

Projektbericht

PFADPLANUNG FÜR EINEN MOBILEN AUTONOMEN ROBOTER

Verfasser / Matrikelnummer:
Fink Thomas/ S0910564041

Projektleiter:
Dipl.-Ing.(FH) Raimund Edlinger

Abgabedatum:
26.08.2010

Inhaltsverzeichnis

1	Aufgabenstellung und Zielsetzung	2
2	Algorithmus	2
2.1	Übersicht	2
2.2	Wichtige Methoden	3
2.2.1	calculate_path	3
2.2.2	computepath	8
2.3	Weitere Methoden	12
2.3.1	Caldistancetopoint	12
2.3.2	Caldisttowall	13
2.3.3	Calneighbor	13
2.3.4	Medpoints	14
3	Messergebnisse	15
4	Fazit und Ausblick	16
5	Literaturverzeichnis	17

Abbildungsverzeichnis

Abbildung 1 (Beispiel Grid Map)	2
Abbildung 2 (Übersicht Algorithmus)	3
Abbildung 3 (Übersicht calculate_path)	4
Abbildung 4 (Koordinatensystem)	5
Abbildung 5 (Zielpunktsuche)	5
Abbildung 6 (Zielpunktsuche)	7
Abbildung 7 (Codeübersicht Zielpunktsuche)	7
Abbildung 8 (Codeübersicht computepath)	9
Abbildung 9 (Messergebnisse Pfadplanung)	15

1 Aufgabenstellung und Zielsetzung

Dieses Projekt befasst sich mit der Erstellung eines Algorithmus für die Pfadplanung. Die Pfadplanung ist die Berechnung eines optimalen Weges von der aktuellen Position des Roboters zu einem Gebiet, welches noch nicht exploriert wurde. Diese Pfadplanung soll die autonome Fortbewegung eines Roboters der Rescue League ermöglichen. Für die Entwicklung des Algorithmus stehen als Eingangsgrößen die aktuelle Position und die sogenannten Occupancy Grid Maps [Kneidinger, 2009],[Thrun, 2005] zur Verfügung. Diese Grid Maps werden mit Hilfe eines 2D-Laserscanners erstellt und stellen eine Wahrscheinlichkeitsverteilung von Hindernissen im Raum dar. Dabei repräsentiert der Wert jedes Pixels die Wahrscheinlichkeit für ein Hindernis. Ein grauer Wert entspricht einer Wahrscheinlichkeit für ein Hindernis von 50% ($P=0.5$). Ein weißer Bereich hingegen würde bedeuten, dass kein Hindernis vorhanden ist. Bei einem schwarzen Bereich ist jedoch mit großer Wahrscheinlichkeit ein Hindernis vorhanden. Eine Grid Map, die in der Testarena der FH-Wels erstellt wurde, ist in Abbildung 1 dargestellt.

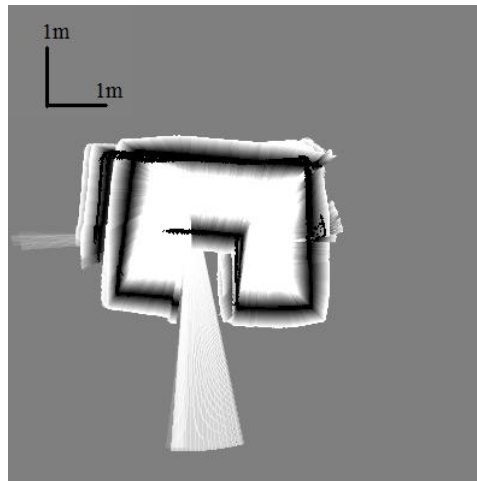


Abbildung 1 (Beispiel Grid Map)

Ziel dieses Projektes ist es, einen Algorithmus zu entwickeln, der aus diesen Grid Maps mögliche Zielpunkte bestimmt und mit Kenntnis der Roboterposition in der Karte den dazugehörigen Pfad berechnet. Dieser Algorithmus soll in das bereits bestehende Mappingprogramm für die Aufzeichnung der Karte eingebaut werden.

