Baguette - Eine Schachengine

Alessio Ragusa, Matthias Sennikow, Patrick Plewka

15.09.19

Contents

1 Einleitung **3**

2 Controller **4**
- 2.1 Allgemein . . . 4
- 2.2 Start . . . 4
- 2.3 Spielvorgang . . . 4

3 UCI **5**
- 3.1 Allgemein . . . 5
  - 3.1.1 Phasenunabhängige Befehle . . . 5
- 3.2 Phase 1 Initialisierung . . . 5
- 3.3 Phase 2 Spielvorgang . . . 6
- 3.4 Klassen . . . 6
  - 3.4.1 UCI . . . 6
  - 3.4.2 UCIBridge . . . 7
  - 3.4.3 UCIListener . . . 7
  - 3.4.4 InfoHandler . . . 7
  - 3.4.5 Log . . . 7
  - 3.4.6 UCICommands . . . 8
  - 3.4.7 Debug . . . 8
  - 3.4.8 UCIOptionHandler . . . 8
  - 3.4.9 OptionValuePair . . . 9

4 Time Management **10**
- 4.1 Einhalten einer mitgegebenen oder berechneten Zeitlimits . . . 10
- 4.2 Berechnung der optimalen Zeit für ein Schachzug . . . 10
- 4.3 Klassen . . . 10
  - 4.3.1 TimeManagement . . . 10
  - 4.3.2 TimeManBlitzChessBased . . . 11
  - 4.3.3 TimeManThread . . . 11

5 Evaluation **12**
- 5.1 Allgemein . . . 12
- 5.2 Evaluationsfunktion . . . 12
  - 5.2.1 Materialwert . . . 12
  - 5.2.2 Piece Square Tables . . . 13
  - 5.2.3 Repetition Score . . . 13
  - 5.2.4 Ist ein Bauer blockiert . . . 13
  - 5.2.5 Königsschutz . . . 13

1 Einleitung

Baguette ist eine Schachengine von Alessio Ragusa, Patrick Plewka und Matthias Sennikow in Rahmen einer Projektarbeit. Sie wurde in der Programmiersprache Java 11 entwickelt worden, verwendet Maven als Build-Management Tool und ist für Systeme mit mindestens Windows 10 oder Linux optimiert.
Die Engine wurde gezielt darauf programmiert Blitzschach bzw. im Modus Sudden Death[Unk, 1. Time Controls, Sudden Death] spielen zu können.
Das Projekt besteht aus insgesamt 9 Teilprojekten, die man in 3 Kategorien einteilen kann:

- Kommunikation:
  Kommunikation mit der Schachoberfläche (durch die standartisierte Schnittstelle UCI) und die Koordination/Kommunikation innerhalb des Gesamtprojektes.
  Teilprojekte: UCI, Controller, Command

- Suchbaum:
  Generierung eines Suchbaums um einen bestmöglichen Schachzug ermitteln zu können.
  Teilprojekte: Search, Time Management, Move Generation, Evaluation

- Schachbrett-Darstellung:
  Die interne Darstellung des Schachbrettes, einer Schachposition und eines Schachzuges.
  Teilprojekte: Board, Move

2 Controller

2.1 Allgemein

Der Controller dient als Verbindungsstück zwischen UCI, Timemanagement und der Suche. Er wird vom Benutzer oder der GUI gestartet, startet die anderen Teile der Engine und vermittelt zwischen ihnen. Der Controller läuft als eigenständiger Thread, dem Controller-Thread. Der Controller ist in der Klasse Controller, die das UCIListener-Interface erfüllt, implementiert.

2.2 Start

Der Controller startet zuerst die UCI-Klasse; diese startet selbständig die restlichen Klassen zuständig für UCI. Der Controller erhält von der UCI-Klasse die eingestellten Optionen. Abhängig von den Optionen erstellt der Controller den Logger. Der Controller meldet sich als Beobachter bei der UCI-Klasse an. Abhängig von der Prozessorkernanzahl startet der Controller eine Menge von Search-Threads. Diese warten vorerst. Dann startet der Controller den UCI-Thread.

