Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Automatisiertes Finden von Gegenbeispielen algebraischer und formalsprachlicher Eigenschaften

Eric Förster

Studiengang:	Informatik
Prüfer/in:	PD Dr. rer. nat. Manfred Kufleitner
Betreuer/in:	Lukas Fleischer, M.Sc.
Beginn am:	29. November 2017
Beendet am:	29. Mai 2018

Kurzfassung

In der Mathematik werden häufig Gegenbeispiele genutzt, um eine Behauptung zu widerlegen. Wir werden uns in dieser Arbeit mit dem Auffinden solcher Beispiele im Bereich der endlichen Halbgruppentheorie beschäftigen. Dabei soll zu einer gegebenen Eigenschaft eine möglichst einfache Halbgruppe mit dieser Eigenschaft gefunden werden. Zur Lösung dieses Problems entwerfen wir in dieser Arbeit geeignete Suchverfahren und Heuristiken.

Inhaltsverzeichnis

1. Einleitung

Beim Studieren von endlichen Halbgruppen kann es hilfreich sein, möglichst einfache Beispiele von Halbgruppen mit bestimmten Eigenschaften zu finden. So können beispielsweise bestimmte Muster und Regelmäßigkeiten erkannt werden oder aufgestellte Hypothesen durch das Finden von Gegenbeispielen widerlegt werden.

Wir werden in dieser Arbeit einige Algorithmen entwerfen, die das folgende Problem lösen: Finde zu einer gegebenen Eigenschaft eine möglichst einfache Halbgruppe, die diese Eigenschaft erfüllt.

Dabei werden wir Eigenschaften unter anderem durch sogenannte *Varietäten* beschreiben. Eine Varietät ist eine Klasse von endlichen Halbgruppen, die unter Division und endlichen Produkten abgeschlossen ist. Eilenberg [2] führte dieses Konzept in 1976 ein, um einen wesentlichen Zusammenhang zwischen formalen Sprachen und endlichen Halbgruppen zu zeigen. Durch diese enge Verbindung werden diese Objekte seither intensiv studiert. Reiterman [8] zeigte, dass sich jede Varietät durch eine Menge von *Gleichungen* charakterisieren lässt. Beispielsweise lässt sich die Varietät aller kommutativen Halbgruppen durch die Gleichung $xy = yx$ beschreiben. Ist diese Gleichungsmenge endlich und sind diese von einer bestimmten Form, so können wir ganz einfach in polynomieller Zeit überprüfen, ob eine Halbgruppe in dieser Varietät liegt, indem wir die Gleichungen durch Einsetzen aller Elemente überprüfen.

Die Arbeit ist wie folgt aufgebaut: Wir führen in Kapitel 2 die benötigten algebraischen Grundlagen ein. Dies beinhaltet vor allem die Theorie der endlichen Halbgruppen und Varietäten. Im Hauptteil der Arbeit nähern wir uns schrittweise einer Lösung des Problems. In Kapitel 3 formalisieren wir das Problem als Suchproblem und erläutern die allgemeine Vorgehensweise. Anschließend versuchen wir in Kapitel 4 das Problem mit klassischen Suchalgorithmen zu lösen, womit wir aber sehr schnell an Grenzen stoßen werden. Um den enormen Suchraum in den Griff zu bekommen, entwickeln wir in Kapitel 5 heuristische Suchverfahren. In Kapitel 6 fokussieren wir uns auf einige Details der Implementierung dieser Algorithmen. Zum Schluss geben wir in Kapitel 7 einige gefundene Beispiele an und vergleichen die verschiedenen Algorithmen mithilfe eines Benchmarks über der sogenannten *Trotter-Weil-Hierarchie*.

2. Algebraische Grundlagen

In diesem Kapitel führen wir die grundlegenden Konzepte der endlichen Halbgruppentheorie ein und geben wichtige Resultate an. Eine ausführliche Einführung findet sich beispielsweise in [1] und in [7].

2.1. Halbgruppen

Eine *Halbgruppe* $(S, \cdot)$ besteht aus einer nichtleeren Menge S und einer assoziativen Verknüpfung $\cdot : S \times S \to S$, das heißt, es gilt $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ für alle $x, y, z \in S$.

Ein Element $e \in S$ heißt *idempotent*, falls $e^2 = e$ gilt. Die Menge aller idempotenten Elemente bezeichnen wir mit $E(S)$. Ist S endlich, so existiert ein $n \in \mathbb{N}$, sodass x^n idempotent ist für alle $x \in S$. Das idempotente Element x^n ist eindeutig bestimmt und wir nennen es das von x erzeugte idempotente Element. Der unäre Operator $\cdot^\omega : S \to S$ ordnet jedem Element x das von x erzeugte idempotente Element x^ω zu.

Ein *Monoid* $(M, \cdot, 1)$ ist eine Halbgruppe mit einem neutralen Element $1 \in M$, sodass $1 \cdot x = x \cdot 1 = x$ für alle $x \in M$ gilt. Es ist leicht zu sehen, dass in einem Monoid das neutrale Element eindeutig bestimmt ist. Eine Halbgruppe S kann zu einem Monoid S^1 ergänzt werden, indem man

$$S^1 := \begin{cases} S & \text{falls } S \text{ ein Monoid ist} \\ S \cup \{1\} & \text{sonst} \end{cases}$$

setzt und die Verknüpfung so erweitert, dass $1 \cdot x = x \cdot 1 = x$ für alle $x \in S^1$ gilt.

Wir nennen T eine *Unterhalbgruppe* von S, falls $T \subseteq S$ und $x \cdot y \in T$ für alle $x, y \in T$ gilt. Ein *Untermonoid* eines Monoids M ist eine Unterhalbgruppe von M, die das neutrale Element $1 \in M$ enthält.

Sei $A \subseteq S$ eine Teilmenge einer Halbgruppe S. Dann ist die von A *erzeugte Unterhalbgruppe* $\langle A \rangle$ die kleinste Unterhalbgruppe von S, welche A umfasst. Sie enthält alle Elemente der Form $x = a_1 \cdots a_n$ mit $a_i \in A$ und $n \in \mathbb{N}$. Ist S ein Monoid, so enthält $\langle A \rangle$ immer das neutrale Element von S.

