Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Interaktive Kantenkontraktion für Minorensuche

Samuel Holderbach

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Stefan Funke
Betreuer/in:	Tobias Rupp
Beginn am:	4. Juli 2020
Beendet am:	4. Januar 2021

Kurzfassung

Ein Minor ist ein Graph, der aus einem Teilgraphen eines anderen Graphen durch die Kontraktion von Kanten entsteht [Moh06]. In der Graphentheorie spielen Minoren eine äußerst wichtige Rolle. Beispielsweise kann man feststellen, ob es möglich ist, einen Graphen überschneidungsfrei in eine Ebene einzuzeichnen, indem man prüft, ob dieser weder den vollständigen Graphen K_5 noch den vollständigen bipartiten Graphen $K_{3,3}$ als Minor enthält [Wag37]. Trotz ihrer Relevanz für die Graphentheorie ist die Minorenrelation weniger intuitiv als zum Beispiel die Teilgraphen- oder die Unterteilungsrelation. Im Zuge meiner Bachelorarbeit habe ich deshalb ein Spiel mit dem Namen *Minorfinder* [MF20] als Teil einer Kollektion kleiner Rätsel-Computerspiele [PPC20] entworfen, bei dem interaktiv Minoren in anderen Graphen gesucht werden können. *Minorfinder* lehnt sich dabei an *Untangle* an; ein Spiel, bei dem man Graphen planar in die Ebene legen muss, welches ebenfalls in der Spielekollektion zu finden ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

Minoren sind die Grundlage für viele wichtige Erkenntnisse in der Graphentheorie sowie für algorithmische Lösungen verschiedener komplexer Probleme. Zu den wichtigsten Resultaten der Forschung bezüglich Minoren zählen unter anderem der Satz von Wagner [Wag37], der die Minorenrelation in Verbindung mit der Planarität von Graphen bringt und das Minorentheorem von Neil Robertson und P.D. Seymour [RS01], das tiefgreifende Erkenntnisse über Minoren liefert. Die Minorenrelation ist trotz ihrer wichtigen Rolle in der Graphentheorie weniger intuitiv als zum Beispiel die Teilgraphen- oder die Unterteilungsrelation. Mit der Entwicklung eines Spiels, das die interaktive Suche nach Minoren durch das Löschen von Knoten bzw. Kanten sowie das Kontrahieren von Kanten ermöglicht, soll das allgemeine Verständnis für Minoren verbessert werden. *Simon Tatham's Portable Puzzle Collection* [PPC20] ist eine Sammlung von kleinen Rätsel-Computerspielen, die durch logische Ansätze gelöst werden können. In der Sammlung befinden sich zu einem großen Teil eigene Kreationen, wie beispielsweise *Untangle*, ein Spiel bei dem Graphen planar in die Ebene gelegt werden sollen; aber auch Spiele mit höherem Bekanntheitsgrad, zum Beispiel *Mines*, besser bekannt als *Minesweeper*, sind vertreten. Die Spielekollektion eignet sich perfekt für die Entwicklung eines solchen Spiels, da sie ein Template für neue Spiele sowie eine hilfreiche API und ein vollständiges Frontend bereitstellt. Im Zuge meiner Bachelorarbeit habe ich das Puzzle mit dem Namen *Minorfinder* [MF20] als Teil der Spielesammlung von Simon Tatham programmiert. *Minorfinder* fungiert als Gegenstück zu herkömmlichen Lehrmethoden, indem es Interessierten spielerisch ein besseres visuelles Verständnis für Minoren vermittelt. Das Spiel ist so aufgebaut, dass man einen bestimmten Minor und einen Ausgangsgraphen angezeigt bekommt. Aufgabe ist es, den Minor aus dem Ausgangsgraphen zu extrahieren. Darüber hinaus hat der Spieler die Möglichkeit, das Programm selbst die Lösung berechnen und anschließend anzeigen zu lassen.

2 Verwandte Arbeiten

Diese Arbeit beschreibt die Entwicklung eines Computerspiels namens *Minorfinder* [MF20] für *Simon Tatham's Portable Puzzle Collection* [PPC20]. Die Spielesammlung umfasst eine Vielzahl von Einzelspieler-Computerspielen, mit denen Spieler ihre logischen Fähigkeiten testen können. *Minorfinder* beschäftigt sich mit der Minorenrelation auf Graphen und orientiert sich vom Aufbau her an *Untangle*, einem Spiel, das mit der Planarität von Graphen ebenfalls ein graphentheoretisches Konzept behandelt. Ein weiteres Puzzle namens *Group* setzt beispielsweise die Gruppentheorie spielerisch um; das Spiel ist allerdings kein Teil der offiziellen Spielekollektion, da es sich noch in der Entwicklung befindet.

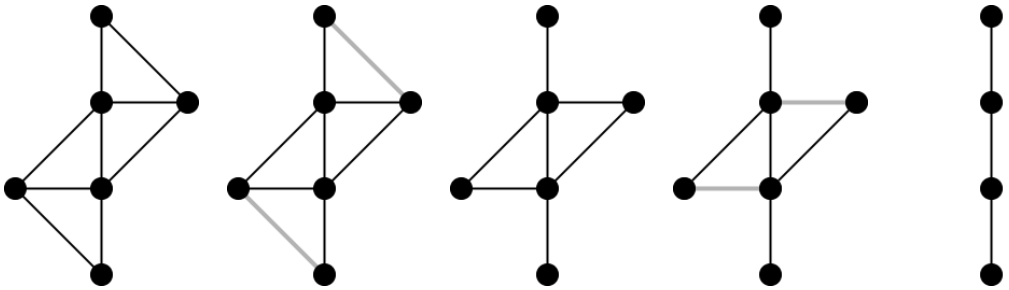
Das Implementieren von *Minorfinder* brachte einige Herausforderungen mit sich, die eng mit grundsätzlichen Problemen aus der Graphentheorie zusammenhängen; eines dieser Probleme ist das Graphisomorphie Problem. [MP13] beschreibt einen allgemeingültigen Algorithmus zum Bestimmen der Automorphismengruppe eines Graphen, den ich auf meine Bedürfnisse angepasst habe. Der Suchbaum, den dieser Algorithmus erzeugt, kann mit Hilfe mehrerer Ansätze optimiert werden, die in [LCA14] genauer erklärt werden.

3 Grundlagen

3.1 Minoren

Im weiteren Verlauf meiner Arbeit beziehe ich mich ausschließlich auf einfache ungerichtete Graphen ohne Schlingen und Mehrfachkanten. Ein einfacher ungerichteter Graph ohne Schlingen und Mehrfachkanten ist ein Graph mit einer Knotenmenge V und einer Kantenmenge $E \subseteq \binom{V}{2}$.

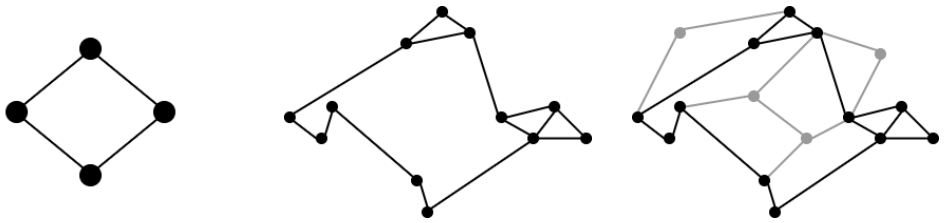
Auf Graphen können verschiedene Relationen definiert werden, die Zusammenhänge zwischen zwei – oder mehreren – Graphen beschreiben. Eine der grundlegenden Graphenrelationen ist die Minorenrelation, die auf der Teilgraphenrelation aufbaut. Ein Teilgraph ist ein Graph, der aus einem anderen Graphen durch das Weglassen von Knoten und Kanten entsteht, d.h. für die Knotenmenge U des Teilgraphen und die Knotenmenge V des Ausgangsgraphen gilt: $U \subseteq V$; selbiges gilt für die Kantenmengen der Graphen. Aus einem Teilgraphen entsteht ein Minor durch Kantenkontraktion. Unter Kantenkontraktion versteht man die Verschmelzung zweier benachbarter Knoten zu einem einzigen unter Beibehaltung aller zu den beiden Knoten inzidenten Kanten – abgesehen von der Kante, die die beiden Knoten miteinander verbindet. Die Minorenrelation ist damit allgemeiner als die Teilgraphenrelation: Jeder Teilgraph ist auch ein Minor, jedoch gilt diese Implikation nicht umgekehrt. In Abbildung 3.1 wird veranschaulicht, wie ein Minor durch Löschung und Kontraktion von Kanten aus einem anderen Graphen entsteht.



(a) Ausgangsgraph **(b)** Ausgangsgraph mit zu löschenden Kanten markiert **(c)** Teilgraph **(d)** Teilgraph mit zu kontrahierenden Kanten markiert **(e)** Minor

Abbildung 3.1: Entstehung eines Minors durch Löschung von Kanten und Kantenkontraktion

Es ist auch möglich, aus einem Graphen einen größeren Graphen zu erzeugen, der den ersten als Minor enthält. Gegeben sei ein Minor G mit einer Knotenmenge $V = \{v_1, v_2, ...\}$ und einer Kantenmenge $E = \{e_1, e_2, ...\}, \forall e_i \in E : e_i := \{v_k, v_l\}, v_k, v_l \in V$. Ersetzt man nun alle Knoten $v_i \in V$ durch Graphen G_{v_i} und alle Kanten $e_i = \{v_k, v_l\} \in E$ durch Kanten $e_i = \{G_{v_k}, G_{v_l}\}$, erhält man einen neuen Grahen, der IG genannt wird. IG steht für *inflated* G; die Bezeichnung zielt also auf das Aufblasen der Knoten von G zu Graphen ab. G ist genau dann der Minor eines Graphen H, wenn IG ein Teilgraph von H ist [ZG]. Abbildung 3.2 zeigt, wie H aus G entstehen könnte.



