BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

A LIBRARY FOR COMPUTING SIMULATION RELATIONS OVER BÜCHI AUTOMATA

KNIHOVNA PRO VÝPOČET RELACÍ SIMULACE NA BÜCHIHO AUTOMATECH

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DANIEL ODVÁRKA**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDŘEJ LENGÁL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

Student: **Odvárka Daniel**
Programme: Information Technology
Title: **A Library for Computing Simulation Relations over Büchi Automata**
Category: Theoretical Computer Science

Assignment:

1. Study the theory of Büchi automata and operations over them.
2. Study the theory of simulations over Büchi automata, in particular *direct simulation*, *delayed simulation*, *fair simulation*, *lookahead simulation*, *multi-pebble simulation*, and others.
3. Design a library in C/C++ providing interface for computing simulations selected together with the supervisor.
4. Implement the designed library including a command line interface.
5. Experimentally evaluate the speed of the developed library and the properties of the computed relations. Discuss the achieved results.

Recommended literature:

- E. Grädel, W. Thomas, T. Wilke. Automata Logics, and Infinite Games. Springer, 2002. ISBN 978-3-540-36387-3
- K. Etessami, T. Wilke T., R.A. Schuller. Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata. In Proc. of ICALP 2001. LNCS 2076. Springer, 2001.
- R. Mayr, L. Clemente. Advanced automata minimization. In Proc. of POPL'13. ACM, 2013.

Requirements for the first semester:

- Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Lengál Ondřej, Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 11, 2022
Approval date: November 3, 2021

Abstract

This thesis presents the topic of simulation relations over Büchi automata and the use case of various simulations. The simulation relations are important for reducing a state space of an automaton, or checking the under approximation of language inclusion. There are also parity games, which are important for computing some of the simulation types we introduce. We will go over multiple algorithms that compute these simulation relations. The mentioned algorithms are implemented in programming language `C++` and are compared to a tool named `RABIT`. Our implementation is better only for smaller sized automata.

Abstrakt

Tato práce popisuje téma simulací relací nad Büchiho automaty a využití těchto relací simulací. Relace simulací jsou důležité pro snižování stavového prostoru automatů, nebo dále pro kontrolu pod aproximace jazykové inkluze. Dále popisujeme téma paritních her, které je úzce spojeno s výpočtem relací simulací pro některé typy simulací. Podívame se také na různé algoritmy pro řešení relací simulací. Zmíněné algoritmy byly implementovány v jazyce `C++` a implementace byla porovnána s nástrojem `RABIT`. Z experimentů je vidno, že je naše implementace lepší pouze pro menší automaty.

Keywords

finite automata, Büchi automata, simulation relations, parity games, reduction of an automaton

Klíčová slova

konečné automaty, Büchiho automaty, relace simulací, paritní hry, redukce automatu

Reference

ODVÁRKA, Daniel. *A Library for Computing Simulation Relations over Büchi Automata.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

Rozšířený abstrakt

Büchiho automat (BA) je koncept rozšiřující konečné automaty o nekonečné vstupy (slova nekonečné délky). Büchiho automat je zařazen do skupiny tzv. ω-automatů. BA akceptuje slovo tehdy, pokud existuje běh na automatu, který navštíví alespoň jeden z konečných stavů nekonečně mnohokrát. Práce se zabývá relacemi simulací, které běží právě nad BA. Těchto simulací je hned několik druhů, my se budeme zabývat hlavně dopřednými simlucemi (férová, přímá, opožděná). Okrajově ovšem také zmíníme kamínkovou simulaci a význam zpětných simulací. Simulace se nejčastěji používají pro redukci statového prostoru automatu, nebo také pro kontrolu jazykové inkluze. V naší práci se budeme zabývat pouze redukcí stavového prostoru. Zmíníme základní techniky jako kvocientování (quotienting) a prořezávání (pruning). Pro výpočet férové a opožděné simulace uvedené algoritmy převádí automat na graf paritní hry, který je poté zpracován a daná simulační relace vypočtena z tohoto grafu. Pro přímou simulaci tento krok není nutný, je ovšem také možné tuto možnost využít a relaci přímé simulace vypočítat z grafu paritní hry. Tato možnost se ovšem moc nevyužívá, jelikož máme efektivnější algoritmy. Paritní hra je orientovaný graf, kde každý uzel je obarven prioritou (v našem případě je priorita $p = \{0, 1, 2\}$). Jedná se o hru dvou hráčů, kteří se po každém tahu vystřídají. Hra trvá, dokud hra buď neskončí (jeden z hráčů se nemůže hnout), nebo se vytvoří nekonečně dlouhá cesta, kterou nazýváme jako tah. Vítěz je ten, čí oponent se nemohl hnout dál. Pokud je hra nekonečně dlouhá, tak je vítěz určen pomocí nejvyšší priority uzlu, přes který se šlo nekonečně mnohokrát. Pokud je tato priorita sudá, tak vyhrává hráč Nula, naopak hráč Jedna.

Hlavním úkolem práce bylo implementovat knihovnu v jazyce `C++`, která bude právě tyto výpočty konat pomocí algoritmů, které jsou v prácí uvedené. Celý program má rozhraní v příkazové řádce, kterou má uživatel možnost využít. Náš program je schopný vypočítat přímou, férovou a opožděnou relaci simulace. Konečnou relaci je možné vypsat do příkazové řádky, a nebo vytvořit `dot` soubor, který zkonvertujeme na obrázek. Zde je poté vizualně vidět, mezi kterými stavy simulace platí. V průběhu implementace jsme narazili na různé komplikace, kvůli kterým jsme museli do algoritmů zasahovat. Naše implementace jsou tedy vždy trochu modifikované, vše je podrobně popsané v práci. Na závěr jsme naši implementaci testovali oproti nástroji RABIT, který má všechny tyto simulace naimplementované. Náš nástroj nedosahuje rychlosti RABITu pro automaty s vyšším počtem stavů a přechodů, avšak pro ty menší si počíná náš nástroj lépe. Experimenty porovnávají jednotlivé simulace vůči sobě, či jednotlivé implementace vůči jiným. Pro budoucí práci je rozhodně možné nástroj optimalizovat a přidat možnost redukovat statový prostor vstupního automatu.

A Library for Computing Simulation Relations over Büchi Automata

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ondřeje Lengála. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Daniel Odvárka
May 6, 2022

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Lengál, Ph. D. for all his time, guidance and thorough feedback regarding this thesis.

Contents

Chapter 1

Introduction

An automaton is a basic tool that sets land for formal verification, model checking and to a theoretical informatics field as a whole. This thesis is focused on a special area of automata - Büchi automata. These automata have an infinite input of symbols. These so called ω-automata are different from the classic finite automata in a way, that their run has to go infinitely many times over some of the states of an ω-automaton to be accepted. There are many various accepting condition for ω-automata, such as Büchi, Strett etc. In this thesis, we will be going over a very important part, that is closely related to the topic of ω-automata and that are simulation relations. A simulation relation can be widely useful. We can use it to reduce a state space of an automaton, under approximate language inclusion of an automaton. We will go over the use case of a reduction of an automaton and some techniques, that help us achieve, preferably, without changing the automaton's language, i.e. accepting words are not changed. There are various simulation types, where not every is suitable for the reduction of the state space of an automaton. We will talk about what these different simulations are good for, e.g. pruning, quotienting. These simulation can be seen as a game between two players that are trying to beat each other. We call them Spoiler and Duplicator. In order to understand this, we will need to mention, what a parity game graph is, what a run over the game graph is etc. We will also mention a way to construct this game arena from an ω-automaton.
There are multiple algorithm for solving the simulation for each of the simulation types, the current ones are mentioned in the upcoming chapters. The aim of this thesis is to implement these algorithms as efficiently as possible using a programming language `C++` with a proper command line interface to operate the program easily. The implementation is described in chapter 6.

Chapter 2

Preliminaries

As the whole thesis is revolving around formal verification, we should define the concept. Formal verification uses an abstract mathematical model of the system or software and formally proves either its correctness or inaccuracy. Nevertheless, it is very computationally intensive. Formal verification [3] uses various methods, such as Theorem Proving, Static Analysis, or Model Checking.

We will start the thesis by defining some basic concepts, which will accompany us during the whole text. *Atomic proposition* a is a named boolean value, which represents a simple value. The value of a can be either *True* or *False*. AP is a finite set of atomic propositions, e.g. $\{a, b\}$. The *alphabet* is then denoted as $\Sigma = 2^{AP}$, i.e. $\{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. An infinite word, i.e. ω-word is an infinite sequence of symbols of the alphabet Σ, $w = w_1 w_2 w_3 \ldots \in \Sigma^\omega$.

Definition 1 *An ω-automaton is a quintuple $\langle Q, \Sigma, \delta, q_i, Acc \rangle$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the state transition function, q_i is the initial state, Acc is the acceptance condition. In case of deterministic ω-automaton, a transition function $\delta : Q \times \Sigma \to Q$ is used.*

The acceptance condition can be set as a set of states, as a set of state-sets or as a function from the set of states to a finite set of natural numbers.

A Büchi automaton A is an ω-automaton, a variant of a finite automaton, which takes in an ω-word (infinite). As the word is infinite, the automaton A would run for an infinite amount of time. Automaton A accepts a run that happens to occur an infinite amount of times over a state $q \in F$. The accepting runs of an ω-automata have to check the entire input word, not just a finite prefix. This implies that acceptance criteria are needed to check infinite runs acceptance. Büchi acceptance helps us to determine whether a run was successful or not. A word $\alpha \in \Sigma^\omega$ is accepted by a Büchi automaton A iff there is a run ρ of A that satisfies the following condition [6]:

$$Inf(\rho) \cap F \neq \emptyset$$

i.e at least one of the states $s \in F$ of the automaton A has to be visited infinitely often during the run ρ.

Definition 2 *A nondeterministic Büchi automaton NBA $A = \langle \Sigma, Q, q_i, \Delta, F \rangle$ has a finite alphabet Σ, a finite set of states Q, an initial states $q_i \in Q$, a finite set of transitions $\Delta \subseteq Q \times \Sigma \times Q$ and a finite set of accepting states $F \subseteq Q$.*

Definition 3 *A run of Büchi automaton A is a sequence $\pi = q_0 a_0 q_1 a_1 q_2 a_2 \ldots$ of states and letters such that for every i, $(q_i, a_i, q_{i+1}) \in \Delta$.*

In this thesis, we will be using this notation $p \xrightarrow{a} q$ to denote $(p, a, q) \in \delta$.

Example 1 *Consider the NBA with the alphabet $\Sigma = \{a, b\}$ in Figure 2.1.*

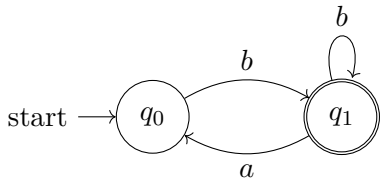

Figure 2.1: Automaton A

In order for a run ρ to be accepted by A, ρ has to visit state q_1 an infinite amount of times. For example, we can have an accepting word ρ described by an ω-regular expression $(ba)^\omega$, so the run is $\pi = q_0, q_1, q_0 \ldots$. The condition of state $q_1 \in F$ being visited infinitely often is fulfilled. Therefore the run ρ is accepted by the automaton A.

In order to compute a x-simulation relation over Büchi automaton we need to know what a preorder is. Preorders are most often denoted as $\sqsubseteq$. The strict version of a preorder is denoted as $\sqsubset$, i.e. $x \sqsubset y$ if $x \sqsubseteq y \wedge y \not\sqsubseteq x$.

Definition 4 *Preorder $\sqsubseteq$ is a binary relation that is reflexive and transitive. Reflexivity tells us that for each $x, x \sqsubseteq x$ holds. Transitivity is that $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$.*

Simulation relations are binary relations on the states of an automaton A. A simulation relates states whose steps are step-wise related, i.e. they can mimic their steps. We will have to think about how a word is accepted, not just whether the word is accepted. We can formally describe a simulation between two states q and p as a parity game between two players, called Spoiler and Duplicator. Duplicator wants to prove that p can step-wise mimic the movement of q. Spoiler wants to disprove that.

We know, that simulation relations are efficiently computable, but they are often limited by their size, which can be much smaller than other GFQ[1]/GFP[2]/GFI[3] preorders. One such example are trace inclusions, which can be obtained by a modification of the simulation game. In simulation games, Spoiler and Duplicator build two paths called π_0, π_1 by choosing a single transition at a time in an alternating fashion. Duplicator usually sees only the next Spoiler's step. We have an option to introduce a *lookahead* in section 3.5, i.e. Duplicator sees a certain amount of Spoiler's steps.

