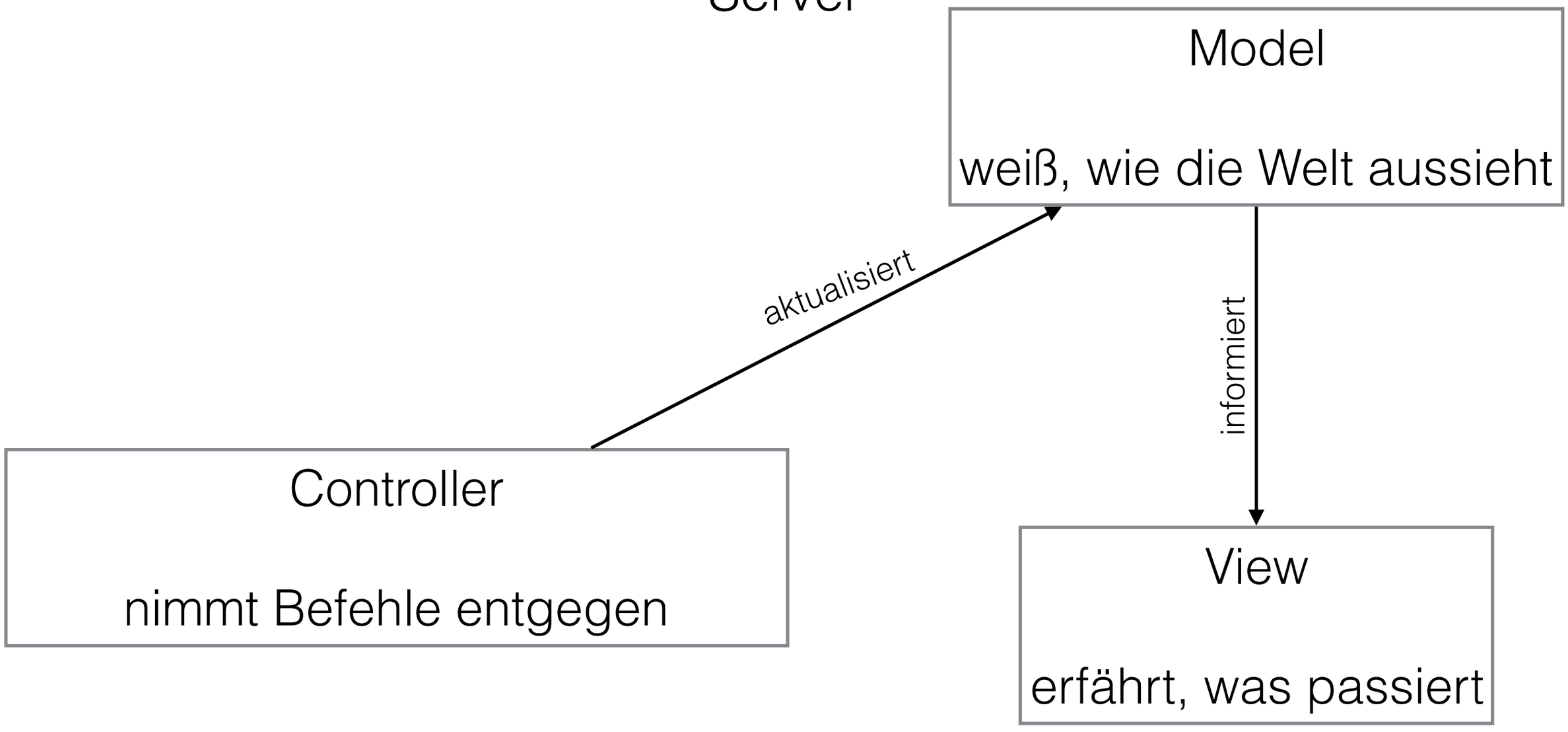
Softwarearchitektur eines Computerspielers

