The Complexity of Non-uniform Problems

14.1 Fundamental Considerations

Although we have not made this explicit, our considerations to this point have been directed toward software solutions. If we want to design an efficient algorithm for an optimization problem like TSP or KNAPSACK, we are thinking of an algorithm that works for arbitrarily many cities or objects. When designing hardware, however, the situation is different. If a processor works with 64-bit numbers, then a divider for 64-bit numbers is supposed to compute the first 64 bits of the quotient.

The corresponding computational model is the *circuit*. Circuits for inputs of length n have Boolean variables $x_1, \dots, x_n$ and Boolean constants 0 and 1 as inputs. They can be described as a sequence $G_1, \dots, G_s$ of gates. Each gate G_i has two inputs $E_{i,1}$ and $E_{i,2}$ that must be among the previous gates $G_1, \dots, G_{i-1}$ and the inputs. The gate G_i applies a binary operation op_i to its inputs. The functions that are computed by such circuits arise naturally. The input variables x_i and the Boolean constants 0 and 1 can be considered as functions as well. If the inputs to a gate G_i are realized by the functions $g_{i,1}$ and $g_{i,2}$, then gate G_i is realized by the function

$$g_i(a) := g_{i,1}(a) \,\mathrm{op}_i\, g_{i,2}(a) .$$

Circuits can be represented more visually as directed acyclic graphs. The inputs and gates form the vertices of the graph. Each gate has two in-coming edges representing its two inputs. In the general case, we must distinguish between the first and second input. If we restrict our attention to symmetric operators like AND, OR, and EXOR, then this is unnecessary. A circuit C realizes the function $f = (f_1, \dots, f_m)\colon \{0,1\}^n \to \{0,1\}^m$ if each component function f_j is realized by an input or a gate. A circuit for addition on three bits is represented in Figure 14.1.1. The sum bit is computed in G_4 and the "carry bit" in G_5.

For the evaluation of the efficiency of circuits two different measures are available. The *circuit size* (or just size) of a circuit is equal to the number

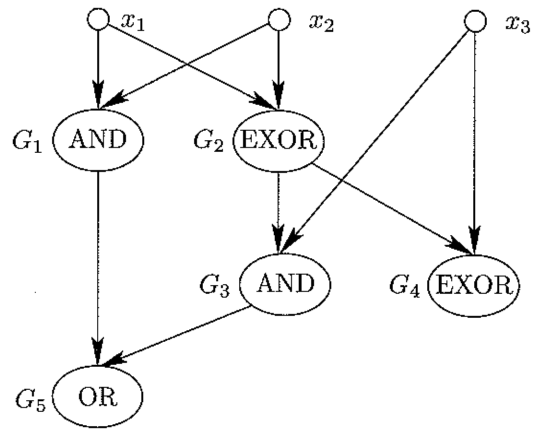

Fig. 14.1.1. A 3-bit adder.

of gates in the circuit and forms a measure of the hardware costs and the sequential computation time. We imagine that the gates are evaluated in the given order, and that the evaluation of each gate has cost 1. In reality, circuits are "parallel processors". In our example in Figure 14.1.1, gates G_1 and G_2 can be evaluated simultaneously, and once G_2 has been evaluated, G_3 and G_4 can be evaluated simultaneously. The depth of a gate is the length of the longest path from an input to that gate. All the gates with depth d can be evaluated simultaneously in the dth time step. The *circuit depth* (or just depth) of a circuit is the maximal depth of a gate in a given circuit. Our example adder has size 5 and depth 3.

Just as we have been concentrating our attention on decision problems, so here we will be primarily interested in Boolean functions $f : \{0,1\}^n \to \{0,1\}$ that have a single output. For the design of hardware, a particular input size may be important, but an asymptotic complexity theoretical analysis can only be based on a sequence $f = (f_n)$ of Boolean functions. A *circuit family* or sequence of circuits $C = (C_n)$ computes $f = (f_n)$ if each f_n is computed by C_n. This leads to the following relationship between decision problems on $\{0,1\}^*$ and sequences of Boolean functions $f = (f_n)$ with $f_n : \{0,1\}^n \to \{0,1\}$. For each decision problem there is a corresponding sequence of functions $f^A = (f_n^A)$ with

$$f_n^A(x) = 1 \Leftrightarrow x \in A\,.$$

On the other hand for any $f = (f_n)$ the decision problem A_f can be defined by

$$x \in A \Leftrightarrow f_{|x|}(x) = 1\,.$$

On the basis of this relationship, in this chapter we will only consider inputs over the alphabet $\{0,1\}$.