[1]Good for quotienting
[2]Good for pruning
[3]Good for inlcusion

Chapter 3

Simulations

This chapter is focused on explaining various simulation relations over Büchi automata. Simulation relations are preorders on the state space requiring that whenever $s \preceq s'$, i.e. s' simulates s, state s' can mimic all stepwise behavior of s. We also need to address that $s' \preceq s$ is not guaranteed, so state s' can perform transitions that state s cannot match. Simulation relations are often used to verify, whether one system correctly implements an other more abstract system. There are many simulation types such as direct, fair and delayed, which will be the primary topic of this chapter. Simulations are usually used to reduce the state space of an automaton, while preserving the language of the automaton or to prune the state space in algorithms for complementing Büchi automata or testing their language inclusion.
A Simulation itself is a binary relation on the states of an automaton. Simulation between two state q_0 and p_0 can be described as a game played by two players - a Spoiler and a Duplicator. The aim of the game is that the latter player wants to prove that q_0 can mimic the steps (behavior) of the state p_0 and the former player wants to disprove this claim. We will talk about this game more in depth in section 3.7.1.
We usually assume that the automaton has no dead ends, i.e., from each of automaton A's states, there is a possibility to get to some state in F with a path of length at least 1. The language defined by the automaton A is $L(A) = \{w \in \Sigma^* \mid A \text{ has an initial and final trace on } w\}$. Given a Büchi automaton A and $(q_0, q_0') \in Q^2$ we can declare multiple types of simulations. Direct, fair and delayed simulation all have different levels of coarseness. We will now talk about them more.

3.1 Direct simulation

Direct simulation requires that the accepting states will match each other immediately, i.e. the strongest condition. Thanks to this strong condition, direct simulation can be used to safely reduce states and transition of an automaton.
Let us say that q and p direct-simulate each other. We can then merge these two states together into one and it will not affect the automaton's language. On top of that, when q is directly simulated by p and both of them can be reached by a predecessor s with the same label, then we know that the transition from s to q can be deleted without changing the automaton's language.

Definition 5 *$q \preceq_{di} q'$ iff $q \preceq_f q' \wedge \forall i : q_i \in F$, then also $q_i' \in F$.*

Every time a run π visits an accepting state q_i, then run π' must also visit an accepting state q'_i at the same time.

Lemma 1 *Direct simulation is GFQ (good for quotienting) and also GFP (good for pruning).*

Formally, a simulation between two states p_0 and q_0 can be described in terms of a game between these two players, Spoiler and Duplicator. The latter wants to prove that q_0 can step-wise mimic the movement of state p_0. This game starts in an initial configuration (q_0, p_0). It starts with Spoiler choosing a symbol $\alpha \in \Sigma$ and a corresponding transition denoted as $\pi_0 = p_0 \xrightarrow{\sigma_0} p_1 \xrightarrow{\sigma_1} \dots$ and $\pi_1 = p_1 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \dots$. Duplicator wins the game if $C^{di}(\pi_0, \pi_1)$ holds:

Definition 6 $C^{di}(\pi_0, \pi_1) \iff \forall(i \geq 0) \cdot p_i \in F \Rightarrow q_i \in F$

Since we already formally defined direct simulation, we can now try and compute a direct simulation relation over a Büchi automaton A.

Example 2 *Let us have a Büchi automaton A from figure 3.1. We now want to compute its simulation relation.*

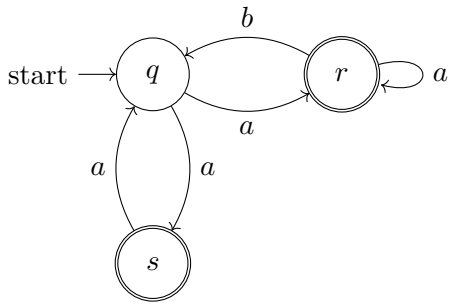

Figure 3.1: Automaton A

$\preceq_{di}$	**q**	**r**	**s**
q	1	1	0
r	0	1	0
s	0	1	1

Table 3.1: Relation table for direct simulation relation over automaton A

We can see how the simulation relation of the automaton A from Figure 3.1 looks like. In Table 3.1 above, we can see the final result, i.e. direct simulation relation. Table 3.1 tells us which states simulate which. We can use that information for quotienting, pruning, language inclusion checks. We will have an example on quotienting and pruning in later

chapter using simulation relations. We will now go over an algorithmic way of computing the direct simulation relation using Algorithm 1.

Algorithm 1: Complement to a direct simulation relation

Input: an NBA $\mathcal{A}$
Output: $\not\sqsubseteq_R$

1: **for** $q \in Q, a \in A$ **do**
2: compute $\delta^r(q,a)$ as a linked list;
3: compute $card(\delta(q,a))$;
4: initialize all $N(a)$ with 0;
5: $\omega = \emptyset$;
6: $\mathcal{C} =$ `NEWQUEUE()`;
7: **for** $i \in F$ **do**
8: **for** $j \in Q - F$ **do**
9: **if** $(i \in F \wedge j \notin F) \vee (\delta(i,a) \neq \emptyset \wedge \delta(j,a) = \emptyset$ *for some* $a \in \Sigma)$ **then**
10: $\omega = \omega \cup \{(i,j)\}$;
11: `ENQUEUE`$(\mathcal{C},(i,j))$;
12: **while** $\mathcal{C} \neq \emptyset$ **do**
13: $(i,j) =$ `DEQUEUE`$(\mathcal{C})$;
14: **for** $a \in A$ **do**
15: **for** $k \in \delta^r(j,a)$ **do**
16: $N(a)_{ik} = N(a)_{ik} + 1$;
17: **if** $N(a)_{ik} == \text{card}(\delta,(k,a))$ **then**
18: **for** $j \in \delta^r(i,a)$ **do**
19: **if** $(j,k) \notin \omega$ **then**
20: $\omega = \omega \cup \{(j,k)\}$;
21: `ENQUEUE`$(\mathcal{C},(j,k))$
22: **return** ω

Example 3 *Let us have a Büchi automaton A. We will show a step-by-step usage of the algorithm from article [7] to compute a direct simulation preorder.*

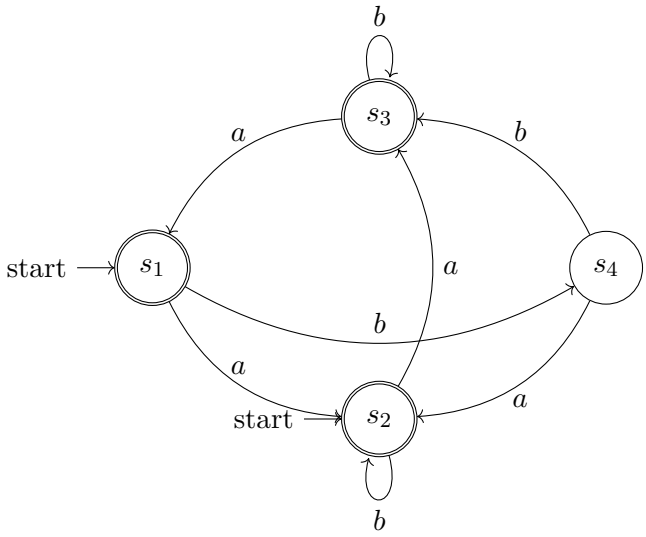

Figure 3.2: Automaton A

We can now compute automaton's A direct simulation relation. We will use algorithm from [7], which runs in time $\mathcal{O}(mn)$ and space $\mathcal{O}(n^2)$. This algorithm is not the fastest one invented, but it is fairly easy to use. This algorithm computes a complement $\not\sqsubseteq$ to a preorder $\sqsubseteq$.
We will now go over the algorithm from article [7], given here as algorithm 1.

Step 1: We first have to compute our linked list $\delta^r(q, a)$ and $card(\delta(q, a))$. Afterwards we have to initialize all $N(a)$s with zeros. Set queue ω as empty for now, same for queue $\mathcal{C}$.

Step 2: What we need to do now is initialize the queue ω, i.e. lines 6-8. Our initialized queue ω should in this case look as follows: $\omega = \{(1, 4), (2, 4), (3, 4)\}$.

Queue ω: $\omega = \{(s_1, s_4), (s_2, s_4), (s_3, s_4)\}$.

Queue $\mathcal{C}$: $\mathcal{C} = \{(s_1, s_4), (s_2, s_4), (s_3, s_4)\}$.

Step 3: We are now at line 10. We start with a pair (s_1, s_4). This pair gets deleted from queue $\mathcal{C}$. At line 12, we choose symbol a first, but there is no transition from node s_4 to s_1 with symbol a, so we end at line 13 with $k = \emptyset$. We now take symbol b, which returns us $k = \{s_1\}$ at line 13. If we proceed, we get to line 15, where we see, that $N(a)_{s_1,s_1}$ is 1 and also $card(\delta, (k, a))$ is 1. At line 16, $j = \emptyset$. No new pair was added to ω.

Queue ω: $\omega = \{(s_1, s_4), (s_2, s_4), (s_3, s_4)\}$.

Queue $\mathcal{C}$: $\mathcal{C} = \{(s_2, s_4), (s_3, s_4)\}$.

Step 4: We take a pair (s_2, s_4). With symbol a, there are no transitions from s_4 with symbol a, so we decide to go to symbol b already. With symbol be, we get $k = \{s_1\}$. Condition at line 15 is satisfied, i.e. $N(a)_{s_2,s_1}$. We get $j = \{s_2, s_4\}$. We add to ω and also to $\mathcal{C}$ pairs (s_2, s_1) and (s_4, s_1).

Queue ω: $\omega = \{(s_1, s_4), (s_2, s_4), (s_3, s_4), (s_2, s_1), (s_4, s_1)\}$.

Queue $\mathcal{C}$: $\mathcal{C} = \{(s_3, s_4), (s_2, s_1), (s_4, s_1)\}$.

Step 5: We take a pair (s_3, s_4). We already know from previous steps, that we can skip the symbol a in this case. With symbol b, we get $k = \{s_1\}$ again. In this case $j = \{s_3, s_4\}$. We add to ω and $\mathcal{C}$ pair (s_3, s_1), we will not add the pair (s_4, s_1), since it is already in the queue ω.

Queue ω: $\omega = \{(s_1, s_4), (s_2, s_4), (s_3, s_4), (s_2, s_1), (s_4, s_1), (s_3, s_1)\}$.

Queue $\mathcal{C}$: $\mathcal{C} = \{(s_2, s_1), (s_4, s_1), (s_3, s_1)\}$.

Step 6: Other pair we take out of the queue $\mathcal{C}$ is (s_2, s_1). We start with symbol a, we recieve $k = \{s_3\}$. Cardinality condition on line 15 is satisfied. We get $j = \{s_1\}$. We add to the queues a pair (s_1, s_3). We take a look at symbol b, which returns $k = \emptyset$. No more additions to the queues.

Queue ω: $\omega = \{(s_1, s_4), (s_2, s_4), (s_3, s_4), (s_2, s_1), (s_4, s_1), (s_3, s_1), (s_1, s_3)\}$.

Queue $\mathcal{C}$: $\mathcal{C} = \{(s_4, s_1), (s_3, s_1), (s_1, s_3)\}$.

Step 7: Other pair we investigate is (s_4, s_1). In case of both symbols a and b, we can not make any additions to the queues. We move to other pair.

Queue ω: $\omega = \{(s_1, s_4), (s_2, s_4), (s_3, s_4), (s_2, s_1), (s_4, s_1), (s_3, s_1), (s_1, s_3)\}$.

Queue $\mathcal{C}$: $\mathcal{C} = \{(s_3, s_1), (s_1, s_3)\}$.

Step 8: Next in order is pair (s_3, s_1). Symbol a gives a $k = \{s_3\}$. Cardinality condition is satisfied. We get $j = \{s_2\}$. We add the pair (s_2, s_3) to the queues. Symbol b is not useful in this step.

Queue ω: $\omega = \{(s_1, s_4), (s_2, s_4), (s_3, s_4), (s_2, s_1), (s_4, s_1), (s_3, s_1), (s_1, s_3), (s_2, s_3)\}$.

Queue $\mathcal{C}$: $\mathcal{C} = \{(s_1, s_3), (s_2, s_3)\}$.

Step 9: As well for the pair (s_1, s_3), we start with symbol a. We get $k = \{s_2\}$. Cardinality condition is satisfied. We get $j = \{s_3\}$. We can now add the pair (s_3, s_2). We continue with symbol b, which arranges that $k = \{s_3, s_4\}$. For s_3, j is empty set. For s_4, the cardinality condition is not satisfied, i.e. $card(\delta(k, a))$ is 2.

Queue ω: $\omega = \{(s_1, s_4), (s_2, s_4), (s_3, s_4), (s_2, s_1), (s_4, s_1), (s_3, s_1), (s_1, s_3), (s_2, s_3), (s_3, s_2)\}$.

