Entwicklung eines Frameworks zur systematischen Evaluierung und Optimierung Minimax-basierter Suchstrategien anhand des Spiels Vier-Gewinnt

Bachelorarbeit

Yassin Liese
410799

22. Februar 2024

Betreuer: Prof. Dr. Benjamin Blankertz
Dr. Stefan Fricke

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Neurotechnologie

Kurzfassung

Seit Jahrzehnten werden Minimax-basierte Suchverfahren genutzt um Computern zu ermöglichen Strategiespiele wie Schach oder Go effizient zu spielen und so die menschliche Leistung zu übertreffen. Im Laufe der Zeit haben sich dabei verschiedene spezielle Algorithmen und Optimierungen herauskristallisiert, die eine effizientere Suche ermöglichen. In dieser Arbeit wird ein *Framework* zur systematischen Evaluierung und Optimierung von Minimax- basierenden Suchstrategien vorgestellt, mit einem speziellen Fokus auf dem Spiel Vier-Gewinnt. Ziel ist es, durch eine analytische Betrachtung und Evaluierung einer praktischen Umsetzung, die Effizienz und Effektivität von verschiedenen Suchalgorithmen und Optimierungen in Spielen mit perfekter Information gegeneinander aufzuwiegen, um so den Algorithmus mit bester Eignung für das Spiel Vier-Gewinnt auszumachen. Die Ergebnisse können so ebenfalls auf andere Spiele und Probleme, mit ähnlichen Eigenschaften, wie Vier-Gewinnt, übertragen werden. Es wird gezeigt, dass sich für die meisten Spielsituationen eine optimierte Variante der NegaC*-Suche, in Kombination mit einer speziellen Heuristik, am besten eignet, in besonders komplexen Positionen kann eine *Principal-Variation*-Suche jedoch überlegen sein.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Vier-Gewinnt ist ein Zwei-Personen Strategiespiel mit perfekter Information. Es wurde 1973 entwickelt und ist seit 1974 lizenziert erwerbbar. Zu Beginn des Spiels erhalten beide Spieler je 21 identische Spielsteine einer Farbe, im Verlauf der Arbeit werden Weiß und Schwarz verwendet, typisch sind jedoch die Farben Gelb und Rot. Das Spiel wird auf einem senkrecht aufgerichteten, hohlen Spielbrett mit 7 Spalten und 6 Zeilen gespielt, sodass Spielsteine, die in eine Spalte von oben eingeworfen werden bis auf die unterste freie Position fallen. Sobald eine Spalte 6 Spielsteine beinhaltet kann sie nicht mehr gespielt werden. Es gibt zwar keine Regel, die besagt, welcher Spieler beginnt, aber aus Gründen der Einfachheit wird nachfolgend angenommen, dass immer der Spieler mit den weißen Spielsteinen das Spiel beginnt. Beide Spieler entscheiden sich nun abwechselnd für je eine Spalte, und werfen einen ihrer Spielsteine in die entsprechende Spalte. Das Spiel endet, sobald einer der Spieler es geschafft hat 4 Spielsteine seiner Farbe horizontal, vertikal oder diagonal in einer Linie anzuordnen, dieser Spieler gewinnt das Spiel. Sollten beide Spieler all ihre Spielsteine aufgebraucht haben ohne, dass einer der beiden Spieler gewonnen hat, endet das Spiel unentschieden.

Obwohl das Spiel Vier-Gewinnt aufgrund seiner einfachen Spielregeln und der begrenzten Größe des Spielbretts auf den ersten Blick einfach zu analysieren scheint, offenbart eine tiefere Betrachtung eine beträchtliche Komplexität. Die Gesamtzahl der validen Spielpositionen beläuft sich auf etwa 4,5 Billionen [4], was die analytische Herausforderung, die bei einer Suche besteht, unterstreicht. Interessanterweise wurde Vier-Gewinnt nahezu simultan, jedoch unabhängig voneinander, von Allis [2] und Allen [1] im Jahr 1988 gelöst. Beide kamen zu dem Ergebnis, dass bei optimalem Spiel der beginnende Spieler (Weiß) gewinnt, sofern er seinen ersten Spielstein in die mittlere Spalte einwirft. Wählt er hingegen eine der beiden direkt angrenzenden Spalten, endet das Spiel in einem Unentschieden, während alle anderen Eröffnungszüge zu einem Sieg für den anderen Spieler (Schwarz) führen.

Trotz der Lösung des Spiels bleibt die Optimierung von Suchalgorithmen für Vier Gewinnt und ähnlich komplexer Spiele eine fortwährende Herausforderung. Dies ist besonders relevant, da sich seit der Lösung des Spiels die algorithmische Landschaft weiterentwickelt hat und neue Suchstrategien und Optimierungsmethoden entstanden sind. Die Übertragbarkeit der Ergebnisse auf andere Spiele und Anwendungsbereiche verstärkt die Bedeutung dieser Forschung. Das Ziel der Arbeit besteht darin, zu untersuchen, welche Algorithmen und Optimierungsansätze sich besonders gut oder schlecht für eine effiziente Suche eignen. Die hierbei entstehenden Erkenntnisse sollen nicht nur für die spezifische Anwendung in Vier Gewinnt, sondern auch für ein breiteres Spektrum anderer Spiele und Probleme von Nutzen sein.

Die Arbeit gliedert sich in mehrere Abschnitte, beginnend mit einer detaillierten Diskussion der verschiedenen Suchalgorithmen und deren theoretischen Grundlagen. Anschließend werden die unterschiedlichen Komponenten des entwickelten *Frameworks* vorgestellt. Kapitel 4 behandelt verschiedene Algorithmische und implementierungsspezifische Optimierungen. In Kapitel 5 werden dann die

vorgestellten Algorithmen und Optimierungen systematisch verglichen und bewertet. Das abschließende Kapitel beinhaltet eine Zusammenfassung der Ergebnisse und einen Ausblick auf zukünftige Arbeiten und weitere Verbesserungen.

2 Suche

2.1 Der Minimax-Algorithmus

Der Minimax-Algorithmus [13] ist ein grundlegendes Konzept der Spieltheorie, zur Berechnung von Lösungspfaden eines Nullsummenspiels, mit perfekter Information. Dies erfolgt durch rekursive Berechnung von Minimax-Werten auf einem Spielbaum. Ein Spielbaum ist ein Entscheidungsbaum, bei dem die Knoten Spielsituationen, und Kanten die Spielzüge darstellen. Jeder Knoten hat also genau so viele Kindknoten, wie es mögliche Züge gibt. In einem Minimax-Spielbaum gibt es MIN- und MAX-Knoten. Bei MIN-Knoten handelt es sich um Spielzustände, in denen MIN am Zug ist, in MAX-Knoten ist MAX am Zug. Da die Spieler abwechselnd am Zug sind, alternieren auch MIN- und MAX-Knoten entlang eines jeden Pfades dieses Spielbaumes. Der Wert eines Knotens kann mit Hilfe einer heuristischen Funktion bestimmt werden, diese bildet eine Spielsituation auf ein Skalar ab. Die beiden Spieler MIN und MAX versuchen, ihrem Namen entsprechend, eine möglichst hohe, bzw. niedrige Bewertung zu erzielen. Positive Werte bedeuten einen Vorteil für MAX und negative Werte entsprechend einen Vorteil für MIN. Geringe Beträge, nahe der Null, deuten auf ein recht ausgeglichene Spielsituation hin, während hohe Beträge deutliche Überlegenheit eines Spielers über den anderen angeben.
Eine solche Heuristik wird benutzt um terminale Zustände zu bewerten. Ein terminaler Zustand ist ein solcher, der das Ende eines Spieles markiert, er besitzt also keine Kindknoten und ist damit selbst ein Blattknoten. Nicht terminale Zustände erhalten ihre Bewertung durch Propagieren der Werte ihrer Kindknoten. Der Minimax-Wert der Wurzel stellt die Lösung des Spielbaumes dar. Es ist der Wert des terminalen Zustands, der erreicht wird, wenn beide Spieler immer bestmögliche Züge durchführen. Dementsprechend erhalten alle Knoten auf dem Pfad zwischen diesem terminalen Knoten und der Wurzel den selben Wert. Ein solcher Pfad wird auch *Principal-Variation* [11] genannt. Ein Beispiel für einen solchen Spielbaum ist in Abb. 2.1 zu sehen. Hier stellt der Minimax-Wert 7 die Lösung des Spielbaumes dar. Die *Principal-Variation* ist rot hervorgehoben.

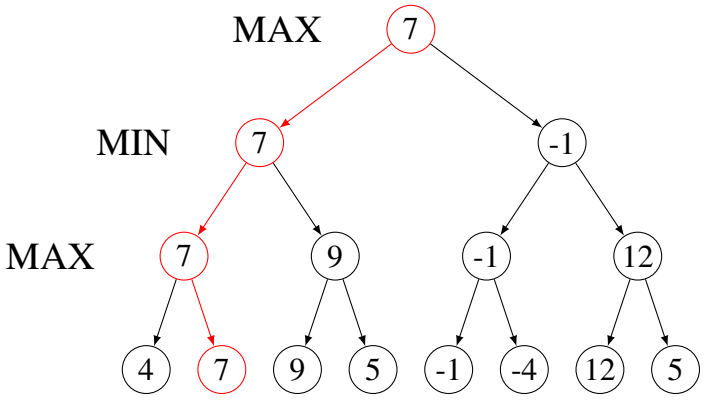

Abbildung 2.1: Beispiel für einen Minimax Spielbaum

Der Minimax-Algorithmus versucht diese Bewertungen nun so zur Wurzel des Spielbaumes zu propagieren, dass diese den Wert des Blattknotens erhält, der erreicht wird, wenn beide Spieler optimal spielen. Bei der klassischen auf Tiefensuche basierenden Implementierung erfolgt dies, indem für jeden Knoten zuerst die Minimax-Werte aller möglichen Kindknoten berechnet werden. Handelt es sich bei dem aktuellen Knoten um einen MAX-Knoten, erhält er den Minimax-Wert des Kindknotens mit der maximalen Bewertung. Handelt es sich um einen MIN-Knoten erhält er den Minimax-Wert des Kindknotens mit der minimalen Bewertung. Der Algorithmus besteht aus zwei Komponenten: eine die vom minimierenden Spieler aufgerufen wird und eine, die vom maximierenden Spieler aufgerufen wird. Je nachdem, ob es sich bei dem aktuell untersuchten Knoten um einen MIN- oder MAX-Knoten handelt, erhält der untersuchte Knoten den Wert des Kindknotens mit minimalem bzw. maximalem Wert. Dies lässt sich zu einer einzelnen Komponente zusammenfassen. Dafür transformiert man das Minimierungsproblem des minimierenden Spielers in ein Maximierungsproblem. Hierzu muss lediglich die Bewertung für den minimierenden Spieler negiert werden, sodass nun beide Spieler individuell ein Maximierungsproblem lösen müssen. Diese Variante ist allgemein als Negamax bekannt und ist in Algorithmus 1 konzeptuell dargestellt.

Algorithmus 1 Negamax-Implementierung des Minimax-Algorithmus

```
1: function negamax(node)
2:     if isTerminal(node) then
3:         return evaluate(node)
4:     value ← −∞
5:     for all child ∈ getChildren(node) do
6:         value ← max(value, −negamax(child))
7:     return value
```

2.2 Alpha-Beta-Suche

Die Alpha-Beta-Suche ist die wohl bekannteste Optimierung des Minimax-Suchverfahrens. Die Idee hierfür besteht schon seit 1956, als McCarthy darauf hinwies, dass zur Berechnung des Minimax-Wertes einige Teile des Suchbaumes außer Acht gelassen werden können [6], um so die Anzahl der zu untersuchenden Zustände und somit ebenfalls die Laufzeit zu verringern. Dazu wird die Minimax-Suche um die beiden Parameter α und β erweitert. α stellt den derzeit besten bekannten Minimax-Wert für MAX dar und β umgekehrt den besten bekannten Minimax-Wert für MIN. Sie stellen untere und obere Schranke für die Bewertung dar und formen so das Suchfenster:

$$\alpha \leq \text{tatsächlicher Minimax-Wert} \leq \beta.$$

Initial werden α und β üblicherweise auf den für den jeweiligen Spieler schlechtesten möglichen Wert gesetzt, in der Regel mit $\alpha = -\infty$ und $\beta = \infty$. Sollte im Verlauf der Suche ein besserer Wert für einen Spieler gefunden werden, muss α bzw. β entsprechend aktualisiert werden. Sollte einer der beiden Spieler nun einen besseren Minimax-Wert außerhalb des Suchfensters finden bedeutet das, dass sich der Gegenspieler schon im vorherigen Zug nicht für den aktuellen Zweig entscheiden würde. Daher ist der entsprechende Zustand bei perfektem Spiel beider Seiten unerreichbar und all seine Geschwisterknoten brauchen nicht weiter untersucht zu werden, es kommt zu einem *Cutoff*. Um nicht

zwei verschiedene Algorithmen für den minimierenden und maximierenden Spieler implementieren zu müssen, kann analog zur Minimax-Suche auch hier eine Negamax-Implementierung genutzt werden. Zusätzlich zum Negieren der Bewertung, müssen hier auch α und β korrekt angepasst werden. Dazu werden in jeder Schicht α und β vertauscht und negiert. Eine solche Implementierung ist in Algorithmus 2 aufgeführt.

Algorithmus 2 Negamax-Implementierung der Alpha-Beta-Suche

```
1:  function alphaBeta(node, α, β)
2:      if isTerminal(node) then
3:          return evaluate(node)
4:      value ← −∞
5:      for all child ∈ getChildren(node) do
6:          score ← −alphaBeta(child, −β, −α)
7:          if score ≥ β then
8:              return score
9:          if score > value then
10:             value ← score
11:         if score > α then
12:             α ← score
13:     return value
```

Abb. 2.2 zeigt welche Teile des Suchbaumes aus Abb. 2.1 durch Verwendung von Alpha-Beta-Suche abgeschnitten werden können. Nachdem der vierte Blattknoten mit dem Wert 9 untersucht wurde nützt es dem maximierenden Spieler nicht noch weitere Geschwister dieses Blattknotens zu erkunden, da der minimierende Spieler sich im Knoten darüber sowieso für den linken Pfad entscheiden wird. Ein besserer Knoten wäre also ohnehin irrelevant. Dasselbe gilt ebenfalls für den rechten Zweig des rechten Teilbaumes. Der minimierende Spieler braucht nicht weiter nach Verbesserungen suchen, da sich der maximierende Spieler in der Wurzel für den Pfad im linken Teilbaum entscheiden wird.

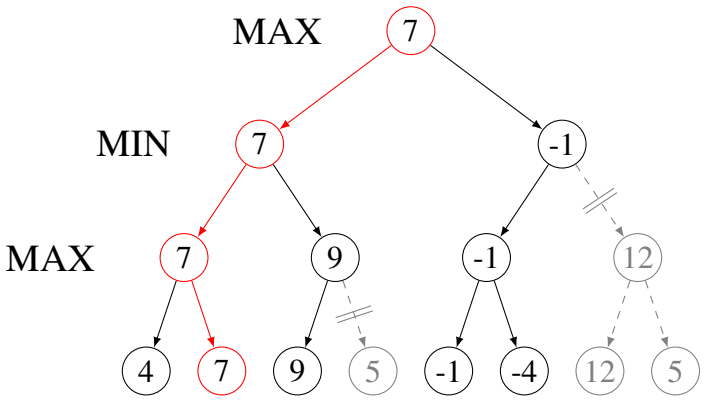

Abbildung 2.2: Beispiel für einen Minimax Spielbaum

2.2.1 *Fail-Soft* und *Fail-Hard*

Bei der Alpha-Beta-Suche unterscheidet man im Wesentlichen zwischen zwei Varianten: *Fail-Soft* und *Fail-Hard*. Der Unterschied zwischen diesen Varianten macht sich bemerkbar, wenn das Ergebnis der Suche nicht innerhalb des Suchfensters liegt. Eine *Fail-Hard*-Implementierung gibt hier direkt α oder β zurück, je nachdem ob das Ergebnis ober- oder unterhalb des Suchfensters liegt. Im Gegensatz dazu kann eine *Fail-Soft*-Implementierung auch Werte zurückgeben die außerhalb des Suchfensters liegen. Der klassische Ansatz ist dabei eine Art Hybrid-Variante. Der in Algorithmus 2 dargestellte Algorithmus gibt den tatsächlichen Minimax-Wert zurück, wenn er oberhalb des Suchfensters liegt, jedoch ist der minimale mögliche Rückgabewert α. Damit werden also Minimax- Werte, die unterhalb von α liegen beschnitten. Um aus dieser Variante nun eine reine *Fail-Soft*-Implementierung zu machen, wird eine zusätzliche Variable $value$ eingeführt. Anfangs wird diese auf einen kleinstmöglichen Wert gesetzt, damit dieser garantiert kleiner ist als die Minimax-Werte der Kindknoten. Nach jedem rekursiven Aufruf wird überprüft, ob die errechnete Bewertung besser ist, als die bisher beste bekannte. Ist dies der Fall erhält die Variable eben diesen Wert. Sollte es nicht zu einem *Cutoff* kommen wird diese Variable am Ende zurückgegeben. Sollte keiner der möglichen Züge zu einem Zustand geführt haben, dessen Bewertung mindestens so gut ist wie α, gilt $value < \alpha$. Somit wird nun, auch für Werte die unterhalb des Suchfensters liegen, der tatsächliche Minimax-Wert des Knotens zurückgegeben. [5] Bei allen Varianten gilt, dass Werte $\leq \alpha$ bzw $\geq \beta$ keine präzisen Werte darstellen sondern lediglich eine obere bzw. untere Schranke. Der Vorteil der Hybridvariante gegenüber der *Fail-Hard*-Variante besteht darin, dass kein zusätzlicher Rechenaufwand nötig ist, die Funktion aber schärfere Schranken an einem Ende des Suchfensters zurückgibt. Eine *Fail-Soft*-Implementierung tut dies sogar an beiden Enden des Suchfensters, allerdings erfordert sie eine weitere Variable. Daher ist hier ein Abwägen zwischen zusätzlicher Laufzeit und Ersparnis durch noch präzisere Ergebnisse notwendig. In Algorithmus 3 sind die unterschiedlichen Implementierungen systematisch dargestellt. Die gefärbten Zeilen sind nur in der Implementierung mit gleicher Farbe enthalten.

