Computational Complexity of Tree Evaluation Problems and Branching Program Satisfiability Problems

Atsuki Nagao

Contents

List of Figures

Abstract

Computational Complexity of Tree Evaluation Problems and Branching Program Satisfiability Problems

Atsuki Nagao

2015

This thesis is concerned with the "**P** vs. **L**"question which is one of the most famous open questions in theoretical computer science. Here, the class **P** is the set of problems which polynomial-time Turing machines can solve and the class **L** the set of problems which logarithmic space Turing machines can solve. Cook, McKenzie, Wehr, Braverman and Santhanam recently introduced the *tree evaluation problem* (*TEP*) and they claimed the conjecture that any branching program solving tree evaluation problems must need super-polynomial states (therefore, the tree evaluation problems are not in **L**.) But they were actually able to show only a lower bound of $\Omega(n^{3/2}/(\log n)^{5/2})$, which is by far insufficient for the separation of **L** and **P**.

To improve upper bounds is also important for theoretical computer science. For NP-complete problems, to design some algorithm running faster than an exhaustive search is useful. Recently, many variety of satisfiability problems are extensively studied. However, there are a very few researches on the satisfiability of branching program. For more fundamental problems, some kind of sorting problems are also studied.

The first result in this thesis is a super-polynomial lower bound of branching programs with a restriction. Cook and et al. also show a super-polynomial lower bound with some restriction against branching programs, but this restriction is related to the order of reading variable, so it is so-called a "semantic" restriction and it seems very difficult to get rid of this restriction. Our restriction is a syntactic and well-known one called the "read-once" restriction. Introducing this read-once restriction, we can show a branching program lower bound of $\Omega(k^h) = \Omega(n^{\log n})$. Note that this lower bound is tight, meaning that the well-known natural construction of branching programs for TEP is best possible under the read-once restriction. This fact also supports the conjecture for **L**$\neq$**P**.

Our second result is an improvement of the Cook and other author's method. Note that their method is only applicable to tree evaluation problems with height three, binary complete trees. We modify this method so as to be available for TEPs for height three, d-ary complete trees, where d is any constant. By this modification, we obtain a lower bound of $\Omega(n^{(2d-1)/d}/(\log n)^{(3d-1)/d})$. This lower bound is again tight.

The third result is about a satisfiability problem for branching programs. A k-indexed Binary Decision Diagram (abbr. k-IBDD) is a branching program with k-layers and each layer consists of an Ordered Binary Decision Diagram (abbr. OBDD). k-IBDD Satisfiability Problem (abbr. k-IBDD SAT) asks whether given a k-IBDD, there exists a consistent path from a root to sink 1 or not. It is known that k-IBDD SAT is NP-complete for any $k \geq 2$. By exhaustive search, k-IBDD SAT with n variables and m states can be solved in $O(m2^n)$ time. One of the important goal for NP-complete problems is to design an algorithm super-polynomially faster than the exhaustive search such as an $O(m2^{n-\omega(\log n)})$ time algorithm. This thesis proposes such an algorithm. Namely, when we are given a k-IBDD with n variables and $m = O(n^c)$ states, our algorithm solves it in $poly(n) \cdot 2^{n-n^\alpha}$ time, where $\alpha = \frac{1}{2^{k-1}}$ and poly(n) a polynomial of n

Our last result is about the complexity of some kind of sorting. One of the basic sorting problems is the Swap-Sort problem: Given a list of n integer numbers in non-increasing order, if we are only allowed to swap two adjacent rows, how many swaps do we need to sort them in non-decreasing order? Peter Winkler introduced "Sorting Pairs in Bins" and Ito, Teruyama and Yoshida extended it to more general problem "Sorting k-Sets in Bins" as below. When we are given n numbered bins each with k numbered balls, such that bin i is adjacent to bins $i-1$ and $i+1$, bin n is not adjacent to bin 1, and the balls in bin i are each numbered $n+1-i$, our task is to get every ball into the bin carrying its number swapping any two balls between adjacent bins. Then the lower bound of a number of swapping is $(1-\frac{k-1}{2k^2+k-1})\frac{k+1}{4}n^2+O(n)$. We propose an algorithm which uses $\frac{k+1}{4}n^2+O(kn)$ swaps, this upper bound asymptotically approaches to the lower bound. We also propose an algorithm which uses less swaps when $k = 3$ and upper bound is $\frac{15}{16}n^2 + O(n)(= 0.9375n^2 + O(n))$.

Acknowledgement I would like to thank my supervisor Professor Kazuo Iwama. His lecture for under graduate students got me interested in the computational complexity. He introduced the computational complexity when I joined the laboratory, I was very happy to start studying on the computational complexity. Through deep discussions with him, I learned how to make the idea and the skills for making robust and brief proofs and how a researcher should be.

I am very thankful to all members of Iwama Lab. for giving me helpful advices and encouragement. In particular, I would like to thank Professor Suguru Tamaki for helpful discussions and suggestions.

I would like to express my gratitude to Professor Toniann Pitassi and students of theory group in Department of Computer Science, University of Toronto. Staying Toronto and discussing among members helped me so much.

I would like to thank all my co-authors for grateful helps during my studies: Kazuhisa Seto and Junichi Teruyama.

Finally, I would like to thank all members of Kyoto University Chorus or Ensemble Reed and my family, who have supported my study at Kyoto University.

Atsuki Nagao

Kyoto, January 7, 2015

Chapter 1

Introduction

Computational complexity theory is a major part of theoretical computer science, which forces on classifying computational problems in terms of necessary resources such as computational time and storage space. A specific computational problem is given as problem instances and a description for the yes/no solutions for these instances. For example, Primarily Testing gives us, as an instance, a natural number k and the solution for this k is yes if and only if k is prime. k is encoded (in a natural fashion, for instance, as a binary number) to a string of length n that becomes input data to an algorithm. Then a solution is computed by the algorithm and its required computation time and memory space are determined as functions in n. Thus each problem needs specific amounts of time and space, which are given as functions $T(n)$ and $S(n)$, respectively. When we analyse the complexity of such a problem, we usually evaluate these T and S from both upper bound and lower bound. Note that the functions have a wide variety of increasing speed; they can be polynomial in n, exponential in n and logarithmic in n depending on the hardness of each problem.

For example, it is a famous result that Primality testing requites only polynomial time [3][27], namely it belongs to class **P**. Class **P** is defined as the family of all decision problems which can be solved deterministically in polynomial time. The equally important class is class **NP** that contains all decision problems which can be solved by "non-deterministic" Turing machines using a polynomial amount of computation time. From this definition, Class **NP** obviously includes class **P**. But there is no known specific problem that is in **NP** but not in **P**. This leads to the question "Is **NP** equal to **P**," called the "**P** vs **NP**" question. Recall that this is one of Millennium Prize Problems stated by the Clay Mathematics Institute[27].

P and **NP** are by far more popular than other classes, but we do have several interesting

classes other than these ones. For instance, **L** is a family of problems that are solvable by Turing machines restricted with logarithmic space (and hence obviously included by **P**). Furthermore, there are several different classes around this **L**:

$$\mathbf{AC^0(6)} \subseteq \mathbf{NC^1} \subseteq \mathbf{L} \subseteq \mathbf{LogDCFL} \subseteq \mathbf{AC^1} \subseteq \mathbf{NC^2} \subseteq P \tag{1.1}$$

Here, **LogDCFL** includes decision problems that are reducible to the problem of deciding membership in a context-free language with log space. In other words, a problem in **LogDCFL** is solvable by Turing machine with a read only input tape, restricted to use a stack and an amount of memory logarithmic in the size of the input. NC^i is the class of decision problems solvable by a uniform family of Boolean circuits, with polynomial size, depth $O((log(n))^i)$ using fan-in-two gates. And $\mathbf{AC^i}$ is similarly defined using unbounded fan-in gates. $\mathbf{AC^0(6)}$ can use mod 6 gates too. Like the "**P** vs **NP**" problem, separation between these classes are of course important goals of complexity theory, especially separation of **L** and **P**.

In [10][47], Cook, McKenzie, Wehr, Braverman and Santhanam recently introduced the *tree evaluation problem* (*TEP*) and they claimed the conjecture that any branching program solving tree evaluation problems must need super-polynomial states. It is not hard to show that a deterministic logspace-bounded polytime auxiliary pushdown automaton decides Tree Evaluation Problems: there is an algorithm like depth first search to solve tree evaluation problems. This implies by [45] that Tree Evaluation Problems belong to the class **LogDCFL**, which lies between **L** and **LogDCFL**, but not be known about relationship with **NL**. Then, if **LogDCFL** is separated from **L**, we can separate $\mathbf{NC^1}$ and $\mathbf{NC^2}$.

More in detail, they consider the Tree Evaluation Problem in class **LogDCFL** for trying to separate **LogDCFL** from **NL**. They paid attention to the space of Turing Machines solving Tree Evaluation Problems. For discussing rather lower classes of space complexity Branching Programs are often used because it is known that Branching Programs can simulate Turing Machines. Branching Programs are computation models which represent a computation flow as a path in directed acyclic graph. When a Turing machine uses $s(n)$ states to solve a problem with input length n, we can construct a Branching program with $c^{s(n)}$ size, for $S(n) \geq \log n$ and some constant c [35]. This leads to the fact that the problem cannot be solved by Turing Machines with log-space if the number of Branching Program states become super-polynomial. Then we can conclude that the problem does not belong to class **L**. But, we have never seen such a large lower bound for any specific problem. The best known lower bound is only $\Omega(n^2/(\log n)^2)$ by Nečiporuk[36]. And this lower

bound technique suggests that if we use the same approach, we cannot show stronger lower bounds. Consequently we need to develop some new techniques for discovering enough large lower bounds. And [10] introduced "a state sequence method" for calculating lower bounds of states for Branching Programs solving Tree Evaluation Problems. They show a lower bound of Tree Evaluation Problem with small height and width by using this method, which however is not better than the one by the Nečiporuk's method ($\Omega(n^{3/2}/(\log n)^{5/2})$).

To improve upper bounds is also important for theoretical computer science. For NP-complete problems, to design some algorithm running faster than an exhaustive search is useful. SAT is one of central problems in theoretical computer science. There exists many variants of SAT and many research have been done. In many cases, they are known to be NP-complete. Recently, many variants of satisfiability problems are extensively studied. However, there are a very few researches on the satisfiability of branching programs. Therefore, addition to above argument, we focus on branching program satisfiability problems. For general branching program satisfiability problems, [8] design an deterministic algorithm running in $O(2^{n-\omega(\log n)})$ time, which solves any instance with n variables and $m = n^{2-\epsilon}$ states, where ϵ is an arbitrary small positive constant. There exist polynomial-time algorithms solving branching program satisfiability problems with some restriction so-called k-OBDD [4]. For more fundamental problems, some kind of sorting problems are also studied. In theoretical computer science, various kinds of sorting problems have been studied [15, 16, 18].

In this research, we first obtain an exponential lower bound for Tree Evaluation Problem under a restriction of branching programs. [10] also shows an exponential lower bound by introducing a restriction to branching programs. This restriction is called a "thrifty" restriction. [47] shows an exact exponential lower bound introducing the thrifty restriction and the "read-once" restriction. But the thrifty restriction is specific for the structure of tree evaluation problems and it is not applicable to other problems. So, there is a natural question of how the lower bound changes if there is only the read-once restriction. We show a similar exponential lower bound as an answer of this question. In this technique, we use a reduction and a bottle neck argument. This reduction is hinted form the state sequence method and gives similar effect and this bottle neck argument is certificated by read-once restriction. This lower bound suggests that if there is no fundamental difference about computation between read-once branching programs and general ones (it looks so), tree evaluation problems are not in **L** and **L** is separated form **LogDCFL** also **P**.

Next, as the second result, we show a couple of stronger lower bounds of general branching programs in the chapter 3. For this lower bounds, we discuss a binary type of Tree

Evaluation Problems. This problem gives us complete d-ary tree, functions, and values like as general Tree Evaluation Problems, and our task is to check whether the value of a root node is 1 or not. We denote this problems as $BT_d^h(k)$, where h is height of complete d-ary tree. Because the original method is available only for $BT_2^3(k)$, we arrange this method to be available for $BT_d^3(k)$. This arrangement is natural and does not change key ideas of the original method. Using this method, we get a lower bound of $BT_d^3(k)$ and its value is $\Omega(n^{\frac{2d-1}{d}}/(\log n)^{\frac{2d+1}{d}})$. This lower bound is equal to [10] conjectured and can be shown by a Nečiporuk like method[36]. This lower bound is also tight.

This results says that, if we want to improve lower bounds of general branching programs, we must analyze branching programs of height 4 or more Tree Evaluation Problems. Hence we try to modify the original method to be available for $BT_2^4(k)$. In the chapter 3 we also discuss extension of height for the method. In this discussion we observe that natural extension does not work for height $BT_2^4(k)$. In the case of height 3 Tree Evaluation Problems, the method divides critical states into disjoint sets where one input set uses only critical states in one disjoint set. But in the case of height 4, there is a branching programs where some critical states can not be divided into disjoint sets for input sets. This difficulty tells us to modify the definition of critical sets may not be useful for height 4 Tree Evaluation Problems.

We also studied upper bounds of branching programs. An *OBDD*(Ordered Binary Decision Diagram) is a binary branching program whose all computation paths keep the order of input variables. And a *k-IBDD*(k-indexed Binary Decision Diagram) is a branching program with k layers and each layer consists of an OBDD. k-IBDD Satisfiability Problem (k-IBDD SAT) asks given a k-IBDD, whether there exists a consistent path from a root to sink 1 or not. [4] shows k-IBDD is an NP-Complete problem and if we use exhaustive search, k-IBDD SAT with n variables and m states can be solved in $O(m2^n)$ time. Here, one of our goal is to design a super-polynomially faster algorithm such as an $O(m2^{n-\omega(\log n)})$ time algorithm.

As the third result, we design an exponentially fast algorithm for k-IBDD SAT. If k-IBDD SAT with n variables polynomial states are given, our algorithm can solve it in $poly(n) \cdot 2^{n-n^\alpha}$ time, where $\alpha = \frac{1}{2^{k-1}}$ and poly(n) represents a polynomial of n. This algorithm uses a partial assignment and longest common sequences. Given a k-IBDD, we can find a longest common sequence on each layers. After finding longest common sequences, we fix every variables except in it. Then we can convert that partial branching program to an OBDD. Because OBDD SAT can be solved in polynomial time by checking reachability, we can check whether the converted OBDD has a consistent path or not in

polynomial time. This idea makes our algorithm super-polynomially faster than exhaustive search.

As the fourth result we also design algorithms for a kind of a sorting. One of the most basic sorting problems is the Swap-Sort problem : Given an list with n integer numbers in non-increasing order, if we are only allowed to swap two adjacent rows, how many swaps do we need to sort them in non-decreasing order? It is well-known fact that $\binom{n}{2}$ swaps are necessary and sufficient. Peter Winkler [50] introduced "Sorting Pairs in Bins" and Ito, Teruyama and Yoshida [26] extended it to a more general problem "Sorting k-Sets in Bins". Sorting k-Sets in Bins gives us n numbered bins each with k numbered balls, such that bin i is adjacent to bins $i-1$ and $i+1$ (but bin n is not adjacent to bin 1), and the balls in bin i are each numbered $n+1-i$. And our task is to swap any two balls between adjacent bins and to get every ball into the bin carrying its number. For this problem, we calculate the lower bound as $(1-\frac{k-1}{2k^2+k-1})\frac{k+1}{4}n^2+O(n)$, and show an algorithm which solve this problem in $\frac{k+1}{4}n^2+O(kn)$ swaps. This means that if k and n increase, this upper bound asymptotically approaches to the lower bound. We also propose an faster algorithm for $k=3$, this needs only $\frac{15}{16}n^2+O(n)(=0.9375n^2+O(n))$ swaps.

In this paper, we discuss super-polynomial lower bounds for read-once branching programs in chapter 2. Lower bound for general branching programs is discussed in chapter 3. After these, we discus algorithms and upper bounds for two problems. In chapter 4, we show algorithms for k-IBDD satisfiability problems. Algorithms for sorting k-set bins problems are also discussed in chapter 5. After all, we summarise all results in chapter 6.

Chapter 2

Read-Once Branching Programs for Tree Evaluation Problems

In this chapter, we discuss *tree evaluation problems (TEPs)*. This problem is introduced to separate the complexity classes **P** and **L** by Cook, McKenzie, Wehr, Braverman and Santhanam [10]. The problem is obviously in **P** and Cook and others showed a conjecture that this problem is not in **L**. They also showed branching programs lower bounds with a special restriction to tree evaluation problems. Their lower bound is $\Omega(k^h)$ for height h TEP, where k is a range of the value which nodes in TEP have. Remark that these lower bounds are tight for upper bound.

Now we try to show super polynomial lower bounds of branching programs with more moderate restriction, "read-once" restriction. The above authors introduced a restriction called *thrifty* against the structure of branching programs (i,e., against the algorithm for solving the problem) and proved that any thrifty branching program needs $\Omega(k^h)$ states, where h is the height of tree evaluation problems. The thrifty restriction roughly means that when the BP reads an internal node v (actually reads its associated function), it has already read all the values of the v's subtree. Thus this algorithmic restriction strongly restricts the order of tree nodes that are read by the branching programs. (The thrifty restriction also applies to nondeterministic branching programs, in which case its meaning is more subtle.) The authors claim that this restriction is "natural," but we can of course think of different kind of branching programs that guess (read) function values first and then check the leaf values if they actually realize the function values. In fact our lower bound proof gets messy in this case. Recall that we have another popular restricted type of branching programs, namely the *read-once* restriction, where a read-once branching program reads each input value at most once in any computation path. In fact the above $O(k^h)$ construction is

not only thrifty but also read-once and [47] proves that if our branching program is both thrifty and read-once, then this explicit construction with $(k+1)^h - k$ states is absolutely optimum. Now the natural question is what if we impose only the read-once restriction. Our new lower bound technique show the same lower bound although branching programs are not thrifty but also read-once. This technique is shown after introduction of key tools.

2.1 Introduction

Settling the **P** vs. **NP** question is obviously the biggest goal of theoretical computer science, but the fact is that almost nothing is known for separation of other complexity classes, either. For example, separation of **L** (= Log space) and **P**, which has been much less popular than **P** vs. **NP**, should be equally important to make clear the whole view of complexity classes. To this end, Cook, McKenzie, Wehr, Braverman and Santhanam introduced a simple but very general problem called the *tree evaluation problem* (*TEP*) [10]. For fixed $h, k > 0$, $FT_h(k)$ is given as a complete, rooted binary tree of height h in which each internal node is associated with a function from $[k]^2$ to $[k]$, and each leaf node with a number in $[k]$. The value of an internal node v is defined naturally, i.e., if it has a function f and the values of its two child nodes are a and b, then the value of v is $f(a, b)$. Our task is to compute the value of the root node by sequentially executing this function evaluation in a bottom-up fashion. Note that the original definition in [10] is based on a d-ary tree. In this chapter, we only consider a binary tree for our TEP.

Our computation model is branching programs (BP's) that are sometimes more useful to discuss complexity bounds rather than Turing machines (TM's) especially for problems having relatively low complexities like TEP. It is known that the size of a branching program (the number of its states) and the space of a TM are closely related, namely a lower bound $s(n)$ for BP's size implies a lower bound $\Theta(\log(s(n)))$ for TM's space. It then turns out that if we can prove that any BP solving $FT_h(k)$ needs at least $k^{r(h)}$ states for any unbounded function r, then this problem is not in **L**. Since it is obviously in **P**, we would be able to separate **L** and **P**. For details of these observations, see [10].

It is not hard to construct a branching program that computes $FT_h(k)$ of size $O(k^h)$ (see Fig. 2.3 given later) and this construction strongly seems optimal. As mentioned above, we only need a much more moderate bound, $k^{r(h)}$, and that is the natural reason why we think this problem would fit our goal. In fact, [10] proves, by using the black pebbling game [37][9], that if our BP's satisfy a certain property, called the *thrifty* restriction, then we do need $\Omega(k^h)$ states. The thrifty restriction roughly means that when the BP reads an internal node v (actually reads its associated function), it has already read all the values

of the v's subtree. Thus this algorithmic restriction strongly restricts the order of tree nodes that are read by the BP. (The thrifty restriction also applies to nondeterministic BP's, in which case its meaning is more subtle.) The authors claim that this restriction is "natural," but we can of course think of different kind of BP's that guess (read) function values first and then check the leaf values if they actually realize the function values. In fact our lower bound proof gets messy in this case.

Recall that we have another popular restricted type of BP, namely the *read-once* restriction, where a read-once BP reads each input value at most once in any computation path. In fact the above $O(k^h)$ construction is not only thrifty but also read-once and [47] proves that if our BP is both thrifty and read-once, then this explicit construction with $(k+1)^h - k$ states is absolutely optimum. Now the natural question is what if we impose only the read-once restriction.

Our contribution. It is shown that if a read-once BP B solves $FT_h(k)$, then B needs $\Omega(k^h)$ states, thus proving a lower bound on the size of read-once BP's similar to that of thrifty BP's. Actually B needs to be read-once only for states reading leaf values, i.e., the result holds for even less restricted BP's such that in every computation path, if the last leaf-reading state s reads a leaf node v, any state appearing before s on the path does not read v. Note that there is no restriction at all on states reading internal nodes (associated with functions). Furthermore, since our main lemma bounds the number of only leaf-reading states, we do not have to care about the number of these non-leaf-reading states.