(a) Minor G **(b)** Der Graph IG **(c)** Der Graph H

Abbildung 3.2: Entstehung des Graphen H aus seinem Minor G

Außer der allgemeinen Definition von Minoren existiert noch eine Vielzahl von spezielleren Definitionen. Eine der bekanntesten Spezialisierungen der Minorendefinition ist die des topologischen Minors, die auf der Unterteilungsrelation basiert. Ein Unterteilungsgraph ist ein Graph, der aus einem anderen Graphen durch die Unterteilung von Kanten – indem zusätzliche Knoten auf diese gelegt werden – entsteht. Ein Graph G_1 enthält einen anderen Graphen G_2 genau dann als topologischen Minor, wenn G_1 einen Teilgraphen enthält, der ein Unterteilungsgraph von G_2 ist [ZG]. Im Zuge der Entwicklung meines Spiels habe ich mich jedoch auf die allgemeine Definition von Minoren festgelegt, da mir diese mehr Flexibilität beim Generieren des Graphen bietet, in dem der Minor enthalten sein soll.

Die Bedeutung von Minoren in der Graphentheorie, besonders in Bezug auf die Planarität von Graphen, aber auch im allgemeinen Sinne, lässt sich besonders anhand einiger Forschungsresultate verdeutlichen. Eine Menge M von Graphen wird als *minor-closed* bezeichnet, wenn für jeden Graphen G in dieser Menge all seine Minoren ebenfalls in M enthalten sind. Als *forbidden minor* von M wird ein Graph bezeichnet, welcher selbst nicht in M liegt – anders als all seine Minoren. Ein Beispiel für eine solche Menge ist die Menge aller planaren Graphen, also die Menge aller Graphen, die überschneidungsfrei in die Ebene eingezeichnet werden können [Moh06]. Der Satz von Wagner besagt, dass ein Graph genau dann planar ist, wenn er weder den K_5 noch den $K_{3,3}$ – also keinen der beiden *Kuratowski-Graphen* – als Minor enthält. Oftmals wird der Satz auch folgendermaßen definiert: Ein Graph ist planar, wenn er keinen Teilgraphen enthält, der ein Unterteilungsgraph des K_5 oder des $K_{3,3}$ ist; diese Definition ist besser bekannt als Satz von Kuratowski. Beide Sätze sind äquivalent, was aus der obigen Definition von Minoren hervorgeht. Die *forbidden minors* der Menge aller planaren Graphen sind also der K_5 und der $K_{3,3}$, da all ihre Minoren planar sind [Moh06]. Einen noch tiefgreifenderen Beweis erbrachten die beiden Forscher Neil Robertson und P.D. Seymour, die zeigen konnten, dass jeder Graph aus einer unendlichen Menge von Graphen isomorph zu einem Minor eines anderen Graphen aus dieser Menge ist [RS01].

3.2 Graph Isomorphie und Automorphismen

Zwei Graphen G und H mit den Knotenmengen $V_G = \{v_{G_1}, v_{G_2}, ..., v_{G_n}\}$ und $V_H = \{v_{H_1}, v_{H_2}, ..., v_{H_n}\}$ werden als isomorph bezeichnet, wenn Abbildungen $\varphi : V_G \rightarrow V_H$ bzw. $\varphi^{-1} : V_H \rightarrow V_G$ existieren, sodass $\forall e_G := \{v_{G_i}, v_{G_j}\} \in E_G,\ i, j \in \{1, 2, ..., n\} : \{\varphi(v_{G_i}), \varphi(v_{G_j})\} \in E_H$ und umgekehrt [MP13]. Die Isomorphie-Eigenschaft lässt sich auch auf Minoren übertragen. Ist ein Graph G ein Minor eines Graphen H, so ist jeder Graph, der isomorph zu G ist, ebenfalls ein Minor von H.

Das Graphisomorphie Problem GI beschreibt das Problem festzustellen, ob zwei Graphen isomorph sind; bis heute ist nicht bekannt, ob ein effizienter Algorithmus existiert, der es in Polynomialzeit löst. Darüber hinaus ist nicht einmal bekannt, ob GI in P liegt oder ob es NP-vollständig ist. Das aktuellste Forschungsergebnis auf diesem Gebiet wird in [Bab16] zusammengefasst. Das Paper stellt einen Beweis vor, der zeigt, dass GI – zumindest in der Theorie – in quasipolynomieller Zeit, genauer gesagt in $\exp((\log n)^{O(1)})$, gelöst werden kann. Die Frage, ob in der Praxis zwei beliebige Graphen in quasipolynomieller Zeit auf Isomorphie getestet werden können, lässt Lazlo Babai jedoch unbeantwortet. Des Weiteren stellt der Artikel die weitreichende These auf, dass GI nur dann NP-vollständig sein kann, wenn alle Probleme aus NP in quasipolynomieller Zeit lösbar sind.

Auch wenn bisher kein Polynomialzeit-Algorithmus für das Graphisomorphie Problem bekannt ist, gibt es wie oben geschrieben dennoch Alternativen zu einem reinen Bruteforce-Algorithmus, der für zwei Graphen jede Mögliche Permutation betrachtet und überprüft, ob diese ein Isomorphismus zwischen den beiden Graphen ist. Eine Permutation ordnet jedem Knoten des einen Graphen einen Knoten des anderen Graphen zu. Für Graphen G und H mit Knotenmengen V_G und V_H ist eine Permutation also eine bijektive Funktion $\rho : V_G \rightarrow V_H$. Eine Permutation ist ein Isomorphismus, wenn ρ zudem – wie die obige Funktion φ – jeder Kante aus E_G eine äquivalente Kante aus E_H zuordnet.

In [MP13] wird ein allgemeingültiger und effizienter Algorithmus zum bestimmen der Automorphismengruppe eines Graphen vorgestellt. Ein Automorphismus eines Graphen G ist ein Isomorphismus von G auf sich selbst, sprich ein Isomorphismus $\varphi : V_G \rightarrow V_G$. Die Menge aller Automorphismen von G bildet mit der Verkettung als Verknüpfung die Automorphismengruppe von G. Zum Finden von Automorphismen werden *Colourings* von Graphen definiert; ein Colouring eines Graphen G mit einer Knotenmenge V ist eine surjektive Funktion $\pi : V \rightarrow \{1, 2, ..., k\}, 1 \leq k \leq |V|$. Die Elemente der Menge $\{1, 2, ..., k\}$ sind also die verschiedenen Farben des Colourings π, mit denen die Knoten des Graphen G «eingefärbt» werden können. Eine Menge von Knoten, die die selbe Farbe besitzen, wird auch Zelle genannt. Um Permutationen eines Graphen zu finden, die ein Automorphismus auf diesen Graphen beschreiben, wird ein Suchbaum definiert, dessen Knoten Colourings des Graphen sind. Die Wurzel des Suchbaums ist das gröbste Colouring, also das mit dem kleinsten k, während seine Blätter die feinsten Colourings sind – sprich genau die Colourings, für die $k = |V|$ gilt. Die Zellen eines jeden Blattes enthalten also jeweils genau ein Element. Damit ist jedes Blatt eine Permutation auf der Knotenmenge des Graphen und somit ein Automorphismus-Kandidat. Das Verfeinern der Knoten übernimmt eine sogenannte *Refinement*-Funktion, die sicherstellt, dass für alle Kinder c_{child} eines Colourings c im Suchbaum $k_{c_{child}} \geq k_c$ gilt. Zudem erklärt [MP13], wie Colourings und somit ganze Teilbäume aus dem Suchbaum geprunt werden können, wodurch die Effizienz der Suche erheblich gesteigert wird.

3.3 Ähnlichkeit von Graphen

Die Ähnlichkeit zweier Graphen kann anhand ihrer *Graph Edit Distance* [GXTL09] angegeben werden. Die *Graph Edit Distance* zwischen zwei Graphen ist kein genauer Messwert; sie bildet die Basis für *inexact graph matching* und gibt für zwei Graphen – ähnlich wie die Levenshtein-Distanz bei Wörtern – die minimalen Kosten an, die nötig sind, um den einen Graphen in einen zu dem anderen isomorphen Graphen umzuwandeln. Es gibt eine Vielzahl von verschiedenen Algorithmen und Methoden für die unterschiedlichsten Graphen, mit deren Hilfe man die *Graph Edit Distance* berechnen oder approximieren kann. Der Großteil der existierenden Algorithmen arbeitet mit den Kosten verändernder Operationen an den Graphen. Zu den gängigsten Operationen gehören folgende:

- Knoten hinzufügen oder löschen
- Knoten-Substitution (Ersetzung eines Knoten-Labels)
- Kanten hinzufügen oder löschen
- Kanten-Substitution

Die Kosten für die einzelnen Operationen sind von Algorithmus zu Algorithmus unterschiedlich festgelegt. Eine Übersicht über die wichtigsten Methoden zur Berechnung der *Graph Edit Distance* ist in [GXTL09] zu finden.

4 Interaktive Kantenkontraktion mit Minorfinder

4.1 Portable Puzzle Collection

Simon Tatham's Portable Puzzle Collection [PPC20] umfasst eine Reihe von kleinen Rätsel-Computerspielen, die mit Hilfe von logischem Denken gelöst werden können. Die ursprüngliche Intention war es, eine Sammlung von Spielen zu erschaffen, die auf möglichst allen Geräten installiert und gespielt werden können. Die Spielekollektion beinhaltet sowohl Nachimplementierungen bekannter Spiele wie beispielsweise *Minesweeper*, das als *Mines* realisiert wurde, als auch einige innovative Eigenkreationen von Simon Tatham oder anderen Entwicklern. Zu diesen gehört unter anderem das Puzzle *Untangle*, bei dem man einen Graphen mit vielen sich überschneidenden Kanten planar in die Ebene legen muss. Da *Minorfinder* ebenfalls ein graphentheoretisches Spiel werden sollte, stellte sich *Untangle* als geeignete Grundlage für seine Entwicklung heraus; die beiden Spiele weisen besonders bei der Visualisierung große parallelen auf.

Die Puzzle-Backends sind ausschließlich in *C* implementiert und werden von einem *Java/C*-Frontend portabel gemacht. Dieses umfasst unter anderem Schnittstellen für *Unix*, *Windows* und *MacOS*. Eine in *C* geschriebene Middleware sorgt dafür, dass die jeweiligen Frontend-Schnittstellen mit den Puzzle-Backends kompatibel sind. Bei der Entwicklung eines neuen Puzzle-Backends wird man durch ein Backend-Template, hohe Modularität, eine API mit vielen nützlichen Funktionen und eine detaillierte Dokumentation unterstützt. Die API umfasst unter anderem Funktionen zum Generieren zufälliger Zahlen bzw. Zahlenfolgen sowie zum Zeichnen von Spielelementen. Des Weiteren verfügt sie über eine Datenstruktur mit dynamischer Größe, in der Elemente sowohl sortiert als auch unsortiert gespeichert werden können. Somit kann man sich bei der Entwicklung eines neuen Spiels fast ausschließlich auf die zugrundeliegende Logik konzentrieren.