For a sequence $f = (f_n)$ of Boolean functions we want to analyze the complexity measures of size and depth. So let $C_f(n)$ denote the minimal size of a circuit that computes f_n and let $D_f(n)$ be defined analogously for circuit depth. In Chapter 2 we claimed that the time complexity of a problem is a robust measure. Does this imply that the time complexity of A and the circuit size of f_A are related? Boolean functions can always be represented in disjunctive normal form. A naive analysis shows that their size and depth are bounded by $n \cdot 2^n$ and $n + \lceil \log n \rceil$, respectively. This is true even for sequences of functions (f_n^A) for which A is not computable. There are even noncomputable languages that for each length n contain either all inputs of that length or none of them. Then f_n^A is a constant function for each n and so has size 0.

Here the difference between software solutions (algorithms) and hardware solutions like circuit families becomes clear. With an algorithm for inputs of arbitrary length we also have an algorithm for any particular length n. On the other hand, we need the entire circuit family to process inputs of arbitrary length. An algorithm has a finite description, as does a circuit, but what about a circuit family? For a noncomputable decision problem A the sequence of DNF circuits just described is not even computable.

An algorithm is a *uniform* description of a solution procedure for all input lengths. When we are interested in such solutions we speak of a uniform problem. A circuit family $C = (C_n)$ only leads to a uniform description of a solution procedure if we have an algorithm that can compute C_n from n. It is possible for there to be very small circuits C_n for f_n that are very hard to compute and larger circuits C'_n for f_n that are much easier to compute. A circuit family $C = (C_n)$, where C_n has size $s(n)$, is called *uniform* if C_n can be computed from n in $O(\log s(n))$ space. In this chapter when we speak of uniform families of circuits we will be content to show that S_n can be computed in time that is polynomial in $s(n)$. It is always easy, but sometime tedious, to describe how to turn this into a computation in logarithmic space.

Every decision problem A on $\{0,1\}^*$ has a non-uniform variant consisting of the sequence $f^A = (f_n^A)$ of Boolean functions. The non-uniform complexity measures are circuit size and circuit depth where non-uniform families of circuits are allowed. A non-uniform divider can be useful. If we need a 64-bit divider, it only needs to be generated or computed once and then can be used in many (millions of) processors. So we are interested in the relationships between uniform and non-uniform complexity measures. In Section 14.2 we will simulate uniform Turing machines with uniform circuits in such a way that time is related to size and space to depth. Circuits can solve noncomputable problems, so they can't in general be simulated by Turing machines. We will introduce non-uniform Turing machines that can efficiently simulate circuits. Once again time will be related to size, and space to depth. Together it turns out that time for Turing machines and size for circuits are very closely related. The relationships between space and depth (and so parallel computation time) are also amazingly tight, but circuits do not provide a model of non-uniform computation that asymptotically exactly mirror the resource of storage space. Such a model will be introduced in Section 14.4.

For complexity classes that contain P, one can ask if all of their problems can be solved by circuits of polynomial size. In Section 14.5 we will show that this is the case for the complexity class BPP. If a similar result holds for NP as well, we obtain a new possibility for dealing with difficult problems. But this is only possible, as we will show in Section 14.7, if the polynomial hierarchy collapses to the second level. Before that we present a characterization of non-uniform complexity classes in Section 14.6.

Circuits form a fundamental hardware model. Only uniform circuits lead to an efficient algorithmic solution. New aspects of the complexity of problems are captured by the non-uniform complexity measures of circuit size and circuit depth. From a practical perspective, it is important to know if a problem is difficult because it requires large circuits or because it is not possible to compute small circuits efficiently.

