Implementierung:

- Die Knöpfe legen wir mittels `GridLayout` in ein `Panel`-Objekt.
- Die weiße Fläche mit schwarzem Quadrat malen wir auf ein `Canvas`-Objekt.
- Ein `Frame`-Objekt besitzt bereits das `BorderLayout`.
- Das `Panel`-Objekt legen wir im Westen des Frames ab, das `Canvas`-Objekt in der Mitte.

```
import java.awt.*;
import java.awt.event.*;
class MyPanel extends Panel implements ActionListener {
   Button b0, b1, b2;
   public MyPanel() {
      setBackground(Color.orange);
      b0 = new Button("red");
      b1 = new Button("blue");
      b2 = new Button("green");
      b0.addActionListener(this);
      b1.addActionListener(this);
      b2.addActionListener(this);
      setLayout(new GridLayout(6,1));
      add(b0); add(b1); add(b2);
   }
   ...
```

- Der Konstruktor `public GridLayout(int row, int col);` teilt die zur Verfügung stehende Fläche in ein Raster von gleich großen Feldern ein, die in `row` vielen Zeilen und `col` vielen Spalten angeordnet sind.
- die Felder werden sukzessive von links oben nach rechts unten aufgefüllt.
- Nicht alle Felder müssen tatsächlich belegt werden.

  Im Beispiel bleibt die Hälfte frei ...
- Gemeinsamer `ActionListener` für alle drei Knöpfe ist (hier) das Panel selbst ...

```
    public void actionPerformed(ActionEvent e) {
       Button b = (Button) e.getSource();
       if (b.getBackground() == Color.orange) {
          if (b == b0) b0.setBackground(Color.red);
          else if (b == b1) b1.setBackground(Color.blue);
          else b2.setBackground(Color.green);
       } else
          b.setBackground(Color.orange);
    }
 }  // end class MyPanel
 ...
```

- Der Aufruf `e.getSource();` liefert das Objekt, das das `ActionEvent`-Objekt erzeugte, hier ein `Button`-Objekt.
- Falls die Hintergrunds-Farbe orange ist, modifizieren wir die Farbe. Ansonsten setzen wir sie auf orange zurück.

```
public class Nest extends Frame {
   public Nest() {
      setSize(200,150); setLocation(500,500);
      add(new MyPanel(), "West");
      add(new MyCanvas(), "Center");
   }
   public static void main(String[] args) {(new Nest()).show();}
} // end of class Nest
class MyCanvas extends Canvas {
   public MyCanvas() { setBackground(Color.white);}
   public void paint(Graphics page) {
      page.setColor(Color.black);
      page.fillRect(50,50,10,10);
   }
} // end of class MyCanvas
```

- Ein neues `Canvas`-Objekt besitzt eigentlich Breite und Höhe 0 — und ist damit unsichtbar!
- Im `BorderLayout` wird eine Komponente jedoch auf die gesamte zur Verfügung stehende Fläche ausgedehnt.
- Wie in der Klasse `Applet` wird zum (Neu-)Malen der `Canvas`-Fläche implizit die Objekt-Methode `public void paint(Graphics page);` aufgerufen.

... Schluss mit Java!!!

Was es sonst noch so in Java gibt:

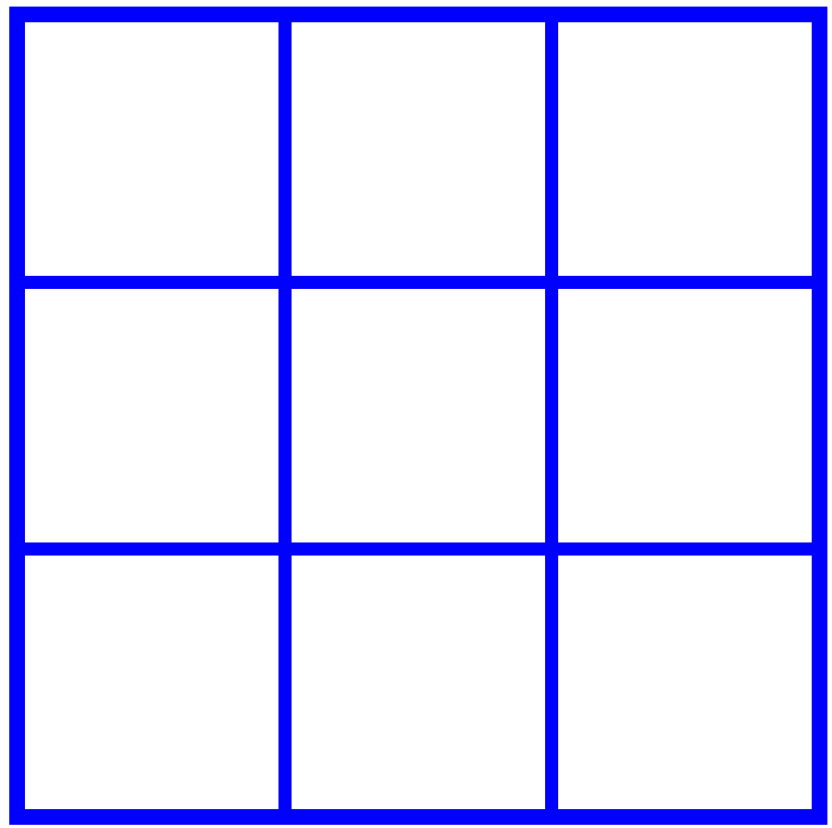
... an nützlichem:	innere Klassen;
... an mysteriösem:	Klassen zur Selbst-Reflektion;
... an comfortablem:	Malen mit Swing;
... an technischem:	Networking mit Sockets, RMI, Jini, Corba, ...
... an sonstigem:	... :-)

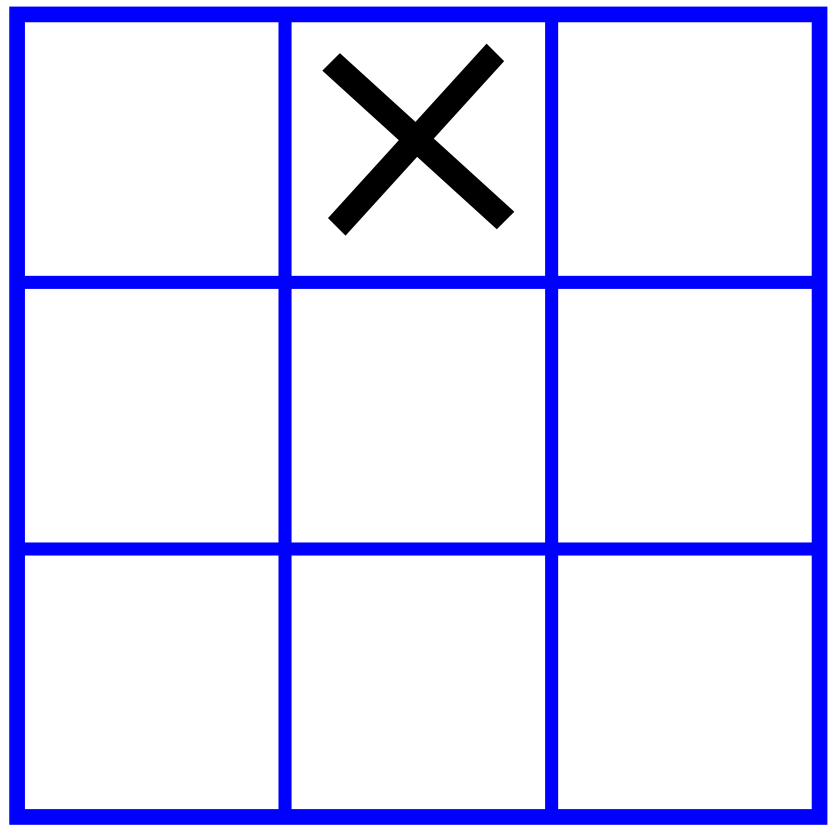
22 Tic-Tac-Toe

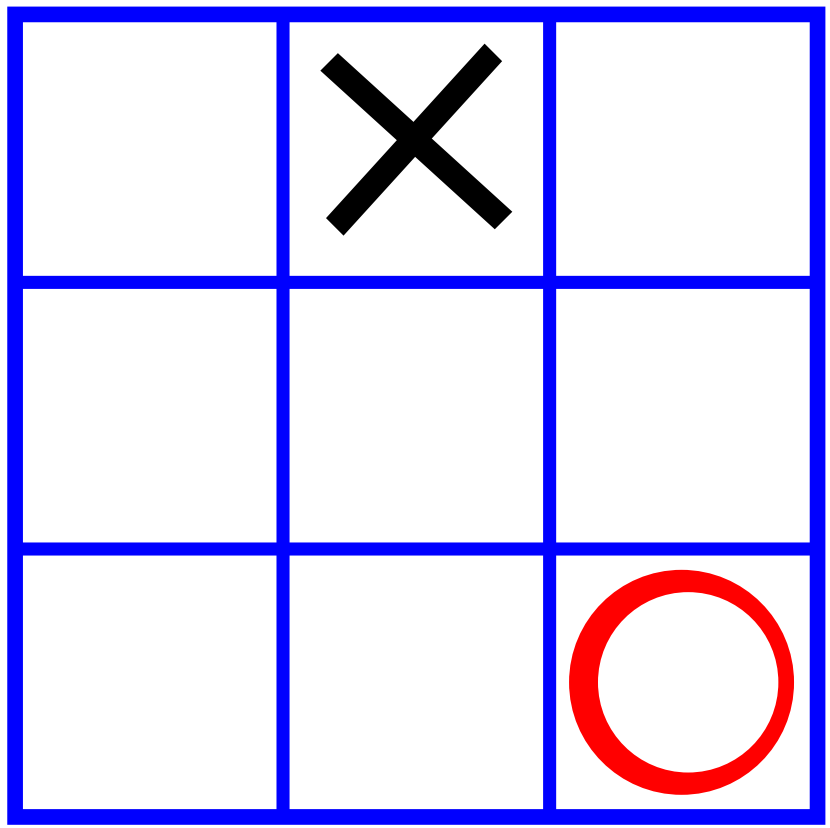
Regeln:

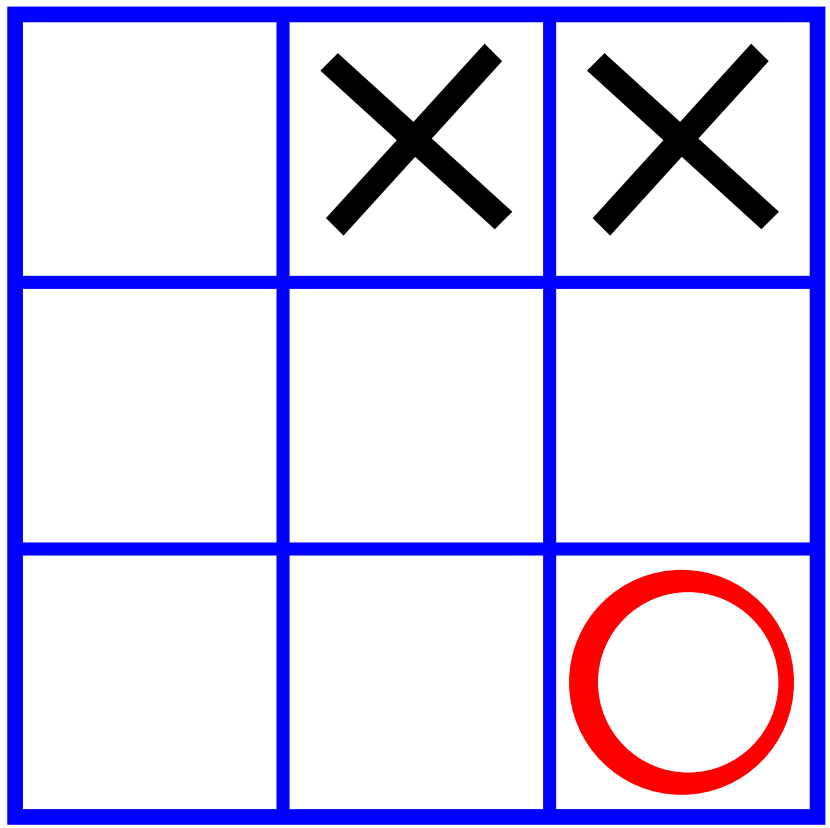
- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

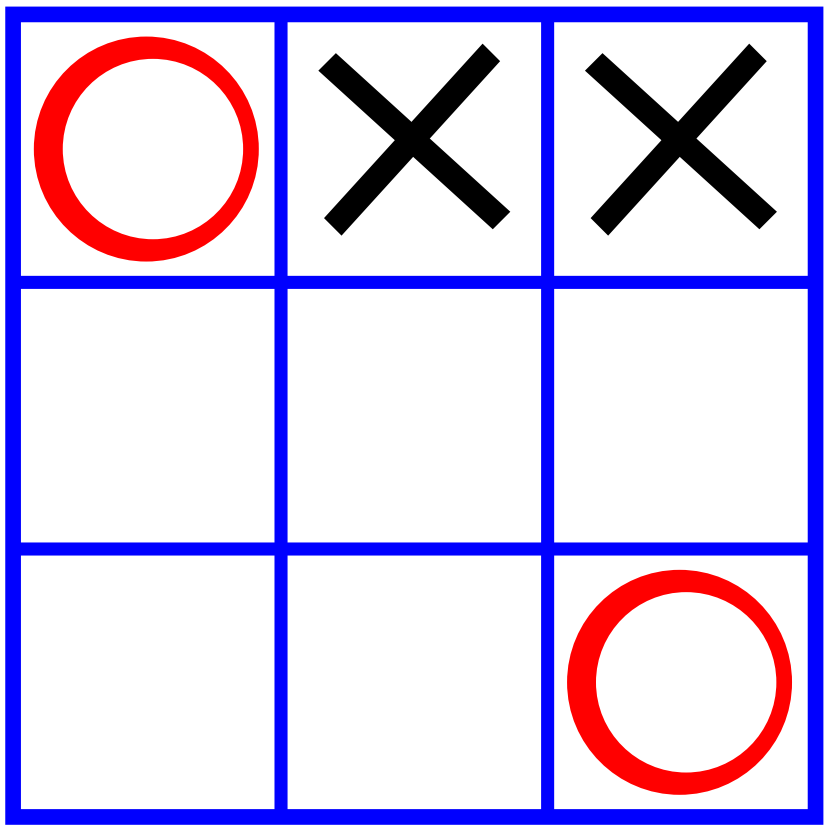
Beispiel:

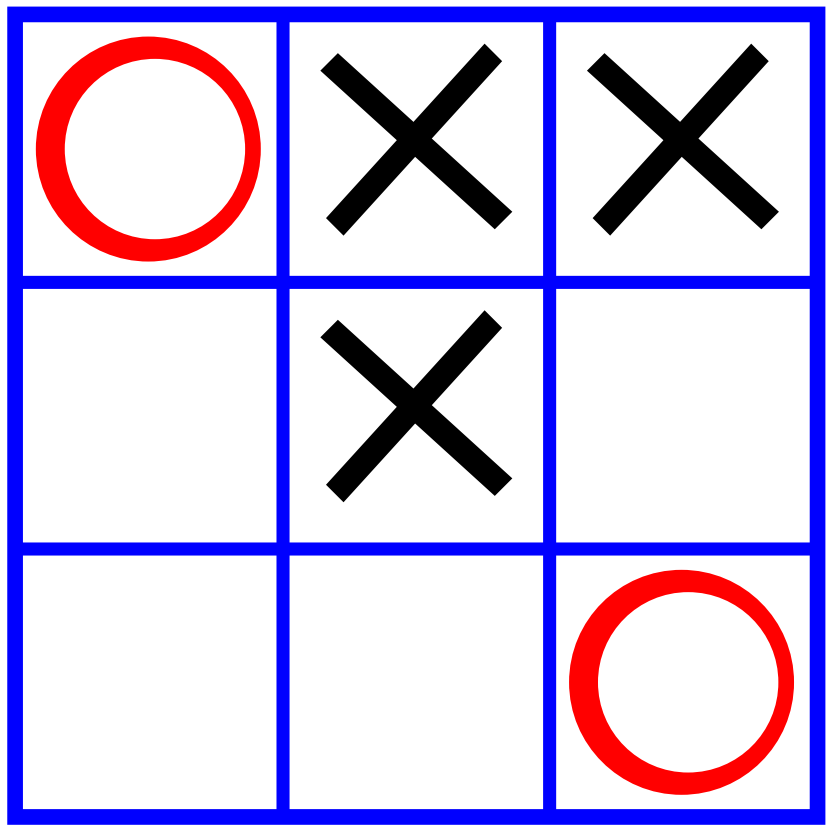


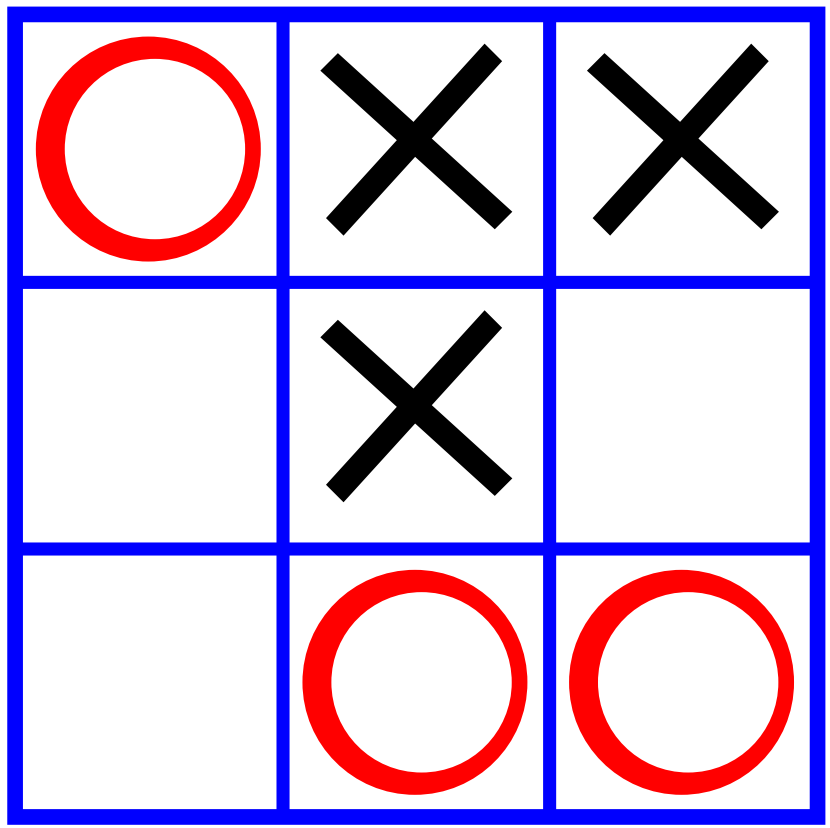


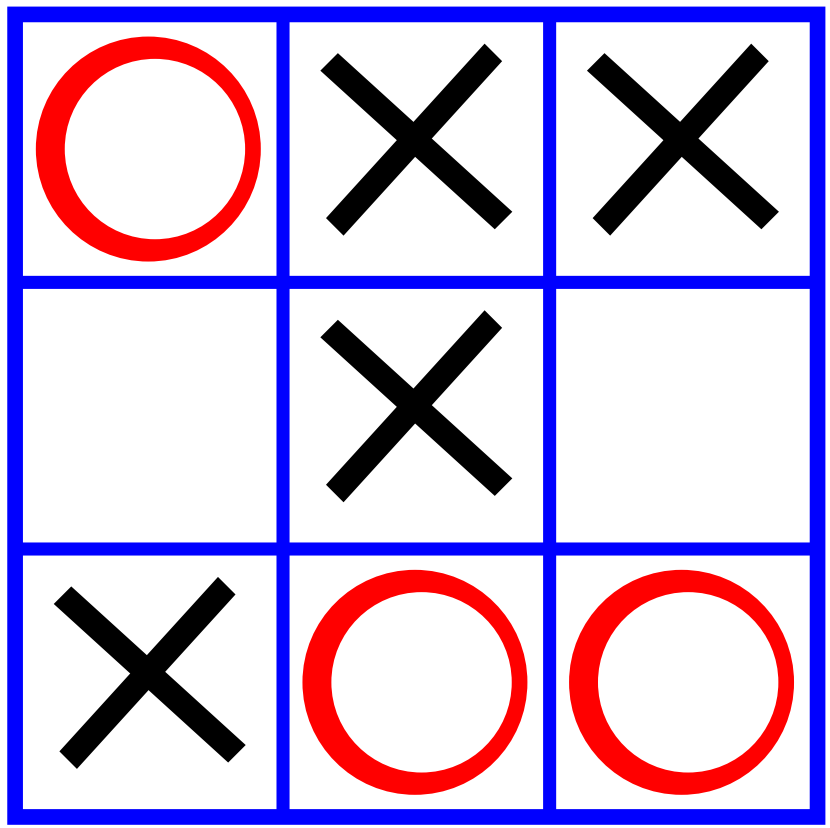












... offenbar hat die anziehende Partei gewonnen.

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?

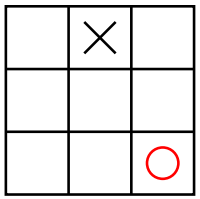
Hintergrund:

Tic-Tac-Toe ist ein endliches Zwei-Personen-Nullsummen-Spiel.
Das heißt:

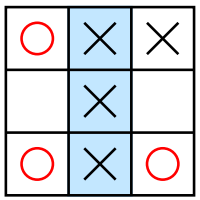
- Zwei Personen spielen gegeneinander.
- Was der eine gewinnt, verliert der andere.
- Es gibt eine endliche Menge von Spiel-Konfigurationen.
- Die Spieler ziehen abwechselnd. Ein Zug wechselt die Konfiguration, bis eine End-Konfiguration erreicht ist.
- Jede End-Konfiguration ist mit einem Gewinn aus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

... im Beispiel:

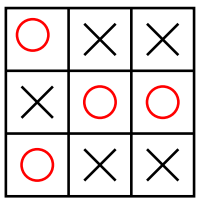
Konfiguration:



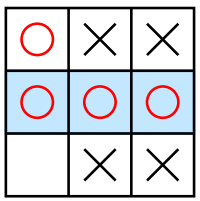
End-Konfigurationen:



Gewinn -1

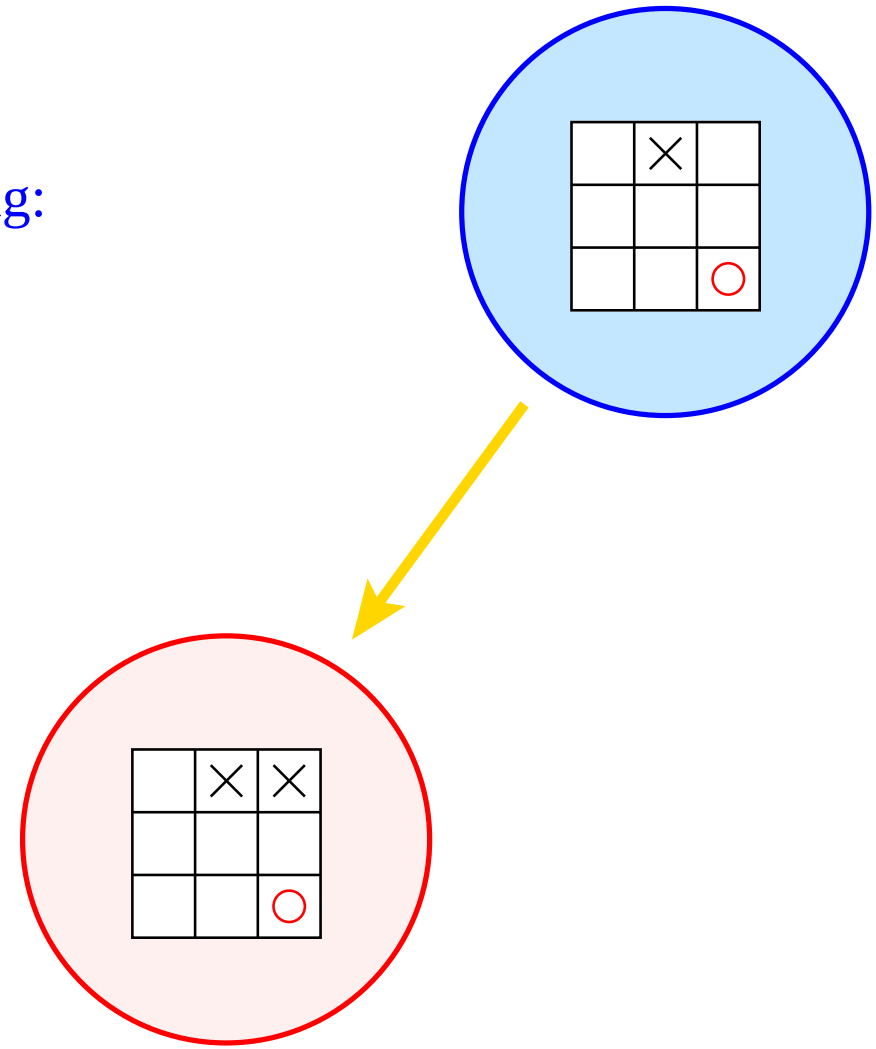


Gewinn 0

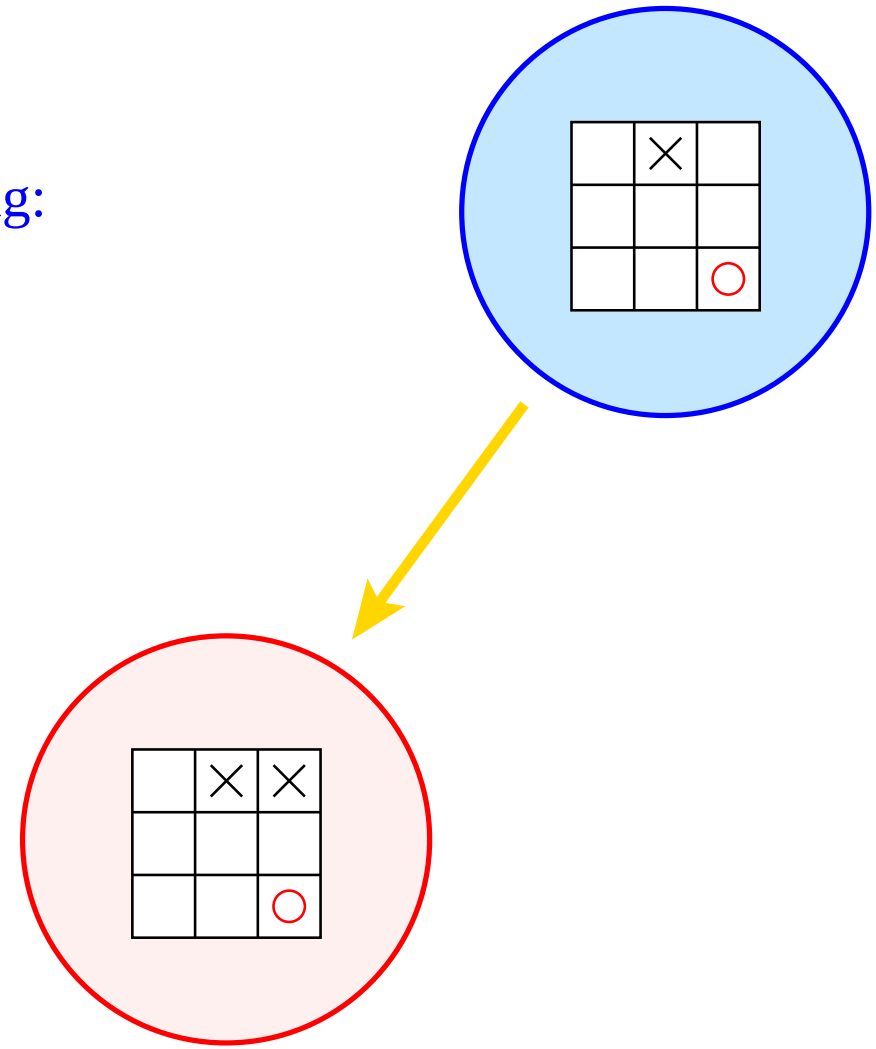


Gewinn +2

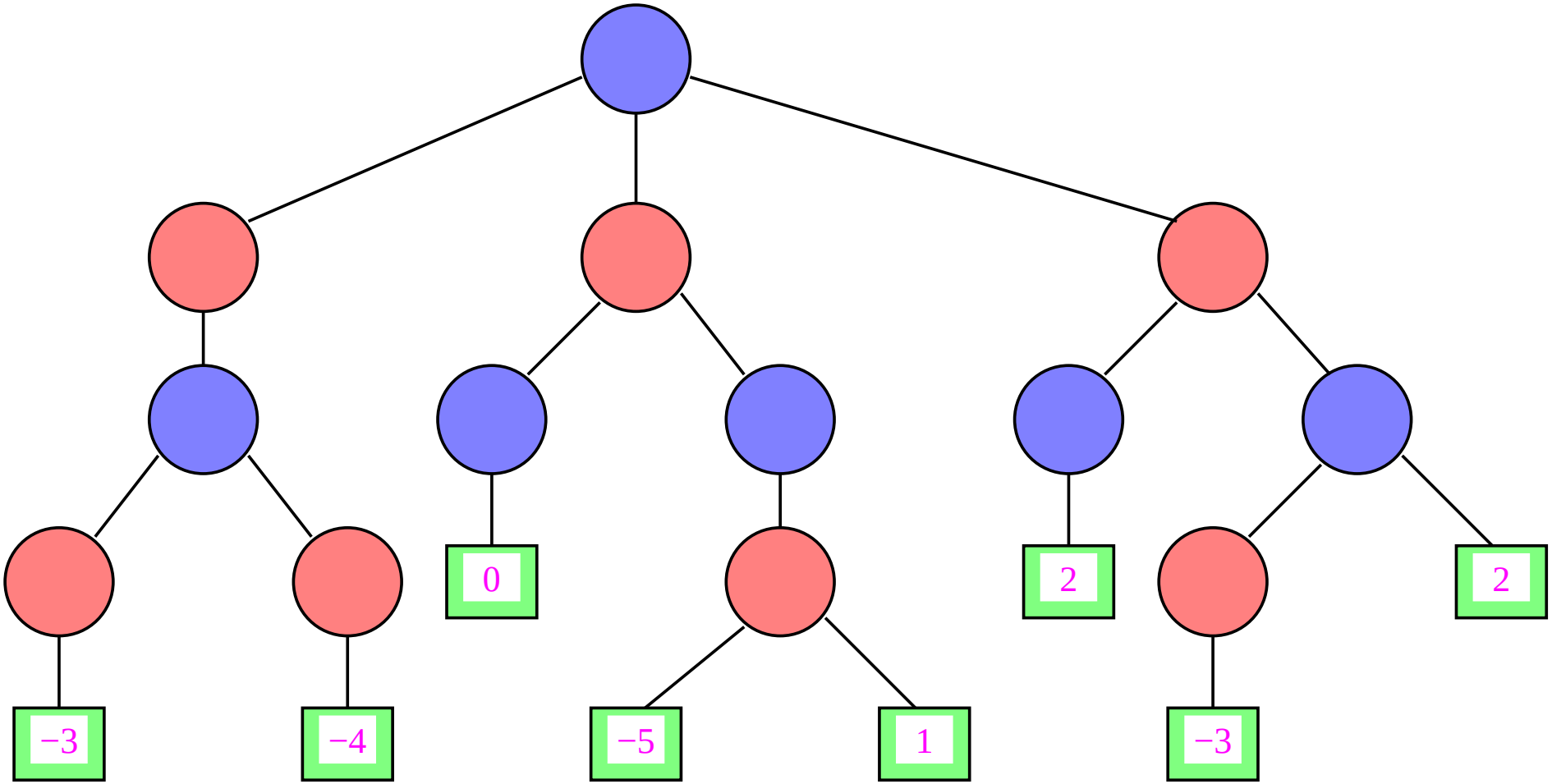
Spiel−Zug:



Spiel−Zug:



Alle möglichen Spiel-Verläufe lassen sich als ein Baum darstellen:


−3
−4
0
−5
1
2
−3
2

Knoten des Spielbaums	$=$	Konfigurationen
Kanten	$=$	Spiel-Züge
Blätter	$=$	End-Konfigurationen

Frage:

Wie finden wir (z.B. als blaue Person) uns im Spiel-Baum zurecht?

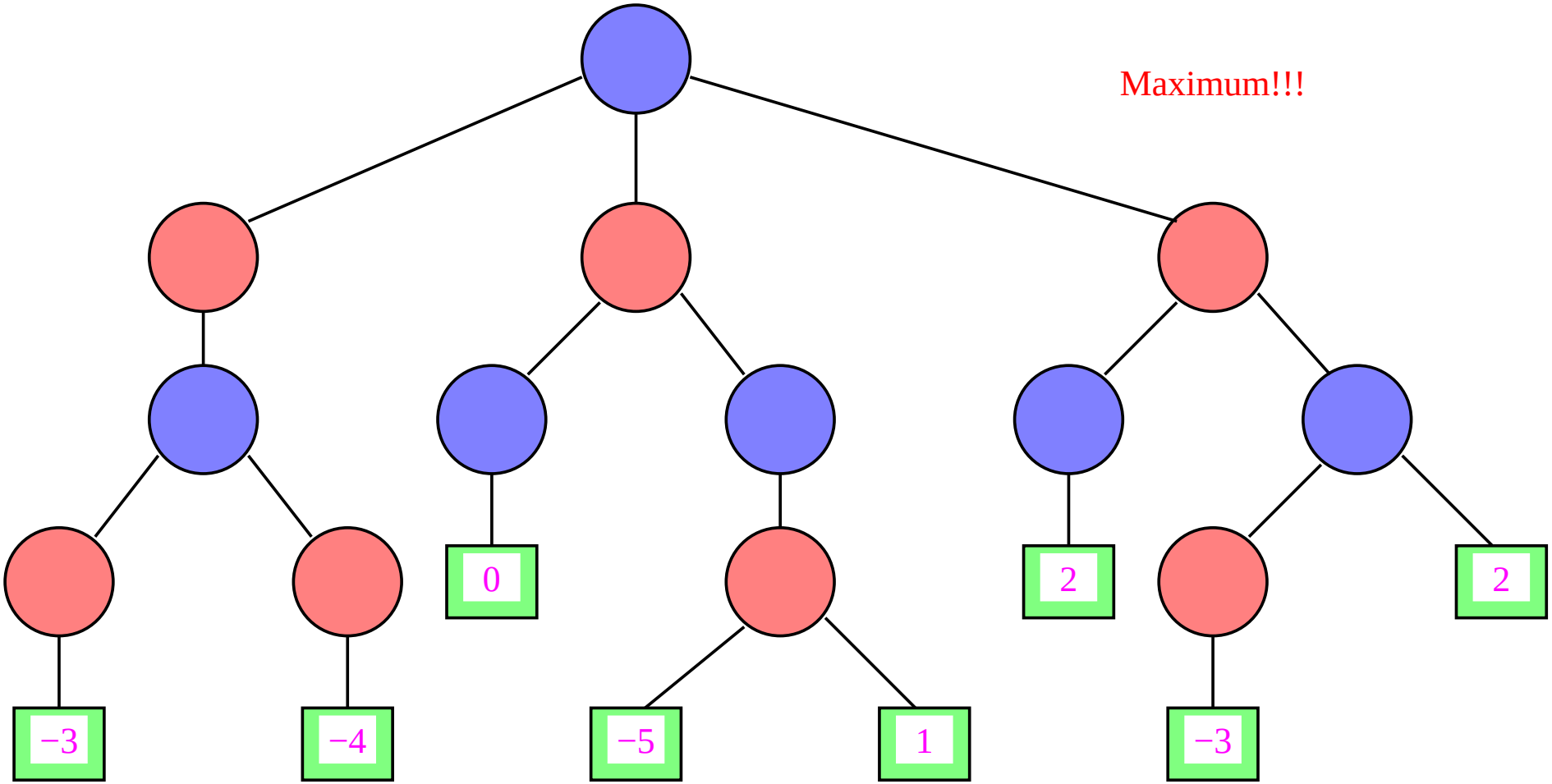
Was müssen wir tun, um sicher ein negatives Blatt zu erreichen?

