Mathematische Modellierung und Optimierung zur Bestimmung kollisionsfreier Wege für Roboter

Diplomarbeit

Fachhochschule Merseburg

Fachbereich Informatik und

Angewandte Naturwissenschaften

Studiengang: Informatik

eingereicht von Christian Rösch

geboren am 05.03.81 in Hohenmölsen

Erstprüfer: Prof. Dr. Jörg Seeländer

Zweitprüfer: Prof. Dr. Karsten Hartmann

Merseburg, den 20.11.2005

Zusammenfassung

Die Diplomarbeit beschäftigt sich mit einer zentralen Fragestellung der Robotik: Der Bestimmung kollisionsfreier Wege für Roboter. Eine besondere Aufgabe besteht darin, den kürzesten Weg zwischen einem gegebenen Paar von Start- und Zielpunkt zu finden. Hierzu ist eine Beschreibung der Geometrie des Roboters und seiner Umgebung erforderlich. Ohne vorzunehmende Einschränkungen hinsichtlich der Freiheitsgrade des Roboters und der Form der Hindernisse ist dieses Problem praktisch nicht lösbar. Daher soll hier der Roboter als Punkt idealisiert werden, welcher sich nur auf einer Ebene (2 dimensionales Problem) bewegen kann. Die Hindernisse werden durch Polygone repräsentiert.

Die Realisierung der Aufgabe kann grob in vier Schritten erfolgen. Zunächst sollen die Polygone sowie deren Platzierung in der Ebene modelliert werden. Danach wird aus diesen Hindernissen unter Beachtung des Start und Zielpunktes ein Graph erstellt, dessen Kanten mögliche Teilwege des Roboters darstellen. Um die Probleme während dieser zwei Schritte, wie z.B. das Finden möglicher Schnitte zwischen Kanten und Polygonen, effizient zu lösen, wird auf Methoden aus der „Algorithmischen Geometrie" zurückgegriffen. Nachdem alle möglichen Kanten bestimmt sind, ist es nötig deren Gewichte zu bestimmen. Diese sind proportional zu deren Längen. Danach kann nun im letzten Schritt mit Algorithmen der Graphentheorie (wie dem von Dijkstra) der Weg durch den Graph vom Start- zum Zielpunkt mit dem geringsten Gewicht gefunden werden, was in diesem Fall gleichbedeutend mit dem kürzesten Weg ist.

Inhaltsverzeichnis

1 Einleitung

Die Industrialisierung am Ende des vorigen Jahrhunderts war geprägt durch die fortschreitende Automatisierung der Produktion. Sollten Maschinen zunächst dem Menschen nur die körperlich schwere Arbeit abnehmen, so übernehmen sie nun vollständig ganze Arbeitsschritte. Der Mensch überwacht und greift nur noch im Fehlerfall ein. Aus den Maschinen werden mehr und mehr selbständige Automaten. Diese brauchen keine Pausen und arbeiten auch unter widrigen Bedingungen mit gleichbleibender Präzision. Sie sind so ideal zur Massenproduktion geeignet. Das Wort „Handarbeit“ ist derweil zu einem Prädikat für Produkte geworden, die eben keine Massenwaren sind, sondern Unikate.

Die weitergehende Automatisierung war auch mit der Verbreitung von Robotern verbunden. Ein Roboter ist laut [Fre91, S. 367] ein „Maschinenmensch“. Heutige Roboter sind dem Menschen aber nur in bezug auf Kraft, Präzision und Belastbarkeit ebenbürtig beziehungsweise sogar überlegen. Die Intelligenz der Roboter bleibt jedoch noch weit hinter der des Menschen zurück. Roboter können nur Spezialarbeiten übernehmen, wie beispielweise die Lackierung eines immer gleichen Karosserieteils. Dazu führt der Roboter wiederholt die vorher manuell festgelegten Bewegungsmuster aus. Die Aufgabe einen Roboter zu bauen, dem nicht mehr gesagt werden muss, wie genau er sich bewegen soll, sondern nur noch wohin, führte zum Forschungsgebiet der „Bewegungsplanung für Roboter“. Mit dieser Thematik beschäftigt sich auch die vorliegende Diplomarbeit.

Die Notwendigkeit, die Bewegungen eines Roboters zu planen, besteht in vielen Situationen. Ein Roboterarm soll ein Objekt an einer bestimmten Position greifen. Ein zweibeiniger Roboter soll seine Beine so bewegen, dass er beim Laufen nicht umkippt. Eine weitere wichtige Fragestellung ist die, wie ein Roboter von seiner aktuellen Position zu einem bestimmten Ziel kommen kann.

1.1 Das Thema

„Mathematische Modellierung und Optimierung zur Bestimmung kollisionsfreier Wege für Roboter“

In dieser Diplomarbeit soll zunächst ein Modell einer Welt erstellt werden, in dem sich der Roboter bewegen kann. Diese Welt wird als Ebene betrachtet, um die Fragestellung auf ein zweidimensionales Problem zu reduzieren. Das Modell der Welt ist durch ein Rechteck abgegrenzt, welches der Roboter nie verlässt. Dies könnte beispielsweise die Grundfläche einer Werk- oder Lagerhalle sein. In der Modellwelt werden verschiedene Hindernisse platziert, welche durch Polygone beziehungsweise Vielecke repräsentiert werden. Dann wird ein Start- und ein Zielpunkt festgelegt, zwischen denen der Weg bestimmt wird.

Nun erfolgt eine Umwandlung dieser Modellwelt in einen bewerteten Graphen, welcher eine Anzahl möglicher Teilwege darstellt. Indem sichergestellt wird, dass der Roboter auf keinem dieser Teilwege mit einem Hindernis kollidiert, wird garantiert, dass auch deren Kombinationen einen kollisionsfreien Weg darstellen. Um den optimalen Weg bestimmen zu können, werden die Teilwege als Kanten des Graphen repräsentiert und mit Gewichten versehen. Danach ist die Anwendung verschiedener Algorithmen aus dem Gebiet der Graphentheorie möglich.

Es ist ein Programm zu entwickeln, in dem der Benutzer zunächst diese vereinfachte Welt eingeben kann. Dies soll möglichst eingängig und intuitiv geschehen können. Also in etwa so, wie aus ähnlichen Programmen bekannt. Der Benutzer kann einfach einige Hindernisse auf die Arbeitsfläche ziehen und dort vielfältig manipulieren. Diese Eingaben müssen zunächst intern gespeichert werden.

Dann beginnt die eigentliche Verarbeitung der Daten. Hierzu müssen verschiedene Algorithmen implementiert werden, welche, wie schon erwähnt, aus der Graphentheorie oder der Algorithmischen Geometrie stammen. Hierbei soll ein besonderes Augenmerk darauf liegen, das die Algorithmen möglichst effizient arbeiten. Daher wird später auch ein genauerer Blick auf die praktisch erreichten Laufzeiten geworfen.

2 Überblick

2.1 Eine kurze Einführung in die Graphentheorie

2.1.1 Einleitung

Ein großer Teil dieser Diplomarbeit basiert auf Erkenntnissen und Algorithmen aus dem Gebiet der Graphentheorie. Dieses ist eines der vielen Themengebiete, welches der Diskreten Mathematik angehört. Wie in jeder mathematischen Disziplin gibt es auch hier einige Grundbegriffe, die bekannt sein sollten, um sich zu verständigen. Die wichtigsten Fachbegriffe der Graphentheorie sollen nun auf den folgenden Seiten geklärt werden. Es kann hier natürlich nicht gelingen, das gesamte Gebiet abzudecken. Wir werden uns daher hier auf die für die weitere Diplomarbeit nötigen Begriffe beschränken. Für weitergehende Informationen sei der interessierte Leser auf [Tit03] oder [CH94] verwiesen. Auch dieses Kapitel stützt sich auf diese Quellen.

2.1.2 Grundbegriffe der Graphentheorie

Definition:

Ein **ungerichteter Graph** G = (V, E) besteht aus zwei Mengen:

- die **Knotenmenge** V, eine Menge von **Knotenpunkten**
- die **Kantenmenge** E, eine Menge von **Kanten**

so dass jede **Kante** einem ungeordneten Paar von **Knotenpunkten** (u, v) zugeordnet ist.

Die Bezeichnungen V und E sind den englischen Worten „vertex“ (Knotenpunkt) und „edge“ (Kante) entlehnt. Knotenpunkte werden manchmal auch als Ecken, Punkte oder einfach Knoten bezeichnet.

Eine kurze Einführung in die Graphentheorie

Definition:

Zwei Knotenpunkte u, v $\in$ V, welche durch eine Kante e= (u, v) verbunden sind, heißen **benachbart** oder **adjazent**. Die Menge aller Knoten v, für die eine Kante e= (u, v) existiert, heißt **Nachbarschaft** von u.

Definition:

Eine Folge v_1, e_1, v_2, e_2, ..., v_{k-1}, e_{k-1}, v_k von Knotenpunkten und Kanten in einem Graphen, in der die Knoten v_i und v_{i+1} jeweils durch die Kante e_i verbunden sind, heißt **Kantenfolge.**

In einer Kantenfolge können Knotenpunkte und/ oder Kanten mehrfach vorkommen.

Definition:

Sind die Knotenpunkte v_1,v_2,...,v_k in der Kantenfolge W= v_1, e_1, v_2,e_2, ..., v_{k-1}, e_{k-1}, v_k paarweise verschieden, dann heißt W ein **Weg**.

Kommt in einer Kantenfolge eine Kante mehrmals vor, so muss auch mindestens einer der dadurch verbundenen Knotenpunkte mehrmals in dieser Kantenfolge enthalten sein. In einem Weg sind daher auch alle Kanten von einander verschieden.

Definition:

In einem **bewerteten Graph** G = (V, E) ist jeder Kante e $\in$ E eine reelle Zahl c(e) zugeordnet. Diese Zahl wird als **Bewertung** von e bezeichnet.

Die Bewertungen können auch als Gewichte oder Kosten betrachtet werden. Viele Probleme der Graphentheorie basieren auf der Suche einer solchen Teilmenge der Menge aller Kanten, bei der die Summe der Bewertungen ein Minimum oder Maximum annimmt. Dies trifft auch auf Wege zu. Deren Bewertungen ergeben sich auf natürliche Weise aus der Summe der Bewertungen aller darin enthaltenen Kanten.

Definition:

Ein **gerichteter Graph** G = (V, E) besteht aus zwei Mengen:

- die **Knotenmenge** V, eine Menge von **Knotenpunkten**
- die **Bogenmenge** E, eine Menge von **Bögen**

so dass jeder **Bogen** einem geordneten Paar von **Knotenpunkten** (u, v) zugeordnet ist.

Offensichtlich „ähnelt" ein gerichteter Graph einem ungerichteten Graphen, nur sind hier nicht zwei Knotenpunkte wechselseitig durch eine Kante verbunden, sondern ein Bogen führt von einem Knotenpunkt zu einem anderen. Ein Bogen wird daher manchmal auch gerichtete Kante oder einfach nur Kante genannt, wenn klar ist, dass es sich um einen gerichteten Graphen handelt. Analog zu Kantenfolgen und Wegen in ungerichteten Graphen gibt es in gerichteten Graphen Bogenfolgen bzw. gerichtete Kantenfolgen und gerichtete Wege.

2.2 Lösungsschritte

2.2.1 Die Welt beschreiben

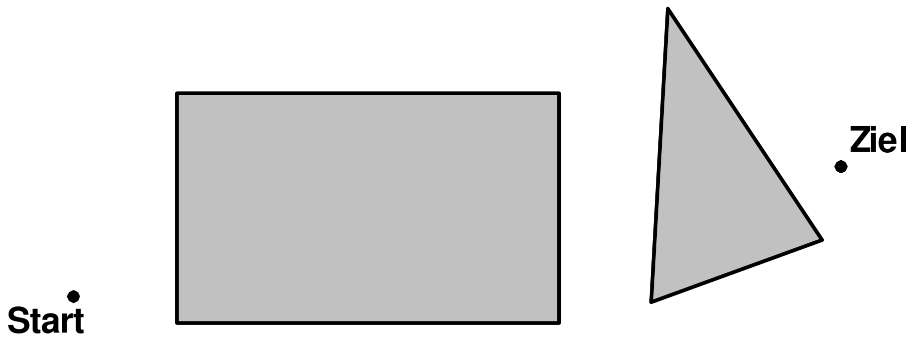

Abbildung 2-1: Ein typischer Problemfall

Im Rahmen der Diplomarbeit soll ein Programm entstehen, in dem der Benutzer eine „virtuelle Welt“ in einer Ebene erstellen kann. Diese besteht hauptsächlich aus Hindernissen sowie einem Start- und Zielpunkt. Die Hindernisse selbst sind ebene Polygone, welche durch den Benutzer beliebig manipuliert werden können. Er kann so gewissermaßen einen „Sandkasten“ erstellen, ein Modell der Welt, in dem ein echter Roboter agieren sollte. Es ist zu klären wie wirkliche Welt und Modell miteinander korrespondieren und mögliche Einschränkungen des Modells zu identifizieren.

2.2.2 Mögliche Teilwege identifizieren

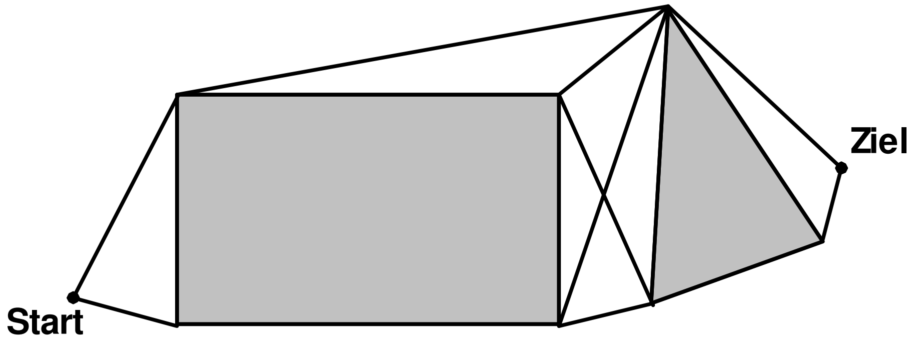

Abbildung 2-2: Mögliche Teilwege zwischen den Hindernissen

Es gibt nun unendlich viele Möglichkeiten sich in der Ebene zu bewegen. Daher sind Methoden zu finden, welche eine endliche Teilmenge dieser Möglichkeiten auswählen. Diese Bewegungen sollen dann als Teilwege dienen, aus denen eine Gesamtbewegung für den Roboter konstruiert werden kann. Diese Gesamtbewegung soll den Roboter auf dem optimalen Weg vom Start- zum Zielpunkt führen. Natürlich soll auch die Bestimmung der Teilwege möglichst effizient implementiert werden. Dadurch ist das Programm nun in der Lage, in der durch den Benutzer erstellten Welt die erwähnten Teilwege zu finden und auszugeben.

2.2.3 Bewerteten Graphen aufbauen

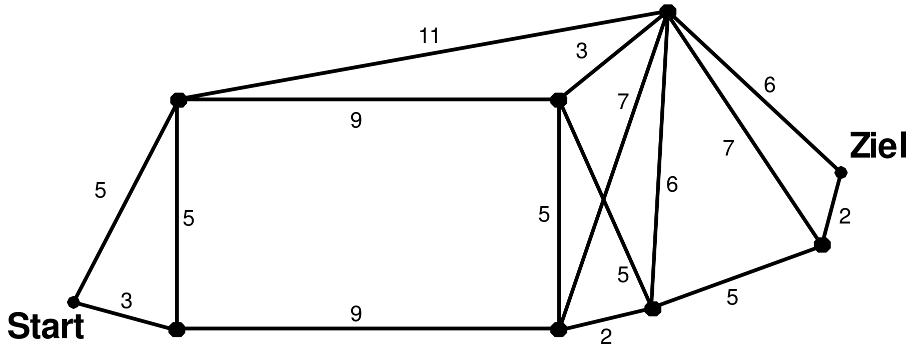

Abbildung 2-3: Den Entfernungen entsprechend bewerteter Graph

Es wird sich zeigen, dass die Teilwege als Kanten eines Graphen betrachtet werden können. Da der Aufwand, um einen der Teilwege zu absolvieren, proportional zu dessen Länge ist, ergibt sich so ein bewerteter Graph. Seinen Kanten werden Bewertungen zugeordnet, welche die Längen der zugehörigen Teilwege repräsentieren.

2.2.4 Optimalen Weg bestimmen

Schließlich muss der kürzeste Weg bestimmt werden. Das bedeutet für den Graph, Kanten derart auszuwählen, dass die Summe ihrer Bewertungen minimal wird. In der Graphentheorie sind einige Algorithmen bekannt, die derartige Probleme lösen. Auch hier soll wieder darauf geachtet werden, solche Algorithmen zu benutzen, die in diesem Anwendungsfall möglichst effizient arbeiten.

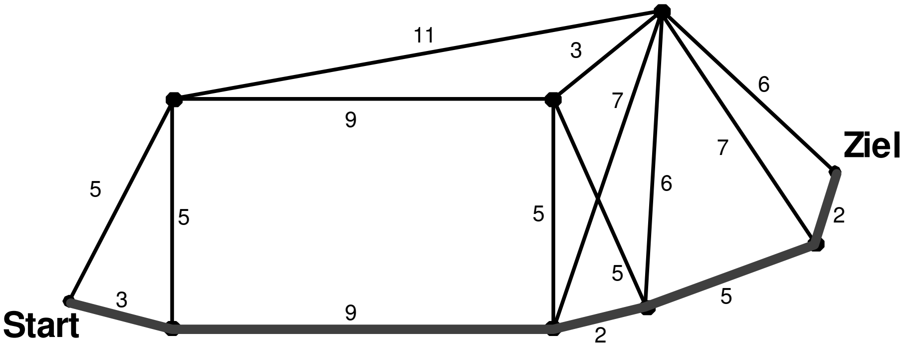

Abbildung 2-4: Ein Weg im Graphen

Ist die Suche nach dem kürzesten Weg im Graphen abgeschlossen, wird diese Lösung zurück auf die Teilwege zwischen den Hindernissen übertragen. Dann kann eine graphische Anzeige der gefundenen Lösung für den Benutzer erfolgen.

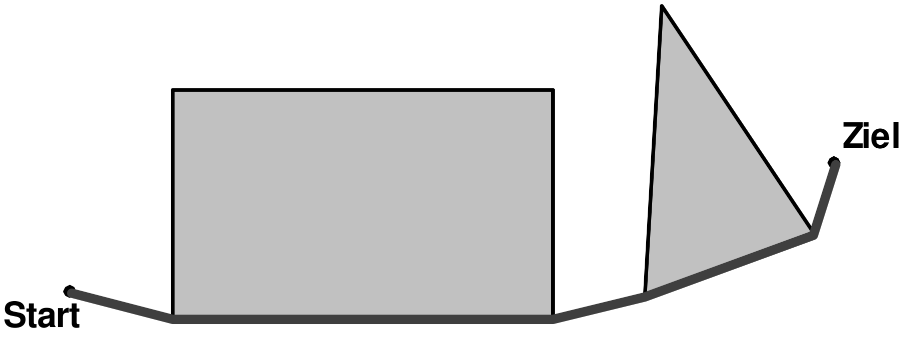

Abbildung 2-5: Präsentation der Lösung

2.3 Bekannte Algorithmen

Nachdem die Lösungsschritte grob dargestellt wurden, sind bereits zwei Aspekte deutlich geworden, welche einen großen Einfluss auf die Performance des zu erstellenden Programms haben.

Zuerst wäre die Bestimmung der Teilwege zu nennen. Dies ist ein geometrisches Problem und fällt in den Bereich der „Algorithmischen Geometrie“. Unter den hier bekannten „sweep“- Algorithmen gibt es auch einen, mit dessen Hilfe sich Schnitte zwischen Strecken in der Ebene finden lassen [Kle97, S. 64ff]. Es wäre denkbar das Problem zu verallgemeinern, etwa die Kanten aller Polygone als einzelne Strecken zu betrachten. Eine Anwendung dieses Algorithmus wäre dann möglich. Noch vielversprechender ist die in [BKOS00, S. 310ff] vorgestellte Methode, welche auf einem „rotational sweep“ basiert. Hierbei wird ein Strahl um seinen Ausgangspunkt rotiert. Die Eckpunkte der Polygone werden so alle nach und nach bearbeitet, sobald sie auf dem Strahl liegen.

Ein weiteres Hauptaugenmerk liegt auf einer wichtigen Anwendung der Graphentheorie. Es handelt sich hierbei um die Bestimmung eines optimalen Weges in einem bewerteten Graphen. Ein optimaler Weg bedeutet in diesem Fall einen Weg mit einer minimalen Länge zu finden. Da im Allgemeinen nicht alle Kanten eine einheitliche Länge haben, können die einfachsten Ansätze wie BFS („breadth first search“ [Sch03, S. 88f]) nicht verwendet werden. Andere bekannte Verfahren, wie der Algorithmus von Moore und Ford [Tur96, S. 247ff] finden auch eine Lösung im allgemeinen Fall, in dem auch negative Bewertungen für Kanten zugelassen sind. Bei genauer Betrachtung der Voraussetzungen ist jedoch auch der spezialisiertere Algorithmus von Dijkstra [Dij59] [Sch03, S. 97ff] anwendbar. Weiterhin soll untersucht werden, ob eine noch effizientere Suche mit Hilfe des A*-Algorithmus [HNR68] [Tur96, S. 257ff] möglich und auch sinnvoll ist.

3 Algorithmen und Konzepte

Im vorigen Kapitel wurde das Problem und seine Lösung umrissen. Im Folgenden soll es nun darum gehen, die einzelnen Lösungsschritte genauer zu analysieren. Die zu verwendenden Algorithmen werden dargestellt. Ein Hauptaugenmerk soll auch auf deren Komplexität liegen, welche später das Laufzeitverhalten der Gesamtlösung wesentlich mitbestimmt.

3.1 Modellierung der Welt

Unsere heutigen Computer sind immer noch zu beschränkt, um die wirkliche Welt in ihrer gesamten Komplexität zu erfassen. Daher ist ein einfaches Modell der Welt nicht nur sinnvoll, sondern nötig. Das Modell sollte so abstrakt wie möglich sein und nur die nötigsten Daten enthalten.

3.1.1 Punkt-Roboter in einer ebenen Welt

Der Roboter, dessen Weg geplant wird, sei als Punkt idealisiert. Dieser Punkt kann sich nur in einer Ebene bewegen. Der erste Grund hierfür ist, dass ein Punkt beliebig rotiert werden kann, ohne seine Ausrichtung oder Form zu ändern. Eine Rotation kann also nicht zu einer Kollision führen und muss daher nicht geplant werden. In diesem Modell besitzt der Roboter also nur zwei Freiheitsgrade. Es handelt sich somit um ein zweidimensionales Problem. Der andere Grund für die Punktform hat ebenfalls mit der Vermeidung von Kollisionen zu tun. Um zu garantieren, dass der Roboter niemals an Hindernissen anstößt, darf sich zu keiner Zeit ein Punkt des Roboters innerhalb eines Hindernisses befinden. Besteht der Roboter nun nur aus einem einzigen Punkt, erleichtert dies die Überprüfung, ob ein potentieller Weg diese Bedingung erfüllt.

Den zweiten Teil des Weltmodells stellen die Hindernisse dar. Die Hindernisse werden als Polygone definiert. Der Roboter soll niemals die durch die Polygone umschlossene Flächen berühren. Weiterhin gibt es einen Startpunkt und einen Endpunkt, zwischen welchen der kürzeste Weg gesucht wird. Diese Punkte dürfen selbstverständlich nicht innerhalb der Hindernisse liegen.

Dieses Modell kann natürlich nur sinnvoll genutzt werden, wenn es möglichst ist, die reale Welt darin abzubilden. Wie man mit einem Roboter mit Punktgröße arbeiten soll, ist nicht offensichtlich. Einen Roboter, der keine Ausdehnung hat, gibt es natürlich nicht. Die Lösung ist nicht nur, den Roboter auf einen Punkt zusammenzuziehen, sondern gleichzeitig die Hindernisse um den gleichen Betrag auszudehnen. Hier ein Beispiel, wie dies aussehen kann.

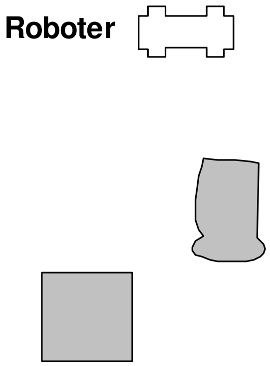

Abbildung 3-1: Ein Roboter und zwei Hindernisse

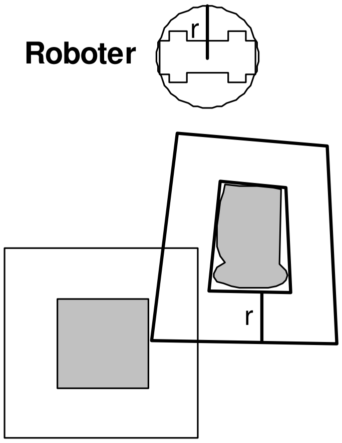

Abbildung 3-2: Sicherheitsabstand für die Hindernisse

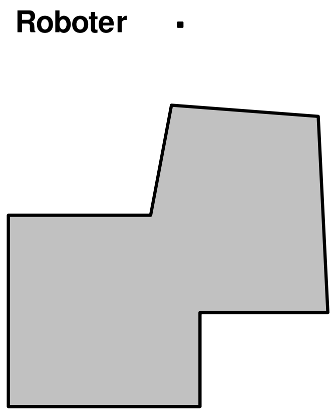

Abbildung 3-3: Die vereinfachte Situation

Abbildung 3-1 zeigt eine komplexe Situation, welche in unser Modell übertragen werden soll. Alle Punkte innerhalb des Kreises im zweiten Bild können durch den Roboter berührt werden, wenn dieser rotiert wird. Daher soll zu jedem Hindernis ein „Sicherheitsabstand“ eingehalten werden, der dem Radius dieses Kreises entspricht. Der Roboter kann dann, unabhängig von seiner aktuellen Rotation, das Hindernis nicht berühren. Die Hindernisse werden also um genau diesen Betrag ausgedehnt. Eines der Hindernisse ist zu Beginn noch kein Polygon, es muss daher durch ein Polygon angenähert werden, welches das Hindernis komplett umschließt. Nach dem Ausdehnen der Objekte entstehen nun nicht die im Bild oben angedeuteten Objekte. Die Ecken wären abgerundet, genaugenommen mit Kreissegmenten deren Radius dem Sicherheitsabstand entspricht. Diese Formen wären jedoch wiederum keine Polygone. Daher wird auch hier wieder auf Polygone, die diese Formen umschließen, zurückgegriffen. Im letzten Schritt müssen nun noch sich berührende Polygone zusammengefügt werden, da sie als ein einzelnes Hindernis betrachtet werden können. Das Ergebnis ist im dritten Bild dargestellt. Wie genau eine solche Transformation algorithmisch durchgeführt werden könnte, ist in [BKOS00] beschrieben.

