Characterizing Hardness in Parameterized Complexity

by
Tarique M. Islam

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2007

© Tarique M. Islam 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

Tarique Islam

I understand that my thesis may be made electronically available to the public.

Tarique Islam

Abstract

Parameterized complexity theory relaxes the classical notion of tractability and allows to solve some classically hard problems in a reasonably efficient way. However, many problems of interest remain intractable in the context of parameterized complexity. A completeness theory to categorize such problems has been developed based on problems on circuits and MODEL CHECKING problems. Although a basic machine characterization was proposed, it was not explored any further.

We develop a computational view of parameterized complexity theory based on resource-bounded programs that run on alternating random access machines. We develop both natural and normalized machine characterizations for the $W[t]$ and $L[t]$ classes. Based on the new characterizations, we derive the basic completeness results in parameterized complexity theory, from a computational perspective. Unlike the previous cases, our proofs follow the classical approach for showing basic NP-completeness results (Cook's Theorem, in particular). We give new proofs of the Normalization Theorem by showing that (i) the computation of a resource-bounded program on an alternating RAM can be represented by instances of corresponding basic parametric problems, and (ii) the basic parametric problems can be decided by programs respecting the corresponding resource bounds. Many of the fundamental results follow as a consequence of our new proof of the Normalization Theorem. Based on a natural characterization of the $W[t]$ classes, we develop new structural results establishing relationships among the classes in the W-hierarchy, and the $W[t]$ and $L[t]$ classes.

Nontrivial upper-bound beyond the second level of the W-hierarchy is quite uncommon. We make use of the ability to implement natural algorithms to show new upper bounds for several parametric problems. We show that SUBSET SUM, MAXIMAL IRREDUNDANT SET, and REACHABILITY DISTANCE IN VECTOR ADDITION SYSTEMS (PETRI NETS) are in $W[3]$, $W[4]$, and $W[5]$, respectively. In some cases, the new bounds result in new completeness results. We derive new lower bounds based on the normalized programs for the $W[t]$ and $L[t]$ classes. We show that LONGEST COMMON SUBSEQUENCE, with parameter the number of strings, is hard for $L[t]$, $t \geq 1$, and for $W[SAT]$. We also show that PRECEDENCE CONSTRAINED MULTIPROCESSOR SCHEDULING, with parameter the number of processors, is hard for $L[t]$, $t \geq 1$.

Acknowledgments

I would like to thank my supervisor Dr. Jonathan F. Buss who guided me through the entire Ph.D. program with utmost care. His encouragement and experienced advices have played a crucial role in the completion of this thesis. I would like to thank the members of my thesis committee, Dr. Alex Lopez-ortiz, Dr. Barbara S. Chima, Dr. Jianer Chen, and Dr. Naomi Nishimura (in alphabetical order of first name) for their valuable suggestions that greatly helped to improve this thesis. I would like to thank the David R. Cheriton School of Computer Science, University of Waterloo for providing an excellent research environment.

I would like to thank my parents and my sister for their support and wishes.

I would like to thank my beloved wife for her encouragement, support, and patience throughout my Ph.D. program.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The theory of parameterized complexity has provided a useful tool to deal with computational problems that are difficult to solve in the classical context. The theory deals with parametric decision problems. An input to a parametric problem is associated with a parameter. The efficiency of an algorithm deciding a parametric problem is determined by the contribution of the parameter to the runtime. An algorithm is considered efficient if its runtime is bounded above by $f(k)n^c$, where n is the length of the input, f is any fixed function, c is a constant, and k is a parameter associated with the input. Parametric versions of many *NP*-hard problems have been shown to be decidable by algorithms having similar runtime [28].

However, the parametric versions of many classically hard problems seem to remain difficult in the parameterized setting. It is believed that parameterized versions of Clique, Dominating Set, CNF-Satisfiability, and Nondeterministic Turing Machine Acceptance are not decidable by deterministic algorithms in time $f(k)n^c$, for any function f and constant c. Parameterized complexity theory provides a framework to formalize the notion of intractability in a parameterized setting. The theory builds upon a special form of reduction, called *fixed-parameter reduction*, a collection of classes to represent different degrees of intractability, and the notion of hardness and completeness.

Although the concepts of reduction, hardness and completeness in parameterized complexity theory are parallel to those in the theory of *NP*-completeness, the development of the parameterized complexity theory has been quite different than the development of the *NP*-completeness theory. The parameterized complexity theory was developed by Downey, Fellows, Abrahamson and other researchers in early nineties [1, 25, 26]. In the original formulation, the theory centers around a fundamental hierarchy of classes, known as the $W[t]$ classes, for $t > 0$. These

classes were originally characterized in terms of a restricted version of the SATISFIABILITY problem, known as the WEIGHTED SATISFIABILITY (WSAT) problem, on different circuit families. We will present these basic concepts formally in Section 1.1. With the circuit characterization, Downey, Fellows and other co-researchers successfully showed that many natural problems are complete for few of the classes defined. For example, WSAT on CNF circuits with clauses of bounded size characterizes the class of problems having the same parameterized complexity as CLIQUE [25, 28]. Also, WSAT on unrestricted CNF circuits results in a class of parametric problems that have the same parameterized complexity as DOMINATING SET [26, 28]. However, the basic completeness results were established through intricate circuit transformations and through a sequence of complicated reductions [25, 26].

Downey, Fellows, and Regan investigated the logical characterization of the classes of fixed-parameter intractable problems [31]. Their work was motivated by the fact that any problem in NP can be represented in existential second order logic (Fagin's Theorem). A true analogue of the logical characterization of NP is not yet known in the parameterized context. Nonetheless, Downey et al. were able to show that the $W[t]$ classes can be viewed as the closure of MODEL CHECKING problem (defined in Subsection 1.1.2) on different fragments of existential second order logic under fixed-parameter reduction [31]. This initial work on the logical characterization of the $W[t]$ classes were later expanded by Chen, Flum and Grohe [19, 36]. Flum and Grohe [36] investigated the effect of having parameter dependent quantification in the formulas, an open question raised by Downey et al. [31]. This later work introduced two new hierarchies known as the L-hierarchy and the A-hierarchy consisting of the $L[t]$ and the $A[t]$ classes, respectively.

Based on the logical characterization, Chen, Flum and Grohe proposed a computational model for the classes of fixed-parameter intractable problems [19, 36]. The model is an alternating version of the standard random access machines (RAM). They characterized different classes in terms of resource-bounded programs that run on this machine model. However, they did not investigate the usefulness of the computational model further.

One may compare the present state of the parameterized complexity theory with that of the theory of NP-completeness. We have at least three different characterizations of the class NP - (i) in terms of nondeterministic Turing machines, (ii) in terms of SATISFIABILITY problem (Cook's Theorem), and (iii) in terms of existential second-order logic (Fagin's Theorem). In comparison, (ii) and (iii) correspond to the circuit characterization and logical characterization of the classes in the parameterized complexity, respectively. Although (i) stands out in terms of importance, its analogue in the parameterized context is still in its infancy. In this thesis,

we develop a computational view of the theory of fixed-parameter intractability. Our research focuses on developing natural variants of the computational models for classes of fixed-parameter intractable problems, using the variants to classify parametric problems according to their degree of intractability, and studying relationships among different classes.

In order to place the results in proper perspective, we first present the basic concepts in parameterized complexity theory (Section 1.1). We then describe the contributions of this thesis in Section 1.2. We end this chapter by mentioning the organization of the rest of the thesis.

1.1 Background

In this section, we describe the basic definitions of parameterized complexity theory. We also introduce the notation that we will use in the rest of this thesis.

Definition 1.1 *A* parametric problem *takes a pair $\langle x, k \rangle$ as input where x is considered as the main input and k is the parameter. Both x and k are strings on some finite alphabets Σ and Γ respectively.*

For example, the parameterized version of Vertex Cover is defined as follows [25, 28].

Vertex Cover (VC)

Input: A graph G, and an integer k.
Parameter: k.
Question: Does G have a vertex cover of size k?

Parametric versions of Clique and Dominating Set are defined in a similar manner (the definitions are provided in the Appendix). In this thesis, we will assume that the parameter k is a natural number. We will use n to denote the length of x and the notation $p(n)$ and $q(n)$ to represent functions that are polynomial in $|x| = n$. The notation $f(k)$, $g(k)$, and $h(k)$ will be used to represent arbitrary (but fixed) functions that depend on the parameter k only (and not on the main input x). For convenience, we will sometimes omit the arguments of these functions.

Definition 1.2 *Let f be any fixed function and p be a polynomial. Given any input of length n and a parameter k, we refer to an expression of the form $f(k)p(n)$ as a* parametric polynomial. *A parametric problem* Q *is said to be* fixed-parameter tractable *if there exists a deterministic algorithm that can decide whether a given input $\langle x, k\rangle \in$ Q in parametric polynomial number of steps. The class of all fixed-parameter tractable problems is denoted by FPT.*

For example, Vertex Cover is in FPT(Sam Buss, reported in the article [9]). We say that a parametric problem in FPT is decidable by a deterministic algorithm in *parametric polynomial time*. Although the notion of tractability in parameterized complexity theory is somewhat relaxed compared with the tractability in the classical sense, not all parametric problems of interest are known to be fixed-parameter tractable. Clique and Dominating Set are two well-known examples of such problems. A theory of fixed-parameter intractability has been developed to classify the parametric problems that are believed to be hard in the context of parameterized complexity. A parametric version of the polynomial time many-to-one reduction has been defined to deal with hardness and completeness issues.

Definition 1.3 *A parametric problem* Q_1 *is* fixed-parameter reducible *to a parametric problem* Q_2 *if there exists an algorithm R such that the following holds.*

1. *R maps an instance $\langle x, k\rangle$ of Q_1 to an instance $\langle x', k'\rangle$ of Q_2 such that $\langle x, k\rangle \in \mathrm{Q}_1 \Leftrightarrow \langle x', k'\rangle \in \mathrm{Q}_2$.*
2. *There exists a function g such that $k' \leq g(k)$.*
3. *R runs in deterministic* parametric *polynomial time.*

The intractability of parameterized problems is represented by different complexity classes. The fundamental classes of fixed-parameter intractable problems are known as the $W[t]$ classes, $t \geq 1$, forming the W-hierarchy (Downey and Fellows). We will present the characterizations of these classes in subsections 1.1.1, 1.1.2, and 1.1.3. The entire W-hierarchy is contained in the class $W[SAT]$, which in turn is contained in the class $W[P]$. Thus the hierarchy of the fundamental complexity classes in parameterized complexity looks as follows.

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \ldots \subseteq W[t] \subseteq \ldots \subseteq W[SAT] \subseteq W[P]$$

Later, Downey et al. defined an extended version of the $W[t]$ classes [32]. These classes are known as the $W^*[t]$ classes, $t > 0$. By definition, $W^*[t]$ contains $W[t]$,

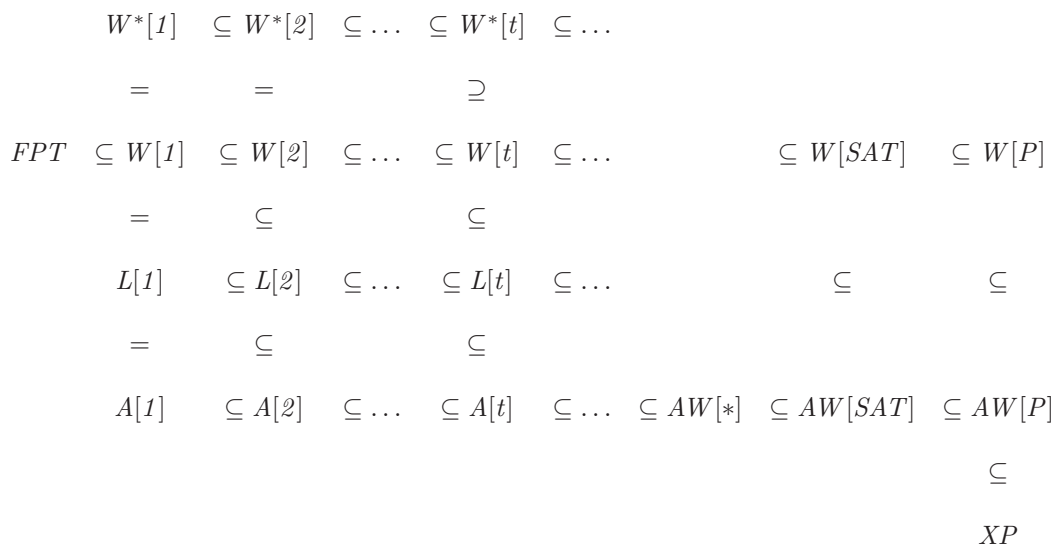


Figure 1.1: Relationships among the classes in parameterized complexity theory

for each $t > 0$. Downey et al. showed [27, 32] that $W^*[t]$ and $W[t]$ are in fact equal for $t = 1, 2$. Whether such equality holds for $t \geq 3$ is still open. However, it is known that $W^*[t] \subseteq W[2t - 2]$ (Flum and Grohe [37], Theorem 8.54).

Recently, Chen, Flum and Grohe [19, 36] developed a new characterization of the $W[t]$ classes, $t > 0$ (Subsection 1.1.2). They considered variants of the model, obtaining two new hierarchies of classes. These hierarchies are known as the L-hierarchy and the A-hierarchy, respectively. The L-hierarchy consists of the $L[t]$ classes, $t > 0$, while the A-hierarchy consists of the $A[t]$ classes, $t > 0$. By definition, $W[t] \subseteq L[t] \subseteq A[t]$, for each $t > 0$. Although, $W[1] = L[1] = A[1]$ [36], it is not known whether equality relation holds for higher levels. In fact, showing inequality at any level beyond the first would imply that $P \neq NP$ [19]. The A-hierarchy is contained in the class $AW[*]$. The relationships among the classes are given in Figure 1.1. The results in this thesis are related to the classes in the W-hierarchy, W^*-hierarchy, the L-hierarchy, and the class $W[SAT]$. We present the characterizations of these classes in detail in the following subsections. For the definitions of the classes $AW[*]$, $AW[SAT]$, $AW[P]$, and XP, the reader is referred to the monograph by Downey and Fellows [28].

The rest of this section discusses the characterization of the classes in different hierarchies in detail.

1.1.1 Circuits and Parametric Classes

The definitions presented in this section are due to Downey and Fellows [25, 26, 28].

In the classical context, *NP* is probably the most important class of (apparently) intractable problems. *NP* can be viewed as the closure of SATISFIABILITY under polynomial-time many-to-one reductions. The original definition of the $W[t]$ classes [25] resembles this characterization of *NP*. A bounded version of the SATISFIABILITY problem plays a central role in defining the fixed-parameter intractable classes. Different classes are obtained by varying the associated circuit complexity.

Definition 1.4 *A circuit consists of* and-*gates,* or-*gates, and* not-*gates where each gate performs the corresponding standard boolean operation. Let $c \geq 2$ be any constant. A gate g is called* large *if the number of inputs to g exceeds c, g is called a* small *gate otherwise. The* weft *of a circuit is the maximum number of large gates in any input-output path. The* depth *of a circuit is the maximum number of gates (small or large) in any input-output path. A circuit is called a* tree circuit *if each gate in the circuit has a single output.*

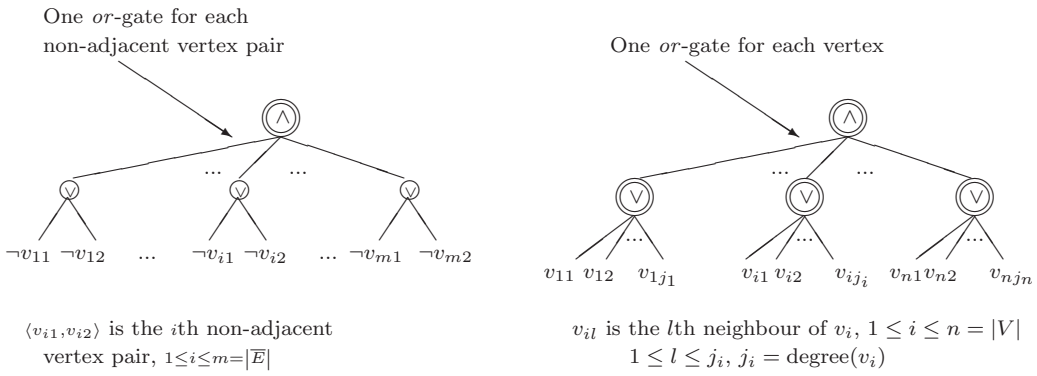
An *assignment* assigns *true/false* values to the input variables. The *weight* of an assignment is the number of *true* variables in the assignment. The WEIGHTED CIRCUIT SATISFIABILITY problem is defined as follows.

Weighted Circuit Satisfiability (WCS)

Input: A circuit C and an integer k.
Parameter: k.
Question: Does C have a weight-k satisfying assignment?

The *weft* of a circuit family determines the degree of parametric intractability of the corresponding WCS problem. The classes in the W-hierarchy are defined as the closure of WCS on restricted circuit families under fixed-parameter reductions.

Definition 1.5 *$W[t]$ is the class of all parametric problems that are fixed-parameter reducible to the* WCS *problem on weft-t, depth-d circuits, where d is a constant and $t \leq d$.*



(a) Clique (b) Dominating Set

Figure 1.2: Let $G = (V, E)$ be a graph and $\overline{E}$ be $(V \times V) \setminus E$. (a) Representation of Clique on G by an instance of WCS on 2-CNF circuits. (b) Representation of Dominating Set on G by an instance of WCS on CNF circuits.

For example, Clique can be fixed-parameter reduced to WCS on 2-CNF circuits (Figure 1.2 (a)). Also, Dominating Set fixed-parameter reduces to WCS on CNF-circuits (Figure 1.2 (b)). Thus Clique and Dominating Set are in $W[1]$ and $W[2]$, respectively.

The classes $W[SAT]$ and $W[P]$ are defined in a similar fashion [25]. The circuit families for $W[SAT]$ and $W[P]$ have no restriction on the depth and weft. However, the circuit family for $W[SAT]$ is restricted to be the family of tree circuits (Definition 1.4) with no bound on depth and weft. As mentioned before, the $W[t]$ classes, $t \geq 1$, form the W-hierarchy. By definition, $W[SAT]$ is a subset of $W[P]$ and $W[SAT]$ contains the entire W-hierarchy.

Downey and Fellows defined a normalized form of circuits, called the t-normalized form, and proved that WCS on t-normalized circuit family is complete for $W[t]$ [25, 28].

Definition 1.6 *A boolean circuit C is said to be* t-normalized *if*

- *C consists of t alternating levels of* and*-gates and* or*-gates with an* and*-gate at the output, and*
- *gates at level i receives inputs from gates at level $i + 1$ only.*

Figure 1.3 illustrates the structure of a t-normalized circuit. A circuit is called *monotone* if all inputs are positive and the circuit contains no *not*-gate. A circuit is called *antimonotone* if all inputs are negated and no *not*-gate appears

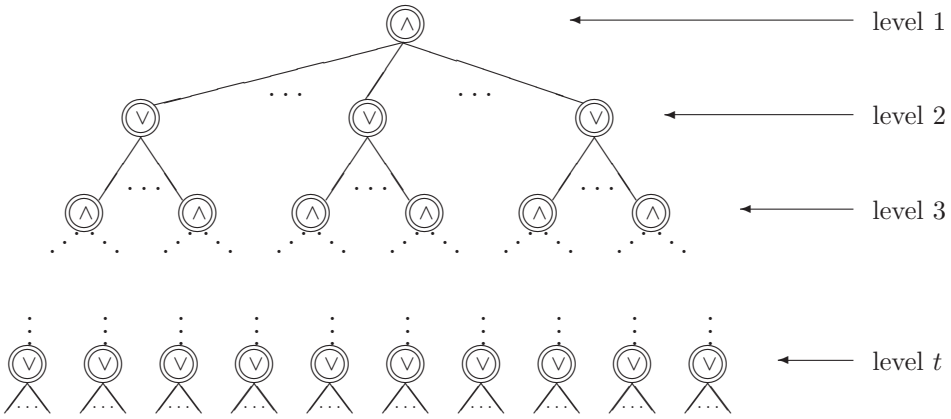


Figure 1.3: The structure of a t-normalized circuit for even t.

elsewhere. The parametric problems Weighted Monotone Circuit Satisfiability, Weighted Anti-monotone Circuit Satisfiability, and Weighted t-normalized Satisfiability, are subsets of Weighted Circuit Satisfiability problem, where the corresponding circuit family is restricted to monotone, antimonotone, and t-normalized circuits, respectively. For example, the circuits constructed for Clique and Dominating Set (Figure 1.2) are antimonotone and monotone, respectively.

The definition of the $W^*[t]$ classes is analogous to that of the $W[t]$-classes (Definition 1.5) with the exception that the corresponding circuits can have depth bounded above by some function of the parameter.

Definition 1.7 *The class $W^*[t]$, $t \geq 1$, consists of all parametric problems that can be fixed-parameter reduced to the* WCS *problem on weft-t, depth-$h(k)$ circuit family, where h is any function.*

No circuit characterization is known yet for the $L[t]$ and $A[t]$ classes. These classes were defined by Flum and Grohe [36] by extending a logical characterization of the $W[t]$ classes which we describe in the next subsection.

1.1.2 Logical Characterization

Descriptive complexity theory provides an alternative measure of complexity of problems. The focus of the theory is to represent computational problems using

logical expressions. The descriptive complexity of a problem is determined by the complexity of the expression required to represent the problem. For many important computational complexity classes C, it has been shown that if two problems belong to C then they have the same descriptive complexity. Details about the contributions of descriptive complexity theory in the classical context can be found in the book by Immerman [39], for example. The first known investigation in the area of logical characterization of classes in parameterized complexity theory was done by Downey, Fellows, and Regan [31]. Flum and Grohe in a later work [36] established a similar characterization for the $W[t]$ classes in a more generalized context. The $L[t]$ and the $A[t]$ classes were obtained by extending the logical characterization of $W[t]$. In the rest of this subsection, we briefly discuss the related concepts.

Three different levels of logical language, namely propositional logic, first-order logic, and existential second order logic, are relevant for the logical characterization of the classes in parameterized complexity theory. Propositional logic is conceptually similar to tree circuits and they both have the same expressive power. The logical characterizations of the classes of fixed-parameter intractable problems are based on first-order logic and existential second-order logic.

As an example, let us consider the Dominating Set problem. The classical version of the problem can be expressed by the second-order expression ψ_{DS}

$$\psi_{\text{DS}} = \exists DS \; \forall v \; \exists w \;\; [\neg \text{Vertex}(v) \vee DS(v) \vee (\text{Edge}(\langle w, v \rangle) \wedge DS(w))] \,,$$

where DS is a relation variable of arity 1, also known as a *monadic relation*. It turns out that expressions having second-order quantifiers over monadic relations are powerful enough to characterize the classes of our interest ($W[t]$, $W[SAT]$, $L[t]$, and $A[t]$) [31, 36].

In the parameterized context, we are interested in solutions of bounded size. There are two ways to achieve the goal. The first approach is to express the problem using second-order existential quantification over relation variables and requiring that the existentially selected relation has the desired size. For example, the parametric Dominating Set problem can be expressed by ψ_{DS} (as in the previous example) and requiring that the monadic relation DS has size k. This leads to the concept of weighted Fagin definability [36] which was originally formulated by Downey et al. [31].

Let φ be a second-order expression with a single free relation variable of a fixed arity. φ weighted Fagin defines a problem WFD_{φ}, as follows.

WFD$_\varphi$

Input: A structure $\mathcal{A}$ and an integer k.
Parameter: k.
Question: Does $\mathcal{A}$ have an interpretation S of the free relation variable in φ such that φ is satisfied and size of S is k?

For example, ψ_{DS} weighted Fagin defines the DOMINATING SET problem when the input structure is restricted to represent a graph.

A second approach is to define an expression for each value the parameter k may take, thereby allowing the length of the expression to be a function of the parameter. In the second approach, k-DOMINATING SET can be expressed by $\psi'_{\text{DS},\kappa}$

$$\psi'_{\text{DS},\kappa} = \exists x_1 \ldots \exists x_k \; \forall v \; \left[\neg \text{Vertex}(v) \vee_{i=1}^{k} (x_i = v \vee \text{Edge}(x_i, v))\right].$$

This approach leads to the parameterized MODEL-CHECKING problem defined by Flum and Grohe [36]. The problem is defined for a class $\mathcal{C}$ of structures and a class $\mathcal{L}$ of expressions.

Model-Checking($\mathcal{C}, \mathcal{L}$)

Input: A structure $A \in \mathcal{C}$ and an expression $\varphi \in \mathcal{L}$.
Parameter: $|\varphi|$.
Question: Does the structure A satisfy φ?

Note the difference between the use of the expression φ in WFD and MODEL-CHECKING. For WFD, the length of φ is fixed and is part of the problem definition whereas for MODEL-CHECKING φ is part of the input and the length of φ is the parameter. For our purpose, the complexity of a logical expression is determined by the number of quantifiers, number of alternations, and the restrictions on the vocabulary.

Definition 1.8 *A first-order expression is in prenex normal form if all the quantifiers appear at the front of the expression. An expression is a Σ_t-expression (respectively, Π_t-expression), for some $t > 0$, if the first quantifier is existential (respectively universal) and there are $t-1$ alternations among quantifiers. A Σ_t-expression is a $\Sigma_{t,u}$-expression if the number of quantifiers in each of the second and subsequent quantifier block is bounded by the constant $u > 0$.*

The $W[t]$ classes are characterized as follows.

Theorem 1.1 *(Downey, Fellows, and Regan [31], Flum and Grohe [36]) Let $\mathcal{F}$ be the class of all structures and $u > 0$ be a constant. Let $\Sigma^R_{t,u}$ be the class of $\Sigma_{t,u}$ expressions on relational vocabularies.*

- $W[t]$ is the closure of Model-Checking$(\mathcal{F}, \Sigma^R_{t,u})$ *under fixed-parameter reduction.*

- $W[t]$ is the closure of WFD_φ *under fixed-parameter reduction, where $\varphi = \exists S\ \varphi'(S)$ such that φ' is a Π_t-expression and S is a relation variable of fixed arity.*

There are two natural ways to extend the logical characterization of $W[t]$ - (i) by allowing unrestricted vocabularies (in particular allowing function symbols in addition to relation symbols) and (ii) allowing parameter bounded quantifiers in the second and subsequent quantification blocks. The origin of the L-hierarchy and the A-hierarchy lie in extensions (i) and (ii) respectively.

Theorem 1.2 *(Flum and Grohe [36]) Let $\mathcal{F}$ be the class of all structures and $u > 0$ be a constant.*

- *$L[t]$ is the closure of* Model-Checking$(\mathcal{F}, \Sigma_{t,u})$ *under fixed-parameter reduction.*
- *$A[t]$ is the closure of* Model-Checking$(\mathcal{F}, \Sigma^R_t)$ *under fixed-parameter reduction.*

In some literature, the $L[t]$ classes have been referred to as $W^{\mathrm{func}}[t]$ classes [37]. By definition, $W[1] = A[1]$. Flum and Grohe showed that $L[t] \subseteq A[t]$, $t \geq 1$ [19]. Chen, Flum, and Grohe [19, 20] translated the Model-Checking characterizations into computational characterizations for the fixed-parameter intractable classes. We describe the related ideas in the next section.

1.1.3 Alternating Random Access Machines and Parametric Classes

As mentioned in previous subsections, a class C in parameterized complexity is associated with a central problem Q_C. A problem Q is included in C if (Type1) an instance of Q can be represented as an instance of the central problem Q_C

or (Type2) Q is fixed-parameter reducible to Q_C. This is in contrast with the classical complexity theory where most interesting classes are defined based on resource-bounded computation on some computational model. Cai et al. proposed a guess-and-check model, also known as the GC model, and tried to develop a computational characterization of the $W[t]$ classes based on the GC model [12]. Although the GC model characterized the Type1 problems, the GC model could not deal with the Type2 problems. The limitation was overcome in the computational model proposed by Chen, Flum, and Grohe [19, 36]. The key idea was to incorporate a deterministic phase at the beginning of the GC model. The resulting model can be viewed as a prepare-guess-and-check (PGC) model [11].

Chen, Flum, and Grohe developed the machine characterization of the $W[t]$ classes based on the Parameterized Model Checking problem [19, 20, 36]. The underlying machine model is known as *WRAM* and is an alternating version of the standard random access machine (*RAM*) [40]. They defined another model, known as *ARAM*, as an extended version of *WRAM*. Also, they characterized the classes in the L-hierarchy and the A-hierarchy, respectively, based on computations on an *ARAM*.

Both WRAM and ARAM have an unbounded number of *standard registers* $r_1, r_2, \ldots$ and so on, and an unbounded number of *guess registers* $g_1, g_2, \ldots$ and so on. A finite set of operations are available to the programs that run on these models. A computation of a program terminates with a special halting instruction. The computation is *accepting* if and only if the 0th standard register r_0 is zero. The distinction between WRAM and ARAM lies in the kind of operations that can be performed on the registers. Before specifying the operations, we describe some concepts related to nondeterminism on these models. The concepts are similar to those for the standard alternating Turing machines [18] and were adapted for the WRAM and ARAM models by Chen, Flum and Grohe [19, 20, 36].

Definition 1.9 *An operation* op *on a* WRAM *or an* ARAM *is* nondeterministic *if* op *can assign any of more than one values from a given range to a register. A nondeterministic operation results in multiple* computation branches*, one branch for each value in the associated range. The nondeterministic operations are of two kinds, (i)* existential *and (ii)* universal*. The computation starting with an existential (respectively a universal) operation* op *accepts if and only if at least one (respectively all) of the computation branches resulting from* op *accepts.*

Definition 1.10 *Let* op_1 *and* op_2 *be two nondeterministic operations in a computation such that no other nondeterministic operation is performed in between them. If* op_1 *and* op_2 *are of different kinds, they constitute an* alternation*.*

Definition 1.11 *(Chen et al. [19, 20]) Any operation (including the ones defined below) can be performed on the registers on an* ARAM*. The operations on a* WRAM *are restricted to the following.*

WOP1: Any deterministic operation that uses the standard registers only.

WOP2: EXISTS $\uparrow j$ *: Existentially guess a value* $\leq r_0$ *and store the value in guess register* g_{r_j}*.*

WOP3: FORALL $\uparrow j$ *: Universally guess a value* $\leq r_0$ *and store the value in guess register* g_{r_j}*.*

WOP4: JEQUAL $i\ j\ c$ *: If* $g_{r_i} = g_{r_j}$ *then jump to the instruction labelled* c*.*

WOP5: JZERO $i\ j\ c$ *: If* $r_{\langle g_{r_i}, g_{r_j}\rangle} = 0$ *then jump to the instruction labelled* c*. Here* $\langle \cdot\,, \cdot \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$*, is any (reasonable) encoding of pair of values such that* $\langle 0, 0\rangle$ *maps to* 0*.*

We refer to the operations WOP1 to WOP5 as the W-operations. The classes of fixed-parameter intractability are characterized by different subclasses of a special class of programs, called AW-programs.

Definition 1.12 *(Chen et al. [19, 20]) Let* f *and* h *be fixed functions and* $c > 0$ *be any constant. A program running on a* WRAM *or an* ARAM *is called an* AW*-program if, on any input* $\langle x, k\rangle$ *with* $|x| = n$*, any computation branch of the program satisfies the following conditions.*

AW1: There are at most $f(k)n^c$ *computation steps.*

AW2: The computation does not store any value greater than $f(k)n^c$ *in any register at any time.*

AW3: The number of nondeterministic (existential and universal) guesses is at most $h(k)$*.*

$W[P]$ is characterized by AW-programs that run on an ARAM and do not make any universal guesses.

Theorem 1.3 *(Chen et al. [19, 20]) A parametric problem* Q *is in* $W[P]$ *if and only if there exists an* AW*-program* R *to decide* Q *such that* R *runs on an* ARAM *and all nondeterministic operations in any computation path of* R *are existential.*

No machine characterization for $W[SAT]$ is known yet. The classes in the different hierarchies are characterized by imposing additional restrictions on the AW-programs.

Definition 1.13 *An AW-program R_t running on a* WRAM *or an* ARAM *is a t-alternating program if*

T1: there are at most $(t-1)$ alternations in any computation branch of R_t and the first nondeterministic step is existential.

A t-alternating program, running on an ARAM *is an $A[t]$-program if*

A1: all the nondeterministic steps are among the last $h(k)$ steps in any computation branch and the first nondeterministic step is existential.

Definition 1.14 *Let $u > 0$ be a constant. A t-alternating program $R_{t,u}$ is a (t,u)-alternating program if*

TU1: the number of nondeterministic guess steps in each of the second and subsequent levels of alternation is at most u.

An $L[t]$-program is a (t,u)-alternating program running on an ARAM *such that*

L1: in any computation branch, all the nondeterministic operations are among the last $h(k)$ steps, for some function h.

A $W[t]$-program satisfies the constraints for an $L[t]$-program but runs on a WRAM*. Thus a $W[t]$-program is an $L[t]$-program that*

W1: can perform only the W-operations (Definition 1.11) in any computation.

Theorem 1.4 *(Chen, Flum, and Grohe [19, 20]) A parameterized problem* Q *is in $W[t]$ (respectively, $L[t]$ or $A[t]$), $t \geq 1$ if and only if* Q *can be decided by a $W[t]$-program (respectively an $L[t]$-program or an $A[t]$-program).*

Feature	$W[t]$	$L[t]$	$A[t]$
Computation model	WRAM	ARAM	ARAM
Access to guess registers	indirect	direct	direct
Number of guesses in each of second and subsequent levels of alternation	constant	constant	$h(k)$

Table 1.1: Comparison of resource bounds for different hierarchies

Table 1.1 presents a comparative summary of the resource bounds that distinguish different kinds of programs.

At this stage, we would like to point out the correspondence between the Model Checking characterization and the computational characterization. The exists and forall operations correspond to universal and existential quantification in Model Checking. The jzero test verifies whether the values in g_{r_i} and g_{r_j} satisfy a relation. The constraints $AW3$, $T1$, and $TU1$ follow from the constraints on the quantifiers in the corresponding Model Checking problems. The constraint $W1$ corresponds to the fact that the vocabulary for $W[t]$ is relational. Finally constraints $A1$ and $L1$ are motivated by the fact that the length of the expression φ in Model Checking is a function of the parameter.

We illustrate the use of the WRAM operations by constructing an algorithm for Dominating Set (Algorithm 1.1). The $WRAM$ code for important parts are given in comments. The algorithm uses the Jzero test to determine whether two existentially guessed vertices are adjacent. The adjacency matrix is constructed in the appropriate registers in the preprocessing phase to ensure that test outcomes are as desired.

1.2 Contributions of the Thesis

This thesis develops a purely computational view of parameterized complexity theory. The work relates to the PGC model with parametric-polynomial-sized checking phase. This includes the $W[t]$ classes, the $L[t]$ classes, and the class $W[SAT]$.

A major contribution of this thesis is to develop a natural computational model for the $W[t]$-classes, $t \geq 1$, constituting the W-hierarchy. The new model allows one to implement algorithms in a natural way. For example, we show that Subset Sum is in $W[3]$ and Maximal Irredundant Set is in $W[4]$. Although, the algorithms we use in both cases were known before, the best known upper bound in

Algorithm 1.1: Algorithm to decide whether a graph has a dominating set of a given size.

DS(a graph $G = (V, E)$, an integer k)

Construct the adjacency matrix for the input graph such that $r_{\langle i,j \rangle} = 1$ if and only if the ith and jth vertices are adjacent or $i = j$, $1 \leq i, j \leq |V|$.

Existentially guess the indices of k vertices. Let the guessed indices be $i_1, \ldots i_k$.

```
/* r_0 ← |V|                                   */
/* For all i, 1 ≤ i ≤ k                        */
/*     r_1 ← i                                 */
/*     Exists ↑ 1                              */
```

Universally guess the index of a vertex.

```
/* r_0 ← |V|                                   */
/* r_1 ← k + 1                                 */
/* Forall ↑ 1                                  */
```

for $p = 1$ *to* k **do**

```
/* NextVertex:                                 */
```

if *vertices* i_p *and* i_q *are adjacent or same* **then**
    Accept in this branch
end

```
/* r_1 ← p                                     */
/* r_2 ← q                                     */
/* Jzero 1 2 NextVertex                        */
/* Halt and Accept                             */
```

end
Reject
End DS

both cases was $W[P]$. We also construct algorithms to show new upper bounds for a number of parametric problems including Reachability Distance in Petri Nets, Bounded DFA Intersection, and Weighted Integer Programming (Definitions can be found in the appendix). Some of these upper bounds are optimal.

The basic completeness results in parameterized complexity theory were originally derived in the context of circuit satisfiability [25, 26]. We give new and much simpler proofs of the basic results from a computational perspective. Our new proofs resemble the development of the theory of *NP*-completeness. In the classical context, the *NP*-completeness of SAT (Cook's Theorem) is established by (i) constructing a nondeterministic polynomial-time algorithm to decide SAT, and (ii) constructing a generic reduction from computations on a nondeterministic Turing machine to SAT. In the parameterized context, we show that (i) an appropriately parameterized version of SAT can be decided by a program running on the

new model, and (ii) the computation of a resource-bounded program can be represented by an instance of parameterized version of SAT. The basic completeness results for different normalized forms of circuits are obtained as direct consequences of our new proof.

We show some new structural results based on the new characterizations. In some cases, these results hint on the computational features that actually distinguish the associated classes. We analyze the relationship between $W[t]$ and $L[t]$ classes for $t \geq 2$ and identify subclasses of $L[t]$ that are contained in $W[t]$. We identify the key computational feature that differentiates the $W[t]$ and $L[t]$ classes. Our results indicate that the difference between programs for $W[t]$ and $L[t]$, in terms of computational power, is less than what it appears to be from their definitions. It is known that monotone-$W[2t+1]$ is contained in $W[2t]$ while antimonotone-$W[2t+2]$ is contained in $W[2t+1]$, for all $t \geq 0$. We use the *extended characterization* to identify larger subclasses of $W[t+1]$ that are contained in $W[t]$. Most of our results have the nice property that they can be extended to circuits of parametric polynomial size.

Establishing lower bounds for parameterized problems is an important area of research in parameterized complexity theory. In addition to providing evidence that the problem is unlikely to be fixed-parameter tractable, the hardness results suggest that fully polynomial time approximation schemes may not exist for the corresponding optimization problems in the classical context. Our contribution in this area is the development of normalized computational models for the $W[t]$ and $L[t]$ classes. The equivalence between the normalized and natural computation models can be viewed as the computational analogue of the Normalization Theorem [25, 26]. Unlike the theory of *NP*-completeness, the defining problems and a few variants of them are the only problems known to be complete for classes beyond $W[2]$. Thus proving lower bounds often involves construction of generic reductions. The normalized models facilitate the construction of such generic reductions. In addition, lower bound proofs for $W[t]$-classes can sometimes be extended easily for the $L[t]$-classes if the proof is based on computational models. Our proof that Longest Common Subsequence parameterized by the number of strings is $L[t]$-hard for all $t > 0$, is a nice example of such extension technique. Since $L[t]$-classes do not yet have any circuit characterization, lower bound proofs based on circuits cannot be extended in a similar manner. Although reduction from natural problems can be used to show lower bounds for $W[1]$ and $W[2]$, a generic reduction may help improving the bound significantly. We show that the Precedence-Constrained Multiprocessor Scheduling, parameterized by the number of processors, is $L[t]$-hard for all $t \geq 1$. The problem was known to be $W[2]$-hard and the hardness

proof was based on a reduction from DOMINATING SET. Since no natural extension of DOMINATING SET is known for higher classes, the hardness proof cannot be extended in any obvious way to derive the $L[t]$-hardness results.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. The new characterizations of fixed-parameter intractable classes appear in Chapter 2. Chapter 3 presents the simplified proofs of some fundamental results in parameterized complexity theory. Some new structural results are established in Chapter 4. We present some new hardness and membership results in Chapter 5. Chapter 6 analyzes the degree of intractability of various problems on finite state machines. Finally we conclude with a sketch of our future research plan in Chapter 7. The definitions of all problems discussed in this thesis can be found in the Appendix.

Chapter 2

New Variants of the Computation Model

In this chapter, we introduce new ARAM characterizations of the classes in the W-hierarchy and the L-hierarchy. As noted in Subsection 1.1.3, the computational characterizations proposed by Flum and Grohe closely follow the features of the corresponding Model Checking problems. Natural implementation of algorithms by $W[t]$-programs often becomes technically involved because of the associated constraints. We will see a representative example in Section 2.3. Such limitations motivated us to develop a more natural variant of the $W[t]$-programs. The $L[t]$-programs do not have many of the limitations of the $W[t]$-programs. We further extend the $L[t]$-programs to develop a new variant that can implement algorithms in a more natural way.

Simulating computations on one model by another is an important and frequently used technique in complexity theory. Also, constructing generic reductions often involves analyzing fragments of possible computations of a machine on a given input (Cook's Theorem, for example). We will use similar techniques for the $W[t]$-programs and $L[t]$-programs in order to establish lower bound and structural results. The tasks become easier if the computational model to start with is as restricted as possible. These observations have motivated us to derive more restricted variants of the $W[t]$-programs and the $L[t]$-programs.

For all variants, the number and pattern of alternations in the programs are kept the same. Thus, the programs characterizing the level t in the W-hierarchy or the L-hierarchy are allowed to make at most t alternations in any computation branch, the first nondeterministic step being existential. The number of existential steps in the first level of alternation is bounded by some function of the parameter

while the total number of the remaining nondeterministic steps is bounded by some constant. The new ARAM characterizations differ from the corresponding original characterizations in terms of the bounds on the resources available to the computation that follows the first set of existential guess steps. We use the following terminology in order to distinguish among the variants. We either use no prefix or use the prefix *basic* for the original characterizations. The characterization with a relaxed set of constraints (relative to the original) is identified by the prefix *extended*. The characterization with further restrictions added to the original set of constraints is identified by the prefix *normalized*.

In one direction, we allow more resources (e.g. number of steps) for the computation that follows the first set of existential steps, to obtain the *extended* characterizations. In the other direction, we introduce additional restrictions on the original models to obtain the *normalized* characterizations. While modifying the resource bounds, we must ensure that the classes resulting from the new characterizations remain the same as the corresponding original class. The resources to alter and the extent to which the corresponding bounds can be varied are the topic of this chapter.

2.1 A New View of the Alternating RAM (ARAM)

For the purpose of developing the new characterizations, we view the ARAM model in a different way than it was originally described [19, 20]. An ARAM has an unbounded number of standard registers and guess registers, as described in Subsection 1.1.3. Any operation can be performed on the standard registers (including the ones defined below for the guess registers). The operations on the guess registers are limited to EXISTS, FORALL, JEQUAL, JZERO, and a newly introduced ASSIGN operation which is defined below. We extend the syntax of these operations so that the operations may use standard registers and constants as operands. In addition to the original parameters (Definition 1.11), the operations take as parameter a *flag* that indicates the types of the operands. A flag F is set to be $\mathrm{F}_{\mathrm{const}}$, $\mathrm{F}_{\mathrm{standard}}$, or $\mathrm{F}_{\mathrm{guess}}$ to specify that the corresponding operand is a constant, a standard register, or a guess register, respectively. Given a parameter i and a flag $\mathrm{F_i}$ for i, we define val(i) as follows.

$$\mathrm{val}(i) = \begin{cases} i, & \mathrm{F_i \text{ is } F_{const}} \\ r_{r_i}, & \mathrm{F_i \text{ is } F_{standard}} \\ g_{r_i}, & \mathrm{F_i \text{ is } F_{guess}} \end{cases}$$

The details of the syntax of the modified operations are given below.

EXISTS $\uparrow j$ F : Existentially guess a value less than or equal to the value in r_0 and store the value in r_jth standard or guess register as indicated by flag F. Here, $\mathrm{F} \in \{\mathrm{F}_{\mathrm{standard}}, \mathrm{F}_{\mathrm{guess}}\}$.

FORALL $\uparrow j$ F : Universally guess a value less than or equal to the value in r_0 and store the value in r_jth standard or guess register as indicated by flag F. Here, $\mathrm{F} \in \{\mathrm{F}_{\mathrm{standard}}, \mathrm{F}_{\mathrm{guess}}\}$.

JEQUAL $i\ j\ c\ \mathrm{F}_1\,\mathrm{F}_2$: If $\mathrm{val}(i) = \mathrm{val}(j)$ then jump to the instruction labelled c. Here $\mathrm{F}_1, \mathrm{F}_2 \in \{\mathrm{F}_{\mathrm{const}}, \mathrm{F}_{\mathrm{standard}}, \mathrm{F}_{\mathrm{guess}}\}$.

JZERO $i\ j\ c\ \mathrm{F}_1\,\mathrm{F}_2$: If $r_{\langle \mathrm{val}(i), \mathrm{val}(j)\rangle} = 0$ then jump to the instruction labelled c. Here $\langle \cdot\,, \cdot \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, is any (reasonable) encoding of pair of values such that $\langle 0, 0\rangle$ maps to 0. Also, $\mathrm{F}_1, \mathrm{F}_2 \in \{\mathrm{F}_{\mathrm{const}}, \mathrm{F}_{\mathrm{standard}}, \mathrm{F}_{\mathrm{guess}}\}$.

ASSIGN $i\ j\ l\ \mathrm{F}_1\,\mathrm{F}_2\,\mathrm{F}_3$: Assign the value of the standard register $r_{\langle \mathrm{val}(i), \mathrm{val}(j)\rangle}$ to the r_lth standard or guess register as indicated by F_3. Here $\mathrm{F}_1, \mathrm{F}_2 \in \{\mathrm{F}_{\mathrm{const}}, \mathrm{F}_{\mathrm{standard}}, \mathrm{F}_{\mathrm{guess}}\}$ and $\mathrm{F}_3 \in \{\mathrm{F}_{\mathrm{standard}}, \mathrm{F}_{\mathrm{guess}}\}$.

Any test on a constant number of registers such that the test outcome causes the computation to deterministically follow one of two instructions without altering the content of any register.

For example, a program running on an ARAM can existentially assign a value to guess register g_{r_i} by executing the operation EXISTS $\uparrow i\ \mathrm{F}_{\mathrm{guess}}$. Likewise, a program can test whether the value in guess register g_{r_j} is equal to some constant d by executing the operation JEQUAL $j\ d\ c\ \mathrm{F}_{\mathrm{guess}}\,\mathrm{F}_{\mathrm{const}}$. The syntax for the W-operations (Definition 1.11) in the original formulation is modified as follows for the new model. Note that the semantics of the operations WOP1 to WOP5 in Definition 2.1 remain the same as those in the original definition (Definition 1.11).

Definition 2.1 *An operation on the new* ARAM *model is a W-operation if the operation is one of the following.*

WOP1: Any deterministic operation that uses the standard registers only.

WOP2: EXISTS *with a guess register as target (i.e.* F *is* $\mathrm{F}_{\mathrm{guess}}$*).*

WOP3: FORALL *with a guess register as target (i.e.* F *is* $\mathrm{F}_{\mathrm{guess}}$*).*

WOP4: JEQUAL *with guess registers as operands (i.e. both* F_1 *and* F_2 *are* $\mathrm{F}_{\mathrm{guess}}$*).*

WOP5: JZERO *with guess registers as operands* (*i.e. both* F_1 *and* F_2 *are* $\mathrm{F}_{\mathrm{guess}}$).

From now on we use ARAM to refer to the modified version of the alternating RAM (as defined above). The definitions of the basic variants of the programs (defined in Subsection 1.1.3) essentially remain the same for the modified ARAM. We present them again for completeness.

Definition 2.2 *(Chen et al. [19, 20]) Let f and h be fixed functions and $c > 0$ be any constant. A program running on a* WRAM *or an* ARAM *is called an* AW*-program if, on any input $\langle x, k\rangle$ with $|x| = n$, any computation branch of the program satisfies the following conditions.*

AW1: There are at most $f(k)n^c$ computation steps.

AW2: The computation does not store any value greater than $f(k)n^c$ in any register at any time.

AW3: The number of nondeterministic (existential and universal) guesses is at most $h(k)$.

Definition 2.3 *An AW-program R_t running on a* WRAM *or an* ARAM *is a t-alternating program if*

T1: there are at most $(t-1)$ alternations in any computation branch of R_t and the first nondeterministic step is existential.

A t-alternating program $R_{t,u}$ is a (t,u)-alternating program if

TU1: the number of nondeterministic guess steps in each of the second and subsequent levels of alternation is at most u, for some constant $u > 0$.

All variants of $W[t]$-programs and $L[t]$-programs discussed in this thesis are restricted (t,u)-alternating programs that run on an ARAM.

Definition 2.4 (***Basic*** *$L[t]$**-program***) *An $L[t]$-program is a (t,u)-alternating program running on an* ARAM *such that*

BL1: in any computation branch, all the nondeterministic operations are among the last $h(k)$ steps, for some function h.

Note that, in the new formulation, a basic $L[t]$-program is allowed to assign nondeterministically guessed values to standard registers.

Definition 2.5 (***Basic*** $W[t]$***-program***) *A basic $L[t]$-program R is a basic $W[t]$-program if*

BW1: R performs W-operations (Definition 2.1) only.

The following easy lemma states that the basic variants on the modified ARAM model are equivalent to the corresponding original variants.

Lemma 2.1 *1. The characterizations of the basic $L[t]$-program presented in Definition 1.14 and Definition 2.4 are equivalent.*

2. The characterizations of the basic $W[t]$-program presented in Definition 1.14 and Definition 2.5 are equivalent.

PROOF. We prove the equivalence by showing that each constraint in the new formulation has a counterpart in the original formulation and vice versa. All new variants are (t, u)-alternating programs and hence satisfy the constraints $AW1$, $AW2$, $AW3$, $T1$, and $TU1$. The constraints $L1$ and $BL1$ are same. For $W[t]$-programs, Constraints $W1$ and $BW1$ indicate that both variants are allowed to perform the same set of operations.

It remains to show that any operation on the new variant of $L[t]$-programs can be simulated on the original variant of $L[t]$-programs in $O(1)$ time, and vice versa. Let R_{ORIGINAL} and R_{NEW} be $L[t]$-programs that satisfy the constraints in Definition 1.14 and 2.4, respectively. R_{NEW} can store the nondeterministically guessed values in standard registers which allows R_{NEW} to perform any operation (including JEQUAL and JZERO tests) on the nondeterministically guessed values. Thus, R_{NEW} can simulate any operation of R_{ORIGINAL} directly.

Except for the ASSIGN operations, simulation of operations on R_{NEW} by operations on R_{ORIGINAL} can be done directly. This is because, R_{ORIGINAL} can perform any operation (except ASSIGN) on the guess registers as well as on the standard registers. R_{ORIGINAL} can simulate ASSIGN i j l on R_{NEW} by computing the index $u = \langle \text{val}(i), \text{val}(j) \rangle$ and then assigning the value in r_u to g_{r_l}. ∎

In the next section, we define some computational properties of the basic $W[t]$-programs and basic $L[t]$-programs. The new variants will be obtained by relaxing or restricting some of these features.

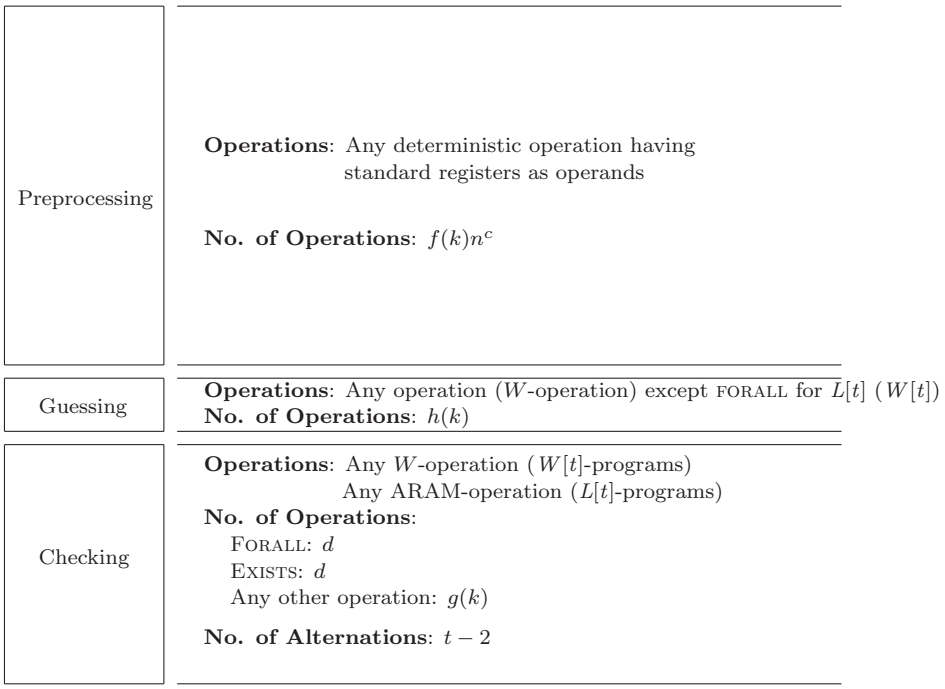


Figure 2.1: Different phases of computation of a basic $W[t]$-program and a basic $L[t]$-program.

2.2 Computational Features of the Basic Programs

The computational features defined in this section will be important in the remaining part of the thesis.

Definitions 2.4 and 2.5 specify the constraints a program running on an ARAM must satisfy in order to be a basic $L[t]$-program and a basic $W[t]$-program, respectively. Conceptually a branch of computation of a basic program (both $W[t]$ and $L[t]$) has three phases, which we define below (Definition 2.6). We would like to stress that computations on all variants of the programs in this thesis will have these three conceptual phases. The term PGC model originates from this conceptual partition of the computation.

Definition 2.6 (***Computational Phases***) *The* preprocessing phase *of a computation branch consists of all deterministic steps that precede the first existential guess step. The* guessing phase *starts with the first existential guess step and consists of all computational steps that precede the first universal guess step in this*

computation branch. The checking phase *starts with the first universal guess step and consists of all subsequent computational steps.*

The conceptual partition of the computation in the PGC model is shown in Figure 2.1. In a typical computation, a basic $W[t]$-program or a basic $L[t]$-program existentially guesses a solution in the guess phase and then checks the validity of the guessed solution in the checking phase. The program may precompute necessary results in the preprocessing phase and then use the precomputed results during the verification process in the checking phase. Algorithm 1.1 is an example of such computations.

All programs discussed in this thesis are halting programs and hence perform a finite number of operations in any computation on any input. The state of a program at any given point of computation on any input can be represented by the values in the relevant registers and the instruction to be executed.

Definition 2.7 (***Configuration***) *Let l be the maximum number of steps any computation branch may perform. The* configuration *of a program at step s in a computation on a given input is represented as $\langle \mathtt{pc}, v_{r_0}, \ldots, v_{r_l}, v_{g_1}, \ldots, v_{g_l} \rangle$, where* pc *is the value of the program counter at time s, v_{r_i} and v_{g_i} are the values stored in the ith standard register and the ith guess register, respectively, before step s is performed.*

We define the following notation for convenience.

- The program counter of α is denoted by $\mathrm{PC}(\alpha)$.
- The operation associated with α is denoted by $\mathrm{OP}(\alpha)$.
- The result of $\mathrm{OP}(\alpha)$ is denoted by $\mathrm{R}(\alpha)$.
- The value specified in α for the ith standard register is denoted by $\mathrm{ST}(\alpha, i)$.
- The value specified in α for the ith guess register is denoted by $\mathrm{GS}(\alpha, i)$.

The possible computations of a program R on a given input $\langle x, k \rangle$ can be represented by a tree of configurations. The resulting tree is called the *computation tree* of R on $\langle x, k \rangle$ [37]. The structure of a configuration tree is shown in Figure 2.2. The concepts of configurations and computation tree are similar to those defined for standard alternating Turing machines [18].

