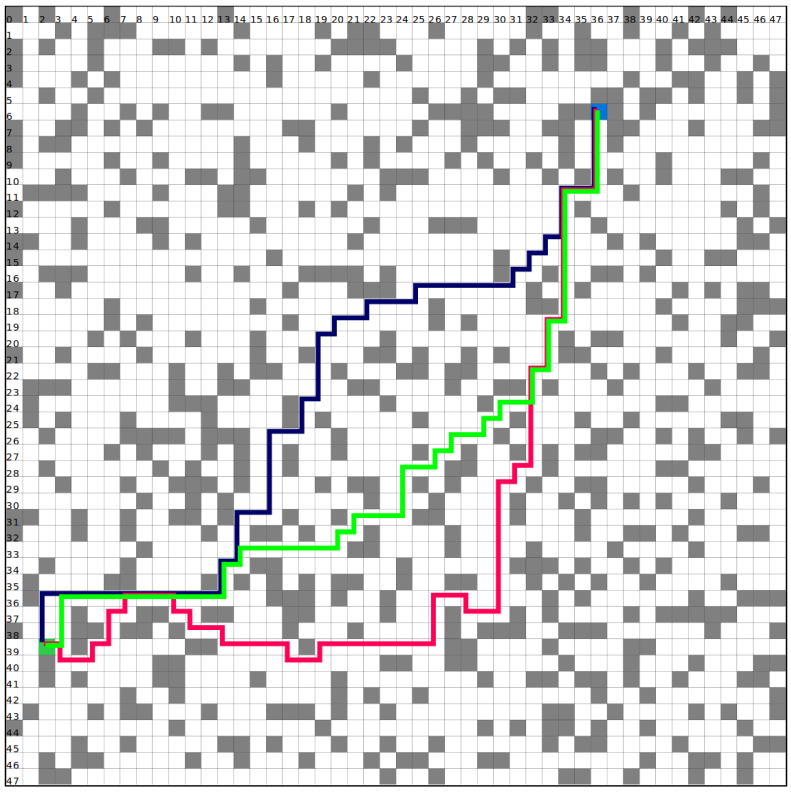
BERUFSMATURITÄTSSCHULE ZÜRICH

BERUFSMATURITÄTSARBEIT

TECHNIK, ARCHITEKTUR, LIFE SCIENCES

Pathfinding-Algorithmen: Einführung und Vergleich mittels einer Webapplikation

Oberthema: MOBILITÄT



ADRIAN STOOP
adrian-stoop@gmx.ch
EVT18a

SEVERIN FÜRBRINGER
severin@fsfe.org
EVT18a

DR. JÜRG PÖTTINGER
Begleitperson

1. Februar, 2018

Abstract

Diese Berufsmaturitätsarbeit behandelt folgende Fragestellung: Was sind die technischen Eigenschaften und Vor- und Nachteile von drei ausgewählten Pathfinding-Algorithmen und wie unterscheiden sie sich? Um diese Fragestellung zu beantworten, wird im Rahmen dieser Arbeit eine Webapplikation entwickelt, die den Benutzer in das Thema einführt und Vergleiche zwischen drei Pathfinding-Algorithmen interaktiv, parallel und mehrmals hintereinander ermöglicht. Zusätzlich erklärt die Arbeit von Grund auf, was ein Pathfinding-Algorithmus ist und beschreibt die einzelnen Pathfinder und deren Funktionsweisen mit Beispielen. Um den Vergleich der Pathfinding-Algorithmen möglich zu machen, generiert die Webapplikation automatisch Raster mit zufällig platzierten Wänden. Die Bedingungen (Parameter) der Vergleiche können vom Benutzer auch angepasst werden.

Die Auswertung der Vergleiche mittels der Webapplikation hat ergeben, dass der BreadthFirstFinder ungefähr den gleichen Weg findet wie der A*, aber um einiges mehr an Operationen und Zeit benötigt. Je grösser der Raum, desto weniger ist der BreadthFirstFinder geeignet, da er in alle Richtungen nach einer Lösung sucht und somit mit zunehmender Expansion langsamer wird. Möchte man nicht zwingend den optimalsten Weg in wenigen Operationen und Zeitaufwand, so ist der BestFirstFinder geeignet.

Die Webapplikation ist unter `bma.fuerbringer.info` zugänglich. Der dazugehörige Quellcode ist frei unter `github.com/fuerbringer/bma` erhältlich.

Inhaltsverzeichnis

1. Einleitung

1.1 Fragestellungen und Ziel der Arbeit

Algorithmen sind überall in unserem Leben zu finden, selbst dort, wo wir sie nicht erwarten: In der Natur, im Weltraum, oder in der Biologie. Da das Autorenteam dieser Berufsmaturitätsarbeit sehr technisch interessiert ist, wird in dieser Arbeit eine bedeutende Klasse der Algorithmen ausgewertet. Das vorgegebene Oberthema ist “Mobilität”. Aufgrund dessen haben wir uns für die zweidimensionalen Pathfinding-Algorithmen entschieden. Einer der bekanntesten Algorithmen in diesem Themenbereich ist der Dijkstra A* (“A-Star”). Es ist allgemein bekannt, dass er der effizienteste ist.
Die Merkmale der Algorithmen sind in unserer Zeit sehr wichtig, da sie die Vor- und Nachteile von Algorithmen bei ihrer Ausführung definieren. Interessante Merkmale sind dabei Aufwand der Rechenleistung, Zeit die benötigt wird für das Ergebnis und Tiefe der Suche nach der besten oder optimalsten Lösung. Intelligente Systeme wie zum Beispiel künstliche Intelligenz, maschinelles Lernen oder der neuesten Deep-Learning Technologie bauen auf der Algorithmentechnologie auf.
Durch Vergleiche dreier Pathfinding Algorithmen wollen wir herausfinden, welcher Algorithmus in welchen Situationen am effizientesten ist und somit prüfen, ob der A* darin tatsächlich der Beste ist. Die drei ausgewählten Pathfinding-Algorithmen sind: A*, BestFirst- und BreadthFirstFinder. Diese drei wurden vom Autorenteam ausgewählt, da im bereits bestehenden Produkt unterschiede in der Ausführung und Pfadsuche sichtbar war.
Falls einer der drei Algorithmen ein anderen Weg wählt, wollen wir den Grund dieser Entscheidung herausfinden. Bei einem anderem Weg werden andere Rechenoperationen ausgeführt. Es kann sein, dass der beste Weg zehnmal mehr Operationen benötigte, um nur wenig effizienter zu sein. Diese Arbeit möchten herausfinden, wie das Verhältnis von Aufwand zu Effizienz bei den drei Algorithmen ist.
Unsere Vermutung hinsichtlich unserer Fragestellung ist, dass die Algorithmen aus einem Uralgorithmus stammen und so technisch weiterentwickelt wurden und das der Dijkstra A* (“A-Star”) in den meisten Situationen der effizienteste Pathfinding-Algorithmus ist. [1, Andrew Walker, StackOverflow, 2012]

1.2 Vorgehen und Methoden

Wir möchten zunächst erklären, was Pathfinding-Algorithmen sind, was sie definiert und wie sie ablaufen. Um dies besser zu visualisieren und die Pathfinder zu vergleichen, wird im Rahmen dieser Berufsmaturitätsarbeit ein technisches Produkt von einer Basis aufbauend entwickelt. Das bestehende Produkt ist hier zu finden: `qiao.github.io/PathFinding.js/visual/`

Folgende Features wurden hinzugefügt oder verbessert:

- Einführung in die Berufsmaturitätsarbeit
  - Was sind Algorithmen und Pathfinding Algorithmen
  - Wozu kann man Algorithmen brauchen
  - Was für Arten Pathfinding Algorithmen gibt es auf der Webapplikation
  - Was ist das Ziel dieser Berufsmaturitätsarbeit
- Benutzerfreundliche Einführung in die Webapplikation
- Visualisierung der einzelnen Pathfinder
- Messungen der einzelnen Pathfinder
- Verbesserte Auswahl der Rastertypen
  - Vorgegebene Raster
  - Zufallsgenerierte Rastertypen
  - Definierung der Rastergröße
- Vergleich mehreren Pathfinder im gleichen Raster[1]
  - Visueller Vergleich
  - Operationenvergleich
  - Streckenvergleich
  - Rechenzeitvergleich
- Glossar mit Fachwörtern und Erklärungen

Dieses Produkt ist eine Webapplikation, die für frei zugänglich ist und deren Quellcode frei verfügbar ist. Ausschnitte aus dem technischen Produkt werden in Form von Bildern in dieser Arbeit einfliessen.

[1] Der Pathfinder-Vergleicher ist das Haupfeature unserer Arbeit.

1.3 Aufbau der Arbeit

Im ersten Teil der Arbeit wird der Leser mit den Grundkenntnissen vertraut gemacht. Dabei geht es um Algorithmen, Graphen und Heuristik. Danach werden die Funktionsweisen der ausgewählten Algorithmen mit Beispielen erklärt. Die einzelnen Vor- und Nachteile der ausgewählten Pfadfinder werden ebenfalls erläutert und danach die Realisierung des Produktes genauer beschrieben. Zum Schluss folgt ein statistischer Vergleich der drei Pathfindern durch Messresultate aus dem entwickelten Produkt.

2. Theorie der Pathfinding-Algorithmen

2.1 Grundwissen

2.1.1 Was sind Algorithmen

Ein Algorithmus beschreibt eine abstrakte Form eines Aufgabenlösungswegs. Er besteht aus klaren Handlungsvorschriften, die in Form einer Kette oder Reihe in Einzelschritten abgearbeitet werden, um ein Problem zu lösen. Ein Algorithmus kann den Satzaufbau einer Sprache vorschreiben (Subjekt, Verb, Objekt) oder die Herstellung eines Gerichts in der Küche als Kochrezept. Es wird immer eine bestimmte Eingabe in eine bestimmte Ausgabe verarbeitet. Das Ergebnis ist dabei immer das Gleiche, ähnlich wie bei einer Funktion in der Mathematik.
Es gibt sehr einfache Algorithmen, wie Verhaltensregeln oder Gesetze, in denen man nachschauen kann, wie man sich in welcher Situation verhält oder was erlaubt ist. Schwierigere Algorithmen findet man eher in der Mathematik, Datenanalyse oder auch im Bewusstsein der Menschen. [2, Wikipedia, 2018]

2.1.2 Was ist ein Pathfinding-Algorithmus

Die Pathfinding-Algorithmen sind eine Unterklasse der Algorithmen, sie spezialisieren sich auf die suche des optimalsten Wegs von einem Punkt A zu einem anderen Punkt B in einem Raum. Der Raum oder die Matrix kann hierbei zweidimensional oder dreidimensional sein, in dieser Arbeit befassen wir uns nur mit den Pathfinding-Algorithmen im zweidimensionalen Raum.
Bei Computerspielen kann der Raum das Spielfeld sein, wo unter anderem eine Computergesteuerte Person (künstliche Intelligenz = KI) immer auf den Spieler zuläuft oder ihn in einem definierten Abstand umkreist.
Bei der heutigen Internetinfrastruktur helfen Pathfinding-Algorithmen unsere Daten über Routenplanung effizient von unserem Computer zum Server der aufgerufenen Website zu leiten und wieder zurück.
Die Pathfinding-Algorithmen suchen sich generell den kostengünstigsten oder kürzesten Weg mit wenig Hindernissen aus. Ein Vogel kann mühelos über einen Berg fliegen und legt bloss die direkte Luftlinie als Wegstrecke zurück, wir Menschen müssen meistens um den Berg herum reisen oder ihn in schlangenlinigförmigen Wanderwegen überwinden.

Ob dieser Wanderweg der optimalste ist, hängt von verschiedenen Faktoren ab, da es eventuell sinnvollere Wege zum Ziel gibt.

- **Zeitoptimierung**: Man möchte am schnellsten am Ziel sein, vorzugsweise sucht man sich eine öffentliche Fortbewegung wie ein Bus.
- **Kostenoptimierung**: Man hat nicht unbegrenzt Geld und sucht sich den kostengünstigsten Weg heraus.
- **Wegoptimierung**: Man möchte auf der Wanderung von gewissen Orten Bilder mit der Kamera festhalten.

Man beachte, dass egal welcher Weg ausgewählt wurde, dieser immer Vor- und Nachteile aufweist. Dies kann man sich unter einem Kuchendiagramm vorstellen. Es gibt drei Stücke: Zeitoptimierung, Kostenoptimierung und Wegoptimierung. Egal für welches Stück bevorzugend man den Kuchen unterteilt, die anderen werden darunter "leiden". [3, Wikipedia, 2018]

2.2 Einführung der Graphen

Pathfinding-Algorithmus "sehen" keine Wände oder Start- und Endpunkte. Sie verarbeiten lediglich eingegebene Daten in ausgehende Daten (Input und Output). In unserer Webapplikation vereinfachen die Pathfinding-Algorithmen die Darstellung des Raumes in nicht-negative Graphen (siehe Unterabschnitt 2.2.2). Daraus kann abgeleitet werden, dass die Pathfinding Algorithmen komplett unabhängig von der visuellen Darstellung des Raumes arbeiten. [4, Andreas Hofmann, 2013]

2.2.1 Was ist ein Graph: Graphentheorie und Pathfinder

Ein Graph ist eine mathematische Struktur zur Darstellung abstrakter Beziehungsstrukturen und besteht aus zwei Elementen: Ecken/Knoten (engl. vertices) und Kanten (englisch edges). Jede Kante verbindet exakt zwei Ecken, es können aber mehrere Kanten auf eine Ecke verweisen. Jeder angeschaute Knoten vom Algorithmus muss über mindesten eine Verbindung erreichbar sein.

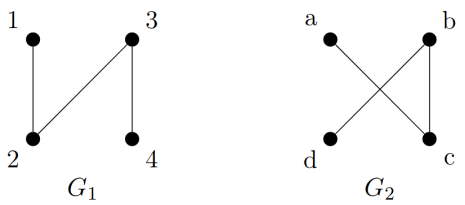


Abbildung 2.1: Zwei einfache Graphen G_1 und G_2 mit Ecken und Kanten. Quelle: Benny Sudakov, *Graph Theory*, 2016, ETH Zürich.

