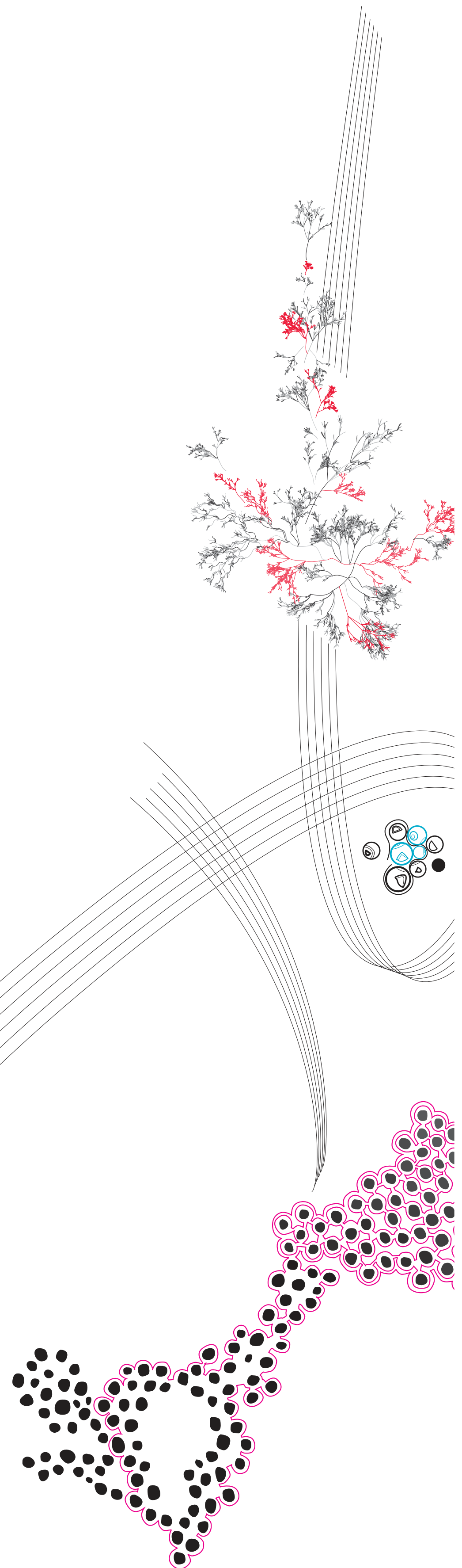

MSc Computer Science
Final Project

Fuzzy Markov chains

Konrad Socha

Supervisor: dr.ing. Ernst Moritz Hahn

June, 2023

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

1 Introduction **1**

2 Context and definitions **3**
2.1 Fuzzy numbers . . . 3
2.2 Markov chains . . . 5
2.3 Fuzzy Markov chains . . . 7
2.4 STORM model checker . . . 8

3 Related works **9**

4 Research question **11**

5 Genetic algorithm for fuzzy numbers **13**
5.1 Restricted matrix multiplication . . . 13
5.2 Algorithm design . . . 15
5.2.1 Fitness functions . . . 16
5.2.2 Wrapper function . . . 16
5.2.3 Initialization . . . 17
5.2.4 Selection . . . 17
5.2.5 Crossover . . . 17
5.2.6 Mutation . . . 20

6 Implementation **23**
6.1 TriangularFuzzyNumber class . . . 23
6.2 FuzzyAnalysisResult class . . . 24

7 Evaluation **26**
7.1 Testing phase . . . 26
7.2 Correctness . . . 26
7.3 Optimization phase . . . 27
7.3.1 Meta-parameters optimization . . . 28
7.3.2 Reflection on parts of the algorithm . . . 30
7.3.3 Comparison of stopping conditions . . . 34
7.3.4 Optimization results . . . 35
7.4 Scalability . . . 36

8 Conclusion and future work **39**

9 Appendix **40**
9.1 Fitness function, raising matrix to the power of n 40
9.2 Fitness function, stationary distribution 40
9.3 Check whether a Markov chain is regular 41
9.4 Check whether a Markov chain is absorbing 43
9.5 Exact algorithm to use as a reference for genetic algorithm's validity 45

Abstract

This paper presents novel methods to enhance the analysis of Markov chains in situations where event probabilities are uncertain, a scenario encountered when information about a system is incomplete or when planning a new system. Fuzzy numbers, which represent imprecise or vague quantities through the use of a membership function, are utilized to model these uncertain probabilities. A genetic algorithm, an optimization technique inspired by natural selection and genetic processes, is utilized to calculate the values of fuzzy transition matrix powers, streamlining the analysis of reachability and stationary distribution for fuzzy Markov chains. Reachability investigates whether a state is accessible within a certain number of steps from another state, while stationary distribution refers to a stable probability distribution that remains unaltered over time. The proposed algorithm is integrated into a probabilistic model checker STORM. Its performance is evaluated through convergence towards optimal solutions and scalability. The evaluation results indicate that although input size, number of steps, and problem type significantly impact the algorithm's performance, it effectively handles uncertainty and offers reliable solutions for fuzzy Markov chain analysis in complex decision-making. This highlights the effectiveness and promise of the proposed method and makes it a valuable contribution to the analysis of fuzzy Markov chains.

Keywords: fuzzy Markov chains, uncertain probabilities, restricted fuzzy matrix multiplication, genetic algorithm, probabilistic model checker, STORM

Chapter 1

Introduction

In this paper we consider an extension of Markov chains that utilizes fuzzy probabilities, namely fuzzy Markov chains [4], and propose methods that improve the analysis of such models.

Classical Markov chains are stochastic models that describe sequences of possible events happening within a system, utilizing states and transitions between them [8]. When using them, we assume that the probabilities of the events (transitions from one state to another) are known. However, in some problems (e.g. reliability engineering, health care management) we do not know the exact probabilities and have to either estimate them from a random sample, or consult experts for such estimates. In both of these cases the probabilities are burdened with uncertainty and are not "crisp" but represented for example by a confidence interval. On such occasions we can model this uncertainty using fuzzy numbers [4], i.e. connected sets of possible values that are mapped to a value between 0 and 1. This mapping assigns weights to each number from the confidence interval.

Using fuzzy numbers we can model the uncertainties of the values in the transition matrix of the Markov chain, substituting them with fuzzy probabilities, where the fuzziness allows us to express the level of confidence within an interval. Using restricted fuzzy arithmetic, it is possible to carry over the basic properties of classical Markov chains, such as convergence of the powers of the transition matrix, onto fuzzy Markov chains [4]. Methods for validating these properties will be fleshed out in the later chapters.

Ideally, thanks to implementing such methods, when one chooses to model some uncertainty in their system using fuzzy Markov chains, they would be able to analyze some properties of this system. The need for such a model could arise when planning a new system or if some information about the existing system is unavailable. The latter is common when dealing with a non-deterministic black-box system, solely by observing its behavior, hence not knowing the probability values. [6]

The project discussed in this paper concerns integrating fuzzy Markov chains into the probabilistic model checker STORM [9] by implementing restricted fuzzy matrix multiplication. It is a fuzzy arithmetic approach where the arithmetic operations on fuzzy numbers are performed under a probabilistic constraint that, despite that the values contain uncertainty, all of the values in each row of the stochastic matrix must sum up to 1. Its challenge is that finding the values of the powers of this matrix, which will be our goal, is computationally difficult and it is preferable to use a directed search algorithm, such as a genetic algorithm, to approximate the solutions. Hence, we decided to implement a genetic algorithm in this project to generate the necessary values.

In Chapter 2 we will present definitions and examples of concepts used in latter chapters. In Chapter 3 we discuss methods for verifying properties of fuzzy Markov chains

currently present in the literature, as well as usage of genetic algorithms used in fuzzy optimization problems. Chapter 4 presents goals and research questions of the project. Chapter 5 goes into more detail with the method that will be implemented. We will present details of the implementation in Chapter 6, and show evaluation results in Chapter 7.

Chapter 2

Context and definitions

We use [5], [7] to define fuzzy subsets, alpha-cuts, and (triangular) fuzzy numbers with inequality operators; and [4], [7] for fuzzy probabilities, Markov chains, and fuzzy Markov chains.

2.1 Fuzzy numbers

Definition 2.1.1 (Fuzzy subset)**.** A *fuzzy subset* $\overline{A}$ of a set Ω is defined by its membership function $\overline{A}\colon \Omega \to [0,1] \subset \mathbb{R}$, where each $x \in \Omega$ is mapped to a real number in the interval [0, 1], and $\mathbb{R}$ represents the set of real numbers.

If $\overline{A}(x)$ is equal to 1, we say that x fully belongs to $\overline{A}$. If $\overline{A}(x)$ equals 0, x does not belong to $\overline{A}$ at all. For values of $\overline{A}(x)$ between 0 and 1, we introduce the term *membership value*. This value signifies the degree of membership or the degree to which x belongs to $\overline{A}$, with 1 being full membership and 0 indicating no membership.

Definition 2.1.2 (α-cut)**.** Let $\overline{A}$ be a fuzzy subset of a set Ω and let $\alpha \in [0,1] \subset \mathbb{R}$ be a real number between 0 and 1, inclusive. Then an *α-cut* of $\overline{A}$, written $\overline{A}[\alpha]$, is defined as: $\overline{A}[\alpha] = \{x \in \Omega \mid \overline{A}(x) \geqslant \alpha\}$.

For $\alpha = 0$, $\overline{A}[0]$ is defined as the closure of the *support* of $\overline{A}$.

Definition 2.1.3 (Support)**.** The support of a fuzzy subset $\overline{A}$ of a set Ω, denoted $sp(\overline{A})$, is defined as the set of all $x \in \Omega$ such that $\overline{A}(x) > 0$.

Therefore, $\overline{A}[0]$ includes all elements $x \in \Omega$ where $\overline{A}(x) > 0$ and potentially some elements where $\overline{A}(x) = 0$, if needed to create a closed set.

Definition 2.1.4 (Fuzzy number)**.** A *fuzzy number* $\overline{N}$ is a fuzzy subset of $\mathbb{R}$ that satisfies the following three conditions:

1. The α-cut with $\alpha = 1$ of $\overline{N}$, denoted $\overline{N}[1]$, is non-empty. This is the set of all elements in $\mathbb{R}$ that fully belong to the fuzzy number $\overline{N}$.
2. All α-cuts of $\overline{N}$, denoted $\overline{N}[\alpha]$ for $0 \leqslant \alpha \leqslant 1$, are closed and bounded intervals in $\mathbb{R}$.
3. The support of $\overline{N}$, denoted $sp(\overline{N})$, is a bounded set in $\mathbb{R}$.

The process of taking α-cuts of fuzzy numbers produces *crisp (non-fuzzy) intervals* in $\mathbb{R}$, denoted $[a, b]$. A 'crisp interval" in this context refers to a standard mathematical interval, where every element either belongs to the set or does not, with no degrees of membership as in fuzzy sets. This process will later allow us to compose fuzzy probabilities.

Here we consider a special case of fuzzy numbers, namely *triangular fuzzy numbers*. These are fuzzy numbers defined by three points that form a triangle in a graphical representation. Figure 2.1 provides an example of a triangular fuzzy number.

Definition 2.1.5 (Triangular fuzzy number)**.** A triangular fuzzy number is a specific type of fuzzy number $\overline{M} = (a/b/c)$ defined by three real numbers a, b, c such that $a < b < c$ and the membership function $\overline{M}\colon \mathbb{R} \to [0,1] \subset \mathbb{R}$ satisfies the following conditions:

1. $\overline{M}(b) = 1$.
2. The graph of $y = \overline{M}(x)$ forms a straight line segment from $(a, 0)$ to $(b, 1)$ for $x \in [a, b]$ and another straight line segment from $(b, 1)$ to $(c, 0)$ for $x \in [b, c]$. In this graphical representation, the fuzzy number forms a triangle where the x-coordinate refers to the value under consideration, and the y-coordinate refers to its degree of membership.
3. $\overline{M}(x) = 0$ for all $x \in \mathbb{R}$ such that $x \leqslant a$ or $x \geqslant c$.

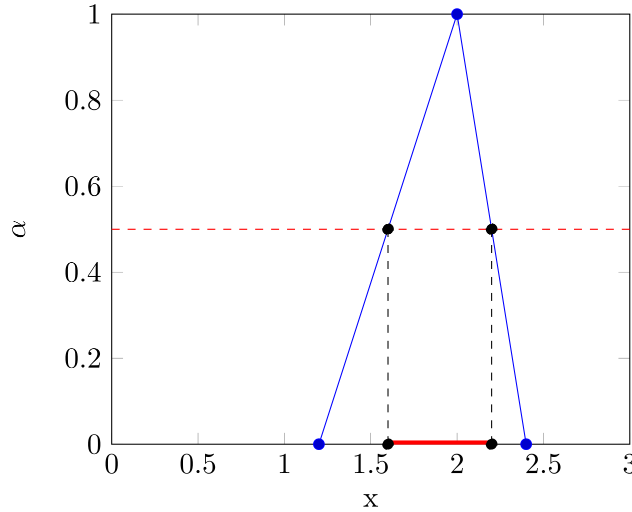

FIGURE 2.1: Triangular Fuzzy Number $\overline{N} = (1.2/2/2.4)$, with $\overline{N}[0.5] = [1.6, 2.2]$

Example 2.1.1 (Triangular fuzzy number)**.** In Figure 2.1, the triangular fuzzy number $\overline{N} = (1.2/2/2.4)$ is depicted. Here, the peak of the triangle is at $x = b = 2$, where the membership function $\overline{N}(x)$ reaches its maximum of 1. The line segment from $(a, 0) = (1.2, 0)$ to $(b, 1) = (2, 1)$ indicates increasing membership values, and the line segment from $(b, 1) = (2, 1)$ to $(c, 0) = (2.4, 0)$ indicates decreasing membership values. Outside the interval $[a, c] = [1.2, 2.4]$, the membership function value is 0, representing non-membership. The dashed red line and the interval $[1.6, 2.2]$ represent the α-cut for $\alpha = 0.5$, showing the set of real numbers whose membership value is at least 0.5. This graphical representation helps to visualize the distribution of membership values in the fuzzy number and the concept of α-cuts.

From this point on, unless stated otherwise, when referring to fuzzy numbers, we will mean triangular fuzzy numbers. On some occasions, we will use the term *triangular-shaped* fuzzy numbers instead.

Definition 2.1.6 (Triangular-shaped fuzzy number)**.** A triangular-shaped fuzzy number is a specific type of fuzzy number $\overline{M} \approx (a/b/c)$ defined by three real numbers a, b, c such that $a < b < c$ and the membership function $\overline{M}\colon \mathbb{R} \to [0,1] \subset \mathbb{R}$ satisfies the following conditions:

1. $\overline{M}(b) = 1$;
2. The graph of $y = \overline{M}(x)$ is continuous, monotonically increasing for $x \in [a, b]$ and monotonically decreasing for $x \in [b, c]$. It does not need to be a straight line in either segment; and
3. $\overline{M}(x) = 0$ for all $x \in \mathbb{R}$ such that $x \leqslant a$ or $x \geqslant c$.