2 Algorithmus

In diesem Abschnitt wird der entwickelte Algorithmus behandelt. Die Codefragmente werden mit Hilfe von Flussdiagrammen dargestellt. Weiters werden die wichtigsten Methoden näher beschrieben.

2.1 Übersicht

Wie bereits im vorherigen Punkt erwähnt, ist die Ausgangssituation für den Pfadplanungsalgorithmus eine bestehende Karte der Umgebung sowie die aktuelle Position des Roboters in dieser Karte. Den prinzipiellen Ablauf des Algorithmus stellt Abbildung 2 dar. Dabei kann der Algorithmus in zwei nacheinander folgenden Sequenzen unterteilt werden. Zuerst werden

aus der Karte die möglichen Zielpunkte berechnet. Zu diesen Zielpunkten werden anschließend die einzelnen Pfade berechnet.

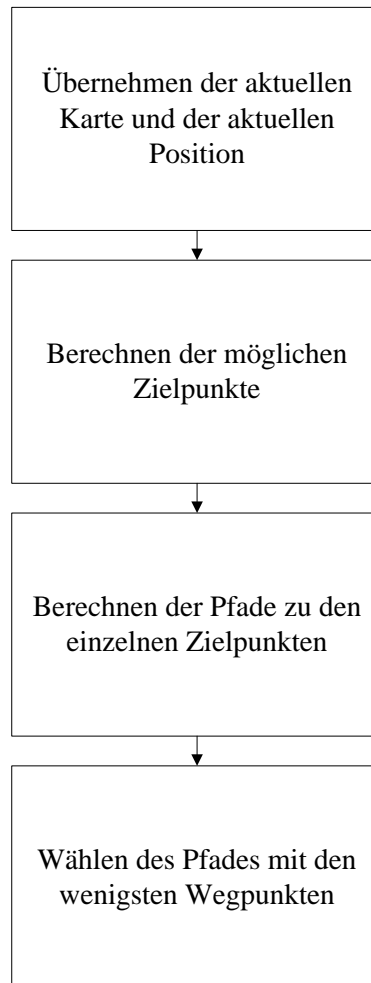


Abbildung 2 (Übersicht Algorithmus)

2.2 Wichtige Methoden

Die beiden wichtigsten Methoden sind `calculate_path` und `computepath`, wobei die zweite Methode in der ersten Methode aufgerufen wird.

2.2.1 `calculate_path`

Diese Methode berechnet die möglichen Zielpunkte und ruft die Methode `computepath` auf um die Wegpunkte zu den Zielpunkten zu berechnen.

Aufruf der Methode:

```
void calculate_path(ts_map_t *map, ts_position_t &position, vector<vector<point>> &patharray)
```

1. Parameter: Übernimmt die aktuelle Karte
2. Parameter: Übernimmt die aktuelle Position des Roboters(Mapping-Koordinaten)
3. Parameter: Nach Ausführung dieser Methode sind darin die einzelnen Zielpunkte und deren Wegpunkte enthalten.

Die einzelnen Schritte, die diese Methode ausführt, sind in der nachfolgenden Grafik veranschaulicht:

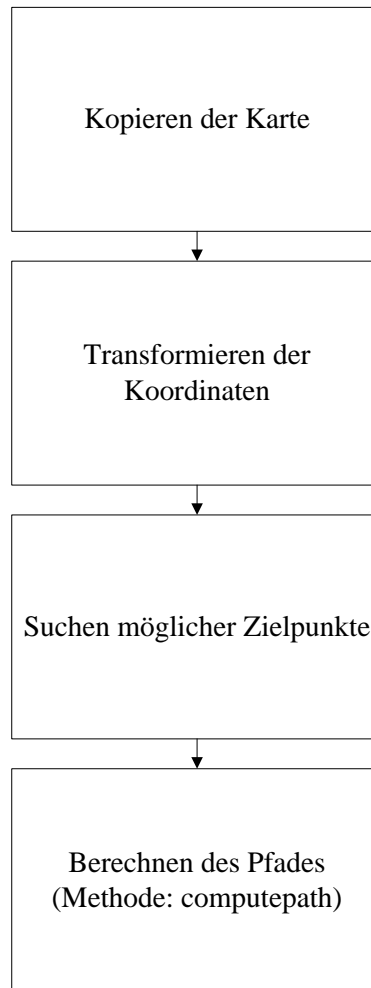


Abbildung 3 (Übersicht calculate_path)