Die Knoten stellen bei unserer Anwendung die x- und y-Koordinaten des Raums dar und die Kanten die Verbindungen der einzelnen Felder. Somit hat ein Knoten bei Ausführung des bestimmten Algorithmus 4 Kanten. Wenn dem Pathfinder die Diagonale zugelassen werden, gelten für die Knoten 8 Kanten. [5, Franz Embacher, 2003]

2.2.2 Nicht negativ gewichtet

In der allgemeinen Graphentheorie sind negativ gewichtete Graphen erlaubt. Bei den Pathfindern jedoch nicht, da es keine negativen Distanzen gibt. Von Punkt A nach B gilt eine positive Gewichtung der Strecke und der Rückweg von B nach A ebenfalls eine positive Gewichtung der Strecke in den Knoten. [6, Krumke; Noltemeier; Schwarz; Wirth, 2000, S. 76]

2.2.3 Pathfinding-Graphen

In unserem Raster kann man sich die Graphenstruktur wie ein Schachbrett vorstellen. Es beginnt am Startpunkt und sucht Graph um Graph nach dem Ziel. In jedem Graph wird gespeichert von wo und wessen Nachbargraph man kam und wie weit man vom Startpunkt entfernt ist. Findet man den gleichen Graph ein zweites mal, durch einen anderen Weg, wird der schnellere Weg zum start in den Graph geschrieben. Findet die Struktur das Ziel, so verweist jeder Vorgehende Graph auf den Graph davor. Diese Graphenkette zeigt dann vom Ziel zum Start. [7, Vinther, Vinther, 2015, S. 22]

2.3 Heuristiken

Im Allgemeinen können Algorithmen auf eine Heuristik zugreifen, um optimierter nach einem Weg zu suchen. Der Begriff “Heuristik” bedeutet mit begrenztem Wissen oder unvollständiger Informationen die optimale Lösung im Vorhinein zu schätzen. Anders gesagt teilt die Heuristik dem Algorithmus mit, welcher Graphen am wahrscheinlichsten zum Zielpunkt führt.
Bei den ausgewählten Pathfinding-Algorithmen beim Vergleich benutzt der A* und BestFirst-Finder die Manhattan Heuristik und der BreadthFirstFinder keine. Der BreadthFirstFinder benutzt in seiner Standardkonfiguration nie eine Heuristik. Die Manhattan Heuristik, auch Cityblock-Metrik, wurde ausgewählt, da das Raster dieser Berufsmaturitätsarbeit einem Schachbrett ähnelt, mit immer dem gleichen Abstand zwischen den Feldern.
Die Manhattan-Heuristik wird folgend für alle Punkte im Raster definiert:

$$d\big((x_1,y_1),(x_2,y_2)\big) = |x_1 - x_2| + |y_1 - y_2|$$

Die Distanz d ist gleich der Länge aller Wege, die P_1 und P_2 entlang der horizontalen und vertikaler Linie verbindet. P_2 ist bei dieser Arbeit immer der Zielpunkt und P_1 der aktuelle Standpunkt im Raster. [8, Patel, 2019]

2.4 Auswahl der Pathfinding-Algorithmen

Das Autorenteam dieser Arbeit hat sich für folgende Pathfinding-Algorithmen entschieden, da diese im Vorfeld schon ersichtlich unterschiedliche Wege im gleichen vorgegebenen Raster ergeben haben.

2.4.1 A*

Der A*-Algorithmus, gesprochen "A Star", untersucht mithilfe einer Heuristik immer den wahrscheinlich nächsten Knoten im Raster, der zum Ziel führt. Jedem Nachbarknoten wird zuerst einen Wert d zugeordnet, der angibt, wie lange der Weg geschätzt von diesem Knoten zum Ziel führt. Der Wert d wird mit der Manhattan-Heuristik ausgerechnet. Jeder Graph mit einem d-Wert wird in eine Liste eingetragen. Der A-Star rechnet immer die neuen Nachbarn des Graphen mit dem kleinsten d-Wert aus. Dadurch kann es vorkommen, das nach langer Suche der A-Star plötzlich wieder einige Graphen zurückspringen muss, da die neu berechneten d-Werte grösser als ein früherer gefundener d-Wert ist. [9, Schmidt, Fuchs]

Funktionsweise

Alle Graphen werden in drei Listen eingetragen:

- Unbekannte Knoten

Beim Start der Suche sind alle Knoten im Raster in dieser Liste.

- Bekannte Knoten

  Wurde einem Knoten der d-Wert der Heuristik zugeteilt, wird er in diese Liste mit dem d-Wert eingetragen. Aus dieser Liste wird immer der Knoten mit dem kleinstem d-Wert ausgewählt, der als nächstes angeschaut wird.

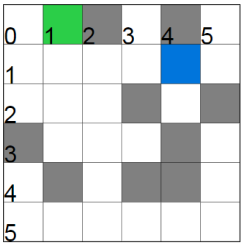
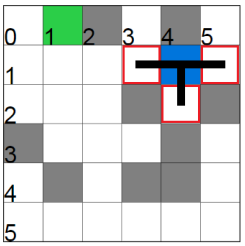
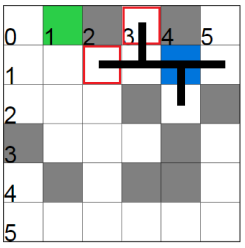
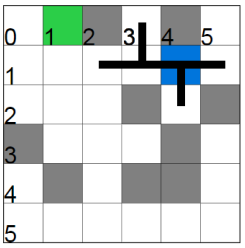
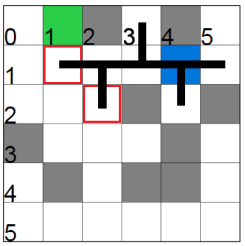
- Untersuchte Knoten

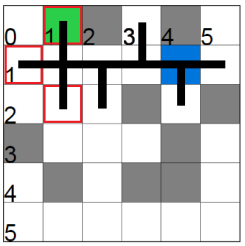
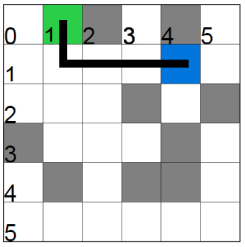
  Zu diesem Knoten wurde der kleinste Weg herausgefunden, das heisst alle Nachbarknoten sind in der Liste mit bekannten Knoten.

Jeder der gefundenen Knoten besitzt einen Verweis auf den Vorgängerknoten. Wird das Ziel erreicht, kann mit Hilfe dieses Zeigers der Pfad bis zum Start ausgegeben werden. [9, Schmidt, Fuchs]

Beispiel

Der Startpunkt ist blau und der Zielpunkt grün, wobei diagonale Verbindungen nicht verfügbar sind.

Bild	Ausführung	Bekannte Knoten	Untersuchte Knoten
	Schreibt Startpunkt in bekannte Liste mit Distanzwert.	$(4/1)\ d=4$	–
	Berechne Distanzwert aller Nachbarn (rot markiert) und schreibe diese in die bekannte Knoten-Liste. Zeige auf den Vorgängigen Knoten (schwarzer Balken). Da $(4/1)$ alle bekannte Nachbarn hat, wird dieser Knoten in die Knoten-Liste der Untersuchten verschoben. Der Knopf mit den tiefsten d-Wert wird Angeschaut und der Prozess wird wiederholt.	$(3/1)\ d=3$ $(5/1)\ d=5$ $(4/2)\ d=5$	$(4/1)\ d=4$
	Da der Graph(3/1) Alle bekannten Nachbarn hat, wird dieser Knoten in der untersuchten Knoten-Liste verschoben.	$(3/0)\ d=2$ $(2/1)\ d=2$ $(5/1)\ d=5$ $(4/2)\ d=5$	$(3/1)\ d=3$ $(4/1)\ d=4$
	Falls Graph $(3/0)$ vor dem Graph $(2/1)$ angeschaut wird (gleiche d-Werte), wird dieser, da er keine gültigen Nachbarn hat, in die untersuchten Knoten-Liste geschrieben.	$(2/1)\ d=2$ $(5/1)\ d=5$ $(4/2)\ d=5$	$(3/0)\ d=2$ $(3/1)\ d=3$ $(4/1)\ d=4$
	Der Prozess wird weitergeführt. Die Listen werden kontinuierlich aktualisiert.	$(1/1)\ d=1$ $(2/2)\ d=2$ $(3/1)\ d=3$ $(5/1)\ d=5$ $(4/2)\ d=5$	$(3/0)\ d=2$ $(2/1)\ d=2$ $(4/1)\ d=4$

	Der A* hat das Ziel erreicht und muss noch den einzelnen Graphen zurückfolgen bis zum Start und kennt somit den Weg.	$(1/0)\ d=0$ $(0/1)\ d=2$ $(1/2)\ d=2$ $(2/2)\ d=2$ $(3/1)\ d=3$ $(5/1)\ d=5$ $(4/2)\ d=5$	$(1/1)\ d=1$ $(3/0)\ d=2$ $(2/1)\ d=2$ $(4/1)\ d=4$
	Die Suche ist Beendet.	–	–

2.4.2 BestFirstFinder

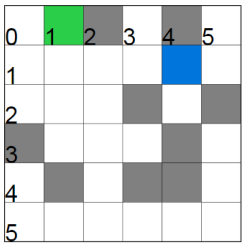
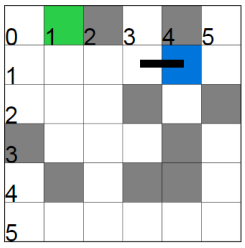
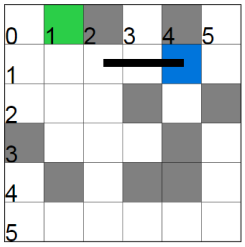
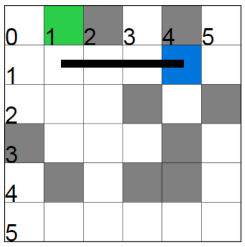
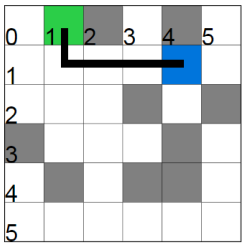
Mit der Benutztung der Mannhattan-Heuristik untersucht der BestFirstFinder nur den vom angeschautem Graph besten Nachbarsgraph, dies ist der Graph mit dem tiefsten Heursitikwert. Jeder beste Graph wird dann in die offene Liste eingetragen und die Suche beim nächsten Graphen wiederholt. Falls nun vom früheren Graph ein kleinerer Heursitikwert berechnet wurde und der jetztige Graph grössere Heuristikwerte in den neuen Nachbarn finden, geht der BestFirstFinder nicht zurück, ausser er landet in einer Sackgasse.

Funktionsweise

Es wird eine Liste für die offenen Knoten erstellt und der Startknoten eingetragen. Auch hier verweist jeder Knoten auf seinen Vorgänger. Der beste Knoten aus der Liste wird *n* genannt. Der Algorithmus schaut die Nachbarn von *n* an und bewertet diese mit der Heuristik, wobei der beste Wert in die Liste eingetragen und der Ablauf wiederholt wird. Kommt ein Knoten, der angeschaut wird nicht weiter, wird der zweitbeste Graph in der Liste angeschaut. Ee kann einer weiteren Liste hinzugefügt werden, der geschlossenen Liste, die den Lösungsweg direkt abspeichert und verhindert, dass dieser Algorithmus nicht in einer Endlosschleife endet. [10, Roblox Developer, 2018]

Beispiel

Der Startpunkt ist blau und der Zielpunkt grün, wobei diagonale Verbindungen nicht verfügbar sind.

Bild	Ausführung	Liste
	Schreibe Startpunkt in die Liste.	$(4/1)\ d=4$
	Wähle den besten Nachbar, anhand seiner Distanzwert d, hier ist es Graph $(3/1)$. Schreibe diesen Wert in die Liste.	$(3/1)\ d=3$ $(4/1)\ d=4$
	Wiederhole die Ausführung bis das Ziel erreicht wurde.	$(2/1)\ d=2$ $(3/1)\ d=3$ $(4/1)\ d=4$
	–	$(1/1)\ d=1$ $(2/1)\ d=2$ $(3/1)\ d=3$ $(4/1)\ d=4$
	Der Weg wurde gefunden.	$(1/0)\ d=0$ $(1/1)\ d=1$ $(2/1)\ d=2$ $(3/1)\ d=3$ $(4/1)\ d=4$

2.4.3 BreadthFirstFinder

Der BreadthFirstFinder benutzt keine Heuristik, da er seinen Weg über die Breitensuche findet. Er expandiert in jede Richtung gleich schnell und bewegt sich nicht wie die anderen zwei Pathfinder in Richtung Ziel. Wenn alle Knoten mit der gleichen Distanz zum Start angeschaut sind, beginnt er mit der Suche der um eins weiter entfernten Knoten. Dieser Pathfinder besitzt eine Liste in der er die noch zu bearbeitenden Knoten speichert, beginnend mit dem am Start

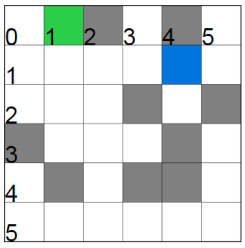
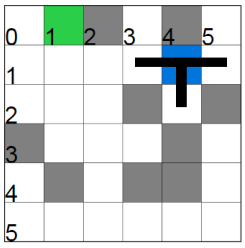
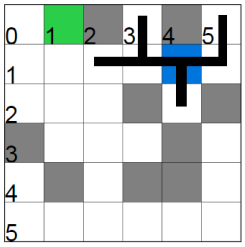
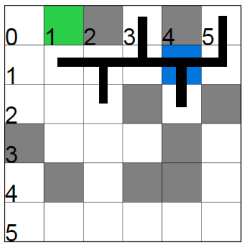
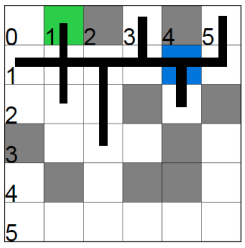
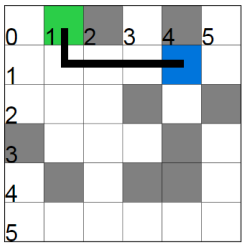
nächsten Knoten. [11, Brilliant.org]

Funktionsweise

Der Algorithmus beginnt beim Startpunkt und speichert ihn in die Warteliste. Er schaut am obersten Knoten in der Warteschlange dessen Nachbarn an und prüft, ob einer der ausgewählten Knoten der Zielknoten ist. Ist dies der Fall, wird die Suche abgebrochen, da der Weg gefunden wurde. Falls keiner der Knoten der Zielknoten ist, speichert er die neuen Knoten in der Warteliste an hinterster Stelle und entfernt den ausgewählten Knoten. Jeder Knote, der neu gefunden wird, zeigt auf den ausgewählten Knoten.

Beispiel

Der Startpunkt ist blau und der Zielpunkt grün. Diagonale Verbindungen sind nicht verfügbar.

Bild	Ausführung	Liste
	Schreibe den Startpunkt in die Liste.	$(4/1)$
	Schaue alle Nachbarn an und speichere sie in die Liste an hinterster Stelle. Falls keiner der neuen Knoten das Ziel ist, führe die Suche fort. Jeder neue Graph verweist auf den Vorgänger. Entferne den aktuell ausgewählten Knoten $(4/1)$.	$(3/1)$ $(5/1)$ $(2/1)$
	Schaue alle Nachbarn der zurzeit in der Liste stehenden Knoten und wiederhole alles vom oben genannten.	$(3/1)$ $(2/1)$ $(5/1)$
	–	$(1/1)$ $(2/2)$
	Das Ziel wurde gefunden und die Suche ist somit beendet. Folge vom Ziel zum Start und gib den Lösungsweg aus.	$(0/1)$ $(1/0)$ $(1/2)$ $(2/3)$
	Der Weg wurde gefunden.	–

2.5 Vor- und Nachteile der einzelnen Pathfinder

Anhand der Beispiele am gleichen Raster sieht man sehr gut, dass alle drei Pathfinder den gleichen Weg durch unterschiedliche Methoden finden.
Der A-Star führt viele Heuristikberechnungen durch, da er von jedem Knoten die Distanz zum Ziel speichern möchte. Dies erfordert mehr Rechenleistung, beziehungsweise mehr Operationen als die beiden anderen. Bei komplexeren Rastern wird er jedoch immer den besseren Weg finden.
Der BestFirstFinder untersucht nur den zuerst gefundenen besten Weg und sucht daher zuerst in der Tiefe. Er braucht weniger Rechenleistung als der A-Star, kann aber Mühe bekommen, wenn der erst gesuchte Weg in eine Sackgasse führt. In diesem Beispiel musste er weniger Knoten anschauen als der A-Star, obwohl er die gleiche Heuristik benutzt.
Der BreadthFirstFinder sucht alle ihm am nächsten gelegenen Knoten ab und wird immer auf eine Lösung kommen, auf Kosten der "unnötigen" Knoten, die weiter entfernt zum Ziel als seine aktuelle Position sind. Die Warteliste wird mit jeder neuen Expansion um ein Vielfaches grösser und durch somit über Dauer langsamer.

3. Realisierung

Dieser Abschnitt soll dazu verhelfen, die technischen Entscheidungen und Schritte zu erläutern und es möglich zu machen, die Kernfunktionen des Projekts zu reproduzieren. Leser, die diese Applikation, oder Teile davon, anderweitig einsetzen möchten, beziehen sich zusätzlich auf den frei verfügbaren Quellcode[14] des gesamten Projekts.

