Fünf in eine Reihe

Optimierung des MinMax-Algorithmus

Fabian Traxler, 1553958

Ausgangssituation

Spieler, der zuerst 5 Steine in einer Reihe hat, gewinnt

Spielfeldgröße unbegrenzt

Zug Vorhersage durch Computer:

- Berechnung mit Min-Max Algorithmus (NegMax Variante)

Algorithmische Optimierungen

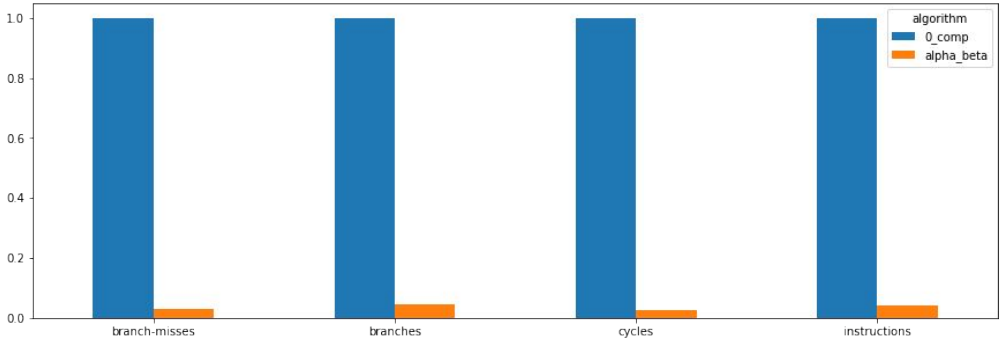
Alpha-Beta Pruning:

- Stop der Suche, wenn feststeht, dass es bereits bessere Möglichkeiten gibt

Transpositionstabellen:

- Wert von gewissen Ausgangssituationen speichern, um redundante Berechnungen zu vermeiden
- Hash Tabelle verwendet mit djb2-Hash
  - \+ start1 & Depth 4 -> 125.698 ersparte Kalkulationen
  - \- Für alle Board States muss Hash berechnet und gecheckt werden

Ergebnisse - Algorithmische Verbesserungen

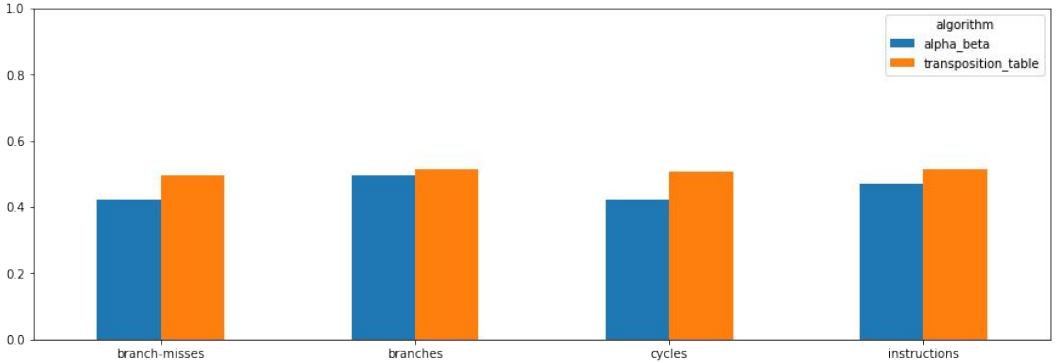


Comp vs. Alpha-Beta:

- Klare Verbesserung
- Gründe:
  - Deutlich weniger Berechnungen nötig wegen frühzeitigen Such-Abbruch

Alpha-Beta vs. Transpositions Tabellen:

- Verschlechterung
- Gründe:
  - Bei geringer Tiefe wenig Nutzen der Speicherung
  - Hash Berechnung jedes mal nötig

