New Circuit Lower and Upper Bounds via Combinatorial Arguments

A THESIS

submitted by

BALAGOPAL

for the award of the degree

of

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

DECEMBER 2015

THESIS CERTIFICATE

This is to certify that the thesis titled **New Circuit Lower and Upper Bounds via Combinatorial Arguments SUBMITTED TO IIT-M**, submitted by **Balagopal**, to the Indian Institute of Technology, Madras, for the award of the degree of **Doctor of Philosophy**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Jayalal Sarma
Research Guide
Assistant Professor
Department of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: xx December 2015

ACKNOWLEDGEMENTS

ABSTRACT

KEYWORDS: Computational Complexity, Circuit Complexity, Branching Programs, Monoid Programs, Pebbling Games, Comparator Circuits, Circuits over Posets and Lattices.

One of the central problems in complexity theory is the question whether languages decidable in logarithmic space (The class L) is the same as languages decidable in polynomial time (The class P). It is widely believed that they are distinct. One approach to prove this would be to consider a language in P, consider "natural" algorithms solving this problem and prove that

1. Natural algorithms are optimal.
2. Natural algorithms require super-logarithmic space.

Cook *et al.* (2012) proposed the Tree Evaluation Problem as a candidate problem in P that is not in the class L. They defined "thrifty" models of computation and show that deterministic thrifty models solving TEP require super-logarithmic space. However, showing that these models are optimal (or even close to optimal) seems hard. So the natural next step would be to consider stronger models of computation solving TEP and prove lower bounds on them. Working in this direction, we showed that a certain sub-class of non-deterministic thrifty programs also require super-logarithmic space to solve the problem. The obvious next step is to prove similar lower bounds for unrestricted non-deterministic thrifty programs. The lower bound proofs for these computation models also have the nice property that they work by showing a natural connection between computation performed by these models and a combinatorial game called the pebbling game defined on DAGs.

We also consider combinatorial accepts of the reversible pebbling game (A variant of the pebbling game used to prove lower bounds for TEP). Recently, Chan (2013) has showed a surprising equivalence between the reversible pebbling number (A one-player game) and the Dymond-Tompa pebble game number and Raz-Mckenzie pebble game number (both two-player games). We show that reversible pebbling number, restricted to trees, is equivalent to the edge-rank colouring number of the tree. This result has many interesting consequences.

1. The reversible pebbling number, even though it is defined on directed trees, depends only on the underlying undirected tree.
2. Reversible pebbling number (along with an optimal strategy for achieving this number) can be computed in linear time.

Constant depth Boolean circuits characterize problems that are highly parallelizable. We consider the problem of detecting graph properties such as existence of 2-colouring, perfect matching, and disjoint paths in constant-width grid graphs using constant depth circuits. We prove that all these problems can be solved using appropriate constant depth circuits. The proofs work by reducing these problems to monoid word problems and then uses the Barrington-Therien (Barrington and Thérien (1988)) framework to prove these upper bounds by showing that these monoids are solvable or aperiodic. Earlier, it was known that the reachability problem on these graphs can be solved using constant depth circuits. These results show that problems harder than the reachability problem in constant width grid graphs are also solvable using constant depth circuits.

We consider the comparator circuit model and the associated complexity class CC (See Mayr and Subramanian (1992), Cook *et al.* (2014)), that captures the complexity of many hard-to-parallelize problems such as the lex-least maximal matching problem and the stable marriage problem. Cook *et al.* (2014) asked the question whether there is a Turing Machine model that captures the class CC. We tackle the other direction and show that the comparator circuit model can capture other complexity classes such as L, NL, P, and NP. Of particular interest is our generalization of circuit models to work over arbitrary fixed lattices instead of the Boolean lattice with two elements – 0 and 1. We show that circuit models that exist in the literature such as Boolean circuits, skew circuits and formulae are robust under this generalization but the comparator circuit model seems to increase in power when working with non-distributive lattices. A very interesting question arising out of our work is whether all non-distributive lattices increase the power of comparator circuits.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

TM	Turing Machine
BP	Branching Programs
NC	Nick's Class
AC	Alternating Circuits
ACC	Alternating Circuits with Counters
NTBP	Non-deterministic Thrifty Branching Programs
RONTBP	Read Once Non-deterministic Thrifty Branching Programs
BINTBP	Bitwise Independent Non-deterministic Thrifty Branching Programs
NINTBP	Nodewise Independent Non-deterministic Thrifty Branching Programs
CC	Comparator Circuits
NL	Non-deterministic Logspace
CVP	Circuit Value Problem
CCVP	Comparator Circuit Value Problem
TEP	Tree Evaluation Problem

CHAPTER 1

Introduction

Increasingly large amounts of data are processed by computers in the modern world. In this context, the spotlight falls on the efficiency of computers or, more concretely, the efficiency of algorithms used to process this data. Current trends indicate that the size of problems handled by computers are going to rise. So algorithms should not only run efficiently on the inputs that are fed to them, but they have to provide a guarantee that they will remain to be reasonably efficient in the face of increase in the size of inputs. Design and analysis of algorithms focus on designing algorithms that provide this guarantee. A fundamental understanding of computation is necessary to identify the problems that can be efficiently solved using a computer.

The first step in undertaking a formal study of computation is to define computation. Several attempts to do this are summarized in Kozen (1997). Later, it was shown that all models proposed with the aim of formalizing computation are equivalent to the Turing Machine model. The branch called computability theory uses this model to broadly divide all problems into computable ones and uncomputable ones.

One shortcoming of this approach is that even if a problem is computable, the best algorithm for solving it may take millions of years even on moderately sized inputs that are of interest to the user. The key point here is that we can only solve problems (in practice) that are efficiently solvable. But, what criteria should we use to decide whether a given problem is efficiently solvable? Informally, we consider various computational resources such as time taken by the algorithm or space used by the algorithm and consider a problem to be efficiently solvable if and only if there is an algorithm that solves the problem using a reasonable amount of resources (an upper bound). The exact definition of “a reasonable amount” depends on the resource and the problem being solved.

Complexity theory is a branch of theoretical computer science that deals with the study of relationships between various complexity classes – sets of problems that share common resource upper bounds. This is best illustrated by stating the most famous open problem in complexity theory, the P vs NP problem. Here, we consider two classes of problems; the class P, the set of problems for which we can compute the solution in polynomial time (polynomial time is considered reasonable) and the class NP – the set of problems for which we can verify whether a given solution is correct in polynomial time. The question is whether $\mathsf{P} = \mathsf{NP}$ or not. Informally, $\mathsf{P} = \mathsf{NP}$ would mean that all problems that are efficiently verifiable are efficiently solvable too. A situation that, our intuition tells, is unlikely. So it is commonly believed that $\mathsf{P} \neq \mathsf{NP}$.

1.1 The L vs P Problem

Another resource of utmost importance is the space consumed by the algorithm during computation. An algorithm that takes at most logarithmic space is considered efficient with respect to the space consumed. This is the class L. It is known that L is contained in P and it is widely believed that L is properly contained P. That is, there are some problems in P that cannot be solved in logarithmic space. This question on the relationship between the classes L and P is called the L vs P problem.

An approach proposed by Cook (1970) to solve the L vs P problem is to start with a problem in P and employ the following two general steps to show that the problem is not in L.

1. Consider "natural" algorithms for solving the problem. Prove that they require super-logarithmic space.
2. Prove that no algorithm can solve the problem more efficiently than "natural" algorithms considered in step 1.

Examples of this approach can be found in Cook (1970), Barrington and McKenzie (1991), Cook *et al.* (2012), Gál *et al.* (2008).

In Cook *et al.* (2012), the authors proposed the Tree Evaluation Problem (TEP) as a candidate for separating L from P. This problem is defined over the family of complete binary trees. As input, we are given the height of the tree and a parameter k; the input also associates, for each leaf node in the tree a value from $[k]$ and for each internal node an arbitrary function $f_i : [k] \times [k] \mapsto [k]$. The goal is to evaluate the tree in a bottom-up fashion where the value at each internal node is obtained by evaluating the corresponding function on the values of its children. The output is the value at the root node. If we can show that any algorithm solving this problem must store a non-constant number of k-ary values (That is, the number of k-ary values stored increases with the size of the tree), then $\mathsf{L} \neq \mathsf{P}$.

Cook *et al.* (2012) proposed to consider thrifty computational models as natural models solving this problem. A thrifty model solving this problem only evaluates functions given as part of the input at the values of its children. This is a natural restriction for this problem since the output does not depend on the values of these functions at any other point.

Cook *et al.* (2012) considered a formal computational model called branching programs, that are a restricted form of circuits, to study about the space complexity of this problem (The circuits we consider are just usual Boolean circuits with AND, OR, and NOT gates. Their relation to other computational models are detailed in Chapter 2). It is known that obtaining super-polynomial size lower bounds on the size of branching programs imply super-logarithmic space lower bounds.

Cook *et al.* (2012) showed that deterministic thrifty branching programs solving this problem must have super-polynomial size. This lower bound is obtained by connecting the computation proposed by thrifty models to a combinatorial game called the black pebbling game. They also defined non-deterministic thrifty models solving this problem, a more powerful model than the deterministic thrifty model. A generalization of the black pebbling game called black-white pebbling game is used to study non-deterministic computation. Cook *et al.* (2012) showed that non-deterministic thrifty models can implement a generalization of black-white pebbling games called fractional black-white pebbling games, thereby yielding asymptotic improvements in size. They asked whether non-deterministic thrifty models also require super-logarithmic space to solve TEP.

In Chapter 3 of this thesis, we show that syntactic read-once non-deterministic thrifty branching programs (RONTBP) require super-polynomial size to the solve the problem. This computational model can solve TEP in asymptotically smaller size than deterministic thrifty branching programs. This lower bound is shown by associating computation performed by RONTBPs with the black-white pebbling game. We then introduce a restricted version of non-deterministic thrifty branching programs called bitwise independent branching programs. We show that this model can implement the algorithm that allows general non-deterministic thrifty programs to gain the asymptotic size improvement. We then prove that the bitwise independent model also require super-polynomial size to solve the problem by associating its computation with the fractional black-white pebbling game.

Combinatorics of Reversible Pebbling Pebbling games, used as a tool for proving lower bounds for TEP, are used to capture space in many other settings. The reversible pebble game on DAGs was introduced by Bennett (1989) as an abstraction that captures space used by reversible algorithms. Recently, Chan (2013) showed that the reversible pebble game is equivalent to two other pebble games on DAGs – the Dymond-Tompa pebble game and the Raz-Mckenzie pebble game – that have been studied in relation to circuit depth. However, no connections were known between reversible pebbling and classical graph parameters not arising out of computation.

In Chapter 3 of this thesis, we show that the reversible pebbling number of (directed) trees is connected to edge rank coloring of trees, a classical graph parameter. Our work introduces the notion of strategy trees for trees, objects that succinctly encode optimal pebbling strategies, and uses this to prove the connection. Our proof also implies that the reversible pebbling number of trees can be computed in polynomial time. Prior to our work, it was not known whether this problem is in NP.

1.2 The P vs NC Problem

We now turn our attention to the study of problems that have efficient parallel algorithms. Boolean circuits are widely used in the study of parallel algorithms. The depth of a Boolean circuit is the maximum (over all input gates) number of gates from the input gate to the output gate. The depth of a circuit corresponds to the time taken by a CREW PRAM (JáJá (1992)), a computational model that characterizes computation with multiple processors. Therefore, low depth circuits can be thought of as characterizing efficient parallel algorithms.

But, what depth can be considered "low"? Since we are only interested in polynomial size circuits (super-polynomial sized circuits are usually considered infeasible), the maximum depth is at most polynomial. A bounded fan-in circuit requires $O(\log(n))$ depth to read all of the input. And, we would like circuits obtained by composing efficient circuits to be considered efficient as well. Therefore, circuits where depth is bounded by $O(\log^k(n))$ where k is a constant is considered low depth. This is the class NC (See 8).

The class NC is in P because P contains all problems having polynomial size circuits. The next question is whether all problems in P have such low depth circuits. The commonly held belief is that this is not possible. The P vs NC problem asks whether or not P is contained in NC.

1.2.1 Bounded Depth Circuits

The smallest class in the NC-hierarchy, called NC^1, is the set of problems solvable by logarithmic depth, bounded fan-in circuits. Recall, that logarithmic depth is the minimum required to read all input bits. But, if we allow gates to have unbounded fan-in, it is possible to restrict the depth even further. This leads us to constant depth circuit classes. Here, the circuit depth is bounded by a constant. In other words, the depth of the circuit cannot increase with the size of the input. For example, the AND of n-inputs can be computed by a circuit having a single AND gate of fan-in n. Of course, we would want to restrict our circuits to have polynomial size as otherwise any Boolean function can be computed by depth-2 circuits with unbounded fan-in AND and OR gates, by simply implementing the CNF of the function. It is known that these constant depth circuit classes are contained in NC^1. Also, constant depth circuit classes represent the lowest level of complexity.

Barrington and Thérien (1988) proposed a framework to study the class NC^1 and constant depth circuit classes AC^0 and ACC^0. They considered a class of problems called monoid word problems. These problems are in NC^1. They showed that the complexity of these monoid word problems vary with the algebraic structure of the monoid. They showed that these complexity classes can be characterized using the algebraic structure of monoids.

In Chapter 4, we use the framework proposed by Barrington and Thérien (1988) and show that testing graph properties such as existence of perfect matching and existence of edge disjoint paths in constant width grid-planar graphs can be done using constant depth circuits. The previous known upper bound for these problems were NC^1. That is, logarithmic depth circuits. We introduce the primary cycle method, that links the algebraic structure of monoids to the combinatorial aspects of the considered graph problems.

1.2.2 Comparator Circuits and Hardness of Parallelization

The set of problems that are P-complete are thought of as problems that have efficient sequential algorithms but are unlikely to have any efficient parallel algorithm. Examples include the Circuit Value Problem and the problem of finding the lexicographically least independent set in a graph. However, some problems such as the stable matching problem and the problem of finding the lexicographically least maximal matching in a graph are not known to be P-complete, nor do they have efficient parallel algorithms. The class CC of problems reducible to comparator circuit (Another restricted form of Boolean circuits) value problem was introduced by Mayr and Subramanian (1992) and they showed that these problems are complete for this class. This explains the above scenario because on the one hand comparator circuits, being fan-out restricted, may not be powerful enough to solve all problems in P; and on the other hand, evaluating deep comparator circuits parallely does not seem feasible.

Recently, Cook *et al.* (2014) showed that comparator circuits are a universal computational model which implies that the class CC satisfies certain nice complexity-theoretic properties. However, a definition of the class CC in terms of the TM is not yet known. This is in stark contrast to other complexity classes such as P, NC, L, and NL that can be defined in terms of TMs as well as circuits.

In Chapter 5, we define generalized comparator circuits – comparator circuits that compute over arbitrary lattices. We design a lattice and show that comparator circuits working over that lattice can compute all problems computable by general Boolean circuits (effectively eliminating the fan-out restriction). Furthermore, we show that skew (A well-studied restriction for Boolean circuits) comparator circuits capture the class L. We characterize many classical complexity classes in terms of comparator circuits. We also note that allowing other circuit models, such as Boolean circuits or

Formulae, to compute over any fixed lattice does not increase their power.

An intriguing open problem arising out of our work is to understand how lattice structure affects the power of comparator circuits. We know that distributive lattices does not change the power of comparator circuits. An open question is whether any non-distributive lattice would allow comparator circuits to perform computation that can be performed by general Boolean circuits.

1.3 Outline of the Thesis

In Chapter 2, we state preliminary definitions needed in the rest of the thesis. The Cook's program for separating L from P and our contributions to this program are described in Chapter 3. We also detail the connection between pebbling and coloring in Chapter 3. In Chapter 4, we look at bounded depth circuits and their connections to algebraic structures called monoids. We show how this connection can be exploited to yield better algorithms (circuits) for some combinatorial problems. In Chapter 5, we look at comparator circuits and detail how our generalization of comparator circuits allows us to characterize classical complexity classes.

CHAPTER 2

Preliminaries

In this chapter, we present basic definitions needed for the rest of this thesis. In Section 2.1, we introduce the Turing Machine and basic complexity classes that arise by considering time taken and space consumed by a Turing Machine. In Section 2.1, we consider the alternative computational model of Boolean circuits and restricted forms of circuits called Branching programs, which characterizes space consumed by a TM in a natural fashion.

2.1 Basic Complexity Classes

We will assume that our computational model is a k-tape deterministic Turing Machine, where k is an arbitrary constant. We will be interested mainly in decision problems. Let Σ be a finite alphabet. The decision problem corresponding to a language $L \subseteq \Sigma^*$ is, given $x \in \Sigma^*$ on a read-only input tape as input determine whether $x \in L$. We will use the shorter form "problem L" to mean "the decision problem corresponding to the language L" frequently in this thesis. Similarly, references to "the complexity of language L" should be considered as "the complexity of the decision problem corresponding to the language L".The time taken by an algorithm on an input x is defined as the number of transitions made by the TM in determining the membership of x in L. The space taken by an algorithm is the number of work tape cells used by the algorithm while determining the membership of x in L. Usually, we are only interested in the growth rate of time taken or space consumed as a function of the input size. Therefore, we express these resource bounds as a function of n, the input size. Furthermore, we omit constant factors and use asymptotic notation while specifying these functions. For example, an algorithm that takes cn steps for all inputs of length n_0 or more, where c and n_0 are constants, is said to have time complexity $T(n) = O(n)$. For more rigorous definitions of these concepts, see Sipser (1997), Arora and Barak (2009). Given a language L, we are only interested in the resource bounds of the best algorithm for L. For example, we may ask whether a language L has an algorithm that takes only $O(n)$ steps. In complexity theory, we turn this question around, and ask which languages can be decided in $O(n)$ time. Generalizing this leads to the complexity classes; DTIME(f(n)) which is the set of problems that can be solved in $f(n)$ time and DSPACE(s(n)) which is the set of problems that can be solved in $s(n)$ space.

Complexity theory is the study of relationships between complexity classes. It is clear that $\mathsf{DTIME}(n) \subseteq \mathsf{DTIME}(n^2)$. Intuition tells us that this containment is strict. i.e., there are languages decidable in $\mathsf{DTIME}(n^2)$ that cannot be decided in $\mathsf{DTIME}(n)$. Hartmanis and Stearns (1964) proved that this statement is indeed true. Infact, they prove more general versions of these theorems called hierarchy theorems. They prove that $\mathsf{DTIME}(f(n)) \subset \mathsf{DTIME}(g(n))$ ($\mathsf{DSPACE}(f(n)) \subset \mathsf{DSPACE}(g(n))$) if $f(n)$ and $g(n)$ are "nice" and $g(n)$ is sufficiently larger than $f(n)$. A precise statement of these results can be found in Hartmanis and Stearns (1964). We will sketch the proof of these theorems. Let us consider $\mathsf{DTIME}(n)$ and $\mathsf{DTIME}(n^2)$. The idea is to define a language L such that it can be decided in n^2 steps, yet different from all languages in $\mathsf{DTIME}(n)$. The definition is given in terms of an algorithm/TM M deciding the language. The machine M on input x simulates the execution of machine M_x given input x (The linear time machine described by string x) and rejects if M_x accepts and accepts otherwise. Since M_x is linear time, M runs in n^2 time. Also, for any linear time machine M_y, M will differ from the language of M_y on input y and hence $L \notin \mathsf{DTIME}(n)$. The time and space hierarhy theorems are proved using this idea.

How can we characterize the class of problems that can be solved efficiently in terms of time or space? Let us consider time first. Obviously, languages decidable in linear time have to be considered efficient, because it takes at least n steps to look at all of the input. In addition, if all that an algorithm does is to call another linear time algorithm a linear number of times, we would consider that efficient as well. But, the new algorithm runs in $O(n^2)$ time. Generally, we would like to consider algorithms that are obtained by composing efficient algorithms in an efficient manner to be efficient as well. The smallest class of problems satisfying this criteria is the class P. Therefore, this is usually considered the class of problems solvable efficiently in terms of time.

Definition 1. $\mathsf{P} = \bigcup_{k \geq 0} \mathsf{DTIME}(n^k)$

Similarly, the class L is considered to be the class of problems efficiently solvable in terms of space consumed. It is defined as follows.

Definition 2. $\mathsf{L} = \mathsf{DSPACE}(\log n)$

We will see later that $\mathsf{L} \subseteq \mathsf{P}$. But is it properly contained in P? This problem has been open for nearly 45 years and forms the motivation for most of the work in this thesis.

To understand why diagonalization proofs such as the ones used to prove hierarchy theorems fail to separate L from P, we need to understand the concept of complete problems. A language L is logspace reducible to L' iff there is a logspace computable function f such that $x \in L \iff f(x) \in L'$. A language $L \in \mathsf{P}$ is said to be complete for the class P with respect to logspace reductions if any L' in P is logspace reducible to L.

So why does proof techniques similar to those used to prove hierarchy theorems to prove that $\mathsf{L} \subset \mathsf{P}$? This is because that proof falls into the category of relativizable proofs and we can show that relativizable proofs cannot separate L from P. We will simply sketch the idea behind relativizable proofs (See Baker *et al.* (1975)). The proof sketch is as follows. We consider TMs augmented with oracles. An oracle for a language L allows the TM M to resolve queries of the form $x \in L$ in logarithmic space. We can define the complexity classes P^L and L^L to be the set of languages decided by poly-time and logspace TMs with access to an oracle to L. Now if L is a P-complete language. Then the class $\mathsf{P}^L = \mathsf{P} = \mathsf{L}^L$ trivially. Therefore, any proof that separates L from P should not hold in the presence of P-complete oracles. Now, the key point is that the proof of hierarchy theorems hold even in the presence of arbitrary oracles. So such proof techniques cannot separate L from P.

2.2 Alternative Computational Models

This section introduces alternative computational models to the TM. Why do we need other computational models if the TM adequately captures the computational complexity of problems in the real world? The reason is that it seems very hard to prove computational lower bounds on the TM model. The only non-trivial (super-linear) time lower bounds known for TMs hold only for single tape TMs. Therefore, there is great interest in alternative models for which proving lower bounds would be easier. Also, many alternative computational models allows us to consider computational resources other than time and space in a more natural fashion. For example, the Boolean circuit model characterizes parallel time better than TMs.

2.2.1 Boolean Circuits

In this subsection, we look at the Boolean circuit model. A Boolean circuit is basically a directed acyclic graph where nodes are marked as AND, OR, or NOT gates. Since they are graphs primarily, it allows us to use tools from graph theory to argue about computation.

Definition 3. *A Boolean Circuit on n inputs is a directed acyclic graph with n source nodes (called input gates) and 1 sink node (the output gate), say w. The source nodes are labelled as $x_1, \ldots, x_n$, the input variables. All non-source nodes (called gates) are labelled as AND, OR, or NOT. The AND and OR gates have in-degree 2 (The in-degree is also called the fan-in of the gate) and NOT gates have in-degree 1. output of a Boolean Circuit on an input $y \in \{0,1\}^n$ is defined inductively as follows. The output is the value $val(w)$ and for any AND or OR v, we define $val(v) = val(v_1) OP val(v_2)$ (following usual Boolean logic) where OP is AND or OR and v_1 and v_2 are the in-neighbours of v. For any NOT gate v, we define $val(v) = NOT val(v_1)$. For any input gate v labelled x_i, we define $val(v) = y_i$.*

The size of a Boolean circuit C, denoted $|C|$ is defined as the number of gates in the circuit. The depth of C is defined as the maximum number of gates in any directed path from an input gate to the output gate.

A Boolean circuit family is a family of Boolean circuits $F = \{C_n\}_{n\geq 0}$, where C_n is a Boolean circuit on n inputs. We say that F computes the language $L \subseteq \{0,1\}^*$ if and only if $val(C_n)$ on input $x \in \{0,1\}^n$ is 1 when $x \in L$ and 0 otherwise.

We say that a language is decided by a Boolean circuit family of size (depth) $s(n)$ ($d(n)$) if $|C_n| \leq s(n)$ (depth of C_n is at most $d(n)$) for all n.

Typically, we use order notation to specify size and depth of Boolean circuit families. The Definition 3 can be extended to other classes of circuits in a straightforward fashion. The variants considered in this paper include circuits with unbounded fan-in (Here AND gates and OR gates could have many inputs), circuits with gates other than AND, OR, and NOT (for example, XOR gates). We do not present formal definitions of these similar models to avoid repetition.

As per Definition 3, a Boolean circuit family is a non-uniform computational model. i.e., different circuits are used for different input lengths. Contrast this with the TM model, where a single TM is used for all input lengths. As descriptions of algorithms, Boolean circuit families are useless as they can even solve undecidable problems. Consider the language Halting problem, a classic undecidable language, encoded in unary (i.e., alphabet of size 1). Any unary language has a Boolean circuit family of size 1. The output wire of the n^{th} circuit simply outputs 0 or 1 according to whether $1^n \in L$. But any reasonable description of algorithms shouldn't be able to decide undecidable languages. We alleviate this problem by introducing the notion of uniformity of circuit families.

Definition 4. *A circuit family F is called* P*-uniform if and only if there is a polynomial time algorithm on input 1^n outputs the description of C_n, the n^{th} circuit in the family F, in time polynomial in n.*

This definition of uniformity can be readily extended to other complexity classes. For example, replacing P by L and polynomial time by logspace defines the class of L-uniform circuit families.

We will slightly abuse the terminology by referring to Boolean circuit families as Boolean circuits. For example, we frequently refer to polynomial size Boolean circuit families as polynomial size circuits. Such abuses are clear from the context as it does not make sense to talk about a polynomial size restriction on circuits as opposed to circuit families.

Once we have uniformity, we can characterize classical complexity classes using circuit families. For example, the class P can also be defined as the class of languages decided by P-uniform circuits of polynomial size. We will only sketch the proof of this fact. If a language has uniform circuits, then a polynomial time TM can run the uniformity algorithm to get a description of the circuit and then evaluate the output of the circuit on the input string. The other direction is much harder and we omit its proof.

The interested reader may consult Arora and Barak (2009) for a complete proof.

We note that the uniformity condition can be done away with when we are proving lower bounds. If we can prove that a language L cannot be decided by circuits of size $s(n)$, then it cannot be decided by uniform circuits of size $s(n)$. But why do we think that proving lower bounds against circuits would be easier than proving lower bounds against TMs if they are equivalent in computational power? The reason is that circuits are primarily directed acyclic graphs and this fact enables the use of many graph theoretic tools to prove theorems related to computation. For example, Hopcroft *et al.* (1977) proves that $\mathsf{DTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n)/\log f(n))$ using graph theoretic tools. To summarize, inorder to prove that $\mathsf{P} \neq \mathsf{NP}$, one need only prove that some problem in NP does not have Boolean circuits of polynomial size.

2.2.2 Branching Programs

We now introduce another non-uniform computational model called branching programs (BPs). This model is also based on DAGs and later on we will see that branching programs are a restricted form of Boolean circuits. The importance of this model stems from the fact that they provide a natural way to characterize space in terms of the size of DAGs.

