Lecture 20

# Boolean circuits

In this lecture we will discuss the *Boolean circuit* model of computation and its connection to the Turing machine model. Although the Boolean circuit model is fundamentally important in theoretical computer science, we will not have time to explore it in depth—our study of Boolean circuits will mainly be aimed toward obtaining a "primordial" NP-complete language through which many other languages can be proved NP-complete.

## 20.1 Definitions

You probably already have something in mind when the term *Boolean circuit* is suggested—presumably a circuit consisting of AND, OR, and NOT gates connected by wires. Indeed this is essentially what the model represents, with the additional constraint that Boolean circuits must be *acyclic*. Formally speaking, we represent Boolean circuits with *directed acyclic graphs*, as the following definition suggests.

**Definition 20.1.** An $n$-input, $m$-output Boolean circuit $C$ is a directed acyclic graph, along with various labels associated with its nodes, satisfying the following constraints:

1. The nodes of $C$ are partitioned into three disjoint sets: the set of *input nodes*, the set of *constant nodes*, and the set of *gate nodes*.
2. The set of input nodes has precisely $n$ members, labeled $X_1, \ldots, X_n$. Each input node and must have in-degree 0.
3. Each constant node must have in-degree 0, and must be labeled either 0 or 1.
4. Each gate node of $C$ is labeled by an element of the set $\{\wedge, \vee, \neg\}$. Nodes labeled $\wedge$ are called AND gates and must have in-degree 2, nodes labeled $\vee$ are called

OR gates and must have in-degree 2, and nodes labeled $\neg$ are called NOT gates and must have in-degree 1.

5. Finally, $m$ nodes of $C$ are identified as output nodes, and are labeled $Y_1, \ldots, Y_m$. An output node can be an input node, a constant node, or a gate node.

When it is said that a directed graph is *acyclic*, it is meant that there is no path in the graph from a node to itself. Because this restriction is in place for Boolean circuits, we cannot have circuits with feedback loops, so it is not possible to implement latches, flip-flops, and so on. This is not a problem because our interest is in describing computations, not hardware. You can reasonably view each gate in a Boolean circuit as representing one computational step rather than one piece of hardware.

A Boolean circuit $C$ having $n$ inputs and $m$ outputs computes a function of the form

$$f : \{0,1\}^n \to \{0,1\}^m \tag{20.1}$$

in the most natural way. In particular, for an input $(x_1, \ldots, x_n) \in \{0,1\}^n$ we first assign a Boolean value to each node as follows:

1. The value of each input node $X_k$ is $x_k$, for $k = 1, \ldots, n$, and the value of each constant node is either 0 or 1, according to its label.
2. The value of each gate node is determined by the gate type and the value of the node or nodes from which there exist incoming edges. (For instance, if $u$ is an AND gate node, and $v_1$ and $v_2$ are the nodes from which there exist edges to $u$, then the value of $u$ is given by the AND of the values of $v_1$ and $v_2$. The situation is similar for OR and NOT gate nodes.)

Once the value of each of the nodes has been defined, we define the output of the function $f$ by setting $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$, where $y_k$ is the value assigned to output node $Y_k$, for $k = 1, \ldots, m$. For example, Figure 20.1 illustrates a Boolean circuit that computes the exclusive-OR of two input Boolean values. (In this case there are no constant nodes.)

It is standard to use whatever name has been assigned to a Boolean circuit to refer also to the function it computes—so for an $n$-input Boolean circuit $C$, we might write $C(x_1, \ldots, x_n)$ rather than introducing a new name $f(x_1, \ldots, x_n)$ to refer to the function $C$ computes. This is purely a shorthand convention used for convenience, and there is no harm in using distinct names for a Boolean circuit and the function it computes if there is a reason to do this.

The *size* of a Boolean circuit $C$ is defined to be its number of nodes, and we write $\text{size}(C)$ to represent this number. The circuit illustrated in Figure 20.1, for instance, has size 7.

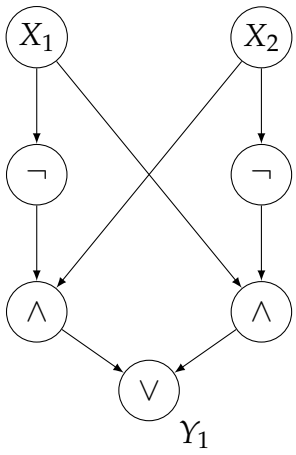


Figure 20.1: A Boolean circuit that computes the function $f(x_1, x_2) = x_1 \oplus x_2$ (where $\oplus$ denotes the exclusive-OR).

