Entwicklung und Optimierung eines Algorithmus zur Lösung von Ubongo 3-D-Aufgaben

Maturaarbeit

Gian Federer

Betreut durch
Thomas Jampen

Richigen, 14. Oktober 2024

Gymnasium Kirchenfeld
Abteilung MN
Klasse M25c

1 Inhalt

2 Einleitung

2.1 Themenfindung

Als ich noch ein Kind war, schenkten mir meine Eltern das Puzzlespiel Ubongo 3-D. Ich mochte dieses Spiel sehr gerne und fragte mich schon damals, wie die Aufgaben erstellt wurden. Während meiner Gymnasialzeit begann ich mich für Informatik zu interessieren. Als es darum ging, ein Thema für meine Maturaarbeit zu finden, kam mir die Idee, diese beiden Dinge zu kombinieren und ein Programm zu schreiben, das die Aufgaben von Ubongo 3-D lösen kann.

2.2 Problemstellung

Ubongo 3-D[1] ist ein Puzzlespiel, bei dem die Spielende versuchen müssen, mehrere vorgegebene Legeteile in einem vorgegebenen Volumen unterzubringen. Ubongo 3-D hat Aufgaben mit drei und solche mit vier Teilen, es können aber auch Aufgaben mit mehr Teilen erstellt und gelöst werden. Es gibt 16 verschiedene Legeteile, die alle aus drei bis

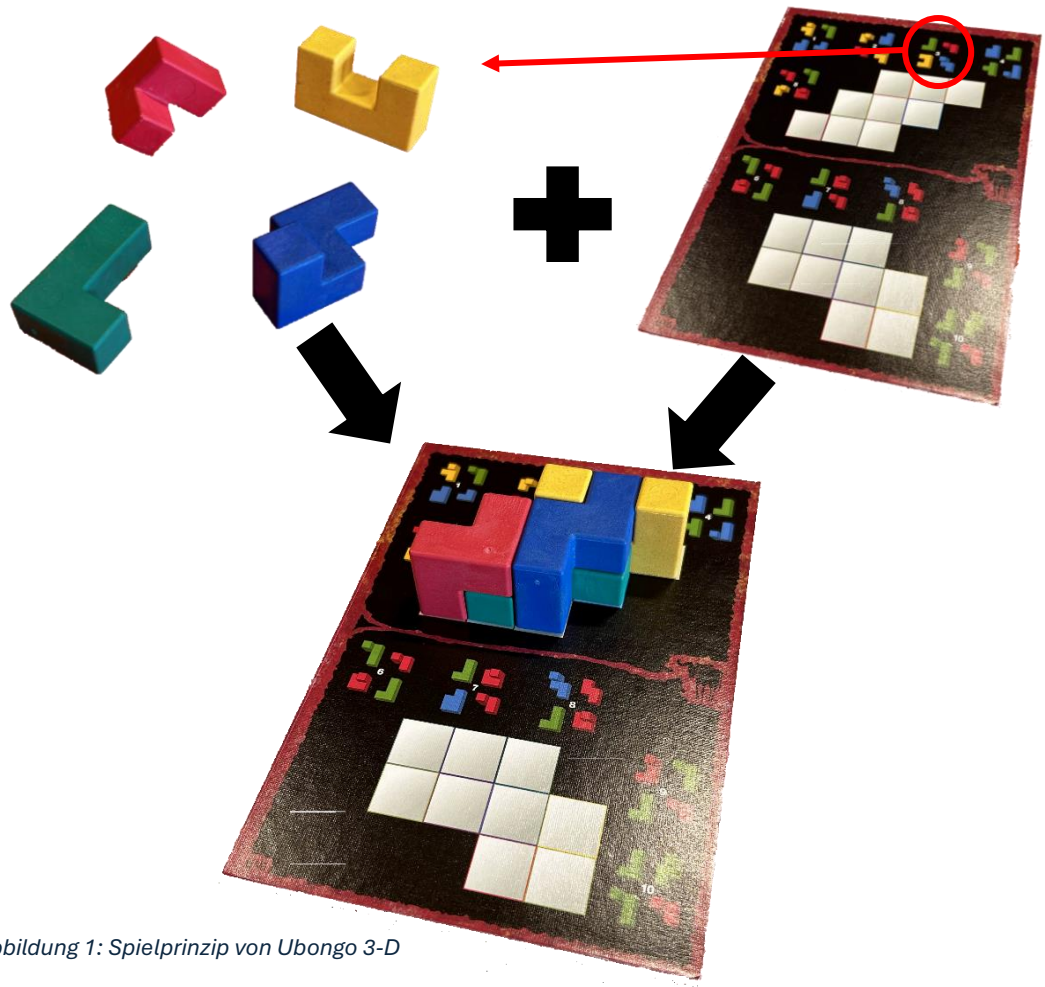


Abbildung 1: Spielprinzip von Ubongo 3-D

Die weisse Fläche, die man oben rechts auf den Karten sehen kann, wird in dieser Arbeit Legefläche oder Grundfläche genannt. Die farbigen Teile werden, wie auch im echten Spiel, Legeteile genannt. Oben links: die vier Legeteile für diese Aufgabe; oben rechts: Karte mit Legefläche und vorgegebenen Legeteilen; unten: gelöste Aufgabe

[1] https://www.kosmos.de/de/ubongo-3-d-master_1683177_4002051683177

fünf Würfeln bestehen. Wichtig anzumerken ist, dass jedes Legeteil in einem 2x2x3 Quader Platz findet.

Jede Spielkarte gibt das Volumen vor, in dem die drei bzw. vier Legeteile untergebracht werden müssen. Die Höhe des Volumens beträgt immer zwei Würfel, die Grundfläche variiert je nach Spielkarte. Das zu füllende Volumen entspricht der Summe aller Würfel der Legeteile. Solange Würfel aus dem Volumen herausragen (sei es seitlich oder in die dritte Ebene), gilt die Aufgabe als nicht gelöst.

In dieser Arbeit wird in einem ersten Schritt ein Programm entwickelt, das die Ubongo 3-D Aufgaben lösen kann. In einem zweiten Schritt wird das Programm so optimiert, dass es die Aufgaben in möglichst kurzer Zeit lösen kann. Die wichtigste Anforderung an den Algorithmus ist die Zuverlässigkeit, d.h. für jede lösbare Aufgabe muss der Algorithmus mindestens eine Lösung finden, erst an zweiter Stelle kommt die Effizienz.

2.3 Lösungsansatz

Der in dieser Arbeit verwendete Lösungsansatz ist dem Spiel mit den echten Legeteilen und Karten sehr ähnlich: Zuerst wird ein erstes Legeteil in das von der Aufgabe vorgegebene Volumen eingesetzt. Passt dieses mit der aktuellen Ausrichtung nicht in das Volumen, wird es gedreht und ein neuer Versuch gestartet, das Teil zu platzieren.

Um diese Idee umzusetzen, müssen verschiedene Elemente programmiert werden. Die 16 Legeteile sowie die verschiedenen Volumina, die sich aus der Grundfläche ergeben, müssen definiert werden. Zusätzlich muss ein Programmelement in der Lage sein, die Legeteile zu drehen, ein anderes, sie zu verschieben, ein drittes, sie zu platzieren und ein letztes, den aktuellen Stand der Aufgabe zu überprüfen.

Zu Beginn wird ein Basisalgorithmus entwickelt, der in der Lage ist, die Aufgaben zu lösen. Die Effizienz des Algorithmus wird in diesem ersten Schritt noch nicht berücksichtigt, es ist jedoch wichtig, dass alle Aufgaben korrekt gelöst werden. Der Basisalgorithmus bildet den Ausgangspunkt für die nachfolgenden Optimierungen, die das Ziel haben, die Effizienz des Algorithmus zu steigern. Um die Effizienzsteigerung nachweisen zu können, muss sie messbar sein. Um den Einfluss von Hintergrundprozessen, Temperaturunterschieden und anderen Faktoren auf die Durchlaufgeschwindigkeit zu minimieren, wurden zwei Datensätze erstellt, die pro Zeitmessung 100-mal gelöst werden.

2.4 Wahl der Programmiersprache

Als Programmiersprache für den Algorithmus wurde Python gewählt. Python ist eine weit verbreitete Programmiersprache mit vielen leistungsfähigen Bibliotheken. Die Popularität von Python zeigt sich auch in der grossen Anzahl von Online-Dokumentationen und Foren, die konsultiert werden können. Zudem ist Python-Code leicht lesbar und wird am Gymnasium Kirchenfeld als Programmiersprache im Informatikunterricht eingesetzt. All diese Gründe haben mich dazu bewogen, Python zu wählen, obwohl es schnellere

Programmiersprachen gibt. Ziel des zweiten Projektteils ist es, eine Effizienzsteigerung durch einen verbesserten Algorithmus zu erreichen und nicht durch die Leistungsfähigkeit einer Programmiersprache oder der Hardware zu einer schnelleren Lösung zu gelangen. Daher spielt die Effizienz der Programmiersprache keine grosse Rolle. Als Programmierumgebung wurde Visual Studio Code (VSC) (Visualstudio, 2024) verwendet, da VSC eine gute Schnittstelle zu GitLab (GitLab, 2024) bietet.

