Besondere Lernleistung – Sem Klauke

2048 Spiel + AI zum lösen dieses

Das Spiel 2048 ist ein Singel-Player Puzzle Spiel. Es wurde 2014 als Open Source Software in JavaScript veröffentlicht.

Es wird auf einem 4x4 Feld gespielt, auf dem sich Kacheln mit 2er Potenzen (2^2 bis 2^{17}) darauf als Zahlenwert befinden. Bei einem Spielzug werden alle Kacheln auf dem Feld in eine Richtung bewegt (oben, unten, links, rechts), die mit den Pfeiltasten bestimmt wird. Treffen dabei 2 Kacheln mit derselben Zahl aufeinander, verschmelzen diese zu einer Kachel mit der nächst höheren 2er Potenz als Zahlenwert. Nach jedem Spielzug kommt eine Kachel mit dem Wert 2 oder 4 hinzu (an eine Zufällige Position).
Das Ziel ist es, so viele Kacheln zu verschmelzen, dass am Ende eine Kachel mit dem Wert 2048 herauskommt. Das Spiel kann dann natürlich auch weitergespielt werden.
Verlieren tut man, wenn alle 16 Felder besetzt sind und somit keine neue Kachel hinzukommen kann.

Um das Spiel zu spielen, öffnen Sie einfach index.html in einem beliebigen Browser (Ausgenommen Internet Explorer – Hier keine Garantie). Probieren Sie es jetzt aus und benutzen Sie auch den „Start AI“ Knopf.

Meine Aufgaben/Anforderungen für dieses Projekt waren:

- Ein Spielbares 2048 Spiel mit GUI programmieren
- Eine Künstliche Intelligenz (AI) programmieren, die das Spiel 2048 spielt
- Dabei eine Erfolgsquote von 100%, also jedes Spiel wird die 2048 Kachel erreicht
- Beweisen, dass der Algorithmus funktionsfähig ist. Z.B. durch Tests

Ordner/*Datei* Zuordnung:

- **documentation** -> Dokumentation des Projektes
- **documents** -> assets/notizen/andere datein – IGNORIEREN
- **js** -> JavaScript Dateien das 2048 Spiels und der AI
- **node_modules** -> node.js Frameworks und Programme (für Tests benötigt, nicht für das Spiel selber) – IGNORIEREN
- **tests** -> Dateien zum starten und konfigurieren von Tests für die AI
- *index.html/index.css* -> GUI für das Spiel 2048. Ausgangspunkt für das Spiel
- *package*.json -> node.js Informationen – IGNORIEREN

Im Folgenden werde ich mein Vorgehen erläutern und auf Schwierigkeiten so wie Ergebnisse bei der Programmierung eingehen.

Am Ende der Dokumentation ist ein „echte“ Dokumentation angehängt, also eine Klassen Dokumentation für den Code.

Modellierung

Zunächst habe ich mir Gedanken über den Grundsätzlichen Aufbau und die Programmiersprache gemacht.

Entschieden habe ich mich für eine Entwicklung mit JavaScript, CSS und HTML, da ich so sehr einfach und schnell eine simple GUI mit HTML und CSS bauen kann. Außerdem ist das Programm dann auch ohne Mehraufwand Cross-plattform.

Für die Modellierung habe ich mir von vornerein ein MVC ähnliches Konzept überlegt:
View
- Es gibt eine index.html, die die GUI abbildet, und in der alle Anderen Datei zusammenlaufen, auch Modell und Controller (mit <script> und <link />)
- Es gibt eine index.css, in der die Gesamte Gestaltung der GUI und die Animationen zu finden sind

Modell
- In der board.js befindet sich die Board-Klasse, die das Spielbrett im Speicher simuliert und mit Funktionen ausstattet. In dieser Klasse befindet sich auch Controller Elemente. Das simple Spiel ist hier komplett implementiert. Also Spielzüge, gewinnen und verlieren ist möglich

Controller
- ai.js beinhaltet die Klassen für den AI Algorithmus, der mithilfe von der Board-Klasse ausgeführt wird. Ausführen des Algorithmus gibt nächst Besen Zug zurück (oben/unten/links/rechts)
- main.js verbindet alles zusammen. Es leiten die Tastatureingaben des Users weiter an das Model, welches das Spielbrett verändert und stellt das Spielbrett aus dem Modell wieder. Auch Spielzüge der AI werden vom Modell zur GUI weitergeleitet.

Als nächster Schritt in der Modellierung habe ich mir Gedanke über den Algorithmus gemacht, den ich für die AI verwenden will. Entschieden habe ich mich für einen minimax Algorithmus, welchen ich später noch genauer erläutere. Später stieß ich noch auf den Expectiminimax Algorithmus, welcher auch geeignet wäre. Aus Zeit Gründen hab ich dieses allerdings nicht ausprobiert.

