

Gottfried Wilhelm Leibniz Universität Hannover
Institut für Theoretische Informatik

NP-Vollständigkeit von ZHED

Bachelorarbeit

Dirk Siekmann

Matrikelnr. 03160670

Hannover, den 6. März 2025

Erstprüfer: Prof. Dr. Arne Meier
Zweitprüfer: Prof. Dr. Heribert Vollmer
Betreuer: Nicolas Fröhlich

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 6. März 2025

Dirk Siekmann

Inhaltsverzeichnis

1	Einleitung	1
2	Grundlagen	2
2.1	NP-Vollständigkeit	2
2.2	ZHED	4
2.3	Rectilinear Planar Monotone 3SAT	5
3	Reduktion von RPM-3SAT auf ZHED	10
3.1	Gadgets	10
3.2	Pseudocode und Laufzeit	16
3.3	Korrektheit	24
4	Programmierung	27
4.1	Packages	27
4.2	Implementierung	28
5	Zusammenfassung und Ausblick	32

1 Einleitung

Die Frage, ob die Komplexitätsklassen P und NP gleich sind, ist eines der größten ungelösten Probleme in der theoretischen Informatik. Um Gleichheit zu zeigen würde es reichen, dass ein NP-schweres Problem (ein Problem, das mindestens so schwer ist, wie alle anderen Probleme in NP) in Polynomialzeit, das heißt effizient, lösbar ist. Gleichheit dieser Klassen würde bedeuten, dass alle Probleme aus NP in Polynomialzeit lösbar wären [13]. Da diese oft praktische Anwendung haben, könnte das möglicherweise zu großen Effizienzsteigerungen führen. Aber auch zu Problemen, da wichtige Kryptographieverfahren wie das RSA-Verfahren [11] darauf basieren, dass bestimmte Probleme in NP, nicht effizient lösbar sind.

Im Jahr 2021 zeigten Sagnik Saha und Erik D. Demaine, dass das Puzzlespiel ZHED [5] NP-vollständig ist, in dem sie das Problem Planar Monotone Rectilinear 3SAT (RPM-3SAT) auf ZHED reduzierten [12].

Ziel dieser Arbeit soll sein, die von Saha und Demaine vorgestellte Reduktion nachzuvollziehen und diese konkret auf ihre Laufzeit und Korrektheit zu untersuchen. Dafür sollen die in [12] verwendeten Bausteine, auch Gadgets genannt, vorgestellt und erläutert werden, warum sich diese in Polynomialzeit bilden und zusammenfügen lassen.

Außerdem soll ein Programm geschrieben werden, das ein gegebenes Problem aus RPM-3SAT in ein ZHED-Spiel umwandelt. Anhand dessen soll veranschaulicht werden, dass das entstehende ZHED Spiel genau dann eine Lösung hat, wenn die zugrunde liegende RPM-3SAT Formel erfüllbar ist.

2 Grundlagen

In diesem Kapitel werden Definitionen und Fachbegriffe, die für diese Arbeit wichtig sind, eingeführt.

2.1 NP-Vollständigkeit

Bevor wir über den Begriff der NP-Vollständigkeit sprechen, führen wir zuerst ein paar andere Begrifflichkeiten ein. Die Definitionen sind angepasst nach Standardliteratur [4, 13].

Sei Σ eine endliche Menge von Symbolen und Σ^* die Menge endlicher Aneinanderreihungen, auch *Wörter* genannt, von Symbolen aus Σ . Eine *Sprache* L ist eine Teilmenge von Σ^* , es wird gesagt L ist eine Sprache über das *Alphabet* Σ .

Eine deterministische Turingmaschine (DTM) ist wie folgt definiert.

Definition 2.1. Eine DTM ist ein 6-tuple $M = (Z, \Gamma, \delta, z_0, A, V)$, wobei

- Z – Menge der Zustände,
- Γ – Menge der Bandsymbole,
- $\delta: Z \times \Gamma \rightarrow Z \times \Gamma \times \{L, R, N\}$, die Übergangsfunktion,
- z_0 – Startzustand,
- A – Menge der akzeptierenden Zustände,
- V – Menge der verwerfenden Zustände.

Eingaben werden von einer DTM genau dann akzeptiert, wenn die Folge von Übergangsfunktionen, die sich aus der Eingabe ergeben, zu einem $z \in A$ führt.

Eine nichtdeterministische Turingmaschine (NTM) unterscheidet sich von einer DTM nur in ihrer Übergangsfunktion

$$\delta: Z \times \Gamma \rightarrow \mathcal{P}(Z \times \Gamma \times \{L, R, N\}).$$

Turingmaschinen, die eine Sprache L *entscheiden*, halten genau dann, wenn die Eingabe w ein Element in L ist, auf einem akzeptierenden Zustand. Liegt w nicht in L hält die Turingmaschine auf einem verwerfenden Zustand.

Die Anzahl der Schritte, die eine Turingmaschine benötigt um zu halten beschreiben wir mit der Zeitkomplexität. Vor dieser führen wir aber noch die O -Notation ein:

Definition 2.2. Seien f und g Funktionen mit $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$. Dann ist

$$f(n) \in O(g(n)) \Leftrightarrow \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty.$$

Zurück zur Zeitkomplexität, für die wir eine monoton wachsende Funktion in den natürlichen Zahlen betrachten. Und beschreiben die Komplexitätsklasse als:

Definition 2.3. Sei $t: \mathbb{N} \rightarrow \mathbb{N}$ eine monotone wachsende Funktion und n die Länge der Eingabe. Die Komplexitätsklasse $\text{TIME}(t(n))$ ist die Menge aller Sprachen, die von einer DTM in maximal $O(t(n))$ Schritten entschieden wird.

Für die NTM existiert eine entsprechende Komplexitätsklasse $\text{NTIME}(t(n))$, bei der die DTM gegen eine NTM ausgetauscht ist.

Eine spezielle Komplexitätsklasse ist die Klasse NP.

Definition 2.4. $\text{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$.

Eine Sprache liegt in NP, wenn sie in *Polynomialzeit* von einer NTM entschieden werden kann.

Für die Klasse NP ist auch der Begriff des *Verifizierers* wichtig.

Definition 2.5. Ein Verifizierer für eine Sprache $L \subseteq \Sigma^*$ ist eine TM M , wobei

$$L = \{w \in \Sigma^* \mid M \text{ akzeptiert } \langle w, c \rangle \text{ für ein Zertifikat } c\}.$$

Damit kann dann eine alternative Definition für die Klasse NP aufgestellt werden.

Satz 2.6. Sei L eine Sprache. Dann gilt

$$L \in \text{NP} \Leftrightarrow L \text{ hat einen Verifizierer, der in Polynomialzeit läuft.}$$

Beweis: Sei M eine DTM und V ein Verifizierer.

" \Rightarrow " Das Zertifikat ist ein Pfad im Berechnungsbaum von M . Dann simuliert V , M mit dem Zertifikat.

" \Leftarrow " M rät im nicht deterministischen Teil das Zertifikat und simuliert dann V .

□

In dieser Arbeit wird es wichtig sein Sprachen auf einander zu *reduzieren*, das bedeutet:

Definition 2.7. Seien $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Gamma^*$ zwei Sprachen. L_1 ist auf L_2 in *Polynomialzeit* m -*reduzierbar*, wenn es eine in Polynomialzeit laufende Funktion $f: \Sigma^* \rightarrow \Gamma^*$ gibt, sodass

für jedes $w \in \Sigma^*$ gilt

$$w \in L_1 \Leftrightarrow f(w) \in L_2.$$

Dies wird geschrieben als $L_1 \leq_m^P L_2$. Und damit kann dann der Begriff der *NP-schwere* eingeführt werden:

Definition 2.8. Sei L_1 eine Sprache. Dann heißt L_1 *NP-schwer*, wenn

$$\forall L_2 \in \text{NP} : L_2 \leq_m^P L_1.$$

Für NP-schwere Probleme gilt,

Satz 2.9. Sei eine Sprache L_1 NP-schwer, dann ist eine Sprache L_2 auch NP-schwer, wenn $L_1 \leq_m^P L_2$.

Mit dem Begriff der NP-schwere und der Komplexitätsklasse NP kommen wir nun zur *NP-Vollständigkeit* nach,

Definition 2.10. Eine Sprache ist NP-vollständig, wenn sie in NP liegt und NP-schwer ist.

2.2 ZHED

ZHED [5] ist ein Puzzle Spiel aus dem Jahr 2017. Es wird auf einem quadratischen Grid mit vier Feldertypen gespielt.

Ein Feld ist das *Zielfeld* und ein oder mehr Felder sind *Zahlfelder*, diese enthalten eine Zahl m . Bei einem Zug kann ein Zahlfeld ausgewählt und nach oben, unten, rechts oder links expandiert werden. In dieser Richtung werden die nächsten m *unmarkierte Felder* zu *markierten Feldern*. Außerdem gilt das Zahlfeld danach als “verbraucht” und wird ebenso zu einem markierten Feld, wie in Abbildung 2.1 dargestellt. Das Spiel ist lösbar, wenn es möglich ist, das Zielfeld zu markieren.

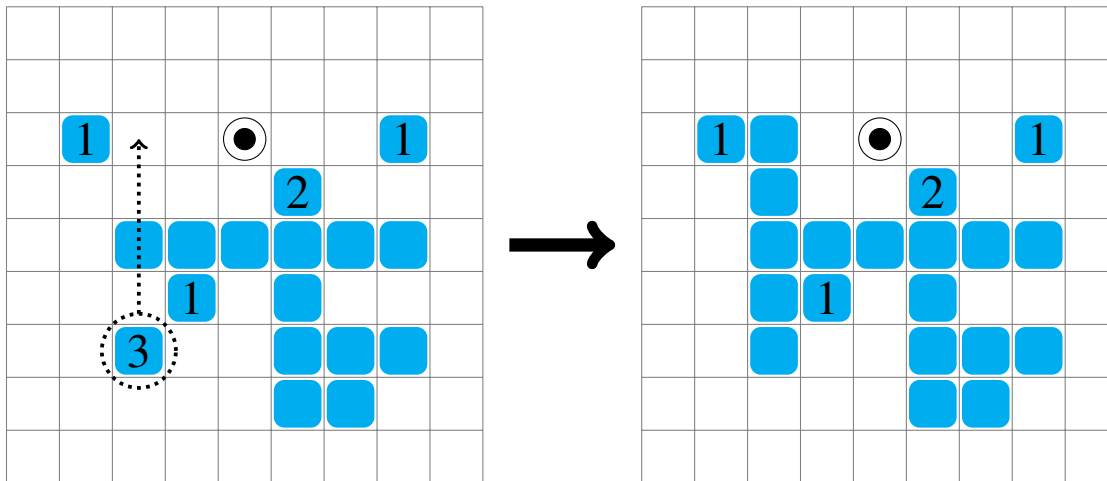


Abbildung 2.1: Darstellung eines ZHED-Spielzugs, bei dem das Zahlfeld auf Position (3, 3) nach oben expandiert wurde.

