Generieren von eindeutig lösbaren japanischen Pencil Puzzle Instanzen mithilfe von Constraint Programming

Bachelorarbeit

Maximilian Block
411898

14. Juli 2024

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

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

Kurzfassung

Neighbor Number von Naoki Inaba ist ein japanisches Paper Puzzle, bei dem Ziffern auf einem n×n Brett platziert werden müssen, sodass gewisse Regeln gelten. In dieser Arbeit werden mögliche Ansätze zum Generieren solcher Instanzen getestet und verglichen. Dabei wird Constraint Programming zum Lösen der Instanzen verwendet, da es dem menschlichen Lösen von solchen Rätseln näher kommt als reine Suchalgorithmen. So kann die gewünschte Schwierigkeit eines Rätsels schon während der Generierung beachtet werden. Am Ende stellt sich heraus, dass diverse Ansätze, je nach gewünschter Größe und Schwierigkeit des Rätsels, besser sind als die anderen.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Neibānanbā/Neighbor Number . . . 1
- 1.2 Literatur . . . 1
- 1.3 Ziel der Arbeit . . . 2
- 1.4 Struktur der Arbeit . . . 2

2 Methoden **3**
- 2.1 Constraint Programming . . . 3
  - 2.1.1 Propagation . . . 3
- 2.2 Neighbor Number Solver . . . 4
  - 2.2.1 Minimale Constraints . . . 5
  - 2.2.2 Redundante Constraints . . . 5
  - 2.2.3 Verschiedene Solver . . . 11
- 2.3 Puzzle-Generierung . . . 11
  - 2.3.1 Erste Lösung generieren . . . 12
  - 2.3.2 Hinweise entfernen oder hinzufügen . . . 12
  - 2.3.3 Top-Down systematische Suche . . . 13
  - 2.3.4 Bottom-Up systematische Suche . . . 14
  - 2.3.5 Top-Down probabilistische Suche mit einem Ast . . . 14
  - 2.3.6 Top-Down probabilistische Suche mit mehreren Ästen . . . 15
  - 2.3.7 Bottom-Up probabilistische Suche . . . 16

3 Ergebnisse **18**

4 Diskussion **25**
- 4.1 Interpretation der Ergebnisse . . . 25
  - 4.1.1 Knoten pro Sekunde . . . 25
  - 4.1.2 Vergleich der systematischen Algorithmen . . . 26
  - 4.1.3 Iterationszahl bei probabilistischen Algorithmen . . . 26
- 4.2 Resultat . . . 27
- 4.3 Vergleich mit bestehender Literatur und Limitation . . . 27

5 Fazit **28**

Abbildungsverzeichnis

1.1 Offizielles Beispiel von Naoki Inaba 1

2.1 Domain nach Aufruf des Propagators des OnlyThree-Constraints 5
2.2 Der Wert Vier begrenzt den Wertebereich der Felder nach oben hin. Hier wird der maximale Wert jeweils in grau dargestellt. Der minimale Wert wird in der Grafik nicht betrachtet. 6
2.3 Die Sechs führt dazu, dass die Differenz zwischen den äußeren Zahlen nicht dem Abstand entspricht. 6
2.4 Beispiel: Der minimale Wert liegt über dem maximalen Wert. 7
2.5 Domain nach Aufruf des Propagators des MinMax-Constraints 8
2.6 Beispiel: Parität pro Feld in einer Zeile 8
2.7 Beispiel: Parität führt zu einem Feld, das frei bleiben muss 9
2.8 Domain nach Propagation des OddEven-Constraints 10
2.9 Domain nach Propagation der OddEven-, MinMax- und NotEqual-Constraints 10
2.10 Einige Knoten aus einem Suchbaum mit dem leeren Brett als Wurzel 12

Tabellenverzeichnis

3.1 25 5x5 Puzzles je Algorithmus und Schwierigkeit wurden generiert 18
3.2 zehn 6x6 Puzzles je Algorithmus und Schwierigkeit wurden generiert 19
3.3 zwei 7x7 Puzzles je Algorithmus und Schwierigkeit wurden generiert 20
3.4 Überprüfte Knoten pro Sekunde je Algorithmus und Schwierigkeit 21
3.5 Vergleich der systematischen Algorithmen 22
3.6 Auswirkung der Iterationszahl auf Probabilistische Suche 24

1 Einleitung

1.1 Neibānanbā/Neighbor Number

„Neibānanbā“ zu englisch auch „Neighbor Number“ ist ein japanisches Pencil Puzzle, welches von Naoki Inaba entwickelt wurde.[1] Ziel des Puzzles ist es, alle Felder des quadratischen Brettes mit Zahlen zu füllen oder frei zu lassen, sodass folgende Regeln eingehalten werden:

- In jeder Zeile und Spalte sollen genau drei Felder mit Zahlen von 1-9 gefüllt werden.
- Die Differenz zweier Ziffern, zwischen denen sich keine andere Ziffer befindet, soll genau der Anzahl an Gitterlinien, die zwischen ihnen liegen, entsprechen. Alternativ kann hier auch der Abstand zwischen den Feldern gezählt werden.
- Keine Ziffer darf innerhalb einer Zeile oder Spalte doppelt vorkommen.

Abbildung 1.1 zeigt das offizielle Beispiel von Naoki Inaba. Dort wurden leer bleibende Felder mit einem Kreuz markiert.

1			
		2	
	4		
			3

1	2	3	×
×	3	2	1
3	4	×	2
2	×	4	3

Abbildung 1.1: Offizielles Beispiel von Naoki Inaba

1.2 Literatur

Da ich keine Literatur zu Neighbor Number finden konnte, griff ich auf Ansätze zurück, die für das Generieren von Sudoku-Instanzen genutzt wurden, weil sich die Rätsel in einigen Punkten ähneln. In beiden Fällen müssen Ziffern auf einem $n \times n$ platziert werden, sodass gewisse Regeln gelten. Das Paper [6] beschreibt ausführlich, wie Sudoku als Constraint Satisfaction Problem modelliert werden kann und welche Vorteile redundante Constraints beim Lösen haben. Im Werk [5] wurde auch veranschaulicht, wie Sudoku als Constraint Satisfaction Problem modelliert werden kann. Jedoch wird

hier versucht, das menschliche Lösen zu simulieren und somit den Schwierigkeitsgrad für Menschen einschätzen zu können. Dabei wird ein Ansatz diskutiert, bei dem die Schwierigkeit bereits in den Generierungsprozess mit einfließt und nicht erst nach der Generierung überprüft wird. In den Werken [3] und [4] wird Constraint Programming allgemein erklärt.

1.3 Ziel der Arbeit

Das Ziel der Arbeit ist das Generien von möglichst interessanten Instanzen des Rätsels Neighbor Number in diversen Schwierigkeitsstufen. Dabei werden verschiedene Ansätze beim Hinzufügen oder Entfernen von Hinweisen verglichen, bei denen die gewählte Schwierigkeit bereits einen Einfluss hat.

1.4 Struktur der Arbeit

Zu Beginn wird Constraint Programming allgemein vorgestellt und erläutert. Insbesondere wird dabei auf Inferenz oder Propagation eingegangen. Anschließend werden das Modell und die genutzten Constraints betrachtet, mit denen die Solver/Lösungsalgorithmen arbeiten. Außerdem wird erläutert, wie verschiedene Varianten des Solver genutzt werden können, um verschieden schwere Puzzle-Instanzen zu erstellen. Des Weiteren wird erklärt, wie die Puzzle Instanzen generiert werden. Dabei werden verschiedene Ansätze erläutert. Zuletzt folgen die Ergebnisse und eine Auswertung und Interpretation dieser generierten Daten.

2 Methoden

Im folgenden Kapitel werden die genutzten Methoden und Ansätze beim Lösen und Generieren von Neighbor-Number-Instanzen vorgestellt.

