KUNGL
TEKNISKA
HÖGSKOLAN

On Lower Bounds for Circuits and Selection

Staffan Ulfberg

Stockholm 1999

Doctoral Dissertation
Royal Institute of Technology
Department of Numerical Analysis and Computing Science

Akademisk avhandling som med tillstånd av Kungl Tekniska Högskolan framläges till offentlig granskning för avläggande av teknisk doktorsexamen fredagen den 17 december 1999 kl 10.00 i sal D2, Lindstedtsvägen 5, Kungl Tekniska Högskolan, Stockholm.

ISBN 91-7170-484-1

TRITA-NA-9911
ISSN 0348-2952
ISRN KTH/NA/R--99/11--SE

Högskoletryckeriet, KTH, Stockholm 1999

Abstract

This thesis presents results on lower bounds for circuit complexity, implications of some of these bounds for relativized complexity, and lower bounds for median selection.

Boolean functions may be computed using circuits consisting of AND, OR, and NOT gates. The size of a circuit is the number of gates it contains, and the size of the smallest circuit computing a function is a measure of the function's complexity. The depth of a circuit is the number of gates on the longest path from any input to the output gate. Monotone circuits only contain AND and OR gates, and bounded depth circuits have a specified maximum allowed depth.

Razborov invented the method of approximation as a way of proving lower bounds for the size of monotone circuits. The method involves approximating the outputs of the gates in a circuit with DNF formulas with certain restrictions. We show how the method can be made more symmetric by using both CNF and DNF formulas. As a consequence we no longer need the Sunflower lemma that has been essential for the method of approximation. The new approximation argument corresponds to Haken's recent method for proving lower bounds for monotone circuit complexity (counting bottlenecks) in a natural way. We demonstrate the method by providing lower bounds for the BMS problem introduced by Haken, for Andreev's polynomial problem, and for Clique; the exponential bounds obtained are the same as the previously best known for the respective problems.

We also introduce a new function based on combinatorial designs, which is only partially explicit, and prove the lower bound $2^{\tilde{O}(n^{1/3})}$ for this function. The new approximation argument is also extended to hold for monotone real circuits, which have gates that compute arbitrary real-valued monotone functions.

For bounded depth circuits, we show that there are functions computable by linear size boolean circuits of depth k that require super-polynomial size perceptrons of depth $k-1$, for $k < \log n/(6 \log\log n)$. A perceptron is a circuit where the output gate has been replaced by a threshold gate.

There is a strong correspondence between results on bounded depth circuits and results on relativized complexity. Using this correspondence, we show that our result on perceptrons implies the existence of an oracle A such that $\Sigma_k^{p,A} \not\subseteq \mathrm{PP}^{\Sigma_{k-2}^{p,A}}$; in particular, this oracle separates the levels in the $\mathrm{PP}^{\mathrm{PH}}$ hierarchy. We show a lower bound for another function, which makes it possible to strengthen the oracle separation to $\Delta_k^{p,A} \not\subseteq \mathrm{PP}^{\Sigma_{k-2}^{p,A}}$. This separation is almost tight, which follows from a relativization of Beigel's result that $\mathrm{P}^{\mathrm{PP}[\log]} = \mathrm{PP}$.

Turning to median selection in the comparison based model of computation, we present a reformulation of the $2n + o(n)$ lower bound of Bent and John for the number of comparisons needed for selecting the median of n elements. Our reformulation uses a weight function. Apart from giving a more intuitive proof for the lower bound, the new formulation opens up possibilities for improving it. We use the new formulation to show that any *pair-forming* median finding algorithm,

i.e., a median finding algorithm that starts by comparing $\lfloor n/2 \rfloor$ disjoint pairs of elements, must perform, in the worst case, at least $2.01n+o(n)$ comparisons. This provides strong evidence that selecting the median requires at least $cn + o(n)$ comparisons, for some $c > 2$. Dor and Zwick have been able to extend these ideas to obtain a $(2+\epsilon)n$ lower bound, for some tiny $\epsilon > 0$, on the number of comparisons performed, in the worst case, by any median selection algorithm.

Keywords: circuit complexity, computational complexity, monotone circuit, constant depth circuit, constant depth perceptron, oracle, relativized complexity, approximation method, comparison based model, median selection.

ISBN 91-7170-484-1 • TRITA-NA-9911 • ISSN 0348-2952 • ISRN KTH/NA/R--99/11--SE

Acknowledgments

Working in the theory group at Nada learning about theoretical computer science, doing research, preparing this thesis, and spending time on other distractions has been a genuine pleasure. I would like to thank everyone in the group, past and present, including guest researchers, for making it the place it is.

My supervisor has been Johan Håstad. It has been a privilege to work with him, and I want to thank him for his time, which he has generously shared to explain and discuss whatever subjects I needed help with, for his encouragement, and for sharing his ideas.

I would also especially like to mention Mikael Goldmann, who is always interested in discussing and helping out with all kinds of problems, and Christer Berg, who is the co-author of several articles that make up this thesis. Thanks also to Avi Wigderson at the Hebrew University of Jerusalem, who shared ideas used in part of this thesis, and to Uri Zwick at Tel Aviv University with whom I prepared the journal version of one of the articles in the thesis.

I have shared my office for a long time with my fellow students Mats Näslund, Anna Redz, and again, Christer. They have helped making the last five years a great time by engaging in interesting discussions, albeit not always research related. The same is also true for Gunnar Andersson, Lars Arvestad, Lars Engebretsen, Lars Ivansson, Öjvind Johansson, and Per Lindberger.

I am also indebted to Stefan Arnborg, Viggo Kann, Jens Lagergren, Karl Meinke, Alex Russell, and Rand Waltzman for giving courses, sharing knowledge of computer science, and providing a pleasant atmosphere.

I am thankful to my parents, Sture and Inger, for the support they have given. Sture also helped proofreading the non-technical parts of the thesis.

Finally, I wish to express my gratitude to Fredrik Almgren for taking the time to proofread draft versions of the thesis, and to Manne Börjel for expertise on aquariums.

Contents

Chapter 1

Introduction

1.1 An introduction to computational complexity

Suppose you are planning to set up a large aquarium, and want to have as many species of fish as possible represented. The problem is that quite a few of them feed on each other, and you want to select the species carefully so that this never happens.

One way to solve this problem is to draw a picture like the one in Figure 1.1. We make a point in the picture for each species, and connect a pair of points if the respective species are a threat to each other for some reason and therefore can not both be in the aquarium.

Figure 1.1. Some common species of aquarium fish, with lines indicating pairs of species that should be avoided in the same water.

In mathematical terms, Figure 1.1 is a *graph;* the points are called *vertices* and the lines interconnecting them *edges.* Finding the largest collection of species that can all be selected is now the same thing as finding the largest collection of vertices such that no pair is connected by an edge. This *computational problem*

is called finding the *maximum independent set* in the graph. It is not hard to verify, even by hand, that the largest possible independent set in Figure 1.1 contains five vertices (there are several possibilities of choosing an independent set of size five).

If we increase the size of the graph to a few thousand vertices, however, a number which does not seem terribly high for a computer, it turns out that no-one has been able to devise a method to solve the problem that is practical on today's computers. Although we have only increased the size of the problem by a factor of a hundred or so, the time to solve it has increased dramatically. Problems with this characteristic are called *computationally hard,* and it seems that finding the maximum independent set is a computationally hard problem. Notice that when formulated in a general mathematical setting, the same problem corresponds to many more real-world problems. For example, consider a number of radio transmitters, some of which interfere with each other. Finding the maximum number of transmitters that may be used at the same time also amounts to finding the maximum independent set in a graph.

There are many other problems that are generally believed to be computationally hard, such as factoring large integers into primes and computing discrete logarithms. Public key cryptography systems rely on the hardness of computing these and some other functions, which means that the existence of functions that are hard to compute is sometimes an advantage.

In contrast to computationally hard problems, consider tasks such as adding and multiplying numbers, searching for words in a dictionary, or counting the number of pairs of species in Figure 1.1 that can be in the same aquarium. For these tasks, there are efficient methods that scale very well with the size of the input data; such problems are called *computationally efficient.*

As computers become faster, the maximum practical size of input data increases for all of the problems mentioned above. For a simple problem like adding two numbers, a ten-fold increase in computing speed would allow ten times as many digits to be added in the same time as before. For a computationally hard problem, however, the same increase in computational power only affects the maximum data size marginally. This means that the computationally hard problems will remain hard for practical purposes for a very long time, even if computers continue to double their computational power every 18 months, which is probably an optimistic estimate.

For theoretical treatment, we are interested in the efficiency with which some particular function can be computed, given a detailed description of the computing device to be used. Functions are mappings of input data to output values; for example, the maximum independent set function maps input graphs to subsets of the vertices. We often restrict our attention to *boolean* functions, for which the input data always consists of a number of *binary digits,* or *bits* (a binary digit is a 0 or a 1); the output from a boolean function is only one bit. It is customary to denote the number of bits in the input data by n.

This restriction might seem like a very hard restriction, but in fact it is not, since input data for any other function may be represented as a sequence of bits.

For example, returning to the example of finding the maximum independent set in a graph, we might simply encode the number of vertices as a binary number, and following it by a 0 or 1 for each pair of vertices, indicating the absence or presence of an edge between them, respectively. For the output, we may use one bit for each vertex, indicating whether it is a member of the maximum independent set or not. We then use one function for each vertex, and treat them all individually. This might seem very inefficient, since each individual function probably has to compute the maximum independent set anyway, but, the added complexity is often negligible compared to the variations between different functions.

There are a wide variety of computing devices, some of them faster than others, and some more suited to a particular task than others. Thus, knowing the time required to compute some function on one device does not tell us the time required to compute the same function on all possible devices, which leads to the question of which one to analyze. Alan Turing introduced a mathematical model of a theoretical computer, called the *Turing machine*, which has become the standard model of computing machinery. The Turing machine captures the most important characteristics of computers, a fact that has not changed during the now over 50 years of rapid computer development. Since the Turing machine is only a model of a computer, we do not need to define its speed. Instead, we count the number of computing steps it requires to complete its program, and call this the running time of the Turing machine. The running time of the Turing machine can be roughly translated to execution time on any specific computer. The other computing resource of major importance is the amount of memory used during a computation. Also in this respect, the Turing machine corresponds very well with real life computers.

As well as writing programs that compute the values of our functions of interest, we might use *boolean circuits* to evaluate the function values. Boolean circuits consist of *boolean gates*, each of which computes one of the simple functions AND, OR, and NOT. The gates are interconnected such that one of them, the output gate, takes the value of the function when the input data is applied to designated input gates. A particular circuit is designed to work for a specific number of input bits; we therefore often refer to collections of circuits, one for each possible number of input bits.

When considering the amount of computing resources required by a circuit that computes a function, we are not concerned with the time required for the computation. Instead, the measures of interest are the number of gates used in the circuit, and the depth of the circuit (which is the distance from the output gate to the most far-away input; electronics people might argue that this is a measure of the time consumed by the circuit for computing the function).

There is a strong connection between the smallest possible circuit computing some function and the number of time steps required to compute the same function by a Turing machine. We may therefore analyze circuits to obtain results on Turing machines, and often do so since they seem combinatorially less complicated than Turing machines.

Unfortunately, however, for both Turing machines and circuits, there are no general methods to determine the most efficient way to compute a function. Even without actually asking for the most efficient method, it seems very hard to determine the amount of computing resources needed for a specific function. For example, multiplying two 1000 digit numbers intuitively seems harder than adding the same two numbers; intuition is right in so far that computers currently add numbers faster than they multiply them, but the difference in speed is probably smaller than intuition first suggests. From a theoretical point of view, we do not know whether there is a difference between the computational complexities of the two problems.

To determine the computational complexity of a function, we normally give *upper bounds* and *lower bounds* on the resources needed to compute it, for some specific model. The upper bound specifies an amount of resources that is sufficient for computation of the function. Notice that it does not in any way implicate that less resources than the stated bound would be insufficient. Most often, upper bounds are proved by demonstrating an *algorithm* (method), or circuit design to solve the particular function, together with analysis of the efficiency of the proposed solution. Thus, the upper bound proofs are constructive, and are useful in practice since they actually tell us how to compute the function in question.

A lower bound says that *any way* of computing the function using the computational model in question requires at least some amount of computing resources (e.g., steps for Turing machines and size for circuits). To prove a lower bound, we have to exclude *all possible* means of computing a function more efficiently than the stated bound. Intuitively, this seems harder than demonstrating that some proposed algorithm works, and in fact proving lower bounds almost always turns out to be very hard. We therefore find it logical that current results in lower bounds are weak. For many functions (such as, for example, addition and multiplication), the trivial lower bound n is easily proved, the only argument needed is that a program that does not even use n program steps can not read the entire input of the function.

While a lower bound for a function does not help to actually compute it, we need both a lower bound and an upper bound for a function if we want to know its computational complexity. Returning to the question of whether multiplication is computationally harder than addition, there are two possible answers to the question. Either the two problems are equally hard, which would be proved by demonstrating a way to multiply that is as fast as adding (it is well known that multiplication is not easier than addition). Or, multiplication is really harder than addition, which can only be proved by a lower bound for multiplication that is greater than some upper bound for addition.

There are mainly two practical uses for lower bounds. The first is that if we know that a lower bound for some function is achieved by some method computing it, further optimization of the computation is not possible. The other practical use is in cryptography. To prove that a cryptographic system is secure, a strong lower bound for the time needed to decipher an encrypted text

without having access to the decryption key is required. However, no-one has been able to prove computational security for any cryptographic system, and further development of lower bounds is therefore desired.

In the best of all worlds, the smallest upper bound and the largest lower bound for all functions would be the same. However, as indicated above, this is only the case for very few functions. For most functions, there is a large gap between the best (highest) lower bound and the best (lowest) upper bound.

Often we do not care about the exact computational complexity of a function, but instead categorize them into *complexity classes.* A complexity class is a collection of functions that can all be computed within some specified amount of computing resources. For example, the complexity class P contains all binary functions that can be computed on a Turing machine in a number of steps that is a polynomial function of the number of input bits (the number of time steps used is bounded by kn^c for some constants k and c). We may also characterize P in terms of circuits: a function is a member of the class P if there is a family of polynomial size circuits computing it. For technical reasons, we also need to require that for the circuit family, a circuit for a specified input size can be constructed efficiently by a Turing machine. The correspondence between running time of Turing machines and size of circuits establishes that both definitions of P result in the same class of functions.

Functions that are in the complexity class P are said to be efficiently computable. Of course, if the constants k and c need to be very large for some function, its computation requires a lot of time, but it turns out that, for practical functions, they are quite small most of the time so that the classification is a reasonable one.

There are many other complexity classes, a few of the most famous being NP and PSPACE. PSPACE is, in analogue to P, defined to contain all functions that can be computed by a Turing machine using an amount of memory that is bounded by a polynomial. PSPACE contains all functions in P as well, since a program that runs in polynomial time can not possibly use more than a polynomial amount of memory. The complexity class NP is characterized in Section 2.6.

Models of computation are not limited to Turing machines and circuits. In the *comparison based model,* the program computing some function is only allowed to examine the input data in a very specific way. Think about functions that take a list of names and telephone numbers (called input *elements*) as the input data and produce an alphabetically sorted list. In the comparison based model the program that computes this function is only allowed to make pairwise comparisons between elements of the input. That is, the program can not actually read the names and telephone numbers but can make decisions only on information about whether one name is lexicographically before or after some other name. The measure of complexity in this model is the number of comparisons needed to compute the function. Note that we do not make any restrictions on the way the program works, or its running time; we only count the number of comparisons it makes.

Many commonly used sorting, searching, and related algorithms work in this way, in that they never really need to understand anything about the input data other than doing pairwise comparisons between elements. This is very practical because the algorithms can easily be adopted to new types of input, and in practice, it also turns out that the number of comparisons needed is a reasonably good estimate of the running time of an algorithm. Determining the computational complexity of a problem in the comparison based model is also an intriguing combinatorial problem, and there are many hobby versions on this theme, like "How many weighings, using a common beam balance, do you need to find the fake coin, given 7 coins of which one is fake and differs in weight from the others?"

1.2 Thesis topics and background

The main topic of this thesis is lower bounds for functions in some variations of the circuit model, and for the comparison based model. A few of the lower bounds for circuits are further used for results about complexity classes.

1.2.1 Monotone circuits

As is mentioned in the introduction, there are no strong lower bounds for computation, and proving such lower bounds currently seems very hard. Although, in practice, it seems that some problems, like maximum independent set, are computationally very hard, there is no proof that they really are any harder than counting the number of vertices in the input graph, which is, of course, very easy to do.

Failing to prove lower bounds for Turing machines and circuits, we try something a little bit easier: we prove lower bounds for restricted models of computation. This means that we clip the wings of the Turing machine or the circuit model, and prove that this new, weaker, model can not compute the function efficiently. Since any proof of a lower bound for the general models automatically holds for the restricted models as well, restricting the models' computational power can only make proving lower bounds easier.

Of course, for practical purposes, lower bounds for such restricted models are of limited value. The main motivation for this line of research is to develop methods of proof, which can one day be extended to work for the general case.

One well studied restricted computational model is that of *monotone circuits*. The monotone computational model is easy to describe and is quite natural: a circuit that is monotone may contain only AND and OR gates, but no NOT gates are allowed. A direct consequence of this restriction is that monotone circuits only can compute *monotone functions*. A function is monotone if (and only if) changing one input bit from 0 to 1 never changes the function's value from 1 to 0. However, many important problems in complexity theory are monotone: one example is the function that is 0 if and only if the input graph contains

an independent set of size k, for some k. (For a graph with n vertices, the input consists of the $\binom{n}{2}$ possible edges in the graph, where 0 and 1 represent the absence and presence of an edge, respectively.) To see that this function is monotone, notice that adding an edge to a graph can never increase the size of the maximum independent set. Other examples of monotone graph problems are Clique and Hamiltonian path, which are both NP-complete. A clique in a graph is a vertex subset that is fully connected, and the Clique function is 1 if a graph with n vertices has a clique of size k. A Hamiltonian path in a graph is a path that visits every vertex exactly once; the Hamiltonian path problem is to determine whether such a path exists in a given graph.

Even for this restricted model, however, there were for a long time no stronger lower bounds than there were for general circuits; the best one was only $4n$ (Tiekenheinrich, 1984). When Razborov (1985b) invented the method of approximation, this changed rapidly. Razborov's new method allowed him to prove a super-polynomial lower bound as he showed that Clique requires monotone circuits of size $n^{\Omega(\log n)}$.

Shortly thereafter, Andreev (1985) applied Razborov's technique to a function (Andreev's polynomial problem—see Section 3.4.1) that is particularly well suited for the method of approximation. It is based on polynomials over finite fields, and for this function Andreev was able to prove an exponential lower bound. Later, the results of Razborov and Andreev were improved by Alon and Boppana (1987), and in particular, they were the first to prove an exponential lower bound for Clique. For a nice exposition of this result, see Section 4 in the survey by Boppana and Sipser (1990).

With these results on monotone circuits, the question of whether NOT gates may help in computing a monotone function arises. An answer to that came when Razborov proved a super-polynomial lower bound for the perfect matching function in bipartite graphs (Razborov, 1985a). This function takes a bipartite graph with vertex sets X and Y, $|X| = |Y|$, and edges $E \subseteq X \times Y$ as input, and checks if there is a subset of E such that each vertex is adjacent to exactly one edge in the subset. The perfect matching function is in the complexity class P, and it follows that general circuits are more powerful for computing monotone functions than are monotone circuits. For perfect matching, the best known lower bound is still the one obtained by Razborov (which is not exponential).

The method of approximation can roughly be described as follows. Assume we have a circuit C that computes the function f for which we want to prove a lower bound. We choose a set of functions $\mathcal{F}$, called *approximators*, and inductively find one of these functions to approximate the output of each gate in the circuit. The goal is to find a set $\mathcal{F}$ with the following properties. It should be possible to select approximators for all gates such that the approximation of each gate introduces an error on only a few input settings. (By introducing an error for an input x, we mean that the approximator for the gate is incorrect, while the gate outputs $f(x)$ and the approximators of its inputs are correct.) Also, all functions in $\mathcal{F}$ should differ from f for many input settings, i.e., no function in $\mathcal{F}$ should be close to the function computed by the output gate of C. If this

goal is accomplished, we get a lower bound for the circuit size since errors in the approximation of the output gate must have been introduced at some gate. The set of approximators used in Razborov's original proofs consists of DNF formulas (boolean formulas in disjunctive normal form, i.e., ORs of ANDs), where there is a limit on the length of each term, and on the number of terms.

Haken (1995) proposed a new method for proving lower bounds for the size of monotone circuits, that he named "Bottleneck counting." He demonstrated the method for a graph problem that resembles Clique (Broken mosquito screens—see Section 3.4.3). Instead of trying to approximate the outputs of the gates in the circuit, he defined a function μ which maps input graphs to gates of the circuit. To prove that the circuit contains many gates, he then showed that on one hand the total number of graphs mapped by μ is large, but on the other hand only few graphs are being mapped to each gate.

We use ideas from Haken's proof, but turn the proof itself into an approximation argument. This can be interpreted as if Haken's approach is in fact an approximation argument in disguise, but more importantly, it leads to elegant new proofs of previous results from the method of approximation. Out of the two possible forms of a depth 2 formula: DNF and CNF, Razborov uses the former. In the new proofs, we do not make this choice at all, in that we approximate each gate with two functions, one of each of the two possible forms. It turns out that with this modification, we do not need to use the sunflower lemma by Erdős and Rado (1960), which was previously a central part of the proofs. The sunflower lemma is traditionally used to reduce the size of the approximator functions while not reducing their approximation qualities too much. The fact that we do not need this lemma anymore is somewhat surprising.

While most of the known lower bounds obtained using the method of approximation benefit from being reformulated using the new set of approximators, it is not clear that all proofs can be reformulated this way. For example, we do not know how to prove Razborov's super-polynomial lower bound for perfect matching (which implies that monotone circuits are strictly less powerful than non-monotone) using the new set of approximators (and, most importantly, without the use of the sunflower lemma).

A monotone real circuit is a circuit where the gates have bounded fan-in and compute arbitrary real-valued monotone functions of their inputs, instead of being restricted to compute the functions AND or OR. Of course, a monotone boolean circuit can also be considered to be a monotone real circuit. Haken's method of Bottleneck counting was extended to hold for monotone real circuits by Haken and Cook (1996); this extension was the analog to an extension made by Pudlák (1997) to Razborov's method. Independent of Berg and Ulfberg, Wigderson (private communications) discovered that Haken's method can be turned into an approximation argument, and he also suggested how to extend the resulting approximation argument to hold for monotone real circuits. The details of the extension to monotone real circuits is the result of a joint work with Wigderson and are presented in Section 3.6. For this kind of circuits, Jukna

(1997) derived a general criterion for lower bounds that holds for both monotone boolean and monotone real circuits.

We want to investigate how far the method of approximation can take us in proving lower bounds for monotone functions, and show that the lower bound $2^{\Omega(n^{1/3}/\ln^{2/3} n)}$ can be obtained for a function based on balanced set systems. Balanced set systems are a relaxed form of combinatorial designs, and the function tests if the set of input bits being 1 is a superset of some block in the set system. Although the function is only partially explicit, the lower bound is the best known for any specific monotone function. We believe that this lower bound is essentially the best that can be proved using the method of approximation. Independently, Jukna (1999) has also showed how to prove lower bounds for functions based on combinatorial designs using his lower bound criterion.

1.2.2 Bounded depth perceptrons

Another way of restricting the computational power of circuits is to limit their maximum permitted depth. In contrast to monotone circuits, bounded depth circuits can compute all boolean functions: a circuit of depth 2 can be constructed by having a number of AND gates that compute all the *minterms* of a function; combining the minterms in a single OR gate that is also the output gate yields such a circuit. A minterm of a function f is a single conjunction of minimal length that, when one, forces f to 1.

Furst et al. (1984) was the first to prove that circuits having their depth bounded by a constant that compute parity (the sum of the input bits modulo 2) and majority (which is 1 if at least half of the input bits are 1) require size that is exponential in the number of input bits.

Restricting our attention to polynomial size circuits (which are the ones of interest for practical purposes), the complexity class AC_0 is defined as containing those functions that can be computed by arbitrary fan-in, constant depth circuits. The result of Furst et al. (1984) shows that parity is not in AC_0.

Sipser (1983) defined a family of functions that are computable by linear size circuits of depth k, and showed that they require super-polynomial size circuits of depth $k-1$. This shows that increasing the permitted depth by one for constant size circuits always adds new functions that can be computed in polynomial size. We say that there is a depth hierarchy within AC_0.

Sipser's result was later improved by Yao (1985) and Håstad (1987, 1989), showing that the functions defined by Sipser actually require exponential size circuits of depth $k-1$. A central part of Håstad's proof is the Håstad switching lemma, which is used to extend a separation of depth 2 circuits to all k by an induction argument.

A threshold gate is a special gate that outputs 1 if more than t of its inputs are 1, for some fixed t. Of course, a single threshold gate can compute majority. Circuits that have a single threshold gate at the top (the output gate), whose inputs are normal boolean circuits, is called a *perceptron*. The circuits that compute the inputs to the threshold gate are called the perceptron's sub-circuits.

Since threshold gates can compute majority, so can small size perceptrons, and thus, polynomial size perceptrons of depth k are more powerful than polynomial size circuits of the same depth.

For perceptrons, Green (1991) proved a lower bound for the size of constant depth perceptrons that compute parity. He was also interested in whether there are functions computable by polynomial size depth k perceptrons that can not be computed by polynomial size perceptrons of depth $k-1$ (the analog of the result by Sipser, Yao, and Håstad for ordinary boolean circuits), and was able to prove an exponential lower bound for depth 3 monotone perceptrons computing a function computable by linear size depth 4 perceptrons (Green, 1995). He proposed that a generalization of this result for non-monotone perceptrons could be used as a basis for induction to prove the separation for all k, using the Håstad switching lemma. In the monotone setting, the separation between depth k and depth $k-1$ perceptrons for all k follows from a stronger result by Håstad and Goldmann (1991) that separates boolean circuits of depth k from threshold circuits of depth $k-1$.