Sei n die Anzahl an Feldern, dann können maximal $n - 1$ Felder Zahlfelder sein, da das Zielfeld notwendigerweise vorhanden sein muss. Wenn nach jedem Zug das gesamte Spielfeld durchlaufen und gegebenenfalls neu gebaut wird, kann ein Spiel maximal $n^2 - n$ viele Schritte haben.

Felder können durch ihre x - und y -Koordinaten und ihren Feldertyp genau beschrieben werden. Dafür sind je Feld $\log(n) + 2$ Bits nötig ($\log \sqrt{n}$ Bits pro Koordinate und 2 Bits für den Feldertyp). Für ein Zahlfeld gilt $m < \sqrt{n}$, da das Spielfeld in einer Richtung nicht mehr Felder hat, damit brauchen wir noch pro Zahlfeld $\log \sqrt{n}$ Bits.

Ein Zertifikat kann also eine Reihe Koordinaten von Zahlfeldern mit Richtungen (2 Bits) sein. Dieses Zertifikat kann in Polynomialzeit überprüft werden. Damit ist ZHED in NP.

2.3 Rectilinear Planar Monotone 3SAT

In der Arbeit von Saha und Demaine [12] wird RPM-3SAT auf ZHED reduziert. In diesem Abschnitt zeigen wir daher zunächst, dass RPM-3SAT selbst NP-schwer ist [2].

Definition 2.11. [3] Ein Graph ist ein 2-Tupel $G = (V, E)$, wobei V die Menge der Knoten und E die Menge der Kanten bezeichnet und es gilt $E \subseteq V \times V$.

Wir teilen E auf in E_1 und E_2 , mit $E_1 \cap E_2 = \emptyset$ und $E_1 \cup E_2 = E$. Die Menge der *Variablen*, einer 3SAT-Formel, wird repräsentiert von E_1 und die Menge der *Klauseln* von E_2 . Dabei sind die Klausel-Knoten über Kanten mit Variablen-Knoten verbunden, genau dann, wenn die Klausel diese Variablen enthält.

Ein Graph muss folgende Eigenschaften erfüllen, um RPM-3SAT anzugehören. Er muss *rectilinear* sein, das bedeutet, er kann gezeichnet werden, sodass alle Knoten, die Variablen repräsentieren, in einer Reihe angeordnet sind. Die Knoten, die Klauseln repräsentieren, liegen

über oder unter den Variablen-Knoten.

Außerdem muss er *planar* sein, es existiert also eine Darstellung des Graphen, bei der sich keine Kanten kreuzen. Jede planare 3SAT-Formel kann so angeordnet werden, dass sie *rectilinear* ist [7, 6]. Damit sind also alle Klauseln so verschachtelt, dass die Planarität erhalten bleibt.

Zuletzt muss ein solcher Graph *monoton* sein, also alle Klauseln unterhalb der Variablen (negative Seite) enthalten nur negierte Variablen und alle Klauseln oberhalb (positive Seite) enthalten nur nicht negierte Variablen. In Abbildung 2.2 ist die RPM-3SAT Repräsentation der Formel

$$\varphi = (x_2 \vee x_8 \vee x_9) \wedge (x_3 \vee x_4 \vee x_5) \wedge (x_4 \vee x_5 \vee x_6) \wedge (x_7 \vee x_8 \vee x_9) \wedge (\bar{x}_1 \vee \bar{x}_6 \vee \bar{x}_7) \\ \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_4 \vee \bar{x}_5 \vee \bar{x}_6) \wedge (\bar{x}_7 \vee \bar{x}_8 \vee \bar{x}_9),$$

dargestellt.

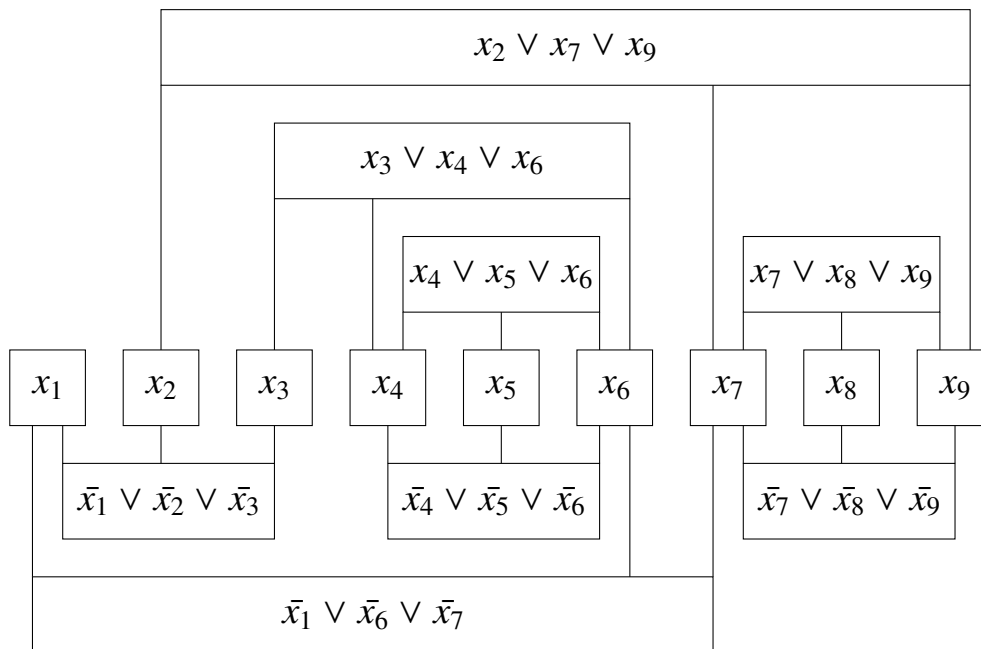


Abbildung 2.2: Eine beispielhafte RPM-3SAT Repräsentation der Formel φ .

Im Folgenden soll nun die Reduktion von *rectilinear planar 3SAT* (im Folgenden *RP-3SAT* genannt) auf *RPM-3SAT* nachvollzogen werden. Die Reduktion folgt der Arbeit von Agarwal et al. [1].

Bei der Reduktion müssen Klauseln untersucht werden, die nicht *monoton* sind, also von der Form $\bar{x} \vee y \vee z$ sind, sie werden als *inkonsistent* bezeichnet. Siehe dazu Abbildung 2.3.

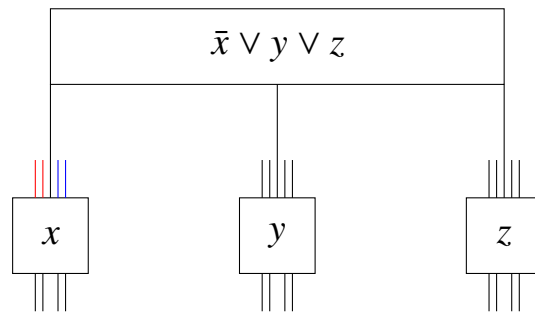


Abbildung 2.3: Eine nicht monotone Klausel. Kanten von x , die linksseitig der inkonsistenten Klausel liegen sind rot angedeutet und rechtsseitige sind blau angedeutet

Hier gilt es nun rechtsseitig der inkonsistenten Variablen zwei neue einzufügen, a und b , für die gilt, dass $x = \bar{a} = b$. Es sind also die Klauseln $x \vee a$, $a \vee b$, $\bar{x} \vee \bar{a}$ und $\bar{a} \vee \bar{b}$ einzufügen. Dann wird \bar{x} durch a in der inkonsistenten Klausel ersetzt und x durch b in allen Klauseln rechtsseitig der inkonsistenten Klausel auf der positiven Seite, wenn x negiert vorlag und auf der negativen Seite im anderen Fall. Dies ist in Abbildung 2.4 dargestellt.

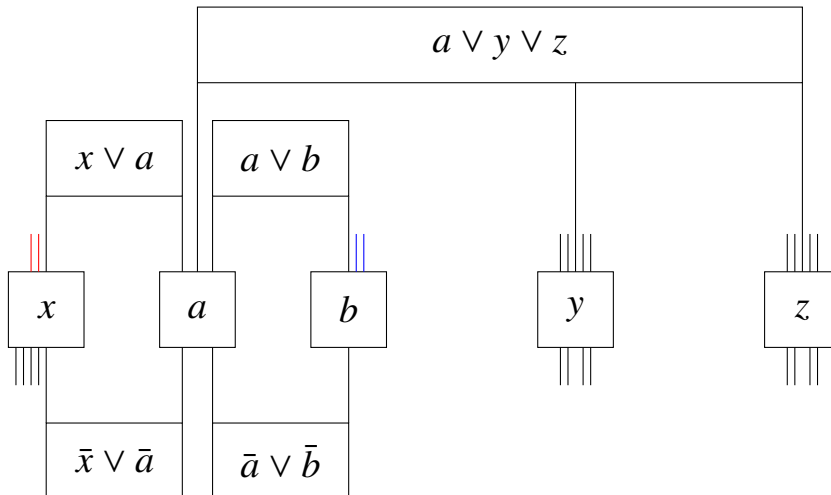


Abbildung 2.4: Fehlerbehebung für eine nicht monotone Klausel.

Die Wahrheitstabelle, für diese Korrektur, ist in Tabelle 2.1 dargestellt. Bei deren Betrachtung fällt auf, dass $(x \vee a) \wedge (a \vee b) \wedge (\bar{x} \vee \bar{a}) \wedge (\bar{a} \vee \bar{b})$ nur dann erfüllbar ist, wenn gilt $x = \bar{a} = b$, wie wir es gefordert hatten.

Tabelle 2.1: Wahrheitstabelle der Korrektur der inkonsistenten Variable.

x	a	b	$x \vee a$	$a \vee b$	$\bar{x} \vee \bar{a}$	$\bar{a} \vee \bar{b}$	$(x \vee a) \wedge (a \vee b) \wedge (\bar{x} \vee \bar{a}) \wedge (\bar{a} \vee \bar{b})$
0	0	0	0	0	1	1	0
0	0	1	0	1	1	1	0
0	1	0	1	1	1	1	1
0	1	1	1	1	1	0	0
1	0	0	1	0	1	1	0
1	0	1	1	1	1	1	1
1	1	0	1	1	0	1	0
1	1	1	1	1	0	0	0

Es könnte auch passieren, dass eine Klausel mit zwei oder drei positiven Variablen auf der negativen Seite liegt. Diese lässt sich nicht ohne weiteres auf die andere Seite legen, da es sonst zu Problemen mit der Planarität kommen kann, wie in Abbildung 2.5 zu sehen ist. Dies gilt auch für den umgekehrten Fall, dass negierte Variablen auf der positiven Seite liegen. Aus Gründen der Übersichtlichkeit werden die Klauseln im Folgenden nur noch als horizontale Linien dargestellt.

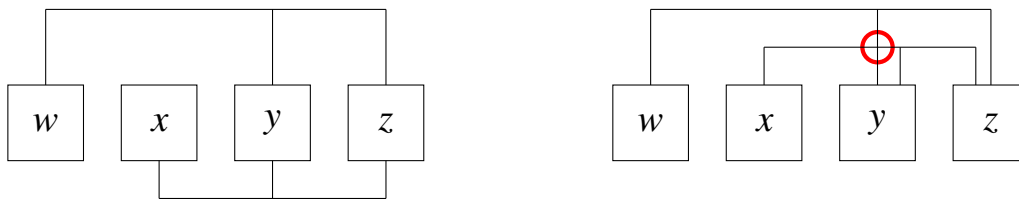


Abbildung 2.5: Eine RP-3SAT Repräsentation der Formel $(w \vee y \vee z) \wedge (x \vee y \vee z)$ auf der linken Seite. Auf der rechten Seite dieselbe Formel, wenn erfolglos versucht wird, diese in RPM-3SAT Repräsentation darzustellen.