2.5 Begriffserklärungen

- Funktion/Unterprogramm:

  «Bei der Definition eines Unterprogramms werden Anweisungen zusammengefasst und mit einem Namen versehen. So können mehrere Anweisungen zu einem späteren Zeitpunkt mit einem einzigen Befehl aufgerufen werden.» (informatik.mygymer.ch, n.d.)

- Liste:

  «Eine Liste ist ein «zusammengesetzter» Datentyp und besteht aus mehreren Elementen. In einer Liste können Zahlen, Strings (also Zeichenfolgen) oder Boole'sche Werte (True resp. False) gespeichert werden.» (informatik.mygymer.ch, n.d.)

- Mehrdimensionales Array:

  Auch Matrix genannt, ist eine Liste, in der sich andere Listen befinden. Dies ermöglicht das Speichern von Daten in mehreren Dimensionen. Zum Beispiel einem 2-dimensionalen Bild oder 3-dimensionalen Objekt.

- Klasse:

  «Eine Klasse dient als Vorlage für Objekte. Eine Klasse ist etwas Abstraktes, sie ist eine Art Gerüst für Objekte. Als Beispiel kann man sich die Klasse Auto vorstellen. Die Klasse Auto definiert, was ein Auto ausmacht und wie sich ein Auto verhalten kann.» (informatik.mygymer.ch, n.d.)

- Objekt:

  «Ein Objekt – auch Instanz einer Klasse genannt – stellt ein konkretes Objekt einer bestimmten Klasse dar.» (informatik.mygymer.ch, n.d.)

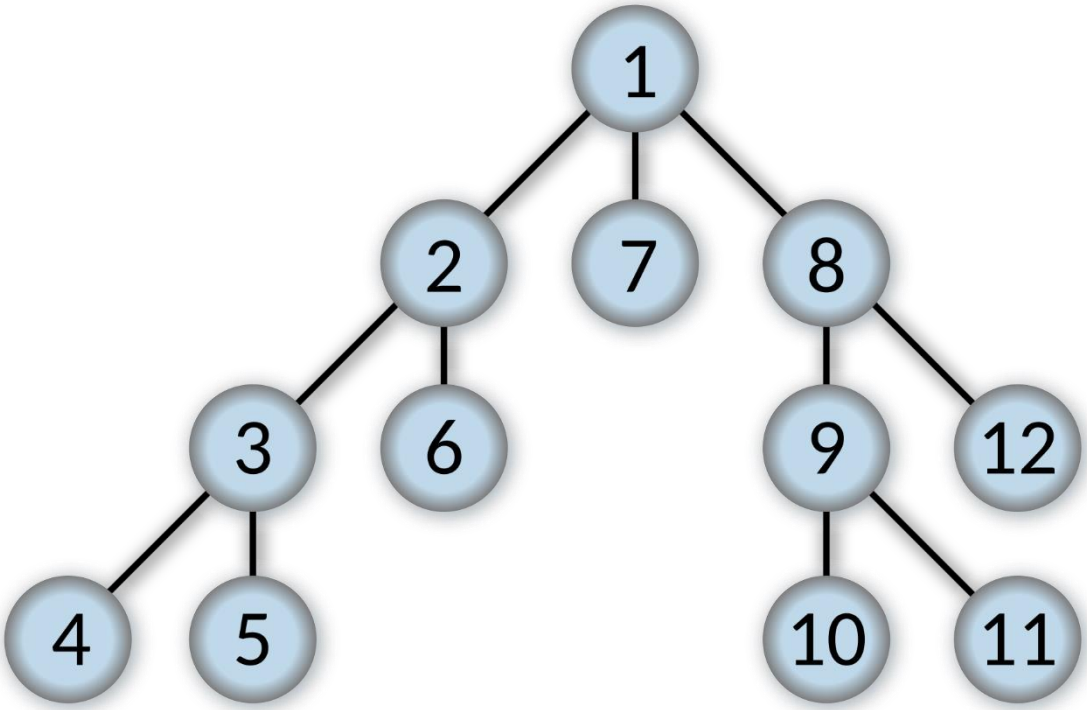
- Zeitkomplexität:

  «Zeitkomplexität [...] beschreibt die Änderung der Ausführungszeit eines Algorithmus in Abhängigkeit von der Änderung der Größe der Eingabedaten. Oder anders gesagt: "Um wie viel verlangsamt sich ein Algorithmus, wenn die Menge der Eingabedaten größer wird?"» (Woltmann, 2020)

3 Theoretische Grundlagen

3.1 Backtracking

Backtracking ist ein Verfahren, das in vielen Lösungsalgorithmen eingesetzt wird. Backtracking basiert auf der bekannten Brute-Force-Methode. Bei der Brute-Force-Methode werden mit hohem Rechenaufwand und ohne Strategie alle möglichen Varianten ausprobiert, um ein Problem zu lösen. Deshalb ist es naheliegend, dass der



Brute-Force-Ansatz ineffizient ist. Beim Backtracking wird ein Lösungsversuch abgebrochen, sobald klar ist, dass dieser Ansatz nicht zu einer Lösung führen kann. In diesem Fall springt der Algorithmus an die letzte Stelle zurück, an der eine Lösung noch möglich war (Abbildung *2*). Wird bei Ubongo 3-D das erste Legeteil neben die 3-dimensionale Form platziert, springt der Backtracking-Algorithmus direkt zur nächsten Variante des ersten Legeteiles und probiert nicht noch alle möglichen Positionen des zweiten und dritten Legeteiles aus, wie das bei einem Brute-Force-Algorithmus der Fall wäre.

Abbildung 2: Schematische Darstellung des Backtracking. (Wikipedia, n.d.)Bei Schritt vier wird klar, dass dieser Weg nicht zur Lösung führt. Der Algorithmus kehrt zu Schritt drei zurück und testet eine weitere Möglichkeit. Auch diese führt nicht zur Lösung, weitere Möglichkeiten gibt es nicht mehr. Daher geht der Algorithmus noch eine Stufe zurück und testet dort eine weitere Möglichkeit usw.

3.2 Hashing

Die Hashfunktion wandelt einen Wert oder Zeichenkette (auch Sting genannt) reproduzierbar in eine Zeichenkette fester Länge um, in der Regel in einen Hexadezimalwert (Abbildung 3). Da die Eingabemenge unendlich und die Ausgabemenge endlich ist, kann es zu Kollisionen kommen. Das bedeutet, dass eine Hashfunktion nicht

injektiv ist und daher nicht rückgängig gemacht werden kann. Dies kann je nach Anwendung ein Problem oder ein Vorteil sein. In der Kryptologie ist dies von Vorteil, da aus dem Hashwert nicht auf das Passwort geschlossen werden kann. Beim Vergleich zweier Elemente können Hash-Kollisionen jedoch ein Hindernis darstellen. Wenn die Hashwerte zweier Elemente gleich sind, gibt es wegen möglicher Hashkollisionen keine Sicherheit, dass die Elemente selbst auch gleich sind. Für sha256, die in dieser Arbeit verwendete Hashfunktion, gibt es 2^256 mögliche Hashwerte. Diese Zahl ist so gross, dass es sehr unwahrscheinlich ist, eine Kollision zu finden; in der Tat wurde bis heute weltweit noch keine einzige gefunden.

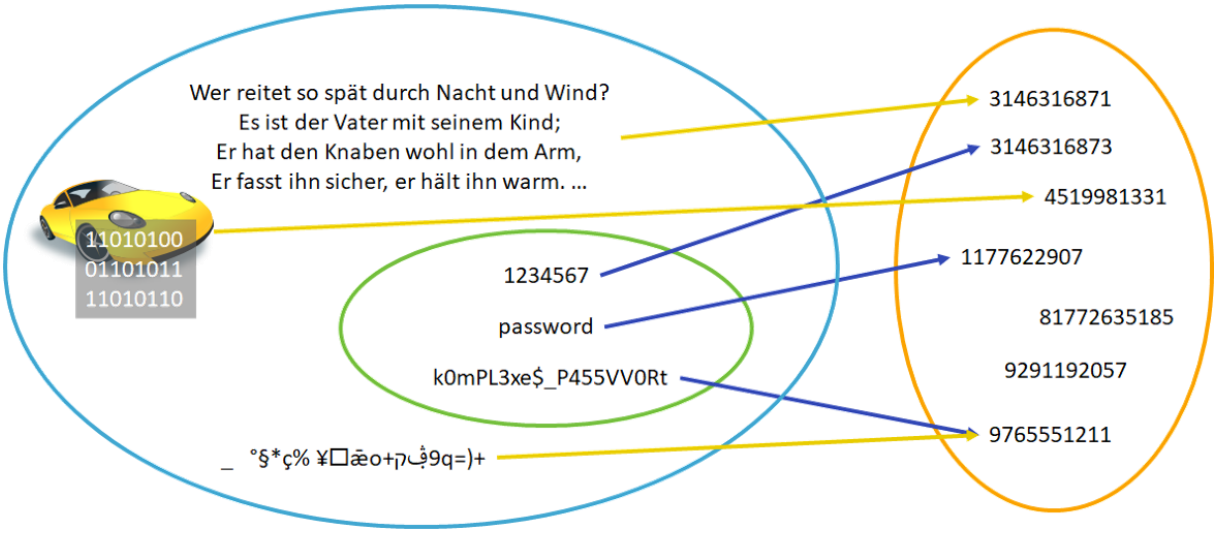


Abbildung 3: Schematische Darstellung einer Hashfunktion. (informatik.mygymer.ch, n.d.)
Hellblau: Eingabewerte, Orange: Hashwerte. Beim untersten Hashwert kommt es zu einer Hashkollision, Grün: Passwörter als spezielle Teilmenge hervorgehoben

3.3 NumPy

NumPy (NumPy, 2024) ist eine Programmierbibliothek für Python, die in dieser Arbeit vor allem für mehrdimensionale Arrays und deren einfache Handhabung verwendet wurde. Mehrdimensionale Listen sind auch in Python schon integriert, jedoch bietet NumPy Funktionen an, mit denen man die Listen einfach modifizieren kann. In diesem Projekt wurden verschiedene Funktionen von NumPy verwendet, die wichtigsten sind:

- `Roll()`
  Diese Funktion ermöglicht es, den Inhalt eines dreidimensionalen Arrays in eine der drei Achsenrichtungen zu verschieben.
- `Rot90()`

  Mit dieser Funktion kann ein dreidimensionales Array um 90° um eine der drei Achsen gedreht werden.