Algorithmus 3 Vergleich zwischen *Fail-Soft*, *Fail-Hard* und der Hybridvariante

```
1:  function alphaBeta(node, α, β)
2:      if isTerminal(node) then
3:          return evaluate(node)
4:      value ← −∞
5:      for all child ∈ getChildren(node) do
6:          score ← −alphaBeta(child, −β, −α)
7:          if score ≥ β then
8:              return score
9:              return beta
10:             return score
11:         if score > value then
12:             value ← score
13:         if score > α then
14:             α ← score
15:     return value
16:     return alpha
17:     return alpha
```

Da alle Algorithmen aber trotzdem den selben Suchbaum mit denselben Knoten explorieren, scheint der Vorteil der *Fail-Soft*-Variante erst einmal nicht offensichtlich. Ein Nutzen könnte die Verwendung des Rückgabewertes als eine Art Heuristik sein. Zwei Verfahren, die diesen Ansatz verfolgen werden in Abschnitt 2.5 und Abschnitt 2.6 näher erläutert. Eine weitere Anwendung wird in Abschnitt 2.3.2 genannt.

2.2.2 Nullfenstersuche

Durch Variation des Suchfensters kann die Alpha-Beta-Suche auch auf spezielle Anwendungsfälle zugeschnitten werden. Wählt man beispielsweise ein Suchfenster mit $\alpha = -1$ und $\beta = 1$ wird die Alpha-Beta-Suche zwar nicht den bestmöglichen Minimax-Wert finden, sondern nur bestimmen, ob der tatsächliche Minimax-Wert größer, kleiner oder gleich null ist, allerdings führt das kleinere Suchfenster zu mehr *Cutoffs* und somit zu einer besseren Laufzeit. Ein solches Suchfenster wird von sogenannten schwachen *Solvern* benutzt. Es ist zu klein um die exakte Lösung zu bestimmen, genügt aber um den Ausgang des Spieles festzustellen: Das Ergebnis $w \leq -1$ stellt einen Sieg für MIN, $w \geq 1$ ein Sieg für MAX und $w = 0$ ein Unentschieden dar. Die extremste Form dieser Idee ist Die Nullfenstersuche. Hierfür wird ein minimal großes Suchfenster gewählt, sodass $\beta = \alpha + 1$ gilt. Das Ergebnis dieser Suche stellt dann nur noch eine obere oder untere Schranke für den tatsächlichen Minimax-Wert dar, produziert aber maximal viele *Cutoffs*. Führt man zum Beispiel eine Failhard-Nullfenstersuche um den Wert w durch mit $\alpha = w$ und $\beta = w + 1$ bedeutet das Ergebnis w, dass der tatsächliche Minimax-Wert kleiner gleich w ist, während $w + 1$ bedeutet, dass der tatsächliche Minimax-Wert größer als w ist.

2.3 Optimierungen der Alpha-Beta-Suche

In diesem Abschnitt werden fortgeschrittene Methoden zur Effizienzsteigerung der Alpha-Beta-Suche für die Anwendung an Spielen diskutiert. Die Optimierungen zielen darauf ab, die Anzahl der zu untersuchenden Zustände zu reduzieren, ohne dabei das Ergebnis zu verändern. Zwei zentrale Ansätze stehen dabei im Vordergrund: Die Zugsortierung zur Maximierung der Anzahl an *Cutoffs* und die Nutzung einer *Transposition-Table* zur Wiederverwendung bereits berechneter Zustandsbewertungen.

2.3.1 Zugsortierung

Durch Anpassung der Reihenfolge, in der die Züge untersucht werden, kann viel Zeit eingespart werden. Die Grundidee hierfür basiert auf der Beobachtung, dass die Alpha-Beta-Suche die meisten *Cutoffs* erzielt, wenn die *Principal-Variation*, also der optimale Pfad durch den Spielbaum, schon früh gefunden wird. Hierdurch kann ein Großteil des Baumes ignoriert werden, da nicht erfolgversprechende Züge durch *Cutoffs* ausgeschlossen werden. Erfolgversprechende Züge sollten also möglichst früh untersucht werden. Diese Funktionsweise kann ebenfalls an dem in Abb. 2.2 aufgeführten Minimax-Baum veranschaulicht werden. Strukturiert man den Baum so um, dass beste Züge immer zuletzt untersucht werden, gibt es keine *Cutoffs* mehr, da die Suche unnötigerweise jeden Zustand betrachten muss. Im Kontrast dazu führt die umgekehrte Sortierung, bei der erfolgversprechende Züge zuerst betrachtet werden, zu besonders vielen *Cutoffs* und somit zu einer Reduzierung der Anzahl untersuchter Zustände. Die Schwierigkeit besteht darin, gute Züge frühzeitig zu erkennen und diese

entsprechend zu priorisieren. In komplexen Spielen kann dies eine Herausforderung darstellen, da die Bewertung der Züge nicht trivial ist und von vielen Faktoren abhängt. Um komplexe Verfahren, die aufwändige zusätzliche Berechnungen benötigen, zu vermeiden, bietet es sich im Fall von Vier-Gewinnt an, die Züge nach Abstand zur mittleren Spalte zu sortieren. Hierbei wird die Annahme getroffen, dass zentrale Spielsteine grundsätzlich einen höheren strategischen Wert besitzen als jene, die in den äußeren Spalten liegen. Diese Annahme basiert auf der Tatsache, dass zentrale Steine an mehr möglichen vierer-Reihen beteiligt sind als Randsteine. Dies lässt sich effizient umsetzen, da die Sortierung lediglich statisch erfolgt und nicht in jedem Zustand neu berechnet werden muss.

2.3.2 *Transposition-Table*

Die *Transposition-Table* wird genutzt, um Zwischenergebnisse zu speichern. Auf diese Weise müssen bereits untersuchte Zustände nicht noch einmal untersucht werden. Dies wird realisiert, indem für jeden untersuchten Zustand ein Eintrag in der Tabelle angelegt wird, der folgende Informationen beinhaltet: Eine Bewertung, die von einer vorherigen Suche berechnet wurde, die dazugehörige Suchtiefe, mit der der Zustand bereits untersucht wurde, und ein Flag, das Auskunft darüber gibt, ob es sich bei dem Wert um eine obere Schranke, eine untere Schranke oder den exakten Minimax-Wert handelt. Zu Beginn der Suche wird überprüft, ob der aktuell zu untersuchende Knoten bereits in der *Transposition-Table* existiert. Ist dies nicht der Fall, wird die Suche wie gewohnt fortgesetzt. Existiert jedoch ein Eintrag, und ist dieser als exakter Minimax-Wert markiert, kann der Wert direkt aus der *Transposition-Table* zurückgegeben werden. Ist der Eintrag eine obere oder untere Schranke, so können die Werte von α und β entsprechend angepasst werden, um das Suchfenster zu verkleinern. Bevor die Suche ihr Ergebnis zurückgibt, muss dieses in der *Transposition-Table* gespeichert werden. Ist der Rückgabewert größer oder gleich β, wird er als untere Grenze gespeichert. Ist er kleiner oder gleich dem ursprünglichen α, wird er als obere Schranke gespeichert. Andernfalls wird der Eintrag als exakt markiert. Durch Verwenden einer *Fail-Soft*-Implementierung der Alpha-Beta-Suche, können obere Schranken noch niedriger und untere Schranken noch höher ermittelt und gespeichert werden. Werden diese Schranken später wiederverwendet, könnten sie zu zusätzlichen *Cutoffs* führen. Auf einer kleinen Menge von Dame-Positionen konnten so knapp 7% weniger Zustände untersucht werden [5].

2.4 *Principal-Variation*-Suche

1979 konnte durch George C. Stockman gezeigt werden, dass mehrere sinnvoll eingesetzte Nullfenstersuchen einen Spielbaum effizienter absuchen als eine konventionelle Alpha-Beta-Suche es tun würde [14]. Das Hauptargument dafür lautet, dass hierbei kein Zustand evaluiert wird, der nicht auch von der Alpha-Beta-Suche evaluiert wird. Stattdessen führen mehr *Cutoffs* dazu, dass sogar beachtlich weniger Zustände abgesucht werden.

Die *Principal-Variation*-Suche [8], kurz PVS, ist eine Variante, die sich dieser Idee bedient. Ihren Namen erhält sie von ihrer Funktionsweise. Sie versucht nämlich die *Principal-Variation* möglichst schnell zu finden und andere Knoten effizient durch Nutzung von Nullfenstersuchen auszuschließen. Die Idee der *Principal-Variation*-Suche ist es, davon auszugehen, dass der erste Kindknoten in jedem Zustand zur *Principal-Variation* gehört. Der Wert für ihn wird mit einer vollständigen Suche für den rekursiven Aufruf ermittelt. Im nächsten Schritt versucht sie zu beweisen, dass der gefundene Wert

tatsächlich Teil der *Principal-Variation* ist, dementsprechend also besser ist, als der aller anderen Geschwisterknoten. Dafür werden für diese Knoten Nullfenstersuchen um α, also mit dem Suchfenster $[\alpha, \beta]$ mit $\beta = \alpha + 1$, durchgeführt, um eine untere Schranke zu erhalten. Kann einer der Geschwisterknoten eine bessere Bewertung erhalten, muss das Resultat der Nullfenstersuche mindestens $\alpha + 1$ betragen. Seine untere Schranke ist damit größer, als die des ersten Kindknotens, und es muss eine vollständige Suche für ihn durchgeführt werden, um den exakten Wert zu ermitteln. Anschließend wird analog versucht zu beweisen, dass dieser Knoten Teil der *Principal-Variation* ist. Die Leistung der *Principal-Variation*-Suche hängt also stark von der Zugsortierung ab, da eine gute Zugsortierung die *Principal-Variation* früh findet, viele erfolgreiche Nullfenstersuchen durchführt und viele *Cutoffs* erzielt. Eine schlechte Zugsortierung hingegen, kann im schlimmsten Fall dafür sorgen, dass ein Großteil aller Knoten mehrfach evaluiert werden muss, einmal mit einer Nullfenstersuche und einmal mit einer vollständigen. Algorithmus 4 zeigt eine Beispielhafte *Fail-Soft*-Implementierung der *Principal-Variation*-Suche.

Algorithmus 4 Negamax-Implementierung der *Principal-Variation*-Suche (*Fail-Soft*)

```
1:  function pvs(node, α, β)
2:      if isTerminal(node) then
3:          return evaluate(node)
4:      value ← −∞
5:      isFirst ← true
6:      for all child ∈ getChildren(node) do
7:          if isFirst then
8:              score ← −pvs(child, −β, −α)
9:          else
10:             score ← −pvs(child, −α − 1, −α)
11:             if score > α then
12:                 score ← −pvs(child, −β, −α)
13:         if score ≥ β then
14:             return score
15:         isFirst ← false
16:         if score > value then
17:             value ← score
18:         if score > α then
19:             α ← score
20:     return value
```

2.5 MTD(f)

MTD(f) ist ebenfalls ein auf der Alpha-Beta-Suche basierender Algorithmus, der mit Nullfenstersuchen arbeitet [10]. Er startet mit einer anfänglichen Schätzung f, die initial übergeben wird, und ermittelt von dort aus den tatsächlichen Minimax-Wert iterativ. Dazu werden die Grenzen des Suchfensters, *min* und *max*, so festgelegt, dass sich der tatsächliche Minimax-Wert garantiert darin befindet. Es wird eine neue Variable g verwendet, die Anfangs den Wert von f zugewiesen bekommt. MTD(f) führt nun wiederholt Nullfenstersuchen um den Wert g durch, wobei das Resultat jeweils

wieder in g gespeichert wird. Je nachdem, ob das Resultat nun eine obere oder untere Schranke darstellt, wird das Suchfenster entsprechend angepasst. Der Algorithmus terminiert, wenn $min < max$ nicht mehr gilt. Obere und untere Schranke nehmen also denselben Wert an. Dieser stellt dann das Ergebnis der Suche dar und wird zurückgegeben. Algorithmus 5 zeigt eine typische Implementierung.

Algorithmus 5 MTD(f)-Implementierung [10]

```
1:  function mtdf(node, f)
2:      g ← f
3:      min ← −∞
4:      max ← ∞
5:      repeat
6:          if g = min then
7:              β ← g + 1
8:          else
9:              β ← g
10:         g ← alphabetaWithMemory(node, β − 1, β)
11:         if g < β then
12:             max ← g
13:         else
14:             min ← g
15:     until min >= max
16:     return g
```

Es wird also immer eine Nullfenstersuche um das Resultat der letzten Iteration durchgeführt. Würde hierfür eine *Fail-Hard*-Implementierung der Alpha-Beta-Suche genutzt werden, müsste der Algorithmus eine Nullfenstersuche für jeden Wert zwischen der initialen Schätzung und dem letzendlichen Resultat durchführen. Eine *Fail-Soft*-Variante hingegen kann den tatsächlichen Wert, mit Hilfe von schärferen Schranken und damit größeren Schritten, mit deutlich weniger Iterationen finden. Wird der Algorithmus mit dem tatsächlichen Minimax-Wert aufgerufen, so benötigt er exakt zwei Iterationen um zu terminieren. Eine um die obere Schranke zu finden und eine für die Untere. *MTD(f)* benutzt zwar ausschließlich Nullfenstersuchen, allerdings wird die Alpha-Beta-Suche wiederholt auf dem selben Knoten aufgerufen. Deshalb ist hierfür die Verwendung einer *Transposition-Table* besonders wichtig.

2.6 NegaC*

Die *NegaC*-Suche* ist die Negamax-Variante der 1982 durch Coplan vorgestellten *C*-Suche* [15]. Ähnlich wie *MTD(f)* führt auch die *NegaC*-Suche* mehrere Nullfenstersuchen durch um den tatsächlichen Minimax-Wert zu bestimmen. Allerdings wird hierfür keine Nullfenstersuche um das Resultat der vorherigen Iteration durchgeführt. Stattdessen wird das Suchfester bisektionsartig stetig verkleinert, um den tatsächlichen Wert zu finden. Es wird also stetig der Mittelwert des Suchfensters gebildet, und eine Nullfenstersuche wird genutzt um festzustellen, ob der tatsächliche Minimax-Wert größer oder kleiner gleich dem Mittelwert ist. Der ermittelte Minimax-Wert wird dann entsprechend zur oberen bzw. unteren Schranke des Suchfensters. Würde man für die Nullfenstersuche nun eine

Fail-Hard-Implementierung verwenden, gleiche das Verfahren einer Art binären Suche. Eine *Fail-Soft*-Implementierung hingegen, liefert noch schärfere Schranken und kann das Suchfenster in jedem Schritt somit stärker einschränken. Eine Mögliche Implementierung ist in Algorithmus 6 aufgeführt.

Algorithmus 6 NegaC*-Implementierung [15]

```
1:  function negaCStar(node, f)
2:      min ← −∞
3:      max ← ∞
4:      while min < max do
5:          med ← min + (max − min)/2
6:          r ← alphabetaWithMemory(node, med, med + 1)
7:          if r ≤ med then
8:              max ← r
9:          else
10:             min ← r
11:     end while
12:     return min
```

Wichtig ist hier zu beachten, dass der Mittelwert des Suchfensters med nicht direkt durch $med = (min + max)/2$ berechnet werden sollte, da dies in Programmiersprachen wie C, angenommen min und max sind Integer, zu einer Endlosschleife führen könnte. Betrachtet man zum Beispiel den Fall $min = -1$ und $max = 0$ mit einem tatsächlichen Minimax-Wert von 0, würde der Algorithmus in einer Endlosschleife hängen bleiben. Da für den Aufruf der Alpha-Beta-Suche immer das Suchintervall $[med, med + 1]$ verwendet wird, sollte med immer kleiner sein als max. Die Berechnung mittels $med = min + (max - min)/2$ stellt dies durch effektive Berechnung von $\lfloor \frac{min+max}{2} \rfloor$ sicher.

3 Framework

In diesem Kapitel sollen einige wichtige Kernkomponenten eingeführt werden, die zur effizienten Simulation eines Vier-Gewinnt-Spiels notwendig sind. Dazu zählen im Wesentlichen eine Zustandsrepräsentation, die Möglichkeit Züge durchzuführen und das Erkennen des Spielendes. Im Anschluss daran werden weitere Komponenten erläutert, die notwendig sind um die in Kapitel 2 eingeführten Suchalgorithmen zu implementieren.

3.1 Zustandsrepräsentation

Die wohl grundlegendste Funktion eines solchen Frameworks besteht darin einen Spielzustand effizient und kompakt darstellen zu können. Im Nachfolgenden wird erklärt wie Bitboards genutzt werden können, um dies, im Fall von Vier-Gewinnt, zu erreichen. Die Beschriebenen Pinzipien stammen dabei ursprünglich aus dem Fhourstones-Benchmark[1] und sind an einer anderen Stelle in einem Github-Repository[2] beschrieben.

3.1.1 Bitboards

Ein Bitboard ist eine Datenstruktur, die seit ca. 70 Jahren genutzt wird, um Spielbretter mit bis zu 64 Feldern als Menge von Bits zu kodieren. Der Vorteil besteht darin, dass so nicht mehr ein *Integer* pro Spielfeld gespeichert werden muss, sondern nur ein einzelnes Bit Auskunft darüber gibt, ob ein bestimmter Spielstein auf dem gegebenen Feld liegt oder nicht [3].

