22.6 Ein Dialog-Fenster

Aufgaben:

1. Anzeige des Spiel-Ausgangs;
2. Ermöglichen der Beendigung bzw. Wiederholung des Spiels.

```
import java.awt.*;
import java.awt.event.*;
public class MyDialog extends Frame implements ActionListener {
   private Button repeat, kill; private Arena arena;
   public MyDialog(Arena frame, String string) {
      arena = frame;
      setLocation(200,200); setSize(250,50);
      setBackground(Color.yellow);
      setForeground(Color.blue);
      setLayout(new FlowLayout());
      add(new Label(string));
      repeat = new Button("new");
      repeat.addActionListener(this); add(repeat);
      kill = new Button("kill");
      kill.addActionListener(this);   add(kill);
   }
```

- `public void setForeground(Color c);` setzt die aktuelle Vordergrund-Farbe auf `c`. Schrift erscheint dann z.B. in dieser Farbe.
- Wir legen ein neues Fenster mit gelbem Hintergrund und blauem Vordergrund an.
- Wir benutzen das `FlowLayout`, um Elemente im Fenster zu positionieren.
- Den Kommentar zum Spiel-Ergebnis übergeben wir dem Konstruktor, der es in einem `Label`-Element anzeigt.
- Weiterhin fügen wir zwei Knöpfe für Wiederholung bzw. Beendigung hinzu.
- Gemeinsamer `ActionListener` beider Knöpfe ist das Fenster-Objekt selbst ...

```
   ...
   public void actionPerformed(ActionEvent e) {
      Button b = (Button) e.getSource();
      if (b == kill) { arena.setVisible(false); System.exit(0);}
      else {
         arena.start(); arena.setVisible(true);
         setVisible(false);
      }
   }
}  // end of class MyDialog
```

- `public void exit(int x);` ist eine Klassen-Methode der Klasse `System`, die die Applikation (mit Rückgabe-Wert `x`) beendet.

- Falls der `"kill"`-Knopf gedrückt wurde, wird die Applikation beendet.
- Falls der `"new"`-Knopf gedrückt wurde, wird für `arena` die `start()`-Methode aufgerufen, d.h. ein neues `Game`-Objekt angelegt und die Komponenten in `arena` neu zu Knöpfen initialisiert.
- `setVisible(false);` lässt das Fenster verschwinden.

22.7 Effizienz

Problem:

Spielbäume können RIESIG werden!!

Unsere Lösung:

- Wir erzeugen die `ME`-Strategie nicht für alle möglichen Spiel-Verläufe, sondern erst nach dem ersten Zug der Gegnerin. Spart ... Faktor 9
- Wir berücksichtigen Zug-Zwang. Spart ... ??!!...:-)
- Wir sind mit akzeptablen `ME`-Zügen zufrieden. Spart ungefähr ... Faktor 2

Achtung:

- Für Tic-Tac-Toe reicht das vollkommen aus: pro Spielverlauf werden zwischen 126 und 1142 `MyChoice`-Knoten angelegt ...
- Für komplexere Spiele wie Dame, Schach oder gar Go benötigen wir weitere Ideen ...

1. Idee: Eröffnungen

- Tabelliere Anfangs-Stücke optimaler Spiel-Verläufe.
- Konstruiere die Strategie erst ab der ersten Konfiguration, die von den tabellierten Eröffnungen abweicht ...

Beispiel: Tic-Tac-Toe

Wir könnten z.B. beste Antworten auf jeden möglichen Eröffnungs-Zug tabellieren:

```
public interface Opening {
   int[] OPENING = {
      4, 4, 4, 4, 2, 4, 4, 4, 4
   };
}
```

- Die Funktion `int nextMove(int place);` schlägt dann den ersten Antwort-Zug in `OPENING` nach.

- Erst bei der zweiten Antwort (d.h. für den vierten Stein auf dem Brett) wird die `ME`-Strategie konstruiert.
- Dann bleiben grade mal höchstens $6! = 720$ Spiel-Fortsetzungen übrig ... die Anzahl der tatsächlich benötigten `MyChoice`-Knoten scheint aber nur noch zwischen 9 und 53 zu schwanken (!!!)

2. Idee: Bewertungen

Finde eine geeignete Funktion `advice`, die die Erfolgsaussichten einer Konfiguration direkt abschätzt, d.h. ohne Aufbau eines Spielbaums.

