Vergleich von Backtracking, Dynamischer Programmierung und ILPs zum Lösen von Nonogrammen

Bachelorarbeit

Malte Jonas Kriese
457908

2. Dezember 2024

Supervisor: Prof. Dr. Benjamin Blankertz
Dr. Stefan Fricke

Technische Universität Berlin
School of Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Neurotechnology

Kurzfassung

In dieser Arbeit beschäftige mich mit dem Vergleich von verschiedenen Algorithmen zum Lösen von Nonogrammen.
Ich habe mich für dieses Thema entschieden, da ich selbst schon sehr lange Nonogramme löse. Durch meine Erfahrung habe ich Interesse, mich näher mit dem Thema auseinander zu setzen, insbesondere in Zusammenhang mit Algorithmen. Ich habe mir daher die Frage gestellt, welche Möglichkeiten es für einen Computer gibt, Nonogramme zu lösen.
Für diesen Vergleich wurden 5 verschiedene Algorithmen (Backtracking, regelbasierter Algorithmus, dynamischer Algorithmus, zwei lineare Optimierungen) in Python implementiert und ihre Laufzeiten beim Lösen von Nonogrammen in verschiedenen Größen verglichen.
Dabei kam heraus, dass der dynamische Algorithmus der schnellste war. Die erste lineare Optimierung und der regelbasierte Algorithmus waren der zweit- beziehungsweise drittschnellste. Beide hatten bei den größten getesteten Nonogrammen (30X30) Probleme. Der viertschnellste Algorithmus war der Backtracking Algorithmus. Er war zwar für viele Nonogramme sehr schnell, aber auch für viele sehr langsam. Am langsamsten war die zweite lineare Optimierung. Diese war noch nicht einmal in der Lage, alle der kleinsten Nonogramme (10X10) im von mir gewählten Zeitlimit von 10 Minuten zu lösen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Nonogramme sind japanische Logikrätsel, welche 1988 von Non Ishida erfunden wurden. Das Problem, ob ein Nonogramm lösbar ist, ist NP-vollständig. [1]
Ein nXm Nonogramm besteht aus einem Feld mit n Zeilen und m Spalten. Dabei müssen zur Lösung des Nonogramms alle Zellen dieses Feldes entweder schwarz oder weiß gefärbt werden. Für jede Zeile werden an der linken Seite und für jede Spalte werden oben Zahlen vorgegeben, welche anzeigen wie viele Zellen in der jeweiligen Zeile beziehungsweise Spalte schwarz gefärbt werden müssen.
Dabei gibt jede Zahl an, wie viele hintereinander liegende Zellen schwarz gefärbt werden müssen. Die zusammenhängenden schwarzen Zellen werden hier Blöcke genannt. Zwischen jedem schwarzen Block muss mindestens eine weißes Zelle liegen. Sind in einer Zeile/Spalte mehrere Blöcke, gibt die erste Zahl jeweils die Länge des ersten Blocks dieser Zeile/Spalte an, die zweite die Länge des zweiten Blocks und so weiter. Diese Zahlen werden hier Bedingungen genannt.
Nonogramme können eine, mehrere oder keine Lösung haben, wobei die meisten veröffentlichten Nonogramme eine haben. In dieser Arbeit werden nur Nonogramme genutzt, welche mindestens eine Lösung haben. Bei Nonogrammen mit mehren Lösungen wird nach dem Finden von einer Lösung nicht nach weiteren Lösungen gesucht.
Im weiteren Verlauf der Arbeit wird Linie als Bezeichnung sowohl für eine Zeile als auch für eine Spalte genutzt.

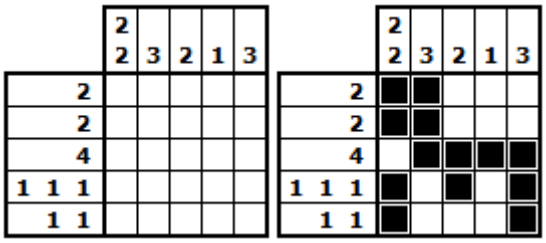

Abbildung 1.1: Ein 5X5 Nonogramm mit seiner Lösung [1]

[1] Abbildung 1.1: https://www.puzzle-nonograms.com/

1.1 Handsimulation

Es gibt für Menschen verschiedene Möglichkeiten, dieses Nonogramm zu lösen. Dabei sind die grauen Zellen im Beispiel Zellen, bei denen noch nicht bekannt ist, ob sie weiß oder schwarz gefärbt werden.
Ein Weg zur Lösung dieses Nonogramms wäre es, zu Beginn die erste Spalte zu färben. Diese kann eindeutig gefärbt werden, da die beiden Blöcke in der unteren und oberen Ecke starten müssen, um genug Platz für die weiße Zelle in der Mitte zu haben.
Daraufhin kann die erste Zeile gelöst werden. Es ist eindeutig, dass die bereits gefärbte Zelle ganz links in der Zeile um 1 erweitert werden muss, um die Bedingung zu erfüllen. Da sich in einer Richtung der Rand des Nonogramms befindet, kann nur die Zelle rechts von dieser Zelle schwarz gefärbt werden. Der Rest der Zeile muss weiß gefärbt werden. Für die zweite Zeile liegen die gleichen Bedingungen vor und sie kann genauso gefärbt werden.
Ebenso kann jetzt die dritte Zeile gefärbt werden, da es nur noch 4 freie Felder gibt, welche auch alle vom Block genutzt werden müssen. Auch die vierte Zeile kann eindeutig gelöst werden, da es nur eine Möglichkeit gibt, 3 einzelne schwarze Zellen in einer Reihe von 5 Feldern unterzubringen.
Durch diese Färbungen sind nun auch die Bedingungen der Spalte 2-4 bereits erfüllt. Es bleibt somit nur noch die letzte Spalte, welche nun eindeutig gelöst werden kann, indem die unterste Zelle der Spalte gefärbt wird, da sich der Block aufgrund von weißen Zellen nicht nach oben erweitern kann. Wenn anschließend die restlichen Zellen weiß gefärbt werden, ist das Nonogramm gelöst.

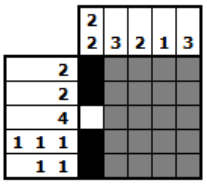

Abbildung 1.2: Nonogramm Schritt 1

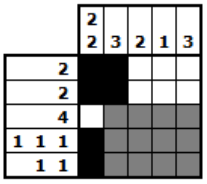

Abbildung 1.3: Nonogramm Schritt 2

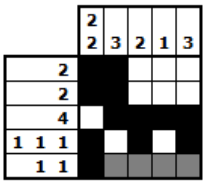

Abbildung 1.4: Nonogramm Schritt 3

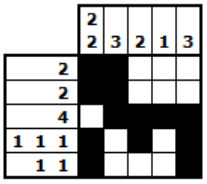

Abbildung 1.5: Nonogramm Schritt 4 [2]

[2] Abbildung 1.2 - 1.4: Nonogamm von https://www.puzzle-nonograms.com/, Lösung selbst erstellt

2 Methoden

Dieser Bereich stellt die für diese Arbeit genutzten Algorithmen vor. Dabei ist der erste Algorithmus (2.1) ein simpler Backtracking Algorithmus. Der zweite Algorithmus (2.2) nutzt logische Regeln, um das Nonogramm zu lösen. Der dritte Algorithmus (2.3) nutzt dynamische Programmierung, um einzelne Linien des Nonogramms zu lösen. Die letzten beiden Algorithmen (2.4) übersetzen das Nonogramm in verschiedene lineare Optimierungsprobleme. Für diese Art von Problemen gibt es bereits gut optimierte Algorithmen, welche dann für die Lösung genutzt werden können. Alle Algorithmen wurden in Python implementiert.

2.1 einfacher Backtracking Algorithmus

Der erste genutzte Algorithmus ist ein simpler Backtracking Algorithmus, welcher alle möglichen Färbungen der Zellen ausprobiert. Dabei wird keine Heuristik verwendet, sondern die Felder werden nur Zeile für Zeile von links oben nach rechts unten durchlaufen. Er stoppt, wenn eine Lösung gefunden wird, die alle Bedingungen erfüllt. Dabei sucht er, wie alle anderen genutzten Algorithmen, nicht nach weiteren Lösungen, sobald er eine Lösung gefunden hat. Er wird nur genutzt, um einen Vergleichswert für die anderen Algorithmen zu bekommen.

2.2 Regelbasierter Algorithmus

Dieser Algorithmus stammt aus dem Artikel „An Efficient Approach to Solving Nonograms“[2] von Yu, Lee und Chen. Der Algorithmus benutzt logische Regeln für die Einschränkung des Bereichs, in dem ein Block liegen kann und um Felder mit nur einer möglichen Färbung zu finden und zu färben. Diese Regeln werden im Folgenden dargestellt.
Wenn nach den logischen Regeln keine Felder mehr gefärbt und keine Bereiche eingeschränkt werden können, wird Backtracking angewendet, um den Rest des Nonogramms zu lösen. Dabei werden nach jedem Backtracking Schritt alle logischen Regeln erneut angewendet, um mehr Felder zu lösen.

Im weiteren Verlauf bezeichnet:
AnzahlBlöcke_j die Anzahl der Blöcke in Linie j,

$\text{Block}_{i,j}$ den i-ten Block in Linie j und

$\text{LängeBlock}_{i,j}$ die Länge des i-ten Blocks in Linie j.

2.2.1 Bereiche

Der Algorithmus definiert für jeden Block des Nonogramms einen Bereich, in dem dieser Block liegen muss. Im Laufe des Algorithmus wird dieser Bereich durch die Anwendung der Regeln immer weiter eingeschränkt, bis er nur noch genauso lang wie der Block selbst ist und somit komplett schwarz gefärbt werden muss.
Am Anfang des Algorithmus wird für jeden Block ein initialer Bereich berechnet, in dem eine Färbung möglich ist. Dabei wird der Startpunkt des Bereiches vom i-ten Block in Zeile j mit der Formel:
$S_{1,j} = 0$,
$S_{i,j} = \sum_{k=1}^{i-1}(LängeBlock_{k,j} + 1)$, für alle i > 2
und der Endpunkt mit der Formel:
$E_{i,j} = (n-1) - \sum_{k=j+1}^{AnzahlBlöcke_j}(LängeBlock_{k,j} + 1)$, für alle i < $AnzahlBlöcke_j$
$E_{AnzahlBlöcke_j,j} = n - 1$
berechnet. Durch diese Formeln wird die minimale Länge, die alle Blöcke davor/danach einnehmen können, berechnet. Für Spalten gelten die gleichen Formeln, nur wird n-1 in den Endpunktformeln durch m-1 ersetzt.
Wird diese Formel auf die letzte Zeile des Beispiels in Abbildung 1.1 angewendet, bekommt man (0,2) und (2,4) als initiale Bereiche.