Perceptrons were studied in depth by Minsky and Papert (1988). Their interest in perceptrons is motivated by the fact that perceptrons are commonly used to model neurons in the human brain, and is an important part of artificial neural networks. They show the following "One-in-a-box" theorem:

Theorem 1.1 (One-in-a-box theorem)**.** *Let $A_1, \ldots, A_m$ be disjoint subsets of the set of binary input variables X and define the predicate*

$$\Psi(X) = \begin{cases} 1 & \text{if, } \forall A_i, \text{ at least one variable in } A_i \text{ is } 1, \\ 0 & \text{otherwise.} \end{cases}$$

If for all i, $|A_i| = 4m^2$, then at least one sub-circuit of any perceptron computing Ψ depends on at least m variables.

Minsky and Papert used the one-in-a-box theorem to, among other things, obtain a lower bound for the number of inputs to each sub-circuit of a perceptron that can determine whether a figure represented by discrete points is connected. See Figure 1.2 for an example.

This theorem is actually exactly the basis we need for an induction proof of the fact that depth k perceptrons of polynomial size can compute functions that can not be computed by depth $k-1$ perceptrons of polynomial size. We show that there are functions computable by linear size boolean circuits of depth k that require super-polynomial size perceptrons of depth $k-1$, for $k < \log n/(6 \log\log n)$, and exponential size perceptrons for constant k.

To use the one-in-a-box theorem as the basis for the induction, we need a modified, somewhat stronger, statement of Håstad's switching lemma. The stronger lemma actually follows if we look at the proof of the original switching lemma more carefully. This was noted by Boppana and Håstad (see Håstad (1987, Lemma 8.3)). Their note, however, was regarding another version of the

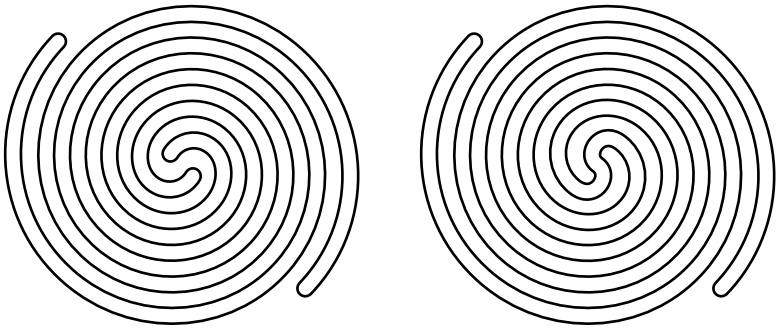

Figure 1.2. A perceptron in which each sub-circuit depends on too few input variables can not determine which of the two figures that is connected.

lemma, and was not explicitly proved; for this reason we include a sketch of the proof where we emphasize the "disjointness property" in Chapter 4.

1.2.3 Oracle separations of complexity classes

The computing power of a Turing machine can be enhanced by allowing it access to an *oracle.* What this means is that we give the Turing machine the capability of computing some specific function f using only one instruction. Technically, the oracle for f is described as a set A that contains the input settings for which f is 1. A Turing machine M with access to the oracle A is denoted M^A; we say that the computation is performed *relative* to the oracle A. For example, if A is an oracle for maximum independent set, which we believe is a hard function, the Turing machine M^A can compute this function in linear time (it is not constant because the Turing machine must still read its input and compile the oracle query). In a sense, we can say that the computational complexity for Turing machines with access to the oracle A, for a function g, is the additional computation needed to compute g if f can be computed at no cost.

As mentioned earlier, we categorize functions into complexity classes, like P, NP, and PSPACE. Having defined these classes, we can use an oracle to define new complexity classes. Given an oracle A, we denote by P^A the class of functions that can be computed in polynomial time in the input size, by a Turing machine with access to the oracle A.

A *complete* problem for a complexity class is a function that is among the computationally hardest in the class. More precisely, if f is a complete function for the complexity class $\mathcal{C}$, any function in $\mathcal{C}$ can be computed efficiently by a Turing machine with access to an oracle for f. Not all complexity classes have complete functions, but as one example, NP has.

We write P^{NP} to denote the complexity class that contains all functions that can be computed in polynomial time with access to an oracle for a function that is complete for NP. It is possible to define $\mathcal{D}^{\mathcal{C}}$ as long as $\mathcal{C}$ has complete problems.

We use the notation $\Sigma_0^p = \mathrm{NP}$ and $\Sigma_{k+1}^p = \mathrm{NP}^{\Sigma_k^p}$. This results in a chain of complexity classes, where the higher levels include any lower levels. It is unknown whether any or all of the levels are in fact different. This chain of complexity classes is called the polynomial time hierarchy.

It is currently not known if P and NP are indeed different (although it is widely believed that they are). Given an oracle A, we may consider the relation between P^A and NP^A. It has been proved that there exist oracles both for which the classes are equal, and for which they are different. Such results are called a *relativization* results.

There is a strong connection between lower bounds for boolean circuits (consisting of AND, OR, and NOT gates) and relativization results about the polynomial time hierarchy. This fact was first established by Furst et al. (1984).

The results by Yao (1985) and Håstad (1987, 1989), that there are functions that can be computed in polynomial size by depth k circuits, but require exponential size circuits of depth $k-1$, imply the existence of an oracle A that *separates* the levels in the polynomial time hierarchy, i.e., $\Sigma_k^{p,A} \not\subseteq \Sigma_{k-1}^{p,A}$.

There is a similar correspondence between the levels in $\mathrm{PP}^{\mathrm{PH}}$ and constant depth *perceptrons*. Green's motivation when trying to separate the computing power of depth k perceptrons and that of and depth $k-1$ perceptrons from each other (see the last section) was that a sufficiently strong such a separation would imply a separation of the $\mathrm{PP}^{\mathrm{PH}}$ hierarchy.

We show that our result on perceptrons in Chapter 4 solves the problem by Green in that it implies the existence of an oracle that separates the levels in the $\mathrm{PP}^{\mathrm{PH}}$ hierarchy.

For the lowest levels in the hierarchy, the fact that the one-in-a-box theorem implies that $\mathrm{NP}^{\mathrm{NP}} \not\subseteq \mathrm{PP}$ under an oracle was noted by Fu (1992). Beigel (1994) has strengthened this separation to obtain that $\mathrm{P}^{\mathrm{NP}} \not\subseteq \mathrm{PP}$ under an oracle, by showing a result for depth 2 perceptrons.

We improve on our separation of the levels in the $\mathrm{PP}^{\mathrm{PH}}$ hierarchy by using his result on perceptrons with bounded weights as a basis for the induction argument, and get an oracle A such that $\Delta_k^{p,A} \not\subseteq \mathrm{PP}^{\Sigma_{k-2}^{p,A}}$. ($\Delta_k^p$ is the complexity class $\mathrm{P}^{\Sigma_{k-1}^p}$.)

Beigel et al. (1991) showed that $\mathrm{P}^{\mathrm{NP[log]}} \subseteq \mathrm{PP}$, and later Beigel et al. (1995) proved the even stronger $\mathrm{P}^{\mathrm{PP[log]}} = \mathrm{PP}$. The notation A[log] denotes that at most $O(\log n)$ oracle queries may be made to the oracle A by the Turing machine. A relativization of these results shows that our result is almost tight.

1.2.4 Median selection

Sorting and selection problems may very well be the most well-studied of all computational problems. Sorting refers to the procedure of ordering the input data,

for example, an unordered telephone directory, into sorted order. By selection we mean to find the ith largest (or smallest) element among the input elements. Some common special cases of selection is finding the largest, the smallest, or the median element.

Comparison based algorithms for solving these problems work by performing pairwise comparisons between elements until enough information about their relative order is obtained. For sorting, the relative order of all elements needs to be known (notice, though, that this does not imply that an algorithm solving the problem needs to compare all pairs explicitly; if it is already known that $a > b$, and the algorithm compares b with c, the result that $b > c$ gives the relation $a > c$ for free). For selection, the algorithm needs to obtain, for the ith largest element, $i - 1$ relations to larger elements, and $n - i$ relations to smaller elements.

Sorting in a comparison based computational model is quite well understood. Any deterministic algorithm can be modeled by a decision tree in which all internal nodes represent a comparison between two elements; every leaf represents a result of the computation. Since there must be at least as many leaves in the decision tree as there are possible re-orderings of n elements, all algorithms that sort n elements use at least $\lceil \log n! \rceil \geq n \log n - n \log e + o(n) \approx n \log n - 1.44n + o(n)$ (log denotes the base 2 logarithm) comparisons in the worst case. The sorting method with best performance in the worst case, called *merge insertion* by Knuth (1973), is due to Ford and Johnson (1959). It sorts n elements using at most $n \log n - 1.33n + o(n)$ comparisons. Thus, the gap between the upper and lower bounds is very narrow in that the error in the second order term is bounded by $0.11n$.

A special case of selection, when $i = \lceil n/2 \rceil$, is finding the median among n elements, where there is a well-defined ordering between all pairs of elements. Although much effort has been put into finding the exact number of required comparisons, there is still an annoying gap between the best upper and lower bounds currently known.

Knowing how to sort, we could select the median by first sorting, and then selecting the middle-most element; it is quite evident that we could do better, but how much better? This question received a somewhat surprising answer when Blum et al. (1973) showed how to determine the median in linear time using at most $5.43n$ comparisons. This result was improved upon when Schönhage et al. (1976) presented an algorithm that uses only $3n + o(n)$ comparisons. Their main invention was the use of *factories* which mass-produce certain partial orders that can be easily merged with each other.

This remained the best algorithm for almost 20 years, until Dor and Zwick (1995) pushed down the number of comparisons a little bit further to $2.95n + o(n)$ by adding *green factories* that recycle debris from the merging process used in the algorithm of Schönhage et al. (1976).

The first non-trivial lower bound for the problem was also presented by Blum et al. (1973) using an adversary argument. Their $1.5n$ lower bound was subsequently improved to $1.75n + o(n)$ by Pratt and Yao (1973). Then Yap (1976),

and later Munro and Poblete (1982), improved it to $\frac{38}{21}n + O(1)$ and $\frac{79}{43}n + O(1)$, respectively. The proofs of these last two bounds are long and complicated.

Fussenegger and Gabow (1979) proved a $1.5n + o(n)$ lower bound for the median using a new proof technique. While this did not improve on the best lower bound, the proof was short and simple. Bent and John (1985) used the same basic ideas when they gave a short proof that improved the lower bound to $2n + o(n)$.

Since our methods are based on the proof by Bent and John, let us describe it in some detail. Given the decision tree of a comparison based algorithm, they invented a method to prune it that yields a collection of pruned trees. Then, lower bounds for the number of pruned trees and for their number of leaves are obtained. A final argument saying that the leaves of the pruned trees are almost disjoint then gives a lower bound for the size of the decision tree.

In Section 6.2 we reformulate the proof by Bent and John by assigning weights to each node in the decision tree. The weight of a node v corresponds to the total number of leaves in subtrees with root v in all pruned trees where v occurs in the proof by Bent and John. The weight of the root is approximately 2^{2n}; we show that every node v in the decision tree has a child whose weight is at least half the weight of v, and that the weights of all leaves are small.

When the proof is formulated in this way, it becomes more transparent, and one can more easily study individual comparisons, to rule out some as being bad from the algorithm's point of view.

For many problems, such as finding the maximal or the minimal element of an ordered set, and finding the maximal *and* minimal element of an ordered set, there are optimal algorithms that start by making $\lfloor n/2 \rfloor$ pairwise comparisons between singleton elements. We refer to algorithms that start in this way as being *pair-forming*. It has been discussed whether there are optimal pair-forming algorithms for all partial orders, and in particular this question was posed as an open problem by Aigner (1981). Some examples were then found by Chen (1993), showing that pair-forming algorithms are not always optimal.

It is interesting to note that the algorithms by Dor and Zwick (1995) and Schönhage et al. (1976) are both pair-forming. It is still an open problem whether there are optimal pair-forming algorithms for finding the median.

In Section 6.3 we use our new approach to prove that any pair-forming algorithm uses at least $2.01227n + o(n)$ comparisons to find the median.

For general comparison based median selection algorithms, Dor and Zwick (1996) have recently been able to extend the ideas described in Chapter 6 to obtain a $(2+\epsilon)n$ lower bound, for some tiny $\epsilon > 0$, on the number of comparisons performed, in the worst case.

Despite recent progress on both upper and lower bounds for median selection, however, the uncertainty in the coefficient of n for finding the median is still $0.95n$, which is larger than the gap of only $0.11n$ for sorting, even though the linear term is the second order term in the case of sorting.

1.3 Thesis overview

The thesis is organized as follows. Chapter 3 deals with lower bounds for monotone circuits, and introduces a symmetric version of Razborov's method of approximation (Razborov, 1985b). This formalism is used to prove lower bounds that match the best known for several well-known monotone functions, and also for a new function inspired by combinatorial designs for which the lower bound $2^{\tilde{O}(n^{1/3})}$ is obtained. We also show how the proofs can be extended to work for monotone real circuits. Most of this chapter is based on an article in Computational Complexity (Berg and Ulfberg, 1999), but also on discussions with Avi Wigderson. Approximately one half of the work leading to these results has been contributed by the author.

Chapter 4 provides lower bounds for perceptrons, which are a variation of boolean circuits where the top gate is replaced by a threshold gate. The lower bounds hold for bounded depth perceptrons. The structure of the proofs are the same as in proofs of lower bounds for bounded depth circuits (Håstad, 1987, 1989). The proofs in this section use a slightly enhanced version of the Håstad switching lemma, which is also contained in this section.

In Chapter 5 we turn to computational complexity and use the results obtained for perceptrons in Chapter 4 to prove oracle separations of complexity classes. Most notably, it is shown that there exists an oracle that separates the levels of the $\mathrm{PP}^{\mathrm{PH}}$ hierarchy, but we also prove a stronger version of this statement. The results in Chapters 4 and 5 were published in Journal of Computer and System Sciences (Berg and Ulfberg, 1998) and were contributed to in equal parts by both authors.

Chapter 6 deals with lower bounds for selecting the median in a comparison based computational model. The $2n$ lower bound (Bent and John, 1985) is proved using a new formalism, which enables us to more easily prove that any *pair-forming* algorithm requires $2.01227n + o(n)$ comparisons to select the median. A pair-forming algorithm starts out by making $\lfloor n/2 \rfloor$ pairwise comparisons between disjoint pairs of elements. This result was obtained in cooperation with Johan Håstad; it was contributed to in equal parts by both authors. Dor and Zwick independently discovered the same result which appears in a joint paper by the four authors (Dor et al., 2000).

Chapter 2

Models and Notation

2.1 Introduction

In this chapter we define some mathematical notation used throughout the thesis, and most of the relevant computational models.

2.2 General notational conventions

We use the usual definitions for O, Θ, Ω, o, and ω:

$$\begin{aligned}
O(f(n)) =& \{g \mid \exists N, c > 0 : \forall n \geq N : g(n) \leq cf(n)\},\\
\Omega(f(n)) =& \{g \mid \exists N, c > 0 : \forall n \geq N : g(n) \geq cf(n)\},\\
\Theta(f(n)) =& \{g \mid \exists N, c_1 > 0, c_2 > 0 : \forall n \geq N : c_1 f(n) \leq g(n) \leq c_2 f(n)\},\\
o(f(n)) =& \{g \mid \forall c > 0 : \exists N : \forall n \geq N : g(n) < cf(n)\},\\
\omega(f(n)) =& \{g \mid \forall c > 0 : \exists N : \forall n \geq N : g(n) > cf(n)\},
\end{aligned}$$

and also

$$\tilde{O}(f(n)) = \{g \mid \exists N, c : \forall n \geq N : g(n) \leq f(n) \log^c n\}.$$

The set of functions

$$\{h(g(n)) \mid g \in O(f(n))\}$$

is denoted by $h(O(f(n)))$.

We use log to denote base 2 logarithms, and ln to denote natural (base e) logarithms.

2.3 Boolean formulas

A boolean formula consists of boolean gates that are interconnected. A single gate computes one of the logical functions AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). An AND gate outputs the value 1 if (and only if) all its inputs are 1, and an OR gate outputs 1 if (and only if) at least one of its inputs is 1. A NOT gate has only one input and outputs 1 if the input is 0 and 1 otherwise. A formula's input consists of the binary variables $x_1, x_2, \ldots, x_n$, and the constants 0 and 1.

The *fan-in* of a gate is the number of inputs that connect to it. AND and OR gates may have arbitrary fan-in, but NOT gates always have fan-in 1. For example, in the formula in Figure 2.1, all AND and OR gates have fan-in 3.

In a formula, the output of each gate is connected to exactly one other gate, except for the output gate, whose output is not connected. Each input variable may be used as input for any number of gates. This implies that the structure of a formula is a tree. Input variables are at the leaves and the output gate at the root of the tree.

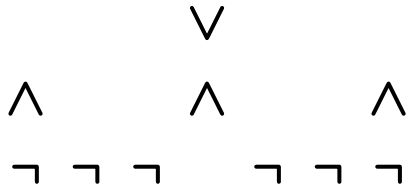

Figure 2.1. A formula that tests if exactly one of its inputs is 1; it computes the function $(x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3)$.

We may assume that a formula has all its NOT gates on the bottom-most level, since, working top-down using DeMorgan's laws, any other formula can be converted to this form without increasing its size. It is often convenient to consider $\neg x_1, \neg x_2, \ldots, \neg x_n$ input variables, to get rid of NOT gates completely.

We define the size of a formula F as the number of gates is contains, and denote this by size(F). The depth of a formula F is the length of the longest path from the root to a leaf of the formula, disregarding NOT gates; we denote this by depth(F). The formula in Figure 2.1 has size 10 and depth 2.

A depth 1 formula consisting of only one gate is called a *disjunction* if the gate is an OR gate, and a *conjunction* if the gate is an AND gate. Two other special forms of boolean formulas that are commonly used are CNF (conjunctive normal form) and DNF (disjunctive normal form) formulas, both of which have depth 2. A CNF formula has an AND gate in the root and OR gates at the bottom-most level; a DNF formula has an OR gate in the root and AND gates on the bottom-most level.

2.4 Boolean circuits

A boolean circuit is a generalization of a formula in that its structure is a directed acyclic graph rather than a tree. Edges point in the direction from the inputs towards the root of the tree. The *fan-out* of a gate in a circuit is the number of gates that connect to its output, or equivalently, the out-degree of the corresponding vertex. Fan-in is defined in the same way as for formulas.

As for boolean formulas, we define the size of a circuit C as the number of gates is contains, and denote this by size(C). The depth of a circuit C is the length of the longest path from the output node to an input of the circuit; we denote this by depth(C).

An observation that sometimes helps in analyzing circuits is that if any NOT gates appear inside a circuit, there is always an equivalent circuit of the same depth and at most twice the size that uses the additional inputs $\neg x_1, \neg x_2, \ldots, \neg x_n$, but does not have any NOT gates. Such a circuit can be constructed by working bottom-up and, for each gate g, adding a gate that computes the negation of g, using the negations of the inputs to g as inputs. Thus, a lower bound for circuits where the only negations occur as negated input variables also implies a lower bound for general circuits. It is standard practice to disregard the negations of input variables when counting the depth and size in this context.

Since a circuit only works for a specific number of input variables, we consider *families of circuits* that compute a function $f : \{0,1\}^* \mapsto \{0,1\}$. The family contains one circuit for each size of the input. Since formulas is a special case of circuits, this holds for formulas as well.

There are families of small size circuits that compute undecidable functions. To see this, consider an undecidable boolean function f and define $g(x) = f(|x|)$. The function g is also undecidable, since otherwise, we could use it to decide f. For each length of the input data, however, g is fixed, so there are families of constant size circuits that compute g.

This example shows that the size of the circuits in a circuit family computing a function does not, in general, correspond to the running time of a Turing machine that computes the same function. If we, however, require that each circuit in a family can be constructed efficiently by a Turing machine, it is not possible to compute the circuit family defined above. Families of circuits, where the circuit for each individual input size can be constructed efficiently by a Turing machine, are called *uniform.* Polynomial size uniform circuits can compute exactly those functions that are in the complexity class P. The lower bounds we prove, however, are all for non-uniform circuit families; any lower bound for non-uniform circuits of course also hold for uniform circuit families.

2.5 Variations of formulas and circuits

A monotone formula is a boolean formula with AND gates ($\wedge$) and OR gates ($\vee$), but without NOT ($\neg$) gates. Size and depth are defined as for general circuits and formulas. Monotone circuits are the circuit analog of monotone formulas.

Monotone circuits and formulas can only compute monotone functions, which is easily seen by considering how the change of an input variable from 0 to 1 propagates through the circuit. There are many interesting monotone functions, however, so this does not in itself render monotone circuits and formulas useless. Razborov (1985a) showed, however, that some monotone functions can be computed more efficiently by non-monotone circuits.

A monotone real circuit consists of analog gates that are monotone real functions of two real variables. The input and output of the circuit is still required to be 0 or 1. It has been shown by Pudlák (1997), Haken and Cook (1996), Jukna (1997), and in this thesis that, for many problems, the method of approximation yields almost the same lower bounds for monotone real circuits as for monotone boolean circuits.

A bounded depth boolean formula or circuit is a boolean formula, or a circuit, respectively, for which an upper bound on the maximum permitted depth has been imposed. Depth 2 formulas and circuits can trivially compute all boolean functions, using negated input variables as inputs, but it is usually easy to show that such circuits require exponential size. Super-polynomial size lower bounds for circuits of constant depth computing parity, majority, and some other functions have been obtained by, among others, Sipser (1983), Yao (1985), and Håstad (1987, 1989).

A perceptron is a circuit with a single threshold gate at the top, whose inputs are the outputs of boolean circuits with AND, OR, and NOT gates, called the perceptron's sub-circuits. A depth k perceptron has sub-circuits of depth $k-1$. A threshold gate is a special gate that outputs 1 if more than t of its inputs are 1, for some fixed t. Polynomial size perceptrons of constant depth can, of course, compute majority and are therefore more powerful than polynomial size constant depth circuits (see the last paragraph).

2.6 Turing machines

2.6.1 Deterministic Turing machines

The deterministic Turing machine is the standard way of modeling a computer. In contrast to formulas, circuits, and related models, Turing machines are not tailored for a specific input size. A Turing machine consists of three components:

- A *finite control,* that consists of a set of states Q and a *transition function* δ.

- An infinite *work tape*, used to store data during the computation. The tape is divided into *cells,* and each cell may contain one symbol of an alphabet Σ. The tape has a leftmost cell, but no rightmost cell. At the start of the computation, the input data is written on the work tape.

- A *tape head,* used to read and write data on the work tape, one cell at a time.

The state of the finite control, the contents of the work tape, and the position of the tape head define the machine's *configuration.*

The finite control has three special states: an *initial state,* an *accepting state,* and a *rejecting state.* At the start of the computation, the Turing machine is put in the initial state, with the input data present on the work tape. The tape head is initially positioned to the far left of the work tape. The computation ends when the Turing machine has reached either the accepting state or the rejecting state. We say that the machine *accepts* or *rejects*; the idea is that the machine should accept input data for which the function to be computed is 1, and reject other inputs.

More technically, Turing machines check whether the input is contained in a given set of strings on $\{0, 1\}$, called a *language.* We say that a Turing machine *decides* the language L if the machine reaches the accepting state for any input in L, and the rejecting state for any input not in L. Given a language L, the *characteristic function* of the language is 1 if and only if the input is a member of the language.

The alphabet Σ used on the working tape is arbitrary but finite. The most commonly used example is $\{0, 1, B\}$, where B is used to mark cells as being empty.

The transition function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{R, L, N\}$ decides what happens in one computation step. Depending on the current state of the finite control and the symbol in the cell at the tape head's current position, the transition function specifies the next state of the finite control, the symbol to be written to the tape cell at the position of the head, and in which direction to move the head (R means right, L means left, and N no move).

The time of a computation is measured as the number of steps required to reach either the accepting state or the rejecting state. The *space* used by the Turing machine during a computation is the number of cells of the tape that are used.

There are a number of variants of the model described in common use. For example, several work tapes, with one tape head for each tape, simplifies the description of many algorithms. It is easy to show that a Turing machine with only one work tape can simulate such machines efficiently.

The Turing machine model can easily be used to define the complexity class P: the complexity class P is the set of all languages that are decided by some Turing machine in time that is a polynomial function of the length of the input string.

2.6.2 Non-deterministic and alternating Turing machines

For a deterministic Turing machine, the transition function decides the next configuration of the machine uniquely from the current configuration. Non-deterministic Turing machines are different in that there are several possible next configurations given the present one.

The transition function is now defined as $\delta : Q \times \Sigma \to \mathcal{P}(Q \times \Sigma \times \{R, L, N\})$, where $\mathcal{P}(A)$ denotes the power set of A. This results in several possible computations of the non-deterministic Turing machine, each of which may either accept or reject. The running time and space requirements of a non-deterministic Turing machines are defined to be the maximum of the respective resources requires over all possible computations.

The non-deterministic Turing machine is said to accept an input x if there exists a computation that ends in the accepting state, and to reject the input x if all computations end in the rejecting state. We use non-deterministic Turing machines to define the complexity class NP: the complexity class NP is the set of all languages that are decided by some non-deterministic Turing machine in time that is a polynomial function of the length of the input string.

Given an input x, we may view the computation performed by a non-deterministic Turing machine M as follows. Every possible machine configuration is represented by a node in a *computation tree;* the root is the machine's initial configuration. A node is the parent to those nodes that represent the possible next configurations; thus the nodes representing halting configurations are the leaves of the tree. Label internal nodes by $\vee$, accepting leaves by 1, and rejecting leaves by 0. The result of the computation can now be determined by evaluating the resulting tree in a natural way, in fact, as if it was a boolean formula.

Notice that, since at most a constant number of configurations can be reached in one step from any configuration, we may assume that at most two next configurations are possible at any time without sacrificing more than a constant factor on the length of any path.

An *alternating Turing machine* is a non-deterministic Turing machine whose states are marked by $\wedge$, $\vee$, 0, or 1; the initial state is marked by $\vee$. States marked $\wedge$ and $\vee$ have at most two next configurations, and states marked 0 and 1 are the halting states, in which the machine rejects or accepts, respectively.

The computation tree for an alternating Turing machine is defined as for non-deterministic Turing machines, where internal nodes are labeled by $\wedge$ or $\vee$, depending on the marking of the corresponding state. Nodes that correspond to accepting states are labeled 1 and nodes that correspond to rejecting states are labeled 0.

We define a Σ_d^p machine as an alternating Turing machine where the maximum number of blocks of consecutive states labeled by $\wedge$ or $\vee$ on a path from the root to a leaf is d, and for which the maximum length of a path is bounded by a polynomial of the input length. We also define the complexity class Σ_d^p as containing those languages that are decided by some Σ_d^p machine. Notice that NP=Σ_1^p.

