1. Einleitung

Rätselspiele kennen viele Menschen von klein auf an. Die meisten Rätsel werden mithilfe simples Ausprobieren oder Knobeln angegangen. Hinter jedem Rätsel stecken allerdings mathematische Algorithmen, die zur Lösung des Rätsels führen. Beide Vorgehensweisen kann dem Brute-Force-Ansatz, das stumpfe Ausprobieren, bis die Lösung gefunden wurde, und dem Backtracking-Ansatz, das strukturierte, schematische Vorgehen an eine Problemstellung, zugeordnet werden.

**Der daraus resultierende Untersuchung ist ein Vergleich der Effizienz von Brute-Force und Backtracking Algorithmen bei der Lösung von Rätselspielen: Eine Fallstudie anhand von Sudoku und dem Labyrinth.**

Für die Untersuchung wird ein Sudoku in TypeScript modelliert und anhand eines Backtracking und Brute-Force Algorithmus gelöst. Zusätzlich wird in der Praxis ein reales Labyrinth, unter der Berücksichtigung der beiden Algorithmen, durchlaufen.

2. Der aktuelle Stand der Forschung

Nach dem heutigen Kenntnisstand sind Backtracking und Brute-Force Ansätze übliche Vorgehensweisen, um durch ein Labyrinth zu kommen und ein Sudoku zu lösen. Im Labyrinth kommt dafür die Tiefensuche (engl. *Deep-first-search = DFS)* zum Einsatz. Der Algorithmus arbeitet sich so lang einen Gang entlang, bis er auf ein Hindernis (Sackgasse) stößt. An dem Punkt kehrt er bis zur letzten Kreuzung um und probiert eine andere Richtung[[1]](#footnote-1). Beim Brute-Force Ansatz wird die Wahl des nächste Weges an einer Kreuzung durch eine willkürliche Entscheidung getroffen. Dies geschieht so lang, bis der Algorithmus ein Weg aus dem Labyrinth gefunden hat.

Ein logischer Ansatz, um ein Sudoku zu lösen, ist die Backtracking Vorgehensweise. Es wird für jede Zelle eine Zahl nacheinander eingesetzt und geprüft, ob diese gültig ist. Wenn nicht, geht der Algorithmus einen Schritt zurück und probiert eine andere Zahl[[2]](#footnote-2). Der Brute-Force Ansatz beschreibt das Ausprobieren aller möglichen Zahlen. Danach prüft der Algorithmus, ob die eingefügten Zahlen gültig sind.

Generell gilt der Brute-Force Ansatz als einfacher zu implementieren. Er ist jedoch für komplexe Problemstellungen aufgrund seiner Komplexität ungeeignet. Der Backtracking Ansatz ist, aufgrund seiner rekursiven Eigenschaft, anspruchsvoller zu implementieren, jedoch für komplexere Problemstellungen geeignet[[3]](#footnote-3).

3. Der Algorithmus

Ein Algorithmus ist eine klare Anweisung zum Durchführen von Berechnungen, bei denen eine Menge als Eingabe verwendet wird, um eine andere Menge als Ergebnis zu erzeugen. Er ist also eine Abfolge von Schritten, die dazu dienen, die Eingabe in die gewünschte Ausgabe umzuwandeln. Ein Algorithmus ist ein Werkzeug, dass verwendet wird, um ein spezifisches mathematisches Problem zu lösen. Dabei definiert das Problem selbst die Beziehung zwischen der benötigten Eingabe und der erwarteten Ausgabe. Der Algorithmus stellt dann eine konkrete Anleitung dar, wie diese Beziehung umgesetzt wird[[4]](#footnote-4).

3.1 Der Brute-Force Ansatz

Wie bereits erwähnt beschreibt der Brute-Force Ansatz das Ausprobieren, um eine Lösung für das Problem zu finden. Es werden alle Szenarien berechnet und anschließend miteinander verglichen. Unter der Betrachtung des gewünschten Resultates wird das entsprechende Ergebnis zurückgegeben[[5]](#footnote-5).Je nach Anwendungsfall hat der Brute-Force Algorithmus eine andere Komplexität.

