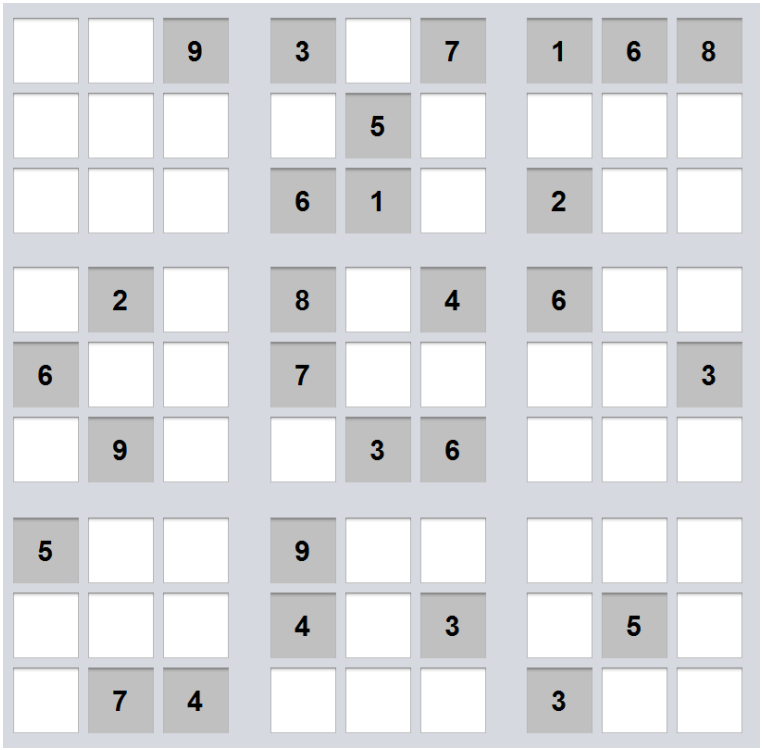
Entwicklung einer JAVA-Anwendung für Sudokus, welche über einen Auto-Solve-Algorithmus verfügt

Informatik

Fachlehrer: Herr Faßbender

Kurs: LK IF7

Schuljahr: 11



Erik Springer

Inhaltsverzeichnis

1 Einleitung

Nachdem ich bei meinem letzten Versuch ein schweres Sudoku-Zahlenrätsel zu lösen, gescheitert bin, habe ich mich entschieden ein Programm zu entwickeln, welches jedes machbare Sudoku lösen können soll. Denn das Problem mit den heutigen Sudokus ist, dass sie computergeneriert sind. Bei schweren Sudokus kommt es dann vor, dass man ein oder mehrere freie Felder raten muss um weiter zu kommen. Das macht es fast unmöglich es zu knacken, denn man weiß erst das man sich vertan hat, wenn das Sudoku am Ende nicht komplett aufgeht. Nun hat man das Problem! Alles wieder wegmachen? Aber woher soll man wissen, welche Felder man seit dem letzten Raten eingetragen hat? Hätte man etwa alle markieren müssen? Es scheint unmöglich für einen Menschen. Klar ist also: ein Programm muss her!

Zielsetzung dieser Facharbeit ist folgendes:

- Implementation von Sudoku-Lösungsalgorithmen
- Effizienzvergleich der Lösungsalgorithmen
- Algorithmus zum Erstellen von Sudoku-Zahlenrätsel
- Implementation einer graphischen Benutzeroberfläche

Die Arbeit ist wie folgt strukturiert: In Abschnitt 2 geht es um die Grundlagen des Sudokus und die für das Projekt verwendete Programmiersprache. Außerdem werden kurz verschiedene Lösungsalgorithmen für Sudoku-Zahlenrätsel erläutert. Abschnitt 3, beschreibt wie das Projekt aufgebaut ist, wie die oben genannten Algorithmen, die Hilfsmethoden und die GUI(Englisch: Graphical user interface, GUI) implementiert wurden. Abschnitt 4 vergleicht die Laufzeiten der Lösungsmethoden. Zum Schluss fasst Abschnitt 5 die Ergebnisse der Facharbeit kurz zusammen.

2 Grundlagen

2.1 Sudoku

Sudoku ein ist populäres Zahlenrätsel, welches einen weltweiten Bekanntheitsgrad hat und viele Menschen unterhält. Es gibt sie in der Zeitung, im Internet und auch als Handy-Spiel. Für das Lösen dieses Rätsels sind keine Rechenkünste von Nöten. Durch logisches Denken kann jeder ein Sudoku Rätsel lösen.
Das Wort Sudoku kommt aus dem japanischen und heißt ”Isolieren der Zahlen”. Es handelt sich um ein Logikrätsel, bei dem es darum geht, durch geschicktes Einsetzen von Zahlen ein 9x9 Gitter zu Füllen. Erlaubt sind dabei nur die Zahlen von 1 bis 9. Die Schwierigkeit liegt darin, dass jede Zahl nur einmal pro Zeile, Spalte und 3x3 Block vorkommen darf. Es gibt unzählige Sudokus, bei denen der Schwierigkeitsgrad von sehr einfach bis extrem schwierig reicht. Einfache Sudokus zeichnen sich durch viele vorgegebene Zahlen aus und schwere Sudokus durch wenige. Jedes Sudoku mit mehr als 16 vorgegebenen Feldern hat genau eine Lösung[2].
Auch gibt es unterschiedliche Varianten. So ist beim Mega-Sudoku zum Beispiel ein 16x16 Gitter zu füllen und beim Diagonal-Sudoku sind auch in die Diagonalen die Zahlen von 1 bis 9 einzuordnen[1].

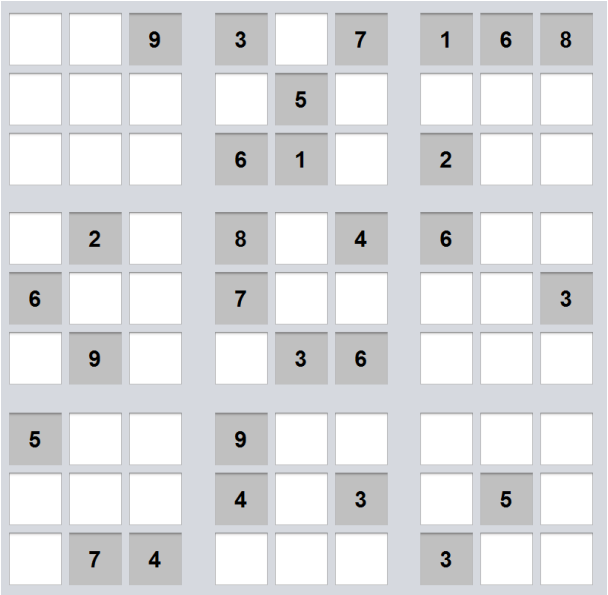


Abbildung 1: Ein Ungelöstes Sudoku

2.2 Programmiersprache und Entwicklerumgebung

Für dieses Projekt wurde die Programmiersprache Java verwendet. Die objektorientierte Programmiersprache erschien 1995 und wurde von Sun Microsystems entwickelt[3]. Das Unternehmen wurde 2010 von Oracle aufgekauft. Java gehört zur Java-Technologie, welche eine Sammlung verschiedener Laufzeitumgebungen ist.

Das Besondere an Java ist, dass im Gegensatz zu anderen Programmiersprachen, der Maschinencode nicht für ein bestimmtes Betriebssystem ist. Der Quelltext wird in einen Bytecode kompiliert, welcher nur von der Java virtual Machine(JVM) ausgeführt werden kann.
Für die Erstellung des Programms habe ich die speziell für Java integrierte Entwicklungsumgebung BlueJ benutzt. BlueJ visualisiert Klassen als verkürztes Klassendiagramm. Darüber kann man auch Quelltext bearbeiten und kompilieren[4].
Außerdem wurde zur Erstellung der graphischen Benutzeroberfläche NetBeans IDE(integrated development environment) verwendet, welches auch hauptsächlich für Java gedacht ist[5]. NetBeans ist sehr beliebt für die Modellierung von Benutzeroberflächen, da dafür gemachte Frameworks integriert sind.

