Systematischer Vergleich von Algorithmen zum Lösen von Nurikabe-Rätseln

Bachelorarbeit

Nick Samuel Fath
353310

4. Oktober 2024

Gutachter: Prof. Dr. Benjamin Blankertz
Prof. Dr. Marc Alexa

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

Kurzfassung

Nurikabe ist ein noch relativ unbekanntes japanisches Puzzle-Rätsel, das seit dem Nachweis der np-Vollständigkeit ein beliebteres Forschungsthema geworden ist. So wurden in den letzten Jahren einige neue Ansätze zum Lösen von Nurikabe entwickelt. In dieser Abschlussarbeit werden drei dieser Algorithmen miteinander systematisch verglichen: ein Ansatz basierend auf *Ant Colony Optimization*, ein deterministischer Ansatz basierend auf *Constraint Programming* und ein kombinierter Ansatz aus *Scatter Search* und *Variable Neighbourhood Search*. Diese werden implementiert und auf eine Vielzahl von Rätselinstanzen angewandt.

Die Ergebnisse offenbaren Stärken sowie Schwächen: Ant Colony Optimization hat eine gute Löserate, tendiert jedoch, wenn auch selten, zu langwieriger Stagnation in lokalen Optima. Constraint Programming überzeugt durch die Robustheit, bei größer werdenden Instanzen nimmt jedoch erwarteterweise die Laufzeit stark zu. Der kombinierte Ansatz aus Scatter Search und Variable Neighbourhood Search ist insbesondere in kleinen Instanzen effizient, bei großen Instanzen konvergiert der Algorithmus nicht schnell genug zur Lösung. Hieraus lassen sich mögliche Optimierungsansätze ableiten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Nurikabe ist ein im Vergleich zu Sudoku außerhalb von Japan noch wenig bekanntes japanisches Puzzle-Rätsel der Kategorie *pencil & paper puzzle* (Stift und Papier-Puzzle). In den letzten Jahren hat es an Beliebtheit gewonnen und rückte dadurch vermehrt in den Fokus von wissenschaftlichen Arbeiten. So haben Holzer, Klein und Kutrib (2008) die np-Vollständigkeit von Nurikabe und Nurikabe-Varianten nachgewiesen [1].

Als np-vollständiges Puzzle ist die Entwicklung von Algorithmen, die das Puzzle in vielen Fällen in annehmbarer Zeit lösen, eine Herausforderung. Diese Arbeit befasst sich mit drei solchen Ansätzen: Die Metaheuristik *scattered neighbourly approach* (Scattered Variable Neighbourhood Search, SVNS) von Bass und Sevkli (2020)[2], die Metaheuristik Ant Colony Optimization (ACO)-Metaheuristik von Dorigo und Di Caro (1999) [3], sowie einem Constraint Programming-Ansatz nach Tamura [4]. Ein übergreifender Vergleich dieser Algorithmen steht zum Zeitpunkt der Arbeit aus. Ziel der Arbeit ist es, die Effektivität und Effizienz dieser Algorithmen gegenüberzustellen. Hierfür werden die Algorithmen implementiert und auf eine Auswahl an Testinstanzen angewandt.

Zunächst führt die Einleitung weiter in die Puzzle-Regeln und die drei Algorithmen ein. Im zweiten Kapitel wird die Methodik erläutert. Die Funktionsweise der Algorithmen wird präzisiert, die Testumgebung und Testinstanzen werden aufgeführt. Das dritte Kapitel befasst sich mit der Präsentation der Ergebnisse, die im vierten Kapitel ausgewertet werden. Das fünfte und letzte Kapitel zieht ein Fazit. Im Anhang finden sich detaillierte Infos über die genutzten Testinstanzen und Testergebnisse.

1.1 Nurikabe

1.1.1 Geschichte

Nurikabe ist ein japanisches Puzzle-Rätsel, das erstmals 1991 [5] bei Nikoli, einem japanischen Verlag mit Fokus auf Rätsel und bekannt für die Verbreitung von Sudoku [6], erschienen ist. Der Name basiert auf einem Fabelwesen der japanischen Mythologie. In dieser ist ein Nurikabe ein Geist, der dafür berüchtigt ist, sich als unüberwindbares Hindernis zu manifestieren und insbesondere nachts Passanten den Weg zu versperren. Diese Eigenschaft ist Kernidee von Nurikabe dem Rätsel.

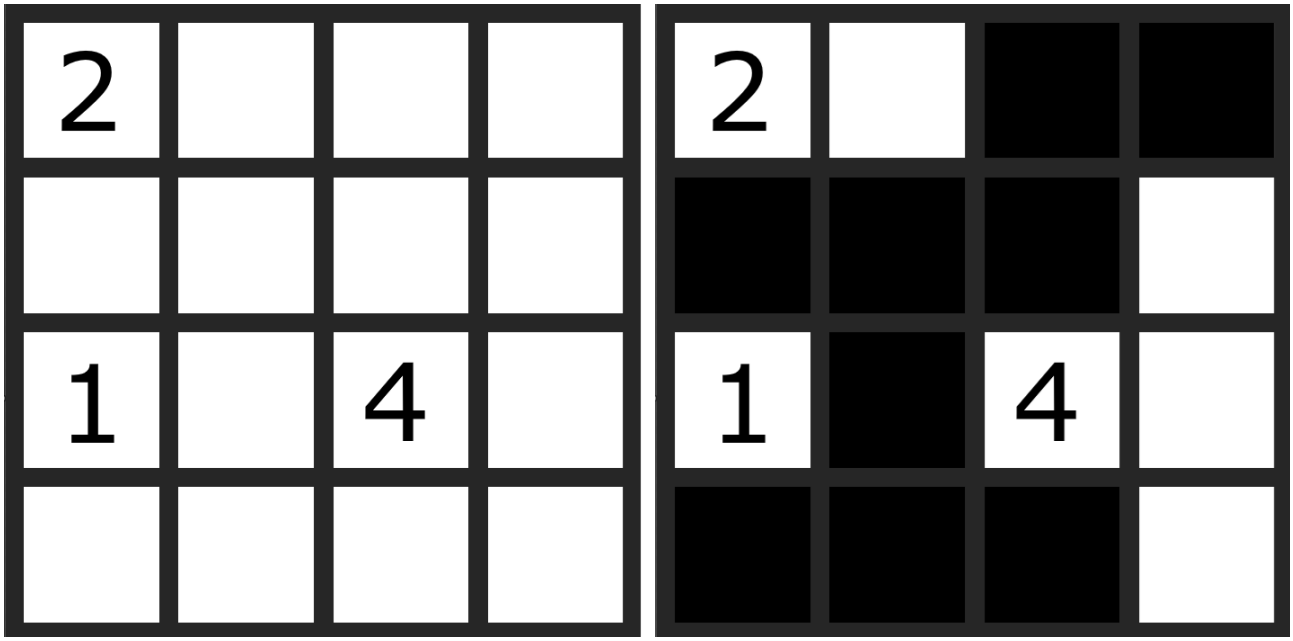

Abbildung 1.1: Ein Nurikabe-Rätsel im Ursprungszustand (links) und gelöstem Zustand (rechts)

1.1.2 Rätselaufbau

Ein Nurikabe-Rätsel besteht wie Sudoku aus einem Gitter, wobei die Anzahl an Zeilen und Spalten je nach Rätsel variieren und nicht identisch sein müssen. Ein Teil der Zellen des Gitters sind im Initialzustand leer. Die restlichen Zellen sind mit einer positiven ganzen Zahl gefüllt.

1.1.3 Regeln

Die mit einer Zahl gefüllten Felder symbolisieren die Wurzel einer Insel. Die Zahl schreibt die Größe der Insel im Lösungszustand vor. Eine Insel besteht aus genau einer Wurzel und alle Zellen einer Insel sind horizontal oder vertikal miteinander verbunden. Die Inselwurzel selbst zählt als Inselfeld.

Zwischen den Inseln symbolisieren die schwarz auszufüllenden Zellen den *Nurikabe*-Geist, der die Inseln voneinander trennt. Naheliegenderweise werden diese Zellen häufig, wie auch im Folgenden, als Seezellen bezeichnet.

Alle Seezellen müssen eine geschlossene Einheit bilden, sie sind also durchgehend miteinander horizontal oder vertikal verbunden. Es darf zudem keinen zusammenhängenden Block (2x2-Block) aus Seezellen geben, wobei ein zusammenhängender Block als ein Quadrat aus vier Zellen verstanden wird.

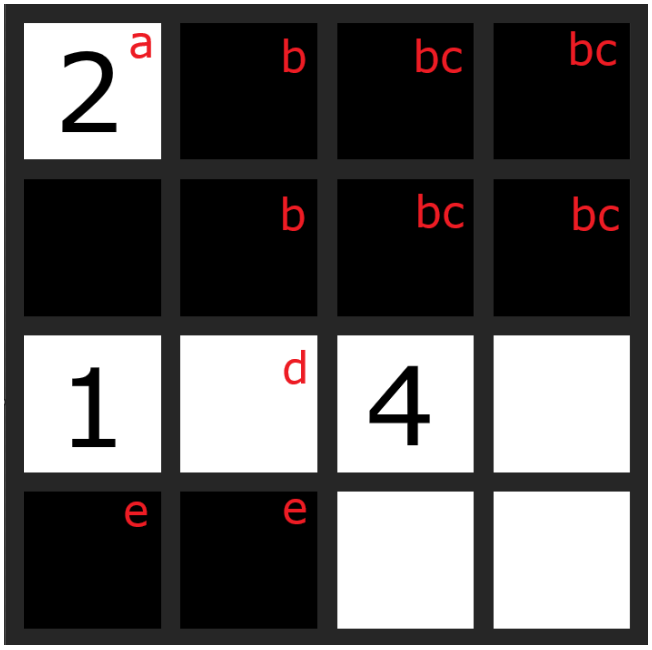

Abbildung 1.2: Ein Nurikabe-Rätsel mit diversen Regelverstößen. (a) zu kleine Insel, (b) und (c) jeweils 2x2-Blöcke, (d) die Inselzelle verbindet zwei Inseln (e) ein Beispiel eines Seefragments, das nicht mit dem anderen Fragment verbunden ist

1.1.4 Eigenschaften von Nurikabe-Rätseln

Eindeutigkeit

Generell sind Instanzen von Nurikabe-Rätseln eindeutig lösbar. Dies ist jedoch nicht inherent der Fall, so existieren auch Rätsel mit mehreren Lösungen [7].

np-Vollständigkeit

Die np-Vollständigkeit von Nurikabe hat McPhail (2003) durch einen Reduktionsbeweis auf das Erfüllbarkeitsproblem für Schaltkreise (CSAT) nachgewiesen [8]. Holzer, Klein und Kutrib (2018) [1] haben durch einen Reduktionsbeweis auf 3-SAT ebenso die np-Vollständigkeit nachgewiesen. Weiter haben Sie gezeigt, dass Nurikabe eingeschränkt auf Inseln der Größe 1 und 2 np-vollständig ist. Zudem bewiesen sie die np-Vollständigkeit der Nurikabe-Varianten, die entweder fragmentierte Seegebiete, 2x2-Blöcke der Seezellen oder beides erlauben.

1.2 Algorithmen

1.2.1 Ant Colony Optimization

Die Ant Colony Optimization (ACO)-Metaheuristik wurde erstmals von Dorigo und Di Caro (1999) [3] als Erweiterung des von Dorigo und Gambardella (1997) eingeführten Ant Colony System [9] vorgestellt, das wiederum aus dem von Dorigo, Maniezzo und Colorni (1991) initiell präsentierten Ant System [10] hervorging. Amos und Lloyd (2019) wandten zunächst eine Variante der ACO zum Lösen von Sudoku an [11]. Zudem haben Amos, Crossley und Lloyd (2019) selbiges Prinzip auf Nurikabe-Rätsel angewandt [12].

ACO, Ant Colony System und Ant System sind naturbasierte Algorithmen, die das Verhalten von Ameisenkolonien bei der Futtersuche immitieren. Ameisen hinterlassen beim Fortbewegen Pheromonspuren. Andere Ameisen werden von diesen Pheromonen angezogen. Die Ameise, die am schnellsten bei der Futtersuche fündig wird, ist am schnellsten wieder im Ameisenbau angelangt, konnte auf ihrer gewählten Strecke also zweimal Pheromonspuren legen. Andere Ameisen sind nun stärker zu dieser Strecke hingezogen. So entstehen die allgemein bekannten Ameisenstraßen mit dem kürzesten Weg zur Futterquelle [13] [14].

