Graph Wave Function Collapse
Bachelorarbeit

Florian Drux

Inhaltsverzeichnis

Kapitel 1

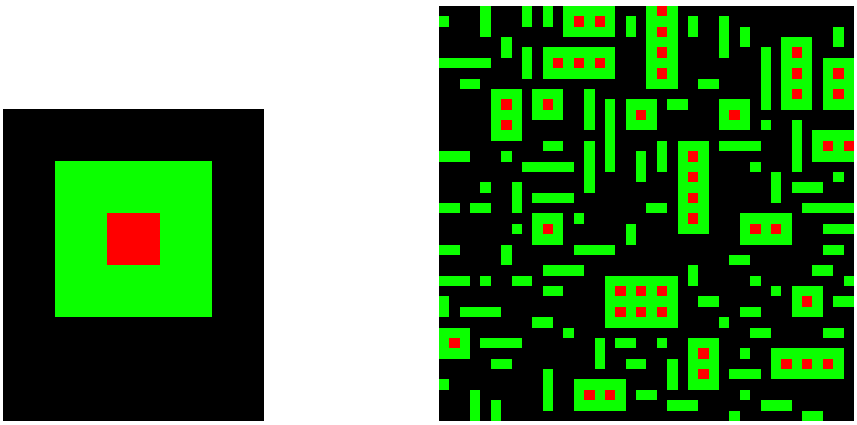
Einleitung

Videospiele und Animationsfilme haben einiges gemeinsam. Eine der Gemeinsamkeiten ist, das manche eine riesige Anzahl an 2D-/3D-Modellen und Texturen benötigen. Über die Jahre wurden diese immer mehr, immer detaillierter und in Folge dessen auch immer zeit- und kostenintensiver.

Hier hilft PCG (**P**rocedural **C**ontent **G**eneration[1]); Es muss nicht jeder Fels und Baum im Wald von Hand gesetzt werden. Wir lassen einfach ein Programm die Elemente erstellen und falls es uns nicht gefällt, so ändern wir ein paar Einstellungen und alles wird in Sekunden neu generiert. Dies eröffnet auch weitere Möglichkeiten, wie bei jedem Start eines Videospiels ein neues Level zu präsentieren.

Mit Hilfe von Constraint Solving lassen sich sich Ausgaben erstellen, welche eine Menge von Restriktionen (Constraints) erfüllen. Eine Form Constraint Solving ist Answer Set Programming, wo mit Hilfe von logischen Schlüssen Ausgaben generiert werden, wie beispielsweise Level [10]. Eine weitere verwendete Methode ist Constraint Propagation, bei welcher die Menge der Werte, die eine Variable annehmen kann, aufgrund von Restriktionen reduziert wird. Dies kann eine Kettenreaktion zur Folge haben, sodass auch andere Variablen Werte nicht mehr annehmen können. Dieses Verfahren wurde unter anderem dazu verwendet, Elemente in einem Level zu platzieren [7] oder im 1-dimensionalem Fall auf Basis von N-Grammen Level zu generieren [4].

[1] z. dt. prozedurale Synthese von Inhalten



(a) (b)

Abbildung 1.1: **1.1a ist eine WaveFunctionCollapse-Eingabe mit** 6×5 **Pixeln. Aus dieser wird die** 40×40 **Ausgabe 1.1b erstellt. Dabei ist jedes** 2×2 **Teilbild der Ausgabe auch in der Eingabe vorhanden.**

Im ersten Teil von Paul C. Merrell's Dissertation *Model Synthesis* ([9]) wird eine Methode zur prozeduralen Generierung von diskreten 3D-Modellen mit Constraint Propagation vorgestellt. Es wird ein kleines 3-dimensionale Feld als Beispieleingabe, welche die Restriktionen beschreibt, verwendet, um ein großes 3-dimensionales Ausgabefeld zu erstellen.

Auf dieser Methode basiert WaveFunctionCollapse, ein Algorithmus zur Synthese von 2-dimensionalen Texturen und Bildern aus Beispieleingaben [6]. Der Algorithmus WaveFunctionCollapse betrachtet ein Eingabebild und generiert ein Ausgabebild. Dabei muss in der Ausgabe jedes Teilbild der Maße $N \times N$ in der Eingabe vorhanden sein. Je öfter ein Teilbild dabei in der Eingabe vorkommt, desto wahrscheinlicher wird es von WaveFunctionCollapse gewählt, um in die Ausgabe eingefügt zu werden. Abbildung 1.1a ist eine Beispieleingabe, Abbildung 1.1b eine dazugehörige Ausgabe.

Nachdem WaveFunctionCollapse veröffentlicht wurde, wurde der Algorithmus von vielen Personen aufgegriffen und erweitert, beispielsweise für dreieckige Elemente auf einer Sphäre, 2-dimensionale Animationen und Prosa [6],[8]. Es existieren nun viele verschiedene Algorithmen, welche sich alle sehr ähneln.

Ziel dieser Arbeit ist es den Algorithmus GraphWaveFunctionCollapse zu entwickeln, welcher auf WaveFunctionCollapse basiert und generisch genug ist, viele unterschiedliche Eingaben zu verarbeiten. Statt eine eigene Version schreiben zu müssen, soll unser Algorithmus entweder alleinstehend verwendbar oder in Leveleditoren, Modellierungssoftware und anderen Programmen integrierbar sein.

GraphWaveFunctionCollapse erhält als Eingabe drei Graphen. Der erste dient als Beispieleingabe und besitzt Knotenwerte. Der zweite ist die Basis der Ausgabe, den Knoten dieses Graphen werden Werte zugewiesen. Der dritte Graph wird verwendet, um mit Hilfe von Teilgraphisomorphie Knotenmengen, Bereiche genannt, in den anderen beiden Graphen auszuwählen und Knoten innerhalb der Bereiche zu identifizieren. Es werden aus der Beispieleingabe Muster entnommen, diese weisen jedem Knoten eines Bereichs einen Wert zu.
Zu Beginn kann jedem Knoten der Ausgabe jeder Wert zugewiesen werden. Am Ende wird jedem Knoten, der sich in mindestens einem Bereich befindet, genau so ein Wert zugewiesen, dass in jedem Bereich der Ausgabe ein Muster vorkommt, welches in der Eingabe vorhanden war. Hierzu wird in jeder Iteration ein Bereich mit der niedrigsten Informationsentropie größer 0 ausgewählt und für diesen ein Muster festgelegt. Die Wahrscheinlichkeit, mit der ein Muster gewählt wird, ist gleich der relativen Häufigkeit des Musters in der Eingabe. Den Knoten wird daraufhin nur noch der entsprechende Wert aus dem Muster zugewiesen. Wir merken uns die möglichen Knotenwerte und die möglichen Muster je Bereich, welche wir mit Constraint Propagation an unsere Auswahl anpassen. Sollte zu einem Bereich kein Muster mehr passen, weil dem Knoten jeweils der benötigte Wert nicht mehr zugewiesen werden kann, so wird der Algorithmus abgebrochen. Die Ausgabe von GraphWaveFunctionCollapse ist eine Wertzuweisung der Knoten, sodass jeder Ausgabebereich ein Muster der Eingabe hat.

In dieser Arbeit werden wir zuallererst (Kapitel 2) eine vereinfachte Version von WaveFunctionCollapse betrachten. Sie ist vereinfacht in dem Sinne, dass sie nicht alle Optimierungen beinhaltet. Wir müssen den Algorithmus verstehen, auf welchen wir unseren basieren, die Laufzeit ist dabei nicht von Belang.

Im Hauptteil (Kapitel 3) werden wir unseren Algorithmus GraphWaveFunctionCollapse entwickeln. Wir betrachten einzelne Elemente von WaveFunctionCollapse und übertragen sie in unseren Algorithmus, welcher statt auf Bildern auf Graphen arbeiten wird. Wir werden dabei darauf achten, dass sich GraphWaveFunctionCollapse bei Graphen, die auf Bildern basieren, möglichst ähnlich zu WaveFunctionCollapse verhält. Wir wollen sicherstellen, dass unsere Verallgemeinerung sich möglichst ähnlich zum Original verhält.

Eine Verallgemeinerung muss jedoch noch andere Eingaben verarbeiten können, welche im Original nicht möglich waren. Im Kapitel 4 werden wir mit unserem Algorithmus ein Videospiellevel kreieren und eine Weltkugel mit Kontinenten, Inseln und Städten bestücken.

In [6] und [8] werden einige Abwandlungen von WaveFunctionCollapse genannt, welche GraphWaveFunctionCollapse mehr Anwendungsmöglichkeiten geben können. Diese (und eigene) Abwandlungen werden in Kapitel 5 kurz beschrieben.

Zuletzt werden in Kapitel 6 die Ergebnisse der Arbeit zusammengefasst. Es wird zudem auf die Nützlichkeit des Algorithmus in der Praxis eingegangen.

Kapitel 2

WaveFunctionCollapse

In diesem Kapitel beschreiben wir den WaveFunctionCollapse-Algorithmus von Maxim Gumin zur PCG von Texturen. Wir betrachten den Algorithmus jedoch nicht wie beschrieben in [6]. Gumin's Beschreibung benutzt ein Feld von sich überlappenden Mustermengen, während wir ein Feld von Farbmengen verwenden werden. Wir halten diese Betrachtungsweise nicht nur für intuitiver, sie wird uns außerdem den Übergang zu GraphWaveFunctionCollapse erleichtern. Für eine Beschreibung, welche mehr in die Details der Implementierung von Maxim Gumin geht, empfehlen wir [8].

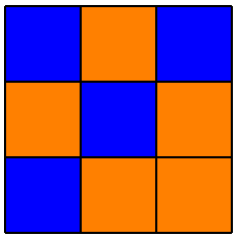
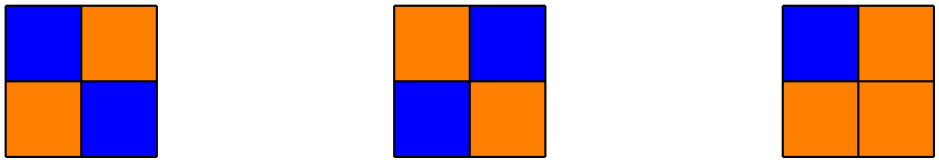


Abbildung 2.1: **Ein Eingabebild für WaveFunctionCollapse. Wir sehen 3×3 Pixel, welche je einen von 2 verschiedenen Werten annehmen.**

Daten: color$[Breite_I][Höhe_I]$ $Bild_I$, int $Breite_O$, int $Höhe_O$, int N
Ergebnis: color$[width_{out}][height_{out}]$ $Bild_O$ oder nichts

```
1  alleMuster, Anzahl ← findeMuster(Bild_I)
2  Farben ← {c ∈ Bild_I}
3  Ausgabefeld ← new 2^color[Breite_O][Höhe_O]
4  für alle e ∈ Ausgabefeld tue
5  |  e ← Farben
6  solange B ← wähleBereich(Bild_O) tue
7  |  beobachte(B, alleMuster, Anzahl)
8  |  wenn propagiere(Ausgabefeld, alleMuster) = fehlgeschlagen
   |   dann
9  |  |  zurück
10 zurück Bild_O(Ausgabefeld)
```

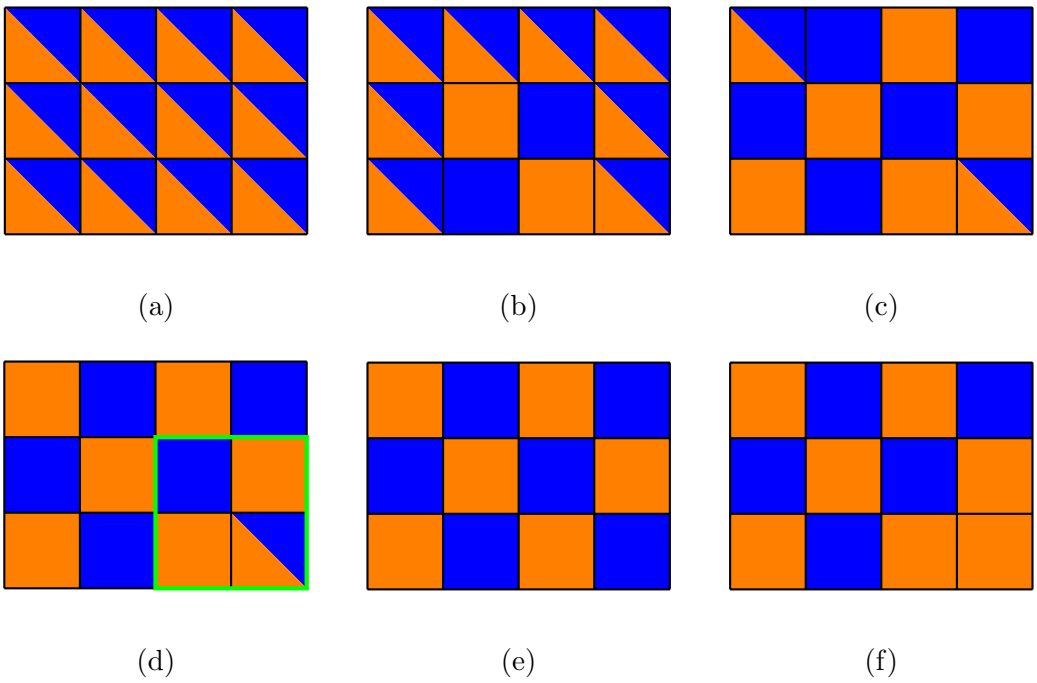
Algorithmus 1: WaveFunctionCollapse mit Farbfeld



M_1 M_2 M_3

Abbildung 2.2: **Die im Bild 2.1 enthaltenen** 2×2 **Muster. Das Muster** M_2 **ist doppelt enthalten, die anderen je 1 mal.**

WaveFunctionCollapse (Pseudocode in Algorithmus 1) erhält als Eingabe ein Bild $Bild_I$ mit Breite $Breite_I$ und Höhe $Höhe_I$, die gewünschte Breite $Breite_O$ und Höhe $Höhe_O$ des Ausgabebildes, sowie eine natürliche Zahl N. $N \times N$ ist die Größe eines lokalen Bereiches. Als Beispiel nehmen wir als $Bild_I$ das Bild 2.1, $Breite_O = 4$, $Höhe_O = 3$, $N = 2$.



(a) (b) (c)

(d) (e) (f)

Abbildung 2.3: **Wir starten mit Ausgabefeld 2.3a. In 2.3b wurde unten in der Mitte das Muster M_2 beobachtet. Eine *propagiere*-Iteration hat aus 2.3c unmögliche Zustände der Feldelemente entfernt. In 2.3d wurden von *propagiere* ein weiterer unmöglicher Zustand entnommen. Der grün eingerahmte 2×2 Bereich rechts unten befindet sich noch in den Zuständen M_1 und M_3, nicht jedoch M_2. In 2.3e wurde rechts unten M_1 beobachtet, in 2.3f M_3.**

Zu Beginn wird *findeMuster* (Z. 1) auf dem Eingabebild ausgeführt. Hier wird jeder $N \times N$ Bereich des Eingabebildes betrachtet. Wir nennen die $N \times N$ Teilbilder Muster. Muster seien gleich genau dann, wenn je die Pixel an den gleichen Positionen des $N \times N$ Gitters die gleiche Farbe haben. Wir verlangen, dass das Eingabebild mindestens $N \times N$ Pixel groß ist und wir somit mindestens ein Muster finden (sonst brechen wir mit einer Fehlermeldung ab). In unserem Beispiel finden wir 3 verschiedene Muster M_1, M_2, M_3. Diese befinden sich in Abbildung 2.2.

Es wird ein 2-dimensionales Feld zur Ausgabe generiert, im folgenden Ausgabefeld genannt. Dieses hat die Maße des zu generierenden Bildes, ist mit diesem jedoch nicht zu verwechseln; Der Unterschied ist, dass jedes Pixel eines Bildes genau eine Farbe besitzt, die Elemente unseres Ausgabefeldes befinden sich jeweils in einer „Superposition“ von Farben. Betrachten wir hierzu Ausgabefeld 1 (Abbildung 2.3a). Jedes Element unseres 3×4 Feldes besitzt nicht *eine* Farbe, sondern ist in den Zuständen blau *und* orange. Es handelt sich jeweils bei den Ausgangszuständen um die Menge der Farben des Eingabebildes.

Nun wird, solange noch Elemente im Ausgabefeld mehrere Zustände haben, ein Bereich mit der Größe $N \times N$ ausgewählt ($wähleBereich$), die Zustände in diesem Bereich auf je einen reduziert (*beobachte*) und die Zustände in der Umgebung reduziert, falls diese nun nicht mehr möglich sind (*propagiere*). Wir erläutern nun die drei genannten Funktionen, jedoch als Erstes, was mit dem ausgewählten Bereich genau geschieht (*beobachte*), um dann einfacher zu erklären, wie wir auswählen.

In *beobachte* wird ein vorher ausgewählter Bereich „beobachtet“; In der Quantenmechanik wird eine physikalische Größe (z.B. Spin eines Elektrons) durch eine Wellenfunktion beschrieben. Wenn sich die Größe in mehreren Zuständen befindet und diese gemessen/beobachtet wird, so findet ein Kollaps der Wellenfunktion statt und die Größe befindet sich nur noch in dem gemessenen Zustand. Analog hierzu beobachten wir einen Bereich unseres Feldes. Die Zustände sind die Muster, welche in den Bereich passen, d.h. die, bei denen für jede Position im Muster die entsprechende Position im Bereich vom Ausgabefeld (auch) die entsprechende Farbe besitzt. Man betrachte als Beispiel den markierten Bereich in Abbildung 2.3d. Für Muster M_1 und M_3 sind alle benötigten Farben vorhanden, in der linken Hälfte sind die Elemente nur in den benötigten Zuständen (oben blau, unten orange), auf der rechten Seite befinden sich die Elemente unter anderem in den benötigten Zuständen. Dies gilt jedoch nicht für M_2, da z.B. das Element links oben nicht im Zustand orange ist.