14.2 The Simulation of Turing Machines By Circuits

The goals of our considerations can be summarized as follows:

- Turing machines with small computation time can be simulated by uniform circuits with small size.
- Turing machines that use little space can be simulated by uniform circuits of small depth.

The first result compares the computation time of Turing machines with the time for the evaluation of a circuit. The second result implies that small space requirement makes possible an efficient computation via parallel processing and is a basis for the *parallel computation hypothesis* about the tight connection between storage space and parallel computation time.

What is the difficulty in simulating a Turing machine step by step with a circuit? Turing machines can incorporate branches (if-statements), and thus which tape cell is read at time t may depend on the input. It is true that configurations are only locally modified at each step, but where this modification occurs depends on the input. Oblivious Turing machines (see Definition 5.4.1) always read the same tape cell in the tth step regardless of the input. As we showed in Lemma 5.4.2, Turing machines can be simulated by oblivious Turing machines with only a quadratic slow-down. We mentioned there that one can actually get by with a logarithmic slow-down factor, i.e., that time $O(t(n)\log(t(n)))$ suffices for the simulation of any Turing machine by an oblivious Turing machine. So we will investigate how we can simulate oblivious Turing machines step by step with circuits.

The start configuration can be described by the input variables and Boolean constants at no cost. Assume we have described the first $t-1$ computation steps and consider step t. Only the state and the symbol on the tape cell being read may change in this step. Since the state space Q and

the tape alphabet Γ are finite, we only need constantly many bits of the description of the configuration to compute the new state and the new contents of the tape cell. More concretely, we are evaluating the Turing program $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$, where the third component is constant for a given t since we are considering only oblivious Turing machines. Even the disjunctive normal form realization of a circuit for δ has only constant size with respect to the input length n. This constant will depend only on the complexity of δ. Together we obtain a circuit of size $O(t(n))$ to simulate $t(n)$ steps of a Turing machine. The circuit is uniform if the tape head position in step t can be efficiently computed, as is the case for the oblivious Turing machines mentioned above. In summary, we have the following result.

Theorem 14.2.1. *An oblivious $t(n)$ time-bounded Turing machine can be simulated by uniform circuits of size $O(t(n))$. A $t(n)$ time-bounded Turing machine can be simulated by uniform circuits of size $O(t(n)\log t(n))$.* □

The corresponding circuits also have depth $O(t(n)\log(t(n))$. To get circuits with smaller depth we need a new idea.

Theorem 14.2.2. *An $s(n)$ space-bounded Turing machine can be simulated by uniform circuits of depth $O(s^*(n)^2)$, where $s^*(n) := \max\{s(n), \lceil \log n \rceil\}$.*

Proof. For space-bounded Turing machines we assume, as was described in Section 13.2, that the input is on a read-only input tape. The number of different configurations is bounded by $k(n) = 2^{O(\log n + s(n))} = 2^{O(s^*(n))}$. We consider the corresponding directed configuration graph that contains a vertex for each configuration. The edge set $E(x)$ depends on the input x. The edge (v, w) belongs to $E(x)$ if the Turing machine on input x can go from configuration v to configuration w in one step. Let the adjacency matrix of this graph be $A(x) = (a_{v,w}(x))$. It is important that $a_{v,w}(x)$ only depends on x_i in an essential way when the ith tape cell is being read in configuration v. So $a_{v,w}(x)$ is one of the functions 0, 1, x_i or $\overline{x}_i$. Thus each of the functions $a_{v,w}(x)$ can be computed by a circuit of depth 1. Now let $a^t_{v,w}(x) = 1$ if and only if on input x the configuration w can be reached from configuration v in t steps. For $t' \in \{1, \dots, t-1\}$ we must go from configuration v to configuration u and then in $t - t'$ steps from configuration u to w. Thus

