

3 Essenzielle Komponenten

3.1 Datenstrukturen

3.1.1 Bitboards

Bitboards sind ein Spezialfall von Bitfields, einem vorzeichenlosen Integer von variabler Länge, bei dem die einzelnen Bits als unabhängige Werte, z.B. Booleans, betrachtet werden. Im Fall von Bitboards sind diese 64 Bit lang, was der Anzahl an Feldern auf einem Schachbrett entspricht, sodass jedes einzelne Bit einem Boolean entspricht, das z.B. aussagt, ob dort eine Figur steht.

Wie in der Schachprogrammierung üblich, haben wir je ein Bitboard pro Spielfigur und je eines pro Farbe verwendet, also insgesamt 7, da Racing Kings ohne Bauern gespielt wird. Durch diese Bitboards kann der ganze Spielzustand effektiv dargestellt werden, da nur die eben erklärten 7 Bitboards und ein Integer zur Adressierung benötigt wird.

Eine alternative Darstellung wäre die Nutzung von Arrays gewesen, die vor Allem im Zugriff aber deutlich ineffizienter sind. Bitboards unterstützen als Integer bitweise Operationen wie z.B. AND, OR, XOR sowie auch Spezialinstruktionen wie CLZ, CTZ und POPCNT.

Durch die Nutzung von Bitboards können also Operationen auf dem gesamten Schachbrett in eine einzige Prozessor-Instruktion gebündelt werden, ähnlich den modernen SIMD-Vektorinstruktionen, welche aber auch auf simpleren Prozessoren möglich sind.

3.2 Tree Search

Alle angewandten Techniken basieren auf einem Spezialfall des Suchbaumes, dem Spielbaum, dieser stellt den Verlauf eines strategischen, vollständig beobachtbaren, statischen, zwei Spieler Spiels dar. In einem Spielbaum entspricht jeder Knoten einer Ebene dem Zug ein- und desselben Spielers. Diese Züge erhalten mit einer Bewertungsfunktion einen Score, anhand dessen der optimale Zug ausgewählt wird.

3.2.1 Minimax

Der MiniMax-Algorithmus bildet die Grundlage für alle nachfolgenden Techniken und Optimierungen. Er basiert auf der Grundidee, den minimal zu erwartenden Gewinn zu maximieren. Dazu werden die im Spielbaum vorkommenden Ebenen nach Min- und Maxebenen eingeteilt. Maxebenen stehen für Züge des Spielers, aus dessen Sicht der Baum aufgebaut ist, siehe Abb 1.

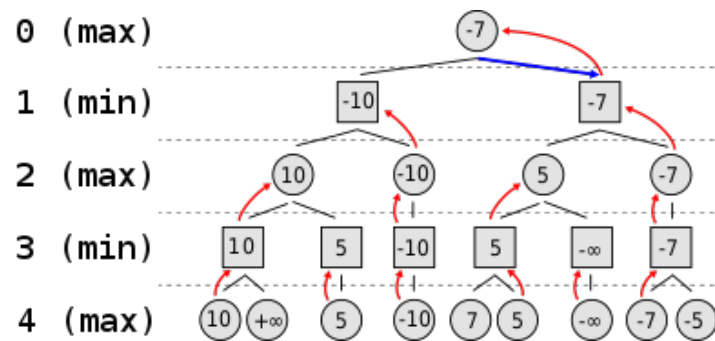


Abbildung 1: Darstellung des Minimax-Algorithmus

In jeder Minebene wird der maximale Zug aus der Ebene davor ausgewählt und anders herum in der Maxebene, so wird der beste Zug durch den Baum nach oben an die Wurzel zurück gereicht.

3.2.2 α - β -Algorithmus

Der α - β -Algorithmus ist eine Erweiterung des Minimax-Algorithmus. Hier ist die Idee, dass die Suche in einem Teilbaum abgebrochen werden kann, wenn klar ist, dass dieser von der darüberliegenden Ebene nicht gewählt werden würde. Dies ist z.B. bei besonders guten Zügen der Fall. Das verkürzt die Laufzeit, da weniger Spielsituationen durchsucht werden müssen. Da unsere Bewertungsfunktion symmetrisch funktioniert, haben wir uns für eine Negamax Implementation entschieden, um den Code kurz zu halten.

3.2.3 Zeitmanagement und iterative Tiefensuche

Bei jedem Aufruf des α - β -Algorithmus wird ihm ein Zeitkonto gegeben. Wenn der Algorithmus dieses überschreitet, dann gibt er den letzten gültigen Wert der iterativen Tiefensuche zurück. Das Zeitkonto wird abhängig von der verbleibenden Zeit der KI gewählt. Folgende Tabelle beschreibt die von uns getroffenen Einteilungen:

Übrige Zeit in Sekunden	< 30	≥ 30	≥ 10	≥ 5
Zeitkonto in Millisekunden	5000	2000	500	100

Da die letzte Iteration der Tiefensuche den Großteil der Zeit beansprucht, geht bei dieser Variante des Zeitmanagements viel Zeit verloren. Damit das nicht passiert, berechnet der Algorithmus in jeder Iteration der Tiefensuche den durchschnittlichen Verzweigungsfaktor (Anzahl an möglichen Bewegungen) einer Spielsituation. Multipliziert man das mit der Zeit für die letzte

Iteration, ergibt das eine mögliche Einschätzung der für die nächste Iteration benötigten Zeit. Diese Einschätzung wird nun verwendet, um zu überprüfen, ob weitere Berechnungen innerhalb des Zeitkontos sinnvoll sind.

