Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik I

Alexander Tiderko

Implementierung und Evaluation der Explorationsstrategie PolyExplore auf einem mobilen Robotersystem

6. November 2006

Diplomarbeit

Betreuer: Prof. Dr. Rolf Klein

Erklärung

Mit der Abgabe der Diplomarbeit versichere ich, gemäß §19 Absatz 7 der DPO vom 15. August 1998, dass ich die Arbeit selbstständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

6. November 2006

Alexander Tiderko

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Geometrische Probleme durch geeignete Algorithmen zu lösen ist das Gebiet der „Algorithmischen Geometrie“, die sich mit den Problemen aus vielen Zweigen, wie geografischer Informationssysteme oder Transportnetzwerken befaßt. Einer davon beschäftigt sich mit der Wege- bzw. Bewegungsplanung mobiler autonomer Robotersysteme. Ist die Umgebung dem Roboter bekannt, sprechen wir von Offlinestrategien. Eine Strategie, die mit unvollständiger Information arbeiten muss, wird auch als Onlinestrategie bezeichnet. Dabei gewinnt der Roboter während der Lösung eines Problems immer mehr Informationen, anhand derer er seine Bewegung anpassen oder sogar revidieren muss. Für ein Gütemaß einer Onlinestrategie werden die entstandenen Kosten mit den Kosten der Offlineberechnung verglichen. In diesem Zusammenhang spricht man von der Kompetitivität [22] der Onlinestrategie.

Definition 1
Eine Onlinestrategie S ist *kompetitiv* mit dem Faktor C, wenn es eine Zahl A gibt, so dass gilt:

$$K_s(P) \;\leq\; C \;\cdot\; K_{opt}(P) \;+\; A$$

Dabei ist P eine Instanz eines Problems Π, K_s die Kosten der Strategie S und K_{opt} die Kosten der optimalen Lösung. Die Zahl C wird „kompetitiver Faktor“ genannt und ist ≥ 1. Die reellen Werte A und C dürfen dabei nur von Π und S abhängen, nicht aber von P. Es besteht also ein linearer Zusammenhang zwischen den Kosten der gewählten und der optimalen Strategie. Viele Beispiele für die Onlinestrategien wie Cashing, Ski-Rental, Paging, selbstorganisierende Datenstrukturen etc. sind in dem Buch von A. Fiat und G. Woeginger [10] zu finden.

Aufgaben in der Bewegungsplanung lassen sich in die Bereiche Lokalisation, Suche, Navigation und Exploration untergliedern.

Die Lokalisation versucht die aktuelle unbekannte Position des Roboters anhand der lokalen Sichtinformationen in der schon bekannten Karte zu bestimmen. Oft wird die Lokalisation mit der Erstellung der Karte kombiniert. Solche Algorithmen werden als SLAM (Self Localization And Mapping) bezeichnet. Beispiele sind in [16, 6] zu finden.

Bei der Suche versucht der Roboter ein unbekanntes Ziel in einer unbekannten Umgebung zu finden. Beispiele für diesen Bereich die Suche nach einem unbekanntem Ziel in einem Labyrinth von Shannon [32], die Suche nach einer Tür in einer Wand [4] oder das Pledge-Algorithmus zum Entkommen aus einem Labyrinth [1].

Im Gegensatz zur Suche ist das Ziel bei der Navigation bekannt. Der Roboter versucht in einer unbekannten Umgebung dieses Ziel anzusteuern. Beispiele sind die Bug-Algorithmen von Lumelsky und Stepanov [25].

Diese Diplomarbeit befasst sich mit der Exploration von Polygonen. Ziel dabei ist es, einen Weg zu finden, von dem aus ein Roboter mit unbeschränkter Sicht jeden Punkt im Inneren des Polygons mindestens einmal sieht, d.h. das ganze Polygon erkundet. Für einfache Polygone wird dieses Problem durch die Strategie „PolyExplore“ von Hoffmann, Icking, Klein und Kriegel [19] gelöst, die in dieser Arbeit untersucht wird. PolyExplore erreicht einen kompetitiven Faktor von 26.5, d.h. der Weg der Strategie in einer unbekannten Umgebung ist höchstens 26.5 mal länger als die *Shortest Watchman Route* — der optimale Weg, der berechnet werden kann, wenn das Polygon von Anfang an bekannt ist. Die Abschätzung in [19] ist jedoch recht grob, der schlimmste bisher bekannte Fall erreicht einen Faktor von 5 [14, 15]. Die untere Schranke für die Erkundung einfacher Polygone liegt bei 1.2825 [14, 15].

Für rechtwinklige Polygone existiert eine sehr einfache Greedy-Strategie von Deng, Kameda und Papadimitrou [7], die einen kompetitiven Faktor von $\sqrt{2}$ erreicht. Bei Polygonen mit Hindernissen kann dagegen kein konstanter kompetitiver Faktor erreicht werden, wie Albers et al. [2] zeigten. Eine Übersicht über existierende Bahnplanungsalgorithmen findet sich bei Kao et al. [31], Berman [5] oder Mitchell [26].

In dieser Arbeit soll untersucht werden, inwiefern die PolyExplore Strategie auf einem realen mobilen Robotersystem einsetzbar ist und welchen Einfluss die gemachten Annahmen wie uneingeschränkte Sicht oder punktförmiger Roboter auf die Strategie haben. Für den Vergleich zwischen den realen Messungen und der Theorie wurde das SAM-Applet[1], welches um die PolyExplore Strategie erweitert wurde, herangezogen.

Das SAM-Applet wurde von U. Bachert im Rahmen einer Diplomarbeit [3] implementiert und basiert auf der *Scout-and-March* Strategie [17]. Diese Strategie ist der Vorläufer von PolyExplore und erreicht einen kompetitiven

[1] für weitere Informationen siehe http://www.geometrylab.de/SAM/

Faktor von 133.

Im Kapitel 2 sollen zunächst die benötigten Strukturen, sowie die Funktionsweise der PolyExplore Strategie erklärt werden.

Das Kapitel 3 beschäftigt sich mit der praktischen Umsetzung der Onlinestrategie. Zuerst wird die Funktionsweise der eingesetzten Hardware des Robotersystems, wie z.B. der Laserscanner oder das Fahrwerk, und ihre Schwächen vorgestellt. Danach werden die benötigten Tools und Verfahren erläutert, um den Roboter zu steuern bzw. seine Schwächen auszugleichen. Basierend auf diesen Verfahren wird PolyExplore dann so modifiziert, dass es auf dem Roboter eingesetzt werden kann. Zusätzlich wird das Konzept der implementierten Software zur Steuerung des Roboters von PolyExplore kurz präsentiert.

Zur Evaluation der Strategie werden Messungen in verschiedenen Szenarien gemacht. Die dabei eingesetzte Szenarien werden zusammen mit den Meßergebnissen in Kapitel 4 vorgestellt und analysiert.

Das Ergebnis der Messungen zusammen mit einem Ausblick auf noch bevorstehende Arbeiten wird in Kapitel 5 präsentiert.

Auf der beigelegten CD befinden sich der Source-Code sowie die Videos von den Experimenten aus dem Kapitel 4.

Kapitel 2

Grundlagen

Dieses Kapitel beschäftigt sich mit der Erklärung der für diese Arbeit notwendigen Strukturen und Verfahren. So wird am Anfang in Kapitel 2.1 das einfache Polygon und das Sichtbarkeitspolygon eines Punktes definiert. Das Kapitel 2.2 beschäftigt sich mit der Definition und Offlineberechnung der Shortest Watchman Route, auf der ein Roboter das Innere des ganzen Polygons erkunden kann. Die Aspekte dieser Offlineberechnung sind für PolyExplore sehr interessant. Der Shortest Path Tree, der in Kapitel 2.3 erklärt wird, ist eine für den PolyExplore notwendige Struktur, um die reflexen Ecken zu klassifizieren. Die PolyExplore Strategie selbst wird in Kapitel 2.4 vorgestellt. Das Kapitel 2.5 erklärt, warum PolyExplore nur in einfachen Polygonen funktioniert.

2.1 Sichtbarkeit in den Polygonen

Die Räume, in denen sich die Roboter bewegen, lassen sich in ihrem Grundriss durch Polygone darstellen. Ein Polygon besteht aus einer endlichen, geschlossenen Folge von Liniensegmenten und dem davon umschlossenen Gebiet. In dieser Arbeit beschränken wir uns auf *einfache Polygone*.

Definition 2 Ein Polygon P ist ein *einfaches Polygon*, wenn der Rand von P zusammenhängend ist und keine Selbstschnitte aufweist. Insbesondere hat ein einfaches Polygon im Innern keine Löcher [22].

Die Liniensegmente aus dem Rand des Polygons werden als *Kanten*, und ihre Endpunkte als *Ecken* von P bezeichnet.

Definition 3 Die Ecken, deren innerer Winkel größer als 180° ist, werden als *reflexe Ecken* bezeichnet [19].

In einem einfachen Polygon können nur die reflexen Ecken die Sicht eines Punktes p im Innern des Polygons einschränken. Dabei ist ein beliebiger

Punkt $q \in P$ von p aus *sichtbar*, wenn das Liniensegment pq ganz in P enthalten ist [22].

Definition 4 Sei P ein Polygon und $p \in P$, dann heißt die Menge $vis(p)$ aller von p aus sichtbaren Punkte das *Sichtbarkeitspolygon* von p in P.

Das Sichtbarkeitspolygon besteht aus den (Teilen von) Kanten von P und künstlichen Kanten, die durch reflexe Ecken entstehen, welche die Sicht einschränken. Ein Beispiel des Sichtbarkeitspolygons ist in der Abbildung 2.1 dargestellt.

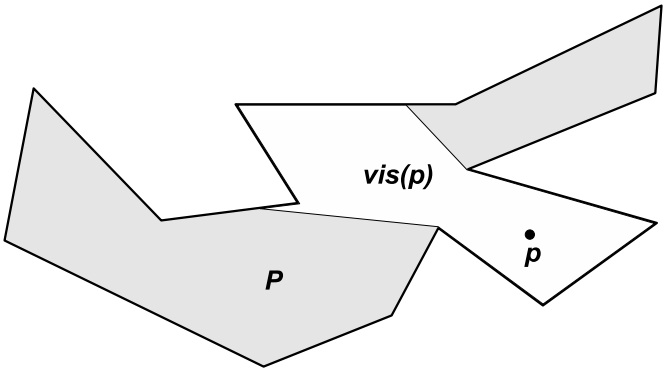

Abbildung 2.1: Das Sichtbarkeitspolygon eines Punktes p in dem Polygon P.

2.2 Shortest Watchman Route

Betrachten wir nun einen Roboter, der sich an einem Startpunkt s in einem einfachen Polygon befindet. Wir nehmen an, dass der Roboter punktförmig ist und über uneingeschränkte Sicht verfügt. Am Punkt s kennt der Roboter im Allgemeinen nicht das ganze Polygon, sondern nur den Teil, der sich durch das Sichtbarkeitspolygon $vis(s)$ dem Roboter erschließt. Bei der Erkundung von Polygonen geht es darum, die Information über das komplette Polygon zu erhalten.

Definition 5 Ein Polygon P gilt als erkundet, wenn der Roboter alle Punkte $q \in P$ mindestens einmal gesehen hat [19].

Der kürzeste Weg, auf dem der Roboter alle Punkte des Polygons sieht und wieder zum Startpunkt zurückkehrt, wird als *Shortest Watchman Route*, kurz *SWR*, bezeichnet. Bei einem einfachen Polygon können nur die reflexen Ecken die Sicht auf weitere Teile des Polygons versperren. Verlängert man die vom Roboter aus nicht sichtbare Kante einer reflexen Ecke ins Innere des Polygons, so muss der Roboter nur diese Verlängerung erreichen, um die Ecke zu erforschen.

Definition 6 Die Verlängerungen der Kanten einer reflexen Ecke ins Innere des Polygons werden als **Cuts** bezeichnet. Ein **notwendiger Cut** einer reflexen Ecke ist einer von den beiden Cuts, der bzgl. des Startpunktes s die Sicht blockiert.

Ein Cut und ein notwendiger Cut sind in der Abbildung 2.2 dargestellt.

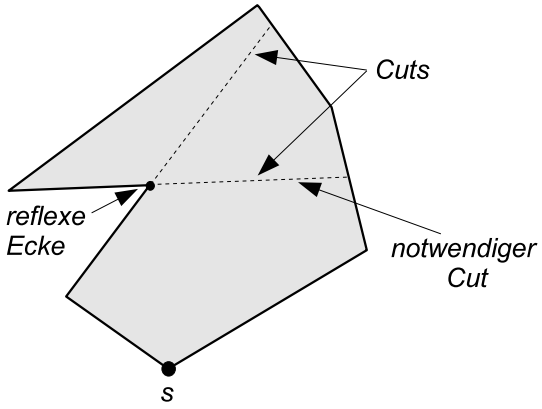

Abbildung 2.2: Ein notwendiger Cut bzgl. des Startpunktes s.

Damit ein Polygon erforscht wird, müssen alle *wesentlichen Cuts* des Polygons besucht werden. Diese sind wie folgt definiert:

Definition 7 Ein Cut c_1 **dominiert** einen Cut c_2, wenn jeder Weg von s zu c_1 den Cut c_2 schneidet. Notwendige Cuts, die von keinem anderen notwendigen Cut dominiert werden, heißen **wesentliche Cuts**.

Bei der Berechnung der Shortest Watchman Route können wir uns auf die wesentlichen Cuts konzentrieren. Die notwendigen Cuts werden dann auf dem Weg zu den wesentlichen Cuts nebenbei erforscht. Die Abbildung 2.3 zeigt ein Beispielpolygon mit den notwendigen und wesentlichen Cuts. c_1, c_2 und c_3 stellen die notwendigen Cuts dar, da jeder Weg zu c_4 oder c_5 über diese Cuts führt. Die Cuts c_4, c_5 und c_6 werden von keinen anderen Cuts dominiert und stellen somit die wesentlichen Cuts dar.

Es gibt viele Arbeiten zur Berechnung von Shortest Watchman Route. An dieser Stelle soll auf eine aktuelle Arbeit verwiesen werden:

Theorem 8 *(Dror, Efrat, Lubiw, Mitchell, 2003)*
In einem einfachen Polygon kann die Shortest Watchman Route zu einem gegebenem Startpunkt s in Zeit $O\left(n^3 \log n\right)$ berechnet werden [9], wobei n die gesamte Anzahl der Kanten des Polygons ist.

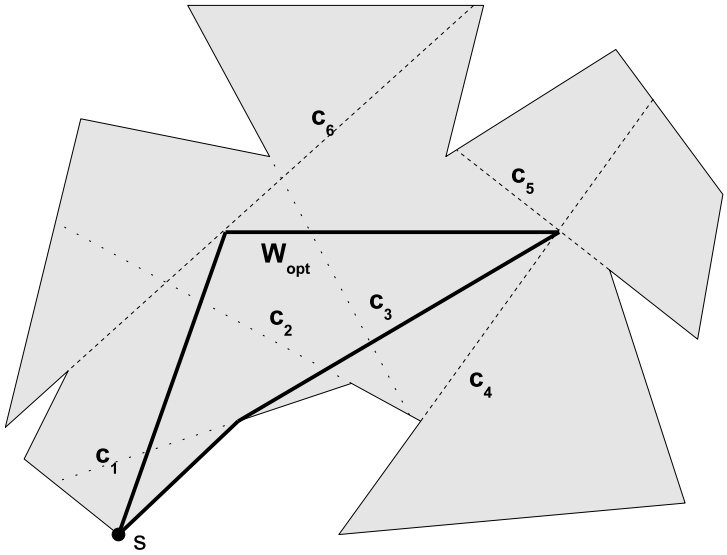

Abbildung 2.3: Shortest Watchman Route, (gepunktet) notwendige und (gestrichelt) wesentliche Cuts des Polygons.

Konkrete Algorithmen zur Berechnung von *SWR* sind Gegenstände der „offline Bewegungsplanung“ für Roboter und werden in dieser Arbeit nicht näher betrachtet.

2.3 Shortest Path Tree

Bevor die Onlinestrategie für die Erkundung einfacher Polygone vorgestellt wird, soll noch eine wichtige Datenstruktur, der *Shortes Path Tree* eines einfachen Polygons vorgestellt werden. Wir betrachten die Gesamtheit aller kürzesten Wege des Polygons P, die von Startpunkt s zu allen Eckpunkten des Polygons gehen. Diese Wege bilden einen Baum der kürzesten Wege (engl. *Shortest Path Tree*), der als *SPT* bezeichnet wird.

Definition 9 Sei P ein Polygon und s der Startpunkt der Exploration auf dem Rand von P. Der **Shortest Path Tree** *SPT(P, s)* enthält die kürzesten in P verlaufenden Wege von s zu allen anderen Ecken von P; Abbildung 2.4 zeigt ein Beispiel eines *SPT*.

Bei einem Polygon mit n Ecken und einem Startpunkt s im Innern des Polygons hat der *SPT* genau $n + 1$ Knoten, darunter $m \leq n$ viele Blätter.

In dieser Arbeit betrachten wir nur einfache Polygone. Bei Polygonen mit Löchern können zu einer Ecke des Polygons mehrere Wege entstehen, wenn etwa der Weg um ein Hindernis links herum genauso lang wäre wie

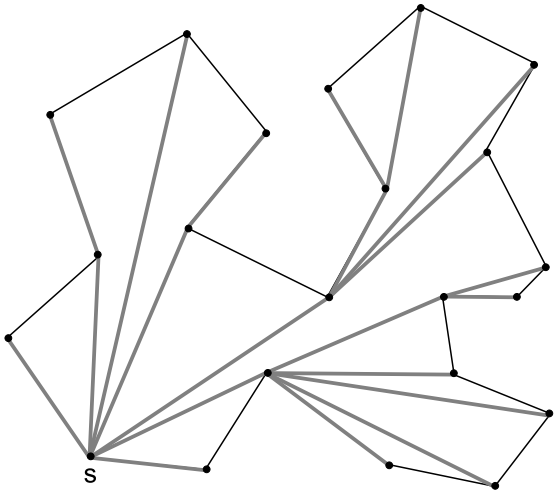

Abbildung 2.4: Das Polygon mit dem Shortest Path Tree von dem Startpunkt s.

rechts herum. Hat das Polygon keine Löcher, so kann die Tatsache, dass keine alternativen Wege existieren für die Onlinestrategien ausgenutzt werden. In diesem Fall kennt die Strategie den kürzesten Weg zu einem Punkt, sobald dieser einmal gesehen wurde.

Um den *SPT* für ein Polygon P mit dem Startpunkt s zu berechnen, können wir zuerst den Sichtbarkeitsgraphen *visG(P, s)* berechnen und anschließend einen abgewandelten Dijkstra-Algorithmus [8] auf *visG(P, s)* anwenden, der zu jeder Ecke des Polygons den kürzesten Weg berechnet.

Der Sichtbarkeitsgraph *visG(P, s)* enthält alle sichtbaren Kanten des Polygons. Um diesen zu berechnen, müssen bei einem Polygon mit n Ecken insgesamt n^2 Kanten mit $n-1$ Kanten auf Schnitt getestet werden, was einen Aufwand von $O(n^3)$ bedeutet. In [21] wird ein Algorithmus vorgestellt, der die Dualität der Segmente ausnutzt und es ermöglicht *visG(P, s)* mit einem Aufwand von $O(n^2)$ zu bestimmen.

Eine andere Möglichkeit wird von Guibas und Hershberger [13] vorgestellt, die es ermöglicht in einem triangulierten Polygon den kürzesten Pfad in $O(\log n + k)$ zu berechnen. Ein einfaches Polygon läßt sich in $O(n)$ triangulieren. Damit kann man den *SPT* mit einem Aufwand von $O(n(\log n + k))$ bestimmen.

2.4 PolyExplore

In diesem Kapitel soll nun die Onlinestrategie für die Erkundung einfacher Polygone beschrieben werden, wie diese von Hoffmann, Icking, Klein und

Kriegel in [19] vorgestellt wurde.

In Kapitel 2.2 haben wir wichtige Aspekte zur Offlineberechnung der Watchman Route kennen gelernt. In einem rechtwinkligem Polygon lässt sich hiermit eine einfache Greedy Explorationsstrategie [7] realisieren: Dazu muss der Roboter alle reflexen unerforschten Ecken im Uhrzeigersinn abwandern. In nicht-rechtwinkligen Polygonen funktioniert diese Strategie nicht. Die Abbildung 2.5 zeigt ein Vergleich zwischen der optimalen Watchman Route und der von der Greedy-Strategie berechneten. Man sieht an diesem Beispiel, dass die Greedy-Strategie in nicht-rechtwinkligen Polygonen nicht kompetitiv ist.

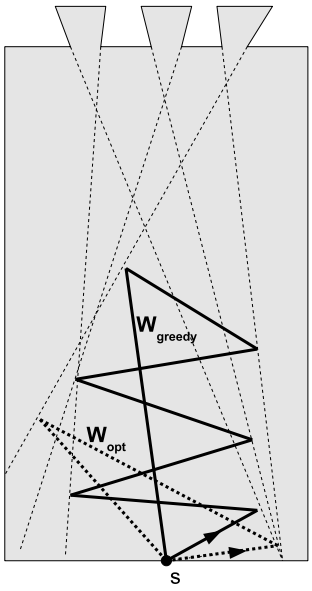

Abbildung 2.5: Greedy-Strategie ist nicht kompetitiv in nicht-rechtwinkligen Polygonen.

An dem Beispiel in Abbildung 2.5 sieht man, dass die Wahl der nächsten, zu erkundenden Ecke für die Exploration nicht unwichtig ist. Daher werden die Wahlkriterien bei PolyExplore für die reflexen Ecken in zwei Kategorien unterteilt: *rechte* und *linke* Ecken.

Definition 10 Sei P ein Polygon, s der Startpunkt und *SPT(P,s)* der Shortest Path Tree von P. Eine reflexe Ecke v von P heißt **linke** Ecke, wenn v links von *SPT(P, s)* liegt, **rechte** Ecke, wenn v rechts von *SPT(P, s)* liegt; siehe Abbildung 2.6.

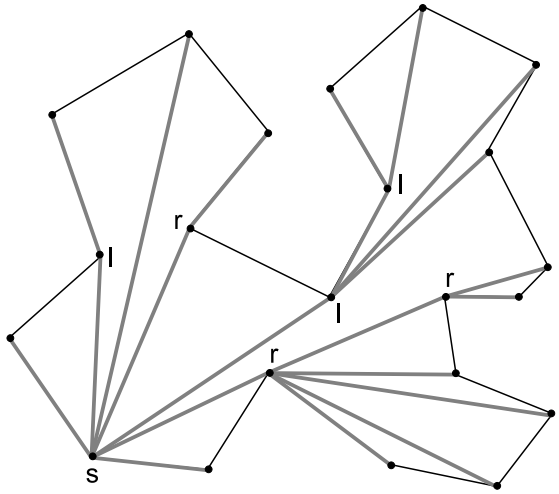

Abbildung 2.6: Shortest Path Tree eines Polygons und linke bzw. rechte Ecken.

2.4.1 Exploration einer Ecke

Eine andere Schwierigkeit bei der Exploration ergibt sich, wenn eine unerforschte Ecke direkt angefahren wird. Dadurch können lange Wege entstehen, obwohl der Roboter nur ein wenig zur Seite hätte fahren müssen, um die Ecke einzusehen. Dies wird in der Abbildung 2.7 *(i)* gezeigt. Ein direktes Anfahren der Ecke ist also nicht kompetitiv. Bei einem unbekannten Winkel α erreicht man einen kompetitiven Faktor von $\frac{\pi}{2}$ durch Anfahren der Ecke auf einem Kreisbogen *arc(s, v)*, siehe Abbildung 2.7 *(ii)*.

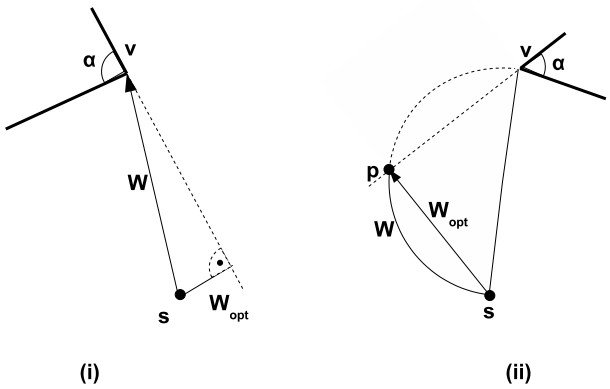

Abbildung 2.7: Erkundung einer reflexen Ecke.

Wenn der Roboter nicht auf einem Kreisbogen fährt, so kann sogar ein noch besserer Faktor erreicht werden. Icking et al. [20] konnten einen kom-

petitiven Faktor von ≈ 1.212 zeigen. Der Kreisbogen ist also kein optimaler Weg, hat für PolyExplore aber nützliche Eigenschaften. So ist der Schnittpunkt p des Kreises *arc(s, v)* mit dem Cut der Ecke v, nach dem Satz des Thales[1], der nächste Punkt zu s.

Der wichtigste Schritt in der Explorationsstrategie ist die Erkundung einer einzelnen Ecke. Diese wird von Algorithmus 2.4.1 behandelt.

