Michael-Ende-Gymnasium Tönisvorst

Schuljahr 2016 / 2017

Implementierung einer künstlichen Intelligenz am Beispiel von Minesweeper

Facharbeit im GK Informatik

Herr Schulz

Vorgelegt von

Vincent Hilla

Krefeld, April 17

1 Vorwort

Das Thema „Implementierung einer künstlichen Intelligenz am Beispiel von Minesweeper“ habe ich aus verschiedenen Gründen gewählt. Erstens wuchs in mir nach dem Informatikunterricht der EF, in dem die Programmierumgebung Greenfoot genutzt wurde, das Interesse, ein Computerspiel komplett ohne solche Hilfsmittel zu programmieren. Zudem wurde meine Aufmerksamkeit durch ein YouTube-Video von Scishow[1], welches das Lösen von Brettspielen durch KI behandelt, und diverse Anwendungen, wie Quickdraw[2] oder Handschrifterkennung[3] auf das Thema KI gelenkt.

Auch wenn eine KI für ein Computerspiel keine Vorteile im praktischen Leben bringt, hat das Themengebiet KI dennoch Relevanz, denn die Anwendungen sind vielfältig. Programme können durch die Automatisierung von Abläufen oder das eigenständige Lösen von Problemen den Alltag vereinfachen. Immer bessere Algorithmen sowie Programmierumgebungen vereinfachen das Entwickeln einer KI ständig. Beispielsweise kann mithilfe von TensorFlow einem Programm das Lernen beigebracht werden.

Bei TensorFlow handelt es sich um eine kostenlose Bibliothek zum Erstellen von komplexen neuronalen Netzwerken[4]. Generell werden dabei Ebenen aus Recheneinheiten gebildet. Jede Einheit erhält die gesamten Ergebnisse der vorherigen Ebene und berechnet mithilfe von einer Funktion und Gewichtungen für die einzelnen Einheiten der letzten Ebene einen neuen Wert, welcher an die nächste Ebene weitergegeben wird. Die erste Ebene besteht dabei nicht aus Recheneinheiten, sondern vielmehr aus Zahlen, die beispielsweise je ein Pixel eines Bildes darstellen. Irgendwann folgt die letzte Ebene, die ein Endergebnis festlegt. So kann bei der Handschrifterkennung jede Einheit dieser letzten Ebene für ein anderes Zeichen stehen und das Zeichen der Einheit, die das höchste Rechenergebnis liefert, gilt als das im Bild abgebildete Zeichen. Lernen könnte dieses Netz nun, indem es eine Eingabe in Form eines Bildes erhält und bei einer

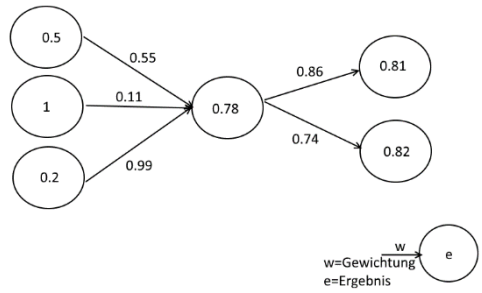

Abbildung 1: Beispielnetzwerk

[1] Vgl. SciShow: “The AI Gaming Revolution” (Video), online abrufbar unter: https://www.youtube.com/watch?v=Xhec39dVGDE, Stand: 28.02.2017.
[2] Vgl. Google: “Quick, Draw!” (Computerspiel), online abrufbar unter: https://quickdraw.withgoogle.com/, Stand: 28.02.2017.
[3] Vgl. Michael A. Nielsen, “Neural Networks and Deep Learning”, Determination Press, 2015, Kapitel 1, online abrufbar unter: http://neuralnetworksanddeeplearning.com/chap1.html, Stand: 28.02.2017.
[4] Vgl. Google: “TensorFlow”, online abrufbar unter: https://www.tensorflow.org/, Stand: 28.02.2017.

falschen Ausgabe über verschiedene Algorithmen die Gewichtungen in den einzelnen Recheneinheiten verändert.[5]

Auch wenn ich überlegt habe, dies zum Thema meiner Facharbeit zu machen, habe ich mich doch dafür entschieden, eigene Algorithmen zu finden, da die Implementierung eines gut funktionierenden Netzwerkes möglicherweise zu umfangreich würde und ein solches Netzwerk hinsichtlich Erfolgsrate und Rechenaufwand nicht die beste Lösung für Minesweeper darstellt.

[5] Vgl. Michael A. Nielsen, “Neural Networks and Deep Learning”, Determination Press, 2015, Kapitel 1, online abrufbar unter: http://neuralnetworksanddeeplearning.com/chap1.html, Stand: 28.02.2017.

2 Inhaltsverzeichnis

3 Einleitung

Diese Facharbeit behandelt das Implementieren, also das Einbauen einer künstlichen Intelligenz am Beispiel von Minesweeper.

„Künstliche Intelligenz (KI) beschäftigt sich mit Methoden, die es einem Computer ermöglichen, solche Aufgaben zu lösen, die, wenn sie vom Menschen gelöst werden, Intelligenz erfordern“[6]. Anders als die meisten Programme probiert eine KI Probleme selbstständig zu lösen, denn viele Programme lösen nicht Probleme, sondern setzen nur klare Anweisungen menschlicher Programmierer um[7].

3.1 Das Thema

Nachdem in der EF der Fokus auf dem Entwickeln eigener Computerspiele mit Java lag, passt Minesweeper als weiteres Spiel gut in diese Unterrichtsreihe. Dabei ist jedoch besonders, dass ich anders als in der EF nicht Greenfoot, sondern Eclipse als Programmierumgebung verwendet habe, da dies mehr Freiheit und Eigenständigkeit ermöglicht, aber auch weniger Hilfestellung bedeutet. Zudem wird auch im Unterricht der Q1 auf den Eclipse ähnlichen JavaEditor und nicht mehr auf Greenfoot gesetzt.

Da das Thema KI sehr groß ist, wird dieses auf die Implementierung in Minesweeper eingegrenzt. Auf Historie, gängige Algorithmen und heutige Anwendungen wird nur sehr begrenzt eingegangen. Der Fokus liegt vielmehr auf der Frage, wie schwer es ist, einem Computer beizubringen, ein Spiel zu gewinnen, wie Algorithmen dazu gefunden werden können und welche Vorteile die KI gegenüber einem Menschen aufweist. In erster Linie wird aber der Quelltext an sich erklärt.

3.2 Was ist Minesweeper?

Das Spiel Minesweeper kann heutzutage über den Windows-Store heruntergeladen werden[8]. Das in dieser Facharbeit verwendete Programm ist jedoch selbst geschrieben, weshalb auch die im Folgenden erklärten Spielregeln vom Original abweichen.

Aufgabe für den Spieler ist es, alle Minen auf dem Spielfeld zu markieren. Dazu kann er per Rechtsklick auf die Maus ein Feld mit einer Flagge kennzeichnen, wodurch dieses Feld auch nicht mehr per Linksklick anklickbar ist. Außerdem kann der Spieler die Kennzeichnung durch wiederholtes Klicken rückgängig machen und per Linksklick

---

[6] Springer Gabler Verlag: „Gabler Wirtschaftslexikon“, Stichwort „Künstliche Intelligenz (KI)“, online abrufbar unter: http://wirtschaftslexikon.gabler.de/Archiv/74650/kuenstliche-intelligenz-ki-v12.html, Stand: 28.02.2017.

[7] Vgl. SciShow: “The AI Gaming Revolution” (Video), online abrufbar unter: https://www.youtube.com/watch?v=Xhec39dVGDE, Stand: 28.02.2017.

[8] Vgl. Microsoft Studios: „Microsoft Minesweeper“ (Computerspiel), online abrufbar unter: https://www.microsoft.com/de-de/store/p/microsoft-minesweeper/9wzdncrfhwcn, Stand: 28.02.2017.

dieses Feld sich aufdecken lassen. Falls auf dem sich aufdeckenden Feld eine Mine liegt, ist die Runde verloren. Sonst wird auf dem Feld von nun an angezeigt, wie viele Minen diagonal, vertikal und horizontal angrenzen. Falls diese Zahl null beträgt, werden automatisch auch alle umliegenden Felder aufgedeckt. Beim Start einer neuen Runde sind alle Felder verdeckt. Beim erstmaligen Anklicken nach dem Rundenstart kann keine Mine erwischt werden, denn diese werden den Feldern erst nach dieser ersten Aktion zugeordnet. Wenn irgendwann alle Minen markiert sind und keine Nicht-Minenfelder markiert sind, ist die Runde gewonnen und es wird eine neue Runde gestartet. Das Spiel findet jedes Mal auf einem 16 mal 16 Felder großem Spielfeld mit 40 verdeckten und zufällig verteilten Minen statt. Dies entspricht dem Schwierigkeitsgrad „Fortgeschrittene“[9].

Das im Folgenden erwähnte, von mir geschriebene Programm existiert in zwei Versionen. Die verkürzte Version, die vollständig erklärt wird, beinhaltet alle wichtigen Funktionen, um Minesweeper spielbar zu machen und optional durch eine KI lösen zu lassen. Neben dieser Version liegt auch eine umfangreichere Version vor, die es ermöglicht, Minenmenge und Spielfeldgröße zu variieren. Zudem ist es dort möglich, verschiedene Einstellungen an der KI vorzunehmen sowie den momentanen Spielstand in einer Datei abzuspeichern und wieder zu laden. Jedoch ist diese Version deutlich umfangreicher, weswegen auf dessen Quelltext nicht eingegangen wird.