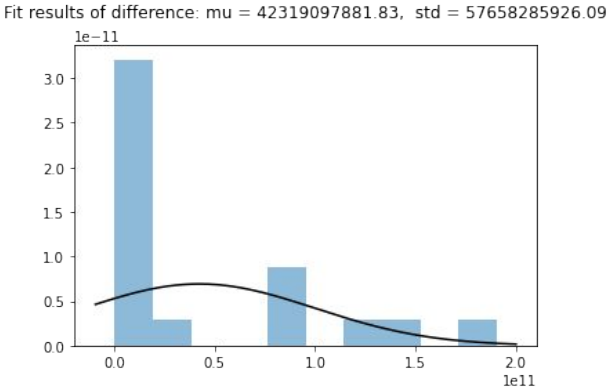


Wilcoxon Test for paired Samples

Startpunkte: x2-x9 + start1 mit Tiefe 2,3,(4,5)

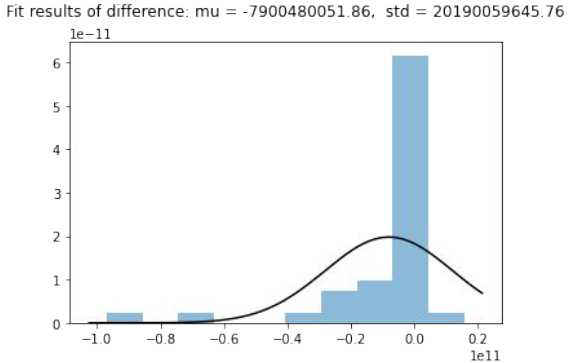
H0: Beide Algorithmen benötigen gleich viele Zyklen
HA: Optimierter Algorithmus benötigt weniger Zyklen

Comp vs. Alpha-Beta



p-Wert = 0.000
=> Alpha-Beta Algorithmus braucht signifikant weniger Zyklen

Alpha-Beta vs. Transposition-Table



p-Wert = 0.9861
=> Transpositionstabllen-Version braucht nicht signifikant weniger Zyklen

Code Optimierungen

1. Wert nur berechnen, wenn tatsächlich benötigt

```
long v = value( b: b1, player, depth);
if (depth > 0) {
    long x, y;
    bestmove( depth: depth - 1,  b: b1,  vp: &v,  bestx: &x,  besty: &y,  alpha: -beta,  beta: -*vp);
    v = -v;
}
```

```
long v;
if (depth > 0) {
    long x, y;
    bestmove( depth: depth - 1,  b: b1,  vp: &v,  bestx: &x,  besty: &y,  alpha: -beta,  beta: -*vp);
    v = -v;
} else {
    v = value( b: b1, player, depth);
}
```

2. Unnötiges Pointer dereferencing vermeiden

```
if (v > *vp) {
    *vp = v;
    *bestx = j;
    *besty = i;
    if (*vp >= beta) {
        return;
    }
}
```

```
if (v > *vp) {
    *vp = v;
    *bestx = j;
    *besty = i;
    if (v >= beta) {
        return;
    }
}
```

Code Optimierungen

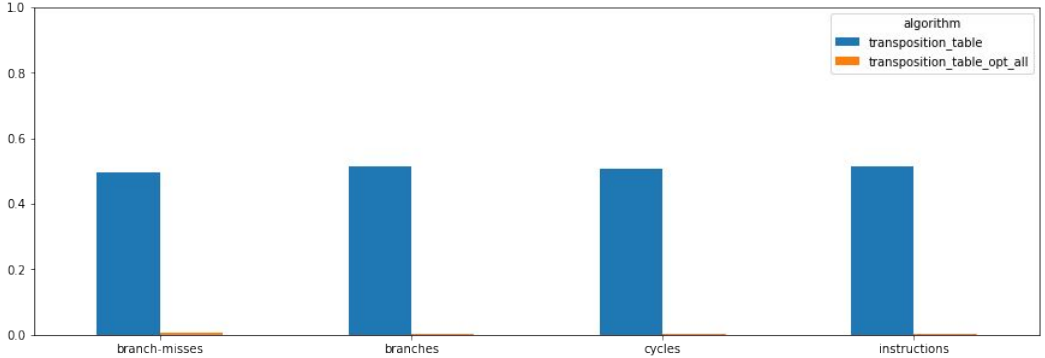
3. Doppelte Value Berechnung von Spieler und Gegenspieler zusammenführen

