Softwarepraktikum SS 2018

Assignment 1

Group - 3

Ramil Sabirov	369500	ramil.sabirov@rwth-aachen.de
Joel Choi	345575	joel.choi@rwth-aachen.de
Eric Remigius	366895	eric.remigius@rwth-aachen.de

Contents

1 Task 1 **3**
- 1.1 Map 1 . . . 3
- 1.2 Map 2 . . . 3
- 1.3 Map 3 . . . 3

2 Task 2 **5**
- 2.1 Pseudo-2D-Array . . . 5
- 2.2 Vector2i Klasse . . . 5
- 2.3 Transition Klasse . . . 6
- 2.4 TileStatus Klasse . . . 6
- 2.5 Tile Klasse . . . 6
- 2.6 Map Klasse . . . 7
- 2.7 Eine Map einlesen . . . 7

3 Task 3 **8**
- 3.1 Player . . . 8
- 3.2 Move . . . 8
- 3.3 GamePhase . . . 8
- 3.4 MapWalker . . . 8
- 3.5 MoveManager . . . 9
- 3.6 Algorithmus . . . 9

4 Task 4 **11**
- 4.1 Zugehörigkeit der Funktion . . . 11
- 4.2 Aufbau der Methode . . . 11
  - 4.2.1 Building phase . . . 11
  - 4.2.2 Bombing phase . . . 12

5 Task 5 **13**
- 5.1 Building phase . . . 13
  - 5.1.1 Mittelspiel . . . 13
  - 5.1.2 Endspiel . . . 14
- 5.2 Bombing phase . . . 15

Task 1

Eine Beschreibung der Maps und eine Erörterung von möglichen Gewinnstrategien auf den jeweiligen Maps.

1.1 Map 1

Die Erste ist eine ziemlich simple Map, bei der es mehrere Löcher gibt, die entweder nur senkrecht und waagerecht oder auch diagonal die an den Löchern angrenzenden Tiles verbindet. Da es schon zu Beginn mehrere Anfangsmöglichkeiten gibt, ist es von Interesse wie die Spieler ihre ersten Züge realisieren. Wie wir in Task 5 erörtern ist die Mobilität von großer Bedeutung. Deshalb wird es wichtig sein zu versuchen die Bereiche mit den Löchern, wo nur senkrechte und waagerechte Transitionen existieren, zu vermeiden um keine Mobilität einbüßen zu müssen.

1.2 Map 2

Auf der zweiten Map ist die Idee mehrere separate Reversi-Spielfelder in einem Spiel unterzubringen, die allesamt minimal verschieden sind und auch jeweils uneinnehmbare Felder an den Ecken besitzen. Es ist von Interesse zu beobachten, welche Spieler welche Spielfelder priorisieren und zu schauen ob die Spieler ein Spielfeld zuerst ausspielen oder zwischenzeitlich auch auf andere Spielfelder ausweichen. Es ist davon auszugehen, dass auch wenn am Anfang die Möglichkeit besteht auf separaten Feldern zu starten, dass im Laufe des Spiels mindestens zwei Spieler um ein Feld konkurrieren werden, da sonst für einen Spieler bei einem Feld ohne Konkurrenz schnell die Bewegungsmöglichkeiten ausgehen werden. Wie bereits erwähnt, gibt es in jedem einzelnen Spielfeld uneinnehmbare Felder, die bei guten Gewinnstrategien von allen Spielern priorisiert werden sollten, wenn sie einnehmbar sind. Man sollte deshalb auch versuchen es zu vermeiden, durch einen Zug einem anderen Spieler die Möglichkeit zu eröffnen diese Felder einzunehmen. Außerdem gibt es 3 Felder die größer als die anderen sind und mehr Spezialfelder besitzen. Diese Felder sollten priorisiert werden, da man dort höhere Mobilität besitzt und mehr Felder einnehmen kann.

