Introduction to Complexity Theory*
Lecture 20: Circuit Depth and Space Complexity

Notes taken by Vered Rosen and Alon Rosen

June 3, 1999

Summary: In this lecture we study some of the relations between Boolean circuits and Turing machines. We define the complexity classes NC and AC, compare their computational power, and point out the possible connection between $uniform\text{-}NC$ and "efficient" parallel computation. We conclude the discussion by establishing a strong connection between space complexity and depth of circuits with bounded fan-in.

1 Boolean Circuits

Loosely speaking, a *Boolean Circuit* is a directed acyclic graph with three types of labeled vertices: *inputs*, *outputs*, and *gates*. The inputs are sources in the graph (i.e. vertices with in-degree 0), and are labeled with either Boolean variables or constant Boolean values. The outputs are sinks in the graph (i.e. vertices with out-degree 0). The gates are vertices with in-degree $k > 0$, which are labeled with Boolean functions on k inputs. We refer to the in-degree of a vertex as its *fan-in* and its out-degree as its *fan-out*. A general definition of Boolean circuits would allow the labeling of gates with arbitrary Boolean functions. We restrict our attention to circuits whose gates are labeled with the boolean functions AND, OR, and NOT (denoted $\bigwedge, \bigvee, \neg$ respectively).

1.1 The Definition

Definition 1 (Boolean Circuit): *A* **Boolean Circuit** *is a directed acyclic graph with labeled vertices:*

- *The* input *vertices, labeled with a variable x_i or with a constant (0 or 1), and have fan-in 0.*
- *The* gate *vertices, have fan-in $k > 0$ and are labeled with one of the boolean functions $\bigwedge, \bigvee, \neg$ on k inputs (in the case that the label is $\neg$, the fan-in k is restricted to be 1).*
- *The* output *vertices, labeled 'output', and have fan-out 0.*

Given an assignment $\sigma \in \{0,1\}^m$ to the variables $x_1, ..., x_m$, $C(\sigma)$ will denote the value of the circuit's output. The value is defined in the natural manner, by setting the value of each vertex according to the boolean operation it is labeled with. For example, if a vertex is labeled $\bigwedge$ and the vertices with a directed edge to it have values a and b, then the vertex has value $a \bigwedge b$.

We denote by $size(C)$ the number of gates in a circuit C, and by $depth(C)$ the maximum distance from an input to an output (i.e. the longest directed path in the graph).

---
* Lecture Notes for a course given by Oded Goldreich at the Weizmann Institute of Science, Israel.

A circuit is said to have *bounded* fan-in if there exists an a-priori upper bound on the fan-in of its AND and OR gates (NOT gates, must have fan-in 1 anyway). If there is no a-priori bound on the fan-in (other than the size of the circuit), the circuit is said to have *unbounded* fan-in.

1.2 Some Observations

- We have already seen how to construct a circuit which simulates the run of a Turing machine M on some input $x \in \{0,1\}^n$ (see the proof of Cook's Theorem , Lecture 2). Using this construction the resulting circuit will be of size quadratic in $T_M(n)$ (the running time of M on input length n), and of depth bounded by $T_M(n)$.

- Circuits may be organized into disjoint layers of gates, where each layer consists of gates having equal distance from the input vertices. Once presented this way, a circuit may be thought of as capturing a certain notion of parallel computation. We could associate each path starting from an input vertex with a computation performed by a single processor. Note that all such computations can be performed concurrently. Viewed in this way, the circuit's depth corresponds to parallel time, whereas the size corresponds to the total amount of parallel work.

- Any bounded fan-in circuit can be transformed into a circuit whose gates have fan-in 2 while paying only a constant factor in its depth and size. A gate with constant fan-in c, can be converted into a binary tree of gates of the same type which computes the same function as the original gate. Since c is a constant so will be the tree's size and depth. We can therefore assume without loss of generality that all gates in bounded fan-in circuits have fan-in 2. Note that the same transformation in the case of unbounded fan-in will increase the depth by a factor logarithmic in the size of the circuit.