$$a^t_{v,w}(x) = \bigvee_u a^{t'}_{v,u}(x) \wedge a^{t-t'}_{u,w}(x) ,$$

where $\bigvee$ represents disjunction. The matrix A is the Boolean matrix product of $A^{t'}$ and $A^{t-t'}$. The depth needed to realize this matrix product is $1 + \lceil \log k(n) \rceil = O(s^*(n))$. Each of the conjunctions requires depth 1, and for each of the disjunctions a balanced binary tree can be used. Again from Section 13.2 we know that we reach an accepting configuration only if we can reach it in $k(n) \le 2^{\lceil \log k(n) \rceil}$ steps. So we compute A^{2^i} for all $1 \le i \le \lceil \log k(n) \rceil$ with $\lceil \log k(n) \rceil = O(s^*(n))$ matrix multiplications. Finally we check if the input x

is accepted with a disjunction of all $a^{2^{\lceil \log k(n) \rceil}}_{v_0,w}(x)$ for the initial configuration v_0 and the accepting configurations w. The depth of this circuit is bounded by

$$1 + (1 + \lceil \log k(n) \rceil) \cdot \lceil \log k(n) \rceil + \lceil \log k(n) \rceil = O(s^*(n)^2) .$$

The corresponding circuits are uniform. The behavior of the Turing machine only plays a role in the computation of the $a_{v,w}(x)$. □

It is not known how to simulate Turing machines with small time and small space bounds with circuits of small size *and* small depth simultaneously. Most likely there are no such simulations.

14.3 The Simulation of Circuits by Non-uniform Turing Machines

Circuit families $C = (C_n)$ form a non-uniform computation model because we are not concerned with how one comes up with circuit C_n for input length n. For a Turing machine to be able to simulate a circuit family it must also have free access to some information that depends on the length of the instance $n = |x|$ but not on the contents of the instance x. A *non-uniform Turing machine* is a Turing machine with two read-only input tapes. The first input tape contains the instance x, and the second input tape contains some helping information $h(|x|)$ that is identical for all inputs of length n. Because of the second input tape, the number of configurations of a non-uniform Turing machine that visits at most $s(n)$ tape cells is larger by a factor of $h(n)$ than the number for a normal Turing machine, namely $2^{\Theta(\log n + s(n) + \log h(n))}$. Frequently the second input tape is denoted as an oracle tape and the help as an oracle. The results of Section 14.2 can be generalized to the situation where we simulate non-uniform Turing machines with (non-uniform) circuits. The help $h(n)$ represents for C_n a constant portion of the input.

We will now show the following simulation results which go in the other direction from the results of Section 14.2:

- Small circuit families can be simulated by fast non-uniform Turing machines.
- Shallow circuit families can be simulated by non-uniform Turing machines with small space requirements.

The latter of these is the second support for the parallel computation hypothesis.

We will use the following notation for circuit families $C = (C_n)$:

- $s(n)$ for the size of C_n and $s^*(n)$ for $\max\{s(n), n\}$,
- $d(n)$ for the depth of C_n and $d^*(n)$ for $\max\{d(n), \lceil \log n \rceil\}$,
- f_n for the function computed by C_n,
- A_f for the decision problem corresponding to $f = (f_n)$.

Theorem 14.3.1. *The circuit family $C = (C_n)$ for a decision problem A_f can be simulated by a non-uniform Turing machine with two work tapes in time $O(s^*(n)^2)$ and space $O(s^*(n))$.*

Proof. For our help we let $h(n)$ be a description of the circuit C_n. This contains a list of all gates, which are represented as triples giving the operator, the first input, and the second input. For each input we note first the type (constant, input bit, or gate) and then its number. The length of this description is $O(s(n) \log s^*(n))$.