1.3 Map 3

Bei dieser Map ist die Startmobilität sehr begrenzt und die Map besitzt zunächst auch keine Besonderheiten. Von daher ist davon auszugehen, dass die Partie anfangs wie ein normales Reversi Spiel verlaufen wird. Jedoch gibt es dann außenrum ein Rechteck bestehend aus Holes, der nur senkrechte, an den Ecken diagonale und

keine waagerechte Transitionen bietet. Dies bedeutet, dass die Felder die an den Rechteck links und rechts angrenzen geringere Mobilität als andere Felder bieten. Von daher sollte man als Spieler versuchen so gut wie möglich in senkrechter Richtung Felder einzunehmen um ein Mobilitätsvorteil zu erhalten und somit besseren Zugriff zu den außerhalb liegenden Feldern zu bekommen.

Task 2

Im folgenden wird jede Klasse die zur Bearbeitung dieser Aufgabe angelegt wurde kurz erläutert. Auf die JUnit Testklassen wird dabei nicht eingegangen, da diese keine funktionale Bedeutung haben.

2.1 Pseudo-2D-Array

Ein Pseudo-2D-Array ist keine von uns angelegte Klasse sondern ein Begriff um ein, in einem eindimensionalen Array gespeichertes, zweidimensionales Array zu beschreiben. Dabei wird jede Zeile des Pseudo-2D-Arrays nacheinander in das eindimensionale Array geschrieben. Dies hat gegenüber normalen zweidimensionalen Arrays den Vorteil, dass alle Einträge am Stück im Speicher liegen und nicht eine Referenz auf ein Array von Referenzen gespeichert ist. somit muss nicht doppelt dereferenziert werden und es kann die Performance verbessern, wenn das gesamte Array in den Cache geladen wird und nicht nur eine Zeile davon. Der Zugriff auf ein solches Array ist auch nicht kompliziert: Um auf Position (x,y) zuzugreifen, muss lediglich die der Index im eindimensionalen Array berechnet werden: $Index = x + y * Breite$, wobei $Breite$ die Breite des 2D-Arrays beschreibt.

2.2 Vector2i Klasse

Diese Klasse repräsentiert einen zweidimensionalen Vektor mit Integer Werten. Diese Vektoren werden benutzt um Positionen und Richtungen auf der Karte zu speichern. Mit diesen Vektoren zu arbeiten, anstatt die Richtungsdefinition aus den Regeln zu benutzen, hat den Vorteil, dass sie das programmieren sehr erleichtern. So kann man einen Positionsvektor recht einfach verändern indem ein Richtungsvektor addiert wird, was in wenigen Zeilen Code gemacht werden kann. Würde man die Richtungen in Form eines Enums speichern, wären sehr viele switch-Statements notwendig um Schritte in eine Richtung zu machen, bzw. einen Richtungswechsel nach einer Transition durchzuführen.

Vector2i bietet zwei Konstruktoren um als Nullvektor, sowie als Vektor mit gegebenen Werten initialisiert zu werden. Des weiteren gibt es je zwei Funktionen um einen Vektor um einen Faktor zu skalieren und um zwei Vektoren zu addieren. Eine dieser Funktionen verändert jeweils das Objekt auf dem sie aufgerufen wurde und die andere ist jeweils eine statische Methode, die ein neues Vector2i Objekt zurück gibt, auf das diese Änderungen angewandt wurden.

2.3 Transition Klasse

In dieser Klasse werden die Transitionen einer Karte gespeichert. Dazu wird ein Endpunkt und eine Ankunftsrichtung, jeweils in Form eines Vector2i, festgehalten. Damit ist eine Transition also gerichtet, obwohl Transitionen laut Spezifikation ungerichtet sind. In unserer Implementation werden deshalb für jede Transition der Karte 2 Transition Objekte erzeugt. Diese Objekte werden an jeweils an ein Tile 'gehängt'. Da nur über den Ursprung auf eine Transition zugegriffen werden kann, muss dieser nicht mit von dieser Transition gespeichert werden. Eine Transition muss mit allen Attributen initialisiert werden, denn zu der Erzeugung einer Transition stehen diese bereits fest und werden im Verlauf des Spiels nicht mehr verändert.

2.4 TileStatus Klasse

TileStatus ist ein recht einfaches Enum, das den Zustand eines Tile codiert. Als Zustand wird hierbei die Belegung sowie die Art dieses Feldes gemeint, also ob es ein Loch oder ein richtiges Feld ist, ob ein Spieler einen Stein dort liegen hat oder es ein Bonusfeld ist. Man könnte diesen Zustand noch weiter aufteilen in verschiedene Enums: Eins, das die Art des Feldes definiert und Eins, das den Spieler festhält, dessen Stein auf diesem Feld liegt, allerdings würde dies nur zu mehr Aufwand führen.