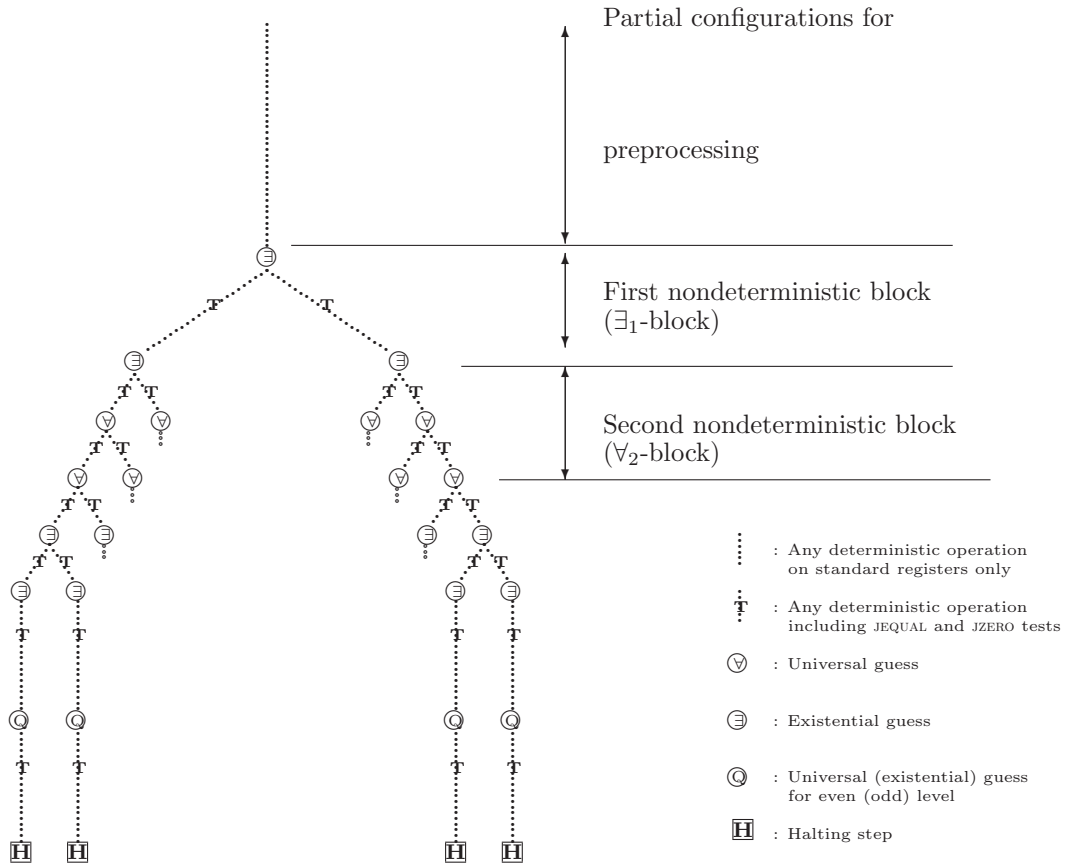


Figure 2.2: Structure of a computation tree of a basic $W[t]$-program or a basic $L[t]$-program. The deterministic configurations following the first existential step may correspond to any deterministic ARAM operation for an $L[t]$-program but are restricted to correspond to W-operations for a $W[t]$-program.

Definition 2.8 (***Computation Tree***) *Consider the computation of a program R on a given input $\langle x, k\rangle$. The computation tree associated with the computation is a tree of configurations such that the following properties hold.*

- *The root of the computation tree is the starting configuration of R on $\langle x, k\rangle$.*
- *Let α be a non-halting deterministic configuration. Also let β be the unique configuration reachable from α in a single step. If α is included in the computation tree then α has β as the only child.*
- *Let α be a nondeterministic configuration. Also let $\{\beta_1, \ldots, \beta_l\}$ be the set of*

configurations that may be reached from α in a single (nondeterministic) step. If α is included in the computation tree, then α has β_i as a child, for each $i, 1 \leq i \leq l$.

- *A halting configuration α in the computation tree must be a leaf.*

In addition to nondeterministic branching, the basic programs use the test operations to perform deterministic branching. The outcomes of any branching operation in the (deterministic) preprocessing phase are fixed for a given input. These test outcomes can be computed by a standard simulation of the preprocessing phase in deterministic parametric polynomial time. Consequently, we are interested in only those tests that appear after the first existential step. Note that a configuration α_{test}, corresponding to a test (JEQUAL or JZERO), has a single child in the computation tree even though the test represents a binary branching. The reason is that the test outcome is unique once the values of the operands are fixed (i.e. the test operations are deterministic). Since a computation branch in the computation tree represents a fixed value for each nondeterministic guess encountered so far, a singe configuration can be reached from α_{test} based on the unique outcome of the associated test on the fixed values. Later we will introduce the partial computation trees (Definition 2.15) which are defined on the assumption that the nondeterministic values from the guess phase are unknown. Under certain conditions, the test configurations result in binary branching in such partial computation trees. We will shortly discuss the partial computation trees in more detail.

We now define some terminology. Let us consider the computation of a basic $W[t]$-program (or a basic $L[t]$-program) R on some input $\langle x, k \rangle$.

Definition 2.9 *A configuration is a nondeterministic configuration if the associated operation is a nondeterministic guess operation. A nondeterministic configuration is existential (respectively universal) if the associated nondeterministic operation is existential (respectively universal). Let* conf_1 *and* conf_2 *be two nondeterministic configurations in some branch in the computation tree such that* conf_1 *is an ancestor of* conf_2 *and no other nondeterministic configuration appears in between* conf_1 *and* conf_2*. We say that* conf_1 precedes conf_2 *and* conf_2 follows conf_1*.*

Definition 2.10 *The nondeterministic configuration corresponding to the first existential operation is at alternation level* 1*. Let* conf_1 *and* conf_2 *be two nondeterministic configurations such that* conf_1 *precedes* conf_2*. If* conf_1 *and* conf_2 *are of the same kind then the configurations are at the same level of alternation. Otherwise, the level of alternation of* conf_2 *is one higher than that of* conf_1*. A nondeterministic*

configuration is a terminal *nondeterministic configuration if it is the last nondeterministic configuration in its level of alternation in some computation branch.*

Definition 2.11 *Let* conf_{i-1} *and* conf_i *be two nondeterministic configurations such that* conf_{i-1} *is an ancestor of* conf_i *in the computation tree. Also let* conf_{i-1} *and* conf_i *be the terminal nondeterministic configurations at alternation levels* $i-1$ *and* i*, respectively. All configurations (i.e. the corresponding computation steps) after* conf_{i-1} *and up to and including* conf_i*, constitute the* ith nondeterministic block *in the computation branch under consideration. The first nondeterministic block starts with the first existential configuration.*

We refer to a nondeterministic block by the type of guess and the level of alternation. In particular, we use $\exists_i$ ($\forall_i$) to refer to the ith existential (universal) block. a guess register whose value has been set by an EXISTS step in the first nondeterministic block is referred to as an $\exists_1$-*guess register*. Any guess register whose value is set in the checking phase is referred to as a *checking guess register*. Each combination of values in the $\exists_1$-guess registers uniquely identifies an $\exists_1$-computation branch. For a given $\exists_1$-branch $b_{\exists_1}$, each combination of values in the checking guess registers uniquely identifies a computation branch b_{check} in the subtree for $b_{\exists_1}$. We refer to each such b_{check} as a *nondeterministic checking branch*. As we will see later, each $\exists_1$-branch corresponds to a parametric polynomial number of nondeterministic checking branches (Property 2.5).

Let us focus on the deterministic simulation of computations of a $W[t]$-program or an $L[t]$-program. Such simulations will play a central role in establishing many of the results in this thesis. With our current knowledge, the only way to deterministically simulate nondeterminism is to enumerate all possible computation branches arising from the nondeterminism. By definition, the programs may have $\Omega(n^{h(k)})$ computation branches. Thus it is unlikely that a deterministic algorithm can simulate the computation in all branches in parametric polynomial time (this belief is of course the basis of the theory of fixed-parameter intractability). Given the inability to perform a complete deterministic simulation in parametric polynomial time, we have to settle for something less. This is precisely the reason we introduce the notion of partial simulation (Definition 2.15). As we will see shortly, the computation of a $W[t]$-program and a variant of the $L[t]$-programs can be partially simulated in deterministic parametric polynomial time. In this context, we will distinguish between the registers whose values are dependent on the $\exists_1$-registers in a nontrivial way (Definition 2.12) from the remaining registers.

Definition 2.12 ($\exists_1$-***operand***) *An operand x in an operation is an* $\exists_1$-*operand if*

- *the current value of x has been set by some $\exists_1$-guess step in the $\exists_1$-nondeterministic block, or*

- *the current value of x has been set by some operation* op *such that* op *uses some $\exists_1$-operand.*

Any operation that uses at least one $\exists_1$-operand is referred to as an $\exists_1$-operation. In particular, tests (JEQUAL and JZERO) on $\exists_1$-operands are referred to as $\exists_1$-tests.

Definition 2.13 (***Partial Configuration***) *A configuration α is a* partial configuration *(with respect to the guess phase) if α specifies the values of all registers except the $\exists_1$-operands. We use a special symbol $\sqcup$ to represent the unknown values in a partial configuration.*

Definition 2.14 formalizes the notion of 'reachable in a single step' or 'yields' for partial configurations.

Definition 2.14 *Let α be a partial configuration in some computation of an AW-program. α yields the partial configurations $\{\beta_0, \ldots, \beta_m\}$, for some $m \geq 0$, in accordance with the following constraints.*

1. *If a register (standard or guess) is not the target operand of $OP(\alpha)$ then the value of the register remains the same in α and β_i, for all i, $0 \leq i \leq m$.*

2. *If* $\mathrm{OP}(\alpha)$ *is an $\exists_1$-test then α yields two partial configurations β_0 and β_1. Here,* $\mathrm{PC}(\beta_0)$ *refers to the instruction that follows* $\mathrm{PC}(\alpha)$ *while* $\mathrm{PC}(\beta_1)$ *refers to the branch target of* $\mathrm{OP}(\alpha)$.

3. *If* $\mathrm{OP}(\alpha)$ *is a test that does not involve any $\exists_1$-operand then α yields a single partial configuration β_0. Here,* $\mathrm{PC}(\beta_0)$ *refers to the instruction that the computation will execute for the known outcome of the test* $\mathrm{OP}(\alpha)$.

4. *If* $\mathrm{OP}(\alpha)$ *is any operation other than a test and α yields a partial configuration β, then* $\mathrm{PC}(\beta)$ *refers to the instruction that follows* $\mathrm{OP}(\alpha)$.

5. *Let* $\mathrm{OP}(\alpha)$ *be any deterministic operation other than branching.* $\mathrm{R}(\alpha)$ *is $\sqcup$ in case at least one operand of α is an $\exists_1$-operand. Otherwise* $\mathrm{R}(\alpha)$ *equals the result produced by* $\mathrm{OP}(\alpha)$.

(a) *The index of the target operand of* $\mathrm{OP}(\alpha)$ *is not an* $\exists_1$*-operand:*
α *yields a single partial configuration* β_0*. In* β_0*, the value of the target register of* $\mathrm{OP}(\alpha)$ *is set to be* $\mathrm{R}(\alpha)$*.*

(b) *The index of the target operand of* $\mathrm{OP}(\alpha)$ *is an* $\exists_1$*-operand:*
α *yields* $l+1$ *partial configurations* $\{\beta_0, \ldots, \beta_l\}$*, where* $[0 \ldots l]$ *is the range of values any standard register (to be used as an index) may store. The value of* $\mathrm{ST}(\beta_i, i)$ *(respectively* $\mathrm{GS}(\beta_i, i)$ *) is set to be* $\mathrm{R}(\alpha)$ *depending on whether the target is a standard or a guess register.*

6. *Let* $\mathrm{OP}(\alpha)$ *be an existential operation in the guess phase.*

(a) *The index of the target operand of* $\mathrm{OP}(\alpha)$ *is not an* $\exists_1$*-operand:*
α *yields a single partial configuration* β_0*. In* β_0*, the target of* $\mathrm{OP}(\alpha)$ *is set to be* $\sqcup$*.*

(b) *The index of the target operand of* $\mathrm{OP}(\alpha)$ *is an* $\exists_1$*-operand:*
α *yields a* $l+1$ *partial configurations* $\beta_0, \ldots \beta_l$*. In* β_i*,* $\mathrm{ST}(\beta, i)$ *or* $\mathrm{GS}(\beta, i)$ *is set to be* $\sqcup$ *depending on whether the type of the target register of* $\mathrm{OP}(\alpha)$ *is standard or guess respectively .*

7. *Let* $\mathrm{OP}(\alpha)$ *be a nondeterministic operation* ($\exists$ *or* $\forall$) *in the checking phase.*

(a) *The index of the target operand of* $\mathrm{OP}(\alpha)$ *is not an* $\exists_1$*-operand:*
α *yields* $l+1$ *partial configurations* $\beta_0, \ldots, \beta_l$*, one partial configuration for each possible value of the guess. Intuitively,* β_i *corresponds to the fact that the value* i *has been guessed by* $\mathrm{OP}(\alpha)$*. In* β_i*, the value of the target register of* $\mathrm{OP}(\alpha)$ *is set to be* i*, for each* i*,* $0 \leq i \leq l$*.*

(b) *The index of the target operand of* $\mathrm{OP}(\alpha)$ *is an* $\exists_1$*-operand:*
α *yields* $(l+1)^2$ *partial configurations* $\{\beta_{0,0}, \ldots, \beta_{l,l}\}$*, where* $[0 \ldots l]$ *is the range of values any standard register (to be used as an index) may store. The value of* $\mathrm{ST}(\beta_{i,j}, i)$ *(respectively* $\mathrm{GS}(\beta_{i,j}, i)$ *) is set to be* j *depending on whether the target is a standard or a guess register.*

8. *If* $\mathrm{OP}(\alpha)$ *is the* HALT *instruction, then* α *does not yield any partial configuration.*

Definition 2.15 *(**Partial computation tree**) Consider the computation of an AW-program* R *on some input* $\langle x, k \rangle$*. A partial (with respect to the* $\exists_1$*-guesses) computation tree* T_{partial} *of* R *on* $\langle x, k \rangle$ *is a tree of partial configurations such that*

- *the root of* T_{partial} *is the starting configuration, and*
- *a partial configuration* α *in* T_{partial} *has a child* β *if and only if* α *yields* β *(Definition 2.14).*

A partial simulation *(with respect to the guess phase) of* R *on* $\langle x, k\rangle$ *is the process of constructing the associated partial computation tree.*

The partial computation tree has a different structure than the computation tree. Figure 2.3 shows the structure of a partial computation tree for basic $W[t]$-program and a basic $L[t]$-program. The branching nodes in a partial computation tree are of the following three types.

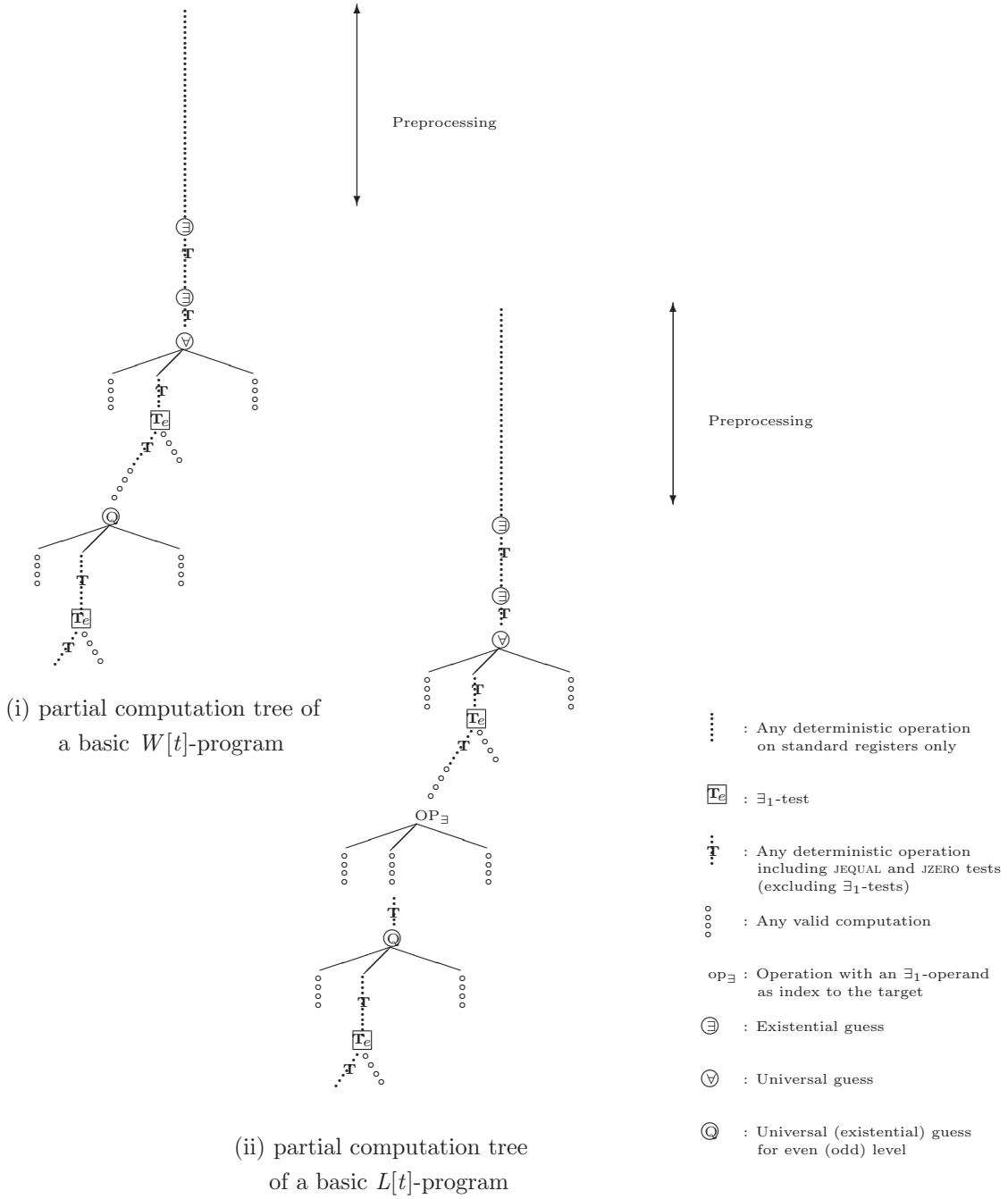
BNodeType1: A nondeterministic partial configuration (corresponding to EXISTS or FORALL) in the checking phase.

BNodeType2: A partial configuration such that the associated operation has an $\exists_1$-operand as index to the target.

BNodeType3: A partial configuration corresponding to some $\exists_1$-test.

Nodes of the first two types arise due to nondeterminism in the guess phase and they do not directly affect the sequence of operations that are executed. However, each branching due to BNodeType3 nodes results in two different sequences of operations. In an actual computation, the $\exists_1$-values determine which sequence gets executed. By constraint $BL1$ (Definition 2.4), all nondeterministic steps are within the last $h(k)$ steps for some function h. Thus the number of $\exists_1$-tests is also bounded by $h(k)$. These $O(h(k))$ binary tests result in at most $2^{O(h(k))}$ different sequences of instructions. We refer to each instruction sequence as an *execution path.* Any computation of a basic program ($W[t]$ or $L[t]$) must correspond to one of these $2^{h(k)}$ execution paths. Note that, the value of an operand of a particular instruction in a given execution path may (and typically will) vary among different nondeterministic computation branches.

Intuitively the partial computation tree represents a verifier. Plugging in the $\exists_1$-values into the partial computation tree, we can decide deterministically whether the guessed values represent a valid solution in time proportional to the size of the partial computation tree. The size of the partial computation tree plays an important role in partial simulation and in the construction of generic reductions.



(i) partial computation tree of a basic $W[t]$-program

(ii) partial computation tree of a basic $L[t]$-program

Figure 2.3: Partial computation tree of (i) a basic $W[t]$-program (ii) a basic $L[t]$-program.

In this thesis, we are primarily concerned with programs having partial computation trees of parametric polynomial size.

In the rest of this section, we analyze properties of the partial computation trees for the basic models. The following property follows from the definition of the partial computation tree and accepting partial configurations.

Property 2.2 *Let x be a nondeterministic node at alternation level i, $2 \leq i \leq t$ in a partial computation tree. Let $T = \{y_1, \ldots, y_m\}$ be the set of all nondeterministic nodes at level $i+1$ such that each $y_j \in T$ is the first nondeterministic node at alternation level $i+1$ along some branch and that y_j is a descendant of x. For a given set $V_\exists$ of $\exists_1$-values, the nondeterministic node x corresponds to an accepting partial configuration if and only if one of the following conditions hold.*

1. *x is a universal node and each node y in T either represents an accepting configuration or is not reachable from x for the values $V_\exists$.*
2. *x is existential and there is at least one node y in T such that y represents an accepting configuration and y is reachable from x for the values $V_\exists$.*

The next theorem specifies a crucial property of the partial computation trees.

Theorem 2.16 *A partial computation tree can be constructed deterministically in time proportional to the size of the tree.*

PROOF. Let us consider the computation of some program R on some input $\langle x, k\rangle$. Let $l = f(k)\,|x|^c$ be the maximum number of steps in any computation branch of R. The initial partial configuration can be constructed deterministically from the input $\langle x, k\rangle$ in $O(l)$ time. The remaining partial configurations can be constructed in a depth-first (or breadth-first) manner. Given a partial configuration α, each child β of α can be constructed from α in $O(l)$ time from the constraints in Definition 2.14. ∎

Property 2.3 *At most a parametric polynomial number of registers (standard or guess) can be used in any computation of an AW-program. These registers are among the first $f(k)n^c$ registers.*

PROOF. The first property follows from the facts that the number of computation steps in any computation branch of a level-t program is bounded above by a parametric polynomial, and that each operation on ARAM involves a constant number of registers. Each register is accessed by specifying its index which in turn has to be stored in some register. By condition AW2 of Definition 2.2, any such index can be at most a $f(k)n^c$, for some function f and constant $c > 0$. This implies the second part of the property. ∎

Property 2.4 *A basic $W[t]$-program or a basic $L[t]$-program has at most $2^{h(k)}$ execution paths.*

PROOF. Let us consider the computation of a basic $W[t]$-program (or a basic $L[t]$-program) R on input $\langle x, k\rangle$. Let us consider a test operation performed before the first existential step. For a given input, the outcome of the test is fixed and the program deterministically follows one of the two branches (independent of the nondeterministic guesses). However any test that uses the nondeterministically guessed values as operands may cause the program to follow any of the two branches, doubling the number of execution paths at the point of the test. Since the number of such tests is at most $h(k)$ (Constraint BL1 in Definition 2.4), such doubling can occur at most $h(k)$ times. ∎

Property 2.5 *In any computation of a basic $W[t]$-program or a basic $L[t]$-program, the number of checking nondeterministic branches is bounded above by a parametric polynomial.*

PROOF. The upper bound of the range of values for a nondeterministic step is specified in the 0th standard register. By constraint AW2 in Definition 2.2, the bound is a parametric polynomial. Thus each nondeterministic operation results in a parametric polynomial number of nondeterministic computation branches. By constraints T1 and TU1 of Definition 2.3, the number of nondeterministic guess steps in the checking phase is bounded by a constant. ∎

Property 2.6 *The size of the partial computation tree of a basic $W[t]$-program for a given input is bounded above by a parametric polynomial.*

PROOF. In a partial computation tree of a basic $W[t]$-program, branching occurs due to (i) nondeterministic guess operations in the checking phase and (ii) two-way branching based on tests on $\exists_1$-values. Thus each leaf corresponds to a combination of nondeterministic checking branch and an execution path. By Properties 2.4 and 2.5, the number of such combinations is bounded above by a parametric polynomial. By Constraint AW1 in Definition 2.2, there can be at most a parametric polynomial number of computation steps in any computation branch. Since any operation uses a constant number of operands, the size of each partial configuration is bounded by a parametric polynomial. ∎

Property 2.6 is of immense importance because the entire partial computation tree can be constructed deterministically in parametric polynomial time (Theorem 2.16). This is useful for constructing generic reductions. Unfortunately an analogue of Property 2.6 does not seem to hold for the basic $L[t]$-programs. The possibility of using an $\exists_1$-operand as an index to a target register may result in unbounded branching for any operation in the checking phase. The number of such operations may be $\Omega(h(k))$ resulting in a partial configuration tree of size $\Omega(n^{h(k)})$. However, in the next section we will develop a normalized variant of the $L[t]$-program having a partial computation tree of size bounded by a parametric polynomial.

Property 2.7 *(Chen, Flum, and Grohe [21]) In any computation of a basic $W[t]$-program, $t \geq 2$, values in standard registers are independent of the nondeterministically guessed values. In other words, the values in standard registers depend on the execution paths only.*

PROOF. By constraint BW1 in Definition 2.5, a basic $W[t]$-program can perform W-operations only. Thus the nondeterministic operations must store the guessed values in guess registers. Only JEQUAL and JZERO tests can use the guessed values as operands and these tests do not modify the content of any register. ∎

Note that a JZERO test uses the value of some standard register to decide the test outcome. Property 2.7 implies that a standard register can never be an $\exists_1$-operand in any computation of a basic $W[t]$-program. In other words, the value of a standard register in a partial configuration is never $\sqcup$. Thus, given a test, the pair of $\exists_1$-values that causes the operation to branch can be computed from the corresponding partial configuration in deterministic parametric polynomial time.

Property 2.8 *As long as the values in guess registers g_{r_i} and g_{r_j} remain the same, $\langle g_{r_i}, g_{r_j} \rangle$ maps to the same standard register.*

PROOF. This is immediate as the mapping $\langle \cdot , \cdot \rangle$ is deterministic. ∎

Property 2.8 is a severe limitation of the basic $W[t]$-programs. In particular, verifying whether the nondeterministically guessed values satisfy multiple relations becomes nontrivial due to the property. Such verification can be done, though, by partitioning the range of values in a complicated manner. An example is given in the next section where we construct an algorithm to decide a contrived version of DOMINATING SET by a basic $W[2]$-program (Algorithm 2.2).

A basic $W[1]$-program can overcome the limitation utilizing its capability to access the nondeterministically guessed values directly. Consider the case where a $W[1]$-program needs to check whether a pair of $\exists_1$-values satisfies relations R_1 and R_2. In the preprocessing phase, the $W[1]$-program can construct two separate lookup tables for R_1 and R_2, respectively. Let the lookup tables be arranged such that $r_{\text{start}_1+\langle i,j\rangle}$ (respectively $r_{\text{start}_2+\langle i,j\rangle}$) specifies whether the value pair (i, j) satisfies the relation R_1 (respectively R_2). In the checking phase, the $W[1]$-program can compute the index $u_1 = \text{start}_1 + \langle i, j\rangle$ (or $u_2 = \text{start}_2 + \langle i, j\rangle$) in constant time. The program then examines the value in r_{u_1} (or r_{u_2}) to decide whether the pair (i, j) satisfies the relation R_1 (or R_2).

We are now ready to present the new variants of the computational model (Section 2.3). The equivalence among the variants is established in Section 2.4.

2.3 The New Variants of the Computational Models

In this section we introduce the new variants of $W[t]$-programs and $L[t]$-programs. The new variants are obtained by relaxing or restricting the computational features identified in the previous section. We also demonstrate the usefulness of the new variants through some examples. Theoretically, any result derived using one of the characterizations can also be obtained using any other equivalent characterization. However, picking a suitable characterization often makes the task at hand much easier to accomplish. For example, an extended $W[t]$-program may be preferred to others for proving upper bounds (with respect to $W[t]$ classes) because of its capability to implement natural algorithms.

Model	**Steps after first $\exists_1$-step**	**Bounds on allowed operations on $\exists_1$-operands**	**Bounds on allowed operations on non-$\exists_1$ guess registers**
Extended-$L[t]$	$f(k)n^c$	EXISTS - $g(k)$, Any other op - $g(k)$	EXISTS/FORALL - constant, Any other op - $f(k)n^c$
Basic-$L[t]$	$g(k)$	EXISTS - $g(k)$, Any other op - $g(k)$	EXISTS/FORALL - constant, Any other op - $g(k)$
Normalized-$L[t]$	$g(k)$	EXISTS - $g(k)$, JEQUAL/JZERO/ASSIGN - $g(k)$	EXISTS/FORALL - constant, JEQUAL/JZERO/ASSIGN - $g(k)$
Extended-$W[t]$	$f(k)n^c$	EXISTS - $g(k)$, JEQUAL/JZERO - $g(k) + d \log n$	EXISTS/FORALL - constant, Any other op - $f(k)n^c$
Basic-$W[t]$	$g(k)$	EXISTS - $g(k)$, JEQUAL/JZERO - $g(k)$	EXISTS/FORALL - constant, JEQUAL/JZERO - $g(k)$
Normalized-$W[t]$	$g(k)$	EXISTS - $g(k)$, JEQUAL/JZERO - $g(k)$	EXISTS/FORALL - constant, JEQUAL/JZERO - $g(k)$ *Unique* accepting/rejecting execution path

Table 2.1: Summary of resource bounds for the original and new variants of the computational models.

2.3.1 $W[t]$-programs

We present two new variants of the computational model for the $W[t]$ classes. The first is an extension to the basic model while the second is a restricted version of the basic model. Both the new variants are (t, u)-alternating programs (Definition 2.3). The distinction among the models comes from the nature and number of operations that they allow after the first nondeterministic step. Properties of the basic models and the new variants are summarized in Table 2.1.

We obtain the *extended characterization* of $W[t]$, $t \geq 2$ by relaxing the access restriction on the Q_i-registers, $Q \in \{\exists, \forall\}$, $2 \leq i \leq t$, as well as the time bound on the checking phase. Note that $W[1]$ equals $L[1]$, by definition. Therefore, a $W[1]$-program has complete access to the $\exists_1$-registers. We therefore develop the extension for higher levels only.

Definition 2.17 (***Extended*** $W[t]$***-program***) *A (t, u)-alternating program is an* extended $W[t]$*-program, $t \geq 2$, if and only if*

EW1: R performs at most $h(k) + O(\log n)$ W-operations (Definition 2.1) on the $\exists_1$-registers.

As was the case with the basic $W[t]$-programs, an extended $W[t]$-program can perform the W-operations only, on the $\exists_1$-registers. However, an extended $W[t]$-program has complete access to all values that are guessed in the checking phase. The nondeterministic instructions can appear anywhere in the computation, not necessarily in the last $g(k)$ steps, for some function g. The nondeterministically guessed values from the second and subsequent blocks can be used in parametric polynomial number of computation steps. However, the number of tests that involve at least one existential register from the first block cannot exceed $h(k) + c \log n$, for some function h and some constant $c > 0$. Note that the previous characterization (Theorem 1.4) has an implicit bound of $h(k)$ on the number of such binary tests.

Let us analyze the computational properties of the extended $W[t]$-programs. Since the number of steps in any computation branch is bounded by a parametric polynomial, the depth of the partial configuration tree is a parametric polynomial as well. The computation tree has a similar structure as that for the basic $W[t]$-programs. The number of $\exists_1$-tests ($\mathbf{T}_e$) in any root to leaf path can be $O(h(k) + c \log n)$ as opposed to $h(k)$. This increases the number of leaves by a parametric polynomial factor only. Thus Properties 2.5, 2.6, and 2.8 still hold. The bound specified in Property 2.4 on the number of execution paths changes to $g(k)n^c$, for some function g and constant c. The checking guess registers may be used in any operation. Thus Property 2.7 changes to 'standard registers do not depend on the $\exists_1$-operands'. In addition, we identify the following property that will be used later for establishing the equivalence among the variants.

Property 2.9 *The following information can be computed (without knowing the $\exists_1$-values) from a given partial computation tree of an extended $W[t]$-program in deterministic polynomial time.*

- *I1: The value of any standard register at any given partial configuration.*
- *I2: The number of nondeterministic steps in each nondeterministic block in each nondeterministic checking branch.*
- *I3: The range for each nondeterministic operation.*
- *I4: The accepting and rejecting execution paths for each nondeterministic checking branch.*
- *I5: The pairs of values that results in a certain outcome for each test (*JEQUAL *or* JZERO*) in each execution path for each nondeterministic checking branch.*

PROOF. The proof for I1, I2, I3, and I4 follows immediately as any partial configuration always specifies the value of any standard register (Property 2.7, appropriately restated as mentioned above). I5 follows from Property 2.7 (restated) and the fact that the encoding $\langle \cdot, \cdot \rangle$ can be computed deterministically. ∎

The restricted $W[t]$-programs can be considered *normalized* forms of the original $W[t]$-programs. Thus, the fact that the two forms are equivalent can be viewed as the *computational analogue of the Normalization Theorem.* As we will demonstrate later, the normalized $W[t]$-programs can play a role similar to t-NORMALIZED WCS for proving hardness results. The added advantage of starting from normalized $W[t]$-programs is that the proofs may be extended to classes in the L-hierarchy without much extra effort. As the t-NORMALIZED WCS does not have an analogue for the $L[t]$-classes, such extensions may not be obvious in the context of circuits.

Definition 2.18 (***Normalized*** $W[t]$***-program***) *A basic $W[t]$ program R is a* normalized-$W[t]$-program*, $t \geq 1$, if on any input, any computation branch of R*

NW1: has at most one nondeterministic step in each nondeterministic block in the checking phase,

NW2: guesses a value from the same range in each nondeterministic step in the checking phase,

NW3: has all deterministic operations that depend on some nondeterministically guessed value, after all the nondeterministic steps,

NW4: has at most $h'(k)$ tests in the checking phase such that each test involves at most one guess register from the guess phase (for $t \geq 2$, only) , and

NW5: for even t (odd t), has a unique rejecting (accepting) execution path.

The additional constraints on the normalized $W[t]$-programs affect the structure of the partial computation tree. Compared to that of a basic $W[t]$-program, the partial computation tree of a normalized $W[t]$-program has a more restricted structure (Figure 2.4). As the normalized $W[t]$-programs are restricted variants of the basic $W[t]$-programs, the properties (2.4 to 2.7) hold for the normalized $W[t]$-programs as well.

We will show the equivalence among the variants in Section 2.4. In the rest of this section, we analyze the usefulness of the new variants of the computational model for $W[t]$. We illustrate the usefulness of the extended $W[t]$-programs by showing the $W[2]$-membership of a contrived version of DOMINATING SET defined as follows.

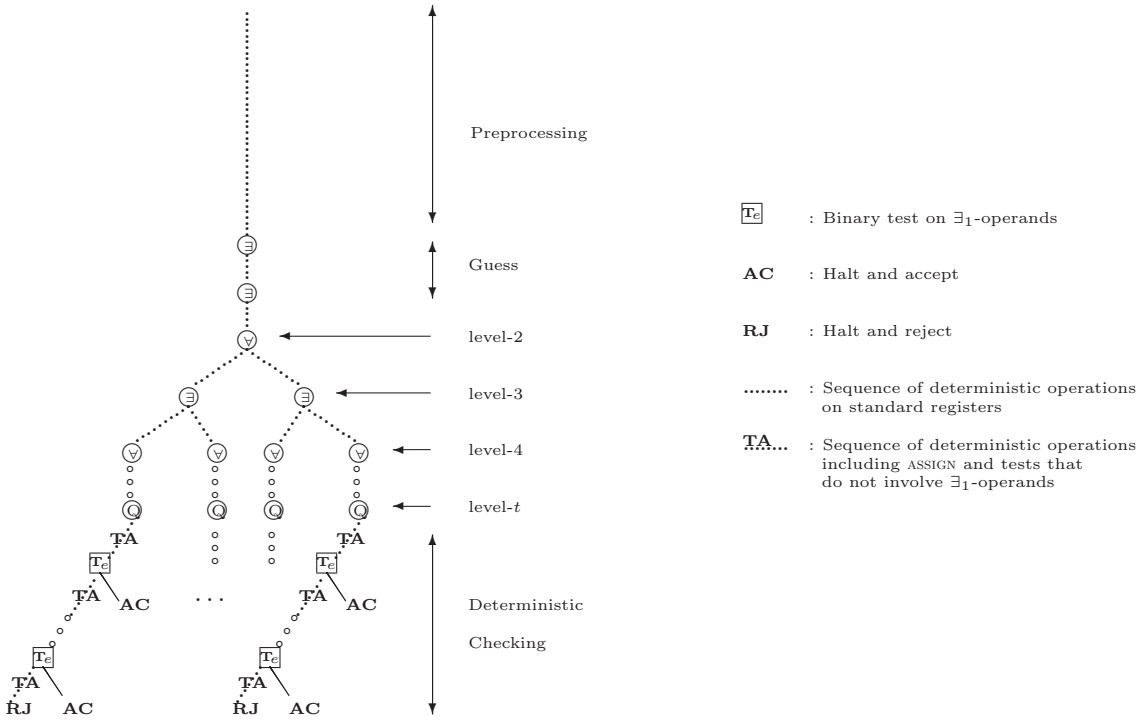


Figure 2.4: Structure of the partial computation trees for normalized $L[t]$-programs. The structure for the normalized $W[t]$-programs is similar with the exception that the operations in the final checking are restricted to W-operations only.

Not-too-close Dominating Set (DS-1)

Input: A graph $G = (V, E)$, a positive integer k, and an integer d.
Parameter: k.
Question: Is there a dominating set $V' \subseteq V$ of size k such that the length of the shortest path between any pair of vertices in V' is at least d?

We can construct an algorithm to decide DS-1 by modifying Algorithm 1.1 in the obvious way. The new algorithm works the same way as Algorithm 1.1 until the universal step is performed. The new algorithm *rejects* in the final phase if the universally selected vertex is not dominated or for some pair of vertices x_i and x_j, $1 \leq i < j \leq k$, the length of the shortest path between x_i and x_j is less than d. The checking corresponds to the formula $\varphi_{\text{DS-1}}$, defined as follows.

Algorithm 2.1: Algorithm to decide NOT-TOO-CLOSE DOMINATING SET by an extended $W[2]$-program.

DS-1-Extended(a graph $G = (V, E)$, an integer k)

`/* Guess` k `vertices for the dominating set */`

1 **Nondeterministic block 1: Existentially** guess the indices (from the range $[1 \ldots |V|]$) of k vertices for the dominating set. Store the values in guess registers $g_1, \ldots, g_k$.

2 **Nondeterministic block 2: Universally** guess the index (from the range $[1 \ldots |V|]$) of a vertex and store it in register g_{k+1}.

3 Construct a lookup table so that $r_{\langle i,j \rangle}$ is 1 if and only if the ith and jth vertices are apart in G.

4 **for** $p = 1$ *to* $k - 1$ **do**
5 **for** $q = p + 1$ *to* k **do**
6 **if** *vertices* g_p *and* g_q *are not apart* **then**
7 Reject in this branch
8 **end**
9 **end**
10 **end**

11 Construct a lookup table so that $r_{\langle i,j \rangle}$ is 1 if and only if the ith and jth vertices are adjacent in G.

12 **for** $p = 1$ *to* k **do**
13 **if** *vertices* g_p *and* g_{k+1} *are adjacent or* $g_p = g_{k+1}$ **then**
14 Accept in this branch
15 **end**
16 **end**
17 Reject in this branch.

18 End DS-1-Extended

$$\varphi_{\text{DS-1}}(u) = \left[\bigvee_{i=1}^{k} (\ x_i = u \ \vee \langle x_i, u \rangle \in E \ ) \right] \wedge \left[\bigwedge_{i=1}^{k} \bigwedge_{j=i+1}^{k} \mathsf{apart}(x_i, x_j) \right],$$

where $\mathsf{apart}(x_i, x_j)$ is *true* if and only if $x_i, x_j \in V$ and the length of the shortest path between x_i and x_j is at least d.

Verification of $\varphi_{\text{DS-1}}$ can be easily implemented by an extended $W[2]$-program $R_{\text{DS-1}}$. $R_{\text{DS-1}}$ existentially guesses k vertices for the solution, universally guesses a vertex u, verifies that (a) the vertices in the solution are pairwise apart and that (b) u is dominated by the solution. $R_{\text{DS-1}}$ computes the lookup table for the apart relation before verifying part (a). The lookup table can be constructed by

computing the shortest distance between each pair of vertices (using the well-known all-pair shortest path algorithm due to Dijstra, for example) and then verifying whether the shortest distance is at least d. Thus the construction of the lookup table requires polynomial time and the entire construction is independent of the values in $\exists_1$-registers. $R_{\text{DS-1}}$ completes the verification of part (a) by performing at most $\binom{k}{2}$ JZERO tests and the newly constructed lookup table. Once this part of the verification is done, $R_{\text{DS-1}}$ replaces the lookup table by the adjacency matrix and verifies part (b). Construction of the adjacency matrix takes polynomial time and is again independent of the $\exists_1$-guesses. Verification of part (b) requires $2k$ additional JEQUAL and JZERO tests. The algorithm accepts in a universal branch if and only if the universally selected vertex is dominated by the existentially guessed solution and all the vertices in the solution are pairwise apart. The details are given in Algorithm 2.1.

Let us analyze whether the algorithm meets the constraints for an extended $W[2]$-program. Lines 1 and 2 takes $O(k+1)$ time. The loops from Line 4 to 10 take $O(\binom{k}{2})$ time. The loop from Line 12 to Line 16 takes $O(k)$ time. Construction of the adjacency matrix (Line 11) takes $O(n^2)$ time. Construction of the lookup table for apart takes $O(n^3)$ time. Thus the number of steps in a computation branch is $O(n^3)$ (Constraint AW1 Definition 2.3). In Lines 1 and 2, the algorithm needs to store the indices to the vertices into guess registers. During the construction of the lookup tables in Lines 3 and 11, the algorithm needs to store indices to the lookup table entries. Each of these indices is at most a polynomial in $|x|$. Thus Constraint AW2 is satisfied. The nondeterministic steps in Lines 1 and 2 satisfy the constraints AW3, T1, and TU1 for $t = 2$. A total of $\binom{k}{2} + k$ tests are performed on the $\exists_1$-registers (Lines 6 and 13). Thus constraint EW1 is also satisfied.

Let us now construct a basic $W[2]$-program in order to decide NOT-TOO-CLOSE DOMINATING SET. By Property 2.8, a basic $W[2]$-program always maps a given pair $\langle u, v\rangle$ to the same standard register (i.e. into the same lookup table entry). This introduces a difficulty in the checking process as two lookup tables need to be used to verify $\varphi_{\text{DS-1}}$, one for the adjacency relation and the other for the relation apart, to perform the checking. A solution for the basic characterization is to existentially guess a second copy of each of the original $\exists_1$-values such that the second copies map to the second lookup table. In addition, the solution needs to make sure that the two copies of corresponding $\exists_1$-values are consistent, which in turn requires another lookup table. We implement the idea by defining a range of value $[0 \dots l]$ that a register may store and then partitioning the range into several subranges in order to represent different entities of interest. In particular, we reserve the values

- $\{0, 1\}$ for special use (to be described later),

Algorithm 2.2: Algorithm to decide NOT-TOO-CLOSE DOMINATING SET by a basic $W[2]$-program.

DS-1(a graph $G = (V, E)$, an integer k)

Construct a lookup table T as follows.

$$T[i,j] = \begin{cases} 1, & i = 0 \text{ and } 2 \leq j \leq |V| + 2 \\ 1, & i = 1 \text{ and } |V| + 3 \leq j \leq 2|V| + 2 \\ 1 & 3 \leq i, j \leq |V| + 2 \text{ and vertices } i-2 \text{ and } j-2 \text{ are adjacent} \\ 1 & 3 \leq i \leq |V| + 2, |V| + 3 \leq j \leq 2|V| + 2 \\ & \text{and } i \text{ and } j \text{ refer to the same vertex in } V \text{ (i.e. } i - 2 = j - |V| - 2 \text{)} \\ 1 & |V| + 3 \leq i, j \leq 2|V| + 2 \\ & \text{and vertices } (i - |V| - 2) \text{ and } (j - |V| - 2) \text{ are apart} \\ 0, & \text{otherwise} \end{cases}$$

`/* Initialize` g_1 `and` g_2 `with values` 0 `and` 1 `respectively */`
Nondeterministic block 1: Existentially guess a value less than or equal to 0 and store in g_1.
Nondeterministic block 1: Existentially guess a value less than or equal to 1, store it in g_2 and ensure that $g_1 \neq g_2$.

`/* Guess` k `vertices for the dominating set */`
Nondeterministic block 1: Existentially guess the indices (from the range $[2 \ldots (|V| + 1)]$) of k vertices for the dominating set. Store the values in guess registers $g_3, \ldots, g_{k+2}$.

`/* Guess second copies of the values in` $g_3, \ldots, g_{k+2}$ `*/`
Nondeterministic block 1: Existentially guess a second copy (from the range $[(|V| + 3) \ldots (2|V| + 2)]$) of the value in g_i and store the value in guess register g_{k+i}, $3 \leq i \leq k + 2$.

Nondeterministic block 2: Universally guess the index (from the range $[2 \ldots (|V| + 1)]$) of a vertex and store it in register g_{2k+3}.

for $p = 3$ *to* $k + 2$ **do**
  if g_p *and* g_{k+p} *do not refer to the same vertex* **then** Reject in this branch
end
for $p = k + 3$ *to* $2k + 2$ **do**
  for $q = p + 1$ *to* $2k + 2$ **do**
    if *vertices* g_p *and* g_q *are not apart* **then** Reject in this branch
  end
end
for $p = 3$ *to* $k + 2$ **do**
  if *vertices* g_p *and* g_{2k+3} *are adjacent* **then**
    Accept in this branch
  end
end
Reject in this branch.

End DS-1

- $[2 \ldots (|V| + 1)]$ to represent the vertices in V, the value i representing the vertex $i - 1$, and

- $[(|V| + 2) \ldots (2\,|V| + 1)]$ to represent the second copies of the vertices in V, the value i representing the vertex $(i - |V| - 1)$.

Intuitively, the lookup table will have two main parts, the first part representing the adjacency matrix and the second part representing the table for apart. The table entries are arranged in such a way that a value pair $\langle i, j \rangle$, $2 \leq i, j \leq (|V| + 1)$, maps to the $[i, j]$th entry of the adjacency matrix while a value pair $\langle |V| + 1 + i, |V| + 1 + j \rangle$, $1 \leq i, j \leq |V|$, maps to the $[i, j]$th entry in the table for apart. The $[0, i]$th entry, $2 \leq i \leq (|V| + 1)$ and the $[1, j]$th entry, $(|V| + 2) \leq j \leq (2\,|V| + 1)$, are set to be one. These entries will be looked at to ensure that a certain $\exists_1$-register stores a first copy (or a second copy) of a vertex. The algorithm is presented in detail as Algorithm 2.2. It is reasonable to ask whether the verification can be implemented in a more natural way on the basic model. However, existence of any natural alternative seems unlikely because of Property 2.8. We will use this technique of partitioning the range of values once again when we establish the equivalence among different variants of $W[t]$-programs.

Thus the extended characterization of the $W[t]$ classes allow us to construct natural algorithms for parameterized problems. We will present several new membership results in Chapter 5 where we will make extensive use of this capability. The extended characterization is also useful for establishing new structural results. In particular, the ability to perform $O(h(k) + \log n)$ tests on the $\exists_1$-registers (as opposed to $O(h(k))$ tests in the original characterization) allows an extended $W[t]$-program to decide problems from nontrivial subclasses from higher levels of the W-hierarchy. We will describe the techniques in detail in Chapter 4 where we present some new structural results regarding the relationship among the classes in the W-hierarchy and the relationship between the $W[t]$ and $L[t]$ classes.

The effect of the normalized $W[t]$-programs becomes apparent once we consider the task of proving lower bounds. The first two levels of the W-hierarchy contain many natural complete problems. However, in order to show lower bounds beyond $W[2]$, one has to choose WCS on weft-t constant depth circuits (or some variant of it) as the starting point. The normalized $W[t]$-programs provide a convenient alternative to t-Normalized WCS. In some cases, the hardness results obtained from a normalized $W[t]$-program may be extended to $L[t]$ classes. In Chapter 5, we will see examples of such extension where we show that the Precedence Constrained Multiprocessor Scheduling parameterized by the number of processors is hard for $L[t]$, $t \geq 1$. Since no analogue of t-Normalized WCS is

known for the $L[t]$ classes, it is not obvious whether similar extensions are possible in the context of circuits. We end this section by showing how the normalized $W[1]$-programs can be used to show lower bounds for $W[1]$-hard problems. As an example we construct a new $W[1]$-hardness proof for CLIQUE based on the normalized $W[1]$-programs. The original $W[1]$-hardness result was established by Downey and Fellows [25].

Theorem 2.10 *(Downey, Fellows [25])* CLIQUE *is hard for* $W[1]$.

PROOF. (A new $W[1]$-hardness proof) Let $R_{\text{NORMALIZED}}$ be a *normalized* $W[1]$-program. We construct a fixed-parameter reduction $\mathcal{A}$ that takes an input $\langle x, k\rangle$ and constructs an instance $\langle x', k'\rangle$ of CLIQUE such that $\langle x', k'\rangle$ is in CLIQUE if and only if $R_{\text{NORMALIZED}}$ accepts $\langle x, k\rangle$. Let $R_{\text{NORMALIZED}}$ make $h_1(k)$ existential guesses and $h_2(k)$ tests, each involving a pair of $\exists_1$-guesses. Also, let the values stored by $R_{\text{NORMALIZED}}$ in any register be at most $(fp-1)$. Note that in the checking phase $R_{\text{NORMALIZED}}$ rejects as soon as any of the tests along the unique accepting execution path is not satisfied. On input $\langle x, k\rangle$, the reduction $\mathcal{A}$ deterministically simulates the computation of the preprocessing phase of $R_{\text{NORMALIZED}}$ on $\langle x, k\rangle$. $\mathcal{A}$ creates the vertices $Y_{\langle i,j\rangle}$, $1 \leq i \leq h(k)$, $0 \leq j \leq (fp-1)$, for $\langle x', k'\rangle$. Including a vertex $Y_{\langle i,j\rangle}$ in the clique corresponds to specifying that the ith $\exists_1$-guess of $R_{\text{NORMALIZED}}$ is the value j. Since the values in the standard registers of $R_{\text{NORMALIZED}}$ are independent of the $\exists_1$-values, $\mathcal{A}$ can compute those values for each computation step of $R_{\text{NORMALIZED}}$. Thus, given a test (JEQUAL or JZERO), and a pair of values for the registers involved, $\mathcal{A}$ can determine whether the computation will *reject* or continue along the unique *accepting* deterministic branch. The instance x' includes an edge between $Y_{\langle i_1,j_1\rangle}$ and $Y_{\langle i_2,j_2\rangle}$ if and only if the values j_1 and j_2, when stored in $\exists_1$-registers g_{i_1} and g_{i_2}, respectively, cause the computation to continue along the unique accepting execution path. No edge is included between $Y_{\langle i,j_1\rangle}$ and $Y_{\langle i,j_2\rangle}$, $j_1 \neq j_2$. The parameter k' is set to $h_1(k)$. Let us assume that x' has a clique of size $h_1(k)$. The clique includes exactly one $Y_{\langle i,j\rangle}$ for each i, $1 \leq i \leq h_1(k)$. This follows from the construction of x' and the requirement that the clique must have size $h_1(k)$. We construct an accepting computation of $R_{\text{NORMALIZED}}$ on $\langle x, k\rangle$ by storing the value j in the ith $\exists_1$-register of $R_{\text{NORMALIZED}}$, in case $Y_{\langle i,j\rangle}$ is included in the clique. By construction, no such pair of values falsifies any test along the unique accepting deterministic branch of $R_{\text{NORMALIZED}}$. The correctness in the other direction can be shown by applying the argument in reverse. ∎

We would like to point out that the hardness results for other basic $W[1]$-complete problems including INDEPENDENT SET and WEIGHTED ANTIMONOTONE 2-SAT can be shown in essentially the same way.

2.3.2 $L[t]$-programs

We define the new variants of $L[t]$-programs in this subsection. We start with the extended characterization. A basic $L[t]$-program has complete access to all nondeterministically guessed values (Definition 2.4). However, the number of operations that can be performed on the nondeterministically guessed values is bounded by a function of the parameter only. The extended characterization relaxes the time bound on the checking phase and on the number of operations on values that has been guessed nondeterministically in the checking phase.

Definition 2.19 (***Extended*** $L[t]$***-program***) *An extended $L[t]$-program, $t \geq 2$, is a (t, u)-alternating program that satisfies the following condition.*

EL1: In any computation branch, the number of operations that uses some $\exists_1$-operand is at most $h(k)$, for some function h.

Note that an extended $L[t]$-program allows the nondeterministically guessed values from the second and subsequent blocks to be used in a parametric polynomial number of operations.

Next we define the *normalized* variant of the $L[t]$-programs. To date, the defining Parameterized Model Checking problem, having both relations and functions in the associated vocabulary, is the only known $L[t]$-complete problem [19]. The normalized $L[t]$-programs can serve as an alternative starting point for showing $L[t]$-hardness results.

Definition 2.20 (***Normalized*** $L[t]$***-program***) *A normalized $L[t]$-program, $t \geq 2$, is a basic $L[t]$-program such that in each nondeterministic computation branch, it*

NL1: has at most one nondeterministic step in each nondeterministic block in the checking phase,

NL2: guesses a value from the same range in each nondeterministic step in the checking phase, and all nondeterministically guessed values are stored in guess registers,

NL3: has all deterministic operations that depend on some nondeterministically guessed value, after all the nondeterministic steps,

NL4: has at most $h'(k)$ tests in the checking phase such that each test involves at most one $\exists_1$-operand (for $t \geq 2$, only), and

NL5: for even t (odd t), has a unique rejecting (accepting) execution path for each nondeterministic checking branch.

The distinguishing feature of a normalized $L[t]$-program is that it does not have direct access to the values that are guessed nondeterministically. Use of these values are restricted to tests (JEQUAL and JZERO) and ASSIGN operation only. These restrictions on the resources result in the following computational features of the normalized $L[t]$-programs.

Property 2.11 *Let* op *be an operation performed by a normalized $L[t]$-program. The index of any operand of* op *can be computed deterministically in parametric polynomial time.*

PROOF. By definition (Section 2.1), an index to an operand of any operation must be stored in a standard register. By Constraint NL4 all the nondeterministically guessed values are stored in guess registers. Thus, only tests and ASSIGN operation can be performed on the nondeterministically guessed values and none of these operations can modify the content of a standard register. ∎

Property 2.12 *The size of the partial computation tree of a normalized $L[t]$-program is bounded above by a parametric polynomial.*

PROOF. By Constraint NL2 and the definition of the operations on ARAM, a normalized $L[t]$-program can not use an $\exists_1$-value as index. A branching node in the partial computation tree is either a nondeterministic operation or an $\exists_1$-test. Thus each leaf represents a unique combination of a nondeterministic checking branch and an execution path. By Constraints T1, TU1, NL4, and NL5 the number of leaves is bounded by a parametric polynomial. ∎

A nice consequence of the normalized characterization of $L[t]$ is that the register usage in the checking phase of any nondeterministic branch can be represented by a directed graph of size $h'(k)$, for some function h'. We call the graph the *assignment graph* of the corresponding checking phase. Let us consider the computation of an $L[t]$-program R. Let, $\mathcal{I}'_{b,p}$ be the sequence of instructions in the execution path p for some nondeterministic checking branch b. Let, $\mathcal{I}_{b,p}$ be the subsequence of $\mathcal{I}'_{b,p}$ consisting of all instructions such that at least one operand in each instruction in $\mathcal{I}_{b,p}$ depends on some $\exists_1$-operand. By Definition 2.20, each instruction in $\mathcal{I}_{b,p}$ is an indexed assignment or a binary test instruction. The assignment graph of the instruction sequence $\mathcal{I}_{b,p}$ is defined as follows.

Definition 2.21 *An* assignment graph *for execution path p in a nondeterministic checking branch b is a directed graph $G_{b,p} = (V_{b,p}, E_{b,p})$ whose vertex set consists of all instructions in $\mathcal{I}_{b,p}$ and all $\exists_1$-registers used by R in this computation path. The edge set $E_{b,p}$ is defined as follows.*

$$\begin{aligned} E_{b,p} \;=\; & \{\, \langle x, y\rangle \mid x, y \in \mathcal{I}_{b,p} \text{ and an operand of } y \text{ is computed by instruction } x \,\} \bigcup \\ & \{\, \langle x, y\rangle \mid y \in \mathcal{I}_{b,p}, x \text{ is an } \exists_1\text{-register and } x \text{ is an operand of } y \,\} \end{aligned}$$

$G_{b,p}$ is essentially a directed acyclic graph with $O(h(k)+k)$ vertices and $O(h(k)+k)$ edges, where k is the number of $\exists_1$-guesses and $h(k)$ is the number of operations in the final checking phase. $G_{b,p}$ can be converted into a binary forest $F_{b,p}$ of size $O(2^{h(k)})$ by duplicating each vertex (along with its subtree) whose outdegree is greater than one. Each non-leaf node of $F_{b,p}$ corresponds to an instruction in $\mathcal{I}_{b,p}$ and each leaf node refers to an existential register from the first nondeterministic block of R. Also, the root node of each tree in $F_{b,p}$ corresponds to a binary test (JEQUAL or JZERO). The assignment graph will be a key ingredient in the $L[2]$-hardness proof of a version of the BOUNDED DETERMINISTIC PDA INTERSECTION problem.

2.4 Equivalence Among the Variants of the Computational Model for $W[t]$

We now establish the equivalence among all three variants of the computational model for $W[t]$. An important step of the proof of the equivalence (as well as the proofs of most hardness and simulation results in this thesis) will be the construction of a partial computation tree and analyzing its nodes. In this context, we will pay particular attention to the branching nodes in the partial computation tree. Consider a *branching node* x having q branching nodes as ancestor in the unique path from the root to x. We refer to x by $\langle b_1, \ldots, b_q\rangle$ where b_i is the branch taken for the ith branching ancestor (starting from the root) of x in order to reach x from the root node. Also, given a branching node $x = \langle b_1, \ldots, b_q\rangle$, we use $\mathcal{B}(\langle b_1, \ldots, b_q\rangle)$ to denote the number of branches resulting from x.

Theorem 2.13 *Let* Q *be a parametric problem. The following are equivalent.*

1. Q *is in* $W[t]$.

2. Q *can be decided by an* extended $W[t]$*-program,* $t \geq 2$ *and by a basic* $W[1]$*-program for* $t = 1$.

3. Q *can be decided by a* normalized $W[t]$*-program.*

PROOF. (3 $\Rightarrow$ 1) A normalized $W[t]$-program is a basic $W[t]$-program.