4 Umsetzung Minesweeper und KI

Im Folgenden wird erklärt, wie Minesweeper und die zugehörige KI in Java umgesetzt wurde. Dazu werden die drei Klassen, aus denen das Programm besteht, einzeln betrachtet. *Methoden* werden dabei mit hellblau, *Klassen* mit dunkelblau, *Variablen* mit braun und *Java Schlüsselwörter*[10] mit rot gekennzeichnet und kursiv gedruckt. Neben diesen Klassen benötigt das Programm mehrere Bilder, welche in einem separaten Ordner gespeichert sind.

Da im Unterricht *for*-each-Schleifen noch nicht behandelt wurden, diese aber häufig im Quelltext auftauchen, werden diese nun erklärt. Solche Schleifen dienen dazu, alle Objekte eines Arrays o.ä. durchzugehen: *for*(*Datentyp variablenname* : *array*) {}. In den geschweiften Klammern können nun Anweisungen folgen, die mit jedem Objekt aus dem Array ausgeführt werden. Dabei kann das momentane Objekt unter *variablenname* abgerufen werden.

[9] Gamesbasis: ”Minesweeper”, online abrufbar unter: http://www.gamesbasis.com/minesweeper.html, Stand: 28.02.2017.
[10] Vgl. Oracle: “Java Documentation“, online abrufbar unter: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html, Stand: 28.02.2017.

4.1 *Main*-Klasse

Diese Klasse ist dafür zuständig, die graphische Oberfläche, also das Fenster, welches sich auf dem Bildschirm öffnet und in dem das Spiel anschließend stattfindet, zu erstellen. Dies umfasst auch das Hinzufügen von *JButtons*, bei denen es sich um Knöpfe handelt, die durch das Anklicken Methoden aufgerufen und *Feld*-Objekten, die später erläutert werden, zum Spielfeld. Ebenfalls wird über diese Klasse kontrolliert, wann die Runde gewonnen ist und alle Felder werden über das Ende einer Runde benachrichtigt. Zudem gehört zu den Aufgaben dieser Klasse auch noch das Bereitstellen von Informationen zum Spielfeld für die KI und das Anweisen dieser zum Durchführen von Aktionen.

4.1.1 Startmethode des Programmes

Beim Start des Programmes wird die *main(String[] args)*-Methode aufgerufen. Diese ruft daraufhin in einem *try*-Block die *openGUI()*-Methode auf.

Try- und *catch*-Blöcke werden in Java benötigt, um *Exceptions* aufzufangen. Diese sind Ereignisse, die im Programm auftauchen können und - wenn nicht abgefangen - das gesamte Programm stoppen. Im *try*-Block werden immer Aktionen durchgeführt, die möglicherweise ein Ereignis erzeugen könnten, und im *catch*-Block stehen Anweisungen, falls ein solches auftritt. Dabei werden beim Auftauchen eines solchen alle noch nicht ausgeführten Anweisungen im *try*-Block übersprungen. In den runden Klammern, die auf *catch* folgen, werden die Ereignisse festgelegt, die durch diesen *catch*-Block aufgefangen werden und unter welcher Variable dieses Ereignis abgerufen werden kann, denn bei einer *Exception* handelt es sich um ein Objekt. Im Quelltext kann dies folgendermaßen aussehen: *try {…} catch (Exception e) {…}.*

An dieser Stelle ist das Auffangen nötig, da dies der letzte Ort ist, an dem alle noch nicht abgefangenen Ereignisse des gesamten Programmes abgefangen werden können. Falls es nun zu einem Ereignis kommen sollte, können über die Methode *printStackTrace()* der Klasse *Exception* in der Konsole wichtige Informationen über diese ausgegeben werden, wodurch es dem Nutzer ermöglicht wird, diese zukünftig zu verhindern oder dem Entwickler zu melden.

4.1.2 Graphische Oberfläche

Durch die *openGUI()*-Methode wird nun eine graphische Oberfläche erstellt.

Als erstes wird dafür ein *JFrame* benötigt. Dabei handelt es sich um ein Fenster, dass auf dem Bildschirm angezeigt wird. In diesem Fall besitzt das Fenster eine nicht veränderbare Größe von 475 x 490 Pixeln und befindet sich auf dem Bildschirm 100 Pixel weit von dem linken oberen Rand entfernt. Dies wird festgelegt durch die Methoden

frame.setBounds(int x, int y, int width, int height) und *frame.setResizeable(boolean resizeable)*. Anschließend sorgt die Methode *frame.setDefaultCloseOperation(int operation)* mit dem Parameter *JFrame.EXIT_ON_CLOSE* dafür, dass beim Schließen des Fensters das gesamte Programm beendet wird. Alle diese Methoden werden dabei auf ein vorher erstelltes Objekt, in diesem Fall mit dem Variablennamen *frame* der Klasse *JFrame* angewendet.

Ein *JFrame* benötigt immer eine einzelne Instanz, also ein Objekt, der Klasse *JPanel*. Dabei handelt es sich um einen Behälter, in dem weitere Elemente wie Knöpfe oder Texte angezeigt werden können. Dazu wird ein solches unter dem Variablenname *contentPane* erstellt und dem Fenster über *frame.setContentPane(Container contentPane)* hinzugefügt.

Jedes *JPanel* benötigt ein Layout, das festlegt, wie die Elemente in diesem Behälter angeordnet werden. In diesem Fall wird ein *BoxLayout* verwendet, welches alle Elemente unter einander platziert. Falls hingegen kein Layout festgelegt wird, wird standardmäßig ein *FlowLayout* verwendet. Dieses ordnet die Elemente ähnlich wie einen Text von links nach rechts an und nutzt dabei, falls nötig, auch mehrere Zeilen.

Dem *contentPane* werden nun zwei weitere Instanzen von *JPanel* hinzugefügt. Die eine, die unter der Variable *pSpielfeld* abrufbar ist, wird später das Spielfeld und die andere zwei Instanzen von *JButton*, mit denen die KI gesteuert werden kann, anzeigen.

Der eine Knopf wird die KI eine einzelne Aktion ausführen lassen. Der andere Knopf wird die KI fortlaufend alle 0,25 Sekunden eine Aktion ausführen lassen, bis er ein weiteres Mal angeklickt wird.

Um dies zu ermöglichen, muss ein *EventListener*, hier *MouseListener* verwendet werden. Dazu sind verschiedene Klassen nötig. Zu allererst eine Instanz von *JButton*, die in allen *MouseListenern*, die im Vorhinein dieser Instanz übergeben wurden, eine bestimmte Methode aufruft. Übergeben werden die *MouseListener* über die Methode *addMouseListener(MouseListener l)* der Klasse *JButton.* Aufgerufen wird beim Anklicken des Knopfes die Methode *mouseClicked(MouseEvent e)* aller hinzugefügten *MouseListener*. Dieser Methode wird als Parameter eine Instanz eines *EventObjects, hier MouseEvent* übergeben, welches Informationen über dieses Event bereitstellt. Bei einem *EventListener* handelt es sich jedoch um ein Interface, eine Klasse mit körperlosen Methoden. Dies bedeutet: Beim Erstellen eines Objektes dieser Klasse muss festgelegt werden, was welche der körperlosen Methoden tut. Dadurch können bei Klick auf einen bestimmten Knopf bestimmte Aktionen durchgeführt werden.

Nachdem alle wichtigen Elemente des Fensters hinzugefügt wurden, muss dieses Fenster noch sichtbar gemacht werden. Dies passiert über die Methode *frame.setVisible(boolean b)*. Damit ist das Fenster fertig, die globale Variable *ki* wird mit *new KI()* initialisiert und es wird die Methode *starteNeueRunde()* aufgerufen.

4.1.3 Erstellung des Spielfeldes

Die Methode *starteNeueRunde()* fügt das bereits erwähnte Spielfeld dem vorher erstellten *JPanel* hinzu. Dazu werden anfangs die globalen Variablen *felderMarkiert* und *minenMarkiert* auf null gesetzt. Diese speichern, wie viele Felder markiert und wie viele davon Minenfelder sind. Danach wird über *pSpielfeld.removeAll()* das *JPanel*, welches das Spielfeld beinhaltet, geleert, um Rückstände einer eventuellen letzten Runde zu entfernen.

Das Spielfeld wird später aus *Feld* Objekten bestehen, die ähnlich wie *JButton* aber mit Bild funktionieren. Diese müssen in einem 16 mal 16 großen Feld angeordnet werden, wofür sich ein *GridLayout* anbietet. Dieses ordnet alle Elemente in Reihen und Zeilen mit einem festgelegten Abstand zwischen den einzelnen Elementen an. Um dies umzusetzen, wird erst einmal festgelegt, dass das *pSpielfeld* ein solches Layout mit 16 Reihen und Spalten, sowie einem Mindestabstand von einem Pixel zwischen diesen besitzt. Dies passiert über die Methode *pSpielfeld.setLayout(new GridLayout(16, 16, 1, 1))*. Anschließend werden 256 Objekte der Klasse *Feld* dem *JPanel* über die Methode *pSpielfeld.add(Component comp)* mithilfe von Schleifen hinzugefügt.