GUI / View

Das Grundgerüst der GUI bildet HTML. Neben dem Button um die AI zu starten gibt es noch ein <table> Element in der index.html, welches eine **leere** 4x4 Matrix darstellt. Dieses ist das Spielbrett. Das Design des Spielbrettes ähnelt hierbei dem Original. Hinzukommt ein Container für die Kacheln auf dem Spielbrett (<div id="pieces"), welcher mit CSS Positionierung über die Tabelle, also das Spielbrett gelegt wird.

Die Kacheln werden durch einzelne Elemente repräsentier. Diese liegen in dem Container und nicht in der Tabelle.

Mit Hilfe von Klassen, die eine CSS Positionierung enthalten, werden die Kacheln auf dem Spielbrett auf das richtige Feld „gelegt". Zum Beispiel hat eine Kachel die Oben-Rechts sein soll die CSS Klasse x4-y4. Unten Links wäre x1-y1 usw.

Kacheln mit verschieden Werten habe auch unterschiedliche Farben. Auch die Farben sind an das Original angelehnt.

Um eine Kachel zu verschieben, wird einfach ihre CSS Klasse geändert. Mit Hilfe von CSS Animationen wird das verschieben als eine Bewegung dargestellt.

Board-Klasse / Modell

Durch den Konstruktor Board() wird die Klasse erstellt, die das Spielbrett repräsentiert und alle Funktionen enthält um das Spielbrett zu manipulieren.
Die Kacheln werden hierbei in einem 2 Dimensionalen Array pieces gespeichert. An leeren Feldern steht null in dem Array. Die Positionen werden also durch Koordinaten repräsentiert. Das übergeordnete Array sind im Prinzip die x Koordinaten, die untergeordneten Arrays die y Koordinaten.
y = 0 unterste Reihe / y = 3 oberste Reihen
x = 0 ganz linke Spalte / x = 3 ganz rechet Spalte

Um die Koordinaten einfacher zu speichern verwende ich die Konvention (Richtige Klassen gibt es nicht) der „cord-objects“. Dies sind einfache JavaScript Objekte mit einem x Attribut und einem y Attribut. Über Punksyntax kann so schnell auf x oder y zugegriffen werden.

Eine Weitere Objekt Konvention ist das „piece-object“. Dieses repräsentiert eine Kachel als Objekt und enthält die Attribute:
value (Zahlenwert auf der Kachel)
merged (Speichert ob die Kachel im letzte Spielzug aus 2 Kacheln zusammengeschmolzen wurde)

Es gibt Funktionen die Kacheln an bestimmten Koordinaten hinzuzufügen, zu löschen, zu verschieben, zurückzugeben usw. Fast alle werden für die wichtigste Funktion moveBoard() benutzt. Dieser übergibt man eine Richtung (in Form eines Vektors der Länge 1) und in diese Richtung wird dann das gesamte Spielbrett verschoben. moveBoard() führt also einen Spielzug durch.
Hierbei kann auch eine Funktion vom GUI mit übergeben werden, welche die Bewegung der Kacheln in der GUI anstößt. Somit ist die Verbindung zum View gegeben, aber optional. Also kann die Klasse Board einfach auch einfach für die AI genutzt werden, um Spiele zu simulieren usw.

Bei der Funktion addRandomPiece() wird eine neue, zufällige Kachel in das Spielbrett eingefügt. Dies passiert ja nach jedem Spielzug. Allerdings wird die Funktion **nicht** vom Modell ausgeführt, dies ist Aufgabe der Controller.
Dabei hat die Kachel zu 90% den Wert 2 und zu 10% den Wert 4. Dies ist übernommen vom Original Spiel.

Wichtig ist noch das Konzept des Levels. Jede Kachel hat ein Level, welches durch den Zahlenwert dargestellt wird. Und zwar ist das Level x wenn:
gilt 2^x = Kachel Wert
Beispiel: (Wert 4: Level 2 / Wert 16: Level 4 / Wert 2048: Level 11)
Dies kann berechnet werden durch:
ln(Kachel Wert) / ln(2)

Verknüpfung / Controller

Die Verknüpfung findet in der main.js Dateien statt.

Tastatur Befehle vom User (Pfeiltasten) werden über das Browser Funktion document.onkeydown abgefangen und dann mit der entsprechenden Richtung des Spielzuges an das Modell weitergeleitet, welches auch von main.js erstellt wurde.

Wird die AI gestartet, wird die AI für jeden Spielzug erneut nach dem nächsten, besten Spielzug gefragt, und diese dann an das selbe Modell weitergegeben.
Das Modell weiß somit nicht woher der Befehl zum Spielzug kommt. AI oder User, beides möglich und gleichwertig.

Die Funktion uiMovePiece wird dem Modell zusätzlich immer übergeben, somit wird immer auch das GUI mit verändert und auch die GUI ist unabhängig von der Quelle des Befehls.