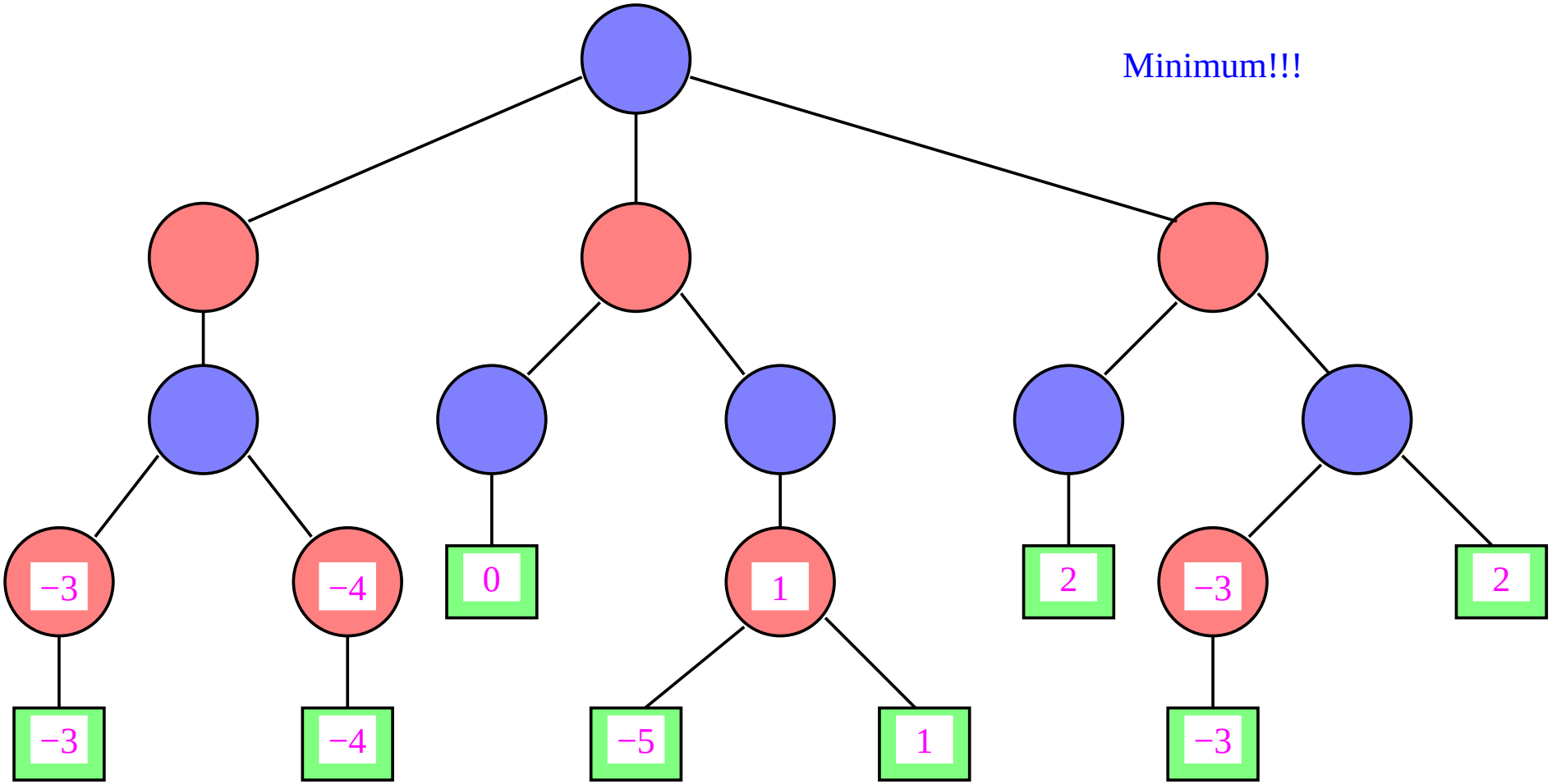
Idee:

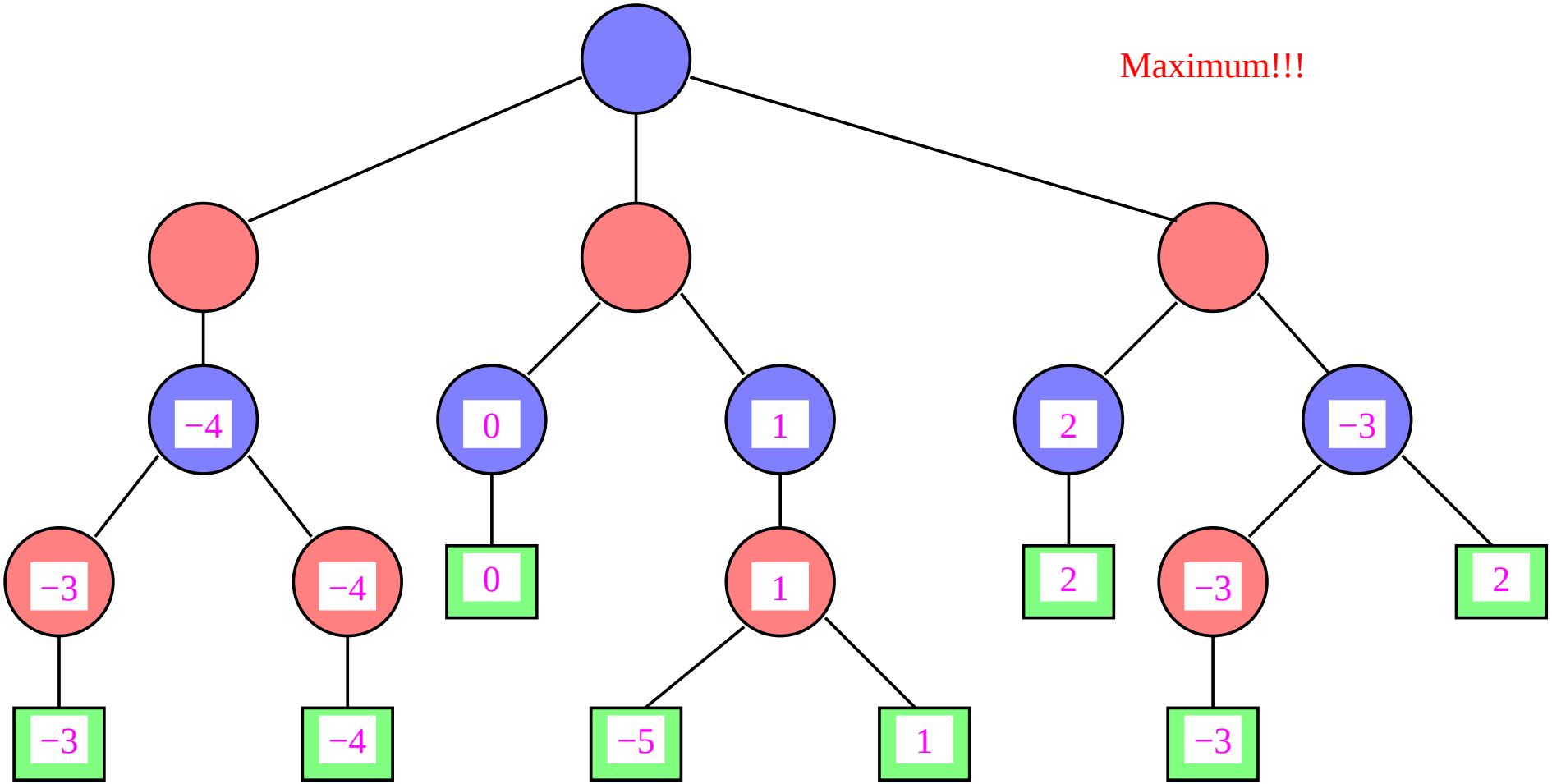
- Wir ermitteln für jede Konfiguration den jeweils besten zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