There are a variety of ways that one could choose to encode Boolean circuits as strings. By assigning a unique number to each node, and then listing each of the nodes one at a time (including their numbers, labels, and the numbers of the vertices from which incoming edges exist), it is not difficult to come up with an encoding scheme whereby each Boolean circuit $C$ has an encoding $\langle C \rangle$ such that

$$|\langle C \rangle| = O(N \log(N)) \tag{20.2}$$

for $N = \text{size}(C)$.

## 20.2 Universality of Boolean circuits

We will now observe that Boolean circuits are universal, in the sense that *every* Boolean function of the form

$$f : \{0,1\}^n \to \{0,1\}^m, \tag{20.3}$$

for positive integers $n$ and $m$, can computed by some Boolean circuit $C$.

Let us begin by observing that if we can build a Boolean circuit for every function of the form

$$g : \{0,1\}^n \to \{0,1\} \tag{20.4}$$

(i.e., the $m = 1$ case of the more general fact we're aiming for), then we can easily handle the general case. This is because every Boolean function of the form (20.3) can always be expressed as

$$f(x_1, \ldots, x_n) = (g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) \tag{20.5}$$

for some choice of functions $g_1, \ldots, g_m$, each of the form

$$g_k : \{0,1\}^n \to \{0,1\}; \tag{20.6}$$

and once we have a Boolean circuit $C_k$ that computes $g_k$, for each $k = 1, \ldots, m$, we can combine these circuits in a straightforward way to get a circuit $C$ computing $f$. In particular, with the exception of the input nodes $X_1, \ldots, X_n$, which are to be shared among the circuits $C_1, \ldots, C_m$, we allow the circuits $C_1, \ldots, C_m$ to operate independently, and label their output nodes appropriately.

It therefore remains to show that for any given function $g$ of the form (20.4), there is a Boolean circuit $C$ that computes $g$. This can actually be done using a rather simple *brute force* type of approach: we list all of the possible settings for the input variables that cause the function to evaluate to 1, represent each individual setting by an AND of input variables or their negations, and then take the OR of the results.

For example, consider the case that $n = 3$ and $g : \{0,1\}^3 \to \{0,1\}$ is a function such that

$$g(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } x_1x_2x_3 \in \{001, 010, 111\} \\ 0 & \text{otherwise.} \end{cases} \tag{20.7}$$

The condition that $x_1x_2x_3 = 001$ is represented by the formula

$$(\neg X_1) \wedge (\neg X_2) \wedge X_3, \tag{20.8}$$

the condition that $x_1x_2x_3 = 010$ is represented by the formula

$$(\neg X_1) \wedge X_2 \wedge (\neg X_3), \tag{20.9}$$

and the condition that $x_1x_2x_3 = 111$ is represented by the formula

$$X_1 \wedge X_2 \wedge X_3. \tag{20.10}$$

The function $g$ is therefore represented by the formula

$$\big((\neg X_1) \wedge (\neg X_2) \wedge X_3\big) \vee \big((\neg X_1) \wedge X_2 \wedge (\neg X_3)\big) \vee \big(X_1 \wedge X_2 \wedge X_3\big). \tag{20.11}$$

Once we have this formula, it is easy to come up with a Boolean circuit that computes the formula, which is the same as computing the function. (Because AND and OR gates are supposed to have just two inputs, we must combine two such gates to compute the AND or OR of three inputs, and of course this can be generalized to ANDs and ORs of more than three inputs. Aside from this, there is little work to do in turning a formula such as the one above into a Boolean circuit.)

There is one special case that we should take note of, which is the function that takes the value 0 on all inputs. There is nothing to take the OR of in this case, so it doesn't fit the pattern described above—but we can instead make use of a constant node whose value is 0 to implement a circuit whose output value is always 0. Alternatively, we could simply implement the formula

$$X_1 \wedge (\neg X_1) \tag{20.12}$$

as a Boolean circuit to deal with this function.