Since there are no restrictions on the order of nodes visited by the BP any longer, there is no obvious way of directly using the pebbling game for lower bound proof. Instead, we use a similar notion from slightly different angle, namely we use what we call a *cut configuration*, a set of the values of $h-1$ nodes that "cut" paths between leaf nodes and the root of the given $FT_h(k)$. The key lemma is that if a last leaf-reading state accepts two or more inputs having different cut configurations, then the function part in the inputs is severely restricted, which means the number of different inputs whose paths go through this state is very small. Thus there must be a lot of inputs whose function part does not have this restriction, and we can imply that those inputs have only one cut configuration for any of the last leaf-reading states. For such a fixed function part, the number of inputs having that cut configuration for the last leaf-reading state is easily bounded from above. Thus follows the lower bound for the number of such states. Of course there should still exist a big gap between this class of BP's and general ones, but at least we can get rid

of the issue of node orders visited by BP's, which was quite annoying for the attempt of generalising our lower bound proofs.

Related Work. Other than the lower bounds for thrifty BP's, [10] includes several important results, for instance, it gives a lower bound, k^3, for *unrestricted* BP's solving $FT_3(k)$, which is tight up to the constant factor. This is still the best lower bound for general BP's solving TEP. [31] studies mainly nondeterministic BP's for TEP. Its main result is that "bitwise-independent" thrifty nondeterministic BP's for TEP have at least $\frac{1}{2}k^{h/2}$ states, which is tight against the upper bounds shown in [10]. Their main technique is so-called the entropy method developed in [30]. See [10] for several other attempts trying to separate relatively low complexity classes. For instance [20] studies the complexity of BP's solving GEN (known to be P-complete) that asks a certain kind of reachability to a target element repeatedly using a binary operation.

Studies on branching programs have been quite popular since their introduction by Masek [34], and there is a large amount of literature even restricted to studies on their size lower bounds (the following is only a small fraction): The best general deterministic lower bound is still $\Omega(n^2/(\log n)^2)$, which was proved almost half a century ago by Nečiporuk [36]. Note that the above lower bound for $FT_3(k)$ is $\Omega(n^{3/2}/(\log n)^{5/2})$ in terms of the binary input length. (For a general d-ary TEP, [10] obtains a stronger $\Omega(n^2/(\log(n))^2)$ lower bound applying the Nečiporuk method.) On the other hand, about read-once branching programs, we have much better lower bounds. In 1984, Žák [51] first obtained a super-polynomial lower bound, $\Omega(2^{\sqrt{n}-\log n})$, for the half-clique function, which was improved to more than $2^{n/3-o(n)}$ by Wegener [46]. For the triangle parity function, Ajtai [1] gave a 2^{cn} lower bound and the value of c was later improved by Simon(1993) [44]. Jukna [29] relaxed the read-once restriction to the k-read-once restriction (i.e., all variables except k ones are read-once). He obtained a lower bound of $2^{\Omega((\frac{n}{k})^{1/2})}$ for $k = O(n/\log n)$ and this is extended by Žak [41] into a hierarchy theorem based on this value k.

2.2 Preliminaries

For the Tree Evaluation Problem (TEP), $FT_h(k)$, we are given a complete binary tree T^h of height h with nodes 1 through $2^h - 1$ (see Fig. 2.1 for $h = 3$). Each internal node $1 \leq i \leq 2^{h-1} - 1$ is associated with some explicit function $f_i : [k]^2 \mapsto [k]$, where $[k] = \{1, 2, \ldots, k\}$. Each leaf node j ($2^{h-1} \leq j \leq 2^h - 1$) is associated with a number in $[k]$. Our task is to compute the value of the function f_1 at the root node in the natural way: Suppose that we have inputs $f_1, f_2, f_3, a_4, a_5, a_6, a_7$ for the tree of Fig. 2.1. Then the

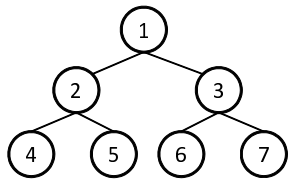

Figure 2.1: $FT_3(3)$

$f_1(x,y)$

$x \backslash y$	1	2	3
1	3	1	2
2	2	2	1
3	3	1	3

$f_2(x,y)$

$x \backslash y$	1	2	3
1	2	1	2
2	3	2	1
3	3	2	1

$f_3(x,y)$

$x \backslash y$	1	2	3
1	2	1	3
2	2	2	3
3	1	1	1

Figure 2.2: Example of f_1, f_2, f_3

value we want to obtain is

$$f_1(f_2(a_4, a_5), f_3(a_6, a_7)).$$

Note that each f_i is given as an explicit sequence of values, e.g., $f_i(1,1)$, $f_i(1,2)$, $f_i(1,3)$, $f_i(2,1)$, $f_i(2,2)$, $f_i(2,3)$, $f_i(3,1)$, $f_i(3,2)$, $f_i(3,3)$ for $k = 3$. In some cases, it is convenient to use a $k \times k$ matrix instead of the above sequence. For instance Fig. 2.2 shows an example of f_1,f_2,f_3 for $h = 3$. Now if (a_4, a_5, a_6, a_7)=$(3, 3, 1, 2)$, then the solution for this inputs is $f_1(f_2(3,3), f_3(1,2))$= $f_1(3,2)$= 1. In our lower bound proof, the nodes located at height $h-1$ (parents of leaves) play an important role. We call them *second-leaves*.

Our computation model is a (deterministic) *branching program* (*BP*) B, which is a directed, rooted, acyclic graph. Its vertices are called *states* including a unique *initial state* and k *sink states*. Each non-sink state (or simply a state if no confusion would arise) has k outgoing edges labelled by 1 through k, while each sink state has no outgoing edges. Each state has a *label* of the form (i_1, i_2, i_3) or j, where $1 \leq i_1 \leq 2^{h-1} - 1$, $1 \leq i_2, i_3 \leq k$ and $2^{h-1} \leq j \leq 2^h - 1$. Each sink state has a label l where $1 \leq l \leq k$. A BP B computes the solution of TEP in the following way. Suppose that our input is $I = (f_1(1,1), f_1(1,2), \ldots, f_{2^{h-1}-1}(k,k),\ a_{2^{h-1}}, \ldots, a_{2^h-1})$. Then its *computation path*, P, for input I is defined as follows. P starts from the initial state. If P is now at a state with label (i_1, i_2, i_3), then P is extended by the edge labelled by $f_{i_i}(i_2, i_3)$. If P is at a state with label j, then it is extended by the edge labelled by a_j. We often say that B "reads" the input attached to the node i_1 (a non-leaf node) or j (a leaf) and branches due to its value between 1 and k. P ends with some sink state; if its label is l, then the outcome of the computation is l. If this outcome is equal to the correct solution for all possible inputs I, then we say B *solves* $FT_h(k)$.

Fig. 2.3 shows an example of a BP that solves $FT_3(3)$. The computation path for the input previously given (f_1, f_2, f_3 in Fig. 2.2, and (a_4, a_5, a_6, a_7) =$(3, 3, 1, 2)$) is given by a thick line. A BP is called *read-once* if all paths from the root to sinks do not have two or more same labels. The BP in Fig. 2.3 is read-once.

Our lower bound proof is based on the following simple idea: Suppose that A ($|A| = m_1$)

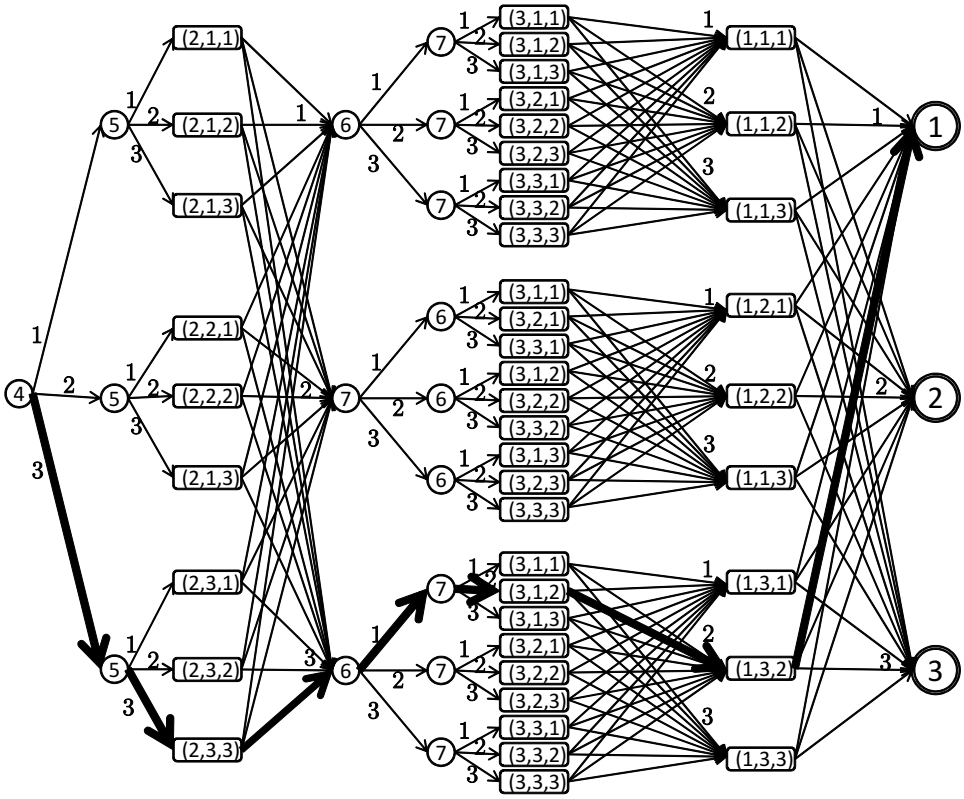

Figure 2.3: An example of a read-once branching program solving $FT_3(3)$

is a carefully selected subset of all the possible inputs for $FT_h(k)$. Let B be any (read-once) BP that solves $FT_h(k)$. Then our proof says that we can always select a set S of states such that each computation path corresponding to each input in A goes through some state in S and any state in S accepts computation paths of at most m_2 inputs in A, concluding that $|S|$ is at least m_1/m_2. To introduce such an input set A, we consider the following constraint for functions f_i: Suppose that

$$X = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1k} \\ \alpha_{21} & \ddots & \cdots & \alpha_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{k1} & \alpha_{12} & \cdots & \alpha_{kk} \end{bmatrix}$$

is the matrix representation of f_i. Then it has to satisfy the following three constraints: (i) $\alpha_{11} \dots \alpha_{1k}$ (= the first row) is a permutation of $(1, \dots, k)$ (ii) $\alpha_{11} \dots \alpha_{k1}$ (= the first column) is a permutation of $(1, \dots, k)$ (iii) For $\forall j \geq 2$, $\alpha_{j1} \dots \alpha_{jk}$ is a permutation that

can be written as $\delta^l(\alpha_{11} \dots \alpha_{1k})$ for some $1 \le l \le k$ where δ is the cyclic permutation

$$\delta = \begin{pmatrix} 1 & 2 & \cdots & k-1 & k \\ 2 & 3 & \cdots & k & 1 \end{pmatrix}$$

and δ^l is a composition of l δ's. Thus each row is a permutation, and it is not hard to see that each column is also a permutation. X is fixed by determining its first row and the first column, and hence there are $k!(k-1)!$ different f_i's. Let F be the class of functions satisfying these constraints. In this paper, we assume that our function f_i is always selected from F unless otherwise stated (but of course, our BP's must give correct solutions for all inputs). Now here are easy but important lemmas.

Lemma 1 *Suppose that two inputs I and I' (their function parts satisfy the constraint) are exactly the same except only one leaf value at node j. Then the final value of $FT_h(k)$ is different between I and I'.*

Proof. Suppose that the final value is the same and consider the path from the root to j. Since the root value is the same and the leaf value is different, there must be a node i on the path such that the value of i is the same but the value of i's next node i' on the path is different, say, a in I and a' in I'. Let i'' be the sibling of i' (both i' and i'' have i as their parent). Then the value of i'' is the same, say b, in both I and I'. Thus we have $f_i(a, b) = f_i(a', b)$ for $a \neq a'$, which contradicts that $f_i \in F$. □

Lemma 2 *Suppose that a BP B solves $FT_h(k)$. Then (1) for any internal node i of $FT_h(k)$ and for any $a, b \in [k]$, there must be a state whose label is (i, a, b) in B. (2) If P is a legal computation path, then for any leaf node j, P includes a state that reads j.*

Proof. For (2), suppose that P corresponds to input I and it does not read j. Then consider another input I' which is different from I only in j. Then B obviously outputs the same value for I and I', contradicting the previous lemma. (1) is proved similarly by considering two inputs I and I' that differ only in $f_i(a, b)$ and such that both inputs actually use $f_i(a, b)$ (meaning the values of i's two children are a and b under I and I'). Note that if I satisfies the restriction, then I' does not. Now one can see, exactly as in the proof of the previous lemma, that the final value is different between I and I', but B outputs the same value, a contradiction. □

The next lemma (hinted by Th. 5.8 and Th. 5.9 of [10]) relates the number of states reading leaf nodes and the number of state reading second-leaf nodes. By this lemma, we

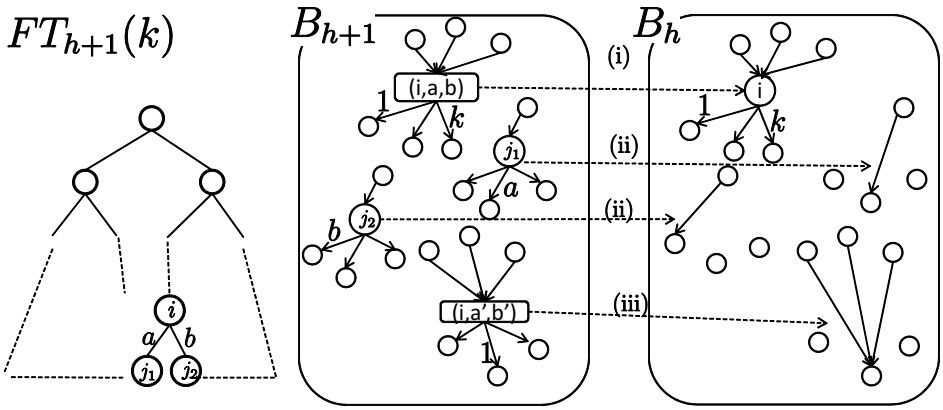

Figure 2.4: Modification rules(i), (ii) and (iii).

can increase the degree of k by one in the lower bound given in the next section. Note that this lemma holds for general BP's (and see Subsec. 4 for its by-product).

Lemma 3 *For $h \geq 1$, if there is a BP B_{h+1} solving $FT_{h+1}(k)$ such that the number of states that read second-leaf nodes is n, then there is a BP B_h solving $FT_h(k)$ such that the number of states that read leaf nodes is at most n/k^2. Furthermore, if B_{h+1} is read-once, so is B_h, also.*

Proof. we construct B_h from the given B_{h+1} as follows. Let i be a second-leaf node of $FT_{h+1}(k)$ and (a, b) is a pair of inputs to f_i such that the number of states in B_{h+1} that read $f_i(a, b)$ is less than or equal to the number of states reading $f_i(a', b')$ for any (a', b'). Let m be the number of such state s reading $f_i(a, b)$. By Lemma 2, there is at least one state that reads $f_i(a, b)$ for any $(a, b) \in [k] \times [k]$. So, m is at most $(1/k^2)\times$(the number of states that read f_i). Now we make the following modification against B_{h+1} (see Fig. 2.4). The basic idea is that we fix the values of the two child (leaf) nodes of i to a and b. Then i looks like a leaf node of $FT_h(k)$ and among the states in B_{h+1} that read i, only $1/k^2$ ones survive by the following construction. This holds for any i and hence the lemma holds. (i) Change the label of the above m states from (i, a, b) to i. (Namely this state reads a leaf node of $FT_h(k)$.) (ii) Suppose that j_1 and j_2 are the two leaf nodes whose parent is i. Then we remove all the states q of B_{h+1} that read j_1 (j_2, respectively) by connecting q's incoming edges to the state to which the edge from q labelled by a (b, respectively) goes. (iii) We remove all the state q of B_{h+1} that read $f_i(a', b')$, $((a', b') \neq (a, b))$, by connecting q's incoming edges to the state to which the edge from q labelled by 1 goes (this "1" is not important or it may be any number in $[k]$).

We repeat this change for all second-leaf nodes of $FT_{h+1}(k)$, obtaining B_h. We omit

the proof that this construction is correct, since it is almost obvious from the construction. □

2.3 Lower Bounds

In this section we obtain a lower bound for the number of states that read leaf nodes of $FT_h(k)$. Then combining it with Lemma 3, we obtain a better lower bound for the number of states that read second-leaf nodes of $FT_h(k)$. Recall that our input satisfies the constraint (its functions belong to F) and all BPs in this section are read-once. Let B be a BP that solves $FT_h(k)$ and P be its arbitrary computation path. (To avoid confusion, we sometimes say that P is a *legal* computation path to emphasise that P is based on an input whose function part satisfies the constraint.) Then by Lemma 2, P reads all leaf values (for any leaf j, there is a state in P that reads j). Let q be the last state on P that reads a leaf value, i.e., there is no state after q on P that reads a leaf. Since B is read-once, q is also the last leaf-reading state on any other legal computation path that includes q. Thus, as far as we are looking at only legal computation paths, we can define a *last leaf-reading state* without specifying a computation path.

Now we define our key tool in the proof in this section. Suppose that $I = (f_1, \ldots, f_{2^{h-1}-1}, a_{2^{h-1}}, \ldots, a_{2^h-1})$ is currently associated with $FT_h(k)$ and let j be a leaf. Then the *cut configuration* (CC) for I with respect j, denoted as $CC(I, j)$, is defined as follows.

$$CC(I, j) = (a_1, a_2, \ldots, a_{h-1})$$

where, (i) a_1 is the value of j's sibling and (ii) if a_i, $1 \leq i \leq h-2$, is the value of node i, a_{i+1} is the value of the sibling of i's parent (see Fig. 2.5). Suppose that we know functions f_1 to $f_{2^{h-1}-1}$. Then if we further know these $n-1$ values as well as the value of j, then we can compute the solution (= the value of node 1). In fact, it is well-known that we can compute the solution in such a way that we need at most $(h-1)\lceil \log k \rceil$ memory space at any stage of its computation (by recursively obtaining the values of a_1, a_2, and so on, in this order first, then the associated function values from bottom to top). What will be done in the rest of this section is to count the number of legal inputs with a certain restriction on its CC that go through a last leaf-reading node. Our first lemma is an upper bound on the number of inputs having a single CC.

Lemma 4 *Fix functions $f_1, \ldots, f_{2^{h-1}-1}$, an arbitrary leaf node, j, and an arbitrary $(a_1, \ldots, a_{h-1})$, $a_i \in [k]$. Then the number of leaf values whose CC with respect to j is $(a_1, \ldots, a_{h-1})$ is at*

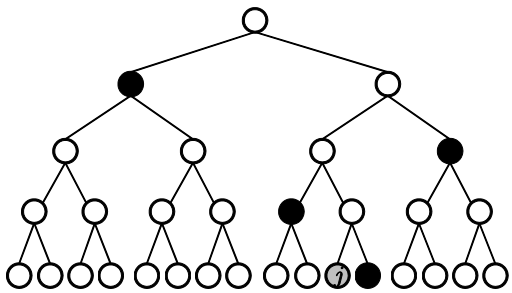

Figure 2.5: Involved nodes in $CC(I, j)$.

most $k^{2^{h-1}-h+1}$

Proof. Let T_v be a subtree of $FT_h(k)$ whose root is a node v at height i of $FT_h(k)$. Let $v_1, \ldots, v_{2^{i-1}}$ (we used a simplified numbering) be the leaf nodes of T_v, and $g(v_1, \ldots, v_{2^{i-1}})$ be the value of v. We first calculate the number of different leaf values $(b_1, \ldots, b_{2^{i-1}})$ such that $g(b_1, \ldots, b_{2^{i-1}}) = a$ for a fixed $a \in [k]$. For fixed $b_1, \ldots, b_{2^{i-1}-1}$, $g(b_1, \ldots, b_{2^{i-1}-1}, x)$ is a function from $[k]$ to $[k]$ (denoted by $g'(x)$). By lemma 1, $g'(x_1) \neq g'(x_2)$ if $x_1 \neq x_2$, in other words, g' is a bijection. Hence, for any $a \in [k]$, value $b \in [k]$ such that $g'(b) = a$ is fixed. Since this holds for any $b_1, \ldots, b_{2^{i-1}-1}$, the number of leaf values $b_1, \ldots, b_{2^{i-1}}$ such that $g(b_1, \ldots, b_{2^{i-1}}) = a$ is $k^{2^{i-1}-1}$.