The Turing machine now processes the gates in their natural order. The first work tape is used to store the values of the previously evaluated gates. The second work tape will store a counter used to locate positions on the first work tape or the input tape. The help tape records where the input values for each gate is to be found. If the Turing machine knows the input values and the operator of a gate, then it can compute the output of this gate and append it to the list of previously known outputs. To evaluate the output of a gate, the Turing machine first retrieves the values of its inputs. For a constant this information is given directly on the help tape. Otherwise the index of the gate or input bit is copied from the help tape to the second tape. If the input is another gate, the head of the first work tape is brought to the left end of the tape. While we repeatedly subtract 1 from the counter stored on the second work tape until we reach the value 1, the tape head on the first tape is moved one position to the right each time. At the end of this procedure, the head on the first work tape is reading the information we are searching for. For an input bit we search analogously on the input tape. It is easy to see that for gate i or bit i, $O(i) = O(s^*(n))$ steps suffice. Since we must process $s(n)$ gates each with two inputs, the entire time is bounded by $O(s(n) \cdot s^*(n)) = O(s^*(n)^2)$. On the first work tape we never have more than $s(n)$ bits, and on the second work tape the number of bits is bounded by $s^*(n)$. □

If we want to obtain a Turing machine with one work tape, we could use the simulation result mentioned in Section 2.3 and obtain a time bound of $O(s^*(n)^4)$. But in this case it is possible to give a direct description of a Turing machine with a time bound of $O(s^*(n)^2 \log s^*(n))$, but we will omit these details. If the given circuit family is uniform, then for an input of length n we can first compute C_n and then apply the simulation we have described.

Theorem 14.3.2. *The circuit family $C = (C_n)$ for a decision problem A_f can be simulated by a non-uniform Turing machine in space $O(d^*(n))$.*

Proof. We no longer have enough space to store the results of all the gates. But this is only necessary if the results of some gates are used more than once. This is not the case if the underlying graph of the circuit is a tree. It is possible to "unfold" circuits in such a way that they become trees. To do this we go through the graph of the circuit in topological order. As we encounter a vertex with r immediate successors we replace it and all of its predecessors with r copies. This increases the size but not the depth of the circuit. Circuits for which all gates have at most one successor are called *formulas*. In Figure 14.3.1 we show the result of unfolding the circuit from Figure 14.1.1.

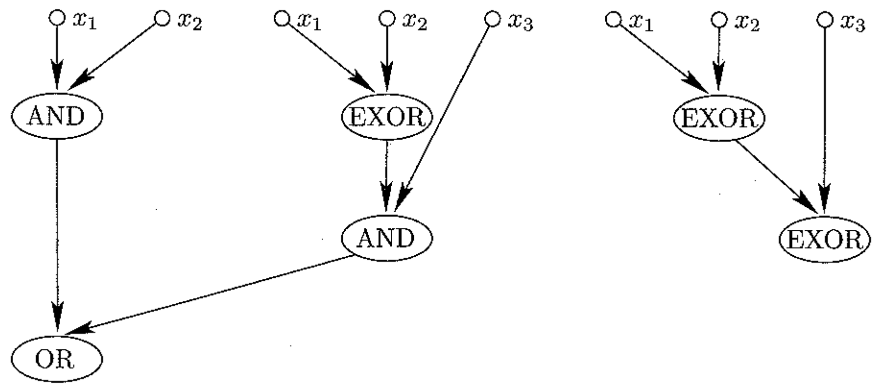

Fig. 14.3.1. The 3-bit adder as a formula.

As help for inputs of length n we use a formula F_n of depth $d(n)$ for f_n. Its description contains a list of all gates in the order of a post-order traversal. For a tree with one vertex this order consists of that vertex. Otherwise it consists of the results of a post-order traversal of the left subtree followed by that of the right subtree, followed by a description of the root. Gates will once again be described as triples of operation, left input, and right input. It is important that because we are using this order we will not need the numbers of the inputs that are gates. This will be seen in the proof of correctness. Formulas of depth $d(n)$ have at most $2^{d(n)} - 1$ gates, each of which can be described with $O(\log n)$ bits. So all together the length of the description is $O(2^{d(n)} \log n)$.

The Turing machine now processes the gates using post-order traversal. The work tape will be used to store the sequence of gate outputs that have not yet been used by their successor. Furthermore, as in the proof of Theorem 14.3.1 we use $O(\log n)$ space to locate values on the input tape. But how do we find values of the inputs to a gate? Because they are processed in the order of a post-order traversal, the left and right subtrees of a gate are processed immediately before that gate. The roots of these subtrees are the only gates whose values have not yet been used. So the values we need are at the right end of the list of gate outputs and the Turing machine works correctly.