Ein *Halbgruppenmorphismus* ist eine Abbildung $\varphi : S \to T$ zwischen zwei Halbruppen, sodass $\varphi(xy) = \varphi(x)\varphi(y)$ für alle $x, y \in S$ gilt. Sind S und T zwei Monoide, so nennt man φ einen *Monoidmorphismus*, falls zusätzlich $\varphi(1) = 1$ gilt. Ein *Isomorphismus* ist ein bijektiver Morphismus. Existiert ein surjektiver Morphismus $\varphi : S \to T$, so ist T ein *Quotient* von S. Eine Halbgruppe T *teilt* S, falls T ein Quotient einer Unterhalbgruppe von S ist.

Für eine Familie $(S_i)_{i\in I}$ von Halbgruppen definieren wir das *Produkt* $\Pi_{i\in I}S_i$ durch die Halbgruppe über dem kartesischen Produkt mit der komponentenweisen Verknüpfung. Ist $I = \emptyset$, so setzen wir das Produkt auf die triviale Halbgruppe $\{1\}$.

Eine *Transformation* einer Menge $P = \{1, \dots, n\}$ ist eine Abbildung $f : P \to P$. Der *Grad* einer Transformation ist die Zahl n. Es gibt n^n viele Transformationen von P. Das *volle Transformationsmonoid* $\mathcal{T}_n$ besteht aus allen Transformationen über P zusammen mit der Operation $f \cdot g := g \circ f$ und dem neutralen Element id_P. Wir können Unterhalbgruppen von $\mathcal{T}_n$ nutzen, um jede endliche Halbgruppe darzustellen:

Satz von Cayley für Halbgruppen: *Sei S eine endliche Halbgruppe. Dann existiert ein $n \in \mathbb{N}$, sodass S isomorph zu einer Unterhalbgruppe von $\mathcal{T}_n$ ist.*

Der Beweis des Satzes findet sich unter anderem in [7].

2.2. Greensche Relationen

Die *Greenschen Relationen* $\mathcal{L}, \mathcal{R}, \mathcal{J}, \mathcal{H}, \mathcal{D}$ sind fünf wichtige Äquivalenzrelationen auf einer Halbgruppe. Zunächst definieren wir die folgenden Ordnungsrelationen auf einer Halbgruppe S:

$$x \leq_{\mathcal{L}} y :\Leftrightarrow S^1x \subseteq S^1y$$
$$x \leq_{\mathcal{R}} y :\Leftrightarrow xS^1 \subseteq yS^1$$
$$x \leq_{\mathcal{J}} y :\Leftrightarrow S^1xS^1 \subseteq S^1yS^1$$

Dabei ist $S^1x := \{sx \mid s \in S^1\}$ und $xS^1 := \{xs \mid s \in S^1\}$. Wir schreiben $x <_{\mathcal{J}} y$, falls $x \leq_{\mathcal{J}} y$ gilt, aber nicht $y \leq_{\mathcal{J}} x$. Wir definieren nun die $\mathcal{L}$-Relation durch

$$x \,\mathcal{L}\, y :\Leftrightarrow x \leq_{\mathcal{L}} y \wedge y \leq_{\mathcal{L}} x$$

Die Relationen $\mathcal{R}$ und $\mathcal{J}$ sind analog definiert. Die letzten beiden Relationen definieren wir durch $\mathcal{H} := \mathcal{L} \cap \mathcal{R}$ und

$$x \,\mathcal{D}\, y :\Leftrightarrow \exists z \in S\colon x \,\mathcal{L}\, z \wedge z \,\mathcal{R}\, y$$

Wir nennen eine Halbgruppe $\mathcal{J}$-trivial, falls aus $x \,\mathcal{J}\, y$ folgt, dass $x = y$ ist. Eine $\mathcal{D}$-Klasse ist *regulär*, falls sie ein idempotentes Element enthält.

2.3. Varietäten

Eine *Varietät* von Halbgruppen ist eine Klasse von endlichen Halbgruppen, die unter Division sowie endlichen Produkten abgeschlossen ist. Daraus folgt sofort, dass die triviale Halbgruppe $\{1\}$ in jeder Varietät enthalten ist. Es ist üblich Varietäten durch Gleichungen zu definieren. Hierzu definieren wir *ω-Terme* induktiv über einer endlichen Menge von Variablen Σ:

- Das leere Wort 1 ist ein ω-Term.
- Für jedes $x \in \Sigma$ ist x ein ω-Term.
- Sind u und v zwei ω-Terme, so sind u^ω und uv ebenso ω-Terme.

Jeder Morphismus $\varphi\colon \Sigma^* \to S$ auf eine endliche Halbgruppe S lässt sich zu einer *Auswertung* auf ω-Terme erweitern, indem wir $\varphi(u^\omega) := \varphi(u)^\omega$ setzen. Sind u und v zwei ω-Terme, so nennen wir $u = v$ eine *Gleichung*. Eine endliche Halbgruppe S erfüllt diese, falls $\varphi(u) = \varphi(v)$ für alle Auswertungen φ gilt. Für eine endliche Folge $(u_i = v_i)_{i\in I}$ von Gleichungen bezeichnen wir die Varietät aller endlichen Halbgruppen, die alle Gleichungen erfüllen, mit $[\![(u_i = v_i)_{i\in I}]\!]$. Wir betrachten in dieser Arbeit die folgenden Varietäten:

$$\begin{aligned}\mathbf{J} &:= [\![y(xy)^\omega = (xy)^\omega, (xy)^\omega = (xy)^\omega x]\!]\\ \mathbf{DS} &:= [\![((xy)^\omega(yx)^\omega(xy)^\omega)^\omega = (xy)^\omega]\!]\\ \mathbf{DO} &:= [\![(xy)^\omega(yx)^\omega(xy)^\omega = (xy)^\omega]\!]\end{aligned}$$

Außerdem bezeichne **Sgp** die Varietät aller endlichen Halbgruppen. Wir können die obigen Varietäten mithilfe der Greenschen Relationen charakterisieren: Die Varietät **J** beschreibt die Klasse aller $\mathcal{J}$-trivialen Halbgruppen. **DS** ist die Varietät aller Halbgruppen, deren reguläre $\mathcal{D}$-Klassen eine Unterhalbgruppe bilden. Falls zusätzlich alle idempotente Elemente in einer solchen Unterhalbgruppe wiederum eine Unterhalbgruppe bilden, so ist eine solche Halbgruppe in **DO**.