Die Welt so zu simplifizieren, hat natürlich auch Nachteile. Zuerst einmal wird nicht auf die Rotation des Roboters geachtet. Ein Weg wird nur als sicher betrachtet, wenn das Objekt in jeglicher Ausrichtung entlang des Weges verschoben werden kann. Um es bildlich auszudrücken: eine Leiter durch eine Tür zu tragen wird für den

Computer so zum unlösbaren Problem. Auch geht bei der Annäherung von beliebigen Formen durch eine endliche Zahl von Polygonen immer etwas der begehbaren Fläche verloren. Im allgemeinen würde hier der Weg nur in der Größenordnung der Näherung länger. Wird jedoch durch die Annäherung der Hindernisse ein schmaler Durchgang unpassierbar, so kann es auch passieren, dass der Weg komplett verändert wird. Im schlimmsten Fall wird überhaupt kein Weg mehr gefunden, wenn es keine Möglichkeit gibt den Durchgang zu umgehen. Glücklicherweise sind diese Nachteile in der praktischen Anwendung weniger bedeutsam. Zunächst können der Roboter und auch die Hindernisse nur mit einer gewissen Genauigkeit gemessen werden. Auch kann keine Robotersteuerung dem berechneten Weg ganz genau folgen. Aus diesen Gründen wird man ohnehin einen gewissen Sicherheitsabstand um die Hindernisse und/ oder den Roboter benötigen.

3.1.2 Visualisierung des Weltmodells

Für einen Roboter wäre es nun ausreichend diese Daten abzuspeichern. Menschen aber können zumeist Dateien, die als Listen von Koordinatenpaaren repräsentiert werden, nicht intuitiv erfassen. Daher ist es vorteilhaft die Welt zu visualisieren. Am Computer geschieht dies durch Pixelgrafik. Diese Grafik arbeitet auch in zwei Dimensionen, ist also ideal, um ebene Objekte darzustellen. Trotzdem ist dies nicht trivial. Eine Pixelgrafik ist sowohl in Größe als auch Genauigkeit limitiert. Das es sich hier jedoch um eine bekannte Anwendung der Computergrafik handelt, gibt es Bibliotheken, welche dies in Grundzügen erledigen. Was hinzugefügt werden muss, ist die Interaktion mit dem Benutzer. Dieser kann erstens die Welt scrollen und skalieren. Mit der Bezeichnung „scrollen“ ist gemeint, dass der sichtbare Ausschnitt der Welt verschoben wird. Durch das Skalieren wird dieser Ausschnitt hingegen verkleinert bzw. vergrößert.

Außer den Hindernissen soll auch ein Koordinatengitter dargestellt werden, welches dem Benutzer die Orientierung erleichtert. Dieses muss auf dieselbe Art und Weise gescrollt und skaliert werden. Zusätzlich kann gewählt werden, wie dicht das Koordinatengitter ist.

3.1.3 Manipulation des Weltmodells

Bevor der Benutzer einen Teil der Welt manipulieren kann, muss er eventuell zunächst einen Teil auswählen. Bei Punkten geschieht dies durch einfaches Anklicken. Es kann stets nur ein Punkt gewählt werden. Um ein oder mehrere Polygone zu wählen, muss ein Rechteck um diese Polygone gezogen werden. Alle Polygone in diesem Rechteck sind dann gewählt und können verändert werden. Die aktuelle Auswahl soll stets durch eine Markierung verdeutlicht werden.

Verschieben des Start- und Zielpunktes

Da Start und Ziel als reine Punkte definiert werden, ist das Verschieben die einzige sinnvolle Aktion, die hier ausgeführt werden kann.

Einfügen und Entfernen von Hindernissen

Die grundlegende Aktion hinsichtlich der Hindernisse ist das Verändern ihrer Anzahl. Zum Löschen eines Polygons sind keine weiteren Eingaben erforderlich. Beim Einfügen eines neuen Hindernisses muss diesem jedoch schon eine gewisse Form zugewiesen werden, auch wenn diese später beliebig verändert werden kann. Eine offensichtliche Form, die ein Benutzer eingeben kann, ist ein Rechteck. Seine Größe und Position kann mit der Maus bestimmt werden. Eine Variation hiervon ist ein regelmäßiges n-Eck. Der Benutzer gibt zusätzlich zu dem Rechteck eine Anzahl Ecken an. Die Ecken werden dann gleichmäßig auf dem größten, in das Rechteck passenden, Kreis verteilt.

Verschieben einer einzelnen Ecke eines Polygons

Jeder Eckpunkt eines Polygons kann einzeln verschoben werden. So kann ein Polygon eine beliebige Form annehmen

Modellierung der Welt

Ändern der Eckenzahl eines Polygons

Beim Ändern der Eckenzahl eines Polygons werden ihm entweder neuen Ecken hinzugefügt oder Ecken gelöscht. Beim Hinzufügen wird hier eine Kante in der Mitte geteilt und dort eine weitere Ecke hinzugefügt. Wird eine Ecke entfernt, wird das Polygon mit den verbleibenden Punkten neu erstellt.

Verschieben eines Polygons

Das Verschieben eines Polygons gleicht dem Verschieben einer Ecke, jedoch werden hier alle Eckpunkte um den gleichen Vektor verschoben, das heißt es liegt eine Translation des Polygons vor.

Strecken/ Stauchen eines Polygons

Beim Stauchen oder Strecken wird die Ausdehnung eines Polygons entlang der X- und Y-Achse verändert.

Rotieren eines Polygons

Wird ein Polygon rotiert, so wird jeder seiner Punkte um einen Mittelpunkt gedreht. Der Mittelpunkt soll hier der Mittelpunkt der aktuellen Auswahl sein. Er wird bestimmt, in dem ein achsenparalleles Rechteck um die Auswahl gelegt wird, dessen Mittelpunkt berechnet wird.

Arbeiten mit dem Koordinatengitter

Das Koordinatengitter kann aktiv zur Ausrichtung von Objekten genutzt werden. Durch Aktivierung dieser Funktion werden alle Eingaben auf den nächsten Gitterpunkt gerundet. Das heißt, ein Punkt kann nur auf einen Gitterpunkt gesetzt werden und eine Verschiebung kann nur um ein Vielfaches des Abstandes zweier Gitterlinien geschehen.

3.1.4 Plausibilitätstest des Weltmodells

Um in den Computer eine Vielzahl von unterschiedlichen Daten einzugeben, muss dem Benutzer eine große Freiheit beim Eingeben dieser Daten gewährleistet werden. Das führt dazu, dass der Benutzer auch eine Reihe von unsinnigen Daten eingeben kann. Manchmal ist sogar die Eingabe der korrekten Daten am effizientesten, wenn während der Eingabe zwischenzeitlich unsinnige Daten entstehen. Als Beispiel soll ein Datum vom 11.2.2004 auf den 31.3.2004 geändert werden. Eine Möglichkeit dies zu tun, wäre zunächst die 1 in eine 3 ändern und danach die 2 in eine 3. Währendessen würde jedoch zunächst das Datum 31.2.2004 eingegeben. Diese Eingabe ist zwar Unsinn, aber es wäre genau so unsinnig, dieses dem Benutzer sofort zu melden indem z. B. seine Arbeit durch einen Warnhinweis unterbrochen wird. Es ist also stets abzuwägen ob und wann eine Eingabe geprüft werden soll.

Start- und Zielpunkt in Hindernissen

Weder Start- noch Zielpunkt sollten in einem Hindernis liegen. Bevor die Daten des Weltmodells also weiterverarbeitet werden, wird diese Bedingung geprüft. Falls festgestellt wird, dass hier ein Konflikt auftritt, wird der Benutzer darauf hingewiesen.

Überschlagene Polygone

Ein Polygon wird überschlagen genannt, wenn es darin zwei oder mehr Seiten gibt, welche sich schneiden. Ein solches Polygon kann nie komplett im oder gegen den Uhrzeigersinn orientiert sein. Daher wird das „Überschlagen“ eines Polygons bereits beim Verschieben einer Ecke verhindert. Wird versucht, eine Ecke so zu verschieben, das daraus ein überschlagenes Polygon resultiert, wird die Eingabe ignoriert. Stattdessen wird die letzte bekannte korrekte Position verwendet.

Orientierung der Polygone

Um bestimmen zu können, wo genau die Fläche liegt, die ein Polygon belegt, müssen alle Polygone gleich orientiert sein. Daher muss nach einer Manipulation eines Polygons seine Orientierung überprüft werden, oder diese Manipulation von vorn herein ausgeschlossen werden.

Zu Strecken degenerierte Polygone

Es gibt zwei Möglichkeiten für das Entstehen eines derartig degenerierten Polygons. Entweder besteht ein Polygon nur aus zwei Punkten oder alle Punkte liegen auf einer Gerade. Der erste Fall wird verhindert, indem das Löschen von Ecken in Dreiecken verboten wird, sowie alle neu erstellten Polygone mindesten drei Ecken haben. Um zu verhindern das alle Punkte auf einer Gerade liegen, wird dies beim Manipulieren der Größe eines Polygons stets überprüft. Tritt dieser Fall ein, wird die Manipulation verweigert.

Berührungen zwischen Hindernissen

Falls sich Hindernisse berühren, so liegen Teile des Randes eines Polygons in einem anderen. Da jedoch die Kanten eines Polygons als mögliche Teilwege betrachtet werden, kann dies zu Fehlern führen. Bevor mit den Polygonen weitergearbeitet werden kann, muss der Benutzer zwei solche Polygone entweder trennen oder zu einem einzigen zusammenführen.

3.2 Teilwege

Trotz der Einschränkungen des Robotermodells auf einen Punkt und der Bewegungsfreiheit auf die Ebene, gibt es immer noch zahlreiche Möglichkeiten, um eine Bewegung von A nach B zu vollführen. Anstatt einer gerade Strecke, könnte man auch einen weiten Rechtsbogen oder eine Zick-Zack-Linie benutzen. Es gibt also tatsächlich eine unendliche Zahl von Wegen, welche den Roboter vom Start zum Ziel führen. Da die Teilwege später in Kanten eines Graphen umgewandelt werden, erhielte man so aber einen Graphen, der unendlich viele Kanten hat. Diesen könnte man weder explizit speichern, noch effizient bearbeiten.

Glücklicherweise kann nicht jeder dieser ein optimaler Weg sein. Man stelle sich den kürzesten Weg als ein Gummiband [Tur96, S. 4f] [BKOS00, S. 308] vor, welches zwischen dem Start- und Zielpunkt gespannt wird. Zieht sich dieses Gummiband zusammen, so liegt es an einigen Ecken der Hindernisse an und ist dazwischen straff gespannt. Das heißt nun, ein jeder Weg besteht aus einer Menge von Geradenabschnitten oder auch Strecken. Die beiden Endpunkte solcher Strecken entsprechen jeweils einer Ecke eines Hindernisses, dem Start- oder dem Zielpunkt. Damit ein solcher Weg kollisionsfrei ist, dürfen diese Strecken natürlich nicht durch das Innere der Hindernisse führen. Die direkte Verbindung der beiden Endpunkte darf also nicht durch ein Hindernis verdeckt sein, sondern beide Punkte müssen sich „sehen“. Dies basiert auf der bekannten Tatsache, das eine Gerade die kürzeste Verbindung zweier Punkte in der Ebene ist.

3.2.1 Sichtbarkeitsgraph

Es soll also ein sogenannter „Sichtbarkeitsgraph“ erstellt werden. In diesem ist die Information enthalten, welche Punkte sich sehen können und welche nicht. Dazu werden nun zunächst alle Teilwege bestimmt. Die Anzahl der Ecken der Hindernisse ist endlich, daher ist es auch die Anzahl der Teilwege. Trotzdem ist jeder kürzeste Weg aus einer bestimmten Kombination dieser Teilwege zusammengesetzt. Dies findet sich auch in [BKOS00, S. 309], wo gezeigt wird, dass die gegenteilige Annahme zu einem Widerspruch führt.

3.2.2 Einfache Lösung

Wie weiter oben festgestellt wurde, verbindet ein Teilweg entweder zwei Eckpunkte beliebiger Hindernisse, einen Eckpunkt mit dem Start- oder Zielpunkt oder den Startpunkt mit dem Zielpunkt. Um eine Liste aller Teilwege zu erhalten, kann nun

Teilwege

jede Kombination dieser Punkte betrachtet werden. Es wird dann nach einem Schnittpunkt der Strecke mit den Hindernissen gesucht. Wird kein solcher Schnitt gefunden, ist die Strecke tatsächlich ein Teilweg, der später in eine Kante umgewandelt werden kann.

Betrachtung zur Laufzeit

Bezeichne man die Anzahl der Eckpunkte mit n, so ist die Anzahl möglicher Kombinationen (n+2)(n+1). Ein Hindernis hat mindestes drei Eckpunkte, womit es im schlechtesten Fall n/3 Hindernisse gibt. Der Test auf Schnittpunkte hat daher eine Komplexität von O(n) für eine der Kombinationen. Daraus resultiert eine Gesamtkomplexität von $O(n^3)$ für diesen Ansatz [BKOS00, S. 310].

3.2.3 Das Sweep Paradigma

In der algorithmischen Geometrie gibt es eine Anzahl Algorithmen [Kle97, S. 51ff], welche mit einem sogenannten „sweep“ arbeiten. Dies bedeutet soviel wie „Ausfegen“. Damit ist beispielsweise gemeint, dass eine Gerade in einer Richtung über die Ebene geschoben wird und diese dabei alle Punkte der Ebene berührt. Wenn das Problem nun auf eine endliche Anzahl von Punkten beschränkt ist, ergeben sich so parallele Geraden, welche durch diese Punkte führen. Diese Geraden werden dann sortiert und nacheinander betrachtet, wobei die Berechnung durch Informationen der vorhergehenden Geraden vereinfacht wird. Die Information, die man hier von Gerade zu Gerade mitnimmt, wird oft auch als „sweep status structure“ bezeichnet. Die Geraden selbst nennt man auch „events“ (= Ereignisse), da dies Ereignisse sind, welche die Status Struktur verändern.

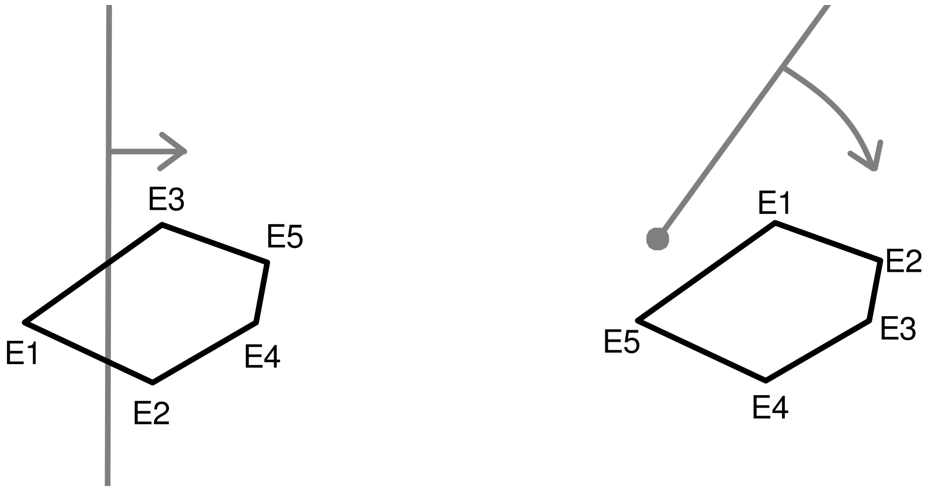

Abbildung 3-4: zwei mögliche „Sweeps“ in der Ebene

Diese Idee ist nun nicht nur auf eine parallelverschobene Gerade beschränkt. Wichtig ist nicht, wie genau man ausfegt, sondern dabei auch die ganze Ebene zu erfassen. Zur Bestimmung der Teilwege soll ein Strahl eingesetzt werden, welcher um einen Punkt rotiert wird. Die Polygone werden in die Strecken zerlegt, welche ihren Kanten entsprechen. Wie in [BKOS00, S. 312ff] beschrieben, eignet sich diese Methode um die Teilwege, welche in diesem Punkt beginnen, zu finden.. Dies wird dort „rotational sweep“ genannt, da der Strahl einmal um 360 Grad rotiert wird. Führt man dies für alle Eckpunkte der Polygone, sowie den Start- und Zielpunkt durch, so hat man alle Teilwege gefunden.

Während nun der Strahl um den Punkt kreist, werden in der Statusstruktur die vom Strahl geschnittenen Strecken gespeichert, welche nach der Distanz vom Punkt geordnet sind. Die Strecke, die dem Punkt am nächsten liegt, verdeckt dabei natürlich weiter entfernte Strecken. Daher sind nur die Punkte auf dieser ersten Strecke sichtbar. Da sich keine der Hindernisse schneiden sollen, können die Strecken auch ihre Reihenfolge nicht tauschen, während der Strahl weiter rotiert. Die Ereignisse sind hier die Eckpunkte der Hindernisse, bei denen jeweils Strecken in die Status Struktur eingefügt oder entfernt werden. Die Sichtbarkeit des Punktes kann einfach durch die Position der zugehörigen Strecke in der Statusstruktur bestimmt werden.

Teilwege

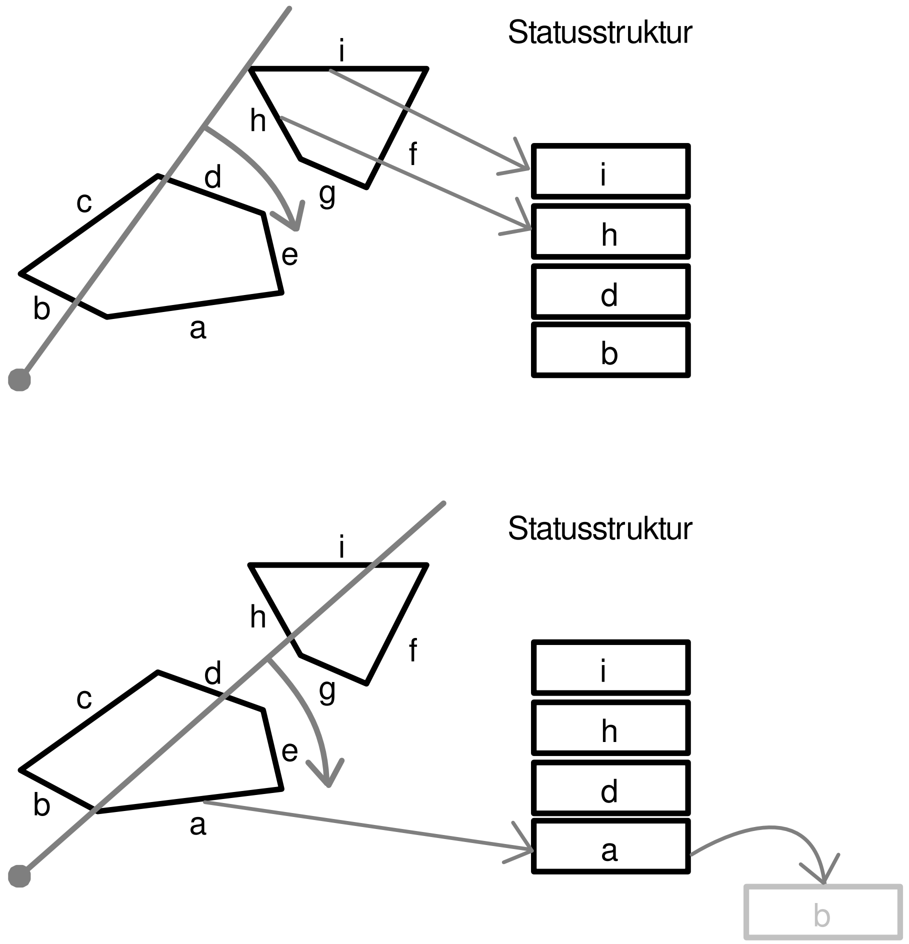

Abbildung 3-5: Zwei Ereignisse und die zugehörige Statusstruktur

Abbildung 3-5 zeigt vereinfacht (das Programm legt die Einträge nicht einfach, wie hier, hintereinander in den Speicher), wie die Statusstruktur bei Ereignissen verändert wird. Im oberen Bild werden keine Teilwege gefunden. Im zweiten Bild würde ein Teilweg zum Startpunkt von „a“ erzeugt, da sich diese Strecke ganz vorn in der Struktur befindet.

Betrachtung zur Laufzeit

Wie in [BKOS00, S. 314] genauer beschrieben, ist die Komplexität dieses Algorithmus hauptsächlich durch das Sortieren der Ereignisse und das Aktualisieren der Status Struktur geprägt. Sei die Anzahl der Eckpunkte wieder mit n bezeichnet, so benötigt das Sortieren eine Zeit von O(n log n). Das Aktualisieren benötigt für jeden Punkt O(log n), für alle Punkte während des gesamten sweeps also ebenfalls O(n log n). Da der sweep nun für jeden der n Punkte ausgeführt wird, erreicht die Gesamtkomplexität O(n² log n).

Aufwand reduzieren

Stellt man sich einmal vor, es gäbe nur ein einziges Hindernis, über den der Strahl rotiert wird, so ständen sofort eine Anzahl Strecken fest, welche vollständig verdeckt sind. Es sind die Strecken, die man auch als Rückseite des Hindernisses bezeichnen würde, eben die vom aktuellen Standpunkt nicht sichtbaren Seiten. Man kann diese Seiten nicht sehen, da sie durch das Hindernis selbst verdeckt werden. Mathematisch beschrieben sind dies Kanten, bei welchen der Mittelpunkt des sweeps auf der gleichen Seite wie das Hindernis selbst liegt.

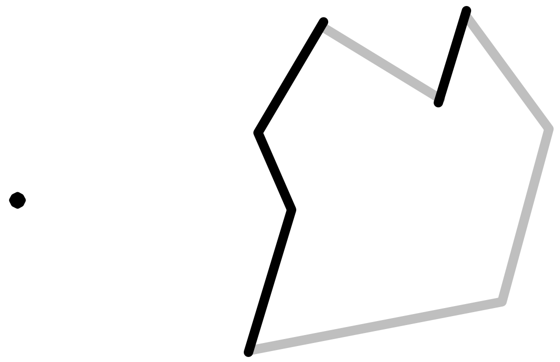

Abbildung 3-6: Die Vorder- und Rückseiten des Polygons

In Abbildung 3-6 wurden diese Kanten grau dargestellt. Natürlich ist es auch möglich, dass einige der anderen, hier schwarzen, Strecken ganz oder teilweise nicht sichtbar sind. Diese Strecken müssen daher weiter untersucht werden. Die Rückseiten der Polygone hingegen können sofort aussortiert werden. Sie selbst sind stets durch die Vorderseiten verdeckt. Auch Kanten anderer Polygone, die noch hinter der Rückseite liegen, sind durch die zugehörige Vorderseite verdeckt. Bei der Suche nach sichtbaren Eckpunkten ist es nun aber unerheblich, durch wie viele Strecken genau ein Eckpunkt verdeckt ist. Es reicht zu wissen, das eine einzige Strecke existiert, die den Punkt verdeckt. Die Vorderseiten reichen also für die Betrachtung aus. Strecken, welche Rückseiten der Polygone bilden, können beim sweep vernachlässigt werden.

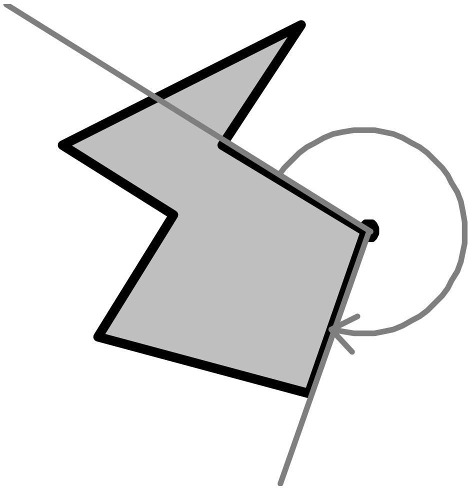

Abbildung 3-7: Ein begrenzter Sweep

In [BKOS00, S. 312] wird vorgeschlagen, den Strahl zunächst entlang der positiven x-Achse auszurichten und dann eine volle 360 Grad-Drehung zu absolvieren. Das gleiche Ergebnis erhält man jedoch auch, wenn man mit einem Strahl in einer beliebigen anderen Richtung beginnt. Weiterhin kann das Verfahren auch nur über einem gewissen Kreisausschnitt durchgeführt werden, wobei alle sichtbaren Punkte in diesem Sektor korrekt gefunden werden.

Dies ist vor allem interessant, da alle Punkte bis auf Start und Ziel, stets Eckpunkte von Hindernissen sind. Das anliegende Hindernis deckt jedoch bereits einen Kreisbogen ab, dessen Untersuchung daher nicht nötig ist. So reicht es in Abbildung 3-7 völlig aus, den Sweep auf den markierten Sektor zu begrenzen.

3.3 Aufbau des Graphen

Die Graphentheorie bietet ein mächtiges Werkzeug, wenn netzartige Strukturen mathematisch analysiert werden sollen. Man denke nur an Computernetze, eine Liga von Fußballmannschaften, wirtschaftliche Verflechtungen oder ein Autobahnnetz. All diesen Beispielen ist gemeinsam, dass sie als ein abstraktes Modell betrachtet werden können, welches mathematisch durch einen Graph dargestellt wird.

Auch aus den Teilwegen, die zwischen den Hindernissen gefunden wurden, soll nun ein Graph entstehen. Da auch die Länge der Teilwege einfließen soll, handelt es sich dabei um einen bewerteten Graphen.

3.3.1 Knotenpunkte

Die Knotenpunkte des Graphen entsprechen den Endpunkten der gefundenen Teilwege. Das bedeutet, es handelt sich um Punkte in der Ebene, entweder um einen Eckpunkt eines Polygons, den Start- oder den Zielpunkt. Jedem Knotenpunkt kann daher auch eine x- und y-Koordinate zugeordnet werden, eben genau die des erwähnten Punktes. Die zum Start- und Zielpunkt gehörigen Knotenpunkte werden im folgenden auch Start- und Zielknotenpunkt genannt.

3.3.2 Kanten

Die Kanten des Graphen korrespondieren zu den gefundenen Teilwegen. Eine Kante verbindet also zwei Knotenpunkte nur, wenn zwischen den zu den Knotenpunkten gehörigen Punkten ein Teilweg existiert. Es ergibt sich nun die Bewertung der Kante aus der Länge des Teilweges. Implizit ist diese Information bereits in den verbundenen Knotenpunkten enthalten. Die Länge des Teilweges entspricht ja gerade dem euklidischen Abstand der beiden Endpunkte.

3.3.3 Überflüssige Informationen im Graphen

Die oben beschriebenen Knotenpunkte und Kanten bilden also das gesamte Netzwerk der Teilwege in einem Graphen ab. In diesem Graphen existieren nun eine Vielzahl von Wegen, von denen später einer mit minimaler Bewertung gesucht wird. Werden nun Kanten oder Knotenpunkte aus dem Graphen entfernt, welche nicht Teil eines solchen Weges sind, so hat dies keine negativen Auswirkungen. Der Weg mit minimaler Bewertung bleibt gleich und auch neue Wege können nicht entstehen. Die Suche des Weges würde jedoch vereinfacht werden, da sich die Anzahl der restlichen Wege (also solcher Wege, die nicht optimal sind) verringert.

