HOCHSCHULE
ESSLINGEN

Nah an Mensch und Technik.

Fakultät Informatik und Informationstechnik

Studiengang Technische Informatik

Studienarbeit

Bachelor of Engineering

Visualisierung eines echtzeitfähigen Routenplan-Algorithmus

Seid Jadadic

Sommersemester 2023

Erstprüfer: Prof. Dr. Reiner Marchthaler
Zweitprüfer: Prof. Dr. Jürgen Koch

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Esslingen, 5. August 2023
Ort, Datum

Seid Jadadic

Inhaltsverzeichnis

1. Einleitung

In diesem Kapitel wird die Motivation für die Wahl dieses Themas erläutert und die Zielsetzung der Studienarbeit festgelegt. Zudem wird ein Überblick über den Aufbau und den Inhalt der Arbeit gegeben.

1.1. Motivation

Die vorliegende Studienarbeit befasst sich mit der Implementierung und Evaluierung des A*-Routenplanungsalgorithmus, einem leistungsfähigen und vielseitig einsetzbaren Routenplanungsalgorithmus.
Für die Wahl dieses Themas gibt es mehrere Gründe, die im Folgenden erläutert werden.

Effiziente Routenplanung spielt in vielen Anwendungsbereichen eine entscheidende Rolle. Von Navigationssystemen und Logistik bis hin zu Robotik und Videospielen sind viele Systeme auf eine schnelle und zuverlässige Wegfindung angewiesen.
Der A*-Algorithmus hat sich dabei als eines der besten Verfahren erwiesen und wird daher in der Praxis häufig eingesetzt.

Die Motivation hinter der Implementierung des A*-Algorithmus ist es, ein grundlegendes Verständnis für diesen Algorithmus zu entwickeln und die zugrundeliegenden Konzepte zu erforschen. Die Arbeit soll es ermöglichen, die Funktionsweise des A*-Algorithmus praktisch in C++ umzusetzen und dadruch einen tieferen Einblick in die Arbeitsweise von Routing-Algorithmen zu gewinnen.

Ein weiterer Aspekt dieser Arbeit liegt in der Möglichkeit, die Leistungsfähigkeit des A*-Algorithmus unter verschiedenen Bedingungen zu evaluieren. Dabei sollen sowohl einfache auch komplexe Graphen analysiert werden, um den Einfluss von Graphengröße und -struktur auf die Laufzeit des Algorithmus zu untersuchen. Der Vergleich mit anderen Routingalgorithmen erlaubt es außerdem, die Vor- und Nachteile des A*-Algorithmus gegenüber alternativen Ansätzen zu diskutieren.

1.2. Zielsetzung der Studienarbeit

Das Hauptziel dieser Arbeit ist es, den A*-Algorithmus in C++ zu implementieren und seine Funktionsweise zu verstehen. Dabei werden die verschiedenen Komponenten des Algorithmus untersucht, wie z.B. die Heuristik, sowie die Datenstrukturen.

Ein weiterer Aspekt ist die Bewertung der Leistungsfähigkeit des A*-Algorithmus. Es werden verschiedene Tests durchgeführt, um die Laufzeit des Algorithmus auf verschiedenen Graphen zu messen und zu vergleichen.

Die Erkenntnisse aus dieser Studienarbeit sollen dazu dienen, die Funktionsweise des A*-Algorithmus besser zu verstehen und seine Leistungsfähigkeit in verschiedenen Anwendungsfällen zu bewerten. Die Ergebnisse können wertvolle Informationen liefern, um den A*-Algorithmus in verschiedenen Bereichen effektiv einzusetzen und mögliche Optimierungsmöglichkeiten aufzuzeigen.

1.3. Aufbau der Studienarbeit

Die vorliegende Studienarbeit befasst sich mit der Implementierung eines A*-Routenplan Algorithmus und untersucht dessen Leistungsfähigkeit in verschiedenen Szenarien.

In der Einleitung wird die Motivation für die Themenwahl erläutert und die Zielsetzung der Studienarbeit definiert. Hier wird ein Überblick über die Problemstellung gegeben und die Vorgehensweise in der Arbeit beschrieben.

Im Abschnitt „Grundlagen “ werden die grundlegenden Konzepte und Begriffe erläutert, die für das Verständnis des A*-Algorithmus und seiner Implementierung relevant sind.

Der Abschnitt „Datenstrukturen und Implementierungsdetails“ befasst sich mit den technischen Aspekten der Implementierung. Hier wird erläutert, wie eine GraphML-Datei eingelesen wird und wie der A*-Algorithmus konkret implementiert wurde.
Es werden relevanten Datenstrukturen und Funktionen erläutert, die für die Implementierung des Algorithmus benötigt werden.

Die „Evaluierung und Performance-Analyse“ bilden den vierten Abschnitt der Arbeit. Hier werden verschiedene Testfälle vorgestellt, die das Einlesen der GraphML-Datei und die Implementierung des Algorithmus überprüfen. Außerdem wird ein Vergleich der Laufzeiten des A*-Algorithmus auf verschiedenen Graphen durchgeführt, um die Performance in verschiedenen Szenarien zu bewerten.

Im abschließenden Abschnitt werden die Ergebnisse und Erkenntnisse der Studienarbeit zusammengefasst.

Der Anhang der Studienarbeit enthält die Quellcodes der implementierten Funktionen sowie die Unit-Tests zur Überprüfung der Implementierung.

2. Grundlagen

Die grundlegenden Konzepte und Techniken des A*-Algorithmus bilden das Fundament für das Verständnis und die Implementierung eines effizienten Routenplanungsalgorithmus. In diesem Kapitel werden die wichtigsten Aspekte des A*-Algorithmus behandelt, darunter die Verwendung von Heuristiken und die Funktionsweise des Algorithmus.

2.1. A*-Algorithmus

Der A*-Algorithmus ist ein leistungsfähiger und weit verbreiteter Algorithmus für die Routenplanung in Graphen. Er kombiniert geschickt die Verwendung von Heuristiken mit einer Suche in der Breite oder in der Tiefe, um den besten Weg zwischen einem Startknoten und einem Zielpunkt zu finden.

Entwickelt wurde der Algorithmus 1968 von Peter Hart, Nils Nilsson und Bertram Raphael. Der Name A* ist eine Anspielung auf die verwendete Bewertungsfunktion, die die Schätzung des kürzesten Weges von einem gegebenen Knoten zu einem Zielknoten unter Berücksichtigung des bereits zurückgelegten Weges berücksichtigt. [1]

Der A*-Algorithmus kombiniert die Vorteile des Dijkstra-Algorithmus und des Best-First-Search-Algorithmus. Er arbeitet, indem er die Knoten des Graphen systematisch durchsucht und dabei den bisher geringsten Kosten von einem Startknoten zu jedem besuchten Knoten folgt. Dazu verwendet er eine Heuristik, die die verbleibenden Kosten vom aktuellen Knoten bis zum Zielknoten abschätzt. Diese Heuristik ist die entscheidende Komponente, die es dem A*-Algorithmus ermöglicht, effizientere Suchpfade zu finden, indem vielversprechende Pfade bevorzugt und weniger vielversprechende Pfade ignoriert werden.

Während des Suchvorgangs berechnet der A*-Algorithmus für jeden Knoten x

$$f(x) = g(x) + h(x) \tag{2.1}$$

wobei $g(x)$ die bisherige Kostenfunktion vom Startknoten zum Knoten x und $h(x)$ die Heuristik zur Schätzung der Kosten vom Knoten x zum Zielknoten ist. Durch die Wahl des Knotens mit dem kleinsten $f(x)$ wird sichergestellt, dass der Algorithmus den kürzesten Weg findet. [2]

Es ist jedoch zu beachten, dass die Effizienz und Genauigkeit des A*-Algorithmus stark von der verwendeten Heuristik abhängt. Eine gut gewählte Heuristik kann die Suche beschleunigen, während eine schlechte Heuristik möglicherweise nicht die optimale Lösung liefert oder die Laufzeit des Algorithmus erhöht.