Nun bietet sich es an, die einzelnen Objekte nochmals extra in einem zweidimensionalen Array mit dem Variablennamen *spielfeld* zu speichern, um schnell auf bestimmte Objekte zugreifen zu können, was für die KI später wichtig wird. Daher wird vor den Schleifen noch solch ein Array erstellt. Als Schleifen wurden hier zwei ineinander verschachtelte *for*-Schleifen verwendet, um die Objekte im Array nach Reihe und Säule beziehungsweise x- und y-Koordinate im Array speichern zu können. Die äußere Schleife zählt von Reihe 0 bis 16 und die innere für jede Reihe Feld 0 bis 16 durch.

Anschließend wird die Methode *reload()* des globalen Objektes *ki* aufgerufen, wodurch dieses das fertige Array abruft und die KI somit Zugang zum Spielfeld hat.

Nachdem alle Objekte der Klasse *Feld* dem *JPanel* und Array hinzugefügt wurden, müssen diese alle noch anklickbar gemacht werden. Dafür wird die Methode *addNewMouseListener()* der Klasse *Feld* auf jedes Objekt angewendet. Nun kann die Runde beinahe beginnen, jedoch besitzen diese Objekte noch kein Bild. Um dies zu ändern, wird die Methode *redraw()* verwendet.

Diese Methode ruft für jedes Objekt der Klasse *Feld* die gleichnamige Methode *redraw()* dieser Klasse auf.

4.1.4 Platzierung der Minen

Nachdem der Spieler nun zum ersten Mal in der Runde eine Kachel auf dem Spielfeld angeklickt hat, ruft dieses angeklickte *Feld*-Objekt die Methode *plaziereMinen()* auf.

Diese Methode beginnt damit, alle Objekte der Klasse *Feld* darüber zu benachrichtigen, dass die Minen nun platziert sind, sodass diese Methode nicht weitere Male aufgerufen wird. Anschließend wird ein Objekt der Klasse *Random* unter der Variable rand erstellt. Dieses ermöglicht es anschließend über die Methode *rand.nextInt(int bound)* zufällige ganze Zahlen zwischen 0 und *bound* zu generieren.

Das Ziel liegt nun darin, 40 Minen auf dem Spielfeld zu verteilen. Dafür kann das *spielfeld*-Array verwendet werden, um die Minen mithilfe von zwei Zufallszahlen zwischen 0 und 16 zufällig zu platzieren. Das *Feld*-Objekt an den zufälligen Koordinaten wird dann über die Methode *setMine(boolean isMine)* dessen Klasse als Mine markiert. Falls dieses *Feld*-Objekt jedoch bereits angeklickt oder als Mine markiert wurde, wird dieses nicht (nochmals) als Mine markiert, sondern ein neues *Feld*-Objekt dafür ausgewählt. Zur Überprüfung werden die Methoden *isMine()* und *isAufgedeckt()* der Klasse *Feld* verwendet.

4.1.5 Anklicken von Feldern

Wenn eine *Feld*-Objekt angeklickt wird und an dieses keine Minen angrenzen, sollen automatisch auch alle angrenzenden *Feld*-Objekte angeklickt werden. Sonst wird im Bild des *Feld*-Objektes angezeigt, wie viele Minen angrenzen. Dazu muss aber jedes *Feld*-Objekt auf alle angrenzenden *Feld*-Objekte und die Menge an angrenzenden Minen zugreifen können.

Dies wird im nächsten Schritt der Methode *plaziereMinen()* ermöglicht, indem für jedes *Feld*-Objekt jeder Nachbar aus dem *spielfeld*-Array bestimmt wird und dabei gezählt wird, wie viele von diesen Nachbarn Minenfelder darstellen. Anschließend werden die Methoden *setNachbarn(Feld[] nachbarn)* und *setMinenNahe(int minenNahe)* der Klasse *Feld* verwendet, um jenen *Feld*-Objekten alle ihre Nachbarn und die Menge an angrenzenden Minen zukommen zu lassen.

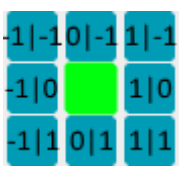

Abbildung 2

Die Nachbarn werden wie folgt bestimmt: In der Abbildung 2 ist dargestellt, wie die Positionen der einzelnen Nachbarn mithilfe von zwei Zahlen relativ zum mittleren, zu untersuchenden *Feld*-Objekt angegeben werden können. Im Quelltext werden diese Koordinaten mithilfe von den zwei Arrays *x* und *y* gespeichert. Anschließend wird

von der Koordinate der mitleren Kachel auf die Koordinaten der einzelnen Nachbarn geschlossen.

Falls eine dieser Koordinaten außerhalb des Spielfeldes liegt, bedeutet dies, dass es diesen Nachbarn nicht gibt, da sich die mittlere Kachel am Rand des Spielfeldes befindet. Für alle Koordinaten, die innerhalb des Spielfeldes liegen, wird mit dem *spielfeld*-Array das an den gegebenen Koordinaten liegende *Feld*-Objekt ermittelt. Ebenfalls wird über die Methode *isMine()* überprüft, ob dieses Objekt eine Mine darstellt und gegebenenfalls die Variable *minenNahe*, welche die Menge angrenzender Minen für das gerade untersuchte Feld speichert, erhöht.

Da der mittleren Kachel ein Array aus Objekten der Klasse *Feld* als Nachbarn übergeben wird und ein Array eine feste Größe hat, aber die Menge an Nachbarn anfangs nicht bekannt ist, müssen die Nachbarn anders abgespeichert werden. Dafür wird eine *ArrayList* genutzt.

Bei dieser handelt es sich gewissermaßen um ein Array mit einer variablen Menge an aufnehmbaren Objekten. Verwaltet werden diese jedoch mithilfe von Methoden und nicht wie bei Arrays mithilfe von eckigen Klammern. Beim Erstellen der *ArrayList* muss ähnlich wie beim Array der Typ der speicherbaren Objekte angegeben werden. Dies passiert bei der *ArrayList* mit einem „kleiner als“-Zeichen gefolgt von dem Typ gefolgt von einem „größer als“-Zeichen: *new ArrayList<Feld>().* Über die Methode *toArray(T[] a)* der Klasse *ArrayList* kann anschließend die Liste an Nachbarn in ein Array umgewandelt und dem untersuchten *Feld*-Objekt übergeben werden. *T* Steht hierbei wie eine Variable für den anfangs festgelegten Typ.

Nun ist Minesweeper für den Spieler so gut wie spielbereit. Jedoch muss das Programm noch überprüfen, wann die Runde gewonnen oder verloren ist. Ebenfalls wurden die Methoden, die aufgerufen werden, falls die *JButtons* zum Steuern der KI angeklickt werden, noch nicht erklärt.

4.1.6 Rundenende

Um zu überprüfen, wann die Runde gewonnen oder verloren ist, werden die globalen Variablen *minenMarkiert* und *felderMarkiert* benötigt. Wie bereits gesagt, sind diese am Anfang jeder Runde gleich null und während des Spielverlaufs kann der Spieler *Feld*-Objekte mit Flaggen markieren. Dadurch werden diese Variablen mithilfe der Methoden *erhöheFelderMarkiert(), veringereFelderMarkiert(), erhöheMinenMarkiert()* und *veringereMinenMarkiert()* verändert.

Da die Feld-Objekte immer erst *felderMarkiert* und dann *minenMarkiert* ändern, müssen nur die Methoden *erhöheMinenMarkiert()* und *veringereMinenMarkiert()* überprüfen, ob

der Spieler gewonnen hat und in diesem Fall die Methode *spielGewonnen()* aufrufen. Es wird also in beiden überpüft, ob *minenMarkiert* gleich *felderMarkiert* und gleich der globalen Variable *MINEN*, welche die Gesammtmenge an Minen angibt, ist.

Ob das Spiel verloren wurde, muss durch die *Main*-Klasse nicht überprüft werden, denn dies erledigen die *Feld*-Objekte. Wenn auf einem solchen eine Mine liegt und dieses angeklickt wird, wird die Methode *spielVerloren()* aufgerufen.

Sowohl *spielGewonnen()* als auch *spielVerloren()* benachrichtigen anfangs alle *Feld*-Objekte über das Ende des Spieles. Dadurch ändern manche ihr Bild, um die Position von unmarkierten Minen sowie richtigen und falschen Flaggen anzuzeigen. Damit der Spieler nun weiß, dass er gewonnen oder verloren hat, wird ein Dialogfeld genutzt.

Die Klasse *JOptionPane* bietet für Dialogfelder verschiedene Methoden, über die beispielsweise um Eingabe, Ja / Nein Antwort oder auch Bestätigung einer Nachricht gefordert werden kann. Das Besondere an diesen Methoden ist, dass diese *static* aufgerufen werden. Dies bedeutet, dass diese ohne Instanz der Klasse angewendet werden können. In diesem Fall wird die Methode *JOptionPane.showConfirmDialog(Component parentComponent, Object message, String title, int optionType)* verwendet. Diese öffnet ein einfaches Fenster mit Titel, einer Nachricht und einem Knopf, auf dem „OK“ steht.

