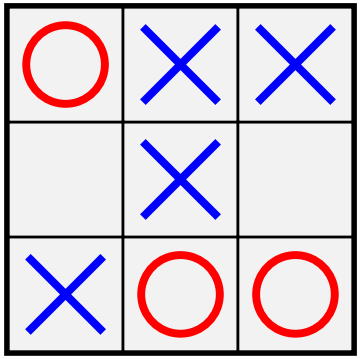
17 Tic-Tac-Toe

Regeln:

- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

Beispiel



Analyse

. . . offenbar hat die anziehende Partei gewonnen.

Fragen

- Ist das immer so? D.h. kann die anziehende Partei immer gewinnen?
- Wie implementiert man ein Tic-Tac-Toe-Programm, das
  - ...möglichst oft gewinnt?
  - ...eine ansprechende Oberfläche bietet?

Hintergrund — Zwei-Personen-Nullsummenspiele

Tic-Tac-Toe ist ein endliches Zwei-Personen-Nullsummen-Spiel, mit perfekter Information. Das heißt:

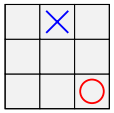
- Zwei Personen spielen gegeneinander.
- Was der eine gewinnt, verliert der andere.
- Es gibt eine endliche Menge von Spiel-Konfigurationen.
- Die Spieler ziehen abwechselnd. Ein Zug wechselt die Konfiguration, bis eine Endkonfiguration erreicht ist.
- Jede Endkonfiguration ist mit einem Gewinn aus $\mathbb{R}$ bewertet.
- Person 0 gewinnt, wenn Endkonfiguration mit negativem Gewinn erreicht wird; sonst gewinnt Person 1.

Man spricht häufig auch von einem Minimum-Spieler (Person 0) und einem Maximum-Spieler (Person 1).

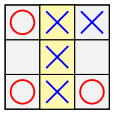
Perfekte Information bedeutet, dass die Spieler alle Informationen besitzen und demzufolge das Spiel (im Prinzip) vollständig berechenbar ist (wie z.B. Mühle, Dame, Schach, Go, etc). Ein Nullsummenspiel ohne vollständige Information ist z.B. Poker.

...im Beispiel

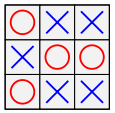
Konfiguration:



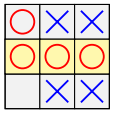
Endkonfigurationen:



Gewinn -1



Gewinn 0

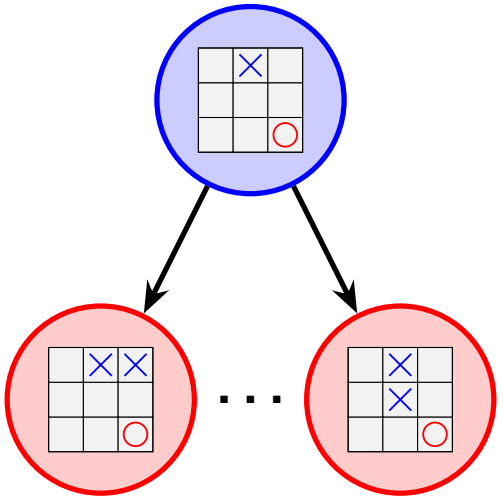


Gewinn +1

...im Beispiel

Der Minimum-Spieler (cross/blau), wählt eine der möglichen Nachfolgekonfigurationen.

Spielzug:



Spielbaum

Ein Spielbaum wird folgendermassen (rekursiv) konstruiert:

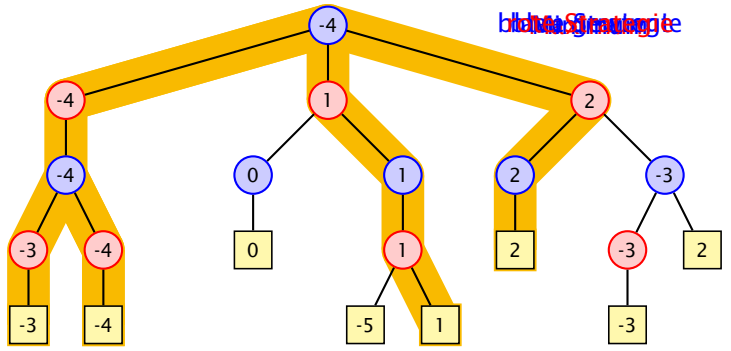
- gegeben ein Knoten v, der eine Spielkonfiguration repräsentiert
- für jede mögliche Nachfolgekonfiguration erzeugen wir einen Kindknoten, den wir mit v verbinden;
- dann starten wir den Prozess rekursiv für alle Kindknoten.