Streng genommen, stellen also alle Kanten und Knoten, welche nicht im optimalen Weg enthalten sind, eine überflüssige Information dar. Diese können die spätere Suche nach dem optimalen Weg verlangsamen, zusätzlich wird unnötig viel Speicher belegt. Es ist daher zu überlegen, ob nicht einige solcher Daten bereits beim Erstellen des Graphen entfernt werden können. Dem möglichen Zeitgewinn bei der Suche steht dabei natürlich zunächst einmal der Aufwand beim Filtern der Informationen gegenüber.

Knoten mit einem Nachbarn

Das folgende Beispiel soll zeigen, dass es tatsächlich solche überflüssigen Informationen in einem Graphen geben kann.

Es wäre möglich, dass die Nachbarschaft eines Knotens aus nur einem einzigen anderen Knotenpunkt besteht. Enthält ein Weg einen solchen Knotenpunkt, so muss er entweder ein Start- oder Endpunkt sein. In einem Weg steht vor und nach dem Knotenpunkt einer seiner Nachbarn. Die Definition verbietet nun, dass ein Knotenpunkt mehrfach in einem Weg enthalten ist. Der einzige benachbarte Knotenpunkt kann also nicht zugleich Vor- und Nachfolger in einem Weg sein. Hieraus folgt, dass dieser Knotenpunkt entweder keinen Vorgänger oder keinen Nachfolger hat. Er ist somit zwingend selbst der erste oder letzte Knotenpunkt in einem Weg.

Es ist nun bekannt, welche beiden Knotenpunkte am Anfang und Ende aller möglichen Wege stehen sollen, eben der Start- und Zielknotenpunkt. Ist also ein Knotenpunkt mit einem Nachbarn nicht selbst einer dieser Knoten, so kann man ausschließen, dass ein Weg zwischen Start und Ziel jemals über diesen führen kann. Ein solcher Knotenpunkt könnte daher aus dem Graphen entfernt werden. Ebenso natürlich die eine Kante, welche ihn mit seinem Nachbarn verbindet.

Diese Erkenntnis hilft nun bei der Behandlung von Hindernissen, die sich teilweise außerhalb des Weltmodells befinden. Das Modell der Welt wird durch ein Rechteck umschlossen und so würden einige Teilwege entstehen, die eben dieses schneiden. Der Teilweg, ist aber bis zum Schnittpunkt gültig, da dieser Teil noch innerhalb des Rechtecks liegt. Hier müsste sowohl ein Knotenpunkt als auch eine Kante, die eben den Teil repräsentiert, erzeugt werden. Dieser Knotenpunkt hat nun aber nur einen Nachbarn! Ein derartiger Knoten muss also, nach dem oben Gesagtem, gar nicht erst erzeugt werden, die Überprüfung dieser Teilwege kann sofort abgebrochen werden.

Die kürzeste Verbindung zwischen zwei Punkten

Aus der Dreiecksungleichung folgt, dass die kürzeste Verbindung zwischen zwei Punkten eine Strecke beziehungsweise ein Abschnitt einer Geraden ist. Anders interpretiert bedeutet dies, ein Weg über zwei Dreiecksseiten kann auf der dritten Seite abgekürzt werden.

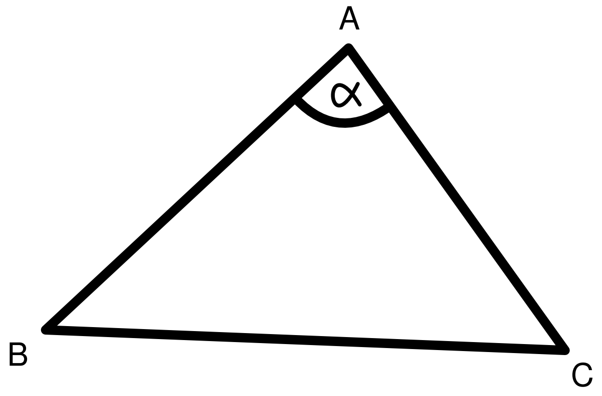

Abbildung 3-8: Ein Dreieck

Dieses Bild stellt beispielhaft eine solche Situation dar. Der Weg BAC ist länger als die Strecke BC allein. Man beachte, dass der Winkel α kleiner als 180° und größer 0° ist, wie es für jeden Winkel in einem Dreieck gilt.

Es soll nun versucht werden, diese Situation auf den Graphen zu übertragen. Der Punkt A korrespondiert zu einem Knotenpunkt im Graph, welcher nicht der Start- oder Zielknoten ist. Es gibt also noch ein zugehöriges Hindernis, A ist Eckpunkt des Polygons welches dieses Hindernis repräsentiert. Um eine allgemeine Aussage über einen einzelnen Knotenpunkt (unabhängig von allen anderen Ecken) zu treffen, soll eine ε-Umgebung um den Punkt A betrachtet werden. Es handelt sich hierbei um einen Kreis mit Mittelpunkt A und dem Radius ε, wobei dieser so klein gewählt sein soll, dass kein anderer Eckpunkt oder Knotenpunkt innerhalb des Kreises liegt.

Wir wissen also:

- ein Sektor des Kreises wird durch das Hindernis belegt
- außerhalb des Kreises befindet sich eine Anzahl Nachbarknoten

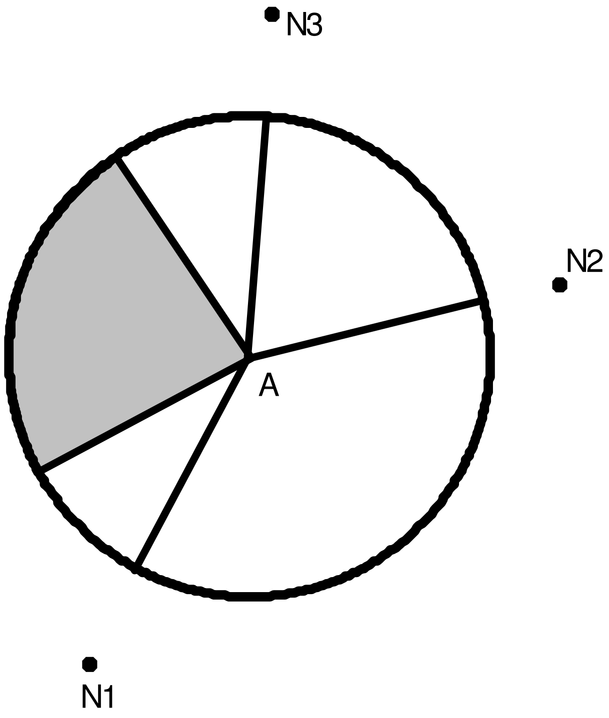

Abbildung 3-9: Ein Eckpunkt mit drei Nachbarn

In dem hier dargestellten Beispiel sei das Hindernis der graue Bereich links. Der Knoten A habe die Nachbarn N_1, N_2 und N_3. Es gibt nun einige Möglichkeiten, wie A in einem Weg enthalten sein kann. Es sollen die beiden Fälle N_1; A; N_2 und N_1; A; N_3 betrachtet werden. Die Strecken zwischen A und einem seiner Nachbarn haben jeweils einen Schnittpunkt mit dem Kreis, der die ε-Umgebung begrenzt. Verbindet man zwei solcher Schnittpunkte, so erhält man eine Sehne. Diese bildet mit dem Punkt A ein Dreieck.

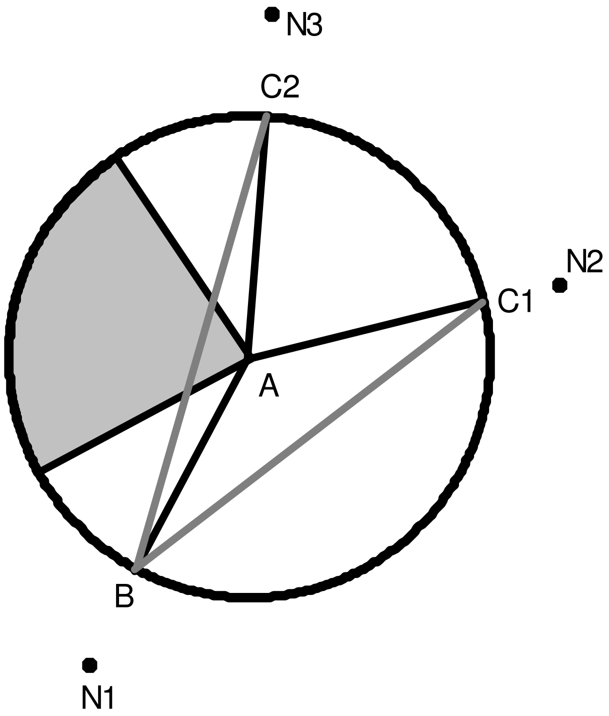

Abbildung 3-10: Potenzielle Abkürzungen

In der Abbildung 3-10 sind die Bezeichnungen an das Dreieck aus Abbildung 3-8 weiter oben angelehnt. Die Strecken BC_1 und BC_2 sind also theoretisch mögliche Abkürzungen. Während BC_1 tatsächlich ein möglicher Teilweg für den Roboter ist, kann BC_2 nicht benutzt werden. Es würde sonst eine Kollision mit dem Hindernis drohen, von dem A ein Eckpunkt ist. Da eine Abkürzung zwischen N_1 und N_2 existiert, bei welcher der Knotenpunkt A nicht benutzt werden muss, ist klar, dass der kürzeste Weg zwischen den beiden Nachbarn nicht über A verläuft.

Wie kann man nun erkennen, ob eine solche Abkürzung zwischen zwei Nachbarn begehbar ist, also das Hindernis nicht schneidet? Nochmals zurück zum Dreieck aus Abbildung 3-8 mit dem markierten Winkel α. Ein solches Dreieck soll gefunden werden. Man betrachtet die Winkel, die A mit den fraglichen Nachbarn einschließt. Die Schenkel bilden zwei Winkel. Falls nicht beide genau 180° sind, ist einer von beiden überstumpf. Demzufolge kann nur der zweite den Innenwinkel im Dreieck bilden. Wenn dieser das Hindernis überdeckt, so wird die Dreiecksseite einen Schnitt verursachen. Eine kollisionsfreie Abkürzung lässt sich dann nicht finden.

Man kann also für einen bestimmten Nachbarsknoten die Nachbarn finden, welche nicht auf einer Abkürzung erreicht werden können. Geometrisch lässt sich leicht ein Gebiet konstruieren, in dem alle diese Nachbarsknoten liegen. Es ist einmal begrenzt durch die Gerade, welche durch A und den Nachbarn führt. Die andere Grenze wird durch eine Seite des Hindernisses gebildet. Dieser Kreisbogen ist quasi nicht sichtbar, sondern durch das Hindernis verdeckt, siehe den dunklen Sektor in Abbildung 3-11.

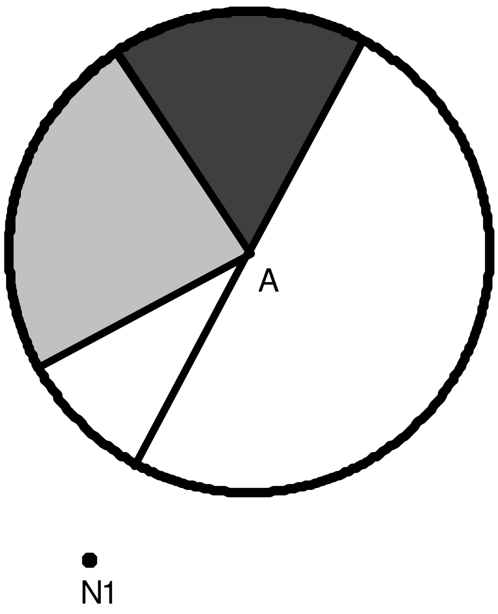

Abbildung 3-11: Ein Nachbar mit dem verdeckten Kreissektor

In dieser Darstellung ist der Bereich, in dem alle Nachbarn liegen, zu denen von N_1 aus nicht abgekürzt werden kann, dunkel gefärbt. Für alle Knotenpunkte, die im weißen Bereich liegen, lässt sich hingegen eine Abkürzung finden. Da in dem dunklen Bereich zumindest ein Knoten liegen sollte, nämlich die nächste Ecke des Hindernisses, kann man nicht mit Sicherheit ausschließen, dass ein kürzester Weg auch die Kante von N_1 nach A enthält. Es kann jedoch auch einige Knoten geben, für die es gar keinen solchen Bereich gibt. Damit wäre die Existenz eines optimalen Weges mit der entsprechenden Kante unmöglich. Dieser Fall tritt immer dann ein, wenn die Verlängerung der Kante durch das Hindernis geht. Es wird dann natürlich zwischen der Verlängerung und der Kante des Polygons kein Bereich mehr eingeschlossen. Der gesamte Kreisbogen außerhalb des Hindernisses ist sichtbar und wird nicht verdeckt. Um diesen Bereich zu konstruieren, benötigt man nur den Scheitelwinkel zu dem Innenwinkel der an A liegenden Ecke des Polygons.

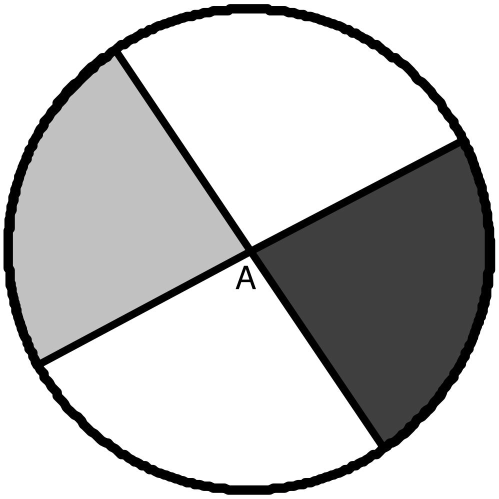

Abbildung 3-12: Der „tote“ Bereich des Eckpunkts

Es ergibt sich so nun ein relativ einfacher Test, um eine Kante zwischen A und einem Nachbarn auszuschließen. Man muss nur überprüfen, ob der Nachbar in einem Bereich liegt, wie er in Abbildung 3-12 dargestellt wird. Die durchschnittliche Anzahl der Knotenpunkte, die so ausgeschlossen werden können hängt von den Innenwinkeln der Polygone ab. Es ist nicht möglich, Polygone mit beliebig vielen kleinen Innenwinkel zu konstruieren. In einem Dreieck ergibt sich beispielsweise immer ein „durchschnittlicher Innenwinkel“ von 60 Grad. In einem Viereck sind es bereits 90 Grad. Es ist daher zu vermuten, dass der Test sich in fast allen Fällen rentiert, da 60 Grad oder mehr des Vollwinkels abgedeckt werden, d.h. also mindestens ein Sechstel der Ebene.

Einerseits kann so die Anzahl der Kanten im Graphen verringert werden. Es ist jedoch weiterhin auch möglich, die benötigte Rechenzeit zu verringern. Der Test, ob ein gültiger Teilweg zwischen zwei Knotenpunkten existiert, ist unnötig, wenn vorher bereits festgestellt wurde, dass ein solcher Teilweg ohnehin nie im kürzesten Weg enthalten sein wird. Da der Test, ob irgend eines der Hindernisse den Teilweg blockiert, durch einen einfachen Vergleich des Winkels ersetzt wird, ist gerade bei Problemen mit vielen Polygonen eine erhebliche Verringerung des Aufwands zu erwarten.

3.3.4 Gerichtete und ungerichtete Graphen

Da alle Kanten im Graphen einen möglichen Teilweg darstellen, wurde bisher stillschweigend von einem ungerichteten Graphen ausgegangen. Ein Teilweg kann in beiden Richtungen gleichermaßen benutzt werden, was eben durch eine ungerichtete Kante symbolisiert wird. Allein bei Start- und Zielknotenpunkten wäre sofort eine Entscheidung über die Richtung, in der anliegende Kanten benutzt werden, möglich. Diese Kanten würde vom Startknoten weg und zum Zielknoten hinführen.

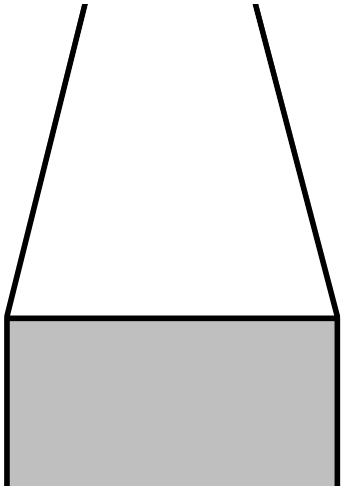

Abbildung 3-13: Ausschnitt eines Problems

Man betrachte einmal das Beispiel aus Abbildung 3-13: Wie man sieht, ist die eine Kante eines Rechtecks mit einem anderen Hindernis verbunden, welches sich oberhalb des Bildes befindet. Um die Darstellung möglichst einfach zu machen, wurde alles andere nicht gezeichnet. Die schwarzen Linien stellen hier die ungerichteten Kanten dar. Weiter oben wurde bereits einmal festgestellt, dass bestimmte Kombinationen von Kanten nicht sinnvoll sind, da sie ohnehin abgekürzt werden könnten. Nun kann beispielsweise die rechte Kante, welche nach oben führt, nicht komplett entfernt werden, da sie mit der rechten Kante des Rechtecks ein Wegstück bildet, das nicht einfach abgekürzt werden kann. Allerdings ist es nicht möglich, zusammen mit der oberen Kante des Rechtecks einen sinnvollen Weg zu finden. Im ungerichteten Graphen kann jedoch durchaus ein solcher Weg gebildet werden.

Man kann nun einen gerichteten Graphen erzeugen, in dem dies nicht möglich ist. Dazu werden zunächst jedem Eckpunkt zwei Knotenpunkte zugeordnet. Diese sind wieder entsprechend der Teilwege verbunden. Für die ersten Knotenpunkte werden nun die Bögen, welche um das Hindernis führen, im Uhrzeigersinn um das Hindernis orientiert. Die Restlichen werden genau entgegengesetzt, das heißt gegen den Uhrzeigersinn, orientiert. Die Bögen, die Verbindungen zu anderen Hindernissen darstellen, werden entsprechend der Kante orientiert, mit der ein sinnvoller Weg möglich ist. Die folgende Abbildung 3-14 zeigt dies für das obige Beispiel.

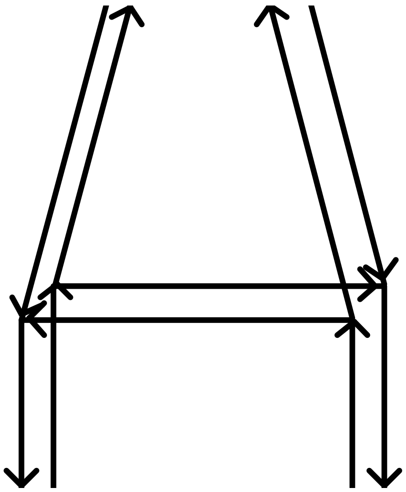

Abbildung 3-14: Ein möglicher gerichteter Graph

Die eigentlich übereinander liegenden Bögen wurden leicht versetzt dargestellt, um den Richtungssinn besser zu verdeutlichen.

Die Anzahl der Bögen, welche an einem Knotenpunkt beginnen oder enden, entspricht der Anzahl Kanten, die im ungerichteten Graph mit dem entsprechenden Knoten verbundenen sind. Da für Wege jedoch stets nur ausgehende Bögen interessant sind (auf den anderen würde man sich ja rückwärts bewegen), ist die durchschnittliche Anzahl der Möglichkeiten, wie ein Weg weitergeführt werden kann, in dem gerichteten Graphen halbiert. Dieser Vorteil wird aber mit der Verdopplung der Anzahl der Knotenpunkte und der (gerichteten) Kanten teuer erkauft. Es wird dann auch doppelt soviel Speicher für den Graphen benötigt. Weiterhin ergeben sich auch noch ganz neue Wege. Diese können ja die Eckpunkte eines Polygons mehrmals enthalten, da jedem Eckpunkt zwei verschiedene Knotenpunkte zugeordnet sind. Auch beim Erstellen des Graphen wird der Aufwand größer.

Einen solchen gerichteten Graphen zu erzeugen kann daher nur sinnvoll sein, wenn Speicherbedarf und die Zeit zum Erstellen des Graphen eine untergeordnete Rolle spielen. Trotzdem sollte auch dann praktisch überprüft werden, ob diese Methode in den typischen Fällen, die man bearbeiten möchte, eine Verbesserung der Laufzeit bringt. Es ist zu vermuten, dass sich die Zeit im besten Fall höchstens halbieren wird.

3.4 Der kürzeste Weg

3.4.1 Einleitung

Die Suche nach kürzesten Wegen in gerichteten und ungerichteten Graphen ist ein wichtiges Anwendungsgebiet der Graphentheorie. Allgemeiner gefasst, sucht man nach einem Weg zwischen einem vorgegebenen Start- und Zielknoten, für den die Summe der Kantenbewertungen minimal ist. Diese Bewertungen können nicht nur als Länge sondern auch als Kosten, Zeiten usw. interpretiert werden. Es sind aus der Literatur zur Angewandeten Graphentheorie verschiedene Algorithmen bekannt, welche dieses Problem lösen. Beschrieben sind diese unter anderem in [Tur96, S. 239ff] und [Sch03, S. 87ff]. Diese Algorithmen unterscheiden sich bezüglich der Voraussetzungen und der Effizienz, mit der das Problem gelöst werden kann. Dabei ist ein Zusammenhang zu erkennen: je strenger die Voraussetzungen gefasst werden, um so effizienter kann ein entsprechender Algorithmus arbeiten.

Von folgenden Voraussetzungen kann ausgegangen werden:

- es wird lediglich der Weg zwischen einem Start- und Zielpunkt gesucht
- es gibt keine negativen Kantenbewertungen bzw. Kantengewichte
- die Kantenbewertungen sind euklidische Entfernungen in der Ebene

Der Algorithmus von Dijkstra ist der klassische Algorithmus, welcher hier zum Einsatz kommt. Eine Modifikation dieses Verfahrens, der A*-Algorithmus, soll ebenfalls betrachtet werden.

3.4.2 Algorithmus von Dijkstra

Die Arbeitsweise dieses Algorithmus kann man sich anschaulich in etwa wie folgenden physikalischen Prozess vorstellen [MN99, S. 254]. Es existiere ein Netzwerk aus Stromkabeln, welches dem Graphen entspricht. Dieses Netz wird am Startknoten zum Zeitpunkt 0 unter Strom gesetzt. Die Zeit, die der Strom benötigt, um ein Kabel zu durchqueren, ist proportional zum Kantengewicht im Graph. Somit ist die Zeit, die benötigt wird, bis der Strom einen Knoten erreicht, proportional zur Summe der Kantengewichte. Der Algorithmus führt nun keine stetige Simulation davon durch. Es ist auch irrelevant, wann der Strom die Hälfte oder zwei Drittel des Kabels zurückgelegt hat. Interessant ist nur der Zeitpunkt, zu dem ein Knoten erreicht wird.

Am Beginn ist diese Zeit nun nur am Startknoten bekannt. Von diesem aus schaut man sich nun alle Nachbarknoten an. Anhand der Kantengewichte (gewissermaßen der „Kabellängen“) kann für diese Nachbarknoten eine obere Schranke für die Weglänge gefunden werden. In diesen Knoten wird nun gespeichert, dass der Strom zum Zeitpunkt t hier ist, wenn er vom Knoten v kommt. Mit Hilfe dieses Vorgängers v wird der Weg später rekonstruiert. Man hat nun eine Menge Knoten: die Nachbarknoten des Startknotens. In jedem Knotenpunkt ist eine Zeit gespeichert, zu welcher der Strom spätestens angekommen ist. Der Algorithmus fährt nun bei dem Knoten fort, an dem diese Zeit minimal ist. Dieser kann den Strom nicht noch früher (durch ein anderes

Kabel) erhalten, da alle Kantenbewertungen positiv sind. Der Knoten wird aus der Menge entfernt und wiederum die Minimalzeit in all seinen Nachbarknoten berechnet. Wird ein Knoten einmal aus der Menge entfernt, so ist seine Entfernung minimal und er kann nie wieder der Menge hinzugefügt werden.

In jedem Schritt wird ein Knoten aus der Menge entfernt. Der Algorithmus ist beendet, wenn alle Knoten aus der Menge entfernt sind. Da deren Anzahl endlich ist, muss der Algorithmus nach endlich vielen Schritten beendet sein. Danach ist in allen Knotenpunkten die berechnete Zeit, also die Summe der Kantengewichte, sowie der Vorgänger auf dem kürzesten Weg zu diesem Knotenpunkt gespeichert. Enthält der Graph Knotenpunkte, die vom Startpunkt nicht erreichbar sind, so werden diese nie vom Algorithmus bearbeitet (sie können nie Nachbarn der untersuchten Knotenpunkte sein). Dort ist dann weder eine Entfernung noch ein Vorgänger gespeichert.

Wird der Algorithmus in der oben beschriebenen Weise ausgeführt, so wird durch eine sogenannte „Rückwärtsrechnung“ der kürzeste Weg zu jedem Knoten gefunden. Ist der Weg zu einem bestimmten Knoten gesucht, so kann der Algorithmus abgebrochen werden, sobald dieser aus der Menge entfernt wurde. Die durchschnittliche Laufzeit wird so zwar geringer, die Komplexität jedoch nicht, da bei der Berechnung der Komplexität vom worst case, also dem Fall mit dem größtem erforderlichen Aufwand, ausgegangen wird. Die folgende Abbildung 3-15 illustriert diesen oben beschriebenen Sachverhalt.