As can be seen from the above definitions, the primary difference between a triangular and a triangular-shaped fuzzy number lies in the shape of the graph of their membership functions. While the former requires straight line segments, the latter allows for curves as well.

Definition 2.1.7 (Comparison between fuzzy and crisp numbers)**.** Let $\overline{N}$ be a fuzzy number and δ be a real number. We define the α-cut of $\overline{N}$ as $[\overline{N}_L(\alpha), \overline{N}_R(\alpha)]$. In essence, α-cut defines a crisp interval at each level of α for a fuzzy number.

We say that $\overline{N} > \delta$ if $\overline{N}_L(\alpha) > \delta$ for all $0 < \alpha \leqslant 1$. Similarly, we say that $\overline{N} < \delta$ if $\overline{N}_R(\alpha) < \delta$ for all $0 < \alpha \leqslant 1$.

Analogously, the non-strict inequalities $\overline{N} \geqslant \delta$ and $\overline{N} \leqslant \delta$ are defined by replacing the strict inequalities in the definitions above with non-strict inequalities. In other words, the comparison has to hold for all elements in all possible α-cuts.

Probabilities can also be represented in terms of fuzzy numbers. We will show this using an example.

Example 2.1.2 (Fuzzy probabilities)**.** Let $X = \{x_1, \ldots, x_n\}$ be a finite set, and let P be a probability function defined on all subsets of X such that $P(x_i) = a_i$, $a_i \in \mathbb{R}$, for all $1 \leqslant i \leqslant n$, where each $0 < a_i < 1$, and $\sum_{i=1}^{n} a_i = 1$. Then, we say that X together with P form a discrete (finite) probability distribution.

Given the presence of uncertainty in the values of a_i, we substitute each a_i with a fuzzy number $\overline{a}_i$, where $0 < \overline{a}_i < 1$ for all i. After this substitution, we still require a valid discrete probability distribution, which implies the existence of $a_i \in \overline{a}_i[1]$ such that $\sum_{i=1}^{n} a_i = 1$.

In other words, we can choose a_i in $\overline{a}_i[\alpha]$ for all α in such a way that we maintain a discrete probability distribution.

Definition 2.1.8 (Interval matrix)**.** An *interval matrix* $A = [a_{ij}]$ is a matrix of intervals where each entry a_{ij} is an interval of real numbers of the form $[a_{ij}, b_{ij}] = \{x \in \mathbb{R} \mid a_{ij} \leqslant x \leqslant b_{ij}\}$. That is, each element in the matrix is an interval on the real number line rather than a single real number.

2.2 Markov chains

Definition 2.2.1 (Markov chain)**.** A *Markov chain* $\mathcal{M}$ is a tuple (S, s_0, P) where:

1. S is a finite set of states.
2. $s_0 \in S$ is the initial state.
3. P is a $|S| \times |S|$ transition probability matrix, where each entry p_{ij} corresponds to the probability of transitioning from state s_i to state s_j. It satisfies $0 \leqslant p_{ij} \leqslant 1$ for all $1 \leqslant i, j \leqslant |S|$, and $\sum_{j=1}^{|S|} p_{ij} = 1$ for all $1 \leqslant i \leqslant |S|$.

The above produces a countably infinite sequence of state changes, or discrete time "steps". We denote elements of P as p_{ij}, and after raising P to the power of n, elements of P^n are denoted by $p_{ij}^{(n)}$. A Markov chain satisfies the memoryless property, also known as the Markov property, which states that the probability of transitioning to any particular state depends solely on the current state and time elapsed, and not on the sequence of states that preceded it. Here we denote probability as *Prob* to differentiate from the transition matrix P:

$$p_{ij} = Prob(s_j \text{ at step } n+1 \mid s_i \text{ at step } n).$$

Definition 2.2.2 (Reachability)**.** In a Markov chain $\mathcal{M}$, a state s_j is *reachable* from a state s_i (in k steps) if there exists such a non-negative integer n ($n \leqslant k$) so that the n-step transition probability $p_{ij}^{(n)} > 0$.

Definition 2.2.3 (Regular Markov chain)**.** A Markov chain $\mathcal{M}$ is said to be *regular* if there exists a positive integer n such that, for all pairs of states s_i and s_j in the state space S, the $(n+1)$-step transition probability $p_{ij}^{(n+1)}$ is greater than zero. In other words, it is possible to transition from any state to any other state in $n+1$ steps with a probability greater than zero.

Definition 2.2.4 (Stationary distribution)**.** A *stationary distribution* π of a Markov chain $\mathcal{M}$ is a row vector of probabilities that satisfies $\pi = \pi P$, where each entry π_i of π is the long-term, time-invariant probability of being in state s_i. The entries of π are non-negative and sum to one: $\sum_{i=1}^{|S|} \pi_i = 1$.

If the Markov chain is ergodic, meaning it is both regular (every state can be reached from every other state in k steps) and aperiodic (the greatest common divisor of the lengths of its cycles is 1), it has a unique stationary distribution. Moreover, as k approaches infinity, the transition matrix P converges to a limit, denoted as $\lim_{k\to\infty} P^k = \Pi$. Each row of the limiting matrix Π is identical and equal to the stationary distribution π of the Markov chain.

Below we present an example of calculating the stationary distribution of a simple crisp Markov chain.

Example 2.2.1 (Stationary distribution: crisp Markov chain)**.** Given the crisp transition matrix

$$\begin{pmatrix} 0.6 & 0.4 \\ 0.2 & 0.8 \end{pmatrix},$$

we can find the stationary distribution by solving the following system of equations

$$\begin{pmatrix} \pi_1 & \pi_2 \end{pmatrix} \begin{pmatrix} 0.6 & 0.4 \\ 0.2 & 0.8 \end{pmatrix} = \begin{pmatrix} \pi_1 & \pi_2 \end{pmatrix},$$

$$\pi_1 + \pi_2 = 1.$$

From the first equation, we have the two equations:

$$\pi_1 = 0.6\pi_1 + 0.2\pi_2,$$

$$\pi_2 = 0.4\pi_1 + 0.8\pi_2.$$

Solving this system of equations along with $\pi_1 + \pi_2 = 1$, we get

$$\pi_1 = \frac{1}{3},$$

$$\pi_2 = \frac{2}{3}.$$

Hence, the stationary distribution is

$$\begin{pmatrix}\pi_1 & \pi_2\end{pmatrix} \begin{pmatrix}\frac{1}{3} & \frac{2}{3}\end{pmatrix}.$$

Definition 2.2.5 (Absorbing state)**.** An absorbing state s_i in a Markov chain is a state such that, once entered, the process will remain in it indefinitely. Mathematically, we say a state s_i is *absorbing* if the probability of staying in the same state s_i in the next time step is 1, that is, $p_{ii} = 1$. Given this, it follows that the probability of transitioning to any other state s_j from s_i must be 0 for all $j \neq i$.

Definition 2.2.6 (Absorbing Markov chain)**.** An *absorbing Markov chain* is a Markov chain that has at least one absorbing state, and every state can reach an absorbing state.

2.3 Fuzzy Markov chains

Definition 2.3.1 (Fuzzy Markov chain)**.** A *fuzzy Markov chain* $\overline{\mathcal{M}} = (S, s_0, \bar{P})$ is a Markov chain where each p_{ij} from the transition matrix is substituted with a fuzzy probability $\overline{p}_{ij}$. These values now make up a fuzzy transition matrix $\overline{P} = (\overline{p}_{ij})$, where there are such $p_{ij} \in \overline{p}_{ij}[1]$ so that $P = (p_{ij})$ is a valid transition matrix for a finite Markov chain (the rows sum up to one).

As we can see from the above definition, despite having uncertainty in the values, the property of rows of the transition matrix summing up to 1 still has to hold. Let us consider we have a method that lets us calculate consecutive powers of $\overline{P}$ that preserves this property, such as the one which will be presented in Chapter 5. We are then able to make claims about *reachability*:

Definition 2.3.2 (Fuzzy reachability)**.** In a fuzzy Markov chain $\overline{\mathcal{M}}$, a state s_j is reachable from a state s_i (in k steps) if there exists such an n ($n \leqslant k$) so that $\overline{p}_{ij}^{(n)} > 0$.

In other words, the state is not reachable if $\overline{p}_{ij}^{(n)} = 0$ (and therefore is actually a crisp number), otherwise it is reachable.

Definition 2.3.3 (Regular fuzzy Markov chain)**.** After replacing every fuzzy probability $\overline{p}_{ij}$ in the stochastic matrix of a fuzzy Markov chain $\overline{P}$ with a crisp number taken from the singleton $\overline{p}_{ij}[1]$, if the resulting crisp Markov chain is a regular Markov chain, then $\overline{P}$ is also a regular (in this case, fuzzy) Markov chain.

The above is true because, if the Markov chain is regular for $\overline{p}_{ij}[1]$, then it is possible to choose a p_{ij} across all α-cuts such that the resulting crisp Markov chain is regular. This is a similar principle as we discussed earlier with preserving a discrete probability distribution after adding uncertainty.

Definition 2.3.4 (Stationary distribution of a fuzzy Markov chain)**.** The *stationary distribution* π of a fuzzy Markov chain is a probability distribution that satisfies $\pi = \pi\overline{P}$, meaning that it remains unchanged through the transitions as determined by the fuzzy transition matrix $\overline{P}$. In this equation, each entry π_i of π represents the long-term, time-invariant probability of the system being in state s_i. The entries of π are non-negative and sum to one: $\sum_{i=1}^{|S|} \pi_i = 1$.

The concepts of ergodicity, uniqueness of the stationary distribution, and convergence of the transition matrix as in the non-fuzzy Markov chain case apply here as well. [4]

Definition 2.3.5 (Absorbing states in fuzzy Markov chains). A state s_i in a fuzzy Markov chain is absorbing if $\overline{p}_{ii} = 1$ and $\overline{p}_{ij} = 0$ (both crisp numbers) for $i \neq j$.

As we can see, absorbing states in fuzzy Markov chains are represented in the same way as in crisp Markov chains, therefore the definition for absorbing chains is also analogous:

Definition 2.3.6 (Absorbing fuzzy Markov chain). An absorbing fuzzy Markov chain is a fuzzy Markov chain in which it is possible to reach an absorbing state in a finite number of steps from every non-absorbing state.

2.4 STORM model checker

STORM [9] is a probabilistic model checker that supports the analysis of both continuous- and discrete-time Markov chains, with different number types (floating-point numbers, exact numbers, parameters), but does not yet support fuzzy numbers nor fuzzy Markov chains. It has a modular setup as well as a Python API which allows for rapid prototyping.

Chapter 3

Related works

In Chapter 5 we will present a method of calculating consecutive powers of the fuzzy stochastic matrix $\overline{P}^n$ based on restricted fuzzy arithmetic, which in the end comes down to a fuzzy optimization problem. In this chapter we present some other known methods for verifying properties of fuzzy Markov chains and compare and contrast it with our chosen method. Since we will attempt to solve our optimization problem with a genetic algorithm, we will also discuss some related attempts at using genetic algorithms to solve fuzzy optimization problems.

[6] gives results on $\overline{P}^n$ for fuzzy Markov chains and [3] shows it has finite convergence, however both papers define their fuzzy Markov chains in terms of *possibility theory*, not fuzzy probabilities. Possibility theory is an alternative to probability theory that employs a possibility measure from 0 to 1, ranging from impossible to possible. In fuzzy Markov chains based on the possibility theory, each row of the transition matrix is a possibility distribution, so it operates under restrictions that each entry is non-negative and the maximum of each row is 1. Then, powers of the transition matrix are calculated using max-min composition (min in place of multiplication and max for addition). Using this method preserves the constraints of possibility distributions, but would not be applicable for probability distributions (because of the constraint that rows have to sum up to 1). We did not use this definition of fuzzy Markov chains as we believe using triangular fuzzy numbers instead conveys more information and is more practical in real-life applications considering one can convert confidence intervals to fuzzy numbers.

Another approach that is actually consistent with our definition of fuzzy Markov chains is to use Zadeh's extension principle as is done in [10]. This principle is a generalized method of extending the crisp domains of functions to fuzzy sets. Therefore, by treating powers of the transition matrix as functions of its elements, it provides extensions of these functions for fuzzy values. However, the authors of the restricted matrix multiplication method [4] claim that their approach is both more comprehensible for humans and computationally efficient. Unfortunately this claim is substantiated solely by presenting two methods on a simple example so that the reader can assess the complexity of both approaches. For lack of better comparisons between the two, we chose restricted matrix multiplication for use in our implementation. It then still remains, however, a hard problem, and so the authors suggest to use a directed search algorithm, such as a genetic one, to approximate the solutions, which we also settled for in this project.

[5] gives an outline for genetic algorithms and fuzzy optimization. Given an optimization problem for maximizing a continuous function $y = f(x)$, a vector $x = (x_1, \ldots, x_n)$ for $x \in D$, $D \subset \mathbb{R}^n$, we are presented with an example algorithm:

1. Generate an initial population P_0 of some population size M with members $x^i \in$

$D, 1 \leqslant i \leqslant M$. Each member is evaluated with $f(x^i)$ as their fitness value.

2. Select $m < M$ best individuals judging by their fitness, rename them to $Q = (q_1, \ldots, q_m)$.
3. Generate the next population P_1 by:
   3.1. Crossover: randomly choose two members of Q: $q_a = (q_{a1}, \ldots, q_{an})$, $q_b = (q_{b1}, \ldots, q_{bn})$, then randomly choose an integer k such that $1 \leqslant k \leqslant n$ and form two children $q'_a = (q_{a1}, \ldots, q_{a(k-1)}, q_{bk} \ldots, q_{bn})$, $q'_b = (q_{b1}, \ldots, q_{b(k-1)}, q_{ak} \ldots, q_{an})$. Redo this step again if either child does not belong in D. When successful, place new members in P'_1 and continue until P'_1 has M members.
   3.2. Mutation: randomly choose s (a very small number compared to M) members in P'_1 and replace a random element in each of them with a random real number x in some interval. Retry this for each member until it is in D. After mutating each chosen member we have a new population P_1. Since s is small, the probability of an infinite loop during crossover is extremely small as there should always be enough "unaltered" members to be able to generate children that are within the domain.
4. A predetermined condition, such as reaching some population number K, designates when the algorithm should stop. If this condition is not met, go to step 2 and generate the next population in the same way.
5. If the stopping condition is met, select best member of P_1 as the final estimate.

[5] also specifies a method of maximizing (minimizing) a fuzzy set using a genetic algorithm. When maximizing $\overline{X}$, it proposes transforming the problem to a multi-objective one: maximizing the central value y, maximizing the area under the membership function to the right of y, and minimizing the area to the left of y.