The union of Σ_d^p, for all d, is called the polynomial time hierarchy, and is denoted PH. It is unknown whether Σ_d^p are different for any (or all) d, but Håstad (1987, 1989) showed that there exists an oracle such that the corresponding classes relative to the oracle are indeed different.

2.6.3 Oracle Turing machines

We sometimes want to enhance the Turing machine model by giving the Turing machine access to an oracle. This means that we give the Turing machine the capability of computing some specific function f using only one computing step. The oracle for f is a set A that contains all inputs for which f is 1. A Turing machine M with access to the oracle A is denoted M^A; we say that the computation is performed relative to the oracle A.

An oracle Turing machine has a special work tape, called the *oracle query tape*. To perform an oracle query, the Turing machine writes a string on the oracle query tape, and then enters a special *oracle query state*. This instantaneously erases the contents of the oracle query tape and replaces it with the symbol 1 or 0, depending on whether the string on the query tape is present in the oracle set or not. We use oracle Turing machines in Chapter 5.

2.7 The comparison based model

The comparison based model is used mostly for solving sorting, searching, selection, and related problems. In this model, we do not measure the amount of time, memory, or other computing resources used during the computation, but count the number of pair-wise comparisons of input elements used. Also, comparing two of the input elements is the only operation permitted on the input data. (It is possible to write a sorting program that does not make a single comparison between input elements; an example of this is the radix sort algorithm. This is, however, not an example of a comparison based algorithm.)

We require that the input elements form an ordered set. That is, for all input elements a and b, we have that $a < b$, $a = b$, or $a > b$. Ordered sets also have the transitive property: if $a < b$ and $b < c$, it holds that $a < c$. Out of the three possible outcomes when comparing two input elements a and b, it is often convenient to ignore the possibility of the outcome $a = b$. When proving lower bounds, we may simply require that there are no such input elements. The bound obtained still holds for general input data, since a function can not become harder by imposing restrictions on its input.

Many commonly used sorting, searching, and selection algorithms are comparison based. One advantage with these algorithms is that they are completely independent of the type of the input data; only the comparison function needs to be modified to sort a new type of elements. It also turns out that the number of comparisons made in these algorithms corresponds very well with their running time.

It is also a combinatorially interesting problem to investigate how many comparisons are needed to solve a particular problem. For example, selecting the median from a set of 5 elements can be done using only 6 comparisons in the worst case, but it is not trivial to see how to do this. See Knuth (1973) for a step-by-step solution of this problem.

One of the most well-known lower bounds is probably the fact that sorting n elements requires $\lceil \log n! \rceil \geq n \log n - n \log e + o(n) \approx n \log n - 1.44n + o(n)$ comparisons in the worst case. Let us sketch a proof of this fact.

A comparison based algorithm can be described in terms of a decision tree. Each node of the tree represents an individual comparison; the root node represents the first comparison made. As the algorithm makes more comparisons, it collects information about the relative order of the input elements, and finally outputs the result when it has reached one of the leaves of the decision tree.

For sorting, all $n!$ ways of permuting the input elements need to be recognized by the algorithm, and thus, the depth of the decision tree is at least $\log n!$.

Chapter 3

Lower Bounds for Monotone Circuits

3.1 Introduction

Monotone circuits is the first restricted computational model that is investigated in this thesis. Monotone circuits do not use any form of negation, which means that monotone boolean circuits only contain AND and OR gates. In monotone real circuits, internal gates are allowed to compute any monotone real function of their inputs, although the output is still required to be 0 or 1. It is shown in Section 3.6 that the proofs in this chapter can be extended to work for monotone real circuits with only minor adjustments of parameter values.

Razborov's method of approximation was a major breakthrough for lower bound proofs when it first appeared (Razborov, 1985b). Razborov was able to show that any monotone circuit that computes the Clique function contains at least $n^{\Omega(\log n)}$ gates. After this breakthrough, Andreev (1985) proved that a function based on polynomials over finite fields requires an exponential number of gates, and Alon and Boppana (1987) improved the results to $2^{\Omega(n^{1/6})}$ for Clique and $2^{\Omega(n^{1/4})}$ for Andreev's problem. The new results all used the method of approximation in essentially the same form as Razborov did.

To apply the method of approximation on the function f we approximate the output of each gate in a circuit C that computes f by an approximator, which is selected from a set of functions $\mathcal{F}$. We then prove that the approximation of each gate in the circuit introduces an error on only a few input settings, and also, that there is no approximator that computes the output gate correctly for many input values. It follows that the circuit must be large since errors in the approximator for the output gate must emerge from errors introduced when approximating some gate in the circuit. The set of approximators used by Razborov is DNF formulas, where there is a limit on the length and number of terms.

Intuitively, this choice of approximators works because each gate performs a very simple calculation step and therefore, given approximators for its inputs, the gate computes a function close to some other function in $\mathcal{F}$. The function for which we prove the lower bound typically has many inputs consisting of a majority of 1s for which the output is 0, and many inputs consisting of a majority if 0s for which the output is 1.

Haken (1995) introduced a new method for proving lower bounds for the size of monotone circuits, and he called this method "Bottleneck counting." His method involves defining a function μ that maps a large subset of the inputs to gates in the circuit, and proving that not too many inputs are mapped to the same gate.

We use the techniques developed by Haken for his Bottleneck counting proof, but in an approximation setting. In fact, in a gate g, our approximators introduce errors for exactly those inputs that are mapped to g by Haken's function μ. Section 3.4.3 contains the proof by Haken using the approximation formalism.

The integration of Haken's techniques into the method of approximation leads to the use of two approximators for each gate in the circuit. That is, instead of choosing only DNF formulas as approximators, we approximate each gate with both a DNF and a CNF formula. Somewhat surprisingly, a central part of previous proofs using the method of approximation, the sunflower lemma (Erdős and Rado, 1960), is not needed anymore, and we think that this simplifies the proofs substantially.

Razborov (1989) showed that the method of approximation will not work very well to prove lower bounds for general circuits. He proved that, at least using the basic technique, no bound better than $\Omega(n^2)$ can be proved for such circuits.

The close relation between Bottleneck counting and the method of approximation unfortunately also leads to the conclusion that Bottleneck counting can not be used to prove strong lower bounds for general circuits.

Pudlák (1997) extended Razborov's method to hold for circuits consisting of bounded fan-in gates computing monotone real functions. The corresponding extension of Haken's method was made by Haken and Cook (1996). Jukna (1997) has also derived a very general criterion for lower bounds that holds for both monotone boolean and monotone real circuits.

Ideas from the bottleneck counting argument were independently also used by Amano and Maruoka (1996) to develop approximators similar to the ones presented in this chapter. They used the approximators to prove lower bounds for Clique and Andreev's polynomial problem. Simon and Tsai (1997) independently showed that Bottleneck counting is actually equivalent to the method of approximation by showing an equivalence between proofs in the two methods.

Wigderson independently discovered that Haken's method can be turned into an approximation argument, and he also noted that this holds for the extension made by Haken and Cook; the details of the extension to monotone real circuits is a joint work with Wigderson and is detailed in Section 3.6.

We also define a function based on balanced set systems, which are a relaxed form of combinatorial designs, and prove the lower bound $2^{\Omega(n^{1/3}/\ln^{2/3} n)}$ for this function. We also believe that this lower bound is essentially the best that can be proved using Razborov's method.

3.2 Preliminaries

The three first monotone functions for which we provide lower bounds (Andreev's polynomial problem, Clique, and Broken mosquito screens) are all graph problems, and we therefore consider the inputs as being graphs when describing the method. An input graph on n vertices is represented by $\binom{n}{2}$ variables $x_{i,j}$ whose value is 1 if the edge (i, j) exists in the graph.

The circuit which we want to prove is large is called C. The output of a gate e in C when the input x is applied to the circuit is denoted by $e(x)$. Let the output gate be e_o so that the circuit C computes the boolean function $e_o(x)$.

Every gate e in C is approximated by two functions, f_e^{D} and f_e^{C}, the *approximators* for the gate e. The approximator f_e^{D} has the form $C_1 \vee C_2 \vee \cdots \vee C_t$, where C_i is a conjunction containing less than c distinct literals. The approximator f_e^{C} has the form $D_1 \wedge D_2 \wedge \cdots \wedge D_s$, where D_i is a disjunction containing less than d distinct literals. (Notice that we put no explicit restriction on the numbers s and t.)

The following characteristics of the approximator functions f_e^{D} and f_e^{C} are essential to the proof.

1. The approximators $f_{e_o}^{\mathrm{D}}(x)$ and $f_{e_o}^{\mathrm{C}}(x)$ fail to correctly represent the output of C for many inputs x.

2. For every gate e, the number of inputs for which the approximators introduce errors (measured in a specific way) is small.

For every x for which the approximators for e_o fail, the error must have been introduced in at least one of the gates in C. We can therefore draw the conclusion that C contains many gates.

For convenience we consider the inputs to the circuit as being gates themselves, and for an input variable $x_{i,j}$ we simply define $f_{x_{i,j}}^{\mathrm{D}} = f_{x_{i,j}}^{\mathrm{C}} = x_{i,j}$. We also define empty disjunctions to have the value 0, and empty conjunctions to have the value 1.

Definition 3.1. For a gate e and an input x for which $e(x) = e_o(x)$, we say that the approximator f_e^{D} *fails* for x if $f_e^{\mathrm{D}}(x) \neq e(x)$, and that the approximator f_e^{C} *fails* for x if $f_e^{\mathrm{C}}(x) \neq e(x)$.

If either f_e^{D} or f_e^{C} fails for x, we say that the approximators for the gate e fail for the input x.

A central part of the proofs is counting the number of errors that are introduced in the gates of C. This is defined as follows.

Definition 3.2. An approximator, f_e^{D} or f_e^{C}, is said to *introduce an error* for the input x if it fails on the input x, but none of the approximators for the input gates to e fail for the input x.

If either f_e^{D} or f_e^{C} introduces an error for the input x, we say that the approximators for the gate e introduce an error for the input x.

We now describe how an AND gate e is approximated assuming that its input gates are already approximated by $f_{e_1}^{\mathrm{D}}, f_{e_2}^{\mathrm{D}}, \ldots, f_{e_m}^{\mathrm{D}}$ and $f_{e_1}^{\mathrm{C}}, f_{e_2}^{\mathrm{C}}, \ldots, f_{e_m}^{\mathrm{C}}$.

The approximator f_e^{C} is simply defined as

$$f_e^{\mathrm{C}} = \bigwedge_{i=1}^{m} f_{e_i}^{\mathrm{C}} = \bigwedge_{i=1}^{s} D_i,$$

in which all disjunctions still have length at most d. If none of $f_{e_1}^{\mathrm{C}}, f_{e_2}^{\mathrm{C}}, \ldots, f_{e_m}^{\mathrm{C}}$ fails for the input x, neither will f_e^{C}, so the approximator f_e^{C} introduces no errors.

The approximator f_e^{D} should be a disjunction of conjunctions; it is formed by converting f_e^{C} into the form

$$\bigvee_{i=1}^{t} C_i.$$

The standard way for doing this is to form a conjunction C_i for every possible way to pick one literal from each disjunction D_i in f_e^{C}. All conjunctions that have at least c distinct literals are then discarded, which means that long conjunctions are approximated by the constant 0.

In the proofs to follow we need to establish upper bounds for the number of errors introduced when forming f_e^{D} from f_e^{C}. To get as good bounds as possible, we have to be more careful when forming f_e^{D} (so that more conjunctions get shorter than c). We shortly describe the process we actually use in detail.

Returning to the definitions of f_e^{D} and f_e^{C}, we note that f_e^{D} is produced from f_e^{C}, which in turn is constructed from the approximators $f_{e_i}^{\mathrm{C}}$ of the input gates. Thus, when approximating an AND gate, we do not use the approximators $f_{e_i}^{\mathrm{D}}$ for the input gates.

The approximator functions for an OR gate e are formed analogously. We construct f_e^{D} from the approximators $f_{e_i}^{\mathrm{D}}$ of the input gates; this introduces no errors. The disjunction of conjunctions is then converted into a conjunction of disjunctions, and disjunctions with d or more distinct literals are discarded (they are in effect approximated by the constant 1). Notice that the way the approximators are constructed we have $f_e^{\mathrm{D}} \leq f_e^{\mathrm{C}}$ for all gates e.

We end this section by describing the details involved when converting f_e^{C} to f_e^{D} in an AND gate (this part is analogous for OR gates).

A set of conjunctions is formed that have the property that at least one of them is satisfied if and only if all disjunctions in f_e^{C} are satisfied. The approximator f_e^{D} is then formed from all resulting conjunctions shorter than c.

The rewriting process can be viewed as the construction of a tree: Each edge in the tree is labeled by a literal. For every node v in the tree we define a

corresponding conjunction that is formed by all literals on the path from the root to v.

At the root we create one labeled edge to a new child for each of the literals in the first disjunction. We say that we have expanded D_1 under the root.

Suppose w is a leaf that was created while expanding D_i, and that C is the conjunction corresponding to w. Then, $D_1, \ldots, D_i$ are all satisfied if C is satisfied. We now take care of D_{i+1}.

First consider the case that when C is satisfied, we know that for all accepting instances of the problem at least one of the literals in D_{i+1}, say $x_{u,v}$, must be satisfied as well. (This happens if D_{i+1} contains a literal in common with C, but it could also happen in some other situations if there are restrictions on the possible inputs.) We then make only one new child under w for the literal $x_{u,v}$ and skip the rest of D_{i+1}. This results in fewer leaves in the tree and therefore fewer conjunctions.

Otherwise, we expand the disjunction D_{i+1} under w.

The result is that if the conjunction corresponding to one of the new children is satisfied, so is D_{i+1}. Conversely, if all of $D_1, \ldots, D_{i+1}$ are satisfied, there is a node on depth $i+1$ in the tree whose corresponding conjunction is satisfied.

When there are no more leaves for which there are remaining disjunctions to expand, we are done and we get one conjunction for each leaf in the tree. Leaves whose corresponding conjunctions have at least c distinct literals are then removed, and this is the reason why f_e^{D} may make an error for some inputs.

Note that such inputs are always accepted by some conjunction (corresponding to a node in the tree) that has exactly c distinct literals, and bounding the number of such conjunctions is the reason for constructing the tree. By counting the number of inputs accepted by each conjunction, we can therefore get an upper bound for the number of errors introduced by f_e^{D}.

3.3 The proof method

Although they differ in details, the basic strategy for the proofs in the following sections is the same. In this section we therefore describe the strategy by giving generic versions of the theorems, and of the lemmas that are used in the proof of the theorems.

The generic theorem and lemmas contain various unspecified entities, such as numerical functions α, β, and γ. The outline can be turned into an actual proof of an actual theorem by specifying these entities (and checking their required properties).

Let $\mathrm{MGP}(n)$ be a monotone language on graphs with n input variables (which indicate the presence or absence of edges). For some function $h(n)$ we want to prove the following.

Generic Theorem. *The monotone circuit complexity for the language* $\mathrm{MGP}(n)$ *is at least* $h(n)$.

Generic proof. Suppose we are given a circuit C that correctly decides $\mathrm{MGP}(n)$. In order to prove that C must consist of at least $h(n)$ gates, we study two subsets of the possible input graphs: the *positive test graphs,* which is a subset of the graphs in $\mathrm{MGP}(n)$, and the *negative test graphs*, which is a subset of the graphs not in $\mathrm{MGP}(n)$. Let $\gamma_1(n)$ be the number of positive test graphs and $\gamma_0(n)$ the number of negative test graphs.

Instead of directly proving that a circuit that decides $\mathrm{MGP}(n)$ must be large, we prove that a circuit that separates the positive and negative test graphs from each other must be large; choosing the test graphs in a suitable way is of course essential.

We start by showing that the approximators for the output gate of C fail for most inputs.

Generic Lemma A. *At the output gate e_o, the approximators either fail for all negative test graphs or they fail for at least half of the positive test graphs.*

Generic proof. Assume that the approximators do not fail for all negative test graphs; otherwise there is nothing to prove. Hence, there exists a negative test graph b such that $f^{\mathrm{C}}_{e_o}(b) = 0$ which implies that $f^{\mathrm{C}}_{e_o}$ contains a disjunction D. Furthermore, for all positive test graphs g on which $f^{\mathrm{C}}_{e_o}$ is correct, we must have $D(g) = 1$. Since D contains less than d distinct literals, we can bound the fraction of positive test graphs for which $f^{\mathrm{C}}_{e_o} = 1$ by multiplying the fraction of positive test graphs for which any specific edge is present with d. A suitable choice of d proves the lemma. □

The next objective is to bound the number of inputs for which the approximators introduce errors at a single gate. Assume e is an AND gate. In this case f^{C}_e introduces no errors at all, and hence, we need only study f^{D}_e. Also, since $f^{\mathrm{D}}_e \leq f^{\mathrm{C}}_e$, no errors are ever introduced for negative test graphs in AND gates. The next two lemmas introduce the functions α and β as bounds for the number of errors introduced in a single AND and OR gate respectively.

Generic Lemma B. *At an AND gate e, the approximator f^{D}_e introduces an error for at most $\alpha(n, c, d)$ positive test graphs.*

Generic proof. To prove the lemma, we give an upper bound on the number of positive test graphs g such that $C(g) = 1$ (g is *accepted* by C), where C is any conjunction with c distinct literals corresponding to a node in the tree that represents the rewriting process.

For each node in the tree that has more than one child, the number of children is bounded by d, and descending to such a child always adds a distinct literal to the corresponding conjunction. There are therefore at most d^c conjunctions with exactly c distinct literals. It remains to find out the number of positive test graphs accepted by each such conjunction.

This is easily done if the edges in the positive test graphs are present independently from each other (as is the case in the lower bound for Andreev's polynomial problem in Section 3.4.1). If the problems are specified in terms of

vertices (e.g., the BMS problem in Section 3.4.3 and Clique in Section 3.4.2), we have to be a bit more careful since edges are not independent in the natural test graphs for these problems. Actually, in the proofs for these specific problems, we do not limit the number of distinct literals in conjunctions, but introduce another more natural measurement of their size.

The lemma should now follow from multiplying the number of conjunctions that have c distinct literals by the number of inputs accepted by each such conjunction. □

The following lemma is analogous and it is proved by bounding the number of errors introduced on negative test graphs when f_e^{C} is formed from f_e^{D}.

Generic Lemma C. *At an OR gate e, the approximator f_e^{C} introduces an error for at most $\beta(n, c, d)$ negative test graphs.*

To finish the proof of the generic theorem we need to consider the two cases from Generic Lemma A. First, if the approximators fail for all negative test graphs, Generic Lemma C implies that $\mathrm{size}(C) \geq \gamma_0(n)/\beta(n, c, d)$. Otherwise the approximators fail for at least half of the positive test graphs and then Generic Lemma B implies that $\mathrm{size}(C) \geq \gamma_1(n)/(2\alpha(n, c, d))$. So if $h(n)$ is less than both these values, we are done. □

3.4 New proofs of previous results

3.4.1 Andreev's polynomial problem

The best lower bound for Andreev's polynomial problem was proved by Alon and Boppana (1987). The function for which this bound was obtained was the same that Andreev (1985) used when he was the first to prove an exponential lower bound for a monotone problem. We here present our new version of this proof.

We are given a bipartite graph $G = (U, V, E)$ with vertex sets $U = \mathrm{GF}[q]$ and $V = \mathrm{GF}[q]$, where q is a prime power. The problem $\mathrm{POLY}(q, s)$ is to decide whether there exists a polynomial p over $\mathrm{GF}[q]$ of degree at most $s-1$ such that $\forall i \in U\colon (i, p(i)) \in E$.

Theorem 3.3 (Alon and Boppana, 1987)**.** *The monotone circuit complexity for* $\mathrm{POLY}(q, s)$ *is* $q^{\Omega(s)}$ *when* $s \leq \frac{1}{2}\sqrt{q/\ln q}$.

Proof. We first define the set of positive test graphs used in the proof. There are q^s different polynomials of degree at most $s-1$, each of which corresponds to a positive test graph in a natural way: the polynomial p defines a test graph with the edges $E = \{(i, p(i)) \mid i \in U\}$.

The negative test graphs are constructed randomly, by having each possible edge appear with probability $1 - \epsilon$ for $\epsilon = (2s \ln q)/q$. Notice that this construction may result in all possible bipartite graphs on $2q$ vertices; therefore we need the following lemma.

Lemma 3.4. *The probability that a negative test graph is in* POLY(q, s) *is at most* q^{-s}.

Proof. The probability that the q specific edges that correspond to a certain polynomial are present in a randomly chosen negative test graph is $(1-\epsilon)^q$. So the probability that the edges corresponding to at least one of the q^s possible polynomials are present in a random graph is at most

$$q^s(1-\epsilon)^q \leq q^s e^{-\epsilon q} = q^s q^{-2s} = q^{-s}.$$

□

We choose the parameters $c = s$ and $d = q^{2/3}/2$.

Lemma 3.5 (Specialization of Generic Lemma A)**.** *At the output gate* e_o*, we either have that the approximator* $f_{e_o}^{\mathrm{C}}$ *is identically 1, or that the approximators for* e_o *fail for at least half of the positive test graphs.*

Proof. If $f_{e_o}^{\mathrm{C}}$ is identically 1 we are in the first case. Otherwise, there is a graph b such that $f_{e_o}^{\mathrm{C}}(b) = 0$, and hence there is a disjunction D in $f_{e_o}^{\mathrm{C}}$. There are q^{s-1} positive test graphs that contain any specific edge since fixing an edge is equivalent to fixing the value of a polynomial in one point. Thus, there are at most $dq^{s-1} = q^{s-1/3}/2$ positive test graphs that contain at least one edge from D. Hence D (and therefore also $f_{e_o}^{\mathrm{C}}$) is 1 for at most a fraction $q^{s-1/3}/(2q^s) = q^{-1/3}/2$ of the positive test graphs. □

Lemma 3.6 (Specialization of Generic Lemma B)**.** *At an AND gate* e *the approximator* f_e^{D} *introduces an error for at most* d^c *positive test graphs.*

Proof. When rewriting f_e^{C} to f_e^{D}, we form at most d^c conjunctions that contain c distinct literals, and for the current function, each removed conjunction introduces an error on at most one positive test graph. This follows since $c = s$ and since a polynomial of degree $s-1$ is completely specified by s function points. □

Lemma 3.7 (Specialization of Generic Lemma C)**.** *At an OR gate* e*, the probability that* f_e^{C} *introduces an error for a random negative test graph is at most* $(c\epsilon)^d$.

Proof. We bound the probability of an error being introduced for a random negative test graph by multiplying the number of disjunctions with d distinct literals, c^d, by the probability that the d literals are 0 in one disjunction, which is ϵ^d. □

Notice that, in particular, the upper bound in the lemma holds for the number of errors introduced for negative test graphs that are not in the language POLY(q, s).

To prove the theorem we have to consider the two cases from Lemma 3.5. If the approximators fail for at least half of the positive test graphs we use

Lemma 3.6 which states that the number of errors introduced in a single AND gate is at most d^c to get

$$\text{size}(C) \geq \frac{q^s}{2d^c} = \frac{2^s q^{s/3}}{2} \in q^{\Omega(s)}.$$

If, on the other hand, the approximator $f^{\mathrm{C}}_{e_o}$ is identically 1, the approximators for e_o fail for a random negative test graph with probability $1 - q^{-s} \geq 1/2$ by Lemma 3.4. The probability of introducing an error in a single OR gate is at most $(c\epsilon)^d$; thus

$$\text{size}(C) \geq \frac{1}{2(c\epsilon)^d} = \frac{1}{2}\left(\frac{q}{2s^2 \ln q}\right)^{q^{2/3}/2} \geq \frac{1}{2} 2^{q^{2/3}/2} \in q^{\Omega(s)}.$$

□

3.4.2 Clique

In this section we prove that deciding whether a graph on n vertices contains any complete subgraph of size m (the problem CLIQUE(m, n)) requires monotone circuits of exponential size.

Theorem 3.8 (Alon and Boppana, 1987)**.** *The monotone circuit complexity for* CLIQUE(m, n) *is* $2^{\Omega(\sqrt{m})}$ *when* $m \leq n^{2/3}$.

Proof. Let the positive test graphs be all possible graphs on n vertices with a clique of size m and no other edges. This makes $\binom{n}{m}$ positive test graphs. We make one negative test graph for each possible coloring of the n vertices using $m - 1$ colors by connecting vertices of different colors by edges. Note that some colorings result in the same graph, but we treat them as different for counting purposes and get $(m-1)^n$ negative test graphs.

In this section we introduce two new measures of size for disjunctions and conjunctions. For each conjunction and disjunction, let a graph correspond to it in the natural way: for every literal, include the corresponding edge in the graph.

For disjunctions, we do not, as before, limit the number of literals they contain, but instead we define their size by n minus the number of connected components in their corresponding graphs, and require that their size is less than d.

For the conjunctions, we introduce a new notation of size that counts the number of vertices they *touch*. The number of vertices that a conjunction touches is the number of different vertices to which any of the edges connect. A conjunction is required to touch less than c vertices.

Choose $c = \lfloor\sqrt{m}\rfloor$, so that conjunctions touch less than $\lfloor\sqrt{m}\rfloor$ vertices. Implicitly, the number of literals in conjunctions is bounded by $c^2/2 \leq m/2$.

Let $d = \lfloor n/(8m)\rfloor$ so that the graphs corresponding to disjunctions have more than $n - \lfloor n/(8m)\rfloor$ connected components; this implies that they touch less than $2d \leq n/(4m)$ vertices.

Lemma 3.9 (Specialization of Generic Lemma A)**.** *At the output gate of C, e_o, the approximators either fail for all negative test graphs or they fail for at least half of the positive test graphs.*

Proof. Assume that there is a negative test graph b such that $f^{\mathrm{C}}_{e_o}(b) = 0$ (otherwise we are done already). Then there is a disjunction D in $f^{\mathrm{C}}_{e_o}$, and we show that this disjunction can be 1 for at most half of the positive test graphs.

A necessary condition for D to be 1 on a positive test graph g is that one of the vertices it touches is in the clique of g. Since any given vertex is part of the clique in a fraction m/n of the positive test graphs, a collection of at most $n/(8m)$ vertices has a vertex in common with less than half the positive test graphs. □

Lemma 3.10 (Specialization of Generic Lemma B)**.** *At an AND gate e, the approximator f^{D}_e can be selected such that it introduces an error for at most*

$$\binom{n-c}{m-c}\left(\frac{n}{2m}\right)^c$$

positive test graphs.