SoPra 2016

Alexander Kampmann

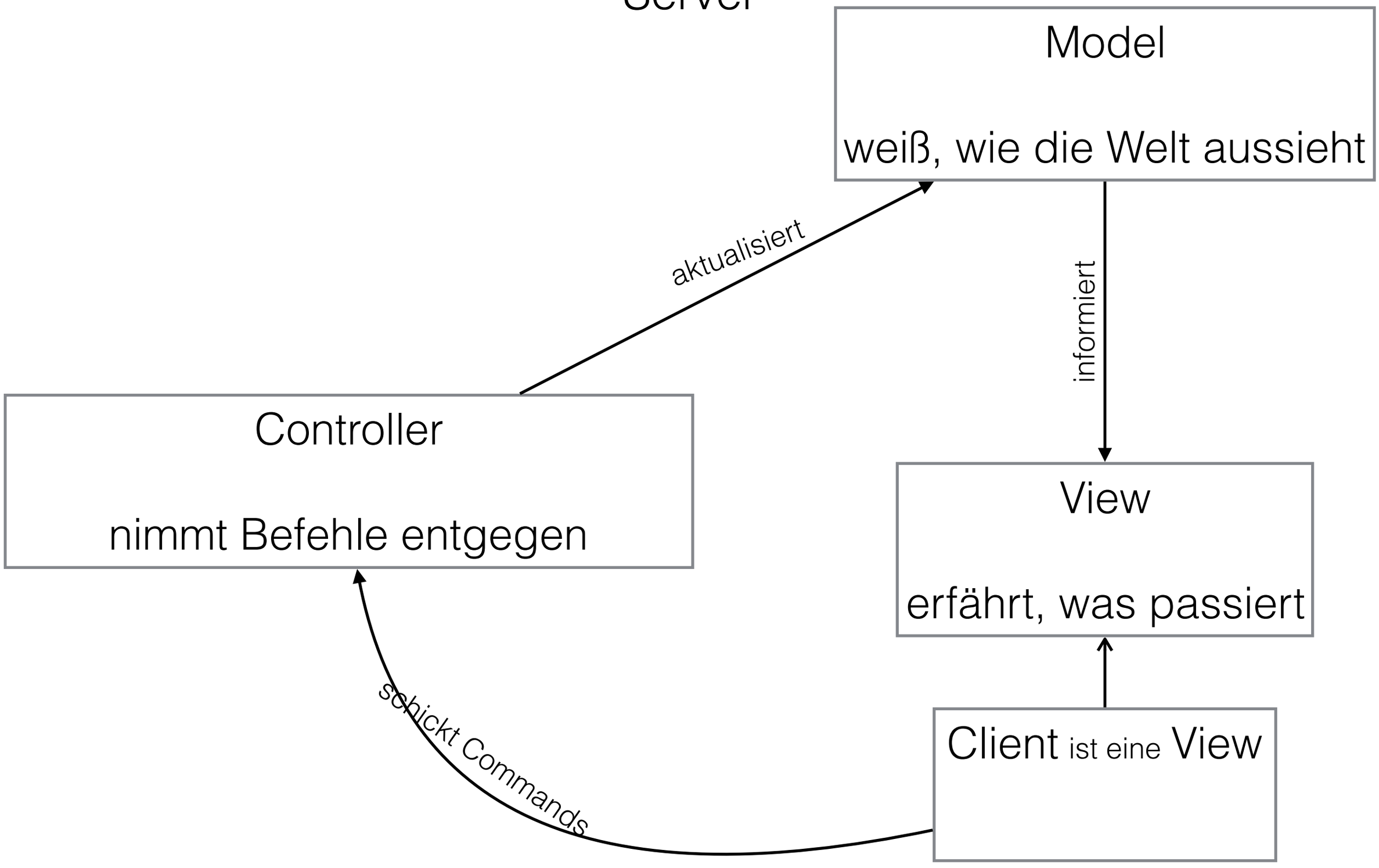
Model - View -Controller

Server



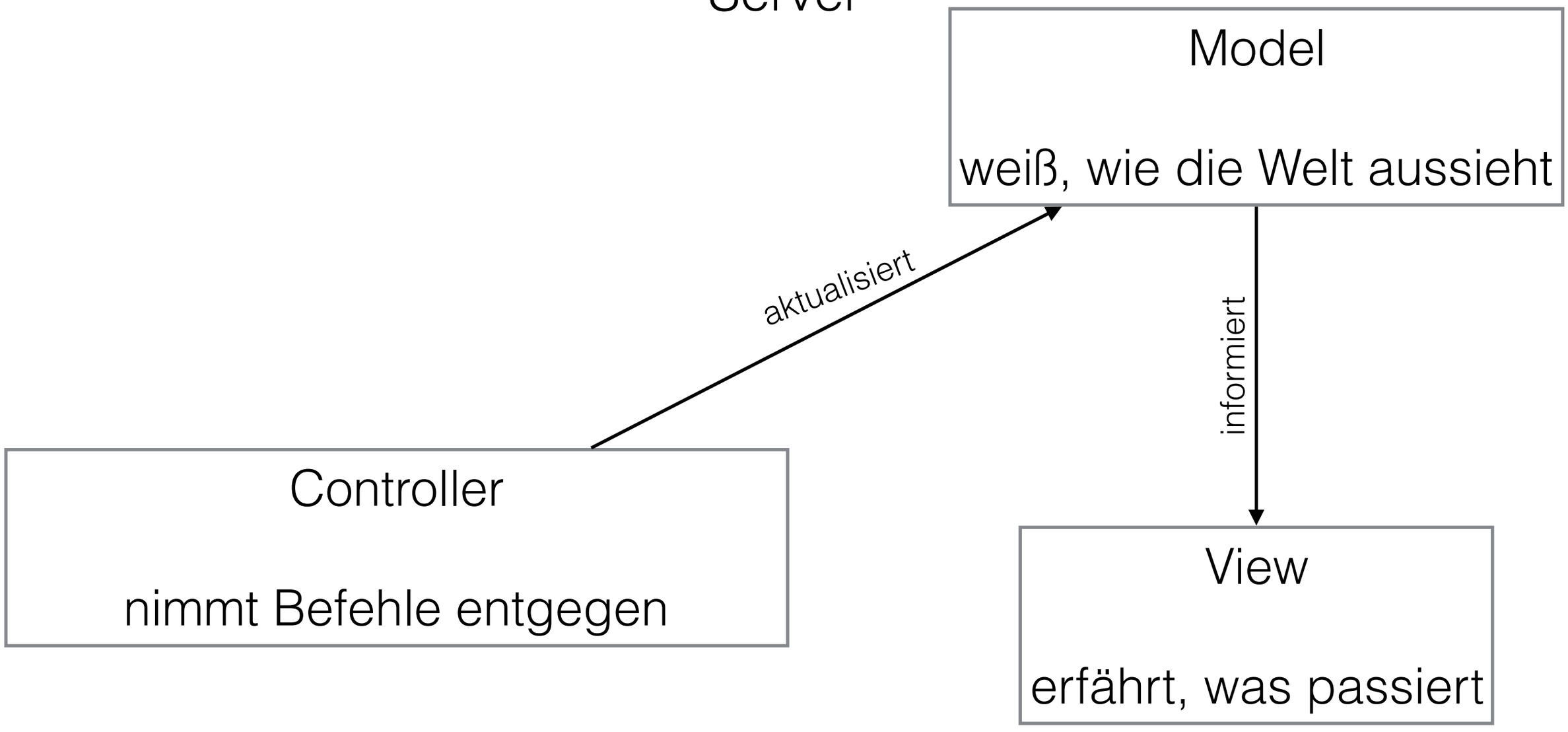
Model - View -Controller

Server

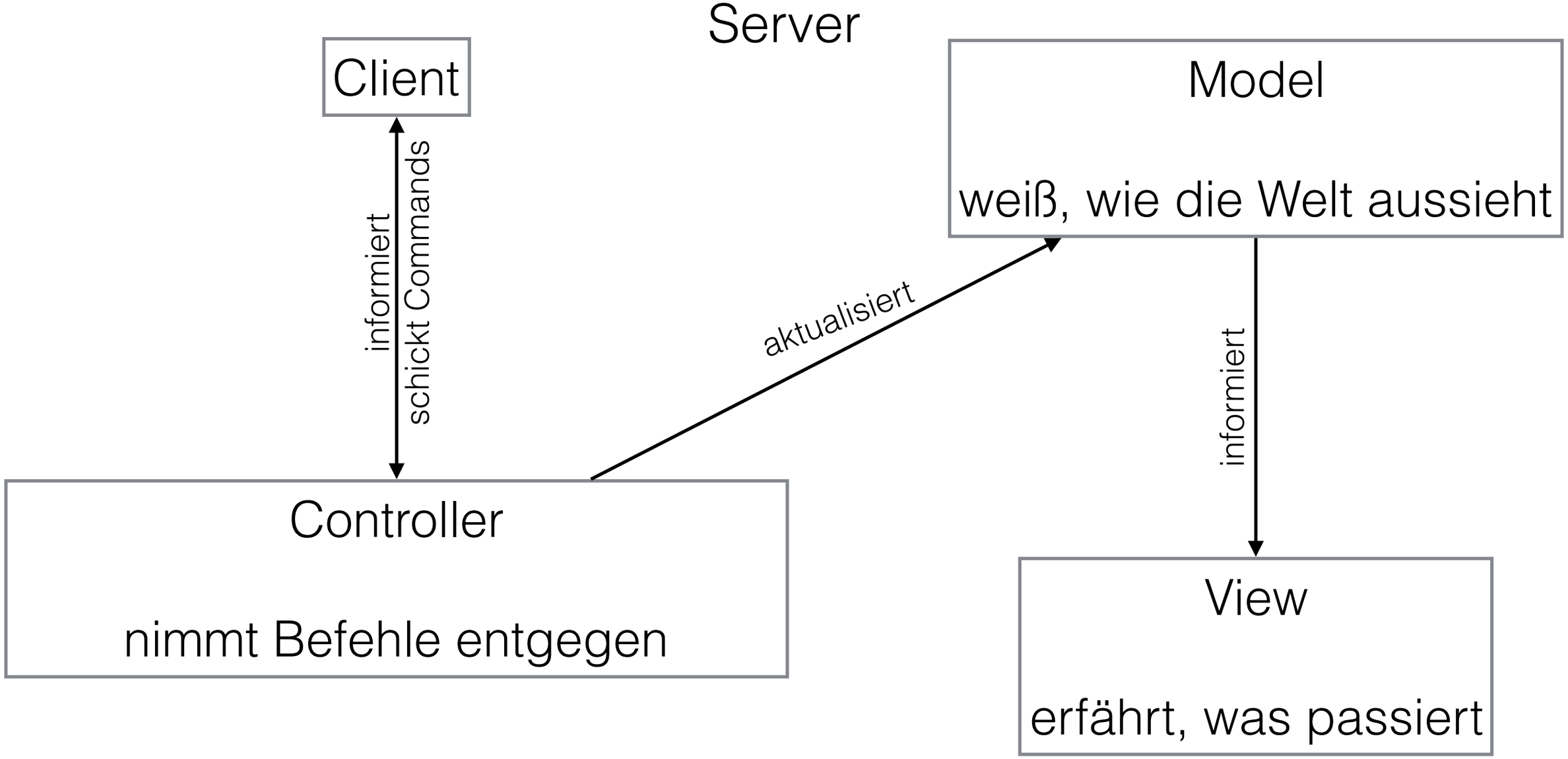


Model - View -Controller

Server

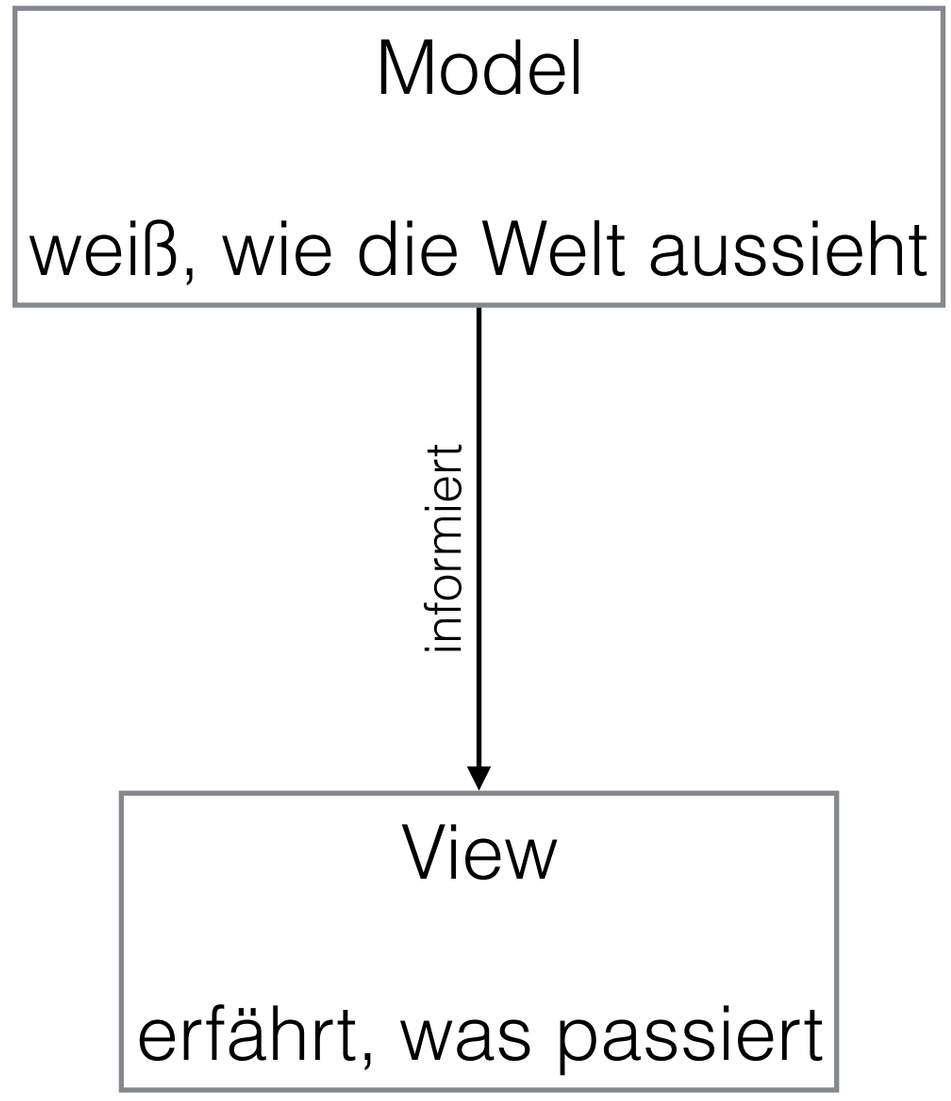


Model - View -Controller



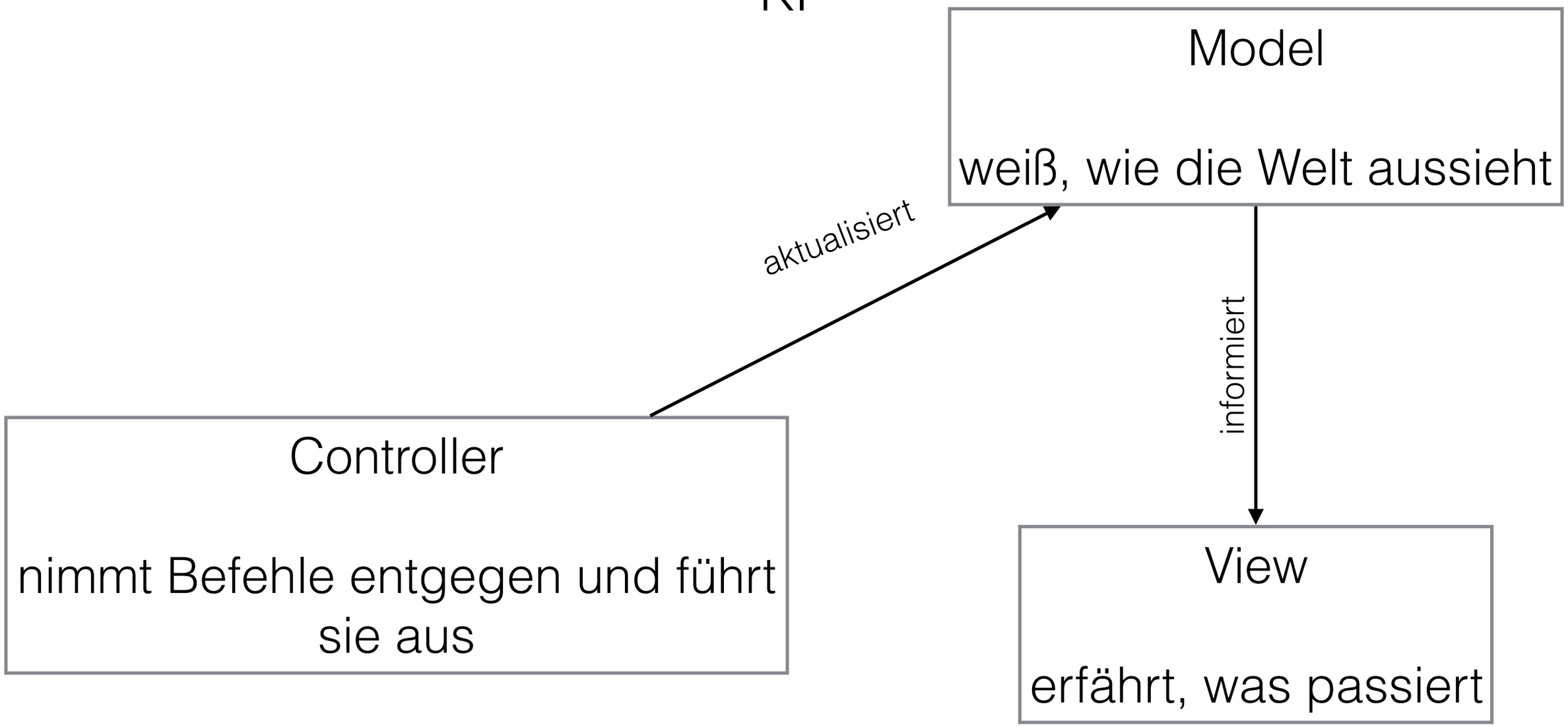
Model - View -Controller

KI



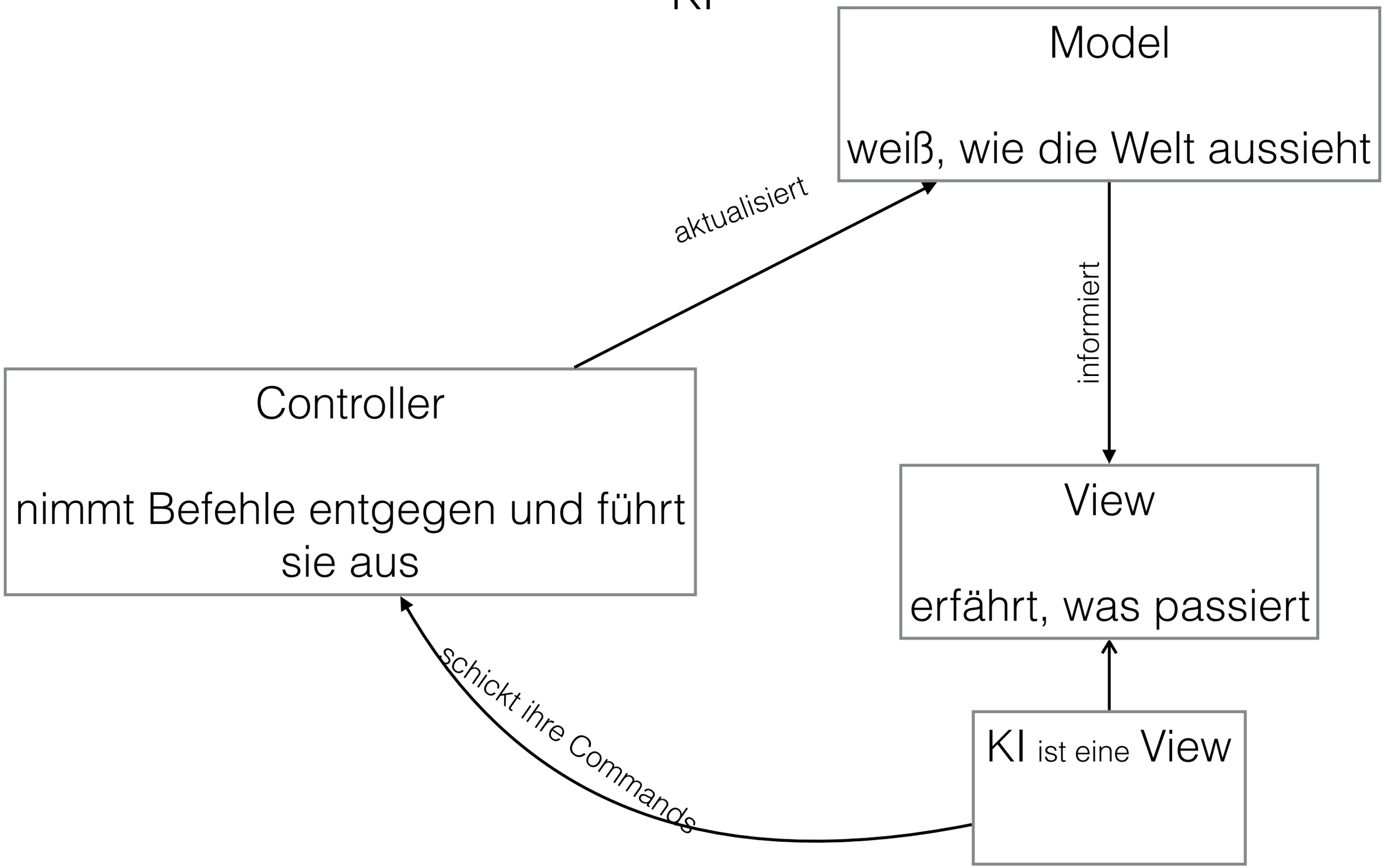
Model - View -Controller

KI



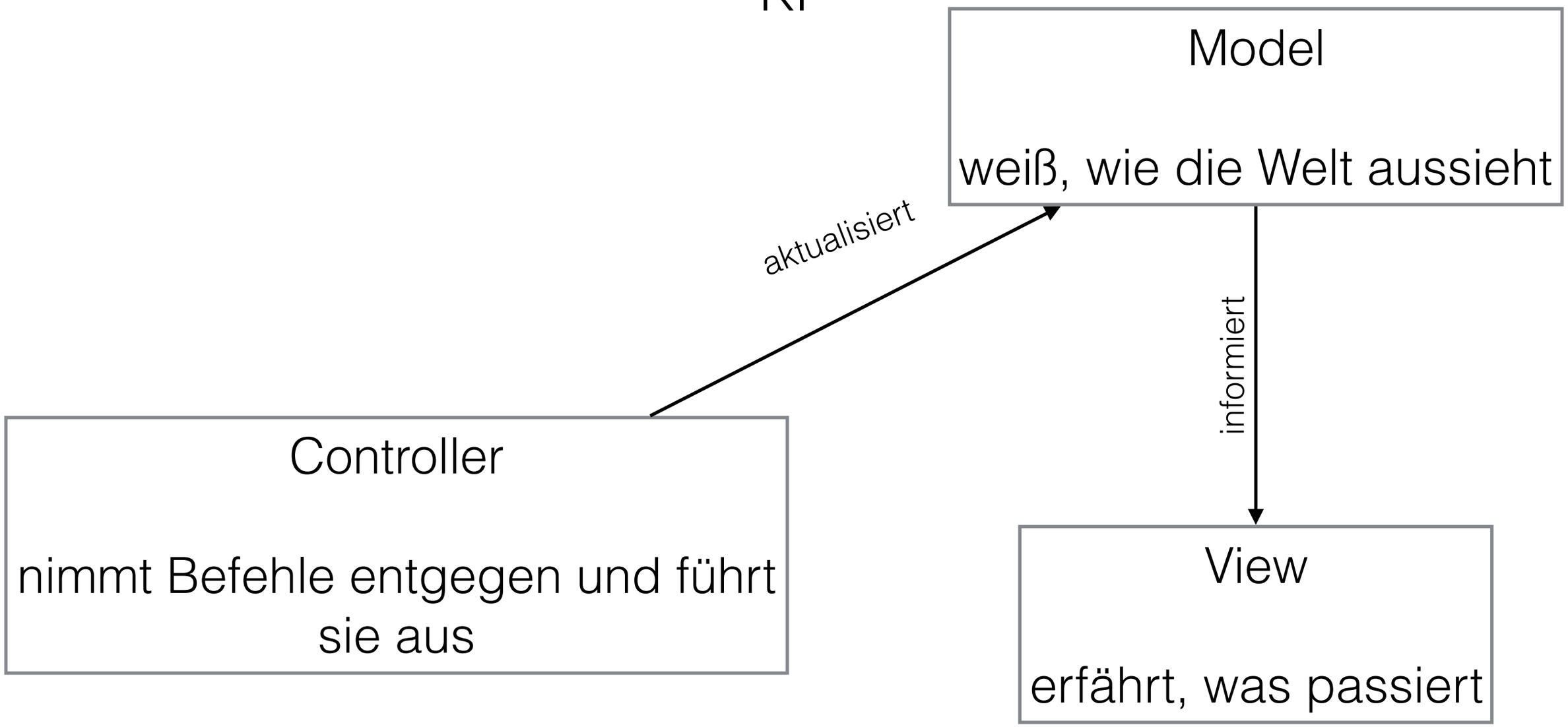
Model - View -Controller

KI

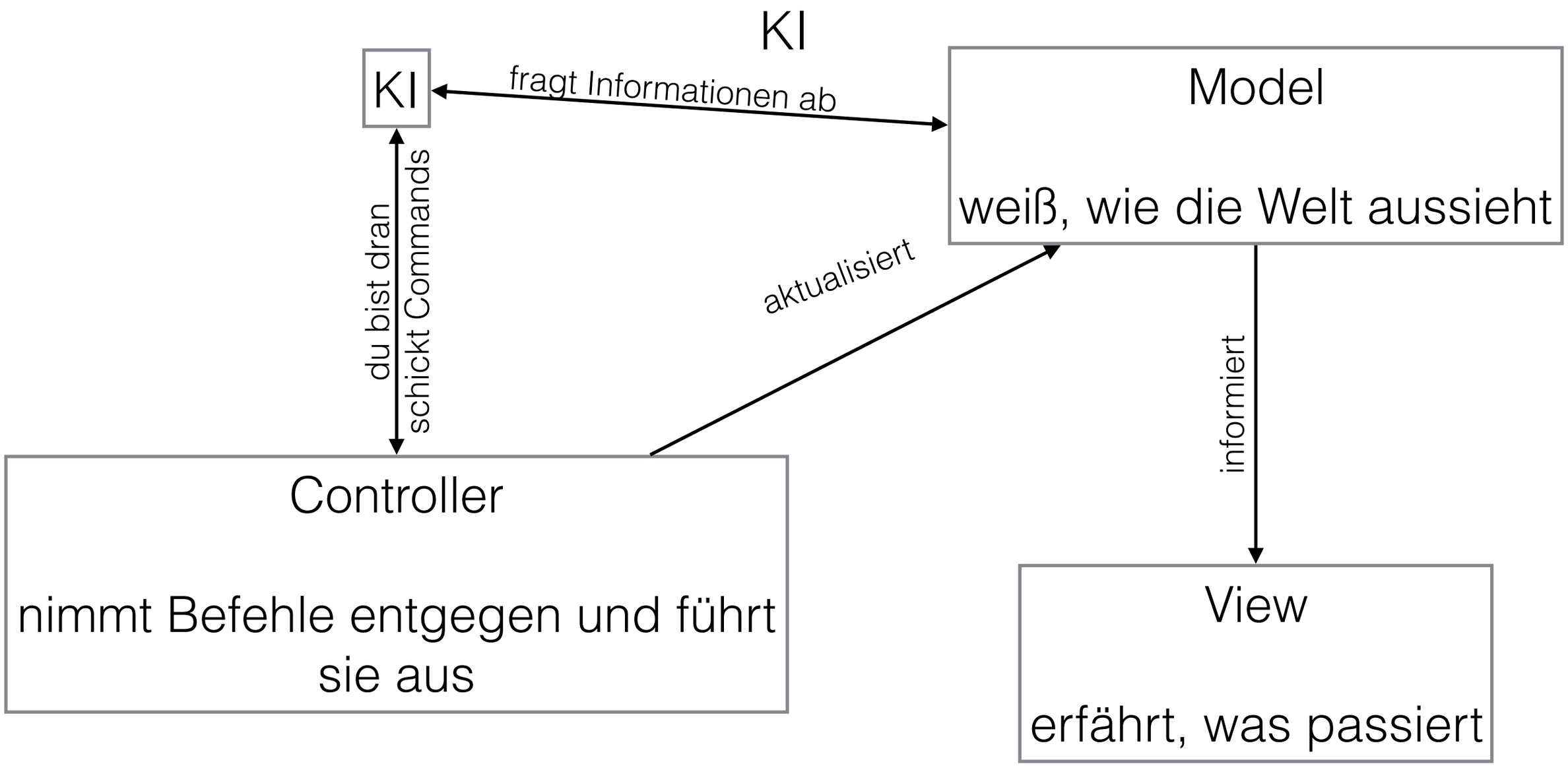


Model - View -Controller

KI



Model - View -Controller



Der Zufall spielt

```
   // Zufallsobjekt erzeugen:
   Random r = new Random();

// alle Richtungen testen und eine wählen
   for(Direction d : Direction.values()) {
      if(mayMove(d) && r.nextBoolean()) {
         server.sendMove(d);
         return;
      }
   }

// dasselbe für alle anderen Commands
```

Der Zufall spielt

```
    // Zufallsobjekt erzeugen:
    Random r = new Random();

// alle Richtungen testen und eine wählen
    for(Direction d : Direction.values()) {
      if(mayMove(d) && r.nextBoolean()) {
        server.sendMove(d);
        return;
      }
    }

// dasselbe für alle anderen Commands
```

Mit Command-Pattern

```
// Zufallsobjekt erzeugen:
Random r = new Random();

// alle Möglichkeiten auflisten
List<Command> commands =
getPossibleCommands(model, getEntityId());

// eine aussuchen
int pos = r.nextInt(commands.size());
Command command = commands.get(pos);

// und an den Server senden
send(command);
```

Mit Command-Pattern

```
// Zufallsobjekt erzeugen:
Random r = new Random();

// alle Möglichkeiten auflisten
List<Command> commands =
getPossibleCommands(model, getEntityId());

// eine aussuchen
int pos = r.nextInt(commands.size());
Command command = commands.get(pos);

// und an den Server senden
send(command);
```

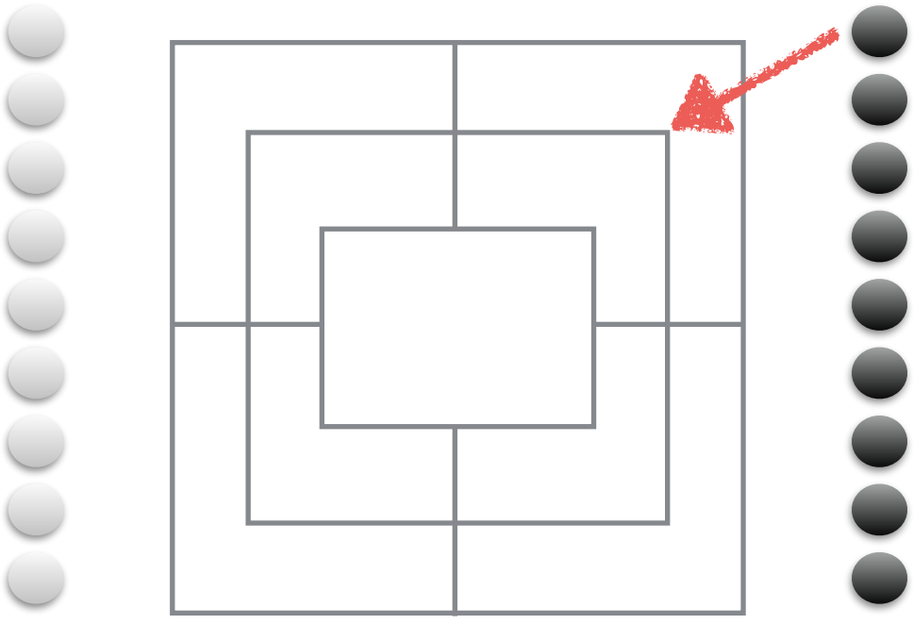
Command-Pattern

Anforderungen an das Model

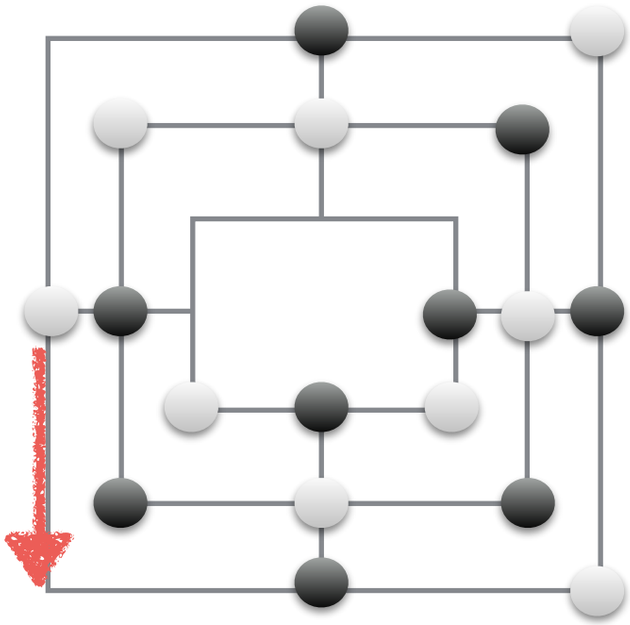
- Ich muss informiert werden, wenn ich dran bin.
- Ich muss prüfen können, ob ein Zug zulässig ist.
- Ich muss mitteilen können, was ich machen möchte.
- Ich muss abfragen können, wie die Spielwelt aussieht.

Mühle

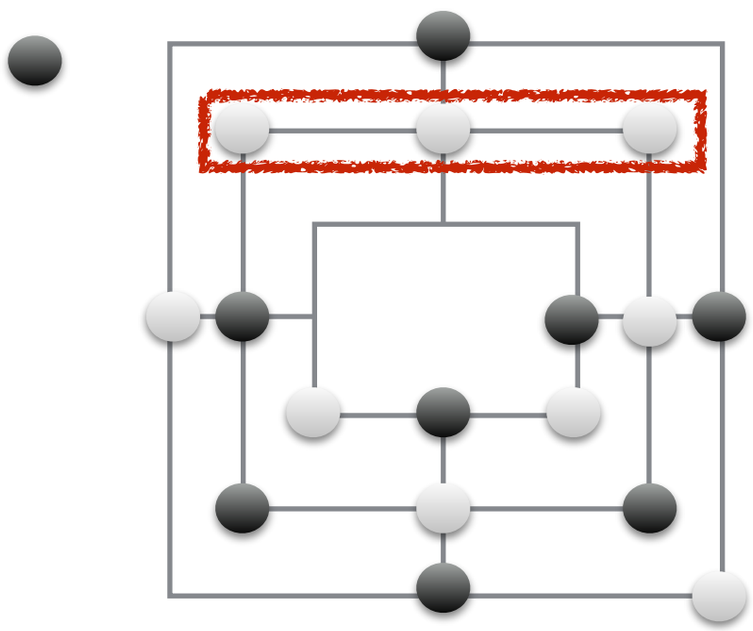
1. Phase: Steine setzen



2. Phase: Steine ziehen

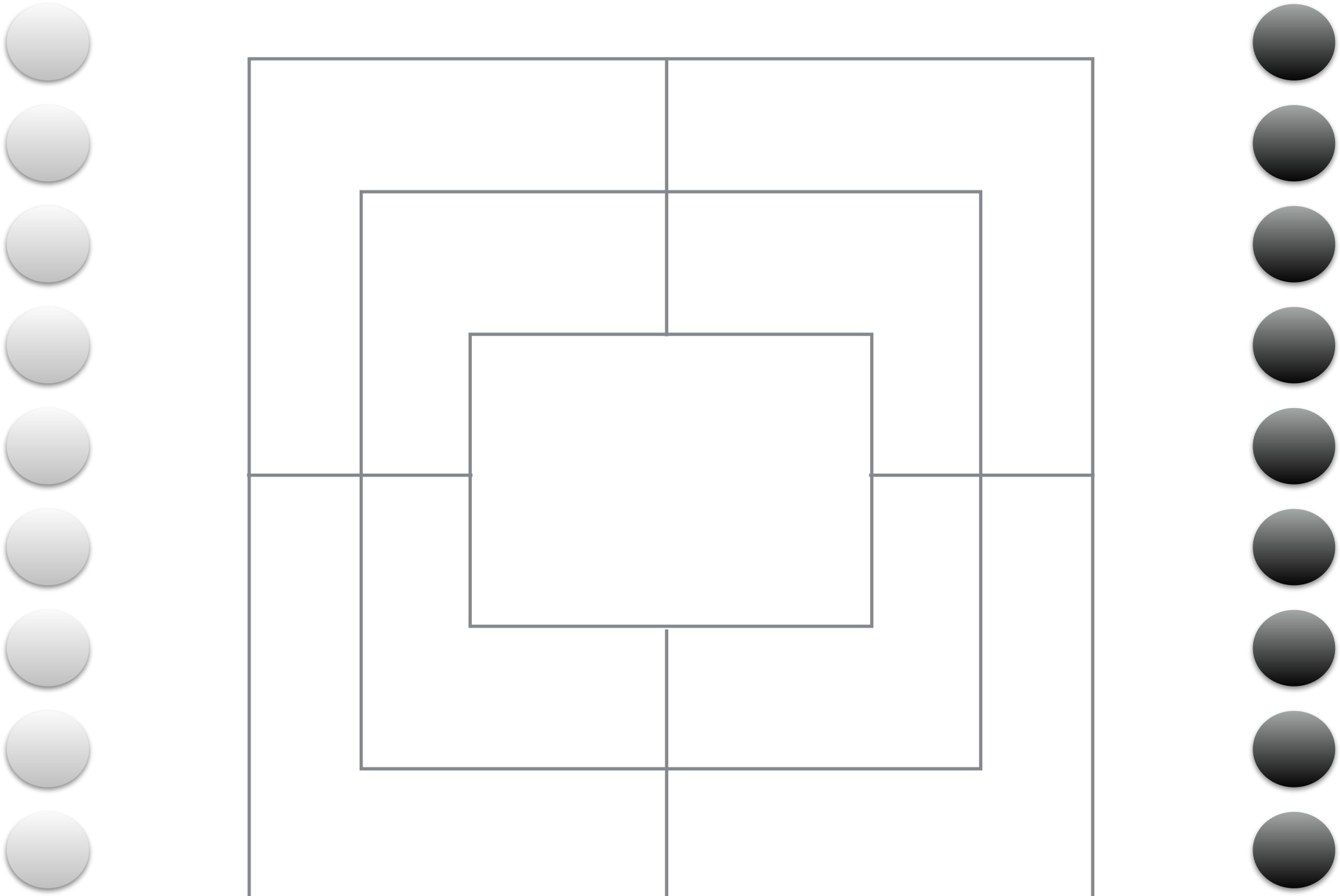


Mühle: Stein klauen



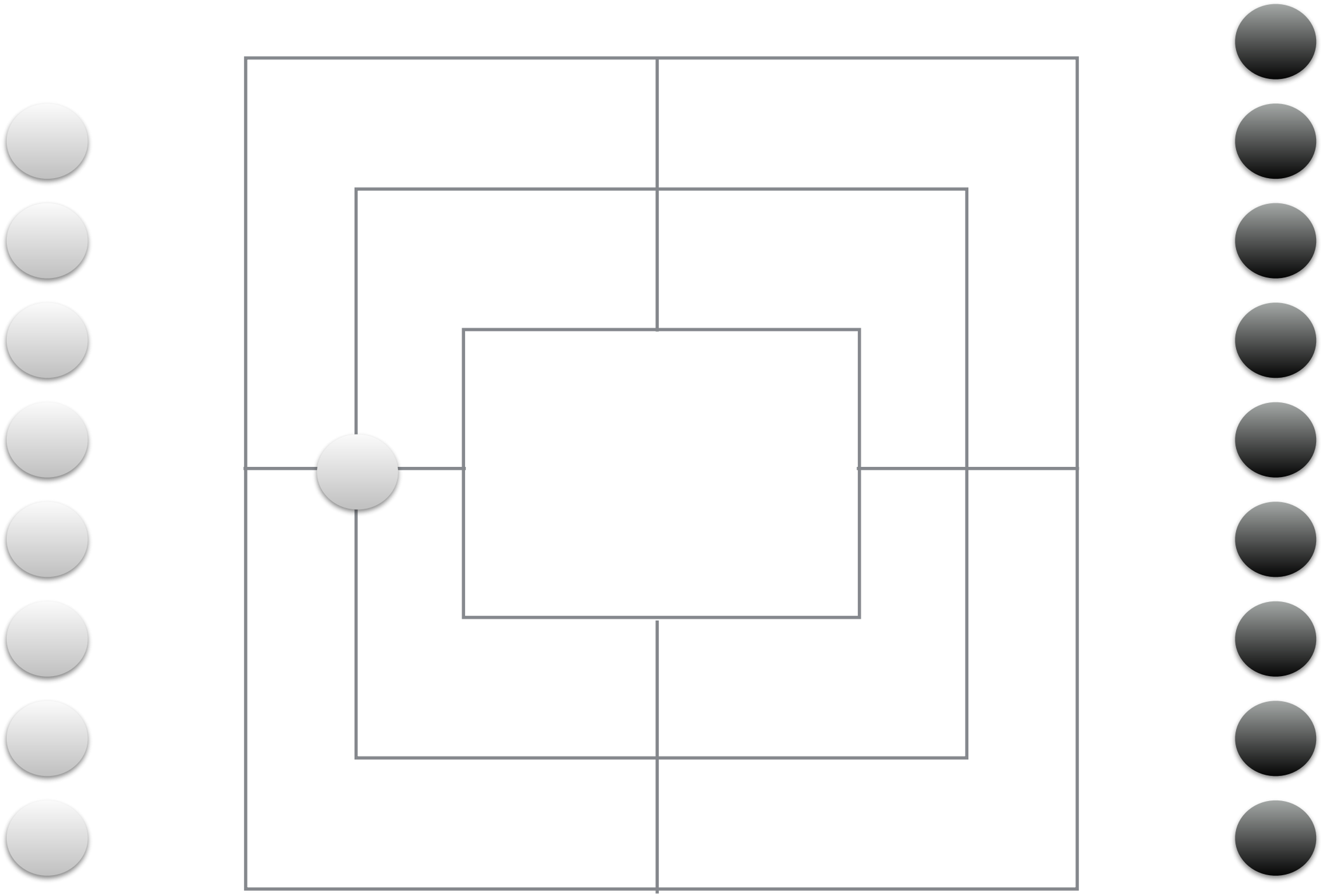
Regel-basierter Computerspieler

- In der Setzen-Phase:
  - Wenn ich zwei Steine in einer Reihe habe, setze ich den Dritten
  - Wenn der Gegner zwei Steine in einer Reihe hat, setze ich auf das dritte Feld
  - Wenn eins der Felder mit vier Nachbarn frei ist, setze ich da hin
  - …



