

BACHELORTHESIS
Adrian Helberg

Template-basierte Synthese von Verzweigungsstrukturen mittels L-Systemen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Adrian Helberg

Template-basierte Synthese von Verzweigungsstrukturen mittels L-Systemen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Michael Neitzke

Eingereicht am: 1. März 2021

Adrian Helberg

Thema der Arbeit

Template-basierte Synthese von Verzweigungsstrukturen mittels L-Systemen

Stichworte

Verzweigungsstruktur, 2D Generierung, L-System, Template, Prozedurale Modellierung, Inverse Prozedurale Modellierung, Baumstruktur, Formale Grammatik, Ähnlichkeit

Kurzzusammenfassung

Prozedurale Modellierung beschreibt effiziente Methoden zur Erzeugung einer Vielzahl an Modellen nach bestimmten Regeln. Die Erstellung eines Systems zur Umsetzung solcher Methoden ist durch einen Mangel an Kontrolle und einer geringen Vorhersagbarkeit der Ergebnisse erschwert. Diese Bachelorarbeit präsentiert ein System zur Synthese von Verzweigungsstrukturen, die einer benutzerdefinierten Struktur ähnlich sind. Dabei wird gezeigt, wie sich aktuelle Ansätze der Forschung in einem Programm umsetzen lassen. Templates werden eingelesen und vom Benutzer zu einer Basisstruktur organisiert. Anschließend wird ein L-System über die Topologie dieser Struktur inferiert, komprimiert und dann generalisiert. Nach der Interpretation des L-Systems können vom Benutzer gesetzte Transformationsparameter aus einer Häufigkeitsverteilung angewendet werden. Zum Schluss wird die resultierende Verzweigungsstruktur visualisiert.

Adrian Helberg

Title of Thesis

Template-based synthesis of branching structures using L-Systems

Keywords

Branching Structure, 2D Generation, L-System, Template, Procedural Modeling, Inverse Procedural Modeling, Tree Structure, Formal Grammar, Similarity

Abstract

Procedural modeling describes efficient methods for creating various models after certain rules. Setting up a system, that implements such methods, lacks of controllability and predictability of the results and is therefore a difficult task. This bachelor thesis presents a system for synthesizing branching structures that are similar to custom structures created by a user. It is shown how current research can be implemented into a program. Templates are used to be organized into a basic structure. Then the topology gets inferred, compressed and generalized into an L-System. After deriving the L-System, custom transformation parameters from a frequency distribution can be applied. Finally, the resulting branching structure is visualized.

Inhaltsverzeichnis

Abbildungsverzeichnis	vii
Abkürzungen	ix
Symbolverzeichnis	x
Listings	xi
1 Einleitung	1
1.1 Problemstellung	2
1.2 Ziele	3
1.3 Methodik	3
1.4 Aufbau	5
2 Grundlagen	6
2.1 Grundbegriffe	6
2.2 Grundlegende Arbeiten	7
2.3 Verwandte Arbeiten	13
3 Konzepte	14
3.1 Probleme & Lösungsansätze	14
3.2 Workflows & Algorithmen	17
3.3 Softwarearchitektur	23
4 Implementierung	30
4.1 Projektstruktur	30
4.2 Technologien	32
4.3 Konzeptumsetzung	33
5 Evaluierung	57

6 Fazit und Ausblick	66
Literaturverzeichnis	68
Glossar	71
Selbstständigkeitserklärung	72

Abbildungsverzeichnis

1.1	Systemarchitektur	5
3.1	System und Systemumgebung	24
3.2	Interaktion zwischen System und Systemumgebung	24
3.3	Subsysteme mit fachlichen Abhängigkeiten	26
3.4	Subsystem GUI	27
3.5	Laufzeitsicht	28
3.6	Infrastruktur Windows-PC	28
4.1	Softwareprojekt Dateistruktur	31
4.2	Softwarepakete mit zugehörigen Funktionen	31
4.3	Startskript Generator.bat	33
4.4	Programm nach Ausführung des Start-Skripts	33
4.5	Tempaltes-Datei zum Einlesen der Template-Strukturen	34
4.6	Erster Anker ist vorselektiert & gesetzte Parameter	35
4.7	Auswahl des ersten Templates	35
4.8	Der Verzweigungsstruktur hinzugefügte Template-Instanz	36
4.9	Vom Benutzer fertiggestellte Verzweigungsstruktur	37
4.10	Generierte Verzweigungsstrukturen (1/2)	37
4.11	Generierte Verzweigungsstrukturen (2/2)	38
4.12	Verzweigungsstruktur & zugehörige Baumstruktur	41
5.1	templates.txt	58
5.2	Grafische Benutzeroberfläche nach der Erstellung einer Basisstruktur	59
5.3	Baumstruktur der erstellten Verzweigungsstruktur	60
5.4	Inferriertes L-System	61
5.5	Komprimierter Baum	62
5.6	Synthetisierte Verzweigungsstrukturen	64

5.7 Untersuchung der synthetisierten Verzweigungsstrukturen	65
---	----

Abkürzungen

HAW Hochschule für Angewandte Wissenschaften.

Symbolverzeichnis

Ω unit of electrical resistance.

Listings

3.1	Erstellen einer Verzweigungsstruktur	17
3.2	Inferieren eines L-Systems aus einer Baumstruktur	18
3.3	Erstellen eines kompakten L-Systems mit Gewichtung w_l	20
3.4	Kostenfunktion C_i mit Gewichtung w_l	20
3.5	Längenfunktion L für Grammatiken	21
3.6	Grammar Edit Distance	21
3.7	Kostenfunktion C_g mit Gewichtung w_0	21
3.8	Generalisieren eines L-Systems mit Gewichtung w_0	22
4.1	Klasse TreeNode zur Erstellung einer Baumstruktur	39
4.2	Klasse TreeNodeIterator als Iterator für die Baumstruktur	40
4.3	Pipeline Klasse zur Organisation von Prozessen	42
4.4	Pipe Interface als Vorlage zur Erstellung eines Teilprozesses einer Pipeline	42
4.5	Erstellen des Pipeline Kontextes	43
4.6	Inferer Klasse zur Inferierung eines L-Systems aus einer Baumstruktur	43
4.7	InfererPipe Klasse als Teilprozess der Pipeline	44
4.8	Inferierungsalgorithmus der Inferer Klasse	45
4.9	Klasse Compressor zur Komprimierung eines L-Systems	46
4.10	Komprimierungsalgorithmus der Compressor Klasse	47
4.11	Algorithmus zum Finden eines maximalen Unterbaums	48
4.12	Algorithmus zum Finden alle Vorkommen eines Unterbaums in einem Baum	49
4.13	Funktion zur Berechnung der Kosten eines L-Systems	50
4.14	Funktion zur Ermittlung der Anzahl Anwendung einer Produktionsregel	50
4.15	Generalizer Klasse zur Generalisierung eines L-Systems	51
4.16	Generalisierungsalgorithmus der Generalizer Klasse	51
4.17	Kostenfunktion zum Vergleich zweier L-Systeme	52
4.18	Längenfunktion über ein L-System	52
4.19	Grammar Edit Distance	53

4.20	String Edit Distance	54
4.21	Modules Klasse mit Funktion zur Ermittlung der String Edit Distance . .	54
4.22	Estimator Klasse zur Erstellung einer Verteilung über Transformationspa- rameter	55
4.23	Verteilungsalgorithmus der Estimator Klasse	55
4.24	Abrufen eines zufälligen Parameters aus der Verteilung für ein Template .	56
4.25	Berechnung des Durchschnitts eines Parameters für ein Template	56

1 Einleitung

Hochschule für Angewandte Wissenschaften (HAW) Ω HAW Hamburg

(TODO: Remove; Damit das Glossar keine Fehler schmeißt -.-)

Effizientes Objektdesign und -modellierung sind entscheidende Kernkompetenzen in verschiedenen Bereichen der digitalen Welt. Da die Erstellung geometrischer Objekte für Laien unintuitiv ist und ein großes Maß an Erfahrung und Expertise erfordert, ist dieses stetig wachsende Feld für Neueinsteiger nur sehr schwer zu erschließen. Die Forschung liefert hierzu einige Arbeiten zur prozeduralen Modellierung, um digitale Inhalte schneller und automatisiert zu erstellen. Gerade wenn es um die Darstellung natürlicher Umgebung geht, ist die Erstellung von ähnlichen Objekten, wie zum Beispiel verschiedene Bäume derselben Gattung eines Waldes, ein schwieriges Problem. Kleine Änderungen in prozeduralen Systemen können zu großen Veränderungen der Ergebnisse führen. Darum beschäftigt sich die inverse prozedurale Modellierung unter anderem mit dem Inferieren von Regeln und Mustern aus gegebenen Objekten, um diese nach bestimmten Regeln zu modellieren. Ein wichtiges Werkzeug hierbei ist die Verwendung formaler Grammatiken als fundamentale Datenstruktur der Informatik, um Strukturen zu beschreiben. Eine spezielle Untergruppe sind die L-Systeme, die häufig bei der Beschreibung von Verzweigungsstrukturen und Selbstähnlichkeit zum Einsatz kommen.

Diese Arbeit soll sich mit der Erstellung eines prozeduralen Systems zur Synthese von Ähnlichkeitsabbildern beschäftigen. Hierzu soll über eine Benutzerschnittstelle eine Struktur erzeugt werden, aus der ein parametrisiertes L-System inferiert werden kann, welches dann zur Generierung von ähnlichen Strukturen genutzt werden kann.

1.1 Problemstellung

Mit der Digitalisierung der Welt steigt auch der Bedarf an digitalen Inhalten. Zu den größten Feldern gehören Gaming- und Unterhaltungsindustrie, Datenvisualisierung und interaktive Anwendungen. Um eine erhöhte Quantität dieser Inhalte liefern zu können, werden Methodiken und Algorithmen gesucht, die eine Erstellung vereinfachen. Während Methodiken zur Kodifizierung bestimmter Strukturen in den Bereich der prozeduralen Modellierung fällt, findet das Ableiten von Regeln Anwendung in der inversen prozeduralen Modellierung. Weiter steigt mit dem digitalen und naturwissenschaftlichen Fortschritt die Anwendung immer komplexerer Strukturen, die ein Herausarbeiten der schwer zu kontrollierenden, prozeduralen Regeln immer schwieriger machen.

Ein Beispiel hierzu aus der Gaming-Industrie ist die frühere Verwendung unorganisierter Modelle. Unorganisierte Modelle sind nur bedingt wiederverwendbar, da nur die vorliegende Modellierung verwendet werden kann. Es besteht eine hohe Speicherkomplexität bei geringer Laufzeitkomplexität. Für kleinste Veränderungen am Modell ist eine erneute Modellierung nötig, die wiederum Speicher benötigt, um sie persistent speichern zu können. Heutzutage werden die Objekte nach bestimmten Kriterien organisiert, um eine automatisierte Modellierung durch Algorithmen zu ermöglichen, um so aus einer Grundstruktur weitere Modelle zu erzeugen. Speicher- und Laufzeitkomplexität nähern sich an. Deshalb werden allgemeingültige, vielseitig anwendbare Algorithmen gesucht, die bestimmte natürliche Eigenschaften von Strukturen herausarbeiten (Reverse Engineering), um diese für die (inverse) prozedurale Modellierung zur Verfügung zu stellen.

Diese Arbeit soll zeigen, wie sich durch die Erstellung eines Systems zur Generierung von ähnlichen Strukturen aus einer Basistruktur, aktuelle Ansätze aus der Forschung in einem Programm umsetzen lassen.

1.2 Ziele

Die Erstellung eines Systems zur Synthese von ähnlichen Strukturen aus einer Basisstruktur soll zentrale Aufgabe dieser Arbeit sein. Aus der Fragestellung leiten sich folgende Teilziele ab:

- Die Erstellung eines Programms zur Umsetzung des erstellten Systems
- Anwenden von Algorithmen und Ansätzen der aktuellen Forschung
- Testen von Metriken und deren Auswirkung auf das Ergebnis
- Erstellung eigener Methodiken und Algorithmen zur effizienten Lösung der Problemstellung
- Schaffen eines Teilsystems zum Extrahieren von Eigenschaften und Regeln einer Basisstruktur

Die Erstellung und Integration eines neuronalen Netzes, das laut aktueller Forschung gute Ergebnisse beim Lernen von Regeln aus einer Eingabestruktur liefert, kann in dieser Arbeit aus Praktikabilitätsgründen nicht behandelt werden.

1.3 Methodik

Der Benutzer des Systems legt atomare Strukturen in Form von Zeichenketten an, die vom Programm eingelesen und als Templates zur Verfügung gestellt werden. Die Templates sind beliebig und können eine einfache Linie oder eine komplexe Verzweigung darstellen. Werden diese Strukturen auf der graphischen Oberfläche platziert und mit diversen Transformationen verändert, spricht man von Instanzen oder Template-Instanzen.

Strukturieren

Der Benutzer nutzt die grafische Benutzerschnittstelle, um aus einzelnen Templates eine zusammenhängende Basisstruktur zu erstellen. Neben der Position der Instanzen werden Transformationsparameter, wie Rotation oder Skalierung, angepasst.

Visualisieren

Der aktuelle Stand der Strukturierung ist jederzeit sichtbar. Liniensegmente und Bindungselemente werden in einem graphischen Element visualisiert und für eine Interaktion zur Verfügung gestellt.

Datenaufbereitung

Das Ergebnis der Strukturierung wird in einer Baumstruktur organisiert, in der jeder Knoten einer bestimmten Template-Instanz entspricht und die eingehenden Kanten die geometrischen Transformationen relativ zum Elternknoten beschreibt.

Inferieren

Aus der Baumstruktur kann eine formale Grammatik in Form eines L-Systems abgeleitet werden. Diese Grammatik beschreibt lediglich die erstellte Basisstruktur und beinhaltet keine Transformationsparameter, da hier nur auf die Topologie des Baumes und nicht auf geometrische Unterschiede der Instanzen eingegangen wird.

Komprimieren

Um sich wiederholende Produktionsregeln zu vermeiden und so sowohl das Alphabet, als auch die Produktionsregelmenge zu kompromieren, wird die Baumstruktur nach identischen, maximalen Unterbäumen durchsucht und durch zusammengefasste Instanzen vereinfacht. Hierbei gilt die Baumstruktur selbst nicht als Unterbaum.

Generalisieren

Ähnliche Produktionsregeln des L-Systems werden mithilfe einer Kostenfunktion zusammengefasst, um diese mit nicht-deterministischen Regeln und Rekursion zu generalisieren. Die Kostenfunktion entscheidet hierbei über den Grad der Ähnlichkeit zweier Produktionsregeln

Randomisieren

Jedes Symbol der Grammatik nimmt eine Liste an Parametern entgegen, die nach bestimmten Kriterien pseudo-randomisiert werden, um Variationen von Template-Instanzen zu erstellen. Das Ausführen des L-Systems kann nun Ähnlichkeitsstrukturen für die Basisstruktur erzeugen.

1.4 Aufbau

Die Methodik zum Umsetzen des beschriebenen Systems wird wie folgt umgesetzt.

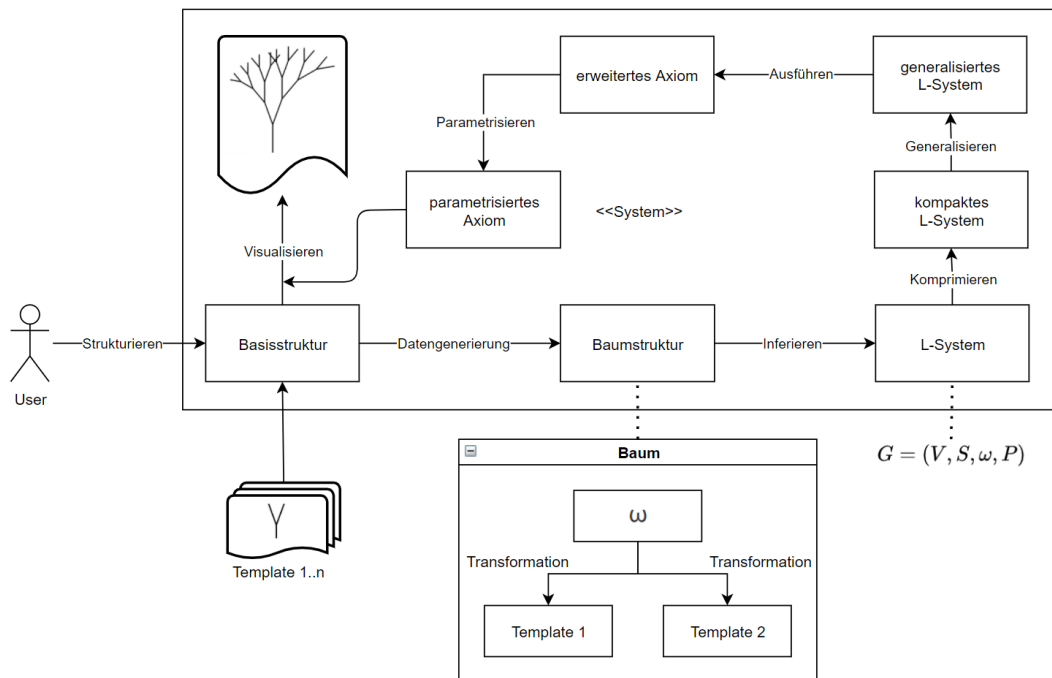


Abbildung 1.1: Architektur des Systems mit einigen Datenstrukturen

Die Darstellung zeigt eine grobe Übersicht der Anwendungsfälle innerhalb des Systems. Der Benutzer strukturiert vom System importierte Templates zu einer Basisstruktur mittels grafischer Bedienelemente. Die Template-Instanzen werden hierbei in einer Baumtopologie organisiert. Anschließend wird ein L-System aus der Baumstruktur generiert und komprimiert. Das kompakte L-System wird generalisiert und ausgeführt. Zuletzt wird das abgeleitete Axiom parametrisiert und dann sichtbar gemacht.