The type of Boolean formula of which (20.11) is an example, where we have an OR of some number of expressions, each of which is an AND of a collection of variables or their negations, is said to be a formula in *disjunctive normal form*. If we switch the roles of AND and OR, so that we have an AND of some number of expressions, each being an OR of a collection of variables or their negations, we have a formula in *conjunctive normal form*. What we've argued above is that every Boolean formula in variables $X_1, \ldots, X_n$ can be expressed in disjunctive normal form (or as a *DNF formula*, for short). One can also show that every Boolean formula can be expressed in conjunctive normal form (or as a *CNF formula*): just follow the steps above for $\neg g$ in place of $g$, negate the entire formula, and let De Morgan's laws do the work.

One thing you may notice about the method described above for obtaining a Boolean circuit that computes a given Boolean function is that it generally results in very large circuits. While there will sometimes exist a much smaller Boolean circuit for computing a given function, it is inevitable that some functions can only be computed by very large circuits. The reason is that the number of different functions of the form $g : \{0,1\}^n \to \{0,1\}$ is doubly exponential—there are precisely

$$2^{2^n} \tag{20.13}$$

functions of this form, as $g(x)$ can be 0 or 1 for each of the $2^n$ choices of $x \in \{0,1\}^n$. In contrast, the number of Boolean circuits of size $N$ is merely exponential in $N$. As a consequence, one finds that some functions of the form $g : \{0,1\}^n \to \{0,1\}$ can only be computed by Boolean circuits having size exponential in $n$ (and in fact this is true for most functions of the form $g : \{0,1\}^n \to \{0,1\}$).

## 20.3 Boolean circuits and Turing machines

The last thing we will do in this lecture is to connect Boolean circuits with Turing machines, in a couple of different ways.

Let us start with a simple observation concerning the following language:

$$\mathrm{CVP} = \left\{ \langle C, x \rangle \;:\; \begin{array}{l} C \text{ is an } n\text{-input, 1-output Boolean} \\ \text{circuit, } x \in \{0,1\}^n, \text{ and } C(x) = 1 \end{array} \right\}. \tag{20.14}$$

The name CVP is short for *circuit value problem*. It is the case that this problem is contained in P; a DTM can decide CVP in polynomial time by evaluating the nodes of $C$ and then accepting if and only if the unique output node evaluates to 1. In short, a DTM can easily simulate a Boolean circuit on a given input by simply evaluating the circuit.

The other direction—meaning the simulation of DTMs by Boolean circuits—is a bit more challenging. Moreover, it requires that we take a moment to clarify what is meant by the simulation of a DTM by a Boolean circuit. In particular, we must observe that there is a fundamental difference between Boolean circuits and DTMs, which is that DTMs are ready to handle input strings of any length, but Boolean circuits can only take inputs of a single, fixed length.[1] The most typical way to address this problem is to consider *families* of Boolean circuits, one for each possible input length, rather than individual Boolean circuits.

For example, suppose that we have a language $A \subseteq \{0,1\}^*$, and we wish to understand something about the *circuit complexity* of this language. A single Boolean circuit $C$ cannot decide $A$, because $C$ expects some fixed number of input bits, as opposed to a binary string having arbitrary length. We can, however, consider a family of circuits

$$\mathcal{F} = \{C_n \;:\; n \in \mathbb{N}\}, \tag{20.15}$$

where $C_n$ is a Boolean circuit with $n$ inputs and 1 output for each $n \in \mathbb{N}$. Such a family decides $A$ if it is the case that

$$A = \{x \in \{0,1\}^* \;:\; C_n(x) = 1 \text{ for } n = |x|\}. \tag{20.16}$$

In words, on an input string $x \in \{0,1\}^*$ of length $n$, we find the circuit $C_n$ from $\mathcal{F}$ that expects an input of length $n$, feed $x$ into this circuit, and interpret the outputs 0 and 1 as accept and reject, respectively.

One word of caution: *every* language $A \subseteq \{0,1\}^*$ is decided by some family of Boolean circuits in the sense just described, so you cannot conclude that a language is decidable from the existence of a family of circuits that decides it. We can, however, ask whether certain languages have families of Boolean circuits that obey various special constraints. The following theorem, which represents the main fact we're aiming for in this section, provides an example.

