Automata and game theoretic models of computation

A thesis submitted in partial fulfilment
of the requirements for
the degree of *Doctor of Philosophy*,
The University of Auckland.

Aniruddh Gandhi
August 2012

Abstract

In this thesis we investigate two finite state models of computation: automata and games on graphs. First we investigate the state complexity of finite word and tree automata from two directions. One direction is to study the interplay between non-deterministic automata (NFA) and deterministic automata (DFA) state complexity. In particular, we show that the exponential gap of the state explosion caused by the subset construction and the complementation of NFA may be filled. Another direction is to investigate the DFA state complexity of natural subclasses of regular languages. We focus on finite word and tree languages and provide improved upper bounds on the state complexity of union and intersection of such languages.

Secondly we generalize finite automata by introducing automata over arbitrary algebraic structures. For a structure $\mathcal{S}$, we use the term $\mathcal{S}$-automata for the class of automata operating over $\mathcal{S}$. An $\mathcal{S}$-automaton has a fixed number of registers and processes finite sequences of elements from the (possibly infinite) domain of $\mathcal{S}$. At every stage, it may test the input against the values in the registers using the relations of $\mathcal{S}$ and then based on the outcome of this test, move to another state after applying some operations from $\mathcal{S}$ to the registers. We investigate certain natural problems such as the validation problem and the emptiness problem and show that they may become decidable or undecidable if we change the underlying structure or the structure of the automaton in various natural ways.

Lastly we investigate the problem of solving Büchi and parity games on trees with back-edges. We present an efficient algorithm that solves a Büchi game played on trees with back-edges and then apply our analysis to parity games. We then present experimental evidence which shows that our algorithm for Büchi games performs asymptotically better than the classical algorithm in many cases. Also we present a concrete class of Büchi games for which the classical algorithm has a quadratic running time and our algorithm has linear running time.

Dedicated to Amma

Acknowledgements

I would like to thank my advisor Prof. Bakhadyr Khoussainov for his guidance and patience. He has been a tremendous source of ideas and inspiration and has always given me his time generously. He was also kind enough to organize two illuminating trips to Cornell University which have taught me a lot.

Thanks are also due to my co-author Jiamou Liu for many fruitful discussions and the many lessons learned by working with him. I would like to thank my colleagues and friends Alexander Melnikov and Tatyana Gvozdeva for many enjoyable conversations and their friendship. I would also like to acknowledge the University of Auckland for supporting my research through the UoA Doctoral Scholarship.

Finally I would like to thank my parents for all their love and support over the years. Without their encouragement and sacrifices, this work would never have been possible.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and motivation

In this thesis we investigate models of computation with a finite number of states. In particular we concentrate on two finite state models of computation: *automata* and *two-player games on graphs*. Our goal in the study of these models of computation is three-fold: the first is to study the algorithmic problems associated with them, the second is to generalize automata to operate over arbitrary algebraic structures and the third is to analyze the complexity of these models of computation.

An automaton has a finite number of states and transitions between them. The automaton reads an input and based on the input and its current state, makes a decision on which state it should transition to. Some states of the automaton are designated as accepting states. The idea is that the automaton accepts or rejects a sequence of inputs based on some condition involving the accepting states (called the *acceptance condition*). For example, the familiar finite state word automaton accepts or rejects finite words depending on whether it ends up in an accepting state after processing the word. Another example is the Büchi automaton which accepts an infinite word if some accepting states occur infinitely often during the processing of the word.

Given an automaton, an important question is to find whether there exists any sequence of inputs accepted by the automaton. This problem is referred to as the *emptiness problem* and it has applications in model checking, verification and logic. We review these connections later in this chapter.

Two-player games are played on a directed graph by two players: Player 0 and Player 1.

Each vertex of the graph is owned by one player. The game is played as follows: the players take turns moving a token from one vertex to another along the edges of the graph. The player that owns the vertex on which the token is currently placed decides where the token should be placed next. Therefore the game continues indefinitely or until a dead end is reached. Here we need to specify the most important ingredient naturally associated with a game: a rule to decide which player wins the game (called the *winning condition*). Analogous to the case of finite automata, there are various winning conditions that can be used and each gives rise to a different class of games. For instance, one way to specify a winning condition is to designate certain vertices as target vertices for Player 0 and say that the player wins the game if she is able to reach any of the target vertices. Such a winning condition gives rise to *reachability games*. Another type of winning condition also designates certain vertices as target vertices for Player 0 but specifies that the player wins if she is able to visit some target vertices infinitely often. Two-player games with this winning condition are called *Büchi games*.

Given a two-player game with a certain winning condition we generally seek to solve the following problem: Find those vertices from which Player 0 (or Player 1) may "win" the game i.e. she can satisfy the winning condition irrespective of the moves of the adversary. We refer to this problem as the *winning region problem* and as we will see later, it is important in modeling reactive systems and has connections with logic.

Finite state models of computation are important in computer science since they allow one to model a real system by a finite mathematical object. This abstraction allows us to reason about the behavior and properties of the system by analyzing the structure of the finite object. In this context we observe that automata and two-player games arise from different approaches to modeling a system.

One approach is to view the system as receiving inputs from the environment and changing its state based on the input and its current state. Many systems like computer programs and industrial automation robots may be naturally modeled using this approach. In this case, the automata model of computation comes to our aid and allows us to analyze the behavior of these systems. Some prominent examples of this approach are the modeling of real-time systems [3] and concurrent and distributed systems [59]. Another application of this approach is the modeling of hybrid systems which consist of discrete programs operating in an analog environment e.g. an airplane or an elevator. The authors of [2] introduced *hybrid automata* to effectively model such systems.

Another approach is to think of the interaction of the system with its environment as a game with two-players: the system and the environment. This view sees the interaction between the system and its environment as an adversarial process where the adversary (environment) is attempting to trick the system into behaving in an undesirable manner. In this case, it is natural to use two-player games as our model. An important example where such an approach has been successfully is in the domain of reactive systems [88, 69]. A game theoretic approach has also been applied for analyzing the security of computer networks [58].

After we have chosen one of these models of computation to model our system, some useful questions that one may ask about a system's behavior are:

1. Does the system end up in an undesirable state for a given sequence of inputs?
2. Does there exist some sequence of inputs to a system that cause it to enter an undesirable state?
3. Can the environment force the system to enter an irrecoverable slide?
4. Is the system specification realizable in practice?

In the automata framework, question 2 is equivalent to asking whether there exists some sequence of inputs accepted by an automaton i.e. the *emptiness problem*. In the context of two-player games, question 3 is equivalent to the *winning region problem*. Also if our system is modeled as a two-player game, finding whether our specification is realizable (question 4) corresponds to the *winning region problem* for the two-player game [1, 76]. Since the model of computation has a finite number of states, questions of this type become decidable in many cases.

Finite state models of computation also play an important role in logic. Logic is essential for formally expressing properties that we would like our system to have and forms the backbone of model checking and verification. Many different kinds of logic are useful depending on the kinds of properties we want to express e.g. propositional logic, predicate logic, linear temporal logic (*LTL*) [75], computational tree logic (*CTL*) [21, 20] etc. Fixed-point logics , particularly the modal μ-calculus introduced by Kozen in [57], are of fundamental importance in the field of model checking and verification [30]. The idea is that we first express a property of the system in the language of some formal logic and then decide if the formula is satisfiable.

There is a deep connection between finite state models of computation and logic. It arises from the insight that formulae of certain kinds of logics can be translated into a finite state model of computation i.e the formula is satisfiable if and only if some property of the constructed finite state model of computation holds[89]. If we have a procedure for deciding whether this property holds on the finite state model of computation, then we can decide if formulae of the logic are satisfiable. This in turn gives us the ability to decide if our system satisfies the property specified by the formula.

There are numerous examples where a formula of a particular logic can be translated into a finite state model of computation. One of the earliest such connections was discovered by Büchi when he proved that monadic second order logic of one successor ($S1S$) is decidable since a formula written in this logic could be translated into a Büchi automaton [15, 29]. The satisfiability of the formula can then be determined by finding out if there exists an input which is accepted by the constructed automaton i.e. solving the *emptiness problem* for the Büchi automaton.

Another important example where finite models of computation are applied to logic is the modal μ-calculus which is of great importance in model checking and verification [30]. The basic idea is that formulae of the modal μ-calculus may be translated into a certain kind of two-player game called *parity games*. Then the satisfiability of the μ-calculus formula can be ascertained by solving the winning region problem for the parity game [31, 32, 86, 90]. All these applications of finite state models of computation to logic trace their roots back to the interaction between Turing machines and the arithmetical hierarchy [79, 85].

The wide range of applications of automata and games in computer science and logic indicate the versatility and robustness of these models. In this thesis, we focus on the following broad themes regarding automata and two-player games:

1. How complex can automata be? This question implicitly assumes a definition of "complexity" of automata. As we will see in Chapter 2, there are various ways of defining complexity of automata. Under this general theme, some natural questions are:

    - What is the increase in complexity if we convert a nondeterministic automaton to a deterministic one?
    - What is the cost of performing various standard operations on automata?
    - How is the analysis of the above questions affected if we restrict the structure of

the automata?

In Chaper 3 we investigate the above questions.

2. Can we extend the automata model for arbitrary algebraic structures? Traditionally, automata have been viewed as reading inputs which are words (or trees) over a finite set of symbols. Also, the only action they may perform is changing their state depending on the input and the current state. However in practice, computer programs process inputs that could be more complicated e.g. sets, natural numbers, real numbers etc. Also computer programs may use natural operations and predicates over these inputs (such as adding or multiplying numbers, union/intersection of sets etc.).

   In this case, the concept of algebraic structures is a natural way to encapsulate the underlying (possibly infinite) domain of inputs and operations/predicates on the set of inputs. An algebraic structure consists of a possibly infinite domain D and finitely many atomic operations $f_1, \ldots, f_m$, relations $R_1, \ldots, R_n$ and constants $c_1, \ldots, c_\ell$ from D.

   In order to model such systems effectively we need to generalize the automata model to operate over arbitrary algebraic structures. In Chapter 4, we introduce a model of automata over structures which fulfills this goal. We analyze various classical automata theoretic problems (such as the *emptiness problem* mentioned earlier) for our new model of automata and show that these problems become decidable under some natural restrictions over the underlying structures or the automata themselves.

3. Can we develop efficient algorithms for solving the winning region problem for two-player games if we restrict the structure of the graphs over which games are played? In this thesis we focus on two-player games on finite directed graphs with Büchi and parity winning conditions. It is well known that the winning region problem for a Büchi game played on a graph with n vertices and m edges is $O(n \cdot (n + m))$ [41](Ch. 2). However the corresponding problem for parity games is known to be in $\mathsf{NP} \cap \mathsf{co} - \mathsf{NP}$ but not in P [41](Ch. 6). Therefore, a natural direction is to analyze the winning region problem for such games for restricted classes of graphs and to develop efficient algorithms for these classes.

   In Chapter 5, we investigate the problem of solving Büchi and parity games on trees

with back-edges which occur frequently in computer science. We show that such games can be decomposed in a nice way and then exploit this structure to develop an efficient algorithm for such games.

The above themes are naturally related to three interconnected fields of research. The first theme lies within the realm of the study of complexity of regular languages, the second concerns generalization of the automata model and the third has its origins in infinite games on graphs. These are active fields of research and we would like to provide some background on them to provide the necessary context for this thesis. To this end, in the subsequent sections we review the developments in the fields of complexity of regular languages, generalizations of the automata model and infinite games on graphs.

1.1.1 Complexity of regular languages

We assume familiarity with the basics of regular languages and finite automata which may be found in standard textbooks such as [47]. There have been two main directions in the study of complexity for automata and formal languages. The first direction is the study of *time and space complexity* for performing decision procedures or operations on automata. This is important for example in designing algorithms that solve the emptiness, reachability, complementation and minimization problems for automata[6, 47, 42]. The second direction is that of *descriptional complexity*, which is a measure of complexity for various classes of formal languages and their operations. This notion is important for example in investigating state explosion that occurs when transforming a nondeterministic automaton into an equivalent deterministic one [47, 51, 36]. It is also important when analyzing the minimal automaton for recognizing complementation of ω-languages recognizable by Büchi automata [80]. A good survey of these two directions may be found in [46]. These two directions of study are interrelated; the descriptional complexity for operations on language also gives lower bound for the time and space complexity for performing these operations.

One way of measuring the descriptional complexity of formal languages is *state complexity*. For a regular language L, its *NFA-state complexity (DFA-state complexity)* is the number of states in a minimal NFA (minimal DFA) that recognizes L. There have been a series of results that study the state complexity of the Boolean operations, the concatenation and the Kleene-star operation on regular languages. For instance, in [45] it is shown that $n + m + 1$

states are necessary and sufficient to recognize the union of regular languages L_1 and L_2, recognized by n state and m state NFA, respectively. Similarly, in [45] it is shown that $n \cdot m$ states are necessary and sufficient to recognize the intersection of regular languages L_1 and L_2 recognized by n state and m state NFA, respectively. The papers [45] and [50] study the state complexity of other operations. Also, there has recently been some work on the study of average state complexity of regular languages and operations thereon. Regular tree languages, which are classes of finite trees that are recognizable by tree automata, are natural extensions of regular word languages. Recently there has been an increasing body of work devoted to state complexity of tree languages [73, 74].

Another way of measuring the descriptional complexity of regular languages is *transition complexity*. For a given regular language L, its transition complexity is the number of transitions in the minimal NFA recognizing L. Transition complexity of a regular language seems to be a better measure of the descriptional complexity of a regular language since the transitions of the minimal NFA are needed to completely specify a regular language. Moreover, the transition complexity of a regular language L may be exponentially greater than the NFA-state complexity of L. The papers [42, 82] investigate the transition complexity of regular languages.

1.1.2 Generalizations of the automata model

As we mentioned earlier, traditionally automata are finite state machines which read inputs from a finite set of symbols and change their state depending on the symbol read. However, most algorithms use methods, operations, and test predicates over an already defined underlying structure. For instance, algorithms that work on graphs or trees assume that the underlying structure consists of graphs and trees with operations such as adding or deleting a vertex or an edge, merging trees or graphs, and test predicates such as the subtree predicate.

Algebraic structures are an apt way to capture this situation. An structure is denoted by $\mathcal{S} = (D; f_0, \ldots, f_n, R_0, \ldots, R_k)$ where each f_i is an atomic operation on D and each R_i is a predicate on D. An algorithm may be thought of as a sequence of instructions that uses the operations and predicates of the structure. This simple observation has led to the introduction of various generalizations of the automata model over arbitrary structures and their analysis. The first example here is the class of Blum-Shub-Smale (BSS) machines [9], where the underlying structure is the ordered ring of the reals. The model is essentially a multiple

register machine that stores tuples of real numbers and that can evaluate polynomials at unit cost. The second example is the work of O. Bournez, et al. [12], where the authors introduce computations over arbitrary structures thus generalizing the work of L. Blum, M. Shub and S. Smale [9]. In particular, among several results, they prove that the set of all recursive functions over arbitrary structure $\mathcal{S}$ is exactly the set of decision functions computed by BSS machines over $\mathcal{S}$. The third example is various classes of counter automata that use counters in different ways [13, 24, 48, 60, 65].

Another motivation for generalizing the automata model comes from the field of program verification and databases. There has recently been a lot of interest the study of finite automata over infinite alphabets [4, 10, 11, 34, 71, 83, 87]. One goal of these investigations is to extend automata-theoretic techniques to words and trees over data values. Several models of computations have been proposed towards this goal. For example, Kaminsky and Francez in [54] proposed register automata. These are finite state machines equipped with a fixed number of registers which may hold values from an infinite domain D. The operations allowed by the automata are equality comparisons between the input and the register values and the copy operation. Another example is pebble automata introduced by Neven, Schwentick and Vianu [71]. Here the automata use a fixed set of pebbles with a stack discipline to keep track of values in the input data words. Operations include equality comparisons of the current pebble values, and dropping and lifting a pebble. Other examples of such automata models include Bojanczyk's data automata [10] and Alur's extended data automata [4]. While all the above automata models allow only equality tests between data values, there has also been automata model proposed for linearly ordered data domains [84].

1.1.3 Infinite games on graphs

In this section, we give a brief account of the development of the field of infinite games on graphs. For a detailed study of two-player infinite games on graphs and their connection with logic, the reader is referred to [41]. As we mentioned previously, two-player infinite games on graphs can be used to model and analyze reactive systems. One of the earliest survey on the interplay between logic and games and the application of such games to the synthesis of digital circuits was published by Church [19] in 1962. In [19], Church set out the major problems in the field at the time including the *synthesis problem* which was to determine whether a specification was actually realizable. However Church did not

explicitly use the terminology of two-player games.

The currently prevalent terminology of infinite games on graphs was first introduced by McNaughton in [64]. McNaughton games are special case of Borel games and therefore by the result of [62], McNaughton games are *determined* i.e. for any given node in a McNaughton game, one of the players always has a winning strategy. In fact, McNaughton showed that such games have finite state winning strategies i.e. the next move of a player is determined by a finite amount of *history* of that particular play. Note that such strategies do not exist in the more general class of Borel games. The finite state winning strategies described in [64] are known as *last visitation record* (LVR) strategies. These strategies were inspired by *last appearance record* (LAR) strategies described earlier by Gurevich and Harrington [43]. Nerode, Remmel and Yakhnis subsequently used finite state winning strategies of McNaughton games to model distributed concurrent systems and hence established the connection between infinite games and distributed systems[69, 70].

The study of minimizing the amount of memory required for winning strategies was started by [31] and [68] where memoryless winning strategies were shown to exist for parity games i.e. the next move of a player is only dependent on its current position. Also in [28], the authors provide examples of McNaughton games with $O(n)$ nodes such that every winning strategy requires $n!$ memory and hence show that the LVR strategy described in [64] is optimal.

1.2 Summary of results

In this section we provide a summary of results from each chapter. The reader is referred to the relevant chapter for a detailed discussion of each topic, formal definitions and proofs.

Chapter 2. Preliminaries

In this chapter, we introduce basic definitions and facts about finite automata, algebraic structures and infinite games. The terminology and concepts introduced here are used throughout this thesis. This chapter is divided into three sections: the first section concerns finite automata and introduces finite word and tree automata, the second section introduces structures and the third section introduces infinite games on graphs.

In the third section, we prove that reachability, Büchi and parity games enjoy memoryless determinacy (Theorems 2.3.2, 2.3.3 and 2.3.4). In particular we provide a new proof of

the memoryless determinacy of parity games which is much simpler than the proof given in [41](Ch. 6).

Chapter 3. Complexity of regular languages

In this chapter, we investigate the complexity of regular languages from two directions: the first direction is to investigate the relationship between the NFA and the DFA-state complexity of a regular language and the second direction is to investigate the DFA-state complexity of natural subclasses of regular word and tree languages. Accordingly, this chapter is divided into two main sections.

In the first section of this chapter (section 3.1) we investigate the interplay between the NFA and DFA-state complexity of regular word languages. It is well known that the cost of constructing a DFA equivalent to a NFA (using the *subset construction*) is exponential i.e. for any n there exists an n state NFA recognizing a language L such that 2^n number of states are needed for a DFA to recognize the complement of the language L [78, 66]. The situation is similar for the case of complementation of NFA's i.e. there exists a NFA with n states such that the NFA recognizing its complement needs 2^n states [66].

In section 3.1.1 we revisit subset construction and investigate the problems around the following question: given n and m such that $n \leq m \leq 2^n$, does there exist a regular language whose NFA-state complexity is n and its DFA-state complexity is m? Then in section 3.1.2, we focus on the following question: given n and m with $n \leq m \leq 2^n$, does there exist a regular language whose NFA-state complexity is n such that the NFA-state complexity of the complement of the language is m? We present asymptotic solutions to these problems. In particular, we prove the following:

1. For every $k > 1$ there exists a regular language L_n over a k-letter alphabet, where $n > k$, such that a minimal NFA recognizing L_n needs exactly n states and the minimal DFA recognizing L_n needs exactly $(k+1)\cdot n - c$ states and $O(n)$ transitions, where $c = (k+1)^2 - 2$ (Theorem 3.1.3).

2. Let $m = \frac{2^{k+1} - pk - 2p}{p-1}$ for any $p, k > 0$. Then for every $n = k + m + 2$ there exists a regular language L_n over the binary alphabet such that the minimal NFA recognizing L_n needs exactly n states and the minimal DFA needs exactly $p \cdot n$ states and $O(n)$ transitions (Theorem 3.1.6).

3. For every $k > 1$ there exists a regular language L_n over a k-letter alphabet such that

the minimal NFA $\mathcal{A}$ recognizing L_n needs n states, where $n > k$, and the minimal DFA recognizing L_n has asymptotically n^k states. The NFA $\mathcal{A}$ has $O(\frac{n^2}{log_2(n)})$ transitions (Theorem 3.1.14).

4. For every $k > 1$ there exists a regular language L_n over the k-letter alphabet, where $n > k$, such that the minimal NFA recognizing L_n needs exactly n states and the minimal NFA recognizing the complement of L_n needs exactly $n(k+1) - c$ states and $O(n)$ transitions, where $c = 2(k+1) - 2$ (Theorem 3.1.16).

5. For every $k > 1$ there exists a regular language L_n over the k-letter alphabet, where $n > k$, such that the minimal NFA $\mathcal{A}$ recognizing L_n needs $O(n)$ states and the minimal NFA recognizing the complement of L_n needs between $O(n^{k-1})$ and $O(n^{2k})$ states. Moreover $\mathcal{A}$ has $O(\frac{n^2}{log_2(n)})$ transitions (Theorem 3.1.21).

The results of this section have been published in [36].

In the second section of this chapter (section 3.2), we investigate the DFA-state complexity of finite word and tree languages. In particular, we show the following:

1. We first consider the class of finite word languages which have all words of the same length, say h (we call these *uniform-length languages* of length h). Suppose $h = 2^i + i$ for some $i \geq 0$. The state complexity of the class of uniform-length languages of length h is $\Theta(2^h/h)$ (see Theorem 3.2.4). Here we consider the case of a binary alphabet but the proof may be extended for an alphabet of greater size.

2. The state complexity of the class of finite word languages whose words have length bounded by h, where $h = 2^i + i$ for some $i \geq 0$, is $\Theta(2^h/h)$ (see Theorem 3.2.6). As in the previous case, we consider a binary alphabet but the result may be generalized for alphabets of greater size.

3. The state complexity of union and intersection of two uniform-length languages is at most $\frac{k-1}{k}mn + m + n - 2$ where $k = |\Sigma|$ and m, n are the number of states in the input minimal automata (see Theorem 3.2.9).

4. Consider two finite word languages L_1, L_2 over alphabet Σ of size k which have state complexities m, n respectively. The state complexity of $L_1 \cup L_2$ (and $L_1 \cap L_2$) is at most $m \cdot n - \log_k(m)(m+n) + 3m + n + 2$ states (see Theorem 3.2.13)

5. Consider two finite tree languages L_1, L_2 with state complexities m, n respectively such that L_1, L_2 that are subsets of the $\{1\}$-labeled k-ary trees. The state complexity of $L_1 \cup L_2$ (and $L_1 \cap L_2$) is at most $m \cdot n - (\log_k \log_2(m))(m+n) + 9m + n$ (see Theorem 3.2.18). This result may also be generalized for finite tree languages which are subsets of Σ-labeled k-ary trees for $|\Sigma| > 1$ (see Corollary 3.2.19).

The results of this section have been published in [39].

Chapter 4. Finite automata over structures

In this chapter, we introduce the notion of finite automata over algebraic structures. Recall that an algebraic structure $\mathcal{S}$ consists of a (possibly infinite) domain D, a finite number of atomic operations $f_1, \ldots, f_m$, relations $R_1, \ldots, R_n$ and constants $c_1, \ldots, c_\ell$ from D. A D-word is a finite sequence $a_1 \ldots a_t$ of elements from the domain D and a D-language is a set of D-words. We use $\mathsf{Op}(\mathcal{S}), \mathsf{Rel}(\mathcal{S})$ to denote the atomic operations and relations of $\mathcal{S}$ respectively. This chapter is divided into five sections.

In the first section of this chapter (section 4.1), we define automata over any given structure $\mathcal{S}$. Such an automaton is equipped with a finite number of states, a fixed number of registers, a read only head that always moves to the right in the tape and transitions between the states.

The automaton processes D-words. During the computation, given an input from the sequence, the automaton tests the input against the values of the registers. Depending on the outcomes of the test, the automaton updates the register values by performing basic operations on the input and the register values and then makes a transition to a state. The automaton accepts a D-word if it reaches an accepting state after procesing it. The language accepted by the automaton is the set of all D-words accepted by it. Given a structure $\mathcal{S}$, we use the term $\mathcal{S}$-*automata* to denote the instantiation of this computation model for $\mathcal{S}$. The formal definition of $\mathcal{S}$-automata is as follows:

Definition 4.1.2 *An* $(\mathcal{S}, k)$-automaton *is a tuple* $\mathcal{A} = (Q, \alpha, \overline{x}, \Delta, q_0, F)$ *where* Q *is a finite set of states, the mapping* α *is a function from* Q *to* $\mathsf{Rel}^{k+\ell}(\mathcal{S})$, $\overline{x} \in D^k$ *are the initial values of the registers,* $q_0 \in Q$ *is the initial state,* $F \subseteq Q$ *is the set of accepting states and* $\Delta \subseteq Q \times \{0,1\}^{k+\ell} \times Q \times \mathsf{Op}^k(\mathcal{S})$ *is the transition relation of* $\mathcal{A}$*. We refer to the first k registers (whose values may change during the course of a run) as* changing *registers.*

The $(\mathcal{S}, k)$*-automaton is* deterministic *if for each* $q \in Q$, $\overline{b} \in \{0,1\}^{k+\ell}$*, there is exactly one* q'

and $\overline{g} \in \mathsf{Op}^k(\mathcal{S})$ *such that* $(q, \overline{b}, q', \overline{g}) \in \Delta$. *An* (deterministic) $\mathcal{S}$-automaton *is an (deterministic)* $(\mathcal{S}, k)$*-automaton for some k.*

In the following, we will use pr_i ($i \in \{1, 2\}$) to denote the operation defined as follows: $\mathrm{pr}_i(x_1, x_2) = x_i$. Also we use $\mathcal{S}[R_1, \ldots, R_m, f_1, \ldots, f_n, a_1, \ldots, a_d]$ to denote the structure obtained by adding the relations $R_1, \ldots, R_m$, the operations $f_1, \ldots, f_n$ and the constants $a_1, \ldots, a_d$ to $\mathcal{S}$.

In the second section of this chapter (section 4.2), we use several examples to demonstrate some simple properties of $\mathcal{S}$-automata. First we show that all regular word languages can be recognized by $\mathcal{S}$-automata. Let $\mathcal{S}$ be a structure. Suppose that the structure $\mathcal{S}$ contains an atomic equivalence relation $\equiv$ of finite index. Let $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ be the set of all equivalence classes of $\equiv$. For every word $w = w_1 \ldots w_n$ over the alphabet Σ, let $R(w)$ be the D-language $\{a_1 \ldots a_n \mid a_i \in w_i \text{ for all } i = 1, \ldots, n\}$. For every language $\mathcal{L}$ over Σ, let $R(\mathcal{L})$ be the D-language $\bigcup_{w \in \mathcal{L}} R(w)$. Then we prove the following theorem.

Theorem 4.2.1 *Let* a_i *be an element from the* $\equiv$*-equivalence class* σ_i*, where* $i = 1, \ldots, k$*. Then each of the following is true.*

- *For every regular language* $\mathcal{L}$ *over* Σ*, the D-language* $R(\mathcal{L})$ *is accepted by an* $(\mathcal{S}[a_1, \ldots, a_k], 0)$*-automaton.*

- *Suppose the signature of* $\mathcal{S}$ *contains only one relation* $\equiv$ *and the atomic operations of* $\mathcal{S}$ *are compatible with* $\equiv$*. For every* $\mathcal{S}$*-automata recognizable D-language W, there is a regular language* $\mathcal{L}$ *over* Σ *such that* $W = R(\mathcal{L})$*.*

Then we show the following two separation results via examples:

- The class of D-languages accepted by $(\mathcal{S}, k)$-automata properly contains the class of D-languages accepted by $(\mathcal{S}, k-1)$-automata (see example 4.2.2).

- Deterministic $\mathcal{S}$-automata form a proper subclass of $\mathcal{S}$-automata. Furthermore, the class of $\mathcal{S}$-automata recognizable D-language is not closed under Boolean operations in the general case (see example 4.2.3).

In the third section (4.3), we provide several examples of determinisitc $\mathcal{S}$-automata and prove that languages recognized by deterministic $\mathcal{S}$-automata are closed under the boolean operations. In particular we show the following:

Theorem 4.3.3 *Let $\mathcal{S}$ be a structure. The class of languages recognized by deterministic $\mathcal{S}$-automata is closed under union, intersection and complementation.*

In the fourth section (4.4), we investigate the *validation problem* which is formulated as follows:

Validation problem. Design an algorithm that, given an $\mathcal{S}$-automaton $\mathcal{A}$ and a path p in $\mathcal{A}$ from the initial state to an accepting state, decides if there exists a D-word $\overline{a}$ such that a run of $\mathcal{A}$ over $\overline{a}$ proceeds along p.

The validation problem for $\mathcal{S}$-automata turns out to be equivalent to solving systems of equations and in-equations over the structure. Formally, deciding the validation problem is equivalent to deciding the existential theory of the structure $\mathcal{S}$. In particular we prove the following theorem.

Theorem 4.4.5 *The validation problem for $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automata is decidable if and only if the existential theory of $\mathcal{S}$ is decidable.*

In the fifth section of this chapter (section 4.5), we investigate the *emptiness problem* for $\mathcal{S}$-automata which can be stated as follows:

Emptiness problem. Design an algorithm that, given a structure $\mathcal{S}$ and an $(\mathcal{S}, k)$-automaton $\mathcal{A}$, decides if $\mathcal{A}$ accepts at least one D-word.

The answer to this questions obviously depends on many parameters: the structure $\mathcal{S}$, the number of registers of the automaton, the structure of the automaton etc. In this section we show that the emptiness problem may become decidable or undecidable by varying these parameters. In section 4.5.1, we restrict the structure of the $\mathcal{S}$-automata to be acyclic. We observe that the emptiness problem for acyclic automata is computationally equivalent to the validation problem and have the following theorem.

Theorem 4.5.1 *For any structure $\mathcal{S}$, the emptiness problem of acyclic $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automata is decidable if and only if $\mathcal{S}$ has decidable existential theory.*

Then we consider acyclic automata over two structures which occur naturally in mathematics: $\mathcal{S}_{\mathbb{Z}} = (\mathbb{Z}; +, \times, \mathrm{pr}_1, \mathrm{pr}_2, =, 0)$ and $\mathcal{S}_{\mathbb{N}} = (\mathbb{N}; +, \times, \mathrm{pr}_1, \mathrm{pr}_2, =, 0)$. Note that these structures do not have decidable existential theory. We prove the emptiness problem for $\mathcal{S}_{\mathbb{Z}}$ ($\mathcal{S}_{\mathbb{N}}$) acyclic automata becomes undecidable with a sufficient number of registers by using a reduction from Hilbert's tenth problem. In particular we show the following.

Proposition 4.5.4 *The emptiness problem for deterministic acyclic $(\mathcal{S}_{\mathbb{Z}}, 11)$-automata and $(\mathcal{S}_{\mathbb{N}}, 12)$-automata is undecidable.*

In section 4.5.2, we investigate the emptiness problem for $\mathcal{S}$-automata when the domain of $\mathcal{S}$ in the natural numbers $\mathbb{N}$. First we show that if we remove the acyclicity constraint from the automaton, the emptiness problem is undecidable for $\mathcal{S}$-automata with a small number of registers. In particular we prove the following theorem.

Theorem 4.5.5 *Let* $\mathcal{S}_1 = (\mathbb{N}; +1, -1, =, \mathrm{pr}_1, 0)$ *and* $\mathcal{S}_2 = (\mathbb{N}, +1, =, \mathrm{pr}_1, \mathrm{pr}_2, 0)$.

(a) The emptiness problem for deterministic $(\mathcal{S}_1, 2)$*-automata is undecidable.*

(b) The emptiness problem for deterministic $(\mathcal{S}_2, 4)$*-automata is undecidable.*

Then we show that if we reduce the number of registers to 1, the emptiness problem becomes decidable. We have the following theorem.

Theorem 4.5.6 *Let* $\mathcal{S}$ *be the structure* $(\mathbb{N}; +, \times, \mathrm{pr}_1, \mathrm{pr}_2, =, \leq, c_1, \ldots, c_\ell)$ *where* $c_1, \ldots, c_\ell$ *are arbitrary constants in* $\mathbb{N}$*. The emptiness problem for* $(\mathcal{S}, 1)$*-automata is decidable.*

Next in section 4.5.3, we put a natural constraint on the allowable transitions of the automata and show that by allowing only those transitions that compare the input or exactly one register with the constants, the emptiness problem may become decidable. More precisely we allow the input to be compared with exactly one of the changing registers using the = relation and compared to the constants using any of atomic relations of the structure. We call such automata as *constant comparing* automata. The subclass of constant comparing automata where the input is only allowed to be compared with constants (and not the changing registers) are called *strongly constant comparing* automata. The reader is referred to definition 4.5.5 for the formal defintion of these automata. We first show that the emptiness problem for constant comparing automata over the structure $(\mathbb{N}; +, \times, \mathrm{pr}_1, \mathrm{pr}_2, =, \leq, c_1, \ldots, c_\ell)$ is decidable for an aribtrary number of registers.

Theorem 4.5.11 *Let* $\mathcal{S}$ *be the structure* $(\mathbb{N}; +, \times, \mathrm{pr}_1, \mathrm{pr}_2, =, \leq, c_1, \ldots, c_\ell)$ *where* $c_1, \ldots, c_\ell$ *are arbitrary constants in* $\mathbb{N}$*. The emptiness problem for constant comparing* $\mathcal{S}$*-automata is decidable.*

Next we analyze the emptiness problem for strongly constant comparing automata and prove the following theorem.

Theorem 4.5.16 *Let* $\mathcal{S} = (\mathbb{N}; +, -, \mathrm{pr}_1, =, \leq, c_1, \ldots, c_\ell)$ *where* $c_1, \ldots, c_\ell$ *are constants in* $\mathbb{N}$*.*

(a) The emptiness problem for constant comparing $(\mathcal{S}, 2)$*-automata is undecidable if* $\ell \geq 2$*.*

(b) The emptiness problem for strongly constant comparing $\mathcal{S}$*-automata is decidable.*

The results of this chapter have been published in [38].

Chapter 5. Infinite games played on trees with back-edges

In this chapter, we investigate infinite games on graphs with *Büchi* and *parity* winning conditions. Given a directed graph G, a *Büchi game* played on G specifies a set of target nodes T in G, and Player 0 wins the game from a node u if the player has a strategy to visit nodes in T infinitely often starting from u. A *parity game* on G associates a priority $\rho(u) \in \mathbb{N}$ with every node u. Player 0 wins the game from a node u if the player has a strategy such that the minimum priority amongst all the nodes visited infinitely often in any play starting from u is even.

Recall that the *winning region problem* is to determine all those nodes from which Player 0 (Player 1) wins the game. It is well known that the winning region problem for Büchi games can be solved in polynomial time. By contrast, though intensely studied, polynomial time algorithms for solving the winning region problem for parity games remain unknown. Parity games are known to be in $\mathsf{NP} \cap \mathsf{Co\text{-}NP}$ but not known to be in P.

The classical algorithm for Büchi games has a running time of $O(n \cdot (n + m))$ where n, m are the number of nodes and vertices of the underlying graph respectively [41]. However the classical algorithm seems to be repetitive in nature and hence it is natural to carry out a more detailed analysis of Büchi games to see if it can be improved. For instance, the paper [17] investigates the class of graphs with constant out-degrees and shows that solving Büchi games played on such graphs takes $O(n^2/\log n)$ time. For graphs with unbounded out-degrees, the paper [16] presents an algorithm that runs in time $O(n \cdot m \cdot \log \delta(n)/\log n)$ where $\delta(n)$ is the out-degree of the game graph. These investigations suggest the idea of designing more efficient algorithms in specified classes of graphs such as trees with back-edges.

We first analyze Büchi games played on trees with back-edges and then apply our analysis to parity games on trees with back-edges. This chapter is divided into five sections. In the first section (5.1), we lay out the basic definitions and terminology of trees with back-edges. We also prove a normal form lemma (Lemma 5.1.2) for games played on trees with back-edges.

In the second section (5.2), we analyze Büchi games played on trees with back-edges. In our analysis, we use the notion of *snares* to classify the winning nodes of Player 0 as follows. Intuitively, a snare of rank 0 is a subtree from which Player 0 has a strategy to stay in the subtree forever and win the game. A snare of rank i, $i > 0$, is a subtree from which

Player 1 may choose between two options: (a) staying in the subtree forever and losing the game, or (b) going to an $(i-1)$-snare. We show that the collection of all snares corresponds exactly to winning nodes of Player 0. We present an efficient algorithm that solves a Büchi game played on trees with back-edges. The algorithm runs in time $O(\min\{r \cdot m, \ell + m\})$ where r is the largest rank of a snare and ℓ is the external path length, i.e., sum over all the leaves, of the distances from the root to each leaf in the underlying tree. In particular, we prove the following theorem.

Theorem 5.2.9 *There exists an algorithm that solves any Büchi game $\mathcal{G}$ played on trees with back-edges in time $O(\min\{r \cdot m, \ell + m\})$ where r is the snare rank, m is the number of edges and ℓ is the external path length of $\mathcal{G}$.*

In the third section (5.3), we apply our analysis for Büchi games to the case of parity games played on trees with back-edges. We reduce the problem of solving parity games played on trees with back-edges to solving Büchi games (Lemma 5.3.1) and prove the following theorem.

Theorem 5.3.2 *Any parity game $\mathcal{G}$ played on trees with back-edges can be solved in time $O(\ell + m)$ where ℓ is the external path length of $\mathcal{G}$ and m is the number of edges in $\mathcal{G}$.*

In the fourth section (5.4), we describe the results of experiments to compare the running our algorithm for Büchi games to the classical algorithm for Büchi games. The experiments can be broadly divided into two categories: average case running time comparison and running time comparison with random sampling. Our experiments clearly show that, in practice, not only our algorithm is much more efficient asymptotically but it also performs better for games with small number of nodes on the set of trees with back-edges.

In the fifth section (5.5), we support the positive results of the experiments described in section 5.4 by providing some concrete examples of Büchi games where our algorithm outperforms the classical algorithm. We present a class of Büchi games on trees with back-edges, $\mathcal{E}$, where our algorithm performs asymptotically better than the classical algorithm. In particular we prove the following.

Proposition 5.5.2 *For the class of Büchi games $\mathcal{E}$, the classical algorithm has a quadratic running time of $O(n \cdot (n + m))$ whereas our algorithm has a linear running time of $O(n + m)$.*

The results of this chapter have been published in [37].

Chapter 2

Preliminaries

As discussed in Chapter 1, our goal is to study two models of computation: finite automata and two-player games. In this chapter we provide some basic definitions and facts about finite automata and two-player games. The definitions and terminology we introduce in this chapter will be used throughout this thesis.

We will use $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ to denote the natural numbers, integers, rational numbers and real numbers respectively. Also we use $\mathbb{N}^+, \mathbb{Q}^+, \mathbb{R}^+$ to denote the positive natural numbers, rational numbers and real numbers respectively.

2.1 Finite automata

In this section we introduce finite word and tree automata and provide some basic facts about them. For a detailed background on automata theory, the reader is referred to [47].

2.1.1 Finite word automata

We use Σ to denote a finite set of symbols which is referred to as the alphabet. We denote the empty word by ϵ. We use Σ^* to denote the set of all finite words over Σ and $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$. A language L is a subset of Σ^*. For $\sigma \in \Sigma$ and $n \in \mathbb{N}$, σ^n is the word obtained by concatenating σ to itself n times and $\sigma^0 = \epsilon$. Also $\sigma^* = \{\sigma^n \mid n \in \mathbb{N}\}$ and $\sigma^+ = \sigma^* \setminus \{\epsilon\}$. We now define finite word automata.

Definition 2.1.1 *A* deterministic finite word automaton *(DFA)* $\mathcal{A}$ *over* Σ *is a 4-tuple* $\langle S, \delta, s_0, F \rangle$ *such that:*

1. *S is the finite set of states.*
2. *$\Delta : S \times \Sigma \to S$ is the transition function.*
3. *$s_0 \in S$ is the initial state.*
4. *$F \subseteq S$ is the set of accepting states.*

A nondeterministic word automaton is defined similarly except that we may have more than one initial state and each state may have multiple outgoing transitions labeled by the same symbol. Formally:

Definition 2.1.2 *A* nondeterministic finite word automaton *(NFA) $\mathcal{A}$ over Σ is a 4-tuple $\langle S, \delta, S_0, F\rangle$ such that:*

1. *S is the finite set of states.*
2. *$\Delta : S \times \Sigma \to 2^S$ is the transition function.*
3. *$S_0 \subseteq S$ is the set of initial states.*
4. *$F \subseteq S$ is the set of accepting states.*

Given a NFA $\mathcal{A} = \langle \mathcal{S}, \delta, S_0, F\rangle$, we define $\delta^+ : S \times \Sigma^+ \to 2^S$ recursively by:

1. $\delta^+(s, \sigma) = \delta(s, \sigma)$ and
2. $\delta^+(s, w \cdot \sigma) = \delta(\delta^+(s, w), \sigma)$

where $s \in S$, $\sigma \in \Sigma$ and $w \in \Sigma^+$. Also for each $X \subset S$ and $w \in \Sigma^+$, we let $\delta^+(X, w) = \bigcup_{s \in X} \delta^+(s, w)$. Intuitively $\delta^+(s, w)$ is the set of all states of $\mathcal{A}$ that may be reached by reading the word w starting from the state s.

A *run* of the automaton $\mathcal{A}$ on the finite word $w = \sigma_1\sigma_2 \ldots \sigma_n$ is the sequence of states $s_0, s_1, \ldots, s_{n-1}, s_n$ such that s_0 is an initial state and $s_{i+1} \in \Delta(s_i, \sigma_i)$. The run is said to be *accepting* if $s_n \in F$. Note that a DFA has exactly one run on a given word while a NFA may have more than one run on the same word. The automaton $\mathcal{A}$ is said to *accept* the word w if it has an accepting run on w. The *language* recognized by an automaton, denoted by $L(\mathcal{A})$, is defined as follows:

$$\{w \in \Sigma^* \mid \mathcal{A} \text{ accepts } w\}.$$

It is well known that the class of languages recognized by NFA's is exactly the same as that recognized by DFA's i.e. for every NFA $\mathcal{A}$, there exists a DFA $\mathcal{B}$ such that $L(\mathcal{A}) = L(\mathcal{B})$. The class of languages recognized by finite word automata are known as *regular languages*, which we denote by $\mathfrak{R}$. Hence we will frequently identify a regular language with the finite automaton recognizing it and vice versa. For a regular language L, the minimal DFA recognizing L is the DFA with the least number of states recognizing L.

For a language $L \subseteq \Sigma^*$, we define the Myhill-Nerode equivalence relation $\equiv_L$ on Σ^* as follows: $x \equiv_L y$ if $xz \in L$ if and only if $yz \in L$ for every $z \in \Sigma^*$. The Myhill-Nerode theorem is a classical resullt that gives a necessary and sufficient condition for a language to be regular:

Theorem 2.1.1 *A language $L \subseteq \Sigma^*$ is regular if and only if the equivalence relation $\equiv_L$ has finite index. Furthermore, if L is regular, the index of $\equiv_L$ is the size of the minimal DFA recognizing L.*

For two languages L_1 and L_2, we define the *concatenation* operation as follows: $L_1 \cdot L_2 = \{w \in \Sigma^* \mid w = xy \text{ and } x \in L_1, y \in L_2\}$. For a language L, we define $L^0 = \{\epsilon\}$ and $L^{n+1} = L \cdot L^n$ for $n \in \mathbb{N}$. The *Kleene star* operation is defined as $L^* = \bigcup_{n\in\mathbb{N}} L^n$. *Kleene's theorem* gives an alternate characterization of regular languages:

Theorem 2.1.2 *A language $L \subseteq \Sigma^*$ is regular if and only if it can be constructed from the empty set and singletons by the application of a finite number of union, concatenation and Kleene star operations.*

The next classical result relies on the fact that if an n-state automaton processes a word w such that $|w| > n$, then any run of the automaton on w must contain at least one repeated state. It is called the *pumping lemma* and may be stated as follows:

Theorem 2.1.3 *Suppose $L \subseteq \Sigma^*$ is a regular language which is accepted by a n-state NFA. For any $w \in L$ such that $|w| \geq n$, there are words $x, y, z \in \Sigma^*$ such that $w = xyz$, $|y| > 1$, $|xy| \leq n$ and $xy^iz \in L$ for all $i \in \mathbb{N}^+$.*

The pumping lemma is often used to show that a language in *not* regular. For example, it is easy to use the pumping lemma to show that the language $\{0^n1^n \mid n \in \mathbb{N}\}$ is not regular.

The class of regular languages is well known to be closed under the boolean operations of union, intersection and complementation. Given a DFA $\mathcal{M} = (S, \Delta, s_0, F)$ the DFA $\mathcal{M}' = (S, \Delta, s_0, S \setminus F)$ recognizes the complement of $L(\mathcal{A})$. Also for DFA's $\mathcal{M}_1 = (S_1, \Delta_1, s_0, F_1)$ and $\mathcal{M}_2 = (S_2, \Delta_2, q_0, F_2)$, we define the *union automaton* $\mathcal{M}_1 \oplus M_2 = (S, \Delta, s_I, F)$ as follows:

1. $S = S_1 \times S_2$.
2. $s_I = (s_0, q_0)$.
3. $F = (F_1 \times S_2) \cup (S_1 \times F_2)$.
4. $\Delta((s,q),\sigma) = (\Delta_1(s,\sigma), \Delta_2(q,\sigma))$ for $s \in S_1$, $q \in S_2$ and $\sigma \in \Sigma$.

It is quite easy to show that union automaton $\mathcal{M}_1 \oplus \mathcal{M}_2$ recognizes the languages $L(\mathcal{M}_1) \cup L(\mathcal{M}_2)$. We can define the *intersection automaton* $\mathcal{M}_1 \otimes \mathcal{M}_2 = (S, \Delta, s_I, F)$ to recognize $L(\mathcal{M}_1) \cap L(\mathcal{M}_2)$ in a similar manner except that $F = F_1 \times F_2$. We refer to the automata $\mathcal{M}_1 \oplus \mathcal{M}_2$ and $\mathcal{M}_1 \otimes \mathcal{M}_2$ collectively as *product automata*.

2.1.2 Finite tree automata

In this section we provide the basic definitions and properties of tree automata. The reader is referred to [40, 23] for a detailed treatment of tree automata. We use $\prec_{\mathsf{pref}}$ to denote the prefix order on words in $\mathbb{N}^*$, i.e., for $u, v \in \mathbb{N}^*$, $u \prec_{\mathsf{pref}} v$ if $v = uw$ for some $w \in \mathbb{N}^*$. For $L \subseteq \mathbb{N}^*$, let $\mathsf{pref}(L) = \{w \in \mathbb{N}^* \mid \exists u \in L : w \prec_{\mathsf{pref}} u\}$. We say that L is *prefix-closed* if $\mathsf{pref}(L) = L$.

For an integer $k \geq 0$, we use $\mathbb{N}_k$ to denote the set $\{0, \ldots, k-1\}$. A k-ary tree t is a non-empty finite prefix-closed subset of $\mathbb{N}_k^*$. The $\prec_{\mathsf{pref}}$-maximal elements in a tree t are the the *leaves*, denoted by $\mathsf{leaves}(t)$. All other elements are *internal nodes*. The empty word ε is the *root*. The height of the tree is the maximal distance from the root to a leaf. We denote the set of all k-ary trees by $\mathcal{T}_k$.

Let Σ be a finite alphabet. A Σ-*labeled k-ary tree* is of the form (t, λ) where t is a k-ary tree and $\lambda : t \to \Sigma$ is a labeling function. We use $\mathcal{T}_k(\Sigma)$ to denote the set of all finite Σ-labeled k-ary trees. Analogous to the case of finite word automata, we define finite tree automata. Note that our definition of tree automata is slightly different from the one given in [40] but is nonetheless equivalent.

Definition 2.1.3 *Let Σ be a finite alphabet. A* deterministic (bottom-up) tree automaton (DTA) over Σ with rank k *is a tuple $\mathcal{M} = (Q, \Delta, q_0, F)$, where Q is the finite set of states, $q_0 \notin Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and*

$$\Delta : ((Q \cup \{q_0\})^k \times \Sigma \to Q$$

is the transition relation.

Intuitively the intial state q_0 may be seen as a "dummy" state such that for any node of the tree having less than k children, q_0 labels the missing children. Formally, for an unlabeled k-ary tree $t \in \mathcal{T}_k$, let $\widehat{t}$ denote the tree $t \cup \{wa \mid w \in t, a \in \mathbb{N}_k\}$.

Definition 2.1.4 *Given any Σ-labeled k-ary tree $T = (t, \lambda) \in \mathcal{T}_k(\Sigma)$, a* run *of $\mathcal{M}$ on T is a mapping $\rho : \widehat{t} \to Q$ such that*

(i) for every $w \in \mathsf{leaves}(\widehat{t})$, $\rho(w) = q_0$, and

(ii) for every $w \in t$, $\rho(w) = \Delta(\rho(wa_1), \ldots, \rho(wa_k), \lambda(w))$.

The run ρ is accepting *if $\rho(\varepsilon) \in F$.*

With $L(\mathcal{M})$ we denote the set of all $T \in \mathcal{T}_k(\Sigma)$ on which the DTA $\mathcal{M}$ has an accepting run; this is called the *tree language recognized by* $\mathcal{M}$. A set $L \subseteq \mathcal{T}_k(\Sigma)$ is called *regular* if there exists a DTA $\mathcal{M}$ over Σ with $L = L(M)$. The *size* of a DTA is the number of states it contains (excluding the state q_0). The *minimal automaton* for a regular tree language $L \subseteq \mathcal{T}_k(\Sigma)$ is the size of the smallest DTA that recognizes L. We use $\mathfrak{R}_{tree}$ denote the class of all regular k-ary tree languages.

It is easy to see that the class of regular tree languages, just like regular word languages, is closed under union and intersection. Let $\mathcal{M}_1 = (Q_1, \Delta_1, q_0, F_1)$ and $\mathcal{M}_2 = (Q_2, \Delta_2, q_0, F_2)$ be two rank k DTA over Σ. We define, using a similar definition as the product automata for word languages, the *product tree automata* $\mathcal{M}_1 \oplus \mathcal{M}_2$ and $\mathcal{M}_1 \otimes \mathcal{M}_2$. The state space of the product tree automata is $Q_1 \times Q_2$. The transition $\Delta : (Q_1 \times Q_2 \cup \{q_0\})^k \times \Sigma \to Q_1 \times Q_2$ is defined as follows. For convenience we write the state q_0 as a pair (q_0, q_0). For any states $p_1, \ldots, p_k \in Q_1 \cup \{q_0\}$ and $q_1, \ldots, q_k \in Q_2 \cup \{q_0\}$, we let

$$\Delta((p_1, q_1), \ldots, (p_k, q_k), \sigma) = (\Delta_1(p_1, \ldots, p_k, \sigma), \Delta_2(p_2, \ldots, p_k, \sigma)).$$

The set F for $\mathcal{M}_1 \oplus \mathcal{M}_2$ (resp. $\mathcal{M}_1 \otimes \mathcal{M}_2$) is defined as $Q_1 \times F_2 \cup F_1 \times Q_2$ (resp. $F_1 \times F_2$). The following lemma is easy to prove.