3.1 Konzept

Die Konzipierung dieser Webapplikation verlangte einige technische und gestalterische Entscheidungen. Einerseits musste die Programmiersprache und Struktur der Applikation geplant werden und andererseits auch das eigentliche Aussehen der Benutzeroberfläche der Webapplikation. Auf diese zwei wesentlichen Aspekte wird in diesem Abschnitt eingegangen.

3.1.1 Pathfinding.js

Eine korrekte Implementation der ausgewählten Pathfinder sind Voraussetzung für das Projekt. Daher wurde, um diese Voraussetzung zu erfüllen, die frei verfügbare Implementation PathFinding.js [12] verschiedener Pathfinder ausgewählt.

3.1.2 Programmierwerkzeuge

Die vorhandenen Fachkenntnisse und Erfahrungen schlugen für die Implementierung entweder PHP oder JavaScript als mögliche Programmiersprachen vor. Zum Entscheid massgebend war, dass die Programmbibliothek PathFinding.js [12], welche in unserer Arbeit eine wichtige Rolle spielt, in JavaScript implementiert wurde. Durch den Einsatz von JavaScript im Front-End und im Back-End, wäre es möglich die Pathfinder Berechnungen beliebig auf dem Rechner des Nutzers und auch auf dem Rechner des Servers auszuführen. Aus diesen Gründen wurde für das Front- und Back-End JavaScript ausgewählt. Folgend werden die wichtigsten Technologien unserer Webapplikation aufgelistet[1].

Programmiersprache JavaScript mit Einbindung von PathFinding.js [12]

[1] Eine komplette Auflistung inklusive der abhängenden Programmbibliotheken macht wenig Sinn, vor allem da unser Quellcode frei ersichtlich ist.

Webserver-Software Express[2] übernimmt die Ausführung von JavaScript im Back-End.

Quellcode-Hosting GitHub[3]. Der Quellcode unser Webapplikation ist frei unter `https://github.com/fuerbringer/bma` zugänglich.

Webhosting Die Vultr[4]-Cloud dient als Hosting-Provider. Prinzipiell kann die Webapplikation jedoch auf beliebigen Unix und GNU/Linux Servern gehostet werden.

3.1.3 Konzipierung der Benutzeroberfläche

Als Webapplikation muss unsere Arbeit deren Nutzen dem Benutzer ausreichend klar machen. Umsomehr ist dies von Bedeutung, da die Webapplikation frei zugänglich ist und daher auch Besucher die Idee der Webapplikation verstehen sollten. Aus diesem Grund wurde unsere Webapplikation für den Nutzer einführend gestaltet. Das heisst, dass der Benutzer beim Aufruf der Webapplikation zunächst auf einer Willkommensseite landet, die ihn wiederum zu einer Einführung in das Thema und Ziel der Arbeit führt. Nach der Einführung kann der Benutzer in der Visualisierung mit einzelnen Pathfindern erste Erfahrungen machen. Im Abschluss kommt der Benutzer zum Hauptteil der Arbeit, dem Pathfinding-Vergleicher. Ziel dieser Struktur ist, dass jede Person wenigstens einen Einblick in die Welt der Pathfinding-Algorithmen erhält und über das nötige Wissen verfügt, das Ziel des Pathfinding-Vergleichers zu verstehen.

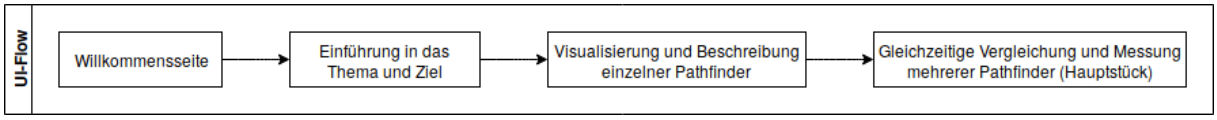


Abbildung 3.1: Webapplikationsstruktur. Quelle: Eigenleistung

[2]Express Webserver für Node Webapplikationen, `https://expressjs.com/`, Stand: 29. Januar 2019
[3]GitHub, `https://github.com/`, Stand: 29. Januar 2019
[4]Vultr - The Infrastructure Cloud™, `https://vultr.com/`, Stand: 29. Januar 2019

Einführungs- und Willkommensseite

Die Einführungsseite erklärt dem Benutzer was Algorithmen sind und führt ihn anschliessend mit einfachen Beispielen in das Thema der Pathfinder ein. Zuletzt werden die Ziele der Arbeit aufgelistet.

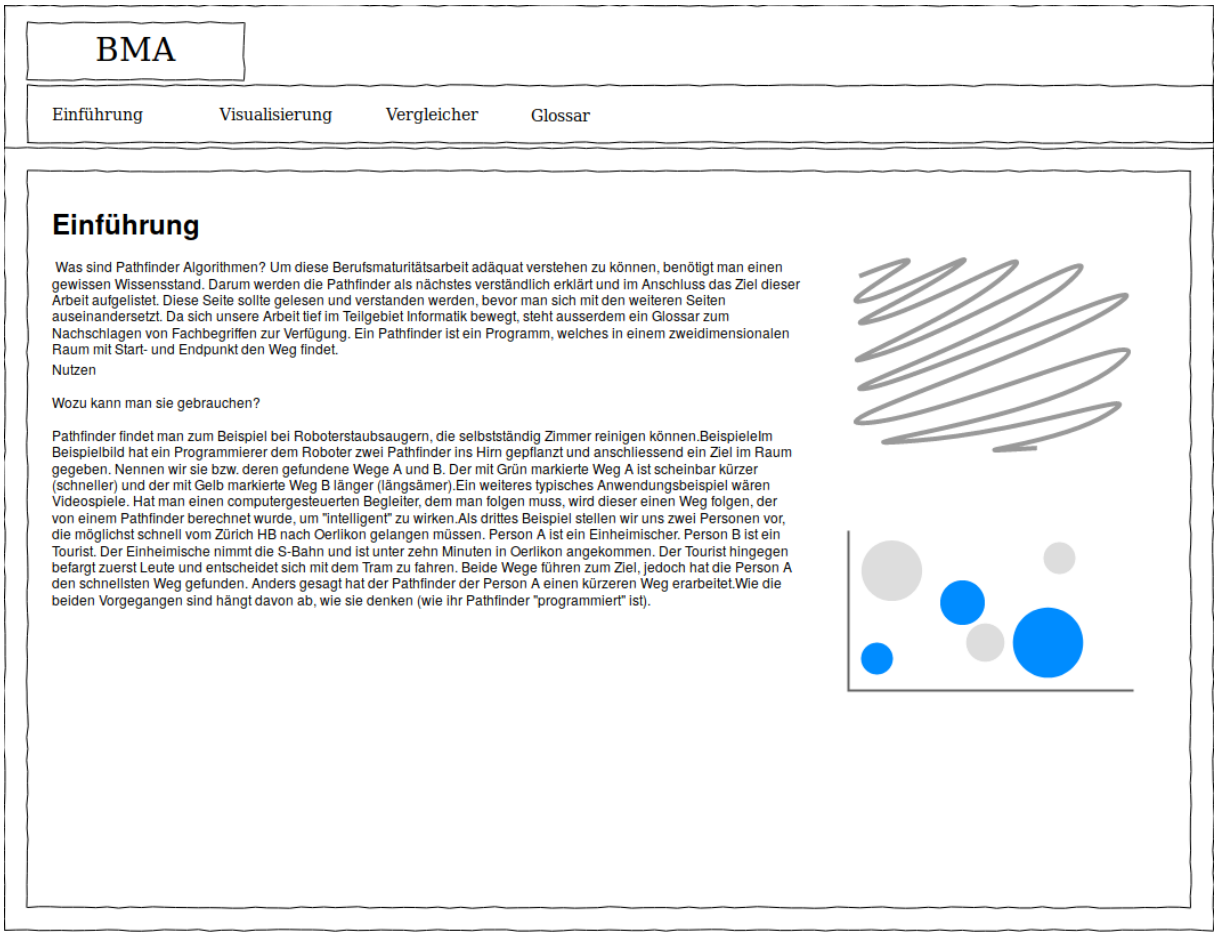


Abbildung 3.2: Einführungsseite. Quelle: Eigenleistung

Visualisierung der Pathfinder

Nach der Einführung erhält der Benutzer die Möglichkeit mit einzelnen Pathfindern zu experimentieren. Dazu kann er auch verschiedene Parameter anpassen. Dies ist die Vorstufe zum Pathfinder-Vergleicher, da hier nochmals die einzelnen Pathfinder auf ihre Eigenschaften beschrieben werden.

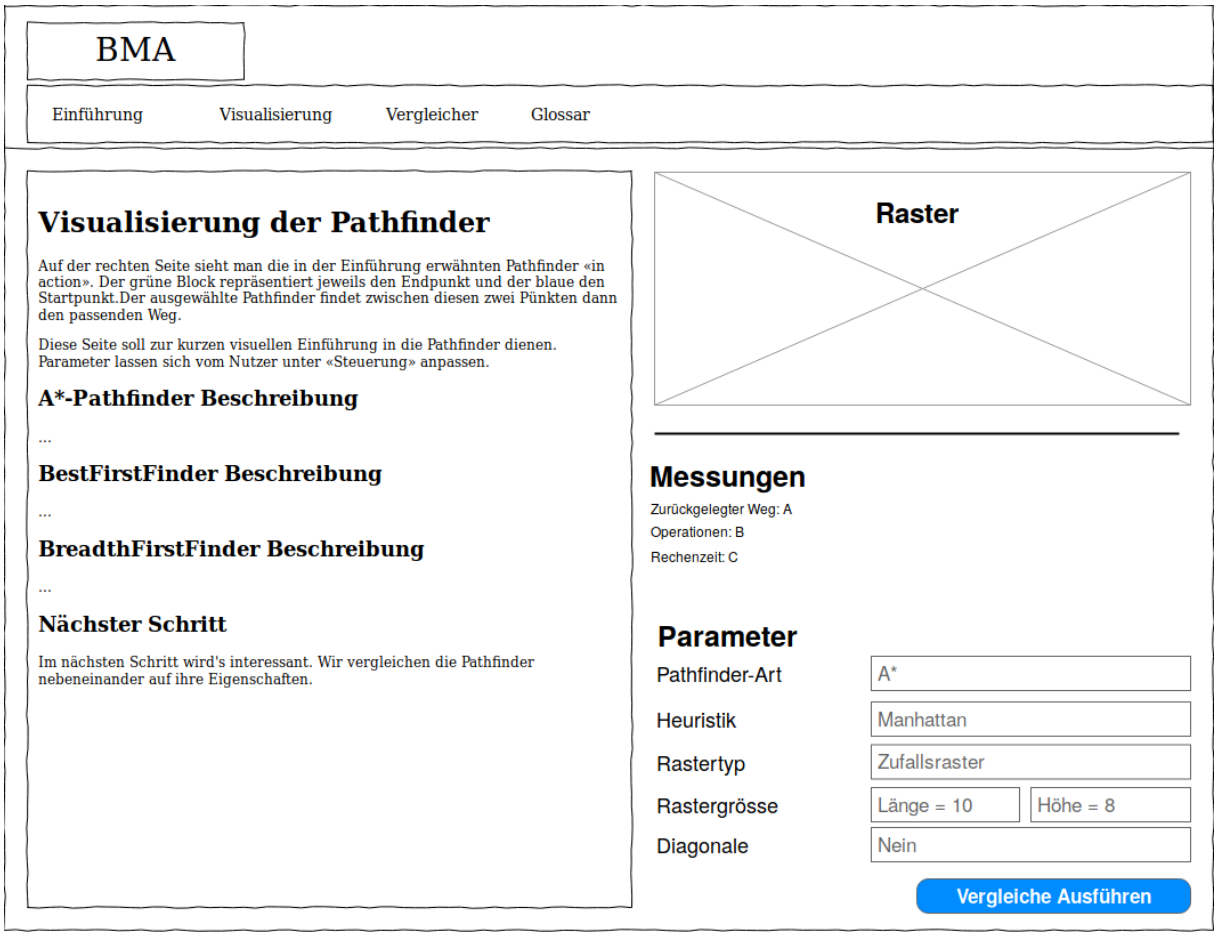


Abbildung 3.3: Benutzeroberflächenkonzept Visualisierer. Quelle: Eigenleistung

Vergleich der Pathfinder

Als Herzstück der Arbeit ermöglicht der Pathfinder-Vergleicher dem Nutzer die in unserer Arbeit ausgewählten Pathfinder zu vergleichen. Hier werden, gleich wie beim Visualisierer, zuerst vom Benutzer die Parameter gewählt. Die parallel ausgeführten Resultate mit Informationen über die Rechenzeit, zurückgelegten Weg und Operationen, sind dann interaktiv anwählbar. Gleichzeitig wird auch unter “Resultate” unten links das Total aller Vergleiche akkumuliert, damit ein Gesamteindruck verschaffen werden kann. Die Resultate dieser Seite dienen im Statistikteil als Datenquelle.

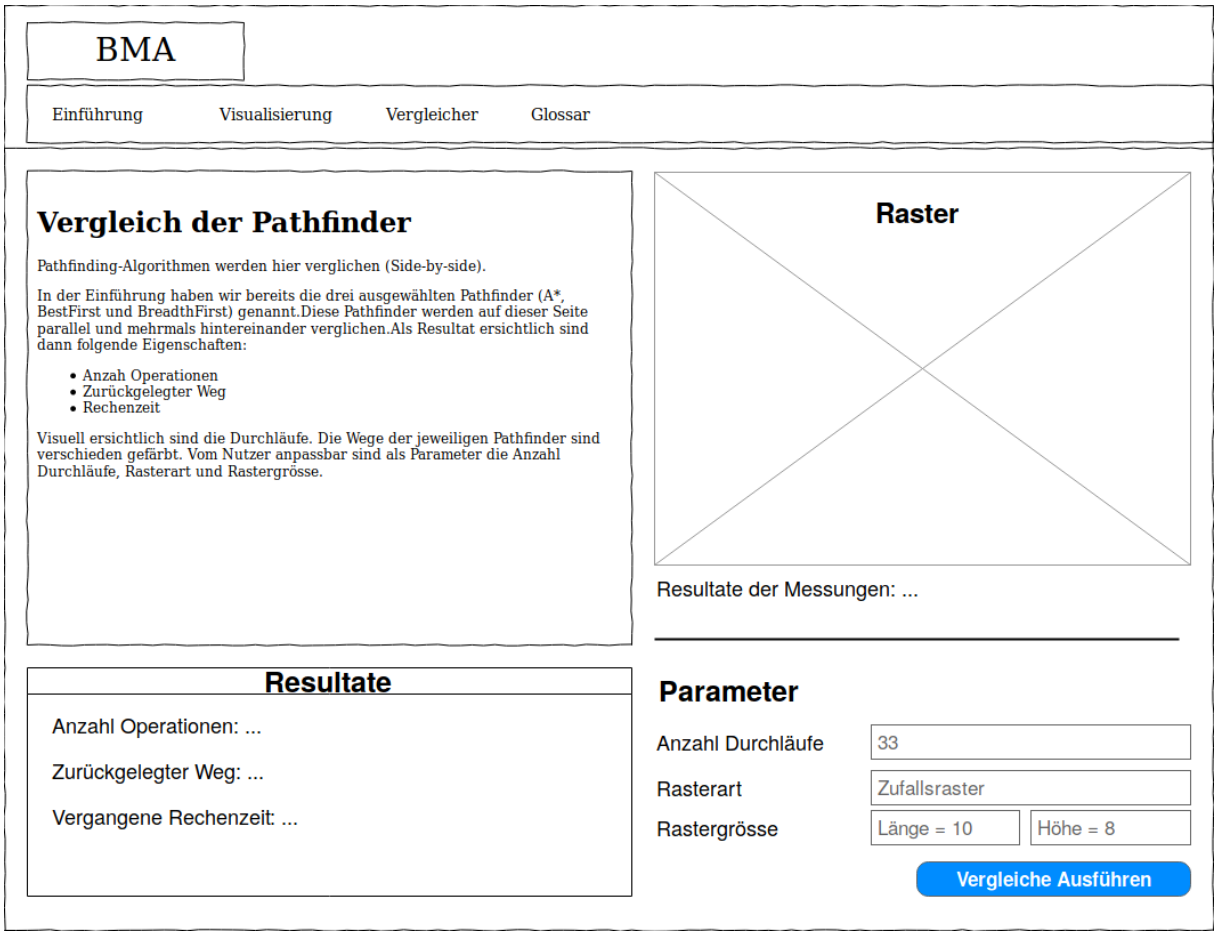


Abbildung 3.4: Benutzeroberflächenkonzept Vergleicher. Quelle: Eigenleistung

3.2 Implementierung des Vergleichers

Vor allem für die statistischen Auswertungen ist es von Bedeutung zu wissen, wie der Vergleicher zu den Resultaten kommt und die Berechnungen durchführt. Um an die statistischen Merkmale zu gelangen, musste je nach Merkmal die Webapplikation oder PathFinding.js erweitert werden.