Proof. When building the tree, let w be the node that was created while expanding the disjunction D_i, and let C be the conjunction corresponding to w.

Consider the case that D_{i+1} contains a literal $x_{u,v}$ for which both endpoints are already touched by C. Then a positive test graph that satisfies C also satisfies D_{i+1}, so we only create one single child under w containing $x_{u,v}$.

Otherwise, D_{i+1} contains a mix of literals, for some none of the two endpoints are touched by C, and for some one endpoint is touched by C.

First consider the literals that have one endpoint that is already touched by C: they all have another endpoint that is not touched by C. For each such endpoint, arbitrarily select one of the literals that connect to the endpoint, and form one child for it. The reason that we may disregard some literals from consideration is that two conjunctions that touch the same vertices will accept the same positive test graphs. The total number of children created this way is less than $2d$ since D_{i+1} touches at most $2d$ vertices.

For the literals that connect to two vertices that are not touched by C, we first select one of their endpoints in some arbitrary way and create a child for it; however, we do not label the edges to these children. At most $2d$ such children are created. Then, we create less than $2d$ children under each of these children for the other endpoint, and the edges down to these children are labeled by literals.

The way we have constructed the tree implies that no node has more than $4d = n/(2m)$ children, and when descending to a child from a node with more than one child, the number of vertices touched by the corresponding conjunction increases by 1. Therefore, there are at most $(n/(2m))^c$ conjunctions touching c vertices.

The number of positive test graphs accepted by a single conjunction touching c vertices is the number of ways to choose the remaining $m - c$ vertices for the clique out of the total remaining $n - c$ vertices, which is $\binom{n-c}{m-c}$.

Finally, the lemma follows from multiplying the bound for the number of conjunctions touching c vertices by the maximum number of inputs they may accept. □

Lemma 3.11 (Specialization of Generic Lemma C)**.** *At an OR gate e, it is possible to select f_e^{C} such that it introduces an error for at most $(m/2)^d(m-1)^{n-d}$ negative test graphs.*

Proof. We start by bounding the number of disjunctions corresponding to nodes in the tree, whose corresponding graphs contain $n - d$ connected components.

When building the tree, let w be the node that was created while expanding the conjunction C_i, and let D be the disjunction corresponding to w.

Consider the case that C_{i+1} contains a literal $x_{u,v}$ for which both endpoints are in the same connected component in the graph corresponding to D. Then a negative test graph that falsifies D also falsifies C_{i+1}, since u and v must have the same color. It is therefore enough to create only one single child under w and label the edge $x_{u,v}$.

Otherwise, w gets at most $m/2$ children, since each conjunction contains at most $m/2$ literals. In this case the number of connected components decreases by one for the children, so we will make d such expansions before we reach a node for which the graph for the corresponding disjunction contains $n - d$ connected components. In all, there can hence be at most $(m/2)^d$ such disjunctions.

It remains to count the number of negative test graphs rejected by a single disjunction whose corresponding graph H contains $n-d$ connected components. The negative test graphs that are rejected by such a disjunction are those that use only one color within each connected component, and their number is $(m-1)^{n-d}$.

Multiplying the number of disjunctions whose corresponding graphs contain $n - d$ connected components by the number of negative test graphs rejected by each yields the bound in the lemma. □

We now determine the lower bound for the circuit size. There are two cases to consider: either $f_{e_o}^{\mathrm{D}}$ fails on at least half the positive test graphs so that

$$\begin{aligned}
\operatorname{size}(C) &\geq \frac{\binom{n}{m}}{2\binom{n-c}{m-c}\left(\frac{n}{2m}\right)^c} \\
&= \frac{1}{2}\frac{n!\,(m-c)!}{(n-c)!\,m!}\frac{(2m)^c}{n^c} \\
&\geq \frac{1}{2}2^c\frac{(n-c)^c}{m^c}\frac{m^c}{n^c} \\
&\in 2^{\Omega(\sqrt{m})},
\end{aligned}$$

or the approximators fail for all negative test graphs so that

$$\begin{aligned}\text{size}(C) &\geq \frac{(m-1)^n}{(m-1)^{n-d}(m/2)^d} \\ &= 2^d \left(\frac{m-1}{m}\right)^d \\ &\in 2^{\Omega(n/m)}.\end{aligned}$$

Since $2^{\Omega(\sqrt{m})} \subseteq 2^{\Omega(n/m)}$ for $m \leq n^{2/3}$, the theorem follows. □

3.4.3 Broken mosquito screens

The Broken mosquito screens (BMS) problem was introduced by Haken (1995) to illustrate how to count bottlenecks to show that monotone P $\neq$ NP. In this section we show that the same result can be obtained using our approximator formalism.

Haken defines the BMS problem for graphs with m^2-2 vertices as the problem of distinguishing *good* and *bad* graphs from each other. Graphs containing $m-1$ cliques of size m and one clique of size $m-2$ (but no other edges) are good; graphs containing $m-1$ independent sets of size m and one independent set of size $m-2$ but all other edges are bad. Hence, by taking the dual of a good graph (i.e., inverting all the input bits) we get a bad graph. Notice that not all graphs are either good or bad, but the definition can be extended to include all graphs. Haken shows that a monotone circuit that distinguishes good graphs from bad must be large.

We use the following extended definition of the BMS problem.

Definition 3.12. Instances of BMS(m) are graphs with m^2-2 vertices ($m > 2$). A graph is in the language BMS(m) if there exists a partition of the vertices into $m-1$ sets of size m and one of size $m-2$, so that each of these subsets forms a clique.

Using this definition, it is easy to see that the BMS problem is monotone and in NP.

Theorem 3.13 (Haken, 1995)**.** *The monotone circuit complexity for the language* BMS(m) *is* $2^{\Omega(\sqrt{m})}$.

Proof. We let the set of positive test graphs G for the BMS problem be all graphs with m^2-2 vertices that contain $m-1$ cliques of size m and one clique of size $m-2$, but no other edges. The set of negative test graphs B is all graphs on m^2-2 vertices that contain $m-1$ independent sets of size m and one independent set of size $m-2$, but where all other edges are present. Our positive and negative test graphs correspond to Haken's good and bad graphs, respectively.

Clearly, all positive test graphs are in the BMS language. Negative test graphs are not in the BMS language since they contain only $m-2$ cliques of size m.

Next we count the number of test graphs. Counting the number of positive test graphs is the same as counting the number of ways that a set of $m^2 - 2$ elements can be partitioned into $m - 1$ sets containing m elements and one set containing $m - 2$ elements. We get

$$|G| = \frac{(m^2-2)!}{(m!)^{(m-1)}(m-2)!(m-1)!}, \tag{3.1}$$

and by duality $|B| = |G|$.

For each conjunction and disjunction, let, as in the previous section, a graph correspond to it in the natural way: for every literal, include the corresponding edge in the graph.

Suppose we want to determine whether a positive test graph satisfies a conjunction C. This is the case when the graph corresponding to C is a subgraph of the positive test graph. Therefore, we only have to know how the vertices of the graph corresponding to C are divided into connected components to determine what positive test graphs are accepted by C.

As in the proof of the lower bound for Clique, we define the size of disjunctions by $m^2 - 2$ minus the number of connected components in their corresponding graphs, and require that their size is less than d. In this section, we analogously define the size of conjunctions analogously, as $m^2 - 2$ minus the number of connected components in their corresponding graphs.

We choose $c = d = \lfloor\sqrt{m}\rfloor$ (since $c = d$ we only use c), i.e., the graphs corresponding to the conjunctions and disjunctions in the approximators have more than $m^2 - 2 - \lfloor\sqrt{m}\rfloor$ connected components. Note that, implicitly, the number of literals in conjunctions and disjunctions is bounded by $c^2/2 \leq m/2$.

Lemma 3.14 (Specialization of Generic Lemma A)**.** *At the output gate of C, e_o, the approximators either fail for all negative test graphs, or they fail for at least half of the positive test graphs.*

Proof. If the approximator $f^{\mathrm{C}}_{e_o}$ for the output gate fails for all the graphs in B we are in the first case. Otherwise, let $b \in B$ be a graph for which the approximator does not fail at e_o.

Since $f^{\mathrm{C}}_{e_o}(b) = 0$, there is a disjunction D in $f^{\mathrm{C}}_{e_o}$, and as noted above, this disjunction contains less than $c^2/2 \leq m/2$ literals.

The fraction of graphs in G that contain a specific one of these literals is less than $1/m$, which can be seen as follows.

A graph in G has $(m-1)\binom{m}{2} + \binom{m-2}{2}$ edges out of the possible $\binom{m^2-2}{2}$ ones. Since all edges are equally likely, this means that a specific edge appears in a fraction

$$\frac{(m-1)\binom{m}{2} + \binom{m-2}{2}}{\binom{m^2-2}{2}} < \frac{1}{m}$$

of the graphs in G.

Thus, the fraction of G that contains at least one of the $m/2$ literals in D is at most $1/2$. So for at least half of all $g \in G$, we have $D(g) = 0$ and therefore also $f^{\mathrm{C}}_{e_o}(g) = 0$. □

Next, we establish an upper bound for the number of test graphs for which the approximators introduce an error at a single gate.

Lemma 3.15 (Specialization of Generic Lemma B)**.** *At an AND gate e, the approximator f^{D}_e can be selected such that it introduces an error for at most*

$$\frac{m^{4c}(m^2-2-2c)!}{2^c(m!)^{(m-1)}(m-2)!(m-1)!} \tag{3.2}$$

positive test graphs.

Proof. We introduce errors on positive test graphs by removing conjunctions whose corresponding graphs contain at most m^2-2-c connected components. When counting the number of positive test graphs for which an error is introduced, we consider different orderings of the vertices within the partitions and the order of the partitions as different graphs, and in the end we divide by the same denominator as in (3.1).

When building the tree, let w be the node that was created while expanding the disjunction D_i, and let C be the conjunction corresponding to w.

Consider the case that D_{i+1} contains a literal $x_{u,v}$ for which both endpoints are in the same connected component in the graph corresponding to C. Then, we create only one single child under w and label the edge $x_{u,v}$. In this case, we know that $x_{u,v}$ is satisfied if C is, so dropping all other children does not introduce errors for any positive test graphs. Clearly, ignoring to create some children never introduces errors for negative test graphs.

Otherwise, w gets less than $c^2/2 \leq m/2$ children, since each disjunction contains less than $c^2/2$ literals. In this case the number of connected components decreases by one for the children, so we will make c such expansions before we reach a node where the graph for the corresponding conjunction contains m^2-2-c connected components. Therefore, there are at most $(c^2/2)^c \leq (m/2)^c$ such conjunctions.

We now count the number of inputs accepted by a single conjunction whose corresponding graph H contains m^2-2-c connected components. Let a denote the number of vertices in H that are part of connected components with more than one vertex. It follows that the number of isolated vertices in H is m^2-2-a, and that the number of connected components in H with more than one vertex is $a-c$.

To get an upper bound for the number of positive test graphs that are compatible with H we count as follows. Each connected component in H with more than one vertex must go into one of the m partitions, which makes at most m^{a-c} choices. Then, each vertex in those connected components can be placed in at most m ways each within the partition. Lastly, the remaining m^2-2-a vertices

can be placed freely in $(m^2 - 2 - a)!$ ways. To sum up, the total number of choices is at most

$$m^{a-c}m^a(m^2 - 2 - a)! = m^{2a-c}(m^2 - 2 - a)!,$$

which is increasing with a. Since the graph H contains $a - c$ connected components with more than one vertex, the total number a of vertices in such components is at most $2c$. This follows since the number of connected components with more than one vertex is at most c. So an upper bound for the number of choices is

$$m^{3c}(m^2 - 2 - 2c)!.$$

Multiplying the maximum number of conjunctions for which the corresponding graphs contain $m^2 - 2 - c$ connected components by the maximum number of inputs accepted by each, and finally dividing with the same denominator as in (3.1) yields the lemma. □

The proof of the lemma above differs slightly from the corresponding proof by Haken. The reason for this is that we make a construction of the approximator for an AND gate from the approximators for the input gates, whereas Haken only proves the existence of a set of short conjunctions that describe the gate well enough.

Because of the duality of the test graphs and since $c = d$, the following lemma can be proved analogously to the previous one.

Lemma 3.16 (Specialization of Generic Lemma C)**.** *At an OR gate e, the number negative test graphs for which the approximator f_e^C introduces an error is bounded by (3.2).*

We now consider the two cases from Lemma 3.14. Either the approximators for the output gate e_o fail for at least half the positive test graphs, or they fail for all negative test graphs. Since $|G| = |B|$, and since we have the bound (3.2) for the maximum number of errors introduced in a single gate, the minimum size of C is

$$\frac{2^c(m^2 - 2)!}{2m^{4c}(m^2 - 2 - 2c)!}.$$

If we expand this expression we get

$$\begin{aligned}
\text{size}(C) &\geq 2^c \frac{(m^2 - 2) \cdots (m^2 - 2 - 2c + 1)}{2m^{4c}} \\
&\geq \frac{2^c}{2} \frac{(m^2 - 2 - 2c + 1)^{2c}}{(m^2)^{2c}} \\
&= \frac{2^{\lfloor\sqrt{m}\rfloor}}{2} \left(\frac{m^2 - 2\lfloor\sqrt{m}\rfloor - 1}{m^2} \right)^{2\lfloor\sqrt{m}\rfloor} \\
&\in 2^{\Omega(\sqrt{m})}.
\end{aligned}$$

□

3.5 Functions based on balanced set systems

In this section we define a function based on *balanced set systems*, and prove a lower bound for the size of circuits computing this function. In fact, we provide the best known monotone lower bound for any reasonably explicit monotone function in that the lower bound obtained is $2^{\tilde{O}(n^{1/3})}$.

3.5.1 Definitions

Definition 3.17. A (t, λ) balanced set system on X is a set $\mathcal{B}$ of subsets of X, such that for all $S \subset X$ of size t, S is a subset of at most λ sets in $\mathcal{B}$. The members of B are referred to as blocks.

Let X be the set of input variables, and define a monotone function from a (t, λ) balanced set system on X by having one minterm for each block in the balanced set system.

Denote $|X|$ by n. We want to construct a (t, λ) balanced set system where all blocks have size q and $|\mathcal{B}| = b$, for $q = n^{2/3}$, $t = (n/\ln^2 n)^{1/3}$, $\lambda = 3\ln n/(\ln\ln n - \ln 8)$, and $b = (n/\ln n)^{t/3}$ (we assume $n > e^8$ so that λ is positive). It is not clear that this is possible, but we show that just selecting the blocks randomly works.

Lemma 3.18. *For a set of variables X of size n, there are b subsets (blocks) of X of size q, such that each t-subset of X is included in at most λ blocks, for b, q, t, λ chosen as in the previous paragraph.*

Proof. We randomly choose b subsets of q elements from X and show that the probability P that any subset of t literals is a subset of more than λ blocks is less than one. We consider the probability that $\lambda+1$ blocks contain a subset of t elements from X simultaneously, and multiply by the number of possible choices for the subset of size t, and by the number of $(\lambda+1)$-subsets of blocks:

$$\begin{aligned}
P &\le \left(\binom{n-t}{q-t}\Big/\binom{n}{q}\right)^{\lambda+1}\binom{n}{t}\binom{b}{\lambda+1} \\
&< \left(\frac{(n-t)!q!}{n!(q-t)!}\right)^{\lambda+1} n^t\, b^{\lambda+1} \\
&< \left(\frac{2q}{n}\right)^{t(\lambda+1)} n^t \left(\frac{n}{\ln n}\right)^{t(\lambda+1)/3} \\
&= \left(\frac{8}{n}\right)^{t(\lambda+1)/3} n^t \left(\frac{n}{\ln n}\right)^{t(\lambda+1)/3} \\
&= \left(\frac{8n^{3/(\lambda+1)}}{\ln n}\right)^{t(\lambda+1)/3}.
\end{aligned}$$

It suffices to show that the mantissa in this expression is less than one:

$$\frac{8n^{3/(\lambda+1)}}{\ln n} < \frac{8n^{3/\lambda}}{\ln n} = \frac{8n^{\frac{(\ln\ln n - \ln 8)}{\ln n}}}{\ln n} = 1.$$

□

Our balanced set systems are very similar to combinatorial designs, and in fact a combinatorial design is a special case of a balanced set system. Designs are, however, known to exist only for very few parameter choices, and we have therefore relaxed the requirements.

3.5.2 De-randomization

The construction of the balanced set system can be made slightly more explicit by de-randomizing it using the method of conditional probabilities, which is standard and can be found in for example the book by Alon and Spencer (1992). The resulting complexity of the construction then becomes $2^{\tilde{O}(n^{1/3})}$ instead of $2^{\tilde{O}(n^{2/3})}$, which is the complexity of the brute force algorithm. Briefly, the de-randomization is done as follows.

Let $s = \binom{b}{\lambda+1}$ denote he number of $(\lambda+1)$-subsets of blocks, and choose some ordering of them. Let A_i denote the event that subset i is illegal (contains a common t-subset of X). We have that the expected number of illegal $\lambda+1$-subsets equals

$$\sum_{i=1}^{s} \mathrm{P}[A_i] = \left(\binom{n-t}{q-t} / \binom{n}{q}\right)^{\lambda+1} \binom{n}{t} s < 1$$

from the computation of the upper bound for the probability P above.

The deterministic construction of blocks is done by adding elements one by one to the blocks (add one element to the first block, one element to the second block, and so on). Suppose we have added k elements to each block, and let C denote this event. We are now about to add an element to position $k+1$ in the first block. To do this, we first compute the expected number of illegal $(\lambda+1)$-subsets that would result by adding an element to this position:

$$\begin{aligned}\sum_{i=1}^{s} \mathrm{P}[A_i \mid C] &= \frac{1}{n-k} \sum_{r=1}^{n-k} \sum_{i=1}^{s} \mathrm{P}[A_i \mid C \wedge x_{j_r} \text{ is added to the first block}] \\ &\geq \min_{1 \leq r \leq n-k} \sum_{i=1}^{s} \mathrm{P}[A_i \mid C \wedge x_{j_r} \text{ is added to the first block}],\end{aligned}$$

where $\{x_{j_r}\}_{r=1}^{n-k}$ denotes the $n-k$ elements not already added to the first block. Now, we choose to add the element x_{j_r} for which the minimum occurs in the last inequality.

Thus, if we in each step of the algorithm choose to add the element that minimizes $\sum_{i=1}^{s} \mathrm{P}[A_i \mid C \wedge x_{j_r} \text{ is added}]$, then the value of this sum can not increase. Since this sum is less than 1 at the beginning, it is less than 1 at the end, and then we have succeeded in constructing the blocks so that no illegal subsets of blocks occur.

The complexity of computing the sum in each step of the algorithm is $2^{\tilde{O}(n^{1/3})}$, and since there are $bq \in 2^{\tilde{O}(n^{1/3})}$ positions for which this sum must be computed, the complexity of this algorithm is $2^{\tilde{O}(n^{1/3})}$.

Although this improves the running time of the trivial algorithm, new results for combinatorial designs would, of course, be needed to obtain a truly explicit function.

3.5.3 Lower bound proof

In this section we prove the lower bound for functions based on balanced set systems. Recall from the Section 3.5.1 that $q = n^{2/3}$, $t = (n/\ln^2 n)^{1/3}$, $\lambda = 3\ln n/(\ln\ln n - \ln 8)$, and $b = (n/\ln n)^{t/3}$.

Theorem 3.19. *The monotone circuit complexity for a function f defined as in Section 3.5.1, based on balanced set systems, is $(\ln n)^{\Omega(n^{1/3}/\ln^{2/3} n)}$.*

Proof. Let the set of positive test inputs be the minterms that define the function; there are b positive test inputs. The negative test inputs are constructed randomly, by having each possible positive literal appear with probability $1-\epsilon$ for $\epsilon = (\ln n/n)^{1/3}$.

Lemma 3.20. *The probability that $f(x) = 1$ for a negative test input x is at most $1/4$.*

Proof. The probability that the q specific literals that correspond to a minterm are present in a randomly chosen negative test input is $(1-\epsilon)^q$. Therefore, the probability that the edges corresponding to at least one of the b possible minterms are present in a random input is at most

$$\begin{aligned} b(1-\epsilon)^q \le be^{-\epsilon q} &= \Big(\frac{n}{\ln n}\Big)^{\frac{1}{3}(n/\ln^2 n)^{1/3}} e^{-n^{2/3}(\ln n/n)^{1/3}} \\ &= \Big(\frac{n}{\ln n}\Big)^{\frac{1}{3}(n/\ln^2 n)^{1/3}} n^{-(n/\ln^2 n)^{1/3}} \\ &< 1/4. \end{aligned}$$

□

We choose the parameter $d = c = t$.

Lemma 3.21 (Specialization of Generic Lemma A)**.** *At the output gate of C, e_o, we either have that the approximator $f^{\mathrm{D}}_{e_o}$ is identically 0, or the probability that the approximators for e_o is 1 for a random negative test input is at least $1/2$.*

Proof. If $f_{e_o}^{\mathrm{D}}$ is identically 0 we are in the first case. Otherwise, there is an input x such that $f_{e_o}^{\mathrm{D}}(x) = 1$, and hence there is a conjunction C in $f_{e_o}^{\mathrm{D}}$. This conjunction contains at most c literals, and each literal has the probability ϵ to be 0 for a random negative test input. Thus, with probability at most $c\epsilon = (1/\ln n)^{1/3}$ the conjunction outputs 0. □

Lemma 3.22 (Specialization of Generic Lemma B)**.** *At an AND gate E, f_E^{D} introduces an error for at most λd^c positive test inputs.*

Proof. When rewriting f_E^{C} to f_E^{D}, we can form at most d^c conjunctions that contain c distinct literals, and for the current function, each removed conjunction of length c introduces an error on at most λ positive test inputs. □

Lemma 3.23 (Specialization of Generic Lemma C)**.** *At an OR gate E the probability that f_E^{C} introduces an error for a random negative test input is at most $(c\epsilon)^d$.*

Proof. We bound the probability of an error being introduced for a random negative test graph by multiplying the number of disjunctions with d distinct literals, c^d, by the probability that the d literals are 0 in one disjunction, which is ϵ^d. □

To prove the theorem we have to consider the two cases from Lemma 3.21. If the approximators fail for all positive test inputs we use Lemma 3.22 which states that the number of errors introduced in a single AND gate is at most λd^c to get

$$\mathrm{size}(C) \geq \frac{b}{\lambda d^c} = \frac{\ln\ln n - \ln 8}{3\ln n}(\ln n)^{c/3} \in (\ln n)^{\Omega(n^{1/3}/\ln^{2/3} n)}.$$

If, on the other hand, the approximator $f_{e_o}^{\mathrm{C}}$ is 1 with probability 1/2 for a random negative test input, the approximators for e_o fail for a random negative test input with probability at least 1/4. The probability of introducing an error in a single OR gate is at most $(c\epsilon)^d$; thus

$$\mathrm{size}(C) \geq \frac{1}{4(c\epsilon)^d} = \frac{(\ln n)^{d/3}}{4} \in (\ln n)^{\Omega(n^{1/3}/\ln^{2/3} n)}.$$

□

3.6 Lower bounds for monotone real circuits

A *monotone real circuit* consists of gates with fan-in two that can compute any monotone real function of its two inputs; for the gate e call this function r_e. Let $e(x)$ denote the real output value from the gate e when the input x is applied to the circuit. The output of the circuit is required to be 0 or 1. This is the model used by Pudlák (1997), Haken and Cook (1996), and by Jukna (1997).

In this section we show how the method of approximation, using our formalism, can be extended to work for monotone real circuits such that Theorem 3.3, Theorem 3.8, Theorem 3.13, and Theorem 3.19 essentially hold for this model as well.

The approach we use to approximate the outputs of the real gates in the circuit is that we consider thresholds of the real gates and approximate the thresholded values. Let $e^t(x)$ denote the boolean function that is 1 if $e(x) \geq t$ and 0 otherwise.

For a gate e with the two inputs e_1 and e_2 the threshold function e^t can be determined from the thresholds of its input gates as

$$e^t = \bigvee_{r_e(t_1,t_2)\geq t} e_1^{t_1} \wedge e_2^{t_2}$$

or

$$e^t = \bigwedge_{r_e(t_1,t_2)<t} e_1^{t_1} \vee e_2^{t_2}.$$

It is convenient to think about these expressions as an infinite OR and infinite AND, respectively. However, since the number of settings of the input variables is finite, the real gates only take discrete values, and therefore, we really only need finite expressions.

Every threshold e^t is approximated by two functions: $f^{\mathrm{D}}_{e^t}$ which is a disjunction of conjunctions of length less than c, and $f^{\mathrm{C}}_{e^t}$ which is a conjunction of disjunctions of length less than d. The approximators for the thresholds of the inputs are 0, 1, or the input itself, depending on the value of t; they can always be represented by at most one literal and thus never fail.

To construct the approximator $f^{\mathrm{C}}_{e^t}$ from the approximators of its two input gates e_1 and e_2, we first form

$$\bigvee_{r_e(t_1,t_2)\geq t} f^{\mathrm{D}}_{e_1^{t_1}} \wedge f^{\mathrm{D}}_{e_2^{t_2}}. \tag{3.3}$$

All the subexpressions $f^{\mathrm{D}}_{e_1^{t_1}} \wedge f^{\mathrm{D}}_{e_2^{t_2}}$ can be turned into a single disjunction of conjunctions, where each conjunction is at most twice the length of the conjunctions in $f^{\mathrm{D}}_{e_1^{t_1}}$ and $f^{\mathrm{D}}_{e_2^{t_2}}$. Thus, the result is a disjunction of conjunctions, and we use the same procedure as before to convert it into the approximator $f^{\mathrm{C}}_{e^t}$.

When forming $f^{\mathrm{C}}_{e^t}$, we introduce errors on negative test inputs as we throw away disjunctions that become longer than d. We want to count the number of inputs x such that $e^t(x) = 0$ while $f^{\mathrm{C}}_{e^t}(x) = 1$ for some t, i.e., the union over t of the errors introduced in a gate by $f^{\mathrm{C}}_{e^t}$.