- Any circuit can be modified in such a way that all negations (i.e. $\neg$ gates) in it appear only in the input layer. Using De-Morgan's laws (that is, $\neg \bigvee x_i = \bigwedge \neg x_i$), each $\bigvee$ gate followed by a negation can be transformed into an $\bigwedge$ gate whose inputs are negations of the original inputs (the same argument applies symmetrically for $\bigwedge$ gates). This way, we can propagate all negations in the circuit towards the input layer without changing the value of the circuit's output, and without changing its depth. Thus, without loss of generality, all "internal" gates in a Boolean circuit are labeled with $\bigwedge$ or $\bigvee$. When measuring the depth of the circuit, negations will be counted as a single layer.

- In an unbounded fan-in circuit, two consecutive $\bigwedge$ (resp. $\bigvee$) gates having identical labels can be merged into a single gate with the same label without changing the value of the circuit's output. We can therefore assume that unbounded fan-in circuits are of the special form where all $\bigwedge$ and $\bigvee$ gates are organized into alternating layers with edges only between adjacent layers.

  Note, however, that the above argument does not necessarily work for bounded fan-in circuits since the merging operation might cause a blow-up in the fan-in of the resulting gates which will exceed the specified bound.

1.3 Families of Circuits

Eventhough a circuit may have arbitrarily many output vertices we will focus on circuits which have only one output vertex (unless otherwise specified). Such circuits can be used in a natural way to define a language (subset of $\{0,1\}^*$). Since we are interested in deciding instances of arbitrary size, we consider *families* of Boolean circuits with one different circuit for each input size.

Definition 2 (Family of Circuits): *A language $L \subseteq \{0,1\}^*$ is said to be decided by a* family of circuits, *$\{C_n\}$, when C_n takes n variables as inputs, if and only if for every n, $C_n(x) = \chi_L(x)$ for all $x \in \{0,1\}^n$.*

Given a family of circuits, we might be interested in measuring the growth-rate of the size and depth of its circuits (as a function of n). This may be useful when trying to connect circuit complexity with some other abstract model of computation.

Definition 3 (Depth and Size of Family) *Let D and S be sets of integer functions ($N \to N$), we say that a family of circuits $\{C_n\}$, has depth D and size S if for all n, $depth(C_n) \leq d(n)$ and $size(C_n) \leq s(n)$ for some $d(\cdot) \in D$ and $s(\cdot) \in S$.*

If we wish to correlate the size and depth of a family $\{C_n\}$ that decides a language L with its Turing machine complexity, it is necessary to introduce some notion of *uniformity*. Otherwise, we could construct a family of circuits which decides a non-recursive language (see Lecture 8). The notion of uniformity which we will make use of is *logspace uniformity*. Informally, we require that a description of a circuit can be obtained by invoking a Turing machine using space which is logarithmic in the length of its output (i.e. the circuit's description). A description of a circuit (denoted $desc(C_n)$), is a list of its gates, where for each gate we specify its type and its list of predecessors. Note that the length of $desc(C_n)$ is at most quadratic in $size(C_n)$.

Definition 4 (logspace uniformity): *A family of circuits, $\{C_n\}$, is called* logspace uniform *if there exists a deterministic Turing machine, M, such that for every n, $M(1^n) = desc(C_n)$ while using space which is logarithmic in the length of $desc(C_n)$.*

The reason we require M to work in space which is logarithmic in its *output* length (rather than its input length) lies in the fact that $size(C_n)$ (and therefore the length of $desc(C_n)$) might be super-polynomial in n. The problem is that the description of a circuit with super-polynomial size, *cannot* be produced by Turing machines working with space which is logarithmic in n (the input size), and therefore such a circuit (one with super-polynomial size) would have been overlooked by a definition using the input length as parameter. Note that if we restrict ourselves to circuits with polynomial size, then the above remark does not apply, and it is sufficient to require that M is a deterministic logspace Turing machine.