Damit ein Puzzle in die Sammlung aufgenommen werden kann, muss es in jedem Fall das Kriterium der *Fairness* erfüllen; das bedeutet, dass ein Spieler mit ausreichenden Fähigkeiten – mit Hilfe dieser Fähigkeiten und nicht alleine durch Glück – dazu in der Lage sein muss, das Puzzle zu lösen. Kann ein Spieler ein Spiel nicht lösen, muss es wiederum auf mangelnde Kenntnisse seitens des Spielers zurückzuführen sein.

4.2 Umfang und Ziel des Spiels

Minorfinder ist ein graphentheoretisches Spiel, bei dem man aus einem zufälligen Graphen einen gegebenen Minor extrahieren muss. Zu Beginn eines neuen Spiels werden einem Spieler – wie in Abbildung 4.1 zu sehen – zwei Graphen angezeigt. Der Graph in der linken oberen Ecke, dessen Knoten rot gefärbt sind, ist ein Minor des Graphen mit den blauen Knoten; letzteren werde ich fortlaufend als Spielgraphen bezeichnen.

Nun soll der Spieler durch das Anwenden verschiedener Spielzüge auf den Spielgraphen, zum Beispiel durch Kantenkontraktion, diesen zu einem Graphen transformieren, der nach der Definition aus Abschnitt 3.2 isomorph zu dem angezeigten Minor ist. In Abschnitt 4.5 werden die verschiedenen möglichen Spielzüge genauer erklärt.

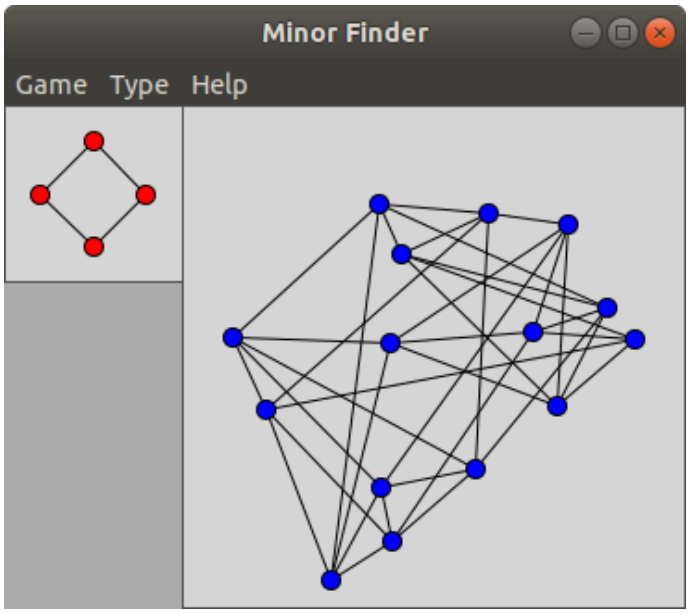
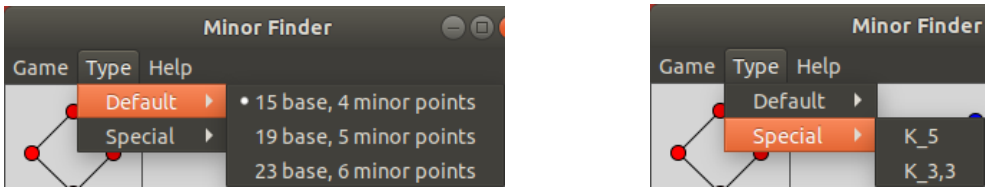


Abbildung 4.1: Anfangszustand eines Spiels

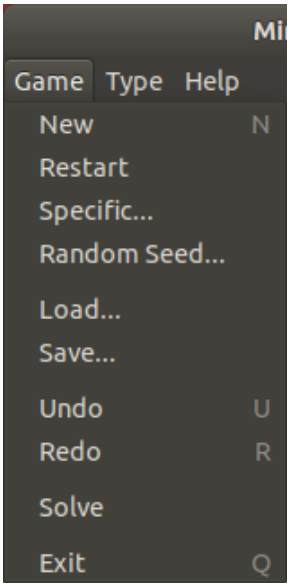
Das Spiel unterscheidet zwischen zwei Spielmodi: *Default* und *Special*. Der Standardmodus generiert zufällige Minoren unterschiedlicher Größen, während der spezielle Modus lediglich Graphen als Minoren enthält, die in der Graphentheorie von besonderer Bedeutung sind. Dazu gehören zum Beispiel die beiden *forbidden minors* der Menge aller planarer Graphen – der K_5 und der $K_{3,3}$. Abbildung 4.2 zeigt das Menü mit den beiden Spielmodi.



(a) Minoren im Standardmodus **(b)** Minoren im speziellen Modus

Abbildung 4.2: Übersicht über die Spielmodi

Das Puzzle-Framework, in welches *Minorfinder* eingebettet ist, bietet eine Reihe von vorgegebenen Funktionen, die entweder bereits über die Middleware implementiert sind oder im Backend noch vervollständigt werden können. Die wichtigsten Funktionen sind in der Menüleiste des *Game*-Menüs aus Abbildung 4.3 zu finden.



New – Neues Spiel starten

Restart – Jetziges Spiel neu starten

Specific – Spielparameter individualisieren

Load – Gespeicherten Spielstand laden

Save – Spielstand speichern

Undo – Zuletzt getätigten Spielzug rückgängig machen

Redo – Zuletzt rückgängig gemachten Spielzug wiederholen

Solve – Lösung anzeigen

Exit – Spiel verlassen

Abbildung 4.3: Hauptmenü mit verschiedenen Funktionen

Die *Solve*-Funktion ist die einzige nicht vorimplementierte Funktion aus dem *Game*-Menü, da das Ermitteln einer Lösung für jedes Puzzle anders funktioniert. Zudem können manche Spiele ohne großen zeitlichen Aufwand gar nicht gelöst werden. Abschnitt 4.7 erläutert, wie die Lösung eines *Minorfinder*-Spiels programmatisch bestimmt wird.

4.3 Spiele generieren

Der schwierigste Teil bei der Entwicklung eines neuen Puzzles liegt in der Regel beim Generieren eines zufälligen initialen Spielzustandes, der durch den Spieler mit Hilfe von validen Spielzügen in einen Lösungszustand transformiert werden kann. Dabei ist es vor allem wichtig, dass sich beim Kreieren neuer Startspielzustände über mehrere Spiele hinweg kein Muster abzeichnet, welches beim Spieler zu einem Lerneffekt führen und ihm somit das Finden einer Lösung deutlich erleichtern könnte.

Beim Erstellen eines zufälligen Startspielzustandes für *Minorfinder* habe ich mir den in Abschnitt 3.1 beschriebenen Umstand zu Nutze gemacht, dass man aus einem Graphen einen anderen Graphen erzeugen kann, der den ersten als Minor enthält. Man generiert also zunächst einen neuen Minor, aus dem man dann in mehreren Schritten den eigentlichen Spielgraphen erzeugt. Das Vorgehen bei der Spielgenerierung ist für die beiden Spielmodi – den Standardmodus und den speziellen Modus – bis auf wenige Ausnahmen identisch. Als erstes werden Koordinaten für die Knoten des Minors berechnet. Die Knoten des Minors werden stets so in einem Kreis angeordnet, dass sie ausnahmslos den selben Abstand zueinander haben. So wird sichergestellt, dass die Knoten des später aus dem Minor entstehenden Spielgraphen gleichmäßig über das gesamte Feld verteilt werden. Eine Ausnahme stellt der $K_{3,3}$ im speziellen Modus dar, der nach dem Generieren des Spielgraphen in zwei Reihen mit jeweils drei Knoten dargestellt wird. Bei einer rein zufälligen Anordnung der Minorknoten kann es dagegen vorkommen, dass sich alle Knoten des Spielgraphen in einem bestimmten Bereich des Feldes befinden; dann müsste der Spieler sich diesen zuerst zurecht legen, bevor er mit dem Spielen beginnen kann.

Anschließend wird der Minor durch das Hinzufügen von Kanten zwischen seinen Knoten vervollständigt. Im Standardmodus haben Minoren entweder vier, fünf oder sechs Knoten. Kanten werden zufällig hinzugefügt. Der spezielle Modus hingegen beinhaltet als Minoren die beiden *Kuratowski-Graphen* K_5 und $K_{3,3}$. In diesem Modus ist das Hinzufügen von Kanten statisch, da die beiden Graphen fest definiert sind. Einen möglichst zufälligen Graphen zu erzeugen, der jedoch trotzdem immer einen gegebenen Minor enthält, stellte sich als schwieriger heraus als ursprünglich angenommen, weshalb ich in diesem Abschnitt näher darauf eingehen werde. Der zugrundeliegende Algorithmus hat sich während der Entwicklung des Spiels mehrmals geändert.

Die erste Implementierung arbeitete wie folgt: Alle Knoten wurden zuerst einer Liste hinzugefügt und nach ihrem Grad, d. h. nach der Anzahl der mit ihnen inzidenten Kanten sortiert. Das Hinzufügen von Kanten erfolgte so, dass in jedem Durchlauf der Knoten mit dem höchsten Grad aus der Liste entfernt wurde und so lange Kanten zwischen diesem und dem Knoten mit dem geringsten Grad, zu dem noch keine Kante existierte, hinzufügt wurden, bis er einen fest definierten maximalen Grad erreicht hatte oder bis die gesamte Knotenliste einmal durchlaufen wurde. Vor dem Erstellen jeder Kante wurde immer überprüft, ob diese eine andere Kante schneidet – anfangs wollte ich den Spielgraphen planar haben, um das Puzzle nicht künstlich schwerer zu machen – oder ob sie einen Knoten schneidet. Wenn eine der beiden Bedingungen zutraf, wurde die Kante verworfen. Die eben beschriebene Prozedur wurde so lange wiederholt, bis lediglich ein Knoten in der Liste übrig war und keine weiteren Kanten mehr hinzugefügt werden konnten. Diese Methode ist im Gegensatz zur jetzigen – mit einer mittleren Laufzeit $\in O(\frac{n^3 \times \bar{m}}{2})$, wobei n die Anzahl der Knoten und $\bar{m}$ die mittlere Anzahl der Kanten darstellt – wesentlich effizienter.