2.2.2 Regeln 1.1 - 1.5

Die folgenden 5 Regeln des Algorithmus versuchen Zellen direkt zu färben.

Regel 1.1

Regel 1.1 bezieht sich auf Bereiche, die maximal eine Länge von 2 mal der Länge ihres Blocks haben. Bei diesen Blöcken können die mittleren Zellen gefärbt werden, da diese unabhängig davon, wo der Block anfängt, immer gefärbt sein müssen.
Für jeden $Block_{i,j}$, wird die k-te Zelle der Linie j schwarz gefärbt, für alle k,
$E_{i,j} - LängeBlock_{i,j} + 1 \leq k \leq S_{i,j} + LängeBlock_{i,j} - 1$.
Konkret berechnet der Teil der Formel vor dem ersten ≤ den letzten möglichen Startpunkt des Blocks und der Teil nach dem zweiten ≤ den ersten möglichen Endpunkt. Wenn dabei herauskommt, dass der erste Endpunkt größer als der letzte Startpunkt ist, müssen alle dazwischen liegenden Zellen schwarz gefärbt werden.

Regel 1.2

Nach Regel 1.2 ist eine Zelle, die in keinem Bereich enthalten ist, weiß zu färben. Dies kann leicht kontrolliert werden. Hierzu wird überprüft, ob der Startpunkt eines Bereiches hinter dem Endpunkt des Bereiches davor liegt. Ist dies der Fall, können alle Zellen zwischen diesen beiden weiß gefärbt werden.

Regel 1.3

Regel 1.3 tritt in Kraft, wenn die erste Zelle des Bereiches von Block i schwarz gefärbt ist und diese Zelle nur zu diesem Bereich und zu Bereichen von Blöcken der Länge 1 gehört. Ist dies der Fall, kann die Zelle davor weiß gefärbt werden.
Dies liegt daran, dass die Zelle entweder zu Block i oder zu einem Block der Länge 1 gehört. Wenn die Zelle zu einem Block der Länge 1 gehört, muss auf beiden Seiten eine weiße Zelle sein, um den Block von anderen Blöcken abzutrennen. Gehört die Zelle zu Block i, muss die Zelle davor ebenso weiß gefärbt sein, da der Block seinen Bereich nicht verlassen kann. Somit ist die Zelle davor in beiden Fällen weiß und sie kann bereits weiß gefärbt werden, auch ohne zu wissen, um welchen Fall es sich handelt.
Das gleiche gilt auch für den Fall, dass das letzte Feld eines Bereiches (range) schwarz gefärbt ist.

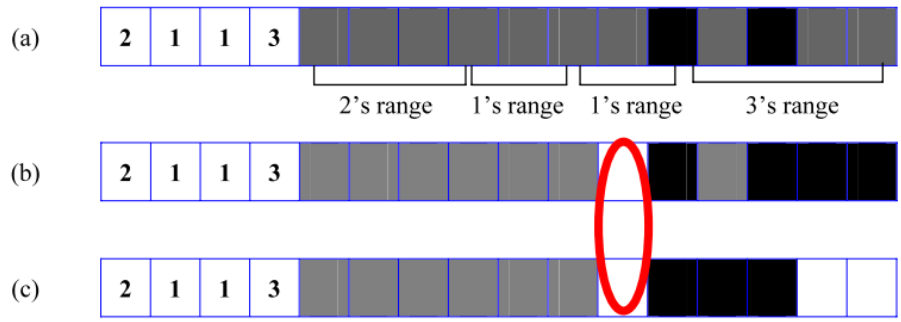

Abbildung 2.1: Ein Beispiel für Regel 1.3. Dabei zeigen (b) und (c) die beiden Möglichkeiten[2]

Regel 1.4

Regel 1.4 bezieht sich auf zwei Mengen von zusammenhängenden schwarz ausgefüllten Zellen zwischen welchen sich nur eine unausgefüllte Zelle befindet. Wenn durch Verbinden der Mengen, indem die mittlere Zelle schwarz gefärbt wird, eine neue Menge entsteht, welche größer ist als der größte Block aller Bereiche, die beiden ursprünglichen Mengen enthalten, kann die Zelle nicht schwarz gefärbt werden. Somit muss sie weiß gefärbt werden.

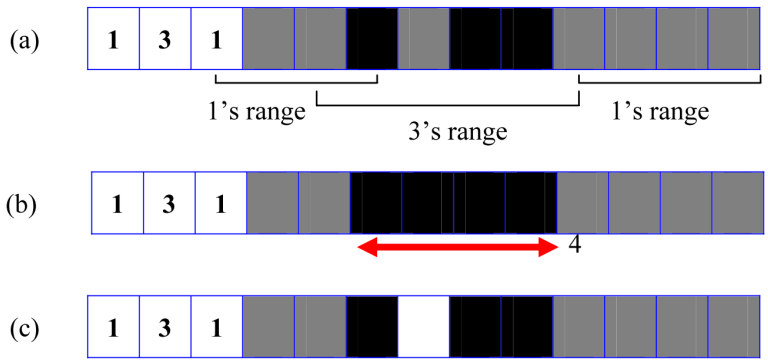

Abbildung 2.2: Ein Beispiel für Regel 1.4. Dabei zeigt (b), dass durch das schwarz Färben der mittleren Zelle ein zu großer Block entsteht. (c) zeigt die richtige Lösung.[2]

Regel 1.5

Regel 1.5 besteht aus zwei Teilen. Der erste Teil tritt in Kraft, wenn eine Menge von zusammenhängenden schwarzen Zellen in einer Richtung in der Ausbreitung begrenzt ist, entweder durch den Rand des Nonogramms oder durch eine weiße Zelle. In diesem Fall wird unter allen Bereichen, die diese Menge von Zellen enthalten, der kürzeste Block gesucht. Wenn die Länge des kürzesten Blocks abzüglich der Anzahl der schwarzen Zellen und abzüglich des Abstands zur Begrenzung ≥ 0 ist, kann die Menge durch weitere schwarze Zellen, weg von der Begrenzung, erweitert werden, bis die Formel nicht mehr erfüllt ist.
Der zweite Teil bezieht sich auf den Fall, wenn der größte Block, dessen Bereich eine Menge von zusammenhängenden schwarzen Zellen enthält, die gleiche Länge wie diese Menge hat. In diesem Fall kann die Menge nicht erweitert werden und an beiden Enden muss die Zelle weiß gefärbt werden.

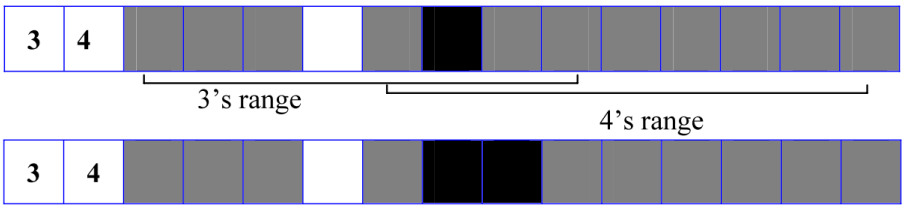

Abbildung 2.3: Ein Beispiel für den ersten Teil von Regel 1.5. Oben ist der Ausgangspunkt und unten das Resultat [2]

2.2.3 Regeln 2.1 - 2.3

Die Regeln 2.1 - 2.3 werden genutzt, um die Bereiche zu verkleinern und dadurch anderen Regeln zu helfen, weitere Zellen zu färben.

Regel 2.1

Regel 2.1 bezieht sich auf dem Abstand zwischen den Startpunkten von einem Bereich und dem Bereich davor. Jeder Startpunkt muss um die Länge des vorherigen Blocks + 1 größer sein als der Startpunkt davor, damit genug Platz für den Block vorhanden ist. Ist dies nicht der Fall muss der Startpunkt nach hinten verschoben werden, bis die Bedingung erfüllt ist. Äquivalent gilt für Endpunkte, dass ein Endpunkt mindestens die Länge des Blocks danach + 1 kleiner als der nächste Endpunkt sein muss.

Regel 2.2

Mit der Regel 2.2 kann der Startpunkt oder der Endpunkt eines Bereiches verändert werden, wenn er sich neben einer schwarzen Zelle befindet. Da zwischen zwei schwarzen Blöcken immer ein weißes Feld sein muss, ist der Startpunkt eines Bereiches um 1 nach hinten zu verschieben, wenn das Feld vor dem Startpunkt schwarz gefärbt ist. Ebenso wird der Endpunkt eines Bereiches um 1 nach vorne verschoben, wenn ein Feld rechts von Endpunkt schwarz gefärbt ist.

Regel 2.3

Diese Regel kommt zum Einsatz, wenn es in einem Bereich eine Menge von zusammenhängenden schwarzen Zellen gibt, welche größer als der zum Bereich gehörende Block ist. In diesem Fall wird überprüft, zu welchen Blöcken die Menge aufgrund ihrer Position und Größe gehören kann. Wenn es möglich ist festzustellen, dass die Menge nur zu Bereichen nach dem jetzigen Bereich gehören kann, wird der jetzige Bereich so verkürzt, dass er zwei Felder vor dieser Menge endet. Ebenso wird der Bereich verkürzt, wenn der Block nur zu Bereichen davor passt. Dabei wird der Startpunkt auf zwei Zellen hinter dem Block gesetzt. Wenn es möglich ist, dass der Block sowohl zu Bereichen davor als auch danach gehören kann, wird nichts geändert.

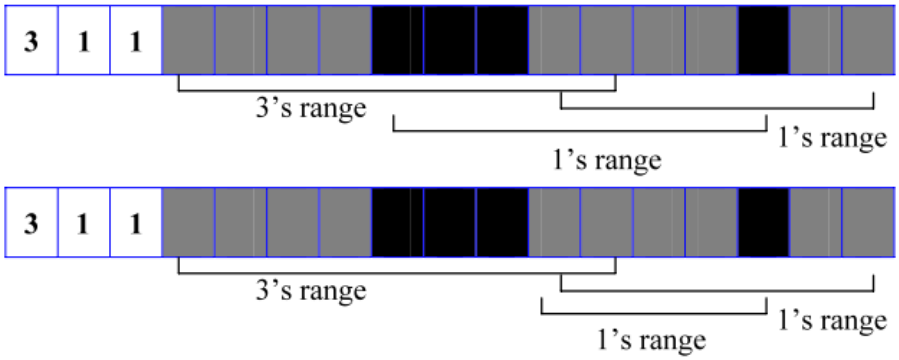

Abbildung 2.4: Ein Beispiel für Regel 2.3 Oben ist der Ausgangspunkt und unten das Resultat[2]

2.2.4 Regeln 3.1 - 3.3

Die Regeln 3.1 - 3.3 sollen sowohl Felder färben, als auch die Bereiche weiter verkleinern.