Now we calculate the number N of leaf values such that their CC with respect to leaf j is $(a_1, \ldots, a_{h-1})$. Let the node taking value a_i be v_i. See Fig. 2.5 again. First, note that node j can take any of the k values. Next, the value of node v_1 is fixed to a_1; it takes only one value. Since node v_2 is a top node of a subtree with height 2, its leaf nodes can take $k^{2^{2-1}-1} = k$ different values by the above fact. Similarly, v_3's leaf nodes can take $k^{2^{3-1}-1} = k^3$ different values and so on. Therefore,

$$\begin{aligned} N &= k \cdot 1 \cdot k \cdot k^3 \cdot \cdots \cdot k^{2^{h-2}-1} \\ &= k \cdot k^{2^1+2^2+\cdots+2^{h-2}-(h-2)} \\ &= k \cdot k^{2^{h-1}-2-(h-2)} = k^{2^{h-1}-(h-1)} \end{aligned}$$

Then this lemma is proved □

Now we are ready to prove our main lemma. We divide an input $(f_1, \ldots, f_{2^{h-1}-1}, a_1, \ldots, a_{2^{h-1}})$ into two parts, the function part (f-part) $\mathbf{f} = (f_1, \ldots, f_{2^{h-1}-1})$ and the leaf value part (l-part) $\mathbf{l} = (a_1, \ldots, a_{2^{h-1}})$. Let B be any (read-once) BP solving $FT_h(k)$ and s be any last leaf-reading state, reading a leaf j. Let c_1 and c_2 be two different CC's with

respect to j whose inputs have the same f-part, and $G(c_1, c_2, s)$ be the set of such f-parts (i.e., if $\mathbf{f} \in G(c_1, c_2, s)$, then there are two l-parts $\mathbf{a}_1$ and $\mathbf{a}_2$ such that $(\mathbf{f}, \mathbf{a}_1)$ and $(\mathbf{f}, \mathbf{a}_2)$ have CC's c_1 and c_2, respectively). Note that there are $(k!(k-1)!)^{2^{h-1}-1}$ different f-parts in total and we denote this number by N_0.

Lemma 5 *Suppose that k is a prime number. Then for any c_1, c_2, s, $|G(c_1, c_2, s)| \leq N_0 \frac{k}{k!}$*

Proof. Let $c_1 = (a_1, \ldots, a_{h-1})$, $c_2 = (b_1, \ldots, b_{h-1})$, and let $v_1, \ldots, v_{h-1}$ be the nodes providing these two CC values. Also let $g_1, \ldots . g_{h-1}$ be the functions associated with $v_1, \ldots, v_{h-1}$ producing both c_1 and c_2 (for different leaf values). Recall that s is a last leaf-reading state (reading node j) and our BP is read-once. Hence, the value of j is first read at s, which means two computation paths realizing c_1 and c_2 are not affected by the value of j until the state s. Furthermore, these paths must go to the same sink-node for each fixed value a of the node j, because after the state s our BP reads only function values being the same for c_1 and c_2.

$$g_1(a_1, g_2(a_2, \ldots, g_{h-1}(a_{h-1}, a) \ldots)) = g_1(b_1, g_2(b_2, \ldots, g_{h-1}(b_{h-1}, a) \ldots)) \tag{2.1}$$

Note that for a fixed a_{h-1}, $g_{h-1}(a_{h-1}, a)$ is a bijection form $[k]$ to $[k]$ and can be represented as a permutation

$$\delta_{h-1} = \begin{pmatrix} 1 & 2 & \cdots & k \\ \alpha_1 & \alpha_2 & \cdots & \alpha_k \end{pmatrix}$$

where $\alpha_1 \ldots \alpha_k$ are the (a_{h-1})th row of the matrix of g_{h-1}. Using similar representations for g_1 to g_{h-2}, (2.1) can be written as

$$\delta_1 \delta_2 \ldots \delta_{h-1} = \delta'_1 \delta'_2 \ldots \delta'_{h-1} \tag{2.2}$$

where δ_i is the (a_i)th row of (the matrix of) g_i and δ'_i is the (b_i)th row of g_i. Due to our constraint for g_i, we can write $\delta'_i = \delta^{l_i} \delta_i$ for some $0 \leq l_i \leq k-1$ (recall that δ is the cyclic permutation).

Now (2.2) can be rewritten as

$$\delta_1 \delta_2 \ldots \delta_{h-1} = \delta^{l_1} \delta_1 \delta^{l_2} \delta_2 \ldots \delta^{l_{h-1}} \delta_{h-1}$$

Suppose that a_i and b_i are the first different values in the two CC's (i,e., $a_1 = b_1, \ldots a_{i-1} =$

b_{i-1}). Then $l_1 = l_2 = \cdots = l_{i-1} = 0$ and hence

$$\begin{aligned}
& \delta_1\delta_2\ldots\delta_{h-2}\delta_{h-1} = \delta^{l_1}\delta_1\delta^{l_2}\delta_2\ldots\delta_{h-2}\delta^{l_{h-1}}\delta_{h-1} \\
\Leftrightarrow \quad & \delta_i\delta_{i+1}\ldots\delta_{h-2} = \delta^{l_i}\delta_i\delta^{l_{i+1}}\delta_{i+1}\ldots\delta^{l_{h-1}} \\
\Leftrightarrow \quad & \delta_i = \delta^{l_i}\delta_i\delta^{l_{i+1}}\delta_{i+1}\ldots\delta^{l_{h-1}}\delta_{h-2}^{-1}\ldots\delta_{i+1}^{-1} \\
\Leftrightarrow \quad & \delta^* = \delta_i^{-1}\delta^c\delta_i \qquad (2.3)
\end{aligned}$$

where $\delta^* = (\delta^{l_{i+1}}\ldots\delta_{i+1}^{-1})^{-1}$ and $\delta^c = \delta^{l_i}$.

Now let

$$\delta_i = \begin{pmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_k \\ 1 & 2 & \cdots & k \end{pmatrix} \text{ and } \delta^* = \begin{pmatrix} 1 & 2 & \cdots & k \\ \beta_1 & \beta_2 & \cdots & \beta_k \end{pmatrix},$$

Then (2.3) can be written as

$$\begin{aligned}
& \begin{pmatrix} 1 & 2 & \cdots & k \\ \beta_1 & \beta_2 & \cdots & \beta_k \end{pmatrix} \\
&= \begin{pmatrix} 1 & 2 & \cdots & k \\ \alpha_1 & \alpha_2 & \cdots & \alpha_k \end{pmatrix} \begin{pmatrix} 1 & 2 & \cdots & k \\ 1+c & 2+c & \cdots & k+c \end{pmatrix} \begin{pmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_k \\ 1 & 2 & \cdots & k \end{pmatrix} \\
&= \begin{pmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_k \\ \alpha_{1+c} & \alpha_{2+c} & \cdots & \alpha_{k+c} \end{pmatrix}
\end{aligned}$$

where $1+c, \ldots, k+c$ are all MOD k. Note that δ^* and δ^c are conjugate and therefore their cycle structures are the same. Since δ is the cyclic permutation, δ^c has a single cycle and therefore δ^* also has a single cycle.

It then turns out that if we fix α_1 to $d \in [k]$, then by the left hand side, d should be mapped to β_d, meaning $\alpha_{1+c} = \beta_d$. Then again by the left hand side, β_d should be mapped to β_{β_d}, meaning $\alpha_{1+2c} = \beta_{\beta_d}$, and so on. Namely once α_1 is fixed, all the other α_i's are sequentially fixed one after another or δ_i itself is fixed. Since k is prime, this sequence of value transfer does not end in the middle. Thus we have at most k different possibilities for δ_i (due to k different values for α_1). Recall that δ_i can take $k!$ different permutation in general. (The whole matrix is determined by fixing the first row and the first column, but it should be noted that it is also determined by fixing any row and then any column). But now there are only k possibilities as shown above. So the number of different g_i is at most $N_0\frac{k}{k!}$ for each combination of other $h-2$ functions. Note that s may have another CC, say c_3, other than c_1 and c_2. Then we have another restriction for functions, which

results in even a smaller number of possible functions. Thus it is enough to consider only the case that the state s has two different CC's, for the upper bound of the lemma. □

Now we imply a contradiction if the number of leaf-reading states is less than k^{h-1}, through the following two lemmas.

Lemma 6 *Suppose that $s_1, s_2, \ldots, s_{k^{h-1}-1}$ are $k^{h-1}-1$ different last leaf-reading states of the BP B. Then there is an f-part $\mathbf{f}_0$ such that for any s_i $(1 \leq i \leq k^{h-1}-1)$, if inputs $(\mathbf{f}_0, \mathbf{a}_1)$ and $(\mathbf{f}_0, \mathbf{a}_2)$ go through s_i, their CC's are the same.*

Proof. We count the number of f-parts $\mathbf{f}$ that do not satisfy the condition of the lemma. By Lemma 5, there are $N_0 \cdot \frac{k}{k!}$ such $\mathbf{f}$ for each combination of state s_i, CC c_1 and CC c_2. Note that we have $k^{h-1}-1$ s_i's and there are at most k^{h-1} different CC's in general. Therefore the number of such $\mathbf{f}$'s is at most

$$N_0 \cdot \frac{k}{k!} \cdot (k^{h-1}-1) \cdot (k^{h-1})^2 \leq N_0 k^{3h-2}/k!,$$

which is strictly less than N_0 for a large (prime) k. Thus an $\mathbf{f}_0$ of the lemma must exists. □

Lemma 7 *B needs at least k^{h-1} last leaf-reading states.*

Proof. Suppose B has at most $k^{h-1}-1$ last leaf-reading states. Then by Lemma 6, there is an f-part $\mathbf{f}_0$ such that inputs having this $\mathbf{f}_0$ as their f-part show at most one CC for any of these last leaf-reading states. However, Lemma 4 shows each state accepts at most $k^{2^{h-1}-h+1}$ l-parts, meaning these $k^{h-1}-1$ states accept at most $k^{2^{h-1}-h+1} \cdot (k^{h-1}-1) < k^{2^{h-1}}$ l-values in total. Since each of the all $k^{2^{h-1}}$ l-values must be accepted by some last leaf-reading state, this is a contradiction. □

Theorem 1 *Any read-once BP B_h solving $FT_h(k)$ needs at least k^h states.*

Proof. The contraposition of Lemma 4 claims that if the number of leaf-reading states of B_h is at least m, then the number of second-leaf-reading states of B_{h+1} is at least $k^2 m$. Now the theorem is immediate from Lemma 7. □

2.4 General Branching Programs for Height-3 TEP

Recall that Lemma 3 holds for general BPs. Also it turns out that the TEP of height two is somewhat special. Thus we can obtain the following general lower bound for BPs for the height-3 TEP with a simpler proof than that of [10].

Theorem 2 *Any (general) BP solving $FT_3(k)$ needs at least k^3 states.*

Proof. Due to Lemma 3, it suffices to show that any BP solving $FT_2(k)$ needs at least k leaf-reading states. In the following, we show it needs at least $k+1$ leaf-reading states, which is optimal by a construction similar to that of Fig. 3. Recall that $FT_2(k)$ has three nodes, 1, 2 and 3, where node 1 is associated with a function f_1 and nodes 2 and 3 are leaf nodes. Suppose that we have a BP B that solves $FT_2(k)$ and that has at most k leaf-reading states. We fix f_1 to an arbitrary function in F and then B can be modified to the BP that reads only leaf nodes; we also denote this BP by B.

We give a new label (in addition to the original label of B), a set of pairs (a,b), $1 \leq a, b \leq k$, to each state and each edge of B by the following rule: (i) The initial node of B has label $\{(a,b) \mid 1 \leq a, b \leq k\}$, i.e., the set of all possible pairs. (ii) Suppose that a state s has a label S and an edge e from s reads node 2 to get value i. Then the label to the edge e is $\{(i,b) \mid 1 \leq b \leq k\} \cap S$, namely the (possibly empty) set of pairs in S whose first element is i. Similarly for the case that s reads node 3 (we do the same thing with the second element of the pair). (iii) Suppose that all the edges entering state s already have labels. Then the label of s is the union of the labels of those incoming edges. Now it is easy to see that such labels "describe" an execution of B in the following sense: Suppose that the label of an edge e includes a pair (a,b). Then the computation path of B goes through this edge e if and only if the values of nodes 2 and 3 are a and b, respectively (and f_1 is the current fixed function).

Now suppose for contradiction that B has at most k states other than k sink states and we look at edges that go to these sink states. Note that the number of all edges is k^2 since each of the k states has k edges. Also note that B has to read both of the two leaf states in its computation path, so at least one edge goes to non-sink states. Consequently the number of the above (going to sink states) edges is at most $k^2 - 1$. Since the total number of pairs is k^2, it is impossible to map all those pairs to the edges in a one-to-one fashion, or one of the following two cases must happen:

(1) Some pair (a,b) does not appear in any label of these edges. B obviously does not do a correct computation when the values of nodes 2 and 3 are a and b, respectively.

(2) Some edge has two (or more) pairs, say (a, b) and (a', b'). Notice that if the state this edge outgoes from reads node 2, then we have $a = a'$. Then the computation of B is not correct again since the output would be the same if the values of node 2 is the same and the values of node 3 are different (recall that our f_1 is in F). Similarly for the case that the state reads node 3 (then $b = b'$).

Thus we can conclude that such B is not a correct BP. □

2.5 Concluding Remarks

The obvious future work is to remove the read-once restriction. Since our main lemma (Lemma 5) heavily depends on the read-once restriction, we do not have any specific approaches to this ultimate goal at this moment. There are a couple of more reasonable sub-goals: One is to prove that if a BP B is thrifty, then B can be converted to a read-once BP without increasing the number of leaf-reading states drastically, or equivalently, to prove that reading a same leaf node twice or more do not help much in thrifty BP's. Another possibility is to attack the case for $h = 4$. This seems more tractable since we can restrict ourselves to the number of leaf-reading states of BP's for $FT_3(k)$ that have several specific properties as shown in [10]. Also this lower bound will outperform the longstanding one by Nečiporuk [36].

Chapter 3

Lower Bounds of General Branching Programs for TEP

In this chapter, we discuss *tree evaluation problems (TEP)*. Cook, McKenzie, Wehr, Braverman and Santhanam also showed branching programs lower bounds for "small" tree evaluation problems, and have a conjecture that "big" tree evaluation would have tight lower bounds with upper bounds.

3.1 Introduction

Next our target is to analyse greater lower bounds of general branching programs. It is known that if any branching program solving TEP must need super polynomial size in input length, then TEP is not in $\mathbf{L}$. They also showed $\Omega(k^3/\log k)$ lower bound for branching programs solving TEP with height 3, 2-ary(binary) tree.

First, to improve their lower bounds, we analyse branching programs solving TEP with height three, d-ary tree, where d is any constant. Because above lower bound technique is only for TEP having height three binary complete tree, we modify this technique to be applicable for height three d-ary complete tree. Using this modification, we show a tight lower bound $\Omega(k^{2d-1}/\log k)$. This lower bound supports the conjecture that upper bounds of branching programs solving any TEP is tight for lower bounds and it leads to "$\mathbf{L} \neq \mathbf{P}$". This result also suggests that if this technique become to apply to height four binary tree evaluation problems, we can also get lower bounds of branching programs solving height four d-ary TEP.

Next, we try to modify for height four tree evaluation problems. Because above modification is correct and useful, we do not change the key idea of the original technique. This

modification seems good, but has many difficulties. In this chapter, although we could not complete modification, we discuss what we think and analyse. These two modifications are discussed in each subsection.

3.2 Extension of Width and Height of TEP

[10] introduced "state sequence method" and shown a lower bound for $BT_2^3(k)$, where $BT_d^h(k)$ is the binary type of tree evaluation problems having rooted, d-ary, complete tree of height h. Our task is to detect whether the root value is 1 or not. This method shows any general branching program must have $\Omega(n^{1.5}/(\log n)^{2.5})$ states. Although this lower bound is not stronger than Nečiporuk's lower bound $\Omega(n^2/(\log n)^2)$, it is tight for an upper bound and supports the conjecture that upper bounds of branching programs for any TEP is tight for lower bounds. This upper bound is shown by constructing a branching program based on a depth first search algorithm. Let n is the length of input and d is some constant, upper bounds of branching programs solving $BT_d^h(k)$ (denoted as $BP(BT_d^h(k))$ is below.

$$BP(BT_d^h(k)) = O(k^{(h-1)(d-1)+1}) = O(n^h) \tag{3.1}$$

Using $r(h)$ as any unbounded function of h, if lower bounds grow up to $\Omega(n^{r(h)})$, TEP are not in class **L**. So one of natural motivation is to generalize this method and apply to larger TEP such as $BT_3^4(k)$ or $BT_2^5(k)$: These upper bound is $O(k^{7/3}/(\log k)^{10/3})$ and $O(k^{5/2}/(\log k)^{7/2})$ respectively. Tight lower bounds for $BT_3^4(k)$ or $BT_2^5(k)$ beat Nečiporuk's lower bound.

One of our observation is modifying the state sequence method. In this subsection, we introduce this method and discuss some modifications. Because the original method is only for $BT_2^3(k)$, now we focus on $BT_2^3(k)$. The outline of the original method is as below.

Set a branching program solving $BT_2^3(k)$ We consider the lower bound of BPs. Therefore, the BP analysed with this method is assumed be optimal.

Restrict input for BP We divide a set of states into k^2 sets of states disjointly. One set of inputs shows computation paths using states in one set. Any other set of inputs does not shows computation paths using states in the same set.

Get computation paths Once a BP gets an input, a BP shows the computation path, that is consisted by states (not label).

Define critical states We find some states which must be used.

See "learning interval" We check the flow of a BP and distinct-ability between this function and input from critical states.

Set a transition function We can see a different transition function if a input has different f_1.

Compare with f_1 The number of various of transition functions is represented by the number of critical states.

Now we discuss the detail of the state sequence method. Note that this method shows a lower bound as $BP(BT_2^3(k)) = \Omega(k^3/\log k)$

Once a branching program gets an input, and computes an output, there exists a computation path, which starts on the root state and ends to the sink state. The computation path is a sequence of states whoes input goes through. Because a branching program is acyclic, one state exists at most once in a computation path.

For thinking a lower bound, we define "critical states" and count the amount of these. There are some labels which is used in some measure in any branching programs. This means that it is suffice to show lower bounds that we count the number of states reading v_2 or v_3. To check how states are critical, check all computation paths under some restrictions for inputs. These restrictions make us see the value of v_2 and v_3 and other nodes have a fixed value. Precisely,

- $v_4 = v_6 = r, v_5 = v_7 = s$
- $f_2(s,t) = v_2^I, f_3(s,t) = v_3^I$
- f_2, f_3's other entries are fixed 0.
- f_1's all entries are 0 or 1 then, $|f_1| = 2^{k^2}$

From this restriction, all inputs are denoted as (v_2^I, v_3^I, f_1).

Now, we can get the computation path $C(a,b,f)$ from input (a,b,f). To count the amount of states, we denote Γ_j as the amount of states labeled as (j,r,s) for $j = 2,3$. Because one state has exact one label, Γ_j is disjoint over all pairs (r,s). If the pair (r,s) are set to make $\Gamma_2 + \Gamma_3$ minimum, it suffice to show

$$\Gamma_2 + \Gamma_3 \geq k/\log k \tag{3.2}$$

To count an amount of states labeled v_2 or v_3, we define "learning interval" This learning interval is shown by "critical state" defined as below. In each computation path,

- The first state is critical.
- If we see first state or a critical state labeled $(2, r, s)$,
  the next critical state is the first one labeled $(3, r, s)$ or $1, \ldots, k$
- If we see a critical state labeled $(3, r, s)$,
  the next critical state is the first one labeled $(2, r, s)$ or $1, \ldots, k$

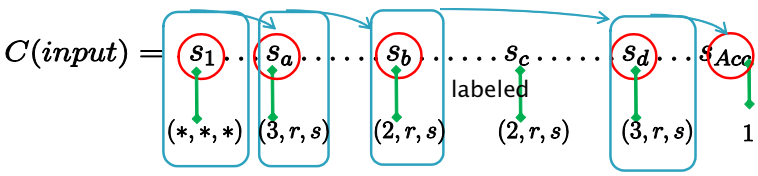

Figure 3.1: An example of a computation path and critical states

Then "learning interval" is defined as a pair of critical states. In above case,(s_1, s_a),(s_a, s_b),(s_b, s_d) and (s_d, s_{Acc}) are learning intervals.

Learning intervals show the amount of states. To count those, we set more assumption on branching programs. Unique root state is denoted as s_1, accepting state as s_{Acc}, and rejecting state as s_{Rej}. Under this assumption, if function f_1 is fixed, then one input shows a computation path. This computation path reach a critical state and read some value, should go to next critical state. This transition is represent as function using critical states. This transition function under fixed function $f_1 = f$ is described as below.

$$\psi_2[f] : [k] \times (\Gamma_2^{r,s} \cup \{s_1\}) \to (\Gamma_3^{r,s} \cup \{s_{Acc}, s_{Rej}\}) \tag{3.3}$$

$$\psi_3[f] : [k] \times \Gamma_3^{r,s} \to (\Gamma_2^{r,s} \cup \{s_{Acc}, s_{Rej}\}) \tag{3.4}$$

This function shows transitions from critical states to critical states. According to input restriction, if a computation path reaches to the state labeled $(2, r, s)$, this computation path goes to the state labeled $(3, r, s)$ or sink states depends on v_2^I. In the case of a computation path reaches to the state labeled $(3, r, s)$, this computation path goes to the state labeled $(2, r, s)$ or sink states depends on v_3^I. This function's range is expressed by amount of states labeled as $(2, r, s)$ or $(3, r, s)$. The various of $\psi_2[f]$ and $\psi_3[f]$ is

$$|(\psi_2[f], \psi_3[f])| = (\Gamma_3 + 2)^{k(\Gamma_2+1)}(\Gamma_2 + 2)^{k\Gamma_3} \leq (\Gamma_2 + \Gamma_3 + 2)^{k(\Gamma_2+\Gamma_3+1)} \tag{3.5}$$

And, here is a fact that this transition function should be different if f_1 has a different function. Here, f_1's outputs are distinct 1 or not 1.