Jedoch ist die Verteilung der Kanten bei dieser Umsetzung nicht wirklich zufällig; das Muster, das sich abzeichnet, ist immer dasselbe: Der Knoten, der im ersten Durchlauf aus der Liste genommen wird, besitzt am Ende einen deutlich höheren Grad, als alle nachfolgenden Knoten.

Deshalb habe ich den Algorithmus so verändert, dass Kanten komplett zufällig hinzugefügt werden. Zudem ist der jetzige Algorithmus dadurch flexibler, dass nun zwei Teilbereiche der Knotenliste angegeben werden können, zwischen deren Knoten Kanten entstehen sollen. Des Weiteren kann ein maximaler Grad angegeben werden, der die Knoten in der Anzahl der zu ihnen inzidenten Kanten einschränkt. Der Algorithmus speichert für jeden Knoten aus dem einen Teilbereich alle möglichen Kanten, sprich alle Knoten aus dem anderen Teilbereich, in einer Liste. Die Knoten des ersten Teilbereichs werden ebenfalls alle in einer Liste abgelegt. Anschließend verfährt der Algorithmus so, dass er sich immer wieder den Knoten mit dem geringsten Grad aus der Liste der Knoten des ersten Teilbereichs holt und mit Hilfe eines Zufallsgenerators einen Knoten aus der dazugehörigen Liste seiner möglichen adjazenten Knoten aus dem zweiten Teilbereich auswählt. Bevor eine neue Kante zwischen den ausgewählten Knoten erstellt werden kann, muss stets überprüft werden, ob es sich um eine valide Kante handelt. Dies ist zum Beispiel nicht der Fall, wenn die Kante eine Schlinge oder Mehrfachkante ist oder einen Knoten schneidet. Ist die Kante valide, kann sie erstellt und der Kantenliste hinzugefügt werden. Nun wird der Knoten aus dem zweiten Teilbereich, aus der Liste der möglichen adjazenten Knoten zum Knoten aus dem ersten Teilbereich, entfernt. Ist diese Liste leer, wird der Knoten aus dem ersten Teilbereich, aus der Liste aller Knoten aus dem ersten Teilbereich, ebenfalls entfernt. Die beschriebene Prozedur wird so lange wiederholt, bis das Hinzufügen von weiteren Kanten für keinen Knoten aus dem ersten Teilbereich mehr möglich ist. Die Laufzeit dieser Methode ist wesentlich höher, als die der vorherigen Methode; für kleine Graphen macht das jedoch keinen spürbaren Unterschied. Außerdem zeichnet sich kein wiederkehrendes Muster in Bezug auf die Verteilung der Kanten ab.

Nachdem das Hinzufügen von Kanten zum Minor abgeschlossen ist, werden seine Knoten $v_i \in V,\ i \in \{1, 2, ..., n\}$ durch Graphen G_{v_i} ersetzt, die wiederum Teilgraphen des Spielgraphen sind. Im Folgenden werde ich von den Graphen G_{v_i} auch als Cluster des Spielgraphen sprechen. Zum Generieren dieser Cluster wird für jeden Minorknoten das Minimum des Abstandes zum Rand des Felds und des Abstandes zu seinem nächsten Nachbar berechnet. In diesem Fall ist mit Nachbar nicht zwangsweise ein adjazenter Knoten gemeint. Mit Hilfe der Abstände kann dann für jeden der Cluster G_{v_i} der Radius eines Kreises – mit dem dazugehörigen Minorknoten v_i als Mittelpunkt – berechnet werden, auf dem dessen Knoten platziert werden sollen. Dadurch wird einerseits vermieden, dass sich die Cluster überlagern; andererseits werden ihre Knoten dadurch besser auf das Feld verteilt. Nun werden den Knoten der Cluster Koordinaten zugewiesen. Die Koordinaten liegen jeweils in gleichen Abständen auf den eben bestimmten Kreisen. Damit die Cluster unterschiedliche Formen annehmen, werden immer doppelt so viele Koordinaten berechnet, wie diese Knoten besitzen; die Anzahl von Knoten unterscheidet sich zudem von Cluster zu Cluster. Aus den berechneten Koordinaten werden dann zufällig genau so viele ausgewählt, dass jeder der Knoten eines jeden Clusters ein Koordinatenpaar, bestehend aus einer x- und einer y-Koordinate, erhält. Zuletzt werden die Cluster G_{v_i} um einen zufälligen Winkel $\phi \in [0, \frac{\pi}{|V_{G_{v_i}}|})$ gedreht; $V_{G_{v_i}}$ bezeichnet dabei die Knotenmenge eines Clusters.

Im Anschluss werden alle Kanten $\{v_k, v_l\},\ k, l \in \{1, 2, ..., n\}$ durch Kanten $\{G_{v_k}, G_{v_l}\}$ ersetzt, d.h. jede der Kanten des Spielgraphen, die zwei Cluster G_{v_i} miteinander verbindet, ist das Äquivalent einer Minorkante. Damit entspricht unser Spielgraph im Moment – wie in Abschnitt 3.1 beschrieben – einem Graphen aus der Klasse IG mit G als Minor.

Um dem Spieler die Lösung nicht zu offensichtlich zu präsentieren, wird der Spielgraph um überflüssige Knoten und Kanten erweitert. Die Knoten werden zufällig im Feld platziert, wobei stets sichergestellt wird, dass sie weder andere Knoten überlagern noch von bestehenden Kanten geschnitten werden. Die Kanten werden dem Graphen ebenfalls zufällig hinzugefügt – mit der selben Methode, mit der auch die Kanten des Minors generiert werden. Der dadurch entstandene Graph enthält den *IG* als Teilgraphen; somit ist *G* Minor des Spielgraphen.

Um die Performance und Effektivität der beschriebenen Ansätze bei der Spielgenerierung zu überprüfen, habe ich sie einigen Benchmarktests unterzogen. Aus der Sicht eines Spielers stellt die Spielgenerierung eine elementare Funktion dar, weshalb ich sie lediglich als Ganzes getestet habe. Eine Anforderung an die Spielgenerierung lautet, dass die erstellten Spiele möglichst verschieden sein sollen, damit Spieler stets einer neuen Herausforderung gegenübergestellt werden. Deshalb habe ich über mehrere Spielgenerierungen hinweg gemessen, wie viele Transformationen nötig gewesen wären, um den alten in den neuen Spielgraphen umzuwandeln. Das verwendete Maß ähnelt der in Abschnitt 3.3 beschriebenen *Graph Edit Distance*, beinhaltet jedoch keine Knoten- bzw. Kantensubstitutionen. Stattdessen wird die Positionsveränderung der Knoten bei der Berechung der Transformationskosten miteinbezogen, da optische Unterschiede – in diesem speziellen Fall – eine genauso wichtige Rolle spielen.

Die Transformationskosten habe ich folgendermaßen festgelegt:

- Knoten hinzufügen oder löschen – **1**
- Kanten hinzufügen oder löschen – **1**
- Knoten verschieben – **1**

Die Summe der Transformationskosten entspricht also der Summe der Tranformationen selbst. Das Ergebnis ist jeweils der Durchschnitt über 100 Spielgenerierungen hinweg für alle möglichen Variationen der Spielparameter.

	Parameter				
#	**MOD**	**SG**	**MIN**	**Iterationen**	**Ø**
0	Default	15	4	100	60
1	Default	19	5	100	82
2	Default	23	6	100	105
3	Special	19	5	100	83
4	Special	23	6	100	105

MOD – Spielmodus, entweder *Default* oder *Special*

SG – Anzahl Knoten Spielgraph

MIN – Anzahl Knoten Minor

Tabelle 4.1: Graphähnlichkeit der Spielgraphen zwischen zwei Spielgenerierungen, gemessen in Transformationskosten in Abhängigkeit der Spielparameter

Die Tabelle 4.1 zeigt einen Anstieg der benötigten Operationen proportional zur Knotenanzahl der Spielgraphen. Für den kleinsten Graphen liegt die Summe der Umwandlungskosten im Durchschnitt bei 60; beim größten Graphen sind es dagegen knapp über 100 Transformationen.
Selbstverständlich könnte man die Transformationskosten durch das Hinzunehmen der Operationen Kanten- und Knotensubstitution optimieren. Da der Spielgraph jedoch nicht als gewöhnlicher Graph zu betrachten ist und da bekannte effiziente Algorithmen zur Berechnung der *Edit Distance* diese ebenfalls nur Abschätzen, habe ich die Operationen bewusst weggelassen. Das Ergebnis zeigt in jedem Fall, dass die generierten Graphen sich auf dem Papier deutlich unterscheiden und Spieler somit mit großer Wahrscheinlichkeit nicht zweimal dem gleichen Spielgraphen gegenübergestellt werden.

Neben der Anforderung bezüglich der Graph-Ähnlichkeit besteht natürlich auch die Anforderung, dass Spiele möglichst effizient und ohne merkbare Verzögerung generiert werden sollen. Zu diesem Zweck habe ich die Laufzeit der Spielgenerierung für alle Variationen der Spielparameter über 100 Iterationen hinweg gemessen und daraus ebenfalls den Durchschnitt berechnet.

	Parameter				
#	**MOD**	**SG**	**MIN**	**Iterationen**	**Ø in µs**
0	Default	15	4	100	686,03
1	Default	19	5	100	746,89
2	Default	23	6	100	1014,53
3	Special	19	5	100	708,87
4	Special	23	6	100	1143,04

Tabelle 4.2: Laufzeit der Spielgenerierung, gemessen in Mikrosekunden in Abhängigkeit der Spielparameter

Genau wie die Transformationskosten steigt auch die Laufzeit der Spielgenerierung mit höherer Knotenanzahl, wie in Tabelle 4.2 zu sehen ist. Die Spielgenerierung ist nach der integrierten *Solve*-Funktion, die in Abschnitt 4.7 beschrieben wird, der aufwändigste Algorithmus, den ich für *Minorfinder* implementieren musste. Ihre Laufzeit liegt jedoch im Mikrosekundenbereich und ist – zumindest für Graphen dieser Größe – vernachlässigbar.