Aber auch dieses Problem lässt sich mit dem zuerst vorgestellten Verfahren lösen. Es werden dafür für alle inkonsistenten Variablen zwei neue eingefügt und die rechtsseitigen Kanten, wie oben beschrieben, um verlegt. Abbildung 2.6 zeigt dies.

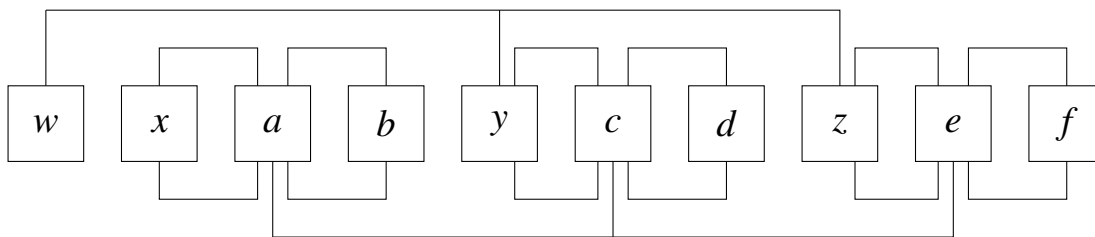


Abbildung 2.6: Eine Formel in RPM-3SAT Repräsentation, die äquivalent zu $(w \vee y \vee z) \wedge (x \vee y \vee z)$ ist.

Das Einfügen der neuen Variablen und der Austausch von x durch \bar{a} in der inkonsistenten Klausel laufen in konstanter Zeit und das Ersetzen von x mit b ist durch die Anzahl der

Klauseln begrenzt. Damit erfolgt die Reduktion in Polynomialzeit und RPM-3SAT ist NP-schwer.

3 Reduktion von RPM-3SAT auf ZHED

In diesem Abschnitt wird die Reduktion von Saha und Damaine [12] vorgestellt und erläutert, wie ein ZHED Puzzle aus einer beliebigen RPM-3SAT Formel gebaut wird. Bei allen Zahlfeldern gilt, dass deren jeweilige Zahl eins ist.

Dafür werden die Bausteine (auch Gadgets genannt), aus denen die Reduktion [12] aufgebaut ist, erklärt und Pseudocode aufgestellt, der die Reduktion beschreibt. Dabei werden auch Anpassungen an den bestehenden Gadgets vorgenommen. Abschließend wird alles auf Korrektheit überprüft.

3.1 Gadgets

Threshold-Gadget

Für die Reduktion werden einige Gadgets benötigt, wie zum Beispiel das Threshold-Gadget. Dieses besteht aus einer Linie alternierender Zahlfelder und unmarkierter Felder. Wenn von diesen welche von anderen Gadgets markiert werden können, bezeichnen wir sie als *Quelle*. Außerdem haben Threshold-Gadgets ein *Ziel*, das $k + 1$ Felder vom letzten Zahlfeld entfernt ist. Es gibt b Quellen. Das bedeutet, dass nur wenn mindestens k Quellen markiert wurden, auch das Ziel markiert werden kann.

Außerdem hat jedes Gadget eine *Bounding Box*. Diese umschließt den Bereich, in dem das Gadget Felder markieren kann.

In Abbildung 3.1 ist ein beispielhaftes Threshold-Gadget dargestellt mit $b = 5$ und $k = 3$, es gilt dabei allgemein $b \geq k$. Die Bounding Box erstreckt sich dabei jeweils sechs Felder über das erste und letzte Zahlfeld hinaus, da wenn fünf Quellen markiert sind, bis zu sechs Felder in einer Richtung durch Zahlfelder des Threshold-Gadgets markiert werden können. Außerdem ist die Bounding Box drei Felder hoch, da ein Zahlfeld auch nach oben oder unten expandiert werden kann. Hier und im Folgenden sind alle Zahlfelder durch schwarze Kästchen, alle unmarkierten Felder durch weiße Kästchen und die Bounding Box durch eine rote Umrandung markiert.

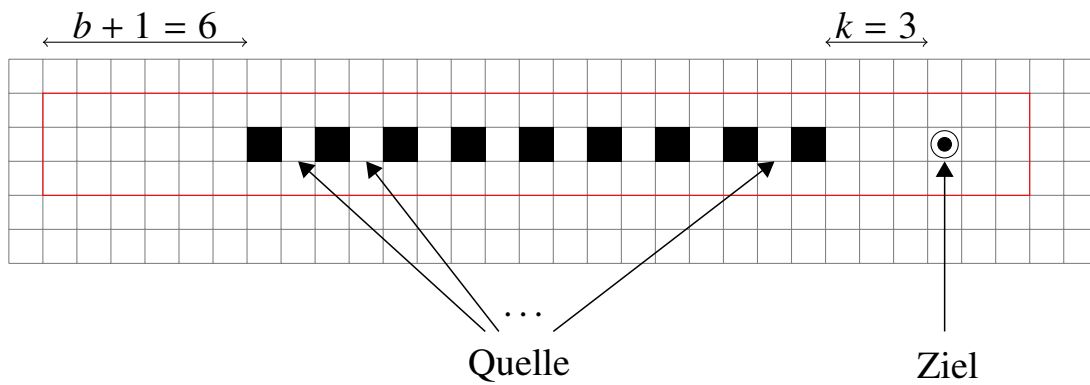


Abbildung 3.1: Darstellung eines Threshold-Gadget, bei dem fünf Quellen von Zahlfeldern außerhalb markierbar sind. Dieses Gadget kann nur aktiviert werden, wenn mindestens drei Quellen markiert sind.

Wenn gilt $k = b = 1$ sprechen wir von einem *Wire-Gadget*, es leitet ein Signal (markiert ein Feld) von der Quelle zum Ziel weiter.

Die Richtung der Signalleitung eines *Wire-Gadgets* kann geändert werden, in dem sein Ziel die Quelle eines anderen *Wire-Gadgets* ist. Es wird dann von einem *Turn-Gadget* gesprochen.

Sei nun $b > 1$, dann erhalten wir mit $k = b$ ein *AND-Gadget* und mit $k = 1$ ein *OR-Gadget*.

Verkettung von Threshold-Gadgets

Wie bei einem *Turn-Gadget* können auch beliebige *Threshold-Gadgets* zueinander stehen. Auch hier gilt das Ziel des einen Gadgets ist die Quelle des anderen. In Abbildung 3.2 ist dies exemplarisch dargestellt. Dabei ist zu beachten, dass die Bounding Box des horizontalen Gadgets um ein Feld nach rechts erweitert wird, da die Quelle (markiert durch ein \times) auch vom vertikalen Gadget markiert werden kann. Ansonsten ist die neue Bounding Box des gesamten Systems die Überschneidung der beiden einzelnen Bounding Boxen.

Die Gadgets sollten so aktiviert werden, dass zuerst das horizontale aktiviert wird und dann das vertikale.

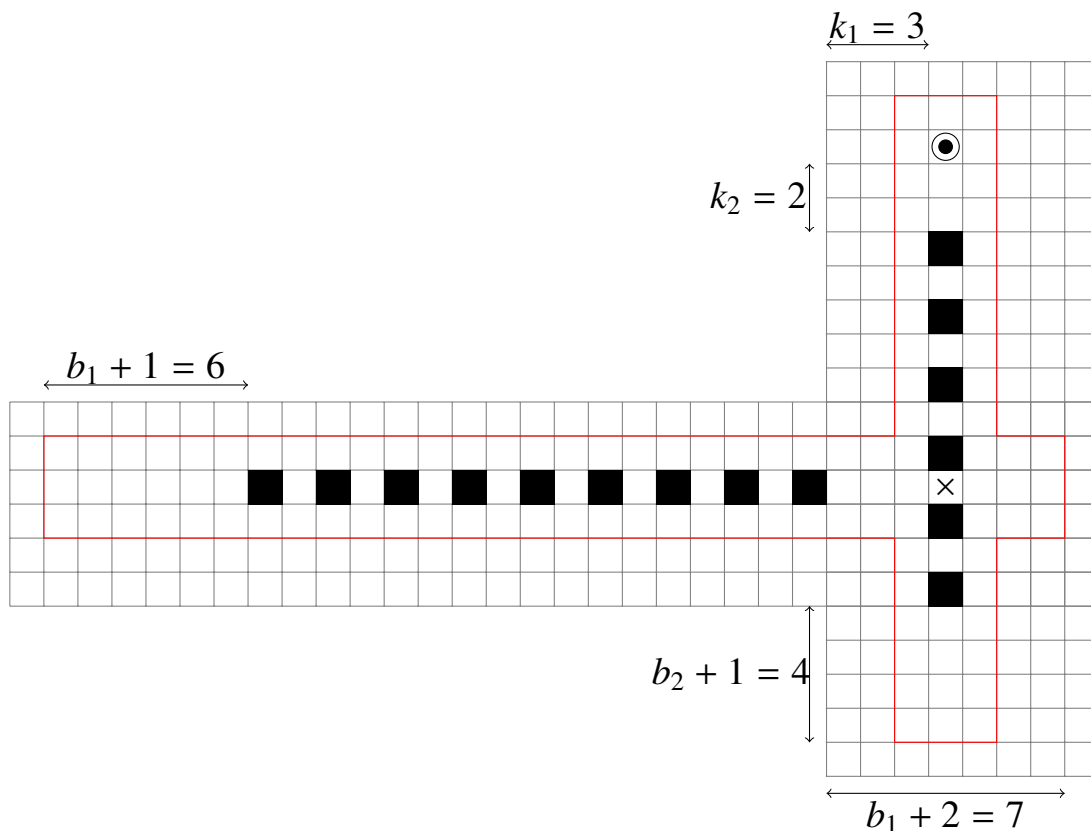


Abbildung 3.2: Eine Verkettung zweier Threshold-Gadgets. Dabei ist das Ziel des horizontalen Gadgets eine Quelle des vertikalen.

Shift-Gadget

Das Ziel eines Threshold-Gadgets kann mittels eines Shift-Gadgets um ein Feld verschoben werden. Ein Shift-Gadget wird aus einem Threshold-Gadget konstruiert, in dem hinter dem letzten Zahlfeld, das am weitesten vom Ziel entfernt ist, ein weiteres Zahlfeld eingefügt wird. Um diese Gadget zu aktivieren, werden alle Zahlfelder in Richtung des Ziels expandiert.

Die Bounding Box des ursprünglichen Threshold-Gadgets wird um ein Feld auf der Seite des Ziels erweitert und um zwei Felder auf der anderen Seite, außerdem wird das Ziel um ein Feld von den Zahlfeldern weg verschoben, siehe dazu Abbildung 3.3.