Regel 3.1

Die Regel wird genutzt, wenn es mehrere Mengen von zusammenhängenden schwarzen Zellen gibt, die nur zu einem Bereich gehören. In diesem Fall können alle Zellen zwischen den schwarzen Zellen schwarz gefärbt werden, da die schwarzen Zellen zu einem Block gehören müssen.

Regel 3.2

Mit dieser Regel wird der Bereich verkleinert, wenn eine weiße Zelle verhindert, dass es genug Platz für den Block gibt. Dafür wird der Abstand zwischen dem Startpunkt des Bereiches und der ersten weißen Zelle des Bereiches genommen. Ist dieser kleiner als die Länge des Blocks des Bereiches, kann der Startpunkt auf die Zelle hinter der weißen Zelle gesetzt werden. Dies wird wiederholt bis der Abstand zwischen Startpunkt und der weißen Zellen groß genug für den Block ist. Anschließend wird diese Regel auf den Endpunkt des Bereichs angewendet.

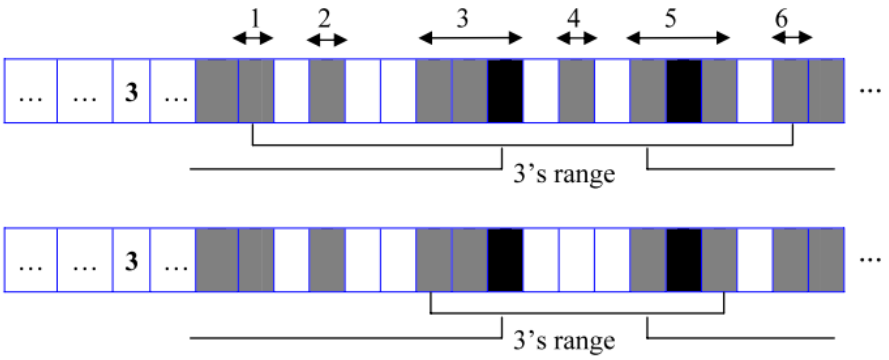

Abbildung 2.5: Ein Beispiel für Regel 3.2. Oben ist der Ausgangspunkt und unten das Resultat [2]

Regel 3.3

Hierbei handelt es sich um eine Gruppe von Regeln, die dann angewendet werden, wenn ein Bereich keine Überschneidung mit dem Bereich davor, beziehungsweise keine Überschneidung mit dem Bereich danach hat. Die Ausführungen zu diesen Regeln beziehen sich nur darauf, dass es keine Überschneidung mit dem Bereich davor gibt und sind äquivalent auch anzuwenden, wenn es keine Überschneidungen mit dem Bereich danach gibt.

Regel 3.3.1 Wenn die erste Zelle des Bereiches gefärbt ist, kann von dieser Stelle aus der gesamte Block gefärbt und der Endpunkt neu gesetzt werden, da die Zelle eindeutig zu diesem Bereich gehört und der Block sich nicht in die andere Richtung ausbreiten kann.

Regeln 3.3.2 Wenn eine weiße Zelle nach einer schwarzen Zelle im Bereich liegt, kann der Endpunkt vor die weiße Zelle gesetzt werden, da die schwarze Zelle entweder zu diesem Block oder einem Block danach gehört und der Block somit in beiden Fällen nicht über die weiße Zelle gehen kann.

Regel 3.3.3 Wenn es mehrere Mengen von schwarzen Zellen im Bereich gibt, kann der Bereich eingegrenzt werden, indem geprüft wird, wie viele Mengen verbunden werden können, bis die resultierende Menge zu groß für den Block wird. Dann kann der Endpunkt auf die Zelle zwei vor dem Beginn der Menge, durch welche der Block zu groß wird, gesetzt werden.

2.2.5 Ablauf

Der Algorithmus beginnt damit, die initialen Bereiche zu berechnen. Anschließend werden die Regeln so lange angewendet, bis es zu keinen Änderungen mehr kommt. Wenn das Nonogramm an diesem Punkt noch nicht gelöst ist, wird Backtracking genutzt, um den Rest zu lösen. Das heißt, es wird ein Feld ausgewählt, welches schwarz gefärbt wird. Daraufhin werden die Regeln erneut angewendet, bis es keine Änderungen mehr gibt. Dies wird wiederholt, bis es entweder eine Lösung gibt oder das Nonogramm nicht mehr lösbar ist. Wenn es nicht mehr lösbar ist, wird in der Lösung des Nonogramms vor den letzten Backtrackingschritt zurückgegangen und das Feld stattdessen weiß gefärbt. Daraufhin wird der Prozess fortgesetzt.
Im Rahmen dieser Arbeit werden für den Vergleich Nonogramme genutzt, welche von Menschen gelöst werden sollen. Diese sind größtenteils durch Nutzung der Regeln zum Lösen von Nonogrammen und nicht durch Ausprobieren lösbar. Daher sind diese meist ohne Backtracking lösbar. Sollte das Nonogramm mehrere Lösungen haben, ist das Backtracking zur Entscheidung zwischen verschiedenen Lösungen nötig.

2.3 Dynamischer Algorithmus

Der Ansatz stammt aus dem Paper „An Efficient Approach to Solving Nonograms “[3] von Wu et al.. Er nutzt die dynamischen Funktionen Fix und Paint. Fix dient der Überprüfung, ob es möglich ist, eine Linie nach den Bedingungen auszufüllen. Paint wird genutzt, um diese Linie so weit es geht auszufüllen. Außerdem nutzt der Algorithmus eine Probe Funktion, welche systematisch verschiedene Färbungen ausprobiert, um weitere Zellen auszufüllen, bevor Backtracking genutzt wird.
Für diesen Algorithmus wird die Färbung des Nonogramms wie folgt kodiert. Eine 1 in einer Zelle bedeutet, dass die Zelle schwarz gefärbt ist. Eine 0 bedeutet, dass sie weiß gefärbt ist und eine 2, dass die Färbung noch nicht bekannt ist.

2.3.1 Fix

Die Funktion Fix überprüft, ob eine schon teilweise gefärbte Linie noch zu Ende ausgefüllt werden kann, oder ob bereits eine Zelle falsch gefärbt wurde und das Nonogramm somit nicht mehr lösbar ist.
Ein Aufruf von Fix(i, j) überprüft dabei, ob es möglich ist, in den ersten i Zellen einer Linie die ersten j Bedingungen zu erfüllen. Dafür probiert er in der letzten Zelle eine 0 oder 1 zu setzen, indem die Funktionen Fix0 (Funktion 2) und Fix1 (Funktion 3) aufgerufen werden, welche untersuchen, ob es mit diesem Setzen möglich ist, die Linie richtig zu färben. Diese Funktionen wiederum rufen erneut Fix mit verringerten Parametern auf, wodurch es zu einer Rekursion kommt. Diese endet, wenn entweder eine Lösung von einem anderen Aufruf mit gleichen Parametern gefunden wird oder i auf 0 reduziert ist.

Funktion 1 Fix(i, j)

```
if i = 0 then
    if j = 0 then
        return True
    else
        return False
    end if
end if
if Tabelle[i][j] ≠ [] then
    return Tabelle[i][j]
end if
Tabelle[i][j] = Fix0(i, j) ∨ Fix1(i, j)
return Tabelle[i][j]
```

Der erste Teil der Fix Funktion ist die Abbruchbedingung. Wenn i den Wert 0 erreicht, ist es eindeutig, ob eine Färbung möglich ist. Wenn 0 Zellen mit 0 Blöcken gefärbt werden sollen, ist dies möglich und True wird zurückgegeben. Wenn allerdings 0 Zellen mit mindestens einem Block gefärbt werden sollen, ist dies eindeutig nicht möglich und es wird False zurückgegeben.
Anschließend wird überprüft, ob es bereits einen Aufruf mit den gleichen Parametern gab. Ist dies der Fall, kann einfach das Ergebnis diese Aufrufs, welches in Tabelle[i][j] gespeichert wurde, zurückgegeben werden, da sich das Ergebnis nicht ändert.
Sind diese beiden Bedingungen nicht erfüllt, werden Fix0 (Funktion 2) und Fix1 (Funktion 3) aufgerufen. Diese simulieren das Setzen von einer 0 beziehungsweise einer 1 in Zelle i und geben zurück, ob die Linie nach diesem Setzen noch lösbar ist. Wenn eine von den beiden Möglichkeiten zu dem Ergbnis kommt, dass die Linie nach dem Setzen noch lösbar ist, gibt es eindeutig eine Lösung für die ursprüngliche Linie. Ist keine der beiden Variationen lösbar, ist auch die ursprüngliche Linie nicht lösbar. Diese wird in der Tabelle für zukünftige Aufrufe abgespeichert und zurückgegeben.

Funktion 2 Fix0(i, j)

```
if i - 1 kleiner als die minimale Länge der Blöcke then
    return False
end if
if Linie[i] ≠ 1 then
    return Fix(i-1, j)
else
    return False
end if
```

Fix0 beginnt mit einer weiteren Abbruchbedingung. Wenn der restliche Platz nach Setzen einer 0, in diesem Fall i - 1, zu kurz für die restlichen Bedingungen ist, kann sofort abgebrochen werden und False zurückgegeben werden. Dies wird überprüft, indem die minimale Länge aller verbleibenden Bedingungen berechnet wird. Hierfür wird die folgende Formel genutzt:
$Min_{i,j} = \sum_{k=1}^{i}(LängeBlock_{i,j} + 1) - 1$
Sie addiert alle Block Längen + 1, was der Länge von allen Blöcken hintereinander mit nur einem Abstand von einem weißem Feld entspricht. Am Ende wird 1 abgezogen, da nach dem letzten Block kein Abstand mehr gebraucht wird.
Anschließend wird die Möglichkeit überprüft, an Position i in der Linie eine 0 zu setzen. Dies ist möglich, wenn bereits eine 0 an der Postion ist oder eine 2, welche überschrieben werden kann. Kann eine 0 geschrieben werden, prüft ein neuer Aufruf von Fix, mit einem um 1 reduzierten i, ob der Rest der Linie noch lösbar ist und dieses Ergebnis wird zurückgegeben. Kann keine 0 geschrieben werden, wird False zurückgegeben.