4.4 Aufbau eines Spiels

Im Anschluss an das Generieren eines neuen Spielzustandes werden die Knoten und Kanten des Minors bzw. des Spielgraphen in eine Zeichenfolge codiert. Die Codierung eines Graphen beinhaltet für Knoten jeweils ihre ID, ihre Koordinaten und ihren Grad und für Kanten jeweils die IDs der mit ihnen inzidenten Knoten. Die IDs der Knoten entsprechen ihrer Position in der Knotenliste. Des Weiteren wird für jeden Cluster des Spielgraphen seine Größe codiert, da diese nicht einheitlich ist. Außer der Zeichenfolge für den Spielzustand gibt es noch eine weitere Zeichenfolge, in die für die automatische Lösungsfunktion des Puzzle-Backends relevante Informationen geschrieben werden können. Die Spielzustandscodierung wird an die Middleware weitergegeben, welche das Backend daraus wiederum einen initialen Spielzustand generieren lässt.

Der Spieler kann die Spielzustandscodierung auch über die *Specific*-Funktion im Menü aus Abbildung 4.3 individuell anpassen. Die Middleware nutzt dann verschiedene Validierungsfunktionen des Backends um die eingegebenen Spielparameter bzw. den manipulierten Startspielzustand zu überprüfen.

Ein Spielzustand umfasst zwei Graphen, den Minor und den Spielgraphen sowie Informationen über den Zustand des Spiels, wie beispielsweise ob es bereits gelöst wurde oder ob der Spieler von der *Solve*-Funktion Gebrauch gemacht hat. Ein Graph besteht wiederum aus Listen all seiner Knoten, seiner sichtbaren Knoten sowie Kanten und der Größen seiner Cluster. Die Spielgraphen verschiedener Spielzustände unterscheiden sich effektiv nur in den Listen ihrer sichtbaren Knoten bzw. Kanten. Die Clustergrößen sind im Moment nur für die automatische Lösungsfunktion von Bedeutung; dennoch sind sie Teil des Spielzustandes, da sie eventuell auch in einem anderen Kontext relevant werden können. Die Middleware der Spielesammlung verwaltet eine Liste von Spielzuständen, die mit jedem weiteren Spielzug erweitert wird und es dem Spieler ermöglicht, zwischen mehreren Spielzuständen hin- und herzuspringen.

4.5 Spielzüge ausführen

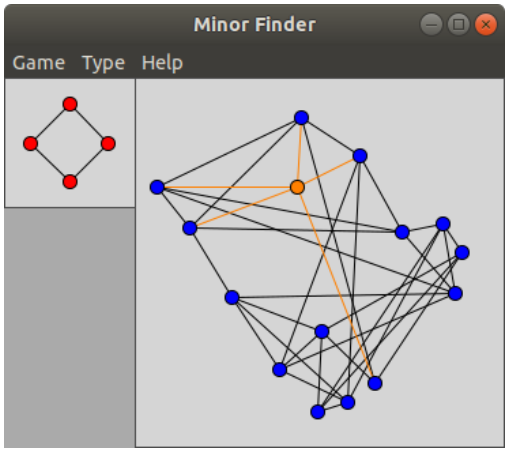
Um den Spielgraphen in einen zu dem Minor isomorphen Graphen zu transformieren, sind verschiedene Spielzüge notwendig. Alle validen Spielzüge sind per Definition von Minoren gegeben; dazu gehören beispielsweise das Löschen von Kanten bzw. Knoten sowie das Kontrahieren von Kanten. Sei ein Spiel mit einem Minor G und einem initialen Spielgraphen S gegeben. Eine Möglichkeit, dieses Spiel zu lösen, besteht darin, zunächst durch das Entfernen von Knoten bzw. Kanten S in einen Graphen der Klasse IG umzuwandeln; aus diesem kann wiederum durch Kantenkontraktion ein Graph erzeugt werden, der isomorph zu G ist. Tatsächlich ist die Reihenfolge, in der die Spielzüge getätigt werden, aber unwichtig, da es genauso möglich ist, in einem ersten Durchlauf einen Minor H aus S zu extrahieren, der – genau wie $S - G$ als Minor enthält; anschließend kann in einem zweiten Durchlauf aus H ein zu G isomorpher Graph erzeugt werden. In beiden Fällen erhält man am Ende eine korrekte Lösung. Neben dem Löschen ist auch das Verschieben von Knoten erlaubt, da die Einbettung von S aufgrund der Isomorphie-Eigenschaft keine Rolle für die Lösungsfindung spielt. Im Gegensatz zu den anderen Spielzügen dient das Verschieben von Knoten jedoch ausschließlich der Übersichtlichkeit, bringt einen Spieler der Lösung aber nicht näher.

Das Löschen von Knoten bzw. Kanten erfolgt durch einen Klick mit der rechten Maustaste auf das zu löschende Objekt; dieses leuchtet dann orange auf, solange der Klick gehalten wird. Bei der Löschung eines Knotens werden mit ihm all seine inzidenten Kanten gelöscht und der Grad aller involvierten Knoten wird um eins verringert; selbiges gilt natürlich für die inzidenten Knoten bei einer Kantenlöschung. Das Speichern des Knotengrades ist besonders wichtig, da dieser benötigt wird, um nach jedem Spielzug überprüfen zu können, ob bereits eine Lösung gefunden wurde. Der wichtigste Spielzug ist jedoch die Kontraktion von Kanten. Diese ist eng verbunden mit dem Verschieben von Knoten. Knoten können mit einem einfachen Rechtsklick ausgewählt, hin- und hergezogen und an einer anderen Stelle wieder abgelegt werden. Zieht man einen Knoten auf einen benachbarten, in diesem Fall adjazenten Knoten, so vergrößert sich dessen Radius; lässt man den bewegten Knoten dann los, wird die Kante, die die beiden Knoten verbindet, kontrahiert. Dabei

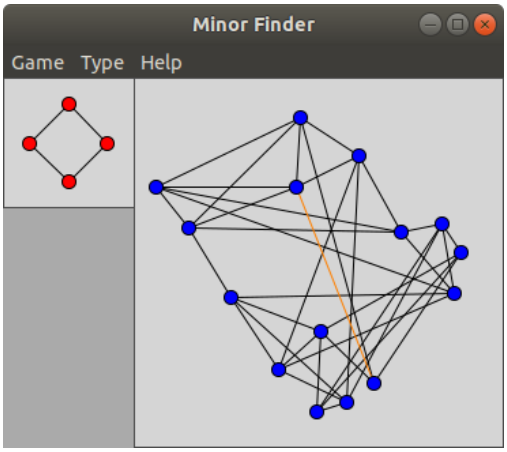
wird der verschobene Knoten gelöscht; dem anderen Knoten werden alle zum gelöschten Knoten inzidenten Kanten übergeben. Bei der Kontraktion einer Kante müssen schließlich alle übrigen Kanten beibehalten werden, insofern sie nicht bereits vorhanden sind.

Der Grad des verbleibenden Knotens erhöht sich also um die Zahl der Kanten, die ihm übertragen werden. Ein Spielzug kann abgebrochen werden, indem die gedrückte Maus aus dem Feld, in dem sich der Spielgraph befindet, bewegt und anschließend losgelassen wird. Abbildung 4.4 zeigt eine Übersicht der in diesem Abschnitt beschriebenen Spielzüge.

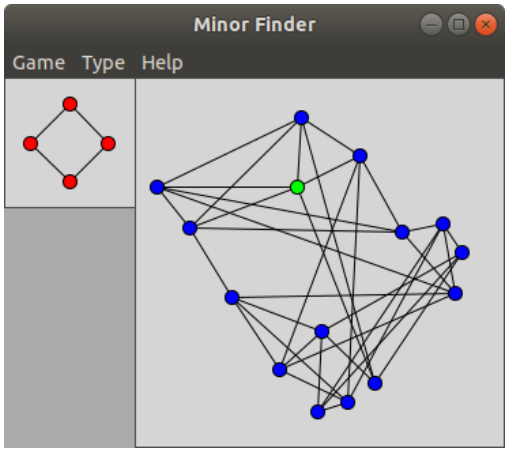
Um überhaupt feststellen zu können, ob ein Spielzug getätigt wird, müssen Mausklicks auf Knoten bzw. Kanten detektiert werden können; zu diesem Zweck werden Heuristiken verwendet. Die Heuristik, die Mausklicks auf Knoten feststellt, nimmt sich den euklidischen Abstand zwischen Knoten und Mausposition zur Hilfe. Ist der für einen Knoten größer als ein gewisser Schwellenwert, wird dieser ignoriert; von den übrigen Knoten wird der ausgewählt, der den geringsten euklidischen Abstand zur Mausposition hat. Weitaus schwieriger ist es, mit äußerster Präzision, einen Klick auf eine bestimmte Kante zu detektieren. Sei T ein Schwellenwert, sei E die Menge aller Kanten, seien e_{0_a} und e_{0_b} die beiden zu einer Kante $e_0 \in E$ inzidenten Knoten, sei p die Mausposition und sei $h : E \to \mathbb{R}$ die Heuristik zum detektieren von Mausklicks auf Kanten. Dann wird nach folgendem Kriterium entschieden, ob die Kante e_0 angeklick wurde: $h(e_0) := \overline{e_{0_a}p} + \overline{pe_{0_b}} - \overline{e_{0_a}e_{0_b}} < T \wedge \forall e \in E : h(e_0) \leq h(e)$. Trifft dieses Kriterium auf mehrere Kanten zu, wird aus diesen Kanten zufällig eine ausgewählt.