Das Programm wird erst dann weiter ausgeführt, wenn der Nutzer den Dialog geschlossen hat. Sobald dies passiert ist, wird die Methode *starteNeueRunde()* aufgerufen und somit eine neue Runde gestartet.

4.1.7 KI-Interaktion

Im Zusammenhang mit der KI sind nun verschiedene Methoden vorhanden. Die get-Methode *getSpielfeld()* ermöglicht es der KI, das Spielfeld abzurufen, um verschiedene Methoden auf die darin gespeicherten *Feld*-Objekte anzuwenden.

Für den Nutzer liegen zwei Knöpfe zum Steuern der KI vor. Der eine lässt diese eine Aktion ausführen, also ruft die Methode *doAktion()* auf. Diese Methode gehört zur Klasse *KI* und wird auf die globale Variable *ki* angewendet.

Der andere Knopf lässt die KI alle 0,25 Sekunden eine Aktion ausführen, bis er nochmals angeklickt wird. Dies passiert, indem er die Methode *autoKI()* aufruft. Dieser Knopf kann entweder „Stopp“ oder „Daueraktion“ anzeigen. Daher wird in dieser Methode zuerst geprüft, ob auf dem Knopf „Daueraktion“ steht. Falls dies der Fall ist, wird dieser Text in „Stopp“ verändert und die Methode *ki.start()* aufgerufen. Sollte hingegen etwas anderes auf dem Knopf stehen, wird dieser Text in „Daueraktion“ geändert und die Methode

ki.stopp() aufgerufen. Der Zugriff auf den Text erfolgt über get- und set-Methoden der Klasse *JButton*.

4.1.8 Besonderheit und Attribute

Diese Klasse hat eine Besonderheit. Wie bereits erwähnt, gibt es in Java Methoden, die *static* aufgerufen werden können. Diese Klasse besteht jedoch nur aus *static*-Methoden und auch alle globalen Variablen sind *static*. Dies hat den Vorteil, dass andere Objekte schnell auf die Methoden dieser Klasse zugreifen können. Beispielsweise kann die KI einfach *Main.getSpielfeld()* aufrufen. Sollte diese Methode nun nicht *static* sein, wäre dieser Methodenaufruf nicht möglich und die KI müsste auf eine Instanz der *Main*-Klasse in Form einer Variable zurückgreifen. Das gleiche Problem hätten auch alle Objekte der Klasse *Feld*, denn diese müssen ebenfalls auf verschiedene Methoden der *Main*-Klasse zugreifen.

Wie auch viele andere Klassen besitzt diese mehrere globale Variablen. Dies muss so sein, um zu ermöglichen, dass verschiedene Methoden auf diese zugreifen können. Besonders sind *int MINEN* und *int SIZE*, denn diese sind *public*, also haben alle Objekte ohne get- oder set-Methoden Zugriffsmöglichkeiten. Jedoch sind beide Variablen als *final* deklariert, was bedeutet, dass dessen Werte nicht veränderbar sind.

4.2 *Feld*-Klasse

Diese Klasse stellt ein Feld auf dem Spielfeld dar. Hierbei handelt es sich um eine von *JLabel* erbende Klasse. *JLabel* sind graphische Elemente, die normalerweise kleine Texte oder Bilder zeigen. In diesem Fall wird ein Bild angezeigt und es werden bestimmte Aktionen beim Anklicken ausgeführt, welche in dieser Klasse festgelegt sind.

Vererbung ermöglicht in Java, dass eine Klasse alle Methoden und Attribute einer anderen Klasse übernimmt. Jede Klasse kann dabei aber nur von einer anderen Klasse erben. Die Klasse *Feld* kann dank Vererbung genauso wie ein *JLabel* verwendet werden, aber er sind weitere Funktionen hinzufügbar, die ein *JLabel* nicht besitzt.

4.2.1 Die Objekterstellung

Der Konstruktor initialisiert verschiedene globale Variablen. Dadurch wird gesichert, dass das Feld anfangs nichts Besonderes anzeigt, er also nur blau ist.

Die *Main*-Klasse ruft, nachdem sie ein Objekt dieser Klasse erstellt hat, dessen Methode *addNewMouseListener()* auf. Diese ermöglicht, dass dieses *JLabel* wie ein *JButton* funktioniert, indem ein *EventListener* hinzugefügt wird. Falls das *Feld*-Objekt nun angeklickt wird, tritt ein Event auf, wodurch die Methode *onClick(int button)* aufgerufen wird. Diese benötigt als Parameter die Nummer der Maustaste, mit der das *JLabel* angeklickt wurde. Da zu einem Event immer ein *EventObject*, in diesem Falle das

MouseEvent gehört, kann über die Methode *getButton()* dieser Klasse die Nummer der Maustaste ermittelt werden.

4.2.2 Ablauf beim Anklicken

Die Methode *onClick(int button)* überprüft als erstes über den *boolean aufgedeckt*, ob das Feld bereits angeklickt wurde. Falls dies der Fall sein sollte, passiert nichts. Sonst wird abhängig von dem Parameter *button* die Methode *linksklick()* oder *rechtsklick()* aufgerufen. Anschließend muss möglicherweise das Bild des *Feld*-Objektes aktualisiert werden, weswegen *redraw()* aufgerufen wird. Welches Bild momentan angezeigt werden soll, ist in der Variable state festgelegt. Dazu liegen verschiedene Variablen, wie *VERDECKT* oder *FLAGGE*, vom Typ *int* vor, dessen Werte *state* annimmt und über die anschließend überprüft werden kann, welches Bild angezeigt werden muss.

Die Methode *rechtsklick()* ist zuständig für das Markieren und Entmarkieren eines *Feld*-Objektes mit einer Flagge. Falls nun *state FLAGGE* entspricht, wird *state* auf den Wert von *VERDECKT* geändert und die Methode *Main.veringereFelderMarkiert()* aufgerufen. Sollte es sich bei dem *Feld*-Objekt um ein Minenfeld handeln, wird zusätzlich *Main.veringereMinenMarkiert()* aufgerufen. Entspricht *state* anfangs hingegen *VERDECKT*, wird dieser Variable der Wert von *FLAGGE* zugeordnet und die Methode *Main.erhöheFelderMarkiert()* sowohl abhängig von *boolean mine* *Main.erhöheMinenMarkiert()* aufgerufen.

Falls nun mit der linken Maustaste geklickt wurde, wird *linksklick()* aufgerufen und das *Feld*-Objekt wird, falls nicht mit einer Flagge markiert, aufgedeckt. Dazu wird als erstes die Variable *aufgedeckt* auf *true* gesetzt. Da beim ersten Klick die Minen noch nicht zugewiesen sind, wird über den *boolean minenPlaziert* entschieden, ob die Methode *Main.plaziereMinen()* aufgerufen werden muss. Da *aufgedeckt* bereits vor Aufruf zu *true* geändert wurde, kann diese Methode dieses *Feld*-Objekt nicht mehr zu einem Minenfeld machen und somit wird sichergestellt, dass beim ersten Klick keine Mine erwischt wird. Anschließend wird überprüft, ob es sich bei diesem *Feld*-Objekt um ein Minenfeld handelt. Falls dies der Fall ist, wird *state* auf den Wert von *MINE_CLICKED* geändert und *Main.spielVerloren()* aufgerufen. Sollte das *Feld*-Objekt jedoch kein Minenfeld sein, wird *state* auf den Wert von *ZAHL* gesetzt und, falls keine Minen angrenzen, werden auch alle benachbarten *Feld*-Objekte über die Methode *onClick(int button)* aufgedeckt.

4.2.3 Aktualisieren und Anzeigen des Bildes

An verschiedenen Stellen kam nun bereits die Methode *redraw()* vor. Diese aktualisiert nach der Vorgabe von *state* das Bild eines *Feld*-Objektes. Dafür werden die anfangs erwähnten in einem anderen Ordner mit dem Namen „res“ gespeicherten Bilddateien verwendet.

Da die Klasse *Feld* von *JLabel* erbt, kann dessen Methode *setIcon(Icon icon)* verwendet werden. Bei einem *ImageIcon* handelt es sich um ein eher kleines Bild, dass in diesem Fall als Parameter icon verwendet wird. Die Bilder liegen jedoch in Form einer Datei vor, müssen also erst einmal zu einem *ImageIcon* werden. Dazu wird der Konstruktor *ImageIcon(String filename)* verwendet. Als *filename* muss jedoch der Speicherort des Bildes angegeben werden, was über die Methode *getClass().getResource(String name)* ermöglicht wird. Für den Parameter *name* wird hier immer „/res/dateiname.png“ verwendet, denn alle Bilder liegen im Ordner „res“ des Programmes. Da der Ordner in den Programmdaten liegt und somit der genaue Ort nicht bekannt ist, muss auf diesen mithilfe von der Java Methode *getResource()* zugegriffen werden. So kann über den Ort des Programmes auf den des Ordners geschlossen werden.

Da das Feld in der Lage sein muss, unterschiedliche Bilder in Abhängigkeit von *state* anzuzeigen, wird ein *switch*-Block verwendet, um festzulegen, was im Parameter *name* für „dateiname“ eingesetzt wird.