2.1 Constraint Programming

Constraint Programming [3, 4] ist ein Programmierparadigma zum Lösen von kombinatorischen Problemen. Eine wichtige Rolle spielen dabei die sogenannten Constraints, welche Relationen zwischen den Werten der Entscheidungsvariablen darstellen. Constraints werden genutzt, um zu spezifizieren, welche Kombinationen an Werten der Entscheidungsvariablen zu einer Lösung gehören können und welche nicht. So kann zum Beispiel gefordert werden, dass zwei Variablen nicht den gleichen Wert haben dürfen oder der Wert der einen größer sein muss, als der der anderen Variable.
Ein Constraint Satisfaction Problem besteht im Wesentlichen aus einer Menge an Entscheidungsvariablen mit jeweils einem Wertebereich/Domain und den Constraints, welche jeweils die Werte einer Menge an Entscheidungsvariablen einschränken.
Eine Lösung eines Constraint Satisfaction Problems ist eine Belegung der Entscheidungsvariablen mit Werten, sodass alle Constraints erfüllt sind, bzw. kein Constraint nicht erfüllt ist. Das Lösen eines Constraint Satisfaction Problems wird oft mit Hilfe von Backtracking Algorithmen durchgeführt. Üblich ist hierbei die Tiefensuche, da diese möglichst schnell eine Lösung finden sollte. Durch systematische Suche kann aber auch der gesammte Suchbaum durchsucht werden, um alle Lösungen zu finden oder zu gewährleisten, dass keine Lösung existiert.
Die Constraints können hierbei genutzt werden, um den Suchbaum zu beschneiden. Dazu dient die Constraint Propagation.

2.1.1 Propagation

Constraint Propagation ist das Beschneiden des Suchbaums mit Hilfe der Constraints. Es kann zum Beispiel schon vor dem Belegen aller Variablen erkannt werden, dass der aktuell durchsuchte Ast des Suchbaums nicht zu einer Lösung führt und somit frühzeitig Backtracking durchgeführt werden kann. Eine andere Möglichkeit ist das Entfernen von Werten aus dem Wertebereich einer Variable, sofern bereits bekannt ist, dass diese Werte nicht zu einer Lösung gehören können.

Die einfachste Form der Propagation ist das Forward Checking. Forward Checking wird nach dem Instanziieren einer Variable zu einem Wert durchgeführt. Dabei werden die Werte, die aufgrund der Constraints nicht mehr zu einer Lösung gehören können, aus dem Wertebereich von anderen Variablen entfernt. Falls zum Beispiel ein Constraint über zwei Variablen x und y mit jeweils einem Wertebereich von $1..5$ verlangt, dass $x > y$ gilt, kann nach dem Instanziieren von x zu dem Wert 3 der Wertebereich von y reduziert werden. Da x größer als y sein muss und x nun den Wert 3 besitzt, würde

ein Instanziieren von y zum Wert 4 den Constraint nicht mehr erfüllen. Dieser Wert kann somit aus dem Wertebereich entfernt werden, da er nicht mehr zu einer Lösung gehören kann. Durch Forward Checking werden alle Werte, die durch die aktuellen Werte der Variablen nicht mehr zu einer Lösung gehören können, entfernt. Dies verkleinert den Suchbaum, da so Äste des Suchbaums abgeschnitten werden, bei denen die Variablen die entfernten Werte annehmen würden.

Außerdem kann durch Forward Checking der Fall auftreten, dass der Wertebereich einer Variable gar keinen Wert mehr enthält. Falls dies geschieht kann sofort Backtracking eingeleitet werden, da das weitere Explorieren des Astes auch nicht mehr zu einer Lösung führen kann.

Auch kann bei Forward Checking der Fall auftreten, dass genau ein Wert im Wertebereich einer Variable übrig bleibt. In diesem Fall kann dieser Wert direkt eingesetzt werden. Dieses Einsetzen kann wiederum durch Forward Checking zu weiteren instanziierten Variablen führen oder dazu führen, dass ohne weitere Suche ein Deadend gefunden wird.

Forward Checking hat für diese Arbeit einer große Bedeutung. Manche der in 2.2.3 vorgestellten Solver versuchen, ganz ohne Suche und nur durch Forward Checking Rätsel zu lösen.

Mehr zu Propagation oder Constraint Programming allgemein ist in den Werken [4, 3] zu finden.

2.2 Neighbor Number Solver

Der Solver wurde in Java mit dem ChocoSolver [2] implementiert.
Da Neighbor Number aus einem $N \times N$ Brett aus Kacheln besteht, werden die Entscheidungsvariablen als $N \times N$ Array von Variablen mit einer Domain von $0 - 9$ dargestellt. Dabei repräsentiert die Null das freie Feld.

Außerdem können alle Instanzen als String dargestellt werden. Dies ist hilfreich, um Rätsel einfacher zwischenzuspeichern und zu vergleichen.
Dieser Boardstring besteht aus der Kantenlänge des Brettes, gefolgt von einem „$*$“ zum leichteren Zerteilen des Strings. Dann folgen alle platzierten Werte im Format „x, y, v“, wobei x und y die Koordinaten bilden und v der Wert des Feldes ist. Diese werden von einem „;“ getrennt. Dabei sind die platzierten Werte immer nach den Koordinaten sortiert, um sie einfacher vergleichbar zu machen.

Die Klasse Namens „BoardStringHelper“ enthält nützliche Methoden, um Informationen über ein Brett zu erhalten, Werte hinzuzufügen oder zu entfernen.

Um verschieden schwere Instanzen zu generieren, werden verschiedene Solver benutzt. Dies beruht auf dem Ansatz, der in der Quelle [5, S.353] erläutert wird. Ich habe einen Solver pro Schwierigkeitsstufe entwickelt. Sie unterscheiden sich in den genutzten Constraints und verwenden keine Suchstrategie.

Wenn ein Solver einer Schwierigkeitsstufe nicht in der Lage ist, ein Rätsel nur durch Forward Checking zu lösen, ist das Rätsel entweder zu schwer und erfordert Strategien, die der Solver nicht hat, oder es

ist einfach nicht ohne Raten oder systematische Suche lösbar.

2.2.1 Minimale Constraints

Die folgenden Constraints sind in jedem Solver vorhanden und stellen sicher, dass jede Belegung der Entscheidungsvariablen auch die Regeln des Rätsels respektiert.

Der „OnlyThree“-Constraint gilt als erfüllt, wenn die Anzahl der Nullen pro Spalte und Zeile genau $n - 3$ entspricht. Damit muss jede Zeile und Spalte genau 3 Ziffern enthalten. Dies wurde mit dem ChocoSolver mit Hilfe des „count“-Constraints umgesetzt. Der „count“-Constraint zählt das Vorkommen eines Wertes in einer Menge an Variablen und vergleicht, ob es einem bestimmten Betrag entspricht. Gleichung 2.1 muss für jede Zeile und Spalte gelten. Dabei sei n die Kantenlänge des Bretts und $count$ die Anzahl an Nullen in der jeweiligen Zeile oder Spalte.

Abbildung 2.1 zeigt ein Beispiel in einer Zeile, bei dem bereits drei von sechs Feldern eine Null enthalten, und somit leer sind. Da noch genau drei weitere Felder existieren, müssen sie alle zwangsläufig eine Ziffer von $1..9$ enthalten. Deshalb wurde die Null aus dem Wertebereich der übrigen Variablen durch Forward Checking entfernt.

$$count = n - 3 \tag{2.1}$$

0	0	1 2 3 4 5 6 7 8 9	0	1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9

Abbildung 2.1: Domain nach Aufruf des Propagators des OnlyThree-Constraints

Der „NumberDifference“-Constraint erzwingt, dass die Differenz zwischen zwei Ziffern, zwischen welchen nur freie Felder oder auch keine Felder liegen, auch der Anzahl an Gitterlinien zwischen ihnen entsprechen muss. Außerdem dürfen diese Ziffern nicht den gleichen Wert haben.

Mit diesen Constraints und einem einfachen Backtracking-Algorithmus würden sich alle Instanzen des Rätsels lösen lassen.

2.2.2 Redundante Constraints

Wie in Quelle [6, S. 16] geschildert, kann das Verwenden von redundanten Constraints, die Deduktionsfähigkeit von Solvern verbessern. Das verbessert einerseits die Performance des Solvers, da so durch Propagation der Suchbaum weiter verkleinert wird, was beim Finden der ersten Lösung zum Generieren der Instanzen von Vorteil ist. Andererseits kann man so auch weitere, vom menschlichen Spieler genutze Strategien implementieren, um den Schwierigkeitsgrad eines Rätsels zu bewerten.

1. Die erste Strategie, die als redundanter Constraint implementiert wurde, betrachtet den minimalen und maximalen Wert, den ein Feld haben kann. Wird eine Zahl in ein Feld platziert, so beschränkt sie alle anderen Felder der Zeile oder Spalte nach oben und unten.

Angenommen eine Vier wird in ein Feld platziert, dann kann in ein Feld neben der Vier entweder eine Fünf oder eine Drei platziert werden. Ein Feld weiter könnte eine Sechs oder eine Zwei platziert werden. Jedoch könnte keine Sieben in dieses Feld.

Abbildung 2.2: Der Wert Vier begrenzt den Wertebereich der Felder nach oben hin. Hier wird der maximale Wert jeweils in grau dargestellt. Der minimale Wert wird in der Grafik nicht betrachtet.

