Diplomarbeit zum Thema

Wegplanung und Fahrt autonomer mobiler Systeme in geometrischen Karten am Beispiel von Zellzerlegungsverfahren

von
Kai Niekammer

zur Erlangung des akademischen Grades
Diplom-Informatiker (FH)

Vorgelegt dem
Fachbereich Informatik und Medien der
Fachhochschule Brandenburg

Erstprüfer: Dipl.-Inform. Ingo Boersch
Zweitprüfer: Prof. Dr.-Ing. Jochen Heinsohn

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Brandenburg, den 1.07.2008

Kai Niekammer

Zusammenfassung

Autonome Mobile Systeme gewinnen immer mehr an Bedeutung, da sie bereits in vielen Gebieten eingesetzt werden um dem Menschen Arbeit abzunehmen oder ihn zu unterstützen. Roboter werden auf fremde Planeten geschickt um deren Oberflächenbeschaffenheit zu erforschen. Daher gewinnen Methoden der künstlichen Intelligenz in Anwendung auf Robotern immer mehr Bedeutung um diese selbständiger agieren und navigieren zu lassen. In Zusammenhang mit selbständiger Navigation ist vor allem das Thema Pfadplanung für autonome mobile Systeme äußerst interessant. Diese Arbeit stellt verschiedene Pfadplanungsverfahren vor und beleuchtet dabei die Vertikale Zellzerlegung genauer.

Abstract

Autonomous mobile systems become more important, because they were applied in many regions to take off work from humans or to support them. Robots were sent to distant planets to explore their surface area. That is why methods of artificial intelligence become more important in appliance to robots, to let them operate and navigate more autonomous. In relation to autonomous navigation, the question of path planning is very interesting for autonomous mobile robots. This paper introduces several methods of path planning and highlights the vertical cell decomposition.

„Sorgfältige Planung ist der Schlüssel zu einem sicheren und zügigen Reiseverlauf“

Odysseus

Inhaltsverzeichnis ..... Seite

1.

EINFÜHRUNG

1.1. Aufgabenstellung

Zielsetzung der Arbeit ist die selbständige Navigation eines simulierten Roboters vom Typ Pioneer in einer vom Anwender erstellten Karte durch Zuweisung von Zielpunkten.

Schwerpunkt der Arbeit ist die Diskussion von Verfahren zur Pfadplanung, sowie die prototypische Umsetzung einer Methode zur exakten Zerlegung des Freiraumes mit anschließender Pfadsuche im resultierenden Graphen. Besonderer Wert wird hierbei auf die Darstellung der Verarbeitungsschritte von der internen Karte bis zur Folge der Wegpunkte gelegt. Die gefundenen Wegpunkte sind unter der Annahme fehlerfreier Aktorik und Sensorik in der Simulation abzufahren, eine eigene Registrierung ist nicht notwendig.

Als Systemumgebung soll Saphira mit dem zugehörigen Simulator genutzt werden. Notwendig ist dafür eine Anbindung der Planungs- und Steuerungskomponente an bzw. in Saphira. Die Erstellung der internen Karte des Roboters und der korrespondierenden simulierten Umwelt kann durch eine Applikation unterstützt werden. Die Funktionsfähigkeit des Gesamtsystems ist durch geeignete Experimente nachzuweisen.

1.2. Analyse der Aufgabe

Schwerpunkt dieser Arbeit ist die autonome Navigation eines simulierten Roboterservers, in diesem Fall vom Typ Pioneer PC2E, von einem Start zu einem Zielpunkt innerhalb einer geometrischen Karte. Ausgangspunkt ist eine vom Benutzer erstellte geometrische Karte, bestehend aus Wänden und Hindernissen. Es soll ein Programm entwickelt werden, dass mit Hilfe geeigneter Methoden, Wegpunkte und befahrbare Pfade berechnet. Danach soll der Simulator dynamisch vom Start- zum Zielpunkt navigiert werden.

1.3. Zu untersuchende Problembereiche

1.3.1. Pfadplanung

Ziel ist es mit einem geeigneten Pfadplanungsverfahren, innerhalb einer geometrischen Karte, einen Pfad durch den Freiraum zu finden. Um einen Pfad von einem Start- zu einem Zielpunkt, innerhalb einer Karte, zu finden bedarf es Methoden und Algorithmen die diesen Pfad ermitteln. Diese Methoden gehören in das Gebiet der Pfadplanung. Hierbei gilt es einen geeigneten Algorithmus zu finden, da nicht jeder Algorithmus für jede Aufgabenstellung geeignet ist. Ergebnis eines Wegplanungsalgorithmus ist ein Graph aus befahrbaren Pfaden. Jedoch erzeugt nicht jeder Algorithmus die zur Navigation benötigten Wegpunkte, sodass diese in einigen Fällen noch zusätzlich berechnet werden müssen.

1.3.2. Kommunikation

Mit der Zielsetzung eine Applikation zu entwickeln, die einen simulierten Roboter dynamisch navigiert, geht die Frage der Kommunikation mit der Entwicklungsumgebung Saphira einher. Hierzu bieten sich einige Möglichkeiten an. Auf direktem Weg bietet Saphira keine Möglichkeit mit anderen Applikationen zu kommunizieren, jedoch lassen sich dynamische Bibliotheken in

Saphira einbinden. Somit ist es möglich, mithilfe einer zu entwickelnden dynamischen Bibliothek verschiedene Formen der Interprozess-Kommunikation bereitzustellen.

Folgende Methoden bieten sich für die Kommunikation an:

Shared Memory

Hierbei erhalten zwei unterschiedliche Prozesse Zugriff auf einen gemeinsamen Speicherbereich und können somit Daten austauschen.

Message Queues

Mithilfe von Message Queues können lokale Prozesse Nachrichten austauschen. Nachrichten bestehen aus dem Nachrichten-Kopf (header), der aus einer positiven Integer-Zahl besteht, welche den Typ der Nachricht beschreibt, und einem Nachrichten-Text (body), der aus einem Character-Array vorgegebener Größe besteht.

Sockets

Sockets stellen eine vollduplexfähige Variante zur Interprozess-Kommunikation dar. Sockets basieren auf Protokollen wie TCP und UDP der Transportschicht. Sie eignen sich vor allem zur Kommunikation von Applikationen in Netzwerken.

1.3.3. Globale Navigation

Bei der globalen Navigation können einige kritische Situationen auftreten. Zum einen können bewegliche Hindernisse, sowie bewegliche Objekte auftreten und die Objektinterpretation des Roboters durcheinander bringen. Eine weitere Problematik birgt der Roboter selbst. Die Kinematik eines Roboters ist nicht so exakt wie es für exakte Navigationen nötig wäre. Zum einen wird die Odometrie bei größeren Entfernungen immer ungenauer und zum anderen liefern die Sonarsensoren nicht immer genaue bzw. falsche Ergebnisse. Die Sonare haben eine Reichweite von 10cm bis 5m, was bedeutet, dass Objekte in einer Entfernung über 5m nicht wahrgenommen werden können.

1.3.4. Fahrt

Für die Fahrt ist ein Pfad notwendig, der mit einem geeigneten Verfahren berechnet werden muss. Die eigentliche Fahrt des Pioneer-Roboters sowie auch mit einem simulierten Roboter ist zudem mit einigen Unsicherheiten behaftet. Ein großer Unsicherheitsfaktor ist die Abweichung der realen Position von der intern vermuteten Position des Roboters. Zwar verfügt der Roboter über eine Odometrie, jedoch ist die Bewegung des Roboters nicht exakt und führt daher zu Abweichungen der realen Position von der geschätzten Position.

Odometrie ist ein Mechanismus, der die Bewegungen des Roboters mit verfolgt, um danach die relative Position zu bestimmen.

Den Abgleich der gemessenen Daten mit der Karte und das damit verbundene Korrigieren des Roboters in der Karte nennt man Registrierung. Es gibt unterschiedliche Arten der Registrierung, die sich in diesem Projekt anbieten und die jeweils ihre Vor- und Nachteile haben. Zunächst sei erwähnt, dass bislang noch keine implementierte Registrierung für alle Artefakte in

Saphira existiert. Es gibt lediglich eine auf Korridore basierende Registrierung. Dafür müssen allerdings auch Korridore existieren.

1.3.5. Positionierung

Es werden zwei Arten von Positionierung unterschieden.

Bei der **absoluten Positionierung** kann die Position anhand von Koordinaten oder Landmarken exakt bestimmt werden. Natürliche Landmarken sind beispielsweise Bäume und Flüsse oder Türen und Wände innerhalb von Gebäuden.

Bei der **relativen Positionierung** wird die aktuelle Position durch Beobachtung der Bewegungsorgane und Berechnen der vergangenen Bewegungen bestimmt. Das geschieht mithilfe von Odometrie oder mit inertialen Navigationssystemen.

Bei der Odometrie wird die eigene Bewegung beobachtet, zum Beispiel mit Encodern, die die Antriebsräder beobachten. Somit kann die neue Position mithilfe der beobachteten Bewegung bestimmt werden.

Das mechanische Gyroskop, oder auch Kreiselkompass, ist ein inertiales Navigationssystem(INS). Diese Art wird auch Trägheitsnavigation genannt, weil solche Geräte die Gesetze der Masseträgheit verwenden um ihre Position zu berechnen.

1.3.6. Karten

Generell werden zwei Arten von Karten unterschieden.
Geographische Karten sind Gittermodelle mit euklidischen Koordinaten. Dabei erhält jede Kachel dieses Netzes Werte. Diese können ganz unterschiedlich sein. Angefangen von Koordinaten bis über Gewißheitsmaße, dass sich in einer Kachel ein Objekt befindet. Menschen denken und handeln nicht in geographischen Karten, denn für einen Menschen ist es nicht unbedingt notwendig seine geometrische Position auf der Erde zu kennen, um sich zu einem bestimmten Ort zu bewegen. Dazu nutz er topologische Karten.
Eine **topologische Karte** ist ein Graph, dessen Knoten entscheidungsbestimmende Situationen oder Orte und dessen Kanten Wege bzw. Aktionen zwischen diesen Orten sind.

1.3.7. Registrierung

Die Registrierung ist ein entscheidender Faktor, der großen Einfluss auf Erfolg oder Misserfolg des Unternehmens autonome Navigation entscheidet. Momentan existiert kein implementierter Registrierungsprozess, da jedoch mit dem Simulator gefahren werden soll, ist eine Registrierung nicht zwingend notwendig. Bei einer Navigation mit einem realen Roboter wäre eine Registrierung unabdingbar, weil nicht gewährleistet werden kann, dass der Roboter möglicherweise mit Hindernissen kollidiert. Das wäre eine sehr kostspielige Angelegenheit. Da es jedoch hauptsächlich um die Navigation mit einem simulierten Roboterserver geht, ist eine Registrierung nicht unbedingt notwendig.

1.4. Versuchsumgebung

Zur Umgebung, in der die entwickelte Applikation später getestet werden soll, gehört zunächst der Saphira-Client, der für die Verbindung zum Pioneer-Roboter sorgt und eine Umgebung darstellt, die viele Steuerungsbefehle für den Roboter beinhaltet. Es wird hauptsächlich mit dem Pioneer-Simulator gestestet, da hierbei verschiedene Welten sowie Situationen gestestet werden können. Mit dem Pioneer Roboter kann ja nur eingeschränkt getestet werden, bzw. stehen nur wenige Räumlichkeiten für Tests zur Verfügung. Im Simulator können Zufallswelten geladen und virtuell befahren werden. Dies bietet umfangreichere Möglichkeiten, die entwickelten Methoden zur Navigation ausgiebig zu erproben und zu verfeinern.

1.4.1. Pioneer P2CE

Der Pioneer P2CE wird von der US-Firma Mobile Robots produziert.

Abb. 1 : Pioneer P2CE [LePy]

Zur Bewegung verfügt das Fahrzeug über zwei Antriebsräder und ein Stützrad.

Die Sensorik des Pioneer P2CE besteht aus 8 Ultraschallsender/-emfänger sowie einer angetriebenen Kamera mit Bildvorverarbeitung. Mit dem Greifer können Objekte aufgehoben und transportiert werden.

Als Odometrie besitzt er einen Encoder auf der Antriebswelle der Motoren.

Desweiteren verfügt er über einen elektronischen Kompaß V2X.

Die eigentliche Steuerung wird extern über den Saphira Client ausgeführt.

Die Kommunikation erfolgt mittels Bluetooth. Der Pioneer Roboter hat eine Länge von 44cm, eine Breite von 33cm sowie eine Höhe von 22cm. Er wiegt 9kg und kann eine maximale Last von 20kg befördern. Das Chassis besteht aus CNC Aluminium.

Der Pioneer P2CE zählt zusammen mit der 3.Generation zu einer neuen Generation intelligenter mobiler Roboter. Schon das Vorgängersystem Pioneer 1 hat sich international bewährt und wurde an vielen Hochschulen für Lehre und Forschung eingesetzt. Roboter dieser Art gehören zu den autonomen mobilen Systemen.

Autonome Systeme sind Systeme, die sich nicht vollständig vorhersagbar verhalten. Sie bilden ihre eigenen Methoden und Strategien um Problemsituationen zu lösen. Man kann sie indirekt über Interaktion steuern.

Kognition ist die Fähigkeit „Wissen“ selbständig anzuwenden um somit sinnvolle Verhaltensentscheidungen zu treffen.

Das **Verhalten** bestimmt, zum Beispiel mit Pfadplanungsverfahren, wohin sich der Roboter bewegen soll.

Die **Bewegung** führt dann die Navigation aus, muss beispielsweise Hindernisse umfahren oder Korridoren folgen.

2.

SYSTEMUMGEBUNG

2.1. Saphira

Saphira ist eine Entwicklungsumgebung, mit der man triviale sowie komplexe Steuerungsprogramme für autonome mobile Systeme entwickeln kann. Dies erfolgt hauptsächlich mit Colbert, einer C-ähnlichen Programmiersprache, mit der es möglich ist Activities bestehend aus simplen Schleifen, Entscheidungsabfragen und GOTO-Befehlen zu formulieren. Es ist möglich vorher definierte Behaviors mit einer bestimmten Priorität zu starten.

Behaviors sind low-level Steuerungsprogramme, wie zum Beispiel einer Wand folgen oder durch eine Tür fahren. Dabei verwendet Saphira Fuzzy-Kontroll-Regeln um Behaviors zu implementieren. Behaviors können parallel ablaufen um mehrere Ziele zu verfolgen, wie zum Beispiel das Folgen eines Korridors und der Verhinderung von Kollisionen. Dabei können Behaviors zusätzlich Prioritäten zugeordnet werden, um die Hierarchie festzulegen. Wichtige Behaviors, wie das Verhindern von Kollisionen, sollten eine hohe Priorität erhalten.

Activities vereinen mehrere Behaviors und auch Activities zu größeren Programmen. Damit lassen sich globale Aufgaben lösen, wie zum Beispiel das Navigieren durch ein Labyrinth.

Saphira bietet die Möglichkeit, dynamische Bibliotheken in den integrierten Evaluator zu laden, um den Sprachumfang von Saphira zu erweitern und auch komplexere Programme entwickeln zu können.

Man kann den Roboter auch interaktiv mit Hilfe der Pfeiltasten steuern oder mit direkten Steuerungsbefehlen.

Beispiel:

move(1000)

-Bewegt den Roboter 1000mm vorwärts

turn(90)

-Dreht den Roboter 90° im Uhrzeigersinn

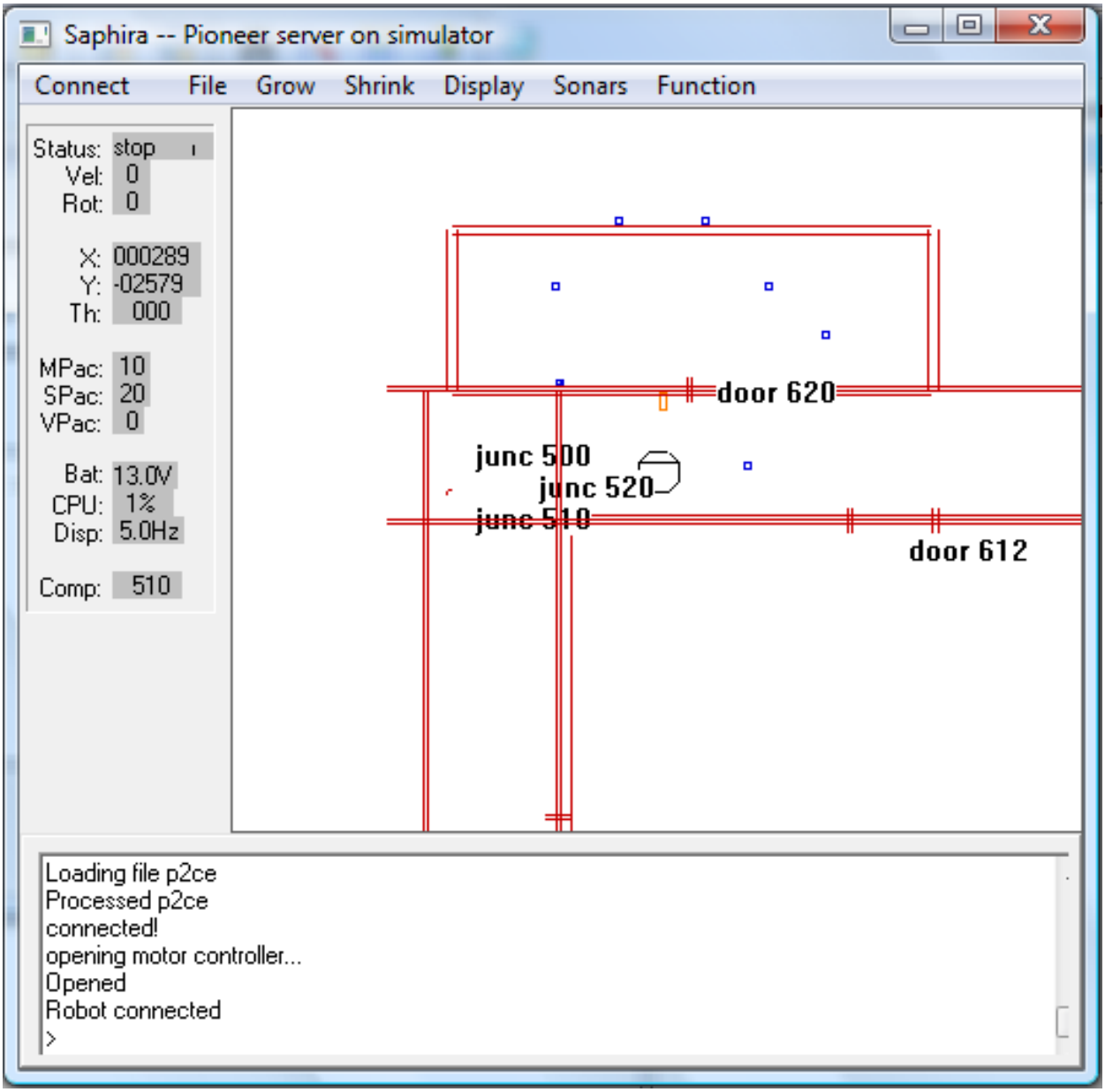


Abb. 2 : Screenshot der Saphira Entwicklungsumgebung

2.2. Der Simulator

Beim Simulator handelt es sich um einen Roboter-Server, der wie ein realer Roboter mit Saphira verbunden werden kann und das Verhalten eines realen Roboters imitiert. Dabei werden Bewegungen, Drehungen sowie Sensorwerte simuliert. Selbst Fehlverhalten und Abweichungen kann der Simulator imitieren. Man hat die Möglichkeit einfache „Welten“ zu laden und somit die Funktionalität des entwickelten Programms zu testen, ohne dafür einen realen Roboter zu benötigen. Da oftmals nicht die Möglichkeit besteht mit einem realen Roboter zu experimentieren, weil es sich zum Beispiel um eine nicht reale Welt handelt, in der gefahren werden soll, ist der Simulator eine hervorragende Lösung um auch Verhalten in solchen Welten zu simulieren. Der große Vorteil des Simulators ist, dass man Programmteile einzeln ohne realen Roboter testen kann. Der Nachteil des Simulators ist, dass er keine dynamischen Hindernisse simulie-

ren kann und nur eine simple 2-dimensionale Darstellung erlaubt. Somit sind keine 3 dimensionalen Welten möglich.

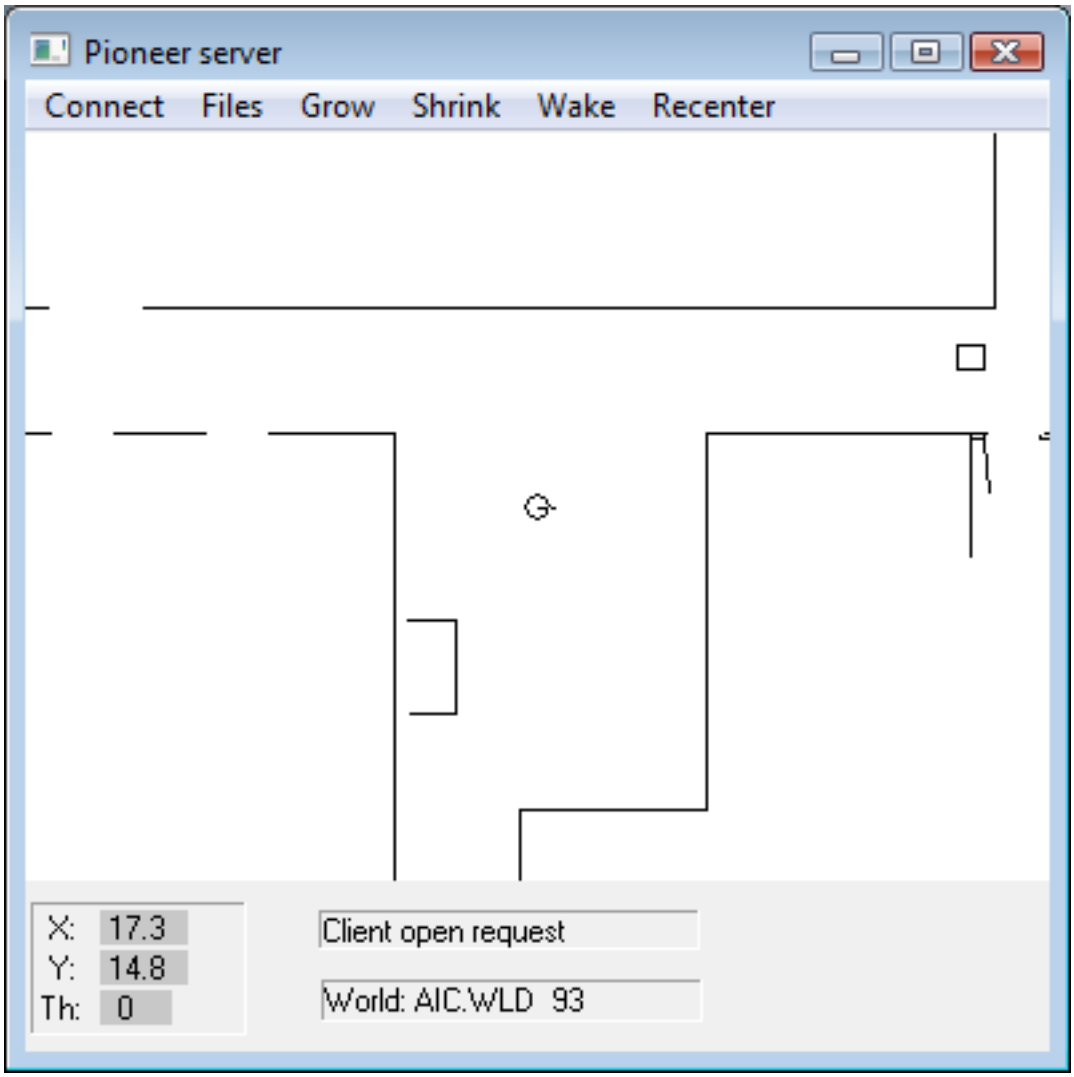


Abb. 3 : Pioneer Simulator

2.3. Karten in Saphira

Saphira arbeitet mit zwei geometrischen Karten. Es gibt zunächst den Local Map Space (LPS). Er ist eine egozentrische Weltsicht in der sich der Roboter im Zentrum befindet. Damit wird das Umfahren von Hindernissen vereinfacht. Sie dient hauptsächlich der lokalen Navigation. Zum Beispiel zum Erreichen eines nahen Zielpunktes oder eine Türdurchfahrt.

Der Global Map Space (GMS) dient der Globalen Navigation, der Navigation im Gesamten, beispielsweise der Fahrt durch ein Labyrinth mit Vermeidung von Umwegen und Sackgassen.

Karten für Saphira liegen im .map Format vor. Hierbei sind alle Artefakte, also Wände, Türen, Korridore usw. in einer Datei aufgelistet. Die folgende Abbildung zeigt einen Ausschnitt aus einer Map Datei.

```
;;
;; Map of a small portion of the SRI Artificial Intelligence Center
;;
;;                X     Y   Th  Length Width
CORRIDOR (1)     2000, 3000, 0,  3500,  800
CORRIDOR (2)     1000, 2000, 90, 6000, 1000
DOOR (3)         3000, 2600, 90,       1000
DOOR (4)         1500, 1000, 180,      1000
JUNCTION (5)     1500, 3000, 0,         800
WALL (6)         1000, 4000, 0,  1000
WALL (8)          800, 3500, 90,  400
WALL              800, 4500, 90,  400
```

Abb. 4 : Auschnitt aus einer .map Datei [SSMa]

Alle Artefakte werden untereinander gespeichert. Optional können diese nummeriert werden. Mit den Werten X und Y wird der Mittelpunkt des Artefaktes angegeben. Th ist der Winkel in dem das Artefakt geneigt ist. Length und Width entsprechen jeweils der Länge und Breite des Artefaktes. Wichtig ist, dass alle Werte in Millimetern angegeben werden. Der Roboter befindet sich im Koordinatenursprung. Unter anderem existieren folgende Artefakte in Saphira:

Wall

Eine Wand wird in Saphira folgendermaßen definiert. Sie hat einen Mittelpunkt, eine Richtung und eine Länge. Sie werden als Landmarken zur Positionierung und Navigation benutzt. Die meisten Roboter sind im Grunde nicht selbständig in der Lage Wände zu erkennen. Der Pioneer Roboter kann beispielsweise lediglich Hypothesen auf der Grundlage seiner Sonarsensoren über das Vorhandensein von Wänden erstellen. Diese werden in einer Datenstruktur gespeichert und können ausgelesen werden.