Schleifenoverhed vermeiden

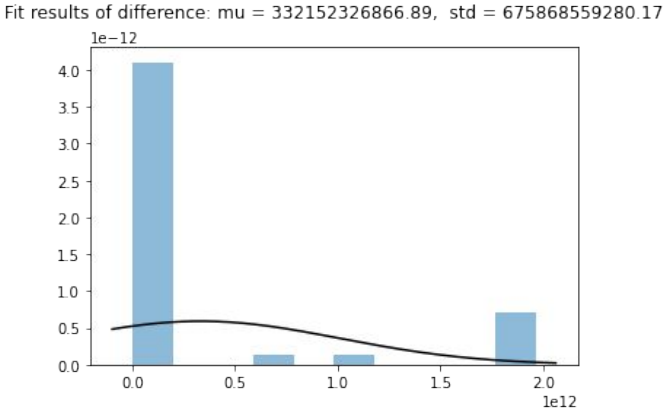
```
long new_value = value1(b,player) - value1(b,  player: otherplayer(player));
```

```
for (long i=0; i<b->maxy; i++)
  for (long j=0; j<b->maxx; j++) {
    if (b->board[i*b->maxx+j]==player) {...}
    if (b->board[i*b->maxx+j]==other) {...}
  }
```

Ergebnisse - Code Optimierung



Nochmal ein klare Verbesserung durch die Optimierungsschritte!

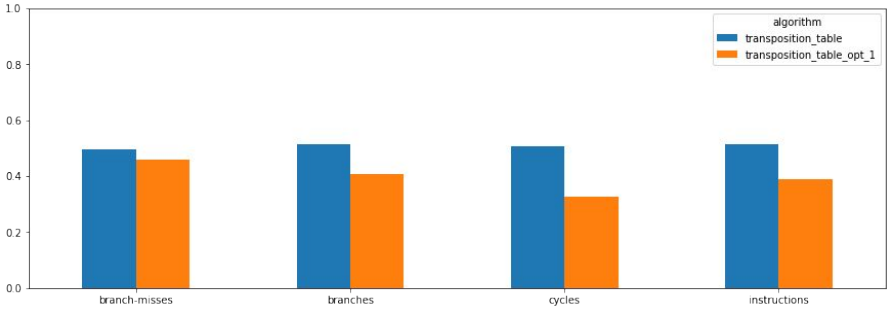


p-Wert = 0.000
=> Optimierter Algorithmus braucht signifikant weniger Zyklen

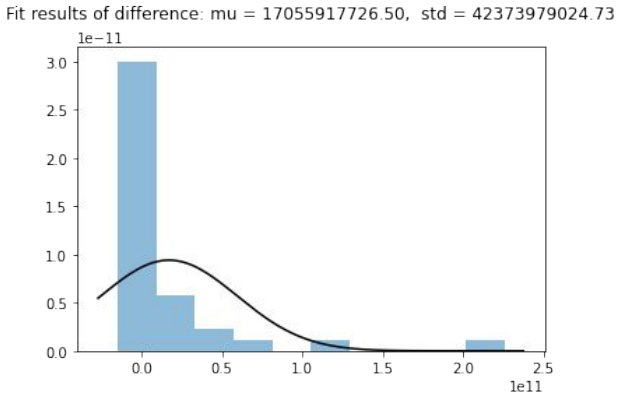
Welche Optimierungsschritte waren tatsächlich sinnvoll?

Test welche Optimierungen sinnvoll waren

Transposition-Table vs. Selektive Wert Berechnung



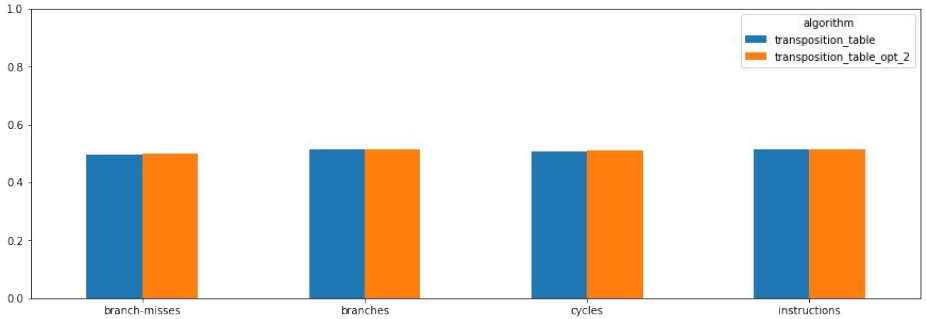
Kleine Verbesserung durch diesen Optimierungsschritt!