In 2.2 wird der maximale Wert für alle Felder in Abhängigkeit zur Vier gezeigt. Jedoch muss in dem letzten Feld nicht zwingend eine Neun platziert werden, falls es nicht leer bleibt. Das Platzieren der Sechs in 2.3 kann dazu führen, dass der Wert des letztens Feldes weder dem maximalen noch dem minimalen Wert entsprechen muss. Daher sind es auch nur obere und untere Schranken. Diese begrenzen den Wert eines Feldes, das nicht frei bleibt, somit auf den Intervall zwischen ihnen.

Abbildung 2.3: Die Sechs führt dazu, dass die Differenz zwischen den äußeren Zahlen nicht dem Abstand entspricht.

Jedoch kann es sein, dass bei einem Feld durch eine Zahl in ihrer Spalte der minimale Wert größer sein kann, als der maximale Wert, der durch eine andere Zahl in ihrer Zeile vorgeben wird. In diesem Fall muss dieses Feld frei bleiben.

In Abbildung 2.4 müssen die grauen Felder leer sein. Das Feld links neben der Sieben muss laut ihr einen Wert von mindestens Sechs enthalten. Von der Eins aus gesehen darf aber nur maximal eine Vier in dieses Feld. Da dies nicht miteinander vereinbar ist, muss es leer bleiben und kann somit sofort als leeres Feld markiert werden.

Da in meinem Modell die Null als leeres Feld gilt, muss sie als Ausnahme behandelt werden. Es muss für jedes Paar an Feldern pro Zeile und Spalte mit den Werten X,Y und dem Abstand $Dist$ folgende Gleichung gelten:

$$X > 0 \rightarrow \Big(Y < (X + Dist) \wedge \Big(Y = 0 \vee Y > (X - Dist)\Big)\Big) \tag{2.2}$$

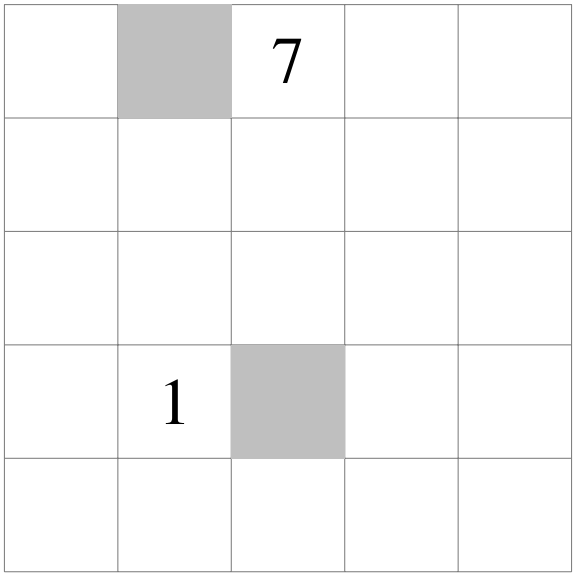

Abbildung 2.4: Beispiel: Der minimale Wert liegt über dem maximalen Wert.

Jedoch lieferte bei der Implementierung als logischen Ausdruck der Propagator nicht die gewünschten Resultate, daher wird dies mit Hilfe von der „notin“ Operation, die es einfach ermöglicht, bestimmte Werte aus dem Wertebereich einer Variable zu entfernen, umgesetzt. Die obere Schranke kann einfach als arithmetischer Ausdruck implementiert werden. Die „notin“ Operation wird genutzt, um die untere Schranke umzusetzen.

Dabei wird für jedes Paar an Feldern x, y in Abhängigkeit vom Abstand zwischen den Feldern für jede Zahl von $1 - 9$ ein Set an Ziffern erstellt, dass nicht in Feld y vorkommen darf, sofern Feld x diesen Wert erhält. Auf diese Weise implementiert, wurden die nicht zulässigen Werte beim Forward Checking entfernt.

Für ein Feld, dass den Abstand $Dist$ zu einem Feld mit dem Wert x hat, ergibt sich das Set mit den nicht erlaubten Werten wie folgt.

$$Set = \{y \mid y \in \{1..9\} \land y < x - Dist\}$$

Für $x = 6$ und $Dist = 2$ ergibt sich somit folgendes Set. Alle Werte in diesem Set können nicht in einem Feld, das zwei Felder neben einem Feld mit einer Vier liegt, enthalten sein.

$$Set = \{1, 2, 3\}$$

Die obere Schranke (als arithmetischer Ausdruck) entfernt zusätzlich die Neun aus dem Wertebereich.

Folgend ein Beispiel, das den übrige Wertebereich nach dem Platzieren eines Wertes und dem Aufrufen des Propagators des MinMax-Constraints zeigt.

0 3 4 5	4	0 3 4 5	0 2 3 4 5 6	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 8

Abbildung 2.5: Domain nach Aufruf des Propagators des MinMax-Constraints

2. Bei der zweiten Strategie wird die Parität von bereits platzierten Ziffern betrachtet. Bereits die erste platzierte Ziffer in einer Reihe oder Spalte gibt an, welche Paritäten alle anderen Ziffern der Reihe oder Spalte haben müssen. Da sich die Differenz zwischen zwei Ziffern mit jedem weiteren Feld um genau eins ändert, muss sich auch die Parität mit jedem Feld wieder ändern. Neben einer geraden Zahl kann nur eine ungerade Zahl platziert werden, ein weiteres Feld daneben muss wieder eine gerade Zahl platziert werden, sofern die Felder überhaupt gefüllt werden.

   Abbildung 2.6 zeigt, wie eine Ziffer für die ganze Reihe vorgibt, welche Parität die Ziffern haben müssen. Graue Felder müssen eine gerade Zahl enthalten, weiße hingegen eine ungerade Zahl. Dies gilt lediglich für Felder, die nicht frei bleiben.

	1				

Abbildung 2.6: Beispiel: Parität pro Feld in einer Zeile

Es kann passieren, dass zum Beispiel für ein Feld von einer Zahl in seiner Spalte gefordert wird, dass es eine gerade Zahl enthalten muss. Jedoch kann eine Zahl der Zeile dann fordern, dass das Feld eine ungerade Zahl enthalten muss. In diesem Fall kann das Feld keine Ziffer enthalten und muss frei bleiben. Dies wird in Abbildung 2.7 veranschaulicht. Beim schwarzen Feld kommt es zu dem geschilderten Konflikt, weswegen es frei bleiben muss.

Anzumerken ist hierbei, dass solche Konflikte in keiner der von Naoki Inaba erstellten Instanzen vorkommen.

Diese Strategie wurde als redundanter Constraint namens „oddEvenConstraint“ implementiert. Sofern eine Ziffer in einer Zeile oder Spalte platziert ist, werden für jedes übrige Feld der Zeile oder Spalte entweder alle geraden oder ungeraden Zahlen aus dem Wertebereich entfernt, abhängig von der Parität und dem Abstand zur platzierten Zahl.

Für jedes Paar an Feldern pro Zeile und Spalte mit den Werten X,Y und dem Abstand $Dist$ muss demnach folgende Gleichung gelten.

$$X > 0 \land Y > 0 \rightarrow \Big(|X \pmod 2 - Y \pmod 2| = Dist \pmod 2 \Big) \tag{2.3}$$

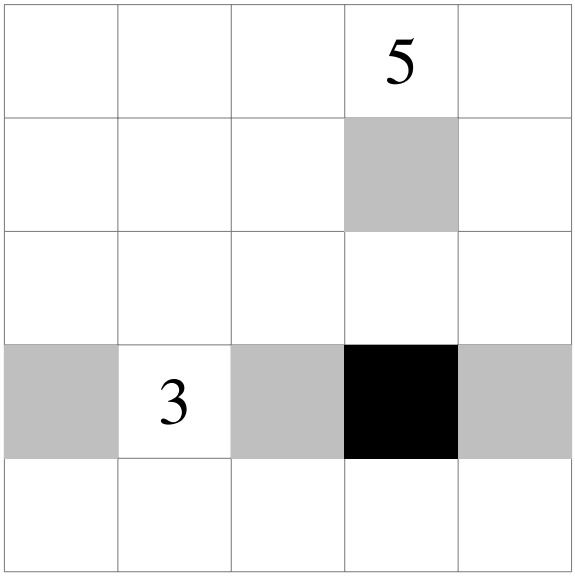

Abbildung 2.7: Beispiel: Parität führt zu einem Feld, das frei bleiben muss

Jedoch lieferte auch hier bei der Implementierung als logischen Ausdruck der Propagator nicht die gewünschten Resultate. Dies wird mit Hilfe von „ifThen“ und „notin“ umgesetzt. „ifThen“ koppelt einen Constraint an eine Bedingung, was dazu führt, dass er nur beachtet wird, wenn die Bedingung erfüllt ist. Es wird dabei zwischen vier Fällen unterschieden. „ifThen“ wird hierbei verwendet, um während des Lösens zwischen den Fällen zu unterscheiden.

a) Der erste Fall tritt ein, wenn der Abstand zwischen den Feldern gerade ist und der Wert des ersten Feldes auch gerade, aber größer als Null, ist.