Corridor

Ein Korridor besteht aus zwei parallel verlaufenden Wänden. Ein Korridor besteht aus einer Breite, einer Länge und einem Winkel. Ein Punkt beschreibt den Mittelpunkt des Korridors.

Door

Eine Tür beschreibt eine Durchfahrt in einer Wand. Dazu wird der Mittelpunkt der Tür, sowie der Winkel und die Breite angegeben.

Junction

Ein Junction-Artefakt beschreibt den Schnittpunkt zweier Korridore, sodass von einem in den anderen gefahren werden kann und die Wände, an denen sich die Korridore schneiden, wegfallen.

Lane

Ein Lane-Artefakt ähnelt dem Corridor bis auf die Tatsache, dass es nicht durch Wände begrenzt ist.

Point

Ein Point-Artefakt stellt einen Punkt auf der Karte, beispielsweise einen Wegpunkt dar.

Der Pioneer Simulator hingegen arbeite mit einem anderen Format. Dem .wld Format, welches das simulierte Abbild unserer oder irgendeiner Umgebung darstellt. In der folgenden Abbildung wird ein kleiner Ausschnitt aus einer World Datei dargestellt.

```
;;; Fragment of a simple world

width 38000
height 30000

  0 0 0 30000                    ; World frontiers
  0 0 38000 0
  38000 30000 0 30000
  38000 30000 38000 0

push 10000 14000 0
```

Abb. 5 : Auschnitt aus einer .wld Datei [SSMa]

Zunächst kann man mit width und height die Breite und Höhe der Welt festlegen. Diese Angabe ist jedoch optional, wird sie nicht angegeben, wird die Welt so groß dargestellt wie sie durch die Ausmaße der Wände ist. Es werden hierbei keine Objekte oder Artifakte gespeichert sondern ausschließlich Wände. Die vier Werte umschreiben den Anfangs- und den Endpunkt einer Wand in x und y Koordinaten. Mit Position kann man die Anfangsposition des Roboters bestimmen. Dieser Ausdruck erwartet drei Werte, die Position in x und y Koordinaten und den Winkel des Roboters. Alle Längenangaben in der .wld und auch der .map Datei werden stets in mm angegeben.

Zusätzlich existieren zwei Konstrukte push und pop, mit denen der Ursprung relativ verschoben und gedreht werden kann. Das bedeutet, dass alle Angaben nach einem Push automatisch um die Position von Push verschoben und gedreht werden. Dies ermöglicht die Angabe von Wänden relativ zu einem anderen Ort. Mit Pop kehrt man wieder zum alten System zurück.

2.4. Parameterdatei

In der Parameterdatei sind die spezifischen Eigenschaften der verschiedenen Roboter abgespeichert. Es existiert eine Parameterdatei für jeden Robotertyp, bzw. wird für jeden Robotertyp eine individuelle Parameterdatei benötigt. Beim Benutzen von Saphira muss diese Parameterdatei geladen werden, um Saphira mitzuteilen welcher Robotertyp mit Saphira gesteuert werden soll. Das ist notwendig, damit Saphira jeden Roboter exakt steuern und dessen Sensorinformationen genau auswerten kann, da sich Ausmaße, Radstände und Sensorpositionen und –anzahl von Roboter zu Roboter unterscheiden. Diese Parameterdatei wird ebenso für den Pioneer Simulator verwendet, um auch hierbei das gewünschte Verhalten des zu simulierenden Roboters zu erreichen. Im folgenden Absatz ist eine Parameterdatei für einen Pioneer 2 CE Roboter zu sehen.

```
;; Parameters for the Pioneer 2 CE Mobile Robot
AngleConvFactor 0.001534 ; radians per angular unit (2PI/4096)
DistConvFactor 0.826 ; mm returned by P2
VelConvFactor 1.0 ; mm/sec returned by P2
RobotRadius 250.0 ; radius in mm
RobotDiagonal 120.0 ; half-height to diagonal of octagon
Holonomic 1 ; turns in own radius
MaxRVelocity 500.0 ; degrees per second
MaxVelocity 2200.0 ; mm per second
RangeConvFactor 0.268 ; sonar range returned in mm
;;
;; Robot class, subclass
;;
Class Pioneer
Subclass p2ce
SonarNum 8 ; 8 total sonars
;; # x y th
;;-------------------------
SonarUnit 0 115 130 90
SonarUnit 1 155 115 50
SonarUnit 2 190 80 30
SonarUnit 3 210 25 10
SonarUnit 4 210 -25 -10
SonarUnit 5 190 -80 -30
SonarUnit 6 155 -115 -50
SonarUnit 7 115 -130 -90
```

Mithilfe dieser Parameterdatei ist es zusätzlich noch möglich, Abweichungen zu parametrisieren. Mit den folgenden Parametern werden diese definiert.

EncodeJitter

Mit diesem Parameter lässt sich der Entfernungsfehler bestimmen.

AngleJitter

Abweichungen bei Drehungen lassen sich mit diesem Parameter definieren.

AngleDrift

Mit diesem Parameter lassen sich Drehabweichungen bei der Vorwärtsbewegung definieren. Setzt man diese drei Parameter auf 0 kann man somit die Abweichungen des Simulators ganz aufheben.

3.

PFADPLANUNG

Die Entwicklung in der Robotik geht immer mehr in die Richtung zu intelligenteren Robotern mit komplexen Navigationsaufgaben. Das bedeutet, dass sie sich in komplexeren und dynamischen Umgebungen bewegen sollen. Mit verschiedensten Sensoren werden Daten der Umwelt gesammelt und ausgewertet. Ziel aller dieser Aktivitäten ist es, einen befahrbaren Weg in den komplexen Umgebungen zu finden. Daher ist die Pfadplanung ein sehr wichtiger Bereich der Robotik um autonomen mobilen Systemen mehr Selbständigkeit zu verleihen.

Konfigurationsraum C

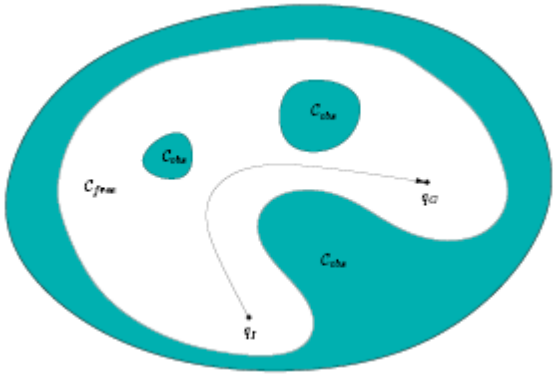


Abb. 6 : Der Konfigurationsraum C [LaSt]

Lavalle beschreibt den Konfigurationsraum, indem die Pfadplanung stattfindet wird in [LaSt] wie folgt:

Der gesamte Raum, indem sich Hindernisse sowie frei befahrbare Wege befinden, wird als Konfigurationsraum $C = R^N$ bezeichnet. Mit N wird die Dimension des Konfigurationsraums festgelegt, in diesem Projekt der 2-dimensionale Raum. Der gesamte Konfigurationsraum besteht aus freibefahrbaren Flächen sowie Hindernissen, wie Wände oder anderen Objekten. Daher wird der Konfigurationsraum in C_{free} und CB_i unterteilt.

C_{free} stellt den für den Roboter frei befahrbaren Raum. Jener Raum, welcher von Hindernissen belegt wird, nennt man den Hindernisraum CB_i und wird folgendermaßen definiert:

$$CB_i = \{q \in C \mid A(q) \cap B_i \neq 0\} \quad mit\ i \in [1, n]$$

A(q) bezeichnet jenen Raum, welcher der Roboter A bei einer bestimmten Konfiguration q einnimmt. B_i stellt jenen Raum dar, welcher das i-te Hindernis einnimmt und mit n ist die Anzahl der Hindernisse gemeint.

$$C = C_{free} \bigcup_{i \in [1,n]} CB_i$$

Jede beliebige Konfiguration q, die in C_{free} liegt, nennt man eine freie Konfiguration.

Mithilfe der Pfadplanung soll nun ein freier Pfad, das bedeutet, dass er sich in C_{free} befinden muss, zwischen einem Start und einem Zielpunkt gefunden werden.

$$r : [0,1] \rightarrow C_{free}$$

$r(0)$ stellt die Startkonfiguration q_{Start} dar und $r(1)$ repräsentiert die Endkonfiguration q_{Ziel}

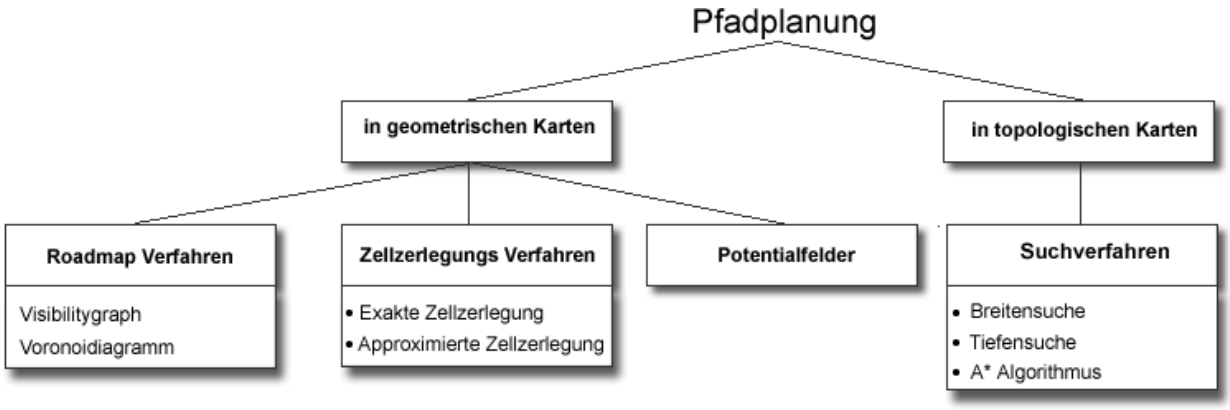


Abb. 7 : Übersicht Pfadplanungsverfahren

Unter Pfadplanung versteht man die Berechnung eines Pfades in einer Geographischen oder Topologischen Karte, von einem Start- zu einem Zielpunkt. In einer topologischen Karte genügt ein Suchverfahren im Graphen für die Wegplanung, da hier keine Wegpunkte berechnet werden müssen. Die größte Herausforderung besteht in der Wegplanung in geographischen Karten, da erst befahrbare Wege in der Karte gefunden werden müssen. Dafür gibt es zahlreiche Verfahren. Man unterscheidet hierbei grob Road Map – Verfahren, Zellzerlegungsverfahren und Potenzialfelder.

3.1. Kriterien zur Bewertung eines Pfadplaners

Es gibt verschiedene Ziele und somit unterschiedliche Anforderungen, die an einen Pfadplaner gestellt werden. Aufgrund folgender Kriterien lassen sich Pfadplaner je nach Anforderung bewerten.

Vollständigkeit

Ein Pfadplaner ist genau dann vollständig, wenn er immer einen Pfad von einem Start zum Zielpunkt findet, sofern ein Pfad existiert. Ist dies nicht der Fall, spricht man von einem unvollständigen Pfadplaner.

Robustheit

Nicht immer sind Positionen von Hindernissen und Objekten in der Karte genau. Ein robuster Pfadplaner muss mit solchen Ungenauigkeiten umgehen können.

Pfadlänge vs. Zeitoptimalität

Oft wird neben der Anforderung einen Pfad zu finden zusätzlich die Forderung nach einem optimalen Pfad gestellt. Beispielsweise soll ein Pfadplaner einen Pfad finden, der die geringste Pfadlänge besitzt oder die wenigste Zeit benötigt.

Abstand zu Hindernissen

Bei vielen Pfadplanungen ist nicht immer der kürzeste oder schnellste Pfad das Ziel, sondern das Finden des „sichersten“ Pfades, der den größten Abstand zu Hindernissen gewährleistet.

3.2. Road Map – Verfahren

Roadmap Verfahren werden angewandt um für eine gegebene Karte das Straßennetz zu berechnen. Die entstandene Straßenkarte beinhaltet die vom Pfadplanungs-Verfahren ermittelten Wege. Hierbei wird versucht Verbindungen im freien Raum mittels Kurven und Linien herzustellen. Ziel ist es den Start- und den Zielpunkt mittels eines Straßennetzes in Form eines Graphen zu verbinden. In diesem Graphen kann dann mit einem Suchalgorithmus ein geeigneter Weg gefunden werden.

3.2.1. Visibility Graph / Sichtbarkeitsgraph

Der Sichtbarkeitsgraph war eines der ersten Verfahren zur Pfadplanung und wurde von NJ Nilsson schon 1969 entdeckt. Hierbei werden alle Hinderniskanten paarweise verbunden. Zusätzlich werden noch alle Objektkanten und der Start- und Zielpunkt miteinander verbunden.
Danach wird in dem entstandenen Graph die Pfadsuche mittels eines Suchverfahrens durchgeführt.

Der Vorteil dieser Variante ist, dass ein vollständiger Graph entsteht und kein Pfad übersehen wird, jedoch steigt die Komplexität dieses Algorithmus exponentiell mit der Anzahl der Objekte bzw. Kanten. Daher ist der Sichtbarkeitsgraph nur sinnvoll für einfache Umgebungen, bei komplexeren Welten werden der Speicherbedarf und auch die Rechenlast bei diesem Verfahren zu hoch. Weiter wird der kürzeste Pfad gefunden, was aber zufolge hat, dass dieser sehr nah an Hindernissen vorbeiführt, diese eigentlich sogar berührt.

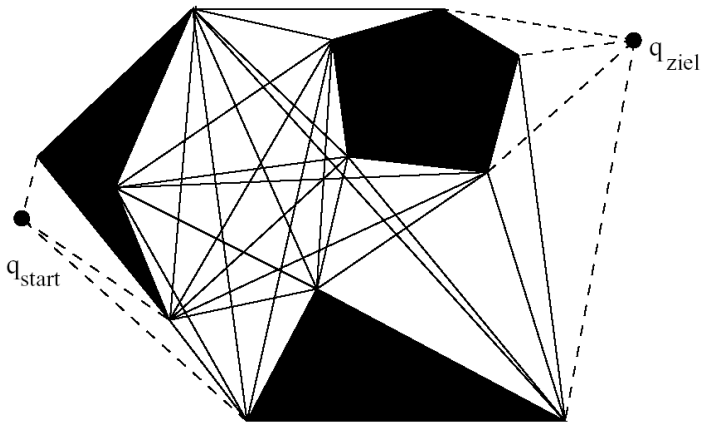


Abb. 8 : Sichtbarkeitsgraph [Deut]

Pseudocode:

```
Programm sichtbarkeit
    Für jeden Eckpunkt
        Für jeden Eckpunkt
            If Eckpunkte verbunden werden können ohne ein Hinderniss zu schneiden
                Füge Kante zum Visibility Graph hinzu
            else
                Continue
            endif
    return Visibility Graph
End Programm Sichtbarkeit
```

Nachteil des Sichtbarkeitsgraphen ist, dass die gefundenen Pfade nah an den Hindernissen vorbeiführen. So kann es schnell zu Situationen kommen, in denen der Roboter nicht weiter navigieren kann. Hierfür ist eine besondere Methode der Hindernisumfahrung notwendig.

3.2.2. Voronoi Diagramme

In [SuBO] wird auf die geschichtliche Entwicklung von Voronoi-Diagrammen eingegangen. Demnach tauchten Voronoi-Diagramme schon in frühester Vergangenheit auf. Schon 1644 nutzte Descartes Verfahren die den Voronoi-Diagrammen sehr ähnelten, um die Lage der Körper im Sonnensystem zu beschreiben. Der erste, der das Konzept von Voronoi Diagrammen studierte, war der deutsche Mathematiker Peter Gustav Lejeune Dirichlet. Er betrachtete dabei den Fall für 2- und 3-dimensionale Räume. Deshalb ist dieses Konzept auch unter Dirichlet Tessellation bekannt. Aber es ist hauptsächlich als Voronoi Diagramm bekannt, weil der russischen Mathematiker Georges Voronoi das Voronoi Diagramm gegen 1908 für n-dimensionale Fälle beschrieb. Seitdem finden Voronoi Diagramme in vielen Gebieten, wie zum Beispiel der Meteorologie und der Kartographie ihre Anwendung.

Ein Voronoidiagramm resultiert aus der Einteilung einer Punktmenge von n Punkten in n Regionen, wobei jede Voronoi Region genau die Punkte enthält die näher zum Punkt n liegen als zu irgendeinem anderen Punkt. Die Punkte die zu mehr als einem Punkt die geringste Entfernung besitzen, bilden die Grenze zwischen den Voronoi Regionen und damit auch das Voronoi Diagramm. Für die Berechnung des Voronoi Diagramms existieren zahlreiche Algorithmen, die sich in ihrer Herangehensweise an das Problem unterscheiden.

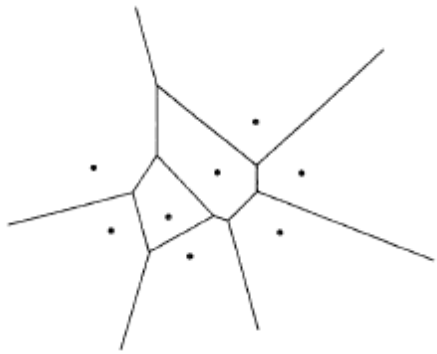


Abb. 9 : Voronoi Diagramm einer Punktmenge [AuFr]

Iterative Konstruktion

Bei der iterativen Konstruktion wird ein bereits existierendes Voronoi Diagramm vorrausgesetzt. Dabei wird ein neuer Punkt in das vorhandene Diagramm eingefügt und nur umliegende Regionen neu berechnet. Dieser Algorithmus wurde von Green und Sibson 1977 entwickelt.

Divide and Konquer

Wie bei vielen anderen Problemen der Informatik wird hierbei der Ansatz verfolgt, das Gesamtproblem in viele Teilprobleme zu unterteilen. Der Hindernisraum wird daher in eine bestimmte Anzahl von Teilräumen zerteilt und für jeden dieser Teilräume das Voronoidiagramm berechnet. Anschließend werden die dabei entstandenen Voronoidiagramme wieder zu einem gesamten zusammengefügt.

Fortunes Algorithmus

Dieser Algorithmus wurde von Steven Fortune um 1985 entwickelt. Bis dato wurde überwiegend die iterative Konstruktion zur Berechnung von Voronoidiagrammen genutzt. Diese Methode beruht auf der Idee, den Hindernisraum von einer Linie, der sogenannten Sweep Linie, entlang der X-Achse, wie eine Zeitleiste, zu durchlaufen und beim Auftreffen auf Hindernisse das Voronoidiagramm zu berechnen. Dabei steht die Lösung für den bereits durchlaufenen Teil des Hindernisraums fest. Dieser Algorithmus ist auch unter Sweepline Algorithmus bekannt.

Wertung

Der größte Vorteil von Voronoidiagrammen ist, dass der ermittelte Pfad immer den maximalen Abstand zu Hindernissen gewährleistet. Die Berechnung von Voronoidiagrammen ist jedoch sehr komplex und benötigt einige Systemresourcen.

3.3. Zellzerlegungsverfahren

Ziel von Zellzerlegungsverfahren ist es, den gesamten Freiraum in Teilprobleme zu unterteilen. Hierbei wird der Freiraum in disjunkte konvexe Regionen zerlegt. Benachbarte Regionen werden dann verbunden und somit entsteht ein Straßennetz in Form eines Graphen, in dem dann mittels Suchalgorithmus ein Weg vom Start- zum Zielpunkt gefunden werden kann. JacobT. Schwartz und Micha Sharir haben in den achtziger Jahren viel zu den heute bekannten Verfahren beigetragen. Jean Claude Latombe faste in seinem Buch „Robot Motion Planing“ viele dieser Verfahren zusammen und führt deren Ideen fort und bietet damit ein interessantes und umfangreiches Kompendium der Robotik.

Man unterscheidet die exakte und die approximierte Zerlegung.

3.3.1. Exakte Zerlegung

Ziel dieser Methode ist es, den Freiraum in eine endliche Menge von konvexen Polygonen zu zerlegen. Dabei dürfen sich die Flächen zweier unterschiedlicher Polygone nicht überschneiden. Zwei Zellen können nur dann miteinander verbunden werden, wenn sie benachbart sind. Die Zielstellung die Fläche in konvexe Polygone zu zerlegen, begründet sich durch die Bewegungsfreiheit eines Roboters in einer solchen Zelle, ohne eine weitere Zerlegung der Zelle berechnen zu müssen. Der Mittelpunkt jeder Zelle kann als Wegpunkt betrachtet werden. Aus dieser Zerlegung kann man nun sehr einfach einen Baum aus Wegpunkten ausgeben. Die Unterknoten eines Knotens sind seine benachbarten Zellen. In diesem Baum kann nun mithilfe

eines Suchalgorithmus ein Pfad gesucht werden. Daraufhin kann eine Sequenz von benachbarten Zellen ausgeben werden, die von einem zum anderen Punkt führt.

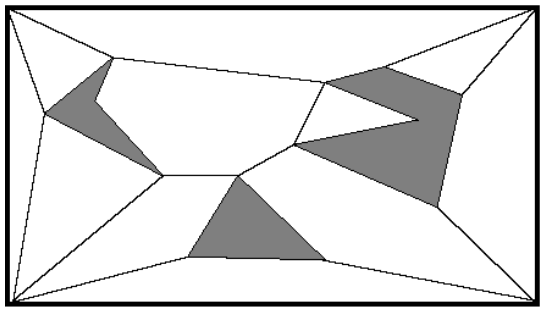


Abb. 10 : Triangulation - Exakte Zellzerlegung[Deut]

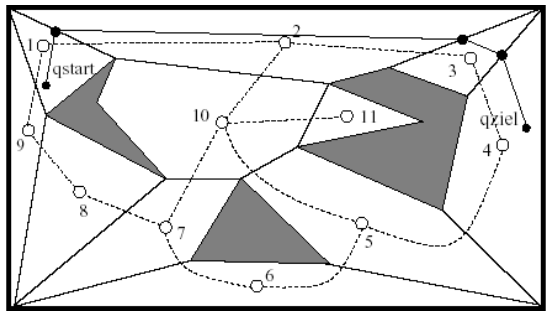


Abb. 11 : Strassennetz im zerlegten Raum [Deut]

Der große Vorteil von exakten Zellzerlegungsverfahren ist, dass sie vollständig sind. Sie finden also in jedem Fall einen Pfad, sofern dieser existiert.

Ein Nachteil ist, dass Pfade oftmals kritisch sind, da sie nahe an Hindernissen vorbeiführen.

3.3.2. Approximierte Zerlegung

Die approximierte Zellzerlegung basiert ebenfalls auf Gitterkarten. Zellen können unterschiedlich groß sein, da nur Zellen, die teilweise belegt sind, weiter zerlegt werden. Ist eine Zelle komplett frei, kann sie befahren werden und wird in einem Array gespeichert. Vollständig belegte Zellen können nicht befahren werden und müssen daher auch nicht weiter betrachtet werden.

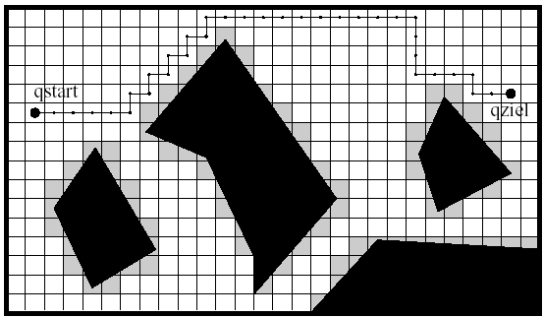


Abb. 12 : Approximierte Zellzerlegung [Deut]

Es gibt verschiedene Ansätze für die approximierte Zellzerlegung, jedoch basieren alle Verfahren darauf, den Konfigurationsraum in vordefinierte Körper zu zerlegen, wie zum Beispiel Rechtecke. Im Gegensatz zur exakten Zellzerlegung wird hierbei nicht der gesamte Raum zerlegt, sondern nur jene Zellen, welche sowohl Hindernisse als auch frei Fläche beinhalten.

Ein Verfahren zur approximierten Zellzerlegung ist der **Quadtree**. Dabei wird der gesamte Konfigurationsraum in 4 Rechtecke zerlegt. Jene Zellen, welche vollständig freien Raum beinhalten oder vollständig von Hindernissen bedeckt werden, brauchen nicht weiter zerlegt werden, weil ihre Konfiguration eindeutig ist. Sind sie vollständig von freiem Raum belegt, sind sie also vollkommen befahrbar, wird die Zelle komplett von einem Hindernis bedeckt, ist sie nicht befahrbar. Nun werden die Zellen betrachtet, die sowohl Freiraum als auch Hindernisse beinhalten und ebenfalls in 4 Rechtecke geteilt. Dieser Ablauf wird solange wiederholt, bis die Zellen eine vordefinierte Größe unterschreiten, bei der es keinen Sinn macht, sie weiter zu zerlegen. Zum Beispiel, wenn die Zelle so groß ist wie der Roboter ist es sinnlos sie weiter zu zerlegen, da eh keine weiteren befahrbaren Zellen entstehen würden.

