Informatik II - Übung 9

Pascal Schärli

IloveExercise9@pascscha.ch

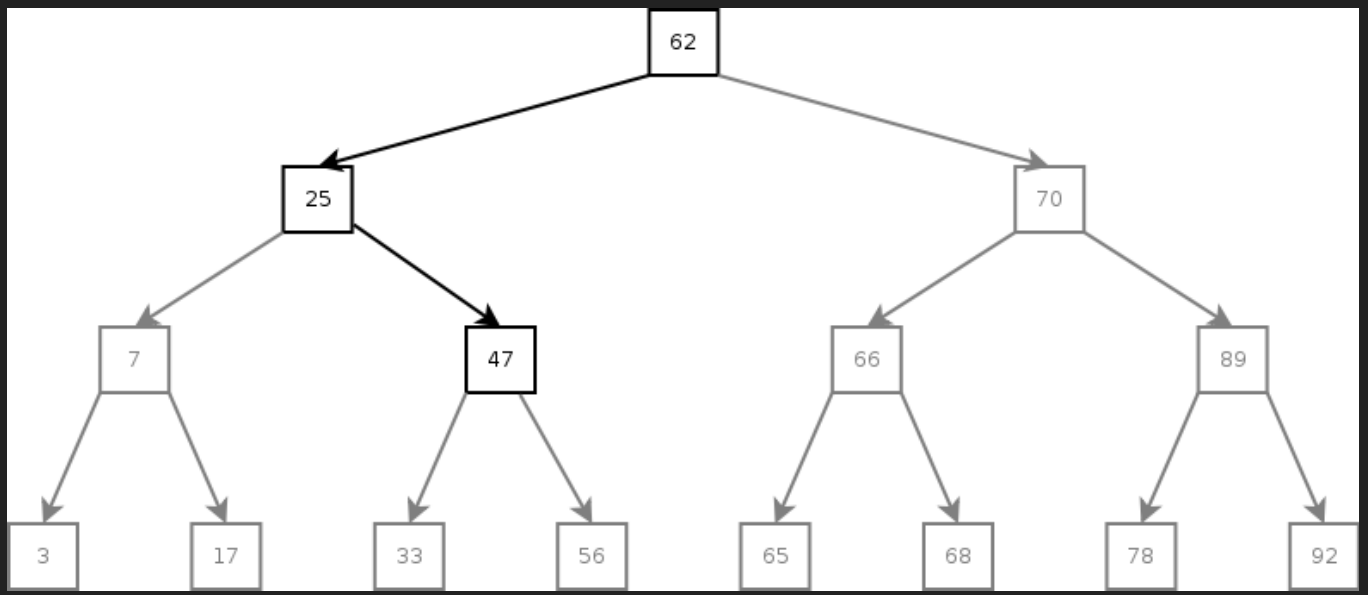
20.11.2020

ETH zürich

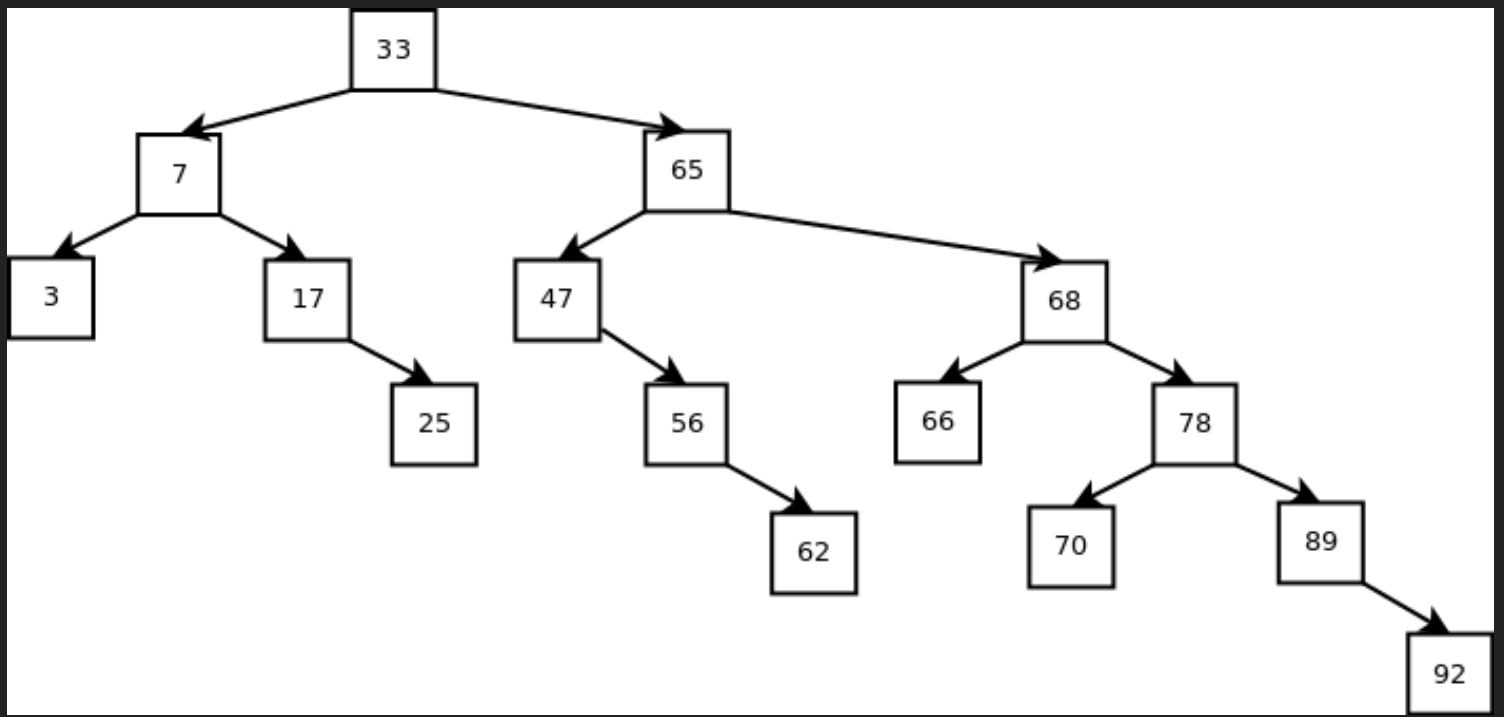
Nachbesprechung

Bild: look again by corpus delicti from the Noun Project

Binäre Suche Textaufgaben



Binäre Suche Textaufgaben



Die Strategie ist schlechter, da die Suchtiefe im Durchschnitt grösser ist.

Binäre Suche Programmieren

```
// IMeasure
private int factor;
private int numberofCalls;

BinarySearch() {
    factor = 2;
    numberofCalls = 0;
}

public void setFactor(int factor) {
    this.factor = factor;
}

public int getNumberofCalls() {
    return numberofCalls;
}
```

Binäre Suche Programmieren

```java
public Value find(List<Unit<Key, Value>> haystack, Key needle) {
    numberofCalls = 0;
    return findRec(haystack, needle, 0, haystack.size());
}

private Value findRec(List<Unit<Key, Value>> haystack, Key needle, int begin, int end) {
    numberofCalls++; // IMeasure
    if (begin == end) {
        return null;
    }
    int middle = begin + (end - begin) / factor;
    Unit<Key, Value> middleThing = haystack.get(middle);
    int match = needle.compareTo(middleThing.key);
    if (match == 0) {
        return middleThing.value;
    } else if (match < 0) {
        return findRec(haystack, needle, begin, middle);
    } else {
        return findRec(haystack, needle, middle + 1, end);
    }
}
```

Binäre Suche Programmieren

```java
static final int[] keys = {3, 7, 17, 25, 33, 47, 56, 62, 65, 66, 68, 70,
        78, 89, 92};

static ArrayList<Unit<Integer, Integer>> numbers;

private static void createKeys() {
    numbers = new ArrayList<Unit<Integer, Integer>>();
    for (int key : keys) {
        numbers.add(new Unit<Integer, Integer>(key, key));
    }
}
```