Definition 5. *A branching program B on n inputs is a DAG with a unique source node and 2 sink nodes labelled 1 and 0. All non-sink nodes (called query states) are labelled $x_1, \ldots, x_n$ and all non-sink nodes have exactly two outgoing edges, one labelled 0 and the other labelled 1. The output value of B on input $y \in \{0,1\}^n$ is defined as follows. A directed path from the source node to a sink node is defined as consistent with the input y iff for any query state on the path labelled x_i, the outgoing edge from the query state on the path is labelled y_i. There is exactly one such path consistent with y. The output value is 1 if the sink node in that path is labelled 1 and the output value is 0 otherwise.*

The definition of a branching program family deciding a language is similar to that of circuit families deciding languages in Definition 3. The size of a branching program is the number of query states. A branching program family is said to have size $s(n)$ if and only if the n^{th} branching program in the family has size at most $s(n)$.

It is known that L-uniform branching programs of polynomial size characterize the class L. So, in order to prove that a language L is not in L, it is sufficient to show that L cannot have polynomial size branching programs (possibly non-uniform). Notice the similarity to the Boolean circuits approach to proving lower bounds against P. Since branching programs are also DAGs, it is possible that tools from graph theory could help in proving such lower bounds. Definition 5 can be generalized to define non-deterministic branching programs.

Definition 6. *A non-deterministic branching program is similar to a (deterministic) branching program as defined in Definition 5 except that query states are allowed to have zero or more outgoing edges with the same label (as opposed to exactly 1 in Definition 5). Therefore, there could be many paths from the source node to sink nodes. The value of a non-deterministic branching program on input y is then defined as 1 if there is at least 1 directed path consistent with y from the source node to the sink node labelled 1 and the value is defined as 0 otherwise.*

Non-deterministic branching program families and their size are defined as in Definition 5.

It is known that poly-size L-uniform non-deterministic branching programs characterize the complexity class NL. Another important observation is that non-deterministic branching programs is just a form of restricted circuits called skew circuits.

Definition 7. *A Boolean circuit C is called skew if and only if for every AND gate in C, one of the in-neighbours of the AND gate is an input gate.*

Since $\mathsf{L} \subset \mathsf{NL}$, it is clear that deterministic branching programs can be characterized as restricted skew circuits. But, what restriction exactly? We settle this question in Section 5.5.

Since separating L from P has proved to be very hard. It makes sense to understand more about the complexity class L. One way to do this is to consider restricted branching programs and try to prove lower bounds against them. The hope is that these techniques or insights gained from these proofs could be transferred to prove lower bounds against general branching programs, or equivalently L.

One such restriction of the branching program model is the model of layered branching programs. In a layered branching program, nodes are arranged into layers with

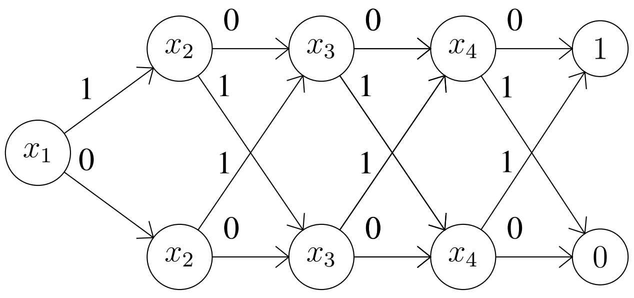


Figure 2.1: A branching program computing the parity of four input bits

the first layer containing only the start node and the last layer containing only accept and reject nodes. Edges only go from a node to another node is the immediate next layer. It is known that poly-size layered branching programs are equivalent in computational power to poly-size branching programs. A natural restriction is to consider layered branching programs that have only a constant number of nodes in each layer. Such branching programs are called constant width branching programs. An example is shown in Figure 2.1. This branching program computes the PARITY function on 4 bits and has width 2. It was assumed that constant width branching programs are severly limited in their computational power and that they cannot even compute the MAJORITY function in polynomial size. However, in his seminal paper, Barrington (1989) proved that width 5 branching programs can compute all languages in NC^1, a circuit based complexity class that contains MAJORITY. This result also revealed a very unexpected connection between a branching program model and a circuit class. To explain this connection, we have to define bounded depth circuit classes.

2.2.3 Bounded Depth Boolean Circuits

When designing Boolean circuits, we saw that we are primarily interested in optimal size circuits for deciding languages. Another resource of interest in the boolean circuit world is the depth of the circuit. The depth of a Boolean circuit is the maximum length of a directed path from an input gate to the output gate. This measure naturally corresponds to parallel time or delay in obtaining the output from a circuit once the inputs are fed to the circuit, assuming that each gate takes a unit of time to compute its output. In this subsection, we look at bounded depth Boolean circuits, they characterize languages that can be decided efficiently in terms of parallel time.

Definition 8. *A language is said to be in class* NC^k *if it can be decided by a Boolean* *circuit family of depth* $O(\log^k(n))$.

The class NC *is defined as* $\mathsf{NC} = \bigcup_{k\geq 1} \mathsf{NC}^k$

The class NC is thought of as the class of problems solvable efficiently in parallel. Indeed, if a problem is in uniform NC^i, then we can solve it in $\log^i(n)$ time if polynomially many processors are available. Therefore, Barrington's theorem reveals a connection between space (captured by branching programs) and parallel time (captured by bounded depth ciruits) in a restricted setting.

What more do we know about the NC hierarchy? The result of It is known that $\mathsf{NC}^1 \subseteq \mathsf{L}$. Also, it can be shown that $\mathsf{L} \subseteq \mathsf{NC}^2$. Since NC is characterized by restricted polynomial size Boolean circuits, it follows that $\mathsf{NC} \subseteq \mathsf{P}$. This leads us to another important open problem in complexity theory. The NC vs P problem. i.e., Are all problems that have efficient sequential algorithms efficiently solvable in parallel time as well?

Earlier, we saw that the notion of P-completeness is used to gain understanding about the L vs P problem. P-completeness also plays an important role in the NC vs P problem. If a language is P-complete, it is considered hard to parallelize. To see why, suppose some P-complete language (with respect to logspace reductions) L can be solved in NC^i, then we can argue that any language in P can be solved in NC^j, where $j = \max i, 2$. Since $\mathsf{L} \subseteq \mathsf{NC}^2$, first use the NC^2 circuit to reduce the input string to an input to the NC^i circuit for L and compute the answer. This new circuit obtained by composition is an NC^j circuit. Since we believe that there are languages in P that does not have efficient parallel algorithms, we believe that P-complete languages are hard to parallelize.

Now, we define constant depth circuit classes. These classes represent the lowest levels of computational complexity. These characterize problems for which the parallel time required to solve the problem is independent of the input size.

Definition 9. AC^0 *is the class of languages decided by constant depth polynomial size families of Boolean circuits with NOT gates and unbounded fan-in AND and OR gates.*

Since any AND or OR gate of fan-in n^k can be replaced by a $k\log(n)$ depth tree of fan-in 2 AND or OR gates to yield the same function, we have that $\mathsf{AC}^0 \subseteq \mathsf{NC}^1$. It is known that the language PARITY, which consists of all binary strings where there are an odd number of 1 bits, is not a member of AC^0. But, PARITY is in NC^1. Therefore, $\mathsf{AC}^0 \neq \mathsf{NC}^1$. But, if we allow constant depth circuits to use unbounded fan-in PARITY gates, then it can compute PARITY trivially. Generalizing this observation leads to the following circuit classes.

Definition 10. $\mathsf{AC}^0[m]$ *is the class of languages decided by constant depth polynomial size families of Boolean circuits with NOT gates and unbounded fan-in AND, OR, and MOD(m) gates, where a MOD(m) gates output is 1 if the number of input bits with value 1 is 0 modulo m. The output is 0 otherwise*

Definition 11. $\mathsf{ACC}^0 = \bigcup_{m \geq 2} \mathsf{AC}^0[m]$

We will refer to the classes AC^0, $\mathsf{AC}^0[m]$, and ACC^0 collectievely as constant depth circuit classes. All of them are in NC^1.

CHAPTER 3

Tree Evaluation Problem and Pebbling Games

In the Section 3.1 of this chapter, we take a look at the program proposed by Stephen Cook to separate P from L. The basic idea is to *design* a problem that can be solved in P but not in L. Here the term design means that the problem need not arise from real world computational problems, but can be defined in a manner that will facilitate proving the required lower bound. Examples of such designed languages include diagonalization languages used to prove hierarchy theorems. The designed problem usually has a straightforward algorithm that runs in P. The crucial part of the design of the problem is in ensuring that this straightforward algorithm is very close to the optimal algorithm in terms of space efficiency. If we can prove that these straightforward algorithms take super logarithmic space and also that they are close to optimal, then we have essentially separated L from P.

We will track the progress of Cook's program upto the latest problem designed to achieve this goal – the Tree Evaluation Problem (defined in Section 3.2). We will consider combinatorial games called pebbling games. These games are used to capture the complexity of natural algorithms (believed to be close to optimal) for solving the tree evaluation problem. We will then look at how this game is used to prove lower bounds for computational models solving the tree evaluation problem (Sections 3.3 and 3.4). We will analyze the method used to prove these lower bounds in Section 3.5. These results can be found in Komarath and Sarma (2015).

We will also look at combinatorial aspects of the reversible pebble game, a pebble game closely tied to the space complexity of reversible algorithms (Section 3.6). These results can be found in Komarath *et al.* (2015*a*).

3.1 Cook's Program

In this section, we will briefly survey Cook's program to separate L from P. Cook (1974) introduced the solvable path systems problem. In this problem, we are given as input a base set X, a source set $S \subseteq X$, a terminal set $T \subseteq X$, and a terneary relation R on the set X. The problem is most naturally defined using an algorithm solving it (As is the case with many languages designed for proving lower bounds). We start with a set A. Initially, A only contains the elements in T and if we have $y, z \in T$ and $R(x, y, z)$ for some x, then we add x to A. We add elements to A until we cannot add any more by applying the above rule. We say that the tuple (S, T, R) is in the language iff the set A contains at least one element of S after all possible elements have been added to A. Cook (1974) proved that this problem is P-complete. He also proved super-logarithmic space lower bounds for marking machines solving the solvable path systems problem. Marking machines capture a class of "natural" algorithms solving this problem and are believed to be close to optimal algorithms in terms of space efficiency. In particular, marking machines capture the algorithm mentioned as part of description of the problem. Intuitively, marking machines solve this problem by keeping markers on the elements of X that are known to be in A. A new marker can be placed on $x \in X$ iff there are markers on some y, z and $R(x, y, z)$ holds. We are allowed to place markers on any element in T unconditionally as A contains elements of T trivially. Here the number of markers correspond to the space used by the algorithm and Cook (1974) proves that a large number of markers are required which implies the required lower bound for the marking machine model.

Barrington and Mckenzie Barrington and McKenzie (1991) took a similar approach by considering the problem GEN and attempted to prove (upper and lower bounds) for increasingly stronger models of computation for solving GEN. In this problem, the input is a multiplication table of the set $\{1, \ldots, n\}$ and a set $S \subseteq \{1, \ldots, n\}$. The question is whether n belongs to the set closure(S), the set of all elements generated via multiplication starting with the elements of S. Barrington and McKenzie (1991) considered the computational model of "oracle" branching programs where each state of the branching program is allowed to ask a question about the input depending on the oracle. An oracle particularly suited to the GEN problem would ask what $i \times j$ is where $i, j \in \{1, \ldots, n\}$. A general BP as in Definition 5 can ask queries of the form "What is the i^{th} bit of the input?" (This is called a branching program with BIT oracle by Barrington and McKenzie (1991)). Barrington and McKenzie (1991) proved exponential size lower bounds for branching programs equipped only with certain "weak" oracles. Gál *et al.* (2008) considered incremental branching programs for solving GEN

which can be thought of as branching programs trying to solve the GEN problem by incrementally finding the elements of the set closure(S).

The common theme running through all these papers is the fact that super-polynomial size lower bounds are proved against *semantic* restrictions of the branching program model. Most branching program models for which good lower bounds are known are *syntactically* restricted (See Razborov (1991)). i.e., the restriction is made on the structure of the DAG underlying the branching program and as such, the restriction has little to do with computational aspects. This has the drawback that these lower bounds yield little insight that can be used for designing more efficient algorithms (or understanding why the current class of algorithms are not efficient enough for solving the problem). But, lower bounds against semantic restrictions do not suffer from this drawback.

3.2 Tree Evaluation Problem

Cook *et al.* (2012) introduced the tree evaluation problem as part of Cook's program. First, we will build up some intuition behind the definition of the problem. Consider the problem of evaluating $(a+b)(c+d)$ given values a, b, c, and d. The straightforward way to calculate this would be to evaluate $a+b$, store that value, then evaluate $(c+d)$, then multiply that value with the stored value $(a+b)$ and output the result. Now consider a larger expression $((a+b)(c+d))((e+f)(g+h))$. To evaluate this expression, we first compute $(a+b)(c+d)$ as before and then store that value. Then we compute $(e+f)$ and store it. Note that at this point we are using one more storage location than for the previous problem – One to store $(a+b)(c+d)$ and one to store $(e+f)$. It seems that if we consider larger and larger expressions of this form, then the storage has to increase super-logarithmically. But alas, this is not true. We can use the fact that multiplication is associative and multiply $(e+f)$ with the stored value of $(a+b)(c+d)$ before computing $(g+h)$. But what if we allow arbitrary operations instead of addition and multiplication. Then it seems that the trivial algorithm is all that we can do. The Tree Evaluation Problem (TEP) takes this a step further. Every operation in the expression is allowed to be an arbitrary one. The motivation behind designing such a problem is to ensure that the straightforward algorithm to evaluate such an expression would be close to the best possible, since we cannot use special properties of operations (like associativity) to make the algorithm more space efficient.

We now give a formal definition of TEP.

Definition 12 (Tree Evaluation Problem)**.** *The input to the problem is the tuple* $(h, k, f_1, f_2, \dots f_{2^{h-1}-1}, \ell_{2^{h-1}}, \dots \ell_{2^h-1})$ *where* $h, k \geq 2$ *are integers given as input in unary, each* ℓ_i *is an integer in* $[k]$ *and each* f_i *is a function from* $[k]^2$ *to* $[k]$ *except for* f_1 *which is a function from* $[k]^2$ *to* $\{0, 1\}$*. The input is associated with the complete binary tree of height* h *as follows: each leaf node* i *is associated with the value* ℓ_i *and each internal node* i *is associated with the function* f_i*. The input is a yes instance iff the value* v_1 *of the root node is 1, where* $v_i = \ell_i$ *if* i *is a leaf node and* $v_i = f_i(v_{2i}, v_{2i+1})$ *if* i *is a non-leaf node.*

We denote the above problem as $\mathsf{BT(h,k)}$ *or simply as* TEP *when* h *and* k *are understood from the context.*

The straightforward way to solve TEP would be compute the values at nodes in a bottom-up fashion. Infact, this algorithm shows that TEP is in LogDCFL.

Theorem 1 (Cook *et al.* (2012))**.** TEP $\in$ LogDCFL

The algorithm in Theorem 1 has the following property. For any internal node i, the algorithm only queries $f_i(v_{2i}, v_{2i+1})$ for all inputs. An algorithm satisfying this restriction is called *thrifty*. The main result proved by Cook *et al.* (2012) is to show that no thrifty algorithm can do better than the algorithm in Theorem 1 in terms of space consumed.

Before going into the details of the lower bound, let us define k-ary branching programs, a computational model for solving TEP. This computational model is universal for TEP in the sense that if there is an algorithm solving TEP in logspace, then there is a k-ary branching program solving TEP in polynomial size. Therefore, to prove that there are no logspace algorithms for TEP, it is enough to show that there are no polynomial size branching programs for solving TEP. We will prove all our lower bounds on the branching program model. We will also show how the thrifty restriction maps to k-ary branching programs.

Definition 13 (k-ary Branching Programs for TEP Cook *et al.* (2012))**.** *A* k*-way branching program* B *for* TEP *is a directed graph. It consists of a designated start state and* *two final states labelled accept and reject. Any non-final state is a query state that is labelled either ℓ_i where i is a leaf node or labelled $f_i(x_1, x_2)$ where i is an internal node and $x_1, x_2 \in [k]$. There are k outgoing edges from any query state that are labelled from 0 to $k-1$, except if the query state queries f_1 in which case the outgoing edges are labelled 0 and 1. A computation path on input I is a directed path from the start state such that each edge in the path is consistent with I.*

Notice that we use the term *nodes* to refer to the vertices of the input tree and the term *states* to refer to the vertices of k-ary BPs.

In the remaining part of this chapter, we will consider non-deterministic algorithms solving TEP. We will show how the thrifty restriction applies to non-deterministic algorithms. Then we will define k-ary non-deterministic branching programs solving TEP. We will introduce a new restriction on non-deterministic thrifty branching programs solving TEP that we call bitwise independence. This restriction is satisfied by natural algorithms solving TEP. Then, as our main result in this chapter, we prove that there are no polynomial size bitwise independent non-deterministic thrifty branching programs solving TEP.

Definition 14 (k-ary Non-deterministic Branching Programs for TEP Cook *et al.* (2012))**.** *A nondeterministic k-way branching program B for* TEP *is a directed multi-graph. It consists of a designated start state and final states – accept and reject. The query states are labelled ℓ_i where i is leaf node of $f_i(x_1, x_2)$ where i is an internal node and* $x_1, x_2 \in [k]$. *Each outgoing edge from a query state is labelled by an integer from* 0 *to* $k-1$ *(0 or 1 if it queries f_1) with multiple outgoing edges possibly having the same label. A computation path on input I is a directed path from the start state such that each edge in the path is consistent with I. At least one such computation must end in the final state labelled accept if the input is a yes instance. If the input is a no instance, then no computation path should end in the accept state.*

The next definition shows how the thrifty restriction applies to the branching program model.

Definition 15 (Non-deterministic Thrifty BPs (NTBPs) for solving TEP Cook *et al.* (2012))**.** *A nondeterministic BP solving* $\mathsf{BT}(\mathsf{h}, \mathsf{k})$ *is called* thrifty *if and only if for any accepting computation path on any instance I any query state labelled $f_i(x_1, x_2)$ sat-*

isfies $x_1 = v_{2i}$ and $x_2 = v_{2i+1}$ (i.e., the internal nodes are queried only at the correct values of its children).

To prove the lower bound, we are going to use combinatorial games called pebbling games as a tool. These pebbling games capture the computation done by thrifty algorithms in a natural fashion. We use the notation $\mathsf{T_h}$ to denote the complete binary tree of height h where height the the number of vertices on any path from the root to some leaf node.

Definition 16 (Pebbling Games Cook *et al.* (2012))**.** *A fractional black-white pebbling configuration of a rooted binary tree $\mathsf{T_h}$ is an assignment of a pair of real numbers $(b(i), w(i))$ to each node i of the tree. The values $b(i)$ and $w(i)$ are called the black and white pebble values, respectively, of node i. We have for every i*

$$\begin{aligned} b(i) + w(i) &\leq 1 \\ 0 \leq b(i), w(i) &\leq 1 \end{aligned} \tag{3.1}$$

The legal pebble moves are as follows.

1. *For any node i, decrease $b(i)$ arbitrarily.*
2. *For any node i, increase $w(i)$ arbitrarily.*
3. *For any node i, if each child of i has pebble value 1, then decrease $w(i)$ to 0.*
4. *For any node i, if each child of i has pebble value 1, then increase $b(i)$ arbitrarily and simultaneously decrease the black pebble values of the children of i arbitrarily.*

The number of pebbles in a configuration is the sum over all nodes i of $b(i) + w(i)$. A fractional black-white pebbling of $\mathsf{T_h}$ using p pebbles is a sequence of (legal) fractional black-white pebbling moves on nodes of $\mathsf{T_h}$ which starts and ends with each node having pebble value 0 and at some point the root node has black pebble value 1, and no configuration has more than p pebbles.

A whole black-white pebbling is a fractional black-white pebbling such that for all configurations and all nodes i, we have $b(i), w(i) \in \{0, 1\}$.

A black pebbling is a whole black-white pebbling such that for all configurations and for all nodes i, we have $w(i) = 0$.

Consider the black pebbling game. This game mirrors the computation done by a deterministic thrifty algorithm. The placement of a black pebble on a node corresponds to the algorithm finding out the value at that node. The removal of a black pebble from a node corresponds to the algorithm throwing away the value at that node (that was computed earlier by the algorithm). The rule that pebbles can only be placed on nodes if all its children are pebbled corresponds to the fact that a thrifty algorithm can only compute the value at a node if and only if the values at the children of that node is known[1].

A non-deterministic algorithm, on the other hand, has the ability to guess values at nodes. However, to be correct, the algorithm has to check whether these guessed values are correct or not before accepting the input. This additional power is captured by white pebbles. Placement of a white pebble on a node corresponds to a non-deterministic algorithm guessing the value at that node. The rule for the removal of white pebbles corresponds to the fact that the algorithm can throw away a guessed value only after verifying it to be correct. It turns out that the black-white pebble game is not enough to capture the computational power of non-deterministic thrifty algorithms. This is because a non-deterministic thrifty algorithm could guess some bits of a value v_i and verify it later but at the same time compute the remaining bits of v_i. To capture this, the notion of fractional pebbles was introduce by Cook *et al.* (2012). We can think of a black (white) pebble value of $0 < r < 1$ on a node v as the algorithm knowing by computing (by guessing) $r\lceil \log(k) \rceil$ bits of v_i.

How can we use pebbling games to prove size lower bounds for BPs? We need one more ingredient. We prove lower bounds for thrifty models using a method that is called the *entropy method*. This method was introduced by Jukna and Zák (2001). The method is as follows: For an arbitrary branching program, consider some subset A of inputs to the program. Then, define a function f that maps each input in A to some state in the branching program. After that, show that for any state s in the branching program, the number of inputs from A mapping to s under f is small (Say S). We can conclude that

[1]If the algorithm is not thrifty, it could figure out the value at a node by other means. For ex., it could read the entire function f_i and if f_i is identically 0, then it can conclude that $v_i = 0$ without figuring out the value of children of i.

the branching program must have at least $|A|/S$ states. Since the branching program that we chose was arbitrary, the lower bound follows.

Here's how we combine pebbling games with the entropy method to prove lower bounds. We start by considering an arbitrary branching program satisfying the appropriate thrifty restriction solving TEP. We then define a set E of hard inputs for thrifty programs. Then we consider an arbitrary accepting computation path for $I \in E$ in the branching program. We show that each state in this path can be associated with a pebbling configuration such that the pebbling configurations along the computation path taken in order form a pebbling of the complete binary tree. The distribution function f then maps I into the state that corresponds to the pebbling configuration with the maximum number of pebbles. To finish the proof, we show that only a small number of inputs from E can go through such a state (i.e., one where the associated pebbling configuration has a large number of pebbles).

The following input set will be used to prove lower bounds for thrifty BPs. We note that this set was also used in Cook *et al.* (2012) to prove lower bounds for deterministic thrifty BPs solving $\mathsf{BT}(\mathsf{h},2)\mathsf{k}$. This set is simply referred to as E in the rest of this chapter.

Definition 17 (Hard Inputs for Thrifty BP)**.**

$$\begin{aligned} E = \{I : & f_1^I(x,y) = 1 \textit{ for all } x,y \in [k] \\ & f_i^I(x,y) \in [k] \textit{ if } x = v_{2i}^I \textit{ and } y = v_{2i+1}^I \textit{ for all internal nodes } i \\ & f_i^I(x,y) = 0 \textit{ if } x \neq v_{2i}^I \textit{ or } y \neq v_{2i+1}^I \textit{ for all internal nodes } i \\ & \ell_i^I \in [k] \textit{ for all leaf nodes } i\} \end{aligned}$$

We now prove a simple result that holds for all NTBPs.

Proposition 1. *Let B be an NTBP solving* $\mathsf{BT}(\mathsf{h},2)$*k. Let $I \in E$ and let $C(I)$ be an accepting computation path for I in B, then all nodes are queried in $C(I)$.*

Proof. If the root node is not queried for some $I \in E$, then the input I' which is the same as I but with $f_1^{I'} = 0$ (i.e., f_1 is the zero function in I') is also accepted by B. Let

i be some non-root node and assume that $C(I)$ does not have a state querying node i. Then the input I' which is the same as I but with $f_i^{I'}(v_{2i}^I, v_{2i+1}^I) \neq f_i^I(v_{2i}^I, v_{2i+1}^I)$ is also accepted by $C(I)$. But then $C(I)$ makes a non-thrifty query when querying the parent node of i for either I or I'. Therefore $C(I)$ does not query the parent of I. By induction, we can conclude that $C(I)$ does not query the root node which is a contradiction. $\square$

As a warmup before our main result, we prove that Read-Once NTBPs (RONTBP) solving TEP must have super-polynomial size. We do this by mapping the computation of an arbitrary RONTBP to a black-white pebbling of the complete binary tree.

First, we define RONTBPs.

Definition 18. *An NTBP solving* $\mathsf{BT}(\mathsf{h}, \mathsf{k})$ *is called* syntactic read-once *if and only if any graph-theoretic path from the start state to the accept state queries each node at most once.*

3.3 Read Once Branching Programs

In this section, we prove tight size bounds for RONTBPs solving TEP. We prove upper bounds for RONTBP by showing that they can implement read-once whole black-white pebbling to solve $\mathsf{BT}(\mathsf{h}, \mathsf{k})$.

Proposition 2. *There is a read-once whole black-white pebbling of* $\mathsf{T_h}$ *using* $\lceil h/2 \rceil + 1$ *pebbles.*

Proof. Cook *et al.* (2012) has given a whole black-white pebbling of $\mathsf{T_h}$ using $\lceil h/2 \rceil + 1$ pebbles. We use T_i to denote the subtree rooted at node i. The pebbling strategy in Cook *et al.* (2012) is given below for completeness. We describe the pebbling procedure for height $h + 2$ tree assuming height h tree has a whole black-white pebbling procedure. The induction hypothesis is that $\mathsf{T_h}$ can be pebbled using $N(h) = \lceil h/2 \rceil + 1$ pebbles and there is a critical time in the pebbling of $\mathsf{T_h}$ such that the root node has a black pebble and the tree has at most $N(h) - 1$ pebbles. This is true for $\mathsf{T_2}$ because we can place two black pebbles on leaves and then slide one to the root and remove the other. Now the root has a black pebble and the tree has $N(h) - 1 = 1$ pebble on it.

1. Place a black pebble on node 4 by running the pebbling procedure on T_4.

2. Run the pebbling procedure on T_5, Stop when node 5 has a black pebble on it.
3. Slide the black pebble on node 4 to node 2.
4. Remove black pebble on node 5.
5. Resume the pebbling for T_5 and run it to completion.
6. Run the pebbling procedure on T_6 and suspend when node 6 has a black pebble.
7. Place a white pebble on node 7.
8. Slide the black pebble on node 6 to node 3.
9. Slide the black pebble on node 2 to root node.
10. Remove black pebble from node 3. (This is the critical time for $\mathsf{T}_{\mathsf{h}+2}2$)
11. Remove black pebble from root node.
12. Resume the pebbling for T_6 and run it to completion.
13. Remove the white pebble on node 7 by running the pebbling procedure for T_7.