We have not found any genetic algorithms in the literature which directly relate to computing the values of the powers of fuzzy stochastic matrices, nor any actual implementations of property checking for fuzzy Markov chains, making our project a potential large contribution to the field. However, there have been some other attempts at using genetic algorithms in fuzzy optimization problems which could be useful to us. [14] proposes a model for solving production problems as fuzzy quadratic programming problems using a fuzzy objective and fuzzy resource constraints. It uses a genetic algorithm with mutation along the weighted gradient, as well as human-computer interaction to determine preferred solutions. [11] is a fuzzy genetic algorithm which searches a state space of a software system model in order to verify reachability and detect deadlocks. This one is not so useful to us as it does not solve a problem that contains fuzzy numbers, but instead the method itself is using a fuzzy inference system to determine appropriate values of some meta-parameters of the algorithm such as population size. [12] focuses on nonlinear objective functions with fuzzy coefficients and fuzzy constraints without using α-cuts in their algorithm. [13] presents a methodology for converting from a fuzzy nonlinear (four-objective) programming problem of optimizing the scheduling performance to an equivalent nonlinear programming problem to be solved. Since finding the stationary distribution of a Markov chain could be encoded as a linear equation system, we will explore to what extend can we use [12] and [13] in our implementation.

Chapter 4

Research question

The goal of this project is formulated as the following question:
Research question: In what ways can we enable the evaluation of structural and probabilistic properties of fuzzy Markov chain-based systems?

We will answer it by extending the probabilistic model checker STORM [9] so that it can build and analyze fuzzy Markov chains. We will focus on a number of properties of fuzzy Markov chains, defined earlier, and explore whether they can be verified using STORM. We therefore formulate the following research sub-questions:

1. How can we represent fuzzy Markov chains in STORM model checker?
   1.1. How can we design the structures and functions in STORM model checker in such a way that it can store information about fuzzy Markov chains, and be able to verify their properties later on?
2. How can we implement a genetic algorithm based on restricted matrix multiplication (Chapter 5) to compute powers of the stochastic matrix?
   2.1. How can we ensure that the algorithm is able to verify whether a matrix is *feasible*? (Definition 5.1.3)
   2.2. How can we make the algorithm search for powers of the stochastic matrix within the domain of feasible values?
   2.3. How can we use this algorithm to verify properties of the Markov chain?
      2.3.1. How can we implement verification of whether the chain is absorbing (Definition 2.3.6) or regular (Definition 2.3.3)?
      2.3.2. How can we implement verification of reachability from state s_i to state s_j in k steps? ($\overline{p}_{ij}^{(k)}$, Definition 2.3.2)
3. How can we modify our algorithm to search for the limit of the stochastic matrix? ($\lim_{n\to\infty} \overline{P}^n = \overline{\Pi}$, Definition 2.3.4)
4. How can we optimize, in terms of performance and accuracy of the outcomes, the above algorithms (specifically the part from Point 2.2)?
   4.1. What should be the values of the meta-parameters of the genetic algorithm?
      4.1.1. What is the optimal initial population size?
      4.1.2. What is the optimal selection sample size?
      4.1.3. What is the optimal mutation sample size?

4.2. What should be the method of creating new generations?

4.2.1. What is the optimal selection method?

4.2.2. What is the optimal crossover method?

4.2.3. What is the optimal mutation method?

4.3. What should be the stopping condition?

4.3.1. What are the results for a stopping condition based on time?

4.3.2. What are the results for a stopping condition based on number of generations?

4.3.3. What are the results for a stopping condition based on estimated error?

The properties we want to check rely on being able to compute the powers of the stochastic matrix $\overline{P}^n$ for $n = 1, 2, 3, \ldots n$. We will show a method for computing them in Chapter 5. A genetic algorithm will be implemented based on [5], described in Chapter 3, however we will tailor it specifically for our context. We will also explore ways to generate valid fuzzy Markov chains given some state space size in order to obtain a test suite for our algorithm.

Chapter 5

Genetic algorithm for fuzzy numbers

The mathematical foundation for the objective functions to be minimized or maximized by the genetic algorithm is presented in this chapter. We will detail a restricted matrix multiplication method for calculating the stochastic matrix powers for a finite fuzzy Markov chain, $\overline{P}^n = (\overline{p}_{ij}^{(n)})$ for $n = 1, 2, 3, \ldots$ as described in references [4] and [7], which will enable the determination of fuzzy transition probabilities for transitioning from any state to another state in n steps. Additionally, we will present a method for calculating the stationary distribution of a fuzzy Markov chain, utilizing a similar approach.

Furthermore, we will propose a design for a genetic algorithm that will be utilized for the calculation of reachability and stationary distribution, based on these objective functions. The proposed algorithm will be based on the general outline specified in Chapter 3, however, its components including Initialization, Selection, Crossover, and Mutation will be specifically tailored to the problem at hand.

5.1 Restricted matrix multiplication

Given a fuzzy Markov chain $(S, s_0, \overline{P})$, we define the domain of fuzzy probability values in such a way that it conforms to the conditions explained in Chapter 2. We start with S' which is the set of all possible vectors of length r of real, crisp numbers ($r = |S|$) which form a valid probability distribution (all elements are non-negative and sum up to 1).

$$S' = \left\{ x = (x_1, \ldots, x_r) \mid x \in \mathbb{R}^r, \forall\, 1 \leqslant i \leqslant r, x_i \geqslant 0, \sum_{i=1}^{r} x_i = 1 \right\}.$$

To obtain the domain for one row $\overline{p}_i$ of the stochastic matrix, we then intersect S' with a Cartesian product of α-cuts of all fuzzy probabilities from this row.

Definition 5.1.1 (Feasible row domain)**.** The domain of row i, for α-cut α, $0 \leqslant \alpha \leqslant 1$, $1 \leqslant i \leqslant r$, is:

$$Dom_i[\alpha] = \left(\bigtimes_{j=1}^{r} \overline{p}_{ij}[\alpha] \right) \bigcap S'$$

To get the matrix domain, we then take the Cartesian product of all row domains.

Definition 5.1.2 (Feasible matrix domain)**.** The domain of a fuzzy matrix, for α-cut α, $0 \leqslant \alpha \leqslant 1$, is:

$$Dom[\alpha] = \bigtimes_{i=1}^{r} Dom_i[\alpha]$$

Definition 5.1.3 (Feasible matrix)**.** We say that every crisp matrix whose values conform to the above domain is *feasible* (with regards to the domain).

We know that in powers of a crisp transition matrix $P^n = (p_{ij}^{(n)})$, their elements are functions of elements in P: $p_{ij}^{(n)} = f_{ij}^{(n)}(p_{11}, \ldots, p_{rr})$. $\overline{p}_{ij}^{(n)}[\alpha]$ is then equal to the set of all values of $f_{ij}^{(n)}$ restricted only to the domain defined previously, $p_{11}, \ldots, p_{rr} \in Dom[\alpha]$:

$$\overline{p}_{ij}^{(n)}[\alpha] = f_{ij}^{(n)}(Dom[\alpha])$$

Note that, $f_{ij}^{(n)}(Dom[\alpha])$ itself does not necessarily have to conform to the domain. Although $f_{ij}^{(n)}$ is always a probability distribution by the nature of stochastic matrices, it is possible that no p_{ij} from any element of $Dom[\alpha]$ is equal to $f_{ij}^{(n)}$. However, that is not a problem, since the domain is only a constraint on P, and not P^n.

To get the desired probability value $\overline{p}_{ij}^{(n)}$, we will find endpoints of its α-cuts to later compose the full fuzzy number. We are able to do this because $\overline{p}_{ij}^{(n)}[\alpha]$ is a closed, bounded interval [4]. This is due to the fact that $f_{ij}^{(n)}$ is continuous and $Dom[\alpha]$ is connected, closed and bounded. We can therefore find the endpoints $\overline{p}_{ij}^{(n)}[\alpha] = [\overline{p}_{ijL}^{(n)}(\alpha), \overline{p}_{ijR}^{(n)}(\alpha)]$ using these equations:

Definition 5.1.4 (Reachability endpoints for fuzzy Markov chains)**.**

$$\overline{p}_{ijL}^{(n)}(\alpha) = \min\left\{ f_{ij}^{(n)}(p) \mid p \in Dom[\alpha] \right\},$$

$$\overline{p}_{ijR}^{(n)}(\alpha) = \max\left\{ f_{ij}^{(n)}(p) \mid p \in Dom[\alpha] \right\}$$

The above optimization problem is difficult and in general requires employing a directed search algorithm [4]. In this project, a genetic algorithm will be implemented to estimate these endpoints. Note that this only produces interval endpoints for given values of α, so the α-cuts of the desired result. We would therefore compute this for several values of α in order to compose the intervals into a fuzzy number. This way, one will be able to check basic properties of fuzzy Markov chains as they carry over from crisp Markov chains.

When it comes to finding the stationary distribution of the regular fuzzy Markov chains, or the rows of $\overline{P}^n \to \overline{\Pi}$, we first set $P = (p_{ij})$ for each $(p_{11}, \ldots, p_{rr}) \in Dom[\alpha]$ and get $P^n \to \Pi$. Then we have:

$$\Gamma(\alpha) = \{w \mid w \text{ } is \text{ } a \text{ } row \text{ } in \text{ } \Pi, P^n \to \Pi, P = (p_{ij}) \in (p_{11}, \ldots, p_{rr}) \in Dom[\alpha]\}$$

$\Gamma(\alpha)$ is an infinite set consisting of all possible, with regards to the domain, rows of Π. Now, to find the elements of $\overline{\Pi}$, let each row of it be $\overline{\pi} = (\overline{\pi}_1, \ldots, \overline{\pi}_n)$. Then, $\overline{\pi}_j[\alpha] = [\pi_{jL}(\alpha), \pi_{jR}(\alpha)], 1 \leqslant j \leqslant n$. To compute the α-cuts of $\overline{\pi}_j$ we use:

Definition 5.1.5 (Stationary distribution endpoints for fuzzy Markov chains)**.**

$$\pi_{jL}(\alpha) = \min\{w_j \mid w \in \Gamma(\alpha)\},$$

$$\pi_{jR}(\alpha) = \max\{w_j \mid w \in \Gamma(\alpha)\}$$

where w_j is the j-th component of the vector w.

Below we present an example of finding the stationary distribution of a fuzzy Markov chain that was created by introducing uncertainty to a crisp Markov chain from Example 2.2.1.

Example 5.1.1 (Stationary distribution: fuzzy Markov chain)**.** Suppose that, given a fuzzy Markov chain with the following fuzzy transition matrix

$$P = \begin{pmatrix} (0.5/0.6/0.7) & (0.3/0.4/0.5) \\ (0.1/0.2/0.3) & (0.7/0.8/0.9) \end{pmatrix},$$

we want to find the 0.5-cut of the first element of the Markov chain's stationary distribution. We can do it by searching through the domain of feasible matrices. That is, per Definition 5.1.3, the set of all crisp matrices that comply with both conditions:

1. All rows sum up to 1,
2. All values fall within the interval designated by 0.5-cut of the corresponding element from $\overline{P}$.

We can visualize the second constraint by showing 0.5-cuts of each element by converting $\overline{P}$ to an interval matrix:

$$P_{interval} = \begin{pmatrix} [0.55, 0.65] & [0.35, 0.45] \\ [0.15, 0.25] & [0.75, 0.85] \end{pmatrix},$$

In this space, we search for two matrices P, such that one minimizes and the other maximizes w_1, where w is a row in the stationary distribution of P. In our case, the matrix that minimizes w_1 is:

$$P_{min} = \begin{pmatrix} 0.55 & 0.45 \\ 0.15 & 0.85 \end{pmatrix},$$

which gives us $\pi_{1L}(0.5) = 0.25$ and for maximizing w_1:

$$P_{max} = \begin{pmatrix} 0.65 & 0.35 \\ 0.25 & 0.75 \end{pmatrix},$$

with $\pi_{1R}(0.5) = 0.41\overline{6}$. We therefore obtain our value: $\pi_1[0.5] = [0.25, 0.41\overline{6}]$

5.2 Algorithm design

To find fuzzy Markov chains' reachability and stationary distribution through genetic algorithms, we define fitness functions based on Equations 5.1.4 (reachability) and Equations 5.1.5 (stationary distribution). The search for the optimal solution, maximizing or minimizing the objective function, is conducted in the space of feasible matrices using adapted Initialization, Selection, Crossover, and Mutation methods from Chapter 3.

We design a structure to store the input and run the genetic algorithm. The input for initializing this structure is an $r \times r$ matrix of triangular fuzzy numbers, but may also include crisp 0s and 1s. The user is then able to run a genetic algorithm on this fuzzy matrix, with an option to either check the stationary distribution, or reachability. Additionally, they can check whether the matrix is regular or absorbing, but this functionality is separate from the genetic algorithm (with the exception that regularity check is automatically performed if the user wants to find the stationary distribution).

The genetic algorithm is performed $2 \cdot acc$ times, where acc is the number of alpha-cuts that will be checked (not including $\alpha = 1$, which is trivial), in order to calculate a single output triangular-shaped fuzzy number. This output number signifies a reachability value or a value from the stationary distribution vector. The algorithm must find the left and right endpoint of each α-cut for the given element in the matrix, and it must do so for each value of α in the range of 0 to 1 with a step size of $\frac{1}{acc}$.

5.2.1 Fitness functions

A member structure represents a single crisp matrix instance belonging to the feasible matrix domain, paired with a fitness value that our algorithm will either minimize or maximize. The fitness value will ultimately (as it is being continuously updated in the Selection stage) contain the approximation of either left or right endpoint of some α-cut of the triangular fuzzy number that is the solution of the given problem. For the reachability problem, a member consists of a 2D crisp matrix P' within the feasible domain, a pair of indices idx that denotes the "from" and "to" states, and the exponent n that represents the number of steps. In the case of the stationary distribution, a member only consists of the feasible matrix P' and index idx.

To update the fitness for reachability, we need a crisp matrix P', steps n and indices idx as input, we calculate the n-th power of the matrix, and return its idx-th element.

To update the fitness for stationary distribution, we calculate the limiting distribution of a given transition matrix and return the element signified by idx. The function checks whether the Markov chain is regular, and if so, calculates the stationary distribution by iterating over the product of the transition matrix and an initial probability vector until the change between successive iterations is below a certain threshold.

The two fitness functions mentioned above use standard, common algorithms for performing matrix multiplication and calculating stationary distribution, therefore their pseudocode can be found in the Appendix (Algorithms 8, 9).

5.2.2 Wrapper function

The algorithm begins by transforming the fuzzy matrix $\overline{P}$ into an interval matrix $intervalMatrix$ by replacing each fuzzy element with its corresponding α-cut, as well as storing the indices of the elements that were triangular fuzzy numbers and not crisp numbers inside a $intervalIndices$ vector. Other inputs for the algorithm are passed by the user and include the number of steps *steps*, the indexes of the element being calculated idx, and the direction $isMin$ (true or false). Optional meta-parameters include *populationSize*, *selectionSample*, *mutationRate*, and a stopping condition, for example *generations*. These are then passed to a wrapper function (Algorithm 1).

