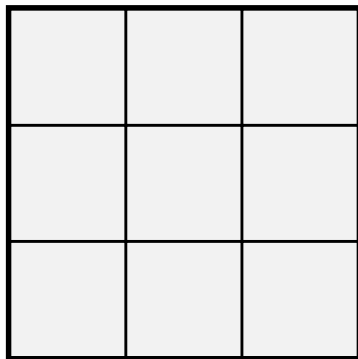


11 Tic-Tac-Toe

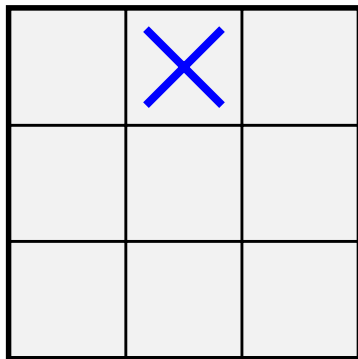
Regeln:

- ▶ 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.

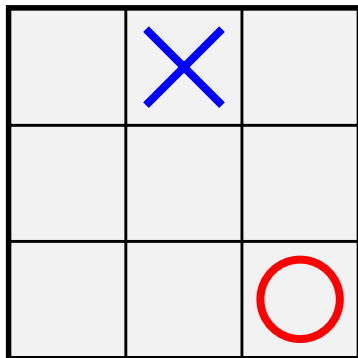
Beispiel



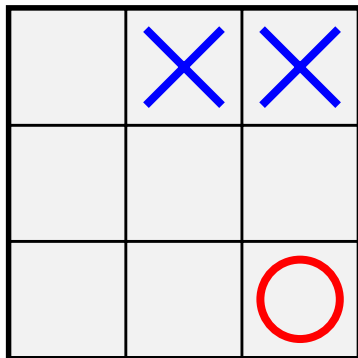
Beispiel



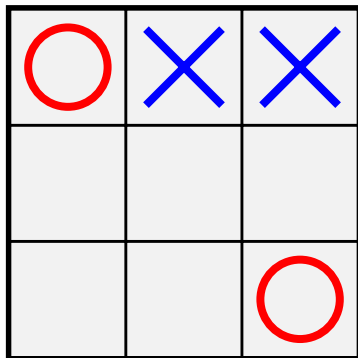
Beispiel



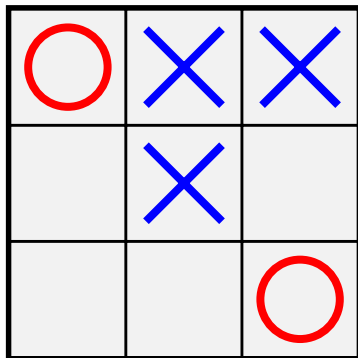
Beispiel



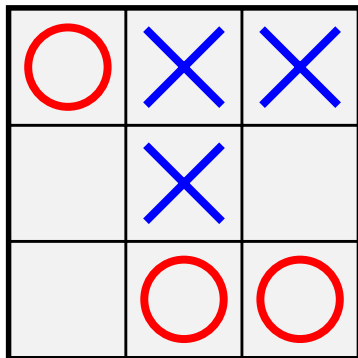
Beispiel



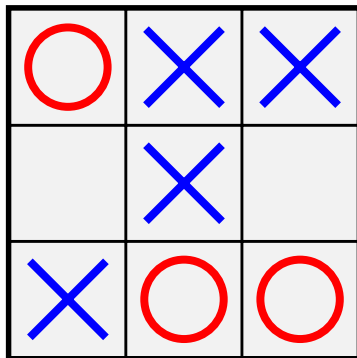
Beispiel



Beispiel



Beispiel



... 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:

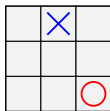
- ▶ Zwei Personen spielen gegeneinander.
- ▶ Was der eine gewinnt, verliert der andere.
- ▶ Es gibt eine endliche Menge von **Spiel-Konfigurationen**.
- ▶ Die Spieler ziehen abwechselnd. Ein **Zug** wechselt die Konfiguration, bis eine **Endkonfiguration** erreicht ist.
- ▶ Jede Endkonfiguration ist 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).

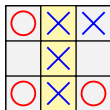
Perfekte Information bedeutet, dass die Spieler alle Informationen besitzen und demzufolge das Spiel (im Prinzip) vollständig berechenbar ist (wie z.B. Mühle, Dame, Schach, Go, etc). Ein Nullsummenspiel ohne vollständige Information ist z.B. Poker.