Dieses Verhalten wird dadurch simuliert, dass für ein zu lösendes Suchproblem mehreren Agenten (Ameisen) lokale Kopien des Problems zur Verfügung gestellt werden. Die Auswahl der Felder, Kanten oder Knoten wird durch eine globale Pheromonmatrix geleitet, wobei jede Entscheidung genau ein eigner Pheromonwert in der Matrix zugeordnet wird. Entscheidungen mit höheren Pheromonwerten werden wahrscheinlicher gewählt. Diese Pheromonmatrix steht allen Ameisen zur Verfügung. Die Ameisen erstellen pro Iteration (Generation) jeweils einen Lösungskandidat. Diese Lösungskandidaten werden nun bewertet, meist mit Hilfe einer Fitnessfunktion, die die Optimalität der Lösung bewertet. Die Entscheidungen des besten Lösungskandidaten werden nun mit extra Pheromonwerten belohnt, sodass die Ameisen in der darauffolgenden Generation mit höherer Wahrscheinlichkeit ebenso diese Entscheidungen treffen. Eine Neuerung von ACO im Vergleich zu Ant System sind lokale Pheromonupdates [3], die den Pheromonwert der gewählten Entscheidung abschwächen, um diversere Lösungskandidaten zu erzeugen und verfrühte Konvergenz einzudämmen.

Das Lösen von np-vollständigen Problemen ist eines der Hauptanwendungsgebiete der ACO-Metaheuristik [3].

1.2.2 Scattered Variable Neighbourhood Search

Die von Bass und Sevkli (2020) entwickelte Metaheuristik *scattered neighbourly approach* (Scattered Variable Neighbourhood Search, SVNS) vereint *Scatter Search* mit *Variable Neighbourhood Search* [2].

Die Metaheurstik Scatter Search wurde von Glover, Laguna und Martí (2000) als evolutionärer Algorithmus entwickelt [15]. Eine Neuheit von Scatter Search im Vergleich zu anderen evolutionären Algorithmen war die Möglichkeit, ohne zufällige Elemente durchgeführt werden zu können. Von dieser Eigenschaft macht SVNS jedoch keinen Gebrauch, es wird der randomisierte Ansatz gewählt.

Variable Neighbourhood Search wurde von Mladenović und Hansen (1997) als Metaheuristik für kombinatorische Optimierungsprobleme präsentiert.

1.2.3 Constraint Programming

Constraint Programming (CP) ist ein beliebtes Programmierparadigma für das Lösen von kombinatorischen Suchproblemen [16]. Durch eine Vielzahl an optimierten Lösungsverfahren können Variablenbelegungen gefunden werden, die die vorgeschriebenen Constraints (Einschränkungen) lösen. Eine Möglichkeit, besonders schwere Probleme zu lösen, besteht im Relaxieren von Einschränkungen, um den Lösungsraum zu erweitern oder die Formulierung des Problems zu vereinfachen. Hierfür werden meist Kostenfunktionen genutzt, die die erlaubterweise verletzten Constraints bestrafen, um dennoch eine möglichst optimale Lösung zu finden. In der CP-Implementation von Tamura [4] besteht der Lösungsraum zunächst auch nicht nur aus der Lösung, sondern wird iterativ eingeschränkt, bis das Rätsel gelöst ist (siehe Abschnitt 2.5).

2 Methoden

2.1 Generell

Der Constraint Programming-Ansatz über die Python-API von CPLEX [17] basiert auf dem Nurikabe-Solver in Copris von Tamura [4] und wurde, bis auf geringfügige Änderungen auf Grund der Unterschiede von CPLEX zu Copris, übernommen.

Sowohl der ACO- als auch der SVNS-basierte Solver sind in Java implementiert. Die Grundidee hinter beiden Implementationen ist eine stets hohe, kostengünstige Informationsverfügbarkeit bei zugleich stetigen, kleinen, lokalen Updates. Dies ist insbesondere für SVNS wichtig, da für die verschiedenen Nachbarschaften und die Fitnessfunktion regelmäßig komplexe Zusammenhänge der aktuellen Iterationsboardbelegung abgefragt werden.

- **Gitter**
  Die Zellen des Gitters werden durch die *Cells*-Klasse modelliert - diese Objekte enthalten neben Zeile und Spalte auch die Größe der Insel, also eine ganzzahlige Zahl, falls es eine Inselwurzel ist. Die Zellen werden nur einmal angelegt für den gesamten Verlauf der Lösungssuche und von den Iterationsboards geteilt.

- **Inseln**
  Inseln werden durch eine *Map* modelliert. Die Schlüssel der *Map* sind die Inselwurzel-Zellen, die Werte ein *Set* der jeweiligen Inselzellen.

- **Nachbarzellen**
  In einer weiteren *Map* werden für jede Insel die aktuellen horizontal oder vertikal angrenzenden Nachbarzellen gespeichert. Wird eine Insel also um eine Zelle erweitert oder verringert, ändert sich die Zahl der Nachbarzellen um maximal drei Zellen.

- **2x2-Blöcke**
  Die 2x2-Blöcke werden im SVNS auch als dynamisches *Set* gepflegt, da sie regelmäßig benötigt werden und ein andauerndes komplettes durchsuchen des Gitters so vermieden wird. Es wird lediglich die 2x2-Block-Wurzel-Zelle gespeichert, also die obere linke Zelle eines Blocks. Wird nun eine Seezelle zu einer Inselzelle, werden alle Zellen, die die Seezelle als Blockmitglied gehabt haben können, aus dem *Set* entfernt. Wird eine Inselzelle zur Seezelle, wird für jeden betroffenen 2x2-Block-Wurzelkandidaten geprüft, ob diese nun einen 2x2-Block ergeben.

- **Seefragmente**
  Als *Set* von *Sets* werden die Fragmente des Seegebiets gespeichert. Dies ist erneut besonders relevant für SVNS, da die Fragmente häufig benötigt werden. Wird eine Inselzelle zur Seezelle, entsteht entweder ein neues Fragment, ein bestehendes Fragment wird um die Zelle erweitert oder mehrere Fragmente werden durch die Zelle zu einem großen zusammenhängenden Gebiet aktualisiert. Wird eine Seezelle zur Inselzelle, wird das Gebiet entweder um die Zelle reduziert oder es führt zu einer Fragmentierung in mehrere Fragmente.

2.2 Testablauf

Vergleichbarkeit durch Zeitlimits

- **ACO**
  Der ACO-Algorithmus hat kein Abbruchkriterium. Daher wird ein Zeitlimit gesetzt, ab dem die Bearbeitung gestoppt wird. In ihrem Ansatz wählten Amos, Crossley und Lloyd eine Bearbeitungszeit von 60 Sekunden [12]. Diese wird in diesem Vergleich auf 90 Sekunden angehoben und gilt für alle Algorithmen.

- **CP**
  Da der CP-basierte Ansatz über eine deterministische Option verfügt [18], wird kein Zeitlimit angewendet und nur ein Durchlauf durchgeführt. Zur Vergleichbarkeit zählen Durchläufe, die länger als 90 Sekunden benötigten, dennoch als fehlgeschlagen.

- **SVNS**
  Der SVNS-Ansatz endet nach einer endlichen Zeit, da eine bestimmte Anzahl an Brettern für eine bestimmte Anzahl an Iterationen bearbeitet wird. Um vom festgelegten Zeitlimit Gebrauch machen zu können, wird ein Zyklus-Prinzip eingeführt: der erste Zyklus beginnt mit einer relativ niedrigen Anzahl an Brettern, um bei einfachen Instanzen keinen zu großen Overhead zu verursachen. Findet dieser keine Lösung und ist die Zeit noch nicht abgelaufen, wird der zweite Zyklus gestartet mit der doppelten Anzahl an Brettern. Der n-te Zyklus wird mit der n-fachen Anzahl an Brettern durchgeführt. Die Anzahl der Iterationen je Brett bleibt stets identisch.

Vorbearbeitung

In Nurikabe-Rätseln gibt es drei sehr offensichtliche Fälle, in denen Zellen Seezellen sein müssen. Inselwurzelzellen der Größe 1 sind von Seezellen umgeben. Zelle, die an zwei Inselwurzeln grenzen, sind Seezellen. Inselwurzelzellen der Größe 2, die in einer der Ecken des Gitters platziert sind, müssen eine Seezelle diagonal neben sich haben. Diese Konditionen werden für SVNS und ACO geprüft und so durchgesetzt, dass die Zellen stets Inselzellen bleiben. Die Durchführung dessen zählt zur Laufzeit des Algorithmus, wie auch bei CP.

Parallelisierung

- **ACO**
  Als naturnaher Algorithmus liegt eine Parallelisierung der *Ameisen* nahe. Die Generierung der Lösungskandidaten jeder Generation wird parallelisiert ausgeführt. Es stellt sich die Frage nach *race conditions* (Wettlaufsituationen). Da alle Ameisen auf eine einzige, globale Pheromonmatrix zugreifen und während der Iteration die Werte durch lokale Updates aktualisieren, muss sichergestellt werden, dass die Aktualisierungen nicht verworfen werden. Für die genaue Umsetzung sind mehrere Aspekte entscheidend:

  - **Zufall**
    Jede Ameise erweitert die Inseln nacheinander, wobei jede Ameise eine zufällige Reihenfolge der Inseln bearbeitet. Bei kleinen Instanzen oder sehr hohen Ameisenzahlen kann es also vorkommen, dass zwei Ameisen zeitgleich auf einer Insel vor derselben Wahl stehen und so gleichzeitig lokale Updates derselben Zelle durchführen möchten. Durch atomare Operationen wird sichergestellt, dass beide Updates ausgeführt werden.

  - **Synchronisierung**
    Ob die Ameisen untereinander synchronisiert werden müssen, ist der nächte Aspekt. Es stellt sich die Frage, ob Ameisen, die eine bestimmte Zelle ausgewählt haben, diese blockieren, bis der Zug vollendet ist. Oder ob gar die aktuell gewählte Insel blockiert werden sollten.

  - **Diversität der Lösungskandidaten**
    Die Grundidee der lokalen Pheromonupdates ist das Vermeiden verfrühter Konvergenzen und die Ermöglichung diverserer Lösungskandidaten. Dies wird durch die Atomizität sichergestellt in Verbindung mit der hohen Anazahl an Lösungskandidaten, die bei größeren Instanzen durch die Vielzahl an Generationen exploriert werden.

- **CP**
  Der CPLEX-Optimierer wird mit der Standardkonfiguration der parallelisierten deterministischen Variante ausgeführt.

- **SVNS**
  Der Scatter Search-Teil des Algorithmus beginnt mit der Generierung von einer substantiellen Zahl an Iterationsboards, von denen nur die besten übernommen werden. Diese Generierung wird parallelisiert ausgeführt. Die folgenden Iterationsschritte sind sequentieller Natur.

  Von Bass und Sevkli (2021) wurde zudem eine parallelisierte Version des sequentiellen Algorithmus vorgestellt [19]. Statt die Iterationsboards sequentiell zu bearbeiten, werden diese parallel bearbeitet. Zwischen den Iterationsboards herrscht keinerlei Informationsaustausch. Es wird zudem eine sinnvolle Interaktion, die im sequentiellen Ansatz vorhanden ist - nämlich die teilweise Ersetzung der Iterationsboards beim Fund neuer bester Lösungskandidaten, um diese erneut explorieren zu können - komplett verworfen, um race conditions und Synchronisierung

zu vermeiden. Daher wird dieser Ansatz bewusst nicht genutzt.

Testumgebung

Alle Tests werden unter Fedora Linux 40 (Workstation Edition) mit Kernel 6.10.11-200.fc40.x86_64, OpenJDK 22.0.2+9, Python 3.10.15 und CPLEX optimization Studio 22.1.1 durchgeführt. Hardwareseitig kommen ein AMD Ryzen 7 5700G und 32 GB DDR4-3000 SDRAM zum Einsatz. Alle Daten und das Betriebssystem befinden sich auf einer SSD. Die Netzwerkverbindung wird getrennt, Hintergrundprozesse geschlossen und ressourcenschonende Einstellungen deaktiviert.

Testinstanzen

Als Testinstanzen dienen die des Ehepaars Janko bereitgestellten Rätsel [20]. Nach der Größeneinteilung von Amos, Crossley und Lloyd [12] wurden zufällige Instanzen gewählt. Insgesamt 25 kleine Instanzen (0-99 Zellen), 17 mittlere (100-199 Zellen) und zehn große (200-299 Zellen).

