose one is given the following problem.

Input: $< G, s, t >$ where G is a DAG, and s and t are 2 special vertices.
Question: Is there a path from s to t in G?

Let us see how one can solve this. Each vertex would require $\log n$ bits. Starting at the initial vertex s, the algorithm proceeds as:-

1. Set v to be the current vertex.
2. Non-deterministically go to some neighbour of v.
3. Check if this vertex is t.
4. If not, go to step 1.

All one needs to remember is the current vertex and the sink vertex. That is, $O(\log n)$ bits are sufficient for the algorithm to answer the *STCONN* question. *STCONN* is $\mathcal{NL}$-Complete. We shall see why.

One needs to show a reduction of every $\mathcal{NL}$ machine, M, on input x to *STCONN* to prove that the latter is $\mathcal{NL}$-Complete. Let us consider the $\mathcal{NL}$ machine shown in Fig 8.1. The work tape is bounded by $O(\log n)$ bits. The *Instantaneous Description* (**ID**), i.e. the configuration of the machine at any instant of time, consists of

Work Tape Contents	Requires $O(\log n)$ bits.
State of the machine	There are a finite number of states, i.e. some constant k.
Input Head Position	Requires $O(\log n)$ bits.
Timer	Polynomial time is sufficient, i.e., $O(\log n)$ bits.

The total number of configurations is $n^{O(1)}$ since each configuration requires $O(\log n)$ bits.

Now, consider the graph $G(V, E)$ whose vertex set, V, is the set of all configurations. A directed edge exists between 2 configurations v_1 and v_2 if v_2 is reachable from v_1 in one step. That is, E consists of edges $< w, q, i, t > \longrightarrow < w', q', i', t+1 >$ if the machine starting in state q, with WT contents w and the input head at i goes to state q', with WT contents w' and the input head at t' in *one step.*

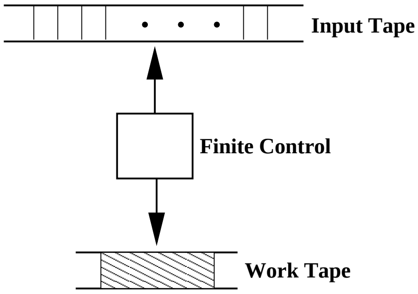

Figure 8.1: A Turing Machine.

Setting $s =< \phi, q_0, 0, 0 >$ and $t =< \phi, q_f, 0, n^k >$, one has an instance of *STCONN*. The reduction itself can be accomplished in $\mathcal{DLOG}$. (Exercise: Show how the reduction can be done in $\mathcal{DLOG}$?)

Note: One can enforce the $\mathcal{NL}$ machine to clear its WT and reset its head before accepting.

Showing $\mathcal{NL} \subseteq \mathcal{P}$ is left as an exercise. It follows from a polynomial algorithm for *STCONN*.

8.2 The Mind block

Showing that $\mathcal{NL}$ is closed under complementation was left untouched for a long time because of a mind block among researchers. Consider some instance of *STCONN* as illustrated in Fig 8.2.

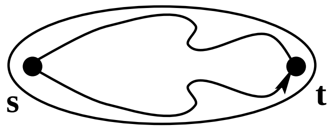

Figure 8.2: An instance of $STCONN$.

The complementary problem is one of finding whether s and t are unconnected. To show $\mathcal{NL}$ is closed under complementation, one would have to do the reduction from $(G, s, t) \longrightarrow (G', s', t')$ such that whenever there is a path from s to t in G, there are no paths from s' to t' in G' and vice-versa.

The algorithm that we presented for *STCONN* used some local information only, i.e. the current and sink vertices. In the process one was able to say something about the existence of a path from s to t. Concluding that there is no path from s to t would be saying something global about

G. One feels that one cannot conclude such global information from some local information and hence the mind block.

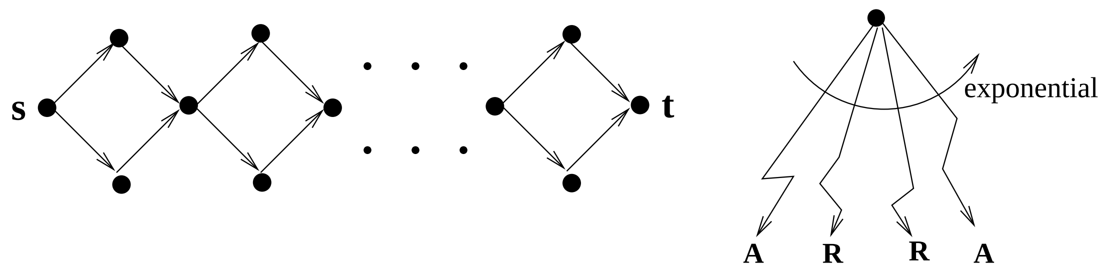

Figure 8.3: The s-t connectivity problem with exponential paths.

Consider another aspect. The graph in Fig 8.3 has an exponential number of paths from s to t. In other words, in an $\mathcal{NL}$ computation too, one could have an exponential number of paths to check. The co$\mathcal{NL}$ problem would require all of these exponential paths to be rejecting. Checking this within the limitations of logspace did not seem possible.

8.3 Inductive Counting

Suppose one is given n bits, $x_1, x_2, \ldots, x_n$, and that there are exactly k 1s in the input. No information about the arrangement of the 1s is given. How would one invert the input bits using *monotone circuits*? (Hint: Threshold gates can be used.)

Consider the i^{th} bit x_i. If it is a 1, the number of 1s in the remaining input bits will be $k-1$. And, if x_i is 0, the sum is k. Hence, setting

$$y_i = \neg x_i = Th_k^{n-1}(X \backslash x_i) \quad \forall i$$

One has already seen *monotone* realizations of Th_k^n.

Let us come back to *STCONN*. Suppose, in addition to being given $< G, s, t >$, one is also given the **count** of the vertices that are reachable from s. How would one solve $\overline{STCONN}$, the complementary problem?

Let us assume the vertices are numbered such that s is 1 and t is n. The following algorithm can decide $\overline{STCONN}$. Assume one knows *count*.

For $v = 1$ to $n-1$
    Guess if v is *reachable* from s.
    If the guess is *reachable*, then
        Prove $< G, s, v >$.

Kill a path if it fails to discover a path from s to v.
If a path is found from s to v, then **decrement** *count*.
If *count* is 0, then **Accept**. Else **Reject**.

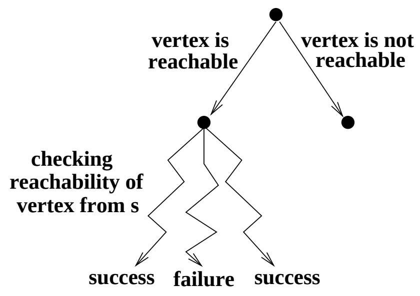

Figure 8.4: The s-t connectivity solution when *count* is given.

Fig 8.4 shows a single step in the iteration of the above algorithm. The space requirements for the above algorithm indicates that it is safely in $\mathcal{NL}$.

Current vertex	Vertex being currently checked. $\Rightarrow O(\log n)$ bits
Vertex count	The reachable vertices count. $\Rightarrow O(\log n)$ bits
Reachable Flag	The guess for the current vertex. 1 bit
Intermediate vertex	$O(\log n)$ bits for showing $s \longrightarrow vertex$.

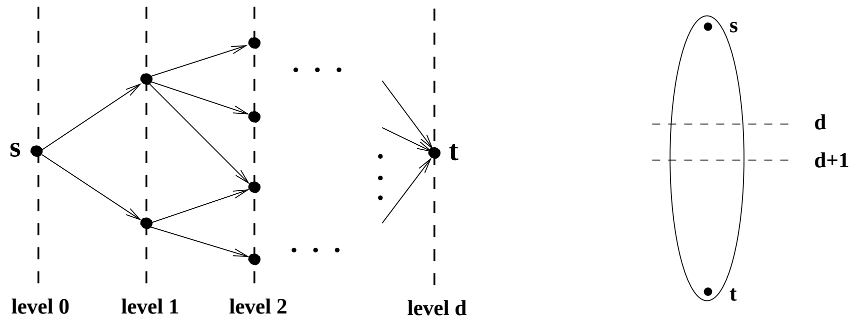

Figure 8.5: The layered graph and inductive counting.

What remains to be shown is that *count* can be found in $\mathcal{NL}$. Let us look at a layered graph as shown in Fig 8.5. For each layer k one shall maintain a count of the number of vertices upto layer k that are reachable from s through the variable $count_k$. $count_1$ would be the vertices in layer 1 that are reachable from s, i.e., the neighbours of s. We shall indicate by way of *inductive counting* how one can find the number of vertices upto layer $(d+1)$ given $count_d$, the number of vertices upto layer d reachable from s.

Set $counter_d = 0$; $count_{d+1} = 0$.
For each vertex v at level $d + 1$.
    For each vertex w at level d.
        Guess w is reachable from s.
        If w is indeed reachable from s then
            increment $counter_d$.
            If there is an edge $w \longrightarrow v$ then.
                Set flag $reachable$.
    If $counter_d = count_d$ then
        If $reachable$ flag set then
            Increment $count_{d+1}$.
    Else, reject.

One stops at a level $k + 1$ when $count_k = count_{k+1}$.

Exercise: The above two schemes need to be merged in order to show that $\mathcal{NL}$ is closed under complementation. How many $\log n$ bits would be required in total?

8.4 Savitch's Theorem

Savitch ([18]) showed that $NSPACE(s(n)) \subseteq DSPACE(s^2(n))$ for $s(n) \geq \log n$. That is, given an $s(n)$ space bounded nondeterministic machine, one can simulate it deterministically in a $s^2(n)$ space bounded machine. Now, if $s(n) = n^{O(1)}$ then, we have $NSPACE(n^{O(1)}) = DSPACE(n^{O(1)}) = PSPACE$.

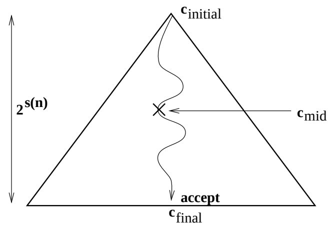

Figure 8.6: Computation tree for $NSPACE(s(n))$.

Consider the computation tree for a $NSPACE(s(n))$ machine, as shown in Fig 8.6, with $c_{initial}$ and c_{final} as the initial and final configurations. One needs to check the existence of a path $c_{initial} \longrightarrow c_{final}$. This path existence question can be encapsulated as a triple, $path(c_{initial}, c_{final}, 2^{s(n)})$. For $c_{initial} \longrightarrow c_{final}$ to be answered in the affirmative, the triples $path(c_{initial}, c_{mid}, 2^{s(n)-1})$ and $path(c_{mid}, c_{final}, 2^{s(n)-1})$ should be true for some c_{mid}. That is, there must be a mid-point, c_{mid}, such that there are paths $c_{initial} \longrightarrow c_{mid}$ and $c_{mid} \longrightarrow c_{final}$ of size $2^{s(n)-1}$ each.