Based on the above definitions, the class $\mathcal{P}/poly$ is the class of all languages for which there exists a family of circuits having polynomial depth and polynomial size (see Lecture 8). We have already seen how to use the transformation from Turing machines into circuits in order to prove that $uniform$-$\mathcal{P}/poly$ contains (and in fact equals) $\mathcal{P}$. We note that the above transformation can be performed in logarithmic space as well. This means that $logspace$-$uniform$-$\mathcal{P}/poly$ also equals $\mathcal{P}$. In the sequel, we choose logspace uniformity to be our preferred notion of uniformity.

2 Small-depth circuits

In this section we consider polynomial size circuits whose depth is considerably smaller than n, the number of inputs. By depth considerably smaller than n we refer to *poly-logarithmic* depth, that is, bounded by $O(\log^k n)$ for some $k \geq 0$. We are interested in separating the cases of unbounded and bounded fan-in, specifically, we will investigate the relation between two main complexity classes. As we will see, these classes will eventually turn out to be subsets of $\mathcal{P}$ which capture the notion of what is "efficiently" computable by parallel algorithms.

2.1 The Classes NC and AC

The complexity class NC is defined as the class of languages that can be decided by families of bounded fan-in circuits with polynomial size, and poly-logarithmic depth (in the number of inputs to the circuits). The actual definition of NC introduces an hierarchy of classes decided by circuit families with increasing depth.

Definition 5 (NC): *For $k \geq 0$, define NC^k to be the class of languages that can be decided by families of circuits with* bounded *fan-in, polynomial size, and depth $O(\log^k n)$. Define NC to be the union of all NC^k's (i.e. $NC \stackrel{\text{def}}{=} \bigcup_k NC^k$).*

A natural question would be to ask what happens to the computational power of the above circuits if we remove the bound on the fan-in. For instance, it is easy to see that the decision problem "is the input string $\sigma \in \{0,1\}^n$ made up of all 1's?" (the AND function) can be solved by a depth-1 unbounded fan-in circuit, whereas in a bounded fan-in circuit this problem would require depth at least $\Omega(\log n)$, just so all the input bits could *effect* the output gate. The classes based on the unbounded fan-in model (namely AC) are defined analogously to the classes in the NC hierarchy.

Definition 6 (AC): *For $k \geq 0$, define AC^k to be the class of languages that can be decided by families of circuits with* unbounded *fan-in, polynomial size, and depth $O(\log^k n)$. Define AC to be the union of all AC^k's (i.e. $AC \stackrel{\text{def}}{=} \bigcup_k AC^k$).*

As we will see in the sequel, AC equals NC. Note however, that such a result does not necessarily rule out a separation between the computational power of circuits with bounded and respectively unbounded fan-in. We have already seen that the class NC^0 of languages decided by constant depth circuits with bounded fan-in, is strictly contained in AC^0, its unbounded fan-in version. We are therefore motivated to look deeper into the NC^k and AC^k hierarchies and try to gain a better understanding of the relation between their corresponding levels.

Theorem 7 *For all $k \geq 0$,*

$$NC^k \subseteq AC^k \subseteq NC^{k+1}$$

Proof: The first inclusion is trivial, we turn directly to prove the second inclusion. Since any gate in an unbounded fan-in circuit of size $\text{poly}(n)$ can have fan-in at most $\text{poly}(n)$, each such gate can be converted into a tree of gates of the same type with fan-in 2, such that the output gate of the tree computes the same function as the original gate (since this transformation can be performed in logarithmic space, the claim will hold both for the uniform and the nonuniform settings). The resulting depth of the tree will be $\log(\text{poly}(n)) = O(\log n)$. By applying this transformation to each gate in an unbounded fan-in circuit of depth $O(\log^k n)$ we obtain a bounded fan-in circuit deciding the same language as the original one. The above transformation will cost us only a logarithmic factor in the depth and a polynomial factor in the size (i.e. the depth of the resulting circuit will be $O(\log^{k+1} n)$, and the size will remain polynomial in n). Thus, any language in AC^k is also in NC^{k+1}. ■

Corollary 8 $AC = NC$

In light of Theorem 7, it might be interesting to ask how far does the NC (resp. AC) hierarchy extend. Is it infinite, or does it collapse to some level? Even if a collapse seems unlikely, at present no argument is known to rule out this option. One thing which **is** known at present, is that AC^0 is strictly contained in NC^1.