Lemma 2.1.4 *The product automaton $\mathcal{M}_1 \oplus \mathcal{M}_2$ recognizes $L(\mathcal{M}_1) \cup L(\mathcal{M}_2)$ and $\mathcal{M}_1 \otimes \mathcal{M}_2$ recognizes $L(\mathcal{M}_1) \cap L(\mathcal{M}_2)$.*

Similar to the case of regular word languages, we also have a *pumping lemma* for regular tree languages. However we need to introduce some terminology before formulating the

pumping lemma for regular tree languages. Fix $k \geq 0$ and let Σ be a finite alphabet. Also let $*$ be a symbol not occuring in Σ. A Σ-*context* C is a $\Sigma \cup \{*\}$ labeled k-ary tree (t, λ) where the label $*$ may only occur on exactly one leaf of t. The leaf labeled by $*$ is called a *hole*. A Σ-context is called *non-trivial* if its domain is not equal to $\{\epsilon\}$.

Given a Σ-context C and a Σ-labeled k-ary tree T, we use $C[T]$ to denote the tree obtained by plugging T into the hole of C. Similarly given two Σ-contexts C_1, C_2, the Σ-context obtained by plugging C_2 into the hole of C_1 is denoted by $C_1[C_2]$. We define C^n inductively by saying that $C^1 = C$ and $C^n = C^{n-1}[C]$. Now we are ready to state the *pumping lemma* for regular tree languages.

Lemma 2.1.5 *Suppose $L \subseteq \mathcal{T}_k(\Sigma)$ is a regular tree language recognized by some DTA $\mathcal{M}$ with n states. Then for any $T = (t, \lambda) \in L$ such that the height$(t) \geq k$, there exist a Σ-context C_1, a non-trivial Σ-context C_2 and a tree $T' \in \mathcal{T}_k(\Sigma)$ such that $T = C_1[C_2[T']]$ and $C_1[C_2^i[T']] \in L$ for every $i \in \mathbb{N}^+$.*

2.1.3 Complexity of regular languages

In chapter 1 (section 1.1.1), we briefly described the concept of state complexity of regular languages. We now give a formal definition of the state complexity of regular word languages.

Definition 2.1.5 *The* NFA (DFA) state complexity *$NSC(L)$ ($SC(L)$) of a regular word language L is the number of states of the minimal NFA (DFA) recognizing L. The* NFA (DFA) state complexity *$NSC(\mathcal{C})$ ($SC(\mathcal{C})$) of a class $\mathcal{C} \subseteq \mathfrak{R}$ of regular word languages is the maximal state complexity of languages in the class. Consider an operation $\mathsf{Op} : \mathfrak{R}^k \to \mathfrak{R}$ ($k \geq 0$). Then given languages $L_1, \ldots L_k \in \mathfrak{R}$, the* NFA (DFA) state complexity *of $\mathsf{Op}(L_1, \ldots, L_k)$ is defined as $NSC(\mathsf{Op}(L_1, \ldots, L_k))$ ($SC(\mathsf{Op}(L_1, \ldots, L_k))$).*

The above definition may be extended to the DTA in a straightforward manner i.e. the state complexity $SC(L)$ of a regular tree language L is the size of the minimal DTA recognizing L.

When we have the minimal NFA recognizing a regular language, the number of transitions in the NFA also serves as an important indicator of the complexity of the language (since the number of transitions may be exponentially more than the number of states). For a regular word language L, we use the term *transition complexity* of L to mean the the number of transitions in the minimal NFA recognizing L.

2.2 Structures

A structure $\mathcal{S}$ consists of a (possibly infinite) domain D and finitely many atomic operations $f_1, \ldots, f_m$, relations $R_1, \ldots, R_n$ and constants $c_1, \ldots, c_\ell$ on the set D. We denote this by

$$\mathcal{S} = (D; f_1, \ldots, f_m, R_1, \ldots, R_n, c_1, \ldots, c_\ell).$$

The sequence of symbols $f_1, \ldots, f_m, R_1, \ldots, R_n, c_1, \ldots, c_\ell$ is called the signature of $\mathcal{S}$ which we denote by $\mathsf{Sig}(\mathcal{S})$. In this thesis we only consider structures whose operations and relations have arity 2. Furthermore all structures we consider in this thesis will have countable domains.

We denote the set of all atomic operations and the set of all atomic relations of $\mathcal{S}$ by $\mathsf{Op}(\mathcal{S})$ and $\mathsf{Rel}(\mathcal{S})$, respectively. The semantics of an operation $f \in \mathsf{Op}(\mathcal{S})$ are that f takes two inputs x_1, x_2 from the domain D and outputs another element $x_3 \in D$ i.e. $f(x_1, x_2) = x_3$. Intuitively, a relation $R \in \mathsf{Rel}(\mathcal{S})$ may be viewed as a predicate which takes two inputs $x_1, x_2 \in D$ i.e. $R(x_1, x_2)$ is either true or false.

A *formula* over $\mathsf{Sig}(\mathcal{S})$ is a formula which uses the relation symbols from $\mathsf{Sig}(\mathcal{S})$ as non-logical symbols. A *sentence* over $\mathsf{Sig}(\mathcal{S})$ is a formula which has no free variables i.e. every variable in the formula is bound by either a $\forall$ quantifier or a $\exists$ quantifier. A sentence is said to be *existential* if all the variables are bound by the existential quantifier ($\exists$).

In *first-order logic*, all variables in a formula may only range over the elements of the domain of $\mathcal{S}$. In this thesis we will only consider formulae from first-order logic. Given a sentence φ over $\mathsf{Sig}(\mathcal{S})$, we write $\mathcal{S} \models \varphi$ if φ is true in $\mathcal{S}$. The *theory* of $\mathcal{S}$, denoted by $\mathsf{Th}(\mathcal{S})$ is the set of all first-order sentences over $\mathsf{Sig}(\mathcal{S})$ that are true in $\mathcal{S}$, that is

$$\mathsf{Th}(\mathcal{S}) = \{\varphi \mid \mathcal{S} \models \varphi \text{ and } \varphi \text{ is a first order sentence over } \mathsf{Sig}(\mathcal{S})\}.$$

The *existential theory* of $\mathcal{S}$, denoted by $\mathsf{Th}_\exists(\mathcal{S})$ is the set of all existential sentences that are true in $\mathcal{S}$, that is

$$\mathsf{Th}_\exists(\mathcal{S}) = \{\varphi \mid \mathcal{S} \models \varphi \text{ and } \varphi \text{ is an existential sentence over } \mathsf{Sig}(\mathcal{S})\}.$$

2.3 Infinite games on graphs

For background on games played on graphs, see e.g. [41]. A *game* is a tuple $\mathcal{G} = (V_0, V_1, E, \text{Win})$ where $G = (V_0 \cup V_1, E)$ forms a finite directed graph (called the underlying graph of $\mathcal{G}$), $V_0 \cap V_1 = \emptyset$ and the set $\text{Win} \subseteq (V_0 \cup V_1)^\omega$. Nodes in the set V_0 are said to be *0-nodes* and nodes in the set V_1 are said to be *1-nodes*. We use V to denote $V_0 \cup V_1$ and $E(u)$ to denote the set $\{v \mid (u, v) \in E\}$. The game is played by Player 0 and Player 1 in rounds. Initially, a token is placed on some *initial node* $v \in V$. In each round, if the token is placed on a node $u \in V_\sigma$, where $\sigma \in \{0, 1\}$, then Player σ selects a node $u' \in E(u)$ and moves the token from u to u'. The play continues indefinitely unless the token reaches a node u where $E(u) = \emptyset$. Thus, a *play* starting from u is a (possibly infinite) sequence of nodes $\pi = v_0 v_1 \ldots$ such that $v_0 = u$ and for every $i \geq 0$, $v_{i+1} \in E(v_i)$. We use $\text{Plays}(G)$ to denote the set of all plays starting from any node in V. The *winning condition* of $\mathcal{G}$, denoted by Win, is a subset of $\text{Plays}(G)$ and Player 0 *wins* a play $\pi \in \text{Plays}(G)$ if $\pi \in \text{Win}$ and Player 1 wins π otherwise. We use $\text{Occ}(\pi)$ to denote the set of nodes that appear in π and $\text{Inf}(\pi)$ to denote the set of nodes that appear infinitely often in π.

A *reachability game* is a game $\mathcal{G} = (V_0, V_1, E, W_{\text{reach}})$. The winning condition W_{reach} is determined by a set of *target nodes* $T \subseteq V$ such that $W_{\text{reach}} = \{\pi \in \text{Plays}(G) \mid \text{Occ}(\pi) \cap T \neq \emptyset\}$. Hence, for convenience, we denote the reachability $\mathcal{G}$ by (V_0, V_1, E, T).

A *Büchi game* is a game $\mathcal{G} = (V_0, V_1, E, W_{\text{buchi}})$. As in the case of reachability games, we specify a set of target nodes $T \subseteq V$. Then the Büchi winning condition can be expressed as follows: $W_{\text{buchi}} = \{\pi \in \text{Plays}(G) \mid \text{Inf}(\pi) \cap T \neq \emptyset\}$. For convenience, we also denote a Büchi game by (V_0, V_1, E, T). It will be clear from the context whether reachability or Büchi games are considered.

A *parity game* is a game $\mathcal{G} = (V_0, V_1, E, W_{\text{parity}})$ along with a *priority function* $\rho : V \to \mathbb{N}$. The winning condition is expressed as follows: $W_{\text{parity}} = \{\pi \in \text{Plays}(G) \mid \min\{\rho(v) \mid v \in \text{Inf}(\pi)\} \text{ is even}\}$. We use the tuple (V_0, V_1, E, ρ) to denote a parity game.

When playing a game, the players use *strategies* to determine the next move from the previous moves. Formally, a *strategy* for Player σ (or a *σ-strategy*), where $\sigma \in \{0, 1\}$, is a partial function $f_\sigma : V^* V_\sigma \to V$ such that if $f_\sigma(v_1 v_2 \ldots v_i) = w$ then $(v_i, w) \in E$ (here $V^* V_\sigma$ denotes the set of all finite paths in the graph G with the last node in V_σ). A strategy f_σ for Player σ is called *memoryless* if $f_\sigma(v_1 v_2 \ldots v_i) = f_\sigma(v_i)$ for all $(v_1 v_2 \ldots v_i) \in V^* V_\sigma$.

A play $\pi = v_0 v_1 \ldots$ is *consistent* with f_σ if $v_{i+1} = f_\sigma(v_0 v_1 \ldots v_i)$ whenever $v_i \in V_\sigma$ $(i \geq 0)$.

A strategy f_σ is *winning* for Player σ on v if Player σ wins all plays starting from v consistent with f_σ. If Player σ has a winning strategy on u, we say Player σ *wins* the game on u, or u is a *winning position* for Player σ. The *σ-winning region*, denoted by W_σ, is the set of all winning positions for Player σ. Note that $W_0 \cap W_1 = \emptyset$. By *solving a game*, we mean to provide an algorithm that takes as input a game $\mathcal{G}$, and outputs all nodes in W_0.

For the sake of simplicity, we assume $E(u) \neq \emptyset$ for all $u \in V$ in any game $\mathcal{G}$. This can be achieved by performing the following whenever $E(u) = \emptyset$: we add two extra vertices u_1, u_2 such that $E(u) = u_1$, $E(u_1) = u_2$ and $E(u_2) = u_1$. In the case of reachability and Büchi games we declare $u_1, u_2 \notin T$ and for parity games we declare u_1, u_2 to have odd priorities. Note that this does not change the winning region of Player 0 for any of the winning conditions described earlier. Hence we may assume that Plays$(G) \subseteq V^\omega$.

A game *enjoys determinacy* if $W_0 \cup W_1 = V$. A well known result by Martin states that all Borel games enjoy determinacy [62]. The next theorem follows from this result since reachability, Büchi and parity games are special cases of Borel games:

Theorem 2.3.1 [62][41] *Reachability, Büchi and parity games enjoy determinacy.*

A game is said to enjoy *memoryless determinacy* if it enjoys determinacy and there is a memoryless winning strategy for Player σ ($\sigma \in \{0, 1\}$) starting from each node in W_σ. In the subsequent sections, we outline proofs of the memoryless determinacy of reachability, Büchi and parity games.

Reachability games.

In this section we outline the proof of the memoryless determinacy of reachability games. The proof is constructive in the sense that it gives an algorithm to solve reachability games and also a memoryless winning strategy for each player. Consider a reachability game $\mathcal{G} = (V_0, V_1, E, T)$. We use G to denote the underlying graph $(V_0 \cup V_1, E)$ and let $m = |E|$ and $n = |V|$.

Theorem 2.3.2 [41] *Reachability games enjoy memoryless determinacy. Furthermore there exists an $O(m + n)$ algorithm to solve reachability games.*

Proof We now describe the classical algorithm to solve reachability games which also

provides a proof of the memoryless determinacy of reachability games. For $Y \subseteq V$, let

$$\mathrm{Pre}(Y) = \{v \in V_0 \mid \exists u : (v,u) \in E \wedge u \in Y\} \cup \{v \in V_1 \mid \forall u : (v,u) \in E \rightarrow u \in Y\}.$$

The algorithm to solve $\mathcal{G}$ computes a sequence of sets $T_0, T_1, \ldots$ where $T_0 = T$, and for $i > 0$, $T_i = \mathrm{Pre}(T_{i-1}) \cup T_{i-1}$. Since the graph is finite, we have $T_s = T_{s+1}$ for some $s \in \mathbb{N}$. We say that a node v has *rank* i ($i \geq 0$) if $v \in T_i \setminus T_{i-1}$. If $u \notin T_s$, we say that it has infinite rank. We claim that a node has finite rank if and only if it is winning for Player 0.

We may prove by induction on the rank that every vertex with a finite rank is winning for Player 0. Consider a node v with rank i for $i > 0$. If $v \in V_1$, then by the algorithm all u such that $(v,u) \in E$ must have rank strictly lesser than i. Also if $v \in V_0$, then there must exist a x such that $(v,x) \in E$ and x has rank stricly lesser than i. By the inductive assumption, there exists a memoryless winning strategy f for Player 0 on x. Hence we may define the following memoryless strategy $g : V_0 \rightarrow V$:

$$g(u) = \begin{cases} x & \text{if } u = v \\ f(u) & \text{otherwise} \end{cases}$$

The strategy g is clearly a winning strategy for Player 0 on v. Hence we have $T_s \subseteq W_0$.

Now suppose that $v \in W_0$ i.e. Player 0 has a winning strategy f such that all plays starting from v reach T. Let $n \in \mathbb{N}$ be the number of nodes of the graph. Then it must be the case that any play π consistent with f must visit a target node in at most n steps since otherwise π would never visit a target node (contradicting the assumption that f is a winning strategy). Hence we must have $v \in T_n$ and therefore v has a finite rank. The above arguments prove that $W_0 = \{v \mid v \text{ has finite rank}\} = T_s$.

We may implement the algorithm for solving reachability games to run in $O(m+n)$. We associate with each node $v \in V$, an integer labels which we denote by $r(v)$. We first compute the out-degrees of every $v \in V_1$ and initialize $r(v)$ to be the out-degree of v. For every $v \in V_0$, we initialize $r(v) = 1$. This can be accomplized in $O(m+n)$.

Then we conduct a reverse breadth first search strating from the set of target nodes T. First we set $r(t) = 0$ for every $t \in T$. For every $(u,t) \in E$ such that $u \in V_0$ and $t \in T$, we set $r(u) = 0$. Also for every $(u,t) \in E$ such that $u \in V_1$ and $t \in T$ we decrement the value of $r(u)$ by 1 (while ensuring that $r(u)$ remains nonnegative). Note that we process each edge exactly once in order to update the integer labels. Then we let $T_1 = \{v \mid (v,t) \in E, r(v) = 0 \text{ and } t \in T\}$

and repeat the reverse breadth first search starting from T_1. We continue this process until no more nodes have their can have their integer labels set to 0. At the end of this process we have $T_s = \{v \in V \mid r(v) = 0\}$. Since each edge is processed exactly once, the reverse breadth first search described above runs in $O(m + n)$. Therefore the overall running time of the algorithm is $O(m + n)$. ∎

Note that the above algorithm for reachability games gives us a memoryless winning strategy for Player 0. We refer to this algorithm as the *reach algorithm*. For any set $X \subseteq V$, we use $\mathsf{Reach}_\sigma(X, G)$ to denote the σ-winning region for the reachability game (V_0, V_1, E, X). In other words, from any node in $\mathsf{Reach}_\sigma(X, G)$, Player σ has a strategy that forces any play starting from this node to visit X. Hence in the above algorithm we have $T_s = \mathsf{Reach}_0(T, G)$.

Büchi games.

Similar to the case of reachability games, we now provide a constructive proof of the fact that Büchi games enjoy memoryless determinacy. Consider a Büchi game $\mathcal{G} = (V_0, V_1, E, T)$. We use G to denote the underlying graph $(V_0 \cup V_1, E)$ and let $m = |E|$ and $n = |V|$.

Theorem 2.3.3 [41] *Büchi games enjoy memoryless determinacy. Furthermore there exists an $O(n \cdot (m + n))$ algorithm to solve Büchi games.*

Proof The algorithm to solve $\mathcal{G}$ computes the sequences of sets $T_0, T_1, \ldots$, $R_0, R_1, \ldots$ and $U_0, U_1, \ldots$ as follows: Let $T_0 = T$. Suppose T_i is defined for $i \geq 0$. Set $R_i = \mathsf{Reach}_0(T_i, G)$ and $U_i = V \setminus R_i$. Set $T_{i+1} = T_i \setminus \mathsf{Reach}_1(U_i, G)$. Hence we have $T_0 \supseteq T_1 \supseteq T_2 \supseteq \ldots$. Since the game is played on a finite directed graph, we must have $T_s = T_{s+1}$ for some $s \in \mathbb{N}$. We claim that a node v is a winning position for Player 0 if and only if $v \in \mathsf{Reach}_0(T_s, G)$.

Consider a node $v \in \mathsf{Reach}_0(T_s, G)$. By the algorithm, we have $\mathsf{Reach}_1(U_s, G) \cap T_s = \emptyset$ and hence Player 0 must have a memoryless strategy f on v (corresponding to the strategy given by the $\mathsf{Reach}_0(T_s, G)$ algorithm) such that any play consistent with f never leaves $\mathsf{Reach}_0(T_s, G)$. The memoryless strategy f is clearly a winning strategy for Player 0 since any play consistent with it visits nodes in T_s infintely often. Hence we have $v \in W_0$ and $\mathsf{Reach}_0(T_s, G) \subseteq W_0$.

Now consider a node $v \in V \setminus \mathsf{Reach}_0(T_s, G)$. We show that Player 1 has a memoryless winning strategy on v. In order to do this we observe the following for every $i \in \mathbb{N}$:

For every $v \in U_i$, Player 1 has a memoryless winning strategy g_i on v such that any play consistent with g_i never visits any nodes from $\mathsf{Reach}_0(T_i, G)$

We may prove the correctness of the above statement by induction. For $i = 0$, it is clear that for any $v \in U_0$ Player 1 has a memoryless winning strategy g_0 such that any play consistent with it never visits any node in $\mathsf{Reach}_0(T_0, G)$ (by definition of U_0). Now consider $v \in U_{i+1}$. By definition of U_{i+1}, Player 1 has a memoryless strategy f which avoids all nodes in $\mathsf{Reach}_0(T_{i+1}, G)$.

Also by the inductive assumption for any node $u \in U_i$, Player 1 has a memoryless winning strategy g_i on u such that any play consistent with it avoids $\mathsf{Reach}_0(T_i, G)$. Furthermore from any node $u \in T \setminus T_{i+1}$, Player 1 has a memoryless strategy f' to force any play into U_i in one move (corresponding to the strategy given by $Reach_1(U_i, G)$). We now define the memoryless strategy $g_{i+1} : V_1 \to V$ for Player 1 as follows:

$$g_{i+1}(v) = \begin{cases} f(v) & \text{if } v \in U_{i+1} \setminus U_i \\ f'(v) & \text{if } v \in T \\ g_i(v) & \text{otherwise} \end{cases}$$

We now claim that g_{i+1} is a winning strategy for Player 1 on $v \in U_{i+1}$. We only need to consider the case when $v \in U_{i+1} \setminus U_i$. Indeed consider any play π starting from v consistent with g_{i+1}. There are two cases: (1) no target node is visited by π or (2) some target node is visited by π. In the first case, π is clearly a winning play for Player 1. In the second case let $\pi = vu_1u_2 \dots u_k t x \dots$ where $t \in T$ is the first target node visited by π. Note that t must be in $T \setminus T_{i+1}$ since $v \in U_{i+1} \setminus U_i$ and by the definition of g_{i+1}, the play $vu_1 \dots u_k$ is consistent with f which avoids all nodes in $\mathsf{Reach}_0(T_{i+1}, G)$. Therefore $x \in U_i$ and by the definition of g_{i+1}, the play from x onwards is consistent with g_i (by the inductive assumption that g_i avoids all nodes in $Reach_0(T_i, G)$). Since g_i is a winning strategy for Player 1, π must be winning for Player 1. This proves the correctness of the statement and it immediately follows that any node $v \in V \setminus \mathsf{Reach}_0(T_s, G)$ is a winning node for Player 1. Therefore $W_0 \subseteq \mathsf{Reach}_0(T_s, G)$ and hence $W_0 = \mathsf{Reach}_0(T_s, G)$ as required.

Since the *reach algorithm* has a running time of $O(m + n)$ and the algorithm for solving Büchi games performs at most n iterations, the running time of the algorithm to solve Büchi games is $O(n \cdot (n + m))$. ∎

Parity games.

Parity games also enjoy memoryless determinacy but unlike reachability and Büchi games, no polynomial time algorithm is known to solve them. A constructive proof of the memoryless determinacy of parity games can be found in in [41](Ch. 6). The proof given in [41] is constructive but quite involved. Here we provide an alternative proof of the memoryless determinacy of parity games which is much simpler. This proof is due to Khoussainov [55] and has not been written down before to the best of our knowledge.

Theorem 2.3.4 [41] *If Player 0 wins a parity game $\mathcal{G} = (V_0, V_1, E, \rho)$ starting from $v \in V$, then Player 0 has a memoryless strategy to do so.*

Before we begin the proof of the above theorem, we need to introduce some terminology. A parity game $\mathcal{N} = (V_0, V_1, E, \rho)$ is called a *choiceless* game for Player σ ($\sigma \in \{0, 1\}$) if $|E(v)| = 1$ for every $v \in V_\sigma$. It is easy to see that a choiceless game for Player σ defines a memoryless strategy for Player σ.

Consider a node $x \in V_0$ such that $|E(x)| > 1$ and let $S \subset E(x)$ be an arbitrary non-empty subset of $E(x)$. We define the two parity games $\mathcal{G}(x, S), \mathcal{G}(x, \overline{S})$ as follows: $\mathcal{G}(x, S) = (V_0, V_1, E_1, \rho)$ where $E_1 = E \setminus \{(x, u) \in E \mid u \notin S\}$. The game $\mathcal{G}(x, \overline{S}) = (V_0, V_1, E_2, \rho)$ is defined analogously and $E_2 = E \setminus \{(x, u) \in E \mid u \in S\}$.

Lemma 2.3.5 *Consider a parity game $\mathcal{G} = (V_0, V_1, E, \rho)$ and let $x \in V_0$ be such that $E(x) > 1$. Then for any node $v \in V$, Player 1 wins $\mathcal{G}$ from v if and only if Player 1 wins both $\mathcal{G}(x, S)$ and $\mathcal{G}(x, \overline{S})$ from v for some $S \subset E(x)$.*

Proof It is easy to see that if Player 1 wins the game starting from v, then it wins both $\mathcal{G}(x, S)$ and $\mathcal{G}(x, \overline{S})$ from v. Now suppose that Player 1 has winning strategies f_0, f_1 in $\mathcal{G}(x, S)$ and $\mathcal{G}(x, \overline{S})$ respectively.

For a play $\pi = v_0 v_1 \dots v_r$ in $\mathcal{G}$, we use $\tau_0(\pi)$ to denote the portion of π that only uses edges in $\mathcal{G}(x, S)$. Similarly we use $\tau_1(\pi)$ to denote the portion of π that only uses edges in $\mathcal{G}(x, \overline{S})$. Also we use $i_x \in \{0, \dots, r\}$ to denote the last occurrence of x in π (if π visits x at least once). We now define a strategy $f : V^* V_1 \to V$ for Player 1 in $\mathcal{G}$ as follows:

$$f(v_0 \dots v_k v_{k+1}) = \begin{cases} f_0(v_0 \dots v_k v_{k+1}) & \text{if } x \text{ does not occur in } v_0 v_1 \dots v_k \\ f_0(\tau_0(v_0 \dots v_{k+1})) & \text{if } x \text{ occurs in } v_0 \dots v_k \text{ and } v_{i_x+1} \in S \\ f_1(\tau_1(v_0 \dots v_{k+1})) & \text{if } x \text{ occurs in } v_0 \dots v_k \text{ and } v_{i_x+1} \notin S \end{cases}$$

Consider a play π consistent with f. The following cases are possible for π:

1. x occurs finitely many times in π: In this case we must have that π always uses edges of $\mathcal{G}(x, S)$ after the last occurrence of x. Hence by the definition of f, the portion of π after the last occurrence of x is consistent with f_0 which is a winning strategy for Player 1. Therefore π must be winning for Player 1.

2. x occurs infinitely often in π: In this case we may form two plays π_0, π_1 where π_i ($i \in \{0, 1\}$) is formed by joining those portions of π where f_i is followed. Suppose that one of the plays π_i is finite. By the definition of f, we must have that the portion of π after the occurrence of the last node of π_i is consistent with f_{1-i} and hence winning for Player 1.

   If π_0, π_1 are both infinite then $\min\{\rho(u) \mid u \in \mathrm{Inf}(\pi_0)\}$ and $\min\{\rho(u) \mid u \in \mathrm{Inf}(\pi_1)\}$ are both odd (since f_0, f_1 are winning for Player 1 in $\mathcal{G}(x, S)$ and $\mathcal{G}(x, \overline{S})$ respectively). Therefore $\min\{\rho(u) \mid u \in \mathrm{Inf}(\pi)\}$ is odd and π is winning for Player 1.

The above cases show that f is a winning strategy for Player 1 in $\mathcal{G}$. ∎

Now we are ready to prove Theorem 2.3.4.

Proof *(Proof of Theorem 2.3.4)* Let $V_0 = \{x_0, \ldots, x_k\}$. We now obtain the games $\mathcal{G}(x_0, S_0)$, $\mathcal{G}(x_0, \overline{S_0}), \ldots, \mathcal{G}(x_k, S_k), \mathcal{G}(x_k, \overline{S_k})$ where each S_i is arbitrary non-empty subsets of $E(x_i)$ (for $i \in \{0, \ldots, k\}$). By lemma 2.3.5, we have that for Player 1 to lose $\mathcal{G}$ from v, she must lose at least one of $\mathcal{G}(x_0, S_0), \mathcal{G}(x_0, \overline{S_0}), \ldots, \mathcal{G}(x_k, S_k), \mathcal{G}(x_k, \overline{S_k})$.

We repeat this process for each of the games $\mathcal{G}(x_i, S_i), \mathcal{G}(x_i, \overline{S_i})$ ($i \in \{0, \ldots, k\}$) and continue iterating this process until we get choiceless games $\mathcal{N}_0, \ldots, \mathcal{N}_\ell$. By lemma 2.3.5 it must be the case that for Player 1 to lose the game from v, she must lose at least one of $\mathcal{N}_0, \ldots, \mathcal{N}_\ell$. Let this choiceless game be $\mathcal{N}_j$ for some $j \in \{0, \ldots, \ell\}$. Recall that choiceless games define memoryless strategies for Player 0 and hence $\mathcal{N}_j$ defines a memoryless winning strategy for Player 0 from v. ∎

We would like to point out that Theorem 2.3.4 implies that the winning region problem for parity games is in NP ∩ co-NP. This is because a non-deterministic Turing machine guesses a memoryless strategy for Player i ($i \in \{0, 1\}$) and then verifies whether this strategy is winning for Player i or not (which can be done in polynomial time). In fact Jurdziński [53] has strengthened this result by showing that the winning region problem for parity

games is in UP ∩ co-UP. Here UP is the class of languages recognized by non-deterministic *unambiguous polynomial-time* Turing machines i.e. Turing machines which have at most one accepting computation of length polynomially bounded in the size of the input. However it is an open problem whether there exists a polynomial time algorithm to determine the winner of a parity game.

Chapter 3

Complexity of regular languages

In this chapter we approach the study of complexity of regular languages from two directions. First we study regular word languages and analyze the relationship between DFA and NFA-state complexity. Then we study the complexity of a natural class of regular languages i.e. finite word and tree languages.

3.1 Complexity of determinization and complementation of NFA's

In this section we focus on regular word languages. We investigate the increase in complexity which occurs during the following transformations: (1) determinization i.e. transforming a NFA to a DFA (section 3.1.1) and (2) computing a NFA $\mathcal{A}'$ which recognizes the complement of a given NFA $\mathcal{A}$ (section 3.1.2). Throughout our analysis, we pay particular attention to the transition complexity of the languages we construct in addition to the state complexity.

3.1.1 State explosion in determinization

The subset construction is one of the fundamental constructions in automata theory that converts non-deterministic finite automata into equivalent deterministic automata. Under the subset construction, the states of the constructed DFA are subsets of the underlying NFA. Therefore, if the underlying NFA has n states the then resulting equivalent DFA has at most 2^n states. Hence, the cost of determinization is an exponential explosion in the number of states [78]. In [66] it was shown that this blow-up in the number of states is

sharp. This sharpness result implies hardness for the complementation problem as well. Namely, for any n there exists an n state NFA recognizing a language L such that 2^n number of states are needed for a DFA to recognize the complement of the language L.

A natural question is to ask whether it is possible to fill in the exponential gap between n and 2^n in the determinization process of NFA to DFA i.e. given n and m such that $n \leq m \leq 2^n$, does there exist a regular language whose NFA-state complexity is n and its DFA-state complexity is m? In [50] for every n and m such that $n \leq m \leq 2^n$ a regular language L is constructed such that its NFA state complexity is n and whose DFA-state complexity is m. However, these precise bounds are obtained in the expense of increasing the alphabet size exponentially on n. The authors of [50] pose the problem if the sizes of the alphabets can be controlled. For instance, can the sizes of alphabets be dependent on n linearly or be of a fixed size. In [51], the authors prove that for for every m, n such that $n \leq m \leq 2^n$, there exists a n state NFA whose DFA state complexity is m for a fixed four letter alphabet. However the n state NFA constructed in [51] have $O(n^2)$ transitions in the worst case.

In this section, we investigate this problem and provide asymptotic solutions. The languages we construct are over either binary alphabets or alphabets that depend on n linearly. These languages exhibit the same behavior as the languages in [51] but the bounds on the number of states are not sharp and the size of the alphabet varies linearly with n. However, the n-state NFA's constructed by us have asymptotically fewer transitions than the NFA constructed by the authors of [51] in the worst case.

Linear state explosion

First we construct regular languages such that the state explosion that occurs in the determinization process is linear in the number of states of the input NFA. Let $\Sigma = \{0, 1, \ldots, k-1\}$ be an alphabet of $k \geq 2$ symbols. We define the following language:

$$L_{k,m} = \{ux \mid \ x \in \sigma^+, \sigma \in \Sigma, u \in \Sigma^* \text{ and } |u| \equiv (m-1) \bmod m\}.$$

The NFA $\mathcal{A}_{k,m}$ recognizing $L_{k,m}$ has $m + k$ states denoted by $\{s_0, s_1, \ldots, s_{m+k-1}\}$. When the automaton is in state s_{m-1}, it non-deterministically guesses the form of the remaining word to be either σ^+ or $u\sigma^+$ ($|u| \equiv m-1 \pmod m$ and $\sigma \in \Sigma$). Formally the NFA $\mathcal{A}_{k,m} = \langle S, \Sigma, \delta, S_I, F \rangle$ is defined as follows:

1. $S = \{s_0, s_1, \ldots, s_{m+k-1}\}$.

2. $S_I = \{s_0\}$ and $F = \{s_m, s_{m+2}, ..., s_{m+k-1}\}$.

3. $\delta(s_i, \sigma) = \begin{cases} \{s_{i+1}\} & \text{if } i < m-1,\ \sigma \in \Sigma \\ \{s_0, s_{m+\sigma}\} & \text{if } i = m-1,\ \sigma \in \Sigma \\ \{s_i\} & \text{if } i \geq m,\ \sigma = i - m \end{cases}$

The NFA $\mathcal{A}_{2,4}$ is shown in Figure 3.1. It is not hard to see that $\mathcal{A}_{k,m}$ has $m + 2k$ transitions. We would now like to show that $\mathcal{A}_{k,m}$ is indeed the minimal NFA accepting $L_{k,m}$.

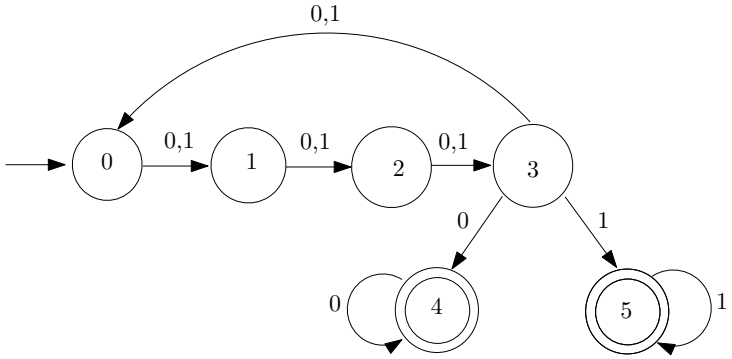


Figure 3.1: The minimal NFA recognizing the language $L_{2,4}$.

Lemma 3.1.1 *NFA $\mathcal{A}_{k,m}$ with $m + k$ states is a minimal NFA accepting $L_{k,m}$.*

Proof Let NFA $C = \langle S', \Sigma, \delta', S'_I, F' \rangle$ be a minimal NFA accepting $L_{k,m}$. It is sufficient to show that C has at least $m + k$ states. Consider a word $w \in \Sigma^*$ of length $m - 1$. Then for any $\sigma \in \Sigma$, the word $w \cdot \sigma \cdot \sigma \in L_{k,m}$, and there is an accepting run for C on $w \cdot \sigma \cdot \sigma$. Let $S'(w \cdot \sigma) = \{s \in S' \mid s \in \delta'^{+}(S'_I, w \cdot \sigma) \wedge \delta'^{+}(s, \sigma) \cap F' \neq \emptyset\}$. Assume for the sake of contradiction that $S'(w \cdot \sigma) \cap S'(w \cdot \alpha) \neq \emptyset$ where $\sigma \neq \alpha$ and $\sigma, \alpha \in \Sigma$. Then the word $w \cdot \sigma \cdot \alpha \notin L_{k,m}$ will be accepted by C, and we have reached a contradiction. Since there are k symbols in the alphabet Σ, the NFA C must have at least k states.

Let $p_0, p_1, \ldots, p_m$ be the accepting run of C on $w \cdot \sigma$ for $\sigma \in \Sigma$. Since no word of length less than m is in the language, the state p_m is an accepting state and states $p_0, p_1, \ldots, p_{m-1}$ are non-accepting states. Now suppose there exist p_i and p_j with $i < j \leq m$ such that $p_i = p_j$ and there is a cycle of length i where $i < m$. Then there exists a word u whose length is less than m and which takes the automaton C to the accepting state p_m without running

through this cycle. This is a contradiction since no word of length less than m is in $L_{k,m}$. Hence we must have that $p_0, p_1, \ldots, p_m$ are all distinct states.

Suppose there exists p_i $(i < m)$ such that $p_i \in S'(w \cdot \alpha)$ for some $\alpha \neq \sigma$. Then the word $w \cdot \alpha \cdot \sigma^{m-i}$ will be accepted by C. However $w \cdot \alpha \cdot \sigma^{m-i}$ is not in $L_{k,m}$ and hence the states $\{p_0, p_1, \ldots, p_m\} \cap S'(w \cdot \alpha) = \emptyset$ for any $\alpha \in \Sigma \setminus \{\sigma\}$. Therefore C must have another m states and hence it has at least $m + k$ states. ∎

The DFA recognizing $L_{k,m}$, upon reading a word w counts the lengths of the prefixes of w modulo m. Once the length equals $m - 1$ modulo m the automaton starts verifying that the rest of the string is from σ^+ for some $\sigma \in \Sigma$. Formally, the DFA $\mathcal{B}_{k,m} = \langle S', \Sigma, \delta', s'_I, F' \rangle$ accepts $L_{k,m}$ with $(k + 1)m + (1 - k)$ states.

1. $S = \{s'_0, s'_1, \ldots, s'_{k-1}\} \cup \{s'_{0,1}, \ldots, s'_{0,k-1}\} \cup \ldots \cup \{s'_{k-1,1}, \ldots, s'_{k-1,k-1}\} \cup \{s'_F\}$.
2. $s'_I = s'_0$
3. $F' = \{s'_F\} \cup \{s'_{i,j} \mid i \leq k - 1 \text{ and } 1 \leq j \leq k - 1\}$.
4. For $\sigma \in \Sigma$, we have the following transitions:

$$\delta'(s'_i, \sigma) = \begin{cases} s'_{i+1} & \text{if } i < m - 1 \\ s'_{\sigma,1} & \text{if } i = m - 1 \end{cases}$$

$$\delta'(s'_{i,j}, \sigma) = \begin{cases} s'_{i,j+1} & \text{if } \sigma = i \text{ and } 1 \leq j < k - 1 \\ s'_j & \text{if } \sigma \neq i \text{ and } i \leq j < k - 1 \\ s'_F & \text{if } j = k - 1 \end{cases}$$

The automaton $\mathcal{B}_{2,4}$ is shown in Figure 3.2.

The following lemma shows that the DFA $\mathcal{B}_{k,m}$ described above is minimal.

Lemma 3.1.2 *The minimal DFA recognizing $L_{k,m}$ has exactly $(k + 1)m + (1 - k)$ states.*

Proof For the proof we use Myhill-Nerode theorem and count the number of $\equiv_{L_{k,m}}$ equivalence classes. Consider a word $x \in \Sigma^*$, we can write it in the form $x = u \cdot w$ where $u, w \in \Sigma^*$ and $|u| \equiv (m - 1) \bmod m$ and $1 \leq |w| \leq m$. There are two cases for the word w:

Case 1: $w \in \sigma^i$ where $\sigma \in \Sigma$ and $1 \leq i \leq m$. For this case we want to show that the number of $\equiv_{L_{k,m}}$ equivalence classes is $k \cdot (m - 1) + 1$. To show this we distinguish the following two possibilities for $w = \sigma^i$:

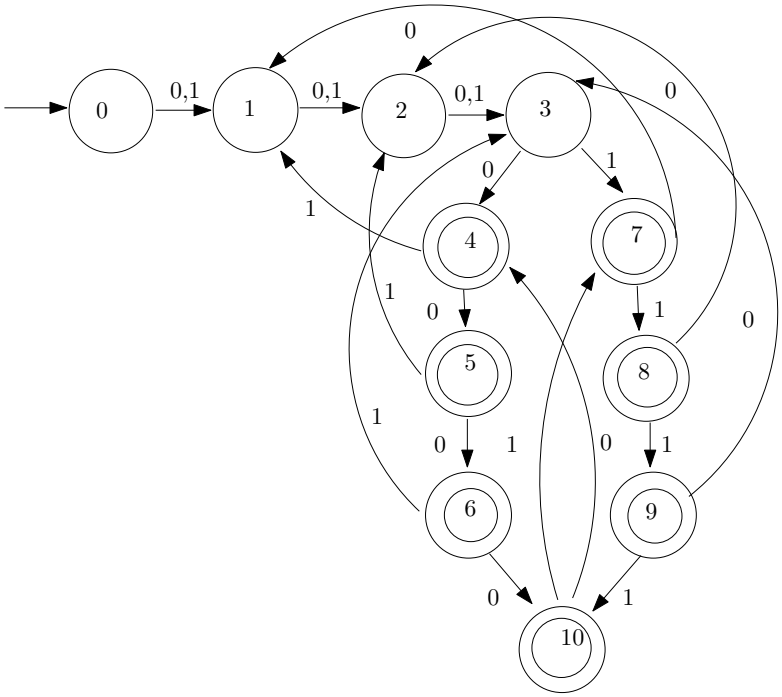


Figure 3.2: The minimal DFA recognizing the language $L_{2,4}$.

1. $|w| < m$: Consider any other word x' such that x' is of the form $u' \cdot w'$ where $|u'| = m-1$ modulo m and w' is of the form α^j with $\alpha \in \Sigma$ and $1 \leq j \leq m$. Then either $|w| = |w'|$ or $|w| \neq |w'|$. First we consider the case when $|w| = |w'|$. In this case it must be that $\sigma \neq \alpha$. It is not hard to see that for $z = \sigma^{m-|w|}$ we have $x \cdot z \in L_{k,m}$. However, $x' \cdot z \notin L_{k,m}$ because $|\alpha^i \sigma^{m-i}| = m$ and $\alpha \neq \sigma$. Hence, $x \not\equiv_{L_{k,m}} x'$. Now we consider the case when $|w| \neq |w'|$. Without loss of generality, we may assume $|w| > |w'|$. Next consider $z = \beta^{m-|w|+1}$, where $\beta \in \Sigma$ with $\beta \neq \alpha$. For this z we have $x \cdot z \in L_{k,m}$ because it is of the form $u\sigma^i\beta^{m-i}\beta$ and $|u\sigma^i\beta^{m-i}| = m-1$ modulo m. However, $x' \cdot z \notin L_{k,m}$. Thus, this possibility proves that there are exactly $k \cdot (m-1)$ number of $\equiv_{L_{k,m}}$ equivalence classes represented by the words of the form $x = u \cdot w$ where $|u| \equiv (m-1) \bmod m$ and $w = \sigma^i$ with $\sigma \in \Sigma$ and $1 \leq i \leq m$.

2. $|w| = m$: Consider any word x' of the form $x' = u' \cdot w'$, where $|u'| = m-1$ modulo m. It is not hard to see that $x \equiv_{L_{k,m}} x'$ since for all $z \in \Sigma^*$ we have $x \cdot z \in L_{k,m} \iff x' \in L_{k,m}$. Now we want to show that x is not $\equiv_{L_{k,m}}$ equivalent to any word y of the form $y = u_0 \cdot \alpha^i$ where $\alpha \in \Sigma$, $1 \leq i \leq m$ and $|u_0| \equiv (m-1) \bmod m$. Take $\beta \in \Sigma$ such that $\beta \neq \alpha$. Then it

is clear that $x \cdot \beta \in L_{k,m}$, but $y \cdot \beta \notin L_{k,m}$.

Thus, *Case 1* proves that there are $k \cdot (m-1) + 1$ equivalence $\equiv_{L_{k,m}}$-classes.

Case 2: Assume that w is not of the form σ^i for $\sigma \in \Sigma$ and $1 \leq i \leq m-1$). We want to show that there are m number of $\equiv_{L_{k,m}}$ equivalence classes all distinct from the equivalence classes provided in *Case 1*.

Consider a word x' of the form $x' = u' \cdot w'$, where u' and w' are components of x' and satisfy the same conditions as the u and w components of x. Then either $|w| = |w'|$ or $|w| \neq |w'|$. First we consider the case $|w| = |w'|$. Then it is not hard to see that $x \equiv_{L_{k,m}} x'$ since for all $z \in \Sigma^*$ that $x \cdot z \in L_{k,m} \iff x' \cdot z \in L_{k,m}$. This is due to the choices of u, u', w and w'. Next we consider the case $|w| \neq |w'|$ and assume that $|w'| < |w|$. For $z = 0^{m-|w|+1}$, we have $x \cdot z \in L_{k,m}$ and $x' \cdot z \notin L_{k,m}$. Therefore $x \not\equiv_{L_{k,m}} x'$.

Now we need to show that x is not $\equiv_{L_{k,m}}$ to any word from *Case 1*. Consider $y = u_0 \cdot \sigma^i$ where $u_0 \in \Sigma^*$, $1 \leq i \leq m$ and $|u_0| \equiv (m-1) \bmod m$. Take $z = \sigma^{m-|x|-1}$, it is not hard to see that $y \cdot z \in L_{k,m}$ but $x \cdot z \notin L_{k,m}$. Hence $y \not\equiv_{L_{k,m}} x$, and in this case $L_{k,m}$ has $m-1$ distinct equivalence classes.

Finally, we consider the case when $x = \epsilon$. Consider a word $x' \in \Sigma^* \backslash \{\epsilon\}$. If $x' \in L_{k,m}$, we set $z = \epsilon$. It is clear that $x \cdot z \notin L_{k,m}$ but $x' \cdot z \in L_{k,m}$. Therefore $x \neq x'$. If $x' \notin L_{k,m}$, then we set $i = |x| \bmod m$ such that $0 \leq i < m$. Now set $z = 0^{m-i+1}$. It is clear that $x \cdot z \notin L_{k,m}$ but $x^{\cdot} \in L_{k,m}$ and thus $x \not\equiv_{L_{k,m}} x'$. Therefore ϵ is in an equivalence class on its own.

We have shown that $L_{k,m}$ has $(k+1)m + (1-k)$ equivalence classes. Therefore, by Myhill Nerode theorem the minimal DFA accepting $L_{k,m}$ has exactly $(k+1)m + (1-k)$ states. ∎

We now reformulate our results above in terms of linear blow-up of the determinization of non-deterministic finite automata.

Theorem 3.1.3 *For every $k > 1$ there exists a regular language L_n over a k-letter alphabet, where $n > k$, such that a minimal NFA recognizing L_n needs exactly n states and the minimal DFA recognizing L_n needs exactly $(k+1) \cdot n - c$ states, where $c = (k+1)^2 - 2$. Moreover, the minimal NFA recognizing L_n needs $O(n)$ transitions.*

Proof The language L_n is $L_{k,m}$ where $n = k + m$. Lemma 3.1.1 shows that this language requires exactly n states to be recognized by a minimal NFA. Lemma 3.1.2 shows that this language requires exactly $(k+1) \cdot n - c$ states to be recognized by a minimal DFA. ∎

One would like to sharpen the theorem above to build a regular language L_n such that the minimal NFA recognizing L_n has exactly n states and the minimal DFA recognizing L_n has exactly $k \cdot n$ states. Below we present another class of languages in which this sharpness can be achieved for infinitely many n.

Let $\Sigma = \{0, 1\}$ and $k, m \in \mathbb{N}^+$. We define the following language

$$U_{k,m} = \{u \cdot 0 \cdot w \mid \ u, w \in \Sigma^*, \ |u| \geq m \ and \ |w| = k\}.$$

Intuitively, $U_{k,m}$ is the set of all words v such that $|v| \geq (m + k + 1)$ and the $k + 1$th letter from the right is 0.

The NFA recognizing $U_{k,m}$, after processing the prefix of an input word of length greater than m nondeterministically guesses that the rest of the string has length k once a 0 is read. Then the automaton verifies that the guess was correct. The NFA $C_{3,6}$ recognizing $U_{3,6}$ which has 11 states is shown in Figure 3.3. It is not hard to see that the NFA $C_{k,m}$ (recognizing $U_{k,m}$) has $m + k + 2$ transitions.

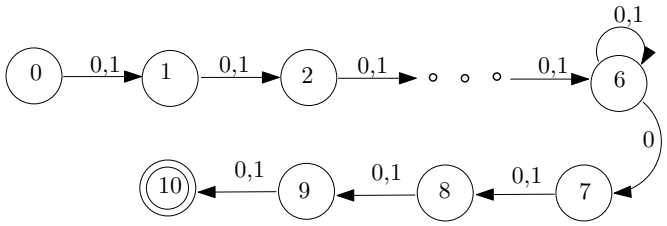


Figure 3.3: The mimimal NFA recognizing the language $U_{k,m}$.

Formally the NFA $C_{k,m} = \langle S, \Sigma, \delta, S_I, F \rangle$ with $m + k + 2$ states accepting $U_{k,m}$ is defined as follows.

1. $S = \{s_0, s_1, ..., s_{m+k+1}\}$.

2. $S_I = \{s_0\}, F = \{s_{m+k+1}\}$.

3. $\delta(s_i, \sigma) = \begin{cases} s_{i+1} & \text{if } 0 \leq i < m \text{ or} \\ & m < i \leq m + k \text{ and } \sigma \in \Sigma \\ s_i & \text{if } i = m, \ \sigma \in \Sigma \\ s_{i+1} & \text{if } i = m \text{ and } \sigma = 0 \end{cases}$

Lemma 3.1.4 *A minimal NFA accepting $U_{k,m}$ has exactly $m + k + 2$ states.*

Proof Assume for a contradiction that there exists an NFA $\mathcal{D} = \langle S', \Sigma, \delta', S'_I, F' \rangle$ accepting $U_{k,m}$ with at most $m+k+1$ states. Consider $0^{m+1}w \in \Sigma^*$ where $|w| = k$. Let $r = p_0, p_1, ..., p_{m+k+1}$ be an accepting run of $\mathcal{D}$ on $0^{m+1}w$. There are $m + k + 2$ states in this run and hence there is at least one state p appearing twice in r. Thus a cycle of length smaller than $m + k + 1$ exists. Let string $v_0, v_1 \in \Sigma^*$ be such that $p \in \delta'^+(s'_0, v_0) \cap \delta'^+(p_i, v_1)$, and string v_2 be such that $\delta'^+(p_i, v_2) \cap F' \neq \emptyset$. Therefore $\delta'^+(s'_0, v_0v_2) \cap F' \neq \emptyset$. However $|v_0v_2| < m + k + 1$ and therefore $v_0v_2 \notin U_{k,m}$. Hence we have reached a contradiction and such a $\mathcal{D}$ does not exist. ∎

Next we show the minimal number of states a DFA requires to accept $U_{k,m}$ is $2^{k+1} + m$. Intuitively, the deterministic automaton needs to remember the first m states of the NFA $\mathcal{A}_{k,m}$. Afterwards, once 0 is read, the DFA needs to remember all the strings of length at most k.

Lemma 3.1.5 *The minimal DFA recognizing $U_{k,m}$ has $2^{k+1} + m$ states.*

Proof For the proof we use Myhill-Nerode Theorem and count the number of $\equiv_{U_{k,m}}$ equivalence classes. Consider a word $x \in \Sigma^*$, where $|x| \geq m$. There are three cases:

Case 1: $|x| \leq m$: Consider any other word $y \in \Sigma^*$ where $|y| \leq m$. There are two possibilities, either $|x| = |y|$ or $|x| \neq |y|$. First we consider the case $|x| \neq |y|$. Without loss of generality we may assume $|x| > |y|$. The word $x \cdot 1^{m-|x|} \cdot 0^{k+1} \in U_{k,m}$ and $y \cdot 1^{m-|x|} \cdot 0^{k+1} \notin U_{k,m}$. Hence, corresponding to each $0 \leq i \leq m$ we have one distinct equivalence class giving us $m + 1$ classes.
Now consider $|x| = |y|$. It is clear for $z \in \Sigma^*$ that $x \cdot z \in U_{k,m} \iff y \cdot z \in U_{k,m}$. Thus, $x \equiv_{U_{k,m}} y$ and we have already counted the equivalence classes. There is a total of $m + 1$ equivalence classes in this case.