One finds c_{mid} by cycling through all configurations. While asserting some triple, one would have to assert some other triples. With the help of the stack, one recurses on the triples to answer the questions that are generated. The size of each triple is $3 \cdot s(n)$ and there would be $s(n)$ triples on the stack. That is, the space requirements would be $3 \cdot s^2(n)$.

8.4.1 A few open questions

Savitch showed that $NSPACE(s(n)) \subseteq DSPACE(s^2(n))$. It is not known whether one can show $NSPACE(s(n)) \subseteq DSPACE(s^{2-\epsilon}(n))$. In the logspace world, several interesting things have been shown are are still to be shown.

We know that $\mathcal{NL} \subseteq \mathcal{P}$ and also that $\mathcal{NL} \subseteq DSPACE(\log^2 n)$. But, it does follow that $\mathcal{NL} \subseteq DSPACE, TIME(\log^2 n, n^{O(1)})$. This is an open question. *Nisan* ([14]) showed that $\mathcal{RL} \subseteq DSPACE, TIME(\log^2 n, n^{O(1)})$. It is open whether the inclusion holds for $DSPACE, TIME(\log^{\frac{3}{2}} n, n^{O(1)})$.

Note: $\mathcal{RL}$ is the *logspace bounded randomized machine* that accepts if the number of accepting paths is greater than $\frac{1}{2}$ of the total number of paths.

Saks & Zhou used ideas from ([14]) to show that $RSPACE(s(n)) \subseteq DSPACE(s^{\frac{3}{2}}(n))$.

Computational Complexity Theory **Spring 1997**

Lecture 14-16 : March 13, 18 & 20 1997

Lecturer: V. Vinay *Scribe: P. R. Subramanya*

In this and the next set of lecture notes, we shall discuss some of the techniques that have been developed to show that $PARITY \notin AC^0$. These results are towards showing $AC^0 \subset TC^0$. *Hastad's Switching Lemma* ([11]) and *Razborov's Approximations* ([15]) shall be discussed.

9.1 Introduction

Let us look at the intuition behind why $PARITY \notin AC_2^0$ (AC_k^0 is the subclass of AC^0 circuits that can be realized in depth k, where k is a constant). $PARITY$ on n inputs has 2^{n-1} *minterms*. Let us look at the bottom level fan-in as shown in Fig 9.1 of a DNF formula for $PARITY$. We claim that the bottom level fan-in of the $\wedge$-gate has to be n. Why?

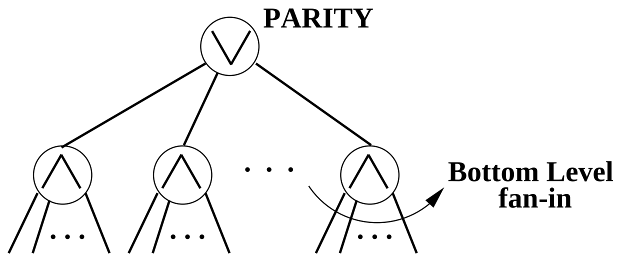

Figure 9.1: An AC_2^0 circuit for $PARITY$.

If the bottom level fan-in for some $\wedge$-gate is less than n, then the term corresponding to the $\wedge$-gate would have some missing input variable(s). Let us pick an assignment to the input variables that fires the $\wedge$-gate in question. In this assignment if one were to change the value of the missing variable, the $\wedge$-gate would continue to fire. This means that $PARITY$ would continue to be the same despite toggling some input variable!

Therefore, each bottom level $\wedge$-gate necessarily has to have a fan-in of n, which would mean that an exponential number of gates will be required to realize $PARITY$ in depth-2. The dual argument holds for the depth-2 realization of a CNF formula for $PARITY$.

Can this argument be extended to show $PARITY \notin AC_3^0$? Or, do you have some intuition for this?

9.1.1 Background

Ajtai ([1]) showed using logic that $PARITY \notin AC_3^0$. *Furst, Saxe & Sipser* ([10]) were the first to show $PARITY \notin AC^0$. They used the notion of *random restrictions* to show that any constant-depth circuit for $PARITY$ cannot be polynomial sized. *Yao* ([22]) went on to show that an exponential size circuit would be required to realize $PARITY$ in constant depth. *Hastad* in his seminal work ([11]) showed a *near optimal bound* (i.e. the upper and lower bounds differ by a constant factor), using the *Switching Lemma* reported in ([11]).

Razborov ([15]) showed the same result as *Hastad*'s using algebraic techniques which he christened *approximations*. *Smolensky* ([19]) improved these results. $AC^0[mod(q)]$ is the class of AC^0 circuits enhanced with $mod(q)$-gates. *Smolensky* showed that if one wants to do $mod(p)$ using $\{\wedge, \vee, mod(q)\}$ gates, and if p and q are relatively prime, then $mod(p) \notin AC^0[mod(q)]$.

While we are on lower bounds, the other known bounds are that $TC_2^0 \neq TC_3^0$. That is, there are some functions in TC_3^0 that are not in TC_2^0. The current status is that one does not know $TC_3^0 \neq TC_4^0$?

Razborov ([16]) simplified the *Switching Lemma* using counting arguments. *Laplante & Fortnow* ([9]) have provided a commentary of the *Switching Lemma* via *Kolmogorov Complexity*. *Beame*'s commentary ([4]) on the *Switching Lemma* uses *Decision Trees*. This latter commentary is what we shall follow.

9.2 Random Restrictions

Furst, Saxe & Sipser ([10]) had the following intuition. Assume that there exists a polynomial-size constant depth circuit for $PARITY$. Look at the bottom 3 levels of a constant depth-k circuit for $PARITY$, as shown in Fig 9.2.

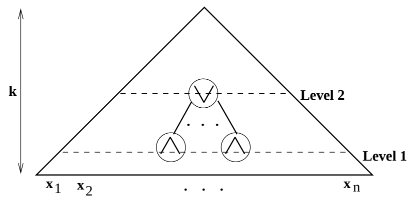

Figure 9.2: A depth-k AC^0 circuit for $PARITY$.

Suppose one applies a *partial assignment* to the circuit, i.e. randomly assign some variables 0, 1 values leaving some others untouched. The *partial assignment* should meet the following objectives:-

(1) Simplification of the formula to reduce the depth from k to $(k-1)$.
(2) The size of the circuit should not increase by much.
(3) There should still be enough variables left untouched.

If one were able to repeatedly apply this process adhering to the above objectives, then one would ultimately reach a depth of 2. In the above process, the size of the circuit after each step has not increased too much. And, the number of steps is a constant, $(k-2)$. Therefore, the circuit for depth 2 that results will be polynomial-sized. But, we already know that there cannot be a polynomial-size constant depth-2 circuit for $PARITY$. And, this contradicts our earlier intuition. Hence, there cannot be a polynomial-size constant depth circuit for $PARITY$.

The intuition will be transformed to truth if one can show the existence of partial assignments that meet the above objectives. This whole notion is termed **random restrictions** ([10]). The number of variables that should survive a *random restriction* is $\sqrt{n}$. This can be achieved if one uses the following random scheme for partial assignment.

$$\left.\begin{array}{l} Pr(X\ survives) = \frac{1}{\sqrt{n}} \\ Pr(X=1) = \frac{1}{2}(1-\frac{1}{\sqrt{n}}) \\ Pr(X=0) = \frac{1}{2}(1-\frac{1}{\sqrt{n}}) \end{array}\right\} \Rightarrow \ Expected\ number\ of\ variables\ untouched\ is\ \sqrt{n}.$$

Consider a wide fan-in gate (i.e. fan-in $> c\log n$) at level 1. Even if a single 0 comes on any of its inputs, it will vanish in the simplification. That is, the probability that a wide fan-in gate stays on to the next step is the probability that no 0 occurs on any of its inputs. This probability is $\left(1-\frac{1}{2}(1-\frac{1}{\sqrt{n}})\right)^{c\log n}$. This probability is very small, and therefore the wide gates will for all practical purposes vanish.

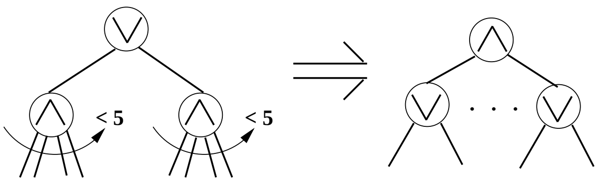

Figure 9.3: A surviving $\wedge - \vee$ block.

Let us now consider the narrow gates (fan-in $< c\log n$). The above argument will not work. Let us look at the survival of a narrow gate as a bounded (for some constant c) fan-in gate. $Pr(\ number\ of\ surviving\ variables\ \geq c) \leq \frac{1}{n^c}$. That is, the gates that do survive will have bounded fan-in (say, < 5) as shown in Fig 9.3. An $\vee$ of $\wedge$s of constant size, can be converted into an $\wedge$ of $\vee$s using standard Boolean Algebra. This would then imply that the $(k-2)^{th}$ and the $(k-1)^{th}$ layers will be $\wedge$s. These can be merged into a single layer, thus reducing the depth of the circuit to $(k-1)$.

9.3 The Switching Lemma

The basic idea in the *random restrictions* technique is to randomly set the variables so that a constant sized circuit can be replaced by some other equivalent one. The $\wedge$ of $\vee$s get *switched* to an $\vee$ of $\wedge$s and vice-versa. *Hastad* ([11]) generalized the conditions under which such restrictions are applicable, into the *Switching Lemma.*

Suppose one is given a DNF formula, where each term has $\leq r$ variables, and a restriction where the probability of a variable surviving is p, is applied to it. One cannot *switch* to a CNF formula, where each clause has size $\leq s$, with a probability $\leq (5pr)^s$. *Beame* ([4]) showed that this probability is $\leq (7pr)^s$ using simpler techniques via *Decision Trees.*

9.3.1 Decision Trees Approach

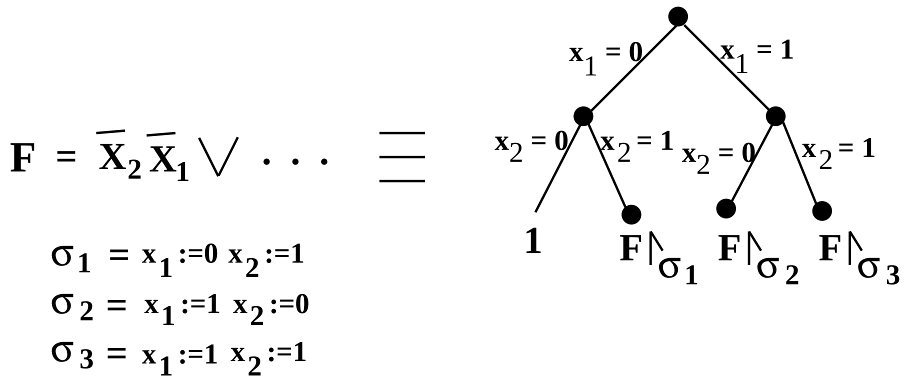

Figure 9.4: A Constructive Example of a Canonical Decision Tree.