Der Bereich befindet sich nach dem beobachten in einem Zustand gleich einem Muster. Wie wird das Muster ausgewählt, auf welchem die Bereichszustände reduziert werden? Um ein Bild zu erhalten, was dem Eingabebild möglichst ähnlich ist, wählen wir aus den passenden Mustern eins zufällig aus. Dabei ist ein Muster umso wahrscheinlicher, welches relativ oft im Ausgangsbild vorhanden ist. Nehmen wir als Beispiel an, dass wir einen 2×2 Bereich haben, bei dem die Elemente blau und orange sind außer rechts unten, rechts unten ist nur orange. Nun passen von der Beispieleingabe (Abbildung 2.1) die Muster M_2 und M_3. Da M_2 doppelt so oft in der Eingabe vorkam, ist die Wahrscheinlichkeit doppelt so groß. Wir erhalten $P(M_1) = 0$, $P(M_2) = \frac{2}{3}$ und $P(M_3) = \frac{1}{3}$.

wähleBereich wählt einen $N \times N$ Bereich aus, welcher die geringste Informationsentropie größer 0 besitzt; Da ein Bereich ausgewählt werden soll, bei dem die Zustände reduziert werden sollen, wählen wir keinen aus, bei welchem jedes Element nur einen Zustand hat (die Entropie ist 0). Sollten wir in einem Bereich bereits die Menge der möglichen Muster reduziert haben, so ist es im allgemeinen wahrscheinlicher, dass wir eines auswählen, bei dem wir im weiteren Verlaufe ein gültiges Bild erhalten, als bei einem, bei dem die Menge der möglichen Muster noch größer ist [8]. Da die Muster gewichtet ausgewählt werden, können wir hier Bereiche bevorzugen, bei denen wir uns sicherer sind, welches Muster ausgewählt wird. Diese Bereiche haben eine geringere Informationsentropie.

Angenommen wir haben einen Bereich B, die Menge an Mustern *alleMuster*, von denen genau die Menge $passendeMuster(B) \subseteq alleMuster$ zu B passt. Jedes Muster $M \in alleMuster$ kommt $Anzahl(M) > 0$ mal im Eingabebild vor. Die Wahrscheinlichkeit, dass ein Muster M für den Bereich B ausgewählt wird ist

$$P_B(M) = \begin{cases} 0 & \text{für } M \notin passendeMuster(B) \\ \dfrac{Anzahl(M)}{\sum\limits_{M' \in passendeMuster(B)} Anzahl(M')} & \text{sonst} \end{cases}$$

Die Shannon-Entropie H(B) (mit Basis 2 für den Logarithmus) für einen Bereich B ist

$$H(B) := -\sum_{\substack{M \in \\ passendeMuster(B)}} P_B(M) \log(P_B(M))$$

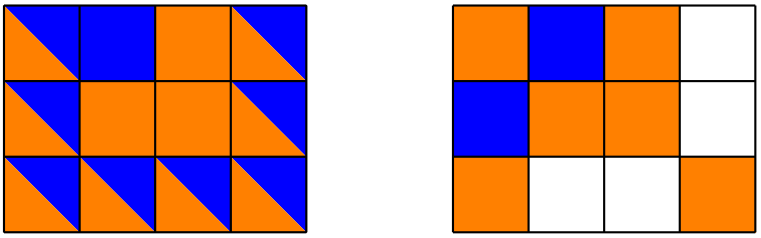
Beispiel. Angenommen wir haben drei Bereiche B_1, B_2, B_3 und unsere 3 Muster M_1, M_2, M_3 mit $Anzahl(M_1) = 1$, $Anzahl(M_2) = 2$, $Anzahl(M_3) = 1$. Es gelte $passendeMuster(B_1) = \{M_1\}$, $passendeMuster(B_2) = \{M_1, M_3\}$, $passendeMuster(B_3) = \{M_2, M_3\}$. Wir erhalten

- $H(B_1) = -(1 * \log(1)) = 0$
- $H(B_2) = -(\frac{1}{2} * \log(\frac{1}{2}) + \frac{1}{2} * \log(\frac{1}{2})) = 1$
- $H(B_3) = -(\frac{2}{3} * \log(\frac{2}{3}) + \frac{1}{3} * \log(\frac{1}{3})) \approx 0.9183$

Da $H(B_1) = 0$ ist B_1's Zustandsmenge bereits genügend reduziert.
wähleBereich wird B_3 zurückgeben, da es der am meisten, doch nicht gänzlich, bestimmte Bereich ist, der mit der geringsten Entropie größer 0.

Sollten jedoch alle Bereiche eine Entropie von 0 aufweisen, so terminieren wir. Alle Elemente sind in einem Zustand, somit haben sie genau eine Farbe, sie lassen sich nun als Pixel bezeichnen und das Ausgabefeld ist ein Ausgabebild. Es kann nicht vorkommen, dass zu einem Bereich keine Muster passen, da dieser Fall bereits von *propagiere* behandelt wird.

In *propagiere* wird die Information der Zustandsreduzierung von *beobachte* verbreitet; Nach Aufrufen von *beobachte* kann es sein, dass im Ausgabefeld Zustände vorhanden sind, zu welchen kein passendes Muster existiert. Diese werden nun entfernt. Betrachten wir das Ausgangsausgabefeld 2.3a. *wähleBereich* hat den Bereich unten in der Mitte gewählt und *beobachte* hat ihn auf Muster M_2 reduziert. Wir erhalten das Ausgabefeld 2.3b. Wenn wir uns nun den 2×2 Bereich links unten ansehen, können wir erkennen, dass nur das Muster M_1 passt. Das Element links unten z.B. ist jedoch in den Zuständen orange und blau. Da wir jedoch immer nur Zustände entfernen und nie welche hinzufügen werden, kann es nie blau werden. Wir reduzieren die Zustände in diesem Fall auf orange. Dieses Verfahren wird für alle Bereiche des Ausgabefeldes ausgeführt (wir werden für GraphWaveFunctionCollapse eine effizientere Methode vorstellen). Aus Ausgabefeld 2.3b wird Ausgabefeld 2.3c. Da wir Zustände reduziert haben kann es jedoch sein, dass nun erneut weniger Muster passen und wir weiter Zustände reduzieren können. Wir reduzieren die Zustände so lange, bis sich auf diese Weise nichts mehr reduzieren lässt und erhalten Ausgabefeld 2.3d.



(a) (b)

Abbildung 2.4: **Wir starten erneut mit Ausgabefeld 2.3a und haben dann in 2.4a oben in der Mitte das Muster M_3 beobachtet.** *propagiere* **hat aus 2.4b unmögliche Zustände der Feldelemente entfernt. Die weißen Feldelemente sind in keinem Zustand, es kann nun kein Bild mehr entstehen.**

Bei der Ausführung von *propagiere* kann es dazu kommen, dass Elemente sich in keinem Zustand mehr befinden. Man betrachte dazu Ausgabefeld 2.4a, welches mit *wähleBereich* und *beobachte* aus Ausgangsausgabefeld 2.3a entstanden ist. Wenn wir *propagiere* ausführen, so erhalten wir Ausgabefeld 2.4b in welchem sich Elemente in keinem Zustand befinden. Da Zustände nur entfernt werden, jedoch nie hinzugefügt, können diese Elemente keine Farbe mehr besitzen, somit kann aus dem Ausgabefeld kein Bild mehr werden. Da WaveFunctionCollapse kein Backtracking implementiert [8], brechen wir mit einer Fehlermeldung ab.
Die Menge der Zustände, in denen sich Elemente befinden, stellen Einschränkungen respektive Bedingungen and die Musterauswahl, ebenso schränkt die Menge aller Muster die Menge der Zustände ein, welche Elemente eines Bereiches annehmen können[1]. Mit dieser Betrachtungsweise nennen wir einen Zustand des Ausgabefeldes, in welchem sich Elemente in keinem Zustand befinden, eine Kontradiktion.

[1] Es handelt sich dabei um ein CSP (**C**onstraint-**S**atisfaction-**P**roblem)

Führen wir unser erstes Beispiel von Ausgabefeld 2.3d an fort. *propagiere* ist zu Ende ausgeführt worden, *wähleBereich* kann nun nur noch den Bereich rechts unten wählen. Sollte *beobachte* M_1 wählen, so erhalten wir Ausgabefeld 2.3e, sonst mit M_3 Ausgabefeld 2.3f. In beiden Fällen wird *propagiere* nach einer Iteration fertig sein, da es keine Zustände zu reduzieren gibt und *wähleBereich* wird feststellen, dass jeder Bereich eine Entropie von 0 aufweist. Das jeweilige Ausgabefeld wird als Bild von WaveFunctionCollapse zurückgegeben.

Kapitel 3

Implementierung mit Graphen

WaveFunctionCollapse wie in Kapitel 2 beschrieben ist beschränkt auf rechteckige Bilder als Ein- und Ausgabe. Es existieren mit WaveFunctionCollapse verwante Algorithmen, welche z.B. im 3-dimensionalen Raum arbeiten oder Dreiecke statt Quadrate als Elemente verwenden [8],[6]. Als Alternative zu spezifischen Algorithmen für Probleme bietet sich jedoch ein allgemeiner Algorithmus an.

Wir werden in diesem Kapitel für GraphWaveFunctionCollapse den ursprünglichen Algorithmus WaveFunctionCollapse mit Graphen rekonstruieren. Wir behalten dabei die grobe Struktur des originalen Algorithmus bei.

Zuallererst (Kapitel 3.1) werden wir Datentypen und -strukturen vorstellen, welche wir in unserer (Pseudocode-)Implementierung verwenden werden. Daraufhin wird in Kapitel 3.2 die Eigenschaften der Graphen erläutert, welche wir als Eingabe, Ausgabe und für die Muster verwenden werden. Wir werden die einzelnen Funktionen von WaveFunctionCollapse rekonstruieren und beginnen mit *findeMuster* in Kapitel 3.3 in welcher wir Muster aus der Beispieleingabe entnehmen werden. In Kapitel 3.4 wird *bestimmeEntropie* vorgestellt, welche die Entropie von Isomorphismen des Ausgabegraphen bestimmt. Daraufhin werden in *initialisiere* (Kapitel 3.5) alle benötigten Mengen und Tabellen initialisiert. Diese benötigen wir, um in jeder Iteration

mit *wähleIso* (Kapitel 3.6) einen Isomorphismus im Ausgabegraphen auszuwählen. Dessen Menge von verwendbaren Mustern wird in *beobachte* (Kapitel 3.7) auf ein Muster reduziert, woraufhin die resultierenden Restriktionen in *propagiere* (Kapitel 3.8) durch den Ausgabegraphen propagiert werden. Zuletzt (Kapitel 3.9) werden die erstellten Funktionen zum Algorithmus GraphWaveFunctionCollapse zusammengefügt.

3.1 Datentypen, Datenstrukturen

Es werden Datentypen und -strukturen mit ihren Eigenschaften aufgelistet, welche wir für GraphWaveFunctionCollapse verwenden können. Genauere Beschreibungen finden sich z.B. in [3]. Es handelt sich dabei nur um Möglichkeiten, es lassen sich natürlich auch andere, passende Datenstrukturen benutzen. Wir werden jedoch für die Laufzeitanalyse von GraphWaveFunctionCollapse die Eigenschaften der hier genannten Strukturen verwenden.

- *int* bezeichnet im Pseudocode den Datentyp für ganze Zahlen.
- *float* hingegen steht für reelle Zahlen.
- *bool* speichert einen Wahrheitswert und nimmt entweder den Wert *wahr* oder *falsch* an.
- *Knoten* ist der Datentyp um einen Knoten zu repräsentieren.

Um kommende Laufzeitanalysen zu vereinfachen, nehmen wir eine konstante Größe der obigen Datentypen an, insbesondere weil wir annehmen, dass konstant große Datentypen (z.B. ein 128-Bit *int*) für alle praktikablen Eingaben genügen.

Tabellen speichern Schlüssel-Wert-Paare. Wir werden sie verwenden, um Funktionen zu repräsentieren. Wenn für eine Tabelle H $H[k]$ mit einem Schlüssel k aufgerufen werden, so wird der entsprechende Wert des Paares zurückgegeben. Mit $H[k] \leftarrow x$ wird der Eintrag mit Schlüssel k und Wert x erstellt, wobei ein eventuell existierender Eintrag mit dem gleichen Schlüssel verworfen wird. $H.entf(k)$ löscht ein Paar mit entsprechendem Schlüssel, sollte es existieren. Wir verwenden Hashtabellen. Für Schlüssel mit konstanter Länge benötigt das Aufrufen, Speichern, Überschreiben und Löschen von Einträgen jeweils

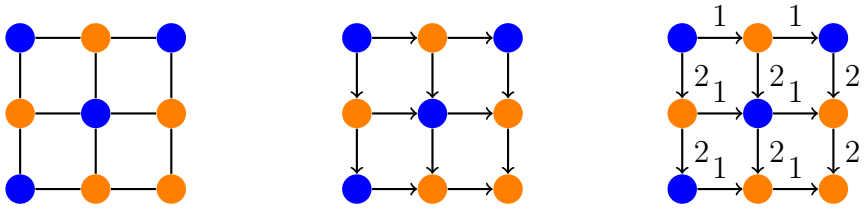
durchschnittlich $\mathcal{O}(1)$ [3]. Wir benötigen für den Test $k \in \mathit{Schlüssel}(H)$ nicht die Menge aller Schlüssel $\mathit{Schlüssel}(H)$, und brauchen analog zum Aufrufen $\mathcal{O}(1)$. Wenn wir uns bei jedem Schlüssel den Hashwert merken, benötigen wir bei jedem Schlüssel je $\mathcal{O}(1)$. Wir gehen davon aus, dass für ein Objekt der Größe n die Laufzeitkomplexität der Hashfunktion $\mathcal{O}(n)$ beträgt. Insbesondere wird das einmalige Berechnen der Hashwerte die Laufzeit nicht mehr als um einen konstanten Faktor erhöhen, da wir über die Schlüssel, welche aus mehreren Elementen zusammengesetzt sind (z.B. Muster), im Algorithmus iterieren.
Die Tabellengröße sei in $\Theta(n)$ bei n Einträgen, wobei Tabellen mit neuen Einträgen wachsen können [3]. Dies erlaubt es uns die unterliegende Tabelle (s. [3]) in $\Theta(n)$ durchzugehen. Dies ist die Komplexität um über die Menge aller Schlüssel $\mathit{Schlüssel}(H)$ zu iterieren.
Alternativ würden sich auch Suchbäume anbieten, welche jedoch für einen Aufruf von $H[k]$ eine Komplexität von $\mathcal{O}(\log(n))$ haben [3].

Mengen speichern Schlüssel, jeder Schlüssel kommt nur einmal in einer Menge vor. Entsprechend lässt sich als unterliegende Datenstruktur eine Tabelle (ohne Werte) verwenden. Da wir für eine Menge M nicht mit $M[k] \leftarrow x$ ein Paar hinzufügen, schreiben wir hier $M.\mathit{einf}(k)$ um nur einen Schlüssel hinzuzufügen. Die Laufzeit- sowie Speicherkomplexität sind bis auf einen konstanten Faktor identisch mit denen einer Tabelle für Schlüssel-Wert-Paare.

Da wir ständig mit Tabellen und Mengen arbeiten werden, sind die Laufzeiten in der Regel als durchschnittliche Laufzeiten zu verstehen. Zum Zwecke der Leserlichkeit sehen wir davon ab, es bei jeder der Laufzeitanalysen zu erwähnen.

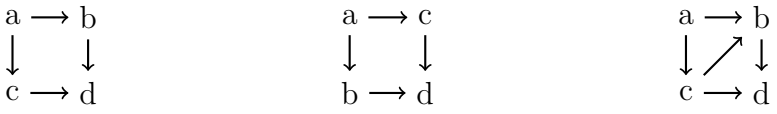
3.2 Graphen

Um WaveFunctionCollapse auf Graphen zu simulieren, sollten wir die wichtigsten Elemente als Graphen darstellen; Die Eingabe, die Muster und das Ausgabefeld. Betrachten wir zuerst die ersten beiden Fälle; Es handelt sich um Bilder, 2-dimensionale Felder dessen Elemente sich je in einem Farbzustand befinden. Als Beispiel verwenden wir Abbildung 2.1.



(a) (b) (c)

Abbildung 3.1: **Das Eingabebild 2.1 wird als Graph G_I dargestellt. Im ungerichteten Graphen 3.1a lässt sich links nicht von rechts unterscheiden. Dies ist im gerichteten 3.1b möglich, jedoch nicht links von oben (s. Abbildung 3.2). In 3.1c lässt sich jeder Knoten genau einem Feldelement zuordnen.**



(a) (b) (c)

Abbildung 3.2: **Bei 3.2a und 3.2b handelt es sich um die gleichen Graphen, wir können nicht zwischen den Knoten b und c unterscheiden. Die zusätzliche Kante (c, b) in 3.2c verhindert diese Uneindeutigkeit.**

Wir können nun jedes Element als einen Knoten in einem Graphen darstellen. Es stellt sich dann die Frage, woher wir wissen, welcher Knoten welches Element repräsentiert. Wir benutzen zu diesem Zweck die Kanten des Graphen um die Struktur der Eingabe zu reflektieren.

Als ersten Versuch verbinden wir jeden Knoten mir den direkten Nachbarn. Betrachten wir Abbildung 3.1a, so lässt sich erkennen, dass wir die Knoten nicht eindeutig einem Feldelement zuordnen können, da wir z.B. nicht zwischen den Richtungen links und rechts unterscheiden können und damit nicht wissen, welcher Knoten das Element links oben repräsentiert.