Die erste Aktion dieser Methode ist das Kopieren der Karte und das Transformieren der Koordinaten. Dabei werden die eindimensionalen Daten der Karte des Mapping-Algorithmus auf ein zweidimensionales Array (Karte: „mappath“) kopiert. Weiters werden die Koordinaten[mm] des Mapping-Algorithmus in ein Bild Koordinatensystem[Pixel] transformiert. Dies erfolgt durch das nachfolgende Codefragment.

```
//Kopieren der Karte map[1d] auf mappath[2d]
for( int i=0; i<TS_MAP_SIZE; i++)
{
    for( int j=0; j<TS_MAP_SIZE; j++)
    {
        mappath[j][i] = (unsigned char)(map->map[n]);
        n++;
    }
}
//Konvertieren der Koordinaten
position.x=position.x/20;
position.y=position.y/20;
```

Das verwendete Koordinatensystem für diesen Algorithmus ist in der nachfolgenden Abbildung 4 dargestellt.

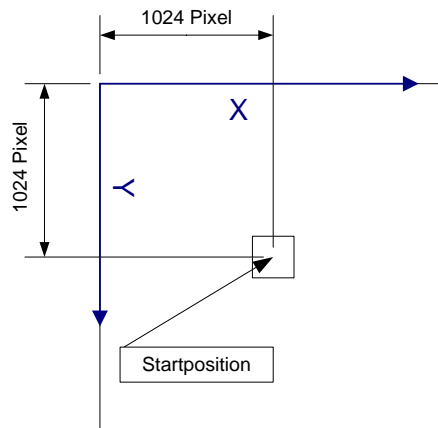


Abbildung 4 (Koordinatensystem)

Nach dem Kopieren der Karte und dem Transformieren der Koordinaten müssen nun die möglichen Zielpunkte gesucht werden. Wie oben beschrieben, wird als Ausgangspunkt für die Berechnung der Zielpunkte die Grid-Map herangezogen. Dabei besitzen die Grauwerte der einzelnen Pixel folgende Bedeutung:

Pixelwert	P... Wahrscheinlichkeit für ein Hinderniss	Farbe
0	1	Schwarz
127	0,5(nicht exploriert)	Grau
255	0	Weiß

Gesucht werden nun Pixel, die noch nicht exploriert sind, deren Nachbarn eine sehr geringe Wahrscheinlichkeit für ein Hindernis aufweisen und in deren Nähe sich keine Wand befindet. Diese gesuchten Pixel sind entlang der grau-weißen Kante in der rot eingekreisten Zone dargestellt, siehe Abbildung 5.

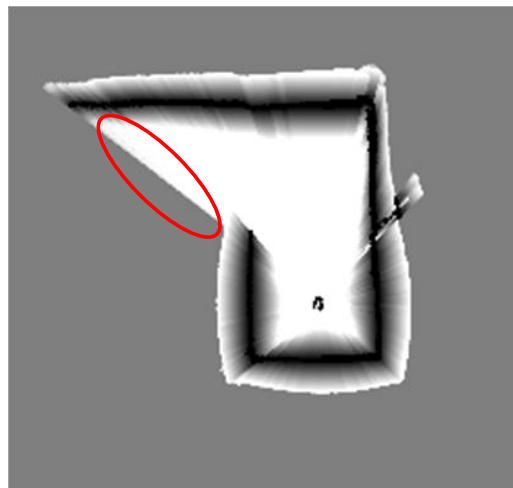


Abbildung 5 (Zielpunktsuche)