[1] Another difference is that DTMs can have arbitrary input alphabets, while Boolean circuits only work for Boolean values. This, however, is somewhat secondary, as we can always imagine that the strings of a language over an arbitrary alphabet have been encoded as binary strings, which will not change the properties of the language that are most relevant to complexity theory.

**Theorem 20.2.** *Let $f : \mathbb{N} \to \mathbb{N}$ be a time-constructible function, and let $M$ be a DTM with input alphabet $\{0,1\}$ whose running time $t(n)$ satisfies $t(n) \leq f(n)$ for all $n \in \mathbb{N}$. There exists a family*

$$\mathcal{F} = \big\{C_n \,:\, n \in \mathbb{N}\big\} \tag{20.17}$$

*of Boolean circuits, where each circuit $C_n$ has $n$ inputs and 1 output, such that the following two properties are satisfied:*

1. *For every $n \in \mathbb{N}$ and every $x \in \{0,1\}^n$, $M$ accepts $x$ if and only if $C_n(x) = 1$.*
2. *It holds that $\text{size}(C_n) = O(f(n)^2)$.*

*Moreover, there exists a DTM $K$ that runs in time polynomial in $f(n)$ that outputs an encoding $\langle C_n \rangle$ of the circuit $C_n$ on input $0^n$, for each $n \in \mathbb{N}$.*

**Remark 20.3.** We will be primarily interested in the case in which $f(n) = O(n^c)$ for some fixed choice of an integer $c \geq 2$, and in this case the DTM $K$ that outputs a description $\langle C_n \rangle$ of the circuit $C_n$ on each input $0^n$ runs in polynomial time.

*Proof.* The proof is essentially a description of the circuit $C_n$ for each choice of $n \in \mathbb{N}$. After the construction of $C_n$ has been explained, it may be observed that $\text{size}(C_n) = O(f(n)^2)$ and that a DTM could output a description of $C_n$ as claimed in the statement of the theorem.

We will start with a few low-level details concerning encodings of the states and tape symbols of $M$. Assume that the state set of $M$ is $Q$, and let $k = \lceil \log_2(|Q| + 1) \rceil$. This means that $k$ bits is just large enough to encode each element of $Q$ as a binary string of length $k$ without using the string $0^k$ to encode a state—we're going to use this all-zero string in a few moments for a different purpose. Aside from the requirement that each state is encoded by a different string of length $k$, and that we haven't used the string $0^k$ to encode a state, the specific encoding used can be selected arbitrarily.

Let us also do something similar with the tape symbols of $M$. This time we'll use the all-zero string to encode the blank symbol $\sqcup$ rather than using it for a different purpose. That is, if the tape alphabet of $M$ is $\Gamma$, then we will encode the tape symbols as binary strings of length $m = \lceil \log_2(|\Gamma|) \rceil$, using the string $0^m$ to encode the blank symbol $\sqcup$. The other symbols can be encoded arbitrarily, so long as each symbol is encoded by a different string of length $m$.

With the encodings of states and tape symbols just described in mind, we can imagine that any one single tape square of $M$ at any moment in time could be

represented by $m + k$ bits in the following manner:

$$\underbrace{\square\ \square\ \square\ \square\ \cdots\ \square\ \square\ \square}_{m \text{ bits to encode a tape symbol}}\ \underbrace{\square\ \square\ \square\ \square\ \cdots\ \square\ \square\ \square}_{k \text{ bits to encode a state (if the tape head is here)}} \tag{20.18}$$

(In this picture, each square represents one bit.) For the $k$ rightmost bits, we associate these meanings to the bit values:

1. The string $0^k$ means the tape head is not positioned over this particular tape square, and that nothing can be concluded regarding the state of the DTM.
2. Any string other than $0^k$ indicates that the tape head is scanning this square, and the state of the DTM is determined by these $k$ bits with respect to our encoding scheme for states.

It is now evident why we did not want to use the string $0^k$ to encode a state, as this string is being used to indicate the absence of the tape head (and no information concerning the state).