...im Beispiel

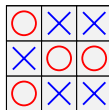
Konfiguration:



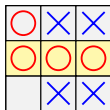
Endkonfigurationen:



Gewinn -1



Gewinn 0

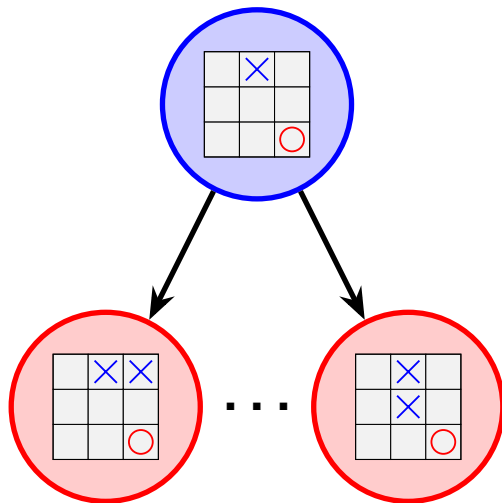


Gewinn +1

...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 Nachfolgekongfiguration 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

Fragen:

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

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

Idee:

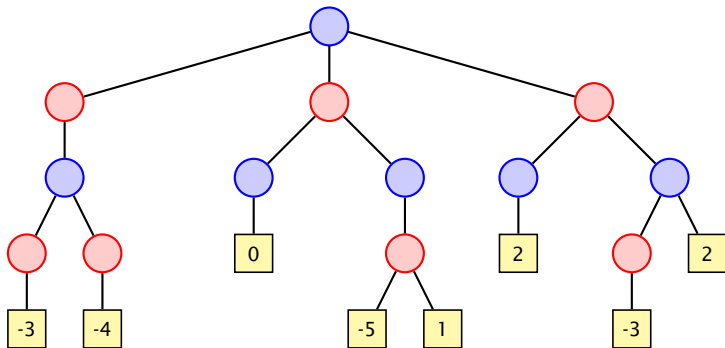
- ▶ Wir ermitteln für jede Konfiguration den jeweils **besten** zu erzielenden Gewinn.
- ▶ Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

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.