   In diesem Fall werden alle ungeraden Zahlen aus dem Wertebereich des zweiten Feldes entfernt.

b) Der zweite Fall tritt ein, wenn der Abstand gerade ist, aber der Wert des ersten Feldes ungerade ist.

   Dieses Mal werden alle geraden Zahlen, die größer als Null sind, aus dem Wertebereich des zweiten Feldes entfernt.

c) Der dritte Fall tritt ein, wenn der Abstand ungerade ist und der Wert gerade und größer als Null ist.

   Hier werden auch alle geraden Zahlen größer als Null entfernt.

d) Der vierte Fall tritt ein, wenn der Abstand und der Wert des ersten Feldes ungerade sind.

   Hier werden alle ungeraden Zahlen entfernt.

Abbildung 2.8 zeigt den übrig gebliebenen Wertebereich der Variablen nach dem Aufruf des Propagators bei einem platzierten Wert.

0 1 3 5 7 9	2	0 1 3 5 7 9	0 2 4 6 8

Abbildung 2.8: Domain nach Propagation des OddEven-Constraints

3. Es existiert noch ein weiterer redundanter „NotEqual“-Constraint, der verhindert, dass Werte doppelt in einer Zeile oder Spalte vorkommen, um bei der Inferenz zu unterstützen.

   Folgend noch ein Beispiel in einer Zeile, bei dem alle drei redundanten Constraints angewandt wurden.

0 3 5	4	0 3 5	0 2 6	0 1 3 5 7	0 2 6 8

Abbildung 2.9: Domain nach Propagation der OddEven-, MinMax- und NotEqual-Constraints

2.2.3 Verschiedene Solver

Folgend werden die verschiedenen Solver für das Generieren der ersten Lösung und das Überprüfen der Schwierigkeit vorgestellt.

1. Der erste Solver wird nur für das Generieren der ersten Lösung verwendet. Er besitzt alle Constraints, um möglichst wenig Backtracking-Suche zu betreiben, da die gewählte Suchstrategie auf Zufall basiert und jedes Mal, wenn kein weiterer Wert durch Inferenz gefunden werden kann, ein zufälliges Feld zu einem zufälligen Wert seines Wertebereichs instanziiert.

2. Der leichte Solver besitzt nur die minimalen Constraints. Auf das Spezifizieren einer Suchstrategie wurde verzichtet, da der Solver keine Suche betreiben soll. Es wird lediglich der Propagator aufgerufen. So können Instanzen generiert werden, die schon mit wenig Strategien lösbar sind. Ein Rätsel, das mit diesem Solver lösbar ist, wird im Weiteren mit dem Schwierigkeitsgrad leicht oder Easy bezeichnet.

3. Der mittlere Solver besitzt alle minimalen Constraints und den MinMax-Constraint. Auch hier wird auf eine Suchstrategie verzichtet. Ein Rätsel, das mit diesem Solver lösbar ist, wird im Weiteren mit dem Schwierigkeitsgrad mittel oder Medium bezeichnet.

4. Der schwere Solver hingegen besitzt alle minimalen und redundanten Constraints. Es wird wieder auf eine Suchstrategie verzichtet. Ein Rätsel, das mit diesem Solver lösbar ist, wird im Weiteren mit dem Schwierigkeitsgrad schwer oder Hard bezeichnet.

2.3 Puzzle-Generierung

Folgende Ansprüche an die generierten Instanzen habe ich mir gesetzt:

- Die Instanzen sollen nur eine Lösung haben.
- Sie sollen ohne Backtracking, also nur durch logische Schlussfolgerung, lösbar sein.

Für das Generieren der Rätsel wurden die Ansätze aus der Quelle [6, s. 25] als Basis gewählt und mit Ansätzen aus der Quelle [5] kombiniert. Das Vorgehen ist hierbei zuerst mit Hilfe von Zufall eine gelöste Neighbor-Number-Instanz zu generieren. Daraufhin können entweder einzeln Werte von der Lösung entfernt werden, solange das Rätsel mit dem Solver der jeweiligen Schwierigkeitsstufe noch eindeutig lösbar ist oder dem leeren Brett einzeln Werte der Lösung hinzugefügt werden, bis es mit dem gewünschten Solver eindeutig lösbar ist. Verschiedene Wege, Werte hinzuzufügen oder zu entfernen, werden in Kapitel 2.3.2 vorgestellt.

Anzumerken ist hierbei noch einmal, dass bei den Solvern für die Schwierigkeitsstufen lediglich der Propagator der Constraints aufgerufen wird und keine Suche betrieben wird. Das hat außerdem den Vorteil, dass auf eine Überprüfung, ob nur eine Lösung existiert, verzichtet werden kann. Der Propagator platziert nur Werte, die auch eindeutig richtig sein müssen, da Forward Checking nur Werte aus dem Wertebereich entfernt, die nicht Teil einer Lösung sein können. Deshalb ist die gefundene Lösung zwangsläufig auch die einzig mögliche. Angenommen es würde eine weitere Lösung für die aktuell betrachtete Instanz existieren, dann könnte der Propagator diese Instanz nicht lösen, da an einer Stelle des Lösevorgangs eine Entscheidung getroffen werden müsste, um eine der beiden Lösungen zu wählen. Dies ist ohne Backtracking oder Raten nicht möglich.

2.3.1 Erste Lösung generieren

Gestartet wird mit einem leeren Brett einer beliebigen Größe. Daraufhin generiert der Solver mit der zufallsbasierten Suchstrategie eine Lösung. Aufgrund der zufallsbasierten Strategie, können sich die generierten Lösungen bei jedem Aufruf unterscheiden. Durch die Constraints wird der Suchbaum ausreichend verkleinert, sodass dies in akzeptabler Laufzeit möglich ist. Somit werden auch Instanzen generiert, die sich nicht nur in den Hinweisen, sondern auch in der Lösung unterscheiden.

2.3.2 Hinweise entfernen oder hinzufügen

Das Hinzufügen oder Entfernen der Hinweise kann als Graphensuchproblem dargestellt werden. Dabei ist jeder Knoten eine Neighbor-Number-Instanz und jede Kante ist dabei das Hinzufügen oder Entfernen eines Wertes. Die Wurzel des Suchbaums ist dabei entweder das leere Brett bei dem Bottom-Up Ansatz oder die fertige Lösung bei dem Top-Down Ansatz. Es können mehrere Pfade zum selben Knoten führen. Deshalb wird bei der systematischen Suche ein Hashset mit bereits besuchten Instanzen befüllt, um zu überprüfen, ob eine Instanz schon bekannt ist, um somit sicherzustellen, dass der gleiche Knoten nicht mehrmals besucht wird. Das Ziel der Suche ist dann das Finden einer Instanz, die mit dem gewünschten Solver lösbar ist, nur eine Lösung hat und möglichst wenig vorplatzierte Werte enthält. Abbildung 2.10 enthält einen Ausschnitt aus einem solchen Suchbaum.