Ein *switch*-Block ist vergleichbar mit mehreren *if*-Abfragen hintereinander, aber anstelle von allgemeinen Bedingungen wird überprüft, ob ein Wert, der in den Klammern nach dem *switch*-Befehl steht, gleich dem Wert hinter einem *case*-Befehl ist. Ist dies der Fall, werden die Anweisungen im *case*-Block ausgeführt. Dabei gibt es jedoch die Besonderheit, dass ab diesem *case*-Block auch die Anweisungen in allen folgenden Blöcken ausgeführt werden, weswegen - um dies zu verhindern - oft am Ende jedes *case*-Blockes *break* oder *return* steht. Dabei beendet *break* nur den *switch*-Block und *return* die gesamte Methode. Die einzelnen *case*-Blöcke unterscheiden sich nur in dem Parameter, der beim Generieren des *ImageIcons* verwendet wird.

4.2.4 Weitere Funktionen und Attribute

Wenn eine Runde gewonnen oder verloren ist, wird von der *Main*-Klasse bei allen Feldern die Methode *spielzuende()* aufgerufen. Diese sorgt dafür, dass angezeigt wird, welche Felder richtig und welche falsch markiert sind. Ist ein Minenfeld nicht markiert, wird *state* auf den Wert von *MINE* gesetzt, falls ein solches markiert ist, wird dessen *state* auf den Wert von *FLAGGE_RICHTIG* gesetzt. Falls ein Feld, dass keine Mine beinhaltet, markiert ist, wird dessen *state* auf den Wert von *FLAGGE_FALSCH* gesetzt. Anschließend wird *redraw()* aufgerufen, um das neue Bild des Feldes auch anzuzeigen.

Die Klasse *Feld* besitzt verschiedene get- und set-Methoden, die sowohl für die Klasse *Main* als auch für die Klasse *KI* wichtig sind. Abgesehen von der Methode *getMinenNahe()* funktionieren alle Methoden ganz normal. Diese jedoch gibt den Wert „9999“ wieder, falls das Feld nicht aufgedeckt ist, denn so wird verhindert, dass die KI an Informationen gelangt, die der Spieler nicht erhalten kann.

Genauso wie die Klasse *Main* hat auch die *Feld*-Klasse verschiedene globale Variablen. Diese umfassen ein Array, welches alle Nachbarfelder beinhaltet, mehrere *int*-Variablen dessen Wert *state* annehmen soll, die Menge an angrenzenden Minen und verschiedene *boolean*-Variablen.

4.3 *KI*-Klasse

Für die KI zuständig ist die *KI*-Klasse. In dieser Klasse befinden sich alle Algorithmen, die verwendet werden, um einen nächsten Spielzug zu bestimmen. Über eine Instanz dieser Klasse wird es der *Main*-Klasse ermöglicht, Aktionen herbeizurufen.

4.3.1 Attribute und Konstruktor

Erst einmal liegen verschiedene wichtige globale Variablen vor. Der *boolean automatisch* gibt an, ob momentan automatisch Aktionen ausgeführt werden sollen. Über das aus Objekten der Klasse *Feld* bestehende zweidimensionale Array *spielfeld* kann auf einzelne *Feld*-Objekte zugegriffen werden und *int minenGesamt* sowie *int felderGesamt* speichern Minen- und Feldmenge des Spielfeldes.

Der Konstruktor initialisiert diese globalen Variablen. Die Werte *minenGesamt* sowie *felderGesamt* werden von der *Main*-Klasse abgerufen und das Array *spielfeld* wird auf null gesetzt, da dieses erst später durch die Methode *reload()* zugewiesen wird. Durch *reload()* wird *spielfeld* auf den Wert von *Main.getSpielfeld()* gesetzt.

4.3.2 Methoden für automatische Aktionen

Die Methode *stopp()* ändert *automatisch* in *false,* während *start()* diese Variable in *true* ändert und anschließend die Methode *autoAct()* aufruft.

Da in Java erst dann die nächste Aktion ausgeführt wird, wenn die vorherige abgeschlossen ist, darf diese Methode nicht einfach dauerhaft 0.25 Sekunden warten, um danach eine Aktion durchführen. Dies würde den Rest des Programmes vollständig stoppen. Jedoch gibt es einen Ausweg, nämlich Multithreading. Dabei können verschiedene Aktionen zur selben Zeit ausgeführt werden, indem verschiedene Aktionsketten genutzt werden. Dazu wird eine Instanz der Klasse *Thread* erstellt, wobei die Aktionen im Körper von dessen Methode *run()* festlegt werden müssen. Diese werden so gewählt, dass *run()* nun wiederholt *doAktion()* ausführt und anschließend 0.25 Sekunden wartet, bis *automatisch* nicht mehr *true* ist. Warten kann eine Aktionskette über die Methode *sleep(long millis)* der Klasse *Thread*. Nach dem Erstellen des *Thread*-Objektes muss noch die Methode *start()* auf dieses angewendet werden, damit die *run()*-Methode aufgerufen wird.

Einen Thread warten zu lassen, kann eine *Exception* werfen, weswegen diese hier aufgefangen und ausgegeben würde. Ebenfalls kann die Methode *doAction()* zu einer

Exception führen, wenn beispielsweise versucht wird, Methoden auf eine leere Objektvariable anzuwenden. Dies kann u. a. passieren, wenn auf die Liste an Nachbarn eines Feldes zugegriffen wird, das noch keine Nachbarliste zugewiesen bekam. Auch wenn letzteres Ereignis hier nicht aufgefangen werden muss, macht dies dennoch Sinn, damit beim Auftreten das Programm dennoch weiter ausgeführt wird.

4.3.3 Methode zum Anfordern einer Aktion

Um eine Aktion auszuführen, wird die Methode *doAction()* verwendet. Diese ist dafür zuständig, die verschiedenen aktionsfindenden Methoden aufzurufen. Falls eine von diesen eine Aktion findet, wird diese getätigt und *doAction()* gestoppt. Dies passiert, um zu garantieren, dass immer nur einzelne Aktionen ausgeführt werden.

Als erstes wird überprüft, ob die Minen bereits platziert sind, indem versucht wird, die Nachbarn eines Feldes abzurufen. Sollte dies nicht möglich sein, wird zufällig ein *Feld*-Objekt angeklickt. Sonst wird als nächstes die Methode *doLogikFeldnachbarn()* aufgerufen. Findet diese keine Aktion, wird *doLogikMinenmenge()* aufgerufen. Wurde immer noch keine Aktion ausgeführt, wird *doPatternLogik()* aufgerufen und, falls wieder keine Aktion gefunden wurde, auf *getChancen()* zurückgegriffen. Diese Methode liefert eine *HashMap* mit den Positionen der *Feld*-Objekte, die am wahrscheinlichsten eine Mine sowie keine Mine sind. Ob keine Aktion ausgeführt wurde, kann über den Rückgabewert vom Typ *boolean* der ersten drei Methoden geprüft werden.

Eine *HashMap* ermöglicht es, einen Schlüssel mit einem Objekt zu verbinden. Über die Methode *put(K key, V value)* kann dabei ein solches Paar hinzugefügt und über *get(Object key)* das zum Schlüssel gehörende Objekt abgerufen werden. *K* und *V* stehen ähnlich wie bei der *ArrayList* für die bei der Erstellung der *HashMap* festgelegten Klassentypen des Schlüssel und zugehörigen Wertes.

Nun muss entschieden werden, ob das Feld mit der höchsten Wahrscheinlichkeit markiert oder das Feld mit der geringsten angeklickt werden soll.

Dazu wird von „1“ die höchste Wahrscheinlichkeit abgezogen und die Differenz mit der geringsten Wahrscheinlichkeit verglichen. Falls nun der errechnete Wert geringer oder gleich der kleinsten Wahrscheinlichkeit ist, wird das *Feld*-Objekt mit der höchsten Wahrscheinlichkeit markiert, sonst das *Feld*-Objekt mit der geringsten angeklickt. Denn wenn eine Wahrscheinlichkeit sehr nahe an „1“ liegt, ist dieses *Feld*-Objekt vermutlich eine Mine, während, wenn jene nahe an „0“ liegt, es sich bei diesem eher nicht um eine Mine handelt.

4.3.4 Algorithmus zur Betrachtung der Nachbarn eines Feldes

Die als erstes aufgerufene Methode *doLogikFeldnachbarn()* probiert, aufgrund der an ein *Feld*-Objekt angrenzenden *Feld*-Objekte eine Aktion zu finden. Dazu werden alle aufgedeckten *Feld*-Objekte mit mindestens einer angrenzenden Mine betrachtet.

Als erstes wird für das im Moment betrachtete *Feld*-Objekt bestimmt, wie viele verdeckte und markierte *Feld*-Objekte angrenzen, was in den Variablen *verdeckteFelder* und *markierteFelder* gespeichert wird. Dank der Methode *getNachbarn()* der Klasse *Feld* können zur Ermittlung dieser Werte einfach alle angrenzenden *Feld*-Objekte abgerufen und betrachtet werden.