2.2. Heuristiken im A*-Algorithmus

Ein zentraler Aspekt des A*-Algorithmus ist die Verwendung von Heuristiken, um den Suchraum effizienter zu durchsuchen. Eine Heuristik ist eine Funktion, die eine Schätzung des verbleibenden Aufwands für die Erreichung des Zielpunkts liefert.
Sie wird verwendet, um die Auswahl der nächsten Knoten zu beeinflussen und somit den Algorithmus auf den vielversprechendsten Pfad zu bringen. [2]

2.2.1. Manhattan-Heuristik

Die Manhattan-Heuristik verwendet die Summe der absoluten Differenzen der Koordinaten zweier Punkte. Sie misst die horizontale und vertikale Entfernung zwischen den Punkten und vernachlässigt dabei diagonale Bewegungen. Die Manhattan-Heuristik ist in Situationen nützlich, in denen nur horizontale und vertikale Bewegungen erlaubt sind, wie zum Beispiel in Schachbrettmustern.

2.2.2. Euklidische-Heuristik

Die Euklidische-Heuristik verwendet den euklidischen Abstand zwischen zwei Punkten in einem Kartesischen Koordinatensystem. Sie ist in Situationen nützlich, in denen es eine klare räumliche Struktur gibt und Bewegungen in beliebigen Richtungen möglich sind.

2.2.3. Wahl einer Heuristik

Die Wahl der Heuristik hängt von der spezifischen Anwendung ab und kann einen erheblichen Einfluss auf die Effizienz und Genauigkeit des A*-Algorithmus haben. Um zu entscheiden, welche Heuristik besser geeignet ist, sollte man die spezifischen Anforderungen und Eigenschaften des Problems berücksichtigen.

Abbildung 2.1.: Vergleich der Heuristiken

In Abbildung 2.1 kann man sehr gut den Unterschied der beiden Heuristiken erkennen. Wenn die Bewegung in beliebigen Richtungen stattfinden können und die räumliche Struktur wichtig ist, liefert die euklidische-Heuristik eine bessere Schätzung. Wenn jedoch nur horizontale und vertikale Bewegungen möglich sind oder die räumliche Struktur diskret ist, ist die Manhattan-Heuristik angemessener.
Da ein Routenplanungsalgorithmus nicht auf horizontale und vertikale Bewegungen beschränkt sein sollte, ist die euklidische Heuristik für diesen Anwendungsfall die bessere Wahl.

2.3. Vergleich mit anderen Routenplanalgorithmen

Die Routenplanung ist ein grundlegendes Problem in vielen Anwendungsbereichen, und es gibt verschiedene Algorithen, die entwickelt wurden, um dieses Problem zu lösen. In diesem Abschnitt wird der A*-Algorithmus mit anderen bekannten Routenplanungsalgorithmen verglichen und seine Stärken und Schwächen aufgezeigt.

2.3.1. Dijkstra-Algorithmus

Der Dijkstra-Algorithmus ist einer der ältesten und am häufigsten verwendeten Algorithmen für die Routenplanung in gewichteten Graphen. Er findet den kürzesten Weg von einem Startknoten zu allen anderen Knoten im Graphen. Im Gegensatz zum A*-Algorithmus verwendet der Dijkstra-Algorithmus keine Heuristiken, sondern durchläuft alle Knoten, um den kürzesten Pfad zu finden [3]. Dies führt dazu, dass der Dijkstra-Algorithmus bei großen Graphen, insbesondere bei Graphen mit vielen Knoten, langsamer sein kann als der A*-Algorithmus.

2.3.2. Bellman-Ford-Algorithmus

Der Bellman-Ford-Algorithmus ist ein weiterer bekannter Algorithmus, der ebenfalls mit gewichteten Graphen arbeitet. Im Gegensatz zum Dijkstra-Algorithmus kann der Bellman-Ford-Algorithmus jedoch auch mit Graphen umgehen, die negative Kantenlängen enthalten. Dies macht ihn zu einer geeigneten Wahl in bestimmten Situationen, in denen der A*-Algorithmus möglicherweise nicht angewendet werden kann. Allerdings ist der Bellman-Ford-Algorithmus in der Regel langsamer als der A*-Algorithmus, insbesondere bei großen Graphen.

2.3.3. Greedy Best-First-Search

Greedy Best-First-Search ist ein weiterer heuristischer Algorithmus, der für die Routenplanung verwendet werden kann. Ähnlich wie der A*-Algorithmus verwendet er eine Heuristik, um die Auswahl der nächsten Knoten zu beeinflussen. Allerdings verwendet Greedy Best-First-Search ausschließlich die Heuristik, ohne die bisherigen Kosten zu berücksichtigen. Dies führt dazu, dass Greedy Best-First-Search möglicherweise nicht den optimalen Pfad findet und zu suboptimalen Ergebnissen führen kann. [4]

2.3.4. Breitensuche (BFS) und Tiefensuche (DFS)

Breitensuche und Tiefensuche sind einfache Suchalgorithmen, die auch für die Routenplanung verwendet werden können. Sie durchsuchen den Graphen schrittweise, ohne Heuristiken oder Kosten zu berücksichtigen. Obwohl diese Algorithmen einfach zu implementieren sind, sind sie möglicherweise nicht die effizienteste Wahl für die Routenplanung in großen Graphen, da sie den Suchraum nicht effizient durchsuchen.

3. Datenstrukturen und Implementierungsdetails

3.1. Einlesen einer GraphML-Datei

Dieses Kapitel behandelt das Einlesen einer GraphML-Datei und die Strukturierung der darin enthaltenen Knoten- und Kanteninformation.
GraphML ist ein XML-basiertes Dateiformat, das häufig zur Darstellung von gerichteten und ungerichteten Graphen verwendet wird. Die Verarbeitung von GraphML-Dateien ist in verschiedenen Anwendungsbereichen wie Netzwerkvisualisierung, soziale Netzwerkanalyse oder Routenplanung unerlässlich.

Das Kapitel beginnt mit einer Analyse der GraphML-Struktur, um ein Verständnis dafür zu entwickeln, wie die Knoten- und Kanteninformationen in der Datei organisiert sind. Darauf aufbauend werden die Funktionen und Variablen in der Header-Datei „car_readFile.hpp“ erläutert, die Teil der Klasse ReadFile sind. Diese Klasse wurde speziell entwickelt, um die GraphML-Datei zu öffnen, zu lesen und die darin enthaltenen Daten effizient in einer unordered_map zu speichern.

Die Details der Implementierung der Klasse sind in der Datei „car_readFile.cpp“ beschrieben. Die Methode readFile() spielt eine zentrale Rolle und ermöglicht es, die Informationen aus der GraphML-Datei zu extrahieren und in der Datenstruktur abzulegen.

3.1.1. Struktur einer GraphML-Datei

Die Struktur einer GraphML-Datei ist in Abbildung 3.1 dargestellt. Es handelt sich um ein XML-Dokument, das Informationen über einen gerichteten Graphen enthält. Jeder Knoten des Graphen wird durch eine eindeutige ID identifiziert. Für jeden Knoten gibt es zwei Schlüssel, die jeweils den x- und y-Koordinaten des Knotens entsprechen.
Der Wert des Schlüssels „d0“ gibt die x-Koordinate des Knotens an, während der Wert des Schlüssels „d1“ die y-Koordinate des Knotens angibt.

```
<?xml version='1.0' encoding='utf-8'?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
<key attr.name="dotted" attr.type="boolean" for="edge" id="d2" />
<key attr.name="y" attr.type="double" for="node" id="d1" />
<key attr.name="x" attr.type="double" for="node" id="d0" />
<graph edgedefault="directed">
<node id="1">
<data key="d0">1</data>
<data key="d1">1</data>
</node>
<!-- additional edges omitted for brevity -->
<edge source="1" target="2">
<data key="d2">true</data>
</edge>
<!-- additional nodes omitted for brevity -->
</graph>
</graphml>
```

Abbildung 3.1.: Struktur einer GraphML-Datei

Es gibt auch einen Kantenschlüssel („d2“), der einen booleschen Wert enthält, der angibt, ob die Fahrbahn mehrspurig ist oder nicht. Damit ist ein Spurwechsel möglich. Der Graph selbst ist als „gerichtet“ definiert, was bedeutet, dass jeder Knoten des Graphen Kanten zu anderen Knoten hat, diese Kanten aber nur in eine Richtung verlaufen.