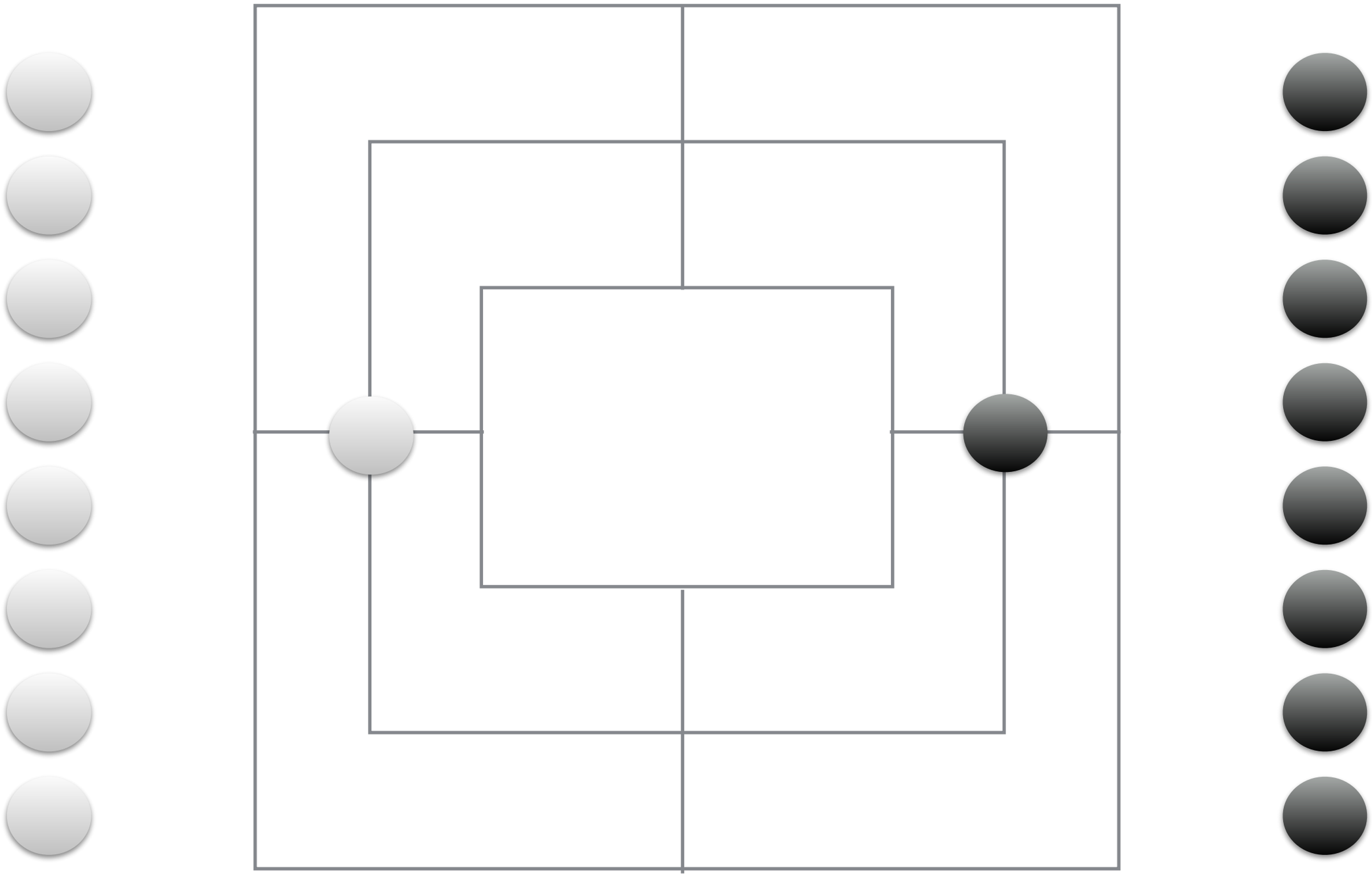
Mensch

Regelbasiert



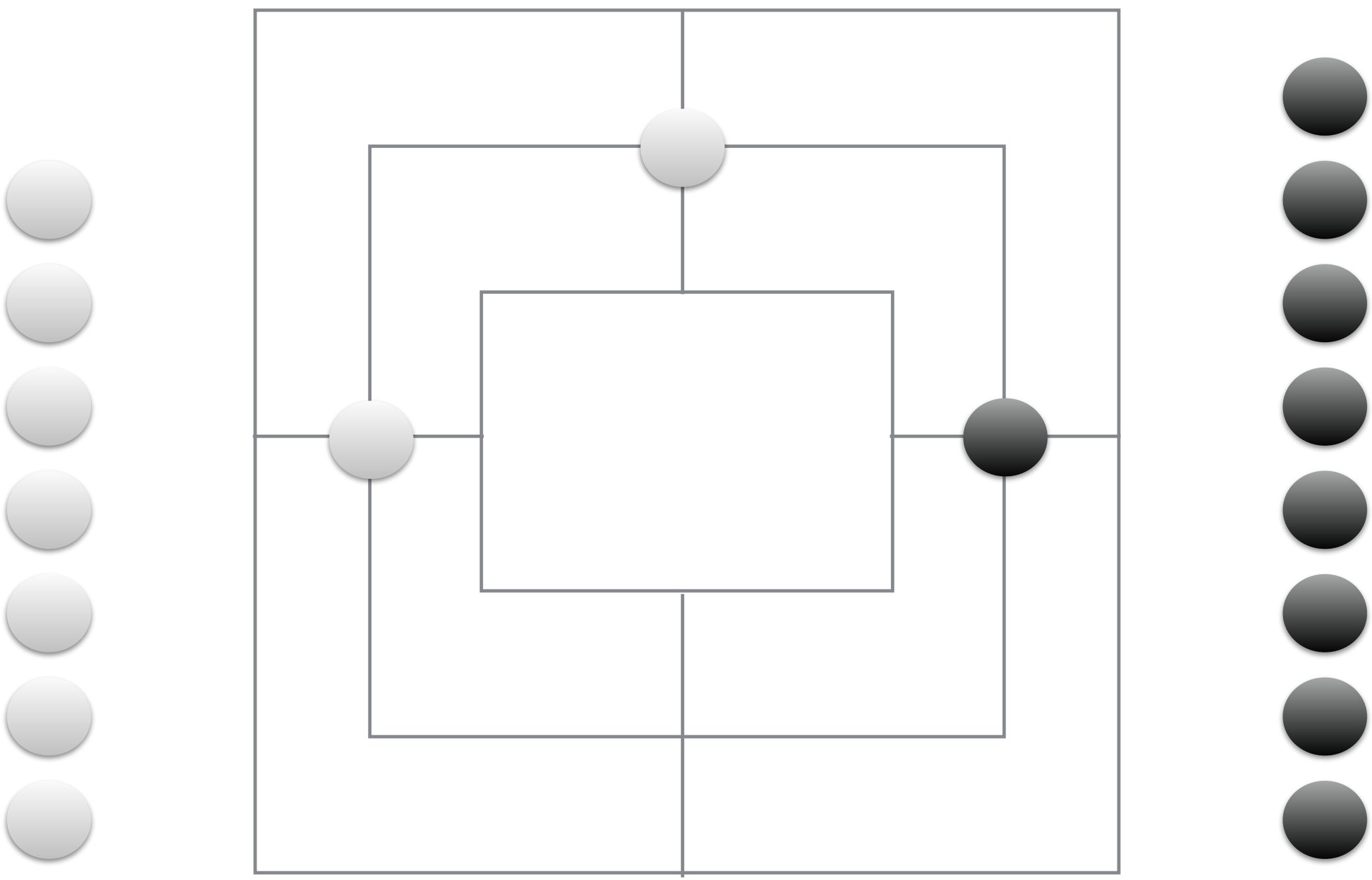
Mensch

Regelbasiert



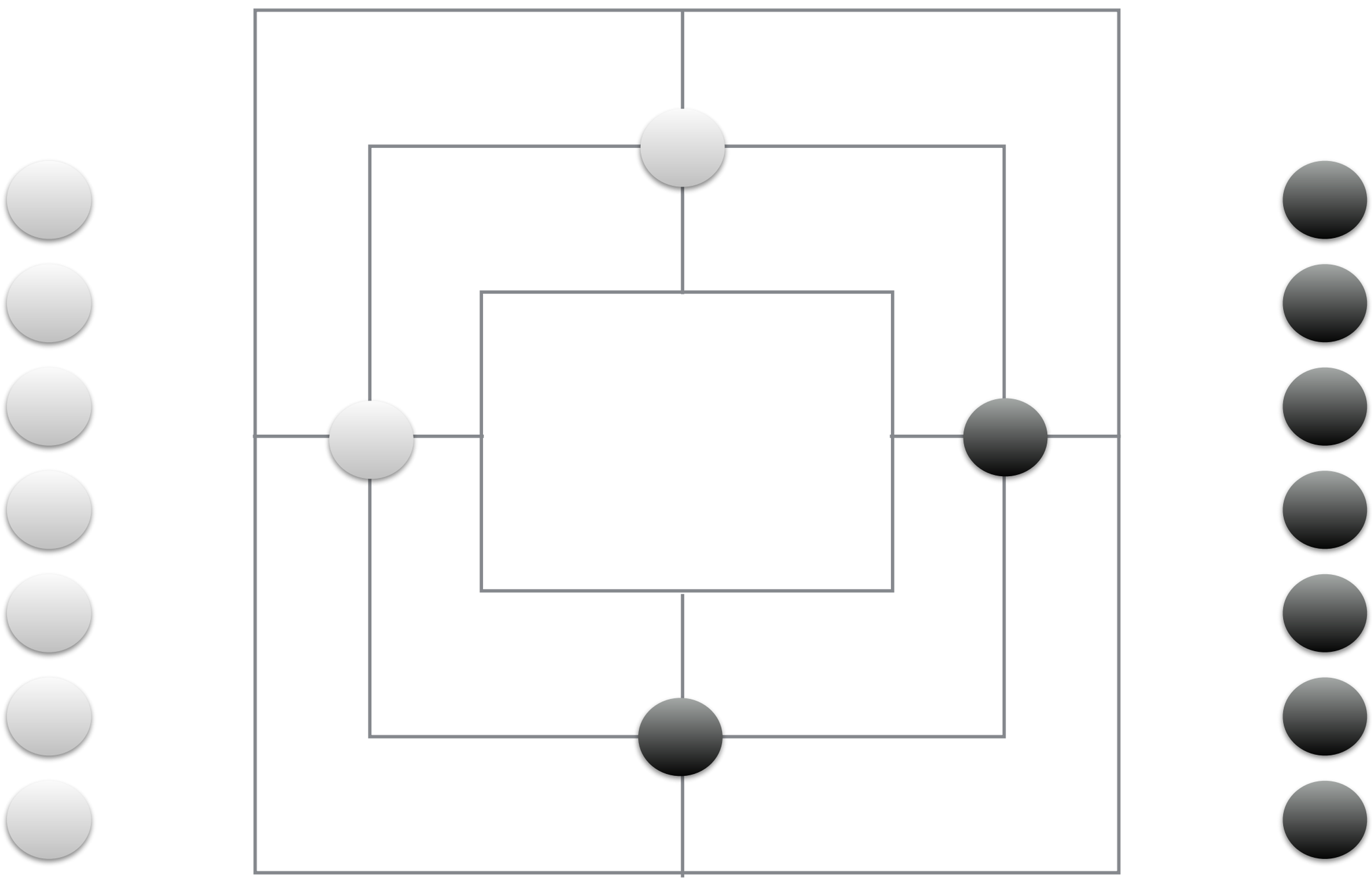
Mensch

Regelbasiert



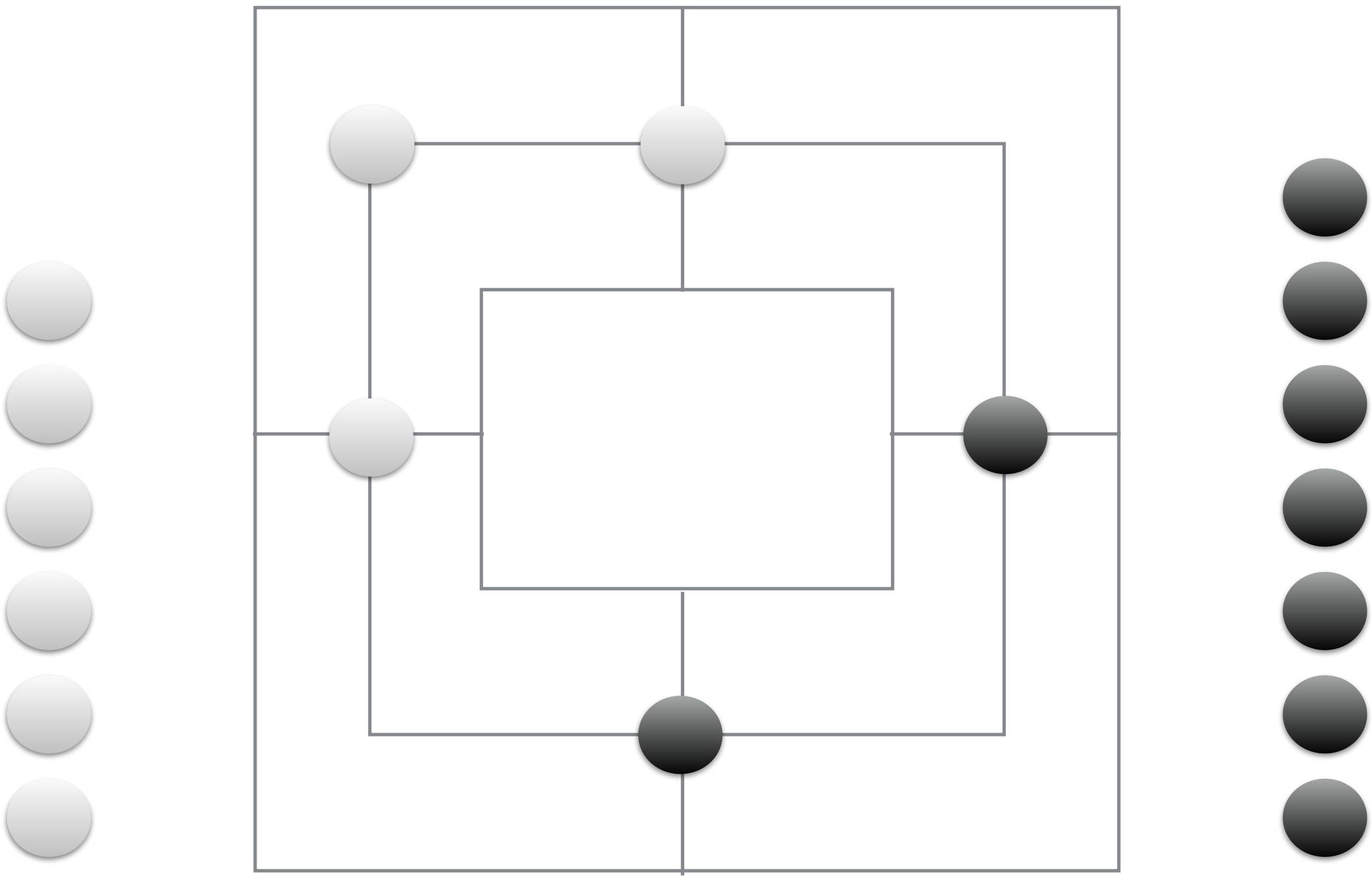
Mensch

Regelbasiert



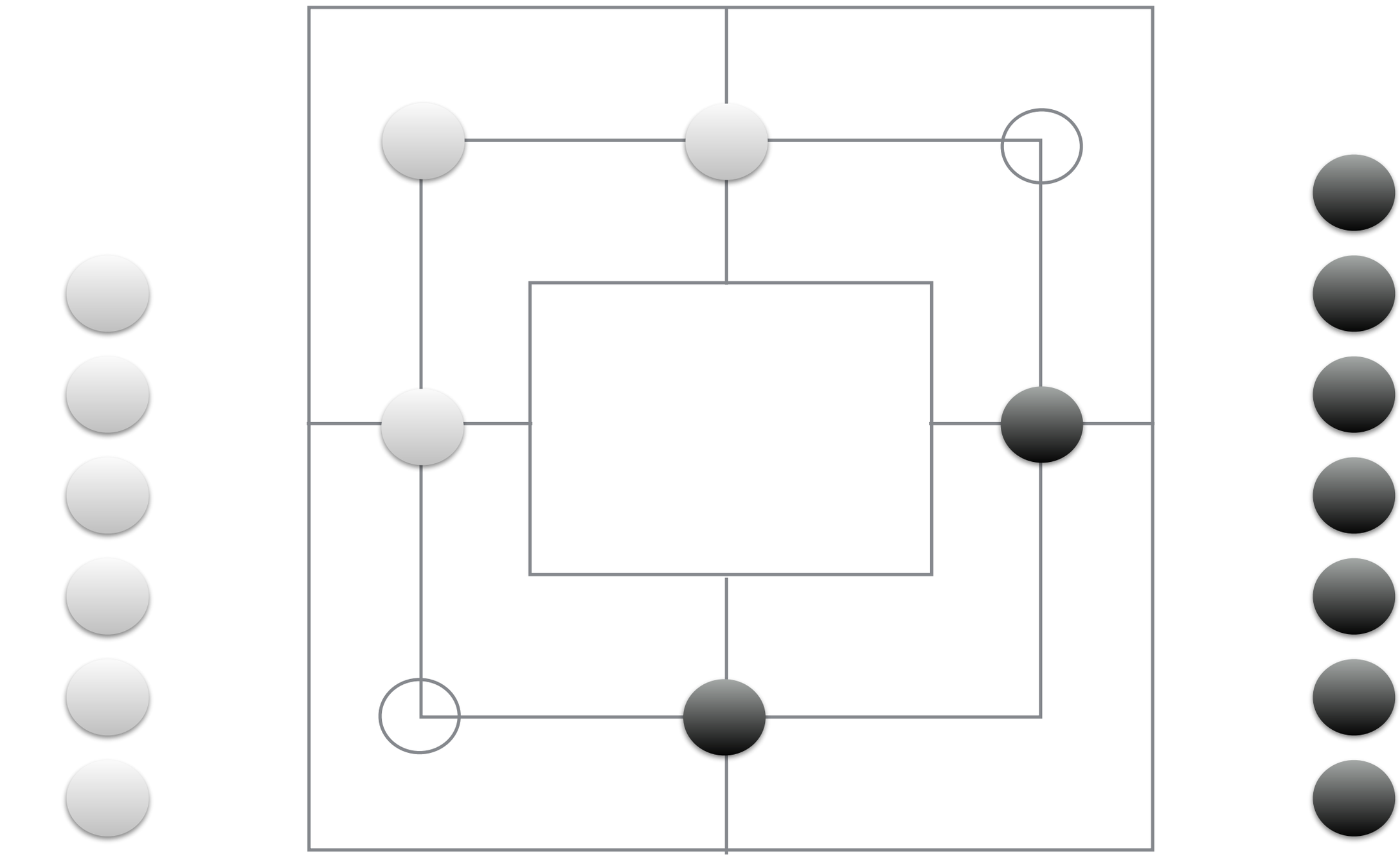
Mensch

Regelbasiert



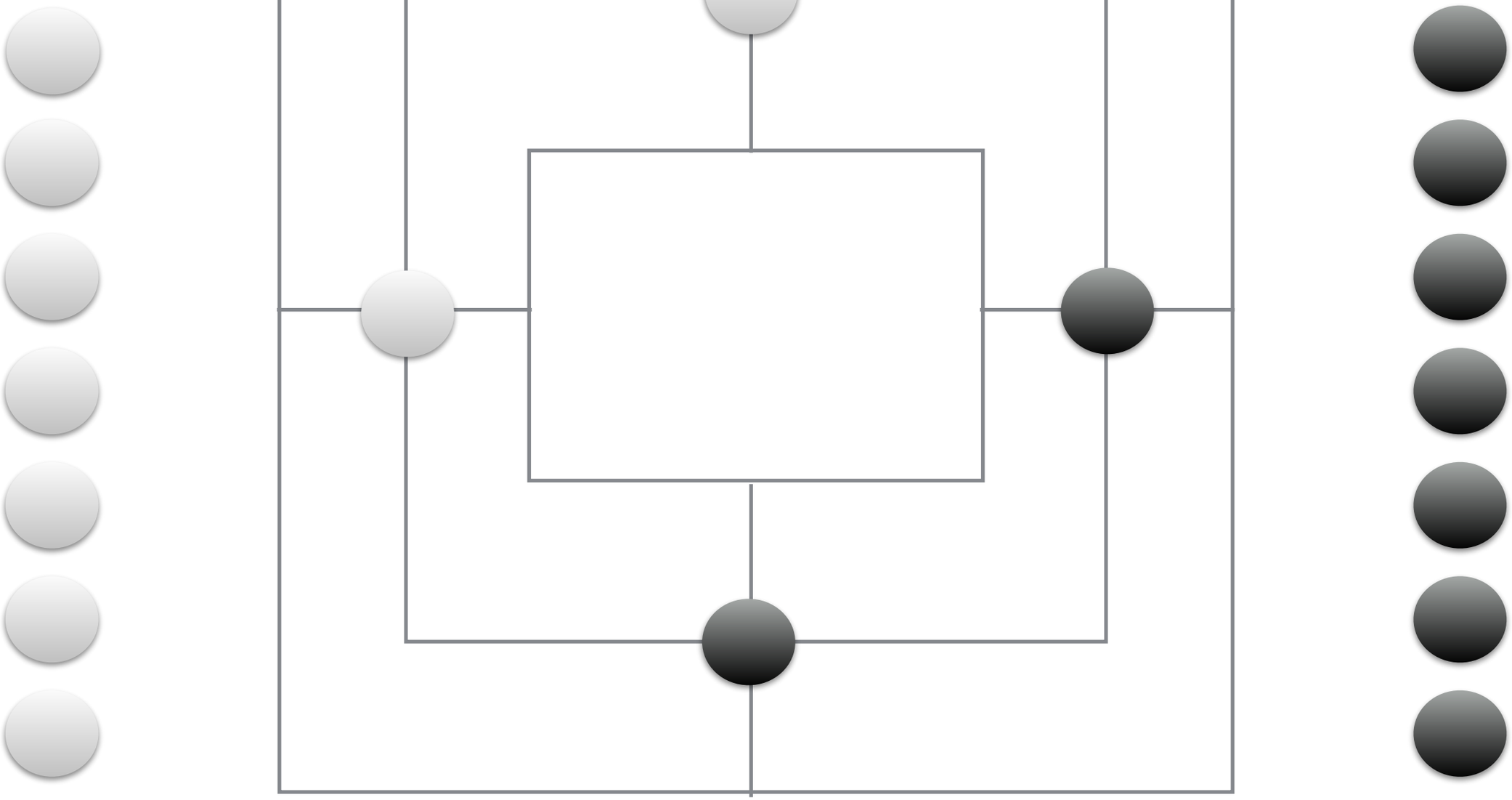
Mensch

Regelbasiert



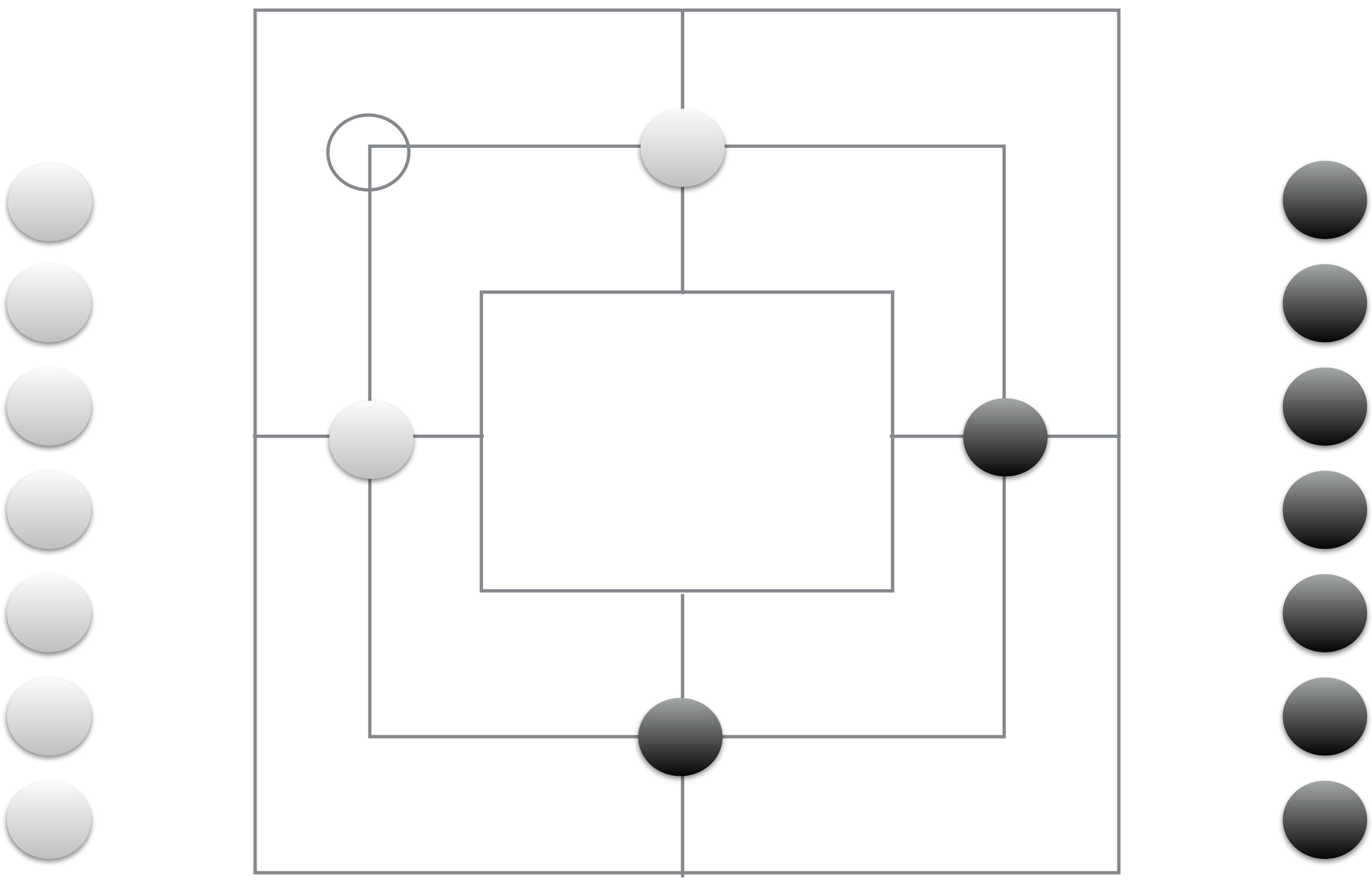
Mensch

Regelbasiert



Mensch

Regelbasiert

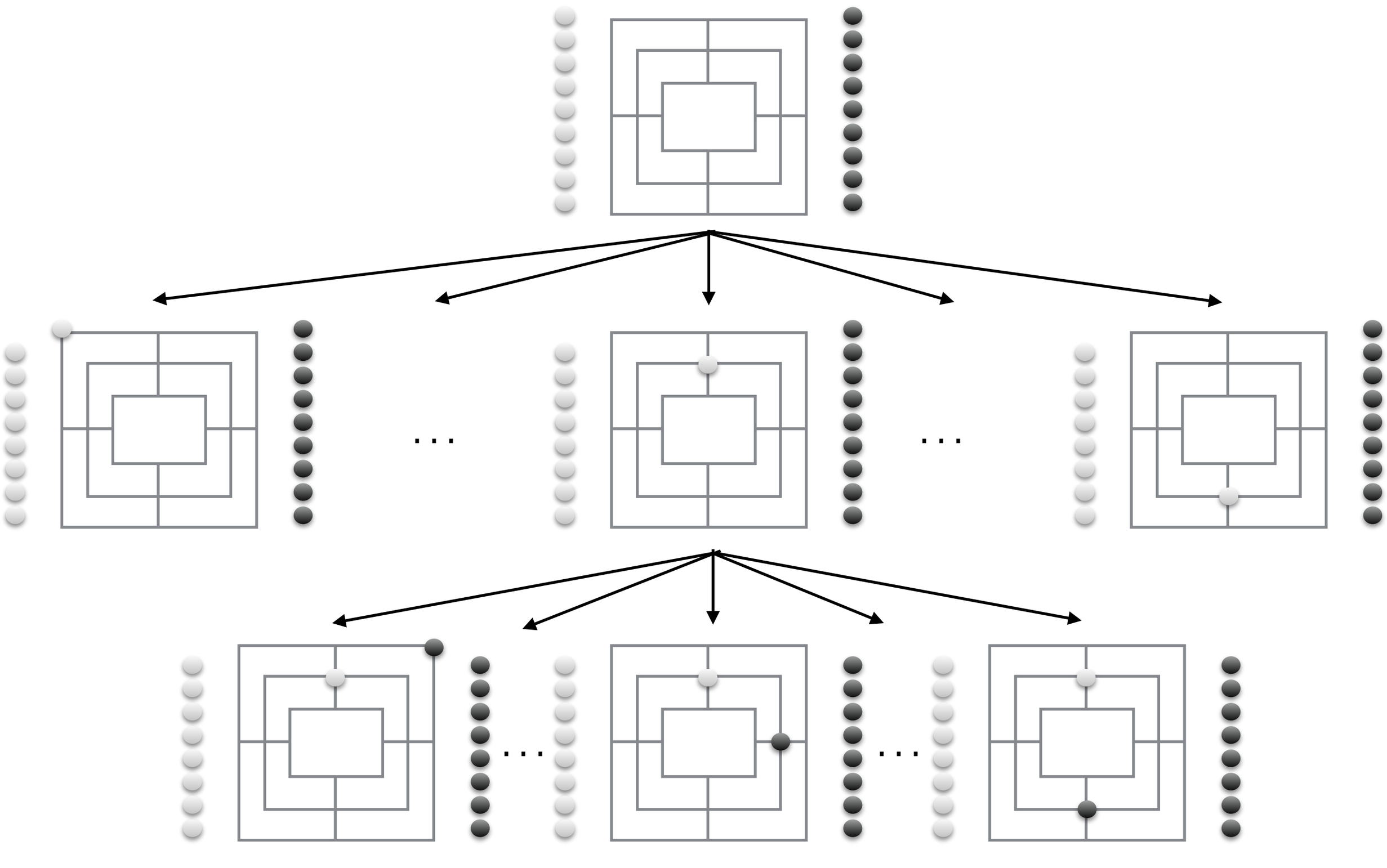


Mensch

Regelbasiert

Der Computer muss lernen, die Pläne seines Gegners zu erahnen

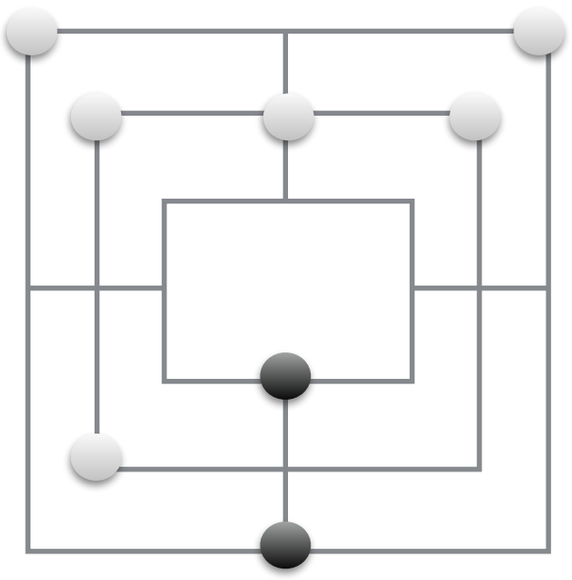
Baumsuche



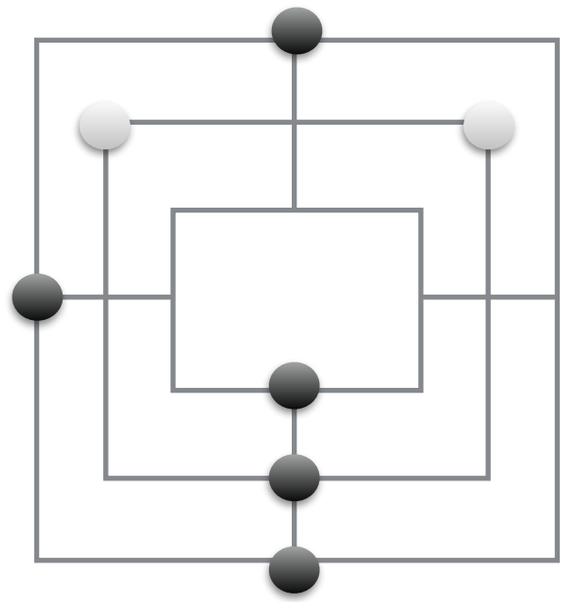
Wie gut ist ein Zustand?

(Für weiß)

Sehr gut



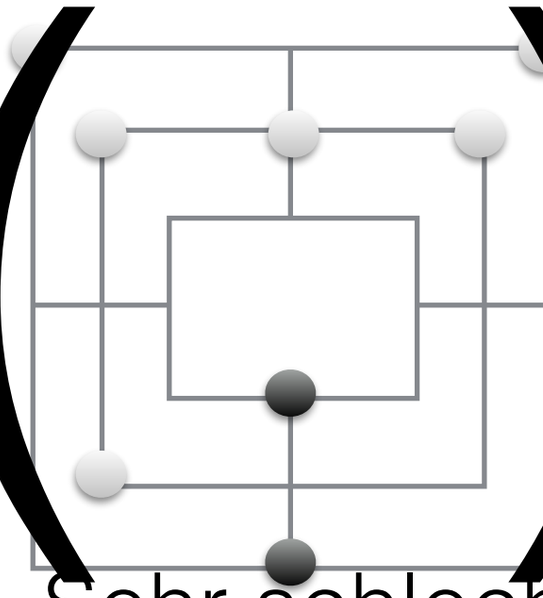
Sehr schlecht



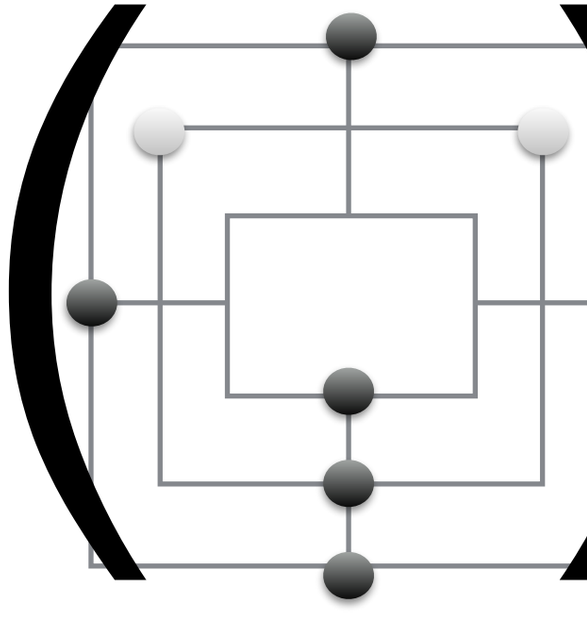
Wie gut ist ein Zustand?