Case 2: $m + 1 \leq |x| \leq m + k + 1$: In this case either $x = u \cdot 0 \cdot w$ or $x = u \cdot 1 \cdot w$ where $u, w \in \Sigma^*$ and $|u| = m$. First we consider $x = u \cdot 0 \cdot w$. Take any other word $y = u' \cdot 0 \cdot w'$ where $u', w' \in \Sigma^*$ *and* $|u'| = m$. If $w \neq w'$, then let w_1 be the suffix such that $w = w_0 \cdot \sigma \cdot w_1$ and $w' = w'_0 \cdot \sigma' \cdot w_1$ ($\sigma, \sigma' \in \Sigma$ and $\sigma \neq \sigma'$). Without loss of generality we may assume $\sigma = 0$ and $\sigma' = 1$. It is clear that $x \cdot 1^{k-i+1} \in U_{k,m}$ and $y \cdot 1^{k-i+1} \notin U_{k,m}$, and hence $x \not\equiv_{U_{k,m}} y$. Therefore $U_{k,m}$ has another

$$2^0 + 2^1 + \ldots + 2^k = 2^{k+1} - 1$$

equivalence classes. When $w = w'$ for all $z \in \Sigma^*$ it is clear that $x \cdot z \iff y \cdot z$ and we have already counted the equivalence classes.

Next we consider x of the form $u \cdot 1 \cdot w$ ($u, w \in \Sigma^*$ and $|u| = m$). If $w \in 1^*$, then it is clear that $x \equiv_{U_{k,m}} u$. Otherwise, let w_1 be such that $w = 1^* \cdot 0 \cdot w_1$. Then $x \equiv_{U_{k,m}} 0^{m+1} \cdot w_1$. In both cases we have already counted the equivalence classes. Thus $U_{k,m}$ has another $2^{k+1} - 1$ equivalence classes in Case 2.

Case 3: $|x| > m + k + 1$: We first consider $x \in U_{k,m}$. Then $x = u \cdot 0 \cdot w$ where $u, w \in \Sigma^*$ and $|w| = k$. It is clear that $x \equiv_{U_{k,m}} 0^{m+1} \cdot w$. Now we consider $x \notin U_{k,m}$, then $x = u \cdot 1 \cdot w$ where $u, w \in \Sigma^*$ and $|w| = k$. If $w \in 1^*$ then $x \equiv_{U_{k,m}} u$, else w can be written as $1^* \cdot 0 \cdot w_1$ where $w_1 \in \Sigma^*$. It is clear that $x \equiv_{U_{k,m}} 0^{m+1} \cdot w_1$. In this case, we have already counted the equivalence classes.

From the above arguments, we have shown that $U_{k,m}$ has $2^{k+1} + m$ equivalence classes. Hence, by the Myhill-Nerode theorem, the minimal DFA accepting $U_{k,m}$ has $2^{k+1} + m$ states. ∎

Let p be a natural number. We fix $m = \frac{2^{k+1} - pk - 2p}{p-1}$ and assume that m is also a natural number. For instance, when $p = 2$ we have $m = 2^{k+1} - 2k - 4$. For such chosen m and p we have the following theorem that sharpens Theorem 3.

Theorem 3.1.6 *For every $n = k + m + 2$ there exists a regular language L_n over the binary alphabet such that the minimal NFA recognizing L_n needs exactly n states and the minimal DFA needs exactly $p \cdot n$ states. The minimal NFA recognizing L_n has $O(n)$ transitions.*

Proof The desired language L_n is $U_{k,m}$. We have shown in Lemma 3.1.4 that $n = m + k + 2$. Furthermore, we have shown in Theorem 3.1.5 that the minimal DFA accepting $U_{k,m}$ needs $m + 2^{k+1}$ states. Since $m = \frac{2^{k+1} - pk - 2p}{p-1}$, the minimal DFA accepting $U_{k,m}$ needs $p(m + k + 2)$ states. From the definition of the NFA for $U_{k,m}$, it is not hard to see that it needs $O(n)$ transitions. ∎

Polynomial state explosion

In this section, we construct regular languages which fill in the exponential gap between n and 2^n in the determinization process. We do so by constructing languages which demonstrate polynomial state explosion in the determinization process i.e. for every k, we

construct a language L such that its NFA-state complexity is n and its DFA state complexity is asymptotically n^k.

Let $\Sigma = \{0, 1, \ldots, k-1\}$ be an alphabet of k symbols. For $m \in \mathbb{N}^+$, we define the following languages:

$$B_k = 0^*1^* \ldots (k-1)^*.$$

$$B_{k,m} = \{u \mid u \in B_k \text{ and } |u| = m\}.$$

$$R_{k,m} = \{u \cdot 0 \cdot w \mid u \in B_k \text{ and } w \in B_{k,m}\}.$$

Lemma 3.1.7 *The number of states sufficient for a NFA accepting $R_{k,m}$ is $km + 2$.*

Proof The following NFA $\mathcal{A}_{k,m}$ accepts $R_{k,m}$ with $km + 2$ states. Let $\mathcal{A}_{k,m} = \langle S, \Sigma, \delta, s_I, F\rangle$ be such that:

1. $S = \{s_0, s_1, \ldots, s_{k_1}\} \cup s_k \cup \{s_{0,1}, \ldots, s_{0,m-1}\} \cup \ldots \cup \{s_{k-1,1}, \ldots, s_{k-1,m-1}\} \cup \{s_F\}$.
2. $s_I = s_0$ and $F = \{s_F\}$.
3. For $\sigma \in \Sigma$ and $s_i \in S$, we add the following transitions:

$$\delta(s_i, \sigma) = \begin{cases} \{s_\sigma, s_k\} & \text{if } i \leq k-1 \\ \{s_{\sigma,1}\} & \text{if } i = k \end{cases}$$

4. For $\sigma \in \Sigma$ and $s_{i,j} \in S$, we add the following transitions:

$$\delta(s_{i,j}, \sigma) = \begin{cases} \{s_{i,j+1}\} & \text{if } j < m-1 \text{ and } i = \sigma \\ \{s_{\sigma,j+1}\} & \text{if } j < m-1 \text{ and } i < \sigma \\ \{s_F\} & \text{if } j = m-1 \text{ and } i \leq \sigma \end{cases}$$

For example, the nondeterministic automaton $\mathcal{A}_{2,3}$ is shown in Figure 3.4. It is clear that $\mathcal{A}_{k,m}$ accepts the language $R_{k,m}$. ∎

Next we analyze the DFA-state complexity of $R_{k,m}$.

Lemma 3.1.8 *For the language $B_{k,m}$, where $1 \leq m$ and k is the size of the alphabet, the cardinality of $B_{k,m}$ is $\prod_{i=1}^{k-1} \frac{(m+i)}{i}$.*

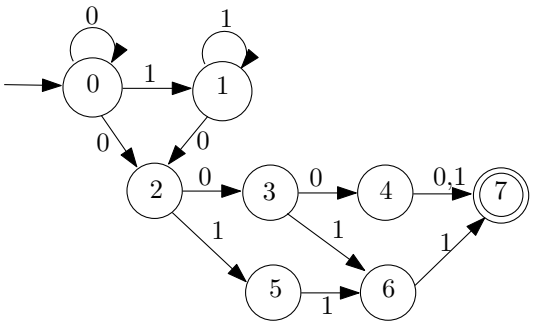


Figure 3.4: The minimal NFA recognizing the language $R_{2,3}$.

Proof We show this by an induction on k. For the base case $k = 2$, it is not hard to see that

$$|B_{2,m}| = (m+1) = \prod_{i=1}^{2-1} \frac{(m+i)}{i}.$$

Assume it is true for $k = n-1$ that

$$|B_{n-1,m}| = \prod_{i=1}^{n-2} \frac{(m+i)}{i}.$$

Words in $B_{n,m}$ are of the form $u_{n-1,m-i} \cdot n^i$ where $0 \leq i \leq m$ and $u_{n-i} \in B_{n-1,m-i}$. Thus, the cardinality of $B_{n,m}$ is:

$$|B_{n,m}| = |B_{n-1,m}| + |B_{n-1,m-1}| + ... + |B_{n-1,0}| = \prod_{i=1}^{n-2} \frac{(m+i)}{i} + \ldots + \prod_{i=1}^{n-2} \frac{(0+i)}{i} = \prod_{i=1}^{n-1} \frac{(m+i)}{i}.$$

This completes the proof. ∎

Lemma 3.1.9 *For the language $B_{k,m}$, where $1 \leq m$ and k is the size of the alphabet*

$$\sum_{i=0}^{m} |B_{k,i}| = \prod_{i=1}^{k} \frac{(m+i)}{i}.$$

Proof We have previously shown in Lemma 3.1.8 that $|B_{k,m}| = \prod_{i=1}^{k-1} \frac{(m+i)}{i}$. Hence we have the following relation: $\sum_{i=0}^{m} |B_{k,i}| = |B_{k,0}| + |B_{k,1}| + ... + |B_{k,m}| = \prod_{i=1}^{k-1} \frac{(i+0)}{i} + \ldots + \prod_{i=1}^{k-1} \frac{(i+m)}{i} = \prod_{i=1}^{k} \frac{m+i}{i}$. ∎

Lemma 3.1.10 *The minimal DFA accepting $R_{k,m}$ needs $\prod_{i=1}^{k} \frac{m+i}{i} + (km+3)$ states.*

Proof We count the number of distinct $\equiv_{R_{k,m}}$-equivalence classes. $R_{k,m}$ is represented by the regular expression $0^* \cdot 1^* \cdot \ldots \cdot k^* \cdot 0 \cdot u$ where $u \in B_{k,m}$. Consider a word $x \in \Sigma^*$, there are two cases:

Case 1: $\exists z \in \Sigma^*$ such that $x \cdot z \in R_{k,m}$. There are the following sub-cases:

1. $x \in 0 \cdot u$ such that $u \in B_{k,i}$ $(0 \leq i \leq m)$: In this case, corresponding to every $u \in B_{k,i}$ we have a distinct $\equiv_{R_{k,m}}$-equivalence class containing the word $0 \cdot u$. Consider distinct words $0 \cdot u$ and $0 \cdot u'$, where $u \in B_{k,n}$ and $u' \in B_{k,j}$ $(0 \leq n, j \leq m)$. Without loss of generality we may assume that $n \leq j$.

   First, consider the case when $n < j$. If $j = m$, then $0 \cdot u' \in R_{k,m}$ but $0 \cdot u \notin R_{k,m}$ and thus $0 \cdot u \not\equiv_{R_{k,m}} 0 \cdot u'$. If $j \neq m$ then let $\sigma \in \Sigma$ be the last symbol that occurs in u'. Then $0 \cdot u' \cdot \sigma^{m-j} \in R_{k,m}$ but $0 \cdot u \cdot \sigma^{m-j} \notin R_{k,m}$ and hence $0 \cdot u \not\equiv_{R_{k,m}} 0 \cdot u'$.

   Next consider the case when $n = j$. Let $u = 0^+ \cdot v$ and $u' = 0^+ \cdot v'$ where $v, v' \in (\Sigma \setminus \{0\})^*$. Since $u \neq u'$, we have $|v| \neq |v'|$. Without loss of generality, assume that $|v| > |v'|$. Then $0 \cdot u' \cdot \sigma^{m-|v'|} \in R_{k,m}$ but $0 \cdot u \cdot \sigma^{m-|v'|} \notin R_{k,m}$ where σ is the last symbol of v'. Thus $0 \cdot u \not\equiv_{R_{2,m}} 0 \cdot u'$.

   By Lemma 3.1.9, we have $\sum_{i=0}^{m} |B_{k,i}| = \prod_{i=1}^{k} \frac{(m+i)}{i}$ and therefore we have $\prod_{i=1}^{k} \frac{(m+i)}{i}$ distinct equivalence classes corresponding to each word of the form $0 \cdot u$.

2. $x \in v \cdot 0 \cdot u$ where $v \in (1^* \cdot \ldots \cdot k^*) \setminus \{\epsilon\}$ and $u \in B_{k,i}$ $(0 \leq i \leq m)$: Let $x = v \cdot 0 \cdot l^j$ and consider another word $w \in v \cdot 0 \cdot p^j$ where $l, p \in \Sigma \setminus \{0\}$ and $l < p$ $(1 \leq j \leq m-1)$. Then $x \not\equiv_{R_{k,m}} w$ since $x \cdot l^{m-j} \in R_{k,m}$ but $w \cdot l^{m-j} \notin R_{k,m}$. Also $x, w \not\equiv_{R_{k,m}} 0 \cdot u'$ $(u' \in B_{k,i})$ since $0 \cdot u' \cdot 0 \cdot 0^m \in R_{k,m}$ but $x \cdot 0 \cdot 0^m \notin R_{k,m}$ (similarly $w \cdot 0 \cdot 0^m \notin R_{k,m}$). Hence, corresponding to each $1 \leq j \leq m-1$ we have k distinct equivalence classes giving us $k \cdot (m-1)$ equivalence classes. We further have one equivalence class for all words of the form $v \cdot 0$. For $u \in B_{k,m}$, all $x \in v \cdot 0 \cdot u$ form another equivalence class.

   Note that $x \in v \cdot 0 \cdot u \cdot \sigma$ ($\sigma \in \Sigma$ and $u \in B_{k,i}$ for $0 \leq i \leq m-1$) is equivalent to all words of the form $v \cdot 0 \cdot \sigma^{i+1}$ and we have already counted these equivalence classes.

   Thus, there are a total of $k \cdot (m-1) + 2$ distinct equivalence classes in this case.

3. $x \in 0^+ \cdot v$ where $v \in (1^* \cdot \ldots \cdot k^*) \setminus \{\epsilon\}$: If $|x| \leq m+1$, then x is of the form $0 \cdot u$ for $u \in B_{k,|x|-1}$ and we have already counted the equivalence class corresponding to x.

   If $|x| > m+1$, let $x = 0 \cdot 0^n \cdot v$ such that $n \geq 0$. If $|v| > m$, then it is easy to see that $x \equiv_{R_{k,m}} \sigma^{|v|}$ where σ is the last symbol of x. Therefore let $|v| \leq m$. If $n + |v| \leq m$, then x is of the form $0 \cdot u$ where $u = 0^n \cdot v \in B_{k,n+|v|}$. If $n + |v| > m$, then $x \equiv_{R_{k,m}} 0 \cdot 0^{m-|v|} \cdot v$. In either case we have already counted the equivalence classes corresponding to x. Hence, all $x \in 0^+ \cdot 1^+$ belong to previously enumerated equivalence classes.

4. $x \in \sigma^+$ ($\sigma \in \Sigma \setminus \{0\}$): For every $\sigma \in \Sigma \setminus \{0\}$, all words of the form σ^+ form a distinct equivalence class. Consider words l^i and p^j such that $l, p \in \Sigma \setminus \{0\}$ and $l < p$ ($i, j \geq 1$). Then $l^i \cdot l \cdot 0^{m+1} \in R_{k,m}$ but $p^j \cdot l \cdot 0^{m+1} \notin R_{k,m}$ and $l^i \not\equiv_{R_{k,m}} p^j$.

   Hence we have $k-1$ equivalence classes in this case.

5. $x = \epsilon$: ϵ forms a distinct equivalence class.

Case 2: $\forall z \in \Sigma^*$ we have $x \cdot z \notin R_{2,m}$: All such x form one distinct equivalence class.

From the above arguments, we can see that there are $(k \cdot m + 3) + \prod_{i=1}^{k} \frac{(m+i)}{i}$ distinct $\equiv_{R_{k,m}}$ equivalence classes. Thus, by the Myhill-Nerode theorem the minimal DFA accepting $R_{k,m}$ has exactly $(k \cdot m + 3) + \prod_{i=1}^{k} \frac{(m+i)}{i}$ states. ∎

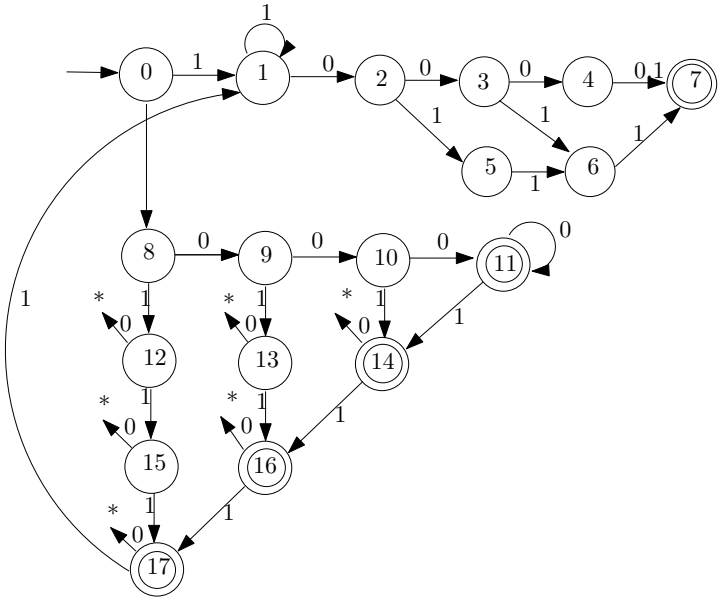


Figure 3.5: The minimal DFA recognizing the language $R_{2,3}$.

As an example the minimal DFA recognizing the language $R_{2,3}$ is shown in figure 3.5. The next lemma analyzes the transition complexity of $R_{k,m}$.

Lemma 3.1.11 *For a fixed k, the NFA recognizing $R_{k,m}$ has $O(n)$ transitions, where $n = km + 2$.*

Proof Let NFA $\mathcal{A}_{k,m} = \langle S, \Sigma, \delta, S_I, F \rangle$ be the NFA recognizing $R_{k,m}$ as defined in Lemma 3.1.7. For a state $s \in S$, let $t(s)$ be the number of states s has transitions to. First we consider the

case $t(s_i)$ where $i \leq k$. For state s_i where $i \leq k-1$, state s_i has transitions to states $s_i, s_{i+1}, \ldots, s_k$ and hence $t(s_i) = k + 1 - i$. Therefore the total number of transitions $s_0, s_1, \ldots, s_{k+1}$ have are

$$\sum_{i=0}^{k-1} t(s_i) = (k+1) + k + \ldots + 2 = \frac{k(k+3)}{2}.$$

Furthermore, it is not hard to see that $t(s_k) = k$. Hence we have accounted for $\frac{k(k+3)}{2} + k$ transitions. Now we consider a state $s_{i,j}$ where $i \leq k-1$ and $j \leq m-2$. The state $s_{i,j}$ has transitions to states $s_{i,j+1}, \ldots, s_{k-1,j+1}$ and hence $t(s_i, j) = k - i$. Hence we have

$$\sum_{i=0}^{k+1} \sum_{j=1}^{m-2} t(s_{i,j}) = (m-2)(1 + \ldots + k) = \frac{(k+1)k(m-2)}{2}$$

Next each state $s_{i,m+1}$ has exactly one transition, hence there are another k transitions. Note that state s_F has no transitions.

Hence in total $\mathcal{A}_{k,m}$ has $\frac{k+1}{2}km + c$ transitions, where $c = \frac{5k^2+11k}{2}$. Therefore the $\mathcal{A}_{k,m}$ recognizing $R_{k,m}$ has $O(n)$ transitions. ∎

In [51], the authors present a n-state NFA such that the minimal DFA requires m states where $n \leq m \leq 2^n$. However, in the worst case (i.e. $m = 2^n - n + 1$) the n-state NFA has $O(n^2)$ transitions as shown by the following lemma.

Lemma 3.1.12 *For $\alpha = 2^n - n + 1$, the n-state NFA $\mathcal{A}$ constructed in [51], such that the DFA-state complexity of $L(A)$ is α, has $O(n^2)$ transitions where the alphabet is $\{a, b, c, d\}$.*

Proof Let $\alpha = 2^n - n + 1$. In this case the states in NFA $\mathcal{A}$ constructed in [51] has the following transitions for symbol d: $\delta(1, d) = \{0, 2\}, \delta(2, d) = \{0, 2, 3\}, \ldots, \delta(n-3, d) = \{0, 2, 3, \ldots, n-2\}$ and $\delta(n-2, d) = \delta(n-1, d) = \delta(n, d) = \{0, 1, 2, 3, \ldots, n-1\}$.

It is not hard to see that $\mathcal{A}$ has $O(n^2)$ transitions for the symbol d. There are $O(n)$ number of transitions for the symbols a, b, c and hence $\mathcal{A}$ has $O(n^2)$ transitions in total. ∎

The next lemma shows that the transition complexity of $R_{k,m}$ is asymptotically less than the number of transitions of the NFA of [51] in the worst case.

Lemma 3.1.13 *For $\alpha = 2^n - n + 1$, the n-state NFA $\mathcal{A}_{k,m}$ recognizing $R_{k,m}$ such that the DFA state complexity of $R_{k,m}$ is $O(\alpha)$ has $O(\frac{n^2}{log_2 n})$ transitions.*

Proof Note that $n = km + 2$ by Lemma 3.1.7. For $k = log_n(2^n - (n-1))$, the DFA state complexity of $R_{k,m}$ is $O(\alpha)$ according to Lemma 3.1.10.

State s_i where $i < k-1$ has $k+1-i$ outgoing transitions and state s_k has k outgoing transitions. States of the form $s_{i,j}$ where $1 \leq i \leq k-1$ have $k-i$ outgoing transitions each. Hence $\mathcal{A}_{k,m}$ has $\frac{1}{2}m(k+1)(k+2)$ transitions. As shown in Lemma 3.1.11, $\mathcal{A}_{k,m}$ has $O(k \cdot n)$ transitions. Since $k = log_n(2^n - (n-1))$, we have $k = \frac{n}{log_2 n} - c$ $(c > 0)$. Hence $\mathcal{A}_{k,m}$ has $O(\frac{n^2}{\log_2 n})$ transitions. ∎

We reformulate the results of the above two lemmas in the following theorem.

Theorem 3.1.14 *For every $k > 1$ there exists a regular language L_n over a k-letter alphabet such that*

1. *The minimal NFA recognizing L_n needs n states, where $n > k$, and the minimal DFA recognizing L_n has $O(n^k)$ states.*

2. *For $\alpha = 2^n - n + 1$, the minimal NFA recognizing L_n and having a blowup of $O(\alpha)$ has $O(\frac{n^2}{log_2 n})$ transitions. This is asymptotically fewer than the $O(n^2)$ transitions required by the NFA with DFA-state complexity α that was described in [51] .*

Proof As shown in Lemma 3.1.7 and Lemma 3.1.10, $km + 2$ states are sufficient for a NFA accepting $R_{k,m}$ and the minimal DFA accepting $R_{k,m}$ has $\prod_{i=0}^{k} \frac{m+i}{i} + (km + 3)$ states. Since $n > k$, this implies the minimal DFA accepting $R_{k,m}$ has $O(n^k)$ states. Our desired language L_n then is $R_{k,m}$ with $n = km + 2$. The second part of the theorem follows from Lemmas 3.1.12 and 3.1.13. ∎

3.1.2 State explosion in complementation

In this section, we investigate the complementation problem for NFA and consider problems around the following question: given n and m with $n \leq m \leq 2^n$, does there exist a regular language whose NFA-state complexity is n such that the NFA-state complexity of the complement of the language is m?

The complementation operation for DFA is efficient and the DFA recognizing the complement of a n-state DFA has at most n states. However for every $n \geq 1$ and the binary alphabet, there exists a n-state NFA such that the minimal NFA recognizing its complement

needs 2^n states [66] (see also [45]). In other words, the complementation problem for the language M is hard in the class of nondeterministic finite automata. Similar to the case of determinization, it is natural to ask whether one can fill in the exponential gap.

In [50] for every n and m such that $n \leq m \leq 2^n$ a regular language L is constructed such that its NFA state complexity is n and the NFA-state complexity of the complement is m. In [52], the authors prove that for every m, n such that $n \leq m \leq 2^n$, there exists a n-state NFA $\mathcal{A}$ such that the NFA state complexity of the complement of $L(\mathcal{A})$ is m for a fixed five letter alphabet. However, in the worst case the number of transitions in the n state NFA is $O(n^2)$.

As in the case of determinization, we provide asymptotic solutions to this question. Our languages exhibit the same behaviour as [52] but the NFA's recognizing these languages have asymptotically fewer transitions than those of [52].

Linear state explosion

First we construct regular languages such that the NFA-state complexity of the complement is asymptotically linear with respect to input NFA-state complexity. Let $\Sigma = \{0, 1, \ldots, k-1\}$ be an alphabet of k symbols. Recall the language $L_{k,m}$ we defined in Section 3.

$$L_{k,m}=\{ux \mid \ x \in \sigma^+, \sigma \in \Sigma, u \in \Sigma^*, \text{ and } |u| \equiv (m-1) \bmod m)\}.$$

We proved that the minimal DFA recognizing $L_{k,m}$ has exactly $(k+1)m + (1-k)$ states. Then it is clear that the DFA recognizing the complement of $L_{k,m}$ has at most $(k+1)m + (1-k)$ many states. Our goal is to show that a succinct representation of this language using NFA still needs exactly $(k+1)m + (1-k)$ many states.

Lemma 3.1.15 *A minimal NFA recognizing the complement of $L_{k,m}$ has at least $(k+1)m + (1-k)$ states.*

Proof Let NFA $\mathcal{A}=\langle S, \Sigma, \delta, s_I, F\rangle$ be a minimal NFA accepting $L^c_{k,m}$, the complement of the language $L_{k,m}$. For $u, v \in \Sigma^*$, define $S(u, v) = \{s \in S \mid s \in \delta^+(s_I, u) \text{ and } \delta^+(s, v) \cap F \neq \emptyset\}$.

First, we show that $\mathcal{A}$ needs at least $k(m-1)+1$ non-accepting states. Consider a word $0^{m-1} \cdot \sigma^i$ where $1 \leq i \leq m$ and $\sigma \in \Sigma$. There are two cases for i:

Case 1: $i < m$: Consider any other word $0^{m-1} \cdot \alpha^j$ where $1 \leq j \leq m-1$ and $\alpha \in \Sigma$. There are two possibilities for i and j:

1. $i = j$: In this case $\alpha \neq \sigma$. It is easy to see that $0^{m-1} \cdot \sigma^i \cdot \alpha^{m-i} \in L^c_{k,m}$. Assume for a contradiction that $S(0^{m-1} \cdot \sigma^i, \alpha^{m-1}) \cap S(0^{m-1} \cdot \alpha^j, \alpha^{m-1}) \neq \emptyset$. Let $s \in S(0^{m-1} \cdot \sigma^i, \alpha^{m-1}) \cap$

$S(0^{m-1}\cdot\alpha^j, \alpha^{m-1})$. Then $\delta^+(s_I, 0^{m-1}\cdot\alpha^j\cdot\alpha^{m-i})\cap F \neq \emptyset$ and hence $0^{m-1}\cdot\alpha^j\cdot\alpha^{m-i}$ is accepted by $\mathcal{A}$. This is a contradiction since $0^{m-1}\cdot\alpha^j\cdot\alpha^{m-i} \notin L^c_{k,m}$.

2. $i \neq j$: Let $\beta \in \Sigma \setminus \{\alpha\}$. Clearly $0^{m-1}\cdot\alpha^j\cdot\beta^{m-i+1} \in L^c_{k,m}$. Assume for the sake of contradiction that $S(0^{m-1}\cdot\sigma^i, \beta^{m-i+1}) \cap S(0^{m-1}\cdot\alpha^j, \beta^{m-i+1}) \neq \emptyset$. Then there must be an $s \in S(0^{m-1}\cdot\sigma^i, \beta^{m-i+1}) \cap S(0^{m-1}\cdot\alpha^j, \beta^{m-i+1})$ and therefore $\delta^+(s_I, 0^{m-1}\cdot\sigma^i\cdot\beta^{m-i+1}) \cap F \neq \emptyset$. Hence $\mathcal{A}$ accepts the word $0^{m-1}\cdot\sigma^i\cdot\beta^{m-i+1}$. This is a contradiction since $0^{m-1}\cdot\sigma^i\cdot\beta^{m-i+1} \notin L^c_{k,m}$.

Since we have $k(m-1)$ words of type $0^{m-1}\cdot\sigma^i$, we have shown that $\mathcal{A}$ needs at least $k(m-1)$ distinct states.

Case 2: $i = m$: Let $\beta \in \Sigma\setminus\{\sigma\}$. Consider any other word $0^{m-1}\cdot\alpha^j$ where $1 \leq j \leq m-1$. Then clearly $0^{m-1}\cdot\alpha^j\cdot\beta \in L^c_{k,m}$. Assume for contradiction that $S(0^{m-1}\cdot\alpha^j, \beta) \cap S(0^{m-1}\cdot\sigma^m, \beta) \neq \emptyset$. Then there must be an $s \in S(0^{m-1}\cdot\alpha^j, \beta) \cap S(0^{m-1}\cdot\sigma^m, \beta)$ and hence $\delta^+(s_I, 0^{m-1}\cdot\sigma^m\cdot\beta) \cap F \neq \emptyset$. Therefore, $\mathcal{A}$ accepts $0^{m-1}\cdot\sigma^m\cdot\beta$ which is a contradiction since $0^{m-1}\cdot\sigma^m\cdot\beta \notin L^c_{k,m}$. Hence $\mathcal{A}$ has at least one more state.

From the two cases above, we conclude that $\mathcal{A}$ has at least $k(m-1)+1$ states. We would now like to show that $\mathcal{A}$ has at least m more states.

Consider a word 0^i for $0 \leq i \leq m-1$. Consider another word 0^j for $0 \leq j \leq m-1$ such that $i \neq j$. Without loss of generality assume that $i < j$. Clearly $0^i\cdot 0^{m-1-j}\cdot 0 \in L^c_{k,m}$. Assume for the sake of contradiction that $S(0^i, 0^{m-1-j}\cdot 0) \cap S(0^j, 0^{m-1-j}\cdot 0) \neq \emptyset$. Then there exists a state $s \in S(0^i, 0^{m-1-j}\cdot 0) \cap S(0^j, 0^{m-1-j}\cdot 0)$ and hence $\delta^+(s_I, 0^j\cdot 0^{m-1-j}\cdot 0) \cap F \neq \emptyset$. Thus $\mathcal{A}$ accepts $0^j\cdot 0^{m-1-j}\cdot 0$ and this is a contradiction since $0^j\cdot 0^{m-1-j}\cdot 0 \notin L^c_{k,m}$.

Now consider a word 0^i ($0 \leq i \leq m-1$) and another word $0^{m-1}\alpha^j$ ($1 \leq j \leq m-1$ and $\alpha \in \Sigma$). Clearly we have $0^i\cdot\alpha^{m-i-1} \in L^c_{k,m}$. Assume for contradiction that $S(0^i, \alpha^{m-i-1}) \cap S(0^{m-1}\alpha^j, \alpha^{m-i-1}) \neq \emptyset$ and there is $s \in S(0^i, \alpha^{m-i-1}) \cap S(0^{m-1}\alpha^j, \alpha^{m-i-1})$. Hence $\delta^+(s_I, 0^{m-1}\cdot\alpha^j\cdot\alpha^{m-i-1}) \cap F \neq \emptyset$ and therefore $\mathcal{A}$ accepts $0^{m-1}\cdot\alpha^j\cdot\alpha^{m-i-1}$. This is a contradiction since $0^{m-1}\cdot\alpha^j\cdot\alpha^{m-i-1} \notin L^c_{k,m}$.

From the above arguments, we can conclude that $\mathcal{A}$ has at least m more states as required. Hence we have shown that $\mathcal{A}$ has at least $k(m-1)+1+m = (k+1)m+(1-k)$ states. ∎

Theorem 3.1.16 *For every $k > 1$ there exists a regular language L_n over k-letter alphabet, where $n > k$, such that*

1. *The minimal NFA recognizing L_n needs exactly n states and the minimal NFA recognizing the complement of L_n needs exactly $(k+1)n - c$ states, where $c = (k+1)2 - 2$.*

2. *The minimal NFA recognizing L_n needs $O(n)$ transitions.*

Proof The language L_n is $L_{k,m}$ where $n = k + m$. We have shown in Lemma 1 a minimal NFA accepting L_n needs exactly n states. In Theorem 3 we have shown the minimal DFA accepting L_n needs exactly $(k+1)n - c$ states, hence the complement of L_n can be accepted by a $(k+1)n - c$ states DFA. Furthermore in Lemma 3.1.15, we have shown that $(k+1)n - c$ states are necessary for a minimal NFA accepting the complement of L_n. Hence this proves $(k+1)n - c$ states are necessary and sufficient for a minimal NFA recognizing L_n^c. ∎

Polynomial state explosion

In this section, we fill in the exponential gap for the complementation of NFA's i.e. for every $n, k \geq 2$, there exists a $O(n)$ state NFA such that the minimal NFA accepting its complement has between $O(n^{k-1})$ and $O(n^{2k})$ states where the alphabet is of size k. Recall the language $B_{k,m}$ we defined in section 3.1.1 i.e. $\{u \mid u \in 0^* \cdot \ldots \cdot (k-1)^* \text{ and } |u| = m\}$. Then it is clear that the following NFA $\mathcal{A} = (S, \Sigma, \delta, s_I, F)$ with $k(m-1) + 2$ states recognizes $B_{k,m}$:

1. $S = \{s_0\} \cup \{s_{0,1}, \ldots, s_{0,m-1}\} \cup \ldots \cup \{s_{k-1,1}, \ldots, s_{k-1,m-1}\} \cup \{s_F\}$ and $\Sigma = \{0, 1, \ldots, k-1\}$.

2. $s_I = s_0$ and $F = \{s_F\}$.

3. For $0 \leq i < k$ and $\sigma \in \Sigma$, $\delta(s_0, \sigma) = \{s_{\sigma,1}\}$.

4. $$\delta(s_{i,j}, \sigma) = \begin{cases} \{s_{i,j+1}\} & \text{if } j < m-1 \text{ and } i = \sigma \\ \{s_{\sigma,j+1}\} & \text{if } j < m-1 \text{ and } i < \sigma \\ \{s_F\} & \text{if } j = m-1 \text{ and } i \leq \sigma \end{cases}$$

The NFA recognizing $B_{2,4}$ is shown in Figure 3.6. It is not hard to see that the NFA for $B_{k,m}$ has $O(k^2 m)$ transitions.

Later we will need to use the following language: $G_{k,m,\alpha} = \{\beta \cdot u \mid \beta \cdot u \in B_{k,m} \text{ and } \beta \in \{0, \ldots \alpha - 1\}\}$ where $\alpha \in \{1, \ldots, k-1\}$. The following NFA C recognizes $G_{k,m,\alpha}$:

1. $S = \{s_{0,0}, \ldots, s_{0,m}\} \cup \ldots \cup \{s_{\alpha-1,0}, \ldots, s_{\alpha,m}\} \cup \{s_{\alpha,1}, \ldots, s_{\alpha,m}\} \ldots \cup \{s_{k-1,1}, \ldots, s_{k-1,m}\}$.

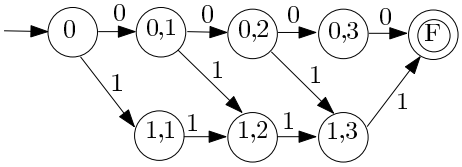


Figure 3.6: The NFA recognizing the language $B_{2,4}$.

2. The initial states are $\{s_{\sigma,0} \mid \sigma \in \Sigma\}$ and $F = \{s_{\sigma,m} \mid \sigma \in \Sigma\}$.

3. For $i, \sigma \in \Sigma$ and $0 \leq j < m$:

$$\delta(s_{i,j}, \sigma) = \begin{cases} \{s_{i,j+1}\} & i = \sigma \\ \{s_{\sigma,j+1}\} & i < \sigma \end{cases}$$

The NFA recognizing $G_{2,2,1}$ is shown in Figure 3.7. It is not hard to see that the NFA for $G_{k,m,\alpha}$ has $\alpha(m+1) + (k-\alpha)m$ states and $O(k^2 m)$ transitions.

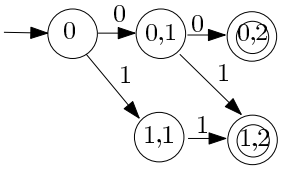


Figure 3.7: The NFA recognizing the language $G_{2,2,1}$.

Lemma 3.1.17 *For every $k, m > 1$, there exists a $O(m)$-state NFA $\mathcal{B}$ such that NFA accepting the complement of $L(\mathcal{B})$ has at least $O(m^{k-1})$ states.*

Proof Consider the language $H_{k,m} = (\Sigma^* \cdot 0 \cdot y \cdot (\Sigma \setminus \{0\}) \cdot \Sigma^*) + (\Sigma^* \cdot 1 \cdot y \cdot (\Sigma \setminus \{1\}) \cdot \Sigma^*) + \ldots + (\Sigma^* \cdot (k-1) \cdot y \cdot (\Sigma \setminus \{k-1\}) \cdot \Sigma^*)$ where the following conditions hold:

1. $|y| = m$ and

2. $y = y_1 \cdot y_2$ such that $a \cdot y_1, y_2 \cdot b \in 0^* \cdot 1^* \cdot \ldots \cdot (k-1)^*$ for some symbols $a \neq b$.

Intuitively, the NFA recognizing $H_{k,m}$ behaves as follows: It guesses the position of a symbol $a \in \Sigma$ and then starts verifying whether the next $m+1$ symbols are in $B_{k,m}$. If at position i in this verification, the automaton reads a symbol α such that the symbol at

position $i-1$ is $\beta > \alpha$ then the automaton tries to verify whether the last $m-i+1$ symbols are in $B_{k,m-i+1}$ and the $(m+1)$th symbol is $b \neq a$.

Formally, let the following be the NFA's recognizing $B_{k,m}, \ldots, B_{k-i,m}, \ldots, B_{1,m}$ respectively (i.e. $\mathcal{A}^i$ recognizes $B_{k-i,m}$ where $0 \leq i \leq k-1$):

$$\mathcal{A}^0 = (S^{A^0}, \Sigma, \delta^{A^0}, s_{I^{A^0}}, F^{A^0})$$

$\ldots$

$$\mathcal{A}^i = (S^{A^i}, \Sigma \setminus \{0, \ldots, i-1\}, \delta^{A^i}, s_{I^{A^i}}, F^{A^i})$$

$\ldots$

$$\mathcal{A}^{k-1} = (S^{A^{k-1}}, \Sigma \setminus \{0, \ldots, k-2\}, \delta^{A^{k-1}}, s_{I^{A^{k-1}}}, F^{A^{k-1}})$$

Let $C^0 = (S^{C^0}, \Sigma, \delta^{C^0}, s_{I^{C^0}}, F^{C^0})$ be the NFA recognizing $G_{k,m-2,k-1}$ and the following be the NFA's recognizing $G_{k,m-1,1}, \ldots G_{k,m-1,k-2}$ (i.e. C^i recognizes $G_{k,m,i}$ where $1 \leq i \leq k-2$) .

$$C^1 = (S^{C^1}, \Sigma, \delta^{C^1}, s_{I^{C^1}}, F^{C^1})$$

$\ldots$

$$C^{k-2} = (S^{C^{k-2}}, \Sigma, \delta^{C^{k-2}}, s_{I^{C^{k-2}}}, F^{C^{k-2}})$$

Also, let $C^{k-1} = (S^{C^{k-1}}, \Sigma \setminus \{k-1\}, s\delta^{C^{k-1}}, s_{I^{C^{k-1}}}, F^{C^{k-1}})$ be the NFA recognizing $G_{k-1,m-1,k-1}$.

The following NFA $\mathcal{D}_{k,m}$ accepts $H_{k,m}$:

1. $S = \{s_0\} \cup S^{A^{k-1}} \cup \ldots \cup S^{A^0} \cup S^{C^{k-1}} \cup \ldots S^{C^1} \cup S^{C^0} \cup \{s_F\}$.
2. $I = \{s_0\}$ and $F = \{s_F\}$.
3. For every $\sigma \in \Sigma$, $\delta(s_0, \sigma) = \{s_0, s_{I^{A^\sigma}}\}$ and $\delta(s_F, \sigma) = \{s_F\}$.
4. For $s \in S^{A^i}$ $(1 \leq i \leq k)$ and $\sigma \in \Sigma$, $\delta(s, \sigma) = \delta^{A^i}(s, \sigma)$. Similarly for $s \in S^{C^j}$ $(0 \leq j \leq k-1)$.
5. For every $0 \leq i \leq k-1$, the following conditions hold:
   (a) $s_F \in \delta(s_F^{A^i}, \sigma)$ for every $\sigma \in \Sigma \setminus \{i\}$.
   (b) For $i > 0$, $s_F \in \delta(s^{C^i}_{\alpha,m-1}, \sigma)$ for every α in the alphabet of C_i and $\sigma \in \Sigma \setminus \{i\}$. For $i = 0$, $s_F \in \delta(s^{C^0}_{\alpha,m-2}, \sigma)$ for every $\alpha \in \Sigma$ and $\sigma \in \Sigma \setminus \{0\}$.

6. For every $1 \leq i \leq k-1$, the following conditions hold:

   (a) For $\sigma \in \{0, \ldots, i-1\}$, $s^{C^i}_{\sigma,0} \in \delta(s_{I^{A^i}}, \sigma)$.

   (b) For $0 \leq j \leq k-i-1$ and $1 \leq j' \leq m-1$, $s^{C^i}_{\sigma,j'} \in \delta(s^{A^i}_{j,j'}, \sigma)$ for every $\sigma \in \{0, \ldots, j+i-1\}$.

7. For $i = 0$, the following is true:

   (a) For $1 \leq j \leq k-1$ and $1 \leq j' \leq m-1$, $s^{C^0}_{\sigma,j'-1} \in \delta(s^{A^0}_{j,j'}, \sigma)$ for every $\sigma \in \{0, \ldots, j-1\}$.

The NFA recognizing $H_{2,4}$ is shown in Figure 3.8. Since the NFA recognizing $B_{k,m}$ has $k(m-1)+2$ states and the NFA for $G_{k,m,\alpha}$ has $\alpha(m+1)+(k-\alpha)m$ states, it is not hard to see that the NFA recognizing $H_{k,m}$ has $O(k^2m) = O(m)$ states.

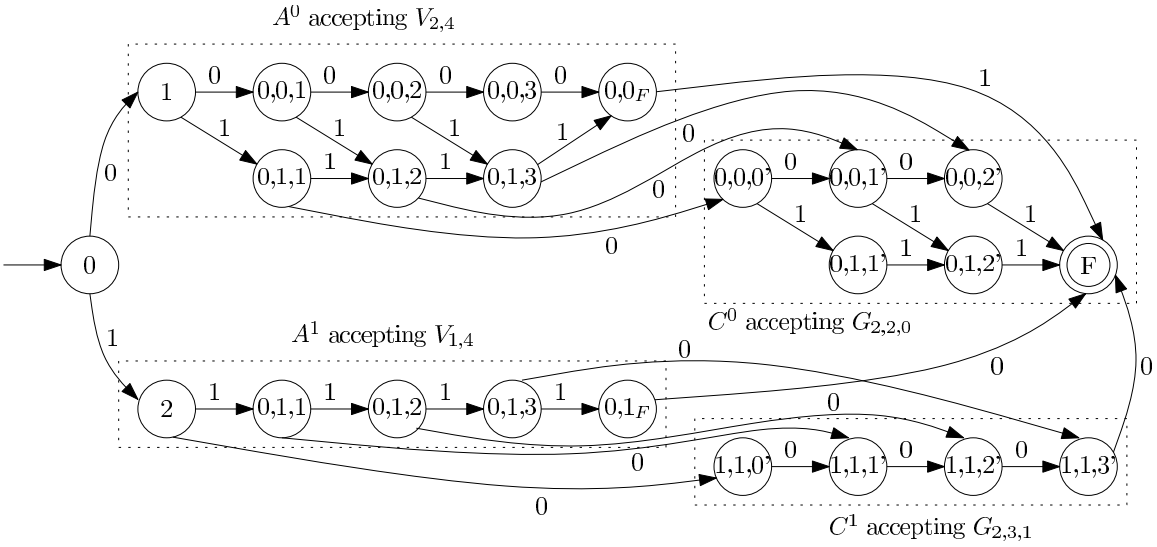


Figure 3.8: The NFA recognizing the language $H_{2,4}$

Let $\mathcal{A} = (S, \Sigma, \delta, s_I, F)$ be a NFA recognizing $H^c_{k,m}$. Consider any word $w \in B_{k,m+1}$. Then $w \cdot w \in H^c_{k,m}$ since any symbols in $w \cdot w$ that are separated by m positions are identical. We define $S(w) = \{s \in S \mid s \in \delta^+(s_I, w) \text{ and } \delta^+(s, w) \cap F \neq \emptyset\}$. Consider any other word $w' \in B_{k,m+1}$. Assume for the sake of contradiction that $S(w) \cap S(w') \neq \emptyset$. Then there is a state $s \in S(w) \cap S(w')$ and we have $\delta^+(s_I, w \cdot w') \cap F \neq \emptyset$ and $\delta^+(s_I, w' \cdot w) \cap F \neq \emptyset$. Hence $\mathcal{A}$ accepts $w \cdot w'$ and $w' \cdot w$.

However w and w' are distinct words and differ for at at least one position $1 \leq p \leq m+1$. Hence $w \cdot w'$ is of the form $x_1 x_2 \ldots x_{p-1} a \ldots x_{m+1} x'_1 x'_2 \ldots x'_{p-1} b \ldots x'_{m+1}$ such that $a \neq b$. There are two cases:

1. $x_{m+1} \leq x'_1$: Since $w, w' \in B_{k,m}$, it is not hard to see that $ax_{p+1} \dots x'_{p-1} \in 0^* \cdot 1^* \cdot \dots \cdot (k-1)^*$ and $b \in 0^* \cdot 1^* \cdot \dots \cdot (k-1)^*$. Hence $w \cdot w'$ is of the form $\Sigma^* \cdot a \cdot y \cdot b \cdot \Sigma^*$ where $y_1 = x_{p+1} \dots x'_{p-1}$ and $y_2 = \epsilon$ and $a \cdot y_1, y_2 \cdot b \in 0^* \cdot 1^* \cdot \dots \cdot (k-1)^*$.

2. $x_{m+1} > x'_1$: In this case $ax_{p+1} \dots x_{m+1} \in 0^* \cdot 1^* \cdot \dots \cdot (k-1)^*$ and $x'_1 \dots x'_{p-1} b \in 0^* \cdot 1^* \cdot \dots \cdot (k-1)^*$. Hence, $w \cdot w'$ is of the form $\Sigma^* \cdot a \cdot y \cdot b \cdot \Sigma^*$ where $y_1 = x_{p+1} \dots x_{m+1}$ and $y_2 = x'_1 \dots x'_{p-1}$ and $a \cdot y_1, y_2 \cdot b \in 0^* \cdot 1^* \cdot \dots \cdot (k-1)^*$.

In both cases $w \cdot w' \notin H^c_{k,m}$ but the word is accepted by $\mathcal{A}$. A very similar argument can be made for $w' \cdot w$. We have arrived at a contradiction. By Lemma 3.1.8 there are $O(m^{k-1})$ words in $B_{k,m+1}$ and hence $\mathcal{A}$ has at least $O(m^{k-1})$ states. ∎

In the following theorem we give an upper bound for the DFA recognizing the complement of the language $H_{k,m}$.

Lemma 3.1.18 *For every $k, m > 1$, the DFA recognizing the complement of the language $H_{k,m}$ has at most $O(m^{2k})$ states.*

Proof We use the Myhill-Nerode theorem to prove this bound. First we observe that for any words $u, v \in H_{k,m}$, we have $u \equiv v$.

Now consider any word $w \notin H_{k,m}$ such that $w \neq \epsilon$. Then w must be of the form $\Sigma^* \cdot a \cdot y$ where $a \in \Sigma$ and $0 \leq |y| \leq m$. Here y is the maximal length word such that $y = y_1 \cdot y_2$ and $a \cdot y_1 \in 0^* \cdot \dots (k-1)^*$.

Consider any other word $w' \notin H_{k,m}$ such that $w' \in \Sigma^* \cdot a \cdot y$. Then it is not hard to see that $w \equiv w'$ since $w \cdot x \in H_{k,m}$ iff $w' \cdot x \in H_{k,m}$ for any $x \in \Sigma^*$. There are at most $O(m^{2k})$ words of the form $a \cdot y$ and hence there are at most $O(m^{2k})$ equivalence classes. ∎

Lemma 3.1.19 *For $n > 0$ and $\alpha = 2^n - n + 1$, the NFA $\mathcal{A}$ constructed in [52] with n states such that the NFA accepting the complement has α states has $O(n^2)$ number of transitions.*

Proof The NFA $\mathcal{A}$ constructed in [52] with n states has exactly has an alphabet of five symbols a, b, c, d, f. The number of transitions for symbols a, b, c, d are exactly the same as those for the NFA constructed in [51] which is $O(n^2)$ by lemma 3.1.12. The symbols f only adds $O(n)$ number of transitions. Hence, the NFA $\mathcal{A}$ constructred in [52] has $O(n^2)$ number of transitions. ∎

Lemma 3.1.20 *For $n > 0$ and $\alpha = 2^n - n + 1$, the $O(n)$-state NFA $D_{k,m}$ accepting $H_{k,m}$, such that the NFA accepting $H^c_{k,m}$ has $O(\alpha)$ states, has $O(\frac{n^2}{log_2 n})$ transitions.*

Proof In order for the minimal NFA for $H^c_{k,m}$ to have $O(\alpha)$ states, we must have $k \in O(\frac{n}{log_2 n})$. The NFA $D_{k,m}$ has $O(k^3 m)$ number of transitions since the NFA's for $B_{k,m}$ and $G_{k,m,\alpha}$ have $O(k^2 m)$ transitions each. Also $D_{k,m}$ has $O(k^2 m)$ states by lemma 3.1.17. Hence $D_{k,m}$ has $O(kn)$ transitions where $n \in O(k^2 m)$. Since $k \in O(\frac{n}{log_2 n})$, it is clear that $D_{k,m}$ has $O(\frac{n^2}{log_2 n})$ transitions. ∎

The following theorem follows from lemmas 3.1.17, 3.1.18, 3.1.19 and 3.1.20.

Theorem 3.1.21 *For every $k, n > 1$, there exists a NFA $\mathcal{A}$ with $O(n)$ states such that:*

1. *The minimal NFA recognizing the complement of $L(\mathcal{A})$ has between $O(n^{k-1})$ and $O(n^{2k})$ states.*

2. *In the worst case, the NFA $\mathcal{A}$ has $O(\frac{n^2}{log_2 n})$ transitions which is asymptotically fewer than the $O(n^2)$ transitions of the NFA described in [52].*

3.2 Complexity of finite word and tree languages

In section 3.1, we analyzed the complexity of regular languages from the point of view of the tradeoff between NFA and DFA-state complexity. Here we investigate the complexity of regular languages from another direction i.e. the state complexity of natural subclasses of regular languages.

The state complexity of union and intersection on regular languages L_1 and L_2 is bounded above by $m \cdot n$ where m, n are the number of states in the minimal DFA's recognizing L_1 and L_2 respectively. It is also known that over the class of all regular languages, this upper bound is tight [91]. This means for some regular languages L_1 and L_2 with state complexity m and n respectively, $m \cdot n$ states are necessary for a DFA to recognize $L_1 \cup L_2$ or $L_1 \cap L_2$. Hence the question naturally arises if we can obtain a better upper bound for natural subclasses of regular languages.

This section addresses this problem for the class of finite languages. Finite languages are important in many practical applications. A good example of such an area is natural language processing where finite automata are used to represent very large (but finite)

dictionaries of words[27, 67]. In many of these applications, the union and intersection operations are frequently required [67].