Funktion 3 Fix1(i, j)

```
if j ≥ 1 then
    if für alle k, mit i-Bedingung[j]<k<i, Linie[k] ≠ 0 then
        if j ≥ 2 then
            if Linie[i-Bedingung[j]] == ≠ 1 then
                return Fix(i-Bedingung[j]-1, j-1)
            else
                return False
            end if
        else
            return Fix(i-Bedingung[j], j-1)
        end if
    end if
end if
return False
```

Fix1 hat die Besonderheit, dass nicht nur eine 1 gesetzt werden muss, sondern so viele wie in der j-ten Bedingung vorgegeben sind. Dafür muss in allen Zellen überprüft werden, ob eine 1 gesetzt werden kann. Wenn nach der jetzigen Bedingung noch eine weitere Bedingung folgt, muss überprüft werden, ob an der Stelle davor eine 0 gesetzt werden kann.
Kann eine dieser Zahlen nicht gesetzt werden, ist es nicht möglich eine 1 in Zelle i zu setzen und es wird False zurückgegeben. Andernfalls wird Fix erneut aufgerufen, um zu überprüfen, ob der Rest lösbar ist. Hierbei wird i um Bedingung[j] reduziert, wenn keine weiße Zelle gesetzt wurde. Wenn eine weiße zelle gesetzt wurde, wird um Bedingung[j] + 1 reduziert. Ebenso wird j um 1 reduziert, da die j-te Bedingung gerade erfüllt wurde.

2.3.2 Paint

Die Paint Funktion (Funktion 4) wird genutzt, um eine teilweise gefärbte Linie weiter auszufüllen. Dabei betrachtet sie ähnlich wie Fix Teile der Linie mit den gleichen Parametern i und j. Ein Aufruf von Paint(i,j) versucht dabei, die ersten i Zellen einer Linie so zu färben, dass die ersten j Bedingungen erfüllt werden.
Dabei werden die Ergebnisse von Fix benutzt, indem kontrolliert wird, ob es möglich ist, in Zelle i eine 0 beziehungsweise 1 zu setzen. Ist dabei nur eins von beiden möglich, wird an Stelle i eindeutig eine 0 beziehungsweise 1 gesetzt und der Rest der Linie wird durch einen Aufruf von Paint0(Funktion 5)/Paint1(Funktion 6) generiert. Dieses Ergebnis wird dann zurückgegeben. Wenn sowohl eine 0 als auch eine 1 gesetzt werden können, werden die Linien für beide Fälle generiert und verglichen. Wenn bei diesem Vergleich eine Zelle in beiden Fällen den gleichen Wert hat, wird dieser für die Lösung genutzt, andernfalls wird eine 2 gesetzt.

Funktion 4 Paint(i, j)

```
if i = 0 then
    return ""
end if
if Tabelle[i][j] ≠ [] then
    return Tabelle[i][j]
end if
if Fix1(i,j) and not Fix0(i,j) then
    Tabelle[i][j] = Paint1(i,j)
    return Tabelle[i][j]
end if
if Fix0(i,j) and not Fix01(i,j) then
    Tabelle[i][j] = Paint0(i,j)
    return Tabelle[i][j]
end if
Tabelle[i][j] = Merge(Paint0(i,j), Paint1(i,j))
return Tabelle[i][j]
```

Die Paint Funktion startet mit der Abbruchbedingung der Rekursion. Wenn i = 0 ist, also die zu färbende Linie eine Länge von 0 hat, wird ein leerer String zurückgegeben. Dieser wird erweitert, während er die Rekursion zurück zum Start durchläuft.
Anschließend wird wie bei Fix geschaut, ob es schon eine Lösung gibt und diese wird zurückgegeben, wenn sie existiert.
Dann wird mit Fix0/Fix1 geschaut, ob es möglich ist, hier eine 0 beziehungsweise 1 zu setzen. Kann nur eine 0 gesetzt werden, geht es mit der Funktion Paint0 (Funktion 5) weiter, welche an der Stelle i eine 0 setzt und den Rest der Linie mit Paint mit einem reduzierten i generiert. Dieses Ergebnis wird dann zurückgegeben. Ebenso wird Paint1 (Funktion 6) aufgerufen, wenn es nur möglich ist, eine 1 zu setzen. Paint1 funktioniert äquivalent zu Paint0, nur dass an Stelle i eine 1 gesetzt wird.
In dem Fall, dass sowohl eine 0 oder 1 gesetzt werden können, wird eine Linie von sowohl Paint0 als auch Paint1 generiert. Diese Ergebnisse werden dann in Merge (Funktion 7) verglichen, sodass eine neue Linie entsteht. Diese hat in allen Zellen, die in beiden Ausgangslinien den gleichen Wert haben, genau diesen Wert. In den Zellen, in denen sich die Ausgangslinien unterscheiden, hat sie eine 2.

Funktion 5 Paint0(i, j)

```
return Paint(i-1, j)+"0"
```

Paint0 soll die Linie nach dem Setzen einer 0 an Position i generieren. Dafür wird einfach Paint mit einem um 1 reduzierten i aufgerufen. Anschließend wird der Rückgabewert dieses Aufrufs mit einer 0 konkateniert und ebenfalls zurückgegeben.
Paint1 generiert die Linie, nachdem eine 1 an Position i einfügt wurde und somit auch eine 1 in den

Funktion 6 Paint1(i, j)

```
if j ≥ 1 then
    return Paint(i-Bedingung[j] - 1, j-1)+“0“+“1“*Bedingung[j]
else
    return Paint(i-Bedingung[j], j-1)++“1“*Bedingung[j]
end if
```

Zellen davor, um die Bedingung j zu erfüllen. Allerdings muss hier unterschieden werden, ob es vor der jetzigen Bedingung j noch eine weitere Bedingung gibt. Wenn es eine weitere gibt, muss eine 0 vor den Zellen mit den Einsen gesetzt werden. Dafür wird Paint mit einem um Bedingung[j]+1 reduzierten i und einem um 1 reduzierten j aufgerufen. Das Ergebnis dieses Aufrufs wird dann mit einer 0 und einer Anzahl von Einsen nach Bedingung[j] konkateniert.
Wenn es keine weitere Bedingung gibt, muss i nur um Bedingung[j] reduziert werden und es wird keine 0 mit dem Ergebnis konkateniert.

Funktion 7 Merge(Linie0, Linie1)

```
neueLinie = ““
k = 1
while k ≤ Länge der Linie do
    if Linie0[k] = Linie1[k] then
        neueLinie += Linie0[k]
    else
        neueLinie += “2“
    end if
    k += 1
end while
return neueLinie
```

Die Merge Funktion wird zur Prüfung genutzt, ob es Überscheidungen zwischen zwei Linien gibt. Dafür wird jeder Eintrag der Linie verglichen. Ist der Eintrag gleich, wird dieser Eintrag in der Ergebnislinie gesetzt. Andernfalls wird eine 2 gesetzt.

2.3.3 Propagate

Die Propagate Funktion wird genutzt, um die Funktionen Fix und Paint auf ein gegebenes Nonogramm anzuwenden, um so viele Felder wie möglich auszufüllen. Dafür werden die Funktionen systematisch auf alle Zeilen und Spalten angewendet.

Funktion 8 Propagate(Nonogramm)

```
toDO = Liste aller Zeilen und Spalten im Nonogramm
alleÄnderungen = []
while toDO != [] do
    Linie = toDO erster Eintrag aus toDO, Linie aus toDO entfernen
    if Fix(Länge von Linie, Anzahl der Bedingungen von Linie) = False then
        return False
    else
        neueLinie = Paint(Länge von L, Anzahl der Bedingungen von L)
        speichere alle Änderungen in der Liste neueÄnderungen.
        Linie = neueLinie
        ergänze toDo um die Zeilen, beziehungsweise Spalten der Änderungen
        alleÄnderungen += neueÄnderungen
        neueÄnderungen = []
    end if
end while
return alleÄnderungen
```

Propagate beginnt damit, alle Zeilen und Spalten in einer Liste (toDo) zu speichern. Anschließend wird das erste Element der Liste entfernt und in der variablen Linie gespeichert, bis die Liste leer ist. Daraufhin wird Fix aufgerufen, um zu prüfen ob die Linie lösbar ist. Ist dies nicht der Fall, wird sofort False zurückgegeben, da wenn eine Linie nicht lösbar ist, das ganze Nonogramm nicht lösbar ist. Wenn die Linie lösbar ist, wird anschließend Paint aufgerufen, um die Linie so weit es geht auszufüllen.
Daraufhin werden die von Paint zurückgegebenen Linien und die ursprünglichen Linien verglichen und alle Unterschiede abgespeichert. Diese Unterschiede können nur die Änderungen von einer 2 zu einer 0 oder zu einer 1 sein. Anschließend wird die neue Linie in das Nonogramm eingesetzt. Dann wird toDo erweitert. Wenn Linie eine Spalte ist, werden die Zeilen, welche durch die neue Spalte geändert wurden, zu toDo hinzugefügt. Dabei werden nur Zeilen hinzugefügt, welche noch nicht in toDO sind. Das gleiche gilt auch für den Fall, dass Linie eine Zeile ist, nur dass Spalten hinzugefügt werden. Diese Erweiterung geschieht, damit die Linie erneut bearbeitet werden kann, da Paint durch die Änderungen eventuell neue Zellen ausfüllen kann. Ebenso können diese neu ausgefüllten Zellen dazu führen, dass eine Linie nicht mehr ausgefüllt werden kann, was dann durch Fix festgestellt wird. Wenn der Algorithmus die while Schleife durchlaufen hat, ohne dass ein Fehler von Fix festgestellt wurde, werden alle Änderungen zurückgegeben.

Ebenso wie beim regelbasierten Algorithmus reicht in den meisten der hier genutzten Nonogramme schon die Nutzung von Propagate für die Lösung.