NumPy-Arrays sind in den meisten Fällen schneller als Python-Listen, da sie den Arbeitsspeicher effizienter nutzen. Dies wird dadurch erreicht, dass NumPy-Arrays nur

Elemente des gleichen Datentyps speichern. NumPy Arrays basieren nicht nur auf Python, sondern auch auf der Programmiersprache C, die effizienter als Python ist. (Liu, 2024)

4 Ubongo 3-D-Algorithmus

4.1 Aufbau des Algorithmus

Ein Ziel dieser Arbeit ist es, die Optimierungsschritte miteinander zu vergleichen. Dies soll ohne grossen Aufwand möglich sein. Deshalb soll der Code möglichst modular aufgebaut werden. Das bedeutet, dass der Code in kleine unabhängige Teile zerlegt wird, die leicht ausgetauscht werden können. Dazu werden verschiedene Dateien erstellt, die jeweils eine eigene Funktion erfüllen, was der Erweiterbarkeit dient.

Der vollständige Quellcode des Algorithmus kann unter folgendem Link eingesehen werden: https://github.com/gianfederer/Ubongo-3-D-Algorithmus.git.

4.2 Verwendete Hilfsmittel

4.2.1 Programmierung mit KI

Die Profilierung des Programmes, das Abspeichern der Daten in Excel wurden mehrheitlich von ChatGPT (ChatGPT, n.d.) erstellt. Die Optimierungen zur Überprüfung wurden auch mithilfe von ChatGPT erstellt, jedoch nicht allein durch KI. Während des ganzen Programmierprozesses wurde auf ChatGPT zurückzugegriffen, um Fehler ausfindig zu machen.

4.2.2 Verwendete Python-Bibliotheken

Eine Python-Bibliothek ist eine Sammlung von vorgefertigten Code-Modulen oder -Paketen.
In dieser Arbeit wurden die folgenden verwendet:

- Profilierung:
  - cProfile (The Python Profilers, 2024)
  - pstats (The Python Profilers, 2024)
- Abspeichern der Daten in einem Excel:
  - Os (Miscellaneous operating system interfaces, 2024)
  - Pandas (Pandas, 2024)
  - OpenPyXL (OpenPyXL, 2024)
- Arbeiten mit Arrays:
  - NumPy (NumPy, 2024)
- Zeitmessungen:
  - Time (time — Time access and conversions, 2024)
- Hashing:

- Hashlib (hashlib — Secure hashes and message digests, 2024)

4.3 Hauptkomponenten

Der Code besteht aus fünf Komponenten. Jede Komponente ist als separate Datei gespeichert (xxx.py), um den Code übersichtlich zu halten. In diesem Abschnitt wird ein grober Überblick über die Komponenten geschaffen, eine detailliertere Beschreibung folgt in Kapitel 4.4.

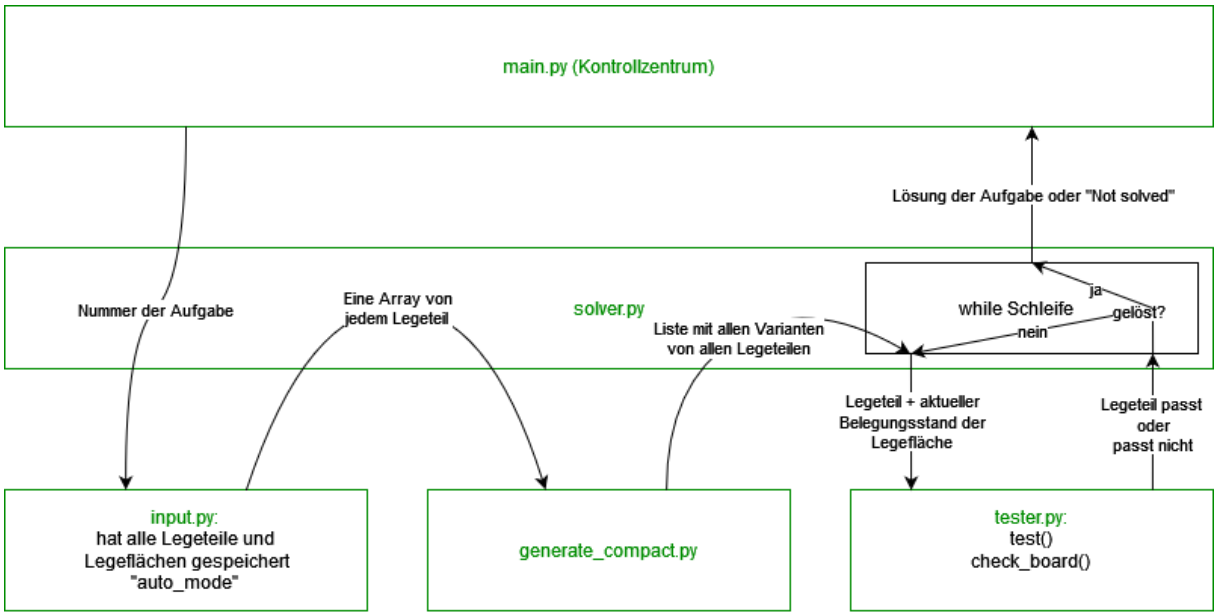


Abbildung 4: graphische Darstellung des Algorithmus
Vereinfachte Visualisierung des Algorithmus zur Lösung von Ubongo 3-D-Aufgaben
Die Hauptkomponenten sind grün dargestellt

1. **main.py** - Steuerung und Profilerstellung
   Hier befindet sich der Startpunkt des Programmes, man könnte sagen, dass diese Datei als Kontrollzentrum fungiert. Die Funktion `solve()` wird von dieser Datei aufgerufen und erhält die notwendigen Daten, um die Aufgabe zu lösen. Auch die Zeitmessung und das Abspeichern der gesammelten Daten in einer Excel-Tabelle wird in main.py durchgeführt, da diese Funktion nur einmal ausgeführt wird und sich daher perfekt für diesen Zweck eignet. Es wird nicht nur die Zeit gemessen, die zur Lösung einer Aufgabe benötigt wird. Mithilfe von cProfile wird untersucht, wie oft welche Funktion aufgerufen wird und wie viel Zeit jede Funktion benötigt. Diese Daten erleichtern die Optimierung des Programmes, da sie einen Überblick über das Optimierungspotential bestimmter Funktionen geben.
2. **solver.py** - Lösung des Puzzles
   Das Rückgrat des Algorithmus, die Backtracking-Logik, befindet sich in dieser Datei. Diese Komponente koordiniert die anderen Hauptkomponenten, die eine kleinere, spezifischere Funktion erfüllen. Die Daten der Puzzleaufgabe werden von input.py an generate_compact.py übergeben, wo sie weiter verarbeitet werden (Mehr dazu unter Punkt: 4 und 5, Kapitel: 4.3). Nachdem die für die Lösung

notwendigen Ausgangsdaten erzeugt wurden, werden mithilfe der Backtracking-Logik und tester.py die richtigen Positionen der Teile ermittelt.

3. **tester.py** - Überprüfung der Platzierung auf der Legefläche
   Die Aufgabe dieser Komponente ist es, zu überprüfen, an welchen Positionen auf der Legefläche die von solver.py vorgegebene Variante eines Legeteiles platziert werden kann. Wenn eine mögliche Platzierung gefunden wurde, wird diese an solver.py zurückgegeben.
4. **generate_compact.py** - Generierung der möglichen Ausrichtungen der Puzzleteile.
   Von jedem Legeteil gibt es eine Variante, die gespeichert ist. Die Legeteile befinden sich in einem 3x3x3-Array (Abbildung 4), da dieses sehr einfach gedreht werden kann. Es gibt eine Basisklasse, die jedes der 3x3x3-Arrays drehen kann und alle Richtungen ermitteln kann, in die ein Legeteil im 3x3x3-Array verschoben werden kann, ohne den Würfel zu verlassen. Schliesslich werden die Legeteile auch bewegt, um alle möglichen Varianten zu erzeugen, wie sich ein Legeteil im 3x3x3-Array befinden kann.