3.1.2. Die Header-Datei „car_readFile.hpp“ und ihre Funktionen

Die Header-Datei enthält die Definitionen der Klasse ReadFile, die zum Lesen und Verarbeiten von GraphML-Dateien verwendet wird. Die Datei beginnt mit Präprozessoranweisungen, um ein mehrfaches Einbinden der Datei zu verhindern. Anschließend werden verschiedene Header-Dateien der Standardbibliothek eingebunden, die für die Funktionalität der Klasse erforderlich sind. Die ReadFile-Klasse selbst ist in der Datei definiert.

Datenstruktur „NodeData“

Vor der Definition der Klasse ReadFile wird die Datenstruktur „NodeData“ definiert, die Informationen über einen Knoten im Graphen enthält. Ein „NodeData“-Objekt besteht aus drei Attributen:

- „m_id“: Eine 16-Bit unsigned Integer-Variable zur Speicherung der x- und y-Koordinaten des Knotens.
- „m_x“ und „m_y“: Float-Variablen zur Speicherung der x- und y-Koordinaten des Knotens.
- „connectedNodes“: Ein Vektor von Integerwerten, der die IDs der Knoten enthält, die mit diesem Knoten verbunden sind.

Die Struktur enthält auch einen Konstruktor, um die Initialisierung von „NodeData“-Objekten mit den gegebenen Werten zu erleichtern, sowie einen überladenen Operator „<“, um die Sortierung der Knoten nach ihrer ID zu ermöglichen.

Klasse „ReadFile“

Die Klasse selbst ist nun definiert. Sie enthält verschiedene öffentliche und geschützte Methoden und Variablen zum Lesen und Speichern von GraphML-Daten.
Es stehen zwei Konstruktoren zur Verfügung:
Der Standardkonstruktor „ReadFile()“ initialisiert die Variable „m_filename“ mit einem vorgegebenen Dateinamen „c_filename“.
Der zweite Konstruktor „ReadFile(string filename)“ akzeptiert einen Dateinamen als Parameter und weist diesen der Variablen „m_filename“ zu.

Die öffentlichen Methoden sind:

1. „readFile()“: Dies ist der Hauptteil der Klasse, in dem die GraphML-Datei geöffnet, gelesen und die Knoten und Kanten in einer „unordered_map“ namens „m_nodes“ gespeichert werden.

2. „setFilename(string filename)“: Mit dieser Methode kann der Dateiname der GraphML-Datei nachträglich geändert werden.

3. „getNodes()“: Diese Methode gibt eine „unordered_map“ zurück, das die Informationen über die gelesenen Knoten enthält.

Die geschützte Methoden dienen dazu, bestimmte Teile des Leseprozess für Knoten und Kanten aus der GraphML-Datei zu isolieren und zu verarbeiten.

1. „readNodeDataFromFile()“: Diese Methode extrahiert die Daten eines einzelnen Knotens aus der GraphML-Datei. Sie liest die Zeilen der Datei, bis die Daten des Knotens vollständig gelesen sind und speichert sie dann im Objekt „m_currentNode“ vom Typ „NodeData“.

2. „readEdgeDataFromFile()“: Diese Methode extrahiert die Daten einer einzelnen Kante aus der GraphML-Datei. Sie liest die Zeilen der Datei, bis die Daten der Kante vollständig gelesen sind und aktualisiert dann die Verbindungen des entsprechenden Quellknotens in der „unordered_map“.

Die geschützten Variablen werden verwendet, um temporäre und während des Lesens relevante Daten zu speichern.

1. „m_currentNode“: Ein Objekt vom Typ „NodeData“, das die Daten eines Knotens temporär speichert, während sie aus der Datei gelesen werden.

2. „m_nodes“: Eine „unordered_map“, die die Informationen über die gelesenen Knoten speichert. Der Schlüssel ist die ID des Knotens (uint16_t) und der Wert ist das entsprechende „NodeData“-Objekt.

3. „graphMlFile“: Ein Dateistream, der die GraphML-Datei repräsentiert, die zum Lesen der Daten geöffnet wird.

4. „m_filename“: Eine Zeichenkette, die den Dateinamen der GraphML-Datei enthält.

5. „m_currentLine“: Eine Zeichenkette, die die aktuelle Zeile darstellt, während die Datei Zeile für Zeile durchlaufen wird.

3.1.3. Implementierung des GraphML-Einleseprozess

Die Methode „ReadFile::readFile()“ bildet den Hauptteil der Implementierung und ist für das eigentliche Einlesen der GraphML-Datei verantwortlich. Sie öffnet die GraphML-Datei mit dem in „m_filename“ gespeicherten Namen und geht diese Zeile für Zeile durch. Dabei werden die Informationen über jeden Knoten und jede Kante extrahiert und in „m_nodes“ gespeichert. Diese Datenstruktur ermöglicht einen schnellen Zugriff auf die Graphendaten und erleichtert spätere Analysen.

Zunächst wird die GraphML-Datei geöffnet, deren Name in der Member-Variablen „m_filename“ gespeichert ist. Dies ermöglicht den Zugriff auf die in der Datei enthaltenen Daten. Das Öffnen erfolgt über den C++-Standard-Dateistream fstream, der zum Lesen und Schreiben von Dateien verwendet wird. Dabei wird die Datei im Lesemodus (d.h. als Eingabedatei) geöffnet, um ihren Inhalt lesen zu können. Nach dem Aufruf von „open()“ ist die GraphML-Datei geöffnet und der Datei-Stream „graphMlFile“ kann auf die Datei zugreifen.

Kann die Datei nicht erfolgreich geöffnet werden, z.B. weil sie nicht existiert oder die Zugriffsrechte fehlen, wird eine Fehlermeldung ausgegeben und eine Exception geworfen, um den ordnungsgemäßen Ablauf zu gewährleisten.

Anschließend wird die Datei Zeile für Zeile durchsucht. Jede Zeile wird daraufhin überprüft, ob sie einen Knoten oder eine Kante darstellt. Dies geschieht mit Hilfe der Tags „<node>“ und „<edge>“, die jeweils den Beginn eines Knotens bzw. einer Kante markieren.

Wenn die aktuelle Zeile einen Knoten darstellt, werden die Daten dieses Knotens aus der Zeile extrahiert. Dabei werden Informationen wie die eindeutige Knoten-ID sowie die x- und y-Koordinaten aus den entsprechenden Tags und Attributen ausgelesen und in der Variable „m_currentNode“ gespeichert.

Nachdem die Knotendaten vollständig eingelesen wurden, wird der aktuelle Knoten in „m_nodes“ gespeichert. Die Knoten-ID dient dabei als Schlüssel, um später einfach auf die gespeicherten Knoten zugreifen zu können.

Gleiches gilt für die Extraktion der Kanteninformation. Wenn die aktuelle Zeile eine Kante darstellt, wird die Information dieser Kante aus der Zeile extrahiert. Die Quell- und Zielknoten der Kante werden ermittelt und die entsprechenden Verbindungen in „m_nodes“ aktualisiert. Dadurch wird die Beziehung zwischen den Knoten im Graphen festgehalten.

Nachdem alle Zeilen der Datei durchgelaufen wurden, ist der Einlesevorgang abgeschlossen und die Methode „readFile“ wird beendet. Die Daten über die Knoten und ihre Verbindungen sind nun in „m_nodes“ gespeichert und stehen für weitere Verarbeitungsschritte zur Verfügung.

3.2. Implementierung des A*-Algorithmus

3.2.1. Die Header-Datei „car_algorithm.hpp“ und ihre Funktionen

Die Header-Datei enthält die Definition der Klasse, die den A*-Algorithmus implementiert. In der Header-Datei wird auch die Node-Struktur definiert, mit der die Knoten und ihre Bewertungen verwaltet werden.

Mehrere Funktionen sind in der Klasse „AStarAlgorithm“ deklariert:

- Die Funktion „heuristic(NodeData& nodeA, nodeData& nodeB)“, die die heuristische Bewertung zwischen zwei Knoten berechnet. Die Heuristik wird verwendet, um die verbleibenden Kosten vom aktuellen Knoten zum Zielknoten abzuschätzen.

- Die Hauptfunktion „aStarAlgorithm(int startId, int goalId)“, die den A*-Algorithmus ausführt, um den kürzesten Weg zwischen den Knoten mit den IDs „startId“ (Startknoten) und „goalId“ (Zielknoten) zu berechnen.

- Die Funktion „getPath()“, die den berechneten kürzesten Pfad als Vektor von Knoten-IDs zurückgibt.