It is easy to see that this pebbling strategy is read-once. In particular, we stress that the pebbling strategy only suspends the pebbling of subtrees and does not remove any pebbles from it until the pebbling for those subtrees are resumed (This is being done in Steps (2) and (5) and Steps (6) and (12)). Since $\mathsf{T}_2 2$ can be pebbled using $\lceil 2/2 \rceil + 1 = 2$ pebbles in a read-once fashion, it follows by induction that the above pebbling strategy for T_h is read-once. □

Theorem 2. *There is a RONTBP solving* $\mathsf{BT}(\mathsf{h},\mathsf{k})$ *using at most* $(2^h - 1)k^{\lceil h/2 \rceil + 1}$ *states.*

Proof. The construction uses the same idea used by Cook et. al. in Cook *et al.* (2012) to construct an NTBP solving $\mathsf{BT}(\mathsf{h},\mathsf{k})$. Let $C_1, \ldots, C_t$ be the read-once whole black-white pebbling of T_h given by Proposition 2. We now describe a BP B that uses this read-once pebbling to solve $\mathsf{BT}(\mathsf{h},\mathsf{k})$. The BP B will have t layers numbered from 1 to t. The first layer consists of only the start state and the last layer consists of only the accepting state. Let B_i denote the set of all nodes with a black pebble on them in the pebbling configuration C_i and let W_i denote the set of all nodes with a white pebble on them in C_i. Let $p_i = |B_i| + |W_i|$. We will prove by induction (on the layer number) that the i^{th} layer of B has at most k^{p_i} states. We "tag" each state in layer i with a set of possible values for these pebbled nodes such that we will have exactly one state for each setting of possible values for these pebbled nodes. We stress that this "tag" is only used to make the description of the BP B easier. For a state s, we denote its tag by $tag(s)$. The key property is that if the set $tag(s)$ contains $v_i = x$, then for any input I with an accepting computation path through s, we have $v_i^I = x$ (Call this property, P). We will prove that property P holds for all states in the BP B by induction on the layer number. We can now desribe the labelling of states of B and the edges of B (which will correspond to the pebbling moves) easily using these tags. We describe the edges from layer i to layer $i+1$ in terms of the pebbling move that takes the pebbling configuration C_i to C_{i+1}. Note that $p_1 = 0$ and the first layer of B only has the start state. Since $k^0 = 1$, this proves the base case for the fact that the number of states in layer i is at most k^{p_i}. Also note that for the start state s, we have $tag(s) = \{\}$ and hence the property P is vacuously true for the start state.

Place a black pebble on j All states in layer i are labelled ℓ_j if j is a leaf node. Otherwise each state s in layer i is labelled $f_j(x, y)$ where x and y are the values of v_{2j} and v_{2j+1} respectievely in $tag(s)$. We direct the outgoing edge labelled v from s to the state s' in layer $i+1$ such that $tag(s') = tag(s) \cup \{v_j = v\}$ for each $v \in [k]$. Note that the $(i+1)^{\text{th}}$ layer has at most $k^{p_i+1} = k^{p_{i+1}}$ states. Consider any input I with an accepting computation path through some state s' in layer $i+1$. If $tag(s')$ contains $v_j = v$ where v is some element in $[k]$, then $v_j^I = v$ as all the incoming edges to s' are from some state s querying node j and the edge from s to s' is labelled v. Now let us assume that $tag(s')$ contains $v_i = u$ for some $i \neq j$ and some $u \in [k]$. Let s be the previous state in the accepting computation path for I. Then $tag(s)$ contains $v_i = u$ by construction and $v_i^I = u$ by the induction hypothesis. So property P is satisfied by all states in layer $i+1$.

Place a white pebble on j All states in layer i are unlabelled (they are "guess" states) and all edges from layer i to layer $i+1$ are unlabelled. From each state s in layer i, add an unlabelled outgoing edge from s to each s' in layer $i+1$ that satisfies $tag(s') = tag(s) \cup \{v_j = v\}$ for some $v \in [k]$. Note that the $(i+1)^{\text{th}}$ layer has at most $k^{p_i+1} = k^{p_{i+1}}$ states. Consider any input I with an accepting computation path through some state s' in layer $i+1$. Now let us assume that $tag(s')$ contains $v_i = u$ for some $i \neq j$ and some $u \in [k]$. Let s be the previous state in the accepting computation path for I. Then $tag(s)$ contains $v_i = u$ by construction and $v_i^I = u$ by the induction hypothesis. Assume that $tag(s')$ contains $v_j = v$ for some $v \in [k]$. The proof that $v_j^I = v$ is deferred until we present the construction corresponding to the removal of this white pebble.

Remove a black pebble from j All states in layer i are unlabelled (they are "forget" states). From each state s in layer i such that $tag(s)$ contains $v_j = v$ for some $v \in [k]$, we add an unlabelled edge to each s' in layer $i+1$ that satisfies $tag(s') = tag(s) - \{v_j = v\}$. Note that the $(i+1)^{\text{th}}$ layer has at most $k^{p_i - 1} = k^{p_{i+1}}$ states. The fact that all states in layer $i+1$ satisfy the property P follows trivially from the induction hypothesis.

Remove a white pebble from j For each state s where $tag(s)$ contains $v_j = v$ for some $v \in [k]$, we label s with ℓ_j if j is a leaf node. Otherwise j is an internal node and we label s with $f_j(x, y)$ where x and y are the values of $2j$ and $2j+1$ in $tag(s)$. We add a single outgoing edge from s labelled v to the state s' in layer $i+1$ such that $tag(s') = tag(s) - \{v_j = v\}$. Note that the $(i+1)^{\text{th}}$ layer has at most $k^{p_i - 1} = k^{p_{i+1}}$ states. We now present the proof that was deferred in the case corresponding to the placement of a white pebble. Let k be the layer where this white pebble was placed on node j (Note that this layer k is the same as layer $i+1$ from case corresponding to the placement of a white pebble). Assume that s is a state in layer k such that $tag(s)$ contains $v_j = v$ for some $v \in [k]$. Let I be any input with an accepting computation path through s. We have to prove that $v_j^I = v$. Let s' be the state in layer i that this accepting computation path passes through. Then by our construction and the fact that node j was not touched by the pebbling from C_k to C_i, we have that $tag(s')$ contains $v_j = v$. Now the state s' queries the node j and the only outgoing edge from s' is labelled v. This implies that $v_j^I = v$. The fact that all states in layer $i+1$ satisfy property P follows trivially from the induction hypothesis.

Slide a black pebble from j **to its parent** $j' = \lfloor j/2 \rfloor$ Each state s in layer i is labelled $f_{j'}(x, y)$ where x and y are the values of j and its sibling in $tag(s)$. Add k outgoing edges from s labelled 0 to $k-1$ such that the edge labelled v is directed to s' in layer $i+1$ such that $tag(s') = tag(s) \cup \{v_{j'} = v\} - \{v_j = x\}$ (assuming value of j is x). Note that the $(i+1)^{\text{th}}$ layer has at most $k^{p_i} = k^{p_{i+1}}$ states. The fact that all states in layer $i+1$ satisfy property P follows from arguments similar to the ones given before.

Slide a black pebble to the root node Each state s in layer i is labelled $f_1(x, y)$ where x and y are the values of nodes 2 and 3 in $tag(s)$. Add an outgoing edge labelled 1 to s' in layer $i+1$ such that $tag(s') = tag(s) - \{v_2 = x\}$ (assuming that the black pebble was slid from node 2 to the root). Note that the $(i+1)^{\text{th}}$ layer has $k^{p_i - 1} \leq k^{p_{i+1}}$ states. The fact that all states in layer $i+1$ satisfy property P follows from arguments similar to the ones given before.

We can remove the unlabelled states with unlabelled outgoing edges by the following procedure. If there is an edge labelled v from some state s to an unlabelled state s' with e outgoing edges, then remove the state s' and add e outgoing edges labelled v from s to the out-neighbors of s'. Let us call this new BP B'. We claim that B' is a RONTBP computing $\mathsf{BT(h,k)}$. It is easy to see that B' computes the same function as the BP B. Each layer in B' corresponds to the placement of some black pebble or to the removal of some white pebble from the tree. Since $C_1, \ldots, C_t$ is a read-once pebbling, this implies that the number of layers in B' is exactly the number of nodes in the tree. So the number of layers in B' is $2^h - 1$. The number of states in a layer in B' is exactly the number of states in the corresponding layer in B. So any layer in B' has at most $k^{\lceil h/2 \rceil + 1}$ states. So the number of non-final states in B' is at most $(2^h - 1)k^{\lceil h/2 \rceil + 1}$. The BP B' is read-once since B' is a layered BP and for any node in the tree there is exactly one layer in B' that queries that node. To see that B' is thrifty. Consider any state s in B' that is labelled $f_j(x, y)$ for some j and some $x, y \in [k]$. Note that an input I can reach state s in B' iff I can reach the corresponding state in B assuming that s corresponds to a labelled state in B. By construction, we have that $tag(s)$ contains $v_{2j} = x$ and $v_{2j+1} = y$. Therefore any input I reaching the state s must have $v^I_{2j} = x$ and $v^I_{2j+1} = y$ by property P. So B' is thrifty. □

We now prove tight lower bounds for size of RONTBPs solving $\mathsf{BT(h,k)}$. The idea is to associate the computation of a RONTBP with a whole black-white pebbling. We associate a whole black-white pebbling configuration with each state in the RONTBP such that if we take an accepting computation path of any instance in E, the sequence of pebbling configurations along the computation path is a valid pebbling of $\mathsf{T_h}$. Then we proceed to show that if we consider a state s that has at least $\lceil h/2 \rceil + 1$ pebbles on a computation path (such a state exists on any accepting computation path), then the number of inputs reaching s must be small. In particular, for any input I on an accepting computation path reaching s, the values of pebbled nodes can be inferred from the state s and the values of unpebbled nodes. This shows that the state s must encode an element from a set of k^p values where p is the number of pebbled nodes. The lower bound follows.

The following definition tells us how to extract a whole black-white pebbling from a RONTBP.

Definition 19 (Pebbling Configuration at a State)**.** *Let B be a RONTBP solving* $\mathsf{BT(h,k)}$ *and let s be a state in B. Let $I \in E$ be arbitrary and let $C(I)$ be an arbitrary accepting computation path for I in B. Then the pebble value of a non-root node i with parent $i' = \lfloor i/2 \rfloor$ in the pebbling configuration associated with the state s is defined as*

- *If the state querying node i' comes after s (or i' is queried by s) and the state querying node i comes before s in $C(I)$, then the node i is black pebbled at state s.*
- *If the state querying node i comes after s and the state querying node i' comes before s in $C(I)$ (with one of them possibly queried at s), then the node i is white pebbled at state s.*
- *Otherwise, the node i is unpebbled at state s.*

We now prove that the black-white pebbling extracted from a RONTBP using Definition 19 is a valid read-once whole black-white pebbling of $\mathsf{T_h}$.

Lemma 1. *Let B be a RONTBP solving* $\mathsf{BT(h,k)}$. *Let $I \in E$ and let $C(I)$ be an accepting computation path for I in B. Then the whole black-white pebbling obtained by considering the pebbling configurations associated with the states on $C(I)$ (as defined in Definition 19) in order is a valid read-once whole black-white pebbling of* $\mathsf{T_h}$.

Proof. The start state clearly corresponds to the empty pebbling configuration. We can assume that the root node of the tree is pebbled and unpebbled at the state immediately following the state in $C(I)$ querying the root node (This will not affect the lower bound since the value at the root node is always 1 for any input in E). Now we have to prove that when a black pebble is placed or a white pebble is removed from some non-leaf node i, both its children are pebbled.

A black pebble is placed on node i at state s Let t be the state immediately preceeding s in $C(I)$. The state t queries the node i according to Definition 19. Now if

$2i$ is queried before the state t in $C(I)$, then $2i$ is black pebbled at t. If the node $2i$ is queried after the state t in $C(I)$, then $2i$ is white pebbled at t. Similarly, we can prove that the node $2i + 1$ is also pebbled at the state t.

A white pebble is removed from node i at state s We can prove that both $2i$ and $2i + 1$ are pebbled at the state immediately preceeding the state s in $C(I)$ by using the argument used in the previous case.

Since B is a RONTBP, each node is queried exactly once in $C(I)$. Note that a black pebble is placed on a node or a white pebble is removed from a node only when that node is queried. Therefore the pebbling is read-once. □

Now we show that the pebbling configuration at some state s defined above is independent of the input I and the accepting computation path $C(I)$ that passes through state s. In other words, this shows that the pebbling configuration at a state only depends upon the state s. Note that if there are no accepting computation paths passing through a state s in a RONTBP, then that state can be deleted from the RONTBP.

Lemma 2. *Let B be a RONTBP solving* $\mathsf{BT(h,k)}$*. Then the pebbling configuration at any state s in B depends only on s. In particular, it is independent of any input I and any accepting computation path $C(I)$ used to define it.*

Proof. Let I and I' be two inputs in E with accepting computation paths C and C' passing through the state s. Consider an arbitrary node i and we argue that the pebble value of node i with respect to C is the same as pebble value of node i with respect to C'. Let $i' = \lfloor i/2 \rfloor$ be the parent of i. We consider three cases based on the pebble value of node i at s with respect to C.

Node i is black pebbled By Definition 19, we have a state r querying i before s and a state t querying i' after s in the computation path C (It is possible that $t = s$). Now the state r' querying i on C' must precede state s in computation path C'. Otherwise the path $start \mapsto r \mapsto s \mapsto r' \mapsto accept$ is a path in B that queries node i twice which is not possible since B is a RONTBP. Similarly, the state t' querying i' must come after s on C' (It is possible that $t = t' = s$).

Node i is white pebbled By Definition 19, we have a state r querying i' before s and a state t querying i after s in the computation path C (It is possible that $t = s$ or $r = s$). Now the state t' querying i on C' must come after the state s (or $t' = s$ if $t = s$) in the computation path C'. Otherwise the path $start \mapsto t' \mapsto s \mapsto t \mapsto accept$ is a path in B that queries node i twice which is not possible since B is a RONTBP. Similarly, the state r' querying i' must come before (or at) s on C'.

Node i is not pebbled We have three cases to consider.

Nodes i and i' are queried before s On C' both i and i' must be queried before the state s as otherwise we can construct a path from start state to accepting state that queries some node at least twice.

Nodes i and i' are queried after s On C' both i and i' must be queried after s as otherwise we can construct a path from start state to accepting state that queries some node at least twice.

Node i is queried at s and i' is queried after s On C', the node i is queried at s and the node i' must be queried after s. If i' is queried before s on C', then we can construct a path from the start state to the accepting state that queries i' twice.

The lemma follows. $\square$

The following lemma is the key ingredient in our lower bound proof.

Lemma 3. *Let B be a RONTBP solving* $\mathsf{BT}(\mathsf{h},\mathsf{k})$ *and let s be a state in B. Let p be the number of pebbled non-root nodes in the pebbling configuration associated with s. Then the number of inputs in E with an accepting computation path through s is at most k^{N-p}.*

Proof. Consider $I, I' \in E$ such that both I and I' has accepting computation paths that pass through s. We claim that if j is an arbitrary non-root node that is pebbled in the pebbling configuration associated with s, then $v_j^I = v_j^{I'}$. The claim implies that the number of inputs in E with an accepting computation path through s is at most k^{N-p}.

We use proof by contradiction. Suppose that there exists I and I' in E and suppose that C and C' are the accepting computation paths for I and I' (resp.) that passes through s. Let C_1, C_1' and C_2, C_2' be the segments of C and C' before and after s respectively. Suppose that I and I' differ in the value of a black (resp. white) pebbled node $2i$ (We are assuming wlog that j is a left child of some node) and x and x' are the values of node $2i$ in I and I' respectievely. Then the computation paths C and C' are as shown in Figure 3.1

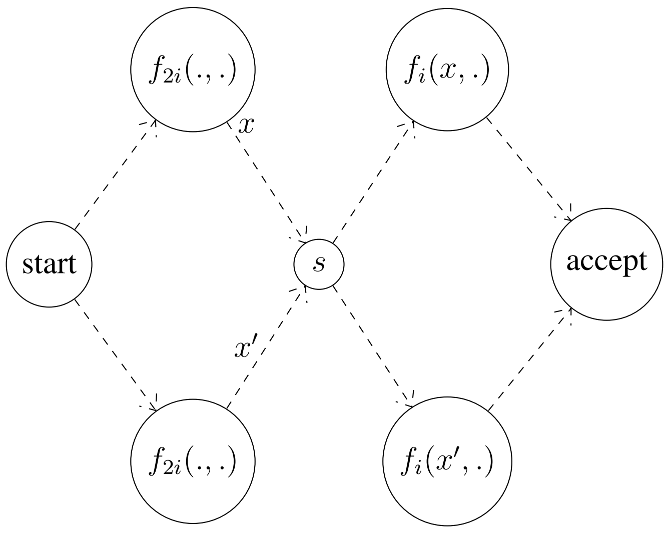


Figure 3.1: Structure of computation paths in a RONTBP for two inputs that differ in the value of a black pebbled node

(resp. Figure 3.2) by Proposition 1. Now since B is a RONTBP, the nodes queried in C_1 are C_2' are disjoint and therefore we can construct an input J with the accepting computation path C_1C_2'. But this path makes a non-thrifty query. □

Theorem 3. *Any RONTBP solving* $\mathsf{BT}(\mathsf{h},\mathsf{k})$ *must have at least* $k^{\lceil h/2 \rceil}$ *states.*

Proof. We give a proof using the entropy method. Let B be a RONTBP solving $\mathsf{BT}(\mathsf{h},\mathsf{k})$. Our input set is the set E given in Definition 17. Now for each input $I \in E$, we choose an arbitrary accepting computation path $C(I)$ and map I to a state s in $C(I)$ such that the whole black-white pebbling configuration associated with s has at least $\lceil h/2 \rceil$ pebbles on non-root nodes. Such a state exists by the whole black-white

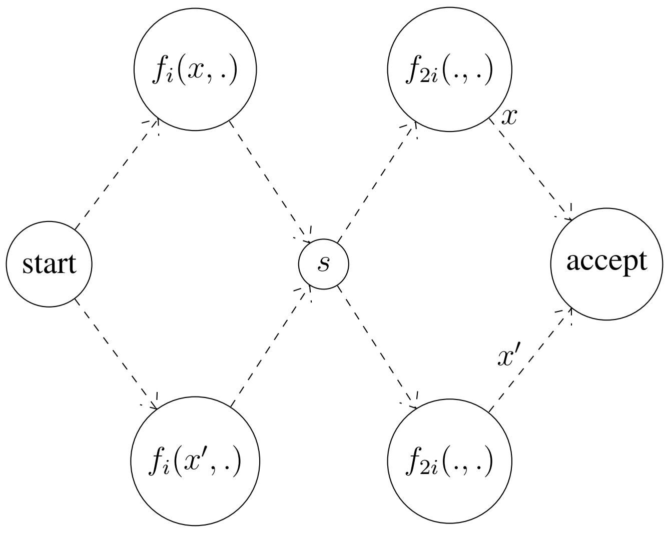


Figure 3.2: Structure of computation paths in a RONTBP for two inputs that differ in the value of a white pebbled node

pebbling lower bounds given by Cook *et al.* (2012) Vanderzwet (2013). Now we can conclude by Lemma 3 that there are at most $k^{N-\lceil h/2 \rceil}$ inputs in E reaching s on an accepting computation path in the RONTBP B. Therefore, there are at least $k^{\lceil h/2 \rceil}$ states in B. □

From Theorem 2 and Theorem 3, we obtain the tight bound $\widetilde{\theta}\left(k^{\lceil h/2 \rceil}\right)$ for RONTBPs solving $\mathsf{BT(h,k)}$.

3.4 Bitwise Independent Branching Programs

Before proving the main result of this chapter, we have to introduce one more definition. That of bitwise independent branching programs. Intuitively, these are branching programs solving TEP that cannot store any information that correlates values at different nodes or even different bits of the value of one node. For ex., a branching program at some state could satisfy the invariant that for all inputs with an accepting computation path through that state, the value at node i is the same as the value at node j. Such a branching program would not be bitwise independent.

Definition 20. *Let $k = 2^\ell$ and let B be a nondeterministic thrifty BP solving* $\mathsf{BT(h, k)}$. *Then B is called* bitwise independent *if and only if there exists an encoding function $\phi : [k] \mapsto \{0,1\}^\ell$ such that for every state s in B the following two conditions are satisfied.*

$$F_s = \mathop{\times}_{i=2}^{N+1} \phi^{-1} \left(\mathop{\times}_{j=1}^{\ell} (\pi(F_s, i))_j \right)$$
$$A_s = \mathop{\times}_{i=2}^{N+1} \phi^{-1} \left(\mathop{\times}_{j=1}^{\ell} (\pi(A_s, i))_j \right)$$

Here the outer Cartesian product is the normal Cartesian product and the inner one concatenates all the bits after forming the Cartesian product. When k is not a power of 2, we consider the largest power of 2 smaller than k. Let this be 2^ℓ. Then B is bitwise independent if and only if the sub-BP B' of B obtained by considering only inputs where all values are from $[2^\ell]$ is bitwise independent.

We now prove the main result of this chapter. That is, BINTBPs solving TEP must have super-polynomial size.

We prove upper bounds for BINTBP by showing that BINTBPs can implement fractional black-white pebbling of $\mathsf{T_h}$ to solve $\mathsf{BT(h, k)}$

Theorem 4. $\mathsf{BT(h, k)}$ *can be solved by a BINTBP using $\widetilde{O}\left(k^{h/2}\right)$ states.*

Proof. Cook et. al. Cook *et al.* (2012) describes an NTBP B that solves $\mathsf{BT(h, k)}$ using $\widetilde{O}\left(k^{h/2}\right)$ states (See Theorem 3.4 (ii) in Cook *et al.* (2012)). The NTBP B implements a fractional black-white pebbling of $\mathsf{T_h}$ similar to the way in which the BP obtained in the proof of Theorem 2 implements a read-once whole black-white pebbling. The NTBP B is a layered BP where each layer corresponds to a fractional pebbling configuration of $\mathsf{T_h}$. If we consider an arbitrary state s in B and an arbitrary node i in $\mathsf{T_h}$ where i has black pebble value b_i and white pebble value w_i in the pebbling configuration corresponding to the state s, then the state s will remember a fraction $b_i + w_i$ of the $\log(k)$ bits of the value of node i. In this case, the set $tag(s)$ for a state s in B specifies for each node i, $b_i \log(k)$ black-pebbled bits and $w_i \log(k)$ white-pebbled bits. Here $b_i \log(k)$ bits of v_i are computed (i.e., for every input I reaching this state, there is an earlier state in that input's computation path that queried the node i and the result of that query matched these bits) and $w_i \log(k)$ bits are guessed (i.e., for every input I with an accepting computation path through this state, there is a state later in its computation path that queries the node i and rejects immediately if the result of that query does not match these bits). The key property P satisfied by all states s in B is

- The set of all inputs $I \in E$ reaching s is exactly the set of all inputs $I \in E$ for which v_i^I matches $b_i \log(k)$ black-pebbled bits specified by $tag(s)$ for all nodes i.
- The set of all inputs $I \in E$ with an accepting computation path through s is exactly the set of all inputs $I \in E$ for which v_i^I matches $(b_i + w_i)\log(k)$ black and white pebbled bits specified by $tag(s)$ for all nodes i.

The proof of property P is a straightforward generalization of the proof of property P in the proof of Theorem 2.

The claim that this NTBP B is a BINTBP follows directly from property P. To see this, consider any state s in the BP B and some node i in the tree that has a black pebble value of b_i and a white pebble value of w_i. Now if we consider the set F_s for any state s, then the fraction of bits corresponding to b_i can take only one particular value in F_s (The value specified in $tag(s)$). The rest of the bits can take all possible combinations. If we consider A_s, then the fraction of bits corresponding to w_i are also fixed (in addition to the fraction of bits corresponding to b_i). The value of fraction of bits corresponding to w_i will be the value guessed by the BP B (These values are specified as white-pebbled bits in $tag(s)$). Even though there are inputs reaching the state s that does not match these $w_i \log(k)$ bits, these inputs will be rejected later in the computation path when the BP B queries node i and finds out the mismatch. So these inputs are not in A_s. The rest of the bits can take all possible combinations. $\square$

Similar to Definition 19, we now define the fractional black-white pebbling configuration at a state in the BINTBP.

Definition 21 (Fractional Black-White Pebbling Configuration at a State)**.** *Let B be a BINTBP solving* $\mathsf{BT(h,k)}$ *and let s be a state such that for some input $I \in E$ s has at least one accepting computation path for I passing through s. Then for any non-root* *node i, we define the black and white pebble values for the configuration at state s as follows.*

$$b(i,s) = 1 - \log_k |\pi(F_s, i)|$$
$$w(i,s) = \log_k \frac{|\pi(F_s, i)|}{|\pi(A_s, i)|}$$

Notice that in a minimal BINTBP, any state s must have at least one accepting computation path passing through it. Otherwise, the state s can be removed. Note that the pebbling configuration only depends on state s by definition.

We now claim that Definition 21 of pebble values satisfy the restrictions imposed on pebble values by (3.1).

Claim 1. *For any non-root node i and state s, $0 \leq b(i,s), w(i,s) \leq 1$.*

Claim 2. *For any non-root node i and state s, $b(i,s) + w(i,s) \leq 1$.*

The following claim establishes the fact that if the total pebble value of the tree (in non-root nodes) is high at a state, then there are only a few inputs on an accepting computation path reaching that state. In other words, if the pebble value at a point of the computation is high, then the entropy at that point is low.

Claim 3. *If the total pebble value of the non-root nodes of the tree at a state s is p, then the number of inputs $I \in E$ reaching s on an accepting computation path is k^{N-p}.*

Proof. Consider a particular non-root node i. Assume that the total pebble value at i is p_i. From this we have $1 - \log_k |\pi(F_s, i)| + \log_k \frac{|\pi(F_s,i)|}{|\pi(A_s,i)|} = p_i$. Therefore $|\pi(A_s, i)| = k^{1-p_i}$. Now by simple counting the total number of inputs on an accepting computation path is $k^{\sum_{i=2}^{N+1}(1-p_i)} = k^{N-p}$. $\square$

We now identify the fractional black-white pebbling of $\mathsf{T_h}$ on an accepting computation path C for an input $I \in E$. First, we identify certain *critical* states in the path C. The pebbling will satisfy the criteria that the pebbling configuration changes (i.e., pebbling moves happen) only at critical states. Then we will show that these pebbling configurations always underestimate the pebble values of nodes given by Definition 21.

Definition 22 (Critical States for Nodes)**.** *The critical state for the root node is the last state querying the root node. Every non-root node j may have multiple critical states. Let s denote* a *critical state of parent of j. If $b(j, s) > 0$, then the last node querying node j before s is a critical state for j. If $w(j, s) > 0$, then the first node querying node j after s is a critical state for j.*

We will follow the convention that the start state and accepting state are critical.

For the lower bound proof we will work with the following fractional black-white pebbling along an arbitrary accepting computation path for an input in E. This pebbling satisfies the condition that pebble values are always underestimated.