One is given a DNF formula $F = t_1 \vee t_2 \vee \ldots \vee t_k$ where each term t_i has $\leq r$ variables. If one were to define an ordering on the variables, and also an ordering on all the terms, then there would be a unique way of writing down the formula. Now, if σ is a *partial assignment* to the variables, then $F \upharpoonright_\sigma$ is the formula that results by restricting F to σ. The *canonical decision tree* for F is inductively defined as,

- If F is a constant 0 or 1, then a single leaf node is the tree.
- Define $F = t_1 \vee F'$. Take the variables occurring in t_1 and draw a complete tree. Each leaf l_σ in the tree corresponds to a partial assignment σ. At l_σ one suspends the *canonical decision tree* corresponding to the formula $F \upharpoonright_\sigma$, i.e. F restricted on σ. There will be a single restriction corresponding to t_1 that makes the formula F go to 1. On all other σs, $F \upharpoonright_\sigma = F' \upharpoonright_\sigma$ (Why?).

A sample *canonical decision tree* is shown in Fig 9.4. Note that, on any path in the *canonical decision tree*, the variables will appear in the same order.

Consider a path, $y_1 \rightarrow y_2 \rightarrow \ldots \rightarrow y_l$, in the *decision tree* terminating in a 0. That is, $y_1 \cdot y_2 \cdot \ldots \cdot y_l = 0$, which implies $\bar{y_1} \vee \bar{y_2} \vee \ldots \vee \bar{y_l} = 1$. This is a clause in CNF. Therefore, looking at all paths that terminate at 0, one can get a CNF formula for the *decision tree*. But, the problem is with the depth of the *decision tree* (i.e. size of each clause). One wants the size of each clause to be $\leq s$. And, one has already seen a method for simplifying formulas.

Suppose, one hits the *decision tree* for F with a *random restriction* in which $\sqrt{n}$ variables survive. One gets a new formula F' with fewer variables and, whose depth (i.e. *decision tree* depth) is at most s. This would pave the way for the *switch* to a CNF formula. Note that, the formula F' is not the original one but a *restricted* one. Therefore, one has to assert that by hitting with a *random restriction*, one does indeed get a *decision tree* for F' with depth $\leq s$.

Let us look at *restrictions* on n variables with l unset and the remaining (i.e. $(n-l)$ variables) set to 0/1. Let R_n^l denote the set of all such *random restrictions*. $|R_n^l| = \binom{n}{l} \cdot 2^{n-l}$. Let S denote the set of all *bad* restrictions (i.e. *random restrictions* which result in a *decision tree* of height $> s$). The *Switching Lemma* avers that

$$\frac{|S|}{|R_n^l|} \leq (7 \cdot \frac{l}{n} \cdot r)^s$$

9.3.2 Proving the Switching Lemma

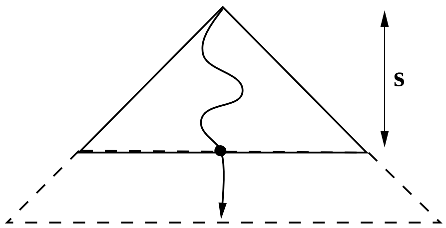

Figure 9.5: Effect of a bad restriction.

One needs to be able to count the number of *bad restrictions* i.e. bound $|S|$. Consider a $\rho \in S$. ρ would have forced the *decision tree* to be of depth $> s$ as shown in Fig 9.5. It is difficult to extract a property of ρ that could cause this and therefore, counting S seems rather difficult through straightforward means. Hence, the basic idea of the proof is to define a *one-to-one* map from S to some other set D such that $|S| \leq |D|$. Now, D is a nice set in the sense that one can count it.

Let us consider a bad path π, i.e. a path of length $> s$ in the decision tree for $F \upharpoonright_\rho$. One shall consider the initial segment of π such that $|\pi| = s$. This would restrict $F \upharpoonright_\rho$ to k terms,

$F\restriction_\rho = \hat{t_1} \vee \hat{t_2} \vee \ldots \vee \hat{t_k}$. Now, for each *level* in the decision tree for $F\restriction_\rho$, one has a segment π_i of π. Each π_i fails to satisfy the term $\hat{t_i}$. That is, $\pi = \pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_k$. And, corresponding to each π_i one has a path (partial assignment) σ_i that satisfies $\hat{t_i}$. Fig 9.6 illustrates this correspondence.

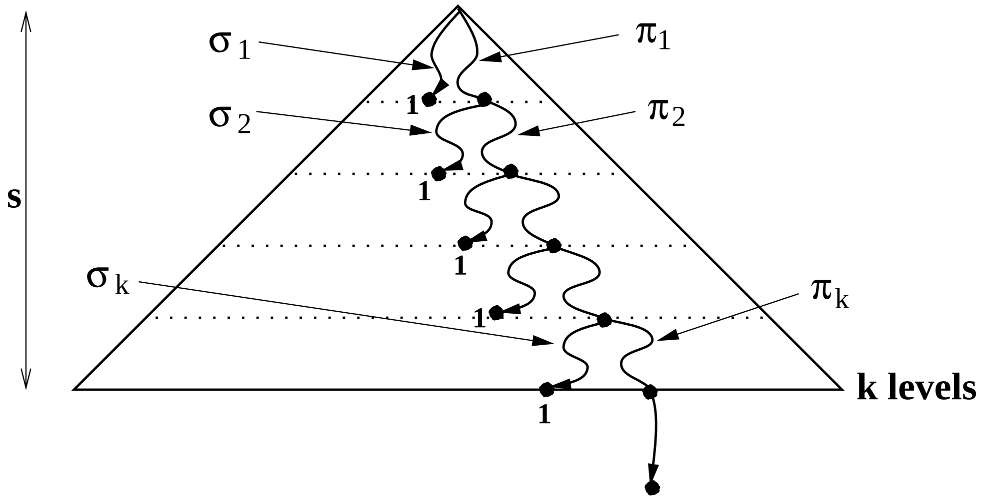

Figure 9.6: $F\restriction_\rho$ split into k levels.

$F\restriction_{\rho\cdot\pi_1}$, the formula that results when F is restricted to $\rho\cdot\pi_1$, would cause some term to vanish. $F\restriction_{\rho\cdot\pi}$ would give a decision tree of depth at most $(l-s)$. That is, $\rho\cdot\pi$ would give a restriction from R_n^{l-s}. The *one-to-one* correspondence that one is looking for is $S \xrightarrow{1-1} R_n^{l-s} \times D$. D has yet to be defined.

Essentially, the *one-to-one* correspondence should be such that, given a $\rho \in S$, one should find a unique element from $R_n^{l-s} \times D$. That is, ρ restricts F to the formula $F\restriction_\rho$. The *partial assignments* $\sigma_1, \sigma_2, \ldots, \sigma_k$ each satisfy a term resulting from the successive application of the *partial assignments* $\pi_1, \pi_2, \ldots, \pi_k$. One has available in hand F and ρ, and hence, the σ_is. Therefore, in order to go to π, if one can maintain information about how each σ_i differs from π_i, it would be possible to uniquely identify π. And, hence a unique tuple from $R_n^{l-s} \times D$ can be associated with ρ..

Towards showing the *one-to-one* correspondence, one shall define $\beta_i \in \{0, 1, -1\}^r$ for each term in F with the following interpretation

$$\beta_{ij} = \begin{cases} 0, & j^{th} \ var \ of \ i^{th} \ term \ is \ not \ active. \\ 1, & j^{th} \ var \ takes \ same \ value \ as \ in \ \pi_i. \\ -1, & j^{th} \ var \ takes \ different \ value \ from \ that \ in \ \pi_i. \end{cases} \qquad \forall j \in [1, r]$$

The total number of 1s and -1s in all of the β_is put together is s, since all the π_is put together set s variables. Once a variable is set to $1/-1$ in β_i, it will not get set in subsequent $\beta_j, j > i$.

These βs constitute the set D. Let us estimate the size of D. If i variables are set in β_1, then the rest of the βs together can set $s-i$ variables to $1/-1$. That is the following recurrence relation gives us the an estimate of $|D|$,

$$|D| = |Beta(s)| = \left(\sum_{i=0}^{min(r,s)} \binom{r}{i} \cdot |Beta(s-i)| \right) \cdot 2^s$$

Beame ([4]) shows that the above recurrence is upper bounded by $(\frac{r}{\ln 2})^s \cdot 2^s$. This leads us to the following bound on $|S|$.

$$|S| \leq |R_n^{l-s}| \cdot |D| \leq \binom{n}{l-s} \cdot 2^{n-l+s} \cdot \left(\frac{r}{\ln 2}\right)^s \cdot 2^s$$

$$\Rightarrow \frac{|S|}{|R_n^l|} \leq \frac{\binom{n}{l-s} \cdot 2^{n-l+s}}{\binom{n}{l} \cdot 2^{n-l}} \cdot \left(\frac{2r}{\ln 2}\right)^s$$

$$= \frac{(n-l)! \cdot l! \cdot 2^s}{(n-l+s)! \cdot (l-s)!} \cdot \left(\frac{2r}{\ln 2}\right)^s = \frac{l \cdot (l-1) \cdot \ldots \cdot (l-s+1)}{(n-l+s) \cdot (n-l+s-1) \cdot \ldots \cdot (n-l+1)} \cdot \left(\frac{4r}{\ln 2}\right)^s$$

$$\Rightarrow \frac{|S|}{|R_n^l|} \leq \left(\frac{l}{n-l}\right)^s \cdot \left(\frac{4r}{\ln 2}\right)^s \leq \left(\frac{4lr}{(n-l) \cdot \ln 2}\right)^s$$

Setting $p = \frac{l}{n}$, we have

$$\frac{|S|}{|R_n^l|} \leq \left(\frac{4 \cdot p \cdot r}{(1-p) \cdot \ln 2}\right)^s$$

It is known that for $p < \frac{1}{7}$, $\frac{|S|}{|R_n^l|} \leq (7 \cdot p \cdot r)^s$. ([4])

One has proved the *Switching Lemma* which says that given a DNF formula F, where each term is of size at most r, then one **cannot** convert it into a CNF formula F', having clauses of size at most s, via a *random restriction* with probability $\leq (7 \cdot p \cdot r)^s$, where $p = \frac{l}{n}$ is the probability of a variable being *unset* in the restriction. A tighter analysis would show that the fraction of *bad restrictions* in R_n^l is $\leq (5 \cdot p \cdot r)^s$.

9.4 Switching Lemma on *PARITY*

Recall that any *depth-2* circuit computing *PARITY* has gates in the bottom level that all have a fan-in of n. Using this observation and the *Switching Lemma*, we shall venture to show $PARITY \notin AC^0$. In fact, we shall show that a *constant depth* circuit with the following constraints cannot compute *PARITY*. Fig 9.7 illustrates these constraints.

- The number of gates in the bottom 2 layers does not matter.