Algorithmus 2.4.1 ExploreRightVertex (**inout:** *TargetList, TodoList*)

BasePoint := aktuelle Position des Roboters;
Target := das erste Element der *TargetList*;

while *Target* ist nicht sichtbar **do**
  gehe auf dem kürzesten Weg von *BasePoint* zum *Target*;
end while

Back := die letzte Ecke auf dem kürzesten Weg von *BasePoint* zur aktuellen Position des Roboters;

while *Target* nicht komplett erkundet ist **do**
  gehe auf dem Kreisbogen *arc(Back, Target)*;
  aktualisiere dabei *TargetList* und *TodoList*;
  wenn das erste Element der *TargetList* sich ändert, dann aktualisiere *Target*;
  wenn die Sicht zu *Back* verloren geht, aktualisiere *Back*;

  // Ausnahmen während der Kreisbewegung;
  if der Rand des Polygons P blockiert den Weg **then**
    bewege dich am Rand des Polygons, bis die Kreisbewegung wieder möglich ist;
  end if
  if die Sicht auf *Target* geht verloren **then**
    bewege dich direkt auf das *Target* zu, bis die blockierende Ecke erreicht wird;
  end if
end while

Die Strategie arbeitet mit zwei Listen: *TargetList* und *TodoList*. Am Anfang enthält die *TargetList* alle rechten Ecken, die von dem Startpunkt s aus sichtbar sind, aber noch nicht erkundet wurden. Die Ecken sind im Uhrzeigersinn entlang des Polygonrandes sortiert. Der Roboter erkundet die erste rechte Ecke r der *TargetList*. Diese Ecke kann schon im vorherigen Schritt gesehen worden sein und kann von der aktuellen Position nicht mehr

[1] Satz des Thales: Alle Winkel im Halbkreis sind rechte Winkel.

sichtbar sein. In diesem Fall muss sich der Roboter auf dem kürzesten Weg zu der Ecke r bewegen, bis die Ecke wieder sichtbar wird. Der kürzeste Weg zwischen zwei Punkten in einem einfachen Polygon ist bekannt, sobald die beiden Punkte gesehen wurden (in einem Polygon mit Löchern trifft diese Aussage nicht zu!).

Während der Erkundung der Ecke bewegt sich der Roboter auf dem Kreisbogen *arc(BasePoint, r)*. *BasePoint* ist der Startpunkt, an dem der Roboter mit der Erkundung von r beginnt. Auf dem Erkundungsweg können neue unerforschte Ecken gesehen werden. Wenn das der Fall ist, dann muss entschieden werden, in welche Liste die neue Ecke kommt. Wenn sie eine rechte Ecke ist und diese Ecke auf dem Weg von dem *StagePoint* — in diesem Fall der Startpunkt der aktuellen Erkundung — **nur** hinter rechten Ecken liegt, wird diese Ecke der *TargetList* hinzugefügt, sonst der *TodoList*. Kommt die neue Ecke vor r in der *TargetList*, muss r aktualisiert werden und der Erkundung der neuen Ecke geht auf dem *arc(BasePoint,* r_{neu}*)* weiter.

Wenn durch eine reflexe Ecke b die Sicht auf den Startpunkt blockiert wird, wird b zu dem neuen *BasePoint* und die Erkundung wird auf dem neuen Kreisbogen *arc(b, r)* fortgesetzt, bis der alte *BasePoint* wieder sichtbar oder die Sicht durch einen weitere reflexe Ecke blockiert wird.

Während der Erkundung von r können zwei Ausnahmesituationen auftreten. Zum einen kann der Roboter die Sicht auf r verlieren und zum anderen kann die Bewegung auf dem Kreisbogen durch den Polygonrand blockiert werden.

Im ersten Fall bewegt sich der Roboter dann auf dem direkten Weg auf die Ecke zu, welche die Sicht auf r versperrt. Sobald r wieder sichtbar wird, soll wieder auf dem *arc(BasePoint, r)* erkundet werden.

Im zweiten Fall bewegt sich der Roboter entlang des Polygonrandes, bis die Bewegung auf dem Kreisbogen *arc(BasePoint, r)* wieder möglich ist.

Wird eine rechte Ecke r erkundet und hat r noch unerforschte linke Ecken, so wird r mit den unerforschten linken Ecken der *TodoList* für spätere Bearbeitung hinzugefügt.

Die Abbildung 2.8 zeigt, wie die Strategie bei der Erkundung einer rechten Ecke vorgeht. Am Anfang ist von dem Startpunkt s aus nur r_3 sichtbar, folglich ist nur r_3 in der *TargetList*. Der Roboter beginnt mit der Erkundung auf dem Kreisbogen $arc(s, r_3)$. Im Punkt a wird r_2 sichtbar und kommt in die *TargetList*. Da r_2 vor r_3 einsortiert wird, wird jetzt r_2 auf dem Kreisbogen $arc(s, r_2)$ erkundet.

Im Punkt b wird die Sicht auf r_2 durch l blockiert. In diesem Fall bewegt sich der Roboter direkt auf die Ecke l zu. Von da aus ist die Kreisbewegung wieder möglich. Aber dadurch wird die Sicht auf s durch l blockiert. Da l jetzt der letzte Punkt auf dem kürzesten Weg von s zu der aktuellen Position ist, wird die Erkundung von r_2 auf dem Kreisbogen $arc(l, r_2)$ bis c

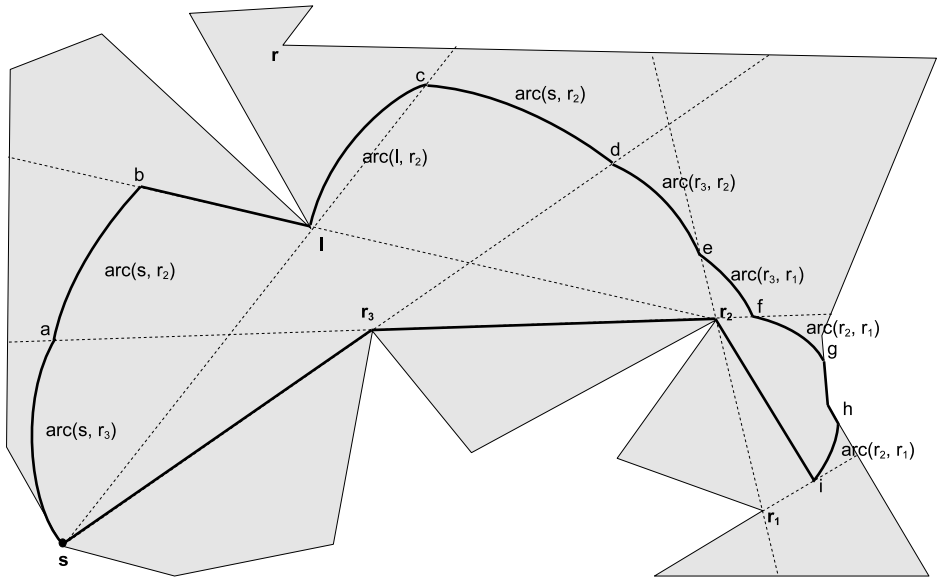

Abbildung 2.8: Erkundung einer rechten Ecke.

fortgesetzt.

In l wird außerdem auch r sichtbar. Da aber der kürzeste Weg von s zu r über die linke Ecke l führt, wird r nicht der *TargetList* hinzugefügt, sondern kommt in die *TodoList*.

Im Punkt c wird wieder der Startpunkt s sichtbar. In diesem Fall wird die Erkundung auf dem Kreisbogen $arc(s, r_2)$ fortgesetzt. In d wird die Sicht auf s wieder blockiert, diesmal durch r_3. Also wird die Erkundung auf dem Kreisbogen $arc(r_3, r_2)$ fortgesetzt.

In e wird die rechte Ecke r_1 sichtbar und kommt in die *TargetList*. Die Erkundung geht nun auf dem Kreisbogen $arc(r_3, r_1)$ weiter bis in f die Sicht auf r_3 durch r_2 blockiert wird. Die Erkundung auf dem Kreisbogen $arc(r_2, r_1)$ wird in g durch den Polygonrand blockiert, so dass der Roboter gezwungen ist die Erkundung entlang des Randes fortzusetzen, bis in h die Kreisbewegung wieder möglich ist. In i ist die Erkundung der rechten Ecke beendet.

Theorem 11 *(Hoffman, Icking, Klein, Kriegel, 2001) Der Weg des Roboters, von dem Startpunkt bis zu dem Cut der zu erforschenden Ecke, der durch den Algorithmus 2.4.1 erzeugt wird, ist nicht länger als das doppelte des kürzesten Weges, der zum Erforschen der Ecke nötig ist. [19]*

Um eine linke Ecke zu erforschen, kann symmetrisch vorgegangen werden, indem rechts durch links ersetzt wird und die Ecken nun gegen den Uhrzeigersinn in der *TargetList* sortiert werden.

2.4.2 Exploration einer Gruppe von Ecken

Nachdem der Algorithmus 2.4.1 fertig ist und die Ecke erforscht ist, kann die *TargetList* immer noch unerforschte rechte Ecken beinhalten. Die auf dem Weg gesammelten Ecken in der *TargetList* bezeichnen wir als eine Gruppe von rechten Ecken. Die Erkundung einer solchen Gruppe beginnt in einem *StagePoint*. Die wichtigste Eigenschaft eines solchen *StagePoints* ist, dass diese auch von der *Shortest Watchmann Route* besucht werden. Der erste *StagePoint* ist der Startpunkt des Roboters.

Die Erkundung der Gruppe der rechten Ecken wird im Algorithmus 2.4.2 beschrieben.

Algorithmus 2.4.2 ExploreRightGroup (**in:** *TargetList*, **out:** *TodoList*)

StagePoint := aktuelle Position des Roboters;
TodoList := leere Liste

while *TargetList* ist nicht leer **do**
ExploreRightVertex (TargetList, TodoList)
// an dieser Stelle befindet sich der Roboter auf dem Cut des letzten *Targets*;
gehe auf dem Cut zu dem Punkt mit dem kürzesten Abstand von *StagePoint*, aktualisiere dabei *TargetList* und *TodoList*;
end while

gehe auf dem kürzesten Weg zu *StagePoint*;

Bei der Erkundung der rechten Gruppe ist der *StagePoint* immer eine linke Ecke. Die *TodoList* ist am Anfang der Exploration einer Gruppe immer leer, die *TargetList* enthält alle sichtbaren rechten Ecken, die entlang des Polygonrandes im Uhrzeigersinn sortiert sind. Nun wird die Prozedur *ExploreRightVertex* aus dem Algorithmus 2.4.1 so oft ausgeführt, bis die *TargetList* keine Ecken mehr enthält. Wenn das der Fall ist, fährt der Roboter zum *StagePoint* zurück. Die Abbildung 2.9 zeigt ein Beispiel für die Funktionsweise der Prozedur *ExploreRightGroup*.

Die Erkundung der rechten Gruppe beginnt in s, also ist s damit der *StagePoint*. Die *TargetList* enthält am Anfang die Ecken r_6 und r_5. Da r_5 als erste Ecke in der *TargetList* ist, bewegt sich der Roboter nun auf dem Kreisbogen $arc(s, r_5)$, bis in a ein weitere Ecke r_1 entdeckt wird. Die Erkundung wird auf dem Kreisbogen $arc(s, r_1)$ fortgesetzt, bis in b die Ecke r_1 erforscht ist. Damit ist der erste Durchlauf der Prozedur *ExploreRightVertex* beendet. Da b der Punkt auf dem Cut von r_1 ist, der die kürzeste Entfernung zu s hat, beginnt der zweiter Durchlauf von *ExploreRightVertex*. Diesmal wird r_2 erforscht mit b als *BasePoint*. In d ist r_2 erkundet und *Ex-*

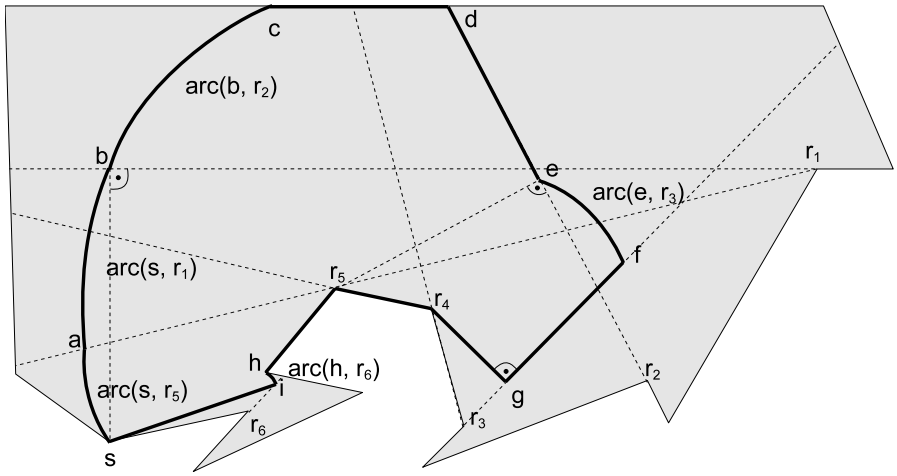

Abbildung 2.9: Erkundung einer Gruppe von rechten Ecken.

ploreRightVertex beendet. Zu diesem Zeitpunkt enthält die *TargetList* noch r_3 und r_6. Die Cuts r_4 und r_5 wurden nebenbei erforscht. Bevor *ExploreRightVertex* wieder aufgerufen wird, muss sich der Roboter zu dem Punkt e bewegen, da dieser Punkt auf dem Cut von r_2 die kürzeste Entfernung zu s hat. Wenn der Roboter bei e ist, wird die Ecke r_3 auf dem Kreisbogen $arc(e, r_3)$ erkundet. In f ist r_3 erkundet und die *TargetList* enthält nur noch r_6. Wieder bewegt sich der Roboter auf dem Cut, diesmal von r_3. *ExploreRightVertex* veranlasst den Roboter anschließend auf dem kürzesten Weg bis h zu fahren, bevor r_6 auf dem Kreisbogen $arc(h, r_6)$ erkundet wird. In i sind alle Ecken erkundet und der Roboter kehrt zu s zurück. Damit ist die Prozedur *ExploreRightGroup* beendet.

Mit den Prozeduren *ExploreRightGroup* und der dazu symmetrischen *ExploreLeftGroup* kann man nun verschiedene Gruppen von Ecken erforschen.

2.4.3 Exploration des ganzen Polygons

Damit das ganze Polygon erforscht werden kann, müssen die beiden Prozeduren *ExploreRightGroup* und *ExploreLeftGroup* rekursiv verschachtelt werden. Dies wird durch den Algorithmus 2.4.3 erreicht.

Die Prozedur *ExploreRightGroupRec* erkundet alle rechten Ecken, die in der *TargetList* enthalten sind. Die auf dem Weg erzeugte *TodoList* enthält jetzt rechte Ecken, hinter denen noch unerforschte linke Ecken sind. Es werden nur die rechten Ecken behalten, die im *SPT* nicht hinter einer rechten Ecke sind, die auch in der *TodoList* ist. Die Ecken werden sowohl in *ExploreRightGroupRec* als auch in der dazu symmetrischen Prozedur *ExploreLeft-*

Algorithmus 2.4.3 ExploreRightGroupRec (**in:** *TargetList*)

ExploreRightGroup (TargetList, TodoList); // *TodoList* wird gefüllt
TodoList: behalte nur die rechten Ecken aus der *TodoList*, die im SPT keine Vorfahren aus der *TodoList* haben;

for alle Ecken v aus der *TodoList* im Uhrzeigesinn **do**
  gehe auf dem kürzesten Weg zu v; // v ist neuer *StagePoint*
  NewTargetList := alle bekannten unerforschten linken Ecken, die Kinder von v sind (gegen Uhrzeigersinn sortiert);
  ExploreLeftGroupRec (NewTargetList);
end for

GroupRec im Uhrzeigersinn sortiert. Nun wird jede dieser Ecken v aus der neuen *TodoList* auf dem kürzesten Weg angefahren und von dort aus *ExploreLeftGroupRec* aufgerufen. Die *TargetList* enthält alle zu dem Zeitpunkt unerforschten linke Ecken, die Kinder von v im SPT sind.

Zum Schluss stellt der Algorithmus 2.4.4 die Prozedur zur Verfügung, die das ganze Polygon erkundet. Zuerst werden mit *ExploreRightGroup* alle vom Startpunkt s sichtbaren rechten Ecken erkundet. Und anschließend wird die rekursive Prozedur *ExploreLeftGroupRec* aufgerufen, welche rekursiv alle anderen Ecken erkundet.

Algorithmus 2.4.4 PolyExplore (**in:** P, s)

TargetList := { alle von s aus sichtbaren rechten Ecken, im Uhrzeigersinn nach der Reihenfolge auf dem Polygonrand von P sortiert };
ExploreRightGroup (TargetList, TodoList);
TargetList := { alle linken Ecken, die im SPT Kinder der Ecken in *TodoList* sind **und** alle von s aus sichtbaren linken Ecken };
sortiere *TargetList* gegen den Uhrzeigersinn;
ExploreLeftGroupRec (TargetList);

Theorem 12 *(Hoffman, Icking, Klein, Kriegel, 2001)*
Die Prozedur PolyExplore *im Algorithmus 2.4.4 erkundet ein einfaches Polygon mit einem kompetitivem Faktor von 26.5. [19]*

In der Praxis ist bisher kein Polygon bekannt, in dem diese Strategie den Faktor von 26.5 erreicht hat. Der schlechteste bekannte Faktor liegt bei etwa 5 [14, 15]. Daher liegt die Vermutung nahe, dass der Algorithmus besser ist, als der bewiesene Faktor.

2.5 Exploration von Polygonen mit Löchern

Die in Kapitel 2.4 beschriebene Strategie funktioniert nur in einfachen Polygonen. Bei den Polygonen mit Löchern funktioniert diese nicht, da die Strategie nur den äußeren Rand von dem Polygon absucht. Das kann man anhand eines einfachen Beispiels zeigen. In der Abbildung 2.10 sieht der Pfad π zwar alle inneren und äußeren Ränder und Ecken, den schraffierten Bereich im Inneren jedoch nicht.

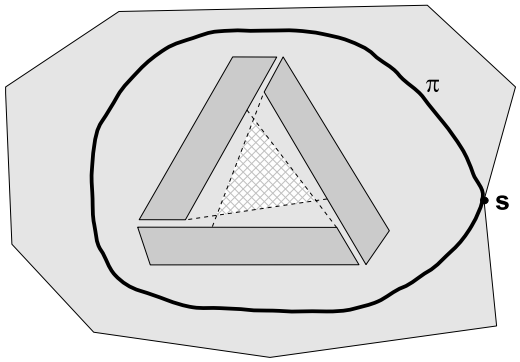

Abbildung 2.10: Bei einem Polygon mit Löchern reicht es nicht alle Ränder zu erkunden.

Tatsächlich lässt sich ein Polygon mit Löchern nicht mit einem konstanten kompetitiven Faktor erkunden.

Theorem 13 *(Albers, Kursawe, Schuierer, 1991)*
Sei A eine Online Strategie, die ein Polygon mit n Löchern mit einem nicht konstant kompetitiven Faktor c erkundet, dann gilt [2]:

$$c \in \Omega\left(\sqrt{n}\right)$$

Kapitel 3

PolyExplore und das reale Robotersystem

Das reale Robotersystem unterscheidet sich stark von dem seinem theoretischen Bruder. So hat es immer eine Ausdehnung und kann sich meistens nicht ideal auf einer vorgegebenen Linie bewegen. Seine Bewegungen müssen eventuell korrigiert werden. Seine Sicht ist auch nicht unbeschrängt und hängt stark ab von den Sensoren, mit welchem es die Umgebung wahrnimmt. So können z.B. Sonar- bzw. Laserscanner oder Videokamera als Sensoren eingesetzt werden. Kapitel 3.1 beschäftigt sich mit den Systemen des Roboters, die in dieser Arbeit eingesetzt wurden, um die PolyExplore Strategie zu evaluieren.

Bevor die PolyExplore Strategie Entscheidungen für das weitere Vorgehen treffen kann, müssen zuerst die Sensordaten verarbeitet werden. Mit diesen Sensordaten muss zuerst eine Karte generiert bzw. aktualisiert werden. Die Karte kann anschließend benutzt werden, um Entscheidungen für die Erkundung zu treffen oder die Position des Roboters in der Karte zu korrigieren. Verfahren für die Erstellung der Karte und die Korrektur der Position des Roboters werden in Kapitel 3.2 vorgestellt.

Während der Erkundung wird die Karte ständig aktualisiert. Aufgrund der Änderungen der Karte muss PolyExplore eventuell alte Entscheidungen revidieren oder neue treffen. Dabei sind diese Entscheidungen auch von der aktuellen Position des Roboters abhängig. Welche Anpassungen und Änderungen in der Strategie vorgenommen werden müssen, um sie auf einem realen Robotersystem einsetzen zu können, wird in Kapitel 3.3 beschrieben.

Um die Software für weitere oder verbesserte Verfahren z.B. zur Kartenerstellung oder Lokalisierung erweiterbar zu halten, wurde ein modularer Aufbau angestrebt. Das Kapitel 3.4 stellt das Konzept der implementierten Software vor.

3.1 Reale Robotersysteme

Als Robotersystem bzw. Roboter werden stationäre oder mobile Maschinen bezeichnet, die nach einem bestimmten Programm festgelegte Aufgaben selbständig oder ferngesteuert erfüllen [35]. Je nach Einsatzgebiet oder Zweck werden die Roboter mit unterschiedlichen Systemen bestückt. Da Tests mit den realen Systemen oft aufwendig, zeitraubend und teuer sind, existiert zu einem Robotersystem fast immer eine Simulationssoftware. Damit lassen sich Probleme einfacher untersuchen und mögliche Fehler in der Durchführung der gestellten Aufgaben rechtzeitig erkennen. In diesem Kapitel werden die benötigten Systeme für die Sensorik, das Fahrwerk und die Odometrie des verwendeten realen Systems beschrieben.

3.1.1 Sensorik

Der Laserscanner in der Abbildung 3.1 ist ein berührungsloses Messsystem der Firma SICK, das die Umgebung zweidimensional in einem Winkel von maximal 180° abtastet.

Abbildung 3.1: Lasermesssystem 200 der Firma SICK.

Die Entfernung wird im Scanner anhand der Pulslaufzeit bestimmt. Dazu wird ein Laserimpuls ausgesandt. Trifft der Laserimpuls auf ein Objekt, wird er reflektiert und im Empfänger des Scanners registriert. Die Zeit zwischen dem Aussenden und Empfangen des Impulses ist direkt proportional zur Entfernung zwischen dem Scanner und dem Objekt (Lichtlaufzeit). Durch den internen Drehspiegel wird der gepulste Laserstrahl abgelenkt und die Umgebung mit maximal 75 Hz flächenförmig abgetastet [33, S. 6]. Die Anzahl der daraus resultierenden Entfernungswerte hängt dabei von der Winkelauflösung und dem maximalen Scanwinkel ab. Bei dem Laserscanner LMS 200 kann alle 0.25°, 0.5° bzw. 1° ein Entfernungswert ermittelt werden. Bei einer Winkelauflösung von 180° ergeben sich 721, 361 bzw. 181 Messwerte, die von der externen Anwendung ausgewertet werden können. Die einzelnen Werte werden hintereinander ausgegeben, so kann anhand der Position im Datenarray die jeweilige Winkelstellung zugeordnet werden (siehe Abbildung 3.2). Als tastendes System benötigt der Scanner keine Reflektoren oder Positionsmarken. Die Reichweite eines solchen Systems hängt dabei von

Abbildung 3.2: Drehrichtung des Scanners LMS 200.

dem jeweiligen Remission[1] des Objektes, sowie der Sendestärke des Scanners ab. Zur Übersicht sind in der Tabelle 3.1 einige Remissionswerte bekannter Materialien aufgeführt.

Material	**Remission**
Photokarton, schwarz matt	10%
Karton, grau	20%
Holz (Tanne roh, verschmutzt)	40%
PVC grau	50%
Papier, weiß matt	80%
Aluminium, schwarz eloxiert	110...150%
Stahl, rostfrei glänzend	120...150%
Stahl, hochglänzend	140...200%
Reflektoren	>2000%

Tabelle 3.1: Remissionswerte bekannter Materialien nach dem KODAK-Standard.

In Abhängigkeit von der Remission sind die Reichweiten in der Abbildung 3.3 dargestellt worden.

3.1.2 Odometrie und Fahrwerk

Die Odometrie ist die Technik der Positionsbestimmung eines Fahrzeuges durch die Beobachtung seiner Räder. Die Odometrie ist ein grundlegendes Navigationsverfahren für bodengebundene Fahrzeuge aller Art (Kfz, Roboter), allerdings wird es aufgrund seiner Fehlereigenschaften selten als alleiniges Verfahren eingesetzt [35].

Bei dem Pioneer2 AT besteht das Fahrwerk aus zwei separaten Elektromotoren, einer für die linke und einer für die rechte Seite. Damit lässt sich

[1] Eine Reflektion, deren Wert relativ zur Reflektionsstärke an einer matten weißen Oberfläche nach dem KODAK-Standard bestimmt wird.

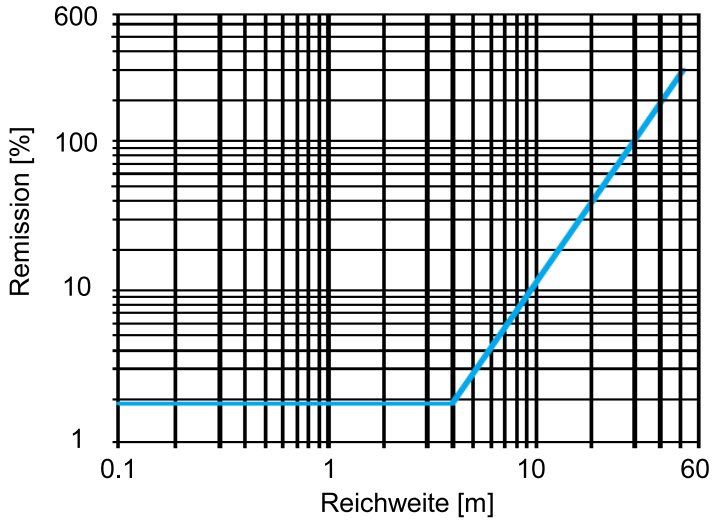

Abbildung 3.3: Reichweite von LMS 200 / LMS 220 in Abhängigkeit der Objektremission. Quelle [33, S. 7].

jede Seite einzeln steuern. Das ist wichtig, damit der Roboter sich drehen kann, da die Räder und Achsen über keinerlei Lenkung verfügen.