Ein großer Nachteil von approximierten Zellzerlegungsverfahren ist, dass sie nicht vollständig sind. Es kann also vorkommen, dass ein existierender Pfad nicht gefunden wird.

3.4. Potenzialfelder

Latombe fasst in [LaJC] wichtige Eigenschaften von Potenzialfeldern zusammen.

Bei einem Potenzialfeld erhält das Ziel ein hohes anziehendes Potenzial. Hindernisse hingegen erhalten ein hohes abstoßendes Potenzial. Die Fahrt wird in Richtung des größten Potenzials durchgeführt.

Hindernisse erhalten das **repulsive Potenzial,** welches eine abstoßende Wirkung auf den Roboter ausübt.

Das Ziel erhält ein **attraktives Potenzial,** welches eine anziehende Wirkung auf den Roboter ausübt. Wichtig ist hierbei, dass das attraktive Potenzial im gesamten Konfigurationsraum C wirkt. Damit wird sichergestellt, dass der Roboter im gesamten Konfigurationsraum dem Potenzial des Zielpunktes ausgesetzt ist, ohne das eine Navigation nicht möglich wäre. Abbildung 13 veranschaulicht die Wirkung eines Potenzialfeldes anhand einer Kugel, die in einem Raum immer zur niedrigsten Stelle rollt.

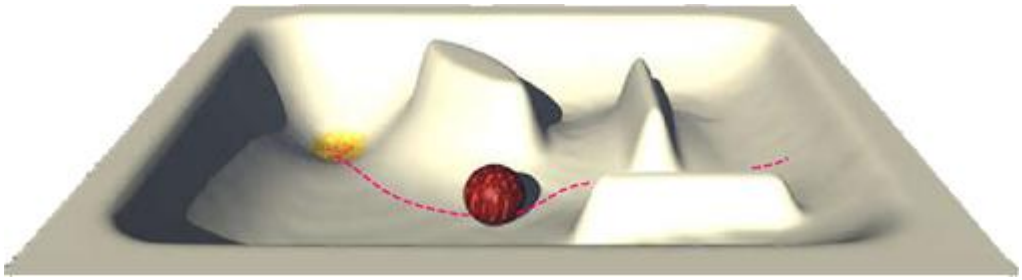


Abb. 13 : Veranschaulichung eines Potenzialfelds, die Kugel rollt zum Punkt mit dem größten Potenzial [Rome]

Ein großer Vorteil, von Potenzialfeldern gegenüber anderen Pfadplanungsverfahren ist, dass sie sich auch für Fahrten in unbekannten Umgebungen eignen, sowie auch in Umgebungen, in denen sich dynamische Hindernisse, wie weitere Roboter, befinden.

Unter bestimmten Umständen ist diese Methode nicht vollständig, da ein Pfad durch mögliche lokale Potenzialminima enden kann, und daher ein globales Potenzialmaxima bzw. der Zielpunkt nicht gefunden werden kann, obwohl ein möglicher Pfad vorhanden ist.

3.5. Suchverfahren

„Sehr viele Probleme der künstlichen Intelligenz lassen sich letztendlich auf ein Suchproblem zurückführen. Die Eigenschaften und Lösungsverfahren von Suchproblemen sind daher von grundlegender Bedeutung für die gesamte künstliche Intelligenz.“ [WV99 S.19]

Bevor der Roboter mit der Navigation beginnen kann, muss zunächst aus dem Graphen aus Wegpunkten ein Weg mittels eines geeigneten Suchverfahrens gefunden werden.

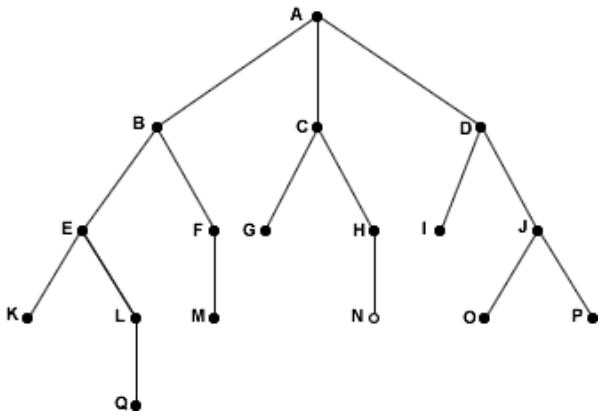


Abb. 14: Suchbaum

Abbildung 14 zeigt einen Suchbaum mit Startknoten A und dem Zielknoten N. In [WV] werden viele Suchverfahren detailliert vorgestellt. Einige dieser Suchverfahren werden im folgenden Absatz vorgestellt. Zur Visualisierung werden diese Methoden anhand des Suchbaumes aus Abbildung 14 dargestellt und die Ermittlung des Weges näher erläutert.

3.5.1. Tiefensuche

Die Tiefensuche gehört zu den uninformierten Suchverfahren. Das bedeutet, dass diese Verfahren keine zusätzlichen Informationen über die Problemstellung besitzen.

Hierzu verwendet man eine Liste. Zu Beginn beinhaltet sie nur den Startknoten. Diesen ersetzt man durch seine nachfolgenden Knoten. Jetzt beginnt man mit dem ersten Element dieser Liste und ersetzt es wieder durch seine Nachfolger. Dies wird nun iterativ durchgeführt. Hat ein Knoten keine Nachfolger und ist selbst auch nicht der Zielpunkt wird er aus der Liste gelöscht. Diese Prozedur wird nun solange wiederholt bis man den Zielpunkt in der Liste erreicht. Da bei

dieser Methode im zuerst bis zum tiefsten Element eines Astes gesucht wird nennt man sie Tiefensuche. In der folgenden Abbildung ist der Verlauf der Tiefensuche dargestellt.

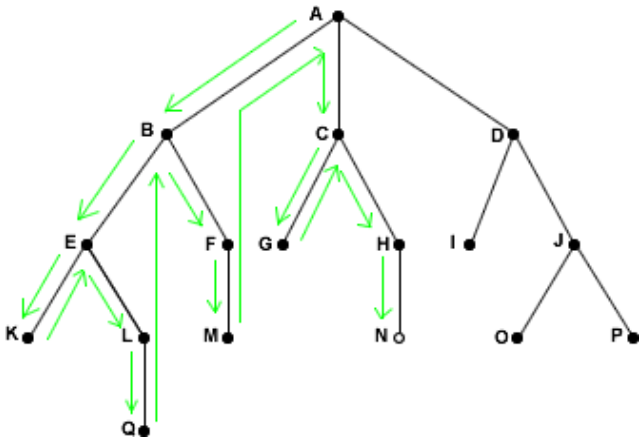


Abb. 15 : Verlauf der Tiefensuche im Suchbaum

3.5.2. Breitensuche

Die Breitensuche ist ein weiteres uninformiertes Suchverfahren, da auch dieses Verfahren keinerlei Informationen über die Problemsituation besitzt. Sie sucht wie schon der Name verrät im Gegensatz zur Tiefensuche jede Schicht des Suchbaumes in der Breite. Hierbei werden die Nachfolgerknoten eines expandierten Knotens ans Ende der Liste verschoben. In der folgenden Grafik wird der Verlauf der Breitensuche verdeutlicht.

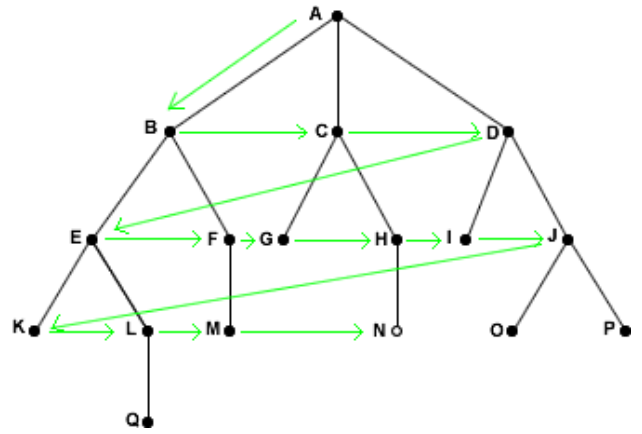


Abb. 16 : Verlauf der Breitensuche im Suchbaum

3.5.3. Heuristische Suche

Hierbei handelt es sich um ein informiertes Suchverfahren. Es kann angewendet werden wenn Information darüber vorliegen, wie erfolgsversprechend ein Knoten ist. Beispielsweise bei einem Routenproblem könnte man bei der Wahl des nächsten Knoten die Entfernung zum Zielpunkt berücksichtigen. Voraussetzung dafür ist natürlich Information darüber, wie weit ein Knoten vom Zielknoten entfernt ist.

3.5.4. A* Algorithmus

Bei Routenproblemen bei denen mehrere Wege zum Zielpunkt führen greift man gern auf diesen Algorithmus zurück, da er in der Lage ist, den kürzesten bzw. schnellsten Weg zu finden. Der A-Stern Algorithmus ist ein informiertes Suchverfahren. Das bedeutet, dass Informationen zu den Kosten um von einem Knoten zum Nächsten zu gelangen vorhanden sein müssen. Da es sich hierbei um Koordinaten in einer Karte handelt sind die Kosten um von einem zum nächsten Wegpunkt zu gelangen bekannt und daher drängt sich dieses Verfahren geradezu auf. Desweiteren wird eine heuristische Schätzfunktion benötigt, die die minimalen Kosten vom aktuellen Knoten zum Zielpunkt schätzt. In diesem Fall wird die Luftlinienheuristik zur Schätzung der Minimalkosten verwendet. Aufgrund der Tatsache, dass es keinen kürzen Weg zwischen zwei Punkten geben kann, außer der direkten Verbindung, also der Luftlinie, ist die Luftlinienheuristik für solche Probleme prädestiniert.

Funktionsweise

Für A Stern Algorithmus wird eine Agenda, einer Datenstruktur zur Speicherung von Knotenfolgen, verwendet. Zu Beginn befindet sich der Startknoten in der Agenda. Dieser wird expandiert. Das bedeutet, dass alle vom Startknoten erreichbaren Knoten in der Agenda gespeichert werden. Danach wird immer der Knoten expandiert, dessen bisherigen Kosten addiert zu den geschätzten Kosten zum Zielknoten am geringsten sind. Dieser Ablauf passiert solange, bis der Zielknoten erreicht wurde.

Pseudocode:

```
Programm astern
        Für jeden erreichbaren punkt von start
                Add weg (start - punkt) zur openlist
        Füge start zur closelist hinzu
        While  not success
                Entnehme kürzeste strecke aus openlist
                If punkt == ziel
                        Success
                endif
                If punkt nicht in closedlist
                        Für jeden erreichbaren punkt von punkt aus
                                Add weg zur closedlist
                Endif
        endwhile
End Programm astern
```

Eigenschaften

Da immer eine Lösung gefunden wird, sofern auch eine existiert, ist der A*Algorithmus vollständig. Der Algorithmus ist zudem optimal, da immer die beste Lösung gefunden wird.

Der einzige Nachteil des A*Algorithmus ist sein Speicherbedarf, da die Agenda bei einigen Problemen sehr groß werden kann, Zusätzlich müssen alle bekannten Knoten im Speicher vorhanden sein. Bei einigen Problemen können das sehr schnell Milliarden werden.

Beispiel – Alle Wege führen nach Rom

Die Funktionsweise des A*Algorithmus wird nun am Beispiel einer Routenplanung verdeutlicht. Hierbei soll der kürzeste Weg von Brüssel nach Rom gefunden werden. Die Zahl an den Kanten steht für die Entfernung zwischen zwei Knoten, die Zahl innerhalb der Knoten zeigt die geschätzte Entfernung zum Zielknoten, die durch die Schätzfunktion bestimmt wurde. Hierbei wurde die Luftlinienheuristik angewandt.

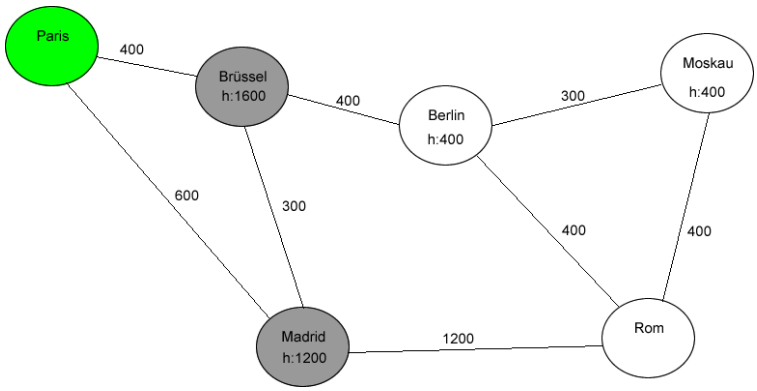


Abb. 17 : A*Algorithmus – Schritt 1

In der Agenda befinden sich [Paris-Brüssel:2000; Paris-Madrid:1800].

Da die Strecke Paris-Madrid die geringsten entstandenen Kosten+geschätzte Kosten besitzt, wird der Knoten Madrid expandiert.

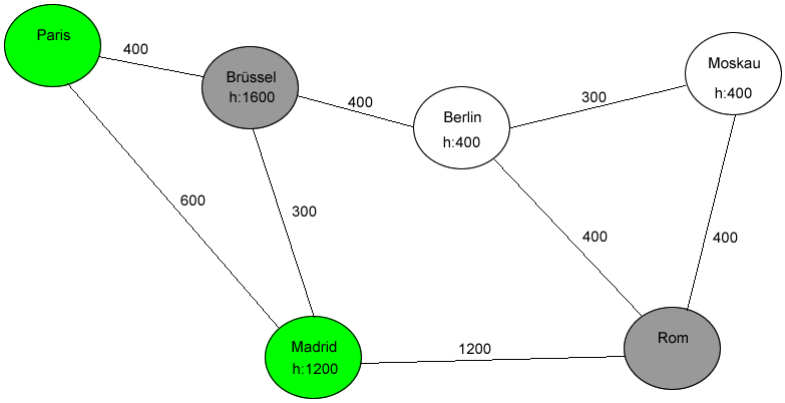


Abb. 18 : A*Algorithmus – Schritt 2

Aktuelle Agenda [Paris-Brüssel:2000; Paris-Madrid-Rom:3000; Paris-Madrid-Brüssel:3700]. Nun wird der Knoten Brüssel expandiert, da dort die Gesamtkosten am geringsten sind.

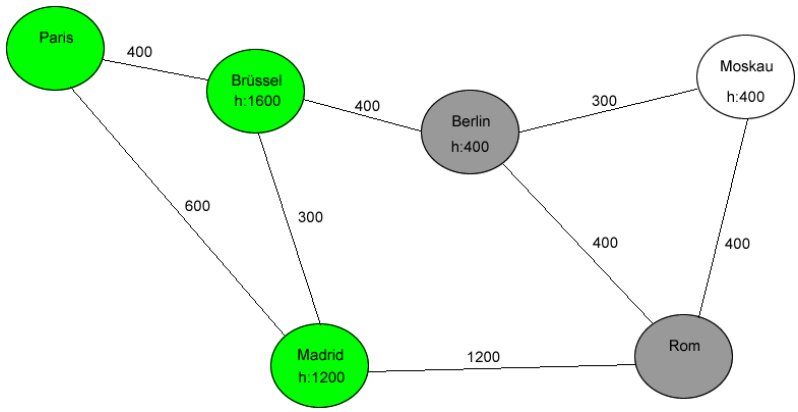


Abb. 19 : A*Algorithmus – Schritt 3

Die aktuelle Agenda sieht folgendermaßen aus [Paris-Brüssel-Berlin:2400; Paris-Madrid-Rom:3000; Paris-Madrid-Brüssel:3700]. Aufgrund der geringsten Gesamtkosten wird nun Berlin expandiert.

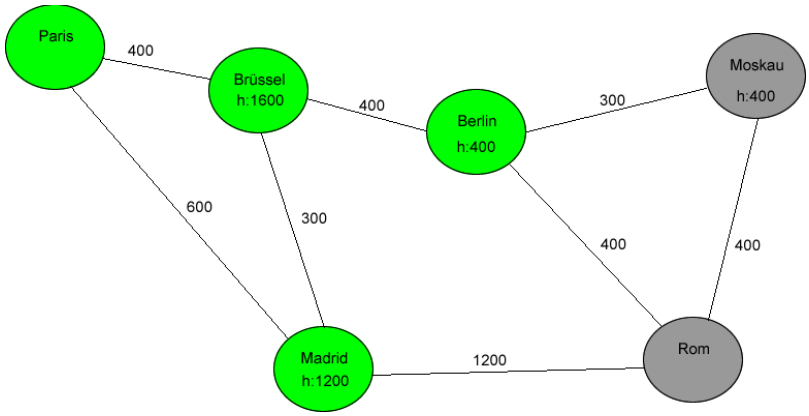


Abb. 20 : A*Algorithmus – Schritt 4

Nachdem Berlin expandiert wurde sieht die Agenda folgendermaßen aus. [Paris-Brüssel-Moskau:2700; Paris-Brüssel-Berlin-Rom:2800; Paris-Madrid-Rom:3000]. Nun muss Moskau expandiert werden.

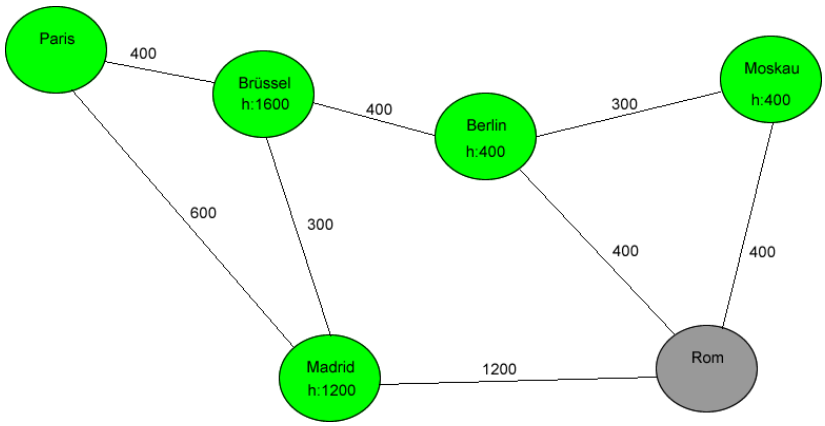


Abb. 21 : A*Algorithmus – Schritt 5

Die aktuelle Agenda sieht folgendermaßen aus [Paris-Brüssel-Berlin-Rom:2800; Paris-Brüssel-Moskau-Rom:3100; Paris-Madrid-Rom:3000; Paris-Madrid-Brüssel:3700]. Da nun der nächste zu expandierende Knoten Rom wäre, ist der A Stern Algorithmus beendet.

Der kürzeste Weg wurde gefunden. Paris-Brüssel-Berlin-Rom mit 2800km ist der kürzeste Weg.

3.6. Entscheidung Wegplanungsalgorithmus

Nach der Analyse der vorgestellten Verfahren, haben sich zwei der Verfahren als „brauchbar“ herausgestellt. Zum einen wäre das Voronoidiagramm eine ideale Lösung, da es ein vollständiges Verfahren ist und der ermittelte Pfad noch dazu den größtmöglichen Abstand zu den Hindernissen gewährleistet. Jedoch ist die Implementierung eines Voronoialgorithmus sehr aufwendig und es existiert nicht gerade viel Literatur zu dem Thema.

Eine weitere Alternative, wäre die Exakte Zellzerlegung, da diese ebenfalls vollständig ist.

Die Entscheidung fiel auf die vertikale Zellzerlegung, da diese mit geringerem Aufwand zu implementieren ist, und den Anforderungen vollkommen genügt.

Der Voronoi-Algorithmus hat den Nachteil, dass er sehr rechenzeitaufwendig ist und anhand des Voronoi-Diagramms noch keine Wegpunkte bekannt sind und diese erst noch ermittelt werden müssen. Der Vorteil des Voronoi-Algorithmus ist allerdings, dass er Pfade mit einer maximalen Entfernung zu Hindernissen ermittelt. Allerdings sind die Pfade, die mit der Vertikalen Zellzerlegung ermittelt werden vollkommen ausreichend und für die Fahrt zufrieden stellend.

4.

KONZEPTION UND LÖSUNGSIDEE

4.1. Architektur

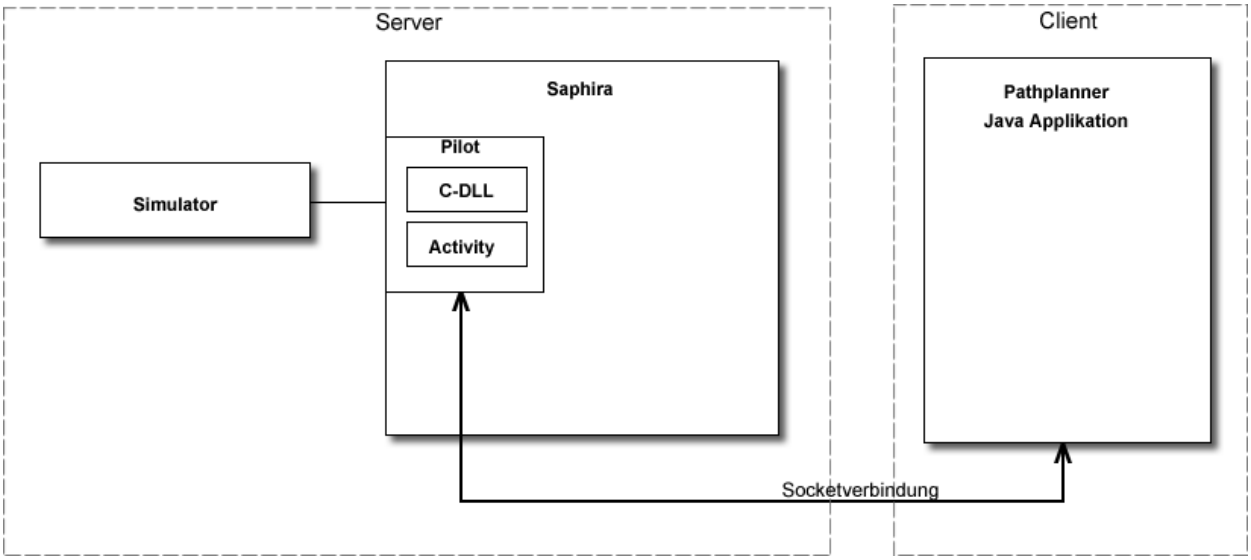


Abb. 22 : Architektur des Systems

In der Abbildung 22 sind beide Module, aus denen die Applikation hauptsächlich besteht, zu sehen. Zunächst gibt es eine in C geschriebene dynamische Bibliothek, die zur Laufzeit in die Saphira Umgebung geladen wird und eine Schnittstelle nach Außen über Sockets bereitstellt. Dieses Modul ist der Pilot, da es zur Steuerung des Pioneer Simulators dient. Zum Piloten gehört noch eine in Colbert geschriebene Activity, die den Ablauf des Piloten steuert. Das zweite Modul ist eine in Java geschriebene Applikation, die mittels Sockets mit der C-Bibliothek kommuniziert. Diese Applikation ist der Pathplanner und beinhaltet die Pfadplanung, sowie auch den Navigator. Der Großteil der Anwendungslogik läuft in der Java Applikation, wie zum Beispiel die Erstellung einer Karte und die Berechnung der Zellzerlegung, sowie die Ermittlung des kürzesten Pfades der Straßenkarte mithilfe des A*Algorithmus, die das Programm zur Navigation des Mobilen Systems, hier des Pioneer Simulators, verwendet. Die C-Bibliothek interpretiert die vom Java-Programm gesendeten Steuerungsbefehle, wandelt diese in Saphira-Befehle um und kümmert sich intern um den Ablauf von notwendigen Behaviors.

4.2. Ablauf der Applikation

Der Ablauf der Applikation kann grob in 2 Phasen unterschieden werden.

Editierphase

Hierbei wird eine Karte mit dem integrierten Map Editor erstellt oder eine Datei im .map Format geladen. Nun können Objekte auf der Karte noch verändert werden. Beispielsweise können zusätzliche Wände erzeugt und vorhandene Wände verschoben, gedreht oder skaliert werden.

Des Weiteren muss ein Roboter Objekt sowie ein Zielpunkt in der Karte platziert werden. Eine erzeugt Karte kann jederzeit gespeichert werden. Dabei wird gleichzeitig eine .wld Datei mit den

aktuellen Roboterkoordinaten für den Simulator gespeichert. Nun kann die Zellzerlegung sowie die Berechnung des Pfades mit Hilfe des A* Algorithmus vorgenommen werden.

Navigationsphase

Jetzt muss Saphira und der Pioneer Simulator gestartet werden. Die gespeicherte Karte wird in Saphira geladen und im Simulator wird die entsprechende .wld Datei geladen.

Es werden nun noch die Dynamische Bibliothek pilot.dll und die Aktivität pilot.act in Saphira geladen. Saphira und der Roboter bzw. Simulator werden nun verbunden.

Sobald die Verbindung zwischen Saphira und einem Roboterserver steht, kann die Aktivität pilot mit dem Befehl start pilot über Saphira gestartet werden. Der Socketserver wurde nun in Saphira gestartet und wartet auf die Verbindung eines Clients. Dabei blockiert Saphira.

Im Pathplanner wird nun mit dem Button Connect die Verbindung zwischen dem Piloten in Saphira und dem Navigator im Pathplanner hergestellt.

Mit Betätigung des navigate Buttons wird die Navigation gestartet und sobald ein Wegpunkt erreicht wurde, verändert der abgefahrene Weg seine Farbe von rot nach grün. Der Roboter in der Karte im Pathplaner wird zum erreichten Wegpunkt verschoben. Sobald der Zielpunkt erreicht ist, wird dies durch eine Erfolgsmeldung bestätigt und es kann mit einer weiteren Navigation begonnen werden oder das Programm beendet werden.

4.3. Kommunikation

Die Kommunikation dieses Systems erfolgt mit Hilfe von Sockets. Der große Vorteil von Socketverbindungen ist die Plattform- sowie Programmiersprachenunabhängigkeit. Dies ist ein Fakt, der bei diesem Projekt notwendig ist, da hierbei ein Javamodul mit einer in C geschriebenen DLL kommuniziert.