When the value of t is decreased, the number of conjunctions obtained in (3.3) increases; let $m(t)$ denote the number of conjunctions. Arrange the conjunctions C_i such that $\{C_i \mid i \leq m(t')\} \subseteq \{C_i \mid i \leq m(t)\}$ for $t' \geq t$. For the tree that

corresponds to the rewriting process, this means that smaller values of t makes deeper trees, but with the top unchanged. So, when we bound the number of errors introduced in $f^{\mathrm{C}}_{e^t}$, we count not only possible errors introduced for t, but also errors introduced for all $t' \geq t$.

Thus, to get a bound for the number of negative test inputs for which we introduce an error for some t, we may consider $f^{\mathrm{C}}_{e^{-\infty}}$ and count the number of negative test inputs rejected by disjunctions corresponding to nodes at level d in the tree.

The number of such negative test inputs can be counted in the same way as for monotone circuits with AND and OR gates, the only difference being that the conjunctions have length $2c$ instead of c. This means that the branching factor of the tree used in the construction of f^{C} is twice of what it is in the boolean case.

A dual argument can be used to bound the number of errors introduced for positive test inputs when constructing the approximator $f^{\mathrm{D}}_{e^t}$, using $t = +\infty$.

At the output gate e_o, we get a bound for the number of inputs for which the approximators fail using exactly the same argument as for AND/OR circuits. The reason for this is that e_o only takes the values 0 and 1, so approximating any threshold between these values is the same as approximating the output value of a circuit consisting of AND and OR gates.

Now, a lower bound for the size of a real circuit can be obtained in the same way as for monotone circuits consisting of AND and OR gates. For an input for which the approximators fail at the output, the error must have been introduced in one of the real gates in the circuit, so we can divide a lower bound for the number of errors at the inputs by an upper bound for the number errors introduced in each gate.

Theorem 3.24 (Pudlák, 1997, Jukna, 1997)**.** *For* $\mathrm{POLY}(q, s)$*, the monotone real circuit complexity is* $q^{\Omega(s)}$ *when* $s \leq \frac{1}{3}\sqrt{q/\ln q}$*.*

Proof. The proof is the same as the proof of Theorem 3.3 with the exception that the number of errors introduced for positive test graphs in one gate is at most $(2d)^c$, and that the probability of introducing an error for a negative test graph in one gate is at most $(2c\epsilon)^d$. The differences are due to the higher branching factors in the trees used to construct the approximators. □

Theorem 3.25 (Pudlák, 1997, Jukna, 1997)**.** *For* $\mathrm{CLIQUE}(m, n)$*, we have that the monotone real circuit complexity is* $2^{\Omega(\sqrt{m})}$ *when* $m \leq n^{2/3}$*.*

Proof. The proof is almost identical to the proof of Theorem 3.8; we only have to change the constants c and d.

Let $c = \lfloor\sqrt{m/2}\rfloor$. Implicitly, the number of literals in conjunctions is bounded by $c^2/2 \leq m/4$.

Let $d = \lfloor n/(16m)\rfloor$. This implies that the graphs corresponding to a disjunctions touch less than $2d \leq n/(8m)$ vertices.

The bound for the number of inputs for which the approximators fail for the output gate still holds, since making c and d smaller only helps for this bound.

For the number of errors introduced in one gate for positive test graphs, notice that the OR of two disjunctions that each touch at most $2d$ vertices touches at most $4d$ vertices.

With the new constants, the bounds given in Lemma 3.10 and Lemma 3.11 still hold. It is easily verified that difference in the obtained bound of the theorem, using the new constants, is absorbed by the implicit constant in the Ω notation. □

Theorem 3.26 (Haken and Cook, 1996)**.** *The monotone real circuit complexity for* BMS(m) *is* $2^{\Omega(\sqrt{m})}$.

Proof. The proof is almost identical to the proof of Theorem 3.13; we only have to change the constant c.

Let $c = \lfloor \sqrt{m/2} \rfloor$. Implicitly, the number of literals in conjunctions is bounded by $c^2/2 \leq m/4$.

As for Clique, the bound for the number of inputs for which the approximators fail for the output gate still holds.

With the new constants, the bounds given in Lemma 3.10 and Lemma 3.11 still hold. It is easily verified that the bound of the theorem, using Ω notation, is not affected by the new constants. □

Theorem 3.27. *For a function based on balanced set systems with the parameters* $q = n^{2/3}$, $t = (n/\ln^2 n)^{1/3}$, $\lambda = 3\ln n/(\ln\ln n - \ln 8)$, *and* $b = (n/\ln n)^{t/3}$ *(see Section 3.5.1), is* $(\ln n)^{\Omega(n^{1/3}/\ln^{2/3} n)}$.

Proof. The proof is the same as the proof of Theorem 3.19 with the exception that the number of errors introduced for positive test inputs in one gate is at most $(2d)^c$, and that the probability of introducing an error for a negative test input in one gate is at most $(2c\epsilon)^d$. Both these changes are absorbed by the implicit constant in the Ω notation in the bound of the theorem. □

3.7 Decision trees as approximators?

A well known lemma (see for example Linial et al. (1989)), states that a function that can be written as both a DNF and a CNF formula, terms and clauses having at most t and s literals, respectively, can be represented by a decision tree of depth st. Is is therefore tempting to try to represent the two approximators for each gate in the proofs in this chapter by a decision tree of depth cd. This would require only one approximator with a simple and natural restriction for each gate.

Of course, our case is a little bit different, since the DNF and CNF formulas do not necessarily represent exactly the same function. However, the lemma can be modified to hold for this case as well.

Lemma 3.28. *Suppose* $f(x) \leq g(x)$ *for all* x. *If the minterms of* f *have at most* t *literals, and the maxterms of* g *have at most* s *literals, then there is a decision*

tree of depth st that evaluates a function m such that $f(x) \leq m(x) \leq g(x)$ for all x.

Proof. Write f as a DNF formula: a disjunction of its minterms, and g as a CNF formula: a disjunction of its maxterms. First notice that each conjunction in the DNF formula for f has at least one literal in common with every disjunction in the CNF formula for g. This follows since if this was not the case, consider what happens if we set all the variables that occur in the conjunction for which this is not the case to 1, and all other variables to 0. The DNF formula then gets the value 1, and the CNF formula the value 0, a contradiction.

The proof is by induction on s. The basis $s = 0$ corresponds to g being constant, so a decision tree of depth 0 that always outputs this constant computes an adequate m.

Suppose the lemma holds for all $s < s'$ and consider $s = s'$. If there are no minterms of f then $f = 0$ and we get a decision tree of depth 0. Otherwise, we construct the decision tree by first making a complete decision tree of depth at most t for the first conjunction in the DNF formula for f. We let the leaf that outputs 1 remain in the final decision tree, since for the variable setting on the path to it, we clearly have $f = 1$, and therefore also $g = 1$.

For a leaf that outputs the value 0, the function m we want to compute is between two functions f' and g', where f' and g' are the functions that result if setting variables in f and g respectively according to the path to the leaf.

The path will surely set one literal in each disjunction for g. If set to 0, the literal can be dropped from the disjunction, and if set to 1, the entire disjunction can be dropped. In any case, g' will only contain disjunctions with at most $s - 1 < s'$ literals (notice that g' may be a constant function, and will surely be if $s = 1$).

We have that $f \leq g$, and setting some variables in both f and g can not change this relationship; hence $f' \leq g'$. We now use the induction hypothesis to find a decision tree for a function m' such that $f' \leq m' \leq g'$; this tree has, according to the induction hypothesis, depth at most $(s - 1)t$, which together with the tree for the expanded conjunction makes a total depth of at most st. □

Notice that the way the one-sidedness goes is important: if we had a bound for the length of the maxterms for f and minterms for g instead, the proof would not work.

Unfortunately, however, it turns out that the decision tree constructed in the lemma above is not necessarily monotone. A decision tree of depth 1 that outputs the negation of one input variable approximates the functions for which we want to prove lower bounds quite well on test inputs, and therefore monotonicity of the decision trees are essential.

To see that the construction may yield a non-monotone function, consider the following counter example:

$$g = (x_1 \vee x_5) \wedge (x_4 \vee x_5 \vee x_2),$$
$$f = (x_1 \wedge x_2) \vee (x_1 \wedge x_3 \wedge x_4) \vee (x_3 \wedge x_5) \vee (x_4 \wedge x_5).$$

Clearly, f and g are monotone, and we have that $f(x) \leq g(x)$ for all x, since each term of f has a common variable with each clause of g. Now consider what happens when we use the construction of the lemma.

We write $(x_1, x_2, x_3, x_4, x_5)$, where $x_i \in \{0, 1, *\}$ to denote an assignment of a subset of the variables; $x_i = *$ indicates that x_i is undecided. Let f_i and g_i be the functions that remain after i conjunctions of f have been expanded. In one path of the constructed decision tree, the following happens.

The first conjunction of f contains x_1 and x_2 which are set to $x_1 = x_2 = 0$:

$$g_1 = g\lceil_{(0,0,*,*,*)} = x_5 \wedge (x_4 \vee x_5),$$
$$f_1 = f\lceil_{(0,0,*,*,*)} = (x_3 \wedge x_5) \vee (x_4 \wedge x_5).$$

In the first conjunction of f_1, set $x_3 = 0$, $x_5 = 1$:

$$g_2 = g\lceil_{(0,0,0,*,1)} = 1,$$
$$f_2 = f\lceil_{(0,0,0,*,1)} = x_4.$$

At this point, the decision tree outputs 1 since g_2 is decided.

In another path of the same decision tree, we get the following outcome.

The first conjunction of f_0 contains x_1 and x_2 which are set to $x_1 = 1$, $x_2 = 0$:

$$g_1 = g\lceil_{(1,0,*,*,*)} = x_4 \vee x_5,$$
$$f_1 = f\lceil_{(1,0,*,*,*)} = (x_3 \wedge x_4) \vee (x_3 \wedge x_5) \vee (x_4 \wedge x_5).$$

In the first conjunction of f_1, set $x_3 = x_4 = 0$:

$$g_2 = g\lceil_{(1,0,0,0,*)} = x_5,$$
$$f_2 = f\lceil_{(1,0,0,0,*)} = 0.$$

In this case, the decision tree outputs 0 since f_2 is decided.

The function that is computed by the decision tree that results from the construction is, thus, not monotone: setting $(x_1, x_2, x_3, x_4, x_5)$ to $(0, 0, 0, 0, 1)$ outputs 1, and setting the the variables to $(1, 0, 0, 0, 1)$ outputs 0.

3.8 Open problems

As we stated in the introduction we believe $\Omega(2^{n^{1/3}})$ to be essentially the best possible lower bound that can be proved with the method of approximation. It

would be nice to prove this, and also to find a truly explicit function for which this bound holds.

On an orthogonal level, a proof that perfect matching in a bipartite graph requires super-polynomial circuit size with some other set of approximators than that used by Razborov would be interesting. In particular, it would be nice to avoid the sunflower lemma in this proof.

Chapter 4

A Lower Bound for Perceptrons

4.1 Introduction

It has been shown in a sequence of papers (Sipser, 1983; Yao, 1985; Håstad, 1987, 1989) that there are functions computable by linear size circuits of depth k that require exponential size circuits of depth $k-1$.

A perceptron is a circuit where the output (top) gate has been replaced by a threshold gate. The inputs to this gate are the outputs of boolean circuits with AND, OR, and NOT gates, called the perceptron's sub-circuits. A depth k perceptron has sub-circuits of depth $k-1$.

Green (1991) used a result by Boppana and Håstad (1987) on approximating parity to prove a lower bound for the size of constant depth perceptrons that compute parity. This bound implies the existence of an oracle that separates $\oplus$P from $\mathrm{PP}^{\mathrm{PH}}$.

Green (1995) also discussed the question of whether there is an oracle that separates the levels in the $\mathrm{PP}^{\mathrm{PH}}$ hierarchy. Since this follows from a sufficiently strong lower bound for the size of depth $k-1$ perceptrons computing functions computable by perceptrons of depth k, Green was working on such a lower bound. He was able to successfully prove an exponential lower bound for depth 3 monotone perceptrons computing a function computable by linear size depth 4 perceptrons, and concluded that if the same result could be proved in the non-monotone case, the Håstad switching lemma could be used to show the separation for all k.

In this chapter we show that there are functions computable by linear size boolean circuits of depth k that require super-polynomial size perceptrons of depth $k-1$, for $k < \log n/(6 \log\log n)$, and exponential size perceptrons for constant k.

The key to making the proof work is to use a separation between polynomial size depth 2 perceptrons with bounded fan-in and general polynomial size depth 2 perceptrons as the basis for the induction. This separation follows from the one-in-a-box theorem (Minsky and Papert, 1988). To use this simpler basis (compared to that Green suggested), we need a somewhat stronger statement of Håstad's switching lemma. The statement as formulated in this chapter actually follows from looking at the proof by Håstad more carefully; we provide the modified proof for completeness.

4.2 The lower bound

We begin this section by defining the function f_k^m, first defined by Sipser (1983), which can be computed by linear size circuits of depth k. Then we show the main theorem, which states that perceptrons of depth k with bounded fan-in that compute this function must be large. As a corollary we get that perceptrons of depth $k-1$ computing f_k^m must be large.

A perceptron is a circuit with a single threshold gate at the top. The threshold gate may have arbitrary weights and outputs 1 if its weighted sum of the input variables exceeds some threshold value. The inputs to the threshold gate are outputs of boolean circuits, the perceptron's *sub-circuits*. The sub-circuits consist of alternating levels of AND and OR gates; their top gate is either an AND gate for all the sub-circuits, or they may all be OR gates. Input variables may be negated, but otherwise no NOT gates may occur in the circuits. A general perceptron can be transformed into this form by at most doubling its size. A depth k perceptron has sub-circuits of depth $k-1$.

Definition 4.1. The function f_k^m uses $m^{k-2}\frac{1}{2}(m\log m)^{1/4}(\frac{1}{2}km\log m)^{1/2}$ input variables. It is defined by a depth k circuit that has the form of a tree. At the leaves of the tree are unnegated variables.

The root is an OR gate with fan-in $\frac{1}{2}(m\log m)^{1/4}$. Below are alternating levels of AND and OR gates with fan-in m. The bottom-most level has fan-in $(\frac{1}{2}km\log m)^{1/2}$.

Recall from Chapter 2 that log denotes the base 2 logarithm.

The theorem is proved analogously to Håstad's proof that circuits of depth k are more powerful than circuits of depth $k-1$. A central part of the proofs is the Håstad switching lemma, which states that if we set a subset of the input variables to 0 or 1 randomly, we can switch the two lowest levels of AND and OR gates without increasing the circuit size very much. Then, two levels of the circuit can be collapsed into one.

The way that we set some input variables of a circuit C to 0 and 1 is through the use of *restrictions*. A restriction ρ is a mapping of the variables to the set $\{0,1,*\}$, where $*$ means that the variable remains unset. Let $C\lceil_\rho$ denote the circuit obtained by replacing each input variable x_i by $\rho(x_i)$. We use the same

distribution of restrictions as Håstad used in (Håstad, 1987, Chapter 6); they are defined below.

Definition 4.2. Let q be a real number and $(B_i)_{i=1}^r$ a partition of the variables (that is, the B_i are disjoint and their union equals the set of all variables). Let $R_{q,B}^+$ be the probability space of restrictions which takes values as follows. For $\rho \in R_{q,B}^+$ and every B_i, $1 \leq i \leq r$, independently

1. With probability q let $s_i = *$ and else $s_i = 0$.
2. For every $x_k \in B_i$ let $\rho(x_k) = s_i$ with probability q and else $\rho(x_k) = 1$.

Similarly a $R_{q,B}^-$ probability space of restrictions is defined by interchanging the roles played by 0 and 1.

Definition 4.3. For a restriction $\rho \in R_{q,B}^+$, let $g(\rho)$ be a restriction defined as follows: for all B_i with $s_i = *$, $g(\rho)$ gives the value 1 to all variables given the value $*$ by ρ, except one to which it gives the value $*$. To make $g(\rho)$ deterministic, we assume that it gives the value $*$ to the variable with the highest index given the value $*$ by ρ.

If $\rho \in R_{q,B}^-$, then $g(\rho)$ is defined similarly, but now takes the values 0 and $*$.

It follows from the definition of $g(\rho)$ that it never assigns values to the same variables as ρ, and for a circuit C, we denote $(C\restriction_\rho)\restriction_{g(\rho)}$ by $C\restriction_{\rho g(\rho)}$.

The variables can be partitioned according to what gates they occur at in the circuit that defines f_k^m. When the restrictions are used, the blocks B_i in the previous two definitions correspond to this partitioning.

We use the Håstad switching lemma for this distribution of restrictions (Håstad, 1987, Lemma 6.3), and in the proof of the main theorem, we need an important property of the switching lemma: after switching, the inputs accepted by the different ANDs form disjoint sets. The fact that this property follows from the proof of the switching lemma was noted by Håstad (1987, Lemma 8.3). Their note, however, was regarding another distribution of random restrictions, and was not explicitly proved; for this reason we include a sketch of the proof where we emphasize the "disjointness property."

Lemma 4.4. *Let G be an AND of ORs, all of size at most t, and ρ a random restriction from $R_{q,B}^+$ or $R_{q,B}^-$. Then the probability that $G\restriction_{\rho g(\rho)}$ can not be written as an OR of ANDs all of size less than s, where the inputs accepted by the different ANDs form disjoint sets, is bounded by α^s where $\alpha = 6qt$.*

The analog lemma is also true for converting an OR of ANDs to an AND of ORs: with high probability $G\restriction_{\rho g(\rho)}$ can be written as an AND of ORs all of size less than s, where the inputs rejected by the different ORs form disjoint sets.

Proof. Let $\mathrm{AND}(G\restriction_{\rho g(\rho)}) \geq s$ denote the event that $G\restriction_{\rho g(\rho)}$ can not be written as an OR of ANDs that accept disjoint sets, all ANDs having size less than s. This definition is slightly different from that in Håstad's original proof, where

the same notation is used to denote the event that $G\lceil_{\rho g(\rho)}$ can not be written as an OR of ANDs of size less than s.

We use induction to prove the lemma, and in fact, we prove that if $G = \wedge_{i=1}^{w} G_i$, where G_i are ORs of fan-in at most t, then

$$\mathrm{P}[\mathrm{AND}(G\lceil_{\rho g(\rho)}) \geq s \mid F\lceil_{\rho} \equiv 1] \leq \alpha^s,$$

for an arbitrary F.

The basis $w = 0$ is obvious. For the induction step first rewrite

$$\begin{aligned}\mathrm{P}[\mathrm{AND}(G\lceil_{\rho g(\rho)}) \geq s \mid & F\lceil_{\rho} \equiv 1] \\ &\leq \max(\,\mathrm{P}[\mathrm{AND}(G\lceil_{\rho g(\rho)}) \geq s \mid F\lceil_{\rho} \equiv 1 \wedge G_1\lceil_{\rho} \equiv 1], \\ &\qquad \mathrm{P}[\mathrm{AND}(G\lceil_{\rho g(\rho)}) \geq s \mid F\lceil_{\rho} \equiv 1 \wedge G_1\lceil_{\rho} \not\equiv 1]).\end{aligned}$$

The first term is taken care of by the induction hypothesis, and the rest of the proof deals with the estimate of the second term.

We denote the set of variables occuring in G_1 by T and we have that $|T| \leq t$. If $G\lceil_{\rho}$ is identically 0, $G\lceil_{\rho g(\rho)}$ does not require long ANDs, so we may assume that it is not 0, and therefore that some of the variables in T must be given the value $*$ by ρ.

A block B is *exposed* if there is a variable $x_i \in B \cap T$ and $\rho(x_i) = *$. If the variables in T belong to the r different blocks $B_1, \ldots, B_r$, we know that some of these blocks are exposed. Let $\exp(Y)$, $Y \subseteq \{1, \ldots, r\}$, denote that the blocks indexed by Y are the ones that are exposed.

We get

$$\begin{aligned}&\mathrm{P}[\mathrm{AND}(G\lceil_{\rho g(\rho)}) \geq s \mid F\lceil_{\rho} \equiv 1 \wedge G_1\lceil_{\rho} \not\equiv 1] \\ &\leq \sum_{\emptyset \neq Y \subseteq \{1,\ldots,r\}} \mathrm{P}[\exp(Y) \mid F\lceil_{\rho} \equiv 1 \wedge G_1\lceil_{\rho} \not\equiv 1] \\ &\qquad \cdot \mathrm{P}[\mathrm{AND}(G\lceil_{\rho g(\rho)}) \geq s \mid F\lceil_{\rho} \equiv 1 \wedge G_1\lceil_{\rho} \not\equiv 1 \wedge \exp(Y)].\end{aligned}$$

In this sum, the first factor is identical to the corresponding first factor in the original proof by Håstad, which uses the following lemma.

Lemma 4.5 (Håstad (1987, Lemma 6.6)). $\mathrm{P}[\exp(Y) \mid F\lceil_{\rho} \equiv 1 \wedge G_1\lceil_{\rho} \not\equiv 1] \leq (2q)^{|Y|}$.

It remains to estimate the second factor, which differs from that in the original proof.

Let Y^* denote the set of variables that remain in the exposed blocks after the restriction $\rho g(\rho)$ when the blocks indexed by Y are exposed. Notice that $|Y| = |Y^*|$ since $\rho g(\rho)$ assigns only one $*$ for each block. We rewrite

$$G = \bigvee_{\sigma \in \{0,1\}^{|Y|}} G\lceil_{Y^* = \sigma} \wedge Q(Y^*, \sigma),$$

where $Q(Y^*, \sigma)$ denotes a predicate that is true when the undecided variables in Y take the values of σ; it can be written as an AND of size $|Y|$.

Partition ρ into ρ_1, that is the restriction on the exposed blocks, and ρ_2, that is the restriction on the blocks that are not exposed. We write $\rho = \rho_1\rho_2$.

We get

$$\begin{aligned}
&\mathrm{P}_\rho[\,\mathrm{AND}(G\restriction_{\rho g(\rho)}) \geq s \mid F\restriction_\rho \equiv 1 \wedge G_1\restriction_\rho \not\equiv 1 \wedge \exp(Y)] \\
&\quad = \mathrm{P}_\rho[\mathrm{AND}(G\restriction_{\rho g(\rho)}) \geq s \mid F'\restriction_{\rho_2} \equiv 1 \wedge G_1\restriction_{\rho_1} \not\equiv 1 \wedge \exp(Y)] \\
&\quad = \mathrm{P}_\rho\Big[\mathrm{AND}\Big(\bigvee_{\sigma\in\{0,1\}^{|Y|}} (G\restriction_{Y^*=\sigma})\restriction_{\rho g(\rho)} \wedge Q(Y^*,\sigma)\Big) \geq s \\
&\qquad\qquad \Big| \; F'\restriction_{\rho_2} \equiv 1 \wedge G_1\restriction_{\rho_1} \not\equiv 1 \wedge \exp(Y)\Big] \\
&\quad \leq \sum_{\sigma\in\{0,1\}^{|Y|}} \mathrm{P}_\rho[\mathrm{AND}((G\restriction_{Y^*=\sigma})\restriction_{\rho g(\rho)}) \geq (s-|Y|) \\
&\qquad\qquad \mid F'\restriction_{\rho_2} \equiv 1 \wedge G_1\restriction_{\rho_1} \not\equiv 1 \wedge \exp(Y)] \\
&\quad \leq \sum_{\sigma\in\{0,1\}^{|Y|}} \max_{\rho_1} \mathrm{P}_{\rho_2}[\mathrm{AND}((G\restriction_{Y^*=\sigma}\restriction_{\rho_1 g(\rho_1)})\restriction_{\rho_2 g(\rho_2)}) \geq (s-|Y|) \\
&\qquad\qquad \mid F'\restriction_{\rho_2} \equiv 1] \\
&\quad \leq 2^{|Y|}\alpha^{s-|Y|}.
\end{aligned}$$

For the first equality, note that the condition $G_1\restriction_\rho \not\equiv 1$ is equivalent to $G_1\restriction_{\rho_1} \not\equiv 1 \wedge G_1\restriction_{\rho_2} \not\equiv 1$. Since ρ_2 assigns only the values 0 and 1 to variables in T, the second of these requirements can be combined with $(F\restriction_{\rho_1})\restriction_{\rho_2} \equiv 1$, and we write this as $F'\restriction_{\rho_2} \equiv 1$.

In the second equality, we do not have to apply the restriction to $Q(Y^*,\sigma)$, since the variables in Y^* are not affected by it anyway.

The first inequality is obtained by noting that if for all σ, $(G\restriction_{Y^*=\sigma})\restriction_{\rho g(\rho)}$ can be written as an OR of ANDs, all of length less than $s-|Y|$, and all of which accept disjoint input sets, then $G\restriction_{\rho g(\rho)}$ can be written as an OR of ANDs, all of length at most s, in a way that they all accept disjoint input sets.

In the second inequality we use that for boolean predicates R and S we have $\mathrm{P}_{\rho_1,\rho_2}[R(\rho_1,\rho_2) \mid S(\rho_1)] \leq \max_{\rho_1 : S(\rho_1)} \mathrm{P}_{\rho_2}[R(\rho_1,\rho_2)]$ to get rid of the two last conditions.

We are finally in a position to use the induction hypothesis, which is done in the last inequality.

We now evaluate the sum to get

$$\begin{aligned}
\sum_{\emptyset\neq Y\subseteq\{1,\ldots,r\}} (2q)^{|Y|}2^{|Y|}\alpha^{s-|Y|} &= \alpha^s \sum_{i=1}^{r} \binom{r}{i}\left(\frac{4q}{\alpha}\right)^i \\
&= \alpha^s((1+4q/\alpha)^r - 1) \leq \alpha^s((1+2/(3t))^t - 1) \\
&\leq \alpha^s((e^{2/(3t)})^t - 1) \leq \alpha^s.
\end{aligned}$$

□

The next lemma shows that applying a restriction to the defining circuit for f_k^m does not reduce it too much: with high probability the remaining circuit computes f_{k-1}^m. We fix the parameter q that is used for the restrictions to be $(\frac{2k}{m}\log m)^{1/2}$ for the rest of the paper.

Lemma 4.6 (Håstad (1987, Lemma 6.8))**.** *If k is even the circuit that defines $f_k^m\lceil_{\rho g(\rho)}$ for a random $\rho \in R_{q,B}^+$, will contain the circuit that defines f_{k-1}^m with probability at least $2/3$, for all m such that $m/\log m \geq 100k$, $m > m_1$, where m_1 is some constant. The same is true for odd k when $R_{q,B}^+$ is replaced by $R_{q,B}^-$.*

Proof. Suppose k is even so that the restriction comes from $R_{q,B}^+$ and the lowest level of the defining circuit for f_k^m consists of AND gates; the other case is analogous.