The algorithm iterates until a predetermined condition is met (in the case of the above, some set number of generation, but other options will be explored), at which point either the left or right endpoint, depending on $isMin$, of the α-cut for the element of the fuzzy stochastic matrix can be determined. Ultimately, the algorithm can be run for either an α-value given by the user, or using another helper function that takes in an acc value and runs the above wrapper for multiple α values. The acc parameter signifies the number of α-cuts the algorithm will calculate (not counting $\alpha = 1$, which is trivial and therefore always calculated by default), from 0 to 1 with equal distance between them. Therefore, the larger the value of acc, the more accurate the final result will be. However, even at just $acc = 1$, the a, b, and c values for the triangular-shaped fuzzy number will be obtained, since a and c values belong to the 0-cut, and b belongs to the 1-cut.

When the user requests running the genetic algorithm for multiple α-cuts by providing the acc parameter, the 1-cut matrix is calculated first, since it is used to check whether a matrix is regular in case the user requests to obtain a stationary distribution vector. It is trivial because it simply comes down to taking the peak (b value) of every element in the fuzzy matrix. The algorithms for checking whether the Markov chain is regular or absorbing are standard algorithms used for crisp Markov chains, just that they are inputted with this 1-cut of the fuzzy matrix. These functions, *isRegular* and *isAbsorbing*, are therefore

Algorithm 1 Genetic Algorithm endpoint wrapper

```
1: function FUZZYGAENDPOINT(intervalMatrix, intervalIndices, steps, idx, isMin,
   populationSize, selectionSample, mutationRate, generations)
2:     intervalRowIndices ← rowIndices(intervalIndices)      ▷ extract only the row
   indices from intervalIndices
3:     population ← initializePopulation(intervalMatrix, populationSize, steps, idx)
   ▷ population is a vector of Members
4:     for i ← 0 to generations do
5:         population ← selectPopulation(population, selectionSample, isMin)
6:         population ← crossPopulation(population, populationSize,
   intervalRowIndices)
7:         population ← mutatePopulation(population, mutationRate, intervalIndices,
   intervalRowIndices, intervalMatrix)
8:     end for
9:     population ← selectPopulation(population, selectionSample, isMin)
10:    if isMin then
11:        return population[0].fitness
12:    else
13:        return population[−1].fitness                ▷ Last element of population
14:    end if
15: end function
```

included in the Appendix (Algorithms 10, 13).

5.2.3 Initialization

First, we present a *randomCrisp* function (Algorithm 2) for generating a matrix consisting of values that are within their corresponding intervals from *intervalMatrix*, while at the same time its rows sum up to 1. We achieve this by first generating random values from each interval of the given row, and then either increasing or decreasing all elements by some correction value in order to make the sum equal to 1. This approach is guaranteed to provide a valid solution, provided that the input fuzzy matrix itself is valid (1-cuts of each row's elements sum up to 1).

The function can then be used to initialize the population, see Algorithm 3.

5.2.4 Selection

The algorithm then selects the n best individuals from the current population based on their fitness values, see Algorithm 4. The fitness value of a member is updated in this stage and contains the approximation of either left or right endpoint of some α-cut of the solution. Two fitness functions are possible: one for reachability, which calculates the power of a given matrix and returns its i-th element, and one for stationary distribution, which checks whether the Markov chain is regular and calculates the stationary distribution by iterating over the product of the transition matrix and an initial probability vector until the change between successive iterations is below a certain threshold.

5.2.5 Crossover

Crossover, see Algorithm 5, involves choosing two members of the current population and creating two new members by combining rows from each member. Namely, it takes a

Algorithm 2 Random crisp matrix generator

```
1: function randomCrisp(intervalMatrix)
2:     n ← intervalMatrix.size
3:     crispMatrix ← matrix(n)                          ▷ empty 2D matrix of size n × n
4:     for all i in intervalMatrix do                                  ▷ For each row i
5:         rangesLengths, randomValues ← [], []
6:         for all r in intervalMatrix[i] do           ▷ For each element from the row
7:             rangesLengths.append(r.upper − r.lower)
8:             randomValue ← randomDouble(r.lower, r, upper).
9:             randomValues.append(randomValue).
10:        end for
11:        difference ← 1.0 − sum(randomValues).
12:        if difference ≠ 0.0 then
13:            margins ← vector(), distributedCorrections ← vector()   ▷ Empty vectors
14:            if difference > 0.0 then
15:                for j ← 0 to n do
16:                    margins.append(intervalMatrix[i][j].upper− randomValues[j])
17:                end for
18:            else
19:                for j ← 0 to n do
20:                    margins.append(randomValues[j]− intervalMatrix[i][j].lower)
21:                end for
22:            end if
23:            marginSum ← sum(margins)
24:            for j ← 0 to n do
25:                distributedCorrections.append(difference·(margins[j]/marginSum))
26:                randomValues[j] ← randomValues[j] + distributedCorrections[j]
27:            end for
28:        end if
29:        crispMatrix.append(randomValues)
30:    end for
31:    return crispMatrix
32: end function
```

Algorithm 3 Initialize population

```
1: function initializePopulation(intervalMatrix, populationSize, steps, idx)
2:     population ← vector()                                          ▷ empty vector
3:     for i ← 0 to populationSize do:
4:         r ← randomCrisp(intervalMatrix)
5:         m ← Member(r, steps, idx)
6:         population.append(m)
7:     end for
8:     return population
9: end function
```

random amount of random rows from the input matrix (disregarding the ones that do not correspond to fuzzy elements, as these will be identical for every member), and these rows will be inherited from the first parent, while the remaining ones will be inherited from the

Algorithm 4 Select population

```
1: function selectPopulation(population, selectionSample, isMin)
2:     for all member in population do
3:         member.updateFitness()
4:     end for
5:     sort(population)          ▷ Comparison for members implemented based on fitness
6:     n ← population.size · selectionSample
7:     selected ← vector()
8:     if isMin then
9:         selected.extend(population[0], population[n − 1])   ▷ extend selected with the first n elements
10:    else
11:        selected.extend(population[−n], population[−1])   ▷ extend selected with the last n elements
12:    end if
13:    return selected
14: end function
```

Algorithm 5 Cross population

```
1: function crossPopulation(population, populationSize, intervalRowIndices)
2:     crossed ← vector()                                    ▷ empty vector
3:     while crossed.size < populationSize do
4:         q1, q2 ← randomSample(population, 2)       ▷ randomSample(arr, n) selects n unique elements from an array
5:         children ← crossParents(population[q1], population[q2], intervalRowIndices)
6:         function crossParents(q1, q2, intervalRowIndices)
7:             randRows ← randAmountOfRandRows(intervalRowIndices)
8:             c1 ← Member(q1)                         ▷ Copy matrix, n, idx from q1
9:             c2 ← Member(q2)                         ▷ Copy matrix, n, idx from q2
10:            for all randomRow in randRows do
11:                c1.matrix[randomRow] ← q2.matrix[randomRow]
12:                c2.matrix[randomRow] ← q1.matrix[randomRow]
13:            end for
14:            return c1, c2
15:        end function
16:        crossed.extend(children)
17:    end while
18:    return crossed
19: end function
```

other parent (and vice versa for the second child).

Example 5.2.1 (Crossover operation)**.** Consider this pair of crisp matrices, randomly selected from a population:

$$q1 = \begin{pmatrix} 0.47 & 0.25 & 0.28 \\ 0.23 & 0.58 & 0.19 \\ 0.07 & 0.48 & 0.45 \end{pmatrix}, \; q2 = \begin{pmatrix} 0.56 & 0.04 & 0.4 \\ 0.27 & 0.45 & 0.28 \\ 0.2 & 0.38 & 0.42 \end{pmatrix}$$

Algorithm 6 Mutate population

```
1: function MUTATEPOPULATION(population, mutRate, intervalIndices, intervalRowIndices, intervalMatrix)
2:     amount ← mutRate · population.size
3:     toMutate ← randomSample(population, amount)    ▷ Assume randomSample gives references to Member objects
4:     for all m in toMutate do
5:         mutMember(m, intervalIndices, intervalRowIndices, intervalMatrix)
6:     end for
7:     return population
8: end function
```

The matrices were picked from a population in a genetic algorithm that is analyzing the following interval matrix:

$$intervalMatrix = \begin{pmatrix} [0.46, 0.65] & [0.03, 0.96] & [0.27, 0.62] \\ [0.22, 0.73] & [0.36, 0.59] & [0.18, 0.76] \\ [0.06, 0.8] & [0.21, 0.54] & [0.3, 0.96] \end{pmatrix}$$

Then, a random amount of random rows is selected. That is, an integer is picked from a closed interval $[0, n-1]$, where n is the number of rows. In this example, 2 was picked, and therefore two rows were randomly selected: 0 and 2 (indexed from 0). This means that child $c1$ will inherit rows 0 and 2 from $q2$, and row 1 from $q1$. Child $c2$ will be the opposite: rows 0 and 2 from $q1$, and row 1 from $q2$. The children will therefore look like this:

$$c1 = \begin{pmatrix} 0.56 & 0.04 & 0.4 \\ 0.23 & 0.58 & 0.19 \\ 0.2 & 0.38 & 0.42 \end{pmatrix} c2 = \begin{pmatrix} 0.47 & 0.25 & 0.28 \\ 0.27 & 0.45 & 0.28 \\ 0.07 & 0.48 & 0.45 \end{pmatrix},$$

They are inserted into a new population and this process repeats until this new population's size reaches *populationSize*.

5.2.6 Mutation

Mutation, see Algorithm 6, involves randomly choosing a small number (*mutRate*) of members from the population and randomly altering a row of interval elements in each member while still maintaining the constraints imposed by P_I and preserving the sum of the row. We achieve this by altering only two elements, and adding some random number to one of them while subtracting the same number from the other. This random number is chosen from such a range that it will not violate constraints of neither element's corresponding interval.

One of the inputs for Algorithm 7, *intervalIndices*, is a vector of x/y pairs, and it lists indices of all elements which are non-zero length intervals in the *intervalMatrix*. A similar input vector is *intervalRowIndices*, which contains just the indexes of rows that have such elements in them. Therefore, we randomly select one row using *intervalRowIndices*, and iterate over *intervalIndices* to extract indices of desirable elements from this random row.

Later on, when we have chosen the elements that we want to mutate (and assigned values to $mutV1$, $mutV2$), we compute the minimum and maximum number that we can add to one of them while subtracting from the other. We do this by assigning the corresponding interval ranges to $mutR1$, $mutR2$ and calculating how much we can shift $mutV1$ and $mutV2$ such that they stay within the intervals $mutR1$ and $mutR2$, respectively.

Algorithm 7 Mutate member

```
1: function MUTMEMBER(member, intervalIndices, intervalRowIndices, intervalMatrix)
2:     row ← random(intervalRowIndices)    ▷ Random element of intervalRowIndices vector
3:     elementsInRandomRow ← vector()
4:     for all intervalIndex in intervalIndices do
5:         if intervalIndex.x = row then
6:             elementsInRandomRow.append(intervalIndex.y)
7:         end if
8:     end for
9:     toMutate1 ← random(elementsInRandomRow)
10:    toMutate2 ← random(elementsInRandomRow)    ▷ toMutate1 and toMutate2 should be different
11:    mutMatrix ← member.matrix
12:    mutV1 ← mutMatrix[row][toMutate1]
13:    mutV2 ← mutMatrix[row][toMutate2]
14:    mutR1 ← intervalMatrix[row][toMutate1]
15:    mutR2 ← intervalMatrix[row][toMutate2]
16:    feasibleMR ← getFeasibleMR(mutV1, mutV2, mutR1, mutR2)
17:    function GETFEASIBLEMR(mutV1, mutV2, mutR1, mutR2)
18:        left ← min(mutV1 − mutR1.lower, mutR2.upper − mutV2)
19:        right ← min(mutR1.upper − mutV1, mutV2 − mutR2.lower)
20:        return Interval(−left, right)
21:    end function
22:    mutOffset ← random(feasibleMR.lower, feasibleMR.upper)
23:    mutMatrix[row][toMutate1] = mutMatrix[row][toMutate1] + mutOffset
24:    mutMatrix[row][toMutate2] = mutMatrix[row][toMutate2] − mutOffset
25:    member.matrix = mutMatrix
26:    Return member
27: end function
```

Example 5.2.2 (Mutation operation)**.** We present mutation executed on one of the new members produced in Example 5.2.1, namely:

$$mutMatrix = \begin{pmatrix} 0.47 & 0.25 & 0.28 \\ 0.27 & 0.45 & 0.28 \\ 0.07 & 0.48 & 0.45 \end{pmatrix}$$

which is an instance of the following interval matrix:

$$intervalMatrix = \begin{pmatrix} [0.46, 0.65] & [0.03, 0.96] & [0.27, 0.62] \\ [0.22, 0.73] & [0.36, 0.59] & [0.18, 0.76] \\ [0.06, 0.8] & [0.21, 0.54] & [0.3, 0.96] \end{pmatrix}$$

All elements in this matrix are non-zero length intervals, therefore we choose a random row out of all rows, and random two elements out of this row. Assume we selected row 1, and elements 1 and 2 out of this row (all indexed from 0). Corresponding elements from *mutMatrix* are assigned to $mutV1$ and $mutV2$ respectively:

$$mutV1 = 0.45, \; mutV2 = 0.28$$

And elements from $intervalMatrix$ are assigned to $mutR1$ and $mutR2$:

$$mutR1 = [0.36, 0.59],\ mutR2 = [0.18, 0.76]$$

We then calculate bounds of the feasible mutation range:

$$left = \min(mutV1 - mutR1_L, mutR2_R - mutV2) = \min(0.09, 0.48) = 0.09$$

$$right = \min(mutR1_R - mutV1, mutV2 - mutR2_L) = \min(0.14, 0.1) = 0.1$$

We randomly select a number in the range $[-left, right]$, let us assume it is 0.09. We therefore add 0.09 to the element 1 and subtract 0.09 from element 2 (both from row 1) in the crisp matrix, producing our final mutated matrix:

$$mutMatrix = \begin{pmatrix} 0.47 & 0.25 & 0.28 \\ 0.27 & 0.54 & 0.19 \\ 0.07 & 0.48 & 0.45 \end{pmatrix}$$

Note that both elements still fall within their respective interval ranges and the sum of all elements in row 2 is still 1.

Chapter 6

Implementation

The code for this project has been published at GitHub [2].

6.1 TriangularFuzzyNumber class

The first phase of implementation is focused on addressing Research Question 1, specifically the representation of fuzzy Markov chains within the STORM model checker. The objective of this phase is to design the structures and functions in the STORM model checker that will enable the storage and verification of properties of fuzzy Markov chains.