- Die Funktion „getReadFile()“, die eine Referenz auf das „ReadFile“-Objekt zurückgibt, um auf die eingelesenen Graphendaten zugreifen zu können.

3.2.2. Implementierung

Die Implementierung des A*-Algorithmus ist in der Datei „car_Algorithm.cpp“ enthalten. Die Hauptfunktion „aStarAlgorithm(int startId, int goalId)“ beinhaltet die Implementierung des A*-Algorithmus zur Suche des kürzesten Weges zwischen einem Start- und einem Zielknoten in einem gegebenen Graphen.

Zu Beginn der Funktion werden zwei Hash-Maps „gScores“ und „cameFrom“ erzeugt, die die g-Werte ($g(x)$), welche die bisher kumulierten Kosten für den Weg vom Startknoten zum betrachteten Knoten repräsentieren, und die Vorgängerinformationen für jeden Knoten während des Suchprozesses speichern.

Der A*-Algorithmus arbeitet in einer Schleife, die solange ausgeführt wird, bis die Prioritätswarteschlange „openSet“ leer ist oder der Zielknoten erreicht wurde.
Bei jedem Schleifendurchlauf wird der Knoten mit der niedrigsten f-Bewertung (für den Zielknoten) aus der Warteschlange „openSet“ genommen und als aktueller Knoten betrachtet. Die f-Bewertung eines Knotens ergibt sich aus der Summe des g-Wertes (kumulierte Kosten vom Startknoten zum aktuellen Knoten) und des h-Wertes (heuristische Bewertung der geschätzten Kosten vom aktuellen Knoten zum Zielknoten). Die heuristische Bewertung wird mit Hilfe der Funktion „heuristic(NodeData& nodeA, nodeData& nodeB)“ berechnet, die die euklidische Distanz zwischen den beiden Knoten „nodeA“ und „nodeB“ im 2D-Raum zurückgibt.
Die euklidische Distanz wird als heuristische Funktion verwendet, um die Luftliniendistanz zwischen den Knoten zu bestimmen und so eine optimistische Schätzung der verbleibenden Kosten zu erhalten.

Wenn der aktuelle Knoten den Zielknoten darstellt, wurde der kürzeste Pfad gefunden und der Rekonstruktionsprozess wird gestartet, um den Pfad zurückzuverfolgen und die Knoten-IDs in den Vektor „m_path“ einzufügen. Auf diese Weise enthält der Vektor den berechneten kürzesten Pfad, der den Weg vom Startknoten zum Zielknoten repräsentiert.

Falls der Zielknoten noch nicht erreicht wurde, werden die Nachbarn des aktuellen Knotens betrachtet und deren g- und f-Werte entsprechend aktualisiert. Ist der Nachbarknoten noch nicht in im „gScores“ enthalten oder wurde ein besserer Weg zu diesem Knoten gefunden, wird der Nachbarknoten in die Prioritätswarteschlange „openSet“ aufgenommen. Die Warteschlange organisiert die Knoten in aufsteigender Reihenfolge ihrer f-Werte, wodurch der Algorithmus die vielversprechendsten Knoten für die Expansion priorisiert.
Die Vorgängerinformation wird ebenfalls in der Hash-Map „cameFrom“ aktualisiert.

Der A*-Algorithmus verwendet die heuristische Bewertung, um den besten nächsten Knoten für die Expansion auszuwählen und somit den Pfad zu optimieren. Durch diese optimierte Expansionsstrategie sucht der Algorithmus effizienter nach dem kürzesten Pfad, indem er die vielversprechendsten Knoten zuerst untersucht, was die Anzahl der betrachteten Knoten und damit die Gesamtlaufzeit des Algorithmus reduziert.

Am Ende der Schleife enthält der Vektor „m_path“ den berechneten kürzesten Pfad, der den Weg vom Startknoten zum Zielknoten darstellt. Dieses Ergebnis kann zur Navigation durch das Graphennetz verwendet werden, z.B. zur Routenplanung in Navigationsanwendungen oder zur Navigation autonomer Fahrzeuge auf Straßennetzen. Die Verwendung des A*-Algorithmus ermöglicht es, den optimalen Pfad unter Berücksichtigung der Heuristik effizient zu finden und somit den Ressourcenverbrauch und die Gesamtfahrzeit zu minimieren.

4. Evaluierung und Performance Analyse

4.1. Validierung und Testen der Implementierung

In diesem Abschnitt gilt es, die implementierten Funktionen und den Routenplanungsalgorithmus zu validieren und zu testen. Das Testen ist ein entscheidender Schritt in der Softwareentwicklung, der sicherstellt, dass der Code zuverlässig und fehlerfrei funktioniert.

Um die Funktionsfähigkeit der Implementierung sicherzustellen, wurden verschiedene Unit-Tests entwickelt. Diese Tests überprüfen, ob die implementierten Funktionen die erwarteten Ergebnisse liefern und ob die Daten korrekt in den dafür vorgesehenen Datenstrukturen gespeichert werden.

Die Unit-Tests wurden mit dem Google Test Framework durchgeführt, einem leistungsfähigen Werkzeug zur Erstellung automatisierter Tests für C++-Programme. Dieses Framework ermöglicht die Definition und Ausführung von Testfällen, um sicherzustellen, dass die einzelnen Komponenten korrekt funktionieren.

4.1.1. Testframework Google Test

Google Test ist ein Framework für automatisierte Tests in C++. Es wurde von Google entwickelt und ist Open-Source. Durch die Verwendungen des Google Test Frameworks können Entwickler sicherstellen, dass die Funktionalität der Klasse bei Änderungen oder Updates nicht beeinträchtigt wird und die Anforderungen der Anwendung erfüllt werden. Die Testsuite bietet eine wichtige Komponente für die Softwareentwicklung, um die Zuverlässigkeit und Qualität der Implementierung sicherzustellen.

Ein wichtiger Bestandteil von Google Test sind Testfälle. Testfälle sind einzelne Testfunktionen, die bestimmte Aspekte des Codes überprüfen. Jeder Testfall besteht aus einer Reihe von Test-Assertions, die sicherstellen, dass bestimmte Bedingungen erfüllt sind. Wenn eine Assertion fehlschlägt, wird der Test als fehlgeschlagen markiert.

Ein weiteres nützliches Feature ist die Möglichkeit, testinterne Daten zu sammeln und auszuwerten. Mit Google Test können Entwickler beispielsweise die Ausführungszeit jedes Tests messen und protokollieren.

Es bietet auch eine Unterstützung für das parallele Ausführen von Tests. So können Entwickler mehrere Tests gleichzeitig ausführen, um die Ausführungszeit zu verkürzen und die Ressourcenauslastung zu minimieren.

Google Test bietet auch Unterstützung für die Organisation von Testfällen in Test-Fixtures. Ein Fixture ist eine Klasse oder Struktur, die als Grundlage für Tests verwendet wird. Es stellt gemeinsame Daten und Funktionalitäten bereit, die von mehreren Testfällen genutzt werden können.
In Abbildung 4.1 ist ein Beispiel eines Test- Fixtures zu sehen.

```
#include <gtest/gtest.h>
#include "driver_assist.h"

class DriverAssistTests : public ::testing::test
{
  protected:
  DriverAssist driverAssist;
};

TEST_F(DriverAssistTests, CheckSpeed)
{
  int speedLimit          = 60;
  int currentSpeed        = 55;
  bool isSpeedWithinLimit =
     driverAssist.CheckSpeed(currentSpeed, speedLimit);
  EXPECT_TRUE(isSpeedWithinLimit);
}
```

Abbildung 4.1.: Beispiel für ein Test-Fixture

In diesem Beispiel wird eine Klasse „DriverAssistTest“ als Fixture verwendet.
Diese Klasse stellt eine Instanz der Klasse „DriverAssist“ bereit.

„TEST_F“ wird verwendet, um einen Testfall für die Funktion „CheckSpeed“ durchzuführen. In diesem Test wird das Geschwindigkeitslimit auf 60 gesetzt und die aktuelle Geschwindigkeit auf 55 geprüft. Da die aktuelle Geschwindigkeit unter dem Geschwindigkeitslimit liegt, sollte die „CheckSpeed“ „true“ zurückgeben. Der Rückgabewert wird von dem Google Testframework überprüft.