2.4. Trotter-Weil-Hierarchie

Bei der Trotter-Weil-Hierarchie handelt es sich um eine unendliche Hierarchie von Varietäten. Sie hat eine enge Verbindung zu bestimmten Fragmenten der Zwei-Variablen-Logik über Wörter. Wir halten uns in dieser Arbeit an die Definition von Kufleitner und Weil [6]. Zunächst definieren wir zwei Äquivalenzrelationen auf einem Monoid M:

$$\begin{aligned}x \sim_K y &:\Leftrightarrow \forall e \in E(M)\colon ex = ey \vee ex, ey <_{\mathcal{J}} e\\ x \sim_D y &:\Leftrightarrow \forall e \in E(M)\colon xe = ye \vee xe, ye <_{\mathcal{J}} e\end{aligned}$$

Außerdem benötigen wir das *Mal'cev-Produkt*: Für eine Varietät **V** bezeichnet $\mathbf{K}\,(m)\,\mathbf{V}$ die Varietät aller Monoide M mit $M/\!\sim_K \in \mathbf{V}$. Das Produkt $\mathbf{D}\,(m)\,\mathbf{V}$ definieren wir analog. Wir definieren nun die *Trotter-Weil-Hierarchie* rekursiv für alle $d \geq 1$ durch

$$\begin{aligned}\mathbf{R}_{d+1} &:= \mathbf{K}\,(m)\,\mathbf{L}_d\\ \mathbf{L}_{d+1} &:= \mathbf{D}\,(m)\,\mathbf{R}_d\end{aligned}$$

mit den Anfangswerten $\mathbf{R}_1 := \mathbf{L}_1 := \mathbf{J}$. Wir interessieren uns in dieser Arbeit vor allem für die Schnittebenen $\mathbf{R}_d \cap \mathbf{L}_d$ der Hierarchie. Dabei fallen die ersten beiden Ebenen zusammen, das heißt, es gilt $\mathbf{R}_2 \cap \mathbf{L}_2 = \mathbf{J}$. Die Bezeichnung als Hierarchie wird durch die Inklusion $(\mathbf{R}_d \cap \mathbf{L}_d) \subseteq (\mathbf{R}_{d+1} \cap \mathbf{L}_{d+1})$ gerechtfertigt.

3. Problemstellung

Wir werden nun das in der Einleitung beschriebene algorithmische Problem formalisieren und die allgemeine Lösungsstrategie skizzieren. Da wir uns für die algorithmische Berechnung von Halbgruppen interessieren, beschränken wir uns in dieser Arbeit auf endliche Halbgruppen. Zunächst präzisieren wir den Begriff der Eigenschaft:

Definition: Eine *Eigenschaft von Halbgruppen* ist eine Abbildung $\chi\colon \mathbf{Sgp} \to \{0,1\}$. Eine Halbgruppe $S \in \mathbf{Sgp}$ erfüllt diese, falls $\chi(S) = 1$ gilt.

Das Problem kann nun wie folgt beschrieben werden:

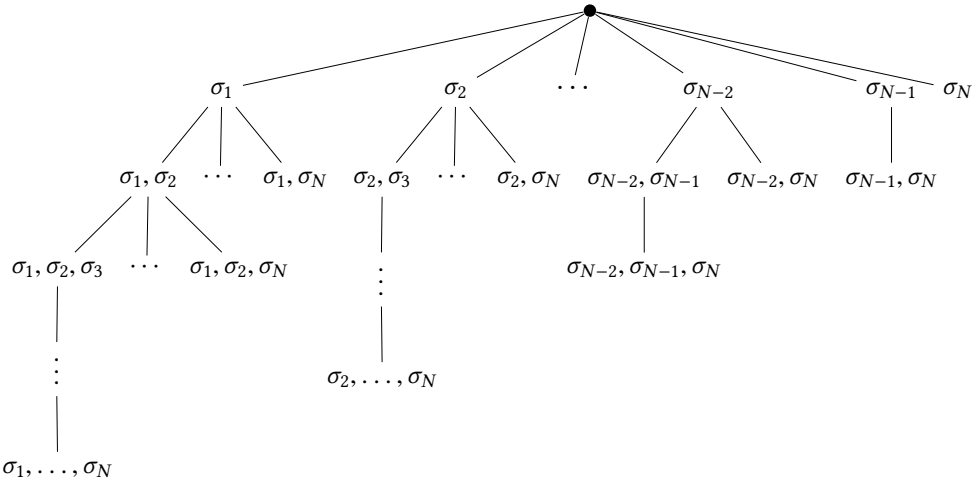
Eingabe:	Zwei berechenbare Eigenschaften χ_1 und χ_2 von Halbgruppen.
Ausgabe:	Eine Halbgruppe $S \in \mathbf{Sgp}$, sodass $\chi_1(S) = 1$ und $\chi_2(S) = 0$ ist.

Nach dem Satz von Cayley für Halbgruppen können wir uns bei der Suche nach geeigneten Halbgruppen auf Unterhalbgruppen von Transformationsmonoiden beschränken. Froidure und Pin beschreiben in [4] unter anderem einen Algorithmus, der aus einer Teilmenge $A \subseteq S$ einer Halbgruppe die von A erzeugte Unterhalbgruppe berechnet:

Algorithmus 3.1 : Berechnung von Unterhalbgruppen

```
function Closure(A)
    S ← A
    Q ← Queue(A)
    while |Q| > 0 do
        x ← Dequeue(Q)
        foreach a ∈ A do
            if xa ∉ S then
                Enqueue(Q, xa)
                S ← S ∪ {xa}
    return S
```

Der Algorithmus berechnet solange alle Elemente der Form $x = a_1 \cdots a_m$ mit $a_i \in A$, bis dadurch keine neuen Elemente mehr dazukommen. Dies entspricht genau der von A erzeugten Unterhalbgruppe. Der grundlegende Ansatz wird es nun sein, alle Teilmengen $A \subseteq \mathcal{T}_n$ eines Transformationsmonoids mit obigen Algorithmus zu einer Halbgruppe zu erweitern und die Eigenschaften zu prüfen. Dies liefert für jedes Transformationsmonoid $\mathcal{T}_n = \{\sigma_1, \ldots, \sigma_N\}$ einen endlichen Suchbaum:



Wählen wir $\sigma_1 = \mathrm{id}$, so können wir die Suche leicht auf Monoide einschränken. Die Knotenanzahl in diesen Bäumen wächst sehr schnell mit $2^{|\mathcal{T}_n|} = 2^{n^n}$. Um den Suchbaum weiter einzuschränken, betrachten wir Eigenschaften des folgenden Typs:

Definition: Eine Eigenschaft $\chi\colon \mathbf{Sgp} \to \{0, 1\}$ von Halbgruppen heißt *monoton*, falls aus $S \subseteq T$ folgt, dass $\chi(S) \geq \chi(T)$ für alle $S, T \in \mathbf{Sgp}$ gilt.

