HAW HAMBURG

BACHELOR THESIS
Leonhard Ken Weich

# Analyse des Markov Junior Algorithmus am Beispiel der Generierung von Labyrinthen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Leonhard Ken Weich

# Analyse des Markov Junior Algorithmus am Beispiel der Generierung von Labyrinthen

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. Peer Stelldinger

Eingereicht am: 25. Juli 2024

**Leonhard Ken Weich**

**Thema der Arbeit**

Analyse des Markov Junior Algorithmus am Beispiel der Generierung von Labyrinthen

**Stichworte**

Markov Junior, Prozedurale Generierung, Labyrinth

**Kurzzusammenfassung**

Die prozedurale Generierung ist heutzutage vielseitig verwendet, da die automatische Generierung von Inhalten viel Zeit und Aufwand erspart. Sie ist ein stetig wachsender Bereich. Darunter gibt es den neuen Markov Junior Algorithmus von Maxim Gumin. Dieser erzeugt Rastergrafiken, welche durch das Grundprinzip der visuellen Ersetzungsregeln generiert werden. Eine Ersetzungsregel beinhaltet ein Eingabemuster, welches durch das entsprechende Ausgabemuster der Regel im Raster ersetzt wird.

Diese Arbeit beschäftigt sich damit, den Markov Junior Algorithmus näher darzustellen und zu analysieren. Dafür wird in dieser Arbeit ein neuer Ansatz zur Generierung von perfekten Labyrinthen mit dem Markov Junior Algorithmus entwickelt. Ein perfektes Labyrinth hat keine geschlossenen Kreise oder unerreichbaren Stellen. Der Ansatz legt den Fokus auf die Kontrollierbarkeit gewisser Eigenschaften, wie den Verlauf des Lösungsweges und der Anzahl an direkten Abzweigungen vom Lösungsweg. Dabei wird dies durch einen modularen Ansatz gelöst, welcher zuerst den Lösungsweg, dann die Abzweigungspunkte vom Lösungsweg und zuletzt die restlichen Wege generiert.

Das Ergebnis zeigt, dass der Markov Junior Algorithmus perfekte Labyrinthe generieren kann und zudem die Kontrolle über die genannten Eigenschaften erlaubt. Der Markov Junior Algorithmus beweist sich zudem als ein mächtiges Tool zur prozeduralen Generierung von simplen bis mittel komplexen Grafiken, bei dem durch wenig Code viel dargestellt werden kann. Jedoch hat die Umsetzung des Ansatzes gezeigt, dass Markov Junior durch Einschränkungen, wie z. B. das Fehlen von Variablen und die schnelle Unübersichtlichkeit der Ersetzungsregeln, eine erhöhte Komplexität aufweist.

**Leonhard Ken Weich**

**Title of Thesis**
Analysis of the Markov Junior algorithm using the example of maze generation

**Keywords**
Markov Junior, Procedural Content Generation, Maze

**Abstract**

Procedural content generation is widely used today as the automatic generation of content saves a lot of time and effort. It is a steadily growing field. Among its innovations is the new Markov Junior algorithm by Maxim Gumin. This algorithm generates grid graphics, which are produced based on the fundamental principle of visual rewrite rules. A rewrite rule includes an input pattern, which is replaced in the grid by the corresponding output pattern of the rule.
This work aims to present and analyze the Markov Junior algorithm in more detail. To achieve this, a new approach to generating perfect mazes using the Markov Junior algorithm is developed. A perfect maze has no loops or unreachable areas. The approach focuses on the controllability of certain properties, such as the course of the solution path and the number of direct branches from the solution path. This is achieved through a modular approach that first generates the solution path, then the branching points from the solution path, and finally the remaining paths.
The results show that the Markov Junior algorithm can generate perfect mazes and also allows control over the mentioned properties. Furthermore, the Markov Junior algorithm proves to be a powerful tool for the procedural content generation of simple to moderately complex graphics, where a lot can be depicted with little code. However, the implementation of the approach has shown that Markov Junior exhibits increased complexity due to limitations such as the lack of variables and the rapid lack of clarity of the rewrite rules.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Abkürzungen

**MJ** Markov Junior.

**SAW** self-avoiding walk.

# 1 Einleitung

## 1.1 Motivation

Prozedurale Generierung ist heutzutage nicht mehr wegzudenken, da sie in vielen Bereichen genutzt wird. Vor allem in der Videospielindustrie hat sie einen großen Einfluss. Bei der prozeduralen Generierung geht es um die automatische Generierung von Inhalten, welche meistens graphischer Natur sind. Daraus ergibt sich der Vorteil, dass die Erstellung solcher Inhalte nicht manuell geschehen muss, was häufig viel Zeit und Aufwand kostet. [25, 26, 27]

Wie bei vielen weit verbreiteten Konzepten gibt es auch bei der prozeduralen Generierung verschiedenste Verfahren. Eines der neueren Verfahren ist der Markov Junior (MJ) Algorithmus von Maxim Gumin [12], welcher Rastergrafiken erzeugt. Dieser verfolgt das Grundprinzip von Ersetzungsregeln. Eine Ersetzungsregel beschreibt ein Eingabemuster, was in dem Raster gesucht wird und ein dazugehöriges Ausgabemuster, welches das Aufkommen des Eingabemusters im Raster ersetzt. Diese werden in Knoten umhüllt. Die Knoten bringen jeweils eine Eigenschaft mit sich, welche bestimmt, wie die Regeln angewendet werden oder wie das Raster manipuliert wird. Dadurch ergibt sich die Möglichkeit komplexere Programme zu erstellen. Eine Anordnung mehrerer Knoten stellt ein MJ Programm dar. So wird von einem Startraster über die Knoten eine Grafik erzeugt. Vergleichbar ist das Grundprinzip mit herkömmlichen Grammatiken.
Aufgrund der Neuheit des Algorithmus ist dieser noch wenig erforscht. Da die prozedurale Generierung ein immer wachsendes Feld ist, ist es von Interesse, neue Verfahren zu präsentieren und zu verstehen, um diese dadurch besser etablieren zu können. Deswegen setzt sich diese Arbeit den Fokus, den MJ Algorithmus darzustellen und zu analysieren.

Dafür wird jener in dieser Arbeit für die Generierung von Labyrinthen genutzt. Ein Labyrinth ist heutzutage als ein Rätsel zu verstehen, in welchem es einen Start und ein Ziel gibt, welche durch einen Weg verbunden sind. Dazu gibt es mehrere Abzweigungen und Sackgassen, wodurch das Finden des Lösungsweges erschwert wird[1].

[1] https://en.wikipedia.org/wiki/Maze - Zugriffsdatum: 20.07.2024

In dieser Arbeit liegt der Fokus auf perfekten Labyrinthen. Diese haben keine geschlossenen Kreise und keine unerreichbaren Stellen [21]. Dessen manuelle Erstellung kann schnell aufwendig werden, weshalb die prozedurale Generierung von Labyrinthen ein schon erforschtes Feld ist. So gibt es auch schon einige Algorithmen zur Generierung von Labyrinthen [9]. Daher versucht diese Arbeit mithilfe von MJ einen neuen Ansatz zur Generierung von Labyrinthen zu erstellen.

Dabei wird die Generierung in zwei Teile geteilt: die Generierung des Lösungsweges und die Generierung der Abzweigungen. Die Generierung der Abzweigungen kann auch in zwei Teile unterteilt werden: die Generierung der Abzweigungspunkte von dem Lösungsweg und die Generierung der restlichen Wege. Der self-avoiding walk (SAW) bildet hierbei das Grundprinzip der Generierung der Wege. Ein SAW ist ein zufälliger Weg, der nicht in sich selbst läuft.
Bereits Kim et al. [15] haben zuerst den Lösungsweg und danach die Abzweigungen generiert. Dieses Prinzip haben sie jedoch außerhalb des Kontextes von MJ verwendet. Der SAW wurde von Maxim Gumin [12] beispielhaft schon in MJ umgesetzt. Die Generierung der restlichen Wege baut auf einem Ansatz von Bellot et al. [2] auf, welcher aber ebenfalls außerhalb des Kontextes von MJ implementiert wurde. Die Umsetzung der bestehenden Konzepte (außer des SAWs) in MJ wird somit durch diese Arbeit neu eingeführt. Die Idee und Umsetzung der Generierung des Lösungsweges und der Abzweigungspunkte werden ebenfalls durch diese Arbeit neu eingeführt. Dazu ist der allgemeine Ansatz in der Kombination der einzelnen Komponenten ein in dieser Arbeit entwickeltes Konzept.

## 1.2 Ziele der Arbeit

In dieser Arbeit soll der MJ Algorithmus vorgestellt werden. Im Mittelpunkt steht dabei die Erklärung des grundlegenden Konzeptes der visuellen Ersetzungsregeln und die Beschreibung der einzelnen Knoten, welche charakteristisch für MJ sind. Dies hat das übergeordnete Ziel, den MJ Algorithmus selbstständig anwenden zu können.
Dazu soll unter der Nutzung des MJ Algorithmus ein neuer Ansatz zur Generierung von perfekten Labyrinthen erstellt werden. Im Fokus steht dabei die Kontrollierbarkeit von einigen Eigenschaften des Labyrinthes, da diese in den herkömmlichen Algorithmen vernachlässigt wird [9]. Das Ziel dabei ist, dass es der neue Ansatz ermöglichen soll, den Verlauf des Lösungsweges des Labyrinthes und die Anzahl an direkten Abzweigungen vom Lösungsweg steuern zu können.

Ein weiteres Ziel ist es, auf der Grundlage der Erstellung des Ansatzes zeigen zu können, was mit MJ möglich ist. Dazu wird der MJ Algorithmus kritisch hinterfragt und dessen Stärken und Schwächen analysiert. Zusätzlich soll das Potenzial von MJ dargestellt werden.

# 2 Stand der Technik

Markov Junior (MJ) wurde 2022 veröffentlicht [12] und ist somit relativ neu, weshalb es bisher kaum Forschungsarbeiten mit konkretem Bezug dazu gibt. Cooper [3] hat direkte Inspiration von den Ersetzungsregeln aus MJ geholt. Mit diesem Prinzip hat Cooper Bedingungen erstellt, um mit den Regeln zu beschreiben, wie ein generiertes Spiellevel gelöst oder gespielt werden kann.

Das allgemeine Konzept der Ersetzungsregeln ist ebenfalls in Grammatiken vorzufinden und im Grundprinzip ist MJ eng verbunden zu diesen. Grammatiken wurden schon häufig für prozedurale Generierung verwendet.
Van Rozen und Heijn [22] analysieren den Nutzen von Grammatiken in Bezug auf die prozedurale Generierung. Dazu verwenden sie eine Generierung eines einfachen Verlieses auf einem Raster als Beispiel. Die Grammatik zur Generierung benutzt die Kacheln des Rasters für die Ersetzungsregeln, welche denen aus MJ sehr nahekommen.
Dormans und Bakkes [5] nutzen Grammatiken, um Spiellevel zu erzeugen. Sie teilen dafür die Generierung in zwei Teile auf: Missionen und Raum. Für die Missionen (Aufgaben im Spiel) wird eine Graph grammar verwendet, um einen Missionsgraphen zu erstellen. Dieser stellt dar, was für Missionen es in welcher Kombination und Reihenfolge gibt. Eine Graph grammar unterscheidet sich von einer normalen Grammatik insofern, als dass statt Strings Graphen ersetzt und erzeugt werden. Für die Generierung des Raumes (die Spielkarte) wird nach Anpassungen des Missionsgraphens eine Shape grammar verwendet. Eine Shape grammar ersetzt und erzeugt geometrische Figuren.
Dormans [4] erweitert dies um die Generierung von Spielmechaniken mithilfe einer Graph grammar.
Merrell [19] präsentiert eine Methode, bei der die Graph grammar aus einem Beispiel abgeleitet wird, um so lokal ähnliche Inhalte zu generieren.
Auch für die Generierung von Labyrinthen können Grammatiken verwendet werden. Etchebehere und Eliseo [6] nutzen L-Systeme, um Labyrinthe unterschiedlicher Komplexität zu generieren. L-Systeme sind nah verwandt mit Grammatiken, wobei der Hauptunterschied eine parallele Ausführung der Regeln ist.

Im Bereich der Labyrinthgenerierung beschäftigen sich einige Forschungsarbeiten mit den graphbasierten Algorithmen.
Shah et al. [23] und Kozlova, Brown und Reading [16] geben jeweils einen Überblick über einen Teil dieser Algorithmen.
Gabrovsek [9] geht dabei zusätzlich noch auf die Performance der einzelnen Algorithmen ein und analysiert die Schwierigkeit der Labyrinthe. Unter diesen Aspekten werden die einzelnen Algorithmen miteinander verglichen.

Weitere Forschungsarbeiten beschäftigen sich mit anderen Verfahren, die eine bessere Kontrolle über die generierten Labyrinthe zulassen.
Ashlock, Lee und McGuinness [1] generieren unterschiedliche Typen von Labyrinthen und experimentieren dazu mit einigen Eigenschaften der Labyrinthe, wie z. B. den Sackgassen. Sie nutzen dafür die such-basierte prozedurale Generierung, bei der man iterativ den Inhalt generiert und nach dem am besten passenden Inhalt sucht. Dies wird meist durch Evaluationsfunktionen unterstützt, welche auswerten, wie gut der generierte Inhalt ist, um so zukünftige Generierungen zu verbessern (vergleichbar mit evolutionären Algorithmen) [27].
Kim et al. [15] nutzen ebenfalls die such-basierte prozedurale Generierung und legen dabei den Fokus darauf, möglichst viel vom Labyrinth anpassen zu können. Dazu teilen Kim et al. die Generierung in die Erzeugung des Lösungsweges und die Erzeugung des restlichen Labyrinthes für eine bessere Kontrolle ein.
Peachey [20] nutzt ein mehrschrittiges Verfahren mit einem neuronalen Netz, um so Labyrinthe einer gewissen Schwierigkeit zu generieren. Das neuronale Netz erzeugt hierbei die Parameter für den eigentlichen Labyrinthgenerator unter Betrachtung der gewollten Schwierigkeit. Die generierten Labyrinthe werden dann über ihre Schwierigkeit analysiert und diese Ergebnisse werden zum Lernen des neuronalen Netzes genutzt.
Nelson und Smith [24] nutzen einen logischen Programmieransatz, um dadurch Labyrinthe und spielbare Verliese zu generieren. Dazu kodieren sie die Logik der Domäne des Labyrinthes/Verlieses als logisches Problem und fügen Constraints hinzu, welche vorschreiben, wie der Inhalt aussehen soll. Das logische Problem wird dann gelöst, sodass man diese Mengen an Lösungen als Grundlage für den zu generierenden Inhalt nutzen kann.

Die Kombination aus MJ und der Generierung von Labyrinthen wurde lediglich innerhalb des Projektes von Maxim Gumin [12] kurz erwähnt. Dabei hat er drei der bekannteren

graphbasierten Generierungsalgorithmen (Recursive Backtracker[1], Aldous-Broder[2] und Wilson's[3]) in MJ umgesetzt. Zusätzlich hat er zum Vergleich eine herkömmliche Implementierung des Wilsons's Algorithmus vorgezeigt.

[1] http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking - Zugriffsdatum: 20.07.2024
[2] http://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm - Zugriffsdatum: 20.07.2024
[3] http://weblog.jamisbuck.org/2011/1/20/maze-generation-wilson-s-algorithm - Zugriffsdatum: 20.07.2024

# 3 Grundlagen

Dieses Kapitel gibt eine Einführung in die wichtigsten Themen, welche eine Voraussetzung für diese Arbeit sind. Dabei wird zuerst die prozedurale Generierung eingeführt. Darauffolgend wird erklärt, wie der Markow-Algorithmus funktioniert und zuletzt wird erläutert, was ein Labyrinth ist und was dieses ausmacht.

## 3.1 Prozedurale Generierung

Prozedurale Generierung ist das automatische Generieren von meist graphischen Inhalten durch Algorithmen. Dabei können die generierten Inhalte aber auch nicht graphisch sein, wie z. B. Musik oder Gedichte, wie bspw. Haikus. [25]

Zwei Aspekte stehen bei der prozeduralen Generierung im Fokus: Parametrisierung und Zufälligkeit.
Durch die Parametrisierung eröffnet sich die Möglichkeit, das Generierte zu kontrollieren und nach seinem Belieben anzupassen, indem man gewisse Eigenschaften des generierten Inhaltes vorgibt (z. B. Größe/Länge des Inhaltes oder Anzahl der Fenster bei einem Haus).
Die Zufälligkeit sorgt dafür, dass generierte Inhalte unterschiedlich aussehen, sodass man durch einen Algorithmus unbegrenzte Variationen eines Inhaltes generieren kann. Hierbei bedeutet zufällig aber nicht, dass ohne Wissen zufällig etwas gemacht wird. Dies würde zu unlogischen Inhalten führen, welche normalerweise nicht das Ziel der prozeduralen Generierung sind. Der Zufall in der prozeduralen Generierung wird daher meist unter gewissen Bedingungen, die man erfüllen muss, verwendet. [26, 27]

Nennenswert ist dabei, dass es auch prozedurale Generierung ohne Zufälligkeit gibt, also einen rein deterministischen Algorithmus. Der Nutzen eines solchen Algorithmus liegt meist in der Speichermenge. Wenn man eine gesamte generierte Welt speichert, nimmt dies viel Speicher ein. Ein Algorithmus jedoch, wird nicht so viel Speicher einnehmen.

Zusätzlich zu diesem Vorteil ergeben sich noch zwei weitere Vorteile durch die prozedurale Generierung. Die Automatisierung erspart das manuelle Designen der Inhalte und spart somit viel Zeit und Aufwand.
Dazu bietet die prozedurale Generierung Möglichkeiten, die manuell nicht umgesetzt werden könnten: Bspw. kann durch automatische Echtzeitgenerierung den Spielenden eines Videospiels eine potenziell unendliche Spielerfahrung gegeben werden. [26, 27]

Aufgrund dieser Aspekte ist ein großer Anwendungsbereich der prozeduralen Generierung die Videospielindustrie. Dort wird von kleinen Gegenständen wie Waffen, z. B. in *Borderlands* (Gearbox Software 2009), bis hin zu Sternensystemen, z. B. in *Elite* (Acornsoft 1984), alles Mögliche automatisch generiert. [25, 26, 27]

## 3.2 Markow-Algorithmus

Der Markow-Algorithmus, benannt nach Andrei Markow, ist ein Stringersetzungssystem und gilt als Turing-vollständig. Als Grundlage existieren ein Alphabet, auf dem der Algorithmus fußt, und eine Menge an Substitutionsregeln von Symbolen aus dem Alphabet, welche als Grammatik bezeichnet werden.[1] [14]

### 3.2.1 Formale Definition

Das Alphabet ist eine nicht leere, endliche Menge an Symbolen. Ein String ist dabei eine endliche Sequenz aus Symbolen des Alphabets.
Seien $p$ und $q$ zwei Strings aus dem Alphabet und $p$ soll durch $q$ ersetzt werden, so ist eine Substitutionsregel ein Ausdruck der Form

$$p \to q \text{ oder } p \to. \ q.$$

Wobei $\to$ und . keine Symbole aus dem Alphabet sind. $\to.$ kennzeichnet hierbei eine Terminationsregel, nach dessen Anwendung der Algorithmus beendet wird.
Der Ausdruck

$$p \to(.) \ q$$

stellt $p \to q$ oder $p \to. \ q$ dar.

[1] https://en.wikipedia.org/wiki/Markov_algorithm - Zugriffsdatum: 20.07.2024

Die Grammatik ist eine endliche, geordnete Liste an Substitutionsregeln

$$p_i \rightarrow(.)\ q_i,\ \text{mit}\ i = 1, 2, \ldots, I.$$

Eine Regel i hat eine höhere Priorität als eine Regel $j$, wenn $i < j$ ist.
Bei einem Eingabestring s aus dem Alphabet kann der Markow-Algorithmus wie folgt beschrieben werden:

1. Setze $i = 1$.
2. Schaue die $i$-te Substitutionsregel der Grammatik an und suche nach dem am weitesten links auftretenden $p_i$ in $s$. Gibt es kein Vorkommen von $p_i$, dann springe zu Schritt 4.
3. Ersetze das gefundene Vorkommen von $p_i$ in $s$ durch $q_i$. Ist die $i$-te Regel eine Terminationsregel, beende den Algorithmus. Ansonsten springe zu Schritt 1.
4. Setze $i = i + 1$. Wenn $i > I$ ist, beende den Algorithmus. Ansonsten springe zu Schritt 2. [14]

## 3.3 Labyrinth

Ein Labyrinth besteht aus einem Weg oder mehreren Wegen, wobei es normalerweise einen Startpunkt und ein Ziel zwischen den Wegen gibt.[2]

Geschichtlich gesehen gibt es zwei Hauptarten, welche zu unterscheiden sind.
Anfänglich war die Definition eines Labyrinthes ein einzelner Weg, welcher ohne Abzweigungen mit Richtungswechseln von einem äußeren Startpunkt zum Ziel im Mittelpunkt des Labyrinthes führt (Abbildung 3.1a).
Heutzutage jedoch wird ein Labyrinth als eine Sammlung von abgezweigten Wegen verstanden, welche einen Lösungsweg vom Start zum Ziel beinhalten. Das Ziel ist es, diesen zu finden, weshalb Labyrinthe gegenwärtig als Rätsel angesehen werden. Diese Art an Labyrinth wird häufig auch als Irrgarten bezeichnet (Abbildung 3.1b).[3]

In dieser Arbeit liegt der Fokus nur auf den Irrgärten, welche hier auch als Labyrinth bezeichnet werden.