Claim 1 *this function is distinct for distinct f.*

Proof : If not, there are functions f, g such that $\psi_j[f] = \psi_j[g]$ but $f(a,b) \neq g(a,b)$. From $C(a,b,f), C(a,b,g)$, both are accepted or rejected. This leads contradiction. □

From this claim and $|f| = 2^{k^2}$,

$$2^{k^2} \leq (\Gamma_2 + \Gamma_3 + 2)^{k(\Gamma_2+\Gamma_3+1)} \tag{3.6}$$

$$k^2 \leq k(\Gamma_2 + \Gamma_3 + 1)\log(\Gamma_2 + \Gamma_3 + 2)$$

$$(\Gamma_2 + \Gamma_3 + 1) \geq k/\log(\Gamma_2 + \Gamma_3 + 2) \tag{3.7}$$

This leads to $\Gamma_2 + \Gamma_3 \geq k/logk$ and $BP(BT_2^3(k) = \Omega(k^3/\log k))$

3.2.1 Width Extension

Now we get a lower bound of $BT_2^3(k)$ as previous subsection. Thus, next target is to verify a lower bound for $BT_d^h(k)$ with $d > 2$ or $h > 3$ for a greater lower bounds. In case of $h > 3$, we find it difficult that to use state sequence method with any arrangement without losing its main ideas. But we find it can be true that to use state sequence method with some arrangement in case of $d > 2$. Fortunately, this arrangement is useful in case that d any integer constant ($d \geq 2$). So we discuss a lower bound of $BT_d^3(k)$ instead of any other TEP.

With arranging state sequence method, we show stronger lower bound than previous subsection.

Theorem 3 (Lower bounds of $BT_d^3(k)$)

$$BP(BT_d^3(k)) = \Omega(k^{2d-1}/\log k) = \Omega(n^{(2d-1)/d}/(\log n)^{(3d-1)/d}) \tag{3.8}$$

Original state sequence method is defined for only $BT_2^3(k)$. That is because the definition of key ideas in state sequence method depend on $BT_2^3(k)$. So we need to arrange state sequence method for calculating a lower bound of $BT_d^3(k)$.

Let us number each node of $BT_d^3(k)$ as suggested by the heap structure. Thus the root node is numbered 1, and the leftmost height 2 node is number 2, the rightmost height 2 node is number $d+1$. The leftmost leaf node is number $d+2$ and the rightmost leaf node is d^2+d+1. An internal node i has function $f_i : [k]^d \mapsto [k], (1 \leq i \leq d+1)$. A leaf node j has a value from 1 to $k, (d+1 \leq i \leq d^2+d+1)$. So an input I can be represented as $(x_{d+2}, \ldots, x_{d^2+d+1}, x_{(d+1,1,\ldots,1)}, \ldots,$

r45mm

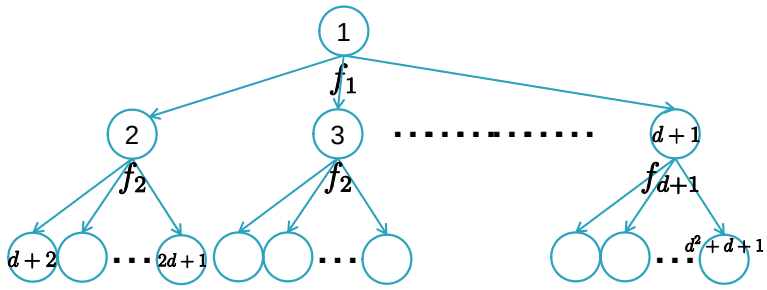

Figure 3.2: Numbering for $BT_d^3(k)$

$x_{(d+1,k,\ldots,k)}, \ldots, x_{(2,k,\ldots,k)}, x_{1[1,\ldots,1]}, \ldots, x_{1[k,\ldots,k]})$. This means that from x_{d+2} to x_{d^2+d+1} denotes the value of node from $d+2$ to d^2+d+1. And from $x_{(d+1,1,\ldots,1)}$ to $x_{(1,k,\ldots,k)}$ denotes a function f_i for a internal node n_i. $x_{(i,k,\ldots,k)}$ whose the value of node 1 when all node i's children has value 1.

Original state sequence method counts amount of state labeled by variables $(2, r, s)$ or $(3, r, s)$ in the case of $BT_2^3(k)$. So we arrange state sequence method for case of $BT_d^3(k)$. More specifically we count amount of state labeled by variables which is related in height 2 nodes, and we define critical state similarly.

In the case of $BT_2^3(k)$, all critical states are divided into disjoint sets. This division is realized because the label (i, r, s) is different if the pair (r, s) is different. We can see similar situations in case of $BT_d^3(k)$. The difference is the number of children which height 2 node has.

Now we start modifying state sequence method to apply for $BT_d^3(k)$. First modification occurs in an input restriction step. Each input of $BT_d^3(k)$ is obviously different from input of $BT_2^3(k)$. So we restrict input carefully not to change the key idea of state sequence method. In this sight, we can divide states similarly which labeled by height 2 node variable in case $BT_2^3(k)$. This leads that divided states keep disjointness like as $BT_2^3(k)$'s input restricted.

Precisely, the state labeled by $(j, l_1, \ldots, l_d)$ and the state labeled by $(j, l'_1, \ldots, l'_d)$ is different if $(l_1, \ldots, l_d)$ is not equivalent to $(l'_1, \ldots, l'_d)$. Then arranged restriction is

- $(x_{d+1+j}, \ldots, x_{2d+j}) = l_j$ $(l_i \in [k], 1 \leq j \leq k)$
- $f_i(l_1, \ldots, l_d) = x_{(i,l_1,\ldots,l_d)} = v_i^I$ $(2 \leq i \leq d+1)$
- f_i's other entries are fixed 1. $(2 \leq i \leq d+1)$
- f_1's all entries are 0 ore 1 then, $|f_1| = 2^{k^d}$

Thanks for this restriction, we can denote an input I as $(v_2^I, \ldots, v_{(d+1)}^I, f_1)$. And we denote Γ_i^{rest} as the set of state labeled by $x_{(i,l_1,\ldots,l_d)}$ as above restriction. As stated earlier, Γ_i^{rest} is disjoint for $\Gamma_i^{\text{rest'}}$ which is a set of state labeled by $x_{i[l'_1,\ldots,l'_d]}$ in case of $(l'_1, \ldots, l'_d) \neq (l_1, \ldots, l_d)$.

Second modification is definition of critical states. To see the states on which branches some computation path in branching programs, we defined critical states in case of $BT_2^3(k)$. But $BT_d^3(k)$'s input data is so different from $BT_2^3(k)$' one. How do we see branching programs solving $BT_d^3(k)$? This answer is shown in the arranged restriction for input.

In case of BT_d^3 with new restriction, we have variables as only $v_2^I, \ldots, v_{(d+1)}^I, f_1$. The purpose of definition of critical states is to find state transition functions, and compare the variety of these functions and the range of input f_1. So we see states labeled by $(i, l_1, \ldots, l_d)(i = 2, \ldots, (d+1))$ where the computation path on branching programs can branch. And we check these states are critical or not. Each computation path starts with the root state, visits some critical states, and end with a sink node. If an input shows the value of the root node as 1, computation path end with s_{Acc}. If an input does not show the value of the root node as 1, computation path end with s_{Rej}. From these discussion, we can arrange the definition for critical states for $BT_d^3(k)$ as below.

Definition 1 (critical state for $BT_d^3(k)$)

- *The first state(root state) is critical.*
- *If we see the first state or a critical state labeled $(2, l_1, \ldots, l_d)$, the next critical state is the first one labeled $(i, l_1, \ldots, l_d)(i = 3, \ldots, (d+1))$ or s_{Acc},s_{Rej}*
- *If we see a critical state labeled $(i, l_1, \ldots, l_d), (i = 3, \ldots, (d+1))$, the next critical state is the first one labeled $(j, l_1, \ldots, l_d), (j \neq i)(j = 2, \ldots, (d+1))$ or s_{Acc},s_{Rej}*

Third modification is for state transition functions but it is tiny. State transition functions in state sequence method is consisted by Learning Intervals. This Learning Intervals look as pair of critical states. Because we arrange how critical state is, we have to arrange how state transition functions is constructed. And this needs a large number of word to description but not complicated as below.

Definition 2 (State transition function for $BT_d^3(k)$)
Each state transition function φ_2, φ_3 or φ_{d+1} is $f \in [k]^d \mapsto \{0, 1\}$. φ_2 shows the next critical state which is in Γ_i^{rest} $(i = 3, \ldots, d+1)$ or which is sink state from a critical state in Γ_2^{rest} or the root node. $\varphi_i(i = 3, \ldots, d+1)$ shows the next critical state in Γ_j^{rest} $(j =$

$2, \ldots, d+1$ *and* $(i = 3, \ldots, d+1)$ *or sink states from a critical state in* Γ_i^{rest}.

$$\varphi_2[f] : [k] \times (\Gamma_2^{rest} \cup \{s_1\}) \mapsto (\Gamma_j^{r,s} \cup \{s_{Acc}, s_{Rej}\})(j = 3, \ldots, d+1)$$
$$\varphi_j[f] : [k] \times \Gamma_j^{rest} \mapsto (\Gamma_i^{r,s} \cup \{s_{Acc}, s_{Rej}\})(j = 3, \ldots, d+1), (i = 2, \ldots, d+1), (i \neq j)$$

Any other definition in state sequence method does not depend on $d = 2$. For example, we can get computation paths from restricted input, we can define learning intervals from critical states, and we can get inequality similarly from state transition functions with no dependant. Like in case of $BT_2^3(k)$, we can count up an amount of critical state from comparing these functions.

Then state sequence method has been arranged for $BT_d^3(k)$. Applying arranged state sequence method to $BT_d^3(k)$, we can prove Theorem 3.

Proof. Now we have a branching programs solving $BT_d^3(k)$. According to modifications, we can denote an input as $(v_2^I, \ldots, v_{(d+1)}^I, f_1)$ uniquely. A computation path should branch at the states labeled by $(2, l_1, \ldots, l_d)$, $(3, l_1, \ldots, l_d)$,..., $(d+1, l_1, \ldots, l_d)$ or $(1, *, \ldots, *)$ in branching programs. Lower bounds are calculated by counting up the amount of these states.

Similarly the case of $BT_2^3(k)$, we can divide some state into disjoint sets. From one restriction case, we can denote Γ_i^{rest} as a set of states labeled by $(i, l_1, \ldots, l_d)$. Γ_i^{rest} is disjoint from $\Gamma_i^{\text{rest'}}$ the set of states used in case of another restriction case. This is clearly true because each state have only one label. Then, we can say belows with denoting Γ_i as the set of states labeled by $(i, l_1, \ldots, l_d)$ in the whole of branching programs.

$$|\Gamma_i| \geq k^d \times |\Gamma_i^{\text{rest}}| \tag{3.9}$$

$$BP(BT_d^3(k)) \geq \sum_{i=2}^{d+1} |\Gamma_i| \tag{3.10}$$

And it suffice to show below.

$$\sum_{i=2}^{d+1} |\Gamma_i^{\text{rest}}| = \Omega(k^{d-1}/\log k) \tag{3.11}$$

Now we count up $\sum_{i=2}^{d+1} |\Gamma_i^{\text{rest}}|$. From definition of critical states, we can set learning intervals similarly in the case of $BT_2^3(k)$.

To count critical states, we can set state transition functions form learning intervals as definition. And we can say a similar claim.

Claim 2

A group of state transition function $|(\varphi_2[f], \ldots, \varphi_{d+1}[f])|$ *is distinct for distinct* $f_1 = f$.

Proof. Use contradiction. If not, there are f, g such that $f(l_1, \ldots, l_d) \neq g(l_1, \ldots, l_d)$ but $\varphi_j[f] = \varphi_j[g] (j = 2, \ldots, d+1)$. In this case, we can check $C(l_1, \ldots, l_d, f)$ and $C(l_1, \ldots, l_d, g)$ with $\varphi_j[f] = \varphi_j[g] (j = 2, \ldots, d+1)$. This should lead to that both are accepted or rejected. But, it should be true that $f(l_1, \ldots, l_d)$ outputs a value different from $g(l_1, \ldots, l_d)$. This means $C(l_1, \ldots, l_d, f)$ should reach a sink node labeled by different sink node reached by $C(l_1, \ldots, l_d, g)$. This is contradiction.

This claim leads to equations for comparison. Note that the root function f's range is 2^{k^d}.

$$|(\varphi_2[f], \ldots, \varphi_{d+1}[f])| \geq |f| \tag{3.12}$$

$$(\sum_{i=2}^{d+1} |\Gamma_i^{\text{rest}}| + 2)^{k(\sum_{i=2}^{d+1} |\Gamma_i^{\text{rest}}|+1)} \geq 2^{k^d} \tag{3.13}$$

$$k(\sum_{i=2}^{d+1} |\Gamma_i^{\text{rest}}| + 1) \log (\sum_{i=2}^{d+1} |\Gamma_i^{\text{rest}}| + 2) \geq k^d$$

$$(\sum_{i=2}^{d+1} |\Gamma_i^{\text{rest}}| + 1) \geq k^{d-1} / \log (\sum_{i=2}^{d+1} |\Gamma_i^{\text{rest}}| + 2) \tag{3.14}$$

$$(\sum_{i=2}^{d+1} |\Gamma_i^{\text{rest}}| + 1) = \Omega(k^{d-1} / \log k) \tag{3.15}$$

Equation 3.13 follows using 3.12, 3.11 and $|f| = 2^{k^d}$. And we take logarithm then 3.13 leads to 3.14. At last, we use Ω representation on 3.15. 3.9, 3.10 and 3.15 leads to $\sum_{i=2}^{d+1} |\Gamma_i^{\text{rest}}| = \Omega(k^{2d-1}/logk)$ and $BP(BT_d^3(k)) = \Omega(k^{2d-1}/\log k)$

3.2.2 Height Extension

Now, we try to extent the state sequence method from $BT_2^3(k)$ to $BT_2^4(k)$. Known upper bound is $BP(BT_2^4(k)) = O(k^4/\log k) = O(n^2/(\log n)^3)$. Although if we show tight lower bounds for $BT_2^4(k)$, it is not larger than Nečiporuk's lower bound. But this extension would be the first step to $BT_2^5(k)$ or $BT_3^4(k)$. If we show tight lower bounds for $BT_2^5(k)$ or $BT_3^4(k)$, it beats Nečiporuk's lower bound.

As above, a state sequence method counts the amount of states labeled the variable of height two nodes. In upper bound case, an amount of these states dominate the amount of states in branching programs. So natural idea is to restrict input to divide these states as below.

r45mm

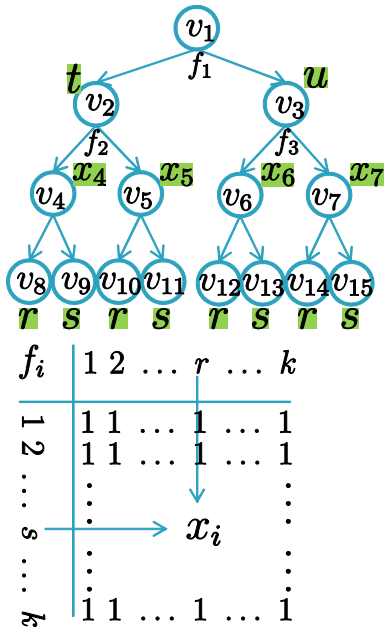

Figure 3.3: Restriction for BT_2^4

- Leaf node : $x_8 = x_{10} = x_{12} = x_{14} = r, x_9 = x_{11} = x_{13} = x_{15} = s$
- Height 2 node : All entries are fixed 1 except $f_i(r, s) = x_i$, $i = 4, 5, 6, 7$, x_i is some constant $\in [k]$.
- Height 3 node : All entries are fixed 1 except $f_2(x_4, x_5) = t, f_3(x_6, x_7) = u$
- f_1's all entries are 0 or 1.(No restriction) Then, $|f_1| = 2^{k^d}$

Next,we get the computation path. Because of optimality of branching programs, computation paths have no $(1, t, u)$) except before sink node. Input restriction shows that the computation path in a branching program should branch at $(i, r, s), (j, t, u), i = 4, 5, 6, 7$ and $j = 2, 3$. Using this restriction, critical states can be defined. Critical states are defined as the state which must branch some inputs. In original method, we set critical state labeled $(2, r, s)$ or $(3, r, s)$ because they can be divided in disjoint sets. A branching program solving $BT_2^4(k)$ shows that states labeled (i, r, s) $(i = 4, 5, 6, 7)$ can make disjoint set about a pair (r, s). Furthermore, it is conjectured that some of them after a state labeled $(2, t, u)$ or $(3, t, u)$ are used in disjointing case. From these fact, we denote $\Gamma_4^{r,s,u}$ as a set of "states labeled $(4, r, s)$ after the state labeled $(3, t, u)$". In this definition, $\Gamma_i^{r,s,*}$ would be a disjoint

set for triplet (r, s, t)or(r, s, u). From these disjoint sets, it suffice to show

$$\Gamma_4^{r,s,u} + \Gamma_5^{r,s,u} + \Gamma_6^{r,s,t} + \Gamma_7^{r,s,t} \geq k^4/\log k \tag{3.16}$$

Critical states tell us which is to be learning intervals on computation path. Paying attention to defining critical states, we can see that elements of $\Gamma_4^{r,s,u}$ or $\Gamma_5^{r,s,t}$ comes after the last $(3, r, s)$, which is not dummy. And elements of $\Gamma_6^{r,s,u}$ or $\Gamma_7^{r,s,t}$ comes after the last $(2, r, s)$, which is not dummy too. So we should check this two relation of critical states. This leads to below definition.

- The root state is critical.
- On each computation path, the last state labeled $(2, x_4, x_5)$ is critical.
- If computation path reach to a critical state labeled $(2, x_4, x_5)$,
  the next critical state is the first one labeled $(6, r, s)$ or $(7, r, s)$ or $1, \ldots, k$.
- If computation path reach to a critical state labeled $(6, r, s)$
  the next critical state is the first one labeled $(7, r, s)$ or $1, \ldots, k$.
- If computation path reach to a critical state labeled $(7, r, s)$
  the next critical state is the first one labeled $(6, r, s)$ or $1, \ldots, k$.
- On each computation sequence, the last state labeled $(3, x_6, x_7)$ is critical.
- If computation path reach to a critical state labeled $(3, x_6, x_7)$,
  the next critical state is the first one labeled $(4, r, s)$ or $(5, r, s)$ or $1, \ldots, k$.
- If computation path reach to a critical state labeled $(4, r, s)$
  the next critical state is the first one labeled $(5, r, s)$ or $1, \ldots, k$.
- If computation path reach to a critical state labeled $(5, r, s)$
  the next critical state is the first one labeled $(4, r, s)$ or $1, \ldots, k$

Then, the transition functions, which shows the chain of critical state, are described as

$$\psi_2[f] : \Gamma_2^{r,s} \mapsto \Gamma_6^{r,s,t} \cup \Gamma_7^{r,s,t} \cup \{s_{Acc}\} \cup \{s_{Rej}\} \tag{3.17}$$

$$\psi_3[f] : \Gamma_3^{r,s} \mapsto \Gamma_4^{r,s,u} \cup \Gamma_5^{r,s,u} \cup \{s_{Acc}\} \cup \{s_{Rej}\} \tag{3.18}$$

$$\psi_4[f] : [k] \times \Gamma_4^{r,s,u} \mapsto \Gamma_5^{r,s,u} \cup \{s_{Acc}\} \cup \{s_{Rej}\} \tag{3.19}$$

$$\psi_5[f] : [k] \times \Gamma_5^{r,s,u} \mapsto \Gamma_4^{r,s,u} \cup \{s_{Acc}\} \cup \{s_{Rej}\} \tag{3.20}$$

$$\psi_6[f] : [k] \times \Gamma_6^{r,s,u} \mapsto \Gamma_7^{r,s,u} \cup \{s_{Acc}\} \cup \{s_{Rej}\} \tag{3.21}$$

$$\psi_7[f] : [k] \times \Gamma_7^{r,s,u} \mapsto \Gamma_6^{r,s,u} \cup \{s_{Acc}\} \cup \{s_{Rej}\} \tag{3.22}$$

And using these transition functions, we would get target lower bound as below inequality.

$$|\psi_4[f]||\psi_5[f]||\psi_6[f]||\psi_7[f]| \geq |f_1| \tag{3.23}$$

$$LHS \leq (|\Gamma_4^{r,s,u}| + |\Gamma_5^{r,s,u}| + |\Gamma_6^{r,s,t}| + |\Gamma_7^{r,s,t}| + 2)^{k(|\Gamma_4^{r,s,u}|+|\Gamma_5^{r,s,u}|+|\Gamma_6^{r,s,t}|+|\Gamma_7^{r,s,t}|+1)} \tag{3.24}$$

$$RHS = 2^{k^2} \tag{3.25}$$

$$(|\Gamma_4^{r,s,u}| + |\Gamma_5^{r,s,u}| + |\Gamma_6^{r,s,t}| + |\Gamma_7^{r,s,t}| + 2)^{k(|\Gamma_4^{r,s,u}|+|\Gamma_5^{r,s,u}|+|\Gamma_6^{r,s,t}|+|\Gamma_7^{r,s,t}|+1)} \geq 2^{k^2} \tag{3.26}$$

$$k(|\Gamma_4^{r,s,u}| + |\Gamma_5^{r,s,u}| + |\Gamma_6^{r,s,t}| + |\Gamma_7^{r,s,t}| + 1) \geq k^2/\log(|\Gamma_4^{r,s,u}| + |\Gamma_5^{r,s,u}| + |\Gamma_6^{r,s,t}| + |\Gamma_7^{r,s,t}| + 2) \tag{3.27}$$

$$|\Gamma_4^{r,s,u}| + |\Gamma_5^{r,s,u}| + |\Gamma_6^{r,s,t}| + |\Gamma_7^{r,s,t}| + 1 \geq k/log(|\Gamma_4^{r,s,u}| + |\Gamma_5^{r,s,u}| + |\Gamma_6^{r,s,t}| + |\Gamma_7^{r,s,t}| + 2) \tag{3.28}$$

$$|\Gamma_4^{r,s,u}| + |\Gamma_5^{r,s,u}| + |\Gamma_6^{r,s,t}| + |\Gamma_7^{r,s,t}| = \Omega(k/\log k) \tag{3.29}$$

This modification seems to work well. But there are some difficulties in this idea. First, this idea depends on how the states are dummy or not. In the trivial construction or some heuristic construction, there are no dummy states. Even branching programs are optimal, there is no way to find some states are dummy or not about some input sets. Second, this idea divides states reading nodes whose lie in the second level of TEPs. But some branching programs can have states which can be used in different input sets about t or u. From these reasons, a natural modification for a state sequence method is not effective for

height four TEP.