Binäre Suche Programmieren

```java
// Suche nach allen vorhandenen Zahlen
private static void measure1(int factor) {
    BinarySearch<Integer, Integer> search = new BinarySearch<Integer, Integer>();
    search.setFactor(factor);

    int sumCalls = 0;

    for (int key : keys) {
        search.find(numbers, key);
        sumCalls += search.getNumberofCalls();
    }

    double result = ((double) sumCalls) / keys.length;

    System.out.println(String.format("Messung Frage 1, Faktor: %d, mittlere Suchtiefe: %.2f"
}
```

```
Messung Frage 1, Faktor: 2, mittlere Suchtiefe: 3.27
Messung Frage 1, Faktor: 3, mittlere Suchtiefe: 3.40
```

Binäre Suche Programmieren

```java
// Suche nach allen Zahlen von 0 bis 99
private static void measure2(int factor) {
    BinarySearch<Integer, Integer> search = new BinarySearch<Integer, Integer>();
    search.setFactor(factor);

    int sumCalls = 0;

    for (int i = 0; i < 100; i++) {
        search.find(numbers, i);
        sumCalls += search.getNumberofCalls();
    }

    double result = ((double) sumCalls) / 100;

    System.out.println(String.format("Messung Frage 2, Faktor: %d, mittlere Suchtiefe: %.2f"
}
```

```
Messung Frage 2, Faktor: 2, mittlere Suchtiefe: 4.74
Messung Frage 2, Faktor: 3, mittlere Suchtiefe: 4.96
```

Binäre Suche Programmieren

```java
// Suche nach allen Zahlen von 0 bis 9
private static void measure3(int factor) {
    BinarySearch<Integer, Integer> search = new BinarySearch<Integer, Integer>();
    search.setFactor(factor);

    int sumCalls = 0;

    for (int i = 0; i < 10; i++) {
        search.find(numbers, i);
        sumCalls += search.getNumberofCalls();
    }

    double result = ((double) sumCalls) / 10;

    System.out.println(String.format("Messung Frage 3, Faktor: %d, mittlere Suchtiefe: %.2f"

}
```

```
Messung Frage 3, Faktor: 2, mittlere Suchtiefe: 4.70
Messung Frage 3, Faktor: 3, mittlere Suchtiefe: 3.90
```

Rucksackproblem und Backtracking Textaufgaben

- Es gibt nicht immer nur eine optimale Lösung
- Gegenbeispiel:
  Gewichte: 1, 2, 3
  Werte: 1, 1, 2
  maximales Gewicht: 3
  - erste Lösung: true, true, false
  - zweite Lösung: false, false, true

Rucksackproblem und Backtracking Programmieren

```java
public class BruteForce implements IRucksack {
    public Selection findBest(ArrayList<Integer> values, ArrayList<Integer> weights, int maxWeight) {
        if (values.size() != weights.size())
            throw new IllegalArgumentException("sizes of values and weights vectors are not equal");

        Selection bestSelection = null;
        int maxValue = -1;

        final int max = (int) Math.pow(2, values.size());
        for (int i = 0; i < max; i++) {
            Selection selection = new Selection(values.size(), i);
            if (selection.sum(weights) <= maxWeight) {
                int value = selection.sum(values);
                if (value >= maxValue) {
                    bestSelection = selection;
                    maxValue = value;
                }
            }
        }
        return bestSelection;
    }
}
```