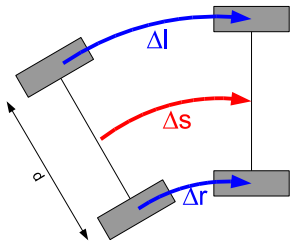

Abbildung 3.4: Berechnung der Odometriewerte

Zur Positionsbestimmung wird für jede Seite die Anzahl n der Radumdrehungen gemessen. Zusammen mit dem bekannten Radumfang u kann nun die Wegdifferenz $\triangle l$ bzw. $\triangle r$ für jede Seite bestimmt werden (siehe Abbildung 3.4):

$$\triangle l = \pi \cdot u \cdot n_{left} \quad (3.1)$$

$$\triangle r = \pi \cdot u \cdot n_{right} \quad (3.2)$$

Die Wegdifferenz $\triangle s$ ergibt sich mit 3.1 und 3.2:

$$\triangle s = (\triangle l + \triangle r)/2$$

Da beim Pioneer2 AT die Räder fest auf einer Achse sind, wird die

Änderung der Fahrtrichtung $\triangle\phi$ aus der Differenz des zurückgelegten Weges zwischen der linken und rechten Seite bestimmt:

$$\triangle\phi \quad = \quad (\triangle l + \triangle r)\,/d$$

Odometrische Messverfahren zur Positionsbestimmung sind sehr stark von der Hardware abhängig. Dadurch können sich sehr schnell Fehler in die Positionsbestimmung einschleichen. Diese Fehler kann man in drei Arten einteilen:

- Entfernungsfehler: Fehler in der Entfernung nach dem Zurücklegen einer geraden Strecke.
- Drehfehler: Fehler, die beim Drehen des Roboters auftreten.
- Driftfehler: Fehler in der Orientierung nach dem Zurücklegen einer geraden Strecke

Diese Fehler können unterschiedliche Ursachen haben. Das kann z.B. an dem unterschiedlichen Reifendruck, schlechter Bodenhaftung, Abnutzungserscheinungen, usw. liegen. Aus diesem Grund können Odometriedaten zur Orientierung genommen werden, müssen aber durch andere gewonnene Daten, z.B. Kompass, Lasermessung, korrigiert werden.

In der Abbildung 3.5 ist ein Beispiel für den Odometriefehler, der nach drei Drehungen auf dem Roboter Pioneer2 AT entstanden ist.

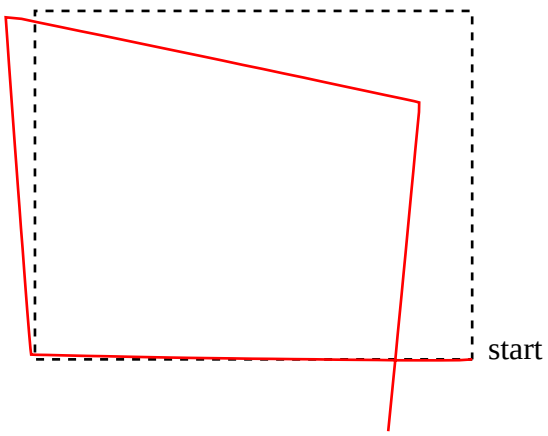

Abbildung 3.5: Odometriefehler bei dem Roboter Pioneer2 AT.

Die gestrichelte Linie zeigt den vom Roboter tatsächlich zurückgelegten Weg und die durchgezogene Linie stellt den von der Odometrie des Roboters bestimmten Weg dar. Man sieht in diesem speziellen Fall, dass die Fehler in diesem Beispiel besonders durch die Drehungen entstanden sind. Drift- bzw. Entfernungsfehler sind hier nicht erkennbar.

3.1.3 ARIA und die Simulationssoftware

Der bei den Versuchen eingesetzte Roboter Pioneer2 AT wurde von *ActivMedia*[2] hergestellt. Zu den ausgelieferten Robotern stellt ActivMedia auch die Programmbibliothek ARIA(Advanced Robotics Interface Application) zur Verfügung. ARIA ist eine objektorientierte Schnittstelle für die Entwickler, die in C++ geschrieben ist und unter Linux oder Win32 eingesetzt werden kann. Sie bietet dem Entwickler die Möglichkeiten, den Roboter und seine Peripherie Geräte zu steuern, die Sensordaten des Roboters abzufragen, uvw. ARIA kommuniziert mit dem Roboter nach dem Client/Server Prinzip. Dabei wird die serielle Schnittstelle benutzt, um mit dem Roboter zu kommunizieren.

Es gibt auch die Möglichkeit den realen Roboter zu simulieren. In diesem Fall wird eine TCP/IP Verbindung genutzt, um mit dem Simulator zu kommunizieren. Der Simulator wird auch von ActivMedia zur Verfügung gestellt. Er bietet die Möglichkeit, die Eigenschaften und das Verhalten jedes Roboters von ActivMedia zu simulieren. Weiterhin kann jede durch Liniensegmente darstellbare 2D Umgebung einfach erstellt und simuliert werden. Das ermöglicht das einfache Testen der selbsterstellten Software, bevor diese auf dem realen Roboter eingesetzt wird.

3.2 Kartendarstellung

Damit ein autonomer Roboter von A nach B fahren kann, muss er seinen Weg planen. Dieses geschieht meistens anhand einer Karte. Eine Karte kann der Roboter online generieren oder sich einer Fertigen bedienen, die er eventuell auch pflegen kann. Die Karten müssen so aufgebaut sein, dass der Roboter mit seinem Sichtsystem, z.B. Sonar- oder Laserscanner, sich in diesen Karten lokalisieren kann. Es gibt unterschiedliche Strukturen, wie die Karten intern dargestellt werden können. Die wohl verbreitetste Kartendarstellung ist das sogenannte *occupancy probability grid*. Die Karte wird durch ein Gitternetz dargestellt. Dabei wird in jede Zelle ein Wahrscheinlichkeitswert für ein Hindernis eingetragen. Der Grundstein für solche Karten wurde in den 80er von Elfes und Moravec in [27] gelegt. Seitdem entstanden viele Arbeiten, welche unter anderem auch auf die speziellen Eigenschaften

[2]für weitere Informationen siehe http://www.mobilerobots.com

der Sonar- bzw. Laserscanner eingehen, siehe [6], [12]. Es entstanden auch effektive Verfahren zur Lokalisierung in den Grid-Karten, wie in [16] vorgestellt wird. Der Vorteil dieser Kartendarstellung liegt in der Möglichkeit der schnellen Aktualisierung. Die Karten sind in ihrer Struktur auch robust, was sich bei Odometriefehler der Roboter als vorteilhaft erweist. Da die PolyExplore Strategie auf einem Polygon aufbaut, müsste entweder die Strategie modifiziert oder die Grid-Karte bei jeder Aktualisierung in ein Polygon umgewandelt werden. Daher erscheint es sinnvoll, direkt eine polygonale Karte zu erstellen.

Bei den polygonalen Karten werden die Sensordaten zu Liniensegmenten aggregiert. In den letzten Jahren sind einige Arbeiten mit Methoden zu der Erstellung der polygonalen Karten erschienen. Leider sind einige von diesen wie z.B. [34] nicht online tauglich, da mehrere Durchläufe durch alle Lasermesspunkte benötigt werden. Andere wiederum haben keine Verfahren zur Lokalisation des Roboters in der erstellten Karte, wie z.B. [30] oder [23]. In [11] wurde eine Methode zur Kartenerstellung mit Lokalisierung vorgestellt. Leider ist das Verfahren gegenüber den Odometriefehlern nicht immer robust. Wie aus den Beispielen in der Arbeit hervor geht, konnte bei längeren Wegen auf dem Pioneer3 AT der Roboter selbstständig in einen vorgegeben Raum fahren, aber die Karte wies grobe Fehler auf.

Nach längerer Recherche stellte sich heraus, dass die Implementation eines robusten Verfahrens für die Kartenerstellung und Lokalisierung den Rahmen dieser Diplomarbeit sprengen würde. Das Verfahren müsste robust und gleichzeitig auch schnell sein, damit es während der Fahrt mehrmals pro Sekunde die neuen Laserscans in die Karte einpflegen könnte. Aus diesem Grund wird hier ein „einfaches“ Verfahren zur polygonalen Kartenerstellung und Lokalisierung vorgestellt, welches für einfache Polygone und für erste Erkenntnisse der PolyExplore Strategie in der realen Umgebung ausreichend ist. Das Kapitel 3.2.1 beschreibt das Verfahren, wie aus den Laser Messpunkten das Sichtbarkeitspolygon erstellt wird. Im nächsten Kapitel 3.2.2 wird dann das Sichtbarkeitspolygon in die bestehende Karte ohne Fehlerkorrektur integriert. Das Kapitel 3.2.3 zeigt, wie man einfache Fehlerkorrektur der aktuellen Position des Roboters realisieren kann.

3.2.1 Erstellung des Sichtbarkeitspolygons

Die Daten, die aus einem 2D Lasermessystem kommen, bestehen aus einzelnen Datenpunkten, die für jeden Winkel eine Entfernung enthalten. Da die Positionsrichtung des Roboters aus seiner Odometrie bekannt ist, werden die Entfernungen in einzelne Punkte des Kartesischen Koordinatensystems transformiert. Das Ziel es jetzt, aus diesen Punkten das Sichtbarkeitspolygon zu bestimmen, welches die aktuelle Sicht des Roboter repräsentiert. Es existieren viele Verfahren, die aus einer Punktemenge die Liniensegmente

extrahieren. Einige von diesen wurden in [28] vorgestellt und miteinander verglichen. Für die Berechnung des Sichtbarkeitspolygons wurde das *Split-and-Merge* Verfahren ausgewählt. Dieses wurde erstmals in [29] vorgestellt. Der Algorithmus 3.2.1 zeigt die Erstellung des Sichtbarkeitspolygons, wie es für PolyExplore implementiert wurde. Bei der Berechnung können wir die Tatsache ausnutzen, dass die Punkte aus dem Laser schon in der richtigen Reihenfolge vorliegen und wir keine weitere Zuordnung durchführen müssen.

Algorithmus 3.2.1 Split-and-merge (**in:** *Points nach Winkeln sortiert*)

Line := verbinde den ersten Punkt p_1 mit dem letzten Punkt p_n der Punktemenge;
i := der Index des Punktes mit dem größten Abstand zu der *Line*;
d := der Abstand zwischen *Line* und p_i;

if d < *threshold* **then**
  // Erzeuge zwei Punktemengen für weitere Berechnung;
  P_1 := alle Punkte von p_1 bis p_i;
  P_2 := alle Punkte von p_i bis p_n;
  Split-and-merge(P_1);
  Split-and-merge(P_2);
else
  Erstelle aus *Points* ein Liniensegment;
end if

Der Algorithmus zerlegt zuerst rekursiv die Punktemenge in einzelne Mengen, die jeweils eine Gerade repräsentieren. Dazu wird als erstes eine Gerade gebildet, die durch den ersten und letzten Punkt der Punktemenge geht. Danach wird der Punkt p_{max} bestimmt, der den größten Abstand zu der Gerade hat. Ist der Abstand größer als der Schwellwert wird die Punktemenge an der Stelle p_{max} in zwei Mengen aufgeteilt, bis schließlich alle Punkte im Schwellwertbereich liegen. Das wird noch mal in der Abbildung 3.6 verdeutlicht. Während der Experimente in der realen Umgebung hat sich der Schwellwert von 5 cm als guter Wert erwiesen.

Aufgrund der begrenzten Sicht und der Winkelauflösung des Laserscanners muss beim Zerlegen der Punktemenge auf ein weiteres Kriterium überprüft werden. Wie man in Abbildung 3.7 sieht, können aufeinander folgende Messpunkte auf einer Geraden liegen und trotzdem zu unterschiedlichen Segmenten gehören. Aus diesem Grund müssen jeweils zwei benachbarte Messpunkte auf ihren Abstand zu einander überprüft und die Punktemenge gegebenenfalls zerteilt werden.

Auf die nicht mehr zerlegbaren Punktemenge wird die *Gaußsche Methode der kleinsten Quadrate* [35] angewandt, um eine Approximationsgerade f, wie sie in der Abbildung 3.8 dargestellt wird, zu erstellen. Hat die Menge zu

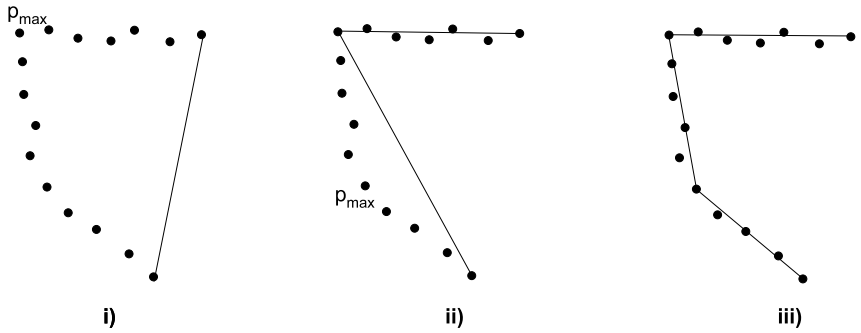

Abbildung 3.6: Der Split-Schritt bei der Erstellung des Sichtbarkeitspolygons.

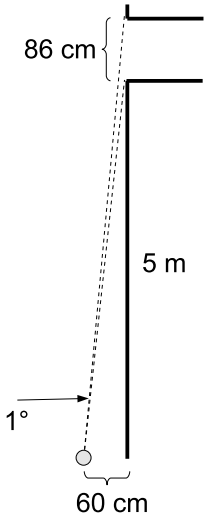

Abbildung 3.7: Abstand der Messpunkte bei einem Grad Unterschied.

wenig Punkte, wird die Menge verworfen.

$$f(x) \quad = \quad b + m \cdot x$$

Die Parameter b und m sollen so gewählt werden, dass die Quadrate der Abweichungen zwischen der entsprechenden Approximationsgerade und den Daten minimal wird im Vergleich zu anderen Wahlen der Parameter, in eine Formel gefasst:

$$\sum_{i=1}^{n} (f(x_i) - y_i)^2 \quad = \quad Minimal \tag{3.3}$$

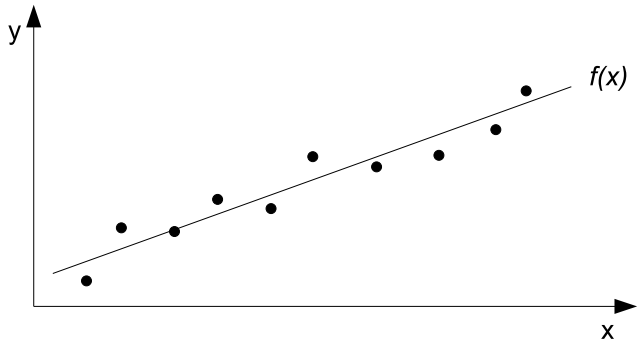

Abbildung 3.8: Approximationsgerade für die Messpunkte.

Als Lösung der Extremwertaufgabe in 3.3 erhalten wir für b und m:

$$b = \frac{\sum_{i=1}^{n} x_i^2 \cdot \sum_{i=1}^{n} y_i - \sum_{i=1}^{n} x_i \cdot \sum_{i=1}^{n} (x_i \cdot y_i)}{N}$$
$$m = \frac{n \cdot \sum_{i=1}^{n} (x_i \cdot y_i) - \sum_{i=1}^{n} x_i \cdot \sum_{i=1}^{n} y_i}{N}$$

mit $N = n \cdot \sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2$

Da in unserem Fall die Messpunkte nach Winkeln sortiert sind, können wir mit den Parametern m und b den ersten und letzten Punkt der Menge verschieben, so dass wir ein Sichtbarkeitssegment erhalten. Die so entstandenen Segmente bilden das aktuelle Sichtbarkeitspolygon des Roboters.

Bei der Berechnung der Parameter m und b muss noch beachtet werden, dass nicht alle Segmente entlang der X-Achse liegen. Ist die maximale Differenz der Y-Koordinaten zwischen je zwei Punkten größer als zwischen den X-Koordinaten, so liegt ein Segment entlang der Y-Achse. In diesem Fall müssen die x und y Werte der Messpunkte vertauscht werden, damit eine richtige Approximationsgerade entsteht, siehe Abbildung 3.9.

Für das aktuelle Verfahren wurden Parameter ausgewählt, wie diese in der Tabelle 3.2 aufgeführt sind.

Beschreibung	**Wert**
Minimale Anzahl an Punkten für die Erstellung der Approximationsgerade	5
Maximaler Abstand zwischen zwei benachbarten Punkten	300 mm
Schwellwert für den Split-Schritt	50 mm

Tabelle 3.2: Parameter für die Erstellung des Sichtbarkeitspolygons.

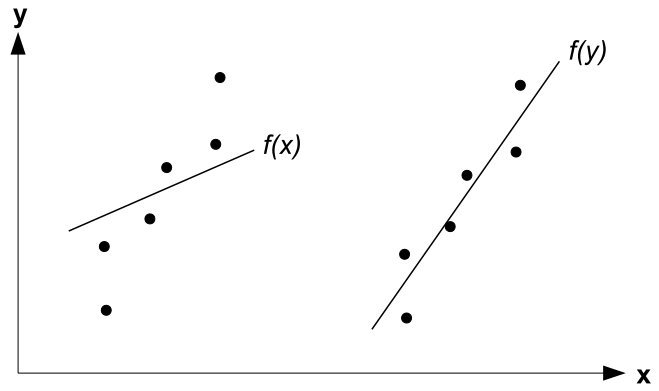

Abbildung 3.9: Approximationsgerade entlang der X-Achse (links) oder der Y-Achse (rechts).

3.2.2 Globale Karte erstellen

In dieser Diplomarbeit beschränken wir uns auf einfache Polygone. Aus diesem Grund kann für die Darstellung einer polygonalen Karte auch eine Liste mit den Liniensegmenten gewählt werden. Die Segmente in der Liste sind dabei entsprechend dem Polygonrand sortiert. Der Abstand zwischen zwei aufeinander folgenden Segmenten entspricht dabei dem unerforschten Teil des Polygons.

Definition 14 Sie L eine Liste mit Liniensegmenten, die entlang des Polygonrandes sortiert sind. Ist der Abstand zwischen zwei aufeinander folgenden Segmenten aus der Liste L größer Null, so wird das Liniensegment, welches aus dem Endpunkt des ersten und dem Startpunkt des zweiten Liniensegmentes gebildet wild, als **unsichtbares Liniensegment** bezeichnet.

Bei einem komplett erkundetem Polygon ist der Abstand zwischen jeweils zwei aufeinander folgenden Liniensegmenten gleich Null und es existieren keine unsichtbaren Liniensegmente.

Nach dem ersten Laserscan besteht die Karte aus einem Sichtbarkeitspolygon, welches entsprechend Kapitel 3.2.1 erstellt wurde. Bei weiteren Laserscans müssen neue Sichtbarkeitspolygone mit der bestehenden Karte vereinigt werden. Wir gehen zunächst davon aus, dass der Odometriefehler gleich Null ist. Trotzdem werden wir nie zwei identische Liniensegmente in unterschiedlichen Laserscans haben. Das liegt zum einen an der fehlerbehafteten Laserabtastung oder Unebenheiten in dem Hindernis (z.B. Betonwand, Pappe, usw.), und zum anderen an der Bewegung des Roboters, bei der Kanten sichtbar werden, die durch eine reflexe Ecke verdeckt waren. Durch den 180° Scanbereich des Lasers werden die Liniensegmente beim Fah-

ren „gekürzt“. Aus diesen Gründen werden die Kanten aus der Karte und dem neuen Sichtbarkeitspolygon, die das gleiche Hindernis repräsentieren, nach der *Gaußschen Methode der kleinsten Quadrate*[3] vereinigt. Das hat zur Folge, dass die aktuell sichtbaren Liniensegmente in der Karte ständig aktualisiert werden.

Bevor die Liniensegmente vereinigt werden können, muss zu jedem Liniensegment aus dem Sichtbarkeitspolygon das entsprechende Liniensegment aus der Karte bestimmt werden. Das Verfahren zur Bestimmung der Liniensegmente muss einen geringen Aufwand haben, damit es online ausgeführt werden kann. Zusätzlich muss dieses Verfahren weniger empfindlich gegenüber den Odometriefehlern sein. Aufgrund der begrenzten Sicht kann nicht vorausgesetzt werden, dass immer eine Ecke existiert, an der sich der Roboter orientieren kann, wie z.B. in einem längeren Gang. Die Suche nach einer Kante, die den kleinsten Abstand hat, führt schon bei kleineren Odomentriefehlern zu einem Misserfolg.

Basierend auf den oben genannten Rahmenbedingungen wurde ein Verfahren entwickelt, welches zwar gegenüber den großen Odometriefehlern nicht robust ist, dafür aber einen geringen Aufwand zur Ausführung braucht. Für jedes Liniensegment L aus dem Sichtbarkeitspolygon wird ein Strahl S erzeugt, dessen Ursprung die Position des Roboters ist und durch die Mitte von L geht, wie in Abbildung 3.10 dargestellt. Nun wird das Liniensegment L_s aus der Karte bestimmt, das von S geschnitten wird und dessen Schnittpunkt am nächsten zum Ursprung des Strahls ist. Ist L_s ein unsichtbares Liniensegment, so wird L in die Karte übernommen, anderenfalls werden L und L_s nach der Gaußschen Methode vereinigt.

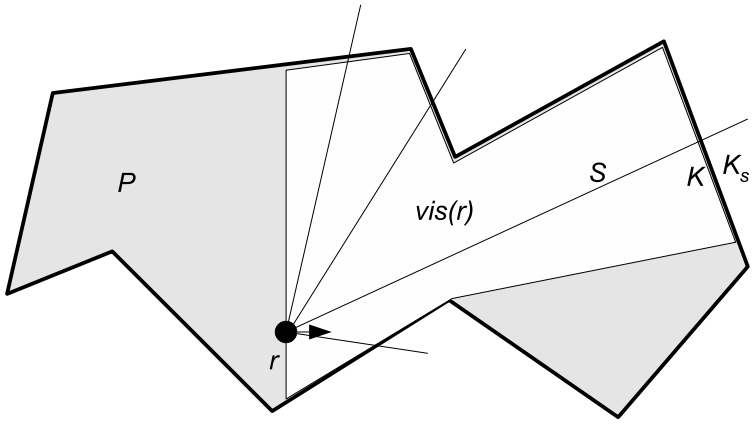

Abbildung 3.10: Das Identifizieren der neuen Liniensegmente in der Karte; Sicht des Roboters: 180°.

[3] siehe Kapitel 3.2.1

Nach einer solchen Aktualisierung kann es vorkommen, dass die aktualisierten Liniensegmente mit den benachbarten Liniensegmenten *echte* Schnitte haben. Bei einem **echten** Schnitt sind die Endpunkte eines Liniensegmentes ungleich dem Schnittpunkt. Diese können in einem Durchlauf entfernt werden, indem die Liniensegmente auf den Schnittpunkt gekürzt werden.

Um die Kantenanzahl des Polygons zu reduzieren, wird anschließend überprüft, ob mehrere Kanten durch eine approximiert werden können. Dies ist dann der Fall, wenn alle Endpunkte der zu approximierenden Geraden im Toleranzbereich der Approximationsgeraden liegen, siehe Abbildung 3.11. In diesem Beispiel werden die Kanten k_2, k_3 und k_4 zu einer neuen Geraden approximiert. Der Toleranzbereich beträgt zur Zeit das dreifache des Schwellwertes bei der Erstellung des Sichtbarkeitspolygons im Kapitel 3.2.1. Experimente haben gezeigt, dass bei einem Toleranzbereich, welcher sehr nahe am Schwellwert lieg, eine viel größere Anzahl an Kanten in der Karte entstehen.

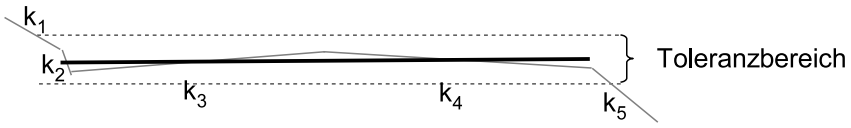

Abbildung 3.11: Approximation mehrerer Polygonkanten zu einer Kante.

Kommen nun Odometriefehler hinzu, so müssen diese korrigiert werden, bevor die neuen Liniensegmente in die Karte eingepflegt werden können. Ohne Fehlerkorrektur wird die Karte sehr schnell unbrauchbar.

An dieser Stelle soll noch erwähnt werden, dass dieses Verfahren während der Diplomarbeit entwickelt wurde. Das Ziel bestand darin, ein einfaches und schnelles Verfahren zu entwickeln. Während der Entwicklung wurde versucht, die Anzahl der Strahlen zu vergrößern. Dies führte zwar zu etwas besseren Resultaten, hatte aber den Nachteil, dass die Rechenleistung des eingesetzten Rechners bei einer größeren Anzahl an Kanten nicht ausreichte, um die Berechnung zur Laufzeit durchzuführen. Auch die Identifizierung der Kanten anhand der Abstandes führte zu keinen guten Ergebnissen, da dadurch viele fehlerhafte Identifizierungen entstanden. Da eine Entwicklung bzw. Implementierung eines robusten Verfahrens den Rahmen dieser Diplomarbeit sprengen würde, wurde das oben beschriebene Verfahren für die Kartenerstellung während der Evaluation der PolyExplore Strategie eingesetzt.