The probability that one or more of the AND gates on the lowest level is reduced to the constant 1 by the restriction, or, equivalently, that one or more OR gates on the next lowest level is reduced to the constant 1 by the restriction, is bounded by 1/6. This fact follows since the AND gate corresponding to the block B_i takes the value 1 with probability

$$\begin{aligned}(1-q)^{|B_i|} &= (1-(\tfrac{2k}{m}\log m)^{1/2})^{\sqrt{\frac{1}{2}km\log m}} \\ &= \left((1-(\tfrac{2k}{m}\log m)^{1/2})^{-\sqrt{\frac{m}{2k\log m}}}\right)^{-k\log m} \\ &< e^{-k\log m} \\ &< \frac{1}{6}m^{-k},\end{aligned}$$

and since there are less than m^k AND gates on the bottom-most level.

Now suppose we are in the (good) case that all AND gates take the values s_i. The probability that the number of remaining inputs to an OR gate (i.e., the number of AND gates not forced to 0) is less than $\sqrt{\frac{1}{2}km\log m}$ is at most $1/6m^{-k}$. This follows since the expected number of such gates is $qm = \sqrt{2km\log m}$, and using Chernoff's inequality we get that the probability of obtaining less than half the expected number is at most $e^{-qm/8} < 1/6m^{-k}$ for $m/\log m \geq 100k$. So, with probability 5/6 none of the OR gates will have less than $\sqrt{\frac{1}{2}m\log m}$ inputs.

To sum up, we get a circuit that contains f_{k-1}^m with probability at least 2/3. □

We use induction over the depth of the perceptron to prove the theorem. For the basis we need the following lemma.

Lemma 4.7. *The bottom fan-in of a depth two perceptron that computes f_2^m is at least $\frac{1}{2}(m\log m)^{1/4}$.*

Proof. The one-in-a-box theorem by Minsky and Papert (see Section 1.2.2) is exactly what we need. Substituting the variable m in the one-in-a-box theorem by $\frac{1}{2}(m \log m)^{1/4}$ proves the lemma. □

We are now ready to prove the main theorem.

Theorem 4.8. *There are no depth k perceptrons computing f_k^m with bottom fan-in $\frac{1}{\sqrt{3k}}(m \log m)^{1/4}$ and less than $2^{\frac{1}{\sqrt{3k}}(m \log m)^{1/4}}$ gates, not counting the gates on the lowest level, for $m \geq m_1$, where m_1 is some constant.*

Proof. Without losing generality, we may assume that $m \log m \geq k^2$, since otherwise we have that $2^{\frac{1}{\sqrt{3k}}(m \log m)^{1/4}} \leq 2^{1/\sqrt{3}}$.

The theorem is proved by induction over k. The basis $k = 2$ follows from Lemma 4.7. We first give an overview for the proof of the induction step, which is proved by contradiction.

We assume that there is a small-sized depth-k perceptron with bounded fan-in computing f_k^m. Then, we apply a random restriction to the inputs, and with high probability we get a perceptron that computes f_{k-1}^m. Also, because of the Håstad switching lemma, we can switch the two lower levels of ANDs and ORs in the perceptron without increasing the fan-in, and therefore collapse two levels to one and obtain a small-sized perceptron of depth $k - 1$ and bounded fan-in computing f_{k-1}^m.

When $k = 3$ the procedure above does not work, since there are only two levels of AND and OR gates in the perceptron. However, we can deal with this as follows. Suppose that the lower level of the depth-three perceptron consists of OR gates (the other case is analogous). When switching an AND of ORs to an OR of ANDs, Lemma 4.4 says that the input sets accepted by the ANDs are mutually disjoint. Therefore, the output from the OR gate is always equal to the sum of the outputs of the AND gates, and we can thus substitute all OR gates that resulted after switching by summation gates, and thereafter collapse the two top-most levels.

We make the intuition formal by assuming that there is a depth-k perceptron P with bottom fan-in $\frac{1}{\sqrt{3k}}(m \log m)^{1/4}$ that has less than $2^{\frac{1}{\sqrt{3k}}(m \log m)^{1/4}}$ gates, not counting the gates on the lowest level, computing f_k^m.

Apply a random restriction from $R_{q,B}^+$ or $R_{q,B}^-$ depending on whether k is even or odd respectively. From Lemma 4.6 we have that with probability at least $2/3$, the perceptron P computes a function at least as hard as f_{k-1}^m. Note that this lemma requires $m/\log m \geq 100k$ which follows from $m \log m \geq k^2$ for large enough m.

Now we want to use Lemma 4.4 to show that all the sub-circuits on the two bottom levels can be switched. We know that the bottom fan-in is at most $t = \frac{1}{\sqrt{3k}}(m \log m)^{1/4}$, and to use the induction hypothesis we have to obtain a circuit with fan-in at most $s = \frac{1}{\sqrt{3(k-1)}}(m \log m)^{1/4}$. For each single sub-circuit, the probability that this fails is at most $(6qt)^s$, which we multiply by the

maximum number of such circuits to get a bound for the probability that this fails for at least one circuit:

$$\begin{aligned}
&2^{\frac{1}{\sqrt{3k}}(m\log m)^{1/4}}(6qt)^s \\
&\quad = 2^{\frac{1}{\sqrt{3k}}(m\log m)^{1/4}}\left(6(\tfrac{2k}{m}\log m)^{1/2}\tfrac{1}{\sqrt{3k}}(m\log m)^{1/4}\right)^{\frac{1}{\sqrt{3(k-1)}}(m\log m)^{1/4}} \\
&\quad \le \left(12(\tfrac{2k}{m}\log m)^{1/2}\tfrac{1}{\sqrt{3k}}(m\log m)^{1/4}\right)^{\frac{1}{\sqrt{3(k-1)}}(m\log m)^{1/4}} \\
&\quad \le 1/2,
\end{aligned}$$

where the last inequality holds since $m\log m \ge k^2$.

Note that the number of gates that are not on the bottom-most level does not increase when switching, which we need to use the induction hypothesis.

Thus, with probability at least $2/3$, $P\lceil_{\rho g(\rho)}$ computes a function at least as hard as f_{k-1}^m, and with probability at least $1/2$, $P\lceil_{\rho g(\rho)}$ is a function that does not compute f_{k-1}^m by the induction hypothesis. Therefore, both of these events must happen simultaneously with positive probability, a contradiction. □

Corollary 4.9. *There are no depth $k-1$ perceptrons that compute f_k^m with less than $2^{\frac{1}{\sqrt{3k}}(m\log m)^{1/4}}$ gates, for $m \ge m_1$, for some constant m_1.*

Proof. Assume there is a perceptron such that the corollary does not hold. Then, adding a gate with fan-in one to all the inputs yields a depth k perceptron that does not exist according to Theorem 4.8. □

Corollary 4.10. *There are functions computable by linear size circuits of depth k that require super-polynomial size perceptrons of depth $k-1$ if $3 \le k < \frac{\log n}{6\log\log n}$, where n is the number of input variables.*

Proof. From Corollary 4.9 we have that if $\frac{1}{\sqrt{3k}}(m\log m)^{1/4} \in \omega(\log n)$ then depth $k-1$ perceptrons computing f_k^m require super-polynomial size. We have $n > m^{k-2}$ and thus $k-2 < \log n/\log m$ and $\log m/\log n < 1$. For $m\log m \ge k^2$ we have $n < m^k$ and thus $k > \log n/\log m$. We get

$$\sqrt{3k} < \sqrt{3\frac{\log n}{\log m}+6} = \sqrt{\frac{\log n}{\log m}\left(3+6\frac{\log m}{\log n}\right)} < 3\sqrt{\frac{\log n}{\log m}}$$

so that

$$\frac{1}{\sqrt{3k}}(m\log m)^{1/4} > \left(\frac{m\log^3 m}{\log^2 n}\right)^{1/4},$$

which is $\omega(\log n)$ if $m\log^3 m \in \omega(\log^6 n)$. This holds if $k < \frac{\log n}{6\log\log n}$, since we then have $(\log^6 n)^k < n < m^k$ so that $m > \log^6 n$. □

Chapter 5

Oracle Separations

5.1 Introduction

The connection between lower bounds for boolean circuits (consisting of AND, OR, and NOT gates) and relativization results about the polynomial time hierarchy (Furst et al., 1984) can be extended to a similar correspondence between the levels in $\mathrm{PP}^{\mathrm{PH}}$ and constant depth perceptrons.

We show that our result on perceptrons from Chapter 4 implies the existence of an oracle that separates the levels in the $\mathrm{PP}^{\mathrm{PH}}$ hierarchy, and in fact that there exists an A such that $\Sigma_k^{p,A} \not\subseteq \mathrm{PP}^{\Sigma_{k-2}^{p,A}}$.

The fact that our basis, i.e., the one-in-a-box theorem by Minsky and Papert, implies that $\mathrm{NP}^{\mathrm{NP}} \not\subseteq \mathrm{PP}$ under an oracle was noted by Fu (1992). Beigel (1994) has strengthened this separation to obtain that $\mathrm{P}^{\mathrm{NP}} \not\subseteq \mathrm{PP}$ under an oracle, and in the last section of this chapter we use his result as a basis for a lower bound for perceptrons with bounded weights. Using this lower bound, we get an oracle A such that $\Delta_k^{p,A} \not\subseteq \mathrm{PP}^{\Sigma_{k-2}^{p,A}}$. (Recall that Δ_k^p denotes the complexity class $\mathrm{P}^{\Sigma_{k-1}^p}$.)

Beigel et al. (1995) proved that $\mathrm{P}^{\mathrm{PP}[\log]} = \mathrm{PP}$. A relativization of this results shows that our result is almost tight.

5.2 Separating the levels of the $\mathrm{PP}^{\mathrm{PH}}$ Hierarchy

In this section we show how the lower bound from Theorem 4.8 implies the existence of an oracle that separates the different levels in the $\mathrm{PP}^{\mathrm{PH}}$ hierarchy.

An oracle A is a fixed set of strings called an *oracle set*. Let y^A denote the characteristic function for the oracle A so that y_z^A is 1 if the string z is in A.

Recall that an *alternating Turing machine* is a non-deterministic Turing machine whose states are marked by $\wedge$, $\vee$, 0, or 1. States marked $\wedge$ and $\vee$ have at most two next configurations, and states marked 0 and 1 are the halting states, in which the machine rejects or accepts, respectively. Let a $\Sigma_d^{p,A}$ machine denote

an alternating Turing machine with access to the oracle A. Such a machine has an additional oracle query tape; when the query tape contains the string z, the machine can enter a special oracle query state to compute y_z^A in one time step.

Given an input value x to a $\Sigma_d^{p,A}$ machine, the result of the computation is determined by evaluating the computation tree in the natural way. The computation tree is defined as follows. Every possible machine configuration is represented by a node, which is labeled by $\wedge$, $\vee$, 0, or 1, depending on the marking of the corresponding state. A node is the parent to those nodes that represent the possible next configurations; thus the nodes representing halting configurations are the leaves of the tree. The maximum number of blocks of consecutive states labeled by $\wedge$ or $\vee$ on a path from the root to a leaf is d, and the block of states closest to the root is an $\vee$ block.

A $\mathrm{PP}^{\Sigma_{d-1}^{p,A}}$ machine is defined like a $\Sigma_d^{p,A}$ machine, but where the machine starts its computation in a state marked by $+$ (a counting state). In the computation tree, the block closest to the root is labeled by $+$, and the machine accepts if the evaluation of the tree exceeds a threshold value.

The complexity classes $\Sigma_d^{p,A}$ and $\mathrm{PP}^{\Sigma_{d-1}^{p,A}}$ contain exactly those languages that can be decided by $\Sigma_d^{p,A}$ and $\mathrm{PP}^{\Sigma_{d-1}^{p,A}}$ machines in polynomial time, respectively.

Given an alternating Turing machine, it can be converted to a machine that makes all oracle queries at the end of the computation with only a polynomial increase in execution time and no extra alternations. By making the oracle queries at the end of the computation, we mean that the machine does not make any alternations between states marked by $\wedge$ and states marked by $\vee$ after the first oracle query has been made. We assume that all machines have this form for the rest of the chapter. The conversion of a machine that does not make all oracle queries at the end of the computation to one that does can be done as follows: Oracle queries are replaced by non-deterministic guesses for the oracle answer. Along one computation branch the machine assumes the answer 0, and along the other it assumes the answer 1; it remembers the question, the guess for the answer, and the marking of the original query state for the rest of the computation. A halting state is replaced by a number of states that verify the guesses made in the computation. If all the guesses were correct, the computation branch accepts or rejects according to the marking of the original halting state. Otherwise, the machine makes sure that the computation branch does not affect the result of the computation by accepting if the first incorrect guess was made in an $\wedge$ state, and rejecting otherwise.

The following relation between alternating oracle Turing machines and perceptrons is similar to for example Lemma 2.1 in (Ko, 1989).

Lemma 5.1. *Let M^A be a $PP^{\Sigma_{d-1}^{p,A}}$ oracle machine which runs in time t on input x. Then there is a depth $d+1$ perceptron P with unit weights and bottom fan-in t which has a subset of the y_z^A as inputs such that for every oracle A, M^A accepts x precisely when P outputs 1 on inputs y_z^A. This perceptron has at most 2^t gates, not counting gates on the bottom-most level.*

Proof. For a fixed x, write down the computation tree for $M^A(x)$ for some oracle A. On a path from the root of the tree to a leaf, let the first node where an oracle query occurs (if one exists) be called a *boundary node*. Boundary nodes mark where in the computation that the machine starts verifying its guesses, and since this is the last part of a computation branch, and since it is deterministic, there is exactly one leaf under each boundary node.

When varying the oracle A we get different computation trees. However, the part of the computation trees that are above the boundary nodes remain the same regardless of the oracle. The part of the computation trees below a boundary node accepts or rejects depending on the oracle. However, note that the only oracle queries that may occur at and below the boundary node are the ones for which guesses have been made above the boundary node; their number is bounded by t.

We now construct a perceptron from the computation tree of $M^A(x)$. Every boundary node in the computation tree is replaced by a DNF or CNF formula, depending on if the boundary node is labeled by $\vee$ or $\wedge$, respectively. In the case of boundary nodes labeled by $\vee$, we make a conjunction for each possible set of oracle answers that makes the computation branch accept, and combine them into a DNF formula. In the case of boundary nodes labeled $\wedge$, make a conjunction for each possible set of oracle answers that makes the computation branch reject, and combine them into a DNF formula. Taking the negation of this formula and applying De Morgan's law yields a CNF formula. Finally, the resulting tree is collapsed to yield a perceptron of depth $d+1$ and bottom fan-in t.

The depth of the original computation tree is at most t, and hence its size at most 2^t. Since new gates from the DNF and CNF formulas only appear on the bottom-most level, the maximum number of gates on higher levels in the resulting perceptron is at most 2^t. □

Theorem 5.2. *There exists an oracle A such that, for all d, there is a language $L_d(A)$ in $\Sigma_d^{p,A}$ which is not recognizable by any $PP^{\Sigma_{d-2}^{p,A}}$ machine, i.e., $\Sigma_d^{p,A} \not\subseteq PP^{\Sigma_{d-2}^{p,A}}$.*

Proof. The intuition for the proof is that a $\mathrm{PP}^{\Sigma_{d-2}^{p,A}}$ machine corresponds to a depth d perceptron with inputs y_z^A for each input x. Theorem 4.8 from the last section is used to show that such a perceptron is too small for computing the function f_d^m of the oracle bits for some m. We define a language $L_d(A)$ that depends on A in a way that for a machine to decide the language properly for all oracles would require the corresponding perceptron to compute f_d^m; thus we can choose an oracle such that the machine makes an error.

Let $x = x_1 x_2 \cdots x_n$ and

$$L_d(B) = \{1^n \mid \exists x_1, x_2, \ldots, x_{n/d} \forall x_{n/d+1}, \ldots, x_{2n/d} \cdots Q x_{n(d-1)/d+1}, \ldots, x_n \colon x \in B\}.$$

Now, a $\mathrm{PP}^{\Sigma_{d-2}^{p,B}}$ machine that decides if $1^n \in L_d(B)$ has a corresponding perceptron of depth d that evaluates a function h_d^n in the variables y_z^B for $|z| = n$, as exemplified in Figure 5.1.

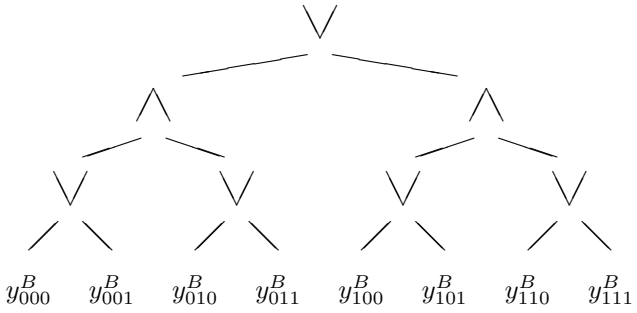

Figure 5.1. To check if $1^3 \in L_3(B)$ corresponds to evaluating the function h_3^3 given by this circuit.

The function h_d^n resembles $f_d^{2^{n/d}}$, the only difference being the fan-in on the top and bottom level which for h_d^n is $2^{n/d}$ while it is lower for $f_d^{2^{n/d}}$. This means that h_d^n contains $f_d^{2^{n/d}}$, and is therefore at least as hard to compute.

By the construction of $L_d(B)$ it is clear that $L_d(B) \in \Sigma_d^{p,B}$, and we now construct an oracle B such that $L_d(B)$ can not be decided by a $\mathrm{PP}^{\Sigma_{d-2}^{p,B}}$ machine for any d.

Let M_i^B for $i = 1, 2, \ldots$ be an enumeration of $\mathrm{PP}^{\Sigma_{d-2}^{p,B}}$ machines for all constants d. The oracle is built in rounds, and in round i we make sure that the machine M_i makes an error for some input.

In round i we do the following: Suppose M_i^B is a $\mathrm{PP}^{\Sigma_{d-2}^{p,B}}$ machine which runs in time cn^c (observe that c and d depend on i). We want to fix some of the y_z^B such that M_i^B makes an error for at least one of the strings in $L_d(B)$. More precisely, we make M_i^B fail on the string 1^{n_i}, where n_i is chosen such that both the following statements are satisfied, which they are for large enough n_i.

1. None of the y_z^B with $|z| = n_i$ have previously been set.
2. $\frac{1}{\sqrt{3d}}\left(2^{n_i/d} n_i/d\right)^{1/4} > cn_i^c$.

The first requirement makes sure that none of the strings previously set in the oracle interfere with the current machine on input 1^{n_i}. For any oracle query y_z^B with $|z| \neq n_i$ that M_i^B may make, we substitute the correct value for those that are already determined, and fix previously undetermined variables to some arbitrary value.

From Lemma 5.1 there is a perceptron of depth d, bottom fan-in cn^c, and with at most 2^{cn^c} gates (excluding those on the bottom-most level) with a subset of the y_z^B as inputs that outputs 1 for the oracles that make M_i^B accept the

string 1^{n_i}. This perceptron is, due to Theorem 4.8, not powerful enough to compute $h_d^{n_i}$ and is thus unable to determine if the string 1^{n_i} is in $L_d(B)$ for all B. It is therefore possible to set the y_z^B of length n_i such that M_i^B makes an error on 1^{n_i}. □

5.3 Improving the oracle separation

Beigel (1994) obtained the oracle separation $\mathrm{P}^{\mathrm{NP}} \not\subseteq \mathrm{PP}$. To do this he introduced the language ODD-MAX-BIT and proved a relation between the bottom fan-in, the maximum weight, and the size for perceptrons deciding it.

Definition 5.3 (Beigel (1994, Definition 2))**.** ODD-MAX-BIT is the set of all strings over $\{0,1\}^*$ whose rightmost 1 is in an odd-numbered position, i.e., the set of strings of the form $x10^k$ where the length of x is even.

We use this function to define a more complex one that is suited for obtaining a lower bound for perceptrons of depth k. We first slightly change the definition of the function f_k^m from the previous section.

Definition 5.4. The function f_k^m is a function of $m^{k-1}\sqrt{\frac{1}{2}km\log m}$ variables. It is defined by a depth k circuit that has the form of a tree. At the leaves of the tree are unnegated variables.

The bottom-most level has fan-in $\sqrt{\frac{1}{2}km\log m}$. The rest of the levels, including the root for $k > 1$, all have fan-in m. The root is an AND gate, and below are alternating levels of OR and AND gates.

The function that we obtain a lower bound for is g_k^m, defined below.

Definition 5.5. For $k = 2$, define g_2^m to be the ODD-MAX-BIT function of $\frac{1}{6}\sqrt{m\log m}$ variables.

For $k > 2$, g_k^m is a function of $m^{k-2}\sqrt{\frac{1}{2}(k-2)m\log m}$ variables. It computes the ODD-MAX-BIT function of the outputs of m disjoint f_{k-2}^m functions.

When applying a random restriction to a circuit computing g_k^m, as is the case with f_k^m, with high probability we get a circuit that computes g_{k-1}^m. As before, fix the parameter q that is used for the restrictions to be $(\frac{2k}{m}\log m)^{1/2}$.

Lemma 5.6. *If k is odd the circuit that defines $g_k^m\lceil_{\rho g(\rho)}$ for a random $\rho \in R_{q,B}^+$ will contain the circuit that defines g_{k-1}^m with probability at least $2/3$, for all m such that $m/\log m \geq 100k$, $m > m_1$, where m_1 is some constant. The same is true for even k when $R_{q,B}^+$ is replaced by $R_{q,B}^-$.*

Proof. When $k > 3$, the proof works as for Lemma 4.6. The case $k = 3$ is now special and let us consider this case; the circuit computes the ODD-MAX-BIT function of m AND gates, each with fan-in $\sqrt{\frac{1}{2}m\log m}$. As in the proof of

Lemma 4.6, with probability at least 5/6, none of the AND gates is reduced to the constant 1, and they thus take the values s_i.

When all m AND gates take the values s_i (which is 0 or $*$) by the restriction, we show that with high probability the remaining circuit contains a circuit that computes the ODD-MAX-BIT function of $\frac{1}{6}\sqrt{m \log m}$ variables.

Divide the m inputs to the ODD-MAX-BIT function (which take the values of $s_1, s_2, \ldots, s_m$) into $\sqrt{m \log m}$ blocks D_i of size $|D_i| = \sqrt{m/\log m}$. The probability that such a block D_i has a $*$ in at least one odd numbered input and also in at least one even numbered input is at least

$$\begin{aligned} 1 - 2(1-q)^{|D_i|/2} &= 1 - 2\left(\left(1 - \sqrt{\tfrac{6 \log m}{m}}\right)^{-\sqrt{\frac{m}{6 \log m}}}\right)^{-\sqrt{6}/2} \\ &> 1 - 2e^{-\sqrt{6}/2} > 1/3, \end{aligned}$$

since the probability that a block does not get any $*$ in an even (odd) numbered input is $(1-q)^{|D_i|/2}$.

We construct a new ODD-MAX-BIT circuit by using one input from every block that has a $*$ at both an odd numbered and an even numbered input. We use an odd numbered input from the first such block, an even numbered input from the next one, and so on. Thus, we obtain a circuit computing ODD-MAX-BIT of as many inputs as there are blocks having both an odd numbered and an even numbered $*$.

There are $\sqrt{m \log m}$ blocks, so the expected number of such such blocks is at least $\frac{1}{3}\sqrt{m \log m}$, and using Chernoff's inequality we obtain that the probability of getting less than $\frac{1}{6}\sqrt{m \log m}$ such blocks is at most $e^{-\sqrt{m \log m}/24} < 1/6$. □

A depth two perceptron that contains no AND gates with negated literals and no identical AND gates is said to be in *positive normal form* by Minsky and Papert (1988). Beigel called this *clean form*, and he formulated the following lemma, which uses the construction from (Minsky and Papert, 1988, page 33).

Lemma 5.7 (Beigel (1994, Lemma 1))**.** *If f is computed by a perceptron with size s, maximum weight w, and order d, then f is computed by a perceptron in clean form with size $2^d s$, weight sw, and order d.*

Lemma 5.8. *There are no depth two perceptrons with unit weights that compute g_2^m with bottom fan-in $m^{1/6}$ and less than $2^{5m^{1/6}}$ gates, for m larger than some constant.*

Proof. A lemma due to Beigel (1994, Lemma 5) gives the following relation between the size s, maximum weight w, and order d for depth two perceptrons on clean form computing ODD-MAX-BIT of N input bits:

$$w \geq \frac{1}{s} 2^{\left\lfloor \frac{N-1}{2\lceil d^2/(\sqrt{8}7-9)\rceil} \right\rfloor}.$$

Suppose there is a depth two perceptron with unit weights that computes g_2^m that has bottom fan-in (order) $m^{1/6}$ and less than $2^{5m^{1/6}}$ gates. Then, due to

Lemma 5.7, there is a perceptron in clean form with fan-in $m^{1/6}$, size $2^{6m^{1/6}}$ and weights bounded by $2^{5m^{1/6}}$ computing ODD-MAX-BIT of $\frac{1}{6}\sqrt{m \log m}$ variables.

If we put these values into Beigel's formula above, we get a contradiction for large enough m. □

Theorem 5.9. *For $k \geq 3$, there are no depth k perceptrons computing g_k^m with unit weights, bottom fan-in $\frac{1}{\sqrt{k}}m^{1/6}$, and less than $2^{6\frac{1}{\sqrt{k}}m^{1/6}}$ gates, not counting the gates on the lowest level, for m larger than some constant.*

When $k = 2$ we have a somewhat stronger result. The bound for the number of gates holds when counting all gates in the circuit.

Proof. We may assume that $m \geq k^3$, since otherwise we have $2^{6\frac{1}{\sqrt{k}}m^{1/6}} \leq 2^6$. The basis for the induction, $k = 2$, follows from Lemma 5.8.

Assume there is a depth-k perceptron P with bottom fan-in $\frac{1}{\sqrt{k}}m^{1/6}$ that has less than $2^{6\frac{1}{\sqrt{k}}m^{1/6}}$ gates computing g_k^m.

