Norinori Rätsel mittels Top-Down-Konstruktion von Zero-Suppressed Decision Diagrams lösen

Bachelorarbeit

Julius Rädisch
409742

11. Oktober 2024

Gutachter: Prof. Dr. Benjamin Blankertz
Zweitgutachter: Dr. Stefan Fricke

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Neurotechnologie

Erklärung über die Verwendung generativer KI-Werkzeuge

Bei der Erstellung und Bearbeitung dieser Arbeit wurden generative KI-Werkzeuge verwendet. Spezifisch wurde ChatGPT von OpenAI in der Version GPT-4 verwendet.

Die Einsatzzwecke umfassen systematische Recherche, Unterstüzung bei der Dokumentenerstellung in Latex und Verbesserung der Texte. Es wurden keine Inhalte oder Formulierung direkt übernommen.

Abstract

ZDDs are a usefull and important tool in algorithmics. Their ability to represent problems with few solutions in a compact way makes them a powerfull instrument for various combinatorial problems. However, the field of their applications leaves much room for further research. Typical problems with few solutions are puzzles, as they usually only have one. The puzzle to be examined as an application for ZDDs in this thesis is Norinori, a puzzle published by the japanese Nikoli publishing house. This is a pen and paper game where you have to mark cells in a grid according to certain rules.

This paper presents an algorithm that finds all solutions to a given Norinori puzzle by constructing a ZDD. Based on time and memory measurements, this algorithm is then evaluated in terms of its practicability. Additionally, the influence of the order of variables is investigated by comparing different variants.

The results of this work show that although this approach is generally applicable, it rapidly loses efficiency when it comes to larger puzzles. This suggest that ZDDs are not optimal in every case and alternative approaches such as the integer programming chosen here for comparison can deliver better results. It is also shown that the order of variables has a significant influence in the performance and is not to be neglected when using ZDDs to solve a problem.

Kurzfassung

ZDDs sind ein nützliches und wichtiges Werkzeug in der Algorithmik. Ihre Fähigkeit, Probleme mit kleinen Lösungsmengen kompakt darzustellen, macht sie zu mächtigen Hilfsmitteln für diverse kombinatorische Probleme. Das Feld der Anwendungsmöglichkeiten von ZDDs lässt allerdings noch viel Platz für weitere Forschung. Klassische Probleme mit einem sehr kleinen Lösungsraum sind Rätsel, da diese in der Regel nur eine richtige Lösung besitzen. Das Rätsel, welches in dieser Arbeit als Anwendungsfall untersucht werden soll, ist das vom japanischen *Nikoli* Verlag veröffentlichte Norinori. Dabei handelt es sich um ein *Pen-and-Paper*-Spiel, bei welchem Kästchen auf einem Gitter nach bestimmten Regeln markiert werden.

In dieser Arbeit wird ein Algorithmus vorgestellt, welcher durch die Konstruktion eines ZDDs sämtliche Lösungen für Norinori-Rätsel findet. Anhand von Zeit- und Arbeitsspeichermessungen wird dieser Algorithmus anschließend hinsichtlich seiner Praktikabilität bewertet. Zusätzlich dazu wird durch den Vergleich verschiedener Varianten der Einfluss der Variablenreihenfolge untersucht.

Die Ergebnisse dieser Arbeit zeigen, dass der Ansatz zwar prinzipiell anwendbar ist, jedoch bei größer werdenden Rätseln schnell an Effizienz verliert. Dies lässt darauf schließen, dass ZDDs nicht in jedem Fall optimal sind und alternative Ansätze wie das hier zum Vergleich gewählte Integer Programming bessere Ergebnisse liefern können. Es wird außerdem gezeigt, dass die Variablenreihenfolge einen signifikanten Einfluss auf die Performance hat und bei der verwendung von ZDDs nicht zu vernachlässigen ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

BDD	Binäres Entscheidungsdiagram (Binary Decision Diagram)
ZDD	Nullunterdrücktes Binäres Entscheidungsdiagram (Zero-suppressed Binary Decision Diagram)

1 Einleitung

1.1 Motivation

ZDDs haben sich in den vergangenen Jahrzehnten als vielseitiges Werkzeug für diverse kombinatorische Probleme wie das *Traveling-Salesman*-Problem oder das *N-Queens*-Problem erwiesen. Das Ausmaß ihrer Möglichkeiten bleibt aber weiterhin ein relevantes Objekt der Forschung. Rätsel stellen ein beliebtes Testobjekt für die Erprobung von algorithmischer Leistungsfähigkeit dar, da sie oftmals eindeutige Lösungen haben und in Variationen verschiedener Komplexität getestet werden können. Da ZDDs besondere Effizienz bei Problemen mit sehr kleinem Lösungsraum aufweisen, ist es naheliegend, sie als Werkzeug der Wahl zu verwenden. Norinori ist ein solches Rätsel, welches zudem bislang nur sehr geringfügig wissenschaftlich untersucht worden ist. Die Lösung von Norinori-Rätseln mit Hilfe von ZDDs verspricht daher, sowohl weitere Einblicke in die Anwendbarkeit und Leistungsfähigkeit dieser zu gewähren, als auch den Forschungsstand zur Lösungsmethodik von Norinori-Rätseln auszubauen.

1.2 Ziele

Ziel dieser Arbeit ist es, die Anwendbarkeit von ZDDs zum Lösen von Norinori-Rätseln zu untersuchen. Es wird ein Algorithmus vorgestellt, der mittels *Top-down*-Konstruktion in der Lage ist, ein ZDD zu erstellen, das alle Lösungen einer gegebenen Norinori-Instanz abbildet. Anhand dieses Algorithmus wird die Effizienz dieser Methodik insbesondere im Hinblick auf die Skalierung mit Rätseln verschiedener Größe untersucht. Des Weiteren wird untersucht, wie groß der Einfluss der Variablenordnung auf die zur Lösung benötigte Zeit ist. Durch Vergleich dieser Ergebnisse mit einer Integer-Programming Methode soll festgestellt werden, wie Effektiv der untersuchte Ansatz ist. Ziel ist es, abschließend zu bestimmen, inwieweit ZDDs einen Mehrwert für die Lösung von Norinori-Rätseln bieten und Perspektiven für zukünftige Forschungsrichtungen im Bereich der algorithmischen Rätsellösungen und der Verwendung von ZDDs aufzuzeigen.

1.3 Struktur der Arbeit

Diese Arbeit ist in sieben Kapitel unterteilt. Nach einer einleitenden Übersicht über das Forschungsthema wird in Kapitel zwei der bisherige Forschungsstand zur Thematik besprochen. Dies beinhaltet

die Arbeit sowohl zum Thema ZDDs, als auch zu Norinori-Rätseln. Im dritten Kapitel werden die theoretischen Grundlagen erläutert. Dazu zählt eine genaue Beschreibung von Norinori-Rätseln, der zentralen Datenstruktur, ZDDs, und des verwendeten Algorithmus, dessen Spezifikation zum Lösen von Norinori-Rätseln in Kapitel vier vorgestellt wird. Aufbauend darauf, geht es in Kapitel fünf darum, die Leistungsfähigkeit des Algorithmus anhand von Benchmarktests auf Rätseln verschiedener Komplexität zu messen. Die Ergebnisse dieser Tests werden im sechsten Kapitel diskutiert, um die Anwendbarkeit des Algorithmus zum Lösen von Norinori-Rätseln zu bewerten. Abschließend werden im siebten Kapitel die aus der Diskussion hervorgegangenen Schlüsse verwendet, um die Forschungsfrage zu beantworten. Außerdem wird ein Ausblick auf zukünftige Forschung gegeben, die auf dieser Arbeit aufbauen könnte.

2 Bisherige Forschung

In diesem Abschnitt wird die relevante Forschung zur Thematik dieser Arbeit beleuchtet. Dazu gehören die bisherigen Arbeiten zu BDDs und ZDDs, welche die Grundlage für diese Arbeit geschaffen haben, andere Anwendungen von ZDDs, um Rätsel zu lösen und die bisherige wissenschaftliche Arbeit zu Norinori-Rätseln.

2.1 Arbeiten zum Thema ZDDs

BDDs wurden erstmals 1978 von Akers[1] vorgestellt. Sie bieten eine Möglichkeit, boolesche Funktionen auf eine Art und Weise zu handhaben, die mit bisherigen Mitteln nicht möglich war. Insbesondere die Anwendung zur Verifizierung von Schaltkreisen wird hierbei als Anwendungsfall betrachtet. Bryant erweiterte 1986 diese Idee[5], indem er Ordered Binary Decision Diagrams (OBDDs) einführte. Diese sind in der Lage, durch weitere Einschränkungen bezüglich der Reihenfolge der Variablen, die Effizienz von diversen Operationen auf booleschen Funktionen zu steigern. Dabei zeigte er, dass die Reihenfolge, in der die Variablen in einem Entscheidungsdiagramm auftauchen, signifikanten Einfluss auf die Größe dieser haben kann. Die optimale Reihenfolge zu finden ist laut ihm zwar selbst ein NP-hartes Problem, jedoch ist es möglich anhand von Heuristiken sehr gute Verbesserungen zu erzielen. Ein weiterer Vorteil dieser OBDDs ist ihre kanonische Form, welche dafür sorgt, dass es für jedes Diagramm exakt eine reduzierte Form gibt, wodurch äquivalente Funktionen durch identische OBDDs repräsentiert werden. OBDDs stellen die am weitesten verbreitete Version von BDDs dar, weshalb BDD oft synonym für OBDD verwendet wird. Seitdem wurden verschiedene Variationen von BDDs beziehungsweise OBDDs vorgestellt. Zumeist zielten die vorgestellten Abwandlungen darauf ab, die Effizienz der Auswertung von Funktionen zu steigern und somit in der Lage zu sein, komplexere Funktionen zu analysieren.

Eine solche Variation, IBDDs (Indexed BDDs), wurde 1992 von Jain et al[9] vorgestellt. Diese lockern die Bedingungen zur Reihenfolge von Variablen auf, indem sie den Graphen in Schichten unterteilen, die zwar in sich eine feste Variablenordnung haben, welche aber zwischen den Schichten variieren kann. Dadurch ist es möglich, die Auswertung von Funktionen aus Teilfunktionen mit sich unterscheidenden optimalen Variablenordnungen signifikant zu beschleunigen.