Next, as we work our way toward a description of the circuit $C_n$, for any choice of $n \in \mathbb{N}$, we imagine an array of bits as suggested by Figure 20.2. To be more precise, the array contains $f(n) + 1$ rows, where each row represents a configuration of $M$ at one step of its computation: the first row represents the initial configuration, the second row represents the configuration of $M$ after one step, and so on. Within each row there are $2f(n) + 1$ blocks—each block represents one tape square, and taken together the blocks represent the tape squares numbered $-f(n), \ldots, f(n)$, relative to the starting location of the tape head (which we view as position 0). Each block consists of $m + k$ bits, and represents a tape symbol and possibly the presence of the tape head and state, as described above. We have chosen to include only blocks for the tape squares numbered $-f(n), \ldots, f(n)$, for the times $0, \ldots, f(n)$, because the computation of $M$ on an input of length $n$ is limited to this region of the tape: the computation lasts no more than $f(n)$ steps, and the tape head cannot possibly escape the region of the tape represented by the tape squares $-f(n), \ldots, f(n)$ within $f(n)$ steps.

What we wish to do next is to essentially hook up all of these bits to form the Boolean circuit $C_n$ that will simulate $M$ on a given input of length $n$ for $f(n)$ steps. The key observation needed to do this is that all of the changes in the configurations represented by the rows of our array are "local." That is, if we want to know the contents of one of the blocks of $m + k$ bits at a particular time, we need only look at three blocks above that block; if the block in question represents tape square $s$ at time $t$, the only blocks we need to examine in order to determine the correct

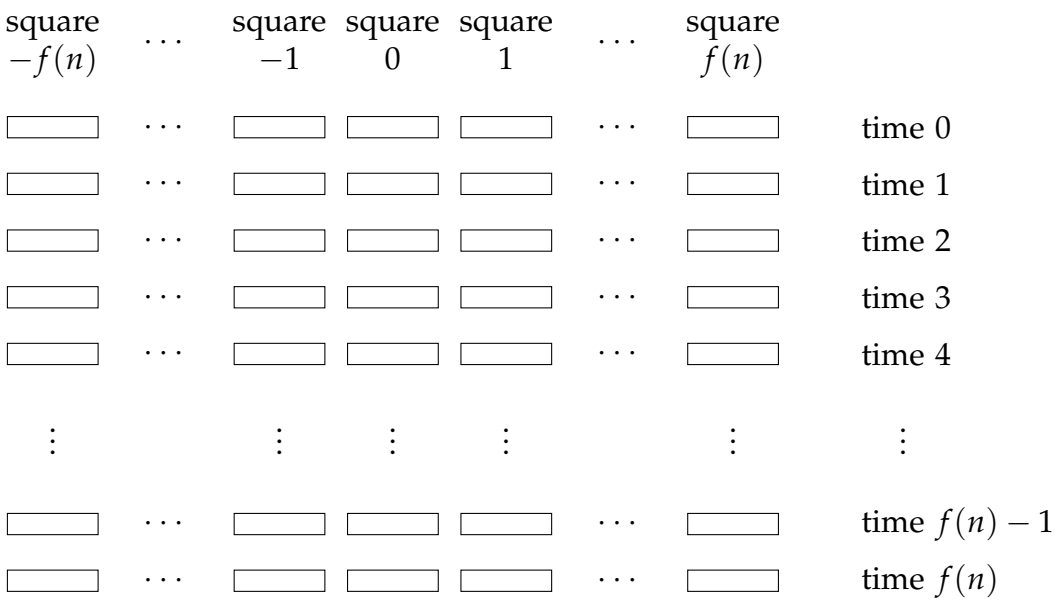


Figure 20.2: An array of bits representing configurations of the DTM $M$ during the first $f(n)$ steps of its computation. Each rectangle correspond to one tape square at one time during the computation, and represents $m+k$ bits.

setting of the bits in this block are the ones corresponding to squares numbered $s-1$, $s$, and $s+1$, at time $t-1$. This dependence can represented by some Boolean function of the form

$$g : \{0,1\}^{3(m+k)} \to \{0,1\}^{m+k}. \tag{20.19}$$

It is not necessary for us to describe precisely how this function $g$ is defined—it is enough to observe that it is determined by the transition function of $M$. It may, however, be helpful to make some observations about this function in order to clarify the proof.

Each of the points that follow refer to the arrangement of blocks suggested by Figure 20.3, in which $u$, $v$, and $w$ are binary strings of length $m+k$ and represent the tape squares $s-1$, $s$, and $s+1$, respectively (each at time $t-1$), while $g(uvw)$ represents tape square $s$ at time $t$.