The state complexity for finite word languages has been investigated in [26, 44, 91]. In [44], the authors proved that the upper bounds for the state complexity of union and intersection of finite word languages are $mn-(m+n)$ and $mn-3(m+n)+12$ respectively. They show that these bounds are tight when the size of the alphabet can be varied depending on m and n. The authors also provide examples of finite word languages (with fixed alphabet size) for which union and intersection have a state complexity of $c \cdot mn$ for some constant c. This shows that one cannot hope to prove that the state complexity for union and intersection of finite languages is asymptotically better than $O(mn)$.

The results of [44] give rise to two questions:

1. What is the state complexity of finite languages when the alphabet size is fixed?

2. In [44], a lower bound of $O(m + n)$ was shown for the difference between $m \cdot n$ and the state complexity of union and intersection of finite languages. Since the asymptotic bound for state complexity of union and intersection cannot be improved beyond $O(m \cdot n)$, can we improve on the difference between $m \cdot n$ and the actual state complexity of union and intersection?

Here we provide answers to these questions. For question (1), we investigate the state complexity of finite languages such that the length of the words in the language is bounded by a parameter h. We show that the state complexity such languages language can be as high as $\frac{ck^h}{h}$ for some constant c (here k is the size of the alphabet) (See Theorem 3.2.6). We also answer question (2) positively by improving the lower bound of the difference between $m \cdot n$ and the state complexity of union and intersection from $O(m + n)$ to $O((\log_k \min\{m, n\})(m + n))$(See Theorem 3.2.13).

Analogous to the case of word languages, we consider the state complexity of finite tree languages. Similar to the word case, the state complexity of a regular tree language L is the number of states in the minimal deterministic tree automaton recognizing it. Using a technique very similar to the finite word languages, we show that the asymptotic lower bound of the difference between mn and the state complexity of union and intersection in this case is $O((\log_k \log_2 \min\{m, n\})(m + n))$ (See Theorem 3.2.18 and Corollary 3.2.19).

3.2.1 Finite languages with bounded word length

In this section we investigate the following question: Given $h \in \mathbb{N}$, how many states are required by a DFA to recognize a finite language whose words have length bounded by h? In other words, we would like to measure the state complexity of the class of finite languages with bounded word length. The goal is to express the state complexity in terms of h. In order to do this, we first analyze the state complexity of a special class of finite languages, called uniform-length languages. These are languages where all words in the language have the same length. We then extend the result to finite languages where the word length is bounded.

State complexity of uniform-length languages

The following definition singles out the languages under investigation.

Definition 3.2.1 *A* uniform-length language with length h *is* $L \subseteq \Sigma^*$ *where all words in* L *have the same length* h*.*

A *level automaton* is a DFA where for each state s (apart from the reject state), all words that take the automaton from the initial state to s have the same length. We call this length the *level* of the state s. The *height* of a level automaton is the maximum level of a state in the automaton. Note that any finite language can be recognized by a level automaton. The following lemma relates level automata with uniform-length languages.

Lemma 3.2.1 *The minimal automaton for any uniform-length language* L *with length* h *is a level automaton with height* h*.*

Proof Let L be a uniform-length language and $\mathcal{M}$ be the minimal automaton recognizing L. If $\mathcal{M}$ is not a level automaton, then there is some state s which is not the reject state such that two words x, y with different lengths both take $\mathcal{M}$ from the initial state to s. If there is a path that goes from s to an accepting state, then L would not be uniform-length. Hence all reachable states from s are non-accepting, which contradicts the minimality of $\mathcal{M}$. Hence $\mathcal{M}$ must be a level automaton. Furthermore $\mathcal{M}$ must have height h as otherwise it would accepts words whose length is not equal to h. ∎

Our goal is to investigate the state complexity of uniform-length languages of length h. In the rest of this section we focus on the case when the alphabet $\Sigma = \{0, 1\}$. The techniques used in the proofs can then be generalized to the cases when $|\Sigma| > 2$.

Before we begin our analysis of uniform-length languages we introduce some notions and facts which we will require. A language is *prefix-free* if any two distinct words w_1, w_2 in the language are not comparable with respect to the prefix relation $\prec_{\mathsf{pref}}$. A prefix-free language $L \subseteq \Sigma^*$ may be naturally identified with a $|\Sigma|$-ary tree $\mathsf{tree}(L)$ as follows: words in L are leaves of $\mathsf{tree}(L)$ and prefixes of all words in L are the internal nodes of $\mathsf{tree}(L)$.

Let L be a prefix-free language. For any $w \in \mathsf{tree}(L)$, let $L(w)$ be the language $\{y \mid wy \in L\}$. Note that for any $w_1, w_2 \in L$, we have $w_1 \equiv_L w_2$ if and only if $L(w_1) = L(w_2)$. Hence by the Myhill-Nerode theorem we have the following lemma which will be useful later.

Lemma 3.2.2 *Let L be a prefix-free language and $w_1, w_2 \in \mathsf{tree}(L)$. The minimal automaton recognizing L reaches the same state upon reading w_1 and w_2 if and only if $L(w_1) = L(w_2)$.*

Since each uniform-length language is prefix-free, we can define $\mathcal{T}_i$ as the class of trees of the form $\mathsf{tree}(L)$ where L is a uniform-language of length i, $i \geq 0$. The class $\mathcal{T}_i$ can be defined inductively as follows:

- The class $\mathcal{T}_0$ contains the only height-0 tree $\{\varepsilon\}$.
- For any $j > 0$, the class $\mathcal{T}_j$ contains all trees of the form

$$\{\varepsilon\} \cup \{0w \mid w \in t_0\} \cup \{1w \mid w \in t_1\}$$

  where $t_0, t_1 \in \mathcal{T}_{j-1} \cup \{\emptyset\}$ and t_0 and t_1 are not both $\emptyset$.

The number of trees in $\mathcal{T}_i$ is given by the following recurrence: $|\mathcal{T}_0| = 1$ and $|\mathcal{T}_j| = |\mathcal{T}_{j-1}|^2 + 2|\mathcal{T}_{j-1}|$ for $j \geq 1$. Solving this recurrence we get

$$|\mathcal{T}_i| = 2^{2^i} - 1. \tag{3.1}$$

Fig. 3.9 shows the general shape of the tree $\mathsf{tree}(L_{\max}(i))$ for the language $L_{\max}(i)$.

We fix a mapping $T_i : \{0,1\}^{2^i} \to \mathcal{T}_i$ such that $T_i(0^{2^i}) = T_i(1^{2^i})$ and for all $w_1, w_2 \in \{0,1\}^{2^i} \setminus \{0^{2^i}, 1^{2^i}\}$, we have $T_i(w_1) \neq T_i(w_2)$ where $w_1 \neq w_2$ and $T_i(w_1), T_i(w_2)$ are not equal to $T_i(0^{2^i})$. For $i \geq 0$, define the uniform length language $L_{\max}(i)$ as $\{wy \mid |w| = 2^i, y \in \mathsf{leaves}(T_i(w))\}$. Note that $L_{\max}(i)$ is a prefix-free language.

Lemma 3.2.3 *For $i \geq 0$, the language $L_{\max}(i)$ has maximal state complexity in the class of all uniform-length languages of length $2^i + i$.*

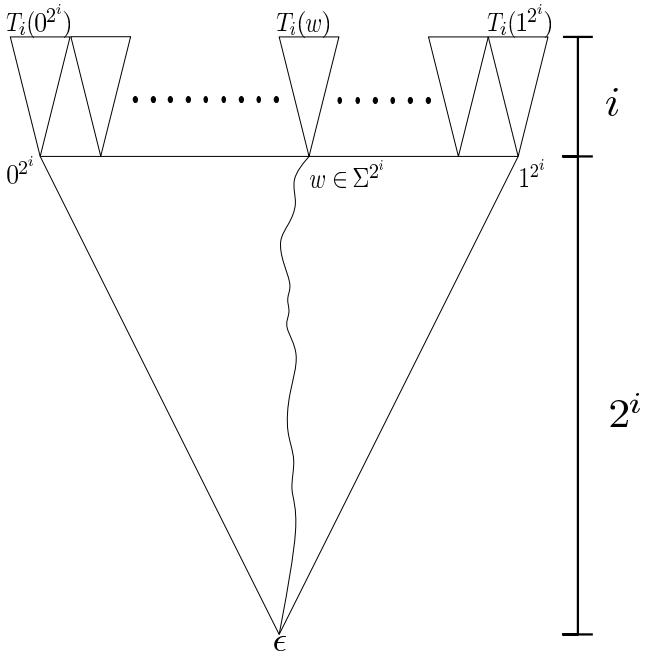


Figure 3.9: Illustration of the tree $\mathsf{tree}(L_{\max}(i))$.

Proof Fix $i \geq 0$. For simplicity we write $L_{\max}(i)$ as $L_{\max}$. An automaton $\mathcal{M}$ recognizing $L_{\max}$ is defined as follows. For every word $w \in \{0,1\}^j$ where $j < 2^i$, $\mathcal{M}$ contains a state s_w at level j. For every tree t in $\mathcal{T}_j$ where $0 \leq j \leq i$, $\mathcal{M}$ contains a state q_t at level $2^i + i - j$. The initial state of $\mathcal{M}$ is the state s_ε. The accepting state of $\mathcal{M}$ is q_t where t is the level-0 tree $\{\varepsilon\}$. The transition function Δ of $\mathcal{M}$ is defined as follows.

- For each $w \in \{0,1\}^j$ where $0 \leq j < 2^i - 1$, set $\Delta(w, \sigma) = s_{w\sigma}$ where $\sigma \in \{0,1\}$
- For each $w \in \{0,1\}^{2^i-1}$, set $\Delta(w, \sigma) = q_{T_i(w\sigma)}$.
- For each $t \in \mathcal{T}_j$ where $0 < j \leq i$, by definition t is of the form $\{\varepsilon\} \cup \{0x \mid x \in t_0\} \cup \{1x \mid x \in t_1\}$ for some $t_0, t_1 \in \mathcal{T}_{j-1} \cup \{\emptyset\}$. For $\sigma \in \{0,1\}$, if $t_\sigma \in \mathcal{T}_{j-1}$, then set $\Delta(q_t, \sigma) = q_{t_\sigma}$; if $t_\sigma = \emptyset$ then set $\Delta(q_t, \sigma)$ as the reject state.

The automaton $\mathcal{M}$ is minimal for $L_{\max}$ because the following two facts.

- For every $w_1, w_2 \in \{0,1\}^j$ where $0 \leq j < 2^i$, we have $L_{\max}(w_1) \neq L_{\max}(w_2)$ whenever $w_1 \neq w_2$. Hence by Lemma 3.2.2 level j of the minimal automaton for $L_{\max}$ must contain 2^j states.

- For every $t \in \mathcal{T}_j$ where $0 \leq j \leq i$, there exists a word w with $|w| = 2^i + i - j$ such that $L_{\max}(w) = t$. Hence by Lemma 3.2.2 level $2^i + i - j$ of the minimal automaton for $L_{\max}$ must contain $2^{2^j} - 1$ states.

Let $\mathcal{M}'$ be the minimal automaton for a uniform-length language of length $2^i + i$. By Lemma 3.2.1 $\mathcal{M}'$ is a level automaton of height $2^i + i$. We would like to prove that for every $0 \leq j \leq 2^i + i$, the number of states in $\mathcal{M}'$ at level j is at most the number of states in $\mathcal{M}$ at level j. This is, again, due to two facts.

- For any $j \geq 0$, the number of states at level j of $\mathcal{M}'$ is at most 2^j.
- For any $j \leq 2^i + i$, the number of states at level j is at most $2^{2^{2^i+i-j}} - 1$. This is due to Lemma 3.2.2 and equation (3.1).

Hence the maximal number of states at every level j, where $0 \leq j \leq 2^i + i$ is $\min\{2^j, 2^{2^{2^i+i-j}}\}$, which matches the number of states in $\mathcal{M}$ at level j. ∎

By the above lemma, the minimal automaton $\mathcal{M}$ for the language $L_{\max}(i)$ has exactly

$$\begin{aligned} & 1 + 2 + 2^2 + \ldots + 2^{2^i-1} + 2^{2^i} - 1 + 2^{2^{i-1}} - 1 + \ldots + 2^{2^0} - 1 + 1 \\ = \; & 2^{2^i+1} + 2^{2^{i-1}} + 2^{2^{i-2}} + \ldots + 2^{2^0} - i - 1. \end{aligned}$$

states.

Theorem 3.2.4 *Suppose $h = 2^i + i$ for some $i \geq 0$. The state complexity for the class of uniform-length languages of length h is $\Theta(2^h/h)$.*

Proof By Lemma 3.2.3 and the above argument, the state complexity $SC(h)$ for the class of uniform-length languages of length h is

$$2^{2^i+1} + 2^{2^{i-1}} + 2^{2^{i-2}} + \ldots + 2^{2^0} - i - 1.$$

We analyze the asymptotic behavior of the above expression as follows. For any $k \geq 1$ we have

$$2^{2^k} + 2^{2^{k-1}} + \ldots + 2^{2^0} < \sum_{j=1}^{2^k} 2^j = 2^{2^k+1}.$$

Hence we have:

$$SC(h) < 2^{2^i+1} + 2^{2^{i-1}+1} < 3 \cdot 2^{2^i}. \tag{3.2}$$

Solving the equation $h = 2^i + i$ we get

$$2^i = \frac{W(2^h \ln 2)}{\ln 2} \tag{3.3}$$

where W is the product logarithmic function. The function W, also called the *Lambert W* function, is a branch of the inverse relation of the function $f(w) = we^w$. $W(x)$ is defined and has a single value for all non-negative real numbers [25]. Substituting (3.3) into (3.2) we get

$$SC(h) < 3 \cdot 2^{\frac{W(2^h \ln 2)}{\ln 2}} = 3 \cdot e^{W(2^h \ln 2)}.$$

Since $W(z)e^{W(z)} = z$ for any complex z, we get

$$\begin{aligned} SC(h) &< 3 \cdot \frac{W(2^h \ln 2)e^{W(2^h \ln 2)}}{W(2^h \ln 2)} \\ &= 3 \cdot \frac{2^h \ln 2}{W(2^h \ln 2)} \\ &= 3 \ln 2 \cdot \frac{2^h}{W(2^h \ln 2)}. \end{aligned}$$

Note that for sufficiently large h we have $2^h \ln 2 > e^{h/2}h/2$ and W is an increasing function on $\mathbb{N}$. Hence we have

$$W(2^h \ln 2) > W(e^{h/2}h/2) = h/2.$$

Therefore $SC(h) < 3 \ln 2 \cdot \frac{2^h}{h/2}$. This shows that $SC(h)$ is $O(2^h/h)$.

Conversely we have

$$SC(h) > 2^{2^i+1} = 2 \cdot 2^{\frac{W(2^h \ln 2)}{\ln 2}} = 2 \cdot e^{W(2^h \ln 2)} = 2 \ln 2 \cdot \frac{2^h}{W(2^h \ln 2)}.$$

Since W is an increasing function on reals above 0, for sufficiently large h

$$W(2^h \ln 2) = W(e^{h \ln 2} \ln 2) < W(e^{h \ln 2} h \ln 2) = h \ln 2.$$

This means that $SC(h) > 2 \ln 2 \cdot \frac{2^h}{h \ln 2} = 2 \cdot \frac{2^h}{h}$. Hence $SC(h)$ is $\Omega(2^h/h)$ and the theorem is proved. ∎

State complexity of finite languages with bounded word length

Let $\mathfrak{F}_h$ denote the class of all finite languages whose words have length bounded by h, where $h \geq 0$. Theorem 3.2.4 shows that the state complexity for the class $\mathfrak{F}_h$ is $\Omega(2^h/h)$. We now show that the state complexity for $\mathfrak{F}_h$ is also $\Theta(2^h/h)$.

Let L be a finite language in $\mathfrak{F}_h$. Fix a new symbol $\sigma \notin \Sigma$. We define the uniform-length language L_U of length h as follows

$$L_U = \{w \in \Sigma^* \mid w = x\sigma^{h-|x|}, x \in L\}.$$

Recall for a regular language R, the equivalence relation $\equiv_R$ has a finite index by the Myhill-Nerode theorem and the state complexity of R is the index of $\equiv_R$. In the following we use $[x]_R$ to denote the equivalence class of x with respect to $\equiv_R$.

Lemma 3.2.5 *For a finite language L of length h, we have $SC(L) \leq SC(L_U)$.*

Proof We only need to consider the case when L is not uniform-length. Note that there is exactly one equivalence class $[w]_L$ (resp. $[w]_{L_U}$) with respect to $\equiv_L$ (resp. $\equiv_{L_U}$) containing all words x where $xy \notin L$ (resp $xy \notin L_U$) for any $y \in \Sigma^*$. Now consider an equivalence class $[w]_L$ such that $wx \in L$ for some $x \in \Sigma^*$. Let $\ell([w]_L) = |\{k \in \mathbb{N} \mid y \in [w]_L, |y| = k\}|$. By the choice of w, we have $\ell([w]) > 1$.

Take two words $w_1, w_2 \in [w]_L$. If $|w_1| \neq |w_2|$, by Lemma 3.2.1, w_1, w_2 do not belong to the same equivalence class of $\equiv_{L_U}$. Otherwise, we have $|w_1| = |w_2|$. In this case, since $w_1x \in L$ if and only if $w_2x \in L$ for any $x \in \Sigma^*$, we must also have $w_1x \in L_U$ if and only if $w_2x \in L_U$. Hence w_1 and w_2 belong to the same equivalence classes of $\equiv_{L_U}$. Hence there are $\ell([w]_L)$ distinct equivalence classes of L_U, $[w]^1_{L_U}, \ldots, [w]^{\ell([w]_L)}_{L_U}$ such that

$$[w]_L \subseteq [w]^1_{L_U} \cup \ldots \cup [w]^{\ell([w]_L)}_{L_U}$$

and $[w]_L \cap [w]^i_{L_U} \neq \emptyset$ for each $i \in \{1, \ldots, \ell([w]_L)\}$.

Also note that a word $w' \not\equiv_L w$ cannot be in any of the equivalence classes $[w]^1_{L_U}, \ldots, [w]^{\ell([w]_L)}_{L_U}$. Suppose for a contradiction that $w' \in [w]^i_{L_U}$. Since $w' \not\equiv_L w$, there is some word x such that either $wx \in L$ and $w'x \notin L$ or $wx \notin L$ and $w'x \in L$. Now pick $y \in [w]_L \cap [w]^i_{L_U}$. Note that $y \equiv_{L_U} w'$ and $|y| = |w'|$. Since $y \equiv_L w$, we have either $yx\alpha^{h-|xy|} \in L$ and $w'x\alpha^{h-|xy|} \notin L$ or $yx\alpha^{h-|xy|} \notin L$ and $w'x\alpha^{h-|xy|} \in L$. This contradicts with the assumption that $y \equiv_{L_U} w'$.

Therefore we have $[w]^1_{L_U} \cup \ldots \cup [w]^{\ell([w]_L)}_{L_U} \subseteq [w]_L$ and hence

$$[w]_L = [w]^1_{L_U} \cup \ldots \cup [w]^{\ell([w]_L)}_{L_U}.$$

The above arguments show that every equivalence class $[w]_L$ of $\equiv_L$ where $wx \in L$ for some $x \in \Sigma^*$ is partitioned into $\ell([w]_L)$ many equivalence classes of $\equiv_{L_U}$. By Myhill-Nerode theorem, $SC(L) \leq SC(L_U)$. ∎

Note that L_U is a uniform-length language of height h over an alphabet with three symbols. However the new symbol σ is only used to "pad" the words of L and therefore at most $h-1$ states in the minimal automaton recognizing L_U are used achieve this "padding". Indeed each language from the class of languages of the type L_U is of the form $\{w \in (\Sigma \cup \{\sigma\})^h \mid w = u \cdot \sigma^{h-|u|}$ and $u \in \Sigma^*\}$. Since at most $h-1$ states are used to recognize the suffix of the form σ^*, we conclude that the state complexity of the class of languages of the type L_U is still $\Theta(2^h/h)$. Hence we have the following theorem.

Theorem 3.2.6 *The state complexity of the class of finite languages over a binary alphabet whose words have length bounded by h, where $h = 2^i + i$ for some $i \geq 0$, is $\Theta(2^h/h)$.*

Remark The above technique can be easily adapted to the cases when $|\Sigma| = 2^{2^\ell}$ for some $\ell \geq 0$. In this case let $k = |\Sigma|$. Using a very similar argument, we could show the following: Suppose $h = k^i \log_2 k + i$ for some $i \geq 0$. The state complexity of the class of finite languages whose words have length bounded by h is $\Theta(k^h/h)$.

3.2.2 Union and intersection of uniform-length languages

In this section we analyze the number of states needed to recognize the union and intersection of two uniform-length languages L_1 and L_2. We do not assume the alphabet Σ has size 2 and let $k = |\Sigma|$. Let $\mathcal{M}_1$ (m states) and $\mathcal{M}_2$ (n states) be the minimal automata recognizing L_1 and L_2 respectively. Clearly the product automaton $\mathcal{M}_1 \oplus \mathcal{M}_2$ or $\mathcal{M}_1 \otimes \mathcal{M}_2$ has $m \cdot n$ states. Our goal is to show that we can reduce the number of states of the product automata for L_1 and L_2.

The next lemma presents automata that recognize the union and intersection of two uniform-length languages. By Lemma 3.2.1 $\mathcal{M}_1$ and $\mathcal{M}_2$ are both level automata, say with heights h_1 and h_2 respectively. Without loss of generality, we assume $h_1 \leq h_2$. Let m_i (resp. n_i) be the number of states in $\mathcal{M}_1$ (resp. $\mathcal{M}_2$) at level i for $i \in \{0, \ldots, \min\{h_1, h_2\}\}$.

Lemma 3.2.7 *There exist level automata $\mathcal{M}_\cup$ and $\mathcal{M}_\cap$ that recognize $L_1 \cup L_2$ and $L_1 \cap L_2$ respectively whose size is at most $\sum_{i=0}^{h_1} m_i \cdot n_i + m + n - 2$ where m_i, n_i are the number of states at level i of $\mathcal{M}_1, \mathcal{M}_2$ respectively.*

Proof Let $S_{i,j}$ be the set of all states in $\mathcal{M}_i$ that are on level j where $0 \leq j \leq h_i$. The set of states S of automaton $\mathcal{M}_\cup$ is

$$\bigcup_{i=0}^{h_1} S_{1,i} \times S_{2,i} \cup (S_1 \setminus \{s_0^1\}) \times \{s_2\} \cup \{s_1\} \times (S_2 \setminus \{s_0^2\})$$

where s_1 and s_2 are the reject state of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively.

The state (s_0^1, s_0^2) is the initial state and the only state at level 0 of $\mathcal{M}_\cup$. The transition function Δ of $\mathcal{M}_\cup$ is defined as

$$\Delta((s_1, s_2), \sigma) = (\Delta(s_1, \sigma), \Delta(s_2, \sigma))$$

for $s_1 \in S_1$ and $s_2 \in S_2$. The accepting states of $\mathcal{M}_\cup$ are all states in $S \cap ((F_1 \times S_2) \cup (S_1 \times F_2))$. It is easy to see that the language of $\mathcal{M}_\cup$ contains all words that are recognized by either $\mathcal{M}_1$ or $\mathcal{M}_2$.

The automaton $\mathcal{M}_\cap$ is defined in the same way except the accepting states are $S \cap (F_1 \times F_2)$. ∎

Since $m = \sum_{i=0}^{h_1} m_i$, we have

$$\begin{aligned} \sum_{i=0}^{\min\{h_1,h_2\}} m_i \cdot n_i &= m_1 n_1 + m_2 n_2 + \ldots + m_{h_1} n_{h_1} \\ &\leq m \cdot \max\{n_i \mid 0 \leq i \leq h_1\}. \end{aligned}$$

Lemma 3.2.8 *The maximal n_i for $0 \leq i \leq h_2$ is at most $\frac{k-1}{k} n$.*

Proof Let i be the level where n_i is maximal. Note that $n_{j+1} \leq k n_j$ for every $0 \leq j < h_2$. Hence at level j of the automaton $\mathcal{M}_2$ where $j < i$, there are at least $\left\lceil \frac{n_i}{k^{i-j}} \right\rceil$ states. Therefore

the automaton contains at least

$$\begin{aligned} & n_i + \left\lceil \frac{n_i}{k} \right\rceil + \left\lceil \frac{n_i}{k^2} \right\rceil + \ldots 1 \\ \geq\ & n_i + \frac{n_i}{k} + \frac{n_i}{k^2} + \ldots + 1 \\ =\ & \frac{kn_i - 1}{k-1} \end{aligned}$$

number of states. Note that the above calculation does not take into account the reject state. Hence

$$\begin{aligned} n &\geq \frac{kn_i - 1}{k-1} + 1 \\ &\geq \frac{kn_i}{k-1}. \end{aligned}$$

Therefore $n_i \leq \frac{k-1}{k} n$. ∎

Combining the two lemmas above we get the following.

Theorem 3.2.9 *The state complexity of union and intersection for two uniform-length languages is at most* $\frac{k-1}{k}mn + m + n - 2$ *where* $k = |\Sigma|$ *and* m, n *are the number of states in the input minimal automata.*

3.2.3 Union and intersection of finite word languages

This section focuses on the state complexity of the union and intersection operation for finite word-languages in general. If L is a finite language, the minimal automaton $\mathcal{M}$ recognizing L contains exactly one self-loop. We single out such automata in the next definition.

Definition 3.2.2 *A* acyclic DFA (ADFA) *is a DFA* $\mathcal{M} = (S, q_0, \Delta, F)$ *that has the following properties:*

1. *There is a state* $s \in S$*, called the* reject state*, such that* $\Delta(s, \sigma) = s$ *for all* $\sigma \in \Sigma$*, and*

2. *Let* $E \subseteq S^2$ *be the edge relation such that* $(s_1, s_2) \in E$ *if and only if* $\Delta(s_1, \sigma) = s_2$ *for some* $\sigma \in \Sigma$*. The graph* $(S \setminus \{s\}, E \restriction S \setminus \{s\})$ *is a directed acyclic graph.*

Intuitively, the above definition states that the transition diagram of $\mathcal{M}$, minus the only self-loop s, is acyclic. We omit the reject state from our calculations that follow.

Let L_1 and L_2 be two finite word-languages recognized by ADFA $\mathcal{M}_1$ (m states) and $\mathcal{M}_2$ (n states) respectively. We adopt the following plan in analyzing the number of states needed to recognize $L_1 \cup L_2$ and $L_1 \cap L_2$. First we take the product automata $\mathcal{M}_1 \oplus \mathcal{M}_2$ and $\mathcal{M}_1 \otimes \mathcal{M}_2$. We then compute a lower bound on the number of states that are unreachable by the product automata when processing any input words. Then using this bound we will compute upper bounds for the minimal automata of $L_1 \cup L_2$ and $L_1 \cap L_2$.

First we introduce some terminology. Let $\mathcal{M} = (S, s_0, \Delta, F)$ be an ADFA. The *low-level* of a state $s \in S$ is the length of the shortest path from s_0 to s. The *high-level* of a state $s \in S$ is the length of the longest path from s_0 to s. The *height* of $\mathcal{M}$ is the maximal high-level of any state. Let h be the height of $\mathcal{M}$. A *witness path* is a transition path $s_0, s_1, \ldots, s_h$ of length h.

For the rest of the section we fix two ADFA $\mathcal{M}_1 = (S_1, s_0^1, \Delta_1, F_1)$, $\mathcal{M}_2 = (S_2, s_0^2, \Delta_2, F_2)$ recognizing finite languages L_1 and L_2 respectively. We say a state (s_1, s_2) is *unreachable* if the product automaton cannot reach this state upon processing any input word.

Lemma 3.2.10 *In the product automaton, any state (s_1, s_2), where $s_1 \in S_1$, $s_2 \in S_2$ and the high-level of s_1 (resp. s_2) is less than the low-level of s_2 (resp. s_1), is not reachable.*

Proof Suppose the high-level of $s_1 \in S_1$ is less than the low-level of $s_2 \in S_2$, and the state (s_1, s_2) is reachable in the product automaton via a path $q_0, q_1, \ldots, q_\ell$ where $q_0 = (s_0^1, s_0^2)$. Then the sequence of the first components of $q_0, q_1, \ldots, q_\ell$ is a path in $\mathcal{M}_1$ from s_0^1 to s_1, and the sequence of second components of $q_0, q_1, \ldots, q_\ell$ is a path in $\mathcal{M}_2$ from s_0^2 to s_2. Note by definition the length ℓ of this path is at most the high level of s_1. Also ℓ must be greater than or equal to the low level of s_2. However this is impossible since the high-level of s_1 is less than the low-level of s_2 and hence (s_1, s_2) cannot be reachable in the product automaton. The case when the high-level of s_2 is less than the low-level of s_1 is proved similarly. ∎

The following lemma computes the number of unreachable states associated with a single state in $\mathcal{M}_1$ and $\mathcal{M}_2$.

Lemma 3.2.11

1. *For each state $s_2 \in S_2$ with high-level $i \geq 1$, the number of $s_1 \in S_1$ such that (s_1, s_2) is unreachable is at least $m - \sum_{j=0}^{i} k^j$.*
2. *For each state $s_1 \in S_1$ with high-level $i \geq 1$, the number of $s_2 \in S_2$ such that (s_1, s_2) is unreachable is at least $n - \sum_{j=0}^{i} k^j$.*

Proof We only prove the first part of the lemma. The second part can be proved in the same way. Fix a state $s_2 \in S_2$ with high-level $i \geq 1$. By Lemma 3.2.10, for any state $s_1 \in S_1$ with low-level smaller than i, the state (s_1, s_2) is unreachable in the product automaton. There are at most k^j states with low-level j. This means at least

$$m - k^0 - k^1 - k^2 - \ldots - k^i = m - \sum_{j=0}^{i} k^j$$

states of the form (s_1, s_2) in the product automaton are not reachable. ∎

In the following, we assume that the heights of the two automata $\mathcal{M}_1$ and $\mathcal{M}_2$ are h_1 and h_2 respectively, and that $h_1 \leq h_2$. Note that since there are at most 2^i states with each low-level i, we have $\log_k(m+1) - 1 \leq h_1$, and $\log_k(n+1) - 1 \leq h_2$.

Lemma 3.2.12 *The number of unreachable states in the product automaton of $\mathcal{M}_1$ and $\mathcal{M}_2$ is at least* $(\log_k(m+1))(m+n) - 3m - n - 2$.

Proof Let $P_1 = s_0, s_1, \ldots, s_{h_1}$ and $P_2 = q_0, q_1, \ldots, q_{h_2}$ be witness paths in $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively. Note that each state (s_i, q_i) has high-level i, as otherwise there would be longer paths in $\mathcal{M}_1$ or $\mathcal{M}_2$, which contradicts with the definition of a witness path.

By Lemma 3.2.11 and the fact that $\log_k(m+1) - 1 \leq h_1 \leq h_2$, for each $i \leq \log_k(m+1) - 1$, there are at least $m - \sum_{j=0}^{i} k^j$ unreachable states of the form (s, q_i) in the product automaton. Therefore the total number of unreachable states of the form (s, q_i) where $0 \leq i \leq h_2$ is at least

$$\begin{aligned}
\sum_{i=1}^{\log_k(m+1)-1} \left(m - \sum_{j=0}^{i} k^j \right) &= (\log_k(m+1))m - m - \sum_{i=1}^{\log_k(m+1)-1} \sum_{j=0}^{i} k^j \\
&\geq (\log_k(m+1))m - m - \sum_{i=1}^{\log_k(m+1)-1} k^{i+1} \\
&\geq (\log_k(m+1))m - m - k^{\log_k(m+1)} \\
&\geq (\log_k(m+1))m - 2m - 1 \qquad (3.4)
\end{aligned}$$

Similarly, by Lemma 3.2.11, for each $i \leq \log_k(m+1) - 1$, there are at least $n - \sum_{j=0}^{i} k^j$ unreachable states of the form (s_i, q) in the product automaton. Hence the total number of

unreachable states of the form (s_i, q) where $1 \leq i \leq h_1$ is at least

$$\sum_{i=1}^{\log_k(m+1)-1} \left(n - \sum_{j=0}^{i} k^j \right).$$

The above is greater than

$$(\log_k(m+1))n - n - m - 1. \tag{3.5}$$

Summing up the values in (3.4) and (3.5), the total number of unreachable states is at least

$$(\log_k(m+1))(m+n) - 3m - n - 2.$$

Note that no state (s_i, q_i) is counted twice in the above sum due to the nature of the states counted as unreachable within lemma 3.2.11. ∎

The next theorem directly follows from the above lemma.

Theorem 3.2.13 *Let $\mathcal{M}_1, \mathcal{M}_2$ be two ADFA over an alphabet of k symbols with m, n states respectively. The number of states in the minimal automaton recognizing $L(\mathcal{M}_1) \cup L(\mathcal{M}_2)$ (and the minimal automaton recognizing $L(\mathcal{M}_1) \cap L(\mathcal{M}_2)$) is at most $m \cdot n - \log_k(m)(m+n) + 3m + n + 2$ states.*

3.2.4 Union and intersection of finite tree languages

Recall the definitions of Σ-labeled k-ary trees and deterministic tree automata (DTA) that we introduced in chapter 2 (section 2.1.2). Also recall that the class of regular tree languages, just like regular word languages, is closed under union and intersection. Similar to the case of regular word languages, given DTA's $\mathcal{M}_1, \mathcal{M}_2$, we may construct the product automata $\mathcal{M}_1 \oplus M_2$ and $\mathcal{M}_1 \otimes \mathcal{M}_2$ which recognize $L(\mathcal{M}_1) \cup L(\mathcal{M}_2), L(\mathcal{M}_1) \otimes L(\mathcal{M}_2)$ respectively. The state complexity of union and intersection on two tree regular language L_1, L_2 is bounded above by $m \cdot n$ where m, n are the state complexity of L_1, L_2 respectively. In the rest of this section, we show that this upper bound can be improved for the class of finite tree languages.

A *unary tree language* consists of Σ-labeled k-ary trees where the alphabet set $\Sigma = \{1\}$. The corresponding tree automata are called *unary tree automata*. Since on a $\{1\}$-labeled tree

(t, λ) all elements have the same label, transition functions for unary tree automata can be simplified as $\Delta : (Q \cup \{q_0\})^k \to Q$. Also we identify (t, λ) with the tree t.

The next lemma demonstrates that any tree language can be coded by a unary tree language while preserving regularity. Furthermore, this coding may be done without making too much sacrifice in state complexity.

Lemma 3.2.14 *For any $T = (t, \lambda) \in \mathcal{T}_k(\Sigma)$, there is a $(k+1)$-ary tree $f(T)$ such that for any tree language $L \subseteq \mathcal{T}_k(\Sigma)$, L is a regular tree language if and only if the set $f(L) = \{f(T) \mid T \in L\}$ is a regular unary tree language. Furthermore, we have $SC(L) \leq SC(f(L)) \leq SC(L) + |\Sigma|$.*

Proof Without loss of generality, assume $\Sigma = \{1, \ldots, m\}$ for some number $m \geq 1$. Take $T = (t, \lambda) \in \mathcal{T}_k(\Sigma)$. The $(k+1)$-ary tree $f(T)$ is defined as the smallest prefix-closed set containing $t \cup \{wk0^{\lambda(w)-1} \mid w \in t\}$.

Consider a regular tree language $L \subseteq \mathcal{T}_k(\Sigma)$ and let $\mathcal{M} = (Q, \Delta, q_0, F)$ be the minimal DTA over Σ such that $L = L(\mathcal{M})$. We define the unary DTA $\mathcal{M}' = (Q', \Delta', q_0, F')$ as follows:

1. $Q' = Q \cup \{s_1, \ldots, s_m\}$ where $s_1, \ldots, s_m \notin Q$ and $F' = F$.

2. The transition function Δ' is defined as follows:

$$\Delta'(p_0, \ldots, p_k) = \begin{cases} s_1 & \text{if } p_0 = \ldots = p_k = q_0 \\ s_{i+1} & \text{if } p_1 = \ldots = p_k = q_0 \text{ and } p_0 = s_i \\ & \text{for } i \in \{1, \ldots, m-1\} \\ \Delta(p_0, \ldots, p_{k-1}, j) & \text{if } p_k = s_j \text{ for } j \in \Sigma \\ & \text{and } p_0, \ldots, p_{k-1} \in Q \end{cases}$$

Consider a k-ary tree $T \in L$ and let $\rho : \widehat{t} \to Q$ be the run of $\mathcal{M}$ on T. Also let $\rho' : \widehat{f(T)} \to Q'$ be the run of $\mathcal{M}'$ on $f(T)$. From the definition of $f(T)$ and the construction of $\mathcal{M}'$, it is clear that $\rho(w) = \rho'(w)$ for every $w \in T$. Hence we see that $L(\mathcal{M}') = f(L)$. Since $|Q'| = SC(L) + |\Sigma|$, we have $SC(f(L)) \leq SC(L) + |\Sigma|$

Now let $\mathcal{H}' = (Q', \Delta', q_0, F')$ be the minimal unary DTA such that $L(\mathcal{H}') = f(L)$. We would like to construct a DTA $\mathcal{H} = (Q, \Delta, q_0, F)$ from $\mathcal{H}'$ such that $L(\mathcal{H}) = L$. However before we do so, we would like to set up some terminology. We use r_i to denote the Σ-labeled k-ary tree (ϵ, λ) such that $\lambda(\epsilon) = i$ for $i \in \Sigma$. We define the function $g : Q' \to \mathcal{P}(\Sigma)$

such that

$$g(q) = \{i \in \Sigma \mid \rho'(k) = q \text{ where } \rho' \text{ is the run of } \mathcal{H}' \text{ on } f(r_i)\}$$

for $q \in Q'$.

Note that since $\mathcal{H}'$ is deterministic, it must be the case that the sets $g(q_1)$ and $g(q_2)$ are disjoint for distinct $q_1, q_2 \in Q'$ and $\bigcup_{q \in Q'} g(q) = \Sigma$. Now we are ready to define $\mathcal{H} = (Q, \Delta, q_0, F)$:

1. $Q = Q'$ and $F = F'$.

2. For every $j \in \Sigma$ and $p_0, \ldots, p_{k-1} \in Q$, we define $\Delta(p_0, \ldots, p_{k-1}, j) = \Delta'(p_0, \ldots, p_{k-1}, q)$ where $q \in Q'$ and $j \in g(q)$.

By the construction of $\mathcal{H}$ and the definition of $f(T)$, it is clear that $\mathcal{H}$ accepts $T \in \mathcal{T}_k(\Sigma)$ if and only if $\mathcal{H}'$ accepts $f(T)$. Since $L(\mathcal{H}') = f(L)$, we have $L(\mathcal{H}) = L$. Also we have $SC(f(L)) = |Q'|$ and $Q = Q'$. Therefore $SC(L) \leq SC(f(L))$ as required. ∎

Note that for any tree languages $L_1, L_2 \in \mathcal{T}_k(\Sigma)$, the languages $f(L_1)$ and $f(L_2)$ satisfy that

$$f(L_1) \cup f(L_2) = f(L_1 \cup L_2) \quad \text{and} \quad f(L_1) \cap f(L_2) = f(L_1 \cap L_2). \tag{3.6}$$

The above equalities will be useful later in this section.

Our goal is to obtain an analogous upper bound of the state complexity of union and intersection on tree languages as the upper bound given in Section 3.2.3. To this end we first study the state complexity of finite unary tree languages. We then use Lemma 3.2.14 to obtain the upper bound for finite tree languages in general. We say a state q in a unary DTA $\mathcal{M}$ is *reachable* if there is a tree t such that the run ρ of $\mathcal{M}$ on t labels the root of t by q. In this case, the run ρ is called the *witness run* of q. Note that a minimal tree automaton does not contain any states that are not reachable.

Let $\mathcal{M} = (S, \Delta, q_0, F)$ be an DTA that recognizes a finite tree language. The *low-level* of a state $q \in S$ is the minimal height of any tree t such that the run of $\mathcal{M}$ on t labels the root of t by q. The *high-level* of a state $q \in S$ is the maximal height of any tree t such that the run of $\mathcal{M}$ on t labels the root of t by q. The *height of* $\mathcal{M}$ is the maximal high-level of any accepting state $q \in F$ in $\mathcal{M}$. In other words, the height of $\mathcal{M}$ is the maximal height of any tree in $L(\mathcal{M})$.

Let L_1 and L_2 be two finite tree languages recognized by DTA $\mathcal{M}_1$ (m states) and $\mathcal{M}_2$ (n states) respectively. Similar to the word automata case, we again would like to compute

a lower bound on the number of states that are not reachable by the product automata $\mathcal{M}_1 \oplus \mathcal{M}_2$ and $\mathcal{M}_1 \otimes \mathcal{M}_2$. Suppose that $\mathcal{M}_1 = (S_1, \Delta_1, q_0, F_1)$ and $\mathcal{M}_2 = (S_2, \Delta_2, q_0, F_2)$. The following lemma can be proved in a similar way as Lemma 3.2.10.

Lemma 3.2.15 *In the product automaton, any state $(q_1, q_2) \in S_1 \times S_2$ where the high-level of q_1 (resp. q_2) is less than the low-level of q_2 (resp. q_1), is not reachable.*

Proof Suppose that the high-level of q_1 is less than the low-level of q_2, and (q_1, q_2) is reachable in the product automaton via a tree t. Let $\rho : \widehat{t} \to (Q_1 \cup \{q_0\}) \times (Q_2 \cup \{q_0\})$ be the witness run of (q_1, q_2). Let $\rho_1 : \widehat{t} \to Q_1 \cup \{q_0\}$ and $\rho_2 : \widehat{t} \to Q_2 \cup \{q_0\}$ be such that for each $w \in \widehat{t}$, we have $\rho(w) = (\rho_1(w), \rho_2(w))$. Then the functions ρ_1 and ρ_2 are witness runs of q_1 in $\mathcal{M}_1$ and q_2 in $\mathcal{M}_2$ respectively. By definition of low-level and high-level, the *height* of the tree t must be no less than the low-level of q_2, and no more than the high-level of q_1. However, this contradicts with the assumption. The case when the high-level of q_2 is less than the low-level of q_1 can be proved in the same way. ∎

As in the case of finite word automata, we let h_1, h_2 be the heights of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively and assume that $h_1 \leq h_2$.

Lemma 3.2.16 *Suppose $\mathcal{M}_1$ and $\mathcal{M}_2$ are unary rank-k DTAs.*

1. *For each state $s_1 \in S_1$ with high-level i where $1 \leq i \leq h_1$, the number of $s_2 \in S_2$ such that (s_1, s_2) is not reachable is at least $n - \sum_{j=0}^{i} 2^{k^{j+1}}$.*

2. *For each state $s_2 \in S_2$ with high-level i where $1 \leq i \leq h_2$, the number of $s_1 \in S_1$ such that (s_1, s_2) is not reachable is at least $m - \sum_{j=0}^{i} 2^{k^{j+1}}$.*

Proof We only prove the first part of the lemma. Note that by definition each k-ary tree t of height i is a subset of $\bigcup_{j=0}^{i} \mathbb{N}_k^j$. Since there are at most

$$1 + k + k^2 + \ldots + k^i = \frac{k^{i+1} - 1}{k - 1} \leq k^{i+1}$$

elements in the set $\bigcup_{j=0}^{i} \mathbb{N}_k^j$, there are at most $2^{k^{i+1}}$ number of k-ary trees of height i. Now fix a state $s_1 \in S_1$ with high-level $i \geq 1$. There are at least

$$n - 2^{k^1} - 2^{k^2} - \ldots - 2^{k^{i+1}} = n - \sum_{j=0}^{i} 2^{k^{j+1}}$$

states of the form $(s_1, s_2) \in S_1 \times S_2$ where s_2 has low-level smaller than i. By Lemma 3.2.15, all these states (s_1, s_2) are not reachable in the product automaton. Similarly, one may prove that for $s_2 \in S_2$ with high-level $i \geq 1$, the number of $s_1 \in S_1$ such that (s_1, s_2) is not reachable is at least $m - \sum_{j=0}^{i} 2^{k^{j+1}}$. ∎

Lemma 3.2.17 *The number of unreachable states in the product automaton $\mathcal{M}_1$ and $\mathcal{M}_2$ is at least* $(\log_k \log_2(m-1))(m+n) - 9m - n + 8$.

Proof Note that the number of states with high-level i, where $0 \leq i \leq h_1$, is bounded above by the number of k-ary trees of height i. Therefore there are at most $2^{k^{i+1}}$ states in $\mathcal{M}_1$ with high-level i. Hence, we have

$$\sum_{i=0}^{h_1} 2^{k^{i+1}} \geq m.$$

Since $\sum_{i=0}^{h_1} 2^{k^{i+1}} < 2^{k^{i+1}+1}$, we have

$$\begin{aligned} 2^{k^{h_1+1}+1} &\geq m \\ k^{h_1+1} &\geq \log_2(m-1) \\ h_1 + 1 &\geq \log_k \log_2(m-1) \\ h_1 &\geq \log_k \log_2(m-1) - 1. \end{aligned}$$

Similarly, one can prove that $h_2 \geq \log_k \log_2(n-1) - 1$.

Let $t_1 \in L_1$ and $t_2 \in L_2$ be trees with height h_1 and h_2 respectively. Let ρ_1 and ρ_2 be witness runs for t_1 and t_2 by $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively. Note that for any words $u_1, u_2 \in t_\alpha$ where $\alpha \in \{1, 2\}$ and $u_1 \prec_{\mathsf{pref}} u_2$, $\rho_\alpha(u_1) \neq \rho_\alpha(u_2)$ as otherwise one may apply a pumping argument to obtain infinitely many trees in the tree language L_α.

Let w be a word in t_2 of length h_2 and let $s_0, s_1, \ldots, s_{h_2}$ be the sequence of states labeled by ρ_2 on the path from the root ε to w in t_2. Note that each s_i (where $0 \leq i \leq h_2$) must has high-level i as otherwise $\mathcal{M}_2$ must have height greater than h_2. By Lemma 3.2.16, for each i where $0 \leq i \leq \log_k \log_2(m-1) - 1$, there are at least $m - \sum_{j=0}^{i} 2^{k^{j+1}}$ states of the form (q, s_i) that are not reachable in the product automaton. Since $\log_k \log_2(m-1) - 1 \leq h_1 \leq h_2$, the

total number of states of the form (q, s_i) is at least

$$\sum_{i=1}^{\log_k \log_2(m-1)-1} \left(m - \sum_{j=0}^{i} 2^{k^{j+1}} \right)$$
$$= (\log_k \log_2(m-1) - 1)m - \sum_{i=1}^{\log_k \log_2(m-1)-1} \sum_{j=0}^{i} 2^{k^{j+1}}.$$

Since $\sum_{j=0}^{i} 2^{k^{j+1}} < \sum_{j=0}^{k^{i+1}} 2^j < 2^{k^{i+1}+1}$, the above is greater than

$$(\log_k \log_2(m-1) - 1)m - \sum_{i=1}^{\log_k \log_2(m-1)-1} 2^{k^{i+1}+1}$$
$$\geq (\log_k \log_2(m-1) - 1)m - 2^{k^{\log_k \log_2(m-1)}+2}$$
$$= (\log_k \log_2(m-1))m - 5m + 4. \tag{3.7}$$

Similarly, let w be a word in t_1 of length h_1 and let $q_0, q_1, \ldots, q_{h_1}$ be the sequence of states labeled by ρ_1 on the path from the root to w in t_1. One may show that the number of states of the form (q_i, s) that is not reachable in the product automaton, where $1 \leq i \leq \log_k \log_2 m$ is at least

$$(\log_k \log_2(m-1))n - n - 4m + 4. \tag{3.8}$$

Summing up the expressions (3.7) and (3.8) we obtain the desired bound of

$$(\log_k \log_2(m-1))(m+n) - 9m - n + 8.$$

Again note that no state (s_i, q_i) is counted twice in the above sum due to the nature of the states that are counted as unreachable in lemma 3.2.16. ∎

The next theorem directly follows from the lemma above.

Theorem 3.2.18 *Let $\mathcal{M}_1$ (m states) and $\mathcal{M}_2$ (n states) be two unary rank k DTA recognizing finite tree languages. The number of states in the minimal automaton recognizing $L(\mathcal{M}_1) \cup L(\mathcal{M}_2)$ (and the minimal automaton recognizing $L(\mathcal{M}_1) \cap L(\mathcal{M}_2)$) is at most $m \cdot n - (\log_k \log_2(m-1))(m+n) + 9m + n$, where m, n are the number of states in $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively.*

By Lemma 3.2.14, the above upper bound also holds for tree languages where the alphabet Σ contains more than 1 letter.

Corollary 3.2.19 *Let $\mathcal{M}_1$ (m states) and $\mathcal{M}_2$ (n states) be two DTA of rank k recognizing finite tree languages. The number of states in the minimal automata recognizing $L(\mathcal{M}_1) \cup L(\mathcal{M}_2)$ and $L(\mathcal{M}_1) \cap L(\mathcal{M}_2)$) is at most $m \cdot n - c(\log_{k+1} \log_2(m))(m+n)$ for some constant $c > 0$ when m, n are sufficiently large.*

Proof Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be minimal DTAs with rank k recognizing finite tree languages L_1, L_2 respectively. By Lemma 3.2.14, the finite unary tree languages $f(L_1)$ and $f(L_2)$ have rank $k+1$ and the minimal automaton recognizing $f(L_1)$ (resp. $f(L_2)$) is at most $m+k$ (resp. $n+k$). By Theorem 3.2.18, the minimal automaton recognizing $f(L_1) \cup f(L_2)$ (and the one recognizing $f(L_1) \cap f(L_2)$) has at most

$$\begin{aligned}&(m+k) \cdot (n+k) - (\log_{k+1} \log_2(m+k-1))(m+n+2k) + 9m + n + 10k \\ &\leq mn - (\log_{k+1} \log_2(m+k-1))(m+n+2k) + k(m+n) + 9m + n + 10k + k^2\end{aligned}$$

states. When m, n are sufficiently large, the above expression is bounded from above by $mn - c(\log_{k+1} \log_2 m)(m+n)$ for some constant $c > 0$. ∎

Chapter 4

Finite automata over structures

In this chapter we generalize finite automata to operate over arbitrary algebraic structures. As mentioned in chapter 1 (section 1.1.2), there are two main motivations for generalizing the automata model. The first motivation comes from the observation that algebraic structures are an apt way of capturing the underlying domain, operations and predicates of an algorithm. Some examples which are inspired by this observation are [9, 12, 13, 24, 48, 60, 65]. The second motivation comes from the fields of program verification and databases where various models of automata operating over infinite alphabets have been proposed [4, 10, 11, 34, 71, 83, 87]. The existence of many such models of automata over either structures or infinite alphabets calls for a general yet simple framework to formally reason about such finite state automata. In this chapter we address this issue and suggest one such framework.

4.1 The Automata Model

In this section, we introduce the notion of finite automata over algebraic structures which accept or reject finite sequences of elements from the domain of the underlying structure. Our main motivation here is that our model is the finite automata analogue of BSS machines over arbitrary structures. Namely, we define finite state automata over any given structure $\mathcal{S}$. Such an automaton is equipped with a finite number of states, a fixed number of registers, a read only head that always moves to the right in the tape and transitions between the states.

Generally speaking, the structures under consideration can be arbitrary structures.

Therefore the operations and relations are not necessarily computable. However, we will always assume that given two elements x_1, x_2 in the domain, computing the value of $f_i(x_1, x_2)$ as well as checking $R_j(x_1, x_2)$ can be carried out effectively for all i and j. Recall that we denote the set of all atomic operations and the set of all atomic relations of $\mathcal{S}$ by $\mathsf{Op}(\mathcal{S})$ and $\mathsf{Rel}(\mathcal{S})$, respectively.