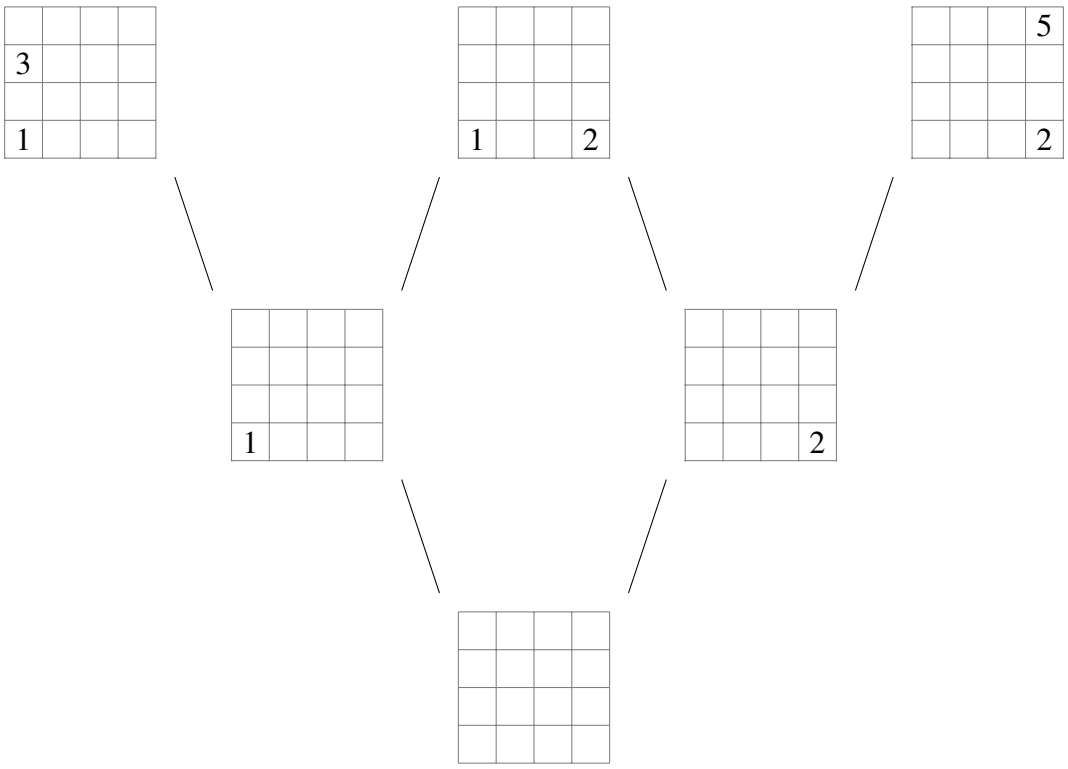

Abbildung 2.10: Einige Knoten aus einem Suchbaum mit dem leeren Brett als Wurzel

Es wurden fünf verschieden Wege, Hinweise hinzuzufügen oder zu entfernen, implementiert und getestet.

2.3.3 Top-Down systematische Suche

Der erste Ansatz startet mit dem gelösten Brett. Dann wird der gesammte Suchbaum an möglichen Kombinationen, Werte der Lösung zu entfernen, durchsucht. Dabei werden besuchte Knoten in einem Hashset gespeichert, um die selben Instanzen nicht mehrfach zu besuchen. Bei jedem besuchten Knoten wird überprüft, ob der Solver der gewünschten Schwierigkeit das Rätsel lösen kann. Hierbei wird nur Propagation angewandt, da so gewährleistet wird, dass das Rätsel ohne systematische Suche lösbar ist.

Während der Suche wird die lösbare Instanz mit den aktuell am wenigsten vorhandenen Werten zwischengespeichert. Die am Ende gespeicherte Instanz ist dann der Rückgabewert und somit auch die generierte Instanz des Rätsels.

Algorithm 1 Top-Down systematische Suche

```
1:  function EXPLORETREETOPDOWN(boardString, n, difficulty)   ▷ Where boardString - solution, n - size
2:      visited = HashSet < String >
3:      stack = Stack < String >
4:      for each number in boardString do
5:          stack.add(boardString without number)   ▷ Adding children of root to stack
6:      end for
7:      minNum = 3 * n
8:      while stack not empty do
9:          currentBoard = stack.pop()
10:         if currentBoard is solvable with solver of difficulty then
11:             if currentBoard has less numbers than ninNum then   ▷ new smallest instance
12:                 minNum = numbers count in currentBoard
13:                 boardString = currentBoard
14:             end if
15:             for each number in currentBoard do
16:                 newBoard = currentBoard without number
17:                 if !visited.contains(newBoard) then
18:                     stack.add(newBoard)   ▷ Adding children that were not visited before
19:                 end if
20:             end for
21:         end if
22:     end while
23:     return boardString
24: end function
```

Algorithmus 1 beschreibt den groben Ablauf der Funktion. Nicht gezeigt wurde ein Parameter, der die Suchtiefe beschränkt. Man könnte so zum Beispiel bei der ersten gefundenen Instanz mit nur sechs Werten die Suche abbrechen. Falls die Suchtiefe nicht erreicht werden kann, da die kleinste Instanz mehr Ziffern enthält, als angegeben ist oder eine gewünschte Größe von Null angegeben wurde, wird der gesammte Suchbaum durchsucht. So generiert er aus der gelieferten Lösung nicht nur eine Rätselinstanz, aus der keine weiteren Hinweise mehr entfernt werden können, sodass das Rätsel

mit den gewünschten Strategien ohne Raten noch lösbar wäre, sondern findet auch eine Instanz mit der kleinsten Anzahl an vorplatzierten Hinweisen.

2.3.4 Bottom-Up systematische Suche

Beim „ExploreTreeBottomUp“ (Algorithmus 2) handelt es sich wieder um einen systematischen Suchalgorithmus. Jedoch wird hier mit dem leeren Brett gestartet. Ein Beispiel eines solchen Suchbaums ist in Abbildung 2.10 zu sehen. Die Knoten werden in Reihenfolge der Breitensuche besucht, da so die erste Instanz, die gefunden wurde, auch zwangsläufig einer mit der minimalen Anzahl an Hinweisen sein muss. Dadurch muss nicht der ganze Suchbaum durchsucht werden, um eine minimale Instanz zu finden. Dafür nutzt dieser Algorithmus, anders als Algorithmus 1 eine Queue anstatt eines Stacks für das Speichern der zu besuchenden Knoten. Auch hier wird ein Hashset verwendet, um Knoten nicht doppelt zu besuchen.

Algorithm 2 Bottom-Up systematische Suche

```
1:  function EXPLORETREEBOTTOMUP(boardString, n, difficulty)   ▷ Where boardString -
    solution, n - size
2:      visited = HashSet < String >
3:      queue = Queue < String >
4:      queue.add(empty board)
5:      visited.add(empty board)
6:      while queue not empty do
7:          currentBoard = queue.pollFirst()
8:          if currentBoard is solvable with solver of difficulty then
9:              return currentBoard
10:         end if
11:         for each number missing in currentBoard do
12:             newBoard = currentBoard with number
13:             if !visited.contains(newBoard) then
14:                 queue.Add(newBoard)          ▷ Adding children that were not visited before
15:             end if
16:         end for
17:     end while
18:     return null
19: end function
```

2.3.5 Top-Down probabilistische Suche mit einem Ast

Es wurden auch probabilistische Ansätze getestet. Bei „ProbabilisticTopDownSingleBranch“ (Algorithmus 3) wird mehrere Male zufällig versucht, den Suchbaum von der Lösung aus zu durchsuchen. SingleBranch bezieht sich hierbei darauf, dass nach dem Überprüfen eines Knotens nur ein Kind als nächstes überprüft wird. Bei jeder Iteration wird mit der Lösung angefangen. Dann wird ein zufälliger Wert entfernt und überprüft, ob das Rätsel noch mit dem gewünschten Solver lösbar ist. Dies wird wiederholt bis das aktuell überprüfte Rätsel nicht mehr lösbar ist. Dann wird das vorherige noch

lösbare Rätsel mit der bisher besten Instanz verglichen. Falls die neu gefundene Instanz kleiner ist, also weniger Hinweise enthält, als die bisher kleinste gefundene Instanz, wird sie nun als bisher beste Instanz zwischengespeichert und die nächste Iteration beginnt wieder bei der Lösung. Die kleinste Instanz wird am Ende zurückgegeben.

Algorithm 3 Top-Down probabilistische Suche mit nur einem Ast

```
1:  function ProbabilisticTopDownSingleBranch(boardString, n, numIterations, difficulty)
    ▷ Where boardString - solution, n - size, numIterations - number of iterations
2:      minNum= 3 * n                                              ▷ max amount of numbers
3:      bestBoardString = boardString
4:      for 0 to numIterations do
5:          isSolvable = true
6:          currentBoard = boardString
7:          lastBoard = currentBoard
8:          while isSolvable do
9:              lastBoard = currentBoard
10:             currentBoard = RemoveRandomNumber(currentBoard)    ▷ Remove a random value
11:             isSolvable = currentBoard is solvable with Solver of given difficulty
12:         end while                                  ▷ lastBoard is last solvable instance
13:         if lastBoard has less numbers than ninNum then                ▷ new smallest instance
14:             minNum = numbers count in lastBoard
15:             bestBoardString = lastBoard
16:         end if
17:     end for
18:     return bestBoardString
19: end function
```

Dieser Algorithmus liefert nicht zwangsläufig ein Rätsel ohne redundante Hinweise. Eine höhere Anzahl an Such-Iterationen liefert tendenziell Rätsel mit weniger Hinweisen, erhöht aber die Laufzeit.