Für das aktuelle Verfahren bei der Kartenerstellung wurden Parameter ausgewählt, wie diese in der Tabelle 3.3 aufgeführt sind.

Beschreibung	**Wert**
Schwellwert für die Approximation der Geraden	150 mm

Tabelle 3.3: Parameter für die Kartenerstellung.

3.2.3 Fehlerkorrektur und Lokalisierung

Für die Fehlerkorrektur wird versucht, die neuen Liniensegmente in der Karte nach dem gleichen Prinzip wie in Kapitel 3.2.2 zu identifizieren. Bei größeren Odometriefehlern kann es zu fehlerhaften Identifizierungen kommen. In Abbildung 3.12 werden zwei Beispiele einer fehlerhaften Identifizierung gezeigt. Eine fehlerhafte Identifizierung kann mit Hilfe des Toleranzbereiches erkannt werden (z.B. ist der Winkel oder Abstand zwischen dem alten und neuen Liniensegment zu groß, so kann es daran liegen, dass das neue Liniensegment falsch zugeordnet wurde).

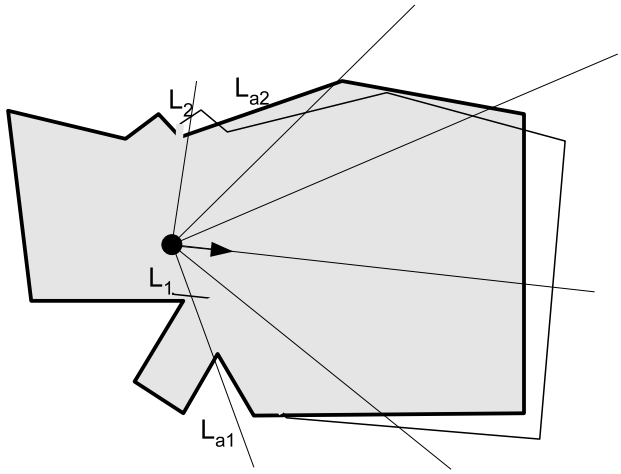

Abbildung 3.12: Fehlgeschlagene Identifizierung der Liniensegmente aufgrund der fehlerhaften Odometrie der Roboters; L_1 mit L_{a1} erkannte (zu große Winkeldifferenz) und L_2 mit L_{a2} unerkannte Fehlidentifizierung.

Leider kann man den Toleranzbereich für die Erkennung solche Fehlidentifizierungen nicht beliebig klein machen. Während der Experimente wurde festgestellt, dass besonders am Anfang einer Drehung ein sehr großer Fehler bei dem Winkel entstand. Dieser Wert hängt stark von der maximalen Drehgeschwindigkeit des Roboters ab. So liegt zur Zeit der Toleranzbereich bei *25*° bei einer maximalen Drehgeschwindigkeit von *20* $\frac{grad}{sec}$. Aus diesem Grund werden weitere Verfahren gebraucht, um Fehlidentifizierungen festzustellen.

Wird eine Fehlidentifizierung aufgrund des Toleranzbereiches erkannt, so wird das Liniensegment bei der Fehlerberechnung ignoriert. Für jedes andere

Liniensegment L_{neu}, welches erfolgreich identifiziert wurde, wird der Fehler zu dem Liniensegment L_{alt} in der Karte berechnet. Der Fehler besteht aus der Winkelabweichung α_f und der Verschiebung (x_f, y_f). Die Abbildung 3.13 verdeutlicht die Fehlerberechnung eines einzelnen Liniensegmentes.

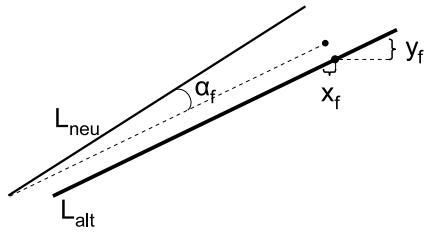

Abbildung 3.13: Fehlerberechnung eines einzelnen Liniensegmentes.

Der Winkel α_f berechnet sich aus der Winkeldifferenz zwischen den beiden Liniensegmenten. Um die Verschiebung zu berechnen, wird L_{neu} um α_f gedreht, anschließend wird das Lot auf L_{alt} durch den Startpunkt s von L_{neu} bestimmt. Die Verschiebung stellt den Fehler (x_f, y_f) dar.

Unter den berechneten Fehlern können sich auch größere Fehler befinden, die durch Fehlidentifizierungen entstanden sind und durch den zu großen Toleranzbereich unerkannt blieben. Diese Ausreißer können nun mit Hilfe der Standardabweichung über die gesamten Fehler gefiltert werden.

Dazu wird für jeden der drei Fehlerarten α_f, x_f, y_f der Mittelwert

$$\bar{f} = \frac{1}{N}\sum_{i=1}^{N} f_i$$

und die Standardabweichung

$$\sigma_f = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(f_i - \bar{f})^2}$$

mit $f \in \alpha_f, x_f, y_f$ bestimmt.

Nun werden alle Fehler, die nicht im Bereich $\bar{f} \pm \sigma_f$ liegen, ausgefiltert und die Mittelwerte über die restliche Menge gebildet. Die berechneten Fehler werden nun dazu genutzt, um die Position des Roboters, sowie alle neuen Liniensegmente zu korrigieren, bevor diese in die Karte eingepflegt werden.

Dieses Verfahren setzt voraus, dass es immer genügend Liniensegmente gibt, die richtig identifiziert wurden. Ist das nicht der Fall, so kann auch nach der Korrektur nicht ausgeschlossen werden, dass eine fehlerhafte Identifizierung, wie z.B. im Fall Abbildung 3.12, unerkannt bleibt. Dies führt in den

meisten Fällen dazu, dass ein neues Liniensegment entsteht oder ein schon bestehendes falsch aktualisiert wird, was bei folgenden Fehlerberechnung für Abweichungen sorgt.

Ein weiteres Problem bei der Fehlerberechnung ist die Approximation der Polygonkanten. Die Approximation wird eingesetzt, um die Anzahl der Kanten zu reduzieren. Ohne Approximation würde es zu einer Vielzahl an kurzen Kanten kommen, was dazu führt, dass schon kleine Fehler zu fehlerhaften Identifizierung führen können. In beiden Fällen führt das zu Abweichungen bei der Fehlerkorrektur. Das hat zur Folge, dass das resultierende Polygon am Ende Abweichungen zum Original aufweist, wie man in Kapitel 4.2 sieht.

Durch die Korrekturen der Position des Roboters kann es dazu kommen, dass in dem korrigierten Weg Sprünge entstehen können. Diese entstehen oft dann, wenn mehrere schon zuvor entdeckte Kanten wieder in die Sicht des Roboters kommen. Dadurch kann es zu einer größeren Korrektur kommen, was dann in den Sprüngen resultiert. Das führt dann dazu, dass die Länge des Weges sich von der tatsächlichen Länge unterscheiden kann. Die von der Odometrie berechnete Länge kann sich natürlich auch von der tatsächlichen Länge unterscheiden, siehe Kapitel 3.1.2.

Das hier vorgestellte Verfahren zur Fehlerkorrektur ist eine „einfache“ Methode, die grobe Fehler in der Odometrie, die besonders bei einer Drehung entstehen, korrigieren soll. Alternative Verfahren werden in [11], [23] vorgestellt.

Für das aktuelle Verfahren bei der Fehlerkorrektur wurden Parameter ausgewählt, wie diese in der Tabelle 3.4 aufgeführt sind.

Beschreibung	**Wert**
Maximaler Winkelfehler	25°
Maximaler Fehler für den Abstand	300 mm

Tabelle 3.4: Parameter für die Fehlerkorrektur.

3.3 PolyExplore

Die PolyExplore Strategie besteht aus drei Phasen, die rekursiv aufeinander folgen.

- Phase 1: Erkunde die Gruppe von rechten Ecken
- Phase 2: Erkunde die Gruppe von linken Ecken
- Phase 3: Fahre zum nächsten *StagePoint*

Damit die Strategie richtig funktioniert ist die richtige Auswahl der Ecken in jeder Phase entscheidend. Bevor wir darauf näher eingehen, betrachten wir zuerst die Erkundung einer Ecke mit einem realen Roboter.

Aufgrund der Ausdehnung des realen Robotersystems kann die zu erforschende Ecke des Polygons nicht direkt angefahren werden. Zuerst muss der Bereich, in dem der Roboter sich störungsfrei bewegen kann, bestimmt werden. Hier ist die Idee von Lozano-Pérez [24] sehr nützlich. Man legt zuerst um den Roboter einen minimalen Kreis. Anschließend zieht man den Mittelpunkt des Kreises entlang des von dem Sensor des Roboters erstellten Polygons. Dadurch entsteht im Inneren des Polygons ein Bereich, der von dem Kreis nicht berührt wird. Dieser Bereich wird als das *Begrenzungspolygon* bezeichnet und ist in der Abbildung 3.14 dargestellt.

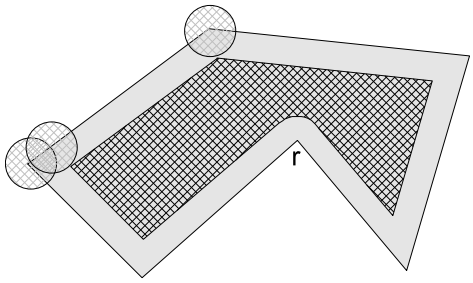

Abbildung 3.14: Das Begrenzungspolygon.

In Abbildung 3.14 sieht man, dass durch reflexe Ecken des Polygons ein Halbkreis in dem Begrenzungspolygon entsteht. Um diese Ecken zu erforschen, muss von der Position des Roboters eine Tangente an den Halbkreis der reflexen Ecke gelegt werden, siehe Abbildung 3.15. Der Punkt, den diese Tangente berührt, wird als Referenzpunkt genommen, um den Pfad zur Erkundung des Ecke zu berechnen. Am Anfang der Erkundung ist das die Tangente t_1, und w_1 ist der berechnete Weg zur Erkundung der Ecke v. Ändert sich die Position des Roboters, muss die Tangente neu angelegt und der Erkundungsweg neu berechnet werden. Das ist beispielhaft durch die Tangenten t_2 und t_3 gezeigt.

In der Implementierung wird der Halbkreis um die Ecke v durch einen Polygonzug wie in Abbildung 3.16 approximiert. Dadurch kann man das gesamte Begrenzungspolygon wie folgt berechnen:

Für jede Polygonkante wird eine Begrenzungslinie wie in Abbildung 3.16 berechnet, die um $2d$ länger und um d in das Polygoninnere verschoben ist. Anschließend werden jeweils zwei benachbarte Begrenzungslinien auf Schnitt überprüft und falls ein Schnittpunkt existiert, werden die beiden auf den Schnittpunkt gekürzt. Existiert kein Schnittpunkt, so liegt das entweder daran, dass der Bereich zwischen zwei Kanten noch nicht erforscht wurde oder dass die beiden Kanten an einer reflexen Ecke v liegen, die einen

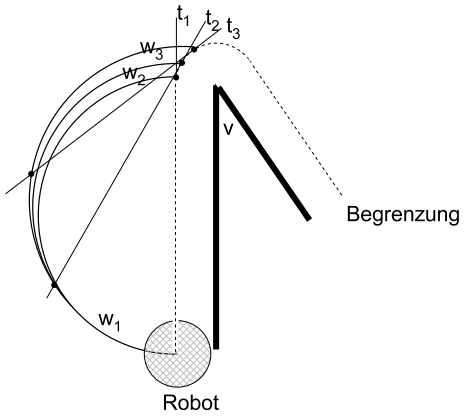

Abbildung 3.15: Erforschen einer reflexen Ecke durch einen Roboter mit Ausdehnung.

Innenwinkel über *270°* hat. In diesem Fall wird die Lücke zwischen den beiden Begrenzungslinien k_1 und k_2 durch ein neues Liniensegment (v_1, v_2) geschlossen.

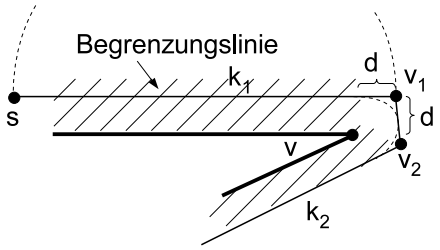

Abbildung 3.16: Berechnung der Begrenzungslinie für das Begrenzungspolygon.

Die Erkundungsbahn einer Ecke wird durch $arc(s, v_1)$ bestimmt und ändert sich während der Erkundung nicht. Sollte der Innenwinkel der Ecke wie in Abbildung 3.16 größer als *270°* sein, so wird der Roboter die Ecke v im Punkt v_1 nicht einsehen können. In diesem Fall muss der Roboter noch zu v_2 fahren, um die Ecke komplett einsehen zu können. Der zusätzliche Weg beträgt maximal $2d$.

Nach jeder Aktualisierung der Karte können im Polygon neue Ecken hinzukommen, alte sich verschieben oder sogar wegfallen. In diesen Fällen müssten komplizierte Aktualisierungen des Begrenzungspolygons und der beiden *Target* und *Todo*-Listen erfolgen. Aus diesem Grund wird in der realen Implementierung von PolyExplore das Begrenzungspolygon und die beiden Listen nach jeder Aktualisierung der Karte neu berechnet.

In einem Begrenzungspolygon können noch unerforschte Ecken leicht gefunden werden. Die Abbildung 3.17 zeigt ein nicht komplett erforschtes Polygon mit seinem Begrenzungspolygon. Um die unerforschte Ecken zu finden, werden zwei aufeinander folgende Kanten k_1 und k_2, die keinen Schnittpunkt haben, betrachtet. Da das Begrenzungspolygon gegen den Uhrzeigersinn orientiert ist, stellt der Endpunkt von k_1 die rechte und der Startpunkt von k_2 die linke unerforschte Ecke dar.

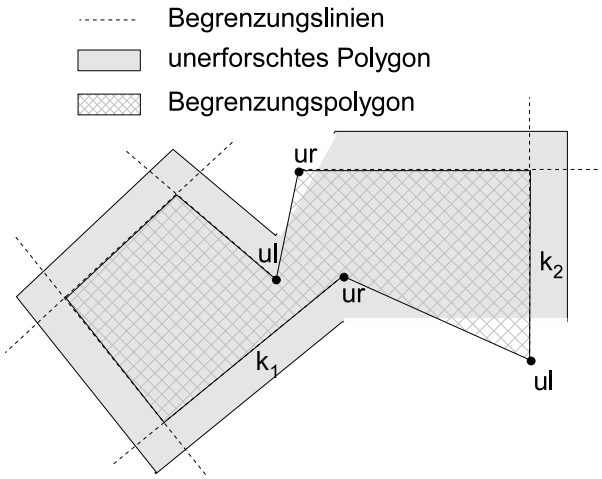

Abbildung 3.17: Das Begrenzungspolygon.

Mit Hilfe des *Shortest Path Tree* für das Begrenzungspolygon können die unerforschten Ecken der *Target* bzw. *Todo*-Liste zugeordnet werden. Befindet sich der Roboter in der ersten Phase, so werden nur die unerforschten rechten Ecken ausgewählt, die von dem aktuellen *StagePoint* direkt gesehen werden oder hinter einer rechten Ecke im *SPT* liegen. Entsprechend werden die linken unerforschten Ecken für die zweite Phase ausgewählt. Existieren nach den ersten beiden Phasen noch weitere unerforschte Ecken, die hinter dem *StagePoint* im *SPT* liegen, so wird der nächste *StagePoint* ausgewählt. Ist der Roboter am neuen *StagePoint* angekommen, so werden die ersten beiden Phasen wiederholt. Liegen hinter dem aktuellen *StagePoint* keine unerforschten Ecken mehr, so kehrt der Roboter zum letzten *StagePoint* zurück und wählt einen neuen *StagePoint*. Dieses Verhalten ergibt sich direkt aus der Rekursion der Strategie. Um diese nachzubilden, werden die einzelnen *StagePoints* auf ein Stack abgelegt. Die Erkundung des Polygons ist beendet, wenn sich auf dem Stack kein *StagePoint* mehr befindet.

Abbildung 3.18 zeigt ein Beispiel für die Auswahl der unerforschten Ecken für die *TargetList*. Die PolyExplore Strategie befindet sich gerade in der ersten Phase bei der Erforschung der rechten Ecke. Da die Ecken ur_3 und ur_4 hinter einer linken Ecke in dem *SPT* liegen, werden nur r_1 und r_2

für die *TargetList* ausgewählt.

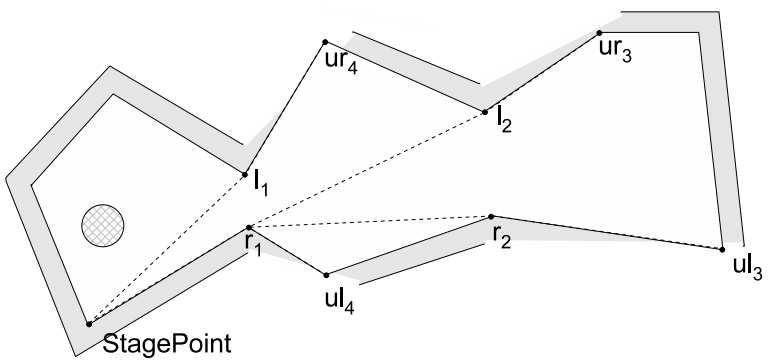

Abbildung 3.18: Unerforschte Ecken in dem Begrenzungspolygon.

Sobald der Roboter mit der Erforschung der rechten Gruppe an den Ecken (r_1 und r_2) fertig ist, kehrt dieser zum *StagePoint* zurück. An dem aktuellen *StagePoint* existieren keine rechten Ecken mehr, die von diesem *StagePoint* für die *TargetList* ausgewählt werden können und die Strategie tritt in die Phase zwei ein. Nun werden die linken Ecken für die *TargetList* ausgewählt.

Nachdem die linke Gruppe erforscht wurde und der Roboter wieder am *StagePoint* ist, existieren keine unerforschten rechten bzw. linken Ecken, die von diesem *StagePoint* erforscht werden können.

An dieser Stelle tritt die Strategie in die dritte Phase ein, in der ein neuer *StagePoint* für die weitere Erkundung ausgewählt und zwar die erste rechte unerforschte Ecke im Uhrzeigersinn. Der Vater dieser Ecke im *SPT* ist der nächste *StagePoint*. Entsprechend werden die linken unerforschten Ecken behandelt.

Die Strategie erkundet jede Ecke auf einem Kreisbogen. In der Implementierung für das reale Robotersystem wird der Kreisbogen durch einen Polygonzug angenähert. Die Länge der einzelnen Kanten des Polygonzuges ist mindestens so lang wie die Länge des Roboters. Diese Methode erleichtert das Steuern des Roboters. Damit bewegt sich der Roboter aber nicht genau auf einer Kreisbahn. Aus diesem Grund kann es passieren, dass sobald eine neue Ecke entdeckt wird und die Erkundung weiter auf einer neuen Kreisbahn verlaufen soll, der Roboter zuerst zu dieser Kreisbahn fahren muss. In diesem Fall fährt der Roboter zu dem nächsten Punkt auf der neuen Kreisbahn.

3.4 Das Konzept der Implementierung

Die PolyExplore Software wurde als eine Experimentierumgebung für das Erkunden von Polygonen entwickelt. Sie funktioniert nach dem Client / Server Prinzip, welches in der Abbildung 3.19 dargestellt ist. Der PolyExplore-Client wird auf einem beliebigen Rechner ausgeführt und ist für die Erstellung der Karte, der Fehlerkorrektur und die Strategieberechnung zuständig. Die Laserdaten bekommt der Client von dem Server, der auf dem Robotersystem läuft. Der Server seinerseits empfängt die Steuerkommandos von dem Client und leitet diese an die Steuerungstreiber des Robotersystems weiter.

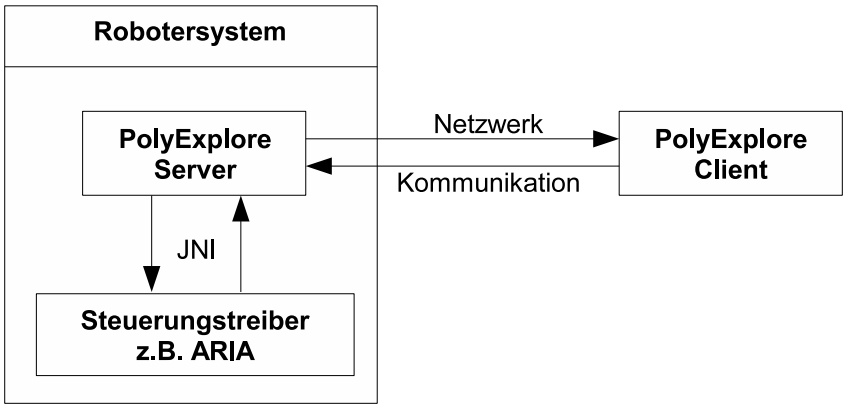

Abbildung 3.19: Server/Client Prinzip der PolyExplore Software.

Dieser Aufteilung macht den PolyExplore Client von dem eingesetzten Robotersystem bzw. den benutzten Treiber unabhängig. Sollten andere Treiber benutzt werden, so reicht es auf der Serverseite die Schnittstelle an die neuen Treiber anzupassen.

Der PolyExplore Client bzw. Server ist in Java implementiert. Als Steuerungstreiber wird ARIA (siehe Kapitel 3.1.3) eingesetzt. Die Kommunikation zwischen PolyExplore Server und ARIA ist über das Java Native Interface (JNI) realisiert, damit der in Java implementierte Server auf die Methoden der in C++ implementierten ARIA-Bibliothek zugreifen kann.

Bei dem Entwurf des Clients wurde besonders Wert darauf gelegt, dass dieser möglichst modular aufgebaut wird. Der Aufbau ist in Abbildung 3.20 abgebildet.

Der Client baut auf dem Roboterinterface auf, welches über das Netzwerk mit dem Server kommuniziert. Die Daten, welche von dem Interface empfangen werden, können zuerst von der Fehlerkorrektur korrigiert und anschließend an das graphische Interface, das Kartenmodul und PolyExplore-Strategie weiter geleitet werden. Die Fehlerkorrektur setzt völlig transparent

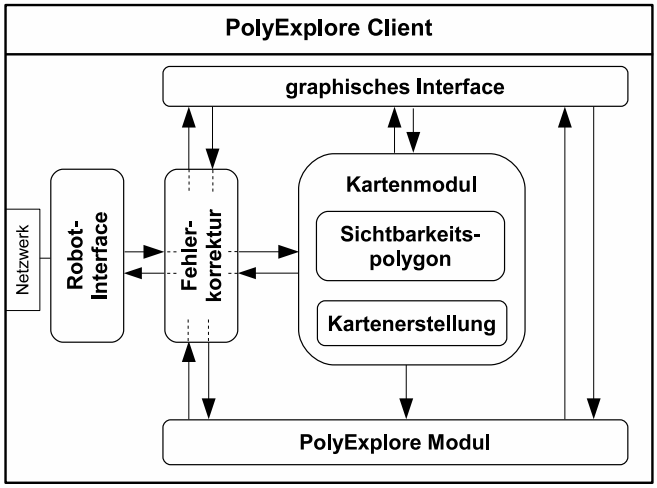

Abbildung 3.20: Modularer Aufbau der Client Software.

auf dem Roboterinterface auf. Für alle Module des Clients sieht es so aus, als würden sie die Daten direkt von dem Roboterinterface empfangen und auch an diesen weiter schicken. Damit lässt sich die Fehlerkorrektur abschalten bzw. auch durch eine andere ersetzen, ohne dass irgendwelche Anpassungen in anderen Modulen notwendig sind.

Das graphische Interface stellt die Daten (wie z.B. Laserdaten, Positionsdaten, das nächste Ziel) graphisch dar. Es ist auch für die Steuerung der Kartenerstellung sowie PolyExplore zuständig und bietet die Möglichkeit, den Roboter direkt zu steuern.

Das Kartenmodul benutzt die Laserdaten zusammen mit den Positionsdaten des Roboters, um die Karte zu aktualisieren. Die aktualisierte Karte bzw. die Position des Roboters werden an das graphische Interface, sowie an PolyExplore weiter gegeben. Die aktuelle Karte wird aber auch von der Fehlerkorrektur benutzt, um zukünftige Odometriefehler zu korrigieren.

Das PolyExplore Modul nutzt die erhaltenen Informationen von dem Kartenmodul, um zusammen mit den Statusmeldungen des Roboters die weitere Erkundung zu planen. Dabei hat es die Möglichkeit den Roboter über das Robotinterface zu steuern.

Durch den modularen Aufbau des Clients ist es möglich, die vorhandene Software um weitere Verfahren (z.B. andere Fehlerkorrekturen, Kartenerstellungen, oder sogar Explorationsstrategien) zu erweitern. Weiterhin besteht die Möglichkeit, die Kommunikation zwischen den Modulen durch eine Netzwerkkommunikation zu ersetzen. Das hat den Vorteil, dass die rechenintensive Prozesse, wie Kartenaktualisierung oder PolyExplore auf getrennten Rechnern laufen können, um die Performance zu steigern.

Zusätzlich lassen sich die einzelnen Parameter über eine Config-Datei

einstellen, siehe Anhang A. Dadurch braucht der Benutzer die Parameter nicht in dem Quellcode zu suchen und spart auch noch das lästige Kompilieren des Projektes.