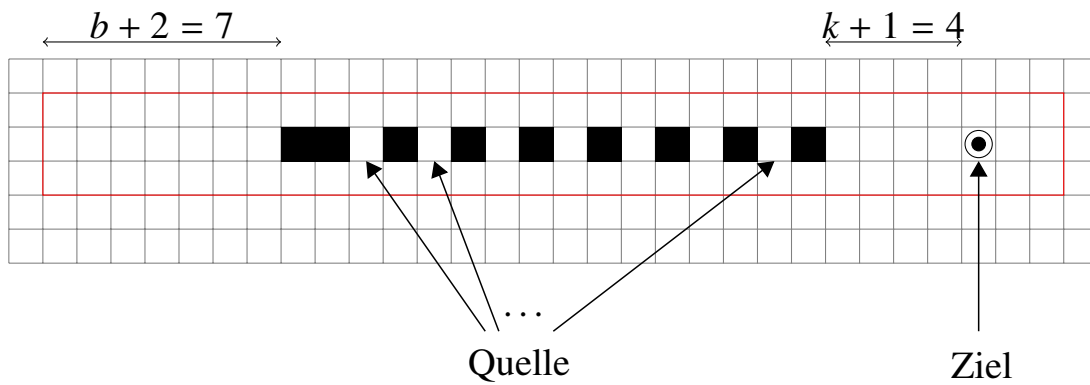


Abbildung 3.3: Ein Shift-Gadget das das Ziel eines Threshold-Gadget verschiebt. Das Threshold-Gadget hat fünf Quellen, von denen drei aktiviert werden müssen, damit das Ziel des Threshold-Gadgets markiert werden kann.

Variable-Gadget

Ein Variable-Gadget besteht aus einer geraden Anzahl L horizontalen Zahlfeldern direkt nebeneinander. Dabei entspricht eine Expansion aller Felder nach rechts, dass der Variablen der Wert *wahr* zu geordnet wird, beziehungsweise *falsch*, wenn alle nach links expandiert werden. Im Abstand von $L/2$ zu den L horizontalen Feldern werden Threshold-Gadgets eingefügt, die das Signal des Variable-Gadgets weiterleiten. Diese werden dabei so angeordnet, dass sich ihre Bounding Boxen nicht überschneiden, siehe dazu Abbildung 3.4.

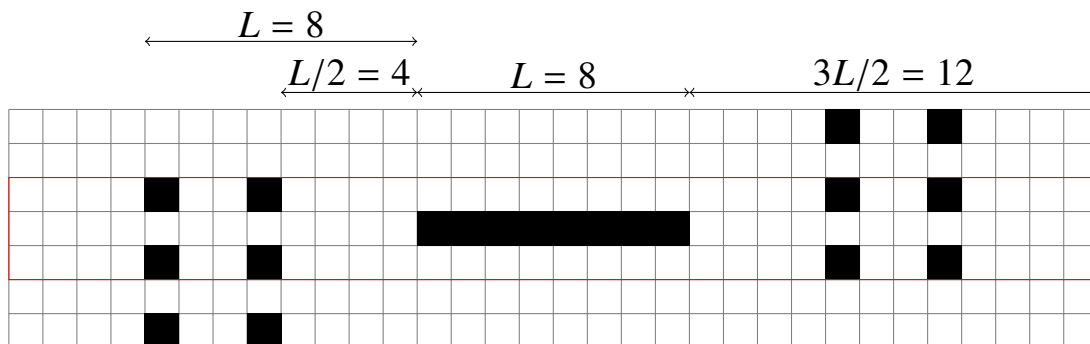


Abbildung 3.4: Ein Variable-Gadget mit $L = 8$ und Threshold-Gadgets auf der linken und rechten Seite, die das Signal des Variable-Gadgets weiterleiten. Wenn alle Variablen Zahlfelder nach rechts expandiert werden, wird der Wert der Variablen auf wahr gesetzt und auf falsch, wenn alle nach links expandiert werden.

Um die Quelle eines der Threshold-Gadgets zu markieren, muss das Variable-Gadget $L/2 + 1$ Felder in eine Richtung expandieren, danach kann es nur noch $L/2 - 1$ Felder in die andere Richtung expandieren, womit es keine der dortigen Threshold-Gadget Quellen markieren kann. Damit ist ausgeschlossen, dass eine Variable sowohl wahr als auch falsch sein kann. Das Variable-Gadget ist dazu in der Lage, Threshold-Gadget mit einer Entfernung von L Feldern (nach links oder rechts) zu aktivieren.

Die Bounding Box ist wieder drei Felder hoch, da alle Zahlfelder die Zahl eins haben. Außerdem können maximal $L/2$ Threshold-Gadgets pro Seite aktiviert werden. Unter der Annahme, dass alle diese zuerst expandiert wurden, spannt sich die Bounding Box $3L/2$ Felder neben den nebeneinanderliegenden horizontalen Zahlfeldern aus und ist damit $4L$ lang (vergleiche Abbildung 3.4). Dieses worst case Szenario tritt ein für den Fall dass $L = 2$.

Clause-Gadget

Durch das Zusammenfügen von Variable-, Wire- und OR-Gadget können wir ein Clause-Gadget bilden, welches eine Klausel unserer RPM-3SAT-Formel darstellt, siehe dazu Abbildung 3.5. Jede Klausel enthält entweder nur positive oder negative Variablen, daher kann eine positive Klausel nach rechts über die Variable-Gadgets geleitet werden und eine negative darunter.

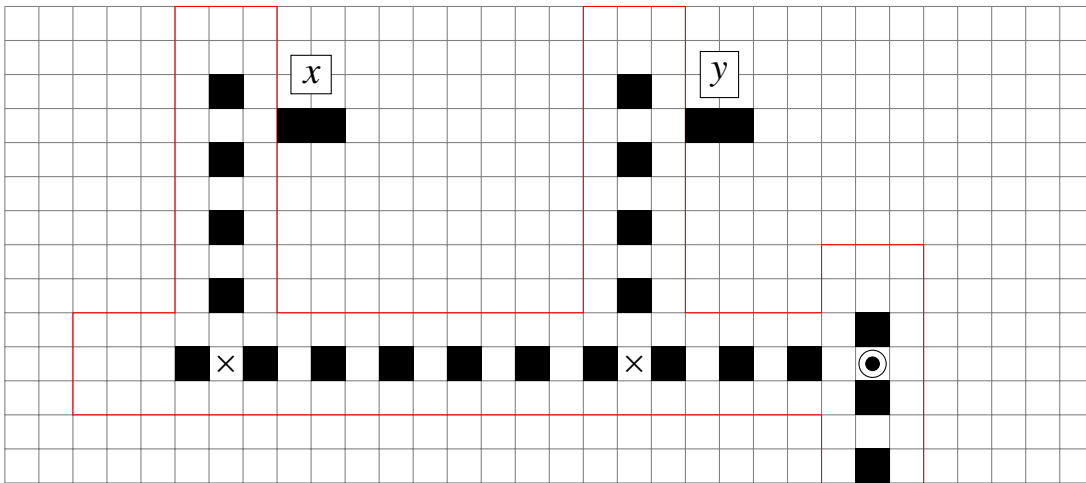


Abbildung 3.5: Ein Clause-Gadget der Klausel $(\bar{x} \vee \bar{y})$. Die Variable-Gadgets können Wire-Gadgets jeweils links neben sich aktivieren. Deren Ziele sind Quellen in dem horizontalen OR-Gadget. Die Kombination dieser Wire- und des OR-Gadgets ist das eigentliche Clause-Gadget. Dieses kann ein Wire-Gadget aktivieren, welches das Signal weiterleitet.

Die Bounding Box ergibt sich nach den bereits oben besprochenen Regeln, wie in Abbildung 3.5 dargestellt.

Crossover-Gadget

Bei der Konstruktion des Puzzles ist es nötig, dass sich Threshold-Gadgets auch kreuzen können, da die Signale der Clause-Gadgets zu einem AND-ALL-Gadget weitergeleitet werden müssen. Dabei können sich dann diese Wire-Gadgets, die ein Klausel-Signal transportieren und Variable-Gadgets kreuzen.

Dafür wird das Crossover-Gadget eingefügt, vergleiche Abbildung 3.6. Hierbei sind zwei Threshold-Gadgets so angeordnet, dass ein unmarkiertes Feld am Kreuzungspunkt liegt. Die

Bounding Box wird dabei um ein Feld auf der Seite mit dem Ziel und auf der gegenüberliegenden Seite verlängert. In der Abbildung gehen wir davon aus, dass das horizontale Gadget zuerst aktiviert werden muss, da sonst das Ziel vom vertikalen Gadget nicht markiert werden kann, weil es nur zwei Quellen gibt und sein Ziel drei Felder vom letzten Zahlfeld entfernt liegt.

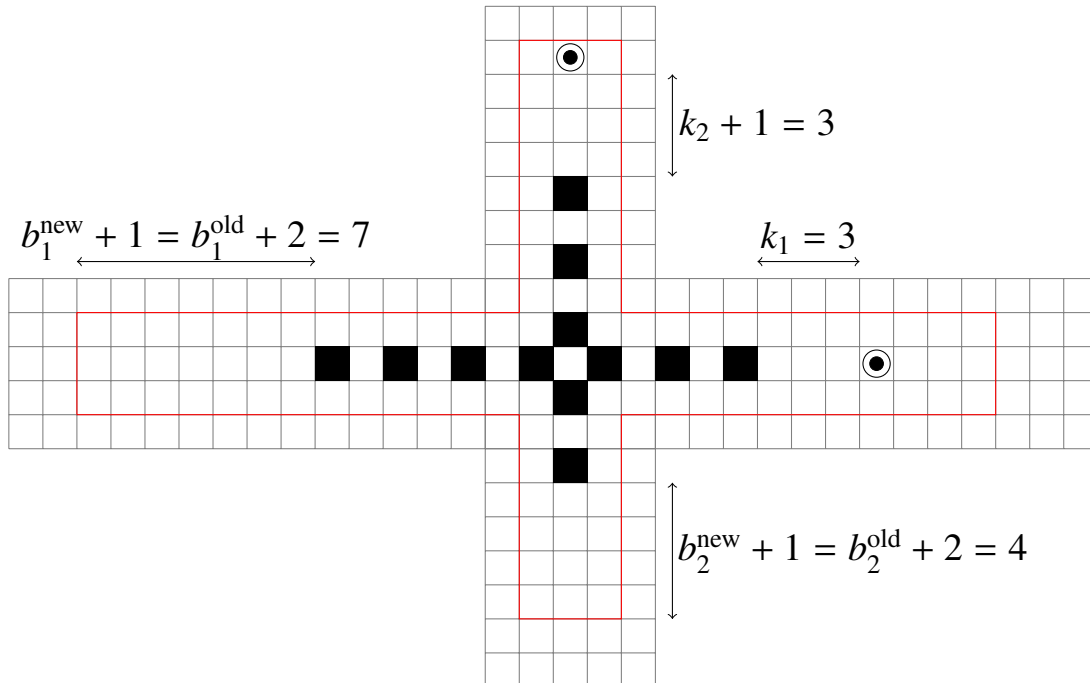


Abbildung 3.6: Ein Crossover-Gadget, bei dem das horizontale Gadget zuerst aktiviert werden soll, wodurch sich dessen Ziel nicht ändert. Das vertikale Gadget kann nur aktiviert werden, wenn das horizontale bereits aktiviert wurde, da es selber nur zwei Quellen hat und das Ziel drei Felder entfernt liegt.