Queue $\mathcal{C}$: $\mathcal{C} = \{(s_2, s_3), (s_3, s_2)\}$.

Step 10: Next is pair (s_2, s_3). Symbol a makes $k = \{s_2\}$, $j = \{s_1\}$. That adds (s_1, s_2) to the queues. With symbol b, $k = \{s_3, s_4\}$. Node s_3 gives us $j = \{s_2, s_4\}$. Only (s_4, s_3) gets added, because (s_2, s_3) was already in ω. For node s_4, $card(\delta(s_4, b))$ is 2, so no new pairs to be added.

Queue ω: $\omega = \{(s_1, s_4), (s_2, s_4), (s_3, s_4), (s_2, s_1), (s_4, s_1),$
$(s_3, s_1), (s_1, s_3), (s_2, s_3), (s_3, s_2), (s_1, s_2), (s_4, s_3)\}$.

Queue $\mathcal{C}$: $\mathcal{C} = \{(s_3, s_2), (s_1, s_2), (s_4, s_3)\}$.

Step 11: The last pair we will be taking care of is (s_3, s_2), the rest of the pairs will not do anything to the result of the algorithm. We would continue as before. The symbol a will not add anything to the queues, so we will skip it. Symbol b gives us $k = \{s_2, s_4\}$. We already know that node s_4 with symbol b does not satisfy the cardinality condition on line 15. Node s_2 gives us $j = \{s_3, s_4\}$. We only add (s_4, s_2) to the queues, since (s_3, s_2) was already there.

Queue ω: $\omega = \{(s_1, s_4), (s_2, s_4), (s_3, s_4), (s_2, s_1), (s_4, s_1),$
$(s_3, s_1), (s_1, s_3), (s_2, s_3), (s_3, s_2), (s_1, s_2), (s_4, s_3), (s_4, s_2)\}$.

Queue $\mathcal{C}$: $\mathcal{C} = \{(s_1, s_2), (s_4, s_3), (s_4, s_2)\}$.

Step 12: There are still three steps left, but we already ended up with the result. The remaining pairs in the queue $\mathcal{C}$ would be processed as the ones before. Nothing would be added due to the pair already being in ω, k or j being empty sets, or the cardinality condition would not be satisfied. Nevertheless, we already found an identity of the set $\preceq_{di}$. We would continue doing the same work until the queue $\mathcal{C}$ was empty.

Below in Table 3.3, we can see the final result, i.e. direct simulation relation over the automaton A from Figure 3.2. Above the example, we can see Algorithm 1, which we used to compute Table 3.3.

$\preceq_{di}$	s_0	s_1	s_2	s_3
s_0	1	0	0	0
s_1	0	1	0	0
s_2	0	0	1	0
s_3	0	0	0	1

Table 3.2: Table for direct simulation relation over automaton A

3.2 Fair simulation

In fair simulation, which has the weakest condition of these simulations, Duplicator has to visit an accepting state infinitely often only if the Spoiler did so. Fair simulation is not enough by itself to safely remove a transition or merge state together without affecting the original language of an automaton.

Definition 7 *$q \preceq_f q'$ iff there are infinitely many j such that $q'_j \in F$ or there are only finitely many i such that $q_i \in F$.*

Lemma 2 *Fair simulation is neither GFQ or GFP.*

Direct simulation between p_0 and q_0 can be described as a game between two players, the same thing applies to fair simulations. Spoiler chooses a symbol $\alpha \in \Sigma$ and a corresponding path denoted as $\pi_0 = p_0 \xrightarrow{\sigma_0} p_1 \xrightarrow{\sigma_1} \dots$ and $\pi_1 = p_1 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \dots$. Duplicator wins the game if $C^f(\pi_0, \pi_1)$ holds:

Definition 8 *$C^f(\pi_0, \pi_1) \iff \pi_0$ is fair, then π_1 is fair*

Fairness means that all words starting from state q have a corresponding run to the same word starting from state q'. And, for all accepting words' runs starting from state q, there must exist a run with accepting word starting from state q', where the words have to correspond, i.e. be the same.

Example 4 *Let us have a Büchi automaton A in Figure 3.3. We will compute a fair simulation relation over the automaton A.*

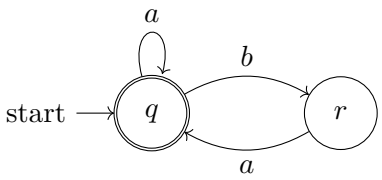

Figure 3.3: Automaton A

$\preceq_f$	q	r
q	1	0
r	1	1

Table 3.3: Table for fair simulation relation over automaton A

As we can clearly see, only $r \preceq_f q$ is satisfied, but not the other way around. There is no transition with symbol b that state r could use to get to state q.

3.3 Delayed simulation

Delayed simulation does not have as strong accepting condition as direct simulation. It introduces a delay in which we can still match a state. If the i-th state of path from state q is accepting, then there exists such $j \geq i$ such that j-th state of matching path from state p is accepting. Therefore we can find more pairs of states that we can reduce safely by merging the together. However, delayed simulation does not guarantee a safe delete of a transition while preserving the language of an automaton.

Definition 9 *$q \preceq_{de} q'$ iff, for all i, if $q_i \in F$, then there is also $j \geq i$ such that $q'_j \in F$.*

Definition 9 tells us that every time run π visits an accepting state q_i, then run π' has to visit at least one accepting state q'_j later on to cover the run π.

Lemma 3 *Delayed simulation is GFQ, but not GFP.*

Also delayed simulation between two states p_0 and q_0 can be describe as a game between two players, Spoiler and Duplicator. The game also starts with Spoiler choosing a symbol $\alpha \in \Sigma$ and a corresponding path denoted as $\pi_0 = p_0 \xrightarrow{\sigma_0} p_1 \xrightarrow{\sigma_1} \ldots$ and $\pi_1 = p_1 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \ldots$, where π_1 is Duplicator's path. Duplicator wins the game if $C^{de}(\pi_0, \pi_1)$ holds:

Definition 10 $C^{de}(\pi_0, \pi_1) \iff \forall(i \geq 0) \cdot p_i \in F \Rightarrow \exists(j \geq i) \cdot q_j \in F$

With this formal definition, we can compute a delayed simulation relation over a Büchi automaton A, defined below. $C^{di}(\pi_0, \pi_1)$ implies $C^{de}(\pi_0, \pi_1)$, which also implies $C^{f}(\pi_0, \pi_1)$.

Example 5 *Let us have a Büchi automaton A in Figure 3.4. We will compute a delayed simulation relation over the automaton A.*

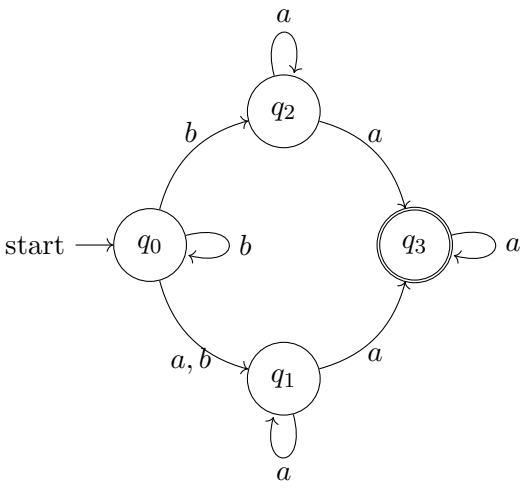

Figure 3.4: Automaton A

$\preceq_{de}$	q_0	q_1	q_2	q_3
q_0	1	0	0	0
q_1	1	1	1	1
q_2	1	1	1	1
q_3	1	1	1	1

Table 3.4: Table for delayed simulation relation over automaton A

We can compute the delayed simulation relation using the formal definition from above, or we can transform the automaton A into a parity game and use Jurdzinski's lifting algorithm [8] to solve the parity game. All these steps can be found in [5]. Table 3.4 above displays a delayed simulation relation that we can use further to minimize an automaton, or check language inclusion... We will have an example on how to minimize automaton using delayed simulation relation in chapter 5.

3.4 Multipebble simulation

In this type of simulation the Duplicator is given n pebbles which can be used to hedge her bets and delay the resolution of nondeterminism. Therefore she has increased power and

can produce coarser GFQ preorders.
Multipebble, direct and delayed simulations are also GFQ preorders, but they are more coarser than their basic versions. They are PTIME computable for a fixed number of pebbles.
However, multipebble simulations are overall PSPACE-hard, which is hard to compute. It is also exponential with the number of pebbles we are using. That is the reason why lookahead simulations are much more popular instead. These are used to compute an under-approximations of multipebble simulations.

3.5 Lookahead simulations

In a standard simulation game, the players are choosing transitions to build a path π_0, π_1. In this case, Duplicator knows only the next step of Spoiler and moves by a single transition. We can obtain a coarser relation if we let Duplicator see a fixed number of *lookaheades* of Spoiler's chosen transitions. In extreme case, the Duplicator can even see the whole Spoiler's path in advance.
The greater the lookahead we choose, the harder the simulation is to compute. In the case of lookahead $k = 1$, the lookahead simulation is identical to the ones we already described. The *lookahead* is put under Duplicator's control. Every round she has to choose according to Spoiler's move how much *lookahead* she needs this round (up to k, a set number of *lookaheads*) [4].
Let us have a configuration of a game denoted by a pair of states (q_0, q_1). From such configuration, one round of the game is played as follows:

- Spoiler chooses a sequence of k consecutive transitions $p_i \xrightarrow{\sigma_i} p_{i+1} \xrightarrow{\sigma_{i+1}} \dots \xrightarrow{\sigma_{i+k-1}} p_{i+k}$
- Duplicator chooses a *lookahead* m such that $1 < m \leq k$
- Duplicator then responses with a sequence of m transitions $q_i \xrightarrow{\sigma_i} q_{i+1} \xrightarrow{\sigma_{i+1}} \dots \xrightarrow{\sigma_{i+m-1}} q_{i+m}$
- The rest of Spoiler's moves $p_{i+m} \xrightarrow{\sigma_{i+m}} p_{i+m+1} \xrightarrow{\sigma_{i+m+1}} \dots \xrightarrow{\sigma_{i+k-1}} p_{i+k}$ is forgotten and the game continues with configuration (p_{i+m}, q_{i+m}). The game either halts or continues forever, which creates two infinite paths.

3.6 Backward direct simulation

Backward direct simulation $\sqsubseteq^{bw-di}$ is defined like ordinary simulation, except the fact, that the transitions are going backwards:
Consider an initial configuration (p_i, q_i), Spoiler takes a transition $p_{i+1} \xrightarrow{\sigma_i} p_i$, Duplicator now replies with a transition $q_{i+1} \xrightarrow{\sigma_i} q_i$, so the configuration after this round is (p_{i+1}, q_{i+1}). Along the way, both Spoiler and Duplicator build infinite backward traces, $\pi_0 = \dots \xrightarrow{\sigma_1} p_1 \xrightarrow{\sigma_0} p_0$ for Spoiler and for Duplicator $\pi_1 = \dots \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_0} q_0$. In order for Duplicator to win, both accepting and initial states have to be matched:

$$\mathcal{C}^{bw}_{I,F}(\pi_0, \pi_1) \iff \forall(i \geq 0) \cdot p_i \in F \Rightarrow q_i \in F \wedge p_i \in I \Rightarrow q_i \in I$$

We see, that $p \sqsubseteq^{bw-di} q$ holds if Duplicator has a winning strategy in the backward simulation game starting from (p, q) with winning condition $\mathcal{C}^{bw}_{I,F}$. Backward simulation relation $\sqsubseteq^{bw-di}$ is an efficiently computable good for quotienting preorder.

Lemma 4 *Backward simulation $\sqsubseteq^{bw-di}$ is a* PTIME *computable* GFQ *preorder on nondeterministic Büchi automata.*

relation on NBA	complexity	quotienting	inclusion	pruning
direct simulation $\sqsubseteq^{di}$	PTIME	✓	✓	✓
fair simulation $\sqsubseteq^{f}$	PTIME	X	✓	X
delayed simulation $\sqsubseteq^{de}$	PTIME	✓	✓	X
direct lookahead simulation $\preceq^{k-di}$	PTIME	✓	✓	✓
fair lookahead simulation $\preceq^{k-f}$	PTIME	X	✓	X
delayed lookahead simulation $\preceq^{k-de}$	PTIME	✓	✓	✓
backward direct simulation $\sqsubseteq^{bw-di}$	PTIME	✓	✓	✓

Table 3.5: Summary of results for simulation-like relations on NBA

Table 3.5 above, we can see which simulations are good for quotienting, good for pruning and good for inclusion. We can also see their complexity. An extended version of this table can be found in [12].

3.7 Infinite games

Infinite games introduced in this section are two-player games played on a finite graph. A game has an arena and a winning condition. We will first declare what an arena is and then we will take a look at winning conditions.