3.2.1 Raster

Pathfinder benötigen einen Raum, um ihre Arbeit zu verrichten. Folglich wird erläutert, wie diese Räume im Quellcode der Arbeit implementiert wurden. Die Implementation der Pathfinding-Algorithmen von PathFinding.js erwartet als Raum eine Liste mit beispielsweise 8×8 Zellen:

$$
\begin{gathered}
[(0,0),(1,0),(2,0),(3,0),(4,0),(5,0),(6,0),(7,0)] \\
[(0,1),(1,1),(2,1),(3,1),(4,1),(5,1),(6,1),(7,1)] \\
[(0,2),(1,2),(2,2),(3,2),(4,2),(5,2),(6,2),(7,2)] \\
[(0,3),(1,3),(2,3),(3,3),(4,3),(5,3),(6,3),(7,3)] \\
[(0,4),(1,4),(2,4),(3,4),(4,4),(5,4),(6,4),(7,4)] \\
[(0,5),(1,5),(2,5),(3,5),(4,5),(5,5),(6,5),(7,5)] \\
[(0,6),(1,6),(2,6),(3,6),(4,6),(5,6),(6,6),(7,6)] \\
[(0,7),(1,7),(2,7),(3,7),(4,7),(5,7),(6,7),(7,7)]
\end{gathered}
$$

Zu beachten ist, dass die obige Liste pro Element (Zelle) lediglich die Koordinate (x,y) als Eigenschaft aufweist. Weitere Eigenschaften pro Zelle, wie die Wandeigenschaft, werden in späteren Abschnitten erwähnt. Diese Art von Raster ist bei der Darstellung von Pixelmatrizen und sonstigen computergrafikorientierten Anwendungen üblich. Man kann sich diese Anordnung auch als den vierten Quadranten eines kartesischen Koordinatensystems vorstellen, dessen y-Koordinate ihren Absolutwert annimmt. Des Weiteren gilt $x \in \mathbb{N}$ und $y \in \mathbb{N}$. Daraus folgt, dass Zwischenschritte nicht möglich sind, beispielsweise das Ziel nicht die Koordinate $(\frac{1}{2}, \frac{42}{11})$ annehmen kann. Ein weiteres Beispiel dafür wären die Pixel eines gewöhnlichen Monitors, denn einen Bruchteil eines Pixels anzusteuern ist ebenfalls nicht möglich. Benötigt man eine derartige Präzision, erhöht man dazu am besten die Auflösung des Raums. Es folgt eine Abbildung eines zweidimensionalen leeren Raumes der oben gegebenen Liste.

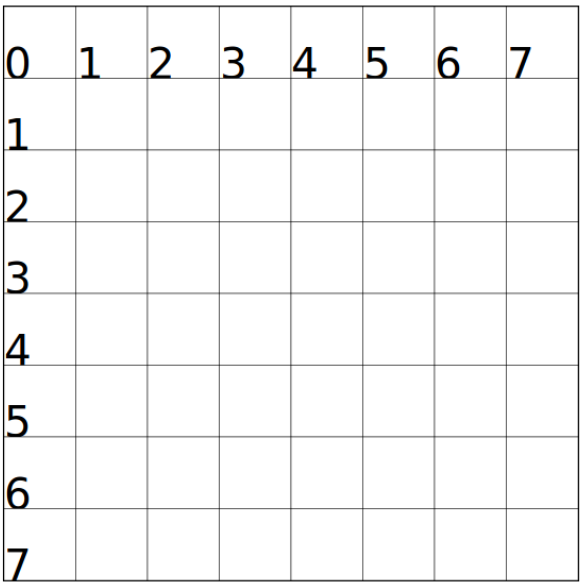


Abbildung 3.5: Ein von unserer Webapplikation generiertes 8×8 Raster. Quelle: Eigenleistung

Generierung der Wände und Gänge

Damit die Eigenschaften der Pathfinder besser zum Vorschein kommen, sie also nicht nur im leeren Raum ausgeführt werden, wurde im nächsten Schritt für den Rastergenerator die Logik zum Verstreuen von Wänden implementiert. Die Rastermatrix erhält dann pro Zelle die Eigenschaft, ob sie vom Pathfinder passierbar ist oder nicht. Die Programmbibliothek PathFinding.js versteht diese Logik genauso, indem man zu einer gegebenen Koordinate definiert, ob sie eine Wand ist. Der folgende prozedurale Pseudocode definiert die in unserer Webapplikation benutzte Routine[5], um Raster mit Wänden zu füllen.

1: **procedure** WANDGENERATOR(*raster*, $freq$)
2: **for** $y \leftarrow 0$ **to count**(Zeilen von *raster*) **do**
3: **for** $x \leftarrow 0$ **to count**(Zellen von *raster* in der Zeile y) **do**
4: $rand \leftarrow random(0, 100)$ ▷ Zufallszahl $[0, 100]$
5: $x \leftarrow \lfloor rand \mod freq \rfloor$ ▷ Rest von $\frac{rand}{freq}$
6: **if** $x = 0$ **then**
7: Zelle in *raster* bei x und y wird zu einer Wand
8: **return** *raster*

Vom Frequenzparameter $freq$ hängt schlussendlich ab, wie oft Wände im Raster vorkommen. Er kann frei gewählt werden. Nähert sich der Wert $freq$ jedoch 1 mit $freq > 1$, erhöht sich auch die Anzahl unlösbarer Raster, da immer weniger mögliche Wege zwischen den verbliebenen Zellen besteht. Wendet man den Wandgenerator in unserem Raster der Abbildung 3.5 an, so erhält man ein Raster mit Wänden.

[5]Die genaue Codestelle in der Webapplikation ist hier ersichtlich: `https://github.com/fuerbringer/bma/blob/master/src/maze.js#L8`

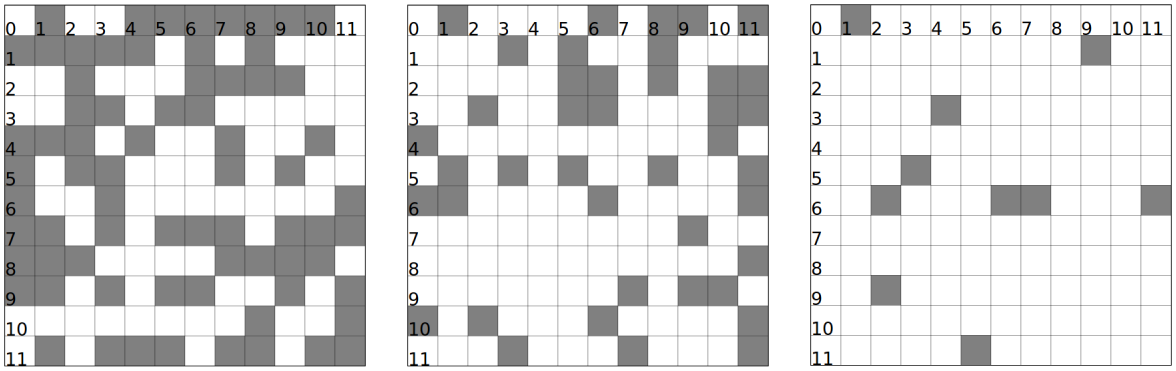


Abbildung 3.6: Raster mit Frequenzparametern $freq = 2$, $freq = 4$ und $freq = 16$ im Wandgenerator. Quelle: Eigenleistung

Wir sehen also, dass mit dem Frequenzparameter $freq$ die Häufigkeit der unpassierbaren Wände geändert werden kann. In unserer Webapplikation haben wir für den Frequenzparameter $freq = 4$ ausgewählt, da dieser Wert ein gutes Gleichgewicht zwischen nicht zu viel unlösbaren Rastern und genügend Wänden ergibt.

Generierung des Start-und Endpunkts

Wählt man zwei zufällige passierbare Zellen als Start- und Endpunkt aus, erhält man ein Raster, mit dem man Pathfinding-Algorithmen anwenden kann. In unserer Webapplikation werden diese zwei Zellen farbig differenziert. Programmatikalisch handelt es sich um Zellen, die Wände sind und in Ziel- resp. Endpünkte umgewandelt werden. Die folgenden zwei Routinen[6] wählen in einem gegebenen Raster einen Start- und einen Endpunkt aus.

1: **procedure** STARTUNDENDPUNKTGENERATOR(*raster*)
2: *frei* ist eine leere Liste ▷ mögliche Zellen für den Start- und Endpunkt
3: **for** $y \leftarrow 0$ **to count**(Zeilen von *raster*) **do**
4: **for** $x \leftarrow 0$ **to count**(Zellen von *raster* in der Zeile y) **do**
5: *zelle* ← Zelle mit (x,y) in *raster*
6: **if** *zelle* ist keine Wand **then**
7: *zelle* der Liste *frei* hinzufügen
8: *raster* ← PUNKTEMARKIEREN(*raster*, *frei*)
9: **return** *raster*
10: **procedure** PUNKTEMARKIEREN(*raster*, *frei*) ▷ Start- und Endpunkte markieren
11: $startIndex \leftarrow \lfloor random(0, \textbf{count}(frei)) \rfloor$ ▷ Zufallsindex für die Liste *frei*
12: **Let**
13: $sx \leftarrow x$-Koordinate von *frei* am Index *startIndex*
14: $sy \leftarrow y$-Koordinate von *frei* am Index *startIndex*
15: **In** Zelle in *raster* bei *sx* und *sy* wird zum Startpunkt

[6]Die genaue Codestelle in der Webapplikation ist hier ersichtlich: `https://github.com/fuerbringer/bma/blob/master/src/maze.js#L27`

16: Element in $frei$ am Index $startIndex$ entfernen
17: $endIndex \leftarrow \lfloor random(0, \textbf{count}(frei)) \rfloor$ ▷ Zufallsindex für die Liste $frei$
18: **Let**
19: $ex \leftarrow x$-Koordinate von $frei$ am Index $endIndex$
20: $ey \leftarrow y$-Koordinate von $frei$ am Index $endIndex$
21: **In** Zelle in $raster$ bei ex und ey wird zum Endpunkt
22: Element in $frei$ am Index $endIndex$ entfernen
23: **return** $raster$

Wendet man nun den Start- und Endpunktgenerator[7] an Rastern mit Wänden (siehe Abbildung 3.6) an, so erhält man folgendes, ein für den PathFinder brauchbares, Raster mit Wänden und Start- und Endpunkt:

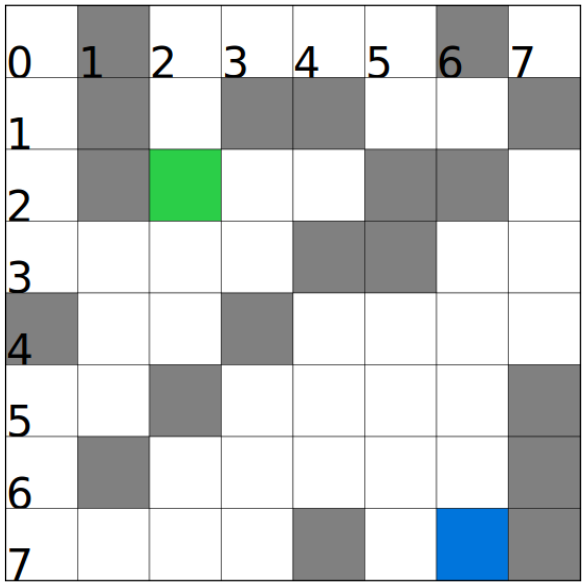


Abbildung 3.7: Ein von unserer Webapplikation generiertes 8×8 Raster mit Wänden und markierten Start- und Endpunkten. Quelle: Eigenleistung

Im letzten Schritt lässt man die PathFinder den Weg zwischen den markierten Start- und Endpunkten finden. Die Parameter, die man PathFinding.js dafür übergeben muss, sind grundsätzlich das Raster und die darin markierten Punkte, die als Start- und Ende dienen. Die markierten Punkte speichert man entweder im Start- und Endpunktgenerator zwischen oder man benutzt eine weitere Routine[8], die die Start- und Endpunkte aus einem Raster extrahiert:

1: **procedure** STARTUNDENDPUNKTFINDEN($raster$)
2: $startPunkt \leftarrow$ leeres Element
3: $endPunkt \leftarrow$ leeres Element

[7]Die genaue Codestelle in der Webapplikation ist hier ersichtlich: `https://github.com/fuerbringer/bma/blob/master/src/maze.js#L27`

[8]Die genaue Codestelle in der Webapplikation ist hier ersichtlich: `https://github.com/fuerbringer/bma/blob/master/src/helper.js#L15`

4: **for all** *zeile* ∈ *raster* **do** ▷ *y*-Achse
5: **for all** *zelle* ∈ *zeile* **do** ▷ *x*-Achse
6: **if** Ist *zelle* ein Startpunkt **then**
7: *startPunk* ← *zelle* ▷ *startPunkt* entspricht der aktuellen Zelle
8: **else if** Ist *zelle* ein Endpunkt **then**
9: *endPunk* ← *zelle* ▷ *endPunkt* entspricht der aktuellen Zelle
10: **return** *startPunkt* und *endPunkt*

Übergibt man PathFinding.js nun die nötigen Parameter des Rasters und dessen Start-und Endpunkte, so erhält man den Weg von Start bis Ende. Wiederholt man dies mehrere Male mit verschiedenen Pathfindern, so ermöglicht man den direkten Vergleich der Pathfinding-Algorithmen. Dies ist eine wichtige Kernfunktion dieser Berufsmaturitätsarbeit.

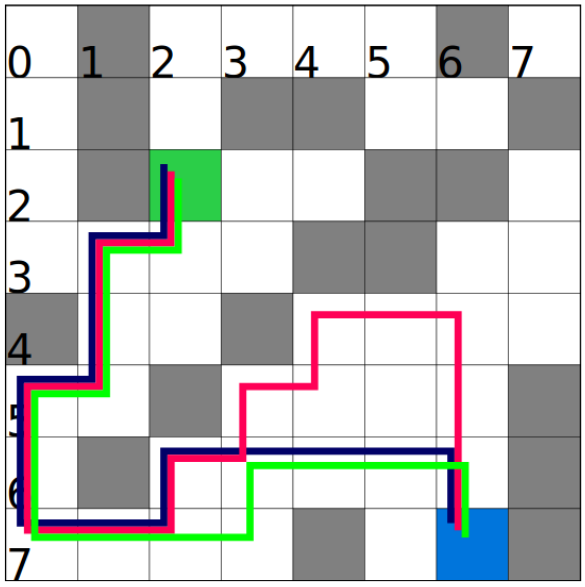


Abbildung 3.8: Ein von unserer Webapplikation generiertes 8×8 Raster mit Wänden und gelöstem Weg. Quelle: Eigenleistung

3.2.2 Auswertung und Merkmale

Zur Auswertung werden Merkmale definiert, welche bei der Statistik relevant sind. Es folgen die drei wichtigsten Merkmale beim Vergleich der Pathfinder in der Webapplikation.