Das vertikale Gadget ist so konstruiert, dass mehrfache Kreuzungen keine ungewollte Markierung des Ziels ermöglichen. Beim horizontalen Gadget, das ein Klausel-Signal leitet, kann es zu Problemen kommen, wenn Crossover-Gadgets in der falschen Reihenfolge aktiviert werden. Dieses Problem lässt sich mit dem sogenannten *Thick-Wire* lösen. Dafür wird gezählt, an wie vielen Crossover-Gadgets das horizontale Gadget beteiligt ist, wir nennen diese Zahl g und dann das zugrunde liegende Clause-Gadget so konstruiert, dass $> g$ Threshold-Gadgets aktiviert werden müssen, um eine Variable zu aktivieren, siehe dazu Abbildung 3.7.

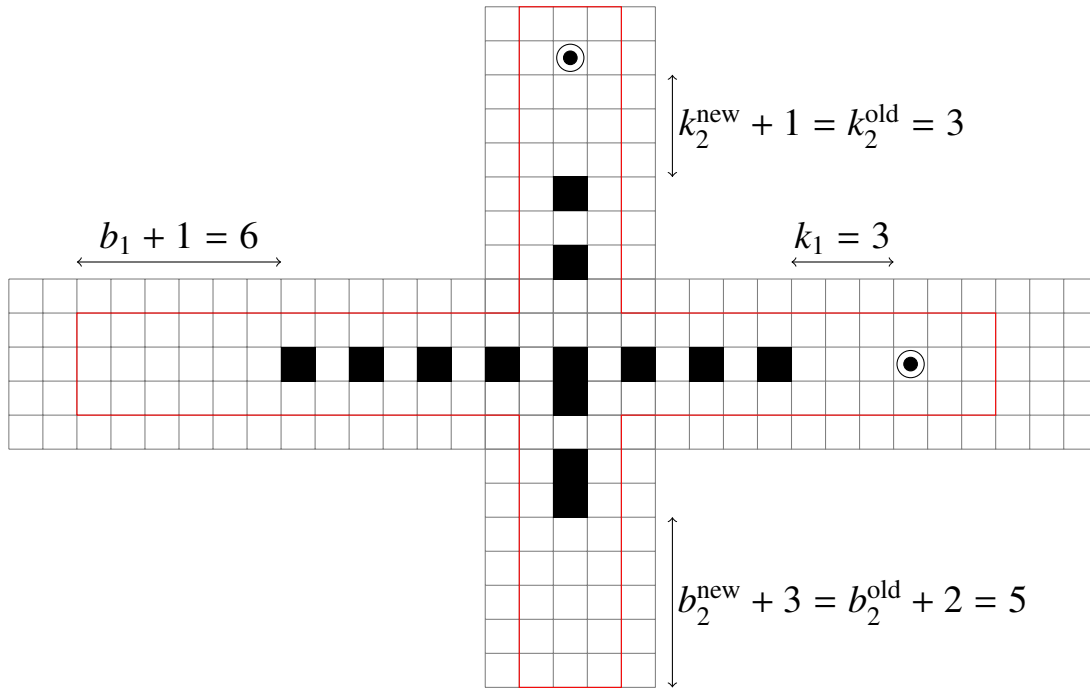


Abbildung 3.9: Ein neues Crossover-Gadget, bei dem das Kreuzungsfeld ein Zahlfeld ist. Das horizontale Gadget ist eine Signalweiterleitung eines Clause-Gadgets.

Dadurch ist es nicht mehr möglich, dass das vertikale, das horizontale Gadget beeinflusst. Der umgekehrte Fall ist aber immer noch möglich.

Wir werden im Folgenden das Puzzle so konstruieren, dass das horizontale Gadget eines neuen Crossover-Gadget immer die Signalweiterleitung eines Clause-Gadgets ist. Auch zu beachten ist, dass jedes zweite neue Crossover-Gadget in Richtung vom Ziel weg, ein weiteres Zahlfeld über dem Kreuzungspunkt hat und dafür das Shift-Gadget entfernt.

Sei n die Anzahl der Variablen und m die Anzahl an Klauseln, dann ist m durch n begrenzt, da es eine RPM-3SAT-Repräsentation der Formel gibt. Die Idee, um m zu bestimmen, ist die, dass wir versuchen, die Klauseln so dicht zu stapeln wie möglich. Dazu sei die Variable mit dem niedrigsten Index aller Klauseln eins. Außerdem hat die Klausel, die von allen anderen eingeschachtelt wird, die Variablen mit Index eins bis drei. Der größte Index in einer Klausel ist der mittlere Index der Klausel, von der sie eingeschachtelt wird. Der größte Index einer jeden Klausel ist um eins größer als ihr mittlerer Index. Damit ergibt sich für den positiven Teil unserer Formel ϕ_+ ,

$$\phi_+ = \bigwedge_{i=2}^{n-1} (x_1 \vee x_i \vee x_{i+1}). \quad (3.1)$$

Damit und dem negativen Teil unserer Formel ergibt sich $m \leq 2n - 4$.

Es sind aber auch Klauseln mit zwei Variablen möglich, für die die obige Argumentation nicht genügt. Allerdings gibt es in einer Menge $\{1, \dots, n\}$ nur $\binom{n}{2}$ verschiedene, zweielementige Teilmengen. Damit gilt sicher: $m \in \mathcal{O}(n^2)$.

Sei nun $C(n)$ die maximale Anzahl an Klauseln in einer RPM-3SAT-Formel, in der alle

Algorithmus 1 : Länge der Variable-Gadgets

Data : RPM-3SAT Formel

Result : 3 Array der Länge n

```
1 array ← Array der Länge  $n$ ;  
2 pos ← Array der Länge  $n$  ;           /* Häufigkeit der Variable  $x_i$  */  
3 neg ← Array der Länge  $n$  ;           /* Häufigkeit der Variable  $\bar{x}_i$  */  
4 for  $i \in \{0, \dots, n - 1\}$  do  
5   | foreach klausel in Formel do  
6   |   | if klausel enthält  $x_i$  then  
7   |   |   |  $pos[i] \leftarrow pos[i] + 1$ ;  
8   |   |   | end  
9   |   |   | if klausel enthält  $\bar{x}_i$  then  
10  |   |   |   |  $neg[i] \leftarrow neg[i] + 1$ ;  
11  |   |   |   | end  
12  |   |   | end  
13  |   | if  $pos[i] > neg[i]$  then  
14  |   |   |  $array[i] \leftarrow pos[i]$ ;  
15  |   |   | else  
16  |   |   |   |  $array[i] \leftarrow neg[i]$ ;  
17  |   |   |   | end  
18 end
```

Um die Laufzeit zu analysieren, nutzen wir im Folgenden die O -Notation. Für die Laufzeitanalyse bedeutet das:

- Zeile 1 - 3: Das Anlegen von Arrays läuft in $O(1)$, da diese nicht vorinitialisiert sind.
- Zeile 4: Für alle n Variablen wird die Schleife einmal durchlaufen, damit ergibt sich $O(n)$.
- Zeile 5: Es werden alle Klauseln einmal durchlaufen, das liegt in $O(n)$.
- Zeile 6, 9: In einer Klausel wird verglichen, ob eine Variable enthalten ist, dafür sind maximal drei Vergleiche nötig und somit ergibt sich $O(1)$.
- Zeile 7, 10: Das Inkrementieren einer Zahl läuft in $O(1)$.
- Zeile 13: Der Vergleich zweier Zahlen verläuft in $O(1)$.
- Zeile 14, 16: Das Auslesen und Schreiben eines Werts in ein Array erfolgt in konstanter Zeit ($O(1)$).

Insgesamt ergibt sich damit eine Laufzeit von $\mathcal{O}(n^2)$ für Algorithmus 1. Dieser Schritt läuft daher in Polynomialzeit.

Als nächstes wird untersucht, wie die Klauseln geschachtelt werden müssen, damit die Signale von Variable-Gadgets keine OR-Gadgets schneiden. Dafür wird geprüft, welche Variable mit dem kleinsten Index einer jeden Klausel den höchsten Index unter allen Klauseln hat.

Wenn sich ein Gleichstand zwischen Klauseln ergibt, dann wird geprüft, welche Variable mit dem höchsten Index innerhalb der Klausel am niedrigsten unter den Klauseln ist. Diese wird dann vor die andere in die Liste geschrieben. Ist aber auch hier ein Gleichstand, wird die Klausel mit drei Variablen vor die Klausel mit zwei Variablen geschrieben, vergleiche Algorithmus 2. Dieser sortiert beispielhaft nur die positiven Klauseln, negative würden aber vergleichbar sortiert werden.

Algorithmus 2 : Sortieren der Klauseln

Data : RPM-3SAT Formel**Result :** Liste mit sortierten Klauseln

```
1 list ← ();
2 foreach pKlausel in Formel do
3   if list ist leer then
4     füge pKlausel zu list hin zu;
5   else
6     foreach elem in list do
7       if elem.lowest < klausel.lowest then
8         insert pKlausel here;
9         continue;
10      end
11      if elem.lowest = pKlausel.lowest und elem.highest > pKlausel.highest
12      then
13        insert pKlausel here;
14        continue;
15      end
16      if elem.lowest = pKlausel.lowest und elem.highest = pKlausel.highest
17      und pKlausel contains 3 Variablen then
18        insert pKlausel here;
19        continue;
20      end
21    end
22  end
23  füge pKlausel zu list hin zu;
24 end
```

Zur Laufzeit von Algorithmus 2:

- Zeile 1: Das Anlegen einer Liste erfolgt in konstanter Zeit und damit $O(1)$.
- Zeile 2: Die Schleife wird für jede Klausel einmal durchlaufen ($O(n)$).
- Zeile 3: Ob eine Liste leer ist oder nicht, lässt sich in konstanter Zeit prüfen, also $O(1)$.
- Zeile 4, 8, 12, 16, 20: Das Einfügen eines Objekts in eine Liste liegt in $O(1)$.
- Zeile 6: Im worst case muss die zu untersuchende Klausel mit allen anderen Klauseln verglichen werden, womit sich $O(n)$ hierfür ergibt.
- Zeile 7, 11, 15: Alle Vergleiche lassen sich in konstanter Zeit durchführen, also $O(1)$.

Für jede Klausel ergibt sich ein neuer worst case in Zeile 6. Beispielsweise muss die erste untersuchte Klausel mit keiner weiteren verglichen werden. Aber die letzte mit $m - 1$. Insgesamt ergibt sich damit,

$$\sum_{i=0}^{m-1} i = \frac{(m-1)^2}{2} \in O(n^2). \quad (3.5)$$

Es handelt sich also auch hierbei um einen in Polynomialzeit laufenden Algorithmus.