In Vier-Gewinnt auf dem standardmäßigen $7 \cdot 6$ Spielbrett mit 42 Feldern, kann jedes Feld einen von drei Zuständen annehmen. Entweder es ist frei, oder von einem weißem bzw. schwarzen Spielstein okkupiert. Demnach sind mindestens zwei Bitboards mit je mindestens 42 Bits notwendig, um einen einzelnen Zustand darzustellen. Ein einzelnes Bitboard kann somit effizient als ein 64-Bit *unsigned Integer* gespeichert werden. Die Zellen des Spielfeldes werden den Stellen des Bitboards dabei wie folgt zugeordnet. Das Feld unten links wird dem Bit 0, also dem am wenigsten signifikanten, zugeordnet. Fas Feld darüber dem Bit 1, usw. bis zum oberen linken Feld, dieses wird dem Bit 5 zugeordnet. Das unterste Feld der zweiten Spalte wird Bit 7 zugeordnet, Bit 6 wurde also übersprungen, da oberhalb des Spielfeldes eine Reihe frei gehalten werden muss. Auf den Grund hierfür wird nachfolgend im Abschnitt *Prüfen auf Spielende* eingegangen. Das Feld darüber wird Bit 8 zugeordnet, bis hin zum oberen rechten Feld, dieses wird Bit 47 zugeordnet. Insgesamt werden also 49 Bits benötigt, um ein einziges Bitboard zu speichern. Die Zuordnung ist in Abb. 3.1 veranschaulicht.

[1] https://tromp.github.io/c4/fhour.html
[2] https://github.com/denkspuren/BitboardC4/blob/master/BitboardDesign.md

6	13	20	27	34	41	48
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	7	14	21	28	35	42

Abbildung 3.1: Zuordnung der Spielfelder zu den Bits des Bitboards

Im folgenden Beispiel ist Weiß am Zug, beide Spieler haben genau 10 Zuge gespielt.

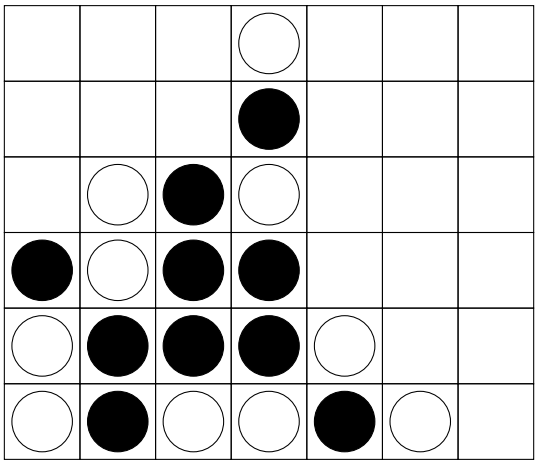

Abbildung 3.2: Beispiel 1: Weiß ist am Zug

Dieses Spiel kann nun mit Hilfe von zwei Bitboards dargestellt werden: Ein Bitboard wird verwendet um das gesamte Spielbrett zu repräsentiert, nachfolgend $full$ genannt, sodass ein gesetztes Bit bedeutet, dass das jeweilige Feld von einem Spielstein beliebiger Farbe besetzt ist. Im konkreten Beispiel würde dies folgendermaßen aussehen:

0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	0	1	0	0	0
0	1	1	1	0	0	0
1	1	1	1	0	0	0
1	1	1	1	1	0	0
1	1	1	1	1	1	0

Abbildung 3.3: Beispiel 1: $full$-Bitboard

Das Bitboard wird also durch den folgenden Bitstring repräsentiert:

0000000 0000001 0000011 0111111 0001111 0001111 0000111

Das zweite Bitboard wird zur Zuordnung der belegten Felder zu den jeweiligen Spielern genutzt und nachfolgend *current* genannt. Dabei sind nur diejenigen Bits gesetzt, die zu Spielfeldern gehören, die von Spielsteinen einer Farbe besetzt sind. Die entsprechenden Bitboards der jeweiligen Spieler sehen dabei folgendermaßen aus:

0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0
0	1	0	1	0	0	0
0	1	0	0	0	0	0
1	0	0	0	1	0	0
1	0	1	1	0	1	0

(a) Beispiel 1: Bitboard für Weiß

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
1	0	1	1	0	0	0
0	1	1	1	0	0	0
0	1	0	0	1	0	0

(b) Beispiel 1: Bitboard für Schwarz

Abbildung 3.4: Beispiel 1: Bitboards beider Spieler

Hier wird schnell klar, dass beim Speichern aller drei Bitboards eine Redundanz bestehen würde, da jedes Bitboard mit Hilfe des exklusiven Oders aus den zwei anderen berechnet werden kann. Die effizienteste Methode ist hier nur eines der farbenspezifischen Bitboards zu speichern, da andernfalls das Ermitteln des Spielers der aktuell am Zug ist weitere Kosten verursacht. Ebenso müssten beim Ausführen von Zügen beide Bitboards betrachtet werden, um die korrekte Position des neuen Spielsteins zu bestimmen. Es ist zwar nicht zwingend notwendig, aber zur späteren Verwendung wird ebenfalls die Anzahl bereits durchgeführter Züge gespeichert.

3.1.2 Züge durchführen

Züge können ebenfalls durch Bitboards dargestellt werden. Dabei ist jedoch nur ein einziges Bit gesetzt, dieses entspricht dem Spielfeld, auf dem der Spielstein landen soll. Ein Zug kann dann folgenderweise angewandt werden:

1. $current$ wird mit $full$ ver-exklusiv-odert
2. $full$ wird mit dem Zug verodert
3. der Zugzähler wird inkrementiert

Dementsprechend ist $current$ immer das Bitboard des Spielers, der als nächstes am Zug ist.

Um nun einen gültigen Zug zu machen muss der Spieler der aktuell am Zug ist zunächst eine der sieben Spalten wählen, die noch nicht voll ist. Ob eine Spalte voll ist kann mit Hilfe von $full$ geprüft werden. Dazu muss lediglich das Bit der obersten Zeile der jeweiligen Spalte ausgelesen werden. Hat der Spieler eine gültige Spalte ausgewählt, muss nun das entsprechende Bitboard für den Zug berechnet werden. Dazu wird zunächst eine Bitmaske generiert, bei der lediglich das unterste Bit der gewählten Zeile gesetzt ist. Diese Maske wird dann mit $full$ summiert. Durch die Strukturierung der Felder und Bits, wie oben beschrieben, entsteht dadurch ein gesetztes Bit an der gewünschten Position und alle anderen Bits der betroffenen Spalte werden zu 0. Nun kann mittels einer weiteren Bitmaske, über die gesamte Spalte, durch Verundung jegliche Informationen über das restliche Spielfeld verworfen werden und man erhält den gewünschten Zug.

Dieses Vorgehen ist im Folgenden anhand eines Beispiels veranschaulicht. Betrachtet man das obige Beispiel, in Abb. 3.2, fällt schnell auf, dass Weiß eigentlich nur einen einzigen sinnvollen Zug hat, da Schwarz sonst sofort, durch Spielen von Spalte drei, gewinnen kann. Weiß entscheidet sich also selber Spalte drei zu spielen.

$$
\begin{pmatrix}
0&0&0&0&0&0&0 \\
0&0&0&1&0&0&0 \\
0&0&0&1&0&0&0 \\
0&1&1&1&0&0&0 \\
1&1&1&1&0&0&0 \\
1&1&1&1&1&0&0 \\
1&1&1&1&1&1&0
\end{pmatrix}
+
\begin{pmatrix}
0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0 \\
0&0&1&0&0&0&0
\end{pmatrix}
\;\&\;
\begin{pmatrix}
0&0&0&0&0&0&0 \\
0&0&1&0&0&0&0 \\
0&0&1&0&0&0&0 \\
0&0&1&0&0&0&0 \\
0&0&1&0&0&0&0 \\
0&0&1&0&0&0&0 \\
0&0&1&0&0&0&0
\end{pmatrix}
=
\begin{pmatrix}
0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0 \\
0&0&1&0&0&0&0 \\
0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0 \\
0&0&0&0&0&0&0
\end{pmatrix}
$$

Abbildung 3.5: Beispiel 1: Berechnung eines Zuges

Abb. 3.5 zeigt wie der zugehörige Zug berechnet wird. Zuerst werden $full$ und die Maske mit nur dem untersten Bit der dritten Spalte summiert, anschließend wird jegliche Information über andere Spalten durch Verundung mit der Spalte-drei-Maske verworfen. Die hierfür notwendigen Bitmasken lassen sich schon vorab, für alle Spalten, erzeugen. Das Berechnen und Ausführen von Zügen ist somit effizient umsetzbar.

3.1.3 Prüfen auf Spielende

Die letzte wichtige, noch fehlende Funktion ist die zur Feststellung des Spielendes. Dafür gibt es im Wesentlichen drei Fälle:

1. Alle Steine wurden gespielt, das Spiel endet in einem Unentschieden.
2. Weiß gewinnt.
3. Schwarz gewinnt.

Der erste Fall kann recht schnell auf zwei verschiedene Arten geprüft werden:

1. Der Zugzähler hat den selben Wert, wie es insgesamt Spielsteine gibt.
2. Alle Bits von $full$, die zu Spielfeldern zählen, sind gesetzt.

Da ohnehin nach jedem Zug auf Spielende geprüft wird, genügt es lediglich die Steine des letzten Spielers zu betrachten. Es bedarf also einer Funktion, die, gegeben ein Bitboard einer Farbe, wahr zurückgibt, wenn auf dem Bitboard mindestens vier Bits horizontal, vertikal oder diagonal in einer Reihe gesetzt sind, und falsch sonst. Eine solche Funktion ist in Algorithmus 7 exemplarisch für die horizontale Richtung beschrieben [3]. Als Eingabe erhält er das zu überprüfende farbenspezifische Bitboard b. Er gibt dann wahr oder falsch aus, abhängig davon, ob in dem übergebenen Bitboard vier konsekutive Bits in horizontaler Richtung gesetzt sind.

Algorithmus 7 vertikale Überprüfung

```
1: function hasFourVertical(b)
2:     x ← b & (b >> 1)
3:     if x & (x >> 2) ≠ 0 then
4:         return true
5:     return false
```

Durch die oben beschriebene Bit-Zuordnung, liegen alle Spielfelder einer Spalte innerhalb des Bitboards ebenfalls unmittelbar nebeneinander. Sollten auf dem Spielbrett also vier Steine gleicher Farbe übereinander liegen, so treten auch in dem dazugehörigen Bitboard vier aufeinander folgende gesetzte Bits auf. Der Algorithmus startet nun damit das originale Bitboard vorerst mit dem um eine Stelle nach rechts verschobenen Bitboard zu verunden. In dem resultierenden Bitboard sollten dann mindestens drei Bits gesetzt werden. Das erste, am wenigsten signifikante, ist die Verundung aus dem ersten und zweiten Bit. Das zweite ist die Verundung aus dem zweiten und dritten, und das dritte die Verundung aus dem dritten und vierten Bit. Dieses resultierende Bitboard wird nun noch einmal mit dem um zwei Stellen nach rechts verschobene Bitboard verundet. Dadurch bleibt ein einzelnes Bit an der Stelle, an dem ursprünglich das am wenigsten Signifikante Bit stand. Es ist die Verundung aus allen vier Bits, und nur dann gesetzt, wenn tatsächlich alle vier Bits gesetzt waren.

Die Funktionsweise wird am in Abb. 3.6 abgebildeten Beispiel deutlich. Das oben links dargestellte farbenspezifische Bitboard soll auf Spielende geprüft werden. In der ersten Zeile wird das originale Bitboard mit dem um eins nach rechts verschobenen verundet, in der zweiten Zeile wird das Resultat der ersten Zeile mit dem um zwei Stellen nach rechts verschobenen Resultat verundet.

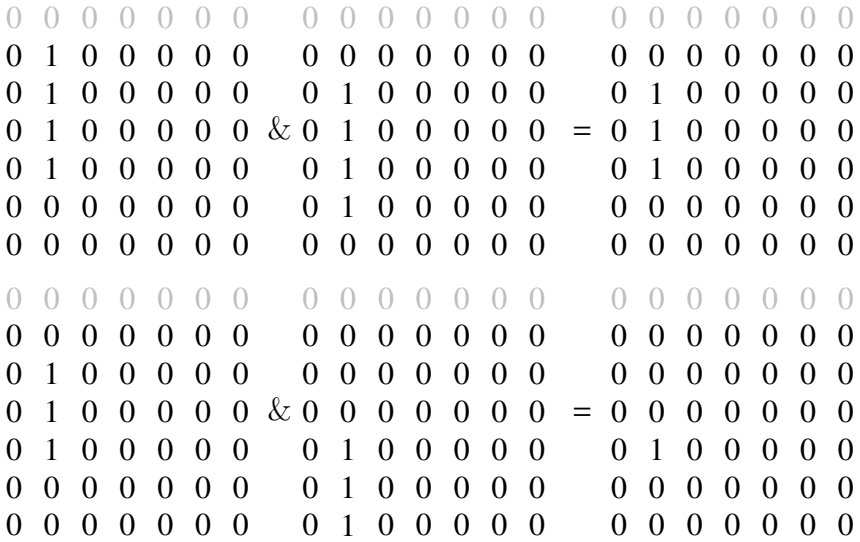

Abbildung 3.6: Beispiel 2: Überprüfung auf Spielende in vertikaler Richtung

Somit wird auch schnell klar, warum die im Abschnitt *Bitboards* erwähnte freie Reihe benötigt wird, sie stellt eine Trennung der einzelnen Spalten dar. Ansonsten würden zum Beispiel drei Steine in den obersten Spielfeldern der ersten Spalte als Sieg anerkannt werden, sobald sich ebenfalls ein Spielstein auf dem untersten Feld der zweiten Spalte befindet.

3.2 Bewertung

Das Ziel einer Bewertungsfunktion ist die Ordnung von Zuständen. Im Falle von nicht-terminalen Zuständen, das sind solche in denen das Spiel noch nicht entschieden ist, soll sie eine ungefähre Auskunft darüber geben, welcher Spieler gerade den Vorteil hat. Sie wird gebraucht, wenn die Suche nur mit limitierter Suchtiefe arbeiten soll. Sollte diese Suchtiefe erreicht sein, ohne dass der Knoten ein tatsächlicher Blattknoten ist, muss sie verwendet werden, um dem Knoten einen geschätzten Wert zuzuordnen. Gleichzeitig ist aber ebenfalls eine Ordnung der terminalen Zustände notwendig um frühe Gewinne von späten zu unterscheiden, und diese zu bevorzugen. Da es sich bei Vier-Gewinnt um ein Spiel für zwei Spieler handelt, kann die Bewertung eines einzelnen Zustands mit Hilfe eines einzelnen *Integers* angegeben werden. Dabei stehen positive Werte für den beginnenden Spieler, also Weiß, und negative Werte für den anderen.

3.2.1 Bewertung von nicht-terminalen Zuständen

Eine Bewertungsfunktion für nicht-terminale Zustände kann in Vier-Gewinnt umgesetzt werden, indem jedem Spielfeld ein fester Wert zugeordnet wird. Dieser Wert berechnet sich aus der Anzahl möglicher Gewinnpositionen, in die das jeweilige Feld involviert ist. Die Felder an den Ecken des Spielfeldes erhalten also den Wert drei. Die möglichen Kombinationen für die untere linke Ecke des Spielfeldes sind die drei Kombinationen, die nach oben, diagonal rechts oben und nach rechts führen. Die Werte für alle Spielfelder sind in Abb. 3.7 dargestellt.

3	4	5	7	5	4	3
4	6	8	9	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	9	8	6	4
3	4	5	7	5	4	3

Abbildung 3.7: Bewertung der Spielfelder

Die beiden Felder in der Mitte erhalten also die höchste Bewertung. Der Bewertung eines Zustands berechnet sich nun durch aufsummieren der einzelnen Felderwerte abhängig von den darauf befindlichen Spielsteinen. Alle Felder mit weißen Spielsteinen werden mit Faktor 1 gewichtet, alle Felder mit schwarzen Spielsteinen mit Faktor -1 und alle leeren Felder mit Faktor 0.
Um eine effiziente Berechnung einer solchen Bewertung durchzuführen, wird diese inkrementell nach jedem Zug aktualisiert. Das bedeutet, dass das Spielbrett in seiner Ausgangsposition eine Bewertung von 0 erhält, und dieser Wert dann nach jedem Zug abhängig von Spieler und Spielfeld aktualisiert werden muss. Um hierfür keine unnötige Fallunterscheidung durchführen zu müssen wird der Wert ähnlich wie im Negamax-Algorithmus immer so berechnet, dass der aktuelle Spieler als maximierender Spieler betrachtet wird. Nach jedem Zug muss also insgesamt folgende Berechnung vorgenommen werden:

$$e_{neu} = -e + \text{Spielfeldwert}$$

Dabei ist e die Bewertung des Zustands vor dem aktuellen Zug und e_{neu} entsprechend die Bewertung nach dem aktuellen Zug. Die Bewertung des aktuellen Zustands für den Spieler, der gerade am Zug ist, ist dann gleich $-e$. Zum Beispiel wäre die Bewertung der Position, in der Schwarz am Zug ist, nachdem Weiß das Spiel mit der mittleren Spalte eröffnet hat, dann gleich -7.

Diese Heuristik ist zwar sehr schnell berechenbar, allerdings kann sie in vielen Situationen aber auch ein wenig hilfreiches Resultat liefern. Die folgende Abbildung zeigt einen Zustand, in dem Weiß im nächsten Zug gewinnen kann.