The first step in this process was the implementation of a number type for triangular fuzzy numbers, as these will serve as the value type for the elements of the stochastic matrix. To accomplish this, a *TriangularFuzzyNumber* class was created in STORM, which includes member variables for *leftBound*, *peak*, and *rightBound* that store the a, b, and c values respectively of a triangular fuzzy number.

In addition, the already existing *SparseMatrix* class was selected as the class that will be utilized as the stochastic matrix in fuzzy Markov chains. To integrate the *TriangularFuzzyNumber* class, the class was added as a class template for both *SparseMatrix* and *SparseMatrixBuilder* (a class used to construct a sparse matrix by adding values incrementally).

To ensure proper compilation in STORM, several arithmetic operators were overloaded for the *TriangularFuzzyNumber* class, utilizing placeholder exception throws for non-implemented functionality. A notable exception was the $+=$ operator, which was required for the proper creation of the model, as it checks whether all rows of the matrix sum to one. In this instance, the operator was defined as summing the peaks of the triangular fuzzy numbers. Furthermore, comparison operators such as $>$ and $<$ were implemented to compare the peaks of the triangular fuzzy numbers, in line with the operator used in reference [4]. A hash value function and output stream $<<$ operator were also implemented to facilitate successful compilation.

As a result of these changes, the user is now able to create a sparse matrix with triangular fuzzy numbers as a template parameter, add values to it using the *SparseMatrixBuilder*, and create models such as a *StandardRewardModel* using the stochastic matrix as a component. These modifications to the data structures pave the way for the implementation of property-checking for fuzzy Markov chains in the subsequent implementation phases.

6.2 FuzzyAnalysisResult class

The *FuzzyAnalysisResult* class was implemented based on the design outlined in Chapter 5 to represent a result of a fuzzy analysis. It has a nested class called *Member*, which represents a member of a population in a genetic algorithm. They both include getter and setter methods, implementations of the algorithms from Chapter 5, as well as some simple helper functions that we provide in the next paragraph. Additionaly, *Member* has an overloaded $<$ operator so that the population can be sorted by their *fitness* atribute in Algorithm 4, and a $==$ operator that compares if the underlying matrices are the same in Algorithm 5.

The *FuzzyAnalysisResult* class has the following helper functions which were not explicitly written out in Chapter 5 nor the Appendix:

1. *fuzzyGA*
   - Input: pair of integers *idx*, integer *acc*, integer *populationSize*. Optional arguments: integer *steps*, floating-point number *threshold*, floating-point number *selectionSample*, floating-point number *mutationRate*, enum *stopping*, boolean *reachability*
   - Behaviour: serves as a wrapper for Algorithm 1 (*fuzzyGA*, which itself is a wrapper for the genetic algorithm). It runs the algorithm for both endpoints for each α-cut, for $\alpha = 0$, $\alpha = 1$, and $acc - 1$ evenly spaced α-cuts in-between. It calculates the values of a triangular-shaped fuzzy number placed at index *idx* in some structure. Possible structures include either the stochastic matrix raised to the power of *steps* (in case of reachability), or in the stationary distribution vector. It runs for reachability by default, otherwise the user has to provide *reachability=false* and some value of *steps*. Three different termination conditions are possible, the choice of which is determined by an enumerated type parameter *stopping*. By default it is based on the number of generations, other options are the number of milliseconds, or the average difference of scores between successive generations (as percentage). *threshold* parameter then determines, depending on the value of *stopping*, the exact number of generations, milliseconds, or percentage. If values of the meta-parameters are not provided, the recommended ones will be used based on the size of the fuzzy matrix (explained in Chapter 7.3). Returns the triangular-shaped fuzzy number as a vector of (x, y) pairs, where y is the membership value of x.
   - Output: vector of floating-point number pairs *result*
2. *getIntervalMatrix*
   - Input: floating-point number *alpha*
   - Behaviour: returns an interval matrix constructed by taking α-cuts of each element of the *matrix* (attribute of the class)
   - Output: 2D Interval matrix *intervalMatrix*, vector of index pairs of (non-zero length) intervals in the matrix *intervalIndices*
3. *getCrispMatrix*
   - Input: none
   - Behaviour: returns a crisp matrix constructed by taking peaks (1-cuts) of each triangular fuzzy element of the *matrix* (attribute of the class)

- Output: 2D crisp (*double*) matrix *crispMatrix*

4. *getAlphaCut*

   - Input: triangular fuzzy number t, floating-point number *alpha*
   - Behaviour: returns the α-cut of t
   - Output: Interval

Member class also contains the following helper function, called in Algorithm 4:

updateFitness

- Input: none
- Behaviour: assign the output of either *matrixMul* or *stationaryDistribution*, depending on the value of *this->reachability*, to *this->fitness*
- Output: none

The fuzzyGAEndpoint and fuzzyGA functions serve as wrappers of the genetic algorithm and are the primary functions that are to be called by the user directly.

The Interval type is built-in in STORM, and defined in the file *RationalFunctionAdapter.h* as *carl::Interval<double>*, coming from the open source C++ library *CArL* [1].

Chapter 7

Evaluation

7.1 Testing phase

The first step in testing the algorithm was to implement unit tests for the deterministic utility functions. These functions form the backbone of the algorithm and were crucial to its performance. We implemented a series of unit tests, including *testMatrixMul*, *testGetIntervalMatrix*, *testGetAlphaCut*, *testGetFeasibleMutationRange*, *testIsRegular*, and *testIsAbsorbing*, to confirm that these functions were working as intended and producing the expected results. The unit tests were run multiple times with different inputs and the results were carefully analyzed to identify any issues or bugs.

After the deterministic functions were confirmed to be working correctly, we moved on to test the parts of the algorithm that involve randomness. The genetic algorithm uses a genetic search strategy and includes elements of randomness such as mutation and crossover. It was important to ensure that these elements of randomness were not producing biased or unexpected results. We did this by observing if the whole range of possible outcomes could be generated by running the algorithm multiple times with different inputs and different random seed.

7.2 Correctness

We verified the results of the genetic algorithm by comparing them to those produced by an exact algorithm using the same input data. This provided a way to confirm that the genetic algorithm is producing accurate and reliable results. The exact algorithm checked every possible combination of values, up to a given precision, that complies with the constraints and returned one with the best fitness value. Fitness functions themselves (i.e. calculating powers of a matrix and the stationary distribution) remain the same as in the genetic algorithm. Of course, because the intervals produced by taking α-cuts are continuous, it is impossible to find the result with infinite precision. Therefore, we used a step of 0.005 when traversing the interval range, and an identical value of tolerance when verifying whether the rows sum up to 1. This way, we were able to obtain exact results up to two decimal places, although floating-point precision and rounding errors could slightly impact the final accuracy.

Another limitation was that the time needed to obtain exact results exceeded reasonable boundaries for matrices bigger than 3×3. That is, for a 4×4 matrix, it reached over 20 minutes without completion. However, we have no reason to suspect that the validity wouldn't scale to larger input sizes. The pseudocode for the exact algorithm is included in the Appendix (Algorithm 15).

(0.1186/0.6295/0.7851) (0.1681/0.3705/0.9169)
(0.0836/0.0847/0.3535) (0.4557/0.9153/0.999)
Matrix 1

(0.05/0.5299/0.9781) (0.2347/0.4701/0.8581)
(0.0008/0.0017/0.4762) (0.4801/0.9983/0.9992)
Matrix 2

(0.6009/0.6234/0.7413) (0.1817/0.3766/0.3955)
(0.1875/0.6253/0.9766) (0.1219/0.3747/0.4586)
Matrix 3

(0.2652/0.3345/0.9675) (0.2098/0.6655/0.7439)
(0.3292/0.7275/0.8695) (0.1466/0.2725/0.9207)
Matrix 4

(0.1619/0.7688/0.8779) (0.0778/0.2312/0.6171)
(0.0971/0.3756/0.5179) (0.0622/0.6244/0.9049)
Matrix 5

FIGURE 7.1: Input matrices for the algorithm comparison

Matrix	Exact, left	Genetic, left	Exact, right	Genetic, right	Difference, larger	Exact, time [s]
1	0.4304	0.4281	0.9251	0.9134	0.0117	45.9
2	0.3423	0.3464	0.9961	0.9990	0.0041	82.1
3	0.2236	0.2256	0.4224	0.4223	0.0029	5.0
4	0.1953	0.1972	0.6920	0.6905	0.0019	88.5
5	0.2291	0.2285	0.8719	0.8671	0.0048	79.9

TABLE 7.1: Comparison of results between the exact and genetic algorithm

Using the exact algorithm as a reference, and observing that values of the genetic algorithm approach the reference results, allowed us to confidently conclude that the genetic algorithm is an accurate solution for checking properties of fuzzy Markov chains.

To illustrate this, in Table 7.1 we present a result comparison, with measurements taken using five 2×2 input matrices (Figure 7.1). The problem which was solved in these tests was reachability from state 2 to itself in 3 steps. The exact algorithm took 60.3 seconds on average to arrive at the results, while the genetic algorithm's time was fixed to 40 seconds (20 seconds per endpoint). The values in input matrices as well as results presented here are rounded to 4 decimal places. As we can see, the results in this test differed by at most 0.0117, proving the genetic algorithm's validity.

Overall, the combination of unit tests and comparison to an exact algorithm helped us to thoroughly test and validate the genetic algorithm for checking properties of fuzzy Markov chains. Through this process, we were able to identify and fix any issues or bugs and develop a high-performing algorithm that could be used in real-world applications.

7.3 Optimization phase

The first goal of the optimization phase of the project was to find optimal values of meta-parameters of the genetic algorithm, namely:

1. Initial population size
2. Selection sample size
3. Mutation sample size

Ideally, the end-user would be able to not provide any of the above parameters but nonetheless get a desired solution in an acceptable amount of time. This will be achievable if we are able to design an algorithm that approximates each of the parameters based on any input size. For the future, the number of generations could also be included in the calculation, however this argument is optional (since the user can choose a different stopping condition), therefore formula that includes this value would have to be approximated separately

and only used on condition that the number of generation is specified. Otherwise, with stopping condition such as amount of time per α-cut, it would produce different results for different machines. α value is also a factor that could be included in the calculations, as our observations showed more deviation from optimal results for lower values of α, but we chose sticking with the same parameters across all α values for the sake of simplicity and because of time constraints.

The further goals were as follows:

1. Reflect on the chosen selection, crossover and mutation methods
2. Compare and contrast the stopping conditions

For the first point, we will discuss pros and cons of the current design and implementation of aforementioned parts of the genetic algorithm, consider whether they can be improved, and if so, in what way. For the second point, we will compare the currently implemented stopping conditions, and discuss which is the best one depending of the particular use case.

7.3.1 Meta-parameters optimization

Grid search was chosen as a method of tuning the parameters for particular input sizes. Namely, an exhaustive search was performed for every combination of population size, selection rate, and mutation rate, each with predefined range and step size.

The grid search was performed for three input matrix sizes: 10×10, 30×30, and 50×50. These numbers were chosen as they could have been analyzed in reasonable time.

When running the benchmark for a particular input size, three different random fuzzy matrices were generated and benchmarked independently. To achieve this matrix generation, an algorithm is performed: selection of a uniformly random floating-point number between 0 and 1 for each peak (b value) of a fuzzy value in a single row, following by row normalization (multiplying every value times $1/rowSum$), and finally selecting a and c values of the fuzzy number as a uniformly random floating-point number between 0 and b, or b and 1, respectively. This is repeated for every row in the matrix.

The analyzed ranges of parameters were chosen as ones that seemed to provide best results in initial, manual and non-formal testing, namely:

1. Population size: 50, 100, 150, 200
2. Selection rate: 0.05, 0.1, 0.15, 0.2
3. Mutation rate: 0.05, 0.1, 0.15, 0.2

Each combination of the above (64 in total) was run 5 times in a given benchmark. Extreme values were discarded, and three medium values were averaged out to produce the benchmark value. Value of α was set to 0.5 for all benchmarks as an approximation, as we assumed that optimizing for 0.5 would on average give best results in practice (i.e. running the algorithm across multiple α-cuts). Number of steps for reachability was set arbitrarily to 3, simply to have a consistent computational complexity of computing the fitness function (matrix multiplication) for a given matrix size.

Since we are considering only the input size in our calculations, our measurements were set to last a relatively long amount of time. The reason for this is that in our optimization we want to prioritize the type of user for whom the accuracy of the result is more important than the amount of time it takes. We assume that the user who cares more about getting an acceptable result as fast as possible should still benefit from our optimization. Therefore, each measurement was taken for a stopping condition based on the following amount of time:

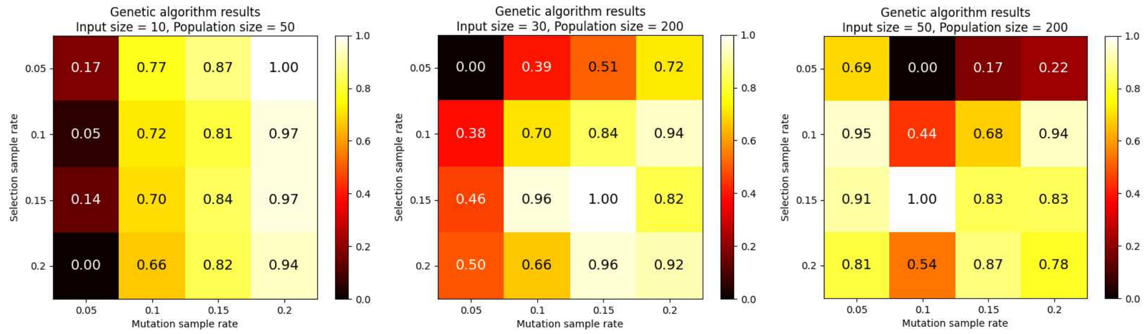

FIGURE 7.2: Results for input sizes 10, 30, 50

1. 10×10 matrix: 25 seconds
2. 30×30 matrix: 49 seconds
3. 50×50 matrix: 81 seconds

The tests were performed on a machine with Intel® Core™ i5-1035G1 CPU with max frequency of 3.6 GHz and 8 GB of RAM. However, since the system ran on a Docker container, only 4 GB of RAM were allocated to it, with additional 8 GB of swap space. Naturally, different machines might need more or less time to compute the same number of generations, but we deemed the chosen time intervals a good compromise between what is able to produce test results in a reasonable amount of time and what is unlikely to make a difference beyond these values.

The heatmaps on 7.2 show how the results compare for the entire analyzed range of selection rates and mutation rates. The population size that is shown is the one whose best result was the best overall. The values inside the squares symbolize how good the result was relative to the best result found, normalized from 0 to 1, where 0 is the worst and 1 is the best result for the given input size and population size.