Rucksackproblem und Backtracking Programmieren

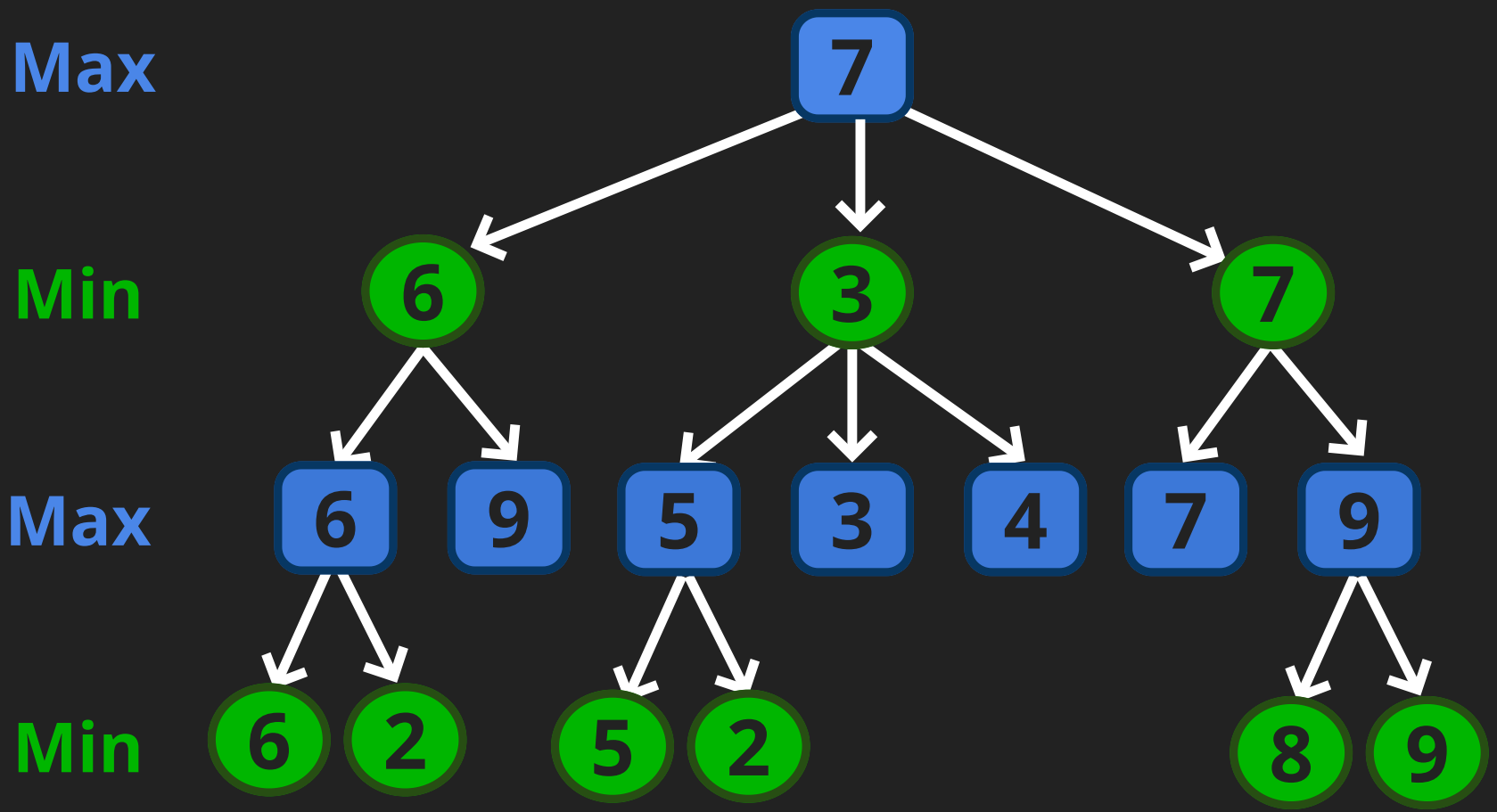
```java
private Selection find(Selection selection, int weight, ArrayList<Integer> values,
                       ArrayList<Integer> weights, int maxWeight) {

    final int depth = selection.size();
    if (depth == values.size()) {
        return selection;
    }

    Selection without = new Selection(depth + 1, selection.bits());
    without.set(depth, false);
    Selection resultWithout = find(without, weight, values, weights, maxWeight);

    if (weight + weights.get(depth) <= maxWeight) {
        Selection with = new Selection(depth + 1, selection.bits());
        with.set(depth, true);

        Selection resultWith = find(with, weight + weights.get(depth), values,
                                    weights, maxWeight);
        if (resultWith.sum(values) > resultWithout.sum(values)) {
            return resultWith;
        }
    }
    return resultWithout;
}
```