2.3 Spielvorgang

Der Controller wartet auf Befehle in seiner Befehlswarteschlange *commandQueue*. Die Befehle stammen von dem UCI- oder dem TimeManagement-Thread. Der UCI-Thread legt potentiell die Befehle *ucinewgame*, *stop*, *position* und *go* in *commandQueue*. Dies geschieht durch die Methoden des UCIListener-Interface, die Controller bereitstellt. Der TimeManagement-Thread legt potentiell den Befehl *stop* in *commandQueue*. Das Timemanagement hat dafür direkten Zugriff auf *commandQueue*.
Bei einem *go* Befehl wird das Timemanagement und die Suche gestartet.
Bei einem *stop* Befehl werden das Timemanagement und die Suche gestoppt und via UCIBridge wird der GUI der gefundene beste Zug mit dem *bestmove* Befehl übergeben.
Bei einem *ucinewgame* Befehl wird die Suche zurückgesetzt.
Bei einem *position* Befehl wird der Suche das momentane Brett und Zug übergeben.
Der Controller merkt sich den letzten Befehl und ignoriert darauffolgende Befehle, die keinen Sinn machen[MK06, l. 41]. Beispielsweise macht ein *go* Befehl keinen Sinn nach *stop*. Vor dem *go* müsste noch mindestens ein *position* Befehl kommen.

3 UCI

3.1 Allgemein

UCI ist eine standardisierte Schnittstelle zwischen grafischer Oberfläche (GUI) und Schachengine. Die Kommunikation findet über die Standardein- und Ausgabe (stdin und stdout) der beiden Programme ab.[MK06, l. 9]
Die Engine soll zu jedem Zeitpunkt Befehle entgegennehmen und verarbeiten können.[MK06, l. 15] Deshalb ist in Baguette ein eigener Thread, genannt UCI-Thread, für die Kommunikation zuständig.
Den Programmablauf kann man in zwei Phasen einteilen:

1. Phase: Initialisierung
2. Phase: Spielvorgang

3.1.1 Phasenunabhängige Befehle

Unabhängig von den Phasen sind nur die Befehle *quit*, *debug on*, *debug off* und *isready*. *quit* beendet Baguette ordnungsgemäss. Das Beenden auf andere Art (zum Beispiel Signale) kann zu korrupten Logs führen.
debug [on | off] schaltet den Debugmodus ein bzw. aus. Im Debugmodus werden interne Meldungen zur Fehlersuche mithilfe des *info string* Befehls ausgegeben.
isready muss von der Engine mit *readyok* beantwortet werden, um Bereitschaft zu signalisieren.

3.2 Phase 1 Initialisierung

Zuerst bestätigen sich Engine und GUI, dass UCI verwendet wird mit *uci* und *uciok*. [MK06, l. 59 - 66] Dann gibt die Engine ihren Namen und ihre Authoren mit *id name* und *id author* an. Die Engine gibt der GUI mithilfe des *option* Befehl alle möglichen Optionen an. Die GUI stellt Optionen mit dem *setoption* Befehl ein und bestätigt diese mit dem *isready* Befehl. Nur zu diesem Zeitpunkt erlaubt Baguette das Ändern von Optionen entgegen der UCI Spezifikationen[MK06, l. 89], da das Pflichtenheft verlangt, dass Optionen vor Spielstart eingestellt werden und manche Optionen zu komplex sind, um sie während des Spiels zu ändern (z.B. die Logdatei).

Diese Phase wird nicht im UCI-Thread sondern im Controller-Thread ausgeführt. Der Controller startet die *initialize* Methode der UCI-Klasse. Die UCI-Klasse liest die *ucioption.properties* Datei aus und gibt ihren Inhalt an die *initialize* Methode der UCIBridge weiter. Die UCIBridge initialisiert nun die Verbindung zu GUI bis zu den *option* Befehlen. Die UCIBridge gibt den Inhalt der *ucioptions.properties* Datei an den UCIOptionHandler weiter. Der UCIOptionHandler erstellt aus dem Inhalt der Datei die *option* Befehle, sendet sie via UCIBridge an die GUI, wertet die *setoption* Befehle aus und gibt

alle Optionen mit ihren Werten an die UCIBridge zurück. Die UCIBridge gibt die Optionen über die UCI-Klasse an den Controller zurück. Die GUI ist jetzt bereit zu Spielen und der Controller kann abhängig von den Optionen den Rest von Baguette starten.