Theorem 9 $AC^0 \subset NC^1$ *(strictly contained).*

Note that $uniform\text{-}NC$ is contained in $\mathcal{P}$. Theorem 9 implies that $uniform\text{-}AC^0$ is *strictly* contained in $\mathcal{P}$. An interesting open question is to establish what is the exact relationship between $uniform\text{-}NC$ and $\mathcal{P}$, it is not currently known whether both classes are equal or not (analogously to the $\mathcal{P}$ vs. $\mathcal{NP}$ problem). As a matter of fact, it is not even known how to separate $uniform\text{-}NC^1$ from $\mathcal{NP}$.

2.2 Sketch of the proof of $AC^0 \subset NC^1$

We prove the Theorem by showing that the decision problem "does the input string have an even number of 1's?" can be solved in NC^1 but **cannot** be solved in AC^0. In this context it will be more convenient to view circuits as computing functions rather than deciding languages.

Definition 10 (Parity): *Let $x \in \{0,1\}^n$, the function Parity is defined as:*

$$Parity(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \sum_{i=1}^{n} x_i \ (mod\ 2)$$

Claim 2.1 *$Parity \in NC_1$.*

Proof: We construct a circuit computing $Parity$, using a binary tree of logarithmic depth where each gate is a *xor* gate. Each xor gate can then be replaced by the combination of 3 legal gates: $a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$. This transformation increases the depth of the circuit by a factor of 2, and the size of the circuit by a factor of 3. Consequently, $Parity$ is computed by circuits of logarithmic depth and polynomial size, and is thus in NC^1. ■

Claim 2.2 *$Parity \notin AC_0$.*

In order to prove claim 2.2 we show that every constant depth circuit computing $Parity$ must have sub-exponential size (and therefore $Parity$ cannot belong to AC^0), more precisely:

Theorem 11 *For every constant d, a circuit computing Parity on n inputs with depth d, must have size $exp(\Omega(n^{\frac{1}{d-1}}))$.*

Proof: **(sketch)**: The Theorem is proven by induction on d and proceeds as follows:

1. Prove that parity circuits of depth 2 must be of large size.

2. Prove that depth d parity circuits of small size can be converted to depth $d-1$ parity circuits of small size (thus contradicting the induction hypothesis).

The first step is relatively easy, whereas the second step (which is the heart of the Theorem) is by far more complicated. We will therefore give only the outline of it.

Base case ($d = 2$): Without loss of generality, we assume that the circuit given to us is an OR of ANDs (in case it is an AND of ORs the following arguments apply symmetrically). Then, any AND gate evaluating to "1" determines the value of the circuit. Consider now an AND gate (any gate in the intermediate layer). Note that all the variables $x_1, \ldots, x_n$ must be connected to that gate. Otherwise, assume there exists an i such that there is no edge going from x_i (and from $\neg x_i$) into the gate: Then, take an assignment to the variables going into the gate, causing it to evaluate to "1" (we assume the gate is not degenerate). Under this assignment, the circuit will output "1", regardless of the value of x_i, which is impossible.
Due to that, we can associate each AND gate with a certain assignment to the n variables (determined by the literals connected to that gate). We can say that this gate will evaluate to "1" iff the variables will have this assignment.
This argument shows, that there must be at least 2^{n-1} AND gates. Otherwise, there exists an assignment σ to $x_1, \ldots, x_n$, such that $Parity(\sigma) = 1$, of which there is no AND gate identified with σ. This means that the circuit evaluated on the assignment σ will output "0", in contradiction.

The induction step: The basic idea of this step lies in a lemma proven by Hastad. The lemma states that given a depth 2 circuit, say an AND of ORs, then if one gives random values to a randomly selected subset of the variables, then it is possible to write the resulting induced circuit as an OR of relatively few ANDs with very high probability. We now outline how use this lemma in order to carry on the induction step:

Given a circuit of depth d computing parity, we assign random values to some of its inputs (only a large randomly chosen subset of the inputs are preset, the rest stay as variables). Consequently, we obtain a simplified circuit that works on fewer variables. This circuit will still compute parity (or the negation of parity) of the remaining variables.