Nach jedem Spielzug fügt der Controller eine neue, zufällige Kachel ein (addRandomPiece())

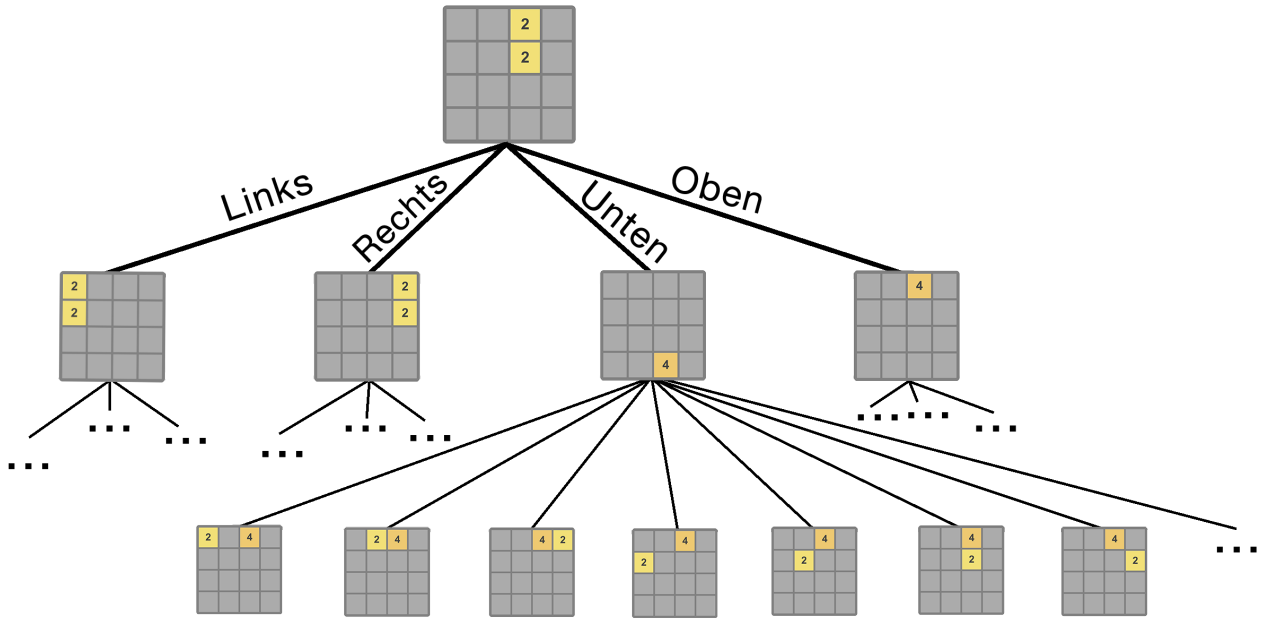
Ein bisschen logging zum Debugging ist hier auch implementiert.
D

AI

AI steht für „Artificial Intelligence“, also Künstliche Intelligenz.

Um den Richtigen Algorithmus zu wählen habe ich mir verschieden Algorithmen zum Lösen von Spielen angeschaut. Natürlich bin ich nicht der Erste der sich an eine AI für 2048 heran wagt. Ich habe also auch Blogposts über schon existierende AI’s gelesen. Dennoch habe ich für mich abgewogen, welchen Algorithmus ich verwenden möchte. Kriterien waren dabei: Zeitkomplexität, Implementierungs Komplexität und Sinnhaftigkeit in Bezug auf das Spiel 2048.

Viele Algorithmen arbeiten nach demselben Prinzip: Sie erstellen eine Baumstruktur, welche den gesamten möglichen Spielverlauf, von der Aktuellen Situation ausgehend, abbildet.
Dabei wird jeder Spielstand zu einem Ast, von dem wieder neue Äste ausgehen. Würde man diese Baumstruktur bis zum Ende des Spiels aufstellen, würden bei den meisten Spielen Bäume mit Billionen Ästen herauskommen. Darum wählt man eine Tiefe (eng depth) für den Algorithmus. Hiermit wird bestimmt wie viele Eben von der Baumstruktur generiert wird. Ein Graphisches Beispiel für eine Baumstruktur für das Spiel 2048:



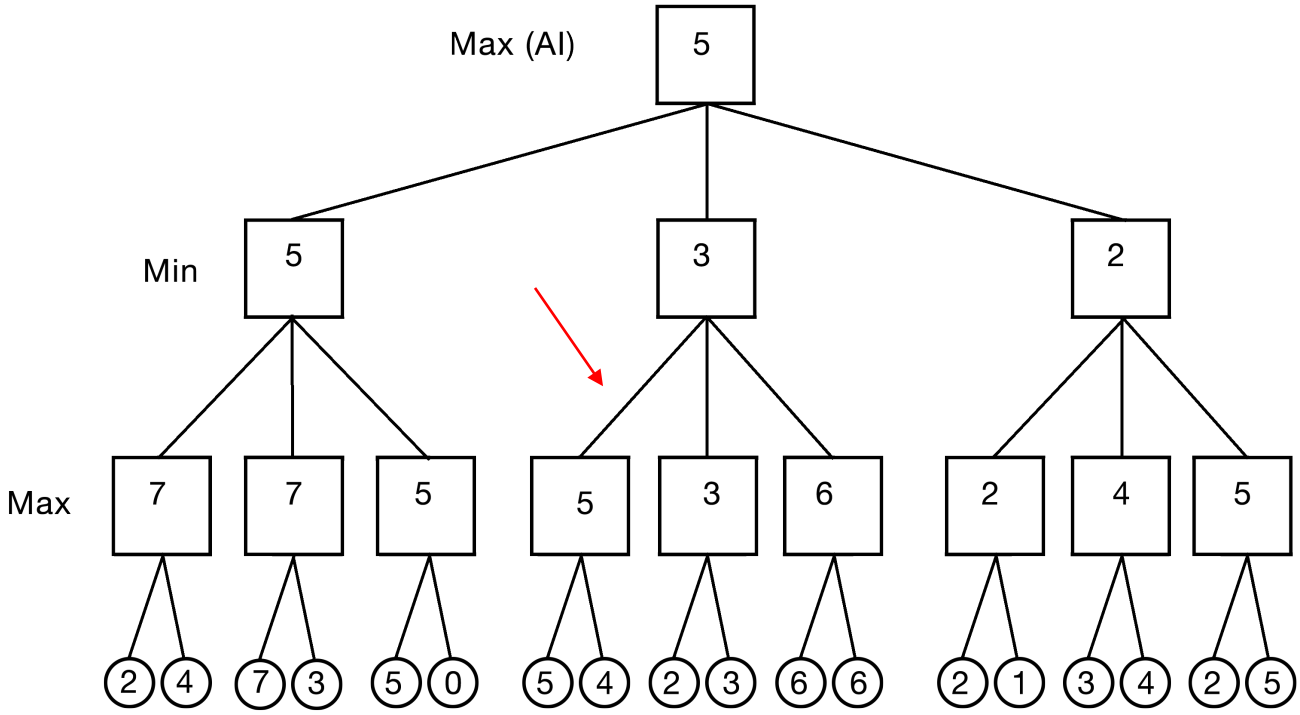
Zu sehen ist Beispiel. Der Ausgangspielstand ist ganz Oben. Die nächste „Ast-Ebene“ ist der Spielzug des Spielers. Hierbei gibt es 4 Äste: Rechts/Links/Unten/Oben.
*Nach dem Spielzug wird eine neue Kachel eingefügt, dies ist die nächste „Ast-Ebene“. Hier sind es (16 - Anzahl Kacheln) * 2 Äste, da auf jedes Freie Feld eine 2 oder 4er Kachel kommen kann. So entsteht schnell ein Großer Baum mit allen möglichen Spielständen. Hier mit der Tiefe 1 (gekürzt).*

In der Art, wie mit der Baumstruktur umgegangen wird, unterscheiden sich die Algorithmen.
Ich habe hier den minimax Algorithmus mit Alpha/Beta Suche gewählt.
Hierbei geht man davon aus das es Zwei Spieler gibt. Der eine (in dem Fall die AI) will bei seinem Zug, den für ihn besten Spielzug ausführen (=maximising player), der andere will bei seinem Zug den für den ersten Spieler schlechtesten Spielzug machen (=minimising player).

Die AI stellt bei meinem Projekt den Maximsing player da dieser immer versucht den Besten Spielzug auszuführen. Der Gegner der AI ist in diesem Fall die zufällig hinzugefügte Kachel nach jedem Spielzug. Diese kann ein Spiel ganz schön versauen. Dabei gehe ich immer vom „worst case“ aus, also das die für die Ai schlechteste Position der neuen Kachel gewählt wird. Somit haben wir den Minimising player.

Aber wie bewertet man ob ein Spielzug gut oder schlecht ist?
Mit Hilfe von Heuristik Funktionen, die ich später ausführlich erläutern werde.
Diese Funktionen generieren einen Heuristik Wert für jedes Spielbrett Situation. Sie Bewerten also wie gut das aktuelle Spielbrett ist. Umso höher der Heuristik Wert, desto besser ist die Aktuelle Situation.

Diese Heuristik Werte werden von den Spielständen am Ende der generierten Baumstruktur generiert. Eine Grafische Darstellung eines strak vereinfachten minimax Algorithmus sieht so aus:



*Dieses Diagramm ist von unten zu lesen. Die Werte und Struktur ist **nicht** auf 2048 bezogen, sondern soll nur den minimax Algorithmus erläutern. Der Maximising player versucht immer den größten Wert zu nehmen, der minimising, den kleinsten.*

Dieses Algorithmus führt zum Ziel, ist aber nicht besonders schnell. Mit Hilfe der Alpha Beta Suche wird die Laufzeit um ein Vielfaches reduziert. Hierbei gibt es die variablen, oder viel mehr Grenzen Alpha und Beta.
Beta ist dabei die **minimale, obere Grenze** und Alpha dabei die **maximale, untere Grenze** für einen Ast. Dies hat zufolge, dass wenn ein neuer Ast untersucht werden soll, muss für dessen Wert N gelten:

$\text{Alpha} \leq N \leq \text{Beta}$

Trifft dies nicht zu, ist der Wert des Astes irrelevant und muss deshalb nicht berechnet werden.