[2] https://en.wikipedia.org/wiki/Maze - Zugriffsdatum: 20.07.2024
[3] https://de.wikipedia.org/wiki/Labyrinth - Zugriffsdatum: 20.07.2024


(a) (b)

Abbildung 3.1: Gegenüberstellung eines klassischen Labyrinthes (a) und eines heutigen Labyrinthes/Irrgartens (b)
(a): https://de.wikipedia.org/wiki/Labyrinth - Zugriffsdatum: 20.07.2024,
(b): https://en.wikipedia.org/wiki/Maze - Zugriffsdatum: 20.07.2024

### 3.3.1 Kategorisierung von Labyrinthen

Labyrinthe können unterschiedlichste Eigenschaften und Formen annehmen. Abhängig von diesen Eigenschaften kann das Verständnis eines Labyrinthes komplett anders sein. Daher ist es wichtig zu klären, was für Labyrinthe in dieser Arbeit behandelt werden.

Labyrinthe kann man in sieben Kategorien einteilen [21], welche die wichtigsten Eigenschaften zusammenfassen:

- Dimension: Diese beschreibt, wie viele Koordinaten benötigt werden, um einen Punkt im Labyrinth beschreiben zu können. Ein Beispiel wäre ein klassisches Labyrinth auf Papier, bei dem jeder Punkt durch zwei Koordinaten beschrieben werden kann. Dieses Labyrinth wäre zweidimensional. Würde man mehrere zweidimensionale Labyrinthe übereinander schachteln und zusätzlich eine Bewegung zwischen den Etagen zulassen, hätte man ein dreidimensionales Labyrinth. Es hat nun eine Höhe und ein Punkt kann nur durch drei Koordinaten beschrieben werden.

- Hyperdimension: Hierbei geht es um das Objekt, welches sich durch das Labyrinth bewegt. In einem zweidimensionalen Labyrinth bewegt man einen Punkt durch dieses und hinterlässt eine Linie. Wenn man nun aber eine Linie durch das Labyrinth führt, welche eine Fläche hinterlässt, spricht man von einem Hyperlabyrinth. Vorstellen kann man sich dieses als einen Würfel, welcher mehrere Freiräume in einer Labyrinth-Struktur in sich geschnitzt hat. So würde man eine Linie von einer zur anderen Seite des Würfels laufen lassen, um das Labyrinth zu lösen. Dabei ist die Linie theoretisch unendlich lang, sodass die Enden immer außerhalb des Würfels

bleiben. Es muss daher darauf geachtet werden, dass sich die Linie nicht in einer Säule (fester Teil im Würfel, der kein Freiraum ist) verfängt.

- Topologie: Diese beschreibt, welche Geometrie der Raum hat, in dem sich das Labyrinth befindet. Ein Beispiel wäre ein Labyrinth auf einem Würfel.
- Tessellation: Diese beschreibt, welche Geometrie die einzelnen Zellen eines Labyrinthes haben. Ein Beispiel wäre ein Labyrinth mit hexagonalen Zellen.
- Routenführung: Diese beschreibt, welche Arten an Wegen ein Labyrinth besitzt. Ein Beispiel wären Sackgassen und ob ein Labyrinth diese besitzt.
- Textur: Diese beschreibt, wie das Design der einzelnen Wege im Labyrinth ist. Ein Beispiel dafür wäre die Länge von geraden Strecken in einem Labyrinth.
- Fokus: Dieser beschreibt, welche Art von Generierungstyp für das Labyrinth verwendet wurde. Ein Beispiel wäre ein Generierungsalgorithmus, welcher nur Wände von dem Labyrinth generiert, um so die gesamte Struktur zu erstellen.

In dieser Arbeit werden zweidimensionale, perfekte Gamma-Labyrinthe auf einer Ebene behandelt. *Perfekt* ist eine Unterkategorie der Routenführung. Ein perfektes Labyrinth hat keine geschlossenen Kreise und keine unerreichbaren Stellen. Von jedem Punkt im Labyrinth gibt es exakt einen Weg zu jedem anderen Punkt. Dadurch gibt es auch nur einen Lösungsweg. *Gamma* ist eine Unterkategorie der Tessellation. Ein Gamma-Labyrinth hat ein rechteckiges Feld, bei dem die Zellen Wege haben, die in rechten Winkeln abzweigen. Die vier fehlenden Kategorien sind vorerst nicht relevant und werden zum Teil noch im Verlauf dieser Arbeit angesprochen.

### 3.3.2 Generierung von Labyrinthen

Labyrinthe können schnell komplex werden und so auch deren manuelle Erstellung. Daher ist eine prozedurale Generierung von Labyrinthen sehr praktisch. Dabei gibt es unterschiedlichste Punkte, die zu beachten sind und im Folgenden erklärt werden.

#### Generierungstypen

Im vorherigen Unterkapitel wurde der *Fokus* angesprochen, welcher den Generierungstypen eines Labyrinthes beschreibt. Dabei werden zwei Arten voneinander unterschieden: Wand hinzufügende und Weg schnitzende Algorithmen. Wand hinzufügende Algorithmen arbeiten auf einem leeren Feld und fügen nacheinander alle Wände ein. Bezogen

auf die Realität kann das mit Heckenlabyrinthen vergleicht werden. Weg schnitzende Algorithmen hingegen arbeiten in einem schon gefüllten Feld und schnitzen die Wege des Labyrinthes nacheinander in das Feld. Auf die Realität bezogen kann dies mit Minensystemen verglichen werden. [21]

In dieser Arbeit liegen Weg schnitzende Algorithmen/Ansätze im Fokus.

### Labyrinthe als Graph

Ein Labyrinth mit den beschriebenen Eigenschaften aus dem vorherigen Unterkapitel kann als ein Graph dargestellt werden. Dafür betrachtet man ein zweidimensionales Raster, welches als Darstellung für das Labyrinth genutzt wird. Jede Kachel des Rasters wäre ein Knoten und sobald es einen Weg zwischen zwei Kacheln gibt, sind die Knoten über eine Kante miteinander verbunden.
Ein perfektes Labyrinth ist daher wie ein Spannbaum. Für die Generierung eines solchen Labyrinthes ist das Ziel daher einen Spannbaum zu generieren. Dabei ist bei dem Ausgangsgraphen für diesen Prozess jeder Knoten über eine Kante mit jedem seiner direkten Nachbarn verbunden. Sobald bei der Generierung eine Kante als Teil des Labyrinthes ausgewählt wird, wird die entsprechende Wand, die die Zellen getrennt hat, aufgebrochen.[4] [9]

[4] https://en.wikipedia.org/wiki/Maze_generation_algorithm - Zugriffsdatum: 20.07.2024

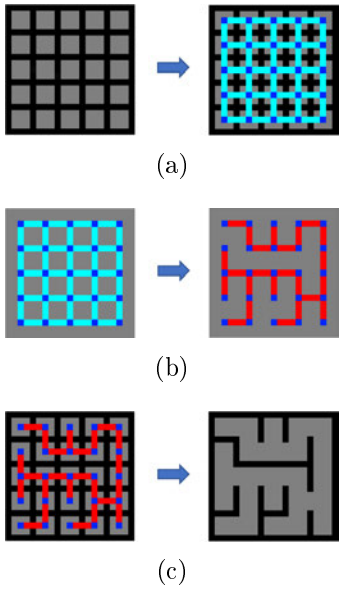


Abbildung 3.2: Darstellung des Konzeptes der Generierung von Labyrinthen mithilfe von Graphen

In Abbildung 3.2 ist dieses Konzept dargestellt. 3.2a zeigt ein Raster mit dessen dazugehörigem Graphen (Knoten in Dunkelblau und Kanten in Cyan). In 3.2b ist der isolierte Graph des Rasters zu sehen und ein Beispiel für einen Spannbaum auf diesem. In 3.2c sieht man, wie dieser Spannbaum genutzt wird, um das Labyrinth zu erzeugen.

### Generierungsalgorithmen

Graphbasierte Generierungsalgorithmen für Labyrinthe sind weit verbreitet. Zu den wohl bekanntesten Algorithmen gehören der rekursive Backtracker, Prim, Kruskal, Aldous-Broder und Wilson.[4] [9]

In dieser Arbeit spielt der rekursive Backtracker Algorithmus eine wichtigere Rolle. Dieser Algorithmus ist ein Weg schnitzender Algorithmus. Die Grundidee hinter dem Algorithmus ist eine zufällige Tiefensuche.

# 4 Markov Junior

Markov Junior (MJ) ist eine probabilistische Programmiersprache, bei der Programme aus einer Menge an Ersetzungsregeln und weiteren Konstrukten bestehen (siehe Kapitel 4.2), welche auf einem Raster (2D aber auch 3D) agieren. In dieser Arbeit liegt der Fokus auf einem zweidimensionalen Raster. Im Endeffekt erzeugt MJ zwei- und dreidimensionale Rastergrafiken. MJ und spezifisch dessen Ersetzungsregeln sind inspiriert von dem Markow-Algorithmus, weshalb es auch seinen Namen trägt. [12]

Eine probabilistische Programmiersprache ist speziell ausgelegt, um probabilistisches Programmieren zu ermöglichen. Probabilistisches Programmieren beschäftigt sich grob zusammengefasst mit dem Herausfinden von Eingabeparametern bei einer vorliegenden Funktion und einer Menge ihrer Ausgabedaten. Dabei wird durch Inferenz aus den Ausgabedaten mit einer gewissen Wahrscheinlichkeit berechnet, welche Eingaben zu dem gewünschten Ziel führen können. [18]

## 4.1 Vergleich zum Markow-Algorithmus

Da MJ direkt von dem Markow Algorithmus inspiriert ist, gibt es viele Überschneidungen. Jedoch gibt es auch einige Unterschiede, welche im Folgenden erklärt werden.

### 4.1.1 Gemeinsamkeiten

Vor allem im grundlegenden Konzept sind sich MJ und der Markow-Algorithmus ähnlich. Statt der Ersetzung einfacher Zeichenketten werden in MJ Kacheln aus einem Raster ersetzt. Die Kacheln werden durch ihre Farben dargestellt. Die Farben in MJ sind vergleichbar mit dem Alphabet des Markow-Algorithmus. Das Raster in MJ ist vergleichbar mit dem Eingabestring im Markow-Algorithmus und das MJ Programm mit allen Regeln kann mit der Grammatik im Markow-Algorithmus verglichen werden.

### 4.1.2 Unterschiede

Da die Ersetzungen auf einem mehrdimensionalen Raster angewendet werden, gibt es zwei Eigenschaften des Markow-Algorithmus, welche in MJ keine Anwendung finden und demnach anders umgesetzt werden müssen. Zum einen gibt es keinen natürlichen Ansatz, einen String in einen anderen String einzusetzen in höheren Dimensionen. Dafür gibt MJ vor, dass alle Eingabe- und Ausgabemuster der Ersetzungsregeln gleich lang sind. Zum anderen kann man nicht einfach wie bei dem Markow-Algorithmus das am weitesten links auftretende Vorkommen ersetzen. Als Lösung führt MJ zwei Möglichkeiten ein: ein zufälliges Vorkommen wählen oder alle möglichen Vorkommen wählen. Dadurch ist MJ nicht-deterministisch und verliert somit seine Turing-Vollständigkeit. [12]
Außerdem unterscheidet sich die Grundlogik des Algorithmus von MJ zu dem Markow-Algorithmus. In MJ gibt es im Allgemeinen keine Priorität von einzelnen Regeln und eine Terminationsregel gibt es ebenfalls nicht. Zusätzlich gibt es weitere Konstrukte zu den Ersetzungsregeln, welche mehr Möglichkeiten der Manipulation des Rasters erlauben. Diese werden im folgenden Kapitel 4.2 näher erläutert.

## 4.2 Knoten

Wie schon genannt, ist die Grundlogik des Algorithmus von MJ anders als bei dem Markow-Algorithmus. Grund hierfür sind Knoten, welche die Grundbausteine eines MJ Programms sind. Ein Knoten hat jeweils eine Eigenschaft, die beschreibt, wie einzelne Regeln angewendet werden oder wie das Raster manipuliert werden soll. Durch die Erweiterung der einfachen Ersetzungsregeln durch die Knoten ist es so möglich, komplexere Programme zu definieren.
Jeder Knoten hat notwendige Parameter, welche diesen ausmachen. Ein Knoten kann auch optionale Parameter haben, um so zusätzlich gewisse Eigenschaften des Knoten steuern zu können.
Es gibt drei Kategorien an Knoten, die man unterscheiden kann: *Rulenodes*, *Branchnodes* und spezielle Knoten.

### 4.2.1 Rulenodes

Rulenodes sind die grundlegenden Knoten von MJ. Sie sind Wrapper für Ersetzungsregeln und sagen aus, wie diese genau angewendet werden sollen. Ein Rulenode muss mindestens eine Ersetzungsregel beinhalten. Hat ein Rulenode mehr als eine Ersetzungsregel, werden alle seiner Regeln betrachtet. Wie die Anwendung der Regeln passiert, hängt von dem

jeweiligen Rulenode ab. Dabei besteht die Gemeinsamkeit darin, dass der Knoten im Normalfall so lange ausgeführt wird, bis keine Regeln mehr anwendbar sind. Es gibt drei verschiedene Rulenodes: *Onenode*, *Allnode* und *Parallelnode*.

**Ersetzungsregeln**

Die notwendigen Parameter eines Rulenodes sind die Ersetzungsregeln. Jede Regel besitzt ein Eingabemuster und ein Ausgabemuster. Bei mehreren Ersetzungsregeln wird jede innerhalb des Rulenodes einzeln definiert.

Das Alphabet in MJ für die Ersetzungsregeln besteht aus den Farben der Kacheln. Sind zwei Farben in einem Muster einer Regel nebeneinander, so bedeutet dies, dass sie im Raster direkt benachbart sind. Man kann sich ein Muster einer Regel also als eine direkte Abbildung eines Teils des Rasters vorstellen. Eine einfache abstrakte MJ Ersetzungsregel kann dann wie folgt aussehen:

Eingabemuster: (Weiß, Schwarz) - Ausgabemuster: (Weiß, Weiß)

Das Eingabemuster beschreibt also eine weiße und eine schwarze Kachel, welche direkt zueinander benachbart sind. Das Ausgabemuster beschreibt zwei weiße, benachbarte Kacheln. Wie beim Markow-Algorithmus wird das Eingabemuster durch das Ausgabemuster ersetzt. Diese Regel ersetzt also ein Vorkommen zweier Kacheln, welche weiß und schwarz sind und benachbart sind, durch zwei weiße benachbarte Kacheln auf dem Raster.
Da das Raster, auf dem wir agieren, zweidimensional ist, bietet MJ auch die Möglichkeit, Muster über zwei Dimensionen zu definieren.

Zusätzlich zu den Farben im Alphabet gibt es noch zwei weitere Konstrukte, die zum Alphabet gehören: *Wildcards* und *Unions*.
Wildcards stehen für jede Farbe. In dem Eingabemuster wird an der Stelle der Wildcard jede Farbe akzeptiert. In dem Ausgabemuster ist eine Wildcard dafür zuständig, dass die Farbe an dieser Stelle unberührt bleibt.
Eine Union wird vor einer Regel definiert und besteht aus einem Symbol und einer Menge an Farben. Das Symbol kann dann in den Eingabemustern genutzt werden und steht stellvertretend für die definierten Farben. Bei der Anwendung der Regeln wird eine zufällige Farbe aus der Menge gewählt. Man kann mehrere Unions mit unterschiedlichen Symbolen definieren.

Eine abstrakte Regel, die eine Union und Wildcards verwendet, kann folgende Struktur haben:

---

**Union**

*Parameter* Symbol: ?, Werte: (Schwarz, Weiß)

**Rulenode**

*Regel* Eingabemuster: (*, ?, *) - Ausgabemuster: (Weiß, Weiß, Weiß)

---

Zuerst wird eine Union erstellt, bei der das Fragezeichen für eine weiße oder eine schwarze Kachel steht. Die Wildcard wird hier durch einen Stern (*) dargestellt. Die Ersetzungsregel sagt somit aus, dass eine weiße oder schwarze Kachel und ihre Nachbarn (egal welche Farbe diese haben), durch drei weiße Kacheln ersetzt werden.

Ein Rulenode bietet zusätzlich an, statt eines Eingabe- oder Ausgabemusters auch eine Datei anzugeben, welche ein Kachelmuster enthält.

Jeder Rulenode hat einen optionalen Parameter, um die maximale Anzahl an Ausführungsschritten des Rulenodes zu bestimmen.

### Symmetrie

Standardmäßig wird ein Eingabemuster einer Regel in jeglicher Rotation betrachtet. Angenommen, man hat ein Eingabemuster (Weiß, Schwarz, Schwarz). Dies steht für eine weiße Kachel mit zwei benachbarten schwarzen Kacheln. Auch das umgedrehte Muster (Schwarz, Schwarz, Weiß) oder dasselbe Muster vertikal ausgerichtet, würden somit übereinstimmen, wenn diese in dem Raster gefunden werden.

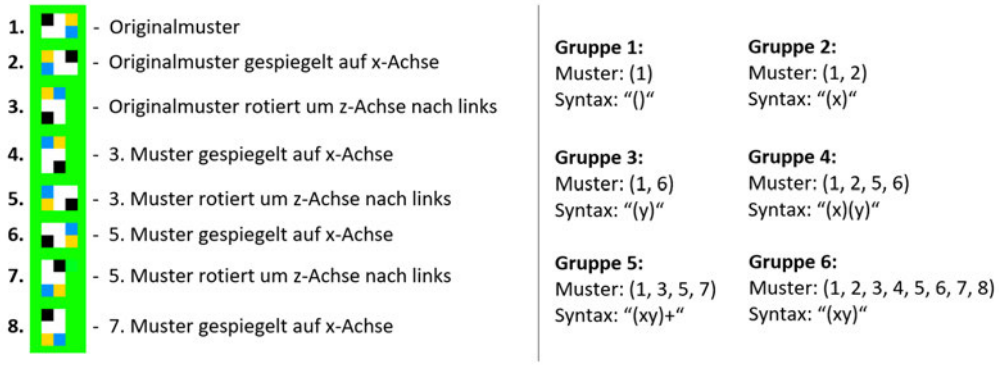



Abbildung 4.1: Symmetriegruppen für Eingabemuster der Ersetzungsregeln mit einem Beispielmuster

Dies kann für jede Regel durch den Symmetrieparameter gesteuert werden. Dieser erlaubt es, unter verschiedenen Symmetriegruppen zu wählen. Innerhalb einer Gruppe ist eine spezifische Auswahl an Rotationen des ursprünglichen Musters enthalten. Dabei unterscheidet man zwischen sechs verschiedenen Symmetriegruppen. Diese sind in Abbildung 4.1 dargestellt.

### Onenode

Der Onenode sucht alle Übereinstimmungen der Eingabemuster seiner Regeln auf dem Raster und wählt zufällig eine dieser zum Ersetzen aus. Dies entspricht der ersten Möglichkeit, welche zuvor angesprochen wurde, um in einer mehrdimensionalen Umgebung Ersetzungsregeln auszuführen.

### Allnode

Der Allnode hingegen setzt die zweite Möglichkeit zur Ersetzung von Mustern in mehrdimensionalen Räumen um, welche in Kapitel 4.1.2 angesprochen wurde. Er ersetzt alle Übereinstimmungen der Eingabemuster mit dem dazugehörigen Ausgabemuster. Dabei werden Überschneidungen von Ersetzungen verhindert. Wird ein Feld schon von einer Regel angefasst, wird dieses nicht mehr von anderen Regeln berührt. Die Wahl der Reihenfolge der Ersetzungen geschieht zufällig.