Eigenschaften:

- jeder Knoten repräsentiert eine Konfiguration; allerdings kann dieselbe Konfiguration sehr oft vorkommen
- Blattknoten repräsentieren Endkonfigurationen
- Kanten repräsentieren Spielzüge
- jedes Spiel ist ein Pfad von der Wurzel zu einem Blatt

Beispiel — Spielbaum

Dieser Spielbaum repräsentiert ein beliebiges Zwei-Personen-Nullsummenspiel. Deshalb sind die Bewertungen in den Blättern nicht nur $\{-1, 0, 1\}$.



Spielbaum

Fragen:

- Wie finden wir uns (z.B. als blaue Person) im Spielbaum zurecht?
- Was müssen wir tun, um sicher ein negatives Blatt zu erreichen?

Spielbaum

Der Spielbaum wird üblicherweise so konstruiert, dass die Wurzel der aktuellen Stellung entspricht, in der wir am Zug sind.

Idee:

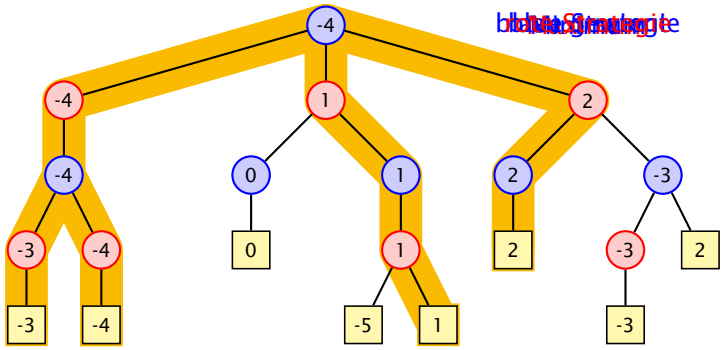
- Wir ermitteln für jede Konfiguration den jeweils besten zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

Fall 1 Die Konfiguration ist blau: wir sind am Zug. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

Fall 2 Die Konfiguration ist rot: der Gegner ist am Zug. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.

Beispiel — Spielbaum

Dieser Spielbaum repräsentiert ein beliebiges Zwei-Personen-Nullsummenspiel. Deshalb sind die Bewertungen in den Blättern nicht nur $\{-1, 0, 1\}$.



Strategien

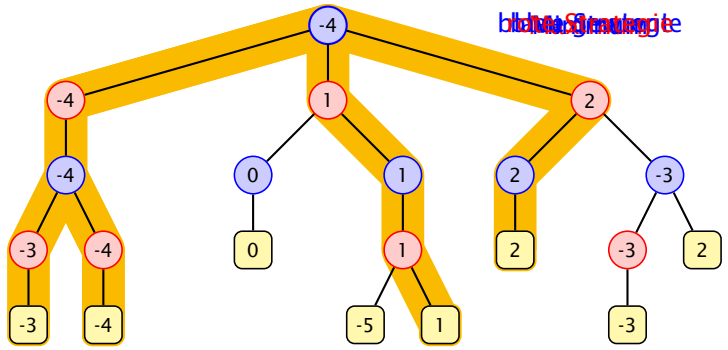
Eine Strategie ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

Eine optimale Strategie ist eine, deren Anwendung garantiert zu einer Endkonfiguration führt, deren Wert mindestens so groß ist wie der berechnete garantierte Gewinn.

Eine akzeptable Strategie ist eine, deren Anwendung einen Verlust des Spiels verhindert, wann immer das möglich ist ...

Beispiel — Spielbaum

Dieser Spielbaum repräsentiert ein beliebiges Zwei-Personen-Nullsummenspiel. Deshalb sind die Bewertungen in den Blättern nicht nur $\{-1, 0, 1\}$.