Ein weiters Feature von Google Test ist die Möglichkeit, Tests mit Parametern auszuführen. Mit dieser Funktion können Entwickler einen Test mehrfach mit unterschiedlichen Eingabewerten ausführen, ohne dass sie mehrere Testfunktionen erstellen müssen. Um einen parametrisierten Test durchzuführen, muss das Fixture wie in Abbildung 4.2 zu sehen ist, erweitert werden.

In diesem Beispiel wird das Fixture „DriverAssistTest“ erweitert, um ein Tupel aus drei Werten als Parameter zu verwenden. Hierfür wird das Makro „TEST_P“ verwendet, um den Test auszuführen. Die Parameter werden dabei über „GetParam“ bereitgestellt und können über die Funktion „std::get“ abgerufen werden.

```
TEST_P(DriverAssistTests, CheckSpeed)
{
  int speedLimit          = GetParam();
  int currentSpeed        = GetParam();
  bool expectedResult     = GetParam();
  bool isSpeedWithinLimit =
     driverAssist.CheckSpeed(currentSpeed, speedLimit);
  EXPECT_TRUE(isSpeedWithinLimit);
}

INSTANTIATE_TEST_CASE_P(CheckSpeedTests, DriverAssistTest,
  ::testing::Values(
  std::make_tuple(70, 55, true),
  std::make_tuple(70, 75, false),
  std::make_tuple(30, 32, false),
  std::make_tuple(30, 30, true)
))
```

Abbildung 4.2.: Beispiel für ein parametrisierten Test

Mit der Instruktion „INSTANTIATE_TEST_CASE_P“ werden mehrere Testfälle mit unterschiedlichen Parametern erstellt. In diesem Beispiel werden vier Testfälle erstellt, bei denen jeweils ein unterschiedliches Tupel mit Geschwindigkeitslimit, aktueller Geschwindigkeit und dem erwarteten Ergebnis übergeben wird.

4.1.2. Testfälle zum Einlesen der GraphML-Datei

Die Datei „unittest_readFile.hpp“ enthält eine Testsuite für die Klasse „ReadFile“. Die Testsuite wurde mit dem Testframework Google Test erstellt und enthält zwei Testfälle, „readTest1“ und „readTest2“, um die Funktionalität der Klasse „ReadFile“ zu überprüfen.

Die Klasse „carReadFile_Fixture“ wird als Test-Fixture erstellt, indem sie aus der Klasse „testing::Test“ und der Klasse „ReadFile“ abgeleitet wird. Die Verwendung einer Test-Fixture ermöglicht die gemeinsame Nutzung von Ressourcen und Funktionen zwischen den Testfällen.

Innerhalb der Test-Fixture werden zwei Membervariablen definiert, „m_ReadFile“ und „m_testNodes“. Die Variable „m_ReadFile“ wird verwendet, um eine Instanz der Klasse „ReadFile“ zu erzeugen und die Funktionalität der Dateieinlesefunktionen zu testen. Die Variable „m_testNodes“ wird verwendet, um Testdaten in Form von „NodeData“-Objekten zu speichern, die später in den Testfällen verwendet werden.

Der erste Testfall ist definiert. In diesem Testfall werden die Dateilesefunktionen der Klasse getestet, indem eine benutzerdefinierte Test-GraphML-Datei eingelesen wird. Zuerst werden Testdaten in Form eines „NodeData“-Objektes erzeugt und im „m_testNodes“-Vektor gespeichert. Dann wird der Dateiname für die Test-GraphML-Datei mit „m_ReadFile.setFilename()“ gesetzt. Danach wird erwartet, das die Datei ohne Fehler gelesen wird, indem das Makro „EXPECT_NO_THROW“ verwendet wird. Schließlich wird das Ergebnis der Dateieinlesefunktion in einer Variable gespeichert und es werden verschiedene „EXPECT_EQ“-Anweisungen verwendet, um sicherzustellen, dass die gelesenen Knotendaten mit den erwarteten Testdaten übereinstimmen.

Der zweite Testfall ist ähnlich wie der erste, unterscheidet sich aber darin, dass er eine andere Test-GraphML-Datei verwendet und zusätzliche Testdaten für mehrere Knoten enthält. Die Funktionsweise des Testfalls ist im Wesentlichen die gleiche, aber es wird eine Schleife verwendet, um mehrere Knoten zu überprüfen.

Dieser Code stellt sicher, dass die Klasse korrekt funktioniert und die gelesenen Daten korrekt gespeichert werden.

4.1.3. Testfälle für die Implementierung des A*-Algorithmus

In der Datei „unittest_Algorithm.cpp“ ist eine vollständige Testsuite für die Implementierung des A*-Algorithmus enthalten. Die Testsuite besteht aus drei Testfällen, „simpleAStarTest1“, „simpleAStarTest2“ und „crossParking“. Das Hauptziel dieser Testsuite ist es, die Funktionalität des A*-Algorithmus zu überprüfen und sicherzustellen, dass er in verschiedene Szenarien korrekte Ergebnisse liefert.

Der erste Testfall überprüft den A*-Algorithmus anhand eines einfachen Beispiels. Der Algorithmus soll den kürzesten Weg zwischen den Knoten 115 und 118 des Graphen in Abbildung 4.3 finden. Dazu wird die Dateilesefunktion der Klasse „ReadFile “ verwendet, um die Graphendaten aus der GraphML-Datei einzulesen. Anschließend wird der A*-Algorithmus mit dem Start- und Zielknoten 115 und 118 aufgerufen und der berechnete Pfad mit einem vordefinierten erwarteten Pfad verglichen. Ist der tatsächliche Pfad leer (kein Pfad gefunden), wird eine Fehlermeldung ausgegeben.

Der zweite Testfall testet den Algorithmus an einem weiteren einfachen Beispiel. Der Algorithmus soll im gegebenen Graphen den kürzesten Weg zwischen den Knoten 57 und 17 finden. Dazu werden wie im vorherigen Testfall die Dateilesefunktion und der A*-Algorithmus aufgerufen und berechnete Pfad mit einem erwarteten Pfad verglichen.

Der dritte Testfall überprüft den A*-Algorithmus anhand eines komplexeren Beispiels. Dazu wird eine weitere GraphML-Datei eingelesen. Auch hier werden die Dateilesefunktion und der A*-Algorithmus aufgerufen und der berechnete Pfad mit einem vordefinierten Pfad verglichen.

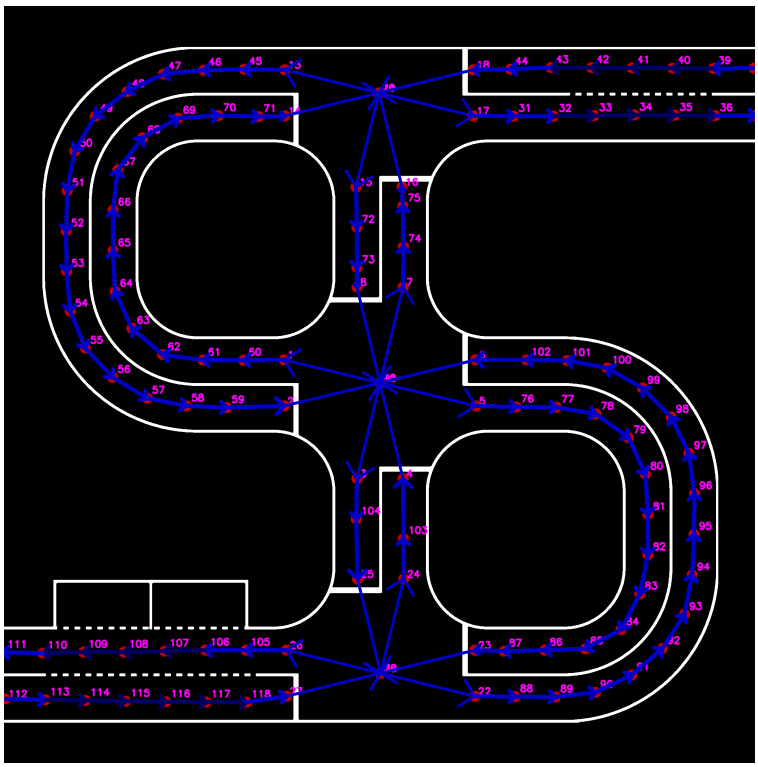

Abbildung 4.3.: Test-Track

Die Testfälle dienen dazu, die Korrektheit und Effizienz der Implementierung des A*-Algorithmus zu überprüfen und sicherzustellen, dass er für verschiedene Szenarien korrekte Ergebnisse liefert. Wenn die tatsächlichen Pfade mit den erwarteten Pfaden übereinstimmen, gelten die Tests als erfolgreich. Andernfalls wird eine Fehlermeldung ausgegeben und der Entwickler muss die Implementierung überprüfen, um mögliche Fehler oder Optimierungsmöglichkeiten zu identifizieren. Die Testsuite stellt die robuste und zuverlässige Funktionalität des A*-Algorithmus in der Klasse sicher, was für die sichere Navigation im Graphennetzwerk in einer autonomen Fahrzeuganwendung von entscheidender Bedeutung ist.