### Parallelnode

Der Parallelnode ist ähnlich zum Allnode. Er ersetzt ebenfalls alle Übereinstimmungen der Eingabemuster, achtet aber nicht auf Überschneidungen. Eine später definierte Er-

setzungsregel im Parallelnode würde daher eine frühere Regel überschreiben, wenn diese gleiche Felder im Raster betreffen.

Er hat zusätzlich noch einen optionalen Parameter, um die Wahrscheinlichkeit festzulegen, dass eine Übereinstimmung eines Eingabemusters einer Regel ersetzt wird.

### Beispielanwendung für Rulenodes

Angenommen, der Rulenode besteht aus folgender Ersetzungsregel:

Eingabemuster: (Weiß, Schwarz) - Ausgabemuster: (Weiß, Weiß)

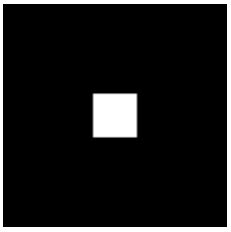


Abbildung 4.2: Auschnitt eines Beispielrasters mit schwarzem Hintergrund und einer weißen Kachel

Zusätzlich nehmen wir an, dass das Raster schwarz ist und auch mindestens eine weiße Kachel enthält (siehe Abbildung 4.2). Dann würde die Regel über die Zeit das schwarze Raster mit weiteren weißen Kacheln befüllen und zwar beginnend von den schon im Raster vorhandenen weißen Kacheln. Abhängig vom Rulenode geschieht das jedoch unterschiedlich, was im Folgenden verdeutlicht wird.

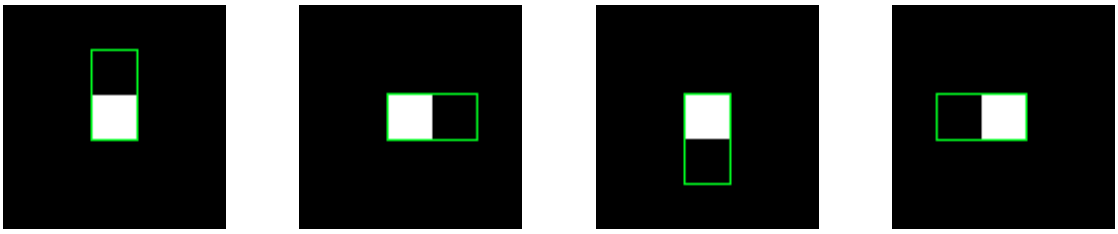
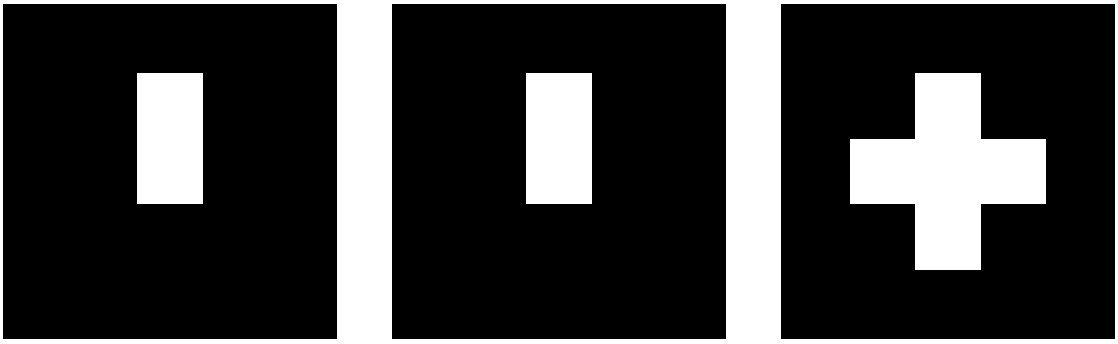


Abbildung 4.3: Alle Übereinstimmungen des Eingabemusters (Weiß, Schwarz) im Rasterauschnitt aus Abbildung 4.2

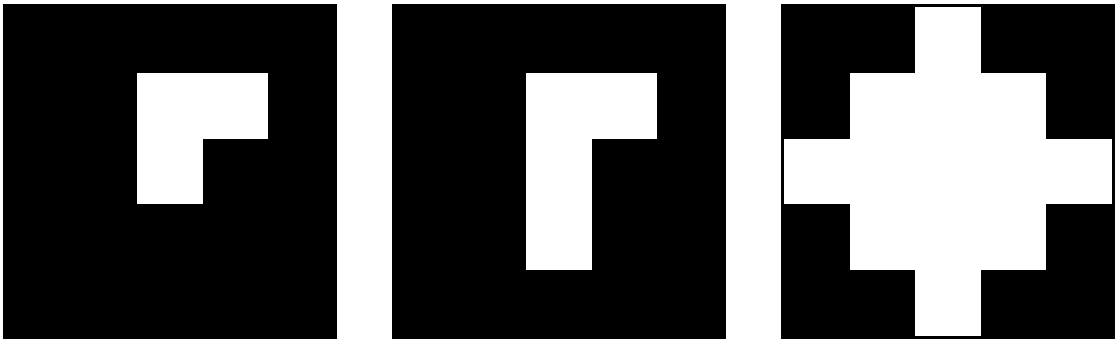
Bei dem Zustand in Abbildung 4.2 würde es für jeden der drei Rulenodes vier Übereinstimmungen des Eingabemusters geben (siehe Abbildung 4.3).



(a) Onenode (b) Allnode (c) Parallelnode

Abbildung 4.4: Ausschnitte des Rasters nach erster beispielhafter Anwendung der Regel

Im ersten Schritt der Ausführung ist zu beobachten, dass auch der Allnode wie der Onenode nur eine Übereinstimmung ersetzt, welche in dem Beispiel zufällig gleich sind (siehe Abbildung 4.4). Dies liegt daran, dass die weiße Kachel in der Mitte durch die erste Ausführung der Regel schon angefasst wird. Es wird nämlich Weiß durch Weiß ersetzt, was der Allnode auch als Veränderung versteht. Da der Allnode Überschneidungen von Regelanwendungen vermeidet, wendet er somit keine weitere Regel mehr an.



(a) Onenode (b) Allnode (c) Parallelnode

Abbildung 4.5: Ausschnitte des Rasters nach zweiter beispielhafter Anwendung der Regel

Nach der ersten Ausführung sind mehr weiße Kacheln vorhanden. Somit gibt es auch mehr Übereinstimmungen des Eingabemusters als zuvor. Dadurch ist bei der zweiten Ausführung nun auch der Unterschied zwischen einem One- und Allnode zu erkennen. Der Onenode hat nur eine Übereinstimmung ersetzt, während der Allnode alle möglichen (hier zwei) Übereinstimmungen ersetzt hat. Der Parallelnode ersetzt wie beschrieben alle Übereinstimmungen unabhängig der Überschneidungen (siehe Abbildung 4.5).

### Erweiterung von Rulenodes

MJ bietet die Möglichkeit, dass Ersetzungen nicht nur zufällig ausgewählt werden, sondern auch abhängig von den Begebenheiten des Rasters ausgewählt werden können. Dafür gibt es zwei Möglichkeiten: *Fields* und *Observations*.

Innerhalb eines Rulenodes können mehrere Fields definiert werden. Ein Field kann eine Regel lenken, in eine bestimmte Richtung ausgeführt zu werden. Dafür werden die betroffene Farbe, für die das Field gilt, und die Zielfarbe für die betroffene Farbe definiert. Jede Regel, die die betroffene Farbe enthält, ist von dem Field betroffen. Die Zielfarbe wird entweder angesteuert oder dient im Gegenteil als Farbe, von der sich die Regel versucht zu entfernen.
Dies wird umgesetzt durch die Berechnung eines einfachen Distanzfeldes von der Zielfarbe aus. Jede direkt benachbarte Kachel hätte somit den Wert 1 und die Nachbarn davon den Wert 2 etc. Damit kann entschieden werden, welche Ersetzung das beste Potenzial hat, sich der Zielfarbe zu nähern (oder zu entfernen). Ein Field muss daher auch definieren, auf welcher Farbe das Distanzfeld berechnet werden soll.
Ein Field wird innerhalb eines Rulenodes definiert. Die betroffene Farbe, die Zielfarbe und die Grundfarbe für das Distanzfeld sind die notwendigen Parameter. Angenommen, das Raster ist schwarz, hat eine weiße und eine gelbe Kachel. Dann kann ein Field für einen Rulenode wie folgt aussehen:

| **Rulenode** | |
|---|---|
| *Regel* | Eingabemuster: (Weiß, Schwarz) - Ausgabemuster: (Weiß, Weiß) |
| *Field* | betroffene Farbe: Weiß, Zielfarbe: Gelb, Grundfarbe: Schwarz |

Diese einfache Ersetzungsregel würde von der weißen Kachel aus Richtung gelber Kachel auf schwarzem Untergrund eine weiße Linie ziehen.
Ein Field hat zudem zwei optionale Parameter: den Nachrechnungsparameter und den Notwendigkeitsparameter. Der Nachrechnungsparameter sagt aus, ob nach jeder Anwendung einer Regel das Distanzfeld neu berechnet werden soll. Der Notwendigkeitsparameter sagt aus, ob die Regel nur ausgeführt werden soll, wenn die Grundfarbe für das Field auch existiert.

Auch *Observations* werden in einem *Rulenode* definiert und können dafür sorgen, Regeln zu lenken. Dies geschieht jedoch strenger als bei den *Fields*. Eine Observation definiert für eine Farbe, welchen Endzustand (Farbe auf dem Raster) sie annehmen soll, nach

Anwendung der betroffenen Regeln. So werden Regelanwendungen ausgewählt, welche zu diesem Zielzustand führen werden.

Sowohl Fields als auch Observations bieten an, zu steuern, wie streng Regeln gewählt werden, um das definierte Ziel zu erreichen. Dies kann über den Temperatur-Parameter eingestellt werden. Der Parameter wird auf den zum Field oder zur Observation dazugehörigen Rulenode gesetzt.

### 4.2.2 Branchnodes

Branchnodes können andere Knoten als ihre Kinder haben. Es gibt zwei Typen von Branchnodes: Einfache Branchnodes und manipulierende Branchnodes.

Einfache Branchnodes haben eine Auswirkung auf den Ablauf der Ausführung der Kindknoten. Es gibt zwei einfache Branchnodes: *Sequencenode* und *Markovnode*.

Manipulierende Branchnodes veränderen zuerst das aktuelle Raster und lassen dann alle Kindknoten nacheinander wie in einem Sequencenode auf dem veränderten Raster ausführen. Es gibt zwei manipulierende Branchnodes: *Mapnode* und *WaveFunctionCollapse Node*.

#### Sequencenode

Ein Sequencenode sorgt dafür, dass alle Kindknoten sequenziell ausgeführt werden.

#### Markovnode

Ein Markovnode funktioniert wie der Markow-Algorithmus. Der erste Knoten wird priorisiert ausgeführt. Ist dieser nicht mehr anwendbar, werden die nächsten Knoten angeschaut. Sobald einer angewendet wurde, springt man wieder zum ersten Knoten zurück und versucht diesen erneut anzuwenden.

#### Mapnode

Der Mapnode ersetzt das aktuelle Raster mit einem neuen Raster, welches sich in der Größe unterscheiden kann. Der Mapnode erlaubt es zu Beginn, Ersetzungsregeln zu definieren, bei denen die Eingabemuster im alten Raster gesucht werden und die Ausgabemuster im neuen Raster entsprechend angewendet werden. Daher ist es im Mapnode möglich, dass sich Eingabemuster und Ausgabemuster in der Größe unterscheiden.

Für den Mapnode ist die Skalierung ein notwendiger Parameter. Die Skalierung sagt aus, wie das neue Raster im Vergleich zum alten Raster skaliert werden soll. Dafür muss für jede Richtung $(x, y, z)$ eine Skalierung angegeben werden. Sollen die einzelnen Muster im Ausgaberaster die gleiche Proportion wie im Eingaberaster haben, so müssen die Ausgabemuster ebenfalls skaliert werden.
Angenommen, das alte Raster ist schwarz und hat gelbe Kacheln auf sich verteilt. Dann kann ein abstrakter Mapnode wie folgt aussehen:

---

**Mapnode**

*Parameter* Skalierung: (2, 2, 1)

*Regel* Eingabemuster: (Gelb) - Ausgabemuster: $\begin{pmatrix} \text{Weiß} & \text{Weiß} \\ \text{Weiß} & \text{Weiß} \end{pmatrix}$

**Kindknoten**

..... .....

**Kindknoten**

..... .....

---

Das neue Raster ist in $x$- und $y$-Richtung doppelt so groß und hat jede gelbe Kachel vom alten Raster durch weiße Kacheln auf dem neuen Raster ersetzt. Dabei sind die Proportionen der gelben Kacheln im ursprünglichen Raster beibehalten, da auch das Ausgabemuster skaliert wurde. Danach wird mit den inneren Knoten im Mapnode sequenziell auf dem neuen Raster standardmäßig fortgefahren.

### WaveFunctionCollapse Node

Ein WaveFunctionCollapse Node nutzt den WaveFunctionCollapse Algorithmus [11] von Maxim Gumin. Dieser ermöglicht das Generieren von Bildern, welche lokal ähnlich zu einem Eingabebild sind. Lokal ähnlich bedeutet, dass die generierten Bilder nur die $N \times N$ Muster enthalten, welche auch im Eingabebild zu finden sind. Zusätzlich soll die Verteilung der Muster ähnlich sein. [13]

Der WaveFunctionCollapse Node ersetzt das alte Raster mit einem gleichgroßen neuen Raster. Auf dem neuen Raster wird dann von einem Beispielbild oder einem selbst definierten *Tileset* mit dem WaveFunctionCollapse Algorithmus ein lokal ähnliches Bild generiert. Das Tileset ist eine Menge aus kleinen Kachelgrafiken und Regeln, wie diese nebeneinander angeordnet sein dürfen. Anhand dieser Regeln generiert der WaveFunctionCollapse Algorithmus die lokal ähnlichen Bilder. Zusätzlich kann man das alte Raster

nutzen, um anhand dessen Farben zu bestimmen, an welchen Stellen im neuen Raster der WaveFunctionCollapse Algorithmus angewendet werden soll.

#### Besonderheit von Branchnodes

Ist ein Branchnode ein Kind von einem anderen Branchnode, so wird der innere Branchnode wiederholend ausgeführt, bis kein Knoten in dem inneren Branchnode mehr übereinstimmt. Mapnodes und WaveFunctionCollapse Nodes sind davon ausgeschlossen.

### 4.2.3 Spezielle Knoten

Spezielle Knoten haben die Eigenschaft, wie Rulenodes das Raster verändern zu können, tun dies aber nicht mit einfachen Ersetzungsregeln und sind keine Branchnodes. Es gibt drei spezielle Knoten: *Pathnode*, *Convolutionnode* und *Conv Chain Node*.

#### Pathnode

Ein Pathnode erstellt zwischen zwei definierten Farben auf dem Raster einen Weg. Es wird standardmäßig der kürzeste Weg genommen. Bei mehreren Vorkommen der Farben wird der kürzeste Weg gewählt. Dieser Knoten wird so lange ausgeführt, bis keine Wege zwischen den definierten Farben mehr erstellt werden können.

#### Convolutionnode

Ein Convolutionnode besitzt wie ein Rulenode mindestens eine Ersetzungsregel. Die Regel gilt aber lediglich für eine Kachel, denn Regeln im Convolutionnode werden abhängig von den Nachbarn der Eingabekachel angewendet. Nur wenn die Nachbarn der Eingabekachel bestimmte Farben haben, wird diese Kachel durch die Ausgabekachel ersetzt. Man kann zwischen der Von-Neumann- und Moore-Nachbarschaft wählen. Die Nachbarfelder und deren Farben werden einmal zu Beginn ermittelt. Dann wird jede Regel nacheinander anhand dieser Nachbarschaften angewendet. Spätere Regeln können daher Vorherige überschreiben.

#### ConvChain Node

Ein ConvChain Node nutzt den ConvChain Algorithmus [10] von Maxim Gumin. Dieser ermöglicht anhand eines Beispielbildes, Bilder zu generieren, welche die gleiche Verteilung an $N \times N$ Mustern des Beispielbildes haben.

Der ConvChain Node wendet den ConvChain Algorithmus dann auf einen definierten Teil des Rasters an.

## 4.3 Programmaufbau

Ein MJ Programm besteht aus mindestens einem Knoten. Damit mehrere Knoten ausgeführt werden können, muss ein Branchnode als Wurzelknoten benutzt werden. Standardmäßig ist dies ein Sequencenode, damit man eine sequenzielle Ausführung ermöglichen kann.

Auch das Raster selbst hat Parameter, die zu Beginn eines Programms (oder in einem manipulierenden Branchnode) angegeben werden müssen. Notwendig sind dabei die Farben, die das Raster annehmen kann. Dazu legt die erste genannte Farbe die Grundfarbe des Rasters zum Programmstart fest. Zusätzlich gibt es die Option, zu bestimmen, ob es einen Ursprung geben soll. Der Ursprung ist die Kachel in der Mitte des Rasters. Ist diese Option gesetzt, so bestimmt die zweite gegebene Farbe die Farbe des Ursprungs. Diese Parameter werden im Wurzelknoten des Programms gesetzt.
So kann der Beginn eines MJ Programms folgendermaßen aussehen:

---

**Sequencenode**
*Rasterparameter* Farben: (Schwarz, Weiß, Gelb), Ursprung: ja
**Kindknoten**
..... .....
**Kindknoten**
..... .....

---

Das Raster kann somit die Farben Schwarz, Weiß und Gelb annehmen. Die Grundfarbe des Rasters zu Beginn ist schwarz. Zusätzlich ist in der Mitte des Rasters eine Kachel weiß gefärbt.

Außerdem gibt es allgemeinere Modellparameter für das Programm, welche außerhalb des Programmcodes definiert werden. Der notwendige Parameter in diesem Fall ist die Größe des Rasters oder die einzelnen Teilgrößen des Rasters (Länge, Breite, Tiefe). Das Raster kann somit ein frei wählbares Rechteck oder ein frei wählbarer Quader sein. Optionale Parameter sind unter anderem die maximale Anzahl an Schritten für die Ausführung oder auch die Dimension des Rasters.

## 4.4 Beispielanwendung

Ein simples MJ Programm ist *BasicSnake* aus dem Originalprojekt von Maxim Gumin [12]. Dieses Programm simuliert ein zufälliges, simples Snake-Spiel.
Im originalen Snake-Spiel bewegt sich eine Schlange gerade oder rechtwinklig auf einem Raster. Das Ziel ist es währenddessen Futter auf dem Spielfeld aufzusammeln, wodurch die Schlange wächst. Zusätzlich muss man Hindernissen und dem eigenen Schlangenkörper ausweichen.[1]

### 4.4.1 Programm

In abstrakter Darstellung und leicht abgewandelt sieht das MJ Programm wie folgt aus:

---

*Allgemeine Parameter* Größe: 19, Dimension: 2

**Sequencenode**
*Rasterparameter* Farben: (Schwarz, Orange, Grau, Lila, Grün, Rot), Ursprung: ja
**Allnode**
*Regel* Eingabemuster: (Orange, Schwarz, Schwarz) - Ausgabemuster: (*, *, Grau)
*Regel* Eingabemuster: (Grau, Schwarz, Schwarz) - Ausgabemuster: (*, *, Grau)
**Onenode**
*Regel* Eingabemuster: (Orange, Schwarz, Grau) - Ausgabemuster: (Lila, Grün, Rot)
**Onenode**
*Parameter* Schritte: 2
*Regel* Eingabemuster: (Rot, Schwarz, Grau) - Ausgabemuster: (Grün, Grün, Rot)
**Onenode**
*Parameter* Schritte: 10
*Regel* Eingabemuster: (Grau) - Ausgabemuster: (Orange)
**Markovnode**
**Onenode**
*Regel* Eingabemuster: (Rot, Schwarz, Orange) - Ausgabemuster: (Grün, Grün, Rot)
**Allnode**
*Regel* Eingabemuster: (Rot, Schwarz, Grau) - Ausgabemuster: (Grün, Grün, Rot)
*Regel* Eingabemuster: (Lila, Grün, Grün) - Ausgabemuster: (Grau, Schwarz, Lila)

---

[1] https://de.wikipedia.org/wiki/Snake_(Computerspiel) - Zugriffsdatum: 20.07.2024

### 4.4.2 Ausgangssituation

Das Raster hat eine Größe von $19 \times 19$ zu Beginn der Ausführung, welche durch die allgemeinen Parameter festgelegt wurde.