Struktur

GameTreeNode
- pos : Position
- type : int
- value : int
- child : GameTreeNode[]
+ GameTreeNode (Position p)
+ getBestMove () : int

Position
- playerToMove : int
- arena : int[]
+ Position ()
+ Position (Position p)
+ won () : int
+ getMoves () : Iterator<Integer>
+ makeMove (int m)
+ movePossible (int m) : boolean
+ getPlToMv () : int

Game
- g : GameTreeNode
- p : Position
- view : View
+ makeBestMove ()
+ makePlayerMove ()
+ movePossible () : boolean
+ finished () : boolean

Das Attribut `view` enthält eine Referenz auf ein Objekt, dass für die Visualisierung sorgt. Darauf werden wir später ausführlich eingehen. Für ein Verständnis der Spielmodellierung kann man die Aufrufe von `view`-Methoden ignorieren.

Die hier aufgeführten Methoden gehören zum Interface `Model` (mehr dazu im Abschnitt über die GUI-Implementierung).

Inner Class

```
1 public class OuterClass {
2     private int var;
3     public class InnerClass {
4         void methodA() {};
5     }
6     public void methodB() {};
7 }
```

- Instanz von `InnerClass` kann auf alle Member von OuterClass zugreifen.
- Wenn `InnerClass` `static` deklariert wird, kann man nur auf statische Member zugreifen.
- Statische innere Klassen sind im Prinzip normale Klassen mit zusätzlichen Zugriffsrechten.
- Nichtstatische innere Klassen sind immer an eine konkrete Instanz der äußeren Klasse gebunden.

Beispiel – Zugriff von Außen

Um von außen ein Objekt der inneren Klasse zu erzeugen, muss man erst ein Objekt der äußeren Klasse generieren.
Dann erzeugt man ein Objekt der Klasse z.B. durch `objOuter.new InnerClass()`, wobei wir hier annehmen, dass `InnerClass` einen Defaultkonstruktor hat.

```
1  class OuterClass {
2      private int x = 1;
3      public class InnerClass {
4          void show() {
5              System.out.println("x = " + x);
6      }   }
7      public void showMeth() {
8          InnerClass b = new InnerClass();
9          b.show();
10 }   }
11 public class TestInner {
12     public static void main(String args[]) {
13         OuterClass a = new OuterClass();
14         OuterClass.InnerClass x = a.new InnerClass();
15         x.show();
16         a.showMeth();
17 }   }
```

"TestInner.java"

Beispiel – Zugriff von Äußerer Klasse

Normalerweise erzeugt man keine Objekte einer inneren Klasse von außen. Stattdessen bietet häufig die äußere Klasse eine Funktion, die ein Objekt der inneren Klasse zur Verfügung stellt.

```
1  class OuterClass {
2      private static int x = 1;
3      public static class InnerClass {
4          void show() {
5              System.out.println("x = " + x);
6      }   }
7      public void showMeth() {
8          InnerClass b = new InnerClass();
9          b.show();
10 }   }
11 public class TestInnerStatic {
12     public static void main(String args[]) {
13         OuterClass a = new OuterClass();
14         OuterClass.InnerClass x =
15                   new OuterClass.InnerClass();
16         x.show(); a.showMeth();
17 }   }
```

Häufig (vgl. verkettete Liste mit innerer Klasse `ListElem`) werden innere Klassen aber auch nur zur Datenkapselung eingesetzt und sind dann privat.

"TestInnerStatic.java"

Eine Anwendung von öffentlichen, inneren Klassen sind z.B. Adapterklassen (vgl. `Iterator`).

Local Inner Class

Eine lokale, innere Klasse wird innerhalb einer Methode deklariert:

```
1 public class OuterClass {
2     private int var;
3     public void methodA() {
4         class InnerClass {
5             void methodB() {};
6         }
7     }
8 }
```

- Kann zusätzlich auf die finalen Parameter und Variablen der Methode zugreifen.

Man kann nicht von außen auf die Klasse zugreifen. Deshalb machen modifier wie `private`, `public` keinen Sinn und sind nicht erlaubt.