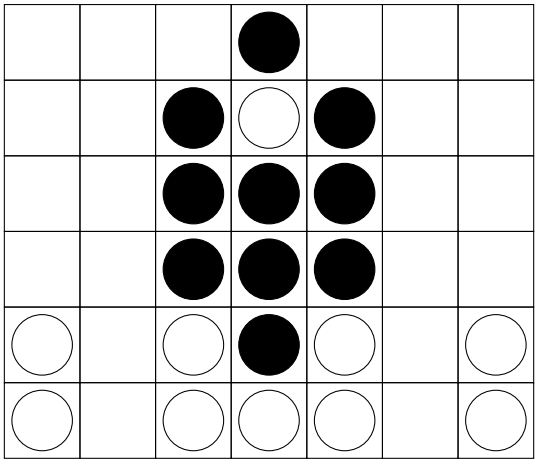

Abbildung 3.8: Beispiel 3: Bewertung zugunsten von Schwarz, obwohl Weiß gewinnt

Trotzdem beläuft sich die entsprechende Bewertung des Zustands auf den Wert -46, weil Schwarz die zentralsten Felder belegt hat. Diese spezielle Position ist zwar quasi unerreichbar, da Weiß mehrere Gewinnmöglichkeiten ignorieren müsste, und Schwarz diese nicht verhindern würde, jedoch spiegelt die Bewertung eine extreme unausgewogene Position, zugunsten von Schwarz wieder, obwohl Weiß hier ganz klar gewinnen kann. Damit ist diese Bewertung zwar simpel zu berechnen, jedoch definitiv nicht optimal. Außerdem kann das ständige Aktualisieren dieser Bewertung, bei tiefen Suchen, in denen das Tiefenlimit groß genug ist, um alle Knoten zu erreichen, auch signifikante Laufzeiteinbußen mit sich bringen, da eine solche Bewertung in diesen Fällen gar nicht notwendig ist.

3.2.2 Bewertung von terminalen Zuständen

Für die Ordnung von terminalen Zuständen wird eine andere Heuristik angewandt. Sie zählt die Anzahl der noch nicht gespielten Felder und gewichtet diese dann, abhängig von dem Spieler der gewonnen hat, mit Faktor 1 oder -1. Im Falle eines Unentschieden wird ihr Wert gleich 0. Die Idee ist also die Bewertung als Differenz aus Anzahl von spielbaren Feldern und Anzahl an durchgeführten Zügen zu berechnen. Allerdings werden Spiele in denen ein Spieler durch das Spielen des letzten Spielsteines gewinnt dann ebenfalls mit 0 bewertet, unabhängig davon welcher der Spieler letztendlich gewonnen hat. Um dieses Problem zu umgehen wird zur Anzahl noch nicht gespielter Felder ein Offset von 1 aufaddiert. Auf diese Weise deutet ein höherer Absolutbetrag auf einen früheren Sieg hin. Da Vier-Gewinnt nach frühestens sieben Zügen von MAX gewonnen werden kann, entspricht der Wert der maximalen für MAX erreichbaren Bewertung:

$$1 \cdot ((42 - 7) + 1) = 36$$

Mit diesen Bewertungsfunktionen ist es allerdings möglich, dass in Suchen mit limitierter Tiefe, in der beide Bewertungsfunktionen zum Einsatz kommen, ein Zustand, in dem Weiß einen Positionsvorteil hat, aber das Spiel verliert, für Weiß besser bewertet wird, als ein Sieg, weil die Wertebereiche beider Bewertungsfunktionen eine nicht-leere Schnittmenge besitzen. Die Information, ob der Zustand nun terminal war oder nicht geht dabei verloren. Um diesem Problem aus dem Weg zu gehen, sollte sichergestellt sein, dass die Bewertung einer Gewinnposition in einem terminalen Zustand immer einen höheren Absolutbetrag erhält, als die einer Vorteilsposition in einem nicht-terminalen Zustand.

Umsetzen lässt sich dies durch ein *Offset*, welches zur Bewertung hinzu addiert wird, falls es sich um einen terminalen Zustand handelt. Eine Abschätzung für die maximale erreichbare nicht-terminale Bewertung lässt sich folgendermaßen berechnen. Beide Spieler wählen ungeachtet der Spielregeln von Vier-Gewinnt abwechselnd Felder auf dem Spielbrett, der Prozess kann jederzeit, nach einem Zug von Weiß, unterbrochen werden, sodass Weiß immer einen Zug mehr gespielt hat als Schwarz. Weiß wählt dabei immer das Spielfeld mit höchstmöglicher Bewertung und Schwarz das mit minimaler Bewertung. Damit erhält Schwarz alle äußeren Felder mit Bewertung ≤ 5 und Weiß alle übrigen, bis auf eines, mit Bewertungen ≥ 6. Dementsprechend wächst die Differenz der Bewertung beider Spieler mit jedem Paar von Zügen stetig an, also führt auch ein früherer Stopp nicht zu einer höheren Bewertung. Summiert man alle Felder von Weiß aus Abb. 3.7 auf erhält man den Wert 184, der Wert für Schwarz liegt bei 84. Die maximal erreichbare Bewertung liegt also bei 100. Dieser Wert ist zwar praktisch nicht erreichbar, er stellt aber dennoch ein obere Schranke für die Bewertung eines nicht-terminalen Zustands dar. Summiert man zur Bewertung jedes terminalen Zustands 100 auf, ist also sichergestellt, dass eine Gewinnposition für einen Spieler immer als höherwertig angesehen wird, als eine Vorteilsposition für denselben Spieler. Allerdings wächst die Größe des benötigten Suchfensters damit ebenfalls stark an.

3.3 Integration der Suchoptimierungen

Dieser Abschnitt soll darlegen wie die in Abschnitt 2.3 genannten Suchoptimierungen effizient implementiert werden können.

3.3.1 Zugsortierung

Wie schon in Abschnitt 2.3.1 beschrieben können Züge in Vier-Gewinnt effizient nach Spalten sortiert werden. Ähnlich zur Bewertung nicht-terminaler Zustände, wird angenommen, dass Spielfelder im Zentrum des Spielbrettes einen höheren strategischen Wert besitzen, als Randsteine. Wenn während der Suche über die Kindknoten des aktuellen Zustands iteriert wird, kann hierbei direkt über eine festgelegte Spaltenreihenfolge iteriert werden. In diesem Fall wurde die Reihenfolge $4, 3, 5, 2, 6, 1, 7$ verwendet, wobei 1 der linkesten und 7 der rechtesten Spalte entspricht. Diese Reihenfolge priorisiert zwar die linke Seite des Spielfeldes strikt gegenüber der Rechten, sollte aber, aufgrund der Symmetrie von Vier-Gewinnt, äquivalent zur Reihenfolge $4, 5, 3, 6, 2, 7, 1$ sein. Um eine solche Priorisierung jedoch zu umgehen, könnte eine Randomisierung der Reihenfolge vorgenommen werden, darauf wurde hier allerdings aufgrund von einer potentiellen Leistungsabnahme und der dadurch eingeschränkten Reproduzierbarkeit verzichtet.

3.3.2 *Transposition-Table*

Wie schon in Abschnitt 2.3.2 erwähnt, wird für jeden untersuchten Zustand ein Eintrag in der *Transposition-Table* angelegt. Zur eindeutigen Zuordnung, der Einträge der *Transposition-Table* zu den Zuständen des Spiels, wird eine Hashfunktion verwendet, die jedem Zustand einen eindeutigen Wert zuordnet. Dieser Wert wird durch Summieren beider Bitboards errechnet, dieser Wert ist eindeutig und stellt jeden Zustand als ein 49-Bit *unsigned Integer* dar. Um die Einträge nun in einem Speicherbereich fester Größe unterzubringen, wird eine weitere Hashfunktion benötigt, die jedem Zustand

einen Speicherort bzw. Index zuweist. Um den Speicher möglichst effizient auszunutzen, wird die Größe der *Transposition-Table* als Primzahl festgelegt und der Index per Modulo berechnet, um eine möglichst gleichmäßige Verteilung der Positionen auf alle möglichen Indizes zu gewährleisten. Hier wurde eine *Transposition-Table* mit etwas mehr als 2^{24} Einträgen verwendet. Da somit unter idealen Bedingungen nur etwas weniger als 17 Millionen Zustände gleichzeitig gespeichert werden können, sind Kollisionen, gerade bei komplexeren Spielpositionen, unvermeidbar. Dadurch könnten mehrere Positionen denselben Index zugewiesen bekommen und sich gegenseitig überschreiben. Um sicherzustellen, dass der Algorithmus die richtige Position aus der *Transposition-Table* liest, muss der eindeutige Hash-Wert der Position ebenfalls gespeichert und beim Lesen abgeglichen werden. Ein Eintrag in der *Transposition-Table* benötigt insgesamt also folgende Variablen: Den Hashwert des Zustands gespeichert als 64-Bit *unsigned Integer*, die Bewertung als 32-Bit *Integer*, das Flag und die Suchtiefe, jeweils als 8-Bit *unsigned Integer*. Durch Auffüllung (*Padding*) durch den Compiler erhält jeder Eintrag so eine Größe von 16 Bytes, die gesamte *Transposition-Table* nimmt also 268.4 Megabytes in Anspruch, und sollte damit sparsam genug sein, um auf den meisten modernen Computern problemlos eingesetzt werden zu können.

4 Optimierungen

In diesem Kapitel werden einige spezielle auf Vier-Gewinnt zugeschnittene Optimierungen des Frameworks, aber auch Suche diskutiert. Ein Teil stammt dabei aus einem verwandten Projekt.

4.1 Externe Optimierungen

Dieser Abschnitt beschreibt und diskutiert jene Optimierungen, die aus dem Projekt von Pascal Pons [9] stammen. Das Projekt zeichnet sich durch eine besonders schnelle Suche aus und scheint die derzeit schnellste Implementierung eines Vier-Gewinnt-*Solvers* zu sein. Im Folgenden sollen vier dieser speziellen Optimierungen erklärt und diskutiert werden.

4.1.1 Optimierte Spielende-Erkennung

Eine fundamentale Verbesserung ist die Optimierung der Spielende-Erkennung. Anstatt, wie in Abschnitt 3.1.3 erklärt, das Spielende zu bestimmen, indem geprüft wird, ob einer der Spieler vier aufeinander folgende Spielsteine hat, wird schon ein Schritt vorher ein Bitboard berechnet, auf dem alle Bits, den Spielfeldern entsprechen, die zum Sieg führen, gesetzt sind. Dabei sind aber ebenfalls Bits zu Spielfeldern gesetzt, die eventuell gar nicht erreichbar sind, weil mindestens eines der darunterliegenden Felder noch nicht gespielt wurde. Ein solches Bitboard für das Beispiel aus Abb. 3.2 ist in Abb. 4.1 für Schwarz dargestellt.

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Abbildung 4.1: Beispiel 1: Gewinnpositionen für Schwarz

Dieses resultierende Bitboard wird nun mit einem Bitboard aller aktuell möglichen Züge verundet und man erhält ein Bitboard, welches alle Züge anzeigt, die sofort zum Sieg führen. Dies wird wiederum verwendet, um erzwungene Züge zu erkennen. In einer weiteren Funktion wird ein Bitboard

berechnet, das eine Maske für alle Züge darstellt, die nicht unmittelbar zum Sieg für den Gegenspieler führen. Ist das Resultat dieser Funktion nun gleich null, bedeutet das, dass der aktuell zu ziehende Spieler sofort gewinnen muss, um nicht im nachfolgenden Zug zu verlieren. Ein solcher Fall tritt allerdings nie ein, da der Gegenspieler einen solchen Fall nicht zulassen könnte. In Algorithmus 8 ist die Funktionsweise dieses Algorithmus dargestellt.

Algorithmus 8 Berechnung aller Züge, die nicht zum Sieg des Gegenspielers führen [9]

```
1: function possibleNonLosingMoves
2:     possible_mask ← possible( )
3:     opponent_win ← opponentWinningPosition( )
4:     forced_moves ← possible_mask & opponent_win
5:     if forced_moves then
6:         if forced_moves & (forced_moves − 1) then
7:             return 0
8:         else possible_mask ← forced_moves
9:     return possible_mask & ¬(opponent_win >> 1)
```

Das Bitboard $possible\_mask$ zeigt alle spielbaren Felder an und $opponent\_win$ zeigt alle Felder an, die zum Sieg für den Gegenspieler führen, wenn dieser eines dieser Felder spielt. Daraus kann nun $forced\_moves$ berechnet werden, hier werden alle erzwungenen Züge gespeichert. Gibt es mehr als einen erzwungenen Zug, so kann der aktuell zu ziehende Spieler nicht mehr gewinnen und es wird ein leeres Bitboard, also null zurückgegeben. Andernfalls werden alle möglichen Züge zurückgegeben, die dem Gegenspieler nicht die Möglichkeit für einen Sieg im nächsten Zug bieten. Diese Funktion wird nun wie folgt in die Suche integriert: Zu Beginn jedes rekursiven Aufrufs der Alpha-Beta-Suche wird die Funktion einmal aufgerufen. Ist das Resultat gleich null bedeutet dies, dass ein Sieg für den Gegenspieler unumgänglich ist und die Suche kann abgebrochen werden. Andernfalls kann das Resultat noch an einer weiteren Stelle genutzt werden. An der Stelle an der die Alpha-Beta-Suche über alle möglichen Züge iteriert kann schnell mittels einer Verundung geprüft werden ob der entsprechende Zug unmittelbar zu einem Sieg des Gegenspielers führen würde. Die Verwendung der optimierten Spielende-Erkennung hat also gleich zwei positive Aspekte. Einerseits kann eine Suche frühzeitig abgebrochen werden, sobald ein Sieg für den Gegenspieler innerhalb von zwei Zügen unvermeidbar ist und andererseits werden beim Iterieren über mögliche Züge keine offensichtlich sinnlosen Züge mehr untersucht.

4.1.2 Erreichbare Bewertungen

In einem Zustand, indem beispielsweise bereits 20 Züge gespielt wurden, lässt sich eine Bewertung von 30 nicht mehr erzielen, da nur noch 22 freie Felder existieren, wenn man das *Offset* aus Abschnitt 3.2.2 außer Acht lässt. Die maximale bzw. minimale erreichbare Bewertung ist also 23 bzw. -23. Diese Eigenschaft kann benutzt werden, um den Spielbaum für die Suche weiter zu beschneiden, indem in jedem Zustand die maximale, bzw. minimale erreichbare Bewertung berechnet wird, um α und β entsprechend zu aktualisieren, falls so das Suchfenster verkleinert werden kann. Während der Suche wird dies allerdings erst getan, nachdem festgestellt wurde, dass der aktuelle Zustand nicht terminal ist. Mit der ursprünglichen Spielende-Erkennung, hieße das, dass das Spiel frühestens mit

dem nächsten Zug gewonnen werden kann. Die maximal erreichbare Bewertung berechnet sich also folgenderweise:

maximale Bewertung = Anzahl an Spielfeldern - Anzahl gespielter Züge - 1 + *Offset*

Die minimale Bewertung ist dementsprechend das negative von der maximalen Bewertung. Mit der optimierten Spielende-Erkennung kann das Spiel nun nicht mehr mit dem nächsten Zug gewonnen werden, sondern erst in dem darauf folgendem. Somit lässt sich auch die maximale Bewertung weiter einschränken:

maximale Bewertung = Anzahl an Spielfeldern - Anzahl gespielter Züge - 2 + *Offset*

Hierdurch entstehen im Optimalfall zusätzliche *Cutoffs*, die den Suchraum weiter verringern.

4.1.3 Zugsortierung

Im Gegensatz zur in Abschnitt 2.3.1 Umsetzung einer Zugsortierung, wird hier keine statische Sortierung auf Basis der Spalten umgesetzt. Stattdessen wird eine neue Heuristik eingeführt, um eine Ordnung von Zügen herzustellen. Dazu wird die in Abschnitt 4.1.1 beschriebene Funktion zur Berechnung eines Bitboards mit gesetzten Bits an Positionen die zum Sieg für den Spieler führen, verwendet. Diese Funktion wird für das Resultat der Veroderung aus $current$ und dem zu bewertenden Zug gebildet. Die Bewertung wird nun durch Zählen der Bits berechnet. Es werden also Züge bevorzugt, die neue Gewinnchancen eröffnen. Sollten zwei Züge die gleiche Bewertung erhalten, wird weiterhin, die ursprüngliche spaltenbasierte Heuristik als Tiebreak verwendet.

4.1.4 Suche

Pascal Pons benutzt in seinem Projekt ebenfalls die in Abschnitt 2.6 eingeführte NegaC*-Suche. Allerdings wurde hier eine spezielle Optimierung vorgenommen. Im Normalfall würde der Algorithmus stetig eine Nullfenstersuche um den Mittelwert aus min und max durchführen. Die optimierte Variante führt im Gegensatz dazu eine Nullfenstersuche um $\lceil \frac{min}{2} \rceil$ durch, wenn

$$\lceil \tfrac{min}{2} \rceil < \lfloor \tfrac{min+max}{2} \rfloor \leq 0 \text{ gilt.}$$

Eine Nullfenstersuche um $\lfloor \frac{max}{2} \rfloor$ wird dann analog durchgeführt, wenn

$$0 \leq \lfloor \tfrac{min+max}{2} \rfloor < \lfloor \tfrac{max}{2} \rfloor \text{ gilt.}$$

Anders formuliert bedeutet das, für diejenigen Iterationen, in denen

$$[-1, 1] \subseteq [min, max]$$

gilt, dass eine Suche um den Mittelwert der für MIN vorteilhaften, erreichbaren Bewertungen durchgeführt wird, wenn $min \leq max$ gilt. Ansonsten wird eine Suche um den Mittelwert der für MAX vorteilhaften, erreichbaren Bewertungen durchgeführt. Somit werden Bewertungen mit hohen Beträgen schneller gefunden als zuvor. Allerdings bedeutet das, dass die Suche für Positionen, die in einem Unentschieden enden besonders viele Iterationen benötigen. Es wird also vorausgesetzt, dass die Suche mehr für Positionen, deren Minimax-Wert einen hohen Betrag hat, aufgerufen wird, als für andere. Eine Mögliche Implementierung ist in Algorithmus 9 aufgeführt. Die rot markiertet Zeilen sind dabei nur in der optimierten Version vorhanden.