2.3 Rekursion und Backtracking

Eine Methode kann sowohl iterativ als auch rekursiv implementiert werden. Eine Iteration entsteht durch eine Schleife, in der eine Anweisung mehrfach ausgeführt wird. Die Rekursion(von lateinisch recurrere = zurücklaufen)[6] beschreibt eine Methode, welche sich selber immer weiter aufruft, bis ein Abbruchkriterium eintritt. Rekursive Methoden sind zwar speicherintensiv, dafür aber übersichtlicher und benötigen weniger Quelltext. Das Backtracking, auch Rücksetzverfahren genannt, ist eine Problemlösungsmethode in der Algorithmik[7] und wird rekursiv implementiert. Dabei geht das Verfahren nach dem trial-and-error-Prinzip vor. Wenn nach einem Lösungsschritt erkennbar ist, dass der aktuelle Weg nicht zur Endlösung führt, geht der Algorithmus einen oder mehrere Schritte zurück, bis eine andere Teillösung ausprobiert werden kann. Dadurch werden alle Teillösungen getestet, die zur Lösung des Problems führen könnten. Wurde keine Lösung gefunden, kann mit Sicherheit gesagt werden, dass es keine gibt.

2.4 Lösungsmethoden

2.4.1 Brute force Backtracking

Eine Möglichkeit ein Sudoku mit dem Computer zu lösen, ist das Backtracking. Der Computer besetzt dabei nun immer das erste leere Feld mit einer erlaubten Zahl, bis er die Lösung findet[8]. Angenommen ein Sudoku hat noch 70 freie Felder und für jedes Feld gibt es 5 mögliche Zahlen, dann gäbe es 5^{70}, also über $8*10^{48}$ Kombination. Früher wurde das Lösen eines Sudokus mit roher Gewalt(englisch brute force), also durch ausprobieren aller Kombinationen, für unmöglich gehalten, bis der erste Versuch sehr zügig eine Lösung lieferte. Dabei muss man bedenken, dass der Computer bei weitem nicht alle Kombination durchgehen wird. Ein Feld ist nicht lösbar, wenn es eine frei Stelle gibt, bei der die Menge, der in Frage kommenden Zahlen 0 ist. Dieser Punkt ist nicht erst bei der letzten freien Stelle erreicht, sonder viel früher. Dadurch fallen eine Vielzahl an Kombi-

nationen raus. Auch wird nach jeder eingesetzten Zahl, die Anzahl der Möglichkeiten für die übrigen freien Felder kleiner. Deshalb verringert sich die Anzahl der Kombinationen auf eine überschaubare Menge, womit das Sudoku Rätsel lösbar wird.

2.4.2 Brute force Backtracking mit dynamischer Reihenfolge

Wie man sich denken kann, ist das Einsetzen in die erste freie Zelle nicht die effektivste Methode. Um die Laufzeit zu optimieren, bietet es sich an, die Zahlen in einer dynamischer Reihenfolge in das Sudoku einzutragen. Ein entscheidendes Kriterium ist dabei die Kandidatenmenge. Die Kandidatenmenge ist die Menge aller möglichen Zahlen, die nach den Sudoku Regeln in ein freies Feld kommen könnten. Hat ein Feld die Kandidatenmenge der Größe 1, ist dies die einzig richtige Zahl an dieser Stelle. Setzt man diese zuerst ein, verringert sich die Kandidatenmenge für die restlichen Felder und die Lösung sollte schneller gefunden werden. Der Aufwand hierbei liegt in der im Auffinden des ersten freien Feldes, mit der geringsten Kandidatenmenge. Im Gegensatz zum normalen Brute force Backtracking wird hier also nicht nur rohe Gewalt angewendet.

2.4.3 Einfaches Lösen (Exact Cover) mit Backtracking

Jedes Sudoku ist bis zu einem bestimmten Punkt lösbar. Einfache Sudokus kann man lösen, ohne dabei ein freies Feld raten zu müssen. Bei erhöhter Schwierigkeit kann es sein, dass man ein oder mehrere Male ein Feld mit einer möglichen ausgewählten Zahl belegen muss, wenn es kein Feld mit weniger als zwei Möglichkeiten gibt. Eine andere Lösungsmethode für Sudokus ist es, das Sudoku zunächst soweit zu lösen wie es geht, ohne dabei Brute Force anzuwenden. Also wird in jedes Feld mit der Größe der Kandidatenmenge 1, die entsprechende Zahl eingetragen. Diesen Vorgang nenne ich einfaches Lösen. Falls kein leichtes Sudoku vorliegt ist es nun noch nicht fertig. Dann wird das Sudoku mit Brute force Backtracking bis zum Schluss gelöst. Diese Methode nennt sich Exact Cover[9].
Eine Abwandlung davon wäre, statt das komplette Sudoku nach dem einfachen Lösen nicht mit Brute force Backtracking zu lösen, sondern nur eine Mögliche Zahl einzusetzen, um dann zu prüfen, ob das einfache Lösen eine Lösung findet. Ist das Feld dann noch nicht vollständig gelöst, wird ein weiteres eingesetzt. Dies wird auch rekursiv implementiert. Kommt es also zu keiner Lösung, springt die Methode ein oder mehrere Schritte zurück.

3 Implementation

3.1 Aufbau

Für das Programm werden eigentlich nur zwei Klassen benötigt, trotzdem ist es sinnvoll die Hauptmethoden von den Hilfsmethoden zu trennen. Also gibt es drei Klassen: Hilfsmethoden, Sudoku und GUI. Die Klasse Sudoku beinhaltet die Hauptmethoden, welche ein gelöstes Sudoku zurück geben. Die Klasse Hilfsmethoden beinhaltet alle Methoden welche den Hauptmethoden dazu dienen, das Sudoku zu lösen. Sowohl die Hilfsmethoden, als auch die Hauptmethoden werden statisch aufgerufen, da es meistens nur einen Parameter gibt, der sich ständig ändert.
Die graphische Benutzeroberfläche soll ein Fenster mit einem 9x9 Gitter zur Eingabe des Sudokus öffnen. Das Sudoku wird als 9x9 Integer Array abgespeichert. Es gibt mehrere Schaltflächen, ein Ausgabefeld, eine Fortschrittsleiste und eine Auswahlleiste für die Wahl der Lösungsmethode. Die GUI verwaltet das angegebene Sudoku und benutzt dabei die statischen Methoden der beiden anderen Klassen.

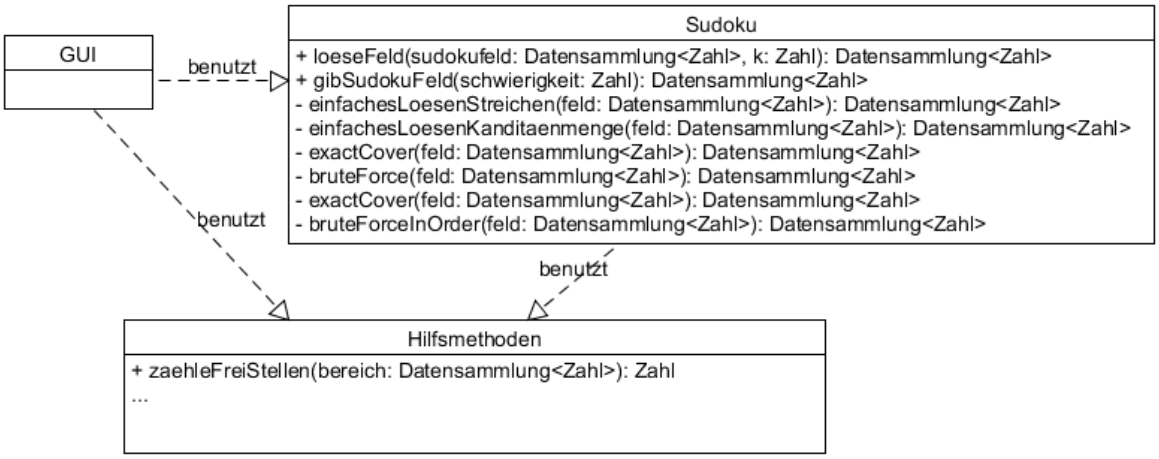


Abbildung 2: UML-Diagramm zum Aufbau

3.2 Sudoku Klasse

In der Sudoku Klasse wurden 3 rekursive Lösungsverfahren, 2 einfache Lösungsverfahren und eine Methode zum Erstellen von Sudokus implementiert. Im Folgenden werde ich erläutern wie diese funktionieren.

3.2.1 Iteratives/einfaches Lösen durch Streichen

Die zwei einfachen Lösungsverfahren orientieren sich an der Vorgehensweise eines Menschen. Sie werden benötigt um ein Feld soweit zu lösen wie möglich, ohne ein Feld zu

raten. Außerdem kann damit überprüft werden, ob ein Sudoku einfach für einen Menschen lösbar ist. In Abschnitt 4 wird die Effizienz der Verfahren getestet.