Die Suche nach diesen Pixel wird mit Hilfe des folgenden Codes durchgeführt:

```
//Farbenzuordnung für die folgenden Berechnungen
uchar weiss=255;
uchar grau=127;
uchar schwarz=0;
uchar weissgrau=200;
uchar wall=50;
```

```

//Suche Pixel die grau sind(noch nicht exploriert) und deren Nachbarn angefahren
//werden können(weissgrau, bereits exploriert und kein Hindernis vorhanden)
//-> Übergang P=0 auf P=0.5
for(int x=ab-1; x<TS_MAP_SIZE-ab; x++)
{
    for(int y=ab-1; y<TS_MAP_SIZE-ab; y++)
    {
        if(mappath[x][y] == grau &&
            (mappath[x+1][y+1]>weissgrau||
             mappath[x+1][y]>weissgrau||
             mappath[x+1][y-1]>weissgrau||
             mappath[x][y+1]>weissgrau||
             mappath[x][y]>weissgrau||
             mappath[x][y-1]>weissgrau||
             mappath[x-1][y+1]>weissgrau||
             mappath[x-1][y]>weissgrau||
             mappath[x-1][y-1]>weissgrau))
        {
            ll=0;
            //Es wird gesucht, ob sich im Umkreis(Abstand ab) eines Punktes
            //eine Wand befindet, wenn nicht wird der Punkt für die
            //weiteren Berechnungen verwendet
            for (int winkel=0;winkel<40;winkel++)
            {
                wi=(double)2*PI*winkel/40;
                rx=ab*sin(wi);
                ry=ab*cos(wi);
                ll++;

                if(mappath[x+(int)rx][y+(int)ry] <= wall)//Threshold um Wand zu
                                                            //erkennen
                {
                    break;
                    ll=0;
                }
                if(ll==40)
                {
                    //Ist keine Wand vorhanden wird der Punkt in das
                    //Array "pointsforpatharray" aufgenommen
                    paktuell.x=x;
                    paktuell.y=y;
                    pointsforpatharray.push_back(paktuell);
                    ll=0;
                }
            }
        }
    }
}

```

Daraus ergeben sich im Array „pointsforpatharray“ die möglichen Punkte. Aus diesen Punkten muss nun ein Zielpunkt, mit Hilfe einer Mittelwertbildung, berechnet werden. Dazu wird aus den einzelnen x und y Koordinaten der Punkte ein Median gebildet, um grobe Ausreißer zu unterdrücken.

Sind mehrere Zielpunkte möglich, wie in Abbildung 6 dargestellt, muss auch dies berücksichtigt werden.

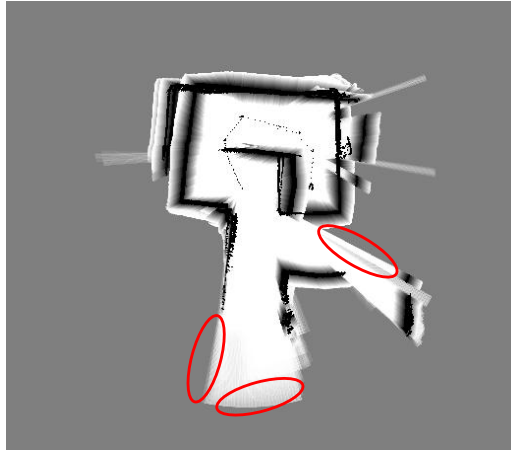


Abbildung 6 (Zielpunktsuche)

Die prinzipiellen Schritte, die aus den möglichen Punkten Zielpunkte berechnen, sind in der nachfolgenden Abbildung 7 dargestellt.