Definition 4.1.1 *A* D-word *of length* t *is a sequence* $a_1 \dots a_t$ *of elements in the domain* D*. A* D-language *is a set of* D*-words.*

Given a structure $\mathcal{S}$ with domain D, we investigate a certain class of programs that process D-words. Informally, such a program reads a D-word as input while updating a fixed number of registers. Each register holds an element in D at any given time. Whenever the program reads an element from the input D-word, it first checks if some atomic relations hold on this input element and the current values of the registers, then applies some atomic operations to update the registers. The program stops when the last element in the D-word is read. If the structure $\mathcal{S}$ is finite, then our model is simply finite automata whose alphabet is the domain of $\mathcal{S}$. In this sense, our model is a finite automata model of BSS machines.

We model such programs using finite state machines and call our model $(\mathcal{S}, k)$-automata ($k \in \mathbb{N}$). An $(\mathcal{S}, k)$-automaton keeps k *changing registers* as well as ℓ *constant registers*. The ℓ constant registers store the constants $\overline{c} = c_1, \dots, c_\ell$ and their values are fixed. Each changing register stores an element of D at any time. We normally use $m_1, \dots, m_k$ to denote the current values of the changing registers. Inputs to the automaton are written on a one-way read-only tape. Every state q is associated with $k + \ell$ atomic relations $P_1, \dots, P_{k+\ell} \in \mathsf{Rel}(\mathcal{S})$. Whenever the state q is reached, the $(\mathcal{S}, k)$-automaton reads the next element x of the input D-word and tests the predicate $P_i(x, m_i)$ for each $i \in \{1, \dots, k\}$ and $P_{k+j}(x, c_j)$ for each $j \in \{1, \dots, \ell\}$. The $(\mathcal{S}, k)$-automaton then chooses a transition depending on the outcome of the tests and moves to the next state. Each transition is labelled with k operations, say $g_1, \dots, g_k \in \mathsf{Op}(\mathcal{S})$. The automaton changes the value of its ith register from m_i to $g_i(m_i, x)$. After all elements on the input tape have been read, the $(\mathcal{S}, k)$-automaton stops and decides whether to accept the input depending on the current state. Here is a formal definition.

Definition 4.1.2 *An* $(\mathcal{S}, k)$-automaton *is a tuple* $\mathcal{A} = (Q, \alpha, \overline{x}, \Delta, q_0, F)$ *where* Q *is a finite set of states, the mapping* α *is a function from* Q *to* $\mathsf{Rel}^{k+\ell}(\mathcal{S})$*,* $\overline{x} \in D^k$ *are the initial values of the registers,* $q_0 \in Q$ *is the initial state,* $F \subseteq Q$ *is the set of accepting states and* $\Delta \subseteq Q \times \{0,1\}^{k+\ell} \times Q \times \mathsf{Op}^k(\mathcal{S})$ *is the transition relation of* $\mathcal{A}$*. The* $(\mathcal{S}, k)$*-automaton is* deterministic *if for each* $q \in Q$*,* $\overline{b} \in \{0,1\}^{k+\ell}$*,*

there is exactly one q' and $\overline{g} \in \mathsf{Op}^k(\mathcal{S})$ such that $(q, \overline{b}, q', \overline{g}) \in \Delta$. An (deterministic) $\mathcal{S}$-automaton *is an (deterministic)* $(\mathcal{S}, k)$*-automaton for some k.*

One can view each state q of an $\mathcal{S}$-automaton as a test state and an operational state; the state q is a test state because the predicates from $\alpha(q)$ are tested on tuples of the form (a, m) where a is the input and m is a value from the registers. The state q is an operational state because depending on the outcomes of the tests, an appropriate list of operations are applied to the tuples (m, a).

To define runs of $\mathcal{S}$-automata, we introduce the following notations. For any $n \in \mathbb{N}$, given a tuple $\overline{P} = (P_1, \ldots, P_n) \in \mathsf{Rel}^n(\mathcal{S})$, $\overline{m} = (m_1, \ldots, m_n) \in D^n$ and $a \in D$, we let $\chi(\overline{P}, \overline{m}, a) = (b_1, \ldots, b_n) \in \{0, 1\}^n$ such that $b_i = 1$ if $\mathcal{S} \models P_i(a, m_i)$ and $b_i = 0$ otherwise, where $1 \leq i \leq n$. Fix an $(\mathcal{S}, k)$-automaton $\mathcal{A}$. A *configuration* of $\mathcal{A}$ is a tuple $r = (q, \overline{m}) \in Q \times D^k$. Given two configurations $r_1 = (q, \overline{m})$, $r_2 = (q', \overline{m'})$ and $a \in D$, by $r_1 \hookrightarrow_a r_2$ we denote that $(q, \chi(\alpha(q), (\overline{m}, \overline{c}), a), q', g_1, \ldots, g_k) \in \Delta$ and $m'_i = g_i(m_i, a)$ for all $i \in \{1, \ldots, k\}$. Note that if g_i is a partial operation, then it must be defined for the arguments m_i, a for $r_1 \hookrightarrow_a r_2$.

Definition 4.1.3 *A* run *of $\mathcal{A}$ on a D-word $a_1 \ldots a_n$ is a sequence of configurations*

$$r_0, r_1, \ldots, r_n$$

where $r_0 = (q_0, x_1, \ldots, x_k)$ and $r_{i-1} \hookrightarrow_{a_i} r_i$ for all $i \in \{1, \ldots, n\}$. The run is accepting *if the state in the last configuration r_n is accepting. The $(\mathcal{S}, k)$-automaton $\mathcal{A}$* accepts *the D-word $a_1 \ldots a_n$ if $\mathcal{A}$ has an accepting run on $a_1 \ldots a_n$. The* language *$L(\mathcal{A})$ of the automaton is the set of all D-words accepted by $\mathcal{A}$.*

We say a D-language L is *(deterministic) $\mathcal{S}$-automata recognizable* if $L = L(\mathcal{A})$ for some (deterministic) $\mathcal{S}$-automata $\mathcal{A}$. The subsequent sections present several examples of $\mathcal{S}$-automata recognizable languages and discuss some simple properties of $\mathcal{S}$-automata. These examples and properties provide justification to investigate $\mathcal{S}$-automata as a general framework for finite state machines.

4.2 Simple Properties of $\mathcal{S}$-automata

In this section we use several examples to establish some simple properties of $\mathcal{S}$-automata and $\mathcal{S}$-automata recognizable D-languages.

$\mathcal{S}$-automata and regular languages: Let $\mathcal{S}$ be a structure. We use $\mathcal{S}[\overline{a}]$ to denote the structure obtained from $\mathcal{S}$ by adding constants $\overline{a}$ to the signature. Suppose that the structure $\mathcal{S}$ contains an atomic equivalence relation $\equiv$ of finite index. Let $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ be the set of all equivalence classes of $\equiv$. For every word $w = w_1 \ldots w_n$ over the alphabet Σ, let $R(w)$ be the D-language $\{a_1 \ldots a_n \mid a_i \in w_i \text{ for all } i = 1, \ldots, n\}$. For every language $\mathcal{L}$ over Σ, let $R(\mathcal{L})$ be the D-language $\bigcup_{w \in \mathcal{L}} R(w)$.

Theorem 4.2.1 *Let a_i be an element from the $\equiv$-equivalence class σ_i, where $i = 1, \ldots, k$. Then each of the following is true.*

- *For every regular language $\mathcal{L}$ over Σ, the D-language $R(\mathcal{L})$ is accepted by an $(\mathcal{S}[a_1, \ldots, a_k], 0)$-automaton.*
- *Suppose the signature of $\mathcal{S}$ contains only one relation $\equiv$ and the atomic operations of $\mathcal{S}$ are compatible with $\equiv$. For every $\mathcal{S}$-automata recognizable D-language W, there is a regular language $\mathcal{L}$ over Σ such that $W = R(\mathcal{L})$.*

Proof For (1), let $\mathcal{L}$ be a regular language over the alphabet Σ. Take a deterministic finite automaton $\mathcal{A}$ accepting $\mathcal{L}$. We define an $(\mathcal{S}[a_1, \ldots, a_k], 0)$-automaton $\mathcal{A}'$ accepting the D-language $R(\mathcal{L})$ as follows.

The automaton $\mathcal{A}'$ has all states in $\mathcal{A}$ with a new state q_{sink}. The initial state and the accepting states of $\mathcal{A}'$ coincide with those states of $\mathcal{A}$. The map α associates with each state of $\mathcal{A}'$ the tuple $(\equiv, \ldots, \equiv)$. The transitions in $\mathcal{A}'$ are the following.

- A transition $(q_{\mathsf{sink}}, \overline{b}, q_{\mathsf{sink}})$ for every $\overline{b} \in \{0, 1\}^k$.
- A transition $(q, (b_1, \ldots, b_k), q')$ where (q, σ_i, q') is a transition of $\mathcal{A}$ with $b_i = 1$ and $b_j = 0$ for all $j \neq i$.
- A transition $(q, (b_1, \ldots, b_k), q_{\mathsf{sink}})$ for every state q in $\mathcal{A}$ where $|\{i \mid b_i = 1\}| \neq 1$.

It is easy to see that $\mathcal{A}'$ is the desired $(\mathcal{S}[a_1, \ldots, a_k], 0)$-automaton.

For (2), let W be a D-language accepted by an $(\mathcal{S}, t)$-automaton $\mathcal{A} = (Q, \alpha, \overline{x}, \Delta, q_0, F)$. We construct a finite automaton $\mathcal{A}'$ for the desired $\mathcal{L}$.

- The states of $\mathcal{A}'$ are $Q \times \Sigma^{t+\ell}$ where ℓ is the number of constant symbols in the signature of $\mathcal{S}$.

- The initial state of $\mathcal{A}'$ is $(q_0, \sigma_{i_1}, \ldots, \sigma_{i_{t+\ell}})$ where $x_j \in \sigma_{i_j}$ for $j = 1, \ldots, t$ and $c_j \in \sigma_{i_{t+j}}$ for $j = 1, \ldots, \ell$.
- A state $(q, \sigma_{i_1}, \ldots, \sigma_{i_{t+\ell}})$ is accepting if $q \in F$.
- A transition from $(q, \sigma_{i_1}, \ldots, \sigma_{i_{t+\ell}})$ to $(q', \sigma'_{i_1}, \ldots, \sigma'_{i_{t+\ell}})$ is labelled by $\sigma \in \Sigma$ if there is a transition $(q, \overline{b}, q', \overline{g}) \in \Delta$ and $a \in D$ that satisfy the following conditions:
  1. $a \in \sigma$
  2. $b_j = 1$ if and only if $a \in \sigma_{i_j}$ for $j = 1, \ldots, t + \ell$
  3. For any $y \in \sigma_{i_j}$, $g_j(y, a) \in \sigma'_{i_j}$ for $j = 1, \ldots, t$.

For the language $\mathcal{L}$ accepted by the automaton $\mathcal{A}'$ we have $R(\mathcal{L}) = W$. ∎

Example 4.2.1 *Using Theorem 4.2.1 one can present many example of D-languages accepted by $\mathcal{S}$-automata.*

(a) Regular languages over $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ are accepted by $((\Sigma; =, \sigma_1, \ldots, \sigma_k), 0)$-automata.

(b) Let $\mathcal{S}$ be a finite structure with domain D where equality is part of the signature. Any D-language acceptable by an $\mathcal{S}$-automaton is a regular language over the alphabet D.

(c) Let $\mathcal{S}$ be $(\mathbb{Z}; \equiv)$ where $\equiv = \{(i, j) \mid i = j = 0 \text{ or } i, j > 0\}$. The following $\mathbb{Z}$-language is recognized by $(\mathcal{S}[-1, 0, 1], 0)$-automata.

$$\{n_0 \ldots n_k \mid k \in \mathbb{N}, n_j \text{ is positive when } j \text{ is even and negative when } j \text{ is odd}\}$$

Separation between $(\mathcal{S}, k)$-automata and $(\mathcal{S}, k + 1)$-automata: We now single out two functions that will be used throughout the paper.

Definition 4.2.1 *For $i \in \{1, 2\}$, define* projection *on the ith coordinate as an operation $\mathrm{pr}_i : D^2 \to D$ such that $\mathrm{pr}_i(a_1, a_2) = a_i$ for all $a_1, a_2 \in D$.*

The next example shows that for infinite structures of the form $\mathcal{S} = (D; =, \mathrm{pr}_1, \mathrm{pr}_2)$, the class of D-languages accepted by $(\mathcal{S}, k)$-automata properly contains the class of D-languages accepted by $(\mathcal{S}, k - 1)$-automata.

Example 4.2.2 *Let $\mathcal{S} = (D; =, \mathrm{pr}_1, \mathrm{pr}_2)$ with D infinite. For $k > 0$, let D_k be the $\mathcal{S}$-language*

$$\{a_0 \dots a_k \mid \forall i, j \in \{0, \dots, k\} : i \neq j \Rightarrow a_i \neq a_j\}.$$

It is clear that an $(\mathcal{S}, k)$-automaton recognizes D_k: The automaton reads the input a_i and stores a_i into the ith register (by applying pr_1 on the ith register) when $a_i \neq a_j$ for all $j < i$, where $0 \leq i \leq k-1$. The automaton accepts $a_0 \dots a_k$ if and only if a_k is different from any register values.

Now suppose D_k is accepted by an $(\mathcal{S}, k-1)$-automaton $\mathcal{A}$. Fix a D-word $w = a_0 \dots a_{k-1}$ where $a_i \neq a_j$ for any $i \neq j$ and say $\mathcal{A}$ reaches state q after reading w. Since there are only $k-1$ registers, there is some $j \in \{0, \dots, k-1\}$ such that no register stores the element a_j. Now if there is a transition that goes from q to an accepting state q' of the form $(q, \overline{0}, q', \overline{g})$, then the $\mathcal{S}$-automaton $\mathcal{A}$ would accept the D-word $wa_j \notin D_k$. Otherwise all transitions from q to an accepting state are of the form $(q, \overline{b}, q', \overline{g})$ where $\overline{b} = (b_1, \dots, b_{k-1})$ and some b_j in $\overline{b}$ is 1. Therefore if $\mathcal{A}$ reads another input a_k and reaches an accepting state, a_k must equal to some register value. This means that $|\{a_k \mid wa_k \in L(\mathcal{A})\}| \leq k-1$, which contradicts with the assumption that D is infinite.

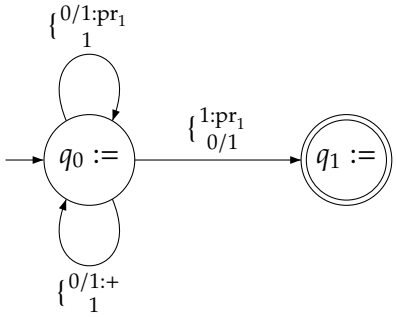
Separation between deterministic and nondeterminstic $\mathcal{S}$-automata: The next example shows that the deterministic $\mathcal{S}$-automata form a proper subclass of $\mathcal{S}$-automata. Furthermore, the class of $\mathcal{S}$-automata recognizable D-language is not closed under Boolean operations in the general case.

Example 4.2.3 *Let $\mathcal{S} = (\mathbb{N}; +, \mathrm{pr}_1, =, 1)$. Let L be the $\mathbb{N}$-language $\{1^n m \mid n, m \in \mathbb{N}, m \leq n\}$. Fig. 4.1 gives an $(\mathcal{S}, 1)$-automaton that recognizes L. We now prove that no deterministic $\mathcal{S}$-automaton recognizes the $\mathbb{N}$-language L. Suppose for a contradiction that a deterministic $(\mathcal{S}, d)$-automaton $\mathcal{A}$ recognizes L. Fix a number $n > k$. The $\mathcal{S}$-automaton $\mathcal{A}$ have exactly one run over the $\mathbb{N}$-word 1^n. Since $n > k$, at the end of this run, there must be one value $m \in \{0, \dots, n\}$ that is not stored by any registers. Since $1^n m \in L$, there must be a transition from the current state to an accepting state that is labelled by 0^{k+1} (which indicates that the input is different from any registers). However, this means that some word $1^n m'$ where $m' > n$ is accepted by $\mathcal{A}$, which is in contradiction with the assumption about $\mathcal{A}$.*

Now consider the $\mathbb{N}$-language $L' = \{1^n m \mid n, m \in \mathbb{N}, m > n\}$. Suppose L' is recognized by some $(\mathcal{S}, k)$-automaton $\mathcal{A}'$. Consider an $\mathbb{N}$-word 1^n where $n \in \mathbb{N}$. Since the only operations of $\mathcal{S}$ are $+$ and pr_1, at the end of any run of $\mathcal{A}'$ on 1^n, no register of $\mathcal{A}'$ would store a value greater than n. Fix a number $n > k$. Take an accepting run of $\mathcal{A}'$ on the $\mathbb{N}$-word $1^n(n+1)$. The last transition of

this run must be labelled by 0^k (which indicates that the input is different from any register values). However, since $n > k$, there is a value $m \in \{0, \ldots, n\}$ that is not stored by any registers before $\mathcal{A}'$ reads the last input and hence the $\mathbb{N}$-word $1^n m$ is also accepted by $\mathcal{A}'$. This contradicts with the assumption about $\mathcal{A}'$. Hence the language L' can not recognized by any $\mathcal{S}$-automaton. Since $\mathbb{N}^* \setminus L = \{\varepsilon\} \cup \{1^n mw \mid n, m \in \mathbb{N}, m \neq 1, w \in \mathbb{N}^+\} \cup L'$, it is easy to see that the class of $\mathcal{S}$-automata recognizable $\mathbb{N}$-languages is not closed under the set operations.

Figure 4.1: An $((\mathbb{N}; +, \mathrm{pr}_1, =, 1), 1)$-automaton accepting the $\mathbb{N}$-language L. The initial value is 0.



4.3 Deterministic $\mathcal{S}$-automata

This section presents examples of deterministic $\mathcal{S}$-automata. An important property of our automata model is that the class of all languages recognized by deterministic $\mathcal{S}$-automata is closed under all the Boolean operations. Furthermore, every language recognized by an $\mathcal{S}$-automaton over a (computable) structure is decidable. Thus deterministic $\mathcal{S}$-automata do not generate undecidable languages. We first show that the class of deterministic $\mathcal{S}$-automata recognizable D-language forms a Boolean algebra. This is a justification to investigate deterministic $\mathcal{S}$-automata as a general framework for finite state machines.

Lemma 4.3.1 (The union lemma) *Given an $(\mathcal{S}, k_1)$-automaton $\mathcal{A}_1$ and an $(\mathcal{S}, k_2)$-automaton $\mathcal{A}_2$ ($k_1, k_2 \in \mathbb{N}$), there exists an $(\mathcal{S}, k_1 + k_2)$-automaton $\mathcal{A}$ accepting the language $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$.*

Proof The construction of the desired automaton $\mathcal{A}$ is very similar to the standard construction for regular languages. Let $\mathcal{A}_1 = (Q_1, \alpha_1, \overline{x}, \Delta_1, p_0, F_1)$, $\mathcal{A}_2 = (Q_2, \alpha_2, \overline{y}, \Delta_2, q_0, F_2)$. The $(\mathcal{S}, k_1 + k_2)$-automaton $\mathcal{A}$ is

$$(Q_1 \times Q_2, \alpha, (\overline{x}, \overline{y}), \Delta, (p_0, q_0), (Q_1 \times F_2) \cup (F_1 \times Q_2))$$

such that $\alpha((p,q)) = (\alpha_1(p), \alpha_2(q))$ and the transition function Δ is defined as follows: For all $p_1 \in Q_1, p_2 \in Q_2, \overline{b} = (b_1, \ldots, b_{k_1+k_2+\ell}) \in \{0,1\}^{k_1+k_2+\ell}$, suppose

$$\Delta_1(p_1, (b_1, \ldots, b_{k_1}, b_{k_1+k_2+1}, \ldots, b_{k_1+k_2+\ell})) = (q_1, (g_1, \ldots, g_{k_1}))$$

and

$$\Delta_2(p_2, (b_{k_1+1}, \ldots, b_{k_1+k_2+\ell})) = (q_2, (g_{k_1+1}, \ldots, g_{k_1+k_2}))$$

where $g_1, \ldots, g_{k_1+k_2} \in \mathsf{Op}(\mathcal{S})$. Then we let $\Delta((p_1,p_2), \overline{b}) = ((q_1,q_2), (g_1, \ldots, g_{k_1+k_2}))$.

Intuitively, on any input D-word w, the automaton $\mathcal{A}$ simulates the computation of $\mathcal{A}_1$ over w using the first k_1 changing registers and the ℓ fixed registers. At the same time, the automaton $\mathcal{A}$ simulates $\mathcal{A}_2$ using the remaining k_2 changing registers and the ℓ fixed registers. ∎

The next lemma can be easily proved since the $\mathcal{S}$-automata are deterministic.

Lemma 4.3.2 (The complementation lemma) *Given an $(\mathcal{S}, k)$-automaton $\mathcal{A}$, there is an $(\mathcal{S}, k)$-automaton $\mathcal{A}^c$ such that $L(\mathcal{A}^c) = D^* \setminus L(M)$ where D is the domain of $\mathcal{S}$.*

The next theorem easily follows from Lemma 4.3.1 and Lemma 4.3.2.

Theorem 4.3.3 (Closure under Boolean operations) *Let $\mathcal{S}$ be a structure. The class of languages recognized by $\mathcal{S}$-automata is closed under union, intersection and complementation.*

Now we present several examples of deterministic $\mathcal{S}$-automata where the structure $\mathcal{S}$ has the set of natural number $\mathbb{N}$ as its domain.

Example 4.3.1 *Let $\mathcal{S} = (\mathbb{N}; \mathrm{pr}_2, <)$. Let L be the language containing all* monotonic sequences, *i.e. $\mathbb{N}$-words of the form $a_1 \ldots a_n$ such that $a_i \leq a_{i+1}$ for all $i \in \{1, \ldots, n-1\}$. An $(\mathcal{S}, 1)$-automaton accepting this language is presented in Fig. 4.2.*

Example 4.3.2 *An* pre-arithmetic progression *is a sequence*

$$a, x, a+x, x, a+2x, x, a+3x, \ldots, a+nx, x$$

where $a, n, x \in \mathbb{N}$. An $((\mathbb{N}; +, =, \mathrm{pr}_1), 2)$-automaton accepting all pre-arithmetic progressions is presented in Fig. 4.3.

Figure 4.2: An $(\mathcal{S}, 1)$-automaton accepting the monotonic sequences. Note $\alpha(q_0) = \alpha(q_1) =<$. A transition (q, b, q', g) is represented by an arrow from state q to q' with label $b : g$. The arrow labeled by $0/1 : \mathrm{pr}_2$ represent both transitions $(q_1, 0, q_1, \mathrm{pr}_2)$ and $(q_1, 1, q_1, \mathrm{pr}_2)$. The initial value is 0.

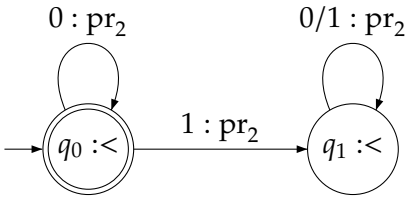
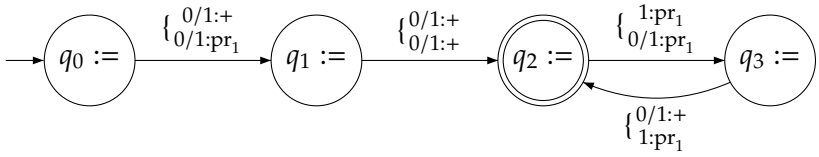
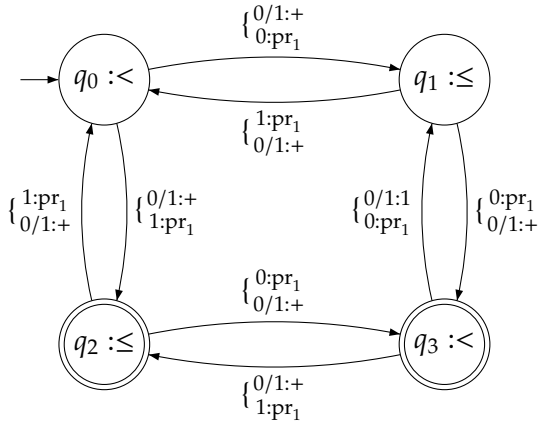


Figure 4.3: An $((\mathbb{N}; +, =, \mathrm{pr}_1), 2)$-automaton accepting all pre-arithmetic progressions. The initial value is (0,0).



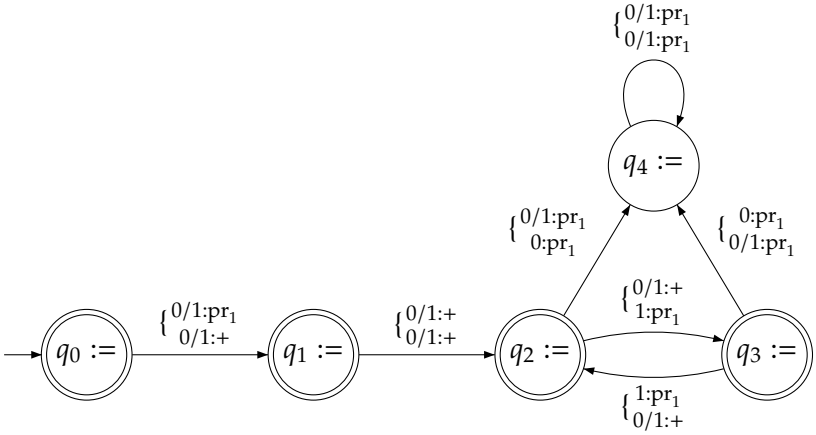
Example 4.3.3 *Let* $\mathcal{S} = (\mathbb{N}; +, \mathrm{pr}_1, <, \leq)$. *For any* $\mathbb{N}$*-words* w*, let* $\mathsf{even}(w)$ *be the sum of numbers on the even positions of* w*, and let* $\mathsf{odd}(w)$ *be the sum of numbers on the odd positions of* w*. Let* L *contain all* $\mathbb{N}$*-words* w *such that* $\mathsf{odd}(w) < \mathsf{even}(w)$*. An* $(\mathcal{S}, 2)$*-automaton accepting* L *is presented in Fig. 4.4.*

Figure 4.4: An $(\mathcal{S}, 2)$-automaton accepting the $\mathcal{S}$-language $L = \{w \mid \mathsf{odd}(w) < \mathsf{even}(w)\}$. The initial value is $(0, 0)$



Example 4.3.4 *Let* $\mathcal{S} = (\mathbb{N}; +, \mathrm{pr}_1, =)$. *Let* F *contain all* Fibonacci sequences, *i.e.* $\mathbb{N}$*-words* $a_1 a_2 \ldots a_n$ $(n \in \mathbb{N})$ *where* $a_{i+2} = a_{i+1} + a_i$ *for* $i \in \{1, \ldots, n-2\}$*. A deterministic* $(\mathcal{S}, 2)$*-automaton accepting* F *is presented in Fig 4.5.*

Figure 4.5: An $(\mathcal{S}, 2)$-automaton accepting the Fibonacci sequences. The initial value is $(0, 0)$.



The next example shows how deterministic $\mathcal{S}$-automata may be used to accept execution sequences of algorithms.

Example 4.3.5 *Let* $\mathcal{S} = (\mathbb{N}; +, \%, \mathrm{pr}_2, =, 0)$ *where* % *denotes the modulo operation on natural numbers, where* $a\%b = r$ *means* $r < b$ *and* $r + bq = a$ *for some* $q \in \mathbb{N}$. *Euclid's algorithm computes the greatest common divisor of two given natural numbers* $x, y \in \mathbb{N}$ *by repeatedly computing the sequence* $a_1, a_2, a_3, \ldots$ *such that* $a_1 = x, a_2 = y$, *and* $a_i = a_{i-2}\%a_{i-1}$ *for* $i > 2$. *The procedure terminates when* $a_i = 0$ *and declares that* a_{i-1} *is* $\gcd(x, y)$. *We call such a sequence* $a_1, a_2, a_3, \ldots$ *an* Euclidean path. *For example, the* $\mathbb{N}$*-word* 384 270 114 42 30 12 6 0 *is an Euclidean path. Note that if* $a_1 \ldots a_n$ *is an Euclidean path, then* $a_{n-1} = \gcd(a_1, a_2)$. *Hence an Euclidean path can be thought of as a computation of Euclid's algorithm. A deterministic* $(\mathcal{S}, 2)$*-automaton accepting the set of all Euclidean paths is presented in Fig. 4.6.*

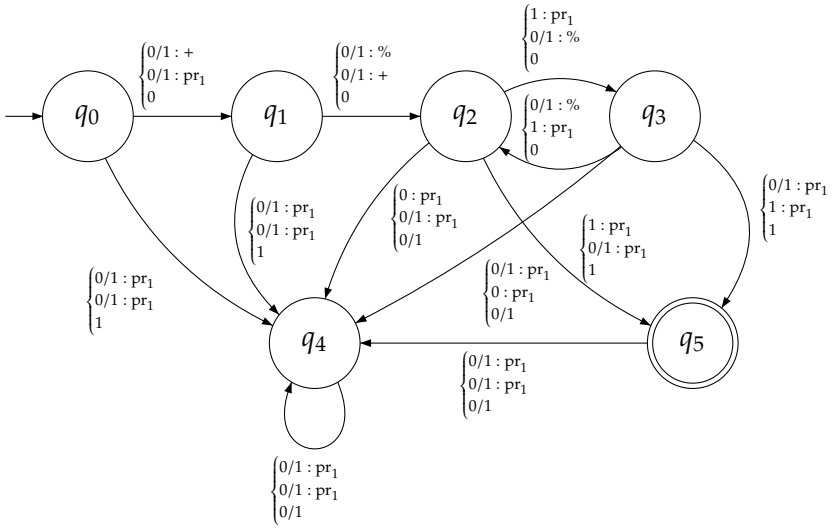
4.4 The Validation Problem

This section discusses the validation problem for automata over a given structure $\mathcal{S}$. The problem is formulated as follows.

Validation problem. Design an algorithm that, given an $\mathcal{S}$-automaton $\mathcal{A}$ and a path p in $\mathcal{A}$ from the initial state to an accepting state, decides if there exists a D-word $\overline{a}$ such that a run of $\mathcal{A}$ over $\overline{a}$ proceeds along p.

Obviously the problem depends on the given structure $\mathcal{S}$. For instance, if $\mathcal{S}$ is a finite structure then, by Example 4.2.1(b), both the validation and the emptiness problem are

Figure 4.6: An $((\mathbb{N}; +, \%, \mathrm{pr}_1, =, 0), 3)$-automaton accepting the Euclidean paths. The initial value is $(0, 0, 0)$. The mapping α maps every state q to the tuple $(=, =, =)$.



decidable. The validation problem for $\mathcal{S}$-automata turns out to be equivalent to solving systems of equations and in-equations over the structure. More formally, we define the following:

Definition 4.4.1 *The* existential theory *of $\mathcal{S}$, denoted by $\mathsf{Th}_\exists(\mathcal{S})$ is the set of all existential sentences true in $\mathcal{S}$, that is,*

$$\mathsf{Th}_\exists(\mathcal{S}) = \{\varphi \mid \mathcal{S} \models \varphi \text{ and } \varphi \text{ is an existential sentence}\}.$$

Lemma 4.4.1 *If $\mathsf{Th}_\exists(\mathcal{S})$ is decidable, then the validation problem for $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automata is decidable, where $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$ is the expansion of $\mathcal{S}$ by the operations $\mathrm{pr}_1, \mathrm{pr}_2$ and the equality predicate $=$.*

Proof Suppose the existential theory of $\mathcal{S}$ is decidable. Then the existential theory of $\mathcal{S}$ expanded by the operations $\mathrm{pr}_1, \mathrm{pr}_2$ $\mathsf{Th}_{\exists(\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2])}$ is also decidable. Let $\mathcal{A} = (Q, \alpha, \overline{x}, \Delta, q_0, F)$ be an $(\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =], k)$-automata. Let $p = q_0, \ldots, q_m$ be a path in the transition diagram of $\mathcal{A}$. Our goal is to construct an existential sentence φ_p (in the signature of $\mathcal{S}$ expanded by pr_1 and pr_2) such that $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2] \models \varphi_p$ if and only if there exists a D-word $\overline{a}$ such that the run of $\mathcal{A}$ over $\overline{a}$ proceeds along p, and therefore prove that the validation problem for $\mathcal{A}$ is decidable. We need to introduce some terminology and notation.

A sequence of transitions $t_1, \ldots, t_m \in \Delta$ *conforms* with the path p if for each $i \in \{1, \ldots, m\}$,

$t_i = (q_{i-1}, \overline{b}, q_i, \overline{g})$ for some $\overline{b} \in \{0,1\}^{k+\ell}$ and $\overline{g} \in \mathsf{Op}(\mathcal{S})$. Since for each edge e in the transition diagram there are at most $2^{k+\ell}$ number of transitions that conform with e, there are only finitely many sequences of transitions that conform with the path p. For any state $q \in Q$ we denote $\alpha(q)$ as $(P_q^1, \ldots, P_q^{k+\ell})$. For any transition $t = (q, (b_1, \ldots, b_{k+\ell}), q', (g_1, \ldots, g_k)) \in \Delta$ and $j \in \{1, \ldots, k+\ell\}$, we let S_t^j be the relation P_q^j if $b_j = 1$ and let S_t^j be $\neg P_q^j$ otherwise. Furthermore, for any $j \in \{1, \ldots, k\}$, let f_j^t denote the operation g_j.

For a sequence $\overline{t} \in \Delta^*$ conforming with p, let $\psi_{\overline{t}}$ be the following existential sentence:

$$\exists y_1, \ldots, y_m : \bigwedge_{i=1}^{m} \bigwedge_{j=1}^{k+\ell} S_{i,j}(y_i, \tau(x_j, y_1, \ldots, y_{i-1}, \overline{c})),$$

where $\alpha(q_{i-1}) = (S_{i,1}, \ldots, S_{i,k})$ and for $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, k+\ell\}$ and $\overline{y} = y_1, \ldots, y_{i-1}$ the term

$$\tau(x_j, \overline{y}, \overline{c}) = \begin{cases} f_j^{t_{i-1}}(f_j^{t_{i-2}}(\ldots f_j^{t_1}(x_j, y_1), y_2)\ldots), y_{i-2}), y_{i-1}) & \text{if } 1 \leq j \leq k \\ c_j & \text{if } j > k \end{cases}$$

By definition it is clear that the sentence $\psi_{\overline{t}}$ holds in the structure $\mathcal{S}$ if and only if $\mathcal{S}$ contains elements $y_1, \ldots, y_m$ such that on the D-word $y_1 \ldots y_m$ the automaton $\mathcal{A}$ will proceed along the transition $t_1, \ldots, t_m$. Note that $\psi_{\overline{t}}$ does not hold if any of the operations f_j^t (where $j \in \{1, \ldots, k\}$ and $t \in \{t_1, \ldots, t_m\}$) are undefined for the arguments specified by $\tau(x_j, \overline{y}, \overline{c})$. Hence there exists a D-word $\overline{a}$ such that the run of $\mathcal{A}$ over $\overline{a}$ proceeds along p if and only if the following sentence holds:

$$\bigvee_{\overline{t} \text{ conforms with } p} \psi_{\overline{t}}.$$

Since there are only finitely many $\overline{t}$ that conform with p, the above sentence is clearly existential. ∎

To prove the other direction, suppose that the validation problem for $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automata is decidable. Let φ is an existential formula over the signature of $\mathcal{S}$. Without loss of generality, we assume φ is of the form $\exists x_1, \ldots, x_k : \bigvee_{i=1}^{m} \psi_i(x_1, \ldots, x_k)$ where $\psi(x_1, \ldots, x_k)$ is a conjunction of literals. Hence φ is equivalent to the sentence

$$\bigvee_{i=1}^{s} \exists x_1, \ldots, x_k : \psi_i(x_1, \ldots, x_k).$$

Our goal is to construct an $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automaton $\mathcal{A}_i$ for each $i \in \{1, \ldots, s\}$ such that

($\star$) $\mathcal{A}_i$ contains exactly one accepting state and there is exactly one path in the transition diagram that goes from the initial state to the accepting state.

($\star\star$) $\mathcal{S} \models \exists x_1, \ldots, x_k : \psi_i(x_1, \ldots, x_k)$ if and only if there exists a D-word that $\mathcal{A}_i$ accepts.

Then applying Lemma 4.3.1 we build an $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automaton $\mathcal{A}$ accepting the $\mathcal{S}$-language $\bigcup_{i=1}^{s} L(\mathcal{A}_i)$. Furthermore, the construction from Lemma 4.3.1 guarantees that there is an unique path in the transition diagram that goes from the initial state to each accepting state in $\mathcal{A}$. Hence checking if $\varphi \in \mathsf{Th}_\exists(\mathcal{S})$ is reduced to deciding the validation problem for $\mathcal{A}$ on at most t paths and is therefore decidable. For the following lemma, we define an *x-assignment* as a formula of the form $x = f(y, z)$ where x, y, z are variables in $\mathcal{S}$.

Lemma 4.4.2 *Let $\{x_1, \ldots, x_k\}$ be a set of variables in $\mathcal{S}$ and let Φ be a set of x_i-assignments ($i \in \{1, \ldots, k\}$) where for each i, Φ contains at most one x_i-assignment. We can effectively construct an $(\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =], k)$-automaton $\mathcal{A}[\Phi]$ such that the following hold.*

- *$\mathcal{A}[\Phi]$ contains exactly one accepting state and there is exactly one path in the transition diagram that goes from the initial state to the accepting state.*
- *Suppose $\mathcal{A}[\Phi]$ processes the input from the initial state and reaches the accepting state. Let $x_1, \ldots, x_k$ respectively take the current values of the k registers of $\mathcal{A}[\Phi]$. Then $\mathcal{S}$ satisfies the conjunction of all formula in Φ.*

Proof Take $x \in \{x_1, \ldots, x_k\}$ in ψ. We say that x has rank 0, denoted by $\mathsf{rank}(x) = 0$, if Φ does not contain an x-assignment. Otherwise, say ψ contains an x-assignment X_j of the form $x = f(y, z)$. We let $\mathsf{rank}(x) = \max\{\mathsf{rank}(y), \mathsf{rank}(z)\} + 1$. In this case we also say $\mathsf{rank}(X_j) = \mathsf{rank}(x)$. We use r to denote the maximum rank of any variable in Var.

We now construct the $(\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =], k)$-automaton $\mathcal{A}[\Phi]$. The initial values of the k registers can be chosen arbitrarily. The automaton $\mathcal{A}[\Phi]$ contains $(2r+1)k+2$ states, which are denoted by

$$q_{0,0}, \ldots, q_{0,k}, q_{1,1}, \ldots, q_{1,2k}, \ldots, q_{r,1}, \ldots, q_{r,2k}, q_{\mathrm{rej}}.$$

The initial state is $q_{0,0}$ and the unique accepting state is $q_{r,2k}$. The mapping α maps every state in $\mathcal{A}[\Phi]$ to the tuple $\{=\}^{k+\ell}$. The state q_{rej} is a *sink*, i.e., all outgoing transitions of q_{rej} go back to q_{rej}. Intuitively, the automaton will work in stages: At stage 0, the automaton will

assign values to all rank-0 variables using the states $q_{0,0}, \ldots, q_{0,k}$. Then for $i \in \{1, \ldots, r\}$, the automaton will assign values to all rank-i variables using the states $q_{i,1}, \ldots, q_{i,2k}$. Formally, we define below the transitions of the automaton $\mathcal{A}$.

- For $i \in \{1, \ldots, k\}$, add the transitions

$$(q_{0,i-1}, \overline{b}, q_{0,i}, (g_1, \ldots, g_k))$$

where $\overline{b} \in \{0,1\}^{k+\ell}$ and for all $j \in \{1, \ldots, k\}$

$$g_j = \begin{cases} \mathrm{pr}_2 & \text{if } j = i \\ \mathrm{pr}_1 & \text{otherwise} \end{cases}$$

Suppose the automaton starts processing the input from the initial state $q_{0,0}$ and arrives at state $q_{0,i}$ for $i \in \{1, \ldots, k\}$, then it must have read i input elements $y_1, \ldots, y_i$, and for $j \in \{1, \ldots, j\}$, the jth register will store the value y_j.

- For $i \in \{1, \ldots, r\}$, add the transitions

$$(q_{i-1,k}, \overline{b}, q_{i,1}, (g_1, \ldots, g_k))$$

such that the following hold.

  - Suppose $\mathsf{rank}(x_1) < i$. Then $\overline{b} \in \{1\} \times \{0,1\}^{k+\ell-1}$ and $g_j = \mathrm{pr}_2$ for all j where $\mathsf{rank}(x_j) = i$ and the x_j-assignment is $x_j = f(x_1, y)$ (for some y and f). All other g_j is pr_1.
  - Suppose $\mathsf{rank}(x_0) \geq i$. Then $\overline{b} \in \{0,1\}^{k+\ell}$ and $g_j = \mathrm{pr}_1$ for all j.

- For $i \in \{1, \ldots, r\}$ and $j \in \{2, \ldots, k\}$, add the transitions

$$(q_{i,j-1}, \overline{b}, q_{i,j}, (g_1, \ldots, g_k))$$

such that the following hold.

  - Suppose $\mathsf{rank}(x_j) < i$. Then $\overline{b} \in \{0,1\}^{j-1}1\{0,1\}^{k+\ell-j}$ and $g_m = \mathrm{pr}_2$ for all m where $\mathsf{rank}(x_m) = i$ and the x_m-assignment is $x_m = f(x_j, y)$ (for some y and f). All other g_m is pr_1.

  - Suppose $\mathsf{rank}(x_j) \geq i$. Then $\overline{b} \in \{0,1\}^{k+\ell}$ and $g_j = \mathrm{pr}_1$ for all j.

- For $i \in \{1, \ldots, r\}$ and $j \in \{k+1, \ldots, 2k\}$, add the transitions

$$(q_{i,j-1}, \overline{b}, q_{i,j}, (g_1, \ldots, g_k))$$

  such that the following hold.

  - Suppose $\mathsf{rank}(x_j) < i$. Then $\overline{b} \in \{0,1\}^{j-1}1\{0,1\}^{k+\ell-j}$ and $g_m = \mathrm{pr}_2$ for all m where $\mathsf{rank}(x_m) = i$ and the x_m-assignment is $x_m = f(y, x_j)$ (for some y and f). All other g_m is pr_1.
  - Suppose $\mathsf{rank}(x_j) \geq i$. Then $\overline{b} \in \{0,1\}^{k+\ell}$ and $g_j = \mathrm{pr}_1$ for all j.

- All other transitions in $\mathcal{A}[\Phi]$ directs to the sink q_{rej}.

Suppose the automaton starts processing the input from the initial state $q_{0,0}$ and arrives at state $q_{i,2k}$ for $i \in \{1, \ldots, r\}$, then it must have read $(2i+1)k$ input elements. Furthermore, if we let x_j take current value of the jth register (for all $j \in \{1, \ldots, k\}$), then $\mathcal{S}$ satisfies the conjunction of all x_m-assignments in Φ where $\mathsf{rank}(x_m) \leq i$. Hence the lemma is proved ∎

Lemma 4.4.3 *Let* $\mathsf{Var} = \{x_1, \ldots, x_k\}$ *be a set of variables and* Const *be the set of constants in* $\mathcal{S}$*. Let* Ψ *be a finite set of formulae of the form* $R(x, y)$ *or* $\neg R(x, y)$ *where* $R \in \mathsf{Rel}(\mathcal{S}) \cup \{=\}$ *and* $x, y \in \mathsf{Var} \cup \mathsf{Const}$ *and* $x \neq y$*. We can effectively construct an* $(\mathcal{S}[\mathrm{pr}_1, =], k)$*-automaton* $\mathcal{B}[\Psi]$ *such that*

- $\mathcal{B}[\Psi]$ *contains exactly one accepting state and there is exactly one path in the transition diagram that goes from the initial state to the accepting state.*

- *Let* $x_1, \ldots, x_k$ *take the initial values of the* k *registers in* $\mathcal{B}[\Psi]$*. There is an input word that proceeds along this path if and only if* $\mathcal{S}$ *satisfies all formulae in* Ψ*.*

Proof Let $\Psi = \{\varphi_1, \ldots, \varphi_m\}$. The automaton $\mathcal{B}[\Psi]$ has $m+2$ states, which are denoted by

$$q_0, q_1, \ldots, q_m, q_{\mathsf{rej}}.$$

The initial state is q_0 and the accepting state is q_m. Similar to the automaton $\mathcal{A}[\Phi]$, the state q_{rej} is a sink. Let $x_1, \ldots, x_k$ be the initial values of the k registers in $\mathcal{B}[\Psi]$. Intuitively,

the automaton $\mathcal{B}[\Psi]$ checks if the formula φ_i holds on the state q_{i-1}. If φ_i holds, then the automaton makes a transition to the next state q_i. Otherwise, the automaton moves to q_{rej}.

Formally, we use $x_{k+1}, \ldots, x_{k+\ell}$ to denote the constants $c_1, \ldots, c_\ell$. For each $i \in \{1, \ldots, m\}$, if $\varphi_i = S(x_i, x_j)$ or $\neg S(x_i, x_j)$ where $i, j \in \{1, \ldots, k+\ell\}$, the mapping α will map the state q_{i-1} to a tuple $\{=\}^{i-1} S \{=\}^{k+\ell-i} \in \mathsf{Rel}^{k+\ell}(\mathcal{S}) \cup \{=\}$ There are transitions

$$(q_{i-1}, (b_1, \ldots, b_{k+\ell}), q_i, (\underbrace{\mathrm{pr}_1, \ldots, \mathrm{pr}_1}_{k}))$$

where $b_j = 1$, $b_i = 1$ if $\varphi_i = S(x_i, x_j)$ and $b_i = 0$ if $\varphi_i = \neg S(x_i, x_j)$, and $b_m \in \{0, 1\}$ for all $m \notin \{i, j\}$. All other transitions in $\mathcal{B}[\Psi]$ directs to the sink q_{rej}. It is clear that all of $\varphi_1, \ldots, \varphi_m$ hold if and only if some input sequence proceeds along the path $q_0, \ldots, q_m$. ∎

Using the lemmas above we are then ready to prove the following lemma.

Lemma 4.4.4 *If the validation problem for $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automata is decidable, then $\mathsf{Th}_\exists(\mathcal{S})$ is decidable.*

Proof By the discussion above, it suffice to construct the $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automaton $\mathcal{A}_i$ that satisfy properties $(\star)$ and $(\star\star)$ for every $i \in \{1, \ldots, s\}$. For ease of notation we will drop the subscript i and simply say we construct an $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automaton $\mathcal{A}$ for the existential sentence $\exists x_1, \ldots, x_k : \psi(x_1, \ldots, x_k)$ where $\psi(x_1, \ldots, x_k)$ is a conjunction of literals. Let Var denote the set $\{x_1, \ldots, x_k\}$ of all free variables that appeared in ψ and Const denote the set of all constants in ψ. Without loss of generality, we assume that the conjunction $\psi(x_1, \ldots, x_k)$ can be written in the form

$$\bigwedge_{i=1}^{s_1} X_i \wedge \bigwedge_{i+1}^{s_2} Y_i$$

such that the following hold:

1. Each X_i is an x_i-assignment for some $i \in \{1, \ldots, k\}$.
2. If X_i is an y-assignment and X_j is an z-assignment and $i \neq j$, then $y \neq z$.
3. Each Y_i is of the form $R(x, y)$ or $\neg R(x, y)$ where $x, y \in \mathsf{Var} \cup \mathsf{Const}$ and $R \in \mathsf{Rel}(\mathcal{S}) \cup \{=\}$.

We construct the $(\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =], k)$-automaton $\mathcal{A}[\{X_1, \ldots, X_{s_1}\}]$ and $\mathcal{B}[\{Y_1, \ldots, Y_{s_2}\}]$ as described in Lemma 4.4.2 and Lemma 4.4.3. The desired automaton $\mathcal{A}$ can be obtained by

joining these two automata such that the accepting state in $\mathcal{A}[\{X_1, \ldots, X_{s_1}\}]$ and the initial state $\mathcal{B}[\{Y_1, \ldots, Y_{s_2}\}]$ is replaced by a single state, the initial state of $\mathcal{A}$ is the initial state of $\mathcal{A}[\{X_1, \ldots, X_{s_1}\}]$ and the only accepting state of $\mathcal{A}$ is the accepting state of $\mathcal{B}[\{Y_1, \ldots, Y_{s_2}\}]$. The correctness of the construction is directly implied from the statements of Lemma 4.4.2 and Lemma 4.4.3. ∎

Combining Lemma 4.4.1 and Lemma 4.4.4, we obtain the following theorem.

Theorem 4.4.5 *The validation problem for $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automata is decidable if and only if $\mathsf{Th}_\exists(\mathcal{S})$ is decidable.*

4.5 The Emptiness Problem

This section discusses the emptiness problem for $\mathcal{S}$-automata.

Emptiness problem. Design an algorithm that, given a structure $\mathcal{S}$ and an $(\mathcal{S}, k)$-automaton $\mathcal{A}$, decides if $\mathcal{A}$ accepts at least one D-word.

4.5.1 The emptiness problem for acyclic $\mathcal{S}$-automata

A *sink state* in an $\mathcal{S}$-automaton is a state whose all outgoing transitions loop into the state itself. All accepting sink states can be collapsed into one accepting sink state, and all non-accepting sink states can be collapsed into one non-accepting sink state. Therefore we can always assume that every $\mathcal{S}$-automaton has at most 2 sink states.

Definition 4.5.1 *We call an $\mathcal{S}$-automaton* acyclic *if its state space without the sink states is an acyclic graph.*

Note that in any acyclic $\mathcal{S}$-automaton, there are only finitely many paths from the initial state to an accepting state. Hence the emptiness problem is computationally equivalent to the validation problem. Theorem 4.4.5 implies the following corollary that we state as a theorem.

Theorem 4.5.1 *For any structure $\mathcal{S}$, the emptiness problem of acyclic $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automata is decidable if and only if $\mathcal{S}$ has decidable existential theory.*

The above theorem immediately provides a wide range of structures $\mathcal{S}$ for which the emptiness problem of acyclic $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automata is decidable. The following corollary lists a few examples of such structures. The structures (a-c) are well-known to have decidable first-order theory, (d) has decidable theory by [14], (e) has decidable theory by [77], and (f-g) have decidable theory since they are instances of automatic structures [49].

Corollary 4.5.2 *The emptiness problem is decidable for acyclic $\mathcal{S}[\mathrm{pr}_1, \mathrm{pr}_2, =]$-automata where $\mathcal{S}$ is the following structures and $c_1, \ldots, c_k$ are constants in the respective domain:*

(a) $(\mathbb{N}; +, <, \leq, c_1, \ldots, c_\ell)$.

(b) $(\mathbb{N}; \times, c_1, \ldots, c_\ell)$.

(c) Any finitely generated Abelian group.

(d) $(\mathbb{N}; +, V_p)$ *where* $p \in \mathbb{N}$ *and the function* $V_p : \mathbb{N}^2 \to \mathbb{N}$ *is defined as*

$$V_p(x, y) = \begin{cases} \text{the greatest power of } p \text{ dividing } x & \text{if } x \neq 0 \text{ and} \\ 1 & \text{if } x = 0. \end{cases}$$

(e) $(\mathbb{N}; +, \mathsf{pow}_2, c_1, \ldots, c_\ell)$ *where the function* $\mathsf{pow}_2 : \mathbb{N}^2 \to \mathbb{N}$ *is the function* $(x, y) \to 2^x$.