Zuerst beschreibe ich, wie die Methode einfachesLoesenStreichen() vorgeht. Um herauszufinden, ob eine Zahl in ein Feld gehört, schaut man sich die Umgebung an. Man überprüft den 3x3 Block(im Quelltext Kasten genannt), die Zeile und Spalte des freien Feldes. Nun sucht man alle Vorkommen der einzusetzende Zahl im Sudoku. Wenn man eine findet, streicht man an dieser Stelle gedanklich die Spalte, die Zeile und den Block. So geht man mit allen Stellen dieser Zahl vor. Wenn man Glück hat, ist nun mindestens einer dieser drei Bereiche, der ausgesuchten freien Stelle, komplett gestrichen. Nun weiß man, dass dies die Stelle ist, wo diese Zahl hingehört. Wie kann man das nun als Methode anwenden? Die Idee ist, ein Feld zu nehmen und eine Zahl komplett zu streichen. Dafür iteriert man das Feld durch und streicht bei jedem Vorkommen der Zahl sowohl Spalte, Zeile als auch Block. Dann sucht man die Bereiche in denen es genau eine freie Stelle gibt. Da gehört nun die Zahl hin. Das wiederholt man solange für alle neun Zahlen, bis sich das Feld nicht mehr ändert. In dem Fall ist es nun gelöst oder nicht mehr weiter zu lösen.

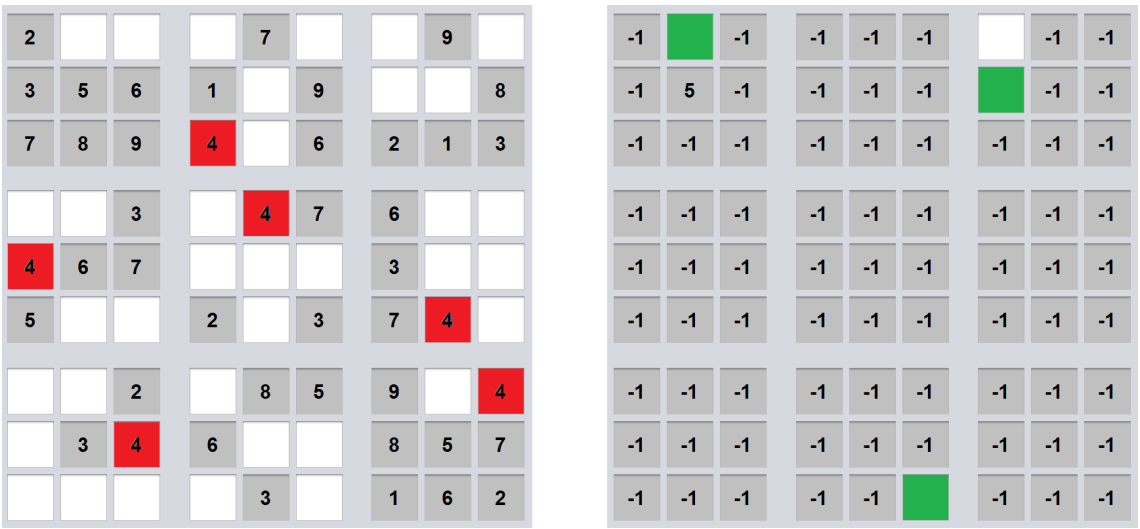


Abbildung 3: Beispiel für ein gestrichenes Feld der Zahl 4

Wie man zum Beispiel in Abbildung 3 sehen kann, wurden alle Stellen der Zahl ”4” rot markiert. Dann wurde bei jeder Stelle, wo die Zahl nicht mehr eingesetzt werden darf, die Umgebung(Kasten, Spalte und Zeile) durch die Zahl -1 ersetzt. Dabei ist es egal, ob die Felder bereits mit Zahlen belegt sind oder nicht. Es kann also vorkommen, dass trotzdem noch andere Zahlen außer der -1 vorhanden sind. Das macht aber kein Unterschied, da man später nur guckt, ob eine Stelle frei, also im Integer-Array mit 0 belegt ist. Das sind hier im Beispiel die grün markierten Felder, in denen die Zahl ”4” im Originalfeld nun eingefügt werden muss.

Außerdem wird überprüft, ob das Sudoku Feld noch allen Regeln entspricht. Eingabefehler werden zwar vorher von der GUI abgefangen, aber trotzdem kann es sein, dass durch eine geratene Zahl des rekursiven Lösens so ein Fehler auftritt, obwohl nur mögliche Zahlen eingesetzt werden. Zum Beispiel kann es vorkommen, dass in einem Kasten zwei Mal dieselbe Zahl steht, weil sowohl in einer Spalte, als auch in einer Zeile, je eine freie Stelle ist, welche sich im gleichen Block befinden.
Abbildung 4 zeigt eine Struktogramm zur Implementation des einfachen Lösens mit dem Streichen der Zahlen.

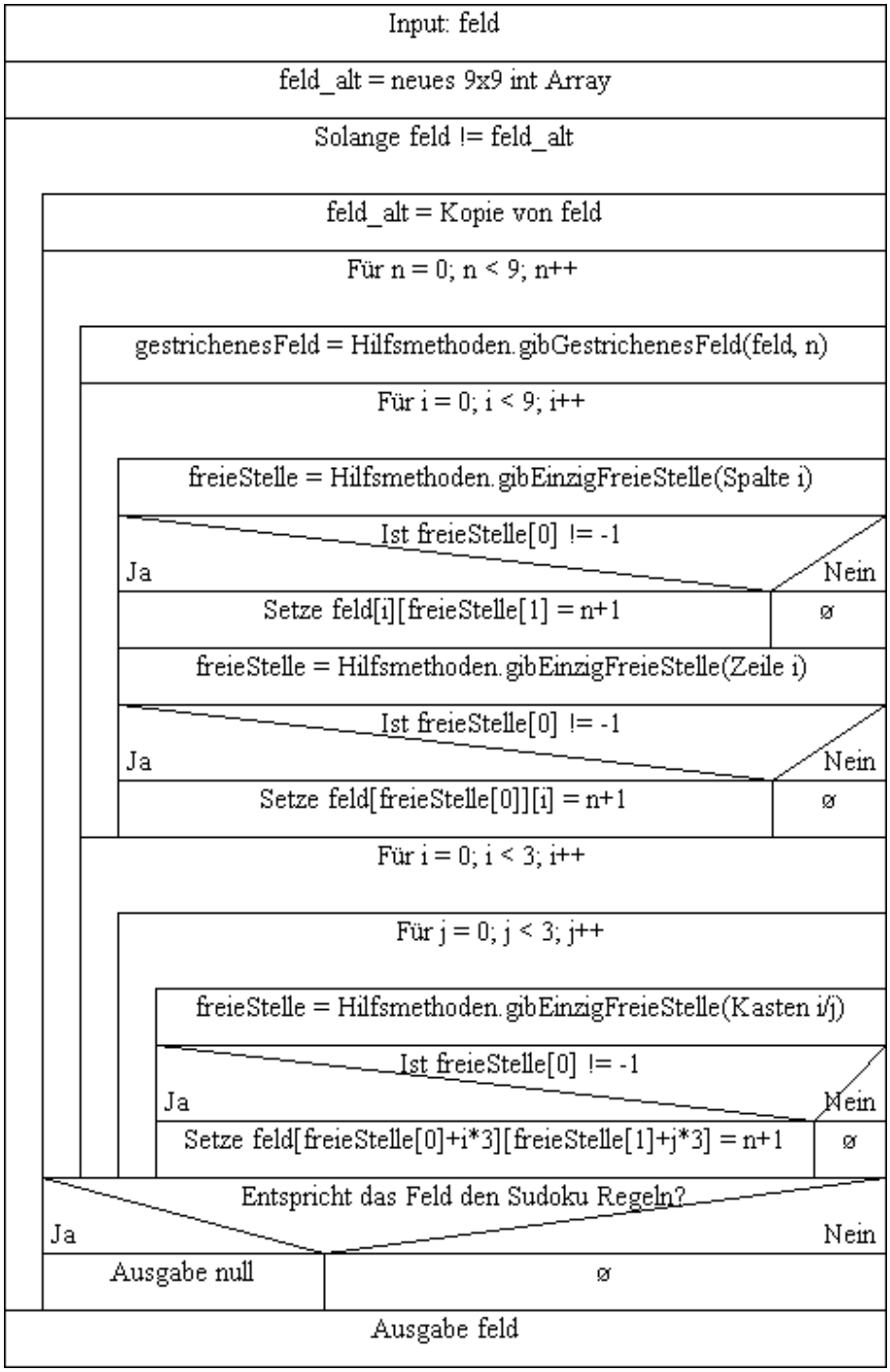


Abbildung 4: Algorithmus zum einfachen lösen eines Sudokus durch Streichen

3.2.2 Iteratives/einfaches Lösen durch die Kandidatenmenge

Die zweite Implementation des einfachen Lösens ist primitiver. Hierfür wird immer nur die erste Zahl aus der Kandidatenliste mit der Länge 1 eingesetzt. Dies wird solange wiederholt bis das Feld gelöst ist, oder es keine freien Stellen mit nur einer Möglichkeit gibt. Der Vorteil gegenüber der ersten Variante ist, dass nicht abgefragt werden muss, ob das Sudoku noch allen Regeln treu ist, da nur mögliche Zahlen eingesetzt werden.