2.3.6 Top-Down probabilistische Suche mit mehreren Ästen

„ProbabilisticTopDownMulti“ (Algorithmus 4) ist eine mögliche Optimierung von Algorithmus 3 und durchsucht den Suchbaum auch von der Lösung aus probabilistisch. Jedoch wird hier nicht bei der ersten nicht mehr lösbaren Instanz abgebrochen und die nächste Iteration gestartet. Stattdessen werden alle Kinder eines Knotens in zufälliger Reihenfolge überprüft und das erste lösbare gefundene Kind wird als nächster Knoten besucht. Also wird die nächste Iteration erst gestartet, wenn keines der Kinder des aktuellen Knotens lösbar ist. Es wird erhofft, dass so eine größere Suchtiefe erreicht werden kann und das Rätsel am Ende somit weniger bereits platzierte Zahlen enthält. Die Kosten dafür sind eine höhere Laufzeit, da so mehr Instanzen überprüft werden. Dieser Ansatz liefert Instanzen ohne redundante Hinweise, da eine Iteration immer mit einer Instanz endet, aus der keine weiteren Hinweise mehr entfernt werden können, sodass der gewünschte Solver das Rätsel noch lösen kann. Sie muss aber nicht die kleinste Menge an Hinweisen enthalten. Dies kann nur mit systematischer Suche gewährleistet werden.

Algorithm 4 Top-Down probabilistische Suche mit mehreren Ästen

```
1: function ProbabilisticTopDownMulti(boardString, n, numIterations, difficulty) ▷
   Where boardString - solution, n - size, numIterations - number of iterations
2:     minNum= 3 ∗ n                                                  ▷ max amount of numbers
3:     bestBoardString = boardString
4:     for 0 to numIterations do
5:         viableChildFound = true
6:         currentBoard = boardString
7:         lastBoard = currentBoard
8:         while viableChildFound do
9:             lastBoard = currentBoard
10:            viableChildFound = false
11:            for number in lastBoard in random order do currentBoard = lastBoard without number
12:                if currentBoard is solvable with Solver of given difficulty then
13:                    viableChildFound = true
14:                    break for
15:                end if
16:            end for
17:            if lastBoard has less numbers than ninNum then ▷ lastBoard is last solvable instance
18:                minNum = numbers count in lastBoard
19:                bestBoardString = lastBoard                        ▷ new smallest instance
20:            end if
21:        end while
22:    end for
23:    return bestBoardString
24: end function
```

2.3.7 Bottom-Up probabilistische Suche

„ProbabilisticBottomUp“ (Algorithmus 5) ähnelt Algorithmus 3 sehr. Es handelt sich hierbei auch um einen probabilistischen Algorithmus, der jedoch den Suchbaum nicht von der Lösung aus durchsucht, sondern mit dem leeren Brett beginnt. Es wird immer ein zufälliger Wert aus der Lösung zur aktuellen Instanz hinzugefügt und überprüft, ob es schon mit dem gewünschten Solver nur mit Propagation lösbar ist. Ab der ersten gefundenen lösbaren Instanz in jeder Iteration wird die nächste Iteration gestartet. Die Instanz mit den wenigsten Hinweisen ist auch hier wieder der Rückgabewert.
Jeder dieser Ansätze gibt am Ende eine Neighbor-Number-Instanz zurück, die nur durch Inferenz oder logische Schlussfolgerung lösbar ist. Bei allen probabilistischen Ansätzen kann die Anzahl an Iterationen angepasst werden. Eine höhrere Anzahl kann bessere Ergebnisse liefern, hat aber auch eine höhere Laufzeit.

Algorithm 5 Bottom-Up probabilistische Suche

```
1: function ProbabilisticTopDownSingleBranch(boardString, n, numIterations, difficulty)
   ▷ Where boardString - solution, n - size, numIterations - number of iterations
2:     minNum= 3 ∗ n                                              ▷ max amount of numbers
3:     bestBoardString = boardString
4:     for 0 to numIterations do
5:         isSolvable = false
6:         currentBoard = empty Board
7:         while isSolvable do
8:             currentBoard = currentBoard with another random number from solution not yet in
   currentBoard
9:             isSolvable = currentBoard is solvable with Solver of given difficulty
10:        end while                                  ▷ currentBoard is first solvable instance
11:        if currentBoard has less numbers than ninNum then                ▷ new smallest instance
12:            minNum = numbers count in currentBoard
13:            bestBoardString = currentBoard
14:        end if
15:    end for
16:    return bestBoardString
17: end function
```

3 Ergebnisse

Alle Experimente wurden mit einem AMD Ryzen 5 1600X mit 3.600 Mhz und 16 GB-DDR4-Ram mit 1.600 MHz durchgeführt.

Für jeden Algorithmus, der in 2.3 vorgestellt wurde, und für jede Schwierigkeitsstufe wurden 25 Instanzen der Größe 5×5 generiert. Bei probabilistischer Suche wurden 50 Iterationen durchgeführt. Bei „ExploreTreeTopDown“ wurde die Suchtiefe nicht begrenzt. Tabelle 3.1 zeigt die durschnittliche Anzahl an vorplatzierten Werten der Instanzen, so wie durchschnittliche Zeit (in Sekunden), die das Generieren einer Instanz gedauert hat.

Algorithmus	Schwierigkeit	Durchschnittliche Anzahl an Werten	Durchschnittliche Laufzeit in Sekunden
ExploreTreeTopDown	Easy	8,9	3,6
ExploreTreeBottom Up		8,6	33,3
ProbabilisticTopDownSingleBranch		10,3	0,1
ProbabilisticTopDownMulti		8,8	2,2
ProbabilisticBottomUp		10	0,9
ExploreTreeTopDown	Medium	6,1	133,9
ExploreTreeBottom Up		6,0	91,3
ProbabilisticTopDownSingleBranch		7,9	1,3
ProbabilisticTopDownMulti		6,0	17,0
ProbabilisticBottomUp		7,9	5,2
ExploreTreeTopDown	Hard	6,0	178,9
ExploreTreeBottom Up		5,9	106,5
ProbabilisticTopDownSingleBranch		7,2	1,9
ProbabilisticTopDownMulti		6,0	16,9
ProbabilisticBottomUp		7,6	8,1

Tabelle 3.1: 25 5x5 Puzzles je Algorithmus und Schwierigkeit wurden generiert

Für Tabelle 3.2 wurden zehn Instanzen pro Schwierigkeitsgrad und Algorithmus generiert. Diesmal in einer Größe von 6×6. Es werden wieder die durschnittliche Anzahl an vorplatzierten Werten der Instanzen und die durchschnittliche Zeit (in Sekunden) pro Instanz gezeigt.

Algorithmus	Schwierigkeit	Durchschnittliche Anzahl an Werten	Durchschnittliche Laufzeit in Sekunden
ExploreTreeTopDown	Easy	11,0	11,1
ExploreTreeBottom Up		11,2	582,7
ProbabilisticTopDownSingleBranch		13,7	0,2
ProbabilisticTopDownMulti		11,1	3,8
ProbabilisticBottomUp		13,7	2,0
ExploreTreeTopDown	Medium	7,7	1501,5
ExploreTreeBottom Up		6,5	739,0
ProbabilisticTopDownSingleBranch		10,2	2,3
ProbabilisticTopDownMulti		7,1	39,9
ProbabilisticBottomUp		9,5	12,9
ExploreTreeTopDown	Hard	6,4	4071,1
ExploreTreeBottom Up		6,4	1801,7
ProbabilisticTopDownSingleBranch		7,6	1073
ProbabilisticTopDownMulti		6,3	45,4
ProbabilisticBottomUp		9,6	19,7

Tabelle 3.2: zehn 6x6 Puzzles je Algorithmus und Schwierigkeit wurden generiert

Des Weiteren wurden noch jeweils zwei Instanzen der Größe 7×7 je Schwierigkeitsgrad und Algorithmus generiert. Jedoch wurden die Experimente der systematischen Suche bei dieser Größe der oberen Schwierigkeitsstufen auf Grund der hohen Laufzeit von über fünf Stunden abgebrochen. Gezeigt werden in Tabelle 3.3 erneut die durchschnittliche Anzahl an Werten und die durschnittliche Laufzeit in Sekunden.