Für die konkrete Kommunikation wurde ein spezielles Protokoll entwickelt, das aus 3 Phasen besteht. Es heißt "*Robot Controll Protokoll*" (RCP)

Zunächst wird der Befehl vom Client zum Server gesendet. Danach bestätigt der Server den Empfang sowie die Richtigkeit des Befehls. In der letzten Phase bestätigt der Server die erfolgreiche Ausführung des Befehls oder gibt im Falle eines Konfliktes eine Fehlermeldung zurück. Abbildung 23 zeigt die Kommunikation des Piloten mit dem Navigator anhand eines Sequenzdiagramms.

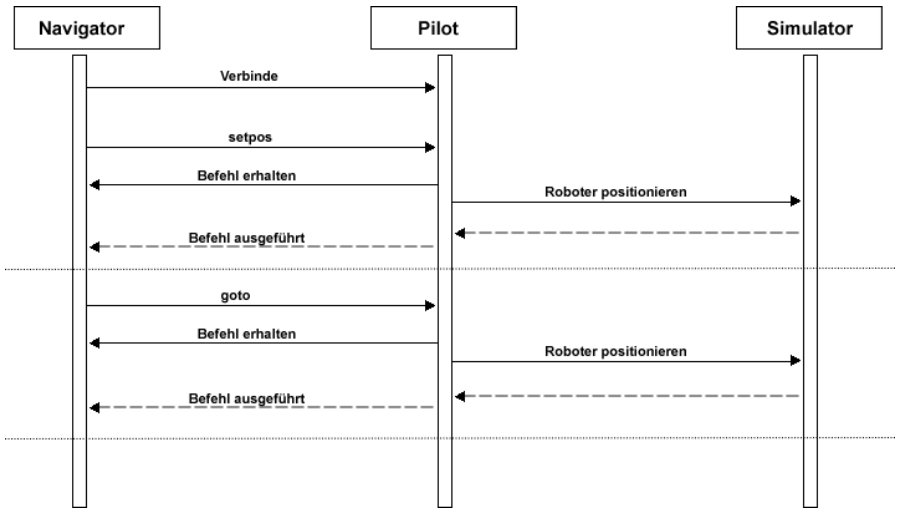


Abb. 23 : Sequenzdiagramm des Robot Control Protokoll

Diese 3 Phasen werden bei jedem übersandten Befehl abgearbeitet. Am Ende der Kommunikation wird vom Client ein Ende Befehl an den Server geschickt. Dieser wird noch vom Server bestätigt und die Kommunikation wird beendet.

Das RC – Protokoll beinhaltet folgende Befehle.

goto <x> <y> <a>

<x> ::= X-Koordinate

<y> ::= Y-Koordinate

<a> ::= Winkel in Grad

- Bewegen des Roboters zur Position x,y

  Der letzte Parameter gibt den Winkel des Roboters an

move <d>

<d> ::= Länge in mm

- Bewegen des Roboters um die angegebene Länge

turn <a>

<a> ::= Winkel

- Drehen des Roboters um die angegebene Gradzahl

setpos <x> <y> <a>

<x> ::= X-Koordinate

<y> ::= Y-Koordinate

<a> ::= Winkel in Grad

- Verschieben des Roboters auf der Karte zur angegebenen Position

4.4. Umsetzung der Karte

Die Karte im Pathplanner besteht ausschließlich aus Wänden und aus Wänden zusammengesetzten Hindernissen in Form von Polygonen.

Internes Kartenmodell

In Abbildung 24 werden alle wichtigen Elemente, die zusammen das Kartenmodell beschreiben aufgezeigt. Im Mittelpunkt steht das MapVO, welches alle Elemente der Karte vereint. Es hält Listen für Wände und Hindernisse(Polygone). Sobald eine Straßenkarte oder ein Pfad ermittelt wurden, stehen auch diese dem MapVO zur Verfügung. Das MapVO steht mit dem PathVO und dem RoadMapVO in 1 zu 1 Beziehung. Das bedeutet, dass eine Karte nur über eine Straßenkarte sowie einen optimalen Pfad verfügen kann. Analog dazu bedeutet dies, dass eine Straßenkarte zu genau einer Karte, sowie ein Pfad zu genau einer Karte gehören.

Wände werden im WallVO gespeichert. Ein WallVO hat mehrere Eigenschaften, wie zwei Endpunkte, einen Mittelpunkt, Winkel und Länge. Eine Karte kann aus mehreren Wänden bestehen. Diese werden in der MapVO in der Liste walls gespeichert. Analog dazu werden auch Hindernisse in einer separaten Liste polygons gespeichert. Ein Hindernis besteht aus mehreren Eckpunkten.

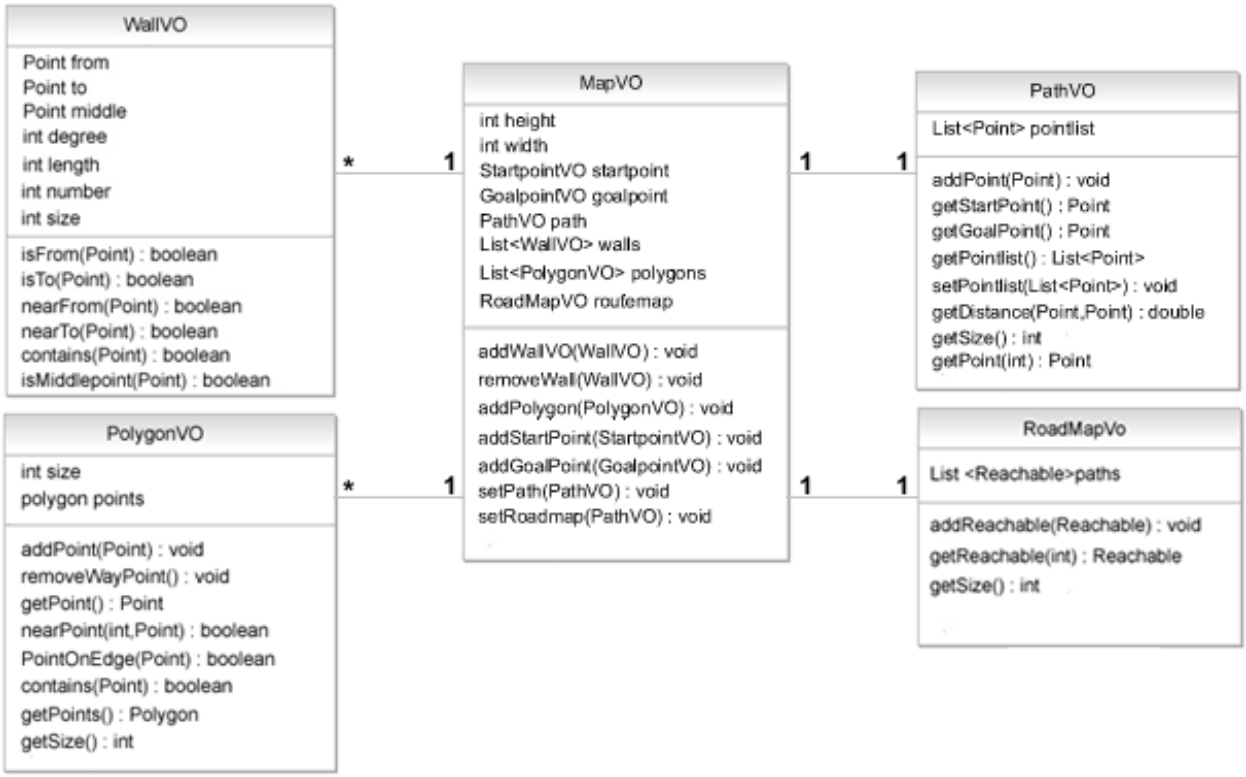


Abb. 24 : Klassendiagramm der Karte

Dateiformat der Karte

Die Karte wird im .map Format abgespeichert. Diese ist genauso aufgebaut wie die Karte von Saphira, die schon unter 2.3. beschrieben wurde. Das ist notwendig um die Karte später auch in Saphira laden zu können. Allerdings gibt es auch einige gravierende Unterschiede. Es werden ausschließlich Wände gespeichert, was bedeutet, dass Polygone in Wände zerfallen. Wird

eine .map Datei im Pathplanner geladen, die Korridor, Lane oder andere Artefakte enthält geladen, werden diese vom Pathplanner in Wände konvertiert. Dabei werden jedoch nur Korridore, Lanes, Walls und Doors berücksichtigt.

4.5. Algorithmen zur Zellzerlegung

Lavalle beschreibt in [LaJC] die beiden Algorithmen zur Zellzerlegung.

4.5.1. Vertikale (trapezoide) Zellzerlegung

Hierbei werden von jedem Eckpunkt jedes Objektes in der Karte Strahlen in beide Richtungen parallel zur Y-Achse ausgesandt, bis sie auf ein Hindernis oder die Kartenbegrenzung treffen. Durch die entstandenen Kanten wird der Konfigurationsraum Cfree in eine endliche Menge von konvexen Polygonen, auch Zellen genannt, zerlegt, so dass sich keine Zellen überschneiden. Benachbarte Zellen können befahren werden. Nun gilt es einen Weg, vom Start- zum Zielpunkt, zu finden. Der gefundene Weg besteht aus der Aneinanderreihung benachbarter Zellen und wird auch als Kanal bezeichnet.

Die Mittelpunkte der entstandenen Kanten bilden nun die Knoten im Graphen. Oft werden auch die Mittelpunkte der Zellen als zusätzliche Wegpunkte definiert. Danach müssen alle erreichbaren Knoten verbunden werden, das sind Knoten benachbarter Zellen, und es entsteht ein Straßennetz.

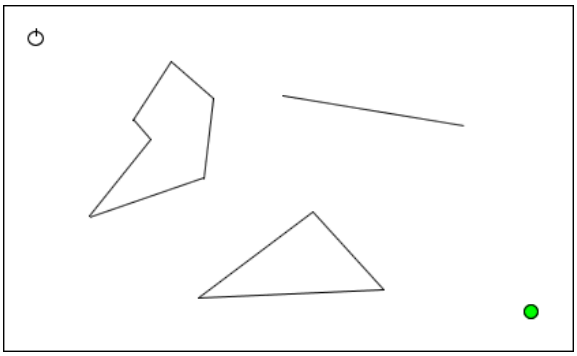


Abb. 25 : Karte vor der Zellzerlegung

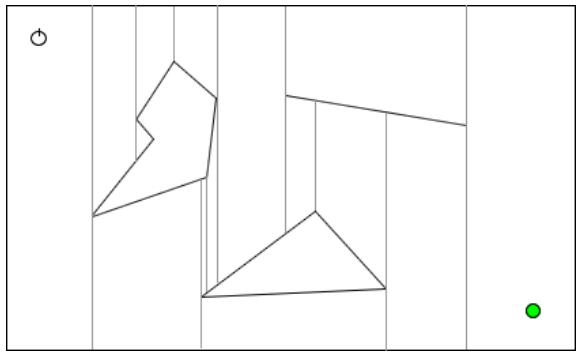


Abb. 26 : Zellzerlegung

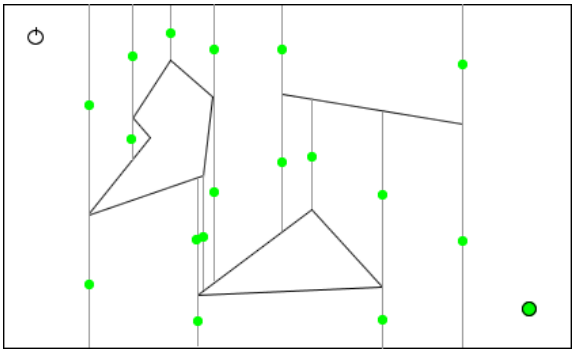


Abb. 27 : Bestimmen der Wegpunkte

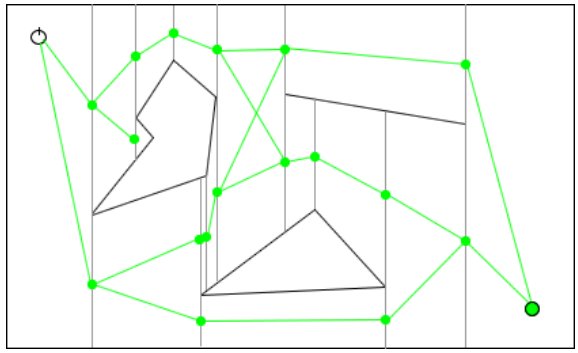


Abb. 28 : Ermitteln der Straßenkarte

4.5.2. Triangulation

Die Triangulation ist eine Alternative zur vertikalen Zellzerlegung. Dabei wird der gesamte Konfigurationsraum in dreieckige Zellen zerlegt, wie in Abbildung 29 zu sehen ist.

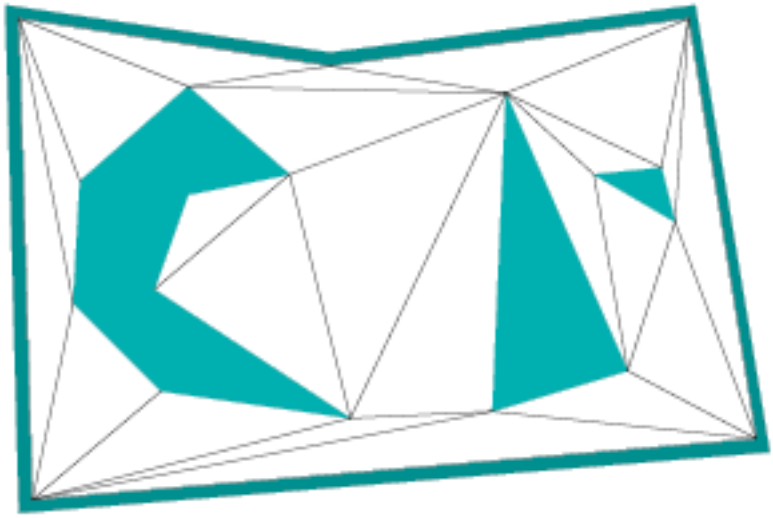


Abb. 29 : Triangulation [LaSt]

Für die Triangulation gibt es verschiedene Algorithmen. Anschließend wird, wie auch schon bei der vertikalen Zellzerlegung, die Straßenkarte bestimmt, indem die Mittelpunkte der Zerlegungskanten sowie die Mittelpunkte der Zellen gewählt werden. Dabei entsteht eine Straßenkarte wie sie in Abbildung 30 zu sehen ist.

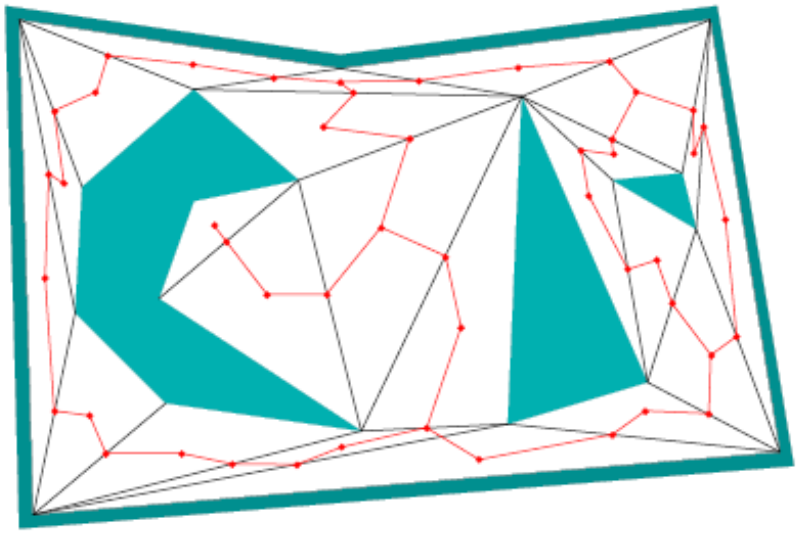


Abb. 30 : Straßenkarte der Triangulation [LaSt]

Anschließend muss auch hierbei mithilfe eines Suchverfahrens ein befahrbarer Pfad ermittelt werden.

4.6. Registrierung

4.6.1. Problem

Die Registrierung ist bei der Kartenbasierten Roboternavigation ein entscheidender Fakt, der über Erfolg und Misserfolg entscheidet. Dabei wird die interne Roboterposition mit der realen Position in Übereinstimmung gebracht. Das ist aber nicht so einfach wie es sich anhört, da man nur über die Daten von Sensoren verfügt, die nicht immer eindeutig sind und die erst ausgewertet werden müssen.

4.6.2. Lösungsansätze

Für diese Aufgabe wurden drei Möglichkeiten für die Registrierung erörtert. Diese werden nun näher beleuchtet, da sie interessante Ansätze für fortführende Arbeiten darstellen.

Korridorbasierte Registrierung

Zunächst wäre da die in Saphira integrierte Registrierung, die mit dem Befehl *sfInitRegistrationProcs()* gestartet wird. Hierbei muss jedoch angemerkt werden, dass sich diese Methode ausschließlich auf Korridore stützt, und auch nur beim Vorhandensein von Korridor Artefakten funktioniert. Das bedeutet, dass eine Karte mit vielen Korridoren, besser noch ausschließlich aus Korridoren bestehend die Grundlage dafür bieten muss. Ist dies nicht der Fall bringt diese Methode kein Ergebnis und ist somit wertlos. Es ist auch mit viel Aufwand verbunden, Korridore in einer Karte zu finden, und diese dann umzuwandeln. Auch kann die gesamte Karte nicht nur aus Korridoren bestehen.

Wandbasierte Registrierung

Daher wäre die zweite Methode eher empfehlenswert. Hierzu muss eine eigene Registrierung auf der Basis von Wänden geschrieben werden. Saphira ist in der Lage mit Hilfe der Sonardaten Hypothesen vom Vorhandensein von Wänden zu erstellen. Diese werden dann in den Strukturen *sfLeftWallHyp* und *sfRightWallHyp* gespeichert. Dafür muss sfInitRegistrationProcs() gestartet werden um Ergebnisse in der Struktur zu erhalten. Sobald das Flag viewable für eines der Hypothesen auf 1 gesetzt ist, wurde eine Hypothese für eine Wall erzeugt, entweder auf der rechten oder der linken Seite des Roboters. Nun kann man die Werte aus der jeweiligen Struktur auslesen und versuchen eine Wall in der Nähe des Roboters zu finden die am besten zu der Hypothese passt und somit die Differenz mit der angenommenen Position vergleichen. Dann kann man mit dem Befehl *sfMoveRobot()* den Roboter in der Karte verschieben, so dass beide Positionen übereinstimmen. Diese Methode sollte als eigenständiger Prozess bzw. Thread im Hintergrund laufen.

Komplette Sensorauswertung

Die dritte Methode wäre wahrscheinlich am effizientesten, jedoch auch die am schwierigsten zu implementierende Methode. Hierbei werden die Sonardaten des Roboters mit der Karte verglichen. Dabei werden auch ältere Daten berücksichtigt. Wenn man eine Übereinstimmung Matching() der Sensordaten und der Karte gefunden hat, kann man wie auch schon bei der zweiten Methode den Roboter in der Karte so versetzen, dass die reale Position mit der Position des Roboters in der Karte übereinstimmt.

Theoretisch wäre die zweite Variante der Registrierung empfehlenswert. Sie bietet einen guten Kompromiss zwischen Aufwand und Nutzen. Dafür wird im Piloten, der in Saphira geladen wird, ein Thread gestartet, der zyklisch die Hypothesen für Wanderkennung überprüft. Sofern das

viewable Flag auf 0 gesetzt ist, existiert keine Hypothese und der Thread legt sich für kurze Zeit schlafen, bis er erneut das *viewable* Flag überprüft. Falls eine Hypothese existiert, werden deren Daten aus der Struktur *sfLeftWallHyp* oder *sfRightWallHyp* gelesen und an den Client im Java Programm gesendet. Der Client verarbeitet diese dann weiter. Dazu überprüft er die gesendeten Daten, die eine erkannte Wand beschreiben, und versucht dann auf der jeweiligen Seite eine Wand in der Karte zu finden. Für die Registrierung wird eine weitere Socket Verbindung erzeugt. Dies ist zwingend notwendig um die Navigation von der Registrierung zu kapseln, da diese völlig unterschiedlichen Aufgaben ihre eigene Verbindung benötigen.

Folgender Grafik verdeutlicht dies sehr anschaulich.

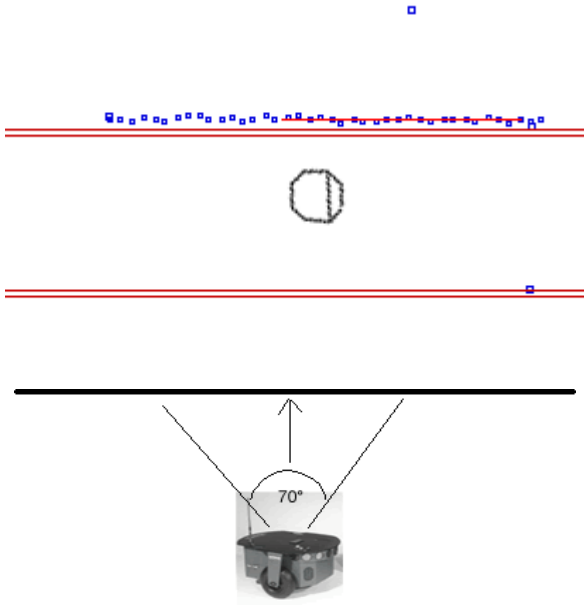


Abb. 31 oben: Vom Roboter vermutete Wand unten: Abtastwinkel für Registrierung

Das obere Bild zeigt einen Ausschnitt aus Saphira. Der Roboter fährt entlang einer Wand, die auch erkannt wurde. Die Daten der Karte sind nun in der sfLeftWallHyp Struktur verfügbar. Man kann auch erkennen, dass diese Wand leicht versetzt zur Wand in der Karte verläuft. Im zweiten Bild wird der Kegel gezeigt in dem nach dieser Wand gesucht wird, und schließlich auch gefunden wird. Es muss nur in diesem 70° großen Bereich gesucht werden, da es sehr unwahrscheinlich ist, dass die reale Position des Roboters so akkut verfälscht ist. Daher kann man davon ausgehen die gesuchte Wand in diesem Bereich zu finden. Nachdem die Wand gefunden wurde, wird nun die Differenz beider Wände berechnet, und der Roboter in der Karte versetzt.

Eine noch bessere Lösung wäre die Registrierung allein in Saphira umzusetzen, indem man sich alle Artefakte ausgeben lässt und dann das Artefakt, das sich am nächsten zur Hypothese befindet zur Korrektion der Roboterposition benutzt. Hierzu wird die Liste sfPointList benutzt, die für jedes Artefakt der Karte einen Pointer speichert. Anhand dieses Pointers kann auf jedes Attribut eines Artefakts zugegriffen werden und zur Registrierung verwendet werden. Ein weiterer überaus wichtiger Vorteil dieser Variante ist die Unabhängigkeit. Diese Methode der Registrierung lässt sich in einem einzelnen Prozess in Saphira ausführen und ist nicht auf externe Applikationen angewiesen. Daher kann diese Variante auch unabhängig benutzt und für andere Benutzer verfügbar gemacht werden.

Auf die Implementierung einer Registrierung wird verzichtet, weil diese für dieses Projekt nicht zwingend notwendig ist, da die Fahrt mit dem Simulator durchgeführt wird und diese mit weit weniger Unsicherheiten verbunden ist, als die Fahrt mit einem realen autonomen mobilen System. Daher reicht es für die beispielhafte Fahrt mit dem Simulator aus, mit einer Kollisionsabfrage zu fahren. Dies soll lediglich die Kollisionen mit Hindernissen verhindern.

Für komplexere und genauere Fahrten ist eine Registrierung allerdings unerlässlich.

4.7. Berechnung der Vertikalen Zellzerlegung

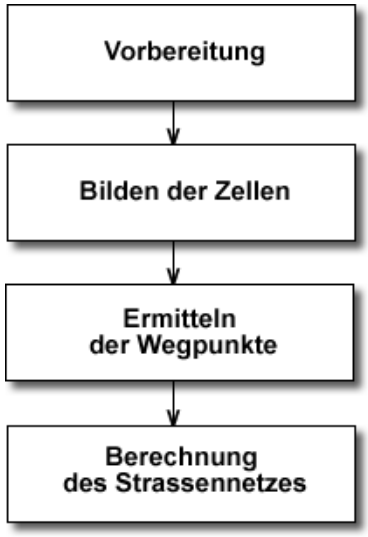


Abb. 32 : Ablauf der Vertikalen Zellzerlegung

Vorbereitung der Zellzerlegung

Bei der Zellzerlegung werden auch Wegpunkte bzw. Pfade ermittelt, die nicht mit dem Roboter befahrbar sind, weil der Pfad beispielsweise zu eng für den Roboter ist. Die Berechnung solcher unbefahrbaren Pfade muss schon vor der eigentlichen Zellzerlegung ausgeschlossen werden um spätere Probleme bei der Navigation auszuschließen. Die Methode *generateNearMapObjects()* ermittelt Hindernisse, zwischen denen kein befahrbarer Pfad existieren kann und verhindert, dass später dort ein Pfad ermittelt wird. Dazu werden die Kanten aller in der Karte befindlichen Hindernisse und Wände miteinander verglichen. Dies geschieht mit der Funktion *ptLineDist()* aus der Klasse *java.awt.geom.Line2D,* die die Entfernung einer Kante mit einem Endpunkt einer weiteren Kante ermittelt. Ist die Entfernung kleiner als die Programmkonstante *distance,* welche sich aus der Breite des Roboters und dem Abstand, der für das Verhindern von Kollisionen eingestellt wurde, zusammensetzt, wird eine virtuelle Wand zwischen beiden Kanten erzeugt, die später verhindern soll, dass zwischen beiden Kanten ein Pfad ermittelt wird, da dieser nicht befahrbar wäre. Abbildung 33 zeigt zwei Kanten, schwarz dargestellt. Vor der Zellzerlegung hat die Funktion *generateNearMapObjects()* ermittelt, dass die Entfernung der beiden Kanten zu gering ist, und daher nicht befahrbar ist. Daher werden virtuelle Kanten(blau) erzeugt. Entweder wird dabei das Lot gefällt, oder eben die Endpunkte verbunden. Dadurch wird verhindert, dass die Zellzerlegung einen Pfad zwischen beiden Kanten ermittelt.