Ein Beispiel an dem Diagramm auf Seite 7:
Die Baumstruktur wird Ast für Ast durchlaufen. Gestartet wird hier im Diagramm links. Gehen wir davon aus, dass der erste der drei Hauptäste schon berechnet ist (also den Wert 5 hat).
Jetzt gehen wir zum nächsten Ast. Immer den Nächst Linken Ast hinunter bis zum Ende zu den Werten. Dort muss sich dann der Maximierende Spieler für den größeren Wert entscheide. Er wählt 5 aus 5 und 4 (siehe Roter Pfeil). Da die Ebene darüber ein Minimalisierender Spieler ist, wird er also entweder die 5 wählen, oder einen ***kleineren*** *Wer als 5. Da aber der ganz linke Ast auf derselben Ebene auch den Wert 5 hat, lohnt es sich nicht, diesen Ast weiter zu berechnen (In dem Diagramm ist er trotzdem berechnet, da diese auch für das Verständnis des minimax Algorithmus alleine dienen soll. Obwohl minimax und Alpha Beta Such fast immer zusammen verwendet werden).*

Diese Grenzen werden eben in Alpha und Beta gespeichert.

Heuristiken

Wie bereits erwähnt, bewertet die Heuristik, wie gut ein Spielbrett ist. Dabei gibt es viele Ansätze für fast jedes Spiel. Fast immer besteht der Heuristik Wert aus mehrerer Kriterien. Diese Kriterien und Bewertungen sind das, was einen Algorithmus ausmachen. Dabei sind 2 Ding immer wichtig:
Was bewertet man und wie bewertet man es.

Meine Überlegungen und Kriterien waren:

Monotonie (Monotonicity)
Beim Spielen merkt man, dass es wichtig ist sein Spielbrett aufgeräumt zu haben, um möglichst viele Felder frei zu halten (um das Risiko zu verlieren zu mindern) und damit Kacheln schnell verschmolzen werden können.
Dazu müssen die Kacheln sowohl vertikal als auch horizontal aufsteigend oder absteigend sortiert sein.
Dabei entsteht schnell das Muster, dass sich in einer Ecke die größte Kachel ansammelt. Dies ist auch gewünscht, da sie dort nicht stört, währen man versucht eine genau so große Kachel zu erspielen.

Einwurf: Mir ist während der Entwicklung aufgefallen, dass ich auch durch einen anderen Faktor als Auf und Absteigend sortieren, Monotonie erreiche. Und zwar, wenn die AI die Baumstruktur durchläuft, fängt sie bei dem Ast „Rechts“ an dann, „Oben“. Darauf Folgen „Unten“ und „Links“. Das Heißt, sind zum Beispiel Werte bei Links und Rechts gleich, wird immer Rechts genommen (-> Alpha Beta Suche). Oder sind Werte von Oben und Unten gleich, wird Oben genommen. Somit wird Oben und rechts präferiert. Was zur Folge hat, dass sich das Größte oder die Größten Teile in der rechten, oberen Ecke sammeln.

Im Code wird der Heuristik Wert so berechnet, dass jede Vertikale Zeile, und jede Horizontale Spalte einzeln Bewertet wird, wie aufsteigen oder absteigend sie ist. Alle 8 Werte werden dann addiert.

Gleichmäßigkeit (smoothness)
Verlieren tut man dann, wenn kein Feld mehr frei ist. Also muss man Kacheln verschmelzen (hilf nebenbei auch beim Gewinnen). Um Kacheln miteinander zu verschmelzen, müssen 2 Kacheln mit demselben Wert in einer Zeile/Spalte und keine adere Kachel dazwischen sein.

Dies Bewertet die Gleichmäßigkeit. Umso mehr Gleiche Kacheln nebeneinander sind, umso höher ist der Gleichmäßigkeit's Wert. Sind sie nicht gleich, wird der Wert umso kleiner, umso größer der Abstand der beiden Kacheln-Werte hat.

Dies wiederspricht sich **nicht** mit der Monotonie Heuristik. Sind zwei Kacheln gleich groß, unterbricht dies nicht die aufsteigende oder absteigende Reihenfolge, sondern wird neutral bewertet.