Algorithmus	Schwierigkeit	Durchschnittliche Anzahl an Werten	Durchschnittliche Laufzeit in Sekunden
ExploreTreeTopDown	Easy	14,0	26,6
ExploreTreeBottom Up		15,0	9910,0
ProbabilisticTopDownSingleBranch		16,8	221,8
ProbabilisticTopDownMulti		14,0	9,3
ProbabilisticBottomUp		17,1	5,8
ExploreTreeTopDown	Medium		>5h
ExploreTreeBottom Up			>5h
ProbabilisticTopDownSingleBranch		13,4	3,8
ProbabilisticTopDownMulti		11,7	68,1
ProbabilisticBottomUp		12,2	35,2
ExploreTreeTopDown	Hard		>5h
ExploreTreeBottom Up			>5h
ProbabilisticTopDownSingleBranch		10,5	8,0
ProbabilisticTopDownMulti		8,	103,6
ProbabilisticBottomUp		9,9	45,6

Tabelle 3.3: zwei 7x7 Puzzles je Algorithmus und Schwierigkeit wurden generiert

Aus den selben Datensätzen, aus denen auch die Daten für die Tabellen 3.1, 3.2 und 3.3 ausgelesen wurden, wurden auch die Anzahl an Knoten, die pro Sekunde überprüft wurden, ausgelesen. In Tabelle 3.4 werden diese nach Größe getrennt je Schwierigkeit und Algorithmus dargestellt.

Knoten / Sekunde	Easy	Medium	Hard
5x5			
ExploreTreeTopDown	899,0	102,3	76,0
ExploreTreeBottomUp	644,9	86,0	62,1
ProbabilisticTopDownSingle	1496,7	170,9	124,0
ProbabilisticTopDownMulti	718,3	89,8	91,3
ProbabilisticBottomUp	710,8	111,3	71,2
6x6			
ExploreTreeTopDown	582,1	49,4	36,1
ExploreTreeBottomUp	364,3	46,9	33,4
ProbabilisticTopDownSingle	624,5	102,0	33,8
ProbabilisticTopDownMulti	520,9	46,6	42,8
ProbabilisticBottomUp	420,8	52,6	34,2
7x7			
ExploreTreeTopDown	394,0		
ExploreTreeBottomUp	203,6		
ProbabilisticTopDownSingle	588,4	60,5	38,6
ProbabilisticTopDownMulti	241,1	41,3	21,2
ProbabilisticBottomUp	168,8	23,8	17,4

Tabelle 3.4: Überprüfte Knoten pro Sekunde je Algorithmus und Schwierigkeit

Wieder aus den selben Datensätzen wurden die Daten in Tabelle 3.5 ausgelesen. Zusätzlich zur durchschnittlichen Zeit pro Instanz, der durchschnittlichen Anzahl an besuchten Knoten und der durchschnittlichen Anzahl an besuchten Knoten pro Sekunde ist hier noch die durchschnittliche Anzahl an vermiedenen Duplikaten aufgelistet. Diese wurden durch die Hashsets vermieden.

	TopDown	BottomUp	Schwierigkeit
5x5			
Zeit in Sekunden	3604	33280	Easy
Anzahl an Knoten	3240	21464	
Anzahl an Duplikaten vermieden	1953	156976	
Knoten pro Sekunde	899	645	
Zeit in Sekunden	133852	91257	Medium
Anzahl an Knoten	13688	7849	
Anzahl an Duplikaten vermieden	23629	62515	
Knoten pro Sekunde	102	86	
Zeit in Sekunden	178905	106532	Hard
Anzahl an Knoten	13593	6620	
Anzahl an Duplikaten vermieden	22959	53192	
Knoten pro Sekunde	76	62	
6x6			
Zeit in Sekunden	11138	582683	Easy
Anzahl an Knoten	6483	212243	
Anzahl an Duplikaten vermieden	2483	1816582	
Knoten pro Sekunde	582	364	
Zeit in Sekunden	1501508	739027	Medium
Anzahl an Knoten	74205	34681	
Anzahl an Duplikaten vermieden	169030	351236	
Knoten pro Sekunde	49	47	
Zeit in Sekunden	4071147	1801723	Hard
Anzahl an Knoten	146861	60220	
Anzahl an Duplikaten vermieden	633490	555626	
Knoten pro Sekunde	36	33	

Tabelle 3.5: Vergleich der systematischen Algorithmen

Bei einem weiteren Experiment wurde der Einfluss der Iterationszahl auf die Resultate der probabilistischen Algorithmen untersucht. Dafür wurden je 100 Instanzen für jeden probabilistischen Algorithmus mit den Iterationszahlen 1, 10 und 50 generiert. Es wurden die Schwierigkeitsstufen „Easy “ und „Hard “ gewählt. Beim Top Down Ansatz mit mehreren Ästen wurde das Experiment bei 50 Iterationen auf Grund von hoher Laufzeit abgebrochen.

Tabelle 3.6 zeigt jeweils die durschnittliche Anzahl an Werten in der fertigen Instanz, die durschnittlich erreichte Schicht, die durchschnittliche Zeit pro Instanz in Sekunden und die durschschnittliche Anzahl der besuchten Knoten. Die durschnittlich erreichte Schicht gibt an, wie viele Zahlen die kleinste lösbare Instanz in jeder Iteration im Durchschnitt hatte.

		TopDownMulti	TopDownSingle	BottomUp
1 Iteration				
Easy	Anzahl an Werten	11,9	16,3	16,3
	Durchschnittlich erreichte Schicht	11,9	16,3	16,3
	Durchschnittliche Zeit in Sekunden	0,063	0,004	0,052
	Anzahl an besuchten Knoten	39	2,66	16
Hard	Anzahl an Werten	7,04	12,47	13,17
	Durchschnittlich erreichte Schicht	7,04	12,47	13,17
	Durchschnittliche Zeit in Sekunden	0,858	0,126	0,422
	Anzahl an besuchten Knoten	38	7	13
10 Iterationen				
Easy	Anzahl an Werten	11,1	14,5	14,6
	Durchschnittlich erreichte Schicht	11,7	16,3	16,3
	Durchschnittliche Zeit in Sekunden	0,650	0,028	0,472
	Anzahl an besuchten Knoten	389	27	163
Hard	Anzahl an Werten	6,4	9,8	9,4
	Durchschnittlich erreichte Schicht	7,3	13,2	12,8
	Durchschnittliche Zeit in Sekunden	8,807	1,094	4,197
	Anzahl an besuchten Knoten	379	58	128
50 Iterationen				
Easy	Anzahl an Werten	11,1	13,5	13,6
	Durchschnittlich erreichte Schicht	11,8	16,4	16,3
	Durchschnittliche Zeit in Sekunden	4,104	0,141	2,366
	Anzahl an besuchten Knoten	1932	131	816
Hard	Anzahl an Werten		8,4	8,2
	Durchschnittlich erreichte Schicht		12,9	12,8
	Durchschnittliche Zeit in Sekunden		5,967	20,390
	Anzahl an besuchten Knoten		306	639

Tabelle 3.6: Auswirkung der Iterationszahl auf Probabilistische Suche

4 Diskussion

4.1 Interpretation der Ergebnisse

Die Tabellen 3.1, 3.2 und 3.3 geben eine grobe Übersicht über die Anzahl an Werten der generierten Instanzen und die Zeit, die im Durchschnitt nötig war, um sie mit den jeweiligen Algortihmen zu generieren. Dabei ist eine möglichst kleine Anzahl an Werten gewünscht. Man kann erkennen, dass die systematischen Algorithmen im Schnitt eine deutlich höhere Laufzeit als die probabilistischen Algorithmen aufweisen. Jedoch ist die Anzahl der Werte bei den systematischen Algorithmen überwiegend kleiner.

Jedoch weist der probabilistische Top-Down Algorithmus mit mehreren Ästen eine Anzahl an Werten auf, die denen der systematischen Algorithmen sehr nah kommt. In allen Experimenten ist die Differenz der durchschnittlichen Anzahl an Werten von diesem Algorithmus zu den Werten der systematischen Algorithmen kleiner als eins. Die Laufzeit fällt dabei zwischen die der systematischen und die der anderen probabilistischen Algorithmen. Bei leichten Rätseln beträgt die Laufzeit immer weniger als zwei Drittel der Laufzeit des ExploreTreeTopDown Algorithmus.
Bei mittleren Instanzen der Größe 6×6 beträgt die Laufzeit jeweils weniger 5% der Laufzeit des schnelleren systematischen Algorithmus. Bei schweren Instanzen beträgt die Laufzeit sogar nur $2,5\%$ des Wertes des schnelleren systematischen Algorithmus.

