# Mathematische Modellierung
# Aufgabe 1 - Nim Spiel


## Maximilian Röck
## Matrikelnummer: 185594


**Erläuterungen zu meiner Implementierung des Nim-Spiel-Algorithmus**

Ich habe mich dazu entschlossen, den Nim-Algorithmus dergestalt zu implementieren, dass der Spieler, welcher den letzten Haufen entfernt, als Gewinner aus dem Spiel hervorgeht. Da es auch die entgegengesetzte Variante des Nim-Spiels gibt, will ich an dieser Stelle hervorheben, dass ich die hierfür notwendigen Anpassungen – falls gefordert – gerne nachträglich noch vornehme.

Da ich den Algorithmus für eine beliebige Anzahl an Haufen und deren Elemente implementieren wollte, musste ich mich mit der Funktionsweise des Algorithmus, wie sie von Bouten veröffentlicht wurde, auseinandersetzen.
Der Zwischenschritt, bei welchem eine Betrachtung der Haufengrößen im Zweier-System notwendig wird, konnte ich allerdings nicht glaubhaft dem eigenen Verständnis nach erklaeren.
Da es durch diese Erklärungslücke nach jedem von meinem Algorithmus erzeugten neuen Spielzustand fraglich ist, ob dieser tatsächlich eine Verluststellung für den Widersacher darstellt, oder dieser nach weiteren Zügen nur zufällig zum Sieg (des Computers) führt, habe ich einen weiteren Algorithmus zum Nachweis der Verluststellungsqualität der erzeugten Spielzustände hinzugefügt. Dieser Beweis-Algorithmus wird im zweiten Abschnitt dieser Erläuterung behandelt.

**Haupt-Algorithmus**

<u>Uebersicht</u>
In einem ersten Schritt werden zunächst alle unmittelbar auf den gegebenen Zustand nachfolgenden möglichen Folgezustände erzeugt.

Diese Folgezustände werden daraufhin hinsichtlich ihrer Eigenschaft, ausschließlich gerade Nim-Summen vorzuweisen, gefiltert.

Enthält die resultierende Liste der nach Nimsummengeradheit gefilterten Folgezustände wenigstens ein Element, so wird dieses als nächster gültiger Zustand, welcher eine Verluststellung für den Widersacher repräsentiert, zurückgegeben.

Entspricht keiner der erzeugten Folgezustände einer Verluststellung und ist die gefilterte Liste somit leer, wird vom gegebenen Ausgangszustand ein "nichtleerer" Haufen entfernt, und dieser modifizierte Zustand als künftiger Folgezustand zurückgegeben.

1. Generierung aller moeglichen unmittelbar nachfolgenden Zustaende

Ein Zustand wird durch die Auflistung seiner Haufengrößen repraesentiert und kann sich als Liste mit Ganzzahlwerten gedacht werden. So entspricht die Liste [2,3,4] einem Zustand, welcher drei Haufen der Größen zwei, drei und vier entspricht.

Um alle Folgezustände zu diesem Beispielszustand zu erzeugen, versehe ich zunächst jedes Element der Liste mit einem Index. Es folgt die Transformation:

[2,3,4]

=>

[(0,2),(1,3),(2,4)]

Das Ergebnis ist eine Liste mit Tupeln, deren erstes Element jeweils den Index des Tupels in der Liste repräsentiert, und deren zweites Element eine Haufengröße darstellt.

Nun wird jeder Tupel der Liste ein weiteres mal umgewandelt. Dabei bleibt das erste Element des Tupels, der Index, unverändert erhalten, während das zweite Element, die Haufengröße, in eine Liste aller möglicher Haufengrößen umgewandelt wird, die bei einer Auswahl des entsprechenden Haufens resultieren können.
Gemäß den Spielregeln darf nämlich pro Spielzug nur ein Haufen modifiziert werden. Dabei muss mindestens ein Element entfernt werden. Es können aber auch alle Elemente des zu modifizierenden Haufens entfernt werden. In so einem Fall würde nach meinem Modell ein leerer Haufen entstehen, welcher durch eine Haufengröße von null repräsentiert wird. Entsprechend erfolgt die Transformation:

[(0,2),(1,3),(2,4)]

=>

[(0, [0,1]), (1, [0,1,2]), (2, [0,1,2,3])]

In einer weiteren Transformation werden nun Tupel in die Liste aller möglicher Zustände, welche sich aus einer Manipulation des jeweiligen Haufens ergeben können, umgewandelt. Da die so erzeugten Zustände ebenfalls als Liste repräsentiert werden, ist das Resultat eine dreifach geschachtelte Liste, sprich eine Liste von Listen von Listen.