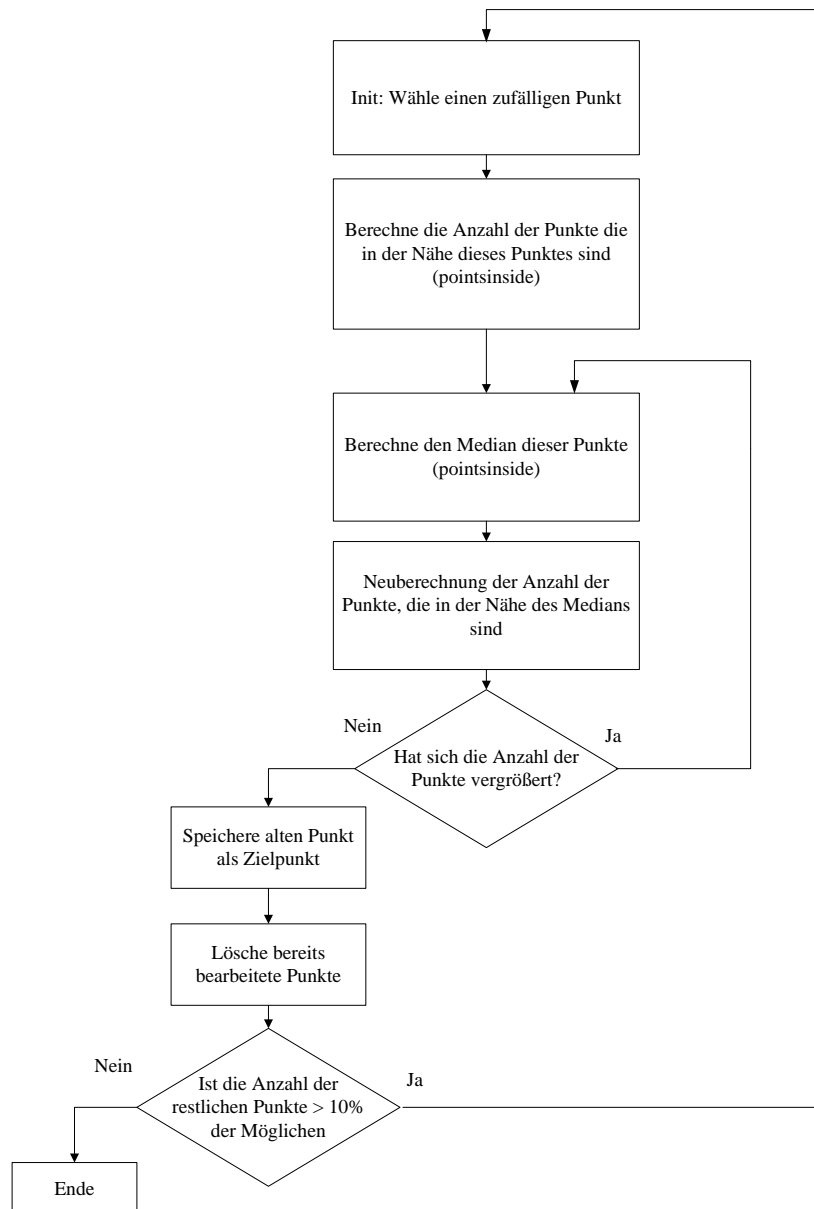


Abbildung 7 (Codeübersicht Zielpunktsuche)

Der vollständige Algorithmus ist im nachfolgenden dargestellt.

```

temppoints=pointsforpatharray;
vector<point> pointsinside;
vector<point> pointsoutside;
vector<point> aimpoints;
int randompoint;

while((temppoints.size())>=pointsforpatharray.size()/10)//Solange die Anzahl der restlichen
//Punkte größer 10% der
//Gesamtanzahl ist
{

    randompoint=(rand()%temppoints.size());
    point aktuellpoint=temppoints[randompoint]; //Wählen eines zufälligen Punktes

    int countafter=1;
    int countfirst=0;
    int count=0;

    //Es wird der Mittelpunkt des Kreises solange verschoben(Median) bis eine
    //maximale Anzahl erreicht wird
    while(countafter>countfirst)//Median bis die maximale Anzahl innerhalb des Kreises
    //erreicht wird
    {
        countfirst=count;

        //Berechnet die Anzahl der Punkte die innerhalb(pointsinside) und
        //außerhalb(pointsoutside) eines Kreises(r=distancepointneighbor) liegen
        caldistancetopoint(aktuellpoint, temppoints, distancepointneighbor, count,
            pointsinside,pointsoutside);

        countafter=count;
        point medpoint;

        //Berechnet den Median der Punkte innerhalb des Kreises
        medpoints(pointsinside, medpoint); //Medianfilter

        aktuellpoint=medpoint;
    }

    aimpoints.push_back(aktuellpoint);//Einfügen der Punkte in das Array "aimpoints"
    temppoints=pointsoutside;
}

```

Nach diesem Algorithmus werden die Zielpunkte in das Array „aimpoints“ gespeichert.