Beispiel – Iterator

```
1 interface Iterator<T> {
2     boolean hasNext();
3     T next();
4     void remove(); // optional
5 }
```

- Ein Iterator erlaubt es über die Elemente einer Kollektion zu iterieren.
- Abstrahiert von der Implementierung der Kollektion.
- `hasNext()` testet, ob noch ein Element verfügbar ist.
- `next()` liefert das nächste Element (falls keins verfügbar ist wird eine `NoSuchElementException` geworfen).
- `remove()` entfernt das zuletzt über `next()` zugegriffene Element aus der Kollektion.

Falls die Kollektion das Entfernen von Elementen nicht erlaubt, bleibt `remove()` unimplementiert und liefert bei Aufruf eine Exception.

Beispiel – Iterator

`curr` zeigt auf das Element, das beim letzten Aufruf von `next()` zurückgegeben wurde.

```
1  public class TestIterator implements Iterable<Integer> {
2      Integer[] arr;
3      TestIterator(int n) {
4          arr = new Integer[n];
5      }
6      public Iterator<Integer> iterator() {
7          class MyIterator implements Iterator<Integer> {
8              int curr = arr.length;
9              public boolean hasNext() { return curr>0;}
10             public Integer next() {
11                 if (curr == 0)
12                     throw new NoSuchElementException();
13                 return arr[--curr];
14             }
15         }
16         return new MyIterator();
17     }
```

Das Interface `Iterable<Integer>` „verspricht“ die Implementierung der Funktion `Iterator<Integer> iterator()`.

"TestIterator.java"

Beispiel – Iterator

Anwendung des Iterators:

```
18      public static void main(String args[]) {
19          TestIterator t = new TestIterator(10);
20          for (Iterator<Integer> iter = t.iterator();
21               iter.hasNext();) {
22              Integer i = iter.next();
23              System.out.println(i);
24          }
25          for (Integer j : t) {
26              System.out.println(j);
27          }
28      }
29  }
```

"TestIterator.java"

Erweiterte Syntax der For-Schleife. Da `t` das Interface `Iterable<Integer>` implementiert, kann man mit dieser Syntax über die Elemente der Kollektion itterieren.

In diesem Fall wird nur 20 mal `null` ausgegeben...

Implementierung – SpielbaumA

`nodeCount` gehört nicht zur Spiellogik sondern zählt nur die Anzahl der Knoten im Spielbaum.

```
1  import java.util.*;
2  public class GameTreeNode implements PlayConstants {
3      static public int nodeCount = 0;
4
5      private int value;
6      private int type;
7      private int bestMove = -1;
8      private Position pos;
9      private GameTreeNode[] child = new GameTreeNode[9];
10
11     public int getBestMove() {
12         return bestMove;
13     }
```

"GameTreeNodeA.java"

- das interface `PlayConstants` definiert die Konstanten `MIN` = -1, `NONE` = 0, `DRAW` = 0, `EMPTY` = 0, `MAX` = 1;

Implementierung – SpielbaumA

```
14      public GameTreeNode(Position p) { nodeCount++;
15          pos = p; type = p.getPlToMv();
16          // hab ich schon verloren?
17          if (p.won() != NONE) { value = p.won(); return; }
18          // no more moves --> no winner
19          Iterator<Integer> moves = p.getMoves();
20          if (!moves.hasNext()) { value = DRAW; return; }
21          value = -2*type;
22          while (moves.hasNext()) {
23              int m = moves.next();
24              child[m] = new GameTreeNode(p.makeMove(m));
25              if (type == MIN && child[m].value < value ||
26                  type == MAX && child[m].value > value) {
27                  value = child[m].value;
28                  bestMove = m;
29  }   }   }   }
```

"GameTreeNodeA.java"

Für den Maximumspieler (`type==1`) verhält sich `-2*type` wie $-\infty$ so dass die nachfolgende `while`-Schleife ein Maximum berechnet. Analog für den Minimumspieler.

Implementierung – SpielbaumA

Die einzigen TicTacToe-spezifischen Informationen in der Klasse `GameTreeNode` sind

- die Größe des Arrays `child`; wir wissen, dass wir höchstens 9 Züge machen können
- wir kennen die Gewinnwerte:

  `MIN` gewinnt : `value` = -1
  unentschieden : `value` = 0
  `MAX` gewinnt : `value` = +1

  deswegen könne wir z.B. `value` mit `-2*type` initialisieren.