Rucksackproblem und Backtracking Programmieren

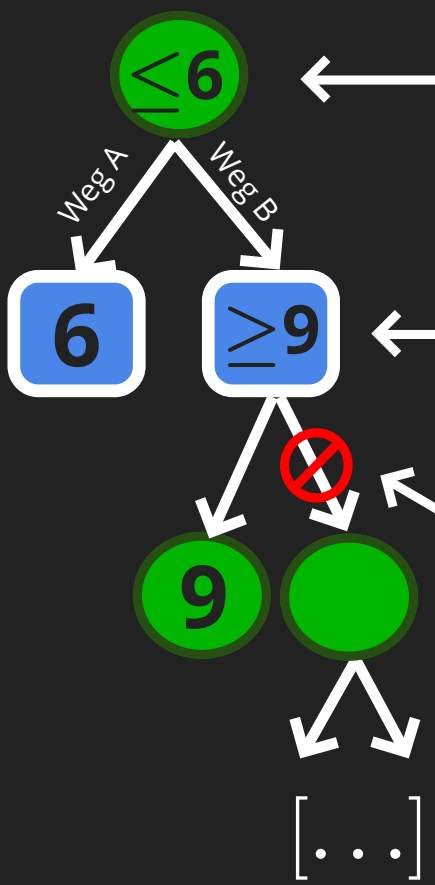
```java
public Selection findBest(ArrayList<Integer> values, ArrayList<Integer> weights,
                          int maxWeight) {
    if (values.size() != weights.size())
        throw new IllegalArgumentException("sizes of values and weights vectors are not equal");

    Selection result = find(new Selection(0), 0, values, weights, maxWeight);
    return result;
}
```

Vorlesung

Minimax



Alpha-Beta



Da Min den Wert aller Kinder minimiert, muss dieser Knoten ≤ 6 sein

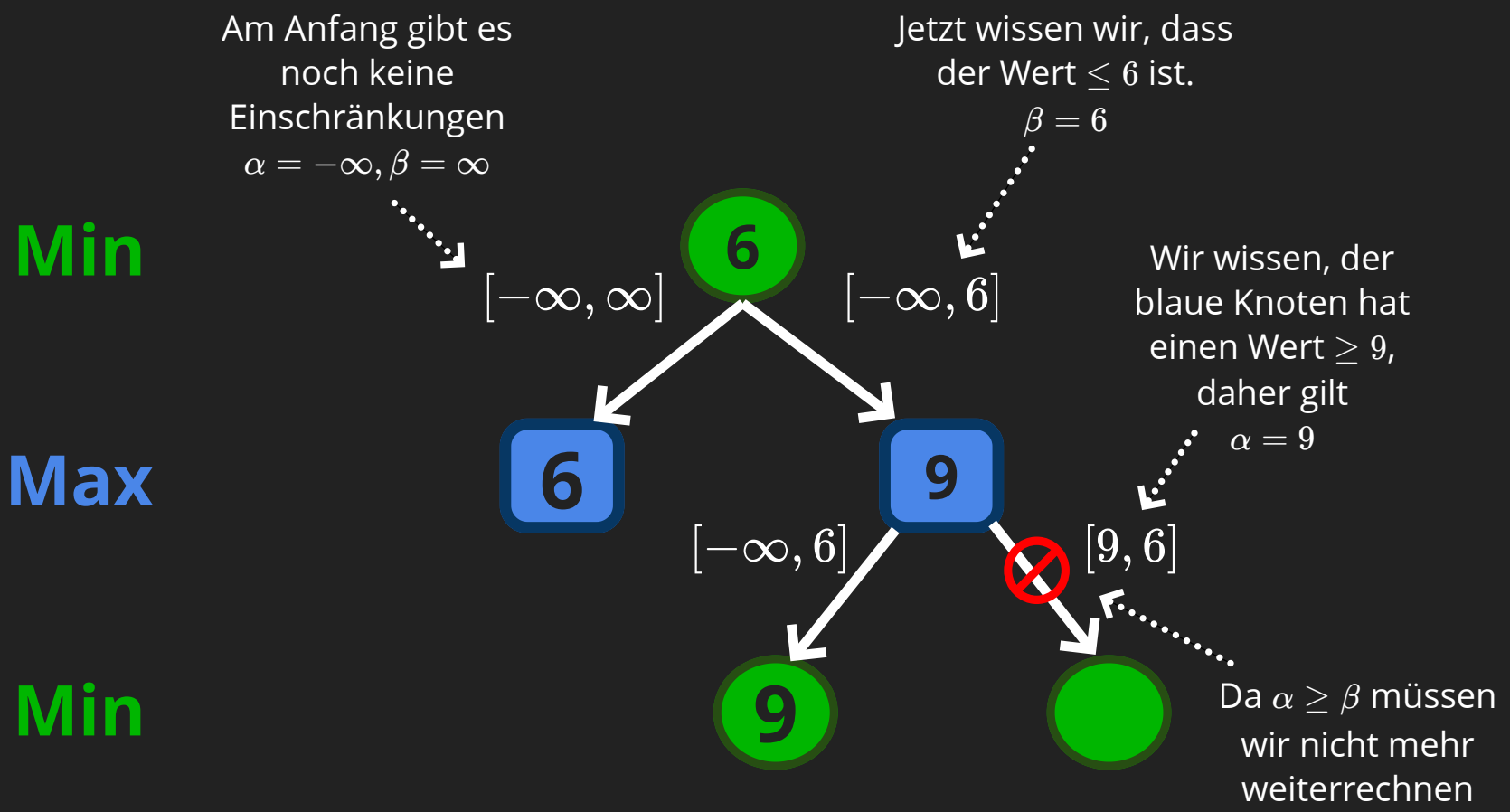
Da Max den Wert aller Kinder maximiert, muss dieser Knoten ≥ 9 sein