(Für weiß)

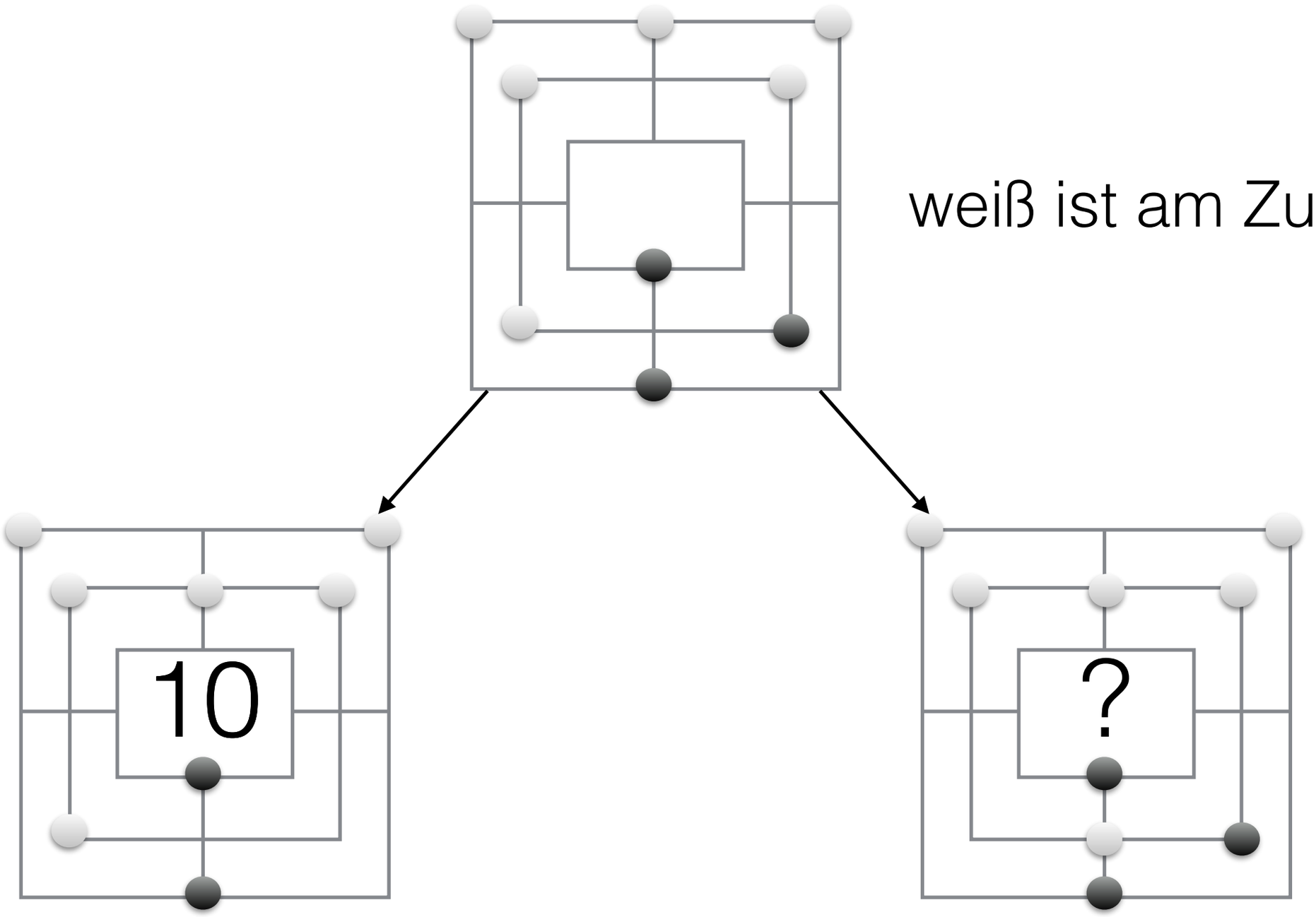
Sehr gut

h() = 10

Sehr schlecht

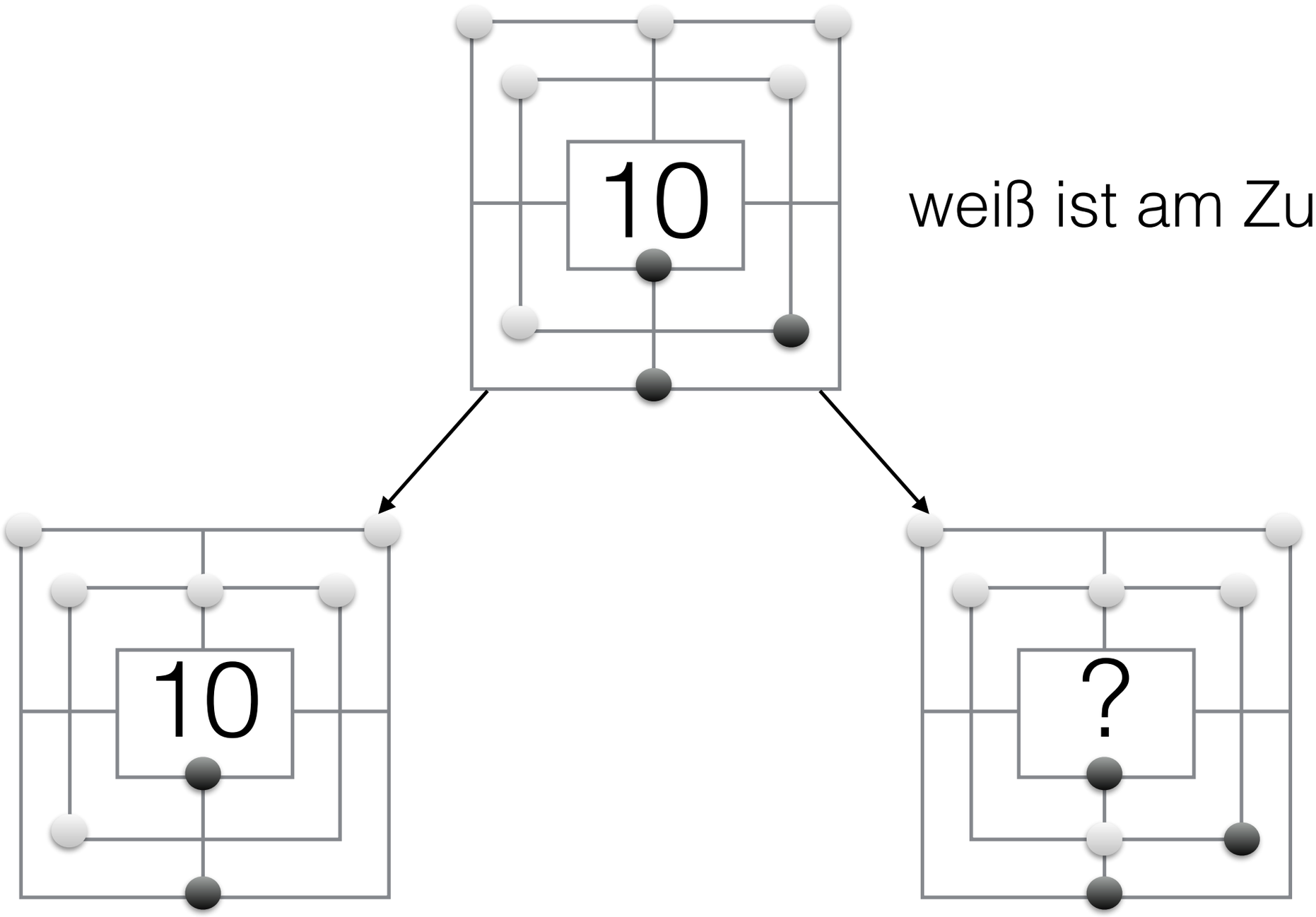
h() = 0

Baumsuche



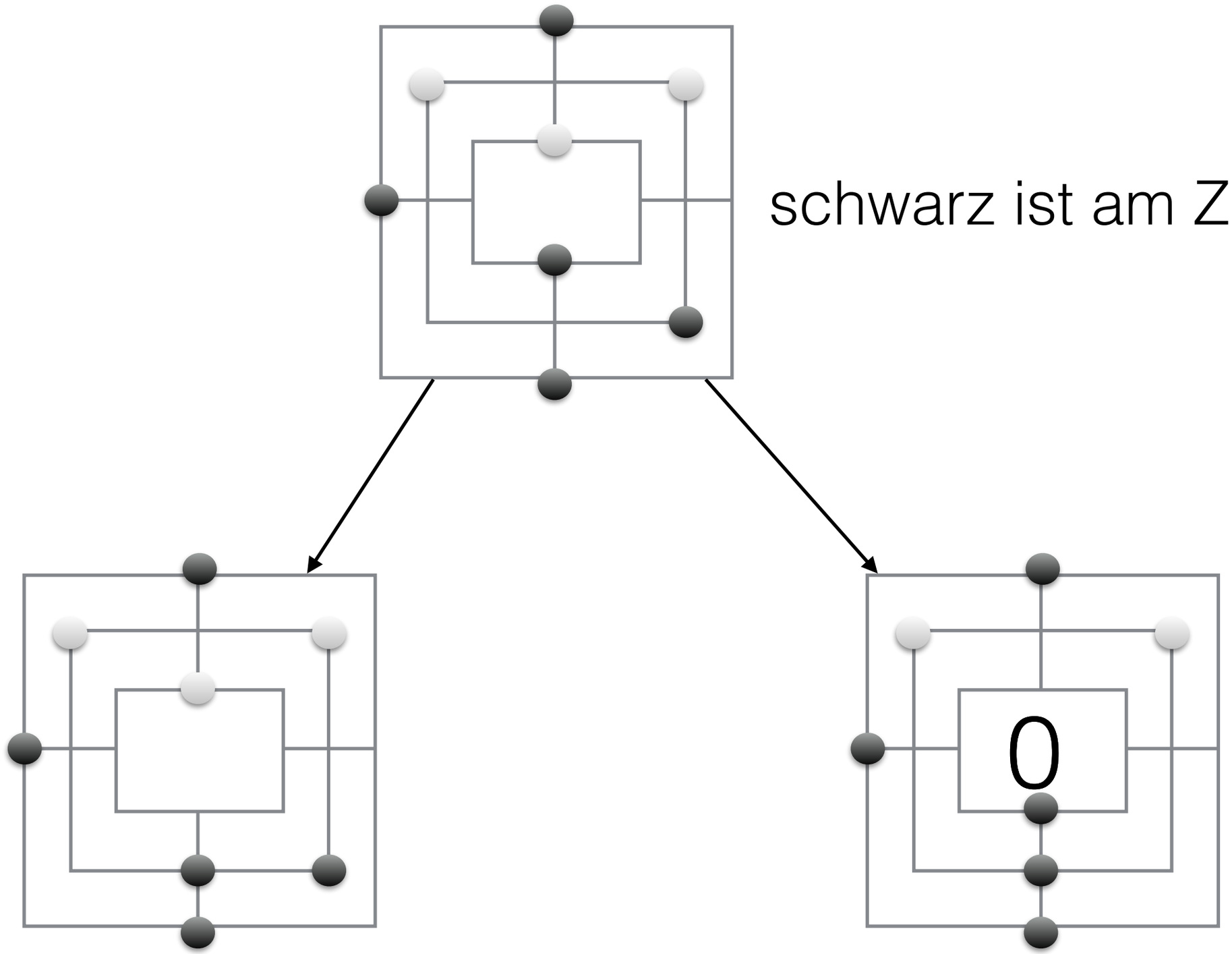
weiß ist am Zug

Baumsuche



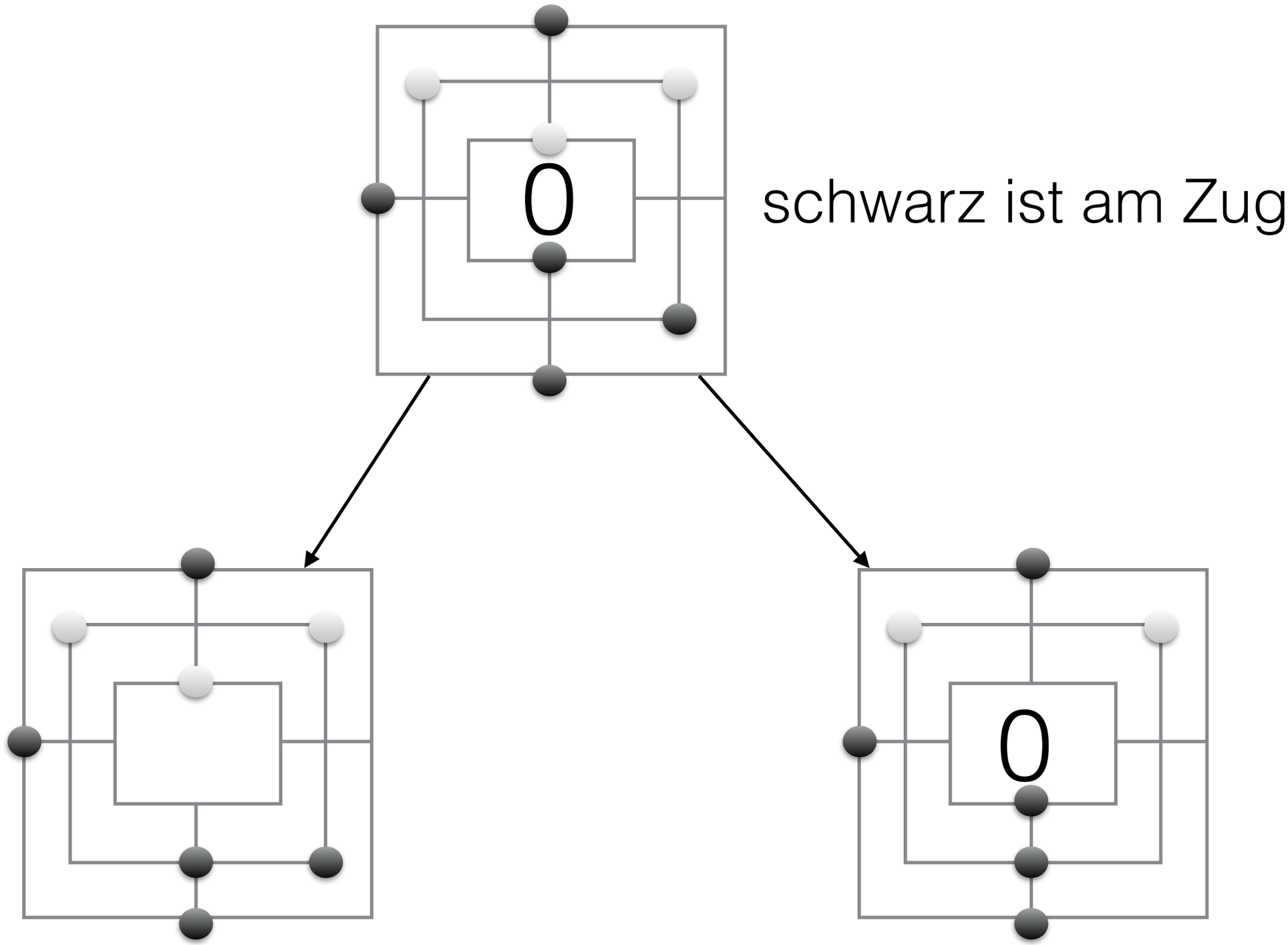
weiß ist am Zug

Baumsuche



schwarz ist am Zug

Baumsuche

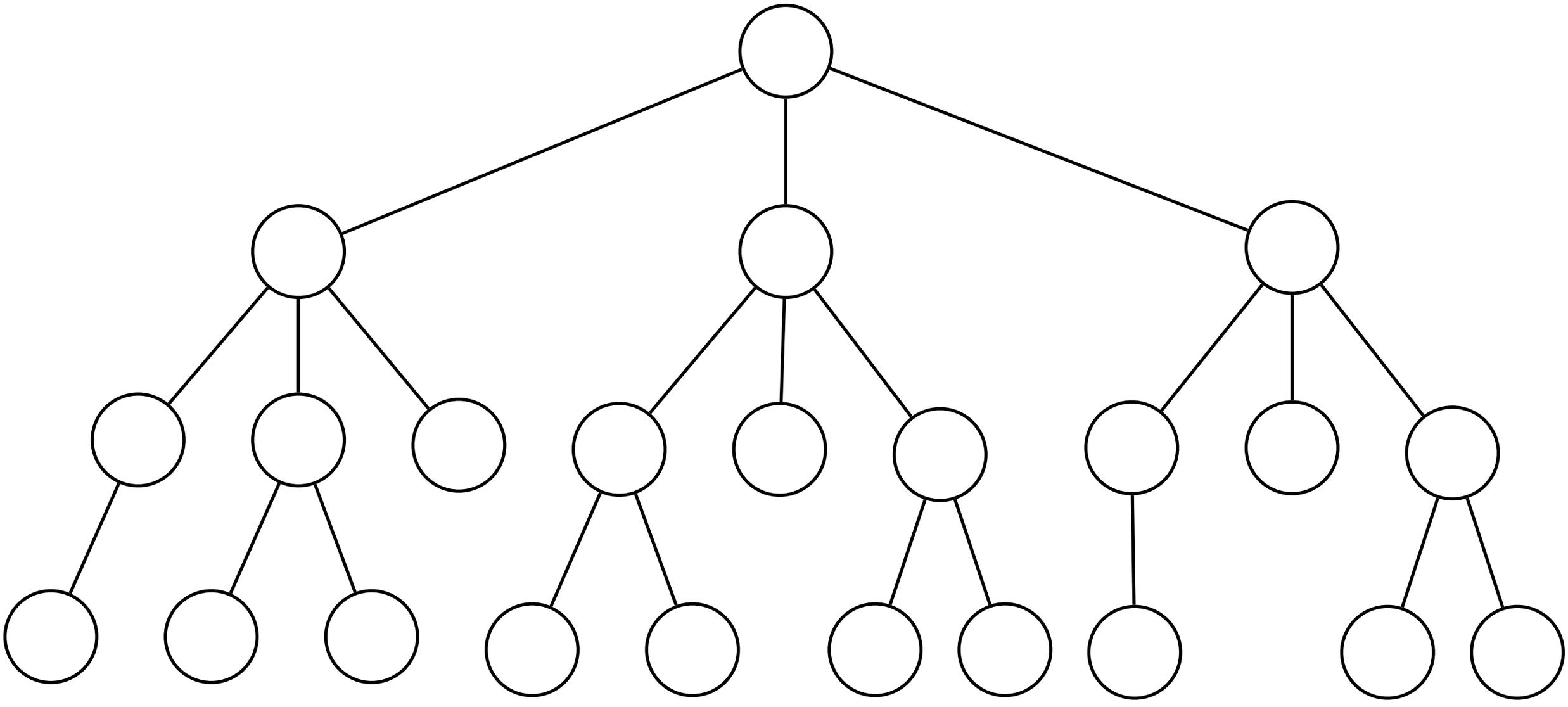


schwarz ist am Zug

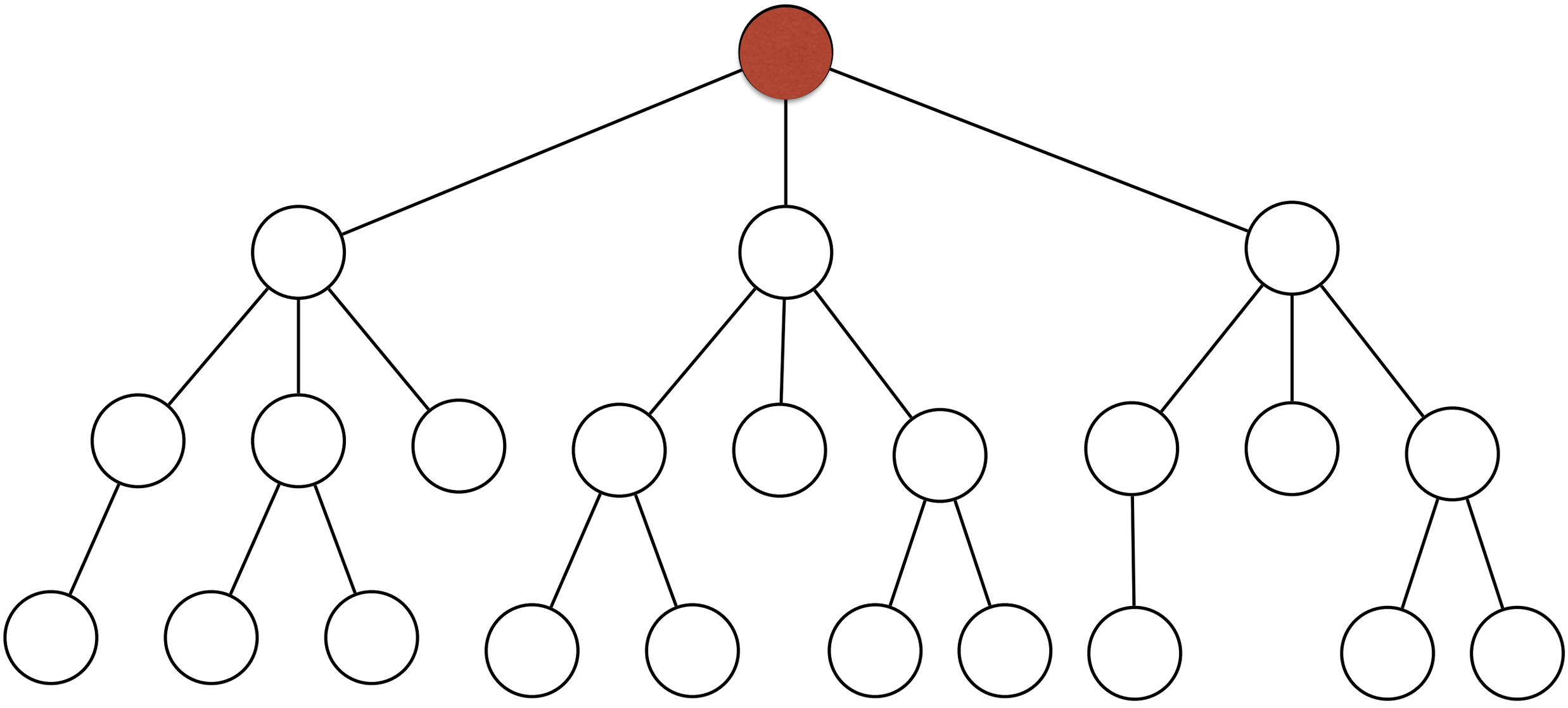
Heuristische Funktion

- Sieg: Hoher Wert
- Niederlage: Niedriger Wert
- Ich bin am Zug: Heuristik maximieren
- Gegner ist am Zug: Heuristik minimieren