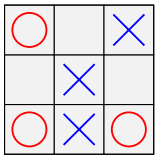
Die anderen Regeln werden in die Klasse `Position` ausgelagert.

Klasse Position – Kodierung

Das Array arena enthält die Spielstellung z.B.:
arena = {1,0,-1,0,-1,0,1,-1,1} bedeutet:

0	1	2
3	4	5
6	7	8

Koordinaten



Konfiguration

1	0	-1
0	-1	0
1	-1	1

Kodierung

Implementierung – Position

```
1  public class Position implements PlayConstants {
2      private int[] arena;
3      private int playerToMove = MIN;
4      public Position() { arena = new int[9]; }
5      public Position(Position p) {
6          arena = (int[]) p.arena.clone();
7          playerToMove = p.playerToMove;
8      }
9      public Position makeMove(int place) {
10         Position p = new Position(this);
11         p.arena[place] = playerToMove;
12         p.playerToMove = -playerToMove;
13         return p;
14     }
15     private boolean free(int place) {
16         return (arena[place] == EMPTY);
17     }
18     public boolean movePossible(int pl) {
19         return (getMoves().hasNext() && free(pl));
20     }
```

Implementierung – Position

Die Methoden String toString() und int won() sind hier nicht gezeigt...

```
21 private class PossibleMoves implements Iterator<Integer> {
22     private int nxt = 0;
23     public boolean hasNext() {
24         if (won() != NONE) return false;
25         for (; nxt<9; nxt++)
26             if (free(nxt)) return true;
27         return false;
28     }
29     public Integer next() {
30         if (!hasNext())
31             throw new NoSuchElementException();
32         return nxt++;
33 }   }
34 public Iterator<Integer> getMoves() {
35     return new PossibleMoves();
36 }
```

"Position.java"

Klasse Game

Die Klasse Game sammelt notwendige Datenstrukturen und Methoden zur Durchführung des Spiels:

```
1  public class Game implements PlayConstants, Model {
2      private Position     p;
3      private GameTreeNode g;
4      private View view;
5
6      Game(View v) {
7          view = v;
8          p = new Position();
9      }
10     private void initTree() {
11         g.nodeCount = 0;
12         g = new GameTreeNode(p);
13         System.out.println("generate tree... (" +
14                            g.nodeCount + " nodes)");
15     }
```

"Game.java"

Klasse Game

```
17      private void makeMove(int place) {
18          view.put(place,p.getPlToMv());
19          p = p.makeMove(place);
20          if (finished())
21              view.showWinner(p.won());
22      }
23      public void makeBestMove() {
24          initTree();
25          makeMove(g.getBestMove());
26      }
27      public void makePlayerMove(int place) {
28          makeMove(place);
29          if (!finished()) {
30              makeBestMove();
31          }
32      }
```

"Game.java"

Klasse Game

```
33      public boolean movePossible(int place) {
34          return p.movePossible(place);
35      }
36      public boolean finished() {
37          return !p.getMoves().hasNext();
38      }
39      public static void main(String[] args) {
40          Game game = new Game(new DummyView());
41          while (!game.finished()) {
42              game.makeBestMove();
43              System.out.println(game.p);
44          }
45  }   }
```

"Game.java"