- There is a restriction on the size of the circuit in the first $(k-2)$ layers. The size should be $\leq 2^{\frac{1}{10}\cdot n^{\frac{1}{k-1}}}$
- The bottom level gate fan-ins is to be at most $r = \frac{1}{10}\cdot n^{\frac{1}{k-1}}$.

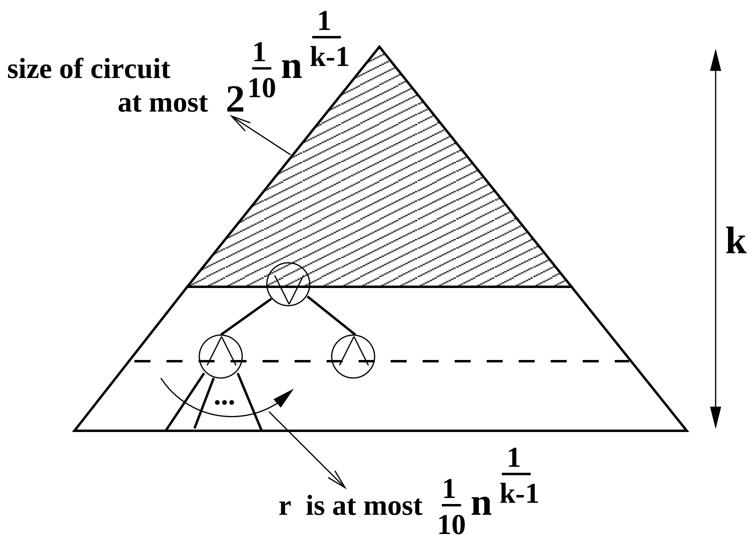

Figure 9.7: Circuits that cannot compute $PARITY$.

We shall use induction on the depth of the circuit to prove that $PARITY$ cannot be computed by circuits of the above kind. The **base case**, when $k = 2$, would imply that $r = \frac{n}{10}$, i.e. the bottom level gates can have a fan-in of at most $\frac{n}{10}$. Such circuits cannot compute $PARITY$ since we already know that any depth-2 circuit for $PARITY$ necessarily has to have a bottom level fan-in of n.

Moving on to the **inductive hypothesis**, assume that circuits of the above kind having depth $(k-1)$ cannot compute $PARITY$. The approach followed for the **inductive step** is

1. Take a *depth-k* circuit conforming to the circuit constraints and hit it with a random restriction. The *Switching Lemma* assures us that with high probability one would be able to switch the bottom 2 layers. This would allow one to merge the $(k-2)^{th}$ and the $(k-1)^{th}$ layers and hence reduce the depth of the circuit to $(k-1)$.
2. One needs to then show that the reduced circuit that results satisfies all the initial conditions to permit the application of the lemma for the *depth-(k-1)* circuit.

Let us set $s = \frac{n^{\frac{1}{k-1}}}{10}$ and $p = n^{-\frac{1}{k-1}}$. The *Switching Lemma* tells us that the fraction of *bad restrictions* is $\leq (5\cdot p\cdot r)^s = (\frac{1}{2})^s$. The size of the circuit upto the $(k-2)^{th}$ level is at most 2^s.

That is, the number of sub-circuits at the $(k-1)^{th}$ level is at most 2^s. This implies that the probability of a restriction being *bad* for some sub-circuit is $\leq 2^s \cdot (\frac{1}{2})^s \leq 1$. That is, there will be a restriction that works for all the sub-circuits.

Now, the number of variables that are unset in a *good restriction* is l, given by

$$l = p \cdot n = n^{\frac{k-2}{k-1}}$$

$$\Rightarrow r = s = \frac{1}{10} \cdot n^{\frac{1}{k-1}} = \frac{1}{10} \cdot n^{\frac{k-2}{k-1} \times \frac{1}{k-2}} = \frac{1}{10} \cdot l^{\frac{1}{k-2}}$$

The number of variables that continue on in the reduced *depth-(k-1)* circuit is l, the bottom level fan-ins, r, is proper, and so is the size of the circuit upto the $(k-3)^{th}$ level. That is, everything is set for the induction to go through. But, the induction hypothesis assumes that no *depth-(k-1)* circuit of the above kind can compute $PARITY$. Hence, no constant depth circuit with *large enough bottom fan-in* can compute $PARITY$. One needs to show this for any AC^0 circuit.

Theorem 6 *No* depth-k *circuit of size* $\leq 2^{(\frac{1}{10})^{\frac{k}{k-1}} \cdot n^{\frac{1}{k-1}}}$ *can compute* $PARITY$*.*

Proof: We shall reduce to the previous setting.

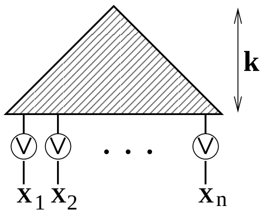

Figure 9.8: Augmented circuit of depth $(k+1)$.

Consider the augmented circuit in Fig 9.8. The depth has increased to $(k+1)$. Set $p = \frac{1}{10}, r = 1, s = (\frac{1}{10})^{\frac{k}{k-1}} \cdot n^{\frac{1}{k-1}}$. The number of sub-circuits is at most 2^s and the fraction of restrictions that are *bad* is $\leq (\frac{1}{2})^s$. That is, the total number of *bad restrictions* for the circuit in Fig 9.8 is $\leq 2^s \cdot (\frac{1}{2})^s \leq 1$. This implies there exists a restriction ρ that can be applied to the circuit with the following parameters.

$$s = (\frac{1}{10})^{\frac{k}{k-1}} \cdot n^{\frac{1}{k-1}}; \quad l = \frac{n}{10}$$

$$\Rightarrow s = \left(\frac{1}{10}\right)^{\frac{k}{k-1}} \cdot (10 \cdot l)^{\frac{1}{k-1}} = \frac{1}{10} \cdot l^{\frac{1}{k-1}}$$

A single random restriction has brought us to the previous setting where the bottom level fan-in is $s = \frac{1}{10} \cdot l^{\frac{1}{k-1}}$ and the size of the circuit upto the $(k-2)^{th}$ layer is at most 2^s. This we know cannot compute $PARITY$. ■

Hastad's ([11]) bound of $2^{O(n^{\frac{1}{k}})}$ is the best known bound for $PARITY$. *Yao* ([22]) showed a bound of $2^{O(n^{\frac{1}{4k}})}$, while *Razborov* ([16]) had the bound $2^{O(n^{\frac{1}{2k}})}$. The constant that is present in the exponent of the size for a *depth-3* realization of $PARITY$ is known exactly. This is not known for a *depth-4* realization.

Exercise: Show that a depth k circuit, where $k = O(\frac{\log n}{\log\log n})$, the size of the circuit { $2^{(\frac{1}{10})^{\frac{k}{k-1}} \cdot n^{\frac{1}{k-1}}}$ } becomes a *polynomial*. Also, show a polynomial-sized realization of depth $\frac{\log n}{\log\log n}$ for $PARITY$.

Before we wind up, we shall show that $PARITY \notin AC^0 \Rightarrow MAJ \notin AC^0$. Now, a function $f(x_1, x_2, \ldots, x_n)$ is a *symmetric* function if the value of the function depends only on the number of 1s in the input, i.e. f depends on $\sum_i x_i$. This means that f can be represented as a *value vector* of size $(n+1)$, i.e. $\langle v_0, v_1, \ldots, v_n\rangle$. The *value vector* can be represented as a **step function** as shown in Fig 9.9.

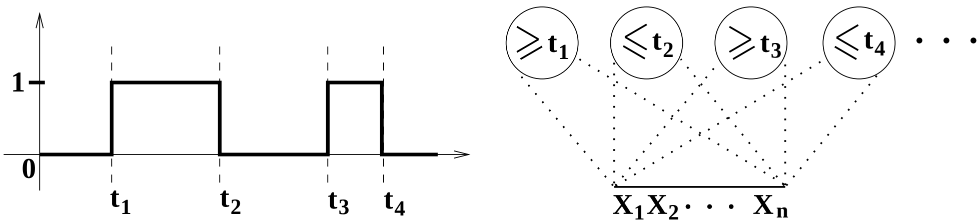

Figure 9.9: Representing a symmetric function as a step function.

If an input falls in the **one region** then $(k+1)$ of the gates shown in the right of Fig 9.9 will fire. While, if the input falls in the **zero region** then only k will fire. Therefore, doing a simple **majority** on the gates will give us our required realization. The depth of the circuit is 2.

A *symmetric function* having k *one regions* can be realized through a TC_2^0 circuit of size $(2k+1)$. The worst case occurs when $k = \frac{n}{2}$, i.e. for $PARITY$. This shows that a *threshold circuit of size at most* $(n+1)$ *realizes* $PARITY$.

Now, if $MAJ \in AC^0$ then it would imply that $PARITY \in AC^0$. Therefore, $MAJ \notin AC^0$. One can show a lot of other problems as not being in AC^0. For instance, $CLIQUE$ is an important problem that we shall show later on as not having a **monotone realization**.The proof technique that has been employed to show that $PARITY \notin AC^0$ basically revolves around showing that some bottom-level gate necessarily has to have a large fan-in. If one can show the same for some other problem, then the $PARITY \notin AC^0$ proof can be cannibalized.

Computational Complexity Theory **Spring 1997**

Lecture 17-18 : March 25 & 27 1997

Lecturer: V. Vinay *Scribe: P. R. Subramanya*

We shall continue in our $PARITY \notin AC^0$ vein. Here we shall discuss *Razborov's Method of Approximations* ([15]) and *Voting Polynomials* by *Aspens, Beigel, Furst & Rudich* ([2]).

10.1 Introduction

Razborov introduced his *method of approximations* ([15], [16]) in the context of *monotone Boolean functions*. The starting point for this work is a theorem by *Shannon*. It says that any random Boolean function requires a circuit of size $\geq \frac{2^n}{n}$. This was an existence result. No actual function was known for a long time that required $\geq \frac{2^n}{n}$ gates. *Razborov* ([15],[16]) was the first to demonstrate that a *monotone circuit* for $CLIQUE$ requires at least $2^{\Omega(\sqrt{n})}$ gates. *Razborov* extended this bound to an AC^0 realization for $CLIQUE$.

The basic idea in the *method of approximations*, as the name suggests, is to approximate each gate. That is, instead of representing a gate by an exact polynomial, one allows for some error to be introduced by each gate. One does this approximation with the intention of arriving at a small degree polynomial that represents the function computed by the circuit. By a small degree polynomial, one means a polynomial with a degree of the order of $\sqrt{n}$. Therefore, if the approximations are good and effective, then the polynomial that results for a circuit will have a small degree.

Let us now come to the question of how the *method of approximations* can be used to prove lower bounds. Assuming that each gate introduces an error e, a circuit C would have a total error that is bounded by $size(C) \cdot e$. Therefore, knowing e if one can show that a circuit has a large error, then the size of the circuit has to be large.