3.3 Phase 2 Spielvorgang

Der UCI-Thread wartet auf eine Eingabe von der GUI. Zuerst filtert die UCIBridge alle phasenunabhängigen Befehle aus den Eingaben aus und bearbeitet diese. Die restlichen leitet sie weiter an die UCI-Klasse, nachdem sie alle unnötigen Leerzeichen entfernt hat. Dies vereinfacht das weitere Auslesen der Eingaben. Die UCI-Klasse verwirft alle Eingaben, ausser die, die oberflächlich wie die Befehle *ucinewgame*, *position*, *go* oder *stop* wirken.
Die UCI-Klasse bildet zusammen mit dem UCIListener-Interface ein Beobachtermuster. UCIListener melden sich bei der UCI-Klasse an und besitzen die Methoden *receivedGo*, *receivedNewGame*, *receivedPosition* und *receivedStop*.
Die Befehle *stop* und *ucinewgame* werden genau überprüft und im Erfolgsfall mit den Methoden *receivedNewGame* und *receivedStop* ohne Parameter an alle UCIListener gesendet. Eine Eingabe, die mit *go* startet, wird ohne Veränderung oder sonstiger Überprüfung mit der Methode *receivedGo* an alle UCIListener gesendet. Eine Eingabe, die mit *position* startet, wird versucht umzuwandeln in ein Board und ein Move Objekt. Scheitert das, wird eine Info gesendet, dass der Befehl nicht korrekt war. Das Board und das Move Objekt werden mithilfe der *receivedPosition* Methode an alle UCIListener gesendet.
Scheitert hierbei eine der Überprüfungen oder die Umwandlung in das Board bzw. Move Objekt, so wird die Eingabe als falsch angesehen und eine Info wird an die GUI gesendet.
Zwei Klassen in Baguette erfüllen das UCIListener-Interface: DebugListener und Controller. Der DebugListener sendet eine Info an die GUI über alle empfangenen Befehle mit ihren Parametern, solange sich die Engine im Debugmodus befindet.
Der Controller erstellt ein Command Objekt mit allen notwendigen Werten und speichert es in seiner *commandQueue*. Der Controller-Thread kann sie von dort entgegennehmen und verarbeiten.

3.4 Klassen

Hier werden alle für UCI relevanten Klassen mit Ausnahme vom Controller erläutert:

3.4.1 UCI

Die UCI-Klasse verbindet die UCIBridge mit allen UCIListenern mit einem Beobachtermuster. Da zu jedem Zeitpunkt nur ein UCI-Thread existieren soll, ist die UCI-Klasse als threadsicherer Singleton implementiert. Der UCI-Thread startet in der zweiten Phase die *awaitCommandsForever* Methode der UCI-Klasse. Diese Methode wartet auf Eingaben von UCIBridge und leitet sie an alle UCIListener weiter.

3.4.2 UCIBridge

UCIBridge bildet die Verbindung zur grafischen Oberfläche. Keine andere Klasse hat Zugriff auf stdin und stdout, da Java keine Garantien über Threadsicherheit von stdin und stdout gibt. UCIBridge erreicht dies, indem es als threadsicherer Singleton implementiert ist. UCIBridge bearbeitet alle phasenunabhängigen Befehle und leitet alle anderen an die UCI-Klasse weiter.

3.4.3 UCIListener

UCIListener ist ein Interface, das zusammen mit der UCI-Klasse ein Beobachtermuster bildet. UCIListener haben Methoden um über die Befehle *stop*, *go*, *ucinewgame* und *position* mit ihren Parametern von der UCI-Klasse informiert zu werden.

3.4.4 InfoHandler

UCI erlaubt es der Engine Infos mit dem *info* Befehl an die GUI zu senden [MK06, l. 248 - 313]. Baguette sendet als Infos die Tiefe des Suchbaums und die Menge der durchsuchten Knoten.
InfoHandler ist ein threadsicherer Singleton, der Infos von allen Teilen von Baguette gesendet bekommt, diese sammelt und gebündelt via UCIBridge an die GUI sendet. InfoHandler ist deshalb ein Singleton, da manche Infos zu einem *info* Befehl gebündelt werden müssen [MK06, l. 264] und die Infos aus allen Teilen von Baguette kommen können.
Zudem erlaubt UCI mit dem Befehl *info string* [MK06, l. 297] beliebige Meldungen an die GUI zu senden. InfoHandler bietet hierfür die beiden Methoden *sendMessage* und *sendDebugMessage* an.
sendMessage sendet einen Text als *info string* an die GUI. Ein mehrzeiliger Text wird in mehrere *info string* Befehle umgewandelt. Jedes Leerzeichen im Text wird zudem durch ein Leerzeichen gefolgt von einem Punkt ersetzt und der Text wird mit Anführungszeichen umgeben. Dies ist notwendig, da UCI der GUI vorschreibt mögliche in der Info befindliche Befehle auszuführen[MK06, l. 299]. Beispielsweise der Befehl *info string bestmove still not changed* würde als illegaler *bestmove* Befehl ausgeführt. Deshalb würde InfoHandler diesen Befehl durch *info string "bestmove .still .not .changed"* ersetzen. Durch die Verwendung von Punkt und Anführungszeichen ist hier kein UCI-Befehl mehr enthalten.
sendDebugMessage macht das gleiche wie *sendMessage*, sendet jedoch nur, wenn Baguette sich im Debugmodus befindet.