Fractional Black-White Pebbling along Critical States We now define the pebbling along critical states on an accepting computation path of input I. The black pebble value of the root node becomes 1 immediately after its critical state and it is immediately unpebbled. Now we define the pebble values of an arbitrary non-root node j. Let s' be a critical state for j', the parent of j. If $b = b(j, s') > 0$ (We say that s' needs this black pebble at j), then this black pebble value must have increased from 0 to b at some point of computation. Now consider the critical state s for j before s' as per Definition 22. The black pebble value of node j is increased from 0 to b at the critical state immediately following s. This state s must exist as otherwise we have $b = 0$. Similarly, if $w = w(j, s') > 0$ (We say that s' needs this white pebble at j), then this white pebble value must decrease from w to 0 at some point of computation. Now consider the critical state s for j after s' as per Definition 22. The white pebble value is reduced from w to 0 at the critical state immediately following s. This state s must exist as otherwise we can construct an input using bitwise independence that differs from I only in the value of node j that has an accepting computation path with a non-thrifty query. We decrease the black pebble values of all nodes to the minimum value required further forward in the computation path and increase the white pebble values only at the critical state that needs them.

The following claims about the validity of the starting and ending pebbling configurations are easily proved.

Claim 4. *The start state has an empty pebbling configuration.*

Claim 5. *The accepting state has an empty pebbling configuration.*

The following lemmas establish the fact that the pebbling sequence along critical states is a valid pebbling sequence.

Lemma 4. *Let s be a critical state for node j, then both of j's children are fully pebbled at s.*

Proof. Let s query $f_j(u, v)$. We have $\pi(A_s, 2j) = \{u\}$ (and $\pi(A_s, 2j + 1) = \{v\}$) by the thrifty property. Then $b(2j, s) + w(2j, s) = 1 - \log_k |\pi(F_s, 2j)| + \log_k \frac{|\pi(F_s,2j)|}{|\pi(A_s,2j)|} = 1$ (and similarly for $2j + 1$). $\square$

Lemma 5. *If the black pebble value of node j is increased or the white pebble value of node j is decreased at state s, then both its children are fully pebbled at the critical state immediately before s.*

Proof. For a node j, the black pebble value is increased or the white pebble value is decreased only at the critical state immediately following a critical state for j. By Lemma 4 both children of node j are fully pebbled at this critical state. $\square$

The following is our key technical lemma and establishes the fact that the pebbling values defined for the critical states never overestimate the actual pebbling values of nodes.

Lemma 6. *Let b and w be the pebble values at state s for an arbitrary non-root node $2j$ with respect to an arbitrary accepting computation path for some input in E, then $b \leq b(2j, s)$ and $w \leq w(2j, s)$.*

Proof. The proof is divided into two parts. First, we show that the black pebble values are never overestimated. Then we show that white pebble values are never overestimated.

We consider an arbitrary state s at which the black pebble value of node $2j$ is defined as $b > 0$. Note that the black pebble value of a non-root node $2j$ is non-zero if and only if there exists a critical state for the parent of $2j$ at which the actual pebble value of $2j$ is b. Therefore, there exists a state s_{2j} that is a critical state for $2j$ before s and s_j that is a critical state for j, the parent of $2j$, after s (with $s = s_j$ possibly.). Now suppose that the actual black pebble value for node $2j$ at state s is $b(2j, s)$ and that $b(2j, s) < b$.

$$\begin{aligned} 1 - \log_k |\pi(F_s, 2j)| &< b \\ \implies |\pi(F_s, 2j)| &> k^{1-b} \end{aligned}$$

Now by the independence assumption we may conclude that there are more than k^{1-b} inputs that differ only at the value of node $2j$ reaching s. By the definition of critical states, there does not exist any node querying $2j$ in $C(I)$ from s to s_j. All these inputs can follow the same path to the critical state s_j. Therefore, the black pebble value is $b(2j, s_j) < b$, a contradiction.

It remains to prove that white pebble values are never overestimated. We will prove that the white pebble value of a node $2j$ is at least the estimated value w between all states from s_j to s_{2j} (both inclusive). Here s_j is a critical state for j at which node $2j$ acquired a white pebble value of w and s_{2j} is the critical state for $2j$ after which this pebble value is removed. In order to prove this, it is sufficient to prove that the ratio $\frac{f'}{a'} = \frac{|\pi(F_{s'},2j)|}{|\pi(A_{s'},2j)|}$ for any state s' is greater than the corresponding ratio $\frac{f}{a}$ at state s_j, where s' is a state on $C(I)$ in the segment from s_j to s_{2j}. By the independence argument, we have $f' \geq f$ by taking projections of all f inputs that differ from I only at node $2j$. We will show that if $a' > a$, then $f' > f$ by an appropriate amount so that the ratio is not reduced.

Since the white pebble value is acquired at state s_j, we have $w(2j, s_j) = w$. Now consider a state s' (Possibly equal to s_{2j}) on the segment of the computation path $C(I)$ between s_j and s_{2j}. Our aim is to prove that $w \leq w(2j, s')$. Let $f = |\pi(F_{s_j}, 2j)|$, $f' = |\pi(F_{s'}, 2j)|$, $a = |\pi(A_{s_j}, 2j)|$ and $a' = |\pi(A_{s'}, 2j)|$. First of all note that $f' \geq f$ since there are no nodes querying $2j$ from s_j to s_{2j} and the independence property guarantees f inputs that differ from I only at node $2j$ will reach s'. Now we will show that whenever $a' > a$, f' is greater than f by the same multiplicative factor. Note that both f and a are powers of two. By the assumption of bitwise independence, we can partition bits of node $2j$ into "fixed" bits and "unfixed" bits for any F_s (and A_s). The only way to add elements to these sets are by unfixing bits. Let us assume that exactly one more bit became unfixed in $\pi(A_{s'}, 2j)$. So $a' = 2a$.

Let r' be a value in $\pi(A_{s'}, 2j) \setminus \pi(A_{s_j}, 2j)$. We claim that $r' \notin \pi(F_{s_j}, 2j)$. We will prove this by contradiction. Suppose $r' \in \pi(F_{s_j}, 2j)$, then by the independence property there is an input J which is the same as I except that $v^J_{2j} = r'$ reaches s' through s_j. Since $r' \in \pi(A_{s'}, 2j)$, there is an accepting path for J through s_j. This accepting path is obtained by using the independence property of $A_{s'}$ and the fact that an accepting computation for I passes through s'. But this path makes a non-thrifty query at s_j. Therefore $r' \notin \pi(F_{s_j}, 2j)$ as claimed. Since $r' \in \pi(F_{s'}, 2j)$, at least one bit must have become unfixed. But this implies $f' \geq 2f$. This proof can be easily extended to the case where $a' = 2^m a$ for any m. $\square$

Now we prove tight size lower bounds for BINTBPs solving $\mathsf{BT(h,k)}$.

Theorem 5. *If B is a BINTBP solving* $\mathsf{BT(h,k)}$*, then B has at least $\frac{1}{2}k^{h/2}$ states.*

Proof. Assume that k is a power of two. We now apply the entropy method. Our input set is the set E described in Definition 17. We now describe our distribution function f. Recall that each instance I in E is a "yes" instance and therefore guaranteed to have an accepting computation path $C(I)$ in B. As we have already seen, we can identify a sequence of critical states in $C(I)$ and associate a fractional black-white pebbling configuration with each critical state such that the sequence of fractional black-white pebbling configurations form a valid fractional black-white pebbling of $\mathsf{T_h}$ (See Claims 1, 2, 4, 5, and Lemmas 4 and 5). But we know that any valid fractional black-white pebbling of $\mathsf{T_h}$ must have a configuration with at least $h/2$ pebbles on non-root nodes Vanderzwet (2013). Let s be the critical state in $C(I)$ that corresponds to this configuration. Our distribution function f maps I to s. Now consider an arbitrary state s in $range(f)$ and consider the set $G_s = f^{-1}(s)$. By Claim 3, we have $|G_s| \leq k^{N-h/2}$. It follows that B has at least $k^{h/2}$ states.

When k is not a power of two, we consider the highest power of two (2^ℓ) smaller than k. Consider the sub-BP of B that solves $\mathsf{BT(h,k)}$ when the values are from the set $[2^\ell]$. By Definition 20 and the lower bound when k is a power of two, we have that this sub-BP of B has at least $2^{\ell^{h/2}} > \frac{1}{2}k^{h/2}$ states. $\square$

From Theorem 4 and Theorem 5, we conclude that BINTBPs solving $\mathsf{BT(h,k)}$ has size $\widetilde{\theta}\left(k^{h/2}\right)$.

3.5 Universality of the Entropy Method

We now analyze the entropy method and show that it can be used to prove any lower bound.

Theorem 6. *Let f be a boolean function with $BP(f) \geq s(n)$. If B is an optimal BP for f, then there exists $A \subseteq \{0,1\}^n$ and $F : A \mapsto States(B)$ such that $|A| \geq s(n)$ and F is one-to-one.*

Proof. Let F_s denote the set of all inputs reaching a state s. We know that there are $s(n)$ states in B. For each state i, we will choose an input x_i such that x_i reaches state i and $x_i \neq x_j$ if $i \neq j$. Then we set the distribution function to $f(x_i) = i$ for each x_i and A is the set of all x_i's. Consider the bipartite graph where one partition is labelled by elements of A and other partition by states in B. We add the edge $\{x_i, j\}$ if and only if the computation path of the input x_i passes through j. We will now show that Hall's condition is satisfied on this bipartite graph. For contradiction assume that there are k distinct states $s_1, \ldots, s_k$ such that $|\bigcup_i F_{s_i}| \leq k$. We now merge all s_i into a single state and replace it with a binary decision tree where each input in $F = \bigcup_i F_{s_i}$ reaches a unique leaf in the decision tree. Such a decision tree will have at most $|F| - 1$ query states. The leaf nodes of the representation tree are merged into accept or reject states according to the function computed by B. Note that this can be done without violating the correctness of BP since only one input reaches a leaf in the decision tree. The new BP has size $\leq s(n) - k + k - 1 = s(n) - 1$. A contradiction to the optimality of B. □

For completeness, we list other results proving lower bounds for BPs solving TEP.

1. Liu (2013) proved super-polynomial size lower bounds for RONTBPs simultaneously. Liu (2013) also proved super-polynomial size lower bounds for the stronger semantic read-once NTBPs. He also proved super-polynomial size lower bounds for read-once deterministic BPs without the thrifty restriction. All proofs are obtained using pebbling arguments similar to the ones that we saw in this chapter.
2. Iwama and Nagao (2014) also proved super-polynomial size lower bounds for read-once deterministic BPs without thrifty restriction. It is notable that their argument does not use any notion of pebbling games. However, their proof is also an application of the entropy method.

3.6 Reversible Pebbling

The reversible pebble game (Bennett (1989)) is a combinatorial game on directed acyclic graphs (DAG) that is used to design space efficient reversible algorithms[2]. This game is a variant of other pebble games that we saw in this chapter such as the black pebble game (Walker and Strong (1973)) and black white pebble game (Cook and Sethi (1974)) studied in connection to space complexity. Recently, Chan (2013) showed that the reversible pebble game on DAGs is the same as two other pebble games defined on DAGs revealing a surprising connection between them. An interesting fact about black white pebbling game is that it is the same as min-cut linear arrangements on trees (Yannakakis (1985)). However, no such connections are known between reversible pebble game and classical graph parameters. Our objective is to study the reversible pebble game on trees and explore its connections to other graph theoretic parameters.

We first define the reversible pebble game on trees. Given a directed rooted tree T with root node r, the goal of the game is to place a pebble on r. Initially, there are no pebbles on T. A pebble can be placed or removed from a node if and only if all its immediate predecessors have pebbles on them. A special case of this rule is that a pebble can be placed or removed from a leaf node at any time. A pebble removed from a node can be placed on another node at a later time. i.e., pebbles can be reused. The goal of the game is to pebble the node r using the minimum number of pebbles. Clearly, this value will depend on (and only on) the structure of the tree T. This parameter is called the reversible pebbling number of T, denoted $R(T)$. This game has connections to the amount of space used by reversible algorithms and depth of Boolean circuits, a matter we will not pursue in this thesis. Instead, we look at the combinatorial and complexity aspects of the game itseld. We ask whether given a tree, can $R(T)$ be computed in polynoimal time? How is $R(T)$ related to other well-known graph parameters such as colouring, matching etc.? To answer these questions, we show that the reversible pebble game number of directed trees is always one more than the edge rank colouring of the underlying undirected tree (Komarath *et al.* (2015*a*)). i.e., surprisingly the parameter $R(T)$ does not depend on the direction of edges (and therefore the root node) of the tree, and only on its underlying undirected structure. The problem of computing the

[2]We will discuss reversible algorithms since we are only interested in the combinatorics of the reversible pebble game in this thesis

reversible pebble game number is PSPACE-complete for DAGs (Chan (2013)). Prior to our work, it was not known whether this problem was in P for trees. Our result implies that the reversible pebble game number of trees can be computed in linear time (Komarath *et al.* (2015*a*), Lam and Yue (2001)).

3.6.1 Basic Definitions

We assume familiarity with basic definitions in graph theory, such as those found in West (2000). A directed tree $T = (V, E)$ is called a *rooted directed tree* if there is an $r \in V$ such that r is reachable from every node in T. The node r is called the root of the tree.

An *edge rank coloring* of an undirected tree T with k colours $\{1, \dots, k\}$ labels each edge of T with a colour such that if two edges have the same colour i, then the path between these two edges consists of an edge with some colour $j > i$. The minimum number of colours required for an edge rank colouring of T is denoted by $\chi'_e(T)$.

Definition 23 (Reversible Pebbling Bennett (1989))**.** *Let G be a rooted DAG with root r. A* reversible pebbling configuration *of G is a set $P \subseteq V$ (the set of pebbled vertices). A* reversible pebbling *of G is a sequence of reversible pebbling configurations $P = (P_1, \dots, P_m)$ such that $P_1 = \phi$ and $P_m = \{r\}$ and for every $i, 2 \leq i \leq m$, we have*

1. *$P_i = P_{i-1} \cup \{v\}$ or $P_{i-1} = P_i \cup \{v\}$ and $P_i \neq P_{i-1}$ (Exactly one vertex is pebbled/unpebbled at each step).*
2. *All in-neighbours of v are in P_{i-1}.*

The number m is called the time taken by the pebbling P. The number of pebbles or space used in a reversible pebbling of G is the maximum number of pebbles on G at any time during the pebbling. The persistent reversible pebbling number *of G, denoted by $R^{\bullet}(G)$, is the minimum number of pebbles required to persistently pebble G.*

A closely related notion is that of visiting *reversible pebbling, where the pebbling P satisfies (1) $P_1 = P_m = \phi$ and (2) there exists a j such that $r \in P_j$. The minimum number of pebbles required for a visiting pebbling of G is denoted by $R^{\phi}(T)$.*

It is easy to see that $R^{\phi}(G) \leq R^{\bullet}(G) \leq R^{\phi}(G) + 1$ for any DAG G. We now define the Dymond-Tompa pebble game.

Definition 24 (Dymond-Tompa Pebble Game Dymond and Tompa (1985))**.** *Let G be a DAG with root r. A Dymond-Tompa pebble game is a two-player game on G where the two players, the pebbler and the challenger takes turns. In the first round, the pebbler pebbles the root node and the challenger challenges the root node. In each subsequent round, the pebbler pebbles a (unpebbled) node in G and the challenger either challenges the node just pebbled or re-challenges the node challenged in the previous round. The pebbler wins when the challenger challenges a node v and all in-neighbours of v are pebbled.*

The Dymond-Tompa pebble number of G, denoted $DT(G)$, is the minimum number of pebbles required by the pebbler to win against an optimal challenger play.

The Raz-Mckenzie pebble game is also a two-player pebble game played on DAGs. The optimal value is denoted by $RM(G)$. A definition for the Raz-Mckenzie pebble game can be found in Raz and McKenzie (1999). Although the Dymond-Tompa game and the reversible pebbling game look quite different. The following theorem reveals a surprising connection between them.

Theorem 7 (Theorems 6 and 7, Chan (2013))**.** *For any rooted DAG G, we have $DT(G) = R^{\bullet}(G) = RM(G)$.*

Theorem 7 is proved by showing that certain restricted pebbling strategies called upstream strategies are enough to achieve optimality with respect to the number of pebbles. Our next goal is to understand this notion as we will use this in our proofs.

Definition 25. *(Effective Predecessor Chan (2013)) Given a pebbling configuration P of a DAG G with root r, a node v in G is called an* effective predecessor *of r iff there exists a path from v to r with no pebbles on the vertices in the path (except at r).*

Lemma 7 defines upstream pebbling strategies and proves that they are enough to achieve optimality.

Lemma 7 (Claim 3.11, Chan (2013))**.** *Let G be any rooted DAG. There exists an optimal pebbler strategy for the Dymond-Tompa pebble game on G such that the pebbler always pebbles an effective predecessor of the currently challenged node.*

The height or depth of a tree is defined as the maximum number of nodes in any root to leaf path. We denote by Ch_n the rooted directed path on n nodes with a leaf as the root. We denote by Bt_h the the complete binary tree of height h. We use $root(Bt_h)$ to refer to the root of Bt_h. If v is any node in Bt_h, we use $left(v)$ ($right(v)$) to refer to the left (right) child of v. We use $right^i$ and $left^i$ to refer to iterated application of these functions. We use the notation $Ch_i + Bt_h$ to refer to a tree that is a chain of i nodes where the source node is the root of a Bt_h.

Definition 26. *We define the language* TREE-PEBBLE *(*TREE-VISITING-PEBBLE*) as the set of all tuples (T, k), where T is a rooted directed tree and k is a integer satisfying $1 \leq k \leq n$, such that $R^\bullet(T) \leq k$ ($R^\phi(T) \leq k$).*

In the rest of this section, we use the term pebbling to refer to *persistent reversible pebbling* unless explicitly stated otherwise.

3.6.2 Pebbling Meets Coloring

In this subsection, we describe the proof of Theorem 8. It helps to think about the complexity of the language TREE-PEBBLE. How can we prove that TREE-PEBBLE is in NP? The first attempt would be to use a pebbling sequence as certificate to show that the input tree T can be pebbled using at most k pebbles. Given a pebbling sequence, it can be verified in polynomial time that the sequence is valid and uses at most k pebbles. But, this does not work as it is not guaranteed that trees will have polynomial time optimal pebbling sequences. So, we need a succinct encoding of optimal pebbling sequences. The following definition provides this.

Definition 27. *(Strategy Tree) Let T be a rooted directed tree. If T only has a single node v, then any strategy tree for T only has a single node labelled v. Otherwise, we define a strategy tree for T as any tree satisfying*

1. *The root node is labelled with some edge $e = (u, v)$ in T.*
2. *The left subtree of root is a strategy tree for T_u and the right subtree is a strategy tree for $T \setminus T_u$.*

The following properties are satisfied by any strategy tree S of $T = (V, E)$.

1. Each node has 0 or 2 children.
2. There are bijections from E to internal nodes of S and from V to leaves of S.

3. Let v be any node in S. Then the subtree S_v corresponds to the subtree of T spanned by the nodes labelling the leaves of S_v. If u and v are two nodes in S such that one is not an ancestor of the other, then the subtrees in T corresponding to u and v are vertex-disjoint.

From the definition of strategy tree, it is clear that strategy trees are linear in the size of the input tree T. In other words, they are succinct enough. The following lemma proves that strategy trees can encode optimal pebbling sequences.

Lemma 8. *Let T be a rooted directed tree. Then $R^{\bullet}(T) \leq k$ if and only if there exists a strategy tree for T of depth at most k.*

Proof. We prove both directions by induction on $|T|$. If T is a single node tree, then the statement is trivial.

(if) Assume that the root of a strategy tree for T of depth k is labelled by an edge (u, v) in T. The pebbler then pebbles the node u. If the challenger challenges u, the pebbler follows the strategy for T_u given by the left subtree of root. If the challenger rechallenges, the pebbler follows the strategy for $T \setminus T_u$ given by the right subtree of the root. The remaining game takes at most $k - 1$ pebbles by the inductive hypothesis. Therefore, the total number of pebbles used is at most k.

(only if) Consider an upstream pebbler that uses at most k pebbles. We are going to construct a strategy tree of depth at most k. Assume that the pebbler pebbles u in the first move where $e = (u, v)$ is an edge in T. Then the root node of S is labelled e. Now we have $R^{\bullet}(T_u), R^{\bullet}(T \setminus T_u) \leq k - 1$. Let the left (right) subtree be the strategy tree obtained inductively for T_u ($T \setminus T_u$). Since the pebbler is upstream, the pebbler never places a pebble outside T_u ($T \setminus T_u$) once the challenger has challenged u (the root). □

We now define a combinatorial game called the matching game which is played on undirected trees. This game acts as a link between the reversible pebbling game and edge rank colouring.

Definition 28. *(Matching Game) Let U be an undirected tree. Let $T_1 = U$. At each step of the matching game, we pick a matching M_i from T_i and contract all the edges in M_i to obtain the tree T_{i+1}. The game ends when T_i is a single node tree. We define the* contraction number *of U, denoted $c(U)$, as the minimum number of matchings in the matching sequence required to contract U to the single node tree.*

Lemma 9. *Let T be a rooted directed tree and let U be the underlying undirected tree for T. Then $R^{\bullet}(T) = k + 1$ if and only if $c(U) = k$.*

Proof. First, we describe how to construct a matching sequence of length k from a strategy tree S of depth $k + 1$. Let the leaves of S be the level 0 nodes. For $i \geq 1$, we define the level i nodes to be the set of all nodes v in S such that one child of v has level $i - 1$ and the other child of v has level at most $i - 1$. Define M_i to be the set of all edges in U corresponding to level i nodes in S. We claim that $M_1, \dots, M_k$ is a matching sequence for U. Define S_i as the set of all nodes v in S such that the parent of v has level at least $i + 1$. Let $Q(i)$ be the statement "T_{i+1} is obtained from T_1 by contracting all subtrees corresponding to nodes (See Property 3) in S_i". Let $P(i)$ be the statement "M_{i+1} is a matching in T_{i+1}". We will prove $Q(0)$ and $Q(i) \implies P(i)$ and $(Q(i) \wedge P(i)) \implies Q(i+1)$. Indeed for $i = 0$, we have $Q(0)$ because $T_1 = U$ and S_0 is the set of all leaves in S or nodes in T (Property 2). To prove $Q(i) \implies P(i)$, observe that the edges of M_{i+1} correspond to nodes in S where both children are in S_i. So these edges correspond to edges in T_{i+1} (by $Q(i)$) and the fact that these edges are pairwise disjoint since no two nodes in S have a common child).

To prove that $(Q(i) \wedge P(i)) \implies Q(i+1)$, consider the tree T_{i+2} obtained by contracting M_{i+1} from T_{i+1}. Since $Q(i)$ is true, this is equivalent to contracting all subtrees corresponding to S_i and then contracting the edges in M_{i+1} from T_1. The set S_{i+1} can be obtained from S_i by adding all nodes in S corresponding to edges in M_{i+1} and then removing both children (of these newly added nodes) from S_i. This is equivalent to combining the subtrees removed from S_i using the edge joining them. This is because M_{i+1} is a matching by $P(i)$ and hence one subtree in S_i will never be combined with two other subtrees in S_i. But then contracting subtrees in S_{i+1} from T_1 is equivalent to contracting S_i followed by contracting M_{i+1}.

We now show that a matching sequence of length at most k can be converted to a strategy tree of depth at most $k + 1$. We use proof by induction. If the tree T is a single node tree, then the statement is trivial. Otherwise, let e be the edge in the last matching M_k in the sequence and let (u, v) be the corresponding edge in T. Label the root of S by e and let the left (right) subtree of root of S be obtained from the matching sequence $M_1, \dots, M_{k-1}$ restricted to T_u ($T \setminus T_u$). By the inductive hypothesis, these subtrees have height at most $k - 1$. $\square$

Lemma 10. *For any undirected tree U, we have $c(U) = \chi'_e(U)$.*

Proof. Consider an optimal matching sequence for U. If the edge e is contracted in M_i, then label e with the color i. This is an edge rank coloring. Suppose for contradiction that there exists two edges e_1 and e_2 with label i such that there is no edge labelled some $j \geq i$ between them. We can assume without loss of generality that there is no edge labelled i between e_1 and e_2 since if there is one such edge, we can let e_2 to be that edge. Then e_1 and e_2 are adjacent in T_i and hence cannot belong to the same matching.

Consider an optimal edge rank coloring for U. Then in the i^{th} step all edges labelled i are contracted. This forms a matching since in between any two edges labelled i, there is an edge labelled $j > i$ and hence they are not adjacent in T_i. □

The proof of Theorem 8 is summarized in Fig. 3.3

We now state the main theorem of this section.

Theorem 8. *Let T be a rooted directed tree and let U be the underlying undirected tree for T. Then we have $R^\bullet(T) = \chi'_e(U) + 1$.*

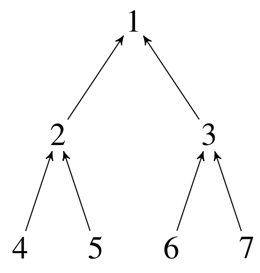
Corollary 1. *$R^\phi(T)$ and $R^\bullet(T)$ along with strategy trees achieving the optimal pebbling value can be computed in polynomial time for trees.*

Proof. We show that TREE-PEBBLE and TREE-VISITING-PEBBLE are polynomial time equivalent. Let T be an instance of TREE-PEBBLE. Pick an arbitrary leaf v of T and root the tree at v. By Theorem 8, the reversible pebbling number of this tree is the same as that of T. Let T' be the subtree rooted at the child of v. Then we have $R^\bullet(T) \leq k \iff R^\phi(T') \leq k - 1$.

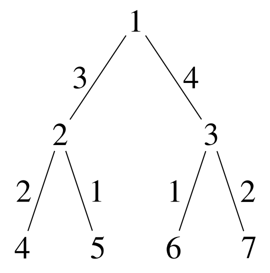
Let T be an instance of TREE-VISITING-PEBBLE. Let T' be the tree obtained by adding the edge (r, r') to T where r is the root of T. Then we have $R^\phi(T) \leq k \iff R^\bullet(T') \leq k + 1$.