(1 $\Rightarrow$ 2) A basic $W[t]$-program is an extended $W[t]$-program, for $t \geq 2$.

(2 $\Rightarrow$ 3) We now show that the computation of an extended $W[t]$-program R_{EXTENDED} can be simulated by a normalized $W[t]$-program $R_{\text{NORMALIZED}}$. We describe the proof for even t, $t \geq 2$. The proof for odd t, $t \geq 3$ can be constructed applying similar techniques and the property of duality.

As specified in Constraint AW2 in Definition 2.2, let $g(k)n^d$ be the maximum value that R_{EXTENDED} may store in any register in any computation step. Let R_{EXTENDED} make $h_1(k) \leq h'(k)$ $\exists_1$-guesses (Constraint AW3). Let $[0 \ldots (l_\exists - 1)]$ be the maximum range over all $\exists_1$-guesses of R_{EXTENDED}. Also let u be the maximum number of nondeterministic operations in any nondeterministic block in the checking phase of R_{EXTENDED} (Constraint TU1 in Definition 2.3). In the first part of the preprocessing phase, $R_{\text{NORMALIZED}}$ partially simulates the computation of R_{EXTENDED} on a given input $\langle x, k \rangle$ and constructs the associated partial computation tree C_{extended}. Let m be the number of branches resulting from the terminal nondeterministic nodes in C_{extended}. In the second part of the preprocessing phase $R_{\text{NORMALIZED}}$ constructs a lookup table $\mathcal{T}$. The purpose of constructing $\mathcal{T}$ is to precompute various information about the computation of R_{EXTENDED} so that all necessary checking can be performed within the resource constraints of a normalized $W[t]$-program. $R_{\text{NORMALIZED}}$ then starts simulating the computation of R_{EXTENDED} on $\langle x, k \rangle$. In order to satisfy Constraint NW2 in Definition 2.18, any nondeterministic operation in $R_{\text{NORMALIZED}}$ will guess a value from the same range $[0 \ldots l]$ where l denotes some parametric polynomial whose value will be defined later. The nondeterministic computation branches corresponding to invalid combinations of guessed values (to be discussed later) are terminated through JZERO tests in the deterministic checking phase.

As noted before (Property 2.8) $R_{\text{NORMALIZED}}$ can not use the same value to index multiple features of C_{extended}. For example, distinct values are needed to refer to the ith value in the range $[0 \ldots (l_\exists - 1)]$ and the ith terminal nondeterministic node in C_{extended}. $R_{\text{NORMALIZED}}$ therefore partitions the range of value it can work with, as follows (the technique is similar to that used for Algorithm 2.2).

- The value 0 is reserved for special purpose.

- The values $[1 \ldots \text{exists}_{\text{end}}]$ are reserved to represent the $\exists_1$-guesses of R_{EXTENDED}, where $\text{exists}_{\text{end}} = 1 + h_1(k) \times l_{\exists_1}$. Each value in this range is interpreted as $\langle i, v_i \rangle$ which represents that the ith $\exists_1$-guess is v_i, where $1 \leq i \leq h_1(k)$ and $0 \leq v_i \leq (l_{\exists_1} - 1)$.

- The values $[(1 + \text{exists}_{\text{end}}) \ldots \text{existspair}_{\text{end}}]$ are reserved to represent the combination of values in pairs of $\exists_1$-registers of R_{EXTENDED}, where $\text{existspair}_{\text{end}} = \text{exists}_{\text{end}} + \binom{h_1(k)}{2}(l_{\exists_1})^2$. Each value in this range is interpreted as $\langle i, v_i, j, v_j \rangle$ which represents that the values in the ith and jth $\exists_1$-registers of R_{EXTENDED} are v_i and v_j, respectively.

- The values $[(1+\text{existspair}_{\text{end}}) \ldots \text{node}_{\text{end}}]$ are reserved to represent the branches resulting from the terminal nondeterministic nodes in C_{extended}, where $\text{node}_{\text{end}} = (\text{existspair}_{\text{end}} + m)$. Each value in this range is interpreted as $\langle b_2, \ldots, b_{t'}, x \rangle$, such that $t' \leq t$. Here $y = \langle b_2, \ldots, b_{t'} \rangle$ represents a unique terminal nondeterministic branch at level t', and x represents an execution path for y.

- The values $[(1 + \text{node}_{\text{end}}) \ldots \text{condition}_{\text{end}}]$ are reserved to refer to various constraints. The values in this range are interpreted either as $\text{cond}_{\langle i \rangle}$, $1 \leq i \leq h_1(k)$, or as $\text{cond}_{\langle i,j \rangle}$, $1 \leq i < j \leq h_1(k)$, or as $\text{cond}_{\langle i,v_i,u \rangle}$, $1 \leq i < u \leq h_1(k)$, $0 \leq v_i \leq (l_{\exists_1} - 1)$. Intuitively, $\text{cond}_{\langle i \rangle}$ ($\text{cond}_{\langle i,j \rangle}$) refers to the requirement that some value is chosen for the ith $\exists_1$-register (the pair corresponding to ith and jth $\exists_1$-registers) of R_{EXTENDED}. On the other hand, $\text{cond}_{\langle i,v_i,u \rangle}$ requires that the condition $\langle i, v_i \rangle \Rightarrow \langle i, v_i, u, \star \rangle$ is satisfied. Let $\text{condition}_{\text{end}}$ be equal to $\left(\text{node}_{\text{end}} + h_1 + (l_{\exists_1} + 1)\binom{h_1}{2}\right)$.

We now describe the simulation in detail.

1. **Preprocessing:**
   (a) Partially simulate the computation of R_{EXTENDED} on $\langle x, k \rangle$ and construct the associated partial computation tree C_{extended}. From C_{extended}, compute the range of each nondeterministic guess, an enumeration of the terminal nondeterministic nodes, and an enumeration of the execution paths for each nondeterministic checking branch.

   (b) Based on the partial computation tree C_{extended}, construct a lookup table $\mathcal{T}$ as follows (the reader may jump to step 2 and then refer back to the description of the table entries as needed). An entry $\mathcal{T}[i, j]$ of the lookup table is set to 1 if one of the following conditions holds. $\mathcal{T}[i, j]$ is set to 0 otherwise.

      i. $i = 0$, $\text{existspair}_{\text{end}} + 1 \leq j \leq \text{node}_{\text{end}}$ and j refers to a level-2 terminal universal node in C_{extended}.

ii. $i = 0$, $\text{node}_{\text{end}} + 1 \leq j \leq \text{condition}_{\text{end}}$.

iii. $(\text{existspair}_{\text{end}} + 1) \leq i, j \leq \text{node}_{\text{end}}$, i and j represent terminal nodes x_i and x_j in C_{extended}, and one of the following holds.

- x_i and x_j are at level q and $q + 1$, respectively, for some q, $2 \leq q < t$, and x_i is an ancestor of x_j. In case x_j is the last terminal node in some branch, the execution path associated with x_j must be rejecting and be valid for the terminal node x_j.
- x_i is the last nondeterministic step in some branch and $x_i = x_j$.

iv. $(\text{node}_{\text{end}} + 1) \leq i \leq \text{condition}_{\text{end}}$, and $i = j$.

v. $1 \leq j \leq \text{exists}_{\text{end}}$, $j = \langle u, \star \rangle$ and $i = \text{cond}_{\langle u \rangle}$, where $\star$ is any value in the range for the uth $\exists_1$-guess of R_{EXTENDED}.

vi. $(\text{exists}_{\text{end}} + 1) \leq i \leq \text{existspair}_{\text{end}}$, $i = \langle u, \star, u', \star \rangle$ and $j = \text{cond}_{\langle u, u' \rangle}$, where first and second $\star$ represent any value in the range for the uth and u'th $\exists_1$ guess of R_{EXTENDED}, respectively.

vii. $1 \leq i \leq \text{exists}_{\text{end}}$, $i = \langle u, v'_u \rangle$ and $j = \text{cond}_{\langle u, v_u, u' \rangle}$, and $v_u \neq v'_u$. In this case the precondition of $\langle u, v_u \rangle \Rightarrow \langle u, v_u, u', \star \rangle$ is satisfied.

viii. $(\text{exists}_{\text{end}} + 1) \leq i \leq \text{existspair}_{\text{end}}$, $i = \langle u, v_u, u', \star \rangle$, and $j = \text{cond}_{\langle u, v_u, u' \rangle}$.

ix. $\text{existspair}_{\text{end}} + 1 \leq i \leq \text{node}_{\text{end}}$, $i = \langle \langle \rangle, b \rangle$, b is a rejecting execution path, $(c + 1) \leq j \leq \text{exists}_{\text{end}}$, $j = \langle u, v_u \rangle$, and the outcome of some test along the rejecting execution path b causes the computation to branch away from b if the uth $\exists_1$-guess of R_{EXTENDED} is v_u.

x. $\text{existspair}_{\text{end}} + 1 \leq i \leq \text{node}_{\text{end}}$, $i = \langle \langle \rangle, b \rangle$, b is a rejecting execution path, $(\text{exists}_{\text{end}} + 1) \leq j \leq \text{existspair}_{\text{end}}$, and for the values represented by j for a pair of $\exists_1$-registers of R_{EXTENDED}, the outcome of some test along the rejecting execution path b causes the computation to branch away from b.

Let l be equal to $\text{condition}_{\text{end}}$.

2. **Nondeterministic block 1:**

(a) Existentially guess a value from the range $[0 \ldots 0]$ and store it in g_0.

(b) Existentially guess $h_1(k)$ values from the range $[0 \ldots \text{exists}_{\text{end}}]$ and store them in registers $g_1, \ldots, g_{h_1(k)}$, respectively.

(c) Existentially guess $h_2(k) = \binom{h_1(k)}{2}$ values from the range $[0 \ldots \text{existspair}_{\text{end}}]$ and store them in guess registers $g_{h_1(k)+1}, \ldots, g_{h_1(k)+h_2(k)}$, respectively.

3. **Nondeterministic blocks 2 to t:**

   Starting with a universal guess, perform $t-1$ alternating nondeterministic steps. One value is guessed from the range $[0 \ldots l]$ in each level of alternation. Store these values in guess registers $g_{\text{chk}+1}, \ldots, g_{\text{chk}+t-1}$ where chk equals $(h_1(k) + h_2(k))$.

4. **Deterministic checking:**
   We describe the deterministic checking for even t so that the last nondeterministic step is universal.

   (a) **Checking for the universally guessed value at level-2:**
   If $\mathcal{T}[\langle g_0, g_{\text{chk}+1}\rangle]$ is 0 then accept in (terminate) this universal branch.
   Any branch that remains active is either a valid terminal nondeterministic branch at level-2 of C_{extended} (Lookup table entry type (i)) or a value referring to some condition (Lookup table entry type (ii)).

   (b) **Checking for guesses at levels 3 to t:** For each i, $2 \leq i \leq (t-1)$ verify that $\mathcal{T}[\langle g_{\text{chk}+i-1}, g_{\text{chk}+i}\rangle]$ is 1. Accept in (terminate) this branch otherwise. In any active branch, the value of $g_{\text{chk}+t-1}$
   - will be equal to $g_{\text{chk}+1}$ in case $g_{\text{chk}+1}$ refers to a condition (Lookup table entry type (iv)), or
   - will refer to a nondeterministic checking branch x and an execution path b such that the nondeterministic checking branch x may compute along b (Lookup table entry type (iii)).

   (c) **$\exists_1$-tests:** For each i, $1 \leq i \leq$ chk, if $\mathcal{T}[\langle g_{\text{chk}+t-1}, g_i\rangle]$ is 1 then accept in this universal branch. The following verifications are done through the tests in this step.

   Checking for consistency among the $\exists_1$-values:
   - For each i, $1 \leq i \leq h_1$, a valid $\langle i, v_i\rangle$ is stored in some register g_j, $1 \leq j \leq h_1$. Also the value v_i is within the range of the ith $\exists_1$-guess of R_{EXTENDED}. The condition is ensured by the checking branch having $\text{cond}_{\langle i\rangle}$ in $g_{\text{chk}+t-1}$ and lookup entries of type (v).
   - For each pair $\langle i, j\rangle$, $1 \leq i < j \leq h_1$, a valid $\langle i, v_i, j, v_j\rangle$ is stored in some register g_j, $h_1 + 1 \leq j \leq$ chk. Also the value v_i and v_j are within the range of the ith and jth $\exists_1$-guesses of R_{EXTENDED}, respectively. The condition is ensured by the checking branch having $\text{cond}_{\langle i,j\rangle}$ in $g_{\text{chk}+t-1}$ and lookup entries of type (vi).
   - Verify that the value v_i specified for the ith $\exists_1$-guess of R_{EXTENDED} is consistent in all g_j, $1 \leq j \leq h_1 + h_2$. This condition is ensured by

branches having $g_{\text{chk}+t-1} = \text{cond}_{\langle u,\star,u'\rangle}$ and lookup entries of type (vii) and (viii).

Checking for the execution paths:

Some test outcome along execution path b in R_{EXTENDED} for the selected nondeterministic checking branch, causes the computation to branch away from the rejecting execution path b in case the ith $\exists_1$-guess of R_{EXTENDED} is v_i (lookup entry type (ix)), or ith and jth $\exists_1$-guesses of R_{EXTENDED} are v_i and v_j, respectively (lookup entry type (x)).

(d) Reject in this universal branch.

The proof for odd t follows from duality. For odd t, $R_{\text{NORMALIZED}}$ considers the accepting execution paths instead of rejecting execution paths during the construction of the lookup table.

The normalization for $W[1]$ can be done in a similar fashion. Let R_{BASIC} be the basic $W[1]$-program to be simulated. A normalized $W[1]$-program $R_{\text{NORMALIZED}}$ can existentially guess the k values for the $\exists_1$-phase and the entire sequence of $h(k)$ operations along with the associated register indices for the deterministic checking phase of R_{BASIC}. The rest of the computation of $R_{\text{NORMALIZED}}$ essentially corresponds to verifying that (with appropriate lookup tables) the existentially guessed computation is an accepting computation of R_{BASIC}. We omit the details as they are similar to what we described above.

Resource Usage: We observe that $R_{\text{NORMALIZED}}$ has a unique rejecting execution path in each nondeterministic checking branch. $R_{\text{NORMALIZED}}$ performs $(h_1(k) + \binom{h_1(k)}{2})$ $\exists_1$-tests in each nondeterministic checking branch and accepts in that branch if and only if any of the test outcomes is 1. Also each $\exists_1$-test involves exactly one $\exists_1$-register. ∎

2.5 Equivalence Among the Variants of the Computational Models for $L[t]$

Theorem 2.14 *Let* Q *be a parametric problem and t be a constant, $t \geq 2$. The following are equivalent.*

1. Q *can be decided by a basic $L[t]$-program.*
2. Q *can be decided by an extended $L[t]$-program.*

3. Q *can be decided by a normalized* $L[t]$*-program.*

PROOF. $(3 \Rightarrow 2)$ Immediate as a normalized $L[t]$-program is also an extended $L[t]$-program.

$(2 \Rightarrow 1)$

Let R_{EXTENDED} be an extended $L[t]$-program. We construct a basic $L[t]$-program R_{BASIC} to simulate the computation of R_{EXTENDED} on any input. The key difference between the computation of R_{EXTENDED} and R_{BASIC} is that R_{EXTENDED} can perform a parametric polynomial number of operations on the guess registers whose values have been set in second and subsequent nondeterministic blocks. However, the number of $\exists_1$-operations in any computation branch of R_{EXTENDED} is still bounded by $h(k)$. Define an $\exists_1$-*free block* to be a maximal sequence of operations none of which involves any $\exists_1$-operand. By definition, each nondeterministic checking branch of R_{EXTENDED} can have at most $h(k)$ $\exists_1$-free blocks. R_{BASIC} simulates the computation in all $\exists_1$-free blocks in the preprocessing phase. Since the partial computation tree of R_{EXTENDED} has a parametric polynomial number of branches and the number of $\exists_1$-operation in each branch is at most $h(k)$, R_{BASIC} can compute all necessary values within the time bound on preprocessing phase. The *state* of R_{EXTENDED} at any point of computation is given by the set of values in all its registers at that point of time. R_{BASIC} computes and stores the state of R_{EXTENDED} immediately before each $\exists_1$-operation in each nondeterministic checking branch. R_{BASIC} now starts simulating the computation of R_{EXTENDED}. All nondeterministic guess steps (existential and universal) are simulated directly. R_{BASIC} skips the non-$\exists_1$-operations of R_{EXTENDED}. During the simulation, R_{BASIC} keeps track of the current state of R_{EXTENDED}. When R_{EXTENDED} is about to perform an $\exists_1$-operation, R_{BASIC} first retrieves from the current state, the appropriate values for all necessary operand registers for the $\exists_1$-operation and then performs the simulation. Since any operation of R_{EXTENDED} involves a constant number of registers, simulation of a single $\exists_1$-operation can be done in a constant number of steps of R_{BASIC}. In case the $\exists_1$-operation requires a lookup, R_{BASIC} retrieves the appropriate value from the current state of R_{EXTENDED}.

$(1 \Rightarrow 3)$

We show how $R_{\text{NORMALIZED}}$ simulates the computation of R_{BASIC}. The key distinctions between the two characterizations are as follows.

D1: R_{BASIC} may use nondeterministically guessed values as indices to other registers, which $R_{\text{NORMALIZED}}$ is not allowed to do.

D2: R_{BASIC} may perform any operation on the nondeterministically guessed values while $R_{\text{NORMALIZED}}$ is allowed to perform JEQUAL, JZERO, and ASSIGN operations only, on these values.

D3: R_{BASIC} may have multiple nondeterministic steps in a nondeterministic block in the checking phase. Also, a nondeterministic checking branch of R_{BASIC} may have multiple accepting or rejecting execution paths.

D4: In a computation of R_{BASIC}, deterministic operations that use the nondeterministically guessed values may appear in-between the nondeterministic steps.

$R_{\text{NORMALIZED}}$ can deal with D2 by using the ASSIGN instruction with appropriately constructed lookup tables. For D1, $R_{\text{NORMALIZED}}$ keeps track of the usage of registers in the computation of R_{BASIC} by maintaining a mapping of indices. The rest of the proof elaborates the ideas. $R_{\text{NORMALIZED}}$ deals with D3 by merging all nondeterministic steps in a nondeterministic block into a single (appropriately expanded) nondeterministic step and guessing a deterministic checking branch as part of the last nondeterministic step. D4 can be dealt with by deferring the simulation of deterministic steps that uses nondeterministically guessed values until all the nondeterministic steps are done. The details of the techniques for D3 and D4 are similar to the proof of Theorem 2.13.

Based on the outcomes of JZERO and JEQUAL instructions, there can be at most $2^{h(k)}$ execution paths for each nondeterministic checking branch of R_{BASIC}. Each such execution path can be described by the sequence of instructions I in that path and the registers used in each of the instructions in I. Consider a register r used by some instruction in I. If the value of r does not depend on any $\exists_1$-register, $R_{\text{NORMALIZED}}$ can compute its value deterministically by simulating the sequence of instructions. $R_{\text{NORMALIZED}}$ replaces any such register reference with its computed value. Each of the remaining register references stores a computed value v that depends on some $\exists_1$-register. $R_{\text{NORMALIZED}}$ reserves a set of guess registers $\mathcal{G}_{\text{reserved}}$ that will be used as representatives of the remaining register references in I during the simulation. In addition, $R_{\text{NORMALIZED}}$ maintains a mapping $\langle \text{timestamp}, i_{\text{basic}}, i_{\text{normalized}} \rangle$ to determine which original register is being represented by which guess register in $\mathcal{G}_{\text{reserved}}$. Note that reserving $O(h(k))$ registers suffice as each of the $h(k)$ instructions in I refers to a constant number of registers. $R_{\text{NORMALIZED}}$ initializes $\mathcal{G}_{\text{reserved}}$ so that the first $\exists_1$-register represents itself. Whenever $R_{\text{NORMALIZED}}$ needs a new representative register during the simulation, it picks the next available register from $\mathcal{G}_{\text{reserved}}$ and updates the mapping accordingly. An important feature to note is that the indices of the registers in $\mathcal{G}_{\text{reserved}}$ are stored in standard registers.

Let us consider the simulation of a computation step of R_{BASIC} by $R_{\text{NORMALIZED}}$. Let the step be $r_{i_1} = r_{i_2}$ op r_{i_3} where op is any operation (ADD, for example) in the instruction set of R_{BASIC}. The simulation is done as follows.

- $R_{\text{NORMALIZED}}$ looks for the representatives of r_{i_2} and r_{i_3} in $\mathcal{G}_{\text{reserved}}$ by performing $O(h(k))$ JEQUAL tests.
- If neither of the operands has a representative in $\mathcal{G}_{\text{reserved}}$ then
  - $R_{\text{NORMALIZED}}$ knows the values of the operands and can compute r_{i_2} op r_{i_3} directly.
  - $R_{\text{NORMALIZED}}$ allocates a representative register for r_{i_1} in $\mathcal{G}_{\text{reserved}}$ and stores the computed value in it.
- Otherwise, let $r_{\mathcal{R}(i_2)}$ and $r_{\mathcal{R}(i_3)}$ be the representative registers for r_{i_2} and r_{i_3} respectively. $R_{\text{NORMALIZED}}$ simulates r_{i_2} op r_{i_3} by two indexed assignments as follows.

  $$\text{ASSIGN temp}_1\ c_{\text{op}}\ \mathcal{R}(i_2)$$
  $$\text{ASSIGN temp}_2\ \text{temp}_1\ \mathcal{R}(i_3)$$

  Here, c_{op} is a constant representing the operation op. $R_{\text{NORMALIZED}}$ sets up the lookup tables appropriately in the preprocessing phase so that the indexed assignments compute the desired values.

Note that simulation of the JZERO tests of R_{BASIC} becomes nontrivial as R_{BASIC} may use the $\exists_1$-values to alter the values in the lookup tables. $R_{\text{NORMALIZED}}$ simulates the JZERO tests as follows.

- Compute the mapping $\langle g_i, g_j \rangle$ by two indexed assignments.

  $$\text{ASSIGN temp}_1\ c_{\text{map}}\ \mathcal{R}(i)$$
  $$\text{ASSIGN temp}_2\ \text{temp}_1\ \mathcal{R}(j)$$

  where $g_{\mathcal{R}(i)}$ and $g_{\mathcal{R}(j)}$ are the representative registers of g_i and g_j, respectively and c_{map} is a constant representing the mapping function.
- Try to locate a representative register for $r_{r_{\text{temp}_2}}$ in $\mathcal{G}_{\text{reserved}}$.

- If no representative is found then simulate JZERO directly.

- Otherwise, branch based on whether the representative for $r_{r_{\text{temp}_2}}$ is 0 or not.

$R_{\text{NORMALIZED}}$ simulates the standard branching instructions of R_{BASIC} using JZERO tests in case the branching involves some $\exists_1$-value. Simulation of each step of R_{BASIC} by $R_{\text{NORMALIZED}}$ involves at most two searches for representatives in $\mathcal{G}_{\text{reserved}}$ followed by $O(1)$ steps. Since, size of $\mathcal{G}_{\text{reserved}}$ is $O(h(k))$, simulating each step of R_{BASIC} takes $O(h(k))$ steps of $R_{\text{NORMALIZED}}$. Thus, $R_{\text{NORMALIZED}}$ can perform the entire simulation within the specified bounds. ∎

Chapter 3

Simplified Proofs of the Basic Results in Parameterized Complexity Theory

In this chapter, we develop the basic results in parameterized complexity theory from a computational perspective. These results include the Normalization Theorem and the parameterized version of Cook's Theorem. We show how the machine characterizations can be used to construct simpler proofs of these results. The new proof techniques resemble the classical approach used in the proof of *NP*-completeness of SAT. The $W[t]$-completeness of Antimonotone WCS (respectively, Monotone WCS) for odd (respectively, even) values of t, $t \geq 2$, is obtained as a byproduct of the proof of the Normalization Theorem.

3.1 A Summary of the Original Proofs

Nondeterministic Polynomial-Time Turing Machine Acceptance is the basic problem in *NP*. In the classical context, Cook's Theorem states that Satisfiability is *NP*-complete. In parameterized complexity theory, the analogues of these two problems are Bounded Turing Machine Acceptance and c-CNF Weighted Circuit Satisfiability for constant c, respectively [25].

Bounded Turing Machine Acceptance

Input: A nondeterministic Turing machine M, a string x, a positive integer k.

Parameter: k.
Question: Does M accept x in at most k steps?

c-CNF Weighted Circuit Satisfiability

Input: A c-CNF circuit $\mathcal{C}$, a positive integer k.
Parameter: k.
Question: Does $\mathcal{C}$ have a weight-k satisfying assignment?

The parameterized version of Cook's Theorem states that both these problems are complete for $W[1]$.

Theorem 3.1 *(The parameterized analogue of Cook's Theorem, Downey and Fellows [25])* Bounded Turing Machine Acceptance *and* c-CNF Weighted Circuit Satisfiability *are complete for* $W[1]$.

Downey and Fellows identified the Weighted t-Normalized Satisfiability as a basic complete problem for $W[t]$, $t \geq 2$. This result is stated formally by the Normalization Theorem.

Theorem 3.2 *(The Normalization Theorem, Downey and Fellows [25])*

Weighted t-Normalized Satisfiability *is* $W[t]$*-complete, for* $t \geq 2$.

We give an overview of the original proofs. The reader is referred to the articles [25], [26] and the monograph [28] for details. Recall that the $W[t]$ classes are defined as the closure of WCS on weft-t, constant depth circuits, under FPT reduction. The proofs of both theorems focus on converting weft-t, constant depth circuits into the corresponding normal form, with weight of truth assignment adjusted appropriately.

Proof sketch. (Parameterized analogue of Cook's Theorem.) (Downey and Fellows [26], [28])

The proof goes through several intermediate steps which are described below. $W[1]$-hardness of several other parametric problems are obtained during the course of the proof.

1. Constructing a fixed-parameter reduction to reduce WCS on weft-1, constant depth circuits to WCS on weft-1, depth-2 c-CNF circuits, for some constant $c > 1$. The reduction starts by converting the input circuit into an equivalent tree circuit and moving the *not*-gates to the inputs by applying DeMorgan's law. The resultant circuit is converted into an equivalent normal form of depth-4. The desired weft-1, depth-2 c-CNF form is obtained by multiple change of variables and by introducing additional large *and*-gates for consistency checking.

2. Constructing a fixed-parameter reduction to reduce WCS on weft-1, depth-2 c-CNF circuits to WCS on weft-1, depth-2 *antimonotone* c-CNF circuits. This step introduces the Bounded Degree Red/Blue Nonblocker problem. An instance of the problem on a graph of maximum degree c can be represented by an antimonotone c-CNF circuit, with the same solution size. The purpose of this step is to show that the Bounded Degree Red/Blue Nonblocker problem is hard for WCS on antimonotone c-CNF circuits.

3. Constructing a fixed-parameter reduction to reduce WCS on antimonotone c-CNF circuits to WCS on antimonotone 2-CNF circuits. The reduction essentially makes a change of variables and adjusts the weight of the truth assignment.

4. Constructing a fixed-parameter reduction to reduce WCS on antimonotone 2-CNF circuits to Clique (via Independent Set).

5. Constructing a fixed-parameter reduction to reduce Clique to Bounded Turing Machine Acceptance. Given an instance $\langle x, k\rangle$ of Clique, the reduction constructs a nondeterministic Turing machine M that accepts the empty string in $h(k)$ steps if and only if the input graph has a clique of size k. M nondeterministically guesses k vertices and checks that they form a clique, in $h(k)$ deterministic steps, for some function h.

∎

As is evident from the proof sketch, the focus of the reductions is on the intrinsic details of circuit manipulation and related combinatorics. Based on the $W[1]$-programs, we give a more direct proof that WCS on antimonotone 2-CNF circuits is hard for $W[1]$. We also show how a basic $W[1]$-program can decide WCS on c-CNF circuits.

PROOF SKETCH. (The Normalization Theorem.) (Downey and Fellows [25], [28])

The membership of WEIGHTED t-NORMALIZED SATISFIABILITY in $W[t]$ follows trivially from the circuit characterization. The hardness result is shown by constructing an *FPT* reduction to transform a circuit $\mathcal{C}_{in}$ of weft t and constant depth into an equivalent t-normalized circuit $\mathcal{C}_t$. The major steps of the reduction are as follows.

1. Converting $\mathcal{C}_{in}$ into a tree circuit. Since the depth of $\mathcal{C}_{in}$ is a constant $d \geq t$, this conversion increases the size of $\mathcal{C}_{in}$ by an $O(n^d)$ factor. Also, the *not* gates are moved to the inputs by applying DeMorgan's law.

2. Arranging the large gates so that the output is a large *and* gate and the remaining large gates are arranged in $(t-1)$ alternating levels. Subcircuits consisting of small gates only may appear in-between the alternating levels of large gates. We call them small-intermediate-subcircuits.

3. Converting each small-intermediate-subcircuit C_{small} into DNF or CNF form depending on the type of the large gate to which the output of C_{small} is connected.

4. Merging small gates with large gates of the same type. For other small gates, applying distributive law moves the large gates towards the output. After this rearrangement, any large *and* gate receives inputs only from large *or* gates at the next level, and vice versa. The large gates, closest to the input level, receive inputs from small gates only. We number the levels of large gates so that the output *and* gate is at level 1 and a large gate at level i receives input from large gates at level $(i+1)$, $1 \leq i < t$.

5. Converting each subcircuit of small gates that appears between a large gate at level t and the inputs, into an equivalent normal form (CNF or DNF) of depth two. The output gates of the normal form is merged with the corresponding large gate at level t.

6. Eliminating the extra level of small gates at the input level by introducing new variables and constructing additional circuits to ensure consistency.

∎

The later steps are intricate and they involve multiple change of variables in a complex manner. A significant part of the circuit transformations is related to the

subcircuits between the input level and the large gates that are closest to the input level.

In the next section, we show how to represent computations on a $W[t]$-program by an instance of WEIGHTED t-NORMALIZED SATISFIABILITY. The constructed circuits are monotone for even levels and antimonotone for odd levels of the hierarchy. We also construct $W[t]$-programs that decide WEIGHTED WEFT- t DEPTH-d CIRCUIT SATISFIABILITY. The $(t-1)$ alternating nondeterministic blocks in the checking phase of the $W[t]$-programs deal with the $(t-1)$ levels of large gates (starting from the output) of an weft-t circuit. The remaining level of large gates and the small subcircuits that provide inputs to them, are dealt with in the deterministic part of the checking phase. Necessary lookup tables are constructed in the preprocessing phase of the $W[t]$-programs so that the number of steps in the deterministic checking part remains within the bounds. Although, the $W[t]$-programs apply some of the early transformations from the original proof, the remaining part of the computation is significantly simpler than the corresponding circuit transformations.

Before getting into the details, we would like to define some features of a circuit that will be important for the results in this chapter and Chapter 4.

Definition 3.1 *(Antimonotone group and monotone group) The* antimonotone group *of a sum* $C_{\text{sum}} = (v_{i_1} \vee \ldots \vee v_{i_j} \vee \neg v_{i_{j+1}} \vee \ldots \vee \neg v_{i_m})$ *is the sum of all negative literals* $(\neg v_{i_{j+1}} \vee \ldots \vee \neg v_{i_m})$ *appearing in* C_{sum}. *The* monotone group *of a product* $C_{\text{prod}} = (v_{i_1} \wedge \ldots \wedge v_{i_j} \wedge \neg v_{i_{j+1}} \wedge \ldots \wedge \neg v_{i_m})$ *is the product of all positive literals* $(v_{i_1} \wedge \ldots \wedge v_{i_j})$ *appearing in* C_{prod}.

3.2 A Simplified Proof of the Parameterized Version of Cook's Theorem

The completeness for BOUNDED TURING MACHINE ACCEPTANCE was shown by Chen et al. [20]. The completeness for WEIGHTED ANTIMONOTONE 2-CNF SAT was shown via BOUNDED TURING MACHINE ACCEPTANCE by Cai et al. [13]. These proofs closely follow the classical approach for proving the basic *NP*-completeness results. We reproduce the proofs here as our new proof of the Normalization Theorem will be based on them.

PROOF.(Parameterized analogue of Cook's Theorem) Let Q be any problem in $W[1]$. By definition, there exists a $W[1]$-program R_{Q} that decides Q. The first

part of the proof describes how to construct a nondeterministic Turing machine M such that R_{Q} accepts an input $\langle x, k\rangle$ if and only if M accepts the empty string in $g(k)$ steps, for some function g. The remaining part of the proof is a modified version of the *NP*-completeness proof of SAT (the Cook's Theorem) and describes how to represent the (bounded) computation of a nondeterministic Turing machine by an instance of Antimonotone 2-CNF Weighted Circuit Satisfiability.

We start by constructing a fixed-parameter reduction $\mathcal{A}_1$ to prove the $W[1]$-hardness of Bounded NTM Computation. Given an input $\langle x, k\rangle$, $\mathcal{A}_1$ deterministically simulates the computation in the preprocessing phase of R_{Q} on $\langle x, k\rangle$. This part of the simulation takes parametric polynomial time and it ends when R_{Q} is about to make the first existential guess. Note that the rest of the computation of R_{Q} (in any nondeterministic branch) can have at most $h(k)$ steps, for some function h. $\mathcal{A}_1$ constructs the NTM M to represent the remainder of R_{Q}'s computation. M contains a parametric polynomial number of states to remember the values that R_{Q} would have in the standard registers at the end of preprocessing phase. M has some additional states to store the instructions of R_{Q}. The transition function of M is constructed in such a way that M nondeterministically guesses k values at the beginning. This part corresponds to the first existential block of R_{Q}. M then simulates the computation of R_{Q} on the nondeterministically guessed values. Whenever the value of a standard register is needed for the first time, M retrieves the value from the corresponding state and writes it on the worktape for future use. M accepts if and only if R_{Q} accepts. Simulation of each step of R_{Q} takes $g'(k)$ steps, for some function g'. Thus, M accepts the empty string in $g(k)$ steps, for some function g, if and only if R_{Q} accepts $\langle x, k\rangle$. This part of the proof was originally given by Chen, Flum and Grohe [20].

The second part of the proof is essentially a fixed-parameter reduction from Bounded NTM Computation to WCS on Antimonotone 2-CNF circuits. This part of the proof was originally developed by Cai et al. [13]. The reduction $\mathcal{A}_2$ is a modified version of the classical *NP*-completeness proof of SAT. Given the description of an NTM M and a parameter k, $\mathcal{A}_2$ constructs an antimonotone 2-CNF circuit C such that M accepts the empty string in k steps if and only if C has a satisfying assignment of weight k^2. Instead of specifying valid transitions, C represents all the inconsistent transitions of M and checks that none of the inconsistent transitions is taken by M during the computation. Thus, each clause of C is of the form $(\neg\alpha \vee \neg\beta)$ (i.e. $\alpha \Rightarrow \neg\beta$) where α and β are configurations of M (defined later), such that configuration β cannot follow configuration α. C is constructed as a large product of these antimonotone sums of two literals. We consider the tableau, having k rows and k columns, associated with the computation

of M (for k steps). Let Q be the set of states of M and Σ be the working alphabet. A configuration of M is represented as $Y_{tphq\sigma}$, where

$$1 \leq t \leq k, \quad 1 \leq p \leq k, \quad 1 \leq h \leq k, \quad q \in Q, \quad \sigma \in \Sigma.$$

A configuration $Y_{tphq\sigma}$ represents the fact that at time t, M is in state q, the head is at position h, and the p-th cell stores the symbol σ. Each input variable of C is some configuration $Y_{tphq\sigma}$ of M. A weight k^2 assignment for C is intended to be a description of the tableau of M such that each *true* variable in the assignment corresponds to a unique cell in the tableau. The clauses in C are of three types. Clauses of the first type are related to the mapping of true variables to cells in the tableau. The second type of clauses are related to transitions of M and they ensure that the *true* variables do not represent an inconsistent tableau (i.e. they represent a valid computation of M). The third type of clauses specify the accepting conditions.

The following formula φ_1 is an example of the first type of clauses, where φ_1 expresses that M cannot be in two different states at any particular time t.

$$\varphi_1 = \prod_{1 \leq p,p' \leq k} \prod_{1 \leq h,h' \leq k} \prod_{\sigma,\sigma' \in \Sigma} \prod_{q \neq q' \in Q} \left(\neg Y_{tphq\sigma} \vee \neg Y_{tp'h'q'\sigma'}\right) \ .$$

The following conditions are expressed in a similar manner.

- At time t, the head can not be in two different positions.
- At time t, a given cell can not store two different symbols.

An example of the second type of clauses is φ_2, which states that if at time t the head is not in position p, then the symbol in that position must be the same at time $t+1$.

$$\varphi_2 = \prod_{1 \leq p \leq k} \prod_{q,q' \in Q} \prod_{\substack{\sigma,\sigma' \in \Sigma \\ \sigma \neq \sigma'}} \prod_{\substack{1 \leq h,h' \leq k, \\ h \neq p}} \left(\neg Y_{tphq\sigma} \vee \neg Y_{(t+1)ph'q'\sigma'}\right).$$

The following conditions are expressed in a similar manner.

- If at time t, the state of the machine is q, and symbol under head is σ, and there is no transition rule of the form $(q, \sigma) \rightarrow (q', \sigma', d')$, then
  - the next state can not be q',

- the symbol under head can not be σ' at time $t+1$, and
- the head position can not be $h+d'$ at time $t+1$.

The third type of clauses specify that the computation must end in an accepting state, i.e. q must be an accepting state.

The size of C is parametric polynomial, and $\mathcal{A}_2$ can construct it in parametric polynomial time. By construction, C has a satisfying assignment of weight k^2 if and only if M accepts the empty string in k steps. A satisfying assignment of weight k^2 must set exactly one variable $Y_{tphq\sigma}$ to *true*, for each pair of t and p. This is ensured by the first type of clauses. The second type of clauses ensure that the true variables do not represent any invalid transition. Finally the third type of clauses ensure that the machine ends in an accepting state. Thus a satisfying assignment of weight k^2 implies that M accepts the empty string in k steps. For the reverse direction we note that, given an accepting computation of M, a satisfying assignment of weight-k^2 can be constructed from the content of the tableau. ∎

Theorem 3.3 *A basic $W[1]$-program can decide* WEIGHTED d-CNF SAT*, where d is a constant.*

PROOF. Let $\mathcal{C}$ be a d-CNF circuit on m input variables and we want to determine whether $\mathcal{C}$ has a satisfying assignment of weight k. We construct a basic $W[1]$-program for the purpose. Recall that for a clause $(x_1 \vee \ldots \vee x_i \vee \neg x_{i+1} \vee \ldots \vee \neg x_d)$ of $\mathcal{C}$, $(\neg x_{i+1} \vee \ldots \vee \neg x_d)$ (or equivalently $\neg\,(x_{i+1} \wedge \ldots \wedge x_d)$) is the *antimonotone group* of the clause (Definition 3.1). For a given truth assignment, we say that an antimonotone group is *active* if all its variables are set to *true* (i.e. the antimonotone group is false as a whole). Let us consider an antimonotone group y and the set of clauses S_y having y as the antimonotone group. If y is active for a given assignment then each clause in S_y must be satisfied by some true variable x such that x appears positively in the clause. There are at most d choices for x as $\mathcal{C}$ is a d-CNF circuit. Thus, there can be at most d^k combinations of up to k variables each of which may satisfy all clauses in S_y by the positive literals. We refer to each such combination as a *deactivating* combination for the antimonotone group y.

In order to decide WCS on $\mathcal{C}$, the program existentially guesses the true variables, the active antimonotone groups, and the deactivating combinations. In the checking phase, the program verifies that each active antimonotone group is deactivated by some combination of true variables. In order to perform the checking within the resource bounds, the program constructs necessary lookup tables in

the preprocessing phase. The lookup tables specify whether (i) an antimonotone group is deactivated by a combination of variables, (ii) a given variable appears in an antimonotone group, and (iii) a given variable appears in a given deactivating combination. The details are as follows.

1. **Preprocessing**:
   (a) Enumerate all possible antimonotone groups consisting of up to d variables. There are $O(m^{d+1})$ different antimonotone groups for $\mathcal{C}$.
   (b) Enumerate all deactivating combinations for all antimonotone groups. There are $O(d^k\, m^{d+1})$ such deactivating combinations in total. It is important to note that only a parametric polynomial number of combinations (out of $O(n^k)$ possible combinations of size k are of interest for any given circuit.
   (c) Construct lookup tables to specify (i) consistency among variables and antimonotone groups, (ii) consistency among variables and deactivating combinations, (iii) whether a combination of variables appear as an antimonotone group in $\mathcal{C}$, and (iv) whether a deactivating combination deactivates an antimonotone group.
2. **Nondeterministic block 1:**
   (a) Existentially guess k true variables, and $l = \left(\sum_{i=1}^{d} k^i\right)$ antimonotone groups.
   (b) Existentially guess 2^k deactivating combinations (possibly with repetition) that are satisfied by the k true variables.
3. (a) Verify that the existentially guessed antimonotone groups and the deactivating combinations are consistent with the k true variables. The program needs to use multiple lookup tables to perform the consistency checking. Since a basic $W[1]$-program has complete access to $\exists_1$-values, this can be done in a straightforward way.
   (b) For each existentially guessed antimonotone group y, repeat the following.
      i. Check whether y is deactivated by any of the existentially guessed deactivating groups. Use the lookup tables to perform each checking in constant time.
      ii. If the antimonotone group is not deactivated, then reject. Continue with the next antimonotone group otherwise.

(c) Accept.

Correctness: In step 3(b), the program ensures that any active antimonotone group (that may falsify some clause) is deactivated by some combination of *true* variables. The program accepts only if each active antimonotone group is deactivated which implies that no clause in $\mathcal{C}$ is falsified.

Resource Usage: The nondeterministic steps in Step 2 satisfy Constraints AW3, T1, and TU1 for $t = 1$. The verification in Step 3 can be done in $O(2^k k^d)$ time. The construction of the lookup tables take parametric polynomial time. Thus Constraints AW1 and BL1 are satisfied. The registers need to store indices to input variables, antimonotone groups, and combinations of up to d variables. Each of these values is bounded by some parametric polynomial. Thus Constraint AW2 is also satisfied. ∎

3.3 A Simplified Proof of the Normalization Theorem

The proof of the Normalization Theorem is constructed in two steps. First, we construct a generic reduction based on the extended $W[t]$-programs to show that Weighted t-Normalized Satisfiability is $W[t]$-hard, $t \geq 2$. We then construct an extended $W[t]$-program to decide Weighted Weft-t Depth-d Circuit Satisfiability.

Lemma 3.4 *(Downey, Fellows [25])* Weighted t-Normalized Satisfiability *is hard for* $W[t]$, $t \geq 2$.

Proof.(A new proof based on the extended $W[t]$-programs) We construct a generic fixed-parameter reduction $\mathcal{A}$ to show the desired result. Let Q be any problem decidable by an extended $W[t]$-program R. $\mathcal{A}$ takes an input $\langle x, k\rangle$ and constructs a t-normalized circuit C such that R accepts $\langle x, k\rangle$ if and only if C has a weight-k' satisfying assignment. $\mathcal{A}$ starts by constructing the partial computation tree T_{Q} corresponding to the computation of R on $\langle x, k\rangle$ and then constructs the circuit C based on T_{Q}.

Let S be the set of all nondeterministic nodes x in T_{Q} such that x is the first nondeterministic node at some level of alternation along some branch of T_{Q}. Also let

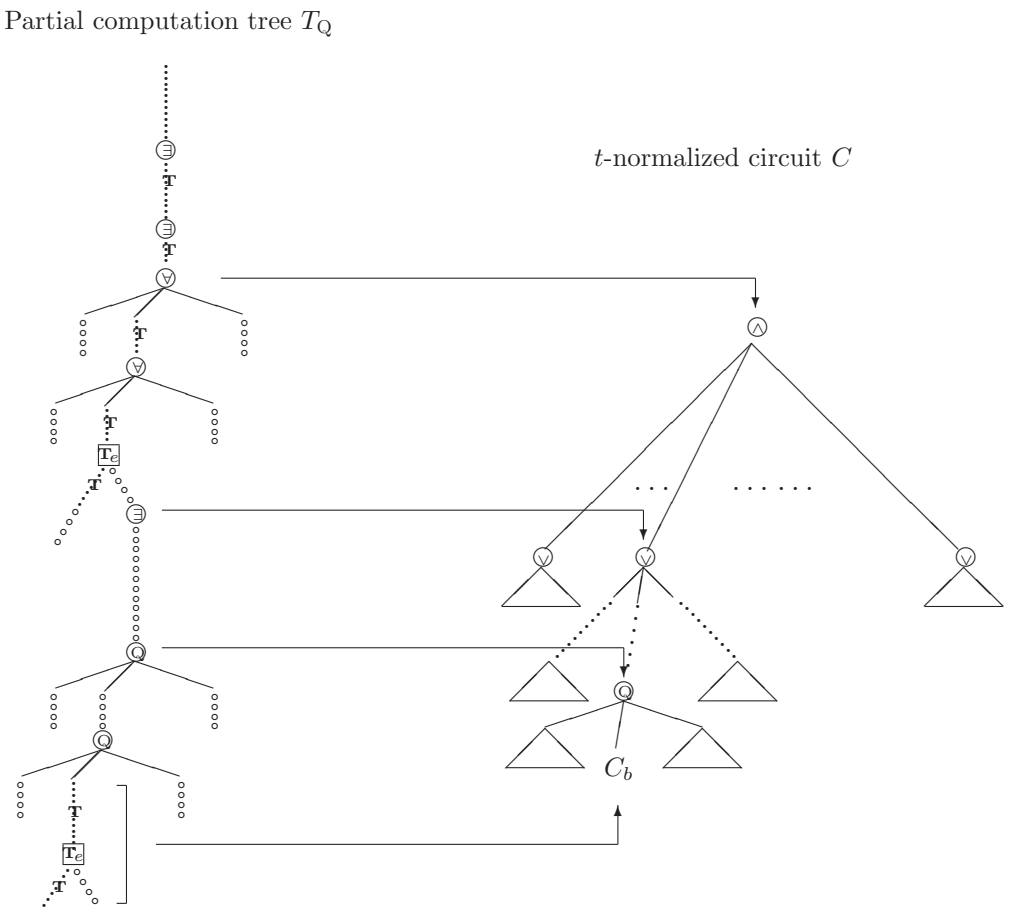


Figure 3.1: Correspondence between the partial computation tree T_Q and the t-normalized circuit C.

L be the set of all nondeterministic nodes y such that y is the last nondeterministic node along some branch of T_Q. C is constructed from T_Q as follows. Figure 3.1 illustrates the correspondence between the partial computation tree T_Q and the components in the circuit C.

1. C includes a subcircuit C_b for each branch b resulting from a node $x \in L$. The subcircuits C_b are constructed so that they satisfy the following lemma.

   Lemma 3.5 *C_b is satisfied if and only if one of the following holds.*

   (a) The nondeterministic step starting b is existential and the computation follows some accepting execution path for nondeterministic checking branch b.

(b) *The nondeterministic step starting b is universal and the computation does not follow any rejecting execution path for nondeterministic checking branch b.*

We describe the construction of C_b later.

2. C includes one gate for each node in S. Let x and y be two nodes in S such that x and y are at alternation levels i and $i+1$ in T_{Q}. Let gates g_x and g_y in C correspond to nodes x and y, respectively. The output of g_y is connected as an input to g_x in case x is an ancestor of y. If x is at the last level of alternation along some branch in T_{Q} then each input of x is connected to the output of some C_b, where b is a branch resulting from a terminal nondeterministic node y that is a descendent of x.

3. The input variables of C are of two kinds. Variables of the first kind are of the form Y_{ij}, $1 \leq i \leq h_1(k)$ and $0 \leq j \leq (fp-1)$. The variable Y_{ij}, when set to *true*, means that the ith $\exists_1$-register of R is the value j. Thus, guessing $h_1(k)$ values in the $\exists_1$-block of R corresponds to setting $h_1(k)$ variables of the first kind to *true*.

   Variables of the second kind are of the form $Y_{iji'j'}$, $1 \leq i < i' \leq h_1(k)$ and $0 \leq j, j' \leq (fp-1)$. The variable $Y_{iji'j'}$, when set to *true*, means that the values in the ith and i'th $\exists_1$-registers of R are j and j', respectively.

We now describe the construction of the subcircuits C_b. The $t-1$ levels of alternation in the checking phase of R_{Q} results in a weft-$(t-1)$ subcircuit whose inputs are connected to the output of C_b subcircuits. Since C has to be t-normalized, the weft of C_b can be at most 2, provided that the output of C_b and the gate to which C_b is connected to, are of the same type. If the last level of alternation for b is existential (respectively universal), C_b is constructed as a 2-normalized DNF (respectively CNF) circuit.

In addition, C_b must satisfy the conditions in Lemma 3.5. Whether the computation follows a given execution path p depends on the outcomes of the binary tests (JEQUAL or JZERO) along p. The computation does not follow p in case at least one test outcome causes the computation to branch away from p. On the other hand, the computation follows p if each test outcome causes the computation to continue along p. C_b specifies these facts in terms of the input variables. Given a desired outcome for a test T along some execution path p for some nondeterministic checking branch b, C_b lists all variables whose corresponding values or value-pairs result in the desired test outcome. For test T, we define a DNF subcircuit σ_T and a

CNF subcircuit π_T as follows. Both σ_T and π_T are constructed in such a way that they are true if and only the outcome of test T causes the computation to branch.

T is JEQUAL i j c :

If g_{r_i} is not an $\exists_1$-register but g_{r_j} is, let

$$\sigma_T = Y_{r_j g_{r_i}} \qquad \text{and} \qquad \pi_T = \prod_{\ell \neq g_{r_i}} \neg Y_{r_j \ell} \ .$$

The reduction computes the value of g_{r_i} from T_{Q} and construct σ_T and π_T accordingly.

If both g_{r_i} and g_{r_j} are $\exists_1$-registers, let

$$\sigma_T = \sum_{0 \leq \ell \leq fp} Y_{r_i \ell\, r_j \ell} \qquad \text{and} \qquad \pi_T = \prod_{0 \leq \ell \neq \ell' \leq fp} \neg Y_{r_i \ell\, r_j \ell'} \ .$$

T is JZERO i j c :

If g_{r_i} is not an $\exists_1$-register and g_{r_j} is an $\exists_1$-register, let

$$\sigma_T = \sum_{\left\{ \ell \mid r_{\langle g_{r_i}, \ell \rangle} = 0 \right\}} Y_{r_j \ell} \qquad \text{and} \qquad \pi_T = \prod_{\left\{ \ell \mid r_{\langle g_{r_i}, \ell \rangle} \neq 0 \right\}} \neg Y_{r_j \ell} \ .$$

The reduction computes the sets $\left\{ \ell \mid r_{\langle g_{r_i}, \ell \rangle} = 0 \right\}$ and $\left\{ \ell \mid r_{\langle g_{r_i}, \ell \rangle} \neq 0 \right\}$ from the partial configuration corresponding to T (Property 2.9 I5).

If both g_{r_i} and g_{r_j} are $\exists_1$-registers, let

$$\sigma_T = \sum_{\left\{ \ell, \ell' \mid r_{\langle \ell, \ell' \rangle} = 0 \right\}} Y_{r_i \ell\, r_j \ell'} \qquad \text{and} \qquad \pi_T = \prod_{\left\{ \ell, \ell' \mid r_{\langle \ell, \ell' \rangle} \neq 0 \right\}} \neg Y_{r_i \ell\, r_j \ell'} \ .$$

The reduction computes the sets $\left\{ \ell, \ell' \mid r_{\langle \ell, \ell' \rangle} = 0 \right\}$ and $\left\{ \ell, \ell' \mid r_{\langle \ell, \ell' \rangle} \neq 0 \right\}$ from the partial configuration corresponding to T (Property 2.9 I5).

In case C_b needs to specify that the outcome of test T does not cause the computation to branch, the reduction uses the subcircuits $\sigma_{\neg T} = \neg \pi_T$ and $\pi_{\neg T} = \neg \sigma_T$ instead of σ_T and π_T.

For a nondeterministic checking branch b whose last level of alternation is universal, the subcircuit C_b is constructed as follows.

$$C_b = \prod_{\substack{\text{rejecting execution} \\ \text{path } p \text{ for } b}} \sum_{\substack{\text{test } T \text{ along} \\ \text{execution path } p}} \psi_T$$

where

$$\psi_T = \begin{cases} \sigma_T & \text{if outcome 0 for } T \text{ causes branching} \\ \sigma_{\neg T} & \text{otherwise} \end{cases}$$

The subcircuit for the other case (i.e. the last level of alternation is existential) has the following form.

$$C_b = \sum_{\substack{\text{accepting execution} \\ \text{path } p \text{ for } b}} \prod_{\substack{\text{test } T \text{ along} \\ \text{execution path } p}} \psi_T$$

where

$$\psi_T = \begin{cases} \pi_T & \text{if outcome 0 for } T \text{ causes branching} \\ \pi_{\neg T} & \text{otherwise} \end{cases}$$

In addition, C has to ensure that (i) at least one Y_{ij} is true for each i, $1 \leq i \leq h_1(k)$, (ii) at least one $Y_{iji'j'}$ is true for each pair i, i', $1 \leq i < i' \leq h_1(k)$, and (iii) the Y_{ij} and $Y_{iji'j'}$ variables are consistent among themselves. For (i) and (ii) C includes a subcircuit $C_{\text{occurrence}}$. For even t, $C_{\text{occurrence}}$ is constructed as a monotone CNF circuit as follows.

$$C_{\text{occurrence}} = \left[\prod_{1 \leq i \leq h_1(k)} \sum_{0 \leq j \leq (fp-1)} Y_{ij} \right] \wedge \left[\prod_{1 \leq i < i' \leq h_1(k)} \sum_{0 \leq j,j' \leq (fp-1)} Y_{iji'j'} \right]$$

For odd t, $C_{\text{occurrence}}$ is constructed as an antimonotone 2-CNF circuit as follows. In this case, the constraints are specified in the form $Y_{ij} \Rightarrow \neg Y_{ij'}$, for all $j' \neq j$.

$$\left[\prod_{\substack{1 \leq i \leq h_1(k) \\ 0 \leq j \leq (fp-1)}} \prod_{\substack{0 \leq j' \leq (fp-1) \\ j \neq j'}} (\neg Y_{ij} \vee \neg Y_{ij'}) \right] \wedge \left[\prod_{\substack{1 \leq i < i' \leq h_1(k) \\ 0 \leq j,j' \leq (fp-1)}} \prod_{\substack{0 \leq v,v' \leq (fp-1) \\ v \neq j \text{ or } v' \neq j'}} (\neg Y_{iji'j'} \vee \neg Y_{ivi'v'}) \right]$$

For (iii) C ensures that the variable $Y_{uvu'v'}$ is *true* only if both Y_{uv} and $Y_{u'v'}$ are *true* (later we show that the if direction follows from other conditions). For odd t, an antimonotone 2-CNF subcircuit $C_{\text{consistency}}$ is constructed for this purpose.

$$C_{\text{consistency}} = \prod_{1 \le i < i' \le h_1(k)} \prod_{\substack{0 \le j,j',l,l' \le (fp-1) \\ j \ne l, j' \ne l'}} [\neg Y_{iji'j'} \vee \neg Y_{il}] \ \wedge \ [\neg Y_{iji'j'} \vee \neg Y_{i'l'}]$$

For even t, $C_{\text{consistency}}$ is constructed as a monotone CNF subcircuit by replacing each negated literal in the expression above by a disjunction of positive literals, as follows.

$$\prod_{1 \le i < i' \le h_1(k)} \ \prod_{\substack{0 \le j,j',l,l' \le (fp-1) \\ j \ne l, j' \ne l'}} \left[\sum_{\substack{u,v,u',v', u \ne i, \\ u' \ne i', v \ne j, v' \ne j'}} Y_{uvu'v'} \vee \sum_{\substack{u,v,u \ne i \\ v \ne l}} Y_{uv} \right]$$

$$\wedge \left[\sum_{\substack{u,v,u',v', u \ne i, \\ u' \ne i', v \ne j, v' \ne j'}} Y_{uvu'v'} \vee \sum_{\substack{u,v,u \ne i' \\ v \ne l'}} Y_{uv} \right]$$

Finally, the output *and*-gates of $C_{\text{occurrence}}$ and $C_{\text{consistency}}$ are merged with the output *and*-gate of C. This finishes the construction of C. The instance of WCS asks for a weight-$\left(h_1(k) + \binom{h_1(k)}{2}\right)$ satisfying assignment for C.

Correctness: By construction, the $t-1$ levels of alternation in the checking phase of R_{Q} results in $t-1$ levels of alternation in C (Figure 3.1). Each C_b is constructed as a 2-normalized circuit and the output gate of C_b is merged with the gate that receives input from C_b. The output *and*-gates of $C_{\text{occurrence}}$ and $C_{\text{consistency}}$ are merged with the output gate of C and hence $C_{\text{occurrence}}$ and $C_{\text{consistency}}$ do not increase weft beyond t. Altogether, C is a t-normalized circuit as required. In addition, the size of C is a parametric polynomial.