2.3.4 Full Probe Konzept

Im Artikel[3] zu diesem Algorithmus wurden ursprünglich 3 verschiedene probing Algorithmen genutzt, die jeweils alle Funktionen des Algorithmus davor enthalten. Da der letzte Algorithmus, FP3, auch der schnellste war, wird hier nur dieser verwendet.
Allgemein funktioniert der probing Algorithmus so, dass zunächst zwei Kopien des Nonogramms erstellt werden. Das ursprüngliche Nonogramm wird im folgenden Hauptnonogramm genannt. In einer Kopie wird an einer Stelle k probiert, eine 0 zu setzen. In der anderen Kopie wird an der Stelle eine 1 gesetzt.
Anschließend wird auf diese Kopien die Propagate Funktion angewendet, um sie soweit wie möglich auszufüllen. Wenn Propagate herausfindet, dass eine der beiden Kopien des Nonogramms nicht lösbar ist, ist es eindeutig, dass die andere Belegung der Zelle richtig ist. Diese kann nun in das Hauptnonogramm eingetragen werden. Ebenso können alle aus dieser Färbung resultieren Färbungen in das Hauptnonogramm eingetragen werden.
Findet Propagate heraus, dass beide Kopien noch lösbar sind, werden sie verglichen. Dies funktioniert ähnlich wie in der Merge Funktion. Wenn an einer Stelle die gleiche Zahl steht, kann diese in das Hauptnonogramm übernommen werden, da - egal was an Stelle k steht - die Zelle gleich gefärbt ist. In diesem Fall werden die Kopien der Nonogramme gespeichert. Wenn das Hauptnonogramm weiter gelöst ist, können diese Kopien ebenfalls weiter ausgefüllt werden. Dies hat den Vorteil, dass die bereits von Propagate gesetzten Zellen nicht erneut gesetzt werden müssen. Dafür ist es wichtig, jedes Mal, wenn eine Zelle im Hauptnonogramm ausgefüllt wird, auch diese Zelle in allen gespeicherten Kopien der Nonogramme auszufüllen.
Der Algorithmus benutzt außerdem Kontrapositionen (wenn A $\implies$ B gilt, muss auch $!B \implies !A$ gelten), um mehr Zellen auszufüllen. Konkret für Nonogramme heißt dies, wenn durch das Ausfüllen von Zelle A mit einer 1, Zelle B mit einer 0 ausgefüllt wird, gilt A=1 $\implies$ B=0. Daraus lässt sich ableiten, das B!=0 $\implies$ A!=1 auch gelten muss. Somit gilt, wenn B mit einer 1 gefärbt wird, A mit einer 0 gefärbt werden muss.
Diese Kontrapositionen können aus Feldern, die beim proben ausgefüllt wurden, abgeleitet werden. Dafür wird eine Tabelle mit Eintragungen für jede Zelle und jede Färbung dieser Zelle angelegt. Wenn der Algorithmus die Kontraposition B!=0 $\implies$ A!=1 findet, wird in der Tabelle an Position B, 1 ein Eintrag mit A=0 hinzugefügt. Wenn dann später B geprobt wird, kann die Zelle A sofort auf 0 gesetzt werden.

2.3.5 Full Probe Umsetzung

Funktion 9 FP3(Nonogramm)

```
P = Liste aller ungefärbter Zellen
while P != [] do
    entferne die erste Zelle aus P und speichere sie als p
    if Probe(p) = False then
        return False
    end if
end while
```

Die FP3 Funktion wird nur zum Aufrufen von Probe3 (Funktion 10) genutzt. Dabei durchläuft sie eine Liste P mit Zellen, welche als Parameter dienen. Ursprünglich besteht diese Liste aus allen Zellen mit einer 2, also allen Zellen, die noch ausgefüllt werden müssen. Im weiteren Verlauf wird sie durch andere Funktionen weiter aufgefüllt. Dies passiert bei Änderungen am Nonogramm, die dafür sorgen, dass ein erneuter Aufruf mit diesem Parameter sinnvoll ist.

Funktion 10 Probe3(Nonogramm, p)

```
Nonogramm0 = ProbeG3(Nonogramm, p, 0)
Nonogramm1 = ProbeG3(Nonogramm, p, 1)
return Evaluate(Nonogramm, Nonogramm0, Nonogramm1)
```

Probe3 ist eine simple Funktion, die zweimal ProbeG3 (Funktion 11) aufruft. Einmal wird an Position p eine 0 und einmal eine 1 gesetzt. Anschließend wird Evaluate (Funktion 12) aufgerufen, um aus dem Ergebnissen der Aufrufe von Probe zu lernen.

In ProbeG3 (Funktion 11) wird damit begonnen zu schauen, ob es schon ein teilweise bearbeitetes Nonogramm für den Parameter gibt. Dadurch kann man sich eine Wiederholung der ersten Schritte sparen. Ist dies nicht der Fall, wird eine neue Kopie erstellt und Zelle p nach der Vorgabe gefärbt. Anschließend wird die Update Funktion aufgerufen, um alle Kontrapositionen zu laden.
Dansch wird in einer while Schleife Propagate und Update (Funktion 12) aufgerufen. Zunächst wird Propagate aufgerufen um so viele Zellen wie möglich zu färben. Danach wird Update aufgerufen, um aus den Kontrapositionen der von Propagate neu gesetzten Zellen weitere Zellen zu färben. Wenn Update False zurückgibt, konnte eine Zelle nicht gefärbt werden und das Nonogramm ist nicht mehr lösbar. Dadurch muss auch ProbeG3 False zurückgeben. Andernfalls wird Propagate erneut aufgerufen, um aus den von Update neu gesetzten Zellen zu lernen. Diese Schleife läuft solange, bis Update keine weiteren Zellen mehr färbt.
Wenn die while Schleife durchlaufen wurde, wird die KontrapositionsTabelle mit den neuen Änderungen upgedatet. Dabei wird außerdem die Liste P um alle Zellen erweitert, für welche es neue Kontrapositionen gibt.

Funktion 11 ProbeG3(Nonogramm, p)

```
if NonogrammTabelle[p] != [] then
    NonogrammCopy = NonogrammTable[p][i]
else
    NonogrammCopy = kopiere Nonogramm und färbe Pixel p 0
end ifUpdateListe = KontrapositionTabelle[p][i]
j == Update(Nonogramm, UpdateListe, 0)
UpdateListeKopie = 0
while UpdateListeKopie != UpdateListe do
    if Propagate(NonogrammCopy) = False then
        return False
    end if
    UpdateListe += alle durch Propagate neu gefärbten Zellen
    UpdateListeKopie = UpdateListe
    j = Update(Nonogramm, UpdateListe, j)
    if j == False then
        return False
    end if
end while
Änderungen = Liste aller Änderungen in Propagate
while Änderungen != [] do
    entferne die erste Änderung aus Änderungen und speichere ihre Zelle als p´ und ihren Wert als
c
    KontrapositionTabelle[p´][!c] += (p, !i)
    wenn p´ noch nicht in P ist, füge p´ zu P hinzu.
end while
return Nonogram
```

Funktion 12 Update(Nonogramm, UpdateListe, j)

```
while len UpdateListe > j do
    Update = UpdateListe [j]
    füge Update in Nonogramm ein, wenn die Zelle dabei mit einer anderen Farbe überschrieben
wird, return False
    füge alle Einträge aus KontrapositionsTabelle[Update] in UpdateListe ein, wenn diese noch
nicht in UpdateListe enthalten sind.
end while
```

Die Update Funktion wird genutzt, um Kontrapositionen zu laden. Dabei wird eine Liste von zu färbenden Zellen übergeben. Beim Durchlaufen werden die Kontrapositionen dieser Zellen ebenfalls in die Liste hinzugefügt und somit später auch gefärbt. Diese Herangehensweise sorgt dafür, dass auch die Kontrapositionen von Kontrapositionen von Zellen genutzt werden.

Funktion 13 Evaluate(Nonogramm, Nonogramm0, Nonogramm1))

```
if Nonogramm0 = False und Nonogramm1 = False then return False
end if
if Nonogramm0 = False then
    Nonogramm = Nonogramm1
    update NonogrammTable
    return alle Änderungen
end if
if Nonogramm0 = False then
    Nonogramm = Nonogramm0
    update NonogrammTable
    return alle Änderungen
end if
suche die in Nonogramm0 und Nonogramm1 gleich gefärbten Zellen und färbe diese im Hauptnonogramm.
update NonogrammTable
update P
return alle Änderungen
```

Die Evaluate Funktion wird zur Überprüfung genutzt, ob durch den Propagate Aufruf etwas Neues gelernt wurde. Zuerst wird geprüft, ob sowohl Nonogramm0 als auch Nonogramm1 False sind. Ist dies der Fall, kann das Hauptnonogramm nicht gelöst werden, da an einer Stelle weder eine 1 noch eine 0 gesetzt werden kann. Somit wird False zurückgegeben.
Wenn Nonogramm0 False ist und Nonogramm1 nicht, ist klar, dass an der Position p nur eine 1 stehen kann. Daher kann das Hauptnonogramm durch Nonogramm1 ersetzt werden, da Nonogramm1 weiter ausgefüllt ist und nur die Annahme besitzt, das p = 1 ist, welche sich jetzt als wahr herausgestellt hat. Außerdem werden alle neuen Eintragungen im Hauptnonogramm auch auf alle im NonogrammTable gespeicherten Nonogramme übertragen. Anschließend wird eine Liste aller Änderungen zurückgegeben. Das Gleiche gilt umgekehrt, wenn Nonogramm1 False ist und Nonogramm0 nicht.

Der letzte Fall ist, dass weder Nonogramm0 noch Nonogramm1 False sind. In diesem Fall wird geprüft, ob es Zellen gibt, die in beiden Nonogrammen gleich gefärbt sind. Ist dies der Fall, ist es eindeutig, dass diese Färbung richtig ist und sie kann in das Hauptnonogramm und alle Nonogramme im NonogrammTable übernommen werden. Desweiteren kann die Liste P um alle Nonogramme erweitert werden, in welchen hierdurch weitere Zellen ausgefüllt wurden. Dies sorgt dafür, dass alle diese Nonogramme mit den neu gefärbten Zellen erneut durchlaufen werden. Anschließend wird eine Liste mit allen im Hauptnonogramm geänderten Zellen zurückgegeben.

2.3.6 Ablauf

Der Algorithmus beginnt damit, zuerst Propagate aufzurufen. Wenn dies noch nicht zu einer Lösung führt, wird darufhin FP3 angewendet. In den wenigen Fällen, in denen auch dies noch nicht zur Lösung führt, wird Backtrackinng angewendet. Dabei werden nach jedem Backtrackingschritt Propagate und FP3 erneut angewendet. Außerdem wird eine Kopie der Kontapositionen und NonogrammTabellen gemacht. Wenn sich das Backtracking falsch entschieden hat, werden diese wieder in den Zustand vor dem Backtrackingschritt gebracht.
Der Artikel[3] benutzt 5 verschiedene Heuristiken für das Backtracking. In dieser Arbeit wird allerdings nur die Heuristik Min-logd genutzt, welche nach den Tests des Artikels am schnellsten war. Für diese Heuristik wird für alle leeren Zellen p der Wert h ausgerechnet:
$h = (V_{min}(p) + V_{log}(p, 0) * V_{log}(p, 1)$
mit:
$V_{min}(p) = min(m_{p,0}, m_{p,1})$ und
$V_{log}(p, n) = log(1 + m_{p,n}) + 1.$
Dabei ist $m_{p,n}$ die Anzahl von Zellen, die im Nonogramm durch einen Aufruf von PropeG3 mit den Parametern p und n gefärbt werden. Dies kann der NonogrammTabelle entnommen werden. Dabei wird die Zelle mit dem höchsten Wert h zuerst durch Backtracking ausprobiert.