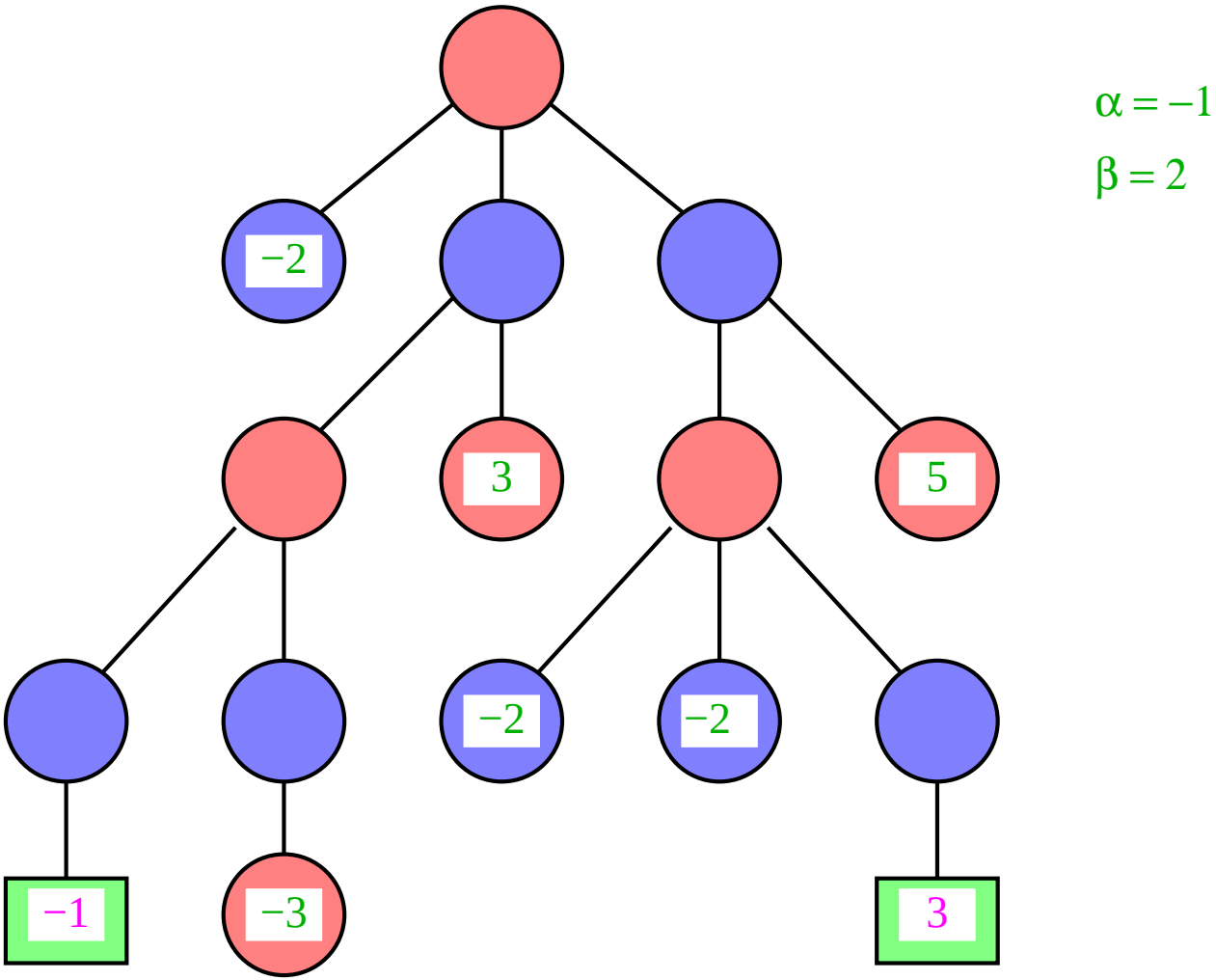
Achtung:

- I.a. ist eine solche Funktion nicht bekannt :-(
- Man muss mit unpräzisen bis fehlerhaften Bewertungs-Funktionen zurecht kommen ...