The monotone CNF form of $C_{\text{occurrence}}$ requires that at least one Y_{ij} is *true* for each i, $1 \le i \le h_1(k)$ and at least one $Y_{iji'j'}$ is *true* for each pair of i and i', $1 \le i < i' \le h_1(k)$. The antimonotone DNF form of $C_{\text{occurrence}}$ requires that at most one variable is *true* from each group mentioned above. These requirements together with the fact that a satisfying assignment must be of weight-$\left(h_1(k) + \binom{h_1(k)}{2}\right)$ ensure that exactly one variable is true from each group. Thus a satisfying assignment uniquely corresponds to a set of $h_1(k)$ values for the $\exists_1$-guesses of R_{Q}.

Next we show by induction that satisfaction of C corresponds to an accepting computation of R_{Q}. Lemma 3.5 forms the base case.

Proof.(Proof of Lemma 3.5) Satisfaction of C_b implies that the computation for nondeterministic checking branch b follows some accepting execution path for

b or does not follow any rejecting execution path (i.e. the computation accepts for b). In the reverse direction, if R_{Q} accepts along some execution path for b, then the corresponding product of C_b is satisfied in the DNF form or all sums of C_b are satisfied in the CNF form. ∎

Let us consider a gate g at level i in C, where $1 \leq i \leq t-2$. By construction, g corresponds to the first nondeterministic node x at alternation level $i+1$ in some branch of the partial computation tree T_{Q}. Let us consider the case when g is an *and*-gate. By construction, x corresponds to an universal partial configuration. g is satisfied if and only if all subcircuits $C_{g,j}$ providing inputs to g are satisfied. Each $C_{g,j}$ corresponds to the computation starting from a node y_j in T_{Q} such that y_j is the first existential node at $(i+1)$th level of alternation along some branch and x is an ancestor of y_j. Inductively, satisfaction of $C_{g,j}$ implies that y_j is an accepting partial configuration. By property 2.2 satisfaction of g implies x is an accepting partial configuration.

Thus the output gate of C is satisfied only if R_{Q} accepts for the $\exists_1$-values represented by the *true* Y_{ij} variables.

In the other direction, an accepting computation of R_{Q} results in a weight-$\left(h_1(k) + \binom{h_1(k)}{2}\right)$ satisfying assignment for C. The assignment is computed by setting Y_{ij} to *true* if and only if ith $\exists_1$-guess of R_{Q} is the value j. Also, $Y_{iji'j'}$ is set to *true* if and only if the i-th and i'th $\exists_1$-guesses of R_{Q} are j and j', respectively. The assignment satisfies $C_{\mathrm{occurrence}}$ and $C_{\mathrm{consistency}}$. Each C_b is also satisfied as the $\exists_1$-values corresponding to the true variables cause the computation to follow some accepting execution path for b or to branch away from all rejecting execution paths for b. ∎

Remark. Note that the t-normalized circuits, constructed by the reduction $\mathcal{A}$, are monotone for even t and are antimonotone for odd t, $t \geq 2$. Thus the construction serves as a proof of the $W[t]$-hardness of Weighted Monotone Circuit Satisfiability (Weighted Anti-monotone Circuit Satisfiability) for even (odd) t, $t \geq 2$.

We now show that an extended $W[t]$-program can decide Weighted Weft-t Depth-d Circuit Satisfiability, $t \geq 2$. Note that t-normalized circuits are special forms of weft-t, constant depth circuits.

Lemma 3.6 Weighted Weft-t Depth-d Circuit Satisfiability *can be decided by an extended $W[t]$-program, $t \geq 1$.*

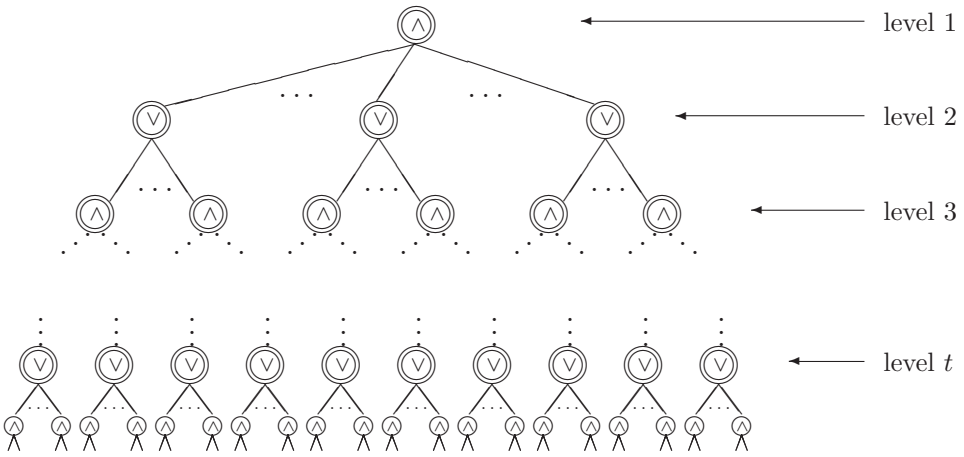


Figure 3.2: The structure of the circuit obtained after preprocessing the input circuit (for even t).

PROOF. We construct an extended $W[t]$-program $R_{\text{WCS-}t\text{-}d}$ to decide WEIGHTED WEFT- t DEPTH- d CIRCUIT SATISFIABILITY. Let $\mathcal{C}_{in}$ be the input circuit. $R_{\text{WCS-}t\text{-}d}$ transforms $\mathcal{C}_{in}$ into an equivalent tree circuit where the large gates are arranged in alternating levels (starting from the output) with small subcircuits appearing at the input level. Figure 3.2 illustrates the structure of the circuit after the preprocessing is done. This transformation is similar to the first part (Steps 1-3 in the proof sketch) of the original proof (Theorem 3.2) and is performed in the preprocessing phase of $R_{\text{WCS-}t\text{-}d}$. After the transformation, $\mathcal{C}_{in}$ has an *and*-gate at the output, followed by $(t-1)$ alternating levels of large *or*-gates and large *and*-gates arranged as in a t-normalized circuit. The large gates closest to the input level receive inputs from small subcircuits. We number the levels of large gates so that the output *and*-gate is at level 1, the large gates that are closest to the input, are at level t, and a large gate at level i receives input from the large gates at level $(i+1)$, $1 \leq i < t$. Each input to a large gate at level t is either a product of c literals or a sum of c literals, where $c > 0$ is a constant.

$R_{\text{WCS-}t\text{-}d}$ existentially guesses the k true variables in the first existential block. With the remaining $t-1$ alternations, $R_{\text{WCS-}t\text{-}d}$ can select a c-CNF (c-DNF) subcircuit at the input level, for odd (even) t. $R_{\text{WCS-}t\text{-}d}$ needs to verify that the $\exists_1$-guesses satisfy (or do not falsify) the selected subcircuit in the final deterministic checking phase. The techniques to deal with c-CNF subcircuits have already been described in the proof of Theorem 3.3. Since an extended $W[t]$-program, $t \geq 2$ can con-

struct multiple lookup tables in the deterministic checking phase, the construction of Theorem 3.3 can be implemented directly by an extended $W[t]$-program. The techniques for c-DNF subcircuits essentially follow from duality. However, we describe them for completeness. Let $\mathcal{D}$ be a c-DNF subcircuit at the input level, and (in some nondeterministic checking branch) $R_{\text{WCS-}t\text{-}d}$ needs to verify whether the $\exists_1$-guesses satisfy $\mathcal{D}$. Let $(x_1 \wedge \ldots \wedge x_i \wedge \neg x_{i+1} \wedge \ldots \wedge \neg x_c)$ be a product in $\mathcal{D}$. Here $(x_1 \wedge \ldots \wedge x_i)$ is the *monotone group* of the product (Definition 3.1). The subcircuit $\mathcal{D}$ is satisfied if at least one of its products is satisfied. A product is satisfied if all variables in its monotone group are *true* and the remaining variables (in this product) are *false*. As before (proof of Theorem 3.3), we say that, for a given truth assignment, a monotone group is *active* if all its variables are *true* in the assignment. Let S_y be the set of products having y as their monotone group. A product in S_y is not satisfied if there is a variable x that appears as a negative literal in the product and x is set to *true*. Since $\mathcal{D}$ is a c-DNF circuit, there are at most c ways to falsify a product having an active monotone group. Thus, there are $O(c^k)$ ways to falsify all products in S_y by their negative literals. As before, we refer to each falsifying combination of variables as a *deactivating* combination for y. $R_{\text{WCS-}t\text{-}d}$ decides Weighted Weft-t Depth-d Circuit Satisfiability as follows.

1. **Preprocessing:**
   (a) Transform input circuit $\mathcal{C}'$ into an equivalent circuit $\mathcal{C}$ so that $\mathcal{C}$ consists of at most t alternating levels of large gates with (possibly) an additional level of small gates at the input level.
   (b) Enumerate all monotone (or antimonotone) groups. For each such group, enumerate the associated deactivating groups.
2. **Nondeterministic block 1:** Existentially guess k true variables, necessary monotone (antimonotone) groups, and the deactivating combinations.
3. **Checking:**
   (a) **Nondeterministic block 2 - t:** Nondeterministically select a c-CNF (or c-DNF) subcircuit at the input level using at most $t-1$ alternating nondeterministic blocks.
   (b) Verify that (i) the $\exists_1$-guesses are consistent among themselves, (ii) the antimonotone groups (monotone groups) together with the deactivating combinations satisfy all clauses (some product) in the chosen c-CNF

(c-DNF) subcircuit. Construct lookup tables as needed during the verification.

The correctness of the algorithm follows from arguments similar to that in the proof of Theorem 3.3 and the discussion above. ∎

3.4 Antimonotone-$W[2t]$ is in $W[2t-1]$, Monotone-$W[2t+1]$ is in $W[2t]$

The monotone (antimonotone) collapse at odd (even) levels of the W-hierarchy is a fundamental structural result in parameterized complexity theory.

Theorem 3.7 *(Downey and Fellows [28])*

(i) Monotone-$W[2t+1] = W[2t]$, $t \geq 0$.

(ii) Antimonotone-$W[2t+2] = W[2t+1]$, $t \geq 0$.

Once again, the original proofs of the results were constructed in the context of circuits and involved circuit manipulations similar to the case of the Normalization Theorem [28]. Here we present new proofs of the same results. The extended machine characterization allows us to construct the proofs from an algorithmic point of view.

Proof.(A new proof in the context of machine characterization) We describe the proof for part (i). The proof for part (ii) follows from duality. Let C' be the monotone weft-$(2t+1)$ input circuit and we want to decide whether C' has a satisfying assignment of weight k. As was done in the proof of Theorem 3.6, C' can be transformed into an equivalent circuit C having at most $(2t+1)$ alternating levels of large gates with an *and*-gate at the output and an additional level of small *or*-gates at the input level. Let the fan-in of any small *or*-gate is bounded by the constant $c > 0$. Consider a CNF subcircuit $C_\wedge$ at the input level, where $C_\wedge$ consists of a large *and*-gate and unbounded number of small *or*-gates having positive literals only.

$$C_\wedge = \bigwedge_{i=1}^{fp} (x_{i,1} \vee \ldots \vee x_{i,c_i}), \qquad c_i \leq c$$

All the sums in $C_\wedge$ must be satisfied in order to satisfy $C_\wedge$. Any given sum can be satisfied by setting one of its (at most) c literals to *true*. Thus, there are $O(c^k)$ ways to satisfy $C_\wedge$ by a weight-k assignment. Enumerate the satisfying combinations for all such CNF subcircuits at the input level of C (there are $O(c^k n)$ of them). Consider a subcircuit consisting of a large *or*-gate $g_\vee$ at level $2t$ and all c-CNF subcircuits providing an input to $g_\vee$. The *or*-gate $g_\vee$ can be satisfied by satisfying one of its input c-CNF subcircuits which in turn can be satisfied by one of $O(c^k)$ associated satisfying combinations. An extended $W[t]$-program can decide whether C' has a weight-k satisfying assignment by (i) constructing a lookup table to specify the combinations that satisfy a given *or*-gate at level $2t$, (ii) existentially guessing k true variables and the associated combinations, (iii) selecting a large *or*-gate at level $2t$ using at most $(2t-1)$ additional alternations, and (iv) verifying whether the selected *or*-gate is satisfied by the combinations guessed in the $\exists_1$-block. We omit the details as they are similar to that in the proof of Lemma 3.6. ∎

Chapter 4

Structural Results

In this chapter, we present some new structural results based on the machine characterization of the W-hierarchy and the L-hierarchy. The results presented in Section 3.4 are the only known structural relationships between $W[t]$ and $W[t+1]$, $t \geq 1$. Our first set of results show that certain nontrivial subclasses of $W[t']$ are contained in $W[t]$, $2 \leq t < t'$. The remaining results place certain nontrivial subclasses of $L[t]$ into $W[t]$, $t \geq 2$. We use the extended $W[t]$-programs to implement some algorithmic techniques in a natural way to show the containment results. Although it is possible to establish the same results using the original circuit characterization or the basic $W[t]$-programs because of the equivalence among the models, the extended $W[t]$-programs make the process much easier.

The following theorem gives an important bound that we will use frequently in the rest of the thesis.

Theorem 4.1 *(Downey et al. [24]) Let f and h be fixed functions, α be a nondecreasing function, and c be a constant. The expression $n^{c/f(k)+h(k)/\alpha(n)}$ is bounded above by a parametric polynomial.*

Cai and Juedes investigated the effect of the existence of algorithms having similar upper bounds [15]. Our motivation for considering such upper bound comes from the following lemma.

Lemma 4.2 *Let f, g, and h be fixed functions, α be a nondecreasing function, and c be a constant. Let R be an extended $W[t]$-program such that R makes at most $f(k)$ $\exists_1$-guesses in any computation branch. Let L be a list containing at most $g(k)n^{c/f(k)+h(k)/\alpha(n)}$ values. A computation branch of R can determine which of the values in L have been guessed in the $\exists_1$-guess steps.*

PROOF. Let us consider the computation along some nondeterministic branch of an extended $W[t]$-program R. Let $f(k)$ be the maximum number of $\exists_1$-guesses performed by R in any computation branch and L be a list as specified in the lemma. R sorts L and performs a binary search in L in order to determine whether the ith $\exists_1$-guess occurs in L, for each i, $1 \leq i \leq f(k)$. R constructs a lookup table T to specify whether a given value is greater than or less than the jth element in L (after sorting). Formally, an entry of T is defined as

$$T[\langle v, j \rangle] = \begin{cases} 1, & \text{if } v \leq L[j] \\ 0, & \text{otherwise} \end{cases}$$

where $1 \leq j \leq |L|$ and v is any value that R may store in a register. R now implements the standard binary search algorithm. During the search, R uses JZERO tests with T as the lookup table to perform comparisons. The total number of $\exists_1$-tests is bounded above by $f(k) \log (g(k) n^{c/f(k) + h(k)/\alpha(n)})$ which is $O(f(k) \log (g'(k) n^d)$ by Theorem 4.1, for some suitable function g' and constant d. ∎

4.1 Relations between Classes in the W-hierarchy

The results in this section show that WCS on certain restricted families of weft-t' constant-depth circuits can be decided by extended $W[t]$-programs, where $2 \leq t < t'$. The results, therefore, imply that the corresponding subclasses of $W[t']$ are contained in $W[t]$. We restrict various properties (number of variables, for example) of the subcircuits at the input level and show that the degree of intractability under the restrictions is lower than that in the unrestricted case.

4.1.1 Number of Input Variables in the Sub-circuits at Input Level

Theorem 4.3 *Let $n_v = \{n^{c/k + h(k)/\alpha(n)}\}$, where $c > 0$ is a constant, h be any function, and α be any unbounded nondecreasing function. Let $\mathcal{F}_1[2]$ be the family of circuits of the form $(C_1 \wedge \ldots \wedge C_m)$ such that each sub-circuit C_i has at most $f(k) n_v$ variables as input, where f is any function. The parameterized problem* WCS *on $\mathcal{F}_1[2]$ is in $W[2]$.*

PROOF. Let C be a circuit in $\mathcal{F}_1[2]$. An extended-$W[2]$-program can decide WCS on C as follows.

1. **Nondeterministic block 1: Existentially** guess k true variables $\{v_1, \ldots, v_k\}$ for C.

2. **Nondeterministic block 2: Universally** select a sub-circuit C_i.

3. Form a sorted list L_i of the $f(k)n_v$ variables of C_i.

4. Perform a binary search in L_i for each v_j, $1 \leq j \leq k$, to determine whether v_j appears in L_i. Mark variable found in L_i as true; mark each of the other variables as false.

5. Deterministically evaluate the circuit C_i on the truth assignment computed for the variables in L_i.

Correctness: By construction, the algorithm accepts if and only if the checking phase accepts for all subcircuits C_i, $1 \leq i \leq m$. For each variable v appearing in C_i, Step 4 of the algorithm determines the truth value of v based on the guesses made in Step 1. Step 5 accepts if and only if C_i is satisfied by the truth assignment. Thus the algorithm accepts the input if and only if $C = (C_1 \wedge \ldots \wedge C_m)$ is satisfied by the existentially guessed weight-k assignment.

Resource Usage: Let us analyze the resource requirements for the algorithm described above. The nondeterministic operations in Steps 1 and 2 satisfy the constraints AW3, T1, and TU1 for $t = 2$. Only Step 4 requires the use of JZERO tests. Each binary search requires $\log(f(k)n_v)$ $\exists_1$-tests, resulting in a total of $k \log f(k) + k \log n_v$ $\exists_1$-tests. Since

$$\begin{aligned} k \log n_v &= k\left(\frac{c}{k}\log n + \frac{h(k)}{\alpha(n)}\log n\right) \\ &= c\log n + kh(k)\cdot\frac{\log n}{\alpha(n)} \\ &\leq c\log n + h'(k) + \log n, \\ &\qquad \text{where } h'(k) = \frac{kh(k)}{\alpha(n_k)}\log n_k, n_k = \alpha^{-1}(kh(k)) \ , \end{aligned}$$

the program meets the required bound on the number of $\exists_1$-tests (Constraint EW1). Step 3 and Step 5 can be done in parametric polynomial time using operations that do not access the $\exists_1$-registers. Thus the algorithm satisfies the Constraint AW1. The algorithm needs to store indices to the input variables, subcircuits, and the

lists L_i and each such index is at most a parametric polynomial (Constraint AW2). ∎

REMARK. Note that the program does not have sufficient resources to check separately at each occurrence of a variable in C_i whether that variable was guessed true—to do so would require too many accesses to the guess registers. The program does have time to try each possible assignment to the variables in L_i, to determine whether C_i is satisfiable, but to do so would not solve the problem at hand, which requires a common assignment to satisfy all sub-circuits.

The extended $W[2]$-program described in the proof above does not use any property that is specific to $W[2]$. Thus the result extends to higher classes in the W-hierarchy.

Corollary 4.4 *For $t \geq 2$, let the circuit family $\mathcal{F}_1[t]$ consist of the circuits of the form*

$$\bigwedge_{i_1} \bigvee_{i_2} \ldots Q_{i_t}\ C(i_1, i_2, \ldots, i_t),$$

where each $C(i_1, i_2, \ldots, i_t)$ is a circuit of unbounded size with at most $f'(k)n_v$ input variables, Q_{i_t} is an and*-gate for even t and an* or*-gate for odd t. An extended $W[t]$-program can decide* WCS *on $\mathcal{F}_1[t]$.*

4.1.2 Number of Monotone Groups and Number of Variables in Each Monotone Group

We start by showing the desired result for the special case of $t = 3$. We then extend the result for higher levels. Let $C = (C_1 \wedge \ldots \wedge C_m)$ be a 3-normalized circuit. Consider a DNF subcircuit $C_\vee$ of C. Let $(l_1 \wedge \ldots \wedge l_r \wedge \neg l_{r+1} \wedge \ldots \wedge \neg l_{m'})$, where $1 \leq r \leq k$, be a product appearing in $C_\vee$ [1]. We say that $(l_1 \wedge \ldots \wedge l_r)$ is the *monotone group* of the product (Definition 3.1).

Theorem 4.5 *Let h be an arbitrary function and let α be any unbounded non-decreasing function. Let $\mathcal{F}_2[2]$ be the family of 3-normalized circuits of the form $(C_1 \wedge \ldots \wedge C_m)$, $m \leq n$ such that each DNF circuit C_i, $1 \leq i \leq m$, separately satisfies the following conditions.*

[1] Products with more than k positive literals cannot be satisfied by a weight-k truth assignment.

1. *The number of distinct monotone groups (of any size) appearing in C_i is bounded by $n^{c/2^k + h(k)/\alpha(n)}$.*

2. *For each monotone group x, there are at most $n^{c/k + h(k)/\alpha(n)}$ variables appearing in all products with monotone group x in C_i.*

The parameterized problem WCS *on* $\mathcal{F}_2[2]$ *is in* $W[2]$.

PROOF. Let C be a circuit in $\mathcal{F}_2[2]$. We construct an extended $W[2]$-program to decide WCS on C, as follows.

1. **Preprocessing:** Construct an enumeration M of all monotone groups appearing in the input circuit C. Note that the number of such monotone groups ($|M|$) is bounded by the length of the input.

2. **Nondeterministic block 1:** Existentially guess k *true* variables. Let V_p be the set of these k variables.

3. **Nondeterministic block 1:** Existentially guess the number $l \leq 2^k$ of monotone groups that are satisfied by the variables in V_p .

4. **Nondeterministic block 1:** Existentially guess $l \leq 2^k$ monotone groups from M, that are satisfied by the assignment. Let $M' \subseteq M$ be the set of these monotone groups. Ensure consistency between V_p and M' by performing $O(k2^k)$ $\exists_1$-tests and an appropriately constructed lookup table.

5. **Nondeterministic block 2:** Universally select a monotone group $x \in M$.

6. **Nondeterministic block 2:** Universally select a subcircuit C_i.

7. Deterministically check that either (i) x is in M' or (ii) x is inconsistent with the variables in V_p (i.e., ensure that M' is maximal). If x is not in M' and x is consistent with the variables in V_p, then *accept* in (i.e. terminate) this computation branch. Checking (i) can be done using JEQUAL tests. For checking (ii), the program computes at most k variables that constitute x, and then checks that at least one of these variables is not included in V_p.

8. Let $M_i \subseteq M$ be the list of monotone groups that appear in C_i. Perform l binary searches on M_i to identify the satisfied monotone groups in M_i. Let S_i be the set of monotone groups in M_i that are satisfied by the existentially guessed assignment ($S_i = M_i \cap M'$).

9. For each monotone group y in S_i, let V_{iy} be the set of input variables that appear in any product of C_i with y as the monotone group.

   (a) Perform k binary searches to determine which of the k variables in V_p are members of V_{iy}.

   (b) Evaluate the products with monotone group y on the (partially) computed assignment. Accept if any such product is satisfied.

Correctness: By the universal selection of a subcircuit C_i in Step 6, the program accepts if and only if the checking phase accepts for all subcircuits C_i in C. Satisfying any product in C_i is sufficient as C_i is a DNF circuit. Step 8 identifies the set of monotone groups of C_i that has been satisfied by the existentially guessed weight-k assignment. Step 9 tries to determine whether any product whose monotone group has been satisfied is not falsified by the negated literals. If at least one such product is found then the checking phase accepts. The verification in Steps 5 and 7 ensures that M' includes exactly those monotone groups that are satisfied by the existentially guessed weight-k assignment. Thus the checking phase accepts if and only if some product in C_i has its monotone group satisfied and has all of the negated variables in the product set to false. This is a necessary and sufficient condition for satisfaction of the DNF subcircuit C_i.

Resource Usage: The nondeterministic operations satisfy the Constraints AW3, T1 and TU1 for $t = 2$. The construction of enumeration M in Step 1 takes polynomial time as the number of monotone groups (i.e. $|M|$) is $O(n)$. The checking in Step 7 can be done using $O(l + k^2)$ $\exists_1$-tests. Note that the universally guessed value x is accessed directly in order to compute the variables that constitute x. The l binary searches in Step 8 requires $c \log n + o(lh(k) \log n)$ $\exists_1$-tests as $|M_i|$ is $n^{c/2^k + h(k)/\alpha(n)}$. The $O(kl)$ binary searches in Step 9(a) require $lc \log n + o(lkh(k) \log n)$ $\exists_1$-tests as $|V_{iy}| \leq n^{c/k + h(k)/\alpha(n)}$ and at most l monotone groups can be satisfied by a weight-k assignment. Evaluation of a subcircuit C_i in Step 9(b) can be done in polynomial time without performing any $\exists_1$-test. Thus the program satisfies Constraints AW1 and EW1. The program needs to store indices to variables, subcircuits, and the monotone groups. As each of these indices is bounded by n, Constraint $AW2$ is also satisfied. ∎

In order to extend the results for odd levels in the W-hierarchy, we use the notion of antimonotone groups (Definition 3.1).

Corollary 4.6 *Let h be any arbitrary function and α be any unbounded nonde-*

creasing function. Let $\mathcal{F}_2[t]$, $t \geq 3$, *be a family of circuits of the form*

$$\bigwedge_{i_1} \bigvee_{i_2} \ldots Q_{i_{t-2}}\ C(i_1, i_2, \ldots, i_{t-2}),$$

such that for even t *(respectively, odd* t*),* Q *is an* and*-gate (respectively,* or*-gate) and* $C(i_1, i_2, \ldots, i_{t-2})$ *is a DNF circuit (respectively, CNF circuit) satisfying the following conditions.*

1. *The number of distinct monotone groups (anti-monotone groups) appearing in* $C(i_1, i_2, \ldots, i_{t-2})$ *is bounded by* $n^{c/2^k + h(k)/\alpha(n)}$*, for even* t *(odd* t*).*
2. *For each monotone group (anti-monotone group)* x*, there are at most* $n^{c/k + h(k)/\alpha(n)}$ *variables appearing in all products (sums) in* $C(i_1, i_2, \ldots, i_{t-2})$ *with monotone group (anti-monotone group)* x*, for even* t *(odd* t*).*

The parameterized problem WCS *on* $\mathcal{F}_2[t]$ *is in* $W[t]$*.*

PROOF. The $W[2]$-program, constructed in the proof of Theorem 4.5, can be easily adapted for higher (even) levels in the W-hierarchy. Only Step 7 needs to be changed to deal with the additional levels of alternation (instead of a single universal branching). The result for odd levels follows from duality. ∎

4.2 Relations between $W[t]$ and $L[t]$

4.2.1 Range of Values

Theorem 4.7 *Let* R_{L} *be a basic* $L[t]$*-program such that in any computation of* R_{L}*, the* $\exists_1$*-operations are restricted to tests and* ASSIGN *only. An extended* $W[t]$*-program* R_{W} *can simulate any computation of* R_{L}*, if the value computed by any* ASSIGN *operation in the computation of* R_{L} *is at most* $f(k)n^{h(k)/\alpha(n)}$*, where* h *is any fixed function and* α *is any unbounded, nondecreasing function.*

PROOF. We prove the result for $t = 2$ by constructing an extended $W[2]$-program to simulate the computation of a restricted $L[2]$-program (as specified in the theorem). Generalization to higher levels is straightforward.

Given any input $\langle x, k\rangle$ the extended $W[2]$-program R_{W} simulates the computation of the restricted $L[2]$-program R_{L} on $\langle x, k\rangle$ as follows.

1. **Preprocessing:** Simulate the preprocessing phase of R_{L} directly. Also construct the partial computation tree C_L of R_{L} for the given input.

2. **Nondeterministic block 1:** Simulate the first existential block directly. Let $V_{\exists_1}$ be the set of existentially guessed values.

3. **Nondeterministic block 1:** Existentially guess all possible pairs of values resulting from $V_{\exists_1}$. Check for consistency with appropriate lookup tables. Let $V_{\exists_1\text{-pair}}$ be the set of pairs of values.

4. **Nondeterministic block 2:** Simulate the universal guesses directly.

5. By definition of the ASSIGN operation, the values in the standard registers specify the result of the operation for any pair of values in its operands. Thus, from the partial computation tree C_L, R_{W} can determine the value that R_{L} would compute by an ASSIGN operation on any pair of values. R_{W} constructs a lookup table T_{assign} to store individual bits of the results of ASSIGN operation, as follows.

$$T_{\mathrm{assign}}[b, \langle u, v\rangle] = \begin{cases} 0, & \text{if the } b\text{-th bit of the result of ASSIGN operation} \\ & \text{on values } u \text{ and } v \text{ is } 0 \\ 1, & \text{if the } b\text{-th bit of the result of ASSIGN operation} \\ & \text{on values } u \text{ and } v \text{ is } 1 \end{cases}$$

   Also, $T_{\mathrm{assign}}[b, \langle u, v\rangle]$ is the same as $T_{\mathrm{assign}}[\langle b, u\rangle, v]$ and $T_{\mathrm{assign}}[u, \langle b, v\rangle]$. Here, the value of b is at most $(\log f(k) + h(k) \log n/\alpha(n))$.

6. Directly simulate all operations, except ASSIGN operations in the checking phase.

7. Simulate each ASSIGN operation by $\log f(k) + h(k) \log n/\alpha(n)$ JZERO tests, each computing one bit of the result.

   - Let the ASSIGN operation be performed on two $\exists_1$-registers storing the values u and v, respectively. A register g in $V_{\exists_1\text{-pair}}$ stores the value $\langle u, v\rangle$ (Step 3). R_{W} performs a JZERO test on b and $\langle u, v\rangle$ to retrieve the b-th bit of the result of ASSIGN from T_{assign}.
   - Let, between the operands of the ASSIGN operation, at least one (say register r) is not an $\exists_1$-operand. Let the value in r be u. R_{W} changes[2]

[2] R_{W} has direct access to any register that is not an $\exists_1$-operand.

the value of r to $\langle b, u\rangle$ and then performs a JZERO test on r and the other operand in order to retrieve the b-th bit of the result of the ASSIGN operation.

Combine the computed bits to construct the complete result of the ASSIGN operation.

Correctness: R_{W} simulates all but the ASSIGN operations of R_{L} directly. The result of each ASSIGN operation of R_{L} is computed one bit at a time and then combined to construct the complete result (Step 7). Thus R_{W} accepts a given input $\langle x, k\rangle$ if and only if R_{L} accepts $\langle x, k\rangle$.

Resource Usage: The nondeterministic operations in steps 2, 3, and 4 satisfy the Constraints $AW3$, $T1$, and $TU1$ for $t = 2$. The simulation of the preprocessing phase of R_{L} and the construction of the partial computation tree C_L takes parametric-polynomial time (Property 2.12). Let l be the maximum value that R_{L} may store in any register at any point of computation. The lookup table T has $O((\log f(k)+h(k)\log n/\alpha(n))l^2)$ entries each storing one bit's worth of information. Each entry can be computed from the corresponding partial configuration in $O(1)$ time. Thus the construction of the entire table takes parametric polynomial time. Simulation of each ASSIGN operation require $(\log f(k) + h(k)\log n/\alpha(n))$ JZERO tests. Thus, all ASSIGN operations can be simulated using at most $h'(k)(\log f(k) + h(k)\log n/\alpha(n))$ JZERO tests, where $h'(k)$ is the number of steps in the final checking phase of R_{L}. The remaining steps (2, 3, 4 and 6) take $O(h'(k))$ time. Thus Constraints $AW1$ and $EW1$ are satisfied. Finally, Constraint $AW2$ is also satisfied as the maximum value that R_{W} needs to store is an index to T.

For the general case, Step 4 needs to be modified so that the program simulates all nondeterministic operations at multiple levels of alternation (as opposed to universal operations at second level of alternation only).
∎

Corollary 4.8 *Let τ be a vocabulary with binary relations and binary functions such that the range of each function is a set of size at most $f(k)n^{h(k)/\alpha(n)}$. For all $t \geq 1$, the parameterized* Model Checking *problem on vocabulary τ and the class of all $\Sigma_{t,u}$ formulas is in $W[t]$.*

Proof. The restricted version of the parameterized Model Checking problem (as specified in the corollary) can be decided by a restricted $L[t]$-program R_{L} such

that the value computed by any ASSIGN operation is at most $f(k)n^{h(k)/\alpha(n)}$. R_{L} uses tests (JEQUAL and JZERO) and ASSIGN operations to compute the relations and the functions respectively. The bound on the range of the functions implies a corresponding bound on the values computed by the ASSIGN operations. The result in the corollary follows from Lemma 4.7. ∎

4.2.2 Height of the Assignment Graph

Theorem 4.9 *Let c be a constant, h be any function, and α be any unbounded nondecreasing function. Also let t be greater than one. An extended $W[t]$-program can simulate the computation of a normalized $L[t]$-program if the height of the assignment graph for any execution path in any computation branch of the normalized $L[t]$-program is at most* $\log\left(c + h(k)/\alpha(n)\right)$.

PROOF. Let R_{L} be a normalized $L[t]$-program satisfying the required properties. Let $G_{b,nb}$ be the assignment graph for the computation in a nondeterministic checking branch nb along execution path b. Let $F_{b,nb}$ be the corresponding binary forest and T be a tree in $F_{b,nb}$. Recall that the root node of T represents a test, each leaf represents an $\exists_1$-register, and each of the remaining nodes represents an ASSIGN operation. By the restrictions, T involves at most $(c+h(k)/\alpha(n))$ $\exists_1$-registers. Thus, at most $f(k)n^{c+h(k)/\alpha(n)}$ combinations of values for these $\exists_1$-registers need to be examined for each tree T in each assignment graph, for some suitable function f. Altogether, analyzing a parametric polynomial number of combinations of values suffices for all trees in all assignment graphs. Based on this observation we construct an extended $W[t]$-program R_{W} to simulate the computation of R_{L} on any input $\langle x, k\rangle$.

1. **Preprocessing:**

   (a) Simulate the preprocessing phase of R_{L} directly.

   (b) Perform a partial simulation of the checking phase of R_{L} and construct the assignment graphs for all execution paths in all nondeterministic checking branches.

   (c) Enumerate all combinations of up to $(c+h(k)/\alpha(n))$ values. As explained before, the number of such combinations is bounded by a parametric polynomial. Let $\mathcal{C}$ be the resultant enumeration.

2. **Nondeterministic block 1:** Simulate the $\exists_1$-steps of R_{L}, directly.

3. **Nondeterministic block 1:** Existentially guess all combinations from $\mathcal{C}$ (possibly with repetition) satisfied by the $\exists_1$-guesses from previous step. There will be $\binom{k}{b}$ such $\exists_1$-guesses, where $b = c + h(k)/\alpha(n)$.

4. **Nondeterministic block 2 to t:** Simulate the nondeterministic steps of R_{L} directly. Let nb be the nondeterministic checking branch that has been selected by the nondeterministic operations in this step. Also let b be the unique accepting (rejecting) execution path for nb for odd (even) t.

5. (a) Construct a lookup table T_{valid} to specify how a value in an $\exists_1$-register of R_{L} affects the computation along b for nb. Let i be an index to an $\exists_1$-register of R_{L} and v_i be a value that R_{L} may store in the ith $\exists_1$-register. $T_{\mathrm{valid}}[\langle\langle nb, b, i\rangle, v_i\rangle]$ is set to 1 if one of the following holds.

   - The last nondeterministic operation in nb is universal, b ends in rejection, and the outcome of at least one test along b causes the computation to branch away from b if v_i is stored in the ith $\exists_1$-register of R_{L}.
   - The last nondeterministic operation in nb is existential, b ends in acceptance, and *no* test outcome causes the computation to branch away from b if v_i is stored in the ith $\exists_1$-register of R_{L}.

   $T_{\mathrm{valid}}[\langle\langle nb, b, i\rangle, v_i\rangle]$ is set to 0 otherwise.

   (b) Perform the following verification using T_{valid} as a lookup table.

   For even t, verify whether at least one value guessed in Step 2 causes the computation to branch away from the unique rejecting execution path.

   For odd t, verify whether all values guessed in Step 2 cause the computation to continue along the unique accepting execution path.

6. (a) Construct a lookup table $T_{\mathrm{consistent}}$ as follows. The entry $T_{\mathrm{consistent}}[\langle x, y\rangle]$ is set to 1 if and only if x is a tree in the assignment forest corresponding to the computation along b for nondeterministic checking branch nb, $y \in \mathcal{C}$, and the test at the root of x is satisfied by the combination y.

   In particular, $T_{\mathrm{consistent}}[\langle x, y\rangle]$ is set to 1 if one of the following holds.

   - The last nondeterministic operation in nb is universal, b ends in rejection, and the outcome of the test represented by the root of x causes the computation to branch away from b if the values at the leaves of x are as specified in y.

- The last nondeterministic operation in nb is existential, b ends in acceptance, and the outcome of the test represented by the root of x causes the computation to continue along b if the values at the leaves of x are as specified in y.

$T_{\text{consistent}}[\langle x, y\rangle]$ is set to 0 otherwise.

(b) Perform the following verification using $T_{\text{consistent}}$ as a lookup table.

For even t, verify that the test outcome for at least one existentially guessed combination in the $\exists_1$-block causes the computation of R_{L} to branch away from the rejecting execution path b.

For odd t, verify that, with the existentially guessed combinations in the $\exists_1$-block, all test outcomes along b are such that the computation of R_{L} continues along the accepting execution path b (and eventually accept).

Correctness: All but the ASSIGN steps of R_{L} are simulated directly by R_{W}. Since R_{L} is a normalized $L[t]$-program, all the ASSIGN operations are performed after all the nondeterministic operations. Thus the simulation of R_{L}'s computation until the last nondeterministic operation in any computation branch is performed verbatim. The computation for any nondeterministic checking branch of R_{L} accepts if and only if

COND1: t is odd and all test outcomes along the unique accepting execution path causes the computation to continue along the unique accepting execution path, or

COND2: t is even and some test outcome along the unique rejecting execution path causes the computation to branch away from the unique rejecting execution path.

Step 5 of R_{W} verifies that the values guessed for the $\exists_1$-registers of R_{L} (in Step 1) satisfy COND1 or COND2, whichever is appropriate. Step 6 verifies the same for $\exists_1$-operands. However, instead of computing the $\exists_1$-operands through ASSIGN operations (which an extended $W[t]$-program is not allowed to perform) R_{W} simply guesses the combination of values that affect the $\exists_1$-operand.

Resource Usage: The simulation of the preprocessing phase of R_{L} (Step 1(a)) takes parametric polynomial time. The partial simulation of R_{L} also takes parametric polynomial time (Property 2.12). The number of combinations in the enumeration $\mathcal{C}$ is $f(k)n^{c+h(k)/\alpha(n)}$ which is bounded by some parametric polynomial

(Theorem 4.1). Thus construction of $\mathcal{C}$ takes parametric polynomial time. The number of entries in T_{valid} is $O(kl^{t+1})$ and each entry can be computed in $O(1)$ time from the corresponding partial configuration. Let n_{AG} be the number of assignment graphs corresponding to the computation of R_{L} on $\langle x, k\rangle$. The size of the lookup table $T_{\text{consistent}}$ is bounded by $|\mathcal{C}|\ n_{AG}$. Each of these entries can be computed in $O(h(k)/\alpha(n))$ time. Thus the construction of the lookup table $T_{\text{consistent}}$ can be done in parametric polynomial time. The verifications in Steps 5(b) and 6(b) can be performed in $O(k + \binom{k}{b})$ steps. Thus the extended $W[t]$-program satisfies the Constraint $AW1$. The nondeterministic operations in Steps 2, 3, and 4 satisfy the Constraints $AW3$, T, and $TU1$. Only Steps 5(b) and 6(b) perform $\exists_1$-tests. Thus the number of $\exists_1$-tests is at most $h'(k)$ for some function h' (Constraint EW1). In addition to the values stored in the registers of R_{L}, the $W[t]$-program needs to store the indices to entries in T_{valid} and $T_{\text{consistent}}$. Thus any value stored in the registers of the extended $W[t]$-program is at most parametric polynomial (Constraint AW2). ∎

Note that Theorem 4.9 does not restrict the number of ASSIGN operations in the checking phase. The checking phase is still allowed to compute $O(h(k)\log n)$ bits from the $\exists_1$-values. Only the number of $\exists_1$-values that contribute to a single bit by means of the ASSIGN operations, is restricted to be $O(c + h(k)/\alpha(n))$. Theorems 4.7 and 4.9 give lower bounds on certain computational features of normalized $L[t]$-programs. Unless $L[t] = W[t]$, (i) the values computed by ASSIGN operations in a normalized $L[t]$-program cannot all be $O(f(k)n^{h(k)/\alpha(n)})$, and (ii) at least one test in some computation branch of a normalized $L[t]$-program is performed on some value that is computed by a sequence of ASSIGN operations involving $\Omega(h(k))$ $\exists_1$-registers. The bounds also translate to PARAMETERIZED MODEL CHECKING problem where the corresponding features are the range of functions and the number of existentially quantified variables from the first block that are involved in a relation in the matrix.

Chapter 5

Categorization Techniques for Fixed-Parameter Intractable Problems

In this chapter, we present some new membership and hardness results. The particular problems we consider include Subset Sum, Reachability Distance for Vector Addition Systems, Maximal Irredundant Set, Integer Linear Programming, Longest Common Subsequence, and Precedence Constrained Multiprocessor Scheduling. The new lower bounds for Longest Common Subsequence implies new lower bounds for other problems including Domino TreeWidth, Feasible Register Assignment, Module Allocation, and Intervalizing Colored Graphs. The definitions of all these problems can be found in the appendix.

5.1 Membership

Two approaches are commonly taken to show that a problem Q is in some class C: (a) constructing an algorithm $\mathcal{A}$ to decide Q such that $\mathcal{A}$ satisfies the associated resource-bounds, and (b) constructing an appropriate reduction from Q to some problem Q$'$ that is known to be in C. The circuit framework (due to Downey, Fellows and other co-researchers) and the model-checking framework (due to Chen, Flum, and Grohe) both suffer from the lack of a computational model. This rules out approach (a). A precondition for applying approach (b) is to have a collection of reference problems already known to be in the target class. Unfortunately, other

than the defining problems, natural complete problems beyond the fourth level of the W-hierarchy have not been identified yet. Thus, proving membership results is a nontrivial task in both these frameworks.

The ARAM characterization, given by Chen, Flum and Grohe, allow us to construct algorithms for showing membership results. However, the computation following the nondeterministic steps is extremely restrictive. One needs to be particularly careful to make sure that the algorithm actually satisfies all the required resource-bounds. The task of satisfying the restrictions often overshadows the natural flow of the actual algorithm.

The extended $W[t]$ characterization, developed in Chapter 2, is a significant advance in this context. An extended $W[t]$-program essentially is a t-alternating program that runs for parametric polynomial time. Only the number of steps involving the $\exists_1$-operands is bounded. The extended characterization allows us to deal with parametric problems from the algorithmic point of view without worrying much about the precise details of the computational model. The extended $W[t]$-programs play a key role in all the membership proofs in this chapter.

In the rest of this section we present the new membership results.

5.1.1 Subset Sum

The parametric problem Subset Sum is defined as follows.

Subset Sum

Input: A set of integers $X = \{x_1, \ldots, x_m\}$, integers s and k.
Parameter: k.
Question: Is there a set $X' \subseteq X$ such that $|X'| = k$ and sum of all the integers in X' is exactly s?

The problem has been shown to be $W[1]$-hard by Downey and Fellows [26]. Fellows and Koblitz have shown that the problem is in $W[P]$ [35]. We show that Subset Sum is in $W[3]$.

Theorem 5.1 Subset Sum *is in* $W[3]$.

Proof. We construct an extended $W[3]$-program R_{SUM} to decide Subset Sum. We assume that all numbers in X are positive. In case X contains negative numbers, R_{SUM} adds the smallest negative number x_{smallest} to all the numbers in X

and adds $k \times x_{\text{smallest}}$ to the target sum s, in the preprocessing phase. Let $b = 2^{\lceil \log k \rceil}$. Also let $l_{\max}$ be the maximum number of bits in the binary representation of x_i, over all i, $1 \leq i \leq m$, or the target sum s (after the preprocessing). Conceptually, R_{SUM} works in two major steps. In the first step, R_{SUM} considers the k existentially selected integers as base-b numbers. R_{SUM} adds them (without considering the carries) to produce two integers s_{sum} and s_{carry} - the first containing the sum digits (in base-b) and the second containing the carry digits (in base-b). In the second step, R_{SUM} performs a binary addition of s_{sum} and s_{carry} to produce s_{result} and checks that s_{result} and s are same. The details are given as Algorithm 5.1. For notational convenience, we use $x^b[p]$ ($x[p]$) to denote the base-b digit (bit) at position p in number x.

Algorithm 5.1: An extended $W[3]$-program R_{SUM} to decide SUBSET SUM

Input: A set of integers $X = \{x_1, \ldots, x_m\}$, a target sum s, an integer k.

Preprocessing:

Make sure that all numbers are positive by adding appropriate offsets (if necessary) to all input numbers and the target sum.

Construct a lookup table T from the input numbers so that given the index of a number and a bit position, the corresponding bit can be determined (by a JZERO test) in constant time.

$$T[\langle i, j \rangle] = \begin{cases} 1, & \text{if the } j\text{th bit of the } i\text{th number is 1} \\ 0, & \text{otherwise} \end{cases}$$

where $1 \leq i \leq m$ and $0 \leq j \leq l_{\max} - 1$.

Nondeterministic block 1: Existentially select indices $i_1, \ldots, i_k$ to k integers from X. Let the indices be stored in registers $g_1, \ldots, g_k$, respectively.

Nondeterministic block 2: Universally guess a bit position p of the target sum s and store the bit position in register g_{k+1}.

Let bit position p correspond to digit position p_b in the base-b interpretation. Extract all base-b digits from the k selected numbers at position p_b and p_{b-1} and compute the base-b digits of s_{sum} and s_{carry} at position p_b assuming a zero carry-in (in base-b) at position p_{b-1}.

`/* Compute the required carry-in from` p_{b-1}`th position. This carry must be 1 or 0 */`

$c_b \leftarrow s^b[p_b] - (s^b_{\text{sum}}[p_b] + s^b_{\text{carry}}[p_b]) \bmod b$ `/* Computation in base-`b `*/`

if c_b *is not* 0 *or* 1 **then**

Reject in this branch

end

/* R_{SUM} continued */

11 Let c_2 be the (binary) carry-in required at bit position p.
$c_2 \leftarrow |s[p] - (s_{\text{sum}}[p] - s_{\text{carry}}[p]) \bmod 2|$ /* Computation in binary */

12 **if** c_2 *is* 1 **then**

13 **Nondeterministic block 2: Universally** guess a bit position p', $0 < p' < p$.

14 Compute the base-b digits of s_{sum} and s_{carry} for the base-b digit position corresponding to bit position p' (as in Step 6). Compute the bits $s_{\text{sum}}[p']$ and $s_{\text{carry}}[p']$ from the base-b digits.

15 **if** *(binary) carry is generated or propagated at position* p' **then**

16 Accept in this branch.

17 **else**

18 **Nondeterministic block 3: Existentially** guess a bit position $p'', p' < p'' < p$.

19 Compute the bits $s_{\text{sum}}[p'']$ and $s_{\text{carry}}[p'']$ as in Step 14.

20 **if** *a carry is generated at bit position* p'' **then** Accept in this branch. **else** Reject in this branch.

21 **end**

22 **else**

23 **Nondeterministic block 2: Universally** guess a position p', $0 < p' < p$.

24 Compute the bits $s_{\text{sum}}[p']$ and $s_{\text{carry}}[p']$ as in Step 14.

25 **if** *no carry is generated at position* p' **then**

26 Accept in this branch.

27 **else**

28 **Nondeterministic block 3:**

29 **Existentially** guess a position $p'', p' < p'' < p$

30 Compute the bits $s_{\text{sum}}[p'']$ and $s_{\text{carry}}[p'']$ as in Step 14.

31 **if** *no carry is propagated through position* p'' **then**

32 Accept in this branch.

33 **else**

34 Reject in this branch.

35 **end**

36 **end**

37 **end**

Correctness: The correctness of the arithmetic involved is easy to verify. We therefore argue that the algorithm R_{SUM} correctly implements the arithmetic. Step 5 of R_{SUM} universally selects a bit position p of the target sum. The computation in each resulting universal branch verifies that the pth bits of the target sum and the sum of the existentially selected numbers agree. By the universal choice of Step 5, R_{SUM} accepts if and only if the verification succeeds for all bit positions. During the verification R_{SUM} computes the pth bit of the partial sum (Steps 6-7) and the required carry-in (Steps 8-11) based on (deterministic) arithmetic as explained before. In the final step, R_{SUM} ensures that the required carry-in is obtained at bit position p using nondeterminism. For a carry-in of 1, R_{SUM} verifies (Step 13) for each bit position $p' < p$ that either a carry is generated or propagated at position p' (Step 14-16) or a carry is generated at some bit position p'', $p' < p'' < p$ (Step 18-21). The verification for a required carry-in of 0 is symmetric.

Resource Usage: In any given computation branch either Steps 12-21 or Steps 22-36 are executed. Thus, the nondeterministic operations (Steps 4, 5, 13, 18; or Steps 4, 5, 23, 28) satisfy the constraints AW3, T1, and TU1 for $t = 3$. Step 2 in the preprocessing phase can be done in polynomial time. The lookup table T has $ml_{\max}$ entries, each entry representing one bit's worth of information. Thus the time needed to construct T is a polynomial in n. Most of the remaining steps are trivial and require constant time. Step 6 can be implemented using $2k \log b$ $\exists_1$-tests using the table T for lookup. The same number of $\exists_1$-tests are needed in each of Steps 14, 19, 24, and 30. No other step uses $\exists_1$-tests. Thus R_{SUM} satisfies Constraint $EW1$. R_{SUM} needs to store the indices to the numbers, indices to the bit positions in a number, sums of at most k bits, and indices to the entries in T during the construction of the lookup table. Each of theses values is at most a polynomial in n. Thus the program satisfies the constraint AW2. ∎

5.1.2 Reachability Distance for Vector Addition Systems (Petri Nets)

C. A. Petri introduced Petri nets to represent concurrent processes in a formal way. Karp and Miller introduced the vector addition system to analyze a particular model of parallel computation. It turned out that the vector addition system was mathematically equivalent to Petri nets. Decidability of various properties of Petri nets has been of interest since the early development of Petri nets [34]. In this subsection, we analyze the complexity of a parameterized version of the REACHABILITY problem (defined below). Although, a precise degree of intractabil-

ity for the problem is not known in the classical context, Lipton showed that the problem is *EXPSPACE*-hard [16].

Reachability Distance for Vector Addition Systems (Petri Nets)

Input: A set $\{\vec{x}_1, \ldots, \vec{x}_m\}$ of m vectors, each consisting of l integers, a non-negative starting vector $\vec{s} = (s_1, \ldots, s_l)$, a non-negative target vector $\vec{t} = (t_1, \ldots, t_l)$, a positive integer k.
Parameter: k.
Question: Is there a set of k indices $i_1, \ldots, i_k$ such that $\vec{t} = \vec{s} + \sum_{j=1}^{k} \vec{x}_{i_j}$ and each of the l integer components in each of the k intermediate sums is non-negative?

The problem is known to be $W[1]$-hard [28, 29]. However, no membership result is known. We show that the problem is in $W[5]$. Our result builds upon the $W[3]$-membership of Subset Sum, presented in the preceding subsection.

Theorem 5.2 Reachability Distance for Vector Addition Systems *is in* $W[5]$.

Proof. We construct an extended $W[5]$-program R_{RDVAS} (Algorithm 5.2) to establish the upper bound. R_{RDVAS} starts by existentially selecting the indices of the k vectors to add. It then universally selects an intermediate step and a component position in the sum vector. The sums of positive and negative components at the selected position are computed separately and compared to ensure that the sum of negative components does not exceed the sum of the positive components. The computation of the intermediate sums uses the algorithm for Subset Sum described before. As was done in the proof of Theorem 5.1, we use $s^b[p]$ ($s[p]$) to denote the base-b digit (bit) at position p of the integer s. The details are given in Algorithm 5.2.

Correctness: By the universal selection in Steps 3 and 4, R_{RDVAS} accepts if and only if the verification succeeds for each component position at each intermediate step. By the existential selection of a bit position at Step 5, the verification accepts if the sum of positive numbers exceeds the sum of negative numbers starting at some (existentially selected) bit position. The rest of the verification involves computation of the sums as in Algorithm 5.1. The correctness of the computation of sum bits (Steps 8 and 9) follows from arguments similar to those for Algorithm 5.1.

Algorithm 5.2: An extended $W[5]$-program R_{RDVAS} to decide Reachability Distance for Vector Addition Systems

Input: m vectors $\vec{x}_1, \ldots, \vec{x}_m$ each consisting of l integers, a non-negative starting vector $\vec{s} = (s_1, \ldots, s_l)$, a non-negative target vector $\vec{t} = (t_1, \ldots, t_l)$, a positive integer k.

1 **Preprocessing:** Let $l_{\max}$ be the maximum number of bits in the binary representation of any number in the input. Construct lookup table T such that given the index of a number and a bit position, the value of the corresponding bit of the number can be retrieved in constant time.

$$T[\langle i, j \rangle] = \begin{cases} 1, & \text{the } j\text{th bit of the } i\text{th number in input is 1} \\ 0, & \text{otherwise} \end{cases}$$

where $0 \leq j \leq l_{\max} - 1$ and $1 \leq i \leq l(m+2)$.

2 **Nondeterministic block 1: Existentially** guess the indices $i_1, \ldots, i_k$ of the k vectors for the solution.

3 **Nondeterministic block 2: Universally** guess the jth intermediate step, $1 \leq j \leq k$.

4 **Nondeterministic block 2: Universally** guess the component position y of the sum to be verified, $1 \leq y \leq n$. Let $V_{j,y}$ be the set of the yth components of the first j vectors among those selected in Step 1, i.e. $V_{j,y} = \left\{ yth \text{ component of } \vec{x}_{i_{j'}} \mid 1 \leq j' \leq j \right\}$.

5 **Nondeterministic block 3: Existentially** guess the bit position p_{diff}, $0 \leq p_{\text{diff}} \leq l$ where the sum of positive components first differ (starting from the most significant bit) from the sum of negative components.

`/* Start of Subset Sum */`

6 **Nondeterministic block 4: Universally** guess a bit position p of the component sum.

7 **if** $p < p_{\text{diff}}$ **then** *Accept* in this branch.

`/* Compute the sums for positive and negative numbers in` $V_{j,y}$ `separately */`

8 Let bit position p correspond to digit position p_b in the base-b interpretation. Extract all base-b digits at position p_b and p_{b-1} from the positive numbers in V_j. Compute the base-b digits of s^+_{sum} and s^+_{carry} at position p_b.

9 Similarly, compute the base-b digits at position p_b of s^-_{sum} and s^-_{carry} for the negative numbers in V_j.

/* R_{RDVAS} continued */

(**Constant-size Existential Block**) Existentially guess the carries c^+ and c^- (00, 01, 10, and 11).

/* If the combination of c^+ and c^- is inconsistent with $s^+_{sum}[p]$ and $s^-_{sum}[p]$ then terminate this branch. */

if $p > p_{\text{diff}}$ **then**
    if $(s^+_{\text{result}}[p] + c^+) \bmod 2 \neq (s^-_{\text{result}}[p] + c^-) \bmod 2$ **then**
        Reject in this branch
else
    /* $p = p_{\text{diff}}$ by Step 7 */
    if $(s^+_{\text{result}}[p] + c^+) \bmod 2 \neq 1$ *or* $(s^-_{\text{result}}[p] + c^-) \bmod 2 \neq 0$ **then**
        Reject in this branch
end
Nondeterministic block 4: Universally select a carry-in c from $\{c^+, c^-\}$.
if c *is* 1 **then**
    Nondeterministic block 4: Universally guess a bit position p', $0 < p' < p$.
    if *carry is generated or propagated at position* p' **then**
        Accept in this branch
    else
        Nondeterministic block 5: Existentially guess a bit position $p'', p' < p'' < p$
        if *a carry is generated at bit position* b'' **then** Accept in this branch **else** Reject in this branch.
    end
else
    Nondeterministic block 4:
        Universally guess a position $p' < p$
    if *no carry is generated at position* p' **then**
        Accept in this branch
    else
        Nondeterministic block 5:
            Existentially guess a position $i, p' < i < p$
        if *no carry is propagated through position* i **then** Accept in this branch
        else Reject in this branch
    end
end

By the existential selection of the carries in Step 10 and subsequent verification in Steps 11-17, the verification accepts if and only if a consistent pair of carries is generated for the selected bit position. The correctness for the verification of desired carry-ins (Steps 19-38) follows from the same of Algorithm 5.1.

Resource Usage: Starting from Step 6 Algorithm 5.2 essentially implements the checking phase of Algorithm 5.1. the Steps 10 to 17 are new and they require $O(1)$ time. Step 1 requires polynomial time while steps 2 to 5 can be performed in $O(h(k))$ steps for some function h. Also, any value used in the computation in Steps 1 to 5, and in Steps 10 to 17 is at most parametric polynomial. Thus Constraints AW1, AW2, EW1 are satisfied. The nondeterministic steps satisfy Constraints AW3 and TU1. The number of alternations is 5 except for the presence of an existential guess from a constant range (Step 10). By applying distributive law (implemented through a different interpretation of values in the associated guess registers) the existential step in Step 10 can be pushed down to nondeterministic block 5. Thus Constraint T1 is also satisfied for $t = 5$.