Definition 11 *An arena α is a quadruple $\alpha = (V_0, V_1, E, f)$, where V_0 is a set of vertices for Player 0 and V_1 is a set of vertices for Player 1. Edge relation E is $\subseteq (V_0 \cup V_1) \times (V_0 \cup V_1)$. E could also be called a set of moves. f is a strategy.*

The games we will talk about are played by two players, called Player 0 and Player 1. We will denote the player by $\sigma = \{0, 1\}$ and consider Player σ. When referring to the other player, we will speak of the player as Player σ's opponent and denote him as Player $\overline{\sigma}$. We set $\overline{\sigma} = 1 - \sigma$ for $\sigma \in \{0, 1\}$. The two players are called Spoiler and Duplicator. Spoiler is the one starting - Player 0. Duplicator is Player 1. These games have multiple outcomes. Either the game halts, which means, that the Spoiler wins, or the game can produce two infinite runs $\pi = q_0 a_0 q_1 a_1 q_2 \dots$ and $\pi' = q'_0 a'_0 q'_1 a'_1 q'_2 \dots$. With these two runs, we can now introduce the rules to determine the winner of a game played between the two players. These rules are not applied to a game graph of any sort, just a regular NBA.

- Direct simulation: Duplicator wins iff, for all i, if $q_i \in F$, then also $q'_i \in F$
- Fair simulation: Duplicator wins iff there are infinitely many j such that $q'_j \in F$ or there are only finitely many i such that $q_i \in F$
- Delayed simulation: Duplicator wins iff, for all i, if $q_i \in F$, then there is $j \geq i$ such that $q'_j \in F$

3.7.1 Parity games

Parity games are probably the most important type of infinite games. Parity games are usually used in verification and synthesis. Solving a parity game is in NP, but it is unknown, if the game can be solved in polynomial time. This is one of the reasons why this topic is a highly active research topic. The best known algorithm can solve a parity game in exponential time.
Parity games are played by two players - player Zero and player One. Every parity game has to be won either by player Zero or player One. For every parity game, there is a winning strategy for one of these players. A strategy is called a winning strategy if, no matter how Spoiler plays the game, Duplicator always wins. Consider player Zero having a winning strategy f_{v_0} for a game $P(G, v_0)$ and will always stick to it, player Zero will always win every parity game $P(G, v_0)$. This means that we can possibly split the game arena into two different sets according to player's Zero and One win opportunity from a vertex. So the solution of a parity game is a set of winning vertices for each player. The goal of solving a parity game is to determine, whether there are winning strategies for each player and how do they look like.

$$W_0 = \{v \in V \mid \text{player Zero has a winning strategy from } v\}$$
$$W_1 = \{v \in V \mid \text{player One has a winning strategy from } v\}$$

Problem of splitting an arena into W_0 and W_1 is in $NP \cap co-NP$. It is yet to be discovered if this problem is in P. We will talk about the principles of parity games and how they are played in detail in Chapter 4.
Parity games are memoryless determined, i.e. memoryless strategy has to return the same vertex regardless of the history of the play. Such strategy does not depend on the history and can be represented without any memory. Therefore we usually work with memory less strategies, because they are easier to work with. Memory less determinacy of a parity game means that in every parity game, both players win memory less. Memory less strategy is a strategy that does not need memory at all, where one can choose M, where M is a finite set, to be a singleton. We can view memory less strategies as partial functions: $V_\sigma \to V$. For ease in notation, we will use this representation in our thesis.

Definition 12 *A game G is determined iff $V = W_0(G) \cup W_1(G)$.*

Definition 13 *A game G is memory less determined iff, for every position $v \in V$, there exists a memory less strategy that wins the game from a position v for some player.*

Let $\star \in \{di, de, f\}$. We can denote a strategy for Duplicator in game $G^\star_A(q_0, q'_0)$ as a function f which determines the next move for Duplicator according to the history of choices of Spoiler up to a certain point in the game. A strategy f for Duplicator is a winning strategy, when there is no chance for a Spoiler to win - the game is always won by Duplicator. The strategy f is winning for Duplicator whenever $\pi = q_0 a_0 q_1 a_1 \dots$ is an infinite run through the automaton A and $\pi' = q'_0 a'_0 q'_1 a'_1 \dots$ is a run defined by $q'_{i+1} = f(q_0 a_0 q_1 a_1 \dots q_{i+1})$, then Duplicator wins based on π and π'.

Definition 14 *Let A be a Büchi automaton. Let $\star \in \{di, de, f\}$. A state q' of this automaton A direct, delayed, fair simulates another state of automaton Aq, if there is a winning strategy for Duplicator $G^\star_A(q_0, q'_0)$. We can denote such a relationship by $q \preceq_\star q'$.*

Proposition 1 *Let A be a Büchi automaton.*

For $\star \in \{di, de, f\}$, $\preceq_\star$ is a so called preorder (transitive and reflexive relation) on the state set Q.

2. *These relations are ordered by containment: $\preceq_{di} \subseteq \preceq_{de} \subseteq \preceq_f$.*
3. *For $\star \in \{di, de, f\}$, if $q \preceq_\star q'$, then we can say that $L(A[q]) \subseteq L(A[q'])$.*

We got through the basics of parity games, now we can take a look into one.

Example 6 *Below we can see a constructed parity game arena $\mathcal{G}$. We distinguished the two players - player Zero and player One, by a color. Nodes that are blue are part of V_1 and nodes of brown color are in V_0. The last value in V_0's nodes is a symbol. Every node has a priority, which is assigned to each of the nodes by a function p. We will mention this topic in much more detail in chapter 4.*

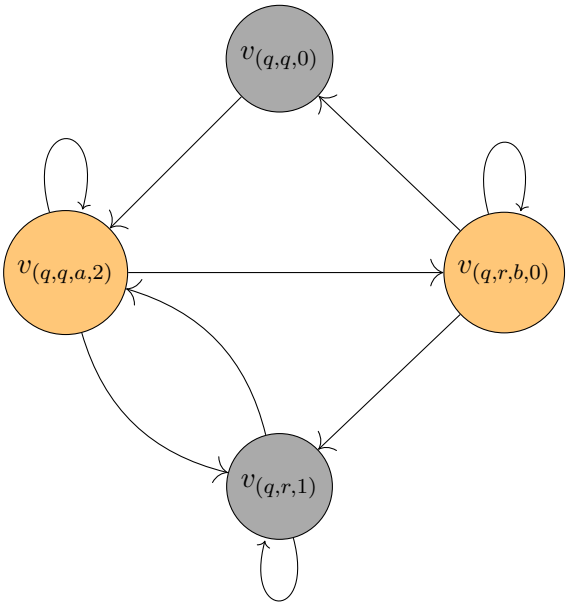

Figure 3.5: Parity game graph

Chapter 4

Transforming simulations into parity games

In this chapter, we will describe an algorithm that computes a parity game graph $G_\star^A$ from a simulation. This step reduces our problem of solving a simulation to solving a parity game. We will describe an algorithm for this conversion for fair and then for delayed simulation. We can create a parity game graph $G_\star^A$, where $\star \in \{di, de, f\}$. We will discuss the option of that in the latter sections more.
We can construct a parity game graph $G_\star^A = (V_0^\star, V_1^\star, E^\star, f^\star)$. This game is played by two players, Spoiler and Duplicator. Therefore player One represents Duplicator and player Zero represents Spoiler. We make both of the players move their pebbles, which are placed on the starting vertex in the game graph, according to their moves. Spoiler is starting and he has to choose a transition (q_i, a, q_{i+1}) and move its pebble to q_{i+1}. Afterwards, it is Duplicator's turn: he has to choose a transition such that $(q'_i, a, q'_{i+1}) \in \Delta$ and moves its (Duplicator's) pebble to q'_{i+1}. If there is no a-transition that starts from q'_i, then the game halts and Spoiler wins.
We will now introduce a straight forward way to construct fair and delayed game from $\star$-simulation, where $\star \in \{di, de, f\}$.

4.1 Fair parity game

A parity game on graph G starting at vertex $v_0 \in V$ is denoted as $P(G, v_0)$. This game is played by two players, player Zero and player One. The game starts by placing a pebble on a vertex v_0. Players One and Zero can move with the pebble according to these rules: with pebble currently being on vertex $v_i \in V_i$, where $i \in \{0, 1\}$. Player Zero (One, respectively) moves the pebble to a neighbor v_{i+1} such that $(v_i, v_{i+1}) \in E$. If this rule cannot be applied, i.e. one of the players cannot move the pebble, because there are no outgoing edges in the game, then the game ends and the player who cannot move loses. Otherwise, the game goes on forever and a path gets defined. We denote such a path as $\pi = v_0 v_1 v_2 \dots$ in G. Such a path is called a play of the game. The only thing left is to determine the game's winner. We do it as follows. Let k_π be the minimum priority that occurs infinitely often in the play π. Player Zero wins if k_π is even, whilst player One wins if k_π is odd.
As we already know how the parity game is played, we can now focus on how to construct the game graph G_A^f for fair parity game. The game graph is $G_A^f = \langle V_0^f, V_1^f, E_A^f, p_A^f \rangle$. This game graph will have only three priorities, i.e. $p_A^f = \{0, 1, 2\}$. Another very important

thing is that for each pair of states $(q, q') \in Q^2$, there will be a vertex $v(q, q') \in V_0^f$ such that player Zero has a winning strategy from $v_{(q,q')}$ iff $q \preceq_f q'$. G_A^f is formally denoted as:

$$V_0^f = \{v_{(q,q',a)} \mid q, q' \in Q \wedge \exists q''((q'', a, q) \in \Delta)\}$$

$$V_1^f = \{v_{(q,q')} \mid q, q' \in Q\}$$

$$E_A^f = \{v_{(q_1,q_1',a)}, v_{(q_1,q_2')} \mid (q_1', a, q_2') \in \Delta\} \cup \{v_{(q_1,q_1')}, v_{(q_2,q_1',a)} \mid (q_1, a, q_2) \in \Delta\}$$

$$\mathrm{p}_A^f(v) = \begin{cases} 0, & (v = v_{(q,q',a)} \vee v = v_{(q,q')}) \wedge q' \in F, \\ 1, & v = v_{(q,q')}, q \in F, \wedge q' \notin F, \\ 2, & \text{otherwise} \end{cases}$$

In order to get a game graph G_A^{de} for delayed simulation we require to make only trivial modification to G_A^f. We will talk about this in the next chapter.

Example 7 *Consider the following Büchi automaton A given in Figure 4.1. Let us transform A into a parity game G_A^f*

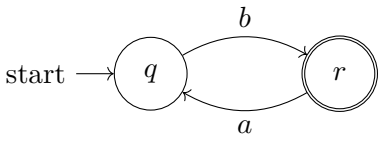

Figure 4.1: Automaton A that we transform into a parity game G_A^f

We use the formal definition from above to create the parity game graph G_A^f. This formal definition can be found in [5]. These formal definitions help us build the parity game graph G_A^f, which can be also found in Figure 4.2, which originally came up with these rules. The game can be solved by using the Jurdzinski's lifting algorithm from [5], or the efficient verstion of it from [5]. The outcome of this algorithm determines the winning set of vertices for each player, which tells us, if there are vertices that are always winning for one of these two players. The running time of this algorithm is $\mathcal{O}(mn)$.

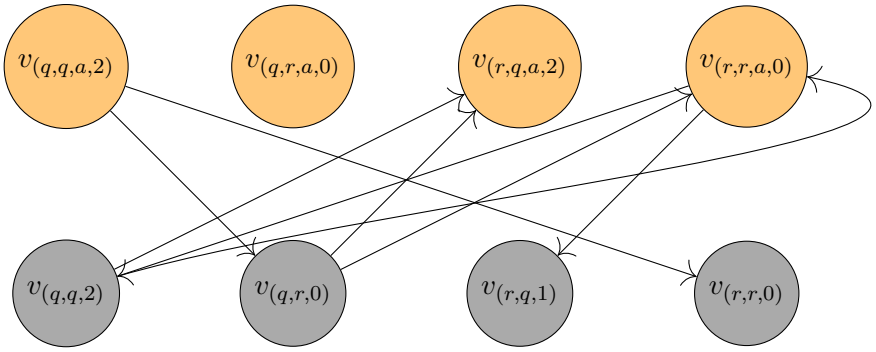

Figure 4.2: Automaton A that we transform into a parity game G_A^f

Above, we can see a complete transformation from a Büchi automaton A into a fair parity game G_A^f. In this stage, we are ready to use Jurdzinski's lifting algorithm to solve this parity game. The result we get is a winning strategy f for each of the players, i.e. player Zero (One, respectively) always wins the game using this strategy f.
After the algorithm's run is complete, every vertex has a value $\rho(v)$ assigned to them. If the value $\rho(v) < \infty$, then player Zero has a winning strategy from that vertex. Otherwise, player One has a winning strategy from the vertex v.