The approach used for proving that $PARITY \notin AC^0$ is to show that a small degree polynomial for $PARITY$ has $\geq \frac{1}{50} \cdot 2^n$ errors. This would imply that any circuit realization for $PARITY$ has to be *exponential-sized.*

10.2 Method of Approximations

Given a Boolean Circuit C, one arithmetizes it in the standard way as shown in Fig 10.1. Note that, the $\vee$-gate's arithmetization can be got via *De Morgan's Law*, i.e. $\neg$ of the $\wedge$-gate. This scheme of arithmetization leads to a representation of an AC^0 circuit as a low-degree polynomial (i.e. $\sqrt{n}$-degree) over an appropriate field ([15], [19]).

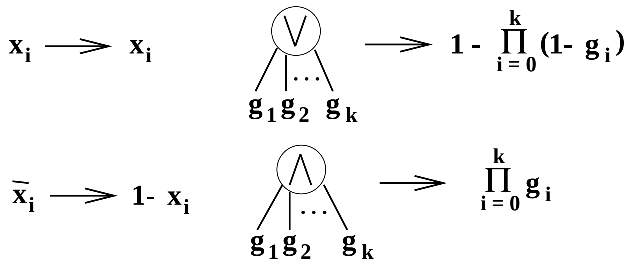

Figure 10.1: Arithmetization of a Boolean Circuit

The claim is that any $\sqrt{n}$-degree polynomial cannot represent $PARITY$ on a large fraction of the points. In the rest of this section, we shall prove this claim.

Let us look at the effect of each gate on the degree of the polynomial that results. Assume that the degree of an $\wedge$-gate is k and its inputs are $g_1, g_2, \ldots, g_k$. An $\wedge$-gate, as shown in Fig 10.1, would give a degree of $\leq k \cdot max_k(degree(g_i))$. This means that at each level the degree of the polynomial grows by some factor (the maximum fan-in of a gate in that level). And, since one is dealing with AC^0 circuits, the degree can be at most polynomial.

Let us now consider the approximation of an $\vee$-gate. One is now working over a field $\mathbb{Z}_p$ and recollect *Fermat's Little Theorem*, $a^{p-1} \equiv 1 mod(p)$. The inputs to the $\vee$-gate are $x_1, x_2, \ldots, x_n$. Define,

$$f(x_1, x_2, \ldots, x_n) = \left(\sum_{i=1}^{n} r_i \cdot x_i \right)^{p-1} \qquad r_i \in \mathbb{Z}_p$$

r_i is a random element from $\mathbb{Z}_p$ and plays the role of selecting the input. The behaviour of f can be understood by considering the following 2 cases.

- **Case i:** $x_1 = x_2 = \ldots = x_n = 0 \Rightarrow f(x_1, x_2, \ldots, x_n) = 0$. f behaves exactly as an $\vee$ on this input point.
- **Case ii:** There is at least one 1 in the input. That is, $\exists j \cdot x_j = 1$. f can be rewritten as,

  $$f(x_1, x_2, \ldots, x_n) = \left(r_j \cdot x_j + \sum_{i=1, i \neq j}^{n} r_i \cdot x_i \right)^{p-1} = (r_j + q)^{p-1}$$

  The $\vee$-gate on these inputs outputs a 1 and we would also like f to behave in the same manner. This will happen in most cases except when $r_j \equiv -q(mod(p))$. And, this happens with probability $\frac{1}{p} \leq \frac{1}{2}$.

The above approximation of an $\vee$-gate generates a fixed degree polynomial since p is fixed. Given an input to f, it is likely to make an error with a probability $\leq \frac{1}{2}$. If one were to repeat the experiment l times, and then do the *proper* $\vee$ (i.e. using the arithmetization stated in Fig 10.1), then the probability of being wrong on an $\vee$ is $\leq \frac{1}{2^l}$. That is, the number of points on which one is likely to be wrong is $\leq 2^{n-l}$. The degree of the polynomial that results is $(p-1) \cdot l$.

One needs to note a subtlety on the base of errors. Consider an internal gate as shown in Fig 10.2. The gate seems to have 2^k settings based on the values that its inputs $g_1, g_2, \ldots, g_k$ can take. Each of these inputs eventually get their settings from the inputs $x_1, x_2, \ldots, x_n$. Therefore, for every gate there are 2^n possible settings and the errors come from these.

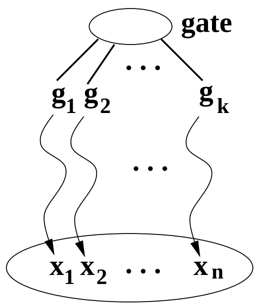

Figure 10.2: An internal gate in the circuit.

Let us now look at the total number of errors that an AC^0 circuit C of *size* s and *depth* d can commit. The $\neg$-gate does not commit any errors while the $\vee$-gate makes errors.

$$errors(C) \leq s \cdot 2^{n-l} \quad and \quad degree(C) \leq ((p-1) \cdot l)^d$$

Note that, our objective being a *low-degree* polynomial approximating C, we need to select p and l appropriately. Setting,

$$p = 3 \quad \Rightarrow degree(C) \leq (2l)^d \quad \Rightarrow (2l)^d = \sqrt{n} \quad \Rightarrow l = \frac{1}{2} \cdot n^{\frac{1}{2d}}$$

This would imply that the errors committed by C is $\leq \frac{s \cdot 2^n}{2^{\frac{1}{2}n^{\frac{1}{2d}}}}$. That is, if we can now show that any $\sqrt{n}$-degree polynomial on $\mathbb{Z}_3$ differs from $PARITY$ on $\geq \frac{1}{50} \cdot 2^n$ points, then we can conclude that

$$\frac{s \cdot 2^n}{2^{\frac{1}{2}n^{\frac{1}{2d}}}} \geq \frac{1}{50} \cdot 2^n \quad \Rightarrow s \geq \frac{1}{50} \cdot 2^{\frac{1}{2}n^{\frac{1}{2d}}}$$

This would then allow us to legitimately conclude that $PARITY \notin AC^0[mod(p)]$ and, specifically that $PARITY \notin AC^0[mod(3)]$. In the next subsection, we shall give an outline of the errors lower bound for $PARITY$ in the context of *Razborov's method of approximations.*

10.2.1 Errors bound for $PARITY$

In order to show the lower bound for errors made by any $\sqrt{n}$-degree polynomial for $PARITY$, we shall move over to the $\{1,-1\}$ domain ([7]). We shall use a *one-to-one onto invertible map* $\{0,1\} \longrightarrow \{1,-1\}$. The mapping is $x_i \longrightarrow (1-2x_i) = y_i$. There exists a function

$$PARITY(y_1, y_2, \ldots, y_n) = \prod_{i=1}^{n} y_i$$

that computes $PARITY$ exactly. x_i can be recovered from y_i using $x_i = \frac{1-y_i}{2}$. Now since we are working in $\mathbb{Z}_3$, the multiplicative inverse of 2 is 2 itself. Hence, x_i can be recovered using $x_i = 2 \cdot (1-y_i)$ in $\mathbb{Z}_3$.

Let $f(x_1, x_2, \ldots, x_n)$ be our $\sqrt{n}$-degree approximate polynomial for $PARITY$. Define A as the set of input points where the functions $PARITY(\ldots)$ and $f(\ldots)$ agree, i.e. $A = \{x_1 x_2 \ldots x_n : PARITY(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_n)\}$. We shall look at this set A of *agreements* and estimate its size through the use of $\mathcal{F}_A$, the collection of mappings $g : A \longrightarrow \{0,1,2\}$. $|\mathcal{F}_A| = 3^{|A|}$ and therefore if we can show that $|\mathcal{F}_A|$ is small, i.e. $|\mathcal{F}_A| \leq 3^q$, then $|A| \leq q$, i.e. the number of points on which $PARITY$ and the $\sqrt{n}$-degree approximator f disagree is $\geq 2^n - q$.

The technique that we shall follow for showing $|\mathcal{F}_A|$ is small is to take a $g \in \mathcal{F}_A$ and assign it to a different polynomial of degree at most $\frac{n}{2} + \frac{\sqrt{n}}{2}$. We then estimate the number of such polynomials.

Look at a polynomial in $\mathbb{Z}_3$. Each term in the polynomial will be of the form $c \cdot x_{i_1} \cdot x_{i_2} \cdot \ldots \cdot x_{i_l}$ with $c \in \{1,-1\}$. That is, it will be a **multilinear monomial** since each $x_i \in \{1,-1\} \Rightarrow {x_i}^2 = 1$ and any ${x_i}^k$ can be replaced by x_i. Hence, our *polynomial* will consist of monomials $m_1, m_2, \ldots, m_l$.

Let us consider a monomial m. If its degree is $\geq \frac{n}{2} + \frac{\sqrt{n}}{2}$, then $m = y_1 \cdot y_2 \cdot \ldots \cdot y_k$ with $k \geq \frac{n}{2} + \frac{\sqrt{n}}{2}$. Let us multiply m by $(y_{k+1} \cdot \ldots \cdot y_n)^2 = 1$,

$$y_1 \cdot y_2 \cdot \ldots \cdot y_k (y_{k+1} \cdot \ldots \cdot y_n)^2 = y_1 \cdot y_2 \cdot \ldots \cdot y_n (y_{k+1} \cdot \ldots \cdot y_n)$$

For the input points in A, we can rewrite this as,

$$= PARITY(y_1, y_2, \ldots, y_n) * (y_{k+1} \cdot \ldots \cdot y_n) = f(x_1, x_2, \ldots, x_n) * (y_{k+1} \cdot \ldots \cdot y_n)$$

Note that this substitution holds for the input points in A alone. Hence, we have managed to show that the degree of our large degree monomial m will now be convertible to one of degree $\leq \sqrt{n} + \frac{n}{2} - \frac{\sqrt{n}}{2} \leq \sqrt{n} + \frac{\sqrt{n}}{2}$. That is, any monomial can be converted into one with less than this degree as long as one is within A. Therefore, the number of monomials of degree $\leq \frac{n}{2} + \frac{\sqrt{n}}{2}$ is $\sum_{i=0}^{\frac{n}{2}+\frac{\sqrt{n}}{2}} \begin{pmatrix} n \\ i \end{pmatrix} < \frac{49}{50} \cdot 2^n$.

Therefore, the number of different polynomials consisting of monomials of degree $\leq \frac{n}{2} + \frac{\sqrt{n}}{2}$ is $< 3^{\frac{49}{50} \cdot 2^n}$. Hence, $|A| < \frac{49}{50} \cdot 2^n$. That is, the number of input points on which any $\sqrt{n}$-degree polynomial approximating $PARITY(\ldots)$ makes an error is $\geq 2^n - \frac{49}{50} \cdot 2^n = \frac{1}{50} \cdot 2^n$.

10.3 Voting Polynomials

Smolensky ([19]) showed that the approximation of $\vee$ by a $\sqrt{n}$-degree polynomial over the field $\mathbb{Z}_3$ is going to differ from $PARITY$ on many points. *Tarui* ([21]) showed the approximation of $\vee$ over the reals. *Aspens et al* ([2]) used *Tarui*'s result to show that $PARITY$ cannot be realized by a circuit of the form shown in Fig 10.3. They used a technique called *Voting Polynomials*. This shall be subject of discussion for the rest of the lecture.