Based on the results we can draw the following blueprint for tuning the parameters:

1. Population size:
   - 50 for input sizes < 30
   - 200 for input sizes $\geqslant 30$
2. Selection rate:
   - 0.05 for input sizes < 30
   - 0.15 for input sizes $\geqslant 30$
3. Mutation rate:
   - 0.2 for input sizes < 30
   - 0.15 for input sizes $\geqslant 30$ and < 50
   - 0.1 for input sizes $\geqslant 50$

More measurements are needed, perhaps with less accuracy thanks to using the above results as a starting point, to remain more accurate on how the parameters should behave for input sizes that were not measured.

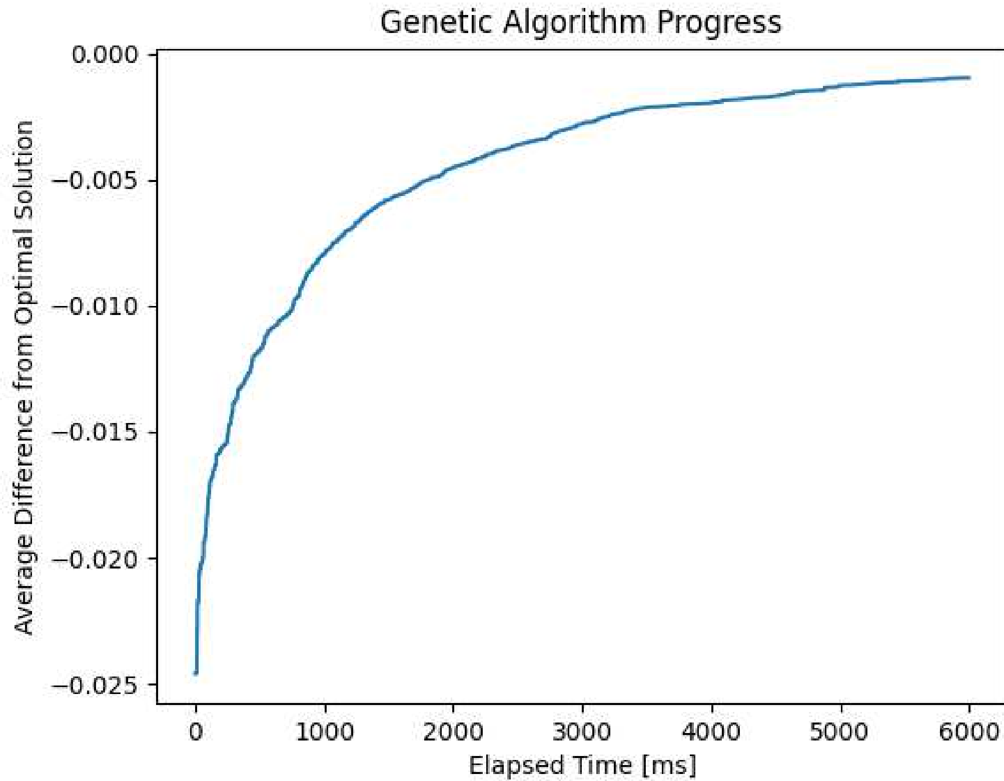

FIGURE 7.3: Difference from the assumed optimal solution against time for a 10×10 matrix

After applying these newly-found parameter values, we measured how quickly the algorithm is able to converge in the same reachability problem that we used in our benchmarks. We logged the results of the algorithm until no better solutions were found for over 10 thousand generations, and assumed the final value to be the optimal solution. We present on Figure 7.3 that the algorithm reaches within 0.01 of this assumed optimum in less than a second for a 10×10 matrix. See Chapter 7.4 for more in-depth performance analysis and comparison with other input sizes.

7.3.2 Reflection on parts of the algorithm

Selection

The selection part of the genetic algorithm is crucial for the algorithm's success as it determines which members of the population will be used to create the next generation. The current design and implementation of the selection part, as presented in Algorithm 4, have both advantages and disadvantages.

The selection process favors individuals with better fitness values, which should increase the chances of producing better solutions in the next generation. It is based on a simple and easy-to-implement algorithm, which makes it efficient and scalable. However, since it doesn't consider diversity among the selected individuals, it may lead to loss of genetic diversity and premature convergence of the algorithm, i.e., it may get stuck in a local optima. We didn't encounter this problem when testing accuracy with smaller inputs, but

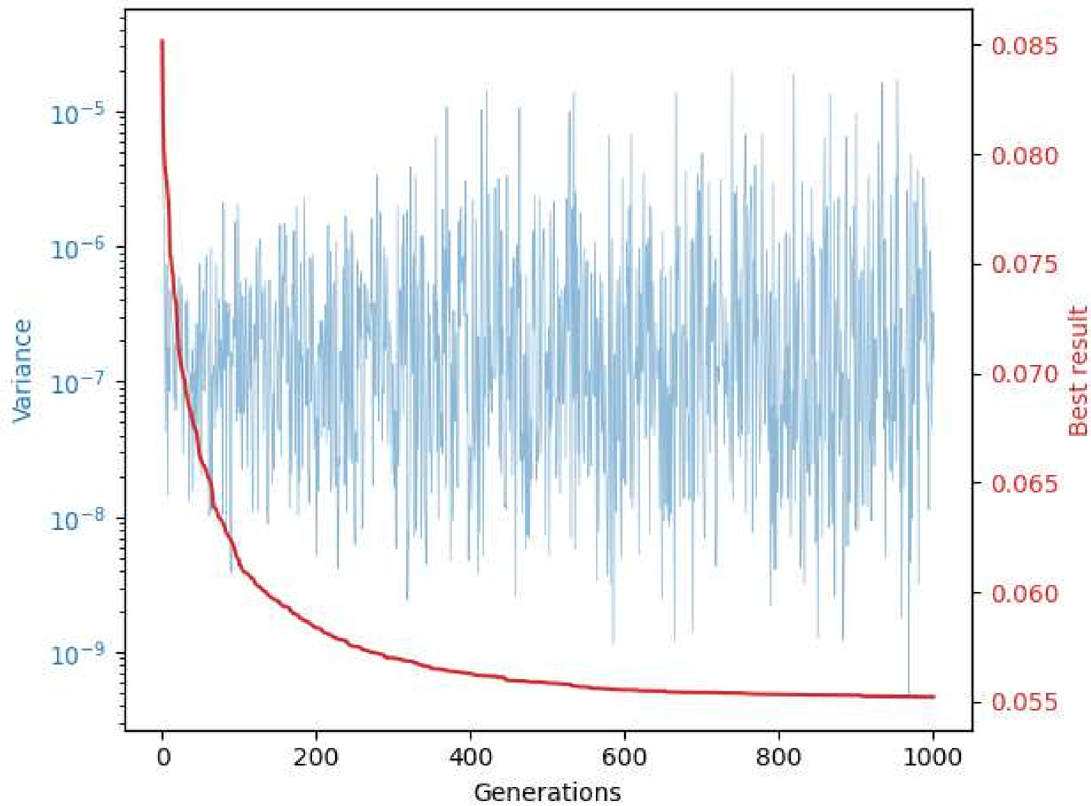

FIGURE 7.4: Best result and variance of the population across 1000 generations

it is possible that this issue appears as the search space becomes larger.

To explore how quickly the algorithm converges and how it relates to genetic diversity, we performed experiments where we ran the genetic algorithm multiple times on the same problem and on the same 10×10 matrix and measured the average fitness value of the best individual in each generation as well as the variance between members. We then plotted the results on a graph (Figure 7.4) with the number of generations on the x-axis and both the average fitness value and the variance on the y-axis.

We observed that the variance remains high despite the converging value of fitness of the best member of the population. Therefore, the algorithm is likely converging towards the true best result, and not towards some local optimum because of low genetic variation.

Crossover

The crossover part of the genetic algorithm, presented in Algorithm 5, involves selecting two members of the current population (parents) and combining their rows to create two new members (children). This way, by copying the entire rows, the crossover operation avoids the problem of having to modify row elements in order to fit the constraints (it is only later in the mutation stage where such complexities arise). This is the main advantage of this approach, making it exceptionally simple to comprehend and implement, as well as scalable and efficient.

However, one might consider whether it is in fact the case that when using this method, fitness of the child correlates to the fitness of its parents. This question may arise when we realize that the fitness functions we are using can be quite chaotic. That is, introducing

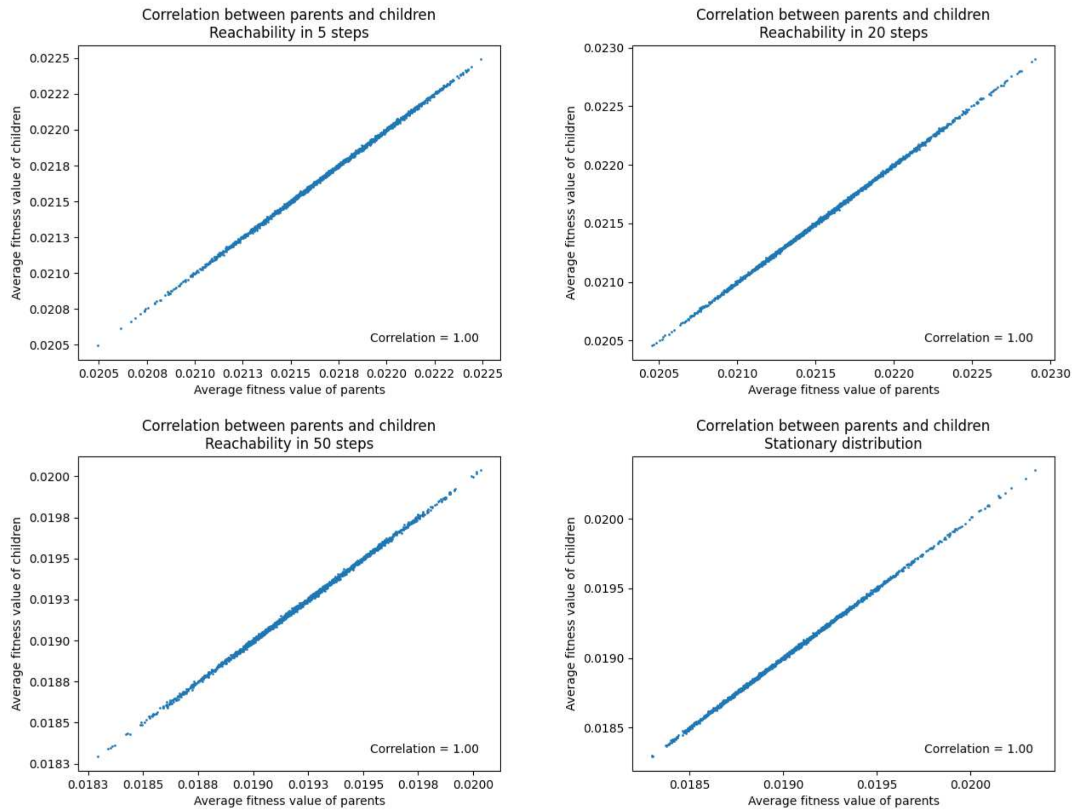

FIGURE 7.5: Correlation between parents and children

such large variation as swapping multiple rows in a parent matrix can possibly make its fitness value not correlated with the fitness value of its child. To ensure that this correlation exists, we performed an experiment in which we generated 64 members (from which we create every possible pair, $\frac{64 \cdot 63}{2} = 2016$ in total) and mapped the average fitness value of the parents against the children's average fitness value. We tested a 50×50 matrix on the reachability problem with 5, 20 and 50 steps, as well as on the stationary distribution problem.

As we can see on Figure 7.5, no matter the type of the problem, the average fitness of children correlated perfectly with the average fitness of their parents. This makes our sole concern resolved and further convinces us that the chosen crossover method is appropriate.

Perfect correlation between parent fitness and child fitness does not necessarily mean that we are limiting genetic variability. It simply means that the crossover operation is effective in passing on the best traits from the parents to the offspring. It does not imply that the offspring will be identical to the parents, as the crossover operation introduces new combinations of genetic information that can produce offspring with new and better traits. Therefore, a high correlation between parent fitness and child fitness is desirable.

Mutation

The current design and implementation of the mutation part of the genetic algorithm, presented in Algorithm 6 and 7, have several pros and cons. The mutation operation introduces genetic diversity in the population and can help the algorithm escape from local optima, while at the same time it is designed to maintain the constraints imposed by the interval matrix, which is essential for finding feasible solutions.

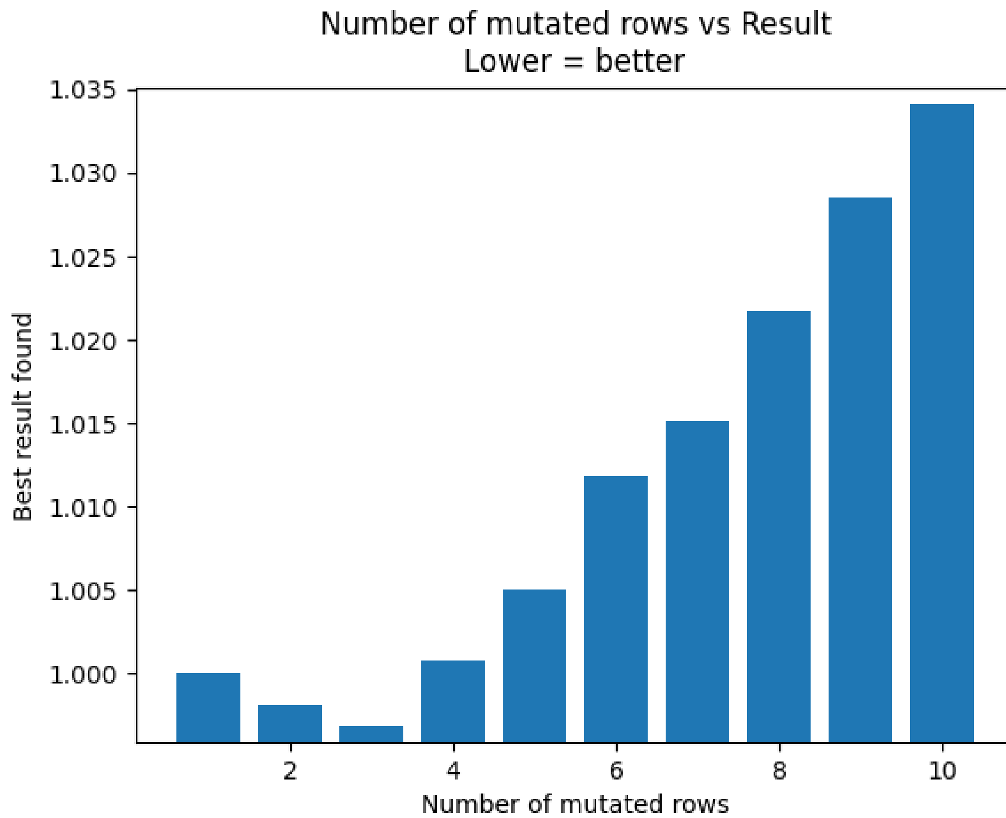

Figure 7.6: Impact of the number of mutated rows on the results

One potential drawback is that the mutation rate is fixed and does not adapt to the progress of the algorithm. This can result in the mutation operation being applied too frequently or not enough, which can hinder the algorithm's performance. One approach is to adapt the mutation rate based on the progress of the algorithm. For example, the mutation rate could be increased if the algorithm gets stuck in local optima or decreased if the algorithm is exploring the search space well.