∎

5.1.3 Maximal Irredundant Set

Let $G = (V, E)$ be a graph, V' a subset of the vertices, and $u \in V'$. A *private neighbour* of u with respect to V' is a vertex u' that is adjacent to u and to no other vertex in V'. The parameterized version of the Maximal Irredundant Set is defined as follows.

Maximal Irredundant Set

Input: A graph $G = (V, E)$, a positive integer k.
Parameter: k.
Question: Is there a set $V' \subseteq V$ such that (1) each vertex $u \in V'$ has a private neighbour, and (2) V' is not a proper subset of any other set $V'' \subseteq V$ that also has this property?

Maximal Irredundant Set is known to be $W[2]$-hard [8] and in $W[P]$ [17]. In this section, we show that Maximal Irredundant Set is in $W[4]$ by constructing an extended $W[4]$-program to decide the problem. The algorithm is essentially the same as Cesati's algorithm [17]. We only show that the algorithm can be implemented by an extended $W[4]$-program.

Theorem 5.3 MAXIMAL IRREDUNDANT SET *is in* $W[4]$.

PROOF. We construct an extended $W[4]$-program R_{MIS} to decide MAXIMAL IRREDUNDANT SET. R_{MIS} existentially guesses an irredundant set V', and the k private neighbours of the vertices in V'. R_{MIS} then checks that V' indeed form an irredundant set in the input graph $G = (V, E)$. To ensure maximality, R_{MIS} universally guesses another vertex w, and verifies that $V' \cup \{w\}$ is not an irredundant set. The details are given in Algorithm 5.3. For notational convenience, we use $N[v]$ to denote the *closed neighbourhood* of a vertex v where the closed neighbourhood of v includes v and all vertices that are adjacent to v.

Correctness: In addition to guessing k vertices for the solution V' (Line 2) R_{MIS} existentially guesses a private neighbour p_i for the ith vertex in V' (Line 3), for each i, $1 \leq i \leq k$. In Lines 4 to 9, R_{MIS} verifies that v_{p_i} is indeed a private neighbour for the ith vertex in V'. This checking ensures that V' forms an irredundant set. R_{MIS} ensures the maximality of V' by verifying that $V' \cup \{v\}$ is not an irredundant set for any choice of $v \notin V'$. By the universal selection of v_u in Line 10 and the checking in Lines 11 to 13, R_{MIS} accepts if and only if the remaining computation (Lines 14 to 22) accepts for all choice of $v_u \notin V'$. In this part of the checking, R_{MIS} existentially selects a vertex v_{i_j} in V' (Line 14) and verifies that any vertex in $N[v_{i_j}]$ is also in $N[v_{i_{j'}}]$ for some $j' \neq j$, $1 \leq j' \leq k+1$. In other words Lines 16 to 21 ensure that v_{i_j} does not have any private neighbour with respect to $V' \cup \{v_u\}$. Thus R_{MIS} accepts if and only if (the existentially guessed) V' is an irredundant set and $V' \cup \{v\}$ is not an irredundant set for any $v \notin V'$.

Resource Usage: Construction of the adjacency matrix in Step 1 takes polynomial time. The nondeterministic operations in Lines 2, 3, 10, 14, and 15 take $O(k)$ time. Each checking for inclusion in a closed neighbourhood can be done using a JZERO test and the lookup table constructed in the preprocessing phase. Thus the loops from Line 4 to 9 take $O(k^2)$ time. By similar arguments, the operations from Line 11 to 13 and from Line 16 to 21 take $O(k)$ time. Thus the program satisfies Constraint AW1. The program needs to store the indices to the vertices and indices to the lookup tables. All these indices are bounded by n^2. Thus Constraint AW2 is also satisfied. The nondeterministic operations in Lines 2, 3, 10, 14, and 15 satisfy the Constraints AW3, T1, and TU1 for $t = 4$. JZERO and JEQUAL tests on the $\exists_1$-registers are performed in Lines 5, 7, 12, 16, and 19. The number of such tests is $(k^2 + 2k)$. No other operation uses any $\exists_1$-operand. Thus Constraint EW1 is also satisfied. ∎

Corollary 5.4 MAXIMAL IRREDUNDANT SET *is in* $W^*[3]$.

Algorithm 5.3: A $W[4]$-program (also a $W^*[3]$-program) R_{MIS} to decide Maximal Irredundant Set

Input: A graph $G = (V, E)$, an integer k

Preprocessing: Construct a lookup table from the input graph G so that $r_{\langle i,j \rangle}$ is 1 if and only if $v_j \in N[v_i]$.

Nondeterministic block 1: Existentially guess the indices $i_1, \ldots, i_k$ to the vertices that form the irredundant set V'.

Nondeterministic block 1: Existentially guess the indices $p_1, \ldots, p_k$ to the private neighbours for the vertices $v_1, \ldots, v_k$, respectively.

`/* Verify that` V' `is an irredundant set */`

for $x = 1$ **to** k **do**
    if $v_{p_x} \notin N[v_{i_x}]$ **then** Reject
    for $y = 1$ **to** k *and* $y \neq x$ **do**
        if $v_{p_x} \in N[v_{i_y}]$ **then** Reject
    end
end

Nondeterministic block 2: Universally guess the index i_{k+1} to a vertex to extend the irredundant set V'.

`/* Verify that` $v_{i_{k+1}} \notin V'$ `*/`

for $x = 1$ **to** k **do**
    if $v_{i_x} = v_{i_{k+1}}$ **then** Accept
end

`/* Guess a vertex in` $V' \cup \{v_{i_{k+1}}\}$ `that witnesses that` $V' \cup \{v_{i_{k+1}}\}$ `is not an irredundant set */`

Nondeterministic block 3: $((k+1)$**-way branching**$)$ **Existentially** guess an index j from $\{1, \ldots, k+1\}$.

`/* Verify that` v_{i_j} `does not have a private neighbour with respect to` $V' \cup \{v_{i_{k+1}}\}$ `*/`

Nondeterministic block 4: Universally guess the index u to a vertex in V.

if $v_u \notin N[v_{i_j}]$ **then** Accept
else
    for $x = 1$ **to** k *and* $x \neq j$ **do**
        if $v_u \in N[v_{i_x}]$ **then** Accept
    end
end
Reject

PROOF. This follows from the fact that the range of values for the third existential quantifier is $[1, k+1]$. ∎

5.1.4 Weighted Integer Programming

In this section, we show new membership results for different versions of WEIGHTED INTEGER PROGRAMMING problem. Downey and Fellows analyzed the fixed-parameter complexity of a binary version of the problem whose definition is given below.

Weighted Binary Integer Programming

Input: A set of variables $X = \{x_1, \ldots, x_m\}$, a set of linear constraints $C = \{c_1, \ldots, c_l\}$ on the variables in X, an integer k, such that the coefficients in the constraints are binary.
Parameter: k.
Question: Is there a binary solution of weight-k to the system of linear equations? Here the weight of a solution is the number of variables set to 1.

Downey and Fellows showed that the problem is $W[2]$-complete [25, 28]. We consider extended versions of the problem where the matrix A and the vector b may not be binary. Let $c \geq 1$ be a constant, f and h be any fixed functions, and α be any unbounded nondecreasing function. We define three versions of the problem and show that they are contained in $W[2]$, $L[2]$, and $W[5]$, respectively. The versions differ in the bound on the coefficients in the constraints.

Weighted Binary Linear Integer Programming - I (BLIP-I)

Input: A set of variables $X = \{x_1, \ldots, x_m\}$, a set of linear constraints $C = \{c_1, \ldots, c_l\}$ on the variables in X, an integer k, such that the coefficients in the constraints are at most $f(k)n^{c/k+h(k)/\alpha(n)}$.
Parameter: k.
Question: Is there a binary solution of weight-k to the system of linear equations?

Weighted Binary Linear Integer Programming - II (BLIP-II)

Input: A set of variables $X = \{x_1, \ldots, x_m\}$, a set of linear constraints $C = \{c_1, \ldots, c_l\}$ on the variables in X, an integer k, such that the coefficients in the constraints are at most $f(k)n^{h(k)}$.
Parameter: k.
Question: Is there a binary solution of weight-k to the system of linear equations?

Weighted Binary Linear Integer Programming - III (BLIP-III)

Input: A set of variables $X = \{x_1, \ldots, x_m\}$, a set of linear constraints $C = \{c_1, \ldots, c_l\}$ on the variables in X.
Parameter: k.
Question: Is there a binary solution of weight-k to the system of linear equations?

The $W[2]$-hardness of Weighted Binary Integer Programming implies that BLIP-I, BLIP-II, and BLIP-III are all $W[2]$-hard. We prove the upper bounds in the rest of this section.

Theorem 5.5 BLIP-II *is in* $L[2]$.

Proof. In order to decide BLIP-II, we construct a basic-$L[2]$-program $R_{\text{BLIP-II}}$ that performs ASSIGN operations only, on the $\exists_1$-registers. Construction of an unrestricted basic $L[2]$-program would have been easier. However, we choose the restricted version in order to obtain the upper bound for BLIP-I by a direct application of Theorem 4.7.

$R_{\text{BLIP-II}}$ existentially guesses the k variables $x_{i_1}, \ldots, x_{i_k}$ to be set to 1 and then universally guesses a constraint c_u to verify. As we are interested in binary solutions, only the coefficients of $x_{i_1}, \ldots, x_{i_k}$ and the constant term of c_u determine whether c_u has been satisfied. Let S_u be the set of these $k+1$ coefficients of interest from c_u. As part of the verification, $R_{\text{BLIP-II}}$ computes two separate sums. The first is the sum of the positive numbers in S_u while the second is the sum of the negative numbers in S_u. By definition of BLIP-II, any coefficient in a constraint is at most $f(k)n^{h(k)}$. Each such coefficient can be represented by $(\log f(k) + h(k) \log n)$ bits. Thus, a coefficient can be stored in $h(k)$ registers, with $b = (\log f(k)/h(k) + \log n)$ bits in each register. Any operation (addition or comparison) on the coefficients, therefore, involves $O(h(k))$ registers. Once the sums are computed $R_{\text{BLIP-II}}$ checks

Algorithm 5.4: A normalized $L[2]$-program $R_{\text{BLIP-II}}$ to decide BLIP-II

Input: Variables $x_1, \ldots, x_m$, linear constraints $c_1, \ldots, c_l$, an integer k

Preprocessing: Arrange the coefficients into a two-dimensional lookup table T_{coeff} such that $T_{\text{coeff}}[\langle x, i\rangle, y]$ stores bits $((i-1)b+1)$ to $i \times b$ of the (signed) coefficient associated with the variable x in constraint y.

Nondeterministic block 1: Existentially guess the k variables $\{x_{i_1}, \ldots, x_{i_k}\}$ to be set to 1. Let registers $g_1, \ldots, g_k$ store these values, respectively.

Nondeterministic block 2: Universally select an index u to the constraint c_u and store the index in g_{k+1}.

`/* Initialize the registers that will store the sum of` ***positive*** `coefficients */`

Initialize $g_{k+2}, \ldots, g_{k+1+h}$ to be 0

`/* Initialize the registers that will store the sum of` ***negative*** `coefficients */`

Initialize $g_{k+2+h}, \ldots, g_{k+1+2h}$ to be 0

if *the constant in* c_u *is positive* **then**

  Set $g_{k+2} \ldots g_{k+1+h}$ to represent the constant in c_u

else

  Set $g_{k+2+h} \ldots g_{k+1+2h}$ to represent the constant in c_u

end

`/* Compute the sums */`

for $s = 1$ **to** k **do** `/* for the` s`th` $\exists_1$`-guess */`

  $carry \leftarrow 0$

  if *coefficient of* x_{i_s} *in* c_u *is positive* **then** offset $\leftarrow 0$

  else offset $\leftarrow h$

  for $j = 1$ **to** h **do**

    `/* for each of the` $h(k)$ `parts of the coefficient */`

    Update $g_{k+j+\text{offset}}$ and $carry$ from the sum $g_{k+j+\text{offset}} + T_{\text{coeff}}[\langle x_{i_s}, j\rangle, c_u] + carry$

  end

end

`/* Verify that` c_u `is satisfied */`

if *the computed sums are consistent with* c_u **then** Accept

else Reject

whether the relation between the sums is as required by c_u. The details are given in Algorithm 5.4.

Correctness: By the universal selection of a constraint in Line 3, $R_{\text{BLIP-II}}$ accepts if and only if the remaining part of checking phase accepts for all constraints. During the verification, $R_{\text{BLIP-II}}$ evaluates the constraint c_u on the existentially guessed solution to ensure that c_u has been satisfied. Thus $R_{\text{BLIP-II}}$ accepts if and only if all constraints are satisfied by the existentially guessed solution of weight-k (Line 2).

Resource Usage: The size of the lookup table T_{coeff} is $h(k)ml$ and each entry can be initialized in $O(n)$ time ($O(1)$ time in case the input is given in a suitable format). Thus Step 1 can be done in $O(h(k)mln)$ time. The initializations in Lines 4 to 10, and the comparisons in Line 19 require $O(h(k))$ time as they are performed on operands that spans $h(k)$ registers. The operation in Line 16 takes $O(1)$ time. Thus the loops from Line 11 to Line 18 takes $O(kh(k))$ time. The remaining operations can be done in $O(k)$ time. Thus $R_{\text{BLIP-II}}$ satisfies Constraint AW1. The registers need to store the indices to the variables, indices to the constraints, indices to the lookup table entries, and at most $(\log f(k)/h(k) + \log n)$ bits of a coefficient. Each of these values is bounded by above by $\max(h(k)mln, (f(k))^{1/h(k)}n)$ (Constraint AW2). The nondeterministic operations in Lines 3 and 4 satisfy Constraints AW3, T1, TU1, and BL1 for $t = 2$. ∎

Corollary 5.6 BLIP-I *is in* $W[2]$.

PROOF. Given an input $\langle x, k\rangle$, the restricted $L[2]$-program $R_{\text{BLIP-II}}$ from previous theorem can be used to decide whether $\langle x, k\rangle$ is in BLIP-I. However, each coefficient can now be stored in a single register. Also, each value computed by an ASSIGN operation is at most $f(k)n^{c/k+h(k)/\alpha(n)}$. By Theorem 4.7, the computation of $R_{\text{BLIP-II}}$ on $\langle x, k\rangle$ can be simulated by an extended $W[2]$-program. ∎

Theorem 5.7 *The exact version of* BLIP-III*, where all constraints are* equality *constraints, is in* $W[3]$.

PROOF. An extended $W[3]$-program $R_{\text{E-BLIP-III}}$ can be constructed from the extended $W[3]$-program R_{SUM} (Theorem 5.1) to decide the exact version of BLIP-III. We give an outline of the algorithm.

1. **Nondeterministic block 1: Existentially** select k variables to be set to 1.
2. **Nondeterministic block 2: Universally** select a constraint c_i.
3. (Start of SUBSET SUM)
   (a) Simulate R_{SUM} to compute the sum S^+ of the positive coefficients in c_i whose indices have been selected in the $\exists_1$-block. Include all positive constants of c_i in the sum.
   (b) Simulate R_{SUM} to compute the sum S^- of the negative coefficients in c_i whose indices have been selected in the $\exists_1$-block. Include all negative constants of c_i in the sum.
   (c) Verify that the computed sums S^+ and S^- are equal.

∎

It is not obvious whether the general version of BLIP-III can be decided by an extended $W[\mathcal{3}]$-program. Checking for bitwise-equality (Step 3(c)) is not sufficient for the general version. However, an extended $W[5]$-program can be constructed using algorithmic techniques from the proof of Theorem 5.2 to decide BLIP-III.

Theorem 5.8 BLIP-III *is in* $W[5]$.

PROOF. The extended $W[5]$-program $R_{\text{BLIP-III}}$ is similar to R_{RDVAS}. The program existentially guesses the k variables $x_{i_1}, \ldots, x_{i_k}$ to be set to 1, universally guesses a constraint c_j, and verifies that c_j is satisfied by the existentially guessed binary solution.

$R_{\text{BLIP-III}}$ performs the verification by retrieving the coefficients of $x_{i_1}, \ldots, x_{i_k}$ in c_j. Let V_j be the set of the retrieved coefficients and any constant in c_j. $R_{\text{BLIP-III}}$ computes the sums s^+ and s^- of positive numbers and negative numbers in V_j, respectively. Finally, $R_{\text{BLIP-III}}$ checks whether s^+ and s^- are consistent with c_j. As was done in Algorithm 5.1, $R_{\text{BLIP-III}}$ interprets the numbers as base-b numbers, for $b = \lceil \log(k+1) \rceil$ and computes the sum digits and carry digits separately. The details are given in Algorithm 5.5.

Algorithm 5.5: A $W[5]$-program $R_{\text{BLIP-III}}$ to decide BLIP-III

Input: Variables $x_1, \ldots, x_m$, linear constraints $c_1, \ldots, c_l$, an integer k

Let $l_{\max}$ be the maximum number of bits in the binary representation of any number in the input. Construct lookup table T such that given the index of variable, an index of a constraint, and a bit position, the value of the corresponding bit can be retrieved in constant time.

$$T[\langle i, \langle p, j \rangle \rangle] = \begin{cases} 1, & \text{the } p\text{th bit of } x_i \text{ in } c_j \text{ is 1} \\ 0, & \text{otherwise} \end{cases}$$

where $1 \leq j \leq l$, $0 \leq p \leq p_{\max}$, and $1 \leq i \leq m$.

Nondeterministic block 1: Existentially guess the indices $i_1, \ldots, i_k$ of the k variables which are to be set to 1.

Nondeterministic block 2: Universally select a constraint c_j, $1 \leq j \leq l$.

if c_j *is an inequality* **then**

    Nondeterministic block 3: Existentially guess the bit position p_{diff}, $0 \leq p_{\text{diff}} \leq l$ where the sum of positive components first differ (starting from the most significant bit) from the sum of negative components.

else

    Set p_{diff} to be -1

end

`/* Start of Subset Sum */`

Nondeterministic block 4: Universally guess a bit position p of the component sum.

if $p < p_{\text{diff}}$ **then** *Accept* in this branch.

`/* Compute the sums for relevant positive and negative coefficients in` c_j `separately */`

Let bit position p correspond to digit position p_b in the base-b interpretation. Extract all base-b digits at position p_b and p_{b-1} from the positive coefficients in c_j. Compute the base-b digits of s^+_{sum} and s^+_{carry} at position p_b.

Similarly, compute the base-b digits at position p_b of s^-_{sum} and s^-_{carry} for the negative numbers in c_j.

/* $R_{\text{BLIP-III}}$ continued */

13 (**Constant-size Existential Block**) Existentially guess the carries c^+ and c^- (00, 01, 10, and 11).

/* If the combination of c^+ and c^- is inconsistent with $s^+_{sum}[p]$ and $s^-_{sum}[p]$ then terminate this branch. */

14 **if** $p > p_{\text{diff}}$ **then**

15 **if** $(s^+_{\text{result}}[p] + c^+) \bmod 2 \neq (s^-_{\text{result}}[p] + c^-) \bmod 2$ **then**

16 Reject in this branch

17 **else**

/* $p = p_{\text{diff}}$ by Step 7 */

18 Set op be $<$ or $>$ as appropriate for c_j.

19 **if** $(s^+_{\text{result}}[p] + c^+) \bmod 2$ op $(s^-_{\text{result}}[p] + c^-) \bmod 2$ **then**

20 Reject in this branch

21 **end**

22 **Nondeterministic block 4: Universally** select a carry-in c from $\{c^+, c^-\}$.

23 **if** c *is* 1 **then**

24 **Nondeterministic block 4: Universally** guess a bit position p', $0 < p' < p$.

25 **if** *carry is generated or propagated at position* p' **then**

26 Accept in this branch

27 **else**

28 **Nondeterministic block 5: Existentially** guess a bit position $p'', p' < p'' < p$

29 **if** *a carry is generated at bit position* b'' **then** Accept in this branch **else** Reject in this branch.

30 **end**

31 **else**

32 **Nondeterministic block 4:**

33 **Universally** guess a position $p' < p$

34 **if** *no carry is generated at position* p' **then**

35 Accept in this branch

36 **else**

37 **Nondeterministic block 5:**

38 **Existentially** guess a position $i, p' < i < p$

39 **if** *no carry is propagated through position* i **then** Accept in this branch

40 **else** Reject in this branch

41 **end**

42 **end**

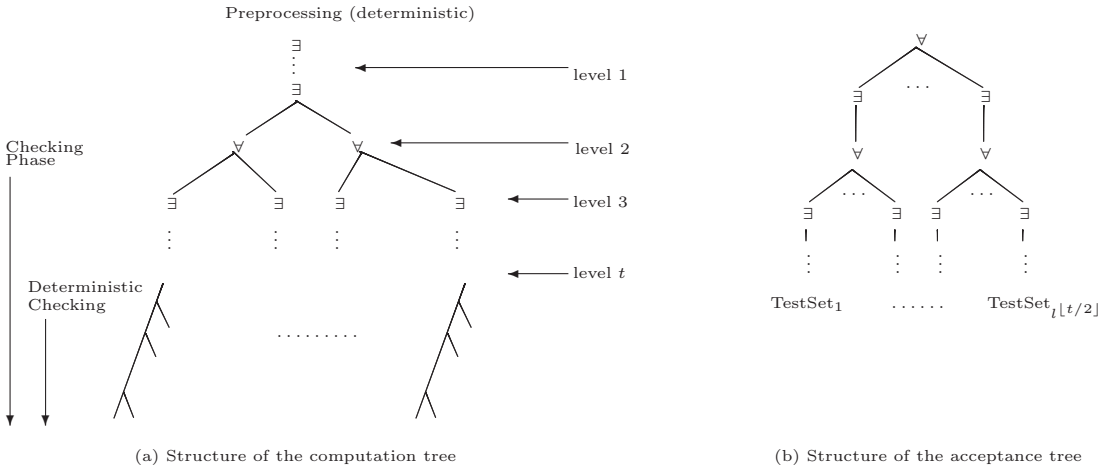
Correctness: The argument for correctness is similar to that for Algorithm 5.2.

Resource Usage: The satisfaction of the resource constraints follows from arguments similar to those for Algorithm 5.2. ∎

5.2 Hardness

The usual approach to show that some problem Q is hard for some class C is either to reduce some C-hard problem to Q or to construct an appropriate generic reduction to Q. The classes $W[1]$ and $W[2]$ are now reasonably populated with natural complete problems. Thus showing hardness results for them may not be difficult. Recently Chen and Zhang showed that certain PRODUCT COVERING problems in 3-tier supply chain model are complete for $W[3]$ and $W[4]$, respectively [23]. However, the defining problems (restricted versions of WCS and MODEL CHECKING) and a few variants of them are the only problems known to be complete for classes above $W[4]$ in the W-hierarchy. Thus showing hardness results beyond $W[4]$ involves the construction of generic reductions. The WCS problem on t-normalized circuits has been the starting point of these generic reductions. Chen, Flum, and Grohe have provided the MODEL CHECKING problem as an alternative to WCS on t-normalized circuits. The normalized computational characterizations, proposed in Chapter 2, serve as yet another alternative for the $W[t]$ classes. The added advantage of working with the later two characterizations is that hardness results can be generalized to higher levels in the W-hierarchy or to the L-hierarchy with little extra effort.

In the rest of this section, we show new hardness results for LONGEST COMMON SUBSEQUENCE and PRECEDENCE CONSTRAINED MULTIPROCESSOR SCHEDULING. Before presenting the hardness results, we define some computational features of the normalized programs that will be used in the hardness proofs. In Section 2.3 we defined the partial computation tree of a normalized $W[t]$-program and a normalized $L[t]$-program. Consider the computation of a normalized $W[t]$-program R_{W} on $\langle x, k\rangle$. Let $\mathcal{C}_W$ be the partial computation tree of R_{W} for input $\langle x, k\rangle$. Figure 5.1 presents the structure of $\mathcal{C}_W$. Starting from the first universal step, the d-th branching node in $\mathcal{C}_W$ corresponds to the d-th nondeterministic step in the computation of R_{W}, $1 \leq d \leq t-1$. Recall that a node in the partial computation tree is *universal* (*existential*) if it corresponds to a universal (existential) step in



(a) Structure of the computation tree

(b) Structure of the acceptance tree

Figure 5.1: The structure of the partial computation tree and an acceptance tree of a normalized $W[t]$-program.

the computation of R_{W}. An edge in the partial computation tree is a universal (an existential) edge if its parent is a universal (an existential) node. An accepting computation of R_{W} can be represented by a sub-tree $\mathcal{C}_{W,\mathrm{accept}}$ of $\mathcal{C}_W$ where $\mathcal{C}_{W,\mathrm{accept}}$ consists of a minimal set of computation paths witnessing the acceptance of $\langle x, k\rangle$ by R_{W}.

Definition 5.1 *Let the $h(k)$ $\exists_1$-guesses of R_{W} be the values $V_{\exists_1} = \{v_1, v_2, \ldots, v_{h(k)}\}$, respectively. An* acceptance tree*, corresponding to the given $\exists_1$-values, is a subtree $\mathcal{C}_{W,\mathrm{accept}}$ of $\mathcal{C}_W$ such that the following holds.*

W1. The root of $\mathcal{C}_{W,\mathrm{accept}}$ is the universal node at alternation level 2 of $\mathcal{C}_W$.

W2. If a universal node u of $\mathcal{C}_W$ is included in $\mathcal{C}_{W,\mathrm{accept}}$, then all *children of u in $\mathcal{C}_W$ are also included in $\mathcal{C}_{W,\mathrm{accept}}$.*

W3. If an existential node e of $\mathcal{C}_W$ is included in $\mathcal{C}_{W,\mathrm{accept}}$, then exactly one *of the children of e in $\mathcal{C}_W$ is included in $\mathcal{C}_{W,\mathrm{accept}}$.*

W4. The depth of $\mathcal{C}_{W,\mathrm{accept}}$ is $t-1$.

W5. Each leaf node x of $\mathcal{C}_{W,\mathrm{accept}}$ is labelled by a subset $\mathrm{TestSet}_x$ *of the tests* (JEQUAL *or* JZERO) *in the unique accepting execution path for odd t and the unique rejecting execution path for even t. A test is included in* $\mathrm{TestSet}_x$ *if and only if it is an $\exists_1$-test and the test outcome for the given $\exists_1$-values $\{v_1, v_2, \ldots, v_{h(k)}\}$ causes the computation to accept for even t and to continue along the unique accepting execution path for odd t.*

W6. For each leaf node x of $\mathcal{C}_{W,\text{accept}}$, size of TestSet_x *must be at least one for even t and $h'(k)$ for odd t, where $h'(k)$ is the number of $\exists_1$-tests along the unique accepting execution path.*

If R_{W} accepts $\langle x, k\rangle$, then a *witness* can be constructed by (i) specifying a list of values $V_{\exists_1} = \{v_1, \ldots, v_{h(k)}\}$ for the $\exists_1$-registers, and (ii) listing the nodes of the computation tree $\mathcal{C}_W$ that constitute an acceptance tree $\mathcal{C}_{W,\text{accept}}$ for $V_{\exists_1}$. Also, given such a witness, we can construct an accepting computation of R_{W} on $\langle x, k\rangle$.

Let us now consider the computation of a normalized $L[t]$-program R_{L} on some input $\langle x, k\rangle$. By Definition 2.20, the structure of the partial computation tree $\mathcal{C}_L$ of R_{L} is similar to that of $\mathcal{C}_W$ up to alternation level t. However, the presence of ASSIGN operations differentiates the computations after that. In particular, the operands of a test in a normalized $L[t]$-program may be computed from the $\exists_1$-values by a sequence of ASSIGN operations. We define the acceptance tree $\mathcal{C}_{L,\text{accept}}$ of the $L[t]$-program R_{L} by modifying Definition 5.1 to incorporate the information about the assignment graphs. The acceptance tree $\mathcal{C}_{L,\text{accept}}$ must satisfy the (appropriately restated) properties W1 to W4 of Definition 5.1. For the remaining two conditions, we make use of the concept of the $\exists_1$-operand (Definition 2.12). For conditions L5 and L6, we consider each $\exists_1$-test i.e. each test performed on some $\exists_1$-operand. In addition to the TestSet, the label of each leaf includes a set of values, one for each node in the assignment graph of the corresponding computation branch. Let $g(k)$ be the maximum number of nodes in any assignment graph.

The new conditions L5 and L6 are given below.

L5. Each leaf node x of $\mathcal{C}_{L,\text{accept}}$ is labelled by a subset TestSet_x of the tests (JEQUAL or JZERO) in the unique accepting execution path for odd t and the unique rejecting execution path for even t. In addition, the label of x includes a set of values $V_{\text{assign}} = \{v_{x,1}, \ldots, v_{x,g(k)}\}$. V_{assign} specifies that, for the given $\exists_1$-values, the value computed for the i-th node in the corresponding assignment graph is $v_{x,i}$. The values in V_{assign} must be consistent with the relationships among the nodes of the assignment graph.

A test is included in TestSet_x if and only if the test is an $\exists_1$-test and the test outcome for the given $\exists_1$-values $\{v_1, v_2, \ldots, v_{h(k)}\}$ and the values in V_{assign} causes the computation to accept for even t and to continue along the unique accepting execution path for odd t.

L6. For each leaf node x of $\mathcal{C}_{L,\text{accept}}$, size of TestSet_x must be at least one for even t and $h'(k)$ for odd t, where $h'(k)$ is the number of $\exists_1$-tests along the unique accepting execution path.

Both hardness proofs in this section involve construction of *witness-based generic reductions*. In this technique, the hardness of a problem Q with respect to some program R is established by mapping a witness of an accepting computation of R to a solution of Q, and vice versa. Note that the branches of the computation tree that are included in the acceptance tree may vary depending on the corresponding existential guesses. However, by conditions NW3 and NL3, the size of the acceptance tree (and hence the length of the witness) remains the same for all such choices.

Property 5.9 *For a given* normalized $W[t]$*-program (or a normalized* $L[t]$*-program)* R *and an input* $\langle x, k\rangle$ *the length of the witness of an accepting computation of* R *is unique.*

Property 5.9 is crucial for witness-based reductions as some feature of Q (the solution size, for example) needs to be specified uniquely, based on the length of the witness.

5.2.1 Longest Common Subsequence

The Longest Common Subsequence problem takes a set of strings (on a common alphabet) as input and asks for a common subsequence of a particular length. Different parametric versions of the problem have been defined based on the selection of the parameter.

Longest Common Subsequence (LCS)

Input: An alphabet Σ, a set of strings $S = \{s_1, \ldots, s_m\}$, an integer k.
Parameter:
m (LCS-1).
k (LCS-2).
m, k (LCS-3).
m, $|\Sigma|$ (LCS-4).
Question: Is there a string x of length k such that x is a subsequence of each string in S?

LCS-2 is known to be $W[2]$-hard and in $L[2]$ [3, 37]. LCS-3 is known to be $W[1]$-complete [3]. Bodlaender et al. showed that the problems LCS-1 and LCS-4 are $W[t]$-hard for all $t > 0$ [3]. In this section, we show that the parametric problems LCS-1 and LCS-4 are hard for

(a) $L[t]$, for all $t \geq 1$, and

(b) $W[SAT]$.

We base our hardness proof on a new class of programs which we define below.

Definition 5.2 *An AW program R running on an* ARAM *is an $L[SAT]$-program if R satisfies the following conditions.*

LS1: There are at most $h(k)$ nondeterministic guess steps in the first nondeterministic block and these guessed values are stored in guess registers.

LS2: The size of the partial computation tree is bounded by a parametric polynomial.

LS3: In any computation branch, the number of operations that use an $\exists_1$-operand is at most $h'(k)$, for some function h' and all such operations are performed after all the nondeterministic operations. Furthermore, any $\exists_1$-operation is restricted to be one of JEQUAL, JZERO, *or* ASSIGN.

Note that the $L[SAT]$-programs may have unbounded alternations in some computation branch. However the ranges for the nondeterministic operations are such that the size of the partial computation tree remains bounded by a parametric polynomial. Thus a deterministic algorithm can construct the partial computation tree of a given $L[SAT]$-program on a given input in parametric polynomial time. Also any operation can be performed on the values that are guessed nondeterministically after the first existential block. We define the acceptance tree of an $L[SAT]$-program analogous to the acceptance tree of a normalized $L[t]$-program. The only difference is that the acceptance tree of an $L[SAT]$-program may have unbounded depth. We use the concept of acceptance tree of an $L[SAT]$-program in our hardness proofs.

Proving hardness of LCS-1 with respect to the $L[SAT]$-programs is sufficient for our purpose because of the following theorem.

Theorem 5.10 *Let* Q *be a parametric problem.*

1. *If* Q *is in $L[t]$ then* Q *can be decided by an $L[SAT]$-program.*
2. *If* Q *is in $W[SAT]$ then* Q *can be decided by an $L[SAT]$-program.*

PROOF. (1) Immediate as a normalized $L[t]$-program is also an $L[SAT]$-program.

(2) Let C be a tree circuit with unbounded weft and depth. We first show that WCS on C can be decided by an $L[SAT]$-program R_{WFSAT}. R_{WFSAT} constructs lookup tables to indicate which variables satisfy (or does not falsify) a given input gate of C. R_{WFSAT} then existentially guesses the k true variables. In the checking phase, R_{WFSAT} selects an input gate of C using unbounded alternation, making a universal guess (existential guess) for each *and*-gate (*or*-gate). Finally, R_{WFSAT} uses k JZERO tests to determine whether the selected input gate is satisfied (or is not falsified) by the existentially guessed assignment. Since the number of gates in the tree circuit C is bounded by n, the size of the resulting partial computation tree of R_{WFSAT} is bounded by a parametric polynomial.

By definition, any problem Q in $W[SAT]$ can be fixed-parameter reduced to WCS on the family of tree circuits. Thus, Q can be decided by an $L[SAT]$-program that performs the reduction in the preprocessing phase and then simulates R_{WFSAT} on the output of the reduction. ∎

We now show that any problem Q, decidable by an $L[SAT]$-program, can be fixed-parameter reduced to LCS-1.

Theorem 5.11 LCS-1 *is hard for the class of problems decidable by $L[SAT]$-programs.*

We prove Theorem 5.11 in two steps. First we show the hardness result with respect to $L[SAT]$-programs whose acceptance tree for a given input has a unique size (Lemma 5.12). Later we show that for any $L[SAT]$-program, there exists an equivalent $L[SAT]$-program whose acceptance tree has the desired property (Lemma 5.17).

Lemma 5.12 LCS-1 *is hard for problems decidable by $L[SAT]$-programs whose acceptance tree has a unique size for each fixed input.*

PROOF. Let, Q be a parameterized problem decidable by an $L[SAT]$-program R_{Q} as specified in the lemma. We construct a generic reduction that takes an input $\langle x, k\rangle$ and constructs an instance $\langle x', k'\rangle$ of LCS-1 such that R_{Q} accepts $\langle x, k\rangle$ if and only if $\langle x', k'\rangle$ is in LCS-1. Let us assume that R_{Q} makes $h_{\exists}(k)$ existential guesses in the first existential block and performs at most $h_{\text{op}}(k)$ additional $\exists_1$-operations in any computation branch (as specified in Definition 5.2). Let $h_{\text{path}}(k) \leq 2^{h_{\text{op}}(k)}$ be

the maximum number of execution paths in any nondeterministic checking branch. Let q be the maximum number of nondeterministic operations in any computation branch of R_{Q}. Also let l be the maximum value that R_{Q} may store in any register at any point of computation. The reduction can compute all these bounds by constructing the partial computation tree of R_{Q} for input $\langle x, k\rangle$.

Recall that we can uniquely identify a branching node y in the partial computation tree by $\langle b_1, \ldots, b_{q'}\rangle$, where y has q' branching ancestors and b_i is the branch taken at the ith branching ancestor of y to reach y from the root. Also, given a branching node $\langle b_1, \ldots, b_{q'}\rangle$, $\mathcal{B}(\langle b_1, \ldots, b_{q'}\rangle)$ denotes the number of branches generated from the branching node. Consider the computation (starting from the first $\exists_1$-step) along execution path x for a nondeterministic checking branch $b = \langle b_1, \ldots, b_{q'}\rangle$. Let $I_{b,x}$ be the ordered (by step number) set of indices to the registers used or modified by some $\exists_1$-operation in the computation under consideration. Let $g(k) \leq h_{\exists} + 3h_{\mathrm{op}}$ be the size of $I_{b,x}$. We assume that $I_{b,x}$ attaches the step number to the indices so that multiple assignments to the same register result in different entries in $I_{b,x}$. Let $I_{b,x}[j]$ be the jth member in $I_{b,x}$.

In what follows, we use $\bigodot$ to denote repeated concatenation of strings, where the order of repetition is specified with the operator $\bigodot$. Given two strings $s_1, s_2 \in \Sigma^*$, we use $s_1 \cdot s_2$ to denote the concatenation of s_1 and s_2. The input alphabet Σ of LCS-1 consists of three kinds of symbols.

[Type1]: The first kind of symbols are of the form $\langle i, v_i\rangle$, $1 \leq i \leq h_{\exists}$, $0 \leq v_i \leq l$. The symbol $\langle i, v_i\rangle$ represents the fact that the ith guess in the first existential block is the value v_i.

[Type2:] Symbols of the second kind are of the form $\langle b_1, \ldots, b_{q'}, x, i, v_i, j, v_j\rangle$, where $\langle b_1, \ldots, b_{q'}\rangle$, for some $q' \leq q$, is the last nondeterministic node along some branch and

$$\begin{aligned} &1 \leq x \leq h_{\mathrm{path}}, \\ &1 \leq i < j \leq g \\ &0 \leq v_i, v_j \leq l, \\ &1 \leq b_u \leq \mathcal{B}\left(\langle b_1, \ldots b_{u-1}\rangle\right) \\ &\qquad \text{for each } u, 1 \leq u \leq q' - 1 \end{aligned}$$

The symbol $\langle b_1, \ldots, b_{q'}, x, i, v_i, j, v_j\rangle$ corresponds to the computation in the nondeterministic checking branch $\langle b_1, \ldots, b_{q'}\rangle$ along the execution path x, with values v_i and v_j in the ith and jth registers in $I_{\langle b_1, \ldots, b_{q'}\rangle, x}$, respectively.

[Type3:] Symbols of the third kind are of the form

$$\langle b_1, \ldots, b_{q'}, x, y, y_1, v_{y_1}, y_2, v_{y_2}, y_3, v_{y_3}\rangle$$

where $\langle b_1, \ldots, b_{q'} \rangle$, for some $q' \leq q$, is the last nondeterministic node along some branch and

$$\begin{aligned}
&1 \leq x \leq h_{\text{path}}, \\
&1 \leq y \leq h_{\text{op}}, \\
&1 \leq y_1, y_2, y_3 \leq g, \\
&0 \leq v_{y_1}, v_{y_2}, v_{y_3} \leq l, \\
&1 \leq b_u \leq \mathcal{B}\left(\langle b_1, \ldots b_{u-1} \rangle\right) \\
&\qquad \text{for each } u, 1 \leq u \leq q' - 1
\end{aligned}$$

The symbol $\langle b_1, \ldots, b_{q'}, x, y, y_1, v_{y_1}, y_2, v_{y_2}, y_3, v_{y_3} \rangle$ corresponds to the computation in the nondeterministic checking branch $\langle b_1, \ldots, b_{q'} \rangle$ along the execution path x, such that the yth ASSIGN operation has the y_1th and y_2th members of $I_{\langle b_1, \ldots, b_{q'} \rangle, x}$ as operands and the y_3th member in $I_{\langle b_1, \ldots, b_{q'} \rangle, x}$ as target. The values in the operands are v_{y_1} and v_{y_2}, respectively, and v_{y_3} is the result of the ASSIGN operation.

The strings are constructed based on the following easy observations which are due to bodlaender et al. [3].

Observation 5.13 *Let S_1 be a string where all the symbols are distinct. Let S_2 be the reverse string of S_1. The length of any common subsequence of S_1 and S_2 is at most one.*

Observation 5.14 *Let $S_1, \ldots, S_m$ be strings such that each S_i has at least one* distinguished symbol *that does not appear in any other string S_j, $i \neq j$. Any common subsequence of $S_1 \cdot S_2 \cdot \ldots \cdot S_{m-1} \cdot S_m$ and $S_m \cdot S_{m-1} \cdot \ldots \cdot S_2 \cdot S_1$ can include distinguished symbols from at most one S_i, $1 \leq i \leq m$.*

We now construct the strings of the LCS instance. Let us denote the common subsequence (if it exists) by CS. We start by defining some helper strings that will be used to construct the actual strings.

The first two strings consist of Type1 symbols only.

$$C_1 = \bigodot_{i=1}^{h_\exists} \bigodot_{v_i=0}^{l} \langle i, v_i \rangle$$

$$C_1' = \bigodot_{i=1}^{h_\exists} \bigodot_{v_i=l}^{0} \langle i, v_i \rangle$$

We define the following strings to represent the computation in the checking phase of any given nondeterministic checking branch $\langle b_1, \ldots, b_{q'} \rangle$ along a given execution path x.

$$\mathrm{CHECK_W}\left(\langle b_1, \ldots, b_{q'} \rangle, x\right) = \overset{g}{\underset{i=1}{\odot}} \; \overset{l}{\underset{v_i=0}{\odot}} \; \overset{g}{\underset{j=i+1}{\odot}} \; \overset{l}{\underset{v_j=0}{\odot}} \; \langle b_1, \ldots, b_{q'}, x, i, v_i, j, v_j \rangle$$

$$\mathrm{CHECK_L}\left(\langle b_1, \ldots, b_{q'} \rangle, x\right) = \overset{h_{\mathrm{op}}}{\underset{y=1}{\odot}} \; \overset{g}{\underset{y_1=1}{\odot}} \; \overset{l}{\underset{v_{y_1}=0}{\odot}} \; \overset{g}{\underset{y_2=1}{\odot}} \; \overset{l}{\underset{v_{y_2}=0}{\odot}} \; \overset{g}{\underset{y_3=1}{\odot}} \; \overset{l}{\underset{v_{y_3}=0}{\odot}} \; \langle b_1, \ldots, b_{q'}, x, y, y_1, v_{y_1}, y_2, v_{y_2}, y_3, v_{y_3} \rangle$$

$$\mathrm{CHECK}\left(\langle b_1, \ldots, b_{q'} \rangle, x\right) = \mathrm{CHECK_W}\left(\langle b_1, \ldots, b_{q'} \rangle, x\right) \cdot \mathrm{CHECK_L}\left(\langle b_1, \ldots, b_{q'} \rangle, x\right)$$

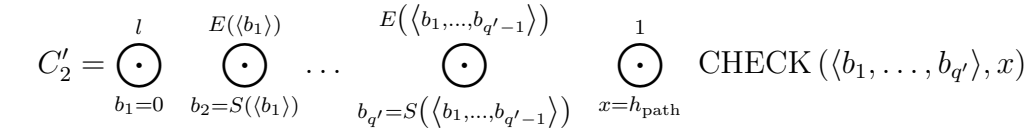
$$C_2 = \overset{l}{\underset{b_1=0}{\odot}} \; \overset{\mathcal{B}(\langle b_1 \rangle)}{\underset{b_2=0}{\odot}} \; \ldots \; \overset{\mathcal{B}(\langle b_1, \ldots, b_{q'-1} \rangle)}{\underset{b_{q'}=0}{\odot}} \; \overset{h_{\mathrm{path}}}{\underset{x=1}{\odot}} \; \mathrm{CHECK}\left(\langle b_1, \ldots, b_{q'} \rangle, x\right)$$

For the next string C_2', we define the functions S and E each of which takes a nondeterministic node x of the partial computation tree as argument. S and E return the starting and ending values, respectively, for the index of the $\odot$ operator that corresponds to the branching from x.

$$S(\langle b_1, \ldots, b_j \rangle) = \begin{cases} 1, & \text{if } \langle b_1, \ldots, b_j \rangle \text{ is a universal node} \\ \mathcal{B}(\langle b_1, \ldots, b_j \rangle), & \text{if } \langle b_1, \ldots, b_j \rangle \text{ is an existential node} \end{cases}$$

$$E(\langle b_1, \ldots, b_j \rangle) = \begin{cases} \mathcal{B}(\langle b_1, \ldots, b_j \rangle), & \text{if } \langle b_1, \ldots, b_j \rangle \text{ is a universal node} \\ 1, & \text{if } \langle b_1, \ldots, b_j \rangle \text{ is an existential node} \end{cases}$$

The string C_2' is constructed from C_2 by reversing the index of all $\odot$ operator that correspond to existential guess operations.



The instance $\langle x', k'\rangle$ includes $(2g(k)+3)$ strings denoted by $S_1, \ldots S_{2g(k)+3}$. The first string S_1 enumerates all symbols in Σ in canonical order.

$$S_1 = C_1 \cdot C_2$$

The second string S_2 is constructed as follows.

$$S_2 = C_1' \cdot C_2'$$

The second string S_2, together with S_1, ensures that CS satisfies the conditions in the following lemma whose proof will be given later.

Lemma 5.15

- *CS includes at most one $\langle i, v_i\rangle$, for each i, $1 \leq i \leq h_\exists(k)$.*
- *For any existential node x in the partial computation tree, CS includes symbols from at most one subtree $T_{x,i}$, for some i, $1 \leq i \leq \mathcal{B}(x)$, where $T_{x,i}$ is the subtree rooted at the ith child of x.*
- *For any two symbols $\langle b_1, \ldots, b_{q'}, x, \star, \star, \star, \star\rangle$ and $\langle b_1', \ldots, b_{q''}', x', \star, \star, \star, \star\rangle$ (here $\star$ denotes any value in the corresponding range) in CS, $x = x'$ whenever $q' = q''$ and $b_i = b_i'$ for all i, $1 \leq i \leq q'$.*

The first condition translates to the fact that at most one value is specified for each $\exists_1$-register. The second condition corresponds to the fact that an acceptance tree includes at most one child of an existential node. The third condition corresponds to the fact that the acceptance tree represents the computation along at most one accepting execution path for each nondeterministic checking branch.

Next we define two strings S_{2u+1} and S_{2u+2} for the uth operand, used in the checking phase by some $\exists_1$-operation, $1 \leq u \leq g(k)$. These two strings ensure that the following holds for each execution path in each nondeterministic checking branch.

Lemma 5.16 *Given a nondeterministic checking branch $b = \langle b_1, \ldots, b_{q'} \rangle$, and an execution path x for b, the symbols in the common subsequence CS specify the same v_u in all places where the uth member in $I_{b,x}$ is referred to.*

The proof of the lemma will be given after we describe the construction of the strings. Let us consider the computation along the execution path x for some nondeterministic checking branch $b = \langle b_1, \ldots, b_{q'} \rangle$. Given a condition φ and a string S, we define restrict(S, φ) to be the maximal subsequence S' of S such that each symbol in S' satisfies the condition φ. For example, RESTRICT $(C_1, (i = u) \implies (v_i = x))$ will result in the following string.

$$\text{RESTRICT}\,(C_1, [(i = u) \implies (v_i = x)]) = \left[\bigodot_{i=1}^{u-1} \bigodot_{v_i=0}^{l} \langle i, v_i \rangle\right] \cdot \langle u, x \rangle \cdot \left[\bigodot_{i=u+1}^{k} \bigodot_{v_i=0}^{l} \langle i, v_i \rangle\right]$$

Let φ be defined as follows.

$$\begin{aligned} \varphi(b, x, u, v_u) = \; & [(i = I_{b,x}[u]) \Rightarrow (v_i = v_u)] \\ & \wedge [(j = I_{b,x}[u]) \Rightarrow (v_j = v_u)] \\ & \wedge [(y_1 = I_{b,x}[u]) \Rightarrow (v_{y_1} = v_u)] \\ & \wedge [(y_2 = I_{b,x}[u]) \Rightarrow (v_{y_2} = v_u)] \\ & \wedge [(y_3 = I_{b,x}[u]) \Rightarrow (v_{y_3} = v_u)] \end{aligned}$$

The strings S_{2u+1} and S_{2u+2} are defined as follows.

$$S_{2u+1} = \bigodot_{v_u=0}^{l} \bigodot_{b_1=0}^{l} \bigodot_{b_2=0}^{\mathcal{B}(\langle b_1 \rangle)} \cdots \bigodot_{b_{q'}=0}^{\mathcal{B}(\langle b_1, \ldots, b_{q'-1} \rangle)} \bigodot_{x=1}^{h_{\text{path}}} \text{RESTRICT}(\ \text{CHECK}(b,x),\ \varphi(b,x,u,v_u)\ )$$

$$S_{2u+2} = \bigodot_{v_u=l}^{0} \bigodot_{b_1=0}^{l} \bigodot_{b_2=0}^{\mathcal{B}(\langle b_1 \rangle)} \cdots \bigodot_{b_{q'}=0}^{\mathcal{B}(\langle b_1, \ldots, b_{q'-1} \rangle)} \bigodot_{x=1}^{h_{\text{path}}} \text{RESTRICT}(\ \text{CHECK}(b,x),\ \varphi(b,x,u,v_u)\ )$$

where $b = \langle b_1, \ldots, b_{q'} \rangle$.

The last string $S_{2g(k)+3}$ is constructed as the longest subsequence of S_1 by removing from S_1 all symbols σ_{invalid} that are inconsistent with the partial computation tree. Any remaining symbol σ in $S_{2g(k)+3}$ satisfies one of the following conditions.

- σ is of the form $\langle i, v_i \rangle$.
- σ is of the form $\langle b_1, \ldots, b_{q'}, x, i, v_i, j, v_j \rangle$, the execution path x for the nondeterministic checking branch $\langle b_1, \ldots, b_{q'} \rangle$ ends in *acceptance*, and no test outcome causes the computation to branch away from x if the values v_i and v_j are stored in the ith and jth members in $I_{\langle b_1,\ldots,b_{q'}\rangle,x}$ respectively.
- σ is of the form $\langle b_1, \ldots, b_{q'}, x, y, y_1, v_{y_1}, y_2, v_{y_2}, y_3, v_{y_3} \rangle$, the execution path x for the nondeterministic checking branch $\langle b_1, \ldots, b_{q'} \rangle$ ends in *acceptance*, and the ASSIGN operation associated with the yth node in the assignment graph has (i) the y_1th, y_2th, and y_3th members in $I_{\langle b_1,\ldots,b_{q'}\rangle,x}$ as the first operand, the second operand, and the target respectively, and (ii) the operation computes the value v_{y_3} if the values in the operands are v_{y_1} and v_{y_2}, respectively. Note that for a given $\langle b_1, \ldots, b_{q'}, x, y \rangle$ the values of y_1, y_2, and y_3 are fixed.

This ends the construction of the strings in $\langle x', k' \rangle$. The length of the target common subsequence k' is set to be $h_\exists + l_{\text{acc}} \left(\binom{g(k)}{2} + h_{\text{op}} \right)$, where l_{acc} is the number of leaves in the acceptance tree of R_{Q}.

We now prove the correctness of the construction. We start with the proofs of Lemma 5.15 and 5.16.

PROOF.(Proof of Lemma 5.15) For a given i, the symbols $\langle i, v_i \rangle$ are stored in increasing order of v_i in S_1 and in decreasing order of v_i in S_2. Let us assume that the common subsequence CS contains two symbols $\langle i, v_i \rangle$ and $\langle i, v_i' \rangle$, $v_i < v_i'$, and $\langle i, v_i \rangle$ appears before $\langle i, v_i' \rangle$ in CS. In this case, CS cannot be a subsequence of S_2 as $\langle i, v_i \rangle$ appears after $\langle i, v_i' \rangle$ in S_2 (Observation 5.14).

For the second condition, we observe that the indices for any given existential node x varies in increasing order in C_2 and in decreasing order in C_2'. Thus any symbol in CS must have a unique value for x by Observation 5.14.

Similarly, the third condition follows from Observation 5.14 and the fact that the indices of the execution paths of any given nondeterministic checking branch varies in increasing order in C_2 and in decreasing order in C_2'. ∎

PROOF.(Proof of Lemma 5.16) By construction of S_{2u+1}, any symbol σ_{u,v_u} specifying value v_u for the uth guess register appear before any symbol $\sigma_{u,v_u'}$ specifying value v_u' for the same register, $v_u < v_u'$. On the other hand, The σ_{u,v_u} appears after $\sigma_{u,v_u'}$ in S_{2u+2}. By Observation 5.14, the symbols in CS must specify a unique value for the uth register used in the checking phase in some nondeterministic checking branch. ∎

We claim that R_{Q} accepts $\langle x, k\rangle$ if and only if the strings in x' have a common subsequence of length $k' = h_{\exists} + l_{\mathrm{acc}}\left(\binom{g(k)}{2} + h_{\mathrm{op}}\right)$. First we show that any such common subsequence CS of the strings in x' can be translated back into an accepting computation of R_{Q} on $\langle x, k\rangle$. CS has at most one $\langle i, v_i\rangle$ (a symbol from C_1) for each i, $1 \leq i \leq h_{\exists}$ (this constraint is enforced by strings S_1 and S_2). Consider the Type3 symbols in CS that correspond to a particular nondeterministic checking branch $b = \langle b_1, \ldots, b_{q'}\rangle$ and execution path x. By construction of the last string $S_{2g(k)+3}$ and the consistency imposed by the strings S_3 to $S_{2g(k)+2}$, CS can include at most h_{op} such symbols for each combination of b and x. At most $\binom{g(k)}{2}$ quadruples $\langle i, v_i, j, v_j\rangle$ can be consistent with the Type1 and Type3 symbols in CS for the combination of b and x (strings S_j, $3 \leq j \leq (2g(k) + 2)$ ensure this constraint). Hence, for each combination of a nondeterministic checking branch b and execution path x, CS can contain at most $\binom{g(k)}{2}$ Type2 symbols. The second parts of strings S_1 and S_2 ensure that all Type2 and Type3 symbols in CS have a unique x for a given b.

All these facts, together with the requirement that CS must contain k' symbols, imply that CS satisfies the following conditions.

- CS contains exactly one symbol $\langle i, v_i\rangle$, $1 \leq i \leq h_{\exists}$. This translates to the fact that the ith $\exists_1$-guess, in the computation of R_{Q}, is v_i.
- CS contains exactly h_{op} symbols of Type3 for each nondeterministic checking branch b. Also all these symbols correspond to some unique execution path x for b. The last string ensures that the values specified for the operands and the result of each $\exists_1$-operation are consistent with the corresponding partial configurations.
- CS contains exactly $\binom{g(k)}{2}$ Type2 symbols for a nondeterministic checking branch b. Also all these symbols correspond to some unique execution path x for b. The last string ensures that execution path x ends in acceptance for b and none of the test-outcomes along x causes the computation to branch away from x.

Thus, existence of a common subsequence of length k' for the strings in x' implies that R_{Q} accepts $\langle x, k\rangle$.

For the reverse direction, we construct a common subsequence CS of the required length from an acceptance tree of R_{Q} as follows.

- We include the $h_{\exists}$ symbols of C_1 (in canonical order) that represent the $h_{\exists}$ existential guesses of R.
- For each leaf in the acceptance tree, we choose h_{op} Type3 symbols, the yth symbol representing the values of the operands and result of the yth ASSIGN operation. These values can be obtained from the corresponding nodes in the assignment graph, the specified $\exists_1$-values, and the labels on the leaves of the acceptance tree.
- Let, A be the set of $\binom{g(k)}{2}$ quadruples $\langle i, v_i, j, v_j \rangle$ that are consistent with the symbols chosen in the previous steps. For each leaf of the acceptance tree, we include $\binom{g(k)}{2}$ symbols from C_2 (in canonical order) that represent the quadruples in A for an accepting execution path in the corresponding nondeterministic checking branch.

∎

Lemma 5.17 *For any $L[SAT]$-program R_1, there exists another $L[SAT]$-program R_2 such that*

Pad-L1: the size of the acceptance tree of R_2 for any given input is unique, and

Pad-L2: R_1 accepts $\langle x, k \rangle$ if and only if R_2 accepts $\langle x, k \rangle$.

PROOF. The $L[SAT]$-program R_2 is constructed as follows.

Step-1 In the preprocessing phase, construct the partial computation tree C_1 that represents the computation of R_1 on $\langle x, k \rangle$.

Step-2 Convert C_1 into a partial computation tree C_2 for input $\langle x, k \rangle$ such that any acceptance tree for C_2 has a unique size and the root of C_2 is accepting if and only if the root of C_1 is accepting.

Step-3 Compute according to C_2.

The rest of the proof presents the details of Step-1 and Step-2.

[Step-1]: R_2 can construct the root of C_1 from $\langle x, k \rangle$. R_2 constructs the rest of C_1 by computing the partial configurations reachable from a given partial configuration. The entire construction of C_1 takes parametric polynomial time as the size of the partial computation tree is bounded by some parametric polynomial.