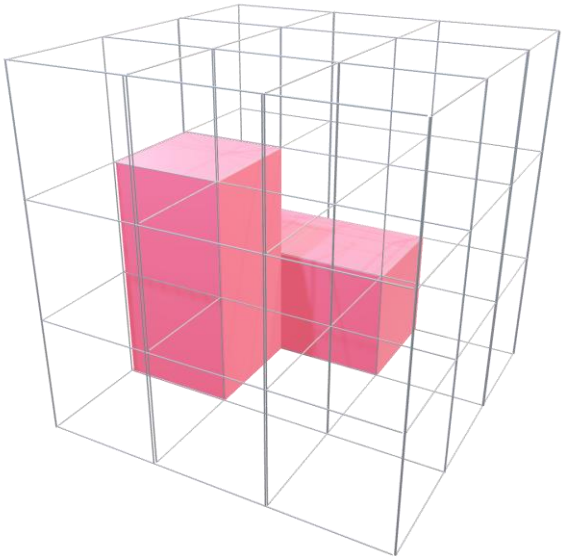


Abbildung 5: Legeteil (Nr. zehn) im 3x3x3-Array. Aus der aktuellen Position kann das Legeteil noch je eine Position nach vorne, oben und rechts verschoben werden.

5. **input.py** - Speichern der Legeteile und Legeflächen
   In der Datei input.py sind alle Legeteile und einige Legeflächen definiert und abgespeichert. Über die Variable «auto_mode» wird in input.py unter anderem festgelegt, ob einzelne Aufgaben oder Aufgabensätze gelöst werden sollen. Bei deaktiviertem «auto_mode» wird der Benutzer aufgefordert, die Legefläche und die zugehörigen Legeteile selbst einzugeben. Die Vorgehensweise ist in Kapitel 4.4.4 beschrieben. Bei aktiviertem «auto_mode» wird über eine Variable, in der die Anzahl der Legeteile definiert ist, festgelegt, welcher der gespeicherten Aufgabensätze gelöst werden soll.

4.4 Programmierung der Hauptkomponenten

4.4.1 Solver.py

In dieser Datei gibt es nur eine Funktion mit dem Namen `solve()`. Nachdem diese Funktion die benötigten Daten erhalten hat, wird eine while-Schleife ausgeführt. Diese

sorgt dafür, dass die Backtracking-Logik versucht Teile zu platzieren, bis eine Lösung gefunden wurde oder alle Varianten durchprobiert wurden. In dieser Schleife befindet sich eine weitere while-Schleife, die dafür sorgt, dass, nach dem Testen in tester.py immer wieder neue Legeteile oder andere Varianten desselben Legeteiles verwendet werden. Es ist wichtig, dass dies allgemein programmiert wird, sodass derselbe Code für alle Legeteile funktioniert. Dies ermöglicht eine einfache Implementierung von Änderungen und eine einfache Erweiterbarkeit auf eine beliebige Anzahl von Legeteilen.

4.4.2 Tester.py

Die Funktion `test()` ist der Hauptteil von tester.py. `Test()` wird mehrmals von `solve()` aufgerufen und erhält eine Variante eines Legeteiles und den aktuellen Belegungszustand der Legefläche. Die Funktion `test()` platziert das Legeteil an jeder möglichen Position auf der Legefläche und prüft, ob die Aufgabe gelöst, noch lösbar ist oder eine Sackgasse ist. Die Funktion `check_board()`, die die Überprüfung durchführt, kann drei Zeichenketten zurückgeben. Diese sind «Mistake», «Not done» und «Solved», je nach Ausgabe wird in der Funktion `solve()` Backtracking initiiert, ein neues Legeteil dazu genommen oder, aufgrund gefundener Lösung, das Backtracking beendet.

4.4.3 Generate_compact.py

Um eine Liste aller Varianten eines Legeteils zu erzeugen, werden verschachtelte Schleifen verwendet. In der ersten Schleife wird das Legeteil um zwei verschiedene Achsen gedreht. Es müssen Rotationen um die verschiedenen Achsen durchgeführt werden, da sonst nicht alle 24 Orientierungen erzeugt werden können. Jede Seite des Würfels kann nach vorne gedreht werden, dann wird der Würfel um die Achse gedreht, die durch die Vorderseite verläuft. So erhält man 6 x 4 Orientierungen, was 24 ergibt. Nach der Drehung des Legeteiles folgen drei Schleifen, in denen das Legeteil jeweils in eine Richtung bewegt wird. Damit das Legeteil nicht aus dem Block herausgeschoben wird, prüft eine Funktion nach der Drehung, in welche Richtungen das Legeteil innerhalb des 3x3x3 Arrays verschoben werden kann.

4.4.4 Input.py

Wenn der «auto_mode» deaktiviert ist, werden die Benutzenden aufgefordert, einige Eingaben zu machen: Zuerst muss die Anzahl der Teile angegeben werden, dann werden die Nummern der Legeteile eingegeben. Damit das Programm auf alle möglichen Legeflächen angewendet werden kann, werden diese nicht mit einer Zahl eingegeben. Zuerst muss die maximale horizontale und vertikale Ausdehnung der Legefläche angegeben werden. Danach erstellt input.py daraus ein Rechteck mit kleinen Quadraten, für die die Benutzenden angeben müssen, ob sie zur Legefläche gehören oder nicht (Abbildung 6).

```
Bitte geben Sie die Anzahl an Teilen an (3 oder 4)  4
Bitte geben Sie die Nummer Ihres Teils an  3
Bitte geben Sie die Nummer Ihres Teils an  10
Bitte geben Sie die Nummer Ihres Teils an  8
Bitte geben Sie die Nummer Ihres Teils an  5
Bitte geben Sie die Länge in horizontaler Richtung an (mind. 3)  4
Bitte geben Sie die Länge in vertikaler Richtung an (mind. 3)  3
('Ist Feld ', 1, ' auf der linken Seite in der Reihe ', 1, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')1
('Ist Feld ', 2, ' auf der linken Seite in der Reihe ', 1, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')1
('Ist Feld ', 3, ' auf der linken Seite in der Reihe ', 1, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')0
('Ist Feld ', 4, ' auf der linken Seite in der Reihe ', 1, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')1
('Ist Feld ', 1, ' auf der linken Seite in der Reihe ', 2, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')0
('Ist Feld ', 2, ' auf der linken Seite in der Reihe ', 2, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')0
('Ist Feld ', 3, ' auf der linken Seite in der Reihe ', 2, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')0
('Ist Feld ', 4, ' auf der linken Seite in der Reihe ', 2, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')1
('Ist Feld ', 1, ' auf der linken Seite in der Reihe ', 3, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')0
('Ist Feld ', 2, ' auf der linken Seite in der Reihe ', 3, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')0
('Ist Feld ', 3, ' auf der linken Seite in der Reihe ', 3, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')0
('Ist Feld ', 4, ' auf der linken Seite in der Reihe ', 3, ' weiss (Eingabe: Für Ja geben Sie 0 ein, für Nein geben Sie 1 ein)  ')0
```

Abbildung 6: ausgefüllte Eingabe

```
Solved
untere Schicht:
 [[0 0 3 0]
 [4 1 2 0]
 [4 1 2 2]]

 obere Schicht
 [[0 0 3 0]
 [1 1 3 0]
 [4 4 4 2]]

 Das Lösen des Puzzles hat 0.1918838999999084 Sekunden gedauert
```

Abbildung 7: Ausgabe passend zu [Abbildung 6: ausgefüllte Eingabe].
Jede Zahl ausser der Null steht für ein Legeteil. Die Null kennzeichnet Felder ausserhalb der Legefläche.

4.5 Optimierung des Basisprogramms

Für die Optimierung des Algorithmus wurden drei Hauptkomponenten: generate_compact.py, input.py und tester.py ausgewählt und durch wiederholte Implementierungen optimiert. Die Resultate der Optimierungen sind im Kapitel 4.6 dargestellt.

4.5.1 Optimierung von generate_compact.py

Die Basisvariante des Algorithmus hat noch einige Stellen, die sehr ineffizient sind. Ein Problem ist die Erzeugung der Teile, da bei symmetrischen Legeteilen einige Varianten doppelt erzeugt werden. Um diese Duplikate zu vermeiden, kann eine Implementierung vorgenommen werden, die prüft, ob das Legeteil, das der Liste hinzugefügt werden soll, sich schon in dieser befindet. Mit der Funktion `np.array_equal` aus NumPy ist es einfach, eine solche Prüfung zu implementieren.

Um eine noch schnellere Überprüfung zu erreichen, wurde Hashing hinzugefügt. Durch Hashing (Kapitel: 3.2) der Arrays und anschliessendem Vergleichen dieser Hashwerte kann Zeit eingespart werden. Diese Effizienzsteigerung ist auf die Zeitkomplexität (Kapitel: 2.5) zurückzuführen. Das Hashing eines Arrays sowie das Vergleichen zweier Arrays haben eine Zeitkomplexität von O(n), da jedes Element einzeln verglichen wird. Der Vergleich zweier Hashwerte hat jedoch nur eine Zeitkomplexität von O(1), da diese eine konstante Grösse haben. Da die Hashwerte gespeichert werden, müssen sie nur einmal pro Legeteil berechnet werden. Der Vergleich der Arrays wird jedoch mehrmals pro Array

durchgeführt. Da die sehr unwahrscheinliche Gefahr besteht, auf eine Kollision (Kapitel: 3.2) zu treffen, wurde eine zweite Überprüfung mit `np.array_equal` hinzugefügt, wenn die Hashwerte gleich sind.