2.4 Lineare Optimierung

Eine Lineare Optimierung(ILP) besteht aus drei Teilen, einer Funktion, Variablen und Beschränkungen. Die Funktion soll durch die Optimierung miniert beziehungsweise maximiert werden und wird Zielfunktion genannt. Durch die Beschränkungen werden die Werte der Variablen eingrenzt. Ziel ist es, Werte für alle Variablen zu finden, die alle Beschränkungen erfüllen und die Zielfunktion miniert beziehungsweise maximiert.
Konkret wird für diese Arbeit eine Variation der Linearen Optimierung genutzt, welche keine Zielfunktion benötigt, da nur ein Ergebnis gesucht wird, welches alle Beschränkungen erfüllt. Ebenso wurden in beiden Optimierungen nur binäre Variablen verwendet.
Für die Umsetzung wird hier das Python Modul pulp genutzt.

2.4.1 Lineare Optimierung 1

Die erste Optimierung basiert auf dem Artikel „Solving Nonograms Using Integer Programming Without Coloring„[4] von Khan. Sie benutzt Variablen für alle möglichen Startpunkte von Blöcken. Ebenso werden drei Gruppen von Beschränkungen genutzt. Die erste sorgt dafür, dass jeder Block nur eine Startpunkt haben kann. Die zweite sorgt dafür, dass zwischen einen Startpunkt und dem nächsten genug Abstand ist. Die dritte Gruppe kümmert sich um die Interaktion von Zeilen und Spalten.

Variablen

Es wird eine Variable für jeden möglichen Startpunkt von jedem Block erstellt. Ähnlich wie im regelbasierten Algorithmus die initialen Bereiche berechnet wurden, werden hier auch die Startpunkte berechnet.
Der erste mögliche Startpunkt eines Blocks i in Zeile j befindet sich an der Stelle:
$S_{1,1,j} = 0$
$S_{1,i,j} = \sum_{k=1}^{i-1}(LängeBlock_{k,j} + 1)$, für $i > 1$
Der Block kann nicht früher beginnen, da sonst nicht genug Platz für die Blöcke davor ist. Der letzte mögliche Startpunkt befindet sich an der Stelle:
$S_{letzte,i,j} = n - \sum_{k=i+1}^{AnzahlBlöcke_j}(LängeBlock_{k,j} + 1)$,
Ebenso kann der Block nicht später starten, da es sonst nicht genug Platz für die Blöcke danach gibt. Alle Zellen zwischen diesen sind ebenfalls mögliche Startpunkte. Wenn die Startpunkte einer Spalte berechnet werden, wird in der Formel n durch m ersetzt.
Wenn eine der Variablen auf 1 gesetzt ist, beginnt der zugehörige Block in der Zelle der Variable. Ist sie auf 0 gesetzt, beginnt der Block bei einer anderen Variablen.

Beschränkungen 1

Die erste Gruppe von Beschränkungen ist, dass für alle Linien i und Blöcke j in der Linie gilt:
$\sum_{k=1}^{AnzahlVar_{i,j}}(V_{k,i,j}) = 1$,
wobei $V_{k,i,j}$ die Variable des k-ten Starpunkt des i-ten Block der Linie j und $AnzahlVar_{i,j}$ der Anzahl der möglichen Startpunkte für den i-ten Block der Linie j entspricht. Diese Beschränkungen sorgen dafür, das jeder Block genau einen Startpunkt haben muss.

Beschränkungen 2

Die zweite Gruppe von Beschränkungen ist, dass für alle Linien i und Blöcke j in der Linie außer dem letztem Block gilt:
$\sum_{k=1}^{AnzahlVar_{i,j}}(V_{k,i,j} * S_{k,i,j}) + LängeBlock_{i,j} + 2 \leq \sum_{k=1}^{AnzahlVar_{i+1,j}}(V_{k,i+1,j} * S_{k,i+1,j})$,
Der Teil $V_{k,i,j} * S_{k,i,j}$ ist dabei immer gleich 0 außer am gewählten Startpunkt, wo sie gleich dem Startpunkt ist. Dadurch kann man kontrollieren, dass der Startpunkt weit genug vor dem Startpunkt des nächsten Blocks liegt, damit genug Platz für den Block ist.

Beschränkungen 3

Für die dritte Gruppe von Beschränkungen wird das OverlapCheckingSet benötigt. Dies sind zwei Matrizen, welche Einträge für jede Zelle haben. Dabei gibt es eine Matrix für die Zeilen und eine für die Spalten.
In der Matrix der Zeilen ist jeder Eintrag der Matrix eine Liste mit allen Variablen für welche gilt: wenn die Variable gleich 1 ist, dann ist die zugeordnete Zelle der Matrix schwarz gefärbt. Dabei enthält diese Matrix nur Variablen, welche zu Startpunkten von Blöcken in Zeilen gehören. Die Matrix für Spalten ist äquivalent definiert, nur dass in ihr nur Variablen sind, welche zu Startpunkten von Blöcken in Spalten gehören.
Mit dem OverlapCheckingSet lässt sich die letzte Gruppe von Beschränkungen aufstellen. Für alle Zeilen z und alle Spalten s gilt:
$\sum_{k=1}^{AnzahlOCSRow_{z,s}}(OCSRow_{k,z,s}) = \sum_{k=1}^{AnzahlOCSCol_{z,s}}(OCSCol_{k,z,s})$
Dabei ist $AnzahlOCSRow_{z,s}$ die Anzahl von Variablen, die sich im OverlapCheckingSet der Spalten an der Stelle z, k befinden und $OCSRow_{k,z,s}$ die k-te Variable im OverlapCheckingSet der Spalten an der Stelle z, k. Äquivalentes gilt für $AnzahlOCSCol_{z,s}$ und $OCSCol_{k,z,s}$ für das OverlapCheckingSet der Zeilen.
Die Summen in der Formel sind dabei genau dann gleich 1, wenn eine der Variablen gleich 1 ist. Sie können nicht höher als 1 sein, da dies sonst der zweiten Gruppe aus Beschränkungen widersprechen würde.

Mit den Beschränkungen wird immer die Zelle untersucht, in der sich z und s überschneiden. Wenn dabei die erste Summe gleich 1 ist, bedeutet dies, dass in der Zeile ein Startpunkt ausgewählt wurde und die Zelle schwarz gefärbt wird. Da eine Zelle nicht nur von einem Block aus einer Zeile gefärbt werden kann, muss sie auch durch einen Block aus einer Spalte schwarz gefärbt werden. Dafür muss auch die zweite Summe gleich 1 sein.
Somit ist die Beschränkung genau dann erfüllt, wenn die Zelle sowohl durch eine Zeilen-Variable als auch durch eine Spalten-Variable schwarz gefärbt wird oder die Zelle von beiden nicht schwarz gefärbt wird.
Die Lösung des Nonogramms erhält man, indem die Optimierung gelöst wird und dann alle gewählten Startfelder, sowie die Längen der Blöcke danach schwarz gefärbt werden. Der Rest wird weiß gefärbt.

2.4.2 Lineare Optimierung 2

Die Grundidee dieser Optimierung stammt von Prof. Dr. Blankertz. Die weiteren Ausführungen stammen von mir. Die Optimierung basiert darauf, jede mögliche Färbung einer Linie zu generieren und eine Variable für jede dieser Färbungen zu nutzen. Für die Optimierungen gibt es zwei Gruppen von Beschränkungen. Die erste sorgt dafür, dass nur eine Färbung für jede Linie genutzt wird. Die zweite sorgt dafür, dass nicht Färbungen für eine Spalte und Zeile genutzt werden, welche die gemeinsame Zelle unterschiedlich färben.

Variablen

Es wird damit begonnen, für jede Linie Variablen für jede mögliche Färbung zu generieren. Dies passiert mit einer simplen rekursiven Funktion. Die Funktion bekommt die Anzahl an Zellen, die noch gefüllt werden müssen und welche Bedingungen in diesem Bereich noch erfüllt werden müssen, übergeben. Anschließend wird der Block für die erste Bedingung an die ersten Stellen gesetzt und die wird Funktion mit der restlichen Länge und einer Bedingung weniger aufgerufen. Diese generiert alle Möglichkeiten für den Fall, dass der Block an der ersten Stelle steht. Anschließend wird eine 0 in die erste Zelle gesetzt und der Block beginnt in der zweiten Zelle und die Funktion wird erneut mit veränderten Parametern aufgerufen. Dies wird mit immer mehr Nullen wiederholt, bis nicht mehr genug Platz für alle Blöcke ist. Anschließend werden alle Rückgabewerte der Aufrufe zurückgegeben und eine Variable für alle Färbungen angelegt.

Beschränkungen 1

Die erste Gruppe von Beschränkungen sorgt dafür, dass in jeder Linie genau eine Färbung genutzt wird. Dafür werden alle Variablen einer Linie summiert und diese Summe muss gleich 1 sein.

Beschränkungen 2

Die zweite Gruppe sorgt dafür, dass zwei Variablen eine Zelle nicht unterschiedlich färben. Dafür muss bei jeder Kombination von einer Färbung für eine Zeile und einer Färbung für eine Spalte überprüft werden, ob sie in der Zelle, in der sie sich überschneiden, den gleichen Wert haben. Ist dies der Fall, passiert nichts. Andernfalls wird eine neue Beschränkung angelegt, welche die beiden zugehörigen Variablen addiert und ≤ 1 setzt. Dadurch kann nur maximal eine der beiden Variablen auf 1 gesetzt werden und sie können somit nicht gleichzeitig gelten.
Diese Optimierung benutzt mehr Beschränkungen als die erste. Allerdings sind diese dafür kürzer, was sich eventuell positiv auf die Laufzeit der Lösung auswirken kann. Die Lösung des Nonogramms erhält man, indem alle Linien genauso gefärbt werden, wie die von der Optimierung gewählten Variablen.

3 Ergebnisse

Um die Algorithmen zu vergleichen, wurden 100 Nonogramme von janko.at [5] genutzt. Dabei wurden in den Größen 10X10, 15X15, 20X20, 25X25 und 30X30 jeweils 20 Nonogramme getestet. Genutzt wurden immer die ersten 20 Nonogramme der jeweiligen Größen. Der genutzte Computer hat einen 12th Gen Intel(R) Core(TM) i7-12700k, 3600 Mhz Prozessor und 32 GB RAM. Die Implementierungen und die Ergebnisse für alle Nonogramme wurden auf GitLab [1] hochgeladen.
Wenn ein Algorithmus ein Nonogramm nach 10 Minuten nicht lösen konnte, wurde er abgebrochen (Timeout). Diese Nonogramme werden in den weiteren Ausführungen mit einer Zeit von 600 Sekunden eingerechnet.
Dabei wird in den Graphen eine logarithmische Skala genutzt. Als Ausreißer sind jeweils die kürzeste und längste Laufzeit eines Algorithmus in einer Größe definiert.