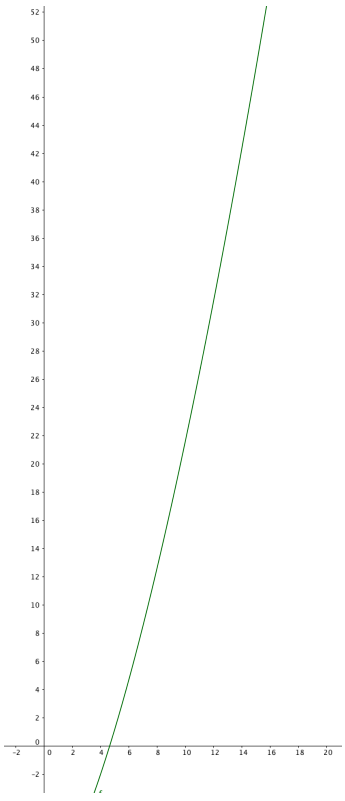
Leere Felder (emptyPieces)
Wie schon erwähnt ist es ratsam, viele Felder Frei zu halten, um nicht zu verlieren. Außerdem zwingt diese Heuristik die AI auch, Kacheln zu verschmelzen, da dies der einzige Weg ist, den Wert für diese Heuristik zu erhöhen.
Umso mehr Felder frei sind, umso höher ist auch der Heuristik Wert.
Allerdings wird er hier noch durch eine Mathematische Funktion etwas abgeflacht. Wenn x die Anzahl der freien Felder ist, dann gilt:
Heuristik Wert = $x^{0.8}$
Damit steigt der Wert im kleinen Bereich schnell an und steigt im größeren Bereich langsamer. Hintergrund ist der, dass wenn nur ca. 1-3 Felder frei sind, schnell wieder mehr Felder Frei werden sollen, aber es nicht so relevant ist, ob 9 oder 10 Felder frei sind.

Höchster Wert (highestValue)
Hierbei wird das Level der Größten Kacheln als Heuristik Wert genommen.
Liegt dieses Zusätzlich in einer der Ecken, wird der Wert noch erhöht (im Moment wird mal 1.6 gerechnet. Wert ist aus Gefühl und Tests gewählt).

Somit wird die Monotonie unterstützt und, viel wichtiger, die AI versucht immer die nächste Größere Kachel zu erreichen. Wenn es z.B. schon eine 1024 Kacheln gibt, wird die AI immer den Weg präferiere, der zur 2048 Kachel führt.

Verschmelzen (merges)
Bei dieser Heuristik, wird der Wert größer, wenn in einem Spielzug viele und Große Kacheln verschmolzen werden. Für jede Kachel die Verschmolzen wird in einem Spielzug wird der Wert genommen (x ist das Level der Kachel):

Heuristik Wert = $x^{1.5} - 10$
(siehe Diagramm Rechst)



Hinterher werden alle Werte Addiert. So werden viele Verschmelzungen und große Verschmelzungen belohnt.

Zuerst hatte ich diese Heuristik auf Alle Level von Kacheln angewendet. Aber das dies im Prinzip schon durch die „Leere Felder" Heuristik unterstützt wird und es sich in Test sehr negativ verhalten hat, greift die Heuristik jetzt erst ab einem Level von 6 (64er Kachel).
Dies hilf insoweit, dass die „Freie Felder" Heuristik nur generell Verschmelzungen belohnt. Somit Wird die gesamte Heuristik größer,

wenn besser „aufgeräumt“ wird, also große Kacheln verschmolzen werden.

Einwurf: Um die Funktionen für diese und die „Freie Felder“ Heuristik zu entwickeln, habe ich „GeoGebra“ (sollte bekannt sein) benutzt, um mir die Funktionen zu designen.

Abstand (distance)

Die Abstand Heuristik hat sich im Nachhinein durch zahlreiche Test als unbrauchbar und uneffektiv herausgestellt. Dennoch möchte ich den Gedankengag kurz erläutern.

Ich wollte eine Heuristik schaffen, die es bestraft (mit kleinen oder negativ Werten), wenn Kacheln mit hohen Werten bewegt werden. Ich dachte dies würde Helfen eine Monotonie Struktur aufrecht zu erhalten. Außerdem hatte ich oft beobachtet, dass meine AI die Größte Kachel aus einer der Ecken bewegte und dann dort eine 2er Kachel auftauchte. Was ziemlich schlecht für die Monotonie ist.

Allerdings haben Test mit und ohne diese Heuristik gezeigt, dass diese total ineffizient ist.

Gruppne (groups)

Die Gruppen Heuristik ist ebenfalls nicht in Gebrauch, da auch sie sehr negative Testergebnisse hervorgebracht hat.

Diese Heuristik sollte Grüppchen Bildung, also z.B. ein paar 2/4/8er Kacheln in mitten von 128/265er Kacheln. Stellte sich als unbrauchbar herraus.
Dies waren alle Heurstiken

Um die gesamte Bewertung eines Spielbrettes zu bekommen, wird nun jede einzelne Heuristik berechnet. Danach werden diese mit ihren Gewichtungen multipliziert. Diese Gewichtungen beschreiben, wie wichtig eine Heuristik ist und dem entsprechend auch, wie hoch ihr Einfluss bei dem Gesamtwert ist.
Die einzelnen Werte für die Gewichtungen werden über eine kleine Config übergeben. Es gibt eine default Config (die Beste, die ich durch Test finden konnte), die aber überschreiben werden kann, um automatisiert Konfigurationen zu testen. Mehr dazu im Kapitel AI-Test.