(a) (b) M_1 (c) M_2 (d) M_3

Abbildung 3.3: **Wir sehen 4 mal einen Graphen G_L, wobei die rechten 3 je ein Muster aus G_I 3.1c haben. Man vergleiche diese mit den Mustern aus Abbildung 2.2.**

Wenn wir einen gerichteten Graph verwenden (Abbildung 3.1b), bei dem die Richtungen der Kanten von links nach rechts beziehungsweise von oben nach unten geht, so lassen sich mehr, doch nicht allen, Knoten Elemente eindeutig zuordnen; Abbildung 3.2 lässt erkennen, dass uns noch der Unterschied zwischen links und oben sowie zwischen rechts und unten fehlt, da zwei unterschiedliche Felder zu dem selben Graphen (Abbildungen 3.2a, 3.2b) führen können. Es existiert zwar die Möglichkeit, diagonale Kanten hinzuzufügen (Abbildung 3.2c), sodass wir eine Bijektion zwischen Bildern und Graphen haben, diese Lösung ist jedoch speziell für (2-)dimensionale Felder.

Mit Hinblick darauf, dass GraphWaveFunctionCollapse auch Eingaben mit anderer Struktur annehmen können soll und eine analoge Lösung für jede neue Eingabestruktur zu finden mühselig sein kann, sollte eine Lösung überhaupt existiert, werden wir die Kanten einfach markieren. In Abbildung 3.1c werden die Kanten von links nach rechts mit einer 1 markiert, die von oben nach unten mit einer 2. Elemente und Knoten lassen sich nun eindeutig zuordnen.

Die Elemente der Felder waren in Farbzuständen. Da es keinen Grund gibt, sich auf Farben zu beschränken, werden wir als Zustände Elemente einer minimalen Menge D_V (**D**omäne) verwenden. Sie ist insoweit minimal, als dass jeder Wert in der Eingabe verwendet wird.

Statt eines Eingabebildes haben wir nun einen Graphen $G_I = (V_I, E_I)$ (**I**nput). G_I ist gerichtet, es gilt $E_I \subseteq V_I \times V_I$. Die Knoten sind mit Werten und die Kanten mit Label markiert durch die Funktionen $Z_I : V_I \mapsto D_V$ (**Z**ustand) und $L_I : E_I \mapsto D_E$ (**L**abel) für zwei nicht leere Mengen D_V, D_E. Aus dem Eingabebild 2.1 wird der Graph 3.1c.

Muster sind in WaveFunctionCollapse ebenso Bilder und würden sich analog zum Eingabebild als Graphen darstellen lassen. Muster hatten jedoch immer die Größe $N \times N$. Entsprechend sind die dazugehörigen Graphen isomorph inklusive der Kantenwerte und exklusive der Knotenzustände. Wir benötigen somit insgesamt nur einen *zusammenhängenden*[1] Graphen $G_L = (V_L, E_L)$ (**L**okaler Bereich) und nur eine Funktion $L_L : E_L \mapsto D_E$, für unser Beispiel haben wir den Graphen 3.3a. Jedes einzelne Muster M ist dann eine eigene Funktion $M : V_L \mapsto D_V$. Die drei Muster sind in Abbildung 3.3. Wir hätten auch G_L mit L_L und M zusammengenommen als Muster bezeichnen können, statt nur M, jedoch werden L_L und G_L, bis auf G_L's Knotenmenge V_L, nach finden aller Muster M und Isomorphismen nicht weiter benötigt. Auch halten wir diese Formulierung für intuitiver, da wir eher sagen würden „Der Graph *hat* ein Muster." als „Der Graph *ist* ein Muster."

Im Ausgabefeld können einzelne Elemente mehrere Farbzustände haben. Wir benutzen statt des Feldes einen gerichteten Graphen $G_O = (V_O, E_O)$ (**O**utput). Für die Kanten verwenden wir die analoge Funktion $L_O : E_O \mapsto D_E$. Da jede Farbe im Ausgabefeld im Eingabebild vorhanden sein muss, verwenden wir für die Markierung der Knoten die Funktionen $Z_O : V_O \mapsto 2^{D_V}$ (wobei $2^{D_V} := \{S | S \subseteq D_V\}$), welche auf Untermengen der Wertemenge D_V abbildet. Ein Knoten $v \in V_O$ befindet sich in $|Z_O(v)|$ vielen Zuständen. Da Knoten sich am Ende nur in einem Zustand befinden können, welcher auch in der Eingabe vorkam, können Knotenwerte mit $alleWerte := \{Z_I(v) | v \in V_I\}$ initialisiert werden.

3.3 Muster finden

Um in WaveFunctionCollapse die Mustermenge *alleMuster* zu bestimmen, haben wir jeden $N \times N$ Bereich betrachtet. Wir müssen in GraphWaveFunctionCollapse bestimmen, welche Knoten im Eingabegraphen G_I jeweils zu einem Bereich gehören, ohne vom Graphen auf das dazugehörige 2-dimensionale Feld zu schließen. Zudem müssen wir wissen, wo sich der Knoten innerhalb des Bereiches befindet.

[1] Nicht zusammenhängende Graphen wären möglich, weisen jedoch meist eine unpraktikable Anzahl an Isomorphismen mit G_I und dem noch nicht erläuterten G_O auf.

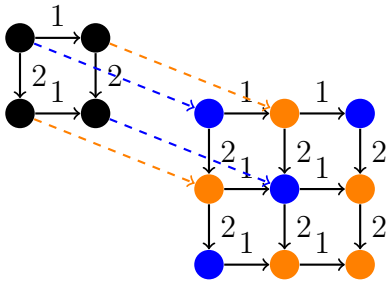


Abbildung 3.4: **Der Graph mit den 4 schwarzen Knoten ist G_L, der mit den 9 bunten Knoten G_I. Die gestrichelten Pfeile sind ein Isomorphismus von G_L zu einem Teilgraphen von G_I, durch welchen wir das Muster M_1 (3.3b) erhalten.**

Wir erhalten aus dem Eingabebild 2.1 mit $N = 2$ die Muster in Abbildung 2.2. Analog müssen wir aus dem Graphen G_I 3.1c die Muster M_1 (3.3b), M_2 (3.3c) und M_3 (3.3d) bestimmen.

Jedes Muster von WaveFunctionCollapse ist ein $N \times N$ Teilbild des Eingabebildes. Wenn wir in GraphWaveFunctionCollapse die Muster M_1 (3.3b), M_2 (3.3c) und M_3 (3.3d) betrachten, so sind diese Teilgraphen des Graphen G_I 3.1c. Wir kennen den Graphen G_L und die Kantenwerte L_L in unserem Beispiel (Abbildung 3.4), da wir wissen, dass sie ein $N \times N$ Feld repräsentieren.

Als Teil der Eingabe für GraphWaveFunctionCollapse haben wir einen Graphen G_L zuzüglich L_L. Dieser beschreibt Bereiche im Graphen G_I, die Knotenmenge jedes Teilgraphen von G_I isomorph zu G_L mit Einbezug der Kantenwerte ist ein Bereich. Wir benutzen nicht den eigentlichen Teilgraphen, da wir nur auf den Knoten arbeiten werden und die Kanten lediglich benötigten, um auf die *Bereichsisomorphismen* zu kommen.

Betrachten wir die Graphen G_L und G_I. Ein Bereichsisomorphismus *Iso* ist ein Teilgraphisomorphismus. Es handelt sich um eine injektive Funktion $Iso : V_L \to V_I$ welche jedem Knoten aus G_L einen Knoten aus G_I zuordnet, wobei keine zwei Knoten in G_L existieren, denen der selbe Knoten zugeordnet wird. Auch gilt, dass jede Kante aus G_L auch in G_I bei den entsprechenden, von *Iso* zugewiesenen Knoten mit dem gleichen Label zu finden ist;

$(u,v) \in E_L \implies (Iso(u), Iso(v)) \in E_I \wedge L_L((u,v)) = L_I((Iso(u), Iso(v)))$
Für die Bereiche im Graphen G_O werden Funktionen $Iso\colon V_L \to V_O$ verwendet. Sie verhalten sich analog, außer dass sie Knoten aus G_O statt G_I zurückgeben. Wenn wir einen Isomorphismus Iso haben, ist der dazugehörige Bereich $B(Iso) := \{Iso(v) | v \in V_L\}$ das Bild der Funktion. Abbildung 3.4 ist ein Beispiel für einen Teilgraphisomorphismus.

Sei $TeilGraphIsos(G_1, L_1, G_2, L_2)$ die Menge der Teilgraphisomorphismen, bei denen der Graph G_2 mit Kantenlabeln L_2 isomorph zu Teilgraphen von G_1 mit Labeln L_1 ist.

Um alle Muster zu bestimmen, müssen wir zuerst
$Isos_I := TeilGraphIsos(G_I, L_I, G_L, L_L)$ bestimmen. Da wir in unserem Beispiel den Aufbau von G_L und G_I kennen, kennen wir auch die Isomorphismen; Für $v \in V_I$ überprüfe ob ausgehende Kanten mit Werten 1 und 2 existieren, diese führen zu 2 Knoten welche dann mit je einer Kante zu einem gemeinsamen Knoten führen. Auf diese oder ähnliche Art und Weise lässt sich $Isos_I$ bestimmen.

Falls wir jedoch als Eingabe Graphen haben, deren Teilgraphisomorpismen unbekannt sind, so müssen sie erst gefunden werden. Als eine Möglichkeit sei der Algorithmus VF2 erwähnt, welcher in [2] erläutert wird. VF2 kann unter anderem die Menge der Isomorphismen zweier Graphen bestimmen, wobei es möglich ist, auf die Äquivalenz von Kantenlabeln zu achten [2]. Die Laufzeit beträgt $\Theta(|V|^2)$ im besten und $\Theta(|V| * |V|!)$ im schlechtesten Fall, der Speicherplatzverbrauch beträgt immer $\Theta(|V|)$ [1].
Alternativ lassen sich natürlich auch andere Algorithmen verwenden.

Nach dem wir auf eine von uns gewählte Art $Isos_I$ bestimmt haben, benötigen wir für jeden Bereichsisomorphimus $Iso \in Isos_I$ das zugehörige Muster M. Iso verweist jeden Knoten $v \in V_L$ auf den Knoten $Iso(v) \in V_I$. M lässt sich entsprechend definieren mit $M(v) := Z_I(Iso(v))$ für $v \in V_L$. In unserem Beispiel 3.4 ergibt sich das Muster M_1 (3.3b).

Zusammenfassend ist die Menge aller Muster
$alleMuster := \{M | \exists Iso \in Isos_I.\ \forall v \in V_L.\ M(v) = Z_I(Iso(v))\}$
und die Anzahl eines Musters M
$Anzahl(M) := |\{Iso \in Isos_I | \forall v \in V_L.\ M(v) = Z_I(Iso(v))\}|$
Die Menge aller Werte, welche wir betrachten ist
$alleWerte := \{Z_I(v) | v \in V_I\}$

```
1  Funktion findeMuster
       Daten: Menge V_L, Tabelle Z_I, Menge Isos_I
       Ergebnis: Menge alleMuster, Tabelle Anzahl
2      Menge alleMuster ← ∅
3      Tabelle Anzahl ← neu(Hashtabelle)
4      für alle Iso ∈ Isos_I tue
5          Tabelle M ← neu(Hashtabelle)
6          für alle v ∈ V_L tue
7              M[v] ← Z_I[Iso[v]]
8          alleMuster.einf(M)
9          wenn M ∉ Schlüssel(Anzahl) dann
10             Anzahl[M] ← 0
11         Anzahl[M] ← Anzahl[M] + 1
12     zurück alleMuster, Anzahl
```

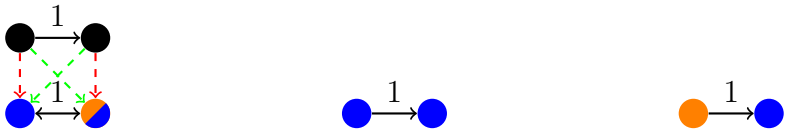
Algorithmus 2: *findeMuster*

Algorithmus 2 ist eine Pseudocodefunktion *findeMuster*. Sie erhält die Knotenmenge V_L sowie die Knotenwerte Z_I des Graphen G_I und die bereits berechneten Isomorphismen $Isos_I$. Wir iterieren über alle $Iso \in Isos_I$ (Z. 4). Für je ein Iso erstellen wir eine Tabelle M (Z. 5) welche ein Muster repräsentiert. In Z. 7 werden in M die Funktionswerte gespeichert. Des weiteren wird M zur Menge *alleMuster* hinzugefügt (Z. 8) und in *Anzahl* eingetragen, wie oft M bis jetzt gefunden worden ist (Z. 9 ff.).

Wir iterieren über $Isos_I$ und über V_L. Es ergibt sich eine Laufzeitkomplexität von $\mathcal{O}(|Isos_I| * |V_L|)$.

3.4 Entropie bestimmen

In WaveFunctionCollapse werden Bereiche (Teilfelder) anhand ihrer Entropie ausgewählt und beobachtet (s. Kapitel 2). In GraphWaveFunctionCollapse werden wir analog Bereichsisomorphismen auswählen.



(a) (b) (c)

Abbildung 3.5: **In 3.5a sehen wir G_L mit 2 schwarzen Knoten und G_O mit 2 bunten Knoten. Es gibt 2 Isomorphismen, einer grün, der andere rot. Für den roten passt nur das Muster 3.5b, für den grünen passt zusätzlich 3.5c. Somit haben die Isomorphismen des gleichen Bereichs eine unterschiedliche Entropie.**

Man könnte meinen, das wir Bereiche $B \subseteq V_O$ auswählen, jedoch würde sich ein Problem ergeben. Bei den Teilfelder war eindeutig, wie ein Muster (Bild) auf einen Bereich (Teilfeld) des Ausgabefeldes angewandt wird; das Element links oben im Bereich befindet sich nach dem Beobachten nur noch im Zustand des Elements im Muster links oben.
Wenn wir in GraphWaveFunctionCollapse für den Bereich des Isomorphismus 3.4 betrachten (die vier Knoten, auf welche die Pfeile zeigen), so lassen sich analog die Muster anwenden. Aber Abbildung 3.5 zeigt, das für ein und denselben Bereich mehrere Isomorphismen existieren können. Zudem können diese Isomorphismen unterschiedliche Mengen an passenden Mustern haben.

Ein Muster M ist genau dann zu einem Isomorphismus Iso passend, wenn für jeden Knoten $v \in V_L$ sich der zugehörige Knoten $v' = Iso(v) \in V_O$ (unter anderem oder ausschließlich) im Zustand $M(v)$ befindet. Somit ist die Menge der passenden Muster $passendeMuster(Iso)$ abhängig von den Zuständen der Knoten in V_O. Dies ist erneut analog zu WaveFunctionCollapse (s. Kapitel 2). Formal definieren wir
$passendeMuster(Iso) := \{M \in alleMuster | \forall v \in V_L.\ M(v) \in Z_O(Iso(v))\}$

Würden wir die Entropie für Bereiche bestimmen wollen, so erschwert es die Berechnungen. Betrachten wir erneut Abbildung 3.5, so haben wir den Bereich $B = V_O$ mit zwei Isomorphismen $Iso_{grün}$, Iso_{rot} (3.5a). Zur Auswahl stehen die Muster M_1 (3.5b), M_2 (3.5c). Für die Mengen der passenden Muster haben wir $passendeMuster(Iso_{grün}) = \{M_1, M_2\}$, $passendeMuster(Iso_{rot}) = \{M_1\}$. Die Entropie eines Bereiches wäre wie in WaveFunctionCollapse abhängig von der Anzahl der Zustände, in denen sich der Bereich nach Anwendung eines Musters befinden kann.

Wir haben nun zwei Möglichkeiten. Entweder erlauben wir alle Muster für alle Isomorphismen oder wir wählen nur solche Muster aus, dass wir für jeden Isomorphismus ein Muster haben. Wir entscheiden uns für die zweite Möglichkeit, da sie uns erlaubt, die Menge der passenden Muster in ausgewählten Bereichen durch das Hinzufügen von Kanten gezielt zu reduzieren. Von einer semantischen Perspektive aus gilt, dass Isomorphismen Restriktionen sind.

Die Entropie eines Bereichs ist demnach abhängig davon, welche Muster verschiedener Isomorphismen zueinander kompatibel sind. Da nur kompatible Muster verwendet werden können, können wir $passendeMuster(Iso_{grün})$ auf $\{M_1\}$ reduzieren, wodurch wir jeden Knoten in B auf den Zustand blau reduzieren können. Dieses Vorgehen werden wir noch in *propagiere* sehen. Wenn wir jedoch nur kompatible Muster verwenden, existiert für jedes Muster für einen Isomorphismus je ein kompatibles Muster in jedem anderem Isomorphismus des selben Bereichs. Insbesondere ist die Anzahl der Zustände, in denen sich nach dem Anwenden eines Musters ein Bereich befinden kann gleich der Anzahl der kompatiblen Muster eines Isomorphismus. Weil wir uns in *propagiere* merken werden, welche Muster passen, können wir direkt die Entropie eines Isomorphismus verwenden. Auch werden wir ohnehin einen Isomorphismus $Iso : V_L \to V_O$ benötigen, um ein Muster $M : V_L \mapsto D_V$ auf Knoten von V_O anwenden zu können (man betrachte die Definitionsmengen der Funktionen).
In dieser Hinsicht werden wir direkt einzelne Isomorphismen auswählen, und somit deren Entropien bestimmen.

Es ergibt sich dadurch, das wir die Entropien der Isomorphismen statt der Bereiche betrachten ein Nebeneffekt. Für zwei Bereiche bei denen die Isomorphismen die gleiche Entropie besitzen gilt, dass wenn unter ihnen zufällig ausgewählt wird, mit höherer Wahrscheinlichkeit ein Isomorphismus in einem Bereich gewählt wird, welcher viele Isomorphismen besitzt. Dies führt dazu, das wir die Bereiche bei der Auswahl präferieren, welche mehr Kanten, somit mehr Restriktionen, haben und eher in einer Kontradiktion beteiligt sind.