2002 führten Andersen et al. eine weitere Abwandlung der OBDDs, Boolean Expression Diagrams (BEDs), ein[2]. Diese ermöglichen eine Repräsentation von Funktionen in linearer Größe, verzichten aber dafür auf Kanonizität.

Die Variation, die in dieser Arbeit im Fokus steht (ZDDs), wurde 1993[11] von Minato vorgestellt.

Diese unterscheiden sich von bisherigen Formen von BDDs insofern, als dass sie boolesche Funktionen nicht mehr vollständig abbilden, sondern den Fokus auf die Menge an Variablenkombinationen legen, für die eine Funktion mit *true* ausgewertet wird. Er stellte dabei verschiedene kombinatorische Anwendungsbeispiele wie Schaltkreisverifikation oder das *N-Queens*-Problem vor. 1994 schloss Minato an diese Arbeit an[12], indem er Algorithmen für ZDDs vorstellte, mittels derer effiziente Berechnungen auf diesen Mengen (im englischen „unate cube set algebra“) durchgeführt werden können. Knuth griff im vierten Band seiner Buchreihe *„The Art of Computer Programming Vol. 4a: Combinatorial Algorithms“*[10] BDDs und ZDDs auf. Dabei stellte er einen Algorithmus vor, mit dem die Variablenordnung bei beiden Arten von Diagrammen optimiert werden kann (siehe Algorithmus J). Außerdem zeigte er diverse Anwendungsbeispiele für ZDDs. Dazu zählen die Verwendung als Wörterbuch für Wörter mit einer gewissen Länge, die Lösung des *Traveling-Salesman*-Problems, die Darstellung und Berechnung von Pfaden in Graphen und das Lösen von *exact-cover*-Problemen.

2012 stellten Yoshinaka et al.[14] Algorithmen vor, wie auf Grundlage des von Knuth beschriebenen *Sympath*-Algorithmus Instanzen von *Numberlink* und *Slitherlink* Rätseln sowohl gelöst, als auch generiert werden können. Hierbei verwendeten sie Top-Down-Konstruktion, um iterativ das ZDD aufzubauen. Ein Jahr später beschrieb Iwashita et al. eine allgemeine Form dieses Top-Down-Konstruktionsalgorithmus[8], der in dieser Arbeit ebenfalls verwendet wird. Dabei beschrieben sie sowohl den generischen Algorithmus, als auch die benötigten Spezifikationen, um ihn auf spezielle Probleme anzuwenden. Zudem zeigten sie Optimierungsmöglichkeiten durch vorausschauendes Ausschließen von Pfaden (engl. „*lookahead*“) und Aufteilen des Problems (engl. „*subsetting*“), welche sie verwendeten, um die in [14] vorgestellten Lösungsalgorithmen für *Numberlink* und *Slitherlink* zu optimieren. 2022 griffen Ishihata et al.[7] diesen Algorithmus auf, um mittels einer Spezifikation *Nageraru* Rätsel zu lösen. Dabei orientierten sie sich zum Teil ebenfalls an Knuths *Sympath*-Algorithmus.

In dieser Arbeit wird ebenfalls auf Grundlage des von Iwashita et al.[8] vorgestellten Algorithmus eine Spezifikation definiert, mit deren Hilfe Norinori-Rätsel gelöst werden sollen. Hierbei werden Ideen aus Knuths Lösung zu *exact-cover*-Problemen[10] aufgegriffen.

2.2 Arbeiten zu Norinori-Rätseln

Allgemein ist der Forschungsstand zu Norinori-Rätseln sehr überschaubar. Dumas et al.[6] haben gezeigt, dass es möglich ist, per *Zero Knowledge Proof* zu beweisen, dass man die Lösung eines Norinori-Rätsels kennt, ohne diese offenzulegen. Dies funktioniert, indem die Lösung für ein Rätsel mit Karten nachgestellt wird, welche anschließend umgedreht werden. Anhand dieser verdeckten Karten ist es mittels des präsentierten Protokolls möglich, nacheinander und unabhängig voneinander die verschiedenen Bedingungen einer Lösung zu überprüfen, ohne die exakte Position der Karten preiszugeben. Biro et al. haben zudem die NP-Vollständigkeit von Norinori bewiesen, indem gezeigt wurde, dass sich *PLANAR 3-IN-1-SAT* Probleme, welche erwiesenermaßen NP-vollständig sind, in Norinori-Rätsel transformieren lassen[3].

Zur Lösung dieser Rätsel selbst gibt es bisher keine wissenschaftlichen Arbeiten. Zwar lassen sich Solver[13] finden, jedoch ohne wissenschaftliche Aufarbeitung.

3 Grundlagen

In diesem Kapitel werden die Grundlagen dargelegt, die für das Verständnis und die Entwicklung des Algorithmus benötigt werden. Dazu gehören eine Erklärung und formale Definition von Norinori-Rätseln sowie *Zero-Suppressed Binary Decission Diagrams* (ZDDs). Des Weiteren wird ein generischer Konstruktionsalgorithmus vorgestellt, welcher im weiteren Verlauf für die Lösung dieser Rätsel spezifiziert wird.

3.1 Norinori

Norinori ist ein Logikrätsel, welches auf einem rechteckigen Gitter bzw. Feld gespielt wird. Das Spielfeld ist in kleinere Bereiche unterteilt, welche durch eine Umrandung gekennzeichnet sind und sich nicht überlappen. Ziel des Spiels ist es, Paare von vertikal oder horizontal benachbarten Zellen (im Folgenden als Dominos bezeichnet) so zu markieren, dass die folgenden Bedingungen erfüllt sind:

1. Jeder Bereich enthält genau zwei markierte Zellen.
2. Dominos dürfen sich nicht horizontal oder vertikal berühren. Diagonale Berührungen sind erlaubt.

Eine Instanz eines Norinori-Rätsels wird als ein Tripel $P = (R, C, A)$ definiert, wobei:

- R die Anzahl der Reihen des Spielfeldes ist,
- C die Anzahl der Spalten des Spielfeldes ist,
- $A = \{A_1, A_2, \ldots, A_k\}$ die Menge von Bereichen ist, wobei jedes A_i eine Menge von Zellen auf dem Spielfeld beschreibt. Jede Zelle wird durch ein Paar (r, c) identifiziert, wobei r die Reihe und c die Spalte bezeichnet. Für alle i, j, mit $i \neq j$, gilt $A_i \cap A_j = \emptyset$.

Der Zustand eines Rätsels S ist die Menge aller Zellen (r, c) die bereits markiert wurden, wobei r und c für die jeweilige Reihe und Spalte der Zelle stehen.

Die Nachbarschaft einer Zelle $N_{(r,c)} = \{(r+1, c), (r-1, c), (r, c+1), (r, c-1)\}$ ist die Menge aller orthogonal benachbarten Zellen. Für Randzellen ist diese Menge entsprechend kleiner.

Ein Rätsel gilt als gelöst, wenn die beiden oben genannten Regeln erfüllt sind:

$$(\forall A_i \in A : |A_i \cap S| = 2) \wedge (\forall (r, c) \in S : \ |N_{(r,c)}| = 1)$$

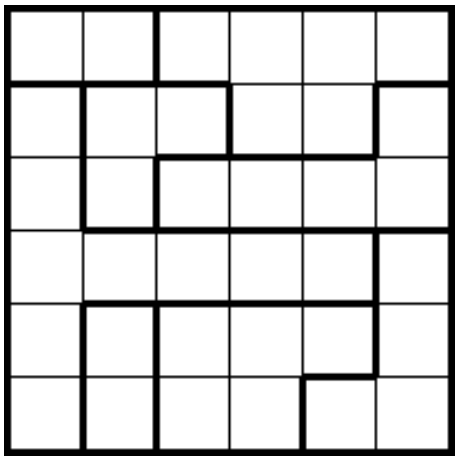


Abbildung 3.1: Ungelöstes Norinori-Rätsel

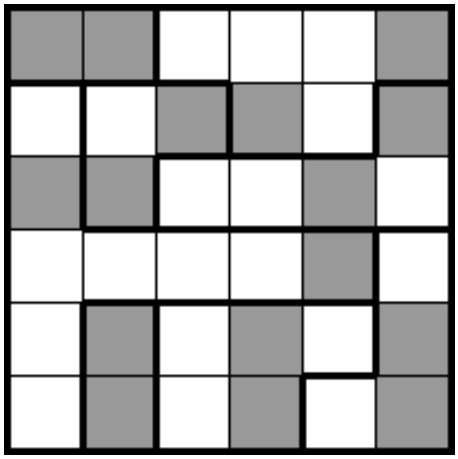


Abbildung 3.2: Gelöstes Norinori-Rätsel.

3.2 ZDDs

Ein Binary Decision Diagram (BDD) ist eine graphische Repräsentation einer booleschen Funktion, die in der Lage ist, diese kompakt darzustellen. Sie ermöglicht es, boolesche Operationen effizient auf verschiedenen Funktionen durchzuführen. Zero-suppressed Binary Decision Diagrams (ZDDs) sind eine Variation von BDDs, die besonders effizient darin sind, Mengen von Teilmengen aus einem gemeinsamen Universum, sogenannte „Cube Sets“ darzustellen. Genauso wie bei BDDs handelt es sich bei ZDDs um einen gerichteten azyklischen Graphen mit den folgenden Eigenschaften:

- Jeder Knoten hat entweder einen Ausgangsgrad von 0 (im Folgenden als Blatt-Knoten bezeichnet), oder hat einen Ausgangsgrad von genau zwei (im Folgenden als innerer Knoten bezeichnet).
- Es existieren genau zwei Blatt-Knoten mit den Werten *true* (1) und *false* (0).
- Jeder innere Knoten repräsentiert eine Variable des zugrundeliegenden booleschen Problems und ist mit einem Label versehen, welches diese Variable identifiziert. Die Reihenfolge dieser Labels ist auf allen Pfaden gleich.
- Jeder innere Knoten hat genau ein 1-Kind und ein 0-Kind, welche die Annahme der Werte *true* und *false* der zugrundeliegenden Variable darstellen. In den folgenden Grafiken ist die 0-Kind-Beziehung mit einem gestrichelten Pfeil, und die 1-Kind-Beziehung mit einem durchgezogenen Pfeil dargestellt.
- Es existiert genau ein Ursprungsknoten, auch Wurzelknoten genannt, mit Eingangsgrad 0.