2.2.2 computepath

Nach dem die Zielpunkte gefunden worden sind, kann nun der Pfad geplant werden. Dies erfolgt mit dem Aufruf der Methode:

```

void computepath(vector<vector<unsigned char>> &map, vector<point> &aimpoints,
    ts_position_t &position, vector<vector<point>> &patharray)

```

1. Parameter: Übernimmt die aktuelle Karte
2. Parameter: Übernimmt das Array in dem die Zielpunkte gespeichert sind
3. Parameter: Übernimmt die aktuelle Position des Roboters
4. Parameter: Gibt die einzelnen Wegpunkte zu den gefundenen Zielpunkten

Wie der prinzipielle Code dieser Methode aussieht kann mit Hilfe der nachfolgenden Abbildung 8 nachvollzogen werden.

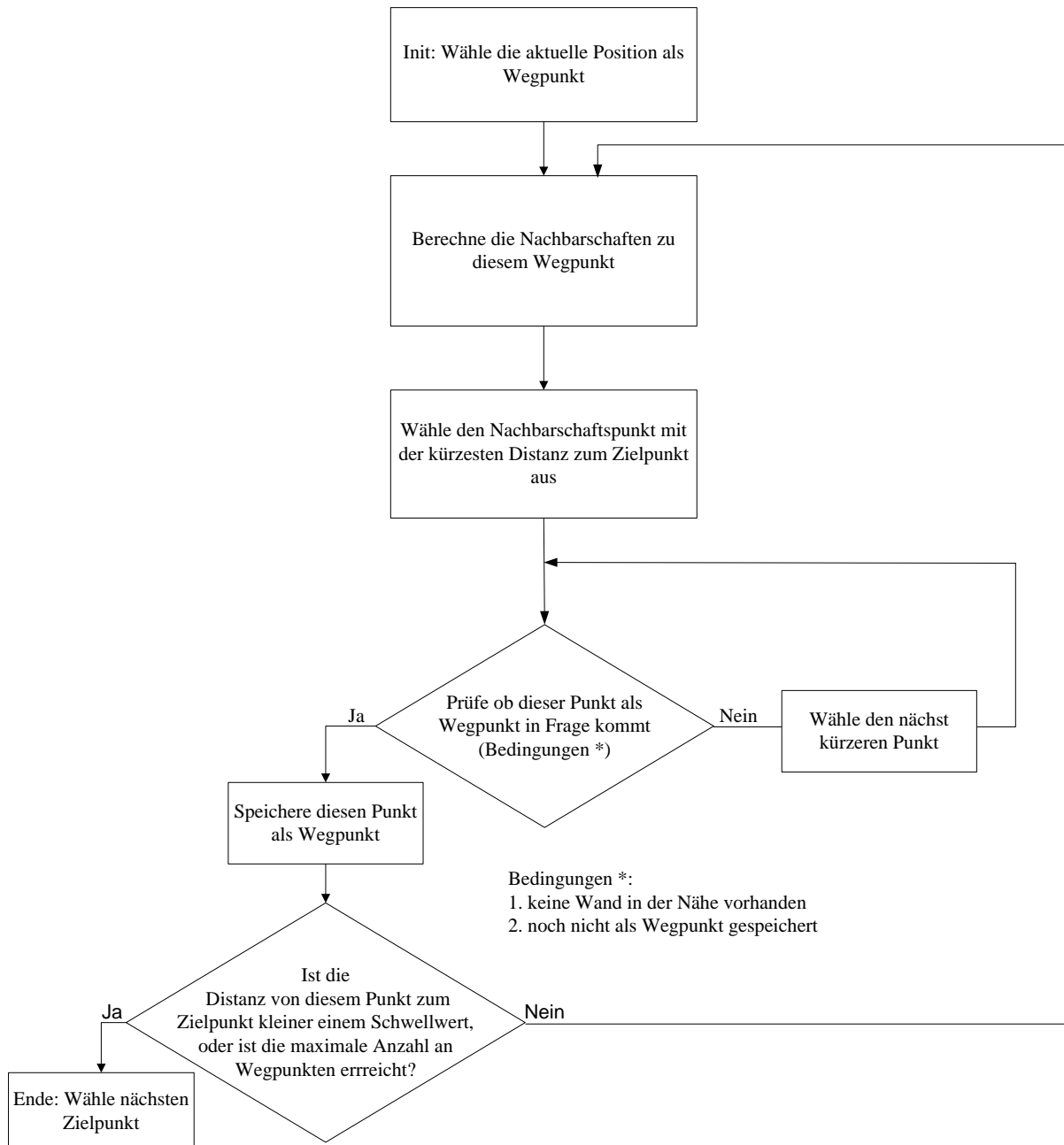


Abbildung 8 (Codeübersicht computepath)

Der vollständige Code für die Planung der Wegpunkte zu den Zielpunkten ist nachfolgend dargestellt:

```

int distneighbor=1;
int disttowall=15; //minimaler Abstand zwischen Roboterposition und Wand
point tempoint;

int maxpoints=500; //maximale Anzahl der Punkte die für einen Pfad möglich sein dürfen
vector<pointneighbor> staticpointneighborarray(8);
vector<pointneighbor> pointneighborarray;
vector<point> path; //Punkte des aktuellen Pfades
vector<point> nopoints; //Punkte die nicht befahren werden können
bool iswall=false; //ob eine Wand in der Nähe des Punktes ist
patharray.resize(aimpoints.size());
  
```

```
for(int i=0;i<aimpoints.size();i++)
{
    //Initialisierung
    int j=0; //Anzahl der While-Schleifendurchläufe
    int g=0;
    int m=0;
    temppoint.x=position.x;
    temppoint.y=position.y;

    iswall=false;
    nopoints.clear();
    path.clear();

    //Solange die maximale Anzahl der Punkte erreicht worden ist,
    //oder die Distanz zum Zielpunkt klein genug ist
    while(j<maxpoints)
    {
        //Berechnet die Nachbarschaften zum aktuellen Punkt
        calneighbor(temppoint,disttoneighbor,aimpoints[i],staticpointneighborarray);
        j++;
        pointneighborarray.clear();
        for(int k=0; k<staticpointneighborarray.size();k++)
        {
            m=0;
            for(int l=0;l<nopoints.size();l++)
            {
                //Prüfung ob sich ein Punkt der Nachbarschaft im Array "nopoints"
                //befindet

                if(!((staticpointneighborarray[k].x==nopoints[l].x)&&