Analog zur Entropieberechnung aus WaveFunctionCollapse (s. Kapitel 2) ist die Wahrscheinlichkeit $P_{Iso}(M)$ für ein Muster M und einen Isomorphismus Iso

$$P_{Iso}(M) = \begin{cases} 0 & \text{für } M \notin passendeMuster(Iso) \\ \dfrac{Anzahl(M)}{\sum\limits_{M' \in passendeMuster(Iso)} Anzahl(M')} & \text{sonst} \end{cases}$$

Die Entropie H eines Isomorphismus Iso ist

$$H(Iso) := -\sum_{\substack{M \in \\ passendeMuster(Iso)}} P_{Iso}(M) \log(P_{Iso}(M))$$

```
1 Funktion bestimmeEntropie
  |  Daten: Tabelle Z_Isos, Anzahl, Iso
  |  Ergebnis: float Entropie
2 |  Tabelle P_Iso ← bestimmeWahrscheinlichkeit(Z_Isos, Anzahl, Iso)
3 |  float Entropie ← 0
4 |  für alle M ∈ Z_Isos[Iso] tue
5 |  |  Entropie ← Entropie − P_Iso[M] * log(P_Iso[M])
6 |  zurück Entropie
```

Algorithmus 3: *bestimmeEntropie*

Algorithmus 3 enthält Pseudocode um die Entropie eines Isomorphismus Iso zu bestimmen. Dafür erhält *bestimmeEntropie* neben Iso noch die Tabelle Z_{Isos} (**Z**ustand), welche für jeden Iso eine Menge der zu Iso passenden Muster zurückgibt, ähnlich zu *passendeMuster* (Z_{Isos} wird in Kapitel 3.5 genauer erläutert). Auch erhält *bestimmeEntropie* die Tabelle *Anzahl* welche Muster auf die Anzahl abbildet, wie oft sie in G_I mit Z_I vorkamen.

Um die Entropie zu bestimmen benötigen, wir P_{Iso}. Hierfür übergeben wir der Funktion *bestimmeWahrscheinlichkeit* (Algorithmus 4) Z_{Isos}, *Anzahl* und Iso. Wir iterieren erst über $Z_{Isos}[Iso]$ (Z. 3 f.) um
$AnzahlPassend := \sum\limits_{M' \in Z_{Isos}(Iso)} Anzahl(M')$
zu berechnen. Danach iterieren wir erneut über $Z_{Isos}[Iso]$ (Z. 6 ff.) um mit Hilfe von *AnzahlPassend* P_{Iso} zu bestimmen. Es ist direkt ersichtlich, dass *bestimmeWahrscheinlichkeit* eine Laufzeitkomplexität von $\mathcal{O}(|Z_{Isos}(Iso)|)$ hat.

```
1 Funktion bestimmeWahrscheinlichkeit
  |  Daten: Tabelle Z_Isos, Anzahl, Iso
  |  Ergebnis: Tabelle P_Iso
2 |  int AnzahlPassend ← 0
3 |  für alle M ∈ Z_Isos[Iso] tue
4 |  |  AnzahlPassend ← AnzahlPassend + Anzahl[M]
5 |  Tabelle P_Iso ← neu(Hashtabelle)
6 |  für alle M ∈ Z_Isos[Iso] tue
7 |  |  P_Iso[M] ← Anzahl[M] / AnzahlPassend
8 |  zurück P_Iso
```

Algorithmus 4: *bestimmeWahrscheinlichkeit*

Beispiel. Wir haben die Muster M_1 (3.3b), M_2 (3.3c) und M_3 (3.3d), welche alle zum Isomorphismus *Iso* passen, wobei $Anzahl(M_1) = 1$, $Anzahl(M_2) = 2$, $Anzahl(M_3) = 1$. Wir erhalten die Wahrscheinlichkeiten $P_{Iso}(M_1) = \frac{1}{1+2+1} = 0{,}25$, $P_{Iso}(M_2) = \frac{2}{1+2+1} = 0{,}5$, $P_{Iso}(M_3) = \frac{1}{1+2+1} = 0{,}25$. Die Entropie ist
$H(Iso) = -0{,}25 * \log(0{,}25) - 0{,}5 * \log(0{,}5) - 0{,}25 * \log(0{,}25) = 1{,}5$.

In Algorithmus 3 Zeile 4 f. wird P_{Iso} verwendet um die Entropie nach der oben beschriebenen Formel zu bestimmen. Auch *bestimmeEntropie* hat eine Laufzeitkomplexität von $\mathcal{O}(|Z_{Isos}(Iso)|)$, was wir mit $\mathcal{O}(|alleMuster|)$ abschätzen.

3.5 Initialisieren

Bevor in WaveFunctionCollapse (Algorithmus 1) Bereiche ausgewählt und bearbeitet werden, werden erst die Muster gesucht und das Ausgabefeld initialisiert. Wir werden in GraphWaveFunctionCollapse ähnlich vorgehen. Es wird ein Tupel verwendet, welches alle Informationen enthält, um Isomorphismen auszuwählen und zu bearbeiten. Wir nennen es einen *GWFC-Zustand* (**G**raph**W**ave**F**unction**C**ollapse). Das Ergebnis einer solchen Iteration wird erneut ein Tupel der gleichen Form sein, in der Initialisierung werden wir es erstellen.

Ein GWFC-Zustand sei ein Tupel der Form
$(V_O, Z_O, V_L, Isos, alleMuster, Anzahl, Z_{Isos}, Ent, Isos_V)$.
Wir kennen bereits alle Elemente außer Z_{Isos}, Ent und $Isos_V$;

- V_O ist die Knotenmenge unseres Graphen G_O, welcher mit Kanten eine Semantik zwischen den Knoten beschreibt.
- $Z_O : V_O \mapsto 2^{D_V}$ ist eine Funktion, welche die Zustandsmengen für die Knoten aus V_O angibt. Diese ändern sich im Laufe des Algorithmus.
- V_L ist die Knotenmenge des Graphen G_L, mit welchem wir die Bereichsisomorphismen bestimmen. Wir werden im Algorithmus des Öfteren über V_L iterieren.
- $Isos = TeilGraphIsos(G_O, L_O, G_L, L_L)$ ist die Menge der Isomorphismen von G_L zu einem Teilgraphen von G_O.
- *alleMuster* ist die Menge der Muster, welche mit $Isos_I = TeilGraphIsos(G_I, L_I, G_L, L_L)$ und Z_I in G_I gefunden werden.
- *Anzahl* bildet ein Muster auf die Anzahl ab, wie oft es in G_I gefunden werden kann.

Dies alles genügt bereits, um in GraphWaveFunctionCollapse zu iterieren und die Zustandsmengen der Knoten zu reduzieren. Z_{Isos}, Ent und $Isos_V$ dienen dazu, redundante Informationen zu speichern, damit wir diese nicht ständig neu berechnen müssen.

$Z_{Isos} : Isos \mapsto 2^{alleMuster}$ ist eine Funktion analog zu Z_O. In Z_O speichern wir die Zustände, in denen sich die Knoten aus V_O jeweils befinden. In Z_{Isos} speichern wir die Zustände der Isomorphismen. Es sind die Muster, welche zu einem Isomorphismus passen, was das Neuberechnen von $passendeMuster(Iso)$ für den jeweiligen $Iso \in Isos$ verhindert.

$Ent : Isos \mapsto \mathbb{R}$ ist eine zu Z_{Isos} analoge Funktion, in ihr werden die Entropien der Isomorphismen gespeichert.

$Isos_V : V_O \mapsto 2^{Isos \times V_L}, v \to \{(Iso, v') \in Isos \times V_L | v = Iso(v')\}$ ist eine Funktion, welche für jeden Knoten v des Ausgabegraphen G_O die Menge von Isomorphismen zurückgibt, in deren Bereich sich v befindet. Zusätzlich wird mit jedem Isomorphismus Iso $v' \in V_L$ angegeben, wobei v' genau der Knoten ist, welcher von Iso auf v abgebildet wird. Es existiert für jeden Iso nur ein solches v', da Isomorphismen injektiv sind.

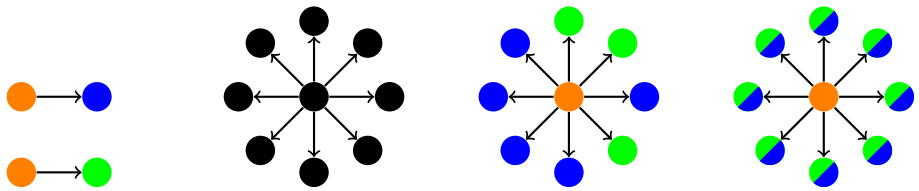
Wir *speichern* in Z_O, Z_{Isos} und *Ent*, indem wir beim Aufrufen der Funktion keinen Rückgabewert berechnen, sondern die Funktionen mit Konstanten definieren. Für jede Eingabe existiert eine Konstante für die Ausgabe und wenn wir neue Werte speichern wollen, so werden die Funktionen durch neue Funktionen mit anderen Konstanten ersetzt. Die Funktionen sind somit Tabellen, aus welcher die Werte nur ausgelesen werden, das Speichern gleicht dem Überschreiben eines Eintrags.

Anhand der Elemente des GWFC-Zustand-Tupels wissen wir auch, welche Eingaben wir für GraphWaveFunctionCollapse benötigen;

- V_O wird in der Eingabe sein.
- Z_O bildet zu Beginn alle Knoten in V_O auf $alleWerte = \{Z_I(v)|v \in V_I\}$ ab, von daher werden Z_I und V_I benötigt.
- V_L benötigen wir ebenfalls.
- *Isos* wird bestimmt aus den Graphen G_O, G_L sowie den Kantenlabeln L_O, L_L.
- *alleMuster* benötigt ebenfalls G_L, L_L, aber auch G_I mit L_I und Z_I.
- *Anzahl* benötigt alles das, was auch *alleMuster* benötigt.

Folglich besteht die Eingabe aus den Graphen $G_I = (V_I, E_I)$, $G_O = (V_O, E_O)$, $G_L = (V_L, E_L)$ inklusive der Kantenlabel L_I, L_O, L_L und den Zuständen Z_I der Knoten aus V_I.

Wir müssen aus der Eingabe den initialen GWFC-Zustand bestimmen. Wie oben geschrieben, definieren wir initial
$Z_O : V_O \mapsto 2^{D_V}, v \to \{Z_I(v')|v' \in V_I\}$.
Dann bestimmen wir
$Isos = TeilGraphIsos(G_O, L_O, G_L, L_L)$,
$Isos_I = TeilGraphIsos(G_I, L_I, G_L, L_L)$,
$Isos_V : V_O \mapsto 2^{Isos \times V_L}, v \to \{(Iso, v') \in Isos \times V_L | v = Iso(v')\}$.
Wie in Kapitel 3.3 beschrieben ist
$alleMuster = \{M|\exists Iso \in Isos_I.\ \forall v \in V_L.\ M(v) = Z_I(Iso(v))\}$



(a) (b) (c) (d)

Abbildung 3.6: **In 3.6a ist 2 mal G_L mit je einem Muster, es existieren keine weiteren. In 3.6b ist G_O und in 3.6c eine mögliche Ausgabe. Es ist direkt anhand der Muster ersichtlich, dass der mittlere Knoten nur orange und die äußeren nur blau oder grün sein können. In 3.6d ist ein zum initialen Z_{Isos} passendes Z_O zu sehen.**

und wir speichern die Anzahl für jedes Muster $M \in alleMuster$
$Anzahl(M) = |\{Iso \in Isos_I | \forall v \in V_L.\ M(v) = Z_I(Iso(v))\}|$.
Zuletzt seien initial
$Z_{Isos} : Isos \mapsto 2^{alleMuster}, Iso \to alleMuster$
$Ent : Isos \mapsto float, Iso \to H(Iso)$
Damit haben wir den initialen GWFC-Zustand bestimmt.

Wir können initial in Z_{Isos} alle Isomorphismen auf $alleMuster$ abbilden, da alle Knoten in allen Zuständen sind und somit jedes Muster passt.

Satz 1. *Initial gilt für alle $Iso \in Isos$, dass $passendeMuster(Iso) = alleMuster$.*

Beweis. Sei $Iso \in Isos$, dann gilt per Definition von $passendeMuster$
$passendeMuster(Iso) \subseteq alleMuster$
Desweiteren gilt initial
$passendeMuster(Iso) \overset{def.\ passendeMuster}{=}$
$\{M \in alleMuster | \forall v \in V_L.\ M(v) \in Z_O(Iso(v))\} \overset{def.\ Z_O}{=}$
$\{M \in alleMuster | \forall v \in V_L.\ M(v) \in \{Z_I(v') | v' \in V_I\}\}$
Aus der Definition von $alleMuster$ folgt
$\forall M \in alleMuster.\ \exists Iso' \in Isos_I.\ \forall v \in V_L.\ M(v) = Z_I(Iso'(v))$
Da Iso' auf V_I abbildet, folgt
$\forall M \in alleMuster.\ \forall v \in V_L.\ M(v) \in \{Z_I(v') | v' \in V_I\}$
$\Rightarrow \forall M \in alleMuster.\ M \in passendeMuster(Iso)$
$\Rightarrow passendeMuster(Iso) = alleMuster$ □

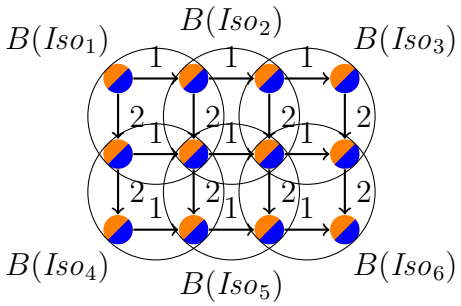


Abbildung 3.7: **Unser initialer G_O inklusive Z_O. Die Kanten von links nach rechts haben das Label 1, die von oben nach unten das Label 2. Mit G_L 3.3a erhalten wir 6 Isomorphismen Iso_1, ..., Iso_6. Die Bereiche $B(Iso_1)$ bis $B(Iso_3)$ sind von links nach rechts in der oberen Hälfte eingekreist, die restlichen in der unteren.**

	Iso_1	Iso_2	Iso_3
Z_{Isos}	$\{M_1, M_2, M_3\}$	$\{M_1, M_2, M_3\}$	$\{M_1, M_2, M_3\}$
Ent	1,5	1,5	1,5
	Iso_4	Iso_5	Iso_6
Z_{Isos}	$\{M_1, M_2, M_3\}$	$\{M_1, M_2, M_3\}$	$\{M_1, M_2, M_3\}$
Ent	1,5	1,5	1,5

Tabelle 3.1: **Die initialen Z_{Isos} und Ent für die Isomorphismen Iso_1, ..., Iso_6 des G_O in Abbildung 3.7.**

Nun kann es sein, dass manche Knoten aus V_O aufgrund der Muster nicht in allen Zuständen sein können. Ein Beispiel findet sich in Abbildung 3.6. Wir wollen Z_O und Z_{Isos} soweit reduzieren, dass in Z_O nur Zustände vorkommen, welche nach Z_{Isos} möglich sind. Z_O *passt* genau dann zu Z_{Isos}, wenn
$\forall v \in V_O.\ Z_O(v) \subseteq \{w \in D_V | \forall Iso \in Isos.\ \exists M \in Z_{Isos}(Iso).\ \exists v' \in V_L.$
$Iso(v') = v \wedge M(v') = w\}$.
Wenn wir Z_O entsprechend reduziert haben, passen eventuell manche Muster nicht mehr zu einem Isomorphismus, da die entsprechenden Knoten nicht mehr in den benötigten Zuständen sind. Z_{Isos} *passt* genau dann zu Z_O, wenn $\forall Iso \in Isos.\ Z_{Isos}(Iso) \subseteq passendeMuster(Iso)$.
Damit Z_O und Z_{Isos} zueinander passen (und Ent zu Z_{Isos}), wird *propagiere* mit V_O aufgerufen. *propagiere* wird in Kapitel 3.8 erläutert.

Beispiel. Wir haben ein G_I in Abbildung 3.1c und G_L in Abbildung 3.3a. Desweiteren haben wir bereits die Muster M_1 (3.3b), M_2 (3.3c) und M_3 (3.3d) in Kapitel 3.3 gefunden. Unser G_O wird analog zu G_I und G_L auf einem 2-dimensionalen Feld basieren, welches wie Ausgabefeld 2.3a die Maße 3×4 besitzt, und ist in Abbildung 3.7. Die Knoten sind in den Zuständen blau und orange. In der Tabelle 3.1 sind die Zustände und Entropien der 6 Isomorphismen zu finden. An den Knoten- und Isomorphismenzuständen ändert *propagiere* nichts und auch die Entropien bleiben entsprechend unberührt.

```
1  Funktion initialisiere
       Daten: Graph G_I, G_O, G_L, Tabelle L_I, L_O, L_L, Z_I
       Ergebnis: Menge V_O, Tabelle Z_O, Menge V_L, Isos, alleMuster,
                 Tabelle Anzahl, Z_Isos, Ent, Isos_V
2      Menge Isos_I ← TeilGraphIsos(G_I, L_I, G_L, L_L)
3      alleMuster, Anzahl ← findeMuster(V_L, Z_I, Isos_I)
4      Menge Isos ← TeilGraphIsos(G_O, L_O, G_L, L_L)
5      Menge alleWerte ← ∅
6      für alle v ∈ V_I tue
7          alleWerte.einf(Z_I[v])
8      Tabelle Z_O ← neu(Hashtabelle)
9      für alle v ∈ V_O tue
10         Z_O[v] ← alleWerte
11     Tabelle Z_Isos ← neu(Hashtabelle)
12     Tabelle Isos_V ← neu(Hashtabelle)
13     für alle v ∈ V_O tue
14         Isos_V[v] ← ∅
15     Tabelle Ent ← neu(Hashtabelle)
16     float Entropie ←
         bestimmeEntropie(alleMuster, Anzahl, zufälligesElement(Isos))
17     für alle Iso ∈ Isos tue
18         Z_Isos[Iso] ← alleMuster
19         Ent[Iso] ← Entropie
20         für alle v' ∈ V_L tue
21             Isos_V[Iso[v']].einf((Iso, v'))
22     propagiere(Z_O, Z_Isos, Anzahl, Ent, Isos_V, V_O)
23     zurück V_O, Z_O, V_L, Isos, alleMuster, Anzahl, Z_Isos, Ent, Isos_V
```

Algorithmus 5: *initialisiere*

Algorithmus 5 ist eine Implementierung der Initialisierung in Pseudocode. In der Rückgabe sind Redundanzen, es ist $alleMuster = Schlüssel(Anzahl)$, $V_L = Schlüssel(M)$ für ein beliebiges $M \in alleMuster$ und $V_O = Schlüssel(Z_O)$, wir geben sie im Pseudocode jedoch zurück, damit wir uns im Rahmen dieser Arbeit nicht merken müssen, welche Mengen Schlüssel welcher Tabelle sind. In einer „richtigen“ Implementierung wäre dies zu vermeiden.