2 Grundlagen

Die Modellierung mithilfe von Grafiksoftware ist eine vergleichbar händische, langwierige Erstellung von Objekten. Hierbei hat der Designer (Modellierer) die volle Kontrolle über die Strukturen des Objektes.

Bei der prozeduralen Modellierung werden spezifische Strukturen eines zu erstellenden physikalischen Objektes generalisiert und meist über eine Grammatik und globale Parameter abgebildet. Während bei der klassischen Modellierung die menschliche Intuition und bei der prozeduralen Modellierung eine parametrisierte Grammatik vorausgesetzt wird, arbeitet die inverse prozedurale Modellierung mit bestehenden Modellen und extrahiert („lernt“) die Strukturen des Objektes, die automatisch in eine formale Grammatik überführt werden können. Die Generierung von prozeduralen Modellen ist ein wichtiges, offenes Problem [5].

2.1 Grundbegriffe

Modellierung

Um einen physikalischen Körper in ein digitales Objekt zu überführen, wird mithilfe von Abstraktion (Modellierung) ein mathematisches Modell erstellt, das diesen Körper formal beschreibt. 3D Grafiksoftware, wie bspw. Blender [26], wird genutzt um geometrische Körper zu modellieren, texturieren und zu animieren.

Prozedurale Modellierung

„It encompasses a wide variety of generative techniques that can (semi-)automatically produce a specific type of content based on a set of input parameters“ [21]

Prozedurale Modellierung beschreibt generative Techniken, die (semi-)automatisch spezifische, digitale Inhalte anhand von deskriptiven Parametern erzeugen (*Übersetzt durch den Autor*). Smelik u. a. beschreibt einen Prozess, welcher durch das Nutzen globaler Parameter und deskriptiven Regeln Modelle erzeugt [21].

Inverse prozedurale Modellierung

Aliaga u. a. spricht bei der inversen prozeduralen Modellierung von dem Finden einer prozeduralen Repräsentation von Strukturen bestehender Modelle [3]. Die Methodik aus Strukturen bestimmte Regeln und Parameter abzuleiten ist der Hauptgegenstand dieses Feldes der Computergrafik und aktueller Gegenstand der Forschung.

2.2 Grundlegende Arbeiten

Smelik u. a. untersucht prozedurale Methoden, um diverse Strukturen, wie Vegetation, Straßen u.v.m zu erzeugen. Es wird ein Überblick aktueller, vielversprechender Studien gegeben, deren Anwendung sowohl in technischen Bereichen, als auch in nicht-technischen, kreativen Bereichen, diskutiert wird. [21] gilt als grundlegender Einstiegspunkt in die Bereiche der prozeduralen Modellierung. Einen aktuellen Stand der Forschung der inversen prozeduralen Modellierung liefert [3]. Aliaga u. a. zeigt, dass IPM-Ansätze in entsprechende Kernprobleme der Informatik aufgeteilt und meist getrennt voneinander durch verschiedene Methodiken und Algorithmen bearbeitet werden. Weiter wird ein Einblick in die Kategorisierung und Bewertung einiger Ergebnisse von IPM-Systemen gegeben. Die Buchrezension [7] geht auf die Arbeit [10] ein und weist darauf hin, dass De la Higuera einen wesentlichen Unterschied zwischen der Induktion einer Grammatik, also das Finden einer Grammatik, welche ein Datum am genauesten beschreibt, und der Grammatikinferrierung, also das Finden einer Zielgrammatik, welche eine bestimmte Zeichenfolge abdeckt, sieht.

De la Higuera ordnet die inverse Generierung von L-Systemen zum Problem der Grammatikinferenz, welches er als gut erforschtes Gebiet beschreibt. Verzweigungsstrukturen im Kontext der inversen prozeduralen Modellierung tauchen in wissenschaftlichen Studien wenig auf. [9] adressiert diese in seiner Arbeit [9] und liefert einige Ansätze hierzu.

Im Folgenden wird eine Übersicht zu einigen Arbeiten zur Modellierung von bestimmten Strukturen gegeben.

Subjekt	Arbeit
Bäume und Landschaften	[8]
Fassaden	[2]
Gebäude	[15]
Städte	[18]
Möbel	[14]
Inneneinrichtung	[28]

L-Systeme

Lindenmayer führt eine mathematische Beschreibung zum Wachstum fadenförmiger Organismen ein. Sie zeigt, wie sich der Status von Zellen infolge ein oder mehrerer Einflüsse verhält [11]. Weiter führt er Ersetzungssysteme ein, die atomare Teile mithilfe von Produktionsregeln ersetzen. Diese L-Systeme nutzt er zur formalen Beschreibung von Zellteilung. Später werden Symbole zur formalen Beschreibung von Verzweigungen, die von Filamenten abgehen, eingeführt [20]. Die bekanntesten L-Systeme sind zeichenkettenbasiert und werden von *Noam Chomsky* in [6] eingeführt. Sie ersetzen parallel Symbole eines Wortes, die von einer Grammatik über eine Sprache akzeptiert werden. L-Systeme können unter anderem parametrisiert oder nicht-parametrisiert und kontextfrei oder kontextsensitiv sein.

L-Systeme sind Grammatiken mit folgender Form:

$$\mathcal{L} = \langle M, \omega, R \rangle, \text{ mit}$$

- M als Alphabet, das alle Symbole enthält, die in der Grammatik vorkommen,
- ω als Axiom oder „Startwort“ und
- R als Menge aller Produktionsregeln, die für \mathcal{L} gelten

Das Alphabet eines parametrisierten Systems enthält Module (Symbole mit Parametern) anstatt Symbole:

$$M = \{A(P), B(P), \dots\} \text{ mit}$$

- $P = p_1, p_2, \dots$ als Modulparameter

Zeichen des Alphabets, die Ziel einer Produktionsregel sind, heißen Variablen. Alle anderen Zeichen aus M sind Konstanten. Das Axiom ω ist eine nicht-leere Sequenz an Modulen aus M^+ mit

- M^+ als Menge aller möglichen Zeichenketten aus Modulen aus M

Produktionsregeln sind geordnete Paare aus Wörtern über dem Alphabet, die bestimmte Ersetzungsregeln umsetzen. Hierbei werden Symbole aus einem Wort, die einer rechten Seite (*engl. right hand side (RHS)*) einer Produktionsregel entsprechen, durch die linke Seite des Paares (*engl. left hand side (LHS)*) ersetzt. Sie sind folgendermaßen aufgebaut:

$$A(P) \rightarrow x, x \in M^*, \text{ mit}$$

- M^* als die Menge aller möglichen Zeichenketten von M inklusive der leeren Zeichenkette ε

Ist die RHS jeder Produktionsregel ein einzelnes Symbol und gibt es zu jeder Variablen eine Regel, spricht man von einem kontextfreien, andernfalls von einem kontextsensitiven L-System.

L-System Interpretation

Lindenmayer-Systeme können Worte über ihr Alphabet interpretieren. Dafür werden Symbole des Wortes, die Ziel einer Produktionsregeln sind, in Iterationen durch die RHS der Produktionsregeln ersetzt. Bei der Ausführung eines L-Systems wird kein beliebiges Wort interpretiert, sondern das in der Grammatik definierte Axiom.

Logo-Turtle-Algorithmus

Der Logo-Turtle-Algorithmus [19] setzt ein Vorgehen zur graphischen Beschreibung von L-Systemen, bei dem jeder Buchstabe in einem Wort einer bestimmten Zeichenoperation zugewiesen wird, um. So kann aus einem L-System ein grafisches Muster generiert werden, das mit einer Abfolge von Zeichenbefehlen an eine „Schildkröte“ gezeichnet wird. Das Triplett (x, y, θ) definiert den Status (State) der Schildkröte. Dieser setzt sich aus der aktuellen Position $\begin{pmatrix} x \\ y \end{pmatrix}$ und dem aktuellen Rotationswinkel θ , der die Blickrichtung bestimmt, zusammen.

Der Algorithmus kann als Komprimierung eines geometrischen Musters gesehen werden. Folgende Symbole mit zugehörigen Steuerungsbefehlen und Statusveränderung sind definiert:

Symbol	Steuerung	Statusveränderung
$F(d)$	Gehe vom derzeitigen Punkt p_1 d Einheiten in die Blickrichtung zu dem Punkt p_2 . Zeichne ein Liniensegment zwischen p_1 und p_2	ja
$+(\alpha)$	Setze neuen Rotationswinkel $\theta = \theta + \alpha$	ja
$-(\alpha)$	Setze neuen Rotationswinkel $\theta = \theta - \alpha$	ja
[Lege den aktuellen State auf einen Stack	nein
]	Hole den State vom Stack und überschreibe den aktuellen mit diesem	nein

Alles zwischen den Symbolen [und] wird als Verzweigung interpretiert.

Bsp. $FF[FF]F$ mit Verzweigung $[FF]$

Extreme Programming

Eine Fallstudie der Universität Karlsruhe [16] untersucht den Einsatz der Softwaretechnik Extreme Programming (XP) im Kontext der Erstellung von Abschlussarbeiten im Universitätsumfeld. Hierzu werden folgende Schlüsselpraktiken untersucht:

- XP als Softwaretechnik zur schrittweisen Annäherung an die Anforderungen eines Systems
- Änderung der Anforderungen an das Systems
- Funktionalitäten (Features) werden als Tätigkeiten des Benutzers (User Stories) definiert
- Zuerst werden Komponententests (Modultests) geschrieben und anschließend die Features (Test-driven Design)
- Keine separaten Testing-Phasen
- Keine formalen Reviews oder Inspektionen
- Regelmäßige Integration von Änderungen
- Gemeinsame Implementierung (Pair Programming) in Zweiergruppen

Aus der Fallstudie geht hervor, dass Extreme Programming einige Vorteile bei der Bearbeitung eines Softwareprojektes einer Bachelorarbeit bietet. Zum Einen können sich Anforderungen an das zu erstellende System durch parallele Literaturrecherche ändern, zum Anderen können Arbeitspakete durch Releases abgebildet werden.

Diese Softwaretechnik wird in der Umsetzung des Softwareprojekts angewendet.

2.3 Verwandte Arbeiten

Guo u. a. führt ein Modell zum Lernen von L-Systemen von Verzweigungsstrukturen mithilfe maschinellen Lernens (Deep Learning) anhand beliebiger Grafiken ein [9]. Hierzu werden atomare Strukturen mit einem neuronalen Netz erkannt, eine hierarchische Topologie (Baumstruktur) aufgebaut, aus der ein L-System inferiert und mit einem Greedy Algorithmus optimiert wird. Ausgabe des Systems ist ein generalisiertes L-System, aus dem ähnliche Strukturen, wie die der Inputgrafik, erstellt werden können.

Das Lernen von Design-Patterns mithilfe von Bayes-Grammatiken ist Gegenstand der Arbeit von Talton u. a.. [25] führt ein System zur Generierung geometrischer Modelle und Websites ein, das eine organisierte Struktur von bezeichneten Teilmodellen entgegennimmt und über einen Prozess der MCMC-Optimierung eine Bayes-optimale Grammatik erstellt, um neue Modelle zu generieren.

Auch [22] etabliert einen MCMC-Ansatz zum Finden prozeduraler Repräsentationen für biologische Bäume. Hier wird zunächst über eine Laplace-Glättung ein Grundgerüst gefunden, das dann in einer Baumtopologie organisiert wird.

Das von Martinovic und Van Gool eingeführte System nutzt ähnlich organisierte Eingabestrukturen in Form von Gebäudefassaden, um durch bayesische Grammatikinduktion eine kontextfreie Grammatik zu induzieren [12]. Das Erstellen kontextfreier Grammatiken mithilfe statistischer Methoden zur Verteilung von zweidimensionalen Clustern wird in der Arbeit von Stava u. a. gegeben.

urbanen Strukturen [17] Polynomiale Algorithmen zum Inferieren von L-Systemen [13] Framework zum Inferieren von L-Systemen aus Vektordaten [23] Generalisieren einen Regelsets aus einem Inputset mit Markov Chain Monte Carlo [25, 24] Lernen von Regeln zum Layout für Gebäude [12] Inverse prozedurale Modellierung von Fassaden [27]

3 Konzepte

Mit der Erstellung eines Programms zur Synthetisierung von Ähnlichkeitsabbildungen einer vom Benutzer erstellten Verzweigungsstruktur soll die Praktikabilität aktueller Forschungsansätze untersucht werden.

Folgende Kernkonzepte werden erläutert und umgesetzt:

- Visualisierung und prozessorientiertes Erstellen von Basisstrukturen,
- Organisation in einer prozessoptimierten, baumähnlichen Topologie,
- L-System-Repräsentationen,
- Algorithmen zur Inferierung, Komprimierung, Generalisierung und
- Verarbeitung von Transformationsparametern

3.1 Probleme & Lösungsansätze

Visualisierung

Um eine geführte Erstellung der Basisstruktur zu ermöglichen, muss diese während der Erstellung sichtbar gemacht werden. Hierzu werden die Templates in Form von Zeichenketten angelegt und mittels Turtle-Grafik visualisiert. Eine Turtle-Grafik beschreibt die Interpretation einer Zeichenkette als Bild durch Ausführen eines Logo-Turtle-Algorithmus. Weiter wird auch zur Evaluation von Ergebnissen eine Visualisierung benötigt. Da die Verzweigungsstrukturen in L-System-Repräsentation vorliegen, wird hierzu eine Interpretationsfunktion benötigt, die diese Ersetzungssysteme in Bildform darstellen. Ein L-System wird durch Ausführung in eine erweiterte Zeichenkette überführt und als Turtle-Grafik beschrieben [19].

Basisstruktur

Der Benutzer nutzt grafische Bendienelemente, um eingelesene Templates auszuwählen, Transformationsparameter anzupassen und um anschließend die Instanzen der Basisstruktur hinzuzufügen. Im Folgenden wird diese Basisstruktur u.a. Grundstruktur und Eingabestruktur genannt.

Baumstruktur

Um Grundstrukturen mittels verschiedener Algorithmen untersuchen zu können, werden die einzelnen Template-Instanzen in einer baumähnlichen Struktur organisiert. Transformationsparameter einer Instanz beschreiben die räumlichen Veränderung gegenüber des zugrundeliegenden Templates und haben daher keine Aussagekraft in Bezug auf die Strukturtopologie der Basisstruktur. Diese Arbeit fokussiert sich auf topologische Eigenschaften von Verzweigungsstrukturen (z.B. Rekursionen). Darum bilden die einzelnen Template-Instanzen die Knoten der Baumtopologie, während die Kanten die räumlichen Transformationen darstellen. So wird eine datenstrukturelle Trennung zwischen Topologie und räumlichen Transformationen geschaffen. Diese baumähnliche Struktur ist inspiriert durch [9], Kapitel 4.2 *Grammar inference*

Inferieren

Das Smallest Grammar Problem, also das Finden der kleinsten, kontextfreien Grammatik, welche eine bestimmte Zeichenkette generiert, ist ein offenes Problem der Informatik mit einem Annäherungsverhältnis von weniger als $\frac{8569}{8568}$ (NP-hard). Primär wird in der Forschung nach Algorithmen gesucht, die ein akzeptables Ergebnis liefern. In dieser Arbeit wird ein Algorithmus präsentiert, der die Knoten der Baumstruktur in einzelne Symbole umwandelt, mit Produktionregeln verknüpft und diese dem resultierenden L-System hinzufügt. Dieses L-System repräsentiert lediglich die Eingabestruktur.

Komprimieren

Um ein kompaktes, gewichtetes L-System zu erzeugen, werden sich wiederholende Unterräume gesucht und ersetzt. Eine Gewichtung wird angewendet, um das zu erzeugende L-System mit kleiner Regelmenge oder mit großer Regelmenge auszustatten. Eine Kostenfunktion stellt hierbei die Anzahl Symbole aller RHS der Produktionsregeln mit der Menge an Anwendungen der LHS gegenüber.