Kapitel 4

Messung und Bewertung

In diesem Kapitel werden Messungen für einen Vergleich zwischen der theoretischen und praktischen Realisierung der Strategie beschrieben, ausgewertet und dann beurteilt. In Kapitel 4.1 werden die Randbedingungen für die Durchführung der Versuche beschrieben. Anschließend werden die Szenarien in Kapitel 4.2 einzeln vorgestellt und die aus den Versuchen gewonnenen Ergebnisse präsentiert und analysiert. Die daraus gewonnenen Resultate werden in dem folgenden Kapitel 4.3 zusammengefasst.

4.1 Vorbereitung und Durchführung

Die Versuche wurden auf dem Roboter *Pioneer2 AT* (siehe Abbildung 4.1) der Firma ActivMedia durchgeführt. In allen Szenarien betrug die maximale Geschwindigkeit des Roboters 250 mm/sec und die Drehgeschwindigkeit 20 grad/sec.

Zur Überprüfung der implementierten Onlinestrategie wurden Szenarien mit solchen Formen ausgewählt, die möglichst viele Aspekte der Onlinestrategie (Erkundung einer Gruppe von Ecken, rekursives Erkunden usw.) testen sollten. Durch die Ausdehnung des Roboters mussten die Szenarien somit eine Mindestgröße haben. Gleichzeitig wurde die Größe der Szenarien durch die verschiedenen Faktoren beschränkt. Wie sich bei den Vortests herausgestellt hat, stellte der große Odometriefehler das größte Hindernis dar. Dieser führte dazu, dass der Fehler irgendwann nicht mehr von der aktuell implementierten Fehlerkorrektur korrigiert werden konnte, was in einer unbrauchbarer Karte resultierte. Zusätzlich stellen größere Szenarien in der aktuellen Implementierung auch mehr Aufwand bei der Aktualisierung der Karte dar, da für die Aktualisierung nicht nur die lokale Umgebung, sondern die gesamte Karte benutzt wird. Weiterhin ist zu berücksichtigen, dass das Aufbauen und Testen einer realen Szene sehr zeitaufwendig ist. Aufgrund dieser Randbedingungen wurden drei Szenarien ausgewählt, für die reale

Abbildung 4.1: Der Roboter (Pioneer2 AT von ActivMedia) ausgestattet mit einem SICK-Laserscanner.

Messungen und Simulationen gemacht wurden. Drei weitere wurden nur in der Simulation getestet.

Um das Ergebnis bewerten zu können wurden die realen Szenarien vermessen. Daraus wurden Polygone erstellt, die mit den vom Roboter erstellten Polygonen verglichen werden konnten. Außerdem wurde aus dem vermessenen Polygon die Karte für den Simulator erstellt. Damit ließen sich die Ergebnisse der Simulation mit den realen Messungen vergleichen.

Zusätzlich sollte der vom Roboter zurückgelegte Weg analysiert werden. Zu diesem Zweck wurde für jedes vermessene Polygon das Begrenzungspolygon berechnet. Dieses diente zur Berechnung des theoretischen Weges. Der theoretische Weg wurde mit Hilfe des SAM-Applets [3] bestimmt, welches auch die Berechnung des Weges von PolyExplore in einem Polygon erlaubt. Das Applet wurde nur soweit modifiziert, dass die Ein- und Ausgabe von Polygonen bzw. Erkundungswegen möglich wurde. Für jedes Polygon wurde damit der theoretische Weg von PolyExplore sowie die SWR berechnet.

Während der Versuche lagen nach jedem erfolgreichen Durchlauf für die Auswertung folgende Daten vor:

- das erstellte Polygon bzw. Begrenzungspolygon
- der gefahrene Weg
- der korrigierte Weg

Bei dem *gefahrenen* Weg handelt es sich um Weg, der direkt aus den

Odometriedaten erstellt wurde. Die korrigierten Odometriewerte wurden als korrigierter Weg zusammengefasst.

Die Daten wurden in einer *Encapsulated Postscript* [1] (EPS) Datei gespeichert. Anschließend wurden diese über das vermessene Polygon, welches im weiteren als das *Originalpolygon* bezeichnet wird, gelegt. Damit die Zeichnungen übersichtlich bleiben, wird das vom Roboter erstellte Polygon nicht mit in den Zeichnungen dargestellt. Dieses kann leicht von dem Begrenzungspolygon abgeleitet werden.

Nun sollen die einzelnen Szenarien basierend auf den Vermessungen beschrieben werden. Für jedes Szenario wird zusätzlich der theoretische Weg von PolyExplore basierend auf dem Begrenzungspolygon beschrieben.

4.2 Messergebnisse

In diesem Kapitel werden die Versuchsergebnisse aus verschiedenen Szenarien vorgestellt und analysiert. Für einen einfacheren Vergleich zwischen der Theorie und Praxis wird für jedes Szenario zuerst der theoretische Weg beschrieben. Anschließend werden dann für dieses Szenario die Versuchsergebnisse vorgestellt und auf die Abweichungen zu dem theoretischen Weg hingewiesen. Diese Abweichungen werden analysiert und der Grund für das Verhalten erläutert.

Es werden drei Szenarien vorgestellt, für die Messungen in der realen Umgebung sowie in der Simulation gemacht wurden. Aufgrund so gewonnenen Erfahrungen wurden für weitere zwei Szenarien Messungen nur in der Simulation gemacht, die in Kapiteln 4.2.4 und 4.2.5 vorgestellt werden.

4.2.1 Szenario A

Bei dem ersten Szenario soll der Roboter die Strategie zur Erkundung der rechten Ecke zeigen. Die ausgewählte Umgebung und die theoretische Wegstrecke (wird in den Abbildungen als *theo. PolyExplore* bezeichnet) für diese Aufgabe sind in Abbildung 4.2 dargestellt.

Die gesamte Testumgebung ist 14.3 Meter lang und 10.2 Meter breit. Der Roboter startet an der Ecke s und soll im Kreisbogen $arc(s, r_1)$ die rechte Ecke r_1 erforschen. Sobald die Ecke r_2 in Sicht kommt, wird die Erkundung auf dem Kreisbogen $arc(s, r_2)$ weiter fortgesetzt. An der Stelle b versperrt c die Sicht auf r_2. Da r_2 noch nicht erkundet ist, fährt der Roboter direkt auf die Ecke c zu und setzt die Erkundung am Rand des Polygons fort, sobald r_2 wieder sichtbar wird. Der Roboter fährt so lange am Rand entlang, bis er die Erkundung in d auf dem Kreisbogen $arc(s, r_2)$ fortsetzen kann. In e verliert

[1] siehe http://partners.adobe.com/public/developer/ps/index_specs.html

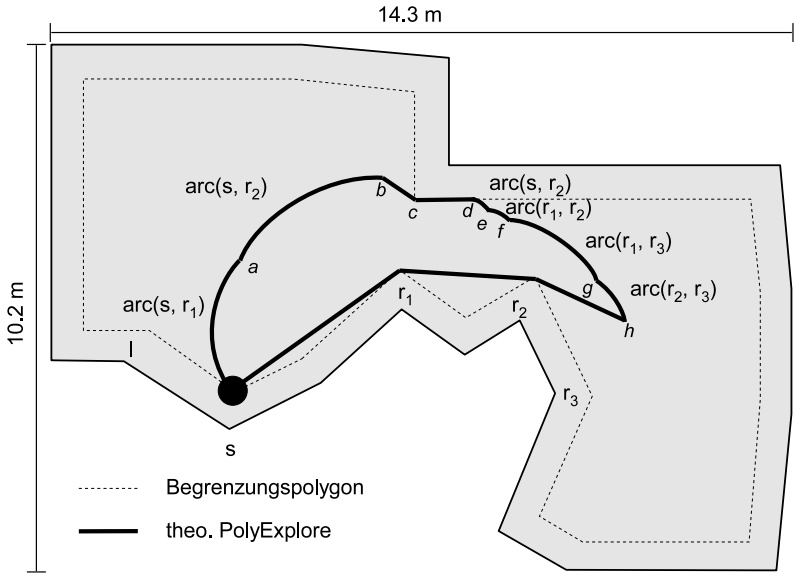

Abbildung 4.2: Szenario A: Erkundung der rechten Ecke.

der Roboter die Sicht auf den Startpunkt, an dieser Stelle muss er die Erkundung auf dem $arc(r_1, r_2)$ fortsetzen. In f wird r_3 sichtbar und die Erkundung geht auf $arc(r_1, r_3)$ weiter. In g verliert der Roboter die Sicht auf r_1. Da r_3 noch nicht erkundet ist, muss er die Erkundung auf $arc(r_2, r_3)$ fortsetzen. In h ist die Erkundung beendet und der Roboter kehrt auf dem kürzesten Weg zum Startpunkt s zurück. Die Länge des zurückgelegten Weges beträgt 20.24 Meter. Die kürzeste Watchman Route für dieses Polygon beträgt 16.88 Meter.

In den Abbildungen 4.3 und 4.4 sind zwei der durchgeführten Messungen für diese Umgebung dargestellt. Bei beiden Messungen kann man Abweichungen in dem vom Roboter erstellten Polygon von der realen Umgebung feststellen. Weiterhin fällt sofort auf, dass der gefahrene und korrigierte Weg sehr stark von einander abweichen, was auf einen großen Odometriefehler hindeutet. Beim ersten groben Vergleich der Erkundungswege in den beiden Abbildungen stellt man fest, dass diese sich unterscheiden, obwohl die gleiche Umgebung nach der gleichen Strategie erforscht wurde. Wodurch das zustande kommt, werden wir später sehen. Zunächst soll der Unterschied zwischen dem theoretisch berechneten und dem korrigierten gefahrenen Pfad untersucht werden.

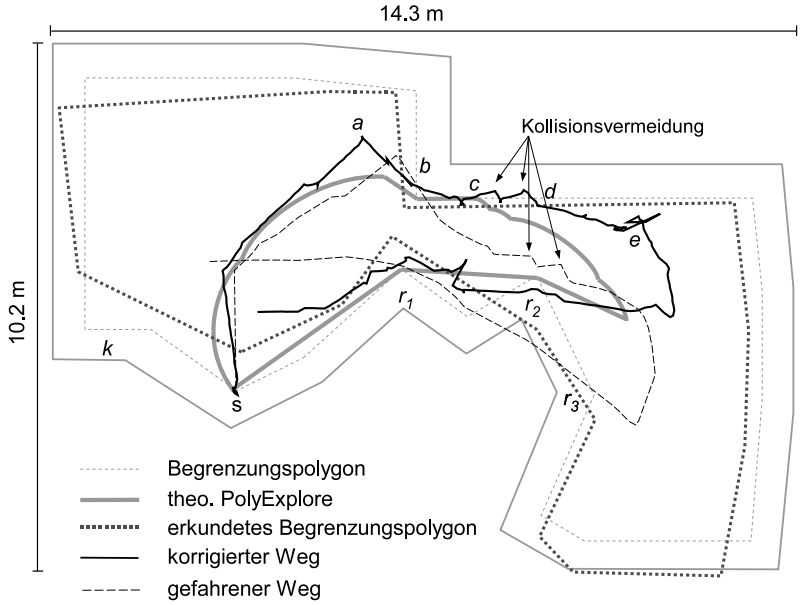

Abbildung 4.3: Szenario A, Versuch 1.

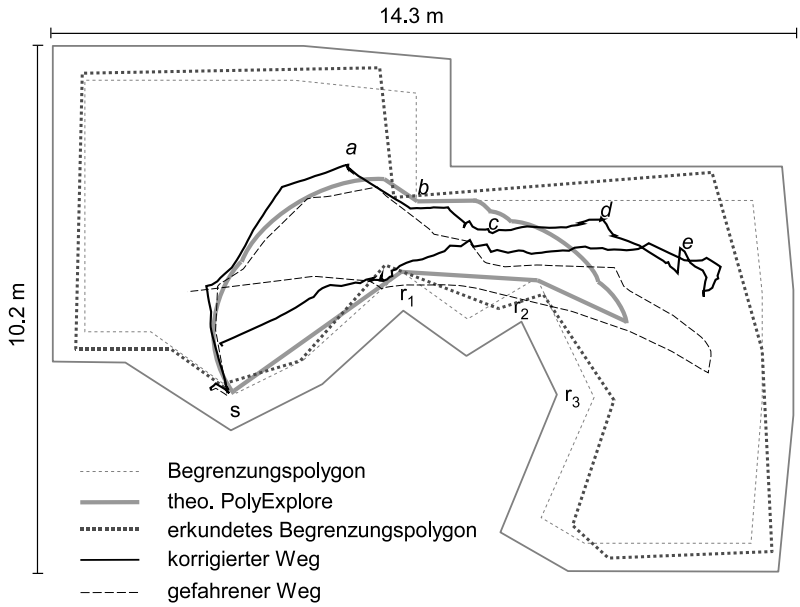

Abbildung 4.4: Szenario A, Versuch 2.

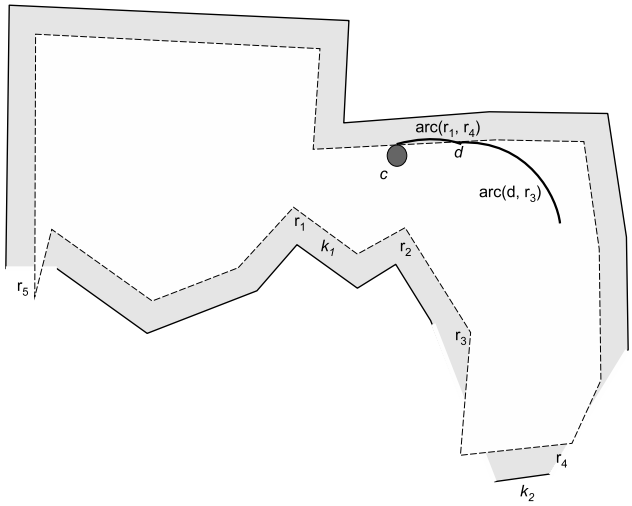

Abbildung 4.5: Szenario A: imaginäre Ecke, die durch die begrenzte Sicht des Roboter entsteht.

Betrachtet man den korrigierten Weg, so findet man Sprünge, welche im nicht korrigierten Weg nicht zu finden sind. Solche Sprünge entstehen, wenn der berechnete Fehler etwas größer ist. Möchte man feststellen, welche Drehungen der Roboter tatsächlich gemacht hat, so muss der nicht korrigierte Weg zusätzlich betrachtet werden.

Weiterhin fällt auf, dass auch der korrigierte Weg sich zu weit außerhalb des Begrenzungspolygons befindet. Dies liegt daran, dass das Polygon während der ganzen Fahrt immer aktualisiert wurde. D.h. die Fehler, die durch die Fehlerkorrektur nicht korrigiert werden, spiegeln sich in der Verschiebung der Polygonkanten wider. Das kann sogar dazu führen, dass Kanten im Laufe der Erkundung durch Approximation wegfallen können, z.B. die Kante k in der Abbildung 4.3.

Nach einem schlechten Start, welcher den ersten Kreisbogen nicht gut approximiert hat, wird der zweite Kreisbogen $arc(s, r_2)$ an Anfang relativ gut angenähert. Der Ausreißer am Punkt a lässt sich auf eine schlechte Fehlerkorrektur an den Kanten zwischen r_1 und r_2 zurückführen. dass der Roboter soweit über die Hindernissecke hinaus fährt, liegt daran, dass zu dem Zeitpunkt diese Ecke des Polygons viel näher an b lag. In b angekommen, folgt der Roboter der Polygonkante bis in c eine neue Ecke entdeckt wird. Diese Ecke ist eine imaginäre Ecke, die in der realen Umgebung nicht vorkommt, aber durch die begrenzte Sicht des Roboters entsteht. Um das Problem besser verstehen zu können, schauen wir uns in der Abbildung 4.5 an, wie das interne Polygon des Roboters zu diesem Zeitpunkt ausgesehen hat.

Von der aktuellen Position c sieht der Roboter nur einen Teil der Kante

k_2. Dadurch scheint es, als ob noch eine rechte Ecke r_4 existiert. So versucht der Roboter auf dem Kreisbogen $arc(r_1, r_4)$ die Ecke zu erforschen. Wie man nun in der Abbildung 4.3 erkennen kann, kommt der Roboter beim Fahren entlang der Polygonkante dieser zu nahe. Für diesen Fall wurde eine Kollisionsvermeidung eingebaut, welche die Aufgabe hat, den Roboter von der Kante auf einen Sicherheitsabstand weg zu fahren. Kurz nach der zweiten Kollisionsvermeidung wurde die Ecke r_4 erforscht. Für die Strategie ging an dieser Stelle die Erkundung der rechten Ecke zu Ende. Die $TargetList$ enthält aber noch r_3, die erforscht werden muss. An dieser Stelle beginnt die Erkundung der nächsten rechten Ecke, die von der aktuellen Position des Roboters ausgeht. Der Roboter bewegt sich nun auf dem Kreisbogen $arc(d, r_3)$. Das erklärt die größere Abweichung in der realen Onlinestrategie von dem theoretischen Weg.

Der Ausschlag in e kann man durch die Probleme in der Fehlerkorrektur erklären, die auftraten, als die Kante k_1 wieder in das Sichtfeld kam.

Vergleicht man nun die Ergebnisse der Versuche 1 und 2 so kann man in den Entscheidungen der PolyExplore Strategie Ähnlichkeiten finden. Der Unterschied in der Weglänge zwischen c und d erklärt sich dadurch, dass der Roboter nicht in dem Zustand der Kollisionsvermeidung war und deswegen der Polygonkante länger folgen konnte, ohne der Kante k_2 näher zu kommen.

dass die Wege in den beiden Versuchen so unterschiedlich aussehen, liegt zum größten Teil an den Odometriefehlern des Roboters bzw. der Fehlerkorrektur.

Untersucht man das Szenario in der Simulation, so stellt man fest, dass der Odometriefehler im Vergleich zum realen Roboter gering ist. Der Erkundungsweg ist fast der berechneten Strecke gleich. dass in der Simulation die Erkundung der r_4 aus der Abbildung 4.5 nicht auftritt, liegt wohl an der Lage der Kante k_2. Wie man beim Vergleich der Abbildungen 4.3 und 4.4 sieht, befindet sich die Kante k_4 an der Grenze von dem Sichtbereich des Roboters. Das kann zum Teil an der Position des Roboters und zum Teil an der Remission der Hindernisse liegen. In der Simulation ist die Sichtweite des Roboters immer gleich. In der Realität (siehe Abbildung 4.15) besteht das Polygon aus Materialien, wie Beton, Kartons, Tischplatten usw., die unterschiedliche Remissionen haben. Das kann dazu führen, dass ein Hindernis früher als ein anderes gesehen wird.

Der zurückgelegte Weg in dieser Umgebung bei unterschiedlichen Versuchen und der dazugehörige kompetitive Faktor ist in der Tabelle 4.1 zusammengefasst.

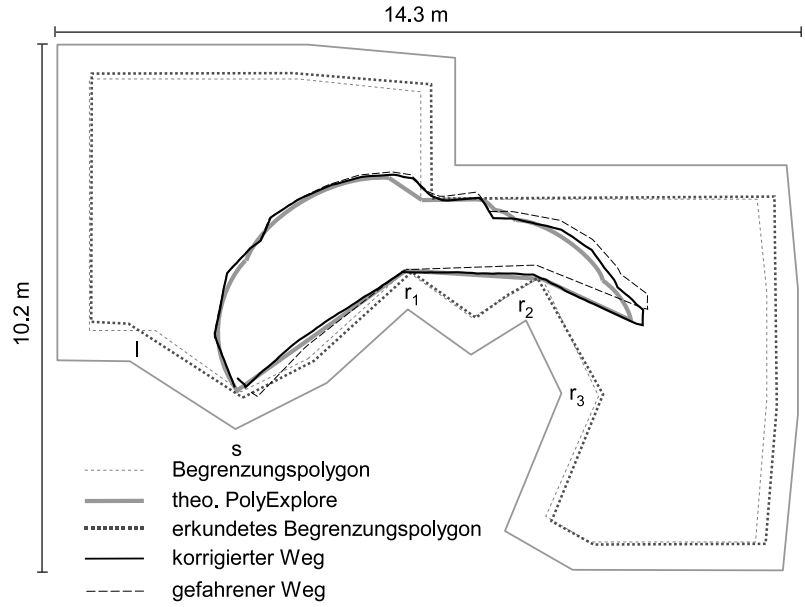

Abbildung 4.6: Szenario A, Simulation.

4.2.2 Szenario B

Nun soll der Roboter linke und rechte Ecken erforschen. Die Herausforderung besteht darin, dass der Roboter zu einem *StagePoint* fahren muss, um die linke Ecke zu erforschen. Die Abbildung 4.7 zeigt die Umgebung, welche für diese Aufgabe aufgebaut wurde.

Die Ausmaße der Testumgebung betragen 13.3 Meter in der Länge und 8.2 Meter in der Breite. Der Roboter startet wieder an der Ecke s und soll zuerst die rechte Ecke r_1 auf dem Kreisbogen $arc(s, r_1)$ erforschen. In a wird die Ecke r_2 entdeckt und die Erkundung wird weiter auf $arc(s, r_2)$ fortsetzt. Die Erkundung der rechten Ecken endet in b und der Roboter kehrt zum Startpunkt s zurück. Von s sind an dieser Stelle keine unerforschte Ecken sichtbar. In der *TodoList* ist allerdings noch l_2, die unerforscht ist. Da r_2 der Vater von l_2 im *SPT* ist, wird r_2 als *StagePoint* ausgewählt. Der Roboter fährt auf dem kürzesten Weg zu r_2. Ab da setzt er die Erkundung auf dem Kreisbogen $arc(r_2, l_2)$ fort. Da aber der Polygonrand im Weg ist, fährt der Roboter am Polygonrand entlang bis schließlich in c die Erkundung des Polygons zu Ende ist und der Roboter auf dem kürzesten Weg zum Startpunkt s zurück fährt. Der zurückgelegte Weg beträgt am Ende 19.98 Meter. *SWR* ist in diesem Beispiel nur 14.25 Meter lang.

Betrachtet man die Ergebnisse der realen Messung in den Abbildungen 4.8 und 4.9, stellt man fest, dass das erstellte Polygon große Abweichungen aufweist. Weiterhin sieht man auch hier, dass die Odometriefehler der

Versuch	SWR	WmF[1]	WoF[2]	KFmF[3]	KFoF[4]
1	16.88 m	27.52 m	23.76 m	1.6303	1.4076
2	16.88 m	28.11 m	24.61 m	1.6653	1.4579
SIM[5]	16.88 m	21.12 m	21.00 m	1.2512	1.2441

[1] WmF = Weg mit Fehlerkorrektur
[2] WoF = Weg ohne Fehlerkorrektur
[3] KFmF = Kompetitiver Faktor mit Fehlerkorrektur
[4] KFoF = Kompetitiver Faktor ohne Fehlerkorrektur
[5] SIM = Simulation

Tabelle 4.1: Szenario A: kompetitiver Faktor der einzelnen Versuche (theoretischer Wert: 1.1989).

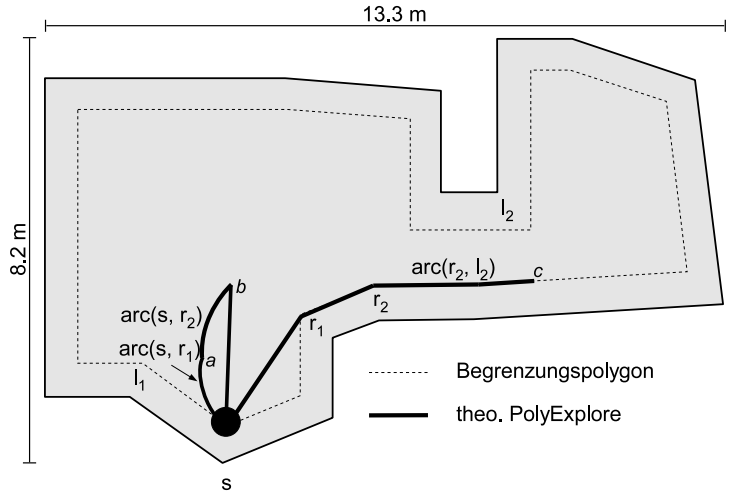

Abbildung 4.7: Szenario B: Rekursives Erkunden.

Fehlerkorrektur Probleme bereitet hat. Die Tatsache, dass im Laufe der Erkundung die Kante e weggefallen ist, hatte den Fehler noch vergrößert.

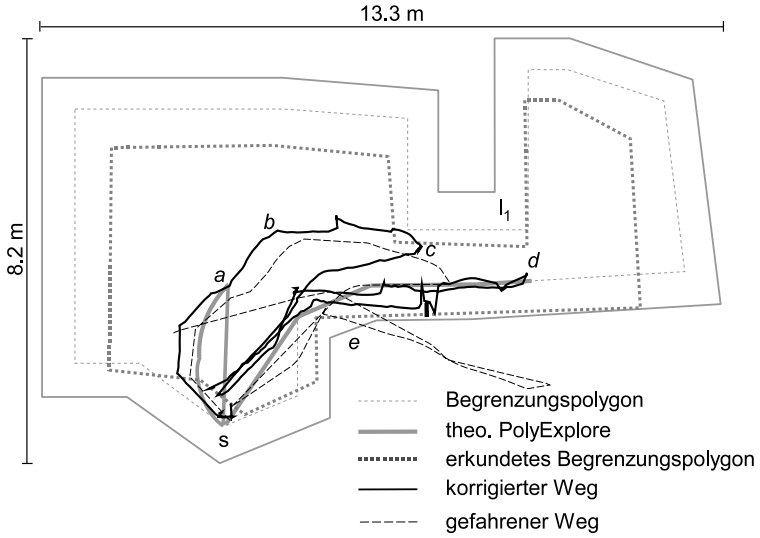

Abbildung 4.8: Szenario B, Versuch 1.