Ist eine Eigenschaft monoton, so können wir während der Suche einen Teilbaum abschneiden, falls dessen Wurzel die Eigenschaft nicht mehr erfüllt. Für zwei monotone Eigenschaften χ_1 und χ_2 gilt offenbar, dass die kombinierte Eigenschaft $\chi = \chi_1 \cdot \chi_2$ ebenso monoton ist. Wir zeigen nun, dass Varietäten, die durch endlich viele Gleichungen definiert sind, monotone Eigenschaften sind:

Lemma 1: *Sei $(u_i = v_i)_{i \in I}$ eine endliche Folge von Gleichungen. Dann ist*

$$\chi(S) = \begin{cases} 1 & \text{falls } S \in [\![u_i = v_i]\!] \text{ für alle } i \in I \\ 0 & \text{sonst} \end{cases}$$

eine monotone Eigenschaft von Halbgruppen.

Beweis: Es genügt die Aussage für $|I| = 1$ zu zeigen. Sei $(u = v)$ eine Gleichung und seien $S, T \in \mathbf{Sgp}$, sodass $S \subseteq T$ gilt. Falls $\chi(S) = 0$ ist, so existiert eine Auswertung φ, sodass $\varphi(u) \neq \varphi(v)$ gilt. Diese ist aber auch eine Auswertung über T, womit $\chi(T) = 0$ ist. Für $\chi(S) = 1$ gilt per Definition immer $\chi(T) \leq 1$. □

4. Deterministische Ansätze

In diesem Kapitel stellen wir zwei deterministische Algorithmen vor, welche das Problem durch systematisches Absuchen der im vorherigen Kapitel vorgestellten Suchbäume lösen. Seien hierzu χ_1 und χ_2 Eigenschaften von Halbgruppen, wobei wir aus Komplexitätsgründen annehmen, dass χ_1 monoton ist. Der Einfachheit halber betrachten wir den Suchbaum eines festen Transformationsmonoid $\mathcal{T}_n$ mit $n \in \mathbb{N}$. Ist die Suche in diesem Baum erfolglos, so kann diese in den Bäumen von $\mathcal{T}_{n+1}, \mathcal{T}_{n+2}, \ldots$ fortgesetzt werden. Diese Algorithmen haben den Vorteil, dass sie *vollständig* sind, das heißt es wird immer eine Lösung gefunden, sofern diese existiert.

4.1. Breitensuche

Die Breitensuche ist ein einfaches Suchverfahren, bei der zunächst alle von der Wurzel aus erreichbaren Knoten überprüft werden. Anschließend überprüfen wir die Folgeknoten dieser Knoten:

Algorithmus 4.1 : Breitensuche

```
function Breadth-First-Search(n, χ1, χ2)
    Q ← Queue({[σ] | σ ∈ T_n})
    while |Q| > 0 do
        A = [τ1, ..., τm] ← Dequeue(Q)
        S ← ⟨A⟩
        if χ1(S) = 1 then
            if χ2(S) = 0 then
                return S
            for i ← m + 1 to n do
                Enqueue(Q, [τ1, ..., τm, σi])
    return null
```

Zunächst bevorzugt der Algorithmus Lösungen mit kleiner Erzeugermenge, da diese der Größe nach aufsteigend besucht werden. Außerdem nutzen wir die Monotonie von χ_1 aus, da wir neue Erzeugermengen nur in die Queue aufnehmen, falls $\chi(S) = 1$ ist. Dadurch schneiden wir Teilbäume ab, die keine Lösungen mehr enthalten können. Der Algorithmus hat aber einen hohen Speicherbedarf, da für jede Erzeugermenge

$A = \{\tau_1, \ldots, \tau_m\}$, die aus der Queue genommen wird, genau $n - (m + 1)$ viele neue Erzeugermengen eingereiht werden. Dies macht den Algorithmus für große $n \in \mathbb{N}$ unpraktikabel.

4.2. Backtracking

Backtracking ist eine rekursive Suchtechnik, bei der Teillösungen schrittweise zu einer Gesamtlösung erweitert werden. Falls die aktuelle Teillösung nicht zu einer Gesamtlösung führen kann, so wird der letzte Schritt rückgängig gemacht. Dadurch werden systematisch alle möglichen Teillösungen betrachtet. Auf unser Problem übertragen heißt das, dass wir mit allen einelementigen Erzeugermengen starten und erstmal immer neue Elemente hinzufügen. Stellen wir dabei fest, dass die Eigenschaft χ_1 nicht mehr erfüllt ist, so benutzen wir die Monotonieeigenschaft und überspringen diesen Teilbaum.

Algorithmus 4.2 : Backtracking

```
function Backtracking-Search(n, χ1, χ2)
    function Backtrack(A = [τ1, ..., τm])
        S ← ⟨A⟩
        if χ1(S) = 1 then
            if χ2(S) = 0 then
                return S
            for i ← m + 1 to n do
                Push-Back(A, σi)
                T ← Backtrack(A)
                if T ≠ null then
                    return T
                Pop-Back(A)
        return null
    foreach σ ∈ T_n do
        S ← Backtrack([σ])
        if S ≠ null then
            return S
    return null
```

Wir können diesen Algorithmus einfach parallelisieren, indem wir die äußerste Schleife parallel ausführen. Im Vergleich zur Breitensuche ist der Speicherverbrauch sehr gering, da immer nur eine Erzeugermenge gespeichert wird. Dafür haben wir aber die Bevorzugung von kleinen Erzeugermengen verloren, da wir nun nach dem Prinzip der Tiefensuche arbeiten.