3. Idee: (α, β)-Pruning

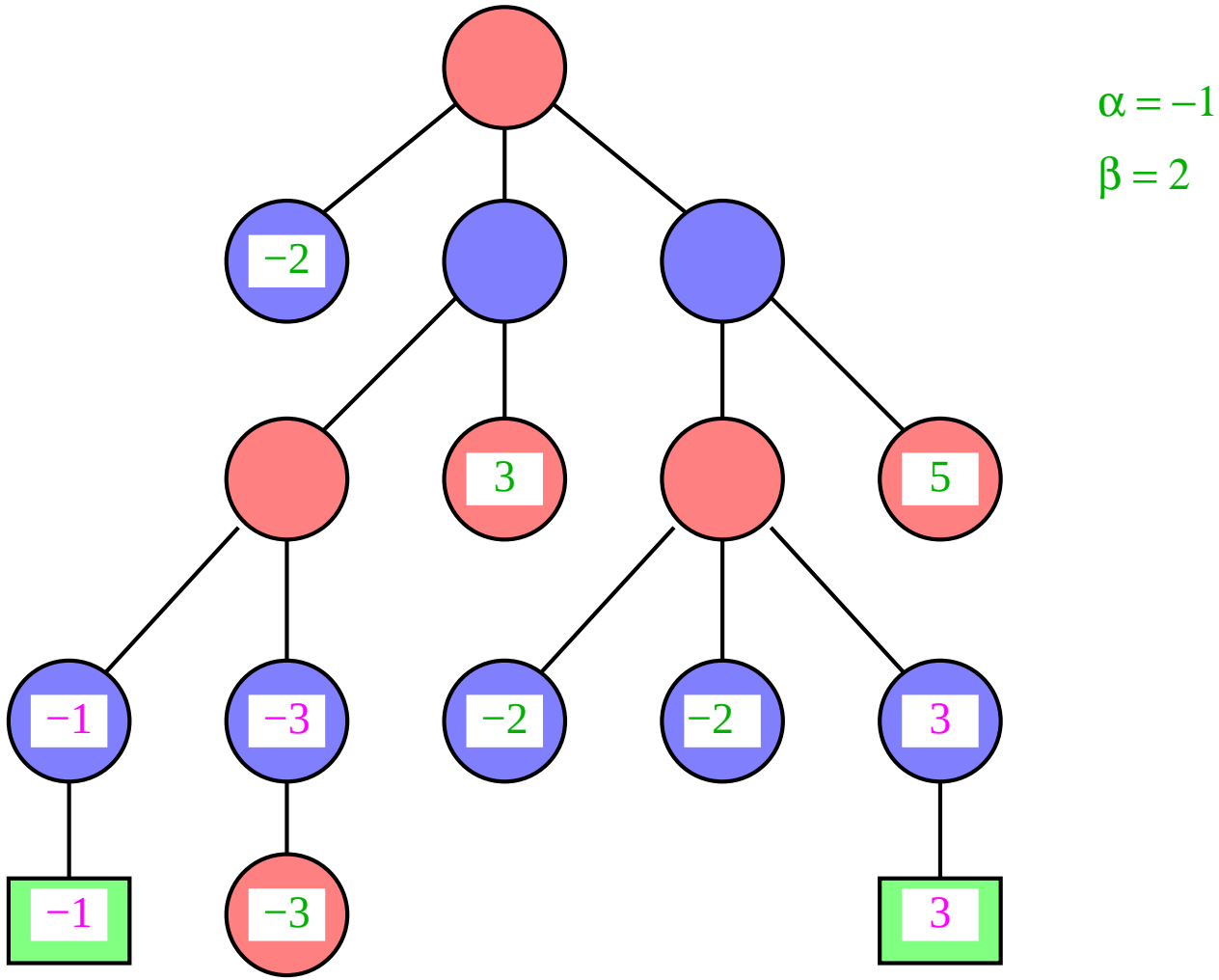
- Wir nehmen an, wir hätten eine halbwegs zuverlässige Bewertungsfunktion `advice`, d.h. es gibt Zahlen $\alpha < 0 < \beta$ so dass für Konfigurationen conf gilt:
  - Ist `advice(`conf`) <` α, gewinnt voraussichtlich die Gegnerin;
  - Ist β `< advice(`conf`)`, gewinnt voraussichtlich das Programm.
- Zur Bestimmung unseres nächsten Zugs, betrachten wir sukzessive alle Nachfolger-Konfigurationen conf.
  - Ist β `< advice(`conf`)`, ist der Zug akzeptabel.