1. There is only one tape head of $M$, so we don't need to worry about the case in which more than one of the strings $u$, $v$, and $w$ indicates the presence of the tape head—the function $g$ can be defined arbitrarily in any such case. Similarly, if $u$, $v$, or $w$ includes an improper encoding of a tape symbol or state, the function $g$ can be defined arbitrarily.

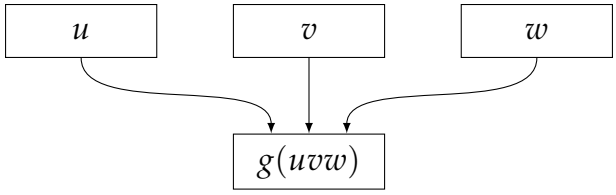


Figure 20.3: The correct values for the block of $m + k$ bits describing a tape square at a particular time depend only on the blocks of bits describing the same tape square and its adjacent tape squares one time step earlier. This dependence can be represented by some function $g : \{0,1\}^{3(m+k)} \to \{0,1\}^{m+k}$.

2. If none of $u$, $v$, or $w$ indicates the presence of the tape head, it will be the case that $g(uvw) = v$. This reflects the fact that there's no action happening around the tape square $s$ at this point in the computation.
3. If $v$ indicates the presence of the tape head (and therefore specifies the state of $M$ as well), then the symbol stored in tape square $s$ might change, depending on the transition function of $M$. The first $m$ bits of the output of $g$ will need to be set appropriately, and it so happens that these first $m$ output bits will be a function of $v$ alone. Under the assumption that the state is a non-halting state, we also know that the last $k$ bits of the output of $g$ will be set to $0^k$, as the tape head is forced to move left or right, off of square $s$.
4. If either $u$ or $w$ indicates the presence of the tape head (and therefore the state of $M$), then we might or might not end up with the tape head on square $s$, so the last $k$ bits of the output of $g$ will need to be set appropriately. The first $m$ bits of the output of $g$ will be in agreement with the first $m$ bits of $v$ in this case, as the symbol stored in square $s$ does not change—the symbol can only change on the square the tape head is scanning.
5. We should be sure to set $g(uvw) = v$ in case $v$ indicates the presence of the tape head and the state being either an accept or reject state. This accounts for the possibility that $M$ finished its computation in fewer than $f(n)$ steps, allowing the correct answer to be propagated to the output of the circuit $C_n$ we are constructing.

Now, because $g$ is a Boolean function, we know that it must be implemented by some Boolean circuit (which we will call $G$). Because $m$ and $k$ are fixed—they only depend on the DTM $M$ and not on the input length $n$ of a string input to $M$—the size of $G$ is also constant, independent of $n$.

Once we have in mind a description of the Boolean circuit $G$, we can envision a large Boolean circuit $C_n$ in which the blocks of bits in the array described above are