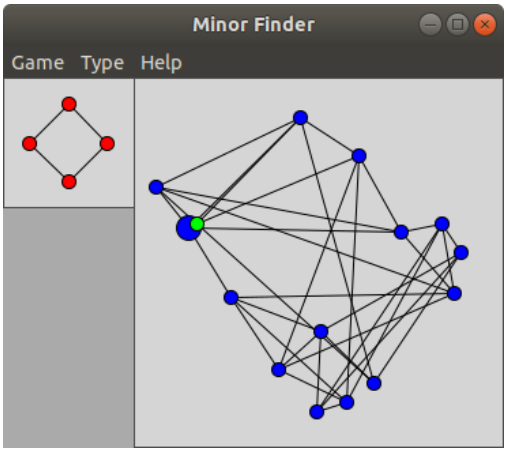
(a) Graph mit zu löschendem Knoten



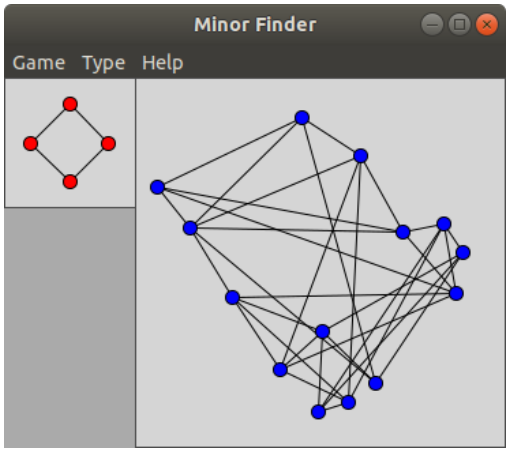
(b) Graph mit zu löschender Kante



(c) Graph mit zu verschiebendem Knoten



(d) Graph mit zu kontrahierender Kante



(e) Graph mit kontrahierter Kante

Abbildung 4.4: Übersicht über mögliche Spielzüge

4.6 Lösungzustände erkennen

Mithilfe der oben beschriebenen Spielzüge kann jedes wie in Abschnitt 4.3 generierte Spiel gelöst werden. Nach jedem getätigten Spielzug muss deshalb überprüft werden, ob der Spieler eine korrekte Lösung gefunden hat. Wenn dies der Fall ist, soll ihm signalisiert werden, dass er erfolgreich war. Das Ziel des Spiels wurde bereits in Abschnitt 4.2 erklärt: Der Spieler soll den Spielgraphen in einen zu dem Minor isomorphen Graphen transformieren.

Es ist also notwendig, festzustellen, ob zwei Graphen isomorph sind. In Abschnitt 3.2 habe ich bereits über Alternativen zu einem reinen Bruteforce-Ansatz gesprochen, wenngleich kein Algorithmus bekannt ist, der das Graphisomorphie Problem in Polynomialzeit lösen würde. Die Graphen, die bei *Minorfinder* auf Isomorphie getestet werden, sind mit vier bis sechs Knoten zwar vergleichsweise klein, jedoch müssten bei einem naiven Ansatz für zwei Graphen mit jeweils n Knoten im Worstcase-Szenario $n!$ Permutationen geprüft werden. Bei dem Minor mit sechs Knoten sind das bereits 720 Permutationen und die Zahl der zu prüfenden Permutationen steigt exponentiell mit der Zahl der Knoten.

Deshalb habe ich den Algorithmus zum erkennen von Lösungszuständen nach dem in Abschnitt 3.2 beschriebenen Prinzip mit *Colourings* implementiert, wobei ich das initiale Colouring nach Möglichkeit so wähle, dass es mehr als eine Zelle besitzt. Zudem prüfe ich die generierten Permutationen nicht gegen den Spielgraphen selbst, da wir nicht an dessen Automorphismengruppe interessiert sind, sondern gegen den Graphen, zu dem der Spielgraph isomorph sein soll – dem Minor. Zunächst wird die Anzahl der Knoten des Minors mit der des Spielgraphen verglichen. Nur wenn beide Graphen gleich viele Knoten besitzen, ist es überhaupt möglich, dass sie isomorph sind. Anschließend wird die Verteilung der Kanten auf die Knoten der Graphen verglichen. Beide Graphen sollten jeweils gleich viele Knoten mit dem selben Grad haben. Sind beide Bedingungen erfüllt, kann die eigentliche Suche nach einer Permutation, die ein Isomorphismus zwischen den Graphen ist, gestartet werden; dazu wird wie in Algorithmus 4.1 beschrieben, vorgegangen. Zunächst wird die Wurzel des Suchbaums initialisiert. Dabei werden genau die Knoten mit der selben Farbe «eingefärbt», die auch den selben Grad haben; dadurch bleibt der Suchbaum meist wesentlich kleiner, als bei einer Bruteforce-Lösung. Ist die Wurzel initialisiert, wird die Baumsuche gestartet. Da lediglich Blätter – bei denen jede Zelle genau einen Knoten enthält und jeder Knoten genau einer Zelle zugeordnet ist – als Isomorphismus in Frage kommen, werden Colourings jeweils so lange expandiert, bis sie nicht mehr verfeinert werden können. Beim expandieren wird immer genau die Zelle aus den Zellen mit mehr als einem Element verfeinert, deren Knoten den geringsten Grad haben. Für jeden Knoten in dieser Zelle entsteht dabei ein neues, feineres Colouring, das ein Kind des aktuellen Colourings ist, wobei der jeweilige Knoten mit einer neuen Farbe «eingefärbt» wird.

Die dabei entstandenen Colourings besitzen also jeweils eine Zelle mehr als ihr gemeinsames Eltern-Colouring. Wenn ein Blatt, sprich eine Permutation erreicht wurde, wird für diese überprüft, ob sie ein Isomorphismus ist. Ist das der Fall, wird der Spieler durch ein kurzes Aufblitzen des Bildschirms im Bereich des Spielfensters darauf hingewiesen, dass er das Puzzle gelöst hat.

Algorithmus 4.1 Algorithmus um zwei Graphen auf Isomorphie zu testen

```
function TestIsomorphy(G, H)
    V_G ← G.vertices
    V_H ← H.vertices

    if |V_G| ≠ |V_H| then
        return false
    end if
    SortByDegree(V_G)
    SortByDegree(V_H)
    if not CompareByDegree(V_G, V_H) then
        return false
    end if

    r ← InitRootColouring(V_H)
    // The vertices of V_H are distributed over the cells of r such that vertices with equal degree go
       into the same cell
    stack ← NewStack()
    stack.push(r)
    while not stack.isempty() do
        c ← stack.pop()
        p_{H→G} ← ExpandColouring(c)
                                  // p_{H→G} denotes a vertex permutation of V_H against V_G
        if p_{H→G} ≠ nil ∧ CheckPermutation(G, H, p_{H→G}) then
            return true
        else
            for all c_child ∈ c.children do
                stack.push(c_child)
            end for
        end if
    end while

    return false
end function

function ExpandColouring(c)
    if c.isleaf() then
        p ← []
        for cell ∈ c.cells do
            p.pushback(cell[0])
                          // Since c is a leaf all its cells must only consist of a single vertex
        end for
        return p
    else
        RefineColouring(c)
              // For every vertex in the lowest degree cell of c with size > 0 create a new child
                 colouring by putting the respective vertex into a new cell
        return nil
    end if
end function
```

Gerade da das Graphisomorphie Problem ein aktuelles Forschungsproblem ist, ist hier natürlich besonders interessant zu sehen, wie effizient der von mir implementierte Algorithmus arbeitet, wenngleich die Minoren, die auf Isomorphie getestet werden, nur wenige Knoten besitzen. Die Laufzeit wurde genau wie bei der Spielgenerierung für alle Variationen der Spielparameter über 100 Iterationen hinweg gemessen und daraus der Durchschnitt berechnet. Zusätzlich zur Durchführung des Algorithmus mit der Knotengrad-Heuristik, bei der zu Beginn jeweils Knoten mit gleichem Grad derselben Zelle der Wurzel des Suchbaums hinzugefügt werden, habe ich den Algorithmus, um die Ergebnisse besser einordnen zu können, auch in einer Bruteforce-Variante durchgeführt, bei der die Wurzel aus lediglich einer Zelle besteht, die alle Knoten beinhaltet.

	Parameter					
#	**MOD**	**SG**	**MIN**	**Heuristik**	**Iterationen**	**Ø in µs**
0	Default	15	4	Keine	100	16,10
1	Default	15	4	Knotengrad	100	16,50
2	Default	19	5	Keine	100	29,22
3	Default	19	5	Knotengrad	100	22,26
4	Default	23	6	Keine	100	38,69
5	Default	23	6	Knotengrad	100	37,91
6	Special	19	5	Keine	100	27,30
7	Special	19	5	Knotengrad	100	27,89
8	Special	23	6	Keine	100	36,33
9	Special	23	6	Knotengrad	100	36,92

Tabelle 4.3: Laufzeit des Isomorphietests, gemessen in Mikrosekunden in Abhängigkeit der Spielparameter

Die Ergebnisse in Tabelle 4.3 zeigen einen leichten Vorteil der Variante des Algorithmus mit Knotengrad-Heuristik gegenüber der Bruteforce-Variante. Lediglich für das Spielparameter-Tripel *(Default, 19, 5)* ist ein deutlicher Unterschied bei der Laufzeit zu erkennen; dabei könnte es sich jedoch um einen Ausreißer handeln. Um das herauszufinden, habe ich einen weiteren Test durchgeführt, bei dem ich überprüft habe, wie viele *Colourings* der Algorithmus im Durchschnitt expandiert bzw. jede wievielte gefundene Permutation ein Isomorphismus ist. Den Test habe ich lediglich für das Parametertripel *(Default, 19, 5)* durchgeführt, da hier – wie oben erwähnt – der größte Unterschied bei der Performance zu verzeichnen war. Da der Minor mit vier Knoten immer so generiert wird, dass all seine Knoten zwei inzidente Kanten besitzen und somit – unabhängig von der verwendeten Heuristik – der selben Zelle der Wurzel des Suchbaums hinzugefügt werden, kann man sich an dieser Stelle den Test sparen; für das Tripel *(Default, 15, 4)* ist der Suchbaum für beide Ausführungen des Algorithmus identisch.

#	Heuristik	Iterationen	Ø COL	Ø PM
0	Keine	20	6,6	2,05
1	Knotengrad	20	5,6	2,05

COL – Anzahl explorierter Colourings

PM – Anzahl überprüfter Permuationen

Tabelle 4.4: Expandierte Kanten und geprüfte Permutationen des Isomorphietests für das Spielparameter-Tripel *(Default, 19, 5)*

Der Test zeigt, dass für Graphen mit fünf Knoten die Variante mit Knotengrad-Heuristik im Durchschnitt ein Colouring weniger expandiert, auch wenn beide Varianten die gleiche Anzahl von Permutationen überprüfen, bevor sie ein Isomorphismus finden. Dies könnte erklären, weshalb die Bruteforce-Variante hier bei der Laufzeit deutlich schlechter abschneidet. Der Minor mit fünf Knoten im Standardmodus hat zudem eine geringere Kantendichte als der Minor mit sechs Knoten im Standardmodus und die beiden Minoren im speziellen Modus, wodurch die Chance, eine Permutation zu finden, die ein Isomorphismus ist, verringert wird. Das würde wiederum erklären, weshalb lediglich für das Parametertripel *(Default, 19, 5)* ein Performance-Unterschied zu sehen ist.