Mit den Ergebnissen aus Algorithmus 1 werden die Variable-, sowie die Threshold-Gadgets, die deren Signale weiterleiten, angelegt. Diese Threshold-Gadgets werden dann nach den Ergebnissen von Algorithmus 2 zu Clause-Gadgets zusammengefasst. Dabei wird vom OR-Gadget in Richtung des Variable-Gadgets gebaut und immer dann ein neues Crossover-Gadget eingefügt, wenn bereits ein Zahlfeld auf dem Weg liegt. Alle Signale von Clause-Gadgets werden in einem großen AND-Gadget, dem AND-ALL-Gadget, zusammengefasst und dessen Ziel wird als Zielfeld festgelegt. Als Input verwendet diese Funktion die Liste *list*, die der Output von Algorithmus 2 ist und die drei Arrays, die von Algorithmus 1 erzeugt werden. Siehe dazu Algorithmus 3.

Algorithmus 3 : ZHED bilden

Data : RPM-3SAT Formel *list*, Integer Array *array*, Integer Array *pos*, Integer Array *neg*

Result : ZHED Puzzle

```
1  $i \leftarrow 0$ ;  
2 foreach  $j$  in array do  
3   erzeuge Variable-Gadget mit  $L \leftarrow 2 + 16(j - 1)$  an der Stelle  $(\frac{3}{2}L + i, 0)$ ;  
4   erzeuge  $pos[i]$  Theshold-Gadgets auf der rechten Seite mit jeweils fünf Feldern  
   Abstand und  $L/2$  Feldern Abstand zum Variable-Gadget;  
5   erzeuge  $neg[i]$  Theshold-Gadgets auf der linken Seite mit jeweils fünf Feldern  
   Abstand und  $L/2$  Feldern Abstand zum Variable-Gadget;  
6    $i \leftarrow i + 4L$ ;  
7 end  
8  $k, k_{pos}, k_{neg} \leftarrow 0$ ;  
9 foreach klausel in list do  
10  if klausel is positiv then  
11     $k_{pos} \leftarrow k_{pos} + 1$ ;  
12     $k \leftarrow k_{pos} - 1$ ;  
13  else  
14     $k_{neg} \leftarrow k_{neg} + 1$ ;  
15     $k \leftarrow k_{neg} - 1$ ;  
16  end  
17  verlängere rechtestes mögliches Threshold-Gadget von klausel.highest um  $7 + 8k$   
   Felder;  
18  verlängere letztes mögliches Threshold-Gadget von klausel.mid um  $7 + 8k$  Felder;  
19  verlängere linkestes mögliches Threshold-Gadget von klausel.lowest um  $7 + 8k$   
   Felder;  
20  verbinde Threshold-Gadgets zu Clause-Gadget;  
21  verlängere Signal des Clause-Gadgets bis  $i + 5$ ;  
22 end  
23 erzeuge AND-ALL-Gadget bei  $i + 7$ ;
```

Für Algorithmus 3 ergibt sich dann folgende Laufzeitanalyse:

- Zeile 1, 6, 8, 11 – 15: Der Laufzahl einen Wert zuzuordnen, erfolgt in konstanter Zeit, also $O(1)$.
- Zeile 2: Es gibt n Element in *array*, damit liegt dieser Schritt in $O(n)$.
- Zeile 3 – 5: Im worst case ist eine Variable an m Klauseln beteiligt, also $O(n)$.

- Zeile 9: Da jede Klausel einmal durchlaufen wird liegt dies in $O(n)$.
- Zeile 10: Zu prüfen, ob eine Klausel positiv ist erfolgt in $O(1)$.
- Zeile 17 – 19: Da k gegen m läuft und die Länge des Puzzles von m abhängt, erfolgen diese Schritte im worst case in $O(n^2)$.
- Zeile 20: Eine Klausel kann die Variablen x_1 und x_n enthalten und da die Länge des Puzzles von m abhängt, liegt dieser Schritt im worst case in $O(n)$.
- Zeile 21: Im worst case ist die Variable mit dem höchsten Index einer Klausel x_3 und da i von m abhängt, liegt dieser Schritt in $O(n)$.
- Zeile 23: Das AND-ALL-Gadget hat m Quellen. Es zu bilden liegt daher in $O(n)$.

Insgesamt liegt die Zeitkomplexität der ersten for-Schleife nur bei $O(n)$, da hier über alle Variablen iteriert wird, aber dabei jede Klausel maximal dreimal (für jede ihrer Variablen einmal) aufgerufen wird. Entsprechend liegt dies in $O(m + n)$ und daher gilt obige Aussage.

Die zweite for-Schleife läuft über alle Klauseln und verlängert dann Threshold-Gadgets um bis zu $7 + 8m$ Felder. Ähnlich wie in Gleichung 3.5 erhalten wir für die Verlängerung der Threshold-Gadgets eine Mitgliedschaft in $O(n^2)$. Zeile 20, 21 und 23 liegen selbst in $O(n)$ und ändern somit für die gesamte Zeitkomplexität nichts.

Allgemein läuft die Reduktion damit in $O(n^2)$, also in Polynomialzeit.

3.3 Korrektheit

Zur Bounding Box, wie sie in [12] beschrieben ist, ist zu sagen, dass diese etwas zu groß ist, da ein Zahlfeld nur Felder markieren kann, die es über eine Grade erreicht. Damit ergibt sich eine Bounding Box, wie sie in Abbildung 3.11 zu sehen ist.

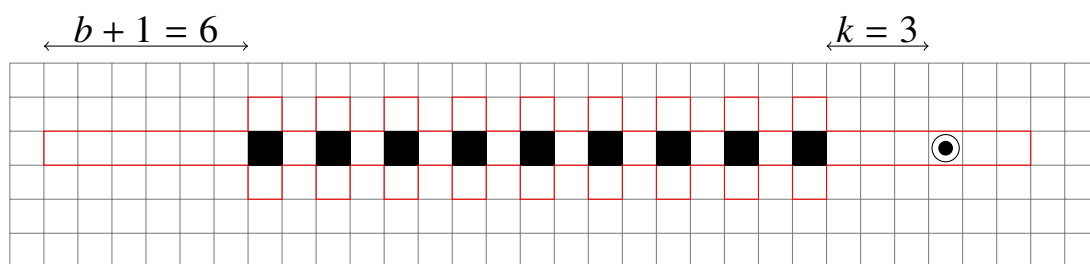


Abbildung 3.11: Darstellung eines Threshold-Gadget, mit einer minimalen Bounding Box. Es gilt $b = 5$ und $k = 3$.

Damit ergibt sich auch, dass die Bounding Boxen in Abbildung 3.8 sich nicht überschneiden, auch wenn das, zu einer positiven Klausel gehörende Threshold-Gadget ein Shift-Gadget enthielte.

Die Ausnahme diskutieren wir hier. Die Bounding Box eines Gadgets wächst um jedes Crossover-Gadget, an dem es beteiligt ist in positive und negative y -Richtung.

Horizontale Gadgets liegen acht Felder auseinander, nach einem Turn- in einem Clause-Gadget kann dieser Abstand auf fünf Felder schrumpfen. Nach der alten Definition der Bounding Box wären dann drei Crossover-Gadgets nötig, damit sich die Bounding Boxen zweier Gadgets überschneiden. Zwischen positiven und negativem Teil unseres Puzzles sind durch das mögliche Shift-Gadget sogar nur ein Crossover-Gadgets nötig, damit sich zwei Bounding Boxen überschneiden. Allerdings ist das Puzzle so konstruiert, dass die Zahlfelder des OR-Gadgets der einen Seite in der gleichen Spalte sind, wie die Zahlfelder der Threshold-Gadgets der anderen Seite. Dadurch können sich diese Gadgets nicht gegenseitig aktivieren.

Sollte durch diese Überschneidung ein neuer Lösungsweg entstehen, ist das im Falle einer lösbaren RPM-3SAT-Formel kein Problem, da das entstehende ZHED Puzzle dann sicher eine Lösung hat. Für den Fall, dass das Puzzle nicht lösbar sein soll, brauchen wir eine andere Argumentation.

Aus der Konstruktion des neuen Crossover-Gadgets ist bekannt, dass das horizontale Gadget die Signalleitung eines Clause-Gadgets ist. Damit das vertikale Gadget in y -Richtung mehr Felder markiert als vorgesehen, muss das Feld am Kreuzungspunkt des Crossover-Gadgets in y -Richtung expandiert werden. Das führt dazu, dass das Signal des Clause-Gadgets nicht beim AND-ALL-Gadget ankommt, was dazu führt, dass das Puzzle auf diese Weise nicht zu lösen ist.

Wir betrachten nun ein Puzzle, bei dem mindestens ein Clause-Gadget nicht aktiviert werden kann und nehmen an, dass durch das fehlerhafte Expandieren dreier Crossover-Gadgets das Clause-Gadget doch aktiviert wird. Damit wissen wir, dass jetzt drei Signale von Clause-Gadgets nicht mehr beim AND-ALL-Gadget Quellfelder markieren können. Für jedes dieser Signale werden mindestens drei Crossover-Gadgets, die falsch expandiert wurden, benötigt, um sie wieder zu aktivieren. Insgesamt steigt der Bedarf an falsch ausgeführten Crossover-Gadgets mit 3^k , $k \in \mathbb{N}$, da die Anzahl der Crossover-Gadgets aber durch die Zahl der Klauseln limitiert ist, kann es von ihnen nicht genügend geben, um ein Clause-Gadget oder sein Signal zu aktivieren und gleichzeitig das Puzzle zu lösen.

Um ein konstruiertes Puzzle zu lösen, müssen erst die Variable-Gadgets aktiviert werden. Deren Signale markieren Quellen in den dazugehörigen OR-Gadgets, welche nach Füllung ihrer Quellen in Richtung ihrer Ziele expandiert werden. Damit können dann die Turn-Gadgets ihrer Ziele markieren. Diese Ziele sind Quellen im Gadget, das das Signal des Clause-Gadgets zum AND-ALL-Gadget leitet. Zuletzt, nach dem die Quellen im AND-ALL-Gadget markiert wurden, wird dieses in Richtung seines Ziels expandiert.

Damit kommen wir zum eigentlichen Korrektheitsbeweis:

Erfüllbare Formel \Rightarrow Lösbares Puzzle

Angenommen, es existiert eine Belegung, sodass die zugrunde liegende RPM-3SAT-Formel erfüllt wird. Dann expandiere alle Variable-Gadgets nach rechts, wenn die zugehörige Variable 1 ist und nach links andernfalls. Dann expandieren wir alle vertikalen Gadgets in Richtung ihrer Ziele unter Berücksichtigung der Crossover-Gadgets. Anschließend expandieren wir alle OR-Gadgets in Richtung ihrer Ziele. Da alle Klauseln bei dieser Belegung erfüllt sind, wird auch mindestens eine Quelle in zugehörigen OR-Gadgets markiert, womit deren Ziele gefüllt werden können. Dann aktivieren wir die Turn-Gadgets und leiten danach die Signale in das AND-ALL-Gadget auf der rechten Seite. Alle Quellen dieses AND-ALL-Gadgets werden dadurch markiert. Wir expandieren abschließend alle Zahlfelder des AND-ALL-Gadgets in Richtung des Ziels und markieren dieses dadurch. Damit haben wir eine Lösung für das Puzzle gefunden.