Dies ist nicht die einzige Doppelung, die in der Basisversion des Algorithmus auftritt. Durch die Implementierung des 3x3x3-Würfels werden die Legeteile manchmal mehrfach an der gleichen Stelle auf der Legefläche platziert. Eine Variante eines Legeteiles, die sich ganz links im 3x3x3 Würfel befindet und den rechten Rand nicht berührt, kann nicht am rechten Rand der Legefläche platziert werden, da der Würfel auf der Legefläche platziert werden muss. Um dieses Problem zu umgehen, wurde zuerst eine zweite Variante des Legeteiles erstellt, die sich ganz rechts im 3x3x3 Würfel befindet. Diese Massnahme löst das Problem, jedoch wird das Legeteil manchmal an derselben Stelle in der Mitte der Legefläche doppelt platziert. Um diesem Problem entgegenzuwirken, wird bei der dritten Optimierung das Legeteil in die linke, untere, hintere Ecke verschoben und das Array auf die kleinstmögliche Grösse zugeschnitten. Dadurch ist es möglich, eine Variante eines Legeteiles überall auf der Spielfläche zu platzieren. Es ist also nicht notwendig, eine zweite Variante desselben Legeteils zu erzeugen, die sich nur an einer anderen Stelle im Array befindet.

In der Basisvariante werden alle Listen aller Legeteile schon zu Beginn erzeugt, jedoch werden nicht immer alle Varianten verwendet. Um dieses Problem zu lösen, werden die Varianten des ersten Legeteiles stückweise generiert. Für die folgenden Legeteile macht dies wenig Sinn, da hier fast immer alle Varianten benötigt werden.

4.5.2 Optimierung von input.py

Menschen, die Ubongo 3-D spielen, tendieren dazu, mit dem ihnen am komplexesten erscheinenden Legeteil zu beginnen. Da die Komplexität eines Legeteiles kein absoluter Wert ist, wird die Implementierung eines solchen Vorgehens erschwert. Mithilfe absoluter Werte, wie der Grösse und Oberfläche eines Legeteiles, kann eine möglichst gute Annahme für einen Komplexitätswert getroffen werden. Um diesen neuen Komplexitätswert zu optimieren, können zum Beispiel die Gewichtung der Faktoren oder die Reihenfolge, in der die Legeteile sortiert werden, geändert werden. In dieser Arbeit wurde nur die Reihenfolge geändert, in der sortiert wurde. Das heisst, die Legeteile werden zuerst nach einem Kriterium sortiert, die Legeteile, die denselben Wert haben, werden anschliessend noch nach dem zweiten Kriterium sortiert.

4.5.3 Optimierung der Funktion `check_board()`

In der `check_board()` Funktion hat die Überprüfung mit `np.count_nonzeros()`[2] zu schnelleren Resultaten geführt als die Überprüfung mit `np.any()`[3].

[2] Mithilfe einer Bedingung innerhalb der Klammern kann geprüft werden, wie viele Elemente in einem Array diese Bedingung erfüllen.
[3] Wenn eine Bedingung in den Klammern ist, gibt es zurück, ob ein Element in einem Array diese Bedingung erfüllt.

Um die Effizienz noch mehr zu steigern, wurde die Überprüfung erweitert. Nach dem Spielprinzip von Ubongo 3-D ist eine Aufgabe unlösbar, wenn ein Hohlraum entsteht, der kleiner ist als drei Würfel, da das kleinste Legeteil ein Volumen von drei Würfeln hat. Zuerst wurde die Überprüfung nach einem Würfel grossen Hohlräume implementiert. Aufgrund guter Resultate dieser Implementierung wurde die Suche auf zwei Würfel grosse Hohlräume erweitert.

4.6 Resultate der Zeitmessungen

Um während der Entwicklung eine Übersicht zu erhalten, wurden Zeittests mit einem Laptop durchgeführt. Der hat jedoch, vermutlich aufgrund unzureichender Kühlung, ungenaue Resultate zurückgeliefert, deshalb wurde für die finalen Zeittests ein Desktop-PC benutzt. Dieser hat eine NVIDIA GeForce RTX 3060[4], 16 GB 3200 MHz RAM[5] sowie einen 12th Gen Intel(R) Core(TM) i5-12400F[6], der mit dem mitgelieferten Kühler von Intel gekühlt wird, verbaut. Da die CPU für den Algorithmus nur einen von sechs Kernen benutzte, war die Auslastung auf ca. 20 %. Deshalb reicht der nicht sehr leistungsstarke mitgelieferte Kühler aus, um die CPU-Erwärmung auf ca. 7 Grad zu begrenzen. Trotzdem wurde dem Prozessor vor jeder Zeitmessung ca. fünf Minuten Zeit zum Abkühlen gegeben. Erst wenn über HWInfo[7] sichergestellt war, dass die Ausgangstemperatur wieder erreicht worden ist, wurde ein neuer Lauf gestartet. Die Zeitmessungen haben alle bei ähnlicher Raumtemperatur stattgefunden (zwischen 18 und 20 °C).
Für die Aufgaben mit drei Teilen wurde ein Aufgabensatz von 120 Aufgaben (Kombination aus Legefläche und Legeteilen) benutzt, für die vierteiligen Aufgaben wurden 40 Aufgaben benutzt.

4.6.1 Problemlösung von dreiteiligen Aufgaben

Als Startpunkt wurde die nicht optimierte Basisversion definiert. In einem ersten Schritt wurde die Erzeugung der Teile verändert.
Die schnellste Version von generate_compact.py wurde als neuen Ausgangspunkt für die weiteren Optimierungen definiert. Somit konnte Zeit gespart werden und die Diagramme werden deutlicher. Mit deutlicher ist gemeint, dass die Varianz der Daten besser ersichtlich wird, da sich die vertikale Achse über einen kleineren Zahlenbereich erstreckt. An diesem Diagramm kann man erkennen, dass Ausreisser nur selten und in geringfügigem Ausmass auftreten.

Die Optimierung hat funktioniert. Die maximale Optimierung von generate_compact.py hat den Zeitaufwand um Faktor vier reduziert. Ins Auge fällt, dass die Implementierung

[4] https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3060-3060ti/
[5] https://www.corsair.com/us/en/p/memory/cmw16gx4m2c3200c16/vengeancea-rgb-pro-16gb-2-x-8gb-ddr4-dram-3200mhz-c16-memory-kit-a-black-cmw16gx4m2c3200c16
[6] https://ark.intel.com/content/www/us/en/ark/products/134587/intel-core-i5-12400f-processor-18m-cache-up-to-4-40-ghz.html
[7] https://www.hwinfo.com/

des Hashings einen grossen Unterschied gemacht hat und dass die zusätzlichen Optimierungsschritte nur noch vergleichsweise kleine Zeitgewinne zur Folge hatten.

Das Hashing (4) und die Veränderung des Arrayformates (2) haben mit einer Reduktion auf ca. ein Drittel der benötigten Zeit je einen grossen Effekt. Wenn man sie jedoch kombiniert (5), ist der zusätzliche Zeitgewinn nur noch geringfügig. Dies liegt daran, dass sich die Veränderung des Arrayformates auf weniger Legeteile auswirken kann, wenn manche durch Vergleichen entfernt werden.

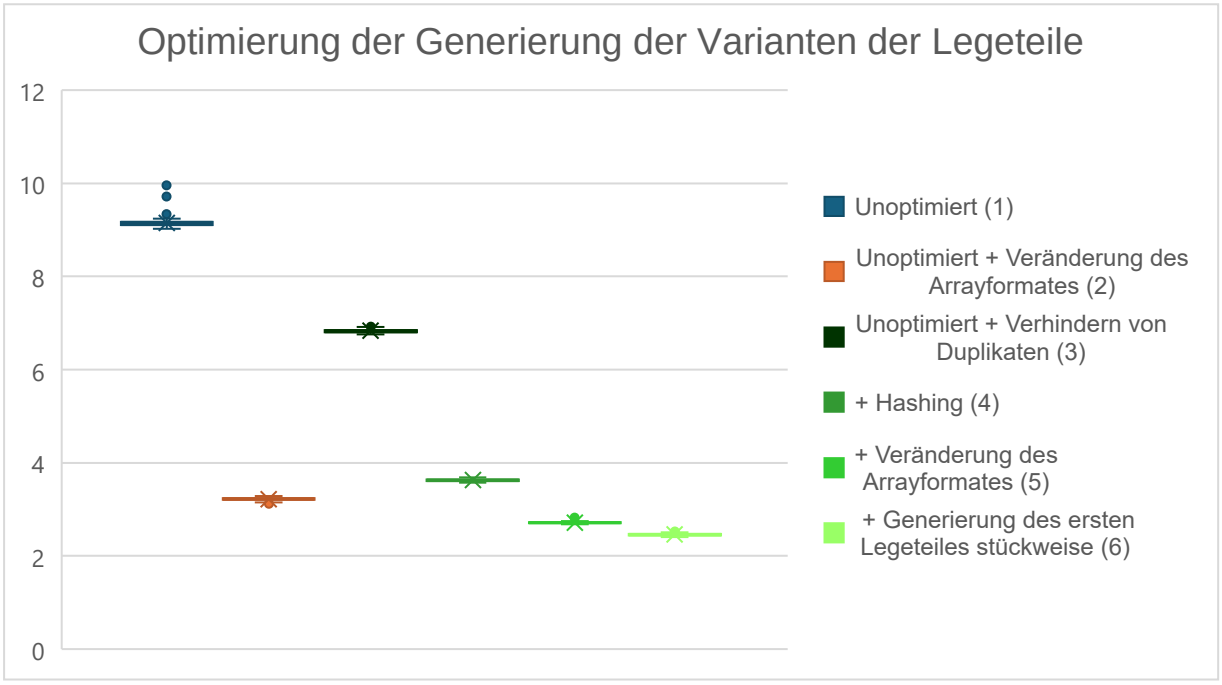


Abbildung 8: Optimierung der Generierung der Varianten der Legeteile (drei Legeteile)

Ohne Sortierung ist die Reihenfolge der Legeteile, also welches zuerst auf der Legefläche eingesetzt wird, so wie es auf den Karten von Ubongo 3-D ist.