3.3 Bewertungsfunktion

Die Bewertungsfunktion ist grundsätzlich das Gehirn der KI. Ohne eine gute Bewertungsfunktion kann durch den Rest des Programms eine beliebige Tiefe erreicht werden und es würden dennoch nicht die optimalsten Züge gewählt werden. Daher ist es wichtig, eine gute Bewertungsfunktion zu schreiben, die alle wichtigen Positionen und Ziele des Spiels wiedergibt.

Die Bewertungsfunktion berechnet ein Score als Integer, der im Suchalgorithmus genutzt wird, um den besten Zug zu ermitteln. Dabei repräsentiert ein positiver Score den Vorteil und ein negativer Score den Nachteil des Spielers. Je größer / kleiner der Score ist, desto höher ist die Gewinn- / Verlierchance. Die Funktion berechnet nur einen Score aus der gegebenen Spielsituation, darum muss der Score in jede Iteration im Suchalgorithmus erneut berechnet werden.

Aufgrund der hohen Rechenkomplexität im Suchalgorithmus wollten wir die Bewertungsfunktion einfach und effizient halten. Sie soll einerseits weniger rechen- und speicherintensiv sein und andererseits die aktuelle Spielsituation solide bewerten. Um dies zu erreichen, berechnet sich der Score unsere Bewertungsfunktion (EV), die sich aus zwei Hauptkomponenten zusammensetzt, wie folgt:

$$EV = \left(\left(\sum_{i=0}^n MV(p_i) + PST(gs, p_i) \right) + BP \right) \cdot pl$$

wobei:

$MV(p)$	=	Material Value
$PST(gs, p)$	=	Piece Square Table
BP	=	Bishop Pair
pl	=	Player { 1, -1 }
p	=	Piece
gs	=	Game State

3.3.1 Material Value

Jede Figur besitzt einen Basiswert (MV), die ihre strategische Stärke und Möglichkeiten widerspiegeln, solange sie auf dem Spielfeld ist. Im folgenden

werden die Werte für weiße Figuren betrachtet. Für schwarze Figuren sind die Eigenschaften der Figuren identisch und die Werte sind in negierter Form. Die Werteinteilung haben wir wie folgt begründet:

- Der König ist die zentrale Figur im Spiel. Er ist die einzige Figur, die nicht geschlagen werden kann. So kann man ihm jede beliebige Zahl zuordnen, da immer zwei Könige auf dem Spielfeld sind. Wir haben uns für 0 entschieden.
- Der Springer hat den Vorteil, dass er über andere Figuren hinwegspringen kann. Das macht ihn vor allem im Startzustand des Spiels zu einer gefährlichen Figur. Diese Fähigkeit des Springens gleicht sich jedoch mit seiner begrenzten Reichweite aus. Im fortgeschrittenen Spielverlauf wird dieser Nachteil deutlich, da er alleine den König nicht blocken kann. Darum besitzt er ein *MV* von +20.
- Der Läufer kann sich diagonal in alle Richtungen beliebig weit auf freie Felder bewegen und bleibt dabei immer auf den Feldern einer Farbe. Wie der Springer auch, reicht ein Läufer nicht aus, um einen König daran zu hindern in den nächsten Rang aufzusteigen. Jedoch sind zwei Läufer in der Lage, eine Blockade für den gegnerischen König zu bilden, da beide zusammen genügend Felder abdecken. Darum vergeben wir bei *BP* einen Bonus von +10 auf den Score, wenn zwei Läufer auf dem Spielfeld sind, ansonsten 0. Sein *MV* beträgt +20.
- Den Turm kann man horizontal und vertikal ziehen. Vor allem die horizontale Bewegung macht diese Figur so wertvoll, weil sie damit alleine den Aufstieg des Königs verhindern kann, solange keine Figur dazwischen steht. Mit seiner vertikalen Bewegung kann er den Rang begrenzen, den ein König erreichen kann. Diese Eigenschaften machen ihn flexibel und strategisch wertvoll. So haben wir uns für ein *MV* von +50 entschieden.
- Die Dame ist die wertvollste Figur im Spiel. Man kann sie wie einen Turm horizontal und vertikal und wie einen Läufer diagonal ziehen. Sie erbt damit alle Eigenschaften vom Turm und Läufer. So wie ein Turm reicht diese Figur aus, um eine effektive Blockade für den gegnerischen König zu errichten. Ihre *MV* setzt sich daher aus der Summe der *MV* des Turms und der *MV* des Läufers zusammen.

Die *MV* alleine können schon bei der Wahl eines Zuges helfen, da bei einem Verlust einer Figur der Score sich gravierend ins Positive oder Negative ändert. Jedoch bringt dieser Wert bei nicht-schlagenden Zügen und

bei der Positionierung der Figuren für den Score nichts, da solche Züge keine Änderung im Score bewirken würden (± 0). Um solche Züge abzudecken, verwenden wir sog. „Piece Square Tables“.