[Step-2]: As a first step R_2 ensures that along any branch, (i) any pair of existential partial configurations has at least one universal partial configuration between them and (ii) the last nondeterministic operation in any computation branch is universal. These restrictions facilitate the next phase of the construction. Let conf_1 and conf_2 be two existential partial configurations in C_1 such that conf_2 is a descendent of conf_1 and no nondeterministic node appears in the path from conf_1 to conf_2 (i.e. conf_1 and conf_2 violate (i)). R_2 inserts an universal configuration conf_3 immediately before conf_2. The operation associated with conf_3 universally guesses a value equal to zero and stores it in some guess register that is not used by any computation step of R_1. Thus the operation has no effect on the acceptance of $\langle x, k\rangle$ by R_1. Violations of condition (ii) are dealt with similarly. Let C_1' be the partial computation tree obtained after the entire partial computation tree C_1 has been processed as above. Introduction of each new universal configuration increases the number of partial configurations in the tree by one and does not increase the number of branches. Thus the size of C_1' remains bounded by a parametric polynomial.

Construction of C_2 from C_1' is the most important part of the proof. New branches are added to certain nodes in C_1' (described later) so that the number of leaves in the acceptance tree rooted at a given existential node remains the same irrespective of the choice of an existential branch. We refer to such addition of branches as *padding*. New operations are included to immediately terminate any branch that arises due to padding. Thus padding branches do not affect the correctness of the computation. Formally, C_2 is constructed in such a way that R_2 performs the sequence of operations in Algorithm 5.6 whenever R_1 makes a universal guess. Let $l_{\text{pad}} - 1$ be the maximum value that R_1 may store in any register at any point of computation. For the purpose of terminating the padded branches, the lookup table is constructed so that $r_{\langle v_1, v_2\rangle}$ is 0 for any value v_1 as long as $v_2 \geq l_{\text{pad}}$. With this formulation, the task of padding reduces to computing the padded range of values for each universal node (i.e. PaddedRange[*curnode*] in Algorithm 5.6). The rest of the proof describes the computation in detail.

Let x be an existential node in C_1' and let $T_{x,i}$ and $T_{x,j}$ be the subtrees rooted at the ith and jth child of x, respectively. Also let the size of the acceptance trees (after padding) for $T_{x,i}$ and $T_{x,j}$ be l_i and l_j, respectively. Let us consider the scenario where $l_i > l_j$. In this case R_2 modifies $T_{x,j}$ to $T'_{x,j}$ so that the size of the acceptance tree of $T'_{x,j}$ equals l_i for any input. The key idea is to increase the number of branches for the root of $T_{x,j}$ in such a way that the acceptance conditions are not affected and that the overall size of the tree remains bounded by a parametric polynomial.

Algorithm 5.6: Sequence of operations that replaces each universal guess step.

Assumptions:

- *curnode* refers to the universal configuration in C' whose operation is to be replaced.
- $l_{\text{pad}} - 1$ is the maximum value that R_1 may store in any register .

1. $r_{\text{temp}} \leftarrow r_0$
2. $r_0 \leftarrow$ PaddedRange[*curnode*]
   /* (FORALL $\uparrow j$) */
3. $g_{r_j} \leftarrow$ Universally guess a value from [0 . . . PaddedRange[*curnode*]]
4. $g_{\text{limit}} \leftarrow$ OriginalRange[*curnode*]
5. If $g_{\text{limit}} < g_{r_j}$ then
6. $\quad g_{r_j} \leftarrow g_{r_j} + l_{\text{pad}}$
7. $\quad$ For each $\exists_1$-register g_i
8. $\quad\quad$ If $r_{\langle g_i, g_{r_j} \rangle} = 0$ then terminate (accept in) this branch
9. $\quad$ EndFor
10. $\quad$ Reject in this branch.
11. Endif
12. $r_0 \leftarrow r_{\text{temp}}$

Algorithm 5.7 presents the construction of C_2. The padding is done by traversing the partial computation tree C_1' in postorder. Let us consider an *existential* partial configuration x and the subtree T_x of C_1' rooted at x. Let $T_{x,i}$ be the subtree rooted at the ith child of x. No padding is needed for x in case it is the only (and hence last) nondeterministic operation in the subtree rooted at x. The reason is that the acceptance tree for T_x always includes a single leaf (computation branch) from T_x. For the general case, let us assume that the subtrees $T_{x,i}$ s have been padded so that the acceptance tree for $T_{x,i}$ always includes l_i leaves. If all l_i s are not the same, padding has to be done for x (Lines 17-28). Let $l_{\max}$ be the maximum of the l_is, $1 \leq i \leq \mathcal{B}(x)$. For each $T_{x,i}$, the padding algorithm adds $l_{\max} - l_i$ additional (padding) branches to the universal node in $T_{x,i}$ that follows x (Figure 5.2). After the padding, the size of the acceptance tree for T_x is always $l_{\max}$ independent of computation branch selected by the existential guess. Let us now consider a universal node y. If the subtrees $T_{y,i}$ satisfy property Pad-C1, y

Algorithm 5.7: Padding a partial computation tree so that the size of the acceptance tree remains the same for all choices of existential branching.

ComputeRange
Input: A partial computation tree C, a node x in C, a list PaddedRange.
Output: Integer.
After the computation PaddedRange[i] is set to the number of branches that node i in C should have after padding.

```
   /* Base cases:  No padding is needed for x                          */
1  if x is a leaf then
2      PaddedRange[x] = 1 Return 1
3  end
4  if x is a non branching node then
5      PaddedRange[x] = ComputeRange(C,y), where y is the child of x
6      Return PaddedRange[x]
7  end
   /* Recursive cases:                                                 */
8  MaxLeafCount ← 0
9  LeafCount ← 0
10 if x is a universal node then
11     for i = 1 to B(x) do
12         Let x_i be the ith child of x
13         LeafCount ← LeafCount + ComputeRange(C,x_i)
14     end
15     PaddedRange[x] ← LeafCount
16     Return LeafCount
17 else
18     for i = 1 to B(x) do
19         Let x_i be the ith child of x
20         LeafCount ← ComputeRange(C,x_i)
21         if LeafCount > MaxLeafCount then MaxLeafCount ← LeafCount
22     end
23     for i = 1 to B(x) do
24         Let u_i be the first universal node in the subtree for the ith child of x
25         PaddedRange[u_i] ← MaxLeafCount
26     end
27     Return MaxLeafCount
28 end
```

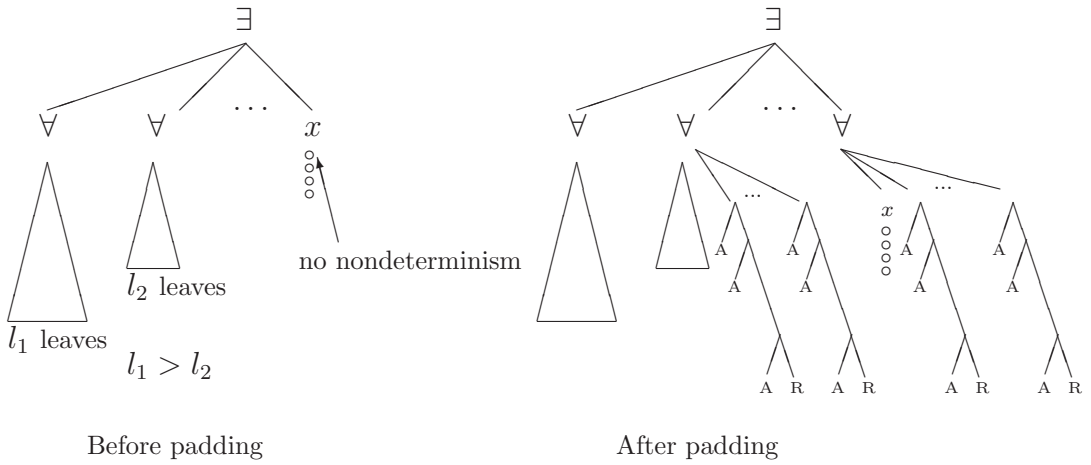


Figure 5.2: Padding for an *existential*-node.

also satisfies Pad-C1. This is because all subtrees $T_{y,i}$ must be satisfied in order to satisfy y. The padding algorithm simply returns to the parent of y in such cases (Lines 2 and Lines 10-15).

It is important to note that the effect of padding is confined to a single level and does not propagate downwards. Thus, padding for an existential node x at any level increases the size of the circuit by at most $\mathcal{B}(x) \times (l-1)$, where l is the number of leaves in the original partial computation tree. Since the number of existential nodes is bounded by n, padding for all existential nodes increases the size of the circuit by a polynomial factor only.

Let us now analyze the runtime of the padding algorithm. The non-recursive part of processing a universal node takes constant time. Non-recursive part of processing an existential node x involves adding at most $(l-1)$ padding branches to at most $\mathcal{B}(x)$ universal nodes. Thus padding for an existential node requires $O(\mathcal{B}(x) \times (l-1))$ time. The postorder traversal performs the non-recursive part of the processing for each node in C_1' once. Thus C_2 can be constructed in polynomial time.

Correctness: The padding branches are terminated without affecting the computation in other sibling branches. Any newly introduced nondeterministic operation stores the guessed value in a register that was not used by R_1. Thus the padding does not affect the correctness of the original computation of R_1 on $\langle x, k \rangle$.

∎

Corollary 5.18 *The following parametric problems are hard for $L[t]$, $t \geq 1$, and $W[SAT]$.*

- Longest Common Subsequence, *parameterized by the number of strings and the size of the alphabet.*
- Bounded DFA Intersection *parameterized by the number of DFA.*
- Bounded DFA Intersection *parameterized by the number of DFA and the alphabet size.*
- (Directed) Colored Cutwidth.
- Domino Treewidth.
- Feasible Register Assignment.
- Module Allocation.
- I/O Deterministic FST Composition *parameterized by number of transducers.*
- I/O Deterministic FST Intersection *parameterized by number of transducers.*
- Intervalizing Colored Graphs.
- Triangulating Colored Graphs.

Proof. LCS-1 can be fixed-parameter reduced to each of the problems mentioned above [2, 3, 7, 41].
∎

5.2.2 Precedence-Constrained Multiprocessor Scheduling

The Precedence-Constrained Multiprocessor Scheduling problem is defined as follows.

Precedence-Constrained Multiprocessor Scheduling (PCMS)

Input: A set of unit length tasks T, a partial order $\prec$ on the tasks in T, an integer $\mathcal{D}$ specifying the deadline, an integer k specifying the number of processors.
Parameter: k.
Question: Is there a mapping $\mathcal{A} : T \to \{1, \ldots \mathcal{D}\}$ such that for all $t_1, t_2 \in T$, $t_1 \prec t_2 \Rightarrow \mathcal{A}(t_1) < \mathcal{A}(t_2)$, and for all i, $1 \leq i \leq \mathcal{D}$, $|\mathcal{A}^{-1}(i)| \leq k$?

The parameterized complexity of the problem was analyzed by Bodlaender and Fellows [5]. They showed that the problem is $W[2]$-hard by constructing a fixed parameter reduction from Dominating Set. The main result presented in this section is the following.

Theorem 5.19 Precedence-constrained Multiprocessor Scheduling *is hard for $L[t]$, for any $t > 0$.*

We construct a generic reduction to show the hardness result. Our construction builds upon the original proof by Bodlaender and Fellows [5] and uses new combinatorics to deal with the additional levels of alternation as well as the computational features of a normalized $L[t]$-program. We construct the proof in three steps. First we show how the computation of a normalized $W[t]$-program can be represented by a scheduling of tasks. We then extend the construction to incorporate the computational features of a normalized $L[t]$-program. This gives us one direction of the hardness proof. For the other direction, we extend the construction for $L[t]$ to ensure that any valid scheduling represents an accepting computation of the corresponding normalized $L[t]$-program. The rest of this section presents the details of the construction.

5.2.2.1 From an Accepting Computation to a Scheduling of tasks

We start with the description of the basic components of the construction and some terminology. Let us consider the computation of a normalized $W[t]$-program R_{W} on some input $\langle x, k \rangle$. Let us assume that R_{W} makes $h(k)$ existential guesses in the first nondeterministic block. Each of the subsequent nondeterministic blocks has one nondeterministic step (existential or universal) as R_{W} is a normalized $W[t]$-program. Note that the bound $h(k)$ implies that R_{Q} performs at most $h(k)$ $\exists_1$-tests, each involving one $\exists_1$-register. Also we assume that the range of values that any register may store is $[0 \ldots l-1]$, where l is bounded above by some parametric

polynomial. In the remainder of this section, we refer to the nondeterministic steps in the computation by the level of alternation. For example, the first existential steps are at level 1, the universal step that follows them is at level 2, the next existential step is at level 3 and so on.

We now define the terminology for scheduling. The definitions apply to both $W[t]$ and $L[t]$ unless stated otherwise. We refer to a unit of time as a *time slot*. Since each task is of unit length, a task is scheduled in exactly one time slot. A sequence of consecutive time slots is referred to as a *time phase*. The precise definitions of the number and length of the time phases will be given later. We say that an edge in the computation tree is at level j if the associated parent node corresponds to j-th level of alternation, $2 \leq j \leq t$. Each edge, from level 2 to level t in the computation tree, corresponds to a unique time phase. Thus, each node in the computation tree corresponds to l time phases determined by the associated edges. We label the time phases by the level of the edge they correspond to. Thus a level-j edge in the computation tree corresponds to a unique level-j time phase, $2 \leq j \leq t$. The time phases for different levels are organized to represent the structure of the computation tree. The time span, consisting of all time slots except the first $(l-1)$, is divided into l non-overlapping level-2 time phases, one time phase for each level-2 edge in the computation tree. In general, each level-j time phase is refined into l non-overlapping level-$(j+1)$ time phases, $2 \leq j \leq (t-1)$. The refinement is done in such a way that, for any pair of time phases p_1 and p_2 corresponding to edges e_1 and e_2 in the computation tree, respectively, the following holds.

1. If e_2 appears in the subtree rooted at the child node associated with e_1, then time phase p_1 begins before p_2 begins and p_1 ends after p_2 ends (i.e. p_1 contains p_2).

2. If e_1 and e_2 have the same parent, and e_1 appears before e_2 in depth first traversal of the computation tree, then p_1 and p_2 are non-overlapping and p_1 ends before p_2 starts.

While the time phases are defined based on the structure of the computation tree, the set of unit length tasks is constructed based on the structure of the acceptance tree. We start with the definitions of tasks for $W[t]$ and describe the additional tasks for $L[t]$ later. Note that the actual nodes of the computation tree $\mathcal{C}_W$ that constitute an acceptance tree may vary depending on the $\exists_1$-values. For the purpose of defining the tasks, it is sufficient to know the structure of the acceptance tree and the required number of tests in the testsets at the leaves of the

acceptance tree, both of which are independent of the $\exists_1$-values. We view the construction of the acceptance tree $\mathcal{C}_{W,\text{accept}}$ from the computation tree $\mathcal{C}_W$ as placing a set of markers on the edges of the computation tree. In this context, the acceptance tree consists of the marked edges (and associated nodes) of the computation tree. A total of $l^{\lfloor j/2 \rfloor}$ markers are available to mark the edges at level j, $2 \leq j \leq t$. As before, we refer to the markers by the level of alternation and say that a marker is *universal* (*existential*) if the corresponding level of alternation is even (odd). The markers are related by parent-child relationship. Also, we assume that all sibling markers are canonically ordered. Let $\langle x', k' \rangle$ be the instance of PCMS being constructed. Each marker is represented by a distinguished set of base tasks in $\langle x', k' \rangle$. The fact that a marker y is placed on an edge x at level j of the computation tree is represented by scheduling all base tasks for y in the level-j time phase corresponding to x. In the reverse direction, if all tasks from the base set of a level-j marker y are scheduled in a single level-j time phase corresponding to some edge x in the computation tree, we interpret the fact as x being included in the acceptance tree at position y.

A separate set of base tasks is included for each $\exists_1$-register. The fact that the i-th $\exists_1$-register is assigned the value v_i is represented by scheduling the corresponding base tasks consecutively starting from the v_i-th time slot. For each node at level t of the computation tree, additional tasks are included to represent the effect of the values in the $\exists_1$-registers on the $\exists_1$-tests in the corresponding computation branch. In addition, a set of test-tasks are included to represent the $\exists_1$-tests corresponding to a level-t-marker. The number and precedences of these tasks are defined so that under certain conditions, a scheduling is possible if and only if the corresponding computation branch accepts.

The computation tree and an acceptance tree for a normalized $L[t]$-program differs from those for a normalized $W[t]$-program from level-t. Thus the concepts of markers and definitions of tasks apply to a normalized $L[t]$-program as well. However, for an $L[t]$-program, the construction of an acceptance tree involves the selection of a consistent set of values for the nodes in the assignment graph. $g(k)$ additional base sets of tasks are included for each leaf of the computation tree to represent the selection of values for the nodes in the corresponding assignment graph. Each of these base sets is functionally similar to the base set of an $\exists_1$-register. However, the effect of the base tasks for a node in the assignment graph are confined to a single level-t time phase.

Although the ideas, mentioned above, allow us to construct a valid scheduling from an accepting computation, they are not sufficient to ensure that the proof works in the reverse direction. A valid scheduling may schedule the base tasks for

a level-j marker in more than one level-j time phases. In such cases, we can not construct a valid witness from the scheduling. In order to deal with such issues, we include additional time phases and constraint tasks so that a valid scheduling always represents a valid witness.

In the next two subsections, we describe how to map an accepting computation of a normalized $W[t]$-program and a normalized $L[t]$-program, to valid schedulings. In these steps, we describe a basic organization of tasks that closely follow the structure of the associated computation trees and acceptance trees. In the last step, we will extend the basic construction in order to make the proof work in the reverse direction.

5.2.2.2 Representing Computation of a $W[t]$-program by a Scheduling

Let us consider a parametric problem Q in $W[t]$, $t \geq 1$. By Theorem 2.13, there exists a normalized-$W[t]$-program R_{Q} to decide Q. The generic reduction $\mathcal{A}$ takes an input $\langle x, k\rangle$ and computes $\langle x', k'\rangle$ such that $\langle x', k'\rangle$ is in PCMS if and only if R_{Q} accepts $\langle x, k\rangle$. The key idea of the construction is to represent the witness of an accepting computation by a particular scheduling of tasks under a given set of precedence rules. Our goal is to ensure that any accepting computation of R_{Q} results in an arrangement of tasks such that all the given precedence conditions are respected and the number of processors required for any time slot does not exceed a given bound.

We start by constructing a string S_W, which we call the *scaffold*, by performing a depth-first traversal (up to level t) of the computation tree $\mathcal{C}_W$. Algorithm 6.1 describes the construction of the scaffold S_W. Based on the scaffold S_W, we define the length of the time phases for different levels of alternation [1]. The deadline $\mathcal{D}$ is set equal to the length of the scaffold (i.e. $\mathcal{D} = |S_W|$). In the construction, the i-th symbol in S_W corresponds to the i-th time slot, $1 \leq i \leq |S_W|$. The level-j time phase for a level-j edge x in $\mathcal{C}_W$ starts at the time slot corresponding to start_x and continues until the time slot corresponding to end_x. The lengths of the time phases, therefore, follows from the positions of start and end symbols in S_W and are defined as follows.

[1] The length of time phases, and the number of unit length tasks will be different in the final construction to be described in the next subsection. In order to explain the basic organization of tasks, we mark these function with a prime (PhaseLen$'$, for example) in this section.

Algorithm 5.8: Construction of the basic scaffold string for normalized $W[t]$-programs

DFT-MAIN-W

Output $l-1$ buffer symbols $\text{buffer}_{\exists_1}$
DFT-W($\text{root}_{\mathcal{C}}$,2)

End DFT-MAIN-W

DFT-W(node x,level j)

foreach *child* y *of* x **do**
Output symbol $\text{start}_{\langle x,y\rangle}$
if x *is at level* t **then**
Output l padding symbols $\text{pad}_{\langle x,y\rangle}$
Output l value symbols $\text{val}_{\langle x,y\rangle}$
else
DFT-W(y,$j+1$)
end
Output symbol $\text{end}_{\langle x,y\rangle}$
end

End DFT-W

$$\text{PhaseLen}'(j) = \begin{cases} 2+2l, & j = t \\ 2 + l \times \text{PhaseLen}'(j+1), & 2 \leq j \leq (t-1) \end{cases}$$

Thus a level-j time phase is large enough to contain l level-$(j+1)$ time phases, one for each of the l computation branches generated from the corresponding nondeterministic step at the j-th alternation level of R_{Q}, where $2 \leq j \leq (t-1)$. A level-j phase includes two additional time slots to schedule two distinguished tasks (Start and End) that indicate the beginning and ending of the scheduling in the level-j time phase. We say that a time slot in a level-t time phase is a padding time slot if the time slot corresponds to one of the pad symbols in the scaffold.

Let $h'(k)$ be the number of $\exists_1$-guesses made by R_{Q}. The number of processors $n_{\text{processor}}$ is defined as follows.

$$n_{\text{processor}} = \begin{cases} 2h'(k) + 3(t-2)/2 + 1, & \text{for even } t \\ 2h'(k) + 3(t-1)/2, & \text{for odd } t \end{cases}$$

The reason for choosing these numbers will become clear once we describe the scheduling techniques.

The Set of Tasks

We now define the set of tasks. The precedences will be defined later. Let $\mathcal{T}$ be the set of unit length tasks in x'. $\mathcal{T}$ consists of several distinguished subsets T_i (the bound on i is a parametric polynomial on $|x|$ and t).

- A *base* set of tasks $T_{\langle \text{base},s \rangle}$ is included in $\mathcal{T}$ for each marker s. The number of such sets is $\sum_{i=2}^{t} l^{\lfloor i/2 \rfloor}$. The number of base tasks for a level-j marker is $\text{PhaseLen}'(j)$, $2 \leq j \leq t$. Two of the tasks in base set $T_{\langle \text{base},s \rangle}$ are distinguished as start_s and end_s to indicate the starting and ending time slots, respectively, for the base tasks.

- Let s_{or} be an *existential* marker for level j, $3 \leq j \leq (t-1)$, and j is odd. A *constraining* set of tasks $T_{\langle \text{constraint},s_{\text{or}} \rangle}$ is included in $\mathcal{T}$ for s_{or}. The purpose of including the constraining tasks is as follows.

  - If s_{or} represents an l-way branching, there will be l designated time phases when the base tasks for s_{or} may start. A scheduling of the tasks $T_{\langle \text{base},s_{\text{or}} \rangle}$ that starts at the i-th of these time phases signifies that the i-th computation branch is taken for s_{or}. The tasks $T_{\langle \text{constraint},s_{\text{or}} \rangle}$ ensure that the base tasks $T_{\langle \text{base},s_{\text{or}} \rangle}$ are scheduled starting from the beginning of one of these time phases.

  The number of constraint tasks for each level-j existential marker is defined as follows.

$$\text{CnstCount}'(j) = \begin{cases} 0, & j = t \\ n_{\text{processor}} - h'(k) - 3(j-1)/2, & 2 \leq j \leq (t-1) \\ & \text{and } j \text{ is odd} \end{cases}$$

- Let s_{and} be a universal level-j marker for some j, $2 \leq j \leq (t-1)$, and j is even. A constraining set of tasks $T_{\langle \text{constraint}, s_{\text{and}} \rangle}$ is included in $\mathcal{T}$ for s_{and}. The tasks in $T_{\langle \text{constraint}, s_{\text{and}} \rangle}$ together with the constraining tasks for the child existential markers of s_{and} ensure that the tasks for the child existential markers are arranged starting from a desired time slot. The number of constraint tasks for a level-j universal marker is defined as follows.

$$\text{CnstCount}'(j) = \begin{cases} 0, & j = t \\ \text{PhaseLen}'(j) - 1, & 2 \leq j \leq (t-1) \text{ and } j \text{ is even} \end{cases}$$

- A set of tasks $T_{\langle \text{test}, s_t \rangle}$ is included for each level-t marker s_t to represent the associated $\exists_1$-tests. If s_t is universal, $T_{\langle \text{test}, s_t \rangle}$ includes a single task. If s_t is existential, the number of tasks in $T_{\langle \text{test}, s_t \rangle}$ equals the number of $\exists_1$-guesses.

- A base set of tasks $T_{\langle \text{base}, s_{\exists_1} \rangle}$ is included for each existential guessing step $s_{\exists_1}$ in the first existential block. We refer to these base sets as the $\exists_1$-*base sets* in order to distinguish them from the base sets for markers. Each $\exists_1$-base set contains $(\mathcal{D} - l + 1)$ tasks. The $\exists_1$-base tasks are scheduled in all level-t time phases and the scheduling is independent of the scheduling of tasks for the markers.

- A set of tasks $T_{\langle \text{invalid}, s_{\exists_1} \rangle}$ is included for each $\exists_1$-step $s_{\exists_1}$. These tasks represent the values that (when stored in the target $\exists_1$-register of $s_{\exists_1}$) would force the computation to continue along the unique rejecting branch for even t and to reject for odd t. We refer to these tasks as $\exists_1$-*invalid tasks*.

  Let, g_j be the j-th $\exists_1$-register, $1 \leq j \leq h'(k)$. Also, let v be a value in $[0 \ldots l]$ that g_j may store and b be a computation branch corresponding to level-t node in the computation tree $\mathcal{C}_W$. The set $T_{\langle \text{invalid}, s_{\exists_1} \rangle}$ includes a task for each value v and each computation branch b of R_{Q} if one of the following conditions holds.

  - t is even and no test is performed on g_j in the checking phase of b.
  - t is even, a test is performed on g_j in branch b and the value v, if stored in g_j, would force the computation to continue along the unique *rejecting* branch.
  - t is odd, a test is performed on g_j in branch b and the value v, if stored in g_j, would force the computation to take a *rejecting* branch.

- Finally, a set of $l-1$ padding tasks are included to represent the $l-1$ buffer$_{\exists_1}$ symbols in the scaffold. We refer to them as $\exists_1$-buffer tasks.

The Precedences of the Tasks

The precedence rules restrict the ways the tasks can be scheduled. We use the following terminology in the construction of the precedence rules.

Definition 5.3 *A sequence of tasks $X = \{x_1, x_2, \ldots, x_m\}$ is said to have a* total order *if given any pair of tasks x_1 and x_2, either $x_1 \prec x_2$ or $x_2 \prec x_1$ holds.*

Observation 5.20 *Let a sequence X of unit length tasks be totally ordered. In any scheduling, at most one task from X can be scheduled in any particular time slot.*

Definition 5.4 *Let $X = \{x_1, \ldots, x_m\}$ be a set of tasks such that the tasks have a total order. A scheduling of the tasks in X is* interval-free *if the tasks are scheduled in $|X|$ consecutive time slots in accordance with the total order.*

Observation 5.21 *Let the tasks in the set $X = \{x_1, \ldots, x_m\}$ have a total order. Let y be another task such that $x_{i-1} \prec y \prec x_{i+1}$, for some i, $1 < i < m$. If X has an interval-free scheduling (i.e. x_{i-1}, x_i, and x_{i+1} are scheduled in consecutive time slots), then y must be scheduled in the same time slot when x_i is scheduled.*

We say that the task y has the *same precedence* as the task x_i with respect to the total order of $x_1, \ldots, x_m$ in case $x_{i-1} \prec y \prec x_{i+1}$, for some i, $1 < i < m$.

We define the precedence rules such that the following holds.

- The $\exists_1$-base tasks in $T_{\langle \text{base}, s_{\exists_1} \rangle}$, for each $\exists_1$-step $s_{\exists_1}$, have a total order. The number of tasks in each $\exists_1$-base set is $|S| - l + 1$. We consider that the i-th task in the ordered sequence corresponds to the $(l + i - 1)$-th character of the scaffold S. Based on this correspondence, we view the ordered sequence of tasks in $T_{\langle \text{base}, s_{\exists_j} \rangle}$ as consisting of l^t *segments* with some padding tasks in between the segments. An $\exists_1$-base task is included in a segment if the task corresponds to some symbol buffer$_x$ or val$_x$ in the scaffold S, where x is some level-t node of $\mathcal{C}$. By construction of the scaffold, each segment consists of $2 + 2l$ $\exists_1$-base tasks that are consecutive in the total order. Any $\exists_1$-base task that is not part of a segment is a padding task. The padding tasks correspond to the start and end symbols in S. In the construction, each segment corresponds to a level-t node in the computation tree $\mathcal{C}$.

- For an $\exists_1$-step $s_{\exists_1}$, let $x_{\langle v,b\rangle} \in T_{\left\langle \text{invalid},s_{\exists_1}\right\rangle}$ be a task representing an invalid value v for the computation branch b. The task $x_{\langle v,b\rangle}$ is assigned the same precedence as the v-th $\exists_1$-base task in segment b of $T_{\left\langle \text{base},s_{\exists_1}\right\rangle}$.
- The tasks in the base set for a marker have a total order.
- For a marker s, the tasks start_s and end_s have the lowest and the highest precedence, respectively, among all the tasks in $T_{\langle \text{base},s\rangle}$.
- For markers s_{child} and s_{parent}, such that s_{parent} is the parent of s_{child}, the precedence of any base task for s_{child} is higher (respectively lower) than the precedence of $\text{start}_{s_{\text{parent}}}$ (respectively $\text{end}_{s_{\text{parent}}}$).
- For sibling markers s_1 and s_2 such that s_1 appears before s_2 in canonical order, end_{s_1} has lower precedence than start_{s_2}.
- The precedence of any constraint task for a *universal* marker s_{and} is higher than the precedence of $\text{start}_{s_{\text{and}}}$ and lower than the precedence of $\text{end}_{s_{\text{and}}}$.
- The *constraint* tasks for a *universal* marker have a total order.
- No constraint task for a universal marker s_{and} has the same precedence as the second base task in $T_{\langle \text{base},s_{\text{and}}\rangle}$.
- All *constraint* tasks for an *existential* marker have the same precedence as the *second* base task (in the total order) for the same existential marker.
- All tasks in $T_{\langle \text{test},s_t\rangle}$, for some level-$t$ marker s_t, have the same precedence as the $(l+2)$-th base task (in the total order) for s_t.
- The $(l-1)$ $\exists_1$-buffer tasks have a total order defined on them. Also, the precedence of any $\exists_1$-buffer task is defined to be lower than the precedence of the first base task for the first level-2 marker.

We now describe the rules for scheduling the tasks in $\mathcal{T}$.

Scheduling Tasks for Existential Steps in the First Nondeterministic Block

Two types of tasks are defined for each $\exists_1$-step - (i) a set of $\exists_1$-base tasks, and (ii) a set of $\exists_1$-invalid tasks. The following rules specify how the tasks are scheduled for a given witness.

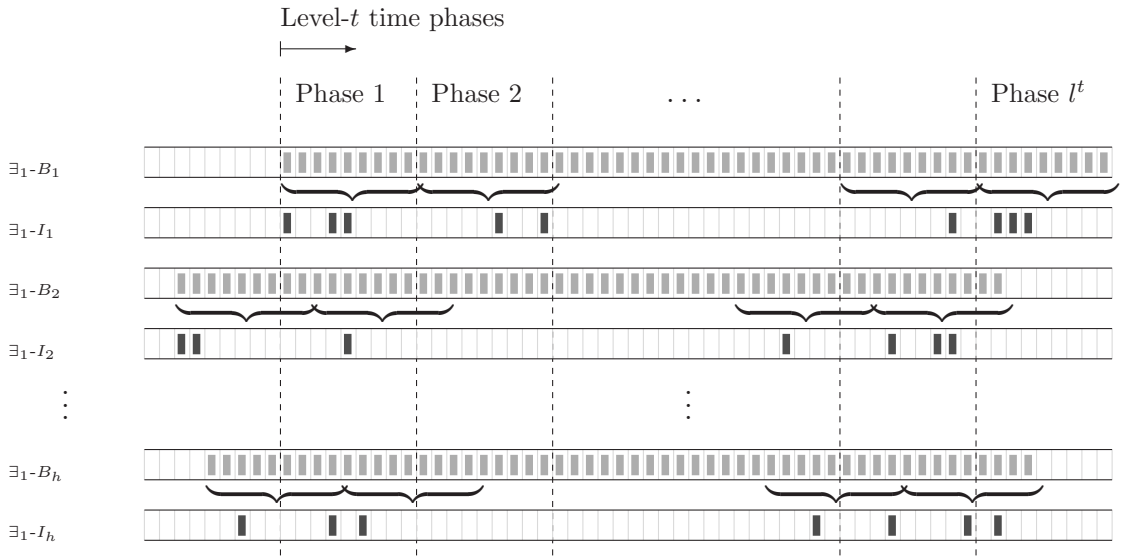


Figure 5.3: Scheduling of tasks for the existential guess steps in the first nondeterministic block in the computation of R_{Q}. The base (respectively invalid) tasks for the j-th $\exists_1$-step are scheduled on the processor marked $\exists_1$-B_j (respectively $\exists_1$-I_j), $1 \leq j \leq h(k)$.

Rule 1: The $\exists_1$-base tasks in $T_{\langle \mathrm{base}, s_{\exists_j} \rangle}$ are scheduled consecutively starting from the $(l - v_j + 1)$-st time slot in order to signify that the j-th $\exists_1$-guess is the value v_j. Note that this arrangement assigns the $(l + v_j)$-th task in each segment of $T_{\langle \mathrm{base}, s_{\exists_j} \rangle}$ to the $(l+2)$-th time slot in the corresponding level-t time phase (Figure 5.3).

Rule 2: Let $x_{\langle v,b \rangle}$ be a task in $T_{\langle \mathrm{invalid}, s_{\exists_1} \rangle}$ and $x_{\langle v,b \rangle}$ corresponds to value v and computation branch b. $x_{\langle v,b \rangle}$ is scheduled in the same time slot where the v-th task in the b-th segment of $T_{\langle \mathrm{base}, s_{\exists_j} \rangle}$ is scheduled. Note that the time slots for the tasks in $T_{\langle \mathrm{invalid}, s_{\exists_1} \rangle}$ become fixed due to the precedence rules and the *interval-free* scheduling of the tasks in $T_{\langle \mathrm{base}, s_{\exists_j} \rangle}$.

Scheduling of Tasks for Nondeterministic Steps in the Checking Phase

We now consider the scheduling of base and constraint tasks for the markers at level j, $2 \leq j \leq t$. Recall that a marker at any *even* level of alternation (i.e. j is even) is *universal* while a marker is *existential* at *odd* levels.

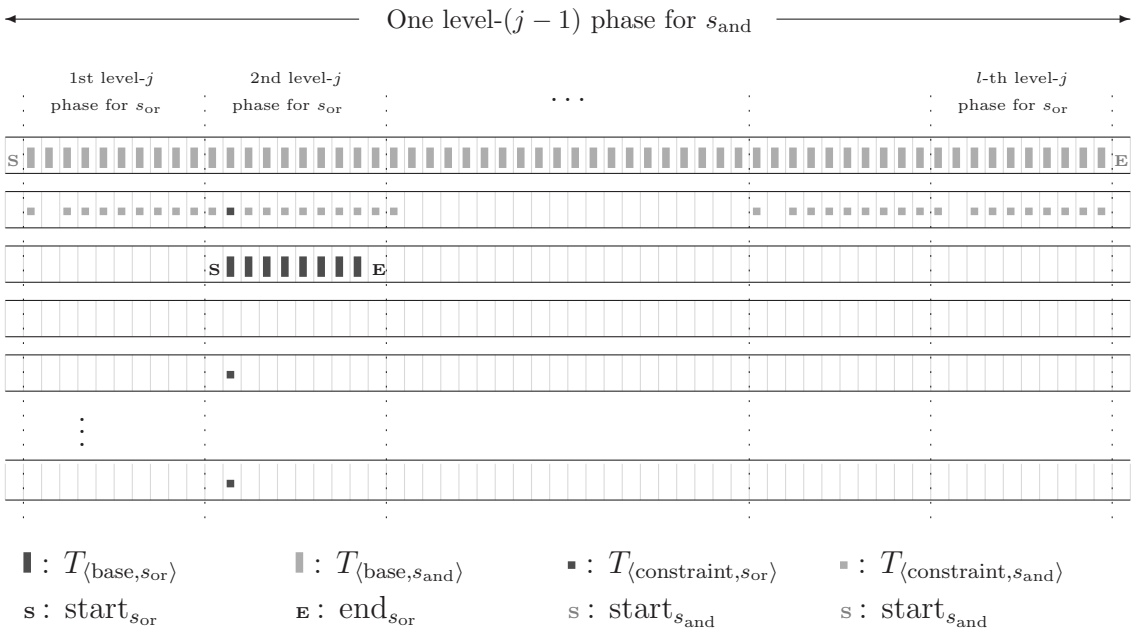


Figure 5.4: Scheduling of base and constraint tasks for a universal marker s_{and} and an existential marker s_{or} where s_{or} is a child of s_{and}.

Rule 3: Let us assume that a level-j marker s_j is placed on a level-j edge x_j in the computation tree. The base tasks for s_j are scheduled in all time slots, one task per slot, in the level-j time phase allocated for x. The scheduling order of the tasks match their order of precedence, with start_{s_j} and end_{s_j} scheduled in the first and last of these time slots, respectively.

Although Rule 3 does not differentiate between universal and existential markers, the distinction in the scheduling of base tasks becomes apparent once we consider them in the context of the structure of the acceptance tree. Let u and e be a universal and an existential node, respectively, in the computation tree. If a branch from u is marked, then all l branches originating from u must be marked whereas exactly one existential branch from e has to be marked in case e is included in the acceptance tree (Definition 5.1). If u is included in the acceptance tree, then base tasks are scheduled in all l time phases associated with u. On the other hand, base tasks are scheduled in exactly one of the l time phases for e in case e is included in the acceptance tree. In particular, the base tasks are scheduled in the i-th of the l time phases to represent the fact that R_{Q} accepts along the i-th existential branch originating from e (i.e. a marker is placed on the i-th edge from e in $\mathcal{C}_W$).

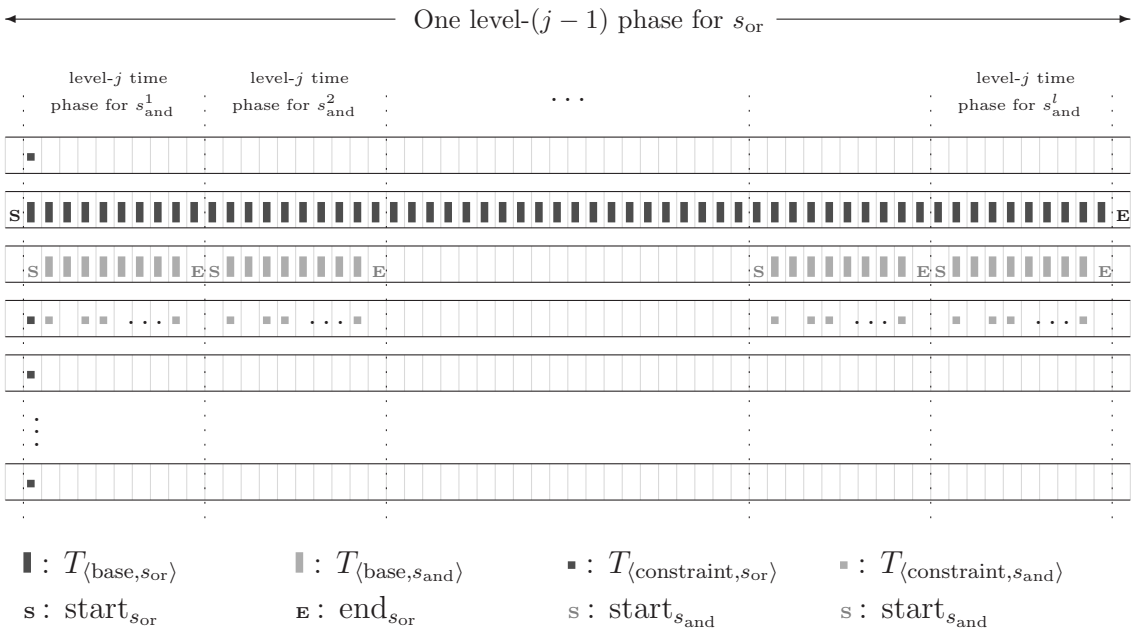


Figure 5.5: Scheduling of base and constraint tasks for an existential marker s_{or} and its l children markers. Here s^i_{and} is the i-th child marker of s_{or}, $1 \leq i \leq l$.

Note that Rule 3, when applied to level-2 universal markers, imply that a base task for some level-2 marker is scheduled in the i-th time slot, for each i, $l \leq i \leq \mathcal{D}$. By the precedence rules, the $(l-1)$ $\exists_1$-buffer tasks must be scheduled in the first $(l-1)$ time slots, the j-th of them being scheduled in the j-th time slot, $1 \leq j \leq l-1$. The scheduling of the $\exists_1$-buffer tasks will be fixed in this way by the other rules in all our constructions. We therefore, do not specify a separate rule for them.

The scheduling of constraint tasks are done differently for universal and existential markers.

Rule 4: Consider a level-j *universal* marker s_{and} and all time slots where a base task for s_{and} has been scheduled. Let $\text{start}_{s_{\text{and}}}$ be scheduled in time slot $\text{TimeStart}_{s_{\text{and}}}$. A constraint task for s_{and} is scheduled in all these time slots except the time slots of the form $(\text{TimeStart}_{s_{\text{and}}} + 2 + i \times \text{PhaseLen}'(j+1))$, $0 \leq i < l$. Thus, no constraint task is scheduled in every $\text{PhaseLen}'(j+1)$-th time slot starting from $(\text{TimeStart}_{s_{\text{and}}} + 2)$ (Figure 5.4).

Rule 5: All constraint tasks for an *existential* marker s_{or} are scheduled in the same time slot where the second base task (in the total order) for s_{or} has been scheduled (Figure 5.5).

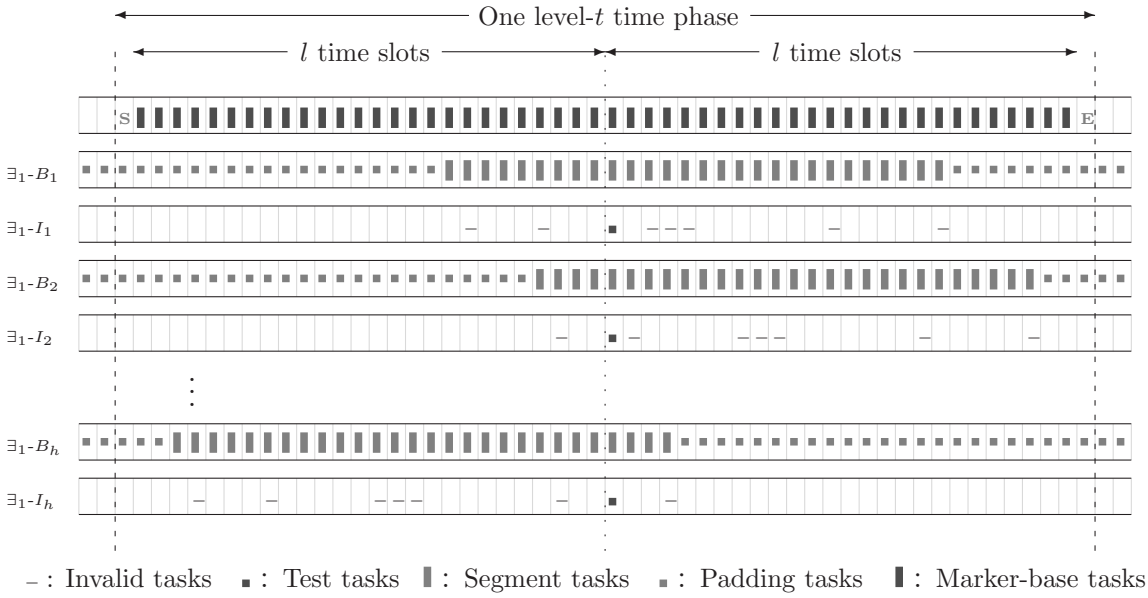


Figure 5.6: Scheduling of tasks in a level-t time phase, for odd t, $t > 1$ (i.e. the level-t marker is an existential marker). The base (respectively invalid) tasks for the j-th $\exists_1$-step are scheduled on the processor marked $\exists_1$-B_j (respectively $\exists_1$-I_j), $1 \leq j \leq h(k)$. All $h(k)$ test tasks are scheduled in the $(l+2)$-th time slot. One test task is scheduled on processor $\exists_1$-I_j, for each j, $1 \leq j \leq h(k)$. The scheduling is possible only if no invalid task is scheduled in the $(l+2)$-th time slot of the level-t time phase under consideration.

Scheduling of Tasks in a Level-t Time Phase

Let us consider a level-t time phase corresponding to edge x in $\mathcal{C}$. Let the base tasks of a level-t marker s_t has been scheduled (as described before) in this time phase. The test tasks associated with s_t remain to be scheduled.

Rule 6: All test tasks in $T_{\langle \text{test}, s_t \rangle}$ are scheduled in the same time slot where the $(l+2)$-th base task (in the total order) for s_t is scheduled (Figure 5.6).

5.2.2.3 Representing Computation of a Normalized $L[t]$-program by a Scheduling

We now extend the previous construction for $W[t]$ to incorporate the computational features of a normalized $L[t]$-program. Let Q' be any problem that can be decided by a normalized $L[t]$-program $R_{\mathrm{Q}'}$. We construct an instance $\langle x', k' \rangle$ of PCMS to represent the computation of $R_{\mathrm{Q}'}$ on a given input $\langle x, k \rangle$. As noted before, the

structure of the acceptance tree $\mathcal{C}_{L,\text{accept}}$ for $R_{Q'}$ differs from $\mathcal{C}_{W,\text{accept}}$ only at the labelling of the leaves. In addition to a TestSet, each leaf of the acceptance tree $\mathcal{C}_{L,\text{accept}}$ represents (existential) selection of values for the nodes in the corresponding assignment graph.

Let us consider the assignment graphs for all nondeterministic checking branches of $R_{Q'}$. For our construction, we are interested in the nodes in the assignment graph that are neither roots nor leaves. We refer to each such node in the assignment graph as an *intermediate node*. Let $g(k)$ be the maximum number of intermediate nodes in any assignment graph. Let us consider the computation of $R_{Q'}$ along some execution path for some nondeterministic checking branch. We view the computation as if $R_{Q'}$

1. existentially guesses up to $g(k)$ values for the intermediate nodes in the assignment graph,
2. verifies that the guessed values for the intermediate nodes are consistent among themselves,
3. and verifies that all necessary tests are satisfied by the nondeterministically guessed values (both the original ones and the newly guessed values).

Since the computation up to the last nondeterministic operation (along any branch) of a normalized $L[t]$-program is similar to that of a normalized $W[t]$-program, the construction for the nondeterministic steps at level i, $2 \leq i \leq (t-1)$ needs no modification (except that each time phase now contains more time slots than before). The construction for all level-t markers needs to be modified in order to incorporate the new computational features described above.

The new scaffold string S_L is generated by DFT-MAIN-L. The scaffold S_L contains $g(k)l^3$ new symbols (denoted as ASSIGN symbols in the algorithm) for each level-t node in the computation tree. The number of time slots in each level-t time phase is increased, accordingly. The lengths of the time phases for levels $j < t$ also increase as a consequence.

$$\text{PhaseLen}'_L(j) = \begin{cases} 2 + 2l + g(k)l^3, & j = t \\ 2 + l \times \text{PhaseLen}'_L(j+1), & 2 \leq j \leq (t-1) \end{cases}$$

Since the number of base tasks is defined in terms of the length of the phases, the number of tasks in each base set increases accordingly. As before, a total

Algorithm 5.9: Construction of the basic scaffold string for normalized $L[t]$-programs

```
DFT-MAIN-L
  Output l − 1 buffer symbols buffer_{∃1}
  DFT-L(root_C, 2)
End DFT-MAIN-L

DFT-L(node x, level j)
  foreach child y of x do
      Output symbol start_⟨x,y⟩
      if x is at level-t then
          Output l padding symbols pad_⟨x,y⟩
          Output l value symbols val_⟨x,y⟩
          foreach p = 1 to g(k) do /* each intermediate node in this assignment graph */
              foreach u = 0 to l − 1 do      /* each value for the left operand of p */
                  foreach v = 0 to l − 1 do  /* each value for the right operand of p */
                      Output l assign symbols assign_⟨⟨x,y⟩,p,u,v⟩
                  end
              end
          end
      else
          DFT-L(y,j + 1)
      end
      Output symbol end_⟨x,y⟩
  end
End DFT-L
```

order is defined on all tasks in a given base set (including the base sets for the new computational features). New tasks are included to represent computational information about the ASSIGN operations. We want the scheduling to represent

E1: the existential selection of values for targets of ASSIGN operations, and

E2: the existential selection of *valid* value-pairs for the operands of ASSIGN operations.

For E1, a base set of tasks is included for each ASSIGN operation in each nondeterministic checking computation branch. We refer to each such base set as an ASSIGN-base set. Functionally the tasks in an ASSIGN-base set are similar to the tasks in an $\exists_1$-base set, both representing existential selection of values. However, the effect of each ASSIGN-base set is confined to a single level-t phase (i.e. to a

single nondeterministic computation branch in the checking phase) as opposed to all level-t time phases.

For E2, we refine the time slots starting from the $(2l + 2)$-th in a level-t time phase, into $g(k)l^2$ ASSIGN-*time phases*. Each ASSIGN-time phase is of length l and corresponds to a particular ASSIGN operation p, and a value-pair $\langle u, v\rangle$ for its operands. For existential selection of a pair of values for the operands of p, we include a set of SELECTOR tasks. These tasks are scheduled in a single block $\langle p, u, v\rangle$ to represent that the value of the target of the p-th ASSIGN operation is computed from values u and v. New invalid tasks, referred to as the ASSIGN-invalid tasks, are included to ensure that the selected value-pair is consistent with the computation. The ASSIGN-invalid tasks are functionally similar to the $\exists_1$-invalid tasks in the previous construction.

The details of the extensions to the construction for $W[t]$ are described below.

1. A segment in an $\exists_1$-base set includes $g(k)l^3$ new tasks. Each of these new tasks corresponds to an $\text{ASSIGN}_{\langle\cdot\rangle}$ symbol in the scaffold. The inclusion of these new tasks makes the length of each segment equal to the number of time slots in a level-t time phase. We view the new tasks in the segment as divided into $g(k)l^2$ blocks, each of length l. Each block corresponds to a combination of values for the two operands of a particular ASSIGN operation. An expanded segment has the following structure.

$$\ldots P \underbrace{B \ldots B}_{\text{Buffer}} \underbrace{V \ldots V}_{\text{value}} \underbrace{A_{\langle 1,0,0\rangle} \ldots A_{\langle 1,0,0\rangle}}_{\text{Block}\langle 1,1,1\rangle} A_{\langle 1,0,1\rangle} \cdots \underbrace{A_{\langle p,u,v\rangle} \ldots A_{\langle p,u,v\rangle}}_{\text{Block}\langle p,u,v\rangle} \cdots \underbrace{A_{\langle g(k),l,l\rangle} \ldots A_{\langle g(k),l,l\rangle}}_{\text{Block}\langle g(k),l,l\rangle} \cdots$$

2. The number of base tasks for a level-j marker is set equal to $\text{PhaseLen}'_L(j)$, for all j, $2 \leq j \leq t$. The base tasks for a level-t marker have a structure similar to a segment in an $\exists_1$-base set.

3. $g(k)$ new base sets of tasks are added for each level-t node in the computation tree. We call them the ASSIGN-base sets. Each ASSIGN-base set corresponds to a particular ASSIGN operation in a nondeterministic checking branch (i.e. an intermediate node in the corresponding assignment graph). Each ASSIGN-base set contains $(\text{PhaseLen}'_L(t) - l - 2)$ tasks having a total order defined on them.

   The precedences of the ASSIGN-base tasks need to be defined so that they are scheduled in a level-t time phase. However, ASSIGN-base tasks are scheduled in each level-t time phase even if no base task for a level-t marker is scheduled

in those time slots. Thus, the precedences of the ASSIGN-base tasks cannot be defined with respect to the base tasks for level-t markers. Instead, we need tasks which are scheduled in fixed time slots in any valid scheduling. Let us consider the set of tasks T_{ref} consisting of the base tasks for all level-2 markers and the $l-1$ buffer$_{\exists_1}$ tasks. The total number of tasks in T_{ref} is $\mathcal{D}$ and the precedence rules imply a total order on all these tasks. In any valid scheduling, each task from T_{ref} is scheduled in a fixed time slot. The tasks in T_{ref}, therefore, can be used to indicate the starting and ending of each time phase. The precedences of the ASSIGN-base tasks for a level-t node s_t are defined with respect to the tasks in T_{ref} that correspond to start$_{s_t}$ and end$_{s_t}$, respectively.

All tasks in an ASSIGN-base set are scheduled consecutively in a single level-t time phase. The time slot when the scheduling starts represents the value computed for the target by the ASSIGN operation.

4. $g(k)l^2(l-1)+2l$ ASSIGN-constraint tasks are included for each level-t marker m. Each ASSIGN-constraint task corresponds to a base task x of m and is assigned the same precedence as x. No constraint task is included for the second base task in each ASSIGN-block of m. Functionally the ASSIGN-constraint tasks are similar to constraint tasks for an universal marker.

5. New *selector* tasks are introduced to represent existential selection of a value-pair (an ASSIGN-block) for the operands of an intermediated node in the assignment graphs. Two types of tasks are included for each intermediate node; (i) $l+1$ SELECTOR-base tasks constituting a SELECTOR-base set, and (ii) $h'(k)+g(k)+1$ SELECTOR-constraint tasks. These tasks are scheduled the same way the base and constraint tasks for an existential marker are scheduled in the time phase for its parent. In this context, tasks of type (i) and (ii) correspond to base and constraint tasks for an existential marker. Together with the ASSIGN-constraint tasks, the SELECTOR-constraint tasks ensure that the SELECTOR-base tasks are scheduled in a single ASSIGN-block in any interval-free scheduling.

6. The number of processors is increased by $2g(k)+2$ in order to allow the scheduling of ASSIGN-base tasks, ASSIGN-invalid tasks, ASSIGN-constraint tasks, and SELECTOR-base tasks.

$$n^L_{\text{processor}} = \begin{cases} 2h'(k)+2g(k)+3(t-2)/2+3, & \text{for even } t \\ 2h'(k)+2g(k)+3(t-1)/2+2, & \text{for odd } t \end{cases}$$

7. The number of constraint tasks for a level-j universal marker is set to be $\text{PhaseLen}'_L(j) - 1$, for all even j, $2 \leq j < t$. The number of constraint tasks for a level-j existential marker is increased to $n^L_{\text{processor}} - h'(k) + 3(j-1)/2$, for all odd j, $3 \leq j < t$. The precedence rules remain the same as before.

8. The number of test tasks for a level-t marker is increased by $g(k) + 1$ for odd t and by 1 for even t.

9. New $\exists_1$-invalid tasks are included to specify how an $\exists_1$-value affect an ASSIGN operation. An invalid task is included for an $\exists_1$-value i and a block $\langle p, u, v\rangle$ if the $\exists_1$-register is the first (respectively second) operand of p and $i \neq u$ (respectively $i \neq v$).

10. A set of ASSIGN-invalid tasks is included for each assign block in each level-t segment of each ASSIGN-base set. Let us consider an intermediate node p in the assignment graph. The inclusion of an ASSIGN-invalid task $x_{\langle p,u,v\rangle,i}$ indicates that the value computed by the p-th ASSIGN operation can not be i if the values of the first and second operands are u and v, respectively. On the other hand, including an invalid task $x_{\langle p',u,v\rangle,i}$, $p' \neq p$, for p, indicates that p is the left (or right) operand of p' and $i \neq u$ (or $i \neq v$). An ASSIGN-invalid task is given the same precedence as the corresponding ASSIGN-base task.

The $\exists_1$-tasks, tasks for all markers, and all test tasks are scheduled as before. Rules 1 to Rule 6 are applied to schedule these tasks. In addition, the following rules are applied to schedule the newly introduced tasks. The scheduling rules are illustrated in Figure 5.7.

Rule 7: ASSIGN-base and ASSIGN-invalid tasks are scheduled in all level-t time phases. The ASSIGN-base tasks are scheduled consecutively starting from the $(l+1-i)$-th time slot in a level-t time phase to represent that the value of the corresponding intermediate node is i. This scheduling assigns the i-th base task in each ASSIGN-block to the first time slot in the corresponding ASSIGN time phase. The interval-free scheduling of the ASSIGN-base tasks fixes the positions of the ASSIGN-invalid tasks.

Rule 8: ASSIGN-constraint tasks are scheduled in a level-t time phase if a marker is placed there. An ASSIGN-constraint task is scheduled in the same time slot where the corresponding base task for the level-t marker has been scheduled.