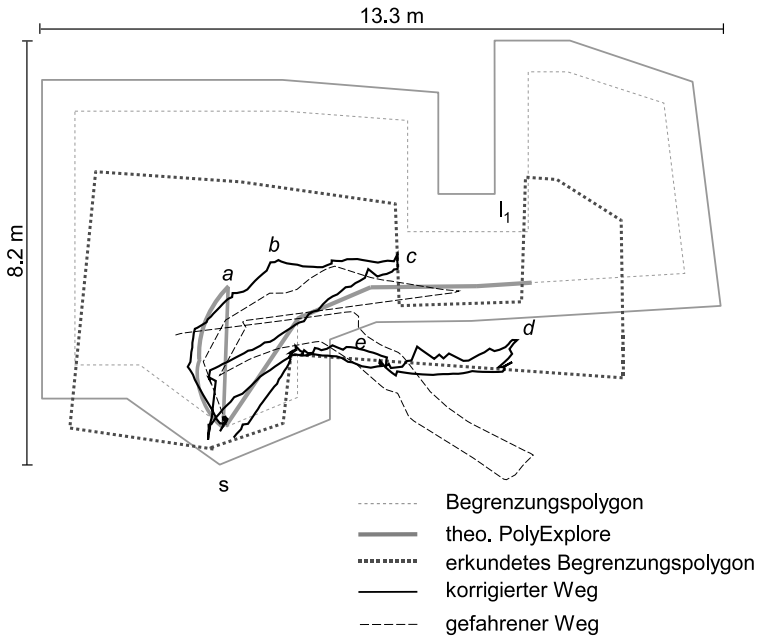

Abbildung 4.9: Szenario B, Versuch 2.

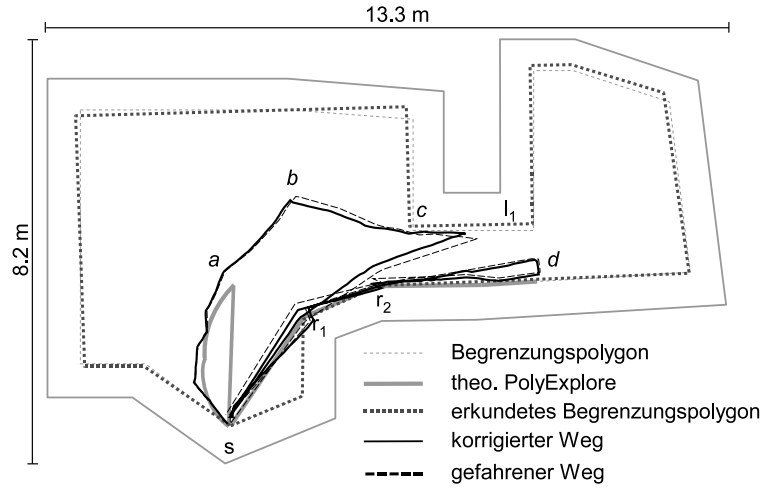

Abbildung 4.10: Szenario B, Simulation.

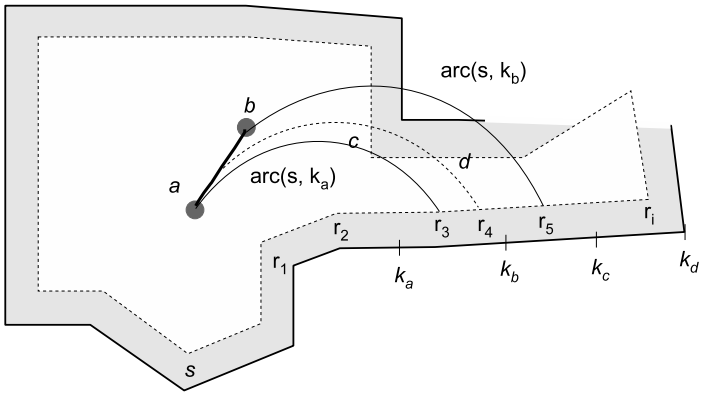

Abbildung 4.11: Erkundung eines Flures.

Vergleicht man die realen Messungen mit den Ergebnissen der Simulation in der Abbildung 4.10, so stellt man fest, dass der Weg, abgesehen von den Odometriefehlern, recht ähnlich aussieht. Für die Analyse der Ergebnisse, beschränken wir uns daher auf die Simulation.

Nach dem Start der Erkundung sieht man die Annäherung des Polygonweges an die Kreisbögen. Die Annäherung ist in allen drei Versuchen nicht besonders gut, was zum Teil an der Größe und dem Größenunterschied der beiden Kreisbögen liegt. An der Stelle a ist die Ecke r_2 erkundet und in der Theorie kehrt der Roboter zum Startpunkt zurück. In der Realität beschreibt der Roboter fast geradlinige Bewegung zum Punkt b. Der Grund für diese Bewegung ist die begrenzte Sicht des Roboters. Die Abbildung 4.11 soll dieses Verhalten verdeutlichen.

Versuch	SWR	WmF[1]	WoF[2]	KFmF[3]	KFoF[4]
1	14.26 m	33.19 m	29.73 m	2.3275	2.0849
2	14.26 m	32.61 m	28.95 m	2.2868	2.0302
SIM[5]	14.26 m	30.78 m	31.41 m	2.1584	2.2027

[1] WmF = Weg mit Fehlerkorrektur
[2] WoF = Weg ohne Fehlerkorrektur
[3] KFmF = Kompetitiver Faktor mit Fehlerkorrektur
[4] KFoF = Kompetitiver Faktor ohne Fehlerkorrektur
[5] SIM = Simulation

Tabelle 4.2: Szenario B: kompetitiver Faktor der einzelnen Versuche (theoretischer Wert: 1.4014).

Wenn der Roboter im Punkt a die Ecke r_2 erforscht hat, reicht seine Sicht, um die Polygonkante k bis zu Stelle k_a zu sehen. Für den Roboter sieht es so aus, als wäre r_3 eine unerforschte rechte Ecke. Also setzt er die Erkundung auf dem Kreisbogen $arc(s, r_3)$ fort. Dadurch kommt er r_3 etwas näher und kann in den Flur weiter einsehen, wodurch r_3 sich nach r_4 verschiebt. Die Erkundung läuft dann auf dem Kreisbogen $arc(s, r_4)$ weiter. Je weiter der Roboter einsehen kann, desto größer wird der Kreisbogen, auf dem sich der Roboter bewegt. Aus diesem Grund sieht der Weg zwischen a und b wie eine Gerade aus. Der Roboter setzt die Erkundung der Kante k solange fort, bis er diese komplett erkundet hat. Anschließend kehrt der Roboter zu s zurück.

Wenn er wieder am Startpunkt ankommt, muss er sich zum nächsten *StagePoint* bei r_2 begeben, um von da aus die linke Ecke l_1 zu erkunden. In d macht der Roboter noch einen Ansatz eines Kreisbogens, da er sonst nicht die komplette Kante hinter l_1 sehen konnte, und beendete die Erkundung dieses Polygons.

Der zurückgelegte Weg in dieser Umgebung bei unterschiedlichen Versuchen und der dazugehörige kompetitive Faktor ist in der Tabelle 4.2 zusammengefasst.

4.2.3 Szenario C

Bei diesem Szenario soll der Roboter eine Gruppe von rechten Ecken erforschen. Zusätzlich soll wie beim letzten Szenario eine linke Ecke von einem weiteren *StagePoint* erkundet werden. Die Testumgebung ist in der Abbildung 4.12 dargestellt.

Die Ausmaße der Testumgebung betragen 7.8 Meter in der Länge und 7.6 Meter in der Breite. Der Roboter startet wieder an der Ecke s. Die Gruppe besteht am Anfang aus zwei rechten Ecken r_1 und r_2. Zuerst wird die rechte Ecke r_1 erkundet. Am Anfang fährt der Roboter am Polygonrand

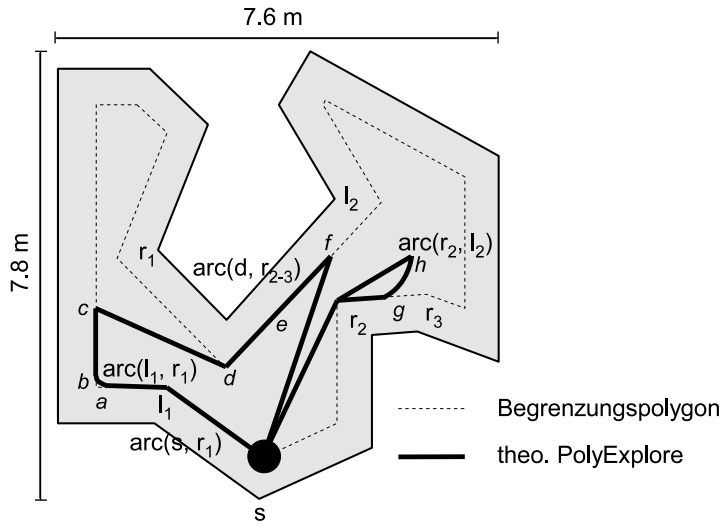

Abbildung 4.12: Szenario C: Erkundung der rechten Gruppe.

bis l_1 entlang, da dieser den Kreisbogen $arc(s, r_1)$ begrenzt. Ab da versucht der Roboter auf dem Kreisbogen $arc(l_1, r_1)$ zu fahren. Der größte Teil des Weges führt am Polygonrand entlang bis in c die Ecke r_1 erforscht ist. Da die $TargetList$ zu diesem Zeitpunkt noch r_2 enthält, aber diese von der aktuellen Position nicht sichtbar ist, fährt der Roboter auf dem kürzesten Weg auf r_2 zu. In d wird r_2 sichtbar. Hier fängt die Erkundung der zweiten rechten Ecke aus der Gruppe auf dem Kreisbogen $arc(d, r_2)$ an. Der Weg führt wieder entlang des Polygonrandes. In f ist r_3 und damit auch die Gruppe erkundet. Der Roboter kehrt zum Startpunkt zurück und fährt zum nächsten $StagePoint$ r_2, um die linke Ecke l_2 zu erforschen. In h ist das Polygon komplett erkundet und der Roboter kehrt zum Startpunkt s zurück. Der zurückgelegte Weg beträgt am Ende 22.52 Meter. SWR ist in diesem Beispiel nur 14.68 Meter.

Bei diesem Szenario bereitete der Odometriefehler besonders am Ende der Fehlerkorrektur Probleme. Dies kann man an dem Weg am Startpunkt in der Abbildung 4.13 sehr gut erkennen. Bei diesem Szenario war der Freiraum für den Roboter sehr begrenzt. Und obwohl es immer genug Kanten gab, an denen sich der Roboter orientieren konnte, kam die Fehlerkorrektur trotzdem nicht zurecht. Dies lässt die Vermutung nahe, dass je öfter der Roboter in der gleichen Umgebung hin und her fährt, desto größere Abweichungen haben die Kanten nach ihrer Aktualisierung. Das führt in weiteren Korrekturen zu noch größeren Fehlern.

Dass die Strategie in diesem Szenario dennoch funktioniert, konnte man in der Simulation, Abbildung 4.14, nachvollziehen. Die einzige größere Abweichung zu dem theoretischen Weg konnte man an der Stelle e feststellen.

Diese entstand durch die wenige Approximationspunkte des Kreisbogens, der in diesem Fall entlang der Polygonkanten verlief. Aus diesem Grund kam der Roboter erst später in den Bereich, in dem er die Sicht auf den Startpunkt verlor und den Kreisbogen $arc(l_1, r_1)$ fahren konnte.

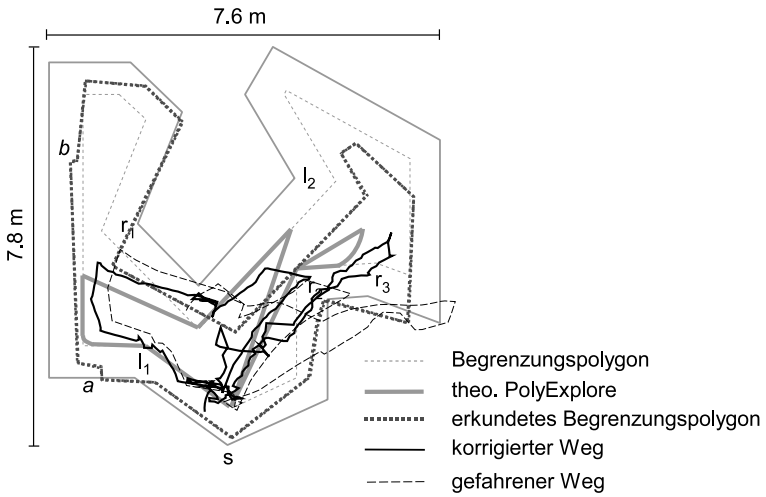

Abbildung 4.13: Szenario C, Versuch 1.

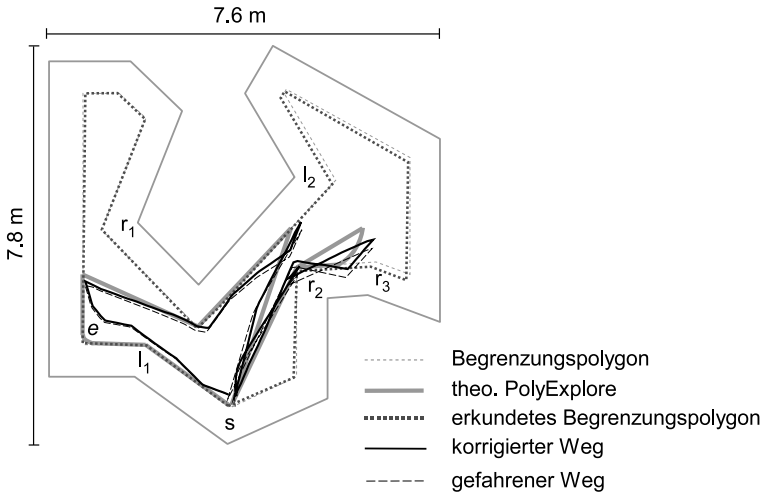

Abbildung 4.14: Szenario C, Simulation.

Abbildung 4.15: Szenario C, Aufbau vom realen Szenario.

Versuch	SWR	WmF[1]	WoF[2]	KFmF[3]	KFoF[4]
1	14.68 m	32.35 m	26.34 m	2.2037	1.7943
SIM[5]	14.68 m	22.97 m	22.85 m	1.5647	1.5565

[1] WmF = Weg mit Fehlerkorrektur
[2] WoF = Weg ohne Fehlerkorrektur
[3] KFmF = Kompetitiver Faktor mit Fehlerkorrektur
[4] KFoF = Kompetitiver Faktor ohne Fehlerkorrektur
[5] SIM = Simulation

Tabelle 4.3: Szenario C: kompetitiver Faktor der einzelnen Versuche (theoretischer Wert: 1.5342).

Die Ecken bei der geradlinigen Bewegung des Roboters, entstanden entweder durch die Wegpunkte, wie z.B. am Polygonrand, oder durch die Korrektur beim Anfahren des Zieles.

Der zurückgelegte Weg in dieser Umgebung bei unterschiedlichen Versuchen und der dazugehörige kompetitive Faktor ist in der Tabelle 4.3 zusammengefasst.

4.2.4 Szenario D

Im letzten Szenario wurde festgestellt, dass längere Fahrten in der gleichen Umgebung der aktuell implementierten Fehlerkorrektur Probleme bereiten. Aus diesem Grund werden weitere aufwendige Versuche in der Simulation durchgeführt, da diese nicht so hohe Odometriefehler aufweist.

Dieses Szenario ist mit den Ausmaßen 13.1 m x 10.1 m nicht viel größer als die vorangegangene Szenarien. Allerdings ist es so aufgebaut, dass der Roboter zu mehreren *StagePoints* fahren muss, um das komplette Polygon zu erkunden. In der Abbildung 4.16 ist die Testumgebung mit dem theoretischen Weg abgebildet.

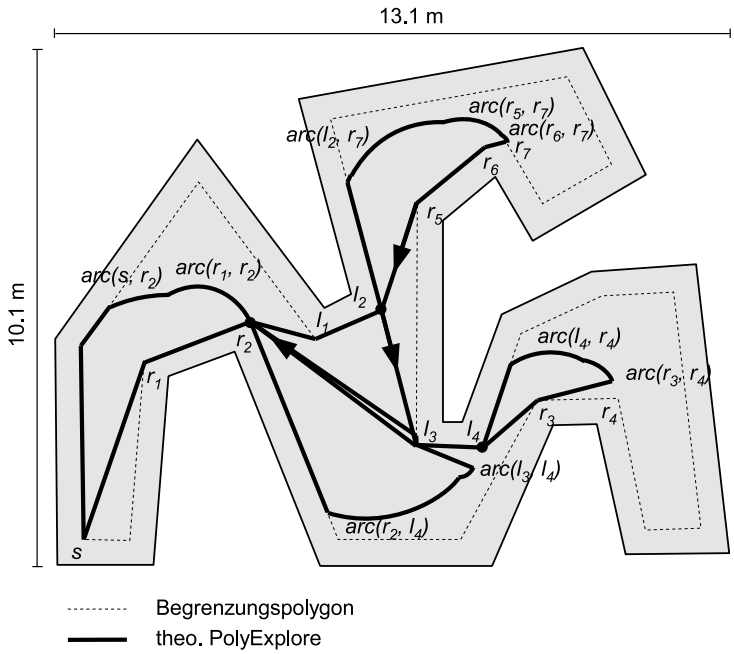

Abbildung 4.16: Szenario D: Erkundung des Polygons.

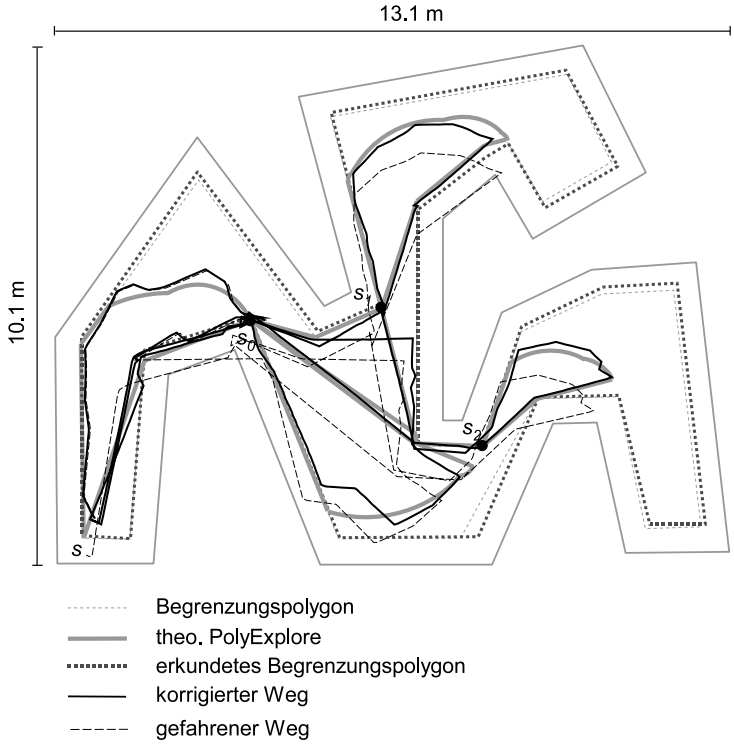

Abbildung 4.17: Szenario D, Simulation.

In diesem Beispiel stellen die Ecken r_2, l_2 und l_4 die *StagePoints* dar. Dabei müssen die Ecken nach l_2 und l_4 unabhängig voneinander erkundet werden.

Die Erkundung startet in s und erkundet zuerst entlang der Polygonkante die Ecke r_1 und anschließend r_2. Bis r_2 erkundet ist, muss sich der Roboter irgendwann von dem Polygonrand lösen und die Erkundung zuerst auf dem Kreisbogen $arc(s, r_2)$ und dann auf $arc(r_1, r_2)$ fortsetzen. Da r_2 einen Innenwinkel über *270°* hat, muss der Roboter bis zu der Ecke fahren, um diese komplett zu erkunden. Sobald r_2 erkundet ist, kehrt der Roboter nach s zurück, da zu diesem Zeitpunkt keine weiter rechte Ecken in der $TargetList$ vorhanden sind.

Von s aus sind keine unerforschten linke Ecken zu sehen. Die $TodoList$ enthält aber noch zwei unerforschte linke Ecken l_1 und l_3, die r_2 als Vater in SPT haben. Also ist r_2 der nächste *StagePoint*, zu dem sich der Roboter bewegt. Von r_2 erkundet der Roboter dann die linke Ecke l_3. Während dieser Erkundung werden noch l_2 und l_4 entdeckt und erkundet. Danach kehrt der Roboter zum *StagePoint* r_2 zurück.

An dieser Stelle sind keine unerforschten Ecken direkt sichtbar, aber die $TodoList$ enthält noch r_5 und r_3. Da die beiden Ecken unterschiedliche Väter in SPT haben, wird r_5 als erstes erkundet, da diese Ecke vor r_3 liegt, wenn man die Ecken im Uhrzeigersinn entlang des Polygonrandes sortiert. Also wird l_2 zum nächsten *StagePoint* zu dem sich der Roboter bewegt.

Von l_2 erforscht der Roboter r_5 bzw. r_6 und r_7 und kehrt zu l_2 zurück, sobald diese Ecken erkundet sind. An dieser Stelle existieren keinen unerforschten Ecken, die den aktuellen *StagePoint* als Vater in SPT haben. Da aber vom letzten *StagePoint* r_2 noch die unerforschte Ecke r_3 erreichbar ist, fährt der Roboter auf dem kürzesten Weg zum nächsten *StagePoint*, in diesem Fall l_4. Von da aus erkundet der Roboter die letzten Ecken r_3 und r_4 des Polygons und kehrt auf dem kürzesten Weg zum aktuellen *StagePoint* l_4 zurück.

Danach begibt sich der Roboter immer zum letzten *StagePoint* und prüft, ob noch unerforschte Ecken vorhanden sind. Irgendwann kehrt er so zum Startpunkt zurück. Dann ist die Erkundung des Polygons beendet. Der zurückgelegte Weg beträgt am Ende 69.51 Meter. die SWR beträgt in diesem Beispiel 38.22 Meter.

Abbildung 4.17 zeigt den Erkundungsweg des Roboters im Simulator. Obwohl der Simulator einen geringen Odometriefehler hat, sieht man deutlich, dass der korrigierte Weg zum Ende immer mehr von dem Weg abweicht, der von der Odometrie des Roboters bestimmt wurde. Trotz dieser Odometriefehler weicht das erstellte Polygon nur sehr gering von dem Originalpolygon ab. Viel interessanter ist jedoch der zurückgelegte Weg des Roboters, um das Polygon zu erkunden.

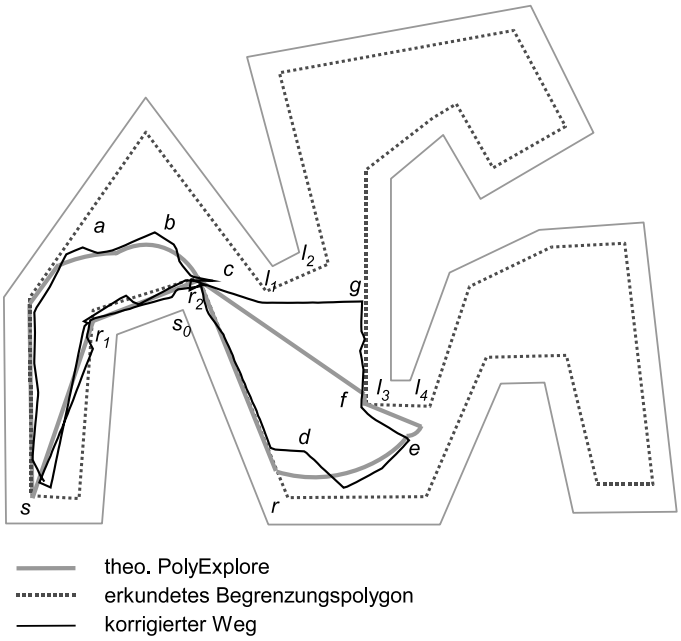

Abbildung 4.18: Szenario D: Erkundung vom *StagePoint* s und s_0.

Um den Weg besser nachvollziehen zu können, wird dieser in zwei Teile zerlegt, so dass die Erkundung von jedem *StagePoint* erkennbar wird. Die Abbildung 4.18 zeigt den Anfang der Erkundung. Der Roboter startet nicht genau in s und muss deswegen erst zur Polygonkante fahren. Danach verläuft die Erkundung am Polygonrand, bis sich der Roboter von diesem löst und auf einer Kreisbahn die Ecke r_2 erkundet. Man sieht in a und b die Approximation der Kreisbahnen. Als der Roboter bei r_2 ankommt, ist die Erkundung der rechten Ecken noch nicht zu Ende. Aufgrund der begrenzten Sicht, kann der Roboter die Ecke r nicht sehen. An der Stelle c kann der Roboter r sehen, beendet die Erkundung der rechten Gruppe und kehrt zum Startpunkt zurück.

Danach begibt sich der Roboter zum nächsten *StagePoint* in s_0 und beginnt mit der Erkundung von l_3, die zunächst am Polygonrand verläuft. Man sieht, dass der Roboter sich früher von dem Rand löst, als in der Theorie. Das liegt daran, dass der Roboter in der Theorie an dieser Stelle schon die Ecke l_4 sehen kann und dabei ist diese zu erforschen. Bei dem realen Roboter wurde die Kante (l_3, l_4) noch nicht eingefügt, da durch die Entfernung zu wenige Laserpunkte auf die Kante treffen, wodurch die Kante in dem Sichtbarkeitspolygon nicht erzeugt wird (siehe Kapitel 3.2.1) und der Roboter weiterhin l_3 erforscht. In d sieht der Roboter dann l_4 und begibt sich zu der Kreisbahn, auf der l_4 erforscht wird.