3.2.3 Chapter Summary

In this chapter, we discuss state sequence method introduced by [10]. This method shows lower bounds of general branching programs solving $BT_2^3(k)$. We propose modifications for this method and show greater lower bound with applying modified method to $BT_d^3(k)$, where d is a constant. This lower bound is $n^{\frac{2d-1}{d}}/(\log n)^{\frac{3d-1}{d}}$, therefore Nečiporuk's lower bound is still the best.

We also try to modify state sequence method to be available for height four tree evaluation problems. For this modification, there are some difficulties. Because this method compare the amount of variety of a function and the amount of states in a disjoint set, we find a lower bound only $k/\log k$ as the amount of states in one disjoint set. Therefore, if we show $k^4/\log k$ lower bounds, we should divide critical states in k^3 disjoint sets. Unfortunately, our input restriction could not realize this situation and we could not get tight lower bounds. To conquer these difficulties, we find the way of dividing critical states or the way to define critical set. For another possibility, a totally new method can be established for tight lower bounds of branching programs for height four tree evaluation problems.

Chapter 4

Efficient Algorithms for k-IBDD Satisfiability

4.1 Introduction

In this chapter, we propose a polynomial space and deterministic algorithm for solving the satisfiability of k-indexed Binary Decision Diagram (abbr. k-IBDD), which is one of a variant of the satisfiability problems (abbr. SAT). Our algorithm runs in time poly(n) $\cdot 2^{n-n^{1/2^{k-1}}}$, which is super-polynomially faster than an exhaustive search. SAT is one of central problems in theoretical computer science. There exists many variants of SAT and many research have been done. In many cases, they are known to be NP-complete. Therefore, it is a natural task to design some algorithm running faster than an exhaustive search, which check all possible satisfying assignments. One of central research for SAT is about CNF SAT. CNF SAT is to ask whether given a conjunctive normal form, there exists an assignment satisfying it. An exhaustive search algorithm solve CNF SAT with $n$variables and m clauses in time $O(m \cdot 2^n)$. In CNF SAT, there exist many excellent algorithms such as [2, 7, 12, 13, 23, 38, 42]. The current best algorithm for CNF SAT with m clause runs in time $O\left(2^{(1-\frac{1}{\log(m/n)})n}\right)$ shown in [7]. This implies that if the number of clause is bounded by cn (c is an arbitrary positive constant), we can solve CNF SAT in time $O\left(2^{(1-\mu(c))n}\right)$, where $\mu(c)$ is some constant depending on c.

Recently, Circuit SAT is also extensively studied. Circuit SAT, is given a Boolean Circuit C with n variables, to check whether there exists an assignment to the input variable such that C outputs 1. It includes CNF SAT as a special case. $\mathbf{AC}^0$ are constant depth circuits consisting of AND, OR gates of unbound fan-in and unary NOT gates. $\mathbf{AC}^0$ SAT with cn size and depth d can be solved in time $O\left(2^{(1-1/O(\log c+d\log d)^{d-1})n}\right)$

[25]. $\mathbf{ACC}^0$ are constant depth circuits consisting of AND, OR, Modulo gates of unbound fan-in and unary NOT gates. $\mathbf{ACC}^0$ SAT with cn size and depth d can be solved in time $O\left(2^{n-\Omega(n^{2^{-O(d)}})}\right)$ [49]. U_2 formula are formulas with binary AND, OR and unary NOT gates. U_2 formula SAT with cn size can be solved in time $O\left(2^{(1-1/c^{O(1)})}\right)$ [40]. B_2 formula are formulas with all binary boolean gates. B_2 formula SAT are with cn size can be solved in time $O\left(2^{(1-1/c^{c^3})}\right)$ [43].

However, there are a very few researches on the satisfiability of branching program. One important result is about k-OBDD SAT. OBDD are branching programs such that all paths from a root to a shank have the same order of the variables. k-OBDD are one of extensions from OBDD such that they consist of k layers and each layer is OBDD with the same order of the variables. k-OBDD SAT asks whether given ak-OBDD there exists an consistent path from a root to a 1-sink. For any constant k, this problem can be solvable in polynomial time shown in [4]. Another result is about the satisfiability of general branching programs. [8] design an deterministic algorithm running in $O(2^{n-\omega(\log n)})$ time, which solves any instance with n variables and $m = n^{2-\epsilon}$ states, where ϵ is an arbitrary small positive constant. However, this algorithm requires an exponential space and cannot applies for an instance of k-IBDD SAT with $\omega(n^2)$ states. In addition, this algorithm uses an exponential space.

In this chapter, the satisfiability of k-indexed Binary Decision Diagram (abbr. k-IBDD). k-IBDD is one of extensions of k-OBDD, which is the same as k-OBDD except that each layer has different order of the variables. k-IBDD SAT is shown to be NP-complete for any $k \geq 2$ in [4]. Our goals is to design an algorithm super-polynomially faster than an exhaustive search such as an $O(m2^{n-\omega(\log n)})$ time algorithm. We get the following theorem,

Theorem 4 *For any instance of k-IBDD SAT with n variables and $m = O(n^c)$ states (c is an arbitrary positive constant), there exists a deterministic polynomial-space algorithm which solves it in poly$(n) \cdot 2^{n-n^\alpha}$ time, where $\alpha = \frac{1}{2^{k-1}}$ and poly(n) represents a polynomial of n.*

4.1.1 Related Work

Chen et al. gave a deterministic exponential space algorithm solves satisfiability problems of general branching programs with n variables and $m = O(n^{2-\epsilon})$ states in $O(2^{n-n^\delta})$ time [8]; where ϵ is an arbitrary small positive constant and δ is a constant such as $0 < \delta < 1$ depending on ϵ. Bollig et al. designed a polynomial time algorithm for k-OBDD SAT

which is a special case of k-IBDD SAT [4]. For k-IBDD SAT, Jain et al. proposed an experimental efficient algorithm [28].

4.2 Preliminaries

Let $X = \{x_1, \ldots, x_n\}$ be a set of variables and Boolean true be 1, false be 0. $\overline{x}$ is the negation of variable $x \in X$. A *nondeterministic branching program* is underlying on rooted directed multigraph $B = (G, \phi_V, \phi_E)$. Each node in G is called *state*. $\phi_V : V \to X \cup \{\mathbf{0}, \mathbf{1}\}$ is a function of label for states, and $\phi_E : E \to \{0, 1\}$ is a function of label for edges. The root state is denoted as r and there exists exact two sink states denoted as t_0, t_1. Let $\phi(t_0) = 0$ and $\phi(t_1) = 1$. t_0 and t_1 are called 0-sink and 1-sink, respectively. For all states $v \in V\{t_0, t_1\}$, $\psi_V(v) \in X$. Each edge in G is given a label 0 or 1 by ϕ_E. A edge (u, v) denotes a directed edge from a state labeled u to a state labeled v,and u is called as *initial vertex*, v is called as *terminal vertex*. Figure 4.1 show an example of a nondeterministic branching program. Each circle represents a state and a symbol in a circle represents a label of the state. An arrowed line represents an directed edge and a 0/1 along the edge represents a label of the edge. For an input $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$, a branching program B trace a *computation path* as sequence of states from a root state to a sink state. If computation path reach a state labeled x_i, it is extended to a next state through an outgoing edge with label a_i. As computation path can reach a sink t_1, B outputs 1, otherwise 0. Let $f : \{0, 1\}^n \to \{0, 1\}$ be a boolean function. For any assignment $a \in \{0, 1\}^n$, if $f(a)$ equals to outputs of B, we call B represents f. The size of B, denoted by $|B|$, is defined as the number of edges in G. If any states except t_0 and t_1 in B has one 0-edge and one 1-edge, B is a *deterministic* branching program. In this chapter, a branching program is a deterministic one unless otherwise noted.

A permutation $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$ is an arbitrary order from 1 to n. For $i \in \{1, \ldots, n\}$, let $\pi^{-1}(i)$ be j with $\pi(j) = i$. Introduced permutations, an *ordered binary decision diagrams (OBDD)* and a *k-indexed binary decision diagrams (k-IBDD)* are defined as below.

Definition 3 *An* **OBDD** *is a branching program with a fixed π. It holds that $\pi^{-1}(i) < \pi^{-1}(j)$ if there exists an edge from a state labeled x_i to a state labeled x_j.*

Definition 4 *A* **k-IBDD** *is a branching program as follows. It can be separated to k layers and i-th layer is a OBDD with a permutation π_i. An arbitrary edge from i layer reaches to j layer or a sink state where $j < i$. If all π_i are the same permutation, it is called as a* **k-OBDD**.

Figure 4.1 represents a nondeterministic OBDD with $\pi = (1, 2, 3)$. Figure 4.2 represents a (deterministic) 2-IBDD with $\pi_1 = (1, 2, 3)$ and $\pi_2 = (2, 3, 1)$.

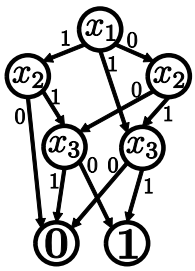

Figure 4.1: a nondeterministic OBDD

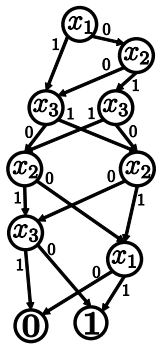

Figure 4.2: 2-IBDD

k-IBDD SAT is a problem which asks whether given a k-IBDD B with n variables and m states, there exists an input $a \in \{0, 1\}^n$ such that B outputs 1 If $k = 1$, a 1-IBDD (equal to an OBDD) can be solved in $O(m)$ time by detecting reachability from a root state to 1-sink. It is known to be NP-complete when $k \geq 2$ [4].

In our algorithm, permutations are sometimes modified. Some the key kinds of permutations or sequences are need to note. Recall $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$. A reverse permutation of π is defined as $\pi^R = (\pi^R(1), \pi^R(2), \ldots, \pi^R(n)) = (\pi(n), \pi(n-1), \ldots, \pi(1))$. A subsequence with length m of π is $\pi' = (\pi'(1), \pi'(2), \ldots, \pi'(m)) = (\pi(i_1), \pi(i_2), \ldots, \pi(i_m))$, where $1 \leq i_1 < i_2 < \cdots < i_m \leq n$. A longest increasing subsequence σ_{inc} is a subsequence with length m of π with maximum m such that it satisfies $\pi'(i) < \pi'(j)$ for all $1 \leq i < j \leq m$. A longest decreasing subsequence σ_{dec} is a subsequence with length m of π with maximum m such that it satisfies $\pi'(i) > \pi'(j)$ for all $1 \leq i < j \leq m$.

About a longest increasing or decreasing subsequence, the following theorem of is well known.

Theorem 5 (The Erdős-Szekeres theorem [17]) *A sequence of real number with length n contains a longest increasing subsequence with length $m \geq \sqrt{n}$ or a longest decreasing subsequence with length $m \geq \sqrt{n}$.*

Proof. Use induction. Let $f(n)$ be the minimum length of a sequence which contains a increasing or decreasing subsequence with length n. For the base case, $f(1) = 1$. Next, assuming that we get a increasing or decreasing subsequence with length n from a length $f(n)$, let us show

$$f(n+1) = f(n) + 2n - 1 \tag{4.1}$$

An equation 4.1 and the base case lead to $f(n) = n^2 + 1$. So, it suffice to show an equation 4.1 to prove this theorem.

Think about a sequence with length $f(n) + 2n - 1$. Let A_1 be the set of numbers of left most $f(n)$ numbers and n_i is the number on the $(f(n) + i)$th left. From assumption, we can obtain a increasing or decreasing subsequence with length n from A_1 Let S_1 be the set of numbers which is a member of this increasing or decreasing subsequence in A_1. After finding a subsequence, let r_1 be a right most number in S_1. Next, Let A_2 be $A_1 \cup \{n_1\} \backslash \{r_1\}$, and S_2 be the set of numbers which is a member of this increasing or decreasing subsequence in A_2. Continue these procedures, we can get $2n$ subsequences form a whole sequence. We think about 3 cases. (i) There exists $n+1$ increasing subsequences. If there exist $r_j \leq r_k (j \leq k)$, $S_j \cup \{r_k\}$ must be a increasing subsequence with length $n+1$. If not, the set of all $r_j (S_j$ is increasing subsequence) should be a decreasing subsequence with length $n+1$. (ii) There exists $n+1$ decreasing subsequence. there must be a decreasing or increasing subsequence from similar arguments. (iii) There exists n increasing subsequences and n decreasing subsequences. In this case, each r_i in increasing subsequences represents a decreasing subsequence, and each r_j in decreasing subsequences represents a increasing subsequence. Let the smallest one of r_i be r_i^* and the largest one of r_j be r_j^* If $r_i^* > r_j^*$, we can take a $n+1$ decreasing or increasing subsequence. If not, without lose of generality, let r_i^* lies on right side of r_j^*. then, r_i^* can join to some decreasing subsequence as the $n+1$th number. In the case of let r_i^* lies on left side of r_j^* should be discuss similarly. Then, in any case, we can get $n+1$ increasing or decreasing subsequence from $f(n)+2n-1$ sequence.□

Note that these subsequence can be found in $O(n^2)$ time.

A partial assignment for $x = (x_1, \ldots, x_n)$ is $a = (a_1, \ldots, a_n) \in \{0, 1, *\}^n$. This means that x_i is assigned to 0 or 1 if a_i is 0 or 1, respectively. For any partial assignment $a \in \{0, 1, *\}^n$, a support of a is defined as $S(a) := \{x_i \mid \alpha_i \neq *\}$. $B|_a$ is a partial branching program B followed by a partial assignment a, and this is constructed as follows.

(1) Remove all edge labeled $\overline{a_i}$ which is an outgoing edge from a state labeled with $x_i \in S(a)$.

(2) For any state v with indegree is 0 except for a root state, remove a state labeled v and all outgoing edges from it.

(3) For any state v labeled $x_i \in S(a)$ except for a root state, let U be $\{u \mid (u, v) \in E\}$. Note that there exists an edge (v, w) labeled a_i. For all $u \in U$, we add an edge (u, w) labeled the same label of (u, v) and remove an edge (u, v). Remove an edge (v, w).

(4) If a label of a root state r is in $S(a)$, remove an edge (r, v), then make a state v a new root state.

Figure 4.3 is a partial branching program of Figure 4.2 followed by a partial assignment $a = (1, *, *)$.

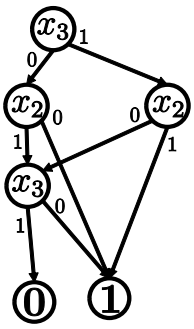

Figure 4.3: a partial branching program

4.3 Algorithms of Transformation for OBDDs

In this section, we introduce two key modules for Theorem 4. One constructs a new OBDD representing conjunction of two OBDDs. The other constructs a new nondeterministic OBDD with reversed permutation from an OBDD.

Lemma 8 *Let B_1 be a nondeterministic OBDD with π which represents a Boolean function f_1. Let B_2 be a nondeterministic OBDD with π which represents a Boolean function f_2. Then there exists an algorithm that constructs an OBDD B with π which represents $f_1 \wedge f_2$ from B_1 and B_2 and takes $O(|B|)$ time,where $|B| \leq 2|B_1| \cdot |B_2|$.*

Proof. We show that Algorithm 1 produce an OBDD satisfying this lemma. This algorithm is extended from Brayant's algorithm [6]. OBDD B constructed by this algorithm simulates both OBDD B_1 and B_2 at the same time. Let r_1 and r_2 be root states of B_1 and B_2, respectively. B starts a computation by moving a token from a state (r_1, r_2). If the token is on a state (v_1, v_2) in B, it means that the token is on a state v_1 in B_1 and a state v_2 in B_2. A computation for $a \in \{0, 1\}^n$ proceeds on B as follows. Let the label of v_1 be x_i and the label of v_2 be x_j. u_1 is defined as a state pointed from v_1 along with an outgoing edge labeled a_i and u_2 is defined as a state pointed from v_2 along with an outgoing edge labeled a_j.

In lines 14–23, the token on B moves from (v_1, v_2) to (u_1, u_2) if v_1 and v_2 have the same label x_i. This means that the token on B_1 and the token on B_2 move from v_1 to u_1 and from v_2 to u_2, respectively.

In lines 24–36, the token on B moves from (v_1, v_2) to (u_1, v_2) if v_1 and v_2 have a different label and $\pi^{-1}(i) < \pi^{-1}(j)$. This means that the token on B_1 moves form v_1 to u_1 and the token on B_2 remains on the current state. In these reason, B follows from a permutation π. In lines 4–5, the token of B reaches 0-sink if one token of OBDD B_1 or B_2 reaches 0-sink. In lines 6–7, the token of B reaches 1-sink if both tokens of OBDD B_1 and B_2 reach 1-sink. Thus, this OBDD constructed by this algorithm represents $f_1 \wedge f_2$.

B has at most $|B_1| \cdot |B_2|$ edges which represents transitions on B_1. B has at most $|B_1| \cdot |B_2|$ edges which represents transitions on B_2. Thus, the total number of edges in B is at most $2|B_1| \cdot |B_2|$. And, this algorithm needs constant steps per one edge. So, this algorithm takes $O(|B|)$ times and $|B| \leq 2|B_1| \cdot |B_2|$ □

Lemma 9 *Let B be an OBDD with π. Then there exist an algorithm that construct a nondeterministic OBDD B^R with π^R in $O(|B|)$ time. The size of B^R, denoted by $|B^R|$, is $O(|B|)$.*

Proof. We show that Algorithm 2 produces B^R satisfies this lemma. In lines 1–5, all edges into 0-sink are removed. In lines 4–13, this algorithm adds new states to all states having the same label parents. For each state v and if a set of states $U = \{u \mid (u, v) \in E\}$ includes at least two different labels, let x_ℓ be a state labeled x_i in U such as $\pi^{-1}(i)$ is maximum. For any state $u \in U$ with $r(u) \neq x_\ell$,

(1) Add a new state w labeled x_ℓ to B.

(2) Add a new edge (u, w) labeled the same label of edge (u, v) to B.

(3) Add two new edges (w, v) labeled 0 and (w, v) labeled 1.

(4) Remove an edge (u, v).

By these operation, we can construct an modified OBDD which represents the same function of the original OBDD B. In addition, for each states in modified OBDD, its parents have the same label. Because each new states have only one in-degree, this modification is not applied to them. Thus, the number of iteration is at most $|E|$ and the size of modified OBDD becomes at most three times as that of B. Figure 4.4 is an example of this operation applied to Figure 4.1.

In lines 14–26, this algorithm construct B^R from above modified OBDD as follows.

(1) A label of any state is replaced from a label of an initial state of a direct edge into it. A label of a root state is rewrite as 1.

(2) Reverse the direction of each edge.

(3) For any state v, if there exists no edge (v, w) labeled $b \in \{0, 1\}$, where $w \in V$, add a new edge (v, t_0) labeled b.

Each computation path on B^R corresponds to the reverse computation path of B. This module takes only $O(|B|)$ steps. Note that B^R can be nondeterministic OBDD if there exist at least 2 edges whose labels are the same and whose terminal vertices are the same. Figure 4.5 is a nondeterministic OBDD produced by this algorithm applied to Figure 4.1. In this way, algorithm can construct a nondeterministic OBDD B^R with π^R which represents the same function of B. The size of B^R, denoted by $|B^R|$, is $O(|B|)$. □

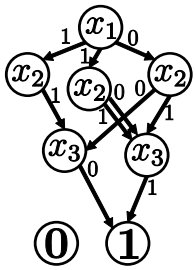

Figure 4.4: preprocessing

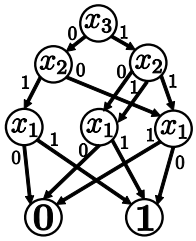

Figure 4.5: B^R

4.4 Main Algorithm

It is known that k-OBDD SAT with n variables can be solvable in poly(n) time [4]. It means that k-IBDD SAT with $\pi_1 = \pi_2 = \cdots = \pi_k$ can be solvable in polynomial time. Therefore, the main idea of our algorithm is to find a common permutation in each π_i and check all the assignment to variables not included that common permutation. At first, we show that a special k-IBDD SAT including k-OBDD SAT can be solvable in polynomial time. Algorithm 3 shows this algorithm.