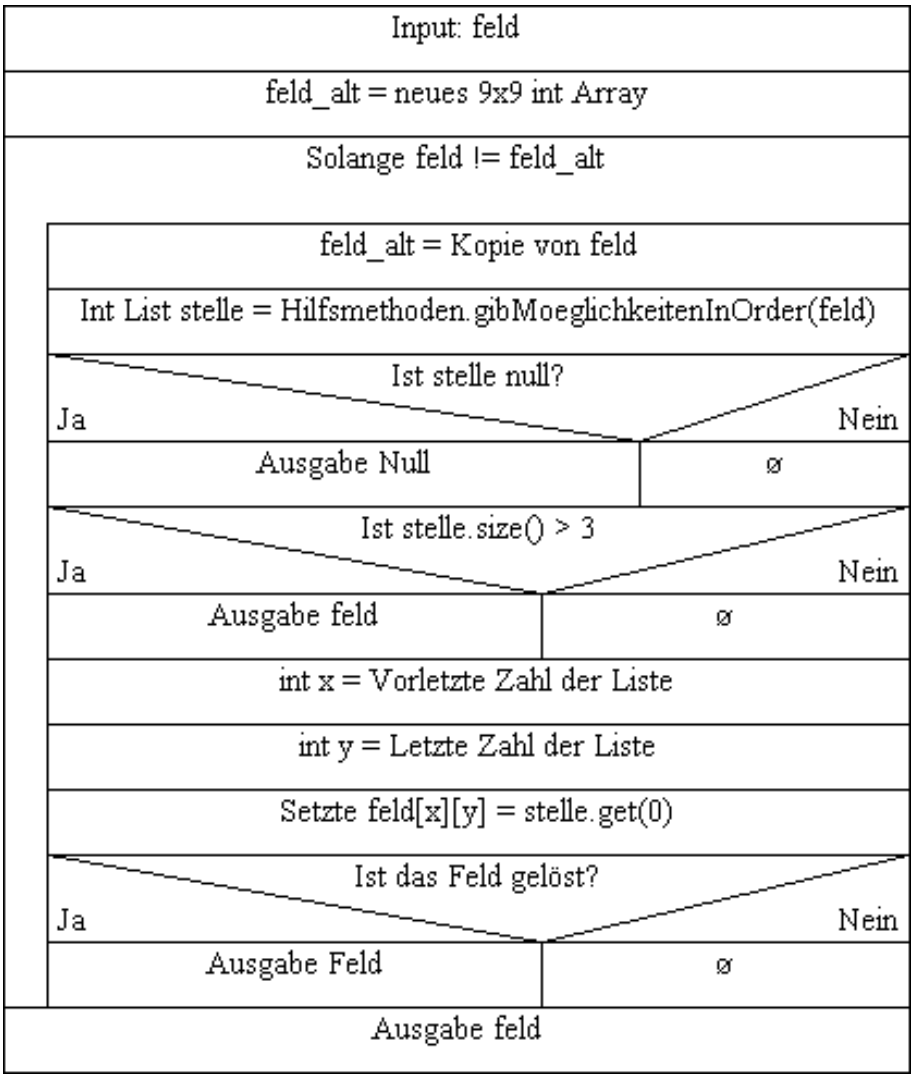


Abbildung 5: Algorithmus zum einfachen lösen eines Sudokus durch die Kandidatenmenge

Die Hilfsmethode gibMoeglichkeitenInOrder() ist das Kernstück dieser Variante. Sie gibt eine Liste mit den möglichen Kandidaten, von der Stelle mit der geringsten Anzahl an Möglichkeiten, zurück. An letzter und vorletzter Stelle der Liste stehen die Koordinaten. Nur wenn die Länge der Liste 3 beträgt, gibt es noch freie Felder mit nur einer Möglichkeit. Ist die Liste länger kann das Feld zurückgegeben werden, da es für diese Methode nicht mehr weiter zu lösen ist.

3.2.3 Exact Cover

Nach dem iterativen Lösen kann es sein, dass das Sudoku entweder gelöst ist, nicht komplett gelöst ist oder es einen Fehler gab, da das Feld unlösbar ist. Wenn es noch nicht komplett gelöst ist, dann deshalb, weil man mit Einsetzen und Streichen nicht mehr weiter kommt. Man muss nun Raten und dann prüfen, ob sich das Sudoku lösen lässt. Hier komm nun das Backtracking ins Spiel.
Man löst das Sudoku also ganz normal und muss dann zwei Fälle überprüfen. Wenn das Sudoku unmöglich ist, gibt das einfache Lösen null zurück und die Rekursion kann an dieser Stelle abgebrochen werden. Ansonsten wird die nächste Stelle eingesetzt, falls das Feld noch nicht gelöst ist. Abbildung 6 zeigt die Exact Cover Methode.