Finally we show by induction on the depth d that there are never more than d outputs stored on the work tape. This implies that the space used is bounded by $O(\log n + d(n)) = O(d^*(n))$. The claim is clearly true when $d = 1$. Now consider a formula of depth $d > 1$. By the inductive hypothesis, $d - 1$

tape cells are sufficient to evaluate the left subformula. This result is stored in one cell, and along with the at most $d - 1$ tape cells needed to evaluate the right subformula, at most d tape cells are required. In the end only two tape cells are being used, and after evaluation of the root, only one tape cell is being used. □

If the given circuit family is uniform, then the information about a gate can be computed in space $O(\log 2^{d(n)}) = O(d(n))$. So in this case even a uniform Turing machine can get by with space $O(d^*(n))$.

There are close connections between circuit families and non-uniform Turing machines, and between Turing machines and uniform families of circuits. Circuit size is polynomially related to computation time, and circuit depth is polynomially related to space.

14.4 Branching Programs and Space Bounds

Now we want to introduce a non-uniform model of computation, the size of which characterizes the space used by non-uniform Turing machines asymptotically exactly. This model of computation has roots not only in complexity theory but also as a data structure for Boolean functions. For this reason there are two names commonly given to this model: *branching program* and *binary decision diagram* (abbreviated BDD).

A branching program works on n Boolean variables $x_1, \ldots, x_n$ and has only two types of elementary commands, which are represented as the vertices of a graph. A branching (or decision) vertex v is labeled with a variable x_i and has two out-going edges: one labeled 0, the other labeled 1. If v is reached in a computation, then the edge leaving v corresponding to the value of x_i is used to arrive at the next vertex. An output vertex w is labeled with a value $c \in \{0,1\}$ and has no out-going edges. If w is reached, then the computation is complete and the value c is given as output. A branching program is a directed acyclic graph consisting of branching vertices (also called internal vertices) and output vertices (also called sinks).

Each vertex v in a branching program realizes a Boolean function f_v in the following way. To compute $f_v(a)$ we start at vertex v and carry out the commands at each vertex until we reach a sink. For branching programs there are two obvious complexity measures. The *length* of a branching program is the length of the longest computation path in the branching program and is a measure of the worst-case time required to evaluate the function. The *size* of a branching program is the number of vertices, and the branching program complexity BP(f) of a Boolean function f is defined as the minimal size of a branching program that computes f. This is the complexity measure that we will be interested in here. Figure 14.4.1 contains a branching program the input vertices of which realize the two output bits for the addition of three bits. To make the diagram more readable we have included two 1-sinks.

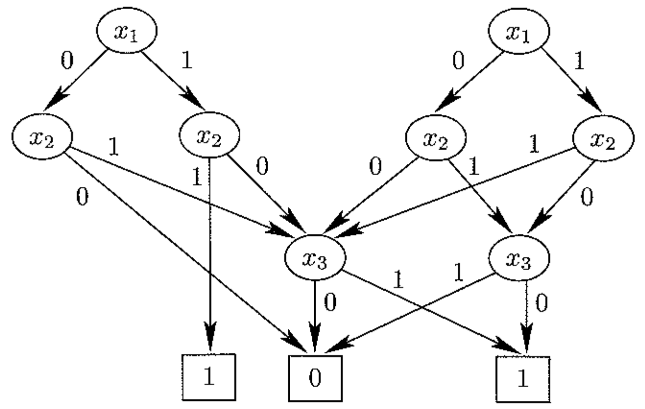

Fig. 14.4.1. A branching program for the addition of three bits.

So why is there a tight connection between the size of branching programs and the space required by non-uniform Turing machines? To evaluate f_v it is sufficient to remember the currently reached vertex. On the other hand, a branching program can directly simulate the configuration graph of a space-bounded Turing machine used in the proof of Theorem 14.2.2. We will formalize this in the theorem below, using $\mathrm{BP}^*(f_n)$ to represent the larger of $\mathrm{BP}(f_n)$ and n, and letting $s^*(n) = \max\{s(n), \lceil \log n \rceil\}$ just as before.