Rule 9: Let p be an intermediate node in the assignment graph for some computation branch b. Let the witness specifies the values v_p, u and v for p, p's

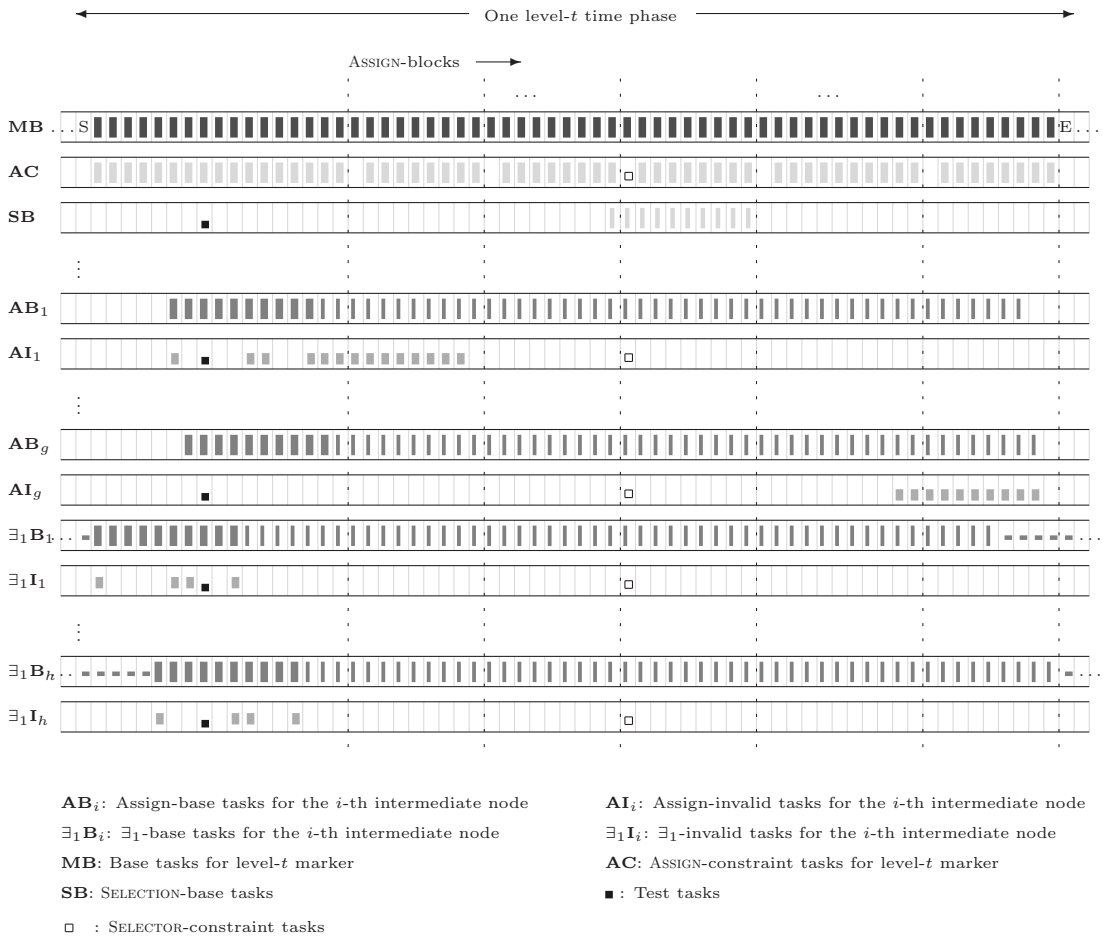


Figure 5.7: A basic scheduling of tasks in a level-t time phase for an $L[t]$-program, $t > 0$.

left operand, and p's right operand, respectively. Except for the first task in the total order, all tasks in a SELECTOR-base set for p and branch b are scheduled in the ASSIGN time phase $\langle p, u, v\rangle$, one task in each time slot. The first SELECTOR-base task in the set is scheduled in the time slot immediately preceding the ASSIGN time phase.

Rule 10: SELECTOR-constraint tasks are scheduled in the time slot where the second task from the corresponding SELECTOR-base set is scheduled.

5.2.2.4 Witness of an Accepting Computation Gives a Valid Scheduling

We next show that a valid scheduling can always be constructed from a given witness by applying the rules. The arguments for the tasks related to the markers and the $\exists_1$-steps apply to both $W[t]$-programs and $L[t]$-programs. The reason is

that (i) the corresponding rules (Rules 1-5) are independent of the actual number of time slots in a time phase, (ii) the difference between the number of processors $n^L_{\text{processor}}$ and the number of constraint tasks for an existential marker remain the same as before, and (iii) all tasks related to the ASSIGN operations are scheduled within a level-t time phase, and hence cannot affect the scheduling of constraint tasks for existential markers. However, the argument for scheduling in level-t time phases for an $L[t]$-program differs from that for a $W[t]$-program.

Lemma 5.22 *Let $T_{\exists_1}$ be the set of all tasks* (base *and* invalid) *related to the $h'(k)$ $\exists_1$-steps. At least $h'(k)$ and at most $2h'(k)$ tasks from $T_{\exists_1}$ are scheduled in the i-th time slot, for each i, $l \leq i \leq (\mathcal{D} - l + 1)$.*

PROOF. The tasks in each of the $h'(k)$ base sets for the $\exists_1$-steps have a total order. Also, a total order is defined on each of the $\exists_1$-invalid sets. Therefore, at most one $\exists_1$-base task and one $\exists_1$-invalid task for each of the $h'(k)$ $\exists_1$-steps can be scheduled at any given time slot. This gives the upper bound of $2h'(k)$. The $(\mathcal{D} - l + 1)$ tasks in each $\exists_1$-base set require $(\mathcal{D} - l + 1)$ consecutive time slots. The first of these $\exists_1$-base tasks can be scheduled in the l-th time slot at the latest while the last can be scheduled at the $(\mathcal{D} - l + 1)$-th time slot at the earliest. Therefore, a base task from each $\exists_1$-base set is scheduled in each time slot in the range l to $(\mathcal{D} - l + 1)$. This gives the lower bound of $h'(k)$. ∎

Lemma 5.23 *Let s_j be a level-j marker and let s_{or} be an existential marker such that s_j is a descendant of s_{or}. If the first base task* start_{s_j} *for s_j is scheduled in time slot* TimeStart_{s_j}*, then the constraint tasks for s_{or} are scheduled in* TimeStart_{s_j} *or before.*

PROOF. Let us consider the time slots when the base tasks for s_{or} are scheduled. Let t_1 and t_2 be the first and second of these time slots. Any base task for a descendant marker of s_{or} is scheduled after $\text{start}_{s_{\text{or}}}$ and hence at time slot t_2 or later (by Rule 3 and the definition of time phases). By Rule 5, all constraint tasks for s_{or} are scheduled in the time slot t_2. ∎

Lemma 5.24 *Let s be any marker and let s_{or} be an existential marker such that s is not an ancestor or a descendant of s_{or}. Let* start_s *and* end_s *be scheduled in time slots* TimeStart_s *and* TimeEnd_s*, respectively. Then the constraint tasks for s_{or} are scheduled before* TimeStart_s *or after* TimeEnd_s*.*

PROOF. Let s^p_{or} and s^p be the ancestors of s_{or} and s, respectively, such that s^p_{or} and s^p are at the same level of alternation and s^p_{or} is not the same as s^p. By definition, the time phases for s^p and s^p_{or} are non-overlapping. The constraint tasks for s_{or} are scheduled in the time phase for s_{or} which is in turn contained in (or the same as) the time phase for s^p_{or}. On the other hand, all base tasks for s are scheduled within the time phase for s which is contained in (or the same as) the time phase for s^p. ∎

Corollary 5.25 *Let s_t be a level-t marker. No constraint task for any existential marker is scheduled in the time slots occupied by the second to penultimate base tasks (in order of precedence) for s_t.*

PROOF. The result follows from Lemma 5.23 and 5.24 and the fact that no constraint task is included for a level-t marker. ∎

The following lemma is stated for $W[t]$-programs only. The corresponding result for the $L[t]$-programs can be obtained simply by replacing $n_{\text{processor}}$ with $n^L_{\text{processor}}$.

Lemma 5.26 *Let s_j be a level-j marker and let the base tasks in $T_{\langle \text{base}, s_j \rangle}$ be scheduled in the time slots from* TimeStart_{s_j} *to* TimeEnd_{s_j}. *The number of base or constraint tasks for any level-j' marker, $2 \leq j' \leq j$ scheduled in time slot i,* $\text{TimeStart}_{s_j} \leq i \leq \text{TimeEnd}_{s_j}$ *is as follows.*

$$n_{\text{occupied}}(i,j) = \begin{cases} n_{\text{processor}} - h'(k) - 1, & j \text{ is odd and } i = \left(\text{TimeStart}_{s_j} + 1\right) \\ 3(j-1)/2, & j \text{ is odd and } i \neq \left(\text{TimeStart}_{s_j} + 1\right) \\ n_{\text{processor}} - h'(k), & j \text{ is even and } i = \text{TimeStart}_{s_j} \\ 3(j-2)/2 + 1, & j \text{ is even and } i = \text{TimeStart}_{s_j} + 1 \\ 3(j-2)/2 + 2, & j \text{ is even and } i > \text{TimeStart}_{s_j} + 1 \end{cases}$$

PROOF. By Rule 3, the base tasks for any given marker are scheduled consecutively. By Rule 4, the constraint tasks for each universal marker are also scheduled consecutively, except at the second time slot in the corresponding time phase. Thus, the following tasks, related to any level-j' marker, $2 \leq j' \leq j$, are scheduled in each time slot in the range under consideration.

- A base task for each ancestor of s_j.
- A constraint task for each universal ancestor of s_j.
- A constraint task for s_j if s_j is a universal marker and the time slot is not the second one in the level-j phase.

The total number of such tasks in each time phase is $3(j-1)/2$ in case j is odd and is $3(j-2)/2+2$ if j is even and i is not the second time slot. If i is the second time slot and s_j is universal, then no constraint task for s_j is scheduled in time slot i.

If j is odd and $i = \text{TimeStart}_{s_j} + 1$ then all constraint tasks for the existential marker s_j are also scheduled in time slot i. The number of such constraint tasks is $n_{\text{processor}} - h'(k) - 3(j-1)/2$. However, no constraint task for level-$(j-1)$ ancestor of s_j is scheduled in time slot i. The total number of tasks scheduled in time slot i in this case is $3(j-1)/2 + n_{\text{processor}} - h'(k) - 3(j-1)/2 - 1 = n_{\text{processor}} - h'(k) - 1$.

If j is even and $i = \text{TimeStart}_{s_j}$, all constraint tasks for the level-$(j-1)$ existential ancestor of s_j are scheduled in time slot i. By the previous argument, $n_{\text{processor}} - h'(k) - 1$ processors are occupied by the tasks from all ancestors of s_j. An additional processor is needed for start_{s_j}. ∎

Corollary 5.27 *Let s_t be a level-t marker and the base tasks in $T_{\langle \text{base}, s_t \rangle}$ are scheduled in the time slots from TimeStart_{s_t} to TimeEnd_{s_t}. The number of base or constraint tasks for any level-j' marker, $2 \leq j' \leq t$ scheduled in time slot i, $\text{TimeStart}_{s_t} + 1 \leq i \leq \text{TimeEnd}_{s_t}$ is as follows.*

$$n_{\text{occupied}}(i,t) = \begin{cases} 3(t-1)/2, & t \text{ is odd} \\ 3(t-2)/2+1, & t \text{ is even} \end{cases}$$

Lemma 5.28 and Theorem 5.29 apply to $W[t]$-programs only. The analogous results for $L[t]$ are given by Lemma 5.31 and 5.32.

Lemma 5.28 *Let the base tasks for a level-t marker s_t be scheduled in a level-t time phase that corresponds to the nondeterministic checking branch b. Let, for odd t, a total of $3(t-1)/2$ (a total of $3(t-2)/2+1$, for even t) base or constraint tasks related to any marker be scheduled in any of these time slots. If the normalized $W[t]$-program R_{Q} accepts in the computation branch b then the test tasks for s_t can be scheduled according to Rule 6.*

PROOF. Since all test tasks in $T_{\langle \text{test},s_t\rangle}$ have the same precedence as the $(l+2)$-th base task in $T_{\langle \text{base},s_t\rangle}$, scheduling according to Rule 6 does not violate the precedence condition for the test tasks. By the hypothesis, the number of base and constraint tasks for the markers scheduled in the same time slot is at most $3(t-1)/2$ for odd t and $3(t-2)/2+1$ for even t. $h'(k)$ additional processors are occupied by a task from each of the $h'(k)$ sets $T_{\langle \text{base},s_{\exists_1}\rangle}$. This leaves $h'(k)$ processors available for scheduling the test tasks and the $\exists_1$-invalid tasks in the time slot under consideration. Let n_{free} be 0 for odd t and $h'(k)-1$ for even t. According to Rule 6, all test tasks in $T_{\langle \text{test},s_t\rangle}$ are scheduled in the same time slot leaving n_{free} processors for scheduling the invalid tasks (if any exists). Thus we must ensure that the number of invalid tasks that are required to be scheduled in this time slot (by the precedence rules) does not exceed n_{free}. We consider the following cases.

(t **is even):** In this case, the last nondeterministic step is universal and $T_{\langle \text{test},s_t\rangle}$ contains a single task. Thus n_{free} equals $(h'(k)-1)$. The outcome of at least one test must cause the computation to branch away from the unique rejecting execution path. Thus the values in at most $(h'(k)-1)$ $\exists_1$-registers may cause the computation to continue along the unique rejecting execution path. In other words, there will be at most $(h'(k)-1)$ invalid tasks to be scheduled in the time slot under consideration.

(t **is odd):** In this case, the last nondeterministic step is existential and there is a unique accepting execution path in the corresponding final checking phase. In the context of scheduling, $T_{\langle \text{test},s_t\rangle}$ contains $h'(k)$ tasks and n_{free} is 0. Since, none of the $\exists_1$-values causes the computation to branch away from the accepting execution path, no invalid task is required to be scheduled by the precedence rule, in the time slot under consideration.

Hence the desired scheduling in the level-t time phase is always possible if the corresponding leaf node in the computation tree of R_{Q} accepts. ∎

Theorem 5.29 *If the normalized $W[t]$-program R_{Q} accepts $\langle x,k\rangle$, then a valid scheduling for $\langle x',k'\rangle$ can be constructed from the witness of an accepting computation of R_{Q}.*

PROOF. Given an acceptance tree for R_{Q} for input $\langle x,k\rangle$ we apply Rules 1 to 6 to construct a scheduling for $\langle x',k'\rangle$. In order to prove that the constructed

scheduling is valid we show that (i) all precedences are satisfied, and (ii) the number of processors needed at any time slot does not exceed $n_{\text{processor}}$.

(Part i): All tasks in any given base set (for a marker or for an $\exists_1$-step) are scheduled according to the associated total order. Thus the *inter-set precedences* of the base tasks are respected. The base tasks for a level-j marker are scheduled in one of the level-j time phases corresponding to the marker. The arrangement of the time phases correspond to the structure of the computation tree. Thus the *intra-set precedences* of the base tasks for all markers are also respected. No additional precedence rule (other than those defining the total order) is associated with the $\exists_1$-base tasks. The precedence of any given constraint task is the same as a specific base task. Rules 4 and 5 schedules the constraint tasks in the same time slot where the corresponding base tasks are scheduled. The same holds for the test tasks (Rule 6). The precedence of an $\exists_1$-invalid task is defined to be the same as the precedence of a unique $\exists_1$-base task both of which are scheduled in the same time slot (Rule 2).

(Part ii): An important fact to note is that the $\exists_1$-invalid tasks are scheduled in level-t time phases only. In other words, no $\exists_1$-invalid task is scheduled in a padding time slot. Thus, the number of processors required for a padding time slot (which is not contained in any level-t time phase) is at most $n_{\text{processor}}$. This follows from Lemma 5.26 and the fact that $h'(k)$ additional $\exists_1$-base tasks for the $h'(k)$ $\exists_1$-steps may be scheduled in those time slots. Since the markers represent an acceptance tree, all level-t markers are placed on accepting level-t nodes in the computation tree. By Corollary 5.27 and Lemma 5.28, the number of processors needed for any time slot in any such level-t time phase is at most $n_{\text{processor}}$. ∎

Theorem 5.29 established the forward direction of the $W[t]$-hardness proof. We now extend the construction for the $L[t]$-programs. Lemma 5.31 and Theorem 5.32 corresponds to the construction for normalized $L[t]$-programs. In what follows, we use the term ASSIGN-tasks to refer to any task related to ASSIGN operations, in general (ASSIGN-base, ASSIGN-invalid, ASSIGN-constraint). Similarly, we use the term SELECTOR-tasks to refer to SELECTOR-base and SELECTOR-constraint tasks in general.

Lemma 5.30 *The number of* ASSIGN*-base and* ASSIGN*-invalid tasks scheduled in any time slot i, $(l+2) \leq i \leq (\text{PhaseLen}_L(t) - l)$ in a level-t time phase is at least $g(k)$ and at most $2g(k)$.*

PROOF. The proof is similar to the proof for the $\exists_1$-base and $\exists_1$-invalid tasks (Lemma 5.22). ∎

Lemma 5.31 *Let the base tasks for a level-t marker s_t be scheduled in a level-t time phase that corresponds to the nondeterministic checking branch b. Let, for odd t, a total of $3(t-1)/2$ (a total of $3(t-2)/2+1$, for even t) base or constraint tasks related to any marker be scheduled in any of these time slots. If the normalized $L[t]$-program $R_{Q'}$ accepts in the computation branch b then the test-tasks,* ASSIGN*-tasks and* SELECTOR*-tasks, for s_t can be scheduled according to Rules 6-10.*

PROOF. Scheduling according to Rules 6-10 respects the precedences of the tasks involved. At most one ASSIGN-constraint task and one SELECTOR-base task are scheduled in each time slot in a level-t time phase (Rules 8 and 9). Together with the ASSIGN-base and ASSIGN-invalid tasks, they occupy at most $2g(k)+2$ processors (Lemma 5.30) in any time slot in a level-t time phase. Scheduling of these tasks is always possible in all time slots where the test tasks and SELECTOR-constraint tasks have not been scheduled (Corollary 5.27). Thus it suffices to show that the scheduling of the tasks in time slots where the test tasks and SELECTOR-constraint tasks are scheduled does not require more than $n^L_{\text{processor}}$.

The argument for scheduling of test tasks are similar to that in the proof of Lemma 5.28. Let t_{l+2} be the $(l+2)$-th time slot in the level-t time phase for b. By Rule 6, all test tasks are scheduled in t_{l+2}. As before, we analyze the cases for odd t and even t separately.

(t is odd): The base and constraint tasks for markers, the $\exists_1$-base tasks, the ASSIGN-base tasks, and the ASSIGN-constraint tasks occupy a total of $h'(k) + g(k) + 3(t-1)/2 + 1$ processors in time slot t_{l+2}. The number of test tasks occupy $h'(k) + g(k) + 1$ additional processors. Since the computation accepts in branch b, the outcomes of all tests on the $\exists_1$-registers and targets of ASSIGN operations cause the computation to continue along the accepting execution path. Thus, no $\exists_1$-invalid or ASSIGN-invalid task needs to be scheduled in t_{l+2}.

(t is even): The number of test tasks in this case is 2. Thus the number of processors occupied by the test tasks, the base and constraint tasks for markers, the $\exists_1$-base tasks, the ASSIGN-base tasks, and the ASSIGN-constraint tasks is $h'(k) + g(k) + 3(t-2)/2 + 4$. As t is even, at least one test outcome along

the unique rejecting branch causes the computation to accept. Thus, no invalid task is scheduled in t_{l+2} for at least one $\exists_1$-step or one of the ASSIGN operations.

It remains to show that the SELECTOR tasks can be scheduled by the specified rules. This part of the argument applies to both even and odd t. Let us consider an intermediate node p in the assignment graph corresponding to the level-t time phase under consideration. Let the witness specify that the value of p is v_p and the values of the operands of p are u and v, respectively. By Rule 7, the ASSIGN-base tasks for p are scheduled consecutively from the v_p-th time slot in this level-t time phase. Similarly, the base tasks ($\exists_1$-base or ASSIGN-base) for the left (respectively right) operand of p are scheduled consecutively starting from the u-th (respectively v-th) time slot in the level-t time phase. Since the values u and v are consistent with the value v_p (by definition of the acceptance tree) no ASSIGN-invalid task for p will be scheduled in the first time slot of the ASSIGN-block $\langle p, u, v\rangle$. Also, by the precedence rules, no invalid task for the operands is scheduled in the first time slot of the ASSIGN-block $\langle p, u, v\rangle$. The SELECTOR-base tasks for p are scheduled in all time slots in ASSIGN-block time phase $\langle p, u, v\rangle$ and the time slot immediately preceding the block (Rule 9). By Rule 10, all SELECTOR-constraint tasks are scheduled in the first time slot of block $\langle p, u, v\rangle$. The scheduling is possible as no invalid task is scheduled in the time slot. The scheduling of all remaining SELECTOR tasks can be done in a similar manner. ∎

Theorem 5.32 *If the normalized $L[t]$-program $R_{Q'}$ accepts $\langle x, k\rangle$, then a valid scheduling for $\langle x', k'\rangle$ can be constructed from the witness of an accepting computation of $R_{Q'}$.*

PROOF. The result can be proved by arguments similar to the proof of Theorem 5.29 and using the results from Lemma 5.31. ∎

5.2.2.5 From a Scheduling to an Accepting Computation

We now extend the basic construction to ensure that existence of a valid scheduling always implies the existence of a witness of an accepting computation. We describe the extension for $L[t]$-programs only.

The arguments presented in the previous section are reversible if the base tasks from any given base set have an interval-free scheduling. This follows from Lemma 5.33 and 5.34.

Lemma 5.33 *1. Let s_{or} be an existential marker at level j, $3 \leq j \leq t$. Let s_i be the i-th child marker of s_{or}. If the base tasks for s_{or} have an interval-free scheduling in a single level-j time phase $\mathcal{P}_j$, then the base tasks for s_i must have an interval-free scheduling in the i-th level-$(j+1)$ time phase in $\mathcal{P}_j$.*

2. Let s_{and} be a universal marker at level $j-1$ and s_{or} be the existential child marker of s_{and}, for some j, $3 \leq j \leq t$. Let the tasks in each $\exists_1$-base set, each Assign*-base set, base set for s_{or}, and base set for each of its ancestors have interval-free scheduling. The tasks in $T_{\langle \text{base}, s_{\text{or}} \rangle}$ must be scheduled in a single level-j time phase.*

Proof. (Part i) The number of base tasks for s_{or} is $2 + l \times \text{PhaseLen}'_L(j+1)$ whereas the number of base tasks for a child marker of s_{or} is $\text{PhaseLen}'_L(j+1)$. The base tasks of all children of s_{or} form a total order such that the first base task for the i-th child has higher precedence than the last base task of the $(i-1)$-th child. All these base tasks have to be scheduled after start_{or} and before $\text{end}_{s_{\text{or}}}$. The result follows from the fact that the number of ordered base tasks for all l children of s_{or} equals the number of time slots available for them.

(Part ii) Since the base tasks for s_{or} have an interval-free scheduling, all constraint tasks for s_{or} must be scheduled in the same time slot where the second base task for s_{or} has been scheduled. We show that the second base task must be scheduled in the second time slot of some level-j time phase under the specified constraints. This is sufficient for our purpose, as the number of base tasks for s_{or} is the same as the length of a level-j time phase.

Let t_2 be the time slot where the second base task (and hence all the constraint tasks) for s_{or} are scheduled. We prove the result by induction.

Base Case: The base case for induction corresponds to level 3. The base and constraint tasks for a level-2 universal marker always have fixed interval-free scheduling in a single level-2 time phase. No level-2 constraint task is scheduled in the second time slot of any level-3 time phase. A total of $n^L_{\text{processor}} - h'(k) - 3$ constraint tasks for s_{or} are scheduled in time slot t_2. Thus, $n^L_{\text{processor}} - 1$ processors are needed at t_2 to schedule any task that is not related to the level-2 ancestor of s_{or}. The number of processors occupied by the tasks related to level-2 universal markers is 1 at the second time slot in each level-3 time phase, and 2 at other time slots. Thus, t_2 must be the second time slot of some (appropriate) level-3 time phase.

Hypothesis: We use the result from part (i), in the induction hypothesis. We assume that the base tasks for the level-j' ancestor of s_{or} are scheduled in a single level-j' time phase, $2 \leq j' < j$.

Induction Step: By induction hypothesis, the base tasks for s_{and} have an interval-free scheduling in a single level-$(j-1)$ time phase. Let $\mathcal{P}_{j-1}$ be the level-$(j-1)$ time phase. Thus, no constraint task for s_{and} is scheduled in the second time slot of any level-j time phase in $\mathcal{P}_{j-1}$.

A total of $n^L_{\text{processor}} - h'(k) - 3(j-1)/2$ constraint tasks for s_{or} are scheduled at time slot t_2. The base and constraint tasks for all ancestors of s_{or} are also scheduled in all time slots in $\mathcal{P}_{j-1}$. The number of such tasks is $3(j-3)/2+1$ for the second time slot in any level-j time phase, and $3(j-3)/2+2$ for the others (follows from the proof of Lemma 5.26). The total number of base tasks for all l child markers of s_{or} is exactly $\text{PhaseLen}'_L(j)-2$ and all of them must be scheduled after $\text{start}_{s_{\text{or}}}$ and before $\text{end}_{s_{\text{or}}}$. One base task for a child marker will be scheduled at t_2. For time slot t_2, a total of $n^L_{\text{processor}} - 3(j-1)/2+2$ processors are needed for scheduling all tasks that are not related to any ancestor of s_{or}. Thus, t_2 must coincide with the second time slot in some level-j time phase in $\mathcal{P}_{j-1}$. ∎

Let $\mathcal{P}$ be a level-t time phase such that the base tasks for some level-t marker have an interval-free scheduling in $\mathcal{P}$. Lemma 5.33 states that, if the base tasks in each base set have an interval-free scheduling, then the base tasks for a given level-j marker must be scheduled in a single level-j time phase. Under these conditions, the number of base and constraint tasks that are scheduled in any time slot of $\mathcal{P}$ is given by Corollary 5.27.

Lemma 5.34 *Let y be a level-t node in $\mathcal{C}$ and let $\mathcal{P}_y$ be the corresponding level-t time phase. $R_{\text{Q}'}$ accepts in branch y if the following conditions hold.*

1. *The base tasks for some level-t marker have an interval-free scheduling in $\mathcal{P}_y$.*
2. *The $\exists_1$-base tasks for each $\exists_1$-step have an interval-free scheduling.*
3. *The* ASSIGN*-base tasks and the* SELECTOR*-base tasks for each intermediate node of the assignment graph for y have an interval-free scheduling.*
4. *The number of base and constraint tasks for any marker that are scheduled in any time slot of $\mathcal{P}_y$ is $3(t-1)/2$, for odd t and $3(t-2)/2+1$, for even t.*

PROOF. Let m be the level-t marker whose base tasks are scheduled in $\mathcal{P}_y$. Since the base tasks for m have an interval-free scheduling, the i-th base task for m is scheduled in the i-th time slot of $\mathcal{P}_y$, $1 \le i \le \text{PhaseLen}'(t)$. Also, by the precedence rules, all test tasks are scheduled in the $(l+2)$-th time slot of $\mathcal{P}_y$. We consider the cases for odd t and even t separately.

t **is odd:** The number of processors occupied in the $(l+2)$-th time slot of $\mathcal{P}_y$ is

$$\underbrace{h'(k)}_{\exists_1\text{-bases}} + \underbrace{h'(k)+g(k)+1}_{\text{test tasks}} + \underbrace{g(k)}_{\text{assign-bases}} + \underbrace{1}_{\text{assign-constraint}} + \underbrace{3(t-1)/2}_{\text{marker-tasks}} .$$

Thus, no invalid task is scheduled in the $(l+2)$-th time slot of $\mathcal{P}_y$ as there is no processor available for them. This fact implies that the outcome of each test along the unique accepting execution path for y causes the computation to continue along the accepting execution path.

t **is even:** The number of processors occupied in the $(l+2)$-th time slot of $\mathcal{P}_y$ is

$$\underbrace{h'(k)}_{\exists_1\text{-bases}} + \underbrace{2}_{\text{test tasks}} + \underbrace{g(k)}_{\text{assign-bases}} + \underbrace{1}_{\text{assign-constraint}} + \underbrace{3(t-2)/2+1}_{\text{marker-tasks}} .$$

Thus, at most $h'(k)+g(k)-1$ invalid tasks can be scheduled at the second time slot of $\mathcal{P}_y$. This fact implies that, in the nondeterministic checking branch corresponding to y, the outcome of at least one test causes the computation to accept.

The consistency among the values for the ASSIGN operations follow from the scheduling of the SELECTOR-tasks. By the hypothesis, all SELECTOR-base tasks for an ASSIGN operation x are scheduled consecutively. Thus, all of the associated SELECTOR-constraint tasks are scheduled in the same time slot where the second of these SELECTOR-base tasks is scheduled. By an argument similar to that for Lemma 5.33 part (ii), the SELECTOR-constraint tasks must be scheduled in the first time slot $t_{1,\langle x,u,v\rangle}$ of ASSIGN time phase $\langle x,u,v\rangle$, for some u, v, $0 \le u, v \le l-1$. The number of processors occupied in $t_{1,\langle x,u,v\rangle}$ (for even t) is as follows.

$$\underbrace{h'(k)}_{\exists_1\text{-bases}} + \underbrace{h'(k)+g(k)+1}_{\text{selector-constraints}} + \underbrace{g(k)}_{\text{assign-bases}} + \underbrace{1}_{\text{selector-bases}} + \underbrace{3(t-2)/2+1}_{\text{marker-tasks}}$$

Hence, no invalid task is scheduled in time slot $t_{1,\langle x,u,v\rangle}$. This ensures that the values represented by the scheduling for the operands of x are consistent with the value represented for x. The argument for odd t is similar. ∎

However, our basic construction does not guarantee that the base tasks of each base set have interval-free scheduling in a given valid scheduling. We, therefore, extend the basic construction to ensure that any valid scheduling represents an

accepting computation of R on $\langle x, k\rangle$. The extension is based on the following idea originally used by Bodlaender and Fellows [2].

Let $X = \{x_1, x_2, \ldots, x_m\}$ be a set of tasks in the basic construction such that the tasks in X have a *total order* defined on them. Also, let a total of D_X time slots be available for scheduling all tasks in X, for some $D_X > |X|$. In the extended characterization we create $(D_X - m + 1)$ copies of the set X, each containing exactly m tasks. Let $X_j = \left\{x_{\langle j,1\rangle}, x_{\langle j,2\rangle}, \ldots, x_{\langle j,m\rangle}\right\}$, $1 \leq j \leq (D - m + 1)$, be the copies of X. The i-th element $x_{\langle j,i\rangle}$ of X_j is interpreted as the j-th copy of the i-th element x_i of the original set X, for each j, $1 \leq j \leq (D_X - m + 1)$. A total of $D_X - m$ sets of padding tasks $P_j = \left\{p_{\langle j,1\rangle}, p_{\langle j,2\rangle}, \ldots, p_{\langle j,D_X-m\rangle}\right\}$, $2 \leq j \leq (D_X - m + 1)$ are also added to the set of tasks. Thus the j-th padding set together with the j-th copy X_j of X contains a total of D_X tasks. The precedence of the padding tasks in a set P_j are defined so that (i) the tasks in P_j have a total order, (ii) the precedence of any task in P_j is higher than that of any task in X_{j-1}, and (iii) the precedence of any task in P_j is lower than the precedence of any task in X_j. The purpose of introducing the padding tasks is to ensure that the first task in X_j is scheduled at least D_X time slots later than the first task of X_{j-1}, $2 \leq j \leq (D_X - m + 1)$. Let X_{extended} be the set of all these tasks, i.e.

$$X_{\text{extended}} = \bigcup_{j=1}^{D_X-m+1} X_j \cup \bigcup_{j=2}^{D_X-m+1} P_j.$$

A time phase $\mathcal{P}$, consisting of D_X time slots, was allocated for scheduling the tasks in X in the basic scheduling. The corresponding time phase $\mathcal{P}_{\text{extended}}$ in the extended scheduling consists of $(D_X - m + 1)D_X$ time slots. Thus a total of $((D_X - m + 1)D_X - (D_X - m))$ tasks from X_{extended} are to be scheduled in $(D_X - m + 1)D_X$ time slots. We view $\mathcal{P}_{\text{extended}}$ as consisting of $(D_X - m + 1)$ copies of the time phase $\mathcal{P}$, each copy consisting of D_X time slots as before.

Lemma 5.35 *There exists at least one copy $\mathcal{P}_j$ of $\mathcal{P}$ such that each of the D_X time slots in $\mathcal{P}_j$ has one task from X_{extended}.*

PROOF. Since the tasks in X_{extended} have a total order, any time slot in $\mathcal{P}_{\text{extended}}$ can have at most one task from X_{extended}. The tasks in X_{extended} occupies $((D_X - m + 1)D_X - (D_X - m))$ time slots leaving $(D_X - m)$ empty time slots in $\mathcal{P}_{\text{extended}}$. Hence, at most $(D_X - m)$ copies of $\mathcal{P}$ can have a time slot not occupied by some task from X_{extended}. ∎

Let $\langle x'', k'' \rangle$ be the modified instance that incorporates the extended scheduling features. The total number of base tasks for l sibling universal markers equals the number of time slots available for them, in case the base tasks for the parent existential marker have an interval-free scheduling. Thus, copying is not needed for universal markers (i.e. at even levels of alternation). On the other hand, the number of base tasks for existential markers (i.e. at odd levels of alternation) is always less than the number of time slots available to schedule them. Thus copying is needed at all odd levels of alternation. Copying is needed for the $\exists_1$-bases, ASSIGN-bases and SELECTOR-bases as well.

For convenience, we use Length_t to denote the number of time slots in a level-t time phase in the new context.

$$\text{Length}_t = \underbrace{[(l-1)g(k)+1]}_{\text{copies for Assign-base}} \times \left(2 + 2l + g(k)l^3 \times \underbrace{\big(g(k)l^3 - l + 1\big)}_{\text{copies for Selector-base}} \right)$$

The phase lengths in the extended scheduling is defined as follows.

$$\text{PhaseLen}_L(j) = \begin{cases} \text{Length}_t, & j = t \\ 2 + l \times \text{PhaseLen}_L(j+1), & 2 \le j \le (t-1), j \text{ is odd} \\ 2 + l((l-1)\text{PhaseLen}_L(j+1)+1)\text{PhaseLen}_L(j+1), & 2 \le j \le (t-1), j \text{ is even} \end{cases}$$

We also define the number of copies required for a level-j time phase.

$$\text{NoOfCopies}_L(j) = \begin{cases} 1, & j \text{ is even} \\ (l-1) \times \text{PhaseLen}_L(j) + 1, & j \text{ is odd} \end{cases}$$

The total number of processors remains the same as before. However, the number of time slots, the number of time phases, and the number of base sets of tasks for markers and $\exists_1$-steps change as a result of the extension.

The extended scaffold $S_{L,\text{extended}}$ is generated by DFT-MAIN-L-EXT. The deadline $\mathcal{D}$ is defined as $|S_{L,\text{extended}}|$. Note that

Algorithm 5.10: Construction of the extended scaffold string $S_{L,\text{extended}}$ for normalized $L[t]$-programs

```
DFT-MAIN-L-EXT
  Output l − 1 padding symbols Pad_{∃1}
  foreach i = 1 to (h'(k) × (l − 1) + 1) do                    /* copying for ∃1-steps */
      DFT-L-EXT(root_C, 2, ⟨i⟩)
  end
End DFT-MAIN-L-EXT

DFT-L-EXT(node x, level j, copy c = ⟨c_1, c_2, ..., c_j⟩)
  foreach child y of x do
      Output symbol start_{⟨⟨x,y⟩,c⟩}
      for i = 1 to NoOfCopies(j) do
          if x is at level-t then
              for r = 1 to (l − 1)g(k) + 1 do                  /* copies for ASSIGN-base tasks */
                  Let c' = ⟨c_1, ..., c_j, i, r⟩
                  Output l padding symbols pad_{⟨⟨x,y⟩,c'⟩}
                  Output l value symbols val_{⟨⟨x,y⟩,c'⟩}
                  for r' = 1 to g(k)l^3 − l + 1 do             /* copies for selector base tasks */
                      Let c'' = ⟨c_1, ..., c_j, i, r, r'⟩
                      for p = 1 to g(k) do
                          /* for each intermediate node in this assignment graph */
                          for u = 0 to l − 1 do
                              /* for each value for the left operand of p */
                              for v = 0 to l − 1 do
                                  /* for each value for the right operand of p */
                                  Output l assign symbols assign_{⟨⟨x,y⟩,c'',p,u,v⟩}
                              end
                          end
                      end
                  end
              end
          else
              DFT-L-EXT(y, j + 1, ⟨c_1, ..., c_j, i⟩)
          end

          if j is odd then
              Output NoOfCopies(j) − 1 padding symbols BasePad_{⟨⟨x,y⟩,⟨c_1,...,c_j,i⟩⟩}
          end
      end
      Output symbol end_{⟨⟨x,y⟩,c⟩}
  end
End DFT-L-EXT
```

$$|S_{L,\text{extended}}| = l - 1 + ((l-1)h'(k) + 1)l \times \text{PhaseLen}(2).$$

The copying at the top level is needed to ensure that the tasks in each of the $h'(k)$ base sets for the $\exists_1$-steps, have an interval free scheduling in at least one copy.

Lemma 5.36 *If $R_{\text{Q}'}$ accepts $\langle x, k\rangle$ then a valid scheduling for $\langle x'', k''\rangle$ can be constructed from any given witness.*

PROOF. Theorem 5.29 states that a basic scheduling can always be constructed for $\langle x', k'\rangle$ from a witness. In the extended scheduling, the basic scheduling may be repeated for all copies with the padding tasks placed in the time slots in-between. This becomes straightforward as the number of padding tasks is exactly equal to the number of corresponding time slots. ∎

Lemma 5.37 *If $\langle x'', k''\rangle$ has a valid scheduling then an accepting computation of $R_{\text{Q}'}$ can be constructed from the scheduling.*

PROOF. The result follows from Lemma 5.33, 5.34 and 5.35, and the fact that the level-2 base tasks always have an interval-free scheduling. ∎

Theorem 5.38 *$R_{\text{Q}'}$ accepts $\langle x, k\rangle$ if and only if $\langle x'', k''\rangle$ has a valid scheduling.*

PROOF. The result follows from Lemma 5.36 and 5.37. ∎

Chapter 6

Finite-State Machines and Classes of Fixed-Parameter Intractable Problems

In this chapter, we analyze the fixed-parameter complexity of two kinds of problems on finite-state machines. We first analyze the Intersection problem which asks for a common input string that is accepted by all finite-state machines in a given collection. We then analyze the Membership problem which asks whether there exists an input string of a certain length that is accepted by a given finite-state machine in a specified number of steps. The definitions of the relevant finite-state machines can be found in the appendix.

6.1 Bounded Intersection Problems

The Intersection problem takes a set of machines as input and asks for a common input string that is accepted by all of them. We analyze the fixed-parameter complexity of the Intersection problem for various machine models including finite automata, pushdown automata, and (multi-tape) Turing machines. We show that certain parameterized versions of the problems are complete or hard for different levels of the L-hierarchy. We consider the $L[2]$-hardness of Bounded Deterministic Pushdown Automata Intersection (defined later) as the most significant among these results. The results for other machine models build upon this $L[2]$-hardness result.

We start with the INTERSECTION problem on deterministic finite-state automata and on I/O-deterministic finite state transducers.

6.1.1 Deterministic Finite Automata

The BOUNDED DFA INTERSECTION problem takes the descriptions of multiple deterministic finite automata (DFA) as input and asks for a string (if there exists any) that is accepted by all the DFA. The problem has different versions depending on the choice of the parameter. The version we are interested in is the following.

Bounded DFA Intersection (BDFAI)

Input: An input alphabet Σ, a set A of m DFA $A = \{A_1, \ldots, A_m\}$ on Σ, a positive integer k. Let Q_i be the set of states of DFA A_i, $1 \leq i \leq m$. Also, let q be $\max\{\, |Q_i| \mid 1 \leq i \leq m \,\}$.
Parameter: k, q.
Question: Is there a string $x \in \Sigma^k$ such that x is accepted by all DFA $A_1, \ldots, A_m$?

BDFAI is known to be $W[2]$-hard and in $W[P]$ [17, 41]. We show that BDFAI is in $W[2]$ and hence is $W[2]$-complete.

Theorem 6.1 BDFAI *is* $W[2]$*-complete.*

PROOF. Since the hardness result is already known [41], proving that BDFAI is in $W[2]$ is sufficient for our purpose. We construct an *extended* $W[2]$-program R_{BDFAI} to decide BDFAI. R_{BDFAI} existentially guesses the common input string of length k, universally selects a DFA A_i, and simulates the computation of A_i on the existentially guessed string. R_{BDFAI} accepts in the checking phase if and only if A_i accepts the existentially guessed input string. The algorithm is essentially the same as the one described by Cesati [17]. We present the details as Algorithm 6.1.

Correctness: By the universal selection of a DFA at Step 3, R_{BDFAI} accepts if and only if the checking phase accepts for all DFA A_i, $1 \leq i \leq m$. By Step 4(b), R_{BDFAI} accepts in the checking phase if and only if the simulation of DFA A_i on the existentially guessed input string ends in acceptance in exactly k steps.

Resource Usage: The nondeterministic guess operations satisfy the Constraints AW3, T1, and TU1 for $t = 2$. The size of the transition table T is $q \log q\, |\Sigma|$

Algorithm 6.1: An extended $W[2]$-program R_{BDFAI} to decide BDFAI.

1. **Preprocessing:** Set aside $O(q \log q \, |\Sigma|)$ space for constructing the transition table of a single DFA later in the computation.

2. **Nondeterministic block 1: Existentially** guess k symbols (representing a string of length k) from the alphabet Σ and store them in guess registers $g_1, \ldots, g_k$, respectively.

3. **Nondeterministic block 2: Universally** guess a DFA A_i from A.

4. (a) Construct the transition table T_i for A_i in the space reserved during the preprocessing phase. Let u, u' be states in Q_i, v be a symbol in Σ. Since $|Q_i| \leq q$, each state can be coded by $\log q$ bits. For each bit position b, $1 \leq b \leq \log q$, the value of $T_i[\langle u, b\rangle, v]$ is set to 1 if A_i, while at state u and reading input symbol v, makes a transition to state u' and b-th bit of the coding of u' is 1. $T_i[\langle u, b\rangle, v]$ is set to 0 otherwise. The construction of T_i can be done in parametric polynomial time and independent of the $\exists_1$-values.

   (b) Simulate A_i on the existentially guessed string of length k. Let the guess register g_{k+1} contain the universally guessed value that refers to DFA A_i. To begin with, R_{BDFAI} assigns the coding of the start state of A_i to the $\forall_2$-register g_{k+1}.

   Repeat the following k times.

   i. Let u be the present state stored in g_{k+1} and the input symbol to be read by A_i is in $\exists_1$-register g_j, for some j, $1 \leq j \leq k$.
   ii. Compute the $\log q$ bits for the next state u'. In order to retrieve bit b, R_{BDFAI} modifies the value in g_{k+1} so that it represents $\langle u, b\rangle$. The b-th bit of u' is obtained by performing a JZERO test on g_{k+1} and g_j.
   iii. Construct the next state u' from the computed bits and assign u' to g_{k+1}.

   If A_i reaches an accepting state after k iterations then *accept* in this branch, and *reject* otherwise.

which is bounded by a polynomial in n. The computation for each entry in T takes $O(1)$ time. Thus the entire transition table can be constructed (Step 4(a)) in polynomial time. Step 4(b) simulates k steps of the selected DFA A_i. Simulation of each step of A_i takes $O(\log q)$ steps. Thus the number of steps in any computation branch is bounded by a parametric polynomial time (Constraint AW1). R_{BDFAI} needs to store indices to the table T, indices to the DFA A_i, and symbols from Σ. Each of these values is at most n and hence Constraint AW2 is also satisfied. Only Step 4(b-ii) uses JZERO tests to retrieve the bits of the next state from T. The number of such tests is $O(k \log q)$. Since both k and q are parameters, Constraint $EW1$ is satisfied. ∎

The proof of Theorem 6.1 shows how the extended characterization of $W[t]$ classes can be utilized to construct natural algorithms to decide parametric problems. The relaxed time bound on the checking phase gives R enough time to construct the transition table after the universal branching. Moreover, direct access to the universally guessed value allows R to simulate the computation of the selected DFA in a natural way.

A straightforward modification of the proof of Theorem 6.1 shows that the INTERSECTION problem on I/O-deterministic finite state transducers (FST) is $W[2]$-complete.

I/O-deterministic FST Intersection

Input: A set of i/o-deterministic finite state transducers $A = \{A_1, \ldots, A_m\}$ such that all of them have common input and output alphabets Σ_i and Σ_o, respectively, a string $s_{in} \in \Sigma_i^+$. Let Q_i be the set of states of FST A_i.
Parameter:

m, $|s_{in}|$, $|\Sigma_i|$ (FST-I-1).

$|s_{in}|$, $|\Sigma_i|$, $q = \max(|Q_i|, 1 \leq i \leq m)$ (FST-I-2) .
Question: Is there a string $s_{out} \in \Sigma_o^{|s_{in}|}$ such that each FST $A_i \in A$ accepts s_{in}/s_{out}?

Wareham has shown that FST-I-1 and FST-I-2 are $W[1]$-hard and $W[2]$-hard, respectively [41]. However, no membership results are known for these problems. We show that FST-I-1 is in $W[1]$ and FST-I-2 is in $W[2]$.

Corollary 6.2 FST-I-2 *is* $W[2]$*-complete.*

PROOF SKETCH. An *extended* $W[2]$-program $R_{\text{FST-I-2}}$ can be constructed to decide FST-I-2 in a similar way as R_{BDFAI} was constructed in Theorem 6.1. The only difference is that $R_{\text{FST-I-2}}$ uses the universal register to store both the current state and the current input symbol in s_{in}. ∎

Corollary 6.3 FST-I-1 *is in* $W[1]$ *and hence is* $W[1]$*-complete.*

PROOF SKETCH. We construct a $W[1]$-program $R_{\text{FST-I-1}}$ to decide FST-I-1. $R_{\text{FST-I-1}}$ works in the same way as $R_{\text{FST-I-2}}$. In the checking phase, $R_{\text{FST-I-1}}$ computes the next state directly as it has complete access to the $\exists_1$-values. Also, instead of universally selecting an FST, $R_{\text{FST-I-1}}$ simulates the FSTs one at a time. Since the number of FSTs is also a parameter, $R_{\text{FST-I-1}}$ can perform the simulation within the time bound. ∎

An extended $W[2]$-program $R_{\text{FST-I-2}}$ can perform $(c \log n + h \log n/\alpha(n))$ $\exists_1$-tests in the checking phase, where $c > 0$ is a constant, α is any unbounded non-decreasing function, and h is any function dependent on the parameters only. Since each nondeterministic computation branch simulates $|s_{in}|$ computation steps of the corresponding FST, $R_{\text{FST-I-2}}$ can perform $(\frac{c \log n + h \log n/\alpha(n)}{|s_{in}|})$ $\exists_1$-tests per simulated step, allowing q to be as large as $f(k)n^{\frac{c+h/\alpha(n)}{|s_{in}|}}$. Similar argument holds for the BDFAI problem, as well. We do not know whether the problems remain in $W[2]$ if we allow q to be unbounded. In the next subsection, we show that the intersection problem on deterministic pushdown automata with unbounded state set is in $L[2]$. Thus, the corresponding problems on DFA and I/O-deterministic FST are in $L[2]$ and are $W[2]$-hard.

6.1.2 Pushdown Automata

We first consider the BOUNDED INTERSECTION problem on *deterministic* pushdown automata (DPDA). Later we extend the results for nondeterministic PDA. The problem can be parameterized in several ways, as follows.

Bounded Deterministic PDA Intersection

Input: A set $A = \{A_1, \ldots, A_m\}$ of deterministic PDA over a common input alphabet Σ, a common stack alphabet Γ, and a positive integer k.
Parameter:
m (BDPDAI-1).
k (BDPDAI-2).
m, k (BDPDAI-3).
m, $|\Sigma|$ (BDPDAI-4).
Question: Is there a string $s \in \Sigma^k$ such that s is accepted by all PDA in A?

The problems BDPDAI-1 and BDPDAI-4 are hard for $L[t]$, $t \geq 1$, and $W[SAT]$. These follow from the corresponding hardness results for the Bounded DFA Intersection problem, parameterized by the number of DFA and the alphabet size (Corollary 5.18). BDPDAI-3 is $W[1]$-complete. The hardness result follows from the $W[1]$-hardness of the corresponding Intersection problem on DFA [41]. The membership result follows from Corollary 6.5 and the fact that $L[1]$ equals $W[1]$. Here, we show that BDPDAI-2 is $L[2]$-complete.

Lemma 6.4 BDPDAI-2 *is in* $L[2]$.

Proof. We construct a basic $L[2]$-program $R_{\text{BDPDAI-2}}$ to decide BDPDAI-2. The program existentially guesses the symbols in the common string s, universally selects a DPDA $A_i \in A$, and finally simulates the computation of A_i on s deterministically. In order to simulate each step of A_i in constant time, $R_{\text{BDPDAI-2}}$ organizes the transition table of A_i in a suitable format. The details are given in Algorithm 6.2.

Correctness: By the universal selection of a DPDA at Step 3, $R_{\text{BDPDAI-2}}$ accepts if and only if the checking phase accepts for all A_i, $1 \leq i \leq m$. The checking phase (Step 4) for A_i accepts if and only if the simulation of A_i on s (guessed existentially in Step 2) ends in acceptance.

Resource Usage: Construction of the lookup tables for all DPDA (Step 1) can be done in polynomial time. The remaining steps (Steps 2-4) can be performed in $O(k)$ time. Thus $R_{\text{BDPDAI-2}}$ satisfies Constraints AW1 and BL1. The nondeterministic operations (Steps 2 and 3) satisfy the Constraints AW3, T1, and TU1 for $t = 2$. $R_{\text{BDPDAI-2}}$ needs to store the indices to the transition tables, the indices to

Algorithm 6.2: An $L[2]$-program $R_{\text{BDPDAI-2}}$ to decide BDPDAI-2.

1. **Preprocessing:** Organize the transition rules of each DPDA into a lookup table so that given a state, the next input symbol and the symbol on top of stack; the next state and the next action of the DPDA (with regard to the stack) can be determined in constant time. Arrange the transition tables of all DPDA in a canonical way.

2. **Nondeterministic block 1: Existentially** guess the k symbols in the string s. Let the guess registers $g_1, \ldots, g_k$ store the symbols, respectively.

3. **Nondeterministic block 2: Universally** select a DPDA $A_i \in A$.

4. Simulate the computation of A_i on s as follows.

   (a) Reserve registers $r_b, \ldots, r_{b+k}$, starting from some appropriate index b, to simulate the stack of A_i.

   (b) Reserve a standard register r_{top} to represent the stack pointer.

   (c) Reserve a standard register r_{in} to represent the input head of A_i.

   (d) Reserve a register r_{cur} to represent the current state. Set r_{cur} to be the initial state of PDA A_i.

   (e) Repeat the following computation at most $|s| = k$ times.

      i. Retrieve the next state and the next action based on the current state r_{cur}, next input symbol $g_{r_{\text{in}}}$, and the symbol $r_{r_{\text{top}}}$ currently on top of the stack. Update r_{cur}, r_{in}, $r_{r_{\text{top}}}$, and r_{top} accordingly. Note that the existentially guessed value $g_{r_{\text{in}}}$ is accessed directly in this computation step.

   (f) Accept in this universal branch if and only if A_i accepts s.

the PDA, the head positions for a selected PDA, and symbols and states of a PDA. As each of these values is bounded by a polynomial, $R_{\text{BDPDAI-2}}$ satisfies Constraint AW2. ∎

Corollary 6.5 BDPDAI-3 *is in* $L[1]$.

PROOF. The membership result for BDPDAI-3 is obtained by replacing the universal selection of a DPDA by a loop that selects the m DPDA one by one for simulation. Since m is a parameter, the number of steps in the checking phase remains within the bound. ∎

Lemma 6.6 BOUNDED DETERMINISTIC PDA INTERSECTION*, parameterized by the length of the common string, is hard for* $L[2]$.

PROOF. We construct a generic reduction $\mathcal{A}$ to prove the hardness result. Let Q be any parametric problem in $L[2]$. By Theorem 2.14, there exists a *normalized* $L[2]$-program R_{Q} to decide Q. Given an input $\langle x, k\rangle$, the reduction $\mathcal{A}$ constructs an instance $\langle x', k'\rangle$ of BDPDAI-2 such that R_{Q} accepts $\langle x, k\rangle$ if and only if $\langle x', k'\rangle$ is in the language of BDPDAI-2. Let h be the function that bounds the number of steps in the checking phase of R_{Q}.

The reduction $\mathcal{A}$ works as follows.

1. $\mathcal{A}$ simulates the preprocessing phase of the computation of R_{Q} on $\langle x, k\rangle$ in parametric polynomial time.

2. Let us assume that R_{Q} makes k existential guesses, each from the range $[0 \ldots (fp-1)]$. $\mathcal{A}$ constructs an alphabet

$$\Sigma = \{\sqcup\} \cup \{\, \sigma_{ij} \mid 1 \leq i \leq k, 0 \leq j \leq (fp-1) \,\}$$

   for $\langle x', k'\rangle$. The symbol σ_{ij} signifies that the ith existential guess is the value j. The symbol $\sqcup$ represents a blank whose significance will be explained later.

3. Let us consider the computation along some universal checking branch nb of R_{Q}. Let b be the unique *rejecting* execution path for nb. The reduction $\mathcal{A}$ constructs a deterministic PDA $D_{nb,b}$ to represent the computation in the nondeterministic branch nb along execution path b. The construction of $D_{nb,b}$ is described later.

4. $\mathcal{A}$ constructs k additional DPDA $D_{\exists,j}, 1 \leq j \leq k$, one for each $\exists_1$-register, to ensure that the string s has the desired structure. We give the description of the DPDA $D_{\exists,j}$ later.

We now describe the construction of the deterministic PDA $D_{nb,b}$ that corresponds the computation of R_{Q} on $\langle x, k\rangle$ along execution path b in universal branch nb. Since, b is a rejecting execution path, $D_{nb,b}$ rejects if and only if all tests (JEQUAL or JZERO) along b cause the computation to continue along b. Recall that the computation along b for a universal branch nb can be represented by an assignment graph $G_{nb,b}$ (Definition 2.21). The reduction $\mathcal{A}$ constructs the DPDA $D_{nb,b}$ so that $D_{nb,b}$ performs a depth-first traversal of the binary forest $F_{nb,b}$ corresponding to $G_{nb,b}$. $D_{nb,b}$ stores the value of the universal guess in its state and replaces each reference to the universal guess register in $F_{nb,b}$ by the corresponding universally guessed value [1] during the computation. $D_{nb,b}$ resolves each reference to an existential register by reading the existentially guessed value from the input string s. $D_{nb,b}$ uses its stack to store information for backtracking. Let g be a node in $F_{nb,b}$. After finishing the traversal of the left subtree of g, $D_{nb,b}$ pushes the computed value $v_{g,\mathrm{left}}$ into the stack and starts the traversal of the right subtree of g. Once the traversal of the right subtree of g is complete, $D_{nb,b}$ pops $v_{g,\mathrm{left}}$ and combines it with the value computed for the right subtree to determine the value at node g. We provide the details of the construction below for completeness.

For each internal node g of $F_{nb,b}$, DPDA $D_{nb,b}$ includes the following states in its state set.

- $\{\langle g, \downarrow\rangle\}$: This state indicates that internal node g has been reached for the first time during the depth-first traversal of $F_{nb,b}$.
- $\{\, \langle g, v_r, \nwarrow\rangle \mid 0 \leq v_r \leq (fp-1) \,\}$: The state $\langle g, v_r, \nwarrow\rangle$ indicates that the depth-first traversal of the right subtree of g has ended and the value computed for the right operand of node g is v_r.
- $\{\, \langle g, v_l, \nearrow\rangle \mid 0 \leq v_l \leq (fp-1) \,\}$: The state $\langle g, v_l, \nearrow\rangle$ indicates that the depth-first traversal of the left subtree of g has ended and the value computed for the left operand of node g is v_l.

For each leaf node g_{leaf} of $F_{nb,b}$, $D_{nb,b}$ includes a single state $\langle g_{\mathrm{leaf}}, \downarrow\rangle$ which indicates that the depth-first traversal has reached g_{leaf} for the first time. In addition, $D_{nb,b}$ includes the following states.

[1] $\mathcal{A}$ knows the values in all universal guess registers.

- q_{accept}: This is the unique *accepting* state of $D_{nb,b}$.
- q_{reject}: This is the unique *rejecting* state of $D_{nb,b}$.

$\mathcal{A}$ constructs the transition function $\delta_{nb,b}$ of $D_{nb,b}$, as follows.