[1] https://git.tu-berlin.de/maltekriese/bachelorarbeit-malte-kriese

3.1 einfacher Backtracking Algorithmus

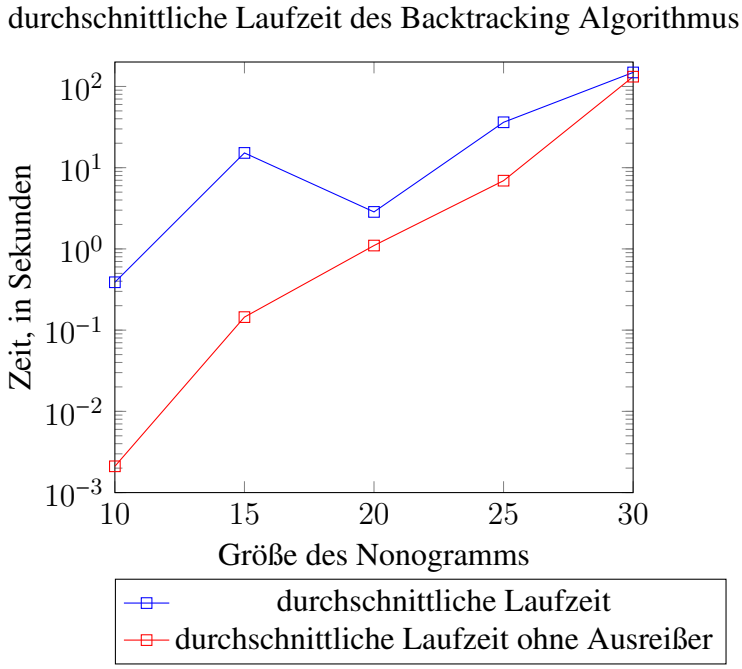

Abbildung 3.1: durchschnittliche Laufzeit des Backtracking Algorithmus

Der einfache Backtracking Algorithmus war der zweitlangsamste. Im Gegensatz zu den schnelleren Algorithmen hatte er schon bei 25X25 Nonogrammen einen Timeout. Insgesamt hatte dieser Algorithmus 5 Timeouts. Sehr auffällig ist, dass die 20X20 Nonogramme im Durchschnitt schneller gelöst wurden als die 15X15 Nonogramme. Nach Entfernung der Ausreißer ist dies nicht mehr der Fall.

Größe	kürzeste Laufzeit	längste Laufzeit	Durchschnitt	Standardabweichung
10X10	0.00000 sec	7.73336 sec	0.38856 sec	1.68501 sec
15X15	0.00100 sec	301.23758 sec	15.19214 sec	65.62408 sec
20X20	0.00598 sec	37.25270 sec	2.85297 sec	8.37223 sec
25X25	0.00694 sec	600.00000 sec	36.22657 sec	130.75195 sec
30X30	0.01694 sec	600.00000 sec	148.75046 sec	240.92550 sec

Tabelle 3.1: Daten des einfachen Backtracking Algorithmus

Wenn man sich nur die kürzesten Laufzeiten anschaut, ist der Algorithmus schneller als die erste Optimierung und der regelbasierte Algorithmus. Im Durchschnitt sind diese Algorithmen bei allen getesteten Größen allerdings schneller.

3.2 Regelbasierter Algorithmus

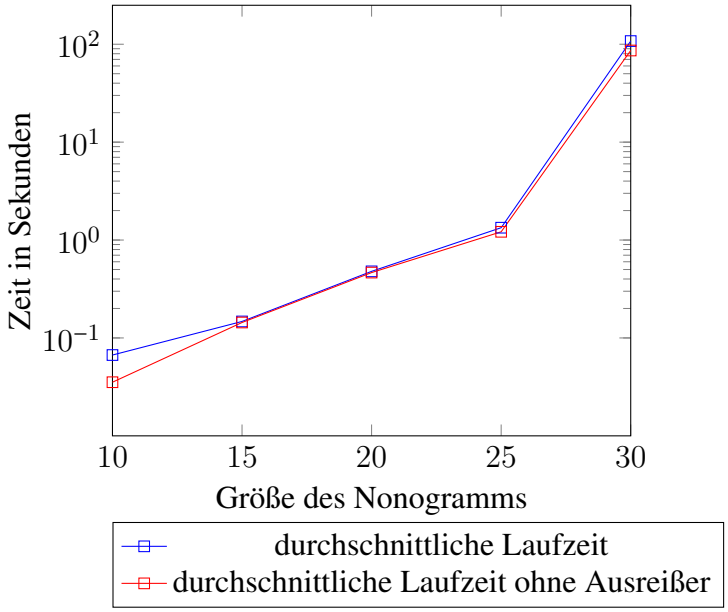

Abbildung 3.2: durchschnittliche Laufzeit des regelbasierten Algorithmus

Der regelbasierte Algorithmus konnte die meisten Nonogramme ohne Probleme lösen. Allerdings gab es drei Timeouts bei den 30X30 Nonogrammen. Er war der drittschnellste der getesteten Algorithmen.

Größe	kürzeste Laufzeit	längste Laufzeit	Durchschnitt	Standardabweichung
10X10	0.02791 sec	0.67474 sec	0.06691 sec	0.13950 sec
15X15	0.10664 sec	0.25216 sec	0.14735 sec	0.03468 sec
20X20	0.30199 sec	0.90099 sec	0.47880 sec	0.15190 sec
25X25	0.75647 sec	4.12520 sec	1.33648 sec	0.82834 sec
30X30	1.52291 sec	600.00000 sec	107.7302 sec	211.24833 sec

Tabelle 3.2: Daten des regelbasierten Algorithmus

3.3 Dynamischer Algorithmus

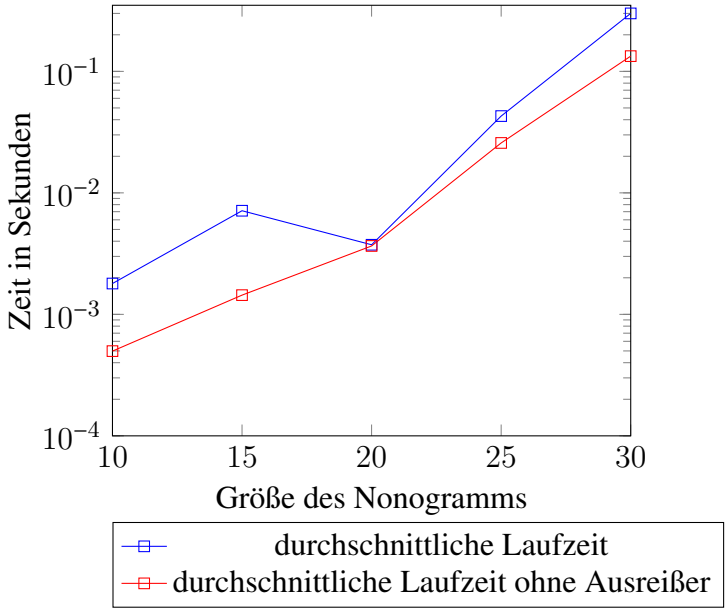

Abbildung 3.3: durchschnittliche Laufzeit des dynamischen Algorithmus

Der dynamische Algorithmus konnte alle Nonogramme ohne Probleme lösen. Dabei war er der schnellste aller Algorithmen und der einzige, der keinen Timeout hatte. Sehr auffällig ist, dass die 20X20 Nonogramme im Durchschnitt schneller gelöst wurden als die 15X15 Nonogramme. Das gleiche konnte man auch beim Backtracking Algorithmus beobachten. Auch hier ist dies nach Entfernung der Ausreißer nicht mehr der Fall.

Größe	kürzeste Laufzeit	längste Laufzeit	Durchschnitt	Standardabweichung
10X10	0.00000 sec	0.02691 sec	0.00179 sec	0.00578 sec
15X15	0.00100 sec	0.11561 sec	0.00713 sec	0.02490 sec
20X20	0.00100 sec	0.00797 sec	0.00374 sec	0.00169 sec
25X25	0.00199 sec	0.39169 sec	0.04286 sec	0.10456 sec
30X30	0.00602 sec	3.59199 sec	0.30005 sec	0.81748 sec

Tabelle 3.3: Daten des dynamischem Algorithmus

Der dynamische Algorithmus war ebenfalls der einzige Algorithmus, der eine bessere kürzeste Laufzeit als der einfache Backtracking Algorithmus hatte.

3.4 Lineare Optimierung 1

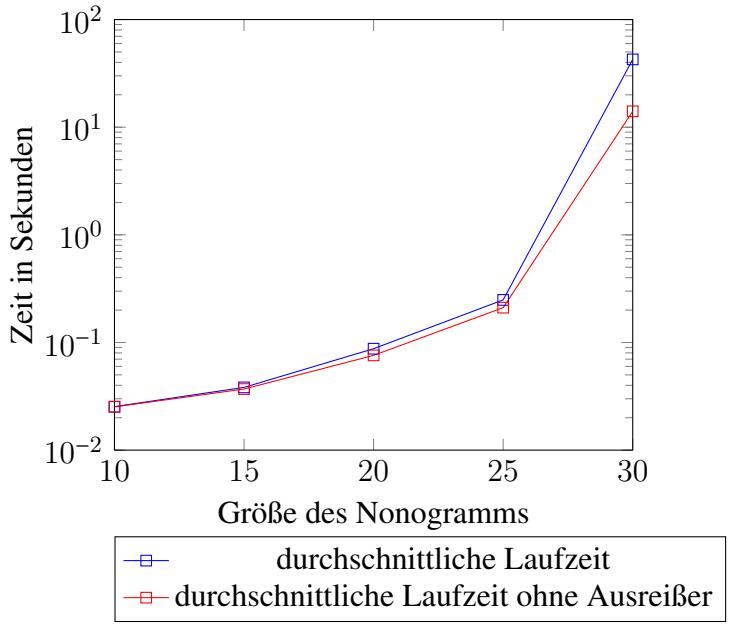

Abbildung 3.4: durchschnittliche Laufzeit der ersten linearen Optimierung

Die erste lineare Optimierung konnte alle Nonogramme, bis auf ein 30X30 Nonogramm, ohne Probleme lösen. Sie war die zweitschnellste Lösungsmöglichkeit.