				
$MV(q) = 70$	$MV(r) = 50$	$MV(b) = 20$	$MV(n) = 20$	$MV(k) = 0$

Tabelle 2: Material Value

3.3.2 Piece Square Table

Jede Figur besitzt eine Piece Square Table *PST* mit verschiedene Werten in jedem Feld. Durch *PST* lassen sich die Positionen der Figuren bewerten. Dabei wird ein Wert auf ihre *MV* addiert, basierend auf ihrer Position in der entsprechenden *PST*. Zum Beispiel ist ein Springer mehr wert im Zentrum als an seiner Startposition. Das Ziel ist es, die Figuren, bei nicht-schlagenden Zügen, durch ihre *PST* strategisch zu positionieren. Nur die *PST* des Königs ist so modelliert, dass er sich in den nächsten Rang bewegen soll. Je höher sein Rang, desto höher ist sein Wert und somit die Gewinnchance.

Um das Gleichgewicht zwischen Schlagzügen, Blockieren des Gegners und Vorrücken des eignen Königs zu gewährleisten, wurden die *PST* für drei Spielzustände definiert, diese sind Early, Mid und End Game. Die folgenden *PST* zeigen die Werte für die weißen Figuren. Um die *PST* für die schwarzen Figuren zu erhalten, müssen die *PST* Werte mit -1 multipliziert werden.

3.3.3 Early Game

Das Early Game fokussiert sich auf die strategische Entwicklung der Dame, Türme, Läufer und Springer. In dieser Phase sollen sich diese Figuren aus ihrer Startposition lösen und nach ihrer *PST* positionieren. Das Ziel ist es, viele gegnerische Figuren zu schlagen, um einen langfristigen Vorteil zu erlangen. Die Figuren verfolgen dabei verschiedene Strategien in ihrer *PST* (siehe Abbildung 2):

- Der König soll sich in den nächsten Rang bewegen. Dabei soll der Rand gemieden werden, da der König dort weniger Ausweichmöglichkeiten hat. So muss der Gegner mehr Felder abdecken, um den König am

Bewegen zu hindern. Wir bewerten deswegen Züge in der Mitte des Spielfelds besser.

- Generell wollen wir den Springer ins Zentrum stellen, um seine Bewegungsmöglichkeiten zu maximieren. Er ist die einzige Figur, die eine generische Figur aus seinem Startzustand schlagen kann. Darum sind seine *PST*-Werte kleiner als ein schlagender Zug (mindestens +20), aber größer als nicht-schlagende Züge der anderen Figuren (maximal +18). Somit ist sichergestellt, dass er auch im Early Game aus dem Startzustand gegnerische Figuren schlagen kann.
- Die Läufer sollen sich in den unteren Rängen aufhalten. Speziell im Late Game wird sich diese Positionierung mit zwei Läufern als Vorteilhaft herausstellen, da sie damit den gegnerischen König in höheren Rängen über die Diagonale vollständig blockieren können.
- Die Türme sollen sich auf der Seite des Königs in höhere Ränge bewegen. Das soll unter anderem den Rang des gegnerischen Königs früh eingrenzen, einen Tradeoff erzwingen und der Dame aus ihrer Startposition helfen.
- Die Dame soll sich im Zentrum aufhalten. Wie der Springer hat sie hier die meisten Bewegungsmöglichkeiten und kann am besten auf gegnerische Züge reagieren. Ein weiterer Vorteil ist die Kontrolle des gegnerischen Königs.

Um eine positive Bilanz zu erreichen, ergibt sich durch die vordefinierten *PST*s und *MV*s folgende Priorität:

Figuren schlagen > Figur außer König bewegen > König bewegen

Es lohnt sich immer, eine Figur zu schlagen, da man einen Score von mindestens +20, zum Beispiel beim Springer schlagen, erreicht. Für nicht-schlagende Züge gilt nach *PST* die Reihenfolge: Springer, Läufer, Turm, Dame und König. Wir positionieren also zuerst die minderwertigen Figuren und versuchen damit gegnerische Figuren zu schlagen, da der eventuell darauf folgende Tradeoff ≥ 0 ist. Königsbewegungen (maximal +10) lohnen sich kaum, weil jeder andere Zug mehr zum Score beiträgt.

Das Early Game endet, wenn nur noch 10 Figuren auf dem Spielfeld sind und geht in das Mid Game über. Das Mid Game wird auch automatisch nach der achten Runde erzwungen, falls die erste Bedingung nicht zutrifft.






		
$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 11 & 11 & 11 & 11 & 11 & 11 & 6 \\ 6 & 11 & 16 & 16 & 16 & 16 & 11 & 6 \\ 6 & 11 & 16 & 16 & 16 & 16 & 11 & 6 \\ 6 & 11 & 11 & 11 & 11 & 11 & 11 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\ 7 & 17 & 17 & 12 & 12 & 17 & 17 & 7 \\ 7 & 17 & 17 & 12 & 12 & 17 & 17 & 7 \\ 7 & 17 & 17 & 12 & 12 & 17 & 17 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 8 & 18 & 18 & 13 & 13 & 18 & 18 & 8 \\ 8 & 18 & 18 & 13 & 13 & 18 & 18 & 8 \\ 8 & 18 & 18 & 13 & 13 & 18 & 18 & 8 \\ 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
		