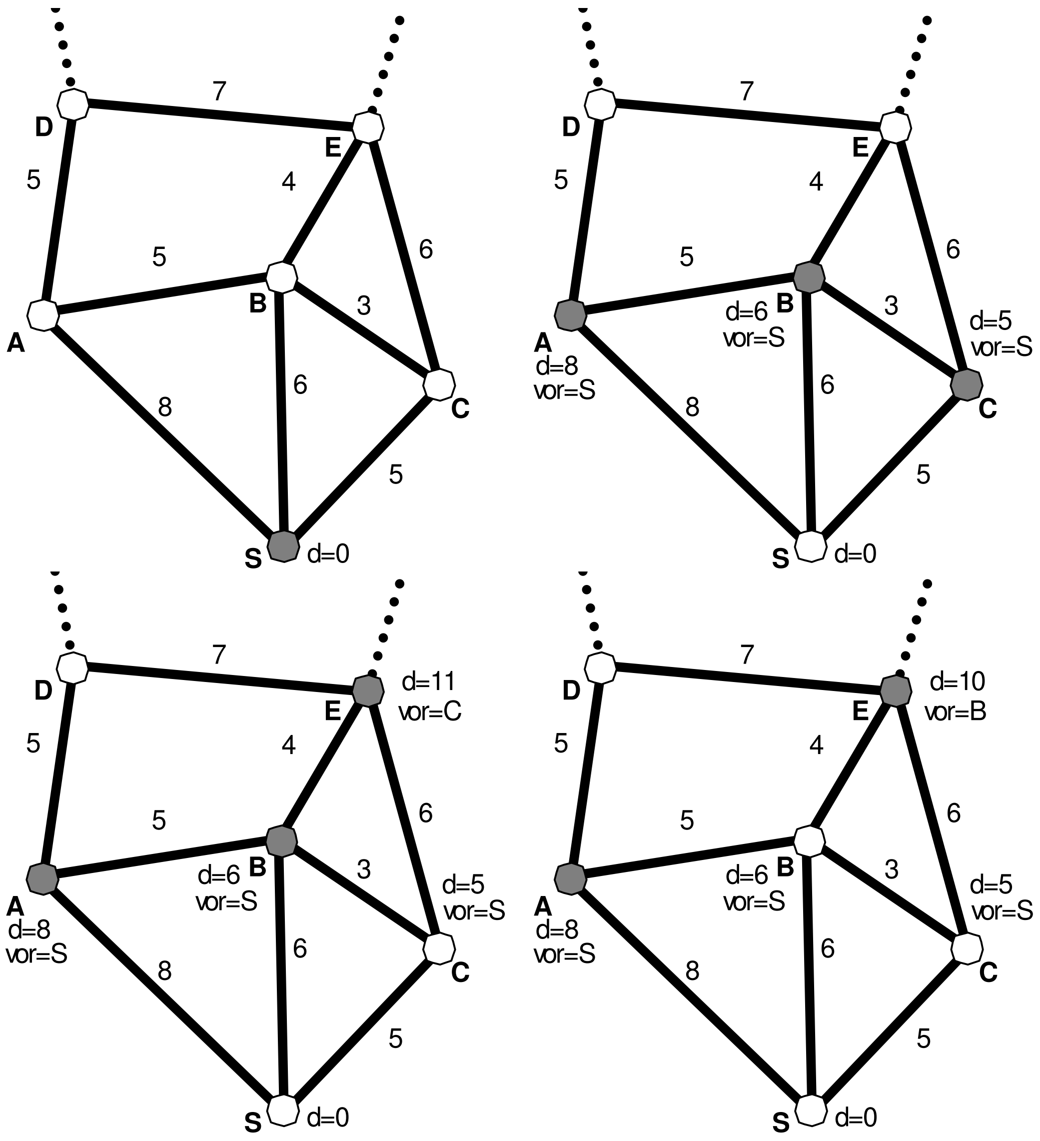

Abbildung 3-15: vier Schritte mit dem Algorithmus von Dijkstra

Auf dem ersten Bild ist nur der Startknoten S bekannt. Im ersten Schritt werden die drei Nachbarknoten von S betrachtet. Jeder wird der Menge hinzugefügt und mit den Kosten d versehen. Im zweiten Schritt wird das minimale d in C gefunden. C hat die Nachbarn B und E. B wird nicht verändert, da der Weg über C länger wäre. Für E wird ein erster Weg gefunden. Dieser geht über C und hat die Kosten d=11. Im nun folgenden Schritt, ist d bei B minimal. Der Weg nach A ist wiederum uninteressant. Nach E wird jetzt jedoch ein besserer Weg gefunden, d und vor werden entsprechend angepasst.

Der formalisierte Algorithmus lässt sich wie folgt darstellen:

Eingabe:

- Graph G mit (V,E)
- eine Kostenfunktion $c(e)$ für alle $e \in E$
- Startknoten $s \in V$; Zielknoten $z \in V$

```
1. Setze $vor(v) = \emptyset$ für alle $v \in V$; $d(s) = 0$; $Q = \{s\}$
2. Wähle $a \in Q$ mit $d(a)$ minimal
3. wenn $a = z$ dann Stop, Ziel erreicht!
4. Entferne $a$ aus $Q$
5. Für alle $e = (v1,v2)$ mit $v1 = a$:
       wenn $vor(v2) = \emptyset$ und $v2 \neq s$,
           dann füge $v2$ zu $Q$ hinzu; $d(v2) = d(v1) + c(e)$; $vor(v2) = v1$
       sonst wenn $d(v2) > d(v1) + c(e)$,
           dann $d(v2) = d(v1) + c(e)$; $vor(v2) = v1$
6. wenn $Q \neq \emptyset$ gehe nach 2
7. Stop, Ziel ist nicht erreichbar
```

Betrachtung zur Laufzeit

Wenn wir uns die Laufzeit des Algorithmus von Dijkstra anschauen, so hängt diese von der Art der Implementierung ab. Wir wollen annehmen, dass die Menge Q der zu untersuchenden Knoten als simple Liste gespeichert ist. Die Suche nach dem Knoten mit minimalem d benötigt hier eine Zeit, welche proportional zur Anzahl der Knoten ist. Bezeichnet man die Anzahl aller Knoten mit n, so beträgt der Aufwand O(n). Jeder Knoten kann höchstens einmal in Q eingefügt werden. So kann gezeigt werden, dass der Gesamtaufwand $O(n^2)$ beträgt, wenn eine Liste benutzt wird [Tur96, S. 253].

Ein wesentlicher Punkt bei der Berechnung der Laufzeit ist die Suche nach dem Knotenpunkt mit minimalen Kosten. Die oben erwähnte Liste bedeutet hier einen recht großen Aufwand. Die in der Informatik bekannte Datenstruktur „Prioritätsliste" erlaubt eine effizientere Ausführung. Sie bietet die benötigten Operationen: Werte einfügen, Werte verringern und minimalen Wert finden. Auch Prioritätslisten können wiederum auf verschiedene Arten implementiert werden. So wird beispielsweise durch den Fibonacci Heap [Koz91, S. 44ff] eine Gesamtlaufzeit von O(m + n log n) erreicht. In [MN99, S. 149] findet sich eine Übersicht über die verschiedenen Implementierungen von Prioritätslisten in der LEDA-Bibliothek und ihre Effizienz. In [MN99, S. 152] finden sich die daraus resultierenden Komplexitäten des Algorithmus von Dijkstra.

Die theoretische Betrachtung der Laufzeiten im schlechtesten Fall erklärt letztendlich nicht, welcher Algorithmus in der Praxis eingesetzt werden soll. Die Dichte eines Graphen bezeichnet das Verhältnis der Anzahl der Knotenpunkte zur Anzahl der Kanten. Sie ist ein wesentlicher Faktor für die tatsächliche Laufzeit.

3.4.3 A* - Algorithmus

Der Algorithmus A* (gesprochen A Stern) wird erstmals in [HNR68] erwähnt. Er stellt eine Variation des Algorithmus von Dijkstra dar [Tur96, S. 257]. Es wird eine Heuristik mit dem Ziel benutzt, die Anzahl der zu untersuchenden Knoten zu verringern. Das Verfahren baut auf einer Schätzfunktion auf. Diese schätzt die restlichen Wegkosten von einem beliebigen Knoten bis zum Ziel. Diese Funktion muss so gewählt werden, das man eine untere Grenze für die Restkosten erhält.

Definition:

Eine Schätzfunktion, welche die Kosten nie überschätzt und nie negative Funktionswerte erzeugt, heißt **zulässig**.

Mit dieser Funktion wird nun die Auswahl des nächsten zu bearbeitenden Knoten verbessert. Im Algorithmus von Dijkstra, besteht das einzige Kriterium in den bis zu diesem Knoten entstandenen Kosten. A* sucht nun einen Knoten, bei dem die Summe aus bisherigen und den geschätzten Restkosten minimal ist. Dies bedeutet im Ergebnis, dass bei A* die Suche nicht wahllos in allen Nachbarknotenpunkten verläuft. Knotenpunkte, welche dem Zielknotenpunkt nahe sein könnten, werden bevorzugt gewählt.

Damit diese Heuristik ein gutes Ergebnis erzeugt, ist es nötig, die Funktion so zu bestimmen, dass die wahren Restkosten möglichst wenig unterschätzt werden. Die Schätzfunktion soll also eine untere Schranke der restlichen Wegkosten sein, die möglichst scharf ist. In einem allgemeinen gewichteten Graphen ist es jedoch quasi nicht möglich, eine solche günstige Funktion zu finden. Dazu muss die geometrische Deutung des Graphen betrachtet werden. Den Knoten des Graphen sind Punkte in der Ebene zugeordnet werden. Ebenso sind den Kanten Strecken zugeordnet. Die Gewichte der Kanten entsprechen den Längen der Strecken. Nun ergibt sich in natürlicher Weise eine mögliche Schätzfunktion: der euklidische Abstand zum Ziel. Aus der Dreiecksungleichung folgt, dass diese Schätzfunktion zulässig ist. Es kann keinen Weg über zwei (oder mehr) Kanten geben, welcher kürzer wäre, als die direkte Verbindung.

Der kürzeste Weg

Kurze Darstellung des Algorithmus:

Eingabe:

- Graph G mit (V,E)
- eine Kostenfunktion $c(e)$ für alle $e \in E$
- Startknoten $s \in V$; Zielknoten $z \in V$
- Schätzfunktion für die Restkosten bis zum Ziel $h(v)$ für alle $v \in V$

```
1. Setze $vor(v)= \emptyset$ für alle $v \in V$; $d(s)= 0$; $Q= \{s\}$
2. Wähle $a \in Q$ mit $d(a) + h(a)$ minimal
3. wenn $Q \neq \emptyset$ gehe nach 2
4. Entferne a aus Q
5. Für alle $e= (v1,v2)$ mit $v1= a$:
       wenn $vor(v2) = \emptyset$ und $v2 \neq s$,
           dann füge v2 zu Q hinzu; $d(v2)= d(v1) + c(e)$; $vor(v2)= v1$
       sonst wenn $d(v2) > d(v1) + c(e)$,
           dann $d(v2)= d(v1) + c(e)$; $vor(v2)= v1$
6. wenn $a = z$ dann Stop, Ziel erreicht!
7. Stop, Ziel ist nicht erreichbar
```

Bestimmte zulässige Schätzfunktionen können zu einem Verhalten führen, welches im Algorithmus von Dijkstra nicht zu beobachten ist: Ein Knotenpunkt wird ausgewählt, obwohl der kürzeste Weg zu ihm noch nicht feststeht. Es kann also nicht mehr festgestellt werden, ob der Weg mit den geringsten Kosten zu einem Knotenpunkt bereits gefunden wurde.

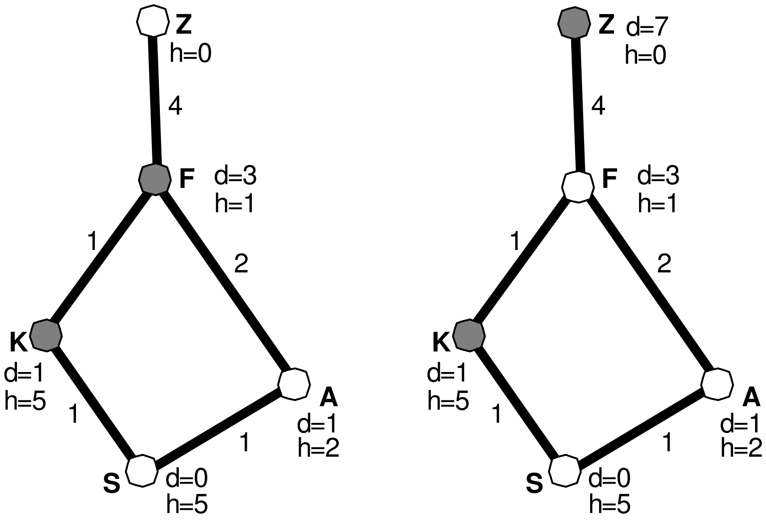

Abbildung 3-16: F wird zu früh entfernt

Mit h ist hier der von der Heuristik geschätzte Restweg bezeichnet. Der kürzeste Weg nach F ist offensichtlich S→K→F. Trotzdem wird F schon gewählt, obwohl noch die längere Verbindung S→A→F favorisiert ist. Der Knotenpunkt K gehört noch zur Menge der Knoten, aus denen weitere Knotenpunkte zur Untersuchung ausgewählt werden (in der Abbildung ersichtlich durch die graue Färbung). Würde ein solches Verhalten am Zielknoten auftreten, so könnte man den Algorithmus nicht vorzeitig abbrechen. Das Entfernen des Zielknoten würde dann ja nicht mehr garantieren, dass der bisherige Weg optimal ist. Es könnte noch ein Knotenpunkt existieren, über den, wie hier K, eine kürzerer Weg zum Ziel existiert. Die genaue Untersuchung wird nun aber zeigen, dass dieses Verhalten nicht am Zielknoten auftreten kann.

Der fälschlich gewählte Knotenpunkt sei F. Zunächst ist festzustellen, unter welchen Gegebenheiten dieser Knotenpunkt gewählt wird, obwohl der optimale Weg vom Start nach F noch nicht gefunden wurde. Dafür muss natürlich ein solcher kürzerer

Weg existieren, der erst zu einem Teil bekannt ist. Der Weg ist bisher nur bis zum Knotenpunkt K bekannt. K ist bereits in die Menge der zu untersuchenden Knotenpunkte eingefügt. Wäre dem nicht so, würde der Weg nie mehr gefunden werden, denn dazu muss K gewählt werden. Wenn der optimale Weg vom Start nach F bis K bekannt ist, so wurde in K bereits die minimale Distanz gespeichert. Lediglich die kürzeste Verbindung von K nach F ist noch nicht gefunden.

Seien nun die Minimalkosten vom Knotenpunkt x zum Knotenpunkt y mit m(x,y) bezeichnet, sowie mit e(x) der Fehler der Schätzfunktion am Knotenpunkt x. Knotenpunkt F wird gewählt, obwohl die bisherige Distanz zu hoch berechnet wurde. Der Betrag, den die Distanz noch zu groß ist, sei g. Diese folgenden vier Gleichungen bzw. Ungleichungen ergeben sich nun direkt aus der Definition der Funktionen:

$$m(x,y) \leq m(x,e) + m(e,y) \qquad \text{für alle } e,x,y \in E \tag{1}$$

$$m(x,z) - e(x) = h(x) \qquad \text{für alle } x \in E \tag{2}$$

$$m(s,F) = d(F) - g \tag{3}$$

$$m(s,K) = d(K) \tag{4}$$

Da der optimale Weg vom Start nach F über K führt, gilt:

$$m(s,K) + m(K,F) = m(s,F)$$

Der rechte Term wird nun entsprechend (3) ersetzt und danach umgeformt:

$$m(s,K) + m(K,F) = d(F) - g$$

$$m(s,K) + m(K,F) + g = d(F) \tag{5}$$

F wird gewählt, weil d(F) + h(F) minimal ist:

$$d(F) + h(F) < d(K) + h(K)$$

Die Funktion d(F) wird nun durch den linken Term in (5) ersetzt. Danach ist eine Vereinfachung durch (4) möglich.

$$m(s,K) + m(K,F) + g + h(F) < d(K) + h(K)$$

$$m(K,F) + g + h(F) < h(K)$$

Durch Umformung ergibt sich nun:

$$g < h(K) - h(F) - m(K,F)$$

Nun wird die Beziehung aus (2) angewandt.

$$g < m(K,z) - e(K) - m(F,z) + e(F) - m(K,F)$$

Nachdem die Ausdrücke auf der rechten Seite etwas umgeordnet wurden, ist erkennbar, dass eine Abschätzung mit Hilfe von (1) möglich ist.

$$g < m(K,z) - m(K,F) - m(F,z) + e(F) - e(K)$$

$$g < e(F) - e(K)$$

Der Fehler der Schätzfunktion ist in jedem Knotenpunkt, also auch in K, größer oder gleich 0, daher folgt:

$$g < e(F)$$

Somit wurde gezeigt, dass F nicht der Zielknoten sein kann. Im Ziel ist der Fehler jeder zulässigen Schätzfunktion gleich null und kann somit unmöglich größer als die positive Zahl g sein. Es ist sogar ganz ausgeschlossen, dass ein Knotenpunkt F überhaupt existiert, wenn die Schätzfunktion die folgende Bedingung [Tur96, S. 259] erfüllt:

Der kürzeste Weg

Definition:

Eine Schätzfunktion, welche die Ungleichung

$$0 \leq h(v1) \leq c(e) + h(v2) \qquad \text{für alle } e=(v1,v2) \in E$$

erfüllt, heißt **konsistent**.

Die obige Ungleichung hat die Form einer Dreiecksungleichung. Diese Ungleichung ist für Entfernungen in der Ebene stets erfüllt. Somit ist gezeigt, dass der euklidische Abstand zum Ziel eine konsistente Schätzfunktion darstellt.

Nehme man den Knotenpunkt v, dessen kürzeste Verbindung zum Ziel über die Knotenpunkte v1,v2, ..., vn führen soll. Durch mehrfache Anwendung der obigen Ungleichung erhält man nun:

$$h(v) \leq c(e) + h(v1) \qquad \text{mit } e=(v,v1)$$

$$h(v) \leq m(v,v1) + h(v1)$$

$$h(v) \leq m(v,v1) + c(e) + h(v2) \qquad \text{mit } e=(v1,v2)$$

$$h(v) \leq m(v,v2) + h(v2)$$

$$\dots$$

$$h(v) \leq m(v,z) + h(z)$$

Sei nun h(z)=0. Dann ergibt sich, dass die Bedingungen für eine zulässige Schätzfunktion erfüllt sind.

Wir halten also fest: der euklidische Abstand zum Ziel ist eine zulässige und konsistente Schätzfunktion.

Betrachtung zur Laufzeit

Für die Komplexität von A* gelten im wesentlichen die gleichen Argumente wie für den Algorithmus von Dijkstra. Der Hauptunterschied zwischen den Algorithmen ist die Berechnung der Schätzfunktion. Angenommen die Berechnung der Schätzfunktion hat einen konstanten Aufwand, also O(1). Eine zulässige Schätzfunktion, die dies erfüllt, ist die Funktion h(v)= 0. Diese ist aber denkbar ungünstig. Wird sie gewählt, verhält sich A* genau wie der Algorithmus von Dijkstra. Es wird dann natürlich auch seine Laufzeit angenommen. Da diese Schätzfunktion den größtmöglichen Fehler hat, gibt es keine Schätzfunktion mit konstantem Aufwand, mit der die Komplexität von A* größer als die des Algorithmus von Dijkstra ist. Hat man eine Schätzfunktion zur Verfügung, die den Restweg genau einschätzt, dann besucht A* auch nur Knotenpunkte, welche auf dem kürzesten Weg zum Ziel liegen. Im Allgemeinen steht eine solche Schätzfunktion natürlich nicht zur Verfügung, bzw. ist der Aufwand zu deren Bestimmung dann nicht mehr konstant sondern entspricht dem des Algorithmus von Dijkstra. Die Schätzfunktion muss für alle untersuchten Knoten aufgerufen werden. Benötigt deren Bestimmung einen Aufwand von O(n) oder schlechter, so benötigt allein die Berechnung der Schätzfunktion im schlechtesten Fall einen Aufwand von $O(n^2)$, womit eine Verbesserung des Algorithmus von Dijkstra nicht möglich wäre. Eventuell bliebe Raum für eine Verbesserung mit einer Schätzfunktion die einen logarithmischen Aufwand hat. Eine Aussage für den average case lässt sich aber nur sehr schwer treffen, da es auch von der Schätzfunktion selbst abhängt, wie oft diese letztendlich berechnet werden muss.

Die Anzahl von besuchten Knotenpunkten beim A*-Algorithmus liegt also zwischen den beiden Extremen, der Anzahl der durch den Algorithmus von Dijkstra bearbeiteten Knotenpunkten und der Anzahl der Knotenpunkte auf dem kürzesten Weg. Sie bestimmt letztendlich die tatsächliche Laufzeit und ist abhängig von der Güte der Schätzfunktion, der Gestalt des Graphen und dem vorgegeben Start- und Endknotenpunkt.

3.5 Arbeitsschritte

Wird ein großes Problem in kleinere Teilprobleme zerlegt, geschieht dies meist zur Vereinfachung. Das Problem als Ganzes zu erfassen und zu lösen, ist weitaus schwieriger als nur einen kleinen Teil davon zu bearbeiten. Kommt man aber zur praktischen Anwendung der erarbeiteten Lösung, so kann man manchmal einen weiteren Vorteil aus diesem „Schritt für Schritt“-Ansatz nutzen. Arbeitet man mit vielen Problemstellungen, die sich stark ähneln, so kann man beobachten, dass einzelne Teilschritte der Lösung bereits früher genau so ausgeführt wurden. Das heißt: ein Ergebnis eines bestimmten Arbeitsschrittes kann nicht nur für eine, sondern gleiche für mehrere Probleminstanzen benutzt werden.

3.5.1 Unnötige Arbeitsschritte vermeiden

Man betrachte die Eingabedaten, die zur Berechnung des optimalen Weges nötig sind. Dies sind zunächst die Polygone, welche die Bereiche beschreiben, mit denen der Roboter kollidieren würde. Hinzu kommt die aktuelle Position des Roboters, sowie die Position, in die er sich bewegen soll, der Start- und Endpunkt also. Solange der immergleiche Roboter in der gleichen Umgebung operiert, werden sich die Hindernisse jedoch nicht verändern. Mit einiger Wahrscheinlichkeit trifft dies beispielsweise auf einen Fabrikroboter zu, der in einer Werkhalle arbeitet. Es erscheint daher sinnvoll einmal anzunehmen, dass man zwar immer wieder verschiedene Start- und Zielpunkte hat, die restliche Problemstellung, das heißt die Lage und Gestalt der Polygone, hingegen gleich bleibt.

Tatsächlich bleibt ein Großteil der berechneten Teilwege gleich, solange nur Start und Ziel verändert werden. Verbindet eine solche Strecke die Eckpunkte von Hindernissen, so ist diese nach wie vor gültig. Und eine Kollision ist auch erst zu befürchten, wenn die Strecke ein Hindernis schneidet. Solange jedoch kein neues Hindernis auftaucht und auch die bereits Bestehenden nicht verschoben oder verformt werden, ist auch dies ausgeschlossen.

Der Graph verändert sich also nur in genau den zwei Knotenpunkten, welche dem Start- und Zielpunkt zugeordnet sind, sowie den damit verbundenen Kanten. Es liegt daher nahe, zunächst einmal nur einen Graphen zu erstellen, welcher aus den Informationen der Hindernisse aufgebaut wird. Dieser „Basisgraph“ kann dann für jede beliebige Lage von Start- und Zielpunkt benutzt werden. Dazu werden ihm die beiden entsprechenden Knotenpunkte, sowie die dort beginnenden Teilwege hinzugefügt. Das Resultat dieser Überlegungen liefert die folgende Aufteilung der Arbeit:

1. Erstellen des Basisgraphen aus den Hindernissen

2. Einfügen von Start- und Zielknotenpunkt

3. Bestimmung des kürzesten Weges in diesem Graphen

Nach jedem Schritt wird das Ergebnis zwischengespeichert, so dass es mehrmals abgerufen und verwendet werden kann. Jeder Schritt ist abhängig vom Ergebnis des vorhergehenden Schrittes sowie eventuell einem Teil der Eingabedaten. Wird dieser Teil der Eingabedaten verändert, so wird das Ergebnis des entsprechenden Arbeitsschrittes ungültig und damit auch das aller folgenden Schritte.

Wenn nun stets genau verfolgt wird, wann sich Eingabedaten ändern, kann genau bestimmt werden, welche Berechnungen erneut ausgeführt werden müssen oder ob ein zwischengespeichertes Ergebnis weiter verwendet werden kann. Es wird hierzu lediglich eine konstante Zeit benötigt, um diese Überprüfung durchzuführen, sowie der Speicherplatz, der durch die Zwischenergebnisse belegt ist. Wird allerdings festgestellt, dass Schritt 1 nicht erneut ausgeführt werden muss, so verringert sich die Gesamtkomplexität der Lösung, da Schritt 1 mit $O(n^2 \log n)$ die höchste Komplexität aller Schritte aufweist. Es wird also nicht nur ein konstanter Teil der Arbeitszeit gespart, sondern der gesparte Teil wächst sogar mit der Größe des Problems an.

4 Erstellung der Software

4.1 Programmiersprache

Als Programmiersprache soll C++ zum Einsatz kommen. Ihr größter Vorteil liegt in der Hardwarenähe und der somit erreichbaren Performance bei moderaten Anforderungen an den Speicherplatz. Weiterhin stehen für C++ viele Programmbibliotheken zur Verfügung, die bei der Entwicklung und Weiterentwicklung des Projektes nützlich sein können. Neben den weiter unten aufgeführten Bibliotheken wäre hier hauptsächlich an Pakete zu denken, welche die Steuerung eines Roboters erlauben bzw. eine Kommunikation mit dem Bordcomputer des Roboters aufbauen.

C++ ist als objektorientierte Sprache auch gut geeignet, Objekte und deren Beziehungen wiederzugeben. Das Konzept der Vererbung und der Polymorphie erlaubt es, Objekte zu konstruieren, welche Teilprobleme lösen. Bei der Benutzung des Objektes, ist es unerheblich, was genau das Objekt dazu tun muss. Man muss nur wissen, dass eine Instanz des Objektes das Teilproblem lösen kann, und nicht welcher genaue Algorithmus dazu benutzt wird. Wichtig ist nur, dass sich alle Objekte nach außen hin gleich verhalten: Sie erzeugen eine korrekte Lösung.

4.2 Bibliotheken

4.2.1 Library of Efficient Datatypes and Algorithms

“Library of Efficient Datatypes and Algorithms“ oder kurz LEDA bedeutet soviel wie Sammlung effizienter Datentypen und Algorithmen. Zu den Hauptthemen, mit denen sich diese Bibliothek beschäftigt, gehören unter anderem Geometrie und Graphen. Diese Bibliothek ist daher exzellent geeignet, das mathematische Rückgrat dieses Projektes zu bilden. Einer der Grundgedanken von LEDA ist es, vorhandene

Implementierungen so zu gestalten, dass sie wiederverwendbar sind oder anders gesagt: das Rad nicht immer wieder neu zu erfinden [MN99]. Die Entwickler hinter LEDA haben festgestellt, dass es nicht effizient ist, Algorithmen für jedes Projekt neu zu implementieren. Algorithmen eines anderen Projektes zu übernehmen, stellt sich jedoch als recht problematisch heraus, wenn diese speziell für dieses eine Projekt erstellt wurden. Es ist dann im Allgemeinen nicht möglich, diese Algorithmen zu kombinieren, da sie nicht abstrakt genug oder gar unvollständig sind. Daher will LEDA diese Aufgabe übernehmen, indem es feste Schnittstellen dafür definiert. Die Benutzung dieser Schnittstellen ermöglicht dann die Verwendung einer Funktion unabhängig von der Kenntnis des exakten Algorithmus. Das bedeutet auch, wenn eine zukünftige Version von LEDA diesen Algorithmus verbessert, muss einfach nur diese neue Version benutzt werden, um davon zu profitieren.

4.2.2 Microsoft Foundation Class

Heutzutage ist das WIMP System die gebräuchlichste Benutzerschnittstelle für interaktives Arbeiten. Sie setzt sich aus Windows, Icons, Maus und Pulldown Menus zusammen. Unter anderem bietet das Betriebssystem Windows Funktionen an, um Benutzerschnittstellen dieser Art zu erstellen. Die „Microsoft Foundation Class“ oder kurz MFC [MSDN2] bildet eine OOP basierte Bibliothek, um diese Schnittstellen zu benutzen. Weiterhin gibt es spezielle Editoren, welche eine graphische Bearbeitung des Aussehens der Benutzerschnittstelle ermöglichen. Diese können daraus dann den relevanten Quellcode generieren, welcher das festgelegte Layout mit Hilfe der MFC darstellt. Dem Programmierer wird auf diese Art etwas Arbeit abgenommen, da er sich nun nur noch um die Funktion hinter der Benutzerschnittstelle kümmern muss und nicht mehr um diese selbst. Die MFC bietet weiterhin Funktionen zur Darstellung von zweidimensionaler Pixelgrafik an. So sind beispielsweise Funktionen zum Zeichnen von Linien und Füllen von Polygonen enthalten.