Next consideration is that the mutation operation only alters one row of the member's matrix. It is possible that it limits the search space and slows down the convergence of the algorithm, especially for larger input size. In an effort to enhance the efficiency and convergence rate of the algorithm, we proposed a modification to the mutation operation by allowing it to alter multiple rows of a member's matrix simultaneously. We examined the effectiveness of this approach by comparing the results of modifying $1, 2, \ldots 10$ rows at a time for when running the genetic algorithm for 25 seconds on a reachability problem with 10x10 input matrix.

As depicted in Figure 7.6, we normalized the results using the original algorithm as a baseline. The data reveals that the optimal number of rows to modify is 3, yielding a 0.3% improvement in average performance. Consequently, we updated the code to enable users to input the number of mutated rows as a parameter. Further research is necessary to determine the preferred number of rows to modify for different input sizes and problem types. However, our current findings suggest that a mutation rate of 0.3 times the input size holds the most promise.

Finally, instead of altering only two elements in the row, the mutation operation can be modified to alter multiple elements randomly. However, one would need to develop

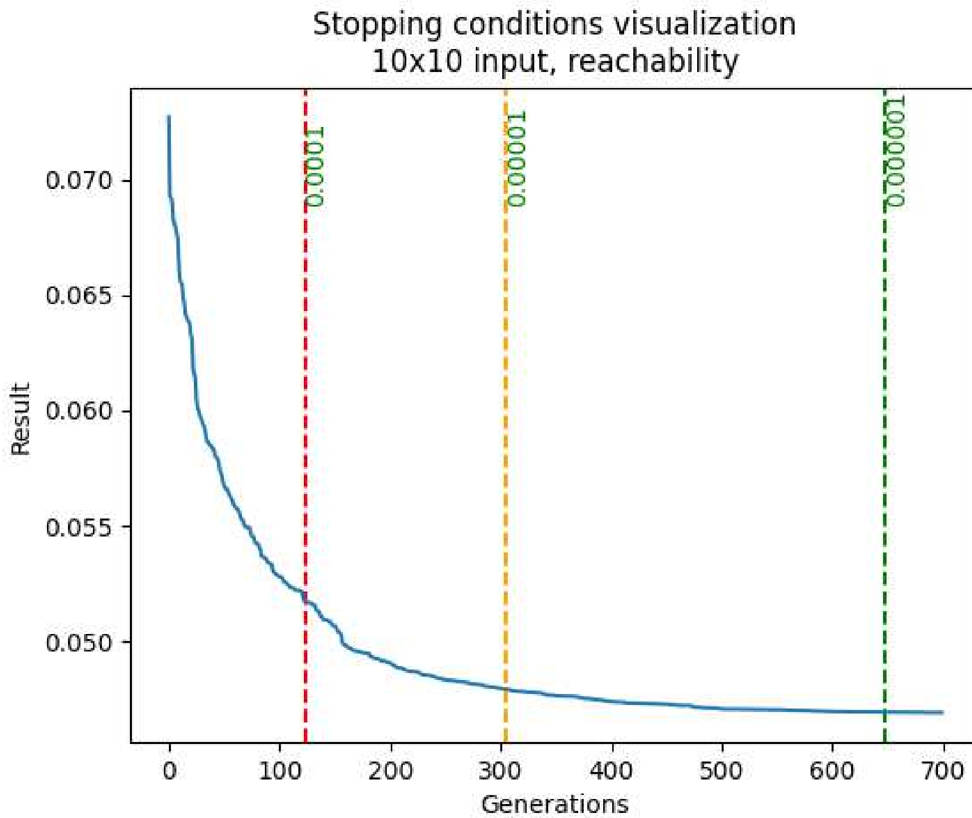

FIGURE 7.7: Visualization of different time-based stopping thresholds

another way to perform such mutation and preserve both constraints (that the sum of the row is preserved and resulting values still fall within their corresponding intervals), since the current method cannot be trivially scaled to more than two elements. If achieved, this could potentially help the algorithm explore the search space more efficiently and reduce the likelihood of getting stuck in local optima.

7.3.3 Comparison of stopping conditions

Stopping conditions determine when the genetic algorithms terminates and outputs the best candidate solution found so far. The choice of stopping condition can have a significant impact on the performance and efficiency of the algorithm. In the case of fuzzy Markov chains, we implemented and therefore can compare and contrast the following stopping conditions for our genetic algorithm:

Time-based stopping condition: This stopping condition terminates the algorithm after a certain amount of time has elapsed, regardless of the progress made by the algorithm. For example, we could specify that the GA should stop after running for 5 minutes. This stopping condition can be useful when we have limited computational resources and need to ensure that the algorithm does not run indefinitely. However, it does not take into account the progress made by the algorithm towards finding a good solution.

Number of generations stopping condition: This stopping condition terminates the algorithm after a certain number of generations, regardless of the progress made by the algorithm. For example, we could specify that the GA should stop after running for 100 generations. This stopping condition can be useful when we have a good idea of the number

of generations needed to converge to a good solution. However, it also does not take into account the progress made by the algorithm towards finding a good solution, and it may terminate prematurely if the algorithm has not converged.

Convergence detection stopping condition: This stopping condition terminates the algorithm when the population has converged to a good solution. In our case, it is detected by monitoring the difference between consecutive generations, and terminating the algorithm when a rolling average of last 100 such differences falls below a certain threshold. This stopping condition takes into account the progress made by the algorithm towards finding a good solution and can terminate the algorithm when it has converged, rather than running for a fixed amount of time or generations. However, it may take longer to converge than time or generation-based stopping conditions.

On Figure 7.7 we can see three different points at which the time-based stopping condition could trigger in some particular problem, depending on the chosen threshold. Red line signifies a threshold of 0.001, which means that for the last 100 generations, the average difference between two consecutive generations got below 0.001 (mind that here, vertical bars have a different meaning than in Figures 7.8 and 7.9). Here, this happened at generation 123 and 2.4 seconds of real-time computing. Subsequently, 0.0001 cutoff was triggered at generation 305 at 4.2 seconds, and 0.00001 cutoff happened at generation 647 and 7.7 seconds.

In general, the best stopping condition for a GA depends on the specific problem being solved and the resources available. In the case of fuzzy Markov chains, convergence detection is likely the most appropriate stopping condition since it takes into account the progress made by the algorithm towards finding a good solution. However, if there are computational resource constraints, a time-based stopping condition may be necessary. If we have prior knowledge about the number of generations needed to converge, a generation-based stopping condition may be appropriate.

7.3.4 Optimization results

The optimization phase of the project has achieved its goals by identifying optimal values of meta-parameters for different input sizes, analyzing the performance of the implemented selection, crossover, and mutation methods, and comparing different stopping conditions. Through grid search, we were able to determine the ideal population size, selection rate, and mutation rate for different input sizes, which has led to improved performance of the algorithm.

The reflection on the selection, crossover, and mutation parts of the algorithm showed that our implementation is efficient and effective in preserving genetic diversity and maintaining a high correlation between parent fitness and child fitness. These findings suggest that our implementation is robust and suitable for solving the reachability and stationary distribution problems for various input sizes.

Furthermore, the analysis of the convergence and the relationship between genetic diversity and the fitness of the best individual demonstrated that our algorithm is likely converging towards the true best result and not getting stuck in local optima. This is an important aspect of a genetic algorithm, as it ensures the algorithm is capable of finding high-quality solutions to complex problems.

Next, we tested different stopping conditions and compared their performance in terms of the quality of the solution and the computation time. Our results showed that the fixed amount of time stopping condition can be a good choice for users who prioritize getting an acceptable result as fast as possible, while the fixed number of generations stopping condition can be more suitable for users who are willing to wait longer for potentially better

solutions. The no significant improvement stopping condition can be used as a compromise between the two, as it allows the algorithm to continue searching for better solutions until it gets stuck in a local optimum. Overall, the choice of the stopping condition should be made based on the specific needs and preferences of the user.

Overall, the optimization phase provided valuable insights into the performance of the genetic algorithm and helped us fine-tune its parameters for optimal results.

7.4 Scalability

Scalability is an crucial aspect of any computational algorithm. The ability of our algorithm to handle larger matrices is necessary for its effectiveness and utility in real-world applications.

In the following measurements, our goal was to compare how quickly does the algorithm converge towards the optimal solution. However, because we lacked testing data for larger input size, we had to assume that the algorithm converges towards the global optimum given enough time. We base this assumption off of the fact that it holds for smaller input sizes (Section 7.2) and because genetic diversity remains high despite convergence (7.3.2). Therefore, for the sake of this section, we assumed that the optimal solution was reached after the algorithm didn't progress (in terms of improving the result) for at least 10 thousand generations.

To achieve scalability in our algorithm, it is important to address several factors. Firstly, the size of the input matrices should be considered. The algorithm should be able to handle different sizes of matrices without significant performance degradation or loss of accuracy. On Figure 7.8, you can see a performance comparison for computing reachability in 5 steps in 5×5, 10×10 and 20×20 matrices. Vertical bars are marking the time when the algorithm reached the optimal solution within two, three, and four decimal places. Exact values are provided in Table 7.2. As we can see input size has a big impact on performance, especially when the user needs an approximation more precise than two decimal places. One way to mitigate this in the future can be through the use of parallel computing techniques, which can potentially improve the efficiency of the algorithm.

Secondly, the needed number of steps in reachability problem can also have an impact on scalability, since the fitness function, ie. raising the matrix to the power of n, is more complex the bigger n gets. Indeed, raising n from 5 to 20 seems to have a moderate impact on performance, as shown in Figure 7.9 and Table 7.2. Having said that, when looking at the average time per generation in Table 7.3, we can see that it is similar. This suggests that the computational expensiveness of the fitness function does not linearly impact performance. Instead, it makes the search space more difficult to navigate, requiring more generations to find the optimal solution within the desired precision. Relationship between the matrices in the search space and their corresponding fitness values might be more chaotic for higher n values, which in turn leads to a slower convergence.

Finally, we considered to what extent can switching problems from reachability to stationary distribution make a difference in performance. In our tests (Figure 7.9 and Table 7.2), it showed to be more demanding than computing reachability in both 5 and 20 steps, at all three precision thresholds. Looking at time per generation values in Table 7.3 again suggests that the reason for this is increased difficulty of search space exploration.

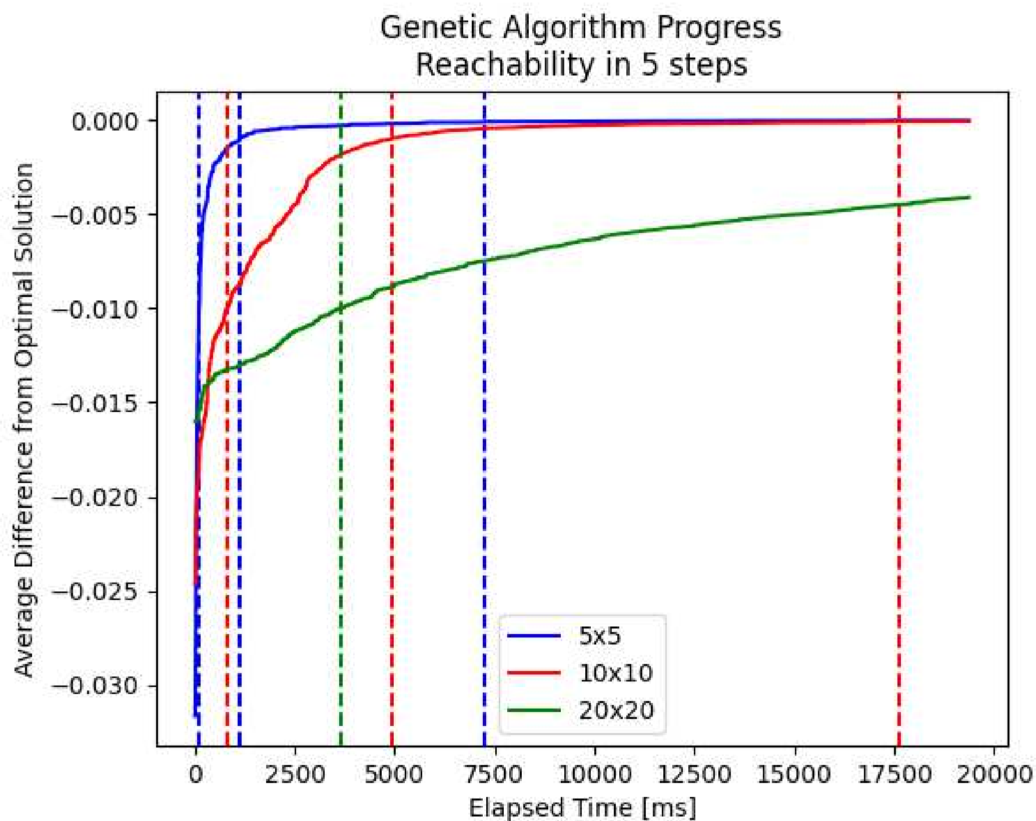

FIGURE 7.8: Performance for input sizes 5, 10, 20; Reachability in 5 steps

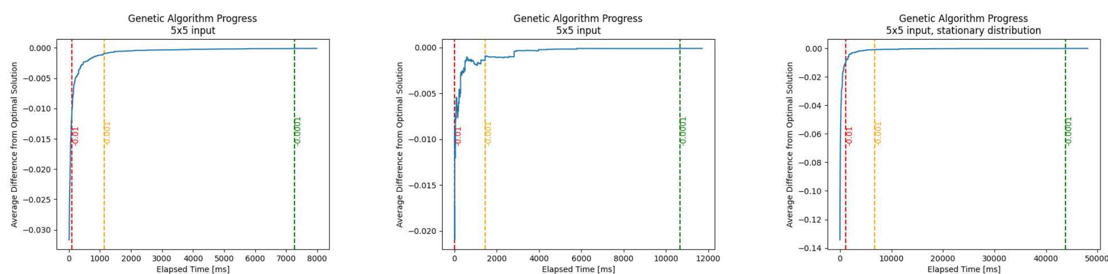

FIGURE 7.9: Performance for reachability in 5 and 20 steps
and stationary distribution (5 × 5 matrix)

Δ	Reachability, 5 steps			Reachability, 20 steps	Stationary distribution
	5 × 5	10 × 10	20 × 20	5 × 5	5 × 5
0.01	100	802	3647	331	1143
0.001	1129	4914	90719	2821	6767
0.0001	7262	17606	378538	15127	43741

TABLE 7.2: Reaching within Δ difference from optimal solution
for various problems and input sizes, time in milliseconds

Δ	Reachability, 5 steps			Reachability, 20 steps	Stationary distribution
	5×5	10×10	20×20	5×5	5×5
0.01	0.1500	0.0569	0.0165	0.1124	0.1211
0.001	0.0951	0.0700	0.0211	0.0992	0.1537
0.0001	0.1109	0.0851	0.0209	0.1305	0.1795

TABLE 7.3: Reaching within Δ difference from optimal solution for various problems and input sizes, milliseconds per generation

Chapter 8

Conclusion and future work

In conclusion, this paper presented an approach for improving the analysis of Markov chains in situations where the probabilities of events are uncertain and modeled by fuzzy numbers. We proposed methods to validate the basic properties of fuzzy Markov chains and discussed the challenges of implementing restricted fuzzy matrix multiplication, a fuzzy arithmetic approach that enables us to handle uncertain probabilities in the transition matrix. We chose to implement it in the form of a genetic algorithm that generates the values of the powers of the fuzzy transition matrix (reachability) as well as stationary distribution, which is appropriate for approximating the solutions of such hard problems. The algorithm was implemented into the probabilistic model checker STORM. Tests indicate that the results of the algorithm converge towards optimal solutions, preserve genetic variation over time, retain high correlation between parent and child fitness, and scale reasonably well, in particular when compared to the exact algorithm.