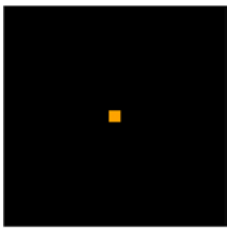


Abbildung 4.6: Grundraster für das Snake-Spiel

Dazu ist es schwarz und hat eine orangene Kachel in der Mitte vorliegen (siehe Abbildung 4.6). Dies wurde durch die Rasterparameter im Wurzelknoten festgelegt. Zudem kann das Raster die Farben Grau, Lila, Grün und Rot annehmen.

### 4.4.3 Ablauf

Der Wurzelknoten des Programms ist ein Sequencenode. Daher werden alle weiteren Knoten sequenziell ausgeführt.

Der erste Knoten, welcher ausgeführt wird, ist ein Allnode. Dieser beinhaltet zwei Ersetzungsregeln. Das Raster wird nach allen Übereinstimmungen beider Eingabemuster der Regeln durchsucht. Zu Beginn stimmt die erste Regel in der Mitte des Rasters an vier Stellen überein. Da es sich um einen Allnode handelt, werden alle Übereinstimmungen, solange diese sich nicht überschneiden, angewendet. Aufgrund der Wildcards (*) im Ausgabemuster wird lediglich eine schwarze durch eine graue Kachel ersetzt, während die orangene und benachbarte schwarze Kachel unberührt bleiben.

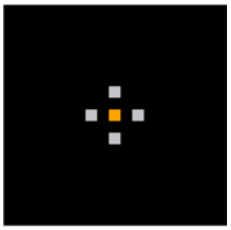


Abbildung 4.7: Raster nach Anwendung der ersten Regel des ersten Allnodes
Eingabemuster: (Orange, Schwarz, Schwarz) - Ausgabemuster: (*, *, Grau)
Eingabemuster: (Grau, Schwarz, Schwarz) - Ausgabemuster: (*, *, Grau)

Das Ersetzen der schwarzen Kachel durch die graue Kachel überschneidet sich nicht für die vier Übereinstimmungen, weshalb alle vier wie in Abbildung 4.7 angewendet werden.

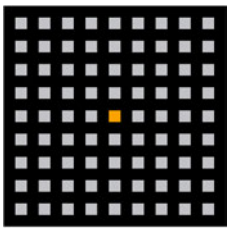


Abbildung 4.8: Raster nach voller Ausführung des ersten Allnodes

Im nächsten Schritt stimmt nur noch die zweite Regel überein, sodass diese im selben Prinzip wie im ersten Schritt angewendet wird. Dies wiederholt sich in den nächsten Schritten, bis auch die zweite Regel nicht mehr übereinstimmt (siehe Abbildung 4.8). Damit wurde die Grundlage des Spielfeldes aufgebaut, bei dem die grauen Kacheln die begehbaren Felder darstellen. Nun wird der nächste Knoten ausgeführt.

Dieser ist ein Onenode mit einer Ersetzungsregel. Es wird wieder nach allen Übereinstimmungen des Eingabemusters der Regel gesucht. Wie zuvor stimmt diese Regel an vier Stellen in der Mitte des Rasters überein.

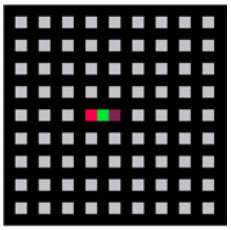
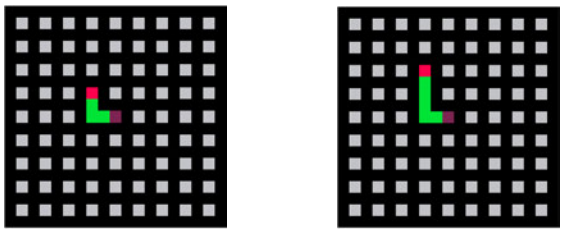


Abbildung 4.9: Raster nach voller Ausführung des ersten Onenodes
Eingabemuster: (Orange, Schwarz, Grau) - Ausgabemuster: (Lila, Grün, Rot)

Da es sich um einen Onenode handelt, wird wie in Abbildung 4.9 zufällig eine Übereinstimmung angewendet. Es wurde die Grundlage für die Schlange erstellt, die einen roten Kopf, einen grünen Körper und ein lilafarbenes Ende hat. Danach stimmt die Regel nicht mehr überein und der nächste Knoten ist an der Reihe.

Dies ist wieder ein Onenode, welcher nun auf zwei Schritte beschränkt ist. Das Eingabemuster stimmt am Kopf der Schlange an drei Stellen überein.



(a) Erste Ausführung

(b) Zweite Ausführung

Abbildung 4.10: Raster nach schrittweiser Ausführung des zweiten Onenodes
Eingabemuster: (Rot, Schwarz, Grau) - Ausgabemuster: (Grün, Grün, Rot)

Zufällig wird eine davon ausgesucht. Aufgrund der Beschränkung der Schritte passiert dies zweimal (siehe Abbildung 4.10). Dies gibt der Schlange eine Grundlänge für den Anfang. Danach ist der nächste Knoten dran.

Auch dies ist wieder ein Ondenode, der auf zehn Schritte beschränkt ist. Das Eingabemuster stimmt bei jeder grauen Kachel überein.

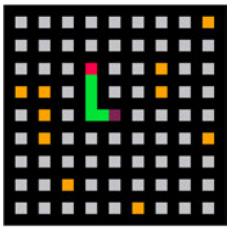


Abbildung 4.11: Raster nach voller Ausführung des dritten Onenodes
Eingabemuster: (Grau) - Ausgabemuster: (Orange)

Zehn von diesen Kacheln werden zufällig durch orangene Kacheln ersetzt (siehe Abbildung 4.11). Diese stellen das Futter des Snake-Spieles dar. Nach der Ausführung der bisherigen Knoten ist nun der Startzustand für das Snake-Spiel generiert worden. Damit ist der nächste Knoten an der Reihe.

Dieser Knoten ist ein Branchnode, genauer gesagt ein Markovnode, welcher zwei Kindknoten beinhaltet. Der Onenode hat eine Regel. Das Eingabemuster dieser Regel beschreibt, ob der Schlangenkopf neben einem Futterfeld ist. Das Ausgabemuster sorgt dafür, dass die Schlange das Futterfeld betritt. Da der Schlangenschwanz unberührt bleibt, wächst die Schlange somit beim Betreten eines Futterfeldes. Der Allnode hat zwei Regeln. Die beiden Regeln zusammen beschreiben die Vorwärtsbewegung der Schlange. Dabei ist die erste Regel für den Kopf der Schlange zuständig. Das Eingabemuster stimmt immer dann überein, wenn vor der Schlange noch ein freies Feld ist. Das Ausgabemuster bewegt dann den Kopf entsprechend nach vorne. Die zweite Regel ist für das Ende der Schlange zuständig. Das Eingabemuster stimmt mit dem schon gegangenen Weg der Schlange vom Ende aus gesehen überein. Das Ausgabemuster verkürzt somit den Schlangenschwanz in Richtung gegangener Strecke.
Innerhalb des Markovnodes wird der erste Knoten priorisiert ausgeführt. Dieser ist der Onenode. Das Eingabemuster des Onenodes hat zu Beginn keine Übereinstimmungen.

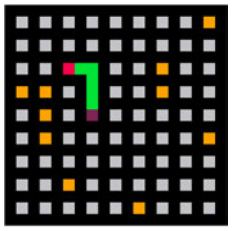


Abbildung 4.12: Raster nach der ersten Ausführung des Allnodes innerhalb des Markovnodes
Eingabemuster: (Rot, Schwarz, Grau) - Ausgabemuster: (Grün, Grün, Rot)
Eingabemuster: (Lila, Grün, Grün) - Ausgabemuster: (Grau, Schwarz, Lila)

Daher wird der nächste Knoten ausgeführt. Dies ist der Allnode. Beide Eingabemuster stimmen überein und überschneiden sich nicht. Daher werden beide angewendet und die Schlange bewegt sich somit in eine zufällige freie Richtung nach vorne (siehe Abbildung 4.12).

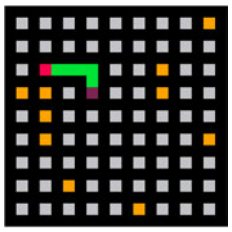


Abbildung 4.13: Raster nach der zweiten Ausführung des Allnodes innerhalb des *Markovnodes*

Nach dieser einen Ausführung des Allnodes wird wieder der erste Knoten im Markovnode angeschaut, um zu prüfen, ob dieser nun ausgeführt werden kann. Dies ist weiterhin nicht der Fall, weshalb wieder der Allnode einmal ausgeführt wird und die Schlange sich nach vorne bewegt (siehe Abbildung 4.13).

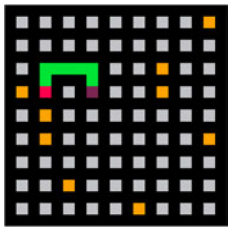
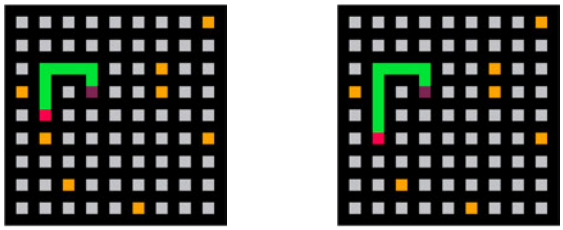


Abbildung 4.14: Raster nach der ersten Ausführung des Onenodes innerhalb des Markovnodes
Eingabemuster: (Rot, Schwarz, Orange) - Ausgabemuster: (Grün, Grün, Rot)

Nun wird der Onenode erneut geprüft und dieses Mal stimmt das Eingabemuster der Regel überein, sodass die Regel angewendet wird. Die Schlange hat ein Futterfeld betreten und ist gewachsen (siehe Abbildung 4.14).



(a) Zweite Ausführung

(b) Dritte Ausführung

Abbildung 4.15: Raster nach schrittweiser Ausführung des Onenodes im Markovnode

Der Onenode stimmt nach dieser Ausführung weiterhin überein und wird solange ausgeführt, bis dieser nicht mehr übereinstimmt (siehe Abbildung 4.15).

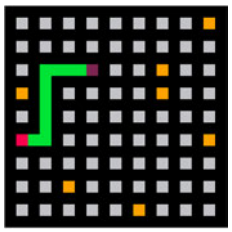


Abbildung 4.16: Raster nach der dritten Ausführung des Allnodes innerhalb des Markovnodes
Eingabemuster: (Rot, Schwarz, Grau) - Ausgabemuster: (Grün, Grün, Rot)
Eingabemuster: (Lila, Grün, Grün) - Ausgabemuster: (Grau, Schwarz, Lila)

Nun wird wieder der Allnode ausgeführt und die Schlange bewegt sich, wie in Abbildung 4.16 zu sehen ist, nach vorne. Danach wird wieder der Onenode geprüft und dieser Ablauf

wiederholt sich, bis keine Regeln mehr übereinstimmen oder die maximale Anzahl an Schritten für das Programm erreicht ist.

# 5 Konzept

In diesem Kapitel werden die grundlegenden Ideen zur Umsetzung der Ziele erklärt. Dafür wird vorerst ermittelt, was die Anforderungen an die generierten Labyrinthe sind. Darauffolgend wird vorbereitend für den Ansatz Markov Junior (MJ) in die prozedurale Generierung eingeordnet und ein Generierungstyp festgelegt.
Der Ansatz beginnt mit der Generierung des Lösungsweges. Aufbauend darauf werden einzelne Abzweigungspunkte vom Lösungsweg generiert. Diese dienen als Grundlage, um im nächsten Schritt alle restlichen Wege zu generieren. Zum Schluss wird das Labyrinth im Design angepasst, um so den Anforderungen gerecht zu werden.
Im Folgenden werden diese Schritte näher erläutert.

## 5.1 Anforderungen

Für die spätere Evaluation, aber auch für eine erfolgreiche Umsetzung der Ziele ist eine Sammlung von Anforderungen an das Endprodukt wichtig. Im Folgenden werden diese aufgezählt.

### 5.1.1 Eigenschaften der Labyrinthe

Jedes Labyrinth hat Eigenschaften. Die angeforderten Grundeigenschaften der in dieser Arbeit generierten Labyrinthe sind folgende:

Jedes generierte Labyrinth ist...

- ... ein Gamma Labyrinth.
- ... zweidimensional.
- ... perfekt.
- ... einheitlich. Das bedeutet, dass es keine Wege gibt, die breiter/größer sind als andere Wege. Dasselbe gilt für die Wände.

- ... klar lösbar. Das bedeutet, dass es einen Startpunkt und ein Ziel im Labyrinth gibt, welche durch einen Weg verbunden sind.
  - Der Start und das Ziel sind gegenüber voneinander an der oberen und unteren Seite des Labyrinthes. Es wird von unten gestartet.

### 5.1.2 Aussehen der Labyrinthe

Ein Labyrinth kann unterschiedlich gestaltet werden. Die Anforderungen an das Aussehen der Labyrinthe lauten wie folgt:

- Die Wege sind weiß und breit, während die Wände schmal und schwarz sein sollen (siehe Abbildung 3.1b). Dies soll die Unterscheidung der Wege und Wände vereinfachen.
- Der Start ist grün markiert und das Ziel ist rot markiert. So soll direkt klar sein, wo der Start und das Ziel sind.

### 5.1.3 Kontrollierbarkeit

Die Generierung der Labyrinthe soll im Allgemeinen gut kontrollierbar sein. Sie soll zulassen, dass gewisse Eigenschaften über Parameter gesteuert werden können. Dabei soll der Fokus auf dem Lösungsweg und seinen Abzweigungen liegen, da diese eine wesentliche Rolle in einem Labyrinth spielen.
Der Lösungsweg soll in seinem Verlauf zwischen Start und Ziel kontrollierbar sein. Dies bedeutet, dass gesteuert werden kann, wie willkürlich der Lösungsweg zwischen Start und Ziel verläuft. Bei höchster Einstellung sollte der Weg zufällig verlaufen. Bei niedrigster Einstellung sollte der Weg in einer Gerade zwischen Start und Ziel verlaufen. Da dies auf Zufälligkeit basieren wird, soll bei den generierten Labyrinthen im Durchschnitt gewährleistet sein, dass unterschiedliche Einstellungen der Willkürlichkeit auch das jeweilige erwartete Ergebnis erzeugen.
Dazu soll es möglich sein, festzulegen, wie viele direkte Abzweigungen von dem Lösungsweg abgehen. Jedes generierte Labyrinth soll die gegebene Anzahl an direkten Abzweigungen aufweisen.
Außerdem soll die Größe des Labyrinthes kontrollierbar sein.

## 5.2 Markov Junior und prozedurale Generierung

MJ ist in der Lage, zwei- und dreidimensionale Rastergrafiken zu erzeugen. Durch den Nichtdeterminismus von MJ erzeugt eine gleiche Eingabe (Grundraster) unterschiedliche Ausgaben. Dabei unterliegen die Ausgaben immer den definierten Regeln. Die Zufälligkeit ist direkt an die Regeln gebunden, weshalb sie immer gewissen Bedingungen genügt. Dies zeigt, dass MJ für die prozedurale Generierung verwendet werden kann. In Kapitel 4.4 ist bspw. zu sehen, wie MJ den Startzustand eines Snake-Spiels generiert (siehe Abbildung 4.11).

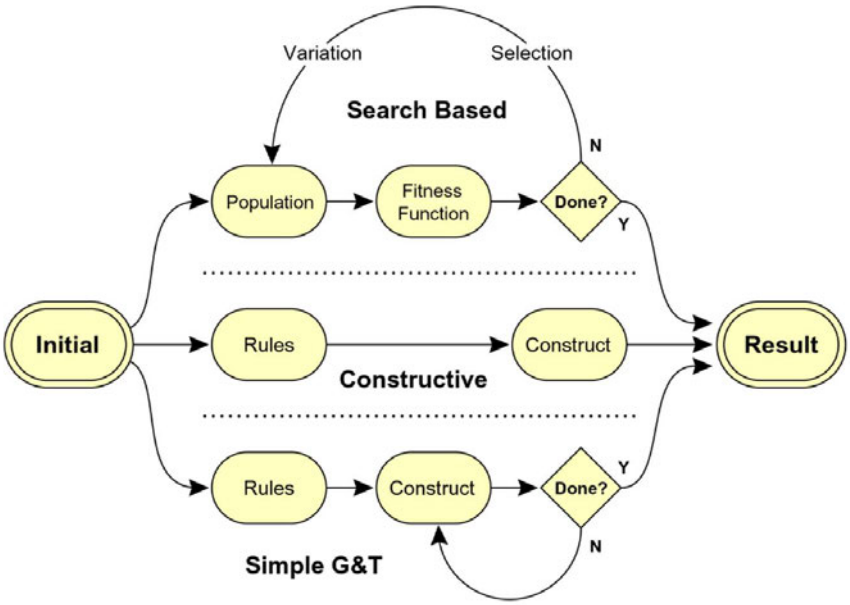



Abbildung 5.1: Drei typische Ansätze zur prozeduralen Generierung [27] um zu verdeutlichen, wo MJ einzuordnen ist

Im Bereich der prozeduralen Generierung gibt es unterschiedliche Vorgehensweisen. Typische Ansätze sind in Abbildung 5.1 zu sehen. MJ kann in den konstruktiven Ansatz eingeordnet werden. Der konstruktive Ansatz definiert ein Regelset und generiert den Inhalt einmal. Anders als bei den anderen beiden Ansätzen ist der Inhalt nach der ersten Generierung fertig und wird nicht mehr verändert. Es muss daher sichergestellt werden, dass der Inhalt während der Generierung korrekt ist. Dies kann durch die Anwendung von Operationen erzielt werden, welche garantieren können, dass kein falscher Inhalt erzeugt wird [27].

Für die Generierung der Labyrinthe mit MJ muss daher ein Regelset gefunden werden, was eine direkte, korrekte Generierung ermöglicht.

## 5.3 Art der Labyrinthgenerierung

Die grundlegende Struktur eines Labyrinthes sind dessen Wege. Sie machen das Labyrinth aus und im Endeffekt geht es um den Lösungsweg, welchen man versucht herauszufinden. Um eine hohe Kontrollierbarkeit bei der Generierung eines Labyrinthes zu erreichen, bietet es sich an, dabei die Wege zu generieren. Zusätzlich ist gefordert, Kontrolle über den Lösungsweg zu haben, was dadurch ebenfalls gut umgesetzt werden kann. Daher ist der *Fokus* (siehe Kapitel 3.3.1) des Labyrinthes ein Weg schnitzender Ansatz. Im Vergleich müssten bei einem Wand hinzufügenden Ansatz immer zwei Wände generiert werden, um einen Weg des Labyrinthes zu erzeugen. Daraus ergibt sich eine höhere Komplexität, wenn es darum geht, die Wege kontrolliert zu generieren. Dabei ist zu beachten, dass eine spezifische Art an Weg schnitzenden Ansätzen verwendet werden muss. Diese sind diejenigen, welche auch wirklich vollständige Wege generieren. Denn es gibt auch Weg schnitzende Ansätze, bei denen an zufälligen Stellen im Labyrinth Teile von Wegen generiert werden, welche dann Stück für Stück zusammengeführt werden. Dies würde wiederum eine hohe Kontrollierbarkeit der Weggenerierung erschweren.

## 5.4 Grundkonzept der Labyrinth Generierung

Im Mittelpunkt der Generierung steht die Kontrollierbarkeit der Labyrinthe. Dafür und auch generell für eine möglichst gute Kontrollierbarkeit, aber auch für die Möglichkeit der Erweiterung (für z. B. die Schwierigkeit) bietet es sich an, einen modularen Ansatz zu wählen. Bei diesem werden einzelne Komponenten des Labyrinthes unabhängig voneinander generiert.

Ein Labyrinth kann dafür in zwei Hauptteile unterteilt werden: den Lösungsweg und dessen Abzweigungen. Die Abzweigungen gehören zum Lösungsweg und füllen das restliche Labyrinth. Deswegen wird zuerst der Lösungsweg generiert und davon die Abzweigungen, um das Labyrinth zu vervollständigen.

Als Grundlage dafür muss ein Ansatz für die Generierung der Wege (Lösungsweg und Abzweigungen) entwickelt werden. Dabei muss beachtet werden, dass die Wege nicht in sich selbst laufen, da das Labyrinth sonst nicht perfekt ist. Zusätzlich soll Zufälligkeit

bei dem Ansatz gewährleistet sein. Für Inspirationen werden dafür schon existierende Algorithmen angeschaut.