Anzahl an Iteration

Da der CP-basierte Ansatz deterministisch ist, wird nur ein Durchlauf durchgeführt.

Sowohl der ACO- als auch der SVNS-basierte Ansatz sind nicht deterministisch. Für eine höhere Aussagekraft wurde daher jede Instanz jeweils 100 Mal mit dem gewählten Zeitlimit von 90 Sekunden gestartet.

2.3 Ant Colony Optimization

2.3.1 Ablauf

Eine Iteration (Generation) des ACO-Algorithmus (Alg. **??**) besteht aus einer Anzahl von Agenten (Ameisen), die unabhängig voneinander versuchen, das Rätsel zu lösen. In jeder Generation (Alg. 2) bekommt jede Ameise eine eigene Kopie der Rätselbretts im Urzustand. Das Rätselbrett wird im Urzustand bis auf die Inselwurzeln mit Seezellen belegt, wodurch die Seezellen insbesondere ein zusammenhängendes Fragment bilden. Danach bearbeiten die Ameisen die Inselwurzeln in einer zufälligen Reihenfolge und versuchen stets, die Inseln auf ihre vorgesehene Inselgröße wachsen zu lassen. Dabei dürfen die Erweiterungen der Inseln jedoch nicht die zusammenhängenden Seezellen fragmentieren.

Welche Zellen für die Expansion der Inseln gewählt werden, wird durch eine *globale Pheromonmatrix* geleitet, wobei Zellen mit einem höheren Wert wahrscheinlicher gewählt werden. Diese wird vor der ersten Iteration initialisiert. Nachdem eine Ameise eine Inselzelle gewählt hat, wird ein *lokales Pheromonupdate* durchgeführt, wobei der Pheromonwert der Zelle in der Pheromonmatrix abgeschwächt wird. Dadurch ist es tendenziell wahrscheinlicher, dass nachfolgende Ameisen eine andere Zelle zur Expansion wählen.

Der Algorithmus speichert stets die Zellenbelegung des besten Lösungskandidaten vorheriger Generation zusammen mit der Güte, die den Abstand zur Lösung bewertet. Die Bewertung geschieht durch eine Fitnessfunktion, die die Anzahl der Regelverstöße zählt. Als Regelverstöße können 2x2-Blöcke der Seefelder entstehen sowie fehlende Inselzellen, da manche Inseln nicht auf die vorgesehene Inselgröße expandiert werden konnten. Nach jeder Generation wird die Güte des besten Lösungskandidaten der aktuellen Generation mit der Güte des bisher besten Lösungskandidaten verglichen. Ist die aktuelle Güte besser, also näher an der Lösung, wird die Inselwahl des aktuellen Lösungskandidaten sowie die aktuelle Güte übernommen. Vor Beginn der neuen Generation wird ein globales Pheromonupdate durchgeführt, das die Inselzellen der gespeicherten besten Belegung mit extra Pheromonwerten belohnt, wobei die Stärke der Belohnung von der Güte des Lösungskandidaten abhängt. Als Mechanismus gegen Stagnation in lokalen Minima wird die Güte des aktuell besten Lösungskandidaten daraufhin abgeschwächt (*Best Value Evaporation*), um nachfolgend neue beste Belegungen zu ermöglichen, die eigentlich weiter von der Lösung entfernt sind.

2.3.2 Parameter

Variable	Domäne	Standardwert
numberOfAnts	$\mathbb{N}$	10
greediness	[0,1]	0.9
bestValueEvaporation	$[0, 1]$	0.001
localPheromoneUpdate	$[0, 1]$	0.1
evaporation	$[0, 1]$	0.2
finalTime	$\mathbb{N}$	90

- *numberOfAnts* beschreibt die Anzahl der Ameisen, die parallel je Generation einen Lösungskandidaten erstellen.

- *greediness* beschreibt die Wahrscheinlichkeit, mit der die Auswahl der Zellen einem gierigen Ansatz folgen, also die Zelle des höchsten Pheromonwerts wählen. Mit einer Wahrscheinlichkeit von (1-*greediness*) wird eine gewichtete Entscheidung getroffen.

- *bestValueEvaporation* beschreibt den prozentualen Anteil, um den die Güte des aktuell besten Lösungskandidaten nach jeder Generation abgeschwächt wird, um verfrühter Konvergenz einzudämmen.

- *localPheromoneUpdate* beschreibt den Faktor, um den sich der Pheromonwert an den Initialisierungswert annähert, also abgeschwächt wird.

- *evaporation* beschreibt den Faktor, um den sich beim globalen Pheromonupdate der Pheromonwert mit der Güte der aktuell besten Lösung belohnt wird.

- *finaleTime* beschreibt die Grenze, bis zu der der Algorithmus ausgeführt wird. Standardmäßig werden 90 Sekunden gewährt.

Algorithm 1 Ant Colony Optimization nach Amos, Crossley, Lloyd [12]

```
1:  function antColonyOptimization(timeLimit, numberOfAnts)
2:      read in puzzle grid
3:      set up pheromoneMatrix
4:      bestAnt ← null
5:      pheromoneToAdd ← 0.0
6:      while currentTime < timeLimit do
7:          for i ← 0, numberOfAnts do
8:              create copy of iterationBoard for ant i
9:              create solution candidate
10:         end for
11:         bestAntOfGeneration ← getBestPerformingAnt()
12:         bestAntFitness ← bestAntOfGeneration.getFitness()
13:         if bestAntFitness == 0 then
14:             return true
15:         end if
16:         if 1/bestAntFitness > pheromoneToAdd then
17:             bestAnt ← bestAntOfGeneration
18:             pheromoneToAdd ← 1/bestAntFitess
19:         end if
20:         globalPheromoneUpdate()
21:         bestValueEvaporation()
22:     end while
23:     return false                    ▷ Keine Lösung im Zeitlimit gefunden
24: end function
```

Algorithm 2 Creation of solution candidate

```
1:  gtdSeaCells ← {}                                          ▷ Speichert garantierte Seezellen
2:  for all islandRoot ∈ shuffledIslandRootsList do
3:      islandSize ← target size of ith island root
4:      for i ← 0, islandSize do
5:          candidateCells ← copy of island-adjacent cells
6:          for all candidateCell ∈ candidateCells do
7:              if candidateCell ∈ gtdSeaCells or would fragment sea if turned to island cell then
8:                  remove candidateCell from candidateCells
9:              else if candidateCell is forced sea cell or is adjacent to another island then
10:                 remove candidateCell from candidateCells
11:                 add candidateCell to gtdSeaCells
12:             end if
13:         end for
14:         if candidateCells is empty then
15:             break
16:         end if
17:         chosenCell ← select cell from candidateCells                              ▷ [1]
18:         localPheromoneUpdate(chosenCell)
19:         expand island of islandroot with chosenCell
20:         update island-adjacent Cells, remove chosenCell from seaFragment
21:     end for
22: end for
23: calculateFitness()          ▷ Anzahl der Regelverstöße: fehlende Inselzellen und 2x2-Blöcke
```

[1] Die Auswahl der Kandidaten folgt, basierend auf der Wahrscheinlichkeit des *greediness*-Parameters, entweder einem *greedy* (gierigen) Ansatz, bei dem die Zelle mit dem höchsten Pheromonwert gewählt wird, oder einem gewichteten Ansatz.

Algorithm 3 Selection among candidate cells

```
1: function CELLSELECTION(greediness, candidateCells)
2:     randomValue ← random value in [0, 1)
3:     if randomValue < greediness then
4:         select candidateCell with largest pheromone value
5:     else
6:         select candidateCell with a weighted search based on their pheromone values
7:     end if
8: end function
```

2.4 Scattered Variable Neighbourhood Search

2.4.1 Ablauf

Der *Scatter Search*-Teil des SVNS-Algorithmus (Alg. 4) nach Bass und Sevkli (2020) [2] beginnt mit der Generierung einer festgelegten Anzahl von Lösungskandidaten (Alg. 5). Hierfür wird für die Anzahl der zu generierenden Inselzellen jeweils eine zufällige Inselwurzel gewählt. Die Auswahl erfolgt gewichtet, wobei Inselwurzeln mit einer größeren Zielgröße häufiger selektiert werden - eine Inselwurzel mit doppelt so großer Zielgröße wird im Vergleich doppelt so häufig gewählt. Die gewählte Insel wird nun um eine Zelle expandiert, wobei die Zelle nicht zwei Inseln miteinander verbinden darf. Nach Verteilung der Inselzellen auf die Inselwurzeln wird eine Fitnessfunktion
(Alg. 6) auf den Lösungskandidaten angewandt, die die Güte feststellt, indem Regelverstöße gezählt und gewichtet werden. Als Regelverstöße sind Größenunterschiede der Inseln möglich, wobei Inseln sowohl zu klein als auch zu groß sein können. Zudem erlaubt der Algorithmus das Fragmentieren der zunächst zusammenhängenden Seezellen. Die Mehranzahl der Fragmente wird ebenso als Regelverstoß bestraft. Ebenso wird die Anzahl der 2x2-Blöcke aus Seezellen bestraft. Der Lösungszustand hat demnach einen Güte von Null.

Nachdem die Lösungskandidaten bewertet wurden, erfolgt die Selektion der besten Lösungskandidaten, wobei der Rest verworfen wird. Der beste Fitnesswert wird gespeichert und falls dieser Null ist, ist das Rätsel bereits gelöst. Jeder Lösungskandidat fungiert als *Iterationsboard*, auf dem weitere Lösungskandidaten generiert werden.

Nun beginnen der *Variable Neighbourhood Search*-Teil des Algorithmus. Eine festgelegte Anzahl an Iterationsdurchläufen wird sequentiell je Iterationsboard durchgeführt. Pro Iteration werden nacheinander die drei folgenden Nachbarschaften angewandt:

- **Surplus Island Swap** (Austausch übergroßer Insel) (Alg. 7)
  Das Ziel dieser Nachbarschaft ist, die Inseln ihren Zielgrößen anzunähern.
  Falls es zu große und somit auch zu kleine [21] Inseln gibt, wird versucht, eine zu große Insel um eine Zelle zu reduzieren, sowie eine zu kleine Insel um eine Zelle zu expandieren. Die Auswahl erfolgt jeweils nach dem Zufallsprinzip.

- **Unify Sea** (Seefragmente zusammenführen) (Alg. 8)
  Das Ziel dieser Nachbarschaft ist, kleinere Seefragmente schrumpfen zu lassen, bis am Ende ein zusammenhängendes Seegebiet entsteht.
  Falls die Seezellen fragmentiert sind, wird versucht, ein Fragment zu verkleinern. Hierfür wird eine angrenzende Insel um eine Zelle des Seefragments erweitert, insofern diese Zelle an eine weitere Insel grenzt. Daraufhin wird versucht, ein anderes Fragment zu vergrößern, indem eine angrenzende Inselzelle zu einer Seezelle umgewandelt wird, wobei die Insel dadurch nicht fragmentiert werden darf. Die Auswahl des Fragments, das verkleinert wird, geschieht gewichtet, wobei kleinere Fragmente wahrscheinlicher gewählt werden.

- **Break Up Blocks** (2x2-Blöcke aufbrechen) (Alg. 9)
  Das Ziel dieser Nachbarschaft ist die Annäherung an den Lösungszustand ohne 2x2-Blöcke. Falls es 2x2-Blöcke gibt, wird versucht, einen Block durch das Expandieren einer angrenzenden Insel aufzubrechen. Daraufhin wird eine Insel um eine Zelle reduziert, wobei durch das Umwandeln zur Seezelle kein neuer 2x2-Block entstehen darf. Die Auswahl der Blöcke sowie der Zellen geschieht nach dem Zufallsprinzip.

Falls keine der drei Nachbarschaften Änderungen herbeiführen konnte, wird eine weitere Nachbarschaft ausgeführt:

- **Regenerate Island** (Insel regenerieren) (Alg. 10
  Das Ziel dieser Nachbarschaft ist es, den aktuellen stagnierten Zustand aufzubrechen.
  Hierfür wird eine Insel zunächst auf die Inselwurzel reduziert und daraufhin auf die vorherige Größe expandiert. Die Zellen der Expansion werden zufällig aus den angrenzenden Zellen gewählt, wobei die gewählte Zelle nicht an eine andere Insel angrenzen darf. Die Wahl der Insel erfolgt ebenso nach dem Zufallsprinzip.