Generalisieren

Da das kompakte L-System eine Repräsentation der vom Benutzer erzeugten Verzweigungsstruktur darstellt, werden ähnliche Regeln miteinander verbunden (Merge) und mit einer Wahrscheinlichkeit versehen, um nicht-deterministische Regeln hinzuzufügen. Eine weitere Kostenfunktion bewertet den Merge zweier Produktionsregeln und wendet eine Gewichtung über die Länge der Grammatik zur Änderungsdistanz der alten (ohne Merge) zur neuen Grammatik um.

Transformationen

Die vom Benutzer vergeben Werte der Transformationsparameter werden während der Erstellung der Eingabestruktur in einer Häufigkeitsverteilung organisiert und bei Ausführung des generalisierten L-Systems angewendet. So werden Transformationen nach ihrer statistischen Häufigkeit angewendet.

3.2 Workflows & Algorithmen

Verzweigungsstruktur erstellen

Um als Benutzer des Systems eine Verzweigungsstruktur zu erstellen, wird folgender Arbeitsablauf umgesetzt:

Algorithmus 31: Erstellen einer Verzweigungsstruktur

```
1   Erster Anker ist vorselektiert
2   Wiederhole, bis Struktur fertiggestellt ist:
3   Selektiere ein Template aus der Liste
4   Setze Parameter
5   Bestätige Auswahl und Parameter
6   Zeichne ausgewähltes Template mit Parametern
7   Wähle nächsten Anker aus
```

L-System inferieren

Aus der Verzweigungsstruktur kann nun ein L-System erzeugt werden. Hierzu wird ein neuer Algorithmus präsentiert:

Zunächst werden L-System-Komponenten initialisiert (Zeile 2-6) für $\mathcal{L} = \langle M, \omega, R \rangle$:

- Das Alphabet wird mit den Symbolen F und S initialisiert, da F als Repräsentation einer grundlegenden Zeichenoperation und S als Axiom in jeder Anwendung des Algorithmus vorkommt
- Die erste Produktionsregel α umfasst die Abbildung des Axioms auf einen neuen Symbol, das nicht im Alphabet vorkommt. Das Alphabet wird stets um ein Symbol lexikografischer Ordnung ergänzt
 - Bsp.: $\{A, B, C\}$ wird ein neues, unbekanntes Symbol hinzugefügt $\rightarrow \{A, B, C, D\}$
- Die Variable β hält zu untersuchende Knoten der Baumtopologie, welche nach Breitensuche iteriert werden. Die erste Iteration startet bei Wurzelknoten S.
- Als letzten Schritt der Initialisierung wird dem Alphabet ein neues Symbol hinzugefügt, das durch die Variable γ gehalten wird.

Die Schleife des Algorithmus beschäftigt sich mit der Iterierung des Baumes und dem Erstellen neuer Symbole und Produktionsregeln für das resultierende L-Systems (Zeile 8-17):

- Die in den Knoten des Baumes gehaltenen Template-Instanzen entsprechen einer Zeichenkette, die durch eine Turtle-Grafik interpretiert, einem vom Benutzer transformierten Tempalte entspricht. Diese Zeichenkette wird in δ gespeichert
- In Zeile 9-12 wird die genannte Zeichenkette auf Verzweigungsvariablen ($A - Z; F$ ausgeschlossen) untersucht. Diese werden durch ein neues Symbol, das dem Alphabet hinzugefügt wird, ersetzt. Anschließend wird eine Produktionsregel, die auf die veränderte Zeichenkette abbildet, der Produktionsregelmenge hinzugefügt.
- Zeile 13 prüft, ob es Symbol im Alphabet gibt, das nicht als Ziel einer Produktionsregel in der Produktionsregelmenge definiert ist. Ist dies der Fall, wird dieses als Ziel der nächsten Produktion gesetzt. Andernfalls schließt der Algorithmus ab
- Schleifen-Attribut ist hier das Setzen des nächsten Knotens am Ende der Schleife (Zeile 17)

Algorithmus 32: Inferieren eines L-Systems aus einer Baumstruktur

```

1      Initialisierung :
2       $M = \{F, S\}$ 
3       $\omega = S$ 
4       $R \leftarrow \{\alpha: S \rightarrow A\}$ 
5       $\beta = \text{nächster Knoten}$ 
6       $M \leftarrow \gamma \in \{A, B, \dots, Z\}, \text{ mit } \gamma \notin M$ 
7      Schleife :
8       $\delta = \text{Wort von } \beta$ 
9       $\forall \{A, B, \dots, Z\} \setminus F \in \delta :$ 
10     Ersetze mit  $\zeta \in \{A, B, \dots, Z\}, \text{ mit } \zeta \notin M$ 
11      $M \leftarrow \zeta$ 
12      $R \leftarrow \{\gamma \rightarrow \delta\}$ 
13     Wenn es ein Symbol  $\eta$  in  $M \setminus \{F, S\}$  gibt mit  $\{\eta \rightarrow \text{bel.}\} \notin R :$ 
14      $\gamma = \eta$ 
15     Sonst :
16     Breche Schleife ab
17      $\beta = \text{nächster Knoten}$ 

```

L-System komprimieren

Guo u. a. führt einen Algorithmus zum Inferieren einer Grammatik aus einer Baumstruktur ein [9]. Zum Einen wird ein L-System aufgebaut, zum Andern die Baumtopologie durch Finden maximaler Subbäume reduziert. Die Reduktion wird im folgenden Algorithmus adaptiert:

Initialisierung (Zeile 2-5):

- Das L-System, welches der Eingabe des Algorithmus entspricht, wird ausgeführt und die resultierende Zeichenkette wird in \mathcal{L}^+ gespeichert
- Ein Gewichtungssparameter w_l wird eingeführt, der eine Kostenfunktion (Zeile 11) nach Anzahl an Symbolen der RHS von Produktionsregeln und Anzahl deren Anwendung gewichten soll.
- Anschließend wird ein maximaler Subbaum durch geschachtelte Iteration des Baumes T gesucht und als T' gesetzt
- Es werden ausschließlich maximale Unterbäume behandelt, die mindestens zweimal im Baum vorkommen

Die Schleife (Zeile 7-13) stellt die Reduzierung dar:

- Zunächst werden alle Vorkommen des maximalen Unterbaumes durch ein neues Symbol ersetzt
- Aus diesem Subbaum wird ein L-System inferiert, das wiederum in eine erweiterte Zeichenkette ausgeführt wird
- Die Zeichenkette wird als LHS einer neuen Produktionsregel gesetzt, die auf das neue Symbol abzielt
- Das alte L-System kann nun mit dem veränderten L-System mittels Kostenfunktion verglichen werden (Zeile 9): Liegen die Kosten des veränderten L-Systems unter den Kosten des Alten, wird die Reduktion beendet. Andernfalls gilt der Subbaum nun als Eingabebaum und das veränderte Ersetzungssystem als Eingabe-L-System

Algorithmus 33: Erstellen eines kompakten L-Systems mit Gewichtung w_l

```

1  Initialisierung :
2   $\mathcal{L}^+ \leftarrow L_s$ 
3   $\mathcal{L} = \emptyset$ 
4  Setze Gewichtungssparameter  $w_l \in [0, 1]$ 
5  Finde maximalen Unterbaum  $T'$  aus  $T$  mit Wiederholungen  $n > 1$ 
6  Schleife (Reduzierung):
7  Ersetze alle Vorkommen von  $T'$  mit dem selben Symbol  $\gamma \in \{A, B, \dots, Z\}$ 
8   $R \leftarrow \{\gamma \rightarrow L_s\}$  mit  $L_s$  aus  $T'$ ,  $R$  aus  $\mathcal{L}$ 
9  Wenn  $C_i(\mathcal{L}) \geq C_i(\mathcal{L}^+)$ 
10 Breche Schleife ab
11  $T \leftarrow T'$ 
12  $\mathcal{L}^+ \leftarrow \mathcal{L}$ 
13 Finde maximalen Unterbaum  $T'$  aus  $T$  mit Wiederholungen  $n > 1$ 

```

Algorithmus 34: Kostenfunktion C_i mit Gewichtung w_l

```

1   $C_i(\mathcal{L}) = \sum_{A(P) \rightarrow M^* \in \mathcal{L}} w_l * |M^*| + (1 - w_l) * N(A(P) \rightarrow M^*)$ 

```

mit $N(\cdot)$ als Zählfunktion für die Anzahl Wiederholungen einer *LHS* einer Regel in einem ausgeführten L-System.

L-System generalisieren

Da das kompakte L-System eine Repräsentation der vom Benutzer erzeugten Verzweigungsstruktur darstellt, werden nun ähnliche Regeln miteinander verbunden und mit einer Wahrscheinlichkeit versehen, um nicht-deterministische Regeln hinzuzufügen. Sowohl Längenfunktionen, Kostenfunktionen und Distanzalgorithmen, als auch der Grundalgorithmus sind aus [9] entnommen und bauen sich wie folgt auf:

- Die Längenfunktion L , die auf eine Grammatik angewendet wird, summiert die Anzahl Symbole des Alphabets mit der Anzahl an RHS der Produktionsregeln und misst somit die Gesamtheit aller Symbole, die das L-System abbilden soll (Länge der Grammatik)
- Der Abstand zweier Zeichenketten kann mit der *String Edit Distance* ermittelt werden. Diese wird über die Funktion D_s abgebildet. Hierbei wird die Anzahl an Operationen summiert, die für die Überführung einer Zeichenkette in eine Andere nötig sind (Zeichenkettenaustausch, -einschub und -löschung)

- Mit D_s kann nun auch der Abstand zweier Grammatiken zueinander bestimmt werden. Diese Funktion wird mit D_g abgebildet
- Die Kostenfunktion C_g nutzt die Länge und Distanz von Grammatiken, um Kosten einer Überführung einer Grammatik in eine andere messen zu können. Sie berechnet also sie die Kosten, um L^* in L^+ zu überführen. Der Parameter w_0 gewichtet hierbei die Differenz der Länge der Grammatiken und die Anzahl Operationen, die zur Überführung nötig sind. Die Überführungskosten werden in der Variable C_g^{old} zur Verfügung gestellt.

Algorithmus 35: Längenfunktion L für Grammatiken

$$1 \quad L(\mathcal{L}) = |M| + \sum_{A(P) \rightarrow M^* \in \mathcal{L}} |M^*|$$

Algorithmus 36: Grammar Edit Distance

$$1 \quad D_g(\mathcal{L}^+, \mathcal{L}^*) = \sum_{(A(P) \rightarrow M_A^*, B(P) \rightarrow M_B^*) \in M(\mathcal{L}^+ \rightarrow \mathcal{L}^*)} D_s(M_A^*, M_B^*)$$

Algorithmus 37: Kostenfunktion C_g mit Gewichtung w_0

$$1 \quad C_g(\mathcal{L}^*, \mathcal{L}^+) = w_0 * (L(\mathcal{L}^*) - L(\mathcal{L}^+)) + (1 - w_0) + D_g(\mathcal{L}^+, \mathcal{L}^*)$$

Algorithmus zum Generalisieren eines L-Systems:

Initialisierung (Zeile 2-4):

- Das zu untersuchende Tuple bestehend aus zwei Produktionsregeln (Regelpaar) und wird in der Variable p^* gehalten
- L-Systeme, die sich infolge eines Merges geändert haben, werden in L^+ und Eingabe-L-systeme in L^* gespeichert

Generalisierung (Zeile 6-13):

- \mathcal{P} ist die Menge aller möglichen Regelpaare aus L^*
- Das Regelpaar, das beim Merge die geringsten Kosten für die Überführung in die neue Grammatik aufweist, wird in p^* gehalten
- Sind diese Kosten positiv, wird die Generalisierung abgebrochen
- Andernfalls werden die Variablen c^* als Delta-Kosten, C_g^{old} und L^* entsprechen gesetzt
- Sollte die Differenz der Überführungskosten positiv sein, wird die Generalisierung abgebrochen

Algorithmus 38: Generalisieren eines L-Systems mit Gewichtung w_0

```

1  Initialisierung :
2  Regelpaar  $p^* = \emptyset$ 
3   $\mathcal{L}^* = \mathcal{L}^+$ 
4   $C_g^{old} = C_g(\mathcal{L}^* + \{p^*\}, \mathcal{L}^*)$ 
5  Schleife :
6  Finde Regelpaar  $p^*$  mit minimalen Kosten  $C_g(\mathcal{L}^* + \{p_i\}, \mathcal{L}^*), \forall p_i \in \mathcal{P}$ 
7  Wenn  $C_g(\mathcal{L}^* + \{p^*\}, \mathcal{L}^*) \geq 0$ 
8  Breche Schleife ab
9   $c^* = C_g(\mathcal{L}^* + \{p^*\}, \mathcal{L}^*) - C_g^{old}$ 
10  $C_g^{old} = C_g(\mathcal{L}^* + \{p^*\}, \mathcal{L}^*)$ 
11  $\mathcal{L}^* = \mathcal{L}^* + \{p^*\}$ 
12 Wenn  $c^* > 0$ 
13 Breche Schleife ab

```

3.3 Softwarearchitektur

Die Gliederung der Inhalte für die Softwarearchitektur erfolgt nach der arc42-Vorlage [4]

Qualitätsziele

Um die wesentlichen Features des Systems in einem Programm umzusetzen, werden folgende Qualitätsziele definiert, priorisiert (absteigend) und umgesetzt:

- **Funktionalität** durch Umsetzen aller Teilsysteme in Vollständigkeit, Korrektheit, Angemessenheit
- **Interoperabilität** durch Nutzen einer allgemeinen Repräsentation von L-Systemen, damit diese auch in anderen Programmen oder Algorithmen verwendet werden kann
- **Erweiterbarkeit** durch offene Entwurfsmuster (Design Patterns)
- **Modularität** durch Implementierung für effiziente Wartung und Erweiterung
- **Effizienz** durch effiziente Programmierung
- **Attraktivität** durch intuitive Benutzung (Benutzerfreundlichkeit)
- **Plattformunabhängigkeit** durch Verwenden des Java-Frameworks

Kontextabgrenzung

Die Systemgrenzen werden zum Einen durch die Interaktion mit dem Benutzer, zum Anderen durch die Interaktion mit dem Dateisystem des Host-Systems und dem Zugreifen und Lesen der Template-Dateien definiert. Hierbei wird die Erstellung der Basisstruktur als nicht-technische Interaktion und das Einlesen der Dateien als technische Interaktion gesehen.

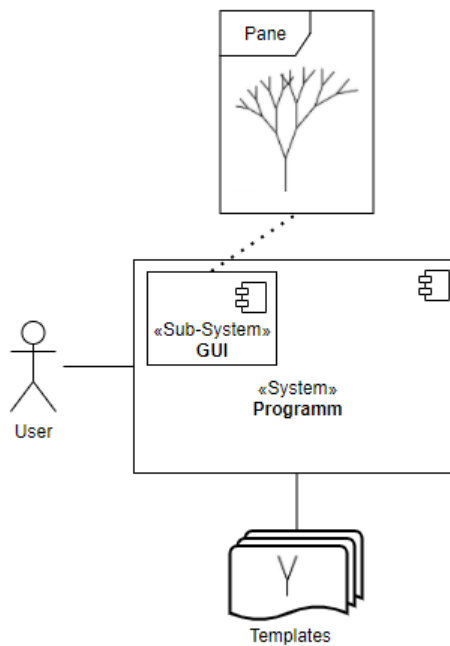


Abbildung 3.1: System und Systemumgebung

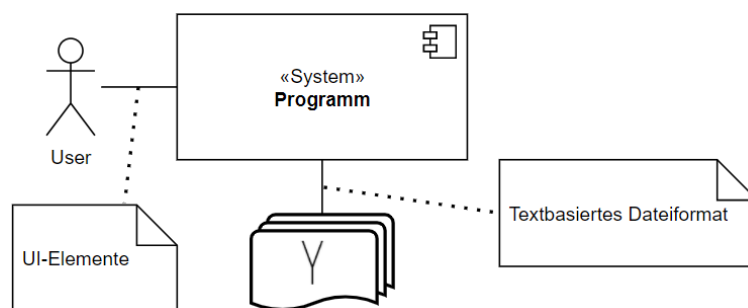


Abbildung 3.2: Interaktion zwischen System und Systemumgebung

Lösungsstrategie

Zur Erreichung der o.g. Qualitätsziele werden folgende Architekturansätze umgesetzt:

Qualitätsziel	Architekturansatz
Funktionalität	Grafische Benutzerschnittstelle Generieren der Baumstruktur Verarbeiten von L-Systemen
Interoperabilität	Durch das Nutzen allgemeingültiger mathematischer Beschreibungen sollen erstellte L-Systeme in Fremdsystemen, wie Online Visualisierer, genutzt werden können
Erweiterbarkeit	Das Nutzen des Pipeline Design Patterns soll das Erweitern des Systems durch Hinzufügen weiterer Teilschritte (Pipes) erleichtern. Trennung der grafischen Oberfläche und der Logik durch Aufbauen des Szenen-graphen über ein XML-Dateiformat
Modularität	Sowohl eine sinnvolle Aufteilung von Funktionalitäten auf Dateien und Software-Pakete, als auch effiziente Datenkapselung und geschlossene Informationskontexte sorgen für Modularität des Programms

Bausteinsicht

Der Benutzer interagiert über das User Interface mit dem Subsystem GUI, das sich mit der Visualisierung, dem Aufbau der Eingabestruktur und dem Anlegen einer internen Baumtopologie beschäftigt. Die Pipeline zum Erzeugen der Ausgabestrukturen beginnt mit dem Inferieren eines L-Systems aus der benutzerdefinierten Struktur und gibt das erzeugte Ersetzungssystem an die folgenden Komponenten weiter. Hierbei gilt der Pipeline-Kontext, welcher innerhalb der Pipeline an den jeweils nächsten Schritt weitergegeben und dort aktualisiert wird, als Eingabe der Pipeline. Hat die Estimator-Komponente eine Verteilung über vom Benutzer angelegte Transformationsparameter angelegt, gilt der Kontext als Ausgabe der Pipeline.