Lemma 10 *Let k be a positive constant. Permutations $\pi_1, \pi_2, \ldots, \pi_k$ satisfies that $\pi_i = \pi_1$ of $\pi_i = \pi_1^R$ for all i $(2 \leq i \leq k)$. There exists a poly(n) time algorithm for k-IBDD SAT with n variables and $m = O(n^c)$ (c is an arbitrary positive constant) states.*

Proof. We extend Theorem 2 in [4] to nondeterministic k-OBDDs. Let B be an arbitrary nondeterministic k-OBDD and it satisfies the condition of this lemma. For i-th layer $(1 \leq i \leq k)$, let B_i be a nondeterministic OBDD and V_i be a set of states for i-th layer

In lines 1–12, our algorithm transforms B to a new B such that for each edge (u, v), there exists i such as $u, v \in V_i$, or $u \in V_i$ and $v \in V_{i+1}$.

Algorithm do the following operations if $(u, v) \in E$ and $j - i > 1$ for $u \in V_i$ and $v \in V_j$.

(1) Add a state w_ℓ labeled $x_{\pi_\ell(1)}$ to B_ℓ for all ℓ $(i < \ell < j)$.

(2) Add an edge (u, w_{i+1}) labeled the same label of (u, v) between B_i and B_{i+1}.

(3) Add two edges $(w_\ell, w_{\ell+1})$ labeled 0 and $(w_\ell, w_{\ell+1})$ labeled 1 for all ℓ between B_ℓ and $B_{\ell+1}(i < \ell < j - 1)$.

(4) Add two edges (w_{j-1}, v) labeled 0 and (w_{j-1}, v) labeled 1 between B_{j-1} and B_j

(5) Remove a edge (u, v).

Note that this procedure does not change added states and edges. Thus, the above operation is iterated E times and the size of new B is at most $2k$ times as that of B.

Each computation path which satisfies B reaches to 1-sink through $r_2 \in V_2$, $r_3 \in V_3$, $\ldots$, $r_k \in V_k$. The number of such paths is at most $(2k|B|)^{k-1}$. Algorithm check whether each path can reach to 1-sink. It chooses $k - 1$ states $r_2, \ldots, r_k$ $(r_i \in V_i)$ and constructs k OBDDs $B'_1, B'_2, \ldots, B'_k$ from $B_1, B_2, \ldots, B_k$, respectively. Note that each OBDD B'_i has a permutation π_i. For simplicity, let r_1 be the root state of B and r_{k+1} be 1-sink. It constructs each OBDD B'_i with a source state r_i from B_i for each i $(1 \leq i \leq k)$ as follows. At first, let V'_i be a set of states such as $t_{i,0}$, $t_{i,1}$ and a states $(\in V_i)$ which is reachable from r_i. A set of edges of B'_i contains any edge e' labeled $\phi_E(u, v)$ with the following property for all edge $e \in \{(u, v) \mid (u, v) \in E \wedge u \in V'_i\}$.

- $e' := (u, v)$ if $v \in V'_i$.
- $e' := (u, t_{i,1})$ if $v = r_{i+1}$.
- $e' := (u, t_{i,0})$ if $v \neq r_{i+1}$.

From this sight, if and only if k-IBDD B has an input whose computation path reach 1-sink, each B'_i shows computation path reaching $t_{i,1}$ with the same input. By Lemma 9, we can construct a nondeterministic OBDD B'^R_i with $\pi^R_i = \pi_1$ which is equivalent to a deterministic OBDD B'_i with $\pi_i = \pi^R_1$. We rename this B'^R_i by B'_i. Thus, it is sufficient to

check if there exists a satisfiable assignment such that it satisfies all k nondeterministic OBDDs $B'_1, B'_2, \ldots, B'_k$ with the same permutation π_1. Applying Lemma 8 to $B'_1, B'_2, \ldots, B'_k$ continuously, we can get the OBDD B^* which outputs 1 if and only if each B'_i has the same input whose computation path goes to $t_{i,1}$. Because $|B_i| \leq |B|$, This construction ends in $O(|B|^k)$ time and the size of $|B^*|$ is $O(|B|^k)$. We can check the satisfiability of OBDD B^* in $O(|B^*|) = O(|B|^k)$ times by solving reachability. Thus, a satisfiability of B can be solvable in $O(|B|^{2k-1})$ time. □

Before proving Theorem4, we show that 2-IBDD SAT can be solvable in time polynomially faster than 2^n. Algorithm 4 shows a deterministic polynomial space algorithm for 2-IBDD SAT.

Theorem 6 *For any instance of k-IBDD SAT with n variables and $m = O(n^c)$ states (c is an arbitrary positive constant), there exists an deterministic polynomial-space algorithm which solves it in poly(n) $\cdot\, 2^{n-\sqrt{n}}$ time, where poly(n) represents a polynomial of n.*

Proof. Let B be a given 2-IBDD with π_1, π_2. Without loss of generality, $\pi_1 = (1, 2, \ldots, n)$. We describe the detail of our algorithm. It computes a longest increasing sequence σ_{inc} and decreasing sequenceσ_{dec} of π_2 and let σ be the longest of the two. $|\sigma|$ denotes the length of σ. Let $Y := \{\sigma(i) \mid 1 \leq i \leq |\sigma|\}$. Then this algorithm check the satisfiability for all partial assignment on $X \backslash Y$. For each partial assignment a, we check if $B|_a$ is satisfiable or not. Note that B is satisfiable if there exists a such that $B|_a$ is satisfiable. B is unsatisfiable if there exists no a such that $B|_a$ is satisfiable. Now $B|_a$ is a 2-OBDD withσ, σ if $\sigma = \sigma_{inc}$, and $B|_a$ is a 2-OBDD withσ^R, σ. By Lemma 10 and [4], for any case, the satisfiability of $B|_a$ can be solvable in polynomial time. By Theorem 5, $|Y| \geq \sqrt{n}$, then $|X \backslash Y| \leq n - \sqrt{n}$. Thus, the overall running time of our algorithm is at most $poly(n) \cdot 2^{n-\sqrt{n}}$. □

Finally, we show a deterministic polynomial space algorithm for k-IBDD SAT in Algorithm 5.

Theorem 4 (Restate) *For any instance of k-IBDD SAT with n variables and $m = O(n^c)$ states (c is an arbitrary positive constant), there exists an deterministic polynomial-space algorithm which solves it in poly(n) $\cdot\, 2^{n-n^\alpha}$ time, where $\alpha = \frac{1}{2^{k-1}}$ and poly(n) represents a polynomial of n.*

Proof. Let B be a given k-IBDD with $\pi_1, \pi_2, \ldots, \pi_k$. Without loss of generality, $\pi_1 = (1, 2, \ldots, n)$. We describe the detail of our algorithm. At first, our algorithm computes a longest increasing and decreasing sequence of π_2 and let σ_2 be the longest of the two. $|\sigma_2|$

denotes the length of σ_2. By Theorem 5, $|\sigma_2| \geq \sqrt{n}$. In line 3, we obtain a subsequence π'_3 from π_3 such that any element in π'_3 is in σ_2. Next, our algorithm computes a longest increasing and decreasing sequence of π'_3 and let σ_3 be the longest of the two. $|\sigma_3|$ denotes the length of σ_3. Note that σ_3 is a increasing or decreasing sequence which consist of the elements in σ_2. As σ_2 itself is a increasing or decreasing sequence, σ_3 is a subsequence of σ_2 or σ_2^R. Thus, π_1, π_2, π_3 has a subsequence σ_3 or σ_3^R. By $|\pi'_3| = |\sigma_2|$ and Theorem 5, $|\sigma_3| \geq n^{1/4}$. Similarly, for $2 \leq i \leq k$, let π'_i be a subsequence of π_i such that any element in π'_i is in σ_{i-1}. Our algorithm computes a longest increasing and decreasing sequence of π'_i and let σ_i be the longest of the two. $|\sigma_i|$ denotes the length of σ_i. $\pi_1, \pi_2, \ldots, \pi_i$ have a subsequence σ_i or σ_i^R, and inductively $|\sigma_i| \geq n^{1/2^{i-1}}$ by Theorem 5. Note that each σ_i can be computed in poly(—σ_i) time. Let $Y := \{\sigma_k(i) \mid 1 \leq i \leq |\sigma_k|\}$. Similarly to the proof of Theorem 6, our algorithm check the satisfiability for all partial assignment on $X \backslash Y$. For each partial assignment a, we check if $B|_a$ is satisfiable or not. As $\pi_1, \pi_2, \ldots, \pi_k$ has a subsequence σ_k or σ_k^R, for each partial assignment a, each layer of k-IBDD $B|_a$ is an OBDD with σ_k or σ_k^R. By Lemma 10, we can check the satisfiability of $B|_a$ in polynomial time. As setting $\alpha = \frac{1}{2^{k-1}}$, $|X \backslash Y| = n - |\sigma_k| \leq n - n^\alpha$. Thus, the overall running time of our algorithm is at most $poly(n) \cdot 2^{n-n^\alpha}$. □

4.4.1 Chapter Summary

In this chapter, we propose the algorithm for k-IBDD SAT and this algorithm runs in $poly(n) \cdot 2^{n-n^\alpha}$ time, where $\alpha = \frac{1}{2^{k-1}}$ and poly(n) represents a polynomial of n. This algorithm finds the longest common sequence among every layers and assign to variables except variables in the common sequence. After the partial assignment, this algorithm conjoins every layers and makes one OBDD. Because OBDD SAT is solved in polynomial time, we can solve k-IBDD SAT faster. For more improving, we should find longer subsequence dividing each layer into smaller layers. This idea would give us more faster algorithms and these are our future works.

Algorithm 1 Conjunction(B_1, B_2)

Require: OBDD $B_1 = (G_1(V_1, E_1), \phi_{V_1}, \phi_{E_1})$ with π which represents f_1, OBDD $B_2 = (G_2(V_2, E_2), \phi_{V_2}, \phi_{E_2})$ with π which represents f_2

Ensure: OBDD $B = (G(V, E), \phi_V, \phi_E)$ with π which represents $f_1 \wedge f_2$

```
V := V_1 × V_2
E := ∅
for all v := (v_1, v_2) ∈ V do
  if φ_{V_1}(v_1) = 0 or φ_{V_2}(v_2) = 0 then
    φ_V(v) = 0
  else if φ_{V_1}(v_1) = 1 and φ_{V_2}(v_2) = 1 then
    φ_V(v) = 1
  else
    φ_V(v) = x_ℓ, where π^{-1}(ℓ) = min{π^{-1}(i), π^{-1}(j)} and φ_{V_1}(v_1) = x_i, φ_{V_2}(v_2) = x_j
for all v := (v_1, v_2) ∈ V do
  E'_i := {(v_i, u_i) | (v_i, u_i) ∈ E_i}, i ∈ {1, 2}
  if φ_{V_1}(v_1) = φ_{V_2}(v_2) then
    for all e_1 = (v_1, u_1) ∈ E'_1 do
      for all e_2 = (v_2, u_2) ∈ E'_2 do
        if φ_{E_1}(e_1) = φ_{E_2}(e_2) then
          e := ((v_1, v_2), (u_1, u_2))
          E := E ∪ {e}
          φ_E(e) = φ_{E_1}(e_1)
  else if φ_V(v) = φ_{V_2}(v_2) then
    for all e = (v_2, u_2) ∈ E'_2 do
      e := ((v_1, v_2), (v_1, u_2))
      E := E ∪ {e}
      φ_E(e) = φ_{E_2}(e_2)
  else
    for all e = (v_1, u_1) ∈ E'_1 do
      e := ((v_1, v_2), (u_1, v_2))
      E := E ∪ {e}
      φ_E(e) = φ_{E_1}(e_1)
return B = (G(V, E), φ_V, φ_E)
```

Algorithm 2 Reverse(B)

Require: An OBDD $B = (G(V, E), \phi_V, \phi_E)$ with π.

Ensure: A nondeterministic OBDD $B^R = (G^R(V^R, E^R), \phi_{V^R}, \phi_{E^R})$ with π^R and represents the same function of B

{preprocedure}

for all $e = (u, v) \in E$ **do**
  if $\phi_V(v) = \mathbf{0}$ **then**
    $E := E \backslash \{e\}$
for all $v \in V$ **do**
  $L_{(*,v)} := \{i \mid x_i = \phi_V(u)$ **and** $(u, v) \in E\}$
  if $|L_{(*,v)}| > 1$ **then**
    Choose $\ell \in L_{(*,v)}$ such that $\pi^{-1}(\ell) \geq \pi^{-1}(i)$ for all $i \in L_{(*,v)}$.
    for all $e = (u, v) \in E$ **do**
      if $\phi(u) \neq x_\ell$ **then**
        $V := V \cup \{w\}$ **and** $\phi_V(w) = x_\ell$
        $E := E \cup \{e_1 := (w, v), e_2 := (w, v), e' := (u, w)\}$
        $\phi_E(e_1) := 0, \phi_E(e_2) := 1, \phi_E(e') := \phi_E(e)$
        $E := E \backslash \{e\}$

{A construction of $G^R(V^R, E^R)$}

$V^R := V$, $E^R := E$
for all $e = (u, v) \in E$ **do**
  $\phi_{V^R}(v) := \phi_V(u)$
$\phi_{V^R}(r) := \mathbf{1}$, where r is a root state of G
for all $e = (u, v) \in E$ **do**
  $e^R := (v, u)$
  $\phi_{E^R}(e^R) = \phi_E(e)$
  $E^R = E^R \cup \{e^R\} \backslash \{e\}$
for all $v \in V^R$ **do**
  if there exists no edge (v, w) labeled $b \in \{0, 1\}$ **then**
    $e^R := (v, t_0)$
    $E^R = E^R \cup \{e^R\}$
    $\phi_{E^R}(e^R) = b$
return $B = (G^R(V^R, E^R), \phi_{V^R}, \phi_{E^R})$

Algorithm 3 (π, π^R)-k IBDD SAT(B)

Require: k-OBDD $B = (G, \phi_V, \phi_E)$ with $\pi_1, \ldots, \pi_k$, where $\pi_i = \pi_1$ or $\pi_i = \pi_1^R$ $(2 \leq i \leq k)$
Ensure: outputs "Yes" if B is satisfiable, "No" otherwise.

```
for all e = (u, v) ∈ E do
  if j − i > 1, where u ∈ V_i, v ∈ V_j then
    for ℓ = i + 1, ..., j − 1 do
      V_ℓ := V_ℓ ∪ {w_i} and φ_V(w_i) := x_{π_ℓ(1)}
    E := E ∪ {e' := (u, w_{i+1})}
    φ_E(w_i) := x_{π_ℓ(1)}
    for ℓ = i + 1, ..., j − 2 do
      E := E ∪ {e'_0 := (w_ℓ, w_{ℓ+1}), e'_1 := (w_ℓ, w_{ℓ+1})}
      φ_E(e'_0) := 0, φ_E(e'_0) := 1
    E := E ∪ {e'_0 := (w_{j−1}, v), e'_1 := (w_{j−1}, v)}
    φ_E(e'_0) := 0, φ_E(e'_0) := 1
    E := E\{e}
for all (r_2, ..., r_k) ∈ V_2 × ··· × V_k do
  for i = 1, ..., k do
    V'_i := {v | v ∈ V_i and v is reachable from r_i} ∪ {t^i_0, t^i_1}
    φ_{V'_i}(t^i_0) := 0, φ_{V'_i}(t^i_1) := 1
    E'_i := ∅
    for all e = (u, v) ∈ E, where u ∈ V'_i do
      if v ∈ V'_i then
        E'_i := E'_i ∪ {e' := (u, v)}
      else if v = r_{i+1} then
        E'_i := E'_i ∪ {e' := (u, t^i_1)}
      else
        E'_i := E'_i ∪ {e' := (u, t^i_0)}
      φ_{E'_i}(e') := φ_E(e)
    B'_i := (G'_i(V'_i, E'_i), φ_{V'_i}, φ_{E'_i})
  for i = 2, ..., k do
    if π_i = π_1^R then
      B'_i :=Reverse(B'_i)
    B'_1 :=Conjunction(B'_1, B'_i)
  if B'_1 has a path from a root state to 1-sink then
    return "Yes"
return "No"
```

Algorithm 4 2-IBDD SAT

Require: 2-IBDD $B = (G, \phi_V, \phi_E)$ with π_1, π_2, where $\pi_1 = (1, \ldots, n)$.
Ensure: outputs "Yes" if B is satisfiable, "No" otherwise.

1: Compute a longest increasing sequence σ_{inc} and a longest decreasing sequence σ_{dec} of π_2.
2: **if** $|\sigma_{inc}| \geq |\sigma_{dec}|$ **then**
3: $\sigma = \sigma_{inc}$
4: **else**
5: $\sigma = \sigma_{dec}$
6: $Y := \{\sigma(i) \mid 1 \leq i \leq |\sigma|\}$
7: **for all** $a \in \{0, 1, *\}^n$, where $S(a) = X \backslash Y$ **do**
8: **if** (σ, σ^R)-k-IBDD SAT$(B|_a)$ = "Yes" **then**
9: **return** "Yes"
10: **return** "No"

Algorithm 5 k-IBDD SAT

Require: k-IBDD $B = (G, \phi_V, \phi_E)$ with $\pi_1, \ldots, \pi_k$, where $\pi_1 = (1, \ldots, n)$
Ensure: outputs "Yes" if B is satisfiable, "No" otherwise.

1: $\sigma_1 := \pi_1$
2: **for** $i = 2, \ldots, k$ **do**
3: $\pi'_i := (\pi_i(j_1), \pi_i(j_2), \ldots, \pi_i(j_{|\sigma_{i-1}|}))$, where $j_1 < j_2 < \cdots < j_{|\sigma_{i-1}|}$ and $\forall p, \exists q, \pi'_i(p) = \sigma_{i-1}(q)$
4: Compute a longest increasing sequence σ_{inc} and a longest decreasing sequence σ_{dec} of π'_i.
5: **if** $|\sigma_{inc}| \geq |\sigma_{dec}|$ **then**
6: $\sigma_i := \sigma_{inc}$
7: **else**
8: $\sigma_i := \sigma_{dec}$
9: $Y := \{\sigma_k(i) \mid 1 \leq i \leq |\sigma_k|\}$
10: **for all** $a \in \{0, 1, *\}^n$, where $S(a) = X \backslash Y$ **do**
11: **if** (σ_k, σ_k^R)-k-IBDD SAT$(B|_a)$ = "Yes" **then**
12: **return** "Yes"
13: **return** "No"

Chapter 5

Efficient Algorithms for Sorting k-Sets in Bins

5.1 Introduction

The Sorting problem is a classical fundamental problem in theoretical computer science. Various kinds of sorting problems have been studied [15, 16, 18]. One of the most basic sorting problems is the Swap-Sort problem : Given an list with n integer numbers in non-increasing order, if we are only allowed to swap two adjacent rows, how many swaps do we need to sort them in non-decreasing order? It is well-known fact that $\binom{n}{2}$ swaps are necessary and sufficient. In this chapter, we consider a natural extension of this problem. Peter Winkler [50] introduced "Sorting Pairs in Bins" and Ito, Teruyama and Yoshida [26] extended it to more general problem "Sorting k-Sets in Bins".

Sorting k-Sets in Bins: We are given n numbered bins each with k numbered balls, such that bin i is adjacent to bins $i-1$ and $i+1$, bin n is not adjacent to bin 1, and the balls in bin i are each numbered $n+1-i$. We may swap any two balls between adjacent bins. How many swaps are necessary to get every ball into the bin carrying its number?

For $k=2$, Winkler [50] showed a lower bound $\lceil \binom{2n}{2}/3 \rceil = \lceil \frac{n(2n-1)}{3} \rceil$ and asked whether this lower bound is optimal or not. It is easy to see n^2 swaps is sufficient by using bubble sort twice. West [48] proposed an algorithm with $\frac{4}{5}n^2$ swaps. Ito, Teruyama and Yoshida [26] gave an affirmative answer to Winkler's question by designing an almost optimal algorithm with $\frac{2}{3}n^2 - O(n)$ swaps, and Püttman [39] independently showed a similar result. Ito, Teruyama and Yoshida also stated (without proof) a lower bound for any k and proposed

the question of whether there exists an algorithm satisfying the lower bound when $k \geq 3$. When $k = 3$, by using bubble sort three times, we can easily get an algorithm with at most $\frac{3}{2}n^2$ swaps. Moreover, by combining bubble sort with the algorithm in [26], the number of swaps decreases to $\frac{7}{6}n^2$. However, the lower bound is $\frac{9}{10}n^2$. There still remains a large gap between the upper bound and the lower bound even when $k = 3$. Therefore, a natural problem is to design a more efficient algorithm for $k = 3$. In this paper, we give two efficient algorithms, one greedy, one recursive, for this problem when $k = 3$ and show that our greedy algorithm is applicable to the problem for any k and its performance approaches to the lower bound as k and n increase.