5. Heuristiken

Die Algorithmen aus dem letzten Kapitel durchsuchen systematisch den kompletten Suchbaum, wodurch sehr viele Pfade betrachtet werden. Dies ist bei einem großen Suchbaum nicht praktikabel, weshalb wir für diese Bäume heuristische Suchverfahren benötigen. Diese Verfahren durchsuchen typischerweise nicht jeden Pfad des Baumes, sondern versuchen einen Knoten solange zu verbessern, bis eine Lösung gefunden wird. Dadurch sind diese Algorithmen in der Regel nicht mehr vollständig. Wir betrachten das Problem zunächst vereinfacht über Gleichungen:

Eingabe:	Zwei Gleichungen $(u = v)$ und $(u' = v')$.
Ausgabe:	Eine Halbgruppe $S \in [\![u = v]\!] \setminus [\![u' = v']\!]$.

Sei $S \in \mathbf{Sgp}$. Wir interessieren uns für die folgenden zwei Fälle:

(i) Falls $S \notin [\![u = v]\!]$ ist, so gibt es eine Auswertung φ mit $\varphi(u) \neq \varphi(v)$. Laut Lemma 1 müssen Elemente aus S entfernt, damit die Gleichung für S wieder erfüllbar wird. Demnach müssen wir ein $x \in \varphi(\Sigma)$ entfernen, da die Auswertung φ dann nicht mehr existiert. Durch diese Änderung bleiben bestehende Auswertungen, in denen x nicht vorkommt, erhalten. Wir bezeichnen diese Elemente mit $\mathsf{Discard}(S)$.

(ii) Falls $S \notin [\![u' = v']\!]$ ist, so gibt es ebenso eine Auswertung φ' mit $\varphi'(u') \neq \varphi'(v')$. Von den Elementen der Menge $\varphi'(\Sigma')$ möchten wir also nach Möglichkeit mindestens eins beibehalten. Diese Elemente bezeichnen wir mit $\mathsf{Keep}(S)$.

Wir können diese Idee verallgemeinern, indem wir in der Eingabe des Problems zusätzlich zwei Algorithmen Discard und Keep voraussetzen, wobei für alle $S \in \mathbf{Sgp}$ gelten soll, dass $\chi_1(S) = 1 \Leftrightarrow \mathsf{Discard}(S) \neq \emptyset$ ist. So können wir obiges Problem beispielsweise auf beliebig viele Gleichungen erweitern, indem wir die obigen Mengen Discard und Keep über alle Gleichungen bilden und vereinigen.

5.1. Lokale Suche

Lokale Suchalgorithmen sind Verfahren zur Lösung von Optimierungsproblemen. Sie versuchen, wie oben beschrieben, einen Knoten durch lokale Änderungen solange, bezüglich einer Zielfunktion, zu verbessern, bis eine Lösung gefunden wird. Ein großes

Problem dieser Algorithmen sind lokale Optima der Zielfunktion, wodurch es passieren kann, dass der Algorithmus den aktuellen Knoten nur noch verschlechtern kann, dieser aber noch nicht optimal ist. Deshalb sind diese Verfahren meistens randomisiert. Für eine detailierte Einführung verweisen wir auf [5]. Bevor wir den Algorithmus angeben, definieren wir eine Abbildung eines Elementes auf die jeweiligen Erzeuger:

$$\mathrm{Gen}_A(x) := \{a_i \mid \exists\, (a_1, \ldots, a_{i-1}, a_i, a_{i+1} \ldots, a_m) \in A^m \colon x = a_1 \cdot \cdots a_m\}$$

Diese können wir leicht berechnen, indem wir in Algorithmus 3.1 die jeweiligen Erzeuger mitspeichern. Selman et al. [9] haben einen sehr wichtigen lokalen Suchalgorithmus für das Erfüllbarkeitsproblem der Aussagenlogik entwickelt. Wir verfolgen hier einen ganz ähnlichen Ansatz:

Algorithmus 5.1 : Lokale Suche

```
function Local-Search(n, χ1, χ2, Discard, Keep)
    for i ← 1 to MAX-TRIES do
        A ←$ P(T_n)
        for j ← 1 to MAX-FLIPS do
            S ← ⟨A⟩
            if χ1(S) = 1 ∧ χ2(S) = 0 then
                return S
            X ← A
            if Discard(S) ≠ ∅ then
                E ← Discard(S) \ Keep(S)
                with probability GREED-RATE and E ≠ ∅ do
                    d ←$ E
                else
                    d ←$ Discard(S)
                X ← Gen_A(d)
            x ←$ X
            σ ←$ T_n
            A ← A \ {x} ∪ {σ}
    return null
```

Wir starten zunächst mit einer zufälligen Erzeugermenge $A \subseteq \mathcal{T}_n$. Die grundlegende Idee ist es nun, jene Erzeuger auszutauschen, die dafür verantwortlich sind, dass Elemente in Discard(S) entstehen. Dabei bevorzugen wir Elemente, die zudem nicht in Keep(S) sind. Finden wir nach einer Weile keine Lösung, so erzeugen wir eine neue Erzeugermenge und wiederholen die Suche. Der Algorithmus ist trivial parallelisierbar, da die äußeren Iterationen unabhängig voneinander ausgeführt werden können.

5.2. Genetische Suche

Genetische Algorithmen sind heuristische Optimierungsverfahren, deren Strategie von der biologischen Evolution inspiriert sind. Diese Verfahren verwalten eine Population von *Genomen*, welche in unserem Fall Teilmengen von $\mathcal{T}_n$ sind. Aus dieser Population werden neue Genome erzeugt, indem man zwei Elterngenome zu einem neuen Genom *rekombiniert*. Zudem werden diese zufällig *mutiert*, um eine Diversität der Population sicherzustellen. Die Qualität eines Genomes wird mit einer Bewertungsfunktion gemessen. Ein Überblick über diese Verfahren findet sich in [10].