Angenommen es existiert eine nicht sortierte Menge A mit

*A:= {5, 2, 7, 7, 3, 8, 1, 9}*

Diese soll mithilfe eines *Bubbles-Sorts* aufsteigend sortiert werden. Wichtig zu beachten ist, dass bei einem Bubble-Sort pro Durchlauf jeweils nur ein Zahlenpaar getauscht werden darf. Es muss zunächst also die ***5***mit der ***2*** getauscht werden. Danach die ***3***mit der ***7***und wieder die ***3*** mit der ersten ***7*** usw. Anhand des Beispiels wird ersichtlich, dass der Brute-Force in diesem Anwendungsfall eine quadratische Komplexität von O(n2) hat.

Ist die Menge A bereits aufsteigend sortiert mit

*A´:= {1; 2; 3; 5; 7; 7; 8; 9)*

beträgt die Laufzeitkomplexität des Algorithmus lediglich O(n), da nur ein Durchlauf nötig ist, um zu prüfen, ob die Liste bereits sortiert ist.

3.2 Der Backtracking Ansatz

Der Backtracking Ansatz ist eine Weiterentwicklung des reinen *„trial and error“*-Prinzips. Durch seine Rekursive Eigenschaft, kann der Algorithmus das ganze Problem in kleinere Teilproblem zerlegen. Zusätzlich optimiert er beim Durchlaufen den Suchraum, in dem systematisch Entscheidungen getroffen werden, die vorher definiert worden. Somit ist er auch in der Lage, getroffene Entscheidungen rückgängig zu machen, wenn diese in eine Sackgasse führen[[6]](#footnote-6). Anhand des Rucksackproblems kann die Funktionsweise des Algorithmus demonstriert werden.

Die Problemstellung ergibt sich wie folgt: Es existiert ein Rucksack mit einer begrenzten Kapazität und Gegenstände, die jeweils ein Gewicht und ein Wert haben. Das zulässige Gewicht des Rucksacks wird durch die Kapazität beschränkt. Ziel ist es die Kombinationen aus Gegenständigen zu finden, die die Kapazität des Rucksackes nicht überschreitet und die höchste Summe der Werte abwirft[[7]](#footnote-7).

Der Algorithmus beginnt mit einem leeren Rucksack. Es wird sukzessiv für jeden Gegenstand entschieden, ob dieser in den Rucksack gelegt wird. Dabei wird überprüft, ob die Kapazität des Rucksacks nicht überschritten wird. Ist das nicht der Fall wird der Gegenstand dem Rucksack hinzugefügt und der Gesamtwert aufaddiert. Dieser Schritt wird rekursiv auf die verbleibenden Gegenstände wiederholt. Wenn allerdings die Kapazität droht überschritten zu werden, wird zurückverfolgt, also ein oder mehrere Gegenstände entnommen, um einen anderen Gegenstand zu wählen. Dort findet das *Back-Tracken* statt. Dies wird nun so lang fortgeführt, bis alle (logischen) Möglichkeiten erkundet wurden. Die beste gefundene Lösung entspricht der optimalen Auswahl von Gegenständen im Rucksack, die den höchsten Gesamtwert erzielt, ohne die Kapazität zu überschreiten.

1. Vgl. Vöcking, et al., 2008 [↑](#footnote-ref-1)
2. Vgl. Saturn Cloud, 2023 [↑](#footnote-ref-2)
3. HIER FEHLT EINE QUELLE [↑](#footnote-ref-3)
4. Vgl. Cormen, et al., 2013 [↑](#footnote-ref-4)
5. Vgl. Vöcking, et al., 2008 [↑](#footnote-ref-5)
6. Vgl. Wolf, et al., 2013 [↑](#footnote-ref-6)
7. Vgl. Vöcking, et al., 2008 [↑](#footnote-ref-7)