Nach Anwendung der Nachbarschaften wird der Fitnesswert des Lösungskandidaten ausgewertet. Falls dieser besser ist als der bisher beste Kandidat, wird der Wert gespeichert. Falls der Fitnesswert Null erreicht hat, ist das Rätsel gelöst, andernfalls wird mit einer festgelegten Wahrscheinlichkeit der Zustand des neuen besten Lösungskandidaten auf die noch nicht bearbeiteten Iterationsboards übertragen. Dies ermöglicht das erneute Explorieren des aktuellen Zustands.

2.4.2 Parameter

Variable	Domäne	Standardwert
boardsToGenerate	$\mathbb{N}$	$500 \cdot cycle$
boardsToKeep	$\mathbb{N}$	$100 \cdot cycle$
probReplace	$[0, 1]$	0.05
maxIterations	$\mathbb{N}$	300
sizeDifferenceWeight	$\mathbb{N}$	1
disjointWeight	$\mathbb{N}$	2
twoByTwoWeight	$\mathbb{N}$	3
finalTime	$\mathbb{N}$	90

- *boardsToGenerate* beschreibt die Anzahl der initiell zu generierenden Iterationsboards.
- *boardsToKeep* beschreibt die Anzahl der besten Iterationsboard, die nach Anwendung der Fitnessfunktion nicht verworfen werden.
- *probReplace* beschreibt die Wahrscheinlichkeit, mit der noch nicht bearbeitete Iterationsboards durch den Zustand des aktuellen Iterationsboards überschrieben werden, nachdem eine neue beste Lösung gefunden wurde.
- *maxIterations* beschreibt die Anzahl an Iterationen, die pro Iterationsboard durchgeführt werden. Eine Iteration beinhaltet das Anwenden der Nachbarschaften.
- *sizeDifferenceWeight* beschreibt, wie stark die Fitnessfunktion die Summe aus den Differenzen aller Inselgrößen von deren Zielgrößen bewertet.
- *disjointWeight* beschreibt, wie stark die Fitnessfunktion die Anzahl an Seefragmenten bestraft.
- *twoByTwoWeight* beschreibt, wie stark die Fitnessfunktion die Anzahl der 2x2-Blöcke bestraft.
- *finalTime* beschreibt das Zeitlimit, ab dem der Algorithmus beendet wird, falls noch keine Lösung gefunden wurde.

Algorithm 4 Scattered Variable Neighbourhood Search nach Bass und Sevkli (2020) [2]

```
1: function ScatteredVariableNeighbourhoodSearch(timeLimit, maxIterations, boardsToGenerate,
   boardsToKeep)
2:     read in puzzle grid
3:     generatedBoards ← generate boardsToGenerate many iteration boards
4:     generatedBoards ← generatedBoards sorted according to fitness function
5:     iterationBoards ← keep the best boardsToKeep many boards of generatedBoards
6:     currentIndex ← 0                ▷ speichert den aktuellen Index, um die Lösung abrufen zu können
7:     topFitness ← fitness Value of first iterationBoard
8:     if topFitness = 0 then          ▷ während der Scatter Search wurde ein Lösungskandidat gefunden
9:         return true
10:    end if
11:    for all iterationBoard in iterationBoards do
12:        for i ← 0, maxIterations do
13:            if i%20 = 0 and timeLimit ≤ currentTime then   ▷ Zeitlimit alle 20 Iterationen prüfen
14:                return false
15:            end if
16:            swapSuccess ← surplusIslandSwap()
17:            unifySuccess ← unifySea()
18:            breakUpSuccess ← breakUpBlocks()
19:            if not (swapSuccess or unifySuccess or breakUpSuccess) then
20:                regenerateIsland()
21:            end if
22:            currentFitness ← fitness value of current iterationBoard
23:            if currentFitness < bestFitness then
24:                bestFitness ← currentFitness
25:                if bestFitness = 0 then
26:                    return true
27:                else
28:                    replaceBoards()
29:                end if
30:            end if
31:            currentIndex + +
32:        end for
33:    end for
34:    return false                    ▷ Keine Lösung innerhalb des Zeitlimits
35: end function
```

Algorithm 5 Board Generation

```
1:  function generateBoard( )
2:      islandRoots ← a list of the island roots
3:      cellsToGenerate ← the total island cells of the solution minus the island roots
4:      islandsToIgnore ← empty set          ▷ speichert die definitiv nicht mehr erweiterbaren Inseln
5:      for i ← 0, cellsToGenerate do
6:          selectedIsland ← weighted selection of all non-ignored island roots          ▷ [1]
7:          attempt to grow selectedIsland by one cell
8:          if not successful then
9:              i ← i − 1                ▷ da keine Insel wuchs, wird die Laufviarable zurückgesetzt
10:             add selectedIsland to islandsToIgnore
11:         end if
12:     end for
13:     set sea fragments and 2x2 blocks
14: end function
```

[1] Die gewichtete Auswahl bevorzugt Inseln, die eine größere Zielgröße haben. Eine doppelt so große Zielgröße wird doppelt so wahrscheinlich gewählt.

Algorithm 6 Fitness Function

```
1: function calculateFitness(sizeDifferenceWeight, disjointWeight, twoByTwoWeight)
2:     sizeDifference ← difference between island cells of current board and solution target
3:     seaFragments ← amount of disjoint sea fragments minus one
4:     twoByTwos ← amount of 2x2 sea blocks
5:     return (sizeDifferenceWeight · sizeDifference
6:             + disjointWeight · seaFragments
7:             + twoByTwoWeight · twoByTwos)
8: end function
```

Algorithm 7 Surplus Island Swap

```
1: function surplusIslandSwap( ) ⟶ returns boolean: true if swapped, false if not
2:     smallslands ← shuffled list of all islands that are too small
3:     largeIslands ← shuffled list of all islands that are too large
4:     if smallIslands is empty then
5:         return false
6:     end if
7:     for all smallIsland in smallIslands do
8:         attempt to expand smallIsland by one cell
9:         if successfuly expanded then
10:            shrink first island in largeIslands by one cell                    ▷ [1]
11:            return true
12:        end if
13:    end for
14:    for all smallIsland in smallIslands do                                     ▷ [2]
15:        adjacentCells ← island-adjacent cells of smallIsland
16:        for all adjacentCell in adjacentCells do
17:            if adjacentCell has to be sea cell then
18:                continue
19:            end if
20:            adjacentIslands ← all islands that are adjacent to the cell
21:            if there are two adjacentIslands and the other island is in largeIslands then
22:                connectedCell ← the cell of largeIsland adjacent to adjacentCell
23:                if connectedCell can be removed from largeIsland without fracturing it then
24:                    remove connectedCell from largeIsland
25:                    expand smallIsland by adjacentCell
26:                    return true
27:                end if
28:            end if
29:        end for
30:    end for
31:    return false
32: end function
```

[1] Falls es eine zu kleine Insel gibt, gibt es auch eine zu große [21]. Eine zu große Insel kann immer um eine Zelle reduziert werden.

[2] Da keine zu kleine Insel expandiert werden konnte - alle angrenzenden Felder also auch an andere Inseln angrenzen -, wird versucht, zunächst eine dieser Inseln um eine Zelle zu reduzieren, die daraufhin die Expansion der zu kleinen Insel ermöglicht.

Algorithm 8 Unify Sea

```
1: function unifySea( ) ⟶ returns boolean: true if unified, false if not
2:     seaFragments ← set of sea fragments
3:     numOfFragments ← number of sea fragments
4:     if numOfFragments = 1 then
5:         return false
6:     end if
7:     for i ← 0, numOfFragments do
8:         seaFragment ← select new sea fragment weighted towards smaller seaFragments
9:         for all seaCell in seaFragment do
10:            if seaCell has to be sea cell then
11:                continue
12:            end if
13:            connectedIslands ← islands adjacent to seaCell
14:            if one connected island then
15:                add seaCell to connected island
16:                attempt to shrink a different sea fragment without fracturing it
17:                if successful then
18:                    return true
19:                else
20:                    revert adding seaCell to connected island
21:                end if
22:            else if two connected islands then
23:                from either island, attempt to remove the island cell adjacent to seaCell
24:                attempt to add seaCell to the other island
25:                if successful then
26:                    return true
27:                end if
28:            end if
29:        end for
30:    end for
31:    return false
32: end function
```

Algorithm 9 Break Up Blocks

```
1: function breakUpBlocks( ) ⟶ returns boolean: true if broken up a block, false if not
2:     twoByTwoRoots ← shuffled list of 2x2 roots
3:     if twoByTwoRoots is empty then
4:         return false
5:     end if
6:     visitedMembers ← empty set                ▷ speichert bereits visitierte 2x2 Mitglieder [1]
7:     for all twoByTwoRoot in twoByTwoRoots do
8:         twoByTwoMembers ← shuffled list of 2x2 members
9:         for all twoByTwoMember in twoByTwoMembers do
10:            if already visited or has to be sea cell then
11:                continue
12:            end if
13:            connectedIslands ← shuffled list of islands adjacent to twoByTwoMember
14:            if one connected island then
15:                add twoByTwoMember to connected island
16:                attempt to shrink any island without causing another 2x2 block
17:                if successful then
18:                    return true
19:                else
20:                    revert adding twoByTwoMember to connected island
21:                end if
22:            else if two connected islands then
23:                from either island, attempt to remove island cell connected to twoByTwoMember
   from island
24:                attempt to add twoByTwoMember to the other island
25:                if successful then
26:                    return true
27:                end if
28:            else
29:                add twoByTwoMember to visitedMembers
30:            end if
31:        end for
32:    end for
33:    return false
34: end function
```

[1] Ein 2x2-Mitglied ist die Wurzel selbst sowie die Zellen rechts und unterhalb der Wurzel.

Algorithm 10 Regenerate Island

```
1: function REGENERATEISLAND( )
2:     island ← select random island
3:     islandSize ← current amount of island cells
4:     shrink island by one cell (islandSize − 1) times
5:     grow island by one cell (islandSize − 1) times
6: end function
```

Notiz: Gegeben der gewählten Implementierung werden jedes Mal, wenn eine Insel um eine Zelle reduziert oder erweitert wird, alle angrenzenden Zellen, Seefragmente und 2x2-Blöcke aktualisiert. Dies führt bei normalgroßen Inseln bereits zu sehr vielen Operationen beim Regenerieren der Insel. Da diese Nachbarschaft jedoch nur sehr selten aufgerufen wird, hat die Optimierung dieser Variante keine Priorität.

Algorithm 11 Replace Boards

```
1: function REPLACEBOARDS(iterationBoards, currentIndex, boardsToKeep, probReplace )
2:     currentBoard ← the current board iterationsBoards[currentIndex]
3:     for i ← currentIndex, boardsToKeep do
4:         rand ← random value in [0, 1)
5:         if rand < probReplace then
6:             replace iterationBoards[i] with deepy copy of currentBoard          ▷ [1]
7:         end if
8:     end for
9: end function
```

[1] Eine deep copy (tiefe Kopie) der Inseln, der inselangrenzenden Zellen, der Seefragmente und der 2x2-Blöcke ist notwendig.

2.5 Constraint Programming

2.5.1 Ablauf

Der Constraint Programming-Ansatz nach Tamura [4] stellt durch Constraints sicher, dass fast alle Anforderungen an die Lösung erfüllt werden: Inseln haben die erforderliche Größe, Inseln sind nicht miteinander verbunden und es gibt keine 2x2-Seeblöcke. Es kann jedoch zu Seefragmenten kommen. Daher ist der Lösungsprozess iterativ: Für jeden Lösungskandidaten wird geprüft, ob die Seezellen fragmentiert sind. Falls nicht, ist das Rätsel gelöst. Falls doch, wird ein weiterer Constraint hinzufügt, die sicherstellt, dass im darauffolgenden Lösungskandidaten mindestens eine Zelle geändert ist.