By virtue of the lemma, it is possible to interchange two adjacent levels (the ones closest to the input layer) of ANDs and ORs. Then, by merging the two now adjacent levels with the same connective, we decrease the depth of the circuit to $d-1$. This is done without increasing significantly the size of the circuit.

Let us now make formal what we mean by randomly fixing some variables.

Definition 12 (random restriction): *A* random restriction *with a parameter ϵ to the variables $x_1, \ldots, x_n$, treats every variable x_i (independently) in the following way:*

$$x_i = \begin{cases} w.p.\ \frac{1-\epsilon}{2} & set\ x_i \leftarrow 0 \\ w.p.\ \frac{1-\epsilon}{2} & set\ x_i \leftarrow 1 \\ w.p.\ \epsilon & leave\ it\ \text{“alive”} \end{cases}$$

Observe that the expected number of variables remaining is $m = n\epsilon$. Obviously, the smaller ϵ is the more we can simplify our circuit, but on the other hand we have fewer remaining variables.

In order to contradict the induction hypothesis, we would like the size of the transformed circuit (after doing a random restriction and decreasing its depth by 1) to be smaller than $exp(o(m^{\frac{1}{d-2}}))$. It will suffice to require that $n^{\frac{1}{d-1}} < m^{\frac{1}{d-2}}$, or alternatively, $m > n \cdot n^{\frac{-1}{d-1}}$. Thus, a wise choice of the parameter ϵ, would be $\epsilon = n^{\frac{-1}{d-1}}$. ■

2.3 NC and Parallel Computation

We now turn to briefly examine the connection between the complexity class *uniform-NC* and parallel computation. In particular, we consider the connection to the parallel random-access machine (PRAM), which can be viewed as a parallel version of the RAM. A RAM is a computing device (processor), consisting of a program, a finite set of registers, and an infinite memory (whose cells are of the same size as the registers). The program is a finite sequence of instructions which are executed sequentially, where at the execution of each instruction the processor reads and writes values to some of its registers or memory cells, as required by the instruction.

The PRAM consists of several independent sequential processors (i.e. RAMs), each having its own set of private registers. In addition there is an infinite shared memory accessible by all processors. In one unit time (of a clock common to all processors), each processor executes a single instruction, during which it can read and write to its private set of registers, or to the shared memory cells. PRAMs can be classified according to restrictions on global memory access. An Exclusive-Read Exclusive-Write (EREW) PRAM forbids simultaneous access to the same memory cell by different processors. A Concurrent-Read Exclusive-Write (CREW) PRAM allows simultaneous reads but no simultaneous writes, and a Concurrent-Read Concurrent-Write (CRCW) PRAM allows simultaneous reads and writes (in this case one has to define how concurrent writes are treated). Despite this variety of different PRAM models, it turns out that they do not differ very widely in their computational power.

In designing algorithms for parallel machines we obviously want to minimize the time required to perform the concurrent computation. In particular, we would like our parallel algorithms to be *dramatically* faster than our sequential ones. Improvements in the running time would be considered dramatic if we could achieve an exponential drop in the time required to solve a problem, say from polynomial to logarithmic (or at least poly-logarithmic). Denote by PRAM$(t(\cdot), p(\cdot))$ the class of languages decidable by a PRAM machine working in parallel time $t(\cdot)$ and using $p(\cdot)$ processors, then we have the following:

Theorem 13 *uniform-NC* = PRAM(polylog,poly)

Hence, the complexity class *uniform-NC* captures the notion of what is "efficiently" computable by PRAM machines (just as the class $\mathcal{P}$ captures the notion of what is "efficiently" computable by RAM machines, which are equivalent to Turing machines). Note however, that the PRAM cannot be considered a physically realizable model, since, as the number of processors and the size of the global memory scales up, it quickly becomes impossible to provide a constant-length data path from any processor to any memory cell. Nevertheless, the PRAM has proved to be an extremely useful vehicle for studying the logical structure of parallel computation. Algorithms developed for other, more realistic models, are often based on algorithms originally designed for the PRAM. Moreover, a transformation from the PRAM model into some other more realistic model will cost us only a logarithmic factor in the parallel complexity.