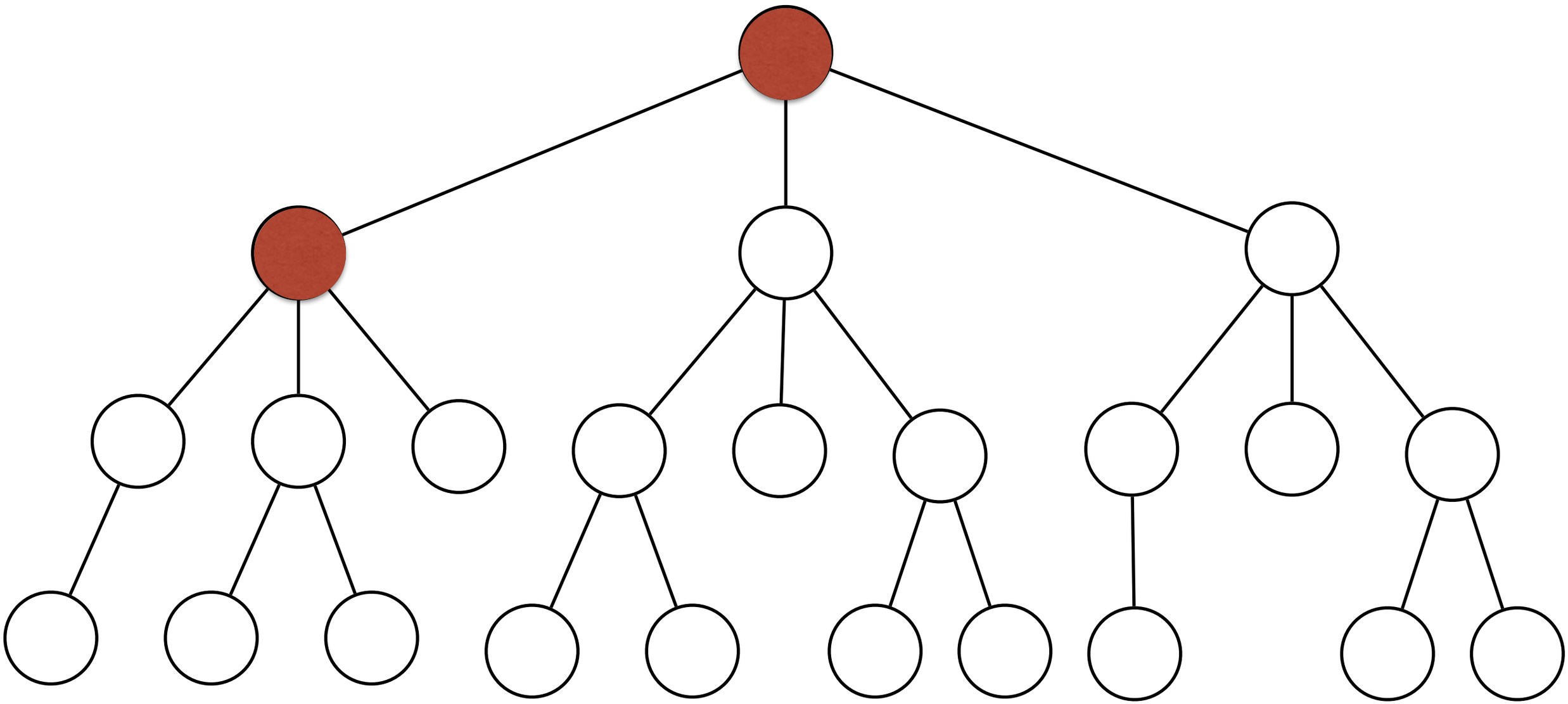
Tiefensuche



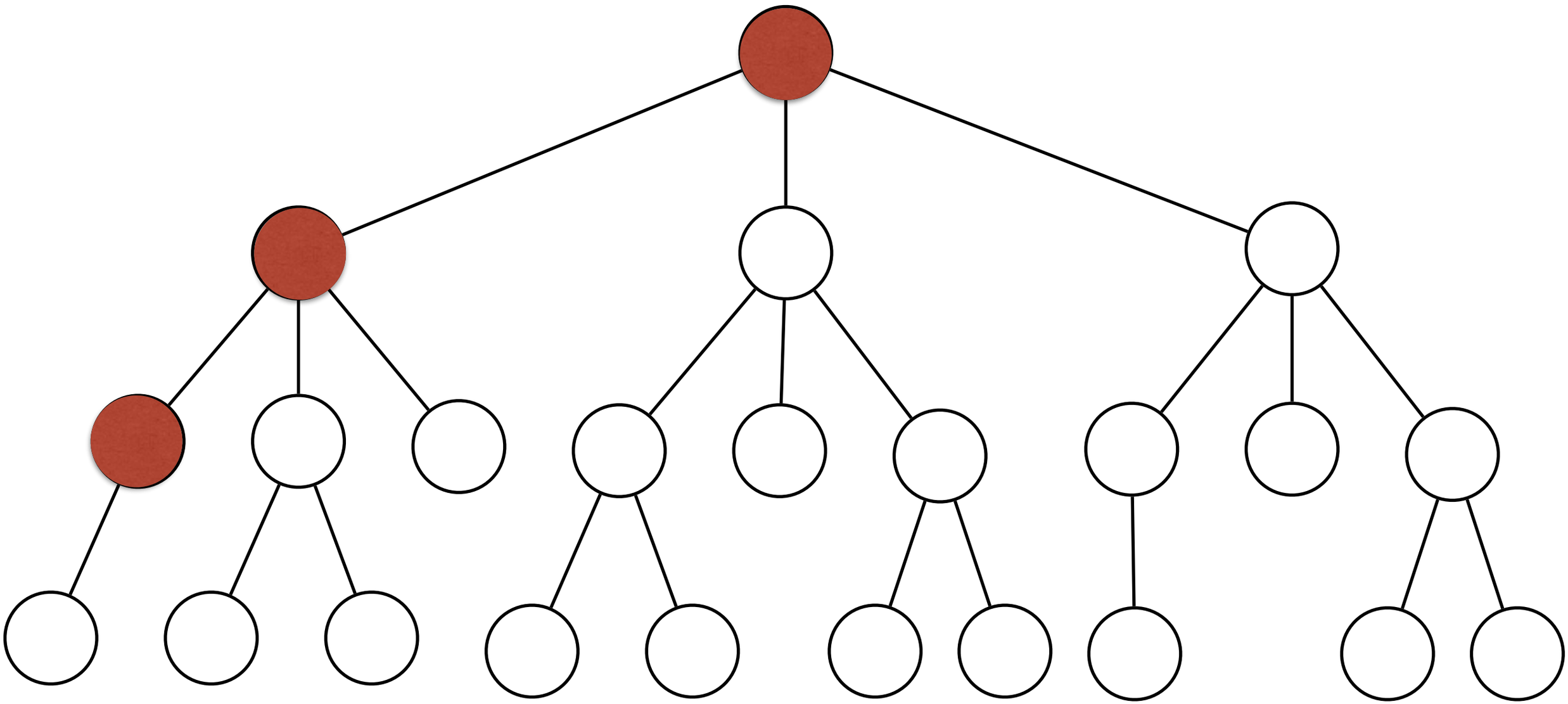
Tiefensuche



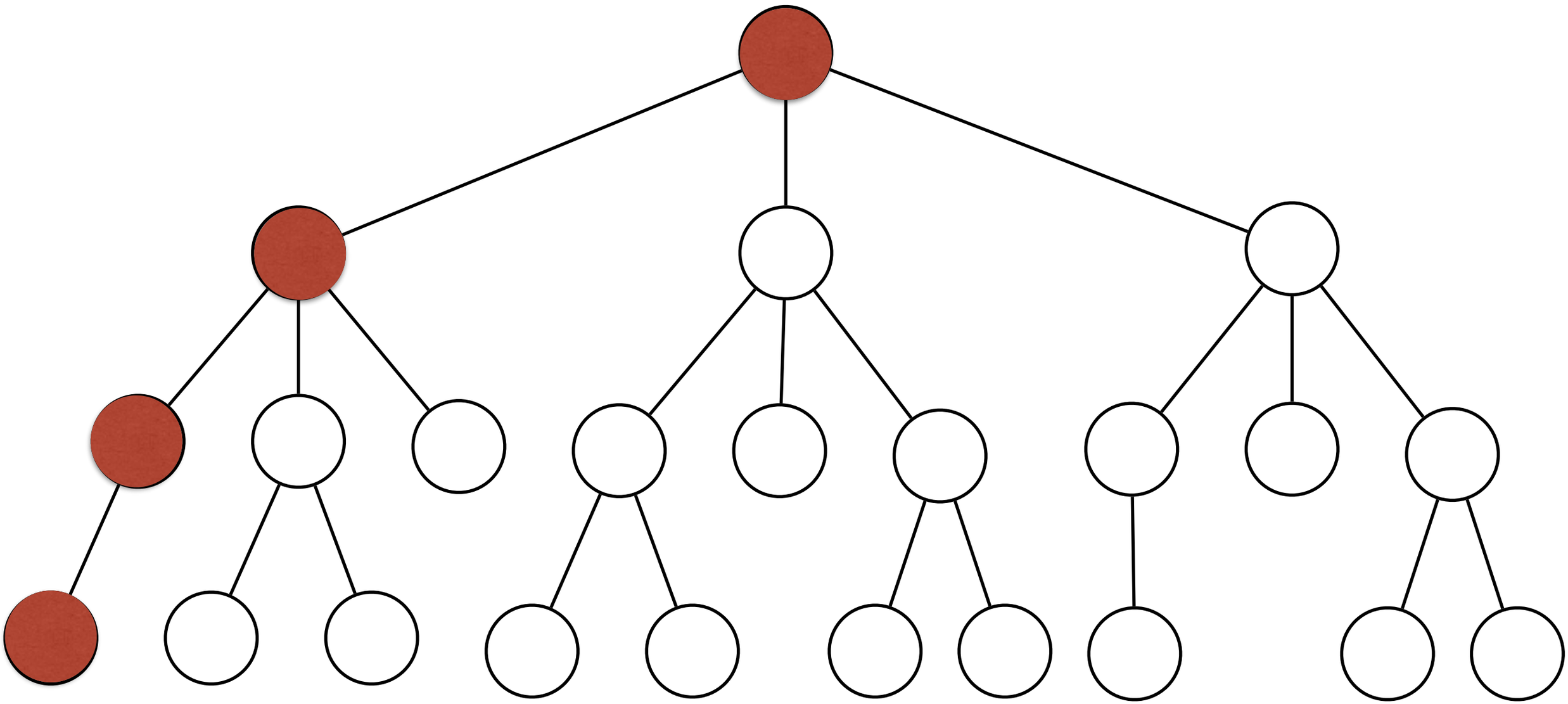
Tiefensuche



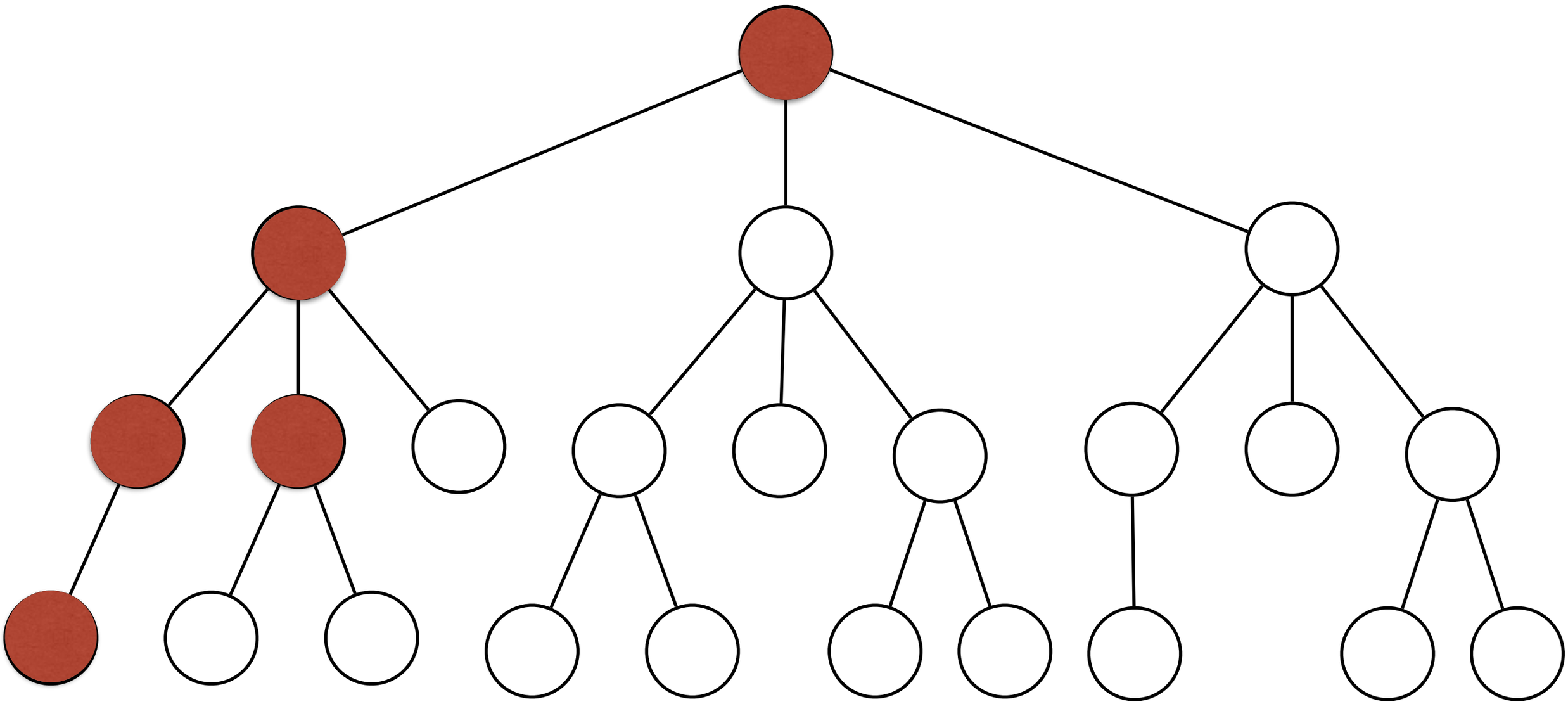
Tiefensuche



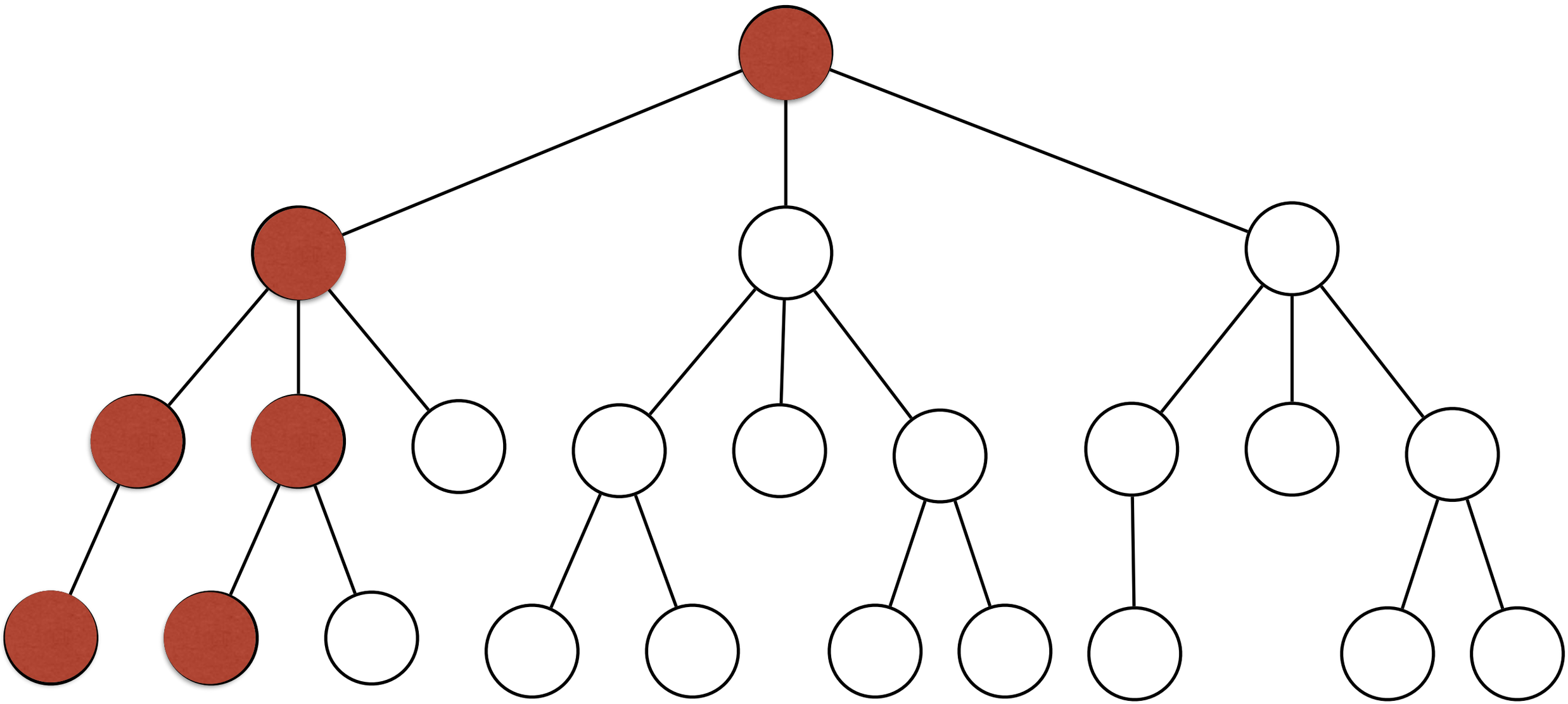
Tiefensuche



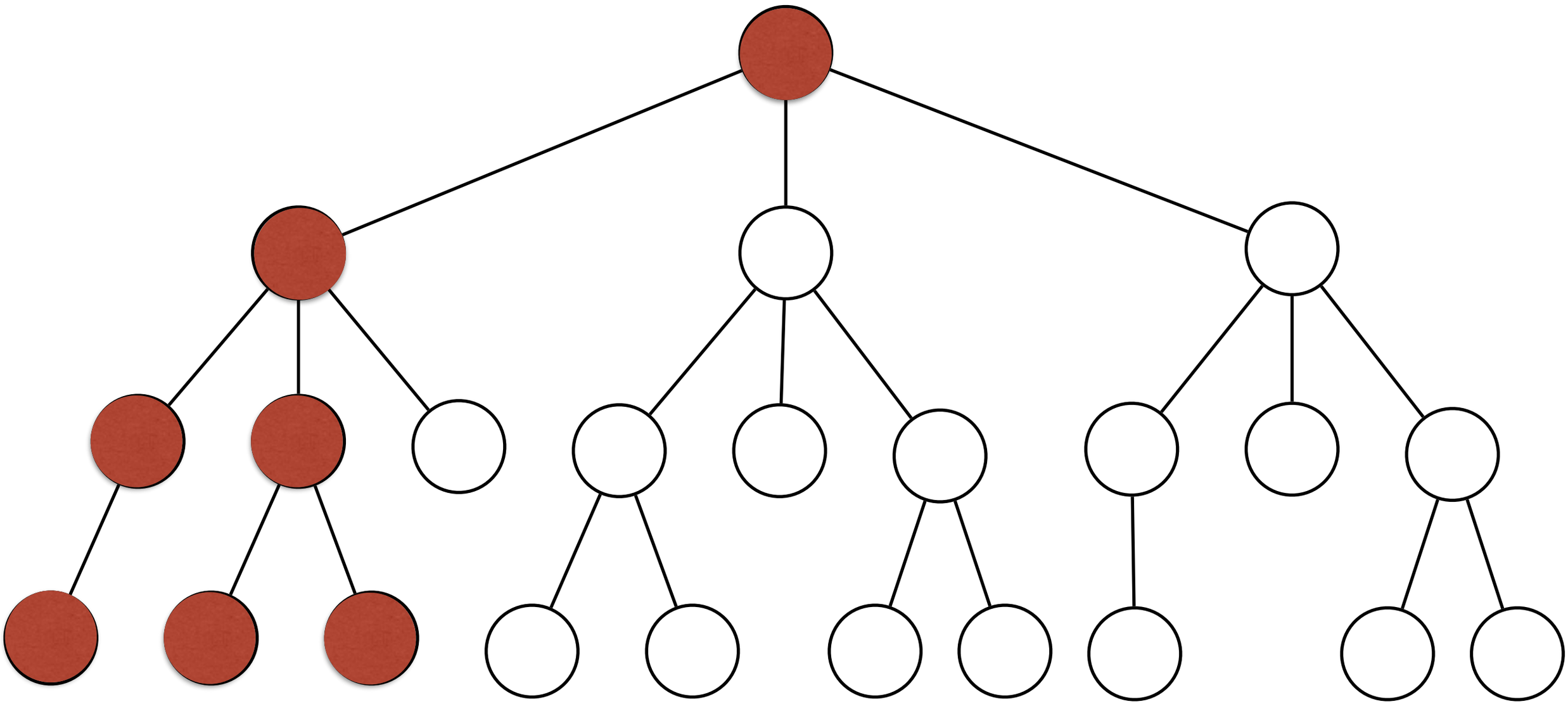
Tiefensuche



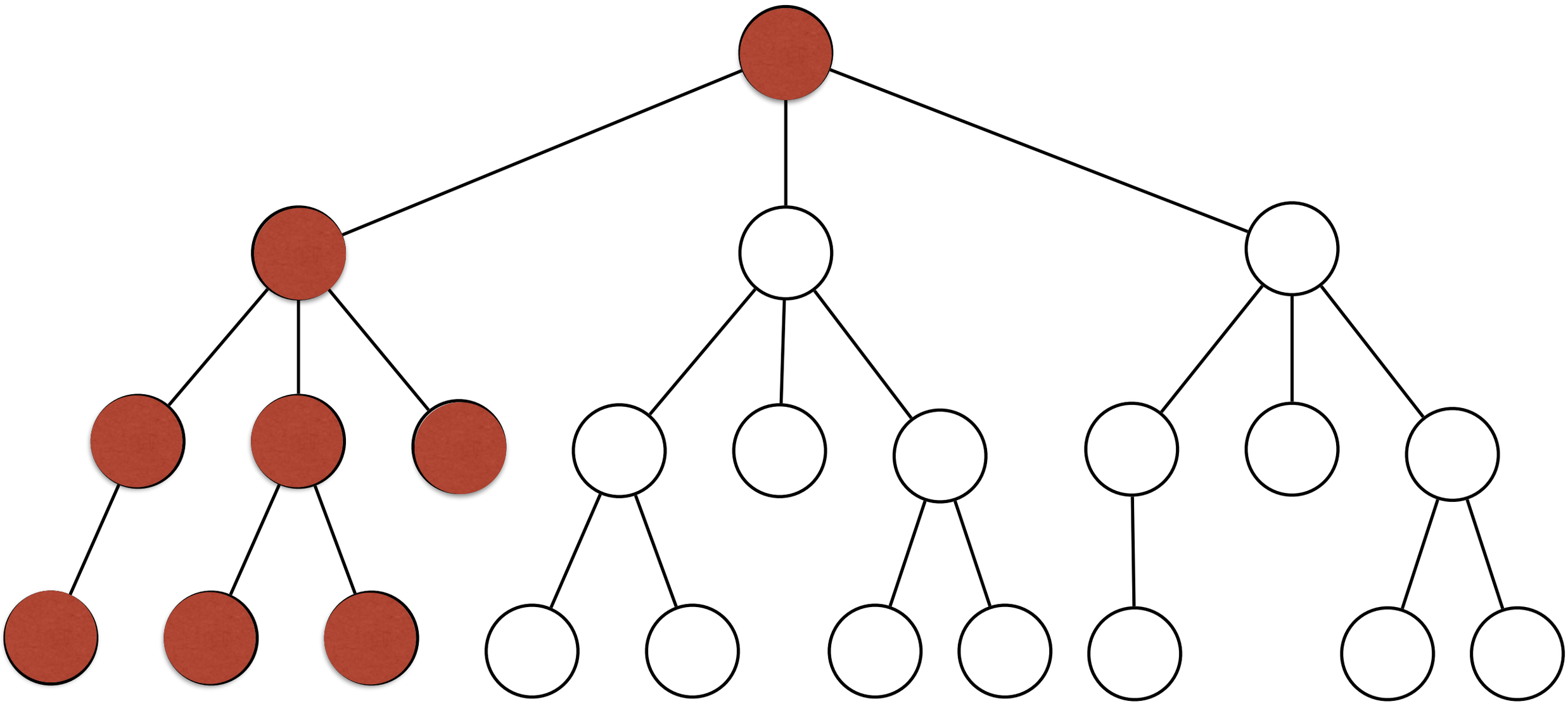
Tiefensuche



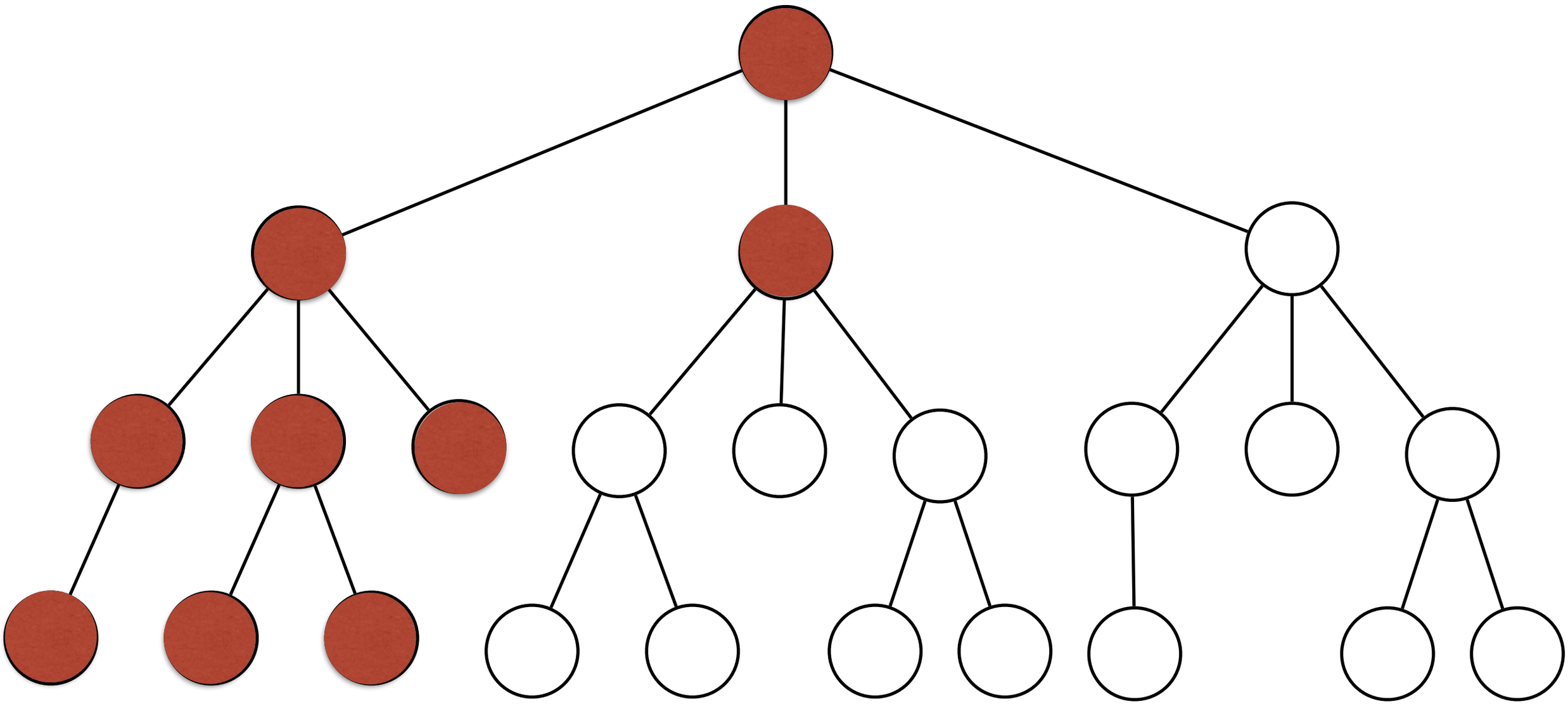
Tiefensuche



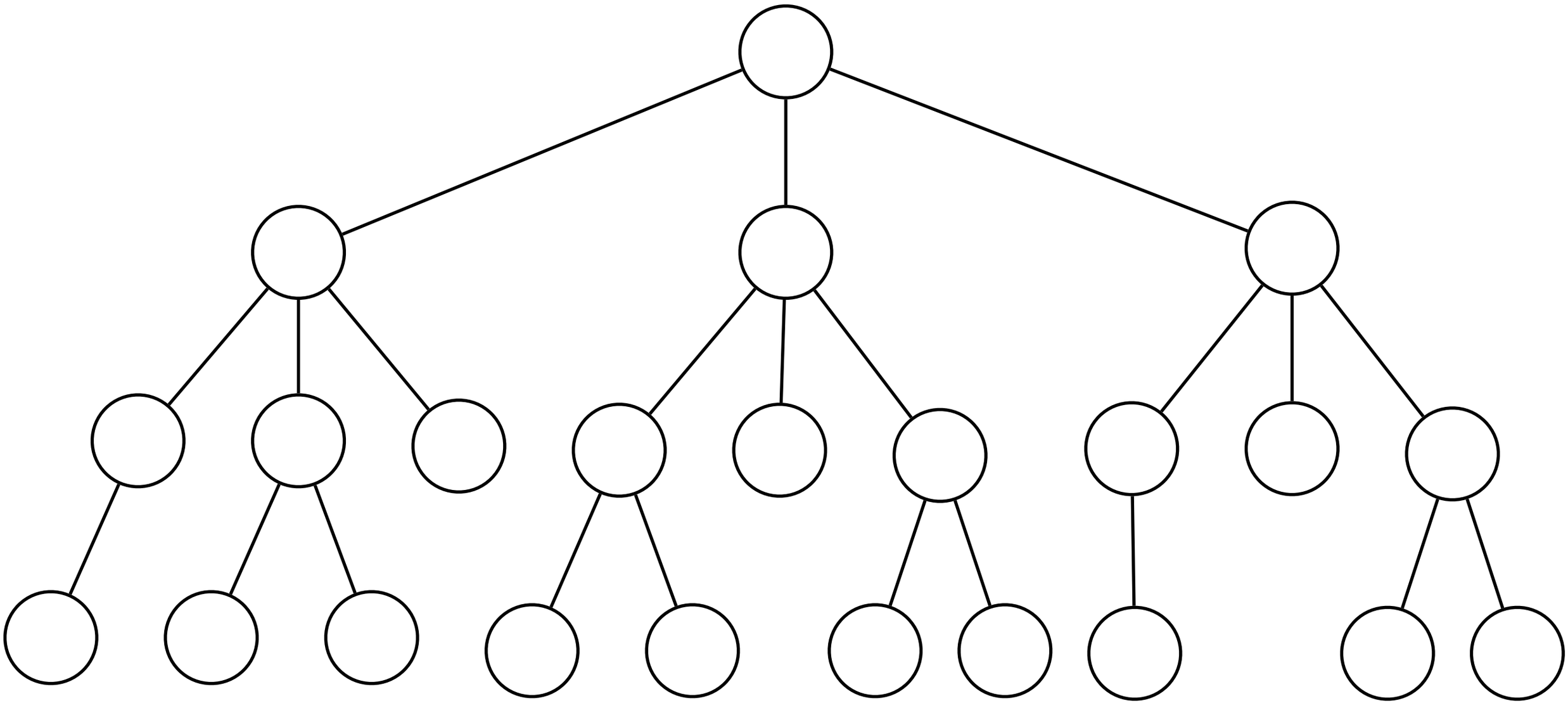
Tiefensuche



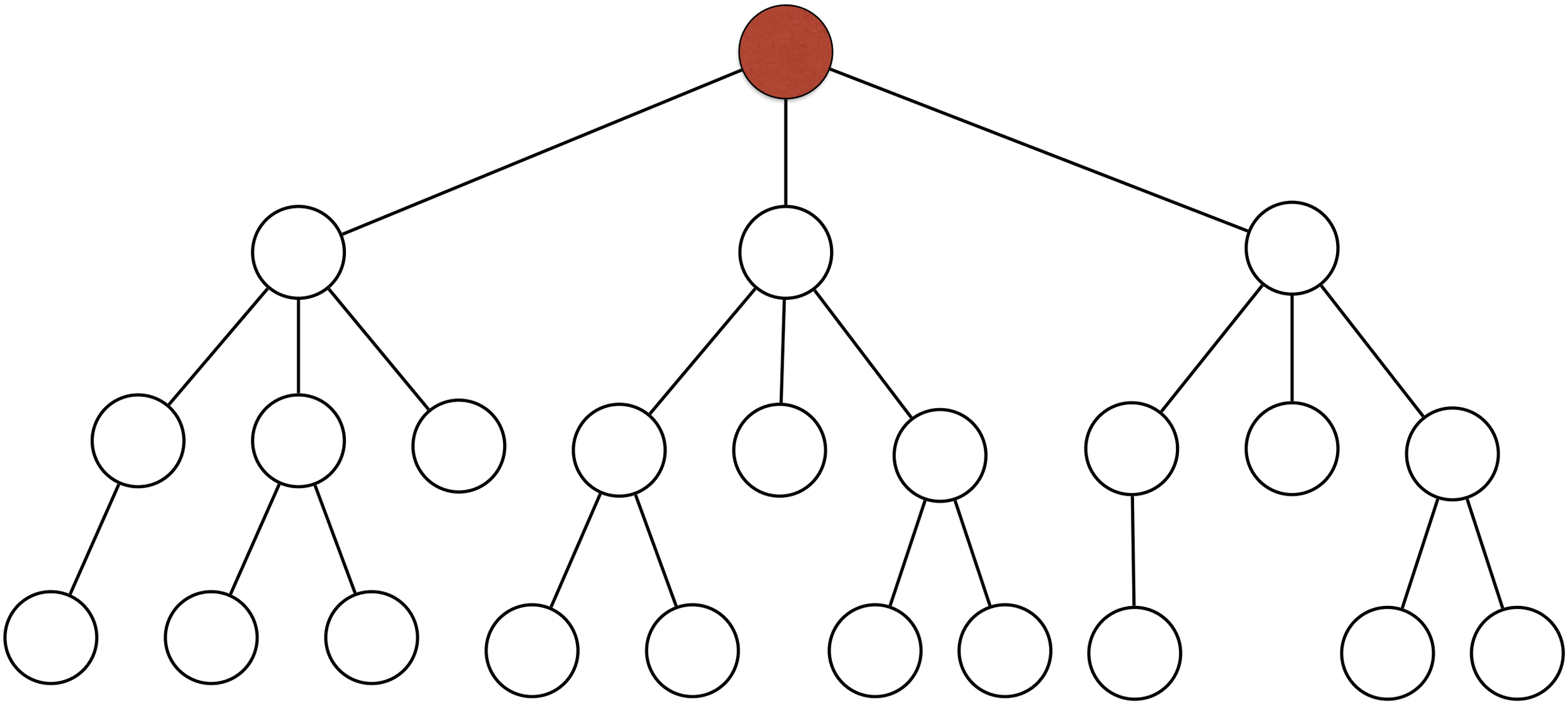
Tiefensuche



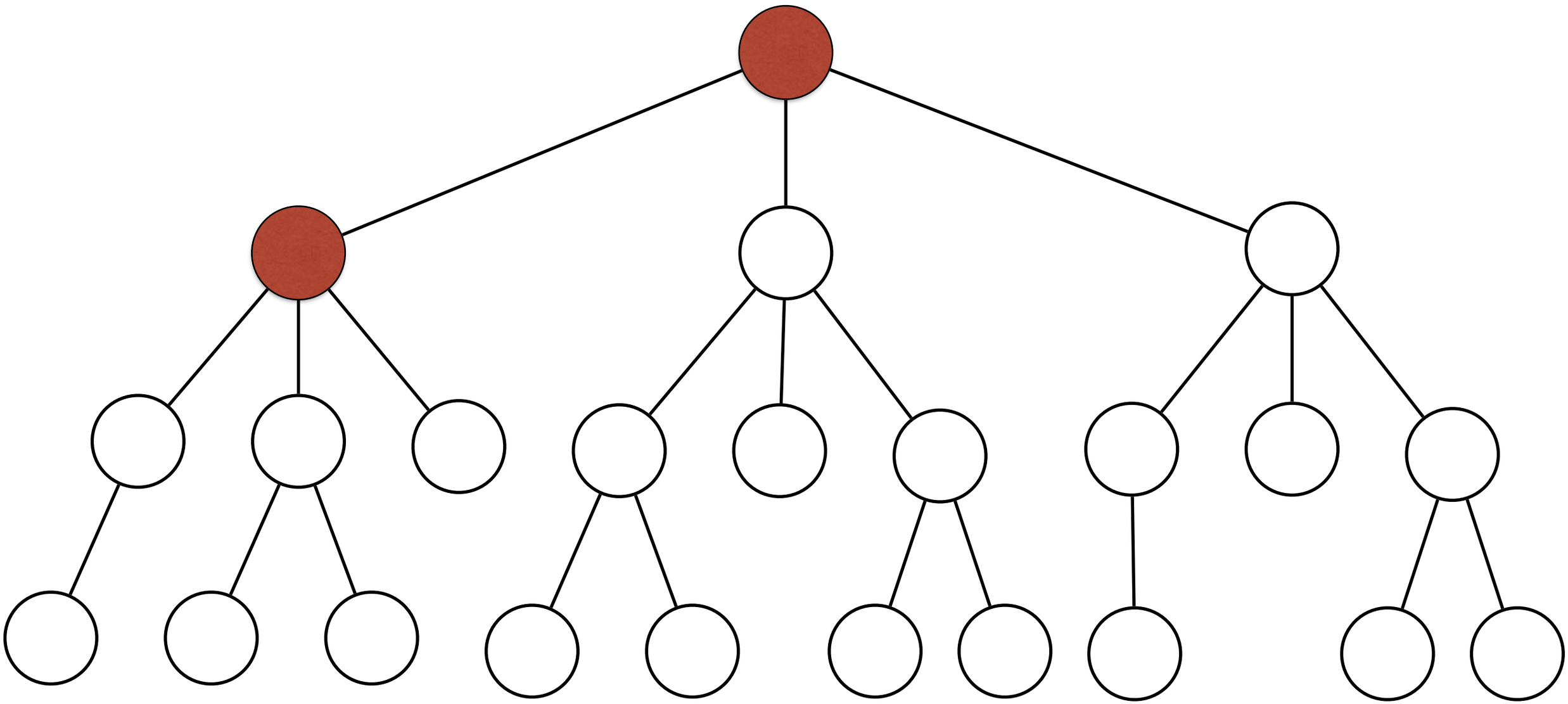
Breitensuche



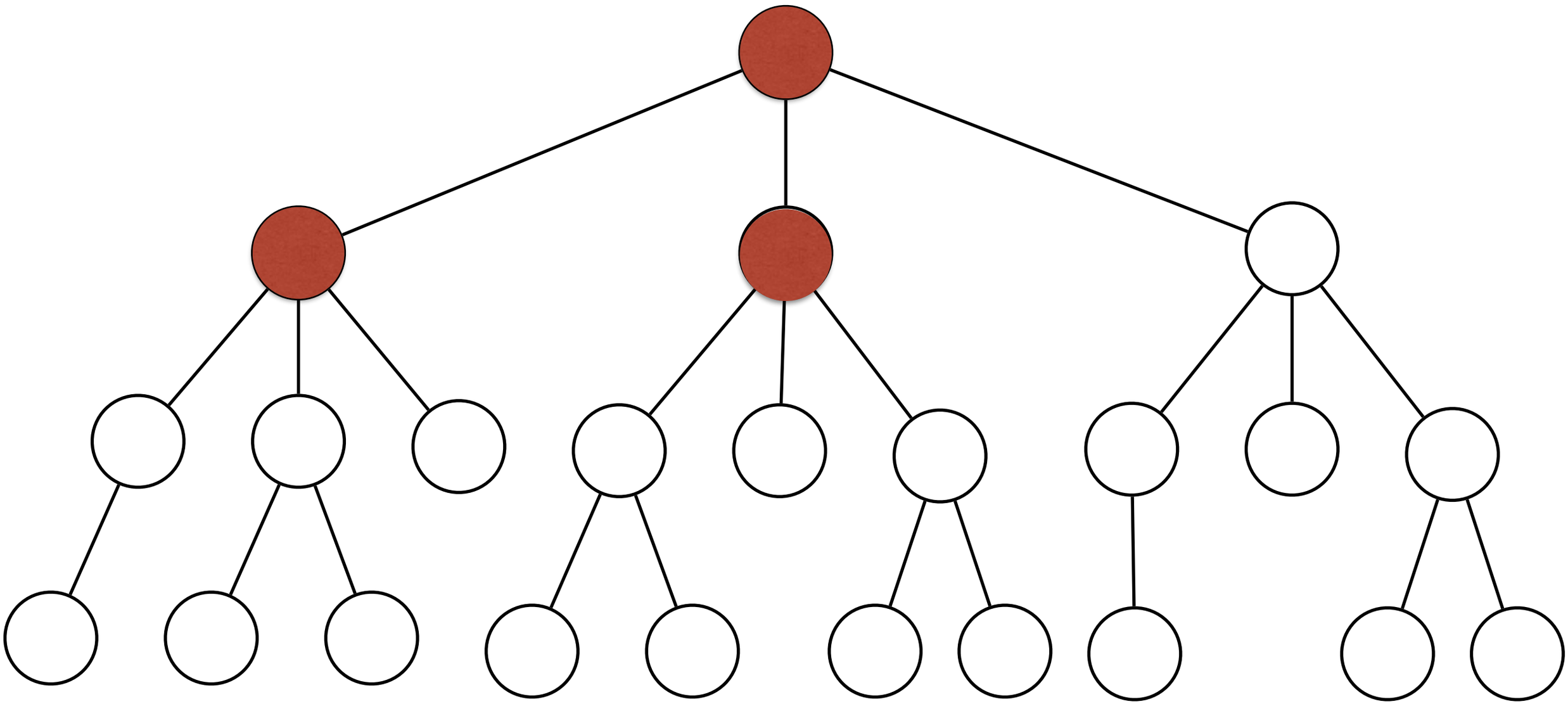
Breitensuche



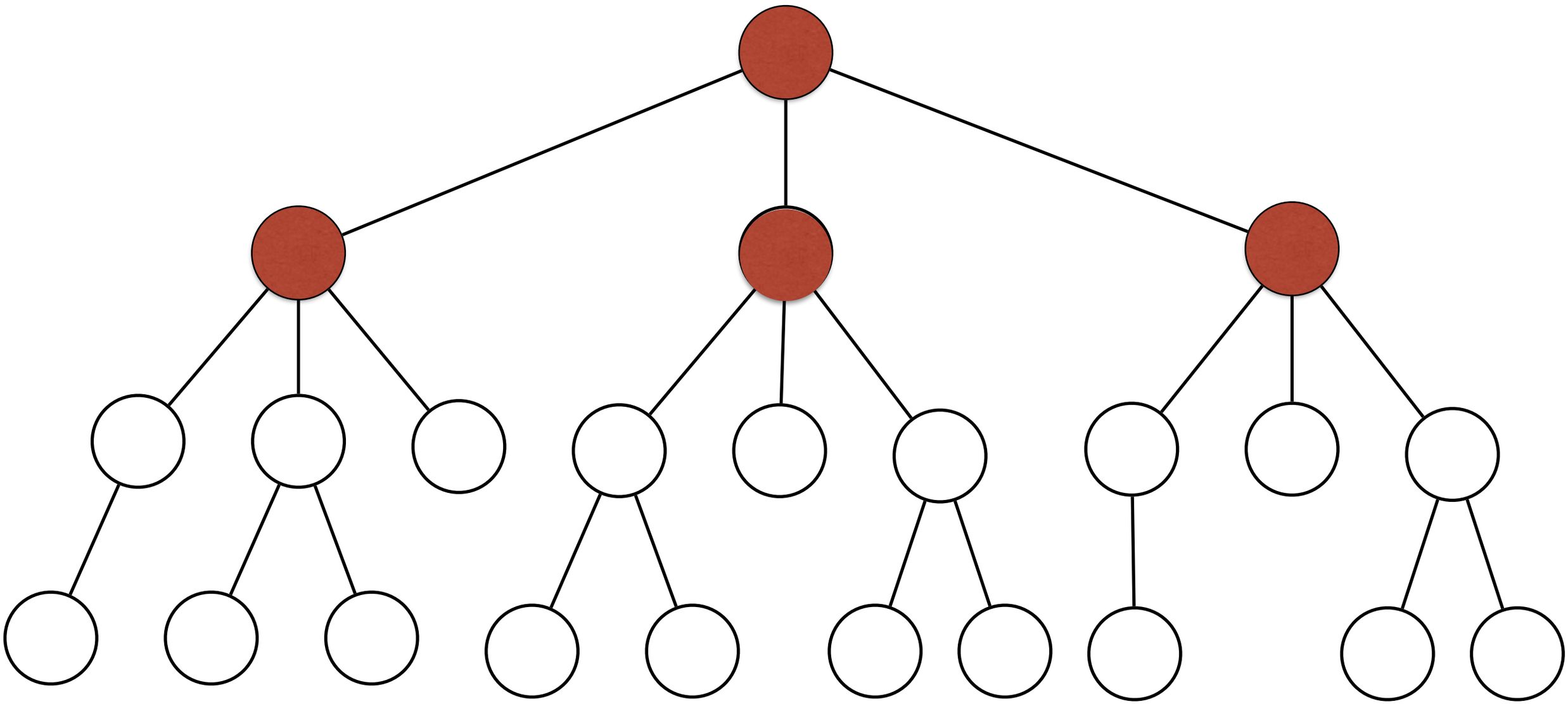
Breitensuche