### 5.4.1 Weggenerierung

Unter den bekanntesten Algorithmen zur Generierung perfekter Labyrinthe (Kapitel 3.3.2) ist der rekursive Backtracker ein Weg schnitzender Algorithmus, welcher die Wege direkt generiert. Dieser kann entweder rekursiv umgesetzt werden oder mithilfe eines Stacks. Der Algorithmus mit einem Stack [9] kann wie folgt beschrieben werden:
Angenommen, es existiert ein Graph über einem Raster, bei dem alle Knoten miteinander über ihre direkten Nachbarn verbunden sind (vergleichbar zur Abbildung 3.2a). Dann lautet der Algorithmus:

1. Wähle einen zufälligen Startknoten, markiere ihn als besucht und packe ihn auf den Stack.
2. Hole den obersten Knoten $v$ aus dem Stack heraus. Ist der Stack leer, beende den Algorithmus.
3. Wähle einen unbesuchten Nachbarn $u$ von $v$ aus.
   a) Wenn es $u$ gibt, packe $v$ auf den Stack.
   b) Ansonsten springe zu Schritt 2.
4. Verbinde Knoten $u$ mit $v$.
5. Markiere $u$ als besucht und packe ihn auf den Stack.
6. Wiederhole Schritte 2-5.

Im Endeffekt generiert dieser Algorithmus immer wieder zufällige, nicht in sich laufende Wege. Diese werden auch self-avoiding walk (SAW)[1] genannt. Ein perfektes Labyrinth und dessen Spannbaum ist also nichts anderes als eine Aneinanderreihung mehrerer SAWs. Dies wird in dieser Arbeit genutzt, um daraus den Spannbaum für die Labyrinthe in einem modularen Ansatz generieren zu können.

[1] https://en.wikipedia.org/wiki/Self-avoiding_walk - Zugriffsdatum: 20.07.2024

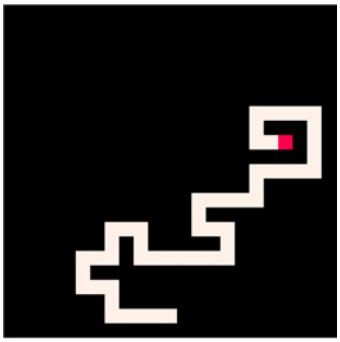


Abbildung 5.2: Beispiel eines SAWs auf einem 2D-Raster (Rot stellt den aktuellen besuchten Punkt dar)

In Abbildung 5.2 ist ein SAW dargestellt. Dieser begeht alle ungeraden Punkte, damit so zwischen dem Weg ein Freiraum ist. In Bezug auf Labyrinthe wären das die Wände. So kann in einem zweidimensionalen Raster direkt eine Labyrinth-Struktur erzeugt werden. Dieses Konzept bildet die Grundlage für die Generierung der Labyrinthe in dieser Arbeit.

## 5.5 Lösungsweg

Die Grundlage des Lösungsweges sind der Start und das Ziel. Nach den Anforderungen sind diese immer an den gleichen Punkten. Es ist daher nur festzulegen, wo die Mitte des Rasters ist, um an dieser Stelle oben und unten am Rand des Rasters zwei Punkte zu generieren. Vor der Generierung des Lösungsweges werden Start und Ziel für das Labyrinth ermittelt und generiert.

Für den Weg ist es nötig, dass dieser vom Start zum Ziel verläuft. Eine Idee ist es, einen SAW zwischen den beiden Punkten zu generieren. Der SAW würde dafür beim Start beginnen.

Notwendig dafür ist jedoch eine Methode, damit der SAW auch das Ziel erreicht. Dies kann umgesetzt werden, indem der SAW immer denjenigen nächsten Punkt auswählt, welcher dem Ziel näher ist. Würde der SAW immer den Punkt, der dem Ziel am nächsten ist, nehmen, würde der generierte Weg lediglich eine gerade Linie zwischen den zwei Punkten sein. Um dies zu vermeiden, wird eine Temperatur eingeführt, welche dafür sorgt, dass nur zu einer gewissen Wahrscheinlichkeit der beste Knoten gewählt wird. Dadurch lässt sich steuern, wie willkürlich der Lösungsweg verlaufen soll, wodurch man, wie in den Anforderungen gegeben, Kontrolle über diesen hat.

Nach diesem Vorgehen ergibt sich ein weiteres Problem. Bei einem SAW kann es passieren, dass dieser in einen Zustand gerät, bei dem alle Nachbarn schon besucht sind. Im rekursiven Backtracker Algorithmus wird dann der schon gegangene Weg zurückverfolgt, um aus dieser Sackgasse herauszukommen. Dies kann auch für diese Weggenerierung verwendet werden. Die Generierung des Lösungsweges ist somit wie ein rekursiver Backtracker Algorithmus mit einer bestimmten Richtung.

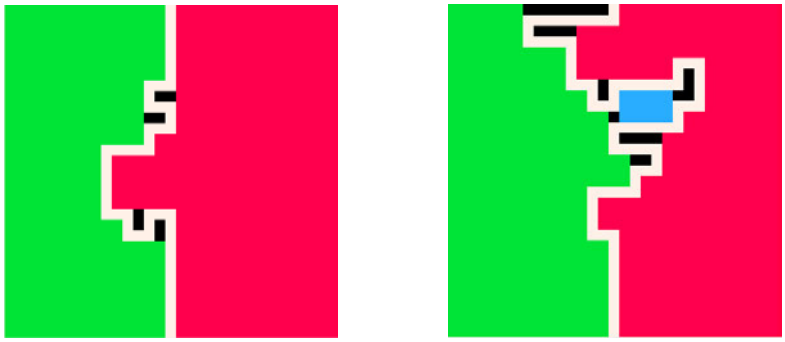
Da die zurückverfolgten Wege direkte Abzweigungen des Lösungsweges wären, werden diese am Ende der Weggenerierung entfernt, um so nur den Lösungsweg zu haben.

Daraus ergibt sich folgender Algorithmus zur Generierung des Lösungsweges:

1. Generiere Start und Ziel gegenüber voneinander.
2. Führe einen rekursiven Backtracker Algorithmus mit bestimmter Richtung aus (vom Start Richtung Ziel).
   a) Führe einen SAW aus, welcher seinen nächsten Punkt abhängig von der Distanz zum Ziel wählt. Eine Temperatur steuert, mit welcher Wahrscheinlichkeit der zum Ziel nächste Punkt gewählt wird. Landet der SAW im Ziel, springe zu Schritt 3. Landet der SAW in einer Sackgasse, fahre mit dem nächsten Schritt fort.
   b) Verfolge den gegangenen Weg zu einer freien Stelle zurück und wiederhole Schritt 2a.
3. Entferne alle zurückverfolgten Wege.

## 5.6 Abzweigungen

Für die Abzweigungen ist der Lösungsweg die Grundlage. Von diesem werden zuerst Punkte markiert, an denen es Abzweigungen geben wird. Die Anzahl entspricht der gewollten Menge an direkten Abzweigungen. Dabei kann es aber zu Problemen kommen, wenn der Lösungsweg einen bestimmten Verlauf hat.



(a) Lösungsweg mit zwei Teilbereichen

(b) Lösungsweg mit drei Teilbereichen

Abbildung 5.3: Beispiel zweier Lösungswege, die das Raster in unterschiedliche Teilbereiche aufteilen

Der Start und das Ziel sind gegenüber voneinander, weshalb der Lösungsweg das Raster im einfachen Fall in zwei Teile aufteilt (siehe Abbildung 5.3a). Dabei ist der Lösungsweg weiß und die farblich markierten Stellen sind die Teilbereiche. Die schwarzen Stellen sind Zwischenräume des Weges, da dieser nur alle ungeraden Kacheln begeht. Diese würden die Wände darstellen und gehören daher zu keinem Teilbereich. Da der Weg jedoch zufällig verläuft, kann er dadurch Teilbereiche von dem Rest des Rasters abgrenzen, sodass mehr als zwei abgegrenzte Bereiche im Labyrinth entstehen, wie es in Abbildung 5.3b der Fall ist.
Wählt man nun zufällig eine feste Anzahl an Abzweigungspunkten vom Lösungsweg, kann es passieren, dass ein Teilbereich keinen Abzweigungspunkt kriegt. Dadurch würde am Ende der Generierung ein Bereich übrig bleiben, welcher keine Wege hat. Dies soll vermieden werden. Dafür werden zuerst alle Teilbereiche identifiziert. Für jeden dieser Teilbereiche wird ein Abzweigungspunkt generiert. So ist sichergestellt, dass das Labyrinth am Ende gefüllt ist. Danach werden die restlichen Abzweigungspunkte generiert, um so auf die gewollte Anzahl an Abzweigungen zu kommen.
Es kann somit geschehen, dass am Ende mehr direkte Abzweigungen vom Lösungsweg existieren, als es gewollt ist (bei sehr vielen Teilbereichen). Dies ist aber nicht zu vermeiden, da das Labyrinth vollständig sein soll. Dieser Ansatz gewährt somit, dass mindestens die gewollte Anzahl an direkten Abzweigungen und maximal die notwendige Anzahl an direkten Abzweigungen vom Lösungsweg existiert.

Nachdem die Abzweigungspunkte generiert wurden, können die Wege der Abzweigungen generiert werden. Anders als bei dem Lösungsweg müssen die Abzweigungen nicht in eine bestimmte Richtung generiert werden. Daher kann ein normaler SAW verwendet werden.

Für jeden Abzweigungspunkt wird dieser ausgeführt. Dabei ist zu beachten, dass die unterschiedlichen SAWs nicht ineinanderlaufen. Dafür zählen die schon besuchten Punkte der jeweiligen SAWs für alle SAWs. Laufen die einzelnen SAW in eine Sackgasse, sind alle Grundwege der direkten Abzweigungen generiert, sodass man zum nächsten Schritt fortfahren kann.
Dieser ist die Generierung aller restlichen Wege, also die Abzweigungen der Abzweigungen etc. Dazu werden nun wiederholend an zufälligen Stellen der schon existierenden Abzweigungen neue Abzweigungen generiert. Dies geschieht nach dem gleichen Prinzip, wie die ursprünglichen Abzweigungen generiert werden, bis das gesamte Labyrinth gefüllt ist. Dieses Vorgehen der Generierung eines Labyrinthes ist von Bellot et al. [2] als *Prim & Kill* Algorithmus vorgestellt worden. Sie haben dafür den Prim Algorithmus und den Hunt & Kill Algorithmus kombiniert. Diese sind zwei bekannte graphbasierte Generierungsalgorithmen für Labyrinthe.

Daraus ergibt sich folgender Algorithmus zur Generierung der Abzweigungen:

1. Identifiziere alle Teilbereiche des Labyrinthes.
2. Generiere einen zufälligen Abzweigungspunkt für jeden dieser Teilbereiche.
3. Generiere zufällig alle restlichen Abzweigungspunkte, um auf die gewollte Anzahl zu kommen.
4. Führe für alle Abzweigungspunkte einen SAW aus.
5. Wähle einen zufälligen Punkt von den bestehenden Abzweigungen und führe von dort einen SAW aus.
6. Wiederhole Schritt 5, bis das Labyrinth gefüllt ist.

## 5.7 Aussehen

Wie in Kapitel 5.4.1 vorgestellt wurde, kann mithilfe eines SAWs eine Labyrinth-Struktur generiert werden, bei der die Wege und Wände jeweils eine Kachel breit sind. Das fertig generierte Labyrinth würde dann jedoch nicht den Anforderungen für das Aussehen entsprechen. Für den Prozess der Generierung bietet sich dieses Verfahren aber an und eine Abwandlung dessen würde zu einer höheren Komplexität führen. Daher wird zuerst die normale Generierung ausgeführt, bei der das Labyrinth gleich breite Wege und Wände hat, und zum Schluss wird das Aussehen angepasst.

# 6 Umsetzung

In diesem Kapitel wird die konkrete Umsetzung des erstellten Konzeptes erklärt und dargestellt.

## 6.1 Benutzte Technologie

Die Originalimplementierung zu Markov Junior (MJ) von Maxim Gumin [12] ist in C# geschrieben. Diese Arbeit baut auf einer Java Implementierung von Niclas zum Felde [7] der Originalimplementierung auf. Die Java Implementierung bietet alle Features der Originalimplementierung an und unterscheidet sich nur leicht in der verwendeten Syntax.

## 6.2 Syntax von Markov Junior

Sowohl die Originalimplementierung als auch die Java Implementierung haben XML als Umsetzung für die Syntax von MJ verwendet. Im Folgenden wird diese dargestellt. Im Fokus liegt dabei die Syntax der Komponenten, welche für die Umsetzung in dieser Arbeit benutzt worden sind.

### 6.2.1 Alphabet von Markov Junior

Das Alphabet ist aus den Farben der Kacheln, Unions und Wildcards aufgebaut, welche alle eine spezifische Syntax haben.

#### Farben

Die Farben werden alle durch einen Buchstaben dargestellt. Die Zuordnung von Buchstabe zu Farbe wird in einer externen Datei konfiguriert. Somit würde ein Muster der Form (Weiß, Schwarz, Gelb) zu *"WBY"* werden. Dabei steht das $W$ für *white* (Weiß), das $B$ für *black* (Schwarz) und das $Y$ für *yellow* (Gelb).

### Union

Unions werden als eigenes XML-Element dargestellt und müssen vor der Regel, welche diese verwendet, erstellt werden:

```
<union symbol="?" values="BW"/>
```

### Wildcard

Wildcards werden in MJ durch einen Stern (*) dargestellt.

### Mehrdimensionale Regeln

Da MJ auf mehrdimensionalen Rastern arbeitet, können auch mehrdimensionale Regeln erstellt werden. Im zweidimensionalen Raum wird dafür ein weiteres Zeichen eingeführt, welches nicht zum Alphabet gehört. Dieses ist ein Slash (/) und kennzeichnet einen Umsprung in die nächste Zeile. Ein Muster der Form $\binom{\text{Weiß}}{\text{Schwarz}}\ \binom{\text{Weiß}}{\text{Schwarz}}$ würde zu *"WW/BB"* werden.

## 6.2.2 Knoten

Alle Knoten werden als eigene XML-Elemente dargestellt und jeder Parameter ist ein Attribut dieses Elementes.

### Rulenodes

Alle Rulenodes haben bis auf den Namen des XML-Elementes dieselbe Syntax:

```
<rule-node-name in="WB" out="WW"/>
```

Ein Onenode heißt *one*, ein Allnode heißt *all* und ein Parallelnode heißt *prl*. Sollen mehrere Regeln im Rulenode angegeben werden, werden diese als eigene XML-Elemente innerhalb des Rulenodes angegeben. Als Beispiel mit einem Onenode würde dies folgendermaßen aussehen:

```
<one>
  <rule in="WB" out="WW"/>
  <rule in="WBB" out="WBW"/>
</one>
```

Ein Field wird ebenfalls innerhalb des Rulenodes angegeben:

```
<one in="WB" out="WW">
  <field for="W" to="Y" on="B"/>
</one>
```

#### Branchnodes

Ein Sequencenode sieht wie folgt aus:

```
<sequence>
  ....
</sequence>
```

Ein Markovnode sieht wie folgt aus:

```
<markov>
  ....
</markov>
```

Ein Mapnode kann folgendermaßen aussehen:

```
<map scale="2 2 1" values="BW">
  <rule in="Y" out="WW/WW"/>

  ....
</map>
```

### 6.2.3 Programmparameter

Auch die Programmparameter haben ihre eigene Syntax, welche im Folgenden dargestellt wird.

#### Allgemeine Parameter

Die allgemeinen Modellparameter werden in einem eigenen XML-Element dargestellt, bei dem jeder Parameter als XML-Attribut definiert wird:

```
<model size="30" steps="1000" d="2">
</model>
```

Dies definiert ein Raster mit einer Größe von $30 \times 30$ und es werden maximal 1000 Schritte ausgeführt.

In diesem Fall unterscheidet sich die Syntax eines MJ Programms der Java-Implementierung von der Originalimplementierung. In der Originalimplementierung werden die Modellparameter in einer externen Datei definiert, während die Java-Implementierung diese als Wurzelknoten definiert, in dem dann der eigentliche Programmcode ist.

#### Rasterparameter

Die Rasterparameter werden in dem ersten Wurzelknoten des Programmcodes als Attribute definiert. So kann der Programmbeginn mit einem Sequencenode wie folgt aussehen:

```
<sequence values="BWY" origin="true">
  ....
  ....
</sequence>
```

## 6.3 Grundlage für die Umsetzung

Für die Generierung sind vorerst Grundlagen zu klären, wie zum Beispiel die Parameter des Programms, aber auch die Farben, die verwendet werden.

### 6.3.1 Größe des Rasters

Der einzig relevante Modellparameter, der hier zu beachten ist, ist die Größe des Rasters. Ein Labyrinth besitzt eine Außenwand, welche die äußersten Wege begrenzt. Bei dem in Abbildung 5.2 dargestellten Konzept für die Labyrinthgenerierung bedeutet dies, dass es einen Rahmen mit einer Breite von einer Kachel geben muss. Direkt angrenzend dazu verlaufen die Wege, welche von einem Startpunkt aus immer zwei Schritte nach vorne gehen, um so einen Freiraum für eine Wand dazulassen. Daher muss die Strecke innerhalb des Rahmens ungerade sein. Zusätzlich sind der Start und das Ziel mittig positioniert. Da dies der Startpunkt für den self-avoiding walk (SAW) ist, müssen beide Seiten neben dem Start und Ziel gerade sein.
Die Größe kann dann durch folgende Formel dargestellt werden:

Sei die Größe des Rasters $x$, so gilt

$$x = 4k + 3$$

für $k \in \mathbb{N}$.

Dabei ist das $k$ eine wählbare Variable, um eine Größe für das Raster zu bestimmen. Dazu steht die drei für die Start-/Zielkachel und die Außenwand auf beiden Seiten des Rasters, welche nicht mit einberechnet werden bei der zweischrittigen Weggenerierung. Die restlichen Kacheln müssen, nachdem sie durch zwei geteilt sind, gerade sein. Daraus ergibt sich die Vervierfachung der Variable.

### 6.3.2 Farben

Folgende Farben werden für die Umsetzung benötigt:

| **Name** | **Buchstabe** | **Farbe** |
|---|---|---|
| *Emerald* (Smaragdgrün) | *E* | |
| *White* (Weiß) | *W* | |
| *Black* (Schwarz) | *B* | |
| *Yellow* (Gelb) | *Y* | |
| *Red* (Rot) | *R* | |
| *Green* (Grün) | *G* | |
| *Orange* (Orange) | *O* | |
| *Purple* (Lila) | *P* | |
| *Blue* (Blau) | *U* | |
| *Pink* (Pink) | *K* | |
| *Brown* (Braun) | *N* | |

### 6.3.3 Wurzelknoten

Der Wurzelknoten ist ein Sequencenode, damit alle Kindknoten sequenziell ausgeführt werden können. Die Rasterparameter werden in diesem entsprechend gesetzt. Dazu zählen alle Farben aus Kapitel 6.3.2. Diese werden in der Reihenfolge übernommen. Da auch der Ursprung gesetzt wird, ist das Raster zu Beginn smaragdgrün und hat eine weiße Kachel in der Mitte.

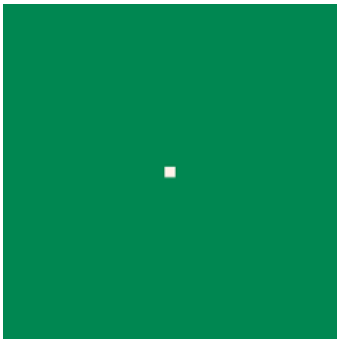


Abbildung 6.1: Startzustand des Rasters

Für die Umsetzung wird ein Beispiel zur Verdeutlichung verwendet. Dabei hat das Raster eine Größe von $31 \times 31$ und die Anzahl an direkten Abzweigungen vom Lösungsweg ist auf zehn gesetzt. Somit sieht das Raster zu Beginn aus wie in Abbildung 6.1.

## 6.4 Lösungsweg

Der Lösungsweg wird durch ein zweischrittiges Verfahren generiert, bei dem zuerst der Außenrand mit Start und Ziel erzeugt wird. Darauf aufbauend wird der Lösungsweg generiert.
In diesem aber auch den folgenden Kapiteln zur Umsetzung sind komplexere und ausschlaggebende Regeln im Fokus, sodass nicht alle Regeln gezeigt werden.