Da Min sowieso den Weg A wählen wird, müssen wir den Wert der restlichen Knoten nicht mehr auswerten

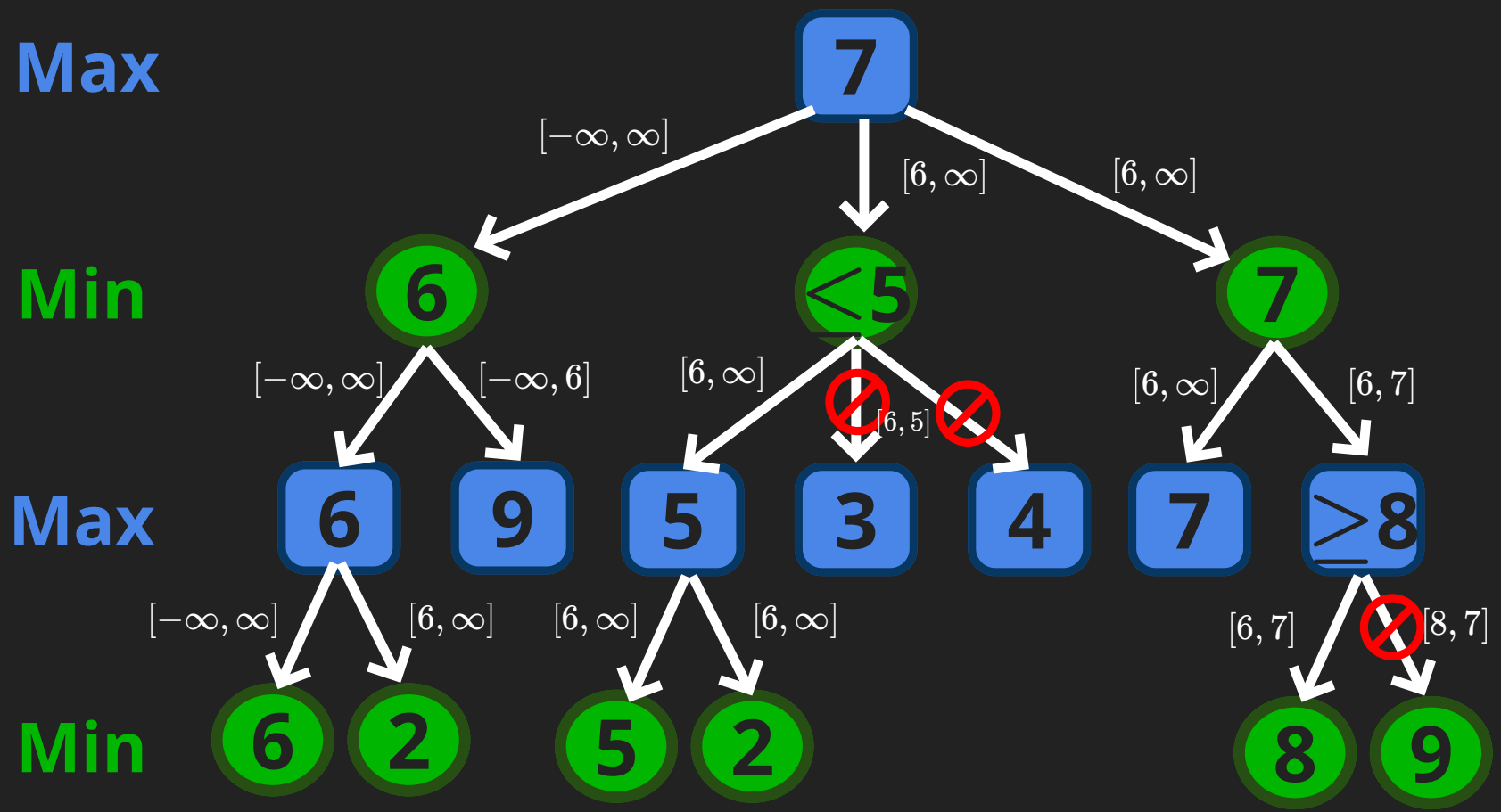
Alpha-Beta

- Jeder Knoten hat zwei Einschränkungen:
  - α: Mindestwert, wecher Max sicher erreichen kann
  - β: Maximalwert, welcher Min höchstens einstecken muss
- Das Intervall $[\alpha, \beta]$ beinhaltet alle möglichen Werten für einen Knoten, mit welchen dieser für die Auswertung relevant wäre
- Falls $\alpha \geq \beta \Rightarrow$ Cut
  - Wenn der α Wert den β Wert überschreitet muss man den Knoten nicht mehr weiter auswerten, da dieser Weg sowieso nicht gewählt werden wird.
- Wir notieren die Werte mit $[\alpha, \beta]$ bei den Kanten, welche vom Knoten wegführen.
- Schnitte nach dem Min-Knoten nennt man α-Schnitte,
  Schnitte nach dem Max-Knoten nennt man β-Schnitte.
  (1h vor der Prüfung schnell auswendig lernen)

Alpha-Beta