In Zeile 2 bestimmen wir die Menge $Isos_I$, welche wir benötigen, um in *findeMuster* die Muster zu finden und zu zählen. *findeMuster* hat eine Laufzeitkomplexität von $\mathcal{O}(|Isos_I|*|V_L|)$ (s. Kapitel 3.3). In Zeile 4 bestimmen wir $Isos$. Daraufhin initialisieren wir der Reihe nach $alleWerte$ (Z. 5 ff.) ($\mathcal{O}(|V_I|)$), Z_O (Z. 8 ff.) ($\mathcal{O}(|V_O| * |alleWerte|)$). Dann Z_{Isos} ($\mathcal{O}(|Isos| * |alleMuster|)$), Ent ($\mathcal{O}(|alleMuster| + |Isos|)$) und $Isos_V$ ($\mathcal{O}(|Isos| * |V_L|)$) (Z. 11 ff.).

Dabei iterieren wir über V_I, V_O respektive $Isos$. Beim Eintragen der Werte in die Tabellen werden diese kopiert, weshalb wir die Größe der kopierten Objekte berücksichtigen mussten. Wenn wir der Einfachheit halber annehmen, das wir je mindestens einen Isomorphismus finden, ergibt sich eine Laufzeit von $\mathcal{O}(|Isos_I|*|V_L|+|V_I|+|Isos|*|V_L|+|V_O|*|alleWerte|+|Isos|*|alleMuster|)$ zuzüglich dem Berechnen der Isomorphismen. Dies kann außerhalb von Polynomialzeit liegen.

Betrachten wir den Graphen $G_n := (\{1, 2, \ldots, n\}, \emptyset)$ für ein $n \in \mathbb{N}$. Da G_n keine Kanten besitzt ist jede Permutation über $\{1, 2, \ldots, n\}$ ein Isomorphismus von G_n zu G_n. Es existieren $n!$ Permutationen, weshalb nicht alle in Polynomialzeit aufgezählt werden können.

Dies zeigt, dass *initialisiere* und somit auch der gesamte Algorithmus GraphWaveFunctionCollapse nur praktikabel sind, sollte die Anzahl der Isomorphismen „gering“ ausfallen. Hierbei ist „gering“ vom Anwendungsfall abhängig, sie sollten jedoch für größere Graphen G_I, G_O nicht $\mathcal{O}(|V_I|)$ beziehungsweise $\mathcal{O}(|V_O|)$ überschreiten.

3.6 Isomorphismus wählen

Nachdem wir einen initialen GWFC-Zustand bestimmt haben (Kapitel 3.5), iterieren wir wie in WaveFunctionCollapse (Kapitel 2) und senken die Zustandsmengen Z_O der Ausgabeknoten V_O. Wie in Kapitel 3.4 beschrieben wählen wir zu Beginn einer Iteration einen Isomorphismus aus, dessen Zustandsmenge auf ein Muster reduziert wird.

In Kapitel 3.5 haben wir die Entropien der Isomorphismen in der Funktion *Ent* gespeichert. Wir werden im weiteren Verlaufe des Algorithmus dafür Sorge tragen, dass *Ent* beim Aufrufen von *wähleIso* die aktuellen Entropien hat. Ebenso werden bei jedem Aufruf Z_O und Z_{Isos} zueinander passen.

wähleIso ist eine Funktion, welche einen Isomorphismus zurückgibt, sodass die Entropie eine der geringsten größer 0 ist. Dies ist eine der am meisten, doch nicht gänzlich bestimmten Isomorphismen. Sollten alle Entropien 0 sein, so können wir für keinen Isomorphismus die Zustandsmenge reduzieren und stoppen GraphWaveFunctionCollapse, da wir fertig sind. Weil Z_O und Z_{Isos} zueinander passen, und *propagiere* den Algorithmus abbricht, sollte ein Knoten oder Isomorphismus sich in keinem Zustand befinden, haben alle Knoten je genau einen Zustand.

Wir bestimmen zuerst die Menge
$minIsos := \{Iso \in Isos | Ent(Iso) > 0 \wedge$
$\neg \exists Iso' \in Isos.\ 0 < Ent(Iso') < Ent(Iso)\}$,
wobei *Ent* die Werte zurückgibt, welche wir mit H bestimmt haben. Sollte ein Element in *minIsos* existieren, so geben wir ein zufälliges Element zurück, ansonsten beenden wir den Algorithmus.

Als Beispiel betrachten wir die Tabelle 3.1. Alle Isomorphismen haben die gleiche Entropie, welche größer als 0 ist, demnach ist $minIsos = Isos = \{Iso_1, \ldots, Iso_6\}$. Wir wählen Iso_2.

Algorithmus 6 ist eine Pseudocodeimplementierung von *wähleIso*. Wir iterieren über alle Isomorphismen $Iso \in Isos$ (Z. 4), dabei merken wir uns die geringste Entropie größer 0 *minEntropie*, sowie die Menge *minIsos*, welche die bereits betrachteten Isomorphismen mit $Ent(Iso) = minEntropie$ enthält. Sollten wir ein $Iso \in Isos$ mit $0 < Ent(Iso) < minEntropie$ finden (Z. 6), so enthält *minIsos* keine der gewünschten Isomorphismen und wir setzten *minIsos* wieder auf eine leere Menge (Z. 8) und *minEntropie* auf die geringere Entropie (Z. 7). Wenn wir diese neue Menge mit konstanter Größe kreieren und sie im Verlaufe wachsen lassen (s. Kapitel 3.1), so lässt sich dies mit einer konstanten Laufzeitkomplexität erledigen. So bestimmen wir die Menge *minIsos* wie oben definiert.

minIsos ist genau dann leer, sollte *minEntropie* nie eine Entropie < 0 gefunden haben, dann können wir nichts zurückgeben (Z. 11 f.). Sollte *minIsos* nicht leer sein, so geben wir ein zufälliges Element zurück. Beispielsweise können wir die Anzahl der Elemente $|minIsos|$ zählen, eine zufällige ganze Zahl $0 < x \leq |minIsos|$, kreieren und den x'ten Eintrag in *minIsos* zurückgeben. Mit $minIsos \subseteq Isos$ folgt, dass dies in $\mathcal{O}(|Isos|)$ geht.

Es ergibt sich für *wähleIso* eine Laufzeitkomplexität von $\mathcal{O}(|Isos|)$.

```
1  Funktion wähleIso
       Daten: Menge Isos, Tabelle Ent
       Ergebnis: Tabelle Iso oder nichts
2      Menge minIsos ← ∅
3      float minEntropie ← ∞
4      für alle Iso ∈ Isos  tue
5          wenn Ent[Iso] > 0 dann
6              wenn Ent[Iso] < minEntropie dann
7                  minEntropie ← Ent[Iso]
8                  minIsos ← ∅
9              wenn Ent[Iso] = minEntropie dann
10                 minIsos.einf(Iso)
11     wenn minEntropie = ∞ dann
12         zurück
13     Iso ← zufälligesElement(minIsos)
14     zurück Iso
```

Algorithmus 6: *wähleIso*

3.7 Isomorphismus beobachten

Nach dem wir einen Isomorphismus gewählt haben, wird dieser beobachtet, er befindet sich dann nur noch in einem Zustand. Da die Entropie eines gewählten Isomorphismus *Iso* größer als 0 ist, wissen wir, das wir aus mehreren Mustern auswählen können ($|Z_{Isos}(Iso)| > 1$). Wir wählen Muster mit unterschiedlichen Wahrscheinlichkeiten; Je öfter ein Muster, welches zum Isomorphismus passt, in der Eingabe vorkommt, desto wahrscheinlicher wird es gewählt.

Nachdem für ein Iso mit $H(Iso) > 0$ ein Muster $M \in Z_{Isos}$ mit Wahrscheinlichkeit $P_{Iso}(M)$ gewählt worden ist, wird Z_{Isos} ersetzt durch ${Z_{Isos}}'$, sodass für $Iso' \in Isos$

$${Z_{Isos}}'(Iso') = \begin{cases} \{M\} & \text{für } Iso' = Iso \\ Z_{Isos}(Iso') & \text{sonst} \end{cases}$$

Nun ist Iso in nur einem Zustand. Entsprechend tauschen wir Ent mit Ent' aus, wobei für $Iso' \in Isos$

$$Ent'(Iso') = \begin{cases} 0 & \text{für } Iso' = Iso \\ Ent(Iso') & \text{sonst} \end{cases}$$

Insgesamt wird der GWFC-Zustand
$(V_O, Z_O, V_L, Isos, alleMuster, Anzahl, Z_{Isos}, Ent, Isos_V)$ ersetzt mit
$(V_O, Z_O, V_L, Isos, alleMuster, Anzahl, {Z_{Isos}}', Ent', Isos_V)$.

Im Bereich $B(Iso) = \{Iso(v) | v \in V_L\}$ sind die Knoten jetzt in mehr Zuständen als sie nach dem neuen Z_{Isos} sein können. Z_O passt nicht mehr zu Z_{Isos}. Hierum wird sich in *propagiere* gekümmert (Kapitel 3.8).

Beispiel. Wir betrachten erneut Tabelle 3.1. In Kapitel 3.6 haben wir Iso_2 gewählt. Nach derselben Tabelle kann das Muster aus $Z_{Isos}(Iso_2) = \{M_1, M_2, M_3\} = Isos$ gewählt werden. Hierbei sind die Wahrscheinlichkeiten $P_{Iso_2}(M_1) = 0{,}25$, $P_{Iso_2}(M_2) = 0{,}5$, $P_{Iso_2}(M_3) = 0{,}25$. M_3 (3.3d) wird gewählt und wir erhalten die Tabelle 3.2. Dabei bleiben Knotenzustände noch wie in Abbildung 3.7.

	Iso_1	Iso_2	Iso_3
Z_{Isos}	$\{M_1, M_2, M_3\}$	$\{M_3\}$	$\{M_1, M_2, M_3\}$
Ent	1,5	0	1,5
	Iso_4	Iso_5	Iso_6
Z_{Isos}	$\{M_1, M_2, M_3\}$	$\{M_1, M_2, M_3\}$	$\{M_1, M_2, M_3\}$
Ent	1,5	1,5	1,5

Tabelle 3.2: **Die Isomorphismenzustände Z_{Isos} und Entropien Ent, nachdem in Tabelle 3.1 in Iso_2 der Zustand M_3 beobachtet worden ist.**

1 **Funktion** *beobachte*
  Daten: *Tabelle Iso*, Z_{Isos}, *Anzahl*, Z_O, *Ent*
  Ergebnis: *Tabelle* Z_{Isos}, *Ent*
2 *Tabelle* $P_{Iso} \leftarrow bestimmeWahrscheinlichkeit(Z_{Isos}, Anzahl, Iso)$
3 $M \leftarrow zufälligesElement(Z_{Isos}[Iso], P_{Iso})$
4 $Z_{Isos}[Iso] \leftarrow \{M\}$
5 $Ent[Iso] \leftarrow 0$
6 **zurück** Z_{Isos}, *Ent*

Algorithmus 7: *beobachte*

In Algorithmus 7 ist der Pseudocode der Funktion *beobachte*. Wie in Kapitel 3.4 benötigen wir P_{Iso} und verwenden dafür *bestimmeWahrscheinlichkeit* (Algorithmus 4). Wir wählen dann ein Muster M mit Wahrscheinlichkeit $P_{Iso}(M)$ aus und geben es zurück. Die Berechnung von P_{Iso} benötigt genauso wie das Auswählen von M durchschnittlich $\mathcal{O}(|Z_{Isos}(Iso)|)$. Damit ist die Laufzeitkomplexität von *beobachte* $\mathcal{O}(|Z_{Isos}(Iso)|)$, was wir mit $\mathcal{O}(|alleMuster|)$ abschätzen.

Wir können ein Muster mit einer bestimmten Wahrscheinlichkeitsverteilung P_{Iso} in $\mathcal{O}(|Z_{Isos}(Iso)|)$ auswählen, indem wir zuerst eine zufällige Fließkommazahl $0 \leq x < 1$ erstellen. Dann werden die Muster in einer willkürlichen doch festen Reihenfolge durchgegangen. Bei $Z_{Isos}(Iso) = \{M_1, M_2, \ldots, M_n\}$ wird M_i gewählt, wenn
$\sum_{j=0}^{i-1} Anzahl(M_j) \leq x < \sum_{j=0}^{i} Anzahl(M_j)$.
Dies ist in Abbildung 3.8 illustriert.

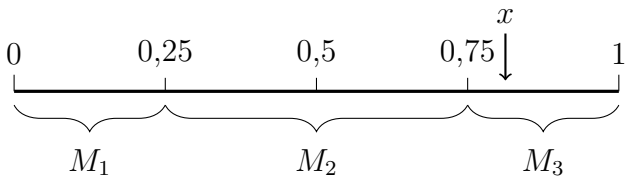


Abbildung 3.8: **Drei Muster mit den Wahrscheinlichkeiten** $P_{Iso_2}(M_1) = 0{,}25$**,** $P_{Iso_2}(M_2) = 0{,}5$**,** $P_{Iso_2}(M_3) = 0{,}25$ **erhalten von** $[0, 1)$ **die Bereiche** $M_1 : [0, 0{,}25)$**,** $M_2 : [0{,}25, 0{,}75)$**,** $M_3 : [0{,}75, 1)$**. Mit der Zufallszahl** $x = 0,8125 \in [0{,}75, 1)$ **beobachten wir** M_3**.**

3.8 Propagieren

In diesem Kapitel wird zuerst erläutert, wie beim propagieren der Zustandsreduktionen die Knotenzustände angepasst werden. Danach werden wir betrachten, wie die Zustände der Isomorphismen angepasst werden, um zum Schluss beides in der Funktion *propagiere* zusammenzufügen.

3.8.1 Knoten

Nachdem wir in Kapitel 3.7 einen Isomorphismus beobachtet haben, sind die Knoten in dessen Bereich noch in zu vielen Zuständen. Z_O muss an Z_{Isos} angepasst werden. Dies gilt auch, wenn initial bereits aus Z_{Isos} folgt, dass manche Knoten nicht in allen Zuständen sein können. Aus diesem Grund müssen wir auch in *initialisiere* (Kapitel 3.5 Algorithmus 5) Z_O behandeln. Die zu bearbeitende Knotenmenge bezeichnen wir mit V_b.

Wie bereits in Kapitel 3.5 definiert, passt Z_O zu Z_{Isos} genau dann, wenn
$\forall v \in V_O.\ Z_O(v) \subseteq \{z \in D_V | \forall Iso \in Isos.\ \exists M \in Z_{Isos}(Iso).\ \exists v' \in V_L.$
$Iso(v') = v \wedge M(v') = z\}$.
Analog passt die Zustandsmenge eines Knoten $v \in V_O$ genau dann zu Z_{Isos}, wenn
$Z_O(v) \subseteq \{z \in D_V | \forall Iso \in Isos.\ \exists M \in Z_{Isos}(Iso).\ \exists v' \in V_L.$
$Iso(v') = v \wedge M(v') = z\}$.
Wir haben in $Isos_V$ für jeden Knoten $v \in V_O$ alle Isomorphismen Iso und dazugehörigen Knoten $v' \in V_L$ gespeichert, sodass $Iso(v') = v$. Folglich passt die Zustandsmenge eines Knotens v genau dann zu Z_{Isos}, wenn

$Z_O(v) \subseteq \{z \in D_V | \forall (Iso, v') \in Isos_V(v).\ \exists M \in Z_{Isos}(Iso).\ M(v') = z\}$,
was äquivalent ist zu

$$Z_O(v) \subseteq \bigcap_{(Iso,v') \in Isos_V(v)} \{M(v') | M \in Z_{Isos}(Iso)\}$$

In anderen Worten betrachten wir für jeden Isomorphismus Iso, in welchem sich unser Knoten v befindet, die Menge der Zustände, welche mit den Mustern des Isomorphismus $Z_{Isos}(Iso)$ v zugeordnet werden können.

Wir werden für alle Knoten $v \in V_b$ die Menge der Zustände neu evaluieren. Wir bestimmen $Z_O{}'$ mit

$$Z_O{}'(v) = \begin{cases} \bigcap_{(Iso,v') \in Isos_V(v)} \{M(v') | M \in Z_{Isos}(Iso)\} & \text{für } v \in V_b \\ Z_O(v) & \text{sonst} \end{cases}$$

Noch werden wir im GWFC-Zustand nicht Z_O mit $Z_O{}'$ ersetzten. Im weiteren Verlaufe wird noch die Menge der Zustände benötigt, welche in einem Knoten entfernt wurden. Die Menge der Knoten, von denen Zustände entfernt worden sind, ist $V_{entf} = \{v \in V_b | |Z_O{}'(v)| < |Z_O(v)|\}$. Die entfernten Zustände per Knoten werden gegeben durch die Funktion
$Z_O^{entf} : V_{entf} \mapsto 2^{alleWerte}, v \to \{z \in Z_O(v) | z \notin Z_O{}'(v)\}$.
Nun können wir Z_O ersetzen.