4.2 Direct parity game

Even though there are better algorithms for solving direct simulation, there is a possibility to reformulate a direct simulation into a parity game. The parity game graph $\mathcal{G}_A^{di}$ is exactly the same as $\mathcal{G}_A^f$, except that all nodes have priority of value 0, i.e. $p_A^{di}(v) = 0$. There is also one more modification, we have to remove some edges:

$$E_A^{di} = E_A^f \backslash (\{(v, v_{(q_1,q_1')}) \mid q_1 \in F \wedge q_1' \notin F\} \cup \{(v_{(q_1,q_1')}, w) \mid q_1 \notin F \wedge q_1' \in F\})$$

Vertex v has a set of successors w. Once the game graph $\mathcal{G}_A^{di}$ is constructed, we run it through Jurdzinski's lifting algorithm to solve the game.

4.3 Delayed parity game

In the previous section we talked about constructing G_A^f. Now we have to make these small modifications we mentioned at the end of the previous section. We play the game the same way we play fair parity game. It is still a two player game. First, we will declare this construction formally.

$$V_0^{de} = \{v_{(b,q,q',a)} \mid q, q' \in Q \wedge b \in \{0,1\} \wedge \exists q''((q'', a, q) \in \Delta)\}$$
$$V_1^{de} = \{v_{(b,q,q')} \mid q, q' \in Q \wedge b \in \{0,1\} \wedge (q' \in F \rightarrow b = 0)\}$$

The extra bit b we have here (it is not present in fair parity game) tells us whether Spoiler's pebble has witnessed an accepting state without Duplicator's pebble having witnessed one yet. The edges of G_A^{de} are denoted as follows:

$$\begin{aligned} E_A^{de} = &\{v_{(b,q_1,q_1',a)}, v_{(b,q_1,q_2')} \mid (q_1', a, q_2') \in \Delta \wedge q_2' \notin F\} \\ &\cup \{v_{(b,q_1,q_1',a)}, v_{(0,q_1,q_2')} \mid (q_1', a, q_2') \in \Delta \wedge q_2' \in F\} \\ &\cup \{v_{(b,q_1,q_1')}, v_{(b,q_2,q_1',a)} \mid (q_1, a, q_2) \in \Delta \wedge q_2' \notin F\} \\ &\cup \{v_{(b,q_1,q_1')}, v_{(1,q_2,q_1',a)} \mid (q_1, a, q_2) \in \Delta \wedge q_2' \in F\}. \end{aligned}$$

All that is left is to describe is a priority function of G_A^{de}.

$$p_A^{de}(v) = \begin{cases} b, & v = v_{(b,q,q')} \\ 2, & v \in V_0 \end{cases}$$

Player Zero has a winning strategy in a parity game $\mathcal{G}$ from vertex $v(b, q, r)$ iff $r \preceq_{de} q$, $b = 1$ and $q \in F$ and $r \notin F$. If bit $b = 0$, otherwise. We have now mentioned everything we need to transform a nondeterministic Büchi automaton into a delayed parity game.

Example 8 *Lets us have a NBA A. We will now try and solve the delayed simulation relation for the automaton A. We will go through the construction of the game graph we just mentioned above. Then we will use Algorithm 2 to solve the parity game. Even though it is a naive algorithm, it should not take a long time to compute the simulation relation.*

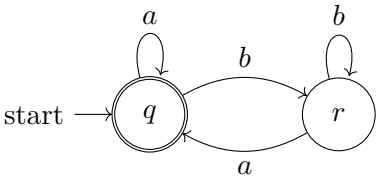

Figure 4.3: Automaton A

We first need to transform the automaton A into a parity game graph $\mathcal{G}$. We can see it below in Figure 4.4.

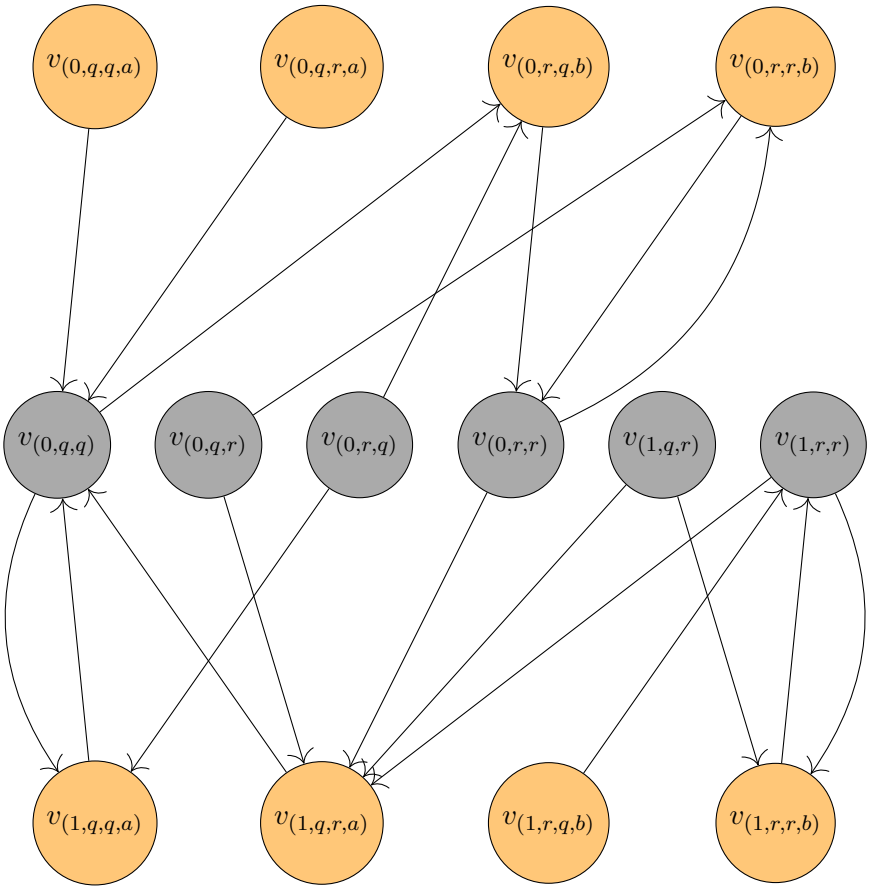

Figure 4.4: Automaton A that we transform into a parity game G_A^f

$\preceq_f$	q	r
q	1	0
r	1	1

Table 4.1: Table for delayed simulation relation over automaton A

Step 1: We start by determining how many vertices with a priority 1 are in the game. This number is denoted as n_1. In our case, $n_1 = 2$. It is important to know, when the $\rho(v)$ overflows to ∞, because $n_1 + 1 = \infty$.

Step 2: We can start with vertices that belong to player Zero (V_0). We can see, that all of these vertices have priority 2, which means that nothing can be changed. We will not do anything this iteration.

Step 3: For the vertices of player One(V_1), we can see that there are two with the priority 1. We will take care of these $(1, q, r), (1, r, r)$, because they will change their $\rho(v)$ value. Both of these vertices will now have $rho(v) = 1$. The rest of player's One vertices will not change, since their priority is 0 and at the same time, there is not a successor for any of those with $\rho(v)$ being ∞. That is it for the first iteration.

Step 4: Player's Zero vertices $(1, r, r, b), (1, r, q, b)$ both only have a successor $(1, r, r)$. That means that they will always take this vertex's $\rho(v)$. Vertices $(1, r, r, b), (1, r, q, b)$ will update theirs $\rho(v) = 1$.

Step 5: In the second iteration we will do the same exact steps as we did before for player One. This time, we will increment the value again - $\rho(v) = 2$ for both vertices $(1, q, r), (1, r, r)$.

Step 6: This is now our third iteration, we will do the exact same steps for both of the players. We will repeat these steps until none of the vertices $\rho(v)$ value changes. That state comes in iteration number five, where the state of the game is as follows: $(1, q, r), (1, r, r)$ both have their $\rho(v) = \infty$, same goes for vertices $(1, r, r, b), (1, r, q, b)$ that belong to player Zero.

Step 7: We can now determine the delayed simulation relation. We can see the result above in Table 4.1.

4.4 Solving a parity game

Since we already know how to construct different types of parity games, we now have to take a look at algorithm for solving parity games. There are multiple parity game solving algorithms, but we will focus on Jurdzinski's lifting algorithm and its modified version from [5].

Let us start with the easier solution - Jurdzinski's lifting algorithm. We have to define a few operators in order to understand what is really going on.

We have a parity game graph $\mathcal{G}$, where n is its number of vertices and m is its number of edges. We assume only three possible priority values: $p : V \to \{0, 1, 2\}$. We have to define $n_1 =\mid p^{-1}(1) \mid$. The algorithm gives every vertex a so called progress measure in the range $D = \{0, \ldots, n_1\} \cup \{\infty\}$. Initially, every one of these vertices of game graph $\mathcal{G}$ have value 0. The progress measure is being incremented until the value of a fixed point is reached. We

assume that D is totally ordered, i.e. D is a partial order in which any two elements are comparable.
For every function $\rho : V \to D$, called a *measure*, and $v \in V$, let:

$$val(\rho, v) = \begin{cases} \langle \min(\{\rho, w) \mid (v, w) \in E\}) \rangle_v, & \text{if } v = V_0, \\ \langle \max(\{\rho, w) \mid (v, w) \in E\}) \rangle_v, & \text{if } v = V_1 \end{cases}$$

Jurdzinski defines a 'lifting' operator, which gets a measure ρ and $v \in V$ and gives us a new, updates measure. In order to define the lifting operator, we first need to define how an individual vertex measure is updated with respect to its neighbors:

$$update(\rho, v) = incr_v(val(\rho, v))$$

The lifted measure, $lift(\rho, v) : V \to D$, is defined as follows:

$$lift(\rho, v)(u) = \begin{cases} update(\rho, v), & \text{if } u = v \\ \rho(u), & \text{otherwise} \end{cases}$$

Now we have declared all the necessary parts in order to describe Jurdzinski's algorithm itself.

Algorithm 2: Jurdzinski's lifting algorithm

1: **foreach** $v \in V$ **do**
2: $\quad \rho(v) := 0$
3: **while** *there exists a* v *such that* $\text{update}(\rho, v) \neq \rho(v)$ **do**
4: $\quad \rho := lift(\rho, v)$

Jurdzinski's lifting algorithm has a running time of $\mathcal{O}(mn)$, where m is the number of edges in the parity game graph $\mathcal{G}$ and n is number of vertices in the parity game graph $\mathcal{G}$. But there is a more efficient implementation of the algorithm 2. We can find it in [5].
This more efficient implementation of the lifting algorithm maintains a set of L of 'pending' vertices v. Measures of this vertex v have to be considered for lifting, because a successor has recently been updated, which results in a requirement to update $\rho(v)$. We also have to look at arrays B and C. These arrays store the value $val(\rho, v)$ for each vertex and also the number of successors u of v. That is denoted as $\langle \rho(u) \rangle_{p(v)} = val(\rho, v)$, which is further denoted as $cnt(\rho, v)$.

Lemma 5 *The lifting algorithm depicted in 3 computes the same function ρ as Jurdzinski's algorithm, in time $\mathcal{O}(m'n_1)$ and space $\mathcal{O}(m')$.*

Algorithm 3: Efficient implementation of the lifting algorithm

1: **foreach** $v \in V$ **do**
2: $\quad B(v) := 0$; $C(v) :=\mid \{w \mid (v,w) \in E\} \mid$; $\rho(v) := 0$;
3: $L := \{v \in V \mid p(v)$ is odd $\}$;
4: **while** $L \neq \emptyset$ **do**
5: $\quad$ let $v \in L$; $L := L \backslash \{v\}$;
6: $\quad t := \rho(v)$;
7: $\quad B(v) := val(\rho, v)$; $C(v) := cnt(\rho, v)$; $\rho(v) := incr_v(B(v))$;
$\quad\quad P := \{w \in V \mid (w,v) \in E\}$;
8: $\quad$ **foreach** $w \in P$ *such that* $w \notin L$ **do**
9: $\quad\quad$ **if** $w \in V_0 \wedge t = B(w) \wedge C(w) > 1$ **then**
10: $\quad\quad\quad C(w) := C(w) - 1$;
11: $\quad\quad$ **if** $w \in V_0 \wedge t = B(w) \wedge C(w) = 1$ **then**
12: $\quad\quad\quad L := L \cup \{w\}$;
13: $\quad\quad$ **if** $w \in V_1 \wedge \rho(v) = B(w)$ **then**
14: $\quad\quad\quad C(w) := C(w) + 1$;
15: $\quad\quad$ **if** $w \in V_1 \wedge \rho(v) = B(w)$ **then**
16: $\quad\quad\quad L := L \cup \{w\}$;

Using this algorithm, we can successfully reduce the state space of the automaton A. We can use quotienting with respect to delayed simulation. This is not true with fair simulation. According to [5], we can indeed say, that quotients with respect to delayed simulations can be substantially smaller than quotients with respect to direct simulation.