Future work could involve extending the modelling language JANI to handle fuzzy Markov chains as well as optimizing the algorithm further to achieve greater scalability, especially with higher input sizes. This could be done by utilizing parallel computing, better meta-parameter optimization (including dynamic selection and mutation rates), and optimizing the fitness functions. Additionally, exploring the applications of fuzzy Markov chains in various fields and domains could be an interesting direction for further research.

Chapter 9

Appendix

9.1 Fitness function, raising matrix to the power of n

The *matrixMul* algorithm takes a matrix P', raises it to a power p, and returns an element at a specific index. This is achieved through iterative matrix multiplication within nested loops. Each element in a temporary matrix *temp* is updated by multiplying and accumulating corresponding elements from the *result* matrix and P'. After each iteration, *result* is updated to hold the current *result* values, which is then reset for the next round of multiplication. The process repeats $p - 1$ times, effectively raising the matrix to the desired power, with the specified element being returned at the end.

Algorithm 8 Matrix exponentiation

```
1: function matrixMul(P', p, idx)
2:     n ← P'.size                                  ▷ Number of rows in P'
3:     temp ← matrix(n, n, 0)       ▷ matrix with size of n × n, filled with 0s
4:     result ← P'
5:     for l ← 0 to p − 1 do
6:         for j ← 0 to n do
7:             for k ← 0 to n do
8:                 for i ← 0 to n do
9:                     temp[i][j] ← temp[i][j] + result[i][k] · P'[k][j]
10:                end for
11:            end for
12:        end for
13:        result ← temp
14:        fill(temp, 0)
15:    end for
16:    return result[idx.x][idx.y]
17: end function
```

9.2 Fitness function, stationary distribution

The *stationaryDistribution* algorithm finds the stationary distribution of an input matrix P' by iteratively multiplying a vector w with P' until the vector's change drops below a given threshold ε. This distribution is the vector that remains unchanged when multiplied with the matrix. Once the vector stabilizes, it is normalized to sum to 1. The algorithm

then returns a specific element from the normalized stationary distribution, as indicated by the provided index.

Algorithm 9 Stationary distribution

```
1:  function STATIONARYDISTRIBUTION(P', idx)
2:      n ← P'.size                                  ▷ Number of rows in P'
3:      w ← vector(n)                                ▷ 2D vector of length n
4:      diff ← 1, prev_diff = 0.
5:      while diff < ε do
6:          prev_w ← w.
7:          for i ← 0 to n do:
8:              w[i] ← 0
9:              for j ← 0 to n do
10:                 w[i] ← w[i] + prev_w[j] · P[j][i]
11:             end for
12:         end for
13:         prev_diff ← diff, diff ← 0
14:         for i ← 0 to n do
15:             diff = diff + |w[i] − prev_w[i]|
16:         end for
17:     end while
18:     sum ← 0
19:     for i ← 0 to n do
20:         sum = sum + w[i]
21:     end for
22:     for i ← 0 to n do
23:         w[i] = w[i]/sum
24:     end for
25:     return w[idx.y]
26: end function
```

9.3 Check whether a Markov chain is regular

The *isRegular* algorithm checks if a given matrix P forms a regular Markov chain. This is determined by two conditions: irreducibility and aperiodicity. Both conditions are checked using their respective helper algorithms, *isIrreducible* and *isAperiodic*.

The *isIrreducible* algorithm examines if there's a path from every state to every other state in the system. This is done by iteratively checking and updating reachable states from each state in the system. If any state isn't reachable from others, the system isn't irreducible.

The *isAperiodic* algorithm verifies that the system doesn't oscillate in a deterministic cycle. For each state, it finds the size of cycles it's part of. If the greatest common divisor of cycle sizes for any two states isn't 1, the system isn't aperiodic.

If the input matrix satisfies both conditions, it forms a regular Markov chain. Otherwise, it doesn't.

Algorithm 10 Check if Markov chain is regular

```
1: function isRegular(P, n)
2:     if isIrreducible(P, n) = false||isAperiodic(P, n) = false then
3:         return false
4:     end if
5:     return true
6: end function
```

Algorithm 11 Check if Markov chain is irreducible

```
1: function isIrreducible(P, n)
2:     reachable ← vector(n)                              ▷ Number of rows in P
3:     reachable[0] ← 1
4:     for k ← 0 to n do
5:         newReachable ← reachable
6:         for i ← 0 to n do
7:             if reachable[i] = 1 then
8:                 for j ← 0 to n do
9:                     if P[i][j] > 0 then
10:                        newReachable[j] ← 1
11:                    end if
12:                end for
13:            end if
14:        end for
15:        reachable ← newReachable
16:    end for
17:    irreducible ← true
18:    for i ← 0 to n do
19:        if reachable[i] = 0 then
20:            irreducible ← false
21:            break
22:        end if
23:    end for
24:    if irreducible = false then
25:        return false
26:    end if
27: end function
```

Algorithm 12 Check if Markov chain is aperiodic

```
1: function isAperiodic(P, n)
2:     d ← vector(n)                              ▷ Number of rows in P
3:     for i ← 0 to n do
4:         d[i] ← 1
5:         for k ← 1 do
6:             if P[i][i] > 0 then
7:                 d[i] ← k
8:                 break
9:             end if
10:            p ← vector(n)
11:            for j ← 0 to n do
12:                for l ← 0 to n do
13:                    p[j] ← p[j] + P[l][j] · P[i][l]
14:                end for
15:            end for
16:            P[i] ← p
17:        end for
18:    end for
19:    aperiodic ← true
20:    for i ← 0 to n do
21:        for j ← 0 to n do
22:            if gcd(d[i], d[j]) ≠ 1 then
23:                aperiodic ← false
24:                break
25:            end if
26:        end for
27:        if aperiodic = false then
28:            return false
29:        end if
30:    end for
31:    if aperiodic = false then
32:        return false
33:    end if
34:    return true
35: end function
```

9.4 Check whether a Markov chain is absorbing

The *isAbsorbing* algorithm checks whether a given crisp matrix P forms an absorbing Markov chain. An absorbing Markov chain is one in which every state can reach an absorbing state (a state that, once entered, cannot be left) in a finite number of steps.

The algorithm starts by identifying all absorbing states in the matrix, i.e., those states where the probability of staying in the same state is 1. If no absorbing states are found, the function immediately returns false, concluding that the Markov chain isn't absorbing.

Next, for each non-absorbing state, the algorithm determines if there's a path to an absorbing state. This is done by computing the state transition probabilities for a step and checking if there's a non-zero probability of reaching an absorbing state. If all non-

absorbing states have a reachable path to an absorbing state, the function returns true, confirming that the Markov chain is absorbing. Otherwise, it returns false.

Algorithm 13 Check if Markov chain is absorbing

```
1:  function isAbsorbing(P, n)
2:      k ← 0
3:      absorbingStates ← vector()
4:      for i ← 0 to n do                                   ▷ Number of rows in P
5:          if P[i][i] = 1 then
6:              absorbingStates.append(i)
7:              k ← k + 1
8:          end if
9:      end for
10:     if k = 0 then
11:         return false
12:     end if
13:     for i ← 0 to n do
14:         reachable ← false
15:         for j ← 0 to absorbingStates.size do
16:             if i = absorbingStates[j] then
17:                 reachable ← true
18:                 break
19:             end if
20:         end for
21:         if reachable = false then
22:             p ← vector[n]
23:             for k ← 0 to n do
24:                 p[k] ← P[i][k]
25:             end for
26:             for k ← 0 to n do
27:                 for l ← 0 to n do
28:                     p[k] ← p[k] + P[i][l] · P[l][k]
29:                 end for
30:             end for
31:             for j ← 0 to absorbingStates.size do
32:                 if p[absorbingStates[j]] > 0 then
33:                     reachable ← true
34:                     break
35:                 end if
36:             end for
37:             if reachable = false then
38:                 return false
39:             end if
40:         end if
41:     end for
42:     return true
43: end function
```

9.5 Exact algorithm to use as a reference for genetic algorithm's validity

The *bruteForceMatrixMul* function performs a brute force search to find the matrix that, when raised to a certain power, yields a specific element with the optimal value, as per a given direction (either maximum or minimum). It receives an interval matrix, in which each cell contains a range of potential values, and explores all possible matrices that can be formed within these intervals.

The function iterates through the interval matrix cell by cell, using nested recursion to explore all combinations of potential cell values. Each time it fills all the cells of *currentMatrix*, it checks if the matrix is valid (e.g., the sum of each row's elements equals 1 with a certain tolerance). If the matrix is valid, it calculates the result by raising the matrix to a given power and retrieving a specific element (*idx*). If the current result is better than the best one found so far, it updates the *bestResult* and *bestMatrix*.

The function uses depth-first search to explore the matrix configuration space and will return the best matrix found, or an empty matrix if no valid matrix is found within the provided intervals.

Algorithm 14 Matrix validity check

```
1: function isValidMatrix(matrix, tolerance)
2:     for all row in matrix do
3:         if abs(sum(row) − 1.0) ⩾ tolerance then
4:             return false
5:         end if
6:     end for
7:     return true
8: end function
```

Algorithm 15 Brute Force Matrix Multiplication

```
1: function BRUTEFORCEMATRIXMUL(intervalMatrix, currentMatrix, bestResult, i,
   j, steps, idx, direction, tolerance = 0.005, iterStep = 0.005)
2:     if i = intervalMatrix.size then
3:         if isValidMatrix(currentMatrix, tolerance) then
4:             currentResult ← matrixMul(currentMatrix, steps, idx.x, idx.y)        ▷
   Fitness function, equivalent to one in the genetic algorithm
5:             if (direction&&currentResult < bestResult)||(!direction&&
   currentResult > bestResult) then
6:                 bestResult ← currentResult
7:                 return currentMatrix
8:             end if
9:         end if
10:        return {}                                                    ▷ Empty matrix
11:    end if
12:    bestMatrix ← matrix()
13:    for x ← intervalMatrix[i][j].lower to intervalMatrix[i][j].upper, step iterStep
   do
14:        currentMatrix[i][j] ← x
15:        newBestMatrix ← bruteForceMatrixMul(intervalMatrix, currentMatrix,
   bestResult, i + (j + 1)/intervalMatrix[i].size, (j + 1)%intervalMatrix[i].size, steps,
   idx, direction, tolerance, iterStep)
16:        if newBestMatrix = {} then
17:            bestMatrix ← newBestMatrix
18:        end if
19:    end for
20:    return bestMatrix
21: end function
```

Bibliography

[1] carl: Carl. https://ths-rwth.github.io/carl/. Accessed: 2023-02-24.

[2] Github: storm_fuzzy. https://github.com/SochaK148/storm_fuzzy. Accessed: 2023-05-08.

[3] K. Avrachenkov and Elie Sanchez. Fuzzy markov chains and decision-making. *Fuzzy Optimization and Decision Making*, 1:143–159, 2002. doi:10.1023/A:1015729400380.

[4] James Buckley. *Fuzzy Probabilities*, volume 115. Physica-Verlag HD, 2003. doi:10.1007/978-3-642-86786-6.

[5] James Buckley and Esfandiar Eslami. *An Introduction to Fuzzy Logic and Fuzzy Sets / J.J. Buckley, E. Eslami.* 2002. doi:10.1007/978-3-7908-1799-7.

[6] J.J. Buckley, T. Feuring, and Y. Hayashi. Fuzzy markov chains. In *Proceedings Joint 9th IFSA World Congress and 20th NAFIPS International Conference (Cat. No. 01TH8569)*, pages 2708–2711 vol.5, 2001. doi:10.1109/NAFIPS.2001.943652.

[7] Esfandiar Eslami and James Buckley. Fuzzy markov chains: Uncertain probabilities. *Mathware & soft computing, ISSN 1134-5632, Vol. 9, Nº. 1, 2002, pags. 33-41*, 9, 2002.

[8] Paul Gagniuc. *Markov Chains: From Theory to Implementation and Experimentation.* 2017. doi:10.1002/9781119387596.

[9] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *Int. J. Softw. Tools Technol. Transf.*, 24(4):589–610, 2022.

[10] Rudolf Kruse, Rüdiger Buck-Emden, and Ralf Cordes. Processor power considerations — an application of fuzzy markov chains. *Fuzzy Sets and Systems*, 21(3):289–299, 1987. doi:https://doi.org/10.1016/0165-0114(87)90130-8.

[11] Nahid Salimi, Vahid Rafe, Hamed Tabrizchi, and Amir Mosavi. Fuzzy genetic algorithm approach for verification of reachability and detection of deadlock in graph transformation systems. In *2020 IEEE 3rd International Conference and Workshop in Óbuda on Electrical and Power Engineering (CANDO-EPE)*, pages 000241–000250, 2020. doi:10.1109/CANDO-EPE51100.2020.9337781.

[12] N Ravi Shankar, G Ananda Rao, J Madhu Latha, and V Sireesha. Solving a fuzzy nonlinear optimization problem by genetic algorithm. *Int. J. Contemp. Math. Sciences*, 5(16):791–803, 2010.

[13] Horng-Ren Tsai and Toly Chen. A fuzzy nonlinear programming approach for optimizing the performance of a four-objective fluctuation smoothing rule in a wafer fabrication factory. *J. Appl. Math.*, 2013:720607:1–720607:15, 2013.

[14] Y. Yin, I. Kaku, J. Tang, and J.M. Zhu. *Data Mining: Concepts, Methods and Applications in Management and Engineering Design*. Decision Engineering. Springer London, 2011.