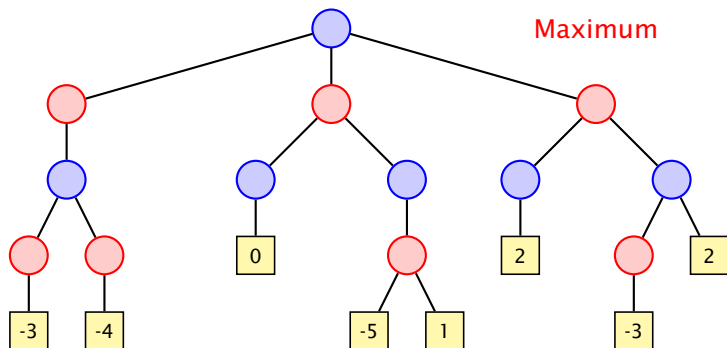
Beispiel — Spielbaum

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



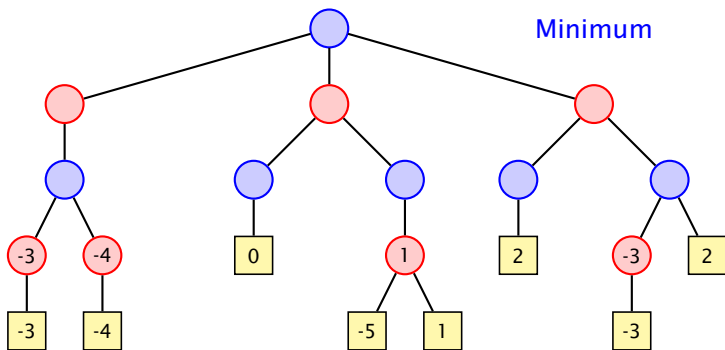
Beispiel — Spielbaum

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



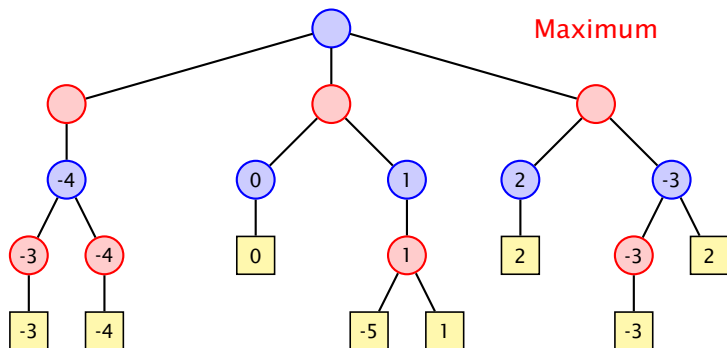
Beispiel — Spielbaum

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



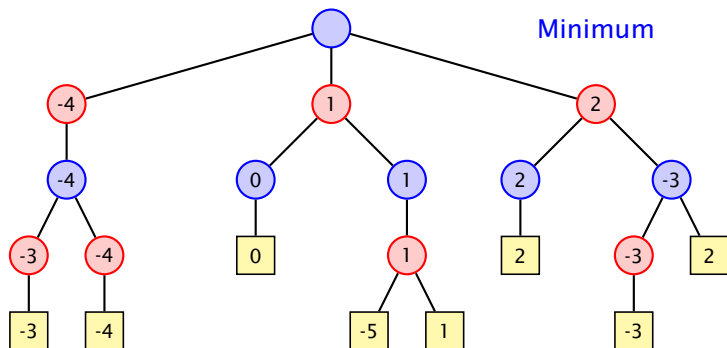
Beispiel — Spielbaum

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



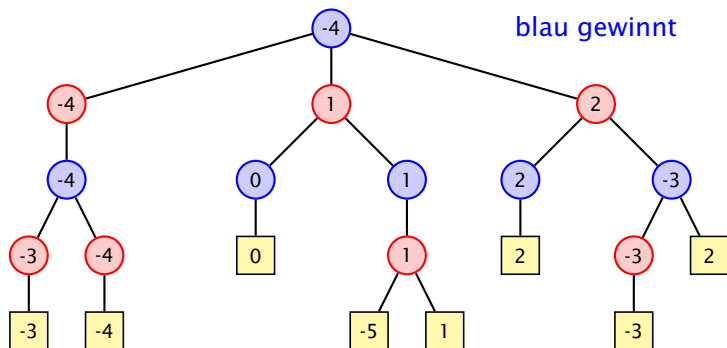
Beispiel — Spielbaum

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



Beispiel — Spielbaum

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



Strategien

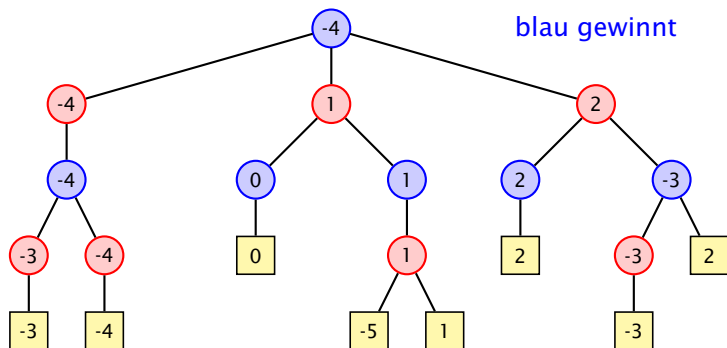
Eine **Strategie** ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

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 ...