Our Contribution

We show two algorithms for Sorting 3-Sets in Bins, one of which can be applied to Sorting k-Sets in Bins. We call the algorithms $Greedy(n, k)$ and $Recursive(n)$. For the Sorting 3-Sets in Bins problem, the number of swaps is $n^2+O(n)$ for $Greedy(n, 3)$ and $\frac{15}{16}n^2+O(n)(= 0.9375n^2+O(n))$ for $Recursive(n)$. These values are close to the $\frac{9}{10}n^2(= 0.9n^2)$ lower bound shown in [26]. For Sorting k-Sets in Bins problem, $Greedy(n, k)$ achieves $\frac{k+1}{4}n^2 + O(kn)$ swaps. This result asymptotically approaches to the lower bound $(1-\frac{k-1}{2k^2+k-1})\frac{k+1}{4}n^2+O(n)$ as k and n increase. Formally, we prove the following two theorems.

Theorem 7 *There exists a greedy algorithm $Greedy(n, k)$ which solves Sorting k-Sets in Bins with $\frac{k+1}{4}n^2 + O(kn)$ swaps.*

Theorem 8 *There exists a recursive algorithm $Recursive(n)$ which solves Sorting 3-Sets in Bins with $\frac{15}{16}n^2 + O(n)$ swaps.*

5.1.1 Related Work

There are very few results to our problem. However, many other kinds of sorting problems have been well studied. For example, Partial Quicksort is the problem that given a list with integer, how many comparison do we need to sort the first l-th smallest elements in the list? It was introduced by [24] and shown a tight upper bound in [33]. Another example is the sorting problem for partially ordered sets, in which some pairs of elements are incomparable. Faigle and Tuŕan introduced this problem [19] and very recently [14] made substantial advances. For other sorting problems, e.g. [5, 11, 15, 16, 18, 21, 22].

Paper Organization In subsection 5.2, we give a proof of the lower bound of Sorting k-Sets in Bins and some notation needed for the description of our algorithms. In subsection 5.3, we present the algorithm $Greedy(n, k)$ and provide an analysis of its performance.

In subsection 5.4, we present a more efficient algorithm $Recursive(n)$ and analyze its performance.

5.2 Preliminaries

Lower Bounds

In this section, we give a precise proof of the lower bound for any k [26]. Our proof is based on the point system used by Winkler [50]. An algorithm starts with 0 points. With each swap, it gets some number of points corresponding to the change in the states of the bins. We can easily calculate the total points t needed to complete the sorting and the maximum points m obtained by any one swap in the algorithm. Thus, we get the lower bound t/m. We prove the following theorem:

Theorem 9 *To solve Sorting k-Sets in Bins, we need at least* $\left\lceil \binom{kn}{2}/(2k-1) \right\rceil$ *swaps if* n *is even, and* $\left\lceil \left(\binom{kn}{2} - \binom{k-1}{2} \right) / (2k-1) \right\rceil$ *swaps if* n *is odd.*

Proof. We first construct our point system. Given two balls numbered x and y, $x \neq y$, (without loss of generality $x > y$), we will refer to these balls as "ball x", "ball y" respectively.

Passing (1 point) : Ball x passes ball y from left to right. In other words, we swap balls x and y.

Catching up (1/2 point) : Before the swap, ball x and y are in different bins. After the swap, they are in the same bin. (See Fig. 5.1.)

Moving on (1/2 point) : Let $u > v$. Before the swap, balls x and y are in the same bin. After swap, ball x is in the u-th bin and ball y is in the v-th bin. (See Fig. 5.2.)

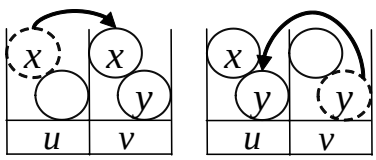

Figure 5.1: Catching up

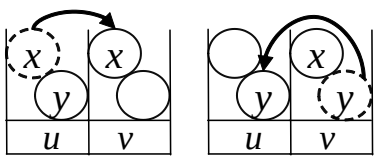

Figure 5.2: Moving on

It is easy to see that for any x and y, a set of "catching up" and "moving on" operations is the same as a "passing" operation.

Next, we consider the case where $x = y$. Because we refer to two different balls, we will continue to refer to "ball x" and "ball y".

Separate (1/2 point) : Before the swap, balls x and y are in the same bin. After the swap, they are in different bins.

Recombine (1/2 point) : Before the swap, balls x and y are in different bins. After the swap, they are in the same bin.

Now, we calculate the total points needed to complete sorting. We first assume the initial state is in reverse order. If balls x and y have different numbers and $x > y$, clearly ball x must pass ball y. Thus, for any pair of different numbered balls, either a passing operation or a set of moving on and catching up operations must occur at least once. If balls x and y have the same number, then balls x and y must separate at some swap and recombine at some other swap. In each case, a 1 point charge is incurred. Thus, to complete sorting, we need at least $\binom{kn}{2}$ points. Note that if n is odd, $k-1$ balls labeled $\lceil n/2 \rceil$ may not move, so the point total is $\binom{k-1}{2}$ less than the case when n is even. Therefore, we charge at least $\binom{kn}{2}$ points if n is even, and $\binom{kn}{2} - \binom{k-1}{2}$ points if n is odd.

Let us consider how many points can be obtained in one swap. Suppose that ball x is in the v-th bin and ball y is in the u-th bin, where $u > v$. The maximum amount of points that we can get by swapping x and y is 1 (by passing). Therefore, it suffices to consider the case of $x > y$. In this case, we focus on the other balls in the v-th bin and u-th bin. Let ball z_i, $i \in \{1, \ldots k-1\}$, be a non-y ball in the u-th bin. If each ball z_i satisfies $z_i \leq x$, for each z_i we can get $\frac{1}{2}$ point by catching up or recombine. In addition, we consider the relation between ball y and ball z, for each z_i satisfying $y \leq z$ we obtain $\frac{1}{2}$ point by applying moving on or separate operations between ball y and ball z_i. The algorithm gets at most $2 \cdot \frac{1}{2}(k-1)$ points by these arguments. By a similar argument on the balls in the v-th bin, it is possible to get at most $2 \cdot \frac{1}{2}(k-1)$ points. Thus, the maximum total point we can get is $1 + 4 \cdot \frac{1}{2}(k-1) = 2k-1$. Let $T(n)$ be the number of swaps necessary to complete sorting. $T(n) \geq \lceil \binom{kn}{2}/(2k-1) \rceil$ if n is even, $T(n) \geq \lceil (\binom{kn}{2} - \binom{k-1}{2})/(2k-1) \rceil$ if n is odd. □

Notations and Definitions

In this section, we provide some notations and definitions used to explain our algorithms. To be distinguishable and comparable, we assign all same-numbered balls an index from 1 to k. Let x_i be the ball labeled x and indexed i. We define the total order of balls as $x_i < y_j$ if $x < y$ or if $x = y$ and $i < j$. We represent the state of balls and bins as in Fig 5.3.

The numbers in the bottom row are the labels of bins and the other numbers correspond to balls.

The *initial state* of the n bins is the state where the i-th bin has k balls labeled $n+1-i$. (See Fig. 5.3.) The *target state* of the n bins is the state where the i-th bin has k balls labeled i. (See Fig. 5.4.) The *solution for n bins* begins with an initial state and ends with a target state for n bins.

If we *move* a ball x_i to the u-th bin, we swap x_i for the lowest-labeled ball in the bin to x_i's right until x_i arrives at the u-th bin. We know the following fact directly from the properties of "move".

n_1	$n-1_1$		2_1	1_1
n_2	$n-1_2$	$\cdots$	2_2	1_2
n_3	$n-1_3$		2_3	1_3
1	2	$\cdots$	$n-1$	n

Figure 5.3: Initial state of n bin as $k=3$

1_1	2_1		$n-1_1$	n_1
1_2	2_2	$\cdots$	$n-1_2$	n_2
1_3	2_3		$n-1_3$	n_3
1	2	$\cdots$	$n-1$	n

Figure 5.4: Target state of n bin as $k=3$

Fact 1 *Suppose that the ball x_i is in the u-th bin and the ball y_j is in the v-th bin with $v < u$. If the ball x_i shifts to the $(u-1)$-th bin by moving the ball y_j to the w-th bin (where $w \geq u$), then the ball x_i must have the minimum label in the u-th bin.*

5.3 Greedy Algorithms

5.3.1 Algorithm Description

Fig. 5.5 introduces a greedy algorithm *Greedy*(n,k) which solves Sorting k-Sets in Bins with n bins for any n,k.

Greedy(n,k), n,k**: integer**
1: **for** $x = n$ to 2
2: **for** $i = k$ to 1
3: Move the ball x_i to the x-th bin
4: **end for**
5: **end for**

Figure 5.5: Greedy Algorithm

Lemma 11 *The algorithm Greedy(n,k) solves Sorting k-Sets in Bins with n bins.*

Proof. Let us consider a ball n_i ($i \in [k]$). At first, we see that the n-th bin contains all balls labeled n_i at the end of the loop on line 1 in this algorithm when $x = n$. All balls

labeled n_i go to the n-th bin on lines 2–4. If a ball n_j ($j \in [k]$) leaves from the the n-th bin, it must be the minimum number in the n-th bin by Fact 1. This situation occurs only when the n-th bin has k balls labeled n_i and j is 1. This means the loop on line 1 with $x = n$ is over. Thus, the n-th bin must contain all balls n_i and the other balls are in the other bins at the end of the loop on line 1 with $x = n$. Next, let us consider the loop on line 1 with $x = n - 1$. By substituting n with $n - 1$ and using the fact that this algorithm does not swap the balls in the $(n - 1)$-th bin with the n-th bin, we get a similar result. By repeating this argument, we conclude that $Greedy(n, k)$ correctly solves Sorting k-Sets in Bins with n bins. □

5.3.2 Analysis of the Number of Swaps

In this section, we estimate the number of swaps used by $Greedy(n, k)$ to solve Sorting k-Sets in Bins with n bins.

Lemma 12 *The number of swaps performed by the algorithm $Greedy(n, k)$ is at most $\frac{k+1}{4}n^2 + O(kn)$.*

Proof. To prove Lemma 12, we prove the following lemma.

Lemma 13 *For any x and any i, where $1 \leq x \leq n$ and $2 \leq i \leq k$, if a ball x_i shifts left from its initial $(n - x + 1)$-th bin, then x_i has the minimum label of all balls in any bin from $(n - x + 1)$ to n.*

Proof. Let us consider the case where a ball x_i shifts left from its initial $(n - x + 1)$-th bin. From Fact 1, the ball x_i is the minimum-labeled ball in the $(n-x+1)$-th bin. Because $i \geq 2$, the $(n - x + 1)$-th bin must have a ball y_j, where $y > x$ and $j \in [k]$. Initially, that ball y_j started in the $(n-y+1)$-th bin, which is further left than the $(n-x+1)$-th bin. In each iteration of the loop on line 1, no balls shift right except when they go to their target bin. So, if the above situation happen, y must be $(n - x + 1)$ and loop iterations from n to $y + 1$ on line 1 are over. By the proof of Lemma 11, for all $u > n - x + 1$, the u-th bin contains all balls u_l ($l \in [k]$). Thus, the proof is completed. □

First, we count the total number of swaps needed for a ball labeled in $[\lfloor n/2 \rfloor + 1, n]$ to go to its target bin.

From Lemma 13, for $\lfloor n/2 \rfloor + 1 \leq x \leq n$ and $2 \leq i \leq k$, a ball x_i does not shift left until it moves to the x-th bin. This means that these x_i balls are in the $(n - x + 1)$-th bin. The

number of swaps needed for a ball x_i to move to the x-th bin is $2x - n - 1$. In addition, the number of swaps needed for a ball x_1 moves to x-th bin is trivially at most $x - 1$.

Thus, the total number of swaps needed for all balls x_i ($x \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}, i \in [k]$) to move to their target bins is at most

$$\sum_{x=\lfloor n/2 \rfloor+1}^{n} \{x - 1 + (k-1) \times (2x - n - 1)\} = \frac{2k+1}{8} n^2 + O(kn). \tag{5.1}$$

Next, we count the total number of swaps required to move the remaining balls to their target bins. To analyse this, we need some lemmas. From Fact 1 and Lemma 13, we have Lemma 14 as follows.

Lemma 14 *For x, $1 \leq x \leq \lfloor n/2 \rfloor$ and $i \in \{2, \ldots, k\}$, suppose that a ball x_i has shifted left at least once and is currently in the l-th bin, where $l < n - x + 1$. Then, all balls in the u-th bin have greater labels than x_i for all $u > l$.*

Lemma 15 *For x, $1 \leq x \leq \lfloor n/2 \rfloor$ and $i \in \{2, \ldots, k\}$, a ball x_i does not shift left from the x-th bin.*

Proof. Assume that a ball x_i shifts left from the x-th bin. From Fact 1, this ball x_i must have the minimum label in the x-th bin. Moreover, from Lemma 14, all balls in the u-th bin are greater than x_i for all $u > x$. This means that the number of balls which are greater than x_i is at least $(n - x + 1)k - 1$. However, the number of balls which have greater labels than x_i is $(n - x)k + k - i = (n - x + 1)k - i$. It leads to a contradiction, because $i \geq 2$. □

Now we have all the tools required to count the total swaps to move the remaining balls to their target bins. From the proofs of Lemmas 11 and 15, for $y \leq \lfloor n/2 \rfloor$, $k - 1$ balls y_i ($i \in \{2, \ldots, k\}$) are in the y-th bin after the iteration of loop on line 1 with $x = y$. Thus we only need to move the ball y_1 to complete the loop iteration on line 1 with $x = y$. That is, the number of swaps we need to move ball y_1 is trivially at most $y - 1$. Therefore, in total, the number of swaps required for the loop on line 1 with $y = \lfloor n/2 \rfloor$ to 2 is at most

$$\sum_{y=2}^{\lfloor n/2 \rfloor} (y - 1) = \frac{1}{8} n^2 + O(n). \tag{5.2}$$

From equations 5.1 and 5.2, in total, the number of swaps $Greedy(n, k)$ needs is at most

$$\frac{2k+1}{8} n^2 + O(kn) + \frac{1}{8} n^2 + O(n) = \frac{k+1}{4} n^2 + O(kn). \tag{5.3}$$

Thus the Lemma 12 is proved. □

Proof of Theorem 7. Directly follows from Lemmas 11, 12. □

5.4 Recursive Algorithms

5.4.1 Basic Ideas

In this section, we present a recursive algorithm which is more efficient for the case $k = 3$. Before describing the details of our recursive algorithm, we outline it using the case where n is a multiple of 6 as an example.

First, we move the ball n_3 from the 1st bin to the n-th bin by using $n-1$ swaps. Next, we move the ball n_2 from the 1st bin to the n-th bin by using $n-1$ swaps, and the state becomes as in Fig. 5.6. Then, we move the ball $n-1_3$ from the 2nd bin to the $(n-1)$-th bin using $n-3$ swaps, and move the ball $n-1_2$ from the 2nd bin to the $(n-1)$-th bin using $n-3$ swaps. Now, the state is as in Fig. 5.7. For $x = n$ to $n/2+1$, we similarly move the ball labeled x_3 to the x-th bin, and then move the ball labeled x_2 to the x-th bin. Now, the resulting state is as in Fig. 5.8.

$n-2_1$	$n-3_1$	$n-4_1$		1_1	1_2	1_3
$n-1_1$	$n-1_2$	$n-2_2$	$\cdots$	3_2	2_2	n_2
n_1	$n-1_3$	$n-2_3$		3_3	2_3	n_3
1	2	3	$\cdots$	$n-2$	$n-1$	n

Figure 5.6: After moving balls n_3 and n_2 to the n-th bin

$n-2_1$	$n-5_1$	$n-6_1$		2_2	2_3	1_3
$n-1_1$	$n-4_1$	$n-2_2$	$\cdots$	3_2	$n-1_2$	n_2
n_1	$n-3_1$	$n-2_3$		3_3	$n-1_3$	n_3
1	2	3	$\cdots$	$n-2$	$n-1$	n

Figure 5.7: After moving balls $n-1_3$ and $n-1_2$ to the $(n-1)$-th bin

Here, for the first $n/3$ bins (i.e. from the 1st bin to the $(n/3)$-th bin), we relabel the ball x_1 to $\lceil x/3 \rceil$ where $x \in [n]$. Using this state as the initial state of the $n/3$ bins problem,

$n-2_1$		1_1	1_2	4_2		$n/2-2_2$	$n/2_3$		2_3	1_3
$n-1_1$	$\cdots$	2_1	2_2	5_2	$\cdots$	$n/2-1_2$	$n/2+1_2$	$\cdots$	$n-1_2$	n_2
n_1		3_1	3_3	6_2		$n/2_2$	$n/2+1_3$		$n-1_3$	n_3
1	$\cdots$	$n/3$	$n/3+1$	$n/3+2$	$\cdots$	$n/2$	$n/2+1$	$\cdots$	$n-1$	n

Figure 5.8: After moving all balls x_3 and x_2 to x-th bin for $x = n$ to $n/2+1$

we can recurse. After the recursive call, returning the labels of the balls in the first $n/3$ bins to their original labels yields a state where the 1st bin contains balls 1_1, 2_1 and 3_1, the 2nd bin contains balls 4_1, 5_1 and 6_1, and so on. (See Fig. 5.9.)

1_1	4_1		$n-2_1$	1_2		$n/2-2_2$	$n/2_3$		2_3	1_3
2_1	5_1	$\cdots$	$n-1_1$	2_2	$\cdots$	$n/2-1_2$	$n/2+1_2$	$\cdots$	$n-1_2$	n_2
3_1	6_1		n_1	3_3		$n/2_2$	$n/2+1_3$		$n-1_3$	n_3
1	2	$\cdots$	$n/3$	$n/3+1$	$\cdots$	$n/2$	$n/2+1$	$\cdots$	$n-1$	n

Figure 5.9: The state after sorting for a recursive structure

Finally, for $x = n$ to 2, we move ball x_1 to the x-th bin as follows. First, when we move the ball n_1 from the $(n/3)$-th bin to the n-th bin, each ball $1_2, 4_2, 7_2, \ldots n/2-2_2$ and $n/2_3, n/2-1_3, \ldots, 1_3$ is shifted left. (See Fig. 5.10.) Next, when moving the ball $n-1_1$ from the $(n/3)$-th bin to the $(n-1)$-th bin, each ball $2_2, 5_2, 8_2, \ldots, n/2-1_2$ and $n/2-1_3, \ldots, 1_3$ is shifted left. (See Fig. 5.11.) Similarly, we move balls $n-2_1, n-3_1, \ldots, 2_1$ to their target bins respectively in this order, the sorting completing.

1_1		1_2	2_2		$n/2-1_2$	$n/2-1_3$		1_3	n_1
2_1	$\cdots$	$n-2_1$	3_2	$\cdots$	$n/2_2$	$n/2+1_2$	$\cdots$	$n-1_2$	n_2
3_1		$n-1_1$	4_3		$n/2_3$	$n/2+1_3$		$n-1_3$	n_3
1	$\cdots$	$n/3$	$n/3+1$	$\cdots$	$n/2$	$n/2+1$	$\cdots$	$n-1$	n

Figure 5.10: After moving the ball n_1 to the n-th bin

For n which are not multiples of 6, the same strategy can be applied. The details are shown in the next subsection.

5.4.2 Formal Description of the Algorithm

We present a recursive algorithm *Recursive*(n). (See Fig 5.12.) This algorithm solves n bins with $k = 3$.

Now, we prove the correctness of our algorithm.

1_1		1_2	3_2		$n/2-1_3$	$n/2-2_3$		$n-1_1$	n_1
2_1	$\cdots$	2_2	4_2	$\cdots$	$n/2_2$	$n/2+1_2$	$\cdots$	$n-1_2$	n_2
3_1		$n-2_1$	5_2		$n/2_3$	$n/2+1_3$		$n-1_3$	n_3
1	$\cdots$	$n/3$	$n/3+1$	$\cdots$	$n/2$	$n/2+1$	$\cdots$	$n-1$	n

Figure 5.11: After moving the ball $n-1_1$ to the $(n-1)$-th bin

---

Recursive(n), n**: integer**
1: **for** $x = n$ to $\lceil n/2 \rceil + 1$
2: Move the ball x_3 to the x-th bin
3: Move the ball x_2 to the x-th bin
4: **end for**
5: **for** $x = 1$ to n
6: Relabel the ball x_1 (in the 1st bin to $\lceil n/3 \rceil$-th bin) to $\lceil x/3 \rceil$
7: **end for**
8: **If** the $\lceil n/3 \rceil$-th bin has any ball y_2 or y_3 ($y \in [n]$)
9: **then** Relabel this ball with $\lceil n/3 \rceil$
10: Solve for the first $\lceil n/3 \rceil$ bins by applying *Recursive*$(\lceil n/3 \rceil)$
(By lines 5–9, all balls in the first $\lceil n/3 \rceil$ bins are numbered at most $\lceil n/3 \rceil$. Therefore the first $\lceil n/3 \rceil$ bins are reduced to an instance of Sorting 3-Sets in Bins with $\lceil n/3 \rceil$ bins.)
11: relabel all balls in the first $\lceil n/3 \rceil$ bins with their original numbers
12: **for** $x = n$ to 2
13: Move the ball x_1 to the x-th bin
14: **end for**

Figure 5.12: Recursive Algorithm

Lemma 16 *The algorithm Recursive*(n) *solves Sorting k-set Bins with n bins.*

Proof. We divide the algorithm into three steps as follows: Step 1 is lines 1–4, Step 2 is lines 5–11 and Step 3 is lines 12–14, respectively. We will show which bins have which balls after each step.