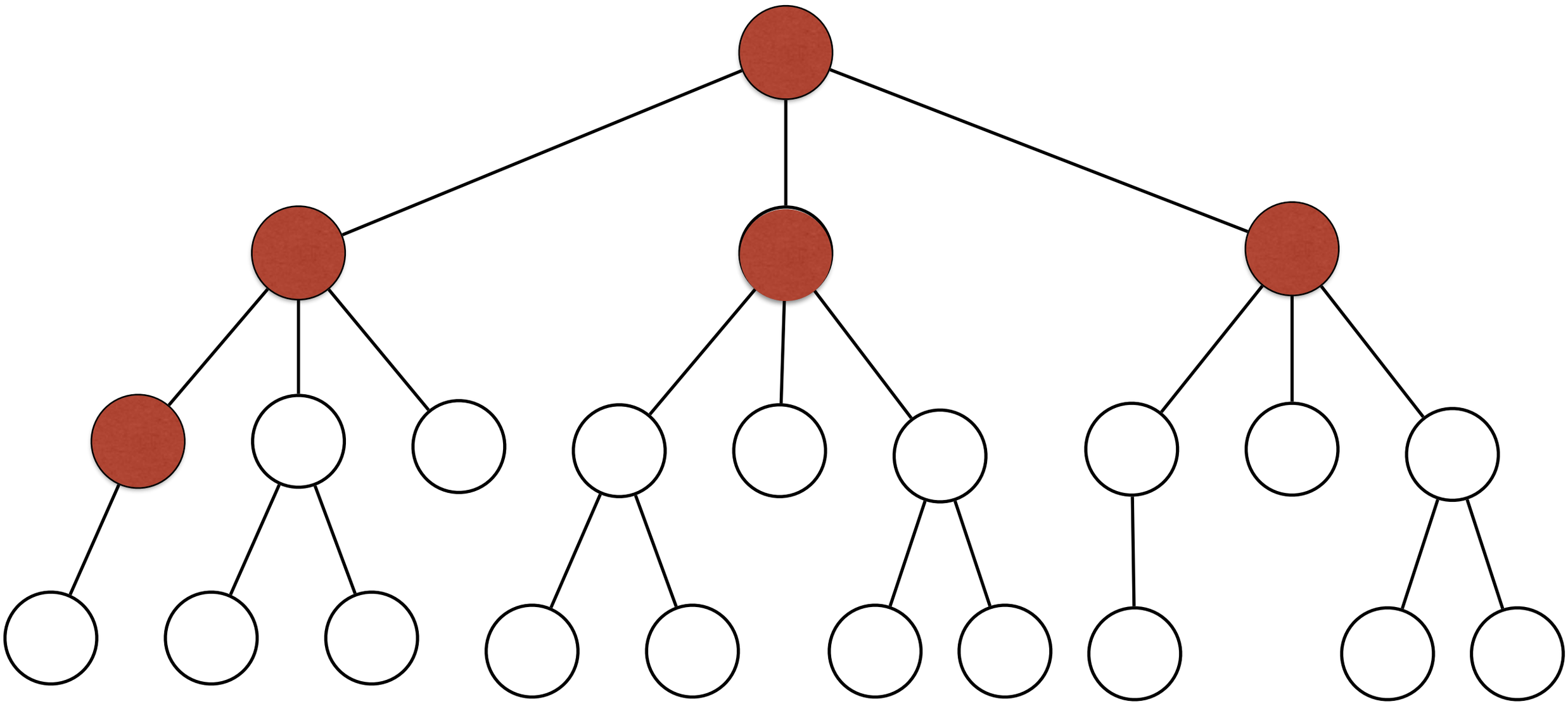
Breitensuche



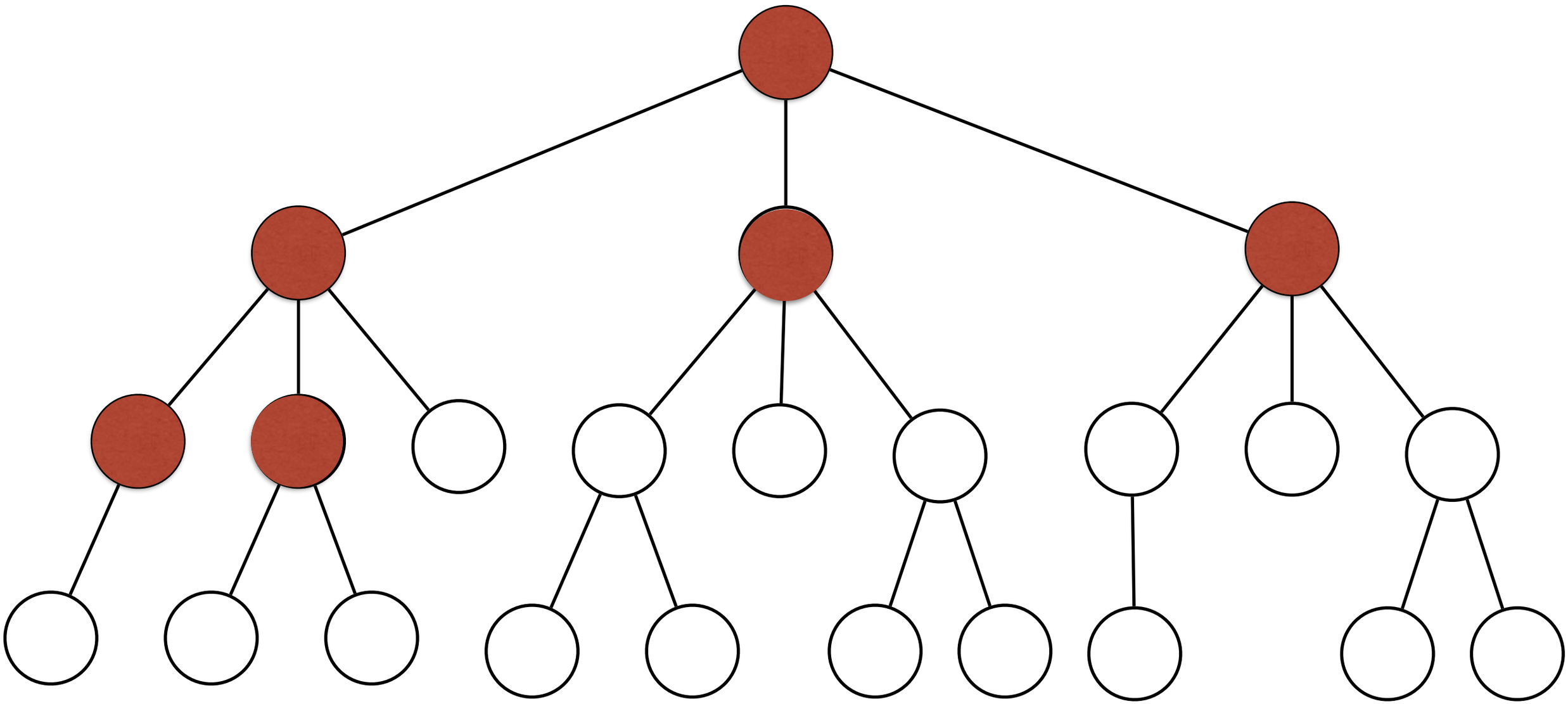
Breitensuche



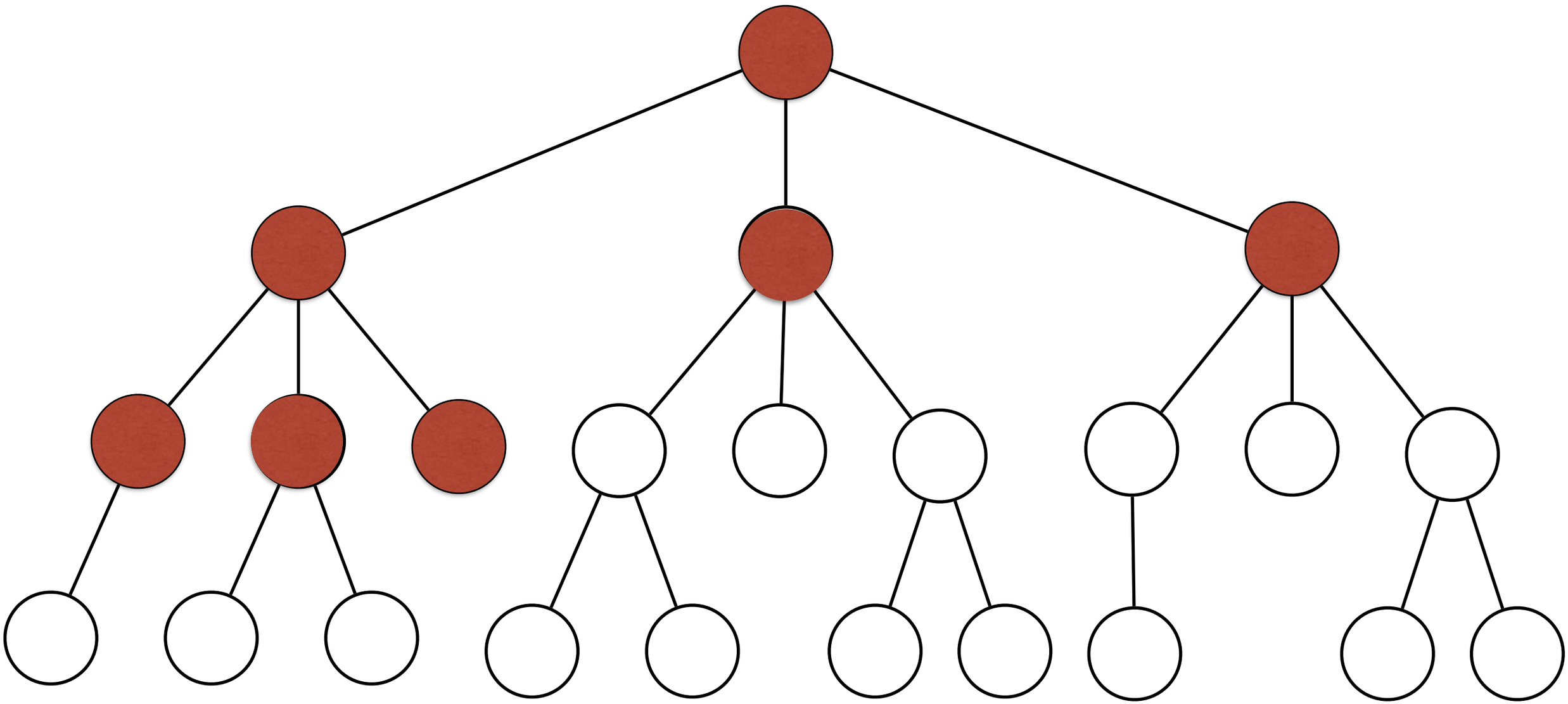
Breitensuche



Breitensuche



Breitensuche



Was, wenn ich nicht das ganze Spiel vorberechnen kann (1)

- Suchbreite beschränken:
  - Wenn ich ein Kind mit maximalem/minimalem Wert gesehen habe, muss ich die anderen Kinder nicht mehr angucken
- Suchtiefe beschränken:
  - Beispiel: Ich gucke 5 Zustände in die Zukunft

Was, wenn ich nicht das ganze Spiel vorberechnen kann (2)

- Heuristik mit anderen Informationen:
  - eigene Lebenspunkte
  - Summe der Gegner-Lebenspunkte
  - Entfernung zum Gegner
  - …
- Damit ist auch Breitensuche möglich

Was, wenn ich nicht das ganze Spiel vorberechnen kann (3)