Chapter 5

Reduction of states and transitions

There can be a problem with simplifying an automaton while preserving its semantics, i.e. its language. We generally try to reduce the number of states and transitions in order to save some memory and time with our computations. The reduction problem is really important because the complexity of decision procedures using Büchi automata usually depends on the size of the input automaton. One of the best known techniques for state and transition reduction is quotienting, where the states of the automaton are identified by a given equivalence and transitions are projected accordingly. We obtain these equivalences from suitable preorders. We define quotenting w.r.t. (with respect to) preorders, i.e. direct and delayed simulation, direct and delayed multipebble simulation and also direct trace inclusion. Even though fair simulation relation is not GFP nor GFQ, it can be used effectively to prune the state space of an automaton [14]. Then there is also pruning reduction technique, which is a little bit more difficult to implement. While quotienting is reducing the amount of states/transitions with a merging technique, we will explore a different approach which prunes, i.e. removes, the transitions. When we think about this approach, first what comes to mind is that it should be possible to remove a certain transition and still preserve the automaton's language, because there should remain some better transition in the automaton. There is an option for us to change the start state of the automaton A to a different state q. This newly created automaton is denoted as $A[q]$.

5.1 Quotienting

Quotienting is one of the most basic techniques for reducing the state space of an automaton, while preserving its language, i.e. semantics. It merges multiple states that are equivalent into a single state. Let us have an automaton $A = (\Sigma, Q, I, F, \delta)$ and let $\sqsubseteq$ be a preorder on Q with induced equivalence $\approx := (\sqsubseteq \cap \sqsupseteq)$. Given a state $q \in Q$, we denote by $[q]$ its equivalence class with respect to $\approx$ (which is left implicit for simplicity) and for a set of states $P \subseteq Q$, $[P]$ is the set of equivalence classes $[P] = \{[p] \mid p \in P\}$. Afterwards we can define a quotient of the automaton A according to preorder $\sqsubseteq$ as a quintuple $A/\sqsubseteq = ([Q], \Sigma, \delta', [q_i], [F])$, where $\delta' = \{([q], \sigma, [q']) \mid \exists q_1 \in [q], q'_1 \in [q'].(q_1, \sigma, q'_1) \in \delta\}$. For every run $\pi = q_1 \xrightarrow{u_1} q_2 \xrightarrow{u_2} q_3 \xrightarrow{u_3} \dots$ of the automaton A on word $u = u_1u_2u_3\dots$ there exists a corresponding run $\pi' = [q_1] \xrightarrow{u_1} [q_2] \xrightarrow{u_2} [q_3] \xrightarrow{u_3} \dots$ of the automaton $A/\sqsubseteq$, which is accepting if the original run π was accepting. We can state that $L(A) \subseteq L(A/\sqsubseteq)$ holds for every preorder $\sqsubseteq$.

Definition 15 *A preorder $\sqsubseteq$ is good for quotienting (GFQ) if $A \approx A/\sqsubseteq$. GFQ preorders are downward closed, i.e. also called lower set is a subset S of X with the property that any element $x \in X$ that is in relation to some other element of S is also an element of set S. When it comes to quotienting, we are interested in finding coarse and efficiently computable GFQ preorders for NBA. The coarser the simulation relation is, the greater reduction of the automaton we can achieve. Basic examples are given by forward simulation relations.*

As we already know, direct and delayed simulation relations are good for quotienting. This fact can be seen in Table 3.5. We will now set an example automaton that we will try and reduce its state space using quotienting.

Example 9 *Consider a following Büchi automaton A in Figure 5.1. Using direct and delayed simulation relations, we will reduce its state space.*

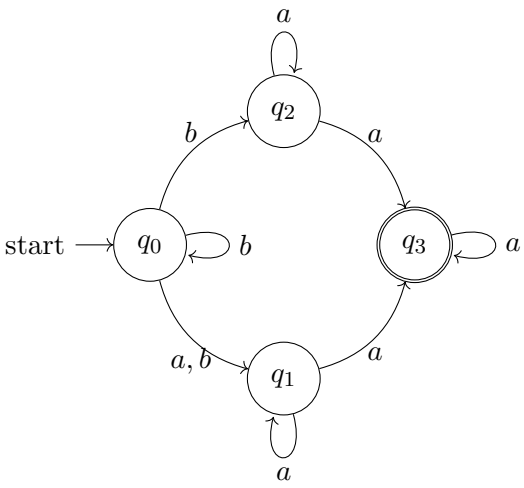

Figure 5.1: Automaton A

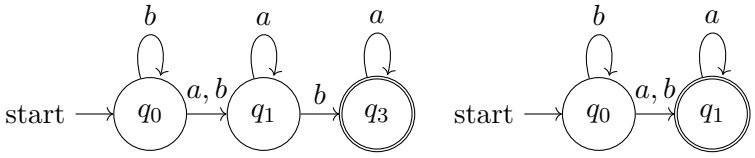

Figure 5.2: Automaton in the left corner is $A/\sqsubseteq^{di}$ and in the right corner $A/\sqsubseteq^{de}$

.

As we can see in Figure 5.2 both of the simulation relation types minimized the state space of the original automaton A. In an automaton A, the states q_1 and q_2 are directly simulating each other, i.e. $q_1 \approx^{di} q_2$ stands. We call this equivalence class as a state r_1. So we have three equivalence classes. Class $[q_0]$ is denoted by r_0 and class $[q_3]$ is denoted by r_3.
On the other hand, all of the states q_1, q_2, q_3 delay-simulate each other, i.e. $q_1 \approx^{d} q_2 \approx^{de} q_3$ stands. That is why automaton $A/\sqsubseteq^{de}$ only has two states.

5.2 Transition pruning reduction technique

Pruning is also one of the widely used techniques for state space reduction. It also preserves an automaton's language. This method does not merge equivalent states into a single state like quotienting, but it takes a different route. Transition pruning reduction technique removes such transitions that can be replaced by some other, naturally better states [11]. Pruning transitions does not only reduce the number of transitions, but also, indirectly, the number of states. That means, when we remove a transition, a state could become a dead end state, and thus be removed from the automaton without changing its language. The reduced automaton is usually much more sparse than the original one, i.e. uses fewer transitions per state and has less non deterministic branching. That fact makes the computation of the pruning perform faster. We will also have an automaton $A = (\Sigma, Q, I, F, \delta)$, then we will declare $P \subseteq \delta \times \delta$, which is a relation on δ and let $maxP$ be the set of maximal elements of P:

$$maxP = \{(p, \sigma, r) \in \delta \mid \nexists (p', \sigma', r') \in \delta : ((p, \sigma, r), (p', \sigma', r')) \in P\}$$

The pruned automaton is defined as $Prune(A, P) := (\Sigma, Q, I, F, \delta')$, where $\delta' = maxP$. We have to note that removing transitions cannot introduce new words into the language, thus $Prune(A, P) \subseteq A$. When also the converse inclusion holds, that means that the language of the automaton is preserved, we say that P is good for pruning (GFP).

Definition 16 *A relation $P \subseteq \delta \times \delta$ is good for pruning (GFP) if $Prune(A, P) \approx A$.*

Just like GFQ (good for quotienting), GFP (good for pruning) is $\subseteq$-downward closed in the space of relations. It is possible to study specific GFP relations obtained by comparing the endpoints over the same input symbol. We have two binary state relations $R_b, R_f \subseteq Q \times Q$ for both source and target endpoints, we define

$$P(R_b, R_f) = \{((p, \sigma, r), (p', \sigma, r')) \in \delta \times \delta \mid pR_bp' \wedge rR_fr'\}.$$

$P(\cdot, \cdot)$ is monotone in both arguments.

Example 10 *Consider NBA called A. Let us use pruning transition technique to reduce the state space of the automaton A.*

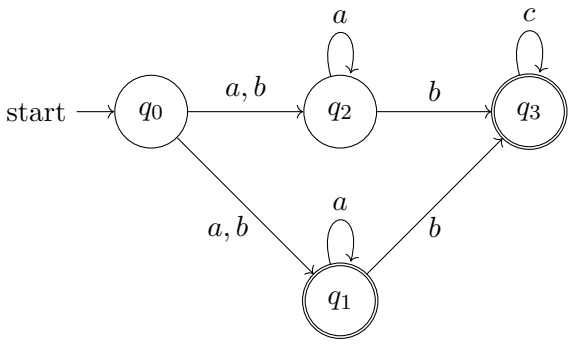

Figure 5.3: Automaton A

$\preceq_{di}$	q_0	q_1	q_2	q_3
q_0	1	0	0	0
q_1	1	1	1	0
q_2	1	1	1	0
q_3	0	0	0	1

Table 5.1: Table for direct simulation relation over automaton A

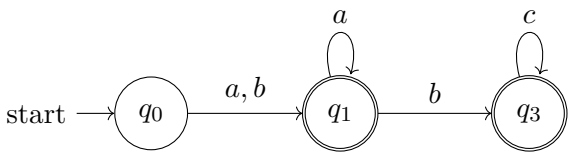

Figure 5.4: Pruned automaton A, $Prune(A, P(id, \sqsubseteq^{di}))$

As we can see in Table 5.1, $q_2 \sqsubseteq^{di} q_1$ holds. Therefore, we can prune the original automaton with respect to $P(id, \sqsubseteq^{di})$. In our case, both transition $q_0 \xrightarrow{\text{a}} q_2$ and $q_0 \xrightarrow{\text{b}} q_2$ will be removed from the automaton, since the transition prevails as $q_0 \xrightarrow{\text{a,b}} q_1$. This is the reason why q_2 becomes a dead state, i.e. state q_2 cannot be reached from any of the initial states, thus can be removed.

5.2.1 Pruning NBA

Results of which simulation are GFP and GFQ are summarized in table 3.5. Both aforementioned techniques lead to a state space reduction of an automaton to some extent. However, by combining these two techniques, we should be able to get better results in general.

Chapter 6

Implementation

In this chapter, we will talk about the implementation of computation the simulations we mentioned in Chapter 3. Everything is implemented in the programming language `C++`, including the command line interface, which is used to determine which simulation we want to compute. The library contains computation of a direct simulation relation, fair simulation relation using parity games, and also delayed simulation relation using parity games. The library is set to work with nondeterministic Büchi automata written in format `.ba`, which is also used in the popular tool `RABIT` [1].

The main task was to implement fair, direct and delayed simulation relation computation. We decided to use a parity game for fair and delayed simulation relations [8]. For the direct simulation relation, we use a basic naive algorithm from [7]. In Chapter 7 we will talk more about the efficiency of these implementations and compare them with the results another tool, which happens to also include these three simulations, named `RABIT` [1].

In order to use some of the algorithms to compute the simulation relations, we need to modify them. We will now go over every modifications we made to the algorithms for solving different problems that occurred during the implementation. We will go over the problems in detail in the upcoming sections.

6.1 Modification of the preorder of a direct simulation

For implementation of a direct simulation, we use algorithm from [7]. This algorithm is designed to compute a preorder complement of nondeterministic finite automata, but it works just fine for ω-automata. The optimal returning data type should be a set of pairs of states of the automaton, so we can use the resulting relation of the direct simulation to, e.g. reduce a state space of an automaton, compute the under approximation of language inclusion etc. The only thing we need to address is that the algorithm 1 computes a complement to the preorder of a direct simulation relation. We would much more prefer to return the preorder.

6.2 Modification of the Jurdzinski's lifting algorithm

We construct the parity game graph as describe in [5]. In this paper, the authors say that these algorithms for solving the parity games only work for parity game graphs that have no self-loops nor dead-ends. We cannot guarantee a parity game graph without a dead-end in our case. We have to introduce extra rules to prevent the algorithm from returning a

result that is not correct. We will now show an example of the algorithm not returning a correct result and explain why it is happening.