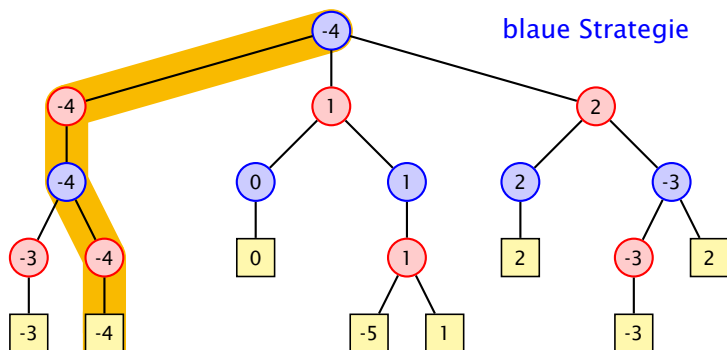
Beispiel — Spielbaum

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



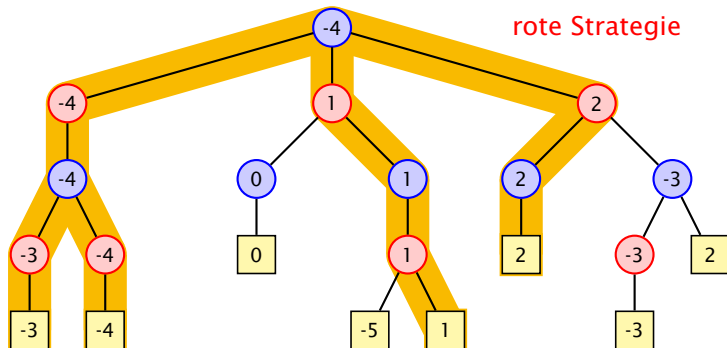
Beispiel — Spielbaum

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



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)
+ bestMove	() : int

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

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

Implementierung – SpielbaumA

```
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 bestMove() {
12         return bestMove;
13     }
```

"GameTreeNodeA.java"

- ▶ das interface **PlayConstants** definiert die Konstanten **MIN = -1**, **NONE = 0**, **MAX = 1**;

Implementierung - SpielbaumA

```
14     public GameTreeNode(Position p) { nodeCount++;
15         pos = p; type = p.playerToMove;
16         // hab ich schon verloren?
17         if (p.won() == -type) { value = -type; return; }
18         // no more moves --> no winner
19         Iterator<Integer> moves = p.getMoves();
20         if (!moves.hasNext()) { value = NONE; 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"

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] == NONE);
17    }
18    public boolean movePossible(int pl) {
19        return (getMoves().hasNext() && free(pl));
20    }
```

Implementierung - Position

```
21 private class PossibleMoves implements Iterator<Integer> {
22     private int next = 0;
23     public boolean hasNext() {
24         if (won() != 0) return false;
25         for (; next<9; next++)
26             if (free(next)) return true;
27         return false;
28     }
29     public Integer next() {
30         if (won() == 0)
31             for(; next<9; next++)
32                 if (free(next)) return next++;
33         throw new NoSuchElementException();
34     } }
35 public Iterator<Integer> getMoves() {
36     return new PossibleMoves();
37 }
```

"Position.java"

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

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.getP1ToMv());
19         p = p.makeMove(place);
20         if (finished())
21             view.showWinner(p.won());
22     }
23     public void makeBestMove() {
24         initTree();
25         makeMove(g.bestMove());
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         for (int i = 0; i < 9; ++i) {
42             if (!game.finished()) {
43                 game.makeBestMove();
44                 System.out.println(game.p);
45             }
46             else System.out.println("no more moves");
47 } } }
```

"Game.java"

Output – Variante A

Der Baum hat natürlich weniger als $9!$
 $(\frac{1}{1!} + \frac{1}{2!} + \dots + \frac{1}{9!}) = 623\,530$ Knoten.

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

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...

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

Implementierung - SpielbaumB

```
14 public GameTreeNode(Position p) { nodeCount++;
15     pos = p; type = p.playerToMove;
16     // hab ich schon verloren?
17     if (p.won() == -type) { value = -type; return; }
18     // no more moves --> no winner
19     Iterator<Integer> moves = p.getMoves();
20     if (!moves.hasNext()) { value = NONE; 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"

Nur Zeile 30 wurde eingefügt.

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.

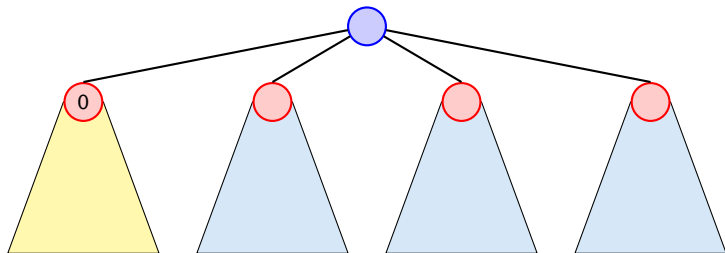
generate tree... (2 nodes)

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.playerToMove;  
5     if (p.won() == -type) { value = -type; return; }  
6     Iterator<Integer> moves = p.getMoves();  
7     if (!moves.hasNext()) { value = NONE; 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    }  
14 // continued...
```

"GameTreeNodeC.java"

Implementierung - SpielbaumC