Ebene 1

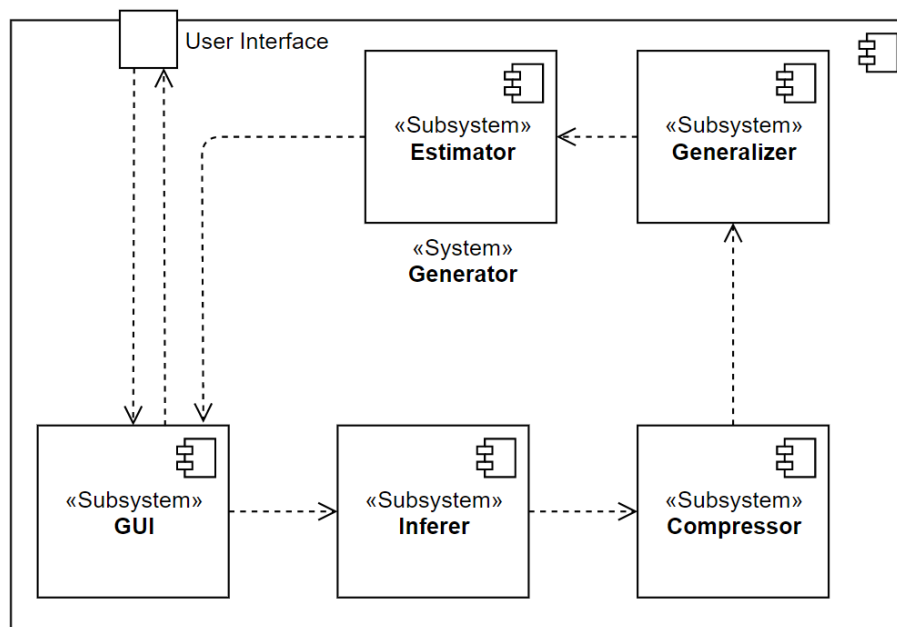


Abbildung 3.3: Subsysteme mit fachlichen Abhängigkeiten

Betrachtet man die GUI-Komponente genauer, setzt sich diese aus vier Subsystemen zusammen. Das Application-Modul beschreibt den Einstiegspunkt für ein JavaFX-Programm. Die grafische Benutzerschnittstelle wird über ein XML-basiertes FXML-Dateischema (Scene graph) mit zugehörigem FXML-Controller (Model) aufgebaut. Der Controller stellt die Logik für eine JavaFX-Oberfläche zur Verfügung. Neben der Erstellung der Basissstruktur, wird die Baumstruktur in einer separaten Komponente umgesetzt.

Ebene 2

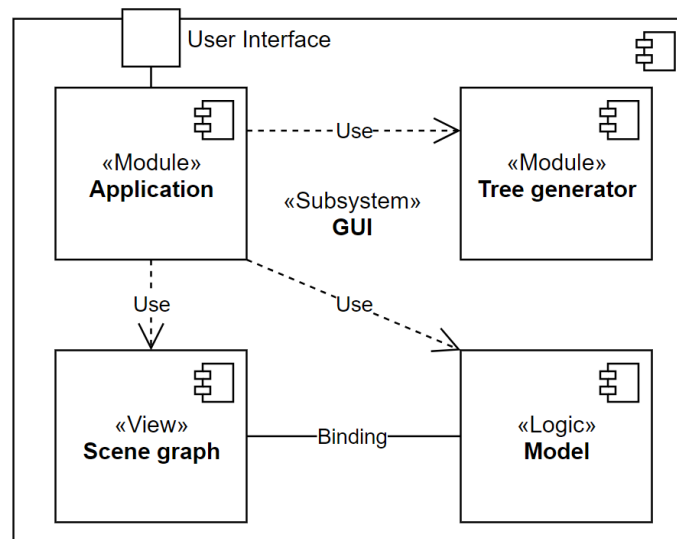


Abbildung 3.4: Subsystem GUI

Laufzeitsicht

Aus der nachfolgenden Abbildung geht hervor, wie sich der Informationsfluss zwischen den einzelnen Subsystemen verhält. Wird der Systemprozess einmal durchlaufen, besteht die Möglichkeit die Ausführung des generalisierten L-Systems mit erneuter Vergabe der Transformationsparameter aus der Häufigkeitsverteilung der Estimator-Komponente zu starten. Der Ablauf des Systems setzt sich wie folgt zusammen:

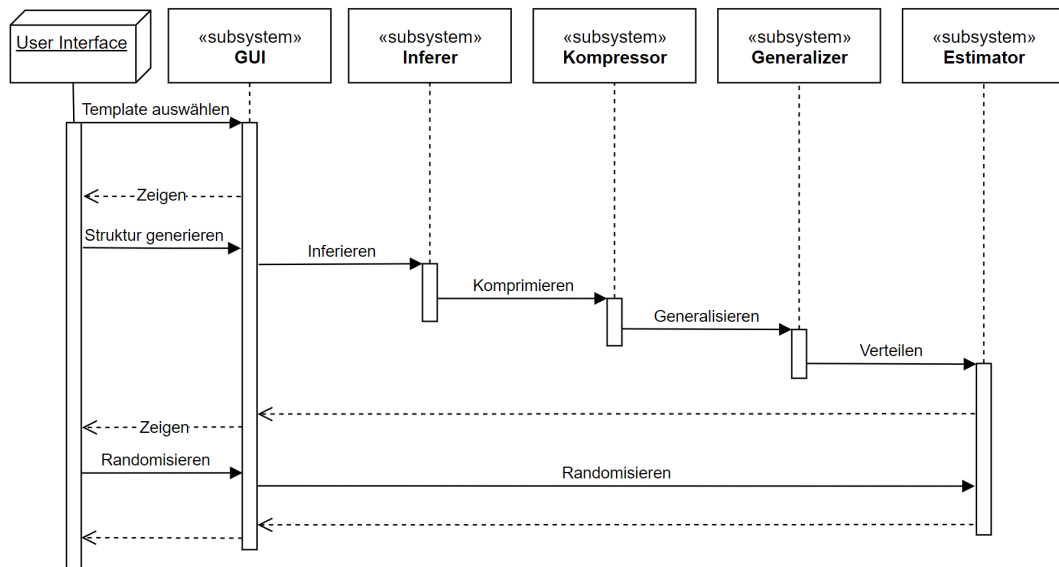


Abbildung 3.5: Laufzeitsicht

Verteilungssicht

Die Ausführung des Programms wird durch ein einfaches Startskript zur Verfügung gestellt. Die folgende Abbildung der Verteilungssicht stellt lediglich die Ausführung auf einer Windows-Maschine dar.

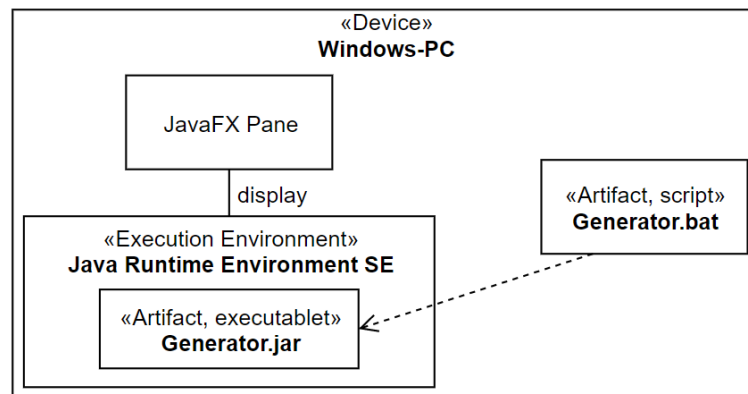


Abbildung 3.6: Infrastruktur Windows-PC

Datenstrukturen

TODO

4 Implementierung

Zur Umsetzung der vorgestellten Konzepte wird im Folgenden auf Softwarepakete, Technologien, Datenspeicherung, Benutzerinteraktion und Arbeitsablauf der erstellten Software eingegangen. Darüber hinaus wird ein Überblick über Quellcode und einige Implementierungsentscheidungen gegeben. Auf Implementierungsdetails zur Nutzung der vorgestellten Technologien wird nicht im Detail eingegangen. Quellcode wird im Wesentlichen gezeigt und anhand von Zeilenangaben (z.B. **5**) erläutert. Softwareumgebungsspezifische Implementierungsdetails werden unter Angabe von drei Punkten (...) weggelassen.

4.1 Projektstruktur

Das Softwareprojekt ist nach der Gradle-Source-Code-Konvention organisiert. Das gesamte Programm befindet sich im Ordner `Generator`, der obligatorische Gradle-Dateien und den Source-Ordner enthält. Sowohl die Quelldateien, also auch die Testdateien sind der Paketstruktur **de.haw** untergeordnet. Die folgende Tabelle gibt einen Überblick über grundlegende Pakete und deren Funktion.

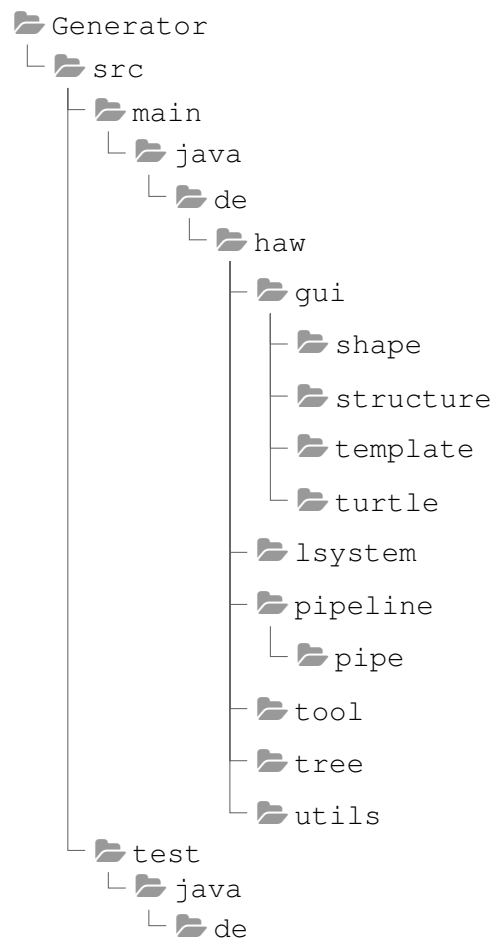


Abbildung 4.1: Softwareprojekt Dateistruktur

Paket	Funktion
<i>gui</i>	Visualisierung des Programms
<i>lsystem</i>	L-System-Repräsentation
<i>pipeline</i>	Umsetzung des Pipeline-Design-Patterns
<i>tool</i>	Methodiken und Algorithmen
<i>tree</i>	Komponenten der Baumstruktur
<i>utils</i>	Hilfskomponenten

Abbildung 4.2: Softwarepakete mit zugehörigen Funktionen

4.2 Technologien

Zur Umsetzung des Softwareprojektes wird eine Java-Anwendung für die Java-Laufzeitumgebung entwickelt. Sie liegt in der Distribution **Amazon Corretto** in der Version 11.0.3_7 vor. Grafische Oberflächen werden mit der JavaFX-Spezifikation von Oracle in der Version 11.0.2 umgesetzt. Zur Automatisierung von Abhängigkeits- und Buildmanagement wird Gradle (Version 6.7) verwendet. Eine Testumgebung, eine Vektorbibliothek, eine Tupelrepräsentation und eine Erweiterung zur mathematischen Standardbibliothek werden über Abhängigkeiten vom Gradle-Framework im Build-Prozess geladen und zur Verfügung gestellt.

Um den test-driven Implementierungsansatz umzusetzen wird JUnit 5 Jupiter als Testumgebung genutzt. Sie setzt sich aus einem Programmierschema und einem Erweiterungsmodell zusammen. Das Jupiter-Projekt liefert zudem die Laufzeitumgebung für Softwaretests. Googles Guava liefert eine Bibliothek mit mathematischen Funktionen. Sie wird benötigt, um eine praktikable Lösung zu Mengenmanipulation nutzen zu können (Bsp. Erstellen von Kombinationspaaren einer Menge). Ausschließlich JavaFX wird außerhalb des Projektes installiert und als Modul im Start-Skript des Programmes hinzugefügt.

Weitere Systeme zur Erstellung des Softwareprojektes sind:

- Versionierung via Git,
- Dot zur Visualisierung von Graphen und
- PlantUML zur Generierung von UML-Diagrammen

4.3 Konzeptumsetzung

Das Skript `Generator.bat`, das zum Starten der Anwendung innerhalb eines Windows-Betriebssystems genutzt werden kann, fügt dem Programm alle externen Module hinzu, die während der Laufzeit genutzt werden, und führt die angegebene jar-Datei aus:

```
1      java --module-path .\javafx-sdk-11.0.2\lib --add-modules  
      ↪     javafx.controls,javafx.fxml,javafx.graphics -jar  
      ↪     Generator-1.0.jar  
2      pause
```

Abbildung 4.3: Startskript `Generator.bat`

Beim Start der Anwendung findet der Benutzer die grafische Oberfläche vor.

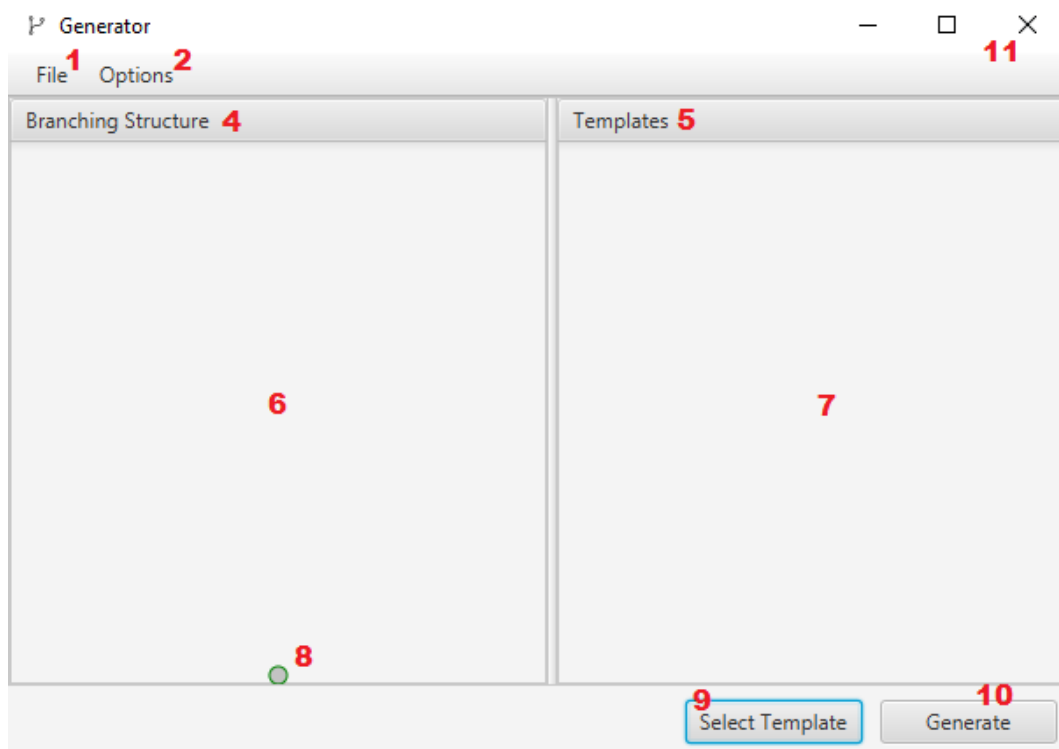
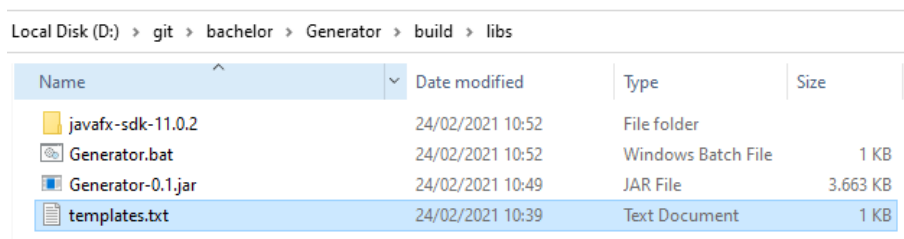


Abbildung 4.4: Programm nach Ausführung des Start-Skripts

Der Menüeintrag **1** bietet Funktionen zum Öffnen des Dateipfades und zum Laden der angelegten Templates-Datei. **2** dient zur Konfiguration der Umgebungsparameter für die Anzahl Iterationen der Ausführung und die Anzahl an Ähnlichkeitsstrukturen, die aus dem resultierenden L-System abgeleitet werden können. Die Gewichtungsparemeter der vorgestellten Algorithmen können ebenfalls hier eingestellt werden. Die Strukturen **4** und **5** gliedern das Programm in die Ansicht der Verzweigungsstruktur (**6**), die vom Benutzer angelegt wird, die Übersicht zur Auswahl eines Templates (**7**) und eine Sicht zur Setzung von Transformationsparametern (**7**). Der Kreis **8** stellt einen Anker dar, an welchen eine Template-Instanz angehängt werden kann. Sowohl über einen Doppelklick, also auch über einen Button (**9**) kann ein Template aus der Tempalte-Liste (**7**) ausgewählt werden. Ist die Verzweigungsstruktur vom Benutzer fertiggestellt, kann die Synthese zur Erstellung der Ähnlichkeitsabbildungen mit **10** erfolgen. **11** schließt die Anwendung.