4.7 Lösungen algorithmisch bestimmen

Wenn ein Spieler Hilfe beim Lösen eines Spiels benötigt, kann er von der *Solve*-Funktion Gebrauch machen. Diese bestimmt algorithmisch und ohne jegliche Hilfe des Spielers eine Folge von Spielzügen, die zu einer Lösung führt.

Befindet sich das Spiel in seinem Anfangszustand oder in einem Zustand, bei dem lediglich die Cluster des Spielgraphen durch Kontraktion oder Löschung ihrer Kanten verkleinert bzw. Knoten die zu keinem Cluster gehören – mitsamt ihrer inzidenten Kanten – gelöscht wurden, kann die Lösung in der Regel problemlos bestimmt werden. Dazu muss das «Aufblasen» des Graphen, das bei der Spielgenerierung geschieht, rückgängig gemacht werden. Zuerst müssen überflüssige Knoten und Kanten, die in den letzten beiden Schritten der Spielgenerierung hinzukommen, entfernt werden. Diese können ganz einfach dem Ende der Knotenliste des Spielgraphen entnommen werden. Außerdem müssen alle zu den zu löschenden Knoten inzidenten Kanten ebenfalls gelöscht werden. Diese sind in eine bei der Spielgenerierung erstellten Zeichenfolge codiert, die für das Bestimmen einer Lösung relevante Informationen beinhaltet. Nachdem überflüssige Knoten bzw. Kanten entfernt wurden, sollte der Spielgraph einem Graphen der Klasse IG – mit Minor G – entsprechen. Ist der Spielgraph ein IG, so kann er durch Kantenkontraktion in den Minor G überführt werden; hierzu müssen lediglich die Kanten der einzelnen Cluster so oft kontrahiert werden, bis jedes der Cluster nur noch aus einem einzigen Knoten besteht. Der Spielgraph sollte dann isomorph zum Minor sein, womit eine korrekte Lösung bestimmt wurde.

Wenn ein Spieler jedoch nicht nach dem Musterlösungsweg vorgeht, schlägt die eben beschriebene Lösungsmethode fehl, da eventuell Kanten kontrahiert wurden, die als Äquivalent einer Minorkante fungieren oder die einen Knoten eines Clusters mit einem Knoten außerhalb dieses Clusters verbinden. Es gibt zwar Algorithmen, die für zwei Graphen in polynomieller Zeit – der Algorithmus von Neil Robertson und P. D. Seymour hat eine kubische Laufzeit – feststellen können, ob der eine den anderen als Minor enthält; diese Algorithmen schließen jedoch das Erzeugen des Minors aus dem anderen Graphen nicht mit ein und sind praktisch nicht umsetzbar [RS95]. Nach aktuellem Stand der Forschung ist ein einfacher Bruteforce-Ansatz somit die beste Methode, um den Minor aus einer beliebigen Mutation des Spielgraphen zu extrahieren.

Der von mir entwickelte Algorithmus, der das Problem in Angriff nimmt, geht rekursiv alle überhaupt möglichen Spielzustände durch, bis er eine Lösung gefunden hat oder aber feststellt, dass keine existiert. Aufgrund der Beschaffenheit des Spielgraphen genügt es, lediglich auf seinen Kanten zu operieren. Der Algorithmus iteriert also auf jeder Rekursionsebene über alle Kanten des Spielgraphen; auf jede Kante wendet er dann die Spielzüge Kantenkontraktion bzw. Kantenlöschung an. Jeder getätigte Spielzug transformiert den aktuell betrachteten Spielzustand in einen neuen, für den ein weiterer rekursiver Aufruf gestartet wird. Dies wird so lange fortgesetzt, bis ein Lösungszustand gefunden wurde oder bis alle möglichen Spielzustände abgearbeitet wurden. Hat der Algorithmus einen Lösungszustand gefunden, gibt er eine Zeichenfolge zurück, in die die Spielzüge geschrieben werden, die zu diesem Zustand geführt haben.

In dieser Ausführung benötigt der Algorithmus, wie in Tabelle 4.5 bzw. Abbildung 4.5 zu sehen ist, trotz der geringen Größe der Minoren relativ lange, um eine Lösung zu ermitteln. Deshalb habe ich versucht, ihn durch Abbruchbedingungen und Heuristiken, die entscheiden, ob ein Spielzug ausgeführt werden soll oder nicht, zu verbessern. So wird auf jeder Rekursionsebene stets überprüft, ob der Spielgraph noch mehr Kanten bzw. gleich viele oder mehr Knoten als der gesuchte Minor hat. Wenn der Spielgraph weniger Kanten oder Knoten als der Minor besitzt, ist davon auszugehen, dass dieser nicht mehr in ihm enthalten ist; die Rekursion wird für diese spezielle Mutation des Spielgraphen abgebrochen. Wenn der Spielgraph gleich viele Kanten wie der Minor besitzt, wird hingegen direkt geprüft, ob die beiden Graphen isomorph sind. In den restlichen Fällen wird vor jeder Kantenlöschung überprüft, ob jeder der involvierten Knoten mindestens eine inzidente Kante mehr besitzt als der Knoten mit dem kleinsten Grad aller Knoten des Minors. Ist eine dieser Bedingungen nicht gegeben, wird der geplante Spielzug nicht durchgeführt. Algorithmus 4.2 beinhaltet den Pseudocode der überarbeiteten Version des Algorithmus.

Damit die Lösung über mehrere Spiele hinweg einheitlich präsentiert werden kann, erhalten die zur Lösung gehörigen Knoten dieselben Koordinaten wie ihre zugehörigen Minorknoten. Um die Knoten ihren jeweiligen Minorknoten zuordnen zu können, speichert die Methode aus Algorithmus 4.1, mit der der Minor und der Spielgraph auf Isomorphie geprüft werden, die gefundene Lösungspermutation.

Für den Bruteforce-Lösungsalgorithmus war es besonders interessant zu sehen, wann dieser an seine Grenzen stößt. Anders als bei den anderen Algorithmen kann seine Laufzeit je nach gegebenem Spielzustand sehr stark variieren; diese liegt, trotz der Verbesserungen, die ich am Algorithmus vorgenommen habe, im Worstcase-Szenario immerhin noch in $O((|E_P|! - |E_M|!) \times 2^{|E_P|-|E_M|})$. Je nachdem, wie viele Kanten mehr der Spielgraph im Gegensatz zum Minor besitzt, kann das Finden einer Lösung durchaus mehrere Sekunden, Minuten oder sogar Stunden dauern.

Algorithmus 4.2 Rekursiver Algorithmus zum Lösen eines Spiels

```
function SolveRecursive(s)
    moves ← {contract, delete}
                    // contract and delete are valid moves in any state s and for any edge e

    V_P ← s.playgraph.vertices
    V_M ← s.minor.vertices
    E_P ← s.playgraph.edges
    E_M ← s.minor.edges
    if |E_P| > |E_M| ∧ |V_P| ≥ |V_M| then
        for all e ∈ E_P do
            for all m ∈ moves do
                if m = delete then
                    if e_a.degree ≤ MinDegree(V_M) ∨ e_b.degree ≤ MinDegree(V_M) then
                        continue
                    end if
                end if
                s' ← m(s, e)
                if TestIsomorphy(s.minor, s'.playgraph) then
                    sol ← NewStack()
                    sol.push(m(s, e))
                    return sol
                else
                    sol ← SolveRecursive(s')
                    if sol ≠ nil then
                        sol.push(m(s, e))
                        return sol
                    end if
                end if
            end for
        end for
    end if

    return nil
end function
```

Da der Algorithmus nur dann Anwendung findet, wenn die Methode zum Lösen von Spielen mit Hilfe der Informationen aus der Spielgenerierung keine Lösung konstruieren kann, bin ich beim Testen des Bruteforce-Ansatzes wie folgt vorgegangen: Als erstes habe ich das aktuelle Spiel selbst gelöst. Aufgrund des zum Lösen nötigen Zeitaufwands habe ich den Test lediglich im Standardmodus mit dem Minor mit fünf bzw. dem dazugehörigen Spielgraphen mit 19 Knoten durchgeführt. Anschließend habe ich eine vorher festgelegte Anzahl von Spielzügen – Veränderungen an der Einbettung des Spielgraphen ausgenommen – rückgängig gemacht. So konnte ich sicherstellen, dass das Spiel in seinem jetzigen Zustand auch wirklich lösbar ist.

Dann habe ich den Algorithmus seine Arbeit machen lassen und die Laufzeit gemessen. Wenn nach zehn Minuten keine Lösung gefunden wurde, habe ich den aktuellen Testdurchlauf beendet und den nächsten gestartet. Um herauszufinden, wie stark sich die zusätzlichen Abbruchbedingungen auf die Laufzeit der Methode auswirken, habe ich den Test sowohl für die erste Implementierung als auch für die überarbeitete Implementierung durchgeführt. Tabelle 4.5 und Abbildung 4.5 beinhalten die Resultate des Laufzeittests.