Zurückgelegter Weg

Der zurückgelegte Weg ist aus praktischer Sicht gesehen ein bedeutendes Merkmal, da man natürlich möchte, dass der resultierende Weg möglichst kurz ausfällt. Die Programmbibliothek PathFinding.js liefert den Weg in einer Liste mit einer Koordinate pro Element. Daraus folgt, dass der Weg s der Anzahl der Elemente der Liste entsprechen muss, wenn man nach Path-

Finding.js Sprünge oder Luftlinien ausschliest[9]. Es gibt demnach keine Geraden, ausser zwischen zwei Benachbarten Punkten. Diagonalen würden nach dieser Logik bei einem diagonalen Sprung als eine Wegeinheit gelten.

$$s = n \begin{cases} P_1(0,0), \\ P_2(1,0), \\ ... \\ P_n(x,y) \end{cases}$$

Operationen

Ein Algorithmus, der für das gleiche Resultat weniger Schritte (Operationen) tätigen muss, ist grundsätzlich besser als ein anderer. Zur Erfassung der Anzahl Operationen mussten die internen Routinen der Programmbibliothek PathFinding.js angepasst werden, da sie solche Messungsmerkmale nicht standardmässig liefert. Dazu wurde der Quellcode von PathFinding.js kopiert[10] und in die Routinen der drei ausgewählten Pathfinder im äusseren und inneren Loop eingefügt. Folgende Dateien[11] wurden dafür angepasst:

- src/finders/AStarFinder.js
- src/finders/BreadthFirstFinder.js

Zu beachten ist, dass die Datei des BestFirstFinders nicht angepasst werden musste, da dieser auf dem A*-Finder basiert und somit alle Änderungen vom A*-Finder vererbt.

Rechenzeit

Ein bedeutendes Merkmal ist die Rechenzeit, also die Zeit, in der der Pathfinder seine Arbeit verrichtet hat. Um diese zu berechnen, wird unmittelbar vor der Ausführung der Wegfindungsroutine des Pathfinders der Zeitstempel t_0 der jetzigen Zeit zwischengespeichert und danach vom Pathfinder die Arbeit verrichtet. Direkt nach dem Beenden der Wegfindung wird ein zweites Mal ein Zeitstempel t_1 zwischengespeichert. Die Differenz ergibt die Zeit t in Millisekunden[12], die für die Berechnung benötigt wurde.

$$t = t_1 - t_0$$

[9] Ein von PathFinding.js berechneter Weg: `https://pathfindingjs.readthedocs.io/en/latest/user-guide/getting-started/`

[10] Diesen Prozess nennt man bei freier Software “Forking”.

[11] Die genauen Anpassungen sind unter dem folgenden Link verfügbar: `https://github.com/fuerbringer/PathFinding.js/commit/51034158638ad7aaa9c5142a827bd4d3de2b8786#diff-22360b18a43ac33ae398e44b998101c7`

[12] Die Einheit Millisekunden ergibt sich aus der Schnittstelle `Performance.now()` von JavaScript

Die Werte t sind pro Pathfinder in der Webapplikation unter “Messung für diesen Einzelversuch:” unter dem Raster ersichtlich. Diese Berechnung demnach pro Durchlauf dreimal getätigt.

Aus diesen drei Merkmalen folgen zusammengefasst die in unserer Webapplikation vermessenen Werte der Pathfinding-Algorithmen.

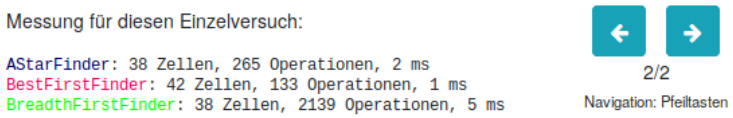


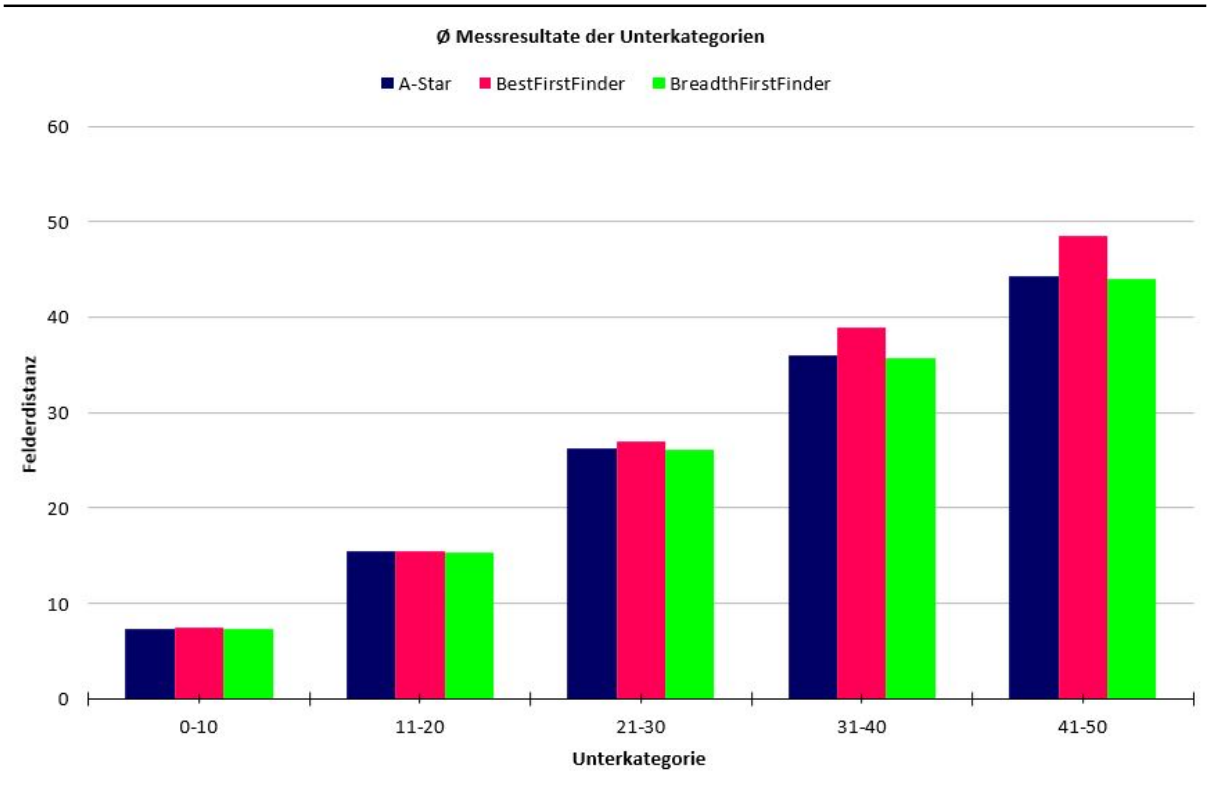
Abbildung 3.9: Vergleicherausschnitt. Quelle: Eigenleistung

4. Auswertung

4.1 Statistiken und Vergleiche der Pathfinding-Algorithmen

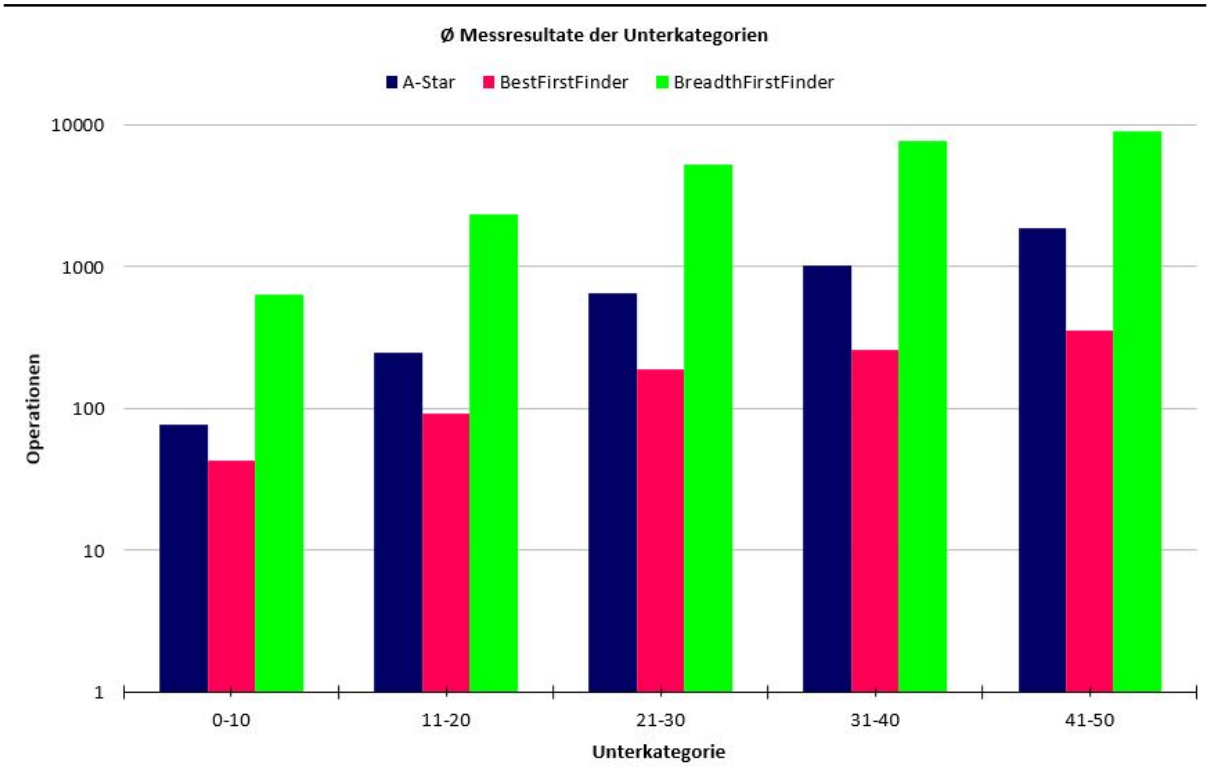
Alle notierten Messresultate pro Vergleich befinden sich im Anhang dieser Arbeit.

Das Autorenteam hat sich auf eine Rastergrösse von 50×50 Felder entschieden und durchlief die Vergleiche 125-mal. Die Rechenzeiten wurden auf 0.1 ms gerundet. Die Diagonalen wurden mitbeachtet, da dies zu optimierten Strecken führen kann.
Die 125 Versuche wurden zuerst in folgende Kategorien in den Resultate unterteilt: Weg des kürzesten Pathfinders von 0-10 Felder, 11-20 Felder, 21-30 Felder, 31-40 Felder und 41-50 Felder. Jede dieser Unterteilung hat 25 Vergleichsergebnisse. Diese Unterteilung hat den Grund, dass auf längeren Wege die Unterschiede nach der Erfahrung dieser Arbeit durch das Autorenteam deutlicher zu sehen sind.

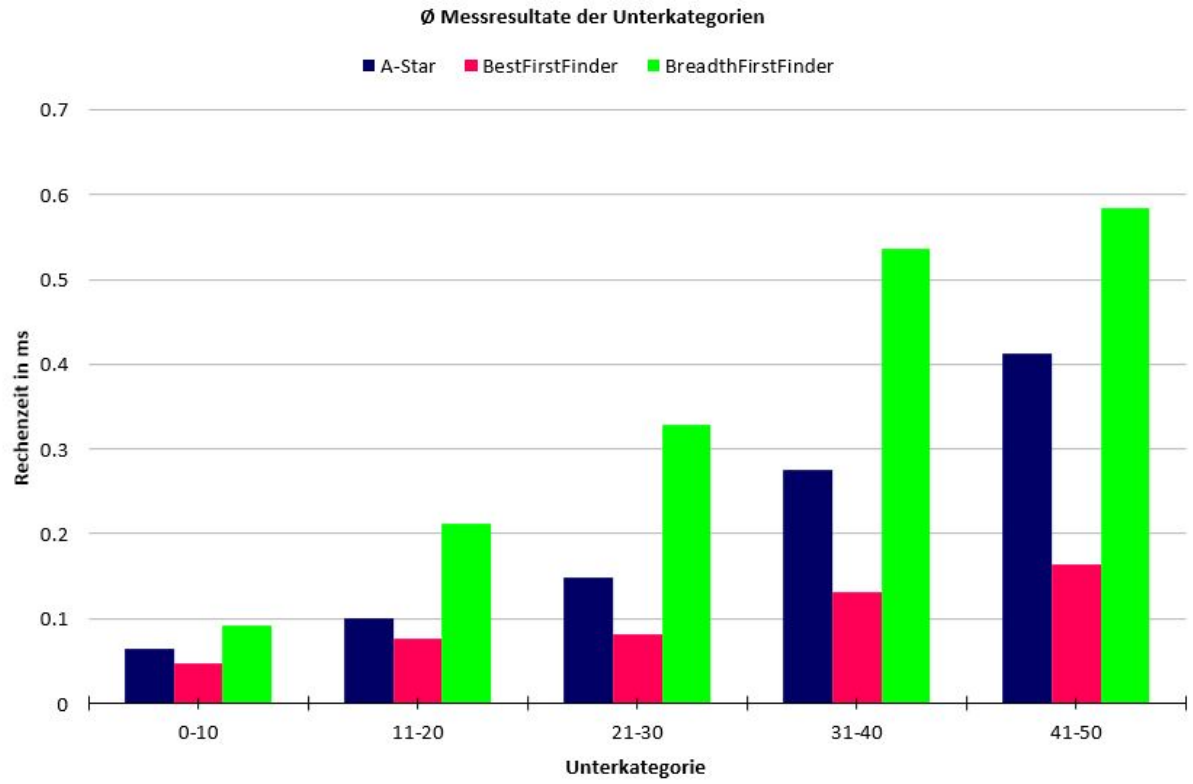


Man erkennt gut, dass der A-Star und BreadthFirstFinder circa die gleiche Felderdistanz finden.

Je grösser der Abstand von Start und Ziel, desto grösser ist der Unterschied zwischen dem BestFirtFinder und den anderen zwei.



Hierbei muss beachtet werden, dass die Diagrammdarstellung logarithmisch ist. Im Aufwand der ausgeführten Operationen steigt der BreadthFirstFinder viel extremer als die anderen. Der A-Star muss im Verhältnis zum BestFirstFinder immer mehr Operationen ausführen. Die Steigung des BreadthFirstFinder senkt sich im Laufe der Felderdistanz. Dies kommt daher, dass er am Rand des Rasters ist und diese Seite nicht mehr weiter erkunden muss. Die Vorteile einer Heuristik wird hier klar verdeutlicht. Der A-Star und BestFirstFinder müssen durch die Wahrscheinlichkeitsberechnung weniger Operationen durchführen.



Da der BestFirstFinder den erstbesten Weg tiefer durchsucht, ist seine Rechenzeit um einiges kürzer als die anderen. Da der BreadthFirstFinder am meisten Operationen hat, folgt, dass seine Rechenzeit höher als die vom A-Star oder BestFirstFinder ist.