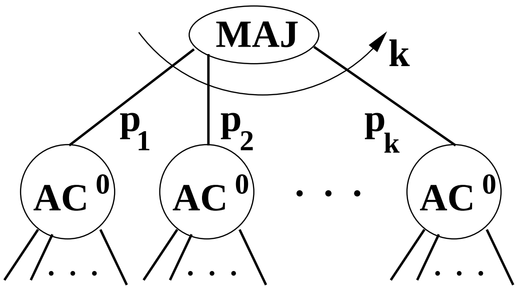

Figure 10.3: Circuits that cannot realize $PARITY$.

10.3.1 Voting Puzzle

Let us consider the following *voting puzzle*. There are an **odd** number of people $(2n+1)$ sitting in a round table. Each has a bit on his/her forehead. So, each can see the bit on the foreheads of all the others except his/her own's. Each person casts a private vote on the $PARITY$ of the bits on the foreheads of all the people. The voters are not allowed to consult or communicate with each other. The *majority* vote is considered as the collective vote on the $PARITY$. What should be each person's voting strategy? Note that, every person has to be wrong at least $\frac{1}{2}$ the number of times (Do you see why?).

It is tempting to conclude that the voters can collectively be correct only **half** the time. The following strategy surprisingly gives the voters a very high probability of being correct. ([2])

> A person seeing an equal number of 0's and 1's, casts a vote of 0. Else, the person assumes that the bit on his/her forehead is the same as the majority bit on the foreheads of others; and then casts a vote accordingly.

Let us analyze this voting strategy. When the number of 0s and 1s differ by *more than one*, each person casts a vote based on the assumption that the bit on his/her forehead is the majority of the bits that he/she sees. Hence, the majority of the voters will be correct in assuming that the bit on their foreheads is indeed the same as bit that is in the majority. This voters would be correct about the $PARITY$ of the bits on their foreheads.

Consider the situation when the number of 0s is exactly *one more* than the number of 1s. As per the strategy, the majority of the voters will cast a vote of 0. If n is *odd*, then the voters will be all wrong. And, if n is *even*, then the voters are correct.

In the symmetric case when the number of 1s is exactly *one more* than the number of 0s, the majority of the voters will again vote 0. If n is *odd*, then the voters are correct. And, if n is *even*, then they are wrong. Therefore, depending on whether n is *odd or even*, the voters are wrong in one of the above 2 situations. Either of the above 2 cases arise in $\binom{2n+1}{n}$ of 2^{2n+1} cases. That is, the voters are wrong with probability $\frac{1}{\theta(\sqrt{n})}$.

There are two other variants of the above *Voting Puzzle*. In the Chicago style voting variant, each voter is allowed to cast as many votes as he/she wants. In the other variant, the number of voters is $\binom{n}{k}$ and each voter looking at just k of the n bits has to vote Chicago style on the $PARITY$ of the n bits. ([2])

10.3.2 Voting Polynomials

A polynomial $p(x)$ *approximates* a function $f(x)$ with error ϵ if $Prob_x\{sgn(p(x)) = f(x)\} \geq 1 - \epsilon$. A polynomial $w(x)$, where $w \neq 0$, *weakly represents* the function $f(x)$ if $w(x) \neq 0 \Rightarrow sgn(w(x)) = f(x)$. Define $d_w(f)$ to be the *degree of the smallest degree polynomial* that *weakly represents* f.

Claim 1 $d_w(PARITY) = n$.

Proof of Claim: We shall prove this by contradiction. Suppose $p(x)$ *weakly represents* $PARITY$ and the degree of $p(x)$ is $\leq (n-1)$. Consider, $\sum_{x \in \{\pm 1\}^n} p(x) \cdot PARITY(x) > 0$. This is true from the definition of $p(x)$ and the fact that $p(x) \neq 0$. The LHS can be rewritten as,

$$\sum_x c_i \cdot x_{i_1} \cdot x_{i_2} \cdot \ldots \cdot x_{i_l} \cdot \prod_i x_i = \sum c_i \cdot x_{i'_1} \cdot x_{i'_2} \cdot \ldots \cdot x_{i'_{l'}} = 0$$

The last identity is true because for each term in the summation, there are $2^{l'-1}$ settings that give a -1 and an equal number will give 1. Hence, we have arrived at a contradiction. ■

Lemma 4 *Suppose* $S \subseteq \{\pm 1\}^n$ *and* $|S| < \sum_{j=0}^{k} \binom{n}{j}$. *Then,* $\exists$ *a polynomial* $q(x)$ *of degree* $2k$ *and* $q(x) \geq 0$ *and* $q(x) \neq 0$, *such that* $q(x)$ vanishes *on* S. *That is,* $q(x) = 0$ *whenever* $x \in S$.

Proof: Let $p(x)$ be a polynomial of degree k. The number of terms that $p(x)$ could have is $= \binom{n}{k} + \binom{n}{k-1} + \ldots + \binom{n}{0}$. This is also equal to the number of coefficients in $p(x)$. The value of $p(x)$ on any input is a linear combination of these coefficients. Therefore, $p(x) = 0$ on all $x \in S$ is a homogeneous system of $|S|$ linear equations on $\sum_{i=0}^{k} \binom{n}{i}$ variables, as indicated in Fig 10.4.

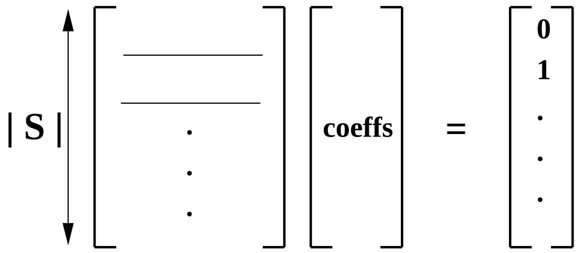

Figure 10.4: A homogeneous system of $|S|$ linear equations.

We know from Linear Algebra that there exists a non-trivial solution since $|S| < \sum_{i=0}^{k} \binom{n}{i}$. So, there exists a $p(x)$ which is 0 on all $x \in S$. Setting $q(x) = (p(x))^2$, we can ensure that $q(x) \geq 0$ on all x. ■

The main result of the paper ([2]) is the following theorem.

Theorem 7 *Let $p(x)$ be a polynomial of degree k and $f(x)$ be some function. If $k < d_w(f)$, then any polynomial $p(x)$ that approximates $f(x)$ must be in error on*

$$errors \geq \sum_{i=0}^{\lfloor \frac{d_w(f)-k-1}{2} \rfloor} \binom{n}{i}$$

Proof: We shall prove this by contradiction. Suppose not. That is,

$$|error\ set| < \sum_{i=0}^{\lfloor \frac{d_w(f)-k-1}{2} \rfloor} \binom{n}{i}$$

From the previous lemma, $\exists$ a polynomial $q(x)$ of degree $d_w(f) - k - 1$ such that $q(x) \geq 0$ and $q(x)$ vanishes on the *error set*. Therefore, $p(x) \cdot q(x)$ *weakly represents* $f(x)$ because $p(x) \cdot q(x) = 0 \quad \forall x \in error\ set$ and since $q(x) \geq 0 \quad \forall x$, $sgn(p(x)) = sgn(f(x))\ \forall x \notin error\ set$. Now, the degree of $p(x) \cdot q(x)$ is $k + d_w(f) - k - 1$, i.e. $d_w(f) - 1$. That is, one has a representation that has a smaller degree than $d_w(f)$! ■

We already saw that for $PARITY$, $d_w(PARITY) \geq n$. This implies that

$$|error\ set\ for\ PARITY| \geq \sum_{i=0}^{\lfloor \frac{d_w(f)-k-1}{2} \rfloor} \binom{n}{i}$$

In other words, lower the degree of the polynomial, higher the number of errors. If one wants the number of errors to be $\leq \frac{2^n}{100}$ then the degree has to be $\Omega(\sqrt{n})$. This comes from the error distribution shown in Fig 10.5.

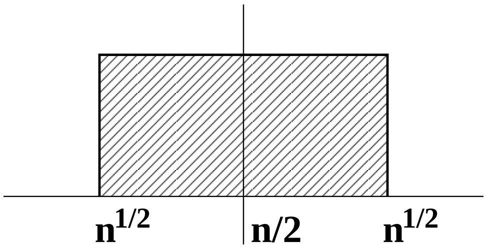

Figure 10.5: Error distribution for $PARITY$

10.4 Approximating $PARITY$

First let us consider the approximation for $\bigvee(x_1, x_2, \ldots, x_n)$. Initially, the input set is $S_0 = \{x_1, x_2, \ldots, x_n\}$ and define $p_0 = \sum_{x \in S_0} x_i$. Now, toss a coin to determine whether to keep a particular element of S_0 or not. One would get the set $S_1 = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ and $p_1 = \sum_{x \in S_1} x$. Repeat this experiment $k = \log n + 1$ times and collect the S_is and compute the p_is. That is, we would have a sequence of sets $S_0, S_1, \ldots, S_k$ and a sequence of numbers $p_0, p_1, \ldots, p_k$.

Claim 2 *There $\exists j$ with probability $\geq \frac{1}{3}$ such that $p_j = 1$ provided $\sum_i x_i > 0$.*

Proof of Claim: Let us consider the sequence $S_0 \longrightarrow S_1 \longrightarrow \ldots \longrightarrow S_k$. There are 3 cases that arise.

Case 1: $p_0 = 1$.
Case 2: $\exists i$ such that $p_{i-1} \geq 2 \bigwedge p_i \leq 1$.
Case 3: $p_k \geq 2$.

Case 3 occurs with probability $\leq n \cdot 2^k = \frac{1}{2}$. Case 2 has the following 2 situations:

$$Prob\{p_{i+1} = 1 | p_i \geq 2\} \leq \begin{pmatrix} p_i \\ 1 \end{pmatrix} \cdot \frac{1}{2} \cdot \frac{1}{2^{p_i - 1}} = \frac{p_i}{2^{p_i}}$$
$$Prob\{p_{i+1} = 0 | p_i \geq 2\} \leq \frac{1}{2^{p_i}}$$

These 2 can be combined to give,

$$Prob\{p_{i+1} = 1 | p_i \geq 2\} \leq \frac{\frac{p_i}{2^{p_i}}}{\frac{p_i}{2^{p_i}} + \frac{1}{2^{p_i}}} = \frac{p_i}{p_i + 1} \geq \frac{2}{3}$$

Cases 1 and 3 occur with probability $\geq \frac{1}{2}$ and hence, the probability that some $p_i = 1$ is $\geq \frac{1}{2} \times \frac{2}{3} \geq \frac{1}{3}$. ■

The claim would imply that if one were to write the polynomial $P(x)$ as,

$$P(x_1, x_2, \ldots, x_n) = 1 - \prod_{i=0}^{k} (1 - p_i)$$

then the following cases arise.