4.2. Leistungsvergleich des A*-Algorithmus: Ausführungszeiten auf verschiedenen Graphen

In diesem Abschnitt werden Laufzeit und Speicherbedarf des A*-Algorithmus anhand von Benchmark-Tests verglichen. Die Tests wurden mit dem Google Benchmark Framework durchgeführt, einer leistungsfähigen C++ Benchmark Suite, die von Google entwickelt wurde. Das Framework ermöglicht es Entwicklern, die Performance von C++-Code zu messen, verschiedene Implementierungen zu vergleichen und Engpässe oder Performance-Probleme zu identifizieren.

Die Ergebnisse der Benchmark-Tests sind in der Tabelle 4.1 dargestellt, in der die Ausführungszeit (Time), die CPU-Zeit (CPU) und die Anzahl der Iterationen (Iterations) für verschiedene Szenarien aufgelistet sind.

Benchmark	Time	CPU	Iterations
Simple	4384524 ns	2703059 ns	237
Complex	179371133 ns	132812500 ns	6
Multiple runs	33912684 ns	27832031 ns	32
Multiple runs with competition track	8878551100 ns	6921875000 ns	1

Tabelle 4.1.: Benchmark-Ergebnisse des A*-Algorithmus

Ein Blick auf die Tabelle zeigt einen deutlichen Unterschied in der Ausführungszeit des A*-Algorithmus auf einfachen und komplexen Graphen. Die Ausführungszeit auf einfachen Graphen betrug durchschnittlich etwa 4 ms, während auf komplexen Graphen durchschnittlich etwa 179 ms benötigt wurden. Diese Ergebnisse bestätigten die Erwartung, dass die Suche nach dem kürzesten Weg auf einfachen Graphen weniger rechenintensiv ist.

Ein weiteres interessantes Ergebnis ist die Stabilität der Laufzeit bei wiederholter Ausführung. Wie in Tabelle 4.1 zu sehen ist, zeigt der Benchmark-Test „Multiple runs“ eine geringe Variation in den Laufzeiten. Dies deutet darauf hin, dass der Algorithmus konsistente und vorhersagbare Ergebnisse liefert, auch wenn es mehrmals ausgeführt wird.

Es ist wichtig anzumerken, dass sich die in dieser Analyse präsentierten Ergebnisse hauptsächlich auf die Ausführungszeit beziehen. Die genaue Messung des Speicherbedarfs erfordert spezielle Werkzeuge und Mechanismen zur Ressourcenüberwachung, die in diesem Rahmen nicht durchgeführt wurden

Die Ergebnisse der Benchmarks zeigen, dass der A*-Algorithmus einen leistungsfähigen Algorithmus zur Routenplanung darstellt. Die Laufzeit hängt jedoch stark von der Komplexität des zugrundeliegenden Graphen ab.

5. Fazit

Die Implementierung des A*-Algorithmus in C++ ermöglichte ein tieferes Verständnis der Funktionsweise des Algorithmus. Durch die praktische Umsetzung konnten die einzelnen Schritte des Algorithmus detailliert nachvollzogen und seine Funktionsweise verinnerlicht werden. Die Wahl der Heuristik und die effiziente Nutzung der Datenstrukturen erwiesen sich als entscheidend für die Leistungsfähigkeit des Algorithmus. Ein sorgfältige Auswahl der Heuristik kann die Effizienz des A*-Algorithmus erheblich verbessern und eine schnellere Suche nach dem optimalen Pfad ermöglichen.

Die Evaluierung der Performance des A*-Algorithmus auf verschiedenen Graphen führte zu interessanten Erkenntnissen. Es zeigte sich, dass die Laufzeit des Algorithmus stark von der Größe und Struktur des Graphen abhängt. In einfachen Graphen mit wenigen Knoten und Kanten konnte der A*-Algorithmus schnell eine optimale Lösung finden. In komplexeren Graphen mit vielen Knoten und Kanten stieg die Laufzeit jedoch deutlich an, da der Algorithmus eine umfangreichere Suche durchführen musste.

Allerdings wurden auch einige Grenzen des A*-Algorithmus aufgezeigt. In bestimmten Situationen, wie z.B. bei sehr komplexen Graphen ohne klar definierte Ziele, verlor der Algorithmus an Effizienz und näherte sich dem optimalen Pfad nur langsam an. In solchen Fällen könnten alternative Routenplanungsalgorithmen, wie der Bellman-Ford-Algorithmus oder die Breitensuche bessere Ergebnisse liefern.

Abschließend ist festzuhalten, dass die Implementierung und Evaluierung des Routenplanungsalgorithmus eine wertvolle Erfahrung war und ein fundiertes Verständnis dieses leistungsfähigen Routenplanungsalgorithmus ermöglicht hat. Die gewonnenen Erkenntnisse über die Leistungsfähigkeit und der Vergleich mit anderen Routenplanungsalgorithmen liefern wertvolle Informationen für zukünftige Anwendungen und Optimierungen des A*-Algorithmus.

Literatur

1. *A*-Algorithmus* de. Page Version ID: 219778390. Feb. 2022. https://de.wikipedia.org/w/index.php?title=A*-Algorithmus&oldid=219778390 (2023).
2. Russell, S. J., Norvig, P. & Davis, E. *Artificial intelligence: a modern approach* 3rd ed. ISBN: 978-0-13-604259-4 (Prentice Hall, Upper Saddle River, 2010).
3. Sedgewick, R. & Wayne, K. D. *Algorithms* 4th ed. ISBN: 978-0-321-57351-3 (Addison-Wesley, Upper Saddle River, NJ, 2011).
4. *Greedy Best first search algorithm* en-us. Section: DSA. Dez. 2022. https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/ (2023).

Abbildungsverzeichnis

Tabellenverzeichnis

A. Anhang

Im Anhang dieser Studienarbeit finden Sie weitere Informationen und Materialien, die zur Vertiefung des Themas dienen.

A.1. Code zum lesen der GraphML-Datei

Hier finden Sie den vollständigen Quellcode für das Einlesen der GraphML-Datei in der Programmiersprache C++. Der Quellcode enthält Kommentare und Erklärungen, die den Algorithmus und seine Funktionsweise veranschaulichen.

A.1.1. car_readFile.hpp

```
#ifndef PJ_RA_CAR_READFILE_HPP
#define PJ_RA_CAR_READFILE_HPP

#include <vector>
#include <string>
#include <fstream>
#include <iostream>
#include <sstream>
#include <algorithm>
#include <unordered_map>

using namespace std;

// Default filename for reading graph data (test.graphml).
constexpr char c_filename[] = "test.graphml";
```

```
// Data structure representing a node in the graph.
struct NodeData
{
  NodeData() : m_id{0U}, m_x{0.0}, m_y{0.0}
  {
  }

  NodeData(uint16_t id, float x, float y, vector<int>
      connectedNodes)
  {
    this->m_id = id;
    this->m_x = x;
    this->m_y = y;
    this->connectedNodes = connectedNodes;
  }

  uint16_t m_id;              // The ID of the node in the
      graph.
  float m_x;                  // The x-coordinate of the node.
  float m_y;                  // The y-coordinate of the node.
  vector<int> connectedNodes; // A list of IDs of nodes
      connected to this node.

  // Custom comparison operator used for sorting nodes by their
      ID.
  bool operator <(const NodeData &other) const
  {
    return (this->m_id < other.m_id);
  }
};

// Class for reading data from a graph file.
class ReadFile
{
  public:
  ReadFile();                     // Default constructor with
      default filename.
  ReadFile(string filename);      // Constructor with a custom
      filename.

  void readFile();                // Read the graph data from
      the file.
```

```
  void setFilename(string filename);                // Set a new
     filename for reading graph data.
  unordered_map<uint16_t, NodeData> getNodes();     // Get the
     map of nodes read from the file.
  void exportMapToInl();                           // Export the
     node data to an inline file (not implemented).

  protected:
  NodeData m_currentNode;              // Current node being
     read from the file.
  unordered_map<uint16_t, NodeData> m_nodes;    // Map to store
     all the nodes read from the file.

  fstream graphMlFile;             // Filestream for reading the
     graph data.
  string m_filename;               // Filename of the graph file
     being read.
  string m_currentLine;            // Current line being
     processed while reading the file.

  // Read node data from the file and store it in the current
     node.
  void readNodeDataFromFile();

  // Read edge data from the file and update connectedNodes of
     nodes accordingly.
  void readEdgeDataFromFile();
};

#endif
```