To be able to apply Lemma 5.6, choose one of $R_{q,B}^+$ or $R_{q,B}^-$ and apply a random restriction to the perceptron. We have that with probability at least $2/3$, the perceptron P computes a function at least as hard as g_{k-1}^m.

The bottom fan-in is at most $t = \frac{1}{\sqrt{k}}m^{1/6}$, and to use the induction hypothesis we have to obtain a circuit with fan-in at most $s = \frac{1}{\sqrt{(k-1)}}m^{1/6}$. For each single gate on the next to the lowest level, the probability that this fails is at most $(6qt)^s$ due to Lemma 4.4, and we multiply this by the maximum number of switchings to get a bound for the probability that the conversion fails for at least one circuit:

$$\begin{aligned} 2^{6\frac{1}{\sqrt{k}}m^{1/6}}(6qt)^s &= 64^{\frac{1}{\sqrt{k}}m^{1/6}}\left(6(\tfrac{2k}{m}\log m)^{1/2}\tfrac{1}{\sqrt{k}}m^{1/6}\right)^{\frac{1}{\sqrt{(k-1)}}m^{1/6}} \\ &\leq \left(6 \cdot 64(\tfrac{2k}{m}\log m)^{1/2}\tfrac{1}{\sqrt{k}}m^{1/6}\right)^{\frac{1}{\sqrt{(k-1)}}m^{1/6}} \\ &\leq 1/2, \end{aligned}$$

where the last inequality holds since $m \geq k^3$.

Going from depth 3 to depth 2 is a special case, since we need to bound the number of gates in the entire depth 2 circuit. Suppose without loss of generality that the lowest level of the depth 3 perceptron consists of OR gates (if it consists of AND gates we can negate the perceptron's weights and use De Morgan's law). Then, the depth 2 perceptron that results after switching has AND gates on the lowest level, and the fan-in of these gates is bounded by s so that each of them accepts at least a fraction 2^{-s} of the inputs. We know from Lemma 4.4 that all the AND gates accept disjoint inputs so we get that the maximum number of

AND gates that results from each switching is bounded by 2^s. Thus, the total number of gates in the resulting depth 2 circuit is at most

$$2^s 2^{6\frac{1}{\sqrt{k}}m^{1/6}} = 2^{\frac{1}{\sqrt{2}}m^{1/6}} 2^{\frac{6}{\sqrt{3}}m^{1/6}} < 2^{\frac{6}{\sqrt{2}}m^{1/6}}.$$

□

Theorem 5.10. *There exists an oracle A such that, for all d, there is a language $L_d(A)$ in $\Delta_d^{p,A}$ which is not recognizable by any $PP^{\Sigma_{d-2}^{p,A}}$ machine, i.e., $\Delta_d^{p,A} \not\subseteq PP^{\Sigma_{d-2}^{p,A}}$.*

Proof. The proof is analogous to the proof of Theorem 5.2, but we change the language to one that can be decided by a $\Delta_d^{p,A}$ machine.

First, let

$$L'_d(B) = \{y \mid \forall x_1, x_2, \ldots, x_{|y|} \exists x_{|y|+1}, \ldots, x_{2|y|} \cdots Q x_{(d-1)|y|+1}, \ldots, x_{d|y|} \colon yx \in B\},$$

where $x_1, x_2, \ldots$ denote the individual bits of x. This language is reminiscent of the language we used in the proof of Theorem 5.2, but it may contain many strings of the same length.

We then define

$$L_d(B) = \{1^n \mid \max(L'_{d-2}(B) \cap \{0,1\}^{n/(d-1)}) \text{ ends in a } 1\}.$$

The idea is that deciding if the string 1^n is in $L_d(B)$ is the same as computing the ODD-MAX-BIT function of $2^{n/(d-1)}$ inputs, where the inputs are the characteristic function of the language $L'_{d-2}(B)$ for inputs of length $n/(d-1)$.

Notice that a deterministic Turing machine using an NP machine as an oracle can compute the ODD-MAX-BIT function of 2^n input variables in polynomial time by doing binary search for the index of the variable with the highest index being 1. Since a $\Sigma_{d-2}^{p,B}$ oracle can decide the language $L'_{d-2}(B)$, we thus have that a P^{NP} machine using a $\Sigma_{d-2}^{p,B}$ machine as an oracle can decide the language $L_d(B)$, and therefore, the language $L_d(B)$ is in $\Delta_d^{p,B}$.

The existence of a $\mathrm{PP}^{\Sigma_{d-2}^{p,B}}$ machine that decides if $1^n \in L_d(B)$ implies a perceptron of depth d that evaluates a function h_d^n in the variables y_z^B for $|z| = n$. The language construction ensures that the function h_d^n is at least as hard to compute as $g_d^{2^{n/(d-1)}}$ (the defining circuit for $g_d^{2^{n/(d-1)}}$ has smaller fan-in at some levels).

Due to Theorem 5.9, however, the perceptron corresponding to a $\mathrm{PP}^{\Sigma_{d-2}^{p,A}}$ machine deciding the language is too small for computing the function g_d^m of the oracle bits for some m.

The construction of an oracle B such that $L_d(B)$ can not be decided by a $\mathrm{PP}^{\Sigma_{d-2}^{p,B}}$ machine for any d is now as in the proof of Theorem 5.2, the only difference being that n_i is chosen such that

$$\frac{6}{\sqrt{d}}\left(2^{n_i/(d-1)}\right)^{1/6} > cn_i^c$$

in each round. □

Chapter 6

On lower bounds for selecting the median

6.1 Introduction

Comparison based algorithms for solving median selection work by performing pairwise comparisons between the elements until the ith largest element among the n input elements is found. The problem of finding the median is the special case of selecting the ith largest in an ordered set of n elements, when $i = \lceil n/2 \rceil$.

Bent and John (1985) proved that median selection requires $2n + o(n)$ comparisons in the worst case. This proof was substantially shorter and more elegant than the proofs of the best lower bounds known before.

Our methods are based on the proof by Bent and John; Section 6.2 contains a reformulation of their proof where we assign weights to each node in the decision tree of the algorithm we want to prove makes many decisions. The weight of a node v corresponds to the total number of leaves in subtrees with root v in all pruned trees where v occurs in the proof by Bent and John. The weight of the root is approximately 2^{2n}; we show that every node v in the decision tree has a child whose weight is at least half the weight of v, and that the weights of all the leaves are small.

When the proof is formulated in this way, it becomes more transparent, and one can more easily study individual comparisons, to rule out some as being bad from the algorithm's point of view.

In Section 6.3 we use our new approach to prove that any pair-forming algorithm uses at least $2.01227n + o(n)$ comparisons to find the median.

Dor and Zwick (1996) have recently been able to extend the ideas described here to obtain a $(2+\epsilon)n$ lower bound, for some tiny $\epsilon > 0$, on the number of comparisons performed, in the worst case, by any median selection algorithm.

6.2 Bent and John revisited

Bent and John (1985) proved that $2n+o(n)$ comparisons are required for selecting the median. Their result is, in fact, more general and provides a lower bound for the number of comparisons required for selecting the ith largest element, for any $1 \leq i \leq n$. We concentrate here on median selection although our results, like those of Bent and John, can be extended to general i.

Although the proof given by Bent and John is relatively short and simple, we here present a reformulation. There are two reasons for this: the first is that the proof gets more transparent; the second is that this formulation makes it easier to study the effect of individual comparisons.

Theorem 6.1 (Bent and John (1985))**.** *Finding the median requires* $2n + o(n)$ *comparisons.*

Proof. Any deterministic algorithm for finding the median can be represented by a decision tree T, in which each internal node v is labeled by a comparison $a : b$. The two children of such a node, $v_{a<b}$ and $v_{a>b}$, represent the outcomes $a < b$ and $a > b$, respectively. We assume that decision trees do not contain redundant comparisons between elements whose relative order has already been established.

We consider a universe U containing n elements. For every node v in T and subset C of U we make the following definitions:

$$\max_v(C) = \left\{ a \in C \;\middle|\; \begin{array}{l} \text{every comparison } a : b \text{ above } v \\ \text{with } b \in C \text{ had outcome } a > b \end{array} \right\},$$

$$\min_v(C) = \left\{ a \in C \;\middle|\; \begin{array}{l} \text{every comparison } a : b \text{ above } v \\ \text{with } b \in C \text{ had outcome } a < b \end{array} \right\}.$$

Before we proceed with the proof that selecting the median requires $2n + o(n)$ comparisons, we present a proof of a somewhat weaker result. We assume that U contains $n = 2m$ elements and show that selecting the two middlemost elements requires $2n + o(n)$ comparisons. The proof in this case is slightly simpler, yet it demonstrates the main ideas used in the proof of the theorem.

We define a weight function on the nodes of T. This weight function satisfies the following three properties: (i) the weight of the root is $2^{2n+o(n)}$. (ii) each internal node v has a child whose weight is at least half the weight of v. (iii) the weight of each leaf is small.

For every node v in the decision tree, we keep track of subsets A of size m which may contain the m largest elements with respect to the comparisons already made. Let $\mathcal{A}(v)$ contain all such sets which are called *upper half compatible* with v. The As are assigned weights which estimate how far from a solution the

case		$w^1_{v_{a<b}}(A)$	$w^1_{v_{a>b}}(A)$
$a \in A$	$b \in A$	$\frac{1}{2}$ or 1	$\frac{1}{2}$ or 1
$a \in A$	$b \in \bar{A}$	0	1
$a \in \bar{A}$	$b \in A$	1	0
$a \in \bar{A}$	$b \in \bar{A}$	$\frac{1}{2}$ or 1	$\frac{1}{2}$ or 1

Table 6.1. The weight of a set $A \in \mathcal{A}(v)$ in the children of a node v, relative to its weight in v.

algorithm is, assuming that the elements in A are the m largest. The weight of every $A \in \mathcal{A}(v)$ is defined as

$$w^1_v(A) = 2^{|\min_v(A)|+|\max_v(\bar{A})|},$$

and the weight of a node v is defined as

$$w(v) = \sum_{A \in \mathcal{A}(v)} w^1_v(A).$$

The superscript 1 in $w^1_v(A)$ is used as we shall shortly have to define a second weight function $w^2_v(B)$.

In the root r of T, all subsets of size m of U are upper half compatible with r so that $|\mathcal{A}(r)| = \binom{2m}{m}$. Also, each $A \in \mathcal{A}(r)$ has weight 2^{2m}, and we find, as promised, that

$$w(r) = 2^{2m}\binom{2m}{m} = 2^{2n+o(n)}.$$

Consider the weight $w^1_v(A)$ of a set $A \in \mathcal{A}(v)$ at a node v labeled by the comparison $a : b$. What are the weights of A in v's children? This depends on which of the elements a and b belongs to A (and on which of them is minimal in A or maximal in $\bar{A}$). The four possible cases are considered in Table 6.1. The weights given there are relative to the weight $w^1_v(A)$ of A at v. A zero indicates that A is no longer compatible with this child and thus does not contribute to its weight. The weight $w^1_{v_{a<b}}(A)$, when $a, b \in A$, for example, is $\frac{1}{2}w^1_v(A)$, if $b \in \min_v(A)$, and is $w^1_v(A)$, otherwise. As can be seen, v always has at least one child in which the weight of A is at least half its weight at v. Furthermore, in each one of the four cases, $w^1_{v_{a<b}}(A) + w^1_{v_{a>b}}(A) \geq w^1_v(A)$.

Each leaf v of the decision tree corresponds to a state of the algorithm in which the two middlemost elements were found. There is therefore only one set A left in $\mathcal{A}(v)$. Since we have identified the minimum element in A and the maximum element in $\bar{A}$, we get that $w^1_v(A) = 4$. So, if we follow a path from the root of the tree and repeatedly descend to the child with the largest weight,

we will, when we eventually reach a leaf, have performed at least $2n + o(n)$ comparisons.

We now prove that selecting the median also requires at least $2n + o(n)$ comparisons. To make the median well defined we assume that $n = 2m - 1$. The problem that arises in the above argument is that the weights of the leaves in T, when the selection of the median, and not the two middlemost elements, is considered, are not necessarily small enough: it is possible to know the median without knowing any relations between elements in $\bar{A}$ (which now contains $m-1$ elements); this is remedied as follows.

In a node v where the algorithm is close to determining the minimum element in A, we essentially force it to determine the largest element in $\bar{A}$ instead. This is done by moving an element a_0 out of A and creating a set $B = \bar{A} \cup \{a_0\}$. This set is *lower half compatible* with v and the median is the maximum element in B. By a suitable choice of a_0, most of $\max_v(\bar{A})$ is in $\max_v(B)$. A set B is lower half compatible with v if $|B| = m$ and it may contain the m smallest elements in U. We keep track of Bs in the *multiset* $\mathcal{B}(v)$.

For the root r of T, we let $\mathcal{A}(r)$ contain all subsets of size m of U as before, and let $\mathcal{B}(r)$ be empty. We exchange some As for Bs as the algorithm proceeds. The weight of a set B is defined as

$$w_v^2(B) = 2^{|\max_v(B)|}.$$

The weight of B estimates how far the algorithm is from a solution, assuming that the elements in B are the m smallest elements. The weight of a node v is now defined to be

$$w(v) = \sum_{A \in \mathcal{A}(v)} w_v^1(A) + 2^{4\sqrt{n}} \sum_{B \in \mathcal{B}(v)} w_v^2(B).$$

In the beginning of an algorithm (in the upper part of the decision tree), the weight of a node is still the sum of the weights of all As, and therefore $w(r) = 2^{2n+o(n)}$.

We now define $\mathcal{A}(v)$ and $\mathcal{B}(v)$ for the rest of T more exactly. For any node v in T, except the root, simply copy $\mathcal{A}(v)$ and $\mathcal{B}(v)$ from the parent node and remove all sets that are not upper or lower half compatible with v, respectively. We ensure that the weight of every leaf is small by doing the following: If, for some $A \in \mathcal{A}(v)$ we have $|\min_v(A)| = \lceil 2\sqrt{n} \rceil$, we select an element $a_0 \in \min_v(A)$ which has been compared to the fewest number of elements in $\bar{A}$; we then remove the set A from $\mathcal{A}(v)$ and add the set $B = \bar{A} \cup \{a_0\}$ to $\mathcal{B}(v)$.

Note that at the root, $|\min_r(A)| = m$ for all $A \in \mathcal{A}(r)$, and that this quantity decreases by at most one for each comparison until a leaf is reached. In a leaf v the median is known; thus, $\mathcal{A}(v)$ is empty.

Lemma 6.2. *Let $\mathcal{A}(v)$ and $\mathcal{B}(v)$ be defined by the rules described above. Then, every internal node v (labeled $a : b$) in T has a child with at least half the weight of v, i.e., $w(v_{a<b}) \geq w(v)/2$ or $w(v_{a>b}) \geq w(v)/2$.*

case		$w^2_{v_{a<b}}(B)$	$w^2_{v_{a>b}}(B)$
$a \in B$	$b \in B$	$\frac{1}{2}$ or 1	$\frac{1}{2}$ or 1
$a \in B$	$b \in \bar{B}$	1	0
$a \in \bar{B}$	$b \in B$	0	1
$a \in \bar{B}$	$b \in \bar{B}$	1	1

Table 6.2. The weight of a set $B \in \mathcal{B}(v)$ in the children of a node v, relative to its weight in v.

Proof. Table 6.1 gives the weights of a set $A \in \mathcal{A}(v)$ at v's children, relative to the weight $w^1_v(A)$ of A at v. Similarly, Table 6.2 gives the weights of a set $B \in \mathcal{B}(v)$ in v's children, relative to the weight $w^2_v(v)$ of B at v. As $w^1_{v_{a<b}}(A)+w^1_{v_{a>b}}(A) \geq w^1_v(A)$ and $w^2_{v_{a<b}}(B) + w^2_{v_{a>b}}(B) \geq w^2_v(B)$, for every $A \in \mathcal{A}(v)$ and $B \in \mathcal{B}(v)$, all that remains to be checked is that the weight does not decrease when a lower half compatible set B replaces an upper half compatible set A. This is covered by Lemma 6.3. □

Lemma 6.3. *If A is removed from $\mathcal{A}(v)$ and B is added in its place to $\mathcal{B}(v)$, and if fewer than $4n$ comparisons have been performed on the path from the root to v, then $2^{4\sqrt{n}}w^2_v(B) > w^1_v(A)$.*

Proof. A set $A \in \mathcal{A}(v)$ is replaced by a set $B = \bar{A} \cup \{a_0\} \in \mathcal{B}(v)$ only when $|\min_v(A)| = \lceil 2\sqrt{n} \rceil$. The element a_0, in such a case, is an element of $\min_v(A)$ that has been compared to the fewest number of elements in $\bar{A}$. If a_0 was compared to at least $2\sqrt{n}$ elements in $\bar{A}$, we get that each element of $\min_v(A)$ was compared to at least $2\sqrt{n}$ elements in $\bar{A}$, and at least $4n$ comparisons have been performed on the path from the root to v, a contradiction. We get therefore that a_0 was compared to fewer than $2\sqrt{n}$ elements of $\bar{A}$ and thus $|\max_v(B)| > |\max_v(\bar{A})|-2\sqrt{n}$. As a consequence, we get that $4\sqrt{n}+|\max_v(B)| > |\min_v(A)|+|\max_v(\bar{A})|$ and thus $2^{4\sqrt{n}}w^2_v(B) > w^1_v(A)$, as required. □

We now know that the weight of the root is large, and that the weight does not decrease too fast; what remains to be shown is that the weights of the leaves are relatively small. This is established in the following lemma.

Lemma 6.4. *For a leaf v (in which the median is known), $w(v) \leq 2m2^{4\sqrt{n}}$.*

Proof. Clearly, the only sets compatible with a leaf of T are the set A containing the m largest elements, and the set B containing the m smallest elements. Since $|\min_v(A)| = |\max_v(B)| = 1$, we get that $w^2_v(B) = 2$ and $A \notin \mathcal{A}(v)$.

Since there are exactly m elements that can be removed from B to obtain a corresponding $\bar{A}$, there can be at most m copies of B in $\mathcal{B}(v)$. □

Let T be a comparison tree that corresponds to a median finding algorithm. If the height of T is at least $4n$, we are done. Otherwise, by starting at the root and repeatedly descending to a child whose weight is at least half the weight of its parent, we trace a path whose length is at least $2n + o(n)$ and Theorem 6.1 follows. □

Let us see how the current formalism gives room for improvement that did not exist in the original proof. The $2n+o(n)$ lower bound is obtained by showing that each node v in a decision tree T that corresponds to a median finding algorithm has a child whose weight is at least half the weight of v. Consider the nodes $v_0, v_1, \ldots, v_\ell$ along the path obtained by starting at the root of T and repeatedly descending to the child with the larger weight, until a leaf is reached. If we could show that sufficiently many nodes on this path have weights strictly larger than half the weights of their parents, we would obtain an improved lower bound for median selection. If $w(v_i) \geq \frac{1}{2}(1+\delta_i)w(v_{i-1})$, for every $1 \leq i \leq \ell$, then the length of this path, and therefore the depth of T, is at least $2n+\sum_{i=1}^{\ell} \log_2(1+\delta_i)+o(n)$.

6.3 An improved lower bound for pair-forming algorithms

Let v be a node of a comparison tree. An element x is a *singleton* at v if it was not compared above v with any other element. Two elements x and y form a *pair* at v if the elements x and y were compared to each other above v, but neither of them was compared to any other element.

A pair-forming algorithm is an algorithm that starts by constructing $\lfloor n/2 \rfloor = m - 1$ pairs. By concentrating on comparisons that involve elements that are part of pairs, we obtain a better lower bound for pair-forming algorithms.

Theorem 6.5. *A pair-forming algorithm for finding the median must perform, in the worst case, at least* $2.00691n + o(n)$ *comparisons.*

Proof. It is easy to see that a comparison involving two singletons can be delayed until just before one of them is to be compared for the second time. We can therefore restrict our attention to comparison trees in which the partial order corresponding to each node contains at most two pairs. Allowing only one pair is not enough as algorithms should be allowed to construct two pairs $\{a, b\}$ and $\{a', b'\}$, and then compare an element from $\{a, b\}$ with an element from $\{a', b'\}$.

We focus our attention on nodes in the decision tree in which an element of a pair is compared for the second time and in which the number of non-singletons is at most ϵm, for some $\epsilon < 1$. If v is a node in which the number of non-singletons is at most ϵm, for some $\epsilon < 1$, then $\mathcal{B}(v)$ is empty and thus $w(v) = \sum_{A \in \mathcal{A}(v)} w_v^1(A)$ and we do not have to consider Table 6.2 for the rest of the section.

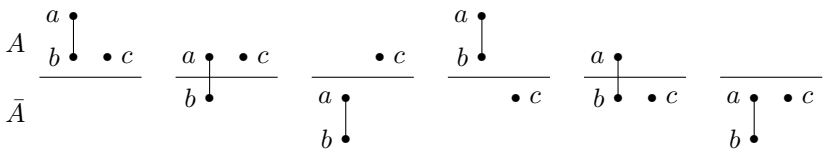

Figure 6.1. The six possible ways that a, b, and c may be divided between A and $\bar{A}$. Note that c is not necessarily a singleton element; it may be part of a larger partial order.

Recall that $\mathcal{A}(v)$ denotes the collection of subsets of U size m that are upper half compatible with v. If $H, L \subseteq U$ are subsets of U, of arbitrary size, we let

$$\mathcal{A}_{H/L}(v) = \{A \in \mathcal{A}(v) \mid H \subseteq A \text{ and } L \subseteq \bar{A}\}.$$

We let $w_{H/L}(v)$ be the contribution of the sets of $\mathcal{A}_{H/L}(v)$ to the weight of v, i.e.,

$$w_{H/L}(v) = \sum_{A \in \mathcal{A}_{H/L}(v)} w_v^1(A).$$

We write $\mathcal{A}_{h_1 \ldots h_r/l_1 \ldots l_s}(v)$ for $\mathcal{A}_{\{h_1,\ldots,h_r\}/\{l_1,\ldots,l_s\}}(v)$ and $w_{h_1 \ldots h_r/l_1 \ldots l_s}(v)$ for $w_{\{h_1,\ldots,h_r\}/\{l_1,\ldots,l_s\}}(v)$.

Before proceeding, we describe the intuition that lies behind the rest of the proof. Consider Table 6.1 from the last section. If, in a node v of the decision tree, the two cases $a \in A$, $b \in \bar{A}$ and $a \in \bar{A}$, $b \in A$ are not equally likely, or more precisely, if the contributions $w_{a/b}(v)$ and $w_{b/a}(v)$ of these two cases to the total weight of v are not equal, there must be at least one child of v whose weight is greater than half the weight of v. The difficulty in improving the lower bound of Bent and John lies therefore at nodes in which the the contributions of the two cases $a \in A$, $b \in \bar{A}$ and $a \in \bar{A}$, $b \in A$ are almost equal. This fact is not so easily seen when looking at the original proof given in (Bent and John, 1985).

Suppose now that v is a node in which an element a of a pair $\{a, b\}$ is compared with an arbitrary element c and that the number of non-singletons in v is at most ϵm. We assume, without loss of generality, that $a > b$. The weights of a set $A \in \mathcal{A}(v)$ in v's children depend on which of the elements a, b, and c belongs to A, and on whether c is minimal in A or maximal in $\bar{A}$. The six possible ways of dividing the elements a, b, and c between A and $\bar{A}$ are shown in Figure 6.1. The weights of the set A in v's children, relative to the weight $w_v^1(A)$ of A at v, in each one of these six cases are given in Table 6.3. Table 6.3 is similar to Table 6.1 of the previous section, with c playing the role of b. There is one important difference, however. If $a, b, c \in A$, as in the first row of Table 6.3, then the weight of A in $v_{a>c}$ is equal to the weight of A in v. The weight is not halved, as may be the case in the first row of Table 6.1. If the contribution $w_{abc/}(v)$ of the case $a, b, c \in A$ to the weight of v is not negligible, there must again be at least one child of v whose weight is greater than half the weight of v.

case			$w^1_{v_{a<c}}(A)$	$w^1_{v_{a>c}}(A)$
$a \in A$	$b \in A$	$c \in A$	$\frac{1}{2}$ or 1	1
$a \in A$	$b \in \bar{A}$	$c \in A$	$\frac{1}{2}$ or 1	$\frac{1}{2}$
$a \in \bar{A}$	$b \in \bar{A}$	$c \in A$	1	0
$a \in A$	$b \in A$	$c \in \bar{A}$	0	1
$a \in A$	$b \in \bar{A}$	$c \in \bar{A}$	0	1
$a \in \bar{A}$	$b \in \bar{A}$	$c \in \bar{A}$	$\frac{1}{2}$	$\frac{1}{2}$ or 1

Table 6.3. The weight of a set $A \in \mathcal{A}(v)$ in the children of a node v, relative to its weight in v, when the element a of a pair $a > b$ is compared with an arbitrary element c.

The improved lower bound is obtained by showing that if the contributions of the cases $a \in A$, $b \in \bar{A}$ and $a \in \bar{A}$, $b \in A$ are roughly equal, and if most elements in the partial order are singletons, then the contribution of the case $a, b, c \in A$ is non-negligible. The larger the number of singletons in the partial order, the larger is the relative contribution of the weight $w_{abc/}(v)$ to the weight $w(v)$ of v. Thus, whenever an element of a pair is compared for the second time, we make a small gain. The above intuition is made precise in the following lemma:

Lemma 6.6. *If v is a node in which an element a of a pair $a > b$ is compared with an element c, and if the number of singletons in v is at least $m + 2\sqrt{n}$, then*

$$\begin{aligned} w(v_{a<c}) &\geq \tfrac{1}{2}w(v) + \tfrac{1}{2}(w_{c/a}(v) - w_{a/c}(v)) \,, \\ w(v_{a>c}) &\geq \tfrac{1}{2}w(v) + \tfrac{1}{2}(w_{a/c}(v) - w_{c/a}(v) + w_{abc/}(v)) \,. \end{aligned}$$

Proof. Both inequalities follow easily by considering the entries in Table 6.3. To obtain the second inequality, for example, note that $w(v_{a>c}) \geq \frac{1}{2}(w(v) + w_{abc/}(v) - w_{c/ab}(v) + w_{ab/c}(v) + w_{a/bc}(v))$. As $w_{c/ab}(v) = w_{c/a}(v)$ and $w_{ab/c}(v) + w_{a/bc}(v) = w_{a/c}(v)$, the second inequality follows. □

It is worth pointing out that in Table 6.3 and in Lemma 6.6, we only need to assume that $a > b$; we do not use the stronger condition that $a > b$ is a pair. This stronger condition is crucial however in the sequel, especially in Lemma 6.8.