Output – Variante A

```
generate tree... (549946 nodes)
x..
...
...
generate tree... (59705 nodes)
x..
.o.
...
generate tree... (7332 nodes)
xx.
.o.
...
generate tree... (935 nodes)
xxo
.o.
...
generate tree... (198 nodes)
xxo
.o.
x..
generate tree... (47 nodes)
xxo
oo.
x..
generate tree... (14 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```

Wenn in jedem Spiel genau 9 Züge gemacht würden, dann hätte der Baum $9! \cdot (\frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{9!}) = 623\,530$ Knoten. Da manche Spiele aber früher beendet sind kommt man auf einen geringeren Wert.

Effizienz

Wie können wir das effizienter gestalten?

1. Den Spielbaum nur einmal berechnen, anstatt jedesmal neu.
   gewinnt nicht sehr viel...
2. Wenn wir z.B. als MaxPlayer schon einen Wert von 1 erreicht haben, brauchen wir nicht weiterzusuchen...

   Spielbaum ist dann unvollständig; Wiederverwendung schwierig...

$\Longrightarrow$ Baue keinen vollständigen Spielbaum; nur Wert und Zug an der Wurzel müssen korrekt sein.

Implementierung – SpielbaumB

Nur Zeile 30 wurde eingefügt.

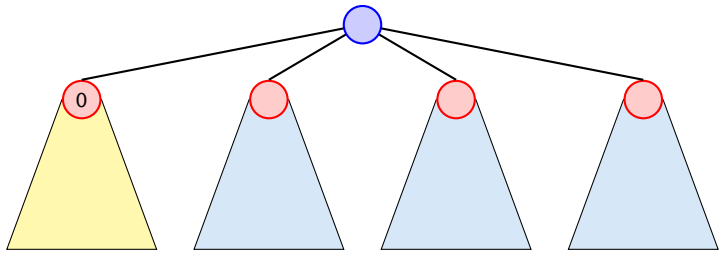
```
14      public GameTreeNode(Position p) { nodeCount++;
15          pos = p; type = p.getPlToMv();
16          // hab ich schon verloren?
17          if (p.won() != NONE) { value = p.won(); return; }
18          // no more moves --> no winner
19          Iterator<Integer> moves = p.getMoves();
20          if (!moves.hasNext()) { value = DRAW; return; }
21          value = -2*type;
22          while (moves.hasNext()) {
23              int m = moves.next();
24              child[m] = new GameTreeNode(p.makeMove(m));
25              if (type == MIN && child[m].value < value ||
26                  type == MAX && child[m].value > value) {
27                  value = child[m].value;
28                  bestMove = m;
29                  // we won; don't search further
30                  if (value == type) return;
31  }   }   }   }
```

"GameTreeNodeB.java"

Output – Variante B

```
generate tree... (94978 nodes)
x..
...
...
generate tree... (3763 nodes)
x..
.o.
...
generate tree... (1924 nodes)
xx.
.o.
...
generate tree... (61 nodes)
xxo
.o.
...
generate tree... (50 nodes)
xxo
.o.
x..
generate tree... (17 nodes)
xxo
oo.
x..
generate tree... (10 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```


Effizienz – Alpha-Beta-Pruning



Ein Wert > 0 innerhalb der blauen Teilbäume kann nicht zu Wurzel gelangen (Wurzel ist MIN-Knoten). Deshalb kann ein MAX-Knoten innerhalb dieser Bäume abbrechen, wenn er einen Wert ≥ 0 erzielt hat.

Analog für MIN.

Einige Werte im Spielbaum sind dann nicht korrekt; aber das wirkt sich nicht auf den Wert an der Wurzel aus. Man muss dafür sorgen, dass bestMove an der Wurzel nicht auf einen Knoten mit einem inkorrekten Wert zeigt. Wenn z.B. zwei Knoten mit Wert 0 existieren (ein echter, und einer dessen wirklicher Wert größer sein könnte, muss bestMove auf den echten zeigen).

Implementierung – SpielbaumC

Änderungen am Konstruktor:

```
1  private GameTreeNode(Position p,
2                       int goalMin, int goalMax) {
3     nodeCount++;
4     pos = p; type = p.getPlToMv();
5     if (p.won() != NONE) { value = p.won(); return; }
6     Iterator<Integer> moves = p.getMoves();
7     if (!moves.hasNext()) { value = DRAW; return; }
8
9     value = -2*type;
10    while (moves.hasNext()) {
11       int m = moves.next();
12       child[m] = new GameTreeNode(p.makeMove(m),
13                                   goalMin,goalMax);
14 // continued...
```

"GameTreeNodeC.java"

Implementierung – SpielbaumC

Zeilen 24/25 können auch durch `if (goalMin>=goalMax) return;` ersetzt werden.

```
15      if (type == MIN && child[m].value < value ||
16          type == MAX && child[m].value > value) {
17        value = child[m].value;
18        bestMove = m;
19
20        // update goals
21        if (type == MIN && goalMax > value) goalMax=value;
22        if (type == MAX && goalMin < value) goalMin=value;
23        // leave if goal is reached
24        if (type == MIN && value <= goalMin) return;
25        if (type == MAX && value >= goalMax) return;
26  }  }} // if, while, Konstruktor
27  public GameTreeNode(Position p) {
28      this(p, MIN, MAX);
29  }
```

Wenn wir Zeilen 21/22 löschen gilt immer `goalMin == MIN` und `goalMax == MAX`. Dann ist dies das Gleiche wie Variante B.