4.2 Interpretation Resultate

Aus den Statistiken kann man folgendes herauslesen:

- Der A-Star und BreadthFirstFinder finden ungefähr den gleich optimalen Weg, wobei der BreadthFirstFinder viel mehr an Operationen aufwenden muss, da er über keine Heuristik verfügt.
- Hat man begrenzte Zeit, so ist der BestFirstFinder gut geeignet, er findet nicht den optimalsten Weg, spart dafür in Zeit und Operationen.
- Je grösser das Raster, desto weniger ist der BreadthFirstFinder geeignet, da er zu viele Operationen benötigt und im Vergleich zum A-Star langsam wird.
- Die Rechenzeit vom BestFirstFinder steigt linear und die vom A-Star und BreadthFirstFinder exponential, solange der BreadthFirstFinder keine Rasterseite erreicht hat.

4.3 Zusammenfassung

Es wurde beschrieben und erklärt, was ein Algorithmus und Pathfinding-Algorithmus ist. Es ist geklärt, wozu man Graphen braucht und wie diese in Pathfinding-Algorithmen einfliessen. Durch ein technisches Produkt werden die Unterschiede des A-Star, BestFirstFinde und BreadthFirstFidner in Vergleichen verdeutlicht. Die Realisierung der Webapplikation ist klar beschrieben und dessen Messresultate in statistischen Diagrammen verglichen und erläutert. Durch die Bearbeitung der Fachliteratur kam heraus, dass die benutzten Pathfinding-Algorithmen nicht aus einem Uralgorithmus stammen. Jeder der Algorithmen basiert auf einer Idee, wie das Vorgehen zum Lösen der Aufgabe bearbeitet werden soll. Der BreadthFirstFidner sucht ausschliesslich in der Breite der Graphen, das heisst er schaut sich alle ihm am nächstliegenden Graphen zuerst an. Der BestFirstFindes sucht in der Tiefe nach der Lösung und vernachlässigt potentiell bessere Lösungen, die am Anfang nicht besser aussehen. Der A-Star geht mit intelligenten Listen vor, mit denen er jeden Zyklus dem wahrscheinlich besten Graph als nächstes untersucht. Durch das erweiterte Produkt mit Vergleichsmöglichkeiten des selben Rasters konnten diese Schlüsse durch die Statistiken belegt werden. Interpretiert man die Resultate mit dem vorgegebenen Oberthema, erkennt man die daraus folgenden Erkenntnisse, wie die einzelnen Pathfinding-Algorithmen in ihren Merkmalen funktionieren und wie sie am besten anzuwenden sind. Ein Problem ist, dass diese Algorithmen in einem zweidimensionalen Raum funktionieren, unsere Welt jedoch dreidimensional ist. Hätte man komplexere dreidimensionale Pathfinding-Algorithmen untersucht, wäre der Aufwand zur Erkenntnis bestimmt höher und hätte mehr Einflüsse im Raum. Die Ortsbestimmung über ein GPS ist schwieriger als das berechnen einer Schachfigur auf einem Spielfeld durch Pathfinding-Algorithmen. Mit der Erkenntnis dieser Arbeit kann beispielsweise ein Spieleentwickler besser bestimmen, welchen Pathfinding-Algorithmus er für sein zweidimensionales Spiel wählen soll und warum.

5. Anhang

5.1 Screenshots der Webapplikation

5.1.1 Pathfinder-Vergleicher

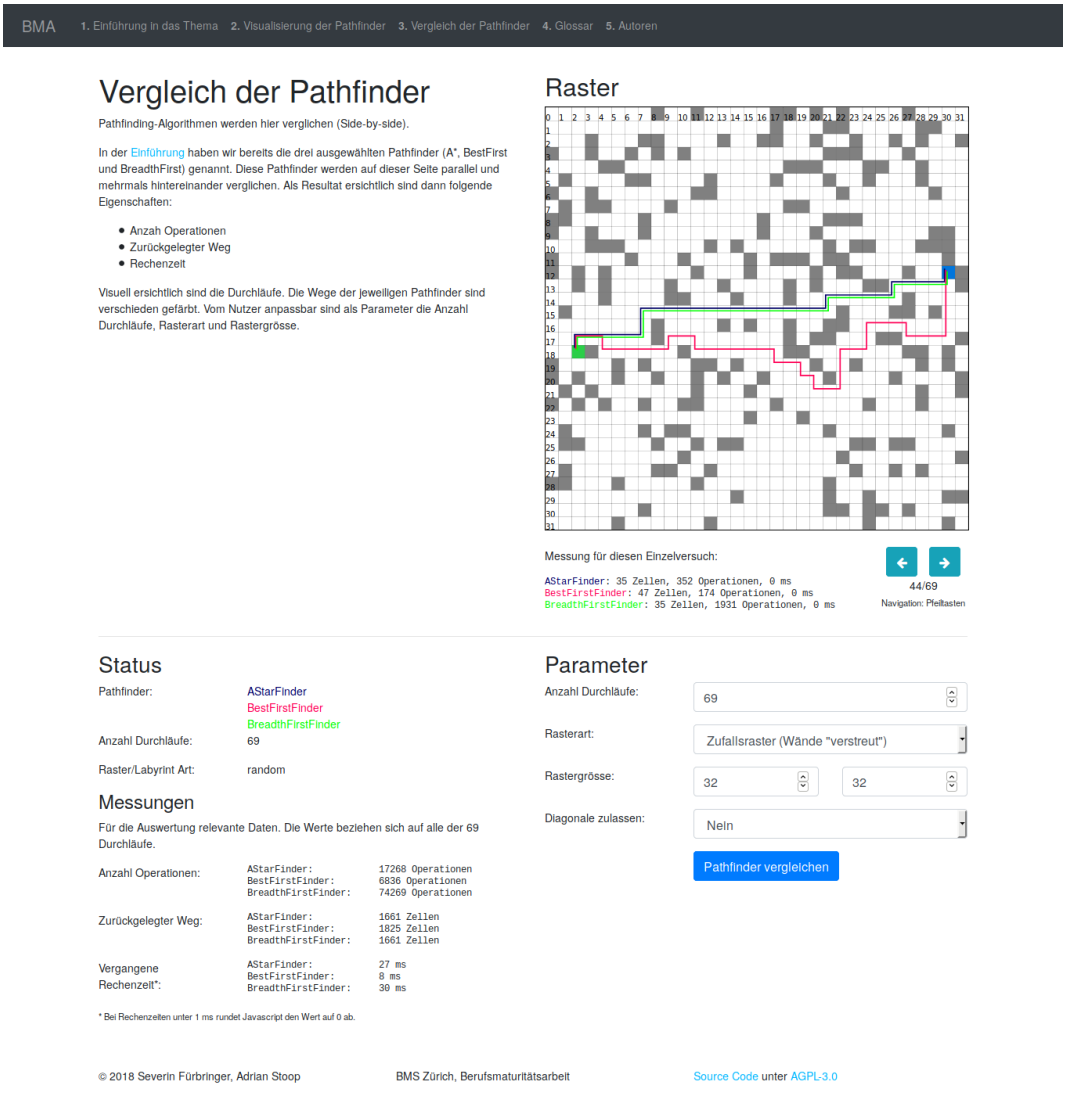


Abbildung 5.1: Pathfinder-Vergleicher-Page. Quelle: Eigenleistung

5.2 Statistiken

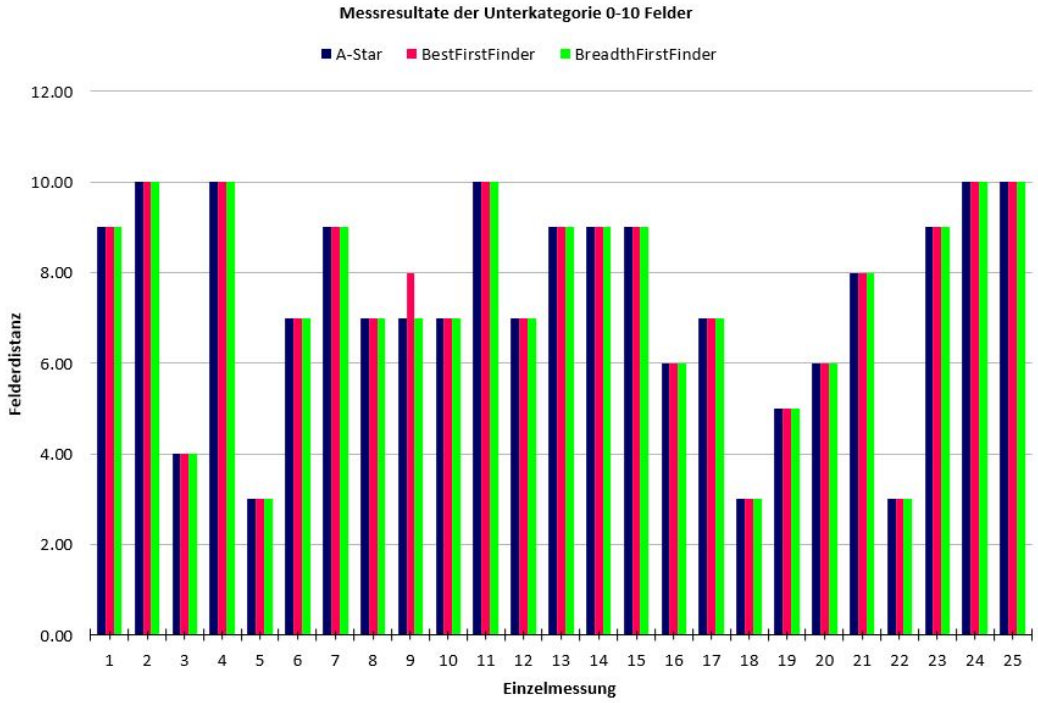


Abbildung 5.2: Messresultate der Unterkategorie 0-10 Felder für Felderdistanz. Quelle: Eigenleistung

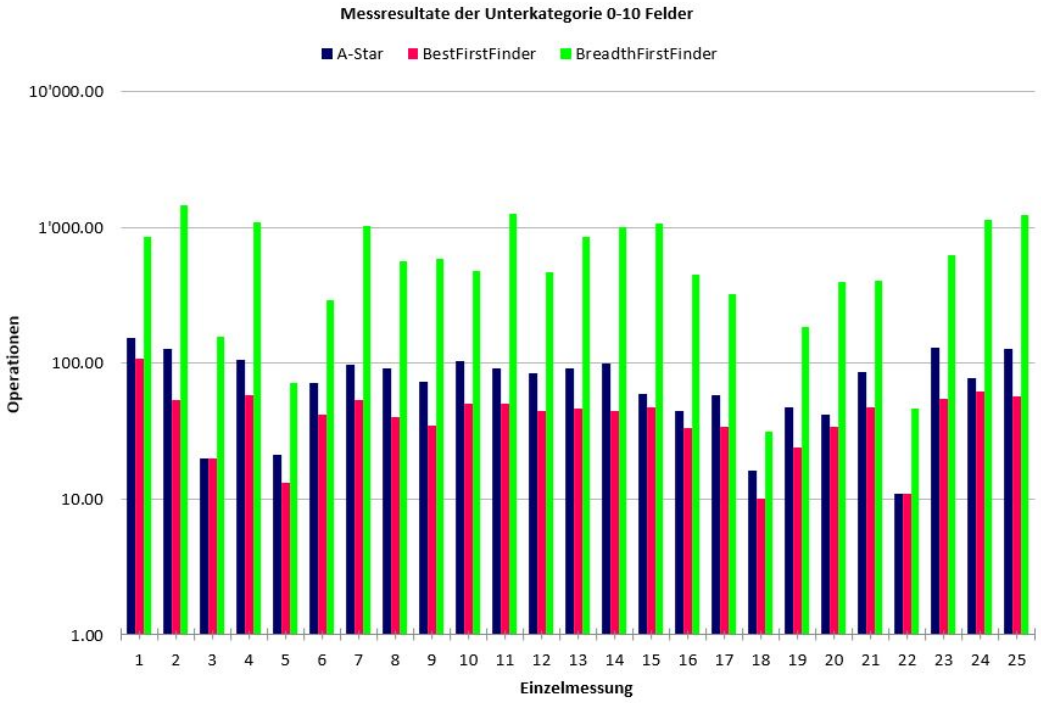


Abbildung 5.3: Messresultate der Unterkategorie 0-10 Felder für Operationen. Quelle: Eigenleistung

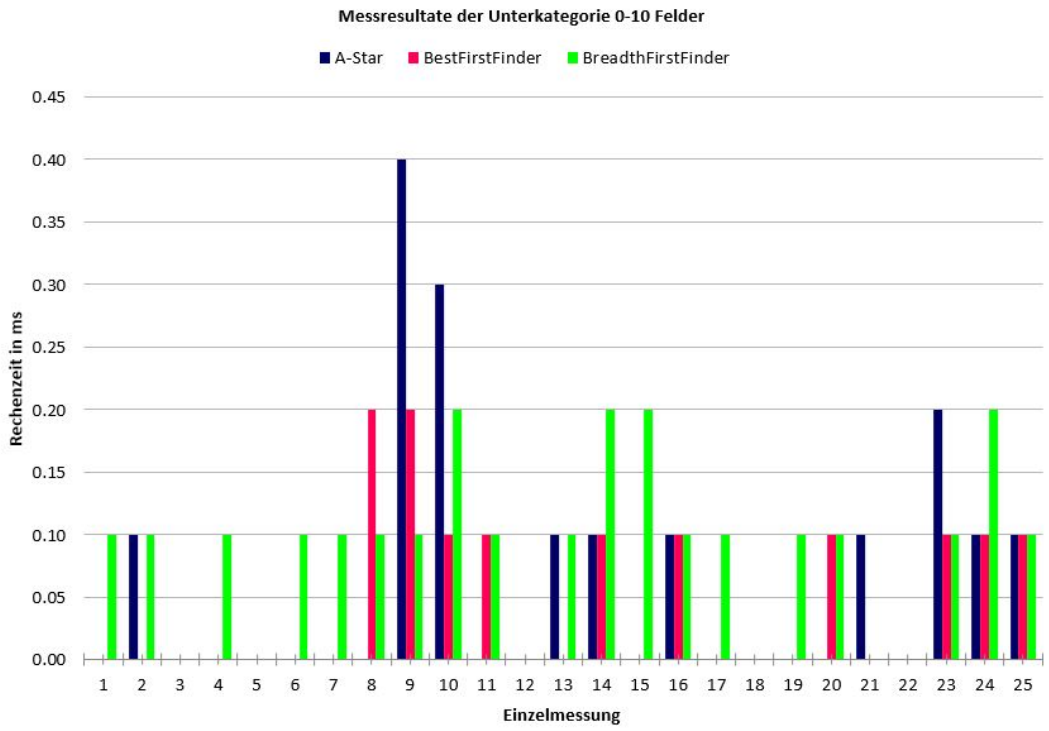


Abbildung 5.4: Messresultate der Unterkategorie 0-10 Felder für Rechenzeit in ms. Quelle: Eigenleistung