Algorithmus 9 Suchalgorithmus des Projekts von Pascal Pons [9]

```
1:  function negaCStarOptimized(node, f)
2:      min ← −∞
3:      max ← ∞
4:      while min < max do
5:          med ← min + (max − min)/2
6:          if med ≤ 0 and min / 2 < med then
7:              med ← min/2
8:          else if med ≥ 0 and max / 2 > med then
9:              med ← max/2
10:         r ← alphabetaWithMemory(node, med, med + 1)
11:         if r ≤ med then
12:             max ← r
13:         else
14:             min ← r
15:     end while
16:     return min
```

4.1.5 minimales Suchfenster

Gerade für die Verwendung eines Suchalgorithmus wie die *NegaC*-Suche*, hilft ein kleineres Suchfenster, um die Anzahl nötiger Iterationen zu minimieren. Das Hinzufügen eines *Offsets*, wie in Abschnitt 3.2.2 beschrieben, scheint hier eher kontraproduktiv, da sie den möglichen Wertebereich und somit, die Größe des für eine vollständige Suche notwendigen Suchfensters, enorm steigert. Eine Möglichkeit dies zu umgehen besteht darin, eine Suche immer bis zum Ende des Spielbaumes durchzuführen, sodass eine Bewertungsfunktion für nicht-terminale Zustände obsolet wird.

Die Bewertung terminaler Zustände kann allerdings ebenfalls weiter optimiert werden. In Abschnitt 3.2.2 wurde die Berechnung der Bewertungsfunktion folgenderweise definiert:

$$e = p \cdot (\text{Anzahl übriger Spielfelder} + 1)$$

wobei $p = 1$ für MAX und $p = -1$ für MIN. MAX kann das Spiel nach frühestens sieben Zügen gewinnen. Gelingt ihm dies nicht hat er jeweils im übernächsten Zug eine weitere Chance. Dementsprechend bedeutet ein Sieg für MAX, dass die Anzahl insgesamt gespielter Spielsteine ungerade sein muss. Für MIN gilt analog, dass die Anzahl insgesamt gespielter Spielsteine gerade sein muss. Da alle positiven Werte der Bewertungsfunktion, ausgeschlossen die 0, einen Sieg für MAX markieren ist klar, dass, durch hinzurechnen des Offsets, die Bewertung keine ungeraden Werte annehmen kann, jeder zweite Wert wird also niemals angenommen. Dementsprechend Arbeitet die von Pascal Pons genutzte Bewertungsfunktion nur mit einem Suchfenster von etwa der halben Größe.

4.2 Weitere Optimierungen

Dieser Abschnitt stellt zwei weitere Optimierungen vor, die die Suche weiter verbessern können.

4.2.1 *Iterative Deepening*

Iterative Deepening oder iterative Tiefensuche ist eine Möglichkeit die Vorteile von Tiefen- und Breitensuche zu vereinen. Die Idee besteht darin, die Tiefensuche wiederholt mit zunehmender Suchtiefe aufzurufen. Dadurch wird der Suchbaum, ähnlich wie zur Breitensuche, Ebene für Ebene traversiert, ohne dabei den Speicherbedarf einer Breitensuche zu haben [7]. *Iterative Deepening* kann zudem in Kombination mit einer *Transposition-Table* verwendet werden, um die Heuristik der Zugsortierung zu verbessern. Dazu werden die Ergebnisse einer vorherigen Suche mit geringerer Suchtiefe verwendet, um Züge entsprechend zu sortieren [12]. Außerdem bringt *Iterative Deepening* den Vorteil, dass schon sehr früh ein Ergebnis bestimmt wird, welches schrittweise, mit jeder weiteren Suchebene, verbessert wird. So kann ein Programm, dem nur begrenzt viel Zeit zur Verfügung steht auch ein gutes Ergebnis finden, wenn die Zeit nicht für eine vollständige Suche reicht.

4.2.2 gespiegelte Positionen in der *Transposition-Table*

Eine weitere Vier-Gewinnt-spezifische Erweiterung der *Transposition-Table* ergibt sich aus der Beobachtung, dass jede Position eine äquivalente gespiegelte Position mit derselben Bewertung hat. Das bedeutet, dass eine Position nicht weiter untersucht werden muss, wenn die gespiegelte Position bereits untersucht wurde. Um dies umzusetzen wird beim Lesen der *Transposition-Table*, falls der gesuchte Eintrag nicht existiert, ebenfalls nach der gespiegelten Position gesucht. Besonders in der Anfangsposition, aber auch in jeder anderen symmetrischen Position, kann dies zu einer erheblichen Reduzierung der zu untersuchenden Züge führen, indem theoretisch nur noch maximal vier von bis zu sieben möglichen Zügen untersucht werden müssen. Das Berechnen des gespiegelten Spielbrettes, sowie der zweite Zugriff auf die *Transposition-Table* erfordert jedoch zusätzlichen Aufwand, der die Suche gleichzeitig verlangsamt. Zudem zeigt Abb. 4.2 eine Position, von der aus keine zwei Positionen erreichbar sind, die zueinander symmetrisch sind. Auf dem gespiegelten Spielbrett liegt ein schwarzer Stein, wo vorher ein weißer war und umgekehrt. In einer solchen Position führt die Nutzung einer spiegelnden *Transposition-Table*, vorausgesetzt sie ist zu Beginn der Suche leer, also niemals zu einer Leistungsverbesserung. Ein solcher Vorteil wird also nur dann erzielt, wenn die *Transposition-Table* bereits einen passenden Eintrag enthält, oder der zu untersuchende Zustand sich spiegeln lässt, sodass kein weißer Stein dort liegt, wo vorher ein schwarzer war oder umgekehrt.

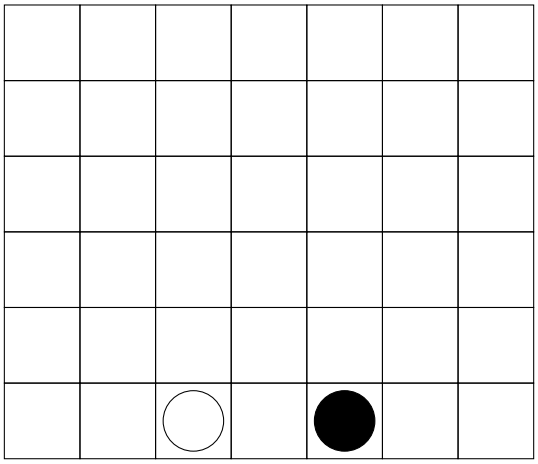

Abbildung 4.2: Früheste nicht spiegelbare Position

5 Vergleich der Suchstrategien

Im folgenden Kapitel sollen die verschiedenen Algorithmen und Optimierungen bezüglich ihrer Laufzeit und Effektivität untereinander verglichen werden. Dazu werden mehrere Simulationen durchgeführt und ausgewertet.

5.1 Art des Vergleichs

Zum Vergleich der verschiedener Algorithmen und Optimierungen, wird eine Suche auf einem festen Datensatz von Spielpositionen durchgeführt. Die dafür verwendeten Datensätze stammen ebenfalls aus dem in Abschnitt 4.1 angesprochenen Projekt von Pascal Pons [9]. Dort gibt es insgesamt sechs Datensätze, klassifiziert nach Spielphase und Schwierigkeit (Anzahl übriger Züge zur vollständigen Berechnung der *Principal-Variation*). Tab. 5.1 zeigt eine Übersicht aller Datensätze und deren Unterschiede.

Datensatz	Name	gespielte Züge	übrige Züge
L3_R1	End-Easy	$28 <$ Züge	übrig < 14
L2_R1	Middle-Easy	$14 <$ Züge ≤ 28	übrig < 14
L2_R2	Middle-Medium	$14 <$ Züge ≤ 28	$14 \leq$ übrig < 28
L1_R1	Begin-Easy	Züge ≤ 14	übrig < 14
L1_R2	Begin-Medium	Züge ≤ 14	$14 \leq$ übrig < 28
L1_R3	Begin-Hard	Züge ≤ 14	$28 \leq$ übrig

Tabelle 5.1: Übersicht der zum Test verwendeten Datensätze [9]

Jeder Datensatz enthält 1000 Positionen und eine zugehörige Bewertung, insgesamt gibt es also 6000 Positionen. Um die Funktionen einzeln auszuwerten wurde ein *Interface* implementiert, welches über die Kommandozeile eine Position, eine Suchtiefe und den Namen des entsprechenden Algorithmus übergeben bekommt. Die Position wird anschließend von dem jeweiligen Algorithmus für die gewählte Suchtiefe ausgewertet und Bewertung und benötigte Laufzeit werden zurückgegeben. Das Programm terminiert erst, per Eingabe eines festgelegten Strings. Die *Transposition-Table* wird zwischen zwei Aufrufen nicht geleert. Das Interface wird von einem Python-Skript benutzt um die zuvor eingelesenen Datensätze entsprechend zu evaluieren und passende Diagramme zu erzeugen. Für den Vergleich wird jeder Algorithmus iterativ mit steigender Suchtiefe aufgerufen um den Zusammenhang von Laufzeit und Suchtiefe darzustellen. Die Suchtiefe beschreibt hierbei die Anzahl noch zu untersuchender Züge. Wenn eine Position, in der schon 20 Züge gespielt wurden also mit Suchtiefe

30 aufgerufen wird, ist das Ergebnis äquivalent zur Suchtiefe 22, da in beiden Fällen die Suchtiefe groß genug ist, um den gesamten Suchbaum, mit allen 42 Ebenen zu explorieren. Jeder Datenpunkt der nachfolgend dargestellten Graphen beschreibt dabei die durchschnittliche Zeit, die ein bestimmter Algorithmus benötigt hat, um eine Position des jeweiligen Datensatzes mit entsprechender Suchtiefe auszuwerten. Das Programm wird nach jeder Iteration, also für jede Suchtiefe, neu aufgerufen, sodass mit jeder neuen Suchebene die *Transposition-Table* geleert wird. Die Anzahl der untersuchten Zustände wird am Ende jeder Iteration, gemittelt über die Anzahl der Positionen des Datensatzes, übergeben und ist in den nachfolgenden Graphen ebenfalls durch gepunktete Linien dargestellt. Das Testsystem besteht aus einem Intel i5-6500-Prozessor mit 16 Gigabyte Arbeitsspeicher.

5.2 Ergebnisse vor Optimierungen

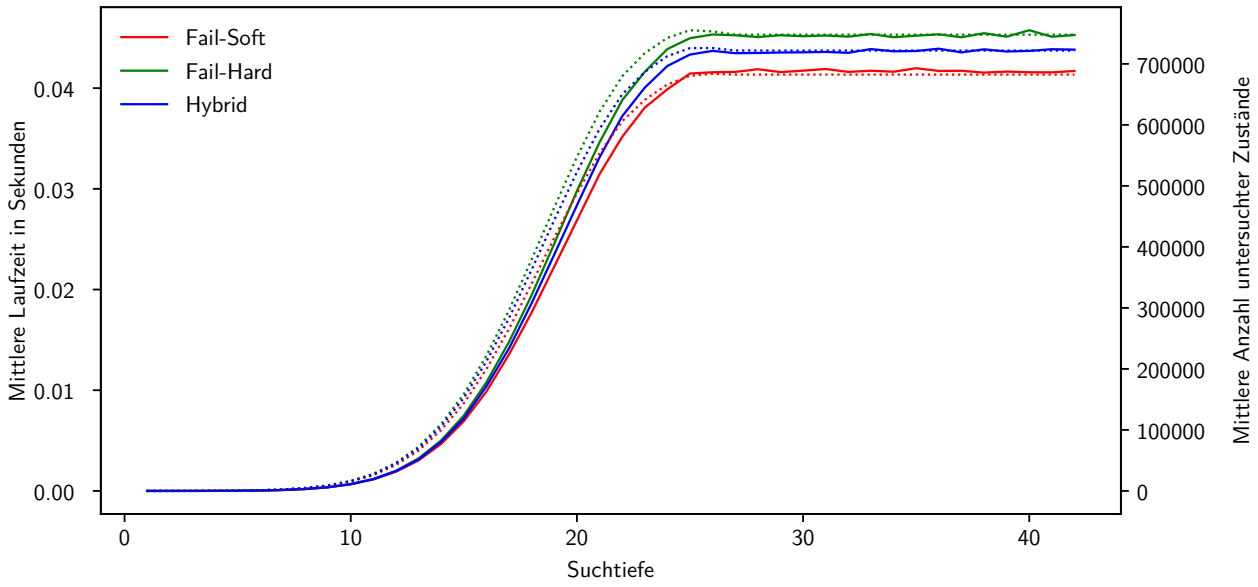

Abbildung 5.1: Vergleich von *Fail-Soft*, *Fail-Hard* und der Hybridvariante auf dem Datensatz L2_R1

Abb. 5.1 zeigt einen Vergleich von *Fail-Soft*-, *Fail-Hard*-, und Hybrid-Implementierung der Alpha-Beta-Suche auf dem Datensatz L2_R1. Sowohl Laufzeit als auch Anzahl untersuchter Zustände stagnieren bei allen Algorithmen bei Suchtiefe 27. Das lässt sich dadurch erklären, dass bei Datensatz L2_R1 minimal 15 Züge gespielt wurden (siehe Tab. 5.1), es sind also maximal noch 27 freie Felder vorhanden. Die Anzahl untersuchter Zustände ist bei der *Fail-Soft*-Implementierung am geringsten und bei der *Fail-Hard*-Implementierung maximal. Die Hybridvariante befindet sich genau zwischen den beiden anderen. Dies passt genau zu den Überlegungen aus Abschnitt 2.2.1, wobei die Anzahl untersuchter Knoten von Hybridvariante zur Fail-Soft-Variante noch einmal stärker reduziert werden kann als von Fail-Hard-Variante zu Hybridvariante. Die Laufzeiten der Algorithmen verhalten sich Recht ähnlich, haben jedoch einen leicht verzögerten Anstieg und steigen entsprechend steiler. Vergleicht man die Algorithmen auf maximaler Suchtiefe, fällt auf, dass das Verhältnis aus Laufzeit und Knotenzahl der Fail-Soft-Variante minimal größer ist, als das der Anderen. Dies lässt sich durch den minimal erhöhten Suchaufwand erklären, der durch das Führen einer zusätzlichen Variable entsteht.

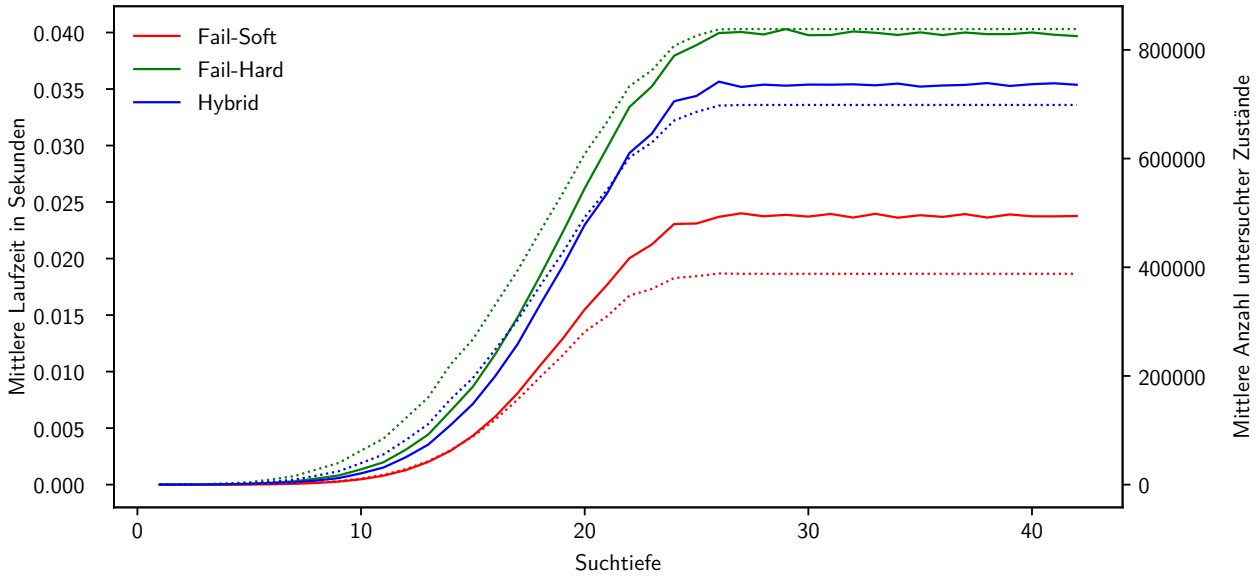

Abbildung 5.2: Vergleich von *Fail-Soft*, *Fail-Hard* und der Hybridvariante auf dem Datensatz L2_R1 bei Verwendung von MTD(f) mit $f = 0$

Allgemein scheint sich die Verwendung einer Fail-Soft-Variante also zu lohnen, im Folgenden werden daher für alle verwendeten Algorithmen Fail-Soft-Implementierungen genutzt.

Ähnliches kann auch in Abb. 5.2 beobachtet werden. Hier werden ähnlich wie zuvor wieder Fail-Soft, Fail-Hard und Hybridvariante gegeneinander getestet, mit dem Unterschied dass diesmal MTD(f), mit Startwert $f = 0$, zur Suche verwendet wurde. Auch hier hat die Fail-Soft-Variante wieder sowohl beste Laufzeit als auch wenigste untersuchte Zustände. Die Differenz zu den anderen beiden Varianten ist hier allerdings viel signifikanter. Die Hybridvariante untersucht mehr als drei mal so viele Zustände wie die Fail-Soft-Variante, und die Fail-Hard-Variante sogar knapp vier mal so viele. Die Ursache hierfür liegt in der Funktionsweise des MTD(f)-Algorithmus. Durch das wiederholte Durchführen einer Nullfenstersuche um das Ergebnis der letzten Iteration nähert sie sich Schritt für Schritt dem tatsächlichen Minimax-Wert. Da eine Fail-Hard-Variante allerdings nur Werte innerhalb des Suchfensters zurück geben kann, ist die Schrittweite einer solchen Implementierung limitiert auf eins. Eine Fail-Soft-Variante hingegen kann durch das Ermöglichen Rückgabewerten außerhalb des Suchfensters größere Schritte zurücklegen und so schneller zum Ergebnis kommen. Dieser Effekt wird durch Verwendung des in Abschnitt 3.2.2 beschriebenen *Offsets* weiter verstärkt. Bei einer initialen Schätzung von 0 und einer Suchtiefe die groß genug ist um einen terminalen Zustand zu erreichen, benötigt eine Fail-Hard-Implementierung im Optimalfall mehr als 100 Iterationen, nur um das *Offset* zu überwinden.