Größe	kürzeste Laufzeit	längste Laufzeit	Durchschnitt	Standardabweichung
10X10	0.02193 sec	0.03189 sec	0.02537 sec	0.00259 sec
15X15	0.02791 sec	0.07276 sec	0.03827 sec	0.01000 sec
20X20	0.04186 sec	0.34285 sec	0.08756 sec	0.06692 sec
25X25	0.05183 sec	1.12819 sec	0.24862 sec	0.23399 sec
30X30	0.10266 sec	600.00000 sec	42.64910 sec	138.27232 sec

Tabelle 3.4: Daten der ersten linearen Optimierung

Größe	Anzahl Variablen	Anzahl Beschränkungen	Länge Beschränkungen
10X10	182.15	167.4	4.36932
15X15	446.35	336.0	6.24514
20X20	874.2	589.4	8.15138
25X25	1605.2	863.9	10.62276
30X30	2741.6	1264.6	12.97798

Tabelle 3.5: Größe der ersten linearen Optimierung

Dabei hatte die erste Optimierung weniger Variablen und Beschränkungen als die zweite Optimierung. Allerdings waren diese Beschränkungen im Durchschnitt länger.

3.5 Lineare Optimierung 2

Die zweite Lineare Optimierung war bei weitem am langsamsten. Sie hatte schon bei den 15X15 Nonogrammen 5 Timeouts und bei den 20X20 14 Timeouts. Nach den 20X20 Nonogrammen wurde das Testen abgebrochen, da eindeutig war, dass die Optimierung nicht praktisch zum Lösen von Nogrammen ist.

Größe	kürzeste Laufzeit	längste Laufzeit	Durchschnitt	Standardabweichung
10X10	0.04286 sec	400.54615 sec	21.79529 sec	87.14698 sec
15X15	0.29900 sec	600.00000 sec	152.44638 sec	258.41129 sec
20X20	2.26243 sec	600.00000 sec	440.73952 sec	250.50126 sec

Tabelle 3.6: Daten der zweiten linearen Optimierung

Diese Daten wären wahrscheinlich noch schlechter, wenn nicht mit einem Timeout von 10 Minuten gearbeitet worden wäre. Dies ist allerdings für diese Arbeit nicht relevant, da so schon erkennbar ist, dass es sich bei der Optimierung um den langsamsten Algorithmus handelt.

Größe	Anzahl Variablen	Anzahl Beschränkungen	Länge Beschränkungen
10X10	306.75	12614.35	2.03075
15X15	1816.6	545454.1	2.00838
20X20	8213.8	10726620.15	2.00276

Tabelle 3.7: Größe der zweiten linearen Optimierung

Die Beschränkungen der zweiten Optimierung sind um einiges kleiner als die der ersten Optimierung. Dies liegt daran, dass ein großer Teil der Beschränkungen zur zweiten Gruppe von Variablen gehören und daher nur 2 Variablen enthalten. Allerdings gibt es um einiges mehr Variablen und Beschränkungen als bei der ersten Optimierung.

4 Diskussion

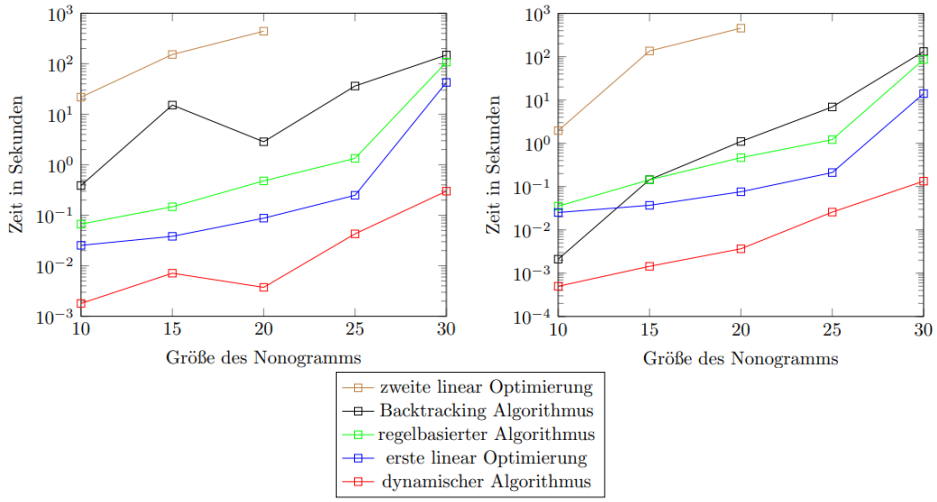

Abbildung 4.1: durchschnittliche Laufzeit aller Algorithmen, mit (links) und ohne (rechts) Ausreißer

Die Ergebnisse zeigen eindeutig, dass der dynamische Algorithmus bei weitem der Beste ist. Insgesamt gab es nur 7 Nonogramme, die der dynamische Algorithmus nicht als schnellster Algorithmus lösen konnte. Davon wurden 2 Nonogramme schneller durch die erste Optimierung gelöst und 6 Nonogramme schneller durch den einfachen Backtracking Algorithmus. Ein Nonogramm wurde sowohl durch die erste Optimierung als auch durch den Backtracking Algorithmus schneller gelöst.

4.1 Dynamischer Algorithmus

Dass die 20X20 Nonogramme durchschnittlich schneller gelöst wurden als die 15X15, liegt an den gewählten Nonogrammen. Bei allen Größen, außer 20X20, gibt es meistens nur wenige Nonogramme (Ausreißer), welche die durchschnittliche Laufzeit stark erhöht haben. Bei den 20X20 gibt es keine solche Nonogramme. Dies sieht man auch am Graphen ohne Ausreißer. Würde man die Anzahl der gewählten Nonogramme erhöhen, würde sich wahrscheinlich auch ein 20X20 Nongrammen mit langer Laufzeit darunter befinden. Der Graph würde dann wahrscheinlich ähnlich zu dem des regelbasierten Algorithmus und der lineare Optimierung 1 aussehen.

4.2 Regelbasierter Algorithmus

Der regelbasierte Algorithmus hat das Problem, dass bei Anwendung aller Regeln auf eine Zeile, maximal genauso viele Zelle gefärbt werden können, wie bei Anwendung von Paint auf eine Zeile. Allerdings wird Paint vom dynamischen Algorithmus um einiges effizienter angewendet. Es wird nur angewendet, wenn es Änderungen in einer Linie gab. Der regelbasierte Algorithmus hat immer wieder alle Regeln auf alle Zeilen angewendet, inklusive bereits gelöster. Hierdurch war er um einiges langsamer. Dies könnte verbessert werden, indem die Regeln nur angewendet werden, wenn sich etwas an einem Block geändert hat. Das könnte zum Beispiel realisiert werden, indem für jede Regel gespeichert wird, welche Blöcke sich seit dem letzten Aufruf verändert haben. Ebenso könnte die Probe Funktion des dynamischen Algorithmus auch für den regelbasierten Algorithmus genutzt werden, um ihn weiter zu beschleunigen.

4.3 Backtracking Algorithmus

An den Daten kann eindeutig gesehen werden, dass die Ergebnisse stark vom gewählten Nonogramm abhängen. Dies liegt daran, dass es sehr unterschiedlich lange dauern kann, bis auf einen Widerspruch gestoßen wird. Der Algorithmus kann sehr lange suchen, nur um immer wieder in den letzten Zellen einen Widerspruch zu finden und anschließend an einer der ersten Stellen erneut anfangen. Ebenso kann er sehr schnell einen Widerspruch finden und somit sehr schnell die richtige Färbung finden. Deswegen ist ein solcher Algorithmus nicht sinnvoll für größere Nonogramme. Er kann zwar sehr schnell sein, ist aber in vielen Fällen besonders langsam.
Dass die 20X20 Nonogramme schneller als die 15X15 Nonogramme sind, liegt ebenso wie bei dem dynamischen Algorithmus an den gewählten Nonogrammen.

4.4 Lineare Optimierung 1

Die erste lineare Optimierung hat gut funktioniert. Allerdings war sie langsamer als der dynamische Algorithmus. Dies war zu erwarten, da der dynamische Algorithmus auf die Lösung von Nonogrammen angepasst ist und somit ihre Eigenschaften ausnutzen konnte. Dies konnte der Algorithmus zu Lösung der Optimierung nicht, da er darauf ausgelegt ist allgemein Optimierungen zu lösen.

4.5 Lineare Optimierung 2

Die zweite lineare Optimierung hat sehr schlecht funktioniert. Zwar waren die Beschränkungen wie geplant kürzer als bei der ersten Optimierung, dafür gab es aber auch viel mehr von ihnen. Für die 10X10 Nonogramme hat sich die Länge der Variablen halbiert, allerdings wurde die Anzahl von Beschränkungen fast verhundertfacht. Diese hohe Anzahl von Bedingungen hat den Vorteil durch die kürzeren Bedingungen überschattet, wodurch die Laufzeit sehr lang war. Daher ist dieser Ansatz nicht sinnvoll.

5 Fazit

Die Frage, welcher Algorithmus am schnellsten ist, lässt sich sehr leicht beantworten. Am schnellsten war der dynamische Algorithmus, gefolgt von der ersten linearen Optimierung und dem regelbasiertem Algorithmus. Der einfache Backtracking Algorithmus ist nicht praktisch zur Lösung von großen Nonogrammen. Und die zweite Optimierung funktioniert noch nicht einmal bei kleinen Nonogrammen gut.
Eine mögliche künftige Forschungsfrage ist, welcher der Algorithmen der beste für farbige Nonogramme wäre. Alle Algorithmen lassen sich relativ simpel umbauen, damit sie farbige Nonogramme lösen können. Die erste Optimierung ist sogar dafür gemacht. Daher stellt sich die Frage, ob diese in der Lage wäre, farbige Nonogramme schneller als der dynamische Algorithmus zu lösen.
Eine weitere Frage ist, wie weit sich der regelbasierte Algorithus verbessern würde, wenn man die vorgeschlagenen Änderungen implementiert.

Literaturverzeichnis

[1] Nobuhisa Ueda and Tadaaki Nagao. Np-completeness results for nonogram via parsimonious reductions. *preprint*, 1996.

[2] Chiung-Hsueh Yu, Hui-Lung Lee, and Ling-Hwei Chen. An efficient algorithm for solving nonograms. *Applied Intelligence*, 35:18–31, 2011.

[3] I-Chen Wu, Der-Johng Sun, Lung-Ping Chen, Kan-Yueh Chen, Ching-Hua Kuo, Hao-Hua Kang, and Hung-Hsuan Lin. An efficient approach to solving nonograms. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(3):251–264, 2013.

[4] Kamil A Khan. Solving nonograms using integer programming without coloring. *IEEE Transactions on Games*, 14(1):56–63, 2020.

[5] janko.at. https://www.janko.at/Raetsel/Nonogramme/index-2.htm. Accessed: 1.12.2024.