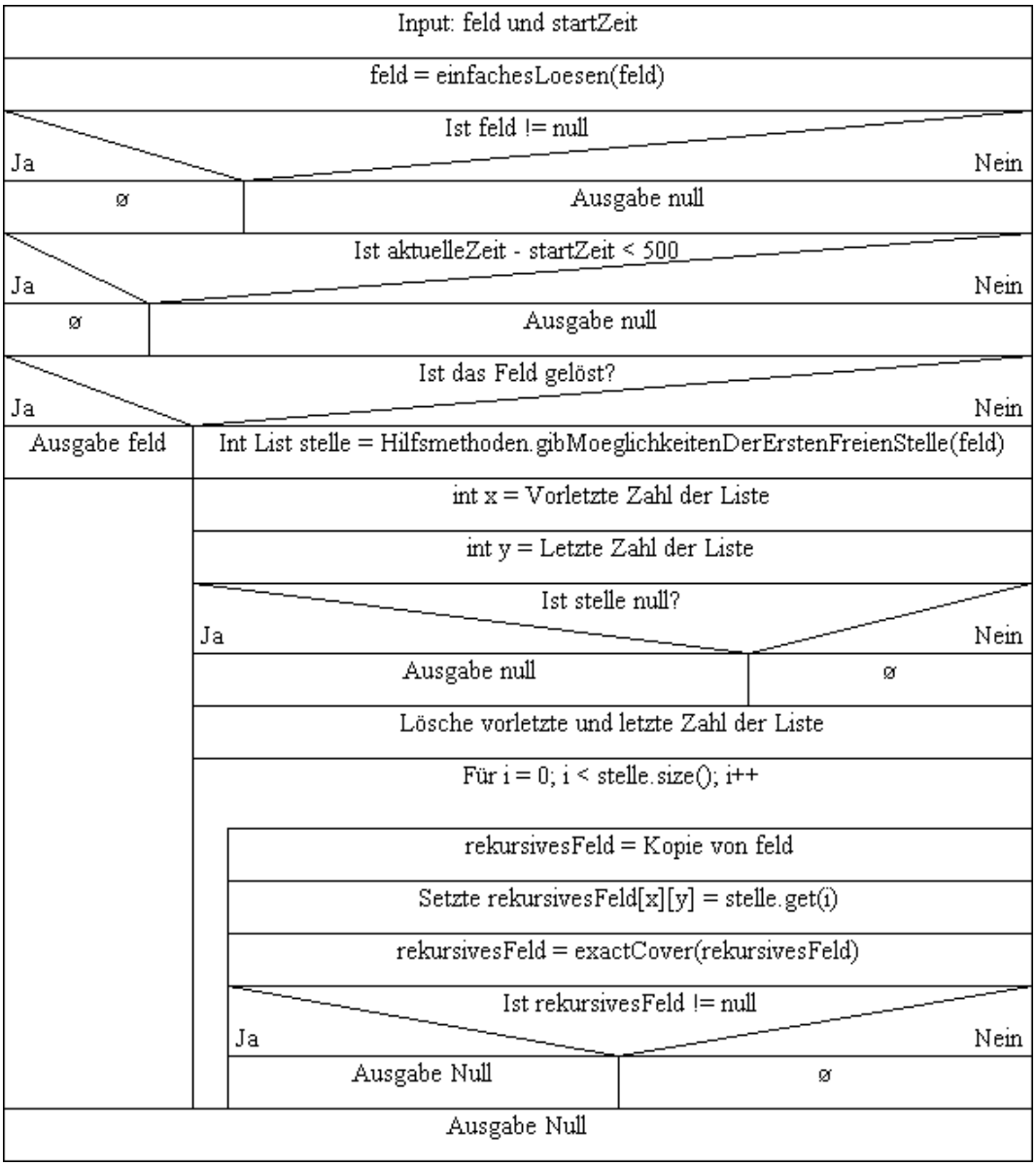


Abbildung 6: Struktogramm zur Exact Cover Methode

3.2.4 Brute force Backtracking

Die beiden Brute force Methoden funktionieren ohne das einfache Lösen. Dafür wird über Backtracking jede Möglichkeit eingesetzt, bis das Feld gelöst ist. Die bruteForce() Methode verwendet die Hilfsmethode gibMoeglichkeitenDerErstenFreienStelle() um die Kandidatenliste der erste freie Stelle zu bekommen. Der einzige Unterschied zur bruteForceInOrder() Methode liegt darin, dass sie gibMoeglichkeitenInOrder() nutzt.

3.2.5 Erstellung eines Sudokus

Die Methode gibSudokuFeld() der Klasse Sudoku soll ein spielbares Sudoku Feld mit der angegebenen Schwierigkeitsstufe als 9x9 Integer Array zurückgeben. Dabei ist zu beachten, dass genau eine Lösung vorliegt, die der Schwierigkeit entspricht.
Die erste Idee war es in jedes Feld der Reihe nach eine mögliche Nummer einzutragen und das ganze Sudoku ausfüllen. Dies kann jedoch trotz Regelkonformität ein unmögliches Feld ergeben, wie in Abbildung 7 zu sehen ist.

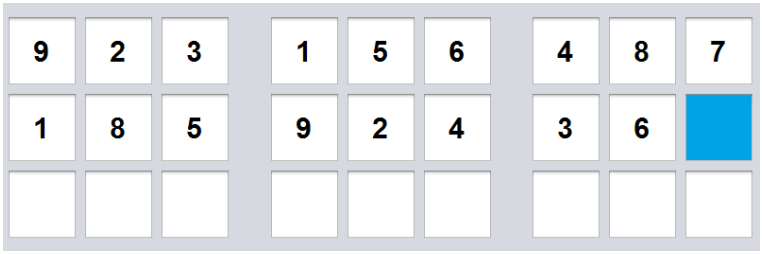


Abbildung 7: Beispiel 2: Obere Reihe eines Sudokus

Für das blau markierte Feld bleibt nur die ”7” übrig. Diese ist allerdings schon im Kasten vorhanden. Man könnte nun über Backtracking an den Punkt zurück, an dem man noch eine ”7” eintragen konnte. Backtracking ist aber für das Erstellen nicht nötig und auch aufwendiger, da mit jeder Zahl die Wahrscheinlichkeit wächst, dass ein Fehler auftritt. Dadurch dauert das Erstellen zu lange.
Stattdessen kann man 20 zufällige Zahlen auf zufällige Stellen verteilen und anschließend lösen. Ist das Feld nicht lösbar wird das Sudoku gelöscht und der Vorgang wiederholt. Die Erstellungszeit ist mit diesem Vorgehen deutlich geringer. Das Problem dabei ist, dass bestimmte Zahlen Kombinationen, wie zum Beispiel ”1-2-3” oben links, öfter auftauchen. Das liegt daran, dass diese Felder meistens frei bleiben und das rekursive Lösen die erst mögliche Zahl einsetzt.
Zur Lösung dieses Problems können unabhängige Zahlen eingetragen werden. Die drei diagonalen 3x3 Kästen beeinflussen sich nicht. Diese füllt man mit zufälligen Zahlen. Dann wird über eine der Lösungsmethoden das Sudoku zu Ende gelöst. Anschließend kann man je nach gewünschter Schwierigkeit eine bestimmte Menge an Zahlen löschen.
Die Schwierigkeitsstufen ”Einfach” hat 40 vorgegebene Zahlen, während ”Sehr Schwer” nur 28 hat. Die Stufe ”Unmöglich” hat nur 17 Zahlen und ist nicht für menschliche Spieler

gedacht, sondern als Test Szenario für den Vergleich von Algorithmen.
Um zu garantieren, dass die Stufe ”Einfach” und ”Normal” nur Sudokus liefern, welche ohne Raten möglich sind, wird zum Ende der Methode abgefragt, ob das Sudoku durch einfaches Lösen lösbar ist. Falls dies nicht der Fall ist, wird die Methode mit der gleichen Schwierigkeitsstufe rekursiv aufgerufen.

4 Vergleich der Lösungsmethoden

Um einen Vergleich über die Laufzeit der Algorithmen zum einfachen Lösen zu führen, habe ich eine Methoden in der Klasse Sudoku erstellt. Diese wurde später wieder entfernt. Die Algorithmen wurden getrennt getestet, damit die Messungen nicht verfälscht werden und wir ein genaues Ergebnis in Nanosekunden erhalten. Um die Zeit in Nanosekunden zu erhalten wurde System.nanoTime() benutzt. Die Lösungsgeschwindigkeit ist abhängig von der Leistungsfähigkeit des Computers.
In Abbildung 8 ist zu sehen, dass die Methode einfachesLoesenStreichen() einen Nachteil gegenüber der Methode einfachesLoesenKandidatenmenge() hat. Ein Test über 10000 Sudoku Felder hat gezeigt, dass die Methode mit der Kandidatenliste je nach Schwierigkeit immer um 200 bis 300 Mikrosekunden schneller ist.

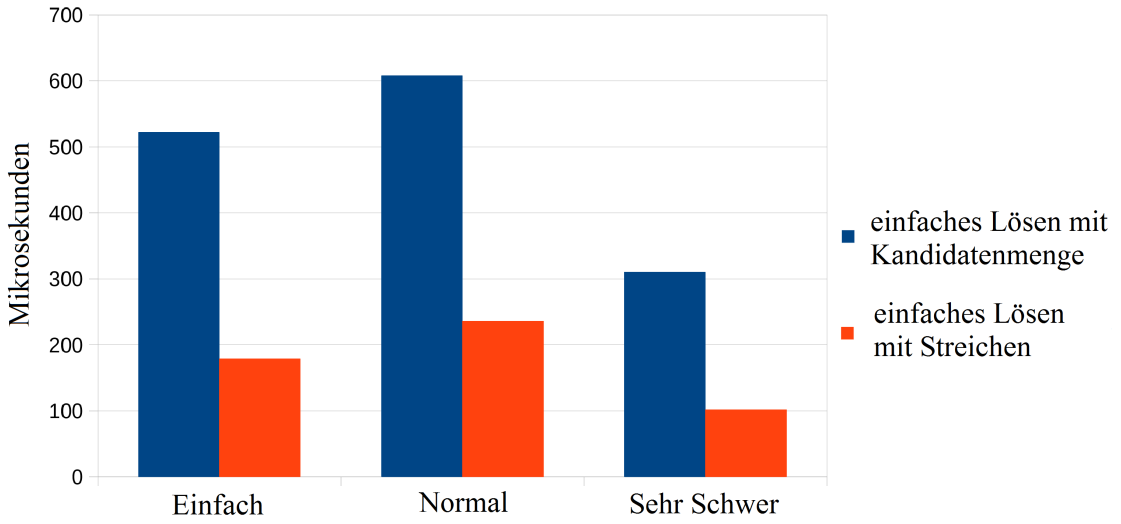


Abbildung 8: Säulendiagramm zur Darstellung der durchschnittlichen Laufzeit(auf 10000 Test-Felder) des einfachen Lösens

Das einfache Lösen durch Streichen benötigt fast doppelt so lange zum Lösen für ein Sudoku. Bei einem Unterschied von 300 Mikrosekunden wirkt das zwar sehr gering, allerdings wird das einfache Lösen für schwere Sudokus bei der Exact Cover Methode bis zu 100 Mal aufgerufen, was zu einer Verzögerung von 3 Millisekunden bei Verwendung der langsameren Methode führen könnte.