Zunächst benötigen wir eine geeignete Bewertungsfunktion $f\colon \mathbf{Sgp} \to [0,1]$. Eine naheliegende Wahl hierfür ist

$$f(S) := \frac{1}{2}\left(\frac{|S| - |\mathsf{Discard}(S)|}{|S|} + (1 - \chi_2(S))\right)$$

Offensichtlich gilt $f(S) = 1 \Leftrightarrow (\chi_1(S) = 1 \wedge \chi_2(S) = 0)$. Der linke Term gibt an, wieviele Elemente die Eigenschaft χ_1 verletzen und der rechte Term gibt an, ob die Eigenschaft χ_2 nicht erfüllt ist. Da wir beide Bedingungen gleichermaßen erfüllen müssen, gewichten wir sie gleich. Das Grundgerüst von genetischen Algorithmen ist immer sehr ähnlich und sieht bei uns so aus:

Algorithmus 5.2 : Genetische Suche

```
function Genetic-Search(n, m, f, Discard, Keep)
    P ← list of m random subsets of T_n
    for i ← 1 to MAX-TRIES do
        P' ← Next-Population(P)
        while |P'| < k do
            A ← Select(P)
            B ← Select(P)
            C ← Crossover(A, B, Discard, Keep)
            with probability MUT-RATE do
                Mutate(C)
            Push-Back(P', C)
        P ← P'
        Measure(P, f)
        Sort(P, f, ≥)
        if ∃ A ∈ P: f(⟨A⟩) = 1 then
            return ⟨A⟩
    return null
```

Wir haben also viele Freiheitsgrade und müssen noch die Funktion Next-Population, Select, Crossover und Mutate festlegen. Eine beliebte Wahl für Next-Population ist das

sogenannte *Elitismus-Modell*. Hierbei wird ein kleiner Anteil der besten Genome aus der aktuellen Population in die nächste Generation unverändert übernommen. Dadurch kann die Qualität der besten Genome offensichtlich nicht schlechter werden.

Die Select-Funktion wählt jeweils ein Elternteil für die Rekombination aus. Wir entscheiden uns hier für die *binäre Turnierselektion*. Dabei werden zwei Genome zufällig aus der Population gezogen und das bessere Genom ausgewählt. Diese Bevorzugung hat die Suche deutlich beschleunigt.

Der Mutate-Operator in genetischen Algorithmen erlaubt es lokalen Optima zu entkommen und stellt die Diversität der Population sicher. Um dies zu erreichen tauschen wir einfach ein Element $x \in A$ durch ein neues zufälliges Element $x' \in \mathcal{T}_n$ aus.

Die wichtigste Komponente des Verfahrens ist der Crossover-Operator. Dieser soll nach Möglichkeit die guten Eigenschaften zweier Eltern A und B auf das Kind C übertragen. Wir wählen das Kind C so, dass

$$C \subseteq ((S \setminus \mathsf{Discard}(S)) \cup (T \setminus \mathsf{Discard}(T)) \cup \mathsf{Keep}(S) \cup \mathsf{Keep}(T))$$

mit $S = \langle A \rangle$ und $T = \langle B \rangle$ ist. Damit eliminieren wir alle Elemente aus $\mathsf{Discard}(S)$ und versuchen eine Halbgruppe mit den „guten“ Elementen als Erzeuger zu konstruieren.

6. Implementierung

Wir beschreiben nun einige Details unserer Implementierung der vorgestellten Algorithmen. Die Algorithmen wurden innerhalb des Amx-Projekts [3] in C++17 implementiert[1]. Amx ist ein Framework zur algorithmischen Behandlung von Monoiden und Automaten. Der Programmcode ist in den Dateien `find.h`, `variety.h` und `amx-find.cc` zu finden. Dabei wurde besonders viel Wert auf eine effiziente und generische Implementierung der Algorithmen gelegt. Demnach besitzen alle Algorithmen eine Funktion als Eingabeparameter, welche die gesuchte Eigenschaft spezifiziert. Eine Eigenschaft kann somit durch beliebigen Code beschrieben werden. Außerdem wurden alle Algorithmen mit Ausnahme der Breitensuche parallelisiert.

Das Kommadozeilenprogramm `amx-find` bietet einen einfachen Zugriff zu den Algorithmen und den vorgefertigten Eigenschaften. Es unterstützt die Befehle `variety` und `trotter-weil`. Der erste Befehl nimmt als Eingabe zwei Listen $(u_i = v_i)_{i\in I}$ und $(u'_j = v'_j)_{j\in J}$ von Gleichungen und sucht eine Halbgrupppe S, für die

$$S \in [\![(u_i = v_i)_{i\in I}]\!] \setminus [\![(u'_j = v'_j)_{j\in J}]\!]$$

gilt. Das genaue Eingabeformat findet sich in Form einer kontextfreien Grammatik im Anhang A. Um die Auswertungsgeschwindigkeit der Gleichungen zu beschleunigen, werden gemeinsame Teilausdrücke eliminiert und nur einmal ausgerechnet. Der zweite Befehl hingegen sucht für eine gegebene Tiefe $d \geq 2$ einen Vertreter

$$M \in (\mathbf{R}_{d+1} \cap \mathbf{L}_{d+1}) \setminus (\mathbf{R}_d \cap \mathbf{L}_d)$$

aus der jeweiligen Schnittebene der Trotter-Weil-Hierarchie. Hierzu benötigen wir für die deterministischen Algorithmen ein Entscheidungsverfahren, das prüft, ob ein Monoid M in der jeweiligen Schnittebene liegt. Die Definition über das Mal'cev-Produkt liefert direkt ein solches Verfahren: Wir starten zunächst mit den zwei Mengen $R_1 := L_1 := M$. Daraus berechnen wir iterativ für alle $i \in \{1, \ldots, d\}$ die Äquivalenzklassen $R_{i+1} := L_i/\sim_K$ und $L_{i+1} := R_i/\sim_D$ und überprüfen dann, ob die Monoide R_{d+1} und L_{d+1} beide $\mathcal{J}$-trivial sind und ob eines der beiden Monoide R_d oder L_d es nicht ist. Für die heuristischen Algorithmen setzen wir

$$\begin{aligned} \mathsf{Discard}(M) &:= \{x \in X \mid X \in R_{d+1}/\mathcal{J} : |X| > 1\} \\ \mathsf{Keep}(M) &:= M \setminus \mathsf{Discard}(M) \end{aligned}$$

[1] `https://theogit.fmi.uni-stuttgart.de/amx/amx`

wodurch wir Elemente bevorzugen, die in einer eigenen $\mathcal{J}$-Klasse liegen. Für alle Befehle kann man zusätzlich angeben, ob man eine Halbgruppe oder ein Monoid sucht. Zudem kann die Anzahl der Elemente sowie die Anzahl der Erzeuger beschränkt werden. Wird eine Halbgruppe gefunden, so wird die serialisierte Verknüpfungstafel ausgegeben. Außerdem gibt es die Option `--minimize`, die versucht ein minimales Beispiel mittels Bisektion zu finden.