(f) $(\mathbb{Q}; +, \leq, c_1, \ldots, c_\ell)$ *where* $\mathbb{Q}$ *is the set of rational numbers.*

(g) The Boolean algebra of finite and co-finite subsets of $\mathbb{N}$.

Theorem 4.5.1 above poses the following question. Let $\mathcal{S}$ be a structure with undecidable existential theory, find k such that the emptiness problem for acyclic $(\mathcal{S}, k)$-automata is undecidable. Speculatively there might be a structure $\mathcal{S}$ with undecidable existential theory (and hence undecidable emptiness problem for acyclic $\mathcal{S}$-automata) such that for each k the emptiness problem for acyclic $(\mathcal{S}, k)$-automata is decidable, but we don't know any such example. Below we provide an example of a structure $\mathcal{S}$ such that the emptiness problem for acyclic $(\mathcal{S}, 1)$-automata is undecidable.

Let $G = (V, E)$ be a computable graph for which testing whether each node is isolated is undecidable (the reader is referred to [33] for the existence of such a graph). Then the following acyclic $(G[\mathrm{pr}_2, =], 1)$-automaton $\mathcal{A}$ has undecidable emptiness problem. The

$(G,1)$-automaton $\mathcal{A}$ has four states $q_0, q_1, q_\mathsf{f}, q_\mathsf{s}$ where $q_\mathsf{f}, q_\mathsf{s}$ are sink states and $F = \{q_\mathsf{f}\}$. The mapping α maps q_0 to $=$ and q_1 to E. The transitions on q_0 and q_1 are

$$\{(q_0, b, q_1, \mathrm{pr}_2) \mid b = 0, 1\} \cup \{(q_1, 0, q_\mathsf{s}, \mathrm{pr}_2)\} \cup \{(q_1, 1, q_\mathsf{f}, \mathrm{pr}_2)\}$$

We now give a more natural example of a structure $\mathcal{S}$ where emptiness problem is undecidable for acyclic $(\mathcal{S}, k)$-automata with small k. Consider the following structure $\mathcal{S} = (\mathbb{Z}; +, \times, \mathrm{pr}_1, \mathrm{pr}_2, =, 0)$. Let $p(x_1, \ldots, x_k)$ be a polynomial in $\mathbb{N}[x_1, \ldots, x_k]$ of the form

$$\sum_{i=1}^{s} c_i \prod_{j=1}^{t_i} x_{i,j}$$

where each $x_{i,j} \in \{x_1, \ldots, x_k\}$. Let $C = \{c_1, \ldots, c_s\}$.

Lemma 4.5.3 *There exists a deterministic acyclic* $(\mathcal{S}[C], k{+}2)$*-automaton* $\mathcal{A}_p$ *such that if* $(a_1, \ldots, a_k, 0, 0) \in \mathbb{N}$ *is the initial value of* $\mathcal{A}_p$ *then there is a unique* $\mathbb{Z}$*-word that* $\mathcal{A}_p$ *accepts and when the accepting run arrives at the accepting state, the last register will have the value* $p(a_1, \ldots, a_k)$*.*

Proof Note that the automaton $\mathcal{A}_p$ has s fixed registers containing the values $c_1, \ldots, c_s$. Let $m_1, \ldots, m_{k+2}$ be the $k+2$ changing registers of $\mathcal{A}_p$. Intuitively, during any accepting run, the first k changing registers do not change their values, the $(k+1)$th register m_{k+1} is responsible for storing the value of each term $c_i \prod_{j=1}^{t} x_{i,j}$, and the last register m_{k+1} stores the partial sums

$$c_1 \prod_{j=1}^{t_1} x_{1,j} + \ldots + c_i \prod_{j=1}^{t_i} x_{i,j}$$

for $i \in \{1, \ldots, s\}$. Formally, $\mathcal{A}_p$ has states

$$\{q_\mathsf{rej}\} \cup \{q_{i,j} \mid 1 \le i \le s, 1 \le j \le t_i + 1\} \cup \{p_i \mid 1 \le i \le s + 1\}.$$

The initial state is p_1 and q_rej is a sink. The transition is defined as follows:

- From state p_i $(1 \le i \le s)$, $\mathcal{A}_p$ makes a transition to $q_{i,1}$ only when the input equals to c_i, and $\mathcal{A}_p$ then applies pr_2 on the register m_{k+1} (thus $m_{k+1} = c_i$ when $\mathcal{A}_p$ reaches $q_{i,1}$).
- From state $q_{i,j}$ $(1 \le j \le t_i)$, $\mathcal{A}_p$ makes a transition to $q_{i,j+1}$ only when the input equals to $x_{i,j}$, and $\mathcal{A}_p$ then applies $\times$ on the register m_{k+1} (thus $m_{k+1} = c_i \times x_{i,1} \times \ldots \times x_{i,j}$) when $\mathcal{A}_p$ reaches $q_{i,j+1}$.

- From state q_{i,t_i+1} $(1 \leq i \leq s)$, $\mathcal{A}_p$ makes a transition to p_{i+1} only when the input equals to m_{k+1}, and $\mathcal{A}_p$ then applies + on the register m_{k+2}. Thus $m_{k+2} = c_1 \prod_{j=1}^{t_1} x_{1,j} + \ldots + c_i \prod_{j=1}^{t_i} x_{i,j}$.
- All other transitions go to q_{rej}.

Hence the lemma is proved. ∎

Proposition 4.5.4 *Let* $\mathcal{S}_{\mathbb{Z}} = (\mathbb{Z}; +, \times, \mathrm{pr}_1, \mathrm{pr}_2, =, 0)$ *and* $\mathcal{S}_{\mathbb{N}} = (\mathbb{N}; +, \times, \mathrm{pr}_1, \mathrm{pr}_2, =, 0)$.

(a) The emptiness problem for deterministic acyclic $(\mathcal{S}_{\mathbb{Z}}, 11)$*-automata is undecidable.*

(b) The emptiness problem for deterministic acyclic $(\mathcal{S}_{\mathbb{N}}, 12)$*-automata is undecidable.*

Proof For (a), we prove the proposition using a reduction from the Hilbert's tenth problem: Given a polynomial $p(x_1, x_2, \ldots, x_k) \in \mathbb{N}[x_1, x_2, \ldots, x_k]$, decide if the following hold

$$\exists x_1, \ldots, x_k \in \mathbb{Z} : p(x_1, \ldots, x_k) = 0.$$

By [63], the problem is not decidable and the number of variables in the polynomial can be bounded by 9.

By Lemma 4.5.3, it is easy to build, for any polynomial $p \in \mathbb{Z}[x_1, \ldots, x_9]$, an $(\mathcal{S}_{\mathbb{Z}}, 11)$-automaton $\mathcal{A}_p$ such that computes the value of $p(a_1, \ldots, a_k)$ for initial values $(a_1, \ldots, a_k)$. One can slightly modify $\mathcal{A}_p$ such that the modified $\mathcal{S}$-automaton accepts a $\mathbb{Z}$-word if and only if $p(x_1, \ldots, x_9) = 0$ has a integer solution.

For (b), we prove use reduction from a slightly different problem: Given two polynomials $p(x_1, x_2, \ldots, x_k), q(x_1, x_2, \ldots, x_k) \in \mathbb{N}[x_1, x_2, \ldots, x_k]$, decide if the following hold

$$\exists x_1, \ldots, x_k \in \mathbb{N} : p(x_1, \ldots, x_k) = q(x_1, \ldots, x_k).$$

Again the problem is already undecidable when the number k of variables is 9.

By slightly modifying the construction in Lemma 4.5.3, it is easy to build, for any two polynomials $p \in \mathbb{N}[x_1, \ldots, x_9]$, $q \in \mathbb{N}[x_1, \ldots, x_9]$, an $(\mathcal{S}_{\mathbb{N}}, 12)$-automata $\mathcal{B}_{p,q}$ such that $L(\mathcal{B}_{p,q}) \neq \emptyset$ if and only if there exist $a_1, \ldots, a_9$ such that $p(x_1, \ldots, x_9) = q(x_1, \ldots, x_0)$. Here we need one more changing register because we need two registers to store the values of the two polynomials separately. ∎

It is a natural open question whether the emptiness problem for acyclic $(\mathbb{Z}, k)$- and $(\mathbb{N}, m)$-automata is decidable where $k < 11$ and $m < 12$. Another natural question is the decidability of the emptiness problem if we remove the acyclicity constraint. The next section discusses this problem.

4.5.2 The emptiness problem for automata on natural numbers

This section investigates the emptiness problem for $\mathcal{S}$-automata when the domain of $\mathcal{S}$ is the natural numbers $\mathbb{N}$, with the atomic operations being arithmetic operations such as addition, subtraction, multiplication and atomic relations being equality and natural ordering. Our goal is to show that there is a tradeoff between the decidability of the emptiness problem and the expressibility of the automata. We define the binary operation $+1$ and -1 on $\mathbb{N}^2$ such that $+1(x, y) = x + 1$ and $-1(x, y) = x - 1$ (Note that the -1 operation is not total). The next theorem shows that if we remove the acyclicity constraint, the emptiness problem is undecidable for $\mathcal{S}$-automata with a small number of registers.

Theorem 4.5.5 *Let* $\mathcal{S}_1 = (\mathbb{N}; +1, -1, =, \mathrm{pr}_1, 0)$ *and* $\mathcal{S}_2 = (\mathbb{N}, +1, =, \mathrm{pr}_1, \mathrm{pr}_2, 0)$.

(a) The emptiness problem for deterministic $(\mathcal{S}_1, 2)$*-automata is undecidable.*

(b) The emptiness problem for deterministic $(\mathcal{S}_2, 4)$*-automata is undecidable.*

Proof The proof of (a) uses a reduction from the emptiness problem of k-counter machines. Intuitively, a one-way k-counter machine is a finite state machine with k registers which can hold natural numbers and the only operations allowed on the registers are $-1, +0$ and $+1$. The machine reads inputs from a finite alphabet Σ and its head is only allowed to move to the right. Upon reading an input, the machine may test whether each register has a value of 0 or not and then move to another state and perform increment/decrement operations on the registers. A formal definition can be found in [48]. The language accepted by a counter machine $\mathcal{M}$ is the set of all words over Σ which take the machine from q_0 to an accepting state. Minsky [65] showed that the emptiness problem for deterministic one-way 2-counter machines is undecidable. One may reduce the emptiness problem for deterministic 2-counter machines to the emptiness problem of $(\mathcal{S}_1, 2)$-automata: The 2 changing registers store the two counters' values respectively. We simulate incrementing, $+0$ and decrementing of the counter by applying the operations $+1, \mathrm{pr}_1$ and -1, respectively.

Also we simulate the 0-test of counters by applying the =-comparison between the changing registers and the constant register storing 0.

For (b), observe that we may simulate the -1 operation by using the pr_2 and $+1$ operations along with two extra changing registers (say x_1, x_2). In order to perform a -1 operation on the i^{th} register, the automaton first reads an natural number and uses pr_2 to store this number in x_1 and x_2. Then it performs a $+1$ operation on x_2 and checks if the value of x_2 is equal to the value of the i^{th} register. If so, then the automaton uses the pr_2 operation to store the value of x_1 into the i^{th} register. Hence it is easy to see that the value of the i^{th} register is decremented exactly by 1.

Therefore for any deterministic $(\mathcal{S}_1, 2)$-automaton $\mathcal{M}$, we may effectively construct a deterministic $(\mathcal{S}_2, 4)$-automaton $\mathcal{M}'$ such that $L(\mathcal{M}) = \emptyset$ if and only if $L(\mathcal{M}') = \emptyset$. It then follows from (a) that the emptiness problem for deterministic $(\mathcal{S}_2, 4)$-automata is undecidable. ∎

The next question is whether the emptiness problem is undecidable if we lower the number of register even further. Below we will show that for the structure $\mathcal{S} = (\mathbb{N}, +, \times, \mathrm{pr}_1, \mathrm{pr}_2, \leq, =, c_1, \ldots, c_\ell)$ where $c_1, \ldots, c_\ell$ are arbitrary constants in $\mathbb{N}$, the emptiness problem is decidable for $\mathcal{S}$-automata with 1 changing register.

Theorem 4.5.6 *Let $\mathcal{S}$ be the structure $(\mathbb{N}; +, \times, \mathrm{pr}_1, \mathrm{pr}_2, =, \leq, c_1, \ldots, c_\ell)$ where $c_1, \ldots, c_\ell$ are arbitrary constants in $\mathbb{N}$. The emptiness problem for $(\mathcal{S}, 1)$-automata is decidable.*

Recall that a configuration of an $(\mathcal{S}, 1)$-automaton $\mathcal{A}$ is a pair (q, m) where q is a state of $\mathcal{A}$ and $m \in \mathbb{N}$ is the value of the changing register. The configuration graph of $\mathcal{A}$ contains all configurations as nodes, and two configurations are connected by an edge if a transition sends the first configuration to the second configuration. It is easy to see that the emptiness problem for $\mathcal{A}$ can be reduced to the reachability problem of the configuration graph. The $\mathcal{S}$-automaton $\mathcal{A}$ accepts an $\mathbb{N}$-word w if and only if there is a path that goes from the initial configuration to an accepting configuration in the configuration graph. A problem arises as the configuration graph of $\mathcal{A}$ is infinite. In the following we provide a way to "collapse" the configuration graph into a finite graph so that emptiness problem on $\mathcal{A}$ can be reduced to the reachability on the resulting finite graph. Without loss of generality, we assume $c_1 < c_2 < \ldots < c_\ell$. We make the following definition.

Definition 4.5.2 *We say that two numbers m_1, m_2 are* $(\mathcal{S}, 1)$-equivalent *if either $m_1 = m_2$ or*

$\min\{m_1, m_2\} > c_\ell$. *In this case we write* $m_1 \approx m_2$. *We use* $[m_1]$ *to denote the* $\approx$*-equivalence class of* m_1 *and* $C_\approx$ *denotes the set of equivalence classes of* $\approx$.

Note that $C_\approx$ is a finite set. We say that two configurations (p, m_1),(q, m_2) of some $(\mathcal{S}, 1)$-automaton are *equivalent* if $p = q$ and $m_1 \approx m_2$. We fix the following notation.

Definition 4.5.3 *An* $\mathcal{S}$-test *is a quantifier-free formula* $\varphi(x, y)$ *over* $\mathcal{S}$ *that is a finite conjunction of literals of the form*

$$x \leq c,\ c < x,\ x = c,\ x \neq c,\ x \leq y,\ y < x,\ x = y,\ \text{or } x \neq y$$

where c *is a constant in* $\mathcal{S}$.

The next lemma reveals the connection between the equivalence relation $\approx$ and the $\mathcal{S}$-tests.

Lemma 4.5.7 *Let* $\varphi(x, y)$ *be an* $\mathcal{S}$*-test condition and let* $m_1, m_2 \in \mathbb{N}$ *be such that* $m_1 \approx m_2$. *For any equivalence class* $C \in C_\approx$, *there is some* $z \in C$ *such that* $\mathcal{S} \models \varphi(z, m_1)$ *if and only if there is some* $z \in C$ *such that* $\mathcal{S} \models \varphi(z, m_2)$. *Furthermore, the set of equivalence classes* C *such that* $\exists z \in C : \mathcal{S} \models \varphi(z, m_1)$ *can be effectively computed from* φ *and* m_1.

Proof We only prove the case when $\min\{m_1, m_2\} > c_\ell$ as the case when $m_1 = m_2 \leq c_\ell$ is trivial. We prove the following statements separately:

(a) For any $z \in \{0, \ldots, c_\ell\}$, $\mathcal{S} \models \varphi(z, m_1)$ if and only if $\mathcal{S} \models \varphi(z, m_2)$.

(b) There is some $z > c_\ell$ such that $\mathcal{S} \models \varphi(z, m_1)$ if and only if there is some $z > c_\ell$ such that $\mathcal{S} \models \varphi(z, m_2)$.

For (a), suppose $\mathcal{S} \models \varphi(z, m_1)$ for some $z \in \{0, \ldots, c_\ell\}$. Then if $\varphi(x, m_1)$ contains a conjunction involving m_1, it must be of the form $c < m_1$, $c \neq m_1$, $x \leq m_1$, $x < m_1$ or $x \neq m_1$. Since $z \leq c_\ell < m_2$, z must also satisfies the same conjunct if m_1 is replaced by m_2. Hence $\mathcal{S} \models \varphi(z, m_2)$. The other direction can be proved in the same way.

For (b), suppose some $z > c_\ell$ satisfies $\varphi(x, m_1)$. Note that $\varphi(x, m_1)$ can be written as $\psi(x) \wedge \varphi'(x, m_1)$ where $\psi(x)$ is a finite conjunction that does not contain the parameter m_1 and $\varphi'(x, m_1)$ is a finite conjunction of literals of the form $c < m_1$, $x \leq m_1$, $x = m_1$ or $m_1 < x$. Since $z > c_\ell$, $\psi(x)$ must not contain any conjunct of the form $x = c$ or $x \leq c$. This means that any $z' > c_\ell$ would satisfy $\psi(x)$. Thus it is sufficient to show that there is some $z' > c_\ell$ that

also satisfies the formula $\varphi'(x, m_2)$. Indeed, if $\varphi'(x, m_1)$ does not contain $m_1 < x$, then m_2 will satisfy $\varphi'(x, m_2)$. Otherwise, $m_2 + 1$ would satisfy $\varphi'(x, m_2)$. This proves the lemma.

Given $\varphi(x, y)$ and $[m_1]$, we can effectively compute any $C \in C_\approx$ such that $\exists z \in C : \varphi(z, m_1)$ by induction on the construction of $\varphi(x, y)$ as follows: If $\varphi(x, y)$ is a literal, then it is clear that we can compute the desired equivalence classes. If $\varphi(x, y)$ is $\varphi_1(x, y) \wedge \varphi_2(x, y)$, then the desired equivalence classes is the intersection of the equivalence classes for $\varphi_1(x, y)$ and $\varphi_2(x, y)$. ∎

The next lemma shows that the atomic operations of $\mathcal{S}$ are consistent with the equivalence relation $\approx$ and can be easily proved.

Lemma 4.5.8 *For any operation $f \in \mathsf{Op}(\mathcal{S})$, for any $m_1, m_2, x_1, x_y \in \mathbb{N}$ such that $m_1 \approx m_2$, $x_1 \approx x_y$, we have*

$$f(m_1, x_1) \approx f(m_2, x_2).$$

We are now ready to describe the reduction from the emptiness problem of $(\mathcal{S}, 1)$-automata to reachability problem of finite graphs. Fix an $(\mathcal{S}, 1)$-automaton $\mathcal{A} = (Q, \alpha, x, \Delta, q_0, F)$. Intuitively, we define a finite graph $G_\mathcal{A}$ by taking the configuration graph of $\mathcal{A}$ and collapsing all the equivalent configurations into one node. Hence nodes in the graph $G_\mathcal{A}$ will be of the form (q, C) where $q \in Q$ and $C \in C_\approx$. Lemma 4.5.7 and Lemma 4.5.8 ensure that the $\mathcal{S}$-automaton recognizes an $\mathbb{N}$-word if and only if there is a path that goes from $(q_0, [x])$ to some node $(q, [y])$ where $q \in F$. Formally, we define the graph $G_\mathcal{A}$ as follows.

Definition 4.5.4 *For any transition $t = (q, (b_0, \ldots, b_\ell), q', f) \in \Delta$, we define the literals $L_{t,0}(x, c_i)$, $\ldots, L_{t,\ell}(x, c_i)$ such that*

$$L_{t,i} = \begin{cases} R_i(x, c_i) & \text{if } b_i = 1 \\ \neg R_0(x, c_i) & \text{otherwise} \end{cases}$$

where $R_0, \ldots, R_\ell = \alpha(q)$. Let $\varphi_t(x, c_0)$ be the $\mathcal{S}$-test $\bigwedge_{i=0}^{\ell} L_{t,i}(x, c_0)$.

The reduced configuration graph *$G_\mathcal{A}$ is $(Q \times C_\approx, E)$, where the edge relation E contains a pair of nodes $((q, C_1), (q', C_2))$ if and only if there is a transition $t = (q, \overline{b}, q', f) \in \Delta$ and $c_0 \in C_1$ such that $\mathcal{S} \models \exists x : \varphi_t(x, c_0)$ and $f(c_0, x) \in C_2$.*

Lemma 4.5.9 *The graph $G_\mathcal{A}$ is effectively computable from $\mathcal{A}$.*

Proof Take a node $(q, C) \in Q \times C_{\approx}$ and a transition $t = (q, (b_0, \ldots, b_\ell), q', f) \in \Delta$. By Lemma 4.5.7, for any $c_0, c_0' \in C$, the same equivalence classes C' satisfy $\exists z \in C' : \mathcal{S} \models \varphi_t(z, c_0)$ and $\exists z \in C' : \mathcal{S} \models \varphi_t(z, c_0')$. Furthermore, all such equivalence classes $C' \in C_{\approx}$ is computable from φ_t and C. By Lemma 4.5.8, for each of such equivalence classes C', there is a unique equivalence class $B \in C_{\approx}$ such that $f(c_0, z) \in B$ for $z \in C'$.

Hence we use the following procedures to compute the out-going edges of (q, C):

- Fix an arbitrary $c_0 \in C$.
- Compute all $C' \in C_{\approx}$ such that $\exists z \in C' : \mathcal{S} \models \varphi_t(z, c_0)$.
- From each such C', take an element z and compute $f(c_0, z)$.
- Add an edge from (q, C) to $(q', [f(c_0, z)])$.

The graph $G_{\mathcal{A}}$ is computed by repeating the above procedures for every node $(q, C) \in Q \times C_{\approx}$. ∎

Finally, we prove the next lemma which concludes the proof of Theorem 4.5.6.

Lemma 4.5.10 *The $(\mathcal{S}, 1)$-automaton $\mathcal{A}$ recognizes some $\mathbb{N}$-word if and only if there is a path that goes from $(q_0, [x])$ to some node $(q, [y])$ where $q \in F$ and $y \in \mathbb{N}$.*

Proof Suppose $(\mathcal{S}, 1)$-automaton $\mathcal{A}$ recognizes an $\mathbb{N}$-word $a_1 a_2 \ldots a_n$. Let

$$(q_0, x), (q_1, m_1), \ldots, (q_n, m_n)$$

be the sequence of configurations produced by the accepting run of $\mathcal{A}$. It is clear by definition of $G_{\mathcal{A}}$ that the sequence $(q_0, [x]), (q_1, [m_1]), \ldots, (q_n, [m_n])$ is a path in $G_{\mathcal{A}}$ and $q_n \in F$.

Conversely, suppose $G_{\mathcal{A}}$ contains a path $(q_0, [m_0]), (q_1, [m_1]), \ldots, (q_n, [m_n])$ where $m_0 = x$ and $q_n \in F$. By definition, there is a transition $t = (q_i, \overline{b}, q_{i+1}, f) \in \Delta$ such that for some $c_0 \in [m_i]$ there is some $x \in \mathbb{N}$ such satisfies $\varphi_t(x, c_0)$ and $f(m_i, x) \in [m_{i+1}]$. By Lemma 4.5.7 and Lemma 4.5.8, for all $c_0 \in [m_i]$ there is some $a_i \in \mathbb{N}$ that satisfies $\varphi_t(a_i, c_0)$, and $f(m_i, a_i) \in [m_{i+1}]$. Hence the $\mathbb{N}$-word $a_1 a_2 \ldots a_n$ is accepted by the $(\mathcal{S}, 1)$-automaton $\mathcal{A}$. ∎

We are now ready to prove Theorem 4.5.6.

of Theorem 4.5.6 To decide the emptiness problem of an $(\mathcal{S}, 1)$-automaton $\mathcal{A}$, we construct the reduced configuration graph $G_{\mathcal{A}}$ and decide whether a path connects $(q_0, [x])$ with $(q, [y])$ for some $q \in F$ and $y \in \mathbb{N}$. ∎

Remark By a similar reduction for Theorem 4.5.5(a), one may reduce the emptiness problem of $(\mathcal{S}_1, 1)$-automata to the emptiness problem of one-way 1-counter machines. Hence the emptiness problem for $(\mathcal{S}_1, 1)$-automaton is decidable (since the emptiness problem for one-way 1-counter machines is decidable [48]). However, it remains to see whether decidability still holds if we change the $+1$ and -1 operations to addition and subtraction in general. Another obvious open question is whether the emptiness problem remains undecidable for $(\mathbb{N}, +, =, \mathrm{pr}_1, \mathrm{pr}_2, 0)$-automata with 2 or 3 registers.

4.5.3 The emptiness problem for constant comparing automata

The previous section shows that one may obtain decidability of the emptiness problem by restricting the number of changing registers of the $\mathcal{S}$-automata. Another way of restricting the automata is to put constraints on the allowable transitions of the automata. We show in this section that by allowing only those transitions that compare the input or a changing register with constants, the emptiness problem may become decidable. Our motivation is to analyze those algorithm in which comparisons occur only between variables and a fixed number of constant values. For example we may allow the comparison $a < 5$ but not the comparison $a < b$ where a, b are variables. With this in mind, we now introduce a class of automata which we call the *constant comparing automata*. We would like to show such automata have a decidable emptiness problem. For the sake of convenience we add the relation $U = \mathbb{N}^2$ to our structures. This can be done without any loss of generality.

Definition 4.5.5 *Let $\mathcal{S}$ be a structure that contains $=$ as an atomic relation and ℓ constants $c_1, \ldots, c_\ell$ in its signature. A constant comparing $(\mathcal{S}, k)$-automaton is $\mathcal{A} = (Q, \alpha, \overline{y}, \Delta, q_0, F)$ such that for every $q \in Q$, $\alpha(q) = (R_1, \ldots, R_{k+\ell})$ satisfies the following conditions:*

- *There is at most one $i \in \{1, \ldots, k\}$ such that R_i is the $=$ relation.*
- *For all $j \in \{1, \ldots, k\}$ apart from i (if it exists) we have $R_j = U$.*

The $(\mathcal{S}, k)$-automaton $\mathcal{A}$ is a strongly constant comparing $\mathcal{S}$-automaton *if no such i exists.*

Note that by definition an $(\mathcal{S}, 1)$-automaton is also a constant comparing $\mathcal{S}$-automaton. Hence the next theorem can be viewed as a generalization of Theorem 4.5.6.

Theorem 4.5.11 *Let $\mathcal{S}$ be the structure $(\mathbb{N}; +, \times, \mathrm{pr}_1, \mathrm{pr}_2, =, \leq, U, c_1, \ldots, c_\ell)$ where $c_1, \ldots, c_\ell$ are arbitrary constants in $\mathbb{N}$. The emptiness problem for constant comparing $\mathcal{S}$-automata is decidable.*

The proof proceeds in the same manner as the proof for Theorem 4.5.6. We define the equivalence relation $\approx_k$ on $\mathbb{N}^k$ in a similar way as $\approx$.

Definition 4.5.6 *We say that two k-tuples $\overline{m} = (m_1, \ldots, m_k)$ and $\overline{s} = (s_1, \ldots, s_k)$ are $(\mathcal{S}, k)$-equivalent if for every $i \in \{1, \ldots, k\}$ we have $m_i \approx s_i$. In this case we write $\overline{m} \approx_k \overline{s}$. We use $[\overline{m}]$ to denote the $\approx_k$-equivalence class of $\overline{m}$ and $C_{\approx_k}$ denotes the set of equivalence classes of $\approx_k$.*

Clearly $C_{\approx_k}$ is a finite set.

Definition 4.5.7 *An $\mathcal{S}$-*test *is a quantifier-free formula $\varphi(z, x_1, \ldots, x_k)$ $(k > 0)$ over $\mathcal{S}$ that is a finite conjunction of literals of the form*

$$z \leq c,\ z > c,\ z = c,\ z \neq c.$$

where c is a constant in $\mathcal{S}$ and exactly one literal of the form $z = x_i$ or $z \neq x_i$ for $i \in \{1, \ldots, k\}$.

The proof of the following lemma depends on the fact that the input can only be tested for equality against at most one changing register.

Lemma 4.5.12 *Let $\varphi(y, x_1, \ldots, x_k)$ be an $\mathcal{S}$-test and let $\overline{m}, \overline{s} \in \mathbb{N}^k$ be such that $\overline{m} \approx_k \overline{s}$. For any equivalence class $C \in C_\approx$, there is $z \in C$ such that $\mathcal{S} \models \varphi(z, \overline{m})$ if and only if there is $z \in C$ such that $\mathcal{S} \models \varphi(z, \overline{s})$. Furthermore, the set of equivalence classes C such that $\exists z \in C : \mathcal{S} \models \varphi(z, \overline{m})$ can be effectively computed from φ and $[m]$.*

Proof Note that if φ does not contain any literal of the form $y = x_i$ or $y \neq x_i$, then either $\mathcal{S} \models \varphi(z, x_1, \ldots, x_k)$ for all $z \in N$ or there exists no such z. Hence we only need to consider the case when φ contains exactly one such literal. Let $b \in \{1, \ldots, k\}$ be such that the literal $y = x_b$ or $y \neq x_b$ occurs in φ. We need only consider the case when $\min(m_b, s_b) > c_\ell$ since the case $m_b = s_b$ is trivial.

Fix an equivalence class $C \in C_\approx$ and assume that there is $z \in C$ such that $\mathcal{S} \models \varphi(z, \overline{m})$. If the literal $y = x_b$ occurs in φ, it must be the case that $z = m_b$ and since $m_b > c_{c_\ell}$, we have $z > c_\ell$. Also we must have $s_b > c_\ell$ and therefore there exists a $z \in C$ such that $\mathcal{S} \models \varphi(z, \overline{s})$. If the literal $y \neq x_b$ occurs in φ, it must be the case that $z \neq m_b$. If $z \leq c_\ell$, then it must be the case that $z \neq s_b$ since $s_b > c_\ell$ and hence we have $\mathcal{S} \models \varphi(z, \overline{s})$. Otherwise if $z > c_\ell$, it must

be the case that no literal of the form $y \leq c$ and $y = c$ occurs in φ. Then there must exist a $z \in C$ such that $\mathcal{S} \models \varphi(z, \overline{s})$.

The proof of the other direction is symmetric. The proof of the fact that the all such equivalence classes C such that $\exists z \in C : \mathcal{S} \models \varphi(z, \overline{m})$ can be effectively computed from φ and $[m]$ is exactly the same as lemma 4.5.7. ∎

The next lemma follows directly from lemma 4.5.8.

Lemma 4.5.13 *For any k-tuple of operations $(f_1, \ldots, f_k)$ where each $f_i \in \mathsf{Op}(\mathcal{S})$, for any $\overline{m}, \overline{s} \in \mathbb{N}^k$ and $x_1, x_2 \in \mathbb{N}$ such that $\overline{m} \approx_k \overline{s}, x_1 \approx x_2$ we have*

$$(f_1(m_1, x_1), \ldots, f_k(m_k, x_1)) \approx_k (f_1(s_1, x_2), \ldots, f_k(s_k, x_2)).$$

Proof We need to show that for every $i \in \{1, \ldots, k\}$ we have $f_i(m_i, x_1) \approx f_i(s_i, x_2)$. Since $m_i \approx s_i$ (by definition of $\approx_k$) and $x_1 \approx x_2$, by Lemma 4.5.8 we must have $f_i(m_i, x_1) \approx f_i(s_i, x_2)$. ∎

Definition 4.5.8 *For any transition $t = (q, (b_1, \ldots, b_{k+\ell}), q', (f_1, \ldots, f_k)) \in \Delta$, let $\alpha(q) = (R_1, \ldots, R_{k+\ell})$. Then we define the literals $L_{t,1}(z, x_1), \ldots, L_{t,k}(z, x_k)$ such that*

$$L_{t,i}(z, x_i) = \begin{cases} R_i(z, x_i) & \text{if } b_i = 1 \\ \neg R_i(z, x_i) & \text{otherwise} \end{cases}$$

We also define the literals $L_{t,k+1}(z, c_1), \ldots, L_{t,k+\ell}(z, c_{k+\ell})$ such that

$$L_{t,i}(z, c_i) = \begin{cases} R_i(z, c_i) & \text{if } b_i = 1 \\ \neg R_i(z, c_i) & \text{otherwise} \end{cases}$$

Let $\varphi_t(z, x_1, \ldots, x_k)$ be the $\mathcal{S}$-test $\bigwedge_{i=1}^{k} L_{t,i}(z, x_i) \wedge \bigwedge_{i=k+1}^{k+\ell} L_{t,i}(z, c_i)$.

The reduced configuration graph *$G_{\mathcal{A}}$ is $(Q \times C_{\approx_k}, E)$, where the edge relation E contains a pair of nodes $((q, C_1), (q', C_2))$ if and only if there is a transition $t = (q, \overline{b}, q', (f_1, \ldots, f_k)) \in \Delta$ and $\overline{m} = (m_1, \ldots, m_k) \in C_1$ such that $\mathcal{S} \models \exists x : \varphi_t(x, \overline{m})$ and $(f_1(m_1, x), \ldots, f_k(m_k, x)) \in C_2$.*

Lemma 4.5.14 *The graph $G_{\mathcal{A}} = (Q \times C_{\approx_k}, E)$ is effectively computable from $\mathcal{A}$.*

Proof It suffices to prove that the edge relation E is effectively computable. Therefore we need to effectively decide whether $((q, C_1), (q', C_2))$ is an edge of $G_{\mathcal{A}}$ or not (here $q_1, q_2 \in Q$

and $C_1, C_2 \in C_{\approx_k}$). By Lemma 4.5.12, for $\overline{m}, \overline{s} \in C_1$ the same equivalence classes $C' \in C_{\approx}$ satisfy $\exists z \in C' : \mathcal{S} \models \varphi_t(z, \overline{m})$ and $\exists z \in C' : \mathcal{S} \models \varphi_t(z, \overline{s})$. Also all such equivalence classes are effectively computable from φ and C_1. Furthermore, by Lemma 4.5.13 there is a unique equivalence class $B \in C_{\approx_k}$ such that $(f_1(m_1, z), \ldots, f_k(m_k, z)) \in B$.

Keeping the above facts in mind, the edge relation can be computed effectively in a manner similar to Lemma 4.5.9. ∎

The following lemma reduces the emptiness problem for constant comparing $(\mathcal{S}, k)$-automata to finding a path from $(q_0, [\overline{y}])$ to some node $(q, [\overline{s}])$ in the reduced configuration graph.

Lemma 4.5.15 *The constant comparing $(\mathcal{S}, k)$-automaton $\mathcal{A}$ recognizes a $\mathbb{N}$-word if and only if there is a path that goes from $(q_0, [\overline{y}])$ to some node $(q, [\overline{s}])$ in the reduced configuration graph $G_{\mathcal{A}}$ where $q \in F$ and $\overline{s} \in \mathbb{N}^k$.*

Proof The proof uses lemmas 4.5.12, 4.5.13, 4.5.14 and uses a similar argument to that of lemma 4.5.10 ∎

Theorem 4.5.11 clearly follows from Lemma 4.5.15. Note that the structure $\mathcal{S}$ in Theorem 4.5.11 does not contain subtraction as part of the signature. If subtraction is added to the signature, one may show by slightly modifying the proof of Theorem 4.5.5(a) that the emptiness problem becomes undecidable. However, the emptiness problem becomes decidable if we restrict to strongly constant comparing $\mathcal{S}$-automata. Formally we prove the following theorem.

Theorem 4.5.16 *Let $\mathcal{S} = (\mathbb{N}; +, -, \mathrm{pr}_1, =, \leq, c_1, \ldots, c_\ell)$ where $c_1, \ldots, c_\ell$ are constants in $\mathbb{N}$.*

(a) The emptiness problem for constant comparing $(\mathcal{S}, 2)$-automata is undecidable if $\ell \geq 2$.

(b) The emptiness problem for strongly constant comparing $\mathcal{S}$-automata is decidable.

Below we first prove (a). The rest of the section proves (b).

Proof of Theorem 4.5.16(a) For (a), one constructs for every one-way 2-counter machine $\mathcal{A}$ a constant comparing $(\mathcal{S}, 2)$-automaton $\mathcal{A}'$ such that $\mathcal{A}$ accepts a non-empty language if and only if $L(\mathcal{A}') \neq \emptyset$. The construction is very similar to the proof of Theorem 4.5.5(a). Note that we need to use two constant symbols c_1, c_2 in the signature of $\mathcal{S}$. Assume $c_1 < c_2$.

The initial values of $\mathcal{A}'$ are c_1. Whenever the counter machine $\mathcal{A}$ increments/decrements its counters, the $\mathcal{S}$-automaton $\mathcal{A}'$ adds/subtracts c_2 from its registers. Whenever $\mathcal{A}$ tests its counter against 0, $\mathcal{A}'$ compares the register values with c_1. ∎

For (b), one uses a reduction to the reachability problem for Vector Addition Systems with States (VASS) [60]. An *k-dimensional VASS* $\mathcal{V}$ is a tuple (Q, Σ, Δ) where Q is a finite set of states, Σ is a finite alphabet, $\Delta \subseteq Q \times \Sigma \times Q \times \mathbb{Z}^k$ is a transition relation. A *configuration of* $\mathcal{V}$ is a pair $(q, \overline{m})$ where $q \in Q$ and $\overline{m} \in \mathbb{N}^k$. Given a configuration $(q_0, \overline{m}_0) \in Q \times \mathbb{N}^k$, a *run from* $(q_0, \overline{m}_0)$ of $\mathcal{V}$ on a word $\sigma_1\sigma_2 \dots \sigma_m \in \Sigma^*$ is a sequence of configurations $(q_0, \overline{m}_0), (q_1, \overline{m}_1), \dots, (q_n, \overline{m}_n)$ such that $(q_{i-1}, \sigma_i, q_i, \overline{m}_i - \overline{m}_{i-1}) \in \Delta$. VASSs are known to be equivalent to Vector Addition Systems (VASs) and also to Petri nets. The *reachability problem* is a well-studied problem and is stated as follows: Given an n-dimensional VASS and two configurations $(q, \overline{m})$ and $(q', \overline{n})$, is there a run from $(q, \overline{m})$ that reaches $(q', \overline{n})$? This problem is decidable by [61].

The next lemma proves (b) in Theorem 4.5.16

Lemma 4.5.17 *One may effectively construct, given a strongly constant comparing $(\mathcal{S}, k)$-automaton $\mathcal{A}$, a k-dimensional VASS $\mathcal{V}_\mathcal{A}$ and configurations $(q, \overline{m})$, $(q', \overline{n})$ of $\mathcal{V}_\mathcal{A}$ such that $L(\mathcal{A}) \neq \emptyset$ if and only if there is a run of $\mathcal{V}_\mathcal{A}$ from $(q, \overline{m})$ to $(q', \overline{n})$.*

Proof Let $\mathcal{A} = (Q, \alpha, \overline{m}, \Delta, q_0, F)$ be a strongly constant comparing $(\mathcal{S}, k)$-automaton. Recall that $\mathcal{S}$ has in its signature ℓ constant symbols $c_1, \dots, c_\ell$. By definition, for any $q \in Q$, the first k components of $\alpha(q)$ are all Us. This means that the transitions the $\mathcal{S}$-automaton may take at state q do not depend on the tests made on the register values. Hence we may assume that the transition Δ of $\mathcal{A}$ is a subset of $Q \times \{0,1\}^\ell \times Q \times \{+, -, \mathrm{pr}_1\}^k$. Suppose a run of the $\mathcal{S}$-automaton reaches state q and $\mathcal{A}$ reads the next input $x \in \mathbb{N}$. A transition $(q, \overline{b}, q', \overline{g})$ (where $\overline{b} \in \mathbb{N}^\ell$) can be taken when $x = c_i$ if $b_i = 1$ and $x \neq c_i$ if $b_i = 0$ for all $i \in \{1, \dots, \ell\}$.

We construct the k-dimensional VASS $\mathcal{V}_\mathcal{A}$ as follows. The alphabet of $\mathcal{V}_\mathcal{A}$ is the set $\{c_1, \dots, c_\ell, c_{\ell+1}\}$ where $c_{\ell+1}$ is different from any of $c_1, \dots, c_\ell$. The VASS $\mathcal{V}_\mathcal{A}$ contains all states in Q. For any transition $(q, \overline{b}, q', \overline{g}) \in \Delta$, we do the following:

(i) If $b_i = 1$ for exactly one $i \in \{1, \dots, \ell\}$, then create a transition

$$(q, c_i, q', (m_1, \dots, m_k))$$

in T where $m_j = c_i$ if $g_j = +$, $m_j = 0$ if $g_j = \mathrm{pr}_1$ and $m_j = -c_i$ if $g_j = -$ for every $j \in \{1, \ldots, k\}$.

(ii) If $b_i = 0$ for all $i \in \{1, \ldots, \ell\}$, then create a fresh state r in $\mathcal{V}_{\mathcal{A}}$ and add the following transitions in T:

- Suppose without loss of generality that c_ℓ is the largest constant in $\mathcal{S}$. For every $y \in \{0, \ldots, c_\ell\} - \{c_1, \ldots, c_\ell\}$, add a transition $(q, c_{i+1}, q', \overline{x})$ where $x_j = y$ if $g_j = +$, $x_j = 0$ if $g_j = \mathrm{pr}_1$ and $x_j = -y$ if $g_j = -$.
- Add a transition $(q, c_{i+1}, r, \overline{x})$ where $x_j = c_\ell + 1$ if $g_j = +$, $x_j = 0$ if $g_j = \mathrm{pr}_1$ and $x_j = -c_\ell - 1$ if $g_j = -$.
- Add a transition $(r, c_{i+1}, r, \overline{x})$ where $x_j = 1$ if $g_j = +$, $x_j = 0$ if $g_j = \mathrm{pr}_1$ and $x_j = -1$ if $g_j = -$.
- Add a transition $(r, c_{i+1}, q', 0^k)$.

Intuitively, suppose a run of $\mathcal{V}_{\mathcal{A}}$ reaches state $q \in Q$ and reads a input $c \in \{c_1, \ldots, c_{\ell+1}\}$. If $c = c_i$ for some $i \in \{1, \ldots, \ell\}$, the VASS $\mathcal{V}_{\mathcal{A}}$ will update the current vector according to the transition $(q, \overline{b}, q', \overline{g}) \in \Delta$ where $b_i = 1$ and $b_j = 0$ for all $j \in \{1, \ldots, \ell\} - \{i\}$. If $c = c_{\ell+1}$, then the VASS $\mathcal{V}_{\mathcal{A}}$ will update the current vector according to the transition $(q, 0^\ell, q', \overline{g})$ in two possible ways: it either (1) changes the values of the vectors by some $c \notin \{c_1, \ldots, c_\ell\}$ and moves to q' directly, or (2) first changes the values of the vectors by $c_\ell + 1$ and moves to state r, then increments or decrements the vector values by some arbitrary amount before moving to state q''.

Finally, we complete the construction of $\mathcal{V}_{\mathcal{A}}$, we add a fresh state q_{final} to $\mathcal{V}_{\mathcal{A}}$, and then add the following transitions:

- For every $q \in F$, add a transition $(q, c_{i+1}, q_{\mathsf{final}}, 0^k)$
- Add transitions $(q_{\mathsf{final}}, c_{i+1}, q_{\mathsf{final}}, \overline{x})$ where $\overline{x} = \{-1, 0, 1\}^k$.

Note that once a state q is reached by a run in $\mathcal{V}_{\mathcal{A}}$, then the run can be extended to reach the configuration $(q_{\mathsf{final}}, \overline{n})$ for any $\overline{n} \in \mathbb{N}^k$.

Recall that q_0 and $\overline{m}$ are respectively the initial state and initial values of $\mathcal{A}$. It is now easy to see that the $\mathcal{S}$-automaton $\mathcal{A}$ accepts some $\mathbb{N}$-word if and only if there is a run of $\mathcal{V}_{\mathcal{A}}$ from $(q_0, \overline{m})$ to the configuration $(q_{\mathsf{final}}, 0^k)$. This finishes the proof of the lemma. ∎

Proof of Theorem 4.5.16(b) The statement immediately follows from Lemma 4.5.17 and the fact that the reachability problem for VASS is decidable. This finishes the proof of Theorem 4.5.16. ∎

Remark Theorem 4.5.16 showed that in a precise way, a strongly constant comparing $\mathcal{S}$-automaton is coded into a VASS. We may show that the other direction is also possible in the following precise sense: Given a VASS $\mathcal{V}$ of dimension d and configurations $(q, \overline{m}), (q', \overline{n})$, we may effectively compute a strongly constant $\mathcal{S}$-automaton $\mathcal{A}_{\mathcal{V}}$ and two states r, r' such that there is a run from $(q, \overline{m})$ to $(q', \overline{n})$ in $\mathcal{V}$ if and only if there is a run from the initial state to r but no run from the initial state to r' in $\mathcal{A}_{\mathcal{V}}$.

We assume that for every integer $x \in \mathbb{Z}$ that occurs on a transition of $\mathcal{V}$, $|x|$ is included as a constant in $\mathcal{S}$. Additionally for every $\sigma \in \Sigma$, we have a unique constant $c_\sigma \in \mathbb{N}$ in $\mathcal{S}$ and $|n_1|, \ldots, |n_d|$ and 1 are all included as constants in $\mathcal{S}$. We construct the $\mathcal{S}$-automaton $\mathcal{A}_{\mathcal{V}}$ as follows.

The states of $\mathcal{A}_{\mathcal{V}}$ include all the states of $\mathcal{V}$, the states r, r' and some additional states as we will explain shortly. The initial state of $\mathcal{A}_{\mathcal{V}}$ is q and it has d changing registers. The initial value of $\mathcal{A}_{\mathcal{V}}$ is $\overline{m}$. Consider a transition $(s, \sigma, s', \overline{y})$ of $\mathcal{V}$. When the automaton $\mathcal{A}_{\mathcal{V}}$ is in state s, it compares the input with the constant c_σ. If the input is equal to c_σ, the automaton moves to a new state h. The automaton processes the next d inputs as follows: for every $i \in \{1, \ldots, d\}$ the automaton compares the input with the constant $|y_i|$. If equal, the automaton adds or subtracts $|y_i|$ from the i^{th} register (depending on whether $y_i \geq 0$ or $y_i < 0$). After updating all the registers in this manner, the automaton moves to state s'.

From the state q', the automaton processes the next d inputs as follows: for every $i \in \{1, \ldots, d\}$ the automaton compares the input with the constant $|n_i|$ and if equal it subtracts $|n_i|$ from the i^{th} register if $n_i \geq 0$ and adds it to the i^{th} register otherwise. After updating the registers, the automaton moves to the state r. From state r, the automaton has d outgoing transitions. For each $i \in \{1, \ldots, d\}$, the corresponding transition compares the input with the constant 1 and if equal, subtracts 1 from the i^{th} register and moves to state r'.

From the definition of $\mathcal{A}_{\mathcal{V}}$, it is easy to see that there is a run from $(q, \overline{m})$ to $(q', \overline{n})$ if and only if there is a run from the initial state to r but no run from the initial state to r' in $\mathcal{A}_{\mathcal{V}}$.

Chapter 5

Infinite games played on trees with back-edges

In this chapter, we concentrate on algorithms for solving the winning region problem for games with Büchi winning conditions played on trees with back-edges. In particular, we present an efficient algorithm that solves a Büchi game played on trees with back-edges. The algorithm runs in time $O(\min\{r \cdot m, \ell + m\})$ where m is the number of edges in the graph, r is the largest rank of a snare and ℓ is the external path length, i.e., sum of the distances from the root and all leaves, in the underlying tree. We then apply our analysis for the case of Büchi games to solve the winning region problem for parity games played on trees with back-edges and show that they can be solved in $O(\ell + m)$.

J. Obdržálek in his work [72] (Chapter 3) outlines a proof that parity games played on trees with back-edges are solved in polynomial time. The work does not provide a detailed analysis of the algorithm but rather concentrates on attacking the general parity game problem. The algorithm detects whether Player 0 wins from the root and it is claimed that this can be done in time $O(m)$ (where m is the number of edges in the graph).

We would like to point out that the running time of any algorithm for solving the winning region problem is heavily dependent on the data structures and underlying model of computation. In particular, under the reasonable assumption that trees with back-edges are encoded as binary strings it can be shown that $O(m \cdot log(m))$ bits are necessary to encode a tree with back-edges with $O(m)$ edges. This can be seen as follows: for a tree $\mathcal{T}$, we determine the main branch by starting from the root and always choosing the next node v such that v has the most number of nodes below it. Now consider a tree $\mathcal{T}$ with $O(m)$

edges such that the main branch has $m/2$ nodes and the first $m/4$ nodes of the main branch have exactly one offbranching leaf. Below the first $(m/4)^{th}$ nodes of the main branch, there is a full binary tree with $m/2$ nodes and $m/4$ leaves. If we now consider the class of trees with back-edges that can be obtained from $\mathcal{T}$ by adding exactly one back-edge per leaf, it can be seen that there are $O((m/4)^{(m/4)})$ trees with back-edges in this class. Hence we need at least $O(m \cdot log(m))$ bits to encode a tree with back-edges with $O(m)$ edges. This fact shows that any algorithm to solve the winning region problem for games on trees with back-edges must have a running time of al least $O(m \cdot log(m))$. Hence it is not clear how the time bound of $O(m)$ of [72] can be achieved when trees with back-edges are encoded using binary strings.

Notwithstanding the above observations, the algorithm of [72] can be modified to run in $O(h \cdot m)$ (where h is the height of the underlying tree). Here we give an alternative algorithm for solving parity games on trees with back-edges based on our analysis for Büchi games played on trees with back-edges which has a running time of $O(\ell + m)$. Since ℓ maybe much smaller than $h \cdot m$, our algorithm performs better than the time bound of $O(h \cdot m)$ in many cases. Importantly, we clarify the data structures and model of computation used (see section 5.1) and hence analyze the problem in greater detail.

5.1 Trees with back-edges

Trees with back-edges are widely used and studied in computer science. The paper [22] studies counterexamples in model checking whose transition diagrams are trees with back-edges. Furthermore, as discussed in [7], they form a natural class of directed graphs that has directed tree-width 1 and unbounded entanglement. Also, consider the trees generated by depth-first search. If the original graph has only tree-edges and back-edges but no cross-edges, then the algorithms described in this chapter can be used. Another use of trees with back-edges is in μ-calculus where the syntax graph of a μ-calculus formula is a tree with back-edges [8]. As pointed out in [8] a finite Kripke structure can be viewed as a tree with back-edges by performing a partial unraveling of the structure.

Foramlly, we consider rooted directed trees where all edges are directed away from the root. All terminologies on trees are standard. The ancestor relation on a tree $\mathcal{T}$ is denoted by $\leq_{\mathcal{T}}$ and the root is its least element. For $u \leq_{\mathcal{T}} v$, let $\text{Path}[u, v] = \{x \mid u \leq_{\mathcal{T}} x \leq_{\mathcal{T}} v\}$. The *level* $\mathsf{lev}(u)$ of a node $u \in V$ is the length of the unique path from the root to u. The *height* h

of the tree is $\max\{\mathsf{lev}(v) \mid v \in V\}$. The *external path length* ℓ is $\sum\{\mathsf{lev}(v) \mid v \text{ is a leaf in } \mathcal{T}\}$.