Nun geht es um die Laufzeitanalyse der rekursiven Methoden. Dafür wurde in der GUI eine Statistikmethode erstellt, welche alle Messungen getrennt ausführt. Alle drei Methoden, Exact Cover, Brute force Backtracking und Brute force Backtracking in dynamischer Reihenfolge wurden auf alle 5 Schwierigkeitsstufe über 500 Sudokus getestet. Nach mehreren Durchläufen konnte man eine ungefähre Übereinstimmung der Ergebnisse feststellen.

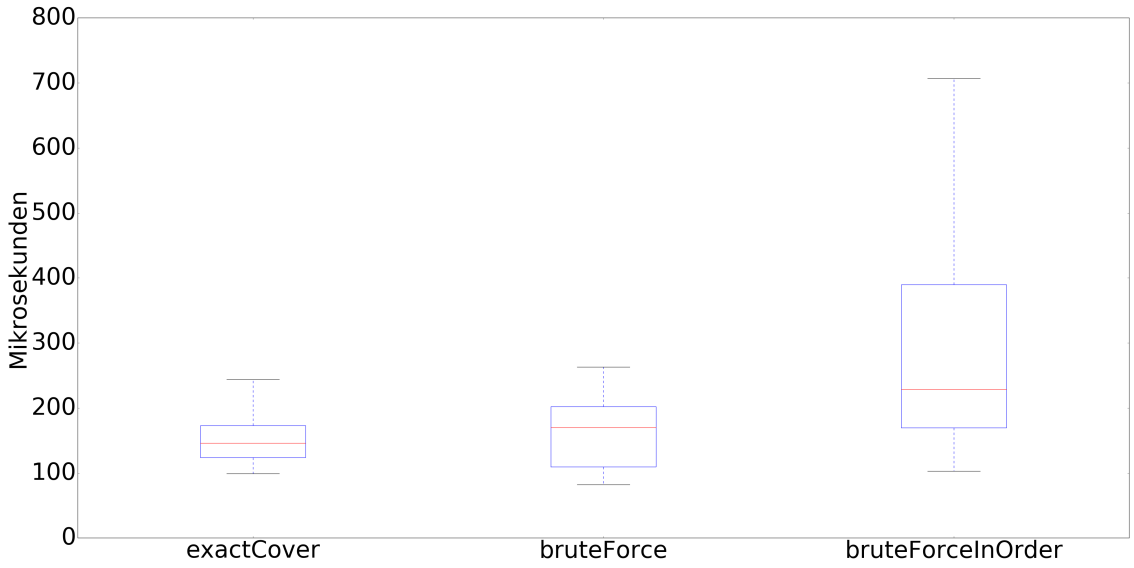


Abbildung 9: Boxplot zur Verteilung der Laufzeiten der Methoden auf Stufe Einfach

Abbildung 9 visualisiert die statistische Verteilung der Laufzeiten zu Sudokus der Schwierigkeitsstufe ”Einfach” als Boxplot. Das Diagramm zeigt keine Ausreißer an. Ausreißer sind Datenpunkte welche größer als 97,5% der Datenwerte sind. Wie man sehen kann hat der ”bruteForceInOrder” die größte Spannweite. Somit ist diese Methode die ineffizienteste für die Stufe ”einfach” ist Der Median liegt mit circa 260µs am höchsten. Auch die durchschnittliche Laufzeit mit 296µs bestätigt das.
Im Gegensatz dazu haben der Brute force und der Exact Cover mit ungefähr 150µs die geringste Spannweite. Der Exact Cover ist in diesem Fall ein Stück schneller als der Brute force. Seine durchschnittliche Laufzeit beträgt 167µs. Außerdem ist er die Stabilste Methode, da er keinen Ausreißer hat. Beide Brute force Methoden können eine Abweichung von bis zu 1000µs vom Median haben. Zurückzuführen ist der Vorteil darauf, dass auf Stufe ”Einfach” die Sudokus ohne Raten möglich sein müssen. Hier greift die Methode zum Iterativen Lösen, welche schneller als eine Rekursion ist.

Nach der Schwierigkeitsstufe ”Normal” konnte der Brute force nicht mehr in die Statistik mit ein bezogen werden. Ab dieser Stufe kann er zwar Sudokus in geringer Zeit lösen, allerdings ist dies nicht mehr der Regelfall. Auch seine durchschnittliche Laufzeit lag dadurch sehr weit über den anderen. Auch zeigen alle Methoden Ausreißer, welche mit zunehmender Schwierigkeit weiter vom Median abweichen.

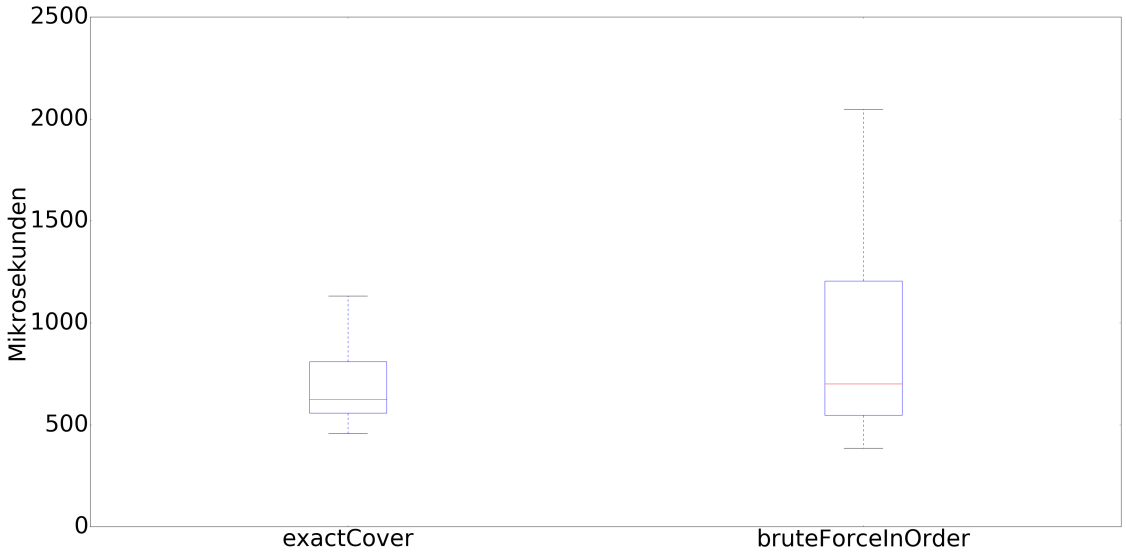


Abbildung 10: Boxplot zur Verteilung der Laufzeiten der Methoden auf Stufe Sehr Schwer

Im zweiten Boxplot, welcher in Abbildung 10 dargestellt wird, erkennt man, dass der ”bruteForceInOrder” eine größere Spannweite nach oben, als der Exact Cover hat und somit der langsamere ist. Ihr Median liegt zwar auf der gleichen Ebenen, aber ihr durchschnittlicher Wert unterscheidet sich um 300 Mikrosekunden.

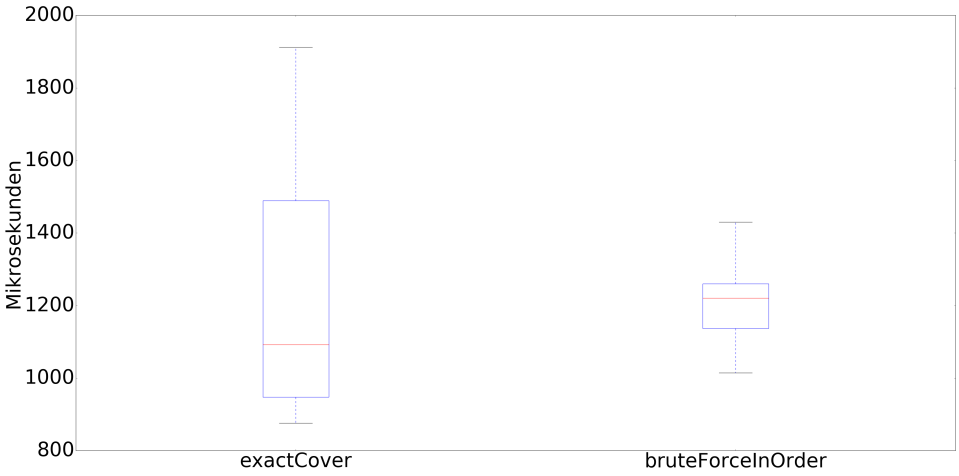


Abbildung 11: Boxplot zur Verteilung der Laufzeiten der Methoden auf Stufe Unmöglich

In Abbildung 11 stellt sich aber der wirkliche Sieger auf Stufe ”Unmöglich” heraus. Auch nach mehrfacher Wiederholung der Messung, bleibt der Brute force In Order deutlich effektiver, als der Exact Cover.