4.2.3 Standard Template Library

Die Standard Template Library ist ein Teil der C++ Standard Bibliothek. Sie enthält nützliche Container, welche als Templates ausgeführt sind. Templates ermöglichen in C++ generische Algorithmen, die mit jedem Objekttyp funktionieren. Speziell werden beispielsweise Listen von Punkten und Polygonen benötigt. Da diese Bibliothek weit verbreitet ist und daher als effizient und fehlerfrei angesehen werden kann, sollte dies dem erneuten Implementieren der nötigen Container vorgezogen werden.

4.3 Quellcodegestaltung

4.3.1 Quellcode Konventionen

Das Ziel dieser Konventionen ist es, einen gut les- und wartbaren Quellcode zu schreiben. Dies wird erleichtert, wenn Bezeichnungen und Formatierungen im Code einem festgelegten Schema folgen. Im Folgenden werden die wichtigsten Regeln angeführt.

- Bezeichner von Klassen, Strukturen und Enumerationen beginnen groß
- Bezeichner von Methoden beginnen klein
- Bezeichner von Variablen beginnen mit einem dem Typ entsprechendem Präfix
- jedes neue Wort in zusammengesetzten Bezeichnern, beginnt groß
- die geschweiften Klammern, die Blöcke umschließen, stehen auf einzelnen Zeilen
- alle Zeilen innerhalb der Klammern sind einen weiteren Tabulator eingerückt

Dieses Schema folgt hauptsächlich der ungarischen Notation. Eine ausführlichere Beschreibung, wie die ungarische Notation benutzt wird und welche Vorteile dies bietet, findet sich in [MSDN1].

4.3.2 Kommentare

Zusätzlich zu den normalen Kommentaren, die in C++ üblich sind, werden in den Quellcodedateien sogenannte „Spezielle Dokumentationsblöcke“ [Dim02] vorhanden sein. Dies sind Kommentare, welche durch das Programm Doxygen aus dem Quellcode extrahiert werden und danach in eine Dokumentation umgewandelt werden. Auf diese Art können Informationen und Inhalte, welche für die Dokumentation des Quellcodes nötig sind, direkt in diesem untergebracht werden. Die exakte Generierung der Dokumentation kann durch einige Kommandos gesteuert werden. In der folgenden Tabelle 4-1 werden die wichtigsten davon kurz beschrieben:

Kommando	Bedeutung
/** //!	Kennzeichnet den Beginn eines speziellen Dokumentationsblocks
\brief	Eine Kurzbeschreibung des folgenden Objektes
\param	Beschreibt einen der Parameter der folgenden Methode/ Funktion
\return	Beschreibt den Rückgabewert der folgenden Methode/ Funktion
\file	Kennzeichnet diesen Dokumentationsblock als Beschreibung einer Quellcodedatei
\todo	Fügt den folgenden Kommentar der „To Do“ Liste hinzu (eine Liste, die noch zu erledigende Aufgaben enthält)

Tabelle 4-1: die wichtigsten Doxygen Kommandos

Eine komplette Liste über alle verfügbaren Kommandos befindet sich in [Dim02]. Da die gesamte Dokumentation im Quellcode enthalten ist, wird garantiert, dass die generierte Dokumentation auf dem gleichen Stand wie der Quellcode ist. Wird einmal vergessen, einen Dokumentationsblock entsprechendend zu erweitern, beispielsweise beim Hinzufügen eines neuen Parameters zu einer Methode, so wird dieses beim nächsten Generieren der Dokumentation gemeldet. Da solche Meldung Dateinamen und Zeilennummer enthalten, kann der Fehler ohne aufwendigen Vergleich der veralteten Dokumentation mit dem neuen Quellcode beseitigt werden.

4.4 Repräsentation der Daten

Wenn ein Programm geschrieben werden soll, das mit Polygonen und Graphen arbeitet, so müssen diese Informationen natürlich auch im Speicher gehalten werden. Dazu muss zunächst eine passende Repräsentationsform gewählt werden. Weiter oben wurden bereits die Bibliotheken LEDA und STL erwähnt. Sie bieten eine breite Auswahl vorgefertigter Typen, welche das gesamte Aufgabengebiet abdecken. Diese Datentypen können auch als gut getestet und daher fehlerfrei und effizient angesehen werden. Es gilt nur noch die zur Lösung der Aufgabe notwendigen Typen entsprechend ihren Eigenschaften zu wählen. Im nächsten Abschnitt soll nun die Wahl der wichtigsten Typen vorgestellt und begründet werden.

4.4.1 Punkte in der Ebene

Wenn in der Informatik Punkte gespeichert werden sollen, so werden diese zumeist in ihre X und Y-Koordinate zerlegt. Diese beiden Zahlen werden dann einfach als Fließkommazahl gespeichert. In vielen Fällen mag dies ausreichen, obwohl es auf diese Art überhaupt nicht möglich ist, jeden beliebigen Punkt darzustellen. Die Genauigkeit einer Fließkommazahl ist beschränkt. Daher erhält man so nur eine endliche Menge von Punkten, welche natürlich nicht ausreichen, um eine Ebene wirklich vollständig zu beschreiben. Dies führt regelmäßig zu Problemen, wenn geometrische Aufgaben bearbeitet werden sollen. Es wird eine Anzahl degenerierter Fälle erzeugt, wenn zum Beispiel nicht mehr genau bestimmt werden kann, wo sich zwei fast parallele Strecken schneiden. Lyle Ramshaw prägte den Begriff der „verzopften Geraden“ [MN99, S. 610ff] für dieses Phänomen.

Um derartige Probleme zu überwinden, gibt es in LEDA den Type rat_point [MN99, S. 583f], welcher für Punkte mit Paaren rationalen Zahlen als Koordinaten steht. Hier werden die Koordinaten eines Punktes als homogene Koordinaten gespeichert. Es gibt dann die drei Zahlen x, y und p, welche den Punkt P(x/p; y/p) beschreiben. Hierbei sind Nenner und Zähler jeweils beliebig große Integer. Dieser Datentyp kann also zur genauen Speicherung von Punkten benutzt werden. Zur Ausgabe ist es wieder nötig, die Informationen in die übliche Darstellung, ein Paar von Fließkommazahlen, umzuwandeln.

4.4.2 Polygone

Da alle Hindernisse als Polygone vorausgesetzt werden, ist dies ein weiterer Datentyp, der in vielen Programmteilen auftaucht. Wiederum findet sich in LEDA ein passender Typ: rat_polygon. Es handelt es sich hier um ein Polygon, beschrieben durch eine Menge von Eckpunkten. Diese werden wiederum als rat_point gespeichert, so dass eine genaue Darstellung eines Polygons in der Ebene erfolgen kann.

Zumeist werden nicht nur einzelne, sondern gleich eine ganze Menge dieser Polygone benötigt. Zu diesem Zweck wird folgender Typ definiert:

```
typedef std::list< leda_rat_polygon* > PolyList;
```

Es handelt sich hier um den Container list aus der STL, welcher spezialisiert wird, um Zeiger auf die LEDA Polygone aufzunehmen. Die Liste unterstützt Einfügen, Entfernen und Iterieren mit einer Komplexität von O(1). Daher wird sie hier dem vector vorgezogen, welcher nur beim selten benötigten wahlfreien Zugriff einen Vorteil böte.

4.4.3 Graphen und Wege

Um Graphen zu repräsentieren, bietet LEDA gleich zwei Datentypen an. Einmal den Typ graph und weiterhin den ugraph. Bei Letzterem handelt es sich um einen ungerichteten Graphen (ugraph = undirected graph). Obwohl hauptsächlich mit solchen Graphen gearbeitet wird, kommt trotzdem der Typ graph, welcher für gerichtete

Graphen steht, zum Einsatz. Es kommen dann die in [MN99, S. 257f] angeführten Methoden zum Einsatz, um einen solchen Graphen als ungerichtet zu betrachten. Die Autoren geben dort weiterhin an: „We also have data type ugraph. We use it very rarely.“ Ein ungerichteter Graph birgt also offensichtlich keinen Vorteil, wenn er im Speicher abgelegt wird. Da die Daten im Computer immer in einer bestimmten Ordnung abgelegt sein müssen, hat selbst ein ungeordnetes Paar von Knotenpunkten stets implizit eine gewisse Ordnung. In LEDA äußert sich das darin, dass einer der Knotenpunkte einer Kante stets source (Quelle) und der andere target (Ziel) genannt wird, egal ob ein Graph nun gerichtet oder ungerichtet ist.

Um Wege in einem Graphen zu beschreiben, kommt eine node_list zum Einsatz. Dies ist eine Liste von Knotenpunkten (englisch „nodes“), mit der ein Weg eigentlich nicht vollständig beschrieben werden kann. Die Kanten zwischen den einzelnen Knoten werden in dieser Liste nicht gespeichert. Da die Kanten aber einen Teilweg beschreiben, von welchem wir wissen, dass er eine Strecke ist, so sind tatsächlich alle Kanten zwischen zwei Knoten äquivalent. Daher ist die Liste der Knotenpunkte vollkommen ausreichend zur Beschreibung des Weges. Im Allgemeinen sollte der Graph auch niemals mehrere Kanten zwischen einem Paar von Knoten besitzen, da dies unnütz wäre. Unter dieser Bedingung lassen sich dann auch die Kanten eindeutig aus der node_list bestimmen.

4.5 Algorithmen und Interfaces

Interfaces definieren einen Satz von abstrakten Methoden und Attributen, die einem bestimmten Zweck dienen. Sie sind abstrakt in dem Sinne, dass hier für die Methoden nur Prototypen definiert werden, sowie eine generelle Aufgabe, für die sie benutzt werden. Es gibt jedoch noch keine Implementierung für diese Methoden.

Dieses Prinzip soll nun auf die Teilaufgaben übertragen werden, zu deren Bewältigung sich verschiedene Algorithmen heranziehen lassen. Das Interface beschreibt die Teilaufgabe, die Eingabedaten sowie die erzeugten Ausgabedaten. Wird ein Algorithmus dann über dieses Interface benutzt, so werden die Daten in den genau vereinbarten Formaten übergeben. Der Benutzer kann dann erwarten, zu einer gegebenen Eingabe stets eine korrekte Lösung zu erhalten, unabhängig vom tatsächlich benutzten Rechenweg.

Es soll allerdings nicht gefordert werden, das man stets exakt dieselbe Lösung erhält. Wie weiter oben schon gezeigt, existieren beispielsweise verschiedene Methoden, um einige unnötige Kanten in einem Graph zu entfernen, ohne dass dies den späteren optimalen Weg beeinflusst. Wäre nun nur die Rückgabe einer ganz bestimmten Kantenmenge erlaubt, ist die Anzahl der möglichen Implementierungen für das Interface unnötig eingeschränkt. Auch die Nutzung von Näherungslösungen wäre so gänzlich ausgeschlossen, da diese ja naturgemäß im Allgemeinen nicht der korrekten Lösung entsprechen.

Da C++ im Gegensatz zu beispielsweise Java, das Schlüsselwort „interface“ nicht kennt, werden hier zumeist sogenannte Interface-Klassen benutzt. Dies sind abstrakte Klassen, welche eine Reihe rein virtueller Methoden haben. In einer von dem Interface abgeleiteten Klasse, kann dieses nun implementiert werden, indem solche virtuellen Methoden überschrieben werden. Da die abgeleitete Klasse in der „is a“-Beziehung zum Interface steht, kann diese Klasse überall, wo ein Pointer oder eine Referenz auf die Interface-Klasse erwartet wird, auch verwendet werden. Die Polymorphie (Vielgestaltigkeit) des Interfaces sorgt dann für die Ausführung der verschiedenen Berechnungsmethoden. Mit Hilfe der späten Bindung wird der konkrete Code der gewünschten Implementierung, das heißt in unserem Fall einer der verschiedenen Algorithmen, zur Laufzeit gewählt.

4.6 GraphBuilder

Ein GraphBuilder ist dafür zuständig, aus den Hindernissen sowie dem Start- und Zielpunkt einen Graphen zu erstellen. Der Graph repräsentiert dann alle Teilwege, aus denen später der Weg konstruiert wird. Die in Kapitel 3.2 vorgestellten Algorithmen sollen also mit dem GraphBuilder Interfaces implementiert werden. Dies geschieht mit den folgenden zwei Methoden.

GraphBuilder

```
virtual bool buildBase(
        std::list< leda_rat_polygon* >* plipplObstacles,
        SRect& rectSize );

virtual bool addStart(
        leda_rat_point pptStart[2],
        graph& gTarget,
        node_array<double>& naX,
        node_array<double>& naY );
```

Die Aufteilung in diese beiden Methoden entspringt den in Kapitel 3.5.1 gezeigten Arbeitsschritten. Zunächst wird mit buildBase aus den Hindernissen sowie der rechteckigen Begrenzung des Weltmodells ein Basisgraph erzeugt. Das Ergebnis wird zunächst nur intern gespeichert und noch nicht ausgegeben. Erst mit addStart, wird ein kompletter Graph erstellt, indem noch der Start- und Zielpunkt hinzugefügt werden. Zusätzlich sind zu allen Knotenpunkten noch X- und Y-Koordinaten angegeben, welche die geometrische Struktur des Graphen repräsentieren. Hierin ist auch die Bewertung des Graphen enthalten, da an Hand dieser Koordinaten der Abstand zweier Knoten bestimmt werden kann.

Die erste Implementierung eines GraphBuilders ist der BuilderBruteforce. Dieser überprüft für alle denkbaren Kombinationen von Eckpunkten, ob die Verbindungslinie eines der Hindernisse schneidet. Dazu wird eine Methode aus der LEDA Bibliothek benutzt, welche die Schnittpunkte zwischen Strecken und Polygonen berechnet. Dieser Ansatz ist durchaus für kleinere Probleme anwendbar. Es muss allerdings festgestellt werden, dass neben dem stark anwachsenden Zeitaufwand bei größeren Polygonmengen ein weiteres Problem besteht. Wie oben erwähnt, werden Schnittpunkte zwischen Strecken und Hindernissen gesucht. Sind zwei solche gefunden, so wird die Strecke als blockiert angesehen und nicht als Teilweg akzeptiert. Im Normalfall bedeutet dies, dass die Strecke tatsächlich durch ein Hindernis führt. Es gibt jedoch eine Möglichkeit, wo dies nicht der Fall ist. Ist die Strecke eine Diagonale in einem der Polygone, so hat sie 2 Schnitte an den beiden verbundenen Eckpunkten. Je nach Form des Hindernisses kann eine Diagonale aber auch vollständig außerhalb liegen, wenn das Polygon eine konkave

Menge ist. Auf ähnliche Weise äußert sich das Problem auch wieder, wenn Start- oder Zielpunkt auf dem Rand eines Polygons liegen. Diese Probleme könnten sicher beseitigt werden, wenn man all diese Spezialfälle durch zusätzliche Abfragen erkennen und umgehen würde. Sinnvoller wäre es jedoch, die Bemühungen in die anderen Algorithmen zu investieren und den BuilderBruteforce nur als Testfall und zum Vergleich der Ausführungszeiten im Programm zu belassen.

Der BuilderRotate hingegen implementiert einen Sweep, wie er in Kapitel 3.2.3 beschrieben wird. Der Algorithmus hält sich weitgehend an die Ausführungen in [BKOS00, S. 310ff]. Ein wichtiger Unterschied besteht nur in der Behandlung der Statusstruktur, welche weiter unten nochmals genauer erklärt wird. Der BuilderRotatePartial arbeitet grundsätzlich genauso, wendet jedoch zusätzlich die Erkenntnisse aus den Kapiteln 3.2 und 3.3 an, wie die Berechnung weiter beschleunigt werden kann. Anstatt den Strahl wirklich um volle 360 Grad zu rotieren, werden hier zunächst die sinnvollen Bereiche für die Rotation bestimmt. Dies geschieht analog zur Abbildung 3-11. Nur Polygonkanten, die für diese beiden Bereiche relevant sind, werden auch bearbeitet. Es werden hier zwei separate Sweeps durchgeführt, welche dann auch jeweils nur die Teilwege im entsprechenden Kreissektor finden. Die Umgebung wird also, wie der Name schon suggerieren soll, nur partiell abgesucht.

4.6.1 Statusstruktur des Sweeps

Einer der am schwierigsten zu realisierenden Teile in BuilderRotate und BuilderRotatePartial besteht in der effizienten Verwaltung der Statusstruktur. In [BKOS00, S. 311] wird vorgeschlagen, diese Daten in einem balancierten Suchbaum zu speichern. Er garantiert, dass die benötigten Operationen innerhalb einer gewissen Komplexität ausgeführt werden können, um die angestrebte Gesamtkomplexität zu erreichen.

Die Notwendigkeit, diese Struktur komplett neu zu implementieren, anstatt beispielweise die Datenstrukturen aus LEDA zu benutzen, ergibt sich aus der Art, wie die Elemente der Struktur geordnet werden. Normalerweise ist für jedes Element ein fester Schlüssel vorhanden, der die Ordnung im Baum definiert. In unserem Fall ist dies der Abstand der Polygonkanten zum Mittelpunkt des Sweeps. Während der Sweep nun voranschreitet, ändern sich aber diese Abstände. Es ist nicht möglich, dass die Kanten die Reihenfolge tauschen, da sie sich dann schneiden müssten. Trotzdem kann kein Schlüssel berechnet werden. Deshalb ist zwingend eine Implementierung nötig, die lediglich mit einer Vergleichsfunktion auskommt.

Im Internet [Sch04][SDS95] findet sich zumeist der RedBlackTree als Möglichkeit der Implementierung eines solchen balancierten Suchbaumes. Der Name lehnt sich an die Knoten in einem solchen Baum an, welche jeweils Rot oder Schwarz gefärbt sind. Betrachtet man die Arbeitsweise eines solchen Baumes, so stellt man fest, dass doch ein beträchtlicher Aufwand betrieben werden muss, um den Baum stets balanciert zu halten. Neben dem zusätzlichen Speicherverbrauch für die Färbung ist auch ein großer Teil des Codes allein für das Umsortieren und Umfärben der Knoten zuständig. Die Tabelle in [MN99, S. 127] bestätigt die Vermutung, dass ein RedBlackTree durch diesen Aufwand in der praktischen Anwendung nicht sehr effizient arbeitet.

[MN99, S. 126f] erwähnt jedoch auch die Skiplists, welche dort zu den effizientesten Implementierungen gehören. Es handelt sich hier um eine Datenstruktur, welche intern mit einem Zufallsgenerator arbeitet, also letztendlich nicht deterministisch ist. Die erwartete Komplexität liegt in dem Bereich des RedBlackTrees, es wären jedoch in einigen konkreten Fällen (deren Erscheinen ja zufällig ist) auch viel schlechtere Laufzeiten denkbar. Diese treten jedoch nur sehr selten auf und werden daher durch die anderen Fälle ausgeglichen. Laut [Pug98] ist die Chance, dass die Suche in einer SkipList mit 1000 Elementen länger als das Fünffache der erwarteten Zeit dauert, ungefähr 1 zu 10^{18}. In der praktischen Anwendung sind Skiplists daher schneller als RedBlackTrees, da hier das komplizierte Ausbalancieren nicht nötig ist. Positiv wirkt sich hier auch aus, dass eine Skiplist allgemein sehr einfach durchsucht werden kann, was zu einem kurzen und schnellen Quellcode führt.

Weiterhin lässt sich hier auch sehr gut das Problem des nicht vorhandenen Schlüssels umgehen. Eine SkipList ist eine verkettete Liste, bei der ein Element nicht nur mit einem, sondern mit mehreren Nachfolgern verkettet ist. Welche und wie viele dies genau sind, wird letztendlich durch die zufällig generierte „Höhe“ eines Elementes bestimmt. Die Elemente werden daher auch als Türme verstanden, wobei jedes Stockwerk einen Nachfolger speichert. Dies ist der jeweils nächste Turm der mindestens diese Höhe hat. Da höhere Türme seltener sind, werden in dieser Höhe mehr Türme „übersprungen“. Daher auch der Name „Skip“list, zur Bezeichnung dieser Vorgehensweise.

In der vorliegenden Implementierung wird nun mit jedem Turm eine Strecke assoziiert. Strecken, die weiter entfernt vom Mittelpunkt des Sweeps sind, werden in der Liste weiter vorn eingeordnet. Bei einem Such- oder Einfügevorgang befindet man sich stets am Ende einer solchen Strecke, die gesucht bzw. eingefügt werden soll. Das Sortierkriterium wird aus der Orientierung dieses Endpunktes der aktuellen Strecke zu den restlichen Strecken gebildet. Der Suchvorgang beginnt nun zunächst mit all den Türmen, welche die maximale Höhe besitzen. Es werden dort die Strecken gesucht, zwischen denen der Punkt liegt. So erhält man eine erste grobe Annäherung, da in dieser Höhe die Türme noch weit voneinander entfernt sind. Nun geht man eine Etage nach unten und die Suche beginnt erneut, jedoch nur zwischen den beiden vorher gefundenen Türmen. Dies wird fortgesetzt, bis man am niedrigsten Level angekommen ist. Dabei wird das Intervall zwischen den Türmen Schritt für Schritt immer kleiner, bis man schließlich die gewünschte Position gefunden hat.

Zu weiteren Details der Skiplists sei nochmals auf [Pug98] sowie [MN99, S. 196ff] und [Goo01] hingewiesen.

4.7 PathFinder

Nachdem der GraphBuilder einen Graphen erstellt hat, kann ein PathFinder auf diesen angewandt werden. Hierin verbergen sich alle Algorithmen, die den kürzesten Weg in einem Graphen bestimmen. Zunächst einmal sind für diese Berechnungen die Daten des bewerteten Graphen nötig.

PathFinder

```
PathFinder( graph& gSrc, EdgeCost& ecSrc );
```

Da stets der Konstruktor ausgeführt werden muss, wenn ein Objekt instanziert wird, so ist dadurch gewährleistet, dass jeder PathFinder mit einem Graphen und einer Instanz einer EdgeCost assoziiert wird. Die EdgeCost wird für die Bewertung des Graphen benötigt (siehe 4.7.1). Weiterhin stehen folgende Methoden zur Verfügung:

```
virtual double findShortestPath(
        node nStart,
        node nTarget,
        node_array<edge>& naePred )= 0;

bool convertPredToList(
        node nStart,
        node nTarget,
        node_array<edge>& naePred,
        node_list& nlPath );
```

In findShortestPath verbirgt sich der eigentliche Algorithmus zur Bestimmung des Weges. Die Methode ist daher rein virtuell und wird für jeden konkreten Algorithmus neu implementiert. Hier wird ein Start- und Zielknotenpunkt übergeben, zwischen welchen der kürzeste Weg gesucht wird. Der Graph und seine Bewertung wird ja schon bei der Konstruktion des Objektes vereinbart und muss daher nicht erneut angegeben werden. Der Rückgabewert ist die Gesamtlänge des Weges, falls ein Weg gefunden wird. In dem Array naePred wird zu jedem Knotenpunkt eine Kante gespeichert. Hier sind während der Suche alle bisher gefundenen Wege abgelegt. Dies geschieht, indem die Kante zum Vorgänger im Weg gespeichert wird. Ist am Ende der Suche ein optimaler Weg gefunden, so steht in diesem Array beim Zielknoten die Kante zu dessen Vorgänger. Bei diesem wiederum ist die nächste Vorgängerkante gespeichert. Dies setzt sich bis zum Startknoten fort.

Der Weg wird also rekonstruiert, indem all diese Vorgänger bestimmt und anschließend in umgekehrter Reihenfolge zu einem Weg zusammengefasst werden. Diese Aufgabe übernimmt die Methode convertPredToList, welche bei allen

Algorithmen in gleicher Art arbeitet. Es werden wiederum die Start- und Endknotenpunkte übergeben und der beschriebene Vorgang durchgeführt. In nlPath wird dann letztendlich eine Liste der Knotenpunkte, die den Weg beschreiben, gespeichert.

4.7.1 EdgeCost

Über das Interface EdgeCost kann die Bewertung eines Graphen festgelegt werden. Dies geschieht unter Verwendung einer Methode, mit der man die Bewertung der Kanten abfragen kann. Eine solche Bewertung kann auch als Kosten, welche bei der Benutzung der Kante entstehen, interpretieren. Daher die Bezeichnung EdgeCost, welche übersetzt Kantenkosten bedeutet. Bei den Kosten muss es sich natürlich nicht grundsätzlich um einen Geldbetrag handeln, es kann auch eine Entfernung oder Zeitspanne gemeint sein.

```
virtual double     getCost( edge& e )= 0;
```

Der Methode wird eine Kante übergeben und man erhält die zugehörigen Kosten als Rückgabewert. Wiederum muss die Methode durch eine konkrete Implementierung überschrieben werden.

Im Rahmen der Diplomarbeit gibt es hier lediglich eine konkrete Klasse: die NodeDistance. Hier werden als Kosten die euklidischen Entfernungen der beiden durch die Kanten verbundenen Knotenpunkte angegeben. Dazu müssen bei Konstruktion der Klasse die Koordinaten aller Knotenpunkte übergeben werden.

Man beachte, dass nicht jede Bewertung grundsätzlich mit jedem PathFinder benutzt werden kann. Beispielsweise setzt der Algorithmus von Dijkstra voraus, dass jede Kante größer als null bewertet wird. Eine Implementierung von EdgeCost, welche negative Kosten liefert, ist jedoch nicht verboten. Um korrekte Ergebnisse mit diesen negativen Bewertungen zu erhalten, ist es dann auch nötig, einen entsprechenden PathFinder zu implementieren.

4.7.2 PFLedaDijkstra und PFDijkstra

Hierbei handelt es sich um zwei Klassen, welche beide mit dem Algorithmus von Dijkstra arbeiten und so das PathFinder Interface implementieren. PFLedaDijkstra arbeitet mit der in der LEDA Bibliothek enthaltenen Version des Algorithmus. Hierzu ist nur die Umwandlung des bewerteten Graphen in die von LEDA geforderten Konventionen nötig. Danach wird die Funktion DIJKSTRA_T aufgerufen, welche die Daten an LEDA weitergibt. Die genaue Vorgehensweise dieses Algorithmus ist in [MN99, S. 333ff] beschrieben. Durch die Benutzung der Funktion wird jedoch der zusätzliche Aufwand erzwungen, um die Daten in der benötigten Form bereitzustellen. Da dies immer den gesamten Graphen betrifft, ist dieser Aufwand von der Größe des Graphen abhängig, aber nicht von der Länge des Weges.

In PFDijkstra kann dies nun umgangen werden, indem diese Implementierung mit genau den Daten, die ein GraphBuilder erzeugt, weiter arbeitet. Dazu wurde der Algorithmus von Dijkstra, so wie in Kapitel 3.4.2 beschrieben, umgesetzt. Der Quellcode wurde auch etwas flexibler gestaltet, so dass verschiedene Arten von Prioritätslisten benutzt werden können. Mit folgender Methode kann diese für den nächsten Suchlauf gewählt werden:

```
void setHeapType( HeapType htPriority );
```

Es kann leicht für Test- oder Vergleichszwecke zwischen den einzelnen Typen gewechselt werden. Wird diese Funktion nicht aufgerufen, d.h. also keine Prioritätsliste explizit festgelegt, so wird automatisch die zur Zeit schnellste Implementierung in der verwendeten LEDA Bibliothek gewählt.