Durch Reduktion lassen sich BDDs und ZDDs in eine kanonische Form bringen, welche überflüssige Knoten und Pfade eliminiert. Diese hat drei primäre Vorteile:

- **Eindeutigkeit:** Für jede boolesche Funktion und jedes Cube Set existiert exakt eine reduzierte Repräsentation als BDD bzw. ZDD.
- **Vergleichbarkeit:** Durch diese eindeutige Darstellung ist es möglich boolesche Funktionen und Cube Sets effizient auf Äquivalenz zu überprüfen. Identische Diagramme weisen auf äquivalente Funktionen bzw. Mengen hin.
- **Effizienz:** Die reduzierte Form stellt die kompakteste mögliche Darstellung der zugrundeliegenden Funktion oder Menge dar. Diese Minimierung resultiert in einer optimierten Basis für die Durchführung von logischen Operationen.

Die erste Regel der Reduktion ist die Eliminierung von redundanten Knoten. Hierbei werden Knoten mit identischem Label, 1-Kind und 0-Kind zu einem Knoten „verschmolzen“. Abbildung 3.3 stellt diese Eliminierung dar.

In der zweiten Reduktionsregel unterscheiden sich BDDs von ZDDs. Diese ist auch maßgeblich für den funktionalen Unterschied zwischen den beiden Datenstrukturen.

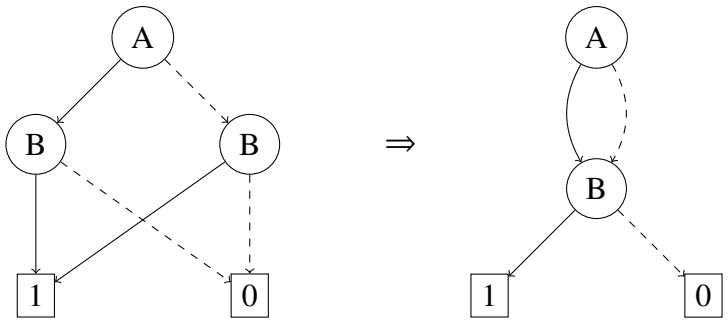


Abbildung 3.3: Zusammenfügung redundanter Knoten

Um BDDs vollständig zu reduzieren, werden Knoten, deren 1- und 0-Kind identisch sind, eliminiert und an ihrer Stelle ihr Kind im Graphen eingehängt. Abbildung 3.4 stellt dies graphisch dar. Durch diese Eliminierung von irrelevanten Knoten werden BDDs auf diese Entscheidungen reduziert, die relevant zur Auswertung der zugrundeliegenden booleschen Funktion sind und ermöglichen somit deren effiziente Auswertung.

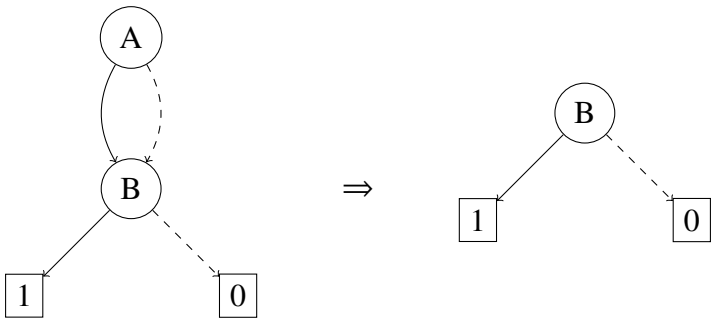


Abbildung 3.4: Eliminierung irrelevanter Knoten bei BDDs

Für die vollständige Reduktion von ZDDs werden sämtliche Knoten eliminiert, deren 1-Kind das *false*-Blatt ist, dies ist die namensgebende Null-Unterdrückung. Dabei wird ihre eingehende Kante auf ihr 0-Kind umgeleitet. Hierdurch entsteht ein Diagramm, welches so kompakt wie möglich die Menge aller Variablenkombinationen repräsentiert, für die das zugrundeliegende Problem mit *true* ausgewertet wird. Im Fall des in Abbildung 3.5 dargestellten reduzierten ZDDs lautet diese Menge $\{11, 00\}$. Anzumerken ist hierbei jedoch, dass Variablen, die unter keiner Bedingung 1 sein dürfen, im ZDD nicht zu erkennen sind. Deswegen kann das gezeigte ZDD genauso die Mengen $\{110, 000\}$, $\{1100, 0000\}$ etc. darstellen.

Da für die Auswertung eines ZDDs nur die Pfade relevant sind, die im 1-Blatt münden, ist es ebenfalls üblich, das 0-Blatt bei der Darstellung von ZDDs auszusparen. Hierdurch wird diese noch kompakter, ohne an Aussagekraft zu verlieren, wie in Abbildung 3.6 gezeigt wird.

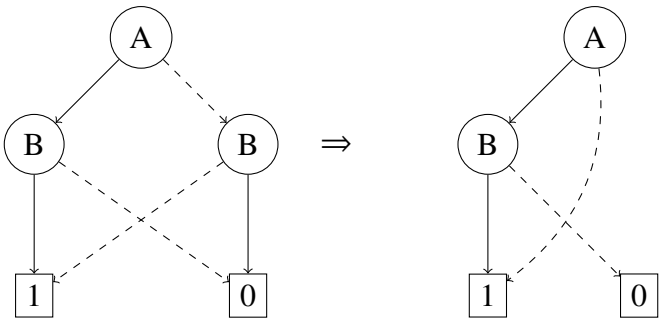


Abbildung 3.5: Nullunterdrückung bei ZDDs

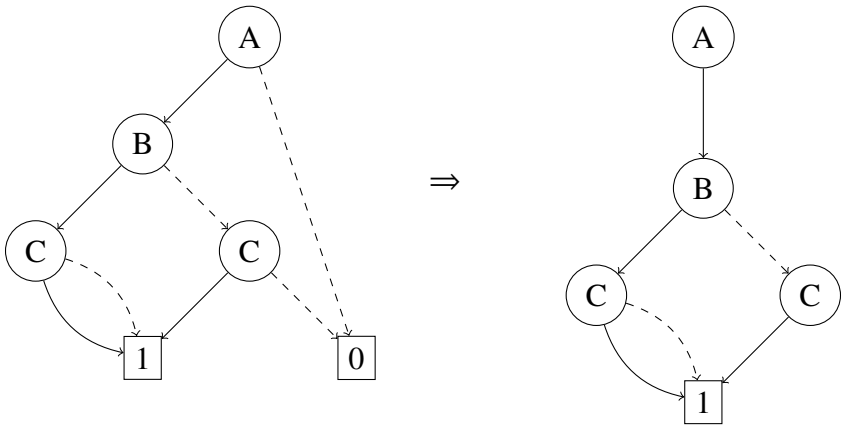


Abbildung 3.6: ZDDs mit und ohne 0-Blatt

3.3 Top-down ZDD-Konstruktion

Iwashita et al. hat in seinem 2013 veröffentlichten Paper „*Efficient Top-Down ZDD Construction Techniques Using Recursive Specifications*“[8] einen generischen Algorithmus für die Konstruktion von ZDDs von oben nach unten vorgestellt. Dieser wird im Folgenden erläutert, um darauf aufbauend für Norinori-Rätsel spezifiziert zu werden.

Für diesen Algorithmus wird die Konfiguration eines Knotens als Tupel $\langle i, s \rangle$[1] definiert. Dabei ist i der Index des Knotens und s der aktuelle Zustand der Suche. Dieser Zustand wird verwendet, um zu bestimmen, ob eine Lösung gefunden wurde, keine Lösung mehr möglich ist, oder die Suche fortgesetzt werden muss. Außerdem werden für diesen Algorithmus zwei Hilfsfunktionen benötigt:

- **getRoot()**: gibt die Konfiguration des Wurzelknotens $\langle i_0, s_0 \rangle$ zurück.
- **getChild**($\langle i, s \rangle, b$): gibt Anhand einer gegebenen Konfiguration $\langle i, s \rangle$ und einer Entscheidung $b \in \{0, 1\}$ die Konfiguration des entsprechenden b-Kindes zurück.

Zu Beginn des Algorithmus werden drei Knoten initialisiert: der Wurzelknoten, das 1-Blatt und das 0-Blatt. Anschließend wird über sämtliche Indizes innerhalb des Entscheidungsraums iteriert. Für

[1] Definition in Anlehnung an Iwashita et al.[8]

jeden Knoten, dessen Konfiguration den entsprechenden Index enthält, werden mittels der *getRoot*-Funktion die Konfigurationen der Kinder ermittelt. Falls die ermittelte Konfiguration direkt einem der Blatt-Knoten entspricht, wird dieses als das entsprechende Kind des aktuellen Knotens festgelegt. Sollte dies nicht der Fall sein, wird im Graphen nach einem passenden Knoten gesucht, der diese Konfiguration bereits repräsentiert. Existiert ein solcher Knoten noch nicht, wird er neu erstellt und an der entsprechenden Stelle im Graphen eingefügt.

Algorithm 1 Top-Down ZDD Konstruktion

```
1:  procedure CONSTRUCT(S)
2:      ⟨i_0, s_0⟩ ← S.getRoot();                          ▷ Wurzelknoten erhalten
3:      erstelle einen neuen Knoten r mit Konfiguration ⟨i_0, s_0⟩;
4:      for i = i_0 to n do
5:          for all Knoten p mit Konfiguration ⟨i, s⟩ für ein beliebiges s do
6:              for all b ∈ {0, 1} do
7:                  ⟨i', s'⟩ ← S.getChild(⟨i, s⟩, b);      ▷ Kindknoten bestimmen
8:                  if ⟨i', s'⟩ ist ein Blatt-Knoten then
9:                      weise ihn als b-Kind von p zu;
10:                 else
11:                     finde oder erstelle Knoten p' mit Konfiguration ⟨i', s'⟩;
12:                     weise p' als b-Kind von p zu;
13:                 end if
14:             end for
15:         end for
16:     end for
17:     return REDUCE(r);                                  ▷ Wende Reduktionsregeln an
18: end procedure
```

4 Lösungsalgorithmus

Zur Spezifikation des beschriebenen Algorithmus werden im Folgenden fünf Bestandteile definiert und, sofern notwendig, deren Aufbau erläutert und hergeleitet: der Zustand des Rätsels, die zu untersuchenden Variablen, die Repräsentation des Rätsels, die *getRoot*-Funktion, sowie die *getChild*-Funktion.