5 Fazit

In meiner Facharbeit ging es um die Entwicklung einer JAVA-Anwendung für Sudokus. Diese sollte über einen Auto-Solve-Algorithmus verfügen. Hierfür habe ich drei Algorithmen implementiert, welche alle funktionieren und für die Anwendung brauchbar sind. Dazu wurde ebenfalls eine komfortabel zu Bedienende GUI entwickelt. Der Vergleich der Laufzeiten zeigt, dass alle drei Lösungsalgorithmen ihre Vor- und Nachteile haben. Der Exact Cover ist der stabilste Algorithmus, was die Schwankungen in der Laufzeit betrifft. Er ist bei einfachen Sudokus der schnellste, während der Brute force In Order für die Lösung von Sudokus mit nur 17 Zahlen die beste Wahl ist. Der einfache Brute force kann auf dieser Stufe nicht mehr mithalten, kann aber trotzdem für einfachere Sudokus verwendet werden.

Literatur

[1] Sudoku Regeln. http://www.conceptispuzzles.com/de/index.aspx?uri=puzzle/sudoku/rules
Eingesehen am 22.04.2017

[2] Der Heilige Gral der Sudokus. http://www.faz.net/aktuell/wissen/physik-mehr/mathematik-der-heilige-gral-der-sudokus-11682905.html
Eingesehen am 22.04.2017

[3] Java(Programmiersprache). https://de.wikipedia.org/wiki/Java_(Programmiersprache)
Eingesehen am 23.04.2017

[4] BlueJ. https://de.wikipedia.org/wiki/BlueJ
Eingesehen am 23.04.2017

[5] NetBeans IDE. https://de.wikipedia.org/wiki/NetBeans_IDE
Eingesehen am 23.04.2017

[6] Iteration und Rekusion. http://www.java-tutorial.org/iteration_und_rekursion.html
Eingesehen am 23.04.2017

[7] Backtracking. http://www.java-tutorial.org/iteration_und_rekursion.html
Eingesehen am 23.04.2017

[8] Algorithmische Lösungsmethoden für das Sudoku. https://de.wikipedia.org/wiki/Sudoku#Analytisch-systematische_Basismethoden
Eingesehen am 23.04.2017

[9] Exact Cover. https://de.wikipedia.org/wiki/Sudoku#Exact_Cover
Eingesehen am 23.04.2017

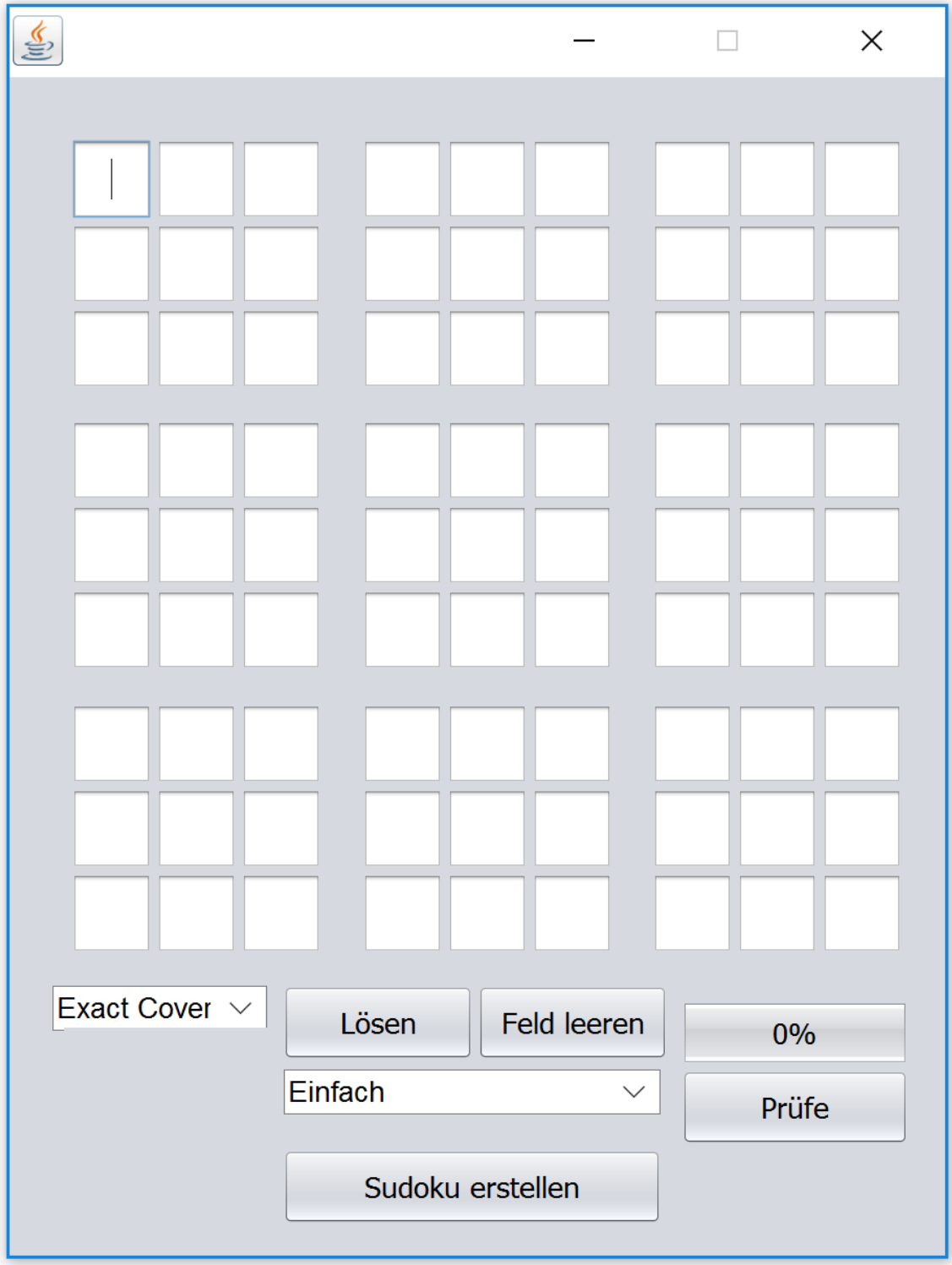
Erklärung

Ich erkläre, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Rheinbach, 27. April 2017

Erik Springer

Anhang



- Bild von der GUI

Class Hilfsmethoden

java.lang.Object
  Hilfsmethoden

public class **Hilfsmethoden**
extends java.lang.Object

Constructor Summary

Hilfsmethoden()

Method Summary

static int[][]	**belegeBereich**(int xa, int xe, int ya, int ye, int[][] feld) Belegt einen Bereich mit den angegebenen Koordinaten
static boolean	**feldMoeglich**(int[][] feld) Prüft ob ein Feld noch allen Regeln des Sudokus entspricht.
static boolean	**geloest**(int[][] feld) Gibt true zurück, falls in jedem Kasten, jeder Zeile und jeder Spalte die Zahlen 1 bis 9 enthalten sind
static int[]	**gibEinzigFreieStelle**(int[][] bereich) Falls es nur eine freie Stelle gibt wird diese als Array zurückgegeben Ansonsten werden die Koordinaten -1/-1 zurückgegeben
static java.util.ArrayList<int[]>	**gibFehlerStellen**(int[][] sudokuFeld) Gibt eine Liste mit den Koordinaten der Stellen zurück, welche einen Fehler verursachen würden
static int[][]	**gibGestrichenesFeld**(int[][] feld, int n) Streicht Zahl n aus dem Array, indem alle Zeilen, Spalten und Kasten mit der Zahl n durch -1 ersetzt werden
static int[]	**gibKasten**(int x, int y) Rechnet die Koordinaten des Kastens aus in der sich die angegebene Stelle befindet.
static int[][]	**gibLeeresFeld**() Gibt ein 9x9 Feld mit 0 initialisiert
static java.util.ArrayList<java.lang.Integer>	**gibMoeglicheZahlen**(int x, int y, int[][] feld) Gibt Liste mit der Kandidatenliste einer Stelle zurück
static java.util.ArrayList<java.lang.Integer>	**gibMoeglichkeitenDerErstenFreienStelle**(int[][] feld) Gibt eine Liste mit den Kandidaten und den Koordinaten der ersten freien Stelle zurück Falls keine Stelle gefunden wird, wird null zurück gegeben
static java.util.ArrayList<java.lang.Integer>	**gibMoeglichkeitenInOrder**(int[][] feld) Gibt eine Liste mit den Kandidaten und den Koordinaten der Stelle