Example 11 *Lets us have a parity game graph $\mathcal{G}$ that we constructed from a NBA A.*

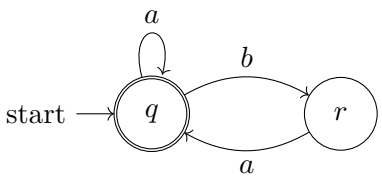

Figure 6.1: Automaton A

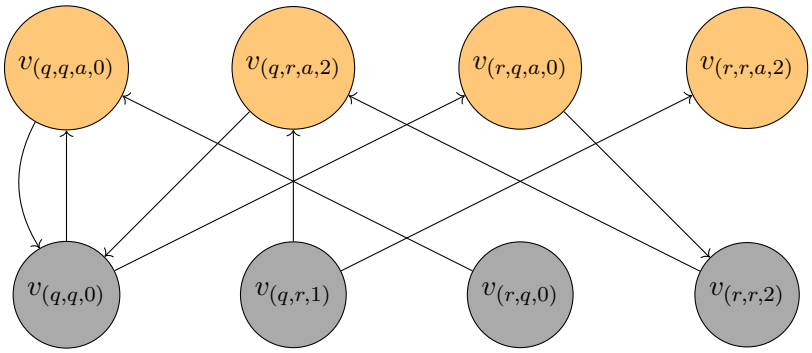

Figure 6.2: Automaton A that we transform into a parity game G_A^f

$\preceq_f$	q	r
q	1	0
r	1	1

Table 6.1: Table for fair simulation relation over automaton A

We can see that $q \not\preceq_f r$, but $r \preceq_f q$ is satisfied. The final result is shown in Table 6.1. A parity game in Figure 6.2 has a dead-end vertex $v_{(r,r,a,2)}$. We also see that the vertex $v_{(q,r,1)}$ of player One should have a value $\rho(v) = \infty$ at the end of the algorithm's run. The rest of the player's One vertices should have value $\rho(v) < \infty$. The vertex $v_{(q,r,1)}$ has nowhere to increase its value to more than $\rho(v) = 1$. We have to introduce a new rule - every vertex of player Zero, that does not have a successor has value $\rho(v) = \infty$, which ensures, that the algorithm runs correctly even on automata with dead-ends.

6.3 Modification of the efficient lifting algorithm

This algorithm was introduced in a paper [5]. This algorithm has been concluded to run in time $(m'n_1)$ and space (m'). We also need to consider an option to modify this algorithm as well, since Algorithm 3 needs to be run over a parity game graph without dead-ends and self-loops. Self-loops are guaranteed not to occur by the way we construct the parity game graph. We have to take care about the 'dead' states. In this case, the condition we

introduced in previous modification in Section 6.2. We will now go over why this condition does not guarantee a correct result. After inspecting this issue, we found out that we need to propagate the value $\rho(v) = \infty$ in a certain fashion to the other vertices. Let us have a vertex v that belongs to player Zero and its value $\rho(v) = \infty$. We have to propagate vertex's $\rho(v)$ value if vertex v has a predecessor w that belong to player One, because player One always wants to increase the progress measure. If we want this rule to be applied to player One as well, we need to have and extra check before propagating the $\rho(w) = \infty$ value. The extra condition is that the sum of successors of w needs to be exactly one. Only then we can propagate the ρ value from player One to player Zero.

6.3.1 Special modification for delayed simulation relation

We found out that there was an issue, still unknown, that caused the program to loop forever. We could not determine a problem on our side, so we opted to use a little modified implementation instead, which worker. The issue was caused by some vertices being added to the working list over and over again. Solution was discovered in this bachelor thesis [13]. The conditions to insert a vertex into the working list are slightly modified. The original efficient lifting algorithm 3 (lines 10-20) have to be modified. The modified version of Algorithm 3 can be seen in Figure 4.

Algorithm 4: Modification of the efficient implementation of the lifting algorithm

1: **foreach** $v \in V$ **do**
2: $B(v) := 0$; $C(v) :=\mid \{w \mid (v,w) \in E\} \mid$; $\rho(v) := 0$;
3: $L := \{v \in V \mid p(v)$ is odd $\}$;
4: **while** $L \neq \emptyset$ **do**
5: let $v \in L$; $L := L \backslash \{v\}$;
6: $t := \rho(v)$;
7: $B(v) :=val(\rho, v)$; $C(v) :=cnt(\rho, v)$; $\rho(v) :=incr_v(B(v))$;
$P := \{w \in V \mid (w,v) \in E\}$;
8: **foreach** $w \in P$ *such that* $w \notin L$ **do**
9: **if** $p(w) \neq 0 \wedge \rho(v) > B(w)$ **then**
10: **if** $w \in V_1$ **then**
11: $L := L \cup \{w\}$;
12: **else if** $w \in V_0 \wedge (p(w) \neq 0 \wedge t = B(w)) \vee (p(w) = 0 \wedge B(w) = 0)$ **then**
13: **if** $C(w) > 1$ **then**
14: $C(w) := C(w) - 1$;
15: **else if** $C(w) = 1$ **then**
16: $L := L \cup \{w\}$;

We will now see an example of an automaton that does not terminate without the modification of the algorithm.

Example 12 *Lets us have an automaton A from Figure 6.3. We will run the algorithm over the delayed parity game graph and see why the algorithm needed the modification.*

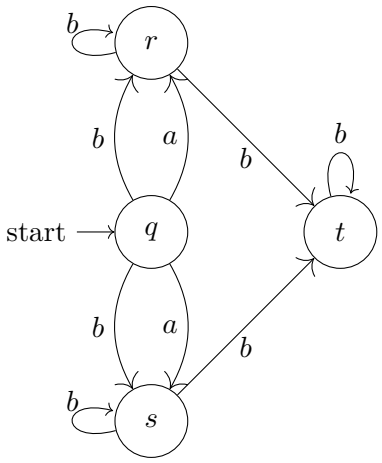

Figure 6.3: Automaton A

We will not be constructing the parity game graph, since there would be more than 80 vertices and 100 edges. Even a relevant part of the game graph would not fit in the page. After the third iteration of the algorithm, the same vertices get put into the working list, but all of them have priority zero or two, which results in the value $\rho(v)$ not changing at all. All of these vertices get stuck in a loop between itself and a neighbour vertex from the other player. We have not figured out, why this issue is happening, while fair simulation relation not needing this modification. We can see the final result of the delayed simulation in Table below 6.2.

$\preceq_{de}$	q	r	s	t
q	1	0	0	0
r	0	1	0	0
s	0	0	1	0
t	0	0	0	1

Table 6.2: Table for delayed simulation relation over automaton A

We do not think that the reason could be related to the fact, that the result is an identity. There were other automata that ran to the end with no issues and had an identity as a result as well.

6.4 Usage of the program

We aimed for a simple command line interface that everyone can use. We will now go over all the available commands.

The desired usage of the program is as follows:

```
./nba-simulations [ --file ] [ --direct | --fair | --delayed ] { --print
                                --dot --fast}
```

Flag of the program	Use case
`file`	Specify file to read the automaton from. We use format `.ba`
`dot`	Write the result into a file with `dot` syntax.
`print`	Result (pair of states) gets printed into the command line.
`fast`	Compute with the efficient version of lifting algorithm
`help`	Prints usage of the program to the command line.
`direct`	Compute direct simulation relation.
`fair`	Compute fair simulation relation.
`delayed`	Compute delayed simulation relation.

Table 6.3: List of all the available flags in the program

We can use the program as follows:

```
./nba-simulations --file=myaut.ba --fair --fast --print
```

This command runs the program over an automaton `myaut.ba` and computes the efficient version of the lifting algorithm for fair simulations. Afterwards, a result gets printed - a set of pair.

The repository also contains a `bash` script which allows the user to select a file to read the automaton from and select a simulation type he wants to compute. The script takes the output of the program (dot file) and converts it to a `.png` file. This final file contains the automaton and colors its transitions to indicate which states simulate each other.

The desired usage of the script is as follows:

```
./convert-dot.sh [ -f ] [ -s ]
```

Flag of the program	Use case
`f`	Specify file to read the automaton from. We use format `.ba`
`s`	Specify what simulation type to run.

Table 6.4: List of all the available flags in the program

We can use the script as follows:

```
./convert-dot.sh -f myaut.ba -s delayed
```

This runs our program over the automaton file `myaut.ba` and computes a delayed simulation relation. After it is done, a `dot` file is generated and then converted to a picture. We will see what states simulate each other.

6.5 Compilation

The repository with the code contains a `Makefile` which can be used to compile the necessary `C++` code and its header files. We will now go through all the source files.

- `automaton.h/.cpp` All the necessary data structures such as states, transitions, alphabet... There is also a parser for the input file. These input files are in `.ba` format.

- `simulations.h` Header with the direct simulation implementation.
- `fair_parity_game.h/fair_parity_game_solver.h` Code with the steps to transform an automaton into a fair parity game. Solver file contains the Jurdzinski's lifting algorithm 2 to solve the fair parity game.
- `delayed_parity_game.h/delayed_parity_game_solver.h` Includes the needed steps to transform an automaton into a delayed parity game graph. Solver file contains the Jurdzinski's lifting algorithm 2 to solve the delayed parity game.

Chapter 7

Experiments

In this chapter, we will be covering various experiments about our implementation of the algorithms we mentioned in previous chapters. We want to determine whether our implementation of the algorithms in fast enough in comparison with widely used tool for manipulating Büchi automata - `RABIT` [1].
For these benchmarks, we used a tool named `pycobench` [9], which allows us to run these tests in parallel (every test creates a new worker). The program `pycobench` outputs a `csv` file that we process easily with a `python` script. We used library `matplotlib` [2] to plot the final results. All of the input automata are from a GitHub repository [10]. All of the upcoming plots will have both of their axis in `ms` and will have a logarithmic line to determine the faster implementation.

7.1 Jurdzinski's lifting algorithm vs. Effective lifting algorithm

This section serves as a comparison between the original lifting algorithm [8] and the effective version of it. We already know that the original lifting algorithm is working naively through all of the vertices of the parity game graph until no changes to progress measure can be made. The effective version of the lifting algorithm takes a different path. It introduces arrays that stores a vertex's neighbour best possible progress measure.
We used exactly `178` nondeterministic BA. Since the original lifting algorithm is naive, it should be much slower with bigger parity game graphs. Most of the cases confirm that idea. There is one known bottleneck to our implementation of the effective version of the algorithm - we take in any parity game graph, even with dead-ends. The original algorithm discards those (transfers these graphs into ones without the dead-ends). Since we want our tool to be robust (take in whatever parity game graph), we had to introduce extra conditions for the algorithm that make it a little bit slower than it should have been.

7.1.1 Fair parity game graph

We will test fair and delayed parity game separately. The construction of the game graphs has some minor changes and also the solving part.

As we can see in Table 7.1, the effective implementation of the lifting algorithm is a little bit faster when it comes to an average time to compute the simulation relation over

Implementation	Average time in ms (standard deviation)
`Original`	1.384 ms
`Efficient`	0.334 ms

Table 7.1: Average compute time for an automaton. Original algorithm is the Jurdzinski's lifting algorithm from [8].

fair parity game graph. The original algorithm timed out 29 times, while the effective only 10 times. The timeout was set to be 60 seconds.

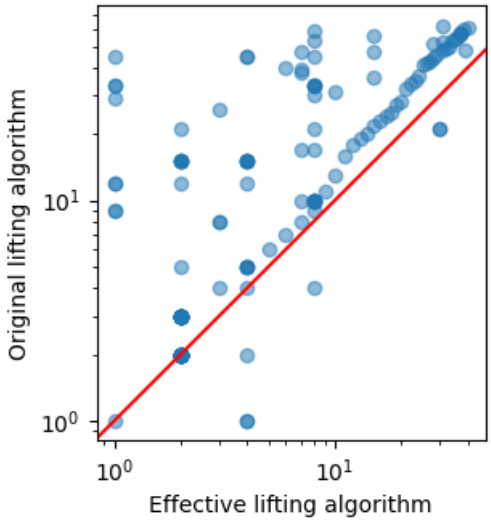

Figure 7.1: Comparison of running times for the effective and original lifting algorithm over a fair parity game graph. Every circle under the red line means that the original implementation was faster for that automaton.

7.1.2 Delayed parity game graph

In this subsection, we will compare the running time of the efficient lifting algorithm and the original one. As we can see in Figure 7.2 the efficient algorithm is much faster in most of the cases. We can inspect the average running time for an automaton in Table 7.2.

Implementation	Average time in ms (standard deviation)
`Original`	2.275 ms
`Effective`	0.352 ms

Table 7.2: Average compute time for an automaton

We have to take in account that delayed parity game graph is just an expanded fair parity game graph. For a delayed parity game graph, there is at most two vertices, that represent a vertex in the fair graph. Hypothetically, the delayed graph can have up to twice as much vertices to go through. That makes the number of the edges much bigger as well.

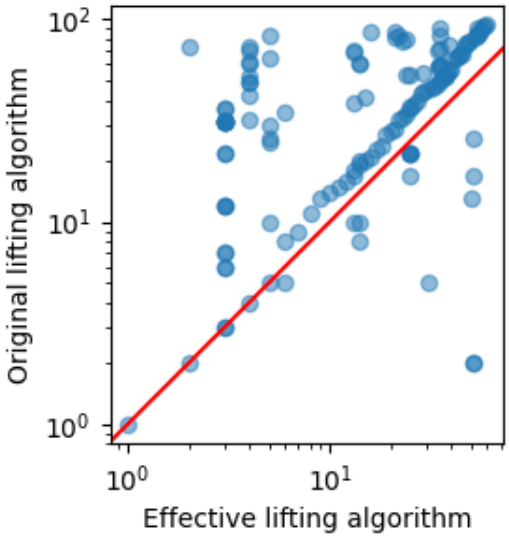

Figure 7.2: Comparison of running times for the effective and original lifting algorithm over a delayed parity game graph. Every circle under the red line means that the original implementation was faster for that automaton.