Abb. 5.3 stellt einen Vergleich der verschiedenen Algorithmen, auf dem selben Datensatz, wie zuvor dar. Alle Algorithmen scheinen auch hier erwartungsgemäß bei einer Suchtiefe von 27 zu stagnieren. Die gemessenen Werte für die Alpha-Beta-Suche sind dieselben, wie in Abb. 5.1. Die Anzahl untersuchter Zustände, bei maximaler Suchtiefe, liegt hier durchschnittlich bei ungefähr 680 Tausend mit ca. 42 Millisekunden Laufzeit pro Position. Der nächstbessere Algorithmus ist MTD(f), er

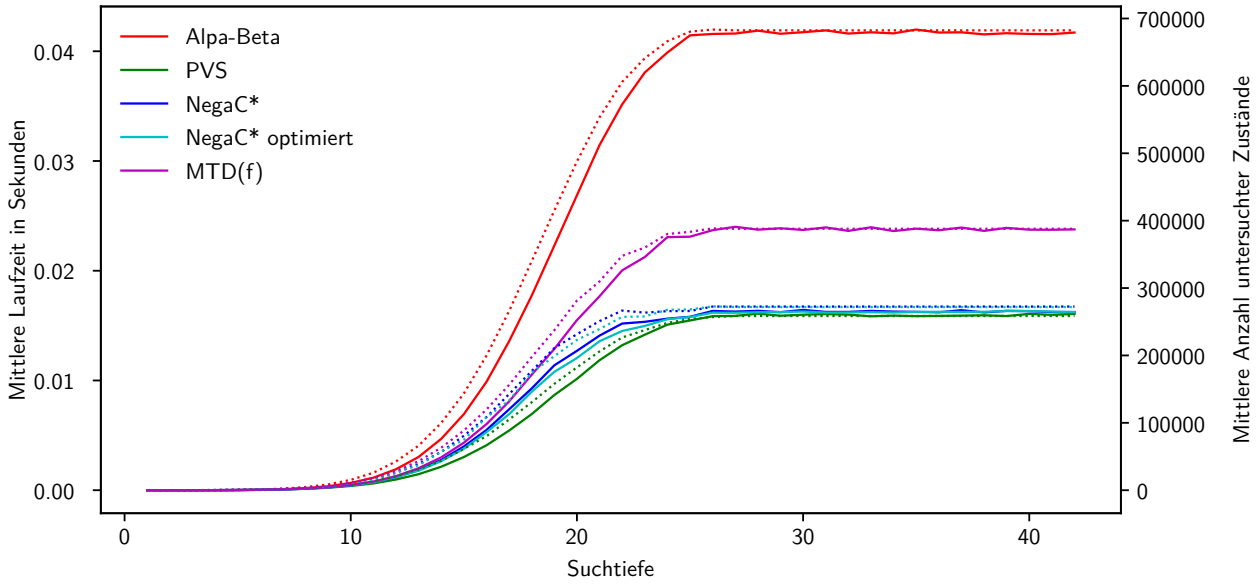

Abbildung 5.3: Vergleich aller Algorithmen auf dem Datensatz L2_R1

untersucht bei voller Suchtiefe knapp 390 Tausend Knoten, also knapp 57% der Alpha-Beta-Suche. Beide Versionen der NegaC*-Suche stagnieren bei etwas über 270 Tausend untersuchten Zuständen, wobei die optimierte Variante minimal mehr Knoten untersucht. Der schnellste Algorithmus ist die *Principal-Variation*-Suche, sie untersucht lediglich 260 Tausend Knoten. Auch hier Verhalten sich die Laufzeiten recht analog zur Anzahl untersuchter Zustände. Es fällt jedoch auf, dass der Vorsprung der *Principal Variation*-Suche bezüglich der Laufzeit kleiner ist, als der der Anzahl untersuchter Zustände. Erklären lässt sich dass durch die etwas komplexere Suchfunktion, im Vergleich zur Alpha-Beta-Suche, welche von beiden NegaC*-Varianten genutzt wird. Daher ist die Laufzeitdifferenz dieser drei Algorithmen auch kaum wahrnehmbar. Alle Algorithmen scheinen eine deutliche Verbesserung zur Alpha-Beta-Suche darzustellen und benötigen so nur minimal nur knapp 39% der Laufzeit.

5.3 Einflüsse der Optimierungen

Die meisten der in Abschnitt 4.1 erklärten externen Optimierungen sind in Abb. 5.4 dargestellt, ausgenommen das minimale Suchfenster, da durch entfernen des *Offsets* keine korrekten Ergebnisse bei nicht maximaler Suchtiefe garantiert werden können. Jeder getestete Algorithmus ist eine Version der Alpha-Beta-*Fail-Soft*-Variante, in der lediglich die genannte Verbesserung eingearbeitet wurde, mit Ausnahme der letzten Version, diese beinhaltet alle Verbesserungen zugleich. Der Datensatz ist derselbe wie zuvor. Der rote Graph zeigt die ursprüngliche Variante ohne Verbesserungen, so wie sie auch in Abb. 5.1 und Abb. 5.3 dargestellt ist. Der grüne Graph zeigt den Verlauf der Variante in der, gemäß Abschnitt 4.1.2, das Suchfenster in jeder Iteration auf tatsächlich erreichbare Bewertung beschränkt wird. Diese Verbesserung beschränkt so die Anzahl untersuchter Zustände auf durchschnittlich etwas mehr als. 420 Tausend, also ungefähr 62% im Vergleich zur klassischen Variante. Das Verhältnis zwischen Laufzeit und Anzahl untersuchter Zustände wächst dabei aber auch geringfügig an, was durch

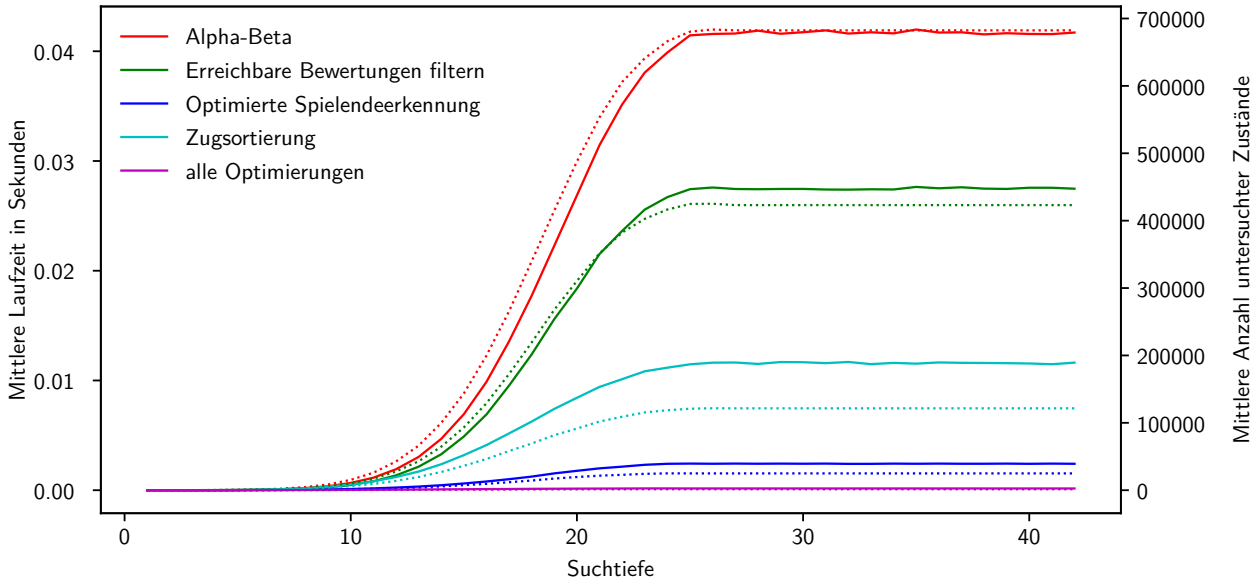

Abbildung 5.4: Vergleich der Optimierungen auf dem Datensatz L2_R1

den zusätzlichen Aufwand zur Berechnung des aktualisierten Suchfensters zu erklären ist. In Dunkelblau ist die Variante dargestellt, welche die optimierte Spielende-Erkennung aus Abschnitt 4.1.1 nutzt. Auch hierbei wächst die relative Laufzeit etwas an, da zusätzlicher Rechenaufwand benötigt wird. Allerdings müssen hierbei nur noch knapp 3.7% der Zustände untersucht werden. Ebenfalls hervorzuheben ist hierbei, dass beide Metriken schon zwei Suchebenen früher stagnieren, im Vergleich zur unveränderten Alpha-Beta-Suche. All dies lässt sich wie folgt erklären: durch die verbesserte Spielende-Erkennung werden Endpositionen schon zwei Züge zuvor gefunden. Somit wird jeder vorige nicht-Blattknoten, zu einem Blattknoten, wenn ein voriger Blattknoten innerhalb von zwei Zügen erreichbar war. Der hellblaue Graph zeigt den Verlauf der Variante, die die Zugsortierung aus Abschnitt 4.1.3 verwendet. Im Vergleich zu den anderen Varianten wächst die relative Laufzeit hier um rund 55% an. Da Aber nur noch knapp 18% der Zustände untersucht werden müssen, liegt die resultierende Laufzeit immer noch deutlich unter der der nicht optimierten Variante. Die Variante, die alle drei Verbesserungen simultan umsetzt untersucht nur noch knapp 0.21% der Zustände und benötigt dafür 0.38% der ursprünglichen Laufzeit. Die relative Laufzeit hat sich insgesamt also fast verdoppelt.

Abb. 5.5 zeigt den Einfluss durch Verwendung von *Iterative Deepening*. Der rote Graph zeigt die Alpha-Beta-Suche, diesmal mit allen Optimierungen, wie zuvor dargestellt. Der Grüne Graph zeigt den Verlauf dieser Variante unter Verwendung von *Iterative Deepening* mit unveränderter Zugsortierung. Allein hierdurch müssen, bei maximaler Suchtiefe, mehr als zehn mal so viele Zustände untersucht werden, wie zuvor, während die Laufzeit auf das etwa siebenfache ansteigt. Das Sortieren der Züge ohne Verwendung von *Iterative Deepening* vergrößert den Suchraum auf das 12-fache und die Laufzeit auf das 18-fache. Beides zusammen produziert einen 44 mal größeren Suchraum mit 67-facher Laufzeit. Der Grund hierfür könnte einerseits bei einer zu schlechten Heuristik zur Bewertung nicht-terminaler Zustände liegen, wodurch das Sortieren basierend auf den Werten aus der *Transposition-Table* keinen wirklichen Mehrwert bietet, aber trotzdem zusätzliche Laufzeitklom-

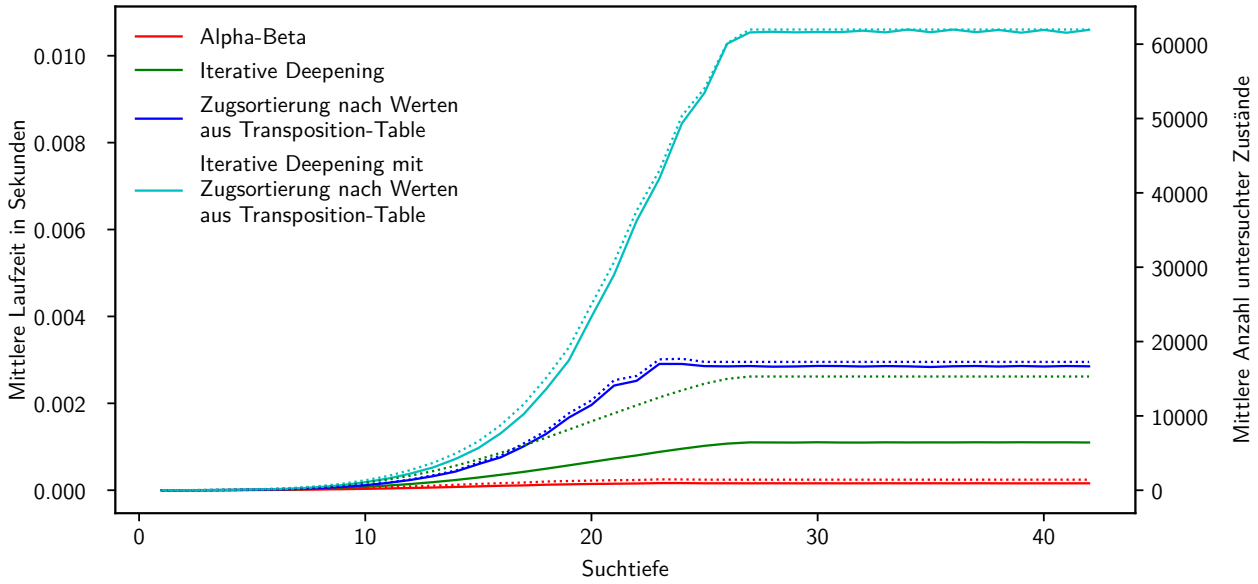

Abbildung 5.5: Vergleich der *Iterative Deepening*-Varianten auf dem Datensatz L2_R1

plexität entstehen lässt. Andererseits besteht auch die Möglichkeit, einer zu starken Heuristik zur Zugsortierung, die damit eine weniger aufwändige und trotzdem bessere Zugsortierung ermöglicht. Außerdem ist die CPU-Cache des Testsystems nicht groß genug, um die gesamte *Transposition-Table* zu speichern, wodurch Zugriffe auf diese für alle Implementierungen teuer werden. Ein Algorithmus der besonders oft auf die *Transposition-Table* zugreift, nimmt dadurch also auch entsprechende Kosten in Kauf. Es ist also klar, dass *Iterative Deepening* zumindest mit den hier verwendeten Heuristiken, und der gewählten Größe der *Transposition-Table*, extreme Nachteile mit sich bringt.

In Abb. 5.6 ist der Vergleich zwischen der optimierten Alpha-Beta-Suche, wie zuvor, und einer Variante, die, wie in Abschnitt 4.2.2 beschrieben, gespiegelte Positionen in der Transposition-Table sucht, dargestellt. Die Variante mit Spiegelung benötigt in etwa 13% mehr Laufzeit, um gesuchte Positionen zu spiegeln und aus der Transposition-Table zu lesen. Trotzdem untersuchen beide Algorithmen, für alle Suchtiefen, exakt gleich viele Zustände. Der Grund hierfür liegt in dem verwendeten Datensatz. Das Spiel ist in allen Positionen des Datensatzes bereits so weit fortgeschritten, dass keine gespiegelten Positionen mehr auftreten können, weil schon die Ursprungspositionen sich gegenseitig ausschließen.

Im Vergleich dazu stellt Abb. 5.7 denselben Vergleich auf dem Datensatz L1_R2 dar. Hier untersucht die Variante, die gespiegelte Positionen beachtet, tatsächlich weniger Zustände, allerdings nur etwas mehr als 1%. Damit benötigt die Spiegelung noch immer ca. 10% mehr Laufzeit für die gleichen Positionen. Es ist also zu erwarten, dass der Nutzen der Spiegelung für Positionen mit weniger gespielten Zügen, im Mittel weiter ansteigt.