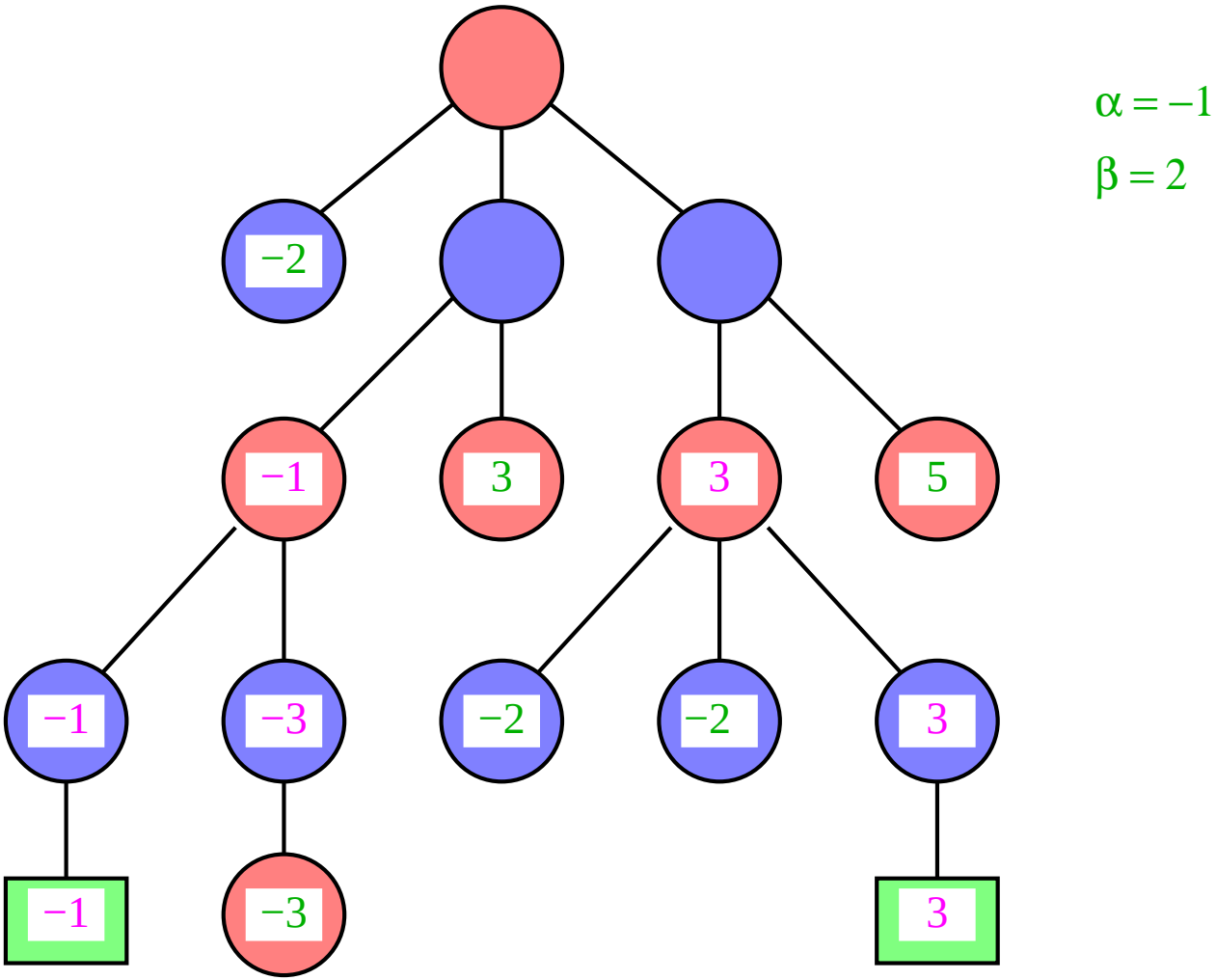
- Gibt es keinen akzeptablen Zug, betrachten wir rekursiv die Nachfolger aller Konfigurationen conf, für die $\alpha <$ `advice(`conf`)` .
- Für gegnerische Konfigurationen gehen wir dual vor ...