                    (staticpointneighborarray[k].y==nopoints[l].y)))
                {
                    m=1;
                }
                else
                {
                    m=0;
                    break;
                }
            }

            //Hinzufügend des Nachbarschaftspunktes zum "pointneighborarray" wenn
            //er nicht in Array "nopoints" ist
            if(m==1)
            {
                pointneighborarray.push_back(staticpointneighborarray[k]);
            }

            //Startbedingung
            if(nopoints.size()==0)
            {
                pointneighborarray=staticpointneighborarray;
            }
        }
    }
}
```

```
//Sortieren der Punkte des "pointneighborarray" nach der Größe der Distanz
//vom Punkt zu Ziel (Index[0]...kürzeste Distanz)
sort(pointneighborarray.begin(),pointneighborarray.end());

//Prüfung ob sich in der Nähe des Punktes eine Wand befindet
iswall=caldisttowall(disttowall,pointneighborarray[0],map);

if(iswall==false)
{
    //Ist keine Wand vorhanden wird der Punkt
    //zum Pfad("path" Array) hinzugefügt
    //und weiters zum Array "nopoints"
    temppoint.x=pointneighborarray[0].x;
    temppoint.y=pointneighborarray[0].y;
    path.push_back(temppoint);
    nopoints.push_back(temppoint);
}
else
{
    int neighborsize=pointneighborarray.size();
    //Solange eine Wand vorhanden ist
    while(iswall==true)
    {
        //Befinden sich nur Wände in der Nachbarschaft so wird der Punkt
        //zum Array "nopoints" hinzugefügt
        if(g==neighborsize-1)
        {
            nopoints.push_back(temppoint);
            g=0;
            break;
        }
        //Entfernt immer den ersten Eintrag des Arrays
        //"pointneighborarray"
        //solange bis keine Wand mehr vorhanden ist
        pointneighborarray.erase(pointneighborarray.begin());
        iswall=caldisttowall(disttowall,pointneighborarray[0],map);
        g++;
    }
    //Speichern des Punktes
    g=0;
    temppoint.x=pointneighborarray[0].x;
    temppoint.y=pointneighborarray[0].y;
    path.push_back(temppoint);
    nopoints.push_back(temppoint);
}
if(pointneighborarray[0].distance<=5)
{
    break;//Abbruchbedingung der While-Schleife
}
}
patharray[i]=path;
}
```