$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 14 & 14 & 9 & 0 & 0 \\ 0 & 9 & 14 & 14 & 14 & 14 & 9 & 0 \\ 0 & 14 & 14 & 19 & 19 & 14 & 14 & 0 \\ 0 & 14 & 14 & 19 & 19 & 14 & 14 & 0 \\ 0 & 9 & 14 & 14 & 14 & 14 & 9 & 0 \\ 0 & 0 & 9 & 14 & 14 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 40 & 45 & 45 & 45 & 45 & 45 & 45 & 40 \\ 30 & 35 & 35 & 35 & 35 & 35 & 35 & 30 \\ 20 & 25 & 25 & 25 & 25 & 25 & 25 & 20 \\ 10 & 15 & 15 & 15 & 15 & 15 & 15 & 10 \\ 5 & 10 & 10 & 10 & 10 & 10 & 10 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	

Abbildung 2: Piece Square Table (Early Game)

3.3.4 Mid Game

Im Mid Game wollen wir anfangen, den König zu entwickeln. Die *PST* des Königs wird deswegen erhöht, sodass es sich lohnt, den König in den nächsten Rang zu bewegen. Da Positionierung von Springer, Läufer, Turm und Dame im Mid Game sehr stark vom Early Game abhängt, werden ihre gewünschte Positionen im *PST* (der höchste Wert) auf relative Positionen (der zweithöchste Wert) geändert. Dame und Türme haben eine bevorzugte Positionierung in höheren Rängen, um sowohl den gegnerischen König zu blocken, als auch den eigenen König zu unterstützen (siehe Abbildung 3). Ansonsten verfolgen sie dieselbe Strategie wie im Early Game. Es entsteht folgende Priorität:

Figuren schlagen > generischen König blockieren
 ≥ König bewegen > Figur außer König bewegen

Auch im Mid Game wollen wir gegnerische Figuren schlagen, daher trägt dies immer noch am meisten für den Score bei. Durch die angepassten *PSTs* der Figuren verschiebt sich die Bewegungspriorität: König, Springer, Läufer, Turm und Dame. Eine Bewegung vom König in den nächsten Rang (mindestens +15) wird nun bevorzugt vor einer Bewegung von Springer (höchstens +14), Läufer (höchstens +13), Turm (höchstens +12) und Dame (höchstens +11), aber weniger als Figuren schlagen (mindestens +20). Somit wird bei

einem nicht-schlagenden Zug eine Königsbewegung bevorzugt. Da der generische König in dieser Phase den letzten Rang erreichen kann, wird indirekt versucht den gegnerischen König zu blockieren. Das passiert, wenn unsere Baumsuche auf einem Blatt endet, an dem der Gegner das Spiel beenden kann und somit den Score $-\infty$ erreicht.

Das Mid Game endet nach Runde 16 oder wenn nur noch 6 Figuren auf dem Spielfeld sind und startet dann das End Game.






		
$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 11 & 11 & 11 & 11 & 11 & 11 & 6 \\ 6 & 11 & 11 & 11 & 11 & 11 & 11 & 6 \\ 6 & 11 & 11 & 11 & 11 & 11 & 11 & 6 \\ 6 & 11 & 11 & 11 & 11 & 11 & 11 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\ 7 & 12 & 12 & 12 & 12 & 12 & 12 & 7 \\ 7 & 12 & 12 & 12 & 12 & 12 & 12 & 7 \\ 7 & 12 & 12 & 12 & 12 & 12 & 12 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 8 & 13 & 13 & 13 & 13 & 13 & 13 & 8 \\ 8 & 13 & 13 & 13 & 13 & 13 & 13 & 8 \\ 8 & 13 & 13 & 13 & 13 & 13 & 13 & 8 \\ 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
		
$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 14 & 14 & 9 & 0 & 0 \\ 0 & 9 & 14 & 14 & 14 & 14 & 9 & 0 \\ 0 & 14 & 14 & 14 & 14 & 14 & 14 & 0 \\ 0 & 14 & 14 & 14 & 14 & 14 & 14 & 0 \\ 0 & 9 & 14 & 14 & 14 & 14 & 9 & 0 \\ 0 & 0 & 9 & 14 & 14 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 75 & 85 & 85 & 85 & 85 & 85 & 85 & 75 \\ 60 & 70 & 70 & 70 & 70 & 70 & 70 & 60 \\ 45 & 55 & 55 & 55 & 55 & 55 & 55 & 45 \\ 30 & 40 & 40 & 40 & 40 & 40 & 40 & 30 \\ 15 & 25 & 25 & 25 & 25 & 25 & 25 & 15 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	

Abbildung 3: Piece Square Table (Mid Game)

3.3.5 End Game

Im End Game wollen wir das Spiel gewinnen und beenden. Das heißt, wir versuchen den König in den letzten Rang zu bewegen und gleichzeitig den gegnerischen König daran zu hindern ans Ziel zu kommen. Dafür wird die *PST* des Königs nochmal erhöht. Da die Ausgangssituation stark vom Mid Game und die Positionierung der Könige abhängt, werden Springer, Läufer, Turm und Dame in zwei Bereiche eingeteilt. Springer, Turm und Dame sollen sich in den oberen Rängen positionieren, da sie hier durch ihre Bewegungsmöglichkeiten am besten die Könige kontrollieren können. Der Läufer soll sich im unteren Bereich einordnen und weiterhin seine Strategie im Early Game verfolgen (siehe Abbildung 4). Wir wollen folgendes erreichen:

König bewegen \geq generischen König blockieren
 \geq Figuren schlagen $>$ Figuren außer König bewegen

Nach der *PST* wird der König für jeden höheren Rang mit mindestens +30 belohnt. Das ist mehr als zweimal so viel wie eine Bewegung von Springer (+14), Läufer (+13), Turm (+12) und Dame (+11). Springer und Läufer haben durch ihre Bewegungsmöglichkeiten einen Nachteil, sodass man mehrere Figuren braucht, um den gegnerischen König effektiv zu blocken. Sie werden dennoch bevorzugt, um die Blockade des eigenen Königs zu lösen. Zum Beispiel durch eine Positionierung, wodurch man generische Zugmöglichkeiten der Figuren blocken kann, sodass es möglich ist, unseren König in den nächsten Rang zu ziehen. Nur noch Turm und Dame schlagen lohnt sich mehr (mindestens +50), da nur diese Figuren alleine unseren König daran hindern könnten, sich in den nächsten Rang zu bewegen. Wie im Mid Game beschrieben wird auch hier indirekt versucht, den gegnerischen König zu blockieren.






		
$\begin{bmatrix} 11 & 11 & 11 & 11 & 11 & 11 & 11 & 11 \\ 11 & 11 & 11 & 11 & 11 & 11 & 11 & 11 \\ 11 & 11 & 11 & 11 & 11 & 11 & 11 & 11 \\ 11 & 11 & 11 & 11 & 11 & 11 & 11 & 11 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 \\ 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 \\ 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 \\ 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 \end{bmatrix}$
		
$\begin{bmatrix} 14 & 14 & 14 & 14 & 14 & 14 & 14 & 14 \\ 14 & 14 & 14 & 14 & 14 & 14 & 14 & 14 \\ 14 & 14 & 14 & 14 & 14 & 14 & 14 & 14 \\ 14 & 14 & 14 & 14 & 14 & 14 & 14 & 14 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 150 & 165 & 165 & 165 & 165 & 165 & 165 & 150 \\ 120 & 135 & 135 & 135 & 135 & 135 & 135 & 120 \\ 90 & 105 & 105 & 105 & 105 & 105 & 105 & 90 \\ 60 & 75 & 75 & 75 & 75 & 75 & 75 & 60 \\ 30 & 45 & 45 & 45 & 45 & 45 & 45 & 30 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	

Abbildung 4: Piece Square Table (End Game)

3.3.6 Konzept

Alle *MV* und *PST* für jede Figur werden in einer Struktur festgehalten. Dabei sind *MV* einfache Integer Werte und alle *PST*-Tabellen ein Array mit 64 Integer Felder, wobei jedes Feld aus der *PST* eine Position auf dem Spielfeld repräsentiert. Bevor wir den Suchalgorithmus starten, werden diese beiden Komponenten mit den oben genannten Werten gefüllt und die Struktur initialisiert. So kann unsere Bewertungsfunktion in der Suche einfach die Werte

aller Figuren aus der Struktur auslesen. Insgesamt spart uns das Rechenzeit ein, die wir dann für die Suche investieren.

3.4 Movegenerator

Der Movegenerator ist ein Kernbestandteil der KI, da er alle möglichen Züge für eine gegebene Spielsituation generiert. Da dies eine relativ komplexe Funktion ist, die für jedes Element des Suchbaums teils mehrfach durchgeführt werden muss, sind vor allem auch Effizienzverbesserungen dieser von hoher Bedeutung für eine gute KI. Denn je schneller dies geschehen kann, desto größer wird die maximale Suchtiefe, die in der gegebenen Zeit erreicht werden kann.

Der Movegenerator wurde nach einem simplen Iterator-Prinzip gebaut, welches einen jeweiligen legalen Zug erst dann generiert, wenn er abgefragt wird, anstatt alle möglichen Züge im Vorhinein zu generieren und danach im Suchbaum abzufragen. Hiervon versprach man sich erhöhte Effizienz, da ggf. Züge, die nie getestet wurden auch nicht generiert werden mussten.

3.4.1 Erste Version

Zum ersten Meilenstein war der Movegenerator nahezu komplett iterativ und prüfte von einer jeweiligen Figur aus nacheinander die Richtungen und Entfernungen von einer Figur sequenziell durch. Dies erwies sich relativ schnell als ein verhältnismäßig langsamer Ansatz, der ersetzt werden sollte.

Außerdem generierte der erste Movegenerator, um an Komplexität zu sparen teils, auch sog. non-legale Züge, da er nicht alle Regeln mitbetrachtete. So wurden z. B. Züge, die einen der Könige ins Schach setzen würden, auch generiert, obwohl sie eigentlich nicht gültig sind. Diese Prüfung musste somit später im Suchbaum stattfinden, was sich auf Dauer auch als eher ineffizient und unsauber herausstellte.

3.4.2 Optimierung mit Bitboards

Für den zweiten Meilenstein wurde der Movegenerator aufgrund der zuvor genannten Mängel von Grund auf überholt. Hauptsächlich lag der Fokus darauf, möglichst viel iterative, sequenzielle Logik durch Bitboard-Operationen (siehe 3.1) zu ersetzen. Als Inspiration und grober Leitfaden diente ein Artikel über einen Schach-Movegenerator, der Bitboards in Rust nutzt [4].