Besonderheiten bei meinem minimax Algorithmus:

- Bei dem Minimalisierenden Spieler, werden nicht alle Pfade berechnet. Zuerst werden alle Heuristik Werte von den neuen Spielbrettern berechnet und nur der oder die Pfade berechnet, bei denen die Heuristik am kleinesten ist. Somit geht man immer vom Worst Case aus, was allerdings auch das Risiko des Verlierens senkt.
- Ich benutze eine Baum Tiefe von 4. Wählt man die Tiefe zu klein, wird nicht genug vorrausschauend gespeilt. Wählt man sie zu groß, so ist der Zeitaufwand sehr groß und evtl. fällt einem die Zufälligkeit der Neuen Kachel in den Rücken, da dann eine große Anzahl an zufälligen Kacheln mit berechnet wird. 4 erschien mir ein guter Kompromiss

AI-Tests

2048 hat mit der Zufällig hinzukommenden jeden Spielzug und mit der zufälligen Ausgangsituation 2 große, Zufällige Elemente, was jedes Spiel anders macht. Also kann man nicht nach einem einzigen Erfolgreichen Test der AI, wo sie 2048 erreicht, ausvon ausgehen, dass die AI immer funktioniert. Dies kann man eigentlich nie so richtig, da immer der Zufall entscheidet.
Daher muss man dies entweder Mathematisch beweisen, z.B. durch Elemente aus der Grafentheorie oder durch Zahlreiche Tests. Da ich Mathematisch nicht so bewandert bin habe ich mich für die empirischen Tests entschieden.

Bei den Tests wollte ich nicht nur Beweisen, dass meine Lösung funktioniert, sondern auch meine AI verbessern. Im AI Kapitel erwähnte ich die Konfigurationen für die Heuristik Werte. Mithilfe der Tests habe ich versucht, die Beste Konfiguration zu finden.

Dafür habe ich mir zu nächst eine Testumgebung geschaffen. Diese verwendet node.js, da so sich meine JavaScript Code in der Kommandozeile ausführen lässt und MySQL Schnittstelle benötigt wird. Mehr zu node.js später.

Zunächst habe ich einen MySQL Datenbank angelegt (auf meinem Server, also mit Passwort von mehreren Rechnern erreichbar). Die genau Tabellen Struktur finden Sie hinten im Anhnag.
Diese Datenbank enthält eine „Warteschlange“ (die Tabelle „scheduled_runs“). Dort warten die Tests um abgearbeitet zu werden. Die Tests haben die Attribute:

- Config -> die verschiedenen Werte für die verschiedenen Heuristiken (dies ist nur eine ID die auf die Config in einer separaten „configs“ Tabelle verweist, in der jede Heuristik für jede Config ihren eigenen Eintrag hat. So können Heuristiken entfernt und hinzugefügt werden, ohne die Struktur der Tabelle zu verändern)
- Version -> Versions Nummer der AI. Wenn Veränderungen am AI Code gemacht werden, wirkt sich das auf die Ergebnisse aus.
- Wichtigkeit -> Wenn ich schnell ein paar wichtige Test eischieben möchte, währen andere noch laufen, kann ich bei diesen die Wichtigkeit hochstellen.
-

Wenn ich die Testdatei test.js auf einem Rechner starte, holt sich diese Datei einen Test aus der Warteschlange, übergibt die Konfiguration des Tests an die AI und startet diese. Ist die AI zu Ende gelaufen, werden in die Tabelle „finished_runs“ geschrieben wie viele 2er/4er/8er/…/65536er Teile auf dem Spielbrett bei Ende des Durchlaufes sind.
So kann man genau das Ergebnis Bewerten.
Wurde eine Test eingelesen wird er aus der Warteschlange gelöscht.

Da es eine große Bandbreite an Konfigurationen gibt, hab ich mir diese zunächst ein paar JavaScript Dateien geschrieben, die mir die Konfigurationen, die SQL Befehle um diese in die Datenbank zu schreiben generiert. Zu finden unter tests/config/config_gen.js.
Außerdem hab ich eine JavaScript Dateien geschrieben, die mir eine Anzahl N Test in die Warteschlange mit der Konfiguration X einfügt. Zu finden unter tests/config/runs_gen.js.

Die Test Datei beenden sich selber, wenn sie einen Test abgeschlossen hat. Darum habe ich ein bash Script (startTest.sh). Dieses startet mehrere Test Dateien gleichzeitig und startet eine neue, wenn eine fertig ist.