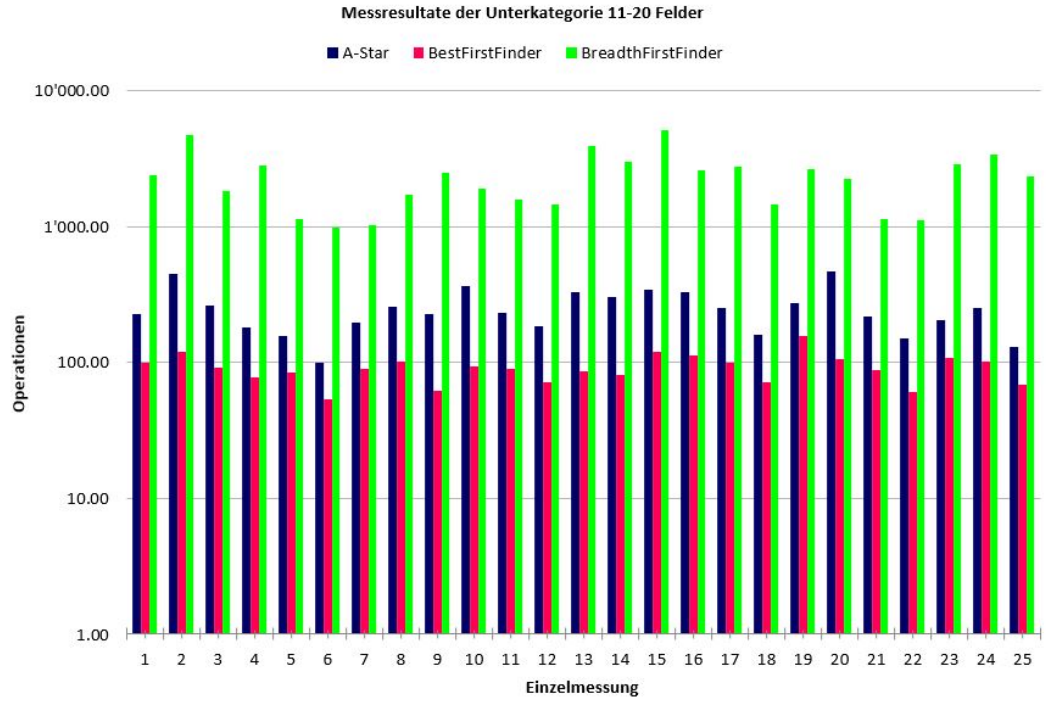


Abbildung 5.5: Messresultate der Unterkategorie 11-20 Felder für Operationen. Quelle: Eigenleistung

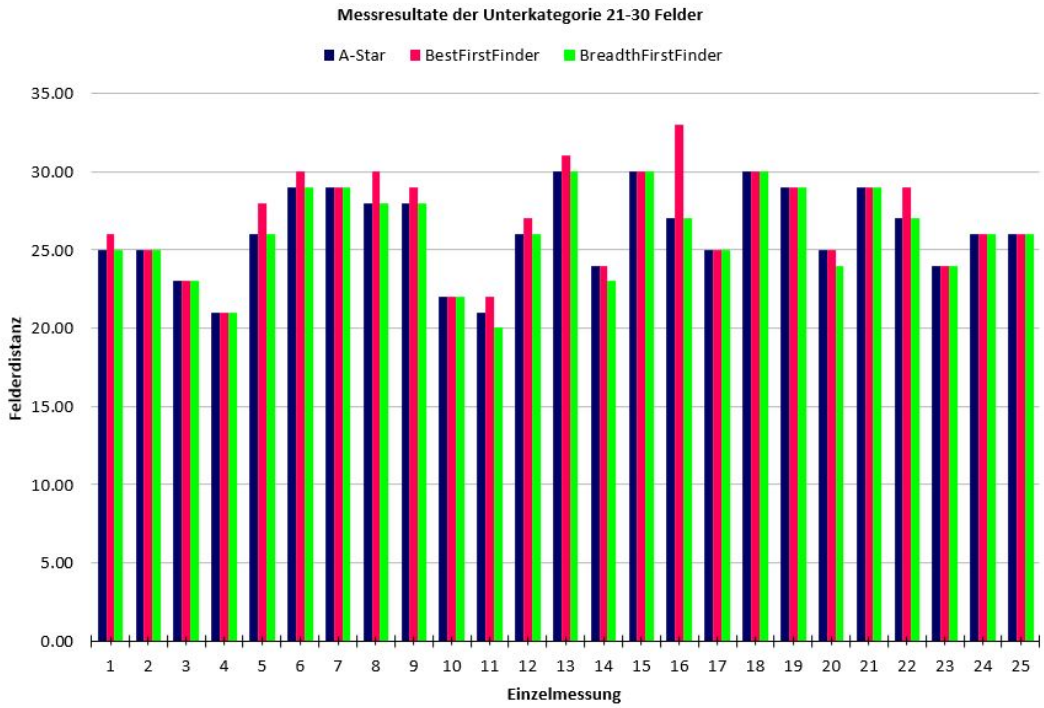


Abbildung 5.6: Messresultate der Unterkategorie 21-30 Felder für Felderdistanz. Quelle: Eigenleistung

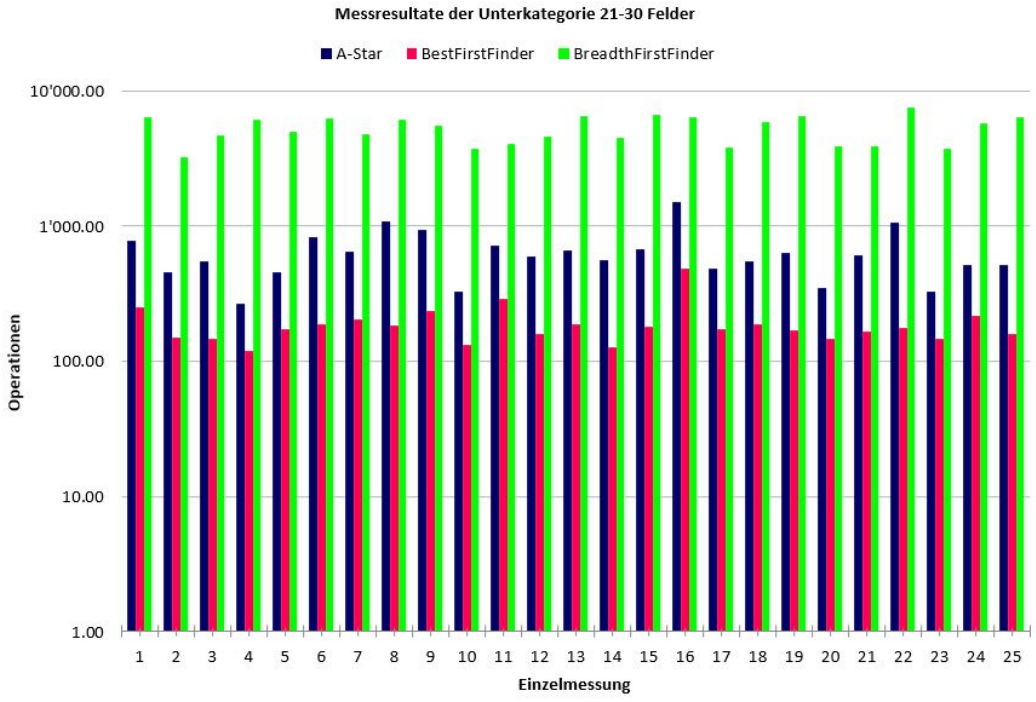


Abbildung 5.7: Messresultate der Unterkategorie 21-30 Felder für Operationen. Quelle: Eigenleistung

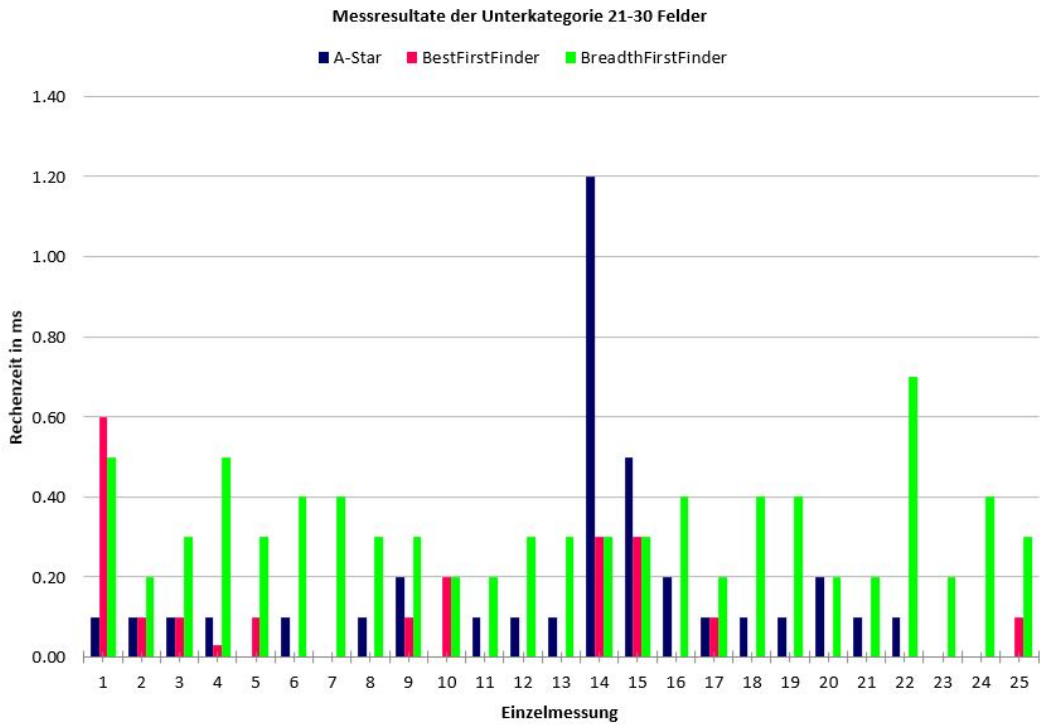


Abbildung 5.8: Messresultate der Unterkategorie 21-30 Felder für Rechenzeit in ms. Quelle: Eigenleistung

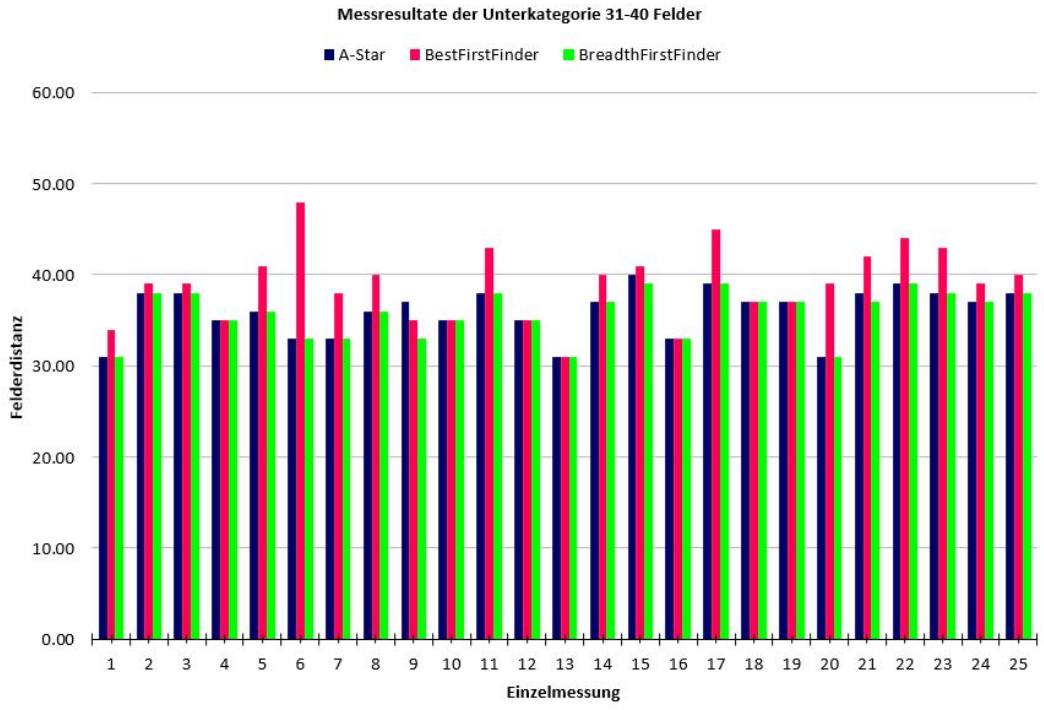


Abbildung 5.9: Messresultate der Unterkategorie 31-40 Felder für Felderdistanz. Quelle: Eigenleistung

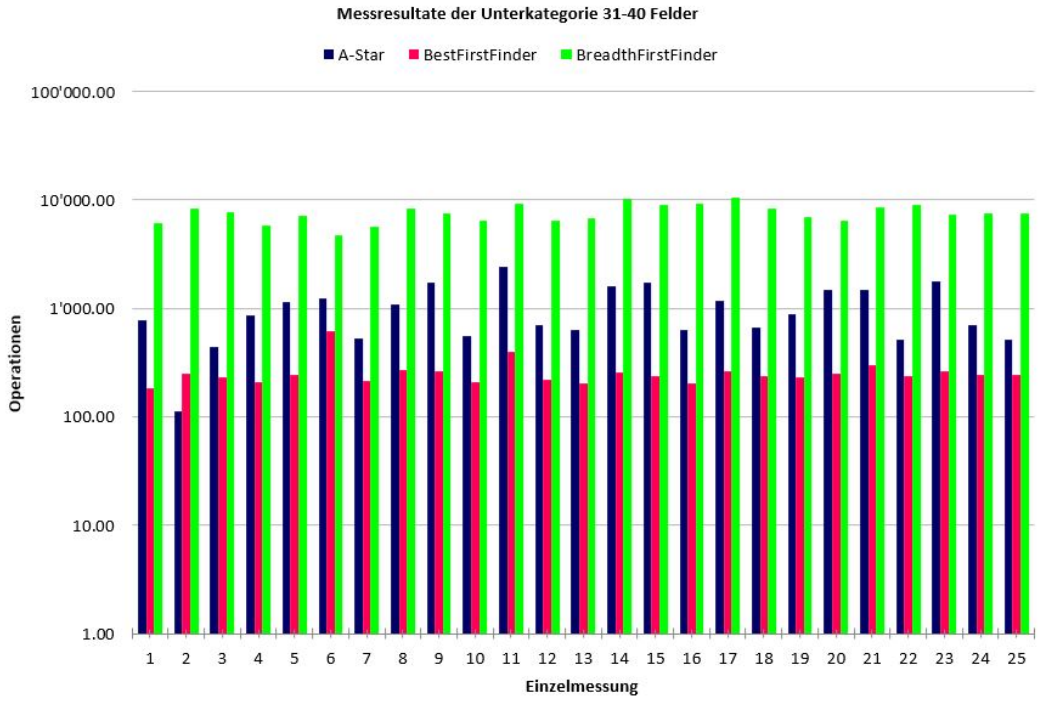


Abbildung 5.10: Messresultate der Unterkategorie 31-40 Felder für Operationen. Quelle: Eigenleistung

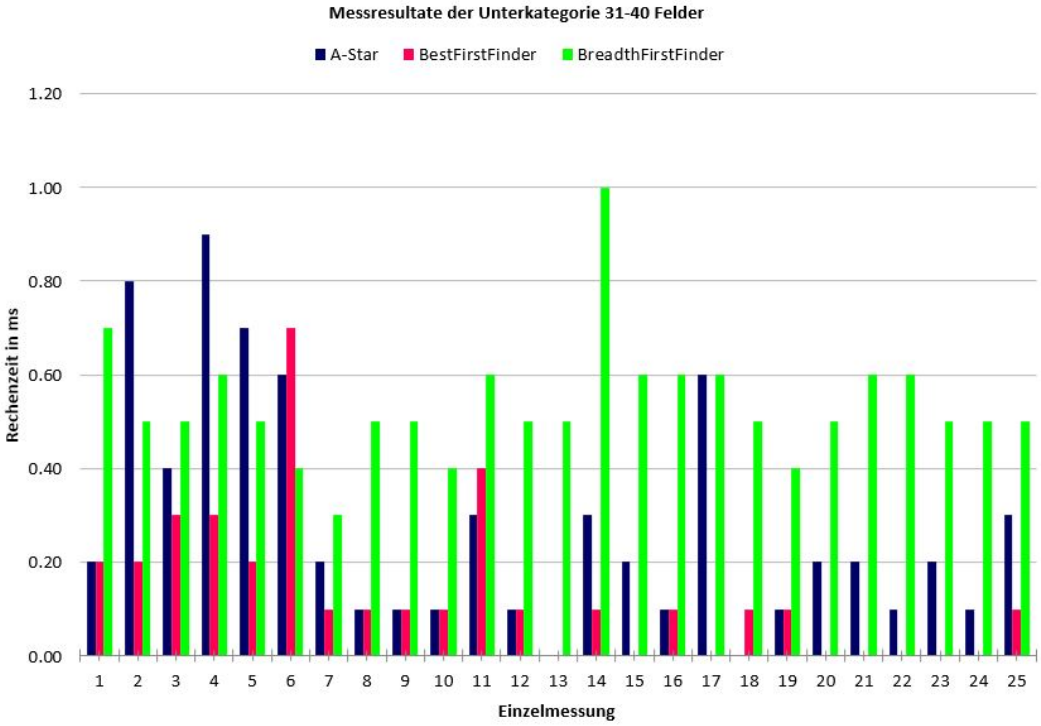


Abbildung 5.11: Messresultate der Unterkategorie 31-40 Felder für Rechenzeit in ms. Quelle: Eigenleistung