Hierzu wird das erste Element des Tupels, der Index, dafür genutzt, eine Austauschung der entsprechenden Haufengröße im ursprünglich gegebenen Zustand, mit einer der möglichen neuen Haufengröße vorzunehmen.

[(0, [0,1]), (1, [0,1,2]), (2, [0,1,2,3])]

=>

[ [ [0,3,4],[1,3,4] ], [ [2,0,4],[2,1,4],[2,2,4] ], [ [2,3,0],[2,3,1],[2,3,2],[2,3,3] ] ]

Anschließend wird eine Verschachtelungsebene aufgelöst, und man erhält eine Liste an Listen. Die inneren Listen repräsentieren dabei alle möglichen direkten Folgezustände.

[ [ [0,3,4],[1,3,4] ], [ [2,0,4],[2,1,4],[2,2,4] ], [ [2,3,0],[2,3,1],[2,3,2],[2,3,3] ] ]

=>

[ [0,3,4],[1,3,4],[2,0,4],[2,1,4],[2,2,4], [2,3,0],[2,3,1],[2,3,2],[2,3,3] ]

2. Filtern der Zustaende nach geraden Nim-Summen

Nachdem nun eine Liste mit allen möglichen unmittelbaren Folgezuständen vorliegt, so müssen nun diejenigen Zustände ausgefiltert werden, welche keine Verluststellung für den Widersacher darstellen.

Ein Zustand gilt dann als Verluststellung, wenn all seine Nimsummen gerade sind. Daher müssen zunächst die Nimsummen für jeden Zustand erzeugt werden.
Hierfür werden zunächst alle Haufengrößen eines Zustands in Binärdarstellung umgewandelt.

[0,3,4]

=>

[“0”,”11”,”100”]

Je nach verwendetem Algorithmus erfolgt die Umwandlung in Binärdarstellung in einen String oder in ein Array. In meinem Algorithmus wird zunächst in die String-Repräsentation umgewandelt.

Um die Nimsummen bestimmen zu koennen, muessen die nun vorliegenden Binärziffern gemäß ihrer Stellung aufsummiert werden. Um die Ziffern gemäß ihrer Position zu gruppieren bietet es sich an, die Liste von Strings als Matrix aufzufassen und diese zu transponieren.
Hierzu sollten die Strings zunächst auf die gleiche Länge gebracht werden. Entsprechend werden sie von links mit Nullen aufgefüllt (im Algorithmus wurden 32 Stellen angenommen).

[“0”,”11”,”100”]

=>

[“000”,”011”,”100”]

Nun erfolgt die Umwandlung in eine Matrix. Da Strings lediglich syntaktischer Zucker für Listen von Buchstaben sind, ist diese Umwandlung mit Hilfe einer Parse-Funktion, welche auf jeden Element dieser Buchstabenliste angewandt wird, trivial.

[“000”,”011”,”100”]

=>

[ [0,0,0] , [0,1,1] , [1,0,0] ]

=>

[[0,0,0] ,
[0,1,1] ,
[1,0,0]]

Nun kann die Transponierung der Matrix erfolgen. Entweder verwendet man eine Bibliotheksfunktion, oder man transponiert “von Hand”. Ich habe mich aus Ermangelung einer passenden Funktion für Letzteres entschieden.

Hierfür werden die Elemente der inneren Liste zunächst mit ihrem Index innerhalb dieser inneren Listen versehen. Dieser Index repräsentiert bei einer Betrachtung als Matrix die Spalte, in welcher sich die Elemente befinden.