```
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             // leave if goal is reached
21             if (type == MIN && value <= goalMin) return;
22             if (type == MAX && value >= goalMax) return;
23             // update goals
24             if (type == MIN) goalMax = value;
25             if (type == MAX) goalMin = value;
26         } } } // if, while, Konstruktor
27     public GameTreeNode(Position p) {
28         this(p, MIN, MAX);
29     }
30 }
```

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... (18014 nodes)

x..

...

...

generate tree... (1957 nodes)

x..

.o.

...

generate tree... (764 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() == -type) { value = -type; return; }  
5     Iterator<Integer> moves = p.getMoves();  
6     if (!moves.hasNext()) { value = NONE; 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             // leave if goal is reached
32             if (type == MIN && value <= goalMin) return;
33             if (type == MAX && value >= goalMax) return;
34             // update goals
35             if (type == MIN) goalMax = value;
36             if (type == MAX) goalMin = value;
37     } } } // if, while, Konstruktor
```

"GameTreeNodeD.java"

Output – Variante D

generate tree... (2914 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

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...

GUI: Model – View – Controller

Modell (Model):

Repräsentiert das Spiel, den aktuellen Spielzustand, und die Spiellogik.

Ansicht (View)

Die externe graphische(?) Benutzeroberfläche mit der die Benutzerin interagiert.

Steuerung (Controller)

Kontrollschicht, die Aktionen der Nutzerin and die Spiellogik weiterleitet, und Reaktionen sichtbar macht.

Typisch für viele interaktive Systeme. Es gibt viele Varianten (Model-View-Presenter, Model-View-Adapter, etc.).

GUI: Model – View – Controller

- ▶ Es gibt viele solcher Standardvorgehensweisen, für das Strukturieren, bzw. Schreiben von großen Programmen (**Design Patterns**, ↑**Softwaretechnik**).
- ▶ Es gibt auch **Anti Patterns**, d.h., Dinge, die man normalerweise nicht tun sollte (die aber trotzdem häufig vorkommen).

TicTacToe – GUI

MainFrame	
- controller	: Controller
+ showWinner	(int who)
+ put	(int place, int type)
+ illegalMove	(int place)
+ init	()

Game	
- view	: View
+ makeBestMove	()
+ makePlayerMove	()
+ movePossible	() : boolean
+ finished	() : boolean

MyController	
- view	: View
- game	: Model
+ checkMove	(int place)
+ switchPlayer	()
+ restart	()

View – Attributes

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4 import javax.swing.border.*;
5 import static javax.swing.SwingUtilities.*;
6
7
8 public class MainFrame extends JFrame implements
9                               PlayConstants,View {
10     private Controller controller;
11     private JDialog dia;
12     private JPanel arena;
13     private JPanel side;
14     private Container c;
```

"MainFrame.java"

```
16     public MainFrame(Controller con) {
17         controller = con;
18         setupUI();
19         init();
20     }
21     private void setupUI() {
22         c = getContentPane();
23         setLocation(100,100);
24         c.setLayout(new FlowLayout());
25         setVisible(true);
26     }
```

"MainFrame.java"

View – Interfacemethoden

```
27 public void init() { invokeLater(() ->{
28     c.removeAll();
29     arena = new JPanel();
30     arena.setBackground(Color.BLUE);
31     arena.setBorder(new LineBorder(Color.BLUE, 5));
32     arena.setLayout(new GridLayout(3,3,5,5));
33     arena.setPreferredSize(new Dimension(600,600));
34     for (int i=0; i<9;i++) {
35         MyButton b = new MyButton(i);
36         b.addActionListener( this::buttonAction );
37         arena.add(b); }
38     side = new JPanel();
39     side.setPreferredSize(new Dimension(200,600));
40     side.setLayout(new BorderLayout());
41     JButton b = new JButton("switch sides");
42     b.addActionListener( this::switchAction );
43     side.add(b, BorderLayout.CENTER);
44     c.add(arena);
45     c.add(side);
46     pack(); }};
```


View – Interfacemethoden

```
47 public void put(int place, int type) {
48     invokeLater()->{
49         JPanel canvas;
50         if (type == MIN) canvas = new Cross();
51         else canvas = new Circle();
52         arena.remove(place);
53         arena.add(canvas, place);
54         revalidate();
55         repaint();
56     });
57 }
58 public void illegalMove(int place) {
59     System.out.println("Illegal move: "+place);
60 }
```

"MainFrame.java"

View – Interfacemethoden