"GameTreeNodeC.java"

In Zeile 15, 16 ist es wichtig, dass wir `bestMove` nur ändern, wenn der neue Wert strikt besser als der alte ist. Es kann sein, dass der Wert von `child[m].value` nicht korrekt ist (z.B. zu klein wenn wir minimieren). Dann wäre eine Auswahl dieses Nachfolgers als besten Zug schlecht.

Output – Variante C

```
generate tree... (16811 nodes)
x..
...
...
generate tree... (1903 nodes)
x..
.o.
...
generate tree... (728 nodes)
xx.
.o.
...
generate tree... (61 nodes)
xxo
.o.
...
generate tree... (50 nodes)
xxo
.o.
x..
```

```
generate tree... (17 nodes)
xxo
oo.
x..
generate tree... (10 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```


Effizienz

Bis jetzt haben wir bei den Effizienzsteigerungen das eigentliche Spiel ignoriert.

- Wenn wir einen Zug haben, der sofort gewinnt, kennen wir den Wert des Knotens und den besten Zug.
- Falls das nicht zutrifft, aber der Gegner am Zug einen sofortigen Gewinn hätte, dann ist der beste Zug dieses zu verhindern. D.h. wir kennen den besten Zug, aber noch nicht den Wert des Knotens.

`int forcedWin(int player)` in der Klasse `Position` überprüft, ob `player` einen Zug mit sofortigem Gewinn hat.

- falls ja, gibt es diesen Zug zurück
- sonst gibt es `-1` zurück

Implementierung – SpielbaumD

```
1  private GameTreeNode(Position p,
2                       int goalMin, int goalMax) {
3    nodeCount++; pos = p; type = p.getPlToMv();
4    if (p.won() != NONE) { value = p.won(); return; }
5    Iterator<Integer> moves = p.getMoves();
6    if (!moves.hasNext()) { value = DRAW; return; }
7    int m;
8    if ((m=p.forcedWin(type)) != -1) {
9        bestMove = m;
10       value = type;
11       return;
12   }
13   if ((m=p.forcedWin(-type)) != -1) {
14       bestMove = m;
15       child[m] = new GameTreeNode(p.makeMove(m),
16                                   goalMin, goalMax);
17       value = child[m].value;
18       return;
19   }
```

"GameTreeNodeD.java"

Implementierung – SpielbaumD

```
20 value = -2*type;
21 while (moves.hasNext()) {
22    m = moves.next();
23    child[m] = new GameTreeNode(p.makeMove(m),
24                                goalMin,goalMax);
25
26    if (type == MIN && child[m].value < value ||
27        type == MAX && child[m].value > value) {
28        value = child[m].value;
29        bestMove = m;
30
31        // update goals
32        if (type == MIN && goalMax > value) goalMax=value;
33        if (type == MAX && goalMin < value) goalMin=value;
34        // leave if goal is reached
35        if (goalMin >= goalMax) return;
36
37 }  } // if, while
```

"GameTreeNodeD.java"

Output – Variante D

```
generate tree... (2738 nodes)
x..
...
...
generate tree... (271 nodes)
x..
.o.
...
generate tree... (106 nodes)
xx.
.o.
...
generate tree... (9 nodes)
xxo
.o.
...
generate tree... (8 nodes)
xxo
.o.
x..
generate tree... (7 nodes)
xxo
oo.
x..
generate tree... (6 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```


Effizienz

Was könnte man noch tun?

- Eröffnungen; für die initialen Konfigurationen den besten Antwortzug speichern.
- Ausnutzen von Zugumstellungen. Überprüfen ob man die aktuelle Stellung schon irgendwo im Spielbaum gesehen hat (Hashtabelle).
- Ausnutzen von Symmetrien.

Aber für Tic-Tac-Toe wäre dieses wohl overkill...

Für komplexe Spiele wie Schach oder Go ist eine exakte Auswertung des Spielbaums völlig illusorisch...