### 6.4.1 Generierung des Außenrandes mit Start und Ziel

Um einen Rand zu erstellen, kann das Raster mit der Farbe des Randes gefüllt werden und dann eine Ersetzungsregel angewendet werden, die nicht am Rand übereinstimmt und somit die Fläche mit einer anderen Farbe füllt. Der Rand soll smaragdgrün sein, weshalb dies die Grundfarbe des Rasters ist. Die Fläche soll schwarz sein.
Dies kann dann durch einen Parallelnode mit folgender Ersetzungsregel umgesetzt werden:

```
in="***/*E*/***" out="***/*B*/***"
```

Diese Regel ersetzt alle smaragdgrünen Kacheln, welche Nachbarkacheln haben (nach der Moore-Nachbarschaft), durch eine schwarze Kachel. Da der Rand nicht alle Nachbarn aus der Moore-Nachbarschaft aufweisen kann, bleibt dieser unberührt.

Für den Start und das Ziel muss die Mitte des Randes sich vom Rest abheben, damit dort ein Muster einer Regel übereinstimmen kann. Dafür wird die mittlere weiße Kachel verwendet, indem diese bis hin zum Rand erweitert wird. So wird eine vertikale Linie generiert.

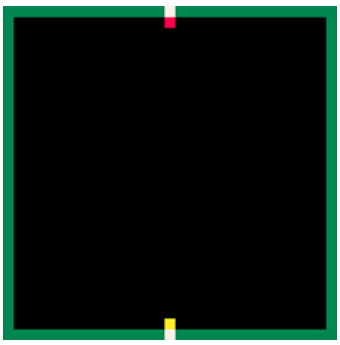


Abbildung 6.2: Das Raster nach der Generierung des Start- und Zielpunktes

An der Stelle, wo sich die mittlere Linie mit dem Rand schneidet, werden jeweils Start und Ziel (Start in Gelb und Ziel in Rot) wie in Abbildung 6.2 generiert. Zudem wird dann die Linie entfernt.

### 6.4.2 Generierung des Lösungsweges

Für den Lösungsweg wird nun, wie im Konzept vorgestellt, ein SAW vom Start ausgeführt. Dabei ist zu beachten, dass für zukünftige Schritte bekannt sein muss, an welchen Punkten es Abzweigungen geben kann. Denn der SAW begeht nur jede zweite Kachel und so kann auch nur von jeder zweiten Kachel eine Abzweigung entstehen. Dies kann direkt bei der Erstellung des SAWs umgesetzt werden. Dafür wird ein Onenode mit folgender Ersetzungsregel genutzt:

```
in="YBB" out="OWY"
```

Die gelbe Kachel steht für den aktuell besuchten Punkt und ist zu Beginn der Start. Der schon besuchte Weg wird durch orangene und weiße Kacheln dargestellt. Die orangenen Kacheln sind dabei an jeder zweiten Stelle, von denen es Abzweigungen geben kann.

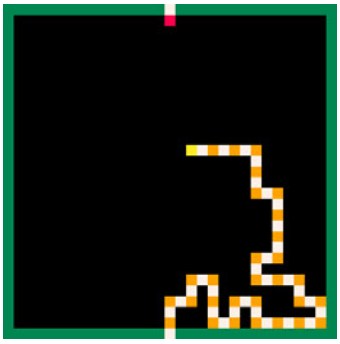


Abbildung 6.3: Generierung des Lösungsweges

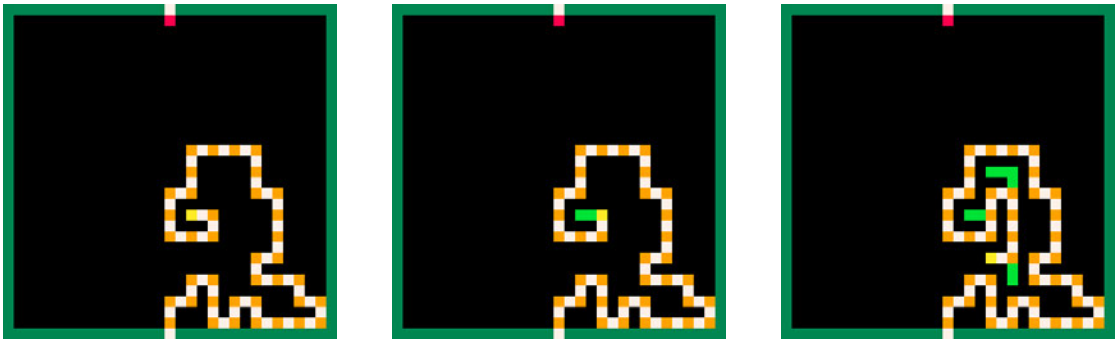
Das Eingabemuster der Regel identifiziert freie Stellen vor dem aktuellen Punkt. Das Ausgabemuster sorgt dann dafür, dass diese freie Stelle durch den Weg ersetzt wird und der aktuelle Punkt an die Spitze getan wird. So wird durch diese Ersetzungsregel ein Wandern des Pfades dargestellt (siehe Abbildung 6.3).

Damit der SAW auch Richtung Ziel verläuft, wird auf dem Onenode ein Field erstellt. Das Field wird für die Farbe Gelb, Richtung der Farbe Rot und auf schwarzem Untergrund erstellt. Mit dem Temperaturparameter lässt sich dann steuern, wie streng der nächste Punkt zum Ziel gewählt werden soll.

Ein Aspekt aus dem Konzept ist jedoch noch offen: die Zurückverfolgung des Weges, sobald der SAW in eine Sackgasse gerät. Da die Zurückverfolgung erst geschehen soll, wenn der selbst meidende Pfad nicht mehr weitergehen kann, wird hierfür ein Markovnode verwendet. Bei diesem ist der Onenode für den SAW der erste Kindknoten und der zweite Kindknoten beschreibt die Zurückverfolgung. Dieses Konzept wurde schon von Maxim Gumin in seinem Projekt [12] für die Umsetzung des rekursiven Backtrackers vorgestellt. Die Zurückverfolgung kann ebenfalls durch einen Onenode mit folgender Ersetzungsregel umgesetzt werden:

```
in="YWO" out="GGY"
```

Das Eingabemuster stimmt nur beim schon gegangenen Weg überein, betrachtet von dem aktuellen Punkt.



(a) Lösungsweg läuft in eine Sackgasse

(b) Zurückverfolgung des Weges

(c) Lösungsweg nach mehrfacher Zurückverfolgung

Abbildung 6.4: Zurückverfolgung bei der Generierung des Lösungsweges

Der aktuelle Punkt wird dann zurückgeschoben und der zurückverfolgte Weg wird grün markiert (siehe Abbildung 6.4). Durch den Markovnode wird garantiert, dass der erste Knoten priorisiert ausgeführt wird und somit immer versucht wird, den SAW zu erstellen.

Jetzt läuft der SAW Richtung Ziel, jedoch würde er vor dem Ziel vorbeilaufen. Dafür wird eine Regel eingeführt, welche darauf achtet, ob die gelbe Kachel vor der roten Kachel ist. Ist dies der Fall, wird der Weg mit dem Ziel verbunden. Dies kann mit einem Onenode umgesetzt werden, welcher an die erste Stelle in den Markovnode getan wird. So hat dieser Knoten die höchste Priorität, sodass sich der Weg direkt mit dem Ziel verbindet, wenn dieser davor ist. Bis dahin werden die anderen beiden Knoten ausgeführt.

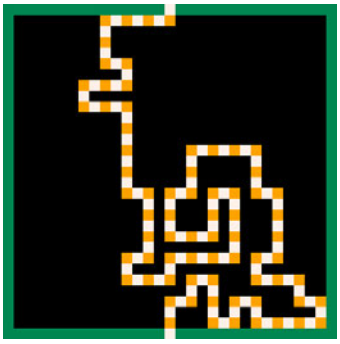


Abbildung 6.5: Fertig generierter Lösungsweg

Zum Schluss werden dann noch alle zurückverfolgten Wege entfernt und der Lösungsweg ist fertig generiert (siehe Abbildung 6.5).

## 6.5 Abzweigungen

Die Generierung der Abzweigungen kann in zwei Hauptteile geteilt werden. Der erste Teil ist die Generierung der Abzweigungspunkte. Der zweite Teil beinhaltet die Generierung der Wege für die Abzweigungen.

### 6.5.1 Abzweigungspunkte

Für die Generierung der Abzweigungspunkte werden zuerst alle Abzweigungen für die Teilbereiche generiert. Darauffolgend werden die restlichen Abzweigungen generiert.

#### Abzweigungspunkte in den Teilbereichen

Die Menge an Abzweigungen soll kontrollierbar sein und immer einer festen Zahl entsprechen. Dafür muss gezählt werden, wie viele Abzweigungen schon generiert worden sind, damit die restliche Anzahl im nächsten Schritt generiert werden kann. Dies wird umgesetzt durch einen in der Generierung eingebauten Zähler. Dabei ist das Grundprinzip, dass eine Anzahl an Kacheln gefärbt wird, welche den Zähler darstellen. Ist eine Abzweigung generiert, wird eine Kachel des Zählers weggenommen. Wenn noch Kacheln des Zählers übrig sind, werden abhängig davon dann pro Kachel die letzten Abzweigungen generiert.

Für den Zähler muss ein freier Bereich im Raster verwendet werden, sodass die restliche Generierung davon nicht beeinflusst wird. Zusätzlich muss der Zähler durch Ersetzungsregeln gut von den Abzweigungspunkten erreicht werden. Dafür lässt sich der Rand verwenden, da dieser nicht für die Generierung relevant ist und zudem über den Lösungsweg von den Abzweigungen erreicht werden kann. Ein Onenode färbt mit dem Schritteparameter eine feste Anzahl an Kacheln.

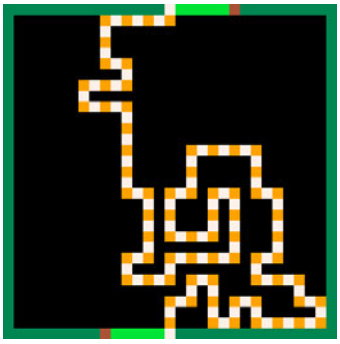


Abbildung 6.6: Das Raster nach Generierung des Zählers

Dabei wird der Zähler zufällig auf beide Seiten des Randes aufgeteilt, um so die späteren Schritte zu optimieren. Dafür wird die Überschneidung des Randes mit dem Lösungsweg als Muster verwendet, um so die Stelle zu identifizieren, an der der Zähler generiert wird. Der Zähler ist grün und wird durch eine braune Kachel begrenzt (siehe Abbildung 6.6).

Danach werden alle Teilbereiche markiert. Dafür werden zwei Parallenodes verwendet. Der erste ersetzt alle schwarzen $2 \times 2$ Felder mit roten $2 \times 2$ Feldern. Dadurch werden alle Bereiche, welche groß genug sind, um Abzweigungen zu haben, markiert. Der zweite Parallelnode hat folgende Ersetzungsregel:

```
in="**O/*R*/O**" out="***/*B*/***"
```

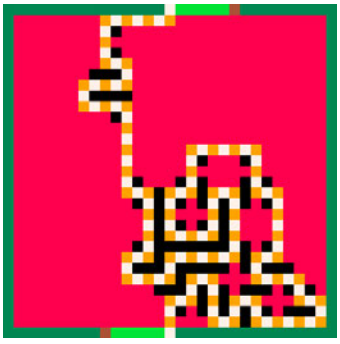


Abbildung 6.7: Das Raster nach der Markierung der Teilbereiche

Diese sorgt dafür, dass alle Ecken der Teilbereiche um den Lösungsweg herum entfernt werden (siehe Abbildung 6.7). Dies ist wichtig, da es passieren kann, dass manche Teilbereiche direkt aneinander grenzen und so sonst nicht zu unterscheiden wären.

Nun kann der Prozess für die Generierung der Abzweigungspunkte starten. Dabei wird eine Sequenz mehrfach wiederholt. Daher wird der Prozess in einem Sequencenode umge-

setzt. Dadurch, dass dieser Sequencenode ein Kindknoten eines anderen Sequencenodes ist, wird dieser auch immer wiederholend ausgeführt, bis keiner seiner Kindknoten mehr anwendbar ist.
Zu Beginn wird der erste Abzweigungspunkt mithilfe eines Onenodes mit folgender Ersetzungsregel markiert:

```
in="ORR" out="PYY"
```

Dabei werden die Abzweigungen nur für die rot markierten Teilbereiche generiert. Dies wird durch den Schritteparameter auf eine Ausführung begrenzt.

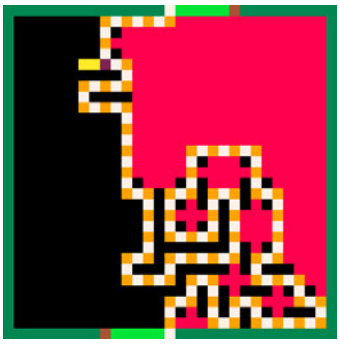


Abbildung 6.8: Das Raster nach Generierung des ersten Abzweigungspunktes

Darauffolgend wird der Teilbereich wieder schwarz gefärbt, da dieser nun schon einen Abzweigungspunkt hat (siehe Abbildung 6.8). Dafür wird ausgehend von der neuen Abzweigung das rote Feld durch ein Blaues ersetzt, damit dieses dann wiederum durch ein Schwarzes ersetzt werden kann. Lediglich Rot durch Schwarz zu ersetzen funktioniert nicht, da sonst auch die anderen Teilbereiche schwarz gefärbt werden würden.

Im kommenden Schritt muss das Wegnehmen einer grünen Zähler-Kachel vorbereitet werden. Dies wird immer über den Lösungsweg geschehen, da dieser direkt mit dem Rand verbunden ist.
Die Einstiegspunkte für den Zähler müssen markiert werden, damit diese sich unterscheiden. Dadurch können Regeln angewendet werden, ohne dass ein anderer Teil des Rasters davon betroffen ist. Die Punkte sind dabei der Start und das Ziel des Lösungsweges. Da dieser Prozess in einem Sequencenode ist, der sich immer wiederholt, muss sichergestellt werden, dass jede Ersetzungsregel aufbauend auf den gerade erzeugten Abzweigungspunkten geschieht. Denn ansonsten würde der Sequencenode auch nach Erzeugung aller Abzweigungen in den Teilbereichen, weiter ausgeführt werden.

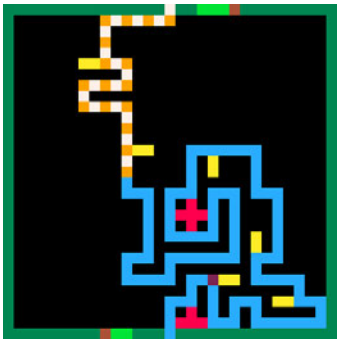


Abbildung 6.9: Flutung des Lösungsweges für den sechsten Abzweigungspunkt

Daher wird ausgehend von der gerade erzeugten Abzweigung in beide Richtungen der Lösungsweg blau geflutet (siehe Abbildung 6.9). Dies wird durch einen Parallelnode mit zwei Ersetzungsregeln umgesetzt:

```
in="UW" out="UU"
in="UO" out="UU"
```

Als Grundlage dient dafür die lilafarbene Kachel des Abzweigungspunktes, welche zuvor genutzt wird, um daneben die erste blaue Kachel zu generieren. Nun können so beide Einstiegspunkte in Abhängigkeit der blauen Kachel im Rahmen markiert werden. Dafür wird eine pinke Kachel verwendet. Danach wird der Lösungsweg ausgehend von den pinken Kacheln wieder zurück gefärbt.
Da der Zähler später auf einer der Seiten leer sein kann, muss sichergestellt werden, dass auch nur ein Einstiegspunkt markiert wird, wenn auf der Seite noch Zähler-Kacheln existieren. Dafür soll ausgehend von dem Einstiegspunkt der Rahmen in Richtung grüner Kacheln geflutet werden. Falls jedoch noch alle grünen Kacheln auf einer Seite existieren, ergibt eine Flutung keinen Sinn.

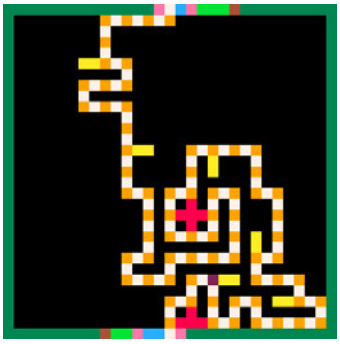


Abbildung 6.10: Das Raster nach der Flutung des Rahmens in Richtung grüner Kacheln

Daher werden vorerst nur die Einstiegspunkte wieder weiß markiert und die pinke Kachel verschoben, wenn keine grüne Kachel direkt angrenzt. Dies stellt den Startzustand für die Flutung dar. Die Flutung wird wieder blau sein, hat aber zusätzlich eine pinke Spitze (siehe Abbildung 6.10). Sie wird durch einen Onenode umgesetzt mit folgender Ersetzungsregel:

```
in="KE" out="UK"
```

Dazu ist ein Field auf dem Onenode definiert, sodass die Flutung auch nur in Richtung einer grünen Kachel passiert:

```
for="K" to="G" on="E"
```

Dazu ist der Notwendigkeitsparameter für das Field gesetzt, um so nur die Flutung auszuführen, wenn auch wirklich noch grüne Kacheln existieren.

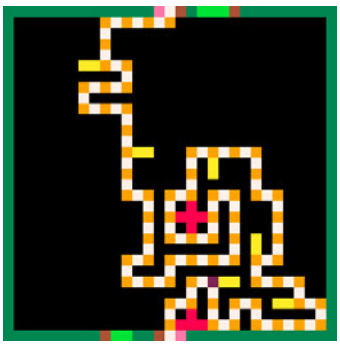


Abbildung 6.11: Das Raster nach der Zurückverfolgung der Flutung

Trifft die Flutung auf eine grüne Kachel, wird abhängig davon eine braune Kachel generiert. Diese wird genutzt, um die Flutung zurückzuverfolgen (siehe Abbildung 6.11). So

kann der Einstiegspunkt dann abhängig von der braunen Kachel wieder markiert werden oder bleibt weiß, wenn keine braune Kachel dort ist.

Jetzt sind die Einstiegspunkte markiert, wenn noch Zähler-Kacheln übrig waren und zuvor ein neuer Abzweigungspunkt generiert wurde. Auf Grundlage der Markierung wird daher dann eine Kachel vom Zähler entfernt.

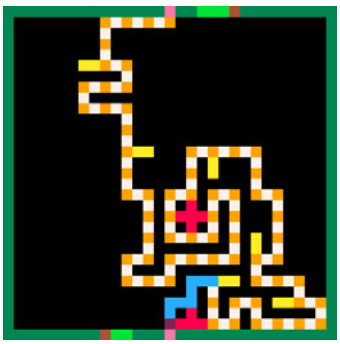


Abbildung 6.12: Das Raster nach der Flutung des Lösungsweges zum nächsten Einstiegspunkt

Dafür wird ausgehend vom Abzweigungspunkt der Lösungsweg in Richtung des nächsten markierten Einstiegspunktes geflutet. Dies wird durch einen Onenode und ein Field im ähnlichen Prinzip wie für die Flutung Richtung grüner Kacheln umgesetzt (siehe Abbildung 6.12). Erreicht die Flutung den Einstiegspunkt, wird dieser durch die Spitze der Flutung ersetzt. Danach wird der Lösungsweg wieder zurück gefärbt.
Um eine grüne Kachel zu entfernen, wird wieder eine Flutung in Richtung grüner Kacheln auf dem Rand ausgeführt. Diese wird abhängig von der lilafarbenen Spitze der vorherigen Flutung durchgeführt. Trifft die Flutung auf eine Zähler-Kachel, wird diese entfernt.

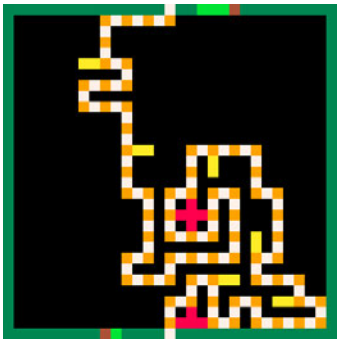


Abbildung 6.13: Das Raster nach Verringerung des Zählers

Zuletzt werden alle Kacheln wieder auf ihre korrekte Ausgangsfarbe zurück gefärbt und der Zähler wurde um eine Kachel verringert (siehe Abbildung 6.13).

In diesem Prozess soll der Sequencenode so lange wiederholt werden, bis keine offenen Teilbereiche mehr existieren. Dies wird dadurch gewährleistet, dass alle Regeln abhängig von den generierten Abzweigungspunkten agieren. Diese werden pro Wiederholung in Abhängigkeit eines offenen Teilbereiches generiert. Dazu wird der Teilbereich schwarz gefärbt und so reduziert sich die Anzahl an offenen Teilbereichen. Daher hört der Sequencenode auf, sich zu wiederholen, sobald es keine offenen Teilbereiche mehr gibt.