Bitboards und vorgenerierte Masken Der Großteil der gewonnenen Effizienz kommt hier daher, dass Masken der möglichen Bewegungen für ver-

schiedene Figuren vorgeneriert werden können, anstatt sie jedes Mal neu auszurechnen. Mithilfe eines kleinen Python-Skripts, das C-Code generiert, der die entsprechenden Bitboard-Masken als Integer-Literals enthält, konnten wir alle diese Masken also direkt in unsere KI einkompilieren und direkt nutzen.

Dies funktioniert aber so einfach erstmal nur für non-Slider, also Könige und Springer, da Slider (Läufer, Türme, Königin) ja auf dem Weg von anderen Figuren blockiert werden können. Daher muss bei letzteren der Teil der Zielfelder, der nicht erreichbar ist, heraussubtrahiert werden. Hierzu wurden zusätzlich von jedem Feld in jede Richtung „Strahlen“ (sog. „rays“) vorgeneriert. So können von einem Bewegungsstrahl einer Figur z. B. alle Felder entfernt werden, die im Bewegungsstrahl der gleichen Richtung der blockierenden (in diesem Fall gegnerischen) Figur liegen. Bei eigenen Figuren muss zusätzlich auch das Feld der blockierenden Figur entfernt werden.

Ein interessantes Problem, das sich hierbei ergibt, ist in einer Bitboard-Maske das „nächste“ blockierte Feld in Strahlrichtung zu finden, da es ja immer die „nächste“ Figur im Strahl ist, die blockiert. Hierzu genügt aber eine einfache Übersetzungstabelle der 8 Richtungen in „vorwärts“ im Bitboard (also in Richtung des MSB) oder „rückwärts“ (also in Richtung des LSB). Sobald die Richtung in der Bitreihenfolge des Bitboards bekannt ist, können entsprechend entweder die CTZ- oder CLZ-Instruktionen genutzt werden, um das „erste“ gesetzte Bit in der jeweiligen Richtung, und somit die blockierende Figur zu finden.

Legale moves und Schach-Situationen Um das zweite Problem der non-legalen Moves zu lösen, bedurfte es der Prüfung und Elimination von 3 speziellen Situationen:

- Bewegung unseres Königs ins Schach
- Bewegungen eigener Figuren, die den Gegner aktiv ins Schach setzen
- Bewegungen von gepinnten Figuren aus ihrer Blockposition, sodass eine Schach-Situation entstehen würde.

Diese werden größtenteils separat und mithilfe einiger Vorberechnungen bei der Initialisierung des Movegenerators bearbeitet:

Eigener König im Schach Was sich zunächst trivial anhört, erfordert tatsächlich erheblich viel Rechenkraft. Denn um zu wissen, welche Bewegungen unseres Königs ihn in Schach setzen würden, müssen wir wissen, welche Felder der Gegner mit seinen Figuren überhaupt erreichen kann. Hierzu

müssen wir also theoretisch den kompletten Movegenerator nochmal auf den Gegner anwenden. Aber wenn wir dann wieder sehen wollen, wo dessen König von unseren Figuren geschlagen werden können, würde das zu unendlicher Rekursion führen.

Daher wenden wir zwar den Movegenerator auf den Gegner an, aber berechnen in dem Fall nur die pseudo-legalen Züge, also auch welche, die ggf. aufgrund der Schach-Regeln nicht erlaubt wären. So verlieren wir im schlimmsten Fall ein paar mögliche Züge, aber sparen erheblich an Rechenkraft.

Auch hierbei ergeben sich aber noch weitere Sonderfälle, die beachtet werden müssen:

Zum einen wäre zum Beispiel der Zug eines angegriffenen Königs weg von seinem Angreifer zwar nicht zwingend im Bereich der vom Gegner aktuell erreichen Felder, aber nach seiner Bewegung ist der Weg ja frei und er somit immer noch schlagbar. Diese Situation sollte in Racing Kings nicht auftreten, da Schach-Situationen ja gar nicht entstehen können, aber zur Sicherheit entfernen wir trotzdem den eigenen König aus der Maske der für den Gegner attackierbaren/blockierenden Felder.

Zum anderen sind z. B. Felder, auf denen unser König schlagen kann, also gegnerische Figuren auch nicht vom Gegner erreichbar, da dort ja seine eigenen Figuren stehen. Nach der Bewegung könnte der König also trotzdem angreifbar sein. Also fügen wir zusätzlich alle gegnerischen Figuren der Maske der vom Gegner schlagbaren/erreichbaren Felder hinzu.

Die Maske der möglichen gegnerischen Züge wird dann zwischengespeichert und bei der Generierung der möglichen Königszüge von dessen Standard-Maske entfernt.

Gegner aktiv in Schach setzen Um zu verhindern, dass wir den gegnerischen König ins Schach setzen, bedienen wir uns der Umkehrbarkeit von Schachbewegungen: Jeder Zug von einem Startfeld zu einem Zielfeld kann auch vom Zielfeld zurück zum Startfeld getätigt werden.

Das heißt, dass wir einfach für jeden Figurtypen vom gegnerischen König aus berechnen, wohin sich die Figur bewegen könnte, wenn sie dort stünde. So erhalten wir für jeden Figurtypen eine Maske von „verbotenen“ Feldern, die wir später im Movegenerator von den möglichen Zügen entfernen können.