The statement of the theorem follows from Theorem 8 and the linear-time algorithm for finding an optimal edge rank coloring of trees (Lam and Yue (2001)). □

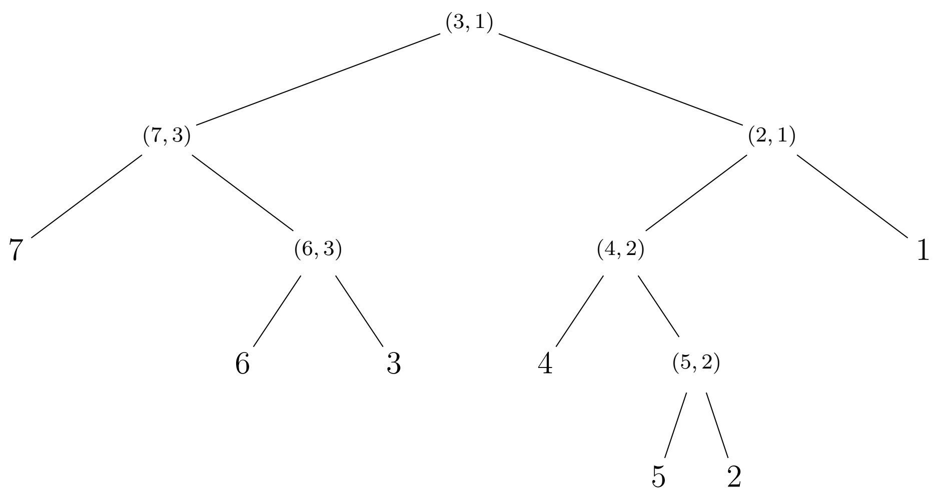
The following corollary is immediate from the equivalence of pebble games (Theorem 7).



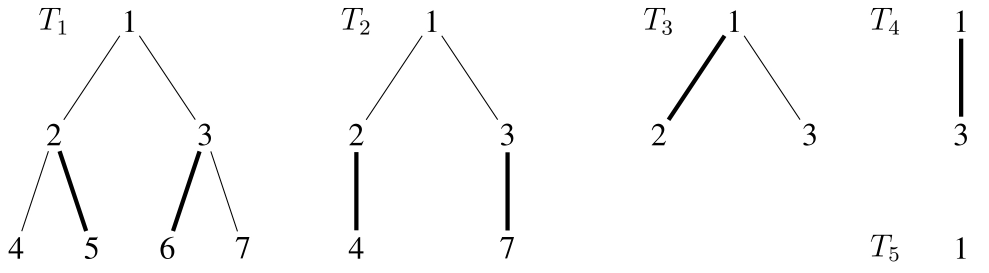
(a) The complete binary tree of height 3



(b) Optimal edge rank colouring



(c) Optimal strategy tree



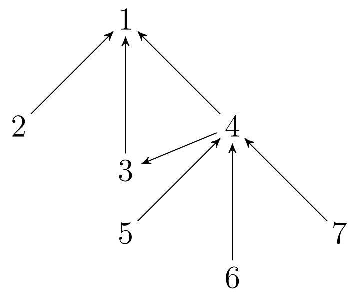
(d) Optimal matching sequence

Figure 3.3: Equivalence of persistent reversible pebbling, matching game and edge rank coloring on trees: An optimal strategy tree and the corresponding matching sequence and edge rank colouring for height 3 complete binary tree.

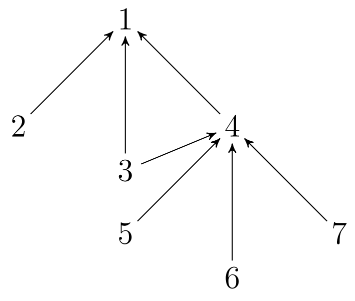
Corollary 2. *For any rooted directed tree T, we can compute $DT(T)$ and $RM(T)$ in polynomial time.*

An interesting consequence of Theorem 8 is that the persistent reversible pebbling number of a tree depends only on its underlying undirected graph. A natural question would be to ask whether this fact generalizes to DAGs. The following proposition shows that this is not the case.

Proposition 3. *There exists two DAGs with the same underlying undirected graph and different pebbling numbers.*



(a) $R^{\bullet}(G_1) = 5$



(b) $R^{\bullet}(G_2) = 6$

Proof. DAGs G_1 and G_2 have the same underlying undirected graph and different persistent pebbling numbers. □

CHAPTER 4

Testing Properties of Constant Width Grid Graphs

In this chapter, we illustrate a technique that uses algebraic tools to prove constant depth circuit upper bounds. We define the problems that we consider in Sections 4.1 and 4.2. We describe the monoid program framework introduced by Barrington and Thérien (1988) that enables us to prove these upper bounds in Section 4.3. We then describe our method, called the primary cycle method, in Section 4.4 with a simple application. We then use the primary cycle method to prove improved upper bounds for problems in Sections 4.5 and 4.6. An extended version of the results presented in this chapter can be found in Hansen *et al.* (2014).

4.1 Constant Width Grid Graphs

Testing for graph properties yield natural complete problems for many complexity classes. For ex., it is known that testing whether a graph is 3-colourable is NP-complete and testing whether there is an s-t path in a directed (undirected) graph is NL-complete (L-complete resp.). Barrington *et al.* (1998) proved that testing the existence of s-t path in upward-planar constant width grid graphs (See Definition 31) is complete for levels of the AC^0 hierarchy thereby yielding natural complete problems for all levels of the hierarchy. On the other hand, it is also known that testing existence of perfect matching, 2 and 3 colourability, and disjoint paths for unrestricted constant width grid graphs is in NC^1. In this chapter, we look at the complexity of testing properties of constant width grid graphs where the graph is given along with a planar embedding of the graph in the grid. We show that in this setting testing the existence of perfect matching, and vertex/edge disjoint paths can be done in ACC^0, and AC^0 respectively, improving the NC^1 upper bounds and simultaneously showing that many problems harder than reachability can be solved in constant depth and providing natural complete problems for various constant depth classes.

In this chapter we will prove improved upper bounds for testing the existence of perfect matching, 2-colouring, and vertex/edge disjoint paths on various classes of constant width grid graphs. This is done by a technique that we call the primary cycle method.

4.2 Preliminaries

First, we define the necessary graph classes.

Definition 29. *A graph is called a grid graph if its vertices form a subset of* $\{1,\ldots,\ell\} \times \{1,\ldots w\}$ *and all edges in the graph are of the form* $\{(u,v),(u',v')\}$ *where* $|u-u'| \leq 1$*. The number* ℓ *is called the length of the grid graph and the number* w *is called the width of the grid graph. If the edges of a grid graph are directed, then it is called a directed grid graph. The assignment of vertices to pairs* (i,j) *is called the embedding of the grid graph on the plane.*

Definition 30. *A grid graph along with an embedding is called grid-planar if no two edges of the graph cross each other. That is, the graph does not contain two edges* $\{(u,v),(u+1,v')\}$ *and* $\{(u,v''),(u+1,v''')\}$ *such that* $v < v''$ *and* $v' < v'''$ *or* $v > v''$ *and* $v' > v'''$*. A similar definition is used for directed graphs.*

Definition 31. *A directed grid-planar graph is called upward-planar if all the edges in the graph are directed such that* $e = ((u,.),(u+1,.))$ *(from left to right in the plane).*

In all problems considered in this chapter, we assume that the graph is given as input by specifying its embedding on the plane. The width of the input graph is assumed to be some fixed value. We also assume that the graph is grid-planar. Henceforth, we simply refer to them as constant width grid-planar graphs.

4.3 Monoid Word Problems

All our algorithms in this chapter work by reducing testing properties of constant width grid-planar graphs to various monoid word problems. In this section, we define various monoid word problems and state their complexity.

Definition 32. *A monoid is a set with an associative binary operation, say $\circ$, and an identity element. Fix a finite monoid M. The monoid word problem corresponding to M is to compute the product $w_1 \circ \cdots \circ w_n$ where $w_i \in M$ are given as input.*

It is easy to see that for any finite monoid, its word problem is in NC^1.

The order of a group is defined as the number of elements in the group. The order of an element R of group G, denoted $o_G(R)$, is the smallest n such that $R^n = E$, where E is the identity in G. It is known that the monoid word problem for any solvable monoid is in ACC^0 (Barrington and Thérien (1988)). Instead of defining solvable monoids, we state an implication that will be sufficient for our purposes.

Theorem 9. *A monoid is solvable if every group contained in the monoid is of odd order.*

Definition 33. *A monoid is aperiodic if and only if no non-trivial group is contained in the monoid.*

Barrington and Thérien (1988) proved that the word problem over solvable monoids is in ACC^0 and the word problem over aperiodic monoids is in AC^0. Therefore, if we manage to show that a language L reduces, in constant depth, to the monoid word problem of some solvable (aperiodic) monoid, then we can conclude that L is in ACC^0 (AC^0 resp.).

4.4 The Primary Cycle Method

To illustrate this technique, we will prove that s-t reachability on upward planar constant width grid graphs is in AC^0. In this problem, we are given as input an upward-planar constant width grid graph G , a vertex s on the leftmost layer, and a vertex t on the rightmost layer. We will design a constant depth circuit that outputs 1 if and only if there is a directed path from s to t in G.

The first step is to reduce this reachability problem to a monoid word problem. Given any graph G of odd length n, we can store all the necessary information in the relation $R(G) \subseteq [k] \times [k]$ as follows: For $1 \le i, j \le k$, the tuple $(i, j) \in R(G)$ if and only if there is a directed path from the node i in the leftmost layer in G to the node j

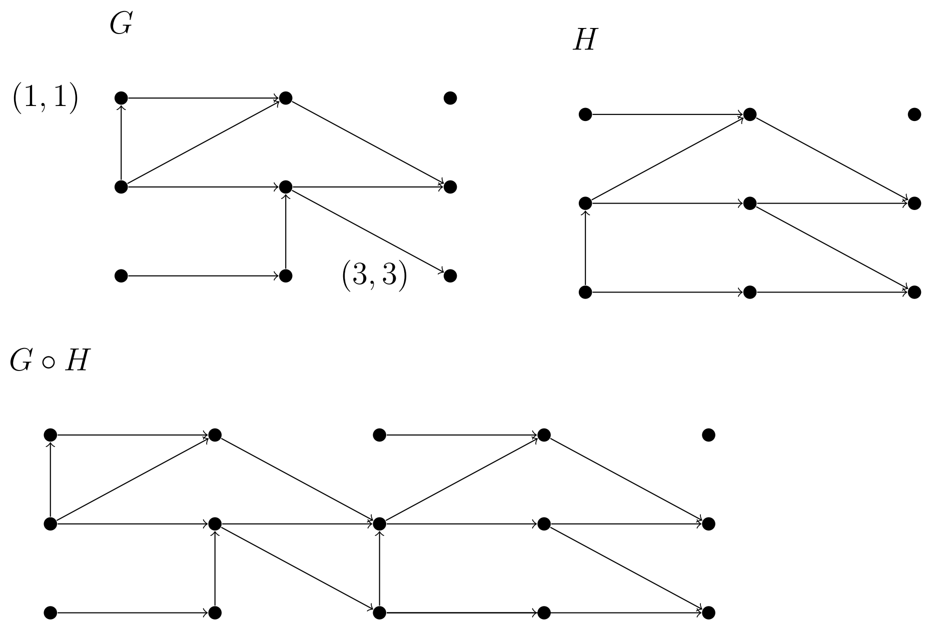


Figure 4.1: The Graph Concatenation Operation

in the rightmost layer in G. For any two grid graphs G and H where the vertices in the rightmost layer of H are the same as the vertices in the leftmost layer of H, we define the graph $G \circ H$ obtained by concatenating H to G by merging those two layers (See Figure 4.1). By convention, we assume that there are no vertical edges in the rightmost boundary of G. If G and H are grid graphs with corresponding relations $R(G)$ and $R(H)$, we observe that $R(G \circ H) = R(G)R(H)$ as there is a path from i to j in $G \circ H$ iff there exists some k such that there is a path from i to k in G and there is some path from k to j in H. Observe that the operation in $R(G)R(H)$ is just relation composition, which is associative. Now we can reduce our reachability problem to the monoid word problem over the monoid of relations on $[k]$.

Given G, split the graph G into graphs G_1, G_2, ..., G_k such that G_1 is the graph induced by the first 3 layers of G, G_2 by the third, fourth and fifth layers of G and so on. Then output the instance of the monoid word problem $R(G_1)R(G_2)\dots R(G_k)$. This reduction can be implemented in AC^0. It is easy to see that there is an s-t path in G iff the monoid product R contains the element (s,t).

If we can prove that the monoid of relations generated by the reachability problem is aperiodic, we can conclude using the Barrington-Therien framework that the reachability problem is in AC^0.

We now introduce a new method that we call the *primary cycle* method that can be used to prove aperiodicity/solvability of monoids.

Definition 34. *A cycle $C = x_1 \to \dots x_n \to x_1$ in the relation digraph of R called a* primary cycle *if and only if*

1. *The cycle C is the smallest cycle in $R[C]$ (The subgraph induced by C).*
2. *The relation E does not contain (x_i, x_j) where $i \neq j$.*

The importance of the primary cycle is that if we take any non-trivial subgroup of a monoid, then any non-trivial element in that group must have a primary cycle.

First, we prove that any non-trivial element in any group must contain a primary cycle of length at least 2.

Proposition 4. *Every $R \in G$ where $R \neq E$ must contain a primary cycle of length at least 2.*

Proof. Let us observe the structure of relation digraph representing R. It is clear that R must contain some cycle (not just self-loops). Suppose it does not, then let ℓ be the length of the smallest simple path (in edges) in R (not taking self-loops). Then any edge in $R^{\ell+1}$ must be obtained by taking a self-loop at least once. So if we take this self-loop one more time, we can obtain this edge in $R^{\ell+2}$. Similarly, for any edge in $R^{\ell+2}$, we must take some self-loop at least twice or it must take at least two self-loops. Therefore, taking one of these self-loops one less time gives this edge in $R^{\ell+1}$. So $R^{\ell+1} = R^{\ell+2}$, which implies $R \notin G$.

Now we will argue that R must have at least one primary cycle of length at least 2. Notice that if R has some cycle then it must have at least one induced cycle. If all such induced cycles have self-loops (i.e., for any cycle in the graph there is a chord or a self-loop on one of its vertices), then by an argument similar to the one in the previous paragraph $R^k = R^{k+1}$ for some k which is a contradiction. So R must contain some induced cycle of length at least 2. Now if for this cycle C, E contains (x_i, x_j) for $i \neq j$ and $x_i, x_j \in C$, then using $RE = ER = R$ we conclude that C has a chord or a self-loop at one of the vertices and hence it is not an induced cycle. Now since $R \in G$, we have for some k that $R^k = E$. Then the relation E must have self-loops at all vertices in C. Therefore C is a primary cycle. $\square$

Proposition 5. *$o_G(R)$ is a common multiple of the lengths of primary cycles in R.*

Proof. Suppose there exists a primary cycle C of length n in R such that n does not divide $m = o_G(R)$. Then $R^m = E$ contains an edge (x_i, x_j) for some $x_i, x_j \in C$, $i \neq j$ which contradicts the assumption that C is a primary cycle. □

To prove that this monoid does not contain a non-trivial group, we will use proof by contradiction. Assume that the monoid contains a cyclic group of order $m \geq 2$. Let H be a graph that corresponds to some generator a of the group. Then H must contain a primary cycle on the elements $a_1, a_2, \ldots, a_m$. Then there exists an i such that the a_i-a_{i+1} path intersects with a_{i-1}-a_i path. Since the graph H is upward planar, they must intersect at a vertex. Therefore, we can conclude that there is an a_{i-1}-a_{i+1} path in H contradicting the fact that $a_1, \ldots, a_k$ forms a primary cycle.

4.5 Perfect Matching

We now consider the complexity of detecting the existence of a perfect matching in a *manhattan* graph.

Definition 35. *A constant width grid-planar graph is called* manhattan *graph if all edges $\{(u, v), (u', v')\}$ satisfy $|u - u'| + |v - v'| = 1$.*

We will prove Theorem 10 in this section.

Theorem 10. *Fix an integer $k \geq 1$. Testing whether a manhattan graph of width-k given as input has a perfect matching can be done in* ACC^0.

For the rest of this section, we assume that the width is a fixed integer k.

Now consider a manhattan graph G and consider a subgraph of G, G', induced by any 3 layers of G. Let the vertices on the left boundary of G' be X and let the vertices on the right boundary be Y. Then in any perfect matching of G, all vertices in G' except those in X and Y has to be matched using the edges of G'. Vertices in X and Y could be matched by the subgraph of G on the left or right of G' respectively. This observation leads to the definition of the following monoid.

Definition 36. *For each grid-planar graph G of odd length ℓ that has no vertical edges in the rightmost layer, we define the corresponding monoid element G^M as the triple (X_1, X_2, R) where $X_1 \subseteq [k]$ is the set of vertices in the leftmost layer of G, $X_2 \subseteq [k]$ is the set of vertices in the rightmost layer of G and $R \subseteq 2^{X_1} \times 2^{X_2}$ is a binary relation such that for any $x_1 \subseteq X_1$, $x_2 \subseteq X_2$ we have $(x_1, x_2) \in R$ if and only if G has a matching that matches all vertices in G except $\overline{x_1}$ in the leftmost layer and x_2 in the righmost layer.*

The monoid product is defined as $(x_1, x_2, R)(x_3, x_4, S)$ = $(x_1, x_4, R \circ S)$ when $x_2 = x_3$ and $\circ$ is the usual composition of binary relations. When $x_2 \neq x_3$, we define the product to be an element 0 for which $0x = x0 = 0$ for any x in the monoid. Now define the monoid $M = \{G^M$: G is an odd length bipartite grid-planar graph$\} \cup \{0\} \cup \{1\}$, where 1 is an the identity element.

The reduction to the monoid word problem is same as that in the s-t reachability problem.

Observe that the monoid operation is just relation composition. We will use the primary cycle method to show that this monoid cannot contain even order subgroups. This combined with Theorem 9 shows that the monoid is solvable.

In order to rule out even order groups, as the following proposition shows, we just have to rule out the two element cyclic subgroup $\{e, a\}$.

Proposition 6. *Every even order group G contains the two element group as a subgroup.*

Proof. If G is a cyclic group of order n and a is a generator of G, then the subgroup generated by $a^{n/2}$ is a two element subgroup. If G is not cyclic, then consider the cyclic subgroup generated by any non-identity element a. By Lagrange's theorem, this subgroup must be an even order cyclic group which must contain an order two subgroup as claimed before. $\square$

Proof of Theorem 10. We will prove that for any R, $o_G(R) \neq 2$. Suppose for contradiction that $o_G(R) = 2$. Then using Propositions 4 and 5 we can conclude that R has a primary cycle $C = x_1 \to x_2 \to x_1$ of length 2. Consider a graph G that corresponds to the element R. Let M_1 be a matching in G that corresponds to $(x_1, x_2) \in R$ and let M_2 be a matching in G that corresponds to the element $(x_2, x_1) \in R$. Consider the graph $S = M_1 \cup M_2$. The graph S^n is defined as the graph obtained by concatenating n copies of S. Note that for any odd n, the graph S^n is a union of two matchings, the matching $M = (x_1, x_2)$ obtained by the concatenation of matchings $M_1 M_2 \dots M_1$ and the matching $N = (x_2, x_1)$ obtained by the concatenation of matchings $M_2 M_1 \dots M_2$. Since $R^n = R$ for any odd n, it must hold true that the graph S^n cannot contain a matching (x_1, x_1). We will prove that for sufficiently large odd n, we can construct an (x_1, x_1) matching thereby contradicting the fact that C is a primary cycle.

Now we introduce some notations and definitions needed to present the proof. We label the vertices on the left side on the i^{th} copy of S in S^n as $1_{(i)}, \dots, k_{(i)}$. The rightmost vertices in S^n are labelled $1_{(n+1)}, \dots, k_{(n+1)}$. A path in S^n is called a *blocking path* if it connects some vertex in the leftmost layer to some vertex in the rightmost layer. If the graph S^n does not have a blocking path, then we can construct an (x_1, x_1) matching in S^n as follows: Consider the set V_L of all vertices in S^n that are reachable from some vertex in the leftmost layer and the set V_R of all vertices in S^n that are not in V_L. Since there is no blocking path, the sets V_L and V_R are disjoint. Now we can obtain a matching (x_1, x_1) in S^n by using the matching the vertices in V_L using edges of M and matching the vertices in V_R using edges of N.

It remains to prove that S^n cannot have a blocking path for large enough n. Suppose a blocking path exists for all n. Since the graph only has constant width, this path must have a *period*. i.e., there must exist a v distinct integers i and j such that there is a segment of this path that connects $v_{(i)}$ and $v_{(j)}$. Our proof will rule out the existence of such segments thus ruling out the existence of blocking paths for large enough n.

We say that a path P crosses a boundary in S^n if it has two consecutive edges e_1 and e_2 such that they belong to different copies of S in S^n. Note that e_1 and e_2 must belong to the same matching M_1 or M_2. If they do not, the vertex common to those edges must be in $x_1 \cap \overline{x_1}$ or $x_2 \cap \overline{x_2}$.

Firse, we rule out paths of period 1. i.e., we claim that, for any n, the graph S^n cannot have a path from $v_{(i)}$ to $v_{(i+1)}$ for any i and v. To simplify the proof, for a graph corresponding to a given monoid element we attach length 2 horizontal paths to the vertices in the left and right side through two new layers. Notice that this does not change the monoid element since the vertices in the graph corresponding to the monoid element which were originally matched inside the graph remains matched inside the graph itself and vice versa.

Suppose for contradiction that such a path P from $v_{(i)}$ to $v_{(i+1)}$ exists. Suppose that P connects to both these vertices from the same side, left or right. Consider the shifted version P' of P in S^{n+1} from $v_{(i+1)}$ to $v_{(i+2)}$. Thus P and P' are two distinct paths that share an edge. Therefore, there must be some vertex, common to P and P'. of degree at least 3. This is impossible since S^{n+1} is a union of two matchings. Thus P must connect to the two vertices $v_{(i)}$ to $v_{(i+1)}$ from opposite sides. This means that it crosses boundaries an even number of times. Also, this is an alternating path (of edges from M_1 and M_2) with an even number of edges. Therefore, we have $v \in x_1 \cap \overline{x_1}$ or $v \in x_2 \cap \overline{x_2}$.

We will now prove that there cannot exist blocking paths of period greater than 1. Assume $n \geq k$. Suppose for contradiction that S^n has a blocking path where there are integers i and j, $j > i + 1$, such that this blocking path has a segment P connecting $v_{(i)}$ to $v_{(j)}$ for some $1 \leq v \leq k$. Now consider the graph S^{n+1}. This graph also has this path P from $v_{(i)}$ to $v_{(j)}$ and also a path P' from $v_{(i+1)}$ to $v_{(j+1)}$ that is simply a "shifted" version of P. These paths are vertex disjoint. Because if P and P' intersect, then we can construct a path from $v_{(i)}$ to $v_{(i+1)}$ in S^{n+1}. But Lemma 11 implies that paths P and P' must interesect. This contradiction concludes the proof. $\square$

We will now state and prove Lemma 11.

Lemma 11. *Let $C' = C + (q, 0)$ be a horizontal shift of the curve C. Then C and C' intersect.*

Proof. Map the region $R = \{(x, y) \mid 0 \leq x \leq p, 1 \leq y \leq w\}$ into the plane by the map $\phi(x, y) = (y \cos(2\pi x/p), y \sin(2\pi/p))$. Then $\phi(C)$ and $\phi(C')$ are closed simple curves in the plane both containing the origin. By the Jordan curve theorem, each of these curves divide the plane into an inside set and an outside set. If they do not intersect then either $\phi(C)$ encloses $\phi(C')$ or $\phi(C')$ encloses all of $\phi(C)$. In particular, this means that the two curves enclose sets of different areas. However $\phi(C')$ is just a rotation of $\phi(C)$ around the origin, and must in particular enclose the exact same area as $\phi(C)$. We conclude that the curves intersect. $\square$

4.6 Edge Disjoint Paths

In this section, we consider the following problem: Given an upward planar constant width grid graph G and two vertices s_1, s_2 on the left boundary of G and two vertices t_1, t_2 on the right boundary of G. Check if there are edge disjoint paths from s_1 to t_1 and s_2 to t_2. This is harder than s-t reachability on such graphs. Given an instance of s-t reachability, we can reduce it to this problem using a projection reduction as follows; increasing the width by 1 and adding a path from the left boundary vertex s' to the right boundary vertex t'. The reduced instance is then (G, s, s', t, t').

Definition 37. *We have $(u, v)R(w, x)$ iff there are edge-disjoint paths between u and w, and v and x (denoted $u \overset{P_{uw}}{\rightsquigarrow} w$ and $v \overset{P_{vx}}{\rightsquigarrow} x$). Here u and v are vertices on the leftmost layer and w and x are vertices on the rightmost layer.*

Now we will apply the primary cycle method and rule out all primary cycles. This will prove that the monoid is aperiodic and therefore the problem is in AC^0.

Proof. Let us assume that an element contains a primary cycle $(u_1, v_1) \to \cdots \to (u_n, v_n) \to (u_1, v_1)$. Assume without loss of generality that v_1 is the bottommost among vertices $\{u_i\} \cup \{v_i\}$. First, we will prove that there is at least one path from u_1 to u_1 and from v_1 to v_1. Then we will prove that there are edge disjoint paths from u_1 to u_1 and v_1 to v_1. This implies the self-loop $(u_1, v_1)R(u_1, v_1)$, contradicting the assumption that it is part of a primary cycle.

We will prove that at least one path must exist from u_1 to u_1. The argument for the existence of at least one path from v_1 to v_1 is similar. If $n = 2$, then the paths $u_1 \rightsquigarrow u_2$ and $u_2 \rightsquigarrow u_1$ must intersect at a vertex and this gives us a path from u_1 to u_1. So assume $n > 2$. Assume without loss of generality that $u_n > u_1$. If $u_2 > u_1$, then all paths $u_1 \rightsquigarrow u_2$ and $u_n \rightsquigarrow u_1$ must intersect to give a path from u_1 to u_1. So assume $u_2 < u_1$. Choose the minimum $2 \le i \le n-1$ such that $u_i < u_1 \le u_{i+1}$. Now the paths $u_1 \rightsquigarrow u_2$ and $u_i \rightsquigarrow u_{i+1}$ intersect to give a path from u_1 to u_{i+1} and this path intersects with $u_n \rightsquigarrow u_1$ to give a path from u_1 to u_1.