Finally, we would like to note that the analogy of NC to parallel computation has some aspects missing. First of all, it ignores the issue of the communication between processors. As a matter of fact, it seems that in practice this is the overshadowing problem to make large scale parallel computation efficient (note that the PRAM model implicitly overcomes the communication issue since two processors can communicate in $O(1)$ just by writing a message on the memory). Another aspect missing is the fact that the division of NC into subclasses based on running times seems to obscure the real bottleneck for parallel computing, which is the number of processors required. An algorithm that requires n processors and $log^2 n$ running time is likely to be far more useful than one that requires n^2 processors and takes $\log n$, but the latter is in NC^1, the more restrictive (and hence presumably better) class.

3 On Circuit Depth and Space Complexity

In this section we point out a strong connection between depth of circuits with **bounded** fan-in and space complexity. It turns out that circuit depth and space complexity are *polynomially related.* In particular, we are able to prove that L (and in fact NL) falls within NC. For the sake of generality, we introduce two families of complexity classes, which can be thought of as a generalized version of NC. ¿From now on, we assume that all circuits are uniform.

Definition 14 ($\mathcal{DEPTH}/\mathcal{SIZE}$): *Let d, s be integer functions. Define $\mathcal{DEPTH}/\mathcal{SIZE}(d(\cdot), s(\cdot))$ to be the class of all languages that can be decided by a uniform family of* bounded *fan-in circuits with depth $d(\cdot)$ and size $s(\cdot)$.*

In particular, if we denote by poly the set of all integer functions bounded by a polynomial and by polylog the set of all integer functions bounded by a poly-logarithmic function (e.g. $f \in$ polylog iff $f(n) = O(\log^k n)$ for some $k \geq 0$), then using the above notation, the class NC can be viewed as $\mathcal{DEPTH}/\mathcal{SIZE}(\text{polylog}, \text{poly})$.

Definition 15 ($\mathcal{DEPTH}$): *Let d be an integer function. Define $\mathcal{DEPTH}(d(\cdot))$ to be the class of all languages that can be decided by a uniform family of* bounded *fan-in circuits with depth $d(\cdot)$.*

Clearly, $NC \subseteq \mathcal{DEPTH}(\text{polylog})$. Note that the size of circuits deciding the languages in $\mathcal{DEPTH}(d(\cdot))$ is not limited, except for the natural upper bound of $2^{d(\cdot)}$ (due to the bounded fan-in)[1]. However, circuits deciding languages in NC are of polynomial size. Therefore, the class $\mathcal{DEPTH}(\text{polylog})$ contains languages which potentially do not belong to NC. We are now ready to connect circuit depth and space complexity.

Theorem 16 *For every integer function $s(\cdot)$ which is at least logarithmic,*

$$\mathcal{NSPACE}(s) \subseteq \mathcal{DEPTH}/\mathcal{SIZE}(O(s^2), 2^{O(s)})$$

Proof: Given a non-deterministic $s(n)$-space Turing machine M, we construct a uniform family of circuits, $\{C_n\}$, of depth $O(s^2)$ and size $2^{O(s)}$ such that for every $x \in \{0,1\}^*$, $C_{|x|}(x) = M(x)$.

Consider the configuration graph, $G_{M,x}$, of M on input x (see Lecture 6). Recall that the vertices of the graph are all the possible configurations of M on input x, and a directed edge leads from one configuration to another if and only if they are possible consecutive configurations of a computation on x. In order to decide whether M accepts x it is enough to check whether there exists a directed path in $G_{M,x}$ leading from the initial configuration vertex to the accepting configuration vertex. The problem of deciding, given a graph and two of its vertices, whether there exists a directed path between them, is called the directed connectivity problem (denoted $CONN$, see also Lecture 6). It turns out that $CONN$ can be decided by circuits with poly-logarithmic depth. More precisely:

Claim 3.1 $CONN \in NC^2$

[1]Thus, an alternative way to define $\mathcal{DEPTH}(d(\cdot))$ would be $\mathcal{DEPTH}/\mathcal{SIZE}(d(\cdot), 2^{d(\cdot)})$.