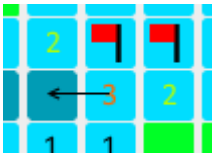

Abbildung 3

Sollte nun die Summe der verdeckten und markierten angrenzenden *Feld*-Objekte der Gesamtzahl der angrenzenden Minen entsprechen, können alle verdeckten *Feld*-Objekte als Minen angesehen und somit durch die Methode *onClick(3)* der Klasse *Feld* durch eine Flagge markiert werden. Da das Markieren von *Feld*-Objekten als Aktion gilt, wird *true* zurückgegeben. Ein Beispiel ist in Abbildung 3 zu sehen.

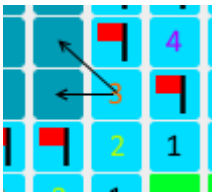

Abbildung 4

Wenn hingegen die zuletzt genannte Bedingung nicht zutrifft, wird überprüft, ob die Menge der angrenzenden markierten *Feld*-Objekte der Gesamtzahl an angrenzenden Minen entspricht, denn dann kann davon ausgegangen werden, dass alle restlichen verdeckten *Feld*-Objekte keine Minen beinhalten, weswegen diese über deren Methode *onClick(1)* aufgedeckt werden. Ein Beispiel ist in Abbildung 4 dargestellt.

Sollte diese Bedingung jedoch ebenfalls nicht zutreffen, muss das nächste *Feld*-Objekt betrachtet werden, bis irgendwann, wenn alle *Feld*-Objekte überprüft wurden, *false* zurückgegeben wird.

4.3.5 Algorithmus zur Betrachtung der gesamten Minenmenge

Die zweite aufgerufene Methode *doLogikMinenmenge()* probiert, aufgrund der Gesamtmenge an Minen auf dem Spielfeld neue Aktionen zu finden. Dazu wird als erstes ermittelt, wie viele markierte und verdeckte *Feld*-Objekte existieren, was in den Variablen *felderMarkiert* und *felderVerdeckt* gespeichert wird. Ermittelt werden diese Werte hier über das Array *spielfeld*, indem darin jedes einzelne *Feld*-Objekt betrachtet wird.

Sollte nun die Menge an markierten Minen größer als die Gesamtmenge an Minen sein, bedeutet dies, dass *Feld*-Objekte, die keine Minenfelder sind, markiert wurden. Folglich müssen alle markierten *Feld*-Objekte entmarkiert werden und *true* wird zurückgegeben,

da eine Aktion ausgeführt wurde. Zu falsch markierten *Feld*-Objekten kann es durch das Markieren aufgrund von Wahrscheinlichkeitswerten kommen.

Sollte andererseits die Gesamtmenge an Minen abzüglich der markierten Minen der Menge an verdeckten *Feld*-Objekten entsprechen, bedeutet dies, dass es sich bei all diesen *Feld*-Objekten um Minen handelt. Logischerweise werden anschließend über deren Methode *onClick(3)* alle diese *Feld*-Objekte markiert und es wird *true* zurückgegeben, da eine Aktion ausgeführt wurde.

Sollte die Gesamtmenge an Minen abzüglich der markierten Minen gleich „0“ sein, würde dies bedeuten, dass alle restlichen *Feld*-Objekte keine Minen sind und somit angeklickt werden sollten. Jedoch ist das Spiel zu Ende, sobald alle Minen markiert wurden, weswegen dieser Fall nicht betrachtet werden muss.

Am Ende der Methode wird *false* zurückgegeben, falls diese nicht schon früher durch *return true* beendet wurde.

4.3.6 Algorithmus zum Suchen von Mustern

Die dritte Methode *doPatternlogik()* sucht nach zwei verschiedenen Mustern. Dazu wird als erstes ein zweidimensionales *int*-Array erstellt, welches das Spielfeld abbilden soll. Eine Stelle im Array gehört nun immer zu einem Objekt aus dem *spielfeld*-Array. Daher werden alle Objekte aus diesem durchgegangen, um dem neuen Array die gewünschten Werte zuzuweisen. Ist das zugehörige *Feld*-Objekt aufgedeckt, wird die Menge an angrenzenden Minen mit unbekannter Position gespeichert. Sollte dieses markiert sein wird der Wert der Variable *Feld.FLAGGE* verwendet und, falls das *Feld*-Objekt verdeckt ist, wird der Wert von *Feld.VERDECKT* angenommen (Siehe Anhang Abbildung 1 für ein Beispiel).

Folgende zwei Muster kommen in Minesweeper häufig vor und werden deshalb gesucht:

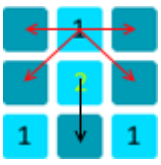

Abbildung 5

Beim ersten Muster folgt auf eine „1“ eine „2“ und an diese grenzen zwei aufgedeckte und ein verdecktes *Feld*-Objekt an, die jedoch nicht an die „1“ angrenzen. Da nur eines der *Feld*-Objekte, die sowohl an die „1“ als auch an die „2“ angrenzen, eine Mine sein kann, muss das verdeckte *Feld*-Objekt, welches an die „2“, aber nicht an die „1“ angrenzt, eine Mine sein und kann somit im nächsten Spielzug markiert werden. In Abbildung 5 handelt es sich dabei um das mittlere *Feld*-Objekt in der unteren Reihe.

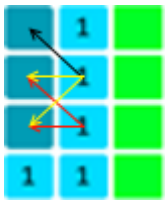

Abbildung 6

Beim zweiten Muster liegen zwei Einsen so nebeneinander, dass beide an die genau gleichen verdeckten *Feld*-Objekte angrenzen. Folglich muss eines dieser eine Mine darstellen. Jedoch grenzt eine der Einsen noch an weitere verdeckte *Feld*-Objekte an, welche somit sicher keine Minen beinhalten, also angeklickt werden können. In Abbildung 6 handelt es sich dabei um das *Feld*-Objekt in der linken oberen Ecke.

Um dies im Programm umzusetzen, werden alle aufgedeckten *Feld*-Objekte des Spielfeldes untersucht. Sollte eines eine „1“ oder „2“ anzeigen, wird probiert, das jeweilige Muster in alle Richtungen anzuwenden, indem für jede Richtung mithilfe des *int*-Arrays überprüft wird, ob die relevanten *Feld*-Objekte das richtige anzeigen. Da *Feld*-Objekte auch am Rand liegen können und somit die relevanten *Feld*-Objekte möglicherweise außerhalb des Spielfeldes lägen, also nicht existieren, verläuft in dieser Methode der Zugriff auf das *int*-Array meistens über die Methode *get(int[][] map, int x, int y, int onException).* Sollten *x* und *y* nun außerhalb von *map* liegen, wird *onException* zurückgegeben.

Für das Anwenden des 12-Muster in jede Richtung liegt im Anhang unter Abbildung 2 ein Beispiel vor.

Falls eine Aktion ausgeführt wurde, wird *true* zurückgegeben. Sonst wird, wenn alle *Feld*-Objekte untersucht wurden, *false* zurückgegeben.

4.3.7 Algorithmus zur Wahrscheinlichkeitsbetrachtung

Die letzte Methode *getChances()*, die verwendet wird, berechnet die Wahrscheinlichkeit, dass es sich bei einem *Feld*-Objekt um eine Mine handelt. Dazu wird zuallererst ermittelt, wie viele *Feld*-Objekte verdeckt und Minen nicht markiert sind, was in den Variablen *felderVerdeckt* und *minenUnmarkiert* gespeichert wird. Anschließend wird der Quotient aus *minenMarkiert* durch *felderVerdeckt* in der Variable *standardProbMine* gespeichert. Dieser Wert gibt die Wahrscheinlichkeit an, dass es sich bei einem *Feld*-Objekt um eine Mine handelt. Sind beispielsweise 40 Minen auf einem komplett verdeckten 16 x 16 Spielfeld vorhanden liegt dieser Wert bei etwa 0,16.

Jedoch sind dabei die Nachbarn der einzelnen *Feld*-Objekte noch nicht mit einbezogen. Daher wird ein zweidimensionales Array aus *double*-Werten angelegt, sodass jedem *Feld*-Objekt ein eigener Wahrscheinlichkeitswert zugeordnet werden kann. Falls ein *Feld*-Objekt nun bereits angeklickt wurde oder mit einer Flagge markiert ist, wird dessen Wert auf „-1“ gesetzt, um zu zeigen, dass dieses *Feld*-Objekt nicht anklickbar ist und somit bei der Ermittlung der höchsten und geringsten Wahrscheinlichkeit nicht beachtet werden soll. Für alle anderen *Feld*-Objekte läuft die Betrachtung folgender Maßen ab:

Es wird die Wahrscheinlichkeit aus Sicht aller Nachbarn des zu betrachtenden *Feld*-Objektes bestimmt, mit welcher es sich bei diesem um eine Mine handelt. Ist ein Nachbar nicht aufgedeckt, liegen keine Informationen über diesen vor und die Wahrscheinlichkeit aus dessen Sicht entspricht *standardProbMine*. Andernfalls wird die Menge an nicht markierten an diesen Nachbarn angrenzenden Minen durch die Menge an nicht aufgedeckten an diesen Nachbarn angrenzenden *Feld*-Objekten dividiert und als Wert verwendet.