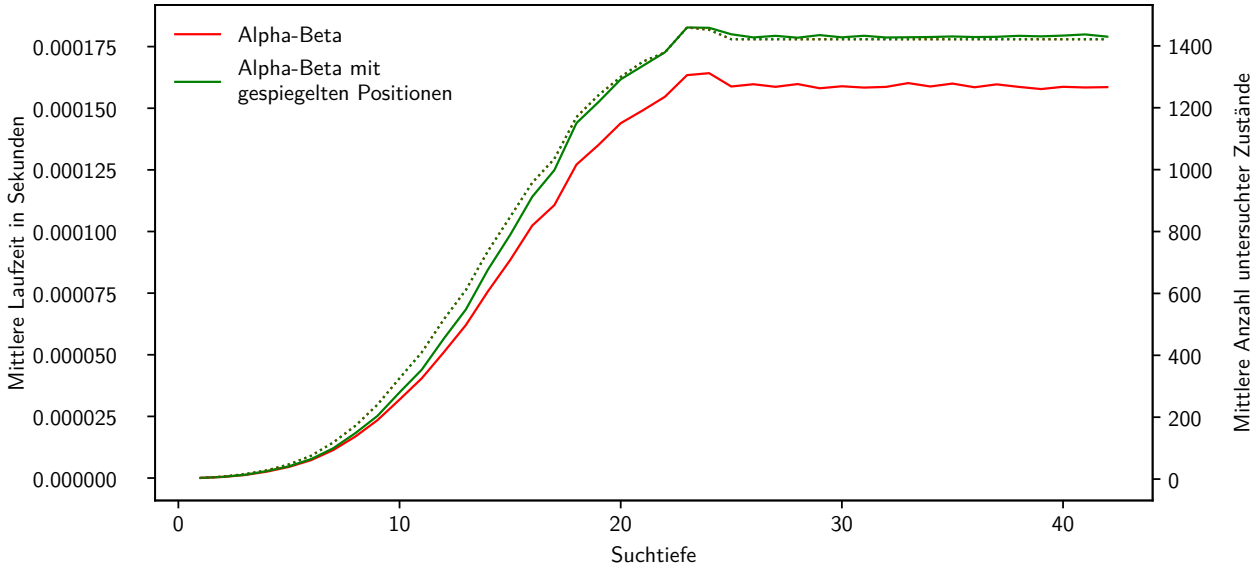

Abbildung 5.6: Vergleich der Alpha-Beta-Suche mit der Version, die gespiegelte Positionen beachtet auf dem Datensatz L2_R1

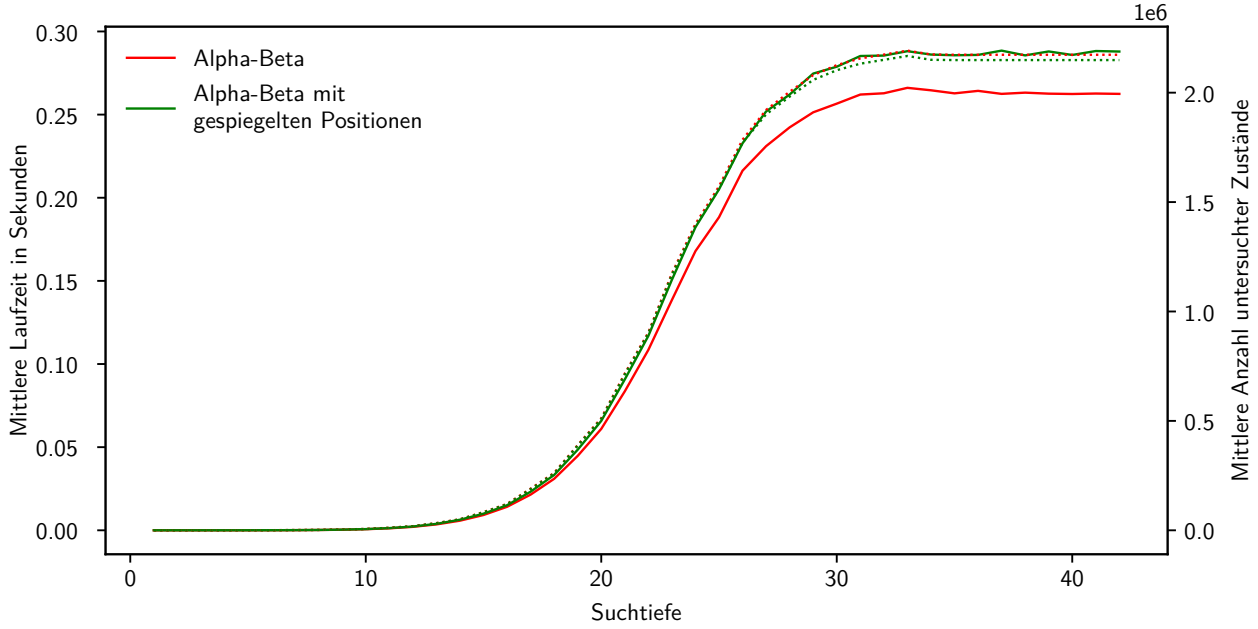

Abbildung 5.7: Vergleich der Alpha-Beta-Suche mit der Version, die gespiegelte Positionen beachtet auf dem Datensatz L1_R2

5.4 Ergebnisse nach Optimierungen

Mit allen vorgenommenen Verbesserungen aus Abb. 5.4 lässt sich nun der Vergleich aus Abb. 5.3 erneut durchführen. Dieser ist wie folgt in Abb. 5.8 aufgeführt.

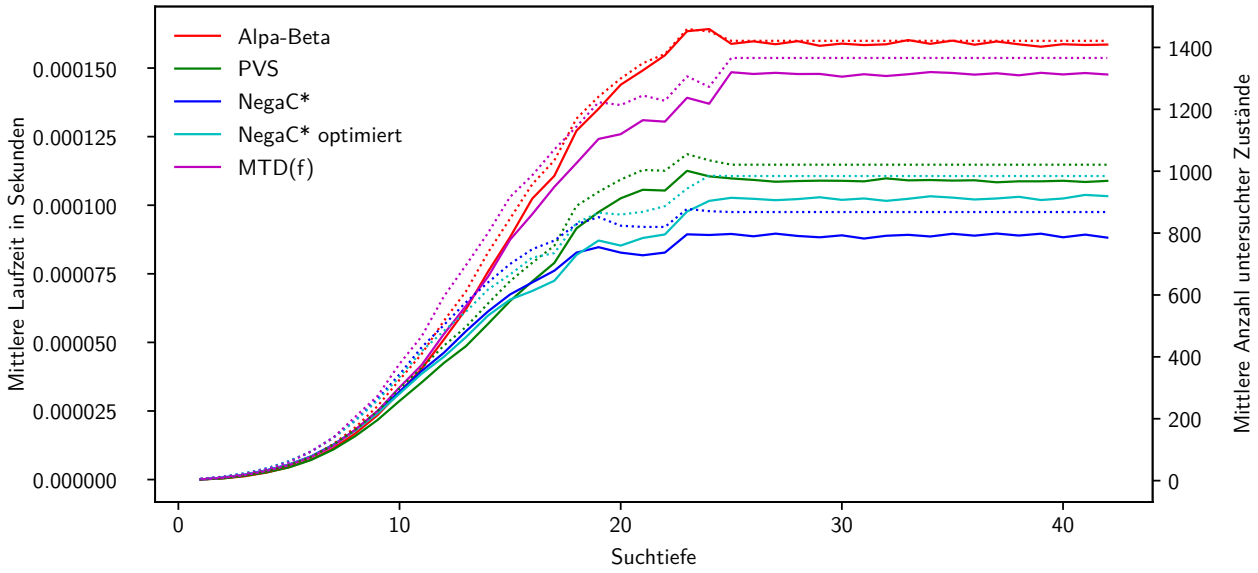

Abbildung 5.8: Vergleich aller Algorithmen mit Verbesserungen auf dem Datensatz L2_R1

Noch immer ist die Alpha-Beta-Suche die Variante, die sowohl am meisten Zustände untersucht als auch die größte Laufzeit aufweist. Allerdings untersucht MTD(f) nur noch knapp 4% weniger Zustände. Die anderen drei Algorithmen können sich wie zuvor gegen MTD(f) durchsetzen, allerdings sind die Unterschiede zwischen ihnen nun eindeutiger wahrnehmbar. Die *Principal-Variation*-Suche untersucht nun 28% weniger Zustände, als die Alpha-Beta-Suche, die optimierte NegaC*-Suche 31% und die unveränderte NegaC*-Suche 39%. Analoge Veränderungen sind auch für die Laufzeit wahrnehmbar. Mit den vorgenommenen Verbesserungen lassen sich die Algorithmen nun auch auf komplexeren Datensätzen vergleichen. Abb. 5.9 zeigt einen solchen Vergleich auf dem Datensatz L1_R2.

Hier ist der Abstand zwischen Alpha-Beta-Suche und den anderen Algorithmen wieder gestiegen. MTD(f) untersucht nun 21% weniger Zustände, beide Versionen der NegaC*-Suche ungefähr 26% und die *Principal-Variation*-Suche 30%, die Laufzeiten verhalten sich analog. Es scheint also, als wäre die *Principal-Variation*-Suche auf Positionen früher Spielphasen effizienter.

Zu diesem Zeitpunkt lässt sich auch die letzte Verbesserung hinzufügen. Hierdurch wird die Größe des Suchfensters, wie in Abschnitt 4.1.5 beschrieben, gestaucht, allerdings fällt dabei die Möglichkeit eines Tiefenlimits weg. Alle folgenden Balkendiagramme stellen daher Anzahl untersuchter Zustände und Laufzeiten nur für die maximale Suchtiefe dar. Die schwarzen Balken geben die Laufzeit und die weißen die Anzahl untersuchter Zustände an. Die roten Fehlerbalken geben die Standardabweichung der Laufzeit für die verschiedenen Positionen an. Abb. 5.10 zeigt einen Vergleich der Algorithmen mit dem schmaleren Suchfenster für den Datensatz L1_R2.

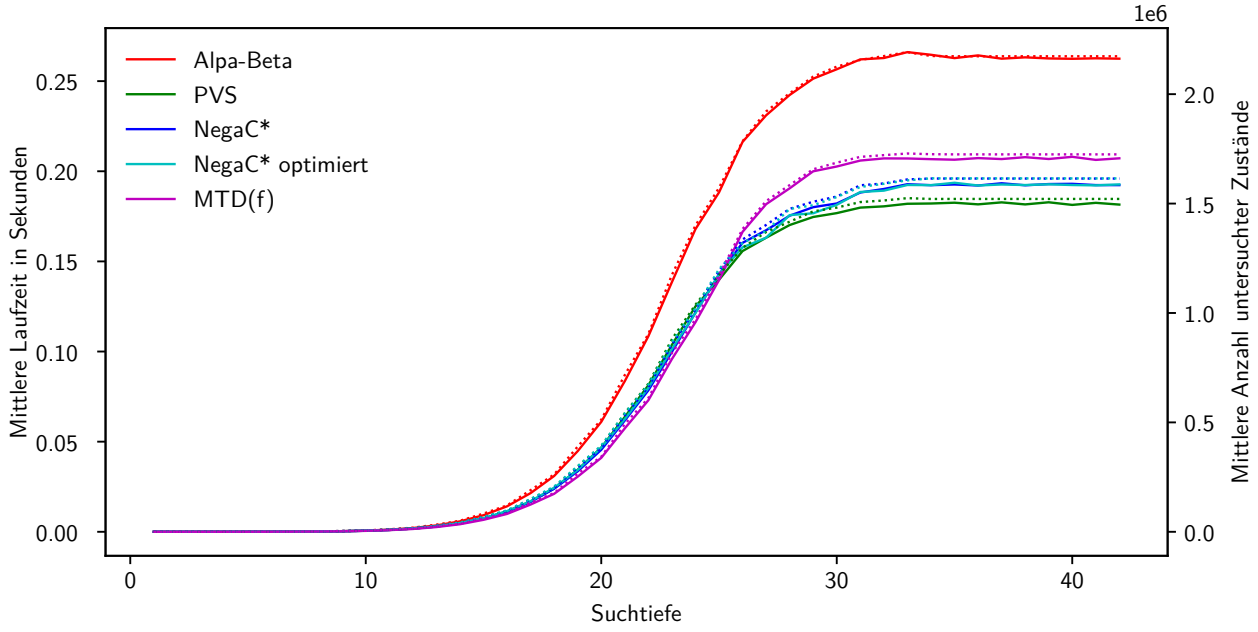

Abbildung 5.9: Vergleich aller Algorithmen mit Verbesserungen auf dem Datensatz L1_R2

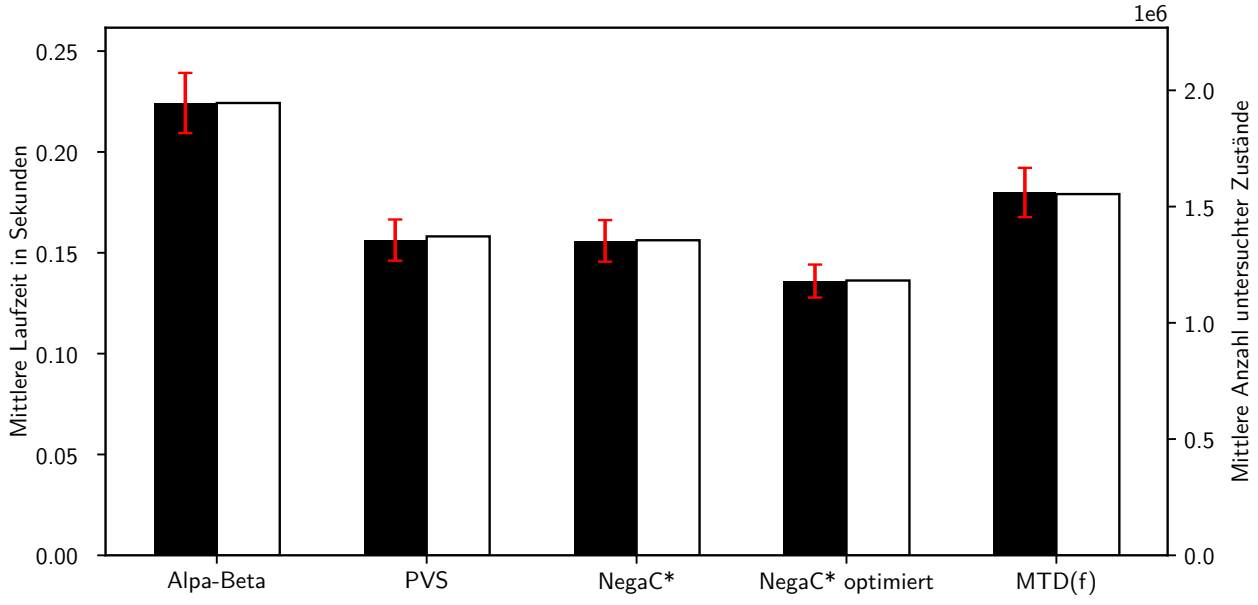

Abbildung 5.10: Vergleich aller Algorithmen mit Verbesserungen und kleinerem Suchfenster auf dem Datensatz L1_R2

Gegenüber des vorigen Graphen, untersucht die Alpha-Beta-Suche nun 10% weniger Zustände als zuvor, MTD(f) 9%, die *Principal-Variation*-Suche ebenfalls 10%, die NegaC*-Suche 16% und die

optimierte Version der NegaC*-Suche 27%. Damit untersucht MTD(f) jetzt 20% weniger Zustände, als die Alpha-Beta-Suche, die *Principal-Variation*- und NegaC*-Suche knapp 30%, wobei die NegaC*-Suche geringfügig effizienter ist, und die optimierte Version der NegaC*-Suche 39%. Alle Algorithmen arbeiten also deutlich effizienter, wenn das Suchfenster kleiner wird, am meisten profitiert jedoch die NegaC*-Suche. Da die Funktionsweise der NegaC*-Suche sehr ähnlich zu eine binären Suche ist, steht auch die Anzahl benötigter Nullfenstersuchen stark im Zusammenhang zur Größe des Suchfensters. Hinzu kommt, dass selbst die grundlegende Alpha-Beta-Suche von einem kleineren Fenster profitiert.

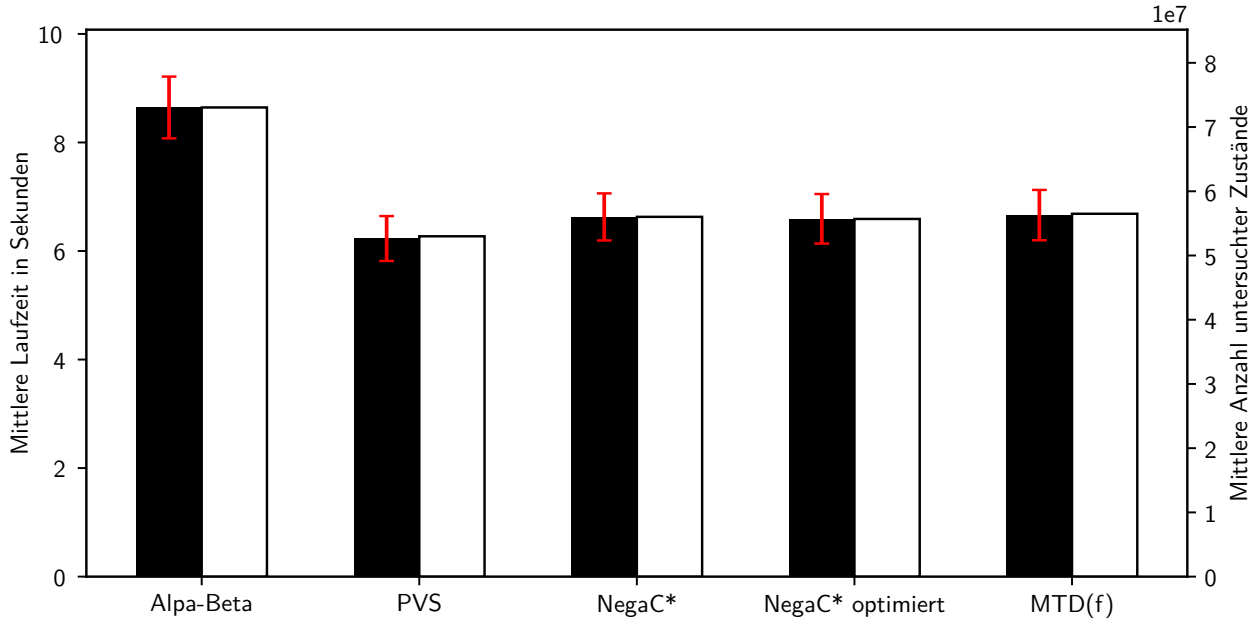

Abbildung 5.11: Vergleich aller Algorithmen mit Verbesserungen und kleinerem Suchfenster auf dem Datensatz L1_R3

Schaut man sich in Abb. 5.11 denselben Vergleich nun auf dem komplexesten Datensatz L1_R3 an, fällt auf, dass die *Principal-Variation*-Suche nun wieder die effizienteste Suche darstellt. Die anderen drei optimierten Variante untersuchen alle ähnlich viele Zustände, jedoch nicht viel mehr als die *Principal-Variation*-Suche. Es ist also wieder vermutbar, dass die *Principal-Variation*-Suche effizienter auf Positionen früher Spielphasen ist.

Abb. 5.12 zeigt denselben Vergleich, aber diesmal nicht auf einem Datensatz von Positionen, sondern auf der Startposition. Diesmal sind beide Varianten der NegaC*-Suche kaum effizienter als die Alpha-Beta-Suche. MTD(f) und die *Principal-Variation*-Suche hingegen erzielen beide eine Effizienzverbesserung von rund 15%, wobei MTD(f) diesmal geringfügig schneller ist. Die Möglichkeit, dass MTD(f) auf Positionen sehr früher Spielphasen am schnellsten ist besteht zwar, jedoch ist es hier wahrscheinlicher, dass die Ursache woanders liegt. In allen Tests wurde für MTD(f) als Startwert $f = 0$ gewählt. Dieser liegt bereits sehr nahe an dem tatsächlichen Minimax-Wert der Startposition von 1, daher benötigt MTD(f) nur verhältnismäßig wenige Nullfenstersuchen, um das Ergebnis zu bestimmen.