Am Anfang gibt es noch keine Einschränkungen
$\alpha = -\infty, \beta = \infty$
Jetzt wissen wir, dass der Wert ≤ 6 ist.
$\beta = 6$
Wir wissen, der blaue Knoten hat einen Wert ≥ 9, daher gilt
$\alpha = 9$
Da $\alpha \geq \beta$ müssen wir nicht mehr weiterrechnen
Min
Max
Min
6
6
9
9
$[-\infty, \infty]$
$[-\infty, 6]$
$[-\infty, 6]$
$[9, 6]$

Alpha-Beta


Max
Min
Max
Min
7
6
≤5
7
6
9
5
3
4
7
≥8
6
2
5
2
8
9
$[-\infty, \infty]$
$[6, \infty]$
$[6, \infty]$
$[-\infty, \infty]$
$[-\infty, 6]$
$[6, \infty]$
$[6, 5]$
$[6, \infty]$
$[6, 7]$
$[-\infty, \infty]$
$[6, \infty]$
$[6, \infty]$
$[6, \infty]$
$[6, 7]$
$[8, 7]$

Übungstool

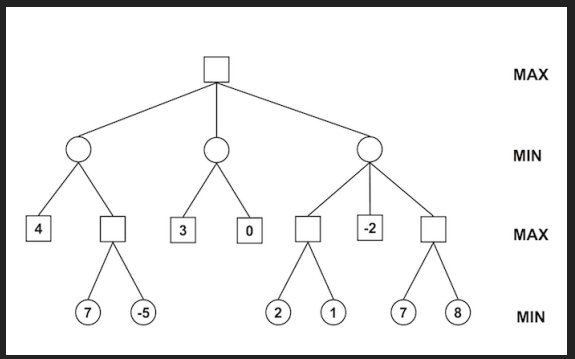
- pascscha.ch/info2/abTreePractice/
- Auswerten von einem Spielbaum mit α, β-Pruning
- Zufällig generierte Bäume $\rightarrow$ genug Übungsmaterial um bis zum Hitzetod des Universums α, β-Pruning zu üben


K!