A.1.2. car_readFile.cpp

```
#include "car_readFile.hpp"

// Default constructor sets the filename to the default value
   (test.graphml).
ReadFile::ReadFile()
{
  this->m_filename = c_filename;
```

```
}

// Constructor with a custom filename provided by the user.
ReadFile::ReadFile(string filename)
{
  this->m_filename = filename;
}

// Read the graph data from the file using the specified
   filename.
void ReadFile::readFile()
{
  // Open the graphML file for reading.
  graphMlFile.open(m_filename);

  // Check if the file was successfully opened.
  if (!graphMlFile)
  {
    cerr << "Fehler beim Öffnen von " << m_filename << "!" <<
       endl;
    throw;
  }

  // Read the file line by line and process the data.
  while (getline(graphMlFile, m_currentLine))
  {
    // If the line contains "<node", read node data from the
       file.
    if (m_currentLine.find("<node") != string::npos)
    {
      readNodeDataFromFile();
      m_nodes[m_currentNode.m_id] = m_currentNode;
    }
    // If the line contains "<edge", read edge data from the
       file.
    else if (m_currentLine.find("<edge", 0) != string::npos)
    {
      readEdgeDataFromFile();
    }
    // Otherwise, continue to the next line.
    else
    {
```

```
      continue;
    }
  }
}

// Read node data from the file and store it in the current
   node.
void ReadFile::readNodeDataFromFile()
{
  // Continue reading until reaching the end of the current
     node data.
  do
  {
    // If the line contains "<node", extract the node ID.
    if (m_currentLine.find("<node", 0) != string::npos)
    {
      string searchString = "id=\"";
      string endString = "\">";

      size_t startPos = m_currentLine.find(searchString) +
         searchString.length();
      size_t endPos = m_currentLine.find(endString);
      string value = m_currentLine.substr(startPos, endPos -
         startPos);

      m_currentNode.m_id = stoi(value);
    }
    // If the line contains "<data key", extract the x and y
       values.
    else if (m_currentLine.find("<data key", 0) != string::npos)
    {
      // Get the x value.
      string searchString = "key=\"d0\">";
      string endString = "</data>";

      size_t startPos = m_currentLine.find(searchString) +
         searchString.length();
      size_t endPos = m_currentLine.find(endString);

      string value = m_currentLine.substr(startPos, endPos -
         startPos);
      m_currentNode.m_x = stof(value);
```

```
      // Get the y value from the next line.
      getline(graphMlFile, m_currentLine);

      searchString = "key=\"d1\">";

      startPos = m_currentLine.find(searchString) +
         searchString.length();
      endPos = m_currentLine.find(endString);

      value = m_currentLine.substr(startPos, endPos - startPos);
      m_currentNode.m_y = stof(value);
    }
    // If the line contains "</node>", the node data reading is
       complete.
    else if (m_currentLine.find("</node>", 0) != string::npos)
    {
      break;
    }
  } while (getline(graphMlFile, m_currentLine));
}

// Read edge data from the file and update connectedNodes of
   nodes accordingly.
void ReadFile::readEdgeDataFromFile()
{
  // Continue reading until reaching the end of the current
     edge data.
  do
  {
    // If the line contains "<edge", extract the source and
       target node IDs.
    if (m_currentLine.find("<edge", 0) != string::npos)
    {
      string searchString = "source=\"";
      string endString = "\" target=\"";

      size_t startPos = m_currentLine.find(searchString) +
         searchString.length();
      size_t endPos = m_currentLine.find(endString);
      string value = m_currentLine.substr(startPos, endPos -
         startPos);
```

```
      int src = stoi(value);

      searchString = "target=\"";
      endString = "\">";

      startPos = m_currentLine.find(searchString) +
         searchString.length();
      endPos = m_currentLine.find(endString);
      value = m_currentLine.substr(startPos, endPos - startPos);

      // Update the connectedNodes list for the source node.
      m_nodes[src].connectedNodes.push_back(stoi(value));
    }
    // If the line contains "</edge>", the edge data reading is
       complete.
    else if (m_currentLine.find("</edge>", 0) != string::npos)
    {
      break;
    }
  } while (getline(graphMlFile, m_currentLine));
}

// Set a new filename for reading graph data.
void ReadFile::setFilename(string filename)
{
  this->m_filename = filename;
}

// Get the map of nodes read from the file.
unordered_map<uint16_t, NodeData> ReadFile::getNodes()
{
  return this->m_nodes;
}

// Export the node data to an inline file (not implemented).
void ReadFile::exportMapToInl()
{
  // Not implemented yet.
}
```

A.1.3. unittest_readFile.cpp

```
#include <gtest/gtest.h>
#include "../car_readFile/car_readFile.hpp"

class carReadFile_Fixture : public ::testing::Test, ReadFile
{
  protected:
  ReadFile m_ReadFile;

  vector<NodeData> m_testNodes;
};

TEST_F(carReadFile_Fixture, readTest1)
{
  // set test Variables
  NodeData testData;
  testData.m_id = 1;
  testData.m_x = stof("2.22");
  testData.m_y = stof("2.8");
  testData.connectedNodes.push_back(3);
  m_testNodes.push_back(testData);

  m_ReadFile.setFilename("unittest_UselessTest-Graph.graphml");
  EXPECT_NO_THROW(m_ReadFile.readFile());

  unordered_map<uint16_t, NodeData> actualNodes =
     m_ReadFile.getNodes();

  EXPECT_EQ(actualNodes[1U].m_id, testData.m_id);
  EXPECT_EQ(actualNodes[1U].m_x, testData.m_x);
  EXPECT_EQ(actualNodes[1U].m_y, testData.m_y);
  EXPECT_EQ(actualNodes[1U].connectedNodes[0],
     testData.connectedNodes[0]);
}

TEST_F(carReadFile_Fixture, readTest2)
{

  // set test Variables
  unordered_map<uint16_t, NodeData> testNodes;

  testNodes[1U] = NodeData(1, stof("2.22"), stof("2.8"), {3,
```

```
    4});
  testNodes[2U] = NodeData(2, stof("2.22"), stof("3.17"), {3});
  testNodes[3U] = NodeData(3, stof("2.79"), stof("3.74"), {});
  testNodes[4U] = NodeData(4, stof("3.16"), stof("3.74"), {1});

  m_ReadFile.setFilename("unittest_Test-Graph.graphml");
  EXPECT_NO_THROW(m_ReadFile.readFile());

  unordered_map<uint16_t, NodeData> actualNodes =
      m_ReadFile.getNodes();

  for (uint16_t cnt{1U}; cnt < 4U; ++cnt)
  {
    EXPECT_EQ(actualNodes[cnt].m_id, testNodes[cnt].m_id);
    EXPECT_EQ(actualNodes[cnt].m_x, testNodes[cnt].m_x);
    EXPECT_EQ(actualNodes[cnt].m_y, testNodes[cnt].m_y);

    for (uint16_t i{0}; i <
        actualNodes[cnt].connectedNodes.size(); ++i)
    {
      EXPECT_EQ(actualNodes[cnt].connectedNodes[i],
          testNodes[cnt].connectedNodes[i]);
    }
  }
}
```

A.2. Code des implementierten Algorithmus

Hier finden Sie den vollständigen Quellcode der Implementierung des A*-Algorithmus in der Programmiersprache C++. Der Quellcode enthält Kommentare und Erklärungen, die den Algorithmus und seine Funktionsweise veranschaulichen. Dies ermöglicht es Ihnen, den Algorithmus zu studieren, zu analysieren und bei Bedarf anzupassen oder zu erweitern.