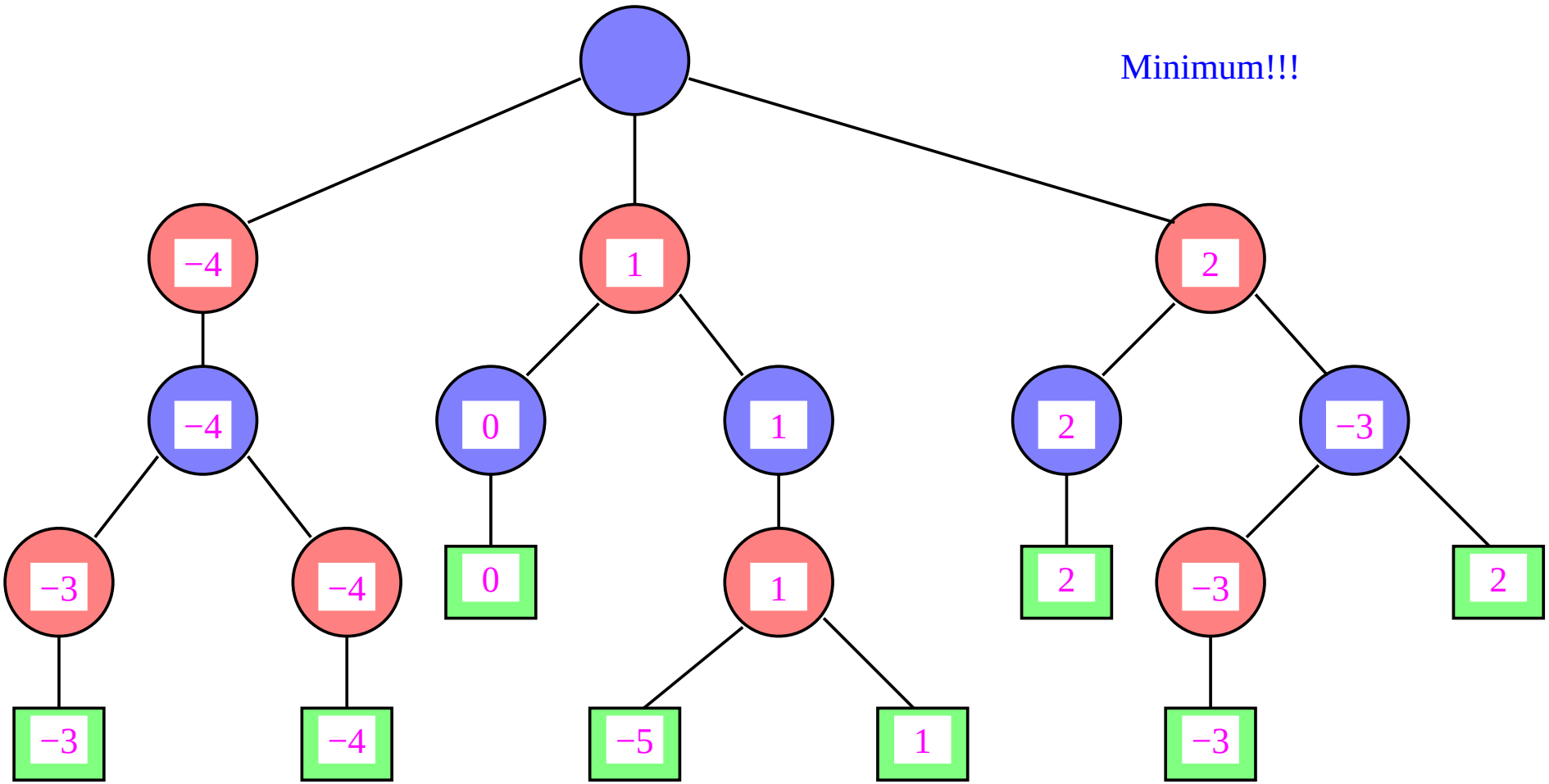
  Fall 1: Die Konfiguration ist blau: wir sind am Zug. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

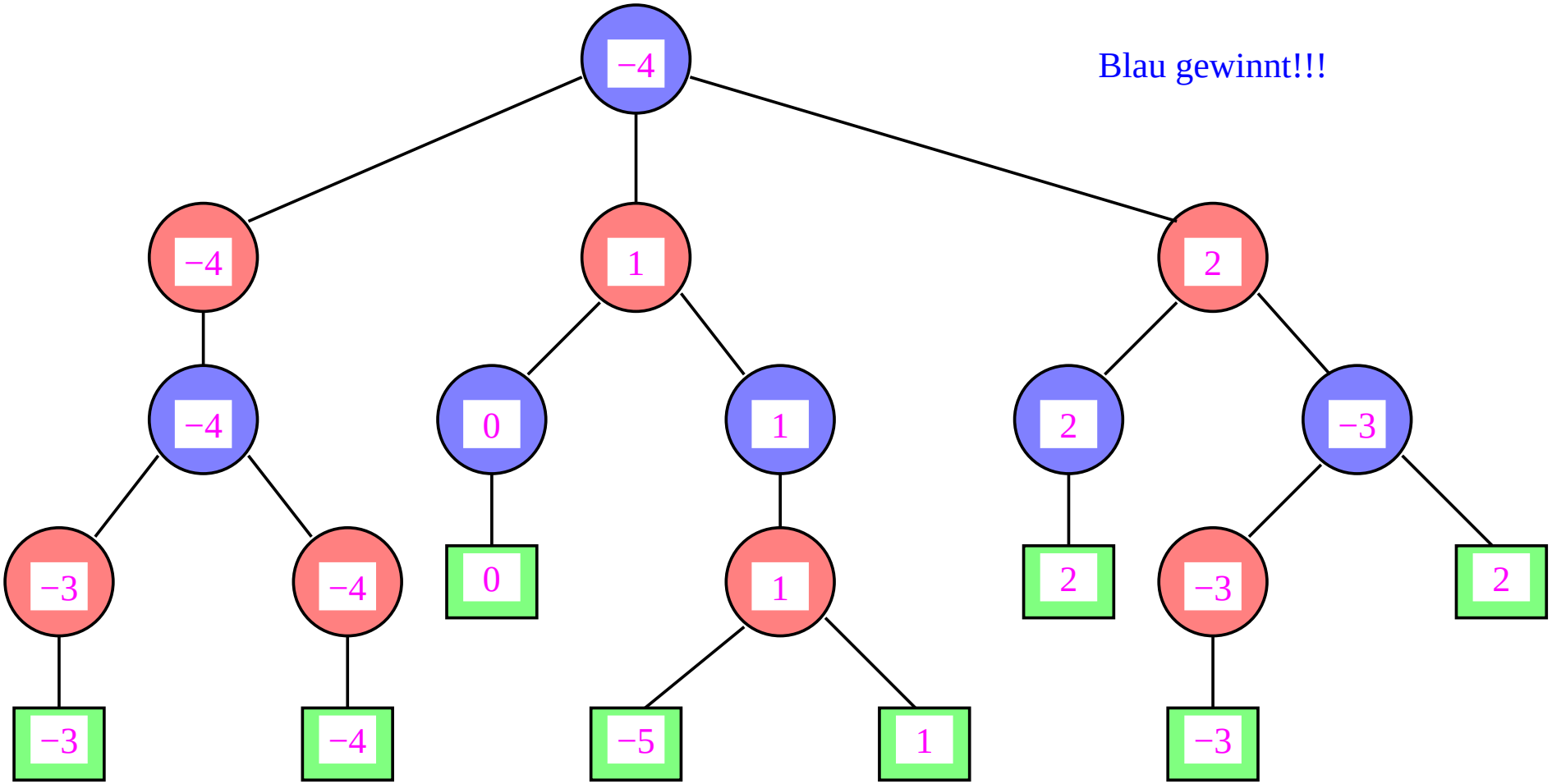
  Fall 2: Die Konfiguration ist rot: der Gegner ist am Zug. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.