			Ø in ms		**MED in ms**	
#	**Schritte zurück**	**N**	**U**	**I**	**U**	**I**
0	5	5	1330,68	0,124	1200,47	0,139
1	7	5	1673,92	0,207	759,99	0,182
2	10	5	66231,58	0,487	4844,79	0,278
3	15	5	-	25105,59	310040,3	1,461

N – Anzahl der Messwerte für gegebene Parameter

MED – Median von **N** Messwerten

U – Unverbesserte Version des Algorithmus

I – Verbesserte Version des Algorithmus

Tabelle 4.5: Laufzeit des Bruteforce-Lösungsalgorithmus, gemessen in Millisekunden in Abhängigkeit der Schritte bis zu einem Lösungszustand für das Spielparameter-Tripel *(Default, 19, 5)*

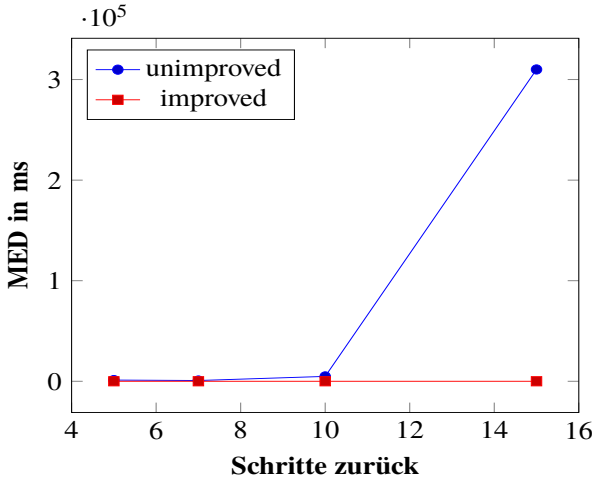


Abbildung 4.5: Anstieg der Laufzeit des Bruteforce-Lösungsalgorithmus, gemessen in Millisekunden in Abhängigkeit der Schritte bis zu einem Lösungszustand für das Spielparameter-Tripel *(Default, 19, 5)*

Wie zu erwarten sehen wir einen exponentiellen Anstieg der durchschnittlichen Laufzeit bzw. des Medians der Laufzeit, je weiter wir uns von einem Lösungszustand entfernen. Durch zusätzliche Abbruchbedingungen und Heuristiken konnte die Performance der *Solve*-Funktion jedoch deutlich gesteigert werden, sodass sie in den meisten Fällen weniger als eine Sekunde benötigt, um eine Lösung zu ermitteln. Dem Algorithmus kommt jedoch auch zugute, dass die Minoren alle eine relativ hohe Kantendichte besitzen, wodurch die Chance erhöht wird, eine Permutation zu finden, die ein Isomorphismus zwischen dem jeweiligen Minor und dem Spielgraphen ist.

5 Zusammenfassung und Ausblick

Der Fokus dieser Arbeit liegt auf der Umsetzung eines graphentheoretischen Spiels, bei dem Minoren interaktiv – durch Kontraktion von Kanten bzw. Löschung von Kanten und Knoten – in größeren Graphen gesucht werden können. Das Spiel mit dem Namen *Minorfinder* habe ich als Teil einer Sammlung kleiner Rätsel-Computerspiele des britischen Softwareentwicklers Simon Tatham realisiert. Im Zuge der Entwicklung stellten sich das Generieren zufälliger Spiele sowie das Erkennen und Ermitteln von Lösungen als die Kernprobleme heraus.

Zu Beginn der Spielgenerierung werden die Knoten des Minors erstellt. Anschließend vervollständigt ein von mir implementierter Algorithmus zum Generieren einer zufälligen Kantenmenge für einen Graphen den Minor. Zum Erzeugen des Spielgraphen konnte ich mir das in Abschnitt 3.1 beschriebene Prinzip zu Nutze machen, bei dem ein Graph so um zusätzliche Knoten und Kanten erweitert wird, dass er den ursprünglichen Graphen als Minor enthält. Auch hier wird zum Erstellen von Kanten die von mir geschriebene Funktion verwendet.

Für das Erkennen von Lösungen war es notwendig feststellen zu können, ob ein mutierter Spielgraph isomorph zu dem jeweils angezeigten Minor ist. Zu diesem Zweck habe ich einen *Colouring*-Algorithmus implementiert, der auf den theoretischen Grundlagen aus Abschnitt 3.2 basiert. Dieser wurde von mir sowohl in einer Bruteforce-Variante, als auch in einer Variante mit Knotengrad-Heuristik umgesetzt, wobei sich herausstellte, dass für kleine Graphen fast kein Unterschied bei der Performance erkennbar ist. Für große Graphen mit geringer Kantendichte kann die Laufzeit des Algorithmus durch die Verwendung der Knotengrad-Heuristik jedoch deutlich gesenkt werden.

Die *Solve*-Funktion ermöglicht das Ermitteln von Lösungen aus einem beliebigen Spielzustand heraus. Dies geschieht entweder, indem man genau umgekehrt wie bei der Spielgenerierung vorgeht oder wenn das nicht zu einer Lösung führen sollte, mithilfe eines rekursiven Bruteforce-Ansatzes, der alle überhaupt möglichen Spielzustände durchläuft, bis er eventuell eine Lösung findet. Dies kann je nach Größe des Suchraums, auf dem der Algorithmus operiert, wenige Millisekunden bis hin zu mehreren Minuten dauern. Durch zusätzliche Abbruchbedingungen sowie die Verwendung von Heuristiken, um zu entscheiden, ob eine Kante kontrahiert bzw. gelöscht werden soll, konnte die Laufzeit jedoch in den meisten Fällen auf unter eine Sekunde reduziert werden.

Bezüglich der Aufnahme des Spiels in die Spielesammlung habe ich den Kontakt mit Simon Tatham hergestellt. Herr Tatham hat die Grundidee des Spiels gelobt und war – bis auf Kleinigkeiten im User Interface, die ich umgehend überdacht und verbessert habe – zufrieden mit meiner Umsetzung.

Ausblick

Aufgrund der Einschränkungen der Algorithmen zum Erkennen und zum automatischen Ermitteln von Lösungen, bei der Performance, ist *Minorfinder* im Moment nur mit einer festgelegten Menge von Minoren bzw. Spielgraphen spielbar. Die Forschung zu den graphentheoretischen Problemen, die die beiden von mir implementierten Algorithmen lösen, ist immer noch im Gange. Da ich mich jedoch auf die Entwicklung des Spiels als Ganzes konzentriert habe, konnte ich nicht derart tief in die Materie gehen, wie es nötig wäre, um die Algorithmen für Graphen beliebiger Größe performant zu machen.

Die Methode zum Testen zweier Graphen auf Isomorphie könnte eventuell durch das Prunen von *Colourings* aus dem Suchbaum weiter verbessert werden [MP13]. Außerdem beschreibt [LCA14] einige Ansätze zum optimieren der Suche, wie beispielsweise *Early Automorphism Detection (EAD)*, die auch auf das Graphisomorphie Problem anwendbar sind. Die Heuristiken, die der Bruteforce-Lösungsalgorithmus im Moment verwendet, können lediglich entscheiden, wann es nicht sinnvoll ist eine Kante zu kontrahieren oder zu löschen, d.h. zu explorieren. Durch das Verwenden von Heuristiken, die in einem beliebigen Spielzustand entscheiden können, ob die Ausführung eines bestimmten Spielzugs wirklich Sinn macht, könnte die Performance der Methode nochmals wesentlich gesteigert werden.

Des Weiteren dient *Minorfinder* als Inspiration zur Implementierung weitere Puzzles mit (graphen-)theoretischem Hintergrund, mit dem Ziel über eine ganze Sammlung solcher Puzzles zu Verfügen. Eine solche Sammlung von Spielen, einschließlich *Minorfinder*, würde Simon Tatham gerne als gesonderte Kategorie in seine Spielesammlung aufnehmen; im Moment sind dafür jedoch noch zu wenige Spiele dieser Art vorhanden. Eine Idee für ein weiteres graphentheoretisches Spiel wäre zum Beispiel das Markieren eines Eulerkreises in einem gegebenen Graphen. Jedoch würden auch viele andere Konzepte aus der Graphentheorie oder Mathematik als Grundlage für ein solches Spiel in Frage kommen.

Literaturverzeichnis

[Bab16] L. Babai. „Graph Isomorphism in Quasipolynomial Time“. In: (Juni 2016), S. 684–697. DOI: https://doi.org/10.1145/2897518.2897542 (zitiert auf S. 19).

[GXTL09] X. Gao, B. Xao, D. Tao, X. Li. „A survey of graph edit distance“. In: (Jan. 2009), S. 113–129. DOI: https://doi.org/10.1007/s10044-008-0141-y (zitiert auf S. 20).

[LCA14] J. L. Lopez-Presa, L. F. Chiroque, A. F. Anta. „Novel Techniques to Speed Up the Computation of the Automorphism Group of a Graph“. In: *Journal of Applied Mathematics* 2014 (Juli 2014), S. 1–16. DOI: https://doi.org/10.1155/2014/934637 (zitiert auf S. 15, 42).

[MF20] S. Holderbach. *Minorfinder*. 2020. URL: https://github.com/elsamuray7/Minorfinder (zitiert auf S. 3, 13, 15).

[Moh06] B. Mohar. „What is a Graph Minor“. In: *Notices of the AMS* 53.3 (März 2006), S. 338 (zitiert auf S. 3, 18).

[MP13] B. D. McKay, A. Piperno. „Practical graph isomorphism, II“. In: *Journal of Symbolic Computation* 60 (Sep. 2013), S. 94–112. DOI: https://doi.org/10.1016/j.jsc.2013.09.003 (zitiert auf S. 15, 19, 42).

[PPC20] S. Tatham. *Simon Tatham's Portable Puzzle Collection*. 2004–2020. URL: https://www.chiark.greenend.org.uk/~sgtatham/puzzles/ (zitiert auf S. 3, 13, 15, 21).

[RS01] N. Robertson, P. Seymour. „Graph Minors. XX. Wagner's conjecture“. In: *Journal of Combinational Theory* (Feb. 2001), S. 325–357. DOI: https://doi.org/10.1016/j.jctb.2004.08.001 (zitiert auf S. 13, 18).

[RS95] N. Robertson, P. Seymour. „Graph Minors .XIII. The Disjoint Paths Problem“. In: *Journal of Combinatorial Theory, Series B* 63.1 (Jan. 1995), S. 65–110. DOI: https://doi.org/10.1006/jctb.1995.1006 (zitiert auf S. 36).

[Wag37] K. Wagner. „Über eine Eigenschaft der ebenen Komplexe“. In: *Mathematische Annalen* 114 (Jan. 1937), S. 570–590. DOI: https://doi.org/10.1007/BF01594196 (zitiert auf S. 3, 13).

[ZG] H. P. Institut. *Zusammenfassung Graphentheorie*. URL: https://hpi.de/friedrich/docs/koetzing/units/graphs/files/Graphentheorie-Zusammenfassung.pdf (zitiert auf S. 18).

Alle URLs wurden zuletzt am 12. 12. 2020 geprüft.

Erklärung

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

Böblingen, 26. 12. 20, S.

Ort, Datum, Unterschrift