To make use of Lemma 6.6 we need bounds on the relative contributions of the different cases. The following lemma is a useful tool for determining such bounds.

Lemma 6.7. *Let $G = (V_1, V_2, E)$ be a bipartite graph. Let δ_1 and δ_2 be the minimal degree of the vertices of V_1 and V_2, respectively. Let Δ_1 and Δ_2 be the maximal degree of the vertices of V_1 and V_2, respectively. Assume that a positive*

weight function w is defined on the vertices of G such that $w(v_1) = r \cdot w(v_2)$, whenever $v_1 \in V_1$, $v_2 \in V_2$ and $(v_1, v_2) \in E$. Let $w(V_1) = \sum_{v_1 \in V_1} w(v_1)$ and $w(V_2) = \sum_{v_2 \in V_2} w(v_2)$. Then,

$$r\frac{\delta_2}{\Delta_1} \cdot w(V_2) \;\leq\; w(V_1) \;\leq\; r\frac{\Delta_2}{\delta_1} \cdot w(V_2).$$

Proof. Let $v_1(e) \in V_1$ and $v_2(e) \in V_2$ denote the two vertices connected by the edge e. We then have

$$\delta_1 \sum_{v_1 \in V_1} w(v_1) \leq \sum_{e \in E} w(v_1(e)) = r \sum_{e \in E} w(v_2(e)) \leq r\Delta_2 \sum_{v_2 \in V_2} w(v_2).$$

The other inequality follows by exchanging the roles of V_1 and V_2. □

Using Lemma 6.7 we obtain the following basic inequalities.

Lemma 6.8. *If v is a node in which $a > b$ is a pair and the number of non-singletons in v is at most ϵm, then*

$$\begin{array}{rcccl}
\frac{1}{2}(1-\epsilon)\cdot w_{ac/b}(v) & \leq & w_{abc/}(v) & \leq & \frac{1}{2(1-\epsilon)}\cdot w_{ac/b}(v) \;, \\
2(1-\epsilon)\cdot w_{c/ab}(v) & \leq & w_{ac/b}(v) & \leq & \frac{2}{1-\epsilon}\cdot w_{c/ab}(v) \;, \\
\frac{1}{2}(1-\epsilon)\cdot w_{a/bc}(v) & \leq & w_{ab/c}(v) & \leq & \frac{1}{2(1-\epsilon)}\cdot w_{a/bc}(v) \;, \\
2(1-\epsilon)\cdot w_{/abc}(v) & \leq & w_{a/bc}(v) & \leq & \frac{2}{1-\epsilon}\cdot w_{/abc}(v) \;.
\end{array}$$

Each one of these inequalities relates a weight, such as $w_{abc/}(v)$, to a weight, such as $w_{ac/b}(v)$, obtained by moving one of the elements of the pair $a > b$ from A to $\bar{A}$. In each inequality we 'lose' a factor of $1 - \epsilon$. When the elements a and b are joined together a factor of 2 is introduced. When the elements a and b are separated, a factor of $\frac{1}{2}$ is introduced.

Proof. We present a proof of the inequality $w_{abc/}(v) \leq \frac{1}{2(1-\epsilon)} \cdot w_{ac/b}(v)$. The proof of all the other inequalities is almost identical.

Construct a bipartite graph $G = (V_1, V_2, E)$ whose vertex sets are $V_1 = \mathcal{A}_{abc/}(v)$ and $V_2 = \mathcal{A}_{ac/b}(v)$. Define an edge $(A_1, A_2) \in E$ between $A_1 \in \mathcal{A}_{abc/}(v)$ and $A_2 \in \mathcal{A}_{ac/b}(v)$ if and only if there is a singleton $d \in \bar{A}_1$ such that $A_2 = A_1 \setminus \{b\} \cup \{d\}$. Suppose that (A_1, A_2) is such an edge. As $a \notin \min_v(A_1)$ but $a \in \min_v(A_2)$, while all other elements are extremal with respect to A_1 if and only if they are extremal with respect to A_2 (note that $b \in \min_v(A_1)$ and $b \in \max_v(\bar{A}_2)$), we get that $w^1_v(A_1) = \frac{1}{2} \cdot w^1_v(A_2)$.

For every set A of size m, the number of singletons in A is at least $(1-\epsilon)m$ and at most m. We get therefore that the minimal degrees of the vertices of V_1 and V_2 are $\delta_1, \delta_2 \geq (1-\epsilon)m$ and the maximal degrees of V_1 and V_2 are $\Delta_1, \Delta_2 \leq m$. The inequality $w_{abc/}(v) \leq \frac{1}{2(1-\epsilon)} \cdot w_{ac/b}(v)$ therefore follows from Lemma 6.7. □

Using these basic inequalities we obtain:

Lemma 6.9. *If v is a node in which $a > b$ is a pair and the number of non-singletons is at most ϵm, for some $\epsilon < 1$, then*

$$\begin{aligned} w_{abc/}(v) &\geq \tfrac{(1-\epsilon)^2}{(2-\epsilon)^2} \cdot w_{c/}(v) \ , \\ w_{a/c}(v) &\geq \tfrac{(1-\epsilon)(3-\epsilon)}{(2-\epsilon)^2} \cdot w_{/c}(v) \ , \\ w_{c/a}(v) &\leq \tfrac{1}{(2-\epsilon)^2} \cdot w_{c/}(v) \ . \end{aligned}$$

Proof. We present the proof of the first inequality. The proof of the other two inequalities is similar. Using inequalities from Lemma 6.8 we get that

$$\begin{aligned} w_{c/}(v) &= w_{abc/}(v) + w_{ac/b}(v) + w_{c/ab}(v) \\ &\leq w_{abc/}(v) + \tfrac{2}{1-\epsilon} \cdot w_{abc/}(v) + \tfrac{1}{(1-\epsilon)^2} \cdot w_{abc/}(v) \\ &= \tfrac{(2-\epsilon)^2}{(1-\epsilon)^2} \cdot w_{abc/}(v) \end{aligned}$$

and the first inequality follows. □

We are now ready to show that if v is a node in which an element of a pair is compared for the second time, then v has a child whose weight is greater than half the weight of v. Combining Lemma 6.6 and Lemma 6.9, we get that

$$\begin{aligned} \tfrac{1}{2} \cdot (w(v_{a<c}) + w(v_{a>c})) &\geq \tfrac{1}{2} \cdot w(v) + \tfrac{(1-\epsilon)^2}{4(2-\epsilon)^2} \cdot w_{c/}(v) \ , \\ w(v_{a>c}) &\geq \tfrac{1}{2} \cdot w(v) - \tfrac{\epsilon}{2(2-\epsilon)} \cdot w_{c/}(v) + \tfrac{(1-\epsilon)(3-\epsilon)}{2(2-\epsilon)^2} \cdot w_{/c}(v) \ . \end{aligned}$$

Let $\alpha = w_{c/}(v)/w(v)$ and $1 - \alpha = w_{/c}(v)/w(v)$. We get that

$$\begin{aligned} \tfrac{1}{2} \cdot (w(v_{a<c}) + w(v_{a>c})) &\geq \left(\tfrac{1}{2} + \tfrac{(1-\epsilon)^2}{4(2-\epsilon)^2} \alpha \right) \cdot w(v) \ , \\ w(v_{a>c}) &\geq \left(\tfrac{1}{2} - \tfrac{\epsilon}{2(2-\epsilon)} \alpha + \tfrac{(1-\epsilon)(3-\epsilon)}{2(2-\epsilon)^2} (1-\alpha) \right) \cdot w(v) \\ &= \left(\tfrac{1}{2} - \tfrac{3-2\epsilon}{2(2-\epsilon)^2} \alpha + \tfrac{(1-\epsilon)(3-\epsilon)}{2(2-\epsilon)^2} \right) \cdot w(v) \ . \end{aligned}$$

As a consequence, we get that

$$\max\{w(v_{a<c}), w(v_{a>c})\} \geq \max \left\{ \tfrac{1}{2} + \tfrac{(1-\epsilon)^2}{4(2-\epsilon)^2} \alpha, \tfrac{1}{2} - \tfrac{3-2\epsilon}{2(2-\epsilon)^2} \alpha + \tfrac{(1-\epsilon)(3-\epsilon)}{2(2-\epsilon)^2} \right\} \cdot w(v) \ .$$

The coefficient of $w(v)$, on the right hand side, is minimized when the two expressions whose maximum is taken are equal. This happens when $\alpha = \frac{2(3-4\epsilon+\epsilon^2)}{7-6\epsilon+\epsilon^2}$. Plugging this value of α into the two expressions, we get that

$$\max\{w(v_{a<c}), w(v_{a>c})\} \geq \tfrac{1}{2}(1 + f_1(\epsilon)) \cdot w(v) \ ,$$

where

$$f_1(\epsilon) = \frac{(3-\epsilon)(1-\epsilon)^3}{(2-\epsilon)^2(7-6\epsilon+\epsilon^2)} \ .$$

It is easy to check that $f_1(\epsilon) > 0$ for $\epsilon < 1$.

A *pair-forming* comparison is a comparison in which two singletons are compared to form a pair. A *pair-touching* comparison is a comparison in which an element of a pair is compared for the second time. In a pair-forming algorithm, the number of singletons is decreased only by pair-forming comparisons. Each pair-forming comparison decreases the number of singletons by exactly two. As explained above, pair-forming comparisons can always be delayed so that a pair-forming comparison $a : b$ is immediately followed by a comparison that touches the pair $\{a, b\}$, or by a pair-forming comparison $a' : b'$ and then by a comparison that touches both pairs $\{a, b\}$ and $\{a', b'\}$.

Consider again the path traced from the root by repeatedly descending to the child with the larger weight. As a consequence of the above discussion, we get that when the ith pair-touching comparison along this path is performed, the number of non-singletons in the partial order is at most $4i$. It follows therefore from the remark made at the end of the previous section that the depth of the comparison tree corresponding to any pair-forming algorithm is at least

$$\begin{aligned} & 2n + \sum_{i=1}^{m/4} \log_2(1 + f_1(\frac{4i}{m})) + o(n) \\ = \quad & 2n + \frac{n}{8} \cdot \int_0^1 \log_2(1 + f_1(t))dt + o(n) \;\approx\; 2.00691n + o(n) \; . \end{aligned}$$

This completes the proof of Theorem 6.5. □

The worst case in the proof above is obtained when the algorithm converts all the elements into *quartets*. A quartet is a partial order obtained by comparing elements contained in two disjoint pairs. In the proof above, we analyzed cases in which an element a of a pair $a > b$ is compared with an arbitrary element c. If the element c is also part of a pair, a tighter analysis is possible. By performing this analysis we can improve Theorem 6.5.

Theorem 6.10. *A pair-forming algorithm for finding the median must perform, in the worst case, at least* $2.01227n + o(n)$ *comparisons.*

Proof. Consider comparisons in which the element from a pair $a > b$ is compared with an element of a pair $c > d$. The nine possible ways of dividing the elements a, b, c, and d among A and $\bar{A}$ are depicted in Figure 6.2. Because of symmetry we may assume, without loss of generality, that the element a is compared with either c or with d.

Let v be a node of the comparison tree in which $a > b$ and $c > d$ are pairs and which one of the comparisons $a : c$ or $a : d$ is performed. Let $A \in \mathcal{A}(v)$. The weights of a set A in v's children, relative to the weight $w_v^1(A)$ of A at v, in each one of these nine cases are given in Table 6.4. The two possible comparisons $a : c$ and $a : d$ are considered separately. The following equalities are easily verified.

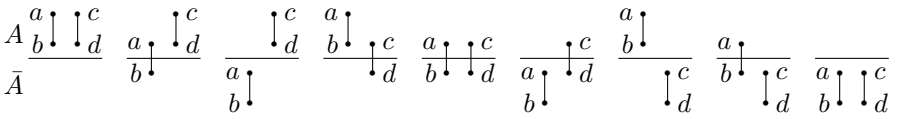

Figure 6.2. The nine possible ways that a, b, c, and d may be divided between A and $\bar{A}$.

case	$w^1_{v_{a<c}}(A)$	$w^1_{v_{a>c}}(A)$	$w^1_{v_{a<d}}(A)$	$w^1_{v_{a>d}}(A)$
$A \in \mathcal{A}_{abcd/}$	1	1	$\frac{1}{2}$	1
$A \in \mathcal{A}_{acd/b}$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$A \in \mathcal{A}_{cd/ab}$	1	0	1	0
$A \in \mathcal{A}_{abc/d}$	$\frac{1}{2}$	1	0	1
$A \in \mathcal{A}_{ac/bd}$	$\frac{1}{2}$	$\frac{1}{2}$	0	1
$A \in \mathcal{A}_{c/abd}$	1	0	$\frac{1}{2}$	$\frac{1}{2}$
$A \in \mathcal{A}_{ab/cd}$	0	1	0	1
$A \in \mathcal{A}_{a/bcd}$	0	1	0	1
$A \in \mathcal{A}_{/abcd}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1

Table 6.4. The weight of a set $A \in \mathcal{A}(v)$ in the children of a node v, relative to its weight in v, when the element a of a pair $a > b$ is compared with an element of a pair $c > d$.

Lemma 6.11. *If $a > b$ and $c > d$ are pairs in v then*

$$\begin{aligned} w_{acd/b}(v) &= w_{abc/d}(v)\,, \\ w_{cd/ab}(v) &= w_{ab/cd}(v)\,, \\ w_{c/abd}(v) &= w_{a/bcd}(v)\,, \\ w_{ac/bd}(v) &= 4{\cdot}w_{ab/cd}(v)\,. \end{aligned}$$

The following inequalities are analogous to the inequalities of Lemma 6.8.

Lemma 6.12. *If $a > b$ and $c > d$ are pairs in v and if the number of non-singletons in v is at most ϵm, for some $\epsilon < 1$, then*

$$\begin{aligned} \tfrac{1}{2}(1-\epsilon)w_{abc/d}(v) &\le w_{abcd/}(v) \le \tfrac{1}{2(1-\epsilon)}w_{abc/d}(v)\,, \\ 2(1-\epsilon)w_{ab/cd}(v) &\le w_{abc/d}(v) \le \tfrac{2}{1-\epsilon}w_{ab/cd}(v)\,, \\ \tfrac{1}{2}(1-\epsilon)w_{a/bcd}(v) &\le w_{ab/cd}(v) \le \tfrac{1}{2(1-\epsilon)}w_{a/bcd}(v)\,, \\ 2(1-\epsilon)w_{/abcd}(v) &\le w_{a/bcd}(v) \le \tfrac{2}{1-\epsilon}w_{/abcd}(v)\,. \end{aligned}$$

Consider first the comparison $a : c$. By examining Table 6.4 and using the equalities of Lemma 6.11, we get that

$$\begin{aligned}
&\tfrac{w(v_{a<c})+w(v_{a>c})}{2} \\
&\quad = w_{abcd/}(v) + \frac{3}{4}w_{acd/b}(v) + \frac{1}{2}w_{cd/ab}(v) + \frac{3}{4}w_{abc/d}(v) \\
&\qquad + \tfrac{1}{2}w_{ac/bd}(v) + \tfrac{1}{2}w_{c/abd}(v) + \tfrac{1}{2}w_{ab/cd}(v) + \tfrac{1}{2}w_{a/bcd}(v) + \tfrac{1}{2}w_{/abcd}(v) \\
&\quad = w_{abcd/}(v) + \tfrac{3}{2}w_{abc/d}(v) + 3w_{ab/cd}(v) + w_{a/bcd}(v) + \tfrac{1}{2}w_{/abcd}(v).
\end{aligned}$$

Minimizing this expression, subject to the equalities of Lemma 6.11, the inequalities of Lemma 6.12, and the fact that the weights of the nine cases sum up to $w(v)$, amounts to solving a linear program. By solving this linear program we get that

$$\frac{w(v_{a<c}) + w(v_{a>c})}{2w(v)} \geq \frac{1}{2}(1 + f_2(\epsilon)) \cdot w(v) \ ,$$

where

$$f_2(\epsilon) = \frac{(3-\epsilon)(1-\epsilon)^3}{(2-\epsilon)^4} \ .$$

It seems intuitively clear that the comparison $a : d$ is a bad comparison from the algorithm's point of view. The adversary will most likely answer with $a > d$. Indeed, by solving the corresponding linear program, we get that

$$\begin{aligned}
w(v_{a>d}) &= w_{abcd/}(v) + \tfrac{1}{2}w_{acd/b}(v) + w_{abc/d}(v) + w_{ac/bd}(v) \\
&\quad + \tfrac{1}{2}w_{c/abd}(v) + w_{ab/cd}(v) + w_{a/bcd}(v) + w_{/abcd}(v) \\
&= w_{abcd/}(v) + \tfrac{3}{2}w_{abc/d}(v) + 5w_{ab/cd}(v) + \tfrac{3}{2}w_{a/bcd}(v) + w_{/abcd}(v) \\
&\geq \tfrac{3}{4}w(v).
\end{aligned}$$

As $\frac{1}{2}(1 + f_2(\epsilon)) \leq \frac{3}{4}$, for every $0 \leq \epsilon \leq 1$, we may disregard the comparison $a : d$ from any further consideration.

It is easy to verify that $(1 + f_1(\epsilon))^2 \geq 1 + f_2(\epsilon)$ which means that the bad case (from a lower bound point of view) is when the algorithm uses two pairs by comparing them to each other. As a result, we get a lower bound of

$$2n + \frac{n}{8} \cdot \int_0^1 \log_2(1 + f_2(t))dt + o(n) \ \approx \ 2.01227n + o(n) \ .$$

This completes the proof of Theorem 6.10. □

6.4 Concluding remarks

We presented a reformulation of the $2n + o(n)$ lower bound of Bent and John for the number of comparisons needed for selecting the median of n elements. Using this new formulation we obtained an improved lower bound for pair-forming

median finding algorithms. As mentioned, Dor and Zwick (1996) have recently extended the ideas described here to obtain a $(2+\epsilon)n$ lower bound for general median finding algorithms, for some tiny $\epsilon > 0$.

We believe that the lower bound for pair-forming algorithms obtained here can be substantially improved. Such an improvement seems to require, however, some new ideas. Obtaining an improved lower bound for pair-forming algorithms may be an important step towards obtaining a lower bound for general algorithms which is significantly better than the lower bound of Bent and John (1985).

Paterson (1996) conjectures that the number of comparisons required for selecting the median is about $(\log_{4/3} 2) \cdot n \approx 2.41n$. His conjecture is based on considering the number of partitions of the elements into two subsets A and B, where $|A| = \lceil n/2 \rceil$ and $|B| = \lfloor n/2 \rfloor$, such that the elements in A may all be larger than the elements in B with respect to the comparisons made. Before an algorithm makes any comparisons, there are $\binom{n}{\lfloor n/2 \rfloor}$ such partitions. The first comparison brings down the number of compatible partitions by a factor of $3/4$, and it seems reasonable that further comparisons can not, at average, do better than that. When the median is known, there is only one compatible partition left, so we get the equation $(3/4)^h \binom{n}{\lfloor n/2 \rfloor} \leq 1$ for the lower bound h.

Bibliography

Aigner, M. (1981). Producing posets. *Discrete Mathematics 35*, 1–15.

Alon, N. and R. B. Boppana (1987). The monotone circuit complexity of boolean functions. *Combinatorica 7*, 1–22.

Alon, N. and J. H. Spencer (1992). *The probabilistic method*. Wiley-Interscience Series in Discrete Mathematics and Optimization. New York: John Wiley & Sons Inc. With an appendix by Paul Erdős, A Wiley-Interscience Publication.

Amano, K. and A. Maruoka (1996). Potential of the approximation method. In *Proc. 37th Ann. IEEE Symp. Found. Comput. Sci.*, pp. 431–440.

Andreev, A. E. (1985). On a method for obtaining lower bounds for the complexity of individual monotone functions. *Soviet Mathematics Doklady 31*, 530–534.

Beigel, R. (1994). Perceptrons, PP, and the polynomial hierarchy. *Computational Complexity 4* (4), 339–349.

Beigel, R., L. A. Hemachandra, and G. Wechsung (1991). Probabilistic polynomial time is closed under parity reductions. *Information Processing Letters 37*, 91–94.

Beigel, R., N. Reingold, and D. Spielman (1995). PP is closed under intersection. *Journal of Computer and System Sciences 50* (2), 191–202.

Bent, S. W. and J. W. John (1985). Finding the median requires $2n$ comparisons. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pp. 213–216.

Berg, C. and S. Ulfberg (1998). A lower bound for perceptrons and an oracle separation of the $\mathrm{PP}^{\mathrm{ph}}$ hierarchy. *Journal of Computer and System Sciences 56*, 263–271.

Berg, C. and S. Ulfberg (1999). Symmetric approximation arguments for monotone lower bounds without sunflowers. *Computational complexity 8*, 1–20.

Blum, M., R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan (1973). Time bounds for selection. *Journal of Computer and System Sciences 7*, 448–461.

Boppana, R. B. and M. Sipser (1990). The complexity of finite functions. In J. van Leeuwen (Ed.), *Handbook of theoretical computer science*, Volume A, Algorithms and complexity, pp. 757–804. Elsevier/MIT Press.

Chen, J. (1993). *Partial Order Productions*. Ph. D. thesis, Lund University, Box 118, S-221 00 Lund, Sweden.

Dor, D., J. Håstad, S. Ulfberg, and U. Zwick (2000). On lower bounds for selecting the median. *SIAM Journal of Discrete Mathematics*. to appear.

Dor, D. and U. Zwick (1995). Selecting the median. In *Proceedings of 6th SODA*, pp. 88–97.

Dor, D. and U. Zwick (1996). Median selection requires $(2+\epsilon)n$ comparisons. In *Proceedings of 37th FOCS*.

Erdős, P. and R. Rado (1960). Intersection theorems for systems of sets. *J. London Math. Soc. 35*, 85–90.

Ford, L. R. and S. M. Johnson (1959). A tournament problem. *American Mathematical Monthly 66*, 387–389.

Fu, B. (1992). Separating PH from PP by relativization. *Acta Mathematica Sinica (New Series) 8*(3), 329–336.

Furst, M., J. B. Saxe, and M. Sipser (1984). Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory 17*, 13–27.

Fussenegger, F. and H. N. Gabow (1979, April). A counting approach to lower bounds for selection problems. *Journal of the Association for Computing Machinery 26*(2), 227–238.

Green, F. (1991). An oracle separating $\oplus$P from $\mathrm{PP}^{\mathrm{PH}}$. *Information Processing Letters 37*, 149–153.

Green, F. (1995). A lower bound for monotone perceptrons. *Mathematical Systems Theory 28*, 283–298.

Haken, A. (1995). Counting bottlenecks to show monotone P $\neq$ NP. In *Proc. 36th Ann. IEEE Symp. Found. Comput. Sci.*, pp. 36–40.

Haken, A. and S. Cook (1996). An exponential lower bound for the size of monotone real circuits. Submitted to J. Comput. System Sci.

Håstad, J. (1987). *Computational Limitations for Small-Depth Circuits*. ACM doctoral dissertation awards. Cambridge, Massachusetts: MIT Press.

Håstad, J. (1989). Almost optimal lower bounds for small depth circuits. In S. Micali (Ed.), *Randomness and Computation*, Volume 5 of *Advances in Computing Research*, pp. 143–170. JAI Press Inc.

Håstad, J. and M. Goldmann (1991). On the power of small-depth threshold circuits. *Computational Complexity 1*(2), 113–129.

Jukna, S. (1997). Finite limits and monotone computations over the reals. In *Twelfth Annual IEEE Conference on Computational Complexity*.

Jukna, S. (1999). Combinatorics of monotone computations. *Combinatorica 19*, 65–85.

Knuth, D. E. (1973). *The Art of Computer Programming, vol. 3, Searching and Sorting*. Addison-Wesley Publishing Company, Inc.

Ko, K.-I. (1989). Relativized polynomial time hierarchies having exactly k levels. *SIAM Journal of Computing 18*(2), 392–408.

Linial, N., Y. Mansour, and N. Nisan (1989). Constant depth circuits, Fourier transform, and learnability. In *30th Annual Symposium on Foundations of Computer Science*, pp. 574–579.

Minsky, M. and S. Papert (1988). *Perceptrons* (Expanded ed.). Cambridge, Massachusetts: MIT Press.

Munro, I. and P. Poblete (1982). A lower bound for determining the median. Technical report, Technical Report Research Report CS-82-21, University of Waterloo.

Paterson, M. S. (1996). Progress in selection. In *5th Scandinavian Workshop on Algorithm Theory, Reykjavík, Iceland*, pp. 368–379.

Pratt, V. R. and F. F. Yao (1973). On lower bounds for computing the i-th largest element. In *14th Annual Symposium on Switching and Automata Theory*, pp. 70–81.

Pudlák, P. (1997). Lower bounds for resolution and cutting planes proofs and monotone computations. *Journal of Symbolic Logic 62*, 981–998.

Razborov, A. A. (1985a). A lower bound on the monotone network complexity of the logical permanent. *Mathematical Notes of the Academy of Sciences of the USSR 37*, 485–493.

Razborov, A. A. (1985b). Lower bounds on the monotone complexity of some boolean functions. *Sov. Math. Dokl. 31*, 354–357.

Razborov, A. A. (1989). On the method of approximations. In *Proc. Twenty-first Ann. ACM Symp. Theor. Comput.*, pp. 167–176.

Schönhage, A., M. Paterson, and N. Pippenger (1976). Finding the median. *Journal of Computer and System Sciences 13*, 184–199.

Simon, J. and S.-C. Tsai (1997). A note on the bottleneck counting argument. In *Twelfth Annual IEEE Conference on Computational Complexity*.

Sipser, M. (1983). Borel sets and circuit complexity. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pp. 61–69.

Tiekenheinrich, J. (1984). A $4n$ lower bound on the monotone network complexity of a one-output boolean function. *Inform. Process. Lett. 18*, 201–202.

Yao, A. (1985). Separating the polynomial-time hierarchy by oracles. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science*, pp. 1–10.

Yap, C. (1976). New lower bounds for medians and related problems. Technical report, Computer Science Report 79, Yale University. Abstract in Symposium on Algorithms and Complexity: New Results and Directions, (J. F. Traub, ed.) Carnegie-Mellon University, 1976.

Index