Instanzen der Größe 7×7 zu generieren ist mit den systematischen Algorithmen nur sehr langsam möglich. Bei den höheren Schwierigkeitstufen betrug die Dauer mehr als fünf Stunden und wurde somit abgebrochen. Eine Ausnahme ist hierbei der Top-Down Ansatz bei der Schwierigkeitsstufe leicht, wo er im Schnitt $26,6$ Sekunden pro Instanz gebraucht hat. Dies könnte darauf rückführbar sein, dass der Suchbaum von der Lösung aus bei dem leichten Solver deutlich kleiner ausfällt, als bei den anderen Schwierigkeitsstufen.

Auch bei den anderen Größen ist zu sehen, dass der systematische Top-Down Ansatz bei leichter Schwierigkeit schneller ist, als der systematische Bottom-Up Ansatz. Dies wird in Kapitel 4.1.2 genauer betrachtet.

4.1.1 Knoten pro Sekunde

In Tabelle 3.4 ist zu sehen, welchen Einfluss die Größe des Rätsels und die Schwierigkeit auf das Lösen des Rätsel haben. Da bei jedem Knoten einmal versucht wird, das Rätsel mit dem jeweiligen Solver zu lösen, bietet die Anzahl der überprüften Knoten pro Sekunde Aufschluss auf den Aufwand beim Lösen.

Deutlich ist zu erkennen, dass sich die Anzahl der Knoten pro Sekunde bei leichter Schwierigkeit deutlich von den anderen unterscheidet.
Dies könnte die Ursache haben, dass der leichte Solver mit deutlich weniger Constraints weniger Zeit bei der Propagation benötigt.
Eine andere Ursache könnte außerdem sein, dass Rätsel, bei der weniger Werte fehlen, schneller lösbar sein könnten. Dies ist vorallem beim probabilistischen Top-Down Ansatz mit nur einem Ast und dem systematischen Top-Down Ansatz zu sehen. Da diese den Suchbaum von der Lösung aus durchsuchen, werden tendenziell mehr Instanzen mit wenig fehlenden Werten überprüft als beim Bottom-Up Ansatz, der den Suchbaum vom leeren Brett aus durchsucht.

4.1.2 Vergleich der systematischen Algorithmen

In Tabelle 3.5 werden die systematischen Ansätze genauer verglichen.

Zu sehen ist, dass der Top-Down Algorithmus bei der Schwierigkeitsstufe leicht deutlich weniger Knoten pro generierter Instanz überprüfen muss als der Bottom-Up Algorithmus.
Dies ist hingegen bei den anderen Schwierigkeitsstufen umgekehrt. In den Tabellen 3.1 und 3.2 ist zu sehen, dass die kleinste Instanz bei höherer Schwierigkeitsstufe deutlich weniger Werte enthält, als bei leichter Schwierigkeitsstufe. Daher muss der Bottom-Up Algorithmus deutlich weniger Knoten überprüfen bis sie gefunden wird.

Daraus lässt sich schließen, dass der systematische Top-Down Algorithmus zum Generieren von einfachen Instanzen besser geeignet ist, während der Bottom-Up Algorithmus besser zum Generieren von mittleren und schweren Instanzen geeignet ist.

In beiden Fällen führte das HashSet dazu, dass erhebliche Mengen an doppelten Knoten vermieden wurden.

4.1.3 Iterationszahl bei probabilistischen Algorithmen

In Tabelle 3.6 wird der Einfluss der Iterationszahl bei probabilistischen Algorithmen dargestellt.
Überraschend ist hierbei, dass die Iterationszahl beim ProbabilisticTopDownMulti Algorithmus kaum Auswirkung auf die Anzahl der Werte der generierten Instanzen hat. Eine Iteration lieferte im Durchschnitt Instanzen miit $11,9$ Werten während 50 Iterationen Instanzen mit durchschnittlich $11,1$ Werten lieferten. Dabei ist die durchschnittlich erreichte Schicht auch sehr nah an der Anzahl an Werten. Die Differenz von der durchschnittlichen Schicht zur besten Schicht pro Instanz ist in jedem Fall kleiner als eins.

Beides lässt darauf schließen, dass der ProbabilisticTopDownMulti Algorithmus schon mit sehr wenigen Iterationen gute Ergebnisse erzielen kann. Durch wenige Iterationen fällt die Laufzeit auch deutlich geringer aus.

Da die durchschnittlich erreichte Schicht, welche angibt, wie viele Ziffern die kleinste lösbare Instanz je Iteration, der Durschnitt von allen Iterationen ist, ändert er sich nicht signifikant bei einer Erhöhung der Iterationszahl.

Jedoch verringert sich beim ProbabilisticTopDownSingle Algorithmus und dem ProbabilisticBottomUp Algorithmus die Anzahl an platzierten Werten stark mit steigender Iterationszahl. Bei einer Iterationszahl von eins lag der Durchschnitt bei leichten Instanzen bei beiden Algorithmen bei $16,3$. Bei 50 Iterationen verbesserten sich diese Werte auf $13,5$ und $13,6$.
Dies lässt sich darauf zurückführen, dass diese Algorithmen möglicherweise nur mit geringer Chance einen Pfad wählen, der im Suchbaum auch tief reicht.

4.2 Resultat

Der ProbabilisticTopDownMulti Algorithmus ist eine Optimierung des ProbabilisticTopDownSingle Algorithmus. Er generiert bessere Instanzen in nicht allzu größerer Laufzeit. Vor allem bei niedriger Iterationszahl. Er ist im Vergleich zu den beiden anderen probabilistischen Algorithmen fast immer zu bevorzugen.

Die systematischen Algorithmen erwiesen sich bei größeren Brettern als weniger nützlich. Jedoch sind sie in der Lage, bei kleineren Brettern in akzeptabler Zeit die kleinsten Instanzen zu finden. Abhängig von der gewünschten Schwierigkeit sind beide Algorithmen jeweils stärker als der andere.

Für das Generieren von Instanzen „on demand“ ist wahrscheinlich nur der ProbabilisticTopDownMutli Algorithmus nützlich, da dieser eine gute Laufzeit bei dennoch recht hoher Qualität der generierten Instanzen aufweist.

4.3 Vergleich mit bestehender Literatur und Limitation

Da es sich um unterschiedliche Rätsel handelt, ist ein Vergleich von dieser Arbeit mit den Werken [6, 5] nur begrenzt möglich. Dennoch konnten die Ansätze genutzt werden, um erfolgreich Neighbor-Number-Instanzen in diversen Schwierigkeitsstufen zu generieren.
Die Ansätze der Generation eines Rätsels aus einer Lösung könnten auch auf andere japanische Paper Puzzles angewandt werden, wenn bei diesen Felder mit verschiedenen Objekten gefüllt werden müssen, sodass bestimmte Regeln eingehalten werden.

Es wurden nur Instanzen der Größe 5×5 bis 7×7 generiert. Dies entspricht den Größen der vier Instanzen, die Naoki Inaba selbst erstellt hat.[1]

Die Schwierigkeit der generierten Rätsel könnte durch weitere Strategien verbessert werden. Gegebenenfalls könnte bei den probabilistischen Algorithmen auch Multithreading eingesetzt werden, um mehrere Iterationen gleichzeitig durchzuführen und somit die Laufzeit noch weiter zu verbessern.

5 Fazit

In dieser Arbeit wurde versucht, mit Hilfe von Constraint Programming japanische Paper Puzzle Instanzen zu generieren.
Dies wurde am Beispiel Neighbor-Number von Naoki Inaba durchgeführt.

Die systematischen Algorithmen waren auf Grund der hohen Laufzeit eher für kleinere Rätsel nützlich. Der optimierte probabilistische Alorithmus hingegen lieferte gute Resultate bei dennoch geringer Laufzeit. Dies gelang auch bei größeren Rätseln.

Künftig könnten andere mögliche Ansätze beim Entfernen oder Hinzufügen von Hinweisen untersucht werden. Falls eine geeignete Heuristik gefunden wird, könnten andere systematische Suchalgorithmen mit Heuristik wie der A*-Algorithmus oder auch der Dijkstra-Algorithmus eine gute Alternative darstellen.

Literaturverzeichnis

[1] Neighbor number. http://inabapuzzle.com/honkaku/neigh.pdf. zuletzt besucht: 09.07.2024.

[2] Choco solver. https://choco-solver.org/. zuletzt besucht: 26.06.2024.

[3] Krzysztof R Apt. Principles of constraint programming, 1999.

[4] Toby Walsh Francesca Rossi, Peter van Beek. Constraint programming. *Foundations of Artificial Intelligence*, 3:181–186, 2008.

[5] Martin Hunt, Christopher Pong, and George Tucker. Difficulty-driven sudoku puzzle generation. *The UMAP Journal*, 29(3):349–362, 2007.

[6] Helmut Simonis. Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, volume 12, pages 13–27, 2005.