4.7.3 PFAStar

In dieser Klasse findet sich eine Implementierung des A*-Algorithmus. Zu den bisher benötigten Informationen kommt nun noch eine Schätzfunktion für die Wegkosten. Diese wird dem Algorithmus bereits im Konstruktor übergeben, so dass die Funktion stets zur Verfügung steht.

```
PFAStar( graph& gSrc, EdgeCost& ecSrc, CostBound& cbMin );
```

Bei CostBound handelt es sich wiederum um ein Interface, welches eine untere Schranke für die Kosten berechnet. Das Interface enthält zwei Methoden.

```
virtual void        setTargetNode( node& nTarget )= 0;

virtual double      getMinCost( node& nSource )= 0;
```

Zunächst wird mit setTargetNode der Zielknotenpunkt festgelegt. Alle weiteren Berechnungen beziehen sich dann auf dieses Ziel. Die Methode getMinCost gibt dann eine untere Schranke der Kosten zurück, welche von einem konkreten Knotenpunkt bis zum Zielknotenpunkt anfallen.

Die Klasse CBDistance implementiert das Interface CostBound. Als Schätzfunktion wird hier der euklidische Abstand zum Zielpunkt benutzt. Die Zulässigkeit und Konsistenz dieser Funktion wurden bereits in Kapitel 3.4.3 gezeigt. Zur Berechnung der Distanzen werden die Positionen der einzelnen Knotenpunkte des Graphen benötigt. Daher werden dem Konstruktor von CBDistance Referenzen auf die entsprechenden Felder mit den Koordinaten übergeben.

PFAStar kann grundsätzlich mit jeder konsistenten Schätzfunktion arbeiten. Es muss jedoch darauf geachtet werden, dass sowohl die Kantenbewertungen als auch diese Heuristik die gleichen Größen messen. Möchte man beispielsweise einen Weg finden, welcher möglichst kurz sein soll, so sind alle Kanten mit Entfernungen bewertet. Verständlicherweise kann hier nun keine Schätzfunktion verwendet werden, die beispielsweise die minimale Zeit angibt, die man bis zum Ziel benötigt. Auch die Schätzfunktion muss dann Entfernungen messen.

Ansonsten ähnelt die Klasse PFAStar stark der Klasse PFDijkstra, nur dass eben der Algorithmus um die Benutzung der Heuristik erweitert wurde. Auch die Funktion zur Wahl der benutzten Prioritätsliste ist vorhanden.

4.8 GraphData

Die Klasse GraphData macht es möglich, die zuvor besprochenen Algorithmen auf einfache Art zu benutzen. Die aktuell verwendeten Klasseninstanzen und Daten werden intern gespeichert. Wird also in dem Programmteil, der für Nutzereingaben zuständig ist, ein neues Polygon erstellt, so werden die Daten an GraphData weitergeben. Die Klasse liefert anschließend auch die neuen Ergebnisse zur Darstellung zurück. Ihr Design orientiert sich daher in gewissem Maße am Facade Pattern [Cap04]. Die Benutzung der verschiedenen Algorithmen wird durch eine Anzahl Methoden ermöglicht, welche diese auswählen und ihre Ausführung starten. Die Ergebnisse der letzten Berechnungen werden intern gespeichert und können beliebig oft abgerufen werden. Die bereitgestellten Informationen schließen auch einige statistische Daten ein, wie die Zeitdauer bestimmter Berechnungen.

```
void setAlgoGB( AlgoGraphBuilder agbNew );
AlgoGraphBuilder getAlgoGB();
std::string getAlgoGBName( AlgoGraphBuilder agbId );
```

Die erste hier gezeigte Methode setAlgoGB bestimmt den Algorithmus zur Konstruktion des Graphen. Die Festlegung gilt für alle folgenden Berechnungen, bis die Methode erneut aufgerufen wird, um einen anderen GraphBuilder zu wählen. Um die aktuelle Auswahl abzurufen, wird die Methode getAlgoGB benutzt. Diese Darstellung kann intern genutzt werden, für die Ausgabe kann auch noch der Name des Algorithmus ausgelesen werden. Dafür ist getAlgoGBName zuständig, welche zu jedem der Algorithmen einen String liefert.

```
void setAlgoPF( AlgoPathFinder apfNew );
AlgoPathFinder getAlgoPF();
std::string getAlgoPFName( AlgoPathFinder apfId );
```

Diese drei Methoden arbeiten auf die gleiche Weise wie die vorangegangenen. Sie sind jedoch nicht für den GraphBuilder, sondern den PathFinder zuständig. Hier wird also der Algorithmus zur Bestimmung des optimalen Weges festgelegt oder abgefragt.

```
void buildGraph(    leda_rat_point pptStart[2],
                    PolyList* pplObstacles,
                    SRect& rectSize );

void buildPath();
```

Diese zwei Methoden starten nun die Berechnungen mit den gewählten Algorithmen. Mit buildGraph wird der komplette Graph erstellt. Als Parameter wird zunächst ein Array mit Start- und Zielpunkt übergeben. Der zweite Parameter sind die Hindernisse, welche in einer Liste von Polygonen gespeichert sind. Zuletzt gibt das Rechteck die Grenzen des Weltmodells an. In dem so erzeugten Graphen, kann der optimale Weg zwischen den Start- und Zielknotenpunkt gesucht werden. Dies geschieht mit einem Aufruf von buildPath, welches nun keine weiteren Parameter benötigt. Beide Methoden haben keinen Rückgabewert, da die Ergebnisse, wie schon erwähnt, nur intern gespeichert werden.

```
void invalidateGraph();
void invalidateStarts();
```

Diese beiden Methoden bestimmen, welche Daten bei Berechnungen ungültig sind, und welche sich seit dem letzen Aufruf nicht geändert haben. Dabei zeigt invalidateGraph an, dass sich eines der Hindernisse geändert hat und folglich der gesamte Graph neu berechnet werden muss. Ein Aufruf von invalidateStarts bedeutet hingegen, dass nur der Start- oder Zielpunkt verändert wurde.

Erfolgt kein Aufruf dieser Methoden, so werden die an buildGraph übergebenen Daten ignoriert. Die Berechnungen werden so durchgeführt, als würden erneut die gleichen Daten, wie beim letzten Aufruf, übergeben. Es kann dann, wie im Kapitel 3.5 beschrieben, ein Großteil der Rechenzeit gespart werden. Die Benutzeroberfläche „weiß“ genau, wann welche Daten manipuliert werden. Daher ist es effektiver, wenn diese dann der Klasse GraphData eine solche Meldung schickt, anstatt GraphData selbst alle Eingabedaten auf mögliche Änderungen prüfen zu lassen. Dies würde nicht nur einen Vergleich erfordern, sondern auch die komplette Speicherung aller Eingabedaten.

GraphData

```
bool checkData(     leda_rat_point pptStart[2],
                    PolyList& plObstacles,
                    PolyList& plError );
```

Wie an verschiedenen Stellen erwähnt, wird man eventuell falsche Ergebnisse erhalten, sobald man Berechnungen mit unerlaubten Eingabedaten durchführt. Daher überprüft diese Methode die Daten auf Schnitte zwischen den Polygonkanten. Weiterhin wird überprüft, ob Start- und Zielpunkt in einem Hindernis liegen. Wird kein Problem gefunden, so wird „wahr“ zurückgegeben. Gibt es jedoch Fehler, so wird in plError eine Liste der betroffenen Polygone gespeichert. Diese Polygone enthalten entweder einen der genannten Punkte oder schneiden sich gegenseitig. Mit Hilfe der Liste kann dem Benutzer dann visualisiert werden, welche Eingabedaten fehlerhaft sind.

```
graph& getGraph();
node_list& getPath();
node_array<double>& getX();
node_array<double>& getY();
```

Nach jeder Berechnung werden intern die Ergebnisse abgelegt. Zu deren Ausgabe können sie über diese vier Methoden abgerufen werden. Die Abfragen können beliebig oft geschehen, ohne dass die Rechnungen erneut ausgeführt werden müssen. Mit getGraph erhält man den Graphen, welcher die Struktur der Teilwege repräsentiert. getPath liefert hingegen eine Liste der Knotenpunkte dieses Graphen, welche den optimalen Weg bilden. Bei dem Graphen muss es sich nicht um den gesamten Sichtbarkeitsgraphen handeln. Beispielsweise liefert der BuilderRotatePartial einige der Kanten nicht zurück, wenn diese ohnehin nicht im optimalen Weg auftauchen können.

Für eine graphische Ausgabe wird noch die Anordnung des Graphen in der Ebene benötigt. Hier finden die Methoden getX und getY Anwendung. Diese liefern zu jedem Knotenpunkt die X- und Y-Koordinate eines korrespondierenden Punktes. Bei der Darstellung eines Graphen müssen seine Kanten nicht unbedingt als Strecken dargestellt werden. In diesem Fall steht aber jede Kante für einen geradlinigen Teilweg. Daher können alle Kanten als Linie visualisiert werden. Eine solche Linie wird zwischen den beiden Punkten gezeichnet, welche zu den beiden Knotenpunkten gehören, die durch die Kante verbunden sind. Da die Teilwege im Allgemeinen in beide Richtung passiert werden können, kann der Richtungssinn der Kanten (es wird ja intern mit einem gerichteten Graphen gearbeitet) vernachlässigt werden. Pfeile, auf niedrig aufgelösten Monitorbildern ohnehin schlecht erkennbar, müssen also nicht gezeichnet werden.

```
int getNrOfNodes();
int getNrOfEdges();
```

```
double getPathLength();
int getNrOfPathNodes();
```

Mit diesen Funktionen lassen sich nun noch eine Reihe Informationen über die Ergebnisse der Rechnungen abrufen. Die beiden Methoden getNrOfNodes und getNrOfEdges beziehen sich auf den erzeugten Graphen. Es lassen sich die Anzahl der enthaltenen Knotenpunkte (Nodes) und der Kanten (Edges) ausgeben. Durch getNrOfPathNodes wird die Zahl der Knotenpunkte im Weg, inklusive Start und Ziel, zurückgegeben. Die Anzahl der Teilwege oder Kanten im Weg erhält man sofort, wenn man von der Zahl der Knotenpunkte noch Eins subtrahiert. Die Bewertung des Weges, das heißt die Summe der Bewertungen der einzelnen Teilwege, liefert die Methode getPathLength.

```
int getTimeBaseGraph();
int getTimeSZGraph();
int getTimeFindPath();
```

Mit diesen Methoden erlangt man Zugriff auf die Ergebnisse der Zeitmessung der einzelnen Arbeitsschritte. Für jeden Schritt wird einzeln die benötigte Rechenzeit in Millisekunden gemessen. Bei getTimeBaseGraph handelt es sich um die Erstellung des Basisgraphen, also ohne den Start- und Zielknotenpunkt. Die Methode getTimeSZGraph gibt die benötigte Zeit zurück, um Start und Ziel dem Graphen hinzuzufügen. Schließlich gibt getTimeFindPath an, welche Zeit für die Bestimmung des optimalen Weges benötigt wird.

Allerdings sind diese Ergebnisse nicht immer vergleichbar. Zunächst einmal hängt die Ausführungszeit stark von der verwendeten Hardware ab, speziell der Taktung der CPU und dem Speicherdurchsatz. Wenn die Software in einer Multitaskingumgebung läuft, steht ihr auch nicht die gesamte Rechenleistung zur Verfügung. Es werden stets noch einige andere Programme abgearbeitet. Dies können einerseits systemkritische Aufgaben, wie das Taskscheduling, sein oder andere Anwendungen, beispielweise Virenscanner, die stets im Hintergrund arbeiten. Aus diesem Grund ergeben sich selbst auf dem gleichem PC Unterschiede. Um repräsentative Ergebnisse zu erhalten, sollten die Messungen also auf der gleichen Hardware vorgenommen werden. Dabei dürfen, soweit möglich, keine anderen Anwendungen arbeiten.

5 Numerische Experimente

Im dritten Kapitel dieser Diplomarbeit wurde bereits die theoretische Komplexität der benutzten Algorithmen diskutiert. Dabei wurden Aussagen über die Laufzeiten getroffen, wenn die Algorithmen besonders „schwierige“ Problemfälle lösen müssen. Solche Fälle werden auch „worst case“ genannt. Ihre Betrachtung ist immer dann wichtig, wenn ein Algorithmus nach einer bestimmten Zeit terminieren muss. Kann diese Zeit im schlechtesten Fall eingehalten werden, so auch in allen anderen Fällen.

Während also derartige Betrachtungen durchaus ihre Berechtigung haben, interessiert in der Praxis auch der „typische“ bzw. „durchschnittliche“ Anwendungsfall. Leider ist die theoretische Betrachtung des „average case“ sehr schwierig. Während man den schlechtesten Fall zumeist eindeutig modellieren kann, ist dies für einen typischen Fall nicht möglich. Hier müsste die Komplexität aller Problemfälle an Hand der Wahrscheinlichkeiten ihres Auftretens gewichtet werden.

In diesem Kapitel soll es daher darum gehen, einige praktisch gemessene Bearbeitungszeiten vorzustellen. Die Messungen sollen analysiert und ihr Zusammenhang mit den theoretisch erwarteten Werten diskutiert werden. Alle Messungen wurden auf einem Pentium 4 mit einer Taktrate von 2400 MHz vorgenommen.

5.1 Szenarien

Für die Messungen der Laufzeiten werden drei verschiedene Szenarien benutzt, jedes betont andere Aspekte der Algorithmen. Zudem ist jedes dieser Szenarien so angelegt, dass die Größe einfach variiert werden kann. Dies ist nötig, da bei zu kleinen Problemen die Messergebnisse zu stark verfälscht sind. Die benutzten Funktionen zur Zeitmessung runden auf jeweils volle Millisekunden, so ist stets ein Fehler von bis zu einer halben Millisekunde möglich. Daher sind bei großen Zeiten die relativen Fehler geringer. Es werden jeweils drei Größen betrachtet. Hierbei ist die zweite Version doppelt so groß wie die erste, der dritte Fall des Szenarios viermal so groß. Es ergeben sich insgesamt neun Probleminstanzen.

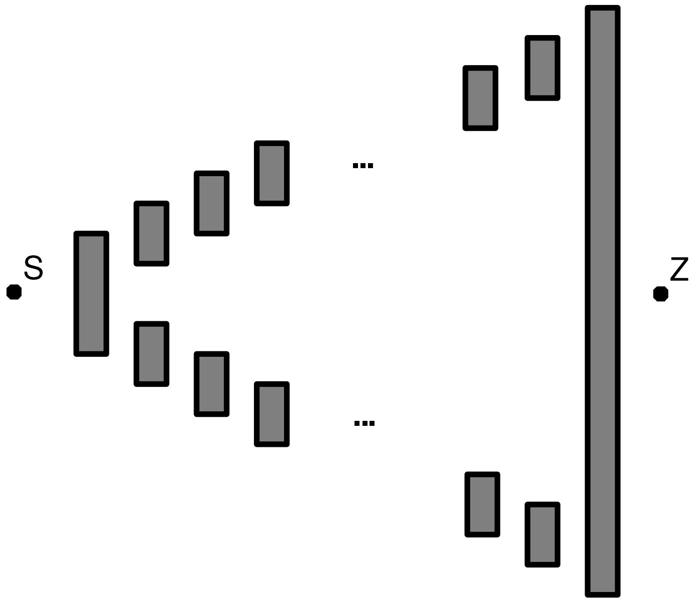

Abbildung 5-1: Szenario 1

Dieses erste Szenario besteht hauptsächlich aus einer Reihe von schräg versetzen Rechtecken. Dabei sind diese Rechtecke genau um die Hälfte der Höhe versetzt. Die Reihe wird nochmals an der X-Achse gespiegelt und darunter angeordnet. Zwei weitere Rechtecke schließen diese Reihen links und rechts ab. Es handelt sich hier um einen Fall, der durch den Algorithmus A* sehr schlecht bearbeitet werden kann. Zunächst wird immer versucht, möglichst direkte Wege zu benutzen, welche jedoch durch das letzte, lange Rechteck blockiert sind. Der korrekte Weg „außen herum“ wird erst ganz am Ende untersucht.

Die Größe kann variiert werden, indem die Anzahl der versetzten Rechtecke verändert wird. In der Abbildung ist dies durch die Punkte „...“ angedeutet. Die kleinste Ausführung des Szenarios beinhaltet insgesamt 100 Rechtecke, d.h. jeweils 49 oben und unten, eines ganz links und eines ganz rechts. Die anderen beiden bestehen demzufolge aus 200 und 400 Rechtecken.

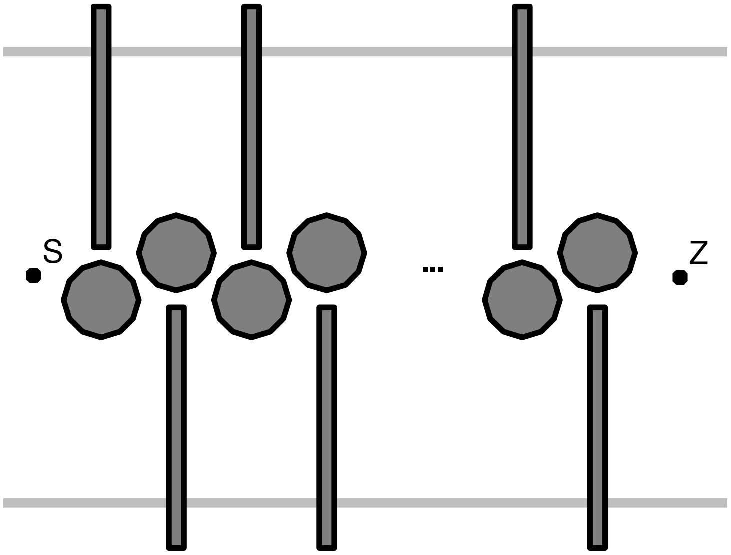

Abbildung 5-2: Szenario 2

Das zweite Szenario besteht nicht nur aus Rechtecken, sondern auch aus regelmäßigen Sechzehnecken. Die Sechzehnecke sind jeweils gegeneinander versetzt, ein Weg kann sich nur „hindurchschlängeln“. Die Größe der Rechtecke ist so gewählt, dass sie sich über die Grenzen des Weltmodells ausdehnen. So wird erreicht, das wirklich nur dieser eine Weg optimal sein kann, da unter- bzw. oberhalb der Rechtecke kein gültiger Weg möglich ist. Durch die großen Innenwinkel in den regelmäßigen Vielecken sollte der BuilderRotatePartial hier eine große Anzahl Kanten einsparen können.

In der einfachsten Version des Problems werden 32 Sechzehnecke und 32 Rechtecke benötigt, also insgesamt 64 Hindernisse. Diese Zahl wird dann jeweils auf 128 und 256 verdoppelt, um die anderen zwei Ausprägungen des Szenarios zu erstellen.

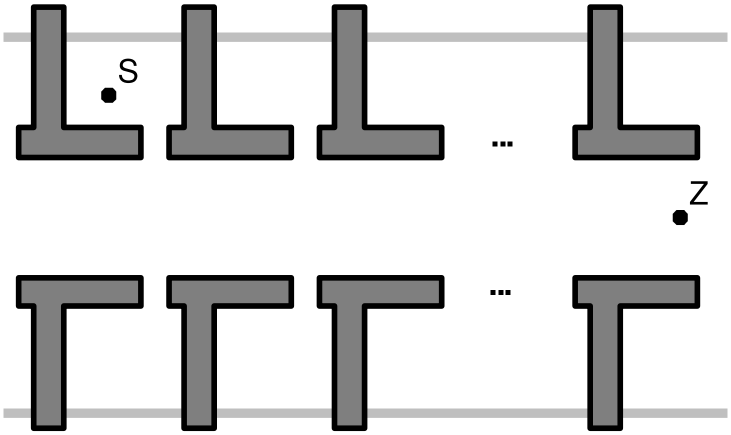

Abbildung 5-3: Szenario 3

Das letzte Szenario ist an einen Korridor und daran angrenzende Zimmer angelehnt. Zwei verbundene Rechtecke stellen jeweils die Mauern zwischen den Zimmern und dem Korridor dar. Die Außenwand des Gebäudes wird durch die rechteckige Begrenzung der Welt markiert. Der Startpunkt liegt jeweils in dem ersten Zimmer und das Ziel am Ende des Korridors. Wieder besitzen die drei Varianten des Szenarios eine unterschiedliche Zahl Polygone und dementsprechend auch mehr „Zimmer“. Die Polygonzahlen belaufen sich in den drei Fällen auf 64, 128 und 256 Stück.

5.2 GraphBuilder

5.2.1 Erzeugung des Basisgraphen

Zunächst soll das Verhalten der einzelnen GraphBuilder bei einer steigenden Zahl von Hindernissen untersucht werden. Danach kommen wir auf das Verhältnis der GraphBuilder untereinander zu sprechen. In der folgenden Darstellung sind die Zeiten angegeben, die der BuilderBruteForce zur Erstellung des Basisgraphen benötigt.

Szenario 1		Szenario 2		Szenario 3	
Hindernisse	**Zeit in ms**	**Hindernisse**	**Zeit in ms**	**Hindernisse**	**Zeit in ms**
100	35490	64	72784	64	49625
200	284328	128	546716	128	390046
400	2334779	256	4283633	256	3109087

Tabelle 5-1: BuilderBruteForce, Erstellen des Basisgraphen

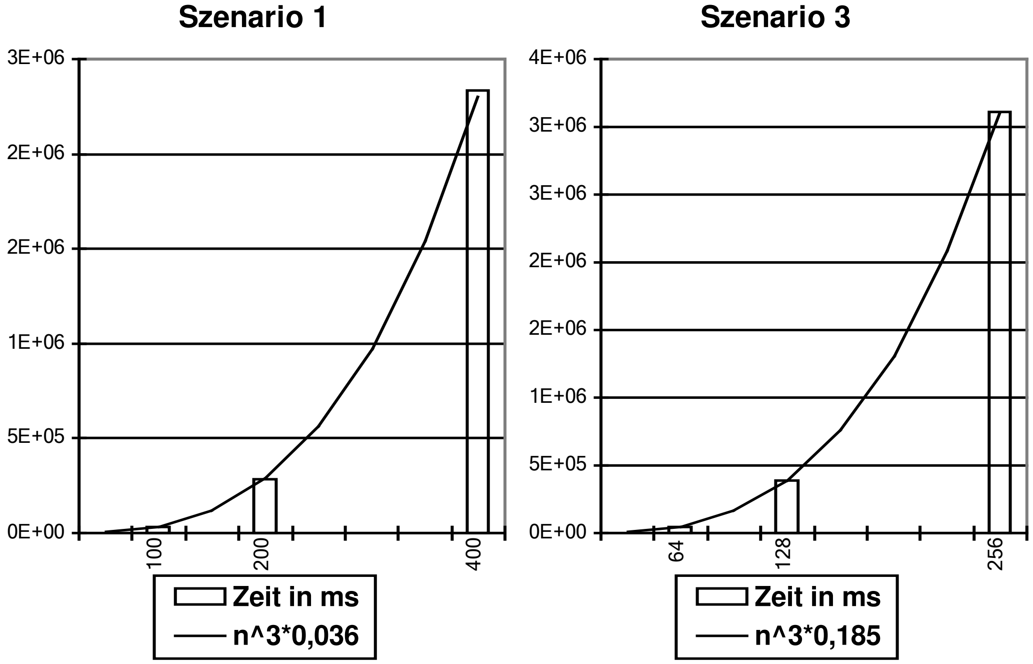

Diagramm 5-1 : BuilderBruteForce, Erstellen des Basisgraphen

In den Diagrammen wurde jeweils noch eine Funktion eingezeichnet, das n befindet sich auf der Abszisse und steht dort für die Anzahl der Hindernisse. Diese Funktionen sind jeweils kubisch. Es ist zu erkennen, dass sie eine gute Näherung für die tatsächlich gemessenen Zeiten bilden. Somit wird die erwartete Komplexität von O(n³) bestätigt. Da die kubischen Funktionen recht schnell wachsen, werden hier Rechenzeiten von über einer Stunde erreicht.

Szenario 1		Szenario 2		Szenario 3	
Hindernisse	Zeit in ms	Hindernisse	Zeit in ms	Hindernisse	Zeit in ms
100	688	64	2874	64	866
200	2840	128	12572	128	3798
400	11692	256	54320	256	15722

Tabelle 5-2: BuilderRotate, Erstellen des Basisgraphen

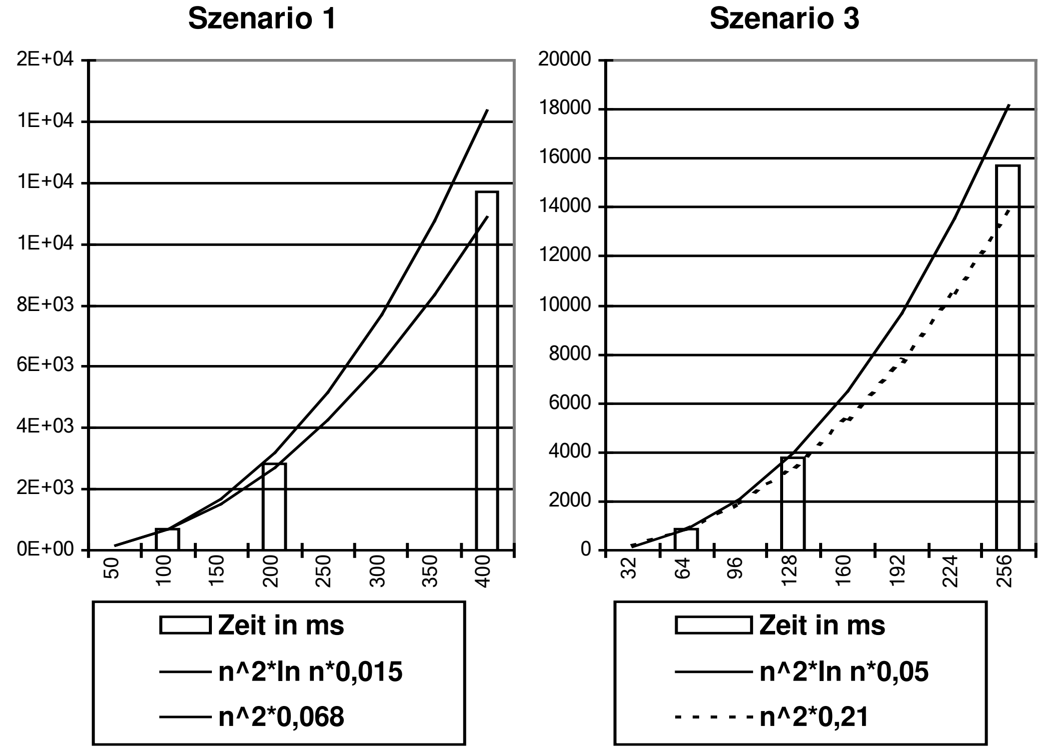

Diagramm 5-2: BuilderRotate, Erstellen des Basisgraphen

Beim BuilderRotate zeigt sich in Tabelle 5-2 und Diagramm 5-2, dass die gemessene Laufzeit in der Praxis nicht ganz so stark steigt, wie es der worst case vermuten lässt. Aber auch im typischen Anwendungsfall, liegt die Zeit noch deutlich über der quadratischen Funktion. Bei größer werdender Hinderniszahl wird der Vorteil gegenüber der kubischen Komplexität jedoch sehr schnell deutlich. Im größten Szenario 2 wird statt einer Stunde nur noch knapp eine Minute gerechnet, man vergleiche dazu Tabelle 5-1 und 5-2.