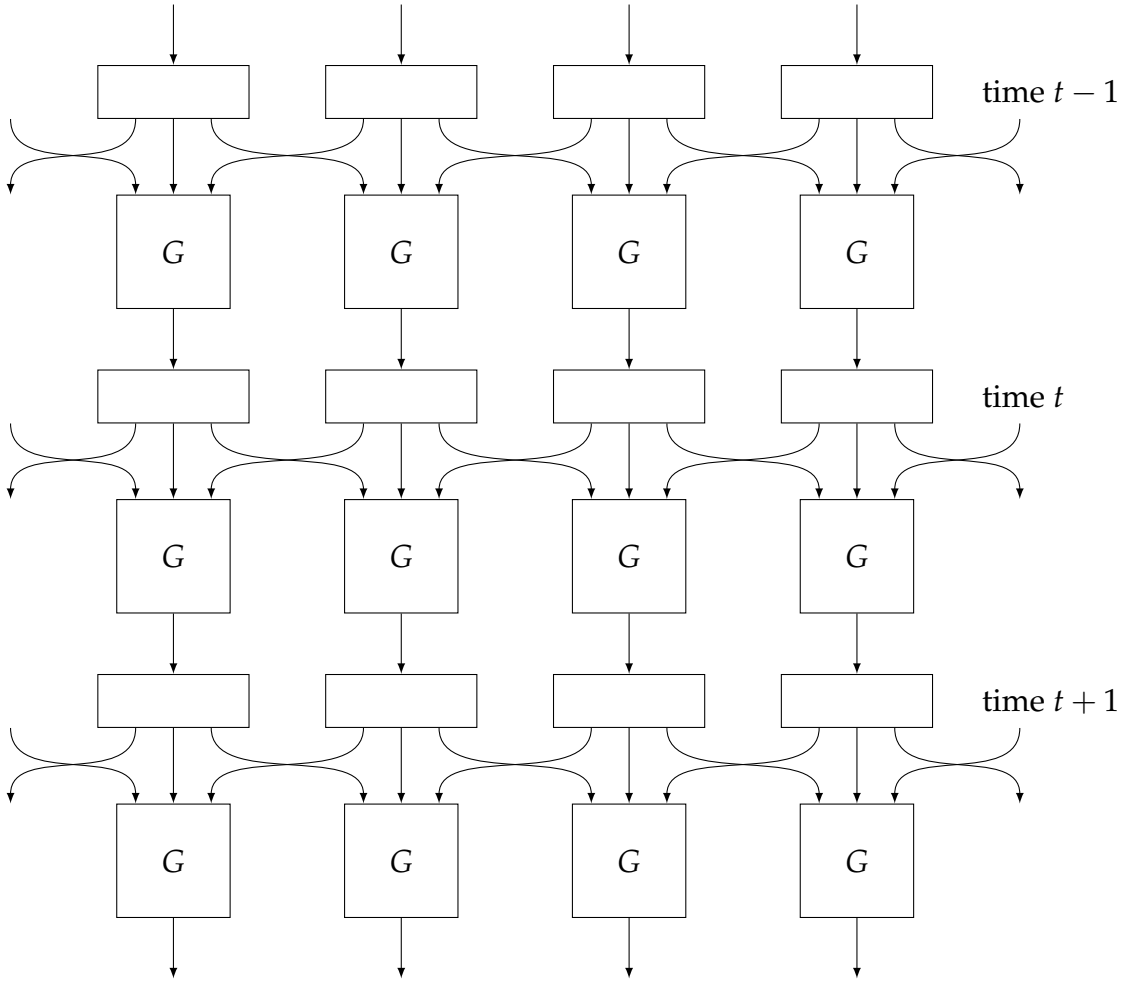


Figure 20.4: The pattern of connections among the blocks representing tape squares and the copies of the circuit $G$ that computes the function $g$.

connected together, as suggested by Figure 20.4. In order to deal with the boundaries, for which hypothetical inputs correspond to blocks not included in our array, we introduce constant nodes giving the correct Boolean values (which are all-zero inputs given our encoding scheme choices).

It remains to create two additional parts of the circuit $C_n$: one part for an initialization stage at the beginning and one part that computes a single output bit corresponding to acceptance or rejection. The actual input string of length $n$ upon which we wish to simulate $M$ should correspond to $n$ input bits to the circuit $C_n$. It is not difficult to make use of constant nodes and $n$ copies of a constant-size Boolean circuit so that $n$ input nodes $X_1, \ldots, X_n$ are connected to the blocks at time 0 numbered $1, \ldots, n$ in such a way that these blocks are initialized appropriately

(so that the top row of blocks correctly represents the initial configuration of $M$ on whatever input string is input to $C_n$). For all of the other blocks at time 0, constant nodes may be used for a correct initialization. Lastly, one may imagine another constant-size circuit that takes $m + k$ input bits and outputs 1 bit, which is to take the value 1 if and only if the $m + k$ input bits represent a block corresponding to a tape square in which the tape head is present and the state is the accept state. The output of the circuit $C_n$ being constructed is then given by the OR of all of these output bits taken over all of the blocks at time $f(n)$. In other words, the output of $C_n$ is 1 if and only if $M$ is in an accept state at time $f(n)$ after being run on whatever input of length $n$ has been input to $C_n$.

The size of the circuit $C_n$ just described is evidently $O(f(n)^2)$. A description of this circuit can certainly be output by a DTM $M$ in time polynomial in $f(n)$. Indeed, the structure of the circuit has a very simple, regular pattern, with the only dependence on $M$ being the description of the constant-size circuits (including $G$ and the circuit used to initialize the blocks in the top row). □

While there are some low-level details of the circuit $C_n$ that have been left to the imagination in the proof above, hopefully the main idea is clear enough that you could (if you were forced to do it) come up with a more detailed description.