Maximum!!!
−3
−4
0
−5
1
2
−3
2


Minimum!!!
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3


Maximum!!!
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3

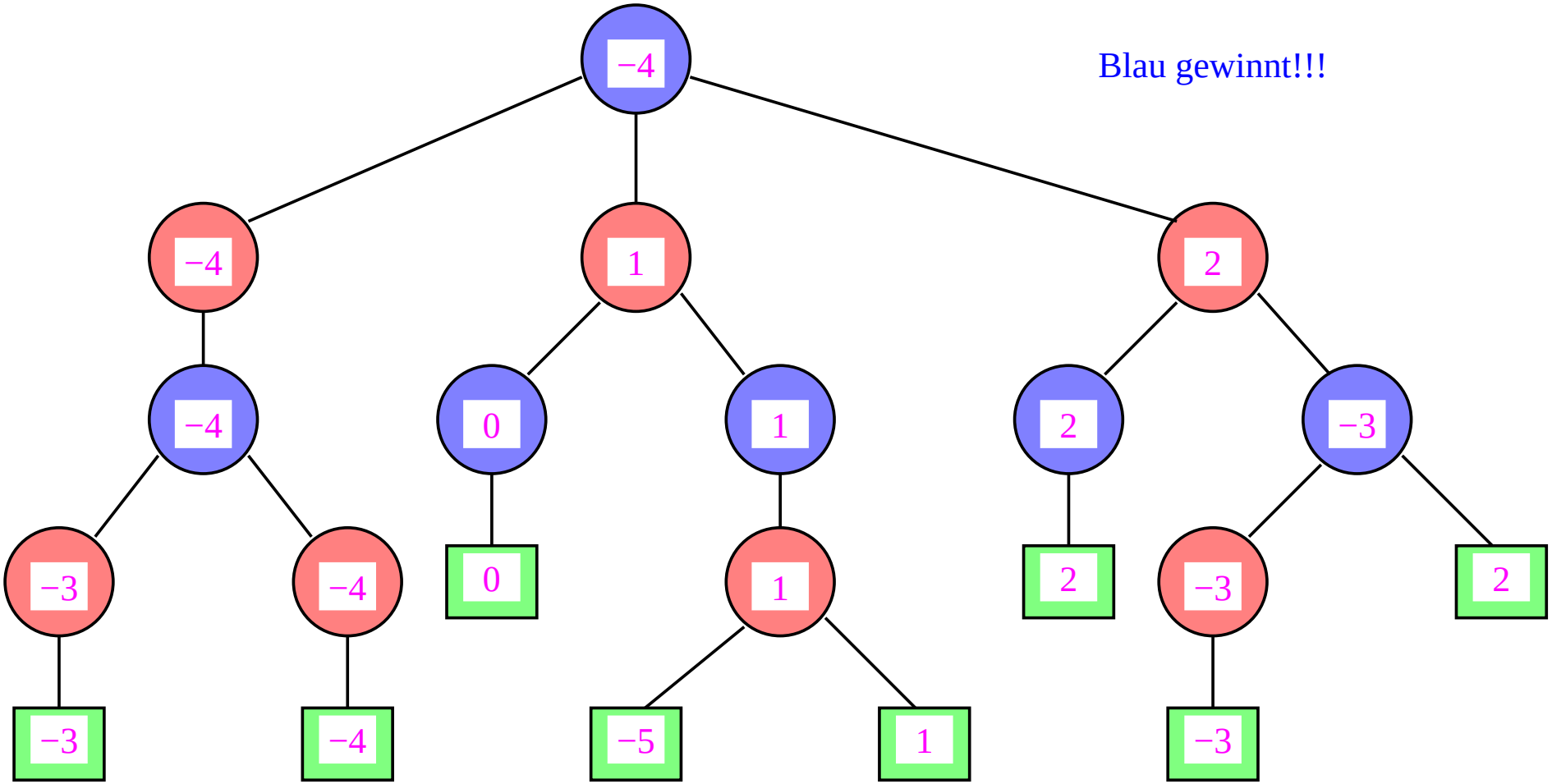

Minimum!!!
−4
1
2
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3

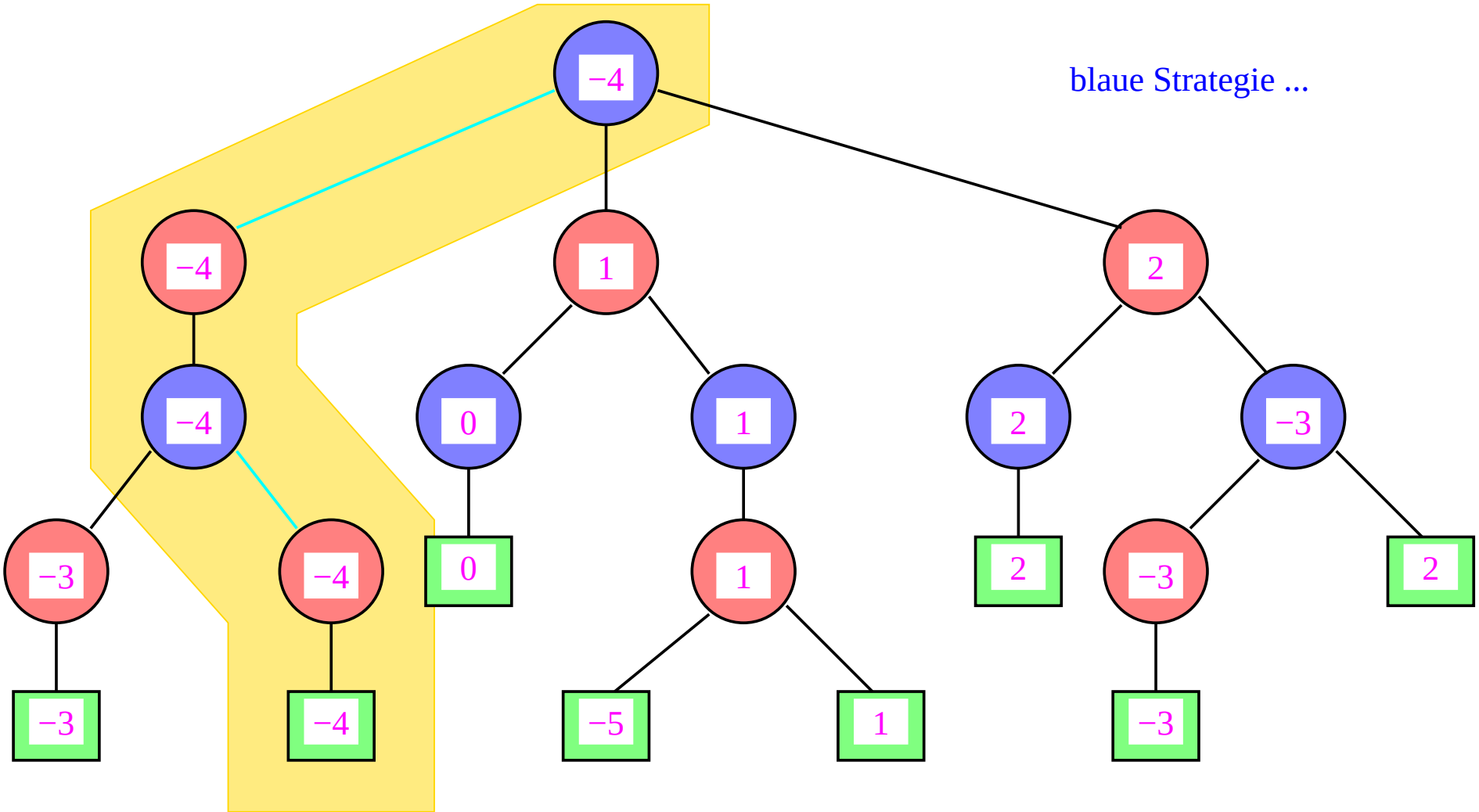

−4
Blau gewinnt!!!
−4
1
2
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3

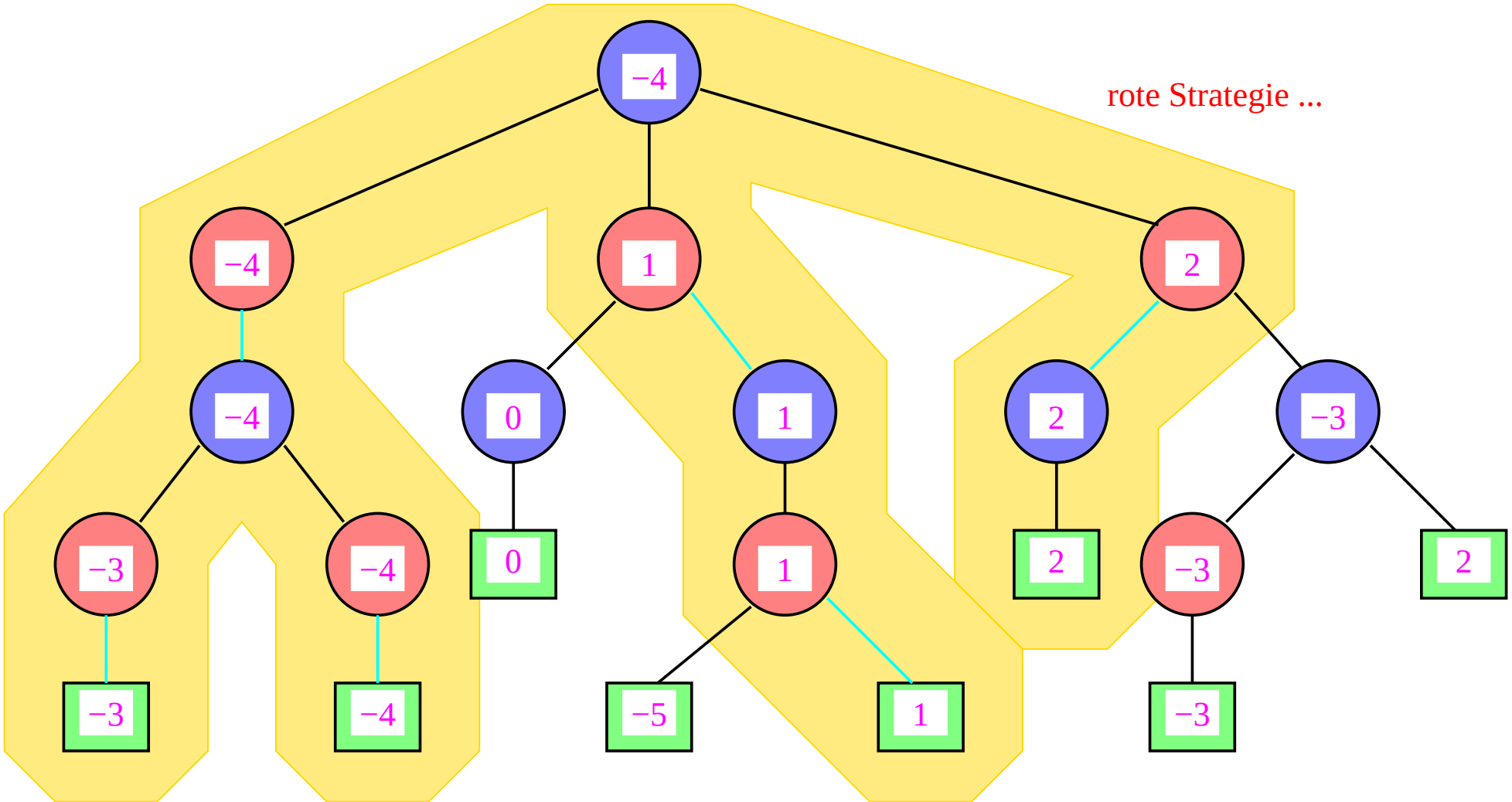
Eine Strategie ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

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

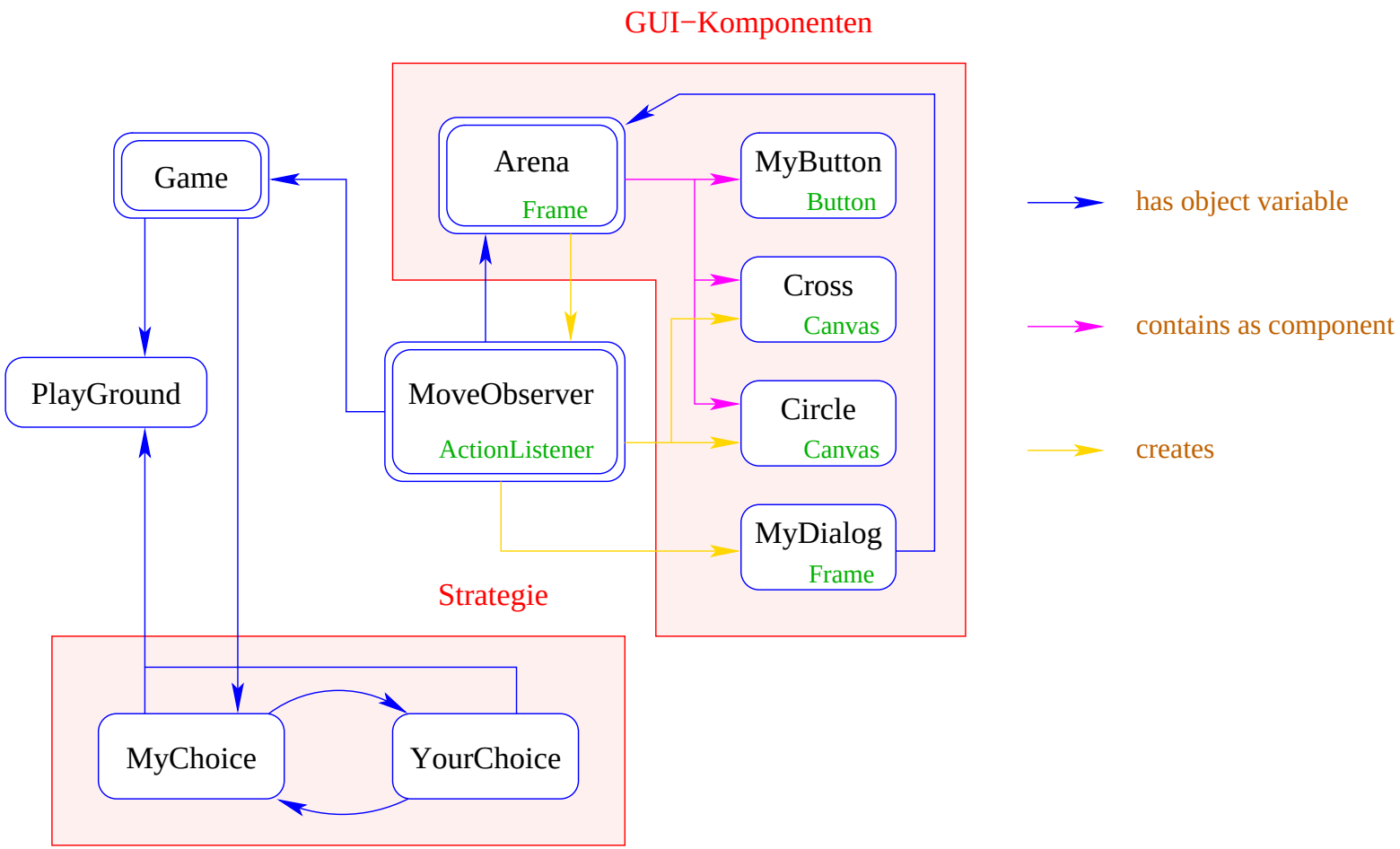
Eine akzeptable Strategie ist eine, deren Anwendung einen Verlust des Spiels verhindert, wannimmer das möglich ist ...


−4
Blau gewinnt!!!
−4
1
2
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3


blaue Strategie ...
−4
−4
1
2
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3


rote Strategie ...
−4
−4
1
2
−4
0
1
2
−3
−3
−4
0
1
2
−3
2
−3
−4
−5
1
−3

22.1 Überblick über das Tic-Tac-Toe-System



Bemerkung:

Der Aufbau besteht konzeptuell aus drei Komponenten:

- $\Rightarrow$ der internen Repräsentation des Spiels, des aktuellen Zustands und der Spiellogik

  $\Longrightarrow$ das Modell

- $\Rightarrow$ der externen graphische Benutzeroberfläche, mit der die Benutzerin interagiert

  $\Longrightarrow$ die Ansicht

- $\Rightarrow$ einer Kontrollschicht, die die Aktionen der Benutzerin an die Implementierung der Spiellogik weiterleitet und die Reaktion darauf auf der Oberfläche sichtbar macht