Des weiteren gibt es noch eine Methode, die einen char entsprechend der Regeln in einen TileStatus umwandelt.

2.5 Tile Klasse

In der Tile Klasse werden die Felder des Spielfeldes verwaltet. Dazu wird jeweils ein TileStatus und ein 3x3 Pseudo-2D-Array von Transitionen gespeichert. Initialisiert werden kann ein Tile standardmäßig als Loch oder mit einem übergebenem TileStatus. Transitionen können erst nach der Initialisierung hinzugefügt werden, da diese auch in der Map Datei erst nach dem Spielfeld definiert sind.

Transitionen werden in dem Array über eine Richtung, also einen Vector2i indiziert. Dazu werden jedoch die beiden Werte des Vektor um eins inkrementiert, damit keine negativen Indices auftreten. Danach wird der Index bestimmt aus: $v.x + v.y * Breite$. Da es sich um ein 3x3 Array handelt ist die Breite folglich 3. Somit gibt es für jede Richtung an in die eine Transition verlaufen kann einen Slot in dem Array. Jedoch gibt es auch einen, der in keine Richtung geht. Dieser wird jedoch nie belegt.

Wird die Methode *addTransition(...)* aufgerufen, wird geprüft, dass keine Transition in der gewünschten Richtung vorliegt. Somit wird verhindert, dass zwei

Transitionen in dieselbe Richtung gehen. Von einem Tile kann nicht geprüft werden, dass eine Transition nur in eine Richtung geht, in der ein Loch liegt, da es keine Möglichkeit gibt auf den Nachbarn der Karte zuzugreifen.

2.6 Map Klasse

Alle Informationen über die Karte werden in dieser Klasse verwaltet. So werden jeweils die Anzahl der Spieler, die Anzahl der Overrides pro Spieler, die Anzahl der Bomben, die Stärke der Bomben, die Anzahl der Transitionen sowie die Dimensionen der Map gespeichert. Des weiteren gibt es ein Pseudo-2D-Array, in dem die Tiles des Spielfeld abgelegt sind. Diese sind jedoch innerhalb der Map Klasse von einer Reihe Löcher umgeben, was das iterieren über die Map vereinfacht: Es muss nicht geprüft werden ob man am Rand der Karte ist, sondern nur ob das nächste Feld ein Loch ist. Somit fasst man zwei Fälle zusammen. Nach außen wwerden die Koordinaten aber so angepasst, als gäbe es diese neuen Tiles nicht.

Eine Map muss mit einem Map-String gemäß Regeln initialisiert werden und kann danach nicht mehr verändert werden. Es kann lediglich lesend auf alle Variablen zugegriffen werden. Man kann auch über Referenzen auf die Tiles der Map zugreifen, sodass diese Ihren Status ändern können.

2.7 Eine Map einlesen

Eine Map wird per Konstruktor aus dem Map-String erzeugt. Dazu wird ein Standard Java Scanner verwendet, der das einlesen von Zahlen sehr vereinfacht und auch die Fehlertoleranz bei zu vielen Whitespaces erhöht. Jedes Mal, wenn dieser Scanner versucht Daten aus dem String zu lesen, wird dies in einem Try-Block verschachtelt und bei einem Fehlschlag eine IllegalArgumentException mit einer Nachricht, entsprechend des fehlgeschlagenen Segments, geworfen.

Zuerst liest der Scanner die Metadaten aus dem Kopf des Strings und speichert diese in die zugehörigen Variablen. Bei den Dimensionen der Karte wird jeweils eine 2 addiert, um Platz für die Umrandung mit Löchern zu haben. Danach wird ein neues Array für die Tiles mit passenden Dimensionen erzeugt. Als nächstes werden die einzelnen Felder aus dem String eingelesen und die Löcher am Rand erstellt. Zuletzt werden noch die Transitionen eingelesen. Dazu werden zuerst die Integer aus dem String gelesen und diese danach zu Vektoren für die Positionen und Richtungen konvertiert. Dabei wird beachtet, dass die Ausgangsrichtungen und Eingangsrichtungen inverse sind, also dass man nach oben ein Tile verlässt aber man beim betreten nach unten geht. Auch wird überprüft, dass eine Transition in keinem Loch startet oder endet und dass in die Richtung der Transition kein benutzbares Tile ist. Zuletzt werden den beiden an der Transition beteiligten Tiles die erstellten entgegengesetzten Transitionen hinzugefügt.