Templates

Eine Datei *template.txt* wird im selben Verzeichnis wie die ausführbare jar-Datei hinerlegt. Die enthält je eine Template-Zeichenkette pro Zeile, die vom Programm eingelesen und als Template zur Verfügung gestellt wird.



Name	Date modified	Type	Size
javafx-sdk-11.0.2	24/02/2021 10:52	File folder	
Generator.bat	24/02/2021 10:52	Windows Batch File	1 KB
Generator-0.1.jar	24/02/2021 10:49	JAR File	3.663 KB
templates.txt	24/02/2021 10:39	Text Document	1 KB

Abbildung 4.5: Tempaltes-Datei zum Einlesen der Template-Strukturen

Visualisierung

Die grafische Oberfläche liegt als MVC-Pattern in Form des Application State (Model), der XML-basierten FXML-Datei (View) und dem JavaFX-Controller vor. Der Arbeitsablauf zur Erstellung der Verzweigungsstruktur kann Schritt für Schritt umgesetzt werden, nachdem die Templates geladen wurden. Dies wird an folgendem Beispiel deutlich gemacht. Die Parameter **Number of iterations**, **Number of generations**, **Rule application ratio** und **Merge application ratio** werden auf die Werte **5**, **5**, **0.5** und **0.5** gesetzt (Standarteinstellung).

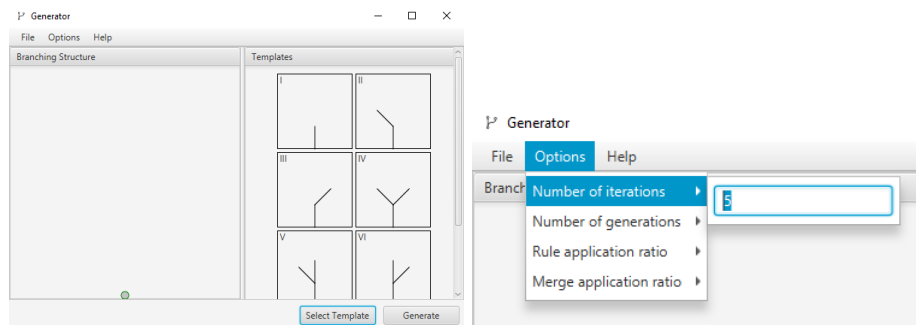


Abbildung 4.6: Erster Anker ist vorselektiert & gesetzte Parameter

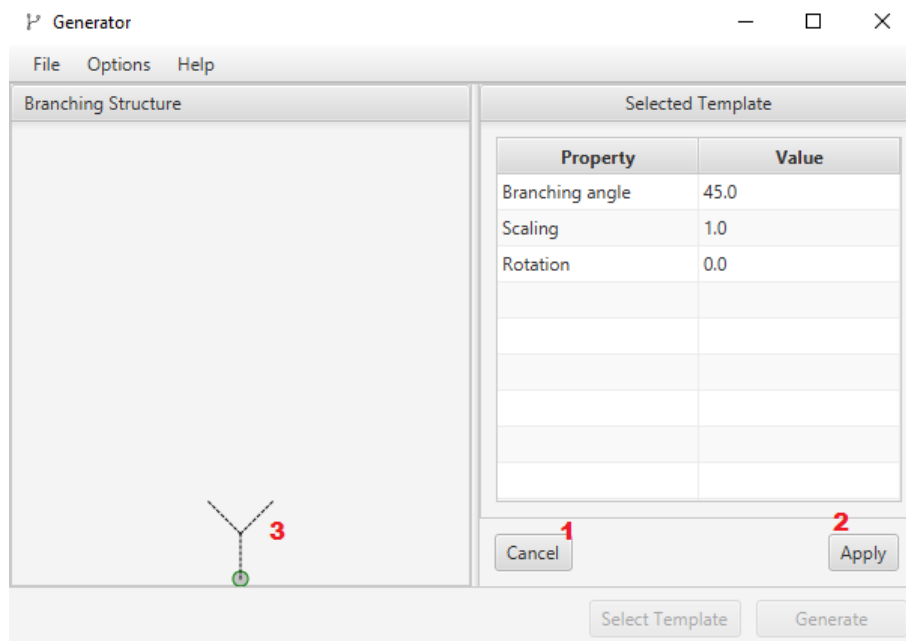


Abbildung 4.7: Auswahl des ersten Templates

Nachdem ein Template ausgewählt wurde, können die Transformationsparameter gesetzt (Doppelklick) werden und die Auswahl rückgängig gemacht (**1**) oder bestätigt (**2**) werden. **3** zeigt einen Entwurf der Tempalte-Instanz, die mit den aktuellen Transformationsparametern angepasst wurde. Mit der Bestätigung wird die Template-Instanz endgültig der Basisstruktur hinzugefügt und der Benutzer gelangt wieder in die Übersicht der zur Verfügung stehenden Templates.

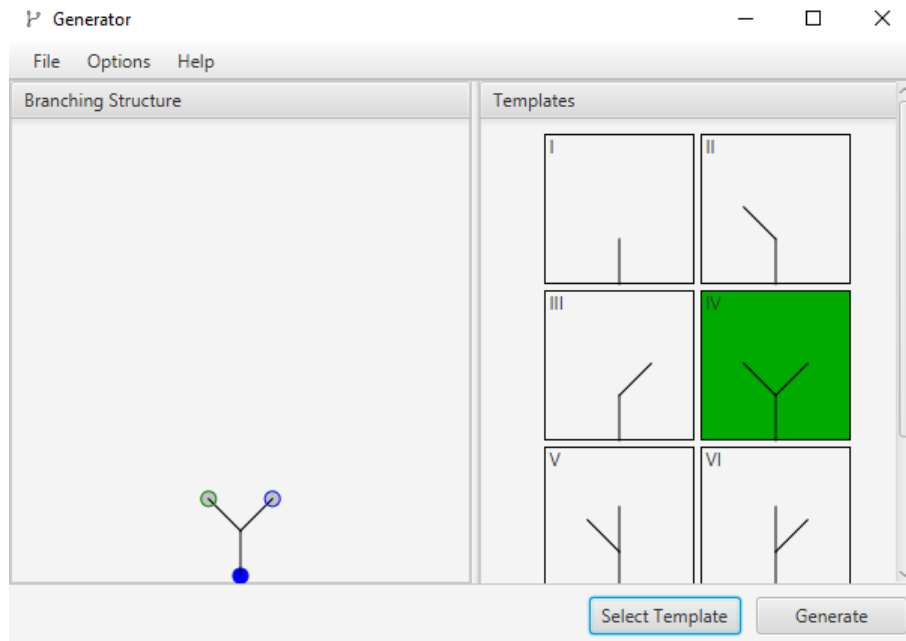


Abbildung 4.8: Der Verzweigungsstruktur hinzugefügte Template-Instanz

Dieser Vorgang wiederholt sich, bis der Benutzer die Struktur fertiggestellt hat.

Baumtopologie

Um eine interne Baumtopologie aufzubauen wird die Klasse `TreeNode` verwendet, um eine iterierbare Baumstruktur aufzubauen:

Listing 4.1: Klasse `TreeNode` zur Erstellung einer Baumstruktur

```
/**
 * Iterable tree node structure containing a payload and children
 * as tree nodes.
 * Every node represents a whole tree
 * @param <T> Tree node content
 */
public class TreeNode<T> implements Iterable<TreeNode<T>> {
    // Payload
    private T data;
    // Children
    protected List<TreeNode<T>> children;
    ...
    @Override
    public Iterator<TreeNode<T>> iterator() {
        return new TreeNodeIterator<>(this);
    }
    ...
}
```

`TreeNode` implementiert die `Iterable`-Schnittstelle, was ein Überladen der `iterator`-Funktion möglich macht. Sie stellt einen Iterator des Baumes zur Verfügung. Dieser Iterator (`TreeNodeIterator`) implementiert die `Iterator`-Schnittstelle und definiert so die Funktionen `hasNext` und `next`.

Listing 4.2: Klasse `TreeNodeIterator` als Iterator für die Baumstruktur

```
/**
 * Tree node iterator for iterating nodes of a tree
 * @param <T> Payload of the tree nodes
 */
public class TreeNodeIterator<T> implements Iterator<TreeNode<T>>{
    ...
    @Override
    public boolean hasNext() {
        return !queue.isEmpty();
    }

    @Override
    public TreeNode<T> next() {
        if (queue.isEmpty()) return null;
        var node = queue.pop();
        queue.addAll(node.getChildren());
        return node;
    }
}
```

Die Funktion `next` zeigt, wie eine Iteration des Baumes als Breitensuche implementiert ist. Diese wird beim Inferieren eines L-Systems aus der Baumstruktur benötigt. Das folgende Beispiel zeigt eine erstellte Baumstruktur (links) und deren erstellte Baumtopologie (rechts).

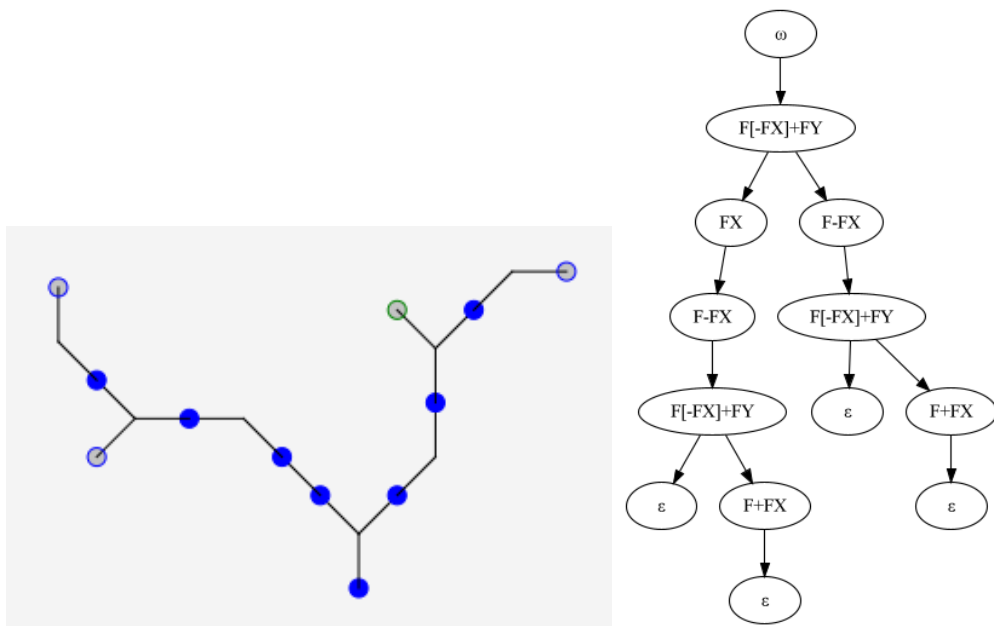


Abbildung 4.12: Verzweigungsstruktur & zugehörige Baumstruktur

Prozesspipeline

Das angewendete Pipeline-Design-Pattern zur Umsetzung der Inferrierung, Komprimierung und Generalisierung von L-Systemem, sowie der Verteilung von Transformationsparametern bei deren Ausführung, setzt sich aus der `Pipeline`-Klasse und dem `PipeInterface` zusammen.

Listing 4.3: Pipeline Klasse zur Organisation von Prozessen

```
/**
 * Pipeline class to set up a pipeline design pattern.
 * Classes implementing the Pipe interface can be executed in a specific
 * order while updating a pipeline context
 * @param <IN> Pipeline input type
 * @param <OUT> Pipeline output type
 */
public class Pipeline<IN, OUT> {
    private final Pipe<IN, OUT> current;

    public Pipeline(Pipe<IN, OUT> pipe) {
        current = pipe;
    }

    public <NewO> Pipeline<IN, NewO> pipe(Pipe<OUT, NewO> next) {
        return new Pipeline<>(input -> next.process(current.process(input)));
    }

    public OUT execute(IN input) {
        return current.process(input);
    }
}
```

Listing 4.4: Pipe Interface als Vorlage zur Erstellung eines Teilprozesses einer Pipeline

```
/**
 * Pipe class as a process to be part of a pipeline
 * @param <IN> Pipe input type
 * @param <OUT> Pipe output type
 */
public interface Pipe<IN, OUT> {
    OUT process(IN input);
}
```

Die eigentliche Umsetzung der Prozesspipeline zur Synthese der Verzweigungsstrukturen wird wie folgt implementiert. Hierbei implementieren die Klassen `InfererPipe`, `CompressorPipe`, `GeneralizerPipe` und `EstimatorPipe` das `Pipe`-Interface und geben die Reihenfolge des Prozesses an. Das `ctx`-Objekt wird als Kontextobjekt in die Pipeline gegeben und bei jedem Prozessschritt angepasst.

Listing 4.5: Erstellen des Pipeline Kontextes

```

...
var ctx = new PipelineContext();
...
// Execute pipeline
var result = new Pipeline<>(new InfererPipe())
    .pipe(new CompressorPipe())
    .pipe(new GeneralizerPipe())
    .pipe(new EstimatorPipe())
    .execute(ctx);
...

```

Inferrierung eines L-Systems aus einer Baumstruktur

Die Inferrierung des L-Systems findet in der Klasse `Inferer` statt, die in der `InfererPipe`-Klasse erstellt wird. Sie nimmt die Baumstruktur entgegen und stellt eine Funktion `infer` zum Inferieren des L-Systems zur Verfügung.

Listing 4.6: Inferer Klasse zur Inferrierung eines L-Systems aus einer Baumstruktur

```

1  /**
2   * Inferer to infer a L-System out of a tree-like data structure
3   */
4  public class Inferer {
5      ...
6      public Inferer(TreeNode<TemplateInstance> tree) {
7          //// Initializing
8          lSystem = new LSystem();
9          // M = {F, S}
10         lSystem.addModule("F", "S");
11         // w = S
12         lSystem.setAxiom("S");
13         // R <- { alpha: S => A }
14         var alpha = new ProductionRule("S", "A");
15         lSystem.addProductionRule(alpha);
16         // beta = next node
17         beta = iterator.next();
18         // M <- gamma in { A, B, ..., Z }, with gamma not in M
19         gamma = lSystem.addModuleNotPresentInAlphabet();
20     }
21     ...
22 }

```

Das erstellte L-System (8) wird während des Inferierens aktualisiert und am Schluss als Ergebnis zurückgegeben. Dann werden die initialen Symbole F und S hinzugefügt und das Axiom auf S gesetzt (10-12). Die initiale Produktionsregel $\alpha : S \rightarrow A$ wird erstellt und der Produktionsregelmengende beigefügt (14-15). Es wird der erste zu untersuchende Knoten des Baumes über seinen Iterator zurückgegeben und als β gesetzt (17). Am Ende der Initialisierung fügt die Methode `addModuleNotPresentInAlphabet` dem Alphabet ein neues Symbol (γ) hinzu, das dort nicht vorhanden ist (19). Dies geschieht nach lexikografischer Reihenfolge.

Nach der Erstellung des `Inferer`-Objektes kann in der zugehörigen Pipe `InfererPipe` das L-System inferriert werden

Listing 4.7: `InfererPipe` Klasse als Teilprozess der Pipeline

```

1  /**
2   * Pipe for executing the infer algorithm.
3   * It takes the pipeline context, set it accordingly to the result
4   * and returns it to the next pipe
5   */
6  public class InfererPipe implements Pipe<PipelineContext, PipelineContext>,
7      Logging {
8      ...
9      @Override
10     public PipelineContext process(PipelineContext input) {
11         ...
12         // Update pipeline context
13         input.lSystem = new Inferer(input.tree).infer();
14         ...
15     }
16 }
```

Die `infer`-Methode extrahiert die Zeichenkette (Wort) des im aktuellen Baumknoten gespeicherten Template-Instanz (9) und speichert sie zur Bearbeitung in `delta`. `Delta` wird nach Verzweigungsvariablen durchsucht (11-14) und diese dann durch ein neues Symbol, das dem Alphabet hinzugefügt wird (18), ersetzt (20). Eine neue Produktionsregel, die γ auf das veränderte Wort abbildet, wird dem L-System beigefügt (24). 31-35 sucht nach Symbolen im Alphabet, die noch kein Ziel einer Produktionsregel sind und hält diese in der Variablen `gamma`. Wird kein solches Symbol gefunden, terminiert die `infer`-Methode und damit der Algorithmus (44). Der aktuelle Knoten wird durch den Iterator der Baumstruktur mit dem nächsten Knoten nach Breitensuche ersetzt (46). Zum Schluss wird das inferrierte L-System zurückgegeben (48).