Nachdem wir die Zustände der Knoten reduziert haben und somit Z_O zu Z_{Isos} passt, kann es sein, dass für Isomorphismen nicht mehr alle Muster anwendbar sind, da die Knoten sich nicht in den notwendigen Zuständen befinden. Z_{Isos} passt nicht mehr zu Z_O.

Wie in Kapitel 3.5 definiert, passt Z_{Isos} zu Z_O genau dann, wenn $\forall Iso \in Isos.\ Z_{Isos}(Iso) \subseteq passendeMuster(Iso)$.

Bei der Menge der Isomorphismen, denen noch unpassende Muster angegeben werden, kann es sich nur um solche handeln, welche auf Knoten abbilden, deren Zustandsmenge wir reduziert haben. Somit ist die zu bearbeitende Isomorphismenmenge $Isos_b = \{Iso \in Isos | \exists v \in V_{entf}.\ (Iso, v') \in Isos_V(v)\}$, welche wir auch bestimmen.

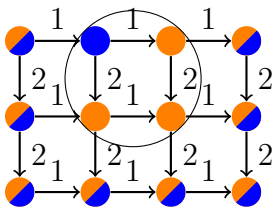


Abbildung 3.9: G_O **mit dem neuen** Z_O **nach dem Propagieren.** $V_b = B(Iso_2)$ **ist eingekreist.**

Wir führen das Beispiel von Kapitel 3.7 fort. Wir haben in Iso_2 den Zustand M_3 beobachtet. Damit ist unser $V_b = B(Iso_2)$ (man betrachte den entsprechenden Kreis in Abbildung 3.7). Alle Isomorphismen außer Iso_2 lassen noch für jeden Knoten $v \in V_b$ jeden Zustand zu, während Iso_2 nur den Zustand $M_3(v')$ für den Knoten $v \in V_b$ mit $(Iso_2, v') \in Isos_V(v)$ erlaubt. Wir erhalten das in Abbildung 3.9 dargestellte Z_O. Da sich jeder Isomorphismenbereich mit $B(Iso_2)$ überlappt (in Abbildung 3.7 überlappen sich alle Kreise mit dem von $B(Iso_2)$), ist unser $Isos_b = Isos$.

In der Funktion *propagiereKnoten* werden wir die Zustandsmengen der Knoten $v \in V_b$ auf $\bigcap_{(Iso,v') \in Isos_V} \{M(v') | M \in Z_{Isos}(Iso)\} \subseteq Z_O(v)$ minimieren. Der Pseudocode befinden sich in Algorithmus 8.

Für jeden Knoten $v \in V_b$ erledigen wir mehrere Dinge;

1. bestimme $Z_O{}'(v)$
2. gilt $Z_O{}'(v) \subsetneq Z_O(v)$?
3. passe Z_O an, erweitere Z_O^{entf} und $Isos_b$
4. teste auf eine Kontradiktion

Teil 1: Nachdem ein $v \in V_b$ gewählt wurde, merken wir uns zu Beginn (Z. 6 ff.) *alteZustände*, die Menge der Zustände, in welchen sich v noch befindet. Wir iterieren über $Isos_V(v)$. In jeder Iteration betrachten wir einen Isomorphismus Iso inklusive Knoten v' und iterieren über $Z_{Isos}(Iso)$ um die Menge $\{M(v') | M \in Z_{Isos}(Iso)\}$ durchzugehen. Insgesamt kreieren wir den Schnitt von $\bigcap_{(Iso,v') \in Isos_V} \{M(v') | M \in Z_{Isos}(Iso)\}$, die neue Zustandsmenge *neueZustände*. Hierfür benötigen wir durchschnittlich $\mathcal{O}(|Isos_V(v)| * |Z_{Isos}(Iso)|)$, was wir mit $\mathcal{O}(|Isos_V(v)| * |alleMuster|)$ abschätzen.

```
1  Funktion propagiereKnoten
       Daten: Menge V_b, Tabelle Z_O, Isos_V, Z_Isos
       Ergebnis: Tabelle Z_O, Z_O^entf
2      Tabelle Z_O^entf ← neu(Hashtabelle)
3      Menge Isos_b ← ∅
4      für alle v ∈ V_b tue
5          // Teil 1
6          Menge alteZustände ← Z_O[v]
7          Menge neueZustände ← alteZustände
8          für alle (Iso, v') ∈ Isos_V[v] tue
9              Menge neueZustände' ← ∅
10             für alle M ∈ Z_Isos[Iso] tue
11                 wenn M[v'] ∈ neueZustände dann
12                     neueZustände'.einf(M[v'])
13             neueZustände ← neueZustände'
14         // Teil 2
15         bool echteUntermenge ← falsch
16         Menge entfernteZustände ← ∅
17         für alle w ∈ alteZustände tue
18             wenn w ∉ neueZustände dann
19                 echteUntermenge ← wahr
20                 entfernteZustände.einf(w)
21         // Teil 3
22         wenn echteUntermenge dann
23             Z_O[v] ← neueZustände
24             Z_O^entf[v] ← entfernteZustände
25             Tupel (Iso, v') ← Isos_V[v]
26             Isos_b.einf(Iso)
27             // Teil 4
28             wenn Z_O[v] = ∅ dann
29                 beende(„Kontradiktion“, v)
30     zurück Z_O, Z_O^entf, Isos_b
```

Algorithmus 8: *propagiereKnoten*

Teil 2: In Z. 15 ff. bestimmen wir, ob es sich bei der neuen Zustandsmenge um eine echte Untermenge handelt. Wir iterieren dabei über $Z_O(v)$, was wir mit $\mathcal{O}(|alleWerte|)$ abschätzen.

Teil 3: Falls es eine echte Untermenge ist, so ist $v \in V_{entf}$. Wir passen Z_O an, und speichern die Menge der entfernten Zustände in der Tabelle Z_O^{entf}, bei der die Schlüssel genau die Menge V_{entf} ist.

Teil 4 (Z. 28 f.): Sollte es sich um eine echte Untermenge handeln, jedoch das neue $Z_O(v)$ leer sein, so befindet sich der Knoten in keinem Zustand. Wir brechen das Programm ab und geben an, dass eine Kontradiktion in v gefunden wurde.

Da wir das Ganze für jeden Knoten in V_b machen, schätzen wir eine durchschnittliche Laufzeitkomplexität von $\mathcal{O}(|V_b|*(max_{v \in V_b}(|Isos_V(v)|)*|alleMuster|+|alleWerte|))$ ab.

3.8.2 Isomorphismen

Für jeden Isomorphismus $Iso \in Isos_b$ bestimmen wir analog zu *propagiere-Knoten* die neue Zustandsmenge, nehmen uns die Mengen der entfernten Knotenzustände jedoch zu Hilfe. Statt jedes Muster gänzlich zu prüfen, prüfen wir nur, ob für einen Knoten der Zustand entfernt wurde, welcher für das Muster benötigt wurde. Die zu entfernende Mustermenge wird je Isomorphismus gegeben durch
$Z_{Isos}^{entf} : Isos_b \mapsto 2^{alleMuster}, Iso \to \{M \in Z_{Isos}(Iso) | \exists v \in V_{entf}.\ \exists v' \in V_L.$
$(Iso, v') \in Isos_V(v) \wedge M(v') \in Z_O^{entf}(v)\}$.
Die Menge der Isomorphismen, von welchen die Zustände reduziert worden sind, ist $Isos_{entf} = \{Iso \in Isos_b | Z_{Isos}^{entf}(Iso) \neq \emptyset\}$.
Mit $Isos_{entf}$ und Z_{Isos}^{entf} können wir dann Z_{Isos} an Z_O anpassen. Wir erstellen ein neues $Z_{Isos}{}'$ mit

$$Z_{Isos}{}'(Iso) = \begin{cases} \{M \in Z_{Isos}(Iso) | M \notin Z_{Isos}^{entf}(Iso)\} & \text{für } Iso \in Isos_{entf} \\ Z_{Isos}(Iso) & \text{sonst} \end{cases}$$

Wir ersetzen Z_{Isos} mit $Z_{Isos}{}'$ und erhalten den Zustand
$(V_O, Z_O, V_L, Isos, alleMuster, Anzahl, Z_{Isos}{}', Ent, Isos_V)$.

	Iso_1	Iso_2	Iso_3	Iso_4	Iso_5	Iso_6
Z_{Isos}	$\{M_2\}$	$\{M_3\}$	$\emptyset$	$\{M_1\}$	$\emptyset$	$\{M_2\}$
Ent	0	0	0	0	0	0

Tabelle 3.3: **Die Isomorphismenzustände Z_{Isos} und Entropien Ent, nachdem sie an Z_O (Abbildung 3.9) angepasst worden sind.**

Da die Zustandsmengen der Isomorphismen in $Isos_{entf}$ reduziert worden sind, muss entsprechend Ent ersetzt werden durch Ent', wobei wir das neue, gerade bestimmte Z_{Isos} verwenden, sodass

$$Ent'(Iso) = \begin{cases} H(Iso) & \text{für } Iso \in Isos_{entf} \\ Ent(Iso) & \text{sonst} \end{cases}$$

Wir ersetzen nun auch Ent mit Ent' erhalten den GWFC-Zustand $(V_O, Z_O, V_L, Isos, alleMuster, Anzahl, Z_{Isos}{}', Ent', Isos_V)$.

Da wir die Zustandsmengen der Isomorphismen in $Isos_{entf}$ reduziert haben, kann es in den dazugehörigen Bereichen sein, dass Knoten nicht mehr in den Zuständen sein können, in denen sie bis jetzt waren. Wir kreieren ein neues V_b, welches eine Vereinigung aller Bereiche $B(Iso)$ mit $Iso \in Isos_{entf}$ darstellt; $V_b = \bigcup_{Iso \in Isos_{entf}} B(Iso)$.

Es kann vorkommen, dass wir alle Zustände eines Isomorphismus entfernt haben. Da jedoch jeder Isomorphismus Iso sich in einem Zustand befinden muss, damit die Knoten im Bereich $B(Iso)$ sich in einem Zustand befinden können, wird es in einem solchen Fall dazu kommen, dass Stellen beim nächsten *propagiereKnoten* gefunden werden. Sie sind im neuen V_b. Somit sorgt ein Isomorphismus ohne Zustand für eine Kontradiktion.

Beispiel. Wir haben aus dem Kapitel 3.8.1 $Isos_b = Isos$ erhalten, unsere jetzigen Z_{Isos} und Ent befinden sich in Tabelle 3.2. Die Bereiche der Isomorphismen sind in Abbildung 3.7 zu sehen (2×2 Knoten in Leserichtung, aufsteigend sortiert nach Nummer) und das neue Z_O befindet sich in Abbildung 3.9. Für Iso_1 haben wir Zustände entfernt, welche für M_1 und M_3 gebraucht werden. Wir erhalten ${Z_{Isos}}'(Iso_1) = \{M_2\}$. Wir bestimmen die Entropie neu, nun gilt $Ent'(Iso_1) = 0$. Nachdem auf diese Weise alle Isomorphismen durchgegangen worden sind, bekommen wir die Tabelle 3.3. Da die Isomorphismen Iso_3 und Iso_5 keine Zustände mehr haben, wird *propagiereKnoten* GraphWaveFunctionCollapse aufgrund einer Kontradiktion beenden.

```
1  Funktion propagiereIsos
       Daten: Menge Isos_b, Hastabelle Z_Isos, Anzahl, Ent, Z_O^entf
       Ergebnis: Tabelle Z_Isos, Ent, Menge V_b
2      Menge V_b ← ∅
3      Menge Isos_entf ← ∅
4      für alle Iso ∈ Isos_b tue
5          für alle v' ∈ Schlüssel(Iso) tue
6              Knoten v ← Iso[v']
7              wenn v ∈ Schlüssel(Z_O^entf) dann
8                  für alle M ∈ Z_Isos[Iso] tue
9                      wenn M[v'] ∈ Z_O^entf[v] dann
10                         Z_Isos[Iso].entf(M)
11                         Isos_entf.einf(Iso)
12     für alle Iso ∈ Isos_entf tue
13         Ent[Iso] ← bestimmeEntropie(Z_Isos, Anzahl, Iso)
14         für alle v' ∈ Schlüssel(Iso) tue
15             V_b.einf(Iso[v'])
16     zurück Z_Isos, Ent, V_b
```

Algorithmus 9: *propagiereIsos*

Der Pseudocode der Funktion *propagiereIsos* befindet sich in Algorithmus 9. In *propagiereIsos* werden wir zuerst Z_{Isos} an Z_O anpassen und dabei $Isos_{entf}$ bestimmen (Z. 2 ff.). Es wird jedes $Iso \in Isos_b$ betrachtet und für jedes Iso jeder Knoten $v \in B(Iso)$. Die Muster werden entfernt, von denen für einen Knoten der benötigte Zustand entfernt worden ist. Sollte ein Muster aus $Z_{Isos}(Iso)$ entfernt worden sein, so ist Iso Teil von $Isos_{entf}$. Wir schätzen $Z_{Isos}(Iso)$ mit $\mathcal{O}(|alleMuster|)$ ab und erhalten eine Laufzeitkomplexität von $\mathcal{O}(|Isos_b| * |V_L| * |alleMuster|)$.

Nachdem Z_{Isos} angepasst und $Isos_{entf}$ bestimmt wurde, iterieren wir über $Isos_{entf}$ (Z. 12 ff.) um Ent anzupassen ($\mathcal{O}(|Isos_{entf}| * |alleMuster|)$) und V_b zu bestimmen ($\mathcal{O}(|Isos_{entf}| * |V_L|)$).

Da wir nur aus betrachteten Isomorphismen Zustände entfernen, ist $Isos_{entf}$ eine Untermenge von $Isos_b$. Für *propagiereIsos* ergibt sich eine Laufzeit von $\mathcal{O}(|Isos_b| * |V_L| * |alleMuster|)$.

3.8.3 Gesamt

Nach dem Beobachten eines Isomorphismus erhalten wir zu bearbeitende Knoten V_b. Wenn wir diese bearbeiten, erhalten wir zu bearbeitende Isomorphismen $Isos_b$. Sollten wir diese bearbeiten, so erhalten wir erneut eine Menge V_b...

Dies ist auch die gesamte Funktion *propagiere*, deren Pseudocode in Algorithmus 10 zu finden ist. Wir starten mit V_b und behandeln die beiden Mengen abwechselnd, bis V_b nach einer Iteration leer ist oder wir auf eine Kontradiktion stoßen.

Satz 2. propagiere *wird halten.*

Beweis. Sei o.B.d.A. V_b zu Beginn nicht leer.
Wenn $V_{entf} = \emptyset$, wird $Isos_b = \emptyset$ gelten. Wenn $Isos_b = \emptyset$ gilt, so wird $Isos_{entf} = \emptyset$ und somit das neue $V_b = \emptyset$.
Wenn jedoch $V_{entf} \neq \emptyset$, folgt $Isos_b \neq \emptyset$. Wenn dann $Isos_{entf} = \emptyset$ ist, folgt wieder, dass das neue $V_b = \emptyset$.
Selbst wenn keine Kontradiktion aufkommt, wird *propagiere* halten, sollte einer der genannten Fälle in einer Iteration auftreten.

Es bleibt ein Fall zu zeigen. Wenn für alle Iterationen $V_b \neq \emptyset$ gilt, wird es zu einer Kontradiktion kommen. Damit ein jedes V_b nicht leer ist, muss davor $Isos_{entf} \neq \emptyset$ gegolten haben. Daraus folgt, dass für mindestens einen Isomorphismus ein Zustand entfernt wurde. Analog muss ein Zustand eines Knotens entfernt werden, damit $V_{entf} \neq \emptyset$ und daraufhin $Isos_b \neq \emptyset$.
Es existieren jedoch zu Beginn insgesamt nur $|alleWerte| * |V_O|$ viele Knotenzustände und ebenso sind es auch nur $|alleMuster| * |Isos|$ viele Isomorphismenzustände. Folglich wird in einer Iteration der letzte Zustand eines Knotens oder Isomorphismus entfernt und wir erhalten eine Kontradiktion. □

1 **Funktion** *propagiere*
  Daten: *Tabelle* Z_O, Z_{Isos}, *Anzahl*, *Ent*, $Isos_V$, *Menge* V_b
  Ergebnis: *Tabelle* Z_O, Z_{Isos}, *Ent*
2 **solange** $V_b \neq \emptyset$ **tue**
3 $Z_O, Z_O^{entf}, Isos_b \leftarrow propagiereKnoten(V_b, Z_O, Isos_V, Z_{Isos})$
4 $Z_{Isos}, Ent, V_b \leftarrow propagiereIsos(Isos_b, Z_{Isos}, Anzahl, Ent, Z_O^{entf})$
5 **zurück** Z_O, Z_{Isos}, Ent

Algorithmus 10: *propagiere*

Die Laufzeit von *propagiere* ist abhängig von der Anzahl der Iterationen. Wir benötigen für jede Iteration, nach welcher *propagiere* nicht hält, je einen Knoten- und Isomorphismuszustand. Folglich ist die maximale Zahl der Iteration für einen *propagiere* Aufruf in $\mathcal{O}(min(|alleWerte| * |V_O|, |alleMuster| * |Isos|))$.