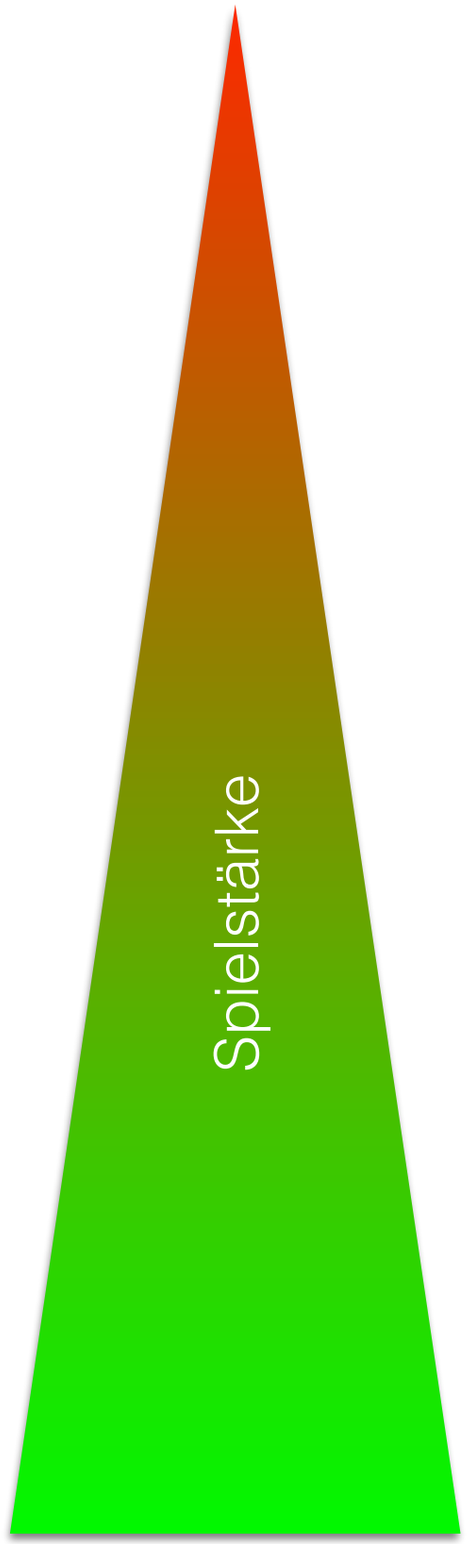
- Machine Learning:
  - Sammeln Sie möglichst viele Zustände einschließlich Ergebnisse
  - Lassen Sie ein neuronales Netz (oder einen anderen Classifier) lernen, ob nach einem bestimmten Zustand ein Sieg kommt

Optionen für eine KI

- Regelbasiert:
  - gute Kontrolle über KI-Verhalten
  - schlechte Anpassung an unerwartete Situationen
- heuristische Baumsuche:
  - KI-Verhalten hängt stark von der Heuristik ab
  - sehr erfolgreich (Schach, Mühle, …)
- Machine Learning:
  - keine Kontrolle über KI-Verhalten (quasi keine Möglichkeit zu debuggen)
  - extrem erfolgreich (Atari-Spiele, Go, …)

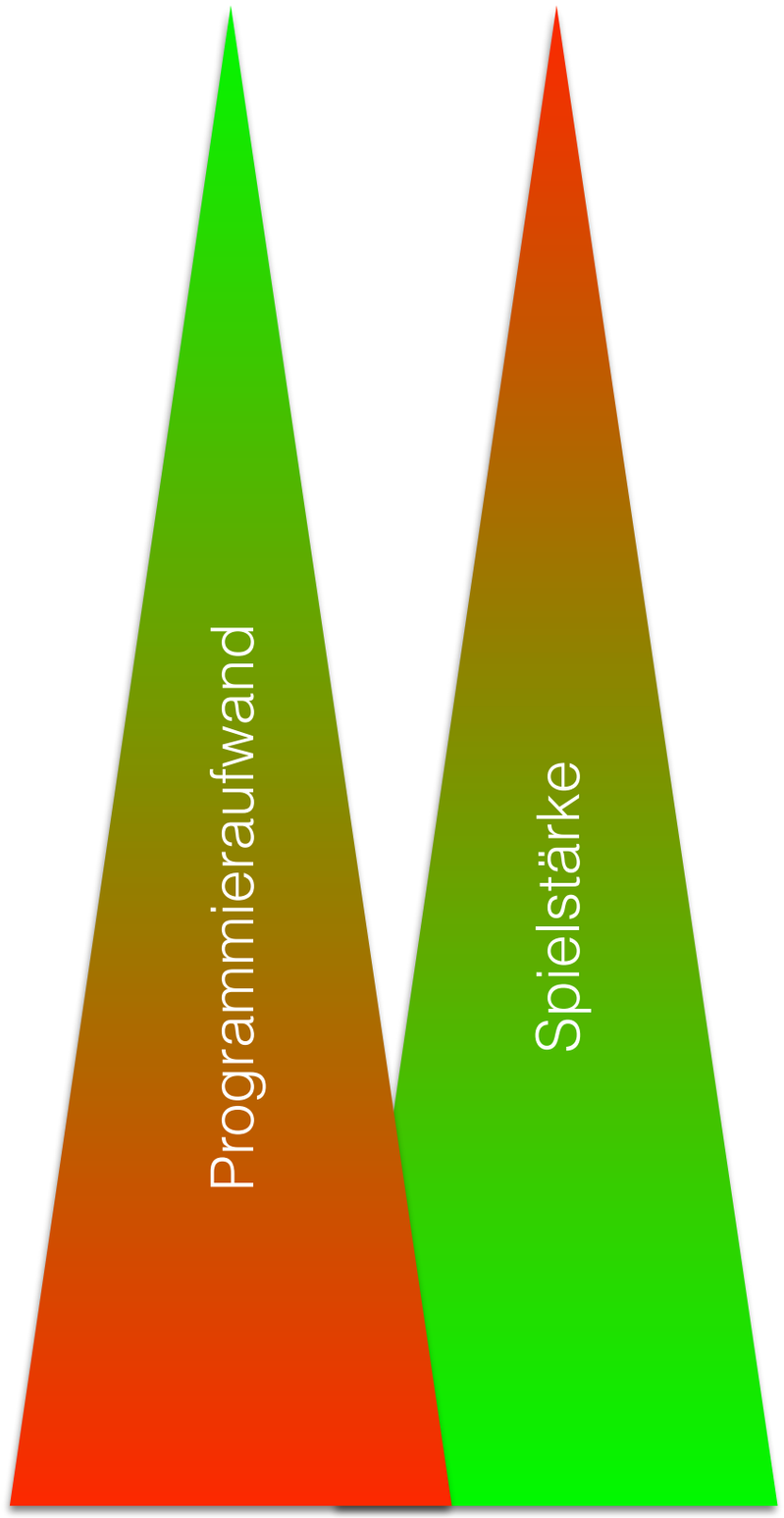
Optionen für eine KI

- Regelbasiert:
  - gute Kontrolle über KI-Verhalten
  - schlechte Anpassung an unerwartete Situationen
- heuristische Baumsuche:
  - KI-Verhalten hängt stark von der Heuristik ab
  - sehr erfolgreich (Schach, Mühle, …)
- Machine Learning:
  - keine Kontrolle über KI-Verhalten (quasi keine Möglichkeit zu debuggen)
  - extrem erfolgreich (Atari-Spiele, Go, …)



Optionen für eine KI

- Regelbasiert:
  - gute Kontrolle über KI-Verhalten
  - schlechte Anpassung an unerwartete Situationen
- heuristische Baumsuche:
  - KI-Verhalten hängt stark von der Heuristik ab
  - sehr erfolgreich (Schach, Mühle, …)
- Machine Learning:
  - keine Kontrolle über KI-Verhalten (quasi keine Möglichkeit zu debuggen)
  - extrem erfolgreich (Atari-Spiele, Go, …)



Wie implementiere ich eine Baumsuche?

- Queue q
- lege den aktuellen Zustand in q
- solange noch Zustände in q sind:
    - nehme den nächsten Zustände aus q
    - erzeuge alle Kind-Zustände
    - lege die Kind-Zustände in q
- wenn keine Zustände mehr in q sind
    - der beste Zustand ist der, wo ich hin will

Beispiel

Heuristik: Abstand zum Wildschwein

- Queue:
  - Move(NORTH_EAST), 3
  - Move(NORTH_WEST), 3
  - Move(WEST), 4
  - Move(EAST), 4
  - Move(SOUTH_WEST), 5
  - Move(SOUTH_EAST), 5

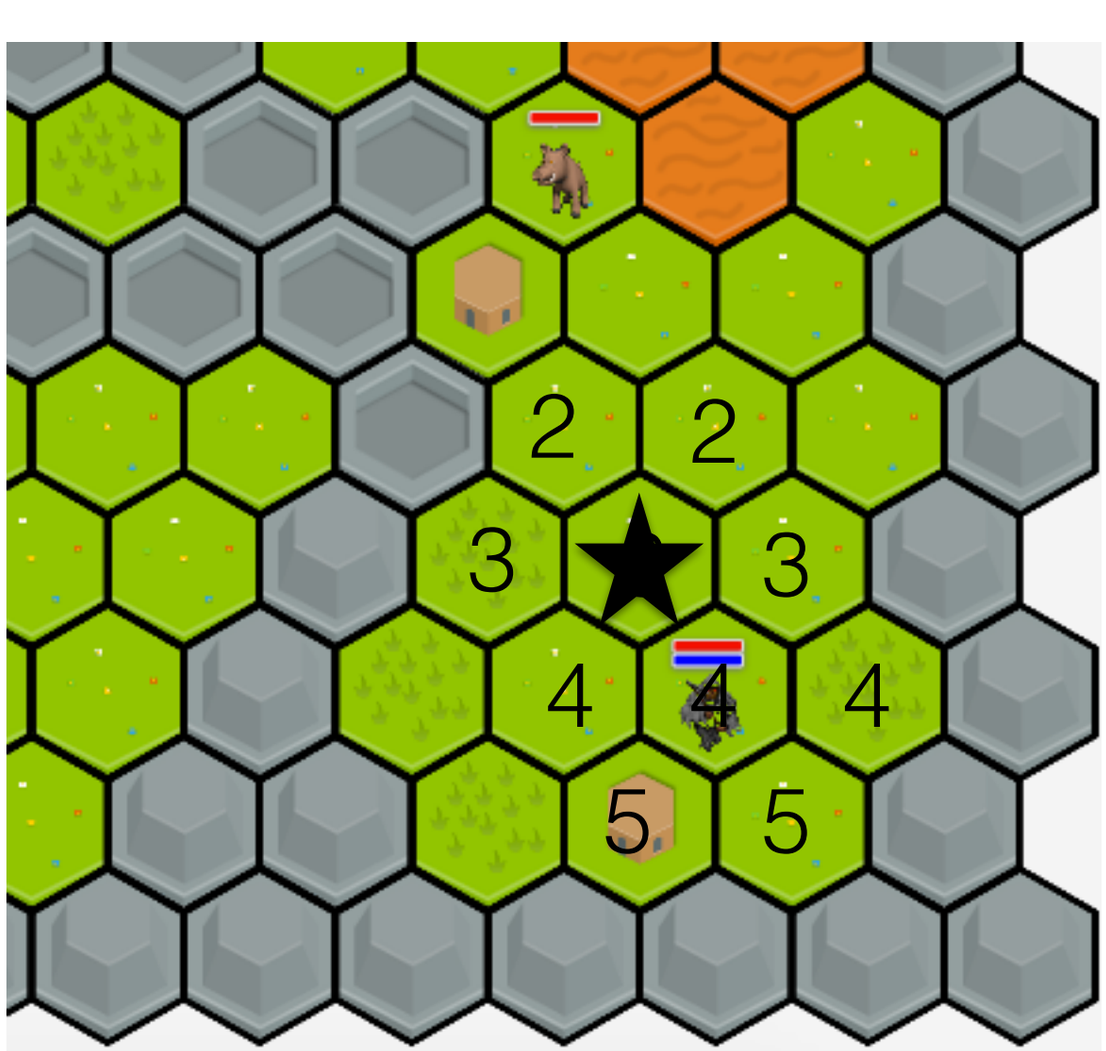
Beispiel

Heuristik: Abstand zum Wildschwein

- Queue:
  - Move(NORTH_EAST), 3
  - Move(NORTH_WEST), 3
  - Move(WEST), 4
  - Move(EAST), 4
  - Move(SOUTH_WEST), 5
  - Move(SOUTH_EAST), 5

Beispiel

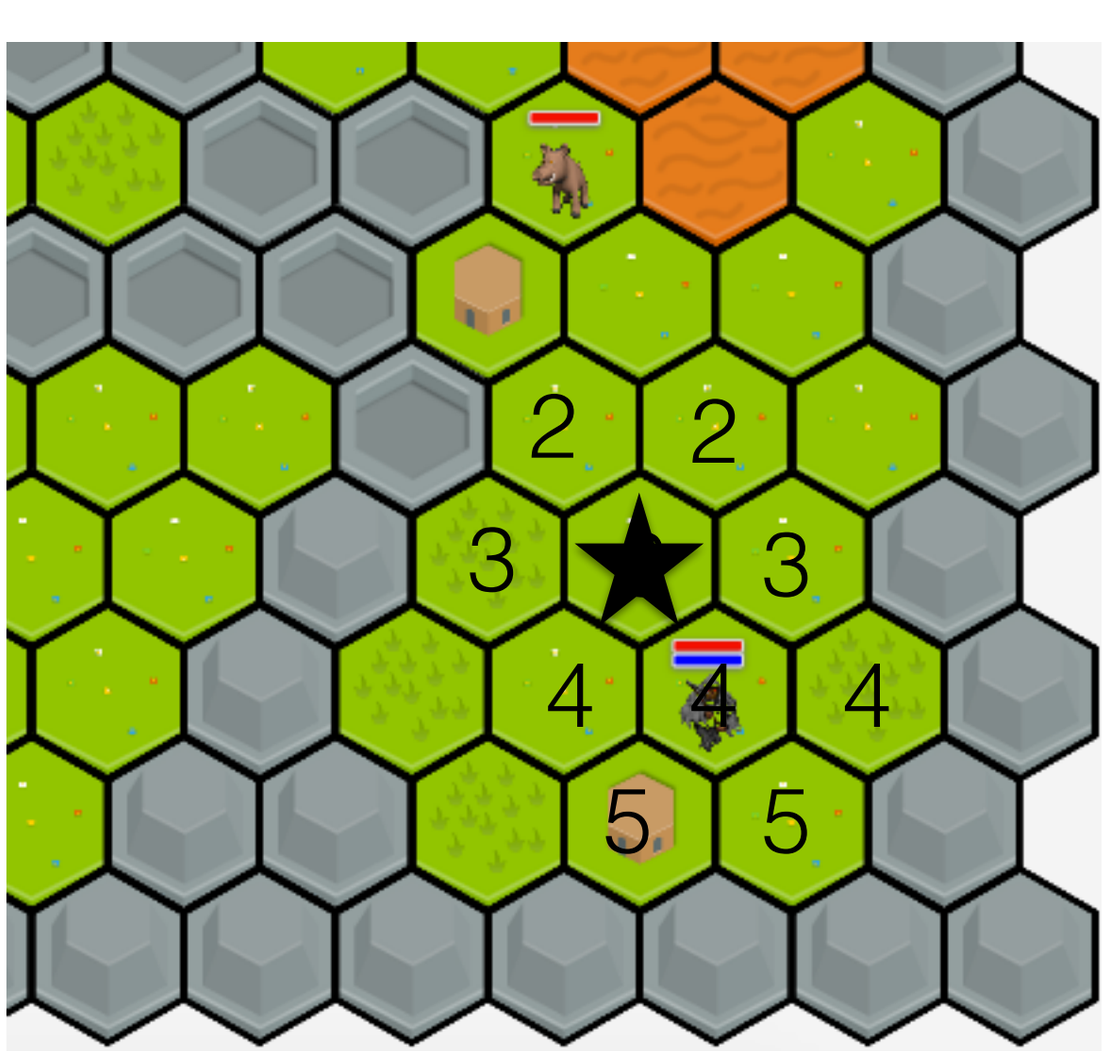
Heuristik: Abstand zum Wildschwein



- Queue:
  - Move(NORTH_EAST), 2
  - Move(NORTH_WEST), 2
  - Move(EAST), 3
  - Move(WEST), 3
  - Move(SOUTH_WEST), 4
  - Move(SOUTH_EAST), 4
  - (alt) Move(NORTH_EAST), 3
  - (alt) Move(NORTH_WEST), 3
  - (alt) Move(WEST), 4
  - (alt) Move(EAST), 4
  - (alt) Move(SOUTH_WEST), 5
  - (alt) Move(SOUTH_EAST), 5

Beispiel

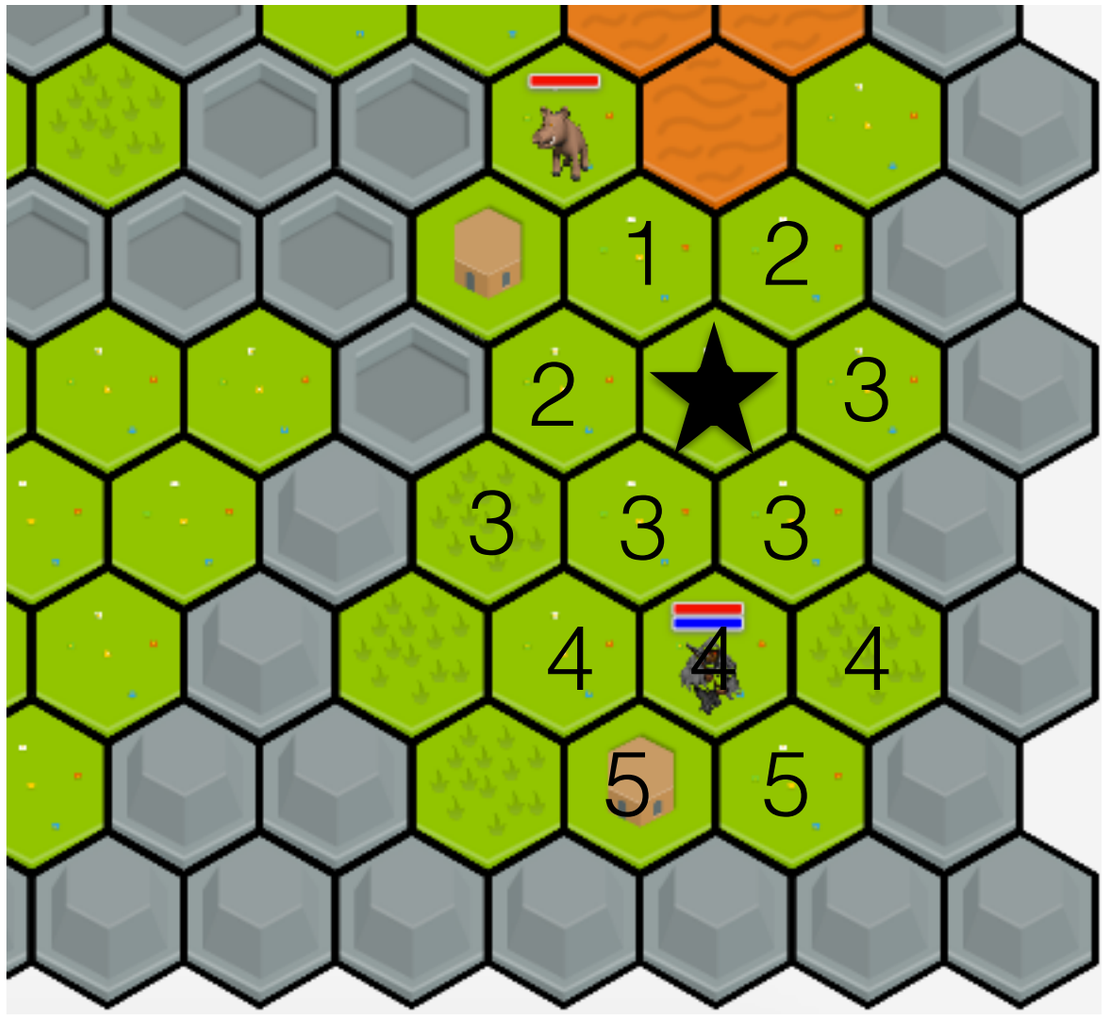
Heuristik: Abstand zum Wildschwein



- Queue:
  - Move(NORTH_EAST), 2
  - Move(NORTH_WEST), 2
  - Move(EAST), 3
  - Move(WEST), 3
  - Move(SOUTH_WEST), 4
  - Move(SOUTH_EAST), 4
  - (alt) Move(NORTH_EAST), 3
  - (alt) Move(NORTH_WEST), 3
  - (alt) Move(WEST), 4
  - (alt) Move(EAST), 4
  - (alt) Move(SOUTH_WEST), 5
  - (alt) Move(SOUTH_EAST), 5

Beispiel

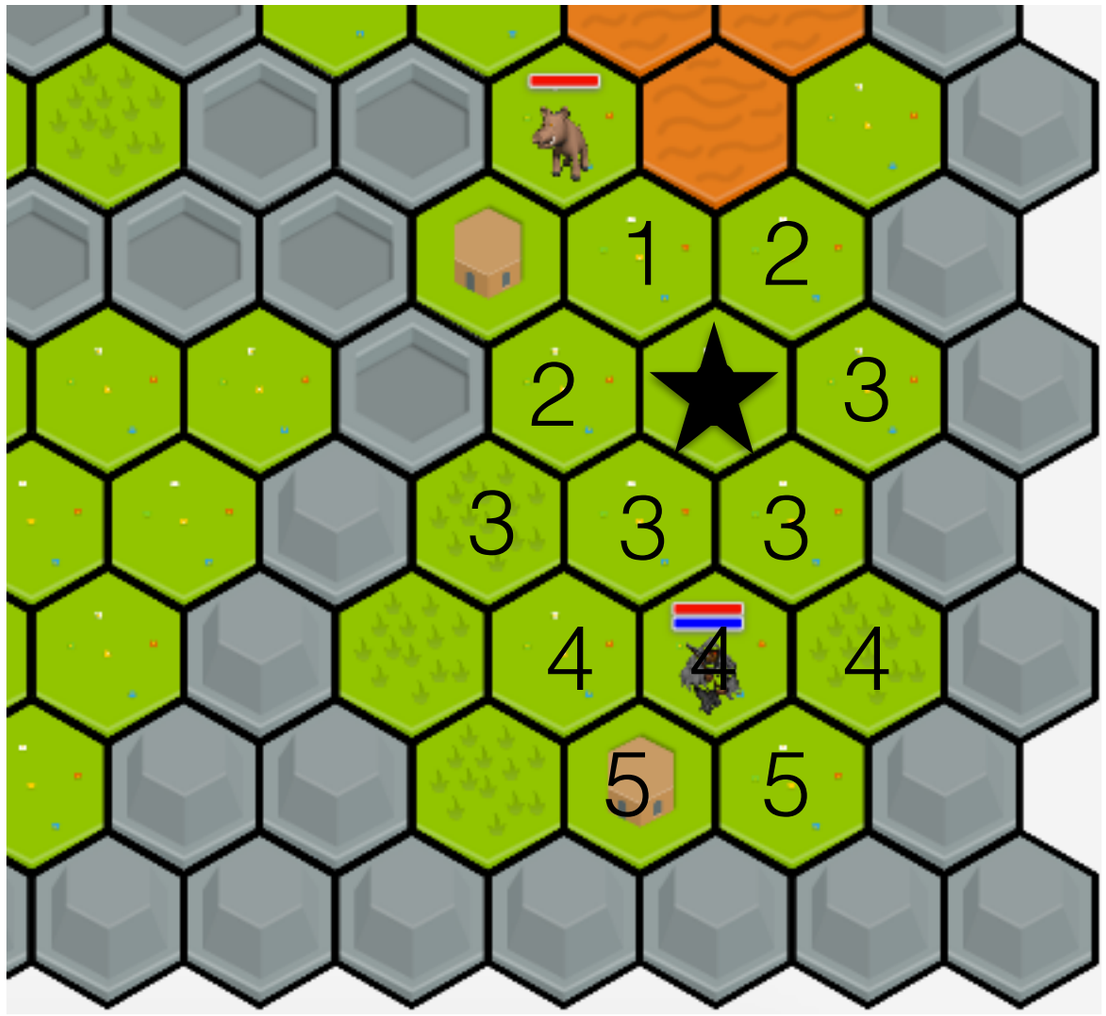
Heuristik: Abstand zum Wildschwein



- Queue:
  - Move(NORTH_EAST), 2
  - Move(NORTH_WEST), 1
  - Move(EAST), 3
  - Move(WEST), 2
  - (alt) Move(NORTH_EAST), 2
  - (alt)Move(NORTH_WEST), 2
  - (alt) Move(EAST), 3
  - (alt) Move(WEST), 3
  - (alt) Move(SOUTH_WEST), 4
  - (alt) Move(SOUTH_EAST), 4
  - (alt) Move(NORTH_EAST), 3
  - (alt) Move(NORTH_WEST), 3
  - (alt) Move(WEST), 4
  - (alt) Move(EAST), 4
  - (alt) Move(SOUTH_WEST), 5

Beispiel

Heuristik: Abstand zum Wildschwein



- Queue:
  - Move(NORTH_EAST), 2
  - Move(NORTH_WEST), 1
  - Move(EAST), 3
  - Move(WEST), 2
  - (alt) Move(NORTH_EAST), 2
  - (alt)Move(NORTH_WEST), 2
  - (alt) Move(EAST), 3
  - (alt) Move(WEST), 3
  - (alt) Move(SOUTH_WEST), 4
  - (alt) Move(SOUTH_EAST), 4
  - (alt) Move(NORTH_EAST), 3
  - (alt) Move(NORTH_WEST), 3
  - (alt) Move(WEST), 4
  - (alt) Move(EAST), 4
  - (alt) Move(SOUTH_WEST), 5

Anforderungen an das Model

- Ich muss informiert werden, wenn ich dran bin.
- Ich muss prüfen können, ob ein Zug zulässig ist.
- Ich muss mitteilen können, was ich machen möchte.
- Ich muss abfragen können, wie die Spielwelt aussieht.

Anforderungen an das Model

- Ich muss informiert werden, wenn ich dran bin.
- Ich muss prüfen können, ob ein Zug zulässig ist.
- Ich muss mitteilen können, was ich machen möchte.
- Ich muss abfragen können, wie die Spielwelt aussieht.
- Ich muss Spielzüge simulieren können.

Wie simuliere ich?