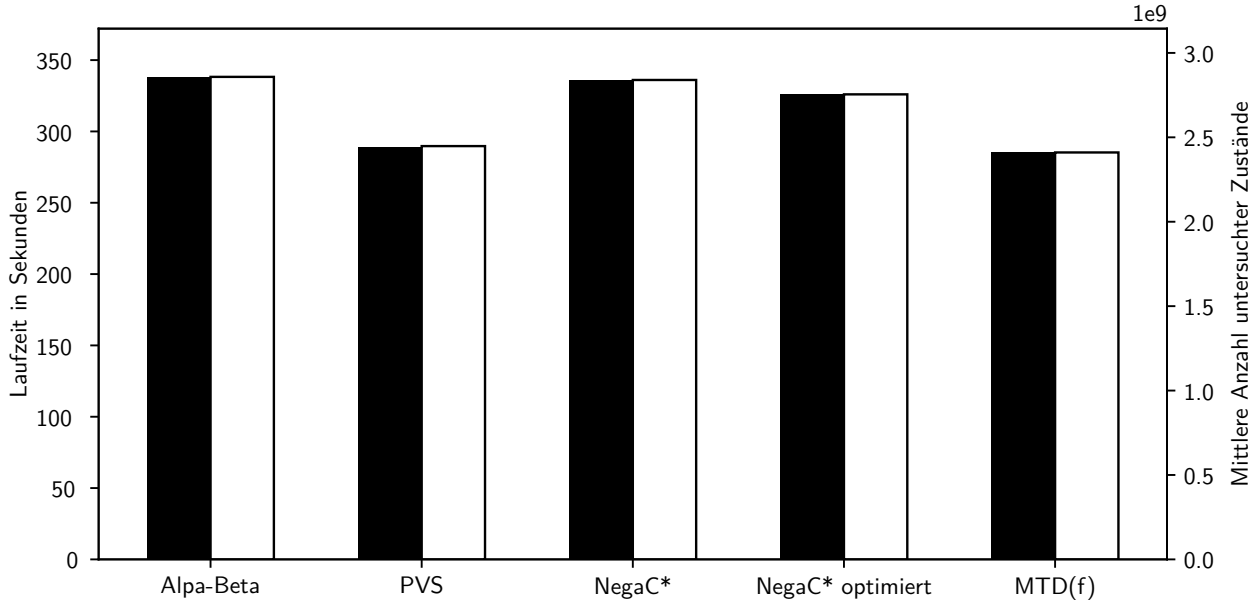

Abbildung 5.12: Vergleich aller Algorithmen mit Verbesserungen und kleinerem Suchfenster auf der Startposition

Nun lässt sich die Auswirkung durch Lesen von gespiegelten Positionen aus der *Transposition-Table* noch einmal auf den komplexeren Positionen testen. In Abb. 5.13 und Abb. 5.14 sind solche Vergleiche für den Datensatz L1_R3 und die Startposition dargestellt. Im Vergleich zu Abb. 5.11 untersuchen alle Algorithmen etwa 2% weniger Zustände, benötigen aber zwischen 8% und 10% mehr Laufzeit. Auch hier scheint der Einfluss also noch immer eher negativ auszufallen.

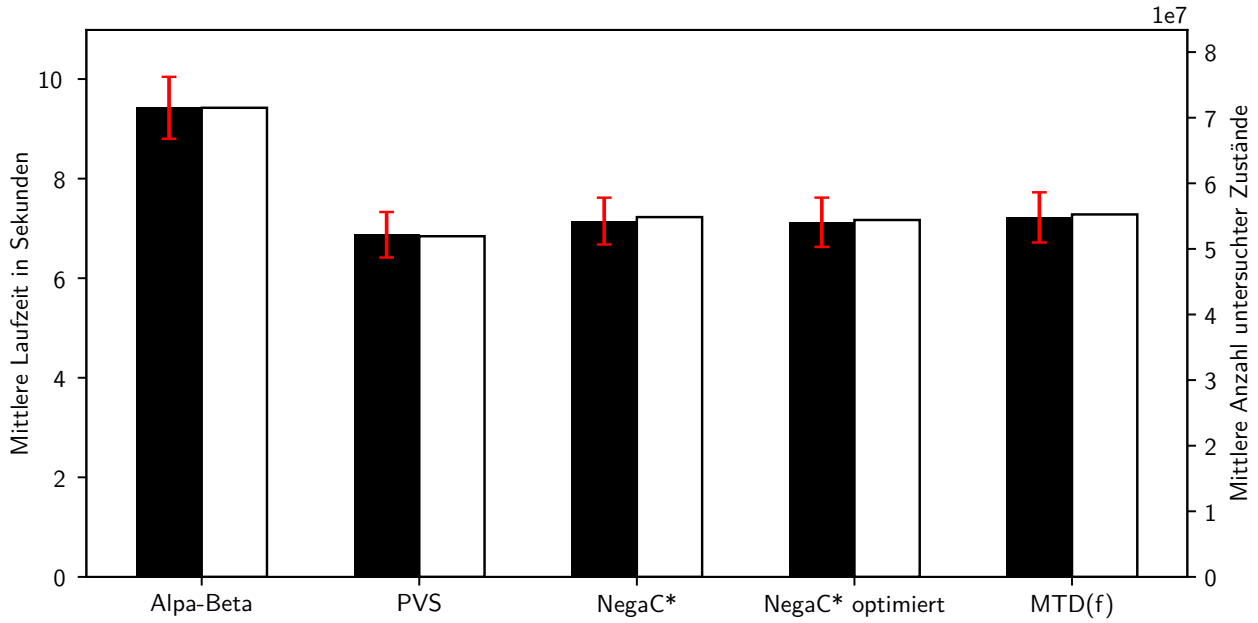

Abbildung 5.13: Vergleich aller Algorithmen mit Verbesserungen, kleinerem Suchfenster und Beachtung von gespiegelten Positionen auf dem Datensatz L1_R3

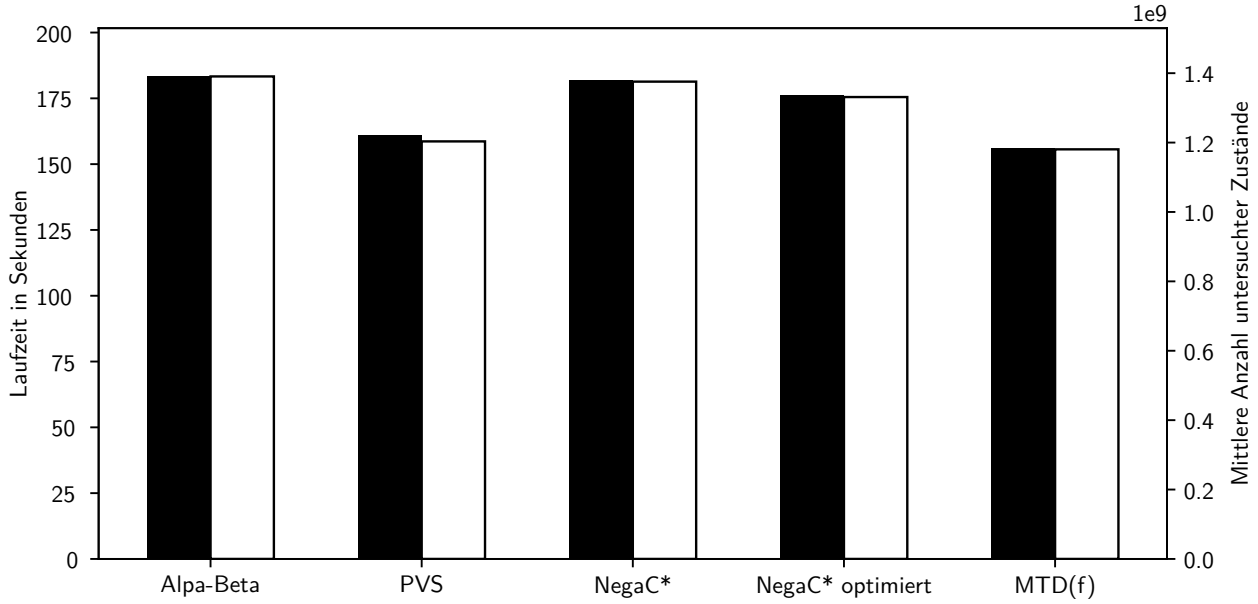

Abbildung 5.14: Vergleich aller Algorithmen mit Verbesserungen, kleinerem Suchfenster und Beachtung von gespiegelten Positionen auf der Startposition

In der Startposition sieht das allerdings anders aus. Hier werden, im Vergleich zu vorher, bei allen Algorithmen nur noch knapp 49% der Zustände untersucht. Auch an der Laufzeit sind signifikante Verbesserungen messbar. Alle Algorithmen benötigen nur noch 54% - 56% der ursprünglichen Laufzeit. Wie erwartet trägt die Symmetrie hier also zu einer deutlichen Reduktion des Suchraumes und

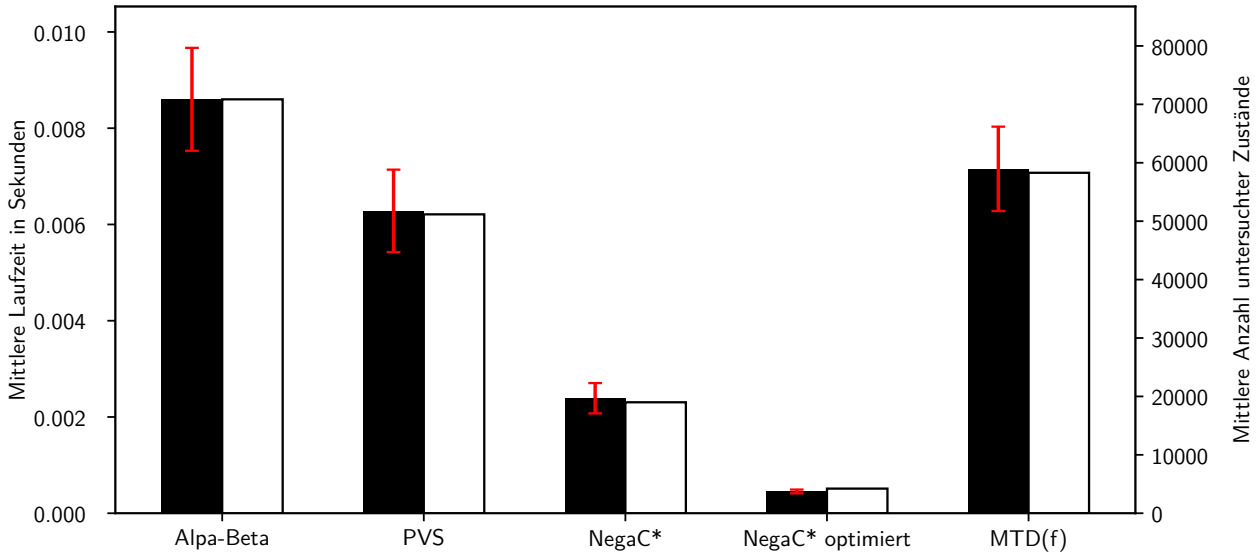

Abbildung 5.15: Vergleich aller Algorithmen mit Verbesserungen und kleinerem Suchfenster auf dem Datensatz L1_R1

somit auch der Laufzeit bei. Interessant ist hierbei, das jetzt gesamte Spiel in unter drei Minuten gelöst werden kann.

Zu diesem Zeitpunkt scheint es so, als wäre die *Principal-Variation*-Suche für Positionen früher Spielphasen, also solchen mit hohem Verzweigungsgrad am Schnellsten. Abb. 5.15 zeigt einen Vergleich für den Datensatz L1_R1. Dieser beinhaltet Positionen einer frühen Spielphase, allerdings kann die *Principal-Variation* hier schon früh gefunden werden. Sie führen bei perfektem Spiel also zu einem frühen Sieg. Im hier dargestellten Vergleich untersucht MTD(f) ungefähr 82% der Zustände der Alpha-Beta-Suche und die *Principal-Variation*-Suche nur 71%. Beide Versionen der NegaC*-Suche stechen hier aber besonders heraus. Die nicht optimierte Version untersucht hier nur 27% und die optimierte sogar nur 6% der Zustände. Dementsprechend scheint die *Principal-Variation*-Suche nicht grundsätzlich für Positionen früher Spielphasen am besten zu sein, sondern nur für solche, die eine tiefe Suche erfordern. Dies sind auch die Positionen, deren Bewertung einen niedrigen Absolutbetrag haben. Auf Positionen mit besonders hohen, bzw. niedrigen Bewertungen kann die NegaC*-Suche deutlich schneller zum Ergebnis kommen, wobei die optimierte Version diesen Aspekt noch einmal verstärkt.

6 Fazit

Betrachtet man alle Optimierungen wird deutlich, dass alle der in Abschnitt 4.1 erwähnten externen Optimierungen eine sehr positive Auswirkung auf sowohl Laufzeit, als auch Suchraumgröße haben. In den meisten Fällen scheint eine *Fail-Soft*-Implementierung einer Hybrid-, und somit auch *Fail-Soft*-Variante, überlegen zu sein. Die Verwendung von *Iterative Deepening* hingegen bringt starke Einbußen mit sich. Die Betrachtung gespiegelter Positionen beim Lesen aus der *Transposition-Table* kostet etwas mehr Laufzeit und bringt in den meisten Positionen nur eine geringfügige bis minimale Suchraumreduktion. Soll die Suche allerdings für spezielle symmetrische Positionen, wie zum Beispiel die Startpositionen, genutzt werden, werden sehr viel mehr Positionspaare gefunden, die zueinander spiegelverkehrt sind und große Suchraumreduktionen und Laufzeitverbesserungen erzielt. Unabhängig davon, dass einige dieser Optimierungen spezifisch für Vier-Gewinnt funktionieren ist dabei hervorzuheben, dass das Zusammenspiel verschiedener Optimierungen durchaus zu Ergebnissen führen kann, die so individuell nicht erwartbar gewesen wären. So fallen die Effekte von *Iterative Deepening* üblicherweise weniger negativ aus, teilweise sogar eher positiv, im Fall von Vier-Gewinnt allerdings nicht. Allerdings heißt das nicht pauschal, dass die Verwendung von *Iterative Deepening* im Falle von Vier-Gewinnt nie sinnvoll ist, so könnte eine andere Bewertungsfunktion zu völlig anderen Ergebnissen führen.

Zu den Algorithmen lässt sich argumentieren, dass mit allen Optimierungen implementiert wie beschrieben, die optimierte Variante der NegaC*-Suche für die meisten Positionen sowohl beste Laufzeit, als auch Suchraumgröße aufweist. Für komplexere Positionen, die eine hohe Suchtiefe erfordern, verschwinden diese Vorteile jedoch und die *Principal-Variation*-Suche setzt sich mit geringem Abstand durch. Dies ist anscheinend aber nur in den ersten Zügen des Spiels der Fall. Für ein Programm, das interaktiv gegen einen Menschen spielen soll, könnte man die *Principal-Variation*-Suche Verwenden um ein Eröffnungsbuch zu generieren, welches die Lösungen für alle in 8 Zügen erreichbaren Zustände beinhaltet. Für die übrigen Positionen käme dann die optimierte NegaC*-Suche zum Einsatz. MTD(f) ist in den meisten Fällen, nach der Alpha-Beta-Suche, die schlechteste aller Varianten. Im Gegensatz zu den anderen Algorithmen ist sie allerdings in der Lage die Suche mit einer anfänglichen Schätzung zu beginnen. Dementsprechend eignet sie sich besonders in Fällen, in denen schon eine recht genaue Schätzung bekannt ist.

Weiterführend gibt es auch noch einige weitere Optimierungen, deren Betrachtung interessant sein könnte. So könnten aufwändigere Heuristiken und eine bessere Zugsortierung, eine effektive Umsetzung von *Iterative Deepening* ermöglichen. Außerdem könnte man eine besonders starke Heuristik implementieren, die zu Beginn der Suche benutzt wird, um einen Startwert für MTD(f) zu generieren. Interessant wäre ebenfalls eine Übergeordnete Funktion, die initial die Eigenschaften eines Zustands, wie Symmetrie, Spielphase und Verzweigungsgrad bestimmt und dann entsprechend einen passenden Suchalgorithmus auswählt.

Literaturverzeichnis

[1] James D Allen. Expert play in connect-four. *Verfügbar unter* https://web.archive.org/web/20131023004851/http://homepages.cwi.nl/~tromp/c4.html, 1990.

[2] Louis Victor Allis. A knowledge-based approach of connect-four. *J. Int. Comput. Games Assoc.*, 11(4):165, 1988.

[3] Cameron Browne. Bitboard methods for games. *ICGA journal*, 37(2):67–84, 2014.

[4] Stefan Edelkamp and Peter Kissmann. Symbolic classification of general two-player games. In Andreas R. Dengel, Karsten Berns, Thomas M. Breuel, Frank Bomarius, and Thomas R. Roth-Berghofer, editors, *KI 2008: Advances in Artificial Intelligence*, pages 185–192, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[5] John P Fishburn. Another optimization of alpha-beta search. *ACM SIGART Bulletin*, 84:37–38, 1983.

[6] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[7] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[8] T Anthony Marsland and Murray Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys (CSUR)*, 14(4):533–551, 1982.

[9] Pascal Pons. Solving connect 4: How to build a perfect ai, 2019. http://blog.gamesolver.org/, zuletzt abgerufen am 22.01.2024.

[10] Aske Plaat. Mtd (f), a minimax algorithm faster than negascout. *arXiv preprint arXiv:1404.1511*, 2014.

[11] Alexander Reinefeld. *Spielbaum-Suchverfahren*, volume 200. Springer-Verlag, 2013.

[12] Alexander Reinefeld and T. Anthony Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[13] Stuart J Russell and Peter Norvig. *Artificial intelligence a modern approach*. London, 2010.

[14] George C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[15] Jean-Christophe Weill. Experiments with the negac* search an alternative for othello endgame search. *Heuristic Programming in Artificial Intelligence, the second computer olympiad*, pages 174–188, 1991.