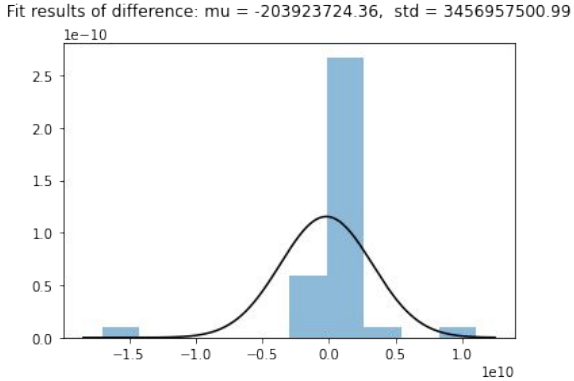
p-Wert = 0.000
=> Selektive Wertberechnung braucht signifikant weniger Zyklen

Test welche Optimierungen sinnvoll waren

Transposition-Table vs. Dereferenzierung vermeiden



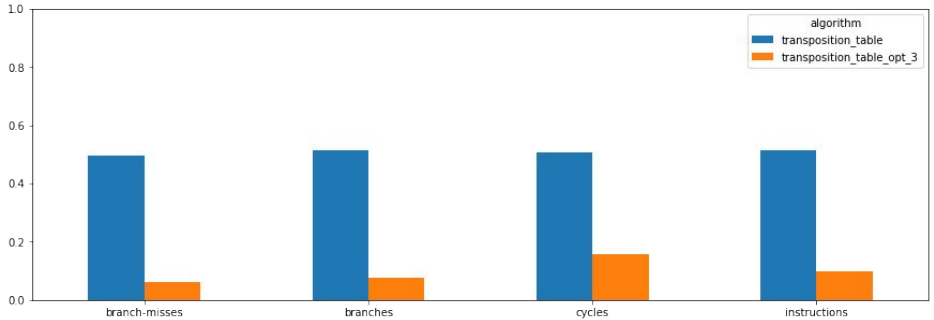
Keine sichtbare Verbesserung durch diesen Schritt



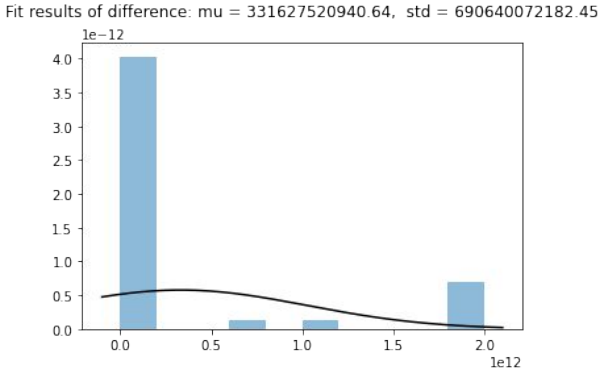
p-Wert = 0.8677
=> Vermeidung von Dereferenzierung braucht nicht signifikant weniger Zyklen

Test welche Optimierungen sinnvoll waren

Transposition-Table vs. Schleifen-Overhead Minimieren



Klare Verbesserung durch diesen Optimierungsschritt!



p-Wert = 0.000
=> Schleifen Overhead Minimierung braucht signifikant weniger Zyklen

Zusammenfassung

Statistisch signifikant wertvolle Optimierungsschritte:

+ Alpha-Beta Pruning
+ Wertberechnung nur berechnen, wenn tatsächlich benötigt
+ Schleifen-Overhead minimieren

Statistisch nicht signifikante Optimierungen:

- Transpositionstabellen -> zu wenige Treffer
- Dereferenzierung vermeiden -> nur 1x im Code

Fragen?