Task 3

Um diesen Task zu bearbeiten wurden wieder mehrere Klassen hinzugefügt.

3.1 Player

Die Player Klasse ist eine recht simple Klasse, die Informationen zu einem, am Spiel teilnehmenden, Spieler bereithält. So wird etwa die Spielernummer, sowie die Anzahl der Bomben und Overridestones gespeichert. All diese Daten müssen zur Initialisierung vorliegen, können später allerdings über Getter und Setter verändert werden.

3.2 Move

Diese Klasse beinhaltet einen Vector2i, der die Koordinaten des Feldes codiert, sowie die Spielernummer und die eventuell anfallenden Sonderinformationen des Spielzuges. Diese Daten müssen zur Initialisierung angegeben werden und können danach nur noch abgefragt werden.

3.3 GamePhase

GamePhase ist ein sehr simples Enum, dass Einträge für die beiden Spielphasen, Bombing Phase und Building Phase, enthält.

3.4 MapWalker

Dies ist eine Klasse, die dazu genutzt wird, über die Map zu laufen. Sie beinhaltet eine Referenz auf die Spielkarte, eine Position und eine Richtung in Form zweier Vector2i, sowie eine bool Variable, die dazu dient diesen Läufer zu deaktivieren. Ein Läufer funktioniert so, dass er an einer Position startet und in eine feste Richtung läuft. Dabei folgt er automatisch Transitionen und passt die Laufrichtung entsprechend an. Bei einem Aufruf der Methode *step()* macht der Walker genau einen Schritt in die gegebene Richtung, falls dies möglich ist. Außerdem gibt sie einen boolean zurück, der angibt ob ein Schritt gemacht wurde. Man kann mittels der *canStep()*-Methode auch prüfen ob der Walker einen Schritt machen kann, oder ob ein Loch den Weg in diese Richtung blockiert.

Um einen Schritt zu machen, prüft ein MapWalker zuerst, ob ob er aktiv ist und ob er einen Schritt nach vorne machen kann. Danach wird geprüft ob das nächste Tile ein Loch ist. Ist es kein Loch, so wird einfach die Position verändert. Ein Richtungswechsel findet nicht statt. Sollte das nächste Tile jedoch ein Loch

sein, so wird die Position des Walkers auf den Endpunkt, sowie die Richtung des Walkers auf die Richtung der Transition gesetzt.

Durch den Einsatz einer solchen Klasse, erspart man sich das komplette iterieren über die Karte und vereinfacht den Umgang mit den Transitionen.

3.5 MoveManager

Der MoveManager verwaltet die Spielzüge, die getätigt werden und getätigt werden können. Dafür hat auch er eine Referenz auf die Spielkarte. Des weiteren pflegt er ein Array mit Playern um die Daten der Spieler parat zu haben. Als letztes hat er noch eine Variable, die die aktuelle Phase in Form eines GamePhase des Spiels beinhaltet, auch wenn diese zur Zeit nicht beeinflusst werden kann, da noch kein richtiges Spiel stattfindet. Der MoveManager muss mit einer fertigen Map initialisiert werden und extrahiert alle für ihn notwendigen Informationen aus dieser. Er stellt auch eine Methode bereit, die einen Move auf dessen Korrektheit.

3.6 Algorithmus

Der Algorithmus prüft zuerst denn Fall, dass das Spiel in der Bombing Phase ist, da dieser sehr einfach zu prüfen ist: Es wird überprüft, ob das getroffene Feld ein Loch ist, ob der ausführende Spieler noch Bomben hat und ob das Spezialfeldattribut auf 0 steht.