Szenario 1		Szenario 2		Szenario 3	
Hindernisse	**Zeit in ms**	**Hindernisse**	**Zeit in ms**	**Hindernisse**	**Zeit in ms**
100	305	64	537	64	237
200	1289	128	2275	128	1032
400	5476	256	9563	256	4485

Tabelle 5-3: BuilderRotatePartial, Erstellen des Basisgraphen

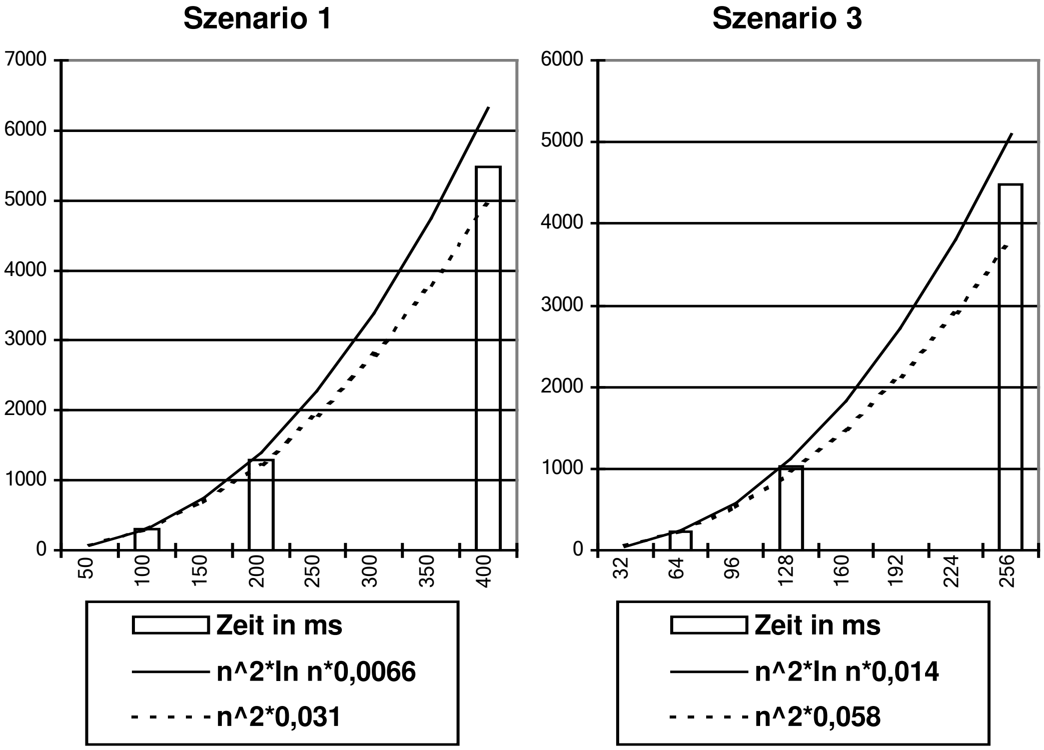

Diagramm 5-3: BuilderRotatePartial, Erstellen des Basisgraphen

Beim BuilderRotatePartial sehen wir in Tabelle 5-3 und Diagramm 5-3 annähernd das gleiche Bild wie zuvor bei dem BuilderRotate. Der Unterschied ist hier, dass die Konstanten der zusätzlich eingezeichneten Funktionen durchweg kleiner sind. Die Laufzeit des BuilderRotatePartial liegt also stets in der gleichen Größenordnung, ist aber generell um einen bestimmten Faktor geringer.

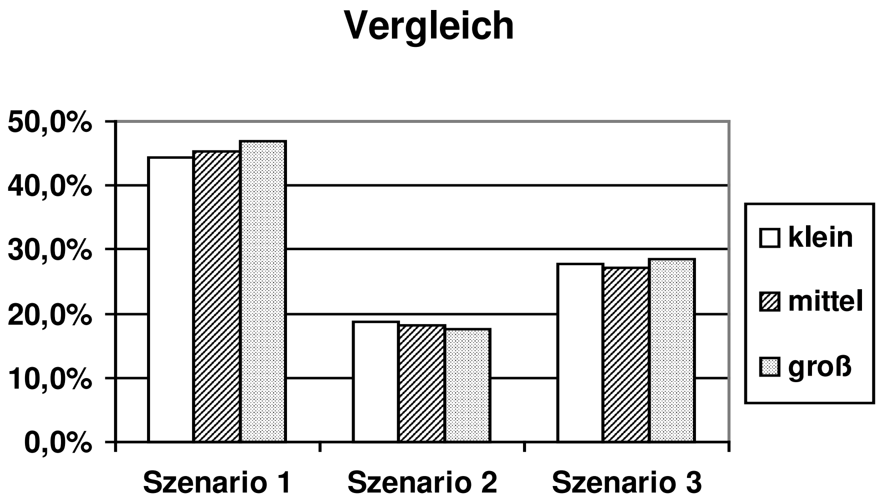

Diagramm 5-4: Basisgraph, BuilderRotate und BuilderRotatePartial

Vergleicht man alle drei GraphBuilder, so fällt zunächst einmal der große Abstand zwischen dem Bruteforce-Algorithmus und den beiden Sweep-Algorithmen auf. Der enorme Vorteil, welcher mit der Hinderniszahl weiter wächst, wurde ja bereits in der theoretischen Betrachtung der Komplexität begründet. Dies wird also eindrucksvoll durch die Messergebnisse untermauert.

Nicht ganz klar war jedoch zu Beginn der Untersuchung, wie groß der Unterschied zwischen den beiden Sweep-Algorithmen sein würde. Im Diagramm 5-4 wurden diese beiden Algorithmen ins Verhältnis gesetzt. Die Zeitspanne, welche BuilderRotate benötigt, entspricht dort 100 Prozent. In den Messungen ergibt sich, dass der BuilderRotatePartial nur etwa 15%-50% der Zeit benötigt, die BuilderRotate an der gleichen Aufgabe rechnet. Wie in Diagramm 5-4 ersichtlich, ist die genaue Prozentzahl nicht von der Größe des Problems abhängig. Es ist eher ein Zusammenhang mit dem durchschnittlichen Innenwinkel der bearbeitenden Hindernisse zu erkennen. Im Szenario 1, welches die kleinsten Innenwinkel aufweist, findet die geringste Zeitersparnis statt. Im Szenario 2, wo die Innenwinkel der Sechzehnecke jeweils nur knapp unter 180 Grad liegen, ist die Rechenzeit hingegen am kürzesten.

5.2.2 Kanten im Graphen

Der erstellte Graph ist nur ein Zwischenergebnis, da ausgehend von diesem Graphen noch der optimale Weg bestimmt werden soll. Im Kapitel 3.3.3 wurde gezeigt, dass verschiedene Graphen letztendlich zum selben Weg führen. Daher ist für einen GraphBuilder jeder dieser Graphen als Lösung erlaubt. Trotzdem ist es natürlich erstrebenswert, dass ein Graph möglichst wenige der unnötigen Informationen enthält. Dies spart Speicherplatz und Zeit bei der weiteren Verarbeitung.

Folgende Tabelle liefert einen Überblick der Kantenzahlen, welche die verschiedenen Algorithmen erzeugen.

		BuilderBruteForce Kantenzahl	**BuilderRotate Kantenzahl**	**BuilderRotatePartial Kantenzahl**
Szenario 1	**klein**	19450	15237	6995
	mittel	76427	58646	25520
	groß	303534	229665	97411
Szenario 2	**klein**	2522	2529	757
	mittel	5082	5101	1492
	groß	10202	10241	2961
Szenario 3	**klein**	5015	5176	2057
	mittel	18262	18621	6216
	groß	69332	70421	20790

Tabelle 5-4: Kantenzahlen in den erzeugten Graphen

Zu beachten ist hier allerdings, dass bei der Anwendung, der in der LEDA Bibliothek implementierten Version des Algorithmus von Dijkstra, die Zahl der Kanten jeweils verdoppelt wird. Da dort die Kanten gerichtet betrachtet werden, muss für beide Richtungen eines Teilweges eine Kante eingefügt werden.

Wie stark die Anzahl der Kanten steigt, hängt hauptsächlich von der Anordnung der Hindernisse ab, weniger von den Algorithmen selbst. Im ersten und letzten Szenario ist es so, dass selbst die Hindernisse ganz links noch direkt mit den Hindernissen am rechten Ende der Welt verbunden werden können. Im zweiten Szenario ist das allerdings nicht möglich. Jedes Hindernis hat hier eine begrenzte Anzahl direkter Nachbarn unabhängig von der Anzahl aller Hindernisse. So erklärt sich, dass der Anstieg einmal quadratischen und einmal nur linearen Charakter hat.

Die Ergebnisse von BuilderBruteForce und BuilderRotate ähneln sich stark. Ein Unterschied sind hier zunächst einmal die Teilwege, die Diagonalen in einem Hindernis darstellen. Diese werden vom BuilderBruteForce ja nicht korrekt erkannt. Allerdings ist dies nicht der Grund für die Diskrepanz im Szenario 1. Hier handelt es sich um den Sonderfall, wenn Teilwege Hindernisse berühren.

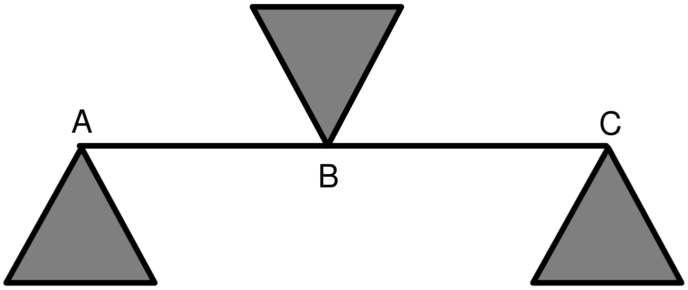

Abbildung 5-4: Teilweg schneidet einzelnen Punkt

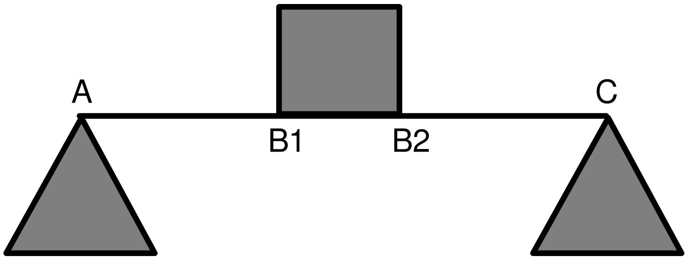

Abbildung 5-5: Teilweg verläuft auf Hinderniskante

Die beiden obigen Abbildungen 5-4 und 5-5 zeigen noch einmal die beiden grundsätzlichen Möglichkeiten. Kurze Strecken könnten zu einer längeren zusammengefügt werden, welche jedoch redundant ist. Natürlich können noch beliebig viele weitere Polygonecken auf der Geraden durch A und C liegen. BuilderBruteForce akzeptiert solche Teilwege, wenn sie nur jeweils einen Punkt des Hindernisses berühren. Das entspräche hier der oberen Abbildung. BuilderRotate akzeptiert grundsätzlich alle diese Teilwege jedoch nur, wenn sie in einer gewissen Reihenfolge abgearbeitet werden. Da A, B und C auf einer Geraden liegen, wird diesen Punkten bei einem Sweep um A oder C der jeweils gleiche Winkel zugeordnet. Daher steht nicht fest, welcher Punkt zuerst bearbeitet wird. Dies wird durch die anfängliche Reihenfolge und das Sortierverfahren bestimmt. Letztendlich wird AC nur akzeptiert, wenn alle Hindernisse, die einen Punkt wie B beinhalten, entweder komplett vor oder komplett nach AC abgearbeitet werden.

Im Szenario 1 kommen nun aber außergewöhnlich viele Teilwege vor, welche sich schräg an die Rechtecke anschmiegen und so jeweils genau eine Ecke berühren. Der BuilderBruteForce akzeptiert alle diese Wege, der BuilderRotate jedoch nur einige wenige. Im Szenario 3 berühren diese Teilwege jeweils Kanten von Hindernisse und werden so vom BuilderBruteForce nie akzeptiert. Daher kommt BuilderRotate hier auf eine geringfügig höhere Kantenzahl, da tatsächlich nur ein Bruchteil dieser unnötiger Kanten erzeugt wird. Beide Algorithmen sind also im Bezug auf die Anzahl erzeugter Kanten nahezu gleichwertig, nur ersterer ist in einigen besonderen Problemfällen benachteiligt.

Interessant gestaltet sich auch der Blick auf das Ergebnis des BuilderRotatePartial. Dieser erzeugt durchweg die wenigsten Kanten, und braucht dafür sogar weniger Rechenzeit. Die Kanten werden nicht durch zusätzliche Berechnungen verworfen, sondern durch solche, die gar nicht erst ausgeführt werden. Es wird also nicht für jede Kante einzeln eine Bedingung überprüft. Das Programm nutzt aus, das die Kanten ohnehin nach dem Winkel geordnet vorliegen. Es übergeht dann die Abschnitte, in denen keine Kanten erzeugt werden müssen.

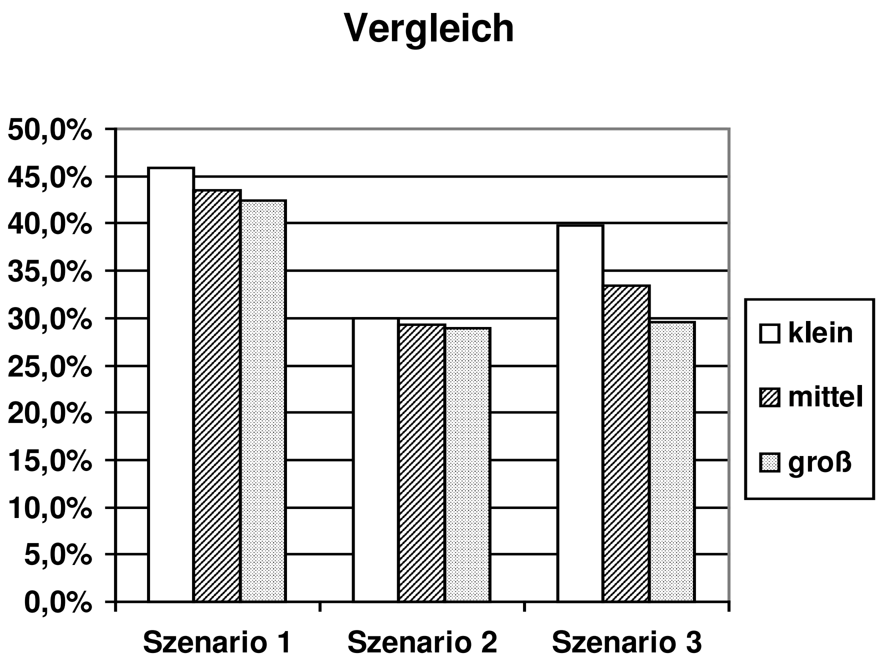

Diagramm 5-5: Kantenzahlen, BuilderRotate und BuilderRotatePartial

Wie das Diagramm 5-5 zeigt, werden beim BuilderRotatePartial zwischen 50 und 70 Prozent der Kanten im Vergleich zu BuilderRotatePartial gespart, dessen Kantenzahlen entsprechen hier 100 Prozent. Wiederum ist ein Zusammenhang zwischen durchschnittlicher Innenwinkelgröße und der Einsparung der Kanten zu erkennen. Zudem scheint der Anteil der verworfenen Kanten immer weiter anzusteigen, wenn mehr Hindernisse bearbeitet werden. Diesbezüglich veranschaulichen die Abbildungen 5-6 und 5-7 den Vergleich des kompletten Sichtbarkeitsgraphen, wie in BuilderRotate erstellt zu dem Ergebnis, welches BuilderRotatePartial erzeugt.

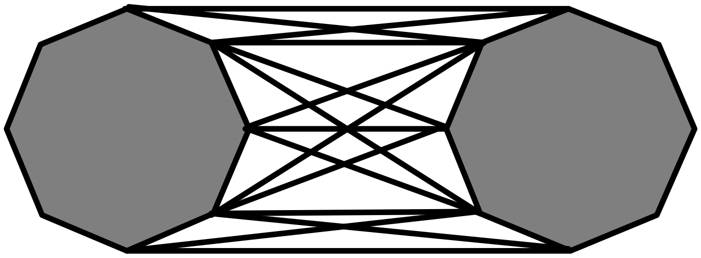

Abbildung 5-6: Sichtbarkeitsgraph

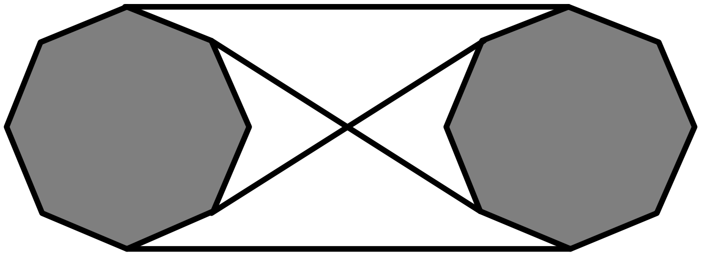

Abbildung 5-7: durch BuilderRotatePartial akzeptierte Kanten

Wie man sieht, werden eine ganze Zahl Kanten zwischen den beiden Achtecken ignoriert. In diesem Fall werden 11 von 15 Kanten aussortiert und nur 4 verbleiben. Zu diesen Kanten zwischen den Achtecken, kommen aber noch die Kanten, die das Achteck selbst beschreiben. Das sind hier natürlich zwei mal acht Stück. Von diesen Kanten wird keine einzige eingespart. Es gibt also 16 Kanten, von denen keine entfernt wird, und nur 15 von denen ein Teil entfernt wird. So ist eine Obergrenze für die Einsparung, 15 von 31 Kanten, gegeben, was in etwa 50% entspricht. Fügt man der Szene nun weitere Hindernisse hinzu, so verändert sich dieses Verhältnis. Je nach Anordnung könnten hier Kanten zwischen jedem Paar von Polygonen möglich sein. Bei zehn Polygonen gibt es beispielsweise schon 45 ($= 10 \cdot 9 / 1 \cdot 2$) Kombinationen aus 2 Hindernissen. Bei zehn Achtecken könnte man so bereits $45 \cdot 15$ Kanten finden, dies sind insgesamt 675 Kanten. Diese stehen nun den nur 80 Polygonkanten gegenüber. Damit ist die Menge, aus der Kanten entfernt werden können, schon mehr als acht mal größer. Der Anteil der entfernten Kanten wächst also, weil der Anteil der möglichen Kandidaten größer wird.

GraphBuilder

		BuilderBruteForce Zeit in ms	BuilderRotate Zeit in ms	BuilderRotatePartial Zeit in ms
Szenario 1	klein	4	5	2
	mittel	18	14	6
	groß	80	56	24
Szenario 2	klein	1	1	1
	mittel	3	3	2
	groß	5	5	4
Szenario 3	klein	2	2	1
	mittel	5	5	2
	groß	19	19	7

Tabelle 5-5: Dauer der Wegsuche bei verschiedenen Graphen

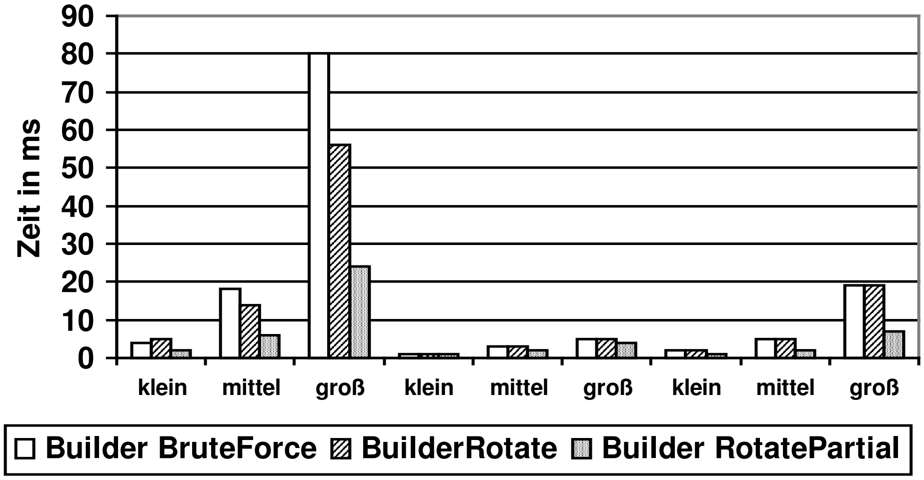

Diagramm 5-6: Dauer der Wegsuche bei verschiedenen Graphen

Die verschiedenen Kantenzahlen der Graphen haben natürlich auch eine Auswirkung auf die Dauer der Wegsuche in diesen Graphen. In dieser Tabelle 5-5 und dem obigen Diagramm 5-6 sind die gemessenen Zeiten des Algorithmus von Dijkstra aufgelistet. Absolut gesehen, sind diese Unterschiede sehr klein. Da das Erstellen des Graphen leicht einige Minute dauern kann, stellen die Zeiten im Bereich einiger Millisekunden nur einen sehr geringen Bruchteil der Gesamtzeit dar. Diese Überlegung trifft allerdings nur zu, wenn auch immer beide Berechnungen ausgeführt werden. Wie schon in Kapitel 3.5 gezeigt, ist es ein Ziel, diese Schritte getrennt auszuführen, weil eventuell Wege mehrmals im gleichen Graphen gesucht werden. Daher ist auch diese Beschleunigung eines einzelnen Schrittes um etwa 50 Prozent nicht uninteressant.

5.3 PathFinder

Wie oben gezeigt wurde, dauert die Suche nach dem optimalen Weg nur eine relativ geringe Zeitspanne im Vergleich zur Erstellung des Graphen. Trotzdem sollen hier nun auch noch die verschiedenen Implementierungen des PathFinder untersucht werden. Zunächst wird der Algorithmus von Dijkstra betrachtet. Die Messungen fanden jeweils in dem Graphen, welcher durch den BuilderRotate erzeugt wurde, statt. Die Ergebnisse sind in der folgenden Tabelle 5-6 aufgelistet.

Szenario 1		Szenario 2		Szenario 3	
Knoten	**Zeit in ms**	**Knoten**	**Zeit in ms**	**Knoten**	**Zeit in ms**
402	5	578	1	386	2
802	14	1154	3	770	5
1602	56	2306	5	1538	19

Tabelle 5-6: Wegsuche mit Algorithmus von Dijkstra

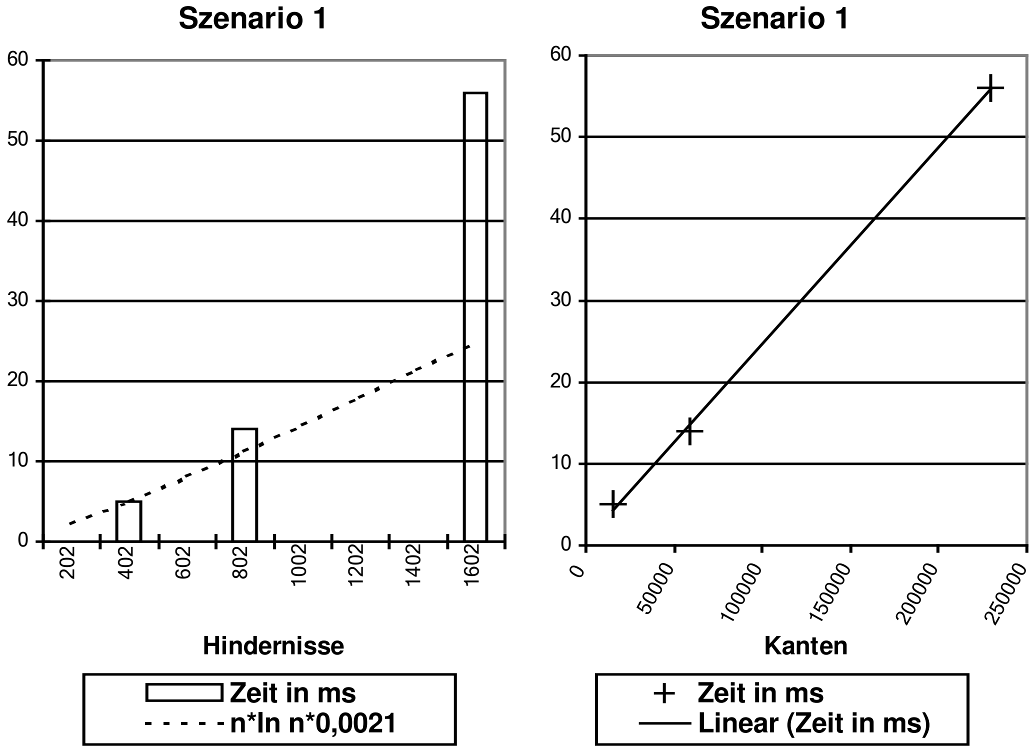

Diagramm 5-7: Wegsuche mit Algorithmus von Dijkstra

Die Komplexität des Algorithmus von Dijkstra wurde im dritten Kapitel mit O(m + n log n) angegeben. Dies bedeutet: die Laufzeit ist sowohl von der Anzahl der Knotenpunkte (n) als auch der Kanten (m) abhängig. Das Diagramm 5-7 links zeigt, dass im ersten Szenario der Einfluss der Kantenzahl auf PFDijkstra sehr groß ist. Wäre der Einfluss der Kantenzahl weniger stark, so würden die Zeiten sich eher an dem gestrichelt eingezeichneten Graphen orientieren. Die Balken liegen immer weiter über der eingezeichneten Funktion. In diesem Szenario wächst die Zahl der Kanten quadratisch zur Hinderniszahl und damit auch zur Anzahl der Knotenpunkte. Daher wird aus O(m+n log n) ein O(n²+n log n)=O(n²). Der Einfluss der Kantenzahl ist also, zumindest in diesem Szenario, so groß, dass ein linearer Zusammenhang zwischen ihr und der Laufzeit entsteht (siehe dazu Diagramm 5-7, rechts).