Proof: Let G be a directed graph on n vertices and let A be the adjacency matrix corresponding to it; that is, A is a boolean matrix of size $n \times n$, and $A_{i,j} = 1$ iff there is a directed edge from vertex v_i to vertex v_j in G. Now, let B be $A+I$, i.e. add to A all the self loops. Consider now the boolean product of B with itself, defined as

$$B^2_{i,j} = \bigvee_{k=1}^{n} (B_{i,k} \wedge B_{k,j}) \tag{1}$$

The resulting matrix will satisfy that $B^2_{i,j} = 1$ if and only if there is a directed path of length 2 or less from v_i to v_j in G. Similarly, B^4's entries will denote the existence of paths in G of length up to 4, and so on. Using $\log n$ boolean multiplications we can compute the matrix B^n, which is the adjacency matrix of the transitive closure of A - containing the answers to all the possible connectivity questions (for every pair of vertices in G). Squaring a matrix of size $n \times n$ can be done in AC^0 (see Equation 1) and therefore in NC^1. Hence, computing B^n can be done via repeated squaring in NC^2. ■

The circuit we build is a composition of two circuits. The first circuit takes as input some $x \in \{0,1\}^n$, a description of M (which is of a constant length) and outputs the adjacency matrix of $G_{M,x}$. The second circuit takes as input the adjacency matrix of $G_{M,x}$ and decides whether there exists a directed path from the initial configuration vertex to the accepting configuration vertex (i.e. decides $CONN$ on $G_{M,x}$). We start by constructing the first circuit. Given x and the description of M we generate all the possible configurations of M on x (there are $2^{O(s)}$ such configurations, each is represented by $O(s)$ bits). Then, for each pair of configurations we can decide if there should be a directed edge between them (i.e. whether these are possible consecutive configurations). This is done basically by comparing the contents of the work tape in the two configurations, and requires depth $O(\log s)$ (note that the size of the resulting circuit will be $2^{O(s)}$). As for the second circuit, since $G_{M,x}$ is of size $2^{O(s)}$, we have (by Claim 3.1) that $CONN$ can be decided on $G_{M,x}$ in depth $O(s^2)$ and size $2^{O(s)}$. Overall, we obtain a circuit C_n of depth $O(s^2)$ and of size $2^{O(s)}$, such that $C_n(x) = M(x)$. ■

Corollary 17 $NL \subseteq NC^2$

Proof: Take $s(n) = \log n$, and get that NL can be decided by circuits of polynomial size and of depth $O(\log^2 n)$. ■

Using the fact that $\mathcal{DEPTH/SIZE}(O(s^2), 2^{O(s)})$ is contained in $\mathcal{DEPTH}(O(s^2))$, we can conclude:

Corollary 18 *For every integer function $s(\cdot)$ which is at least logarithmic,*

$$\mathcal{NSPACE}(s) \subseteq \mathcal{DEPTH}(O(s^2))$$

We are now ready to establish a reverse connection between circuit depth and space complexity. This is done by proving a result which is close to being a converse to Corollary 18 (given that for every function $s(\cdot)$, $\mathcal{DSPACE}(s) \subseteq \mathcal{NSPACE}(s)$).

Theorem 19 *For every function $d(\cdot)$ which is at least logarithmic,*

$$\mathcal{DEPTH}(d) \subseteq \mathcal{DSPACE}(d)$$

Proof: Given a uniform family of circuits $\{C_n\}$ of depth $d(n)$, we construct a deterministic $d(n)$-space Turing machine M such that for every $x \in \{0,1\}^*$, $M(x) = C_{|x|}(x)$. Our algorithm will be the composition of two algorithms, each using $d(n)$ space. The following lemma states that the above composition will give us a $d(n)$-space algorithm, as required (this is a special case of the Composition lemma - the search version, see Lecture 6).