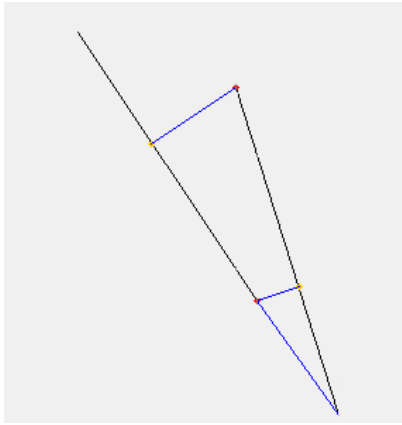


Abb. 33 : Vorbereitung der Zellzerlegung

Bilden der Zellen

Zu Beginn der Zellzerlegung wird eine Liste mit allen Kanten bzw. Wänden der Karte erstellt. Diese wird im späteren Verlauf genutzt um mögliche Schnittpunkte zu finden.

Bei der Ermittlung der Wegpunkte werden Wandpunkte und Polygonpunkte getrennt voneinander betrachtet, da sich die Prozedur der Zellzerlegung hierbei leicht unterscheidet.

Zunächst wird die Liste mit den Wänden genommen und für jede Wand von jedem Punkt aus in beiden Richtungen parallel zur Y-Achse geprüft ob ein Schnittpunkt mit einem Hindernis existiert. Dafür wird überprüft ob sich die entstandene Halbgerade und eine in der Liste vorhandene Kante schneiden. Hierbei ist ausschließlich der Schnittpunkt mit der geringsten Entfernung von Interesse. Dafür wird die zu Beginn erstellte Kantenliste zu Hilfe genommen. Existiert kein Schnittpunkt mit einem Hindernis wird der Schnittpunkt mit den Grenzen der Karte genommen. Es entstehen nun Kanten von dem jeweiligen Wandeckpunkt und dem Schnittpunkt. Hierbei wird der Raum in drei- und viereckige konvexe Polygone zerlegt. Zwischen benachbarten Zellen kann navigiert werden. Das bedeutet gleichzeitig, dass Kanten benachbarter Zellen überfahren werden können.

Nun wird die Liste mit den Polygonen genommen. Für jeden Polygonpunkt muss zunächst dessen Lage bestimmt werden. In Abbildung 34 sind die vier möglichen Lagen eines Polygonpunktes zu sehen. Anhand dieser Lage entscheidet sich, in welche Richtungen parallel zur Y-Achse nach Schnittpunkten gesucht wird.

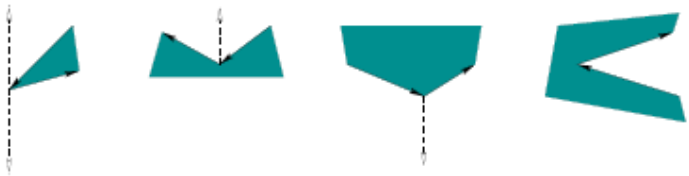


Abb. 34 : Mögliche Lagen eines Polygoneckpunktes [Lavalle]

Abbildung 35 zeigt die Zellzerlegung eines Polygons. Die Zellzerlegungskanten werden schwarz dargestellt. Es ist zu sehen, dass innerhalb des Polygons(blau) nicht zerlegt wird, da dort nicht navigiert werden kann. Der umliegende Freiraum wurde in konvexe Polygone zerlegt.

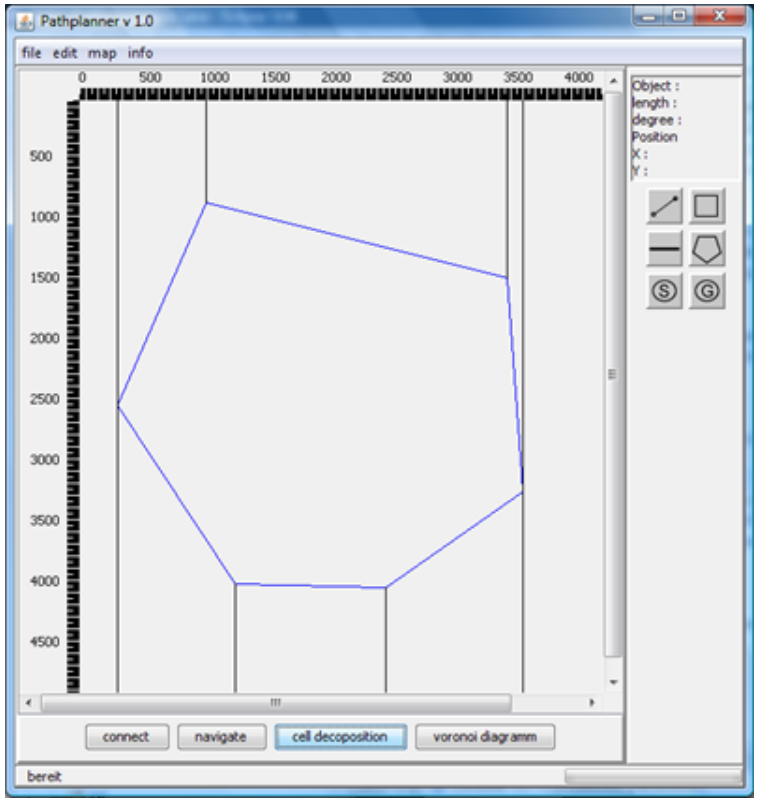


Abb. 35 : Pathplanner - Polygon und dessen Zellzerlegung

Ermitteln der Wegpunkte

Im vorangegangenen Schritt wurde der Freiraum in Zellen zerlegt. Es sind neue vertikale Kanten(Zellzerlegungskanten) entstanden. Um für die spätere Navigation den größtmöglichen Abstand zu Hindernissen zu gewährleisten, werden die Mittelpunkte dieser Kanten als Wegpunkte gewählt. Allerdings wird hierbei überprüft, ob diese Mittelpunkte ebenfalls, den nötigen Abstand zu den Hindernissen besitzen. Dazu wird die Länge der Zellzerlegungskante zu Hilfe gezogen. Ist sie kürzer als die Konstante *distance(),* wird kein Wegpunkt erzeugt. Abbildung 36 zeigt zwei Kanten, die zu nahe sind, die Zellzerlegungskante ist zu kurz. Sie wird rot eingefärbt und es wird kein Wegpunkt erzeugt.

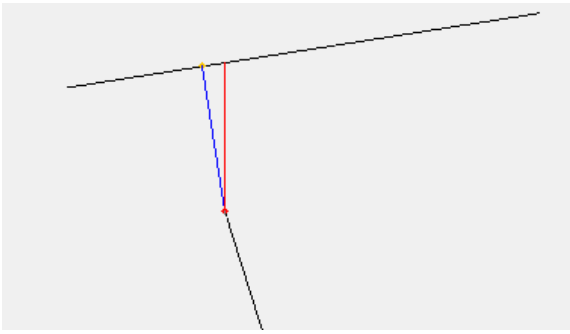


Abb. 36 : Virtuelle Kanten verbinden zu nahe Hindernisse

Berechnen des Straßennetzes

Im ersten Schritt wurden alle Wegpunkte der Karte ermittelt. Daraus ergibt sich allerding noch nicht, welche Wegpunkte untereinander erreichbar sind. Dies wird im folgenden Schritt ermittelt. Dabei wird überprüft, ob ein Pfad von einem Wegpunkt zu einem Anderen einen Schnittpunkt mit einem Hindernis oder einer Zellzerlegungskante besitzt. Ist dies der Fall, sind die beiden Wegpunkte nicht miteinander verbunden.
Jetzt muss analysiert welche Zellen nebeneinander liegen. Diese Beschreiben dann den Weg, den der Roboter befahren kann. Dies wird recht trivial gelöst indem einfach für jeden Wegpunkt geprüft wird, welche Wegpunkte er erreichen kann ohne dabei eine Objektkante oder eine Zellteilungskante zu überqueren.

Die Prozedur zur Überprüfung ob ein Schnittpunkt mit einer anderen Strecke existiert wird nun erklärt.

Es muss die Lage der Beiden Strecken zueinander betrachtet werden. Für die beiden folgenden Fälle kann kein Schnittpunkt existieren, daher kann hier abgebrochen werden.

Zum einen wenn beide x-Werte der beiden Punkte der einen Strecke entweder kleiner als der kleinere Punkt oder beide größer als der größere Punkt. Man kann an der Grafik sehr gut erkennen, das in diesem Fall kein Schnittpunkt existieren kann.

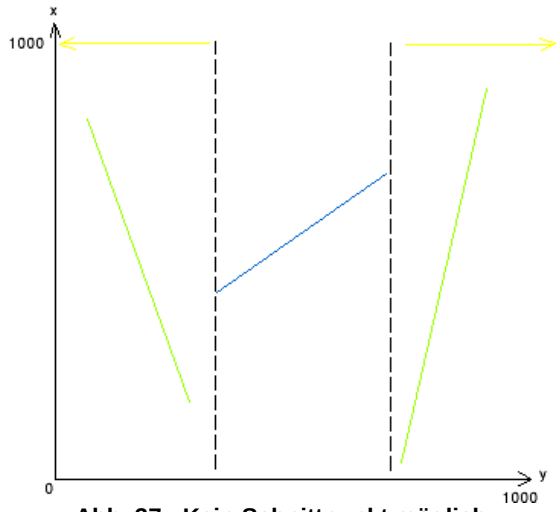


Abb. 37 : Kein Schnittpunkt möglich

Bei der zweiten Möglichkeit wird überprüft ob beide y-Werte der Punkte der Strecke größer sind als die der oder ob sie kleiner sind. Auch hier kann man an der Grafik erkennen, dass in diesem Fall kein Schnittpunkt existieren kann.

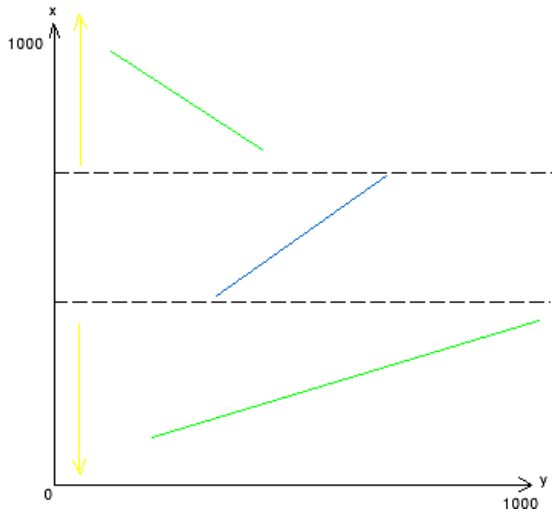


Abb. 38 : Kein Schnittpunkt möglich

Sollte einer dieser beiden Fälle eintreten, kann es keinen Schnittpunkt geben und die Schleife kann verlassen werden, andern Falls muss ein möglicher Schnittpunkt berechnet werden, und falls dieser existiert überprüft werden ob dieser auf einen der beiden Strecken liegt, denn nur dann ist es auch wirklich ein echter Schnittpunkt! Mithilfe der eben erläuterten Methode wird bereits ein Großteil von Strecken ausgeschlossen, wodurch die Rechenzeit erheblich verringert wird, da nicht für jede Strecke ein Schnittpunkt berechnet wird.

Um die Kantenbeziehungen in gerichteten und ungerichteten Graphen darzustellen, wird eine Adjazenzliste verwendet. Hierbei werden alle Wegpunkte in einer Liste gespeichert. Für jeden Wegpunkt existiert eine weitere, Liste die die Wegpunkte, auch adjazente Knoten genannt, beinhält, die von dem jeweiligen Wegpunkt erreichbar sind. Die Adjazenzliste erleichtert die Anwendung eines Suchalgorithmusses. Folgende Abbildung zeigt ein Beispiel, wie ein Graph in einer Adjazenzliste gespeichert wird.

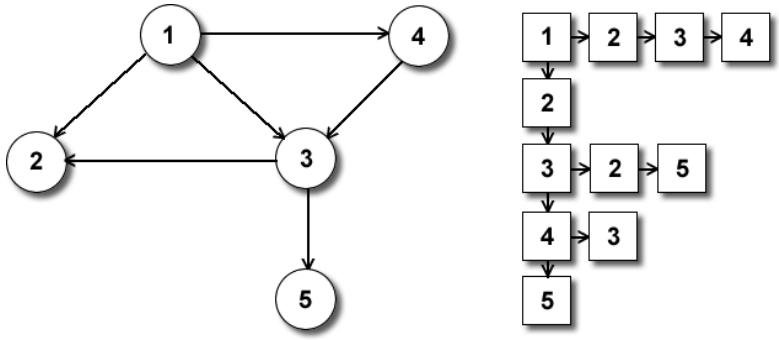
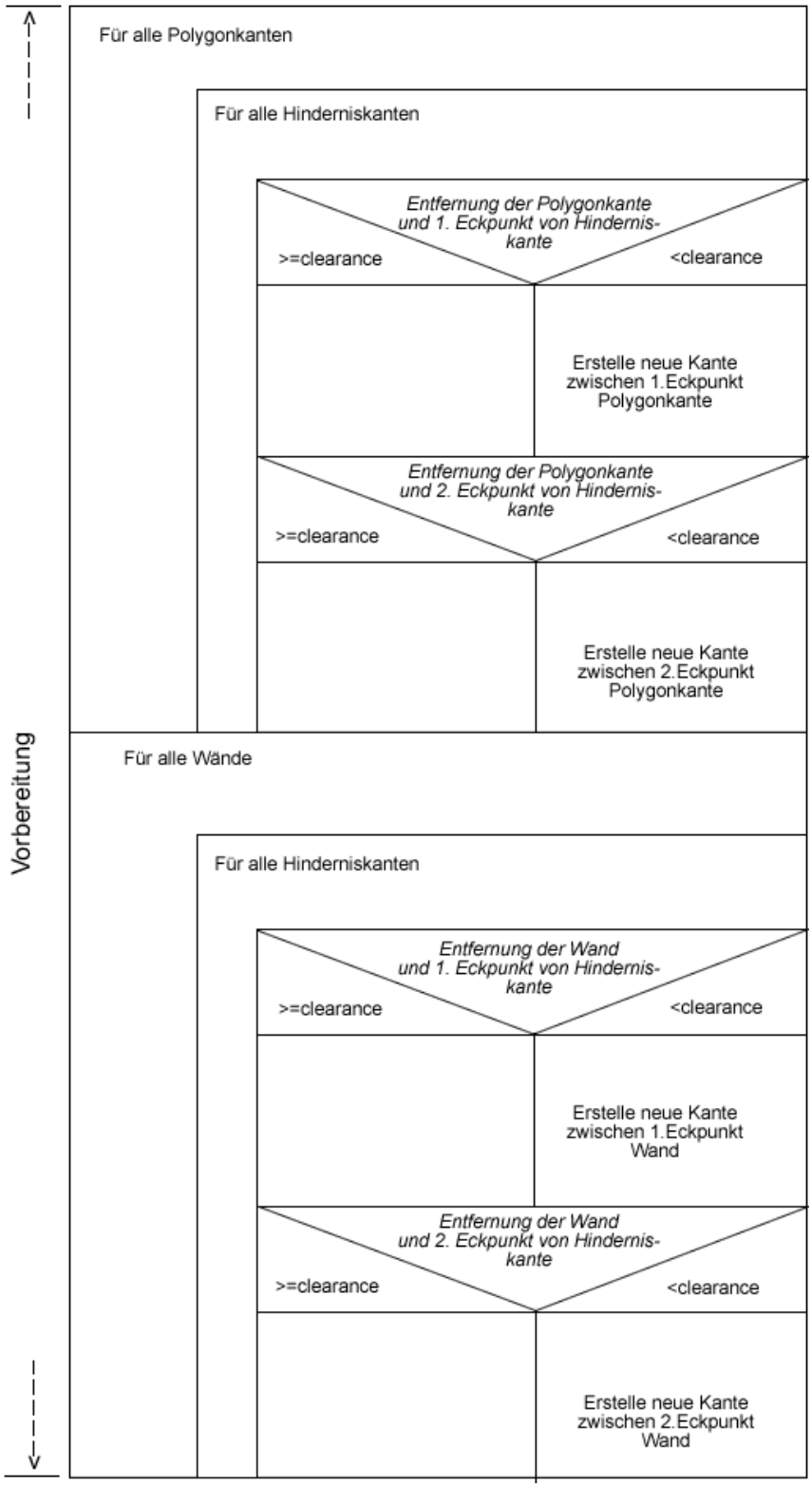


Abb. 39: Graph und Adjazenzliste

Die Basisstruktur bildet die Liste aller Knoten. Für jeden Knoten wird eine Liste der Nachfolger entlang gerichteter Kanten abgespeichert.

Abbildung 40 zeigt noch einmal das Struktogramm für die gesammte Zellzerlegung.


Vorbereitung
Für alle Polygonkanten
Für alle Hinderniskanten
Entfernung der Polygonkante und 1. Eckpunkt von Hindernis-kante
>=clearance
<clearance
Erstelle neue Kante zwischen 1.Eckpunkt Polygonkante
Entfernung der Polygonkante und 2. Eckpunkt von Hindernis-kante
>=clearance
<clearance
Erstelle neue Kante zwischen 2.Eckpunkt Polygonkante
Für alle Wände
Für alle Hinderniskanten
Entfernung der Wand und 1. Eckpunkt von Hindernis-kante
>=clearance
<clearance
Erstelle neue Kante zwischen 1.Eckpunkt Wand
Entfernung der Wand und 2. Eckpunkt von Hindernis-kante
>=clearance
<clearance
Erstelle neue Kante zwischen 2.Eckpunkt Wand

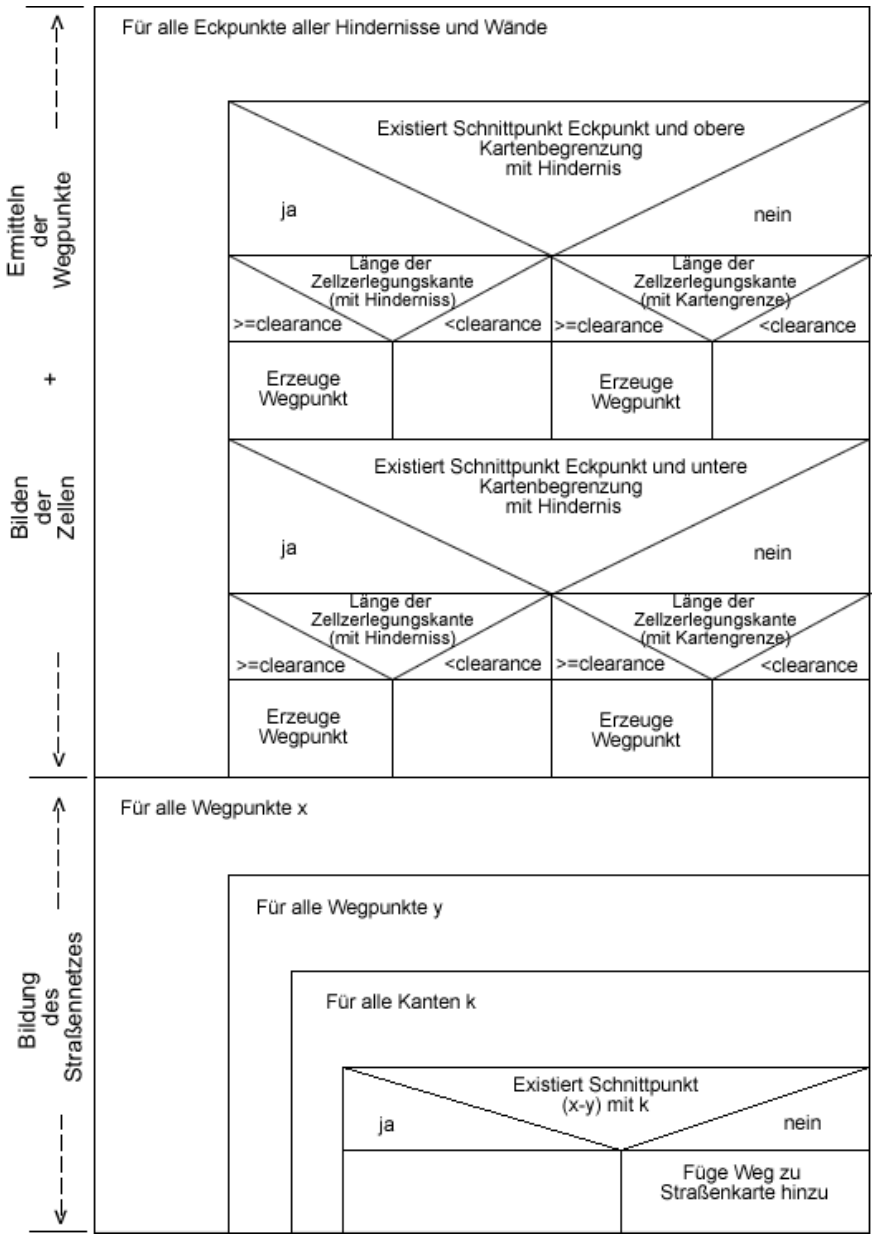


Abb. 40 : Struktogramm Zellzerlegung

4.8. Qualität des gefundenen Wegs

Mithilfe des vorhergehenden Zellzerlegungsalgorithmus wird ein Pfad in einem 2D Raum, bestehend aus frei befahrbaren Flächen und Hindernissen, gefunden. Im Anschluss daran gilt es, die Qualität des gefundenen Weges zu bewerten und mögliche Problemsituationen zu lösen. Folgende Szenarien können dabei auftreten.

Szenario 1 – Wegpunkt zu nah an Hinderniss

Anders als bei der Pfadplanung mit Voronoidiagrammen, kann es bei der Zellzerlegung dazu kommen, dass sich Wegpunkte sehr nah an einem Hindernis bzw. Wand befinden. Abbildung 41 veranschaulicht diesen Fall und macht das Problem sehr deutlich. Zwei Wegpunkte (rot gefärbt) liegen sehr nahe an der Wand und sind für den Roboter nicht erreichbar. Es sind Karten vorstellbar, wo Wegpunkte sogar noch näher an Hindernissen liegen.

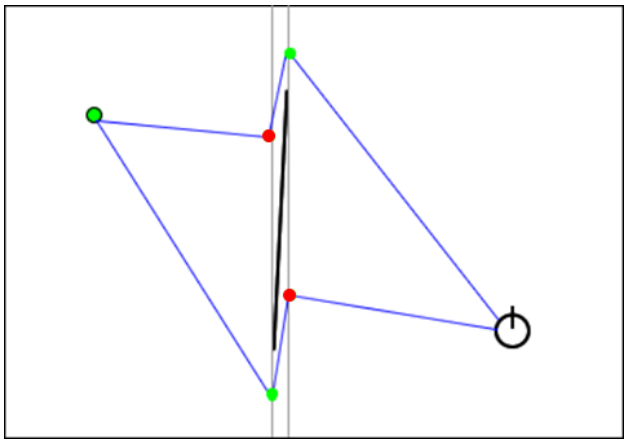


Abb. 41 : Wegpunkt zu nah am Hindernis

Die Lösung ist recht trivial. Wegpunkte, die sich zu nahe an Hindernissen befinden sind recht einfach zu lokalisieren. Nachdem man sie ausfindig gemacht hat, werden sie einfach im rechten Winkel zum Hindernis wegbewegt. Abbildung 42 zeigt die Karte mit dem Straßennetz nach der Korrektur. Eine Navigation sollte nun kein Problem darstellen.

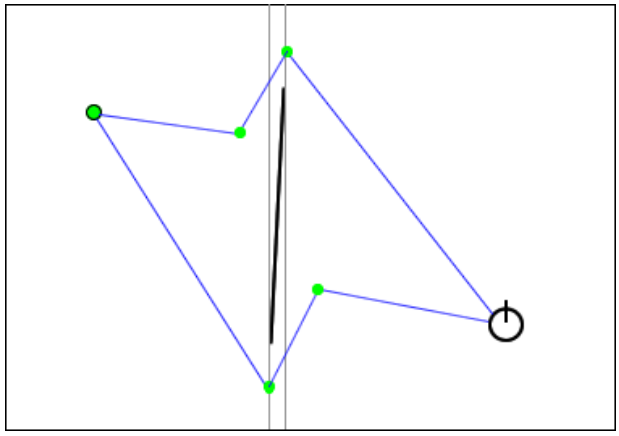


Abb. 42 : Wegpunkt zu nah am Hindernis - Lösung

Szenario 2 – Weg zu nah an Hinderniss

Ein weiteres Problem stellen Wege dar, die zu nah an Hindernissen vorbeiführen. Das kann dazu führen, dass die Motoren blockieren und eine weitere Navigation unmöglich wird. Der fol-

gende Wegpunkt erscheint nicht erreichbar zu sein. Abbildung 43 veranschaulicht diesen Fall, wobei nur der betroffene Pfad zu sehen ist und der kritische Teil rot eingefärbt ist.