	zurück, welche am wenigsten Kandidaten hat.
static java.util.ArrayList<int[]>	**gibStellenDerZahl**(int n, int[][] bereich) Durchläuft den übergebenen Bereich und speichert dabei jedes Vorkommen der Zahl n als Array der Länge 2 in einer Liste
static int[][]	**gibTeilbereich**(int xa, int xe, int ya, int ye, int[][] feld) Gibt einen Teilbereich mit den angegebenen Koordinaten aus dem Sudoku Feld zurück
static int[][]	**kopiere**(int[][] array) Gibt eine Kopie des übergebenen 2D Arrays zurück
static void	**print**(int[][] feld) Schreibt das übergebenen Feld auf die Konsole
static boolean	**pruefe**(int[][] bereich) Überprüft ob die Zahlen von 1 bis 9 in einem Array vorhanden sind
static boolean	**suche**(int[][] bereich, int x) Durchläuft den übergebenen Bereich und gibt true beim ersten Vorkommen der Zahl zurück
static java.util.ArrayList<java.lang.Integer>	**testeMoeglicheZahlen**(int[][] bereich, java.util.ArrayList<java.lang.Integer> zahlen) Löscht alle Zahlen des Bereichs aus der Zahlen Liste
static int	**zaehleFreieStellen**(int[][] bereich) Durchläuft den übergebenen Bereich und zählt dabei die freien Stellen
static int	**zaehleZahl**(int n, int[][] bereich) Durchläuft den übergebenen Bereich und zählt dabei jedes Vorkommen der Zahl n

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Hilfsmethoden

public **Hilfsmethoden**()

Method Detail

belegeBereich

public static int[][] **belegeBereich**(int xa,
int xe,
int ya,
int ye,
int[][] feld)

Belegt einen Bereich mit den angegebenen Koordinaten

Parameters:

xa - x-Koordianate Anfang
xe - x-Koordianate Ende
ya - y-Koordianate Anfang
ye - y-Koordianate Ende
feld - Sudoku Feld

Returns:

9x9 Int Array mit belegtem Bereich

---

feldMoeglich

public static boolean **feldMoeglich**(int[][] feld)

Prüft ob ein Feld noch allen Regeln des Sudokus entspricht.

Parameters:

feld - Sudoku Feld

Returns:

Wahrheitswert

---

geloest

public static boolean **geloest**(int[][] feld)

Gibt true zurück, falls in jedem Kasten, jeder Zeile und jeder Spalte die Zahlen 1 bis 9 enthalten sind

Parameters:

feld - Sudoku Feld

Returns:

Wahrheitswert

---

gibEinzigFreieStelle

public static int[] **gibEinzigFreieStelle**(int[][] bereich)

Falls es nur eine freie Stelle gibt wird diese als Array zurückgegeben Ansonsten werden die Koordinaten -1/-1 zurückgegeben

Parameters:

bereich - Ausschnitt eines 2D Arrays

Returns:

Koordinaten der freien Stelle

---

gibFehlerStellen

public static java.util.ArrayList<int[]> **gibFehlerStellen**(int[][] sudokuFeld)

Gibt eine Liste mit den Koordinaten der Stellen zurück, welche einen Fehler verursachen würden

Parameters:

feld - Sudoku Feld

---

gibGestrichenesFeld

public static int[][] **gibGestrichenesFeld**(int[][] feld,
int n)

Streicht Zahl n aus dem Array, indem alle Zeilen, Spalten und Kasten mit der Zahl n durch -1 ersetzt werden

Parameters:
feld - Sudoku Feld
Returns:
Gestrichenes Feld

---

gibKasten

public static int[] **gibKasten**(int x,
int y)

Rechnet die Koordinaten des Kastens aus in der sich die angegebene Stelle befindet.

Parameters:
x - x-Koordinate
y - y-Koordinate
Returns:
Koordinaten eines Kastens

---

gibLeeresFeld

public static int[][] **gibLeeresFeld**()

Gibt ein 9x9 Feld mit 0 initialisiert

Returns:
leeres 9x9 Array

---

gibMoeglicheZahlen

public static java.util.ArrayList<java.lang.Integer> **gibMoeglicheZahlen**(int x,
int y,
int[][] feld)

Gibt Liste mit der Kandidatenliste einer Stelle zurück

Parameters:
x - x-Koordinate
y - y-Koordinate
feld - Sudoku Feld
Returns:
Kandidatenliste

---

gibMoeglichkeitenDerErstenFreienStelle

public static java.util.ArrayList<java.lang.Integer> **gibMoeglichkeitenDerErstenFreienStelle**(int[][] feld)

Gibt eine Liste mit den Kandidaten und den Koordinaten der ersten freien Stelle zurück Falls keine Stelle gefunden wird, wird null zurück gegeben

Parameters:

feld - Sudoku Feld

Returns:

Kandidatenliste mit Koordinaten der Stelle am Ende

---

gibMoeglichkeitenInOrder

public static java.util.ArrayList<java.lang.Integer> **gibMoeglichkeitenInOrder**(int[][] feld)

Gibt eine Liste mit den Kandidaten und den Koordinaten der Stelle zurück, welche am wenigsten Kandidaten hat. Falls keine Stelle gefunden wird, wird null zurück gegeben

Parameters:

feld - Sudoku Feld

Returns:

Kandidatenliste mit Koordinaten der Stelle am Ende

---

gibStellenDerZahl

public static java.util.ArrayList<int[]> **gibStellenDerZahl**(int n,
int[][] bereich)

Durchläuft den übergebenen Bereich und speichert dabei jedes Vorkommen der Zahl n als Array der Länge 2 in einer Liste

Parameters:

n - gesuchte Zahl

bereich - Ausschnitt eines 2D Arrays

Returns:

Liste aus Arrays mit Koordinaten der Zahl n

---

gibTeilbereich

public static int[][] **gibTeilbereich**(int xa,
int xe,
int ya,
int ye,
int[][] feld)

Gibt einen Teilbereich mit den angegebenen Koordinaten aus dem Sudoku Feld zurück

Parameters:

xa - x-Koordianate Anfang

xe - x-Koordianate Ende

ya - y-Koordianate Anfang

ye - y-Koordianate Ende

feld - Sudoku Feld

Returns:

Teilbereich

---

kopiere

public static int[][] **kopiere**(int[][] array)

Gibt eine Kopie des übergebenen 2D Arrays zurück

Parameters:

array -

Returns:

Kopie eines 2D Arrays

---

print

public static void **print**(int[][] feld)

Schreibt das übergebenen Feld auf die Konsole

Parameters:

feld - Sudoku Feld

---

pruefe

public static boolean **pruefe**(int[][] bereich)

Überprüft ob die Zahlen von 1 bis 9 in einem Array vorhanden sind

Parameters:

bereich - Ausschnitt eines 2D Arrays

Returns:

Wahrheitswert

---

suche

public static boolean **suche**(int[][] bereich,
int x)

Durchläuft den übergebenen Bereich und gibt true beim ersten Vorkommen der Zahl zurück

Parameters:

bereich - Ausschnitt eines 2D Arrays

x - gesuchte Zahl

Returns:

Wahrheitswert

---

testeMoeglicheZahlen

public static java.util.ArrayList<java.lang.Integer> **testeMoeglicheZahlen**(int[][] bereich,
java.util.ArrayList<java.lang.Integer> zahlen)

Löscht alle Zahlen des Bereichs aus der Zahlen Liste

Parameters:

bereich - Ausschnitt eines 2D Arrays

zahlen - Mögliche Zahlen

Returns:

ArrayList

---

zaehleFreieStellen

public static int **zaehleFreieStellen**(int[][] bereich)

Durchläuft den übergebenen Bereich und zählt dabei die freien Stellen

Parameters:

bereich - Ausschnitt eines 2D Arrays

Returns:

Anzahl der freien Stellen im Bereich

---

zaehleZahl

public static int **zaehleZahl**(int n,
int[][] bereich)

Durchläuft den übergebenen Bereich und zählt dabei jedes Vorkommen der Zahl n

Parameters:

n - gesuchte Zahl

bereich - Ausschnitt eines 2D Arrays

Returns:

Häufigkeit der Zahl n als Int