In der Building Phase wird als erstes geprüft, ob das Zielfeld des Moves ein Loch ist. Falls dem so ist, ist der Zug nicht gültig. Als nächstes wird geschaut ob das Feld bereits besetzt ist und der Spieler keine Overridestones mehr hat. Auch in diesem Fall ist der Zug ungültig. Danach werden die Spezialfeld Attribute überprüft, so dass diese den Netzwerkspezifikationen entsprechen. Ein Abweichen resultiert auch in einem ungültigen Zug. Wenn das Zielfeld ein Expansionstone ist, so kann dieser nun als gültiger Zug betrachtet werden, denn der Spieler hat an dieser Stelle des Programms noch Overridestones, sonst hätte der Algorithmus den Zug bereits als ungültig bewertet. Es muss also nur noch geprüft werden, dass ein Stein umgedreht werden kann in dem Zug. Dafür werden acht MapWalker erstellt, die in alle Richtungen von dem Zielfeld weg laufen. Zuerst machen diese nur einen Schritt und es wird geprüft, ob das Zielfeld an ein besetztes Feld angrenzt, das nicht von dem Ziehenden Spieler besetzt ist. Sollte es kein solches Feld geben, kann kein Stein umgedreht werden und der Zug ist ungültig. Danach laufen die Walker weiter in ihre Richtungen, bis sie auf ein leeres Feld oder auf ein vom ziehenden Spieler besetztes Feld stoßen. Im zweiten Fall, muss nur geprüft werden, dass es nicht das Startfeld ist, da auch im Kreis gegangen werden kann, durch Transitionen. Ist es nicht das Startfeld, so wurde ein gültiger Zug gefunden. Trifft ein Walker auf ein Loch oder ein leeres Feld, so bleibt er stehen, damit er nicht zufällig auf ein

besetztes Feld gerät und somit ein Pfad mit einem leeren Feld in der Mitte als gültig erkennt.

Durch die Verwendung der MapWalker werden auch die Transitionen korrekt beachtet.

Die Überprüfung auf Korrektheit der Spezialfeldattribute wurde an dieser Stelle gemacht und nicht am Netzwerkinterface, da vom Server nur gültige Züge übertragen werden sollten und bei der Ermittlung der gültigen Züge, die später wichtig sein wird, diese Felder auch von Bedeutung sein werden und das Netzwerkinterface nichts mit der Ermittlung der möglichen Folgezüge zu tun hat.

Task 4

Einige Erläuterungen zur Implementierung der Funktionalität, welche einen Spielzug durchführt und den momentanen Zustand in den Nachfolgezustand überführt.

4.1 Zugehörigkeit der Funktion

Die Methode mit der Signatur $applyMove(Move\ m)$ und die jeweils benötigten Hilfsmethoden wurden funktional passend untergebracht in der Klasse MoveManager, welche verantwortlich ist für das Verwalten und Verändern der Spielsituationen.

4.2 Aufbau der Methode

Zunächst wird in der Methode davon ausgegangen, dass der übergebene Zug valide ist. Ansonsten verhält sich die Methode beliebig. Die Methode spaltet sich auf in einen Teil, welcher in der Bauphase und einen welcher in der Bombphase arbeitet.

4.2.1 Building phase

Durch Nutzung unseres Werkzeugs für eine Iteration über die Map, der MapWalker, wird in jede Richtung von der gesetzten Steinposition aus über die Map iteriert bis entweder

a) ein eigener Stein gefunden wird, dann müssen alle dazwischen in die eigene Farbe gebracht werden)

b) ein nicht besetztes Feld (oder Loch) gefunden wird, dann hat der Spielzug in diese Himmelsrichtung keine Auswirkung.

Das 'Drehen' der Steine geschieht durch die Methode
$flipStone(Vector2i\ pos, int\ playernumber)$, welche das Spielfeld aktualisiert und außerdem noch die Datenstruktur anpasst, die jeder Spieler hält. In dieser Felder, unsere Wahl war das HashSet, werden die Positionen der Steine gespeichert, die der Spieler gerade belegt hat.
Die Wahl des HashSets, begründet sich dadurch, dass keine Positionen doppelt vorkommen sollen/können und die Reihenfolge der Elemente beim Iterieren über das Set keine Rolle spielt.
Der zusätzliche Speicherbedarf und Verwaltungsaufwand soll ein effizientes Verfahren bei den Bonusfeldern ermöglichen. So ist das Tauschen von den Steinen zweier Spieler mit relativ wenig Aufwand möglich, da die Positionen stets bekannt sind. Der Effekt des Inversionfeldes wird durch eine Reihe von Vertauschungen zwischen den Steinen zweier Spieler umgesetzt.