Zur Implementierung wurde die von Hiroaki Iwashita zur Verfügung gestellte *TdZdd*-Bibliothek[1] verwendet. Diese bietet eine C++ Implementierung des oben beschriebenen Algorithmus. Im Folgenden präsentierter Code wird sich auf Grund dieser Implementierung etwas von der obigen Definition unterscheiden. So wird von der *getRoot*- und der *getChild*-Funktion lediglich der Index zurückgegeben. Die Veränderung des Zustands des Rätsels passiert *in place* anhand übergebener Referenzen.

4.1 Zustand

Der Zustand des Rätsels besteht aus zwei Komponenten. Die erste Komponente ist die Menge aller platzierten Zellen. Dies entspricht der in 3.1 gegebenen Definition des Zustandes und wäre rein Inhaltlich ausreichend, um diesen zu beschreiben. Allerdings ergeben sich im Hinblick auf die Performance einige Vorteile daraus, zu speichern, welche Zellen nicht mehr platziert werden dürfen und/oder können. Daher besteht die zweite Komponente des Zustandes aus der Menge dieser Zellen.

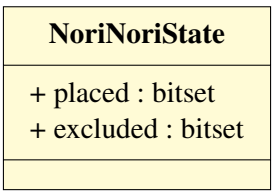


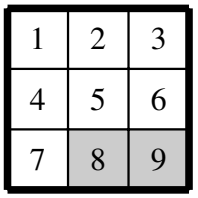
Abbildung 4.1: NoriNoriState-Klassendiagramm

4.2 Variablen

Die Menge an Variablen definiert den Suchraum. Im Fall des Norinori-Rätsels gibt es eine Variable für jedes mögliche Paar an orthogonalen Zellen, im Weiteren als Domino bezeichnet. Diese repräsentiert die Entscheidung, diese in die Lösung aufzunehmen, oder nicht. Anders als in der obigen

[1] https://github.com/kunisura/TdZdd

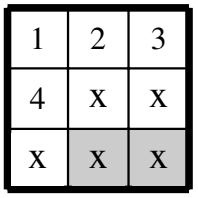
Definition, wird die Position einer Zelle hierbei nicht Anhand ihrer Zeile und Spalte dargestellt, sondern Anhand des Indexes, der aus der vollständigen Nummerierung der Zellen im Spielfeld resultiert. Dies ermöglicht eine Repräsentation jeder Variable durch ein *Bitset*, welches eine Stelle für jede Zelle im Spielfeld hat (*R*C*). Die Bits an den Positionen der von der jeweiligen Variable betroffenen Zellen werden hierbei auf *1* gesetzt.



$\Rightarrow \quad \{0, 0, 0, 0, 0, 0, 0, 1, 1\}$

Abbildung 4.2: Binärdarstellung einer Variable

Zusätzlich wird analog zu jeder Variable eine Hilfsvariable erstellt, die speichert, welche Zellen durch das Annehmen einer Variable in Zukunft nicht mehr platziert werden können.



$\Rightarrow \quad \{0, 0, 0, 0, 1, 1, 1, 1, 1\}$

Abbildung 4.3: Durch eine Variable ausgeschlossene Zellen

4.2.1 Variablenordnung

Da die Reihenfolge, in der die Variablen untersucht werden, maßgeblich für die Größe des resultierenden ZDDs ist, hat sie auch starken Einfluss auf die Zeit, in der dieses konstruiert wird. Als Vorbereitung für den eigentlichen Algorithmus werden die Variablen in zwei Schritten sortiert:

1. Für den ersten Schritt, werden die Variablen nach ihrer Position im Spielfeld sortiert. Dabei werden für jede Variable die Indizes des zugrundeliegenden Zellenpaares betrachtet. Der größere der beiden Indizes wird als Vergleichswert verwendet. Die Variablen werden nun nach diesem Wert in absteigender Reihenfolge sortiert (siehe Abbildung 4.4). Durch diese Sortierung kommt es möglichst frühzeitig dazu, dass eine einzelne Zelle in der weiteren Suche nicht mehr berücksichtigt wird, wodurch sie in die Menge an ausgeschlossenen Zellen aufgenommen werden kann. Dieser Ausschluss von Zellen wird in Abschnitt 4.5.2 weiter erläutert.

2. Im zweiten Schritt werden Variablen neu einsortiert, die entweder an einen Bereich der Größe zwei angrenzen, oder nur eine seiner Zellen beinhalten. Dabei handelt es sich um Variablen, die das Rätsel alleine bereits unlösbar machen würden. Diese werden zuerst untersucht, da durch einen frühzeitigen Ausschluss dieser Variablen der Suchraum möglichst klein gehalten wird. Besonders relevant sind hierbei die Variablen, die nur eine Zelle eines Bereichs beinhalten. Ein Beispiel für eine solche Variable wird in Abbildung 4.5 gezeigt. Durch das Annehmen dieser

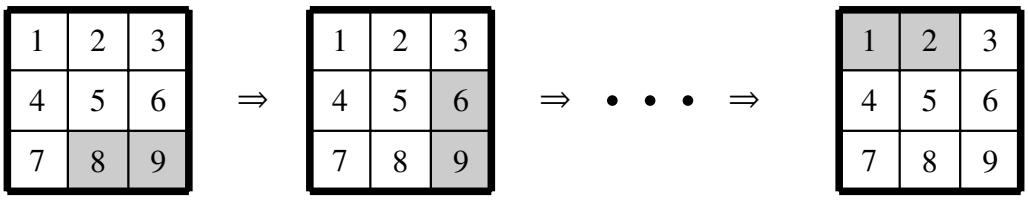


Abbildung 4.4: Vorläufige Reihenfolge der Variablen

Variable würde Zelle 8 nicht platziert werden dürfen und der markierte Bereich somit nicht ausgefüllt.

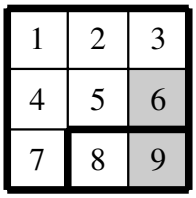


Abbildung 4.5: Variable, die nie verwendet werden darf

Als nächstes werden alle Variablen untersucht, die einem dieser Bereiche genau entsprechen. Durch den vorherigen Ausschluss aller Variablen, die diesen Bereich nur teilweise ausfüllen, ist es möglich festzustellen, dass die Zellen dieses Bereichs ohne diese Variable nicht mehr belegt werden können und somit das Rätsel durch eine Ablehnung dieser unlösbar wird, wodurch der Suchraum weiter beschnitten werden kann.

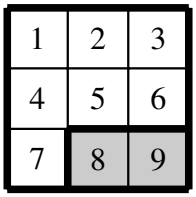
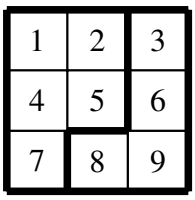


Abbildung 4.6: Variable, die verwendet werden muss

4.3 Rätsel-Repräsentation

Zur Repräsentation einer Norinori-Instanz wird die Klasse *NoriNoriProblem* definiert. Diese beinhaltet, wie in Abschnitt 3.1 beschrieben, die Anzahl an Zeilen und Spalten, sowie die Bereiche, welche als Menge an Bits dargestellt werden (siehe Abbildung 4.7). Darüber hinaus werden in dieser Klasse sämtliche Variablen in ihrer korrekten Reihenfolge als *dominos* gespeichert. Analog dazu werden sämtliche Zellen, die durch das Platzieren eines Dominos ausgeschlossen werden, als *forbidden_cells* gespeichert. Außerdem wird für jede Zelle gespeichert, welche die letzte Variable ist, die diese platzieren kann. Diese Liste *last_occurences* wird dafür verwendet, die Unlösbarkeit eines Zustandes besser bestimmen zu können. Mehr dazu in Abschnitt 4.5.2.



$\Rightarrow \quad \{\{1,1,0,1,1,0,1,0,0\},\{0,0,1,0,0,1,0,1,1\}\}$

Abbildung 4.7: Darstellung von Bereichen

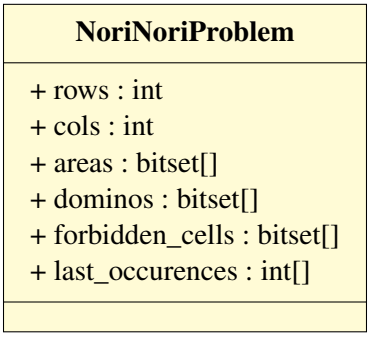


Abbildung 4.8: NoriNoriProblem-Klassendiagramm

4.4 getRoot-Funktion

Die *getRoot*-Funktion initialisiert als Zustand sowohl die platzierten, als auch die ausgeschlossenen Zellen mit einer leeren Menge. Der Index des Wurzelknotens ist die Anzahl an Variablen $R * (C-1) + (R-1) * C$.

---

function GETROOT(s)
  $s.placed \leftarrow$ new bitset(P.getCols() $\times$ P.getRows())
  $s.excluded \leftarrow$ new bitset(P.getCols() $\times$ P.getRows())
  return $R \times (C-1) + (R-1) \times C$
end function

4.5 getChild-Funktion

Die *getChild*-Funktion beinhaltet die zentrale Logik des Algorithmus. Sie erhält den aktuellen Zustand des Rätsels *s*, sowie die aktuelle Suchebene *level* und eine Entscheidung *take* $\in \{0,1\}$. Im Folgenden wird zuerst die vollständige Funktion präsentiert, um im Anschluss die einzelnen Bestandteile zu erläutern.

Allgemein unterscheidet die Funktion zwischen zwei Fällen, abhängig vom Wert von *take*, also ob die untersuchte Variable in die Lösung aufgenommen wird oder nicht. Ein Wiedergabewert von 0 entspricht einer Zuweisung des 0-Blattes, ein Wiedergabewert von -1 entspricht einer Zuweisung des 1-Blattes, also einer gefundenen Lösung. Referenzen auf *P* beziehen sich auf die Instanz des Norinori-Problems.

Input: ▷ *s*: der aktuelle Zustand
▷ *level*: die aktuelle Suchebene
▷ *take*: die Entscheidung, die Variable anzunehmen
Referenzen: ▷ *P*: die Norinori-Instanz
Output: ▷ *-1*: Lösung gefunden
▷ *0*: unlösbarer Zustand erreicht
▷ *level - 1*: aktualisierte Suchebene

```
1:  function GETCHILD(s, level, take)
2:      if take then                                          ▷ Variable wird angenommen
3:          if (P.dominos[level] ∧ s.excluded).any() then
4:              return 0                                      ▷ Zug ist ungültig
5:          end if
6:          s.excluded ← s.excluded ∨ P.forbidden_cells[level]   ▷ aktualisiere Zustand
7:          s.placed ← s.placed ∨ P.dominos[level]
8:          for all area ∈ P.areas do                         ▷ prüfe jeden Bereich auf Lösbarkeit
9:              if (area ∧ s.placed).count() > 2 or ISIMPOSSIBLE(area, s) then
10:                 return 0                                  ▷ unlösbarer Zustand erreicht
11:             end if
12:         end for
13:         if P.areas.size() × 2 = s.placed.count() then
14:             return −1                                     ▷ Lösung wurde gefunden
15:         end if
16:     else                                                  ▷ Variable wird abgelehnt
17:         for all cell ∈ P.dominos[level] do
18:             if P.last_occurences[cell] = level then
19:                 s.excluded.set(cell)          ▷ Zelle kann in Zukunft nicht mehr gesetzt werden
20:             end if
21:         end for
22:     end if
23:     return level − 1                                      ▷ setze die Suche fort
24: end function
```

4.5.1 Fall 1: Annehmen einer Variable

Falls $take = 1$, wird der Fall untersucht, dass die entsprechende Variable in die Lösung aufgenommen wird. Dafür wird zunächst überprüft, ob es sich um einen legalen Zug handelt, indem die Zellen der Variable mit den bereits ausgeschlossenen Zellen abgeglichen werden und im Falle einer Überschneidung, 0 zurückgegeben wird.

```
3: if (P.dominos[level] ∧ s.excluded).any() then
4:     return 0
5: end if
```

Handelt es sich um einen legalen Zug, wird der Zustand des Rätsels aktualisiert. Daraufhin wird für jeden Bereich überprüft, ob dieser noch zu lösen ist. Das heißt, dass die maximale Anzahl an verwendeten Zellen von 2 noch nicht überschritten wurde, aber noch zu erreichen ist.

```
6: s.excluded ← s.excluded ∨ P.forbidden_cells[level]
7: s.placed ← s.placed ∨ P.dominos[level]
8: for all area ∈ P.areas do
9:     if (area ∧ s.placed).count() > 2 or ISIMPOSSIBLE(area, s) then
10:        return 0
11:    end if
12: end for
```

Sollte es auch hier zu keinem Abbruch der Suche kommen, wird zuletzt überprüft, ob der aktuelle Zustand eine valide Lösung für das Problem ist. Dies wird lediglich anhand der Anzahl der platzierten Zellen getan, da sämtliche anderen Bedingungen implizit bereits vorher überprüft wurden.

```
13: if P.areas.size() × 2 = s.placed.count() then
14:     return −1
15: end if
```

Ist auch das nicht der Fall, wird *level* aktualisiert und die Suche fortgesetzt, wobei jedoch eine Aktualisierung von *level* auf 0 bedeutet, dass der gesamte Suchraum durchsucht wurde, ohne eine Lösung zu finden, was ebenfalls zum Abbruch der Suche führt.

```
23: return level − 1
```

4.5.2 Fall 2: Ablehnen einer Variable

Falls $take = 0$, wird die entsprechende Variable nicht in die Lösung aufgenommen. In diesem Fall wird überprüft, ob eine der von der Variable betroffenen Zellen zum letzten Mal hätte platziert werden können. Sollte dies der Fall sein, kann die entsprechende Zelle in die Menge der ausgeschlossenen Zellen aufgenommen werden. Dadurch kann später besser festgestellt werden, ob ein Bereich noch lösbar ist oder nicht. Anschließend wird, wie in Fall 1, *level* aktualisiert, und die Suche fortgesetzt, sofern nicht die finale Suchtiefe erreicht wurde.

```
17: for all cell ∈ P.dominos[level] do
18:     if P.last_occurences[cell] = level then
19:         s.excluded.set(cell)
20:     end if
21: end for
```

5 Ergebnisse

Im Folgenden werden die Ergebnisse der Leistungsmessung präsentiert. Hierbei wurde der Algorithmus zur Lösung verschiedener Rätsel-Instanzen verwendet. Der erste dabei gemessene Parameter ist die benötigte Zeit zum Lösen einer Instanz. Diese ist besonders aussagekräftig für den Vergleich mit alternativen Lösungsansätzen. Zusätzlich dazu wurde der Arbeitsspeicherverbrauch erfasst. Frühere Arbeiten[7] haben gezeigt, dass mit zunehmender Problemgröße der Arbeitsspeicherverbrauch signifikant ansteigt, was sich als limitierender Faktor für die Effizienz dieses Algorithmus erweisen könnte. Der verwendete Code ist in dem folgenden Repository zu finden: https://git.tu-berlin.de/jraedisch1406/norinorizdd

Zunächst wird im folgenden Abschnitt auf die Details der Messmethodik eingegangen, bevor die konkreten Ergebnisse der Messungen beider Parameter vorgestellt und aufbereitet werden. Anschließend wird der Einfluss der Variablenordnung anhand verschiedener Varianten präsentiert, um deren Signifikanz zu bewerten. Zuletzt werden die Laufzeit-Messwerte eines alternativen Integer-Programming Solvers vorgestellt, um anhand derer im weiteren Verlauf die Performance des vorgestellten Algorithmus zu bewerten.

5.1 Technische Details

Der Algorithmus wurde unter Zuhilfenahme der *TdZdd*-Bibliothek[1] in C++ implementiert und mittels g++ in der Version 13.1.0 kompiliert. Zur Ausführung der Experimente wurde ein 64-bit Windows 11, Version 10.0.22621 mit acht Intel Core i7 2.5GHz CPU und 32GB RAM verwendet. Allerdings wurden alle Experimente auf lediglich einem Kern ausgeführt. Der Integer-Programming Solver wurde in Python mittels der PuLP-Bibliothek implementiert. Zur Messung wurde die selbe Hardware wie beschrieben und Python 3.11.0 verwendet.

5.2 Messmethodik

Zur Messung der Parameter wurden Rätsel-Instanzen von den Quellen
https://www.janko.at/Raetsel/Norinori/ und https://de.puzzle-norinori.com/ verwendet. Sofern möglich, wurde pro Spielfeldgröße mit jeweils 100 Instanzen getestet. Allerdings war bei größeren Rät-

[1] https://github.com/kunisura/TdZdd

selformaten die Verfügbarkeit von Instanzen begrenzt, weshalb ab einer Spielfeldgröße von 12x12 die Testgruppengröße deutlich kleiner ausfällt.

Zur Ermittlung der Laufzeit wurde jede Instanz zehn Mal gelöst und der Durchschnitt der gemessenen Zeiten errechnet. Bei einer Zeit von mehr als 60 Minuten wurden die Messungen abgebrochen und als Timeout gewertet.

Zur Messung des Arbeitsspeicherverbrauchs wurde die *Valgrind*-Software unter Zuhilfenahme des *Massif*-Tools verwendet. Aus dem resultierenden Bericht wurde der höchste während der Ausführung gemessene Wert als Richtwert genommen. Da die Messung des Arbeitsspeicherverbrauchs auf diese Weise jedoch die Laufzeit signifikant erhöht, wurden Laufzeit und Arbeitsspeicherverbrauch separat voneinander gemessen. Außerdem war es dadurch aus Zeitgründen nicht möglich, für Instanzen größer als 13x13, den Arbeitsspeicherverbrauch zu messen. Sofern möglich wurden von jeder Rätsel-Größe 20 Instanzen gemessen.

5.3 Messwerte

5.3.1 Laufzeit

Die Tabelle 5.1 zeigt den Durchschnitt und den Median der benötigten Zeit in Sekunden, sowie deren Varianz und die Anzahl an nicht terminierten Instanzen. Nicht terminierte Instanzen wurden bei der Berechnung der Kennzahlen ignoriert. Bis zu einer Größe von 13x13 konnte jede Instanz gelöst werden, wobei die benötigte Zeit bei zunehmender Rätsel-Größe massiv zunimmt. Ab einer Größe von 14x14 konnten nicht alle Instanzen gelöst werden, was die berechneten Kennzahlen verzerrt.

Darüber hinaus führt die Kombination aus stark steigender Varianz und sinkender Testgruppengröße bei größer werdenden Rätseln zu weniger aussagekräftigen Zahlen. Daher werden im Folgenden hauptsächlich die Spielfeldgrößen 6x6 bis 13x13 betrachtet.

Spielfeldgröße	Instanzen	Durchschnitt	Median	Varianz	nicht terminiert
6x6	100	0,034	0,019	0,002	0
8x8	100	0,635	0,177	2,27	0
10x10	100	3,062	0,709	52,645	0
12x12	10	250,821	11,260	257.666,7	0
13x13	10	334,363	123,287	170.353,1	0
14x14	3	94,7	94,7	11.370,5	1

Tabelle 5.1: Benchmark-Kennzahlen nach Spielfeldgröße

Abbildung 5.1 stellt den Durchschnitt der Laufzeiten der jeweiligen Spielfeldgrößen grafisch dar. Zur Visualisierung der Verteilung der jeweiligen Laufzeiten zeigen die Abbildungen 5.2 bis 5.6 diese

anhand von Histogrammen. Die x-Achsen wurden logarithmisch eingeteilt, um die Verteilung trotz enormer Varianz aussagekräftig darstellen zu können.

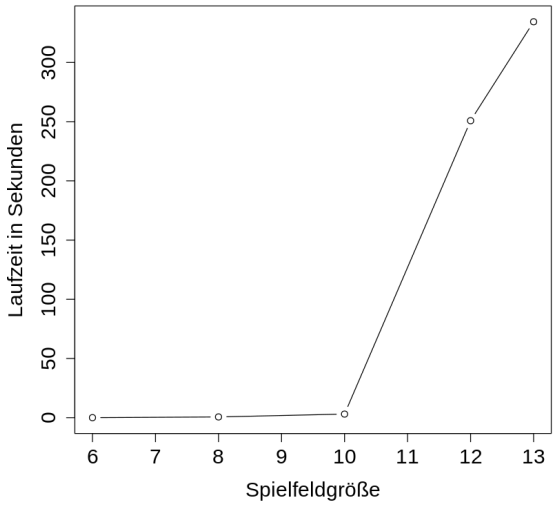


Abbildung 5.1: Durchschnittslaufzeiten nach Rätselgröße bis 13x13

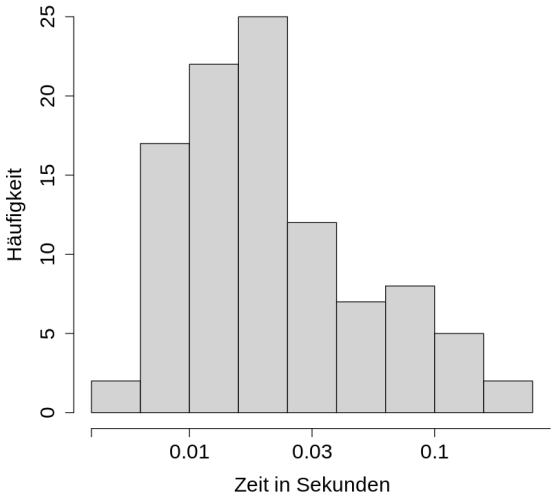


Abbildung 5.2: Häufigkeiten der Laufzeiten für 6x6 Instanzen

5.3.2 Arbeitsspeicherverbrauch

Tabelle 5.2 zeigt den Durchschnitt, den Median sowie die Varianz des verwendeten Arbeitsspeichers in KB. Im Median ist analog zur Laufzeit ein eindeutiger Anstieg mit wachsender Spielfeldgröße zu erkennen. Der Durchschnitt der Werte zeigt zwar die Tendenz, bei wachsender Spielfeldgröße ebenfalls größer zu werden, wächst allerdings nicht monoton. Eine Besprechung dieses Verhaltens ist in Abschnitt 6.1 zu finden.

Spielfeldgröße	Instanzen	Durchschnitt	Median	Varianz
6x6	20	264,325	260,2	438,07
8x8	20	484,515	315,05	250.322,8
10x10	20	444,24	330,65	141.096,39
12x12	10	25.940,37	1.458,65	$2,82 \times 10^{11}$
13x13	10	12.903,511	8.891	$2,38 \times 10^{8}$

Tabelle 5.2: Arbeitsspeicherverbrauch nach Spielfeldgröße in KB

Abbildung 5.7 stellt den Durchschnitt des Arbeitsspeicherverbrauchs der jeweiligen Spielfeldgrößen grafisch dar. Zur Visualisierung der Verteilung der jeweiligen Laufzeiten wurden diese wieder in in Form von Histogrammen aufbereitet. Die Abbildungen 5.8 bis 5.12 zeigen diese. Im Sinne der Anschaulichkeit wurden die x-Achsen wieder logarithmiert, mit einer Ausnahme von Abbildung 5.8, da eine lineare Skalierung dort die Verteilung aussagekräftiger wiedergibt.

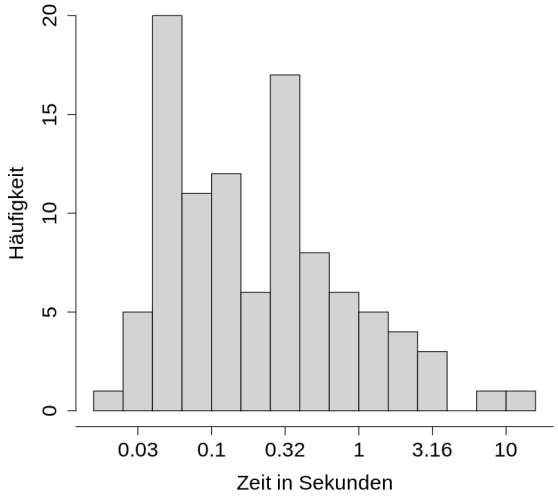


Abbildung 5.3: Häufigkeiten der Laufzeiten für 8x8 Instanzen

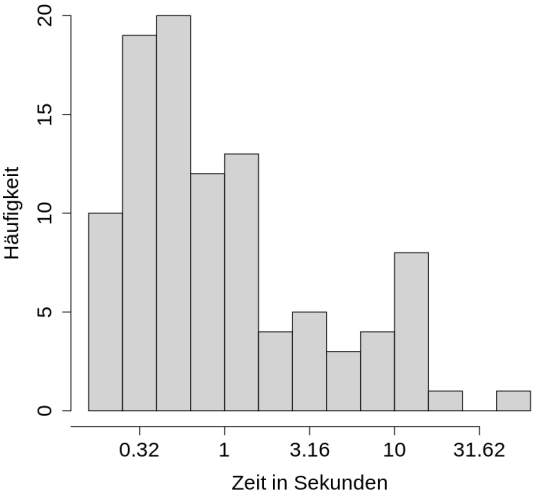


Abbildung 5.4: Häufigkeiten der Laufzeiten für 10x10 Instanzen

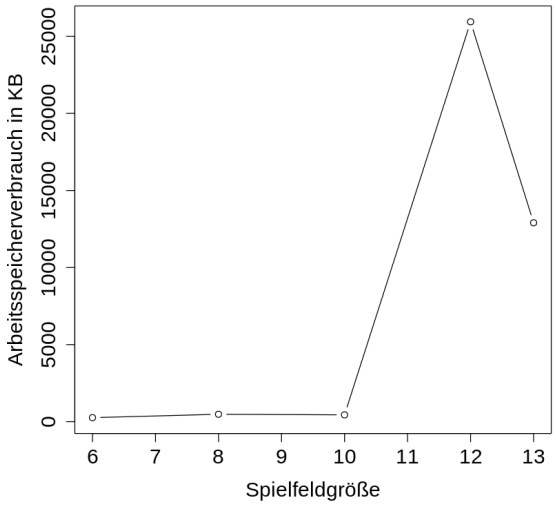


Abbildung 5.7: Durchschnittslaufzeiten nach Rätselgröße bis 13x13

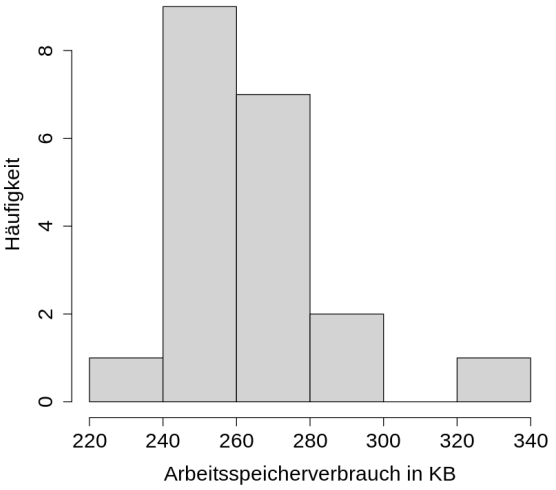


Abbildung 5.8: Häufigkeiten des Arbeitsspeicherverbrauchs für 6x6 Instanzen

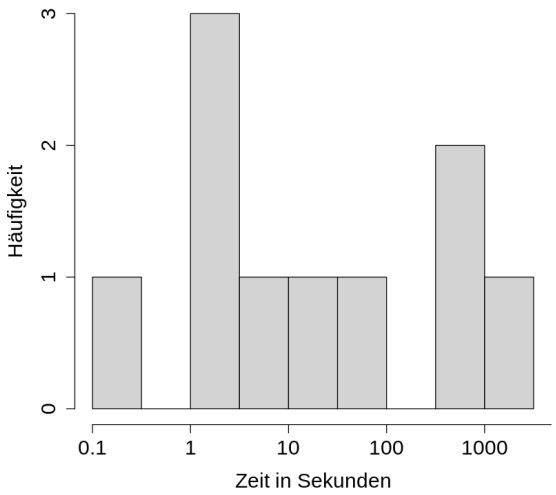


Abbildung 5.5: Häufigkeiten der Laufzeiten für 12x12 Instanzen

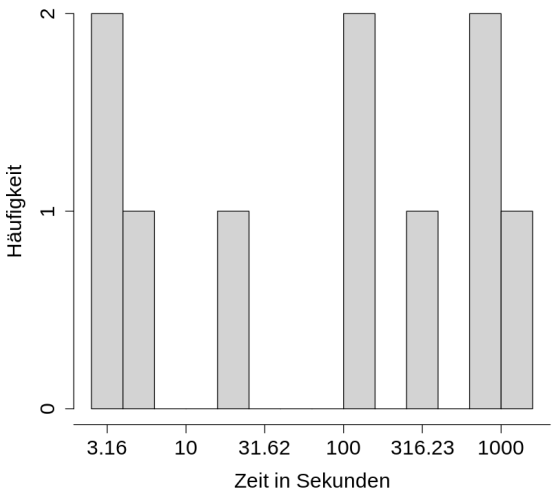


Abbildung 5.6: Häufigkeiten der Laufzeiten für 13x13 Instanzen

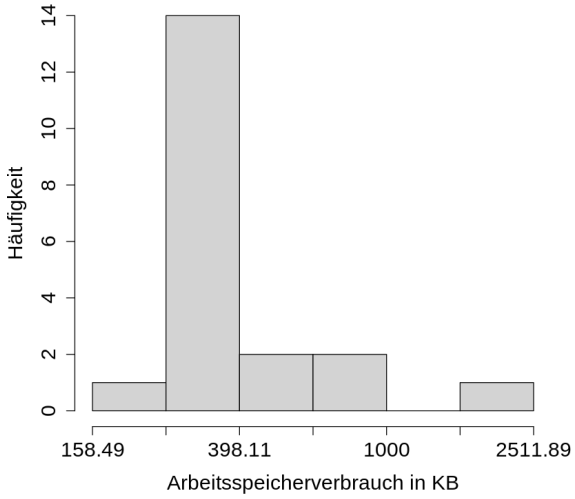


Abbildung 5.9: Häufigkeiten des Arbeitsspeicherverbrauchs für 8x8 Instanzen

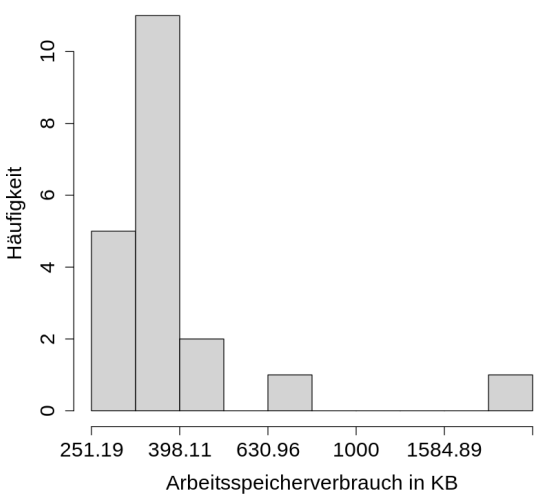


Abbildung 5.10: Häufigkeiten des Arbeitsspeicherverbrauchs für 10x10 Instanzen

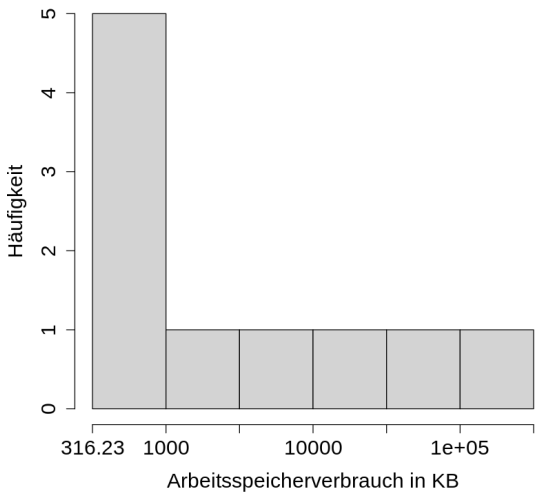


Abbildung 5.11: Häufigkeiten des Arbeitsspeicherverbrauchs für 12x12 Instanzen

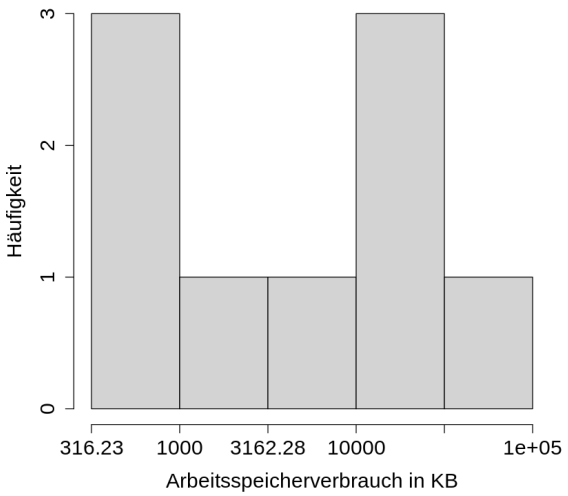


Abbildung 5.12: Häufigkeiten des Arbeitsspeicherverbrauchs für 13x13 Instanzen

5.4 Einfluss der Variablenordnung

Um den Einfluss der Variablenordnung zu bewerten, wurden die folgenden drei Varianten verwendet, um Rätsel verschiedener Größen zu lösen:

1. **Keine Sortierung:** Die Variablen werden nicht vorab optimiert und verbleiben in der Reihenfolge, in der diese generiert werden. Dies Entspricht im Falle unserer Implementierung einer Reihenfolge, bei der zuerst sämtliche horizontal angeordneten Dominos und anschließend die vertikal angeordneten Dominos untersucht werden.

2. **Sortierung nach Zellen-Index:** Dies entspricht Schritt eins aus der in 4.2.1 beschriebenen Sortierung.

3. **vollständige Sortierung:** Hierbei wird die in 4.2.1 beschriebene Sortierung vollständig durchgeführt.

Die Ergebnisse der Messungen werden in Tabelle 5.3 dargestellt. Für die Messungen wurde ein Timeout von 15 Minuten festgelegt. Bei den 10x10 Instanzen ohne Sortierung kam es dabei zu sehr vielen Timeouts, weshalb hier kein Durchschnitt angegeben ist. Im Vergleich zu keiner Sortierung konnte bei Rätseln der Größe 6x6 eine Beschleunigung um das 15,8-fache erreicht werden, bei Rätseln der Größe 8x8 sogar um das 485,2-fache. Zwischen der Sortierung nach Zellen-Index und der vollständigen Sortierung stieg der Beschleunigungsfaktor ebenfalls von 2,6 bei 6x6-Instanzen, auf 8,7 bei 8x8-Instanzen und 9,67 bei 10x10-Instanzen.

Variablenordnung	6x6	8x8	10x10
keine Sortierung	0,5351	308,1597	-
nach Zellen-Index	0,0886	5,5356	29,624
vollständige Sortierung	0,0339	0,6351	3,0619

Tabelle 5.3: Durchschnittliche Laufzeit in Sekunden für verschiedene Rätselgrößen und Variablenordnungen

5.5 Integer Programming Solver

Zur Einordnung der Leistungsfähigkeit des ZDD-basierten Solvers wurde zusätzlich ein Ansatz aus Basis von *Integer-Programming* implementiert und getestet. Dabei wird ein Rätsel in ein lineares Gleichungssystem überführt, welches mittels des *PulP-Solvers*[2] gelöst wird. Dabei wurden drei Arten von Bedingungen als Gleichungen modelliert:

- Jeder Bereich hat exakt zwei markierte Zellen
- Jede markierte Zelle hat mindestens einen Nachbarn
- Jede markierte Zelle hat maximal einen Nachbarn

Tabelle 5.4 zeigt die durchschnittlichen Laufzeiten sowie deren Varianz für den Integer-Programming Solver. Bei einer Spielfeldgröße von 6x6 ist die benötigte Zeit zum Lösen eines Rätsels zwar etwas höher als beim ZDD-Ansatz, allerdings steigen die Zeiten mit wachsender Spielfeldgröße wesentlich schwächer an, weshalb bereits ab einer Größe von 8x8 die Leistung des ZDD-Solvers übertroffen wird. Auf Grund der deutlich geringeren Laufzeit war es möglich, den Solver auf Rätseln bis zu einer Größe von 20x20 zu testen, wobei mit steigender Größe die Verfügbarkeit von testbaren Instanzen immer weiter abnimmt. Abbildung 5.13 zeigt die durchschnittlich benötigten Zeiten im Vergleich zueinander.

[2] https://pypi.org/project/PuLP/

Rätselgröße	getestete Instanzen	Laufzeit	Varianz
6x6	100	0,0488	0,0023
8x8	100	0,0537	0,002
10x10	100	0,0629	0,0023
12x12	10	0,0789	0,0001
13x13	10	0,0879	0,0002
14x14	3	0,0901	0
15x15	15	0,106	0,0003
16x16	6	0,1204	0,0002
17x17	2	0,1361	0,0001
18x18	1	0,107	-
19x19	1	0,1176	-
20x20	1	0,1847	-

Tabelle 5.4: Durchschnittliche Laufzeiten nach Rätselgröße mit Integer-Programming Solver

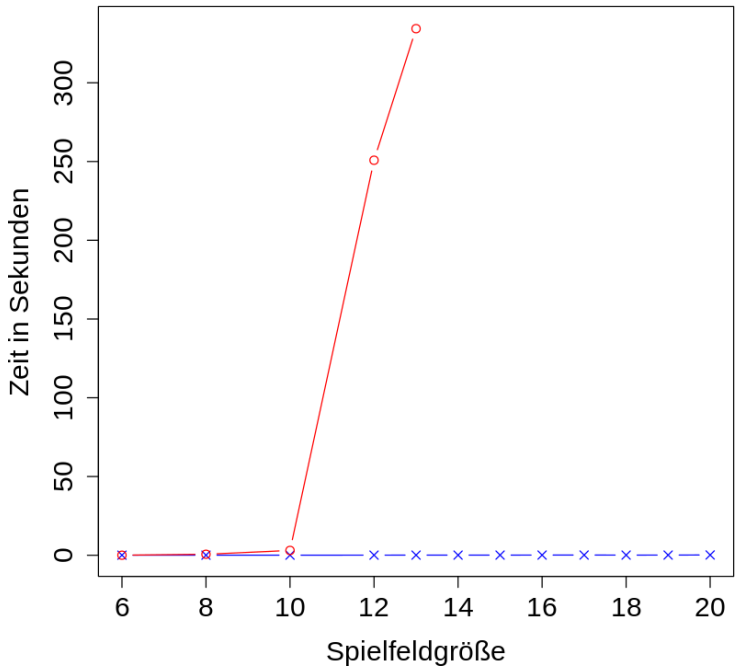


Abbildung 5.13: Vergleich der Laufzeiten vom ZDD-Solver (rot) und dem Integer-Programming-Solver (blau)

6 Diskussion

In diesem Kapitel werden die in Kapitel 5 präsentierten Ergebnisse interpretiert und analysiert. Zunächst werden die Messwerte bezüglich der Laufzeit und des Speicherverbrauchs bewertet und bezogen auf die allgemeine Praktikabilität des Algorithmus eingeordnet. Anschließend wird das Ergebnis dieser Arbeit mit bestehender Literatur verglichen. Daraufhin wird der wissenschaftliche Beitrag den diese Arbeit geleistet hat diskutiert, gefolgt von einer Besprechung der Limitationen der Arbeit.

6.1 Bewertung der Ergebnisse

Sowohl die benötigte Zeit, als auch der benötigte Arbeitsspeicher steigen mit wachsender Rätselgröße enorm an. Der Arbeitsspeicherverbrauch steigt dabei im Durchschnitt nicht monoton, allerdings wachsen die Werte im Median monoton und weisen dabei eine enorme Varianz auf, was die Durchschnittswerte stark verfälschen kann. Daher ist davon auszugehen, dass mit einer größeren Testgruppe die Durchschnittswerte ebenfalls monoton wachsen würden. Als limitierender Faktor für den Algorithmus hat sich eindeutig die Zeit herausgestellt, nicht der Arbeitsspeicher. Dieser steigt zwar mit wachsender Rätselgröße auch stark an, jedoch erreicht er bei keinem der in dieser Arbeit untersuchten Rätseln Werte, die nicht von gängigen Computern geleistet werden können. Zwar konnten bei den Größen 6x6 und 8x8 noch Durchschnittswerte von unter einer Sekunde erreicht werden, jedoch wurden bereits ab einer Größe von 10x10 Werte erreicht, die, besonders in Kombination mit einer erheblichen Varianz, den vorgestellten Algorithmus für den tatsächlichen Gebrauch zum Lösen von Norinori-Rätseln unpraktikabel machen. Ein starkes Wachstum der Laufzeit war angesichts der NP-Vollständigkeit von Norinori zwar zu erwarten, jedoch zeigt der Vergleich mit dem Integer-Programming-Solver dass eine effiziente Lösung von erheblich größeren Rätseln möglich ist.

Hinsichtlich der Variablenordnung konnte eindrucksvoll gezeigt werden, dass diese einen signifikanten Einfluss auf die Performance des Algorithmus hat. Der erste Schritt der Sortierung hatte dabei einen deutlich größeren Einfluss als der zweite. Obwohl für Rätsel der Größe 10x10 nicht alle Messwerte vorliegen, deutet die Tendenz darauf hin, dass der Beschleunigungsfaktor mit zunehmender Rätselgröße weiter zunimmt. Dies deutet auf eine exponentielle Beschleunigung hin, dessen Einfluss mit steigender Rätselgröße weiter zunimmt. Es bleibt jedoch unklar, ob diese Entwicklung bei immer größeren Rätseln bestehen bleibt, oder neue Limitationen auftreten, die diesen Effekt einschränken könnten.

6.2 Vergleich mit bestehender Literatur

Yoshinaka et al. präsentierten zwei Algorithmen, welche, ebenfalls mittels ZDD-Konstruktion, die Rätsel Slitherlink und Numberlink lösen [14]. Für beide Rätsel konnten fast durchweg wesentlich bessere Zeiten als mit einem CPLEX- oder Sugar-basierten Solver erreicht werden, wobei der Sugar-basierte Solver bei der größten verfügbaren Numberlink Instanz jedoch besser abschnitt. Bei Slitherlink konnte der vorgestellte Algorithmus bei mehreren der größten Instanzen mit sowohl mit dem CPLEX-, als auch mit dem Sugar-basierten Solver nicht mithalten. Diese Ergebnisse spiegeln eine Tendenz wider, welche auch bei den Ergebnissen dieser Arbeit erkennbar ist: Bei größeren Problemen scheinen andere Solver besser abzuschneiden, als ein ZDD-Konstruktionsalgorithmus, wobei dieser Punkt in dieser Arbeit deutlich früher erreicht wurde. Daher ist das Lösen dieser Rätsel eher als ein Proof of Concept anzusehen, als eine effiziente Lösungsmethode. Besonders betont wird dies durch den Vergleich zu einem maßgeschneiderten Slitherlink Solver[14], welcher in jedem Fall deutlich bessere Performance aufwies als der ZDD-basierte Solver. Ishahata et al. war im Gegensatz dazu in der Lage, mit seinem ZDD-Konstruktionsalgorithmus für Nageraru-Rätsel bei jeder Rätselgröße wesentlich bessere Ergebnisse zu erzielen, als mit einem Sugar-basierten Solver [7].

Die Algorithmen von Yoshinaka et al. und Ishahata et al. haben gemeinsam, dass sie als Kreis- bzw. Pfadfindungsproblem repräsentiert werden können und in Folge dessen bei ihrer Zustandsdefinition die von Knuth vorgestellte[10] *mate*-Tabelle verwenden können. In dieser Hinsicht unterscheiden sich die Algorithmen von dem in dieser Arbeit vorgestellten, da die Darstellung des Zustandes grundlegend anders funktioniert. Die Effizienz dieser Algorithmen konnte in dieser Arbeit jedoch nicht erreicht werden. Mögliche Gründe hierfür werden im Abschnitt 6.4 näher besprochen. Bemerkenswert ist ebenfalls, dass sowohl Yoshinaka et al. als auch Ishahata et al. ihre Ergebnisse mit lediglich der Optimierung des Zustandsmanagements und der Abbruchbedingungen in der *getChild*-Funktion erreichen konnten. Im Gegensatz dazu wurde ein maßgeblicher Teil der Leistungssteigerung des hier präsentierten Algorithmus durch die Anpassung der Variablenordnung erreicht. Aus diesen Unterschieden resultierende Forschungsfragen werden im Fazit näher besprochen.

6.3 Beitrag der Arbeit

In dieser Arbeit wurde gezeigt, dass es möglich ist, Norinori-Rätsel mittels ZDD-Konstruktion zu lösen und damit das Gebiet der untersuchten Anwendungsfälle von ZDDs erweitert. Dabei stellt diese Arbeit die erste dar, die sich mit der algorithmischen Lösung von Norinori-Rätseln befasst. Zugleich wurde jedoch gezeigt, dass der untersuchte Ansatz nicht wettbewerbsfähig ist und zur Lösung von Norinori-Rätseln ein Integer-Programming-Ansatz deutlich bessere Ergebnisse erzielt. Des Weiteren konnten die von Bryant[5] getätigten Aussagen zum Einfluss der Reihenfolge der Variablen anhand konkreter Zahlen anschaulich bestätigt werden. Hierdurch wird noch einmal unterstrichen, welche Bedeutung die Variablenordnung bei der Nutzung von ZDDs zur Lösung von Problemen hat.

6.4 Limitationen der Arbeit

Die Reihenfolge, in welcher die Variablen untersucht werden, hat sich als kritischer Faktor für die Effizienz dieses Algorithmus herausgestellt. Laut Bollig et al. [4] ist das Finden der optimalen Variablenordnung selbst jedoch ein NP-vollständiges Problem. Daher basiert die Variablenordnung ausschließlich auf Heuristiken. Es bleibt also ungeklärt, ob durch weitere Heuristiken weitere Leistungssprünge in der Performance des Algorithmus erreicht werden könnten. Ein weiterer kritischer Aspekt des Algorithmus ist die Optimierung der *getChild*-Funktion durch das frühzeitige Ausschließen von Zuständen. Optimierungen hier können den Suchraum exponentiell verkleinern und dadurch ebenfalls zu massiven Performanceverbesserungen führen. Wie bei der Variablenordnung ist auch hier nicht abschließend geklärt, ob die gewählten Ausschlusskriterien optimal sind. Der Vergleich mit anderen Arbeiten wie von Yoshinaka et al.[14] und Ishahata et al.[7] zeigt, dass es durchaus möglich ist, gewisse Rätsel mittels ZDDs effizient zu lösen. Daraus ergibt sich die Frage, ob Norinori ein geeigneter Testfall für ZDDs ist, oder ob dessen spezifische Eigenschaften die Effizienz eines ZDD-basierten Lösungsansatzes allgemein beeinträchtigen.

7 Fazit

Diese Arbeit sollte die Frage beantworten, inwiefern ZDDs anwendbar sind, um Norinori-Rätsel zu lösen. Dabei konnte gezeigt werden, dass es mittels dieser durchaus möglich ist, alle Lösungen für ein gegebenes Rätsel zu bestimmen. Bei kleinen Rätseln liefert der vorgestellte Ansatz sogar gute Ergebnisse, allerdings nimmt die Effizienz mit steigender Rätselgröße sehr schnell ab, weshalb man nicht von einem allgemein praktikablen Algorithmus sprechen kann. Vor allem der Vergleich mit alternativen Lösungsansätzen hat dies bestätigt. Darüber hinaus konnte dargelegt werden, dass die Variablenordnung ein nicht zu vernachlässigender Faktor bei der Optimierung von Algorithmen, die auf ZDDs beruhen, darstellt.

Diese Ergebnisse deuten darauf hin, dass ZDDs möglicherweise nicht für jede Art von Problem gleichermaßen geeignet sind, weshalb es wichtig ist, die Natur eines Problems sorgfältig zu analysieren, bevor sich für einen ZDD-basierten Ansatz entschieden wird.

Es bleiben einige unbeantwortete Fragen, die in zukünftigen Arbeiten aufgegriffen werden können: Sind ZDDs für diesen Anwendungsfall ungeeignet oder gibt es weitere Optimierungen, welche den Algorithmus effizient genug machen, um mit anderen Lösungsmethoden zu konkurrieren? Gibt es weitere oder alternative Heuristiken für die Variablenordnung, die zu besseren Ergebnissen führen? Falls ZDDs *generell* ungeeignet für das Lösen von Norinori-Rätseln sein sollten: Worin unterscheiden sich Probleme, für die ZDDs geeignet sind von denen, für die sie es nicht sind? Was für Charakteristiken sind maßgeblich verantwortlich für den Unterschied?

Bezüglich Norinori-Rätseln bleibt die Frage offen, wie man sie am effizientesten löst. Ein *Integer-Programming-Solver* war in der Lage den vorgestellten Algorithmus zu schlagen, aber inwiefern lassen sich diese Ergebnisse noch überbieten?

Zusammenfassend kann man sagen, dass ZDDs geeignet sind, um kleine Norinori-Rätsel zu lösen, zum effizienten lösen von größeren Rätseln allerdings alternative Methoden verwendet werden sollten, um eine bessere Performance zu erzielen.

Literaturverzeichnis

[1] Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[2] Henrik Reif Andersen and Henrik Hulgaard. Boolean expression diagrams. *Information and Computation*, 179(2):194–212, 2002.

[3] Michael Biro and Christiane Schmidt. Computational complexity and bounds for norinori and lits. In *Proc. 33rd European Workshop on Computational Geometry, Malmö, Sweden*, pages 29–32, 2017.

[4] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[5] Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[6] Jean-Guillaume Dumas, Pascal Lafourcade, Daiki Miyahara, Takaaki Mizuki, Tatsuya Sasaki, and Hideaki Sone. Interactive physical zero-knowledge proof for norinori. In *Computing and Combinatorics: 25th International Conference, COCOON 2019, Xi'an, China, July 29–31, 2019, Proceedings 25*, pages 166–177. Springer, 2019.

[7] Masakazu Ishihata and Fumiya Tokumasu. Solving and generating nagareru puzzles. In *20th International Symposium on Experimental Algorithms (SEA 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[8] Hiroaki Iwashita and Shin ichi Minato. Tcs-tra-1369 tcs technical report efficient top-down zdd construction techniques using recursive specifications by. 2013.

[9] J. Jain, M. Abadir, J. Bitner, D.S. Fussell, and J.A. Abraham. Ibdds: an efficient functional representation for digital circuits. In *[1992] Proceedings The European Conference on Design Automation*, pages 440–446, 1992.

[10] Donald E. Knuth. *The Art of Computer Programming, Vol. 4a: Combinatorial Algorithms, Part 1*, chapter 7.1.4 Binary Decision Diagrams. Addison Wesley Professional, 2011.

[11] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, page 272–277, New York, NY, USA, 1993. Association for Computing Machinery.

[12] Shin-ichi Minato. Calculation of unate cube set algebra using zero-suppressed bdds. In *Proceedings of the 31st annual Design Automation Conference*, pages 420–424, 1994.

[13] Noq. Norinori solver. https://www.noq.solutions/norinori. Zugriff am: 08.10.2024.

[14] Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by zdds. *Algorithms*, 5(2):176–213, 2012.