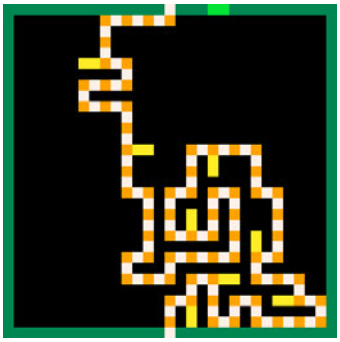


Abbildung 6.14: Das Raster nach der Generierung aller Abzweigungspunkte für die Teilbereiche

Nach dem Prozess hat jeder Teilbereich eine Abzweigung und der Zähler wurde dazu entsprechend verringert (siehe Abbildung 6.14).

#### Restliche Abzweigungspunkte

Bevor für die restlichen Zählerkacheln Abzweigungspunkte generiert werden, wird zuerst der braune Zählerrand entfernt. Für den folgenden Prozess muss erneut etwas wieder-

holend ausgeführt werden. Daher wird wieder ein Sequencenode verwendet. In diesem wird zuerst abhängig von einer grünen Kachel das gesamte Raster blau gefärbt. Dies wird durch einen Onenode und einen Parallelnode umgesetzt. Der Onenode hat vier Ersetzungsregeln:

```
in="GB" out="EU"
in="G*B" out="E*U"
in="G*/*B" out="E*/*U"
in="G*/**/*B" out="E*/**/*U"
```

Der Onenode sorgt dafür, dass abhängig von einer grünen Kachel am Rand die nächste schwarze Kachel blau gefärbt wird. Dabei gibt es mehrere Fälle, da der Lösungsweg im Weg sein kann. Zusätzlich wird damit die grüne Kachel entfernt und so der Zähler wieder verringert. Der Schritteparameter limitiert die Ausführung auf einen Schritt. Aufbauend darauf wird ein Parallelnode verwendet, welcher drei Ersetzungsregeln hat:

```
in="UB" out="UU"
in="UOB" out="UOU"
in="UWB" out="UWU"
```

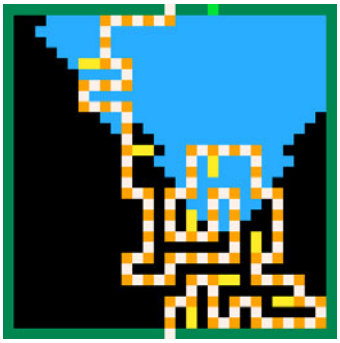


Abbildung 6.15: Färbung des Rasters in Abhängigkeit einer Zähler-Kachel

Dieser sorgt dafür, dass durch die eine blaue Kachel das restliche Raster blau gefärbt wird (siehe Abbildung 6.15). Auch hier kann der Lösungsweg im Weg sein, weshalb es mehrere Regeln geben muss. Nun wird abhängig von der blauen Fläche ein Abzweigungspunkt generiert. Danach wird die blaue Fläche wieder schwarz gefärbt.

In dem aktuell beschriebenen Prozess soll der Sequencenode so lange wiederholt werden, bis es keine Zähler-Kacheln mehr gibt. Das wird dadurch gewährleistet, dass alle Regeln abhängig von den grünen Kacheln sind. Da diese pro Wiederholung reduziert werden, hört der Sequencenode auf, wenn alle Zähler-Kacheln entfernt wurden.

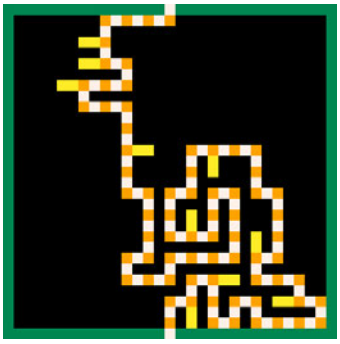


Abbildung 6.16: Das Raster nach der Generierung aller Abzweigungspunkte

Nach der vollen Ausführung des Prozesses sind alle Abzweigungspunkte generiert (siehe Abbildung 6.16).

### 6.5.2 Wege

Vorerst wird der Lösungsweg grün markiert, damit von ihm aus keine weiteren Wege generiert werden.

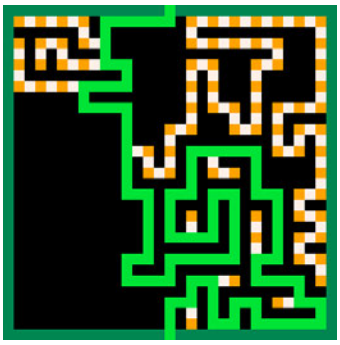


Abbildung 6.17: Das Raster nach der Generierung aller Wege für die Abzweigungen

Danach werden von jeder Abzweigung mithilfe eines Allnodes parallel SAWs generiert (siehe Abbildung 6.17). Dies geschieht im selben Prinzip wie die Generierung des Lösungsweges. Der Unterschied ist der, dass es keine Zurückverfolgung gibt und die Generierung komplett zufällig geschieht. Nachdem für jeden Abzweigungspunkt die ersten Wege generiert wurden, werden nun die weiteren Wege generiert. Dies geschieht wiederholend in einem Sequencenode.

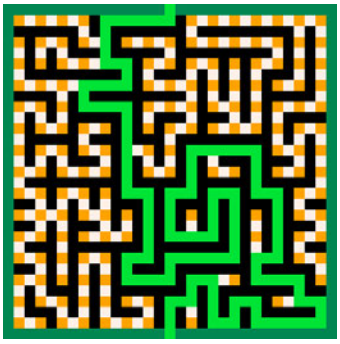


Abbildung 6.18: Das Raster nach der Generierung aller Wege (Lösungsweg in Grün)

Dabei werden immer wieder von einem zufälligen möglichen Abzweigungspunkt (orangene Kacheln) SAWs generiert. Dies geschieht so lange, bis das Labyrinth gefüllt ist (siehe Abbildung 6.18). Zum Schluss werden auch die restlichen Wege grün markiert, um darauf aufbauend das Design anzupassen.

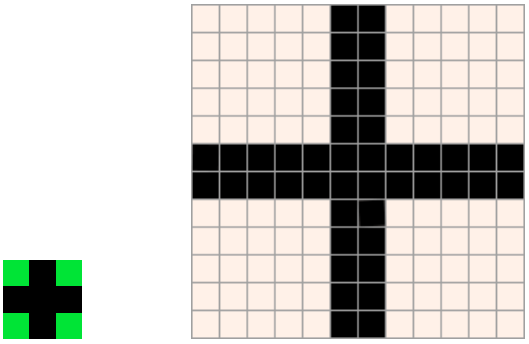
## 6.6 Aussehen

Das Labyrinth ist in seiner Struktur fertig generiert. Dabei sind die Wände und Wege jeweils eine Kachel breit. Dies war für die Generierung von Vorteil. Nun sollen die Wände jedoch schwarz und schmal sein und die Wege weiß und breit. Dafür wird ein Mapnode verwendet. Dieser skaliert das generierte Labyrinth um das Vierfache. Damit ist es möglich, das Originalraster zu nutzen, um dessen Strukturen auf das neue, größere Raster zu übertragen. Dabei werden die Wege und Wände anders skaliert und gefärbt, um so das gewollte Design zu erreichen.

Folgende Regel ist ein Beispiel für das Übertragen einer Wandkreuzung vom alten in das neue Raster:

```
in="GBG/BBB/GBG"
out="************/************/************/************/
     *****BB*****/****BBBB****/****BBBB****/*****BB*****/
     ************/************/************/************"
```

Das Eingabemuster wird im alten Raster gesucht und stimmt bei jeder Wandkreuzung überein. Das neue Raster ist zu Beginn weiß gefärbt, weshalb jede Wildcard im Ausgabemuster bedeutet, dass dort weiße Kacheln sind.



(a) (b)

Abbildung 6.19: Beispielausschnitt der Skalierung durch einen Mapnode vom Originalraster (a) zum neuen Raster (b)

Das Ausgabemuster ist um das Vierfache größer als das Eingabemuster. Dabei sind die Wände nicht im Vierfachen skaliert worden, was dazu führt, dass diese schmaler sind. In Abbildung 6.19 ist zu sehen, wie diese Skalierung funktioniert. In 6.19b ist zur Verdeutlichung die Rasterung mit angegeben.

Dieses Verfahren muss dann für alle weiteren Wandstrukturen, wie zum Beispiel eine einfache gerade Wand, ausgeführt werden.

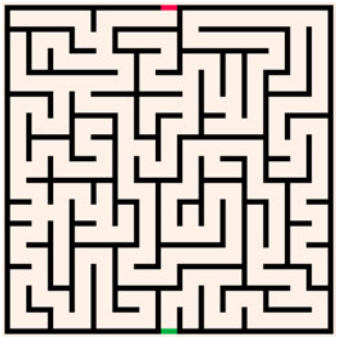


Abbildung 6.20: Das Labyrinth nach vollendeter Generierung

Dazu werden dann noch der Start (Grün) und das Ziel (Rot) farblich markiert. Damit ist das vollendete generierte Labyrinth in Abbildung 6.20 zu sehen.

# 7 Evaluation

In diesem Kapitel wird hinterfragt, wie erfolgreich das Konzept umgesetzt werden konnte und ob damit alle Anforderungen erfüllt wurden. Zusätzlich wird die eigene Arbeit an der Generierung des Labyrinthes verwendet, um daraus Schlüsse über Markov Junior (MJ) zu schließen.

## 7.1 Auswertung des Konzeptes und dessen Umsetzung

Für die Auswertung wird betrachtet, wie gut die Anforderungen erfüllt worden sind und analysiert, was für Probleme sich aus dem Konzept ergeben haben. Zusätzlich wird geschaut, wie performant der Ansatz ist.

### 7.1.1 Anforderungserfüllung

Die Anforderungen bestanden aus drei Hauptteilen: grundlegende Eigenschaften des Labyrinthes, das Aussehen des Labyrinthes und die Kontrollierbarkeit der Generierung.

#### Eigenschaften des Labyrinthes

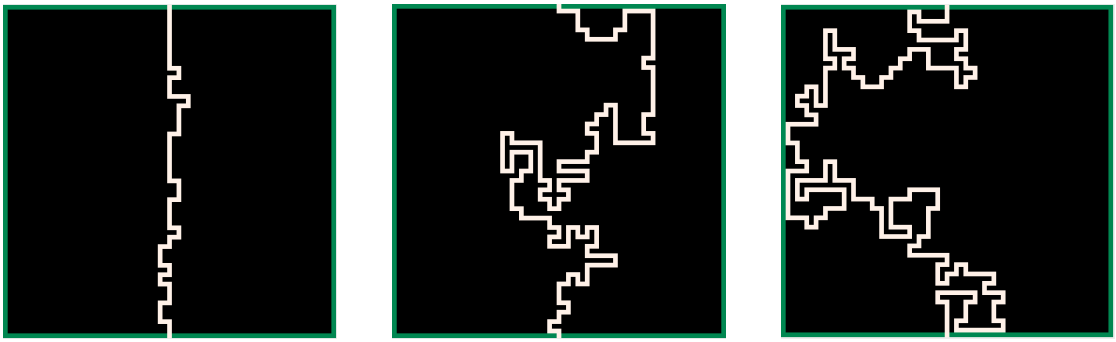
Das Hauptkonzept der Generierung ist der self-avoiding walk (SAW) und eine mehrfache Aneinanderreihung von diesem. Maxim Gumin hat dies in seinem Projekt [12] schon vorgestellt und die eigene Implementierung hat zudem gezeigt, dass dieser in MJ umsetzbar ist. In anderen schon existierenden Algorithmen (rekursiver Backtracker [9] oder auch Prim & Kill [2]) wurde gezeigt, dass dadurch perfekte Labyrinthe generiert werden. Auch die eigene Implementierung hat dies gewährleistet. Zudem konnte dadurch auch die Einheitlichkeit aller generierten Labyrinthe erfüllt werden.
Die Kombination der Implementierung des SAWs und MJs Raster sorgen dafür, dass auch die gamma-Eigenschaft des Labyrinthes immer erfüllt ist.
Außerdem stellt der Beginn der Generierung sicher, dass es einen Start und ein Ziel gibt. So hat jedes generierte Labyrinth auch diese Eigenschaft aufweisen können.

### Aussehen des Labyrinthes

Das Konzept für das Aussehen konnte durch den Mapnode von MJ umgesetzt werden. So hat jedes generierte Labyrinth weiße, breite Wege und schwarze, schmale Wände aufweisen können. Auch der Start und das Ziel sind bei jedem generierten Labyrinth in Grün und Rot markiert.

### Kontrollierbarkeit der Generierung

Das Labyrinth sollte in zwei Hauptbestandteilen kontrollierbar sein: der Verlauf des Lösungsweges und die Anzahl an Abzweigungen.
Das Konzept zur Kontrollierbarkeit des Lösungsweges konnte durch einen Onenode und einem Field umgesetzt werden. Der Temperaturparameter steuert dabei wie streng der Lösungsweg zum Ziel verläuft. Um zu schauen, ob dadurch auch der Verlauf des Lösungsweges wie zu erwarten gesteuert werden kann, wurden Messungen ausgeführt. Dabei wurde ein Raster der Größe $71 \times 71$ verwendet und gezählt, wie viele Kacheln der Lösungsweg ausmacht. Die niedrigste Anzahl wären 71 Kacheln, was eine gerade Linie zwischen Start und Ziel wäre. Zu erwarten wäre, dass eine niedrige Temperatur im Durchschnitt kürzere Wege generiert und eine hohe Temperatur willkürlichere Wege generiert.



(a) niedrig - 2 (b) mittel - 8 (c) hoch - 14

Abbildung 7.1: Beispiele für Lösungswege unterschiedlicher Temperaturen

| Temperatur | Gerundete Durchschnittsanzahl |
|---|---|
| niedrig (2) | 103 |
| mittel (8) | 301 |
| hoch (14) | 405 |

Tabelle 7.1: Ergebnisse zur Messung der Durchschnittsanzahl an Kacheln des Lösungsweges nach 30 Messungen bei einem Raster der Größe $71 \times 71$

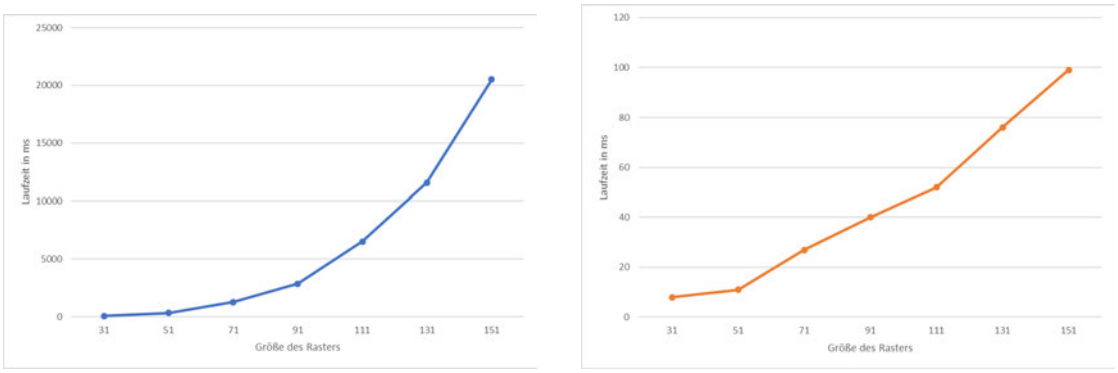
Die Ergebnisse in 7.1 und Beispielwege in Abbildung 7.1 zeigen, dass dies der Fall ist und eine Kontrolle des Verlaufes des Lösungsweges, wie in den Anforderungen (siehe Kapitel 5.1.3) spezifiziert, möglich ist.
Das Konzept für die Abzweigungen konnte durch einen selbst erstellten Zähler in MJ umgesetzt werden. Dabei musste zusätzlich auf die Teilbereiche des Labyrinthes geachtet werden, damit alle Bereiche des Labyrinthes Wege besitzen und es somit perfekt und einheitlich ist. Dadurch sind bei mehreren generierten Labyrinthen mehr direkte Abzweigungen gezählt worden, als sie eingestellt wurden. Dies entspricht daher nicht den zuvor gestellten Anforderungen (siehe Kapitel 5.1.3). Jedoch ist dies notwendig gewesen, um die angeforderten Grundeigenschaften (siehe Kapitel 5.1.1) der generierten Labyrinthe zu erfüllen, welche Priorität haben. Durch den Ansatz wurde aber sichergestellt, dass mindestens die gewollte Anzahl und maximal die notwendige Anzahl an direkten Abzweigungen existiert. Alle generierten Labyrinthe konnten dies nachweisen.

### 7.1.2 Probleme

Die Anforderung für das Aussehen besagt, dass der Start grün und das Ziel rot markiert sein sollen, um diese unterscheiden zu können. Dies wurde umgesetzt, indem der Eingang

und der Ausgang eine entsprechend farbliche Linie bekommen haben. Dabei wurde nicht beachtet, dass dadurch unklar ist, was der Start und das Ziel sind. Denn erkennbar sind diese, jedoch gibt es keine Legende, welche darstellt, welche Farbe zum Start oder Ziel gehört.
Zusätzlich sieht der Ein- und Ausgang geschlossen aus, was auch zur Verwirrung führen kann (siehe Abbildung 6.20). Diese Punkte zeigen, dass ein einfaches Weglassen der Markierung das Labyrinth intuitiver machen könnte.

Im Bereich der Kontrollierbarkeit der Labyrinthe haben sich die meisten Probleme ergeben.
Der Lösungsweg kann zwar, wie zuvor gezeigt, in seinem Verlauf kontrolliert werden, jedoch ist dies nur über einen Parameter steuerbar, der eine Wahrscheinlichkeit repräsentiert. Daher ist lediglich der Durchschnitt passend zur Temperatur und Ausreißer sind nicht unwahrscheinlich. So kann eine hohe Temperatur auch mehrfach ein Labyrinth generieren, bei dem der Lösungsweg wenige Kacheln einnimmt.
Ein weiteres Problem ergibt sich aus den Abzweigungen des Lösungsweges. Zwar ist garantiert, dass immer die gewollte Anzahl an Abzweigungen vom Lösungsweg abgeht, jedoch sagt dies nichts über die Länge dieser aus. Das bedeutet, dass es viele Abzweigungen geben kann, die eine direkte Sackgasse sind. Dies wurde bei mehrfacher Generierung für viele Labyrinthe beobachtet. Hauptsächlich sind dies die Abzweigungen, welche für kleine Teilbereiche aufgebraucht werden. Auch wenn diese Abzweigungen sind, haben sie keinen direkten Einfluss auf das Lösen des Labyrinthes, da diese direkt ignoriert werden können. Daraus ergibt sich die Frage, ob solche Abzweigungen auch mit in die Gesamtanzahl einberechnet werden sollen.
Zuletzt ergibt sich ein Problem aus der Umsetzung des Konzeptes für die Abzweigungspunkte. In MJ musste dafür ein eigener Ansatz entwickelt werden, welcher viele Schritte beinhaltet und so die Komplexität des Programms erhöht hat.




(a) Eigener Ansatz (b) Rekursive Backtracker

Abbildung 7.2: Laufzeitvergleich zweier Ansätze zur Generierung von Labyrinthen

In Abbildung 7.2a ist zu sehen, wie sich die Laufzeit für die Ausführung der Generierung des eigenen Ansatzes bei unterschiedlichen Labyrinthgrößen verändert. Im Vergleich dazu sieht man in Abbildung 7.2b die Laufzeit für den rekursiven Backtracker Algorithmus in MJ [12], welcher in die gleiche Umgebung des Ansatzes aus dieser Arbeit eingebunden wurde (Start und Ziel mittig, Anpassung des Aussehens am Ende). Zu beachten ist, dass sich die maximale Laufzeit in beiden Grafiken unterscheidet.
In dem Ansatz aus dieser Arbeit ist ein exponentieller Verlauf zu erkennen, während der rekursive Backtracker eher einen linearen Verlauf hat. Zudem ist die Laufzeit im eigenen Ansatz im Allgemeinen deutlich höher. Eine erhöhte Laufzeit ist zu erwarten, da der eigene Ansatz mehr Möglichkeiten für die Generierung bietet. Jedoch zeigt der Verlauf, dass der Ansatz nicht effizient ist.

## 7.2 Analyse von Markov Junior

Durch die Arbeit mit MJ für die Generierung der Labyrinthe ist Einiges über MJ klar geworden. Diese Erkenntnisse werden im Folgenden näher erläutert.