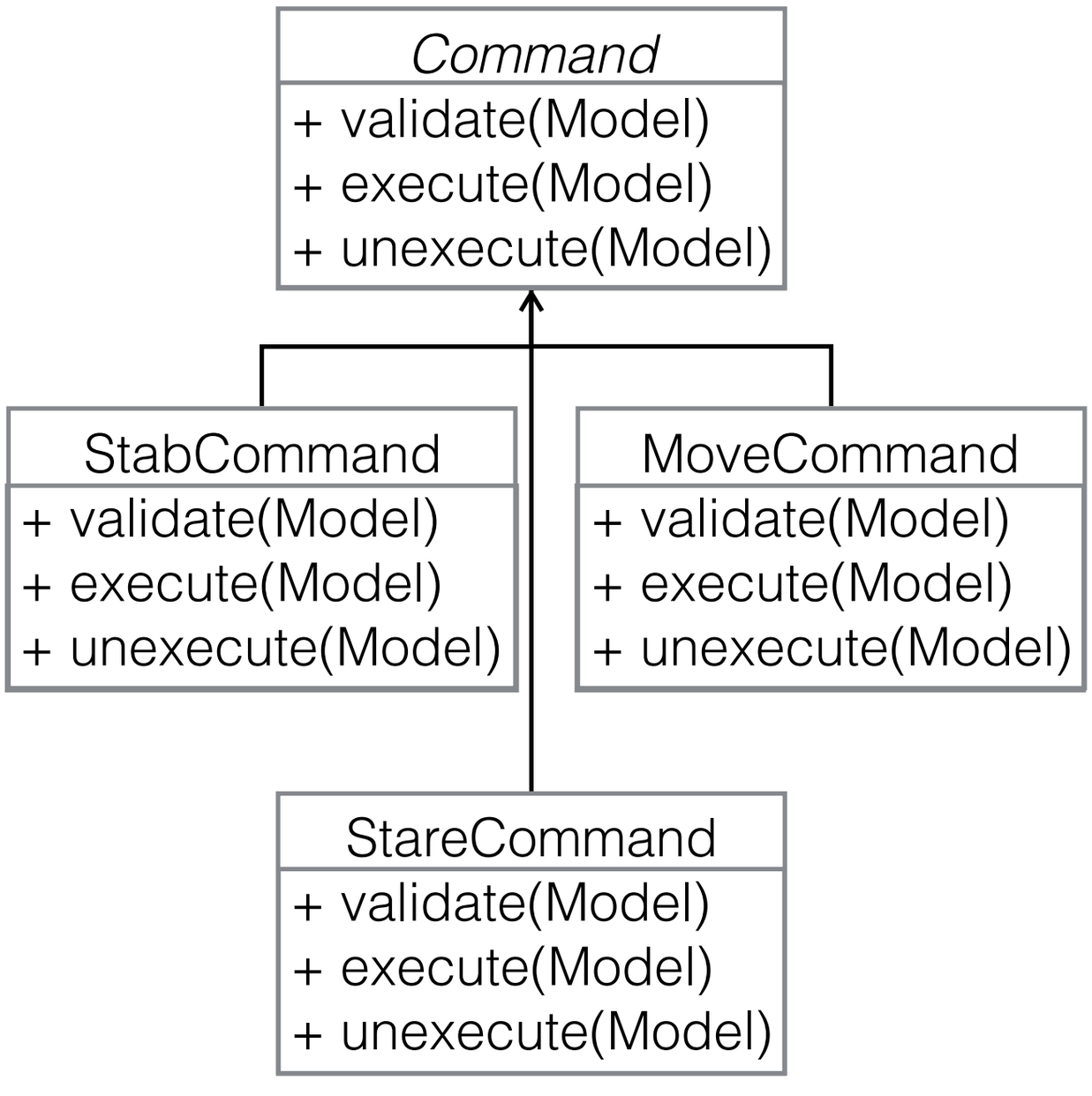
- Einen Zug “ausprobieren”, ohne das Model zu ändern
- Ich möchte wissen, wie die Welt nach dem Zug aussieht, um die Heuristik für diesen Zustand berechnen zu können

Ich simuliere,

➡ indem ich den Zug auf einer Kopie des Models ausführe

➡ indem ich im Command-Pattern ein “unexecute()” implementiere

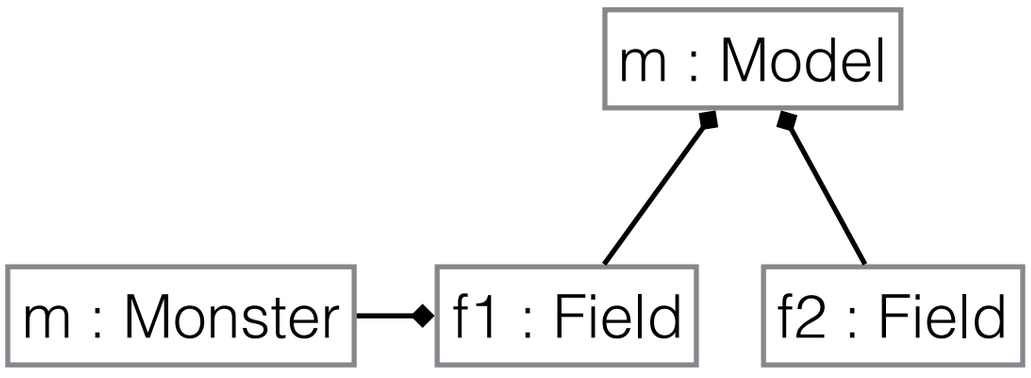
Command-Pattern



- Oberklasse “Command” mit abstrakten Methoden
- eine Unterklasse für jeden Command mit Implementierungen

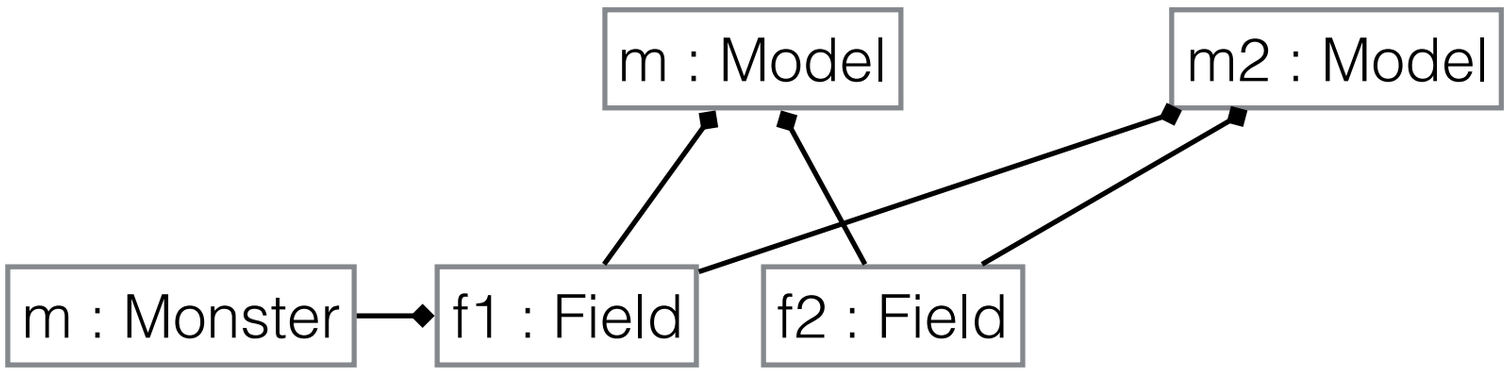
Kopieren des Model

```
Model copy(Model m) {
  Model m2 = new Model();
  m2.f1 = m.f1;
  m2.f2 = m.f2;
  return m2;
}
```



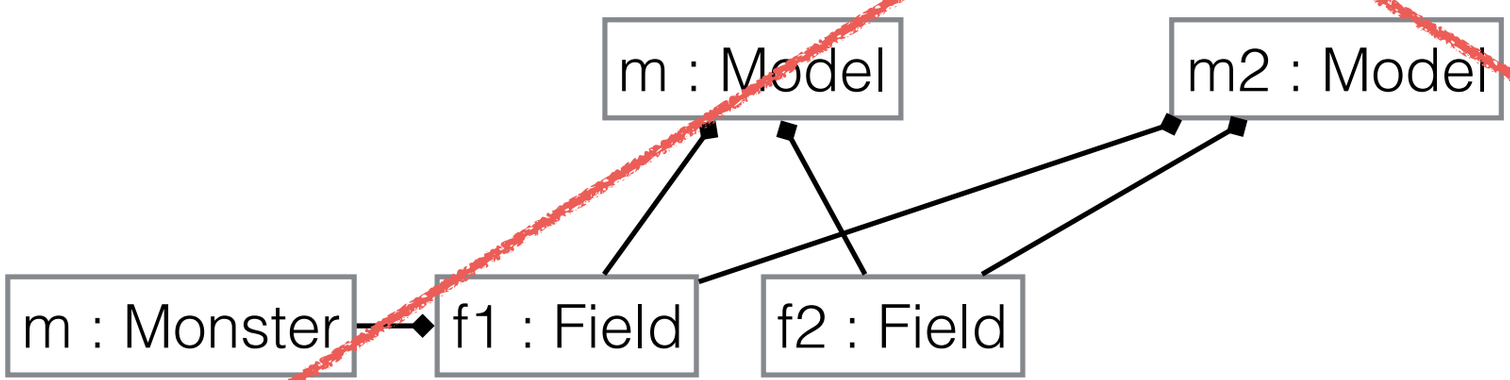
Kopieren des Model

```
Model copy(Model m) {
  Model m2 = new Model();
  m2.f1 = m.f1;
  m2.f2 = m.f2;
  return m2;
}
```



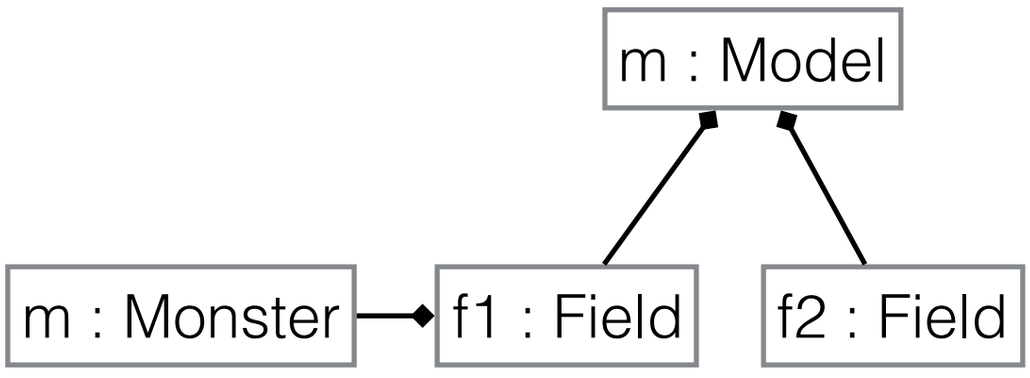
Kopieren des Model

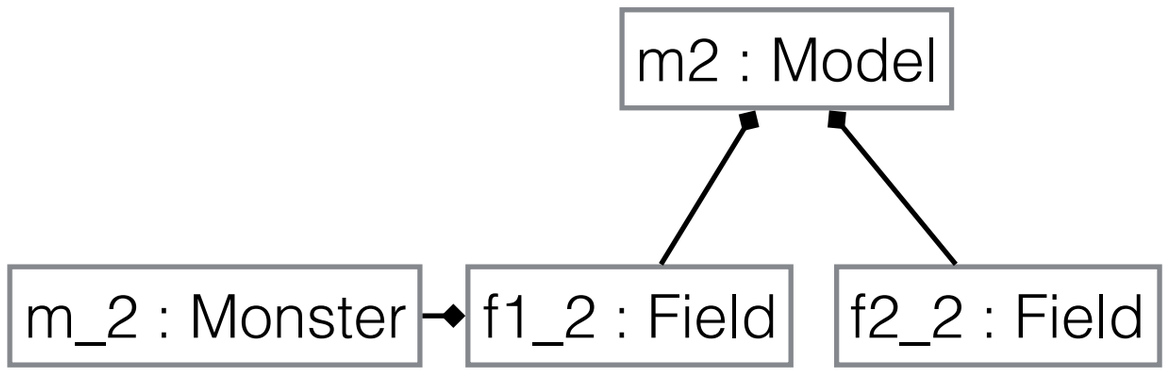
```
Model copy(Model m) {
  Model m2 = new Model();
  m2.f1 = m.f1;
  m2.f2 = m.f2;
  return m2;
}
```



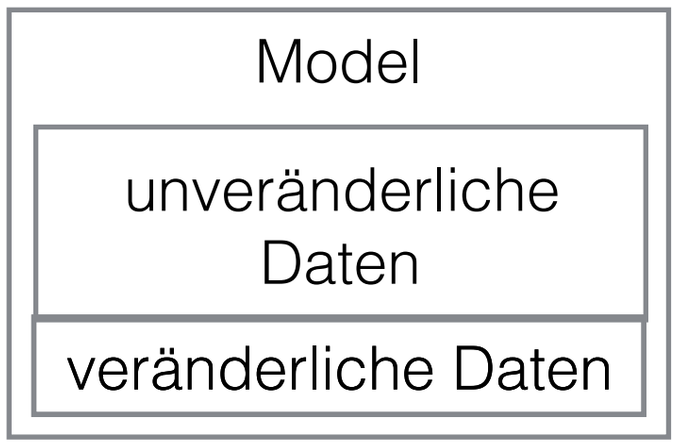
Kopieren des Model

```
Model copy(Model m) {
  Model m2 = new Model();
  m2.f1 = copy(m.f1);
  m2.f2 = copy(m.f2);
  return m2;
}
```





Memento-Pattern



Memento-Pattern

1. kopieren

Memento-Pattern

Memento-Pattern

2. Simulieren

Model

unveränderliche Daten

veränderliche Daten

veränderliche Daten

Memento-Pattern

Memento-Pattern

3. Zurücksetzen

Memento-Pattern

3. Zurücksetzen

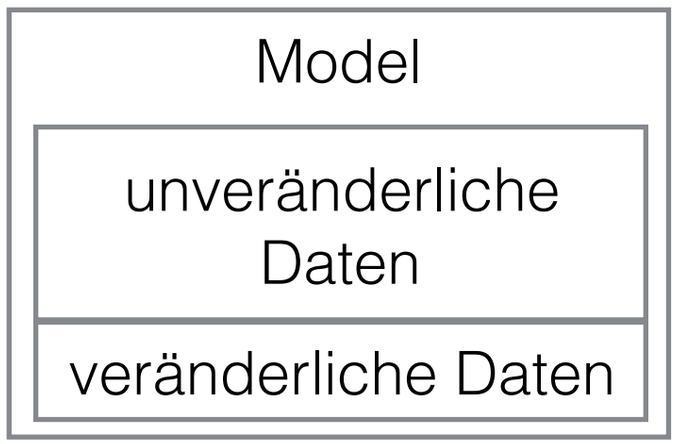
Model

unveränderliche Daten

veränderliche Daten

Memento-Pattern

3. Zurücksetzen



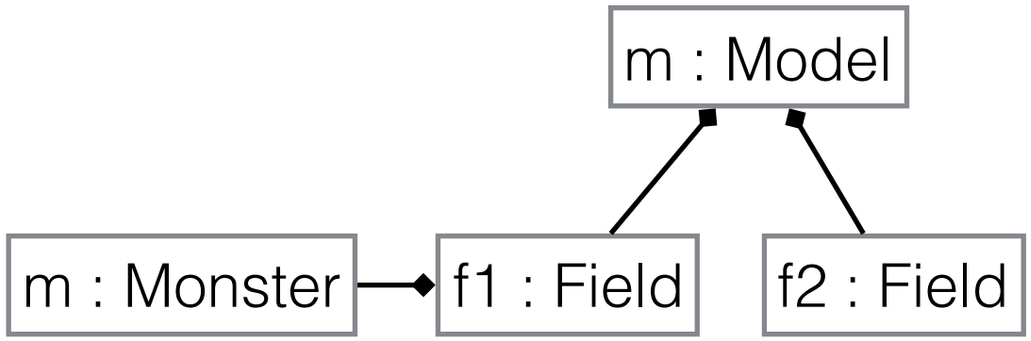
Vorteile:

- Leichter zu implementieren als unexecute() (weniger Code!)
- Weniger Daten zum Kopieren als bei voller Kopie

Memento-Pattern

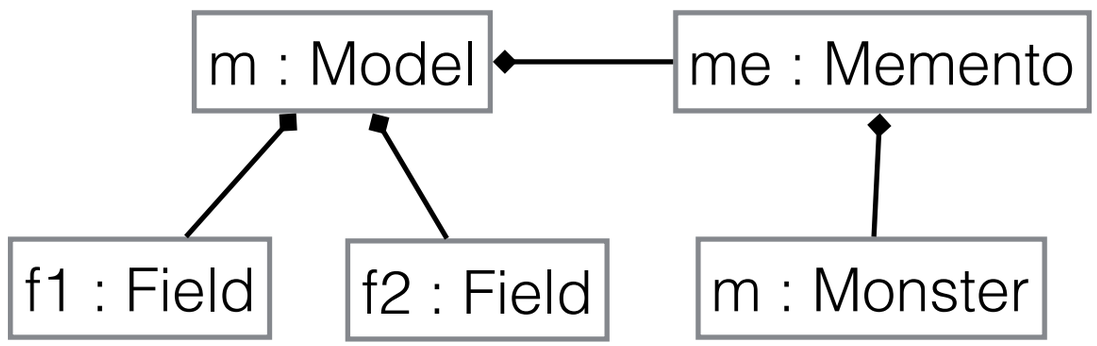
Nachteile

Normaler Entwurf



- Der Entwurf entfernt sich vom intuitiven Objektmodell

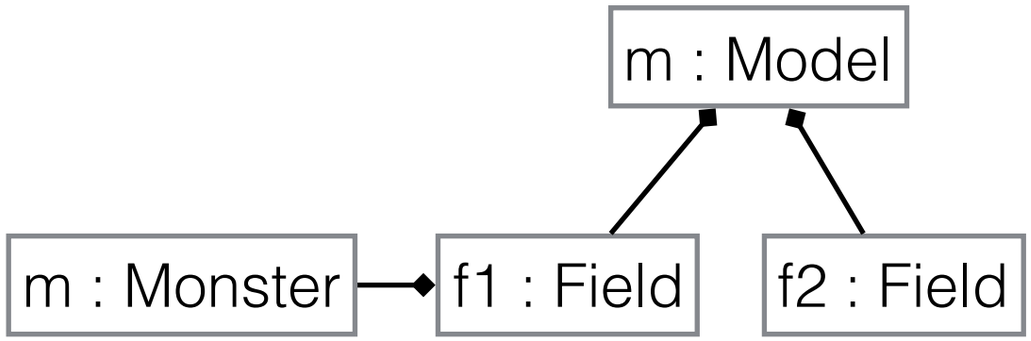
Mit Memento-Pattern



Memento-Pattern

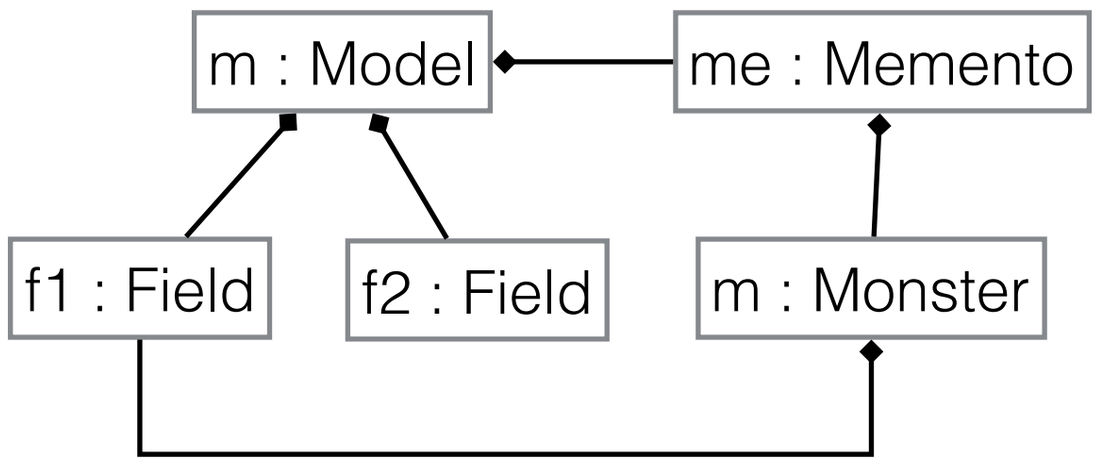
Nachteile

Normaler Entwurf



- Der Entwurf entfernt sich vom intuitiven Objektmodell

Mit Memento-Pattern

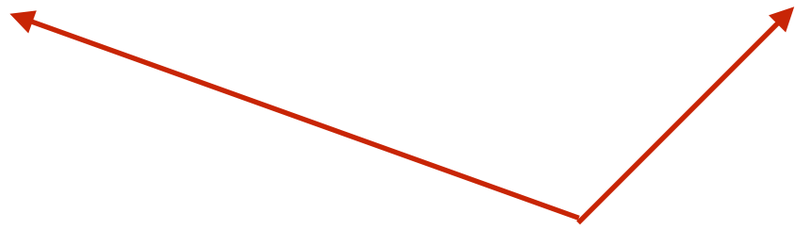


Single-Dot Rule

```
Model model = /* … */;
Monster m = model
    .getField(1, 3).getMonster();
```

Single-Dot Rule

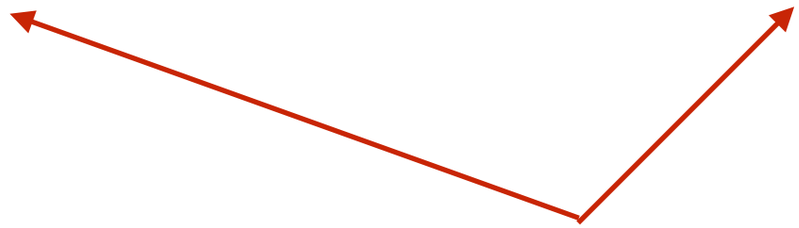
```
Model model = /* … */;
Monster m = model
    .getField(1, 3).getMonster();
```



Single-Dot Rule?

Single-Dot Rule

```
Model model = /* … */;
Monster m = model
    .getField(1, 3).getMonster();
```



Single-Dot Rule?

```
Model model = /* … */;
Field f = model.getField(1, 3);
Monster m = f.getMonster();
```

Single-Dot Rule

```
Model model = /* … */;
Monster m = model
    .getField(1, 3).getMonster();
```

Single-Dot Rule?

```
Model model = /* … */;
Field f = model.getField(1, 3);
Monster m = f.getMonster();
```

Andere Syntax, selbe Semantik, das zählt nicht!

Stellen Sie sich vor, Sie wollen auf Memento-Pattern umbauen

```
Model model = /* … */;
Monster m = model
    .getField(1, 3).getMonster();
```

```
Model model = /* … */;
Field f = model.getField(1, 3);
Memento memento = model.getMemento();
Monster m = memento.getMonsterAt(f);
```

Single-Dot Rule eingehalten

```
Model model = /* … */;
Monster m = model
      .getField(1, 3).getMonster();
```

```
Model model = /* … */;
Monster m = model.getMonster(1, 3);

class Model {

  Monster getMonster(int x, int y) {
    Field f = getField(x, y);
    return f.getMonster();
  }

}
```

Anforderungen an das Model

- Ich muss informiert werden, wenn ich dran bin.
- Ich muss prüfen können, ob ein Zug zulässig ist.
- Ich muss mitteilen können, was ich machen möchte.
- Ich muss abfragen können, wie die Spielwelt aussieht.
- Ich muss Spielzüge simulieren können.

Heuristiken

- eigene Lebenspunkte maximieren
- Lebenspunkte der Gegner minimieren
- Entfernung zum Gegner (minimieren oder maximieren?)
- …

Mehrere Heuristiken kombinieren (1)

- Gewichtete Summe (weighted sum):

$$h_c(m) = \alpha_1 h_1(m) + \alpha_2 h_2(m) + ... + \alpha_n h_n(m)$$

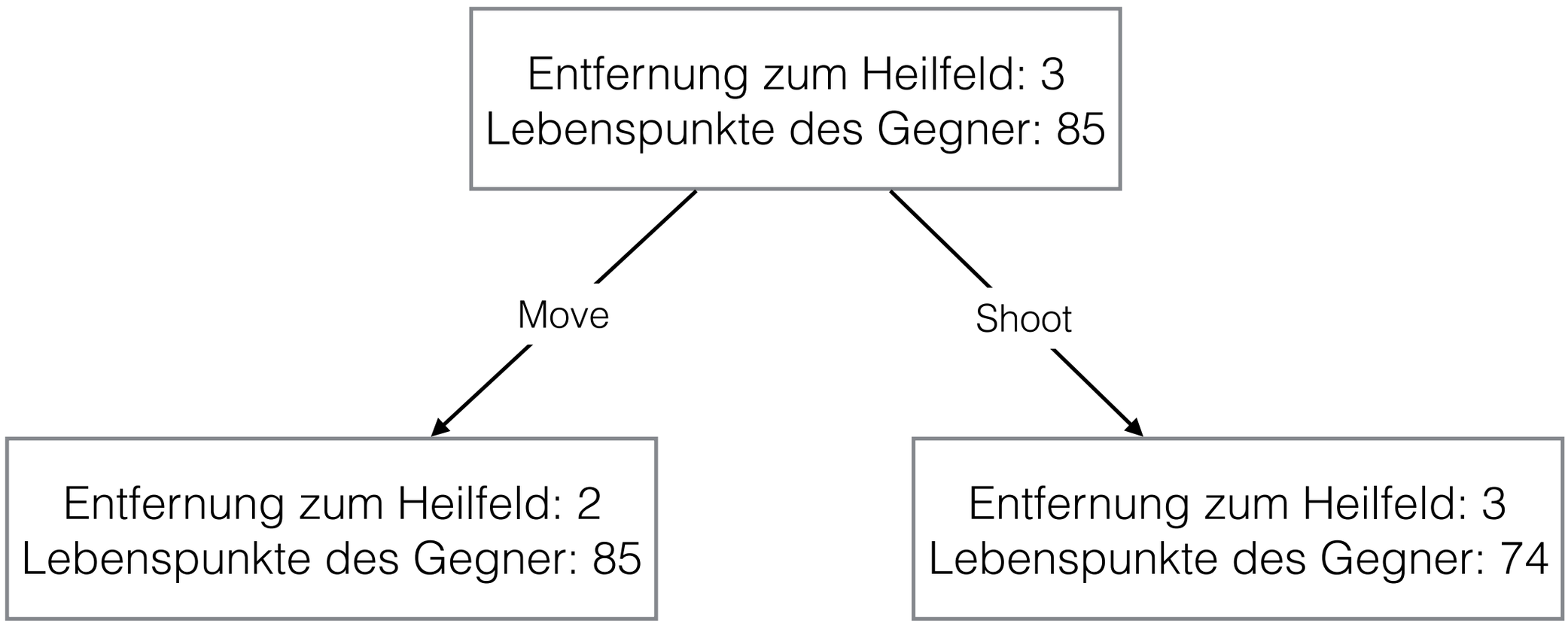
Mehrere Heuristiken kombinieren (1)

- Gewichtete Summe (weighted sum):

$$h_c(m) = \alpha_1 h_1(m) + \alpha_2 h_2(m) + ... + \alpha_n h_n(m)$$

Steuern, welche Heuristik am Wichtigsten ist

Mehrere Heuristiken kombinieren (2)



Lebenspunkte ändern sich viel schneller als Entfernungen (und sind potenziell größer), deshalb dominieren sie die Heuristik

Heuristiken normalisieren

- Heuristiken zwischen 0 und 1 bringen:

$$h'(m) = \frac{h(m)}{\max h}$$

- Heuristiken auf eine gemeinsame Einheit bringen:
  - statt: Entfernung zum Heilfeld
    Wie viele Commands brauche ich noch da hin?

  - statt: Lebenspunkte des Gegners
    Wie viele Commands brauche ich noch um ihn zu töten?

Heuristiken invertieren

- Maximieren oder Minimieren?
  - Ob “kleiner Wert” guter Zustand oder schlechter Zustand heißt, ändert nur den Vergleich im Algorithmus
- Heuristik umdrehen:

$$h'(m) = -h(m)$$

$$h'(m) = \max h - h(m)$$

Anforderungen an das Model

- Ich muss informiert werden, wenn ich dran bin.
- Ich muss prüfen können, ob ein Zug zulässig ist.
- Ich muss mitteilen können, was ich machen möchte.
- Ich muss abfragen können, wie die Spielwelt aussieht.
- Ich muss Spielzüge simulieren können.