Dazu werden die Nachbarn des Nachbarn abgerufen und für jeden von diesen wird überprüft, ob er aufgedeckt oder mit Flagge markiert ist. Im ersten Fall wird *nMöglicheMinenNahe* und im anderen *nMarkiertNahe* erhöht. Beides sind Variablen des Typs *int*, die im Vorhinein erstellt wurden. Anschließend wird die Menge an nicht markierten Minen als Differenz aus der Menge an angrenzenden Minen, die über die Methode *getMinenNahe()* des Nachbarn bereitsteht, minus *nMarkiertNahe* gebildet. Diese Differenz wird durch *nMöglicheMinenNahe* dividiert. Das Ergebnis gibt die Wahrscheinlichkeit aus Sicht des Nachbarn an.

Grenzen an einen Nachbarn nun beispielweise drei verdeckte *Feld*-Objekte (unter denen auch das zu betrachtende *Feld*-Objekt) und eine Flagge an, während insgesamt zwei Minen angrenzen, würde aus Sicht dieses Nachbarn das zu betrachtende *Feld*-Objekt mit einer Wahrscheinlichkeit von 33% eine Mine sein, es wird also der Wert 0,3 verwendet.

Die Summe dieser Wahrscheinlichkeiten aller Nachbarn wird durch die Menge an Nachbarn dividiert und das Ergebnis als Wahrscheinlichkeit für das betrachtete *Feld*-Objekt im *int*-Array gespeichert.

Zurückgegeben wird am Ende die bereits angesprochene *HashMap*. Dazu muss jedoch noch das *Feld*-Objekt mit dem höchsten und geringsten Wahrscheinlichkeitswert bestimmt werden, was über zwei *for*-Schleifen passiert, die alle Elemente aus dem zweidimensionalen *double*-Array durchgehen.

Im Anhang ist unter Abbildung 3 ein Beispiel zu finden. Dieses zeigt, wie die Wahrscheinlichkeitswerte zu einem Spielfeld aussähen.

5 Künstliche Minesweeper-Intelligenz

Nachdem nun erklärt wurde, wie das Programm funktioniert, folgt nun eine Betrachtung seiner Funktionsfähigkeit sowie von Verbesserungsmöglichkeiten.

5.1 Bewertung

Meine Variante von Minesweeper ist nicht immer gewinnbar. Manchmal treten Fälle auf, in denen eine Mine mit gleicher Wahrscheinlichkeit auf verschiedenen Feldern liegen könnte. Somit wird die KI keine Erfolgsquote von 100% erreichen.

Während eines Tests, in dem die KI 4675 Runden abschloss, hatte diese eine Erfolgsquote von etwa 47% mit 2188 gewonnenen und 2487 verlorenen Spielen. Der schnellste Sieg wurde dabei nach 257ms errungen. Durchgeführt habe ich diesen Test mit der ausführlichen Version des Programms, aber unter gleichen Bedingungen wie in der kurzen Variante (abgesehen von einer minimierten Wartezeit zwischen einzelnen Aktionen). Denn in jener werden in der Konsole automatisch Statistiken ausgegeben. Auf dem beiliegenden Datenträger findet sich im Ordner „Anhang“ die Datei „KILog.txt“, welche ein Protokoll zu diesem Test darstellt.

Hinsichtlich Geschwindigkeit ist die KI also deutlich besser als jeder Mensch. Durch das Verringern der Wartezeit zwischen Aktionen auf null Sekunden können häufig Zeiten von unter 400ms erreicht werden. Der Weltrekord von Kamil Muranski liegt bei 7,03 Sekunden[11] und mit der Bestzeit aus dem eben genannten Test ist die KI etwa 27-mal schneller!

Auch die Erfolgsquote würde ich als gut beurteilen, denn die Fälle, in denen die KI verloren hat, hätte auch ich größtenteils nicht gewonnen. Zu Vergleichszwecken habe ich zwei weitere Tests durchgeführt. Im ersten habe ich 20 Runden gespielt und dabei eine Erfolgsquote von 5% erreicht. Im zweiten Test wurden 4152 Runden lang nur zufällig Felder angeklickt, wobei keine Runde gewonnen wurde. Dies zeigt, dass die Lösungsalgorithmen auf jeden Fall besser funktionieren als vollständiger Zufall. Auch schneidet die KI bei diesem Spiel deutlich besser ab als ein Anfänger, wie ich es einer bin.

Der Leistungsaufwand ist dabei für die meisten Computer problemlos zu tragen. Es werden zwischen 100 und 200 MB Ram verwendet und die CPU Auslastung des bei mir verwendeten Intel Core i7-3770 mit 3,40 GHz schwankte während den Runden unter 1% und kurz nach dem Erstellen des Spielfeldes unter 10%.

5.2 Verbesserungsmöglichkeiten

Im Bereich KI gibt es viele verwendbare Algorithmen. Besonders gut einbaubar wäre ein Bruteforce-Algorithmus. Ebenfalls könnte Multithreading genutzt werden.

[11] Vgl. The Authoritative Minesweeper: “Minesweeper Level Rankings” (Tabelle), Stichwort “Intermediate”, online abrufbar unter: http://www.minesweeper.info/scorelists.html, Stand: 28.02.2017.

„Die **Brute Force** -Methode bzw. **Methode der rohen Gewalt** ist der Fachbegriff für eine Lösungsmethode schwerer Probleme aus dem Bereich der Informatik und der Spieltheorie, die auf dem Ausprobieren aller (oder zumindest eines erheblichen Teils der in Frage kommenden) Varianten beruht.“[12] Im Fall von Minesweeper könnte ein solcher Algorithmus also alle möglichen Anordnungen für Minen betrachten und somit Felder finden, die immer, selten oder gar nicht Minen sein können und anklicken beziehungsweise markieren.

Multithreading ist meistens schneller als alle aktionsfindenden KI-Algorithmen hintereinander auszuführen, da ein Computer mehrere Aktionen gleichzeitig durchführen kann. In diesem Programm müssten dafür die Methoden *doLogikFeldnachbarn(),* *doLogikMinenmenge(),* *doPatternlogik()* und *getChances()* in einzelnen *run()* Methoden von *Thread*-Objekten aufgerufen werden. In der *doAction()*-Methode müssten dann wiederum die *start()*-Methoden dieser Objekte aufgerufen werden.

Neben diesen Optimierungen könnte außerdem die Methode *getChances()* *doLogikFeldnachbarn()* vollständig ersetzen, da beide Methoden die Nachbarn eines Feldes betrachten. Den *Feld*-Objekten, die von letzterer angeklickt werden, würden von ersterer immer eine Wahrscheinlichkeit von null und denen, die markiert werden, eine von eins zugeordnet werden. Nur kann erstere Methode auch Aktionen finden, wenn letztere keine findet und ist somit besser geeignet. Erstere Methode könnte aber auch verbessert werden, indem diese *Feld*-Objekte mit einer Wahrscheinlichkeit von null oder eins sofort anklickt beziehungsweise markiert und nicht erst alle anderen *Feld*-Objekte betrachtet, um am Ende eine *HashMap* zurückzugeben.

Der Algorithmus zur Mustersuche klickt beim 11-Muster bisher immer nur ein *Feld*-Objekt an, könnte aber in manchen Fällen auch mehrere Aktionen in einem Durchlauf durchführen, wie es die Methode *doLogikFeldnachbarn()* tut. Dies wäre wünschenswert, wenn zwar an die eine „1“ dieselben verdeckten *Feld*-Objekte, wie an die zweite „1“ angrenzen aber an erstere noch weitere verdeckte *Feld*-Objekte angrenzen, die nicht an letztere angrenzen. Diese könnten alle angeklickt werden, es würde aber nur eines ausgewählt. Ein solcher Fall ist in Abbildung 7 abgebildet. Die mit schwarzen Pfeilen markierten *Feld*-Objekte könnten alle angeklickt werden, aber bisher würde dies nur mit einem passieren.

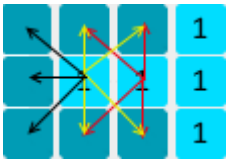

Abbildung 7

Auch würde die KI deutlich schneller das Spielfeld lösen, indem die Wartezeit in der *autoAct()*-Methode auf null gesetzt würde. Jedoch würde dies die CPU-Auslastung

[12] schach-computer.info Wiki: „Brute Force“, online abrufbar unter: http://www.schach-computer.info/wiki/index.php?title=Brute_Force, Stand: 28.02.2017.

deutlich erhöhen und besonders könnte der Spieler viel schwerer den getätigten Aktionen folgen sowie nach einer bestimmten Anzahl stoppen.

Zuletzt zeigen Statistiken von minesweeper.info, dass ein Unterschied zwischen dem ersten Klick in einer Ecke und der Mitte besteht. Während der erste Klick in einer Ecke häufiger mehrere *Feld*-Objekte aufgedeckt, werden in der Mitte seltener mehrere, aber dann eine größere Anzahl an *Feld*-Objekten aufgedeckt. Ersteres führt letzten Endes zu einer höheren Erfolgsrate, da die Ausgangsposition in mehr Fällen gut ist, während letzteres hohe Geschwindigkeitsrekorde begünstigt, da eine große Menge aufgedeckter *Feld*-Objekte am Anfang zu weniger nötigen Aktionen zum Lösen des Spielfeldes führt.[13]

6 Reflexion

Zurückdenkend gab es verschiedene Schwierigkeiten. Ich begann damit, das Spiel an sich zu programmieren und dieses hin und wieder zu spielen. Dadurch fielen mir nach und nach anwendbare Methoden auf, um dieses möglichst schnell und sicher zu gewinnen. Diese Methoden jedoch als klare Aufgabenabfolgen zu formulieren und besonders auch in Java umzusetzen, wies gewisse Schwierigkeiten auf. Besonders schwierig war aber das Erklären des Quelltextes in geringem, aber dennoch verständlichen Umfang.