  $\Longrightarrow$ die Kontrolle

Zusammen: Model-View-Control

- Ein solcher Aufbau ist typisch für viele interaktive Anwendungssysteme.
- Neben MVC gibt es für viele weitere häufige Anwendungssituationen nützliche Standard-Vorgehensweisen.

  Diese nennt man auch Design Patterns.

  ↑ Software Engineering

22.2 Berechnung einer Strategie

- Die Knoten des Spiel-Baums sind aus den Klassen
  `YourChoice` und `MyChoice`.
- `MyChoice` implementiert Knoten, in denen das Programm zieht.
- `YourChoice` implementiert Knoten der Spielerin.
- Jeder Knoten enthält:
  das aktuelle Spielbrett `ground`,
  einen garantierten Gewinn `value`
  sowie (Verweise auf) die Nachfolger-Knoten.
- `MyChoice`-Knoten enthalten zusätzlich den empfohlenen Zug
  `acceptableChoice`.

```
public class YourChoice implements PlayConstants {
   private PlayGround ground; private int value;
   private MyChoice[] answer;
   public YourChoice(PlayGround g, int place) {
      ground = new PlayGround(g,place,ME);
      answer = new MyChoice[9];
      value = ME;
      PossibleMoves moves = new PossibleMoves(ground);
      for(int choice = moves.next(); choice!=-1; choice = moves.next())
         if (ground.won(choice,YOU)) { value = YOU; continue;}
         answer[choice] = new MyChoice(ground,choice);
         int win = answer[choice].value();
         if (win < value) value = win;
      }
   }
   ...
```