4.2.2 Bombing phase

Hier wird auf eine rekursive Hilfsmethode zurückgegriffen, welche um alle Felder zu bomben, welche zum Bombfeld höchstens eine Entfernung von n haben, sich selber aufruft für die Nachbarn mit dem Radius $n - 1$.
Um effizienter zu arbeiten, wird hier eine HashMap genutzt, welche speichert an welchen Positionen man mit welchem Radius bereits war. Dann vermeidet man es von derselben Position aus mit kleinerem oder gleichgroßen Radius erneut zu suchen und bereits gefundene Felder zum wiederholten Male zu finden.

Task 5

Eine kurze Recherche über die klassische Variante des Spiels und einige Testspiele gegen eine Reversi-KI ergaben folgenden Vorschlag für eine Evaluationsfunktion:

5.1 Building phase

Entgegengesetzt dem eigentlichen Spielziel in der building phase, welches wäre am Ende mit den meisten Steinen dazustehen, ergab sich aus empirischen Beobachtungen, dass die Anzahl der Steine am Anfang und in der Mitte des Spieles (fast) keinerlei Rolle spielt. Erst gegen Ende des Spieles wird dieser Faktor entscheidend.
Das ist der Grund, warum sich unsere Bewertungsfunktion unterteilt in Mittelspiel und Endspiel. Im Mittelspiel werden die Anzahl der Steine nicht betrachtet, wohingegen sie am Ende einen sehr großen Teil der Bewertungsfunktion ausmachen.

5.1.1 Mittelspiel

allgemeines Positionsspiel:
In diesem Teil werden hauptsächlich Aspekte der klassischen Variante von Reversi betrachtet und die zusätzlichen Möglichkeiten dem Abschnitt "spezielles Positionsspiel" überlassen.
Einen besonders hohen Wert haben sogenannte *stabile Felder*, welche sich dadurch auszeichnen, dass sie von einem Gegner nicht mehr auf klassischem Wege übernommen werden können. Daran angrenzende sogenannte *schwache Nachbarn*, sollten hingegen nicht besetzt werden, da sie den Gegnern gerade die Möglichkeit geben, die *stabilen Felder* zu ergattern. Dementsprechend wird die Kontrolle eines *stabilen Feldes* belohnt mit $PosVal_g$ += 5 und die Kontrolle eines *schwachen Nachbarn* bestraft mit $PosVal_g$ -= 5. Man beachte jedoch folgendes: Wenn das stabile Feld einmal eingenommen ist, ist der Kampf um das Feld beendet und der *schwache Nachbar* verliert seinen Status als schwach. Ist das *stabile Feld* von einem selbst belegt, wird der *schwache Nachbar* sogar zu einem guten Feld, da er zusätzliche Stabilität bekommt. Belohnung mit $PosVal_G$ += 2.
Die Belegung *stabilen Felder* ist aufgrund der Existenz von override stones jedoch nicht endgültig. Weswegen der allgemeine Positionswert skaliert wird mit der Anzahl der gegnerischen Overrides und verbleibenden Bonusfelder. Wir erhalten als allgemeinen Stellungswert schließlich:

$$rPosVal_G := \frac{PosVal_G}{\#OppOverrides + 0.5 * \#Bonusfelder + 1}$$

spezielles Positionsspiel:
Hier wird versucht die speziellen Aspekte der Spielvariante zu bewerten. Zunächst gibt es analog zu dem allgemeinen Positionsspiel auch hier *schwache Nachbarn*, jedoch von Spezialfeldern. Einen sehr großen Wert haben Choice Felder. Weswegen die Besetzung angrenzender Felder abgestraft wird mit $PosVal_S$ -= 10. Einen etwas weniger wichtigen Wert haben Bonusfelder, demnach wird die Strafe geringer: $PosVal_S$ -= 3. Auch hier ist zu beachten, dass nach Einnahme der Spezialfelder, die Nachbarn wieder zu gewöhnlichen Feldern werden. Außerdem ist ein *stabiles Feld* auch nie ein *schwacher Nachbar*. Da expansion stones keine schwerwiegende Spielveränderung mit sich führen und die Belegung von Inversionsfelder sowohl 'gut' als auch 'schlecht' sein können, werden diese der Einfachheit halber als normale Felder gehandhabt. Der letzte Punkt ist die eigene Anzahl der override stones. Diese sind ein sehr großer Vorteil, da sie einem Mobilität sichern (siehe Abschnitt 'Mobilität') und hervorragende Chancen das Spiel zum Ende hin zu seinem Vorteil zu drehen. Diese werden also großzügig belohnt mit $PosVal_S$ += $10 * \#EigeneOverrides$.

Mobilität:
Die Mobilität ist ein entscheidender Faktor in dem Spiel, da man bei Auslauf seiner Zugmöglichkeiten die kostbaren override stones verwenden oder sogar aussetzen muss. Außerdem hat man bei vielen Möglichkeiten eine gute Chance, dass eine besonders gute Variante dabei ist und kann nicht in einen ungünstigen forcierten Ablauf hineingezwängt werden. Man behält demnach eine gewisse Initiative. Wir bezeichnen Züge die keinen override stone kosten als *kostenlose Züge*. Damit bekommt die Mobilität folgende Formel:

$$PosVal_M := -15 + 2 * \#kostenloseZüge$$

Also bei 7-8 möglichen Zügen wäre der Break-Even-Punkt erreicht, was bei einem potentiell recht großen Feld wahrscheinlich noch zu klein gewählt ist. Außerdem sind wir uns noch nicht ganz sicher, ob der lineare Zusammenhang gut gewählt ist. Das muss noch zu sammelnde Spielerfahrung zeigen. Insgesamt ergibt sich als Bewertungsfunktion im Mittelspiel:

$$PosVal := rPosVal_G + PosVal_S + PosVal_M$$

5.1.2 Endspiel

Im **Endspiel** vernachlässigen wir das Positionsspiel und konzentrieren uns auf den Punkt Mobilität und Feldkontrolle.

Wir setzen:

$$PosVal := \frac{\#kostenloseZüge}{\#freieFelder} * 30 + \frac{ProzentKontrolliert}{100/\#Spieler} * 20$$

Die Skalierung der kostenlosen Züge kommt daher, dass natürlicherweise zum Ende der Bauphase hin die maximalen Möglichkeiten geringer werden. Die Skalierung des kontrollierten Spielfeldanteils, kommt daher, dass z.B. 12,5% (=1/8) bei 8 Spielern und 50% (=1/2) bei 2 Spielern gleichwertig ist und dementsprechend auch gleich viel in die Funktion einfließen soll.
Um einen sanfteren Übergang von Mittelspiel zu Endspiel zu schaffen, fließen abhängig von der Anzahl an verbleibenden Spielzügen die beiden Bewertungsfunktionen unterschiedlich zusammen.

- >3 verbleibende Spielrunden -> $0.0 * Endgame + 1.0 * Middlegame$
- 3 verbleibende Spielrunden -> $0.1 * Endgame + 0.9 * Middlegame$
- 2 verbleibende Spielrunden -> $0.5 * Endgame + 0.5 * Middlegame$
- 1 verbleibende Spielrunde -> $1.0 * Endgame + 0.0 * Middlegame$

5.2 Bombing phase

Diese Funktion ist hingegen recht einfach gehalten. Wir zählen die Anzahl der Steine, welche 'gesprengt' werden müssen, damit der jeweilige Spieler den ersten Platz einnimmt. Also

$$PosVal := -(\#SteineBisPlatz1).$$

Befindet man sich auf dem ersten Platz, so zählt die Anzahl der Steine, die man mehr als der zweite Platz hat:

$$PosVal := \#EigeneSteine - \#SteinePlatz2$$

Als Beispiel: Befindet man sich auf dem dritten Platz und der erste Spieler hat 20 und der zweite Spieler hat 10 Steine mehr, dann liefert die Bewertungsfunktion $-30 = -(10 + 20)$ als Wert.
Wir differenzieren nicht zwischen den Plätzen 2-8, da diese alle für uns als 'verloren' gelten.