- **Case 1:** $x_1 = x_2 = \ldots = x_n = 0$. This implies $P(\ldots) = 0$.
- **Case 2:** $\exists j \cdot x_j = 1$. From the above claim, with probability $\geq \frac{1}{3}$, there $\exists i \cdot p_i = 1$. This implies that $P(\ldots) = 1 - 0 = 1$ with probability $\geq \frac{1}{3}$.

This is an approximation of $\bigvee(x_1, x_2, \ldots, x_n)$. The degree of $P(\ldots)$ is $O(\log n)$. Now, if one wants to reduce the error to be $\leq \epsilon$, then one can make $O(\log \frac{1}{\epsilon})$ copies and then take their products. What we would get is,

$$1 - \prod_{all\ copies} \prod_{i} (1 - p_i) \quad \Rightarrow error \leq (\frac{2}{3})^{c \log \frac{1}{\epsilon}}$$

By choosing an appropriate c one would get the error to be $\leq \epsilon$. The degree has now become $O(\log \frac{1}{\epsilon} \cdot \log n)$.

Now suppose, a circuit of depth d and size s is given to us. Let us chose $\epsilon = \frac{1}{4 \cdot s^2}$. Taking the size of the circuit as bound on the number of inputs to the output gate, the polynomial corresponding to the output gate would have a degree $\leq O((\log s \cdot \log s)^d) = O((\log s)^{2d})$. The total error for the circuit would be $\leq size\ of\ circuit * error\ at\ each\ gate \leq \frac{1}{4s}$.

Let us now consider the circuit shown in Fig 10.3. Note that s is the size of the complete circuit. Hence, the size of each of the AC^0 components can be at most s and $k \leq s$. $MAJORITY$ over the reals is defined in the following way.

$$MAJ(p_1, p_2, \ldots, p_k) \begin{cases} = 1, & if \ \left(\sum_{i=1}^{k} p_i - \frac{k}{2}\right) > 0 \\ = 0, & otherwise \end{cases}$$

The fraction of errors at the MAJ-gate would be $\leq s \cdot \frac{1}{4s} = \frac{1}{4}$. This implies that there exists a *voting polynomial* of degree $O((\log s)^{2d})$ which approximates $PARITY$ with $\frac{2^n}{4}$ errors. But, we already saw that any approximate polynomial for $PARITY$ has $\Omega\sqrt{n}$ degree. This implies,

$$\Rightarrow (\log s)^{2d} > n^{\frac{1}{2}} \quad \Rightarrow \log s > n^{\frac{1}{4d}} \quad \Rightarrow s > 2^{n^{\frac{1}{4d}}}$$

That is, even with MAJ on top, we cannot compute $PARITY$ in constant-depth polynomial-sized circuits.

Computational Complexity Theory	**Spring 1997**
Lecture 19-20 : April 1 & 3 1997	
Lecturer: V. Vinay	*Scribe: P. R. Subramanya*

Razborov ([15]) was the first to show an **exponential lower bound** for **monotone circuits**. This set of lectures shall focus on *Razborov*'s result that $CLIQUE$ requires an exponential-sized monotone circuit.

Recall that a *monotone circuit* is one that does not contain any negations.

11.1 Sunflower Lemma

Razborov used the **Sunflower Lemma** in his exponential lower bound proof for $CLIQUE$. This lemma discovered by *Erdós & Rado* seems to find *order in chaos*. Fig 11.1 illustrates a **sunflower**.

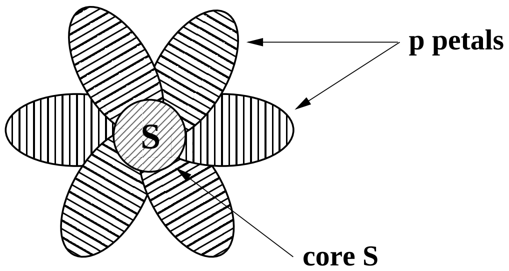

Figure 11.1: A sunflower with a common core S and p petals.

Lemma 5 *Let $\mathcal{F} = \{S_1, S_2, \ldots, S_k\}$ be a set system over some universe such that*

(1) $|S_i| \leq l$
(2) $k > (p-1)^l \cdot l!$

Then, $\exists$ a **sunflower** *with p petals. That is, $\exists F \subset \mathcal{F}$, where $F = \{S_{i_1}, S_{i_2}, \ldots, S_{i_p}\}$, such that for any two sets $A, B \in F$, $A \cap B = S$.*

Proof: We shall use induction on l to prove the lemma.
Base Case: $l = 1$. $\Rightarrow k > (p-1)$ i.e. $k \geq p$. The core, S, is the null set ϕ and we have p disjoint sets as the petals.
Inductive Step: Assume the result is true for $(l-1)$. Order the original set system. Pick sets

$\hat{S}_1, \hat{S}_2, \ldots, \hat{S_m}$ such that each is disjoint. At some point one will not be able to pick any more sets. Two cases arise. The first case is when $m \geq p$. There is nothing to be done here. The core set is $S = \phi$.

The other case is when $m < p$. Consider the set $Z = \bigcup_{i=1}^{m} \hat{S}_i$. Z by construction intersects with every set in $\mathcal{F}$ and, $|Z| \leq m \cdot l \leq (p-1) \cdot l$. Each element of Z is contained on the average in $\frac{k}{|Z|} > \frac{(p-1)^l \cdot l!}{(p-1) \cdot l}$ sets of $\mathcal{F}$. Hence, there must be some element $x \in Z$ that intersects $\geq (p-1)^{l-1} \cdot (l-1)!$ sets in $\mathcal{F}$. Take the collection of sets that contain x and suppress x in each of these sets. The cardinality of each set is now bounded by $(l-1)$ and there are $k \geq (p-1)^{l-1} \cdot (l-1)!$ sets. That is, the induction hypothesis is satisfied. Hence, there exist p petals but of size at most $(l-1)$. Throw x back into these p petals. ■

The *Sunflower Lemma* indicates that in any large collection of sets from some universe, which seems like chaos, there is order in it. It is not known whether the second condition in the *Sunflower Lemma* can be improved. That is, when exactly do we start seeing *sunflowers*.

11.2 *CLIQUE*

The *CLIQUE* problem can be stated as, given a graph $G = (V, E)$, is there a clique of size k. Note that, *CLIQUE* is a monotone property. That is, adding edges to a graph increases the chances of finding a clique, but never reduces it. A naive way to look for a clique of size k in G would be to look at all $\binom{n}{k}$ sets of vertices and check if any one of them is a clique. Once $k \geq n^{\frac{1}{e}}$, the number of sets to be checked is no longer a polynomial.

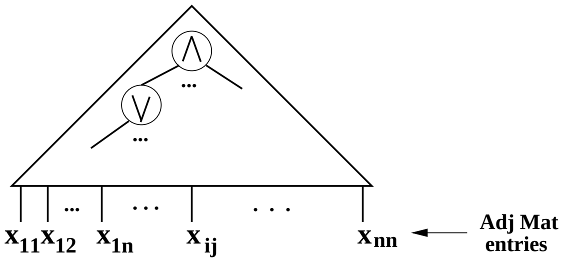

Figure 11.2: A circuit that computes *CLIQUE*

Let us now discuss *Razborov*'s proof for an exponential sized lower bound for monotone circuits computing *CLIQUE*. Fig 11.2 shows a general circuit that computes *CLIQUE*. With every gate in the circuit we associate a bunch of **clique indicators**, written as $\{\lceil X_i \rceil\}_{i=1}^{m}$. A *clique* *indicator* is defined as,

$$\lceil X \rceil \begin{cases} = 1, & if\ on\ the\ set\ X\ there\ is\ a\ clique. \\ = 0, & otherwise. \end{cases}$$

If any one *clique indicator* in a collection of them is ON, then the whole collection is considered to be ON. That is, the collection behaves like an $\bigvee$.

The parameters for $CLIQUE$ are,

k	size of the clique being searched.
l	bound on the size of each clique indicator.
m	bound on the size of each *collection* of *clique indicators*.

We need to address the issue of merging *clique indicators* at the circuit gates based on the gate type. Consider the $\bigvee$-gate. Its operands (2 in number w.l.o.g) are collections of *clique indicators*. This corresponds naturally to the $\bigcup$ of the 2 collections as indicated in Fig 11.3.

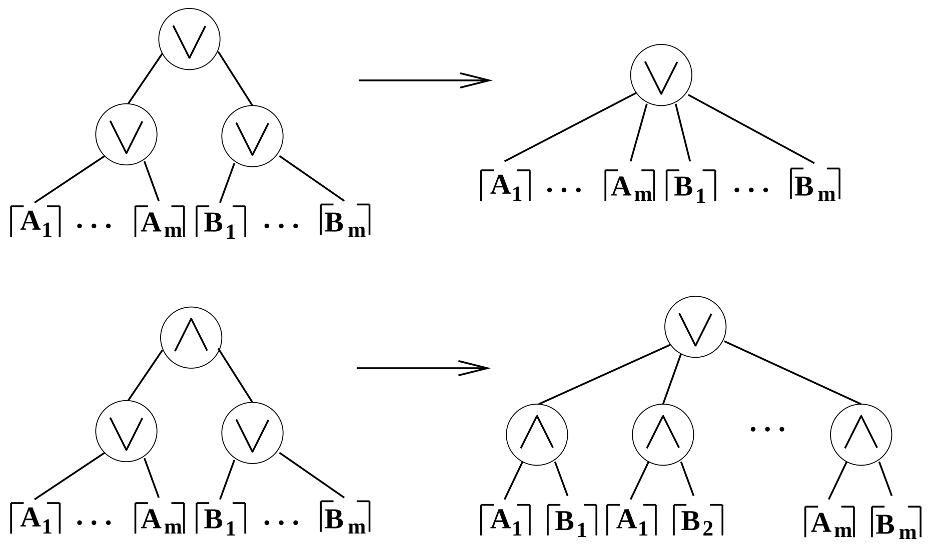

Figure 11.3: Merging of *clique indicators*

ORing of 2 collections could result in the collection size, m', growing upto $2m$. But, we want $m' \le m$. A technique called **plucking** is used and involves the following steps.

Repeat until $m' \le m$.
   Choose a p such that $m' > (p-1)^l \cdot l!$.
   Find a *sunflower*.
   Pluck the petals off.
   Retain the core. Number of sets is reduced by $\ge (p-1)$.

ANDing 2 collections could result in a collection of size as high as m^2. Fig 11.3 shows the $\bigwedge$ operation on 2 collections. To get the collection size back on the rails, we need

- **Union:** The $\bigwedge$ of 2 *clique indicators* is merely their union, i.e. $\lceil A_i \rceil \wedge \lceil B_j \rceil = \lceil A_i \cup B_j \rceil$. The size of the union could exceed l. Solution is to **erase** all such unions.
- **Plucking:** At this stage all the *clique indicators* are $\leq l$. One now performs *plucking* to ensure that the size of the collection is $\leq m$.

