Technische Universität Berlin **SoSe 2023**
Fakultät II, Institut für Mathematik
Sekretariat MA 6–2, Antje Schulz
Prof. Dr. Michael Joswig
Dr. Frank Lutz, Martin Knaack, Marcel Wack

Programmierprojekt Computerorientierte Mathematik II

Abgabe: 14.07.2023 bis 17 Uhr

Änderung vom 30.06.2023

Aufgrund der Schließung des Mathematikgebäudes und der damit verbundenen Komplikationen für Sie, wird das Programmierprojekt auf diesen ersten Teil verkürzt. Für die Bearbeitung stehen Ihnen aber die gesamten 4 Wochen bis zum 14.07.2023 zur Verfügung. Da die Rechnerbetreuung nicht mehr im gewohnten Umfang stattfinden kann, möchte ich Sie dazu ermutigen, Fragen zum Projekt im Forum, an Ihren Tutor oder per Mail an *wack@math.tu-berlin.de* zu stellen. Wir wünschen ihnen trotz der Umstände viel Erfolg.

Aufgabe

Es soll ein Programm geschrieben werden, das ein (verallgemeinertes) Nim-Spiel analysiert und spielt. Ein Nimspiel ist ein Spiel für zwei Spieler, bei dem abwechselnd Steine von einem Stapel gezogen werden. Ein Spiel gilt als gewonnen, wenn eine bestimmte Siegbedingung erreicht wird. Oft wird hier der Fall angenommen, dass eine Spielerin gewinnt, sobald sie den letzten Stein gezogen hat. Für diese Aufgabe sind die genauen Regeln sowie die Siegbedingung Teil der Eingabe.

Beispiel eines Spielablaufs

Die Spieler heißen `:P1` und `:P2`. Auf dem Spielfeld befinden sich 6 Steine. Beide Spieler können beliebig oft die beiden Züge „1 Stein nehmen“ oder „2 Steine nehmen“ ausführen.
Wir betrachten nun den Ablauf eines Spiels. Spieler `:P1` beginnt und nimmt 2 Steine auf. Es bleiben 4 Steine übrig und `:P2` ist am Zug. Spielerin `:P2` nimmt nun 1 Stein auf. Es liegen noch 3 Steine auf dem Stapel. Dann entscheidet sich `:P1`, ebenfalls 1 Stein aufzunehmen, und `:P2` gewinnt das Spiel, indem sie die verbleibenden 2 Steine aufnimmt.

Die Frage ist, ob `:P1` im dem Beispiel oben den Sieg hätte erzwingen können. Um das Spiel zu analysieren und eine sichere Gewinnstrategie zu finden, kann die aus der Vorlesung bekannte *extensive Form* des Spiels verwendet werden. Ziel des Programmierprojekts ist es, durch Analyse dieser Spielbäume eine hinreichend starkes Programm zu entwickeln, das beliebige Nim-Spiele spielen kann.

Der MinMax-Algorithmus

Eine einfache Möglichkeit, diese Spielbäume zu nutzen, ist ein MinMax-Algorithmus; siehe auch p.131 im Handout zur Vorlesung am 06.06.23. Ein MinMax-Algorithmus ist ein Algorithmus, der einen Knoten im Spielbaum bewertet, indem er das Minimum bzw. das Maximum der Werte seiner Kinder berechnet. Konkret für ein Nim Spiel sagen wir, dass ein Knoten den Wert 1 hat, wenn es eine sichere Gewinnstrategie für `:P1` gibt. Analog gilt -1, wenn `:P2` eine sichere Gewinnstrategie verfolgen kann. Rekursiv bedeutet dies, dass ein Knoten, in dem `:P1` an der Reihe ist, den Maximalwert seiner Kinderknoten als Wert erhält. Da `:P1` nur Züge ausführt, die den Wert für