Theorem 14.4.1. *The decision problem A_f corresponding to $f = (f_n)$ can be solved by a non-uniform Turing machine in space $O(\log \mathrm{BP}^*(f_n))$.*

Proof. For help on inputs of length n we use a description of a branching program G_n of minimal size for f_n. This description includes a list of the vertices, where each vertex is described by its type (inner vertex or sink), its number, and its internal information. For a sink the latter consists of the value that is output by the sink, and for an inner vertex it consists of a triple including the index of the variable to be processed, the index of the 0-successor, and the index of the 1-successor. Furthermore, we will always let the vertex representing f_n have index 1. In this way each of the $\mathrm{BP}(f_n)$ vertices has a description of length $O(\log \mathrm{BP}^*(f_n))$. We use the work tape to remember the current vertex, so at the beginning of the computation it contains the number 1. If a sink is reached, then we make the correct decision and stop the computation. Otherwise we search for the value of the variable to be processed on the input tape. After that the new current vertex is known, namely the x_i-successor. We look for its information on the help tape and update the current vertex index on the work tape. □

Theorem 14.4.2. *An $s(n)$-space bounded Turing machine can be simulated by a branching program of size $2^{O(s^*(n))}$.*

Proof. We already know that the number of different configurations of the Turing machine on an input of length n is bounded by $2^{O(s^*(n))}$. The branching program G_n has a vertex for each of the configurations that is reachable from the start configuration. Accepting configurations are 1-sinks and rejecting configurations are 0-sinks. An inner vertex for configuration K is labeled with the variable x_i that is being read from the input tape in configuration K. The 0-child of this vertex is the configuration that is reached in one step from K if $x_i = 0$. The 1-child is defined analogously. Since we only consider Turing machines that halt on all inputs, the graph is acyclic and we have a branching program. The Boolean function describing the acceptance behavior of the Turing machine on inputs of length n is realized by the vertex labeled with the initial configuration. □

What changes if the given Turing machine is non-uniform and the help has length $h(n)$? The number of configurations and therefore the size of the simulating branching program grows by a factor of $h(n) \le 2^{\lceil \log h(n) \rceil}$. This has led to the convention of adding $\lceil \log h(n) \rceil$ to the space used by a non-uniform Turing machine. Or we could instead define $s^{**}(n) = \max\{s(n), \lceil \log n \rceil, \lceil \log h(n) \rceil\}$. The term $\lceil \log n \rceil$ has the same function for the input tape as the term $\lceil \log h(n) \rceil$ has for the help tape.

Corollary 14.4.3. *An $s(n)$-space bounded non-uniform Turing machine can be simulated by a branching program of size $2^{O(s^{**}(n))}$.* □

These results can be summarized as follows for the "normal" case that $s(n) \ge \log n$, $\mathrm{BP}(f_n) \ge n$, and $h(n)$ is polynomially bounded:

Space and the logarithm of the branching program size have the same order of magnitude.

For a language $L \in \mathsf{NP}$, $L \in \mathsf{P}$, or $L \in \mathsf{NTAPE}(\log n)$, we can try to show that $L \notin \mathsf{DTAPE}(\log n)$ by proving a superpolynomial lower bound for the branching program size of the function $f^L = (f_n^L)$. This is the most common line of attack for such results. To this point, such lower bounds for branching program size grow more slowly than quadratically (see Chapter 16).

14.5 Polynomial Circuits for Problems in BPP

We have already discussed several times that BPP is "not much larger" than P. It is possible that BPP = P, but this is still an open question. Now we want to offer some support for the claim that problems in BPP are not much more difficult than problems in P. For a decision problem $A \in \mathsf{BPP}$ the Boolean functions $f^A = (f_n^A)$ can be computed by circuits of polynomial size. If these circuits were uniform, then it would follow that BPP = P. But so far, only non-uniform circuits for f_n^A have been found. The trick is that for a BPP algorithm we can choose the error-probability to be so low that by the pigeonhole principle there must be an assignment for the random bits for which the BPP