Ordnet man die Teile jedoch nach dem Volumen, kann man beobachten, dass es effizienter ist, das grösste Teil zuerst zu platzieren. Dies ist dem Fakt geschuldet, dass ein kleines Legeteil mehr gültige Positionen auf der Legefläche hat. Weil jede gültige Position des ersten Legeteiles eine Vielzahl an Möglichkeiten, die nachfolgenden Legeteile zu platzieren, nach sich zieht, ist es wichtig, dass das erste Legeteil nur wenige gültige Positionen hat.

Da 15 der 16 Ubongo 3-D-Legeteile entweder aus vier oder fünf Würfeln bestehen, können diese nicht immer optimal sortiert werden. Deshalb wurde ein zweites Sortierungskriterium eingebaut, die Oberfläche eines Legeteiles. Damit können die Legeteile ein wenig nach der Kompaktheit geordnet werden. Zuerst wurde nach der Grösse sortiert (grösstes zuerst) und die, die dasselbe Volumen haben, noch nach der Oberfläche (grösste Oberfläche zuerst). Mit diesem Ansatz wurden jedoch schlechtere Resultate erzielt, weshalb die Sortierung umgedreht wurde (zuerst nach der Oberfläche und danach nach dem Volumen). Mit dieser Sortierung konnte das beste Resultat erzielt werden.

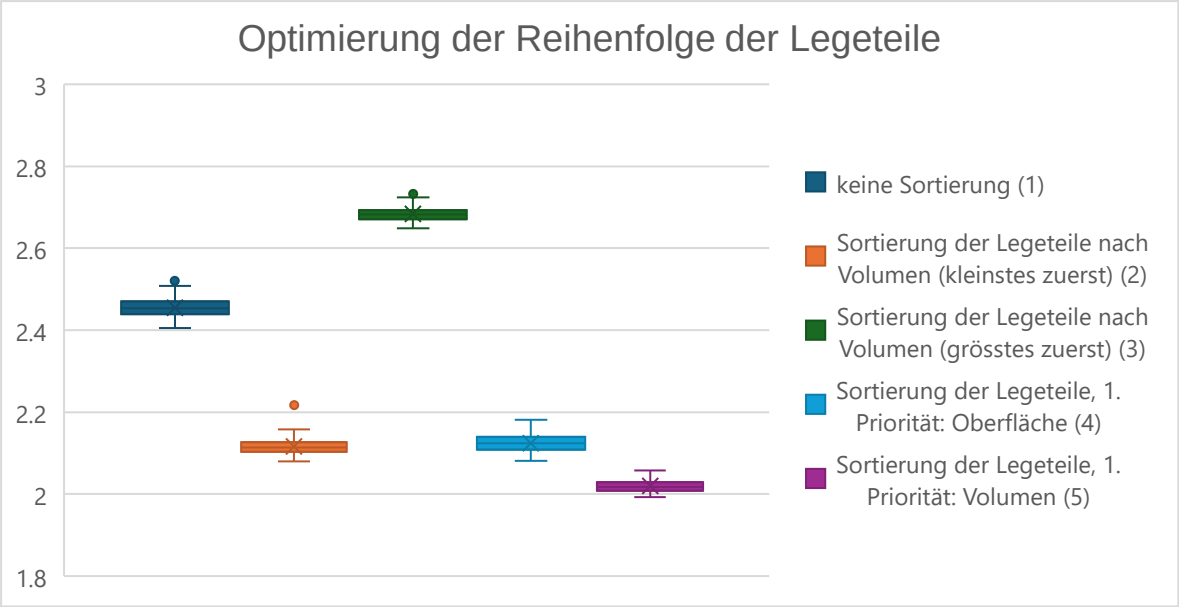


Abbildung 9: Optimierung der Reihenfolge der Legeteile (drei Legeteile)

Da die `check_board()` Funktion sehr oft aufgerufen wird, wurde diese optimiert.

Mit `np.count_nonzeros()` konnte eine Verbesserung festgestellt werden. Bei der Überprüfung nach freien isolierten Feldern konnte eine grosse Verbesserung festgestellt werden. Der Rechenaufwand, um die isolierten Felder zu finden, ist geringer, als wenn mit solchen Konstellationen weitergerechnet wird.

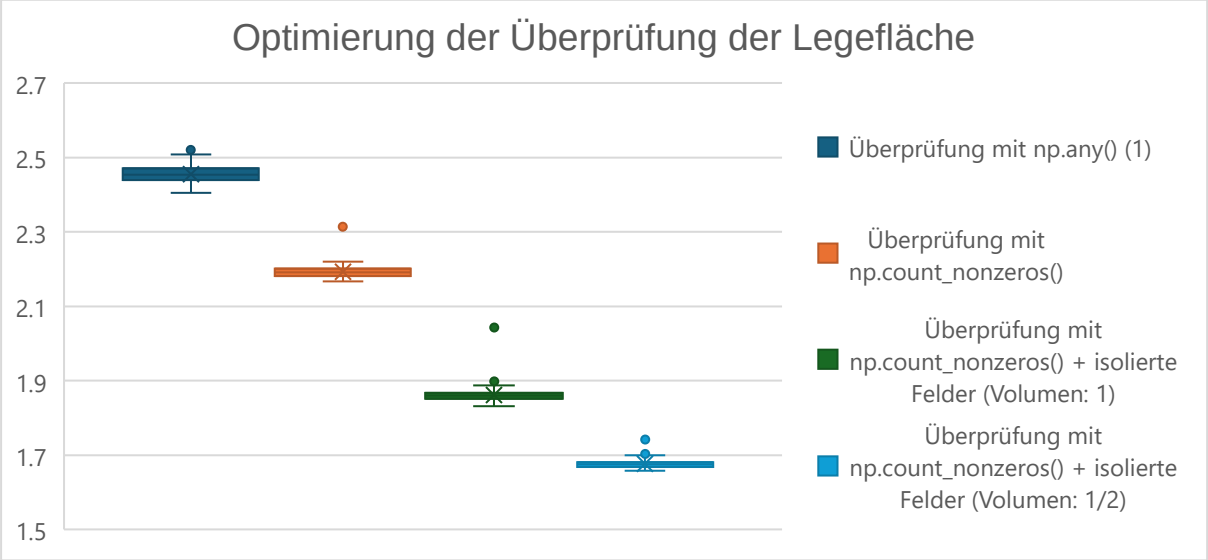


Abbildung 10: Optimierung der Überprüfung der Legefläche (drei Legeteile)

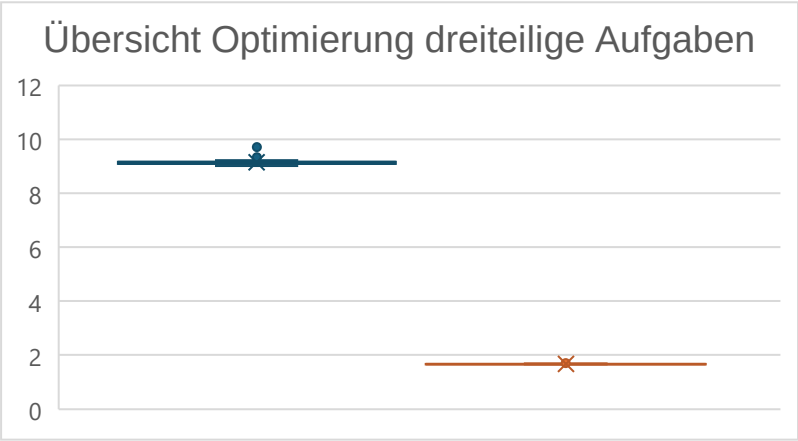


Abbildung 11: Übersicht gesamte Optimierung (drei Legeteile)

Dieses Diagramm vergleicht die Basis-Version (blau) des Algorithmus mit der fertig optimierten Version (von allen drei oben aufgeführten Kategorien die beste Optimierung [orange]). Die vollständig optimierte Version benötigt nur noch 22,25 % der Zeit, die die Basis-Version benötigt.