- $\delta_{nb,b}(\langle g, \downarrow\rangle, \sqcup, \epsilon) \rightarrow \langle\langle g_l, \downarrow\rangle, \epsilon\rangle$: Here, g is an internal node of $F_{nb,b}$ and g_l is the left child of g. When $D_{nb,b}$ reaches an internal node g for the first time, it starts a depth-first traversal of the subtree rooted at g_l.
- $\delta_{nb,b}(\langle g, v_l, \nearrow\rangle, \sqcup, \epsilon) \rightarrow \langle\langle g_r, \downarrow\rangle, v_l\rangle$: Here, g is an internal node of $F_{nb,b}$ and g_r is the right child of g. The state $\langle g, v_l, \nearrow\rangle$ indicates that $D_{nb,b}$ has finished the traversal of the subtree rooted at the left child of g and the value computed for the left operand of g is v_l. $D_{nb,b}$ pushes v_l onto the stack and begins the depth-first traversal of g_r.
- $\delta_{nb,b}(\langle g, v_r, \nwarrow\rangle, \sqcup, v_l) \rightarrow \langle\langle g_{\text{parent}}, v_g, \nearrow\rangle, \epsilon\rangle$: Here, both g and g_{parent} are internal nodes of $F_{nb,b}$ and g is the left child of g_{parent}. The state $\langle g, v_r, \nwarrow\rangle$ indicates that the traversal of the subtree rooted at the right child of g has finished and the value computed for the right operand of g is v_r. The symbol v_l on the stack indicates that the value of the left operand of g is v_l (computed before). Let, v_g be the value computed at node g with left and right operands v_l and v_r, respectively. $D_{nb,b}$ backtracks to g_{parent} with v_g as the value of the left operand of g_{parent}.
- $\delta_{nb,b}(\langle g, v_r, \nwarrow\rangle, \sqcup, v_l) \rightarrow \langle\langle g_{\text{parent}}, v_g, \nwarrow\rangle, \epsilon\rangle$: This is similar to the previous scenario. The only difference is that g is the right child of g_{parent}.
- $\delta_{nb,b}(\langle g_{\text{leaf}}, \downarrow\rangle, \sigma_{ij}, \epsilon) \rightarrow \langle\langle g_{\text{parent}}, v_j, \nearrow\rangle, \epsilon\rangle$: Here, g_{leaf} is a leaf node of $F_{nb,b}$ and is the left child of its parent g_{parent}. Let us assume that g_{leaf} corresponds to the ith existential register. The input symbol σ_{ij} indicates that the ith existential guess is the value j. PDA $D_{nb,b}$ backtracks to g_{parent} with v_j as the value of its left operand.
- $\delta_{nb,b}(\langle g_{\text{leaf}}, \downarrow\rangle, \sigma_{ij}, \epsilon) \rightarrow \langle\langle g_{\text{parent}}, v_j, \nwarrow\rangle, \epsilon\rangle$: This is similar to the previous case with the exception that g_{leaf} is the right child of g_{parent}.
- $\delta_{nb,b}(\langle g_{\text{test}}, v_r, \nwarrow\rangle, \sqcup, v_l) \rightarrow \langle\langle g_{\text{next}}, \downarrow\rangle, \epsilon\rangle$: The internal node g_{test} represents a binary test (JEQUAL or JZERO) and the test-outcome for the value-pair $\langle v_r, v_l\rangle$ causes the computation to continue along b. $D_{nb,b}$ starts the traversal of the subtree rooted at g_{next} which immediately follows g_{test} in depth-first traversal order of $F_{nb,b}$.

- $\delta_{nb,b}(\langle g_{\text{test}}, v_r, \nwarrow\rangle, \sqcup, v_l) \rightarrow \langle q_{\text{accept}}, \epsilon\rangle$: Here, the test-outcome for the value-pair $\langle v_r, v_l\rangle$ causes the computation to branch away from the rejecting execution path b. $D_{nb,b}$ moves to the *accept* state q_{accept}.

- $\delta_{nb,b}(\langle g_{\text{last}}, v_r, \nwarrow\rangle, \sqcup, v_l) \rightarrow \langle q_{\text{reject}}, \epsilon\rangle$: Here, g_{last} corresponds to the last binary test along b and the outcome of the test for value-pair $\langle v_r, v_l\rangle$ causes the computation to continue along b and reject eventually. $D_{nb,b}$ moves to the *reject* state q_{reject}.

- $\delta_{nb,b}(q_{\text{reject}}, \sigma \in \Sigma, \epsilon) \rightarrow \langle q_{\text{reject}}, \epsilon\rangle$: Once entered, PDA $D_{nb,b}$ remains in the *reject* state until the entire input is read.

- $\delta_{nb,b}(q_{\text{accept}}, \sigma \in \Sigma, \epsilon) \rightarrow \langle q_{\text{accept}}, \epsilon\rangle$: Once entered, PDA $D_{nb,b}$ remains in the *accept* state until the entire input is read.

In any other situation (for example, $D_{nb,b}$ expects a $\sqcup$ at the input but the input head reads some other symbol), PDA $D_{nb,b}$ moves to the rejecting state q_{reject}.

The constructions of the remaining k PDA $D_{\exists,i}$, $1 \leq i \leq k$ are straightforward. Each $D_{\exists,i}$ scans the entire input s and ensures that the following conditions are satisfied.

- All σ_{ij}, appearing in s, have the same j value.

- A σ_{ij}, for some j, appears exactly where the DPDA $D_{nb,b}$ expects a σ_{ij}, for some j, in s.

Since the size of $F_{nb,b}$ is at most $2^{h(k)}$, PDA $D_{nb,b}$ needs to compute for $O(2^{h(k)})$ steps. The length of the common string s, therefore, is $O(2^{h(k)})$. The number of DPDA is at most $(fp+k)$ while the number of states and the length of the transition function description for each DPDA are bounded by some parametric polynomial. Hence, the reduction $\mathcal{A}$ can construct the instance of BDPDAI-2 in parametric polynomial time.

The construction described above shows that $D_{nb,b}$ accepts if and only if some test-outcome causes the computation in nb to branch away from the rejecting execution path b. Thus, a string s is accepted by all DPDA in D if and only if s represents a valid assignment to the existential registers of R_{Q} (ensured by DPDA $D_{\exists,i}$, $1 \leq i \leq k$) and none of the rejecting execution paths in any universal branch is taken (ensured by DPDA $D_{nb,b}$, for all nb and b). This happens exactly when the $L[2]$-program R_{Q} accepts. ∎

Theorem 6.7 BOUNDED DETERMINISTIC PDA INTERSECTION *is* $L[2]$*-complete.*

PROOF. This follows from Lemma 6.4 and Lemma 6.6. ∎

Note that the proof of Lemma 6.4 can be modified in a straightforward way to show that the BOUNDED MULTI-TAPE DTM INTERSECTION problem, parameterized by the length of the common input string and the maximum number of tapes in a DTM, is in $L[2]$. A multi-tape deterministic Turing machine can simulate the computation on a single-tape deterministic Turing machine or a deterministic PDA in proportional time. Thus the BOUNDED INTERSECTION problem on DPDA, DTMs and multi-tape DTMs (with appropriate parameters) are all $L[2]$-complete. Unfortunately, a precise degree of parameterized intractability is not known for the corresponding problem on DFA. The problem is known to be $W[2]$-hard [41] and in $L[2]$ (Lemma 6.4). Whether the problem is $L[2]$-hard or is in $W[2]$ remains as an interesting open question.

We now analyze the BOUNDED INTERSECTION problem on nondeterministic machines. For nondeterministic machine models, we include the bound on the number of computation steps as part of the definition. The reason is that, unlike the deterministic case, a nondeterministic machine M may have ϵ-transitions. Thus, the bound on the length of the common input string does not necessarily bound the number of computation steps in an accepting computation of M. The BOUNDED INTERSECTION problem on nondeterministic pushdown automata, for any fixed function h, is defined as follows.

h-bounded PDA Intersection

Input: A set $A = \{A_1, \ldots, A_m\}$ of PDA over a common input alphabet Σ, a common stack alphabet Γ and a positive integer k.
Parameter:
m (h-BPDAI-1).
k (h-BPDAI-2).
m, k (h-BPDAI-3).
m, $|\Sigma|$ (h-BPDAI-4).
Question: Is there a string $s \in \Sigma^k$ such that s is accepted by all PDA in A in at most $h(k)$ steps?

The $L[t]$-hardness of the INTERSECTION problem on deterministic PDA implies that h-BPDAI-1 and h-BPDAI-4 are $L[t]$-hard. An important consequence of the proof of Lemma 6.6 is that h-BOUNDED PDA INTERSECTION, parameterized by the length of the common string, is hard for $L[3]$.

Corollary 6.8 h-BOUNDED PDA INTERSECTION, *parameterized by the length of the common input string* (h-BPDAI-2), *is hard for* $L[3]$.

PROOF. We modify the reduction $\mathcal{A}$, constructed in the proof of Lemma 6.6, to show the $L[3]$-hardness of h-BPDAI-2. Let $\mathcal{A}_{h\text{-BPDAI-2}}$ be the new reduction. $\mathcal{A}_{h\text{-BPDAI-2}}$ works exactly like $\mathcal{A}$ until $\mathcal{A}$ is about to construct a deterministic PDA $D_{nb,b}$ for execution path b in universal branch nb. $\mathcal{A}_{h\text{-BPDAI-2}}$ constructs a nondeterministic PDA N_{nb}, instead, so that N_{nb} represents the rest of the computation in the universal branch nb. Let us consider a modified version of $D_{nb,b}$, denoted by $D_{nb,i,b}$ which is constructed as follows. $D_{nb,i,b}$ assumes the existence of an $\exists_3$-register whose value is set to i. The computation of $D_{nb,i,b}$ is similar to $D_{nb,b}$ otherwise. $\mathcal{A}_{h\text{-BPDAI-2}}$ constructs a nondeterministic PDA N_{nb} such that it can simulate any DPDA $D_{nb,i,b}$, $0 \leq i \leq (fp-1)$. N_{nb} nondeterministically selects a $D_{nb,i,b}$ and then simulates $D_{nb,i,b}$ deterministically. The nondeterministic step of N_{nb} corresponds to the $\exists_3$-guess in the universal branch nb. ∎

However, it is not obvious whether h-BPDAI-2 can be decided by an $L[3]$-program. The difficulty arises from the fact that a nondeterministic PDA can make $h(k)$ existential guesses whereas a universal branch in the computation of an $L[3]$-program can make a constant number of existential guesses only. However h-BPDAI-2 can be decided by an $A[3]$-program as such programs can have $h(k)$ $\exists_3$-steps. By similar arguments, one can show that h-BOUNDED INTERSECTION problem on *nondeterministic Turing* machines, with parameter the length of the common string and the maximum number of tapes, is $L[3]$-hard and in $A[3]$.

6.1.3 Remarks

Table 6.1.3 summarizes the fixed-parameter complexity of the INTERSECTION problem on different computation models for different choice of parameters. It is interesting to note the common structure of all the reductions in this subsection. The reduction essentially constructs a machine (FA, PDA, TM) for each nondeterministic branch in the checking phase. The operation (*intersection*, in this case) corresponds to the universal branching in case of $W[2]$ or $L[2]$. The actual machine model differs depending on the nature of computation in the final part of the checking phase. Other generic reductions also follow a similar pattern. For example, in the case of the LCS problem, the reduction constructs one string for each nondeterministic branch. The universal step of the corresponding AW-program translates to the verification that the solution is a subsequence of all the constructed strings.

Finite-state machine	Parameter k, q	Parameter k	Parameter m	Parameter m , Σ
DFA	$W[2]$-complete	$W[2]$-hard in $L[2]$	$L[t]$-hard $t \geq 1$	$L[t]$-hard $t \geq 1$
NFA	$W[2]$-complete		$L[t]$-hard	$L[t]$-hard
I/O-deterministic FST	$W[2]$-complete		$L[t]$-hard	$L[t]$-hard
DPDA		$L[2]$-complete	$L[t]$-hard	$L[t]$-hard
PDA		$L[3]$-hard, in $A[3]$	$L[t]$-hard	$L[t]$-hard
DTM		$L[2]$-complete	$L[t]$-hard	$L[t]$-hard
NTM		$L[3]$-hard in $A[3]$	$L[t]$-hard	$L[t]$-hard

Table 6.1: Parameterized complexity of Bounded Intersection problem for different computation models

In general, the reduction constructs an object for each nondeterministic checking branch and defines an operation that corresponds to the universal step (for $W[2]$ or $L[2]$). The constructed objects may be graphs, matrices and so on.

The results related to automata can be extended to higher levels in two ways. We can either define a new operation that takes care of the additional levels of alternation or we can construct machines that have the additional alternation as part of their computation. The later observation suggests that the higher levels of both the W-hierarchy and the L-hierarchy can be related to parametric problems on alternating finite automata. Note that the reductions cannot be extended to $A[2]$ (or $A[t]$) classes in a naive way. The number of nondeterministic steps in an $A[2]$-program can be $\Omega([fp]^k)$, in general. A fixed-parameter reduction cannot construct $\Omega([fp]^k)$ automata in parametric polynomial time. However, this does not rule out the possibility that the defined operation (DFA intersection for example) on the constructed objects may involve k universal steps.

Another interesting aspect of the hardness results related to intersection problem on finite automata is the size of the state set. In case of DFA and I/O-deterministic FST, the Bounded Intersection problems are $W[2]$-complete as long as the number of states is $O(f(k)n^{c/k+h(k)/\alpha(n)})$. This follows from extending the membership proofs of Theorem 6.1 and Corollary 6.2, where the sizes of the state sets were parameters. We can also view these results as restrictions of the $L[2]$-membership

proofs of the corresponding versions with unbounded state sets. We believe that similar properties exist for other parametric problems, as well. This result is stated formally in Theorem 4.7.

6.2 Bounded Membership Problem on Two-way Machines

Given a finite state machine M, the BOUNDED MEMBERSHIP problem asks for a string that is accepted by M in a bounded number of steps. In this section, we analyze the complexity of the BOUNDED MEMBERSHIP problem on finite state machines having *two-way* input tapes. We define the problem for any fixed function h, as follows.

h-Bounded 2DFA Membership

Input: A two-way DFA M, an input alphabet Σ, an integer k.
Parameter: k.
Question: Is there a string $s \in \Sigma^k$ such that M accepts s in $h(k)$ steps?

h-BOUNDED MEMBERSHIP problem for PDA with two-way input tape and for NTMs, with number of tapes bounded by the parameter, can be defined similarly. We show that all these problems are $W[1]$-complete. In contrast, the corresponding problem for one-way DFA can be solved in deterministic polynomial time. We begin by showing that h-BOUNDED NTM MEMBERSHIP is in $W[1]$. The $W[1]$-membership of the corresponding problems on 2DFA and 2PDA, follow immediately.

Lemma 6.9 *h-BOUNDED NTM MEMBERSHIP is in $W[1]$.*

PROOF. We construct a $W[1]$-program R_{NTM} to show the membership result, as follows.

1. **Preprocessing:**

   Reorganize the transition function of the input NTM M so that the next action of M can be computed in constant time from the current state, current input symbol, and the current symbol on the worktape.

2. **Nondeterministic block 1:**

   Existentially Guess k symbols for the string x, $h(k)$ symbols for the nondeterministic choices in the computation of M.

3. Simulate M on x for $h(k)$ steps with the existentially guessed symbols as the witness (if needed).

∎

In the other direction, proving $W[1]$-hardness of h-BOUNDED 2DFA MEMBERSHIP will suffice as the computation of a 2DFA can be simulated by the other machines in proportional time.

Lemma 6.10 *h-BOUNDED 2DFA MEMBERSHIP (h-B2DFAM) is $W[1]$-hard.*

PROOF. Let Q be a problem in $W[1]$. By Theorem 2.13, there exists a normalized $W[1]$-program R_{Q} to decide Q. We construct a fixed-parameter reduction $\mathcal{A}$ that takes an input $\langle x, k\rangle$ and constructs an instance $\langle x', k'\rangle$ of the h-B2DFAM such that R_{Q} accepts $\langle x, k\rangle$ if and only if $\langle x', k'\rangle$ is in h-B2DFAM. Let M be the constructed 2DFA. M considers the k symbols in the input string as the k existential guesses of R_{Q}, the ith symbol being the ith $\exists_1$-guess of R_{Q}. M stores the values in the standard registers of R_{Q} in its states. During the computation, M verifies that each test along the unique accepting branch of R_{Q} is satisfied by the corresponding symbol pair in the input string x. M scans x back and forth to retrieve the required pair of symbols.

We include a brief description of how M verifies that a test is satisfied. Among others, M includes the following states.

- $q_{\langle \mathrm{jeq-start}, i, j\rangle}$: This state signifies that a JEQUAL test on $\exists_1$-registers i and j needs to be performed.

- $q_{\langle \mathrm{jeq-find}_1, i, j, c\rangle}$: This state signifies that the ith symbol of the input string needs to be read and the input head is currently on the c-th symbol.

- $q_{\langle \mathrm{jeq-restore}_1, i, j, v_i, c\rangle}$: This state signifies that the ith symbol of the input string is v_i, the input head is currently on the c-th symbol, and the head needs to be moved to the first input symbol.

- $q_{\langle \text{jeq}-\text{find}_2,i,j,v_i,c\rangle}$: This state signifies that the ith symbol of the input string is v_i, the jth input symbol needs to be read and the input head is currently on the c-th symbol.

- $q_{\langle \text{jeq}-\text{restore}_2,i,j,v_i,v_j,c\rangle}$: This state signifies that the ith and the jth input symbols are v_i and v_j, respectively, the input head is currently on the c-th symbol, and the head needs to be moved to the first input symbol.

- $q_{\langle \text{jeq}-\text{success},i,j\rangle}$: This state signifies that the JEQUAL test on $\exists_1$-registers i and j is satisfied by the ith and jth symbol in the input string.

Simulation of JEQUAL i j d is performed as follows.

- M starts the simulation while in state $q_{\langle \text{jeq}-\text{start},i,j\rangle}$, with the input head on the first input symbol.

- M goes through the $i-1$ states, $q_{\langle \text{jeq}-\text{find}_1,i,j,c\rangle}$, $1 \leq c \leq i-1$, each time moving the head one position to the right, until it reaches the state $q_{\langle \text{jeq}-\text{find}_1,i,j,i\rangle}$.

- If the ith input symbol is v_i, M moves to state $q_{\langle \text{jeq}-\text{restore}_1,i,j,v_i,i\rangle}$ and starts moving the input head back to the first position.

- M goes through the states $q_{\langle \text{jeq}-\text{restore}_1,i,j,v_i,c\rangle}$, $i \geq c > 1$, each time moving the head one position to the left, until state $q_{\langle \text{jeq}-\text{restore}_1,i,j,v_i,1\rangle}$ is reached.

- M moves to state $q_{\langle \text{jeq}-\text{find}_2,i,j,v_i,c\rangle}$, moves the head to the jth input symbol, read the symbol, and move the head back to the leftmost position, as before. If the jth input symbol is v_j, M moves to state $q_{\langle \text{jeq}-\text{restore}_2,i,j,v_i,v_j,1\rangle}$ at the end of this phase.

If $\langle v_i, v_j\rangle$ falsifies the test, M *rejects*. Otherwise, M starts simulating the next test. M *accepts* if all tests are satisfied. Since the number of tests in the unique accepting branch is at most $h(k)$, the entire computation takes time $h'(k)$, for some function h'. ∎

Corollary 6.11 *h-*BOUNDED TWO-WAY DFA MEMBERSHIP*, h-*BOUNDED TWO-WAY PDA MEMBERSHIP*, and h-*BOUNDED SINGLE-TAPE NTM MEMBERSHIP *are $W[1]$-complete.*

We also analyze the complexity of the following unbounded version of the membership problem, which is defined for some fixed function f and constant c.

(f, c)-Short 2DFA Membership

Input: A two-way DFA M, an input alphabet Σ, an integer k.
Parameter: k.
Question: Is there a string $s \in \Sigma^k$ such that M accepts s in $f(k)n^c$ steps?

Theorem 6.12 Short 2DFA Membership *is $W[SAT]$-hard and in $W[P]$.*

Proof. It suffices to construct an AW-program R_{S2DFAM} for an ARAM to show the membership result (Theorem 1.3). Given the description of a 2DFA as input, and k as the parameter, R_{S2DFAM} existentially guesses k symbols for the input string and then simulates the 2DFA for $f(k)n^c$ steps. Since R_{S2DFAM} runs on an ARAM, it can access all $\exists_1$-values directly. This allows R_{S2DFAM} to perform the simulation in a straightforward way.

In order to show the hardness result, we construct a generic reduction from Weighted Formula SAT [2]. Given a tree-circuit (or a formula) C, we construct a 2DFA M such that C has a weight k satisfying assignment if and only if M accepts some string of length k in $f(k)n^c$ steps.

M considers the k symbols in the input string as the k *true* variables in a truth assignment for C, the ith symbol representing the ith true variable. M performs a depth-first traversal on C. At each leaf, M verifies whether any of the true variables satisfy the literal at the leaf in case the literal is positive. For negative literals, M needs to ensure that none of the true variables falsify the negative literal.

The construction for the depth-first traversal is similar to that in the proof of Lemma 6.6. The construction for the checking at the leaves is similar to that in Theorem 6.10. ∎

[2] No machine characterization for $W[SAT]$ is known.

Chapter 7

Concluding Remarks

In this thesis, we have developed a computational view of parameterized complexity theory. The major contributions of this thesis are (i) a natural machine characterization for the $W[t]$ classes, $t \geq 2$, (ii) a normalized machine characterization for the $W[t]$-classes, $t \geq 1$ and (iii) a normalized machine characterization for the $L[t]$ classes, $t \geq 2$.

The basic results in parameterized complexity theory have been reestablished in the computational framework in a much simpler and more intuitive way. The new proof techniques are similar to those in the theory of *NP*-completeness. For example, the $W[1]$-completeness proof of Weighted Antimonotone 2-CNF SAT resembles the *NP*-completeness proof of SAT (Cook's Theorem). The proof of the Normalization Theorem is constructed as a direct extension to the $W[1]$-completeness proof. In addition, new structural results have also been derived using the computational models. The new results significantly strengthen the previous fundamental results. For example, we have shown that under certain restrictions, WCS on circuits of unbounded size and depth can be decided by $W[2]$-programs.

The natural characterization of $W[t]$ allows us to establish upper bounds by constructing natural algorithms. Nontrivial upper bounds beyond $W[2]$ for natural problems have been rare. With the new characterization, we have been able to show new membership results for levels 2 to 5 of the W-hierarchy. The new upper bounds are significantly stronger than the previous known bounds and in many cases, they match the corresponding lower bounds, giving new completeness results. For example, we have shown that Subset Sum is in $W[3]$ and Maximal Irredundant Set is in $W[4]$ (and in $W^*[3]$), significantly improving the previous upper bound of $W[P]$ in both cases. We have also shown that the Reachability Distance in Vector Addition Systems (Petri Nets) is in $W[5]$. No upper

bound was known previously, for this problem. We have shown that the exact and general versions of Binary Integer Linear Programming are in $W[3]$ and $W[5]$ respectively. These upper bounds are important as many natural problems can be reduced to Subset Sum or Binary Integer Linear Programming (Exact Cheap Tour, and Short Cheap Tour, for example).

On the other hand, the normalized variants of the computational models are useful for proving lower bounds. Proving lower bounds beyond the second level of the W-hierarchy and beyond the first level of the L-hierarchy require the construction of generic reductions. The normalized variants of the computational models provide a uniform starting point for the generic reductions for all levels of the W-hierarchy and the L-hierarchy. Because of the uniformity, generic reductions based on the normalized programs can easily be extended to higher levels. Thus, one may start by establishing lower bounds for classes at the bottom level of the hierarchy and then extend the proof to higher levels to establish stronger lower bounds. The idea has been utilized in the $L[t]$-hardness proofs for LCS and PCMS. Our results related to the $L[t]$-hardness and $L[2]$-completeness are the first such results known for problems other than the defining Model Checking problems. The new $W[SAT]$-hardness result for LCS shows that most of the problems previously known to be $W[t]$-hard, are in fact $W[SAT]$-hard (Corollary 5.18).

Partial simulation of the checking phase of a program is the key ingredient of all generic reductions in this thesis. Unfortunately, the technique does not extend to the $A[t]$ classes. The reason is that an $A[t]$-program may have $\Omega(n^k)$ nondeterministic checking branches and a deterministic algorithm cannot generate all of them in parametric polynomial time. As mentioned in Chapter 7, $A[t]$-hardness results may be derived by defining operations or properties that deal with the parameter-bounded alternation in the checking phase. For example, the $A[2]$-complete problem Clique Dominating Set [19] deals with the $\forall_2$-guess steps by requiring a solution verifier to check that some property holds for all cliques of a certain size. We would like to address the $A[t]$-hardness issue in our future work.

Our categorization results reveal an interesting phenomenon exhibited by many natural problems including Bounded DPDA Intersection (BPDAI) and Weighted Binary Integer Programming -II. The parameterized complexity of these problems vary if the upper bound on certain properties of the problems are varied, keeping the solution space as it is. For example, BPDAI, with no bound on the state set and the size of the stack alphabet, is $L[2]$-complete. If the size of the state set and stack alphabet have upper bounds of the form $f(k)n^{c/k+h(k)/\alpha(n)}$, the problem becomes $W[2]$-complete.

A motivation behind the development of the theory of parameterized complexity

was to refine the classical measure of intractability. With the new computational models, we believe that precise categorization of parametric problems will become much easier. The next logical step would be to investigate whether the intractable problems belonging to the bottom levels of the hierarchies are "easier to solve" than those belonging to higher levels. Can we develop new techniques to deal with problems in, say $W[1]$ or $W[2]$, in parametric polynomial time? Of course, a prerequisite of analyzing such questions is to formalize the notion of "solution" and "easiness".

For the purpose of answering such questions, we have started developing a framework for parametric approximation. Independently, the same framework has been proposed in several recent works [14, 22, 30]. In this framework, the parametric optimization (minimization, in particular) version of a problem Q is defined as follows.

$f(k,n)$-min Q

Input: $\langle x, k\rangle$.
Parameter: k.
Output: Either (i) a solution for Q of cost at most $f(k,n)$, or (ii) a statement that the cost of the minimum solution is strictly greater than k.

Downey and Fellows showed that the Minimum Independent Dominating Set cannot be approximated to any fixed ratio (in the parameterized framework) unless $W[2] = FPT$ [30]. Similar hardness of approximation results with respect to $W[1]$ were shown for other problems including Minimum Turing Machine Acceptance, and Minimum Weighted Circuit SAT for a slightly different notion of parameterized approximation [22]. In the other direction, Cai and Xuang showed that all problems in *MaxSNP* are fixed-parameter tractable in this new framework [14]. We are interested in finding similar positive results. The motivation comes from the fact that the optimization version of the $W[2]$-complete problem Dominating Set and Set Cover can be approximated to an approximation ratio of $\log n$ in deterministic polynomial time. We would like to investigate whether fixed-parameter approximation algorithm with $f(k,n) \leq O(k \log n)$ can be constructed for other problems in $W[2]$, based on the previous approximation results. Fixed-parameter reductions in general are not approximation-preserving. Thus, $W[2]$-membership of a problem Q does not necessarily imply that Q is fixed-parameter approximable to $O(\log n)$ ratio. We would like to characterize fixed-parameter intractable problems (problems in $W[2]$, in particular) based on their fixed-parameter approximation property.

We conclude by mentioning that our computational framework is complementary to the existing characterizations. The original circuit characterization by Downey, Fellows, and other co-researchers laid the foundation for the theory of parameterized complexity. The recent work by Chen, Flum, and Grohe presents a logical view of the framework in terms of Model Checking problems. The computational model allows us to approach the problems from an algorithmic point of view. We believe that all three views (circuit, model-checking, and computational) will co-exist, each providing its own advantages, and all working together for the advancement of the theory of fixed-parameter intractability.

Appendix

Finite State Machines

We briefly present the definitions of different finite state machines that are relevant to the results in this thesis. Details can be found in any standard text such as the book by Hopcroft and Ullman [38] and the article by Wareham [41].

Finite Automaton

A *finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is the input alphabet consisting of a finite number of input symbols,
- $q_0 \in Q$ is the starting state,
- $F \subseteq Q$ is the set of final states,
- δ is a transition function from $Q \times (\Sigma \cup \{\epsilon\})$ to 2^Q (the power set of Q), where ϵ denotes the empty string.

At each step, a finite automaton (FA) reads the next input symbol (or do not read any input symbol at all keeping the input head in the same place) and changes its state as allowed by the transition function. An FA *accepts* an input string if and only if the FA is in one of the final states when the end of input is reached.

An FA as defined above is also known as a *nondeterministic finite automaton* (NFA). A standard finite automaton can read its input once. In other words, the input head of a standard finite automaton is not allowed to move backwards. A

two-way nondeterministic finite automaton (2-NFA) is a nondeterministic finite automaton whose input head can move in both directions. The transition function of a 2-NFA is a mapping from $Q \times \Sigma$ to $2^{Q \times \{\text{left,right}\}}$.

A finite automaton is *deterministic* if the transition function maps $Q \times \Sigma$ to Q. A deterministic finite automaton (DFA) is a two-way deterministic finite automaton (2-DFA) if the input head can move in both directions.

Finite State Transducer

The output from a finite automaton is either *accept* or *reject*. In contrast, a finite state transducer can produce a string as output. Formally, a finite state transducer is a 6-tuple $(Q, \Sigma_{\text{in}}, \Sigma_{\text{out}}\delta, q_0, F)$, where

- Q is a finite set of states,
- Σ_{in} is the input alphabet consisting of a finite number of input symbols,
- Σ_{out} is the output alphabet consisting of a finite number of output symbols,
- $q_0 \in Q$ is the starting state,
- $F \subseteq Q$ is the set of final states,
- δ is a transition function from $Q \times \Sigma_{\text{in}}^{\star}$ to $2^{Q \times \Sigma_{\text{out}}^{\star}}$ (the power set of $Q \times \Sigma_{\text{out}}^{\star}$).

A finite state transducer is *I/O-deterministic* if for each state $q \in Q$, each (sub)string $x_{\text{in}} \in \Sigma_{\text{in}}^{\star}$, and each (sub)string $x_{\text{out}} \in \Sigma_{\text{out}}^{\star}$, there can be at most one state $q' \in Q$ such that (q', x_{out}) is in $\delta\,(q, x_{\text{in}})$.

Pushdown Automata

A pushdown automaton (PDA) is an extension to finite automaton. In addition to all the components of an FA, a PDA includes a stack to store information during the computation. Formally, a pushdown automaton is a 6-tuple $(Q, \Sigma_{\text{in}}, \Sigma_{\text{stack}}\delta, q_0, F)$, where

- Q is a finite set of states,
- Σ_{in} is the input alphabet consisting of a finite number of input symbols,
- Σ_{stack} is the stack alphabet consisting of a finite set of symbols,

- $q_0 \in Q$ is the starting state,
- $F \subseteq Q$ is the set of final states,
- δ is a transition function from $Q \times (\Sigma_{\text{in}} \cup \{\epsilon\}) \times \Sigma_{\text{stack}}$ to $2^{Q \times \Sigma_{\text{stack}}}$ (the power set of $Q \times \Sigma_{\text{stack}}$), where ϵ denotes the empty string.

Similar to an FA, at each step, a PDA reads the next input symbol (or does not read any input symbol at all), changes its state in accordance with the transition function. In addition, a PDA can insert a symbol on the top of stack, or update or delete the symbol currently on top of stack. A PDA is *deterministic* if the transition function δ is a mapping from $Q \times \Sigma_{\text{in}} \times \Sigma_{\text{stack}}$ to $Q \times \Sigma_{\text{stack}}$.

Definitions of Parametric Problems with Associated Bounds

Vertex Cover

Input: A graph $G = (V, E)$, and an integer k.
Parameter: k.
Question: Does G have a vertex cover of size k? A vertex cover for G is a subset S of V such that each edge has at least one of the adjacent vertices in S.

Bounds: In FPT, by S. Buss, reported by J. Buss and J. Goldsmith [9] .

Clique

Input: A graph G, and an integer k.
Parameter: k.
Question: Does G have a clique of size k? A clique for G is a subset S of V such that all vertices in S are pairwise adjacent in G.

Bounds: $W[1]$-complete, Downey and Fellows [26].

Dominating Set (DS)

Input: A graph G, and an integer k.
Parameter: k.
Question: Does G have a dominating set of size k? A dominating set for G is a subset S of V such that each vertex in V is either included in S or is adjacent to some vertex in S.

Bounds: $W[2]$-complete, Downey and Fellows [25].

Longest Common Subsequence (LCS)

Input: An alphabet Σ, a set of strings $S = \{s_1, \ldots, s_m\}$, an integer k.
Parameter:
m (LCS-1).
k (LCS-2).
m, k (LCS-3).
m, $|\Sigma|$ (LCS-4).
Question: Is there a string x of length k such that x is a subsequence of each string in S?

Bounds: (for LCS-2 and LCS-3)

LCS-2: $W[2]$-hard (Bodlaender et al. [3]), in $L[2]$ (Flum and Grohe [37]).

LCS-3: $W[1]$-complete (Bodlaender et al. [3]).

LCS-1 and LCS-4:

Previous lower bound:

$W[t]$-hard for all $t > 0$, by Bodlaender et al. [3].

New lower bound:

$L[t]$-hard for all $t > 0$, $W[SAT]$-hard, Theorem 5.11.

Precedence-Constrained Multiprocessor Scheduling (PCMS)

Input: A set of unit length tasks T, a partial order $\prec$ on the tasks in T, an integer $\mathcal{D}$ specifying the deadline, an integer k specifying the number of processors.
Parameter: k.
Question: Is there a mapping $\mathcal{A} : T \to \{1, \ldots \mathcal{D}\}$ such that for all $t_1, t_2 \in T$, $t_1 \prec t_2 \Rightarrow \mathcal{A}(t_1) < \mathcal{A}(t_2)$, and for all i, $1 \leq i \leq \mathcal{D}$, $|\mathcal{A}^{-1}(i)| \leq k$?

Previous lower bound: $W[2]$-hard, by Bodlaender and Fellows [5].

New lower bound: $L[t]$-hard for all $t > 0$, Theorem 5.19.

Previous upper bound: None.

New upper bound: None.

Bounded DFA Intersection

Input: An input alphabet Σ, a set $A = \{A_1, \ldots, A_m\}$ of m DFA on Σ, a positive integer k. Let Q_i be the state set of DFA A_i, $1 \leq i \leq m$. Also, let $q = \max\{\, |Q_i| \mid 1 \leq i \leq m \,\}$.
Parameter:
m (BDFAI-1).
k (BDFAI-2).
m, k (BDFAI-3).
m, $|\Sigma|$ (BDFAI-4).
k, q (BDFAI-5).
Question: Is there a string $x \in \Sigma^k$ such that x is accepted by all DFA $A_1, \ldots, A_m$?

BDFAI-2:

Lower bound: $W[2]$-hard, by Wareham [41].

Previous upper bound: In $W[P]$, by Cesati [17].

New upper bound: In $L[2]$, follows from Lemma 6.4 .

BDFAI-3:

Bounds: $W[1]$-complete. (Hardness by Wareham [41], membership by Cesati [17]).

BDFAI-1 and BDFAI-4:

Previous lower bound: $W[t]$-hard for all $t > 0$, by Wareham [41].

New lower bound: $L[t]$-hard for all $t > 0$, $W[SAT]$-hard, Corollary 5.18.

BDFAI-5:

Lower bound: $W[2]$-hard, by Wareham [41].

Previous upper bound: In $W[P]$, by Cesati [17].

New upper bound: In $W[2]$ (Theorem 6.1).

I/O-deterministic FST Intersection

Input: A set of i/o-deterministic finite state transducers $A = \{A_1, \ldots, A_m\}$ such that all of them have common input and output alphabets Σ_i and Σ_o respectively, a string $s_{in} \in \Sigma_i^+$. Let Q_i be the state set of FST A_i.
Parameter:

m, s_{in}, $|\Sigma_i|$ (FST-I-1)

$|s_{in}|$, $|\Sigma_i|$, $Q = \max(|Q_i|, 1 \leq i \leq m)$ (FST-I-2)

m, $|\Sigma_i|$, $|\Sigma_o|$ (FST-I-3).
Question: Is there a string $s_{out} \in \Sigma_o^{|s_{in}|}$ such that each FST $A_i \in A$ accepts s_{in}/s_{out}?

FST-I-1:

Lower bound: $W[1]$-hard, by Wareham [41].

Previous upper bound: None.

New upper bound: In $W[1]$, Corollary 6.3.

FST-I-2:

Lower bound: $W[2]$-hard, by Wareham [41].

Previous upper bound: None.

New upper bound: In $W[2]$, Corollary 6.2.

FST-I-3:

Upper bound: None.

Previous lower bound: $W[t]$-hard, for all $t > 0$, by Wareham [41].

New lower bound: $L[t]$-hard, for all $t > 0$, and $W[SAT]$-hard, Corollary 5.18.

I/O Deterministic FST Composition

Input: A set $A = \{A_1, \ldots, A_k\}$ i/o-deterministic finite state transducers having the same input and output alphabet Σ, an order $\{i_1, \ldots, i_k\}$ specifying how these FSTs need to be composed, and a string $u \in \Sigma^+$..
Parameter: k.
Question: Is there a sequence of strings $\{s_1, \ldots, s_k\}$ with $s_0 = u$ and $s_i \in \Sigma^{|u|}$ such that A_{i_j} accepts s_{j-1}/s_j?

Upper bound: None.

Previous lower bound: $W[t]$-hard, for all $t > 0$, by Wareham [41].

New lower bound: $L[t]$-hard, for all $t > 0$, and $W[SAT]$-hard, Corollary 5.18.

Bounded Deterministic PDA Intersection

Input: A set $A = \{A_1, \ldots, A_m\}$ of deterministic PDA having a common input alphabet Σ and a common stack alphabet Γ, and a positive integer k.
Parameter:
m (BDPDAI-1).
k (BDPDAI-2).
m, k (BDPDAI-3).
m, $|\Sigma|$ (BDPDAI-4).
Question: Is there a string $s \in \Sigma^k$ such that s is accepted by all PDA in A?

BDPDAI-2:

Previous lower bound: $W[2]$-hard, follows from $W[2]$-hardness of BDFAI-2 (Wareham [41]).

Previous upper bound: None.

New bounds: $L[2]$-complete, by Lemma 6.6 and 6.4.

BDPDAI-3:

Bounds: $W[1]$-complete. Hardness follows from $W[1]$-hardness of BDFAI-3 (Wareham [41]), membership by Corollary 6.5.

BDPDAI-1 and BDPDAI-4:

Previous lower bound: $W[t]$-hard for all $t > 0$, by Wareham [41].

New lower bound: $L[t]$-hard for all $t > 0$, $W[SAT]$-hard, follows from the hardness results for BDFAI-1 and BDFAI-4, respectively (Corollary 5.18).

h-bounded PDA Intersection

Input: A set $A = \{A_1, \ldots, A_m\}$ of PDA over a common input alphabet Σ, a common stack alphabet Γ and a positive integer k.
Parameter:
m (h-BPDAI-1).
k (h-BPDAI-2).
m, k (h-BPDAI-3).
m, $|\Sigma|$ (h-BPDAI-4).
Question: Is there a string $s \in \Sigma^k$ such that s is accepted by PDA A_i, for all i, in at most $h(k)$ steps?

h-BPDAI-2:

Previous bounds: None.

New lower bound: $L[\mathcal{3}]$-hard, by Corollary 6.8.

New upper bound: In $A[\mathcal{3}]$, follows from Lemma 6.4 and the discussion following Corollary 6.8.

h-BPDAI-3:

Bounds: $W[1]$-complete, by Corollary 6.11.

h-BPDAI-1 and h-BPDAI-4:

Previous lower bound: $W[t]$-hard for all $t > 0$, by Wareham [41].

New lower bound: $L[t]$-hard for all $t > 0$, $W[SAT]$-hard, follows from the hardness results for BDFAI-1 and BDFAI-4, respectively (Corollary 5.18).

h-Bounded Single-tape NTM Membership

Input: A single-tape nondeterministic Turing machine M with input alphabet Σ, an integer k.
Parameter: k.
Question: Is there a string $s \in \Sigma^k$ such that M accepts s in $h(k)$ steps?

Previous bound: None.

New bound: $W[1]$-complete, Corollary 6.11.

h-Bounded Two-way DFA Membership

Input: A two-way DFA M with input alphabet Σ, an integer k.
Parameter: k.
Question: Is there a string $s \in \Sigma^k$ such that M accepts s in $h(k)$ steps?

Previous bound: None.

New bound: $W[1]$-complete, Corollary 6.11.

h-Bounded Two-way PDA Membership

Input: A two-way PDA M with input alphabet Σ, an integer k.
Parameter: k.
Question: Is there a string $s \in \Sigma^k$ such that M accepts s in $h(k)$ steps?

Previous bound: None.

New bound: $W[1]$-complete, Corollary 6.11.

Maximal Irredundant Set

Input: A graph $G = (V, E)$, a positive integer k.
Parameter: k.
Question: Is there a set $V' \subseteq V$ such that (1) each vertex $u \in V'$ has a private neighbour, and (2) V' is not a proper subset of any other set $V'' \subseteq V$ that also has this property?

Previous lower bound: $W[2]$-hard, by Bodlaender and Fluiter [8].

New lower bound: None.

Previous upper bound: in $W[P]$, by Cesati [17].

New upper bound: In $W[4]$ (Theorem 5.3), in $W^*[3]$ (Corollary 5.4).

Reachability Distance for Vector Addition Systems (Petri Nets)

Input: A set $T = (\vec{x_1}, \ldots, \vec{x_m})$ of m vectors, each consisting of n integers, a non-negative starting vector $\vec{s} = (s_1, \ldots, s_n)$, a non-negative target vector $\vec{t} = (t_1, \ldots, t_n)$, a positive integer k.
Parameter: k.
Question: Is there a set of k indices $i_1, \ldots, i_k$ such that $\vec{t} = \vec{s} + \sum_{j=1}^{k} \vec{x}_{i_j}$ and each of the n integer components in each of the k intermediate sums is non-negative?

Previous lower bound: $W[1]$-hard, Downey et al. [29].

New lower bound: None.

Previous upper bound: None.

New upper bound: In $W[5]$, Theorem 5.2.

Short 2DFA Membership

Input: A two-way DFA M, an input alphabet Σ, an integer k.
Parameter: k.
Question: Is there a string $s \in \Sigma^k$ such that M accepts s in parametric polynomial time?

Previous lower bound: None.

New lower bound: $W[SAT]$-hard, Theorem 6.12.

Previous upper bound: None.

New upper bound: In $W[P]$, Theorem 6.12.

Subset Sum

Input: A set of integers $X = \{x_1, \ldots, x_m\}$, an integer s .
Parameter: A positive integer k.
Question: Is there a set $X' \subseteq X$ such that $|X'| = k$ and sum of all the integers in X' is exactly s?

Previous lower bound: $W[1]$-hard, Downey and Fellows [26].

New lower bound: None.

Previous upper bound: In $W[P]$, Fellows and Koblitz [35].

New upper bound: In $W[3]$, Theorem 5.1 .

Weighted Binary Integer Programming

Input: A set of variables $X = \{x_1, \ldots, x_m\}$, a set of linear constraints $C = \{c_1, \ldots, c_l\}$ on the variables in X, an integer k, such that the coefficients in the constraints are binary.
Parameter: k.
Question: Is there a binary solution of weight-k to the system of linear equations?

Bounds: $W[2]$-complete, Downey and Fellows [25, 28].

Weighted Binary Linear Integer Programming - I (BLIP-I)

Input: A set of variables $X = \{x_1, \ldots, x_m\}$, a set of linear constraints $C = \{c_1, \ldots, c_l\}$ on the variables in X, an integer k, such that the coefficients in the constraints are at most $f(k)n^{c/k+h(k)/\alpha(n)}$.
Parameter: k.
Question: Is there a binary solution of weight-k to the system of linear equations?

Previous lower bound: $W[2]$-hardness follows from the hardness of Weighted Binary Integer Programming, Downey and Fellows [25, 28].

New lower bound: None.

Previous upper bound: None.

New upper bound: In $W[2]$, Corollary 5.6.

Weighted Binary Linear Integer Programming - II (BLIP-II)

Input: A set of variables $X = \{x_1, \ldots, x_m\}$, a set of linear constraints $C = \{c_1, \ldots, c_l\}$ on the variables in X, an integer k, such that the coefficients in the constraints are at most $f(k)n^{h(k)}$.
Parameter: k.
Question: Is there a binary solution of weight-k to the system of linear equations?

Previous lower bound: $W[2]$-hardness follows from the hardness of Weighted Binary Integer Programming, Downey and Fellows [25, 28].

New lower bound: None.

Previous upper bound: None.

New upper bound: In $L[2]$, Theorem 5.5.

Weighted Binary Linear Integer Programming - III (BLIP-III)

Input: A set of variables $X = \{x_1, \ldots, x_m\}$, a set of linear constraints $C = \{c_1, \ldots, c_l\}$ on the variables in X.
Parameter: k.
Question: Is there a binary solution of weight-k to the system of linear equations?

Previous lower bound: $W[2]$-hardness follows from the hardness of Weighted Binary Integer Programming, Downey and Fellows [25, 28].

New lower bound: None.

Previous upper bound: None.

New upper bound: In $W[5]$, Theorem 5.8.

(Directed) Colored Cutwidth

Input: A graph $G = (V, E)$, an edge coloring $c : E \to \{1, \ldots, r\}$, a positive integer k.
Parameter: k.
Question: Is there a 1:1 linear layout $f : V \to \{1, \ldots, |V|\}$ such that for each color $j \in \{1, \ldots, k\}$ and for each i, $1 \leq i \leq |V| - 1$, we have $|\{uv : c(uv) = j \text{ and } f(u) \leq i \text{ and } f(v) \geq i + 1\}| \leq r$?

Previous lower bound: $W[t]$-hard, for all $t > 0$, by Bodlaender et al. [6, 7].

New lower bound: $L[t]$-hard, for all $t > 0$, and $W[SAT]$-hard, Corollary 5.18.

Domino Treewidth

Input: A graph $G = (V, E)$, a positive integer k.
Parameter: k.
Question: Is the domino treewidth of G at most k?

Previous lower bound: $W[t]$-hard, for all $t > 0$, by Bodlaender et al. [4].

New lower bound: $L[t]$-hard, for all $t > 0$, and $W[SAT]$-hard, Corollary 5.18.

Feasible Register Assignment

Input: A directed acyclic graph $G = (V, E)$, a positive integer k, a register assignment $r : V \to \{R_1, \ldots, R_k\}$.
Parameter: k.
Question: Is there a linear ordering f of G, and a sequence $S_0, S_1, \ldots S_{|V|}$ of subsets of V, such that $S_0 = \emptyset$, $S_{|V|}$ contains all vertices of in-degree 0 in G, and for all i, $1 \leq i \leq |V|$, $f^{-1}(i) \in S_i$, $S_i \setminus \{f^{-1}(i)\} \subseteq S_{i-1}$ and S_{i-1} contains all vertices u for which $(f^{-1}(i), u) \in E$, and for all j, $1 \leq j \leq k$, there is at most one vertex $u \in S_i$ with $r(u) = R_j$?

Previous lower bound: $W[t]$-hard, for all $t > 0$, by Bodlaender et al. [4].

New lower bound: $L[t]$-hard, for all $t > 0$, and $W[SAT]$-hard, Corollary 5.18.

Module Allocation on Graphs of Bounded Treewidth

Input: A set of modules $M = \{1, \ldots, m\}$, a set of processors $P = \{1, \ldots, p\}$, a cost function $e : (M \times P) \to \mathbb{R}$ which specifies the cost of executing a module on a processor, a communication cost function $C : (M \times P \times M \times P) \to \mathbb{R}$ such that $C(x, p, x', p')$ represents the communication cost when modules x and x' are assigned to processors p and p' respectively, a communication graph $G = (M, E)$, and a positive real number l.
Parameter: treewidth$(G) = k$.
Question: Does there exist an assignment of modules to processors such that the total cost of execution is less than or equal to l?

Previous lower bound: $W[t]$-hard, for all $t > 0$, by Bodlaender et al. [4].

New lower bound: $L[t]$-hard, for all $t > 0$, and $W[SAT]$-hard, Corollary 5.18.

Intervalizing Colored Graphs or DNA Physical Mapping

Input: A graph $G = (V, E)$ and a coloring $c : V \to \{1, \ldots, k\}$.
Parameter: k.
Question: Does there exist a supergraph $G' = (V, E')$ of G which is properly colored by c and which is an interval graph?

Previous lower bound: $W[t]$-hard, for all $t > 0$, by Bodlaender et al. [7].

New lower bound: $L[t]$-hard, for all $t > 0$, and $W[SAT]$-hard, Corollary 5.18.

Triangulating Colored Graphs or Perfect Phylogeny

Input: Graph $G = (V, E)$, a coloring $c : V \to \{1, \ldots, k\}$.
Parameter: k.
Question: Does there exist a supergraph $G' = (V, E')$ of G which is properly colored by c and which is triangulated?

Previous lower bound: $W[t]$-hard, for all $t > 0$, by Bodlaender et al. [7].

New lower bound: $L[t]$-hard, for all $t > 0$, and $W[SAT]$-hard, Corollary 5.18.

Bibliography

[1] K.A. Abrahamson, R.G. Downey, and M.R. Fellows,"Fixed-parameter Tractability and Completeness IV: On Completeness of $W[P]$ and *PSPACE* Analogs," *Annals of Pure and Applied Logic* 73 (1995), 235-276.

[2] H.L. Bodlaender, R.G. Downey, M.R. Fellows, M. T. Hallett, and H. T. Wareham,"Parameterized complexity analysis in computational biology," *CABIOS Computer Applications in the Biosciences* 11, 1 (1994), 49 - 57.

[3] H.L. Bodlaender, R.G. Downey, M.R. Fellows, and H. T. Wareham,"The parameterized complexity of sequence alignment and consensus," *Theoretical Computer Science* 147, 1-2 (1995), 31 - 54.

[4] H.L. Bodlaender and J. Engelfriet,"Domino Treewidth," Technical Report UU-CS-1994-11, Department of Computer Science, Utrecht University, Utrecht, the Netherlands.

[5] H.L. Bodlaender and M.R. Fellows," W[2]-hardness of precedence constrained K-processor scheduling," *Operations Research Letters* 18, 2 (1995), 93-97.

[6] H.L. Bodlaender, M.R. Fellows, and M.T. Hallett, "Beyond *NP*-completeness for Problems of Bounded Width: Hardness for the W-hierarchy (extended abstract)", STOC 26, 1994, 449–458.

[7] H.L. Bodlaender, M.R. Fellows, M.T. Hallett, H.T. Wareham, and T.J. Warnow, "The Hardness of Perfect Phylogeny, Feasible Register Assignment and Other Problems on Thin Colored Graphs", *Theoretical Computer Science* 244, 1-2 (2000), 167 - 188.

[8] H.L. Bodlaender and B. de Fluiter, "Intervalizing k-colored graphs", *Proceedings of the 22th International Colloquium on Automata, Languages, and Programming (ICALP95)*, *Lecture Notes in Computer Science*, 944, 1995, 8798.

[9] J. Buss and J. Goldsmith, "Nondeterminism within P," *SIAM Journal on Computing* 22, (1993) 560–572.

[10] J. Buss and T. Islam, "Simplifying the Weft Hierarchy," *Theoretical Computer Science* 351, (2006) 303–313.

[11] J. Buss and T. Islam, "Algorithms in the Weft Hierarchy," *Theory of Computing Systems*, 2006.

[12] L. Cai, J. Chen, R.G. Downey, and M.R. Fellows, "On the Structure of Parameterized Problems in *NP*", *Information and Computation* 123, (1995), 38–49.

[13] L. Cai, J. Chen, R.G. Downey, and M.R. Fellows, "The Parameterized Complexity of Short Computation and Factorization," *Proceedings of the Sacks Conference* 1993, in Archive for Math Logic, 1997.

[14] L. Cai and X. Huang, "Fixed-parameter Approximation: Conceptual Framework and Approximability Results," in *Proceedings of the International Workshop on Parameterized and Exact Computation (IWPEC 2006)*, *Lecture Notes in Computer Science* 4169, 2006, 96–108.

[15] L. Cai and D. Juedes, "On the Existence of Subexponential Parameterized Algorithms", *Journal of Computer and System Sciences* 67 (2003), 789–807.

[16] E. Cardoza, R. Lipton, A.R. Meyer,"Exponential space complete problems for Petri nets and commutative semigroups (Preliminary Report)", Proceedings of the eighth annual ACM symposium on Theory of computing, 1976, 50–54.

[17] M. Cesati, "The Turing Way to Parameterized Complexity," *Journal of Computer and System Sciences* 67, 4 (2003) 654–685.

[18] A.K. Chandra, D. Kozen, and L.J. Stockmeyer, "Alternation", *Journal of the ACM* 28, 1 (1981), 114–133.

[19] Y. Chen and J. Flum, "Machine Characterization of the Classes of the W-hierarchy," in *Computer Science Logic: CSL 2003*, Lecture Notes in Computer Science, Springer, 2003, 114–127.

[20] Y. Chen, J. Flum and M. Grohe, "Bounded Nondeterminism and Alternation in Parameterized Complexity Theory," in *18th Ann. IEEE Conf. Computational Complexity*, 2003, 18–29.

[21] Y. Chen, J. Flum and M. Grohe, "Machine-based Methods in Parameterized Complexity Theory," *Theoretical Computer Science* 339, 2 (2005), 167–199.

[22] Y. Chen, M. Grohe, and M. Grüber, "On Parameterized Approximability," in *Proceedings of the International Workshop on Parameterized and Exact Computation (IWPEC 2006)*, *Lecture Notes in Computer Science* 4169, 2006, 109–120.

[23] J. Chen and F. Zhang, "On Product Covering in 3-tier Supply Chain Models: Natural Complete Problems for $W[3]$ and $W[4]$", *Theoretical Computer Science* 363, (2006), 278–288.

[24] R.G. Downey, V. Estivill-Castro, M.R. Fellows, E. Prieto, F.A. Rosamond,"Cutting Up is Hard to Do: the Parameterized Complexity of k-Cut and Related Problems," Electronic Notes in Theoretical Computer Science 78, (2003), 205–218.

[25] R.G. Downey and M.R. Fellows,"Fixed-Parameter Tractability and Completeness I: Basic Results," *SIAM Journal on Computing* 24, 4 (1995), 873 - 921.

[26] R.G. Downey and M.R. Fellows,"Fixed-Parameter Tractability and Completeness II: On Completeness for $W[1]$," *Theoretical Computer Science* 141, 1-2 (1995), 109 - 131.

[27] R.G. Downey and M.R. Fellows,"Threshold Dominating Sets and an Improved Characterization of $W[2]$," *Theoretical Computer Science* 209 (1998), 123 - 140.

[28] R.G. Downey and M.R. Fellows, *Parameterized Complexity*, Springer, New York, 1999.

[29] R.G. Downey, M.R. Fellows, B. Kapron, M.T. Hallet, and H.T. Wareham,"Parameterized Complexity of Some Problems in Logic and Linguistics (extended abstract)", In *Proceedings of the 2nd Workshop on Structural Complexity and Recursion-theoretic Methods in Logic Programming*, 1993, *Lecture Notes in Computer Science* 813, 1994, 89-101.

[30] R.G. Downey, M.R. Fellows, and C. McCartin, "Parameterized Approximation Problems," in *Proceedings of the International Workshop on Parameterized and Exact Computation (IWPEC 2006)*, *Lecture Notes in Computer Science* 4169, 2006, 121–129.

[31] R.G. Downey, M.R. Fellows, and K. Regan, "Descriptive Complexity and the W-hierarchy," in P. Beame and S. Buss, editors, *Proof Complexity and Feasible Arithmetic*, *AMS-DIMACS* Volume Series 39 (1998), 119–134.

[32] R.G. Downey, M.R. Fellows, and U. Taylor,"On the Parametric Complexity of Relational Database Queries and a Sharper Characterization of $W[1]$," *Combinatorics, Complexity and Logic*, Proceedings of DMTCS '96, Springer-Verlag (1996), 194 - 213.

[33] R.G. Downey, M.R. Fellows, A. Vardy, and G. Whittle,"The Parameterized Complexity of Some Fundamental Problems in Coding Theory," *SIAM Journal on Computing* 29, 2 (1999) 545 - 570.

[34] J. Esparza and M. Nielsen,"Decidability Issues for Petri Nets - a survey", *J. Inform. Process. Cybernet. EIK* 30, 3 (1994), 143–160.

[35] M. Fellows and N. Koblitz, "Fixed Parameter Complexity and Cryptography," *Proeedings of the Tenth International Symposium on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC'93)*, Lecture Notes in Computer Science 673, Springer-Verlag (1993) 121–131.

[36] J. Flum and M. Grohe, "Fixed-Parameter Tractability, Definability, and Model-Checking," *SIAM J. Computing* 31,1 (2001), 113–145.

[37] J. Flum and M. Grohe, *Parameterized Complexity Theory*, Springer, 2006.

[38] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[39] N. Immerman, *Descriptive Complexity*, Springer-Verlag, 1999.

[40] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

[41] H. T. Wareham, "The Parameterized Complexity of Intersection and Composition Operations on Sets of Finite-State Automata," *International Conference on Implementation and Application of Automata*, Lecture Notes in Computer Science 2088, Springer (2000), 302–310.