Dabei hat *propagiereKnoten* eine Laufzeit von $\mathcal{O}(|V_b| * (max_{v \in V_b}(|Isos_V(v)|) * |alleMuster| + |alleWerte|))$. Je Isomorphismus erhalten wir für V_b maximal $|V_L|$ Knoten, da bei der ersten Iteration V_b auf genau einem beobachtetem Isomorphismus basiert und in jeder darauffolgenden Iteration sich die Isomorphismen in $Isos_{entf}$ überlappen können, wobei die Knoten der Überlappung nicht doppelt bearbeitet werden.
Wir haben demnach Aufrufe von *propagiereKnoten* immer dann, wenn die Zustandsmenge eines Isomorphismus reduziert wurde, und einmalig in *initialisiere* für Knoten in den Bereichen aller Isomorphismen.
Für $|alleMuster| > 1$, können wir die maximale Laufzeit im Programmverlauf abschätzen mit
$\mathcal{O}(|Isos| * |V_L| * |alleMuster| * (max_{v \in V_O}(|Isos_V(v)|) * |alleMuster| + |alleWerte|))$.

propagiereIsos wird nur aufgerufen, wenn der Zustand eines Knotens reduziert worden ist. Wir schätzen die Laufzeit im gesamten Programmverlauf auf $\mathcal{O}(|alleWerte| * |V_O| * max_{v \in V_O}(|Isos_V(v)|) * |V_L| * |alleMuster|)$.

Betrachten wir erneut die Laufzeit von *propagiereKnoten*. Es gilt $|Isos| \leq |V_O| * max_{v \in V_O}(|Isos_V(v)|)$. Entsprechend können wir $\mathcal{O}(|Isos| * |V_L| * |alleMuster| * |alleWerte|)$ mit $\mathcal{O}(|V_O| * max_{v \in V_O}(|Isos_V(v)|) * |V_L| * |alleMuster| * |alleWerte|)$ abschätzen. Dies ist die Laufzeitkomplexität von *propagiereIsos*. Damit hat *propagiere* eine Laufzeit von $\mathcal{O}(|V_L| * |alleMuster| * max_{v \in V_O}(|Isos_V(v)|) * (|Isos| * |alleMuster| + |alleWerte| * |V_O|))$.

3.9 GraphWaveFunctionCollapse

Nun, da wir WaveFunctionCollapse gänzlich mit Graphen durchgegangen sind, müssen wir nur noch die dabei entstanden Funktionen zu dem Algorithmus GraphWaveFunctionCollapse zusammenfügen. Der Pseudocode befindet sich in Algorithmus 11.

Aus den Eingabegraphen G_I, G_O, G_L wird mit den Kantenlabels L_I, L_O, L_L und den Knotenzuständen Z_I des Graphen G_I in *initialisiere* ein GWFC-Zustand erstellt (Z. 1). Solange noch Isomorphismen existieren, welche sich in mehreren Zuständen befinden, wird ein solcher ausgewählt (Z. 1). Er wird beobachten, es wird seine Zustandsmenge auf ein Muster reduziert (Z. 3). Wir erstellen $V_b = B(Iso)$ und propagieren die Information durch den Graphen G_O, ausgehend von V_b (Z. 7). Sollte es zu keiner Kontradiktion kommen, wird GraphWaveFunctionCollapse Z_O zurückgeben (Z. 8), was jedem Knoten in V_O, welcher sich in mindestens einem Isomorphismusbereich befindet, eine Menge mit genau einem Zustand zuweist. Allen anderen Knoten wird die Menge *alleWerte* zugewiesen.

Daten: *Graph* G_I, G_O, G_L, *Tabelle* L_I, L_O, L_L, Z_I
Ergebnis: *Tabelle* Z_O oder Kontradiktion

```
1 V_O, Z_O, V_L, Isos, alleMuster, Anzahl, Z_Isos, Ent, Isos_V ←
    initialisiere(G_I, G_O, G_L, L_I, L_O, L_L, Z_I)
2 solange Iso ← wähleIso(Isos, Ent) tue
3 |   Z_Isos, Ent ← beobachte(Iso, Z_Isos, Anzahl, Z_O, Ent)
4 |   Menge V_b ← ∅
5 |   für alle v' ∈ V_L tue
6 |   |   V_b.einf(Iso[v'])
7 |   Z_O, Z_Isos, Ent ← propagiere(Z_O, Z_Isos, Anzahl, Ent, Isos_V, V_b)
8 zurück Z_O
```

Algorithmus 11: GraphWaveFunctionCollapse

Satz 3. *GraphWaveFunctionCollapse hält nach maximal* $|V_O|$ *Iterationen.*

Beweis. Wir wissen bereits, dass jede in GraphWaveFunctionCollapse aufgerufene Funktion hält. Sollte eine Kontradiktion aufkommen, so wird der Algorithmus beendet. Ansonsten wird in jeder Iteration ein Isomorphismus in mehreren Zuständen gewählt, dessen Bereich somit nicht gänzlich bestimmt ist. Dank *propagiere* werden wir keinen Isomorphismus in mehreren Zuständen finden, sollte jeder Knoten maximal einen Zustand haben. Folglich existiert dort mindestens ein Knoten, welcher mehrere Zustände hat. Nach dem Beobachten und Propagieren hat dieser Knoten nur noch einen Zustand. Da wir $|V_O|$ viele Knoten haben, folgt die Aussage. □

Nun können wir die Laufzeitkomplexität von GraphWaveFunctionCollapse bestimmen. Die Initialisierung benötigt $\mathcal{O}(|Isos_I| * |V_L| + |V_I| + |Isos| * |V_L| + |V_O| * |alleWerte| + |Isos| * |alleMuster|)$ (Kapitel 3.5) zuzüglich dem Propagieren und Finden der Isomorphismen. Das Propagieren benötigt im gesamten Verlauf des Algorithmus $\mathcal{O}(|V_L| * |alleMuster| * max_{v \in V_O}(|Isos_V(v)|) * (|Isos| * |alleMuster| + |alleWerte| * |V_O|))$ (Kapitel 3.8.3). Es bleiben je Iteration für das Auswählen der Isomorphismen $\mathcal{O}(|Isos|)$ (Kapitel 3.6) und für das Beobachten $\mathcal{O}(|alleMuster|)$ (Kapitel 3.7). Zudem generieren wir in $\mathcal{O}(|V_L|)$ in jeder Iteration V_b.

Sollten wir obiges zusammenfügen, so erhalten wir, vom Finden der Isomorphismen abgesehen,
$\mathcal{O}(|Isos_I|*|V_L|+|V_I|+|Isos|*|V_O|+|V_L|*|alleMuster|*max_{v\in V_O}(|Isos_V(v)|)*(|Isos|*|alleMuster|+|alleWerte|*|V_O|))$ als durchschnittliche Laufzeitkomplexität von GraphWaveFunctionCollapse.

In Kapitel 3.1 haben wir uns dafür entschieden, die verwendeten Datentypen als in der Größe konstant anzusehen. Dennoch kann die obige Abschätzung ein Gefühl darüber vermitteln, welche Mengen welchen Einfluss auf die Laufzeit haben können. Insbesondere scheint es empfehlenswert, die Menge der Isomorphismen per Knoten $|Isos_V(v)|$, die Menge der Muster $|alleMuster|$ sowie $|V_L|$ so gering wie möglich zu halten. Wenn wir Eigenschaften der Eingabe kennen, so können wir diese einsetzen um ein noch besseren Überblick über die Laufzeit zu erhalten.

Beispiel. Wir werden wie in unseren obigen Beispielen WaveFunctionCollapse mit Graphen ausführen. Die Maße des Eingabebildes sind $Höhe_I \times Breite_I$, die des Ausgabebildes $Höhe_O \times Breite_O$. Damit sind $|V_I| = Höhe_I * Breite_I$ und $|V_O| = Höhe_O * Breite_O$. Desweiteren hat ein Bereich die Größe $|V_L| = N^2$.
Wir können aus Bildern Graphen generieren, indem wir über die Pixel iterieren, wir benötigen für G_I $\mathcal{O}(Höhe_I * Breite_I)$. Analog benötigen wir zum Kreieren von G_O und Konvertieren von G_O zum Ausgabebild $\mathcal{O}(Höhe_O * Breite_O)$ und für G_L $\mathcal{O}(N^2)$. Die Isomorphismen lassen sich mit einer Laufzeit von $\mathcal{O}(Höhe_I * Breite_I * N^2)$ beziehungsweise $\mathcal{O}(Höhe_O * Breite_O * N^2)$ aufzählen.
Bereiche können nicht rotiert oder gespiegelt werden, folglich erhalten wir $max_{v\in V_O}(|Isos_V(v)|) = N^2$ und können $|Isos|$ mit $|V_O|$ sowie $|Isos_I|$ mit $|V_I|$ abschätzen.
Zusammengefasst können wir WaveFunctionCollapse mit einer Laufzeit von $\mathcal{O}(Höhe_I * Breite_I * N^2 + (Höhe_O * Breite_O)^2 + Höhe_O * Breite_O * N^4 * |alleMuster| * (|alleMuster| + |alleWerte|))$ ausführen.

Nun da wir die Laufzeitanalyse haben, können wir auch begründen, weshalb wir Kontradiktionen zulassen; GraphWaveFunctionCollapse ist effizient. Wenn $\mathcal{P} \neq \mathcal{NP}$, so existiert kein effizienter Algorithmus, welcher immer eine passende Wertzuweisung der Knoten ausgibt, sollte eine existieren. Es ist nicht einmal effizient entscheidbar, ob eine Wertzuweisung bei gegebenen Restriktionen möglich ist.

Satz 4. *Angenommen $\mathcal{P} \neq \mathcal{NP}$. Es ist im allgemeinen nicht effizient entscheidbar, ob für eine GraphWaveFunctionCollapse-Eingabe eine passende Ausgabe existiert.*

Beweis. Wir zeigen, dass das Problem $\mathcal{NP}$-vollständig ist. Man betrachte folgende Eingabe;

- G_I ist ein Graph mit k Knoten, von denen jeder Knoten je eine Kante zu jedem anderen Knoten hat. Die Werte der Knoten sind paarweise verschieden.
- G_L ist ein Graph mit 2 Knoten und einer Kante vom ersten zum zweiten Knoten.
- G_O ist ein zusammenhängender Graph mit mindestens 2 Knoten. Es gibt keine Kante von einem Knoten zu sich selbst (Schleife).

Nun haben wir als Muster jede Kombination von 2 unterschiedlichen Werten aus G_L für 2 benachbarte Knoten. Sollte GraphWaveFunctionCollapse erfolgreich terminieren, so existiert eine erfolgreiche Ausgabe und die Knotenwerte in der Ausgabe sind entsprechend paarweise verschieden. Dies entspricht einer Färbung von G_O mit maximal k Farben. Die Bestimmung, ob ein Graph k-knotenfärbbar ist, ist im allgemeinen $\mathcal{NP}$-vollständig [5]. Entsprechend ist im allgemeinen die Bestimmung, ob eine passende Ausgabe existiert, $\mathcal{NP}$-vollständig. □

Kapitel 4

Anwendungsbeispiele

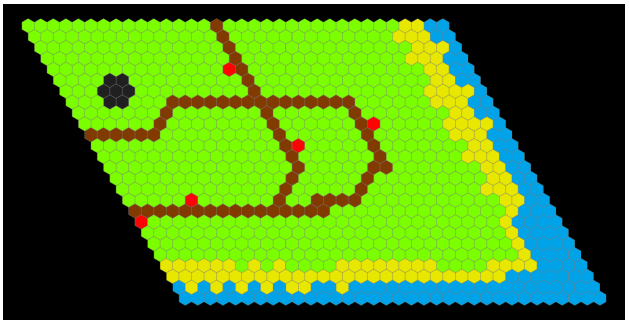
Die GraphWaveFunctionCollapse-Beispiele von Kapitel 3 waren zumeist analog zu den WaveFunctionCollapse-Beispielen in Kapitel 2. GraphWaveFunctionCollapse wäre jedoch nutzlos, könne es nicht auch Eingaben anderer Struktur bearbeiten.

Wohl einfach sind Beispiele, welche sich sehr an Bilder anlehnen. Knoten werden erneut Positionen des Raums repräsentieren und die Werte beschreiben, was sich an der jeweiligen Position befindet.

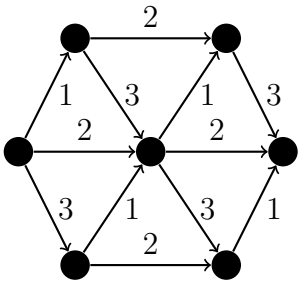
Unser erstes Beispiel ist ein Level eines Videospiels mit Blick aus der Vogelperspektive. Das Level ist mit regelmäßigen Hexagonen aufgebaut, wobei jedes Hexagon entweder Gras, Strand, Wasser, Weg, Haus oder Fels sein kann. Als Eingabe verwenden wir Abbildung 4.1a.

Jedes Hexagon ist ein Knoten. Sie haben Kanten zu ihren Nachbarn rechts oben mit dem Label 1, rechts mit dem Label 2 und rechts unten mit dem Label 3. Die Label verhindern, dass sich Muster drehen. Es könnte sonst sein, das zum Beispiel links im Ausgabelevel Wasser ist, wogegen wir uns in diesem Beispiel entscheiden.

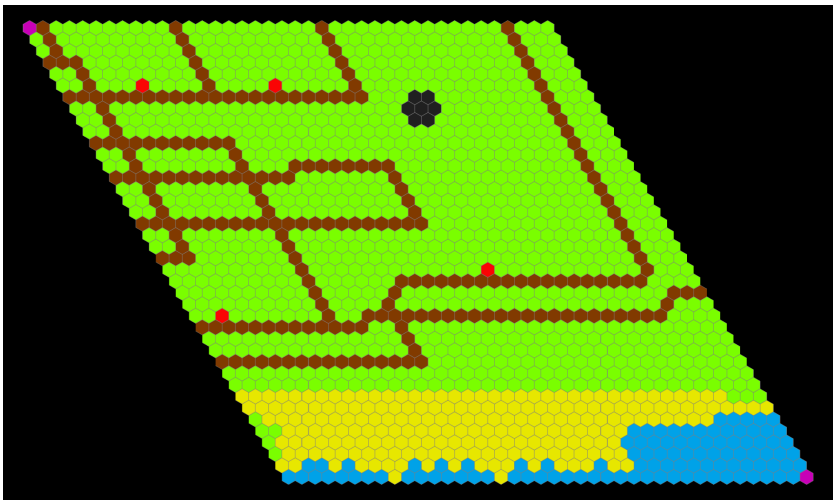
Als Bereiche für die Muster nehmen wir ein Hexagon inklusive aller direkten Nachbarn. Wir haben ein Hexagon inklusive der Nachbarn in der Mitte von G_I gewählt und verwenden den knoteninduzierten Teilgraphen (Abbildung 4.1b) als G_L.



(a)

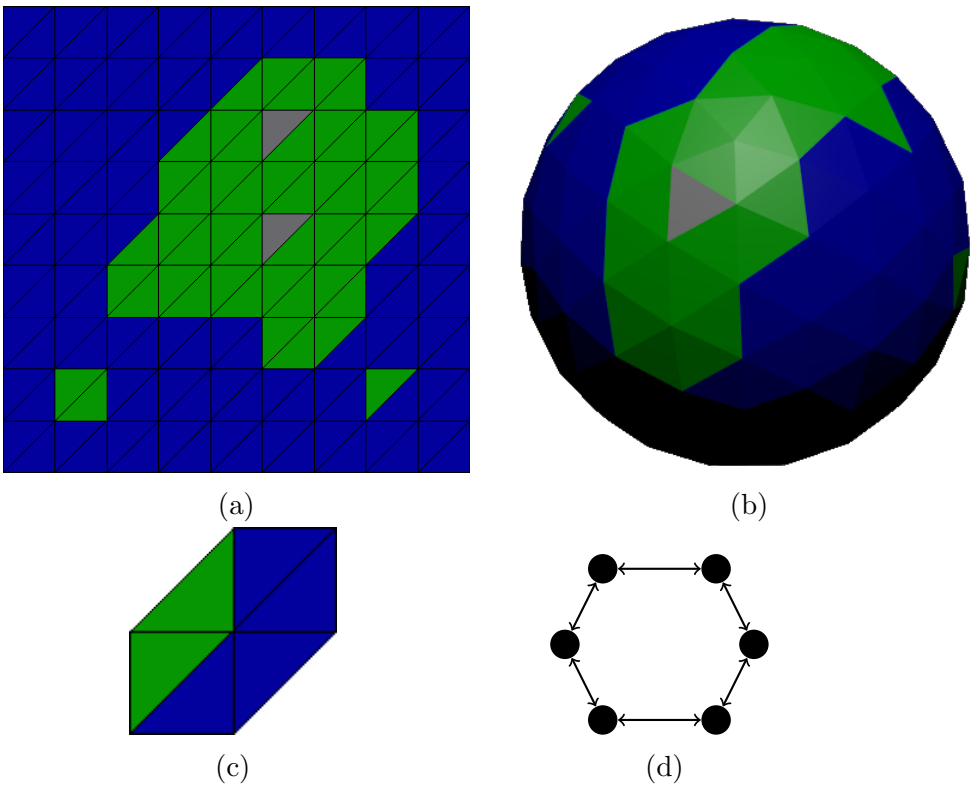


(b)



(c)

Abbildung 4.1: **4.1a ist ein Beispiellevel mit regelmäßigen Hexagonen. Wir sehen grünes Gras, gelben Strand, blaues Wasser, braunen Weg, rote Häuser und grauen Fels. Als G_L verwenden wir 4.1b. Das ausgegebene Level ist in 4.1c, die beiden violetten Hexagone sind in allen Zuständen.**



(a) (b) (c) (d)

Abbildung 4.2: **4.2a ist die Eingabe. Jedes Dreieck ist ein Knoten und ist jeweils mit Dreiecken mit gemeinsamen Kanten verbunden. In 4.2d ist G_L und ein gefundenes Muster ist in 4.2c. Die Ausgabe ist die in 4.2b zu sehende Icosphäre.**

Wir kreieren ein G_O mit einer Struktur analog zu G_I und erhalten von GraphWaveFunctionCollapse das Level in Abbildung 4.1b. Links oben und rechts unten sehen wir Hexagone, welche sich in keinem Isomorphismusbereich befanden und nun noch alle Zustände haben. Diese müssen manuell auf einen Zustand gesetzt werden.