Auch hier muss der Sonderfall betrachtet werden, dass eigene Figuren nicht und gegnerische Figuren geschlagen werden können und eine Figur sich somit z. B. in einen Bereich begeben, der von einer anderen Figur vom König aus gesehen blockiert wird. So kann die zweite Figur gepinnt werden, was ggf. sehr sinnvoll sein kann.

Gepinnte Figuren bewegen Der vorige Fall berücksichtigt einen weiteren Fall jedoch nicht. Es kann auch durch die Bewegung einer Figur ein Weg für einen Slider freigegeben werden, sodass eine Schach-Situation entsteht. Solche Figuren, die sich nicht aus einem bestimmten Bereich wegbewegen dürfen, nennen wir „gepinnt“.

Im Gegensatz zum Schach gilt diese Regel bei Racing Kings auch für beide Könige, nicht nur den jeweils einigen. Also müssen wir bei der Initialisierung des Movegenerators von beiden Königen aus in jeweils allen 8 Richtungen nach jeweils passenden Slidern suchen, die ggf. Figuren pinnen könnten. Wenn solche Slider gefunden werden, dann kann mit einem entgegengesetzten Strahl vom Slider aus eine Maske des potenziellen Pin-Bereichs erstellt werden, in dem dann nach gepinnten Figuren gesucht wird. Eine Figur ist hier nur gepinnt, wenn sie die einzige in diesem Strahl ist. Dies ist ggf. schwer zu erkennen, wenn mehrere passende Slider in dem Strahl vom König aus sind, da der nähere andere Figuren, aber der hintere auch den näheren pinnen könnte. Daher müssen die potenziell pinnenden Slider vom König aus der Reihe nach auf gepinnte Figuren geprüft werden.

Wenn eine einzelne gepinnte Figur gefunden wird, wird für deren Position eine Maske zwischengespeichert, die nur den Pin-Bereich enthält, da sich die Figur nicht aus diesem herausbewegen darf. Dieser wird später bei der Movegenerierung auf die möglichen Züge maskiert. Nicht gepinnte Figuren haben also eine Maske, die aus nur 1-Bits besteht, da sie sich im Sinne des Pinnens frei bewegen können.

3.5 Gameserver-Kommunikation

Eine weitere essenzielle Komponente der KI ist selbstverständlich die Kommunikation mit dem Gamserver. Diese findet über eine HTTP-REST-Schnittstelle inklusive eines SSE-Streams für Events statt.

Zunächst wollten wir diese auch komplett selbst in C schreiben, was sich aber aufgrund der erforderlichen SSL-Verschlüsselung als relativ komplex erwies. Als effiziente, einfache Alternative bot sich das bekannte HTTP-Werkzeug Curl [7], als C-Bibliothek als auch als Kommandozeilenanwendung, über die sämtliche Funktionen mit akzeptabler Effizienz erreicht wurden.

4 KI-Techniken

4.1 MTD(f)

Bei MTD(f) handelt es sich um eine Erweiterung zum regulären α - β -Algorithmus [11]. Dieser basiert auf dem 1979 vorgestellten SSS* Algorithmus, der bereits zu seiner Erfindung effizienter war als ein herkömmlicher α - β -Algorithmus. SSS* setzte sich jedoch nicht gegen den bewährten α - β -Algorithmus durch, da er eine sehr aufwendige und große Datenstruktur braucht. Diese Datenstruktur kann jedoch mit einem MTD-Algorithmus, einem "memory enhanced test driver", umgangen werden [10].

MTD-Algorithmen basieren auf Null-Window-Suchen. Hier ist die Idee, dass α und β beim Aufruf initialisiert werden. Der tatsächliche Score liegt genau dann innerhalb $[\alpha, \beta]$, wenn der zurückgegebene Wert ebenfalls innerhalb des Intervalls oder auch Fensters liegt. Bei uns ist das Fenster $[-(\beta + 1), \beta]$, da wir Negamax verwenden. Da das Berechnen eines solch kleinem Fensters deutlich schneller geht als das Berechnen eines unendlich großen Fensters, wie es bei herkömmlichem α - β -Algorithmen der Fall ist, kann diese Information genutzt werden, um einen effizienten Test durchzuführen, ob der tatsächliche Score über, unter oder gleich einem Wert ist.

Diese Form von Test wird nun von dem MTD(f)-Algorithmus benutzt, um sich dem echten Score anzunähern. Dazu wird ein Intervall $I = (-\infty, \infty)$ initialisiert und dann iterativ ein Wert $f \in I$ bestimmt. Dieser Wert wird folgendermaßen generiert:

$$f_i = \begin{cases} g, & \text{falls } i = 0 \\ \min(I_i) + 1, & \text{falls } \min(I_i) = f_i \\ \max(I_i), & \text{sonst} \end{cases}$$

Hier ist $\min(I_i)$ und $\max(I_i)$ das jeweils kleinste bzw. größte Element in I . I_i und f_i entsprechen den Werten der Variablen in der i -ten Iteration. Nun wird über die Null-Window-Suche berechnet, ob f_i über bzw. unter dem optimalen Score liegt und dementsprechend als neue Intervallgrenze in I gesetzt. Das Intervall wird so lange verkleinert, bis es nur noch den optimalen Score beinhaltet, welcher auch ein herkömmlicher α - β -Algorithmus finden würde. Der Algorithmus ist umso schneller, je besser der Startwert g gesetzt wird. Eine gute Einschätzung für diesen Wert kann über eine iterative Tiefensuche getätigt werden. Hier entspricht g immer dem optimalen Score der letzten Iteration. In der ersten Iteration ist $g = 0$. Dieser Algorithmus kann nun weiter über eine Transposition-Table optimiert werden.