4.6.2 Problemlösung von vierteiligen Aufgaben

Wie schon bei den Aufgaben mit drei Legeteilen reduzieren auch bei den Aufgaben mit vier Legeteilen alle Implementierungen den Zeitbedarf (Abbildung *2*Abbildung 12). Auffällig ist jedoch, dass die Verhinderung von Duplikaten auf die Aufgaben mit vier Legeteilen einen viel grösseren Effekt hat als auf die Aufgaben mit drei Legeteilen. Dass bei vier Legeteilen die Einsparung der ersten Optimierung ca. 4/5 beträgt und bei drei Legeteilen nur ca. ein Viertel, ist mathematisch schwierig zu erklären, da die Backtrackinglogik sehr unberechenbar ist. Es können jedoch ein paar Vereinfachungen vorgenommen werden, um zu zeigen, weshalb es zwischen der Anzahl Legeteile und der Zeit eine exponentielle Abhängigkeit gibt: Die Backtrackinglogik wird durch einen Brute-Force-Ansatz ersetzt und es wird angenommen, dass es von jedem Teil 50 % der Varianten doppelt gibt. Somit müssen 1.5^(n-1) mal die Zeit ohne doppelte Varianten durchgeführt werden, wobei n die Anzahl Legeteile bedeutet. Für den Exponenten muss n-1 gewählt werden, da die Duplikate des ersten Legeteiles keinen Einfluss haben, weil das Programm abbricht oder zur nächsten Aufgabe übergeht, sobald eine Lösung gefunden wird.

Wie schon bei den Aufgaben mit drei Legeteilen hat die Formatänderung des Arrays (5) einen grösseren Effekt als die stückweise Erzeugung des ersten Legeteiles. Dies ist darauf zurückzuführen, dass die stückweise Erzeugung unabhängig von der Anzahl der Legeteile ist und der für das Backtracking benötigte Rechenaufwand für Aufgaben mit mehr Legeteilen grösser wird. Deshalb wird der Rechenaufwand für die Generierung des ersten Legeteiles kleiner im Vergleich zum Gesamtrechenaufwand.

Man kann auch beobachten, dass die letzte Optimierung (6) bei drei Legeteilen schneller war als das Hashing (3). Jedoch ist mit vier Legeteilen Nummer sechs sogar langsamer als das Vermeiden von Duplikaten (2).

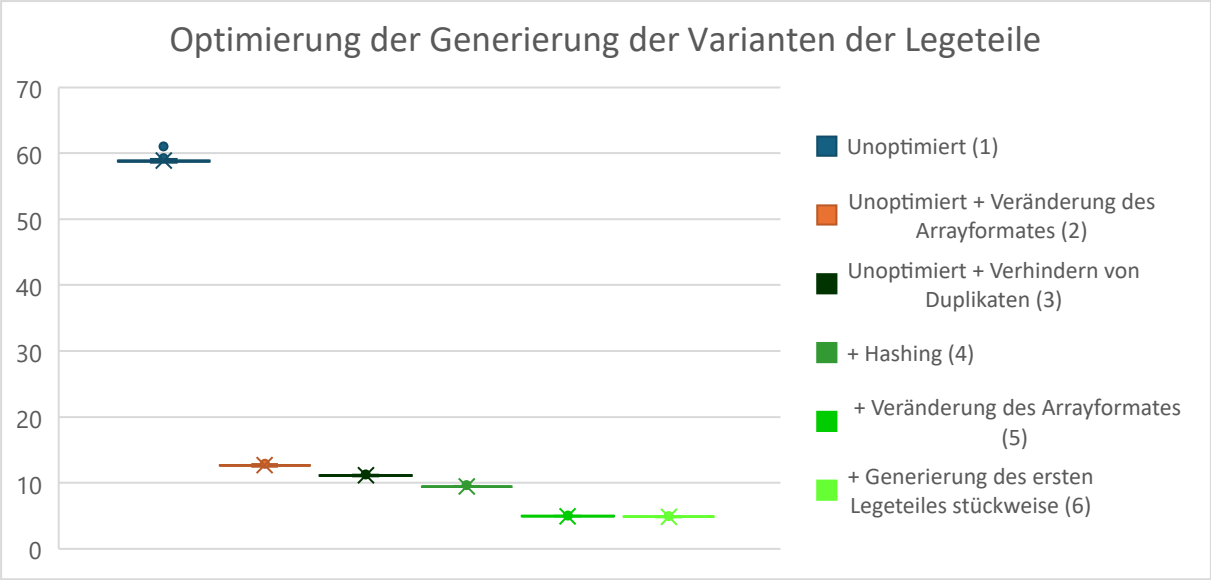


Abbildung 12: Optimierung der Generierung der Varianten der Legeteile (vier Legeteile)

Bei der Optimierung der Reihenfolge nach Komplexitätsgrad der Legeteile sieht man ein sehr ähnliches Bild, wie schon bei den dreiteiligen Aufgaben.

Die Aufgabensätze der drei- und vierteiligen Aufgaben benötigen mit der letzten Optimierung (5) ungefähr gleich lang, das heisst, dass eine vierteilige Aufgabe ca. dreimal zeitintensiver ist, da in einem Aufgabensatz nur 40 und nicht 120 enthalten sind.

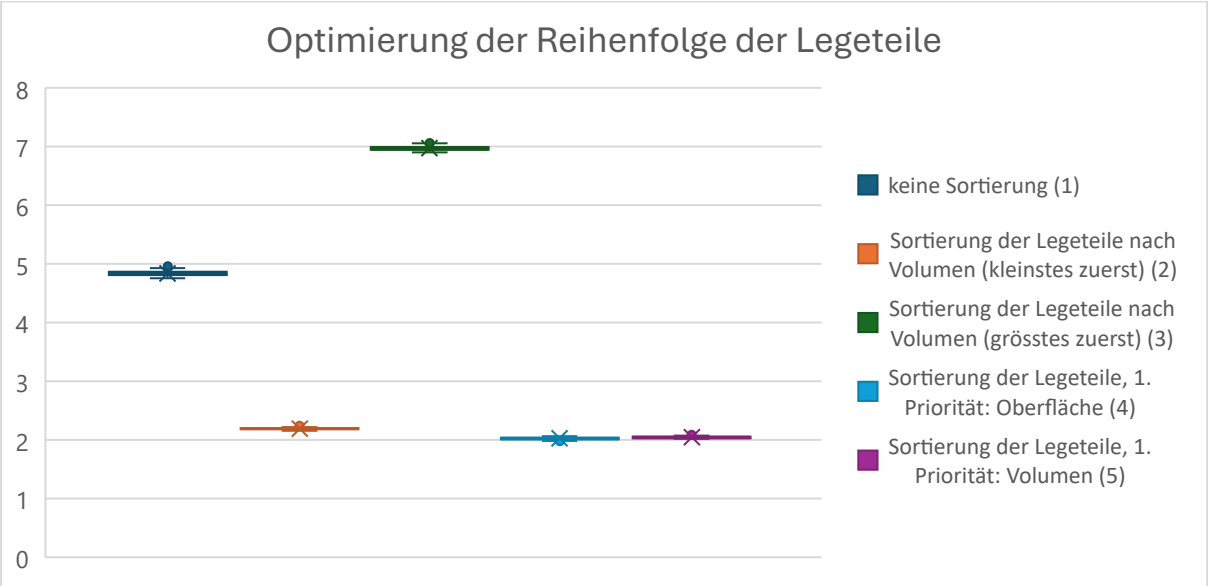


Abbildung 13: Optimierung der Reihenfolge der Legeteile (vier Legeteile)

Auch bei dieser Optimierung sind die Resultate den der Aufgaben mit drei Legeteilen sehr ähnlich.

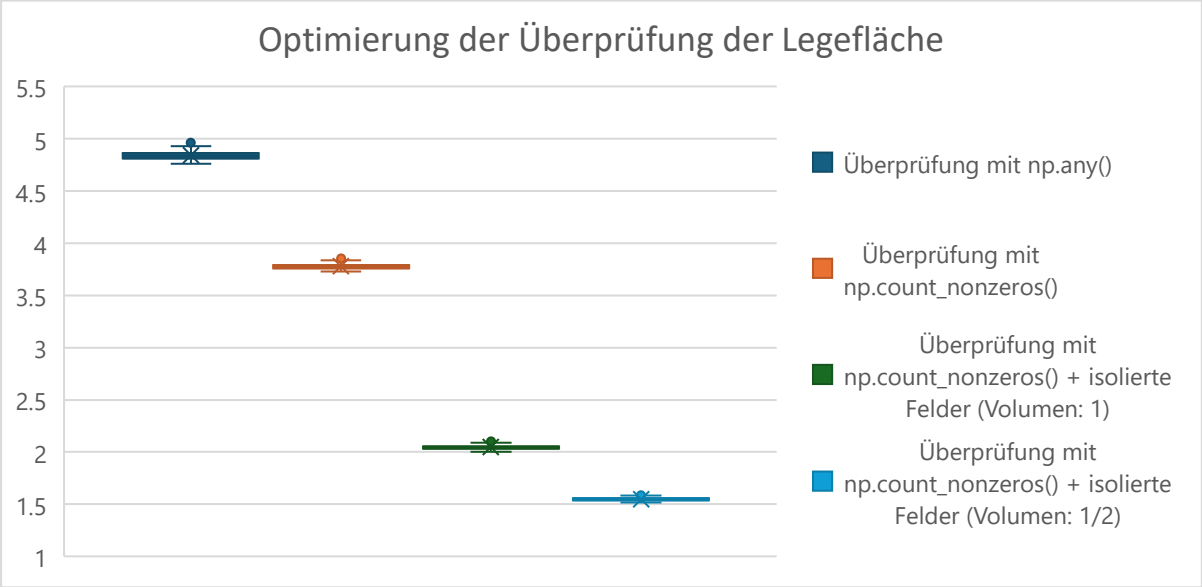


Abbildung 14: Optimierung der Überprüfung der Legefläche (vier Legeteile)