Vorbesprechung

Spieltheorie

- Was ist die Suchtiefe
  - Ein Baum mit nur einem Knoten hat die Suchtiefe 0
- Wendet Minimax auf dem Spielbaum an
- Was ist die optimale Strategie für Max?
  - Bei jedem Max-Knoten den besten Zug bestimmen
- Wendet die Alpha Beta an
  - Beschriftet Kanten mit den $[\alpha, \beta]$ Werten
  - Zeichnet ein, welche Äste ihr nicht berechnen müsst



Reversi [Teil 3]

Minimax Algorithms

- Programmiert einen Reversi Spieler, welcher mithilfe vom Minimax Algorithmus den bestmöglichen Zug bestimmt.
- Da es nicht möglich ist das Spiel bis zum Schluss durchzurechnen soll die Suchtiefe einstellbar sein.
- Als Bewertung könnt ihr vorerst wieder die Steindifferenz nehmen

Reversi [Teil 3]

Minimax Algorithms

Es könnte hilfreich sein, eine hilfs-Klasse zu schreiben, welche eure besten Züge speichert.

```
class BestMove {
  /**
   * The coordinates of the proposed next move
   */
  public Coordinates coord;

  /**
   * The value of the proposed next move according to the min-max analysis
   */
  public int value;

  public BestMove(int value, Coordinates coord) {
    this.value = value;
    this.coord = coord;
  }
}
```

Reversi [Teil 3]

Zeitbegrenzung

- Im Turnier hat euer Spieler nur 5 Sekunden Zeit
- Verändert eure **nextMove** Funktion so, dass sie die Minimax Analyse immer wieder mit grösseren Suchtiefen durchführt, bis die Zeitlimite erreicht ist.
- System.currentTimeMillis() gibt die Zeit seit dem 1. Januar 1970 in Millisekunden an, verwendet dies um eure Zeit zu messen.

Reversi [Teil 3]

Zeitbegrenzung

Erstellt eine eigene Exception, die ihr bei einem Timeout werfen könnt

```
class Timeout extends Throwable {
  private static final long serialVersionUID = 1L;
}
```

Reversi [Teil 3]

Zeitbegrenzung

Die max-Funktion sieht in etwa so aus:

```
 private BestMove max(int maxDepth, long timeout, GameBoard gb, int depth) throws Timeout {
        if (time is up){
        throw new Timeout();
    }

    if(depth == maxDepth){
        //TODO: use eval() to determine the value of this position
    }

    for (all possible moves)
        clone board;
        make move on clone;
        temp_move = min(maxDepth, timeout, cloned_board, depth+1);
        save in best_move if best move so far;
    return best_move
}
```

Reversi [Teil 3]

Bessere Bewertungsfunktion eval()

- **Mobility**
  - Wie viele Zugmöglichkeiten habe ich, bzw. der Gegner.
- **Ecken**
  - Ecken sind Strategisch sehr Wertvoll
- **Anzahl Steine**
  - Zumindest am Anfang ist es meistens besser, möglichst wenig Steine zu haben, um die Mobilität zu maximieren. Am Schluss ändert sich das dann offensichtlich.

- Gute Gewichtung durch ausprobieren herausfinden
  - Wenn ihr in der Run-Configuration -f anfügt werden Animationen nicht gezeigt
  - mit -c wird das GUI gar nicht angezeigt

Viel Spass!


I'M JUST OUTSIDE TOWN, SO I SHOULD BE THERE IN FIFTEEN MINUTES.
ACTUALLY, IT'S LOOKING MORE LIKE SIX DAYS.
NO, WAIT, THIRTY SECONDS.
THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.