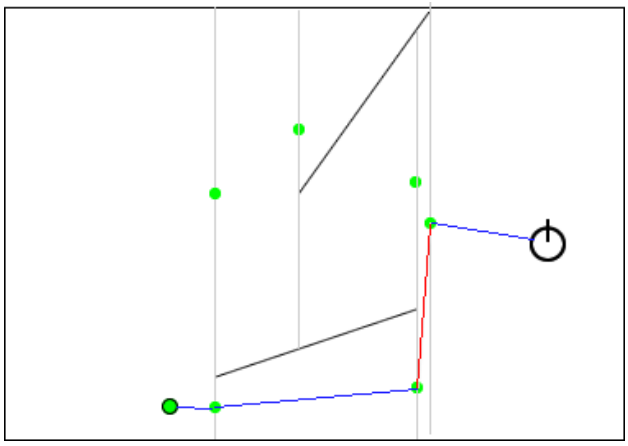


Abb. 43 : Weg zu nah am Hindernis

Dieses Problem löst das Behavior *sfAvoidCollision(),* welches Hindernissen ausweicht und versucht zu umfahren. Jedoch wurde nach einigen Tests bemerkt, dass *sfAvoidCollision()* keine Wände erkennt, wenn der Simulator im selben Winkel der Wand darauf zu fährt.

Szenario 3 – Dreieckige Zellen benötigen zusätzlichen Wegpunkt

Abbildung 44 zeigt deutlich, dass man bei der Zellzerlegung nicht immer mit den ermittelten Wegpunkten, bestehend aus den Mittelpunkten der Zellzerlegungskanten, auskommt. Es existiert ein Weg vom Roboter zum Zielpunkt, jedoch wird dieser nicht gefunden, weil ein wichtiger Wegpunkt fehlt.

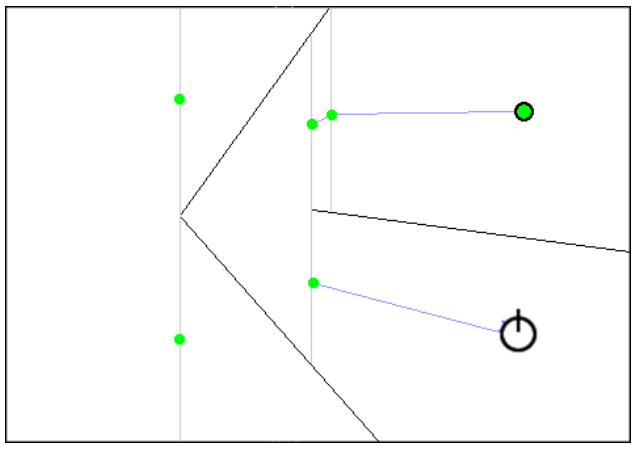


Abb. 44 : Dreieckige Zellen benötigen Wegpunkt

Diese Situationen sind jedoch recht leicht zu identifizieren und genauso leicht zu lösen. Bei der Zellzerlegung wird überprüft, ob eine dreieckige Zelle entsteht und ein Wegpunkt in dem geometrischen Mittelpunkt der Zelle hinzugefügt. Abbildung 45 zeigt, dass nach Hinzufügen des Wegpunktes in den Mittelpunkt der dreieckigen Zelle, der Pfad gefunden wird.

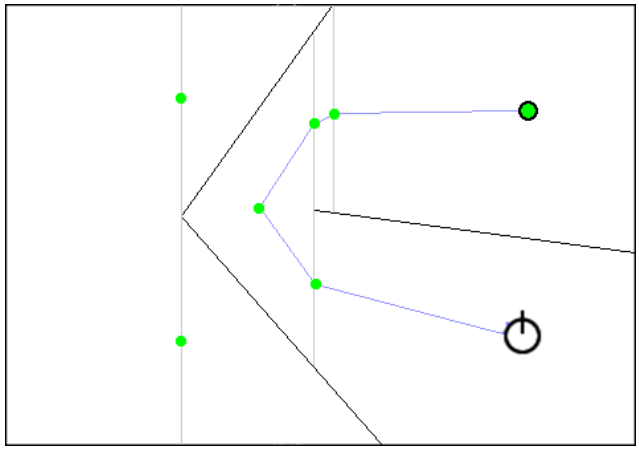


Abb. 45 : Dreieckige Zellen benötigen Wegpunkt - Lösung

Szenario 4 – Parallel zur Y-Achse verlaufende Kanten

Ein weiteres Problem stellt sich, bei der Existenz von parallel zur Y-Achse verlaufende Kanten, dar. In Abbildung 46 ist zu sehen, dass bei solchen Kanten nicht genügend Wegpunkte gefunden werden und es dazu kommen kann, dass befahrbare Pfade nicht gefunden werden.

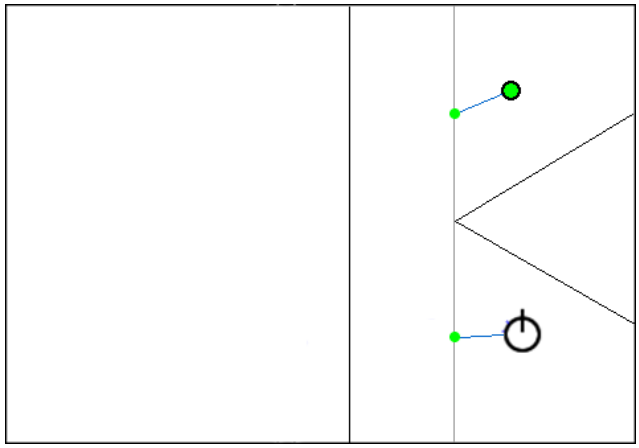


Abb. 46 : Senkrechte Kanten

Lavalle beschrieb dieses Problem schon in seinem Buch *Planning Algorithms*. Er schlägt vor, die gesamte Karte um einen kleinen Winkel zu drehen, so dass es fast ausgeschlossen ist, dass sich danach senkrecht zur Y-Achse verlaufende Kanten darin befinden.

“This usually means that if all of the data points are perturbed by a small amount in some random direction, the probability that the special case remains is zero. Since a vertical edge is no longer vertical after being slightly perturbed, it is not in general position. The general position assumption is usually made because it greatly simplifies the presentation of an algorithm.”
[Lavalle S.255]

Eine interessante Möglichkeit, die jedoch einen hohen Anteil an Entwicklungsarbeit verschlingen würde. Eine andere Möglichkeit, die auf dieser Idee basiert, ist betreffende Kanten virtuell während der Zellzerlegung zu drehen. Abb… zeigt die leicht gedrehten Kanten und die daraus resultierenden Wegpunkte, die den existierenden Pfad ermitteln lassen.

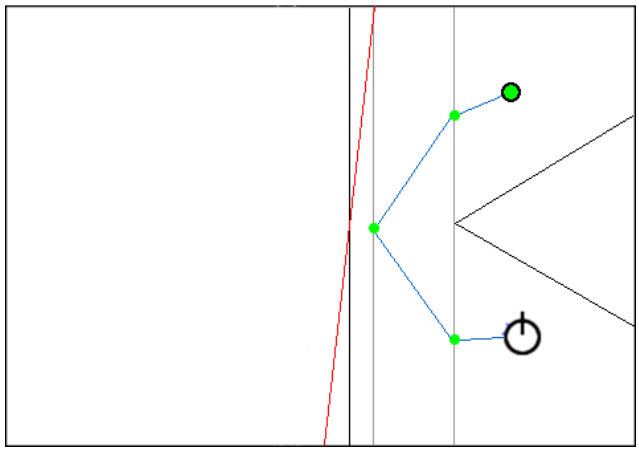


Abb. 47 : Senkrechte Kanten - Lösung

4.9. Ablauf des A* Algorithmus

Unter 3.5.4 wurde der A* Algorithmus bereits an einem Beispiel veranschaulicht. Nun wird dieser in die Applikation implementiert. Dazu sind folgende Datenstrukturen nötig.

Datenstruktur der Strassenkarte

Die Datenstruktur für die Straßenkarte setzt sich wie folgt zusammen.

Jeder Punkt und seine Liste von erreichbaren Punkten sind in einer Datenstruktur Reachable gespeichert. Alle diese Datenstrukturen sind wiederum in einer weiteren Liste Roadmap gespeichert. Diese Struktur repräsentiert somit die gesamte Straßenkarte bzw. Roadmap. Diese Struktur ist nach dem Muster der Adjazenzliste aufgebaut, die für solche Abhängigkeitsbeziehungen prädestiniert ist.

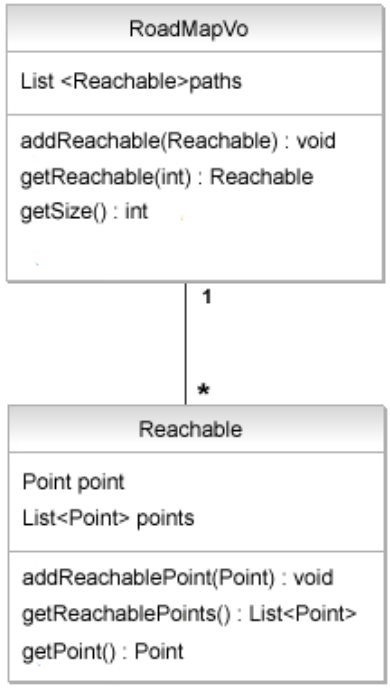


Abb. 48 : Klassendiagramm der Roadmap

Datenstruktur der Agenda

Die Datenstruktur der Agenda ist ähnlich der des Straßennetzes. Hierbei werden alle Wege in einer Liste gespeichert. Zusätzlich erhält jeder Weg noch einen Wert für seine Länge, also die Entfernung vom Anfangs- bis zum Endpunkt des Weges. Wird ein Punkt hinzugefügt, wird die Länge neuberechnet. Die Liste für die Agenda enthält dann diese Weg-Objekte.

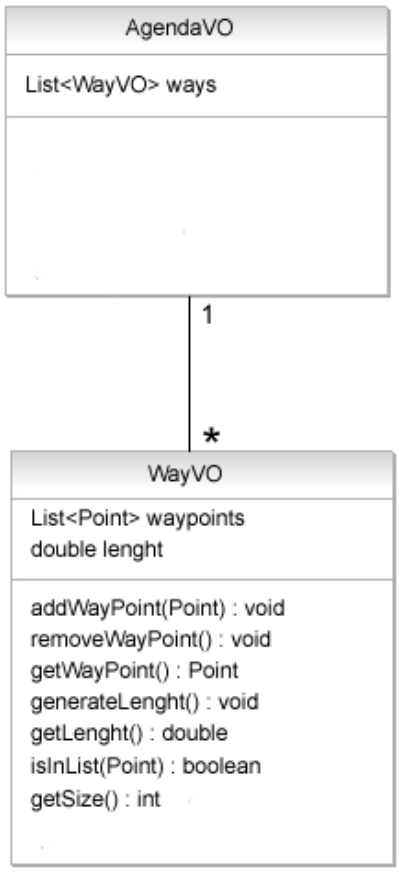


Abb. 49 : Klassendiagramm der Agenda

Zu Beginn des Algorithmus wird das erste Element der Straßenkarte entnommen, welches gleichzeitig den Startpunkt und die Liste, der von ihm erreichbaren Punkte repräsentiert.

Nun wird in der Agenda jeweils für jeden erreichbaren Punkt vom Startpunkt aus eine separate Liste erstellt. Diese bestehen zunächst aus 2 Elementen, dem Start- und einem der erreichbaren Punkte. Nun wird die Agenda durchsucht, und die Liste mit dem geringstem Aufwand, das heißt geringsten Entfernung zum aktuellen Punkt plus der Entfernung zum Zielpunkt, entnommen. Die Entfernung zum Zielpunkt ist während der Berechnung noch nicht bekannt, daher wird, wie schon erwähnt eine Schätzfunktion, in diesem Fall die Luftlinienheuristik angewendet. Das bedeutet, dass als mögliche Entfernung zum Zielpunkt die Länge der Luftlinie zu Hilfe genommen wird. Die Länge jeden Weges kann anhand der jeweiligen Eigenschaft des Objektes abgerufen werden.

Die entnommene Liste wird expandiert, dies bedeutet, dass der letzte Punkt der Liste genommen wird und deren erreichbaren Punkte aus der Reachablestruktur gelesen werden. Für jeden

Punkt wird überprüft, ob dieser bereits schon einmal expandiert wurde. Falls das der Fall ist wird er verworfen. Ist er nicht in der Agenda enthalten wird eine neue Liste erstellt und wieder in die Agenda angehängt. Dieser Ablauf wird solange wiederholt, bis der zu expandierende Punkt, der Zielpunkt ist.

Ist der letzte Punkt der entnommenen Liste der Zielpunkt, so hat man den kürzeste Pfad gefunden und der Algorithmus kann den Pfad ausgeben und terminieren. Dieser wird dann an den Navigationsalgorithmus zur weiteren Verarbeitung weitergegeben.

4.10. Entwicklung dynamischer Bibliotheken (DLLs) für Saphira

Oftmals reichen die Möglichkeiten, die Saphira bietet nicht aus um Probleme mit speziellen Anforderungen zu lösen. Daher besteht die Möglichkeit, dynamische Bibliotheken in Saphira zu laden und damit die Funktionalität zu erweitern.

Es muss eine Funktion **sfLoadInit**() definiert werden. Diese wird nach dem Laden in Saphira ausgeführt. Desweiteren kann eine Funktion **sfLoadExit** () definiert werden, dessen Code beim Beenden von Saphira ausgeführt wird. Unter Microsoft Visual C++ sieht das folgendermaßen aus:

```
#include "saphira.h"

__declspec(dllexport) void sfLoadInit() {
        sfMessage("Dll geladen");
}

__declspec(dllexport) void sfLoadExit() {
        sfMessage("Dll entladen");
}
```

Wichtig ist, dass die Headerdatei saphira.h geladen wird und die Bibliothek SF.lib dem Projekt hinzugefügt wird, da diese wichtige Funktionen von Saphira bereitstellt. Beispielsweise gibt sfMessage eine Meldung im Saphirafenster aus.

- sfAddEvalFn(char *name, void *fn, int rtype, int nargs, ...)

Mit diesem Befehl können Funktionen in Colbert bekannt gemacht werden.
Mit *name wird der Name der Funktion angegeben, durch den diese in Colbert aufgerufen werden kann.
Der zweite Parameter ist der Name der Funktion in C. Als dritter Parameter wird der Typ des Rückgabewertes angegeben. Danach folgt die Anzahl der Übergabeparameter und im folgenden Paarweise die Namen der Parameter und deren Werte.

- sfAddEvalVar(char *name, int type, (fvalue *)&cvar)

Mit diesem Befehl können Variablen in Colbert bekannt gemacht werden. Mit *name wird der Name der Variablen angegeben mit dem diese in Colbert angesprochen werden kann. Als zweiter Parameter wird der Typ der Variablen angegeben und als letzter Parameter wird die Referenz der Variablen übergeben.

- sfAddEvalConst(char *name, int type, (fvalue)val)

Mit diesem Befehl können Konstanten in Colbert bekannt gemacht werden. Die Parameter sind äquivalent zum Befehl für Variablen. Nur als letzter Parameter muss ein Wert für die Konstante angegeben werden.

- sfAddEvalStruct(char *name, int size, (char *)&s, int num, ...)

Mit diesem Befehl können Strukturen in Colbert bekannt gemacht werden. Auch bei diesem Befehl wird mit *name der Name der Struktur übergeben, durch den die Struktur in Colbert angesprochen werden kann. Als zweiter Parameter wird die Größe der Struktur angegeben.

5.

IMPLEMENTIERUNG

5.1. Der Pilot

Als Pilot wird das Modul, das in Saphira läuft, bezeichnet, welches für den Ablauf der Fahrt des Roboters zuständig ist. Der Pilot besteht aus zwei Komponenten. Eine Komponente ist eine in C geschriebene dynamische Bibliothek, die Funktionen für die Socketverbindung und die Kommunikation mit dem Pathplanner bereitstellt. Die zweite Komponente ist eine in Colbert geschrieben Activity, welche die Funktionen, die die C Bibliothek bereitstellt, nutzt.

5.1.1. Pilot.act

Im Piloten laufen folgende Aktivitäten ab. Zu Beginn wird *sfAvoidCollision()* gestartet.

```
act pilot{
	start connectpp();
	while(1){
		start getValue();
		if(newpos){
			start moveToPoint;
			sfSMessage("goal reached");
			start sendValue;
			newpos=0;
		}
	}
}
```

Die Aktivität *pilot()* ist die Hauptaktivität, die auch zu Begin gestartet wird. Zuerst wird die Unteraktivität *connectpp()* aufgerufen, die auf die Verbindung des Clients, dem Pathplanner, wartet. Sobald dies geschehen ist, wird eine Endlosschleife aufgerufen, die Befehle mit *getValue()* vom Pathplanner empfängt und ausführt. Falls ein *goto* Befehl empfangen wurde, wird die Unteraktivität *moveToPoint()* aufgerufen, welche den Roboterserver dann zum übergebenen Punkt navigiert. Danach wird die Ausführung des Befehls mit *sendValue()* bestätigt.

```
act moveToPoint{
	start sfGoToPos(100, goal, 300) priority 0;
}
```

Die Aktivität *moveToPoint()* startet ein *sfGoToPos()* Behavior. Der erste Parameter bestimmt die Geschwindigkeit, mit der der Simulator fahren soll, der zweite den Zielpunkt und der dritte Parameter gibt den Zielradius an, ab dem das Ziel als erreicht gilt.

```
act connectpp{
	sfconnect();
}
```

Die Aktivität *connectpp()* ruft die Funktion *sfconnect()* aus der DLL-Bibliothek auf, welche den Socketserver startet und auf die Verbindung eines Socketclients wartet. In dieser Zeit ist Saphira blockiert.

```
act sendValue{
	sendmessage();
}
```

sendValue() ruft die Funktion sendmessage aus der DLL-Bibliothek auf, welche eine Nachricht über die Socketverbindung an den Pathplanner sendet.

Nachdem die Aktivität in Saphira geladen wurde, wird sie mit start pilot gestartet. Der erste Befehl startet die Routine für die Herstellung der Verbindung zur Java-Applikation. Die Aktivität blockiert solange, bis eine Verbindung erfolgreich zustande gekommen ist. Danach wird mit *while(1)* eine Endlosschleife gestartet. In jedem Schleifendurchlauf wird mit der Aktivität *getValue* ein Befehl aus der Socketverbindung gelesen. Daraufhin wird geprüft ob ein neuer Wegpunkt existiert. Falls das der Fall ist, wird die Aktivität *moveToPoint* gestartet, welche den Roboter zum Wegpunkt navigiert. Danach wird mit der Aktivität *sendValue* eine Meldung zur Java-Applikation gesendet. Nun wird die Schleife aufs neue durchlaufen und die Abfolge der Befehle abgearbeitet.

5.1.2. Pilot.dll

Die wichtigste Funktion der dynamischen Bibliothek Pilot.dll ist die Bereitstellung der Socketverbindung. Die Funktionen connectpp() und *CreateSocketConnection()* starten den Socketserver und warten darauf, dass sich ein Client verbindet. Für die Verbindung wird der Port 4444 benutzt.

```
boolean connectpp(){
	connection_status=2;
	if(createSocketConnection(1)){
		sfSMessage("socketserver is running...");
		sc = accept(s,0,0);
		if(sc==-1){
			sfSMessage("client could not connected...");
		}else{
			sfSMessage("client connected...");
		}
		connection_status=1;
	}else{
		sfSMessage("socketserver could not run...");
		connection_status=0;
	}
}

boolean createSocketConnection(int typ){
	WSADATA wsa;
	int port=0;
	if(typ==1){
		port=4444;
	}else{
		port=4445;
	}


	//initialize Windows Socket


	if(WSAStartup(MAKEWORD(1,1),&wsa)){
		sfSMessage("socket could not be initialized...");
		return false;
	}else{
		sfSMessage("socket initialized...");
	}

	//socket erstellen
	if(typ==1){
```

```
		s = socket(AF_INET,SOCK_STREAM,0);
		if(s==-1){
			sfSMessage("socket could not be created...");
			return false;
		}else{
			sfSMessage("socket is created...");
		}
		server.sin_family = AF_INET;
		server.sin_addr.s_addr = INADDR_ANY;
		server.sin_port = htons(port);

		if(bind(s,&server,sizeof(server))==-1){
			sfSMessage("could not bind...");
			return false;
		}else{
			sfSMessage("socket binded...");
		}

		if(listen(s,1)==-1){
			sfSMessage("could not listen...");
			return false;
		}else{
			sfSMessage("socket is listening...");
		}
	}else{
		s1 = socket(AF_INET,SOCK_STREAM,0);
		if(s1==-1){
			sfSMessage("socket could not be created...");
			return false;
		}else{
			sfSMessage("socket is created...");
		}
		server.sin_family = AF_INET;
		server.sin_addr.s_addr = INADDR_ANY;
		server.sin_port = htons(port);

		if(bind(s1,&server,sizeof(server))==-1){
			sfSMessage("could not bind...");
			return false;
		}else{
			sfSMessage("socket binded...");
		}

		if(listen(s1,1)==-1){
			sfSMessage("could not listen...");
			return false;
		}else{
			sfSMessage("socket is listening...");
		}
	}
	return true;
}
```

Die Funktion *sfLoadInit()* wird beim Laden der Bibliothek ausgeführt und macht Funktionen und einige Variablen für die Aktivität Pilot.act bekannt, wie zum Beispiel die anzufahrenden Punkte, die jeweils in der Variable *goal* gespeichert werden.

```
__declspec(dllexport) void sfLoadInit() {
	sfAddEvalVar("connection_status", sfINT, &connection_status);
	sfAddEvalVar("goal",sfPTR,&goal);
	sfAddEvalVar("newpos", sfINT, &newpos);
	sfAddEvalVar("rcom", sfINT, &rcom);
	sfAddEvalFn("ppstatus", printStatus, sfVOID, 0);
	sfAddEvalFn("sfconnect", connectpp,sfVOID,0);
```

```
        sfAddEvalFn("sendmessage", sendMessage,sfVOID,0);
        sfAddEvalFn("recmessage", receiveMessage,sfVOID,0);
        sfAddEvalFn("moveforward", moveforward,sfVOID,0);
        startBehaviors();
        initProcs();
}
```

5.2. Ablauf des Piloten

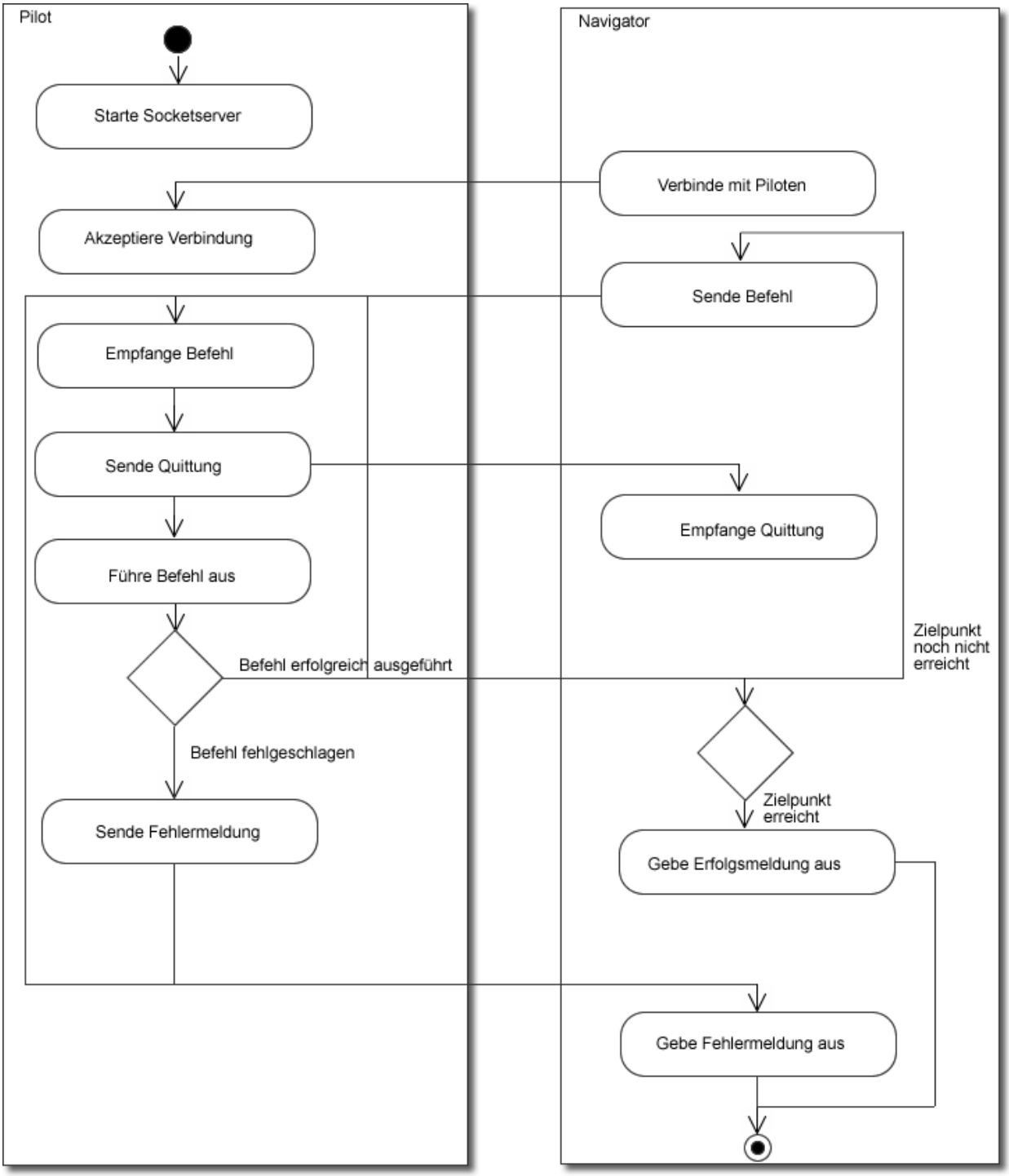


Abb. 50 : Aktivitätsdiagramm des Piloten und Navigators

Zunächst wird eine Funktion zum Verbinden mit der Java Applikation gestartet. Erst wenn eine Verbindung besteht wird die Navigationsschleife betreten, welche solange ausgeführt wird, bis das Ziel erreicht wurde. Erst wird ein Befehl über die Socketverbindung geholt. Dieser wird dann ausgeführt und bei erfolgreicher Ausführung wird eine entsprechende Nachricht an die Java Applikation gesendet. Bei auftreten eines Konfliktes wird ebenfalls eine entsprechende Nachricht gesendet und erforderliche Maßnahmen zur Fehlerbehandlung eingeleitet. Sobald der Zielpunkt erreicht wurde, wird die Schleife verlassen und die Activity erfolgreich terminiert.