Definition 5.1.1 *A directed graph $G = (V, E)$ is a* tree with back-edges *if $E = E^{\mathsf{T}} \cup E^{\mathsf{B}}$ where $\mathcal{T} = (V, E^{\mathsf{T}})$ is a rooted directed tree and all edges in E^{B} are of the form (v, u) where $u <_{\mathcal{T}} v$. Edges in E^{B} are called* back-edges*. We denote a tree with back-edges by $(V, E^{\mathsf{T}} \cup E^{\mathsf{B}})$. We refer to leaves of $\mathcal{T}$ as* leaves of G*. A Büchi game is* played on a tree with back-edges *if its underlying graph is a tree with back-edges.*

Let $G = (V, E^{\mathsf{T}} \cup E^{\mathsf{B}})$ be a tree with back-edges. A *subtree* of G is a subgraph of G that is also a tree. In particular, all the edges of a subtree are from E^{T} and the root of the subtree is not necessarily the root of G. A *subtree with back-edges* consists of a subtree and all induced back-edges on the subtree. We use $\mathcal{B}$ to denote the class of all Büchi games played on trees with back-edges. A game $\mathcal{G} \in \mathcal{B}$ is denoted by the tuple $(V_0, V_1, E^{\mathsf{T}}, E^{\mathsf{B}}, T)$ where $T \subseteq V$ are target nodes. Recall that we may assume without loss of generality that for any node $u \in V$, we have $E^{\mathsf{T}}(u) \cup E^{\mathsf{B}}(u) \neq \emptyset$.

In the rest of the chapter we will present our algorithm for solving the winning region problem for games in $\mathcal{B}$. Our claim on the running time of the algorithm depends on the following assumptions on the data structures and the underlying model of computation. A node u in a game $\mathcal{G} \in \mathcal{B}$ is stored as $(p(u), \mathrm{tar}(u), \mathrm{pos}(u), \mathrm{Ch}(u), \mathrm{InBk}(u), \mathrm{OutBk}(u))$, where $p(u)$ is a pointer to the parent of u, $\mathrm{tar}(u)$ is **true** if and only if $u \in T$, $\mathrm{pos}(u) = \sigma$ if and only if $u \in V_\sigma$, $\mathrm{Ch}(u)$ is a list of children of u, $\mathrm{InBk}(u)$ is a list of incoming back-edges into u, $\mathrm{OutBk}(u)$ is a list of outgoing back-edges from u. We assume that the underlying model of computation is a random access machine (RAM) and that manipulating registers (of logarithmic lengths) takes constant time. Hence, accessing $p(u)$, $\mathrm{tar}(u)$, $\mathrm{pos}(u)$ as well as the first elements of $\mathrm{Ch}(u)$, $\mathrm{InBk}(u)$ and $\mathrm{OutBk}(u)$ takes constant time. In the following we define a canonical form for all games $\mathcal{G} \in \mathcal{B}$.

Definition 5.1.2 *A Büchi game $\mathcal{G} \in \mathcal{B}$ is* reduced *if the following conditions hold:*

- *For all $(u, v) \in E^{\mathsf{B}}$, u is a leaf in the underlying tree with back-edges $(V, E^{\mathsf{T}} \cup E^{\mathsf{B}})$.*
- *All target nodes are leaves.*
- *Each leaf in $(V, E^{\mathsf{T}} \cup E^{\mathsf{B}})$ has exactly one outgoing back-edge.*

The following lemma is easy to see.

Lemma 5.1.1 *Suppose $\mathcal{G}$ is a reduced Büchi game and π is a play in $\mathcal{G}$. Let R be the set of leaves that are visited infinitely often by π. Then we have*

$$\mathrm{Inf}(\pi) = \bigcup \{Path[v,u] \mid (u,v) \in E^{\mathsf{B}}, u \in R\}.$$

The next lemma reduces solving games in the class $\mathcal{B}$ to solving games that are reduced.

Lemma 5.1.2 *Given a Büchi game $\mathcal{G} = (V_0, V_1, E_1^{\mathsf{T}}, E_1^{\mathsf{B}}, T) \in \mathcal{B}$, there exists a reduced game $\mathrm{Rd}(\mathcal{G}) = (U_0, U_1, E_2^{\mathsf{T}}, E_2^{\mathsf{B}}, S)$ such that*

- *$V \subseteq U$ and $|U| \leq |V| + |E_1^{\mathsf{B}}|$.*
- *A node $v \in V$ is winning for Player 0 in $\mathcal{G}$ if and only if v is winning for Player 0 in $\mathrm{Rd}(\mathcal{G})$.*
- *$\mathrm{Rd}(\mathcal{G})$ is constructed from $\mathcal{G}$ in time $O(|E_1^{\mathsf{T}} \cup E_1^{\mathsf{B}}|)$.*

Proof The game $\mathrm{Rd}(\mathcal{G})$ is constructed from $\mathcal{G}$ as follows :

1. For each back-edge $(u,v) \in E_1^{\mathsf{B}}$, add a new leaf $\alpha(u,v)$ and subdivide the edge (u,v) into $(u, \alpha(u,v))$ and $(\alpha(u,v), v)$.
2. $S = \{\alpha(u,v) \mid \mathrm{Path}[v,u] \cap T \neq \emptyset\}$.

See Fig. 5.1 for an example. It is easy to see that $\mathrm{Rd}(\mathcal{G})$ is a reduced game and that $V \subseteq U$ and $|U| \leq |V| + |E_1^{\mathsf{B}}|$. Suppose $v \in V$ is winning for Player 0 in $\mathcal{G}$. Let f be the winning strategy for Player 0 in $\mathcal{G}$ from v. We define a strategy g for Player 0 in the game $\mathrm{Rd}(\mathcal{G})$ such that

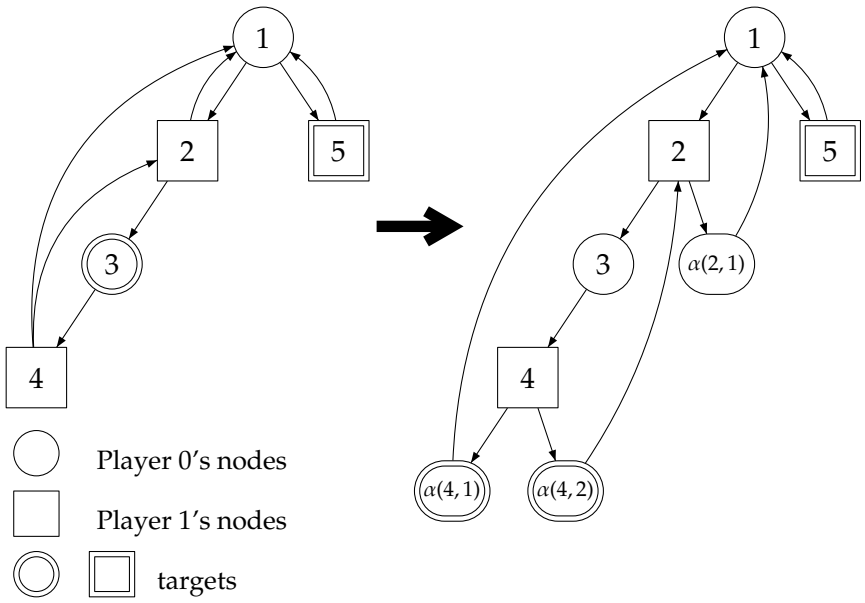
$$g(u) = \begin{cases} f(u) & \text{if } f(u) \in E_1^{\mathsf{T}}(u), \\ \alpha(u, f(u)) & \text{if } f(u) \in E_1^{\mathsf{B}}(u). \end{cases}$$

Let π be a play starting at v that is consistent with g. Then π can be written as a sequence in the following form:

$$u_0, u_1, \cdots, u_{i_0}, \alpha(u_{i_0}, u_{i_0+1}), u_{i_0+1}, \cdots, u_{i_1}, \alpha(u_{i_1}, u_{i_1+1}), u_{i_1+1}, \cdots$$

where $u_0 = v, u_1, u_2, \ldots$ are nodes in V. Note further that $u_0, u_1, \ldots$ forms a play π' in $\mathcal{G}$ that is consistent with f. Consider a node $u_j \in T$ occurring in π' and let u_{j+k} be the first node that occurs in π' after u_j such that $(u_{j+k}, u_{j+k+1}) \in E_1^{\mathsf{B}}$. There are two cases: (a) $u_{j+k+1} \leq_{\mathcal{T}} u_j$ in which case the node $\alpha(u_{j+k}, u_{j+k+1}) \in S$ occurs in π after u_j. (b) $u_{j+k+1} >_{\mathcal{T}} u_j$ in which case

Figure 5.1: Example of a Büchi game played on a tree with back edges and the equivalent reduced game.



there must be a target node in $\mathrm{Path}[u_{j+k+1}, u_{j+k}]$ in $\mathcal{G}$. If this were not the case, π' would be a winning play for Player 1 contradicting our assumption that f is a winning strategy for Player 0. Hence the node $\alpha(u_{j+k}, u_{j+k+1}) \in S$ occurs in π after u_j.

In both the cases, whenever a target node u_j occurs in π', a node $\alpha(u_{j+k}, u_{j+k+1}) \in S$ occurs in π. By the assumption that f is a winning strategy for Player 0, we have $\mathrm{Inf}(\pi') \cap T \neq \emptyset$. It then follows that $\mathrm{Inf}(\pi) \cap S \neq \emptyset$.

On the other hand, suppose $v_0 \in V$ is winning for Player 0 in $\mathrm{Rd}(\mathcal{G})$. Let $f : U_0 \to U$ be the winning strategy for Player 0 starting from v. We define a strategy $g : V_0 \to V$ for Player 0 in $\mathcal{G}$ such that

$$g(u) = \begin{cases} f(u) & \text{if } f(u) \in V, \\ w & \text{if } f(u) \notin V \text{ and } w \in E_2^{\mathsf{B}}(f(u)). \end{cases}$$

Take a play $\pi = v_0, v_1, \ldots$ consistent with g. If we subdivide all back edges (v_i, v_{i+1}) in this sequence by the node $\alpha(v_i, v_{i+1})$, we obtain a play π' consistent with f and $\mathrm{Inf}(\pi') \cap V = \mathrm{Inf}(\pi)$. By Lemma 5.1.1, $\mathrm{Inf}(\pi') = \bigcup\{\mathrm{Path}[v, u] \mid u \in R, (u, v) \in E_2^{\mathsf{B}}\}$ where R is the set

of leaves in $\mathrm{Rd}(\mathcal{G})$ that π' visits infinitely often. Since π' is a winning play for Player 0, there is a node $\alpha(u,v) \in R \cap S$. This means that $\mathrm{Path}[v,u] \cap T \neq \emptyset$. Hence we have $\mathrm{Inf}(\pi) = \mathrm{Inf}(\pi') \cap V \supseteq \mathrm{Path}[v,u]$ and π is a winning play for Player 0 in $\mathcal{G}$.

The *target level* of a node $u \in V$ is the number of target nodes that occur on the unique path from the root to u. To construct $\mathrm{Rd}(\mathcal{G})$ from $\mathcal{G}$, we first compute the levels and target levels for all nodes in $V_0 \cup V_1$. This can be done by a preorder traversal of the tree starting from the root. We use $k(u)$ and $\ell(u)$ to denote resp. the target level and level of u. The value of $k(u)$ and $\ell(u)$ are set to 0 when u is the root. We increment the ℓ-value by 1 as the tree traversal visits a node of a higher level. If a target node u is visited, we increase the k-value by 1; see Algorithm 1. The algorithm is executed with parameters $(r,0)$ where r is the root of $(V, E_1^{\mathsf{T}} \cup E_1^{\mathsf{B}})$.

Algorithm 1 AssignLabel(u,i).

```
1: if tar(u) then k(u) ← i + 1
2: else k(u) ← i end if
3: for v ∈ Ch(u) do
4:     ℓ(v) ← ℓ(u) + 1
5:     Run AssignLabel(v, k(u)).
6: end for
```

The algorithm then copies the nodes and edges in the tree (V, E_1^{T}) to $\mathrm{Rd}(\mathcal{G})$. When a back-edge (u,v) is detected, the algorithm creates a new node $\alpha(u,v)$, and connects u (resp. $\alpha(u,v)$) with $\alpha(u,v)$ (resp. v). Finally, the node $\alpha(u,v)$ is set as a target in S if u and v have different target levels. Algorithm 1 runs in time $O(|V|)$ because the algorithm visits each node in the tree exactly once. The construction of $\mathrm{Rd}(\mathcal{G})$ takes $O(|E_1^{\mathsf{T}} \cup E_1^{\mathsf{B}}|)$ time because each edge (tree edge or back-edge) in $\mathcal{G}$ is visited exactly once. ∎

5.2 Solving Büchi games played on trees with back-edges

Our goal is to describe an algorithm that solves a Büchi game played on trees with back-edges. By Lemma 5.1.2, it suffices to describe an algorithm that solves reduced Büchi games.

5.2.1 Snares

Let $\mathcal{G} = (V_0, V_1, E^{\mathsf{T}}, E^{\mathsf{B}}, T)$ be a reduced Büchi game. Let u be a leaf in the tree $\mathcal{T} = (V, E^{\mathsf{T}})$. Since $\mathcal{G}$ is reduced, we abuse the notation by writing $E^{\mathsf{B}}(u)$ for the unique node v such that $(u, v) \in E^{\mathsf{B}}$. We will often identify a subset $S \subseteq V$ with the subgraph of $G = (V_0 \cup V_1, E^{\mathsf{T}} \cup E^{\mathsf{B}})$ induced by S. Recall that W_0 denotes the 0-winning region of $\mathcal{G}$. We now give a refined analysis of the set W_0 by introducing the notion of snares. Essentially, we will construct a sequence $S_0 \subseteq S_1 \subseteq S_2 \subseteq \dots$ of subsets of nodes in V that contain all nodes in W_0.

Definition 5.2.1 *For a subset $S \subseteq V$, a* snare strategy in S *is a strategy for Player 0 such that all plays consistent with the strategy starting from a node in S stay in S forever. A* 0-snare *is a subtree S of the tree (V, E^{T}) such that all leaves in S are targets and Player 0 has a snare strategy in S.*

Note that by definition, Player 0 wins the Büchi game $\mathcal{G}$ from any nodes in a 0-snare. On the other hand, there can be winning positions of Player 0 that do not belong to any 0-snares. To capture the entire winning region W_0 of Player 0, we inductively define the notion of i-snares for all $i \in \mathbb{N}$. Let S_0 be the set $\{u \mid u \text{ belongs to a 0-snare in } \mathcal{G}\}$, and let $T_0 = T$. For $i > 0$, define the set

$$T_i = \{x \mid E^{\mathsf{B}}(x) \in S_{i-1}\},$$

where the sets $S_1, S_2, \dots$ are defined inductively as follows.

Definition 5.2.2 *For $i > 0$, an* i-snare *is a subtree S of the tree (V, E^{T}) such that*

- *All leaves of S are in $T \cup T_i$.*
- *From any node $v \in S$, Player 0 has a snare strategy in $S \cup S_{i-1}$.*

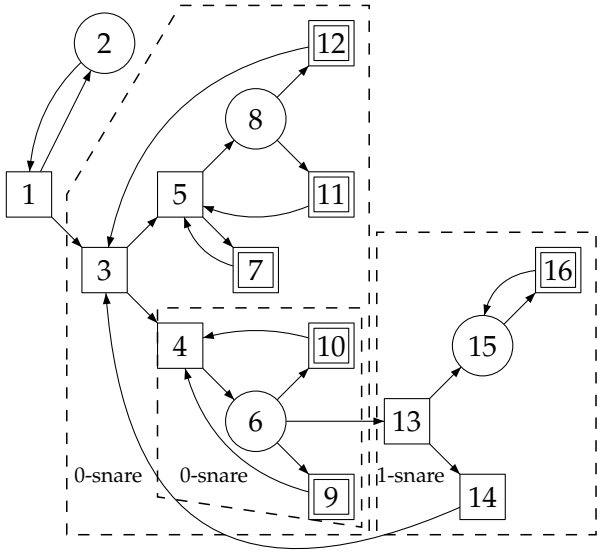
We let S_i denote the set of all nodes that are in an i-snare.

Note that the sequence of nodes $S_0, S_1, \dots$ satisfies that

$$S_0 \subseteq S_1 \subseteq S_2 \subseteq \cdots$$

The *snare rank* of the node $u \in V$ is $\min\{i \mid u \in S_i\}$. The *snare rank* of the Büchi game $\mathcal{G}$ is the maximum snare rank of the nodes in $\mathcal{G}$. From now on we always use r to denote the snare rank of $\mathcal{G}$. Note that the definition requires that $\mathcal{G}$ is a reduced game. When $\mathcal{G}$ is not reduced, the snare rank of $\mathcal{G}$ is defined on the game $\mathsf{Rd}(\mathcal{G})$. A *snare* is an i-snare for some $i \in \{0, \dots, r\}$.

Figure 5.2: Example of a Büchi game with snares shown.



As an example, consider the Büchi game shown in Fig. 5.2. The 0-snares and 1-snare are shown in the figure. Note that the subtree with back-edges rooted at 5 and containing $7, 8, 11, 12$ is *not* a 0-snare. Node 13 is the root of a 1-snare. Note that from 13, Player 1 has two options: 1) move to 15 and lose the game since 16 is a target node *or* 2) move to 14 and lose the game since the play must move to the 0-snare rooted at 3.

Proposition 5.2.1 *The snare rank r of $\mathcal{G}$ is bounded by the height h of the tree (V, E^{T}).*

Proof Let H_r be a snare of rank r with root u_r. It is clear from the definition of an i-snare that there must be a leaf $x \in H_r$ such that there is a back-edge from x to some node $v \in S_{r-1}$. Also note that all nodes in Path$[u_r, x]$ belong to H_r and therefore $v <_{\mathcal{T}} u_r$.

Let u_{r-1} be the root of the snare H_{r-1} which contains v ($u_{r-1} \leq_{\mathcal{T}} v <_{\mathcal{T}} u_r$). Since $u_{r-1} <_{\mathcal{T}} u_r$, we have $\mathsf{lev}(u_{r-1}) < \mathsf{lev}(u_r)$. We can apply a similar argument to the snare H_{r-1} to find a snare H_{r-2} which has a root $u_{r-2} <_{\mathcal{T}} u_{r-1}$ ($\mathsf{lev}(u_{r-2}) < \mathsf{lev}(u_{r-1})$). In this manner we find a sequence of nodes $u_r, u_{r-1}, u_{r-2}, ..$ such that each u_i is the root of an i-snare and for each i we have $\mathsf{lev}(u_{i-1}) < \mathsf{lev}(u_i)$. Since the tree (V, E^{T}) has height h, the value of r is bounded by h. ∎

The following lemmas reduce the problem of solving a reduced Büchi game to computing snares.

Lemma 5.2.2 *If $u_0 \in S_i$ for some $i \in \{0, \ldots, r\}$, then $u_0 \in W_0$.*

Proof We prove the lemma by induction on i. Suppose u_0 belongs to an i-snare S for some $i > 0$. Let f be a snare strategy in $S \cup S_{i-1}$. Then a play starting from u_0 that is consistent with f either stays in S forever or goes to an $(i-1)$-snare. If all plays starting from u_0 consistent with f reach S_{i-1}, then $u_0 \in W_0$ by the inductive assumption.

Suppose $\pi = u_0, u_1, ..$ is a play consistent with f that does not reach S_{i-1}. Since the game $\mathcal{G}$ is reduced, the play π eventually reaches a leaf node. Let u_{j_0} be the first leaf node visited by π. Then $u_{j_0} \in T_i \cup T$. If $E^{\mathsf{B}}(u_{j_0}) \in S_{i-1}$, then $u_{j_0+1} \in S_{i-1}$ which is impossible by assumption. Hence, by definition, u_{j_0} is a target node. Now applying the same argument to the play starting from u_{j_0+1}, we obtain u_{j_1} which is the second leaf node visited by π. Continuing this argument, we obtain a sequence of target nodes $u_{j_0}, u_{j_1}, u_{j_2}, \ldots$ of nodes in π where $j_0 < j_1 < j_2 < \cdots$. Hence π is a winning play for Player 0. This concludes the proof that $u_0 \in W_0$. ∎

Now our goal is to show that every node in W_0 belongs to some snare. We need the following definition:

Definition 5.2.3 *Let f be a winning strategy for Player 0 on a node u. We define the* tree induced by f on u *as a subtree $\mathcal{T}_u^f = (V_u^f, \leq_T)$ of the underlying tree (V, E^{T}) such that the root of $\mathcal{T}_u^f$ is u and for every node $w >_T u$, $w \in V_u^f$ whenever*

- *$w = f(x)$ for some $x \in V_0 \cap V_u^f$, or*
- *$w \in E^{\mathsf{T}}(x)$ for some $x \in V_1 \cap V_u^f$.*

The edge relation $E_{u,f}$ is E^{T} restricted to V_u^f.

Lemma 5.2.3 *If $u \in W_0$ then $u \in S_i$ for some $i \in \{0, \ldots, r\}$.*

Proof Suppose u does not belong to any snare. Assume for the sake of contradiction that $u \in W_0$. Let f be the winning strategy for Player 0 starting from u. Consider the tree $\mathcal{T}_u^f$ induced by f on u. If for all leaves v in $\mathcal{T}_u^f$ we have $E^{\mathsf{B}}(v) \in S_i$ for some $i \in \mathbb{N}$, then by definition $\mathcal{T}_u^f$ is a $(j+1)$-snare where $j = \max\{i \mid v \text{ is a leaf in } \mathcal{T}_u^f \text{ and } E^{\mathsf{B}}(v) \text{ has snare rank } i\}$. This is in contradiction with the assumption. Therefore, let L be the non-empty set containing all leaves v of $\mathcal{T}_u^f$ such that $E^{\mathsf{B}}(v)$ does not belong to any snare. We define a node u_1 as follows.

- If all nodes in L are targets, then there is some $v \in L$ with $E^{\mathsf{B}}(v) <_{\mathcal{T}} u$ as otherwise $\mathcal{T}_u^f$ is a 0-snare. In this case, let $u_1 = E^{\mathsf{B}}(v)$.
- Suppose a node $v \in L$ is not a target. If $E^{\mathsf{B}}(v) \geq_{\mathcal{T}} u$, then the path $u, v, E^{\mathsf{B}}(v), v, E^{\mathsf{B}}(v), \cdots$ defines a play consistent with f but is winning for Player 1. Thus $E^{\mathsf{B}}(v) <_{\mathcal{T}} u$ and we let $u_1 = E^{\mathsf{B}}(v)$.

Note that in both cases, $u_1 <_{\mathcal{T}} u$ and u_1 does not belong to any snare. Also since f is a winning strategy for Player 0 , u_1 is a winning position for Player 0. Applying the same argument as above, we obtain the sequence $u >_{\mathcal{T}} u_1 >_{\mathcal{T}} u_2 >_{\mathcal{T}} \cdots$ such that no node in this sequence is in a snare. The sequence is finite and let u_k be the last node in it. Any play consistent with f starting at u_k stays in the induced tree $\mathcal{T}_{u_k}^f$ forever. Therefore if all leaves in $\mathcal{T}_{u_k}^f$ are targets, u_k belongs to a 0-snare which is impossible. Hence let y be the leaf in $\mathcal{T}_{u_k}^f$ that is not a target. Then the sequence $u_k, y, E^{\mathsf{B}}(y), y, E^{\mathsf{B}}(y), \cdots$ defines a winning play that is consistent with f and is winning for Player 1. This is in contradiction with the fact that f is a winning strategy for Player 0. ∎

Combining Lemma 5.2.2 and Lemma 5.2.3, we have the following:

Theorem 5.2.4 *Given a reduced Büchi game $\mathcal{G}$, we have $W_0 = S_r$.*

5.2.2 Finding snares

In the following, we first describe a method to compute 0-snares in the game $\mathcal{G}$, then generalize this method to compute all i-snares where $i > 0$. Let $\mathcal{T} = (V, E^{\mathsf{T}})$. Recall from Section 2.3 that for any $X \subseteq V$, Player 0 has a strategy to force any play into X from $\mathsf{Reach}_0(X, \mathcal{T})$ by using only tree-edges. For simplicity, we denote $\mathsf{Reach}_0(X, \mathcal{T})$ by $\mathsf{Reach}(X)$. Note that if $v \in S_0$ then $v \in \mathit{Reach}(T)$.

For every node $v \in \mathsf{Reach}(T)$, we define a value $\mathsf{b}_0(v) \in \mathbb{N}$ inductively as follows:

- If v is a leaf, let $\mathsf{b}_0(v) = \mathsf{lev}(E^{\mathsf{B}}(v))$. Notice that $v \in T$.
- If v is an internal node and $v \in V_0$, let $\mathsf{b}_0(v) = \max\{\mathsf{b}_0(u) \mid u \in E^{\mathsf{T}}(v) \cap \mathsf{Reach}(T)\}$.
- If v is an internal node and $v \in V_1$, let $\mathsf{b}_0(v) = \min\{\mathsf{b}_0(u) \mid u \in E^{\mathsf{T}}(v)\}$.

The intuition behind the definition of $b_0(v)$ is that Player 0 would like to stay as close to the leaves as possible, while Player 1 attempts to move the play as close to the root as possible.

For any node $v \in V$, and a strategy f for Player 0, let

$$\mathsf{b}_0(f, v) = \min\{\mathsf{lev}(E^{\mathsf{B}}(w)) \mid w \text{ is a leaf in the tree } \mathcal{T}_v^f\}.$$

In other words, $\mathsf{b}_0(f, v)$ is the largest number k with the following property: the first leaf w that appears in any play starting from v and consistent with f has $\mathsf{lev}(E^{\mathsf{B}}(w)) \geq k$. In other words if Player 0 adopts the strategy f starting from v, then $b_0(f, v)$ is the closest level to the root that Player 1 can guarantee to move to after following exactly one back-edge. Note that when v is itself a leaf, $\mathsf{b}_0(f, v) = \mathsf{lev}(E^{\mathsf{B}}(v))$ for any strategy f. For any $X \subseteq V$ and $v \in \mathsf{Reach}(X)$, let $\mathcal{S}_{\mathsf{Reach}}(X, v)$ denote the set of all 0-strategies that force any play into X from v. The next lemma relates $b_0(v)$ with $b_0(f, v)$ for all $f \in \mathcal{S}_{\mathsf{Reach}}(T, v)$. We prove the lemma by induction on the level of v:

Lemma 5.2.5 *For every $v \in \mathsf{Reach}(T)$, $\mathsf{b}_0(v) = \max\{\mathsf{b}_0(f, v) \mid f \in \mathcal{S}_{\mathsf{Reach}}(T, v)\}$.*

Proof The lemma is clear when v is a leaf in $\mathsf{Reach}(T)$. Now suppose v is an internal node and the lemma holds for all nodes $u \in E^{\mathsf{T}}(v) \cap \mathsf{Reach}(T)$. If $v \in V_0$, we have

$$\begin{aligned}
\mathsf{b}_0(v) &= \max\{\mathsf{b}_0(u) \mid u \in E^{\mathsf{T}}(v) \cap \mathsf{Reach}(T)\} \\
&\overset{\text{(Ind.Hyp.)}}{=} \max\left\{\max\{b_0(f, u) \mid f \in \mathcal{S}_{\mathsf{Reach}}(T, u)\} \mid u \in E^{\mathsf{T}}(v) \cap \mathsf{Reach}(T)\right\} \\
&= \max\left\{\max\{\mathsf{b}_0(f, v) \mid f \in \mathcal{S}_{\mathsf{Reach}}(T, v), f(v) = u\} \mid u \in E^{\mathsf{T}}(v) \cap \mathsf{Reach}(T)\right\} \\
&= \max\{\mathsf{b}_0(f, v) \mid f \in \mathcal{S}_{\mathsf{Reach}}(T, v)\}.
\end{aligned}$$

Suppose $v \in V_1$. Below we need the following equality for any sets $F_1, \ldots, F_m \subseteq \mathbb{N}$:

$$\max\{\min\{x_i \mid 1 \leq i \leq m\} \mid x_1 \in F_1, \ldots, x_m \in F_m\} = \min\{\max\{F_i\} \mid 1 \leq i \leq m\}. \tag{5.1}$$

Let $E^{\mathsf{T}}(v) = \{u_1, \ldots, u_m\}$ and let $F_i = \{\mathsf{b}_0(f, u_i) \mid f \in \mathcal{S}_{\mathsf{Reach}}(T, u_i)\}$ for $i \in \{1, \ldots, m\}$. Note that for any 0-strategy f, $f \in \mathcal{S}_{\mathsf{Reach}}(T, v)$ if and only if the subtree of $\mathcal{T}_v^f$ rooted at u_i, $1 \leq i \leq m$, is of the form $\mathcal{T}_{u_i}^{f_i}$ for some $f_i \in \mathcal{S}_{\mathsf{Reach}}(T, u_i)$. Let $f \in \mathcal{S}_{\mathsf{Reach}}(T, v)$ and $f_1, \ldots, f_m$ be the 0-strategies as described above. By definition,

$$\mathsf{b}_0(f, v) = \min\{\mathsf{b}_0(f_i, u_i) \mid 1 \leq i \leq m\} \tag{5.2}$$

Then we have

$$\begin{aligned}
\mathsf{b}_0(v) &= \min\{\mathsf{b}_0(u_i) \mid 1 \le i \le m\} \\
&\overset{\text{(Ind.Hyp)}}{=} \min\Big\{\max\{\mathsf{b}_0(f_i, u_i) \mid f_i \in \mathcal{S}_{\mathsf{Reach}}(T, u_i)\} \mid 1 \le i \le m\Big\} \\
&= \min\{\max\{F_i\} \mid 1 \le i \le m\} \\
&\overset{\text{by (5.1)}}{=} \max\{\min\{x_i \mid 1 \le i \le m\} \mid x_1 \in F_1, \ldots, x_m \in F_m\} \\
&= \max\{\min\{\mathsf{b}_0(f_i, u_i) \mid 1 \le i \le m\} \mid f_1 \in \mathcal{S}_{\mathsf{Reach}}(T, u_1), \ldots, f_m \in \mathcal{S}_{\mathsf{Reach}}(T, u_m)\} \\
&\overset{\text{by (5.2)}}{=} \max\{\mathsf{b}_0(f, v) \mid f \in \mathcal{S}_{\mathsf{Reach}}(T, v)\}
\end{aligned}$$

∎

For every $u \in \mathsf{Reach}(T)$, let $S_0(u) = \{v \geq_{\mathcal{T}} u \mid \forall w \in \mathrm{Path}[u, v] : \mathsf{b}_0(w) \geq \mathsf{lev}(u)\}$. The following lemma provides a way to check if a node belongs to a 0-snare.

Lemma 5.2.6 *For every $v \in \mathsf{Reach}(T)$, v belongs to a 0-snare if and only if $v \in S_0(u)$ for some $u \leq_{\mathcal{T}} v$ such that $\mathsf{b}_0(u) \geq \mathsf{lev}(u)$.*

Proof Suppose v belongs to a 0-snare S that is rooted at some node u. Let f be the snare strategy in S. Since all leaves of S are targets, for all $w \in \mathrm{Path}[u, v]$, $f \in \mathcal{S}_{\mathsf{Reach}}(T, w)$, and by definition of a snare strategy, all plays starting from w that are consistent with f will stay in S forever. In particular, we have $\mathsf{b}_0(f, u) \geq \mathsf{lev}(u)$. By Lemma 5.2.5, $\mathsf{b}_0(u) \geq \mathsf{b}_0(f, u) \geq \mathsf{lev}(u)$. Furthermore, for all nodes $w \in \mathrm{Path}[u, v]$, $\mathsf{b}_0(w) \geq \mathsf{b}_0(f, w) \geq \mathsf{lev}(u)$. Hence, $v \in S_0(u)$.

Conversely, let $u \in \mathsf{Reach}(T)$ be such that $\mathsf{b}_0(u) \geq \mathsf{lev}(u)$. We prove that the set $S_0(u)$ forms a 0-snare. It is clear that all leaves in $S_0(u)$ are targets. By Lemma 5.2.5, for every node $v \in S_0(u)$, there is a strategy f_v such that any leave w in the tree $\mathcal{T}_v^{f_v}$ is a target and $\mathsf{lev}(E^{\mathsf{B}}(w)) \geq \mathsf{b}_0(u)$. Therefore we define a strategy for Player 0 such that $g(v) = f_v(v)$ for all node $v \in V_0 \cap S_0(u)$. Now let π be a play starting from some $v \in S_0(u)$ and consistent with g. Whenever π reaches a leaf w, we have $\mathsf{lev}(E^{\mathsf{B}}(w)) \geq \mathsf{lev}(u)$ and thus $E^{\mathsf{B}}(w) \in S_0(u)$. Hence any play starting from $S_0(u)$ and consistent with g will stay in $S_0(u)$ forever. This means $S_0(u)$ is a 0-snare. ∎

Lemma 5.2.6 gives us a way to check if a node belongs to a 0-snare. In particular, the following equality holds:

$$S_0 = \bigcup \Big\{S_0(u) \mid u \in \mathsf{Reach}(T), \mathsf{b}_0(u) \geq \mathsf{lev}(u)\Big\}.$$

We now apply our reasoning above to i-snares where $i > 0$. For $i > 0$, recall that $T_i = \{w \mid E^{\mathsf{B}}(w) \in S_{i-1}\}$. Note that a node belongs to an i-snare only if it belongs to $\mathsf{Reach}(T \cup T_i)$. Recall that h is the height of the tree $\mathcal{T}$. We inductively define a function $\mathsf{b}_i : \mathsf{Reach}(T \cup T_i) \to \{0, \ldots, h\}$ in the same way as b_0 with the following difference: If $v \in T_i$, let $\mathsf{b}_i(v) = h$.

For any node $v \in V$ and a strategy f, let $\mathsf{b}_i(f, v) = h$ if all leaves in the tree $\mathcal{T}_u^f$ belong to T_i and let $\mathsf{b}_i(f, v) = \mathsf{b}_0(f, v)$ otherwise. In other words, $\mathsf{b}_i(f, v)$ is the largest number $k \in \{0, \ldots, h\}$ with the following property: the first leaf w that appears in any play starting from v and consistent with f has either $E^{\mathsf{B}}(w) \in S_{i-1}$ or $\mathsf{lev}(E^{\mathsf{B}}(u)) \geq k$. In the same way as the proof of Lemma 5.2.5, we can prove the following lemma:

Lemma 5.2.7 *For every $v \in \mathsf{Reach}(T \cup T_i)$, $\mathsf{b}_i(v) = \max\{\mathsf{b}_i(f, v) \mid f \in \mathcal{S}_{\mathsf{Reach}}(T \cup T_i, v)\}$.*

For every $u \in \mathsf{Reach}(T \cup T_i)$, let $S_i(u) = \{v \geq_{\mathcal{T}} u \mid \forall w \in \mathrm{Path}[u, v] : \mathsf{b}_i(w) \geq \mathsf{lev}(u)\}$. We can prove the next lemma similarly as proving Lemma 5.2.6 with every appearance of Lemma 5.2.5 replaced by Lemma 5.2.7.

Lemma 5.2.8 *For any node $v \in \mathsf{Reach}(T_i \cup T)$, v belongs to an i-snare if and only if $v \in S_i(u)$ for some $u \leq v$ such that $\mathsf{b}_i(u) \geq \mathsf{lev}(u)$.*

Hence, we obtain the following equality for every $i \in \{0, \ldots, r\}$.

$$S_i = \bigcup \Big\{ S_i(u) \mid u \in \mathsf{Reach}(T \cup T_i), \mathsf{b}_i(u) \geq \mathsf{lev}(u) \Big\}. \tag{5.3}$$

5.2.3 An algorithm for solving Büchi games on trees with back-edges

Recall that for a tree $\mathcal{T}$, the *external path length* ℓ is $\sum\{\mathsf{lev}(v) \mid v \text{ is a leaf in } \mathcal{T}\}$. For any game $\mathcal{G}$ played on trees with back-edges (not necessarily reduced), by the *height* and *external path length* of $\mathcal{G}$ we respectively mean the height and external path length of the underlying tree of $\mathcal{G}$.

Theorem 5.2.9 *There exists an algorithm that solves any Büchi game $\mathcal{G}$ played on trees with back-edges in time $O(\min\{r \cdot m, \ell + m\})$ where r is the snare rank, m is the number of edges and ℓ is the external path length of $\mathcal{G}$.*

Proof By Lemma 5.1.2, we first compute in time $O(m)$ the reduced game $\mathrm{Rd}(\mathcal{G})$. The rest of the algorithm works on $\mathrm{Rd}(\mathcal{G})$, which we simply write as $\mathcal{G}$. For $i \geq 0$, assume S_{i-1} has been computed. By Lemma 5.2.8 and (5.3), Algorithm 2 computes the set S_i. By Lemma 5.2.2

Algorithm 2 FindSnare[i]($\mathcal{G}$). (Outline)

1: Compute the set $T_i = \{w \mid E^{\mathsf{B}}(w) \in S_{i-1}\}$.
2: Compute $\mathsf{Reach}(T \cup T_i)$.
3: For all $u \in \mathsf{Reach}(T_i \cup T)$ do:
4: Compute $\mathsf{b}_i(u)$
5: If $\mathsf{b}_i(u) \geq \mathsf{lev}(u)$, compute $S_i(u)$ and add $S_i(u)$ to S_i.

and Lemma 5.2.3, we obtain that $W_0 = S_r$. Hence, to compute the entire winning region W_0, it suffices to run FindSnare[0]($\mathcal{G}$), Findsnare[1]($\mathcal{G}$),..., in order. The algorithm terminates after running FindSnare[$r+1$]($\mathcal{G}$) where $S_r = S_{r+1}$ (and thus r is the snare rank).

In FindSnare[i]($\mathcal{G}$), $i \in \{0, \ldots, r+1\}$, we compute $\mathsf{b}_i(u)$ for all $u \in \mathsf{Reach}(T_i \cup T)$ in order: we only compute $\mathsf{b}_i(u)$ when $\mathsf{b}_i(v)$ for all $v \in E^{\mathsf{T}}(u) \cap \mathsf{Reach}(T_i \cup T)$ have been computed. When $\mathsf{b}_i(u) \geq \mathsf{lev}(u)$, we apply a depth-first search on the subtree rooted at u to compute the set $S_i(u)$. After $S_i(u)$ has been computed, we contract all the nodes in $S_i(u)$ into a meta-node $M_{S_i(u)}$ and redirect edges as follows: any edge (u, v) where $u \in S_i(u)$ and $v \notin S_i(u)$ is substituted by an edge $(M_{S_i(u)}, v)$ and conversely any edge (v, u) where $v \notin S_i(u)$ and $u \in S_i(u)$ is substituted by $(v, M_{S_i(u)})$. Hence, each edge in $\mathcal{G}$ is visited a fixed number of times and the FindSnare[i]($\mathcal{G}$) algorithm takes time $O(m)$.

To further reduce the running time of the algorithm, we maintain a variable $\mathsf{b}(u)$ for every node u throughout the entire algorithm. When FindSnare[i]($\mathcal{G}$) is executed, $\mathsf{b}(u)$ will store the value of $\mathsf{b}_i(u)$. During the first iteration (when FindSnare[0] is performed), $\mathsf{b}(u) = \mathsf{b}_0(u)$ for all $u \in \mathsf{Reach}(T)$ and undefined for all other nodes. In the subsequent iterations, we do the following to compute $\mathsf{b}_i(u)$ for $i > 0$ and $u \in \mathsf{Reach}(T \cup T_i)$:

1. If u is a leaf in T_i, set $\mathsf{b}(u) = h$. If u is a leaf in T, the value of $b(u)$ remains unchanged.

2. Then "propagate" the value of $\mathsf{b}(u)$ to ancestors of u as follows: let v be the parent of u. If $v \in V_1$ and $b(v) > b(u)$, then set $b(v) = b(u)$. If $v \in V_0$ and $b(v) < b(u)$, then set $b(v) = b(u)$. This process continues until we reach a node $w <_{\mathcal{T}} u$ such that $b(w)$ does not need to be updated or w is the root.

Hence, at any iteration of the algorithm, we only change the value of $\mathsf{b}(v)$ when $\mathsf{b}(u)$ is changed for some leaf $u \geq_{\mathcal{T}} v$. Also, for any leaf u, if the value of $\mathsf{b}(u)$ is set to h, it is never changed again. Therefore, the number of times we visit a node $v \in V$ is at most the number of leaves in the subtree rooted at v. This means that the algorithm runs in time $O(\ell + m)$

(since the external path length of $\mathrm{Rd}(\mathcal{G})$ is at most $\ell + m$). By the arguments above, we conclude that the algorithm runs in time $O(\min\{r \cdot m, \ell + m\})$. ∎

5.3 Solving parity games played on trees with back-edges

We now apply Theorem 5.2.9 to obtain an algorithm for solving parity games on trees with back-edges. Recall the definition of parity games from Section 2.3: In a *parity game* $\mathcal{G}$, each node $u \in V$ is associated with a priority $\rho(u) \in \mathbb{N}$ and Player 0 wins $\pi = v_0 v_1 \cdots$ if and only if $\min\{\rho(v) \mid v \in \mathrm{Inf}(\pi)\}$ is even. Also recall that we may assume that $E(u) \neq \emptyset$ for any $u \in V$. The following lemma reduces the problem of solving parity games played on trees with back-edges to solving reduced Büchi games.

Lemma 5.3.1 *Given a parity game $\mathcal{G} = (V_0, V_1, E_1^{\mathsf{T}}, E_1^{\mathsf{B}}, \rho)$ played on trees with back-edges, there is a reduced Büchi game $\mathcal{H} = (U_0, U_1, E_2^{\mathsf{T}}, E_2^{\mathsf{B}}, T)$ such that*

- *$V \subseteq U$ and $|U| \leq |V| + |E_1^{\mathsf{B}}|$.*
- *A node $u \in V$ is winning for Player 0 in $\mathcal{G}$ if and only if u is winning for Player 0 in $\mathcal{H}$.*
- *$\mathcal{H}$ is constructed in time $O(\ell + m)$ where ℓ is the external path length of $\mathcal{G}$.*

Proof To define the sets $U_0, U_1, E_2^{\mathsf{T}}$ and E_2^{B}, we use the same construction as in the proof of Lemma 5.1.2. The target set T in the game $\mathcal{H}$ is defined as

$$T = \{\alpha(u, v) \mid (u, v) \in E_1^{\mathsf{B}}, \min\{\rho(x) \mid x \in \mathrm{Path}[v, u]\} \text{ is even}\}.$$

We prove the following claim.

Claim. A node $u \in V$ is winning for Player 0 in $\mathcal{G}$ if and only if u is winning for Player 0 in $\mathcal{H}$.

Fix $u \in V$. Suppose u is a winning position of Player 0 in $\mathcal{G}$. Let f be the winning strategy for Player 0 at u. By Theorem 2.3.1 we know that f is a memoryless strategy. Define the strategy g in the same way as in the proof of Lemma 5.1.2. Any play π starting from u and consistent with g in $\mathcal{H}$ defines a play π' starting from u and consistent with f in $\mathcal{G}$ such that $\mathrm{Inf}(\pi') = \mathrm{Inf}(\pi) \cap V$. Since f is a winning strategy for Player 0, $\min\{\rho(x) \mid x \in \mathrm{Inf}(\pi')\}$ is even. Let $e = \min\{\rho(x) \mid x \in \mathrm{Inf}(\pi)\}$. There must be a back edge $(x, y) \in E_1^{\mathsf{B}}$ that is visited

infinitely often by π' and $\min\{\rho(z) \mid z \in \text{Path}[y, x]\} = e$. By definition, the node $\alpha(x, y) \in T$ and appears in π infinitely often. Hence π is winning for Player 0.

Conversely, suppose u is winning position of Player 0 in $\mathcal{H}$. Then u belongs to a snare by Theorem 5.2.4. Let i be the snare rank of u and let S be the i-snare containing u. Let f be a snare strategy for Player 0 in $S \cup S_{i-1}$ (recall that S_{i-1} denotes all nodes in $\mathcal{H}$ that are in an $(i-1)$-snare). Define the strategy $g : V_0 \to V$ in the same way as in the proof of Lemma 5.1.2. Let π be a play consistent with g starting from u in $\mathcal{G}$. Our goal is to prove that π is a winning play for Player 0 in $\mathcal{G}$. Suppose for the sake of contradiction that π is winning for Player 1.

Note that π corresponds to a play π' consistent with f such that $\text{Inf}(\pi) = \text{Inf}(\pi') \cap V$. By definition of an i-snare, each leaf in the snare S is either a target or has a back edge that goes to S_{i-1}. Assume π' never visits S_{i-1}. In this case, all leaves visited by π' are targets. Let R be the set of leaves visited by π', then by Lemma 5.1.1, $\text{Inf}(\pi') = \bigcup\{\text{Path}[x, \alpha(y, x)] \mid \alpha(y, x) \in R\}$. Therefore

$$\text{Inf}(\pi) = \text{Inf}(\pi') \cap V = \bigcup\{\text{Path}[x, y] \mid \alpha(y, x) \in R\}.$$

Since $R \subseteq T$, for all $\alpha(y, x) \in R$, $\min\{\rho(z) \mid z \in \text{Path}[x, y]\}$ is even. Hence $\min\{\rho(z) \mid z \in \text{Inf}(\pi)\}$ is also even and π is winning for Player 0. This is in contradiction with the assumption that π is winning for Player 1.

Hence π' (and π) must visit a node $u_1 <_{\mathcal{H}} u$ that belongs to an $(i-1)$-snare. Now apply the same argument on u_1. Continuing this process, we obtain a sequence of nodes $u = u_0 >_{\mathcal{H}} u_1 >_{\mathcal{H}} u_2 >_{\mathcal{H}} \cdots$ such that for all $j \in \mathbb{N}$, u_{j+1} has snare rank strictly smaller than the snare rank of u_j. This is clearly a contradiction.

Hence the claim is proved.

We use the depth-first search (DFS) algorithm on the underlying tree $\mathcal{T}$ of $\mathcal{G}$. Consider a path P in $\mathcal{T}$ from the root to a leaf. We represent the length of P by $|P|$. We use the algorithm of [35] to preprocess P in time $O(|P|)$ such that for any $u, v \in P$ such that $u <_{\mathcal{T}} v$, we may find the value of $\min\{\rho(x) \mid x \in \text{Path}[u, v]\}$ in constant time. Then it is clear that preprocessing every such path in $\mathcal{T}$ takes $O(\ell)$ time (where ℓ is the external path length of $\mathcal{G}$) and subsequently finding $\min\{\rho(x) \mid x \in \text{Path}[u, v]\}$ for any nodes $u <_{\mathcal{T}} v$ takes constant time.

Hence for every back edge $(u, v) \in E_1^{\mathsf{B}}$, $\min\{\rho(x) \mid x \in \text{Path}[v, u]\}$ maybe found in constant time after the above preprocessing has been completed. Therefore an algorithm constructs

the reduced Büchi game $\mathcal{H}$ as follows:

1. Preprocess every path P of $\mathcal{T}$ from the root to a leaf using the algorithm of [35].
2. For each back edge $(u, v) \in E_1^{\mathsf{B}}$, create a new leaf $\alpha(u, v)$ by subdividing (u, v). Set $\alpha(u, v)$ as a target if and only if $\min\{\rho(x) \mid x \in \mathrm{Path}[v, u]\}$ is even (note that this takes constant time now).

This procedure takes time $O(\ell + m)$. ∎

By Lemma 5.3.1, we obtain the following theorem.

Theorem 5.3.2 *Any parity game $\mathcal{G}$ played on trees with back-edges can be solved in time $O(\ell + m)$ where ℓ is the external path length of $\mathcal{G}$ and m is the number of edges in $\mathcal{G}$.*

5.4 Experimental results

Since the worst case performance of our algorithm for Büchi games is the same as that of the classical algorithm, we carried out experiments to compare the actual performance of the two algorithms. The experiments can be broadly divided into two categories:

1. Average case running time comparison: We systematically enumerate all games on trees with back-edges with order $n \in \mathbb{N}$ nodes. For each game, we compare the running times of the classical algorithm and our algorithm.

2. Running time comparison with random sampling: We considered three classes of rooted trees: (a) rooted trees with unbounded out-degree (denoted by **RANUD**) , (b) rooted binary trees (denoted by **RANBT**) and (c) rooted trees where all internal nodes have only a single child node (denoted by **RANDL**) We consider the class **RANDL** since it is the simplest class of rooted trees.

   Whenever we refer to a random Büchi game $\mathcal{G}$ from one of these classes of trees, we mean that the underlying tree of G is a randomly generated tree from that class. Since for a given $n \in \mathbb{N}$, there is a unique tree of order n from **RANDL**, we describe how we generate random samples of order n from the classes **RANUD** and **RANBT**:

   - **RANUD** : We first construct a random free tree T_{free} of order n by first generating a sequence of $(n-2)$ random integers chosen independently and uniformly from

$\{0, 1, \ldots, n-1\}$ and then applying a reverse Prüfer transformation to this sequence [18] . We then randomly select a root from the nodes of T_{free}, hence obtaining a rooted tree T.

- **RANBT**: We construct a random rooted binary tree by the algorithm of [5].

Given a random rooted tree T from **RANUD** or **RANBT**, we generate a random reduced Büchi game $\mathcal{G}$ from T as follows:

- For each leaf v of T, we make a random choice of an ancestor u of v and declare a back-edge from v to u. In this manner we obtain a random tree with back-edges T_b.
- From the nodes of T_b, we randomly select nodes of Player 0 and target nodes to obtain a random Büchi game $\mathcal{G}$ on trees with back-edges. Note that $\mathcal{G}$ is reduced.

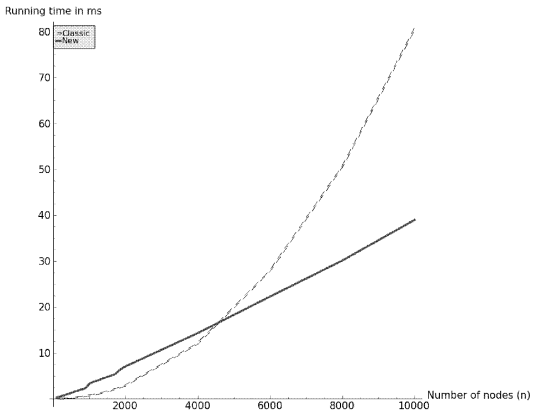
Given a rooted tree T from **RANDL**, for every node v of T we randomly choose an ancestor u of v and add a back-edge from v to u. Then we randomly choose nodes of Player 0 and target nodes as before to obtain a random game $\mathcal{G}$. We then use the procedure described in Lemma 5.1.2 to convert $\mathcal{G}$ to a reduced game.

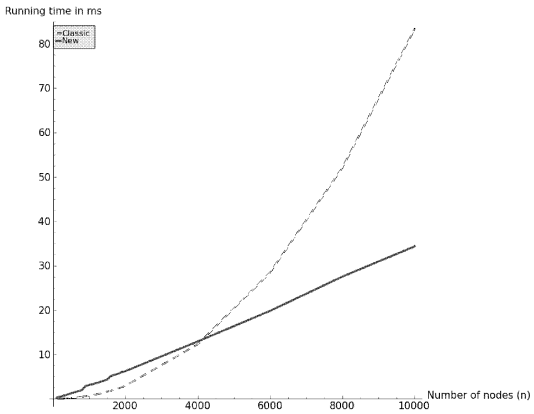
In our experiment, we generated random Büchi games from the three classes described above and then compared the running times of the classical algorithm and our algorithm.