Listing 4.8: Inferrierungsalgorithmus der Inferer Klasse

```
1  /**
2   * Return a L-System inferred from the given tree structure
3   * @return Inferred L-System
4   */
5  public LSystem infer () {
6      var done = false;
7      while (!done) {
8          // delta = word of beta
9          var delta = (beta == null || beta.isEmpty()) ? "" : beta.getData().
            ➔ getTemplate().getWord();
10         // For all variables in delta
11         var variableMatches = Pattern.compile("[A-EG-Z]")
12             .matcher(delta)
13             .results()
14             .collect(Collectors.toList());
15         for (var v : variableMatches) {
16             var index = delta.indexOf(v.group(0));
17             // Add new module not present in the alphabet
18             String zeta = lSystem.addModuleNotPresentInAlphabet();
19             // Replace variable with new module not present in alphabet
20             delta = delta.substring(0, index) + zeta + delta.substring(
                ➔ index + 1);
21         }
22
23         // R ← { gamma → delta }
24         lSystem.addProductionRule(new ProductionRule(gamma, delta));
25
26         // Find an eta that is in the alphabet but not part of the LHS of
27         ➔ any production rule
28         var modules = lSystem.getAlphabet();
29         for (var eta : modules) {
30             if (eta.equals("F")) continue;
31             if (eta.equals("S")) continue;
32             var lhs = lSystem.getProductionRules().stream()
33                 .map(ProductionRule::getLhs)
34                 .filter(x -> x.equals(eta))
35                 .findFirst()
36                 .orElse(null);
37
38             if (lhs == null) {
39                 gamma = eta;
40                 break;
41             }
42         }
43     }
44 }
```

```
41
42         // There is no symbol in the alphabet not being part of a lhs
           ↳ of a production rule?
43         if (modules.indexOf(eta) == modules.size() - 1) done = true;
44     }
45     beta = iterator.next();
46 }
47 return lSystem;
48 }
```

Die übrigen Prozessschritte sind ähnlich aufgebaut. Auch die Klassen Compressor, Generalizer und Estimator implementieren eine Funktion zum Ausführen der jeweiligen Algorithmen, während die Initialisierung im Konstruktor ausgeführt wird. Dabei wird die Implementierung der Algorithmen eng an der Konzeptionierung gehalten (siehe Kapitel 3.2). Alle weiteren Klassen der Basisalgorithmen werden im Folgenden gezeigt und Spezialfälle erläutert.

Komprimierung des L-Systems

Listing 4.9: Klasse Compressor zur Komprimierung eines L-Systems

```
1  /**
2   * Compressor class for compressing a L-System.
3   * The algorithms searches for identical, maximal subtrees
4   * and replaces them with combined instances
5   */
6  public class Compressor {
7      ...
8      public Compressor(TreeNode<TemplateInstance> tree, LSystem lSystem,
           ↳ float wL, Random randomizer) {
9          //// Initializing
10         this.tree = tree.copy();
11         this.LPlus = lSystem;
12         // L =
13         this.L = new LSystem();
14         // wL [0, 1]
15         weighting = wL;
16         this.randomizer = randomizer;
17     }
18     ...
19 }
```

Listing 4.10: Komprimierungsalgorithmus der Compressor Klasse

```
1  /**
2   * Return a compressed L-System by finding maximum sub-trees
3   * and replacing them
4   * @return Compressed L-System
5   */
6  public LSystem compress() {
7      // T' ← T
8      var subtree = FindMaximumSubTree(tree);
9      while (subtree != null && !subtree.isEmpty()) {
10         // Get extended string representation from (repetitive) sub tree
11         var subTreeDerivation = new Inferer(subtree).infer().derive();
12         // Data to be set in the tree to replace old node structure
13         //    ↳ representing the subtree
14         Template template = new Template(subTreeDerivation);
15         // Estimate parameters for new template instance derivation
16         var estimator = new Estimator(randomizer);
17         var occurrences = getOccurrences(subtree, tree);
18         // Average parameter
19         for (var o : occurrences) {
20             estimator.estimateParameters(o);
21         }
22         // Replace occurrences of the sub tree
23         for (var o : occurrences) {
24             var derivationInstance = new TemplateInstance(template);
25
26             int scalingSum = 0, rotationSum = 0, branchingAngleSum = 0;
27             int counter = 0;
28             for (var node : o) {
29                 if (node.isEmpty()) continue;
30                 counter++;
31                 scalingSum += estimator.averageParameterValueForTemplate("
32                     ↳ Scaling", node.getData().getTemplate().getId());
33                 rotationSum += estimator.averageParameterValueForTemplate("
34                     ↳ Rotation", node.getData().getTemplate().getId());
35                 branchingAngleSum += estimator
36                     ↳ averageParameterValueForTemplate("Branching_angle",
37                     ↳ node.getData().getTemplate().getId());
38             }
39
40             derivationInstance.setParameter("Scaling", (float) (scalingSum
41                 ↳ / counter));
42             derivationInstance.setParameter("Rotation", (float) (
43                 ↳ rotationSum / counter));
44         }
45     }
46 }
```

```
37         derivationInstance.setParameter("Branching_angle", (float) (
38             ↳ branchingAngleSum / counter));
39
40         o.setData(derivationInstance);
41         o.removeChildren();
42     }
43     L = new Inferer(tree).infer().minimize();
44     if (Ci(L) >= Ci(LPlus)) {
45         break;
46     }
47     // T <- T'
48     subtree = FindMaximumSubTree(tree);
49     // L+ <- L
50     LPlus = L;
51 }
52 return LPlus.clean();
53 }
```

Listing 4.11: Algorithmus zum Finden eines maximalen Unterbaums

```
1 /**
2  * Search and for maximum sub-tree that appears more than one time
3  * in the tree and return it
4  * @param tree Tree to be searched
5  * @return Maximum sub-tree
6  */
7 private TreeNode<TemplateInstance> FindMaximumSubTree(TreeNode<
8     ↳ TemplateInstance> tree) {
9     var globalIterator = tree.iterator();
10    globalIterator.next();
11    // Iterate
12    while (globalIterator.hasNext()) {
13        var globalNode = globalIterator.next();
14        var localIterator = tree.iterator();
15        var l = localIterator.next();
16        while (l != globalNode) l = localIterator.next();
17        while (localIterator.hasNext()) {
18            var localNode = localIterator.next();
19            if (!globalNode.isLeaf()) {
20                // Check for equality / check for appearance > 1 in the tree
21                if (Trees.compare(globalNode, localNode)) return globalNode;
22            }
23        }
24    }
25 }
```

```
23     }
24     return null;
25 }
```

Listing 4.12: Algorithmus zum Finden alle Vorkommen eines Unterbaums in einem Baum

```
1  /**
2   * Find all occurrences of a sub-tree in a tree
3   * @param subtree Sub-tree to be searched for
4   * @param tree Tree that contains the sub-tree
5   * @return List of sub-trees
6   */
7  private List<TreeNode<TemplateInstance>> getOccurrences(TreeNode<
    ↳ TemplateInstance> subtree, TreeNode<TemplateInstance> tree) {
8      var iterator = tree.iterator();
9      iterator.next();
10     // Store occurrences
11     var occurrences = new ArrayList<TreeNode<TemplateInstance>>();
12     // Iterate through the tree
13     while (iterator.hasNext()) {
14         var node = iterator.next();
15         if (Trees.compare(node, subtree)) {
16             // Subtree to be replaced found
17             occurrences.add(node);
18         }
19     }
20     return occurrences;
21 }
```

Listing 4.13: Funktion zur Berechnung der Kosten eines L-Systems

```
1  /**
2   * Return the costs of a L-System combining the length of the alphabet
3   * and the number of rule applications of the production rules
4   * @param lSystem L-System to be measured
5   * @return Costs of the L-System
6   */
7  private float Ci(LSystem lSystem) {
8      var costs = 0;
9      for (var rule : lSystem.getProductionRules()) {
10         costs += weighting * rule.getRhs().length() + (1 - weighting) *
            ➔ countRuleApplications(lSystem, rule.getLhs());
11     }
12     return costs;
13 }
```

Listing 4.14: Funktion zur Ermittlung der Anzahl Anwendung einer Produktionsregel

```
1  /**
2   * Counts the production rule applications in a string and returns it
3   * @param lSystem L-System to be searched
4   * @param lhs LHS of a production rule
5   * @return Number of production rule applications
6   */
7  private int countRuleApplications(LSystem lSystem, String lhs) {
8      int occurrences = 0;
9      var pattern = Pattern.compile(lhs);
10     // Check axiom
11     var axiomMatcher = pattern.matcher(lSystem.getAxiom());
12     while (axiomMatcher.find()) occurrences++;
13     // Check production rules
14     var ruleMatcher = pattern.matcher(lSystem.getProductionRules().stream()
15         .map(ProductionRule::getRhs).collect(Collectors.joining()));
16     while (ruleMatcher.find()) occurrences++;
17     return occurrences;
18 }
```

Generalisierung des L-Systems

Listing 4.15: Generalizer Klasse zur Generalisierung eines L-Systems

```
1  /**
2   * Generalizer class to add non-deterministic rules
3   * to a L-System
4   */
5  public class Generalizer {
6      ...
7      public Generalizer(LSystem lSystem, float w0) {
8          ///// Initialization
9          // Generalized grammar  $L^* = L^+$  (compact grammar)
10         LStar = lSystem.copy();
11         // metric weight balancing
12         this.w0 = w0;
13         //  $C^{old}_g = C_g(L + \{p\}, L)$ 
14         Cg_old = 0;
15         // cStar
16         cStar = 0;
17     }
18     ...
19 }
```

Listing 4.16: Generalisierungsalgorithmus der Generalizer Klasse

```
1  /**
2   * Generalize a L-System by adding non-deterministic rules
3   * @return Generalized L-System
4   */
5  public LSystem generalize() {
6      do {
7          // Exclude  $S \rightarrow A$  from set combinations
8          var productionRulesWithoutS = new ArrayList<>(LStar.
9              ➔ getProductionRules());
10         if (productionRulesWithoutS.remove(0) == null)
11             throw new RuntimeException("No_rule_S_→_A_found");
12         // Generate all possible merging rule pairs  $P \quad L^*$ 
13         var combinations = Sets.combinations(new HashSet<>(
14             ➔ productionRulesWithoutS), 2);
15         var minimalCosts = Float.MAX_VALUE;
16         LSystem minimalLSystem = null;
17         // Find a pair  $p_i$  with the minimal  $C_g(L^* + \{p_i\}, L^*)$ 
18         for (var c : combinations) {
19             var LStarMerged = LStar.merge(c);
```



```

18         float costs = Cg(LStarMerged, LStar);
19         if (costs < minimalCosts) {
20             minimalCosts = costs;
21             minimalLSystem = LStarMerged;
22         }
23     }
24     if (minimalCosts >= 0) break;
25     cStar = minimalCosts - Cg_old;
26     Cg_old = minimalCosts;
27     LStar = minimalLSystem;
28 } while (cStar <= 0);
29
30 return LStar;
31 }

```

Listing 4.17: Kostenfunktion zum Vergleich zweier L-Systeme

```

1 /**
2  * Calculates the costs of editing a grammar leveraging the grammar
3  * length and the grammar edit distance both weighted with a weight
4  * balancing w0      [0.0, 1.0]
5  */
6 private float Cg(LSystem lStar, LSystem lPlus) {
7     return w0 * (L(lStar) - L(lPlus)) + (1 - w0) * Dg(lStar);
8 }

```

Listing 4.18: Längenfunktion über ein L-System

```

1 /**
2  * Measure and return the length of a L-System
3  * @param lSystem L-System to be measured
4  * @return L-System length
5  */
6 private int L(LSystem lSystem) {
7     // Set size of the alphabet M
8     var M_size = lSystem.getAlphabet().size();
9     // Sum of RHS symbols over all production rules
10    var sum_rulesRHSs = lSystem.getProductionRules().stream()
11        .map(r -> r.getRhs().length())
12        .mapToInt(Integer::intValue)
13        .sum();
14    return M_size + sum_rulesRHSs;
15 }

```

Listing 4.19: Grammar Edit Distance

```
1  /**
2   * Grammar edit distance: Overall costs to convert a grammar to another
3   * by a set of merging operations  $M(L+ \rightarrow L^*)$ 
4   */
5  private float Dg(LSystem lStar) {
6      // Grammar edit distance
7      var editDistance = 0;
8      // Get all multi-module production rules
9      var rules = lStar.getProductionRules().stream()
10         .filter(rule  $\rightarrow$  rule.getLhs().length() > 1)
11         .collect(Collectors.toList());
12
13     // Go through all merging operations (merging two rules)
14     while (!rules.isEmpty()) {
15         var operation1 = rules.get(0);
16         // Find corresponding rules
17         var otherOperations = rules.stream()
18            .filter(r  $\rightarrow$  !r.equals(operation1) && r.getLhs().equals(
19                 $\Rightarrow$  operation1.getLhs()))
20            .collect(Collectors.toList());
21
22         rules.remove(operation1);
23
24         if (!otherOperations.isEmpty()) {
25             var iterator = otherOperations.iterator();
26             do {
27                 //  $Ds(M\_A, M\_B)$ 
28                 var operation = iterator.next();
29                 editDistance += Ds(operation1.getRhs(), operation.getRhs());
30                 rules.remove(operation);
31             } while (iterator.hasNext());
32         }
33
34     return editDistance;
35 }
```

Listing 4.20: String Edit Distance

```
1  /**
2   * Calculate and return the edit distance between two strings
3   * @param M_A String A
4   * @param M_B String B
5   * @return Edit distance between A and B
6   */
7  private float Ds(String M_A, String M_B) {
8      return Modules.editDistanceOptimized(M_A, M_B);
9  }
```

Listing 4.21: Modules Klasse mit Funktion zur Ermittlung der String Edit Distance

```
1  public class Modules {
2      ...
3      /**
4       * Calculate and return the edit distance between two strings
5       * Time complexity: O(m*n)
6       * Space complexity: O(m)
7       * @param str1 String A
8       * @param str2 String B
9       * @return Edit distance between A and B
10      */
11     public static int editDistanceOptimized(String str1, String str2) {
12         int len1 = str1.length();
13         int len2 = str2.length();
14         int [][] DP = new int[2][len1 + 1];
15         for (int i = 0; i <= len1; i++) DP[0][i] = i;
16         for (int i = 1; i <= len2; i++) {
17             for (int j = 0; j <= len1; j++) {
18                 if (j == 0) DP[i % 2][j] = i;
19                 else if (str1.charAt(j - 1) == str2.charAt(i - 1)) DP[i %
20                     ↳ 2][j] = DP[(i - 1) % 2][j - 1];
21                 else DP[i % 2][j] = 1 + Math.min(DP[(i - 1) % 2][j], Math.
22                     ↳ min(DP[i % 2][j - 1], DP[(i - 1) % 2][j - 1]));
23             }
24         }
25     }
```

Der Algorithmus editDistanceOptimized ist aus Praktikabilitätsgründen aus [1] entnommen.