Consider the topmost path from u_1 to u_1 (marked α in Figure 4.2) and the bottommost path from v_1 to v_1 (marked β). We name the region above the topmost u_1 to u_1 path, R_1. We claim that these paths are edge disjoint. Suppose for contradiction that they intersect at an edge e. Fix the intervals $I_1 = [u_1, v_1]$ and $I_2 = (1, u_1)$. Both u_2 and v_2 cannot belong to I_1 as then any paths $u_1 \rightsquigarrow u_2$ and $v_1 \rightsquigarrow v_2$ must edge intersect at e. Now we can assume without loss of generality that $u_2 \in I_2$ and the path $u_1 \rightsquigarrow u_2$ crosses e through R_1 or passes through e (otherwise we can construct a $v_1 \rightsquigarrow v_1$ that is below the bottommost such path). Assume all $u_1 \rightsquigarrow u_2$ paths pass through e. Then we claim that $v_2 \in I_1$. If not, all paths from v_1 to v_2 must pass through e contradicting the existence of edge disjoint paths from u_1 to u_2 and v_1 to v_2. Therefore, we can conclude that there is at least one $u_1 \rightsquigarrow u_2$ path crossing e through R_1 (as shown in Figure 4.2) or that $v_2 \in I_1$. If all paths $u_1 \rightsquigarrow u_2$ pass through e and $v_2 \in I_1$, then at least one path from v_1 to v_2 must cross e through R_1. In any case, we can conclude that there is a directed segment crossing e through R_1 and ending at some vertex $u < u_1$ (marked ϕ in Figure 4.2). Let i be the highest index such that $u_i \in I_2$. (The following addition on indices wrap around at n). We want to claim that $u_n \rightsquigarrow u_1$ passes through e. Note that if $i = n$, then any path from u_n to u_1 not passing through e would allow us to construct a u_1 to u_1 (or else a v_1 to v_1) path above the topmost (bottommost respectievely) such path. All the paths $u_i \rightsquigarrow u_{i+1}$ (marked δ) must pass through e as otherwise it would intersect with $u_1 \rightsquigarrow u_2$ to create a segment above e. By the same argument, all paths $u_{i+1} \rightsquigarrow u_{i+2}, \ldots, u_n \rightsquigarrow u_1$ (The last one marked γ) must pass through e. Now if $v_n \in I_1$, then all the paths $v_n \rightsquigarrow v_1$ must pass through e. If $v_n \in I_2$, then the path $v_n \rightsquigarrow v_1$ implied by the primary cycle is of one of the two forms as shown by dotted paths marked 1 and 2 in Figure 4.2). It follows that this path cannot cross e through R_1 as it would intersect with the segment ϕ to create a u_1 to u_1 path above the topmost such path. So the path $v_n \rightsquigarrow v_1$ must pass through e and it will intersect with $u_n \rightsquigarrow u_1$. A contradiction. □

4.7 Additional Results

These additional results were obtained as part of the collaborative work reported in Hansen *et al.* (2014). All upper bounds in this section were obtained by reducing to monoid word problems. These results illustrate the reach of this approach.

Theorem 11. *Testing whether a constant width grid-planar graph has a 2-coloring can be done in* $\mathsf{AC}^0[2]$.

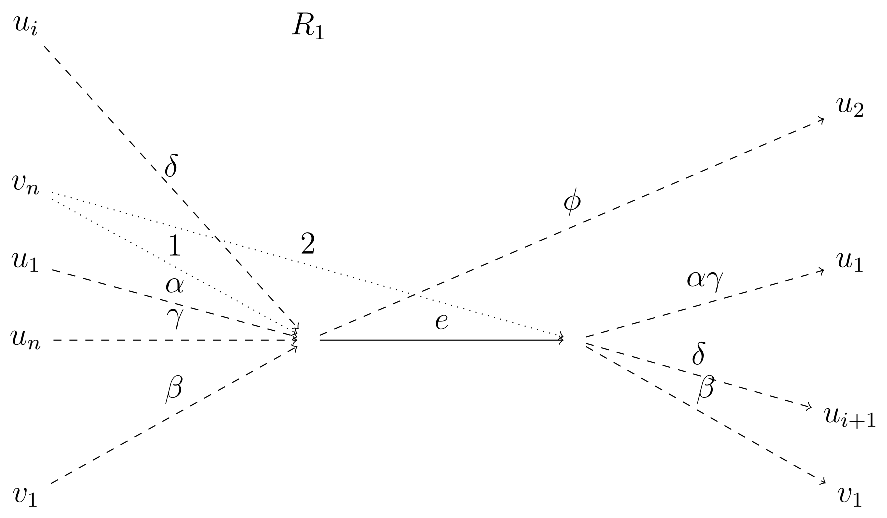


Figure 4.2: Edge Disjoint Paths on Upward-Planar grid graphs

The edge disjoint paths problem considered in Section 4.6 can be generalized to take multiple source-terminal pairs as input. i.e., for some fixed m, the input is $(G, s_1, t_1, \ldots, s_m, t_m)$ and the goal is to test whether there are pairwise edge disjoint paths from s_i to t_i for all i. We showed that this generalization is also in AC^0.

A close variant of the edge disjoint paths problem is the vertex disjoint paths problem. We showed that this problem can be solved in AC^0 for constant width grid-planar graphs.

We also showed lower bounds for many problems on constant width grid-planar graphs. For example, we showed that the 2-coloring problem has a lower bound of AND $\circ$ XOR $\circ$ AC^0. That is, the circuit value problem where the input circuit has an AND gate at the top-level, only XOR gates at the second level, and the inputs to all these XOR gates come from AC^0 circuits, can be reduced to the 2-coloring problem. We know that this circuit class is very close to $\mathsf{AC}^0[2]$. So this lower bound implies that this upper bound is very close to optimal. All circuit upper and lower bounds, obtained as part of the collaborative work reported in Hansen *et al.* (2014), are listed in Table 4.3.

Problem	Upper bound	Lower bound
2-coloring	$\mathsf{AC}^0[2]$	AND $\circ$ XOR $\circ$ AC^0
3-coloring	NC^1	NC^1
Bipartite perfect matching	ACC^0	AC^0
Perfect matching	NC^1	AND $\circ$ OR $\circ$ XOR $\circ$ AC^0
Hamiltonian cycle	NC^1	NC^1
Disjoint paths variants	AC^0	AC^0

Figure 4.3: Complexity of problems on constant width grid-planar graphs

CHAPTER 5

Comparator Circuits and Classical Complexity Classes

In this chapter, we define the comparator circuit model (Section 5.1). We define generalized comparator circuits working over finite bounded posets in Section 5.2. We show how to characterize classes P and NP in terms of these generalized comparator circuits in Sections 5.3 and 5.4 respectively. We define skew comparator circuits and show that they characterize the class L in Section 5.5. We consider generalizations of Boolean formulae over lattices in Section 5.6. Finally, we describe comparator circuits over growing lattices in Section 5.7 and show that they can also be used to characterize the class P in terms of comparator circuits. All results presented in this chapter can be found in Komarath *et al.* (2015*b*).

5.1 Preliminaries

We begin by defining comparator circuits and the class CC.

Definition 38. *A* comparator circuit *has a set of n* lines *$\{w_1, \ldots, w_n\}$ and an ordered list of gates (w_i, w_j). Each line can be fed as input a value that is either (Boolean) 0 or 1. We define $val(w_i)$ to be the value of the line w_i. Each gate (w_i, w_j) updates $val(w_i)$ to $val(w_i) \wedge val(w_j)$ and $val(w_j)$ to $val(w_i) \vee val(w_j)$ in order. After all gates have updated the values, the value of the line w_1 is the output of the circuit.*

In an annotated comparator circuit, lines are also allowed to have x_i or $\overline{x_i}$ as initial values. The output of such a circuit is a function of the values of the input variables $x_1, \ldots, x_n$. The output is obtained by substituting the value of the i^{th} input bit to lines labelled x_i and logical negation of the i^{th} input bit to lines labelled $\overline{x_i}$. The rest of the evaluation proceeds as mentioned for comparator circuits.

Figure 5.1 shows a schematic representation of a comparator circuit. Each gate (w_i, w_j) is represented by an arrow from the line w_i to the line w_j. It is quite clear from the definition that any comparator circuit can be implemented by a general Boolean

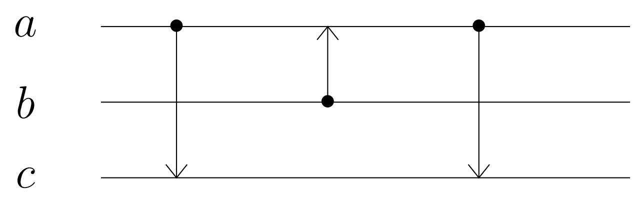


Figure 5.1: The schematic representation of a Comparator Circuit

circuit using AND, OR, and NOT gates that is at most the size of the original comparator circuit. Since polynomial size Boolean circuits capture the class P, it follows that CC $\subseteq$ P.

We now present an alternative definition of comparator circuits that is more similar to the definition of Boolean circuits. This definition is useful for proving some theorems related to comparator circuits. It is easy to see that this definition is equivalent to Definition 38.

Definition 39. *A comparator gate is a two input, two output gate where one of the outputs produce the AND of the two input values and the other output produces the OR of the two input values. A comparator circuit is a circuit that consists only of comparator gates. Each input gate of such a circuit is labelled 0, 1 or in the case of annotated comparator circuits also by x_i or $\overline{x_i}$ for some input bit i.*

We can extend Definitions 38 and 39 to define families of comparator circuits deciding languages as in Definition 3. We define the complexity class CC as the set of languages decided by logspace-uniform polynomial size comparator circuits.

Which problems are logspace complete for the class CC? The canonical hard problem for this class is the Comparator Circuit Value Problem.

Definition 40. *(Comparator Circuit Value Problem Given (C, x) as input, where C is a comparator circuit on n inputs and $x \in \{0,1\}^n$, find the output of the comparator circuit C when fed x as input. The circuit C is encoded as an ordered list mirroring Definition 38.*

This problem can be proved to be logspace hard for CC can be proved as follows. Fix any language L in CC. For input x with $|x| = n$, simply output the n^{th} comparator circuit, using the logspace uniformity algorithm, for L and the string x.

We still have not proved that the problem CCVP belongs to the class CC. In order to be able to do this, we must design a comparator circuit that can take other comparator circuits as input and simulate their computation efficiently. Such circuits are called *universal*. Cook et al proved Theorem 12 which shows that comparator circuits are universal. That is, there is a comparator circuit that when given another comparator circuit C on n inputs and x with $|x| = n$ as input, determine the output of C on x. Furthermore, the size of this universal circuit is polynomial in size of the input. With this theorem, we can conclude that CCVP is indeed complete for CC.

Theorem 12 (Cook *et al.* (2014))**.** $\mathsf{CCVP} \in \mathsf{CC}$

Theorem 12 also implies that the class CC can be defined equivalently as problem that are logspace reducible to CCVP or problems that have polynomial size comparator circuits. An open question proposed in Cook *et al.* (2014) is to give a TM characterization of the class CC. Such a characterization will give more insight into the class CC and its relationship between other classical complexity classes. Our main goal in this chapter is to define a generalization of comparator circuits. We then show that this generalization captures the complexity classes P and NP.

It is also known that $\mathsf{NL} \subseteq \mathsf{CC}$. In fact, the bounds $\mathsf{NL} \subseteq \mathsf{CC} \subseteq \mathsf{P}$ are the best bounds that we know about the class CC.

Recall that the class NL is also defined in terms of skew Boolean circuits. It is natural to ask what such a restriction to comparator circuits would give? We introduce skew comparator circuits and show that they capture the class L.

We say that an AND gate in a comparator gate is *used* if there is the AND output wire of that comparator gate is connected to some other gate in the circuit. An AND gate in the circuit is called *skew* if and only if at least one input to that gate is the constant 0 or the constant 1 or (in the case of annotated circuits) an input bit x_i or $\overline{x_i}$ for some i.

Definition 41 (Skew Comparator Ciruits)**.** *A comparator circuit is called a* skew comparator circuit *if and only if all used AND gates in the circuit are* skew.

Now, we can define the complexity class corresponding to skew comparator circuits.

Definition 42 (SkewCC)**.** *The complexity class* SkewCC *consists of languages that can be decided by polynomial size skew comparator circuits.*

We define SkewCCVP to be the circuit evaluation problem for skew comparator circuits. Note that given the ordered list representation of a comparator circuit, it is easy to check whether an AND gate is used or not. For ex., if the i^{th} gate is (w_1, w_2), then the AND output of this gate is unused iff there is no element in the list of gates with w_1 as a member at a position greater that i in the list.

We also want to use comparator circuits to define complexity classes like P and NP. Since these classes are larger than CC, we have to generalize the comparator circuit model. Before doing this, we state the motivation for considering this generalization.

Consider the problem of computing the PARITY of two input bits a and b. Boolean circuits require 3 gates to do this. However, if we allow the circuit to compute over arbitrary lattices where the AND gate computes GLB and OR gate computes the LUB over the lattice, we can compute the parity of two inputs with a single gate over the four element distributive lattice. A natural question to ask is whether allowing circuits to compute over arbitrary lattices will increase its computational power. i.e., change the complexity class characterized by the circuit class. It is easy to see that we can simulate general circuits working over any finite lattice by using a Boolean circuit only with a constant factor increase in size and depth. This is because the LUB and GLB of any finite lattice can be implemented by a constant depth Boolean circuit. What about comparator circuits? We show that this generalization allows comparator circuits to capture the complexity class P (which is believed to strictly contain the class CC and therefore cannot be simulated using polynomial factor blowup in size by comparator circuits).

5.1.1 Definitions from Order Theory

Before presenting the formal definition of generalization to lattices, we state some basic definitions from order theory. A more detailed treatment can be found in standard textbooks (See Davey and Priestley (1990)).

A set P along with a reflexive, anti-symmetric and transitive relation denoted by $\leq_P$ is called a *poset*. An element $m \in P$ is called the *greatest element* if $x \leq m$ for all x in P. An element $m \in P$ is called the *least element* if $m \leq x$ for all x in P. A poset is called *bounded* if it has a greatest and a least element. Note that any finite poset can be converted into a finite bounded poset by adding two new elements 0 and 1 and adding the relations $m \leq 1$ and $0 \leq m$ for every element m in the poset. *Minimal upper bounds* of two elements x, y in P, denoted by $x \vee y$, is the set of all $m \in P$ such that $x \leq m$, $y \leq m$ and there exists no m' distinct from m such that $x \leq m'$, $y \leq m'$ and $m' \leq m$. *Maximal lower bounds* of two elements x, y in P, denoted by $x \wedge y$, is the set of all $m \in P$ such that $m \leq x$, $m \leq y$ and there exists no m' distinct from m such that $m' \leq x$, $m' \leq y$ and $m \leq m'$. A poset P is called a *lattice* if every pair of elements x and y has a unique maximal lower bound and a unique minimal upper bound. In a lattice, the minimal upper bound (maximal lower bound) of two elements is also known as the *join* (*meet*). Since minimal upper bound and maximal lower bound are unique in a lattice, we drop the set notation when describing them. i.e., instead of writing $a \vee b = \{x\}$, we simply write $a \vee b = x$. A lattice L is called *distributive* if for every elements $a, b, c \in L$ we have $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$. An order embedding of a poset P into a poset P' is a function $f : P \mapsto P'$ such that $f(x) \leq_{P'} f(y) \iff x \leq_P y$. We say that the lattice L is a sub-lattice of L' if $L \subseteq L'$ and L is also a lattice under the meet and join operations inherited from L'. In this case, we say that L' *embeds* L.

We now state some technical theorems from the theory which we crucially use. The following theorem shows that given a poset one can find a lattice that contains the poset.

Theorem 13 (Dedekind-Macneille Completion Davey and Priestley (1990))**.** *For any poset P, there always exist a smallest lattice L that order embeds P. This lattice L is called the* Dedekind-MacNeille completion *of P.*

One crucial property of Dedekind-MacNeille completion is that it preserves all meets and joins that exist in the poset. i.e., if a and b are two elements in the poset and $a \vee b = x$ in the poset, then we have $f(a) \vee f(b) = f(x)$ in the Dedekind-MacNeille completion of the poset, where f is the embedding function that maps elements in P to elements in L.

We now state a very important theorem that concerns the structure of distributive lattices.

Theorem 14 (Birkhoff's Representation Theorem Davey and Priestley (1990))**.** *The elements of any finite distributive lattice can be represented as finite sets, in such a way that the join and meet operations over the finite distributive lattice correspond to unions and intersections of the finite sets used to represent those elements.*

The n^{th} *partition lattice* for $n \geq 2$, denoted Π_n, is the lattice where elements are partitions of the set $\{1, \ldots, n\}$ ordered by refinement. Equivalently, the elements are equivalence relations on the set $\{1, \ldots, n\}$ where the glb is the intersection and lub is the transitive closure of the union.

Theorem 15 (Pudlák and Tůma (1980))**.** *For any finite lattice L, there exists an i such that L can be embedded as a sublattice in Π_i.*

We can describe elements of the partition lattice Π_n by using undirected graphs on the vertex set $\{1, \ldots, n\}$. Given an undirected graph $G = (\{1, \ldots, n\}, E)$, the corresponding element $A_G \in \Pi_n$ is the equivalence relation $A_G = \{(i, j) : j \text{ is reachable from } i \text{ in } G\}$. We may choose transitively closed graphs (disjoint union of cliques) as the canonical representation for elements of partition lattices.

Some Relations in Partition Lattices: A formula over a lattice is defined analogously to a Boolean formula. The Boolean AND and OR operations are generalized to glb and lub operations of the lattice and the formula may contain elements of the lattice as constants (Similar to Boolean values 0 and 1 in a Boolean formula). In this section, we prove the existence of a certain formula over partition lattices.The following statements hold[1] for any partition lattice Π_i where $i \geq 2$. In the following propositions, the element 0 refers to the least element of the lattice and the element 1 refers to the greatest element of the lattice.

Proposition 7. *For any $A, B \in \Pi_i$ such that $A \not\leq B$, there exists a formula $\texttt{DIST}_{A,B}$ over Π_i such that $\texttt{DIST}_{A,B}(x) = 1$ if $x = A$ and strictly less than 1 if $x = B$.*

Proof. There are two cases to consider. Case when $[A > B]$ Let $P \in \Pi_i$ be the element corresponding to a path with exactly one vertex from each partition in A. We define $\texttt{DIST}_{A,B}(x) = x \vee P$. Case when $[A \not> B]$ Let $e_1, \ldots, e_m$ be the edges in $B \backslash A$ (using canonical representation) and let B_i denote the element in the partition lattice that correspond to the undirected graph having only the edge e_i. Let $g(x) = x \vee B_1 \vee \ldots \vee B_m$. We have $g(A) > g(B) = B$. Then define $\texttt{DIST}_{A,B}(x) = \texttt{DIST}_{g(A),B}(g(x))$. $\square$

Proposition 8. *For any $A \in \Pi_i$, there exists a formula $\texttt{GE}_A(x)$ that is 1 iff $x \geq A$. In addition, there exists a formula $\texttt{GE'}_A(x)$ that evaluates to 1 if $x \geq A$ and evaluates to 0*

[1]Since we have not seen them explicitly in the literature, we include the proofs in this thesis.

otherwise.

Proof. For the first part, simply define the formula $\mathtt{GE}_A(x) = \bigwedge_{B \not\geq A} \mathtt{DIST}_{A,B}(x)$ when $A \neq 0$. Define $\mathtt{GE}_0$ as identically 1.

For the second part, consider the formula f_Z that is defined if and only if $Z \neq 1$ and it maps 1 to 1 and Z to 0 (the images of the rest of the elements in the lattice can be arbitrary). Let Z have $k \geq 2$ partitions. Let $e_1, \ldots, e_m$ be the edges of the complete k-partite graph on these k partitions. Let $B_1, \ldots, B_m$ be lattice elements such that B_i corresponds to the undirected graph that contains only the edge e_i. $f_Z(x) = \bigvee_{i=1}^{m} (x \wedge B_i)$. Now to complete the second part, define the formula $\mathtt{GE'}_A(x) = \bigwedge_{B<1} f_B(\mathtt{GE}_A(x))$ ($\mathtt{GE'}_0$ is identically 1). $\square$

5.2 Generalization to Finite Bounded Posets

We now have enough background to consider circuits over posets. After defining this generalization, we prove the existence of universal circuits for these models (A genaralization of Theorem 12). The existence of these generalized universal comparator circuits imply that the classes characterized by comparator circuit families over fixed finite bounded posets also have canonical complete problems – The comparator circuit evaluation problem over the same fixed finite bounded poset.

Definition 43 (Comparator Circuits over Fixed Finite Bounded Posets)**.** *A comparator circuit family over a finite bounded poset P with an accepting element*[2] $a \in P$ *is a family of circuits* $C = \{C_n\}_{n \geq 0}$ *where* $C_n = (W, G, f)$ *where* $f : W \mapsto (P \cup \{(i, g) : 1 \leq i \leq n$ *and* $g : \Sigma \mapsto P\})$ *is a comparator circuit. Here* $W = \{w_1, \ldots, w_m\}$ *is a set of lines and* G *is an ordered list of gates* (w_i, w_j)*.*

On input $x \in \Sigma^n$*, we define the output of the comparator circuit* C_n *as follows. Each line is initially assigned a value according to* f *as follows. We denote the value of the line* w_i *by* $val(w_i)$*. If* $f(w) \in P$*, then the value is the element* $f(w)$*. Otherwise* $f(w) = (i, g)$ *and the initial value is given by* $g(x_i)$*. A gate* (w_i, w_j) *(non-deterministically)*

[2]In the definition of general Boolean circuits it is implicit that the element 1 is the accepting element. However, it does not make any difference even if we use 0 as the accepting element. This is because a Boolean circuit that accepts using 0 can be easily converted to one that accepts using 1 by complementing the output. This is not true for comparator circuits over bounded posets in general. Using different elements as accepting elements may change the power of the comparator circuit.

updates the value of the line w_i into $val(w_i) \wedge val(w_j)$ and the value of the line w_j into $val(w_i) \vee val(w_j)$. The values of lines are updated by each gate in G in order and the circuit accepts x if and only if $val(w_1) = a$ at the end of the computation for some sequence of non-deterministic choices.

Let Σ be any finite alphabet. A comparator circuit family C over a bounded poset P with an accepting element $a \in P$ decides $\mathsf{L} \subseteq \Sigma^*$ if and only if $C_{|x|}(x) = a$ if and only if $x \in \mathsf{L}$ $\forall x \in \Sigma^*$.

Remark 1. *Note that when the underlying poset is a lattice, the output of all gates in the comparator circuit is deterministic. In other words, the non-determinism in the circuit comes from the fact that two elements in a poset need not have unique lubs and glbs.*

We now define complexity classes captured by this generalization.

Definition 44. *We define the complexity class (P, a)–CC as the set of all languages accepted by poly-size comparator circuit families over the finite bounded poset P with accepting element $a \in P$.*

If the complexity class does not change with the accepting element, i.e., (P, a)–$\mathsf{CC} = (\mathsf{P}, \mathsf{b})$–$\mathsf{CC}$ for any $a, b \in P$, we simply write P–CC to refer to the complexity class (P, a)–CC.

We note that for any bounded poset P with at least 2 elements, we can simulate a Boolean lattice by using 0 (least element) and some $a > 0$ in P. Therefore, we have $\mathsf{CC} \subseteq (\mathsf{P}, \mathsf{a})$–$\mathsf{CC}$.

Definition 45. *For any finite bounded poset P and any $a \in P$, the comparator circuit evaluation problem (P, a)–CCVP is defined as the set of all tuples (C, x) such that C on input x has a sequence of non-deterministic choices where it outputs $a \in P$ where C is a comparator circuit over P.*

We now describe an encoding for the (P, a)–CCVP problem. The input is encoded by a binary string of the form $1^n 0 1^m 0 \{0, 1\}^{n(n-1)m+n}$. Here the last $n(n-1)m$ bits of the string can be viewed as m blocks of $n(n-1)$ bits where the i^{th} block has exactly one set bit, say (k, j) where $k \neq j$, and it encodes the fact that the i^{th} gate is from line k to line j. The n bits prior to these bits encode the initial values of n lines. This encoding is logspace-equivalent to the ordered list representation. We call strings of this form (n, m)-valid. A given N-bit string can be valid for at most one (n, m) pair. We first prove that a universal comparator circuit exists even for comparator circuit model working over arbitrary finite fixed posets.

Proposition 9. *For any bounded poset P, there exists a universal comparator circuit $U_{n,m}$ over P that when given (C, x) as input, where C is a comparator circuit over P with n lines and m gates, simulates the computation of C. That is, $U_{n,m}$ has a non-deterministic path that outputs $a \in P$ if and only if C has such a path, for any $a \in P$. Moreover, the size of $U_{n,m}$ is $\mathsf{poly}(n, m)$.*

Proof. We simply observe that the construction for a universal circuit for the class CC in Cook *et al.* (2014) generalizes to arbitrary bounded posets. The gadget shown in Figure 5.2 enables/disables the gate $g = (y, x)$ depending on the "enable" input e. Here the inputs e and $\overline{e}$ satisfy the following property. If $e = 0$ ($e = 1$), then $\overline{e} = 1$ ($\overline{e} = 0$ resp) where 0 and 1 are the least and greatest elements of the bounded poset P respectively. If the enable input is 1, then gate g is active. If the enable input is 0, then the gate g acts as a pass-through gate. i.e., the lines labelled x and y retain their original values.

Now to simulate a single gate in the circuit C, the universal circuit uses $n(n-1)$ such gadgets where n is the number of lines in C. The inputs e and $\overline{e}$ for each gadget is set according to C. The circuit C can be simulated using $n(n-1)m$ gates where m is the number of gates in C. $\square$

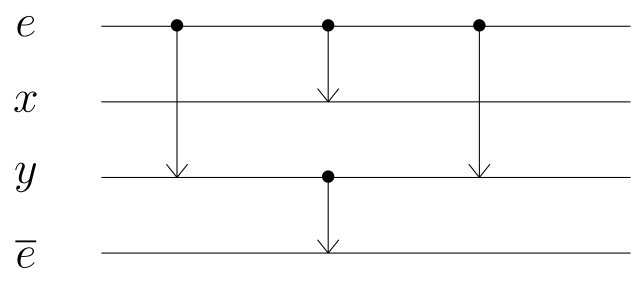


Figure 5.2: Conditional Comparator Gadget

The following proposition is a generalization of the corresponding theorem for Boolean comparator circuits in Cook *et al.* (2014).

Proposition 10. *The language* $(\mathsf{P}, \mathsf{a})\text{−}\mathsf{CCVP}$ *is complete under logspace reductions for the class* $(\mathsf{P}, \mathsf{a})\text{−}\mathsf{CC}$ *for all finite bounded posets P and any $a \in P$.*

Proof. The problem (P,a)–CCVP is trivially hard for the class (P,a)–CC. Let $\mathsf{L} \in (\mathsf{P},\mathsf{a})$–$\mathsf{CC}$ via a logspace-uniform circuit family $\{C_n\}$. Now given x as input, we output the tuple (C_n, x) by running the uniformity algorithm.

The fact that (P,a)–$\mathsf{CCVP} \in (\mathsf{P},\mathsf{a})$–$\mathsf{CC}$ follows from Proposition 9.q Given a string, it can be checked in logspace whether it is (n, m)-valid once n and m are fixed. Let $V_{n,m}$ be a logspace uniform comparator circuit over the 0–1 lattice that takes an N bit string as input and outputs 1 iff the input is an (n, m)-valid string. Let $N = 2n + m + 2 + n(n - 1)m$ be the total length of the input. The uniformity machine on input N writes out the description of $\bigvee_{(n,m)} U_{n,m} \wedge V_{n,m}$ over all (n, m) pairs satisfying $N = 2n + m + 2 + n(n - 1)m$. □

5.3 Comparator Circuits over Lattices

First, we show that comparator circuits over distributive lattices is also the class CC.

Theorem 16. *Let L be any finite distributive lattice and $a \in L$ be an arbitrary element. Then* $\mathsf{CC} = (\mathsf{L},\mathsf{a})$–$\mathsf{CC}$.