```
61     public void showWinner(int who) {
62         String str = "";
63         switch(who) {
64             case -1: str = "Kreuz gewinnt!"; break;
65             case 0: str = "Unentschieden!"; break;
66             case 1: str = "Kreis gewinnt!"; break;
67         }
68         final String s = str;
69         invokeLater(()->dia = new MyDialog(this,s));
70     }
```

"MainFrame.java"

View - ActionListener

```
72     public void switchAction(ActionEvent e) {
73         controller.switchPlayer();
74     }
75     public void buttonAction(ActionEvent e) {
76         MyButton button = (MyButton) e.getSource();
77         int place = button.getNumber();
78         controller.checkMove(place);
79     }
80     public void dialogAction(ActionEvent e) {
81         JButton b = (JButton) e.getSource();
82         if (e.getActionCommand() == "kill") {
83             System.exit(0);
84         } else {
85             controller.restart();
86             dia.dispose();
87         }
88     }
```

"MainFrame.java"

Controller – Attributes

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class MyController implements PlayConstants,
5                                     Controller {
6     private Model game;
7     private View view;
8     public void setup(Model m, View v) {
9         game = m; view = v;
10    }
```

"MyController.java"

Controller – Methoden

```
11     public void checkMove(int place) {
12         if (game.movePossible(place)) {
13             game.makePlayerMove(place);
14         }
15         else view.illegalMove(place);
16     }
17     public void switchPlayer() {
18         if (game.finished()) return;
19         game.makeBestMove();
20     }
21     public void restart() {
22         view.init();
23         game = new Game(view);
24 } }
```

"MyController.java"

Main

```
1 public static void main(String[] args) {
2     invokeLater()->{
3         Controller c = new MyController();
4         View v = new MainFrame(c);
5         Model m = new Game(v);
6         c.setup(m,v);
7     });
8 }
```

"MainFrame.java"

Was ist hier falsch?

Was passiert wenn wir einen sehr grossen Spielbaum berechnen?

Was ist hier falsch?

Was passiert wenn wir einen sehr grossen Spielbaum berechnen?


```
1 private void initTree() {  
2     try {Thread.sleep(4000);}  
3     catch(InterruptedException e) {}  
4     g.nodeCount = 0;  
5     g = new GameTreeNode(p);  
6     System.out.println("generate tree... (" +  
7         g.nodeCount + " nodes");  
8 }
```

"GameNew.java"

Die GUI reagiert nicht mehr, da wir die gesamte Berechnung im **Event Dispatch Thread** ausführen.

View - ActionListener

```
72 public void switchAction(ActionEvent e) {
73     ctrl.exec()->ctrl.switchPlayer();
74 }
75 public void buttonAction(ActionEvent e) {
76     MyButton button = (MyButton) e.getSource();
77     int place = button.getNumber();
78     ctrl.exec()->ctrl.checkMove(place);
79 }
80 public void dialogAction(ActionEvent e) {
81     JButton b = (JButton) e.getSource();
82     if (e.getActionCommand() == "kill") {
83         System.exit(0);
84     } else {
85         ctrl.exec()->ctrl.restart();
86         dia.dispose();
87     }
88 }
```

"MainFrameNew.java"

Controller – Attributes

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import java.util.concurrent.locks.*;
4
5 public class MyController extends Thread
6                             implements PlayConstants,
7                                     Controller {
8     private Model game;
9     private View view;
10    final Lock lock = new ReentrantLock();
11    final Condition cond = lock.newCondition();
```

"MyControllerNew.java"

Controller – Methoden

```
13     Runnable r = null;
14     public void exec(Runnable r) {
15         if (lock.tryLock()) {
16             this.r = r;
17             cond.signal();
18             lock.unlock();
19         }
20     }
21     public void run() {
22         lock.lock(); try {
23             while (true) {
24                 while (r == null)
25                     cond.await();
26                 r.run();
27                 r = null;
28             }}
29         catch (InterruptedException e) {}
30         finally { lock.unlock(); }
31     }
```

Controller – Methoden

```
32     public void setup(Model m, View v) {  
33         game = m; view = v;  
34         start();  
35     }
```