Verteilung der Transformationsparameter

Listing 4.22: Estimator Klasse zur Erstellung einer Verteilung über Transformationsparameter

```
1  /**
2   * Estimator class to create a distribution of different
3   * transformation parameters
4   */
5  public class Estimator {
6      ...
7      public Estimator(Random randomizer) {
8          /// Initialization
9          parameters = new HashMap<>();
10         this.randomizer = randomizer;
11     }
12     ...
13 }
```

Listing 4.23: Verteilungsalgorithmus der Estimator Klasse

```
1  ...
2  /**
3   * Create transformation parameter distribution of a given tree
4   * @param tree Tree the parameters are distributed from
5   */
6  public void estimateParameters(TreeNode<TemplateInstance> tree) {
7      // Determine different parameters
8      tree.getData().getParameters().forEach((key, value) -> parameters.
9          ➡ putIfAbsent(key, new HashMap<>()));
10     // Iterate tree and extract parameters
11     for (var node : tree) {
12         if (node == null || node.isEmpty()) continue;
13         var templateID = node.getData().getTemplate().getId();
14         for (var p : node.getData().getParameters().entrySet()) {
15             var parameterEntry = parameters.get(p.getKey());
16             var templateEntry = parameterEntry.get(templateID);
17             if (templateEntry == null) {
18                 var valueList = new ArrayList<Float>();
19                 valueList.add(p.getValue().floatValue());
20                 parameterEntry.put(templateID, valueList);
21             } else {
22                 templateEntry.add(p.getValue().floatValue());
23             }
24         }
25     }
26 }
```

```
24     }
25 }
26 ...
```

Listing 4.24: Abrufen eines zufälligen Parameters aus der Verteilung für ein Template

```
1 ...
2 /**
3  * Estimate and return a parameter for a given template by name
4  * @param parameter Parameter name
5  * @param templateID Corresponding template
6  * @return Estimated parameter value
7  */
8 public float estimateParameterValueForTemplate(String parameter, int
    ↳ templateID) {
9     var entries = parameters.get(parameter).get(templateID);
10    return entries.get(randomizer.nextInt(entries.size()));
11 }
12 ...
```

Listing 4.25: Berechnung des Durchschnitts eines Parameters für ein Template

```
1 ...
2 /**
3  * Calculate and return the average value for a parameter for a
4  * given template by name
5  * @param parameter Parameter name
6  * @param templateID Corresponding template
7  * @return Averaged parameter value
8  */
9 public float averageParameterValueForTemplate(String parameter, int
    ↳ templateID) {
10    var entries = parameters.get(parameter).get(templateID);
11    return (float) entries.stream().mapToDouble(v -> v).average().orElse(0);
12 }
13 ...
```

5 Evaluierung

Die Synthese von Verzweigungsstrukturen wird in den vorangegangenen Kapiteln konzeptionell betrachtet und in einem Softwareprojekt umgesetzt. Im Folgendem werden Teilaspekte an einem fortlaufenden Beispiel evaluiert und bewertet. Diese Aspekte umfassen

- das Nutzen von Templates
- die Erstellung und Visualisierung von Verzweigungsstrukturen
- Aufbau einer Baumstruktur
- Extrahierung von Regeln und Mustern
- Komprimierung von L-Systemen
- Erweitern von L-Systemen
- Synthese der Ähnlichkeitsstrukturen
- Auswirkung der Gewichtungparameter

Templates

Die Nutzung von Templates als Terminale einer Grammatik findet Anwendung in vielen wissenschaftlichen Arbeiten. Aliaga u. a. liefert hierzu eine umfassende Übersicht [3]. Mit der Verwendung einer allgemeinen Repräsentation mithilfe von *turtle*-Befehlen, wird eine Wiederverwendbarkeit der genutzten Templates sichergestellt. Die in dieser Arbeit erstellte Software nutzt ein minimalistisches System zum Einlesen der Templates aus einer textbasierten Datei, während die Forschung zur inversen prozeduralen Modellierung den Einsatz von neuronalen Netzen als vielversprechende Methode Strukturen zu erkennen und Regeln abzuleiten, zeigt.

Während neuronale Netze eine Eingabestruktur analysieren und in einer baumähnlichen Struktur organisieren, knüpft diese Arbeit hier an und nutzt stattdessen aus Praktikabilitätsgründen eine benutzergeführte Anordnung von Templates zu einer Verzweigungsstruktur.

Beispiel: Die Template-Zeichenketten haben folgende Form:

```
1 FX
2 F-FX
3 F+FX
4 F [-FX] +FY
5 F [-FX] FY
6 F [FX] +FY
7 F [-FX] [FY] +FZ
8 F [+FX] F [-FY] FZ
```

Abbildung 5.1: templates.txt

F , $-$, $+$, $[]$ und $]$ sind dem Turtle-Algorithmus bekannte Befehlssymbole. Alle anderen Symbole (z.B. X , Y , Z) stellen Verzweigungsvariablen dar, um anzuzeigen, an welcher Stelle der Templatestruktur eine neue Verzweigung abgehen kann.

Erstellung und Visualisierung

Durch die Ausführung bestimmter *turtle*-Befehle der template-basierten Struktur, wird diese visualisiert. Darum bietet es sich an, einfache grafische Elemente zu nutzen, um Verzweigungsstrukturen sichtbar zu machen (z.B. Canvas). Das Programm nutzt stattdessen interaktive Elemente innerhalb eines JavaFX Pane gegenüber statischen Elementen, um die Strukturen zu zeichnen, da der Prozess der Erstellung benutzergeführt und damit interaktiv sein muss. So können Elemente genutzt werden, mit denen der Benutzer kommunizieren kann (z.B. klickbare Kreise). Es ist nun möglich Verzweigungsstrukturen in einem simplen Arbeitsablauf zu erstellen.

Beispiel: Aus der Erstellung der Verzweigungsstruktur ergibt sich:

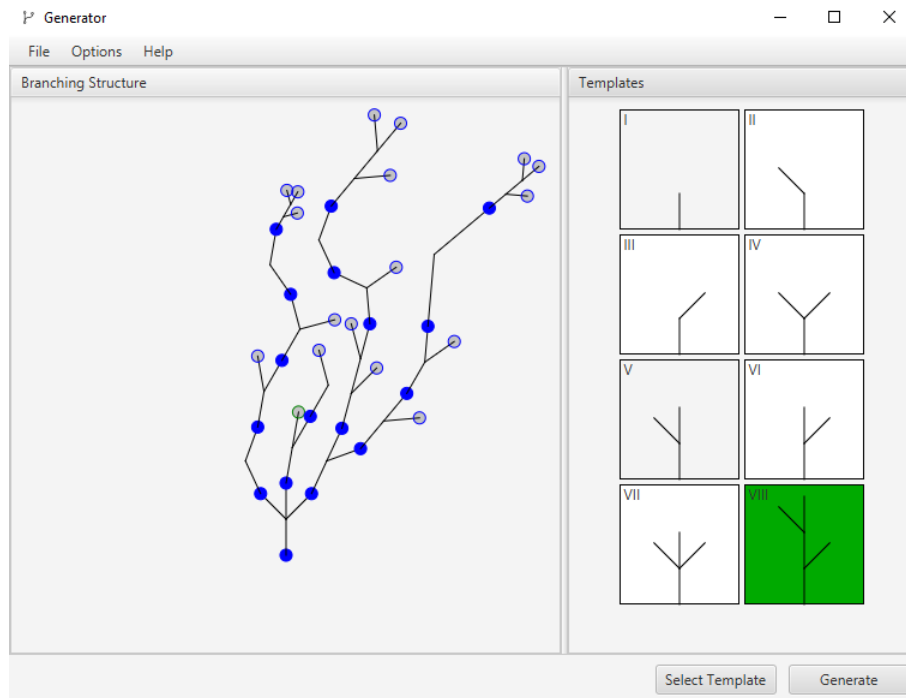


Abbildung 5.2: Grafische Benutzeroberfläche nach der Erstellung einer Basisstruktur

Aufbau einer Baumstruktur

Baumähnliche Strukturen eignen sich gut, um Transformationsparameter und topologische Anordnung von Tempaltes datenstrukturell zu trennen. Ansätze zur ganzheitlichen Betrachtung von Eingabestrukturen, also das Einbinden der Transformationen, und zur Trennung von Transformation und Topologie sind Gegenstand der Forschung. Das von Benes u. a. vorgestellte System nutzt beispielsweise einen ganzheitlichen Ansatz, um sog. *Guides* zu erstellen, welche Teilsysteme der Eingabestruktur beschreiben. Hier unterscheiden sich Strukturen, die zwar identische Verzweigungen aufweisen, sich aber in Ausprägungen von Transformationsparametern (z.B. der Winkel von Verzweigungen) voneinander entfernen. Auch in der Arbeit [23] von Stava u. a. zeigt sich eine Organisation von „Clustern“ mit Transformationen, die in eine Signifikanzbewertung einfließen.

Auf der anderen Seite zeigen Nishida u. a. und Guo u. a., dass zum Einen eine separate Untersuchung der Transformationen mithilfe eines zweiten, auf seine Aktivität zugeschnittenen, neuronalen Netz [17], zum Anderen, dass die meisten räumlichen Transformationen die Erkennung mittels neuronalem Netz nicht signifikant beeinflussen, sinnvoll ist.

Diese Arbeit legt der Fokus auf die datenstrukturelle Trennung von Transformationen und topologischer Anordnung und erstellt somit eine Baumstruktur, die Template-Instanzen als Knoten und räumliche Transformationen als eingehende Kanten darstellt.

Beispiel: Aus der Eingabestruktur ergibt sich folgender Baum (die räumlichen Transformationen, die die eingehenden Kanten jedes Knotens darstellen, sind hier nicht visualisiert):

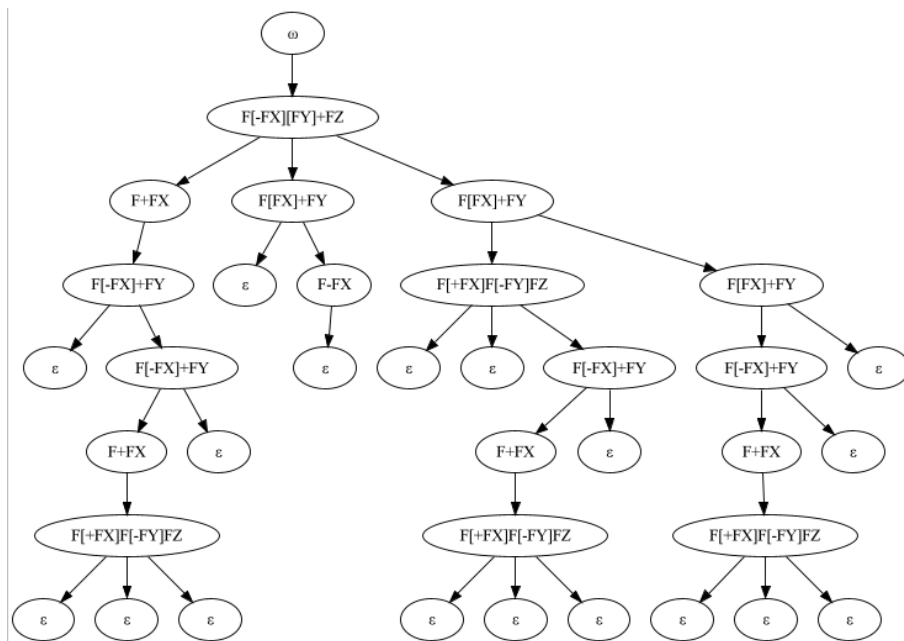


Abbildung 5.3: Baumstruktur der erstellten Verzweigungsstruktur

Extrahieren von Regeln und Mustern

Die in 3.2 vorgestellte Methodik zum Inferieren eines L-Systems aus einer Baumstruktur stellt einen Algorithmus vor, der auf die spezielle Baumstruktur zugeschnitten ist. Die L-Systeme entsprechen lediglich der Eingabestruktur und beinhalten keine Transformationen.

Beispiel: Ausgeführte Ersetzungssysteme werden über einen JavaFX Dialog visualisiert:

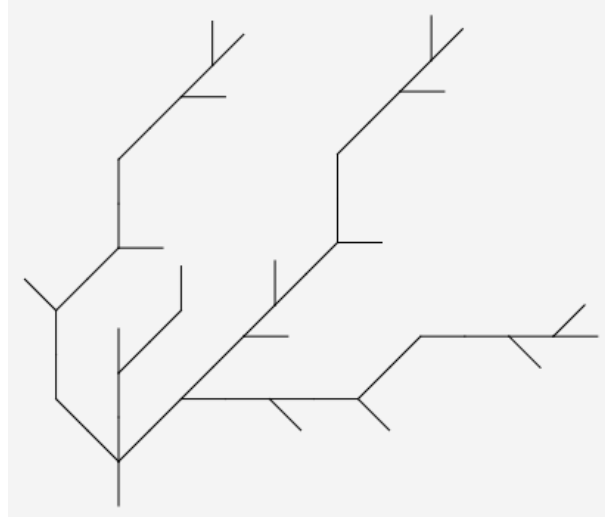


Abbildung 5.4: Inferriertes L-System

Die Zeichenkettenrepräsentation des L-Systems ($\mathcal{L} = \{M, \omega, R\}$) lautet:

```

1 LSystem{
2   [F, S, A, B, C, D, E, G, H, I, J, K, L, M, N, O, P, Q, R, T,
   ➡ U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k],
3   S,
4   [S -> A, A -> F[-FB][FC]+FD, B -> F+FE, C -> F[FG]+FH, D ->
   ➡ F[FI]+FJ, E -> F[-FK]+FL, G -> _, H -> F-FM, I -> F[+FN
   ➡ ]F[-FO]FP, J -> F[FQ]+FR, K -> _, L -> F[-FT]+FU, M ->
   ➡ _, N -> _, O -> _, P -> F[-FV]+FW, Q -> F[-FX]+FY, R ->
   ➡ _, T -> F+FZ, U -> _, V -> F+Fa, W -> _, X -> F+Fb, Y
   ➡ -> _, Z -> F[+Fc]F[-Fd]Fe, a -> F[+Ff]F[-Fg]Fh, b -> F
   ➡ [+Fi]F[-Fj]Fk, c -> _, d -> _, e -> _, f -> _, g -> _,
   ➡ h -> _, i -> _, j -> _, k -> _]
5 }
```

Das inferrierte L-System zeigt, dass die Eingabestruktur richtig in eine Grammatik überführt wurde. Dabei steht der Unterstrich (`_`) für das leere Wort ε und damit für die leeren Knoten der aufgebauten Baumstruktur.

Komprimieren des L-Systems

Die Zeichenkettenrepräsentation des inferrierten L-Systems zeigt einige Redundanzen auf. Zum Beispiel bilden

$$1 \quad P \rightarrow F[-FF+FF[+F]F[-F]F]+F$$

und

$$1 \quad Q \rightarrow F[-FF+FF[+F]F[-F]F]+F$$

das gleiche Muster. Um solche Redundanzen zu entfernen, wird das L-System in der vorgestellten Komprimierungs-Pipe reduziert. Hierbei werden identische, maximale Unterbäume und deren leere Kindknoten zusammengefasst. Der Gewichtungssparameter für das Beispiel wird auf 0.5 gesetzt. Es ergibt sich folgender Baum:

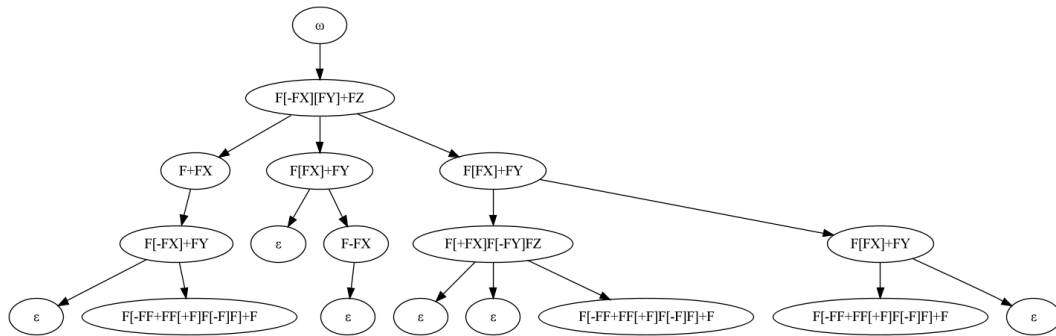


Abbildung 5.5: Komprimierter Baum

Die Baumstruktur zeigt, dass der Komprimierungsalgorithmus die maximalen Subbäume erfolgreich reduziert. Das komprimierte L-System setzt sich wie folgt zusammen:

```

1 LSystem{
2     [F, S, A, B, C, D, E, G, H, I, J, K, L, M, N, O, P],
3     S,
4     [S -> A, A -> F[-FB][FC]+FD, B -> F+FE, C -> F[FG]+FH, D ->
        ➡ F[FI]+FJ, E -> F[-FK]+FL, G -> _, H -> F-FM, I -> F[+FN
        ➡ ]F[-FO]FP, J -> F[FL]+FG, K -> _, L -> F[-FF+FF[+F]F[-F
        ➡ ]F]+F, M -> _, N -> _, O -> _, P -> F[-FF+FF[+F]F[-F]F
        ➡ ]+F]
5 }

```

Generalisieren des L-Systems

Der vorgestellte Generalisierungsalgorithmus setzt das L-System von der Eingabestruktur ab, in dem Produktionsregeln miteinander verbunden und mit einer Wahrscheinlichkeit versehen werden. Der Gewichtungparameter wird für das Beispiel auf 0.5 gesetzt. Es ergibt sich das finale L-System:

```
1 LSystem{
2   [F, S, E, G, K, M, N, A],
3   S,
4   [S -> A, E -> F[-FK]+FA, G -> _, K -> _, M -> _, N -> _, A
      ➡ -> F-FM, A -> F[-FA][FA]+FA, A -> F[FG]+FA, A -> F[FA]+
      ➡ FG, A -> F+FE, A -> F[+FN]F[-FA]FA, A -> F[-FF+FF[+F]F
      ➡ [-F]F]+F, A -> F[FA]+FA, A -> _]
5 }
```

Die Produktionsregelmengen zeigt einige Regeln auf, die dasselbe Ziel haben. Sie werden bei ihrer Anwendung zufällig ausgewählt.