3.4.5 Log

Log schreibt alle Meldungen von stdin und stdout in eine Logdatei. Ob in eine Datei geschrieben werden soll und in welche wird durch eine Option festgelegt. Da die Logdatei erst am Ende der ersten Phase bekannt ist, werden nur Meldungen nach dem Festlegen der Optionen gespeichert. Die UCIBridge gibt Log alle ein- und ausgehenden Texte und

Log speichert sie in der Logdatei.
Zudem erlaubt Log noch eine Liste von OptionValuePairs in die Logdatei zu schreiben. Der Controller sendet Log die Liste nachdem alle Optionen festgelegt wurden.

3.4.6 UCICommands

Eine abstrakte Klasse, welche nur dazu dient alle UCI-Befehle als Konstanten zu speichern.

3.4.7 Debug

Eine abstrakte Klasse, die speichert, ob sich Baguette im Debugmodus befindet.

3.4.8 UCIOptionHandler

Mit dem *option* Befehl kann die Engine Optionen anbieten und mit dem *setoption* Befehl kann die GUI diese festlegen. Da im Laufe der Entwicklung nicht genau feststeht, wieviele und welche Optionen die Engine anbieten wird, behandelt UCIOptionHandler diese dynamisch.
In der *ucioptions.properties* Datei können Optionen hinterlegt werden.
Ein Beispieleintrag:

```
option.name.hash = Hash
option.type.hash = spin
option.default.hash = 4096
option.min.hash = 4
option.max.hash = 4096
```

Alle Zeilen beginnen mit *option* gefolgt von einem sogenannten Subtag. Mögliche Subtags sind

- *name* Der Anzeigename der Option
- *type* Der Typ der Option
- *default* Der Standardwert der Option
- *min* Der Minimalwert der Option
- *max* Der Maximalwert der Option

Die Zeilen mit den Subtags *name* und *type* sind für jeden Eintrag verpflichtend. Die anderen sind abhängig vom Typ der Option. UCI kennt die Typen

- *spin* Eine ganze Zahl zwischen einem Maximal- und einem Minimalwert.
- *string* Ein Text.

- *button* Ein Knopf, den die GUI anzeigen kann. Ein *setoption* Befehl mit diesem Typ signalisiert, dass der Knopf vom Benutzer gedrückt wurde.
- *combo* Eine feste Auswahl an Texten.
- *check* Ein Wahrheitswert.

Alle Typen ausser *button* verlangen zudem eine Zeile mit dem Subtag *default*. Die Subtags *min* und *max* sind nur erlaubt beim Typ *spin*.
Nach dem Subtag kommt eine ID. Diese dient dazu die Option innerhalb des Programms eindeutig zu identifizieren. Ein Abruf des Werts dieser Option soll nur die ID verlangen.
Schliesslich kommt noch der Wert der Zeile.
Aus diesen Informationen erstellt UCIOptionHandler aus dem Beispielseintrag den Befehl *option name Hash type spin default 4096 min 4 max 4096*.

Beim Empfangen von *setoption* Befehlen, überprüft UCIOptionHandler, ob die Option existiert, der Typ korrekt und der Wert erlaubt ist. Ist das der Fall, speichert UCIOptionHandler den Wert mit der Option in einer Liste von OptionValuePairs. Das Überschreiben einer schon festgelegten Option wird dabei unterstützt. Hat die GUI alle Optionen gesendet, werden nicht festgelegte Optionen mit ihren Standardwerten in die Liste der OptionValuePairs eingefügt und die Liste an den Controller geschickt.
Das Abfragen des Beispieleintrags könnte dann durch

```
controller.getOptionsValue("hash");
```

erfolgen. Um keine Probleme mit Javas Typsystem zu bekommen, werden die Werte der Optionen als String gespeichert. An den notwendigen Stellen muss der Wert dann noch umgewandelt werden. UCIOptionHandler garantiert aber, dass der String umgewandelt werden kann.