7. Experimentelle Ergebnisse

In diesem Kapitel bewerten wir die in Kapitel 4 und 5 beschriebenen Verfahren. Dazu führen wir zunächst Beispiele an, die von unserer Implementierung gefunden werden. Anschließend vergleichen wir die Algorithmen mithilfe eines Benchmarks über der Trotter-Weil-Hierarchie.

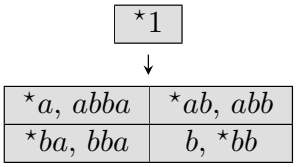
7.1. Einfache Beispiele

Dieser Abschnitt zeigt zwei einfache Beispiele, die von allen Algorithmen problemlos gefunden werden. Geben wir die Varietät $[\![x^\omega = 1]\!] \setminus [\![xy = yx]\!]$ ein, so findet das Programm die Verknüpfungstafel

$\cdot$	1	a	ab	b	ba	bb
1	1	a	ab	b	ba	bb
a	a	1	b	ab	bb	ba
ab	ab	bb	1	ba	b	a
b	b	ba	a	bb	ab	1
ba	ba	b	bb	a	1	ab
bb	bb	ab	ba	1	a	b

Dies beschreibt offenbar eine nicht-abelsche Gruppe mit sechs Elementen, wodurch diese isomorph zur symmetrischen Gruppe S_3 ist.

Ein weiteres Beispiel ist die Varietät **DS** \ **DO**. Hier findet das Programm ein Monoid M mit den Greenschen Relationen:



Es gilt $M \in$ **DS**, da die beiden regulären $\mathcal{D}$-Klassen $\{1\}$ und $\{a, b, ab, ba, abb, bba, abba\}$ jeweils Unterhalbgruppen von M sind. Zudem gilt aber $M \notin$ **DO**, da in der zweiten $\mathcal{D}$-Klasse die idempotenten Elemente a, ab, ba, bb keine Unterhalbgruppe bilden.

7.2. Trotter-Weil-Hierarchie

Wir vergleichen nun die Laufzeiten der Algorithmen mithilfe eines Benchmarks über die Schnittebenen der Trotter-Weil-Hierarchie. Dazu beschreiben wir zunächst die Testumgebung des Benchmarks. Alle Algorithmen wurden bis zu einer Tiefe $d = 7$ jeweils zehnmal ausgeführt, wobei die maximale Suchzeit drei Minuten beträgt. Alle Algorithmen bis auf die Breitensuche wurden parallel ausgeführt. Details finden sich in dem Skript `trotter-weil-bench.py`. Der Benchmark wurde auf einem PC mit einem Intel i5-4460 und 16 GB RAM ausgeführt. Das Betriebssystem ist Windows 10 und der verwendete Compiler ist MSVC++ 14.13. Die Ergebnisse finden sich in Abbildung 7.1 als logarithmischer Plot sowie im Anhang B.

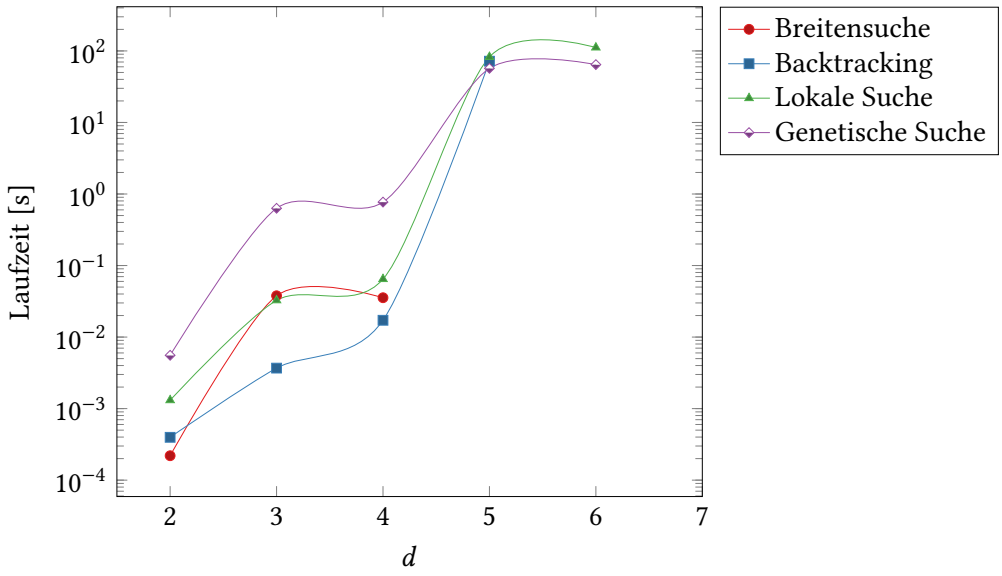


Abbildung 7.1.: Laufzeitenplot des Benchmarks

Wir sehen, dass die Breitensuche nur bis $d = 4$ terminiert. Damit eignet sie sich, wie erwartet, nur für kleine Beispiele. Backtracking findet sehr schnell eine Lösung, stößt aber bei $d = 5$ an seine Grenzen. Lokale Suche und genetische Suche finden auch noch bis $d = 6$ eine Lösung, wobei die lokale Suche schneller als die genetische Suche ist. Zudem konnten wir mit der lokalen Suche sowie der genetischen Suche Beispiele bis $d = 8$ finden. Diese sind aber aufgrund der hohen Ausführungszeit nicht Teil des Benchmarks.

8. Zusammenfassung und Ausblick

Wir haben das in der Einleitung beschriebene Problem formalisiert und näher untersucht. So konnten wir das Problem mithilfe des Satzes von Cayley als Suchproblem formulieren und somit erste Algorithmen finden. Dabei stellte sich heraus, das paralleles Backtracking bei kleinen Suchräumen erstaunlich schnell zu einer Lösung kommt. Dennoch stößt das Verfahren bei großen Beispielen schnell an seine Grenzen. Dies zeigte uns die Notwendigkeit von heuristischen Suchverfahren. Demnach haben wir einen lokalen Suchalgorithmus und einen genetischen Algorithmus entworfen, die auch bei größeren Suchräumen noch Beispiele finden.

Es wäre sehr hilfreich zu untersuchen, ob man alle Halbgruppen effizienter enumerieren kann. Insbesondere wäre eine Aufzählung ohne Duplikate wünschenswert, da es signifikant weniger Unterhalbgruppen als Teilmengen eines Transformationsmonoids gibt. Darüber hinaus gibt es noch viel Optimierungspotential bei den heuristischen Algorithmen. Gerade bei der genetischen Suche wäre es interessant zu wissen, ob es noch bessere genetische Operatoren gibt, die eine schnellere Konvergenz der Suche ermöglichen.