### 7.2.1 Möglichkeiten

MJ ist eine probabilistische Programmiersprache, aufbauend auf dem Markow-Algorithmus, welcher Turing-vollständig ist. Zwar ist MJ nicht mehr Turing-vollständig, besitzt aber weiterhin eine Mächtigkeit, um verschiedenste Funktionen zu berechnen. Im Projekt von Maxim Gumin wurde beispielsweise gezeigt, wie mithilfe von MJ *Sokoban* Level gelöst werden. Furnas [8] stellt ein sehr ähnliches Programm vor, welches mit visuellen Ersetzungsregeln wie MJ arbeitet. Dabei wird dies in einem Beispiel genutzt, um getrennte

Komponenten zu zählen und die Anzahl durch römische Zahlen anzuzeigen. Das ist auch in MJ umsetzbar und zeigt, dass MJ in der Lage ist, unterschiedlichste Funktionen auszuführen.
Das Lösen des Sokoban Levels, aber auch das *Circuit* Modell von Maxim Gumin [12] zeigen zudem, dass MJ auch für Animationen verwendet werden kann.

Wie schon in Kapitel 5.2 erklärt, ist MJ auch in der Lage dazu, Inhalte prozedural zu generieren. Die Umsetzung in dieser Arbeit zur Generierung eines kontrollierbaren Labyrinthes ist ein Beispiel dafür.

### 7.2.2 Stärken und Vorteile

MJ baut auf dem Markow-Algorithmus auf, welcher Turing-vollständig ist. Dies zeigt also, wie mächtig dieses Grundkonzept der Ersetzungsregeln ist. Zwar ist MJ nicht mehr Turing-vollständig, jedoch bietet das Konzept der Knoten eine Vielfalt an neuen Funktionen. Daraus ergeben sich nicht nur neue Möglichkeiten, sondern auch Wege, gewisse Dinge einfacher umsetzen zu können. Dabei kann der Pathnode zum Beispiel direkt verwendet werden, um einen Weg zwischen zwei Punkten zu erstellen, was sonst durch mehrere Regeln umgesetzt werden müsste. Dazu gibt es auch Knoten, wie den WaveFunctionCollapse Node, welche einen komplett eigenen Algorithmus zur prozeduralen Generierung in MJ einbetten, was wiederum die Mächtigkeit von MJ erhöht.

Zudem erlaubt das Konzept der visuellen Ersetzungsregeln von MJ, durch wenig Code viel erzeugen zu können. Die wiederholte Anwendung der Regeln mit ihren Knoten, welche weitere Logik hinzufügen, ermöglicht dieses Verhalten. Ein Beispiel ist die Umsetzung des rekursiven Backtrackers in MJ [12]:

```
<markov values="BRGW" origin="True">
  <one in="RBB" out="GGR"/>
  <one in="RGG" out="WWR"/>
</markov>
```

Dieser kann in nur vier Zeilen Code dargestellt werden, was in anderen Programmiersprachen nicht möglich sein wird.

MJs grundlegende Logik ist visueller Natur. Daher können gewisse Strukturen, welche in dem zu erzeugenden Inhalt vorzufinden sind, häufig intuitiv in MJ abgebildet werden. Ein Beispiel dafür ist der SAW. Einfach beschrieben darf der aktuelle Punkt dabei nur nach vorn geschoben werden, wenn davor eine ausreichend freie Fläche ist. Und genau

diese Beschreibung kann so direkt auf MJ übertragen werden und eine entsprechende Ersetzungsregel ist dazu intuitiv zu erstellen.

Außerdem bietet MJ eine Visualisierung an, was bei anderen Anwendungen zuerst erstellt werden muss. Zusätzlich ist MJs Ausführung Schritte basiert, sodass auch jeder Schritt visuell angezeigt werden kann. Dies ist hilfreich zur Veranschaulichung des Ablaufes eines Programms.

### 7.2.3 Schwächen und Nachteile

Grammatiken sind im Allgemeinen sehr mächtig, jedoch ist die Erstellung einer Grammatik und dessen Regeln meist komplex. Es besteht häufig eine große Unklarheit darüber, was für eine Auswirkung die Regeln haben und wie Veränderungen der Regeln dazu einfließen. Daher ist ein Prozess von *Trial and Error* nicht untypisch beim Erstellen einer Grammatik. [19] [22]
MJ baut auf demselben Prinzip auf, wodurch sich dieses Problem auch für MJ ergibt. Das Erstellen eines Regelsets für die Generierung eines komplexeren Inhaltes wird daher schnell aufwendig. Auch bei der eigenen Umsetzung war dies zu bemerken.
Zudem kann MJ in den konstruktiven Ansatz für prozedurale Generierung eingeordnet werden. Daher muss durch das Regelset sichergestellt sein, dass alle Bedingungen für den generierten Inhalt erfüllt sind. Das erhöht ebenfalls die Komplexität, um mithilfe von MJ prozedural Inhalte zu generieren.

MJ limitiert sich auf das Notwendigste in Bezug auf die Möglichkeiten des Programmierens. Es gibt keine bedingten Anweisungen, Zählschleifen, Methoden oder Variablen. Alles basiert auf den visuellen Ersetzungsregeln. Wird nun jedoch etwas wie eine Variable benötigt, in der sich ein Zustand gemerkt wird, muss dies durch Umwege umgesetzt werden, wodurch MJ Programme schnell komplex werden. Die eigene Umsetzung ist ein Beispiel dafür. In dieser war es notwendig, dass abhängig von den schon generierten Abzweigungen die restlichen Abzweigungen generiert werden. Dafür musste gezählt werden, wie viele schon generiert worden sind. Die Implementierung dieses Zählers hat einen großen Teil des Codes ausgemacht und das Programm in seiner Komplexität erhöht.

In MJ ist es möglich, gewisse Programmparameter zu setzen. Diese werden direkt in dem Code gesetzt. Es gibt aber keine Möglichkeit, dass von außen spezifische Parameter übergeben werden. In Bezug auf die prozedurale Generierung stellt dies einen großen Nachteil dar, da es häufig erwünscht ist, die Generierung über eine Benutzerschnittstelle steuern zu können.

Möchte man zusätzlich spezielle Eigenschaften eines generierten Inhaltes steuern, welche mehrere Komponenten der Generierung betreffen, wird man mit MJ nicht weit kommen. Dies begründet sich mit dem Fehlen von Methoden, Variablen und bedingten Anweisungen, wodurch ganze Bereiche gekapselt werden können und bedingt entschieden werden kann, ob der eine oder andere Bereich für die Generierung verwendet werden soll.

Die abstrakte Darstellung der Programmlogik durch die Ersetzungsregeln führt schnell zu unübersichtlichem Code. Eine Ersetzungsregel kann häufig nur eine kleine Vorbereitung sein für die eigentliche Logik hinter der Generierung. Dies kann zwar durch Kommentare erklärt werden, aber ohne das gesamte Bild vor Augen zu haben, kann dies schwierig sein, einzuordnen und nachzuvollziehen. Zusätzlich kann es bei längeren Ersetzungsregeln kompliziert sein, sich deren Muster im Raster vorzustellen, um zu verstehen, was hinter der Regel steckt.

### 7.2.4 Nutzen und Potenzial

Zur Generierung oder Lösung von simplen bis mittel komplexen Grafiken oder Problemen ist MJ gut geeignet. In vielfachen Beispielen aus dem Projekt von Maxim Gumin [12] wurde dies gezeigt. Für hochauflösende und sehr komplexe Inhalte ist MJ zur Generierung nicht geeignet. Im Bereich der prozeduralen Generierung würden die von MJ generierten Inhalte gut zu Retro-Videospielen passen, da diese ebenfalls auf simpleren Rastergrafiken aufbauen.
Das Konzept der Knoten lässt eine einfache Schnittstelle offen für Erweiterungen durch weitere Knoten. Dadurch besteht das Potenzial für weitere Möglichkeiten und eventuell einer übersichtlicheren Darstellung des Codes. Damit wäre MJ auch besser zugänglich für komplexere Themen.
Zusätzlich kann eine Erweiterung von interaktiven Regeln, wie es auch Maxim Gumin angesprochen hat [12], MJ Programme zu Videospielen machen.

### 7.2.5 Markov Junior für die Labyrinthgenerierung

Die Umsetzung des Konzeptes in dieser Arbeit hat zeigen können, dass MJ für die grundlegende Struktur von Labyrinthen geeignet ist. Auch Beispiele aus dem Projekt von Maxim Gumin [12] für schon existierende graphbasierte Generierungsalgorithmen für Labyrinthe haben aufweisen können, dass MJ geeignet ist, um Labyrinthe zu generieren. In Bezug auf die Kontrollierbarkeit der Labyrinthe hat sich MJ jedoch an einigen Stellen als weniger geeignet erwiesen, da sich durch MJ eine erhöhte Komplexität ergeben hat.

Dies ist mit den fehlenden Konstrukten wie Methoden oder Variablen zu begründen, wodurch Umwege in MJ gesucht werden mussten.

# 8 Fazit

## 8.1 Zusammenfassung

Diese Arbeit hat den neuen Markov Junior (MJ) Algorithmus von Maxim Gumin näher erklärt, indem alle wichtigen Bestandteile vorgestellt wurden und an einem umfangreichen Beispiel erläutert wurden.
Dazu wurde in dieser Arbeit unter der Nutzung des MJ Algorithmus ein neuer Ansatz zur Generierung von perfekten Labyrinthen erstellt. Dieser Ansatz erlaubt es, den Verlauf des Lösungsweges zu steuern und festzulegen, wie viele Abzweigungen von dem Lösungsweg abgehen. Dafür wurde die Generierung in die Generierung des Lösungsweges und die Generierung der Abzweigungen aufgeteilt, um eine bessere Kontrollierbarkeit zu ermöglichen. Zusätzlich kann die Größe des Labyrinthes eingestellt werden.
Es hat sich eine exponentielle Entwicklung der Laufzeit bei einer proportionalen Steigerung der Rastergröße für den entwickelten Ansatz ergeben. Dies hat gezeigt, dass der Ansatz nicht effizient ist.
In der Arbeit wurde verdeutlicht, dass MJ für Vieles verwendet werden kann. Dazu zählen das Lösen von Funktionen/Problemen, das Animieren von Situationen, aber auch die prozedurale Generierung.
Zusätzlich wurde erläutert, dass MJ durch das Konzept der Ersetzungsregeln und Knoten sehr mächtig ist und mit wenig Code viel erzeugen kann. Die direkte Visualisierung ist ein weiterer Vorteil, wodurch sich MJ gut für prozedurale Generierung eignet. Dabei ist aufgefallen, dass dies für simplere oder mittel komplexe Grafiken gilt.
Zudem wurde in dieser Arbeit darauf eingegangen, dass das Grundkonzept der Ersetzungsregeln zwar mächtig ist, aber dazu auch komplex und unübersichtlich sein kann. Dies wurde damit begründet, dass es zum einen häufig unklar ist, was für einen Effekt spezifische Regeln haben und dass zum anderen in dem Regelset garantiert sein muss, dass alle Bedingungen für den generierten Inhalt erfüllt sind.
Es hat sich herausgestellt, dass MJ für die Generierung der grundlegenden Struktur der Labyrinthe geeignet ist. In Bezug auf die Kontrollierbarkeit jedoch hat sich MJ als weniger geeignet gezeigt. Durch dessen Limitierungen wie bspw. das Fehlen von Methoden

oder Variablen mussten Umwege gefunden werden, um Teile des Konzeptes umgesetzt zu kriegen. Diese haben die Komplexität des Ansatzes deutlich erhöht.

## 8.2 Mögliche Erweiterungen

### 8.2.1 Schwierigkeit des Labyrinthes

Diese Arbeit hat sich mit der Kontrollierbarkeit von Labyrinthen auseinandergesetzt. Ein im Allgemeinen wichtiger Faktor des Labyrinthes ist dessen Schwierigkeit. Diese zu kontrollieren, wäre daher ein interessanter Punkt zur Erweiterung.
Bei der Schwierigkeit ist zu beachten, dass die Perspektive des Lösenden eine zentrale Rolle spielt. Wird das Labyrinth von der Vogelperspektive aus gelöst, hat man einen gesamten Überblick über das Labyrinth. Jedoch kann ein Labyrinth auch für ein Videospiel verwendet werden, in dem man sich in dem Labyrinth befindet. In diesem Fall sieht man lediglich den Teil des Labyrinthes, der direkt vor einem ist. Bevor man sich mit der Schwierigkeit eines Labyrinthes beschäftigt, muss dies geklärt werden. Die folgenden Erweiterungen und Gedanken bauen auf dem Lösen des Labyrinthes aus der Vogelperspektive auf.
Aufbauend auf dem entwickelten Ansatz steht die Annahme im Raum, dass eine höhere Temperatur für den Lösungsweg, mit einer höheren Anzahl an Abzweigungen vom Lösungsweg zu einem komplexeren Labyrinth führen sollte. Dazu sollten die Abzweigungen möglichst gleichmäßig verteilt sein und direkte Sackgassen sollten verringert werden. Hat dies wirklich einen Einfluss auf die Schwierigkeit, wäre es indirekt auch mit diesem Ansatz möglich, die Schwierigkeit einzustellen. Dies zu analysieren wäre daher ein passender erster Schritt, um aufbauend auf dieser Arbeit den Begriff der Schwierigkeit einzuführen. Dies ist jedoch lediglich eine theoretische Hypothese. Die Schwierigkeit eines Labyrinthes lässt sich auch formal darstellen. So hat McClendon [17] eine formale Beschreibung der Schwierigkeit eingeführt. Als Erweiterung gilt es dann daraus zu extrahieren, wie durch ein Regelset in MJ diese formale Definition für die Schwierigkeit erreicht werden kann. Bellot et al. [2] haben aufbauend auf McClendon ein Konzept von *nicht signifikanten Wänden* eingeführt. Diese beschreiben, welche Wände irrelevant sind, wenn der Mensch beim Lösen des Labyrinthes das Labyrinth durchsucht. Dadurch sollen Labyrinthe auf ihren Spaßfaktor analysiert werden können. Das kann ebenfalls verwendet werden, um zu schauen, wie das in MJ umsetzbar ist.

### 8.2.2 Verbesserung der Effizienz

Der in dieser Arbeit umgesetzte Ansatz hat sich als wenig effizient herausgestellt. Der Grund dafür ist die Komplexität, welche sich durch das Erstellen eines Zählers ergeben hat. Als Erweiterung bietet es sich daher an, zu forschen, ob es entweder eine ganz andere Umsetzung geben kann oder ob die aktuelle Umsetzung durch die Anpassung von Regeln optimiert werden kann.
Eine Idee wäre dazu, den Zähler extern von der Generierung zu halten und einen Bereich für Kontrollelemente im Raster zu erstellen. Dadurch muss der Zähler nicht im Rahmen und somit auch nicht in das Labyrinth eingebaut werden. Dies würde die Übersichtlichkeit verbessern, aber von den Schritten und Regeln her sollte dies weniger einen Effekt haben.

### 8.2.3 Parametrisierung

Wie sich in der Analyse des MJ Algorithmus ergeben hat, ist es nicht möglich, spezifische Parameter von außen an MJ weiterzugeben. Dies ist jedoch ein wichtiger Teil der prozeduralen Generierung. Und auch in dieser Arbeit wäre es von Vorteil gewesen, wenn dies über die Benutzerschnittstelle möglich gewesen wäre. Daher kann das als Erweiterung für diese Arbeit genutzt werden, um MJ so anzupassen, dass für prozedurale Generierung Parameter von außen angegeben werden können. Für diese Arbeit würde man die Anzahl an Abzweigungen und die Temperatur für den Lösungsweg angeben wollen können.

# Literaturverzeichnis


[1] Ashlock, Daniel ; Lee, Colin ; McGuinness, Cameron: Search-based procedural generation of maze-like levels. In: *IEEE Transactions on Computational Intelligence and AI in Games* 3 (2011), Nr. 3, S. 260–273

[2] Bellot, Victor ; Cautrès, Maxime ; Favreau, Jean-Marie ; Gonzalez-Thauvin, Milan ; Lafourcade, Pascal ; Le Cornec, Kergann ; Mosnier, Bastien ; Rivière-Wekstein, Samuel: How to generate perfect mazes? In: *Information Sciences* 572 (2021), S. 444–459

[3] Cooper, Seth: Sturgeon-MKIII: Simultaneous level and example playthrough generation via constraint satisfaction with tile rewrite rules. In: *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 2023, S. 1–9

[4] Dormans, Joris: Level design as model transformation: a strategy for automated content generation. In: *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, 2011, S. 1–8

[5] Dormans, Joris ; Bakkes, Sander: Generating missions and spaces for adaptable play experiences. In: *IEEE Transactions on Computational Intelligence and AI in Games* 3 (2011), Nr. 3, S. 216–228

[6] Etchebehere, Gustavo S. ; Eliseo, Maria A.: L-Systems and Procedural Generation of Virtual Game Maze Sceneries. In: *Proc. SBGames* (2017)

[7] Felde, Niclas zum: *Implementierung von MarkovJunior in Java*

[8] Furnas, George W.: New graphical reasoning models for understanding graphical interfaces. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1991, S. 71–78

[9] Gabrovšek, Peter: Analysis of maze generating algorithms. In: *IPSI Transactions on Internet Research* 15 (2019), Nr. 1, S. 23–30

[10] Gumin, Maxim: *ConvChain*. 2016. – URL https://github.com/mxgmn/ConvChain. – Zugriffsdatum: 20.07.2024

[11] Gumin, Maxim: *WaveFunctionCollapse*. 2016. – URL https://github.com/mxgmn/WaveFunctionCollapse. – Zugriffsdatum: 20.07.2024

[12] Gumin, Maxim: *MarkovJunior*. 2022. – URL https://github.com/mxgmn/MarkovJunior. – Zugriffsdatum: 20.07.2024

[13] Karth, Isaac ; Smith, Adam M.: WaveFunctionCollapse is constraint solving in the wild. In: *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017, S. 1–10

[14] Katzenelson, Jacob: The Markov algorithm as a language parser—Linear bounds. In: *Journal of Computer and System Sciences* 6 (1972), Nr. 5, S. 465–478

[15] Kim, Paul H. ; Grove, Jacob ; Wurster, Skylar ; Crawfis, Roger: Design-centric maze generation. In: *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 2019, S. 1–9

[16] Kozlova, Aliona ; Brown, Joseph A. ; Reading, Elizabeth: Examination of representational expression in maze generation algorithms. In: *2015 IEEE Conference on Computational Intelligence and Games (CIG)* IEEE (Veranst.), 2015, S. 532–533

[17] McClendon, Michael S.: The complexity and difficulty of a maze. In: *Bridges: Mathematical connections in art, music, and science*, 2001, S. 213–222

[18] Meent, Jan-Willem van de ; Paige, Brooks ; Yang, Hongseok ; Wood, Frank: An introduction to probabilistic programming. In: *arXiv preprint arXiv:1809.10756* (2018)

[19] Merrell, Paul: Example-based procedural modeling using graph grammars. In: *ACM Transactions on Graphics (TOG)* 42 (2023), Nr. 4, S. 1–16

[20] Peachey, Liam: Parameterized maze generation algorithm for specific difficulty maze generation. In: *Association for Computing Machinery* 1 (2022), Nr. 1

[21] Pullen, Walter D.: *Think Labyrinth: Maze Algorithms*. – URL http://www.astrolog.org/labyrnth/algrithm.htm. – Zugriffsdatum: 20.07.2024

[22] Rozen, Riemer van ; Heijn, Quinten: Measuring quality of grammars for procedural level generation. In: *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 2018, S. 1–8

[23] Shah, Ms Shivani H. ; Mohite, Ms Jagruti M. ; Musale, Anoop G. ; Borade, Jay L.: Survey paper on maze generation algorithms for puzzle solving games. In: *International Journal of Scientific & Engineering Research* 8 (2017), Nr. 2, S. 1064–1067

[24] Shaker, Noor ; Togelius, Julian ; Nelson, Mark J. ; Nelson, Mark J. ; Smith, Adam M.: ASP with applications to mazes and levels. In: *Procedural Content Generation in Games* (2016), S. 143–157

[25] Smith, Gillian: An Analog History of Procedural Content Generation. In: *FDG* Boston, MA (Veranst.), 2015

[26] Togelius, Julian ; Kastbjerg, Emil ; Schedl, David ; Yannakakis, Georgios N.: What is procedural content generation? Mario on the borderline. In: *Proceedings of the 2nd international workshop on procedural content generation in games*, 2011, S. 1–6

[27] Togelius, Julian ; Yannakakis, Georgios N. ; Stanley, Kenneth O. ; Browne, Cameron: Search-based procedural content generation: A taxonomy and survey. In: *IEEE Transactions on Computational Intelligence and AI in Games* 3 (2011), Nr. 3, S. 172–186

## Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen 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 |
|---|---|---|