5.3. Komponenten

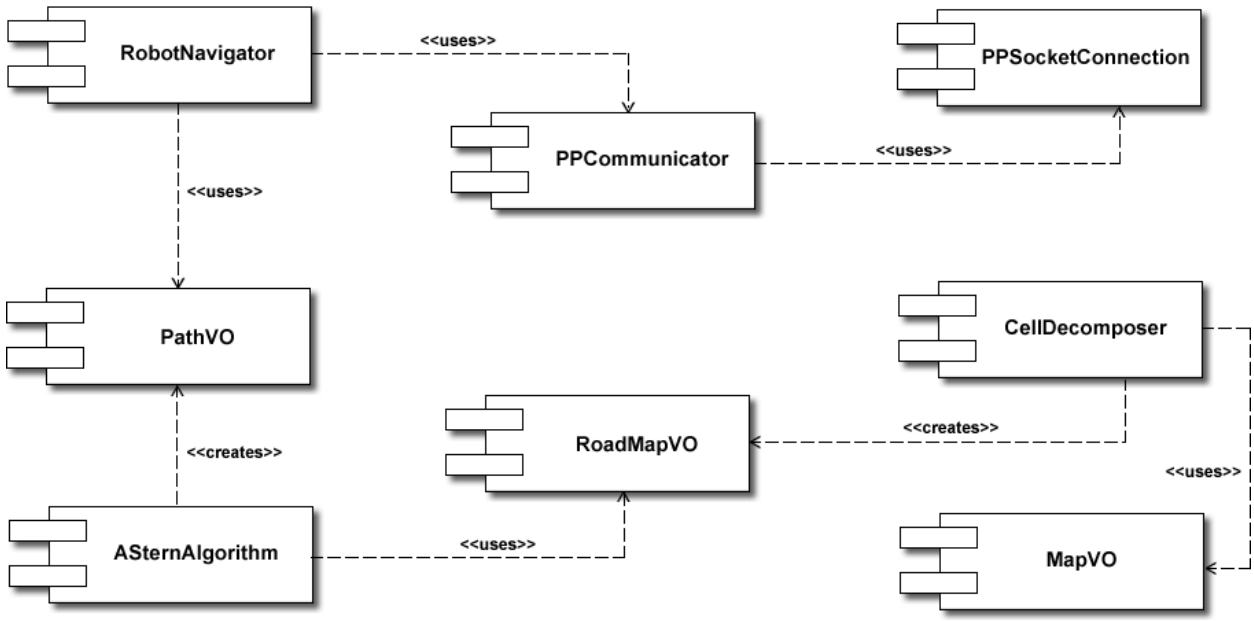


Abb. 51 : Komponentendiagramm des Pathplanner Moduls

In Abbildung 51 sieht man einen Ausschnitt wichtiger Komponenten des Java Moduls.

Der **CellDecomposer** erzeugt mithilfe der **MapVO**(Karte) die **RoadMapVO**(Straßenkarte), welche daraufhin vom **ASternAlgorithm** verwendet wird um das **PathVO**(Pfad) zu errechnen.

Der **RobotNavigator** wird aktiv, wenn ein Pfad mittels Zellzerlegung einer Karte berechnet wurde und der Befehl zum Navigieren betätigt wurde. Der **RobotNavigator** benutzt zur Navigation ein **PathVO** das einen Pfad durch die Karte repräsentiert. Der **RobotNavigator** kommuniziert mittels des **PPCommunicators** mit Saphira. Der **PPCommunicator** wandelt die Steuerungsbefehle des **RobotNavigators** in Bytefolgen um die dann mittels einer **PPSocketConnection**(Socket Verbindung) und dem *Robot Control Protokoll* an das C-Modul und die Aktivität des Piloten in Saphira weitergeleitert werden und den Piloten damit steuert.

5.4. CellDecomposer

Der CellDecomposer ist das Kernstück der Pfadplanung und ermittelt aus einer vom Benutzer erzeugten Karte das Straßennetz. Da er aus mehr als eintausend Codezeilen besteht, werden nur wichtige Methode näher erläutert.

Die Methode *generateNearMapObjects()* ermittelt Objekte auf der Karte, also Wände und Polygone, die sich zu nahe aneinander befinden und verbindet diese mit virtuellen Kanten, sodass dort später keine Wegpunkte sowie Pfade gefunden werden können. Diese Methode läuft in 2 Phasen ab. Zunächst wird die Liste aller Polygone auf der Karte gewählt und für jeden Eckpunkt jedes Polygons geprüft, ob sich Kanten der Karte zu nah befinden. Dazu wird ein *Line2D.Double* Element erzeugt, welches mit der Funktion *ptLineDist()* den Abstand eines Punktes zu einer Kante zurückgibt. Ist dieser Abstand kleiner als die Programmkonstante *distance,* welche den Mindestabstand von Objekten beschreibt um vom Roboter durchfahren zu werden, kann zwischen ihnen nicht navigiert werden und es wird eine virtuelle Kante erzeugt.

```
private void generateNearMapObjects(){
              PolygonVO poly;
              WallVO wall;
              for(int i=0;i<map.getPolygons().size();i++){
                     poly=(PolygonVO)map.getPolygons().get(i);
                     for(int y=0;y<poly.getSize();y++){
                            Point from = poly.getPoint(y);
                            for(int j=0;j<kant.size();j++){
                                   Point tschnitt;
                                   Line2D.Double el=new
                     Line2D.Double(kant.get(j).getFrom(),kant.get(j).getTo());
       if(!from.equals(kant.get(j).getFrom())&&!from.equals(kant.get(j).getTo(
))){
                                          if(el.ptLineDist(from)<clearance
```

5.5. PPSocketConnection

Um dem PPCommunicator eine Kommunikation mit Saphira zu ermöglichen bedarf es einer Socketverbindung. Diese wird durch PPSocketConnection initialisiert. Diese ist von der Klasse Socket aus dem java.net Packet abgeleitet. Beim Aufruf des Konstruktors wird der Konstruktor der Superklasse mit super aufgerufen. Dieser bekommt IP Adresse und Port übergeben. Der Konstruktor wirft einige Ausnahmen.

5.6. PPCommunicator

Der PPCommunicator erzeugt mithilfe der PPSocketConnection eine Socketverbindung mit Saphira. Dies erfolgt mit der folgenden Methode.

```
public boolean createConnection() throws UnknownHostException, IOException{
       try {
              ppconnection = new PPSocketConnection(ip_adress,port);
              System.out.println("connected to host : "+ip_adress+" on port : " +port);
              connected = true;
              return true;
       } catch (ConnectException e) {
              System.err.println("can't connect host : "+ip_adress);
              e.printStackTrace();
              return false;
```

```
        } catch (UnknownHostException e) {
              System.err.println("can't find host : "+ip_adress);
              return false;
        } catch (NoRouteToHostException e) {
              System.err.println("can't connect to server !");
              return false;
        } catch (IOException e) {
              return false;
        }
}
```

Hier wird zunächst ein PPSocketConnection Objekt erstellt, welches die IP Adresse und den Port für die Verbindung übergeben bekommt. Das Boolean Attribut wird auf true gesetzt, damit andere Methoden später überprüfen können, ob eine Verbindung zu Saphira besteht und Befehle gesendet werden können. Beim Verbinden zu Saphira über Sockets können einige Ausnahmen auftretet, die mithilfe von try und catch abgefangen werden müssen.

Folgende Ausnahmen können auftreten:

ConnectException

Ist am Zielport des Hosts kein Server vorhanden, kann diese Ausnahme auftreten.

UnknownHostException

Diese Ausnahme tritt dann auf, wenn der Hostname nicht aufgelöst werden konnte, wenn der angegebene Hostname nicht bekannt ist.

NoRouteToHostException

Ist der Host beim Aufbauversuch der Verbindung nicht erreichbar, wird diese Ausnahme geworfen. Dies kann passieren, wenn die Verbindung durch eine Firewall behindert wird oder wenn der Host nicht online ist.

IOException

Diese Ausnahme signalisiert einen allgemeinen Ein-/Ausgabefehler, der verschiedene Ursachen haben kann.

Die gesamte Kommunikation läuft über den PPCommunicator. Im folgenden Absatz werden die beiden wichtigsten Methoden des PPCommunicators erläutert.

```
private void sendCommand(String command){
      if(connected){
            try{
                  BufferedWriter out = new BufferedWriter(new OutputStreamWri-
                  ter(ppconnection.getOutputStream()));
                  out.write(command);
                  out.flush();
                  System.out.println("data successfully sent...");
            }catch(Exception e){
                  System.err.println(e);
            }
      }
}
```

Das Senden von Befehlen erfolgt über einen BufferedWriter, der mithilfe eines OutputStreamWriters in den OutputStream der Socketverbindung schreibt. Bevor Befehle gesendet werden können, muss natürlich eine Socketverbindung stehen. Dies wird mithilfe des Boolean Wertes connected überprüft. Wichtig ist hierbei auch, dass hierbei Ausnahmen abgefangen werden und BufferedWriter nach jedem Schreibvorgang geflusht wird. Das bedeutet, dass alle Werte aus dem BufferedWriter gelöscht werden, so dass er neue Werte schicken kann.

```
public String receiveMessage(){
	String message="";
	if(connected){
		try{
			BufferedReader in = new BufferedReader(new InputStreamRead-
			er(ppconnection.getInputStream()));
			message=in.readLine();
			System.out.println("data successfully recieved..."+message);
		}catch(Exception e){
			System.err.println(e);
		}
	}
	return message;
}
```

Das Empfangen von Nachrichten erfolgt analog zum Senden von Befehlen. Erst wird durch die Boolean Variablen connected überprüft, ob eine Socketverbindung besteht. Danach wird ein BufferedReader erzeugt, der mithilfe eines InputStreamReaders aus dem InputStream der Socketverbindung liest. Auch hierbei kommt der gesamte Quellcode in einen try Block um mögliche Ausnahmen abzufangen.

5.7. RobotNavigator

Der Robot Navigator ist als Thread implementiert, damit während der Navigation der Rest der Applikation nicht blockiert wird, und zum Beispiel in der Karte gescrollt werden kann. Der Robot Navigator ist hauptsächlich dafür zuständig mithilfe der Liste von Wegpunkten dem Piloten Navigationsbefehle zu senden und damit den Roboter zu navigieren. Dafür benutzt er den Robot Communicator, der die Socketverbindung herstellt und die Befehle in RCP Befehle wandelt und diese mithilfe der Socketverbindung an den Piloten schickt. Sobald der Robotnavigator eine Fehlermeldung erhält, ist die Navigation gestoppt.

Im folgenden Absatz wird die Methode navigateRobot näher erläutert. Diese bekommt den Pfad über das Attribut path übergeben.

```
public void navigateRobot(PathVO path){
	if(firstway){
		communicator.setPos(path.getPoint(0).y*10,path.getPoint(0).x*10,0);
		firstway=false;
	}
	for(int i=1;i<path.getSize();i++){
		actpoint=cocalc.generateMapCoordinates(path.getPoint(i));
		if(communicator.goTo(actpoint.x*10, actpoint.y*10,0)){
		//if(communicator.goTo(actpoint.y*10, actpoint.x*10,0)){
			System.out.println("Goal reached");
		}else{
			System.out.println("Goal not reached");
		}
	}
	System.out.println("Yeah...you reached the Goal!!!!");
	//TODO GoalReached?
	//TODO mit PPCommunicator.receiveMessage auf Antwort warten
```

```
}
```

Zuerst wird geprüft ob es sich um die erste Navigation der aktuellen Laufzeit handelt. Wenn dies der Fall ist muss der Roboter in der Karte in Saphira platziert werden, dazu wird die Methode communicator.setpos ausgeführt, welche mittels des Communicators den Roboter in Saphira positioniert. Danach wird jeder Wegpunkt aus dem Pfad entnommen und über den Communicator der Befehl goTo ausgeführt. Wurde der letzte Wegpunkt erfolgreich angefahren ist die Navigation erfolgreich und die Methode wird verlassen.

5.8. ASternAlgorithm

Die Klasse ASternAlgorithm ist dafür verantwortlich, die kürzeste Route in der Straßenkarte zu finden. Zunächst wird dem Konstruktor die Straßenkarte übergeben. Der eigentliche A* Algorithmus läuft innerhalb der Methode generateRoute() ab.

```
public PathVO generateRoute(){
Point goal=routemap.getReachable(routemap.getSize()-1).getPoint();
expandedPoints = new ArrayList<Point>();
Reachable start = routemap.getReachable(0);
List<Point> reachablepoints = start.getReachablePoints();
expandedPoints.add(start.getPoint());
for(int i=0;i<reachablepoints.size();i++){
	WayVO tl=new WayVO();
	tl.addWayPoint(start.getPoint());
	tl.addWayPoint((Point) reachablepoints.get(i));
	agenda.add(tl);
}
while(success==false){
	int shortest=0;
double templenght=Math.sqrt((goal.x-agenda.get(0).getWayPoint(agenda.get(0).getSize()-
1).x)*(goal.x-agenda.get(0).getWayPoint(agenda.get(0).getSize()-1).x)+(goal.y-
agenda.get(0).getWayPoint(agenda.get(0).getSize()-1).y)*(goal.y-
agenda.get(0).getWayPoint(agenda.get(0).getSize()-1).y));
double minentf=agenda.get(0).getLenght()+templenght;
for(int i=1;i<agenda.size();i++){
	double tmplenght=Math.sqrt((goal.x-
agenda.get(i).getWayPoint(agenda.get(i).getSize()-1).x)*(goal.x-
agenda.get(i).getWayPoint(agenda.get(i).getSize()-1).x)+(goal.y-
agenda.get(i).getWayPoint(agenda.get(i).getSize()-1).y)*(goal.y-
agenda.get(i).getWayPoint(agenda.get(i).getSize()-1).y));
	System.out.println("Temp Länge : "+tmplenght);
	if((agenda.get(i).getLenght()+tmplenght)<minentf){
		minentf=agenda.get(i).getLenght();
		shortest=i;
	}
}
System.out.println("Kürzester Weg : "+shortest+" : "+minentf);
newlist=agenda.get(shortest);
agenda.remove(shortest);
Point lastpoint=(Point)newlist.getWayPoint(newlist.getSize()-1);
if(lastpoint.equals(goal)){
	System.out.println("Route gefunden");
	success=true;
}
System.out.println("ExpandedPoints "+expandedPoints);
if(expandedPoints.contains(lastpoint)){
	System.out.println("Enthält letzten Punkt "+lastpoint);
	continue;
}
expandedPoints.add(new Point(lastpoint.x,lastpoint.y));
System.out.println("Letzter Punkt hinzugefügt "+lastpoint);
```

```
System.out.println("Letzter Punkt : "+lastpoint.x+","+lastpoint.y);
for(int i=0;i<routemap.getSize();i++){
	if(lastpoint.equals(routemap.getReachable(i).getPoint())){
		for(int j=0;j<routemap.getReachable(i).getReachablePoints().size();j++){
			WayVO tlist = new WayVO();
			for(int r=0;r<newlist.getSize();r++){
				tlist.addWayPoint(newlist.getWayPoint(r));
			}
		tlist.addWayPoint(routemap.getReachable(i).getReachablePoints().get(j));
			agenda.add(tlist);
			System.out.println(" Newlist: "+newlist.getSize());
			showAgenda();
		}
		System.out.println("gefunden!");
	}
}
}
points = new ArrayList<Point>();
for(int i=0;i<newlist.getSize();i++){
	points.add(newlist.getWayPoint(i));
}
route = new PathVO(points);
return route;
}
```

Zuerst wird die Agenda initialisiert, indem der Startknoten expandiert wird, und die entstandenen Wege in der Agenda gespeichert werden. Danach wird eine Endlosschleife augerufen, die solange wiederholt wird, wie die Variable *success* auf false gesetzt ist. Danach wird der kleinste Weg in der Agenda ermittelt und aus der Agenda entnommen. Daraufhin wird überprüft ob es sich um den Zielknoten handelt. Ist dies der Fall wird *success* auf true gesetzt und die Endlosschleife verlassen. Weiter wird überprüft, ob der Knoten bereits expandiert wurde. Ist das der Fall wird die Endlosschleife von vorn abgearbeitet. Ansonsten werden, anhand der Straßenkarte, die vom entnommenen Knoten aus erreichbaren Knoten ermittelt und die neuen Wege in die Agenda gespeichert. Die jeweils expandierten Knoten werden in einer Liste *expandedPoints* gespeichert. Nachdem die Endlosschleife verlassen wurde, wird der kürzeste Pfad zurückgegeben.

5.9. Einordnung in das Schichtenmodell

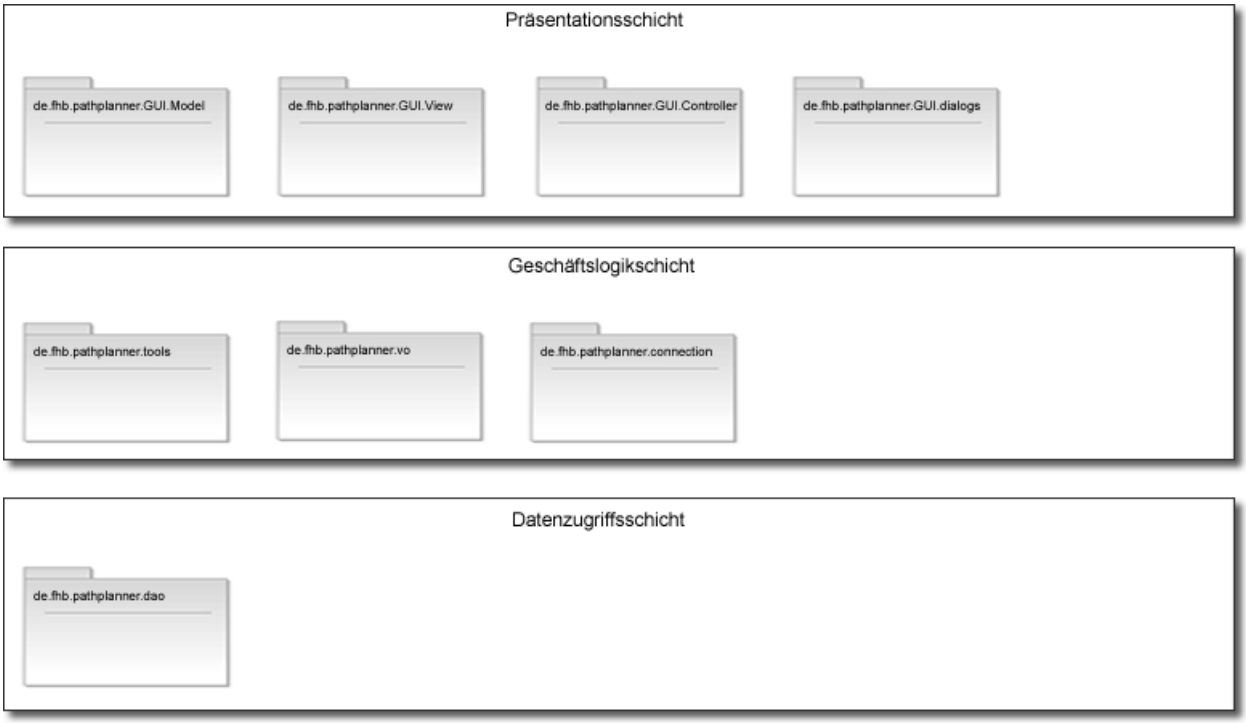


Abb. 52 : Schichtenmodel

Die Applikation ist in 3 wesentliche Schichten aufgeteilt, welche jeweils ihre ganz spezifischen Eigenschaften und Aufgaben haben. Die Abhängigkeitsbeziehungen der Komponenten sind hierbei durch die Architektur eingeschränkt. Die Aufteilung einer Applikation in Schichten hat den großen Vorteil, dass sich bestimmte Komponenten nach ihren Aufgaben strukturieren lassen und daher eine logischen und übersichtlichen Struktur gewährleiten. Desweiteren wird dadurch auch die Weiterentwicklung und Erweiterung enorm erleichtert. Auch im Hinblick auf die Wartung und Austauschbarkeit von Komponenten ist die Aufteilung eines Softwaresystems in Schichten von großem Vorteil. Eine Schicht hat hierbei immer nur Zugriff auf die darunterliegende Schicht, wodurch Zyklen im Abhängigkeitsgraphen vermieden werden. Es ist durchaus möglich Schichten zu überspringen, sofern eine strikte Ordnung zugrunde liegt.

5.9.1. Datenzugriffsschicht

Alle Klassen die zur Datenzugriffschicht gehören befinden sich im Package DAO. Diese Klassen haben Zugriff auf externe Datenquellen und sind daher auch zur Persistentmachung der Daten verantwortlich. Es existiert ein MapSaveDAO und ein MapLoadDAO. Das MapSaveDAO speichert die Kartendaten die in den Value Objects gespeichert sind in eine externe Datei ab. Das MapLoadDAO lädt Daten aus einer Datei in die dazugehörigen Value Objects. Diese befinden sich im Package VO.

5.9.2. Präsentationsschicht - Benutzerschnittstelle

Die Präsentationsschicht ist ausschließlich für die grafische Darstellung und Präsentation der Applikation verantwortlich. Alle dazugehörigen Klassen befinden sich im Package GUI. Alle nötigen Daten zur Visualisierung werden aus den Value Objects bezogen.

Die graphische Benutzerschnittstelle wurde nach dem Model-View-Controller-Konzept umgesetzt. Dabei wurden alle Komponenten der graphischen Benutzerschnittstelle in Model- View- und Controllerobjekte untergliedert. Jedes dieser Objekte hat ganz spezifische Eigenschaften und Aufgaben. Der Vorteil, die Präsentationsschicht zu untergliedern ist, dass sich nicht alle Objekte untereinander kennen müssen. Hierbei gilt das Beobachteter – Beobachter Prinzip, indem der Controller einer Komponente, die Controller der Kindkomponenten überwacht.

- **Model**

Das Modelobjekt ist passiv und einzig dafür konzipiert, Daten und Zustände der Dialogkomponente zu speichern und zur Verfügung zu stellen. Zusätzlich enthält sie die gesamte Verarbeitungslogik. Daten und Zustände werden durch den Controller manipuliert.

- **View**

Hierbei handelt es sich einzig und allein um die Sicht auf die Komponente und ihre graphische Darstellung. Werte und Zustände kann die View Komponente vom Model beziehen. Ein Model kann durchaus mehrere Views gleichzeitig besitzen. Somit können beliebig viele Sichten auf ein Model erzeugt werden.

- **Controller**

Der Controller ist zuständig für die Ereignisverarbeitung einer Komponente. Daraufhin leitet er alle notwendigen Ereignisse in die Wege und sorgt dafür, dass sich die Model- sowie die Viewkomponente aktualisiert. Er ist quasi das Bindeglied zwischen Model und View. Es wird zusätzlich ein Benachrichtigungsmechanismus implementiert, durch den alle zu einem Model zugehörigen Views benachrichtigt und aktualisiert werden. Dies entspricht dem Observer-Pattern (Beobachtermuster). Die Kommunikation zwischen unterschiedlichen Komponenten der Oberfläche erfolgt ausschließlich über die jeweiligen Controller.

Das Model-View-Controller Konzept wird nun am Beispiel der Map-Kompente erläutert.

- **MapModel**

Das MapModel enthält das MapVO, also die komplette Karte

- **MapView**

Das MapView zeichnet die Karte, indem es die MapVO vom MapModel holt.

- **MapController**

Sobald Ereignisse auf der Karte geschehen, muss der MapController entscheiden welche weiteren Ereignisse er auslösen muss. Beispielsweise wird mit der linken Maustaste auf die Karte geklickt. Nun wird ermittelt welchen Zustand das MapModel zurückgibt. Wir befinden uns beispielsweise im Editiermodus für Wände. Der MapController löst im MapModel das Erstellen einer neuen WallVO aus. Gleichzeitig löst er im MapView die Methode zum Neuzeichnen aus.

5.9.3. Geschäftslogikschicht

Diese Schicht verarbeitet alle anfallenden Daten und beinhaltet alle Verarbeitungsmechanismen. Die gesamte Anwendungslogik ist in der Geschäftslogikschicht vereint. Sie steuert die Datenzugriffsschicht um beispielsweise Daten aus einer externen Quelle zu laden. Sie verwaltet alle Klassen die zur Manipulation und Verarbeitung der Daten benötigt werden.

6.

TEST UND FUNKTIONSNACHWEIS

Im folgenden Kapitel soll die Funktionalität der entwickelten Applikation anhand von Tests und Experimenten geprüft beziehungsweise nachgewiesen werden. Dafür werden gezielt Extremsituationen simuliert.

6.1. Erstellen einer Karte

Eine Karte wird erstellt, indem im Menü unter dem Punkte *file*, auf *new map* geklickt wird. Es öffnet sich ein Dialog, indem die Breite und die Höhe in Millimetern der zu erstellenden Karte eingegeben werden müssen. Danach wird mit Druck auf den OK-Button bestätigt und es wird eine Karte erzeugt.

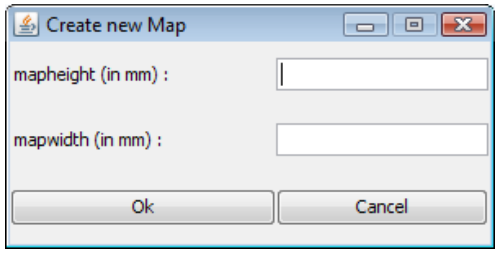


Abb. 53 : Dialog zum Erstellen einer Karte

Nun kann ein Werkzeug zum Erstellen von Objekten in der Karte gewählt werden. Dazu wird das gewünschte Werkzeug aus der Werkzeugleiste gewählt.