Interessant ist die Stelle e. Hier ist der reale Roboter mit der Erkundung

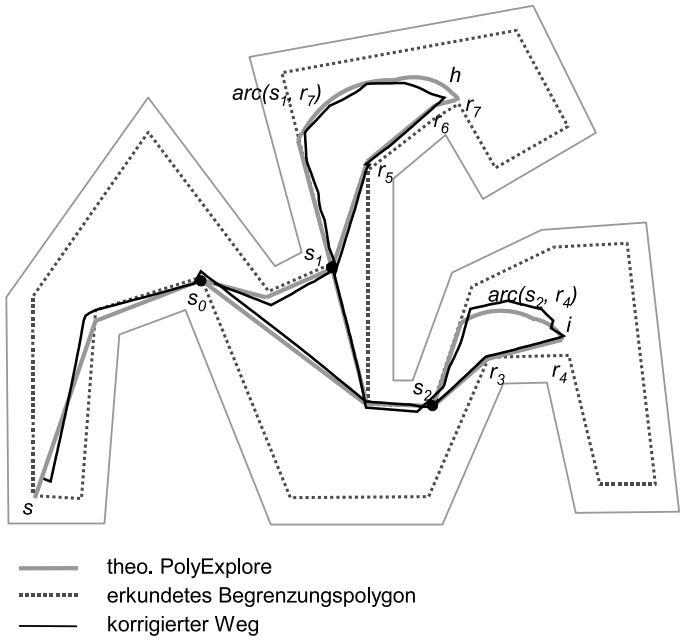

Abbildung 4.19: Szenario D: Erkundung vom *StagePoint* s_1 und s_2.

von l_4 fertig. In der Theorie muss der Roboter noch weiter fahren, um hinter die Begrenzungslinie sehen zu können. Dieser Unterschied entsteht dadurch, dass der reale Roboter nicht das Begrenzungspolygon erkundet und die Kante hinter l_4 nicht so lang ist. Und weil die Kante nicht so lang ist, kann der reale Roboter die Kante schon vorher komplett sehen.

In f ist der theoretische PolyExplore mit der Erkundung der linken Gruppe fertig, der reale Roboter muss aufgrund seiner begrenzten Sicht noch bis g fahren, um l_2 komplett sehen zu können. Danach kehrt der Roboter zu s_0 zurück.

Versuch	SWR	WmF[1]	WoF[2]	KFmF[3]	KFoF[4]
SIM[5]	38.22 m	75.91 m	74.57 m	1.9861	1.9210

[1] WmF = Weg mit Fehlerkorrektur
[2] WoF = Weg ohne Fehlerkorrektur
[3] KFmF = Kompetitiver Faktor mit Fehlerkorrektur
[4] KFoF = Kompetitiver Faktor ohne Fehlerkorrektur
[5] SIM = Simulation

Tabelle 4.4: Szenario D: kompetitiver Faktor der Simulation (theoretischer Wert: 1.8187).

Die Abbildung 4.19 zeigt die weitere Erkundung von s_0 aus. Vergleicht man den theoretischen mit dem realen Weg, so stellt man fest, dass diese

bis auf die Abweichungen durch die Approximation der Kreisbahn durch ein Polygonzug gleich sind. Das liegt daran, dass die Räume des Polygons relativ klein sind, wodurch die Aspekte der unzureichenden Sicht des realen Roboters nicht zum Tragen kommen.

Wie man der Tabelle 4.4 entnehmen kann, führen die Abweichungen durch die begrenzte Sicht zu einem größerem kompetitiven Faktor. Aufgrund der Länge des Weges und nur kleinen Abweichungen bei der Erkundung von den *StagePoints* s_1 und s_2 ist der Unterschied zu dem theoretischen Wert klein.

4.2.5 Szenario E

Wie auch bei letzten Szenario, soll in diesem die Erkundung des Polygons von mehreren *StagePoints* getestet werden. Dabei sollen die *StagePoints* dieses Mal in einer Reihe liegen. Die Abbildung 4.20 zeigt das Szenario mit dem theoretischen Weg. Das Szenario ist zwar kleiner als das letzte, hat aber genau so viele *StagePoints*.

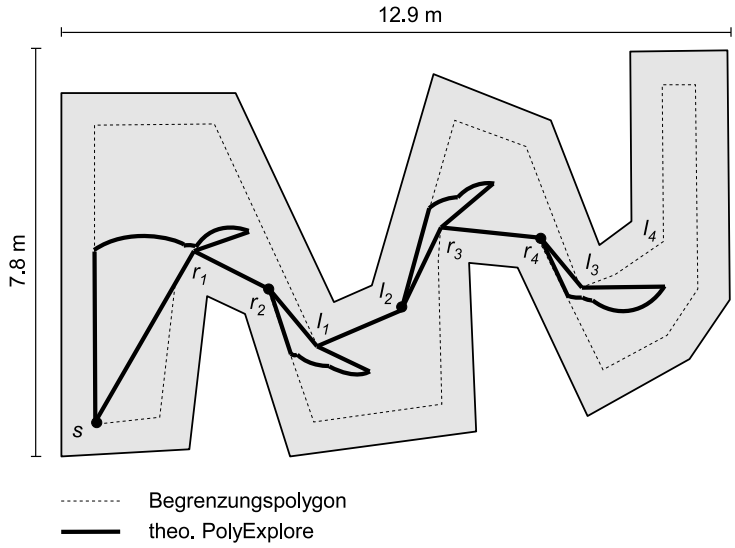

Abbildung 4.20: Szenario E: Erkundung des Polygons.

Der Roboter startet wieder in s, erkundet zuerst r_1 bzw. r_2 und kehrt zu s zurück. Danach begibt sich der Roboter zu Ecke r_2, die als neuer *StagePoint* ausgewählt wurde. Von da erkundet der Roboter l_1 bzw. l_2 und kehrt zu r_2 zurück. Als nächstes wird l_2 zum *StagePoint*. Von da erkundet der Roboter die Ecke r_3 bzw. r_4. Anschließend ist r_4 der letzte *StagePoint*. Nach der Erkundung von l_3 bzw. l_4 kehrt der Roboter zum Startpunkt zurück und die Erkundung des Polygons ist beendet.

Der komplette Erkundungsweg betragt 56.26 m während die *SWR* 30.22 m lang ist.

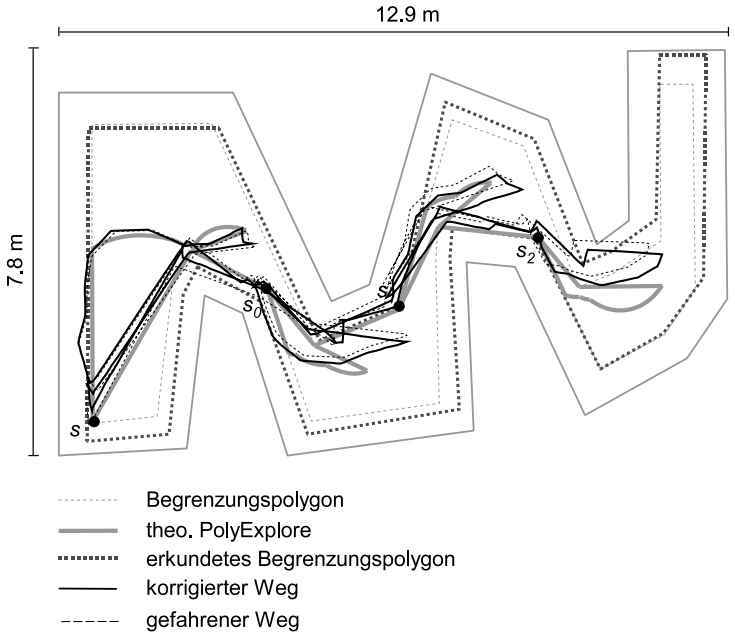

Abbildung 4.21: Szenario E, Simulation.

Vergleicht man nun den zurückgelegten Weg von dem realen Roboter in der Abbildung 4.21 mit dem theoretischen Weg, so stellt man fest, dass nur kleine Abweichungen erkennbar sind. Einige dieser Abweichungen kann man wieder auf die Sicht des realen Roboters zurückführen. Andere sind durch die schlechte Fehlerkorrektur entstanden, wie man an dem Weg und Polygon nach dem s_2 sieht.

Insgesamt ergibt sich nur eine geringe Abweichung von dem theoretischen Wert, wie man der Tabelle 4.5 entnehmen kann.

Versuch	SWR	WmF[1]	WoF[2]	KFmF[3]	KFoF[4]
SIM[5]	30.22 m	58.64 m	57.43 m	1.9404	1.9004

[1] WmF = Weg mit Fehlerkorrektur
[2] WoF = Weg ohne Fehlerkorrektur
[3] KFmF = Kompetitiver Faktor mit Fehlerkorrektur
[4] KFoF = Kompetitiver Faktor ohne Fehlerkorrektur
[5] SIM = Simulation

Tabelle 4.5: Szenario E: kompetitiver Faktor der Simulation (theoretischer Wert: 1.8616).

4.2.6 Szenario F

Als letztes Szenario soll der Worstcase Fall untersucht werden. In der Diplomarbeit von R. Hagius [14] wurde die Form eines Polygons vorgestellt, in dem die Strategie den schlimmsten bekannten kompetitiven Faktor von 5 erreicht. Den Beweis für diesen Faktor findet man in der gleichen Arbeit. Basierend auf diesem Beispiel wurde ein Polygon erstellt, welches dem Beispiel nahe kommt. Dieses Polygon ist in Abbildung 4.22 dargestellt.

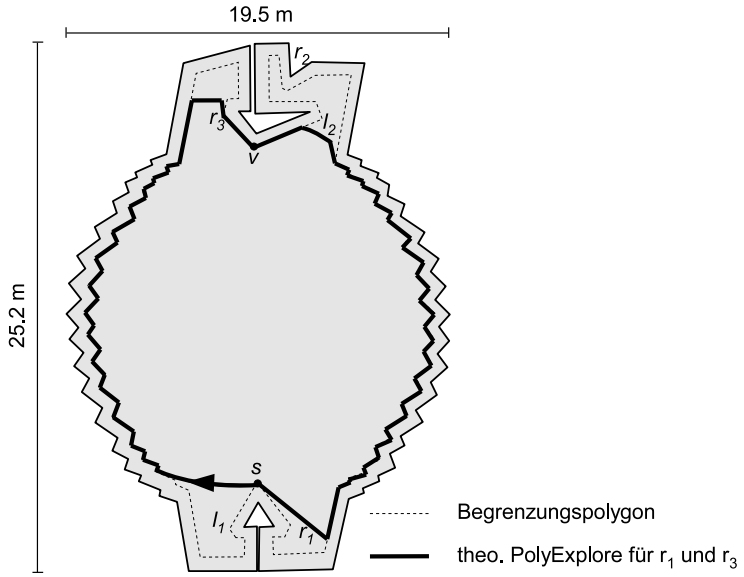

Abbildung 4.22: Szenario F, Erkundung der rechten Gruppe im Worstcase Szenario.

Das Polygon sollte die Form eines Kreises mit dem Durchmesser $\overline{sv}$ haben. Der Rand des Polygons entspricht der *angle hull* Konstruktion aus [18]. In einer solchen Konstruktion sind alle Ecken entlang des Halbkreises rechte Ecken, so dass man sowohl von s als auch von v in alle Ecken einsehen kann. Geht die Anzahl dieser Ecken auf dem Halbkreis gegen das Unendliche, kommt die Länge entlang der Ecken am Halbkreis dem Wert $2\overline{sv}$ sehr nahe. Bei einer realen Umgebung, aber auch in der Simulation ist die Anzahl dieser Ecken begrenzt, so dass man an dieser Stelle nicht mehr den kompetitiven Faktor von 5 erreicht. Ein weiterer Aspekt bei der Konstruktion des Polygons ist die Größe der Taschen bei s und v. Diese müssen minimal im Vergleich zur Länge $\overline{sv}$ sein. Damit das erreicht wird, muss $\overline{sv}$ sehr lang sein, da die Taschen aufgrund der Ausdehnung des Roboters eine Mindestgröße haben müssen.

Das Polygon, welches für die Simulation erstellt wurde, ist 25.2 m lang und 19.5 m breit. Zuerst wird der theoretische Weg von PolyExplore betrach-

tet. Der Roboter startet bei s. Von da aus erkundet er zuerst die Ecke r_3. Der Weg geht entlang des Polygonrandes bis r_3 erkundet ist. Bei einer idealen Konstruktion des Polygons würde die Länge dieses Weges gegen $\overline{sv}$ gehen. Anschließend wird die nächste Ecke r_1 aus der rechten Gruppe erkundet. Dazu muss sich der Roboter wieder entlang des Polygonrandes bewegen, bis die Ecke erkundet ist. Danach kehrt der Roboter zum Startpunkt zurück. Dieser Weg ist in der Abbildung 4.22 dargestellt. Anschließend muss die linke Gruppe von Ecken, bestehend aus l_1 und l_2, analog erkundet werden.

Danach muss der Roboter noch r_2 erkunden. Dazu muss er sich zum nächsten *StagePoint* bei l_2 begeben und von da r_2 erkunden. Anschließend kehrt der Roboter zum Startpunkt zurück.

Für dieses Polygon erhalten wir in der Theorie einen kompetitiven Faktor von 3.5373.

Nun sollte die Strategie in einer Simulation getestet werden. Leider stellte sich schnell heraus, dass die aktuelle Implementierung große Schwierigkeiten mit dem Szenario hatte. Aufgrund der begrenzten Sicht des Roboters und der Größe des Polygons konnte der Roboter nicht das ganze Polygon sehen. Die Ecken, die aus diesem Grund erforscht wurden, führten den Roboter durch die Mitte des Polygons, wodurch er nur selten den Rand sehen konnte, um seine Position zu korrigieren. Das führte schließlich dazu, dass die Karte nicht mehr brauchbar war. Leider kann die Reichweite des Lasers nicht im Simulator einstellen, um dieses Problem umzugehen. Eine weitere Schwierigkeit, die dazu kam, war die Anzahl der Segmente, aus der die Karte bestand. Durch die große Anzahl der Segmente brauchte die Strategie viel Zeit, um die nächste Entscheidung zu treffen. Da der Simulator, der Server und der Client unabhängig voneinander sind, wartet der Simulator nicht auf die Ergebnisse des Clients und setzt seine Bewegung fort, bis ein neues Befehl von dem Client kommt. Ist der Client ausgelastet, so kommen die Befehle zu spät und die Strategie funktioniert nicht richtig. Es gibt auch keine Möglichkeit den Simulator einzuweisen, auf die nächsten Befehle zu warten. Aus diesen Gründen musste die Simulation frühzeitig abgebrochen werden. Einer der Wege, den der Roboter bis zum Abbruch zurückgelegt hat, wird in der Abbildung 4.23 gezeigt.

An der Startposition sieht der Roboter etwa bis zur Stelle bei ur_1. Die Erkundung startet also bei s auf dem Kreisbahn $arc(s, ur_1)$. Diese Kreisbahn führt den Roboter durch die Mitte des Polygons, wo der Roboter keine neuen Polygonränder sieht. An der Stelle a kann der Roboter weiteres Stück vom Polygonrand sehen. Dadurch wird ur_2 zur neuen Ecke, die der Roboter auf der Kreisbahn $arc(s, ur_2)$ erforschen soll. Zu diesem Zeitpunkt befindet sich der Roboter aber etwas weit von der neuen Kreisbahn entfernt. Daher fährt der Roboter auf dem geraden Weg zu dem nächsten Punkt auf der neuen Kreisbahn.

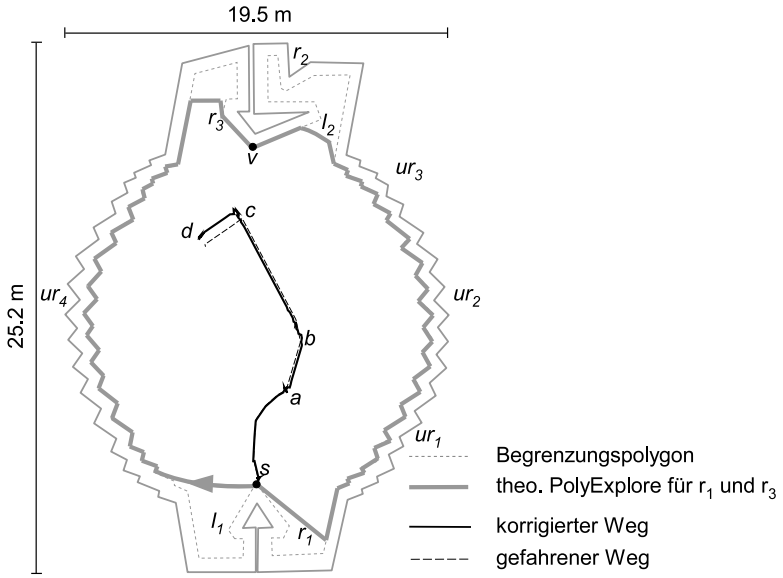

Abbildung 4.23: Szenario F, Anfangsweg der erfolglosen Simulation.

In b wird ein weiterer Teil des Polygonrandes sichtbar. Nun muss der Roboter wieder zu der neuen Kreisbahn fahren. Als der Roboter in c ankommt, wird auch die andere Seite des Polygons sichtbar. Dadurch wird ur_4 die neue rechte Ecke, die jetzt auf der Kreisbahn $arc(s, ur_4)$ erforscht werden soll. Da aber der nächste Punkt von der Position c zu der Kreisbahn $arc(s, ur_4)$ die unerforschte Ecke selbst ist, fährt der Roboter direkt auf die Ecke zu. Dies entspricht aber nicht der Idee von PolyExplore. In d werden weitere Kanten des Polygons sichtbar und an dieser Stelle musste die Simulation abgebrochen werden, da die erstellte Karte wegen o.g. Gründen nicht mehr für die Erkundung geeignet war.

Obwohl die Simulation nicht erfolgreich gewesen ist, so konnte man den ersten Eindruck gewinnen, welchen Einfluss diebegrenzte Sicht des Roboters auf den Erkundungsweg der Strategie hat.

4.3 Abschließende Bewertung

Wenn man einen Vergleich zwischen der theoretischen und der realen Onlinestrategie anstellt, sollte beachtet werden, dass hier zwei Strategien verglichen werden, die unterschiedliche Sichtweiten haben. Bei der Theoretischen wird eine unbegrenzte Sicht angenommen während in der Realität die Sichtweite begrenzt ist. Das führt dazu, dass in der Realität die Onlinestrategie im Allgemeinen nicht komptitiv zur *Shortest Watchman Route* kompetitiv ist. Das kann man sich anhand eines Beispiels in der Abbildung 4.24 klar

machen. Würde man den Gang verlängern, so bleibt der optimale Erkundungsweg W_{opt} immer gleich, der Weg der Strategie $W_{begrenzt}$ würde mit der Verlängerung des Gangs auch länger werden.

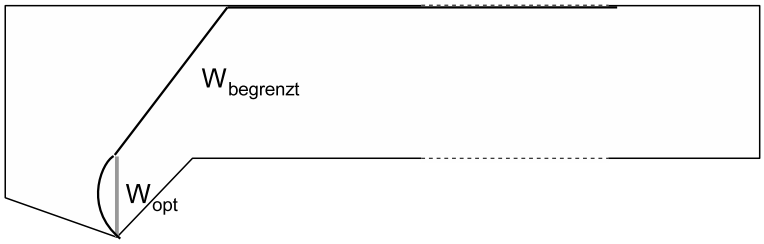

Abbildung 4.24: Strategie mit begrenzter Sicht ist nicht zu SWR kompetitiv.

Um den Vergleich in dieser Arbeit trotzdem machen zu können, wurden relativ kleine Szenarien ausgewählt. Wie man aber im Kapitel 4.2.2 gesehen hat, kann der Effekt der begrenzten Sicht bei der eingesetzten Technik schon in kleinen Räumen auftreten, oder durch kurze Kanten (Kapitel 4.2.4).

In der Tabelle 4.6 werden die Ergebnisse aus dem Kapitel 4.2 zusammengefasst. Es werden für jedes Szenario der kompetitive Faktor für die theoretische Berechnung und die Ergebnisse aus den Versuchen gegenüber gestellt. Für die Versuche in der realen Umgebung wurde zwischen dem kompetitiven Faktor für den Weg mit und ohne Fehlerkorrektur unterschieden. Für die Simulation wurde der Mittelwert aus den beiden Wegen gebildet, da der Unterschied sehr gering war und beide Wege von dem tatsächlich zurückgelegten Weg abweichen können.

Szenario	TPE[1]	PPEmF[2]	PPEoF[3]	Simulation
A	1.1989	1.6478	1.4328	1.2477
B	1.4014	2.3072	2.0576	2.1806
C	1.5342	2.2037	1.7943	1.5606
D	1.8187	—	—	1.9536
E	1.8616	—	—	1.9204
F	3.5373	—	—	—

[1] TPE = theoretischer PolyExplore
[2] PPEmF = praktischer PolyExplore mit Fehlerkorrektur
[3] PPEoF = praktischer PolyExplore ohne Fehlerkorrektur

Tabelle 4.6: Der durchschnittliche kompetitive Faktor der einzelnen Szenarien im Vergleich zu dem theoretischen Faktor.

Betrachtet man die Simulationsergebnisse von den Szenarien A, C, D und E, so stellt man nur eine geringe Abweichung zu den theoretischen Werten fest. In diesen beiden Szenarien hat der Roboter zu jedem Zeitpunkt genug von dem Polygon gesehen, um die Ecken richtig zu identifizieren.

Auch war sein Odometriefehler nicht so groß, dass es Auswirkungen auf den zurückgelegten Weg hatte. In den realen Messungen ist das leider nicht der Fall, was sich in einem höherem kompetitiven Faktor niederschlägt. Der hohe Unterschied zwischen dem korrigierten und nicht korrigierten Weg liegt nicht nur an den Odometriefehlern, sondern auch daran, dass der korrigierte Weg nicht stetig ist. Jedes Mal, wenn eine Korrektur vorgenommen wird, kann der Roboter einen Sprung in seiner Position machen. Diese Entfernung wird auch mit in die Berechnung übernommen.

Interessant ist auch das Ergebnis von dem Szenario B. In diesem Fall musste der Roboter ein Flur erforschen, so dass eine Ecke außer Sicht lag. Das führte dazu, dass der Weg in der realen Messung und in der Simulation länger als der theoretische war. Betrachtet man jedoch nur die realen Messungen und die Simulation, so stellt man fest, dass die Faktoren nur einen kleinen Unterschied aufweisen. Der erste Eindruck, dass die Fehlerkorrektur in diesem Fall funktioniert hat, wird schnell widerlegt, wenn man sich die Abbildungen 4.8 und 4.9 anschaut. Man stellt fest, dass die erstellten Polygone relativ zum Originalpolygon kleiner ausfallen. Auch der am Anfang der Erkundung fast stetige Weg wird zum Ende hin immer zackiger. Noch extremer sieht man die Verschlechterung des Weges in der Abbildung 4.13. dass die Faktoren also so nahe bei einander liegen, kann auch ein Zufall sein. dass aber alle Faktoren viel größer als der theoretische Wert sind, liegt an den Auswirkungen der begrenzten Sicht des Roboters.

Auffallend ist, dass in allen realen Versuchen der korrigierte Weg des Roboters zum Ende immer zackiger wird, was auf eine fehlerhafte Korrektur hinweist. Wodurch kommt das? Der Grund liegt zum größten Teil in der Approximation der Hindernisse. Kanten, die im Bereich einer Gerade liegen, werden wie in Abbildung 4.25 zu einer Kante zusammen gefasst. Solange der Roboter die beiden Kanten sieht, hebt sich der Fehler auf. Sollte sich der Roboter drehen und anschließend nur eine Hinderniskante sehen, wird seine Position zu P_{korr} fälschlicherweise an der Polygonkante korrigiert.

Zusätzlich werden dadurch die Polygonkanten falsch aktualisiert. Dies kann dazu führen, dass einige Polygonkanten aus dem Polygon verschwinden und im schlimmsten Fall das Polygon unbrauchbar wird.

Versucht man jedoch das Polygon sehr genau darzustellen, erhält man am Ende so viele Polygonkanten, dass der Aktualisierungsaufwand zu hoch ist.

In der Simulation konnte dieses Verhalten nicht beobachtet werden. Das kann verschiedene Ursachen haben. Zum einen macht der Roboter nicht so große Odometriefehler in einem kurzen Intervall, zum anderen besteht der Polygonrand in der Simulation aus geraden Kanten. In der Realität wurden Polygonränder aus Tischen und Kartons aufgebaut, wie es in der Abbildung 4.15 zu sehen ist. Aus diesen Gründen kann eine Simulation den

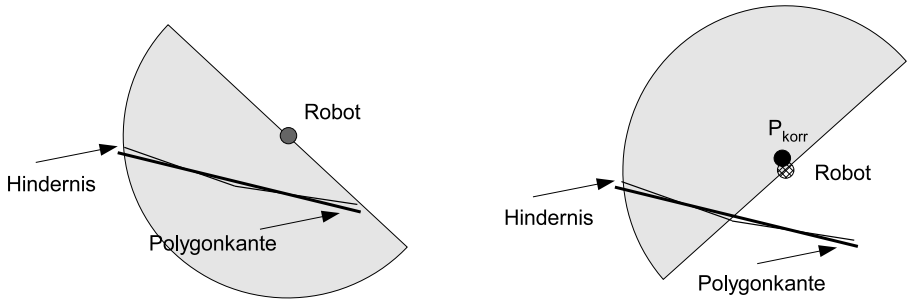

Abbildung 4.25: Approximation der Hindernisse durch eine Kante (links), und falsche Korrektur nach einer Drehung des Roboters (rechts).

realen Roboter nicht ersetzen.