Somit konnte ich durch rumprobieren mit den Richtigen Tests eine gute Konfiguration finden. Ob es die Beste ist, kann ich nicht sagen, denn:
Ich stieß auf das Problem der Rechenkapazität. Dazu eine kleine Beispielrechnung:

Zunächst wollte ich 12752 Konfigurationen testen (siehe test/configs/Test1.sql).
Jede wollte ich 100 mal durchtesten, also 100* 12752=1275200 Test.
Ein Test dauert im Durschnitt 10min auf meinem Rechner. Mit ein bisschen Optimierung könnte man sicherlich 7 Minuten erreichen. Rechnen wir also damit. 1275200*7min/60 ≈ 149000 Stunden.
Da JavaScript immer single threaded ist, brauche ich einen Kern für einen Test. Mein i7 hat 8 Kerne. Also kann ich 8 Test gleichzeitig laufen lassen: 149000 h / 8 / 24= 776 Tage.
776 Tage wäre ein bisschen Lange um alles zu Testen. Auch wenn man durch lernen aus Tendenzen der Ergebnisse Zwei Drittel verwerfen könnte, wären es immer noch ca. 250 Tage.
Würde ich mir Server im Internet mit 20 kernen mieten, komme ich auf ca. 3000€ für eine Wochen Rechnen.

Ich habe also nur ca. 7000 Test mit 620 Konfigurationen durchlaufen. Was auch schon einige Zeit gedauert hat. Ich habe allerdings 3 Rechner verwendet, die Parallel gerechnet haben, die die Warteschlange ja über meine Server erreichbar ist.

Die MySQL Datenbank liegt als SQL Dump im Ordner documents. Sie können sich aber auf meine Datenbank einloggen:

Host: vps.semklauke.de
Username: 2048project
Passwort: 2017#2048#sqlPassword

Die Auswertung habe ich jetzt etwas unschön in PHP zusammengeschustert, aber alle Werte sind für mich ablesbar.
Mit meiner jetzigen Konfiguration sind es ca. 75% der Spiele bei denen ich die 2048 erreiche.

Um zu beweisen, dass der Algorithmus wirklich effektiv ist, habe ich randomeTest.js geschrieben.
Diese Spielt das Spiel 2048 mit zufälligen Spielzügen, oder bewegt sich immer im Kreis.
Bei 200 versuchen hat ist der Zufallsalgorithmus nur einmal bis 512 gekommen (mit Kreisbewegung).

Also sollte der Algorithmus effektiv sein.

JavaScript Konzepte

Ich würde gerne noch ein paar Worte zu JavaScript und meinem Umgang damit verlieren.
JavaScript ist eher wenig typisiert und hat deshalb auch keine Richtigen Klassen.

Daher verwende ich die prototype Struktur, die einer Klasse ähnelt.
Um einfach Daten übergeben zu können, habe ich mir meine eigenen Konventionen für Objekte gegeben. Z.B. speichere ich Koordinaten immer in einem Objekt mit einem „x“ und einem „y“ Attribut.

Node.JS

Normaler weise, wird JavaScript vom Browser Interpretiert. Node.JS ist Kommandozeilen Programm, welches die V* JavaScript Engine aus dem Chrome Browser nimmt und es somit ermöglicht, JavaScript in der Kommandozeile auszuführen.
Zusätzlich kommt ein Package Manager. Es können also Module hinzugefügt werden. Damit ist es z.B. möglich sich mit JavaScript auf eine MySQL Datenbank einzuloggen oder Dateien zu lesen oder zu schrieben.
Node.Js wird für viele große Projekte verwendet (z.B. Ebay / Paypal)

Test Datenbank

configs

Field	Type	Null	Key	Default	Extra
recID	int(11) unsigned	NO	PRI	NULL	auto_increment
configID	int(11) unsigned	NO		NULL	
name	varchar(255)	NO			
value	float	NO		NULL	

versions

Field	Type	Null	Key	Default	Extra
recID	int(11) unsigned	NO	PRI	NULL	auto_increment
version_date	timestamp	NO		CURRENT_TIMESTAMP	
outdated	int(2)	NO		0	

scheduled_runs

Field	Type	Null	Key	Default	Extra
recID	int(11) unsigned	NO	PRI	NULL	auto_increment
configID	int(11)	NO		NULL	
versionID	int(11)	NO		NULL	
priority	int(11)	YES		NULL	

finished_runs

Field	Type	Null	Key	Default	Extra
recID	int(11) unsigned	NO	PRI	NULL	auto_increment
configID	int(11) unsigned	NO		NULL	
versionID	int(11) unsigned	NO		NULL	
2	int(11)	NO		0	
4	int(11)	NO		0	
8	int(11)	NO		0	
16	int(11)	NO		0	
32	int(11)	NO		0	
64	int(11)	NO		0	
128	int(11)	NO		0	
256	int(11)	NO		0	
512	int(11)	NO		0	
1024	int(11)	NO		0	
2048	int(11)	NO		0	
4096	int(11)	NO		0	
8192	int(11)	NO		0	
16384	int(11)	NO		0	
32768	int(11)	NO		0	
65536	int(11)	NO		0	

Quellen Verzeichnis

Hilfe beim Verständnis des Minimax Algorithmus:
http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html
https://www.cs.swarthmore.edu/~meeden/cs63/f05/minimax.html

Original Spiel
http://gabrielecirulli.github.io/2048/

Node.Js
https://nodejs.org/en/
https://www.npmjs.com/package/mysql2