Algorithm 12 Constraint Programming nach Tamura [4]

read in puzzle grid
numOfIslands ← the number of islands
cellDomains ← $\{0, \ldots, numOfIslands\}$ ▷ 0 steht für Seezellen
cellGrid ← set up constraint grid, each cell has to be in *cellDomains*
constrain each island root cell's domain to a unique positive integer in *cellDomains*
for all *cell* in *cellGrid* **do**
    if *cell* is root of 2x2 block **then**
        add 2x2 constraint ▷ [1]
    end if
    add constraint for connected islands ▷ [2]
    if *cell* is island root with target size 1 **then**
        set domain of all neighbours to 0
    end if
    if *cell* is island root **then**
        add constraints that if adjacent cell is connected to other island root, it has to be a sea cell
    end if
end for
if solution has at least two sea cells **then**
    add constraint that sea cells have to be connected to at least another sea cell
end if
construct trees for islands and add constraints for their sizes ▷ [3]
repeat
    solve model
    seaFragments ← number of sea fragments
    if *seaFragments* > 1 **then** ▷ keine valide Lösung
        restrict solution space by adding constraint that one cell has to change
    end if
until *seaFragments* = 1

[1] Die Wurzel ist die obere linke Zelle eines 2x2-Blocks, also jede Zelle außer die der letzten Zeile oder Spalte. Für jede dieser Zellen wird ein Constraint gesetzt, der die 2x2-Mitliederzellen auswertet und aufsummiert - da die Domäne von Inselzellen positiv ist, lässt sich so der Constraint leicht durchsetzen.
[2] Wenn eine Zelle positiv, also eine Inselzelle ist, müssen die Nachbarzellen entweder dieselbe Zahl, also dieselbe Insel, zugeteilt bekommen, oder die 0 als Seezelle.
[3] Um Inselgrößen sicherzustellen, wird durch Constraints ein Baum aus gerichteten Kanten aufgespannt. Die Inselwurzel ist die Wurzel des Baums und hat nur nach außen gerichtete Kanten für die angrenzenden Inselmitglieder. Jede Zelle hat nur eine eingehende Kante. Wenn eine Kante zwischen zwei Zellen existiert, müssen beide denselben Wert haben, also zur selben Insel gehören. Zuletzt werden durch Constraints die ausgehenden Kanten der Inselwurzel rekursiv addiert und sollen zusammen mit der Inselwurzel der Zielgröße der Insel entsprechen.

3 Ergebnisse

Es folgt die Auswertung der kleinen, mittleren und großen Testinstanzen anhand von kompakten Tabellen und mit Pandas [22], Seaborn [23] und matplotlib [24] erstellen Grafiken. Eine ausführlichere tabellarische Übersicht der Testdaten ist im Anhang (chapter 6) zu finden.

3.1 Kleine Testinstanzen (0-99 Zellen)

Tabelle 3.1: Ergebnisse kleiner Testinstanzen

	Instanzen			Laufzeit (ms)			
Solver	Gelöst	Gesamt	%	Ø	Median	Min	Max
ACO	2395	2500	95.8	1882	15	0.0	89 121
CP	25	25	100.0	185	58	16.26	2684.86
SVNS	2500	2500	100.0	121	8	3.0	11 667

In einigen Fällen war ACO so schnell, dass laut Javas *nanoTime* [25] keine Zeit verstrich. Alle kleinen Testinstanzen wurden von CP und SVNS gelöst. ACO hat in wenigen Fällen die Instanzen innerhalb des vorgegebenen Zeitrahmens nicht lösen können.

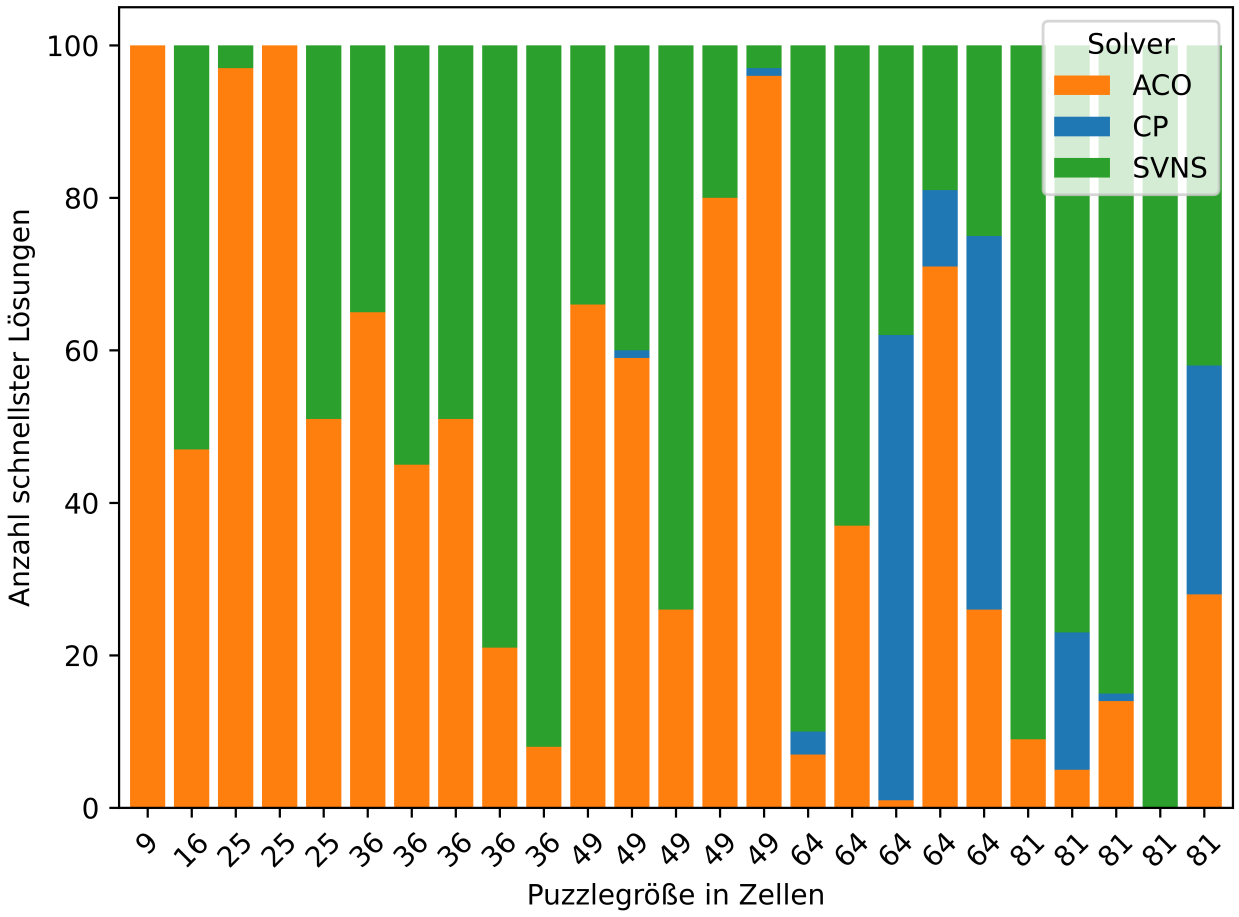

Abbildung 3.1: Anzahl der im Vergleich am schnellsten gelösten kleinen Testinstanzen je Solver

In den kleineren Instanzen haben ACO oder SVNS stets die schnellste Lösung gefunden. In größer werdenden Instanzen liefert der CP-Solver vermehrt schnellste Ergebnisse.

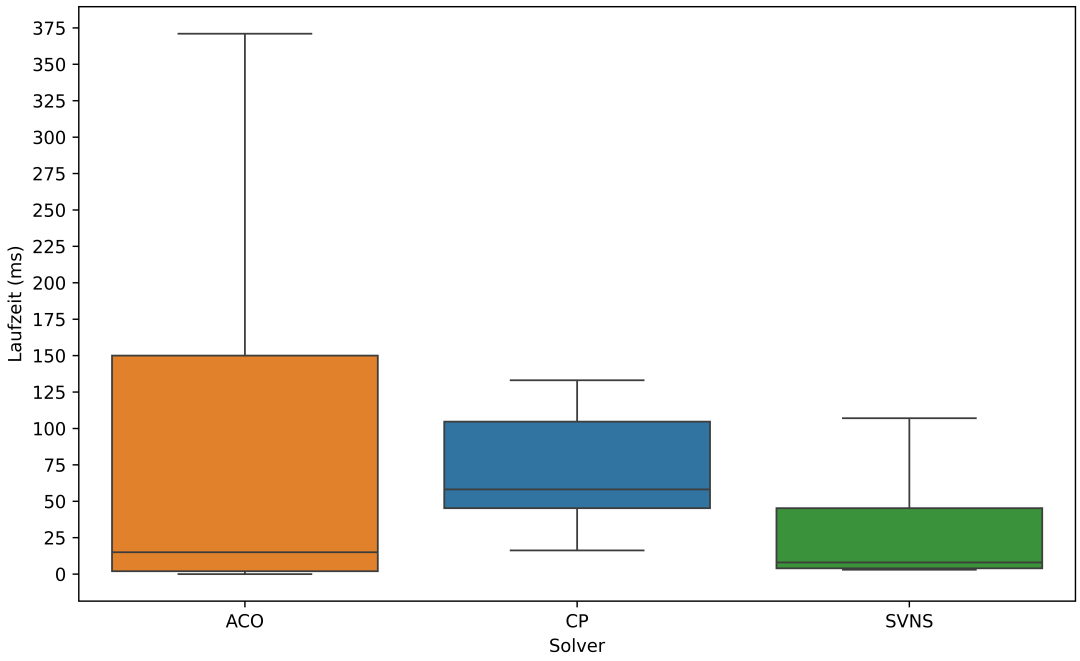

Abbildung 3.2: Boxplot der Laufzeiten aller gelösten kleinen Testinstanzen, ohne Ausreißer

ACO hat die größte Streuung beim Lösen der kleinen Testinstanzen. Der obere Whisker von SVNS liegt gleichauf mit dem oberen Quartil des CP-Solvers.

3.2 Mittlere Testinstanzen (100-199 Zellen)

Tabelle 3.2: Ergebnisse mittlerer Testinstanzen

	Instanzen			Laufzeit (ms)			
Solver	Gelöst	Gesamt	%	Ø	Median	Min	Max
ACO	1514	1700	89.1	7662	1109	9.0	89 518
CP	17	17	100.0	6962	199	86.93	64 305
SVNS	1253	1700	73.7	8589	770	5.0	89 006

Alle mittleren Testinstanzen wurden vom CP-Solver innerhalb des Zeitlimits gelöst. ACO scheiterte bei jeder zehnten Instanz am Zeitlimit, SVNS bei jeder vierten. Erneut gibt es eine starke Streuung der Laufzeiten bei ACO und SVNS, wobei auch die Laufzeit des CP-Solvers für manche Instanzen stark ansteigt.

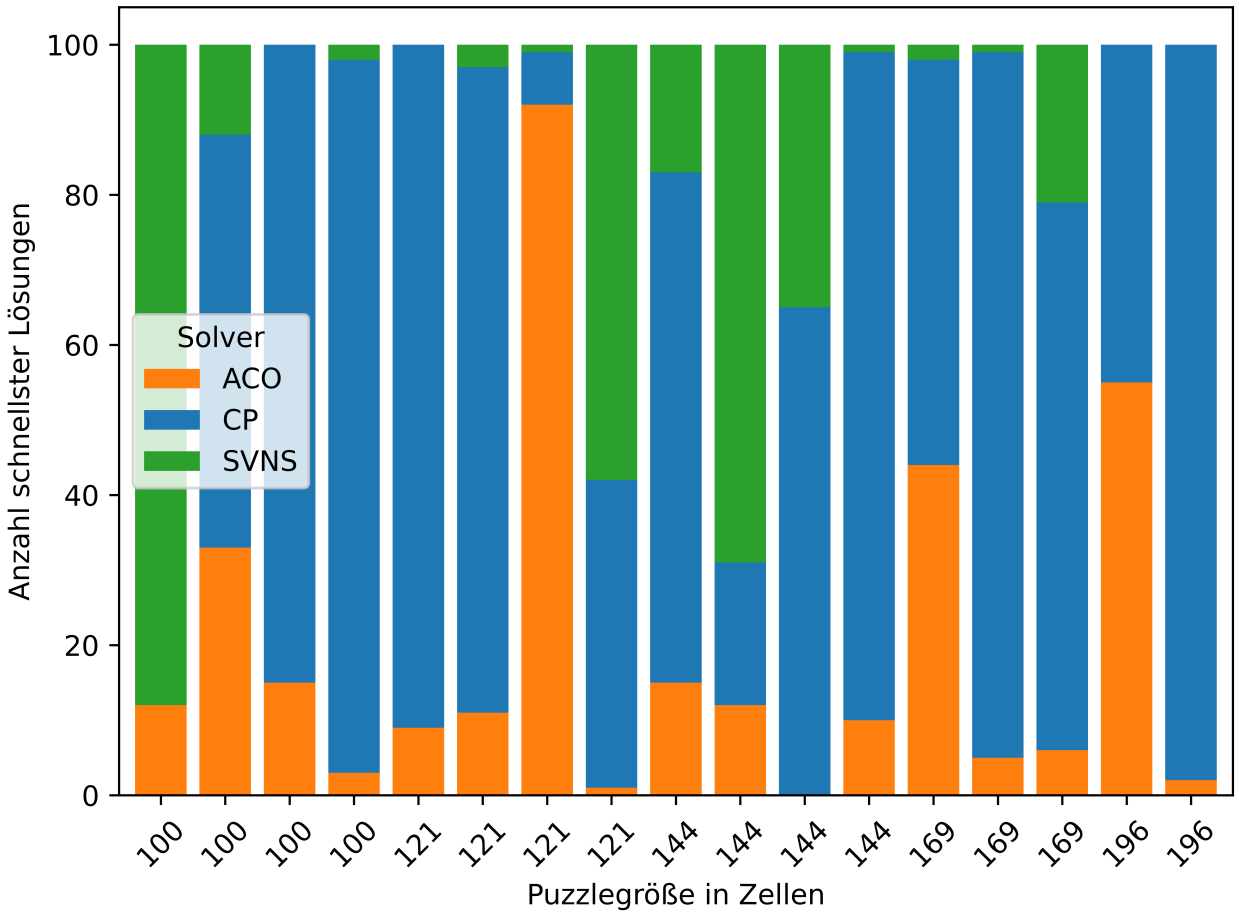

Abbildung 3.3: Anzahl der im Vergleich am schnellsten gelösten mittleren Testinstanzen je Solver