Razborov found the $CLIQUE$ problem too general for his liking. He focused his attention on two classes of $CLIQUE$ instances, called **positive** and **negative test cases**. The *positive test cases* are those instances that have a complete graph on k vertices. There are $\binom{n}{k}$ of these, and they are the *minimal class* of positive instances of $CLIQUE$, i.e. removal of a single edge makes the instance a negative one.

The *negative test cases* are $(k-1)$-partite graphs. These are $CLIQUE$ instances that result from colouring n vertices with $(k-1)$ colours and connecting any 2 vertices with different colours. That is, $(k-1)^n$ cases in this class.

We are interested in the number of errors that the circuit could commit. At the output gate, two cases arise. An *empty list* could be associated with the output gate. On all the positive cases, the output gate gives a 0 and makes a mistake. On the negative cases, the behaviour is perfect. In the second case, the output gate has at least one *clique indicator* in its collection. The question now is, how many errors (i.e. firing when it should not) can be accrued on the negative test cases.

Consider $\lceil X \rceil \leq l$. The vertices in the *clique indicator* could come from l of the $(k-1)$ partitions, and the gate fires. The number of such cases is $\binom{k-1}{l} \cdot l! \cdot (k-1)^{n-l}$. That is, the fraction of the negative test cases for which $\lceil X \rceil$ makes a mistake is

$$\frac{\binom{k-1}{l} \cdot l \ \cdot (k-1)^{n-l}}{(k-1)^n} = \frac{(k-1)!}{(k-l-1)! \cdot (k-1)^l} = \frac{(k-1)\cdot(k-2)\cdot \ldots \cdot (k-l)}{(k-1)^l}$$

$$= \frac{k-1}{k-1} \cdot \frac{k-2}{k-1} \cdot \ldots \cdot \frac{k-l}{k-1} = 1 \cdot (1 - \frac{1}{k-1}) \cdot (1 - \frac{2}{k-1}) \cdot \ldots \cdot (1 - \frac{l-1}{k-1})$$

Taking logs, we have

$$e^{\sum_{j=1}^{l-1} \log(1 - \frac{j}{k-1})} = e^{-\sum_{j=1}^{l-1} \frac{j}{k-1}} = e^{-\frac{l \cdot (l-1)}{2 \cdot (k-1)}}$$

Taking $l = \lfloor \sqrt{n} \rfloor$, the fraction of the negative test cases on which a single *clique indicator* makes an error is $\geq \frac{1}{2}$.

We have bounded the number of errors that could be made by the circuit. We are now interested in finding the number of gates in the circuit. We shall consider the positive and negative test cases separately and find out which gates contribute to its errors.

Consider the *positive test cases* and let us bound the number of errors that could be committed for these by the $\vee$ and $\wedge$ gates.

- **OR:** This cannot make mistakes on the *positive test cases.* Mistakes are made by an $\vee$-gate if it outputs a 0 when one of its children gave a 1. Intuitively, *plucking* retains the **core** of the *clique indicators* that it discarded. Hence, it has reduced the size of the *clique indicators* that were discarded, i.e. if the discarded *clique indicators* were 1, the core will also be 1.
- **AND:** *Unioning* by itself is not a problem, and neither is *plucking*, as we saw. *Erasing* clique indicators is problematic since, the act of removal of a union of clique indicators, might cause some clique not to be considered at all. Such positive test cases occur when at least $(l+1)$ vertices of a clique of size k are fixed. And, since at most m^2 *clique indicators* could be erased at an $\wedge$-gate, we get a total of $m^2 \cdot \binom{n-l-1}{k-l-1}$.

Knowing the number of errors the circuit makes on the *positive test cases*, and the fact that these are all contributed by $\wedge$-gates, gives us a bound on the number of $\wedge$-gates.

$$\Rightarrow Number\ of\ \wedge\ gates \geq \frac{\binom{n}{k}}{m^2 \cdot \binom{n-l-1}{k-l-1}}$$

Now, consider the *negative test cases*, and let us look at its sources of errors. An $\vee$-gate gives a 0 only when all of its inputs are 0s. Since *plucking* reduces the size of *clique indicators*, some of these and hence, their collections might be fired when they should not. This situation can be abstracted as follows. $Z_1, Z_2, \ldots, Z_p$ are the petals such that $\forall_{i \neq j} Z_i \cap Z_j = Z$ and each $Z_i = 0$ but $Z = 1$. In the context of *colouring*, this can be stated as, Z gets a **proper colouring** (PC) (i.e. each vertex of Z is a different colour) while none of the Z_i are *properly coloured.* Using the fact that $Prob\{A \wedge B\} \leq Prob\{B|A\}$ we get,

$$Prob\{Z\ is\ PC\ \ \wedge Z_1 Z_2 \ldots Z_p\ are\ not\ PC\} \leq Prob\{Z_1 Z_2 \ldots Z_p\ are\ not\ PC | Z\ is\ PC\}$$

From the fact that if Z is properly coloured, the probability that Z_i is not is less than if Z_i were to be not properly coloured independently of Z.

$$\leq \prod_i Prob\{Z_i\ is\ not\ PC | Z\ is\ PC\} \leq \prod_i Prob\{Z_i\ is\ not\ PC\}$$

Now, the probability that a single Z_i is not *properly coloured* is given by $\frac{\binom{l}{2}}{k-1}$ and hence,

$$\Rightarrow Prob\{Z\ is\ PC \wedge Z_1 Z_2 \dots Z_n\ are\ not\ PC\} \leq \left(\frac{\binom{l}{2}}{k-1}\right)^p$$

There can be at most m pluckings at an $\vee$-gate, while there are $O(m^2)$ pluckings for a $\wedge$-gate. Hence, the number of errors that could be caused by a **single** gate for the *negative test cases* is bounded by $m^2 \cdot (\frac{\binom{l}{2}}{k-1})^p \cdot (k-1)^n$. The bound on the total number of negative test case errors is $\frac{1}{2} \cdot (k-1)^n$. The number of gates is bounded by,

$$\#\ gates\ \geq \frac{1}{2} \cdot \frac{(k-1)^n}{m^2 \cdot (\frac{1}{2})^p \cdot (k-1)^n} = \frac{2^{p-1}}{m^2}$$

Combining the two bounds for the number of gates that we have found, we have

$$\#\ gates\ \geq min \left\{ \frac{\binom{n}{k}}{m^2 \cdot \binom{n-l-1}{k-l-1}}, \frac{2^{p-1}}{m^2} \right\}$$

Choosing the following set of values for our parameters,

$$l = \lfloor \sqrt{k} \rfloor \quad ; \quad k = \left(\frac{n}{8\log^2 n}\right)^{\frac{1}{3}} \quad ; \quad p = \lceil \sqrt{k} \cdot \log n \rceil + 1$$

we get a **lower bound** of $2^{\Omega(\sqrt{k})}$. The basic idea employed in the analysis above is summarized in the table below of *test case type* versus *gate type*,

	positive test cases	**negative test cases**
OR gate	no errors	errors due to plucking
AND gate	errors due to erasures	errors due to plucking

Bibliography

[1] M. Ajtai, Σ_1^1-*formulae for finite structures,* *Annals of Pure & Applied Logic*, Vol. 24, pp 1 - 48, 1983.

[2] J. Aspens,R. Beigel,M. Furst and S. Rudich, *The Expressive Power of Voting Polynomials,* in: *Proc. 23rd Annual ACM Symp. on Theory of Computing*, pp 402 - 409, 1991. also in: Combinatorica, pp 1 - ??, 1994.

[3] David. A . Barrington, *Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1, JCSS*, Vol. 38, pp 150 - 164, 1989.

[4] P. Beame, *A Switching Lemma Primer,* manuscript, 1994.

[5] P.Beame, S.A.Cook & H.J.Hoover, *Log Depth Circuits for Division and Related Problems, SIAM Journal on Computing*, Vol. 15, No. 4, pp 994 - 1003, Nov 1986.

[6] M. Ben-Or and R. Cleve, *Computing algebraic formulas using a constant number of registers, SIAM Journal of Computing*, Vol. 21, No. 1, pp 54 - 58, Feb 1992.

[7] R. B. Boppana and M. Sipser, *The Complexity of Finite Functions,* in: J. van Leeuwen, ed. *The Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, Elsevier, pp 757 - 804, 1990.

[8] R. P . Brent, *The Parallel Evaluation of General Arithmetic Expressions, JACM*, Vol. 89, No. 2, pp 201 - 206, Feb 1974.

[9] L. Fortnow and S. Laplante, *Circuit Lower Bounds a la Kolmogorov,* manuscript, 1995.

[10] M. Furst,J. Saxe and M. Sipser, *Parity, circuits and the polynomial time hierarchy, Mathematical Systems Theory*, Vol. 17, pp 13 - 27, 1984.

[11] J. Hastad, *Almost optimal lower bounds for small depth circuits,* in: S. Micali, ed., *Advances in Computer Research, Vol. 5: Randomness and Computation* (JAI Press, Greenwich, CT); see also *Computational Limitations for Small Depth Circuits* (MIT Press, Cambridge, MA, 1986).

[12] N. Immerman *Nondeterministic Space is closed under complementation, SIAM Journal of Computing*, Vol. 17, No. 5, pp 935 - 938, Oct 1988.

[13] M . Nair, *On Chebyschev-Type Inequalities for Primes, The American Mathematical Monthly*, Vol. 89, No. 2, pp 126 - 129, Feb 1982.

[14] N. Nisan $\mathcal{RL} \subseteq \mathcal{SC}$, *Proc. 24th ACM Symp. on the Theory of Computing (STOC)*, pp 9 - 623, 1992.

[15] A. A. Razborov, *Lower bounds on the size of bounded depth networks over a complete basis with logical addition, Mathematical Notes*, Vol. 41, No. 4,pp 333 - 338, 1987.

[16] A. A. Razborov, *On the method of approximations,* in *Proc. 21st Annual Symp. on Theory of Computing*, pp 167 - 176, 1989.

[17] A. A. Razborov, *Bounded Arithmetic and Lower Bounds in Boolean Complexity,* to appear in *Feasible Mathematics II*, 1993.

[18] W. J. Savitch *Relationship between nondeterministic and deterministic tape classes, Journal of Computer and System Sciences*, Vol. 4, pp 7 - 192, 1970.

[19] R. Smolensky, *Algebraic methods in the theory of lower bounds for Boolean Circuit Complexity, Proc. 19th Annual ACM Symp. on Theory of Computing*, pp 77 - 82, 1987.

[20] R. Szelepcsényi *The method of forcing for nondeterministic automata, Bulletin of the EATCS*, Vol. 33, pp - 100, 1987.

[21] Jun Tarui, *Randomized polynomials, threshold circuits and the polynomial hierarchy,* manuscript, 1990.

[22] A. C. Yao, *Separating the polynomial-time hierarchy by oracles, Proc. 26th Annual IEEE Symp. on Foundations of Computer Science*, pp 1 - 10, 1985.