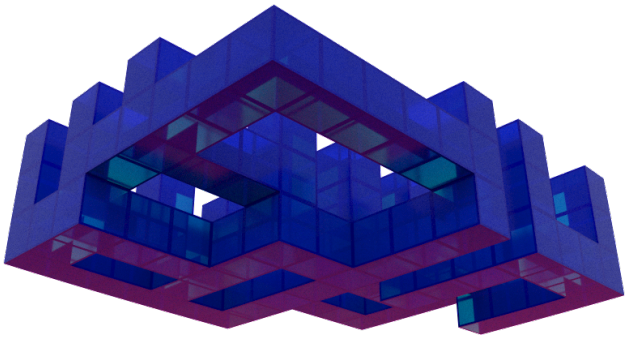
Nun da wir ein Videospiellevel haben, wollen wir für unser zweites Beispiel das Level in einer Weltkugel auswählen können. Wir kreieren eine 2-dimensionale Karte aus Dreiecken (Abbildung 4.2a). Jedes Dreieck ist entweder ein Wasser-, Gras- oder Stadtlevel. Wir sehen, dass sich außer am Rand die Dreiecke zu sechst je einen Punkt teilen. Wir verbinden Dreiecke mit den Dreiecken, mit denen sie sich je eine (geometrische) Kante teilen, wobei alle Label gleich sind; Drehungen der Muster sind erwünscht. Mit G_L aus Abbildung 4.2d erhalten wir Muster von je 6 Dreiecken, die sich einen gemeinsamen Punkt teilen. Ein solches Muster ist in Abbildung 4.2c zu sehen.

Wir haben eine Programmbibliothek für Graphen in eine 3D-Modellierungssoftware integriert und für eine Icosphäre[1] einen Graphen erstellt. In diesem sind ebenso die Dreiecke mit gemeinsamen geometrischen Kanten verbunden. An manchen Punkten hat unsere Icosphäre nur 5 Dreiecke, diese Stellen haben in Folge dessen weniger Restriktionen als andere, jedoch sind alle Dreiecke in mehreren Isomorphismenbereichen.

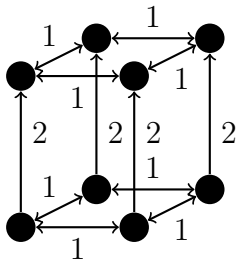
Wir konnten mit Hilfe der Bibliothek die Ausgabe von GraphWaveFunctionCollapse direkt als Texturinformationen auf das Modell der Icosphäre anwenden. Wir erhalten das Modell aus Abbildung 4.2b.

Worauf beim Verwenden von GraphWaveFunctionCollapse geachtet werden muss sind die „Ränder“ des Graphen. Während die Ränder des Levels keine Probleme bereiteten, hatte die Weltkugel gar keine Ränder. Betrachten wir das 3-dimensionale Labyrinth in Abbildung 4.3a. Es besteht aus Würfeln und hat die Maße $11 \times 8 \times 3$, wobei die X-Achse nach rechts hinten, die Y-Achse nach links hinten und die Z-Achse nach oben geht. Jede Position, an der ein Würfel sein kann, ist ein Knoten, der Knotenwert gibt an, ob an der entsprechenden Position ein Würfel ist oder nicht. Wir verbinden die Knoten mit allen direkten Nachbarn in der X, Y-Ebene mit einer Kante mit Label 1. Zusätzlich sind sie mit den direkten Nachbarn nach oben in der Z-Achse mit einer Kante mit Label 2 verbunden.
Als G_L verwenden wir den Graphen 4.3b mit welchem wir $2 \times 2 \times 2$ Blöcke als

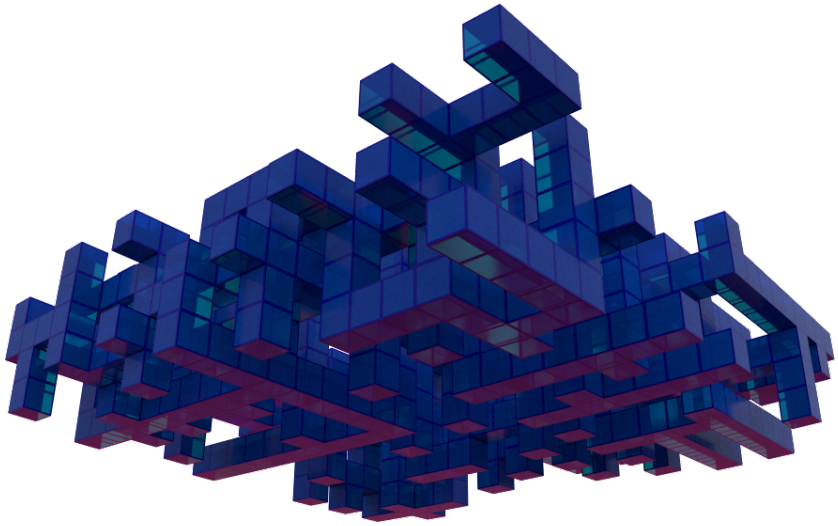
[1] Ein Objekt aus regelmäßigen Dreiecken, welches einer Kugel ähnelt



(a)



(b)



(c)

Abbildung 4.3: **4.3a ist ein aus Würfeln gebautes $11 \times 8 \times 3$ Labyrinth. Mit G_L 4.3b erhalten wir $2 \times 2 \times 2$ Blöcke als Muster, welche wir in der X, Y-Ebene (Label 1) beliebig drehen können. In der $20 \times 20 \times 5$ Ausgabe 4.3c sehen wir viele Sackgassen.**

Muster erhalten. Diese Muster können in der X, Y-Ebene gedreht werden. Als $20 \times 20 \times 5$ Ausgabe erhalten wir Abbildung 4.3c. Während in der Eingabe die meisten Pfade verbunden waren, sind am Rand der Ausgabe viele Sackgassen zu finden. GraphWaveFunctionCollapse bearbeitet die Ränder nicht anders als die Mitte des Labyrinths, die Pfade währen *dort* nicht abgebrochen, würde der Graph dort weitergehen.

Das Labyrinth-Beispiel zeigt, dass mehr Kontrolle über die Ausgabe von GraphWaveFunctionCollapse erwünscht sein könnte, als Alternative zur Nachbearbeitung. Hier empfiehlt sich, GraphWaveFunctionCollapse wie im folgendem Kapitel vorgestellt abzuwandeln.

Kapitel 5

Abwandlungen

Wir haben den Algorithmus GraphWaveFunctionCollapse vorgestellt und Anwendungsbeispiele gegeben. In [6] und [8] wurden Erweiterungen und Abwandlungen von WaveFunctionCollapse erwähnt, welche teils in angepasster Form mit GraphWaveFunctionCollapse kompatibel sind. In diesem Kapitel werden wir Abwandlungen nennen, welche neue Möglichkeiten bei der Verwendung von GraphWaveFunctionCollapse eröffnen können, insbesondere, da die meisten zueinander kompatibel sind.

Alternative Initialisierung. Während wir in GraphWaveFunctionCollapse in der Initialisierung Isomorphismen und Muster suchen, gibt es in manchen Fällen wenig Grund dazu. Gerade bei regelmäßigen Graphen, wie die aus den Anwendungsfällen in Kapitel 4, können die Isomorphismen intuitiv bekannt sein. Es genügt oft eine Schleife um die Isomorphismen aufzuzählen und so die benötigte Laufzeit zu senken, welche gerade bei großen Graphen zu einem Problem werden kann.

Ebenso gibt es keinen Grund eine Beispieleingabe zu erzwingen, wenn die gewünschte Mustermenge bereits bekannt ist. Eine direkte Angabe von *alleMuster* und *Anzahl* ermöglicht es, G_I, L_I und Z_I auszulassen. Hierbei wird *alleWerte* aus *alleMuster* bestimmt, es ist die Vereinigung der Bilder aller Muster.

Interaktivität. Diese Abwandlung nutzt die Definition des GWFC-Zustands. Dieser Zustand beinhaltet alle Informationen, welche GraphWaveFunction-Collapse benötigt um zu iterieren. Nun können wir vor einer Iteration, vor dem Auswählen eines Isomorphismus, den Zustand durch einen anderen ersetzen. Wir können den Algorithmus unterbrechen und manuell Zustände entfernen.

Man betrachte folgendes Szenario; Wir arbeiten mit einem Levelleditor und während wir ein Level von GraphWaveFunctionCollapse generieren lassen, stellen wir den aktuellen Zustand nach jeder Iteration dar. Wir betrachten, wie immer mehr der Zustand Gras beobachtet wird, und sind der Ansicht, dass ein Fluss in der Mitte eine gute Idee sei. Da der Editor nach jeder Iteration den Zustand speichert, können wir mehrere Zustände zurückspringen, als sich dort noch kein Gras befand. Wir reduzieren manuell manche Knotenzustände, sodass sie nur noch den Zustand Fluss haben. Um den Fluss herum reduzieren wir die Zustände wieder auf Gras. wir lassen GraphWave-FunctionCollapse weiter iterieren und erhalten ein Level mit unserem Fluss.

Hierbei ist es wichtig zu beachten, dass vor jeder Iteration Z_O und Z_{Isos} zueinander passen müssen. Falls wir manuell Isomorphismenzustände entfernt haben, können wir *propagiere* mit dem Bild der Isomorphismen ausführen. Sollten wir jedoch Knotenzustände reduziert haben, so benötigen wir ein *propagiere′* welches nicht mit Knoten V_b startet, sondern mit Isomorphismen $Isos_b$. Es sind diejenigen Isomorphismen, in deren Bild Knoten mit manuell reduzierten Zuständen sind.

Ein Sonderfall ist, bereits bei der Eingabe die Zustände zu reduzieren. Wenn beispielsweise Level wie aus Kapitel 4 aneinander angrenzen, so können wir dir Menge der mögliche Ausgaben beschränken, sodass am Rand an bestimmten Stellen immer Weg ist, und an den restlichen nie.

G_Ls. Wie in Abbildung 4.1c zu sehen ist, werden manche Knoten nicht von einem Isomorphismus erfasst. Während es in diesem Fall nur 2 Knoten sind, können es sich bei anderen Eingaben um bei weitem mehr handeln. Wenn wir nicht nur *ein* LG verwenden, so können wir dieses Problem beheben.

Wir übergeben eine Menge, deren jedes Element ein lokaler Graph G_L ist. Für jedes einzelne G_L werden eigene V_L, *Isos*, *alleMuster*, *Anzahl*, Z_{Isos}, *Ent* und $Isos_V$ bestimmt. Beim Auswählen eines Isomorphismus wird aus allen Isomorphismen aller G_L einer ausgewählt. Nachdem dieser propagiert wurde, werden die Knotenzustände Z_O an alle Isomorphismenzustände Z_{Isos} aller G_L angepasst. Auch werden alle Z_{Isos} an Z_O angepasst. Die Ausgabe wird dann von allen G_L eingeschränkt.

Dies eröffnet einige Möglichkeiten. Ein Beispiel wäre, dass man bei einem Bild möchte, dass alle 1×5 Bereiche als Muster verwendet werden, aber auch alle 5×1 Bereiche. Hierbei alle möglichen 5×5 Kombinationen zu bestimmen könnte unpraktikabel sein.
Ein weiteres Beispiel wären Graphen, bei denen wir zusätzliche Restriktionen hinzufügen möchte, wie der Weltkugel (Abbildung 4.2), welche an manchen Stellen weniger Restriktionen hatte als an anderen (s. Kapitel 4).

Partieller Neustart. Sollten wir eine Kontradiktion erhalten, so brauchen wir nicht GraphWaveFunctionCollapse neu zu starten. Wir merken uns einfach manche der GWFC-Zustände und bei einer Kontradiktion nehmen wir einen alten Zustand und starten von diesem. Hierbei sollte jedoch die Kontradiktionen gezählt und gegebenenfalls der Algorithmus abgebrochen werden, da sonst eine Endlosschleife entstehen kann, sollte keine Ausgabe vom jeweiligen Zustand möglich sein.

Es gibt noch eine Möglichkeit, nur teilweise neu zu starten. Wir wählen einen Radius r. Bei einer Kontradiktion betrachten wir den Knoten oder Isomorphismenbereich, in dem die Kontradiktion stattfand. Zusätzlich betrachten wir auch alle Knoten, welche wir von der Kontradiktion aus über bis zu r Kanten erreichen können, wobei wir nicht zwischen Vor- und Rückwärtskanten unterscheiden. Die Knoten werden in alle Knotenzustände versetzt und ebenso werden alle Isomorphismen, deren Bereiche gänzlich in der betrachteten Knotenmenge liegt, in alle Isomorphismenzustände versetzt. Da manche Isomorphismenbereiche nur teilweise in der betrachteten Knotenmenge liegen, könnten die Knotenzustände Z_O nicht zu den Isomorphismens-

zuständen Z_{Isos} passen. Sollten Z_O und Z_{Isos} ohne Kontradiktion aneinander angepasst worden sein, so kann GraphWaveFunctionCollapse fortgesetzt werden. Es ist wichtig, direkt beim Auftreten der Kontradiktion die Zustände zurückzusetzen, da es sonst sein kann, dass durch das Propagieren große Teile des Graphen sich in keinem Zustand mehr befinden. Es ist ebenso auf Endlosschleifen zu achten.

Kapitel 6

Ergebnisse

In dieser Arbeit haben wir den PCG-Algorithmus GraphWaveFunctionCollapse entwickelt. Wir haben den Ausgangsalgorithmus WaveFunctionCollapse betrachtet und die einzelnen Funktionen so rekonstruiert, dass sie auf Graphen arbeiten. Statt der ursprünglichen Teilbildern wurden Knotenmengen als Bereiche verwendet, welche wir mit einem Graphen G_L und Teilgraphisomorphie bestimmten. Wir wichen davon ab, Bereiche zur Beobachtung auszuwählen, um Uneindeutigkeiten bei der Bestimmung der Entropie zu vermeiden und nahmen die Teilgraphisomorphismen. Die möglichen Muster der Isomorphismen und Werte der Knoten passten wir mit Constraint Propagation aneinander an, wobei es zu Kontradiktionen kommen kann, welche nicht innerhalb von Polynominalzeit verhindert werden können. Es wurden Abwandlungen von GraphWaveFunctionCollapse genannt, um die Möglichkeiten des Algorithmus zu erweitern.

Wir haben Anwendungsbeispiele gegeben, bei denen wir geometrische Konstrukte als Graphen interpretiert haben und die Ausgabe auf ähnliche Konstrukte anwendeten. Unter anderem haben wir dabei unseren Algorithmus in eine Modellierungssoftware integriert. Gerade die Integration in andere Software scheint uns eine Möglichkeit wie GraphWaveFunctionCollapse in der Praxis von Nutzen sein kann.

Mit den Anwendungsbeispielen haben wir gezeigt, dass unser Algorithmus nicht auf Bilder beschränkt ist sondern viele Ein- und Ausgaben unterschiedlicher Struktur bearbeiten kann. Als negativen Punkt zeigte sich, dass zusätzliche Arbeit in Relation zu WaveFunctionCollapse nötig ist. Eingaben müssen in Graphen konvertiert, ein Ausgabegraph muss erstellt und am Ende aus diesem die gewünschte Ausgabe generiert werden. Zudem muss der Graph G_L kreiert werden, dessen Aufbau von der jeweiligen Anwendung abhängt. Sollte GraphWaveFunctionCollapse jedoch in ein Programm integriert werden, so können G_L und die Konvertierung der Graphen eingebettet werden, ein einmaliger Arbeitsaufwand. Auch wenn ein nur auf bestimmte Ein- und Ausgaben beschränktes Programm, welches auf WaveFunctionCollapse basiert, geschrieben werden soll, so dürfte in einigen Fällen GraphWaveFunctionCollapse eine einfache Möglichkeit bereitstellen, die Idee des Programms zu testen.

Die Anwendungsbeispiele beschränkten sich auf euklidische Geometrie. Wir haben ein großes Interesse, weitere Anwendungsmöglichkeiten zu finden und sehen diese unter anderem in Animationen, selbstähnlichen Fraktalen, nichteuklidischem Leveldesign und der Musik.

Literaturverzeichnis

[1] Cordella, Luigi P, Pasquale Foggia, Carlo Sansone und Mario Vento: *A (sub) graph isomorphism algorithm for matching large graphs*. IEEE transactions on pattern analysis and machine intelligence, 26(10):1367–1372, 2004.

[2] Cordella, Luigi Pietro, Pasquale Foggia, Carlo Sansone und Mario Vento: *An improved algorithm for matching large graphs*. In: *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, Seiten 149–159, 2001.

[3] Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest und Clifford Stein: *Introduction to algorithms*. MIT press, 2009.

[4] Dahlskog, Steve, Julian Togelius und Mark J Nelson: *Linear levels through n-grams*. In: *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*, Seiten 200–206. ACM, 2014.

[5] Garey, M. R., D. S. Johnson und L. Stockmeyer: *Some Simplified NP-complete Problems*. In: *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, Seiten 47–63, New York, NY, USA, 1974. ACM.

[6] Gumin, Maxim: *WaveFunctionCollapse Readme.md*. `https://github.com/mxgmn/WaveFunctionCollapse/blob/master/README.md`, Mai 2018. abgerufen: 02.06.2018.

[7] Horswill, Ian Douglas und Leif Foged: *Fast Procedural Level Population with Playability Constraints.* In: *AIIDE*, 2012.

[8] Karth, Isaac und Adam M. Smith: *WaveFunctionCollapse is Constraint Solving in the Wild*. In: *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, Seiten 68:1–68:10, New York, NY, USA, 2017. ACM.

[9] Merrell, Paul C: *Model synthesis*. Doktorarbeit, University of North Carolina at Chapel Hill, 2009.

[10] Neufeld, Xenija, Sanaz Mostaghim und Diego Perez-Liebana: *Procedural level generation with answer set programming for general video game playing*. In: *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th*, Seiten 207–212. IEEE, 2015.