7.2 Direct simulation relation

In this section, we will take a look at how our implementation stands against RABIT's implementation of the direct simulation relation. We will compare the times of these two programs on nondeterministic BA. We have `628` of these automata. Both of these implementation are naive, so they will not be really fast, when it comes to larger automata. Both of these implementations easily go through smaller automata (around 15 states). The most interesting thing we found out is that RABIT's implementation is much slower when run on small automata (at most 10 states). Our implementation really thrives with the smaller automata, but when it comes to the ones with more states and transitions, the speed falls down. Most of the times for smaller automata for our implementation is `0.0 ms` (instant result), compared to RABIT's `0.1 ms`.

Program	Average time in ms (standard deviation)
`RABIT`	0.126 ms
`Our implementation`	0.094 ms

Table 7.3: Average compute time for an automaton

Figure 7.3 does not look that convincing for our case at the first glance, but we have to take in account that the darker blue circles depict a large quantity of automata than the lighter ones. That is why we took an average time of computation as well.

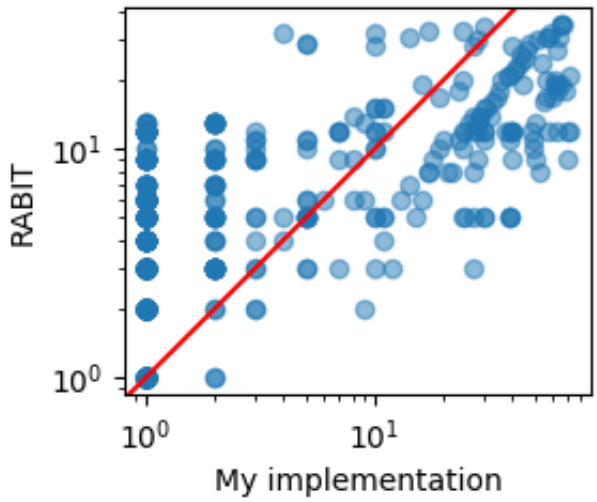

Figure 7.3: Comparison of running times for our implementation compared to RABIT. Every circle under the red line means that RABIT's implementation was faster for that automaton.

7.3 Fair simulation relation

This section will discuss the comparison between the effective implementation of the lifting algorithm and RABIT's implementation. Our input was composed of `177` various semi-deterministic BA. For our implementation, `22` of those timed out. The timeout was set to 60 seconds. We are then left with `155` dots in Figure 7.4. In Figure 7.4, we can see that our algorithm beats RABIT's implementation most of the times, when an input automaton does not have that many states (faster times). Our implementation falls off a lot with bigger automata. We think, that the bottleneck of our implementation is somewhere in the initialization of the parity game graph, but we did not determine a place, where we could make marginal improvements. We could also try and introduce some new conditions to skip some of the vertices from even occurring in the iterations.
We will now look at Table 7.4 and see the average computation time for an automaton from our test suite. Our implementation is quite slower due to the aforementioned reasons.

Program	**Average time in ms (standard deviation)**
`RABIT`	0.092 ms
`Our implementation`	0.260 ms

Table 7.4: Average compute time for an automaton

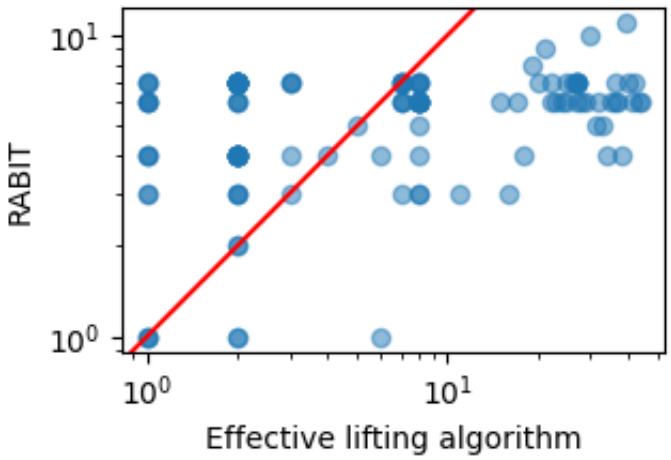

Figure 7.4: Comparison of running times for our implementation of fair simulation compared to RABIT. Every circle under the red line means that RABIT's implementation was faster for that automaton.

7.4 Delayed simulation relation

This is one of our final tests. We compared our implementation of the delayed simulation relation and RABIT's implementation. After going through RABIT's implementation (code) of the delayed simulation, it does not seems like RABIT is using parity games at all. We ran the comparison over set of BA that holds exactly `283` automata. The timeout was set to be 60 seconds again. This time, the RABIT's implemenation timed out `0` times, while ours timed out `55` times. In Figure 7.5 with the result, we can see exactly `228` dots (automata). We can see, that for most of the cases, our RABIT's implementation rose superior to ours.
We do not know the exact bottleneck to our implementation, but we suspect it is the construction of the delayed game graph and we also could improve on discarding some of the vertices from processing. We could not come up with a better result than this.
Below we can see Table 7.5 with the average time of computation for a single automaton from the test suite.

Program	**Average time in ms (standard deviation)**
`RABIT`	0.099 ms
`Our implementation`	0.520 ms

Table 7.5: Average compute time for an automaton

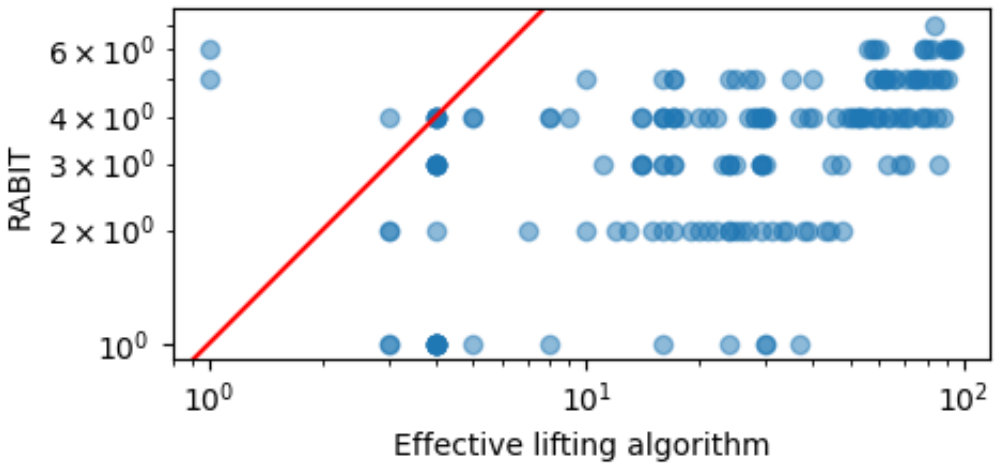

Figure 7.5: Comparison of running times for our implementation of delayed simulation compared to RABIT. Every circle under the red line means that RABIT's implementation was faster for that automaton.

7.5 Comparison of various simulation types

Our final test compares all three of the implemented simulation relations computations. We will look into the speed of these simulations over semi-deterministic BA. We used exactly `247` automata as an input to our program. There were `29` automata that could not be computed in time (they were timed out). The direct simulation timed out `8` times, the fair simulation timed out `29` times and the delayed simulation timed out `20` times. The timeout was set to 60 seconds again. Even though some of these simulation relations seem to be faster than other, not every simulation relation is suitable for every task, e.g. fair simulation is not good for quotienting.
The result of an average run for an automaton can be seen in Table 7.6. For this set of semi-deterministic BA, direct simulation was much superior to the other simulation, when it comes to raw speed. The reason is that direct simulation does not need to check the big parity game graph, but it works only with the original automaton.

Simulation	**Average time in ms (standard deviation)**
`Direct`	0.0013 ms
`Fair`	0.245 ms
`Delayed`	0.631 ms

Table 7.6: Average compute time for an automaton

7.6 Conclusion of the experiments

As we mentioned in some of the experiments, there are some improvements that could be made to the implementation. We are talking especially about the fair and delayed simulation

relation, which tend to time out much more often compared to the direct simulation. When computing the simulation relation over a smaller sized automaton, our implementation is doing fairly well, compared to RABIT.

Chapter 8

Conclusion

In the first part of this thesis, we got through what simulations are and what they are used for. We learnt that they can be used for various things, such as language inclusion checking and reduction of the state space of an automaton. The important part about reducing a state state of an automaton is that we want to preserve the language of the original automaton, while reducing the automaton as much as we can. We took a look into several simulation types such as direct, fair, delayed etc. These three are the most important since they are more 'balanced' than the other simulation types (not big compensations to the speed of the algorithms). We can now say what these simulation are used for and why so. Some of these simulations are good for state space reduction, some are better for the under approximation of the language inclusion. After going through all of the necessary simulations and the theory behind them, we needed to clarify what part are the parity games playing in the field of simulations. Since simulations can be transferred into parity game graphs, we needed to know how to solve them. For that, we mentioned several algorithms, such as the original Jurdzinski's lifting algorithm [8] or the more efficient version of it, which was introduced in [5]. The important use case for simulations is state space reduction of an automaton and there are multiple ways to reduce the state space. We introduced the two basic principles - quotienting and pruning.
The second part of the work was the library written in `C++`. The task was set to implement multiple algorithms for simulations relation. We implemented direct, fair and delayed simulation relations. The fair and delayed simulations are using an algorithm that transfers the automaton into a parity game graph. The whole program has a command line interface with multiple flags to use. We can specify what simulation to run with a specific algorithm. The other useful functionality is that we are able to print the original automaton with transitions that simulate each other. Ours implementation can be faster than RABIT, which is a really good result, but when it comes to bigger automata, our implementation is much slower. We suspect that the reason is that the RABIT's algorithm is faster and really well optimized. The library as a whole is working well, without any known issues.
There were multiple challenges along the implementation of the algorithms, we even had to modify some of them a bit to get them working in our case. There is room for optimization in the code. We could take a closer look at the construction of the parity game graph or how the solver of the parity games could skip some of the vertices of the graph to make the solving faster. There is obviously room for improvement, as we already mentioned, in the performance department, but we could also add some functionality to the command line interface. We would like to add a possibility to reduce the state space of an automaton as well. There would be multiple options to reduce the state space with different simulation

types. We could also add the other aforementioned simulation types and experiment with their results more, to see which of these are best for different tasks.

Bibliography

[1] *RABIT tool* [http://languageinclusion.org/doku.php?id=tools]. Accessed: 2020-06-01.

[2] *Matplotlib library* [https://matplotlib.org/]. 2022.

[3] Baier, C. and Joost Pieter, K. *Principles of model checking*. The MIT Press, 2008. ISBN 978-0-262-02649-9.

[4] Etessami, K. A Hierarchy of Polynomial-Time Computable Simulations for Automata. *CONCUR 2002*. 2002, p. 131–144.

[5] Etessami, K., Wilke, T. and Schuller, R. A. Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata. *Lecture Notes in Computer Science - LNCS*. 2001, p. 694–707.

[6] Grädel, E., Thomas, W. and Wilke, T. *Automata logics, and infinite games: a guide to current research*. Springer-Verlag, 2002. ISBN 3-540-00388-6.

[7] Ilie, L., Navarro, G. and Yu, S. On NFA Reductions. *Lecture Notes in Computer Science - LNCS*. 2004, p. 112–124.

[8] Jurdzinski, M. Small Progress Measures for Solving Parity Games. *STACS*. 2000, p. 290–301.

[9] Lengál, O. *Pycobench* [https://github.com/ondrik/pycobench]. GitHub, 2021.

[10] Lengál, O. *A repository for automata benchmarks* [https://github.com/ondrik/automata-benchmarks]. GitHub, 2021.

[11] Mayr, R. and Clemente, L. Advanced automata minimization. *ACM SIGPLAN Notices*. 2013, p. 63–74.

[12] Mayr, R. and Clemente, L. Efficient reduction of nondeterministic automata with application to language inclusion testing. *Logical Methods in Computer Science*. 2019, p. 1–73.

[13] Tischner, D. *Minimization of Büchi Automata using Fair Simulation*. Bachelor's thesis.

[14] Yi, J. and Zhang, W. Efficient state space reduction for automata by fair simulation. *Proceedings of the 2007 international conference on fundamentals of software engineering*. 2007, p. 380–387.