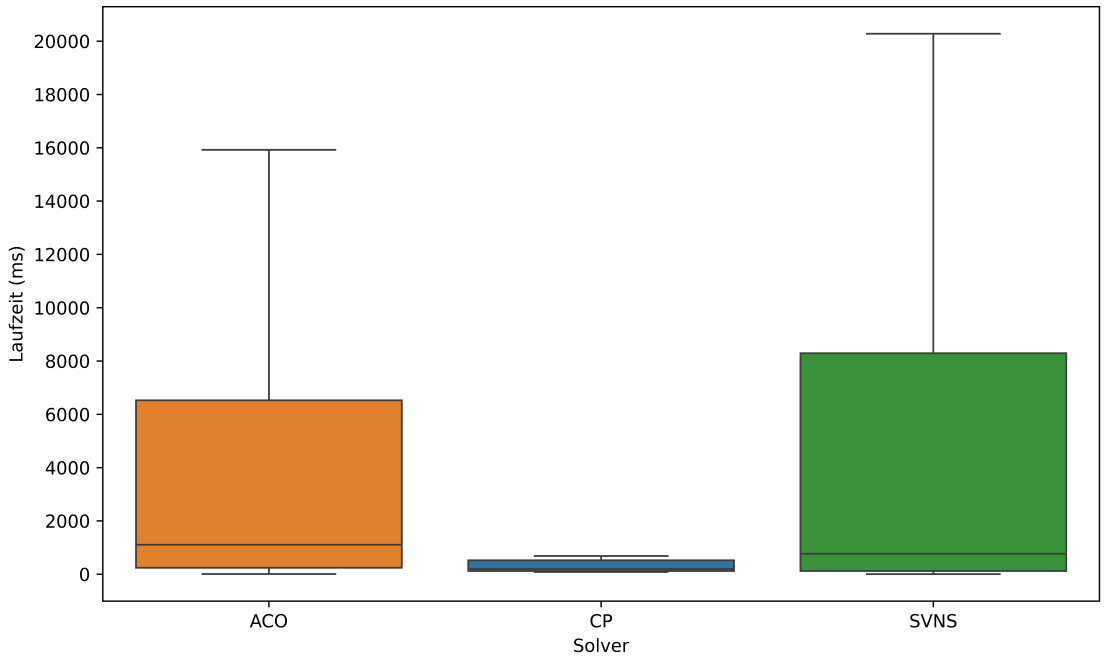

Abbildung 3.4: Boxplot der Laufzeiten aller gelösten kleinen Testinstanzen, ohne Ausreißer,

Bei den mittleren Testinstanzen hat bereits SVNS die höhere Streuung als ACO. Die Laufzeiten von CP streuen nur wenig.

3.3 Große Testinstanzen (200-299 Zellen)

Tabelle 3.3: Ergebnisse großer Testinstanzen

	Instanzen			Laufzeit (ms)			
Solver	Gelöst	Gesamt	%	Ø	Median	Min	Max
ACO	509	1000	50.9	10 989	3317	52.0	88 589
CP	8	10	80.0	855 548	4619	570.42	6 744 183
SVNS	1	1000	0.1	83 320	83 320	83 320	83 320

SVNS hat lediglich eine einzelne Instanz gelöst, ACO jede zweite. CP benötigt bei großen Instanzen teilweise deutlich länger, daher die Löserate von 80% - zwei Instanzen haben, teils deutlich, länger als das Limit von 90 Sekunden benötigt.

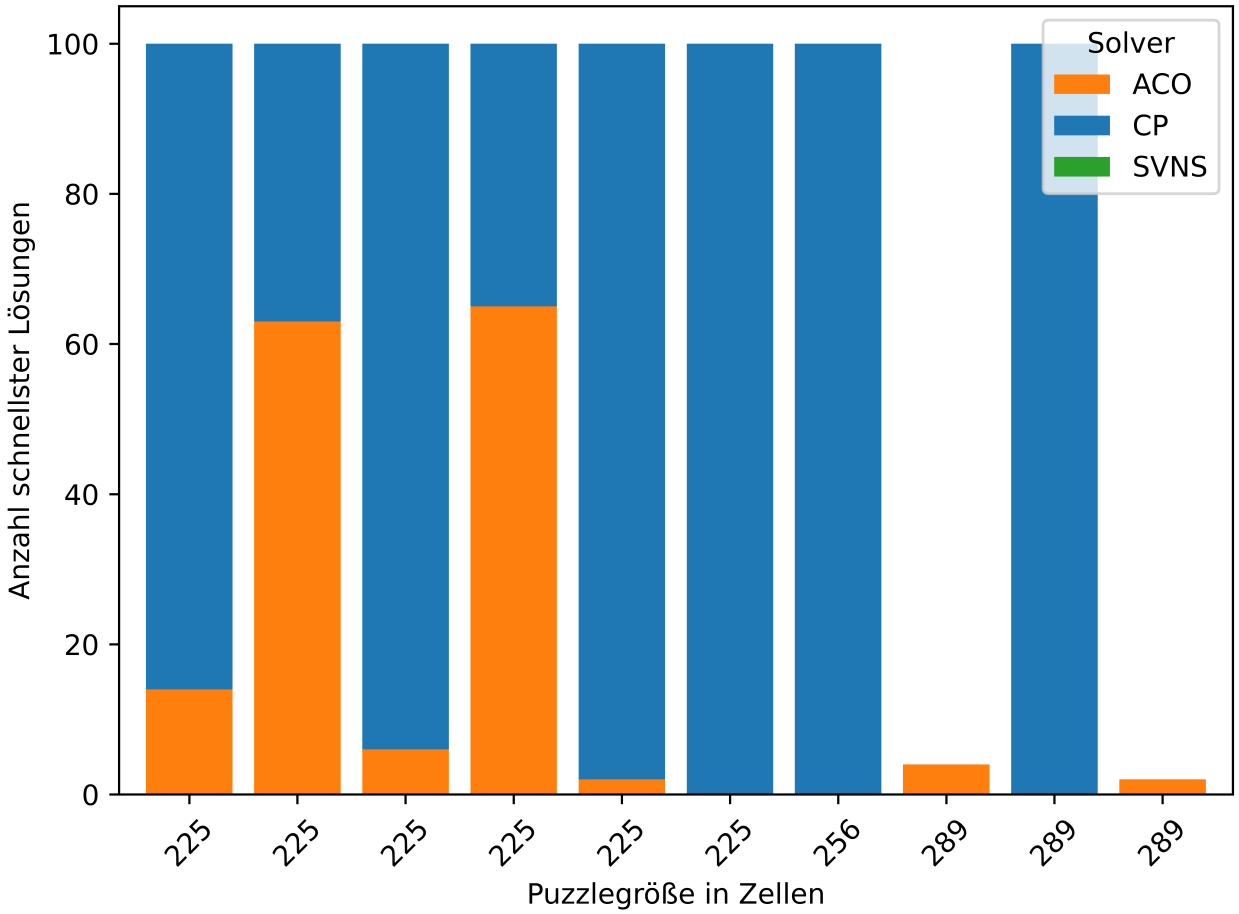

Abbildung 3.5: Anzahl der im Vergleich am schnellsten gelösten großen Testinstanzen je Solver

3.4 Gesamt

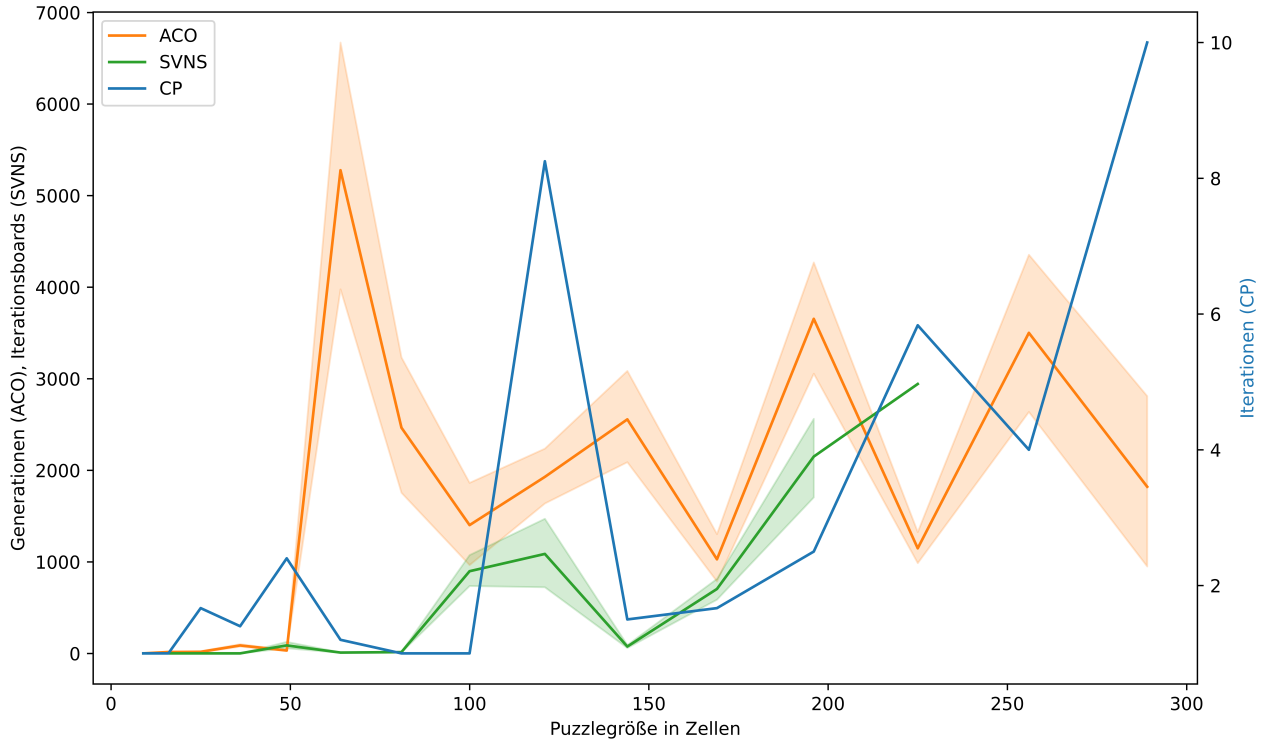

Abbildung 3.6: Verlauf der Anzahl an Generationen (ACO), Iterationsboards (SVNS), Iterationen (CP) aller gelösten Instanzen in Abhängigkeit der Puzzlegröße, mit getrennten Y-Achsen und von Pandas generierten Konfidenzintervallen

Es ist eine Abhängigkeit zwischen Puzzlegröße und benötigten Iterationsschritten bei CP erkennbar. Diese ist auch ersichtlich bei SVNS. Bei ACO ist keine lineare Tendenz zu erkennen.