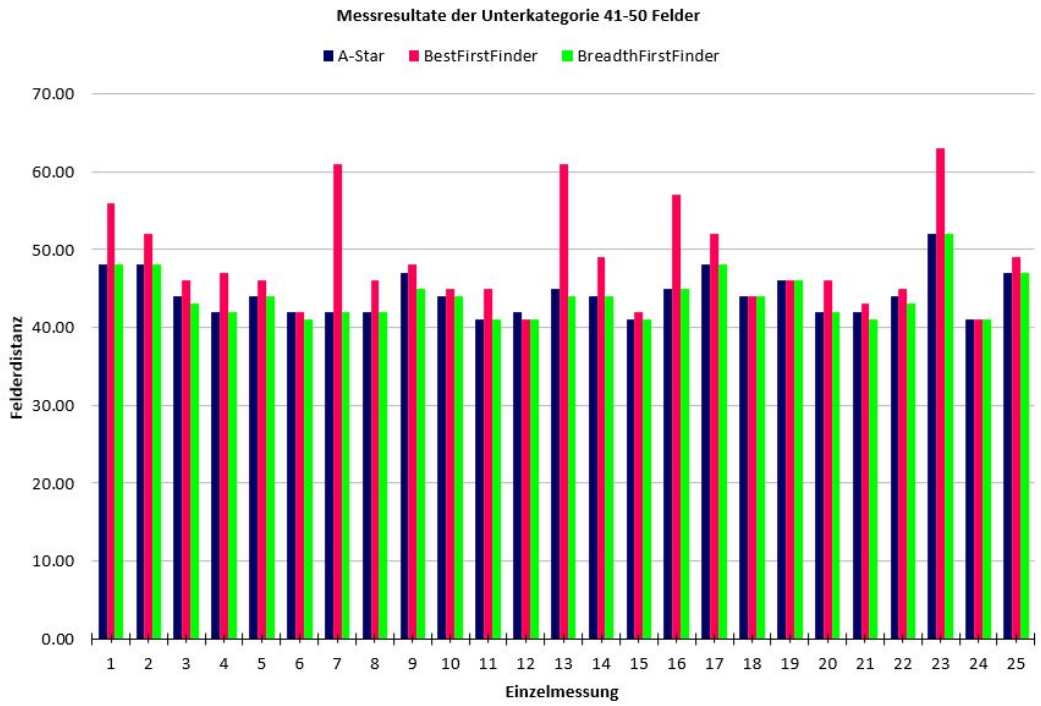


Abbildung 5.12: Messresultate der Unterkategorie 41-50 Felder für Felderdistanz. Quelle: Eigenleistung

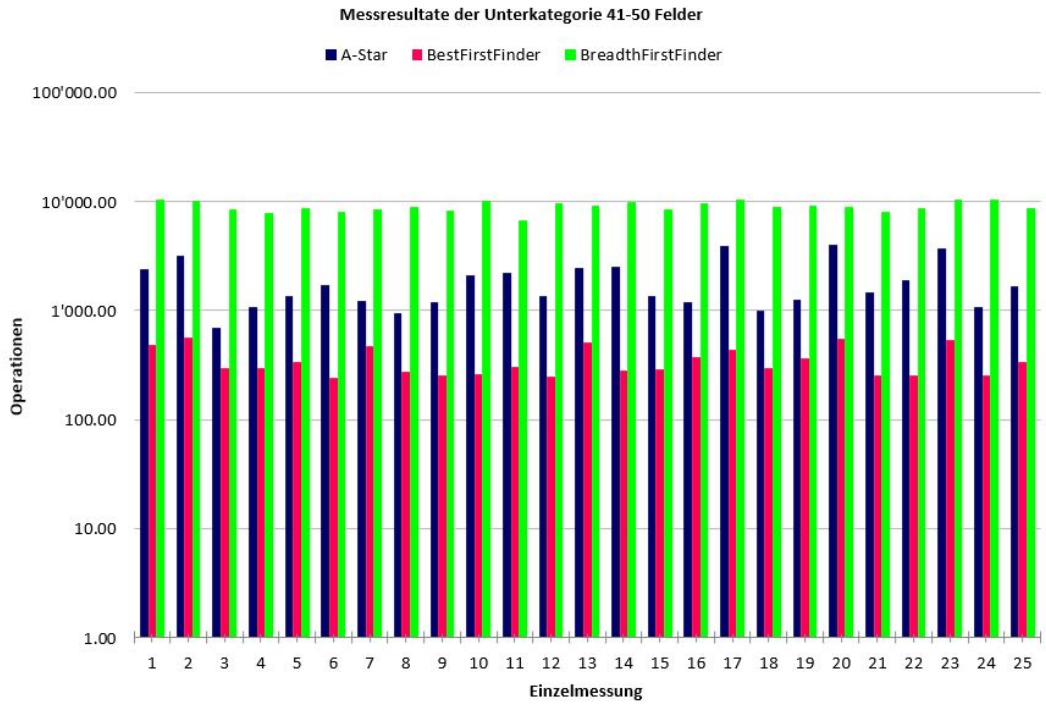


Abbildung 5.13: Messresultate der Unterkategorie 41-50 Felder für Operationen. Quelle: Eigenleistung

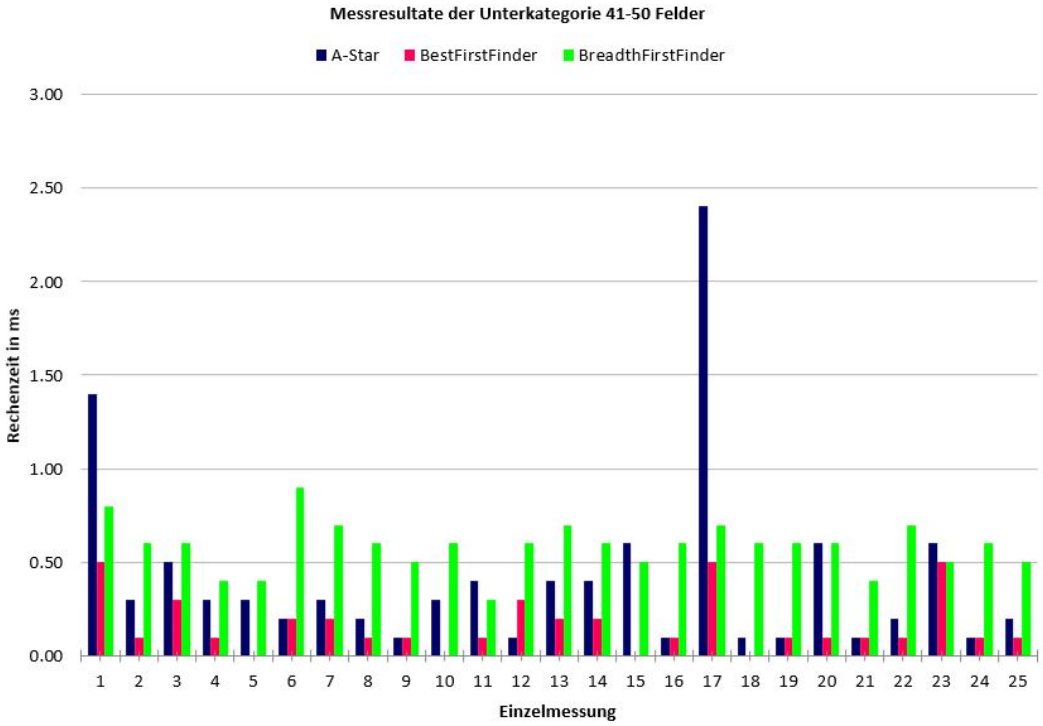


Abbildung 5.14: Messresultate der Unterkategorie 41-50 Felder für Rechenzeit in ms. Quelle: Eigenleistung

5.3 Danksagung

5.3.1 Personen

5.3.2 Schriftsetzung

Die Schriftsetzung wurde durch die freie Software LaTeX und deren Autoren Leslie Lamport et. al. ermöglicht.

6. Glossar

Fachwörter- und Begriffsverzeichnis

In diesem Kapitel werden wichtige Wörter aus der Fachsprache erklärt, die in unserer Arbeit oftmals vorkommen.

A* Sprich "A Star". Ein Pathfinding Algorithmus (siehe Fachbegriff Algorithmus). Dieser nutzt zusätzlich heuristische Mittel, um den Weg zu berechnen (siehe Heuristik).

Algorithmus Ein Ablauf, Prozess oder Programm, der eine Liste mit Anweisungen schrittweise befolgt, um Daten umzuwandeln.

Back-End Der Server—mit guten Systemtechnikern rund um die Uhr stellt das Back-End die Website unter einer URL, wie `bma.fuerbringer.info`, zur Verfügung.

BestFirstFinder Ein Pathfinding-Algorithmus. Nicht zu verwechseln mit dem BreadthFirstFinder.

BreadthFirstFinder Ein Pathfinding-Algorithmus. Nicht zu verwechseln mit dem BestFirstFinder.

Dijkstra Ein Pathfinding Algorithmus (siehe Fachbegriff Algorithmus). Dieser nutzt keine heuristische Mittel und kommt in unserer Arbeit nicht direkt vor.

Front-End Der Browser—der Teil einer Website oder Webapplikation, den der Benutzer sieht und mit dem er interagiert. In unserem Fall werden die wichtigsten Berechnungen im Front-End verrichtet.

Heuristik Hilfsalgorithmus, unter anderem für Pathfinder, der zusätzlich hilft zwischen einzelnen Schritten die Kosten für eine mögliche Operation einzuschätzen.

Matrix Anordnung von Elementen. In unserem Fall ein zweidimensionales Raster, in dem Wände, Korridore und Wege platziert sind.

Operationen Eine "Operation" beschreibt in unserem Fall einen Zyklus eines Pathfinders. Je mehr Zyklen ein Pathfinder in Relation zu einem anderen benötigt, desto weniger effizient ist er.

Parameter Die Randbedingungen für einen Durchlauf bzw. ein Experiment.

PathFinding.js Die Zentrale Programmbibliothek, deren Pathfinder Implementationen wir in unserer Webapplikation nutzen.

Pathfinder Eine Art Algorithmus, der in einem gegebenen Raum mit eingezeichnetem Start- und Endpunkt den schnellsten weg Findet.

recbacktracker "Recursive Backtracker, ein Labyrinthalgorithmus, den wir in unserer Webapplikation zur Generierung von perfekten (immer lösbaren) Labyrinthen verwenden. Dieser Algorithmus kommt im Vergleicher nicht zum Zug.

Pseudocode Definiert einen Algorithmus schriftlich in einer allgemein verständlichen Sprache, damit dieser in beliebigen Programmiersprachen umgesetzt werden kann.

PHP Eine freie serverseitige Programmiersprache, die seit über zwei Jahrzehnten einen hohen Marktanteil hat.

JavaScript Nicht zu verwechseln mit Java. Eine clientseitige Programmiersprache, die in jedem Browser integriert ist. Sie wird hauptsächlich zwecks der Interaktivität benutzt.

Implementieren/Implementation Im Kontext der Software wird damit die Planung und Entwicklung des Quellcodes bezeichnet.

Pixelmatrizen Eine Liste mit Pixeldaten, die zum Beispiel bei Computergrafikapplikationen gebraucht wird. Der "Inhalt" des Bildschirm kann ganzheitlich in einer Pixelmatrix abgelegt werden.

Prozedural Beschreibt programme, die schrittartig ablaufen.

Routine Unter diesem Begriff kann man im Rahmen dieser Arbeit einen Teil eines Algorithmus, einer Funktion oder einer Prozedur verstehen.

Unix und GNU/Linux Betriebssysteme abstammend vom AT&T UNIX.

Literaturverzeichnis

[1] Andrew Walker; StackOverflow, *Is A* the best pathfinding algorithm?*, https://stackoverflow.com/a/9515734, [Abrufdatum: 28.01.2019]

[2] Wikipedia, *Algorithmen*, https://de.wikipedia.org/wiki/Algorithmus, [Abrufdatum: 28.01.2019]

[3] Wikipedia, *Pathfinding*, https://de.wikipedia.org/wiki/Pathfinding, [Abrufdatum: 28.01.2019]

[4] Andreas Hofmann, Fachhochschule Technikum Wien, *Pathfindingthfinding-Algorithmen in verschiedenen Spielegenres*, http://devblog.viking-studios.net/wp-content/uploads/2013/04/Pathfinding-Algorithmen-in-verschiedenen-Spielegenres.pdf, [Abrufdatum: 28.01.2019]

[5] Franz Embacher, Institut für Theoretische Physik der Universität Wien, *Von Graphen, Genen und dem WWW*, https://homepage.univie.ac.at/franz.embacher/Lehre/aussermathAnw/Graphen.html, [Abrufdatum: 27.01.2019]

[6] Sven Oliver Krumke, Hartmut Noltemeier, Stefan Schwarz, Hans-Christoph Wirth, Bayerischen Julius-Maximilians-Universität Würzburg, *Graphentheoretische Konzepte und Algorithmen*, http://www.matheraetsel.de/texte/graphentheorie.pdf, Seite 76, [Abrufdatum: 27.01.2019]

[7] Anders Strand-Holm Vinther, Magnus Strand-Holm Vinther, Aarhus University - Computer Science, Juni 2015, *Pathfinding in Two-dimensional Worlds*, http://tildeweb.au.dk/au121/advising/thesis/anders-strand-holm-vinther_magnus-strand-holm-vinther.pdf, Seite 22, [Abrufdatum: 28.01.2019]

[8] Amit Patel, *A*'s Use of the Heuristic*, http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#manhattan-distance, Abs. *Manhattan distance*, 2019, [Abrufdatum: 28.01.2019]

[9] Thorsten Schmidt, David Fuchs, Universität Tübingen, *A-Stern Algorithmus*, http://www.geosimulation.de/methoden/a_stern_algorithmus.htm, [Abrufdatum: 26.01.2019]

[10] Roblox Corporation, *Best-First Search*, https://developer.roblox.com/articles/Best-first-search#Pseudo_Code, 5. Juli 2018, [Abrufdatum: 28.01.2019]

[11] Brilliant.org, *Breadth-First Search (BFS)*, https://brilliant.org/wiki/breadth-first-search-bfs/, [Abrufdatum: 28.01.2019]

[12] Xueqiao Xu et al., *PathFinding.js*, https://github.com/qiao/PathFinding.js, Quellcode auf GitHub, Letzter Aufruf: 29. Januar 2019

[13] Cui Xiao; Shi Hao, *A*-based Pathfinding in Modern Computer Games*, IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1, Januar 2011.

[14] Severin Fürbringer; Adrian Stoop, *BMA Quellcode*, https://github.com/fuerbringer/bma, 29. Januar 2019

[15] Severin Fürbringer; Adrian Stoop, *BMA-Webapplikation*, https://bma.fuerbringer.info, 29. Januar 2019

Abbildungsverzeichnis