$\alpha = -1$

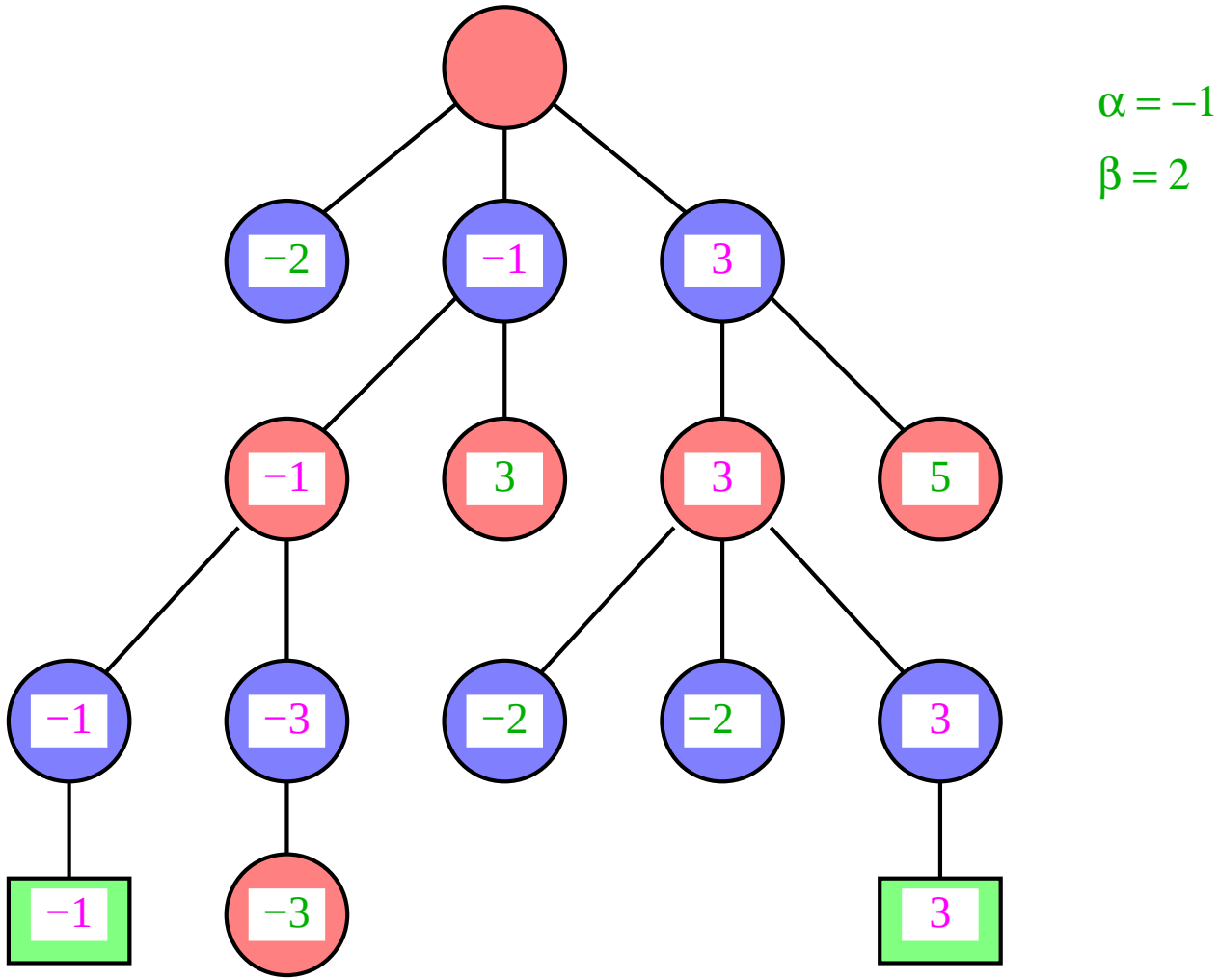
$\beta = 2$

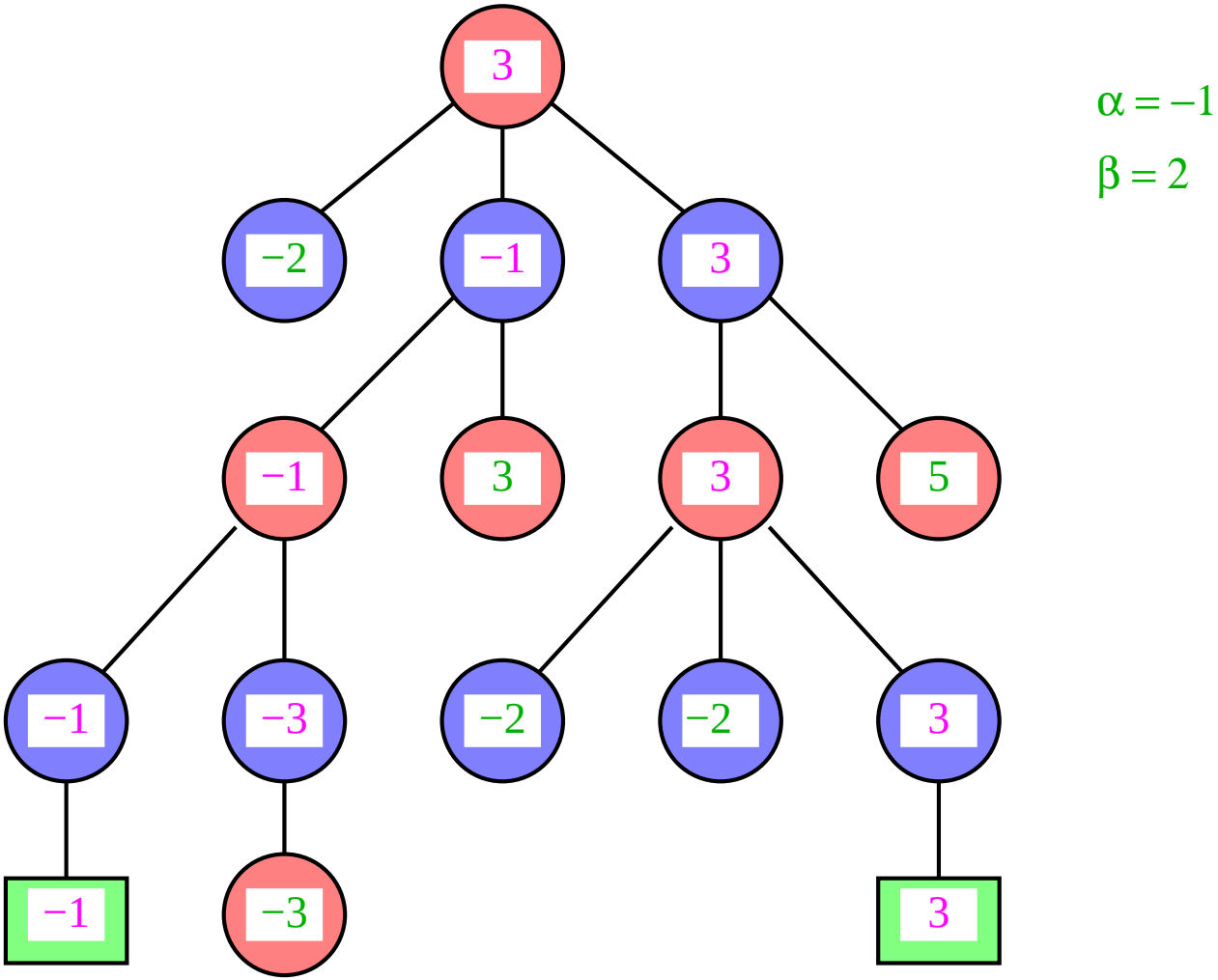

α = −1
β = 2
−2
3
5
−1
−3
−2
−2
3
−1
−3
3



$\alpha = -1$

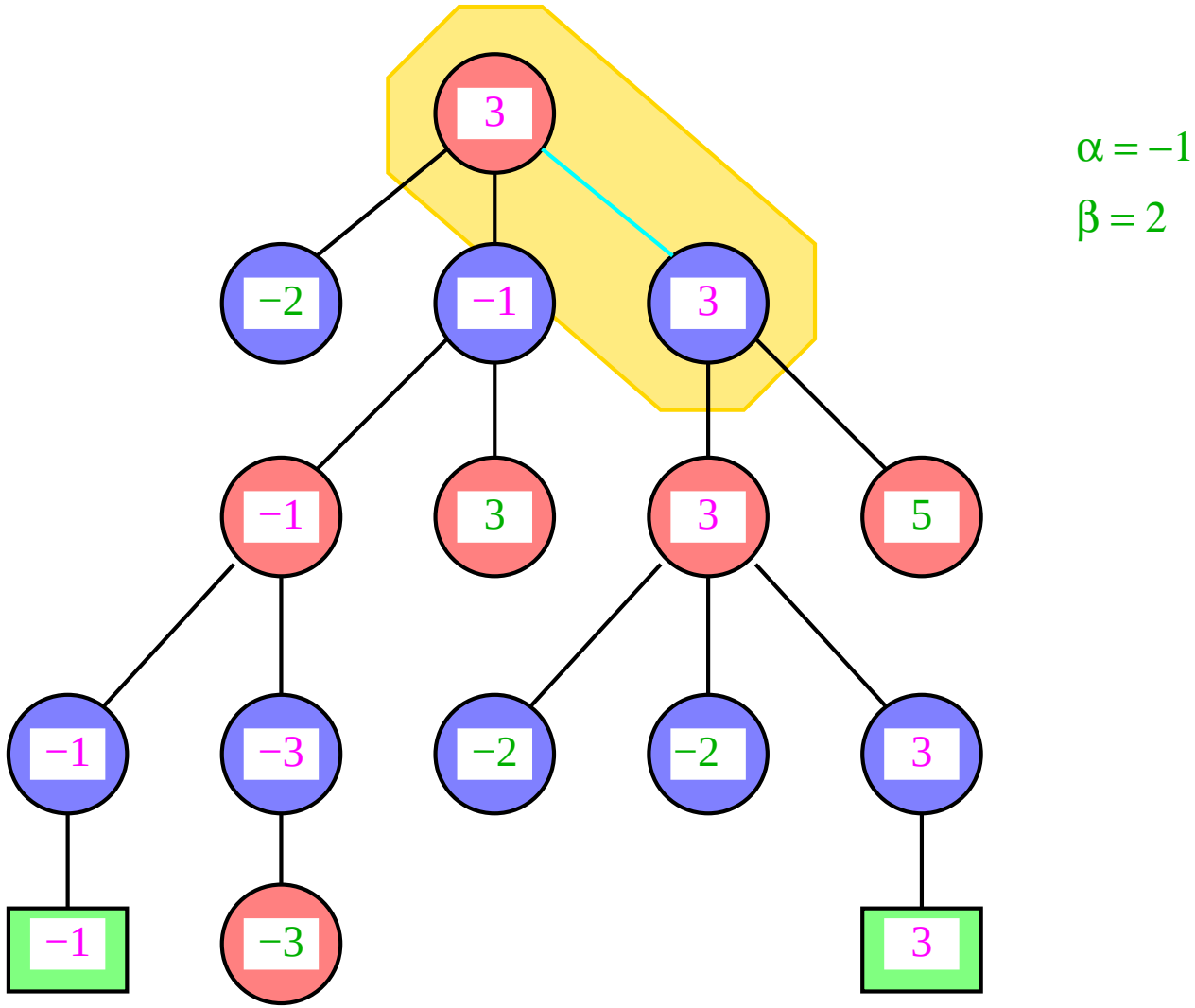
$\beta = 2$


α = −1
β = 2
−2
−1
3
−1
3
3
5
−1
−3
−2
−2
3
−1
−3
3



$\alpha = -1$

$\beta = 2$



$\alpha = -1$

$\beta = 2$

Vorteil:

Die Anzahl der zu untersuchenden Konfigurationen wird (hoffentlich ;-) beträchtlich eingeschränkt!

Nachteil:

Ist die Bewertungsfunktion offensichtlich fehlerhaft, lässt sich das Programm austricksen ...

Frage:

Wie findet man eine Bewertungsfunktion deren Fehlerhaftigkeit nicht so offensichtlich ist???

Ausblick:

- Nicht alle 2-Personen-Spiele sind endlich.
- Gelegentlich hängt der Effekt eines Zugs zusätzlich vom Zufall ab.
- Eventuell ist die aktuelle Konfiguration nur partiell bekannt.

$\Longrightarrow$ Spieltheorie