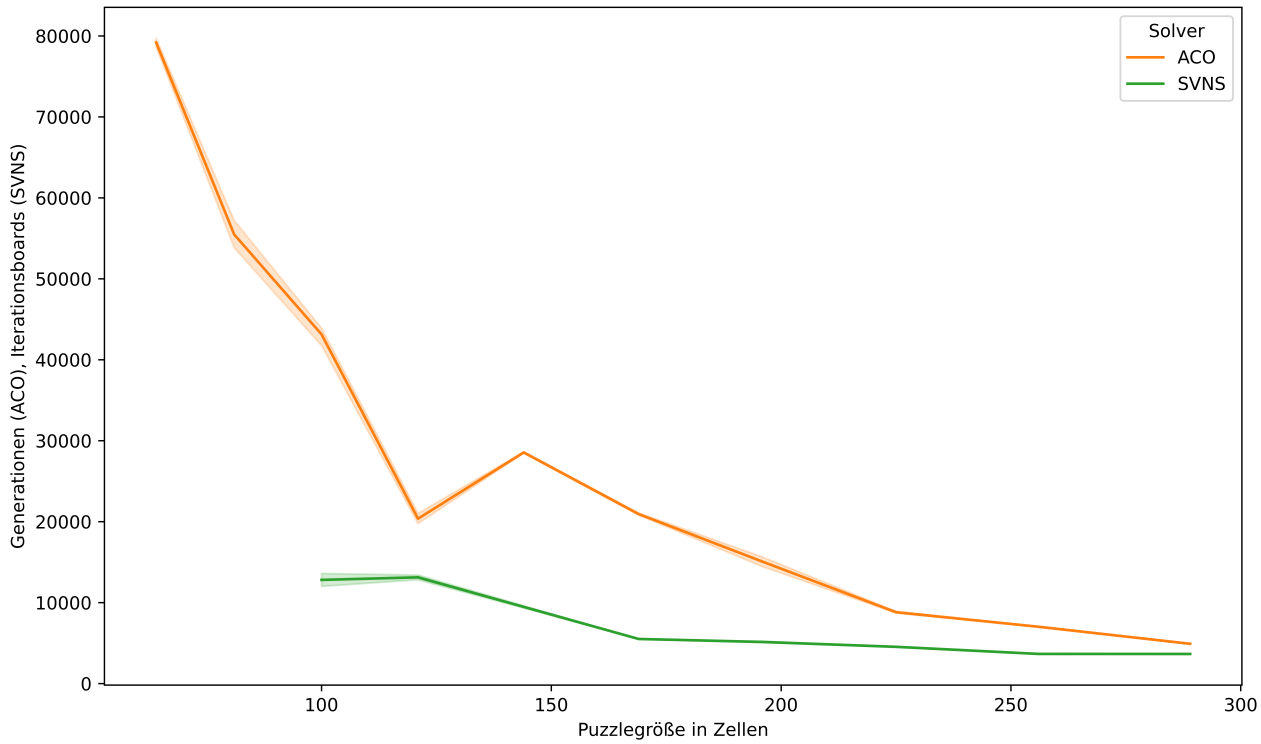

Abbildung 3.7: Verlauf der Anzahl an Generationen (ACO) und Iterationsboards (SVNS) aller ungelösten Instanzen in Abhängigkeit der Puzzlegröße

Die innerhalb der 90 Sekunden bearbeitete Anzahl an Generationen fällt bei ACO stark ab bei steigender Puzzlegröße. Bei SVNS sinkt die Anzahl an bearbeiteten Iterationsboards, die Tendenz ist jedoch deutlich schwächer.

4 Diskussion

4.1 ACO

Der ACO-basierte Ansatz zum Lösen von Nurikabe-Rätseln überzeugt sowohl bei kleinen als auch mittleren Testinstanzen. Auch bei großen findet er in annehmbarer Zeit in vielen Fällen eine Lösung. Gemessen an der einfachen Natur des Algorithmus sind dies gute Ergebnisse, insbesondere verglichen mit SVNS.

Auffallend stark ist der Rückgang an bearbeiteten Generationen bei steigenden Puzzlegrößen. So sind es beim Puzzle 48 mit einer Größe von 64 Zellen bei den Iterationen ohne Lösungsfindung innerhalb der 90 Sekunden meist noch über 80.000 Generation, sinkt es bei Puzzle 227 mit 121 Zellen bereits auf circa 30.000 Generationen. Umso mehr überrascht die durchschnittlich benötigte Zahl an Generationen in Abhängigkeit der Puzzlegröße. Sie skaliert nicht ersichtlich mit der Puzzlegröße. Für die Einführung eines möglichen Abbruchkriteriums scheint dies ein interessanter Aspekt zu sein. Stagnation in lokalen Optima ist ein Problem des Algorithmus, was an der starken Divergenz der Zeiten zu erkennen ist.

Die Ergebnisse von Amos, Crossley und Lloyd [12] konnten repliziert werden, wobei die Lösungsfindungsrate in der Implementierung und Testumgebung dieser Arbeit tendenziell besser ist. So ergaben sich in ihrer Publikation für die kleinen Testinstanzen eine Erfolgsrate von 94.2%, die mit 95.8% (Tabelle 3.1) übereinstimmt. Bei den mittleren Testinstanzen stehen 78.8% 89.1% (Tabelle 3.2) gegenüber. Dies kann allein durch die geringere Auswahl an Testinstanzen zu erklären sein. Zudem wird das höher gesetzte Zeitlimit teilweise ausgeschöpft, die langsamste Lösung wurde nach 89,5 Sekunden gefunden. Bei den großen Testinstanzen stehen 26,7% 50,9% (Tabelle 3.3) gegenüber. Weitere mögliche Gründe sind die Parallelisierung der Ameisen sowie das Festlegen der offensichtlichen Seefelder.

4.2 CP

Der CP-Ansatz nach Tamura ist insbesondere in mittleren und großen Testinstanzen schwierig zu schlagen. Bei kleinen Instanzen sind SVNS und ACO meist überlegen, da diese kein komplexes Konstrukt aus Constraints erzeugen müssen. Wie für CP in np-vollständigen Problemen gängig, wird der

Zeitaspekt bei größeren Instanzen kritisch. Nicht nur die Anzahl an Constraints erhöht sich, auch werden mehr Iterationen benötigt, in der jedes Mal alle Constraints erfüllt werden müssen. So steigt die Anzahl der Variablen von 155 bei Puzzles der Größe 25 auf 660 der Größe 100 zu 1515 der Größe 225. Die Constraints steigen von 276 bei 25 Zellen großen Rätseln zu 1201 bei 100 Zellen und schließlich 2776 bei 225 Zellen (Abschnitt 6.7). Dadurch kommt es zu deutlich mehr Branches.

4.3 SVNS

SVNS kann in den kleinen und mittleren Testinstanzen mit ACO mithalten und findet oftmals schneller zur Lösung. Der Median gelöster kleiner Instanzen ist lediglich die Hälfte des Medians von ACO. In den mittleren Testinstanzen nähert es sich auf 70% an. Bei den großen Instanzen werden die Schwächen von SVNS offensichtlich. Die implementierten Nachbarschaften reichen nicht aus, um aus lokalen Optima in annehmbarer Zeit auszubrechen. Die Anzahl bearbeiteter Iterationsboards sinkt nur langsam bei größer werdenden Instanzen, dennoch steigt die Anzahl benötigter Iterationsboards gleichzeit stark an. Das Zeitlimit ist demnach ein kritischer Faktor.

Da Bass und Sevkli die Bezeichnung nicht von Janko übernommen und nicht alle Parameter veröffentlicht haben [2], lassen sich die genauen Ergebnisse nur grob vergleichen. Die sieben 25-Zellen großen Instanzen brauchen im Schnitt 3,6 Sekunden - die drei 25-Zellen großen Instanzen in der Implementierung dieser Arbeit im Schnitt 3 Millisekunden. Den zwei 100 Zellen großen Instanzen mit im Schnitt 216,4 Sekunden stehen vier Instanzen mit 20 Millisekunden, 418 Millisekunden, 18,6 Sekunden und 11,1 Sekunden gegenüber. Es ist fraglich, ob allein die parallelisierte Ausführung der Scatter Search die Geschwindigkeitsunterschiede erklären kann.

4.4 Limitationen

Da SVNS und ACO nicht deterministisch sind und zu starken Variationen neigen, ist eine größere Anzahl an Testinstanzen sinnvoll für die Interpretation der Daten.

5 Fazit

Wie durch die np-Vollständigkeit bereits nachgewiesen, ist Nurikabe kein leicht zu lösendes Rätsel. Die drei untersuchten Algorithmen erwiesen sich als interessante Kandidaten mit eigenen Vor- und Nachteilen, keiner kann jedoch uneingeschränkt überzeugen.

Zu verbessern gilt bei SVNS vor allem die Performanz bei größeren Nurikabe-Rätseln. Die Iterationsschleife kann verbessert werden, da sie nur mit Anpassungen, wie die implementierte Zyklus-Prozedur, adaptiv auf die Schwere des Problems reagieren kann. Ein sinnvolles Abbruchkriterium existiert noch nicht. Der größte Fokus sollte jedoch auf die Verbesserung und Erweiterung der Nachbarschaften gelegt werden, wie auch von Bass und Sevkli [2] erwähnt.

ACO überzeugt durch den simplen Ansatz, ist jedoch anfällig für starke Varianz der Lösungsfindung. Eine Anpassung des Algorithmus durch weitere Heuristiken sollte zielführend sein.

CP liefert gute Ergebnisse. Als np-vollständiges Problem skaliert die Implementierung jedoch nur eingeschränkt mit großen Instanzen. Eine Kombination mit anderen Metaheuristiken könnte zielführend sein. So haben beispielsweise Solnon und Jussien (2013) einen Ansatz entwickelt, ACO und CP miteinander zu kombinieren [26]. Das Explorieren dieser Möglichkeit bietet sich als interessante Forschungsfrage an.

6 Anhang - Tabellen

6.1 Kleine Testinstanzen (0-99 Zellen)

Tabelle 6.1: Kleine Testinstanzen

Größe	Instanz	Anzahl Inseln	größte Insel	Inselanteil
3x3	101	2	5	67%
4x4	102	1	4	25%
5x5	103	4	4	64%
	104	5	4	44%
	105	4	4	40%
6x6	20	5	4	42%
	111	7	2	39%
	112	6	4	36%
	113	5	5	39%
	114	4	3	25%
7x7	26	8	5	45%
	34	6	5	39%
	121	9	2	37%
	122	6	3	37%
	123	6	4	49%
8x8	42	7	7	36%
	47	9	6	42%
	48	8	9	45%
	49	10	6	48%
	50	8	6	41%
9x9	141	7	7	40%
	143	8	4	40%
	144	10	4	37%
	145	9	5	30%
	83	10	5	37%
∑25 kleine Instanzen				

6.2 Ergebnisse kleiner Testinstanzen

Tabelle 6.2: Ergebnisse kleiner Testinstanzen - Teil 1

			Laufzeit (ms)			
Instanz	Solver	Lösungsrate	Ø	Median	Min	Max
101	ACO	1.0	0.0	0	0	0
	SVNS	1.0	3.75	4	3	5
	CP	1.0		16.3		
102	ACO	1.0	6.26	5	0	28
	SVNS	1.0	3.74	4	3	5
	CP	1.0		26.7		
103	ACO	1.0	0.6	0	0	7
	SVNS	1.0	3.87	4	3	5
	CP	1.0		26.4		
104	ACO	1.0	0.01	0	0	1
	SVNS	1.0	4.07	4	3	6
	CP	1.0		58.2		
105	ACO	1.0	22.0	4	0	363
	SVNS	1.0	4.76	4	3	11
	CP	1.0		57.8		
20	ACO	1.0	14.9	6	1	185
	SVNS	1.0	15.6	10	4	146
	CP	1.0		82.3		
111	ACO	1.0	25.6	7	0	201
	SVNS	1.0	5.21	5	3	15
	CP	1.0		34.2		
112	ACO	1.0	8.7	6	0	86
	SVNS	1.0	5.55	5	3	16
	CP	1.0		36.4		
113	ACO	1.0	89.2	20	0	518
	SVNS	1.0	6.39	6	4	16
	CP	1.0		74.2		