Erfüllbare Formel \Leftarrow Lösbares Puzzle

Wir nehmen an, das Spiel ist lösbar. Das AND-ALL-Gadget kann das Ziel nur markieren, wenn alle seine Quellen aktiviert wurden, da das Puzzle lösbar ist, ist dies der Fall. Alle diese Quellen können nur von Clause-Gadgets aktiviert werden. Dafür muss in jedem Clause-Gadget mindestens eine Quelle im OR-Gadget markiert sein. Diese Quellen können nur aktiviert werden, wenn ihr Threshold-Gadget von einem Variable-Gadget aktiviert wurde, in dem mehr als die Hälfte deren Zahlfelder in x -Richtung expandiert wurden.

Für jede Variable in der Formel vergleichen wir, in welche Richtung ihr zugehöriges Gadget expandiert wurde. Wenn es nach rechts expandiert wurde, setzen wir die Variable auf 1 und auf 0 sonst. Diese Belegung sorgt dafür, dass alle Klauseln erfüllt werden, da alle Clause-Gadgets aktiviert sind und damit ist die Formel mit dieser Belegung insgesamt erfüllt.

4 Programmierung

In diesem Teil geht es um die Implementierung des oben vorgestellten Pseudocodes in Python 3.10.6 [9]. Dabei bietet Python einige Vorteile, wie die Ähnlichkeit zum Pseudocode und eine umfangreiche Standardbibliothek. Außerdem ist Python weit verbreitet und Tutorials zu diversen Themen stehen zur Verfügung.

4.1 Packages

Für die Implementierung wurden die folgenden Packages verwendet:

itertools

Für die Bestimmung einer gültigen Belegung der RPM-3SAT-Formel wird das Package *itertools* [10] verwendet. Die Funktion `product` gibt das kartesische Produkt von Mengen zurück, die der Funktion als Eingabe gegeben wurden. Sei nun n die Anzahl der Variablen unserer Formel. Dann erhalten wir, wenn wir n mal die Menge `{True, False}` übergeben, alle möglichen Belegungen für unsere Variablen.

pygame

Für die graphische Darstellung der Ergebnisse wird *pygame* [8] verwendet. Damit ist es sehr leicht möglich eine graphische Darstellung des Ergebnisses unserer Reduktion anzuzeigen. Dazu bietet es Funktionen an, mit denen ein User oder eine Userin mit dem Programm interagieren kann, indem *pygame* Tastatur und Maus einbindet.

sys

Bei fehlerhaften Eingaben wird die Ausführung des Programms mit `sys.exit(1)` abgebrochen. Fehlerhafte Eingaben sind dabei Klauseln, die dreimal dieselbe Variable enthalten und RPM-3SAT-Formeln, die Indizes überspringen. Das heißt, wenn der höchste Index h ist, ein Index $< h$ nicht in der Formel auftaucht. Dabei gilt $h \in \mathbb{N} \setminus \{0\}$.

4.2 Implementierung

Die graphische Oberfläche wird aus einem zweidimensionalen Boolean Array aufgebaut. Immer dann, wenn ein Eintrag True ist, wird ein schwarzes Kästchen (Zahlfeld) erzeugt, sonst wird ein weißes Kästchen (unmarkiertes Feld) erzeugt, außerdem wird das Zielfeld als rotes Kästchen dargestellt. Das dabei erzeugte Spielfeld ist im allgemeinen nicht quadratisch, wie wir es in Abschnitt 2.2 gefordert hatten. Um das Feld in eine quadratische Form zu bekommen, müssten nur Reihen von unmarkierten Feldern ergänzt werden. Da dies die Übersichtlichkeit nicht verbessert und auch sonst keinen funktionalen Mehrwert liefert, wurde es weggelassen.

In der GUI, siehe Abbildung 4.1, kann durch die Verwendung der Pfeiltasten und durch einen Linksklick und Bewegung der Maus, während die linke Maustaste gedrückt bleibt, gescrollt werden. Auch Zoomen ist durch Verwendung des Mausekkrads möglich. Des weiteren können die Bounding Boxen der einzelnen Gadgets durch Anklicken (links Klick) sichtbar gemacht und wieder versteckt werden (rechts Klick).

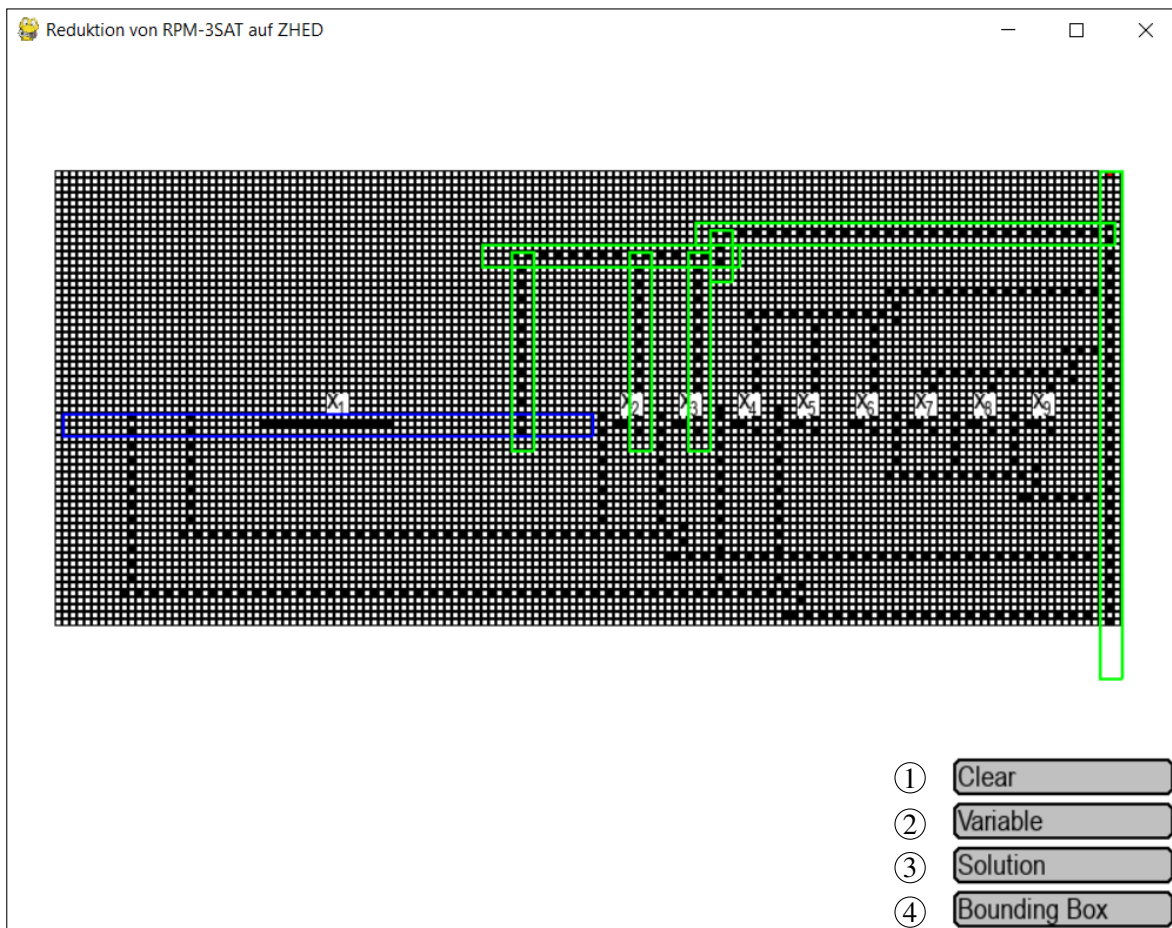


Abbildung 4.1: Die GUI des Programms zur Reduktion von RPM-3SAT auf ZHED. Die Bounding Boxen des Variable-Gadgets von x_1 , sowie des Clause-Gadgets der Klausel ($x_1 \vee x_2 \vee x_3$) und das AND-ALL-Gadget sind markiert. Außerdem werden die Namen der Variablen angezeigt.

Die Buttons im unteren rechten Bereich des Fensters ermöglichen:

- ① Alle angezeigten Bounding Boxen wieder zu verstecken,
- ② Die Namen der Variablen anzuzeigen, die ein Variable-Gadget repräsentiert,
- ③ Die Bounding Boxen aller Gadgets anzuzeigen, die bei einer Lösung aktiviert werden würden,
- ④ Alle Bounding Boxen gleichzeitig sichtbar zu machen.

Eine Klausel wird im Programm durch ein Element des Typs *Clause* dargestellt. Diese werden durch drei Integerwerte (Index der Variablen), von denen mindestens einer verschieden sein muss, und einen Boolean, der angibt, ob die Klausel positiv oder negativ ist, charakterisiert. Hätten alle Variablen einer Klausel denselben Index, könnte diese Variable so gesetzt werden, dass die Klausel erfüllt wird. Dann könnten alle Klauseln, die durch die Variable erfüllt wurden und jedes weiteres Vorkommen der Variable aus der Formel gestrichen werden.

Die RPM-3SAT-Formel wird im Programm durch eine Liste von *Clauses* dargestellt. Es ist zu beachten, dass alle Indizes von eins bis hin zum höchsten mindestens einmal in der Formel vorhanden sein müssen. Alternativ hätten die Variablen auch umbenannt werden können, um die Bedingung zu erfüllen, da dies aber nichts mit der Reduktion zu tun hat, wurde es nicht implementiert.

Das Programm ermittelt zuerst den höchsten Index und prüft die Formel auf Korrektheit nach den obigen Bedingungen. Auch wird mittels *itertools.product* eine erfüllende Belegung gefunden, sollte eine existieren. Anschließend wird Algorithmus 1 ausgeführt, um die Länge der Variable-Gadgets zu bestimmen. Diese wird gefolgt von Algorithmus 2, für die Sortierung der Klauseln.

Für den Aufbau des Spielfeldes ist es wichtig, dessen Länge (*length*), Höhe (*height*) und y-Koordinate der Variable-Gadgets (*v_height*) zu berechnen. Für diese ergibt sich:

$$\begin{aligned}
 length &= \underbrace{7}_{\text{Abstand zum AND-ALL-Gadget}} + \sum_{i \in array} \underbrace{4 \cdot (16i - 14)}_{\text{Länge der Variable-Gadgets}} + \underbrace{2}_{\text{Ein Feld Abstand nach links und rechts}}, \\
 height &= \underbrace{8 \cdot |rank_p| - 1}_{\text{Abstand des höchsten OR-Gadgets zu Variable-Gadgets}} + \underbrace{8 \cdot |rank_n| - 1}_{\text{niedrigstes OR-Gadget}} + \underbrace{6}_{\text{zwei Turn-Gadgets}} + \underbrace{1}_{\text{Varibale-Gadget}} + \underbrace{1}_{\text{Zielfeld}} \\
 &+ k_{\text{AND-ALL}} + \underbrace{2}_{\text{ein Zahlfeld über und unter Clause-Gadget Signal}}, \\
 v_height &= height - \underbrace{1}_{\text{Varibale-Gadget}} - \underbrace{3}_{\text{Turn-Gadget}} - \underbrace{8 \cdot |rank_n| - 1}_{\text{niedrigstes OR-Gadget}} - \underbrace{1}_{\text{Zahlfeld unter Clause-Gadget Signal}}.
 \end{aligned}$$

Dabei ist mit *array* das entsprechende Ergebnis aus Algorithmus 1 gemeint. Die Werte $|rank_p|$ und $|rank_n|$ sind die Mächtigkeiten der Ergebnisse aus Algorithmus 2 für positive Klauseln bzw. negative Klauseln. Außerdem ist $k_{AND-ALL}$ der k -Wert des AND-ALL-Gadgets.