Lemma 3.2 *Let M_1 and M_2 be two $s(n)$-space Turing machines. Then, there exists an $s(n)$-space Turing machine M, that on input x outputs $M_2(M_1(x))$.*

Our algorithm is (given input $x \in \{0,1\}^n$):

1. Obtain a description of C_n.
2. Evaluate $C_n(x)$.

A description of a circuit is a list of its gates, where for each gate we specify its type and its list of predecessors. Note that the length of the description of C_n might be exponential in $d(n)$ (since the number of gates in C_n might be exponential in $d(n)$), therefore we must use Lemma 3.2. The following claims establish the Theorem:

Claim 3.3 *A description of C_n can be generated using $O(d(n))$ space.*

Claim 3.4 *Circuit evaluation for bounded fan-in circuits can be solved in space=O(circuit depth).*

Proof: (of Claim 3.3) By the uniformity of $\{C_n\}$, there exists a deterministic machine M such that $M(1^n) = desc(C_n)$, while using $\log(|desc(C_n)|)$ space. Since $|desc(C_n)| \leq 2^{O(d(n))}$, we get that M uses $O(d(n))$ space, as required. ■

Proof: (of Claim 3.4) Given a circuit C of depth d and an input x, we want to compute $C(x)$. Our implementation will be recursive. A natural approach would be to use the following algorithm:

Denote by $VALUE(C_x, v)$ the value of vertex v in the circuit C when assigning x to its inputs, where v is encoded in binary (i.e. using $\log(size(C)) = O(d)$ bits). Note that $VALUE(C_x,\text{'}output\text{'})$ is equal to the desired value, $C(x)$. The following procedure obtains $VALUE(C_x, v)$:

1. If v is a leaf then return the value assigned to it.
   Otherwise, let u and w be v's predecessors and **op** be v's label.
2. Compute recursively $\sigma \leftarrow VALUE(C_x, u)$ and $\tau \leftarrow VALUE(C_x, w)$.
3. Return σ **op** τ.

Notice that Step 2 in the algorithm requires two recursion calls. Since we are going only one level down each recursion call, one may hastily conclude that the space consumed by the algorithm is $2^{O(d)}$. Remember however, that we are dealing with space, this means that the space consumed in the first recursion call can be reused by the second one, and therefore the actual space consumption is $O(d^2)$ (since there are $O(d)$ levels in the recursion, and on each level we need to memorize the vertex name which is of length $O(d)$).

This is still not good enough, remember that our goal is to design an algorithm working in space $O(d)$. This will be done by representing the vertices of C in a different manner: Each vertex will be specified by a path reaching it from the output vertex. The output vertex will be represented by the empty string ϵ. Its left predecessor will be represented by "0", and its right predecessor by

"1", the left predecessor of "1" will be represented by "10", and so on. Since there might be several paths reaching a vertex, it might have multiple names assigned to it, but this will not bother us.

Consequently, each vertex is represented by a bit string of length $O(d)$. Moreover, obtaining from a vertex name its predecessor's or successor's name is done simply by concatenation of a bit or deletion of the last bit. The following procedure computes $VALUE(C_x, \overline{path})$ in $O(d)$ space:

1. Check whether $\overline{path}$ defines a leaf. If it does, then return the value assigned to it. Otherwise, let **op** be the label of the corresponding vertex.

2. Compute recursively $\sigma \leftarrow VALUE(C_x, \overline{path} \circ 0)$ and $\tau \leftarrow VALUE(C_x, \overline{path} \circ 1)$.

3. Return σ **op** τ.

When computing this procedure, C_x will be written on the input tape and $\overline{path}$ will be written on the work tape. At each level of the recursion, $\overline{path}$ determines precisely all the previous recursion levels, so the space consumption of this algorithm is $O(d)$, as required. ■

■

Corollary 20 $NC^1 \subseteq L \subseteq NL \subseteq NC^2$

Proof: The first inclusion follows from Theorem 19, the second inclusion is trivial and the third inclusion is just Corollary 17. ■