[Step 1] We will consider three cases, where each case is with respect to the index of the balls (e.g. x_1, x_2, x_3).

A. Let x_1 be an arbitrary ball with index 1 and $x \in [n]$. Let m be the number of times the ball x_1 moves left. Note that the ball x_1 is initially in the $(n-x+1)$-th bin. We consider what happens when we move balls $n-y+1_3$ or $n-y+1_2$ from the y-th bin to the $(n-y+1)$-th bin for each y. If the ball x_1 is located in the u-th bin, where $u > y$, then x_1 shifts left once during this movement. After iterating $2 \cdot \lfloor n/2 \rfloor$ times, the ball x_1 goes to the $(n-x+1-m)$-th bin, where m is the largest integer which

satisfies $\lceil m/2 \rceil < n - x + 2 - m$. That is,

$$m = \max\{0 \le i \le n-1 \mid i + \lceil i/2 \rceil < n - x + 2\}.$$

We set r and q such that $n - x + 2 = 3r + q, q \in \{0, 1, 2\}$. Then, m is $2r - 1$ if $q = 0$ and $2r$ if $q = 1, 2$. After Step 1, the ball x_1 is in the bin numbered

$$n - x + 1 - m = 3r + q - 1 - m = \begin{cases} r & (q = 0, 1) \\ r + 1 & (q = 2) \end{cases}.$$

As $x = n - 3r - q + 2$, the r-th bin contains the three balls $n - 3r + 1_1$, $n - 3r + 2_1$, and $n - 3r + 3_1$ after Step 1, where $x \in [n]$, $1 \le r \le \lceil n/3 \rceil$.

B. Let x_2 be an arbitrary ball with the index 2 and $x \in \{1, \ldots, \lceil n/2 \rceil\}$. By analysis similar to the proof of Lemma 13, one can see the following fact.

Fact 2 *The ball x_2 does not shift before the ball $n - x + 1_2$ moves to the $(n - x + 1)$-th bin in Step 1.*

We consider what happens to x_2 when we move balls $n - y + 1_2$ or $n - y + 1_3$ from the y-th bin to the $(n - y + 1)$-th bin, where $n - y + 1 \le n - x + 1$. If the ball x_2 is in the u-th bin, where $u > y$, x_2 shifts left during this movement. By a similar argument to case A, the ball x_2 goes to the $(n + x - m)$-th bin, where m is the maximum integer which satisfies $\lceil m/2 \rceil < (n - x + 1) - (m - 1 - 2x + 1)$; that is,

$$m = \max\{0 \le i \le n-1 \mid i + \lceil i/2 \rceil < n + x + 1\}.$$

We set $n + x + 1 = 3r + q, q \in \{0, 1, 2\}$. By calculation, m is $2r - 1$ if $q = 0$ and $2r$ if $q = 1, 2$. Thus, after Step 1, the ball x_2 is in bin

$$n + x - m = 3r + q - 1 - m = \begin{cases} r & (q = 0, 1) \\ r + 1 & (q = 2) \end{cases}.$$

As $x = 3r + q - n - 1$, the r-th bin contains the three balls $3r - n - 2_2$, $3r - n - 1_2$, and $3r - n_2$ after Step 1, where $\lceil (n+1)/3 \rceil \le r \le \lceil n/2 \rceil$.

C. Any ball x_3 $(x = 1, \ldots, \lceil n/2 \rceil)$ does not shift.

[Step 2] Applying the recursive algorithm, balls x_1 $(x = 1, \ldots, n)$ are sorted in the first $\lceil n/3 \rceil$ bins. Each ball x_1 is in the $\lceil x/3 \rceil$-th bin.

[Step 3] Let us observe the state after moving the ball n_1 to the n-th bin. Note that the ball n_1 is in the $\lceil n/3 \rceil$-th bin after Step 2.

Balls of the form x_1 $(x = 1, \ldots, n-1)$ do not shift, since their bin numbers are less than or equal to $\lceil n/3 \rceil$.

Balls of the form x_2 $(x = 1, \ldots, \lceil n/2 \rceil)$ are each in some bin between $\lceil (n+1)/3 \rceil$ and $\lceil n/2 \rceil$ after Step 2. From the analysis of Step 1 for x_2, the r-th bin has three balls $3r - n - 2_2$, $3r - n - 1_2$, and $3r - n_2$. As the ball $3r - n - 2_2$ shifts left once while the ball n_1 moves to the right, the r-th bin will have the three balls $3r - n - 1_2$, $3r - n_2$, and $3r - n + 1_2$ after completing this movement.

Before moving ball n_1, each ball of the form x_3 $(x = 1, \ldots, \lfloor n/2 \rfloor)$ is in the $(n - x)$-th bin, which is not its target bin. These balls each shift left once as the ball n_1 moves to the n-th bin. That is, every ball x_3 $(x = 1, \ldots, \lfloor n/2 \rfloor)$ shifts to the $(n - x - 1)$-th bin.

From the above arguments, the state of the first $n - 1$ bins after moving the ball n_1 to the n-th bin is the same as the state after Step 2 when we apply $Recursive(n-1)$ to Sorting 3-Sets in Bins with $n - 1$ bins. After the second-last execution of the Step 3 loop (when $x = 3$), the state of the first two bins is the same as the state after Step 2 in the solution of Sorting 3-Sets in Bins with two bins. In other words, the state is as follows:

1_1	1_3
1_2	2_2
2_1	2_3
1	2

We can complete the sorting by moving 2_1 to the second bin. □

5.4.3 Analysis of the Number of Swaps

We count the number of swaps for $Recursive(n)$.

Lemma 17 *The number of swaps performed by $Recursive(n)$ is less than $\frac{15}{16}n^2 + 12n$.*

Proof. We will use induction. Let $S(n)$ be the number of swaps performed by $Recursive(n)$. When $n = 2$, $S(2) = 3 < \frac{15}{16}2^2 + 24$.

Suppose that $S(l) < \frac{15}{16}l^2 + 12l$ holds for any $l < n$. We count the number of swaps performed by $Recursive(n)$, where $n \geq 3$.

In Step 1, we need $2i - n - 1$ swaps to move each ball i_3 and i_2 to the i-th bin, so the total number of swaps in Step 1 is at most

$$\sum_{i=n}^{\lceil n/2 \rceil + 1} (2i - n - 1) \times 2 = 2 \cdot \lceil n/2 \rceil \cdot (n - \lceil n/2 \rceil) \leq \frac{1}{2} n^2.$$

Step 2 needs $S(\lceil n/3 \rceil) < \frac{15}{16} (\lceil n/3 \rceil)^2 + 12 \lceil n/3 \rceil$ swaps.

It remains to count the number of swaps in Step 3. We consider the state of each bin. The r-th bin ($r \in \{1, \ldots, \lceil n/3 \rceil\}$) has three balls each labeled $3r - 3 + j_1$ ($j \in \{1, 2, 3\}$.) The total number of swaps needed for these three balls to move to the target bin is at most

$$\sum_{j=1}^{3} (3r - 3 + j - r) = 6r - 3.$$

Therefore, Step 3 needs at most

$$\sum_{r=1}^{\lceil n/3 \rceil} 6r - 3 = 3(\lceil n/3 \rceil)^2$$

swaps. Thus,

$$\begin{aligned} S(n) &< \frac{1}{2} n^2 + \frac{15}{16} (\lceil n/3 \rceil)^2 + 12 \lceil n/3 \rceil + 3(\lceil n/3 \rceil)^2 \\ &< \frac{15}{16} n^2 + 12n, \qquad (\because \lceil n/3 \rceil < n/3 + 1 \text{ and } n \geq 3) \end{aligned}$$

as required. □

Proof of Theorem 8. Directly follows from Lemmas 16 and 17. □

5.4.4 Chapter Summary

In this chapter, we propose two algorithms for sorting k-set in bins. One is the greedy algorithm, which is based on a simple idea. This algorithm shows that some balls need only the smallest number of swaps. The another is the recursive algorithm, which divides an original instance into a sub instants and sorted balls. This algorithm shows that the step for merge does not much swaps. Using these algorithms, we have non-trivial upper bounds.

Chapter 6

Conclusion

In this thesis, we study the lower bounds of branching programs and upper bounds of algorithms solving Sorting k-set Bins Problems. Both of lower bounds and upper bounds are the fundamental point of view for computational complexity. We conjecture that the class $\mathbf{L}$ is not equivalent to the class $\mathbf{P}$. Through a study of branching program, we have interest in branching program satisfiability problems and establish efficient upper bounds for this problems. Some idea comes up with a good upper bound for a mathematical puzzle.

In Chapter 2, we have a new super polynomial lower bound for TEP by introducing read-once restriction to branching programs. Using the key idea of Cook et al., we induce a number of leaf reading state to a number of critical states. We do not count the amount of leaf reading states, but count the amount of "last leaf reading states". This technique does not depend on the height of TEP. The lower bound for $FT_h(k)$ is $\Omega(k^h)$= $\Omega(n^{\log(n)})$ and tight. From this lower bound, our motivation naturally goes to if branching programs are not restricted, lower bound is to be super polynomial or not? This is so big issue on computational complexity.

In Chapter 3, we discuss general branching programs solving TEP. We modify the method Cook et al. showed, and show a new lower bound of branching programs as $\Omega(k^{2d-1}/\log k) = \Omega(n^{\frac{2d-1}{d}}/(\log n)^{\frac{2d+1}{d}})$ for $FT_d^3(k)$. This modification does not change the key ideas in the original method and makes it to be available for TEP with height 3 and d-ary complete tree, where d is any constant. This lower bound is also tight. Therefore this result gives us a further motivation for modification to apply to greater tree evaluation problems.

We also try to modify the original method for height 4 tree evaluation problems. But there are some difficulties and modification is not completed. To overcome these difficulties, we must find a way to divide leaf reading states into disjoint sets. If we do not introduce

any restriction, we can construct a branching program whose critical states are not divided into disjoint sets for distinct k^3 input sets. We seem to need substantially new approaches.

In Chapter 4, we propose a exponentially faster algorithm solving k-IBDD satisfiability problems. This algorithm solves $O(n^c)$ size k-IBDD SAT in $O(2^{n-\omega(\log n)})$ time. This algorithm uses partial assignments and longest common sequences. Because an IBDD has states in some order, a longest common sequence shows the labels which are not fixed. We can solve a polynomial size OBDD SAT with testing reachability. This algorithm would be improved if we can take more "efficient" longest common sequences.

In Chapter 5, we consider the sorting k-set bins problems. This problem is a natural extension of Winkler's "Sorting Pairs in Bins". We propose two algorithms; $Greedy(n, k)$ and $Recursive(n)$. $Greedy(n, k)$ works for any n and k for sorting k-set bins problems and needs $\frac{k+1}{4}n^2 + O(n)$ swaps. $Recursive(n)$ works for $k = 3$ and any n and needs $\frac{15}{16}n^2 + O(n)$ $(= 0.9357n^2 + O(n))$ swaps. So, $Recursive(n)$ are faster than $Greedy(n, 3)$ in the case of $k = 3$. On the other hand, the known lower bound is $(1 - \frac{k-1}{2k^2+k-1})\frac{k+1}{4}n^2 + O(n)$. This means that, if k and n increase, upper bound from $Greedy(n, k)$ asymptotically approaches to the lower bound.

There are two future works for us. One is to improve the lower bound of branching programs solving tree evaluation problems. If we do not introduce any restriction to branching programs, our next target is to analyse height 4 tree evaluation problems. Since there are no greater than k^3 lower bound of branching programs solving any tree evaluation problems, we do not need to analyze a tight lower bound; our next target is a little greater lower bound. If we introduce some restriction to branching programs, it would be "read-k-times" restriction. This is a natural extension of read-once restriction. There exist so many studies about read-k-times restriction, and they would help us.

The other is to improve the upper bound of algorithms solving any branching program satisfiability problems. One result shows a exponentially faster algorithm. Or, there would be an efficient algorithm which solves other restricted branching program satisfiability. These algorithms also contribute to more knowledge in computational complexity.

Bibliography

[1] M. Ajtai, L. Babai, P. Hajnal, J. Komlós, P. Pudlák, V. R odl, E. Szemerédi, and G. Turán. Two lower bounds for branching programs. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 30–38. ACM, 1986.

[2] Vikraman Arvind and Rainer Schuler. The quantum query complexity of 0-1 knapsack and associated claw problems. In *Algorithms and Computation*, pages 168–177. Springer, 2003.

[3] Pedro Berrizbeitia. Sharpening "primes is in p" for a large family of numbers. *Mathematics of computation*, 74(252):2043–2059, 2005.

[4] Beate Bollig, Martin Sauerho, Detlef Sieling, and Ingo Wegener. *On the power of different types of restricted branching programs.* Dekanat Informatik, Univ., 1994.

[5] Miklos Bona and Ryan Flynn. Sorting a permutation by block moves. *arXiv preprint arXiv:0806.2787*, 2008.

[6] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[7] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. A duality between clause width and clause density for sat. In *Computational Complexity, 2006. CCC 2006. Twenty-First Annual IEEE Conference on*, pages 7–pp. IEEE, 2006.

[8] Ruiwen Chen, Valentine Kabanets, Antonina Kolokolova, Ronen Shaltiel, and David Zuckerman. Mining circuit lower bound proofs for meta-algorithms. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 20, page 57, 2013.

[9] S. Cook. An observation on time-storage trade off. *Journal of Computer and System Sciences*, 9(3):308–316, 1974.

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

[11] Daniel W Cranston, I Hal Sudborough, and Douglas B West. Short proofs for cut-and-paste sorting of permutations. *Discrete Mathematics*, 307(22):2866–2870, 2007.

[12] Evgeny Dantsin, Edward A Hirsch, and Alexander Wolpert. Algorithms for sat based on search in hamming balls. In *STACS 2004*, pages 141–151. Springer, 2004.

[13] Evgeny Dantsin, Edward A Hirsch, and Alexander Wolpert. Clause shortening combined with pruning yields a new upper bound for deterministic sat algorithms. In *Algorithms and Complexity*, pages 60–68. Springer, 2006.

[14] Constantinos Daskalakis, Richard M Karp, Elchanan Mossel, Samantha J Riesenfeld, and Elad Verbin. Sorting and selection in posets. *SIAM Journal on Computing*, 40(3):597–622, 2011.

[15] Harry Dweighter. E 2569 in: Elementary problems and solutions. *Amer. Math. Monthly*, 82(1):1010, 1975.

[16] Sergi Elizalde and Peter Winkler. Sorting by placement and shift. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 68–75. Society for Industrial and Applied Mathematics, 2009.

[17] P Erdös and G Szckeres. A combinatorial problem in geometry. In *Classic Papers in Combinatorics*, pages 49–56. Springer, 1987.

[18] Henrik Eriksson, Kimmo Eriksson, Johan Karlander, Lars Svensson, and Johan Wästlund. Sorting a bridge hand. *Discrete Mathematics*, 241(1):289–300, 2001.

[19] Ulrich Faigle and Gy Turán. Sorting and recognition problems for ordered sets. *SIAM Journal on Computing*, 17(1):100–113, 1988.

[20] A. Gál and P. McKenzie M. Koucký and. Incremental branching programs. *Theory of Computing Systems*, 43(2):159–184, 2008.

[21] William H Gates and Christos H Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27(1):47–57, 1979.

[22] Mohammad H Heydari and I Hal Sudborough. On the diameter of the pancake network. *Journal of Algorithms*, 25(1):67–94, 1997.

[23] Edward A Hirsch. Exact algorithms for general cnf sat. *Encyclopedia of Algorithms*, pages 286–289, 2008.

[24] Charles Antony Richard Hoare. Algorithm 63: partition, algorithm 64: quicksort, algorithm 65: find. *Communications of the ACM*, 4(7):321–322, 1961.

[25] Russell Impagliazzo, William Matthews, and Ramamohan Paturi. A satisfiability algorithm for ac 0. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 961–972. SIAM, 2012.

[26] Hiro Ito, Junichi Teruyama, and Yuichi Yoshida. An almost optimal algorithm for winkler ’s sorting pairs in bins. *Progress in Informatics*, 9:3–7, 2012.

[27] Arthur M Jaffe. The millennium grand challenge in mathematics. *Notices of the AMS*, 53(6), 2006.

[28] Jawahar Jain, James Bitner, Magdy S Abadir, Jacob A Abraham, and Donald S Fussell. Indexed bdds: Algorithmic advances in techniques to represent and verify boolean functions. *Computers, IEEE Transactions on*, 46(11):1230–1245, 1997.

[29] S. Jukna and A. Razborov. Neither reading few bits twice nor reading illegally helps much. *Discrete Applied Mathematics*, 85(3):223–238, 1998.

[30] S. Jukna and S. Žák. On uncertainty versus size in branching programs. *Theoretical computer science*, 290(3):1851–1867, 2003.

[31] B. Komarath and J. Sarma. Pebbling, entropy and branching program size lower bounds. In *30th International Symposium on Theoretical Aspects of Computer Science (STACS)*, page 622, 2013.

[32] Meena Mahajan. Polynomial size log depth circuits : Between nc1 and ac1. *Bulletin of the EATCS*, 91:42–56, 2007.

[33] Conrado Martínez and Uwe Rösler. Partial quicksort and quickpartitionsort. In *21st International Meeting on Probabilistic, Combinatorial, and Asymptotic Methods in the Analysis of Algorithms (AofA’10)*, pages 47–60. DMTCS Proceedings, 1991.

[34] W. Masek. *A fast algorithm for the string editing problem and decision graph complexity.* PhD thesis, Massachusetts Institute of Technology, 1976.

[35] William Joseph Masek. *A fast algorithm for the string editing problem and decision graph complexity.* PhD thesis, Massachusetts Institute of Technology, 1976.

[36] È. Nečiporuk. A boolean function. In *Doklady of the Academy of the USSR*, volume 169, pages 765–766, 1998. English translation in Soviet Mathematics Doklady 7:4, pp.999–1000.

[37] M. Paterson and C. Hewitt. Comparative schematology. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 119–127. ACM, 1970.

[38] Pavel Pudlák. Satisfiability algorithms and logic. In *Mathematical Foundations of Computer Science 1998*, pages 129–141. Springer, 1998.

[39] Annett Püttmann. Krawattenproblem. HTML version is available on http://www.springer.com/cda/content/document/ cda_downloaddocument/SAV_Krawattenraetsel_Loesung_Puettmann.

[40] Rahul Santhanam. Fighting perebor: New and improved algorithms for formula and qbf satisfiability. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 183–192. IEEE, 2010.

[41] P. Savický and S. Žák. A read-once lower bound and a (1,+ k)-hierarchy for branching programs. *Theoretical Computer Science*, 238(1):347–362, 2000.

[42] Rainer Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms*, 54(1):40–44, 2005.

[43] Kazuhisa Seto and Suguru Tamaki. A satisfiability algorithm and average-case hardness for formulas over the full binary basis. *computational complexity*, 22(2):245–274, 2013.

[44] J. Simon and M. Szegedy. A new lower bound theorem for read-only-once branching programs and its applications. *Advances in Computational Complexity Theory*, 13:183–193, 1993.

[45] Ivan Hal Sudborough. On the tape complexity of deterministic context-free languages. *Journal of the ACM (JACM)*, 25(3):405–414, 1978.

[46] I. Wegener. On the complexity of branching programs and decision trees for clique functions. *Journal of the ACM (JACM)*, 35(2):461–471, 1988.

[47] D. Wehr. Exact size of smallest minimum-depth deterministic bps solving the tree evaluation problem. Unpublished. http://www.cs.toronto.edu/˜wehr/.

[48] Douglas B. West. http://www.math.illinois.edu/ dwest/regs/sortpair.html.

[49] Ryan Williams. Nonuniform acc circuit lower bounds. *Journal of the ACM (JACM)*, 61(1):2, 2014.

[50] Peter Winkler. Mathematical puzzles: a connoisseur's collection. *AMC*, 10:12, 2004.

[51] S. Žák. An exponential lower bound for one-time-only branching programs. In *Mathematical Foundations of Computer Science 1984*, pages 562–566. Springer, 1984.

List of Publications

K. Iwama and A. Nagao "Read-once branching programs for tree evaluation problems." In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25, pages 409–420. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

A. Nagao, K. Seto, and J. Teruyama. "Efficient algorithms for sorting k-sets in bins." In Sudebkumar Prasant Pal and Kunihiko Sadakane, editors, *WALCOM*, volume 8344 of *Lecture Notes in Computer Science*, pages 225–236. Springer, 2014.

K. Iwama and A. Nagao "Read-once branching programs for tree evaluation problems." The 7th Asian Association for Algorithms and Computation Annual Meeting, Proc. of AAAC 2014, April, 2014 (Hangzhou, China)

K. Iwama and A. Nagao "A Lower Bound of Tree Evaluation Problem with Constant Ary." The 5th Asian Association for Algorithms and Computation Annual Meeting, Proc. of AAAC 2012, April, 2012 (Shanghai, China)