Die restliche Bildung des Spielfelds entspricht im wesentlichen Algorithmus 3, außer dass die Threshold-Gadgets, die von den Variable-Gadgets aktiviert werden können, erst bei Bedarf erzeugt werden. Außerdem wird für jedes Gadget, dessen Bounding Box, wie in Abschnitt 3.1 beschrieben, mit berechnet.

Abschließend untersuchen wir die neuen Crossover-Gadgets etwas genauer. Siehe dazu Abbildung 4.2, in der drei Signalleitungen von einem Variable-Gadget dargestellt sind. Für Signalleitungen muss gespeichert werden, an wie vielen Crossover-Gadgets sie beteiligt sind, um deren Bounding Boxen zu bestimmen.

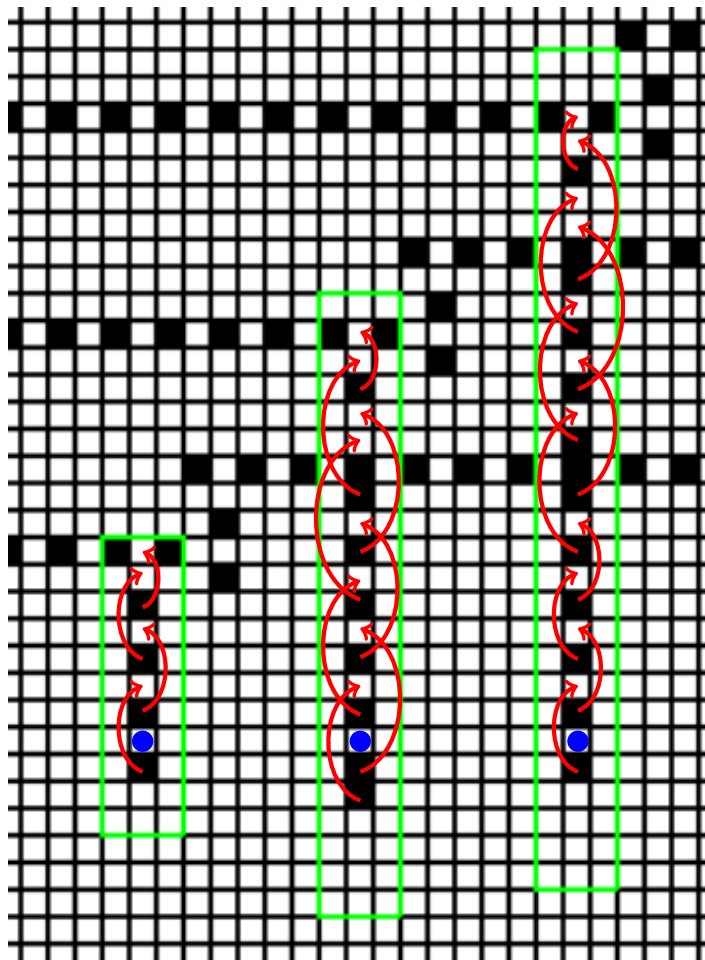


Abbildung 4.2: Signalleitungen von einem Variable-Gadget, von denen eins an keinem Crossover-Gadget beteiligt ist, ein anderes an einem und das letzte an zweien. Die Bounding Boxen der Signalleitungen ist in Grün dargestellt. Die Felder mit den blauen Punkten sind die Felder, die von dem Variable-Gadget markiert werden können. Die roten Pfeile zeigen, welche Felder von den Zahlfeldern markiert werden.

Wie in der Abbildung zu sehen ist, können die Signalleitungen ihr Zielfeld nur markieren, wenn das Variable-Gadget die Felder mit den blauen Punkten vorher markierte.

Eine Signalleitung, die an einer ungeraden Anzahl an Crossover-Gadgets beteiligt ist, enthält ein Shift-Gadget (vgl. mittlere Signalleitung in Abbildung 4.2). Die, die an einer geraden Anzahl an Crossover-Gadgets beteiligt sind, nutzen aus, dass durch diese Crossover-Gadgets ihr Ziel um eine gerade Anzahl an Felder verschoben wird. Das Weglassen entsprechend vieler Zahlfelds sorgt dann dafür, dass das Ziel da bleibt, wo es liegen soll (vgl. rechte Signalleitung in Abbildung 4.2). Ein drittes Crossover-Gadget unter den drei verbundenen Zahlfeldern des vorherigen Crossover-Gadgets hätte dann wieder nur zwei verbunden Zahlfelder. Ein viertes dann wieder drei und so weiter. Wichtig ist, dass das oberste Crossover-Gadget nur zwei verbunden Zahlfelder besitzt, da sonst das Ziel auch markiert werden könnte, wenn das Variable-Gadget in die andere Richtung expandiert würde.

5 Zusammenfassung und Ausblick

Zusammenfassung

In dieser Arbeit haben wir das Puzzlespiel ZHED vorgestellt und gesehen, dass es NP-vollständig ist, auch wenn alle Zahlfelder die Zahl eins enthalten. Es gilt jedoch allgemein, dass ZHED NP-vollständig ist, auch für Zahlfelder, die ungleich eins sind [12].

Die für die, von Saha und Demaine entworfene, Reduktion nötigen Gadgets wurden vorgestellt und analysiert. Das wichtigste von ihnen ist das Threshold-Gadget, auf dem zum Beispiel das AND- und das OR-Gadget basieren. Die Ausnahme ist das Variable-Gadget. Diese Gadgets lassen sich dann zu einem ZHED Spiel zusammenfügen, das genau dann lösbar ist, wenn die zugrundeliegende RPM-3SAT-Formel erfüllbar ist.

Für die in dieser Arbeit betrachtete Reduktion wurden das Crossover-Gadget neu definiert, sodass immer eine Signalleitung eines Clause-Gadgets an ihnen beteiligt ist, und auf Thick-Wire verzichtet. Diese waren in der ursprünglichen Reduktion nötig, damit sich Bounding Boxen unterschiedlicher Gadgets sich nicht überschneiden. Es wurde daher gezeigt, dass die Reduktion auch dann noch korrekt ist, wenn sich Bounding Boxen überschneiden. Voraussetzung dafür ist, dass Crossover-Gadgets die Bounding Boxen so wachsen lassen, dass sie sich überschneiden.

Auf dieser Grundlage wurde Pseudocode vorgestellt, der die Reduktion beschreibt und gezeigt, dass dieser in Polynomialzeit läuft. Dafür wurde auch bewiesen, dass die Anzahl der Klauseln linear von der Anzahl der Variablen abhängt.

Der Pseudocode ist so aufgeteilt: In Algorithmus 1 werden die Längen der Variable-Gadgets bestimmt. Algorithmus 2 sortiert die Klauseln so, dass sich keine ungewollten Überschneidungen von Gadgets ergeben. Abschließend werden diese Ergebnisse in Algorithmus 3 genutzt um das ZHED Spiel zu konstruieren.

Es erfolgte eine Umsetzung des Pseudocodes in Python. Das dabei konstruierte ZHED Spielfeld wird graphisch dargestellt. In dieser Darstellung ist es möglich sich die Namen der Variablen anzeigen zu lassen. Außerdem kann durch klicken auf einzelne Gadgets deren Bounding Box angezeigt werden. Ein weiteres Funktion im Programm ist es, die Bounding Boxen aller Gadgets anzuzeigen, die bei einer Lösung des Spiels aktiviert würden. Voraussetzung dafür ist, dass eine solche Lösung existiert.

Ausblick

Die Reduktion könnte optimiert werden, indem die Größe der Bounding Box von Variable-Gadgets verkleinert wird. Für jedes weitere Vorkommen nach dem ersten wächst die Bounding Box eines Variable-Gadgets um 64 Felder. Allerdings müsste die Bounding Box nur t Felder über das letzte Threshold-Gadget hinaus reichen, wenn t die Anzahl der Threshold-Gadgets ist, die das Variable-Gadget an dieser Richtung aktivieren kann.

Auch könnte es interessant sein zu sehen, wie sich die Reduktion vereinfachen lässt, wenn auch Zahlfelder mit Werten größer eins zu gelassen würden.

Die RPM-3SAT-Formel ist gegenwärtig im Programm fest einprogrammiert. Eine Eingabe für die Formel im Programm wäre noch wünschenswert. Dafür sollte dann die Formel gleichzeitig in ihrer graphischen Darstellung angezeigt werden, für eine besserer Übersichtlichkeit für die User/innen.

ZHED könnte für andere Probleme, die auf einem Grid liegen, interessant sein, aber auch für Probleme, bei denen die Reihenfolge der Ausführungen von Aktionen eine wichtige Rolle spielt. Da bekannt ist, dass ZHED NP-vollständig ist, könnte eine Reduktion von ZHED auf entsprechende Probleme für Beweise genutzt werden.

Literaturverzeichnis

- [1] P. K. Agarwal, B. Aronov, T. Geft, and D. Halperin. On two-handed planar assembly partitioning with connectivity constraints, 2023. URL <https://arxiv.org/abs/2009.12369>.
- [2] M. de Berg and A. Khosravi. Optimal binary space partitions in the plane. In M. T. Thai and S. Sahni, editors, *Proceedings of the 16th Annual International Conference on Computing and Combinatorics*, COCOON'10, page 216–225, Berlin, Heidelberg, 2010. Springer-Verlag.
- [3] R. Diestel. *Graphentheorie*. Springer-Verlag, Berlin, Heidelberg, 3 edition, 2006.
- [4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Bell Telephone Laboratories, Incorporated, 1979.
- [5] Ground Control Games. ZHED, 2017. URL <https://gcontrolgames.com/zhed/>. Zuletzt zugegriffen am 29.12.2024.
- [6] D. E. Knuth and A. Raghunathan. The problem of compatible representatives, 1992. URL <https://arxiv.org/abs/cs/9301116>.
- [7] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2): 329–343, 1982. doi: 10.1137/0211025. URL <https://doi.org/10.1137/0211025>.
- [8] Pygame Community. Pygame: A set of python modules designed for writing video games, 2025. URL <https://www.pygame.org/>. Version 2.6.1, Zugriff am 5. Februar 2025.
- [9] Python Software Foundation. *Python Language Reference*, 2025. URL <https://docs.python.org/3/reference/>. [Online; accessed 05-February-2025].
- [10] Python Software Foundation. *Python itertools module*, 2025. URL <https://docs.python.org/3/library/itertools.html>. Teil der Python Standardbibliothek.
- [11] L. A. R. L. Rivest, A. Shamir. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

- [12] S. Saha and E. D. Demaine. Zhed is np-complete, 2021. URL <https://arxiv.org/abs/2112.07914>.
- [13] M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, Boston, Massachusetts, 2 edition, 2006.