3.4.9 OptionValuePair

Eine Klasse, die eine Option mit einem Wert speichert.

4 Time Management

Das Time Management kümmert sich um das Zeitverhalten der Schachengine. Dies beinhaltet das Einhalten einer mitgegebenen oder eines berechneten Zeitlimits und die Berechnung der optimalen Zeit für einen Schachzug als Zeitlimit.

4.1 Einhalten einer mitgegebenen oder berechneten Zeitlimits

Es ist in Blitzschach wichtig, die gegebene Gesamtzeit nicht zu überschreiten, denn Zeitüberschreitung bedeutet ein automatischer Sieg für die Gegnerseite.
Um das zu verhindern, muss dafür gesorgt werden, dass das Time Management die Engine möglichst bei Ablauf stoppt. Dafür läuft das eigentliche Time Management in einen eigenen Thread, der nach Zeitablauf das Stop-Signal sendet.

4.2 Berechnung der optimalen Zeit für ein Schachzug

Besonders für Blitzschach ist es wichtig, seine gegebene Zeit so weit wie möglich so aufzuteilen, dass einerseits genug Zeit für die Berechnung eines Schachzuges besteht und dass nicht zuviel von der Gesamtzeit für nur ein Zug aufgebraucht wird. Zudem darf keine allgemeine Zeitüberschreitung passieren, weil sonst der Gegner automatisch gewinnt.
Um das zu gewährleisten, geht die Engine von insgesamt 80 Zügen aus. Die ersten 40 erhalten 50 Prozent der Zeit, der Rest bekommt immer weniger Zeit. Falls ein Inkrement dazugegeben wird, wird es halbiert aufsummiert.
es wird außerdem 5 Millisekunden abezogen für die Tatsache, dass der gesamte Prozess vom Begin des GO-Befehls bis zu den eigentlichen Ablaufen des Zeitlimits durchschnittlich 4-5 Millisekunden braucht.

4.3 Klassen

Das Time Management besteht aus einen Interface und zwei Klassen:

4.3.1 TimeManagement

Ein Interface bestehend aus drei Methoden:

1. init(totalTimeLeftInMsec, inc, movesCnt):
   Initialisierung über, bei der das Zeitlimit berechnet werden soll.
   Die Parameter:
   - totalTimeLeftInMsec: wieviel Gesamtzeit übrig ist.
   - inc: Das zu Verfügung gestellte Inkrement.
   - movesCnt: die bisherige Anzahl an Schachzügen.

Das TimeManagement sollte vor der Initialisierung bereits in einem nicht Initialisierten Zustand sein (entweder noch nie gestartet oder reset() wurde davor aufgerufen.

2. init(moveTime):
   Initialisierung des Time Management mittels einer genauen Zeit

3. isEnoughTime():
   Überprüfung, ob noch genug Zeit ist.

4. reset():
   Zurücksetzen des Zeitlimits.

Über dieses Interface kann man für verschiedene Schacharten ein Time Management festlegen.

4.3.2 TimeManBlitzChessBased

Eine Implementierung des TimeManagement Interface gezielt auf Blitzschach bzw Schacharten mit einen gesamten Zeitlimits bis zu 10 Minuten.
Ursprünglich war es geplant, dass im Suchbaum bei jeder neuen Suchtiefe einmal abgefragt wird, ob noch genug Zeit da ist für eine weitere Suchtiefe (isEnoughTime()), jedoch kam das Problem auf, das man ab einer bestimmten Suchtiefe zu lange auf einer Suchtiefe stecken bleibt und so das Zeitlimit überschritten hat.

4.3.3 TimeManThread

Diese Klasse ist eine Thread Implementierung, die dafür gedacht ist, mit einem TimeManBlitzChessBased Objekt ein Zeitlimit zu berechnen und das innerhalb eines Threads ablaufen lassen. Dessen Implementierung in run() sieht so aus:

```
1     long start = System.nanoTime();
2     try {
3         Thread.sleep(timeFrame); // - overhead by go/stop cmd
4     } catch (InterruptedException e) {
5     }
6     if (!isInterrupted()) {
7         queue.add(new Command(Command.CommandEnum.STOP));
8     }
9     long estimated = (System.nanoTime() - start) / 1000000;
10    InfoHandler.sendDebugMessage("TimeMan: " + estimated);
```

Die Zeilen 6-8 Dienen für den Fall, dass vor dem Ablaufen des Zeitfensters bereits ein STOP-Befehl eingegangen ist. Beim senden des STOP-Befehl wird jeder laufende TimeManThread unterbrochen (interrupted), und die Threads werden kein STOP-Befehl aussenden wenn sie Unterbrochen wurden.