Tabelle 6.3: Ergebnisse kleiner Testinstanzen - Teil 2

			Laufzeit (ms)			
Instanz	Solver	Lösungsrate	Ø	Median	Min	Max
114	ACO	1.0	82.9	52	0	529
	SVNS	1.0	4.01	4	3	5
	CP	1.0		38.4		
26	ACO	1.0	8.72	5	0	36
	SVNS	1.0	10.0	8	4	33
	CP	1.0		77.8		
34	ACO	1.0	23.1	9	0	292
	SVNS	1.0	18.0	12	4	82
	CP	1.0		47.7		
121	ACO	1.0	31.0	15	0	192
	SVNS	1.0	5.23	5	4	11
	CP	1.0		45.3		
122	ACO	1.0	10.1	2	0	106
	SVNS	1.0	10.6	8	3	36
	CP	1.0		45.4		
123	ACO	1.0	25.8	6	0	305
	SVNS	1.0	1406	452	4	11 667
	CP	1.0		292		
42	ACO	1.0	1012	394	3	5464
	SVNS	1.0	39.8	28	4	261
	CP	1.0		131		
47	ACO	1.0	57.1	29	0	375
	SVNS	1.0	13.5	10	4	56
	CP	1.0		53.7		
48	ACO	0.8	33 513	22 667	1	89 121
	SVNS	1.0	257	162	9	1759
	CP	1.0		123		
49	ACO	1.0	56.0	11	0	1084
	SVNS	1.0	66.2	45	4	359
	CP	1.0		55.6		

Tabelle 6.4: Ergebnisse kleiner Testinstanzen - Teil 3

			Laufzeit (ms)			
Instanz	Solver	Lösungsrate	Ø	Median	Min	Max
50	ACO	1.0	786	152	1	6882
	SVNS	1.0	195	147	7	853
	CP	1.0		78.7		
141	ACO	0.63	6017	974	54	43 584
	SVNS	1.0	409	275	13	2548
	CP	1.0		2685		
143	ACO	0.86	1219	424	6	41 864
	SVNS	1.0	68.8	52	5	254
	CP	1.0		105		
144	ACO	1.0	367	206	6	2109
	SVNS	1.0	38.8	29	5	168
	CP	1.0		133		
145	ACO	0.7	3385	650	33	78 124
	SVNS	1.0	15.7	12	5	68
	CP	1.0		86.4		
83	ACO	0.96	8791	353	10	82 143
	SVNS	1.0	411	258	6	2237
	CP	1.0		218		

6.3 Mittlere Testinstanzen (100-199 Zellen)

Tabelle 6.5: Mittlere Testinstanzen

Größe	Instanz	Anzahl Inseln	größte Insel	Inselanteil
10x10	1	17	2	34%
	2	16	3	41%
	3	11	10	46%
	4	14	3	42%
11x11	227	16	11	46%
	518	10	9	45%
	721	7	27	55%
	150	16	7	40%
12x12	164	20	8	40%
	421	26	4	39%
	422	24	6	42%
	423	25	5	40%
13x13	292	29	5	42%
	293	27	5	43%
	297	32	5	40%
14x14	987	22	6	44%
	1087	32	5	39%
$\sum$17 Instanzen				

6.4 Ergebnisse mittlerer Testinstanzen

Tabelle 6.6: Ergebnisse mittlerer Testinstanzen - Teil 1

			Laufzeit (ms)			
Instanz	Solver	Lösungsrate	Ø	Median	Min	Max
1	ACO	1.0	66.6	54	10	237
	SVNS	1.0	20.1	14	5	89
	CP	1.0		86.9		
2	ACO	1.0	234	121	9	3112
	SVNS	1.0	418	302	11	2966
	CP	1.0		92.6		
3	ACO	0.99	2534	702	38	63 271
	SVNS	0.99	18 585	13 474	626	67 594
	CP	1.0		199		
4	ACO	0.85	10 391	3643	27	88 930
	SVNS	0.99	11 146	6194	22	77 117
	CP	1.0		121		
227	ACO	0.84	7587	3806	61	45 860
	SVNS	0.27	37 937	33 840	2339	86 909
	CP	1.0		344		
518	ACO	0.16	20 522	13 556	110	68 313
	SVNS	0.08	42 673	44 034	13 405	76 326
	CP	1.0		31 431		
721	ACO	0.94	10 751	4988	586	71 760
	SVNS	0.02	27 092	27 092	7696	46 487
	CP	1.0		64 306		
150	ACO	1.0	4674	2226	130	24 524
	SVNS	1.0	139	106	10	479
	CP	1.0		132		
164	ACO	1.0	3723	1727	25	28 402
	SVNS	1.0	2938	1774	44	18 300
	CP	1.0		525		
421	ACO	1.0	318	154	25	3366
	SVNS	1.0	93.8	65	9	395
	CP	1.0		112		

Tabelle 6.7: Ergebnisse mittlerer Testinstanzen - Teil 2

			Laufzeit (ms)			
Instanz	Solver	Lösungsrate	Ø	Median	Min	Max
422	ACO	0.86	23 499	15 100	138	89 518
	SVNS	1.0	314	196	22	1645
	CP	1.0		126		
423	ACO	1.0	7250	628	42	70 489
	SVNS	1.0	2827	1784	77	21 727
	CP	1.0		111		
292	ACO	1.0	954	437	38	6080
	SVNS	0.98	14 581	9104	72	76 939
	CP	1.0		349		
293	ACO	0.97	11 716	5235	192	71 356
	SVNS	0.95	30 284	25 341	182	88 919
	CP	1.0		275		
297	ACO	1.0	614	388	47	2260
	SVNS	1.0	418	288	35	2039
	CP	1.0		148		
987	ACO	0.86	19 493	10 435	155	81 225
	SVNS	0.0	-	-	-	-
	CP	1.0		19 307		
1087	ACO	0.67	27 962	20 724	204	84 877
	SVNS	0.25	55 045	53 943	5420	89 006
	CP	1.0		686		

6.5 Große Testinstanzen (200-299 Zellen)

Tabelle 6.8: Große Testinstanzen

Größe	Instanz	Anzahl Inseln	größte Insel	Inselanteil
15x15	386	34	7	44%
	387	39	5	42%
	388	36	7	41%
	389	35	7	43%
	390	33	7	43%
	264	27	9	42%
16x16	1065	31	8	43%
17x17	211	32	11	45%
	212	34	8	45%
	108	32	11	44%
$\sum$10 Instanzen				

6.6 Ergebnisse großer Testinstazen

Tabelle 6.9: Ergebnisse großer Testinstanzen (200-299 Zellen)

			Laufzeit (ms)			
Instanz	Solver	Lösungsrate	Ø	Median	Min	Max
386	ACO	1.0	10 020	7120	453	62 231
	SVNS	0.0	-	-	-	-
	CP	1.0		2559		
387	ACO	1.0	876	592	52	6355
	SVNS	0.0	-	-	-	-
	CP	1.0		808		
388	ACO	1.0	8074	5620	273	52 134
	SVNS	0.01	83 320	83 320	83 320	83 320
	CP	1.0		570		
389	ACO	1.0	1827	938	204	28 182
	SVNS	0.0	-	-	-	-
	CP	1.0		1292		
390	ACO	0.85	29 432	19 763	192	88 589
	SVNS	0.0	-	-	-	-
	CP	1.0		1424		
264	ACO	0.02	44 560	44 560	28 318	60 803
	SVNS	0.0	-	-	-	-
	CP	1.0		10 107		
1065	ACO	0.15	45 059	47 682	8464	84 357
	SVNS	0.0	-	-	-	-
	CP	1.0		6678		
211	ACO	0.04	36 623	31 511	7055	76 414
	SVNS	0.0	-	-	-	-
	CP	0.0		6 744 184		
212	ACO	0.01	18 333	18 333	18 333	18 333
	SVNS	0.0	-	-	-	-
	CP	1.0		14 081		
108	ACO	0.02	40 904	40 904	17 123	64 686
	SVNS	0.0	-	-	-	-
	CP	0.0		1 773 777		

6.7 Constraints und Branches

Tabelle 6.10: Constraints und Branches der ersten Iteration ausgewählter Instanzen

Instanz	Anzahl Zellen	Variablen	Constraints	Branches
104	25	155	276	675
105	25	155	276	4573
106	25	155	276	6274
2	100	660	1201	22 337
3	100	660	1201	179 757
4	100	660	1201	77 329
264	225	1515	2776	479 966
386	225	1515	2776	166 377
387	225	1515	2776	36 143

Literaturverzeichnis

[1] Elliot Doe, Mark HM Winands, Dennis JNJ Soemers, and Cameron Browne. Combining Monte-Carlo Tree Search with Proof-Number Search. In *2022 IEEE Conference on Games (CoG)*, pages 206–212. IEEE, 2022.

[2] Paul Bass and Aise Zulal Sevkli. A scattered neighborly approach to solving nurikabe. *J. Comput. Sci. Coll.*, 36(4):59–70, October 2020. ISSN 1937-4771.

[3] Marco Dorigo and Gianni Di Caro. Ant colony optimization: A new meta-heuristic. volume 2, page 1477 Vol. 2, 02 1999. ISBN 0-7803-5536-9. doi: 10.1109/CEC.1999.782657.

[4] Nurikabe Solver in Copris. https://cspsat.gitlab.io/copris-puzzles/nurikabe/index.html. Accessed: 20.09.2024.

[5] Nikoli. *Puzzle Communication Nikoli, 33rd issue*. Nikoli, 1991.

[6] Sudoku auf der Website von Nikoli. https://www.nikoli.co.jp/en/puzzles/sudoku/. Accessed: 20.09.2024.

[7] Nurikabe-Rätsel mit zwei möglichen Lösungen auf der Website des Ehepaars Janko. https://www.janko.at/Raetsel/Nurikabe/1086.a.htm/, . Aufgerufen: 20.09.2024.

[8] Brandon P. McPhail. The complexity of puzzles: Np-completeness results for nurikabe and minesweeper, 2003. Thesis for the Degree Bachelor of Arts.

[9] M. Dorigo and L.M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997. doi: 10.1109/4235.585892.

[10] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Positive feedback as a search strategy. *Tech rep., 91-016, Dip Elettronica, Politecnico di Milano, Italy*, 04 1999.

[11] Huw Lloyd and Martyn Amos. Solving sudoku with ant colony optimization. *IEEE Transactions on Games*, PP:1–1, 09 2019. doi: 10.1109/TG.2019.2942773.

[12] Martyn Amos, Matthew Crossley, and Huw Lloyd. Solving nurikabe with ant colony optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '19, page 129–130, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367486. doi: 10.1145/3319619.3338470. URL https://doi.org/10.1145/3319619.3338470.

[13] S Goss, Serge Aron, Jean-Louis Deneubourg, and Jacques Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76:579, 1 1989.

[14] J. L. Deneubourg R. Beckers and S. Goss. Trails and u-turns in the selection of the shortest path by the ant lasius niger. *Theoretical Biology*, 159:397, 1992.

[15] Fred Glover, Manuel Laguna, and Rafael Marti. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 29, 01 2000.

[16] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4. URL https://www.sciencedirect.com/science/bookseries/15746526/2.

[17] IBM Decision Optimization CPLEX Modeling for Python. https://ibmdecisionoptimization.github.io/docplex-doc/, . Accessed: 20.09.2024.

[18] parallel mode switch - IBM CPLEX Documentation. https://www.ibm.com/docs/en/icos/22.1.1?topic=parameters-parallel-mode-switch, . Accessed: 20.09.2024.

[19] Paul Bass and Aise Zulal Sevkli. Sequential and parallel scattered variable neighborhood search for solving nurikabe. In Nenad Mladenovic, Andrei Sleptchenko, Angelo Sifaleras, and Mohammed Omar, editors, *Variable Neighborhood Search*, pages 99–110, Cham, 2021. Springer International Publishing. ISBN 978-3-030-69625-2.

[20] Nurikabe auf der Website des Ehepaars Janko. https://www.janko.at/Raetsel/Nurikabe, . Accessed: 20.09.2024.

[21] Benoît, Rittaud and Heeffer, Albrecht. The pigeonhole principle, two centuries before Dirichlet. *MATHEMATICAL INTELLIGENCER*, 36(2):27–29, 2014. ISSN 0343-6993. URL http://doi.org/10.1007/s00283-013-9389-1.

[22] pandas - python data analysis library.

[23] seaborn: statistical data visualization - seaborn 0.13.2 documentation. https://seaborn.pydata.org. Accessed: 03.10.2024.

[24] Matplotlib . visualization with python. https://matplotlib.org. Accessed: 03.10.2024.

[25] System (java se 22 amp; jdk 22). https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/System.html#nanoTime(). Accessed: 30.09.2024.

[26] C. Solnon. *Ant Colony Optimization and Constraint Programming*. ISTE. Wiley, 2013. ISBN 9781118618899. URL https://books.google.de/books?id=CCZcyPxJebAC.