Abb. 54 : Werkzeugleiste

Die verschiedenen Werkzeuge zur Erzeugung und Manipulation von Karten werden nun von links oben nach rechts unten aus Abbildung 54 werden nun näher erklärt.

Erzeugung eines viereckigen Hindernisses

Es wird mit der linken Maustaste in die Karte geklickt, wo sich ein Eckpunkt des rechteckigen Hindernisses befinden soll und die Maustaste gehalten. Nachdem man die Maus an die Position bewegt hat, bis das gewünschte rechteckige Hindernis erstellt wurde, kann die linke Maustaste losgelassen werden.

Erzeugung einer Wand

Wie bei der Erzeugung eines viereckigen Hindernisses wird zunächst die linke Maustaste an der gewünschten Position des ersten Endpunktes der Wand geklickt und gehalten. Nachdem die Maus zur Position des zweiten Endpunkts bewegt wurde, kann die Maustaste wieder losgelassen werden.

Erzeugung eines polygonalen Hindernisses

Zur Erzeugung eines Polygons wird in der Karte an der Position eines der Polygon-Eckpunkte die rechte Maustaste kurz geklickt. Wird die Maus nun über die Karte bewegt, wird gleichzeitig eine Wand zur aktuellen Position gezeichnet. Wird erneut die rechte Maustaste betätigt, wird der nächste Eckpunkt bestimmt und mit dem vorhergehenden verbunden. Dies wird so oft wiederholt, bis die gewünschte Anzahl an Eckpunkte erzeugt wurde. Dann wird das Zeichnen des Polygons mit Klicken der linken Maustaste beendet.

Erzeugung eines Start- und Zielpunktes

Zur Positionierung eines Start- und Zielpunktes wird mit der linken Maustaste an der gewünschten Position in der Karte geklickt.

Editieren der Karte

Dazu wird der Mauszeiger über eine Wand oder ein Hindernis in der Karte bewegt. Befindet sich der Mauszeiger über eine Wand, färbt sich diese rot ein, befindet er sich über ein Hindernis färbt sich dieses grau ein. Indem man die linke Maustaste gedrückt hält, können Hindernisse und Wände auf der Karte verschoben werden. Bewegt man den Mauszeiger über einen Eck- bzw. Endpunkt einer Wand oder eines Hindernisses, kann dieser einzelne Punkt verschoben werden.

Erzeugung einer Tür

Dabei wird kein Türartefakt erzeugt, sondern eine bestehende Wand geteilt und dadurch eine Durchfahrt erzeugt. Die geschieht, indem man den Mauszeiger über die gewünschte Wand bewegt und mit der linken Maustaste bestätigt. Wird eine Tür in einer Wand eines Hindernisses erzeugt, zerfällt dieses in einzelne Wände.

Löschen

Das Löschen einer Wand oder eines Hindernisses erfolgt, indem man den Mauszeiger über das zu löschende Objekt bewegt und mit der linken Maustaste klickt.

Unter der Werkzeugleiste befindet sich ein Auswahlfenster, mit dem verschiedene in der Karte befindliche Elemente ein- und ausgeblendet werden können. Unter dem Fenster, indem die Karte angezeigt wird, befinden sich Buttons zur Programmsteuerung.

6.2. Initialisieren von Saphira und dem Simulator

Zunächst wird der Pioneer Simulator gestartet. Es wird eine Paramterdatei und daraufhin, die .wld Datei, die mit dem Pathplanner erstellt wurde, geladen. Danach wird die Karte in Sahira geladen. Dies ist nicht zwingend notwendig, da Saphira die Karte nicht zum Navigieren benötigt, allerdings hat man so einen besseren Überblick und kann beispielsweise die Sensorwerte des Simulators mit der Karte in Saphira vergleichen. Jetzt wird die dynamische Bibliothek pilot.dll hinzu geladen. Danach wird noch die Datei pilot.act zu Saphira hinzu geladen und es kann mit dem Simulator verbunden werden. Wenn eine Verbindung zum Simulator hergestellt

werden konnte, kann die Aktivität pilot mit dem Befehl *start pilot* gestartet werden. Saphira ist nun blockiert, da der Socketserver auf die Verbindung mit einem Client und ankommenden Befehlen wartet.

Nun kann im Pathplanner der Button *connect* betätigt werden. In der Anzeige rechts neben der Karte wird nun der Verbindungsstatus angezeigt. Sobald Die Verbindung zwischen Pathpalnner und Saphira besteht, kann mit der Navigation begonnen werden, indem der Button *navigate* betätigt wird. Man kann nun im Simulator und auch in Saphira beobachten, wie der Roboter von Start- zum Zielpunkt navigiert.

6.3. Test 1 – Viele kleine Hindernisse

Dieser Test ist vor allem für den A-Stern Algorithmus eine Belastungsprobe, da durch viele Hindernisse viele Wegpunkte entstehen, daraus wiederrum eine komplexe Straßenkarte resultiert und somit viele mögliche Wege vom Start- zum Zielpunkt existieren. Auch die Agenda wird sehr viel Speicher benötigen, da in ihr sehr viele Wege gespeichert werden müssen. Für diesen Test wurde folgende Karte erstellt. Diese ist auf der Begleit-CD im Ordner Tests mit dem Namen Test1.map zu finden

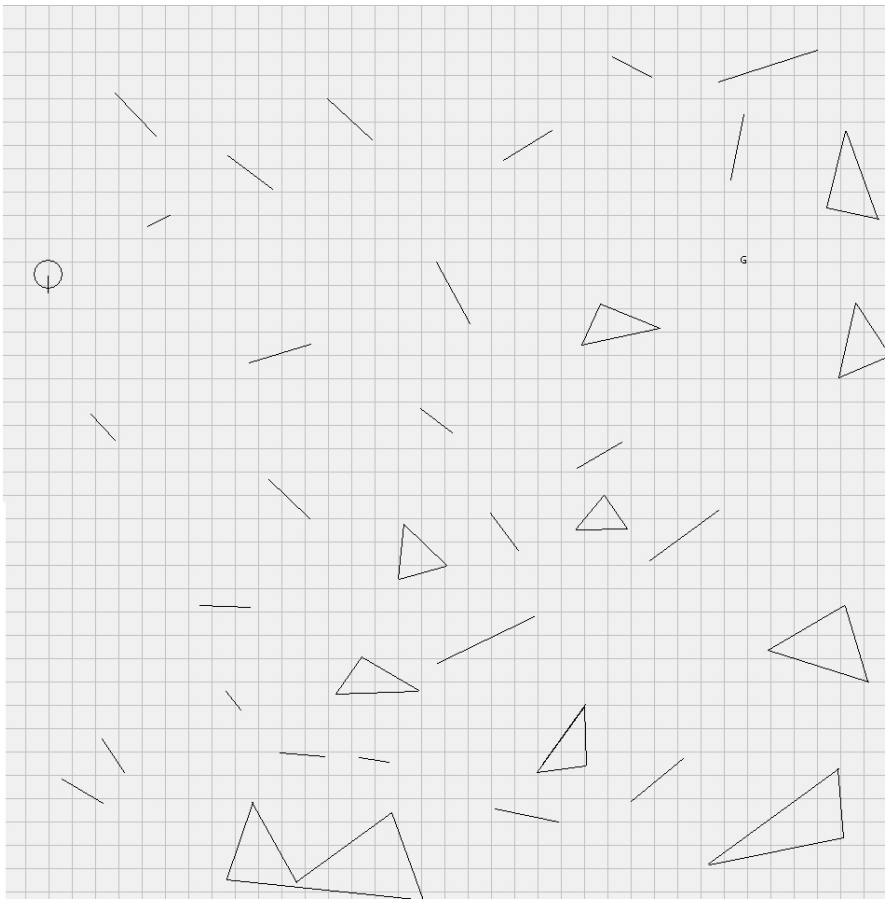


Abb. 55 : Test 1 - Karte

Es existieren viele kleine Hindernisse. Die Laufzeit steigt merklich, aber dennoch nur im Sekundenbereich. Nachdem die Straßenkarte erstellt wurde, hat der A-Stern Algorithmus folgenden Weg berechnet.

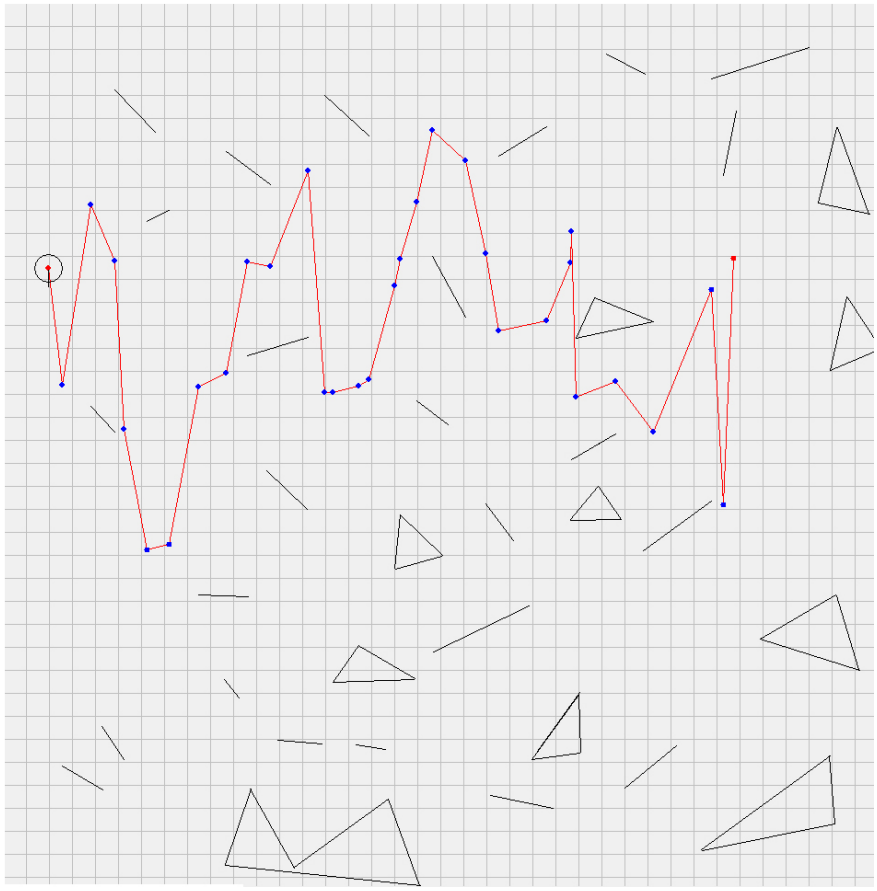


Abb. 56 : Test 1 – gefundener Weg

Damit wäre dieser Test erfolgreich bestanden. Die Applikation kann, trotz vieler Hindernisse und komplexer Straßenkarte, einen Weg finden. An diesem Test ist auch sehr deutlich zu erkennen, dass bei der Zellzerlegung oft unnötige Wege gefahren werden.

6.4. Test 2 – Nur ein Weg zum Ziel

Oft führt nur ein Weg zum Ziel. Dieser Fall soll in dem folgenden Test überprüft werden. Dabei wurde eine Karte mit den Ausmaßen von 100m x 100m erstellt. Diese Karte wird von einem sehr großen Hindernis geteilt, welches eine kleine Durchfahrt besitzt. Diese sollte von der Applikation gefunden und ein Weg durch diese Durchfahrt berechnet werden. Zusätzlich existieren noch weitere kleine bis mittlere Hindernisse, welche die befahrbare Fläche zusätzlich einschränken. Folgende Grafik zeigt die entsprechende Stelle und den von der Applikation gefundenen Weg. Diese Karte ist auf der Begleit-CD im Ordner Tests mit dem Namen Test2.map zu finden.

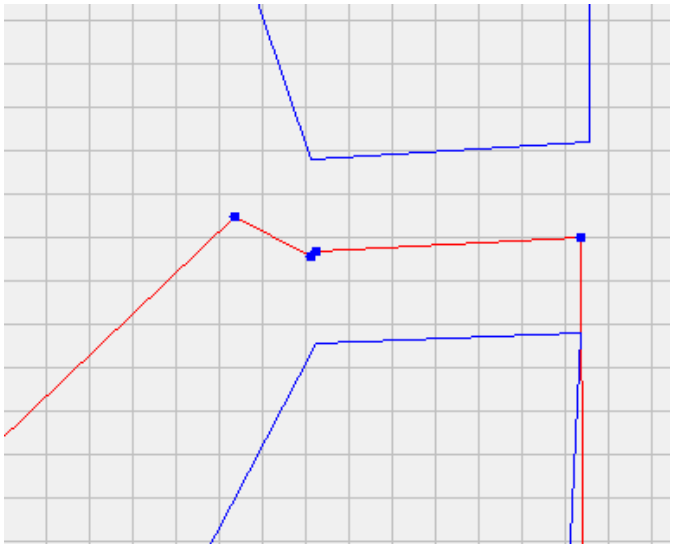


Abb. 57 : Test 2 – gefundene Durchfahrt

Die Applikation hat den einzig befahrbaren Weg gefunden und damit ist dieser Test bestanden.

6.5. Test 3 – Viele Wege zum Ziel

Dieser Test soll nachweisen, dass die Applikation in der Lage ist, unter vielen möglichen Wegen den kürzesten zu finden. In Abbildung 58 ist die entsprechende Karte für diesen Test zu sehen, die auf der Begleit-CD unter im Ordner Tests mit dem Namen test3.map zu finden ist. Es ist schon zu erkennen, dass viele mögliche Wege zum Zielpunkt existieren. Nun soll die Applikation den kürzesten Weg finden.

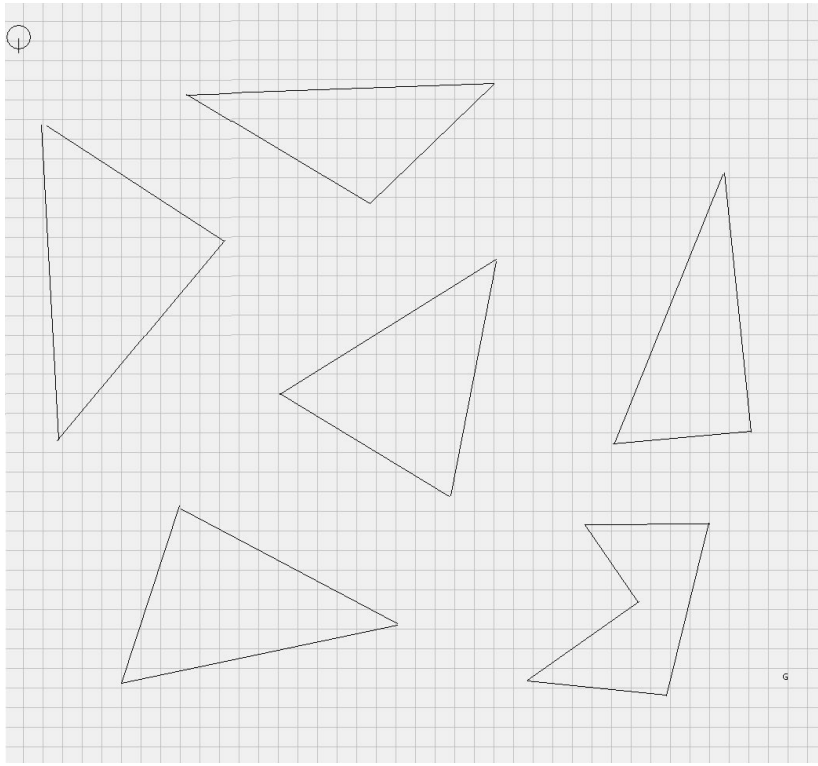


Abb. 58 : Test 3 - Karte

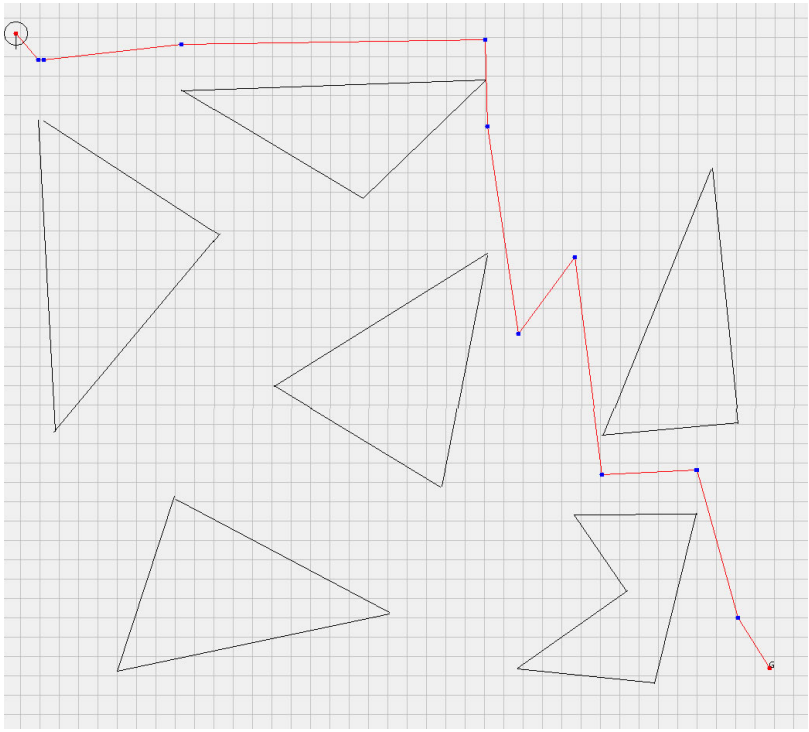


Abb. 59 : Test 3 – gefundener Weg

In Abbildung 59 ist der gefundene Pfad zu sehen. Die Applikation hat den kürzesten Pfad, des Straßennetzes von Start- zum Zielpunkt gefunden und damit wäre auch dieser Test erfolgreich bestanden.

7.

ZUSAMMENFASSUNG

7.1. Ergebnisse

Es wurde eine Applikation erstellt, mit der in einer geographischen Karte mittels vertikaler Zellzerlegung befahrbare Pfade gefunden werden. Nach Anwendung des A* Algorithmus wird der kürzeste Weg des ermittelten Straßennetzes gefunden. Jedoch ist gerade das Straßennetz meist nicht unbedingt optimal, da teilweise unnötige Umwege gefahren werden müssen.

Theoretisch wird also in einer beliebigen geographischen Karte ein optimaler Weg von einem Start- zu einem Zielpunkt gefunden. Nicht beachtet wird allerdings das Auftauchen von dynamischen Hindernissen.

Die simulierte Fahrt mit dem Pioneer Simulator ist dennoch mit einigen Ungenauigkeiten behaftet, welche oft nur zu weniger befriedigenden Ergebnissen führt. Allerdings sind diese an den ungenauen Sensormessdaten und …. zu erklären. Eine gut funktionierende Registrierung sollte diese Effekte minimieren und zu besseren Ergebnissen führen.

Festgestellt wurde weiter, dass die Zellzerlegung für einige Fälle ein eher unbefriedigendes Ergebnis lieferte, was sich in schwer befahrbaren Pfaden wiederspiegelte. Diese waren selbst für den Simulator kaum zu befahren, so dass anzunehmen ist, dass sie mit einem realen Roboter fast gar nicht befahrbar sind. Dies hatte zur Folge, dass weitere Strategien erforderlich waren, um solche Problemsituationen erfolgreich zu bewältigen.

Jedoch wurde für die meisten Fälle ein gutes Ergebnis erzielt und ein gut befahrbarer Pfad gefunden.

Viele ausgiebige Tests haben immer wieder gezeigt, dass die vertikale Zellzerlegung zwar in der Lage ist jeden befahrbaren Pfad innerhalb der erzeugten Karte zu finden, die gefundene Straßenkarte jedoch nicht immer optimal ist. Oft werden unnötige Wege gefahren. Besonders fiel dieses Problem bei sehr hohen Zellen auf. Zudem traten bei den Tests oft Schwierigkeiten bei sehr schmalen Zellen auf, da dort gefundene weg sehr nah an Hindernissen vorbeiführten und einen sehr hohen Anspruch an die Kollisionsvermeidung des Simulators stellten. Meist war diese jedoch erfolgreich.

7.2. Ausblick

Für weiterführende Arbeiten in diesem Gebiet gibt es genug Material. Denkbar wäre eine Umsetzung der Wegplanung mit dem Pathplaner für reale Fahrten. Dazu wäre es allerdings notwendig, eine der genannten Registrierungsverfahren zu implementieren.

Eine nachträgliche Bearbeitung des Pfades wäre auch eine denkbare Möglichkeit um die Befahrbarkeit der gefundenen Pfade zu verbessern. Sinnvoll wäre auch die Implementierung anderer Pfadplanungsverfahren. Im Bezug auf einen Pfad mit größtmöglichem Abstand zu Hindernissen, wäre hier ein genaueres Augenmerk auf die Wegplanung mit Voronoi Diagrammen zu legen, da sich diese während der Arbeit als beste Alternative herausstellten.

Ein weiteres denkbares und faszinierendes Anwendungsbeispiel, wäre die Navigation autonomer mobiler Systeme auf entfernten Planeten. Hier wäre vorstellbar, mit einer Sonde in der Umlaufbahn eines Planeten aufgenommene Fotos, zu einer Karte zusammenzusetzen und in das

.map Format umzuwandeln um anschließend befahrbare Wege auf interessanten Arealen des Planeten zu ermitteln und dann abzufahren.

Auch für Büroboten ist eine eigenständige Pfadplanung sicherlich interessant, da vor dem Einsatz nicht noch eine zeitaufwendige Pfadplanung von den Entwicklern vorgenommen werden müsste, sondern ausschließlich eine Karte von der Umgebung angefertigt werden müsste, in der dann befahrbare Wege durch den Boten selbst ermittelt werden würden.

7.3. Bekannte Fehler

Trotz sorgfältiger Funktionstests, kann es gegebenenfalls zu Fehlern im Programmablauf kommen. So stürzte Saphira beispielsweise gelegentlich ab, nachdem der Pilot gestartet wurde und die Navigation begonnen werden sollte. Die Ursache dafür konnte bislang noch nicht ergründet werden, jedoch funktionierte Saphira dann meist beim zweiten Anlauf.

Ein weiterer bekannter Fehler tritt auf, wenn der Roboter nah an einer Wand vorbei navigieren soll und das Behavior *sfAvoidCollision()* eigentlich eine Kollision vermeiden soll. Hierbei kam es gelegentlich dazu, dass der Roboter gegen die entsprechende Wand fuhr, anstatt dieser auszuweichen. Nach Analyse des Problems, ist wahrscheinlich der Grund dafür, dass die wenigen Sonarwerte, die die Wand bei der Lage lieferte, nicht genügten, um von Saphira als Hinderniss wahrgenommen zu werden. Bei anderen Lagen von Wänden funktionierte die Kollisionsvermeidung jedoch tadellos.

8.

ANHANG

8.1. Quellen

Bücher

[CB] Choset & Budick: The hierarchical generalized voronoi graph, 2000

[GL] Garber & Lin: Constrained-based motion planning using voronoi diagrams, 2002, ISBN 1-58113-506-8

[WV] Boersch, Ingo; Heinsohn, Jochen; Socher-Ambrosius, Rolf: Wissensverarbeitung, 2. Auflage 2007, ISBN 978–3–8274–1844–9

[LaJC] Latombe, Jean-Claude : Robot Motion Planing, 1. Auflage 1991, ISBN 978-0-7923-9206-4

[AuFr] Aurenhammer, Franz: Voronoi Diagrams —A Survey of a Fundamental Geometric, !991, ISSN 0360-0300

[SuBO] Okabe, Boots, and Sugihara, *Spatial Tesselations: Concepts and Applications of Voronoi Diagrams*, Wiley, 1992.

[Rome] Rome, E., Über Navigation autonomer, mobiler Roboter (AMR), KIFS 1996

Websites

[MPiR] Motion Planning in Robotics
URL: http://cse.stanford.edu/class/sophomore-college/projects-98/robotics/basicmotion.html
Besuch: 01.06.2008

[TADM] The Algorithm Design Manual
URL: http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK4/NODE187.HTM
Besuch: 03.04.2008

[LePy] Learning Pyro
URL: http://cs.brynmawr.edu/BeyondLegos/PyroModules/Introduction/Morphology.html
Besuch: 17.06.2008

Ebooks

Martin Greilich: Voronoi-Diagramme
http://www.informatik.uni-trier.de/~naeher/Professur/courses/ws2004/exercises/alggeo/Voronoi-Diagramme.doc

Prof. J. Zhang: Einführung in die Robotik
http://tams-www.informatik.uni-hamburg.de/lehre/ws2004/vorlesungen/einfuehrung_in_die_robotik/folien/02.pdf

Michael Dürig & Stefan Graf & Rolf Moser: Intelligente Roboter - Realisierungskonzept
http://www.eye.ch/~robots/docs/Realisierungskonzept.pdf

Martin Grotz: Graphalgorithmen
http://www2.informatik.uni-erlangen.de/Lehre/SS2005/HalloWelt/Graphen.pdf?language=de
http://www-gs.informatik.tu-cottbus.de/~wwwgs/alg_v08b.pdf

Th. Ottman: Geometrische Algorithmen
http://ad.informatik.uni-freiburg.de/lehre/ss01/geomalg/vorlesung-uebungen/lektion9/lektion9.pdf

[LaSt] Steven M. LaValle: Planning Algorithms

Sonstige Dokumente

[SSMa] Saphira Software Manual

[Deut] Deutsch, Ch., 2004, Pfadplanung für einen autonomen mobilen Roboter, DA an TU Graz

8.2. Abbildungsverzeichnis