5 Evaluation

5.1 Allgemein

Die Evaluation stellt eine Funktion dar, die zu einer Schachposition eine Bewertung berechnet, mathematisch gesehen:

$$f(Schachposition) = Bewertung$$

Für die Bewertung werden verschiedene Aspekte der gesamten Spielposition analysiert, unter anderem:

1. Materialwert
2. Piece Square Tables (PST)
3. Repetitionscore
4. Ist ein Bauer geblockt
5. Königsschutz

im Endeffekt besteht unsere Evaluationsfunktion aus:

$$f(Schachposition) = Materialwert + PST + Repetitionscore + IsteinBauergeblockt + Königsschutz$$

5.2 Evaluationsfunktion

5.2.1 Materialwert

Jeder Art von Schachfigur wird ein *Materialwert* zugewiesen, in Baguette (entnommen vom Chessprogramming Wiki) sind es:

- König: Unendlich
- Dame: 9
- Turm: 5
- Springer, Läufer: 3
- Bauer: 1

Es wird von jedem Spieler der die Materialwerte zusammenaddiert:

$$matWhite = sum(pieces\ of\ site\ white)$$

$$matBlack = sum(pieces\ of\ site\ black)$$

das Ergebnis ist dann die Substraktion der Materialwerte Schwarz von den Materialwerten Weiß:

$$material = matWhite - matBlack$$

Die Einheit, die für die Materialwerte benutzt wird ist *Centipawns* (1 Bauer = 100 Centipawn).

5.2.2 Piece Square Tables

Piece Square Tables (PST) sind 8x8 Matrizen, die für eine bestimmte Seite und eine bestimmte Art von Schachfigur für jedes Feld auf den Schachbrett eine spezifische Bewertung hinzufügt. Die *PSTs* sollen so Positionen, die allgemein für eine bestimmte Schachfigur von Vorteil sind so im Suchbaum hervorheben. Genauso andersherum sollen so Positionen die ein Nachteil für eine Figur darstellen eine niedrigere Priorität bekommen bzw abgeschnitten werden.
Als Beispiel ist hier die eingebaute PST für Bauer Weiß [Mic, l. Piece Square Tables, Pawn]:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 50 & 50 & 50 & 50 & 50 & 50 & 50 & 50 \\ 10 & 10 & 20 & 30 & 30 & 20 & 10 & 10 \\ 5 & 5 & 10 & 25 & 25 & 10 & 5 & 5 \\ 0 & 0 & 0 & 20 & 20 & 0 & 0 & 0 \\ 5 & -5 & -10 & 0 & 0 & -10 & -5 & 5 \\ 5 & 10 & 10 & -20 & -20 & 10 & 10 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

5.2.3 Repetition Score

Der *Repetition Score* geht von dem Schachzug, der zu evaluieren ist den Suchbaum bis zur Wurzel hoch und überorüft wie oft der gleiche Schachzu bereits gemacht wurde. Für jede Wiederholung wird ein Bauer (100 Centipawns) abgezogen.

5.2.4 Ist ein Bauer blockiert

Wenn ein Bauer im nächsten Zug im Feld sich nicht mehr nach Vorne bewegen kann (weil vor dem Bauer dann eine weitere Schachfigur steht) und er nicht eine gegnerische Figur seitlich wegnehmen kann, gilt er als Blockiert und es wird dafür von der Bewertung ein halber Bauer (50 Centipawns) abgezogen.

5.2.5 Königsschutz

Der Königsschutz überprüft, ob ein König um sich herum genug Schutz von anderen Schachfiguren hat.
Für jede Schachfigur des gleichen Spielers, die auf den 8 Feldern rund um den König sind gibt es einen halben Bauern (50 Centipawns) Bonus.

References

[Mic] Thomasz Michniewski. Simplified evaluation function. https://www.chessprogramming.org/Simplified_Evaluation_Function (aufgerufen am 27.9.19.

[MK06] Stefan Meyer-Kahlen. Description of the universal chess interface (uci). https://www.shredderchess.de/download.html (aufgerufen 26.9.19), April 2006.

[Unk] Unknown. Time management, chess programming wiki. https://www.chessprogramming.org/Time_Management, (aufgerufen am 30.09.19).