A. Grammatik der Varietätsbedingungen

Wir geben nun das Eingabeformat der Varietätsbedingungen als EBNF an:

⟨*condition*⟩ ::= ⟨*variety*⟩ ‘\’ ⟨*variety*⟩
| ⟨*variety*⟩
| ‘\’ ⟨*variety*⟩

⟨*variety*⟩ ::= ‘[’ ⟨*equation*⟩ { ‘,’ ⟨*equation*⟩ } ‘]’

⟨*equation*⟩ ::= ⟨*expression*⟩ ‘=’ ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*factor*⟩ { [‘∗’] ⟨*factor*⟩ }

⟨*factor*⟩ ::= ⟨*term*⟩ [‘^’ ⟨*exponent*⟩]

⟨*term*⟩ ::= ⟨*variable*⟩
| ‘1’
| ‘(’ ⟨*expression*⟩ ‘)’

⟨*exponent*⟩ ::= ⟨*integer*⟩
| ‘w’

⟨*variable*⟩ ::= ⟨*letter*⟩ [‘_’ ⟨*integer*⟩]

B. Messwerte des Benchmarks

Auf den folgenden Seiten finden sich für jeden Algorithmus die genauen Daten des Benchmarks über der Trotter-Weil-Hierarchie. Die Laufzeiten sind dabei in Sekunden angegeben und ein „-“ in einer Zelle bedeutet, dass der Algorithmus nach drei Minuten kein Ergebnis geliefert hat.

Tiefe	Laufzeit									
2	2.42E−1	2.00E−1	2.15E−1	2.06E−1	2.68E−1	2.29E−1	2.10E−1	2.06E−1	2.10E−1	2.09E−1
3	3.94E1	3.70E1	3.70E1	3.68E1	3.74E1	3.71E1	3.71E1	3.72E1	4.02E1	3.90E1
4	3.66E1	3.49E1	3.57E1	3.52E1	3.51E1	3.59E1	3.51E1	3.70E1	3.43E1	3.52E1
5	-	-	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-

Tabelle B.1.: Breitensuche

Tiefe	Laufzeit									
2	5.70E−4	3.80E−4	4.31E−4	4.66E−4	3.72E−4	3.78E−4	3.67E−4	3.36E−4	3.73E−4	2.96E−4
3	4.94E−3	5.57E−3	3.31E−3	3.36E−3	3.22E−3	3.35E−3	3.28E−3	3.54E−3	3.11E−3	3.14E−3
4	1.69E−2	2.14E−2	1.60E−2	1.71E−2	1.65E−2	1.71E−2	1.63E−2	1.71E−2	1.68E−2	1.65E−2
5	4.49E1	8.21E1	7.66E1	4.73E1	8.65E1	4.92E1	8.71E1	8.30E1	8.45E1	7.73E1
6	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-

Tabelle B.2.: Backtracking

Tiefe	Laufzeit									
2	1.39E−3	1.13E−3	1.15E−3	1.46E−3	1.21E−3	1.24E−3	1.57E−3	1.30E−3	1.34E−3	1.29E−3
3	5.44E−2	4.20E−2	5.48E−2	1.40E−2	4.23E−2	4.91E−2	1.96E−3	4.03E−2	1.92E−2	8.00E−3
4	5.46E−3	2.34E−3	2.99E−2	1.08E−2	1.20E−1	8.52E−2	4.56E−2	1.86E−1	1.39E−1	2.20E−2
5	1.34E2	-	6.71E1	-	2.45	-	5.37E1	-	-	1.54E2
6	1.72E2	-	1.79E2	-	-	-	-	-	9.29E1	1.88
7	-	-	-	-	-	-	-	-	-	-

Tabelle B.3.: Lokale Suche

Tiefe	Laufzeit									
2	7.16E−3	5.01E−3	5.51E−3	5.30E−3	4.41E−3	5.84E−3	4.28E−3	5.83E−3	7.18E−3	5.04E−3
3	1.74	1.97E−1	1.69	3.70E−1	7.73E−3	3.95E−1	6.71E−2	2.64E−2	1.13	6.94E−1
4	6.84E−2	2.58E−1	1.33E−2	2.91E−2	8.99E−2	1.38E−2	3.81	2.73	6.61E−1	2.71E−2
5	-	-	6.12E1	9.59E1	-	-	-	1.42E1	-	-
6	7.37	2.84E1	4.93E1	-	5.09	1.24E2	-	2.96E1	1.70E2	1.02E2
7	-	-	-	-	-	-	-	-	-	-

Tabelle B.4.: Genetische Suche

Literatur

[1] J. Almeida. *Finite Semigroups and Universal Algebra*. Bd. 3. Series in Algebra. World Scientific, 1994.

[2] S. Eilenberg und B. Tilson. *Automata, Languages, and Machines*. Bd. 59. Pure and Applied Mathematics. Academic Press, 1976.

[3] L. Fleischer, O. Naumann und V. Baldini. „A Toolkit for Manipulation of Automata and Monoids“. Studienarbeit. Institut für Formale Methoden der Informatik, Universität Stuttgart, 2013.

[4] V. Froidure und J.-É. Pin. „Algorithms for computing finite semigroups“. In: *Foundations of Computational Mathematics*. Hrsg. von F. Cucker und M. Shub. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, S. 112–126.

[5] H. Hoos und T. Stützle. *Stochastic Local Search: Foundations and Applications*. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science, 2004.

[6] M. Kufleitner und P. Weil. „On logical hierarchies within FO^2-definable languages“. In: *Logical Methods in Computer Science* 8.3 (Aug. 2012).

[7] J.-É. Pin. *Varieties Of Formal Languages*. Hrsg. von R. E. Miller. Plenum Publishing Co., 1986.

[8] J. Reiterman. „The Birkhoff theorem for finite algebras“. In: *algebra universalis* 14.1 (Dez. 1982), S. 1–10.

[9] B. Selman, H. A. Kautz und B. Cohen. „Local Search Strategies for Satisfiability Testing“. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. 1996, S. 521–532.

[10] M. Srinivas und L. M. Patnaik. „Genetic algorithms: a survey“. In: *Computer* 27.6 (Juni 1994), S. 17–26.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift