Lösen von Slitherlink Puzzles mithilfe von Zero-suppressed Decision Diagrams im Vergleich zu anderen Lösungsmethoden

Bachelorarbeit

Jakob Berschneider
453351

4. November 2024

Gutachter: Prof. Dr. Benjamin Blankertz
Dr.- Ing. Stefan Fricke

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

Kurzfassung

Slitherlink ist ein beliebtes japanisches Logikrätsel. Zahlreiche Veröffentlichungen haben sich bereits damit beschäftigt, wie Slitherlink Puzzles effizient durch Computer gelöst werden können. Diese Arbeit präsentiert verschiedene Lösungsmethoden und vergleicht diese in Hinblick auf Rechenzeit. Neben Integer Programming (IP) und SAT-Modellierung wird insbesondere auch ein Lösungsverfahren unter Verwendung der Graphenbibliothek Graphillion, welche auf Zero-suppressed Binary Decision Diagrams (ZDDs) basiert, untersucht. Zudem wird ein Presolver vorgestellt, der anhand verbreiteter Muster Teillösungen für Slitherlink Puzzles vorberechnet. Für den Vergleich wurden die unterschiedlichen Methoden in Python implementiert und für insgesamt 533 Instanzen verschiedener Größen getestet. Die Ergebnisse zeigen, dass der SAT-Solver in den meisten Fällen die beste Leistung erbringt. Der Presolver konnte zudem signifikante Verbesserungen erzielen, indem er viele Puzzles vollständig löste oder den Suchraum für alle anderen Methoden reduzierte. Der ZDD-Ansatz zeigte deutliche Schwankungen, insbesondere bei größeren Puzzles, und war insgesamt weniger leistungsfähig. Bei der Suche nach allen Lösungen für eine modifizierte Instanz war der ZDD-Ansatz wiederum mehr als doppelt so schnell wie der SAT-Solver.

Inhaltsverzeichnis

1 Einleitung **1**

2 Methoden **3**
2.1 Integer Programming 3
2.2 SAT-Modellierung 9
2.3 Zero-Suppressed Decision Diagrams 13
2.3.1 ZDD-Operationen 15
2.3.2 ZDDs zum Aufzählen von möglichen Pfaden oder Kreisen 16
2.3.3 Lösen von Slitherlink mithilfe von ZDDs 20
2.4 Verwendung eines Presolvers 22
2.4.1 Färben von Slitherlink-Zellen 22
2.4.2 Lokale Reduktionsregeln 23
2.4.3 Abgeschlossene Regionen 25
2.4.4 Laufzeitabschätzungen 26

3 Ergebnisse **28**
3.1 Vergleich von IP-Solvern 28
3.2 Vergleich von SAT-Solvern 29
3.3 Vergleich von IP, SAT und ZDD 30
3.4 Suche nach allen gültigen Lösungen 32
3.5 Verwendung des Presolvers 32

4 Diskussion **36**

5 Fazit **39**

Abbildungsverzeichnis

1.1 Slitherlink Puzzle (links) und eingezeichnete Lösung (rechts). 1

2.1 Lösung mit drei Zyklen für ein Slitherlynx Puzzle. 3
2.2 Slitherlink Puzzle, welches nur Nullen als Zelleneinträge hat. 4
2.3 Lösung mit zwei Zyklen für ein Slitherlynx-Puzzle. Der rot markierte Kreis würde ohne den anderen Kreis eine Lösung für die zugrundeliegende Slitherlink-Instanz liefern. 5
2.4 Iterativer Lösungsprozess für Slitherlink mittels ILP-Modellierung. 6
2.5 Iterativer Lösungsprozess für Slitherlink mittels SAT-Modellierung. 12
2.6 Einfaches ZDD. Gestrichelte Linien stehen für 0-arcs, durchgezogene Linien stehen für 1-arcs. Angepasst von [16]. 14
2.7 Unreduziertes ZDD. 15
2.8 Gittergraph $G_{3,3}$ mit Nummerierung der Knoten. 16
2.9 Simulation der ersten Schritte des SIMPATH-Algorithmus zur ZDD-Generierung anhand des Graphen $G_{3,3}$. 19
2.10 Reduziertes ZDD für alle Pfade eines Graphen $G_{3,3}$. Angepasst von [7]. 20
2.11 Generierung von Slitherlink-Lösungen mit Graphillion anhand einer Beispielinstanz. Angelehnt an [3]. 22
2.12 Initiale Färbung für das Slitherlink-Puzzle aus Abbildung 1.1. 23
2.13 Regel für 0-Zellen. Angepasst von [12]. 24
2.14 Regel für 3-Zellen mit benachbarter Zelle der selben Farbe. Angepasst von [12]. . . . 24
2.15 Regel für 3-Zellen in einer Ecke. Angepasst von [14]. 24
2.16 Regel für 2-Zellen neben einer Linie. Angepasst von [14]. 25
2.17 Regel für diagonal benachbarte 1-Zellen. Angepasst von [14]. 25
2.18 Erkennen von abgeschlossenen Regionen im teilweise gelösten Puzzle aus Abbildung 2.12. 26

3.1 Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für verschiedene IP-Solvern. 29
3.2 Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für verschiedene SAT-Solvern. 29
3.3 Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für IP, SAT und ZDD. 30
3.4 Verteilung der gemessenen Zeiten für die verschiedenen Puzzles der Größe (10, 10) mit IP, SAT und ZDD. 31
3.5 Verteilung der gemessenen Zeiten für die verschiedenen Puzzles der Größe (12, 16) mit IP, SAT und ZDD. 31
3.6 Vergleich der Zeiten für die Suche nach allen gültigen Lösungen mit SAT und ZDD. . 32

3.7 Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für den Presolver und SAT. Es wurden nur solche Instanzen getestet, die vollständig durch den Presolver lösbar sind. 33
3.8 Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für IP ohne und mit Verwendung des Presolvers. 34
3.9 Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für SAT ohne und mit Verwendung des Presolvers. 35
3.10 Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für ZDD ohne und mit Verwendung des Presolvers. 35

Tabellenverzeichnis

2.1 Potenziell gültige Linienbelegung für innere Knoten. Alle Ungleichungen sind erfüllt. 7
2.2 Ungültige Linienbelegung für innere Knoten. Es exsitiert jeweils eine ungültige Ungleichung für diese Variablenbelegungen. 8
2.3 Operationen zur Manipulation von Mengenfamilien nach [3]. 15
2.4 Potentiell mögliche und verbotene Teilgraphen bei vorgegebenen Zahlen im Puzzle. . 21

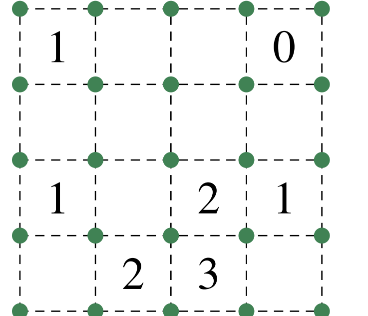
3.1 Vergleich der Korrektheit für verschiedene IP-Solver für alle untersuchten Puzzles. . 28
3.2 Vergleich der Korrektheit für SAT, IP und ZDD für alle untersuchten Puzzles. 32
3.3 Verbesserung der durchschnittlichen Laufzeiten durch Verwendung des Presolvers in Sekunden. IP*, SAT* und ZDD* meinen hierbei die optimierten Versionen. 34

4.1 Kürzeste und längste Rechenzeiten des ZDD-Solvers für Puzzles der Größe (12, 16). 36

1 Einleitung

Die Entwicklung von Lösungsalgorithmen für Puzzles hat innerhalb des Gebiets der Algorithmen und Datenstruktren große Aufmerksamkeit erlangt. Logische Rätsel wie Slitherlink bieten konkrete Problemstellungen anhand weniger Regeln, was sie ideal für die praktische Untersuchung unterschiedlicher algorithmischer Ansätze machen. So können Puzzles nicht als bloßes Unterhaltungsmedium verstanden werden, sondern eine Umgebung für das Testen und Optimieren von Lösungsmethoden darstellen.
Slitherlink ist ein Logikrätsel, das erstmals 1989 durch den japanischen Sudoku-Herausgeber Nikoli veröffentlicht wurde. Es gehört zu den beliebtesten Puzzles von Nikoli und ist z.B. auch unter den Namen Fences, Number Line, Sli-Lin, Takegaki oder Rundweg bekannt. Neben Puzzles wie Numberlink oder Masyu gehört Slitherlink zu den Link Puzzles. Bei dieser Art von logischen Rätseln ist es die Aufgabe des Spielers, Pfade oder Kreise in einem Gitter zu finden unter der Berücksichtigung lokaler und globaler Bedingungen.
Das Spielfeld von Slitherlink besteht aus einem rechteckigen $n \times m$ Gittergraph. In den einzelnen Zellen können ganze Zahlen im Bereich von 0 bis 3 stehen. Das Ziel ist es, benachbarte Knoten so zu verbinden, dass ein Zyklus entsteht. Dabei sind die folgenden drei Regeln zu beachten[1]:

1. Verbinde benachbarte Knoten mit horizontalen oder vertikalen Kanten, sodass genau ein Zyklus entsteht.
2. Die Zahlen geben an, wie viele Linien eine Zelle umgeben müssen. Leere Zellen können durch beliebig viele Linien umgeben werden.
3. Der Zyklus kreuzt sich nicht selbst und es gibt keine Verzweigungen.



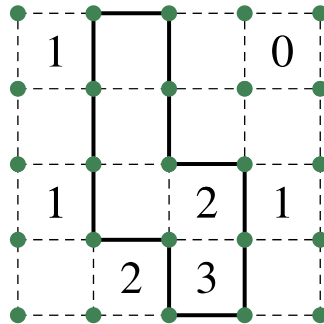


Abbildung 1.1: Slitherlink Puzzle (links) und eingezeichnete Lösung (rechts).

Gut designte Slitherlink Puzzles haben eine eindeutige Lösung. Abbildung 1.1 zeigt exemplarisch ein Puzzle und dessen Lösung.

[1] https://www.nikoli.co.jp/en/puzzles/slitherlink/

Es gibt mehrere veröffentlichte Arbeiten, welche sich mit Slitherlink beschäftigen. Yato und Seta [15] haben 2003 erstmals nachgewiesen, dass Slitherlink innerhalb der Klasse der NP-vollständigen Probleme liegt. Das heißt unter der Annahme, dass P $\neq$ NP, gibt es keinen Algorithmus, um jede Slitherlink-Instanz in Polynomzeit lösen zu können, und jedes Problem aus der Klasse NP kann mit polynomiellen Zeitaufwand auf eine Slitherlink-Instanz reduziert werden.
Eine verbreitete Lösungsmöglichkeit für Puzzles bietet der Constraint-basierte Ansatz. Dabei generiert man aus einem gegebenen Spielfeld ein mathematisches Modell, welches sich dann durch darauf spezialisierte Solver lösen lässt. Für Slitherlink haben Miyauchi und Tanaka [11] eine SMT (Satisfiability Modulo Theories)-Lösung entwickelt, welche für einige Fälle bessere Ergebnisse als vorherige Slitherlink-Solver erreichen konnte. SMT ist eine Verallgemeinerung des Erfüllbarkeitsproblems der Aussagenlogik (SAT), welches sich mit der Erfüllbarkeit von aussagenlogsichen Formeln beschäftigt. Ein einfacherer Ansatz, der die Formulierung von Slitherlink als Integer Linear Program (ILP) beinhaltet, wurde durch Sugimura [17] vorgestellt. Im Rahmen dieser Arbeit soll der ILP-Ansatz und dessen Übersetzung als SAT-Problem in den Abschnitten 2.1 und 2.2 untersucht werden.
Zero-suppressed Decision Diagrams (ZDDs) wurden als Datenstruktur erstmals 1993 durch Minato [9] vorgestellt. Ein ZDD ist eine Spezialform eines binären Entscheidungsdiagramms (BDD), mit dem sich Mengenfamilien kompakt darstellen und effizient modifizieren lassen. Gemeinsam mit weiteren Autoren hat Minato [10] einen Solver für Slitherlink entworfen. Dieser generiert ein ZDD, welches alle Lösungen für eine gegebene Slitherlink-Instanz repräsentieren soll. Später hat ein Teil der am ZDD-Algorithmus für Slitherlink beteiligten Autoren um Minato [3] mit Graphillion eine ZDD-basierte Graphenbibliothek für Python vorgestellt, mit der sich Slitherlink in einer einfacheren Weise lösen lässt. Der Abschnitt 2.3 dieser Arbeit beschäftigt sich mit ZDDs und dem Lösen von Slitherlink mithilfe von Graphillion.
Darüber hinaus wird im Abschnitt 2.4 ein Presolver für Slitherlink vorgestellt. Ein menschlicher Spieler geht beim Lösen von Slitherlink für gewöhnlich so vor, dass er das Puzzle auf bekannte Muster zu untersucht und so kleinere Teillösungen bestimmt, die sich dann zu einem geschlossenen Kreis zusammenfügen. In unterschiedlichen Puzzles lassen sich oft wiederkehrende Muster finden. Diese Arbeit soll untersuchen, inwiefern solche Muster dazu beitragen können, ein Puzzle vollständig zu lösen oder den Suchraum für die anderen Lösungsmethoden zu reduzieren.
Das Kapitel 3 stellt die Messergebnisse für die verschiedenen Solver, die im Kapitel 2 vorgestellt werden, für mehrere Szenarien vergleichend gegenüber. Im Kapitel 4 werden die Messergebnisse interpretiert und mit bereits bestehenden Arbeiten verglichen, bevor im Kapitel 5 ein abschließendes Fazit gezogen werden soll.

2 Methoden

Im Folgenden sollen einige Methoden zum Lösen einer Slitherlink-Instanz vorgestellt werden. Eine formalere Beschreibung für Slitherlink-Instanzen soll im Weiteren die folgende Definition geben:

Definition 1: Slitherlink (angepasst von [16])

Eine Slitherlink-Instanz sei gegeben durch das Tupel $(G_{n,m}, h)$. Dabei sei $G_{n,m} = (V, E)$ ein $n \times m$ Gittergraph und $h(E') : 2^E \rightharpoonup \{0, 1, 2, 3\}$ eine partielle Funktion für $E' \subseteq E$ mit $|E'| = 4$. Eine Lösung für eine gegebene Instanz $(G_{n,m}, h)$ ist ein Kreis C in $G_{n,m}$ falls $\forall E' \in \mathrm{Dom}(h) : h(E') = |C \cap E'|$.

2.1 Integer Programming

Integer Linear Programming (IP) ist eine leistungsfähige Methode zur Lösung von linearen Optimierungsproblemen (LP), wobei die Entscheidungsvariablen ganzzahlige Werte annehmen. Die Technik findet breite Anwendung z.B. auch in der Lösung von Puzzles. Das Lösen von Integer LP (ILP) gilt als NP-vollständig [4]. Es gibt eine Vielzahl von hochwertigen proprietären Solvern für diese Probleme. Verbreitete Beispiele sind Gurobi[1] oder CPLEX[2].
Ein Ansatz um Slitherlink mit Integer Programming zu lösen wurde von Yuka Sugimura [17] vorgestellt. Dabei wird ein Subproblem Slitherlynx formuliert, welches mithilfe von ILP-Solvern gelöst werden kann.
Bei der Modellierung von Slitherlink als ILP stellt sich die Bedingung "genau ein Zyklus" als größte Herausforderung heraus, da man für diese Bedingung globale Informationen über das gesamte Puzzle benötigt. Slitherlynx hat die selben Regeln wie Slitherlink, jedoch wird die Bedingung, das genau ein Zyklus entstehen muss, aufgegeben und auch mehrere Zyklen als Lösung akzeptiert. Die Zyklen müssen außerdem disjunkt sein bzw. sie dürfen sich nicht kreuzen. Die Abbildung 2.1 zeigt eine gültige Slitherlynx-Lösung für das Slitherlink Puzzle aus Abbildung 1.1.

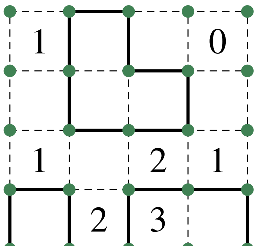


Abbildung 2.1: Lösung mit drei Zyklen für ein Slitherlynx Puzzle.

[1] https://www.gurobi.com/
[2] https://www.ibm.com/de-de/products/ilog-cplex-optimization-studio/cplex-optimizer

Definition 2: Slitherlynx

Eine Slitherlynx-Instanz sei gegeben durch das Tupel $(G_{n,m}, h)$. Dabei sei $G_{n,m} = (V, E)$ ein $n \times m$ Gittergraph und $h(E') : 2^E \rightharpoonup \{0, 1, 2, 3\}$ eine partielle Funktion für $E' \subseteq E$ mit $|E'| = 4$. Eine Lösung für eine gegebene Instanz $(G_{n,m}, h)$ ist eine nicht-leere Menge K an Kreisen C_i in $G_{n,m}$ falls $\forall E' \in \mathrm{Dom}(h) : h(E') = \sum_{C_i \in K} |C_i \cap E'|$ und $\forall C_1, C_2 \in K : C_1 \neq C_2 \Rightarrow C_1 \cap C_2 = \emptyset$.

Zur Modellierung von Slitherlynx als ILP wird für jede Kante $e \in E$ eine binäre Variable $x_e \in \{0, 1\}$ verwendet, welche angibt, ob die Kante in der Lösung enthalten ist (1) oder nicht enthalten ist (0). Da alle Variablen binär sind, handelt sich um ein 0-1-ILP. Es sind die folgenden Nebenbedingungen zu erfüllen:

$$\sum_{e \in E} x_e > 0 \tag{2.1}$$
$$\forall v \in V \text{ mit } |N(v) = \{a, b, c, d\}| = 4 : \quad x_a + x_b + x_c + x_d \leq 2 \tag{2.2}$$
$$-x_a + x_b + x_c + x_d \geq 0 \tag{2.3}$$
$$x_a - x_b + x_c + x_d \geq 0 \tag{2.4}$$
$$x_a + x_b - x_c + x_d \geq 0 \tag{2.5}$$
$$x_a + x_b + x_c - x_d \geq 0 \tag{2.6}$$
$$\forall v \in V \text{ mit } |N(v) = \{a, b, c\}| = 3 : \quad x_a + x_b + x_c \leq 2 \tag{2.7}$$
$$-x_a + x_b + x_c \geq 0 \tag{2.8}$$
$$x_a - x_b + x_c \geq 0 \tag{2.9}$$
$$x_a + x_b - x_c \geq 0 \tag{2.10}$$
$$\forall v \in V \text{ mit } |N(v) = \{a, b\}| = 2 : \quad -x_a + x_b \geq 0 \tag{2.11}$$
$$x_a - x_b \geq 0 \tag{2.12}$$
$$\forall E' \in \mathrm{Dom}(h) : \quad \sum_{e \in E'} x_e = h(E') \tag{2.13}$$

Durch die Bedingung 2.13 wird gewährleistet, dass alle Zellen mit der korrekten Anzahl an Linien umgeben sind. Die Bedingung 2.1 sorgt dafür, dass die Lösungsmenge nicht leer sein kann bzw. mindestens eine Kante in der Lösung enthalten ist. Sie ist notwendig, da es Puzzles wie in Abbildung 2.2 geben kann, die auch ohne eine Kante bereits alle Zellen entsprechend ihrer Zahl abdecken.

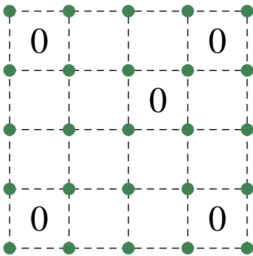


Abbildung 2.2: Slitherlink Puzzle, welches nur Nullen als Zelleneinträge hat.

Die Ungleichungen 2.2-2.12 sichern die Kreiseigenschaft, dass alle Knoten in einem Kreis Grad 2 haben müssen. Alle anderen Knoten des Eingabegraphen $G_{n,m}$ dürfen keine Kante in einem Lösungskreis haben, das heißt sie müssen Grad 0 haben. In einem Gittergraphen können nur Knoten vom Grad 2 (Eckknoten), Grad 3 (sonstige Randknoten) oder Grad 4 (innere Knoten) enthalten sein. Daher wird unterschieden zwischen Knoten mit 4 Nachbarn ab 2.2, 3 Nachbarn ab 2.7 und 2 Nachbarn ab 2.11.
Die Tabellen 2.1 und 2.2 zeigen die Korrektheit der Bedingungen für innere Knoten. Es lässt sich erkennen, dass durch diese Wahl der Bedingungen eindeutig falsche Variablenbelegungen, die in offene Enden oder Verzweigungen resultieren würden, ausgeschlossen werden.
Für ein Slitherlink- bzw. Slitherlynx-Puzzle mit n Reihen und m Spalten hat der Eingabegraph $G_{n+1,m+1}$ genau 4 Ecken. An der oberen und unteren Grenze befinden sich jeweils $m-1$ Randknoten, an der linken und rechten Grenze jeweils $n-1$ Randknoten. Im Inneren gibt es $(n-1)\cdot(m-1)$ Knoten, die zu allen Seiten benachbart sind. Mit einer Anzahl an Zahlen im Puzzle z ergibt sich insgesamt die folgende Anzahl an Nebenbedingungen für das ILP:

$$1 + 2 \cdot 4 + 4(2(n-1) + 2(m-1)) + 5(n-1)(m-1) + z = 5mn + 3m + 3n + z - 2$$

Diese ILP-Modellierung liefert jedoch zunächst nur eine Lösung für Slitherlynx. Es sind also auch mehrere Kreise als Lösung zulässig. Um die Forderung nach genau einem Zyklus zu erfüllen, wird das ILP in einem iterativen Prozess nach Sugimura [17] mittels Schnittebenenverfahren (engl. cutting plane method) solange durch zusätzliche Bedingungen erweitert, bis ein zusammenhängender Kreis als Lösung gefunden wird. Durch die folgende Ungleichung werden Kreise für Sliterlynx ausgeschlossen, welche bereits zuvor exploriert wurden:

$$\forall C_i \in K: \quad \sum_{e \in C_i} x_e \leq |C_i| - 1 \tag{2.14}$$

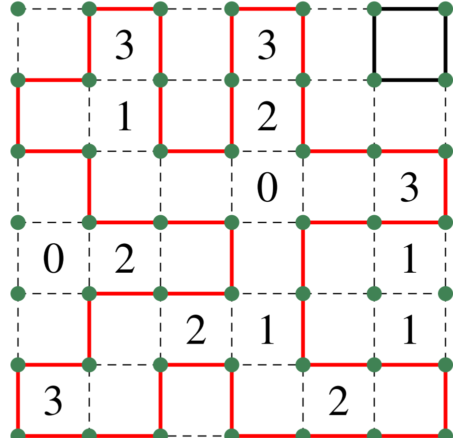


Abbildung 2.3: Lösung mit zwei Zyklen für ein Slitherlynx-Puzzle. Der rot markierte Kreis würde ohne den anderen Kreis eine Lösung für die zugrundeliegende Slitherlink-Instanz liefern.

In der Übersicht in Abbildung 2.4 wird der iterative Prozess zum Finden einer Lösung für Slitherlink schematisch dargestellt. Die Frage “Gibt es ein C_i, das M alleine löst?” meint dabei, dass überprüft wird, ob die Auswahl eines einzelnen Kreises aus der aus der Slitherlynx-Lösung ausreicht, um alle

Bedingungen für das ILP-Modell M zu erfüllen. Das heißt, es wird für alle Kreise C_i aus der Lösung K überprüft, ob die Belegung $\forall e \in E \setminus C_i : x_e = 0$ und $\forall e \in C_i : x_e = 1$ das Modell M löst. Diese Überprüfung ist notwendig, da es in Puzzles wie in Abbildung 2.3 vorkommen kann, dass es Kreise in einer Slitherlynx-Lösung gibt, die keinen Einfluss auf die Zellen mit vorgegebener Linienanzahl haben. Würde man in einem solchen Fall nur prüfen, ob $|K| = 1$, würde es das Puzzle unlösbar machen, da der Kreis, welcher die Slitherlink-Instanz alleine lösen könnte, ausgeschlossen wäre.

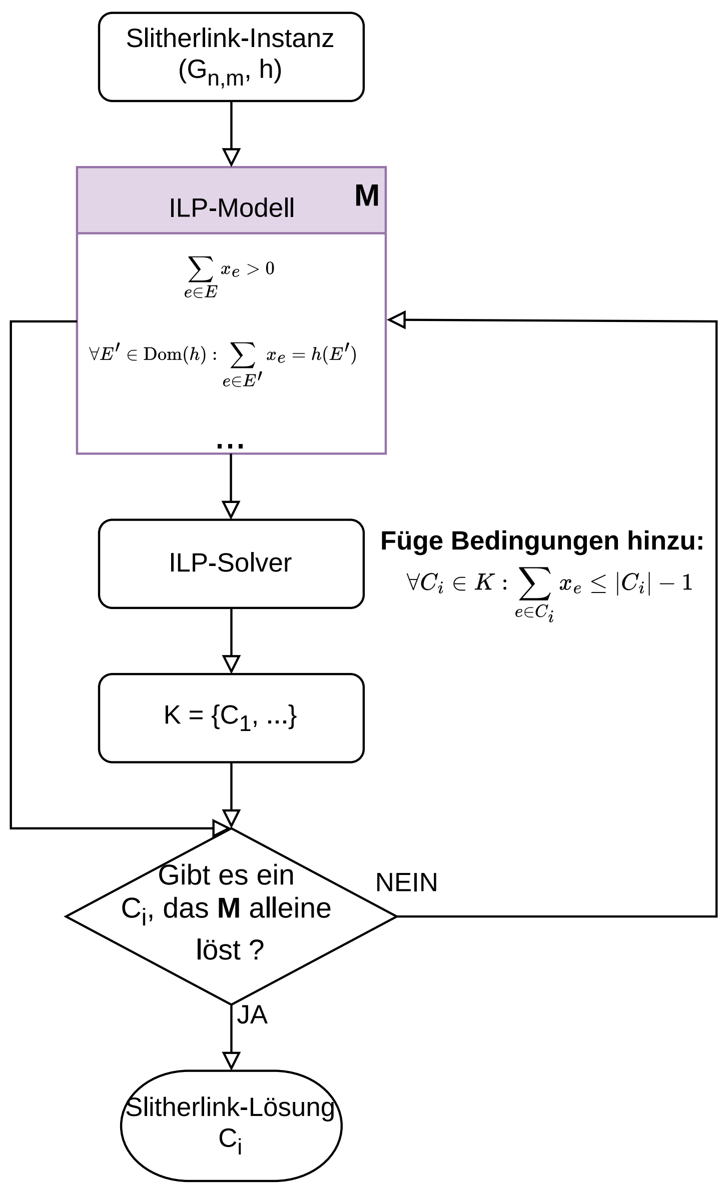


Abbildung 2.4: Iterativer Lösungsprozess für Slitherlink mittels ILP-Modellierung.

Tabelle 2.1: Potenziell gültige Linienbelegung für innere Knoten. Alle Ungleichungen sind erfüllt.

x_a	x_b	x_c	x_d	Grafik	$x_a + x_b + x_c + x_d \leq 2$	$-x_a + x_b + x_c + x_d \geq 0$	$x_a - x_b + x_c + x_d \geq 0$	$x_a + x_b - x_c + x_d \geq 0$	$x_a + x_b + x_c - x_d \geq 0$
0	0	0	0		$\top$	$\top$	$\top$	$\top$	$\top$
1	1	0	0		$\top$	$\top$	$\top$	$\top$	$\top$
1	0	1	0		$\top$	$\top$	$\top$	$\top$	$\top$
0	1	1	0		$\top$	$\top$	$\top$	$\top$	$\top$
1	0	0	1		$\top$	$\top$	$\top$	$\top$	$\top$
0	1	0	1		$\top$	$\top$	$\top$	$\top$	$\top$
0	0	1	1		$\top$	$\top$	$\top$	$\top$	$\top$

Tabelle 2.2: Ungültige Linienbelegung für innere Knoten. Es exsitiert jeweils eine ungültige Ungleichung für diese Variablenbelegungen.

x_a	x_b	x_c	x_d	Grafik	$x_a + x_b + x_c + x_d \leq 2$	$-x_a + x_b + x_c + x_d \geq 0$	$x_a - x_b + x_c + x_d \geq 0$	$x_a + x_b - x_c + x_d \geq 0$	$x_a + x_b + x_c - x_d \geq 0$
1	0	0	0	a b c d	$\top$	$\bot$	$\top$	$\top$	$\top$
0	1	0	0	a b c d	$\top$	$\top$	$\bot$	$\top$	$\top$
0	0	1	0	a b c d	$\top$	$\top$	$\top$	$\bot$	$\top$
1	1	1	0	a b c d	$\bot$	$\top$	$\top$	$\top$	$\top$
0	0	0	1	a b c d	$\top$	$\top$	$\top$	$\top$	$\bot$
1	1	0	1	a b c d	$\bot$	$\top$	$\top$	$\top$	$\top$
1	0	1	1	a b c d	$\bot$	$\top$	$\top$	$\top$	$\top$
0	1	1	1	a b c d	$\bot$	$\top$	$\top$	$\top$	$\top$
1	1	1	1	a b c d	$\bot$	$\top$	$\top$	$\top$	$\top$

2.2 SAT-Modellierung

Neben der Formulierung von Problemen als ILP gibt es einen weiteren stark verbreiteten Ansatz, um spezielle Probleme auf ein allgemeines Problem zu reduzieren. Dabei wird das Problem in eine aussagenlogische Formel übersetzt und von einem Solver für das Erfüllbarkeitsproblem der Aussagenlogik (SAT) gelöst. Es wird eine erfüllende Belegung für die Variablen der gegebenen Formel gesucht. Die Formeln werden in der Regel in Konjunktiver Normalform (KNF) angegeben. SAT gilt wie Integer Programming ebenfalls als NP-vollständig.

Wegen der Änhlichkeit zwischen IP und SAT wurde in verschiedenen Arbeiten bereits gezeigt, das sich ILP-Formulierungen in SAT-Formulierungen umwandeln lassen und diese für die identischen Probleme unterschiedlich gute Ergebnisse erzielen können [1, 8]. Im Fall von Slitherlynx, wo wir nur binäre Entscheidungsvaribablen im ILP haben, bietet sich eine SAT Formulierung des Problems besonders an, da die Variablen für SAT ebenfalls binär sind (wahr oder falsch) und es in diesem Fall keine zu optimierende Zielfunktion gibt.

Zur Modellierung von Slitherlynx als KNF wird wie bei dem ILP-Modell für jede Kante $e \in E$ eine binäre Variable s_e verwendet, welche angibt, ob die Kante in der Lösung enthalten ist (wahr) oder nicht enthalten ist (falsch). Um ein identisches Ergebnis mit dem SAT-Solver zu erreichen werden die Bedingungen wie folgt übersetzt:

Um wie in Ungleichung 2.1 auszuschließen, dass die Lösungsmenge leer ist, wird eine Klausel, die alle Variablen enthält verwendet. Falls keine Variable als wahr belegt wird, würde diese Disjunktion nicht erfüllt werden.

$$\bigvee_{e \in E} s_e \tag{2.15}$$

Genau dann, wenn $x_a + x_b + x_c + x_d \leq 2$ (2.2), können höchstens zwei der binären Variablen den Wert 1 haben. Um dies in Form einer aussagenlogischen Formel darzustellen, muss sichergestellt werden, dass jede Kombination von drei Variablen aus s_a, s_b, s_c und s_d nicht gleichzeitig als wahr belegt werden kann:

$$v \in V \text{ mit } |N(v) = \{a, b, c, d\}| = 4 : \quad \neg s_a \vee \neg s_b \vee \neg s_c \tag{2.16}$$

$$\neg s_a \vee \neg s_c \vee \neg s_d \tag{2.17}$$

$$\neg s_a \vee \neg s_b \vee \neg s_d \tag{2.18}$$

$$\neg s_b \vee \neg s_c \vee \neg s_d \tag{2.19}$$

In Tabelle 2.2 lässt sich beobachten, dass $-x_a + x_b + x_c + x_d \geq 0$ (2.3) nur für den Fall $x_b = x_c = x_d = 0$ und $x_a = 1$ nicht erfüllt wird. In anderen Worten wird die Bedingung also erfüllt, wenn $\neg(s_a \wedge \neg s_b \wedge \neg s_c \wedge \neg s_d)$. Umgeformt in KNF ergibt sich dann:

$$\forall v \in V \text{ mit } |N(v) = \{a, b, c, d\}| = 4 : \quad \neg s_a \vee s_b \vee s_c \vee s_d \tag{2.20}$$

Analog zu den Ungleichungen zu 2.2 und 2.3 können die weiteren ILP-Nebenbedingungen 2.4-2.12 für Slitherlynx wie folgt übersetzt werden:

$$
\begin{aligned}
v \in V \text{ mit } |N(v) = \{a,b,c,d\}| = 4: \quad & s_a \vee \neg s_b \vee s_c \vee s_d && (2.21)\\
& s_a \vee s_b \vee \neg s_c \vee s_d && (2.22)\\
& s_a \vee s_b \vee s_c \vee \neg s_d && (2.23)\\
\forall v \in V \text{ mit } |N(v) = \{a,b,c\}| = 3: \quad & \neg s_a \vee \neg s_b \vee \neg s_c && (2.24)\\
& \neg s_a \vee s_b \vee s_c && (2.25)\\
& s_a \vee \neg s_b \vee s_c && (2.26)\\
& s_a \vee s_b \vee \neg s_c && (2.27)\\
\forall v \in V \text{ mit } |N(v) = \{a,b\}| = 2: \quad & \neg s_a \vee s_b && (2.28)\\
& s_a \vee \neg s_b && (2.29)
\end{aligned}
$$

Damit alle Zellen mit der korrekten Anzahl an Linien umgeben sind, wie durch Ungleichung 2.13 sichergestellt, wird für die SAT-Formulierung nach 0-, 1-, 2- und 3-Zellen unterschieden. Bei 0-Zellen genügt es für jede Seite eine Klausel zu verwenden, um die Belegung dieser Seite auszuschließen.

$$
\begin{aligned}
\forall E' = \{a,b,c,d\} \text{ mit } h(E') = 0: \quad & \neg s_a && (2.30)\\
& \neg s_b && (2.31)\\
& \neg s_c && (2.32)\\
& \neg s_d && (2.33)
\end{aligned}
$$

Um die Disjunktion zu 2.34 für Zellen mit dem Zahlenwert 1 zu erfüllen, muss mindestens eine Linie ausgewählt werden. Damit auch gewährleistet wird, dass höchstens eine Linie gewählt wird, darf keine Kombination von zwei der vier Seiten gleichzeitig in der Lösung sein.

$$
\begin{aligned}
\forall E' = \{a,b,c,d\} \text{ mit } h(E') = 1: \quad & s_a \vee s_b \vee s_c \vee s_d && (2.34)\\
& \neg s_a \vee \neg s_b && (2.35)\\
& \neg s_a \vee \neg s_c && (2.36)\\
& \neg s_a \vee \neg s_d && (2.37)\\
& \neg s_b \vee \neg s_c && (2.38)\\
& \neg s_b \vee \neg s_d && (2.39)\\
& \neg s_c \vee \neg s_d && (2.40)
\end{aligned}
$$

Für 2-Zellen muss die Anzahl an Linien in der Lösung höchstens 2 sein, analog zur Übersetzung für Ungleichung 2.2 können hier Disjunktionen gebildet werden.

$$
\begin{aligned}
\forall E' = \{a,b,c,d\} \text{ mit } h(E') = 2: \quad & \neg s_a \vee \neg s_b \vee \neg s_c && (2.41)\\
& \neg s_a \vee \neg s_c \vee \neg s_d && (2.42)\\
& \neg s_a \vee \neg s_b \vee \neg s_d && (2.43)\\
& \neg s_b \vee \neg s_c \vee \neg s_d && (2.44)
\end{aligned}
$$

Auf der anderen Seite besteht auch die Bedingung, dass mindestens 2 aus den vier Linien a, b, c, und d gewählt werden müssen. Dies ist genau dann erfüllt, wenn eine mögliche Kombination aus 2 Seiten als wahr belegt wird. Als aussagenlogische Formel also:

$$(s_a \wedge s_b) \vee (s_a \wedge s_c) \vee (s_a \wedge s_d) \vee (s_b \wedge s_c) \vee (s_b \wedge s_d) \vee (s_c \wedge s_d)$$

Umgeformt in KNF ergibt sich dann:

$$\forall E' = \{a, b, c, d\} \text{ mit } h(E') = 2: \quad s_a \vee s_b \vee s_c \tag{2.45}$$
$$s_a \vee s_c \vee s_d \tag{2.46}$$
$$s_a \vee s_b \vee s_d \tag{2.47}$$
$$s_b \vee s_c \vee s_d \tag{2.48}$$

Die Formulierung für 3-Zellen entspricht der Formulierung für 1-Zellen, wobei die Vorzeichen der Literale invertiert werden. Durch die Disjunktion zu 2.49 wird erreicht, dass mindestens eine Linie nicht ausgewählt wird. Damit auch höchstens eine Linie gewählt wird, darf keine Kombination von zwei der vier Seiten gleichzeitig nicht in der Lösung enthalten sein.

$$\forall E' = \{a, b, c, d\} \text{ mit } h(E') = 3: \quad \neg s_a \vee \neg s_b \vee \neg s_c \vee \neg s_d \tag{2.49}$$
$$s_a \vee s_b \tag{2.50}$$
$$s_a \vee s_c \tag{2.51}$$
$$s_a \vee s_d \tag{2.52}$$
$$s_b \vee s_c \tag{2.53}$$
$$s_b \vee s_d \tag{2.54}$$
$$s_c \vee s_d \tag{2.55}$$

Die Konjunktion aller Disjunktionsterme von 2.15 bis 2.55 ergibt eine aussagenlogische Formel, welche durch einen SAT-Solver eine Lösung für Slitherlynx liefert. Für ein Slitherlink bzw. Slitherlynx Puzzle mit n Reihen, m Spalten und z_i Zahlen $i \in \{0, 1, 2, 3\}$ im Puzzle liefert diese SAT-Modellierung die folgende Anzahl an Klauseln:

$$1+2\cdot4+4(2(n-1)+2(m-1))+8(n-1)(m-1)+4z_0+7z_1+8z_2+7z_3 = 8nm+4z_0+7z_1+8z_2+7z_3+1$$

Um wie beim Integer Programmming eine Lösung für Slitherlink zu erhalten, werden der aussagenlogischen Formel in einem iterativen Prozess zusätzliche Klauseln hinzugefügt. Dieser Prozess wird in Abbildung 2.5 schematisch dargestellt. Die folgende Disjunktion wird genau dann nicht erfüllt, wenn alle Kanten eines bereits explorierten Kreises $C_i \in K$ mit in der Lösung enthalten sind:

$$\forall C_i \in K: \quad \bigvee_{e \in C_i} \neg s_e \tag{2.56}$$

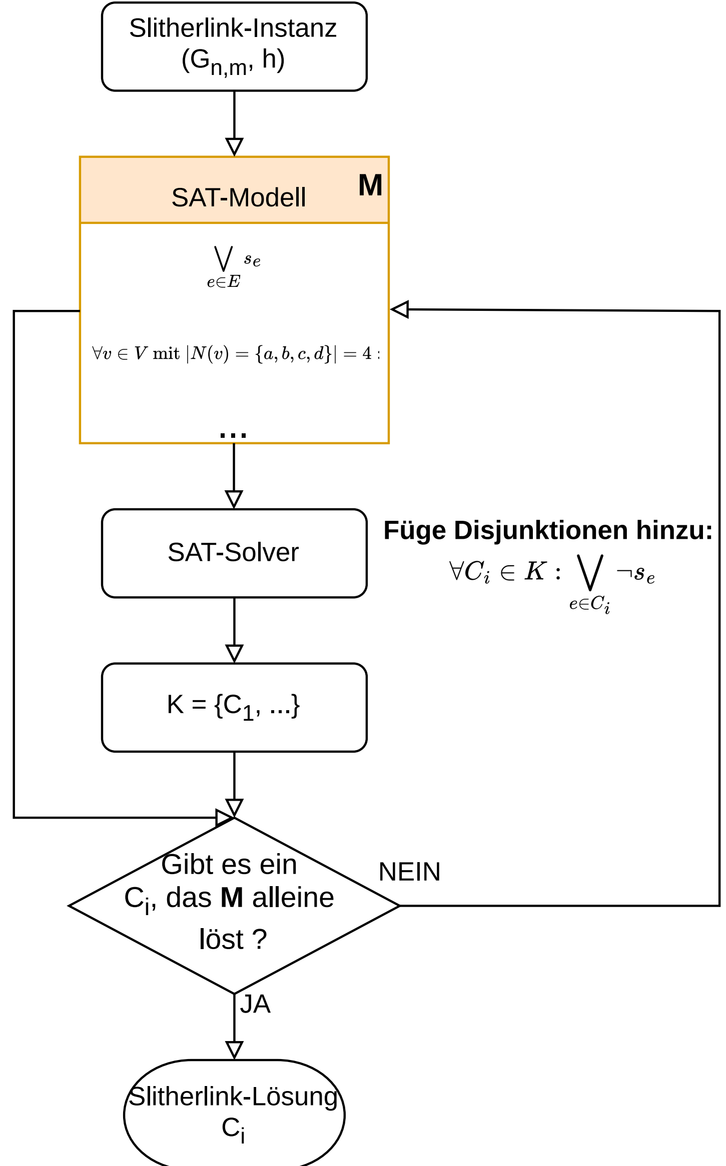


Abbildung 2.5: Iterativer Lösungsprozess für Slitherlink mittels SAT-Modellierung.

2.3 Zero-Suppressed Decision Diagrams

Ein Zero-Suppressed Binary Decision Diagram (ZDD) ist eine Spezialform von binären Entscheidungsdiagrammen. ZDDs eignen sich unter anderem besonders, um Mengenfamilien kompakt darzustellen und effizient modifizieren zu können, aber auch zur kompakten Repräsentation möglicher Pfade in ungerichteten Graphen. Daher sind sie eine sehr interessante Datenstruktuktur für das Lösen von Link Puzzles wie Slitherlink.

Definition 3: Zero-Suppressed Binary Decision Diagram (ZDD) (angepasst von [16])

Ein ZDD ist ein gerichteter azyklischer Graph, der mit Labels für die einzelnen Knoten und Kanten versehen ist. Es handelt sich um eine kompakte Repräsentation von Mengenfamilien über einer universellen, total geordneten Menge U (Universum). Dabei muss Folgendes gelten:

- Es gibt genau einen Wurzelknoten ohne eingehende Kanten.
- Es gibt genau zwei Endknoten ohne ausgehende Kanten (0-terminal und 1-terminal).
- Alle Nicht-Endknoten haben zwei ausgehende Kanten mit Label 0 und 1 (0-arc und 1-arc).
- Alle Nicht-Endknoten haben ein Label aus U.
- Das Label eines Elternknotens muss entsprechend der totalen Ordnung kleiner als das Label seiner Kindknoten sein.

Für jeden Knoten n sei das j-child der Knoten, der über das j-arc mit n verbunden ist. Der Elternknoten eines j-childs sei dessen j-parent. Yoshinaka et. al. [16] definieren die folgenden Eigenschaften für ZDDs:

Definition 4: Gültiger Pfad

Ein Pfad von der Wurzel eines ZDDs sei gültig, wenn er nicht in 0-terminal endet.

Definition 5: Reduziertes ZDD

Ein ZDD ist reduziert, wenn

- es keine zwei unterschiedlichen Knoten mit gleichem Label, 0-child und 1-child gibt und
- es keinen Knoten mit 0-terminal als 1-child gibt.

Definition 6: Repräsentation von Mengenfamilien

Für jeden gültigen Pfad π kann eine Menge $Z(\pi) \subseteq E$ der Form

$$Z(\pi) = \{u| \text{ 1-arc eines Knoten mit dem Label } u \text{ ist in } \pi\}$$

gebildet werden. Für jeden Knoten $n \neq$ 0-terminal kann eine Menge $\mathcal{Z}(n) \subseteq 2^U$ gebildet werden:

$$\mathcal{Z}(n) = \{Z(\pi)|\pi \text{ ist ein gültiger Pfad, der in } n \text{ endet}\}$$

$\mathcal{Z}$(1-terminal) entspricht der durch das ZDD repräsentierten Mengenfamilie.

Ein 0-arc in einem Pfad ausgehend von einem Knoten n im ZDD repräsentiert die Entscheidung "n ist nicht Teil der Menge", während ein 1-arc das Gegenteil "n ist Teil der Menge" bedeutet. In Abbildung 2.6 wird ein ZDD über dem Universum $U = \{A, B, C\}$ dargestellt. In diesem Beispiel gibt es fünf verschiedene gültige Pfade nach 1-terminal: $A \rightarrow_1 C \rightarrow_0 1$, $A \rightarrow_0 B \rightarrow_1 C \rightarrow_0 1$, $A \rightarrow_0 B \rightarrow_0 C \rightarrow_1 1$, $A \rightarrow_1 C \rightarrow_1 1$ und $A \rightarrow_0 B \rightarrow_1 C \rightarrow_1 1$. Das ZDD repräsentiert die folgende Mengenfamilie:

$$\mathcal{Z}(\text{1-terminal}) = \{\{A\}, \{B\}, \{C\}, \{A, C\}, \{B, C\}\}$$

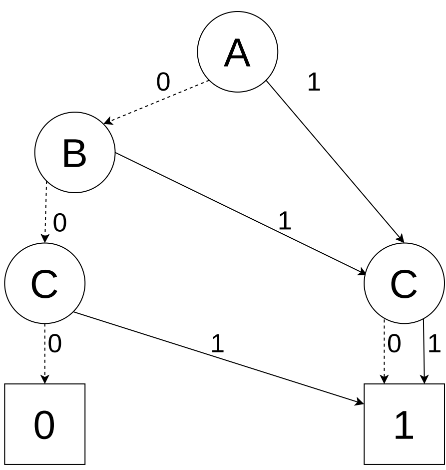


Abbildung 2.6: Einfaches ZDD. Gestrichelte Linien stehen für 0-arcs, durchgezogene Linien stehen für 1-arcs. Angepasst von [16].

Die Abbildung 2.7 zeigt die unreduzierte Version des ZDDs aus Abbildung 2.6. Der zweite und der dritte Knoten mit dem Label C haben identische Nachfolger und der zweite Knoten mit dem Label B endet mit 1-arc in 0-terminal. Jedes reduzierte ZDD repräsentiert dieselbe Mengenfamilie wie das entsprechende ZDD in seiner unreduzierten Form. Die beiden ZDDs stellen also die gleiche Mengenfamilie dar. Knuth [7] hat einen Algorithmus entworfen, mit dem sich jedes ZDD mit linearen Zeitaufwand reduzieren lässt.
Damit es im Weiteren nicht zu Verwechslungen zwischen ZDD-Graphen und Graphen von Slitherlink-Instanzen kommt, werden für diesen Abschnitt die Bezeichungen "Ecken" und "Arcs" für ZDDs sowie "Knoten" und "Kanten" für sonstige Graphen verwendet.

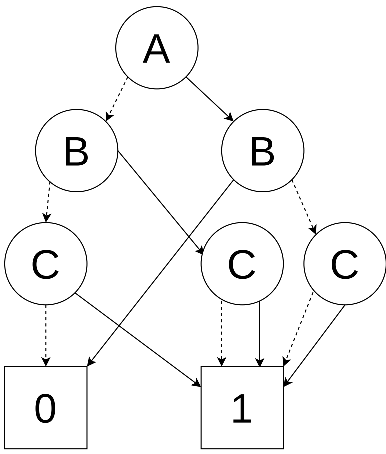


Abbildung 2.7: Unreduziertes ZDD.

2.3.1 ZDD-Operationen

Ein entscheidender Nutzen von ZDDs ist es, dass man mathematische Operationen auf Mengenfamilien wie Vereinigungsmenge oder Schnittmenge effizient berechnet werden können. Tabelle 2.3 zeigt Mengenoperatoren angewandt auf zwei Mengenfamilien $\mathcal{S}_1$ und $\mathcal{S}_2$.

Tabelle 2.3: Operationen zur Manipulation von Mengenfamilien nach [3].

Operation	Definition
Vereinigungsmenge	$\mathcal{S}_1 \cup \mathcal{S}_2 = \{S \mid S \in \mathcal{S}_1 \vee S \in \mathcal{S}_2\}$
Schnittmenge	$\mathcal{S}_1 \cap \mathcal{S}_2 = \{S \mid S \in \mathcal{S}_1 \wedge S \in \mathcal{S}_2\}$
Differenz	$\mathcal{S}_1 \setminus \mathcal{S}_2 = \{S \mid S \in \mathcal{S}_1 \wedge S \notin \mathcal{S}_2\}$
Teilmengen	$\mathcal{S}_1 \curvearrowleft \mathcal{S}_2 = \{S_1 \in \mathcal{S}_1 \mid \exists S_2 \in \mathcal{S}_2 (S_1 \subseteq S_2)\}$
Obermengen (Inklusion)	$\mathcal{S}_1 \curvearrowright \mathcal{S}_2 = \{S_1 \in \mathcal{S}_1 \mid \exists S_2 \in \mathcal{S}_2 (S_1 \supseteq S_2)\}$
Exklusion	$\mathcal{S}_1 \not\curvearrowright \mathcal{S}_2 = \mathcal{S}_1 \setminus (\mathcal{S}_1 \curvearrowright \mathcal{S}_2)$

Die ersten drei Operationen sind bekannt aus allgemeiner Mengenalgebra und lassen sich auf Mengenfamilien übertragen. Die Teilmengen Operation $\mathcal{S}_1 \curvearrowleft \mathcal{S}_2$ gibt alle Mengen aus $\mathcal{S}_1$ zurück, die vollständig in einer Menge aus $\mathcal{S}_2$ enthalten sind. $\mathcal{S}_1 \curvearrowright \mathcal{S}_2$ liefert eine Mengenfamilie, die nur Mengen aus $\mathcal{S}_1$ enthält, in denen eine Menge aus $\mathcal{S}_2$ vollständig inkludiert ist. Umgekehrt werden durch den $\not\curvearrowright$-Operator solche Mengen aus $\mathcal{S}_1$ exkludiert, die eine Menge aus $\mathcal{S}_2$ enthalten. Laut Minato [10] können Vereinigung, Schnitt und Differenz in der Praxis in linearer Zeit zur Größe des ZDDs berechnet werden. Ähnliche Operationen wie Teilmengen und Obermengen wurden von Knuth [7] entwickelt.

2.3.2 ZDDs zum Aufzählen von möglichen Pfaden oder Kreisen

Donald Knuth [7] hat mit SIMPATH einen Algorithmus entwickelt, welcher aus einem gegebenen ungerichteten Graphen $G = (V, E)$ und zwei Knoten s und t ein ZDD generiert, das alle Pfade zwischen diesen Knoten repäsentiert. Die Idee ist es, Pfade als Mengen von Kanten zu betracheten. Das bedeutet, in dem generierten ZDD werden die Ecken mit Kanten aus dem Eingabegraphen gelabelt.
Vor Anwendung des Algorithmus sollten die Knoten $1, \dots, n$ so durchnummeriert werden, dass kein Knoten außer der erste (s) vor allen seiner Nachbarn auftritt. Dies kann durch eine vollständige Breitensuche erreicht werden. Die Abbildung 2.8 zeigt exemplarisch eine mögliche Nummerierung der Knoten für $G_{3,3}$, wobei der Startknoten s in der obere linke Knoten ist. Durch die Nummerierung der Knoten gibt es eine totale Ordnung für Kanten gemäß lexikografischer Ordnung. Für das zu generierende ZDD wird die Kantenmenge E dann zum Universum $U = E$. Durch diese Sortierung der Kanten soll erreicht werden, dass man sich nicht zu viele Informationen von Anfangsknoten bis zum Ende merken muss.

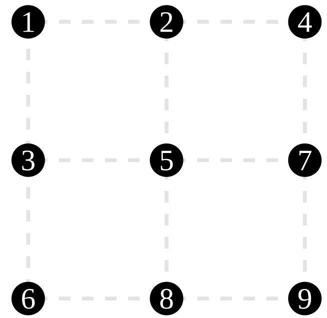


Abbildung 2.8: Gittergraph $G_{3,3}$ mit Nummerierung der Knoten.

Der Zustand einer Ecke wird in einer Tabelle $mate[1, \dots, n]$ gespeichert. Dabei gilt:

$$mate[i] = \begin{cases} i & \text{, falls } i \text{ noch nicht betrachtet wurde} \\ 0 & \text{, falls } i \text{ bereits zweimal betrachtet wurde} \\ r & \text{, falls bereits untersuchte Kanten einen Pfad mit } i \text{ und } r \text{ als Endpunkte bilden} \end{cases}$$

Anders formuliert: Falls i nicht im Teilpfad enthalten, ist $mate[i] = i$. Sollte i in der Mitte eines Teilpfades sein, dann ist $mate[i] = 0$. In einem Pfad gibt es zwei Endpunkte, die nur einen Nachbarn haben. Für den Fall, dass i ein Teil des Pfades sein sollte, aber nur einen Nachbarn im Pfad hat, ist $mate[i] = r$ mit r als zweiten Endpunkt des Pfades.
Initial werden für die Start- und Zielposition s, t $mate[1] = t$ und $mate[t] = 1$ gesetzt. Für alle anderen $i \in \{1, \dots, n\}$ wird $mate[i] = i$ gesetzt. Zudem werden Wurzelecke und die beiden Endecken generiert. Die noch zu untersuchenden Kanten werden in einer Queue gespeichert. Zu Beginn befindet sich nur die Wurzelecke, welche die erste Kante repräsentiert, in der Queue. Dann wird in einer Art Breitensuche die Queue durchiteriert. Es wird für jede Ecke des ZDDs 0-arc und 1-arc untersucht. Falls der Knoten nach 0-arc oder 1-arc in einem Endzustand wäre, wird dieser mit der entsprechenden Endecke verbunden, andernfalls wird eine neue Ecke für das ZDD erzeugt und in die Queue hinzugefügt. Die neue Ecke wird mit der nächsten Kante aus E entsprechend der Kantenordnung gelabelt. Für jede Ecke, die in die Queue hinzugefügt wird, wird zusätzlich der Zustand der $mate$-Tabelle aus Sicht der Ecke gespeichert.

Nach Wahl einer Kante $j - m$ (1-arc), ändert sich der Zustand, das heißt die $mate$-Tabelle für die neue Ecke wird aktualisiert (siehe Algorithmus 1).

Algorithmus 1 $mate$-Update - Aktualisieren der $mate$-Tabelle bei Wahl einer Kante $j - m$

```
1: ĵ ← mate[j]
2: m̂ ← mate[m]
3: if j ≠ ĵ then
4:     mate[j] ← 0
5: end if
6: if m ≠ m̂ then
7:     mate[m] ← 0
8: end if
9: mate[ĵ] ← m̂
10: mate[m̂] ← ĵ
```

Falls j zuvor unverbunden war, ist $\hat{j} = j$. Wenn j zuvor über einen Pfad mit mit einem Knoten k verbunden war, dann ist $\hat{j} = k$. Für diesen Fall wird durch die Aktualisierung eine zweite Kante mit dem Knoten j in die Lösung aufgenommen. Der Eintrag in der $mate$-Tabelle wird auf 0 gesetzt, um so zu markieren, dass der Knoten j nicht weiter betrachtet werden muss bzw. in der Mitte eines Teilpfades ist. Der vorherige Endpunkt für den Teilpfad mit dem Knoten m ist $\hat{m}$. Mit der Wahl von einer Kante mit m wird dieser Endpunkt $\hat{m}$ nun auch der Endpunkt für den vereinigten Teilpfad mit der Kante $j - m$ durch die Zuweisung $mate[\hat{j}] \leftarrow \hat{m}$.
Sei $j - m$ die aktuell zu untersuchende Kante mit $j < m$. Sei $j' - m'$ die in der nächsten Iteration betrachtete Kante und l' der bisherige Maximalwert für j_e für bereits untersuchte Kanten $\{j_1 - m_1, j_2 - m_2, \ldots, j_k - m_k\}$. Sei i der kleinste Integer mit $i > j$ und $i \neq m$. Um mögliche Endzustände zu erkennen wird wie folgt vorgegangen:
Es wird der Fall betrachtet, dass wir die Kante auswählen möchten (1-arc). Falls $mate[j] = 0$ oder $mate[m] = 0$ kann diese Kante nicht ausgewählt werden, d.h. 1-arc zu 0-terminal. Für den Fall, dass $mate[j] = m$ und $mate[m] = j$, haben wir einen Endzustand gefunden. Falls $i \leq l'$ kann die Kante nicht gewählt werden, d.h. 1-arc zu 0-terminal. Sonst haben wir eine $s - t$-Pfad gefunden, d.h 1-arc zu 1-terminal.
Falls ein i mit $j \leq i < j'$, $mate[i] \neq 0$ und $mate[i] \neq i$ existiert, ist die darauf folgende Ecke 0-terminal. Dies gilt sowohl für den Fall, dass die Kante ausgewählt werden soll (1-arc), als auch für den Fall, dass die Kante nicht ausgewählt werden soll (0-arc). Es gibt in diesem Fall einen offenen Endpunkt, d.h. ein Knoten wurde bereits über eine Kante in den Pfad aufgenommen, kann aber mit den noch zu betrachtenden Knoten nicht mehr erreicht werden.
Kholilurrohman und Minato [5] haben eine Pseudocode-Formulierung für SIMPATH entwickelt, welche in Algorithmus 2 in angepasster Form beschrieben wird.
Das Vorgehen des SIMPATH-Algorithmus soll im Folgenden anhand des Gittergraphen $G_{3,3}$ aus Abbildung 2.8 für alle Pfade von 1 nach 9 veranschaulicht werden. Die schrittweise Erzeugung des ZDDs wird in Abbildung 2.9 für die ersten Schritte dargestellt. Die Grafiken neben den ZDD-Ecken visualisieren dabei den durch die Ecke repräsentierten Zustand. Gestrichelte Linien kennzeichnen noch offene Kanten, durchgezogene Linien markieren die Auswahl einer Kante, und keine Linie bedeutet, dass eine Kante nicht ausgewählt wurde.

Algorithmus 2 SIMPATH

```
1: for i = 1 to n do
2:     mate[i] ← i
3: end for
4: mate[s = 1] ← t, mate[t] ← 1
5: create ZDD-root labeled with first edge
6: create 0-terminal, 1-terminal
7: Q ← new Queue
8: Q.enqueue(root)
9: while Q is not empty do
10:     node ← Q.dequeue
11:     for 0-arc/1-arc do
12:         if state following 0-arc/1-arc is not terminal then
13:             create childnode
14:             add 0-arc/1-arc from node to childnode
15:             Q.enqueue(childnode)
16:         else
17:             add 0-arc/1-arc from node to 0-terminal/1-terminal
18:         end if
19:     end for
20: end while
21: reduce ZDD
```

Im Schritt ① wird die Wurzel mit dem Label der ersten Kante $1-2$ generiert und in die Queue hinzugefügt. Sei $mate'$ der Zustand der $mate$-Tabelle nach Auswahl einer Kante (1-arc). Wir simulieren die ersten Iterationen der Queue:

1. Kante $j - m = 1 - 2$: Keiner der beiden Wege führt zu einem Endzustand. Es wird jeweils eine neue Ecke im ZDD für die Kante $1-3$ erstellt (Schritte ② und ③).

j	m	$mate[1,\ldots,9]$	$mate'[1,\ldots,9]$
1	2	9 2 3 4 5 6 7 8 1	0 9 3 4 5 6 7 8 2

2. Kante $j - m = 1 - 3$, nachdem $1-2$ im Schritt ② nicht gewählt wurde: Die nächste zu untersuchende Kante gemäß Kantenordnung wäre $2-4$, d.h. $j' = 2$. Für $i = 1$ gilt $j \leq i < j'$. Es gilt $mate[1] = 9$, also $mate[1] \neq 0$ und $mate[1] \neq 1$. Somit führt im Schritt ④ der 0-arc zur 0-terminal Endecke. In diesem Fall wurde erkannt, dass es keinen Pfad von 1 nach 9 geben kann, welcher weder die Kante $1-2$ noch die Kante $1-3$ enthält. Der 1-arc führt im Schritt ⑤ zu keinem Endzustand.

j	m	$mate[1,\ldots,9]$	$mate'[1,\ldots,9]$
1	3	9 2 3 4 5 6 7 8 1	0 2 9 4 5 6 7 8 3

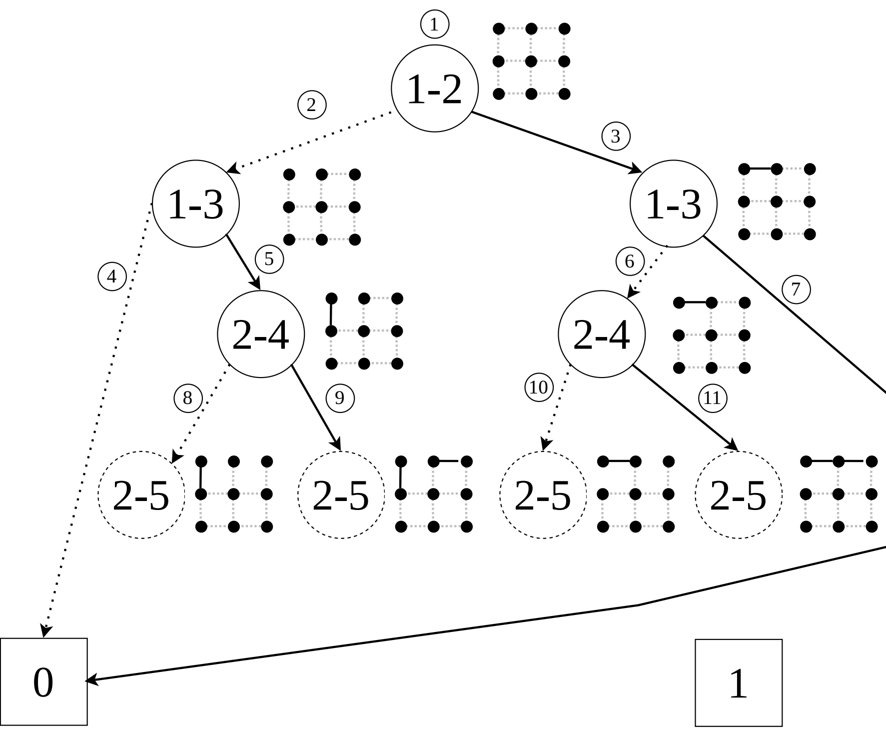


Abbildung 2.9: Simulation der ersten Schritte des SIMPATH-Algorithmus zur ZDD-Generierung anhand des Graphen $G_{3,3}$.

3. Kante $j - m = 1 - 3$, nachdem $1 - 2$ im Schritt ③ gewählt wurde: Der 0-arc (Schritt ⑥) führt zu keinem Endzustand. Wohingegen der 1-arc (Schritt ⑦) zu einem Endzustand führt, da $mate[j = 1] = 0$. Es kann keinen Pfad von 1 nach 9 geben, der sowohl die Kante $1 - 2$ als auch die Kante $1 - 3$ enthält.

j	m	$mate[1, \ldots, 9]$	$mate'[1, \ldots, 9]$
1	3	0 9 3 4 5 6 7 8 2	-

4. Kante $j - m = 2 - 4$, nachdem $1 - 3$ im Schritt ⑤ gewählt wurde: Keiner der beiden Wege führt zu einem Endzustand.

j	m	$mate[1, \ldots, 9]$	$mate'[1, \ldots, 9]$
2	4	0 2 9 4 5 6 7 8 3	0 4 9 2 5 6 7 8 3

5. Kante $j - m = 2 - 4$, nachdem $1 - 2$ gewählt wurde und $1 - 3$ im Schritt ⑥ nicht gewählt wurde: Keiner der beiden Wege führt zu einem Endzustand.

j	m	$mate[1, \ldots, 9]$	$mate'[1, \ldots, 9]$
2	4	0 9 3 4 5 6 7 8 2	0 0 3 9 5 6 7 8 4

6. ...

Nachdem alle Knoten in der Queue untersucht worden sind, kann das ZDD reduziert werden und es ergibt sich laut Knuth ein ZDD wie in Abbildung 2.10. Es gibt 12 verschiedene Pfade von 1 nach 9 im Graphen aus Abbildung 2.8. Das ZDD aus Abbildung 2.10 scheint kein optimaler Ansatz zu sein, um alle Pfade im 3×3-Gitter darzustellen. Bei größeren Graphen wird der Vorteil ersichtlich. Für einen 8×8-Gittergraph gibt es laut Knuth 789.360.053.252 Pfade von der obersten linken in die unterste rechte Ecke, welche durch ein ZDD mit gerade einmal 33580 Knoten dargestellt werden kann.

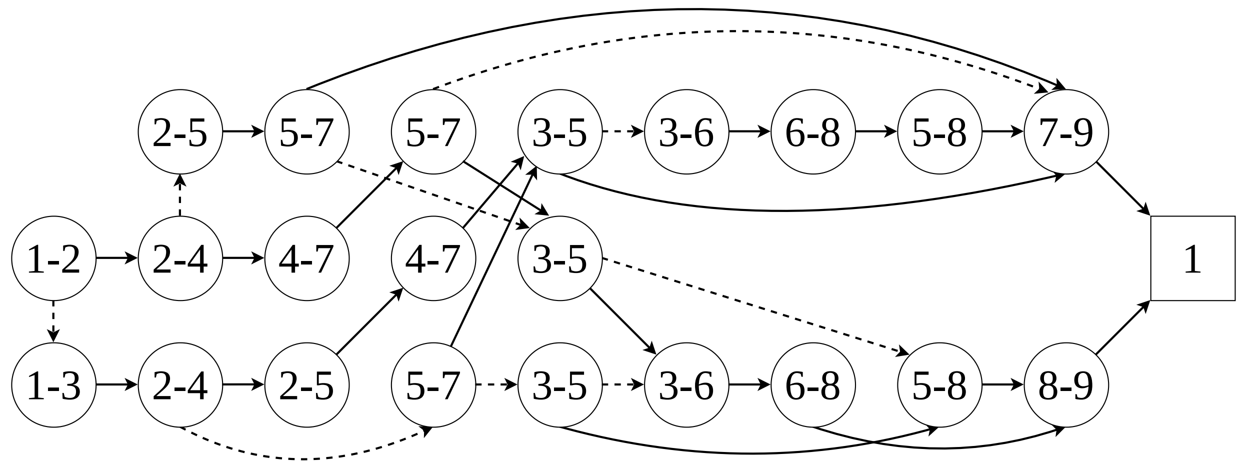


Abbildung 2.10: Reduziertes ZDD für alle Pfade eines Graphen $G_{3,3}$. Angepasst von [7].

Im Fall von Slitherlink sind jedoch nicht $s - t$-Pfade von Interesse, sondern Kreise ohne vorgegebenen Start- und Endpunkt. Laut Knuth lässt sich dies dadurch erreichen, dass bei der Initialisierung der *mate*-Tabelle die Zuweisungen $mate[1] \leftarrow t$ und $mate[t] \leftarrow 1$ ausgelassen werden (Z. 4 in Algorithmus 2).

2.3.3 Lösen von Slitherlink mithilfe von ZDDs

Yoshinaka et. al. [16] haben in 2012 eine Methode vorgestellt, Slitherlink Puzzles mithilfe von ZDDs zu lösen. Dabei wird in einem ähnlichen Vorgehen wie bei SIMPATH ein ZDD generiert, welches alle möglichen Lösungen für eine gegebene Slitherlink-Instanz repräsentiert. In 2013 hat ein Teil der Autoren mit Graphillion eine ZDD-basierte Python Bibliothek vorgestellt [3].
Diese Bibliothek ermöglicht es, Operationen wie in Tabelle 2.3 auf großen Mengen von Graphen (GraphSets) effizient auszuführen. Graphillion ist als Python-Modul mit C++-Erweiterungen implementiert. Ein GraphSet wird in komprimierter Form als C++-Objekt dargestellt. Python hat lediglich die Referenz zu diesem Objekt. Die Autoren konnten so einen Slitherlink-Solver mit Graphillion entwickeln, welcher einfacher zu verstehen und mit einer Reduktion der Anzahl an Codezeilen um 93% weniger komplex sei. Laut Untersuchungen sei die Performance vergleichbar mit dem vorherigen dedizierten Slitherlink-Solver, wobei der dedizierte Solver schnellere Ausführungszeiten hatte.
Der Solver iteriert zunächst über alle angegebenen Zahlen-Hinweise eines Slitherlink Puzzles und erstellt Untergraphen, um die Bedingungen, dass 0, 1, 2 oder 3 Kanten eine Zelle umgeben, zu erfüllen. Dabei werden alle Graphen aufgezählt, welche potenziell ein Teil der Lösung sein können, und

mit der $\curvearrowright$-Operationen inkludiert. Teilgraphen, welche definitiv nicht Teil der Lösung sein dürfen werden, werden mit dem $\not\curvearrowright$-Operator ausgeschlossen.
In Tabelle 2.4 werden für die verschiedenen möglichen Zahlen in einem Slitherlink Puzzle die Teilgraphen abgebildet, welche inkludiert oder exkludiert werden. Zuletzt werden mit einer vordefinierten Methode `cycles` alle möglichen Kreise in den verbliebenen Graphstrukturen aufgezählt.

Tabelle 2.4: Potentiell mögliche und verbotene Teilgraphen bei vorgegebenen Zahlen im Puzzle.

	Inkludiere	Exkludiere
0		0 0 0 0
1	1 1 1 1	1 1 1 1 1 1
2	2 2 2 2 2 2	2 2 2 2
3	3 3 3 3	3

Eine formale Beschreibung in Mengenschreibweise für das Vorgehen zum Finden einer Lösung für eine Slitherlink-Instanz $(G_{n,m} = (V, E), h)$ soll durch den Algorithmus 3 gegeben werden.

Algorithmus 3 Lösen von Slitherlink-Instanzen anhand von Mengenfamilien

1: $\mathcal{G} \leftarrow 2^E$
2: **for each** $E' \in \text{Dom}(h)$ **do**
3: $\quad \mathcal{G} \leftarrow \mathcal{G} \curvearrowright \{E'' | E'' \subseteq E' \wedge |E''| = h(E')\}$
4: $\quad \mathcal{G} \leftarrow \mathcal{G} \not\curvearrowright \{E'' | E'' \subseteq E' \wedge |E''| = h(E') + 1\}$
5: **end for**
6: $\mathcal{G}$`.cycles()`

Abbildung 2.11 veranschaulicht das Vorgehen an einem simplen Beispiel. Es werden ausgehend von der Potenzmenge 2^E alle Teilgraphen mit drei Kanten um “3” inkludiert. Das heißt es werden nur noch Teilgraphen akzeptiert, welche einen der vier angegebenen Graphen enthalten. Dann wird der Graph mit vier Kanten um “3” exkludiert, so dass diese Graphstruktur in keinem Teilgraphen enthalten sein kann. Die Zelle “1” wird in einer ähnlichen Weise bearbeitet. Das resultierende GraphSet hat dann

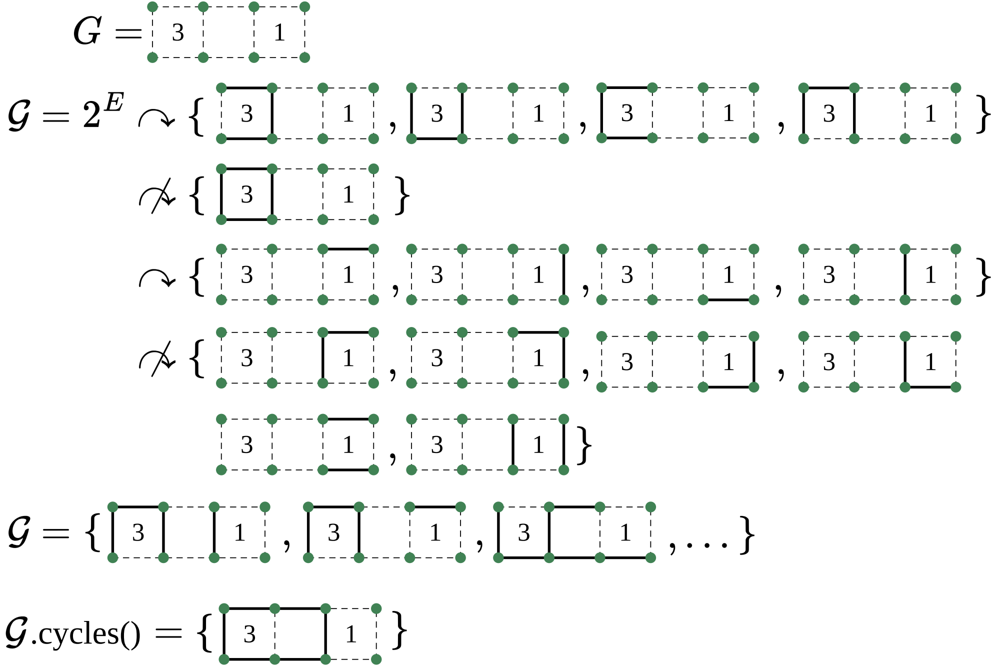


Abbildung 2.11: Generierung von Slitherlink-Lösungen mit Graphillion anhand einer Beispielinstanz. Angelehnt an [3].

nur noch Graphen, welche alle durch Zahlen im Puzzle gegebenen Einschränkungen (drei Kanten um "3" und eine Kante um "1") erfüllen. Die Methode `cycles` liefert dann gültige Slitherlink-Lösungen.

2.4 Verwendung eines Presolvers

Beim Lösen von Slitherlink-Puzzles beginnt ein menschlicher Spieler typischerweise damit, kleinere eindeutige Teillösungen zu finden und diese zu einem geschlossenen Kreis zu verbinden. Der in diesem Abschnitt untersuchte Ansatz zielt darauf ab, anhand bekannter Regeln, wie z.B. in [14], bereits vor dem eigentlichen Lösungsverfahren Kanten zu identifizieren, die sicher Teil der Lösung sein müssen oder nicht Teil der Lösung sein dürfen.
In verschiedenen Slitherlink-Puzzles lassen sich häufig ähnliche Muster erkennen, die Rückschlüsse auf die Lösung zulassen. Anhand solcher spielspezifischer Informationen lässt sich der Suchraum zum Teil deutlich reduzieren.

2.4.1 Färben von Slitherlink-Zellen

Eine Slitherlink-Lösung ist genau ein geschlossener Zyklus. Dieser Zyklus unterteilt die einzelnen Zellen in zwei verschiedene Bereiche: einen inneren und einen äußeren Bereich. Eine bekannte Lösungsstrategie ist das Färben der Zellen in einen Außen- und Innenbereich (siehe z.B. [12]). Wenn man zwei benachbarte innere Zellen oder benachbarte äußerere Zellen hat, heißt das, dass keine Linie

zwischen beiden Zellen gezogen wird. Umgekehrt müssen Zellen, zwischen denen keine Linie gezogen werden darf, gleich gefärbt sein. Analog gilt auch, dass genau dann eine äußere und eine innere Zelle benachbart sind, wenn zwischen diesen Zellen eine Linie gezogen werden muss.
Um den Zustand eines teilweise gelösten Puzzles effizient zu repräsentieren, wird für den Presolver lediglich die aktuelle Färbung jeder Zelle gespeichert. Es ist nicht notwendig den Zustand jeder Kante eines Eingabegraphen $G_{n,m}$ zu speichern (insgesamt $n(m-1)+m(n-1)$ Kanten), stattdessen genügt es $n \cdot m$ Zellen zu speichern.
Die Abbildung 2.12 zeigt die initiale Färbung für den Presolver. Zu Beginn werden alle Zellen mit unterschiedlichen Farben markiert. Zudem werden Hilfszellen verwendet, welche sich an den Rändern des eigentlichen Eingabegraphen befinden. Diese müssen außerhalb des Zyklus sein und werden als äußere Zellen markiert (dunkelgrau).

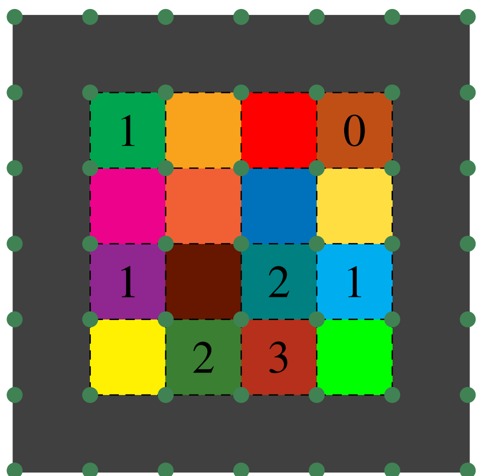


Abbildung 2.12: Initiale Färbung für das Slitherlink-Puzzle aus Abbildung 1.1.

2.4.2 Lokale Reduktionsregeln

Die grundlegende Idee des Presolver ist es, über alle Zellen des Puzzles zu iterieren und diese auf bekannte lokale Muster zu überprüfen. Sollte ein Muster erkannt werden, werden die Zellen entsprechend vordefinierter Regeln neu eingefärbt. Solange sich durch diese Iteration die Anzahl der unbekannten Farben im Puzzle verringert, wird der Vorgang wiederholt. Einige ausgewählte Reduktionsregeln sollen im Folgenden vorgestellt werden.
Die Abbildungen zeigen jeweils eine Ausgangssituation auf der linken Seite und den aus der Regel abgeleiteten Folgezustand auf der rechten Seite. Gestrichelte Linien kennzeichnen noch offene Kanten, durchgezogene Linien markieren die Auswahl einer Kante, und keine Linie bedeutet, dass eine Kante ausgeschlossen werden kann. Zellen mit einer helleren und dunkleren Farbe des selben Farbtons befinden sich auf gegenüberliegenden Seiten des geschlossenen Zyklus.

Linien stimmen mit der Zellennummer überein:

Für den Fall, dass die Anzahl der Linien mit der Nummer in einer Zelle übereinstimmt, können alle anliegenden noch nicht gewählten Kanten ausgeschlossen werden. Dies ist bei Zellen mit “0” wie in Abbildung 2.13 besonders gut zu erkennen. Wenn umgekehrt die Anzahl ausgeschlossener Kanten

korrekt ist, müssen alle übrigen Kanten in der Lösung sein. Die Abbildung 2.14 zeigt, dass wenn wir wissen, dass eine Kante an einer 3-Zelle ausgeschlossen werden kann, alle anderen Kanten in der Lösung sein müssen.

Abbildung 2.13: Regel für 0-Zellen. Angepasst von [12].

Abbildung 2.14: Regel für 3-Zellen mit benachbarter Zelle der selben Farbe. Angepasst von [12].

Null oder zwei angrenzende Linien an jedem Knotenpunkt:

Die Erkenntnis, dass jeder Knoten nur Null oder zwei angrenzende Linien aus der Lösung hat, führt zu vielen verschiedenen Reduktionsstrategien, wie z.B. die Folgenden.

- Falls bei einer Zelle mit “3” beide an einem Eckpunkt angrenzenden Kanten ausgeschlossen sind, müssen die beiden Linien der Zelle, welche an der Ecke angrenzen, gewählt werden. Andernfalls wäre eines der beiden Enden der Teillösung an jenem Eckpunkt und der Weg könnte nicht weitergeführt werden. Die blauen Linien in Abbildung 2.15 zeigen die einzigen beiden Möglichkeiten, die Linien um die “3” zu ziehen. Eine solche Situation entsteht zum Beispiel wenn eine “0” diagonal an die “3” angrenzt.

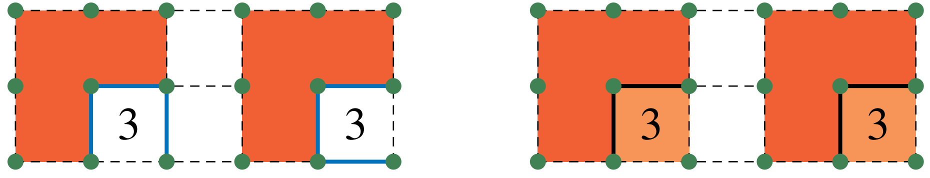
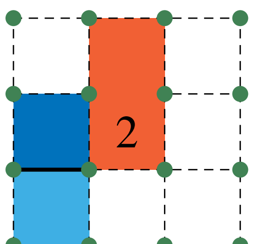


Abbildung 2.15: Regel für 3-Zellen in einer Ecke. Angepasst von [14].

- Falls bei einer 2-Zelle eine beliebige umgebende Linie ausgeschlossen ist, kann eine Linie, welche an eine der beiden nicht an der ausgeschlossenen Linie angrenzenden Ecken kommt, nicht

sofort im rechten Winkel von der 2-Zelle weggehen. Andernfalls wäre es nicht mehr möglich zwei Linien um die 2-Zelle zu ziehen. Die Linie im rechten Winkel kann also ausgeschlossen werden. Daraus folgt, dass die ankommende Linie in genau eine der anderen beiden Richtungen weiter gezogen werden muss und in die andere Richtung ausgeschlossen werden muss. Damit die Zelle insgesamt noch von zwei Linien umgeben werden kann, muss die zweite Linie für die Zelle auf der verbliebenen freien Seite liegen. Die Abbildung 2.16 zeigt eine Situation, in der die Regel angewendet werden kann. Die horizontale Linie, welche im Ausgangszustand vorhanden ist, darf am rechten Ende nicht nach unten weitergezogen werden, denn sonst wären drei Seiten der 2-Zelle ausgeschlossen.



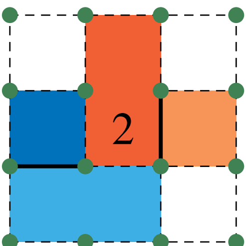
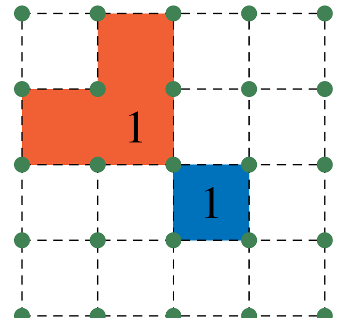


Abbildung 2.16: Regel für 2-Zellen neben einer Linie. Angepasst von [14].

- Falls zwei 1-Zellen diagonal benachbart sind, haben diese einen gemeinsamen Eckpunkt in der Mitte. Wenn für eine der beiden Zellen beide Kanten, die nicht an dem gemeinsamen Eckpunkt liegen, ausgeschlossen werden können, muss eine der beiden anderen umliegenden Kanten der Zelle gewählt werden. Somit muss der gemeinsame Eckpunkt auf dem Lösungspfad liegen. Da der Pfad am gemeinsamen Punkt weitergeführt werden muss, kann für die andere 1-Zelle auch nur eine der beiden Kanten, die am gemeinsamen Eckpunkt liegen, gewählt werden. Die anderen beiden Kanten der Zelle können also ausgeschlossen werden. Die Abbildung 2.17 zeigt eine Situation, in der man die Regel anwenden kann. Der gemeinsame Eckpunkt ist dort in der Mitte des Puzzles.



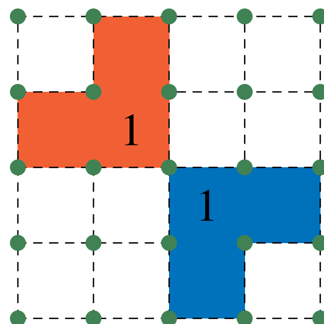
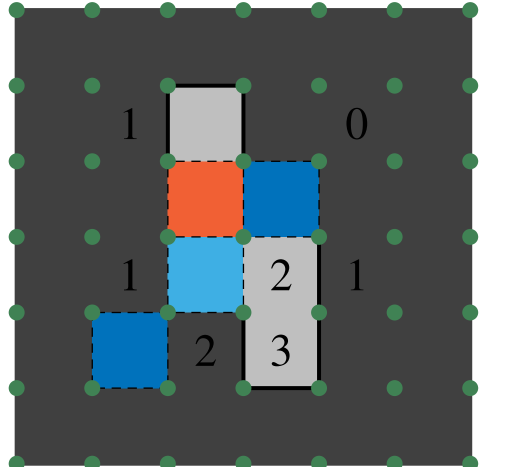


Abbildung 2.17: Regel für diagonal benachbarte 1-Zellen. Angepasst von [14].

2.4.3 Abgeschlossene Regionen

Zusätzlich zu den lokal überprüften Regeln untersucht der entwickelte Presolver das Puzzle auch auf Basis globaler Informationen. Wie bereits im Abschnitt 2.4.1 beschrieben, darf es genau einen

abgeschlossene und zusammenhängende Innenregion geben. Für Regionen, deren Farbe noch nicht endgültig festgelegt ist, prüft der Presolver, ob die Markierung der Region als Innen- oder Außenregion zu einer abgeschlossenen Region führt, die nicht mehr mit Zellen der gleichen Farbe verbunden werden kann und somit zu einem Widerspruch führt. Findet sich auf diesem Weg ein Widerspruch, weiß man, dass diese Färbung nicht erlaubt ist und somit die gegenteilige Färbung vorgenommen werden muss.
Die Überprüfung, ob, nachdem ein Bereich als Innen- oder Außenbereich gefärbt wurde, eine abgeschlossene Region entsteht, erfolgt mittels Breitensuche. Die Breitensuche beginnt jeweils an einem Startpunkt innerhalb der zu überprüfenden Region. Von diesem Punkt aus werden alle benachbarten Zellen in die Queue eingefügt, mit Ausnahme der Zellen, die mit der entgegengesetzten Farbe der zu prüfenden Region eingefärbt sind. Während der Breitensuche wird die Anzahl der Zellen mit der Farbe der zu überprüfenden Region gezählt und mit der tatsächlichen Anzahl solcher Zellen im Puzzle verglichen. Stimmen diese beiden Zahlen nicht überein, wurde eine abgeschlossene Region gefunden. Ein Beispiel für eine abgeschlossene Region liefert die Abbildung 2.18. Färbt man die orangefarbene Zelle zum Außenbereich (dunkelgrau) um, so entsteht eine abgeschlossene Region (hellgrau). Diese kann nicht mehr mit den anderen Zellen der gleichen Farbe verbunden werden. Daher muss die orange Zelle zum Innenbereich gehören.



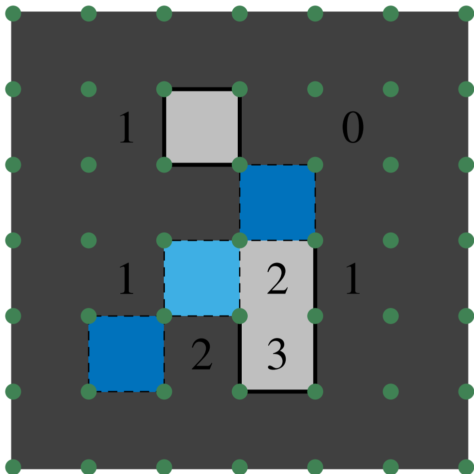


Abbildung 2.18: Erkennen von abgeschlossenen Regionen im teilweise gelösten Puzzle aus Abbildung 2.12.

2.4.4 Laufzeitabschätzungen

Die zuvor beschriebenen exakten Lösungsverfahren haben jeweils einen exponentiellen Zeitaufwand im Bezug auf die Puzzlegröße, liefern dafür jedoch in der Theorie garantiert eine korrekte Lösung. Die für den Presolver verwendeten Regeln liefern hingegen nicht immer ein vollständig gelöstes Puzzle, können jedoch in Polynomzeit angewendet werden.
Für einen Eingabegraphen $G_{n,m}$ gibt es insgesamt $N = n \cdot m$ Zellen. Zur Überprüfung einer lokalen Regel wird im Worst Case über alle N Zellen iteriert. Eine Zelle kann in konstanter Zeit auf Muster wie aus den Abbildungen 2.13-2.17 überprüft werden. Wurde ein Muster entdeckt erhält eine konstante Anzahl an Regionen eine neue Farbe. Das Umfärben einer Region kann in $O(N)$ geschehen,

z.B. durch das Überprüfen aller Zellen auf die neu einzufärbende Farbe und Neuzuweisung einer Farbe für eine gefundene Zelle. Die lokalen Regeln lassen sich so insgesamt in $O(N) + O(N) = O(N)$ anwenden.
Die Breitensuche auf dem Gitter an Zellen für die Überprüfung auf abgeschlossene Regionen kann mit einer Laufzeit von $O(N^2)$ nach oben abgeschätzt werden. Auch hier wird im Worst Case über alle N Zellen iteriert. Mit einem Aufwand von $O(N)$ im Fall einer Neufärbung ergibt sich so insgesamt eine Laufzeit von $O(N^3) + O(N) = O(N^3)$.

3 Ergebnisse

Die im Kapitel 2 vorgestellten Methoden wurden in Experimenten auf einem Computer mit einem Intel Core i5-4210U @ 1.7 - 2.7 GHz Prozessor und 16GB RAM unter Ubuntu 22.04.4 LTS getestet. Die Programme wurden mit Python 3.10.12 implementiert[1].
Als Python Schnittstelle für LP-Solver wurde die Bibliothek PuLP verwendet. Diese ermöglicht es linerare Optimierungsprobleme zu modellieren. Ein großer Vorteil von PuLP ist die Flexibilität bei der Wahl des Solvers (z.B. CPLEX, Gurobi). Die SAT-Modellierung wurde mithilfe der Python Bibliothek PySAT getestet, die eine einheitliche Schnittstelle zu verschiedenen modernen SAT-Solvern bietet. Der hier untersuchte ZDD-Solver wurde wie in Abschnitt 2.3.3 erwähnt mit Graphillion implementiert.
Python dient somit hauptsächlich als Schnittstelle für Programme, die in kompilierten, hardwarenahen Sprachen wie C (für CPLEX) und C++ (für Graphillion) entwickelt wurden. Der Presolver wurde ebenfalls als Python-Modul ausgelagert und in der Programmiersprache Rust implemeniert, die eine ähnliche Performance wie C++ bietet.
Als Datengrundlage für die Untersuchungen wurden Slitherlink Puzzles von janko.at[2] verwendet. Es wurden Beispielinstanzen unterschiedlicher Größen und Komplexität mit einem Zeitlimit von 60 Sekunden verglichen. Dabei wurden 10 Puzzles der Größe (5, 5), 393 Puzzles der Größe (10, 10), 72 Puzzles der Größe (12, 16) und 58 Puzzles der Größe (20, 36) berücksichtigt. Die Größe bezieht sich hier auf die Anzahl der Zeilen und Spalten an Zellen im Slitherlink Puzzle.

3.1 Vergleich von IP-Solvern

Die Instanzen wurden jeweils für den vorinstallierten PuLP-Solver, CPLEX und Gurobi getestet. In Abbildung 3.1 lässt sich erkennen, dass CPLEX und Gurobi eine korrekte Lösung schneller finden als der PuLP eigene Solver. Die kommerziellen Solver laufen vor allem bei den kleineren Puzzles ähnlich gut. Für Puzzles der Größe (20, 36) ist Gurobi im Durchschnitt etwa 0,38 Sekunden schneller als CPLEX. Außerdem konnte nur Gurobi alle Puzzles im gegebenen Zeitlimit korrekt lösen, wie in Tabelle 3.1 dargestellt. CPLEX lieferte für 7 Instanzen eine falsche Lösung und der PuLP-Solver brachte nur in 78.24% der Fälle das gewünschte Ergebnis innerhalb von 60 Sekunden.

Tabelle 3.1: Vergleich der Korrektheit für verschiedene IP-Solver für alle untersuchten Puzzles.

	PuLP	**CPLEX**	**Gurobi**
Anzahl ungültiger Lösungen	65	7	0
Anzahl an Timeouts	51	0	0
Anteil korrekt gelöster Puzzles	417/533 = 78.24%	526/533 = 98.69%	533/533 = 100.00%

[1] Code Repository: https://git.tu-berlin.de/berscjak/slitherlink-ba
[2] https://www.janko.at/Raetsel/Slitherlink/index.htm

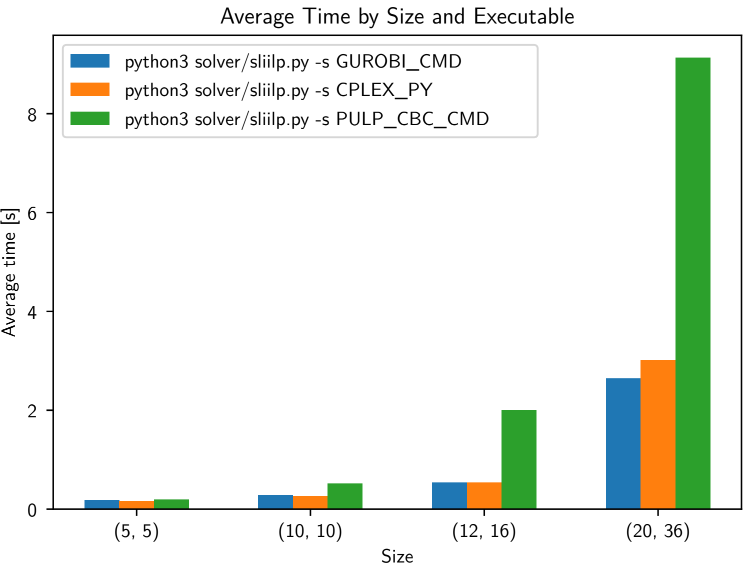


Abbildung 3.1: Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für verschiedene IP-Solvern.

3.2 Vergleich von SAT-Solvern

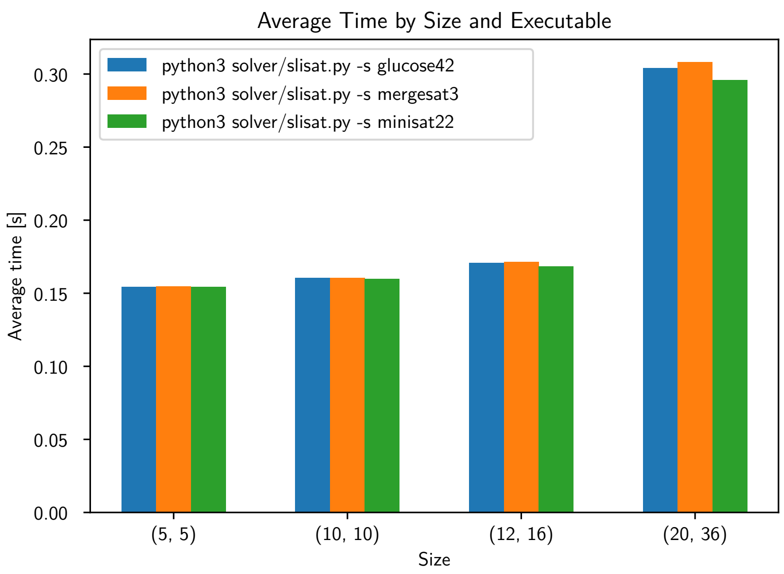


Abbildung 3.2: Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für verschiedene SAT-Solvern.

Beim Vergleich verschiedener PySAT-kompatibler Solver lassen sich in Abbildung 3.2 keine signifikanten Unterschiede zwischen den getesteten Solvern feststellen. Die Unterschiede für die verschiedenen Puzzlegrößen sind auch gering. Die Durchschnittszeiten bei den Größen (5, 5) und (12, 16) sind

auf einem ähnlichen Niveau. Für das Experiment wurden die SAT-Solver Glucose (4.2.1)[3], Mergesat (3.0)[4] und Minisat (2.2)[5] getestet. Bei den untersuchten Instanzen erzielte Minisat im Durchschnitt leicht kürzere Laufzeiten. Im Gegensatz zu den IP-Solvern gab es keine Timeouts oder ungültige Lösungen.

3.3 Vergleich von IP, SAT und ZDD

Das Balkendiagramm in Abbildung 3.3 stellt die Durchschnittszeiten für SAT (mit Minisat), IP (mit Gurobi) und den ZDD-Ansatz gegenüber. Für kleine Puzzles der Größe (5, 5) war der ZDD-Solver am schnellsten. Für die übrigen Größen war der SAT-Solver stets am besten. Bei der Größe (10, 10) waren IP und ZDD ähnlich gut. Für die Größe (12, 16) war der ZDD-Ansatz mit durchschnittlich ca. 8,22 Sekunden deutlich langsamer als die anderen Methoden. Innerhalb des Zeitlimits von 60 Sekunden konnte der ZDD-Ansatz für kein untersuchtes Puzzle der Größe (20, 36) eine Lösung finden.

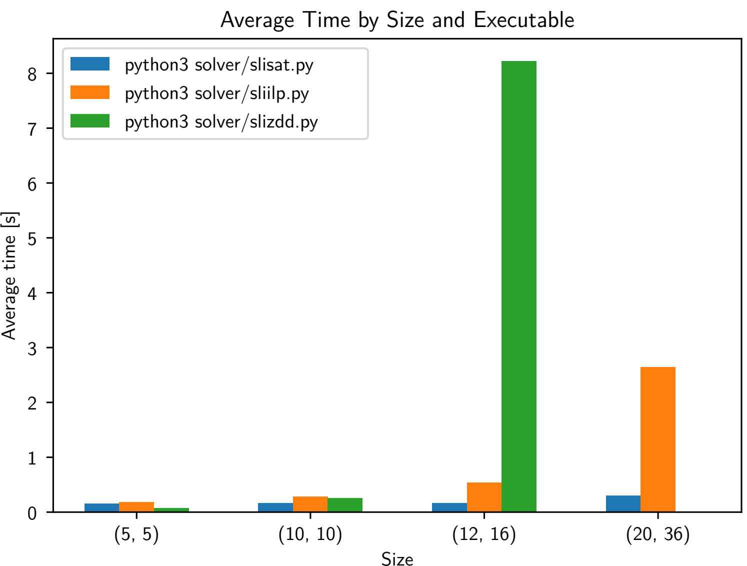


Abbildung 3.3: Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für IP, SAT und ZDD.

Betrachtet man die Verteilung der Zeiten in den Abbildungen 3.4 und 3.5, lässt sich erkennen, dass die Messdaten für den ZDD-Solver am meisten gestreut sind. Für Puzzles der Größe (10, 10) war der SAT-Solver zwar im Durchschnitt etwa 37.92% schneller als der ZDD-Ansatz, jedoch liegt der Medianwert auf einem ähnlichen Niveau. Abhängig vom untersuchten Puzzle hatte der ZDD-Ansatz deutliche Ausreißer nach oben (Maximalwert ca. 2,559 Sekunden) und hat zudem das kleinste Minimum mit ca. 0,075 Sekunden. Im Gegensatz dazu hatte der SAT-Solver einen Maximalwert von ca. 2,232 Sekunden und eine Minimalwert von ca. 0,146 Sekunden. Bei Puzzles der Größe (12, 16) wird die Streuung der ZDD Zeiten noch deutlicher. Während die Messwerte für den ZDD-Solver im

[3] https://pysathq.github.io/docs/html/api/solvers.html
[4] https://github.com/conp-solutions/mergesat
[5] http://minisat.se/MiniSat.html

Bereich zwischen ca. 35,408 Sekunden und ca. 0,615 Sekunden waren, waren die Zeiten für IP und SAT deutlich konzentrierter (s. Abbildung 3.5).

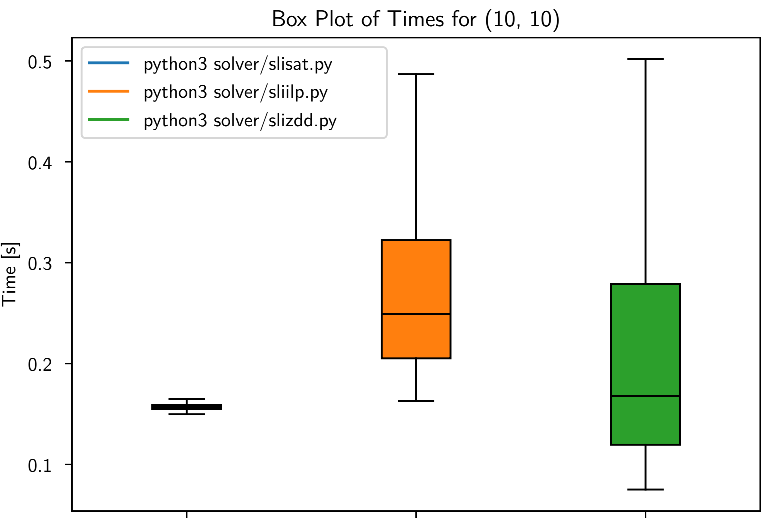


Abbildung 3.4: Verteilung der gemessenen Zeiten für die verschiedenen Puzzles der Größe (10, 10) mit IP, SAT und ZDD.

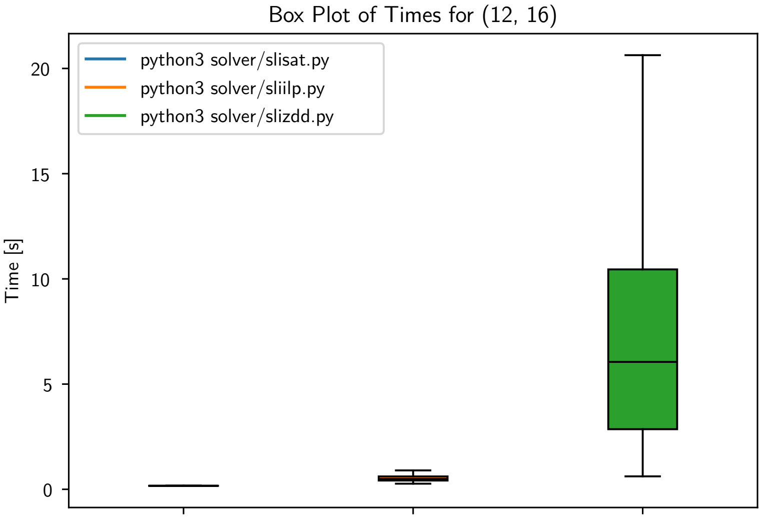


Abbildung 3.5: Verteilung der gemessenen Zeiten für die verschiedenen Puzzles der Größe (12, 16) mit IP, SAT und ZDD.

Wie in Tabelle 3.2 dargestellt hat der ZDD-Solver insgesamt 59 mal das Zeitlimit von 60 Sekunden errreicht. Davon waren alle 58 Puzzles der Größe (20, 36) und ein (12, 16) Puzzle betroffen. Außerdem gab es zwei mal eine Fehlermeldung aufgrund mangelnder Speicherressourcen.

Tabelle 3.2: Vergleich der Korrektheit für SAT, IP und ZDD für alle untersuchten Puzzles.

	SAT	IP	ZDD
Anzahl an Fehlermeldungen	0	0	2
Anzahl an Timeouts	0	0	59
Anteil korrekt gelöster Puzzles	533/533 = 100.00%	533/533 = 100.00%	472/533 = 88.55%

3.4 Suche nach allen gültigen Lösungen

Der implementierte ZDD-Ansatz sucht stets nach allen gültigen Lösungen für eine Slitherlink-Instanz. Um die Suche nach allen gültigen Lösungen vergleichen zu können, wurde der SAT-Solver so angepasst, dass er nicht nach der ersten gefundenen Lösung abbricht, sondern solange iteriert, bis keine gültige Variablenbelegung mehr gefunden werden kann.
Für den Vergleich wurde ein (20, 20) Puzzle verwendet, bei dem 12 zufällige Zahlen aus dem Puzzle entfernt wurden. Durch diese Anpassung gab es für das Puzzle dann keine eindeutige Lösung mehr, sondern insgesamt 113.100 unterschiedliche Lösungen. Die gemessenen Zeiten sind in Abbildung 3.6 dargestellt. Mit ca. 40,916 Sekunden war der ZDD-Ansatz für das modifizierte Puzzle mehr als doppelt so schnell wie der SAT-Solver mit ca. 86,096 Sekunden.

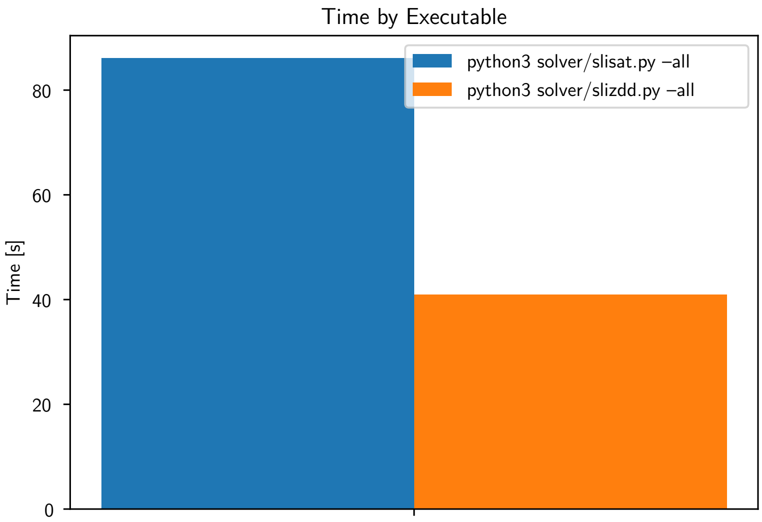


Abbildung 3.6: Vergleich der Zeiten für die Suche nach allen gültigen Lösungen mit SAT und ZDD.

3.5 Verwendung des Presolvers

Alleine durch die Verwendung des Presolvers konnten 40% der (5, 5) Puzzles, 78,88% der (10, 10) Puzzles, 55.56% der (12, 16) Puzzles und 67,24% der (20, 36) Puzzles vollständig gelöst werden, ohne dass ein weiteres Suchverfahren notwendig war. Der Presolver konnte im Vergleich zum SAT-Ansatz in Abbildung 3.7 diese Puzzles erheblich schneller lösen. Durchschnittlich benötigte der Presolver

etwa 0,019 bis 0,033 Sekunden für die verschiedenen Puzzlegrößen. Für Puzzles der Größe (20, 36) war der Presolver durchschnittlich ca. 88,6% schneller.

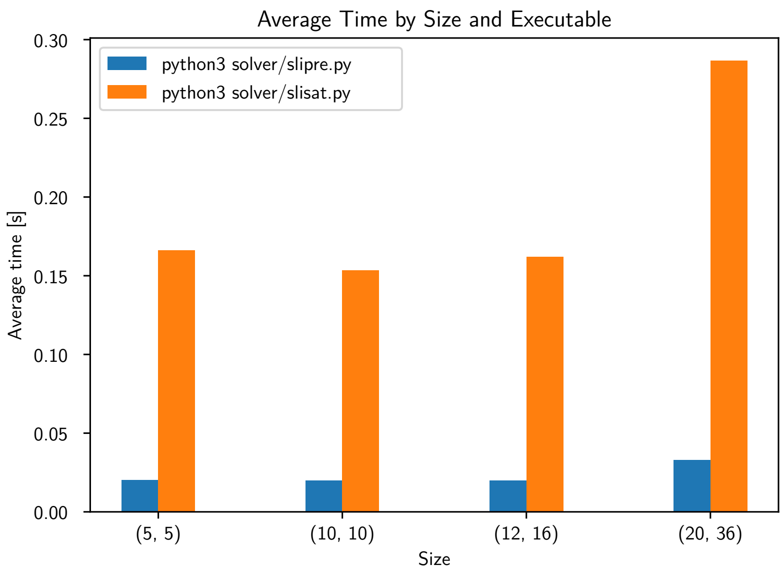


Abbildung 3.7: Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für den Presolver und SAT. Es wurden nur solche Instanzen getestet, die vollständig durch den Presolver lösbar sind.

In einem weiteren Schritt wurde untersucht, welche Verbesserungen durch die Reduktion des Suchraums mit den Informationen des Presolvers über sichere und ausgeschlossene Kanten für die verschiedenen Lösungsmethoden erreicht werden konnte. Dabei wurden solche Puzzles betrachtet, die sich nicht vollständig durch den Presolver lösen lassen. Die Balkendiagramme der Abbildungen 3.8, 3.9 und 3.10 stellen die Zeitgewinne bei Verwendung des Presolvers grafisch dar.
In Tabelle 3.3 sind die relativen durchschnittlichen Laufzeitverbesserungen aufgelistet. Insgesamt konnte für alle Größen und Lösungsmethoden eine Verbesserung erreicht werden. Für IP waren Verbesserungen von 14,0% bis 61,1% zu beobachten, wobei mit steigender Größe der Effekt größer war. Der relative Zeitgewinn war beim SAT-Solver etwas geringer, für Puzzles der Größe (20, 36) war die angepasste Variante ca. 8,9% besser. Der ZDD-Solver konnte am meisten von der Reduktion des Suchraums profitieren. Bei den untersuchten Puzzles der Größe (10, 10) waren die durch den Presolver optimierten Versionen für SAT- und ZDD-Solver ähnlich gut. Der ZDD-Solver zeigte eine Reduktion von 96,3% für die Größe (12, 16). Zudem gab es im Vergleich keinen Timeout, während zuvor für eine Instanz das Zeitlimit erreicht wurde. Mithilfe des Presolvers konnte der ZDD-Ansatz auch Puzzles der Größe (20, 36) lösen. 68,42% der nicht vollständig durch den Presolver lösbaren (20, 36) Puzzles konnten damit korrekt gelöst werden.

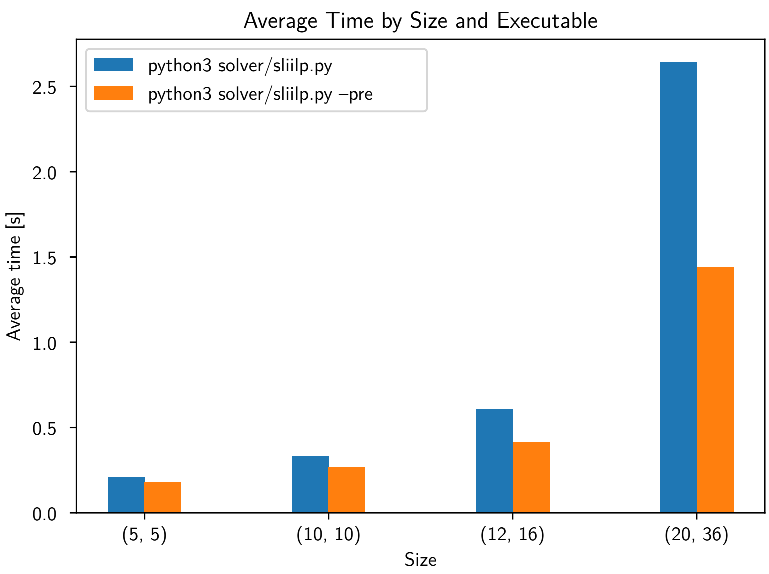


Abbildung 3.8: Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für IP ohne und mit Verwendung des Presolvers.

Tabelle 3.3: Verbesserung der durchschnittlichen Laufzeiten durch Verwendung des Presolvers in Sekunden. IP*, SAT* und ZDD* meinen hierbei die optimierten Versionen.

Größe	**IP**	**IP***	**SAT**	**SAT***	**ZDD**	**ZDD***
(5, 5)	0,21103	0,18142 (-14,0%)	0,16260	0,15699 (- 3,4%)	0,07453	0,06717 (- 9,9%)
(10, 10)	0,33314	0,26891 (-19,3%)	0,15898	0,15329 (- 3,5%)	0,32813	0,15723 (-52,1%)
(12, 16)	0,60843	0,41171 (-32,3%)	0,18318	0,16424 (-10,3%)	6,61747	0,24123 (-96,3%)
(20, 36)	2,64385	1,44270 (-45,4%)	0,30274	0,27590 (-8,9%)	-	4,61004

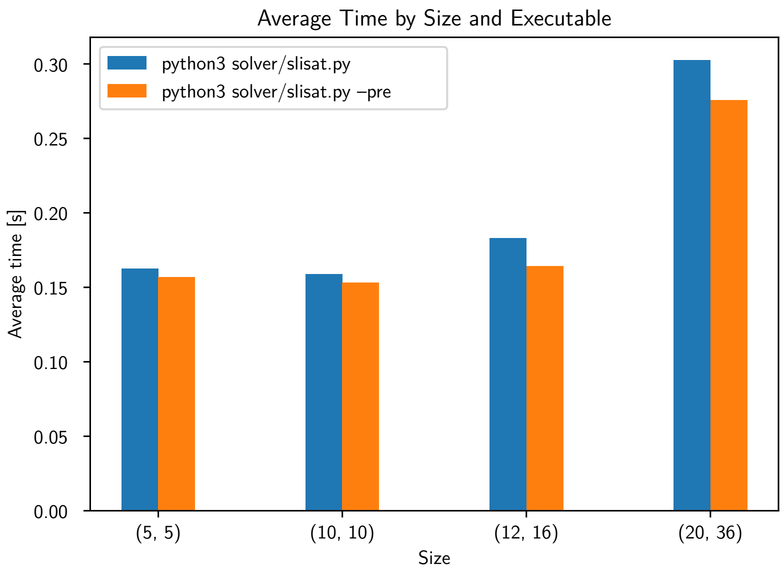


Abbildung 3.9: Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für SAT ohne und mit Verwendung des Presolvers.

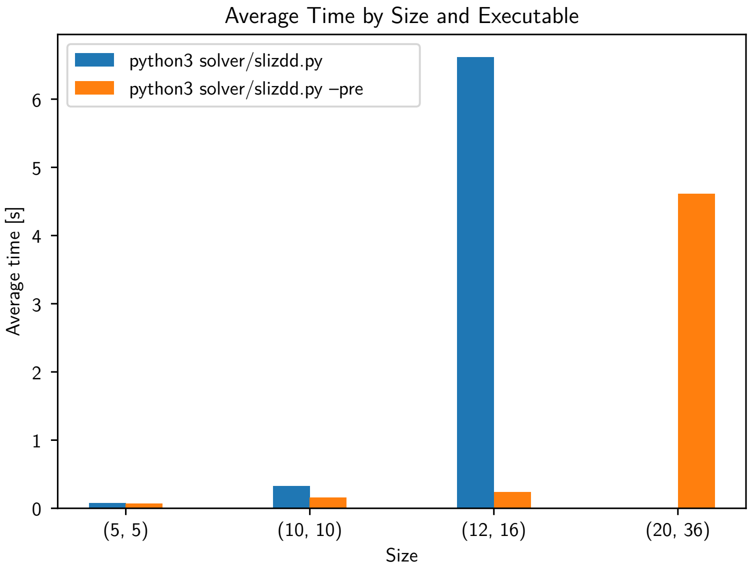


Abbildung 3.10: Vergleich der durchschnittlichen Laufzeiten nach Puzzlegröße für ZDD ohne und mit Verwendung des Presolvers.

4 Diskussion

Unter den drei untersuchten Ansätzen hat der SAT-Solver im Allgemeinen die beste Performance gezeigt. Gerade bei den größeren Beispielinstanzen lässt sich dies eindeutig beobachten. Obwohl die SAT-Formulierung lediglich eine Übersetzung der IP-Lösung ist, waren die Open-Source SAT-Solver deutlich schneller als die getesteten kommerziellen IP-Solver. Dies könnte darauf zurückzuführen sein, dass die IP-Solver darauf optimiert sind, eine Zielfunktion unter Berücksichtigung von Bedingungen zu maximieren oder minimieren, während SAT-Solver lediglich nach erfüllenden Variablenbelegung suchen müssen.
Mit steigender Puzzlegröße nimmt auch die durchschnittliche Rechenzeit zu. Für IP und ZDD lässt sich ein exponentieller Anstieg der Rechenzeit mit steigender Puzzlegröße beobachten. Beim SAT-Solver befinden sich die Messergebnisse für die Größen (5, 5) bis (12, 16) jedoch alle auf einem ähnlichen Niveau. Auch wenn Puzzles der Größe (12, 16) 7,68-mal so viele Zellen wie (5, 5) Puzzles haben, benötigt der SAT-Solver im Durchschnitt nur etwa 1,069-mal mehr Zeit. Dies deutet darauf hin, dass ein erheblicher Anteil der verbrauchten Zeit für die Probleminitialisierung bei PySAT verwendet wird.
Beim IP- und ZDD-Ansatz lassen sich zum Teil sehr große Schwankungen bei unterschiedlichen Puzzles der selben Größe feststellen. Der Erfolg der getesteten Methoden ist stark vom konkreten Fallbeispiel abhängig[1]. Betrachtet man in Tabelle 4.1 bspw. die drei (12, 16) Instanzen, die am schnellsten (janko0313, janko0086, janko0926) und am langsamsten (janko0165, janko0163, janko0355) vom ZDD-Solver gelöst werden konnten, so fällt auf, dass die langsamer lösbaren Puzzles verhältnismäßig viele Zellen mit einer 2 enthalten, während sich Puzzles mit vielen 0-, 1- oder 3-Zellen durch den ZDD-Solver schneller gelöst werden konnten. Dies lässt sich womöglich damit begründen, dass bei 2-Zellen die größte Anzahl an Teilgraphen durch den Algorithmus inkludiert wird, wie in Tabelle 2.4 beobachtet werden konnte.

Tabelle 4.1: Kürzeste und längste Rechenzeiten des ZDD-Solvers für Puzzles der Größe (12, 16).

Instanz	Anzahl an 0-Zellen	Anzahl an 1-Zellen	Anzahl an 2-Zellen	Anzahl an 3-Zellen	Laufzeit
janko0313	17	28	13	16	0,61544 s
janko0086	6	27	20	31	1,05674 s
janko0926	6	23	23	20	1,09127 s
janko0355	2	19	52	24	31,16379 s
janko0163	4	13	51	16	31,25386 s
janko0165	5	13	28	25	35.40781 s

Die Graphillion-Autoren [3] haben ihren Lösungsansatz für jeweils eine Slitherlink-Instanz der Größen (10, 10), (10, 18) und (20, 36) getestet. In Anbetracht der Tatsache, dass die Test mit einer

[1] Detaillierte Zeiten für einzelne Instanzen: `https://git.tu-berlin.de/berscjak/slitherlink-ba/-/tree/main/results?ref_type=heads`

schnelleren CPU und dem doppelten verfügbaren Arbeitsspeicher durchgeführt wurden, lassen sich die Ergebnisse nur bedingt miteinander vergleichen. Ein allgemein exponentieller Anstieg bei der Rechenzeit in Abhängigkeit von der Puzzlegröße konnte auch dort beobachtet werden. Für das (10, 10) und das (10, 18) Puzzle lag die verwendete Zeit im Bereich um 0,1 Sekunden. Bei den Experimenten in dieser Arbeit konnte der ZDD-Solver ein (10, 10) Puzzle im besten Fall in 0,075 Sekunden lösen und es gab ein (12, 16) Puzzle, welches sich in 0,615 Sekunden lösen ließ. Zudem soll das Puzzle der Größe (20, 36) in unter 60 Sekunden gelöst worden sein, was in den Versuchen dieser Arbeit nicht erreicht werden konnte.

In der Arbeit, die zuerst einen dedizierten ZDD-basierten Slitherlink-Solver vorgestellt hat [16], wurden mehrere verschiedene Instanzen derselben Größe für den ZDD-Algorithmus sowie für CPLEX mit einer IP-Formulierung getestet. Die Testumgebung war mit 512 GB RAM auch deutlich stärker. Es waren aber ähnlich wie den Testversuchen dieser Arbeit deutliche Unterschiede bei den gemessenen Zeiten für verschiedene gleich große Instanzen zu beobachten. So brauchte der ZDD-Algorithmus für (10, 10) Puzzles 0,044 bis 0,384 Sekunden, für CPLEX lagen die Zeiten für diese Größe zwischen 0,07 und 0,28 Sekunden.

In [3] geben die Autoren an, dass ihr ZDD-Algorithmus aus [16] der schnellste wäre, um alle möglichen Lösungen aufzulisten. Mit den Ergebnissen aus Abschnitt 3.4 konnte diese Aussage nachgewiesen werden. Der allgemein sehr effiziente SAT-Solver konnte bei der Suche nach allen Lösungen mit einer mehr als doppelt so kurzen Zeit deutlich durch den ZDD-Ansatz geschlagen werden.

In 2020 wurden 840 Slitherlink-Instanzen von janko.at mit einem Copris-Solver von Naoyuki Tamura [13] getestet. Copris ist eine in Scala eingebettete Constraint Programmiersprache. Die Tests wurden unter Verwendung von GlueMiniSat (2.2.10) als SAT-Solver durchgeführt. Dabei brauchte der Copris-Solver für eines der 840 Puzzles durchschnittlich 7,5 Sekunden. Die im Rahmen dieser Bachelorarbeit untersuchte SAT-Formulierung mit PySAT lag hingegen selbst bei der Puzzlegröße (20, 36) durchschnittlich bei etwa 0,3 Sekunden für alle getesteten Solver.

Innerhalb der theoretischen Informatik ist die Anwendung von Datenreduktionsregeln ein verbreiteter Ansatz um NP-schwere Probleme effizienter Lösen zu können. Datenreduktionsregeln werden dabei als Algorithmen mit polynomiellen Zeitaufwand betrachtet, die für eine gegebene Probleminstanz eine reduzierte, äquivalente Instanz des selben Problems (Kernel) zurückgeben. Beispiele für Datenreduktionsregeln für das Problem Vertex Cover wurden durch Figiel et. al. [2] gezeigt. In diesem Sinne lässt sich das in Abschnitt 2.4 vorgestellte Vorgehen als Anwendung von Datenreduktionsregeln begreifen.

Die Ergebnisse aus Abschnitt 3.5 zeigen, dass mit Hilfe von Datenreduktionsregeln signifikante Verbesserungen für die verschiedenen Lösungsverfahren von Slitherlink erreicht werden konnten. Insgesamt konnte der implementierte Presolver 73,7% der untersuchten Puzzle vollständig lösen. Dabei war dieser klar schneller als die anderen Methoden. Dies lässt sich damit begründen, dass die exakten Lösungsmethoden Slitherlink nur mit exponentiellen Zeitaufwand lösen können, während der Presolver in Polynomzeit arbeitet. Zudem sind Regeln dieser Art spezifisch auf das Problem Slitherlink zugeschnitten, während die anderen Ansätze allgemeiner sind. Darüber hinaus sei zu bedenken, dass Online verfügbare Logikrätsel, wie z.B. auf janko.at, in der Regel so gestaltet sind, dass sie sich von Menschen mittels logischen Schlussfolgerns lösen lassen. Der Erfolg beim Einsatz von verbreiteten menschlichen Lösungsmustern, wie es bei dem Presolver der Fall ist, ist demnach zu erwarten.

Insgesamt trägt diese Arbeit zum Vergleich ausgewählter deterministischer Lösungsansätze für Slitherlink unter ähnlichen Testbedingungen bei. Einzelne Ergebnisse verwandter Arbeiten konnte jedoch nicht im Detail reproduziert werden. Es wurde gezeigt, dass mit Hilfe eines Presolvers andere Lö-

sungsansätze optimiert werden konnten. Neben den hier thematisierten Vorgehen gibt es z.B. auch probabilistische Ansätze wie Monte-Carlo-Tree-Search (MCTS) um Puzzles zu lösen, die in dieser Arbeit nicht behandelt wurden. Kiarostami et. al. [6] haben mittels MCTS gute Ergebnisse für Slitherlink erzielen können. Außerdem beleuchtet diese Arbeit lediglich die aufgebrachte Rechenzeit zum Lösen einer Instanz und bietet keinen weiteren Einblick auf sonstigen Ressourcenverbrauch z.B. im Bezug auf Arbeitsspeicher.

5 Fazit

Die vorliegende Arbeit stellt unterschiedliche Methoden zum Finden von Lösungszyklen für das Logikrätsel Slitherlink vergleichend gegenüber. Neben den Constraint-basierten Ansätzen durch Integer Programming und SAT-Modellierung wurde die Möglichkeit zum Lösen von Slitherlink durch Zero-Suppressed Decision Diagrams untersucht.
Im direkten Vergeich war der SAT-Solver in fast allen Fällen der schnellste Ansatz. Es wurde gezeigt, dass es sinnvoll sein kann, ILP-Formulierungen in SAT-Formulierungen umzuwandeln. So hat sich die reine Übersetzung als aussagenlogische Formel deutlich effektiver erwiesen als die urspüngliche ILP-Formulierung. Insgesamt war der der untersuchte ZDD-Ansatz weniger leistungsfähig im Vergleich zu den anderen Methoden. Besonders bei den größeren Testinstanzen wird klar, dass die Performance moderner SAT- oder IP-Solver nicht erreicht werden kann. Zudem wiesen die Messergebnisse für das ZDD-Programm deutliche Schwankungen für die selbe Puzzlegröße aus, während vor allem die SAT-Zeiten relativ stabil sind. Für einige getestete Instanzen konnte der ZDD-Solver keine Lösung innerhalb von 60 Sekunden finden. Beim Suchen nach allen gültigen Lösungen war der ZDD-Solver hingegen deutlich besser als der SAT-Solver.
Außerdem hat diese Arbeit gezeigt, wie ein Presolver mithilfe bekannter Muster zur Lösung von Slitherlink beitragen kann. Für einen großen Teil der Puzzles konnte der Presolver eine vollständige Lösung bieten, während er bei anderen den Suchraum für die verschiedenen Lösungsmethoden reduzieren konnte. Besonders der ZDD-Solver konnte davon profitieren. Für zukünftige Arbeiten könnte die Entwicklung eines dedizierten Solvers mit Backtracking-Suche unter Verwendung von weiteren Reduktionsregeln ein vielversprechender Ansatz sein. Insbesondere kann der Vergleich der Effizienz einzelner Datenreduktionsregeln von Interesse sein.
Zusammenfassend lässt sich sagen, dass von den untersuchten Methoden die SAT-Modellierung, idealerweise in Kombination mit dem Presolver, die vielversprechendste Vorgehensweise ist. Zero-Suppressed Decision Diagrams mögen zwar eine interessante Datenstruktur sein, erwiesen sich jedoch als weniger leistungsfähig und zuverlässig bei der Suche nach einer Lösung für eine gegebene Slitherlink-Instanz. Ein größerer Vorteil von ZDDs liegt vielmehr in der Möglichkeit, Operationen auf Mengenfamilien effizient ausführen zu können. Bei Problemstellung, die nicht nur auf die Suche nach einer einzigen erfüllenden Lösung abzielen, wie z.B. die Suche nach allen Lösungen für ein Puzzle, kann die Verwendung von ZDDs hingegen von großem Nutzen sein.

Literaturverzeichnis

[1] Hannah Brown, Lei Zuo, and Dan Gusfield. Comparing integer linear programming to sat-solving for hard problems in computational and systems biology. In *International Conference on Algorithms for Computational Biology*, pages 63–76. Springer, 2020.

[2] Aleksander Figiel, Vincent Froese, André Nichterlein, and Rolf Niedermeier. There and back again: On applying data reduction rules by undoing others. *arXiv preprint arXiv:2206.14698*, 2022.

[3] Takeru Inoue, Hiroaki Iwashita, Jun Kawahara, and Shin-ichi Minato. Graphillion: Software library designed for very large sets of graphs in python. *Hokkaido University, Division of Computer Science, TCS Technical Reports*, 2013.

[4] Ravindran Kannan and Clyde L. Monma. On the computational complexity of integer programming problems. In Rudolf Henn, Bernhard Korte, and Werner Oettli, editors, *Optimization and Operations Research*, pages 161–172, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg.

[5] Muhammad Kholilurrohman and Shin-ichi Minato. An efficient algorithm for enumerating eulerian paths. *Hokkaido University, Division of Computer Science, TCS Technicial Reports*, 2014.

[6] Mohammad Sina Kiarostami, Mohammadreza Daneshvaramoli, Saleh Khalaj Monfared, Aku Visuri, Helia Karisani, Simo Hosio, Hamed Khashehchi, Ehsan Futuhi, Dara Rahmati, and Saeid Gorgin. On using monte-carlo tree search to solve puzzles. In *Proceedings of the 2021 7th International Conference on Computer Technology Applications*, pages 18–26, 2021.

[7] Donald E Knuth. *The art of computer programming, volume 4, fascicle 1*. Addison-Wesley Educational, Boston, MA, March 2009.

[8] Ruirning Li, Dian Zhou, and Donglei Du. Satisfiability and integer programming as complementary tools. In *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No. 04EX753)*, pages 880–883. IEEE, 2004.

[9] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, pages 272–277, 1993.

[10] Shin-ichi Minato. Techniques of bdd/zdd: brief history and recent activity. *IEICE TRANSACTIONS on Information and Systems*, 96(7):1419–1429, 2013.

[11] Tetsuo Miyauchi and Kiyofumi Tanaka. Solving slitherlink with fpga and smt solver. *Journal of Information Processing*, 28:959–969, 2020.

[12] Jonathan Olson. How slitherlink should be solved. `https://jonathanolson.net/slitherlink/`. Accessed: 3. November, 2024.

[13] Naoyuki Tamura. Slitherlink solver in copris. https://cspsat.gitlab.io/copris-puzzles/slitherlink/index.html, 2020. Accessed: 2. November, 2024.

[14] Wikipedia. Slitherlink — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Slitherlink&oldid=1239925987, 2024. Accessed: 3. November, 2024.

[15] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(5):1052–1060, 2003.

[16] 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(20):176–213, 2012.

[17] Sugimura Yuka. Slither link solution using integer programming, September 2005. Mini Research Meeting "Combination Games and Puzzles", Kyoto University, Faculty of Engineering, Information Lecture Room 1, Kyoto.