Proof. By Birkhoff's representation theorem, every finite distributive lattice of k elements is isomorphic to a lattice where each element is some subset of $[k]$ (ordered by inclusion) and the join and meet operations in the original finite distributive lattice correspond to set union and set intersection operations in the new lattice. We will use this to simulate a circuit over an arbitrary finite distributive lattice L of size k using a circuit over the 0–1 lattice. Each line w in the original circuit is replaced by k lines $w_1, \dots, w_k$. The invariant maintained is that whenever a line in the original circuit carries $a \in L$, these k lines carry the characteristic vector of the set corresponding to the element a. Now a gate (w, x) in the original circuit is replaced by k gates $(w_1, x_1), \dots, (w_k, x_k)$ in the new circuit. The correctness follows from the fact that meet and join operations in the original circuit correspond to set union and set intersection which in turn correspond to AND and OR operations of the characteristic vectors. □

Theorem 16 can be used to prove CC upper bounds for problems. Using an arbitrary distributive lattice instead of the Boolean lattice could help in designing polynomial size comparator circuits for the problem in question. An application of this method to design CC algorithms for the stable matching problem can be found in Mayr and Subramanian (1992) (See also Section 6.2 in Cook *et al.* (2014)). In their paper, they use the three element lattice $\{0, 1, *\}$ where $0 < 1 < *$ to solve the problem.

Now we consider comparator circuits over fixed finite lattices. Note that when characterizing the class P in terms of Boolean circuits, the fan-out of gates is required to be at least 2. In fact, Mayr and Subramanian's Mayr and Subramanian (1992) primary motivation while introducing the class CC was to study fan-out restricted circuits. We show that if comparator circuits are given the freedom to compute over any lattice (as opposed to the Boolean lattice on 2 elements), then the fan-out restriction is irrelevant.

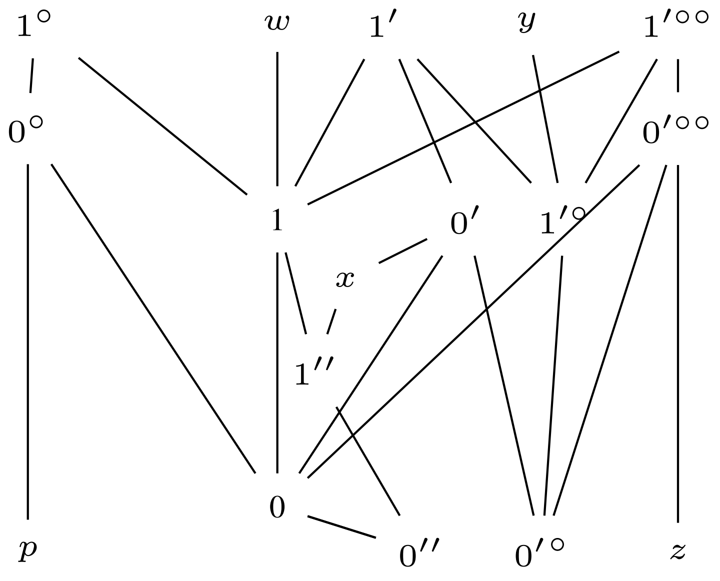


Figure 5.3: The poset for simulating P

The following lemma describes a fixed finite lattice over which comparator circuits capture P. However, it is not clear whether the class captured by comparator circuits over this lattice is independent of the accepting element. In Theorem 17, we show that there exists a lattice that captures P irrespective of the accepting element. The language MCVP consists of all tuples (C, x) where C is a Boolean circuit with only AND, OR and input gates. Here $x \in \{0, 1\}^n$ where n is the number of input gates to C and x specifies the value of each of these input gates. In the proof, we will reduce in logspace the language MCVP which is complete for the class P under logpace reductions to the comparator circuit value problem over the finite lattice given in Figure 5.4.

Lemma 12. *Let L be the lattice in Figure 5.4. Then* $\mathsf{P} = (\mathsf{L}, 1)\text{−}\mathsf{CC}$ *(Note that 1 is not* *the maximum element in the lattice).*

Proof. Let (C, x) be the input to MCVP. For each wire in C, we add a line to our comparator circuit. The initial value of the lines that correspond to the input wires of C are set to 0 or 1 of the poset shown in Figure 5.3 according to whether they are 0 or 1 in x. The comparator circuit simulates C in a level by level fashion maintaining the invariant that the lines carry 0 or 1 depending on whether they carry 0 or 1 in C. We will show how our comparator circuit simulates a level 1 OR gate of fan-out 2. The proof then follows by an easy induction.

Since $0 \leq_P 1$ an AND (OR) gate in C can be simulated by a meet (join) operation in P. The gadget shown in Figure 5.5 is used to implement the fan-out operation. The idea is that the first gate in the gadget implements the AND/OR operation and the rest of the gates in this gadget "copies" the result of this operation into the lines o_1 and o_2 that correspond to the two output wires of the gate. The reader can verify that the elements of P satisfy the following meet and join identities. Figure 5.5 shows how one could use the following identities to copy the output of $a \vee b$ into two lines (labelled o_1 and o_2).

The identity $0 \vee 1 = 1$ is used to implement the Boolean AND/OR operation. This is used by the first gate in Figure 5.5. Once the required value is computed. We add a gate between the line carrying the result of the AND/OR operation and a line with value x. As the following identities show, this makes two "copies" of the result of the Boolean operation. $0 \vee x = 0'$, $1 \vee x = 1'$, $0 \wedge x = 0''$, $1 \wedge x = 1''$

Now, the following identities can be used to convert the first copy ($0'$ or $1'$) into the original value (0 or 1). $0' \wedge y = 0'^{\circ}$, $1' \wedge y = 1'^{\circ}$, $0'^{\circ} \vee z = 0'^{\circ\circ}$, $1'^{\circ} \vee z = 1'^{\circ\circ}$, $0'^{\circ\circ} \wedge w = 0$, $1'^{\circ\circ} \wedge w = 1$

Similarly, the following identities can be used to convert the second copy ($0''$ or $1''$) into the original value (0 or 1). $0'' \vee p = 0^{\circ}$, $1'' \vee p = 1^{\circ}$, $0^{\circ} \wedge w = 0$, $1^{\circ} \wedge w = 1$

The lattice in Figure 5.4 is simply the Dedekind-MacNeille completion of P. Since the Dedekind-MacNeille completion preserves all existing meets and joins, the same computation can also be performed by this lattice.

To see that for any lattice L and any $a \in L$, $\mathsf{(L, a)}\text{−}\mathsf{CC}$ is in P, observe that in poly-time we can evaluate the n^{th} comparator circuit from the comparator circuit family for the language in (L, a)−CC. □

Lemma 12 shows that the complexity class captured by the comparator circuit could change (Assuming CC $\neq$ P) depending on the underlying lattice and the accepting element. In the following theorem, we show that if we consider any partition lattice, say Π_i, that embeds L (in Lemma 12), then the complexity class is P irrespective of the accepting element. We crucially use the fact that the circuit in the proof of Lemma 12 outputs only the elements 0 and 1 in L.

Theorem 17. *There exists a constant i such that* Π_i−CC $=$ P.

Proof. We know that there exists a finite lattice L and an $a, b \in L$ such that for any language M $\in$ P there exists a comparator circuit family over L that decides M by using a to accept and b to reject. Also $b < a$. By Pudlák and Tůma (1980), we know that there exists a constant i such that L can be embedded in Π_i. It remains to show that the accepting element used does not change the complexity. In fact, we will show that for any $X, Y \in \Pi_i$ where $X \neq Y$, we can design a comparator circuit family over Π_i that accepts M using X and rejects using Y. Let A and B be the elements in Π_i that a and b gets mapped to by this embedding ($B < A$). Then there exists a circuit family C over Π_i ,deciding M, that accepts using A and rejects using B. We will construct a circuit family C' over Π_i from C such that C' uses 1 to accept and 0 to reject. Here 1 and 0 are the maximum and minimum elements in Π_i. Now if we let x be the output of a circuit in the circuit family C, we can construct C' by computing $\mathtt{GE'}_A(x)$. Similarly, we can construct a circuit family C'' that accepts using 0 and rejects using 1 by reducing the language M to $\overline{\mathsf{MCVP}}$ and then applying the construction in Lemma 12 and then computing $\mathtt{GE'}_A(x)$ on the output of this circuit. The required circuit family is then the one computing $(X \wedge C') \vee (Y \wedge C'')$. □

If we can show that there exists a finite distributive lattice such that the poset in Figure 5.3 can be embedded in that lattice while preserving all existing meets and joins, then P $=$ CC. In the following theorem, we show that such an embedding is not possible.

Theorem 18. *The poset in Figure 5.3 cannot be embedded into any distributive lattice while preserving all meets and joins.*

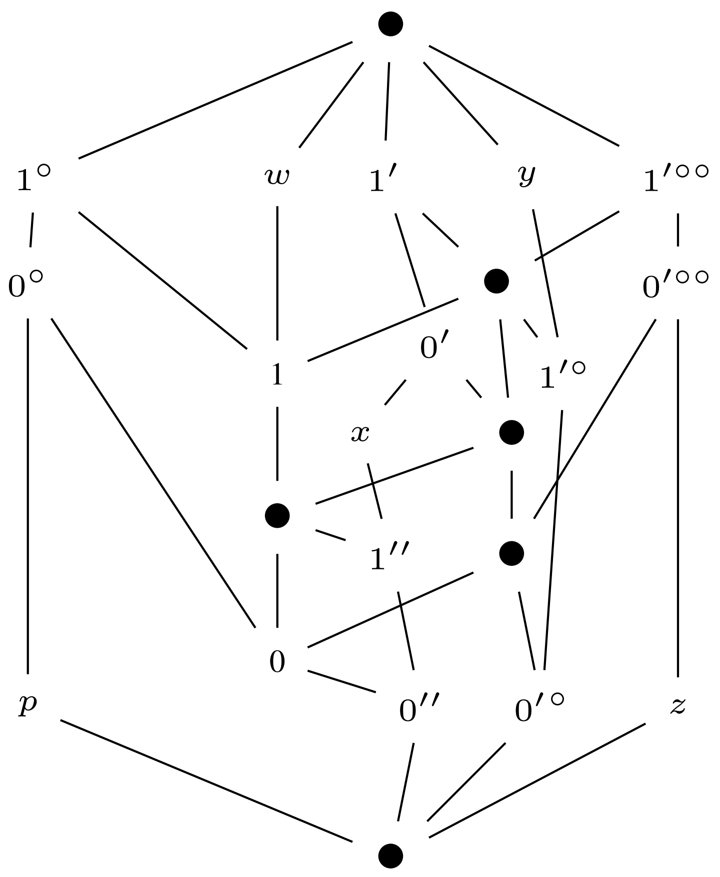


Figure 5.4: The lattice for simulating P

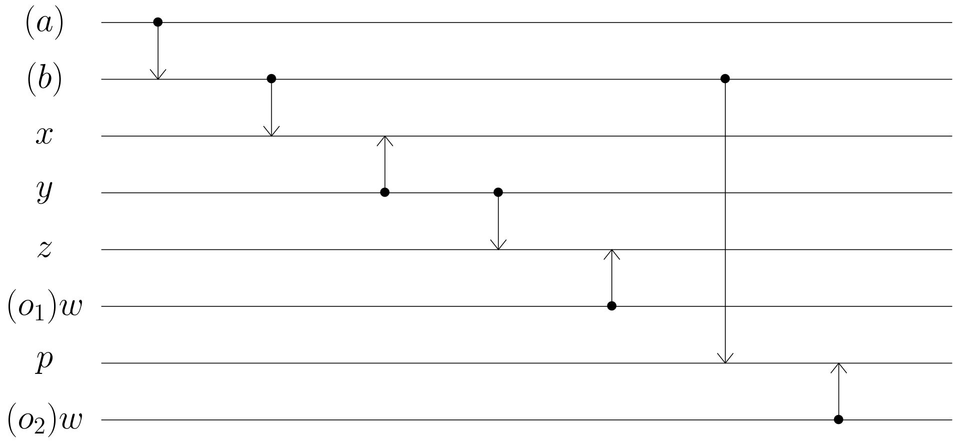


Figure 5.5: Gadget for implementing fan-out in a comparator circuit

Proof. We use proof by contradiction. Assume that such an embedding exists. Then by Birkhoff's representation theorem, the elements of the poset in Figure 5.3 can be labelled by finite sets such that lub and glb operations over the embedding distributive lattice correspond to union and intersection of these sets respectively. We will denote the set labelling each element of the poset in Figure 5.3 by the corresponding uppercase letter except that 0, 1, $0'$, $1'$, $0''$ and $1''$ are labelled by A, B, A', B', A'' and B'' respectively.

Let $\{x_1, \ldots, x_k\} = B \setminus A$. Since $A \subset B$, we have $k \geq 1$. Our first goal is to prove that all of these x_i must be in B'' too. Since $B \subset W$, we have for all i that $x_i \in W$. Now suppose for contradiction that there exists an i such that $x_i \in P$, then we can conclude that $x_i \in A$ since $A = (A'' \cup P) \cap W$. So for all i we have $x_i \notin P$. Since $B = (B'' \cup P) \cap W$ and $x_i \notin P$, we have $x_i \in B''$. But then for all i we have $x_i \in A'$ as $A' \supset B''$. So $A' \supseteq B$ which is a contradiction since A' and B are incomparable. □

5.4 Comparator Circuits over Bounded Posets

In this section, we consider the most general form of comparator circuits. i.e., we consider comparator circuits over fixed finite bounded posets. We show that the resulting complexity class is the class NP.

Theorem 19. *Let P be any poset and let $a \in P$ be an arbitrary element in P, then* $\mathsf{(P,a){-}CC} \subseteq \mathsf{NP}$. *Also, there exists a finite poset P and an $a \in P$ such that* $\mathsf{NP} = \mathsf{(P,a){-}CC}$.

Proof. First we prove that there exists a poset P and an accepting element $a \in P$ such that $\mathsf{NP} \subseteq \mathsf{(P,a){-}CC}$. Let P be the poset in Figure 5.6. We will reduce the well-known NP-complete problem SAT into (P, a)−CCVP. We can assume wlog that the circuit does not contain any NOT gates.

Note that the poset P contains the poset in the proof of Theorem 17. This is represented by the hexagon in Figure 5.6. The elements marked 0 and 1 inside this hexagon are the elements marked 0 and 1 in Figure 5.3. This containment ensures that we can implement all operations that we used while simulating MCVP to be used here as well. Let C be the input to the SAT problem. The 0 and 1 values carried by wires will be

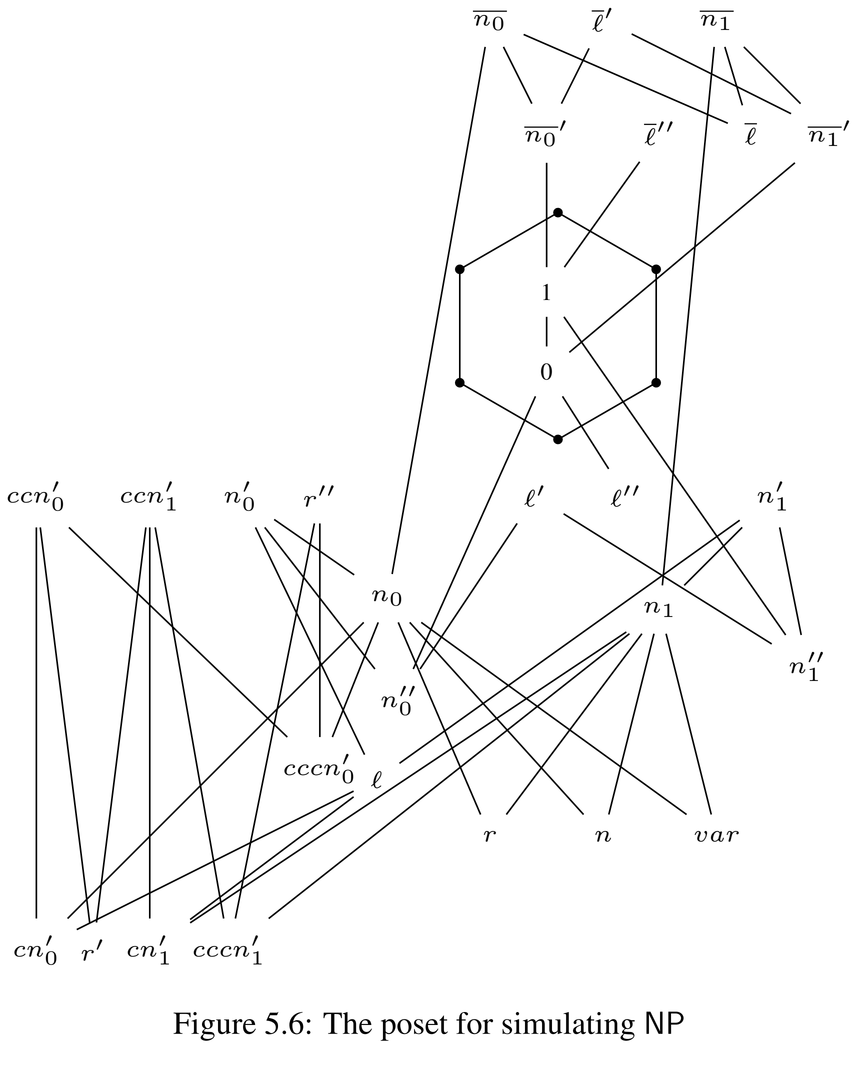


Figure 5.6: The poset for simulating NP

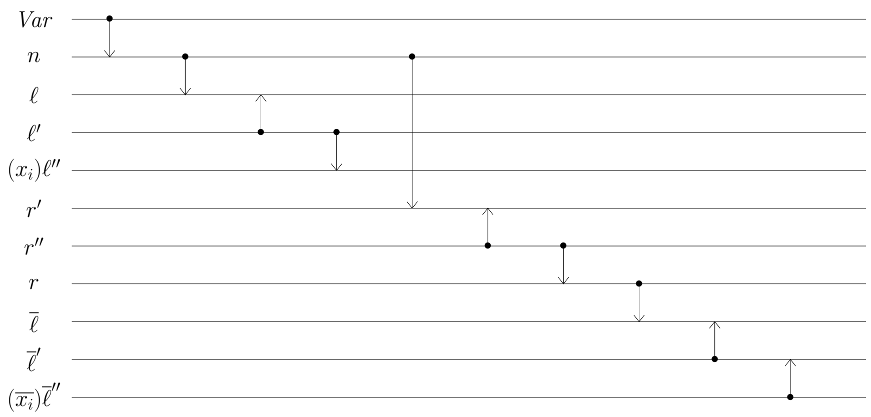


Figure 5.7: Nondeterministcally generate x_i and $\overline{x_i}$

represented by 0 and 1 in P as in the proof of Theorem 17. The non-trivial part is to simulate the input variables $x_1, \ldots, x_n$. These input variables to C are handled by non-deterministically generating 0 or 1 (of P) on the lines corresponding to the wires attached to these input gates. We also have to ensure that when we non-deterministically generate the values of input variables, the values generated for x_i and $\overline{x_i}$ are consistent. This is ensured by generating x_i non-deterministically and then complementing the generated value to get $\overline{x_i}$. The fan-out operation is implemented as in the proof of Theorem 17.

Note that the minimal upper bounds for the elements Var and n in the poset P are n_0 and n_1. These values stand for a non-deterministically generated 0 and 1 resp. Now for each variable x_i we take the minimal upper bound of these two elements in P to non-deterministically generate the value of x_i. The only thing that remains to be done is to make the corresponding $\bar{x}_i$ variable consistent. i.e., when a 0 is generated non-deterministically for x_i, we have to ensure that all lines carrying $\bar{x}_i$ in that non-deterministic path carry the value 0. The sequence of meet and join identities that we are going to describe can be used to implement this computation. Figure 5.7 shows how to generate x_i and $\overline{x_i}$ consistently in a non-deterministic fashion using the identities given below.

The following identity enables us to non-deterministically generate a 0 or a 1. Note that we are only generating n_0 and n_1 at this point. But we will later convert this into 0 or 1 that are used for implementing the Boolean operations.

$$Var \vee n = \{n_0, n_1\}$$

Now we use the following identities to convert n_0 or n_1 into a 0 or a 1 respectively.

$$\begin{aligned} \ell \vee n_0 &= n_0' & \qquad \ell \vee n_1 &= n_1' \\ \ell' \wedge n_0' &= n_0'' & \qquad \ell' \wedge n_1' &= n_1'' \\ \ell'' \vee n_0'' &= 0 & \qquad \ell'' \vee n_1'' &= 1 \end{aligned}$$

Note that the original n_0 or n_1 that was generated will be destroyed by the above sequence of operations (By doing $\ell \wedge n_0$ for ex.). Using the following identities, we ensure that the original value generated non-deterministically is restored.

$$\begin{aligned} \ell \wedge n_0 &= cn_0' & \qquad \ell \wedge n_1 &= cn_1' \\ r' \vee cn_0' &= ccn_0' & \qquad r' \vee cn_1' &= ccn_1' \\ r'' \wedge ccn_0' &= cccn_0' & \qquad r'' \wedge ccn_1' &= cccn_1' \\ r \vee cccn_0' &= n_0 & \qquad r \vee cccn_1' &= n_1 \end{aligned}$$

Now we use the restored value along with the following identities to generate the value for the line carrying $\overline{x_i}$.

$$\begin{aligned} \overline{\ell} \vee n_0 &= \overline{n_0} & \qquad \overline{\ell} \vee n_1 &= \overline{n_1} \\ \overline{\ell}' \wedge \overline{n_0} &= \overline{n_0}' & \qquad \overline{\ell}' \wedge \overline{n_1} &= \overline{n_1}' \\ \overline{\ell}'' \wedge \overline{n_0}' &= 1 & \qquad \overline{\ell}'' \wedge \overline{n_1}' &= 0 \end{aligned}$$

$\square$

5.5 Skew Comparator Circuits

In this section, we study the skew comparator circuits defined in the preliminaries. We show that SkewCC is the class L. Recall that the class NL can be characterized as the set of all languages computed by logpsace-uniform Boolean circuits with skewed AND gates. So the result in this section draws a parallel between the P vs CC problem and the NL vs L problem. It immediately follows that SkewCC over distributive lattices also characterize the class L.

We begin by considering a canonical complete problem for the class L. The language $\mathsf{DGAP1}$ consists of all tuples (G, s, t) where $G = (V, E)$ is a directed graph where each vertex has out-degree at most one and $s, t \in V$ and there is a directed path from s to t. We use a variant of $\mathsf{DGAP1}$ problem in our setting. The variant (called $\mathsf{DGAP1}'$) is that the out-degree constraint is not applied to s. It is easy to see that $\mathsf{DGAP1}'$ is also in L. Indeed, for each neighbour u of s, run the $\mathsf{DGAP1}$ algorithm to check whether t is reachable from u.

Theorem 20. $\mathsf{SkewCC} = \mathsf{L}$

Proof. ($\subseteq$) Let $\mathsf{L} \in \mathsf{SkewCC}$. We will prove that $\mathsf{L} \in \mathsf{L}$ by reducing L to $\mathsf{DGAP1}'$. The reduction is as follows. Observe that we can reduce the language L to $\mathsf{SkewCCVP}$ by a logspace reduction (Using the uniformity algorithm). Then we reduce $\mathsf{SkewCCVP}$ to $\mathsf{DGAP1}'$ as follows. Let C be an instance of $\mathsf{SkewCCVP}$. For each wire in C add a vertex to the graph G. The vertex corresponding to the output wire is the destination vertex t. Add a source vertex s. The edges of G are as follows. For each vertex v that corresponds to an input wire of C having value 1, add the edge (s, v) to the graph. Now consider a comparator gate g in C with input wires e_1 and e_2 and AND output wire e_3 and OR output wire e_4. There are two cases.

Gate g has an AND output Assume wlog that e_2 is an input wire to C. If $e_2 = 1$, then add the edges (e_1, e_3) and (e_2, e_4) to G. If $e_2 = 0$, then add the edge (e_1, e_4) to the graph G.

Gate g has an unused AND output Add the edges (e_1, e_4) and (e_2, e_4). Note that it is easy to check in logspace whether the AND output of a gate is used or not.

Simply scan forward on the input to check whether any gate in the input after g is incident on the AND output line of g or not.

It is clear that G has an s–t path if and only if C outputs 1. This follows from the observation that every vertex v in G where $v \neq s$ corresponds to a wire in C and v is reachable from s if and only if the wire corresponding to v carries the value 1. All vertices other than s in G has out-degree at most 1. Furthermore, the reduction can be implemented in logspace.

($\supseteq$) Let $\mathsf{L} \in \mathsf{L}$ and let B be a poly-sized layered branching program deciding L. We will design a skew comparator circuit C to simulate B. Let s be a state in B reading x_i and let the edge labelled 1 be directed towards a state t and let the edge labelled 0 be directed towards a state u. Then the gadget shown in Figure 5.8b simulates this part of the BP B (We say that this gadget corresponds to the state s). The truth table for this gadget is shown in the Table 5.8a. This table assumes that the lines t and u carry the value 0 initially. The value of the line labelled s will be 1 on input x just before the gates in this gadget are evaluated if and only if the input x reaches the state s in B. It is clear that after all the gates in this gadget are evaluated, the value of the line labelled t (or u) is 1 if and only if the input x reaches t (or u resp.) in B. Now the circuit C is as follows. For each state in B introduce a line in C and for each state in each layer from the first layer to the last layer, in that order, add the gates in the gadgets corresponding to these states in the same order to C. Note that the lines annotated x_i and $\overline{x_i}$ in a gadget are only used in that gadget. When these values are required again, new annotated lines are used. The line corresponding to the accepting state is the output line. The initial value of lines corresponding to each state other than the start state of B is 0 and the initial value of the line corresponding to the start state is 1. Also the circuit is a skew circuit since all used AND gates in the gadget are skew. For establishing the correctness, we observe that the following claim holds. The circuit C outputs 1 on input x if and only if there is a path in B from the start state to the accepting state on input x. To complete the correctness proof, we prove the following claim:

Claim 6. *The circuit C outputs 1 on input x if and only if there is a path in B from the start state to the accepting state on input x.*

Proof. Let the i^{th} *block* of C include all the gadgets corresponding to all the states in layer i of B.We will prove the more general claim that after all gates up to and including the i^{th} block is evaluated, if we consider all the lines that correspond to states in the $(i+1)^{\text{th}}$ layer of B, the only line that will have a value 1 will correspond to the state on $(i+1)^{\text{th}}$ layer reached on input x. We will prove this by induction on the layer number.

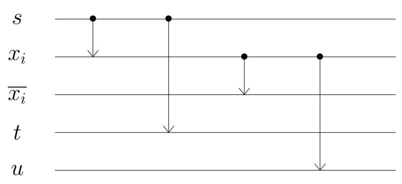
Base case: $i = 0$ Since there is only the start state in layer 1 and it is initialized to the value 1, the base case is true.