Szenario 1		Szenario 2		Szenario 3	
Knoten	**Zeit in ms**	**Knoten**	**Zeit in ms**	**Knoten**	**Zeit in ms**
402	32	578	6	386	12
802	132	1154	11	770	49
1602	578	2306	23	1538	223

Tabelle 5-7: Wegsuche mit PFLedaDijkstra

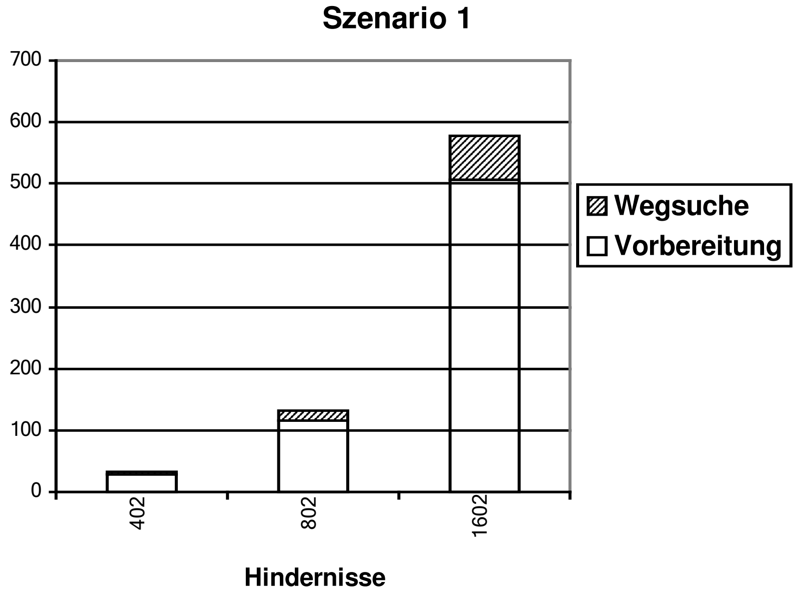

Diagramm 5-8: Wegsuche mit PFLedaDijkstra

Die Implementierung PFLedaDijkstra sollte eigentlich ähnliche Laufzeiten wie PFDijkstra erreichen können, da hier derselbe Algorithmus verwendet wird. Allerdings verlangt die Implementierung in der LEDA Bibliothek, dass alle Richtungen und alle Bewertungen schon explizit gespeichert sind. Bevor der Algorithmus also gestartet wird, muss zunächst ein Graph erzeugt werden, in dem gerichtete Kanten für beide möglichen Durchlaufrichtungen eines Teilweges existieren. Dann müssen für jede dieser Kanten die Kosten berechnet und in einer großen Tabelle gespeichert werden.

Dies benötigt ein Vielfaches an Zeit der eigentlichen Berechnung. Insgesamt wird die Fünf- bis Zehnfache Zeitdauer gemessen, je nach Verhältnis der Kantenzahl zur Anzahl der Knotenpunkte.

Szenario 1		Szenario 2		Szenario 3	
Knoten	**Zeit in ms**	**Knoten**	**Zeit in ms**	**Knoten**	**Zeit in ms**
402	6	578	1	386	0
802	19	1154	3	770	0
1602	73	2306	5	1538	0

Tabelle 5-8: Wegsuche mit Algorithmus A*

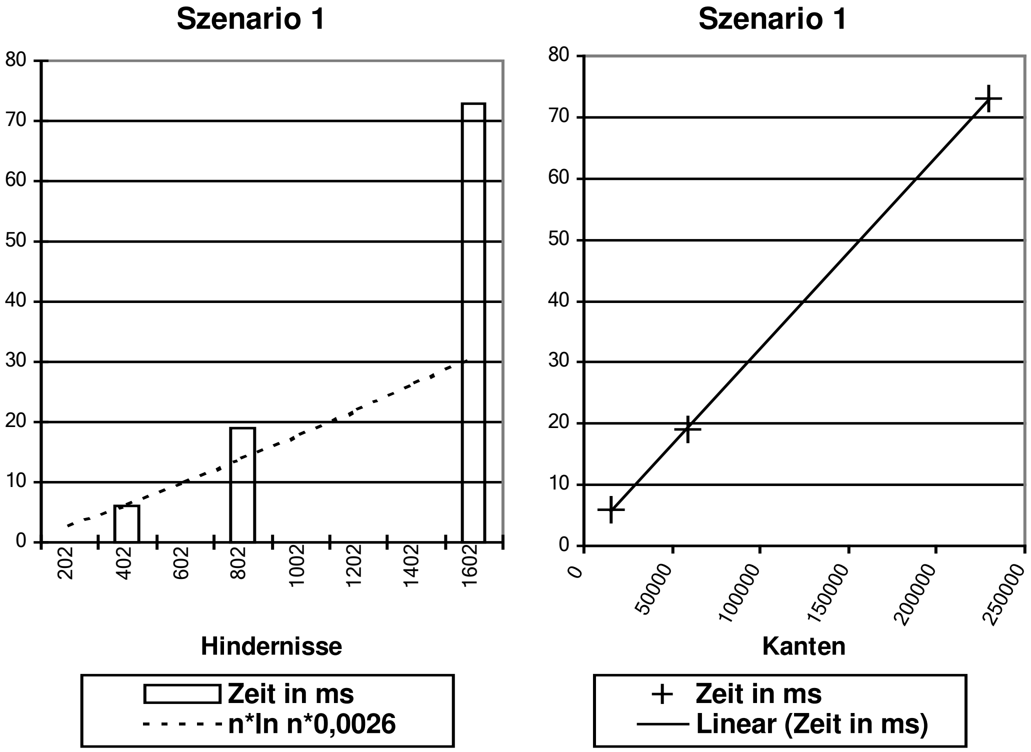

Diagramm 5-9: Wegsuche mit Algorithmus A*

In den ersten beiden Szenarien ist die Laufzeit des A*-Algorithmus mit dem Algorithmus von Dijkstra vergleichbar. Im ersten ist der Aufwand zur Wegsuche proportional zur Anzahl der Kanten. Dies liegt aber hauptsächlich daran, dass Szenario 1 so aufgebaut ist, dass der A*-Algorithmus damit sehr schlecht zurechtkommt. Bei der

Suche nach dem Weg werden immer solche Kanten bevorzugt, deren Endpunkt möglichst nahe am Ziel liegt. Im Szenario 1 geht der Algorithmus dadurch aber „irre“, denn auch wenn man schon nahe beim Ziel ist, muss immer noch das letzte Hindernis, das große Rechteck ganz rechts (siehe Abbildung 5-1), umgangen werden. Dadurch entsteht wieder ein Umweg, so dass diese Wege dann doch verworfen werden. Damit ist hier die Zeit verschwendet, die zur Berechnung der Schätzfunktion nötig ist. Sie gibt letztendlich keinen Hinweis auf den optimalen Weg im Szenario 1. Dieser Fall ist so konstruiert, dass die Schätzfunktion quasi immer falsch entscheidet. In der Praxis sollte ein solcher Fall allerdings nicht allzu oft auftreten. Wenn die Heuristik zumindest auf einem kurzem Stück des Weges eine Einsparung ermöglicht, sollte der Aufwand ihres Einsatzes zumindest ausgeglichen werden.

Ebenso unwahrscheinlich ist es, ständig einen Fall wie in Szenario 3 anzutreffen. Hier wird das Ziel sehr schnell gefunden. Und zwar so schnell, dass die Zeitmessung die Zeit nicht registriert und auf Null abrundet. Sobald die Suche das „Zimmer“ verlassen hat, gibt es eine direkte Verbindung zum Ziel. Diese wird dann auch sofort gewählt. Es wird keines der anderen Zimmer untersucht und daher eine kurze Rechenzeit erzielt.

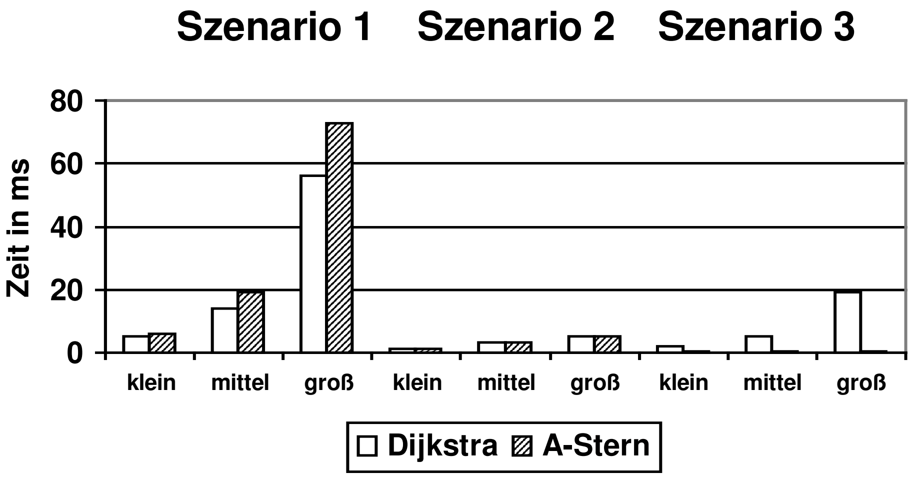

Diagramm 5-10: Algorithmus von Dijkstra und Algorithmus A*

In dem obigen Diagramm 5-10 werden die Ergebnisse von PFDijkstra und PFAStar gegenüberstellt. Die Ergebnisse des A*-Algorithmus liegen im Szenario 1 etwas über denen des Algorithmus von Dijkstra. Wie beschrieben handelt es sich aber dort um eine eher ungewöhnliche Situation, in welcher der Einsatz der Heuristik keinen Vorteil bringt. Im zweiten Szenario ist das Verhältnis ausgeglichen, das heißt beide Algorithmen benötigen in etwa die gleiche Rechenzeit. Im letzten Szenario funktioniert die Schätzfunktion dann sehr gut und A* benötigt wesentlich weniger Zeit.

Es gibt also in drei verschiedenen Szenarien drei unterschiedliche Ergebnisse. So kann es keine eindeutige Empfehlung geben, wie die Wahl zwischen dem Algorithmus von Dijkstra und dem A*-Algorithmus ausfallen soll. Der erstgenannte Algorithmus arbeitet immer so, dass er vor dem Erreichen des Ziels schon alle Knotenpunkte im Graph besucht hat, zu denen einen Weg mit einer geringeren Bewertung als zum Ziel existiert. Dafür dauert die Untersuchung der einzelnen Knoten nicht so lange wie beim A*-Algorithmus, da keine Schätzfunktion oder Ähnliches ausgewertet wird. Hier wird wiederum durch den Einsatz einer solchen Heuristik die Reihenfolge, in der die Knotenpunkte untersucht werden, abgeändert. Es werden zunächst solche Knotenpunkte untersucht, die möglichst nah an der direkten Strecke zum Ziel liegen. In vielen Fällen führt dies dazu, dass beim A*-Algorithmus weitaus weniger Knotenpunkte untersucht werden. Dann fällt die Rechenzeit, welche von der Heuristik benötigt wird, nicht mehr ins Gewicht. Als Faustregel kann man sagen, dass der A*-Algorithmus immer dann sehr wenige Knotenpunkte untersuchen muss, wenn der optimale Weg „nahe" an der Strecke zwischen Start-Ziel liegt.

6 Ausblick

Im Rahmen der Diplomarbeit wurde ein Programm entwickelt, in dem der Benutzer einen Hindernisparcours vorgibt. Dieser besteht aus Polygonen sowie einem Start- und Zielpunkt. Verschiedene im Programm implementierte Algorithmen können dann aus diesen Daten einen optimalen Weg bestimmen. Bei der Realisierung des Programms wurde Wert darauf gelegt, dass einzelne Teile des Programms leicht austausch- und erweiterbar sind. Es soll aufgezeigt werden, welche grundsätzlichen Möglichkeiten bestehen, das Einsatzgebiet des Programms zu erweitern.

Der erwähnte modulare Aufbau des Programms ermöglicht, durch Erweiterung des vorhandenen Quelltexts, Probleme zu bearbeiten, welche nicht im Fokus dieser Arbeit liegen. Dazu können entweder neue Teile in das bestehende Programm eingefügt oder einzelne Module in einem anderen Projekt wiederverwendet werden. Um diesen Vorgang zu erleichtern, ist der Quellcode durchgängig nach den in Kapitel 4.3 aufgestellten Regeln strukturiert.

6.1 Implementierung von Interfaces

Um das Programm zu erweitern, ist es am einfachsten, eines der definierten Interfaces neu zu implementieren. Dieses neue Modul verhält sich dann nach außen hin genau so wie vorher definiert. Daher muss im restlichen Programm kein Code modifiziert werden, der auf das Modul zugreift. Einzig die Stelle, an der das Modul gewählt wird, muss noch verändert werden.

6.1.1 EdgeCost und CostBound

Um eine Metrik festzulegen, mit der die Entfernung zweier Knoten gemessen wird, wird das Interface EdgeCost neu implementiert. Zu jeder Metrik, die auch mit dem A*-Algorithmus benutzt werden soll, wird zusätzlich eine konsistente Schätzfunktion benötigt. Diese muss in einer von CostBound abgeleiteten Klasse codiert werden. Bereits durch diese relative kleine Änderung können verschiedene neue Aspekte betrachtet werden.

Möchte man beispielsweise den Weg mit den wenigsten Kanten finden, so reicht es aus, als Entfernung jeweils die Zahl 1 anzugeben. Man kann sich aber auch davon lösen, Entfernungen zu benutzen und gibt stattdessen die Zeit für einen Teilweg an. So ergibt sich die Möglichkeit, dass der Roboter keine gleichförmige Bewegung ausführt, sondern auch beschleunigt und abbremst. Eine andere Idee wäre, dass der Roboter sich nicht auf allen Wegen gleich schnell bewegen kann. Er könnte dann eine Asphaltstraße einem Schotterweg vorziehen. Sogar die Bearbeitung von dreidimensionalem Gelände wäre so möglich. Für Teilwege, die aufwärts führen, würde man hier die Zeit, je nach Steigung, entsprechend verlängern. Der optimale Weg würde dann eher weniger steile Stücke enthalten, um diesen Zeitverlust zu minimieren.

Allerdings kann nicht für alle Metriken garantiert werden, dass wirklich der optimale Weg gefunden wird. Es wird lediglich der beste Weg gefunden, der sich aus den Teilwegen zusammensetzen lässt. Aber es sind auch noch Wege denkbar die „zwischen“ diesen Teilwegen verlaufen. Für den euklidischen Abstand lässt sich zwar beweisen, dass alle diese Wege länger sein müssen. Im allgemeinen Fall gilt dies allerdings nicht. Hier kann nur garantiert werden, dass ein Weg gefunden wird, sofern ein solcher existiert. Ob dieser optimal ist oder wie viel er vom optimalen Weg durchschnittlich und im Höchstfall abweicht, muss erst noch untersucht werden.

6.1.2 GraphBuilder

Die Suche nach dem optimalen Weg benötigt also bei einer anderen Metrik eine ganz andere Teilmenge der möglichen Bewegungen als Teilwege. Daher ist es auch möglich, einen neuen GraphBuilder zu implementieren, welcher diese Wege wählt. Im Allgemeinen ist es allerdings außerordentlich schwer, eine endliche Anzahl Teilwege zu bestimmen, aus denen der optimale Weg, im Hinblick auf eine bestimmte Kostenfunktion, konstruiert werden kann. Denkt man noch mal an das genannte

Beispiel der Steigungen an einem Berg: schon allein die Zahl aller Wege, die waagerecht verlaufen, ist unendlich. Und auf jedem einzelnen dieser Wege können wieder unendlich viele waagerechte Teilwege gefunden werden. Da Start und Ziel selten auf einer Höhe liegen, werden dann auch noch alle Wege benötigt, auf denen man Höhenunterschiede möglichst optimal bewältigen kann. In solchen Fällen wird es nötig sein, Näherungslösungen zu finden, um dem optimalen Weg zumindest nahe zu kommen. Hier muss man natürlich fragen, ob nicht schon durch die Projektion in die Ebene, eine genügend genaue Lösung erzielt wird.

Statt eines GraphBuilder, der andere Teilwege sucht, könnten natürlich auch die vorhandenen noch verbessert werden. Einerseits könnte versucht werden, die aktuell benötigte Rechenzeit zu senken oder noch mehr unnötige Kanten aus dem Graph auszuschließen. Man könnte jedoch auch eine Näherungslösung anstreben, welche einfacher zu berechnen ist als ein kompletter Sichtbarkeitsgraph. Benötigt man beispielsweise ein sehr detailliertes Wegenetz eines ganzen Gebäudes, so könnte man zunächst alle Zimmer einzeln berechnen. Die entstehenden Teilgraphen würden dann an den Türen zusammengefügt, woraus sich dann der gesamte Graph ergibt. Alle Wege innerhalb eines Zimmers wären so noch optimal. Nur wenn ein Weg durch eine Tür führt, so weicht er dort vom optimalen Weg ab. Der Vorteil wäre jedoch, dass zu jedem Knotenpunkt nicht mehr alle möglichen Verbindungen auf alle Hindernisse überprüft werden müssen. Es ist ausreichend, die Verbindungen in einem Raum mit den Hindernissen innerhalb dieses Zimmers zu testen.

6.1.3 PathFinder

Zum gegenwärtigen Zeitpunkt existieren PathFinder, welche den Algorithmus von Dijkstra und den A*-Algorithmus implementieren. Beide arbeiten mit positiven Kantenbewertungen. Eine sinnvolle Erweiterung für das Programm könnte sein, dass auch negative Bewertungen bearbeitet werden können. Dazu wäre beispielsweise dann der Algorithmus von Moore und Ford geeignet. Er arbeitet korrekt, solange keine geschlossenen Kantenzüge (Kreise) mit negativer Gesamtbewertung im Graph auftreten.

Auch eine Verbesserung oder Erweiterung der beiden vorhandenen Algorithmen ist durchaus denkbar. In [MN99, S. 333ff] wird ein zweiseitiger Suchalgorithmus vorgestellt, bei dem der Algorithmus von Dijkstra gleichzeitig am Start- und am Zielpunkt eingesetzt wird. Es wäre interessant, diesen mit der vorhandenen Klasse PFDijkstra zu vergleichen. Auch eine neue Implementierung des A*-Algorithmus wäre möglich, damit man auch Schätzfunktionen, welche zwar zulässig aber nicht konsistent sind, benutzen kann.

6.2 Weiterverwendung in der Praxis

In dem vorliegenden Programm können die Ergebnisse der Berechnungen nur graphisch dargestellt werden. Ein späteres Ziel sollte jedoch sein, einen Roboter mit diesen Ausgaben zu steuern. Grundsätzlich ist es dazu nötig, das Ausgabemodul zu ersetzen oder zumindest zu erweitern. Im Moment werden die berechneten Daten in Form einiger Linien und Punkte ausgegeben. Diese müssten stattdessen in Bewegungsbefehle für einen Roboter umgewandelt werden.

Ein berechneter Weg besteht aus einer Menge von „Wegpunkten“. Ein Roboter müsste sich jeweils auf gerader Linie von einem Wegpunkt zum nächsten bewegen. Danach sollte er sich auf den übernächsten Punkt ausrichten und dann mit diesem fortfahren und so weiter. Ein Roboter kann seine Befehle nur mit begrenzter Genauigkeit ausführen. Daher muss er immer wieder überprüfen, ob er noch auf dem berechneten Weg ist. Dazu könnte er beispielsweise fortlaufend den Abstand zum aktuellen Teilweg messen und bei einem zu großen Fehler eine Korrektur durchführen. Auch wäre es möglich, dass in regelmäßigen Abständen der Weg komplett neu berechnet wird, beispielsweise wenn eine bestimmte Entfernung zurückgelegt ist. Somit wird vermieden, das sich die Fehler aufsummieren. Für diese möglichen Abweichungen vom Weg sollte beim Erstellen des Weltmodells bereits ein Sicherheitsabstand eingeplant werden. Ebenso ist dies nötig, wenn sich der Roboter nicht auf der Stelle drehen kann.

Weiterverwendung in der Praxis

Weiterhin sollte bedacht werden, dass der Roboter auf ein unerwartetes Hindernis treffen könnte. Dies wäre natürlich auszuschließen, indem man den Roboter in einem abgeschlossenen Raum arbeiten lässt, den niemand sonst betreten darf. Das würde die Einsatzmöglichkeiten eines Roboters aber stark einschränken und ist daher nicht immer praktikabel. Es können verschiedene Strategien entwickelt werden, wie der Roboter sich dann verhalten soll. Er könnte einfach warten, bis das Hindernis verschwindet, wenn es sich beispielsweise um einen Menschen oder einen anderen Roboter handelt. Treffen sich allerdings mehrere Roboter, so muss eine übergeordnete Instanz entscheiden, wer wartet und wer ausweicht. Ist ein Ausweichen nötig, könnte versucht werden, sich am Hindernis entlang zu tasten. Dies wird fortgeführt, bis man wieder zurück auf dem korrekten Weg ist. Auch wäre es möglich, die entsprechenden Kanten aus dem Graphen zu entfernen und die Berechnung des optimalen Weges dann erneut durchzuführen.

Ein Roboter, welcher derartige neue Hindernisse, eventuell erst nachdem sie mehrmals auftraten, in sein Weltmodell einfügt, wäre folglich sogar lernfähig. Es wäre ihm möglich, sich an eine veränderte Umwelt anzupassen. Nachdem ein unerwartetes Hindernis in seine interne Karte aufgenommen wurde, wird beim nächsten Mal ein optimaler Weg geplant, der das Hindernis von vorn herein umgeht. Führt man dies konsequent weiter, so sollte der Roboter ein spezielles Suchprogramm haben. Dieses könnte immer dann aktiviert werden, wenn der Roboter keinen anderen Arbeitsauftrag bearbeiten muss. Begibt er sich dann in diesen Modus, so geht er nach einem bestimmten Schema das gesamte Arbeitsgebiet ab und erneuert sein Weltmodell. Das bedeutet, es wird überprüft, ob es noch alle Polygone gibt, die darin verzeichnet, sind oder ob irgendwo ein neues Hindernis verzeichnet werden muss. Ein solcher Roboter könnte sogar ohne jegliche Information starten und sich selbst ein Weltmodell aufbauen, indem er zunächst das gesamte Gebiet nach Hindernissen absucht.

7 Literatur

[BGZ96] Hans-Joachim Bungartz, Michael Griebel, Christoph Zenger, Einführung in die Computergraphik, Vieweg, 1996

[BKOS00] M. de Berg, M. van Kreeveld, M. Overmars, O. Schwarzkopf, Computational Geometry: algorithms and applications, Springer, 2000

[Cap04] Peter Cappello, Facade Pattern, http://www.cs.ucsb.edu/~cappello/50/lectures/patterns/Facade.html (06.08.2004), 2004

[CH94] John Clark, Derek Allan Holton, Graphentheorie, Spektrum Akademischer Verlag, 1994

[Dij59] E. W. Dijkstra, A note on two problems in connection with graphs, Numerische Mathematik 1, 1959

[Dim02] Dimitri van Heesch, Doxygen Manual, http://www.doxygen.org/ (21.11.2003), 2002

[Ede87] Herbert Edelsbrunner, Algorithms in Combinatorial Geometry, Springer, 1987

[Fel04] Stefan Felsner, Geometric Graphs and Arrangements, Vieweg, 2004

[Fre91] Fremdwörterbuch, Der kleine Duden, 3. Auflage, Bibliographisches Institut und F.A. Brockhaus AG, 1991

[GKHK65] W. Gellert, H. Küstner, M. Hellwich, H. Kästner, Kleine Enzyklopädie: Mathematik, Bibliographisches Institut Leipzig, 1965

[Goo01] Michael T. Goodrich, Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing, http://www.ics.uci.edu/~goodrich/pubs/discex2001.pdf (19.07.2004), 2001

[HB94] Doald Hearn, M. Pauline Baker, Computer Graphics Second Edition, Prentice Hall, 1994

[HK00] Horst W. Hamacher; Kathrin Klamroth, Lineare Netzwerk-Optimierung, Vieweg, 2000

[HNR68] P. E. Hart, N. J. Nilsson, B. Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Trans. on Systems Science and Cybernetics 2, 100-107, 1968

[Jun99] Dieter Jungnickel, Graphs, Networks and algorithms, Springer, 1999

[Kle97] Rolf Klein, Algorithmische Geometrie, Addison Wesley, 1997

[Koz91] Dexter C. Kozen, The design analysis of algorithms, Springer, 1991

[Las96] Michael J. Laszlo, Computational geometry and computer graphics in C++, Pearson Education, 1996

[MN99] Kurt Mehlhorn, Stefan Näher, LEDA, A Platform for Combinatorial and Geometric Computing, Cambridge University Press, 1999

[MSDN1] Hungarian Notation, http://msdn.microsoft.com/library/en-us/dnvsgen/html/HungaNotat.asp (05.07.2004)

[MSDN2] General MFC Topics , http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/html/_core_mfc.3a_.overview.asp (05.07.2004)

[Nil82] N.J. Nilsson, Principles of Artificial Intelligence, Springer, 1982

[Oro94] Joseph O'Rourke, Computational Geometry in C, Cambridge University Press, 1994/1998

[Pug98] William Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees, http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/niemann/s_skip.pdf (19.07.2004), 1998

[Rit02] Ritter, http://vsis-www.informatik.uni-hamburg.de/teaching/ws-02.03/p3/Folien/P3_02_Teil1_6.pdf (25.03.2004), 2002

[Sah99] Sartaj Sahni, Data Structures, Algorithms, & Applications in Java - Pairing Heaps, http://www.cise.ufl.edu/~sahni/dsaaj/enrich/c13/pairing.htm (25.03.2004), 1999

[Sch03] Alexander Schrijver, Combinatorial Optimization Volume A, Springer, 2003

[Sch04] Uwe Schmidt, http://www.fh-wedel.de/~si/vorlesungen/c/beispiele/redblacktree/ (18.07.2004), 2004

[SDK96] Alfred Schmitt, Oliver Deussen, Marion Kreeb, Einführung in graphisch-geometrische Algorithmen, Teubner, 1996

[SDS95] San Diego State University, http://www.eli.sdsu.edu/courses/fall95/cs660/notes/RedBlackTree/RedBlack.html (18.07.2004), 1995

[Stö99] Horst Stöcker, Taschenbuch mathematischer Formeln und moderner Verfahren, 4. Auflage, Verlag Harri Deutsch, 1999

[Tit03] Peter Tittmann, Graphentheorie, Fachbuchverlag Leipzig, 2003

[Tur96] Volker Turau, Algorithmische Graphentheorie, Addison-Wesley, 1996

[You96] Michael J. Young, Introduction to graphics programming for Windows 95: vector graphics using C++, Academic Press, 1996

8 Abbildungsverzeichnis

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

____________________	____________________
Ort, Datum	Christian Rösch

Ich erkläre mein Einverständnis zu einer Veröffentlichung der vorliegenden Arbeit im Internet.

☐ Ja ☐ Nein

Thesen zur Diplomarbeit

Thema: Mathematische Modellierung und Optimierung zur Bestimmung kollisionsfreier Wege für Roboter

von Christian Rösch
Fachhochschule Merseburg
Fachbereich Informatik und Angewandte Naturwissenschaften
Studiengang: Informatik
2004

These I: Ziel dieser Arbeit ist die Konzeption und Realisierung eines Programms, welches kollisionsfreie Wege für Roboter in einer zweidimensionalen Modellwelt plant.

These II: Verschiedene Algorithmen zur Bestimmung des Sichtbarkeitsgraphen sowie Suche des optimalen Weges von einem Start- zu einem Zielknoten in diesem Graphen wurden untersucht, implementiert und verglichen.

These III: Der naive Ansatz, alle Teilstrecken auf Kollision mit allen Hindernissen zu testen, erweist sich in der Praxis bei größeren Szenarien als zu langsam. Bewährt hat sich ein Ansatz unter Verwendung des Sweep-Paradigmas.

These IV: Bei komplexen Weltmodellen, das heißt zahlreichen Polygonen mit vielen Kanten, wird sowohl bei Erstellung des Graphen als auch bei der Bestimmung des optimalen Weges eine große Zeitersparnis erreicht, wenn unnötige Kanten ausgeschlossen werden.

These V: Durch den objektorientierten Designansatz können die Ergebnisse dieser Arbeit gut erweitert und wiederverwendet werden.