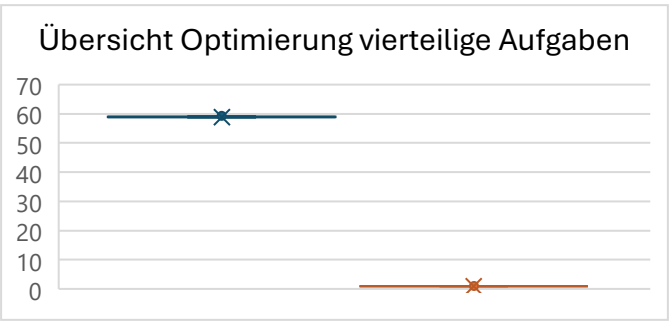


Abbildung 15: Übersicht gesamte Optimierung (vier Legeteile)

In diesem Diagramm sind die Basis-Version des Algorithmus (blau) und die finale optimierte Version zu sehen (orange). Die optimierte Version hatte im Durchschnitt 1,02 Sekunden, um 40 Aufgaben zu lösen. Die Basis-Version benötigt hingegen 58.85 Sekunden. Dies entspricht 1.73 % der von der Basis-Version benötigten Zeit. Zu Beginn wurden die dreiteiligen Aufgaben (pro Aufgabe) 19,300-mal schneller gelöst, am Schluss nur noch 1.83-mal. Somit waren die Optimierungen bei den vierteiligen Aufgaben 10,53-mal effektiver.

4.7 Verbesserungspotential

4.7.1 Ein- und Ausgabe

Heute sind sich Benutzer sehr benutzerfreundliche Schnittstellen gewohnt. Die Benutzerfreundlichkeit der Schnittstelle wurde in dieser Arbeit nicht berücksichtigt. Die Ein- und Ausgabe (Kapitel 4.4.4) funktionieren mithilfe der Konsole. Bei der Eingabe wird die Legefläche im Binärsystem angegeben (detaillierte Beschreibung in Kapitel: 4.4.4). Bei der Eingabe gibt es nebst der Gestaltung noch ein anderes Problem, das Umgehen mit Fehlern. Wenn etwas, das vom Programm nicht erwartet wird, eingegeben wird, zum Beispiel ein Wort anstatt einer Zahl, wird der Algorithmus die Lösung nicht finden und die Eingabe auch nicht wiederholen. Sollte der Benutzer aus Versehen einen unzulässigen Wert eingeben, muss der Algorithmus neu gestartet werden.

Die Lösung der Aufgabe wird als Array ausgeben. Die Lösung ist jedoch für ungeübte Benutzer nicht so schnell und intuitiv ersichtlich wie bei einer grafischen Ausgabe des Resultats. Es wäre möglich, eine graphische Oberfläche zu kreieren, jedoch ist dies zeitaufwendig und hätte den Rahmen dieser Maturaarbeit gesprengt. Da die Ausgabe nur zu Entwicklungszwecken benutzt wurde, konnte auf die graphische Umsetzung verzichtet werden.

4.7.2 Erweiterung auf beliebig viele Teile

Der Algorithmus kann momentan nur Aufgaben mit drei oder vier Teilen lösen. Um Aufgaben mit beliebig vielen Teilen zu lösen, müsste man drei Listen mithilfe der Anzahl der Teile erzeugen sowie den Output anpassen.

4.7.3 Besserer Schwierigkeitsscore bei input.py

Das Ordnen der Legeteile mithilfe der Oberfläche und des Volumens der Legeteile ruft gute Resultate hervor. Um jedoch ein noch besseres Resultat zu erzielen, müsste man zuerst analysieren, bei wie vielen Aufgaben die Teile optimal angeordnet werden. Es gibt für jede Aufgabe sechs respektive 24 Anordnungen der Legeteile, wobei eine die effizienteste ist. Die Effizienz einer Schwierigkeitsbewertung kann durch Vergleichen mit den optimalen Anordnungen abgeschätzt werden. Mit dieser Methode können verschiedene Schwierigkeitsbewertungen miteinander verglichen werden.

4.7.4 Mehr Teile stückweise generieren

Im aktuellen Algorithmus werden nur vom ersten Legeteil nicht alle Varianten des Legeteiles im 3x3x3 Würfel zu Beginn erzeugt, dies könnte auf mehr Legeteil ausgedehnt werden. Allerding ist zu prüfen, bis zu welcher Anzahl Legeteilen dies zu einer Effizienzsteigerung und ab wann zu einem Effizienzverlust führen würde.

4.7.5 Testen aller Verbesserungen einzeln

Einer der grössten Nachteile der durchgeführten Zeittests ist, dass nicht alle Optimierungen einzeln getestet wurden. Pro Kategorie wurden die Optimierungen manchmal nur hinzugefügt. Jede Optimierung einzeln zu testen, wäre mit erheblich mehr Zeitaufwand verbunden gewesen und mit zusätzlichem Programmieraufwand, um noch mehr Klassen zu erstellen.

4.7.6 Multicoreprocessing / andere Programmiersprache

Ich denke, dass es sehr interessant wäre, Programmiersprachen miteinander zu vergleichen. Da dies jedoch den Rahmen dieser Maturaarbeit sprengen würde, wurde dies unterlassen. Aus reinem Interesse wollte ich Multicoreprocessing implementieren, ich habe jedoch schnell realisiert, dass eine gute Implementierung komplex ist.

5 Produktbewertung

Der entwickelte Basis-Algorithmus funktioniert und wurde stark optimiert. Zu Beginn bot die nicht optimale Programmierung (zum Beispiel die Erzeugung von Duplikaten) Spielraum für massive Verbesserungen. Die Basis-Version wäre sicherlich effizienter gewesen, wenn sie von einem routinierteren Programmierer geschrieben worden wäre. Die grosse Effizienzsteigerung ist auch ein Zeichen meiner Lernkurve. Es konnten aber auch Optimierungen implementiert werden, die nicht nur suboptimale Programmierung ausbesserten, zum Beispiel die Überprüfung auf isolierte Felder oder das Sortieren der Legeteile nach Komplexität.

Die Effekte der Optimierungen sind sichtbar und lassen sich mit der variierenden Anzahl der Legeteile in Zusammenhang bringen. Es besteht trotz zahlreicher Optimierungen sicherlich noch Raum zur Verbesserung, zum Beispiel eine Optimierung von Multicoreprocessing (Kapitel: 4.7.6).

Der erstellte Algorithmus findet keinen grossen Anwendungsbereich, da die erfüllte Funktion noch beschränkt ist. Er bietet jedoch Potenzial für andere Programme, beispielsweise für eine Anwendung, die neue Ubongo 3-D-Aufgaben erstellen kann.

6 Danksagung

Ich danke meinem Betreuer Thomas Jampen für die wertvollen Inputs und grossartige Zusammenarbeit. Zusätzlich möchte ich meinen Eltern danken für die hilfreichen Gespräche am Familientisch und das Verständnis und die Unterstützung in der intensiven Schlussphase.

7 Literaturverzeichnis

ChatGPT. (kein Datum). Von https://chatgpt.com/ abgerufen

geeksforgeeks. (24. 06 2024). Abgerufen am 10. 10 2024 von https://www.geeksforgeeks.org/introduction-to-backtracking-2/

GitLab. (15. 10 2024). Von https://about.gitlab.com/ abgerufen

hashlib — Secure hashes and message digests. (15. 10 2024). Von https://docs.python.org/3/library/hashlib.html abgerufen

informatik.mygymer.ch. (kein Datum). Abgerufen am 10. 10 2024 von https://informatik.mygymer.ch/base/

Liu, L. (13. 01 2024). *medium*. Abgerufen am 10. 10 2024 von https://medium.com/@yuxuzi/frequent-interview-question-why-is-numpy-faster-than-python-lists-06df9c6cd3bb

Miscellaneous operating system interfaces. (15. 10 2024). Von https://docs.python.org/3/library/os.html abgerufen

NumPy. (15. 10 2024). Von https://NumPy.org/ abgerufen

OpenPyXL. (15. 10 2024). Von https://openpyxl.readthedocs.io/en/stable/ abgerufen

Pandas. (15. 10 2024). Von https://pandas.pydata.org/ abgerufen

The Python Profilers. (15. 10 2024). Von https://docs.python.org/3/library/profile.html abgerufen

time — Time access and conversions. (15. 10 2024). Von https://docs.python.org/3/library/time.html abgerufen

Visualstudio. (15. 10 2024). Von https://code.visualstudio.com abgerufen

Wikipedia. (kein Datum). *Backtracking*. Abgerufen am 14. 10 2024 von https://de.wikipedia.org/wiki/Backtracking

Woltmann, S. (28. 05 2020). *happycoders*. Abgerufen am 10. 10 2024 von https://www.happycoders.eu/de/algorithmen/o-notation-zeitkomplexitaet/

Yasar, K. (05 2024). *techtarget*. Abgerufen am 10. 10 2024 von https://www.techtarget.com/searchdatamanagement/definition/hashing

8 Abbildungsverzeichnis