Induction Assume that the claim is true for i. Let s be the state in the $(i+1)^{\text{th}}$ layer that is reached by x and let t be the state in the $(i+2)^{\text{th}}$ that is reached by x. Now from the truth table in Table 5.8a it is clear that after the gadget for state s is evaluated the value of line t will become 1. Also, from the truth table, it is clear that the values of all the other lines that correspond to states in the $(i+2)^{\text{th}}$ layer remains 0. Notice that all gates in block $i+1$ incident on t are OR gates. So once the value of line t becomes 1, it remains so until block $i+2$. □

Let s be the number of states in B. Then the number of lines in C is at most $3s$ and the number of gates in C is at most $4s$. Since B is poly-size, so is C.

s	x_i	$\overline{x_i}$	t	u
0	1	0	0	0
0	0	1	0	0
1	1	0	1	0
1	0	1	0	1

(a) Truth Table for the gadget for BPs



(b) The gadget for simulating BPs

It is easy to see that this reduction can be implemented in NC^1. □

Since the construction in Theorem 16 preserves skewness of the circuit, we have the following corollary.

Corollary 3. *Let L be any distributive lattice and let a be any element in L, then* (L, a)–$\mathsf{SkewCC} = \mathsf{L}$.

We now turn to skew comparator circuits working over arbitrary fixed finite lattices. We show that, over the arbitrary lattices, poly-size skew comparator circuit families capture the class NL.

Theorem 21. $\mathsf{\Pi_i{-}SkewCC} = \mathsf{NL}$ *(Here the constant i is the same as in Theorem 17)*

Proof. First, we show that $\mathsf{NL} \subseteq \mathsf{\Pi_i{-}SkewCC}$, The class NL can be defined as the set of all languages computable by logspace uniform Boolean circuits where all AND gates are skewed. Applying the reduction in Theorem 17 to a skew circuit, we get a skew comparator circuit over Π_i. In other words, the reduction in Theorem 17 preserves the skewness of the circuit. Since $\mathsf{NL} = \mathsf{coNL}$, it follows that this reduction holds irrespective of the accepting element.

To show that $\mathsf{\Pi_i{-}SkewCC} \subseteq \mathsf{NL}$, we describe an NL evaluation algorithm for the circuit value problem $\mathsf{\Pi_i{-}SkewCCVP}$. i.e., given a skewed comparator circuit C over Π_i and an $A \in \Pi_i$, we can decide in NL whether the circuit C outputs A or not. Consider the undirected graph representation G of A and let $V_1, \dots V_k$ be the partitioning of the vertex set such that the graphs induced by $V_1, \dots V_k$ are maximal cliques in the graph and all edges in G are in one of these induced graphs. From now on, we consider values carried by wires in the circuit as undirected graphs representing some element in the partition lattice. Statements such as – the value of the wire w contains the edge $\{i, j\}$ – should be interpreted using this equivalence.

We define the *proof path* for an edge e (of the graph corresponding to an element in Π_i) in a circuit C as a path from an input gate of C to the output gate of C satisfying the following properties.

- The value at the input gate contains e.
- For each AND gate in the path, the other input comes from an input gate and the value of that gate contains e.

It is easy to see that a proof path for e in C shows that the output element of C contains the edge e. Also a proof path can be verified in logspace in a read-once fashion. We call a proof path partial if it does not start from the input gate.

We prove the following claim.

Claim 7. *The circuit C outputs a value greater than or equal to A iff there exist spanning trees T_i for $G[V_i]$ for each i such that for each edge $e \in T_i$, there exists a proof path in C for e.*

Proof. (if) The proof path for e implies that the output element must contain the edge e. Since the output elements are equivalence relations, existence of spanning trees imply that all the edges connecting any two vertices in any spanning tree are present. Finally, since any edge of A must be present in at least one of the $G[V_i]$s, it follows that the output element must be at least A.

(only if) Start with a partitioning V_1', V_1'' (which is a cut) of the vertex set V_1. We will show that there exists an edge e crossing this cut that has a proof path in C. The output gate has value A. So there is an edge $e = \{a, b\}$ that crosses this cut and e is in the value of the output wire. Now assume that we have a partial proof path for e in C such that e crosses the cut. Now consider the gate from which the current partial proof path starts from. If it is an AND gate, then extend the partial path by including the input wire to the AND gate that does not come from an input gate (if there is no such wire, we are done). If it is an OR gate, then there exists a path $\{a, v_1\}, \ldots \{v_j, b\}$ such that each edge in this path is contained in the values of one of the input wires to the OR gate. Let e' be an edge in this path that crosses the cut. Extend the partial path to include the input wire to the OR gate whose value contains e'. Now we have a partial proof path for e' in C such that e' crosses the cut. Continuing this process, we will end up with a proof path for some edge that crosses the cut.

Apply this procedure recursively on V_1' and V_1'' to obtain valid paths for each edge in some spanning tree of V_1. Then do this for each i from 1 to k to obtain all required proof paths. □

This claim gives an NL algorithm to check whether the output of C is greater than or equal to A. Simply guess the spanning trees T_i (which are constant sized) and then guess the proof paths which are verified in a read-once fashion. We can also check in NL whether the output if strictly greater than A by guessing one additional edge e that is not in A. Since $\mathsf{NL} = \mathsf{coNL}$, we can also check in NL whether the output is $\not> A$. So to check whether the output is equal to A, check that the output is $\geq A$ and $\not> A$. □

5.6 Formulae over Lattices

It is well known that languages decided by poly-size formulae is the class NC^1. By definition, the class NC^1 is also the class of languages decided by log-depth Boolean circuits with bounded fan-in AND and OR gates. We can modify Definition 43 to define formulae over finite bounded posets. We denote by (L, a)–$\mathsf{Formulae}$, where L is a lattice and $a \in L$, the class of all languages decided by poly-size formulae over L using a as the accepting element. In this section, we show that the languages computed by poly-sized formulae over any fixed finite lattice is the class NC^1. The proof for the Boolean case is by P.M.Spira (1971) and it works by depth reducing an arbitrary formula of poly size to a Boolean formula of poly size and log depth. The depth reduction is done by identifying a separator vertex in the tree and then evaluating the separated components (which are smaller circuits) in parallel. We show that a similar argument can be extended to the case of finite lattices as well. Our main theorem in this section is the following.

Theorem 22. *Let L be any finite lattice and let a be an arbitrary element in L. We have* (L, a)–$\mathsf{Formulae} = \mathsf{NC}^1$.

Proof. ($\supseteq$) Any lattice with at least 2 elements contains the 0–1 lattice as a sublattice. Also since NC^1 is closed under complementation, the class does not change even if the acceptor is 0.

($\subseteq$) Let F be a poly-size formula family over L. Let i be such that L can be embedded in Π_i. Let F' be the formula family over Π_i that corresponds to F. We will now construct a log-depth poly-size formula family F'' that computes the same language as F'. We will use F' to denote a formula in the family F'. Let v be the tree separator of the tree corresponding to F'. For each $a_i \in \Pi_i$, we will construct two formulae. The first one, say F_1^v, computes the value at the root of F' assuming that value at v is a_i and the other, say F_2^v computes the value at the node v and applies $\mathtt{GE'}_{a_i}$ on that value. Then we compute the sub-formula $F_1^v \wedge F_2^v$. After that we take the lub over all such sub-formulae (one for each a_i). This construction is applied recursively on F_1^v and F_2^v to obtain a log-depth poly-size formula equivalent to F.

Suppose the correct value of the sub-formula of F' rooted at v is a_i. Then the only sub-formulae $F_1^v \wedge F_2^v$ outputting a non-zero value are the ones corresponding to

$a_j \leq a_i$. The non-zero value output by such a sub-formula is b_j, the value obtained at the root when the value of v is a_j. But we know that b_i, the actual value of the original formula is greater than or equal to the value b_j of any sub-formula by monotonicity of lub and glb. So the topmost lub will always output the correct value b_i.

The final formula is log-depth, poly-size since the formulae $\mathtt{GE}'_a$ is constant depth. Now we can construct an NC^1 circuit from F'' by encoding each element in Π_i in binary and replacing each gate in F'' by constant-sized circuits computing the lub and glb over Π_i. $\square$

5.7 Comparator Circuits over Growing Lattices

We can generalize the comparator circuit model even further by allowing it to compute over lattices that grow with the size of the input. If the size of the lattice is polynomial in the size of the input and if the lattice can be computed by the uniformity machine, then the languages computed by these circuits are in the class P. However, since we have the freedom to change the lattice according to the size of the input, we may be able to capture the class P using structurally simpler lattices. It is conceivable that the class P could be captured by a family of distributive lattices, while no finite lattice capturing P can be distributive.

In this section, we present a formal definition of comparator circuits over growing posets and then present a lattice family that capture the class P. Then we will show that, even for this simpler lattice, an embedding to a family of distributive lattices is not possible (Similar to Theorem 18).

Definition 46 (Comparator Circuits over Growing Bounded Posets)**.** *A comparator circuit family over a growing bounded poset family $P = \{P_n\}$ with accepting set $A = \{A_n\}$ where $A_n \subseteq P_n$ is a family of circuits $C = \{C_n\}_{n \geq 0}$ where $C_n = (W, G, f)$ where $f : W \mapsto (P_n \cup \{(i, g) : 1 \leq i \leq n$ and $g : \Sigma \mapsto P_n\})$ is a comparator circuit. Here $W = \{w_1, \dots, w_m\}$ is a set of lines and G is an ordered list of gates (w_i, w_j).*

On input $x \in \Sigma^n$, we define the output of the comparator circuit C_n as follows. Each line is initially assigned a value according to f as follows. We denote the value of the line w_i by $val(w_i)$. If $f(w) \in P_n$, then the value is the element $f(w)$. Otherwise $f(w) =$

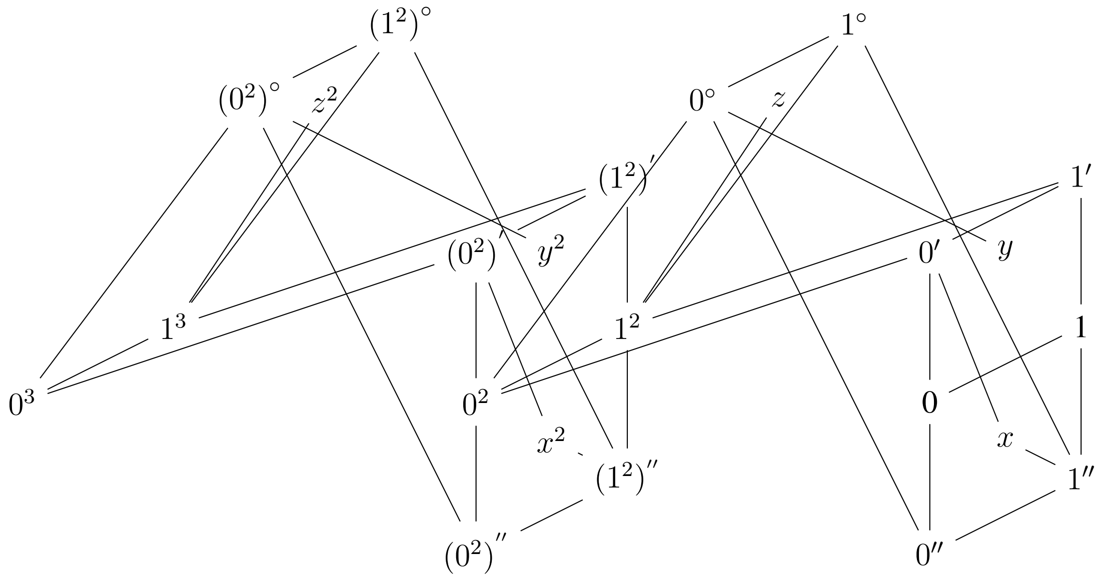


Figure 5.9: A growing poset family for simulating P

(i, g) and the initial value is given by $g(x_i)$. A gate (w_i, w_j) (non-deterministically) updates the value of the line w_i into $val(w_i) \wedge val(w_j)$ and the value of the line w_j into $val(w_i) \vee val(w_j)$. The values of lines are updated by each gate in G in order and the circuit accepts x iff $val(w) = a \in A_n$ at the end of the computation for some sequence of non-deterministic choices.

Let Σ be any finite alphabet. A comparator circuit family C over a growing bounded poset family P_n with an accepting $A_n \subseteq P_n$ decides $\mathsf{L} \subseteq \Sigma^*$ iff $C_{|x|}$ correctly decides whether $x \in \mathsf{L}$ for all $x \in \Sigma^*$.

The circuit family is called P-uniform if there exists a TM that given 1^n as input runs in poly*(n) time and outputs P_n, A_n and C_n.*

First, we show a lattice family that captures P.

Theorem 23. *The comparator circuit family over DM completions for the poset family in Figure 5.9 captures the class* P.

Proof. We construct a comparator circuit over the poset family in Figure 5.9 from a layered circuit with NOT gates only at the input level. The elements 0^i and 1^i in the poset correspond to the logical values 0 and 1 at the i^{th} level of the circuit. As in the proof of Lemma 12, there is a sequence of lubs and glbs that creates two copies of the logical value at the i^{th} level and then convert them to the corresponding value in the $(i+1)^{th}$ level.

We can prove that the DM completion of the poset P_n has at most $O(m^{11})$ elements, where $m = |P_n|$. The elements of the DM completion of P_n consists of ordered pairs (A, B) where $A, B \subseteq P_n$ and $A = UP(B)$ and $B = DOWN(A)$. Here $UP(A)$ ($DOWN(A)$) is the set of all elements in the poset that is greater (less) than or equal to all elements in A. Note that in the poset P_n, if $|A| > 11$, then we have $DOWN(A) = \phi = B$ and then we have $A = P_n$. Therefore the DM completion has at most $O(m^{11})$ elements. Also, there exists an algorithm that can compute this completion in $\mathsf{poly}(n)$ time (Ganter and Kuznetsov (1998)). The P-uniformity of the comparator circuit family follows. □

Now we prove that even this growing lattice family cannot be embedded into any distributive lattice.

Theorem 24. *The poset in Figure 5.9 cannot be embedded in any distributive lattice.*

Proof Sketch. The proof is similar to the proof of Theorem 18. We use the same labelling used in the proof of Theorem 18.

We have $A^2 = (A'' \cup Y) \cap Z = A' \cap Z$ and $B^2 = (B'' \cup Y) \cap Z$. Since $B^2 \supset A^2$, we have $x \in B^2 \backslash A^2$. So $x \in Z$ and $x \in (B'' \cup Y) \backslash A'$. Now if $x \in Y$, then $x \in A'' \cup Y$ and so $x \in A^2$. But if $x \notin Y$, then $x \in B''$ which implies $x \in A'$ which in turn implies $x \in A^2$. A contradiction. □

CHAPTER 6

Conclusion and Open Problems

We contributed to Cook's Program (Chapter 3), which aims to make progress towards settling L vs P, by designing problems that are deemed suitable for separating L from P. This program involves proving lower bounds for semantically restricted models of computation that capture known algorithms to the problem in question. Such lower bounds help us to determine the limitations of current algorithms for solving the problem and may lead to better algorithms. Our main contribution to this program was to prove super-logarithmic space lower bounds for bitwise independent non-deterministic thrifty branching programs, a computational model that can implement the currently known non-deterministic algorithm to solve this problem. The obvious open problem is to prove lower bounds for NTBPs without the bitwise independence restriction.

Problem 1. *Prove that any NTBP solving* TEP *must have super-polynomial size.*

We also suggest trying to prove lower bounds for a computational model that is intermediate in power to BINTBPs and NTBPs – namely nodewise independent NTBPs. In a NINTBP, we relax the bitwise independence condition and require only that the values at different nodes of the tree be independent.

Problem 2. *Prove that any NINTBP solving* TEP *must have super-polynomial size.*

We also studied the tools used to prove this lower bound – namely, the entropy method and pebbling games. We then showed that the entropy method is very general in that it can be used to prove any lower bound on BPs (Although, the proof of this fact does not yield any insight on how to go about proving such lower bounds).

We also did a combinatorial study of pebbling games and showed that it is connected to a classical graph parameter (at least when restricted to trees). An obvious open problem arising out of this work is whether all trees have polynomial time optimal pebblings.

Problem 3. *Prove or disprove: Trees can be optimally pebbled in at most n^k steps, where k is a constant with respect to the number of nodes in the tree n.*

We then looked at bounded depth Boolean circuits (Chapter 4), a restricted class of Boolean circuits that characterize natural complexity classes inside L. We saw that these circuits essentially characterize efficient parallel algorithms and designed such circuits for many problems not known to have constant depth circuits. The main open problem here is to improve upper bounds for the perfect matching problem on constant width grid graphs.

Problem 4. *Prove that testing whether a manhattan graph has a perfect matching can be done in* AC^0.

We then saw that the class CC can be used to characterize hardness of parallelization for some problems. Our contribution here was to show that the classical complexity classes such as P, NP, L, and NL can be characterized in terms of comparator circuits. We do this by introducing a novel way to generalize circuits – allowing them to compute over arbitrary lattices instead of the 0-1 lattice. We saw that using a distributive lattice does not alter the computational power of comparator circuits and also that there exists a non-distributive lattice that allows comparator circuits to solve all languages in P. An interesting question is whether *all* non-distributive lattices have this power.

Problem 5. *Prove or disprove: For any non-distributive lattice L, polynomial size comparator circuit families over L characterize* P.

A proof of this statement would imply the existence of a dichotomy between CC and P based on lattice structure.

REFERENCES

1. **Arora, S.** and **B. Barak**, *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. ISBN 978-0-521-42426-4.

2. **Baker, T. P.**, **J. Gill**, and **R. Solovay** (1975). Relativizations of the P =? NP question. *SIAM Journal on Computing*, **4**(4), 431–442.

3. **Barrington, D. A.** and **P. McKenzie** (1991). Oracle branching programs and logspace versus p. *Information and Computation*, **95**(1), 96 – 115. ISSN 0890-5401.

4. **Barrington, D. A. M.** (1989). Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, **38**(1), 150–164.

5. **Barrington, D. A. M.**, **C. Lu**, **P. B. Miltersen**, and **S. Skyum**, Searching constant width mazes captures the AC^0 hierarchy. *In* **M. Morvan**, **C. Meinel**, and **D. Krob** (eds.), *Proceedings of 15th Annual Symposium on Theoretical Aspects of Computer Science, STACS 98, Paris, France, February 25-27, 1998*, volume 1373 of *Lecture Notes in Computer Science*. Springer, 1998. ISBN 3-540-64230-7.

6. **Barrington, D. A. M.** and **D. Thérien** (1988). Finite monoids and the fine structure of NC^1. *Journal of the ACM*, **35**(4), 941–952.

7. **Bennett, C. H.** (1989). Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, **18**(4), 766–776.

8. **Chan, S. M.**, Just a pebble game. *In Proceedings of the 28th Conference on Computational Complexity, CCC 2013, Palo Alto, California, USA, 5-7 June, 2013*. IEEE, 2013.

9. **Cook, S.** and **R. Sethi**, Storage requirements for deterministic / polynomial time recognizable languages. *In Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74. ACM, New York, NY, USA, 1974.

10. **Cook, S. A.**, Path systems and language recognition. *In* **P. C. Fischer**, **R. Fabian**, **J. D. Ullman**, and **R. M. Karp** (eds.), *Proceedings of the 2nd Annual ACM Symposium on Theory of Computing, May 4-6, 1970, Northampton, Massachusetts, USA*. ACM, 1970.

11. **Cook, S. A.** (1974). An observation on time-storage trade off. *J. Comput. Syst. Sci.*, **9**(3), 308–316.

12. **Cook, S. A.**, **Y. Filmus**, and **D. T. M. Le** (2014). The complexity of the comparator circuit value problem. *ACM Transactions On Computation Theory TOCT*, **6**(4), 15:1–15:44.

13. **Cook, S. A.**, **P. McKenzie**, **D. Wehr**, **M. Braverman**, and **R. Santhanam** (2012). Pebbles and branching programs for tree evaluation. *ACM Transactions on Computation Theory (TOCT)*, **3**(2), 4.

14. **Davey, B. A.** and **H. A. Priestley**, *Introduction to lattices and order*. Cambridge University Press, 1990. ISBN 0521365848 9780521365840 0521367662 9780521367660.

15. **Dymond, P. W.** and **M. Tompa** (1985). Speedups of deterministic machines by synchronous parallel machines. *J. Comput. Syst. Sci.*, **30**(2), 149–161.

16. **Gál, A.**, **M. Koucký**, and **P. McKenzie** (2008). Incremental branching programs. *Theory of Computing Systems*, **43**(2), 159–184. ISSN 1432-4350.

17. **Ganter, B.** and **S. O. Kuznetsov**, Stepwise construction of the Dedekind-MacNeille completion. *In Conceptual structures: theory, tools and applications. 6th international conference, ICCS '98, Montpellier, France, August 10–12, 1998. Proceedings*. Berlin: Springer, 1998. ISBN 3-540-64791-0, 295–302.

18. **Hansen, K. A.**, **B. Komarath**, **J. Sarma**, **S. Skyum**, and **N. Talebanfard**, Circuit complexity of properties of graphs with constant planar cutwidth. *In* **E. Csuhaj-Varjú**, **M. Dietzfelbinger**, and **Z. Ésik** (eds.), *Proceedings of 39th International Symposium on Mathematical Foundations of Computer Science 2014, MFCS 2014, Budapest, Hungary, August 25-29, 2014, Part II*, volume 8635 of *Lecture Notes in Computer Science*. Springer, 2014. ISBN 978-3-662-44464-1.

19. **Hartmanis, J.** and **R. E. Stearns**, Computational complexity of recursive sequences. *In 5th Annual Symposium on Switching Circuit Theory and Logical Design, Princeton, New Jersey, USA, November 11-13, 1964*. IEEE Computer Society, 1964.

20. **Hopcroft, J. E.**, **W. J. Paul**, and **L. G. Valiant** (1977). On time versus space. *Journal of the ACM*, **24**(2), 332–337.

21. **Iwama, K.** and **A. Nagao**, Read-once branching programs for tree evaluation problems. *In* **E. W. Mayr** and **N. Portier** (eds.), *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, volume 25 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014. ISBN 978-3-939897-65-1.

22. **JáJá, J.**, *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992. ISBN 0-201-54856-9.

23. **Jukna, S.** and **S. Zák** (2001). On uncertainty versus size in branching programs. *Electronic Colloquium on Computational Complexity (ECCC)*, **8**(39).

24. **Komarath, B.** and **J. Sarma** (2015). Pebbling, entropy, and branching program size lower bounds. *ACM Transactions On Computation Theory (TOCT)*, **7**(2), 8.

25. **Komarath, B.**, **J. Sarma**, and **S. Sawlani**, Reversible pebble game on trees. *In* **D. Xu**, **D. Du**, and **D. Du** (eds.), *Proceedings of 21st International Conference on Computing and Combinatorics, COCOON 2015, Beijing, China, August 4-6, 2015*, volume 9198 of *Lecture Notes in Computer Science*. Springer, 2015*a*. ISBN 978-3-319-21397-2.

26. **Komarath, B.**, **J. Sarma**, and **K. S. Sunil**, Comparator circuits over finite bounded posets. *In* **M. M. Halldórsson**, **K. Iwama**, **N. Kobayashi**, and **B. Speckmann** (eds.), *Proceedings of 42nd International Colloquium on Automata, Languages, and Programming, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Part I*, volume 9134 of *Lecture Notes in Computer Science*. Springer, 2015*b*. ISBN 978-3-662-47671-0.

27. **Kozen, D.**, *Automata and computability*. Undergraduate texts in computer science. Springer, 1997. ISBN 978-0-387-94907-9.

28. **Lam, T. W.** and **F. L. Yue** (2001). Optimal edge ranking of trees in linear time. *Algorithmica*, **30**(1), 12–33.

29. **Liu, D.** (2013). Pebbling arguments for tree evaluation. *CoRR*, **abs/1311.0293**.

30. **Mayr, E. W.** and **A. Subramanian** (1992). The complexity of circuit value and network stability. *Journal of Computer and System Sciences*, **44**(2), 302–323.

31. **P.M.Spira**, On time-hardware complexity tradeoffs for boolean functions. *In Proceedings of 4th Hawaii Symp. on System Sciences*. 1971.

32. **Pudlák, P.** and **J. Tůma** (1980). Every finite lattice can be embedded in a finite partition lattice. *Algebra Universalis*, **10**(1), 74–95. ISSN 0002-5240.

33. **Raz, R.** and **P. McKenzie** (1999). Separation of the monotone NC hierarchy. *Combinatorica*, **19**(3), 403–435.

34. **Razborov, A. A.**, Lower bounds for deterministic and nondeterministic branching programs. *In* **L. Budach** (ed.), *Fundamentals of Computation Theory, 8th International Symposium, FCT '91, Gosen, Germany, September 9-13, 1991, Proceedings*, volume 529 of *Lecture Notes in Computer Science*. Springer, 1991. ISBN 3-540-54458-5.

35. **Sipser, M.**, *Introduction to the theory of computation*. PWS Publishing Company, 1997. ISBN 978-0-534-94728-6.

36. **Vanderzwet, F.** (2013). Fractional Pebbling Game Lower Bounds. *ArXiv e-prints*.

37. **Walker, S.** and **H. Strong** (1973). Characterizations of flowchartable recursions. *Journal of Computer and System Sciences*, **7**(4), 404 – 447. ISSN 0022-0000.

38. **West, D. B.**, *Introduction to Graph Theory*. Prentice Hall, 2000, 2 edition. ISBN 0130144002.

39. **Yannakakis, M.** (1985). A polynomial algorithm for the min-cut linear arrangement of trees. *Journal of the ACM*, **32**(4), 950–988.

LIST OF PAPERS BASED ON THESIS

Komarath, B. and **J. Sarma** (2015). Pebbling, entropy, and branching program size lower bounds. *ACM Transactions On Computation Theory (TOCT)*, **7**(2), 8.

Komarath, B., **J. Sarma**, and **S. Sawlani**, Reversible pebble game on trees. *In* **D. Xu**, **D. Du**, and **D. Du** (eds.), *Proceedings of 21st International Conference on Computing and Combinatorics, COCOON 2015, Beijing, China, August 4-6, 2015*, volume 9198 of *Lecture Notes in Computer Science*. Springer, 2015*a*. ISBN 978-3-319-21397-2.

Komarath, B., **J. Sarma**, and **K. S. Sunil**, Comparator circuits over finite bounded posets. *In* **M. M. Halldórsson**, **K. Iwama**, **N. Kobayashi**, and **B. Speckmann** (eds.), *Proceedings of 42nd International Colloquium on Automata, Languages, and Programming, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Part I*, volume 9134 of *Lecture Notes in Computer Science*. Springer, 2015*b*. ISBN978-3-662-47671-0. *Invited to the special issue of the Journal "Information and Computation"*

Hansen, K. A., **B. Komarath**, **J. Sarma**, **S. Skyum**, and **N. Talebanfard**, Circuit complexity of properties of graphs with constant planar cutwidth. *In* **E. Csuhaj-Varjú**, **M. Dietzfelbinger**, and **Z. Ésik** (eds.), *Proceedings of 39th International Symposium on Mathematical Foundations of Computer Science 2014, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Part II*, volume 8635 of *Lecture Notes in Computer Science*. Springer, 2014. ISBN 978-3-662-44464-1.

Komarath, B. and **J. Sarma**, Pebbling, entropy and branching program size lower bounds. *In* **N. Portier** and **T. Wilke** (eds.), *Proceedings of 30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, volume 20 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. ISBN 978-3-939897-50-7.