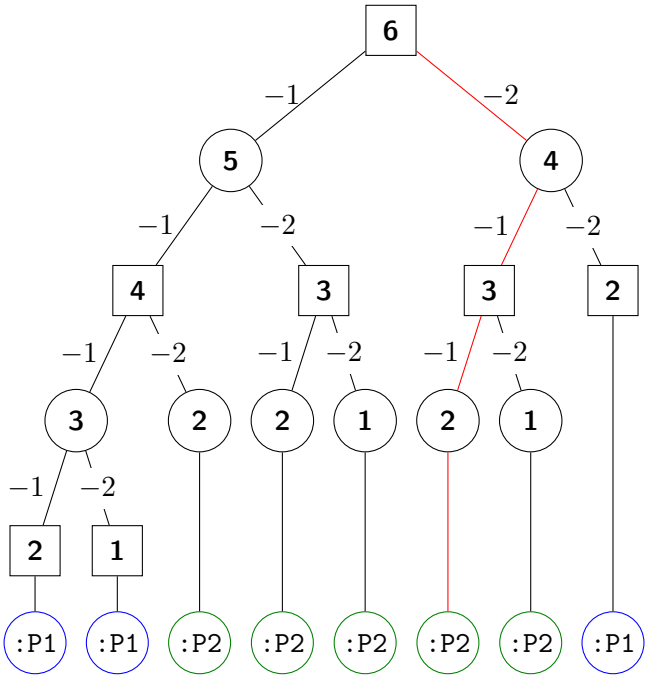

Abbildung 1: Die extensive Form eines Nim Spiels. Sieger in der letzten Reihe markiert. Der rote Pfad entspricht dem Spielablauf des Beispiels. In den runden Knoten ist `:P2` an der Reihe

ihn maximieren. Für `:P2` gilt das Gegenteil, für sie sind Positionen mit minimalem Wert optimal. Die Blätter des Spielbaums sind Knoten mit 0 verbleibenden Steinen. Ihr Wert ist 1, wenn `:P2` an der Reihe ist, und 1, wenn `:P1` an der Reihe ist. (Dies scheint vertauscht zu sein, aber null Steine für den aktiven Spieler `:P2` bedeutet, dass `:P1` den letzten Stein gezogen hat.)

```
# PseudoCode (rekursiven) MinMax-Algorithmus
# Input: Spielbaum in extensiver Form
# Output: Wert des Wurzelknotens
function MinMax(Knoten)
    if Knoten is ein Blatt then
        return value of leaf
    else
        if Knoten ist ein :max-Knoten then
            return max von MinMax(Kinder von Knoten)
        else
            return min of MinMax(Kinder von Knoten)
```

Optimierung
Da diese Bäume sehr schnell sehr groß werden gibt es zwei Optimierungsmöglichkeiten.

1. Alpha-Beta-Pruning
Beim *Alpha-Beta-Pruning* wird versucht, die Berechnung der Werte mancher Knoten des Baumes zu vermeiden. Dazu werden die jeweils ungünstigsten bekannte Wert in einer Variablen α (für `:P1`) und in β (für `:P2`) gespeichert.

```
# PseudoCode (rekursiven) AlphaBeta-Algorithmus
# Input: Spielbaum in extensiver Form
# Output: Wert des Wurzelknotens
function AlphaBeta(Knoten, alpha, beta)
    if Knoten is ein Blatt then
        return value of leaf
    else
        if Knoten ist ein :max-Knoten then
            for Kind in Kinder von Knoten do
                set value to be maximum of alpha and AlphaBeta(Kind, alpha, beta)
                if value >= beta then
                    break
                end
            end
            set alpha to be maximum of alpha and value
            return alpha
        else
            for Kind in Kinder von Knoten do
                set value to be minimum of beta and AlphaBeta(Kind, alpha, beta)
                if alpha >= value then
                    break
                end
                set beta to be minimum of beta and value
            end
            return beta
```

2. Zwischenspeichern von Knoten
In einem Spielbaum kann es vorkommen, dass mehrere Knoten den gleichen Spielstand wiedergeben und somit den gleichen Wert haben. Durch das Zwischenspeichern (Hashing per `Dict`) der berechneten Knoten und das Referenzieren auf diese Knoten werden Duplikate vermieden.