Während das Finden und Umsetzen der Algorithmen hinter den Methoden *doLogikFeldnachbarn()* und *doLogikMinenmenge()* noch recht einfach war, wurde es bei *doPatternlogik()* und *getChances()* etwas schwieriger. Einzig die Grundidee hinter *doPatternlogik()* stammt von minesweeper.info und somit nicht von mir. In deren Minesweeper-Wiki werden verschiedene Strategien erklärt, wie auch das 12- und 11-Muster, welches dieser Methode zu Grunde liegt[14].

Doch gerade die Frage, wie Muster in dem *spielfeld*-Array erkannt werden sollen, war besonders knifflig. Anfangs überlegte ich, Reihen und Zeilen nach bestimmten Zahlenketten, wie 1221 abzusuchen. Jedoch erwies es sich als sinnvoller, direkt nach dem 12- und 11-Muster zu suchen. Nun kann dieses aber sowohl horizontal als auch vertikal und in verschiedenen Variationen auftreten. Letzten Endes entschied ich mich dafür, einfach für jedes *Feld*-Objekt zu überprüfen, ob die relevanten *Feld*-Objekte für das Muster in der Umgebung auftauchen. Dabei muss jedes Muster in jede Himmelsrichtung angewendet werden.

[13] Vgl. MinesweeperWiki: „Strategy“, online abrufbar unter: http://www.minesweeper.info/wiki/Strategy, Stand: 28.02.2017.
[14] Ebd.

Als besonders problematisch zeigte sich die Frage nach einem Bruteforce-Algorithmus. Meine Überlegung war, das Spielfeld nach kleineren vom Rest des Feldes durch aufgedeckte *Feld*-Objekte abgegrenzte Flächen zu durchsuchen und in diesen alle möglichen Positionen für Minen heraus zu finden. Denn so könnten *Feld*-Objekte gefunden werden, auf denen keine Minen liegen können. Doch dies umzusetzen wurde schnell viel zu umfangreich und auch traten immer mehr Probleme auf, wie das Überprüfen, ob die Minenkonfiguration möglich ist. Vermutlich wäre es rückblickend sinnvoll gewesen, erst einen Programmablaufplan zu entwickeln. Dennoch stellt es kein Problem dar, dass ich diesen Algorithmus nicht fertiggestellt habe, denn durch das 11- und 12-Muster welches von *doPatternlogik()* gesucht wird, können diese abgegrenzten Flächen oft ebenfalls gelöst werden.

Hinsichtlich Programmierfähigkeiten hat mich die Facharbeit definitiv weitergebracht. Die insgesamt über 6000 Zeilen Programmcode und 14 Klassen für die ausführliche Version des Programes stellten eine gute Übung zur Objektinteraktion, aber auch zur Lösung allgemeiner Probleme in Java dar. Im Erstellen graphischer Oberflächen wurde ich ebenfalls deutlich besser.

7 Literaturverzeichnis

- Gamesbasis: ”Minesweeper”, online abrufbar unter: http://www.gamesbasis.com/minesweeper.html, Stand: 28.02.2017.
- Google: “Quick, Draw!” (Computerspiel), online abrufbar unter: https://quickdraw.withgoogle.com/, Stand: 28.02.2017.
- Google: “TensorFlow” (Software), online abrufbar unter: https://www.tensorflow.org/, Stand: 28.02.2017.
- Michael A. Nielsen, “Neural Networks and Deep Learning”, Determination Press, 2015, Kapitel 1, online abrufbar unter: http://neuralnetworksanddeeplearning.com/chap1.html, Stand: 28.02.2017.
- Microsoft Studios: „Microsoft Minesweeper“ (Computerspiel), online abrufbar unter: https://www.microsoft.com/de-de/store/p/microsoft-minesweeper/9wzdncrfhwcn, Stand: 28.02.2017.
- MinesweeperWiki: „Strategy“, online abrufbar unter: http://www.minesweeper.info/wiki/Strategy, Stand: 28.02.2017.
- Oracle: “Java Documentation”, online abrufbar unter: https://docs.oracle.com/javase/7/docs/api/, Stand: 28.02.2017.
- Oracle: “Java Language Keywords“, online abrufbar unter: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html, Stand: 28.02.2017.
- schach-computer.info Wiki: „Brute Force“, online abrufbar unter: http://www.schach-computer.info/wiki/index.php?title=Brute_Force, Stand: 28.02.2017.
- SciShow: “The AI Gaming Revolution” (Video), online abrufbar unter: https://www.youtube.com/watch?v=Xhec39dVGDE, Stand: 28.02.2017.
- Springer Gabler Verlag: „Gabler Wirtschaftslexikon“, Stichwort „Künstliche Intelligenz (KI)“, online abrufbar unter: http://wirtschaftslexikon.gabler.de/Archiv/74650/kuenstliche-intelligenz-ki-v12.html, Stand: 28.02.2017.
- The Authoritative Minesweeper: “Minesweeper Level Rankings” (Tabelle), Stichwort “Intermediate”, online abrufbar unter: http://www.minesweeper.info/scorelists.html, Stand: 28.02.2017.
- The Eclipse Foundation: „Elipse Neon“ (Software), online abrufbar unter: https://eclipse.org/downloads/, Stand: 01.03.2017.

8 Anhang

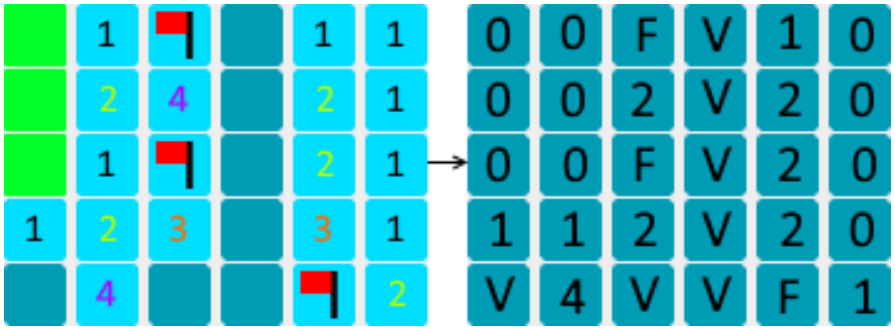

Abbildung 1

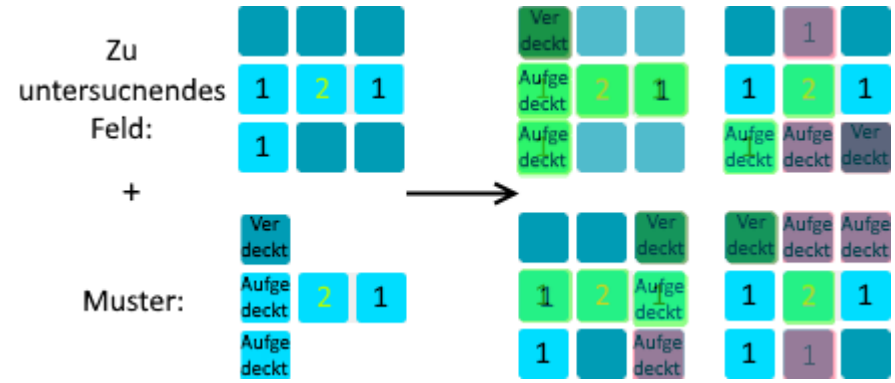

Abbildung 2

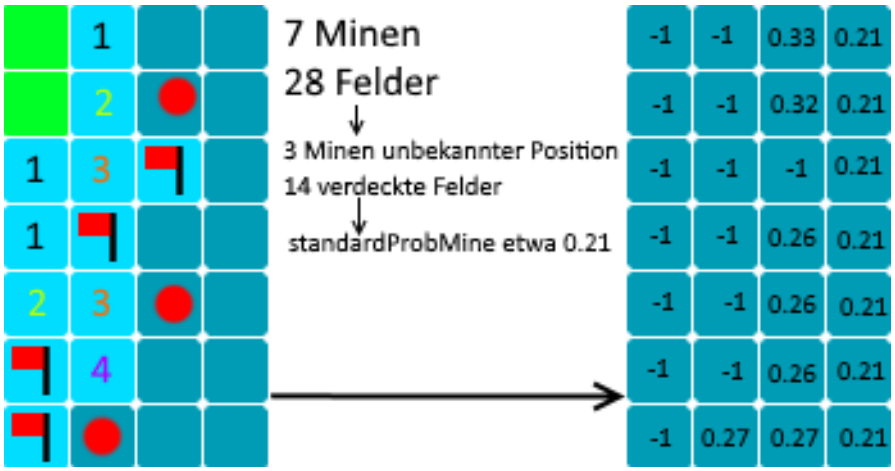

Abbildung 3

9 Erklärung der Eigenständigkeit

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die im Literaturverzeichnis angegebenen Hilfsmittel verwendet habe.

Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Ort / Datum	Unterschrift