[[0,0,0] ,
[0,1,1] ,
[1,0,0]]

=>

[[(0,0),(1,0),(2,0)] ,
 [(0,0),(1,1),(2,1)] ,
 [(0,1),(1,0),(2,0)]]

Nun können die Elemente gemäß ihrem Spaltenindex Gruppiert werden

[[(0,0),(1,0),(2,0)] ,
 [(0,0),(1,1),(2,1)] ,
 [(0,1),(1,0),(2,0)]]

=>

[[(0,0),(0,0),(0,1)] ,
 [(1,0),(1,1),(1,0)] ,
 [(2,0),(2,1),(2,0)]]

Anschließend werden die Indizes wieder entfernt.

[[(0,0),(0,0),(0,1)] ,
 [(1,0),(1,1),(1,0)] ,
 [(2,0),(2,1),(2,0)]]

=>

[[0,0,1] ,
 [0,1,0] ,
 [0,1,0]]

Um die Nimsummen des Zustands zu erhalten, werden die Elemente der inneren Listen aufsummiert.

[[0,0,1] ,
 [0,1,0] ,
 [0,1,0]]

=>

[1 ,
 1 ,
 1]

=>

[1,1,1]

Nun wird für jede Nimsumme ihre Geradheit bestimmt. Nur wenn alle Nimsummen gerade sind, handelt es sich bei dem Zustand um eine Verluststellung und der Zustand wird nicht ausgefiltert.

[1,1,1]

=>

[False,False,False]

=>

False && False && False

=>

False

Bei dem in diesem Beispiel behandelten Zustand handelt es sich folglich nicht um eine Verluststellung und er wird somit aussortiert.

3. Rueckgabe des ersten zutreffenden Zustandes oder Reduzierung des gegebenen Zustandes um einen Haufen

Sind nun direkten Folgezustände gemaess des vorangegangenen Algorithmus gefiltert worden, so wird das erste Element der Ergebnisliste als neuer Zustand zurückgegeben.

Entspricht keiner der Folgezustände dem Kriterium der Nimsummengeradheit, so liegt eine leere Ergebnisliste vor. Um einen neuen Zustand zu erhalten wird vom gegebenen Zustand lediglich ein nicht-leerer Haufen entfernt, sprich seine Haufengröße auf null gesetzt.

[0,3,4]

=>

[0,0,4]

**Beweis-Algorithmus**

Die Motivation für den Beweisalgorithmus wurde anfangs bereits dargelegt. Wurde durch den Haupt-Algorithmus eine neue potentielle Verluststellung erzeugt, so soll nun nachgewiesen werden, dass diese, unabhängig von den Spielzügen des Gegenspielers, stets zum Sieg für den Computer führt.

Die Details der hierfür notwendigen Zwischenschritte wurden bereits während der Erläuterung des Haupt-Algorithmus dargelegt. Daher soll hier nun lediglich eine grobe Beschreibung erfolgen.

Dem Beweisalgorithmus wird eine Liste mit potentiell vom Gegenspieler erzeugten Zuständen übergeben. Enthält diese Liste einen Zustand, welcher nur aus “leeren” Haufen besteht und somit den Endzustand des Spiels repräsentiert, ist klar, dass dieser Endzustand vom vorherigen Spieler herbeigeführt wurde und dieser somit (nach den hier angenommenen Regeln) das Spiel gewonnen haben muss.
Wurde dieser Endzustand vom Computer herbeigeführt, so wird “True” zurückgegeben, anderenfalls “False”.

Ein Rückgabewert von “True” sagt somit aus, dass der zu Beginn übergebene und vom Computer erzeugte Zustand, bei Berücksichtigung aller möglichen nachfolgenden Spielzüge des “menschlichen” Gegenspielers, stets zum Sieg des Computers führt.

Ist dem nicht der Fall, und es ist kein Endzustand in der Liste der übergebenen Zustände enthalten, so wird, abhängig eines weiteren Arguments, welches als Indikator für den Spieler, welcher die übergebenen Zustände erzeugt hat, die Folgezustände für jeden der übergebenen Zustände erzeugt.

Ist der Computer “an der Reihe”, so wird für jeden der übergebenen Zustände lediglich ein Folgezustand mit Hilfe des Haupt-Algorithmus generiert, und die Anzahl der Folgezustände entspricht der Anzahl der übergebenen Zustände.

Ist der Mensch am Zug, so werden für jeden vom Computer erzeugten Zustand alle möglichen Folgezustände erzeugt und der Spielverlauf setzt sich in einer “quasi” Baumstruktur fort.