Anhand der gemachten Erfahrungen sollen nun die Diskrepanzen zwischen der Theorie und der Realität mit ihrer Relevanz aufgelistet werden:

- Odometrie- und Laserfehler (hoch)
  Aufgrund besonders hohen Odometriefehlern entstanden oft große Abweichungen zu dem Originalpolygon, was natürlich auch zu den Abweichungen in dem Erkundungsweg führte. Ein Weg zur Lösung des Problem könnte es sein, Roboter mit weniger hohem Odometriefehler einzusetzen und andere Korrekturverfahren verwenden, die z.B. auf der Gridkarten basieren und viel robuster sind, siehe [16]. Ganz wird man den Odometriefehler aber nicht korrigieren können.

- unzureichende Sicht des Roboters (hoch)
  Wie man gesehen hat, kommt dieser Aspekt schon bei relativ kleinen Räumen zustande. Zum einen Teil liegt es an der Reichweite des Lasers. Diese könnte man noch durch höhere Leistung vergrößern. Ein weiterer Punkt ist die Auflösung des Laser und die damit verbundene Erstellung der Polygonkanten, wie sie in Kapitel 3.2.1 vorgestellt wurde. Damit muss der Roboter sich weiter von der Wand entfernen, um diese in seiner Länge sehen zu können.

- Approximation der Kreisbahn durch ein Polygonzug (gering)
  Abhängig von der Genauigkeit der Approximation kann es bei Kreisbahnen mit einem kleinen Radius passieren, dass der Roboter direkt auf das Ziel zufährt, wie z.B. in Szenario D beim Erkunden von r_7. Man könnte natürlich versuchen die Wegpunkte näher bei einander zu setzen oder den Roboter direkt auf einer Kreisbahn zu steuern. Allerdings haben wir in Kapitel 3.1.2 gesehen, dass gerade die Drehungen einen großen Odometriefehler verursachen.

- Strategieplanung basiert auf dem Begrenzungspolygon — in der Praxis vervollständigt der Roboter das Originalpolygon (gering)
  Wie man in der Abbildung 4.18 sehen konnte, kann es dazu führen, dass der Roboter eine Kante schon komplett sehen kann, bevor er den Rand des Begrenzungspolygons erreicht hat. In der Theorie muss der Roboter aber bis zu dem Rand des Begrenzungspolygons fahren. Dieser Effekt könnte sich noch verstärken, wenn man die Sicht des Roboters verbessern würde.

- 180° Laserscanner (mittel)
  Bei dem durchgeführten Versuchen konnte der Roboter immer nur einen Ausschnitt von 180° sehen. Das hatte zur Folge, dass der Roboter sich immer auf einer Stelle drehen musste, um die Umgebung um sich herum erkunden zu können. Das hat zur Folge, dass der Roboter viele unnötige Drehungen machen musste. Der Einsatz eines zweiten Laserscanners für die 360° Sicht wäre an dieser Stelle für die Erkundung sinnvoll, aber teuer.

Da einige Aussagen über die Diskrepanzen zwischen der Theorie und der Realität auf den Erfahrung aus der Simulation beruhen, sollten an dieser Stelle auch die Diskrepanzen zwischen der Simulation und der Realität kurz angesprochen werden.

- Schwere Anpassung der Fehler an das reale Robotersystem
  Es sind viele Versuche und Messungen erforderlich, um die Fehler, die der reale Roboter macht in der Simulation nachzubilden. Während die eingestellten Werte in der Simulation immer bleiben, treten in der Realität immer wieder Abweichungen in Form von verschiedenen Untergründen oder aber auch Abnutzungserscheinungen auf.

- Schwieriger Nachbau der realen Umgebung
  Die Simulation einer realen Umgebung setzt eine genaue und zeitaufwendige Vermessung voraus. Auch nach einer genauen Vermessung kann man mit dem eingesetzten Simulator nicht die Eigenschaften eines Hindernisses simulieren. So entsteht in dem Simulator meist eine viel zu ideale Welt.

Zusammenfassend kann man sagen, dass bei verbesserter Fehlerkorrektur für die Odometrie des Roboters und hinreichend kleinen Räumen die Onlinestrategie sich auf einem realen Robotersystem realisieren lässt. Bei größeren Räumen lässt sich aufgrund der begrenzten Sicht des Roboters das Verhalten der Strategie, wie z.B. in den Szenarien A oder F, nur schwer vorhersagen.

Kapitel 5

Zusammenfassung

In dieser Arbeit wurde die Explorationsstrategie PolyExplore auf einem realen Roboter implementiert. Anhand dieser Implementierung wurde das Verhalten der Strategie sowohl in der Simulation, als auch in der realen Umgebung analysiert und bewertet. Die größte Herausforderung stellte die Korrektur der Odometriefehler des Roboters dar. Diese sind besonders bei einer Drehung des Roboters sehr hoch. Noch größere Fehler entstehen bei einer Drehungen während der Fahrt und davon gibt es bei PolyExplore reichlich. Durch die Approximation der Kreisbögen durch einen Polygonzug und eine einfache Fehlerkorrektur konnten die ersten Versuche in relativ einfachen und kleinen Umgebungen durchgeführt werden. Für umfangreiche Tests müsste eine verbesserte Fehlerkorrektur eingesetzt werden.

Trotz der einfachen Fehlerkorrektur ließen sich viele Szenarien aufbauen, anhand derer das Verhalten der Strategie in der realen Umgebung auf einem realen Roboter getestet werden konnte. Die Versuche mussten oft mehrmals durchgeführt werden, um brauchbare Resultate zu erzielen, aber angesichts der einfachen Fehlerkorrektur, war das zu erwarten. Das erste und wesentliche Problem, mit welchem die Strategie konfrontiert wird, ist die begrenzte Sicht des Roboters. Diese liegt nicht nur an der Reichweite des Laserscanners, sondern auch an seiner Auflösung, wie man in Kapitel 3.2.1 sehen konnte.

Besonders stark äußert sich das Problem in den großen Räumen, deren Ausmaße der Roboter nicht von Anfang an sehen kann. Wie man anhand des Szenario F im Kapitel 4.2.6 sehen konnte, läuft der Roboter gegen Uhrzeigersinn die imaginären rechten Ecken am Rand des Raumes ab. Irgendwann versagt die Strategie und der Roboter ist gezwungen diese Ecke direkt anzufahren.

Waren die Räume ausreichend klein, so dass die Sicht des Roboters ausreichte, um diese komplett sehen zu können, so verhielt sich die Strategie der Theoretischen recht ähnlich. Dass es aber auch bei zu kleinen Räumen

es zu Abweichungen kommen kann, konnte man in dem Szenario D in Kapitel 4.2.4 sehen. Diese entstanden indirekt durch das Begrenzungspolygon, welches als Referenz für die theoretische Berechnung benutzt wurde. Der Roboter aktualisiert das Originalpolygon und kann einige Kanten bei kleinen Räumen schon sehen, bevor er den Cut der gleichen Kante im Begrenzungspolygon erreicht hat. Die Abweichungen, die dadurch entstanden sind, waren im Vergleich zu dem gesamten Weg minimal.

Für die Art der einfachen Polygone, in der die Sicht des Roboters nicht zu seinem Nachteil wird, lässt sich also die Strategie auf einem realen Roboter realisieren. Bei anderen Polygonen ist das Verhalten der Strategie von der Sichtweite des Roboters abhängig. Damit ist die Strategie nicht zu der SWR mit unbegrenzter Sicht kompetitiv. Für den Einsatz in der realen Umgebung müsste die Strategie an die begrenzte Sicht des Roboters angepasst werden, um auch in großen Räumen funktionieren zu können.

Kapitel 6

A Benutzerhandbuch

Die PolyExplore Software wurde als eine Experimentierumgebung zur Steuerung des Roboters entwickelt. Außer der manuellen Steuerung gibt es die Möglichkeit den Roboter von einer Explorationsstrategie steuern zu lassen. Die aktuellen Sensor- und Positionsdaten, sowie viele der Statusmeldungen werden dabei auf einer graphischen Oberfläche dargestellt, um dem Benutzer so viel Komfort wie möglich zu bieten. Die Software lässt sich sowohl unter Microsoft Windows als auch unter Linux ausführen.

Im Kapitel 6.1 werden die Voraussetzungen für das Ausführen der Software beschrieben und das Kapitel 6.2 erklärt die Bedienung der Software.

6.1 Voraussetzungen

Die PolyExplore Software liegt als *polyExplore.jar* vor und benötigt für die Ausführung mindestens Java 1.5[1]. Für das Auslesen und Darstellen der Trace-Dateien sind damit die Voraussetzungen erfüllt.

Will man den Roboter steuern oder simulieren, so wird zusätzlich die ARIA[2] Bibliothek benötigt. Die PolyExplore Software wurde mit der ARIA Version 2.2.0 getestet. Obwohl die ARIA Bibliothek auch Java Klassen zur Verfügung stellt, werden aufgrund des Funktionsumfanges die C++ Methoden der ARIA benutzt. Die Kommunikation zwischen Java und C++ geschieht über die JNI Schnittstelle.

Für die Benutzung der PolyExplore Software unter Windows wurde schon *pioneer.dll* vorkompiliert, um den Zugriff auf die C++ Methoden der ARIA Bibliothek zu erlauben. Unter Linux muss die entsprechende *pioneer.so* noch erzeugt werden. Um diese benutzen zu können, muss der Pfad sowohl zu den ARIA Bibliotheken als auch zu *pioneer.dll* bzw. *pioneer.so*

[1] http://java.sun.com/
[2] http://www.activrobots.com/SOFTWARE/index.html

dem System bekannt sein. Dazu muss man unter Windows die *PATH* und unter Linux *LD_LIBRARY_PATH* Variable um diesen Pfad erweitern.

6.2 Bedienung

6.2.1 Starten des Servers

Um den Server zu starten reicht der Aufruf von:

java -cp polyExplore.jar examples.robotServer.RobotServer [args...]

Werden keine Argumente übergeben, so startet der Server mit den Standardeinstellungen. Dabei wird angenommen, dass der Server auf einem Roboter ausgeführt wird und der Server lauscht auf der Adresse *localhost* mit dem Port *15000*. Um die Standardeinstellungen zu ändern steht eine Reihe von Argumenten zur Verfügung, die an den Server beim Start übergeben werden können.

-i Mit diesem Argument, gefolgt von einer gültigen IP Adresse, kann die lokale IP Adresse gesetzt werden, an welcher der Server lauschen soll. Die Standardeinstellung ist *localhost*.

-p Damit kann der lokale Port gesetzt werden, an welchem der Server lauschen soll. Die Standardeinstellung ist *15000*.

-x Wird dieses Argument übergeben, so wird eine graphische Oberfläche für die Statusmeldungen des Servers gestartet.

-s Soll der Server auf einer Simulation starten, so muss dieses Argument beim Start gesetzt sein. Hat nur eine Relevanz, wenn *-f* nicht benutzt wird.

-f Mit diesem Argument kann eine Datei übergeben werden, so dass der Roboter die Odometrie und Laserdaten aus dieser Datei ausliest und an den Client weiterleitet.

-h Damit kann die Hilfe für den Server aufgerufen werden.

Nachdem der Server gestartet wurde, wartet er, bis ein Client sich zu diesem Server verbindet. Erst nachdem der Client versucht sich mit dem Roboter zu verbinden, stellt der Server die Verbindung zu Roboter, Simulator oder der Datei her. Der Server macht nichts anderes als die Daten von Client an den Roboter bzw. vom Roboter an den Client weiterzuleiten.

6.2.2 Starten des Clients

Bevor der Client gestartet wird, muss der Server gestartet sein, da der Client sich nur einmal zu dem Server verbindet. Wurde keine Verbindung nach dem Start des Clients hergestellt, so muss der Client neu gestartet werden. Der Client kann durch einen einfachen Aufruf mit den Standardeinstellungen gestartet werden:

java -cp polyExplore.jar examples.robotClient.RobotClient [args...]

Beim Server, kann man auch den Client mit geänderten Einstellungen, die man über Argumente übergibt, starten.

-li Mit diesem Argument gefolgt von einer gültigen IP Adresse, kann die lokale IP Adresse gesetzt werden, an welcher der Client lauschen soll. Die Standardeinstellung ist *localhost*.

-lp Damit kann der lokale Port gesetzt werden, an welchem der Server lauschen soll. Die Standardeinstellung ist *15001*.

-ri Mit diesem Argument wird die IP Adresse des Servers, zu dem sich der Client verbinden soll, gesetzt. Die Standardeinstellung ist *localhost*.

-rp Damit wird der Port des Servers, zu dem sich der Client verbinden soll, gesetzt. Die Standardeinstellung ist *15000*.

-c Mit diesem Argument kann man die Konfigurationsdatei dem Client übergeben, in der die allgemeinen Parameter für die Erstellung des Sichtbarkeitspolygons, Fehlerkorrektur u.a. gesetzt werden. Wird keine Datei übergeben, so versucht der Client die *polyexplore.cfg* im aktuellen Pfad zu finden. Ist keine Konfigurationsdatei vorhanden, werden die Standardwerte genommen.

-h Damit kann die Hilfe zum Starten des Clients aufgerufen werden.

6.2.3 Bedienung des Clients

Die Abbildung 6.1 zeigt den Aufbau des Clients. Im zentralen Bereich wird der Roboter und die aktuelle Karte angezeigt. Rechts hat das Fenster einen Bereich, in dem die aktuellen Informationen zu dem Roboter und der Karte angezeigt werden. Unten im Fenster befindet sich ein Schiebebalken mit dem man die Kartendarstellung vergrößern und verkleinern kann.

Der Roboter selbst wird durch ein Rechteck mit einem Pfeil dargestellt. Der Pfeil zeigt die Richtung des Roboter an. Die Karte wird durch grüne (hier hellgraue) Linien dargestellt. Der unerforschter Bereich wird in der Karte durch blaue (hier dunkelgraue) Linien markiert. Durch einen Klick mit der linken Maustaste auf eine Stelle in der Karte wird die Stelle in

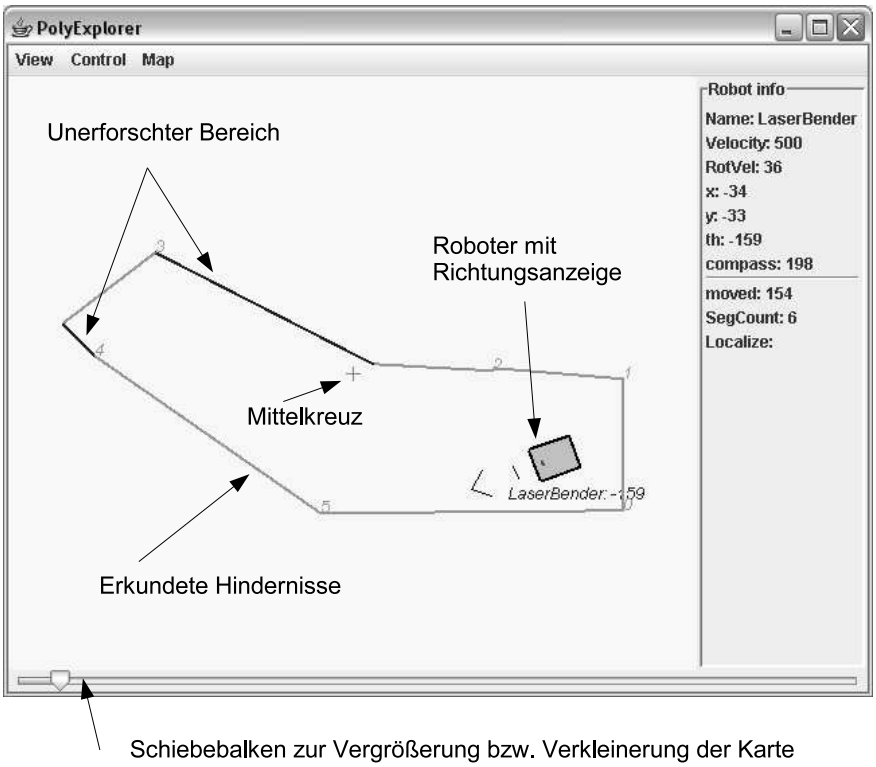

Abbildung 6.1: Die Darstellung des Fensters bei dem Client.

dem Fenster zentriert. Das Zentrum in dem Fenster wird durch ein Kreuz markiert.

Mit Hilfe des Client kann man den Roboter über drei unterschiedliche Modi steuern lassen. Alle drei Modi sind in über das Menü *Control* erreichbar.

Free Drive

In diesem Modus erscheinen in dem Informationsbereich Pfeiltasten mit denen der Roboter gesteuert werden kann. Dort kann man auch die maximale Geschwindigkeit für das Fahren und Drehen eingegeben werden. Alternativ kann der Roboter auch über die Tastatur mit den Tasten ALT+[A,W,S,D] gesteuert werden.

Target Drive

In diesem Modus wird der Roboter mit der Maus gesteuert. Dabei klickt man in auf die Stelle in der Karte, an die der Roboter fahren soll. Der Roboter begibt sich auf dem direkten Weg zu dieser Stelle. Man muss beachten, dass

es keine Kollisionskontrolle gibt.

PolyExplore

In diesem Modus erkundet der Roboter das Polygon nach der Strategie PolyExplore. Der Startpunkt ist dabei die Position, an der dieser Modus zum ersten Mal eingeschaltet wird. Man kann die Erkundung unterbrechen, indem man den Modus wieder ausschaltet. Wird der Modus zum zweiten Mal eingeschaltet, fährt der Roboter mit der Erkundung fort.

Zur Hilfe kann man weitere Informationen über das Menü *View* anzeigen lassen. Dazu zählen die Laserpunkte, das Sichtbarkeitspolygon, das aktuelle Ziel, der Shortest Path Tree usw., was auch für die Erkundung durch PolyExplore interessant sein kann.

Die aktuelle Karte kann man über das Menü *Map* als reine Textdatei oder als EPS[3] Graphik exportieren.

Konfigurationsdatei

Mit Hilfe der Konfigurationsdatei können die Standardparameter in PolyExplore geändert werden. Die Datei wird beim Starten dem Client übergeben. Die folgende Tabelle 6.1 zeigt die einzelnen Parameter, die eingestellt werden können.

[3] siehe http://partners.adobe.com/public/developer/ps/index_specs.html

Parameter	**Beschreibung**
BOUND_DIST	Der Abstand welchen der Roboter von der Wand einhalten soll.
EXPL_REACHED_DIST	Dieser Abstand wird benutzt, um festzustellen, ob der Roboter das Ziel erreicht hat.
VISP_MAX_DIST	Maximale Entfernung der Laserpunkte, die für die Berechnung des Sichtbarkeitspolygons berücksichtigt werden. Das ist besonders bei dem Simulator wichtig, da der Simulator sonst falsche Werte liefert.
VISP_MIN_POINTS_FOR_GAUSS	Die minimale Anzahl der Punkte, damit Gaußsche Methode der kleinsten Quadrate angewandt wird.
VISP_FIT_TOLERANCE	Ist der Anstand zwischen dem Punkt und der Gerade größer als dieser Wert, wird die Gerade aufgebrochen.
VISP_CLUSTER_THRESHOLD	Eine Gerade wird erst für das Sichtbarkeitspolygon erzeugt, wenn die Anzahl der Meßpunkte diesen Schwellwert überschreitet.
VISP_MAX_POINT_DIST	Ist der Abstand zwischen zwei Punkten größer, so werden diese nicht verbunden, auch wenn diese auf einer Geraden liegen.
MAP_FIT_TOLERANCE	Ist der Anstand zwischen dem Punkt und der Gerade kleiner als dieser Wert, wird die Gerade in zwei Gerade aufgebrochen.
MAP_MAX_POINT_DIST	Ist der Abstand zwischen den Punkten größer als dieser Wert, so werden diese nicht verbunden, auch wenn diese auf einer gerade liegen.
MERGE_MAX_ERROR_ANGLE	Ist der Winkel zwischen den beiden Kanten größer als dieser Wert, wird der Fehler nicht berücksichtigt.
MERGE_MAX_ERROR_DIST	Ist der Abstand zwischen zwei Kanten größer als dieser Wert, findet keine Fehlerberechnung zwischen den Kanten statt.
LOG_READINGS	Ist diese Variable gesetzt, so werden die Laser- und Positionsdaten in eine Datei geschrieben.

Tabelle 6.1: Die Parameter, welche in einer Datei dem Client übergeben werden können, um diesen zu konfigurieren.

Literaturverzeichnis

[1] H. Abelson and A. A. diSessa. *Turtle Geometry*. MIT Press, Cambridge, 1980.

[2] S. Albers, K. Kursawe, and S. Schuierer. Exploring unknown environments with obstacles. *10th ACM-SIAM Sympos. Discrete Algorithms*, page 842–843, 1999.

[3] U. Bachert. Erkundung unbekannter polygonaler Umgebungen. Diplomarbeit, FernUniversität Hagen, Fachbereich Informatik, 1997. http://www.geometrylab.de/SAM/.

[4] R. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Inform. Comput.*, 106:234–252, 1993.

[5] P. Berman. On-line searching and navigation. In A. Fiat and G. Woeginger, editors, *Competitive Analysis of Algorithms*. Springer-Verlag, 1998.

[6] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun. *Principles of Robot Motion*. MIT-Press, 2005.

[7] X. Deng, T. Kameda, and C. Papadimitriou. How to learn an unknown environment I: The rectilinear case. *J. ACM*, 45(2):215–245, 1998.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.

[9] M. Dror, A. Efrat, A. Lubiw, and J. S. B. Mitchell. Touring a sequence of polygons. In *Proc. 35th Annu. ACM Sympos. Theory Comput.*, pages 473–482, 2003.

[10] A. Fiat and G. Woeginger, editors. *On-line Algorithms: The State of the Art*, volume 1442 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1998.

[11] A. Garulli, A. Giannitrapani, A. Rossi, and A. Vicino. Simultaneous localization and map building using linear features. Technical report, Università di Siena, 2005.

[12] G. Grisetti, C. Stachniss, and W. Burgard. Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling. *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, page 2443 – 2448, 2005.

[13] L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *Proc. 3rd Annu. ACM Sympos. Comput. Geom.*, pages 50–63, 1987.

[14] R. Hagius. Untere Schranken für das Online-Explorationsproblem. Diplomarbeit, FernUniversität Hagen, Fachbereich Informatik, Mai 2002.

[15] R. Hagius, C. Icking, and E. Langetepe. Lower bounds for the polygon exploration problem. In *Abstracts 20th European Workshop Comput. Geom.*, pages 135–138. Universidad de Sevilla, 2004.

[16] D. Hähnel, W. Burgard, D. Fox, and S. Thrun. An efficient fastslam algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2003.

[17] F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. A competitive strategy for learning a polygon. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 166–174, 1997.

[18] F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. The polygon exploration problem ii: The angle hull. Technischer bericht 245, FernUniversität Hagen, 1998.

[19] F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. The polygon exploration problem. *SIAM J. Comput.*, 31:577–600, 2001.

[20] C. Icking, R. Klein, and L. Ma. How to look around a corner. *In Proc. 5th Canad. Conf. Comput. Geom.*, pages 443–448, 1993.

[21] T. Kamphans, R. Klein, and E. Langetepe. Offline Bewegungsplanung für Roboter. Vorlesungsskript, Universität Bonn, Institut für Informatik, 2001.

[22] R. Klein. *Algorithmische Geometrie*. Springer Verlag, 2005.

[23] L. J. Latecki, R. Lakaemper, X. Sun, and D. Wolter. Building polygonal maps from laser range data. *ECAI Int. Cognitive Robotics Workshop, Valencia, Spain*, 2004.

[24] T. Lozano-Pérez. Spatial planing: A configuration space approach. *IEEE Trans. Comput.*, 32:108–120, 1983.

[25] V. J. Lumelsky and A. A. Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.

[26] J. S. B. Mitchell. Geometric shortest paths and network optimization. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 633–701. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[27] H. Moravec and A. Elfes. High resolution maps from wide angle sonar. *IEEE International Conference on Robotics and Automation*, 1985.

[28] V. Nguyen, A. Martinelli, N. Tomatis, and R. Siegwart. A comparison of line extraction algorithms using 2D laser rangefinder for indoor mobile robotics. *Proceedings of the IEEE/RSJ Intenational Conference on Intelligent Robots and Systems, IROS'2005*, 2005.

[29] T. Pavlidis and S. Horowitz. Segmentation of plane curves. *IEEE Transactions on Computers*, C-23(8):860–870, 1974.

[30] S. T. Pfister, S. I. Roumeliotis, and J. W. Burdick. Weighted line fitting algorithms for mobile robot map building and efficient data representation. *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference*, pages 1304– 1311, 2003.

[31] N. S. Rao, S. Kareti, W. Shi, and S. Iyengar. Robot navigation in unknown terrains: Introductory survey of non-heuristic algorithms. Technical Report ORNL/TM-12410, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, July 1993.

[32] C. E. Shannon. Presentation of a maze solving machine. In H. von Foerster, M. Mead, and H. L. Teuber, editors, *Cybernetics: Circular, Causal and Feedback Mechanisms in Biological and Social Systems, Transactions Eighth Conference, 1951*, pages 169–181, New York, 1952. Josiah Macy Jr. Foundation.

[33] SICK. *Lasermesssysteme LMS 200 / LMS 211 / LMS 220 / LMS 221 / LMS 291*, 2003.

[34] M. Veeck and W. Burgard. Learning polyline maps from range scan data acquired with mobile robots. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.

[35] www.wikipedia.de.