A.2.1. car_Algorithm.hpp

```
#ifndef CARALGORITHM_HPP
```

```
#define CARALGORITHM_HPP

#include <vector>
#include <queue>
#include <unordered_map>
#include <cmath>

#include "../car_readFile/car_readFile.hpp"

using namespace std;

// Tolerance value for comparing floating-point numbers.
const float toleranz = 0.03f;

// Represents a node used in the A* algorithm.
struct Node
{
  int id;          // The ID of the node in the graph.
  float fScore;    // The f-score value used in the A* algorithm
     for node comparison.
  float gScore;    // The g-score value used in the A* algorithm
     for tracking the path cost.

  // Custom comparison operator used for priority queue
     ordering.
  bool operator<(const Node& other) const { return fScore >
     other.fScore; }
};

// The A* algorithm class that inherits from the ReadFile class.
class AStarAlgorithm : ReadFile
{
  public:
  // Executes the A* algorithm to find the shortest path
     between two nodes.
  void aStarAlgorithm(int startId, int goalId);

  // Shortens straight road segments by removing intermediate
     nodes with similar positions.
  void shortenStraightRoad();

  // Returns the path found by the A* algorithm.
```

```
  vector<uint16_t> getPath();

  // Returns a reference to the ReadFile object used for
     reading input data.
  ReadFile& getReadFile();

  private:
  ReadFile m_readFile;              // Object to read input data
     from a file.
  vector<uint16_t> m_path;          // The final path found by
     the A* algorithm.

  // Calculates the heuristic value (estimated cost) between
     two nodes A and B.
  // This heuristic is used to guide the A* algorithm towards
     the goal efficiently.
  float heuristic(NodeData& nodeA, NodeData& nodeB);

  // Checks if the x-coordinate of two points is within the
     tolerance range.
  bool tolerancX(float currX, float prevX);

  // Checks if the y-coordinate of two points is within the
     tolerance range.
  bool tolerancY(float currY, float prevY);
};

#endif
```

A.2.2. car_Algorithm.cpp

```
#include "car_Algorithm.hpp"

vector<uint16_t> AStarAlgorithm::getPath()
{
  // Return the final path found by the A* algorithm.
  return this->m_path;
}

ReadFile& AStarAlgorithm::getReadFile()
{
```

```
  // Return a reference to the ReadFile object used for reading
     input data.
  return this->m_readFile;
}

// Calculate the heuristic value (estimated cost) between two
   nodes A and B.
// This heuristic is used to guide the A* algorithm towards the
   goal efficiently.
float AStarAlgorithm::heuristic(NodeData& nodeA, NodeData&
   nodeB)
{
  return sqrt(pow(nodeA.m_x - nodeB.m_x, 2) + pow(nodeA.m_y -
     nodeB.m_y, 2));
}

// Check if the x-coordinate of two points is within the
   tolerance range.
bool AStarAlgorithm::tolerancX(float currX, float prevX)
{
  return (((currX - toleranz) <= prevX) && (prevX <= (currX +
     toleranz)));
}

// Check if the y-coordinate of two points is within the
   tolerance range.
bool AStarAlgorithm::tolerancY(float currY, float prevY)
{
  return (((currY - toleranz) <= prevY) && (prevY <= (currY +
     toleranz)));
}

void AStarAlgorithm::aStarAlgorithm(int startId, int goalId)
{
  // Create two hash maps to store the gScore and fScore for
     each node.
  unordered_map<uint16_t, float> gScores;
  unordered_map<uint16_t, uint16_t> cameFrom;
  priority_queue<Node> openSet;

  gScores[startId] = 0.0f;
  openSet.push({ startId,
```

```
    heuristic(m_readFile.getNodes()[startId],
    m_readFile.getNodes()[goalId]), 0.0f });

while (!openSet.empty())
{
  Node current = openSet.top();
  if (current.id == goalId)
  {
    // Reconstruct the path by backtracking from the goal
       node to the start node.
    int node = current.id;
    m_path.push_back(node);
    while (cameFrom.find(node) != cameFrom.end())
    {
      node = cameFrom[node];
      m_path.push_back(node);
    }
    // Reverse the path to get the correct order.
    reverse(m_path.begin(), m_path.end());
    // Path found, exit the loop.
    break;
  }

  openSet.pop();
  vector<int> currentIdConnectedNodes =
     m_readFile.getNodes()[current.id].connectedNodes;
  for (const auto& neighborId : currentIdConnectedNodes)
  {
    float tentativeGScore = gScores[current.id] +
       heuristic(m_readFile.getNodes()[current.id],
       m_readFile.getNodes()[neighborId]);
    if (gScores.find(neighborId) == gScores.end() ||
       tentativeGScore < gScores[neighborId])
    {
      gScores[neighborId] = tentativeGScore;
      float fScore = tentativeGScore +
         heuristic(m_readFile.getNodes()[neighborId],
         m_readFile.getNodes()[goalId]);
      openSet.push({ neighborId, fScore, tentativeGScore });
      cameFrom[neighborId] = current.id;
    }
  }
```

```
  }
}

void AStarAlgorithm::shortenStraightRoad()
{
  // Remove intermediate nodes in the path that are within the
     tolerance of straight road segments.
  for(uint16_t i{1U}; i < (m_path.size() - 1); ++i)
  {
    if(tolerancX(m_readFile.getNodes()[m_path.at(i)].m_x,
       m_readFile.getNodes()[m_path.at(i + 1)].m_x)
    || tolerancY(m_readFile.getNodes()[m_path.at(i)].m_y,
       m_readFile.getNodes()[m_path.at(i + 1)].m_y))
    {
      m_path.erase(m_path.begin() + i);
    }
  }
}
```

A.2.3. unittest_Algorithm.cpp

```
#include <gtest/gtest.h>
#include "../car_algorithm/car_Algorithm.hpp"

class carAlgorithm_Fixture : public ::testing::Test
{
  protected:
  AStarAlgorithm m_AStarAlgorithm;
};

TEST_F(carAlgorithm_Fixture, simpleAStarTest)
{
  vector<uint16_t> expectedPath = {115, 116, 117, 118};

  m_AStarAlgorithm.getReadFile().setFilename("Test_track.graphml");
  m_AStarAlgorithm.getReadFile().readFile();

  m_AStarAlgorithm.aStarAlgorithm(115, 118);

  vector<uint16_t> actualPath = m_AStarAlgorithm.getPath();
```

```
  if (actualPath.empty())
  {
    ASSERT_FALSE(true) << "No Path was found";
  }
  for (uint16_t i{0}; i < actualPath.size(); ++i)
  {
    EXPECT_EQ(expectedPath.at(i), actualPath.at(i));
  }
}

TEST_F(carAlgorithm_Fixture, simpleAStarTest2)
{
  vector<uint16_t> expectedPath = {57, 58, 59, 2, 9, 7, 74, 75,
      16, 20, 17};

  m_AStarAlgorithm.getReadFile().setFilename("Test_track.graphml");
  m_AStarAlgorithm.getReadFile().readFile();

  m_AStarAlgorithm.aStarAlgorithm(57, 17);

  vector<uint16_t> actualPath = m_AStarAlgorithm.getPath();

  if (actualPath.empty())
  {
    ASSERT_FALSE(true) << "No Path was found";
  }
  for (uint16_t i{0}; i < actualPath.size(); ++i)
  {
    EXPECT_EQ(expectedPath.at(i), actualPath.at(i));
  }
}

TEST_F(carAlgorithm_Fixture, crossParking)
{
  vector<uint16_t> expectedPath = {86, 77, 82, 80, 92, 93, 94,
      68, 73, 71, 124, 125, 126, 127, 128, 59, 64, 62, 148, 149,
      150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160,
      161, 162, 163};

  m_AStarAlgorithm.getReadFile().setFilename("Competition_track.graphml");
  m_AStarAlgorithm.getReadFile().readFile();
```

```
  m_AStarAlgorithm.aStarAlgorithm(86, 163);

  vector<uint16_t> actualPath = m_AStarAlgorithm.getPath();

  if (actualPath.empty())
  {
    ASSERT_FALSE(true) << "No Path was found";
  }
  for (uint16_t i{0}; i < actualPath.size(); ++i)
  {
    EXPECT_EQ(expectedPath.at(i), actualPath.at(i));
  }
}
```