Synthese

Führt man das generalisierte L-System aus, ergeben sich die Ähnlichkeitsstrukturen. Eine Evaluierung wird stichprobenartig durchgeführt, erweist sich aber als schwierig, da es bei der Erstellung der Ausgabestrukturen um Wahrscheinlichkeiten beim Auftreten gewisser Muster handelt. Da die Algorithmen eine akzeptable Lösung eines schwierigen Problems liefern sollen, ist die Bewertung der Ergebnisse subjektiv.

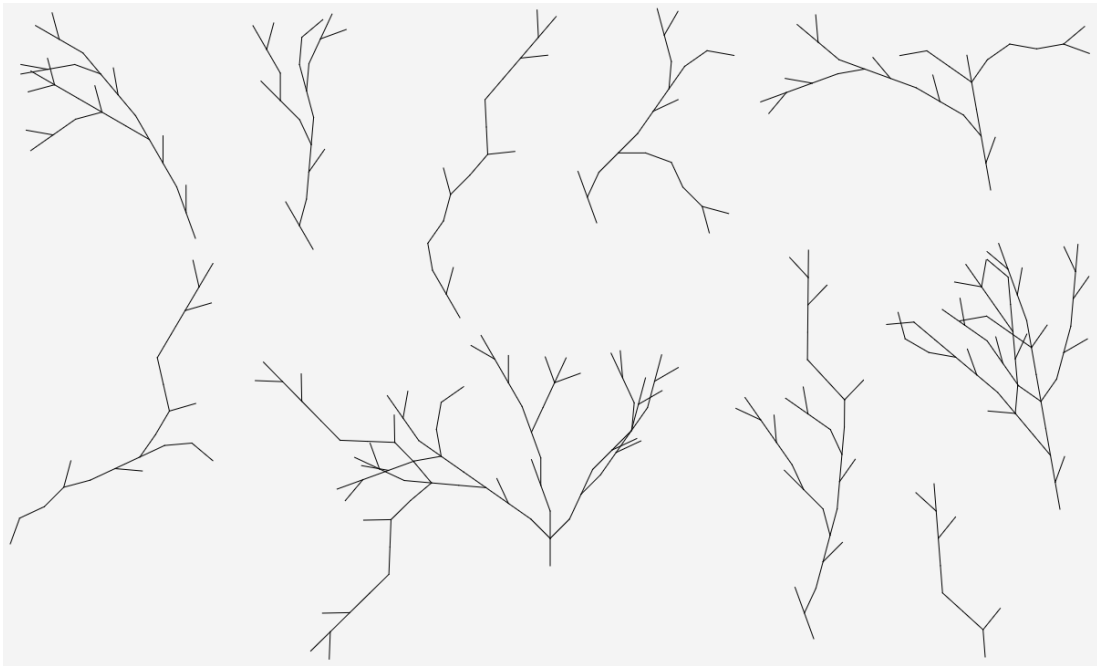


Abbildung 5.6: Synthetisierte Verzweigungsstrukturen

Vergleicht man die Eingabestruktur mit den synthetisierten Strukturen, weisen diese gewisse Eigenschaften auf (siehe folgende Abbildung):

- Wiederholtes Auftreten einer Struktur, die beim Synthetisieren als maximaler Unterbaum erkannt wurde (rot)
- Anwendung von benutzerdefinierten Transformationsparametern (blau)
- Ähnliche Häufigkeiten (grün)
- Isoliertes Auftreten von Templates des maximalen Subbaums, die über den Unterbaum hinaus einzeln vorkommen (gelb)

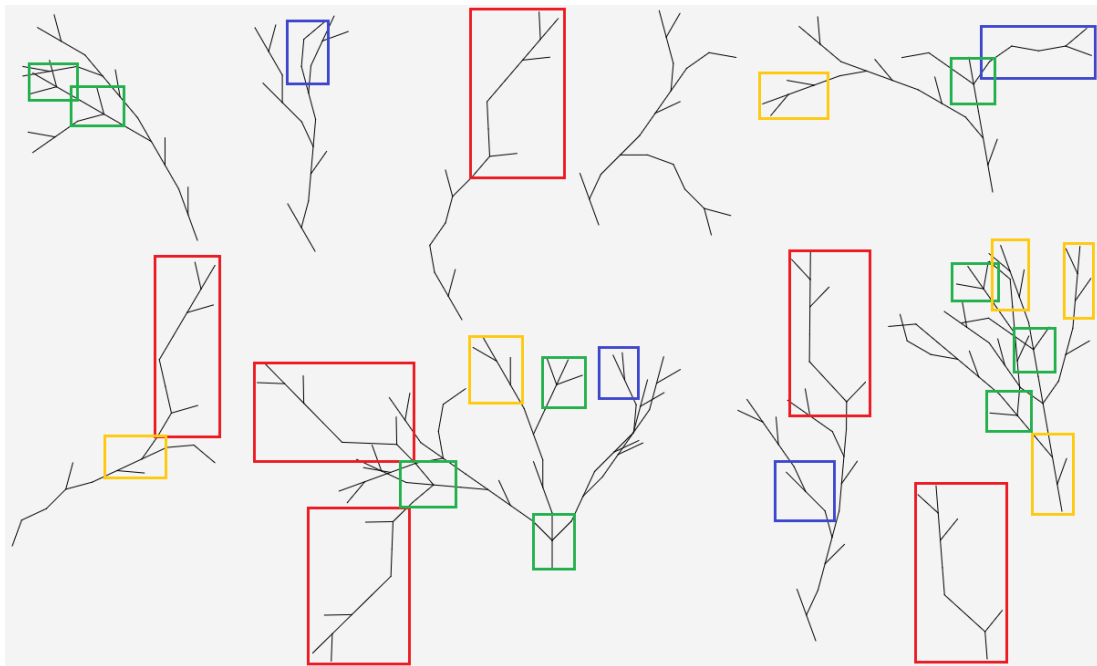


Abbildung 5.7: Untersuchung der synthetisierten Verzweigungsstrukturen

Auswirkung der Gewichtungparameter

Der Parameter w_l des vorgestellten Komprimierungsalgorithmus gewichtet eine Funktion, welche die Kosten eines L-Systems ermittelt, sodass der Algorithmus L-Systeme mit unterschiedlich großer Produktionsregelmenge anders behandelt. Hierbei bewertet ein Wert von 0.0 eine große Produktionsregelmenge höher. Entspricht w_l einem Wert von 1.0, wird eine kleinere Produktionsregelmenge überbewertet. Der Durchschnittswert 0.5, der im Beispiel verwendet wird, sorgt für eine moderate Menge an Produktionsregeln bei moderater Länge der einzelnen Regel. Der im Generalisierungsalgorithmus vorgestellte Gewichtungparameter w_0 legt fest, inwiefern die Länge oder *String Edit Distance* zweier Grammatiken von Bedeutung sind. Bei einem Wert von 0.0 überbewertet der Algorithmus die Metrik zur Berechnung der Anzahl Operationen, um eine Grammatik in eine zweite zu überführen. Der Wert 1.0 entspricht einer Überbewertung der Längenmetrik. Im Beispiel wird der Durchschnittswert 0.5 verwendet.

w_l und w_0 sind im Programm als *Rula application ratio* und *Merge application ratio* bezeichnet und können vom Benutzer vor der Generierung festgelegt werden.

6 Fazit und Ausblick

Diese Arbeit beschreibt die Konzeption und Implementierung eines prozeduralen Systems zur Erstellung von Verzweigungsstrukturen, die einer Eingabestruktur ähneln. Dabei werden Anforderungen untersucht, die zur Erstellung einer individuellen Software nötig sind. Eine intuitiv nutzbare Anwendung zur benutzergesteuerten Strukturierung von eingelesenen Templates und anschließender Generierung von Ähnlichkeitsstrukturen, ist im Rahmen dieser Arbeit entstanden.

Fazit

L-Systeme eignen sich gut, um Verzweigungsstrukturen mathematisch zu beschreiben und so in einem Programm zu verwenden. Sie können durch Ausführung und Interpretation durch einen Logo-Turtle-Algorithmus visualisiert und mittels grafischer Bedienelemente sichtbar gemacht werden. Die Steuerung von Gewichtungsparemtern durch den Benutzer verhindert unnötige *Trial and Error*-Szenarien, da das Ergebnis des Prozesses gesteuert werden kann.

Ausblick

Die Eingabestrukturen lassen sich ohne erneute Konzeptionierung in andere Strukturen, die ebenfalls einer Baumtopologie entsprechen, überführen. L-Systeme beschreiben Baumstrukturen grundlegend und können diese durch deren Auflösung visualisieren. So können bspw. Websites, die durch das *DOM* spezifiziert sind, auf diese Weise interpretiert und erstellt werden. Hierzu muss die Software um eine Bereitstellung des erzeugten, generalisierten L-Systems erweitert werden.

Eine Weiterführung des Softwareprojekts kann durch den Einsatz neuronaler Netze zum

Lernen struktureller Regeln und Ableiten von Transformationsparametern erreicht werden. Auf der einen Seite lässt sich die Erstellung von Ähnlichkeitsstrukturen durch neuronale Netze automatisieren und vereinfachen, auf der anderen Seite entsteht dadurch ein initialer Mehraufwand der Implementierung.

Weiter lassen sich die Konzepte zur Synthese zweidimensionaler Strukturen in zukünftigen Arbeiten in den dreidimensionalen Raum überführen. Zum Schluss ist das Erkennen von Überlappungen von Strukturen ein schwieriges Problem der Computergrafik und kann durch zusätzliche Algorithmen umgesetzt werden.

Literaturverzeichnis

- [1] AASHISH1995: *Edit Distance / DP-5*. <https://www.geeksforgeeks.org/edit-distance-dp-5/>. – Zugriffen: 25.01.2021
- [2] ALHALAWANI, Sawsan ; YANG, Yong-Liang ; LIU, Han ; MITRA, Niloy: Interactive Facades Analysis and Synthesis of Semi-Regular Facades. In: *Computer Graphics Forum* 32 (2013), 05
- [3] ALIAGA, Daniel G. ; DEMIR, Ilke ; BENES, Bedrich ; WAND, Michael: Inverse Procedural Modeling of 3D Models for Virtual Worlds. In: *ACM SIGGRAPH 2016 Courses*. New York, NY, USA : Association for Computing Machinery, 2016 (SIGGRAPH '16). – URL <https://doi.org/10.1145/2897826.2927323>. – ISBN 9781450342896
- [4] HRUSCHKA, Dr. P. ; STARKE, Dr. G.: *arc42 Softwarearchitektur-Template*. <https://arc42.de/template/>. – Pragmatisches Muster für die Erstellung, Dokumentation und Kommunikation von Software- und Systemarchitekturen
- [5] BENES, Bedrich ; STAVA, Ondrej ; MECH, Radomir ; MILLER, Gavin: Guided Procedural Modeling. In: *Computer Graphics Forum* 30 (2011), 05, S. 325–334
- [6] CHOMSKY, N.: Three models for the description of language. In: *IRE Transactions on Information Theory* 2 (1956), Nr. 3, S. 113–124
- [7] DAELEMANS, Walter: Colin de la Higuera: Grammatical inference: learning automata and grammars: Cambridge University Press, 2010, iv + 417 pages. In: *Machine Translation* 24 (2010), 12, S. 291–293
- [8] DEUSSEN, Oliver ; LINTERMANN, Bernd: *Digital Design of Nature: Computer Generated Plants and Organics*. 1st. Springer Publishing Company, Incorporated, 2010. – ISBN 3642073638

- [9] GUO, Jianwei ; JIANG, Haiyong ; BENES, Bedrich ; DEUSSEN, Oliver ; ZHANG, Xiaopeng ; LISCHINSKI, Dani ; HUANG, Hui: Inverse Procedural Modeling of Branching Structures by Inferring L-Systems. In: *ACM Trans. Graph.* 39 (2020), Juni, Nr. 5. – URL <https://doi.org/10.1145/3394105>. – ISSN 0730-0301
- [10] HIGUERA, Colin De la: *Grammatical Inference: Learning Automata and Grammars*. USA : Cambridge University Press, 2010. – ISBN 0521763169
- [11] LINDENMAYER, A: Mathematical models for cellular interactions in development. I. Filaments with one-sided inputs. In: *Journal of theoretical biology* 18 (1968), March, Nr. 3, S. 280—299. – URL [https://doi.org/10.1016/0022-5193\(68\)90079-9](https://doi.org/10.1016/0022-5193(68)90079-9). – ISSN 0022-5193
- [12] MARTINOVIC, Andelo ; VAN GOOL, Luc: Bayesian Grammar Learning for Inverse Procedural Modeling, 06 2013, S. 201–208
- [13] MCQUILLAN, Ian ; BERNARD, Jason ; PRUSINKIEWICZ, Przemyslaw: *Algorithms for Inferring Context-Sensitive L-Systems*. S. 117–130, 01 2018. – ISBN 978-3-319-92434-2
- [14] MERRELL, Paul ; SCHKUFZA, Eric ; LI, Zeyang ; AGRAWALA, Maneesh ; KOLTUN, Vladlen: *Interactive Furniture Layout Using Interior Design Guidelines*. In: *ACM SIGGRAPH 2011 Papers*. New York, NY, USA : Association for Computing Machinery, 2011. – URL <https://doi.org/10.1145/1964921.1964982>. – ISBN 9781450309431
- [15] MÜLLER, Pascal ; WONKA, Peter ; HAEGLER, Simon ; ULMER, Andreas ; VAN GOOL, Luc: Procedural Modeling of Buildings. In: *ACM Trans. Graph.* 25 (2006), 07, S. 614–623
- [16] MULLER, M. M. ; TICHY, W. F.: Case study: extreme programming in a university environment. In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, 2001, S. 537–544
- [17] NISHIDA, Gen ; GARCIA-DORADO, Ignacio ; ALIAGA, Daniel G. ; BENES, Bedrich ; BOUSSEAU, Adrien: Interactive Sketching of Urban Procedural Models. In: *ACM Trans. Graph.* 35 (2016), Juli, Nr. 4. – URL <https://doi.org/10.1145/2897824.2925951>. – ISSN 0730-0301
- [18] PARISH, Yoav ; MÜLLER, Pascal: Procedural Modeling of Cities, 08 2001, S. 301–308

- [19] PRUSINKIEWICZ, P: Graphical Applications of L-Systems. In: *Proceedings on Graphics Interface '86/Vision Interface '86*. CAN : Canadian Information Processing Society, 1986, S. 247–253
- [20] PRUSINKIEWICZ, P. ; LINDENMAYER, Aristid: *The Algorithmic Beauty of Plants*. Berlin, Heidelberg : Springer-Verlag, 1990. – ISBN 0387972978
- [21] SMELIK, Ruben M. ; TUTENEL, Tim ; BIDARRA, Rafael ; BENES, Bedrich: A Survey on Procedural Modelling for Virtual Worlds. In: *Comput. Graph. Forum* 33 (2014), September, Nr. 6, S. 31–50. – URL <https://doi.org/10.1111/cgf.12276>. – ISSN 0167-7055
- [22] STAVA, O. ; PIRK, S. ; KRATT, J. ; CHEN, B. ; MECH, R. ; DEUSSEN, O. ; BENES, B.: Inverse Procedural Modelling of Trees. In: *Comput. Graph. Forum* 33 (2014), September, Nr. 6, S. 118–131. – URL <https://doi.org/10.1111/cgf.12282>. – ISSN 0167-7055
- [23] STAVA, Ondrej ; BENES, Bedrich ; MECH, Radomir ; ALIAGA, Daniel ; KRISTOF, Peter: Inverse Procedural Modeling by Automatic Generation of L-systems. In: *Computer Graphics Forum* 29 (2010), 05, S. 1467–8659
- [24] TALTON, Jerry ; LOU, Yu ; DUKE, Jared ; LESSER, Steve ; MECH, Radomir ; KOLTUN, Vladlen: Metropolis Procedural Modeling. In: *ACM Transactions on Graphics* 30 (2011), 04
- [25] TALTON, Jerry ; YANG, Lingfeng ; KUMAR, Ranjitha ; LIM, Maxine ; GOODMAN, Noah ; MECH, Radomir: Learning design patterns with Bayesian grammar induction. In: *UIST'12 - Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (2012), 10. ISBN 978-1-4503-1580-7
- [26] VAZQUEZ, Pablo ; SIDDI, Francesco: *About Blender*. <https://www.blender.org/about/>. 2017. – Blender Foundation (2002)
- [27] XIAO, Jianxiong ; FANG, Tian ; ZHAO, Peng ; OFEK, Eyal ; QUAN, Long: Image-based Façade Modeling. In: *ACM Transactions on Graphics* 27 (2008), 12, S. 161
- [28] ZHANG, Song-Hai ; ZHANG, Shao-Kui ; LIANG, Yuan ; HALL, Peter: A Survey of 3D Indoor Scene Synthesis. In: *Journal of Computer Science and Technology* 34 (2019), 05, S. 594–608

Glossar

HAW Hamburg Die HAW Hamburg ist die formalige Fachhochschule am Berliner Tor.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Template-basierte Synthese von Verzweigungsstrukturen mittels L-Systemen

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original