Implementierung
Wir speichern alle globalen Parameter eines Nim-Spiels in einer Struktur `GlobalState`, die unter anderem `gameStates` enthält, das die Menge aller vom Wurzelknoten aus erreichbaren Zustände speichert, um, wie oben erwähnt, Duplikate im Spielbaum zu vermeiden. Konkret werden folgende Eigenschaften benötigt

- `players::Vector{Symbol}` Die Spieler, die am Spiel teilnehmen. Für uns `[:P1,:P2]`. Das erste Element dieses Vektors ist der Startspieler.
- `moves::Vector{Dict{Int,Real}}` Die möglichen Züge der Spieler. Der Schlüssel ist die Anzahl der wegzunehmenden Steine und der Wert, wie oft dieser Zug ausgeführt werden kann.
- `winCondition::Function` Eine Funktion, die entscheidet, ob ein Spielzustand gewonnen ist. Muss als Eingabe ein `struct` vom Typ `Nim` erhalten.
- `root::Nim` Die Wurzel des Spielbaums.
- `gameStates::Dict{Nim,Real}` Eine `Dict` aller möglichen Boardstates im Spielbaum mit ihrem Wert als value.

Der eigentliche Boardstate und Knoten im Spielbaum wird als Objekt vom Typ `Nim` wie folgt spezifiziert

- `active::Symbol` Der Spieler am Zug.
- `board::Int` Die Anzahl der Steine, die noch im Spiel sind.
- `moves::Vector{Dict{Int,Real}}` Die zu diesem Zeitpunkt noch möglichen Züge der Spieler.
- `children::Dict{Int,Nim}` Die verschiedenen Zustände, die durch einen Zug von der aktuellen Position aus erreicht werden können.

Auch wenn die Spezifikation hier allgemein gehalten ist, kann davon ausgegangen werden, dass immer zwei Spieler am Spiel teilnehmen. Unterschiedliche Siegbedingungen sollten jedoch erlaubt sein. Der berechnete Wert eines Spiels liegt in $[-1, 1] \subset \mathbb{R}$. Wobei der Wert -1 ist, wenn `:P2` sicher gewinnt, und 1, wenn `:P1` gewinnt. Wir sagen, dass das Spiel unentschieden ist, wenn einer der beiden Spieler keine Züge mehr hat. Das Spiel hat dann den Wert 0. Implementieren Sie sowohl den einfachen MinMax-Algorithmus als auch den gleichen Algorithmus mit Min-Max-Pruning auf dem Spielbaum eines Nim-Spiels, um den Wert des Wurzelknotens zu bestimmen. Konkret benötigen wir folgende Funktionen

- `GlobalState(players,moves,winCondition,board)::GlobalState` Ein Konstruktor für den Typ `GlobalState`. Dieser erhält zusätzlich die Variable `board` für die Konstruktion des Wurzelknotens.
- `minMax(gState::GlobalState, state::Nim)::Real` Verwenden Sie den Min-Max-Algorithmus, um den Wert des Zustands zu berechnen.
- `alphaBeta(gState:GlobalState, state::Nim)::Real` Verwendet den Min-Max-Algorithmus mit Alpha Beta Pruning, um den Wert des Zustands zu berechnen.

Implementieren Sie die beiden Typen `GlobalState` und `Nim` sowie die zwei Funktionen in einer `Julia` Datei.

Abgabe Bitte präsentieren Sie Ihr Programm bis zum 14.07.2023 bei Ihren Tutor. Darüber hinaus geben sie ihr Programm bis zum selben Datum über den `comajudge` ab, der zugehörige Aufgabenname ist `co2_Project`.

Beispielaufrufe

```
players = [:P1,:P2];
moves = [Dict{Int,Real}(1=>Inf,2=>Inf),Dict{Int,Real}(1=>Inf,2=>Inf)];
winCondition = (game::Nim)->game.board <= 0;
board = 6;

game = GlobalState(players,moves,winCondition,board);

# Beispiel
alphaBeta(game,game.root)
> -1
```

Hinweise

Zum Vergleich von Objekten des Typs `Nim` wird die Funktion `isequal` verwendet. Diese muss aus technischen Gründen überschrieben werden. Um genau zu sein, müssen Sie `isequal` so definieren, dass zwei Elemente vom Typ `Nim` genau dann gleich sind, wenn die drei `field`-Variablen `board`, `active` und `moves` gleich sind. Wir geben ihnen die zugehörige `hash`-Funktion an.

```
Base.isequal(a::Nim,b::Nim) =  #Ihr Code hier..
Base.hash(a::Nim,h::UInt) =  hash((a.board,a.active,a.moves),h)
```