4.2 Transposition Table und Zugsortierung

Transposition-Tables sind eine KI-Technik die 1970 zuerst eingesetzt wurde [14]. Sie dienen dazu, die Suche zu optimieren.

Durch den hohen Verzweigungsgrad von schachtartigen Spielen können Spielzustände meist durch mehr als eine Zugsequenz hervorgerufen werden, diese Züge nennt man Transposition. Wird bei einer MiniMax-KI ein bereits bewerteter Zustand erneut erreicht über eine andere Zugsequenz, so wird die KI ihn erneut bewerten. Um sich diesen doppelten Aufwand zu sparen haben wir Transposition-Tables genutzt. In dieser werden die unterschiedlichen berechneten Stellungen in eine Hash-Tabelle gespeichert, neben dem Spielzustand werden auch zusätzlich die Tiefe, alle berechneten Züge mit deren Scores und ein Fenster, welches den optimalen Score eingrenzt, gespeichert.

Wenn eine Transposition gefunden wurde, deren Suchtiefe größer oder gleich der aktuellen ist und das aktuell zu durchsuchende Fenster nicht innerhalb des Fensters der Transposition liegt, dann kann sofort terminiert werden, da wir das Ergebnis der alten Transposition nutzen können. Andernfalls wird diese Spielsituation noch einmal durchsucht. Hier kann nun die Zugreihenfolge abhängig der Scores des letzten Durchlaufs der Spielsituation genutzt werden um, eine bessere Zugsortierung zu erzielen. Natürlich können nur die Scores der Züge gespeichert werden, welche auch durchsucht wurden, deshalb werden alle Züge, welche nach einem Cutoff generiert worden sind, mit einem maximal niedrigen Score gespeichert, sodass sie erst nach den anderen Zügen durchsucht werden.

Unsere Transposition-Table ist eine Hash Tabelle. Als Hashing Algorithmus nutzen wir Zobrist Hashing. Dieser hat den Vorteil, inkrementell den Hashwert bei einer Bewegung anpassen zu können und somit Iterationen zu sparen. Da wir einen 64-Bit Integer Wert für die Hashs verwenden, kann es zu Kollisionen kommen. Diese werden bei uns ohne Strategie behandelt und es wird lediglich der alte Wert mit dem neuen ersetzt. Das ist sinnvoll, da es aufgrund des großen Wertebereiches einer 64-bit Integer zu sehr wenigen Kollisionen kommt.

4.3 Quiescence Search

Quiescence Search oder zu Deutsch Ruhesuche ist eine KI Technik die, im 3-ten Meilenstein in die KI eingebaut wurde[3]. Das Grundprinzip der Ruhesuche besteht darin, unruhige Positionen über die normale Suchtiefe hinaus zu untersuchen. Was eine unruhige Position ist, hängt dabei immer von der Definition des Programmierers ab. In unserer KI ist eine ruhige Spielsituation, wenn im letzten Zug kein Charakter geschlagen wurde. Ein schlagender

Zug kann somit auch über die maximale Tiefe hinaus den Suchbaum durchlaufen.

4.4 Null Move Reduction

Eine nächste Erweiterung unserer Minimax-KI waren Reduktionen. Hier ist die Idee, dass man versucht mehr Tiefe gegen Breite bei der Durchsuchung des Baums, zu tauschen. Das bedeutet, dass manche Knoten, welche weniger gute Züge machen, weniger durchsucht werden als andere, welche beispielsweise eher zu einem Cutoff führen. Das ist natürlich eine heuristische Einschätzung und kann in manchen Fällen zu schlechteren Ergebnissen, als α - β -Algorithmen ohne Reduktionen kommen.

Eine Variante davon ist die Null-Move-Reduction [1]. Sie basiert auf der Annahme, dass es fast immer besser ist einen Zug auszuführen, als in einer Runde nichts zu tun. Man simuliert dabei in jeder Spielsituation einen "Null Move", also einen Zug, der keine Figur bewegt, und führt eine α - β -Suche ohne Null-Move-Reduction durch mit reduzierter Suchtiefe. Wenn es nun trotzdem zu einem Cutoff kommt, dann wird die Suchtiefe auch bei der Suche für den normalen Zug reduziert.

4.5 Startspielbibliotheken

Die Idee von Startspielbibliotheken basiert darauf, dass unser Zeitmanagement viel Zeit für die Startzüge zur Verfügung stellt. Diese sind jedoch häufig sehr ähnlich und können somit vorberechnet werden, sodass einerseits Zeit gespart wird und andererseits gute Startzüge getätigt werden.

Ein weiterer Vorteil dieser Technik ist die einfache Implementierung. Hier kann die Transposition Tabelle verwendet werden, indem sie einfach zu Beginn des Spieles mit den bereits berechneten Spielen initialisiert wird.