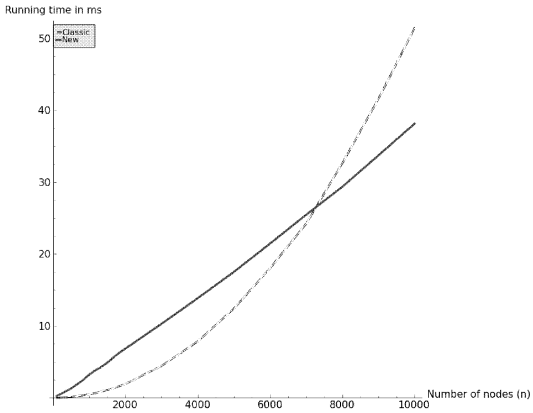
We implemented both the algorithms using the *Sage* mathematics software system [81]. All the experiments were performed on an *Intel Core 2 Duo processor (*2.4 *GHz)* with a *L2 cache* of 4 MB and *RAM* of 3 GB. The source code for the experiments and the implementation of the relevant algorithms is included in appendix I.

Results: *Average case running time comparison* : Figure 5.3 (bottom right) shows the average running time of the classical algorithm and our algorithm for games of size $n \in \{5, 6, \ldots, 13\}$. It is clear that our algorithm performs better than the classical algorithm in all cases. Moreover as n increases, the difference in average case running time between the two algorithms increases. This is particularly evident for $n = 13$, where our algorithm performs an order of magnitude better than the classical algorithm.

The picture becomes even clearer when we analyze the results of the experiments with random sampling. In order to enable a more convenient comparison between the two algorithms we have scaled down the running times of the classical algorithm by a factor of







No. of nodes	Average running time in ms	
	Classical	New
5	0.032809	0.028340
6	0.037290	0.030867
7	0.041399	0.033830
8	0.045444	0.036673
9	0.050211	0.039603
10	0.054622	0.042404
11	0.059377	0.045168
12	0.097281	0.057533
13	0.111520	0.061552

Figure 5.3: The top left graph shows the comparison of algorithms for **RANUD**, the top right graph for **RANBT** and the bottom left for **RANDL** (the dashed lines represent the classical algorithm). The bottom right table compares the average running times of the two algorithms. Note that the running time of the classical algorithm has been scaled down by 10^2 in the graphs.

10^2 for all experiments with random sampling. The sizes of the random games are in the range $n \in \{100, \ldots, 10000\}$.

Random games from **RANUD** : Figure 5.3 (top left) shows the graph comparing the running time of the classical algorithm versus our algorithm for random Büchi games from the class **RANUD**. The sample sizes are as follows: 10^6 random games for $n \in \{100, \ldots, 2000\}$, 10^5 random games for $n \in \{4000, \ldots, 6000\}$, $5 \cdot 10^4$ games for $n = 8000$ and $4 \cdot 10^4$ games for $n = 10000$.

As the graph shows, the classical algorithm exhibits a quadratic growth in running time whereas our algorithm has a linear growth in running time (with respect to n). This

is better than the worst case bound of $O(min\{r \cdot m, l\})$ mentioned in Theorem 5.2.9.

Random games from **RANBT**: Figure 5.3 (top right) shows the graph comparing the running time of the classical algorithm versus our algorithm for random Büchi games from the class **RANBT**. The sample sizes are as follows: 10^5 games for $n \in \{100, \ldots, 2000\}$ and 10^4 games for $n \in \{4000, 10000\}$. Again the classical algorithm shows a quadratic growth in running time as compared to our algorithm which has a linear growth in running time (w.r.t. n) which is better than the worst case bound of $O(min\{r \cdot m, \ell\})$.

Random games from **RANDL**: Figure 5.3 (bottom left) shows the graph comparing the running time of the classical algorithm versus our algorithm for random Büchi games from the class **RANDL**. The sample sizes are identical to the **RANBT** case. Here also, our algorithm shows a linear growth as compared to the quadratic growth shown by the classical algorithm.

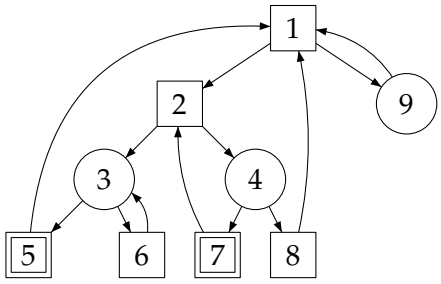
Hence the experiments demonstrate that our algorithm outperforms the classical algorithm in all the three classes. In particular, for $n > 4000$, our algorithm performs two orders of magnitude better than the classical algorithm for all three classes of Büchi games. The experiments clearly show that, in practice, not only our algorithm is much more efficient asymptotically but it also performs better for games with small number of nodes on the set of trees with back-edges.

5.5 Examples where our algorithm performs better than the classical algorithm

We would like to support the positive results of the experiments described in section 5.4 by providing some concrete examples of Büchi games where our algorithm outperforms the classical algorithm. In this section, we present a class of Büchi games on trees with back-edges, $\mathcal{E}$, where our algorithm performs asymptotically better than the classical algorithm.

We first describe the game $\mathcal{G}_0$ with the smallest number of nodes in this class and then use $\mathcal{G}_0$ as a basic unit to construct larger games from $\mathcal{E}$. Consider the game $\mathcal{G}_0$ shown in Fig. 5.4. We have $V_0 = \{3, 4, 5, 6, 7, 8, 9\}$ and $V_1 = \{1, 2\}$ and the target nodes $T = \{5, 7\}$. It is not hard to see that Player 0 has no winning nodes since Player 1 can force any play from the target nodes to node 1 which is clearly a winning node for Player 1.

Figure 5.4: The game $\mathcal{G}_0$.



Let us now analyze the running time of the classical algorithm on $\mathcal{G}_0$. In the first iteration, we have $T_0 = \{5, 7\}$, $R_0 = \mathsf{Reach}_0(T_0, \mathcal{G}) = \{2, 3, 4, 5, 6, 7\}$ and $U_0 = V \setminus R_0 = \{1, 8, 9\}$. Then we set $T_1 = T_0 \setminus \mathsf{Reach}_1(U_0, \mathcal{G}) = \{7\}$ and compute R_1 and U_1. The algorithm terminates at the end of the second iteration since $T_2 = \emptyset$. By contrast, our algorithm needs only one iteration to compute the set W_0.

We now define the games in $\mathcal{E}$ inductively:

1. The game $\mathcal{G}_0$ is as described earlier. Let C denote a copy of the subtree rooted at node 2 in $\mathcal{G}_0$. We denote a node in C by v_c where v is a node in $\mathcal{G}_0$.

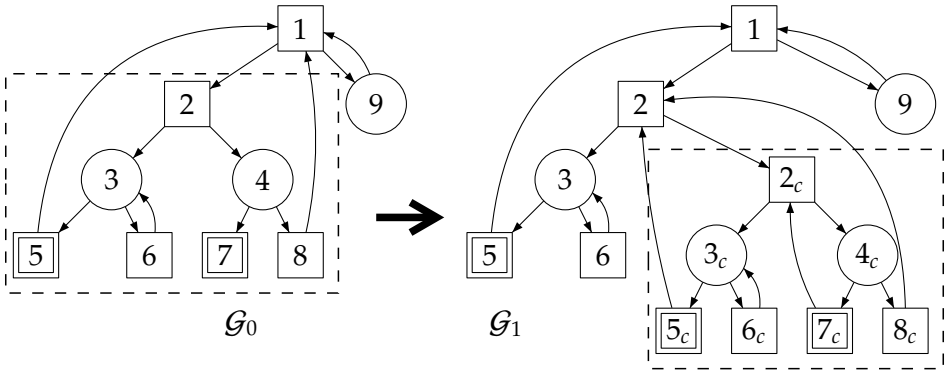
2. For $k > 0$, we construct the game $\mathcal{G}_k$ from $\mathcal{G}_{k-1}$ as follows: Let $x_0, \ldots, x_{k-1}$ be the path in the underlying tree of $\mathcal{G}_{k-1}$ such that x_0 is the left child of the root and for every $i > 0$, x_i is the right child of x_{i-1}. We replace the right subtree rooted at x_{k-1} by C and add back-edges from 5_c and 8_c to x_{k-1}. We declare the nodes 5_c and 7_c to be target nodes in addition to all the target nodes of $\mathcal{G}_{k-1}$.

Figure 5.5 shows the construction of $\mathcal{G}_1$ from $\mathcal{G}_0$. We now give a definition which will help capture the behavior of the classical algorithm on the games in $\mathcal{E}$.

Definition 5.5.1 *Given a Büchi game* $(V_0, V_1, E^{\mathsf{T}}, E^{\mathsf{B}}, T) \in \mathcal{E}$*, for* $x \in T$*, we define* $\chi(x) = \max\{n \in \mathbb{N} \mid$ *Player 0 has a strategy to visit at least n target nodes from* $x\}$.

Hence in the game $\mathcal{G}_0$, we have $\chi(5) = 0$ and $\chi(7) = 1$. Note that a game $\mathcal{G}_k \in \mathcal{E}$ has $k+2$ target nodes and for every $n \in \{0, \ldots, k+2\}$ there is a target node x in $\mathcal{G}_k$ such that $\chi(x) = n$. The following proposition is not hard to prove:

Figure 5.5: The construction of the game $\mathcal{G}_1$ from $\mathcal{G}_0$.



Proposition 5.5.1 *Given a Büchi game $\mathcal{G}_k \in \mathcal{E}$, the classical algorithm terminates after exactly $k+2$ iterations. Our algorithm performs exactly one iteration on $\mathcal{G}_k$.*

Intuitively due to the nature of the construction, given a game $\mathcal{G}_k \in \mathcal{E}$, the classical algorithm needs to perform one iteration for every target node $x \in \mathcal{G}_k$ i.e. for every $i > 0$, $T_i = T_{i-1} \cup \{x\}$ where x is a target node with $\chi(x) = i - 1$. On the other hand, our algorithm performs exactly one iteration just as in the case of $\mathcal{G}_0$.

By the above proposition and the fact that a game $\mathcal{G}_k$ has exactly $k+2$ target nodes, we can see that the classical algorithm has a running time of $O(n \cdot (n+m))$ for the games in $\mathcal{E}$ where n, m are the number of nodes and edges respectively. Since our algorithm performs exactly one iteration for any game $\mathcal{G}_k \in \mathcal{E}$ and $n+m$ increases only linearly with k, we see that our algorithm has a running time of $O(n+m)$. The following proposition expresses this fact:

Proposition 5.5.2 *For the class of Büchi games $\mathcal{E}$, the classical algorithm has a quadratic running time of $O(n \cdot (n+m))$ whereas our algorithm has a linear running time of $O(n+m)$.*

Chapter 6

Open problems and future work

Here we provide a brief overview of some open problems related to each of the previous chapters and outline possible future work.

- In chapter 3, we investigated the state complexity of the union and intersection of finite word and tree languages. We improved the known upper bounds in the case of finite word languages and obtained an upper bound for the case of finite tree language. A natural open problem is to obtain sharp upper bounds for these operations on finite word and tree languages.

- In chapter 4, we generalized the automata model to operate over an arbitrary algebraic structure. It is still a speculation that our model provides a general framework for all other known models of automata. However, the generality of our model comes form the following observations: (1) we can vary the underlying structures and thus investigate models of finite automata over arbitrary structures, (2) in a certain precise sense our machines can simulate Turing machines, (3) many known automata models (e.g Pushdown automata, Petri nets, visibly pushdown automata) can easily be simulated by our model but whether decidability results for these models can be derived from decidability results of our model remains to be seen.

  There are a lot of possiblities for future work on this topic. One natural direction is to obtain more generic results on the emptiness problem. This may require to identify the common properties of the automata over different structures discussed in this thesis, and see how different existing types of automata with external memory (e.g.

bounded reversal counter machines, flat counter automata, pushdown automata) fit into this general framework.

Another interesting direction for future work is to identify structures for which this type of automata enjoy closure under the set operations (even in the nondeterministic case) and hence identify connections of these automata with certain logic over the underlying structures.

A third possible direction is to analyze automata over structures whose domains are not natural numbers. Some interesting examples of such structures include real closed fields, the boolean algebra of finite and co-finite subsets with the subset predicate etc.

In this thesis we have focused our attention on the decidability of emptiness problem for our automata model. However other classical automata-theoretic decidability problems such as the universality problem, the language inclusion problem and the equivalence problem are also topics for future work.

- In chapter 5, we developed an effcient algorithm for Büchi games on trees with back-edges and then applied our analysis to solve parity games on trees with back-edges. A possible direction for future work would be to analyze games with other winning conditions played on trees with back-edges. A particularly interesting case is that of Müller games played on trees with back-edges.

  Another possible direction for future work is to use our analysis of Büchi games on trees with back-edges to develop *fully dynamic algorithms* for such games i.e. the algorithm updates the winning region when some changes are allowed to be made to the game graph. Fully dynamic algorithms have been developed previously for reachability games played on trees [56] and it may be possible to apply a similar approach in our case.

Appendix I

Code listing for chapter 5

```
import os
import sage.all
import time


print_counter = 0

def next(n, L, par):
    q = 0
    p = n - 1
    while L[p] == 1:
        p = p - 1
    if p == 0:
        return
    elif L[p] == 2 and L[p-1] == 1:
        L[p]=1
        par[p] = par[p-1]
        return
    else:
        q = p - par[p]
```

```
        for i in range(p,n):
            L[i] = L[i-q]
            if par[i-q] < p-q:
                par[i] = par[i-q]
            else:
                par[i] = q + par[i-q]
        return
    p = n-1


def gen_trees(n,start, finish,file_name):
    global print_counter
    f= open(file_name,'w')
    f.write('n = '+str(n)+', start ='+str(start)+', finish='+
        str(finish)+'\n')
    print_counter = 0

    if finish == -1:
        finish = sage.rings.infinity.Infinity
    timing = [0.0,0.0]
    enum = 0
    goal = range(n)
    goal[0]=0
    for i in range(1,n):
        goal[i] = 1
    L = range(n)
    par = range(-1,n-1)
    par[0]=0
    e = gen_trees_be(L,par,enum,timing,start,finish, f)
    enum = e[0]
    if e[1]:
```

```
        f.write('games:' + str(finish-start+1)+', Classic:' +
            str(timing[0]*1000) + ', New:'+str(timing[1]*1000)+'
            \n')
        f.close()
        print 'games ='+ str(finish-start+1) + ', n= '+str(n)
        print 'Classical: '+str(timing[0]*1000)
        print 'New Avg: '+str(timing[1]*1000)
        return

    while L!= goal:
        next(n,L,par)
        e = gen_trees_be(L,par,enum,timing,start,finish,f)
        enum = e[0]
        f.flush()
        os.fsync(f.fileno())
        if e[1]:
            f.write('games:' + str(finish-start+1)+', Classic:'
                + str(timing[0]*1000) + ', New:'+str(timing
                [1]*1000)+'\n')
            f.close()
            print 'games ='+ str(finish-start+1) + ', n= '+str(
                n)
            print 'Classical: '+str(timing[0]*1000)
            print 'New Avg: '+str(timing[1]*1000)
            return

    num_games = (enum-start)+1
    f.write('games:' + str(num_games)+', Classic:' + str(timing
        [0]*1000) + ', New:'+str(timing[1]*1000)+'\n')
    f.close()
    print 'games ='+ str(num_games) + ', n= '+str(n)
    print 'Classical Avg: '+str(timing[0]*1000)
```

```
    print 'New Avg: '+str(timing[1]*1000)


def get_leaves(L):
    leaves = []
    for i in range(len(L)-1):
        if(L[i] >= L[i+1]):
            leaves.append(i)
    leaves.append(len(L)-1)
    return leaves

def gen_trees_be(L,par,enum,timing,start,finish,file):
    back_edges = {}
    be_reverse = {}
    be_iso = {}
    leaves = get_leaves(L)
    be_levels = []
    for i in range(len(leaves)):
        be_levels.append(range(L[leaves[i]]))
    for x in sage.misc.mrange.mrange_iter(be_levels):
        iso = False
        c = count_list(x)
        if be_iso.has_key(c):
            candidates = be_iso[c]
            for y in candidates:
                if isomorphic1(L,par, x, y):
                    iso = True
                    break
            if not iso:
                be_iso[c].append(x)
        else:
            be_iso[c] = [x]
```

```
        if not iso:
            be = gen_be(L,par,x)
            back_edges = be[0]
            be_reverse = be[1]
            e = gen_games(L,par,back_edges,be_reverse, leaves,
               enum,timing,start,finish,file)
            enum = e[0]
            if e[1]:
                return enum,e[1]
            back_edges = {}
            be_reverse = {}
    return enum,False

def gen_be(L,par, levels):
    back_edges = {}
    be_reverse = {}
    leaves = get_leaves(L)
    for j in range(len(levels)):
        be_target = trace_up(par, leaves[j],L[leaves[j]]-levels
           [j])
        back_edges[leaves[j]] = be_target
        if be_reverse.has_key(be_target):
            be_reverse[be_target].append(leaves[j])
        else:
            be_reverse[be_target] = [leaves[j]]
    return back_edges, be_reverse

def count_list(list):
    count = range(len(list))
    for i in range(len(list)):
        count[i] = list.count(i)
    return str(count)
```

```
def isomorphic(L, par , l1 , l2 , pr = False):
    be1 = gen_be(L,par,l1)[0]
    be2 = gen_be(L,par,l2)[0]
    g1 = game_to_graph(par,be1)
    g2 = game_to_graph(par,be2)
    if pr:
        print g1.is_isomorphic(g2,certify=True)
    return g1.is_isomorphic(g2)

def isomorphic1(L,par,l1,l2):
    classes = classify_leaves(L,par)
    leaves = get_leaves(L)
    v = classes.values()
    for c in v:
        s1 = []
        s2 = []
        for y in c:
            s1.append(l1[leaves.index(y)])
            s2.append(l2[leaves.index(y)])
        s1.sort()
        s2.sort()
        if s1!=s2:
            return False
    return True

def trace_up(par,leaf, amt):
    curr = leaf
    for i in range(amt):
       curr = par[curr]
    return curr
```

```
def children(par):
    c = {}
    for i in range(1,len(par)):
        if c.has_key(par[i]):
            c[par[i]].add(i)
        else:
            c[par[i]] = set([i])
    return c

def classify_leaves(L,par):
    classes = {}
    isom = tree_iso(L,par)
    leaves = isom[1]
    labels = isom[0]
    for l in leaves:
        type = get_leaf_type(par,l,labels)
        if classes.has_key(type):
            classes[type].append(l)
        else:
            classes[type] = [l]
    return classes

def get_leaf_type(par,leaf,labels):
    curr = leaf
    s = ''
    while par[curr]!=curr:
        s = s + str(labels[curr])
        curr = par[curr]
    s = s + str(labels[curr])
    return s

def tree_iso(L,par):
```

```
    child = children(par)
    label = range(len(L))
    leaves = set(get_leaves(L))
    frontier = leaves
    temp = set([])
    while len(frontier)!=0:
        strings = {}
        for x in frontier:
            if child.has_key(x):
                ch = child[x]
                s = ''
                for y in ch:
                    s = s + str(label[y])
                strings[x] = s
            else:
                strings[x] = '0'
        d = strings.values()
        d.sort()
        e = {}
        c = 0
        for i in d:
            if not e.has_key(i):
                e[i] = c
                c = c+1
        for x in frontier:
            label[x] = e[strings[x]]
            if x!=0:
                temp.add(par[x])
        frontier = temp
        temp = set([])
    return label,leaves
```

```
def gen_games(L,par,back_edges,be_reverse,leaves,enum,timing,
    start,finish, file):
     global print_counter
     lim_reached = False
     t = sage.combinat.subset.Subsets(leaves)
     for targets in t:
         for i in range(2):
             V0 = []
             V1 = []
             for j in range(len(L)):
                 if L[j]%2 == i:
                     V0.append(j)
                 else:
                     V1.append(j)
             game = [L,par,back_edges,be_reverse, leaves, set(V0
                 ),set(V1)]
             enum = enum+1

             if(enum>=start and enum <=finish):
                 print_counter = print_counter+1
                 c = children(par)
                 t1 = time.time()
                 W1 = classic_solve(game, 0, targets)
                 t2 = time.time()

                 timing[0]= timing[0]+t2-t1

                 t1 = time.time()
                 W0 = new_alg(game, 0, targets,c)
                 t2 = time.time()

                 timing[1]= timing[1]+t2-t1
```

```
                if print_counter==20000:
                    file.write('games:' + str(enum)+', Classic:
                        ' + str(timing[0]*1000) + ', New:'+str(
                        timing[1]*1000)+'\n')
                    print_counter = 0

            if enum==finish:
                lim_reached = True
                return enum, lim_reached
    return enum,lim_reached

def classic_solve(game, of_player, targets):
    curr_nodes = set(range(len(game[0])))
    targets = set(targets)
    W = set([])
    winning = set([])
    while True:
        W = avoid_set(game, of_player, targets, curr_nodes)
        winning.update(W)
        curr_nodes.difference_update(W)
        if len(W) == 0:
            break
    return winning

def avoid_set(game, of_player, targets, curr_nodes):
    R = reach(game,of_player, targets, curr_nodes)
    tr = curr_nodes.copy()
    tr.difference_update(R)
    return reach(game, 1-of_player, tr, curr_nodes)


def new_alg(game, of_player, targets, children):
```

```
    curr_targets = targets
    W = set([])
    curr = []
    temp = []
    credit = {}
    while len(curr_targets)!=0:
        t= min_max(game,of_player, curr_targets,curr,  children
           ,credit)
        for x in t[0]:
            if game[3].has_key(x):
                leaves = game[3][x]
                for y in leaves:
                    temp.append(y)
            W.add(x)
        curr = t[1]
        curr_targets = temp
        temp = []
    return W

def min_max(game, of_player,start,curr,children,credit):
    snares = set([])
    temp = set([])
    frontier = set([])
    proc = set([])
    if len(curr) == 0:
        curr = range(len(game[0]))
        for i in range(len(game[0])):
            if i in game[5+of_player]:
                curr[i] = -1
            else:
                curr[i] = len(game[0])+1
        for x in start:
```

```
            curr[x] = game[0][game[2][x]]
            frontier.add(x)
    else:
        for x in start:
            curr[x] = len(game[0])+1 #infinity
            snares.add(x)
            frontier.add(x)

    for x in frontier:
        node = x
        parent = game[1][node]
        while parent != node:
            if parent in game[5+of_player] and curr[node]>curr[
               parent]:
                curr[parent] = curr[node]
                if curr[parent]>=game[0][parent]:
                    snares.add(parent)
                temp = parent
                parent = game[1][parent]
                node = temp
            elif parent in game[6-of_player]:
                if credit.has_key(parent):
                    dict = credit[parent]
                    dict[node] = curr[node]
                else:
                    dict = {}
                    dict[node] = curr[node]
                    credit[parent] = dict
                dict = credit[parent]
                if len(dict) == len(children[parent]):
                    min_val = min(dict.values())
                    if min_val !=curr[parent]:
```

```
                        curr[parent] = min_val
                        if curr[parent]>=game[0][parent]:
                            snares.add(parent)
                        temp = parent
                        parent = game[1][parent]
                        node = temp
                    else:
                        node = parent
                else:
                    node = parent
            else:
                node = parent
    return snares,curr


def reach(game, of_player,targets, curr_nodes):
    c = init_label(game, of_player, curr_nodes)
    reach_set = set([])
    t = targets
    while len(t)!=0:
        temp = tree_reach(game, of_player, t, c, curr_nodes)
        reach_set.update(temp[0])
        t = temp[1]
    return reach_set


def tree_reach(game, of_player, targets, c, curr_nodes):
    reach_set = set([])
    new_tar = set([])
    for t in targets:
        propagate(game, t, c, reach_set, curr_nodes)
    for x in reach_set:
```

```
        if game[3].has_key(x):
            for y in game[3][x]:
                if y not in reach_set:
                    new_tar.add(y)
    return reach_set,new_tar

def init_label(game, of_player, curr_nodes):
    c = range(len(game[0]))
    for i in curr_nodes:
        if i in game[5+of_player]:
            c[i]=1
        elif i in game[6-of_player]:
            count = game[1].count(i)
            if i==0:
                count = count -1
            if count!=0:
                c[i] = count
            else:
                c[i] = 1
    return c

def propagate(game, start, label,reach_set, curr_nodes):
    if start not in curr_nodes:
        return
    anc = start
    while label[anc]!=0:
        label[anc] = label[anc] - 1
        if label[anc] == 0:
            reach_set.add(anc)
            anc = game[1][anc]
            if anc not in curr_nodes:
                return
```

```
        else:
            return

def game_to_graph(par, back_edges):
    g = to_graph(par)
    for be in back_edges.iteritems():
        g.add_edge(be[0],be[1])
    return g

def show_game(game,targets):
    d={'#FF0000':[],'#FFFF00':[], '#00FF00':[]}
    for v in range(len(game[0])):
        if v in targets:
            d['#FF0000'].append(v) #targets are red
        elif v in game [5]:
            d['#FFFF00'].append(v) #P0 is yellow
        else:
            d['#00FF00'].append(v) #P1 is green
    g = game_to_graph(game[1],game[2])
    s = g.plot(vertex_colors = d)
    s.show()

def to_graph(par):
    g = sage.graphs.digraph.DiGraph()
    for i in range(len(par)):
        g.add_vertex(i)
        if i != par[i]:
            g.add_edge(par[i],i)
    return g

def tree_to_game(g,root):
    L = range(g.order())
```

```
    par = range(g.order())
    l = {}
    L[0]  = 0
    l[root]  = 0
    par[0] = 0
    lab = {}
    lab[root] = 0
    label = 1
    dfs = list(g.depth_first_search(root))
    for x in dfs[1:len(dfs)]:
        neighbors = g.neighbors(x)
        for n in neighbors:
            if l.has_key(n):
                lab[x] = label
                l[x] = l[n] + 1
                L[label] = l[n] + 1
                par[label] = lab[n]
                label = label+1
    return L,par

def random_game(n):
    tree = RandomTree(n)
    root = int(round(sage.misc.prandom.random()*(n-1)))
    p = tree_to_game(tree,root)
    leaves = get_leaves(p[0])
    levels = []
    for l in leaves:
        levels.append(int(round(sage.misc.prandom.random()*(p
            [0][1]-1))))
    be = gen_be(p[0],p[1],levels)
    targets = set([])
    for l in leaves:
```

```
        coin = int(round(sage.misc.prandom.random()))
        if coin == 1:
            targets.add(l)
    V0 = set([])
    V1 = set([])
    for l in range(n):
        coin = int(round(sage.misc.prandom.random()))
        if coin == 0:
            V0.add(l)
        else:
            V1.add(l)
    return [p[0],p[1],be[0],be[1],leaves, V0,V1], targets

def check(W0,W1,game):
    N = W1.union(W0)
    return not W1.isdisjoint(W0) or len(N) != len(game[0])

def exp(n,start,finish,file_name):
    gen_trees(n,start,finish,file_name)

def exp_series(n1,n2, file_name):
    for i in range(n1,n2+1):
        gen_trees(i, 1, -1, file_name)

def exp_random(start,end, lim,gap, file_name, print_c):
    f= open(file_name,'w')
    if gap<=0:
        gap = 1
    count = start
    while count<= end:
        print_counter  = 0
        time1=0.0
```

```
        time2=0.0
        for j in range(lim):
            game = random_game(count)
            print_counter = print_counter + 1
            t1 = time.time()
            W0 = new_alg(game[0],0,game[1],children(game[0][1])
                )
            t2=time.time()
            time2=time2+t2-t1
            if print_counter == print_c:
                f.write(str(count)+','+str(j+1)+',New: '+str(
                    time2*1000))
                f.flush()
                os.fsync(f.fileno())

            t1 = time.time()
            W1 = classic_solve(game[0],0,game[1])
            t2 = time.time()
            time1 = time1 + t2-t1

            if print_counter == print_c:
                print_counter = 0
                f.write(',Classical: '+str(time1*1000)+'\n')
                f.flush()
                os.fsync(f.fileno())

        print 'n = '+ str(count)
        print 'Classical = '+str((time1*1000))
        print 'New = '+str((time2*1000))
        print '\n'
        f.write(str(count)+','+str(lim)+','+str(time1*1000)+','
            +str(time2*1000)+'\n')
```

```
        f.flush()
        os.fsync(f.fileno())
        count = count + gap
    f.close()

def RandomTree(n):
    g = sage.graphs.graph.Graph()
    code = [ int(round(sage.misc.prandom.random()*(n-1))) for i
        in xrange(n-2) ]
    avail = [ True for i in xrange(n) ]
    count = [ 0 for i in xrange(n) ]
    for k in xrange(n-2):
        count[code[k]] += 1
    g.add_vertices(range(n))
    idx = 0
    while idx < len(code):
        xlist = [k for k in range(n) if avail[k] and count[k
           ]==0 ]
        if len(xlist)==0: break
        x = xlist[0]
        avail[x] = False
        s = code[idx]
        g.add_edge(x,s)
        count[s] -= 1
        idx += 1
    last_edge = [ v for v in range(n) if avail[v] ]
    g.add_edge(last_edge)
    return g

def exp_random_binary(start,end, lim,gap, file_name, print_c):
    f= open(file_name,'w')
    if gap<=0:
```

```
        gap = 1
    count = start
    while count<= end:
        print_counter = 0
        time1=0.0
        time2=0.0
        for j in range(lim):
            game = random_game_binary(count)
            print_counter = print_counter + 1
            t1 = time.time()
            W0 = new_alg(game[0],0,game[1],children(game[0][1])
                )
            t2=time.time()
            time2=time2+t2-t1
            if print_counter == print_c:
                f.write(str(count)+','+str(j+1)+',New: '+str(
                    time2*1000))
                f.flush()
                os.fsync(f.fileno())

            t1 = time.time()
            W1 = classic_solve(game[0],0,game[1])
            t2 = time.time()
            time1 = time1 + t2-t1

            if print_counter == print_c:
                print_counter = 0
                f.write(',Classical: '+str(time1*1000)+'\n')
                f.flush()
                os.fsync(f.fileno())

        print 'n = '+ str(count)
```

```
        print 'Classical = '+str((time1*1000))
        print 'New = '+str((time2*1000))
        print '\n'
        f.write(str(count)+','+str(lim)+','+str(time1*1000)+','
           +str(time2*1000)+'\n')
        f.flush()
        os.fsync(f.fileno())
        count = count + gap
    f.close()

def number_unbalanced(string):
    stack = []
    for i in range(len(string)):
        b = string[i]
        if b == '(':
            stack.append((b,i))
        else:
            if len(stack)!=0:
                y = stack.pop()
            else:
                return False,len(stack),''
    return True, len(stack),'('*len(stack)

def go_up(par,degree,start):
    curr = start
    while curr>0:
        curr = par[curr]
        if degree[curr]<2:
            return curr
    return curr
```

```
def gen_random_binary(n):
    L = [0]*int(n)
    par = [0]* int(n)
    degree = [0] * int(n)
    level = 0
    curr_node = -1
    max = -1

    k = 2*n
    r  = 0
    temp = ''
    for i in range(2*n):
        x = number_unbalanced(temp)
        r = x[1]
        temp = x[2]
        p = cal_prob(r,k)
        l = sage.misc.prandom.random()
        if r>0 and l<=p:
            temp = temp+')'

            if degree[curr_node] < 2:
                degree[curr_node] = degree[curr_node]+1
            else:
                if curr_node>max:
                    max = curr_node
                curr_node = go_up(par,degree,curr_node)
                degree[curr_node] = degree[curr_node]+1
                level = L[curr_node]
        else:
            temp = temp + '('

            if curr_node>max:
```

```
                max = curr_node
            if (curr_node+1) !=0:
                if degree[curr_node]==2:
                    curr_node = go_up(par,degree,curr_node)
                    level = L[curr_node]

                par[max+1] = curr_node
                degree[curr_node] = degree[curr_node]+1
                L[max+1] = level+1
                level = level+1
            curr_node = max + 1
        k = k-1
    return L,par


def cal_prob(r,k):
    return (r*(k+r+2.0))/(2.0*k*(r+1.0))

def random_game_binary(n):
    p = gen_random_binary(n)
    leaves = get_leaves(p[0])
    levels = []
    for l in leaves:
        levels.append(int(round(sage.misc.prandom.random()*(p
            [0][1]-1))))
    be = gen_be(p[0],p[1],levels)
    targets = set([])
    for l in leaves:
        coin = int(round(sage.misc.prandom.random()))
        if coin == 1:
            targets.add(l)
    V0 = set([])
```

```
    V1 = set([])
    for l in range(n):
        coin = int(round(sage.misc.prandom.random()))
        if coin == 0:
            V0.add(l)
        else:
            V1.add(l)
    return [p[0],p[1],be[0],be[1],leaves, V0,V1], targets

def exp_random_line(start,end, lim,gap, file_name, print_c):
    f= open(file_name,'w')
    if gap<=0:
        gap = 1
    count = start
    while count<= end:
        print_counter = 0
        time1=0.0
        time2=0.0
        for j in range(lim):
            game = func_line(count)
            print_counter = print_counter + 1
            t1 = time.time()
            W0 = new_alg(game[0],0,game[1],children(game[0][1])
                )
            t2=time.time()
            time2=time2+t2-t1
            if print_counter == print_c:
                f.write(str(count)+','+str(j+1)+',New: '+str(
                    time2*1000))
                f.flush()
                os.fsync(f.fileno())
```

```
            t1 = time.time()
            W1 = classic_solve(game[0],0,game[1])
            t2 = time.time()
            time1 = time1 + t2-t1

            if print_counter == print_c:
                print_counter = 0
                f.write(',Classical: '+str(time1*1000)+'\n')
                f.flush()
                os.fsync(f.fileno())

        print 'n = '+ str(count)
        print 'Classical = '+str((time1*1000))
        print 'New = '+str((time2*1000))
        print '\n'
        f.write(str(count)+','+str(lim)+','+str(time1*1000)+','
            +str(time2*1000)+'\n')
        f.flush()
        os.fsync(f.fileno())
        count = count + gap
    f.close()

def func_line(n):
    internal = sage.functions.other.ceil((n-2)/2.0) + int(round
        (sage.misc.prandom.random()*((n-2)-sage.functions.other.
        ceil((n-2)/2.0))))
    return random_game_line(n,internal)

def gen_be_linear(L,par, levels):
    back_edges = {}
    be_reverse = {}
    for j in levels:
```

```
        be_target = trace_up(par, j[0],L[j[0]]-j[1])
        back_edges[j[0]] = be_target
        if be_reverse.has_key(be_target):
            be_reverse[be_target].append(j[0])
        else:
            be_reverse[be_target] = [j[0]]
    return back_edges, be_reverse

def random_game_line(n,internal):
    to_add = n-2-internal
    p = [range(internal+2), [0]+range(internal+1)]

    leaves = []
    levels = []
    targets = set([])

    x =range(1,internal+1)
    x.reverse()
    leaves.append(internal+1)
    levels.append((internal+1,int(round(sage.misc.prandom.
       random()*internal))))
    coin = int(round(sage.misc.prandom.random()))
    if coin==1:
        targets.add(internal+1)

    nodes_remaining = internal
    for i in x:
        prob = to_add/nodes_remaining
        coin = int(round(sage.misc.prandom.random()))
        if to_add>0 and coin <= prob:
            new_leaf = len(p[0])
            level = p[0][i]+1
```

```
            p[0].append(level)
            p[1].append(i)
            levels.append((new_leaf,int(round(sage.misc.prandom
                .random()*(level-1)))))
            leaves.append(new_leaf)

            coin2 = int(round(sage.misc.prandom.random()))
            if coin2 == 1:
                targets.add(new_leaf)
            to_add = to_add-1
        nodes_remaining = nodes_remaining - 1
    be = gen_be_linear(p[0],p[1],levels)
    V0 = set([])
    V1 = set([])
    for l in range(n):
        coin = int(round(sage.misc.prandom.random()))
        if coin == 0:
            V0.add(l)
        else:
            V1.add(l)

    return [p[0],p[1],be[0],be[1],leaves, V0,V1], targets
```

Bibliography

[1] Abadi, M., Lamport, L., Wolper, P., Realizable and unrealizable specifications of reactive systems, In: Proc. 17th ICALP (G. Ausiello et al., eds.), Lecture notes in Computer Science 372 (1989), pp. 1-17, Springer-Verlag, 1989.

[2] Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H., Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems, In: Hybrid Systems, Lecture Notes in Computer Science 736, pp. 209-229, Springer-Verlag, 1993.

[3] Alur, R., Dill, D.L., A theory of timed automata, In: Theoretical Computer Science 126, pp. 183-235, Elsevier Science, 1994.

[4] Alur, R., Cerný, P., Weinstein, S., Algorithmic analysis of array-accessing programs, In: Proceeding of CSL 2009, Lecture Notes in Computer Science 5504, pp. 86-101, Springer, 2009.

[5] Arnold, D.B., and Sleep, M.R., Uniform random generation of balanced parenthesis strings, In: ACM Trans. Program. Lang. Syst. **2**, pp. 122-128, 1980.

[6] Berstel, J., Boasson, L., Carton, O., Fagnot, I., Minimization of Automata, In: Automata: from Mathematics to Applications, European Mathematical Society, 2010.

[7] Berwanger, D., Grädel, E., Entanglement - a measure for the complexity of directed graphs with applications to logic and games. In: Proceedings of LPAR'04, pp. 209-223, 2004.

[8] Berwanger, D., Grädel, E., Lenzi, G., The variable hierarchy of the μ calculus is strict, In: Theory of Computing Systems 40, no. 4, pp. 437-466, 2007.

[9] Blum, L., Shub, M., Smale, S., On a Theory of Computation and Complexity over the Real Numbers: NP-completeness, Recursive Functions and Universal Machines, In: Bulletin of the American Mathematical Society 21(1), pp. 1-46, 1989.

[10] Bojanczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C., Two-Variable Logic on Words with Data, In: Proceedings of LICS 2006, pp. 7-16, IEEE Computer Society, 2006.

[11] Bojanczyk, M., David, C., M., Muscholl, Schwentick, T., Segoufin, L., Two-variable logic on data trees and XML reasoning, In: Proceedings of PODS 2006, pp. 10–19, ACM, 2006.

[12] Bournez, O., Cucker, F., Jacobé de Naurois, P., Marion, J.-Y., Computability over an Arbitrary Structure: Sequential and Parallel Polynomial Time, In: Proc. FOSSACS03/ETAPS03, Lecture Notes in Computer Science 2620, pp. 185-199, 2003.

[13] Bozga, M., Iosif, R., Lakhnech, Y., Flat parametric counter automata, In: Proceedings of ICALP '06, Lecture Notes in Computer Science 4052, pp. 577-588, 2006.

[14] Bruyére, V., Hansel, G., Michaux, C., Villemairez, R., Logic and p-Recognizable Sets of Integers, In: Bull. Belg. Math. Soc 1, pp. 191–238, 1994.

[15] Büchi, J.R., Weak second-order arithmetic and finite automata, In: Z. Math. Logik Grundl. Math. 6, pp. 66-92, 1960.

[16] Chatterjee, K., Henzinger, T., Piterman, N., Algorithms for Büchi games. In: Proc. of the 3rd Workshop of Games in Design and Verification (GDV'06), 2006.

[17] Chatterjee, K., Jurdziński, M., Henzinger, T., Simple stochastic parity games. In: Proc. of CSL'03. LNCS 2803:100-113. Springer, 2003.

[18] Cho, M., Kim, D., Seo S., and Shin, H., Colored Prüfer Codes for $k-$Edge Colored Trees, In: Electron. J. Combin. **11** no. 1, #N10, 2004.

[19] Church, A., Logic, arithmetic and automata, In: Proc. of the Intl. Cong. of Mathematicians (Stockholm, Sweden), 1962.

[20] Clarke, E.M., Emerson, E.A., Design and syntheis of synchonization skeletons using branching time temporal logic, In: Lecture Notes in Computer Science 131, pp. 52-71, Springer, 1981.

[21] Clarke, E.M., Emerson, E.A., Characterizing correctness properties of parallel programs as fixpoints, In: Proc. 7th ICALP, Lecture Notes in Computer Science 85, Springer-Verlag, 1980.

[22] Clarke, E., Lu, Y., Veith, H., Jha, S., Tree-Like counterexamples in model checking. In: Proc. of LICS'02, pp.19-29, IEEE Computer Society, 2002.

[23] Comon, H. et al., Tree Automata Techniques and Applications, available on `http://www.grappa.univ-lille3.fr/tata`, 2007.

[24] Comon, S., Jurski, Y., Multiple counters automata, safety analysis and Presburger arithmetic. In: Proceedings of CAV '98, Lecture Notes in Computer Science 1427, pp. 268–279, Springer, 1998.

[25] Corless, R, Gonnet, G., Hare, D., Jeffrey, D., Knuth, D., On the Lambert W function, In: Advances in Computational Mathematics 5, pp. 329–359, Springer, 1996.

[26] Câmpeanu, C., Culik, K., Salomaa, K., Yu, S., State Complexity of Basic Operations on Finite Languages, In: Lecture Notes in Computer Science 2214, pp. 148-157, Springer, 2001.

[27] Dacuik, J., Watson, B.W., Watson, R.E., Incremental Construction of Minimal Acyclic Finite State Automata and Transducers, In: Proceedings of Finite State Methods in Natural Language Processing (FSMNLP1998), pp. 48-56, 1998.

[28] Dziembowski, S., Jurdziński, M., Walukiewicz, I., How much memory is needed to win infinite games?, In: Proc. LICS '97, pp. 99-110, 1997.

[29] Elgot, C.C., Decision problems of finite automata design and related arithmetics, In: Trans. Amer. Math. Soc. 98, pp. 21-52, 1961.

[30] Emerson, E.A., Model checking and the μ-calculus, In: Descriptive Complexity and finite models, pp. 185-214 , American Mathematical Society, 1996.

[31] Emerson, E.A., Jutla, C.S., Tree automata, μ-calculus and determinacy, In: Proc. 32nd FOCS, pp. 368-377, IEEE Computer Society Press, 1991.

[32] Emerson, E.A., Jutla, C.S., Sistla, A.P., On model-checking for fragments of the μ-calculus, In: Proc. 5th CAV, Lecture Notes in Computer Science 697, pp. 385-396, 1993.

[33] Ershov, Y., Goncharov, S., Marek, V., Nerode, A., Remmel, J., Handbook of Recursive Mathematics: Recursive Model Theory, Studies in Logic and the Foundations of Mathematics. North Holland, 1998.

[34] Figueira, D., Reasoning on words and trees with data, PhD thesis, ENS Cachan, France, 2010.

[35] Fischer, J., Heun, V., A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array, In: LNCS 4614, pp. 459-470, Springer-Verlag, 2007.

[36] Gandhi, A., Ke, N.R., Khoussainov, B., State complexity of determinization and complementation for finite automata, In: Proceedings of the 17th Computing: the Australasian Theory Symposium (CATS2011), CRPIT 119, pp. 105-110, 2011.

[37] Gandhi, A., Khoussainov, B., Liu, J., Efficient Algorithms for Games Played on Trees with Back-edges, In: Fundam. Inform. 111(4), pp. 391-412, 2011.

[38] Gandhi, A., Khoussainov, B., Liu, J., Finite Automata over Structures (Extended Abstract), In: Procs. of TAMC 2012, LNCS 7287, pp. 373-384, 2012.

[39] Gandhi, A., Khoussainov, B., Liu, J., On State Complexity of finite word and tree languages, In: Procs. of DLT 2012, LNCS 7410, pp. 392-403, 2012.

[40] Gécseg, F., Steinby, M., Tree languages, In: Handbook of Formal Languages, Vol. 3: Beyond Words (eds. G. Rozenberg and A. Salomaa), pp. 1-68, Springer, 1997.

[41] Grädel, E., Thomas, W., Wilke, T. (Eds.): Automata, Logics, and Infinite Games: A Guide to Current Research. Lecture Note in Computer Science 2500. Springer, 2002.

[42] Gramlich, G., Schnitger, G., Minimizing NFA's and regular expressions, In: J. Comput. Syst. Sci. 73, pp. 908-923, Academic Press Inc., 2007.

[43] Gurevich, Y., Harrington, L., Trees, Automata and Games, In: Proc. of STOCS '82, pp. 60-65, 1982.

[44] Han, Y., Salomaa, K.: State complexity of union and intersection of finite languages. International Journal of Foundations of Computer Science, 19(3): 581–595, World Scientific, 2008.

[45] Holzer, M., Kutrib, M., State complexity of basic operations on nondeterministic finite automata, In: Lecture Notes in Computer Science 2608, pp. 148-157, Springer, 2003.

[46] Holzer, M., Kutrib, M., Descriptional and computational complexity of finite automata - A survey, In: Information and Computation 209, pp. 456-470, 2011.

[47] Hopcroft, J., Motwani, R., Ullman, J., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 2006.

[48] Ibarra, O., Reversal-bounded multicounter machines and their decision problems, In: J. ACM 25(1), pp. 116-133, 1978.

[49] Ishihara, H., Khousainov, B., Rubin, S., Some Results on Automatic Structures, In: Proceedings of LICS 2002, pp. 253-, IEEE Computer Society, 2002.

[50] Jirásek, J., Jirásková, G., Szabari, A., State complexity of concatenation and complementation of regular languages, In: Lecture Notes in Computer Science 3317, pp. 178-189, Springer, 2005.

[51] Jirásek, J., Jirásková, G., Szabari, A., Deterministic blow-ups of minimal nondeterministic finite automata over a fixed alphabet, In: Proc. 11th DLT, Lecture Notes in Computer Science 4588, pp. 254–265, Springer, 2007.

[52] Jirásková, G., On the state complexity of complements, stars, and reversals of regular languages, In: Proc. 12th DLT, Lecture Notes in Computer Science 5257, pp. 431-442, Springer, 2008.

[53] Jurdziński, M., Deciding the winner in parity games is in UP ∩ co-UP, Information Processing Letters 68 (3), pp. 119-124, 1998.

[54] Kaminsky, M., Francez, N., Finite memory automata, In: Theor. Comp. Sci. 134(2), pp. 329-363, 1994.

[55] Khoussainov, B., Proof of memoryless determinacy of parity games, Personal communication, 2010.

[56] Khoussainov, B., Liu, J., Khaliq, I., A Dynamic Algorithm for Reachability Games Played on Trees, In: Proc. of MFCS 2009, Lecture Notes in Computer Science 5734, pp. 477-488, Springer-Verlag, 2009.

[57] Kozen, D., Results on the propositional μ-calculus, In: Theoretical Computer Science 27, pp. 333-354, 1983.

[58] Lye, K.-W., Wing, J.L., Game strategies in network security, In: Int. J. Inf. Secur. 4, pp. 71-86, 2005.

[59] Lynch, N.A., Tuttle, M.R., Hierarchical correctness proofs for distributed algorithms, In: Proc. 6th ACM SIGACT-SIGOPS Symp. on Princ. of Dist. Comp., pp. 137-151, 1987.

[60] Leroux, J., Sutre, G., Flat counter automata almost everywhere!, In: Proceedings of ATVA '05, Lecture Notes in Computer Science 3707, pp. 489-503, Springer, 2005.

[61] Leroux, J.: The general vector addition system reachability problem by presburger inductive invariants, In: Procedings of LICS 2009: 4–13, IEEE Computer Society, 2009.

[62] Martin, D.: Borel determinacy, Ann. Math. 102, No. 2(Sep., 1975), pp. 363-371.

[63] Matiyasevich, Y., Hilbert's Tenth Problem, MIT Press, Cambridge, Massachusetts, 1993.

[64] McNaughton, R., Inifinte games played on finite graphs, In: Annals of Pure and Applied Logic 65, pp. 149-184, 1993.

[65] Minsky, M., Recursive unsolvability of Post's problem of "Tag" and other topics in theory of Turing machines, In: Annals of Math. 74(3), 1961.

[66] Moore, F.R., On the Bounds for State-Set Size in the Proofs of Equivalence Between Deterministic, Nondeterministic, and Two-Way Finite Automata, In: IEEE Trans. Comput. 20, pp. 1211-1214, IEEE Computer Society, 1971.

[67] Mohri, M., On some applications of finite-state automata theory to natural language processing, Natural Language Engineering 2, Cambridge University Press, 1996.

[68] Mostowski, A., Games with forbidden positions, Tech. Report 78, Instytut Matematyki, Uniwersytet Gdański, Poland, 1991.

[69] Nerode, A., Yakhnis, A., Yakhnis, V., Concurrent programs as strategies in games, In: Logic from Computer Science (Y. Moschovakis, ed.), Springer, 1992.

[70] Nerode, A., Remmel, J., Yakhnis, A., McNaughton games and extracting strategies for concurrent programs, In: Annals of Pure and Applied Logic 78, pp. 203-242, 1996.

[71] Neven, F., Schwentick, T., Vianu, V., Finite state machines for strings over infinite alphabets, In: ACM Tran. Comput. Logic 15(3), pp. 403-435, 2004.

[72] Obdržálek, J., Algorithmic Analysis of Parity Games, PhD thesis, Univ. of Edinburgh, 2006.

[73] Piao, X., Salomaa, K.: Operational state complexity of Deterministic Unranked Tree Automata, In: Proc. of the Twelfth Annual Workshop on Descriptional Complexity of Formal Systems (DCFS 2010), EPTCS 31: 149–158, 2010.

[74] Piao, X., Salomaa, K.: Transformations between different models of unranked bottom-up tree automata, In: Fundamenta Informaticae 109(4), pp. 405–424, IOS Press, 2011.

[75] Pnueli, A., The temporal logic of programs, In: Proc. 18th FOCS, pp. 46-57, 1977.

[76] Pnueli, A., Rosner, R., On the synthesis of a reactive module, In: Proc. 16th ACM Symp. on Principles of Progr. Lang., pp. 179-190, Austin, 1989.

[77] Point, F., On Decidable Extensions of Presburger Arithmetic: From A. Bertrand Numeration Systems to Pisot Numbers, In: J. Symb. Log. 65(3), pp. 1347-1374, 2000.

[78] Rabin, M.O., Scott, D., Finite automata and their decision problems, In: IBM J. Res. Develop. 3, pp. 114-125, 1959.

[79] Rogers, H., Theory of recursive and effective computability, McGraw Hill, 1968.

[80] Safra, S., On the complexity of ω-automata. In: Proc. 29th IEEE Symp. on Foundations of Computer Science (FOCS1988), pages 319–327, White Plains, 1988.

[81] Sage open source mathematical software, `http://www.sagemath.org/`.

[82] Schnitger, G., Regular expressions and NFA's without ϵ-transitions, In: Proc. 23rd STACS, Lecture Notes in Computer Science 3884, pp. 432-443, 2006.

[83] Segoufin, L., Automata and logics for words and trees over an infinite alphabet, In: Proc. of CSL 2006, Lecture Notes in Computer Science 4207, pp. 41-57, Springer, 2006.

[84] Segoufin, L., Torunczyk, S., Automata based verification over linearly ordered data domains, In: Proceedings of STACS 2011, pp. 81-92, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[85] Soare, R., Recursively enumerable sets and degrees, Perspectives in mathematical logic, Springer-Verlag, 1987.

[86] Stirling, C., Local model checking games, In: Proc. Concur'95, Lecture Notes in Computer Science 962, pp. 1-11, 1995.

[87] Tan, T., Graph reachability and pebble automata over infinite alphabets, In: Proc. of LICS 2009, pp. 157-166, IEEE Computer Society, 2009.

[88] Thomas, W., On the synthesis of strategies in infinite games, In: Proc. 12th STACS, Lecture Notes in Computer Science 900 (1995), pp. 1-13, Springer, 1995.

[89] Thomas, W., Languages, automata and logic, In: Handbook of Formal languages, vol. 3, Springer-Verlag, 1997.

[90] Walukiewicz, I., Pushdown processes: Games and model checking, In: Information and Computation 164 (2), pp. 234-263, 2001.

[91] Yu, S.: State complexity of regular languages, In: Journal of Automata, Language and Combinatorics 6(2), pp. 221–234, 2001.