Die einzelnen Wegpunkte sind im Array „patharray“ enthalten. Sind alle Wegpunkte zu den Zielpunkten berechnet, kann der Weg mit den wenigsten Wegpunkten als neuer Pfad gewählt werden.

2.3 Weitere Methoden

In den Algorithmen kommen verschiedene Methoden vor, die im Folgenden noch näher beschrieben werden:

2.3.1 Caldistancetopoint

Diese Methode berechnet die Anzahl der Punkte, die innerhalb eines Bereichs sind. Dieser Bereich ist ein Kreis mit dem Radius „distancepointneigbor“ und dem Mittelpunkt „randompoint“. Die Punkte, die innerhalb dieses Bereichs sind, werden in dem Array „pointsinside“ gespeichert, die die außerhalb liegen werden in dem Array „pointsoutside“ gespeichert. Der Code dafür ist im Folgenden dargestellt.

```
void caldistancetopoint(point &randompoint,
                        vector<point> &temppoints,
                        int &distancepointneigbor,
                        int &count,
                        vector<point> &pointsinside,
                        vector<point> &pointsoutside)
{
    count=0;
    double dx, dy;//Abstände vom aktuellen zum random Punkt
    double xrandompoint=randompoint.x;
    double yrandompoint=randompoint.y;
    pointsinside.clear();
    pointsoutside.clear();

    for(int i=0; i<temppoints.size(); i++)
    {
        dx=temppoints[i].x-xrandompoint;
        dy=temppoints[i].y-yrandompoint;
        if(sqrt(pow(dx,2)+pow(dy,2))<distancepointneigbor)
        {
            pointsinside.push_back(temppoints[i]);
            count++;
        }
        else
        {
            pointsoutside.push_back(temppoints[i]);
        }
    }
}
```

2.3.2 Caldisttowall

Berechnet, ob sich eine Wand in der Nähe des Punktes "point" befindet. Überprüft wird dabei, ob am Umfang des Kreises ($r = \text{distantowall}$) ein Wert für eine hohe Hinderniswahrscheinlichkeit (schwarz in der Karte) auftritt. Ist eine Wand vorhanden, wird die boolesche Variable `iswall` auf `true` gesetzt:

```
bool caldisttowall(int disttowall, pointneighbor point, vector<vector<unsigned char>> mappath)
{
    bool iswall=false;
    double ab, wi, rx, ry;
    ab=(double) disttowall;
    for (int winkel=0;winkel<40;winkel++)
    {
        wi=(double)2*PI*winkel/40;
        rx=ab*sin(wi);
        ry=ab*cos(wi);
        if(mappath[point.x+(int)rx][point.y+(int)ry] > 125)//Threshold um Wand zu
        //erkennen
        {
            iswall=false;
        }
        else
        {
            iswall=true;
            break;
        }
    }
    return iswall;
}
```

2.3.3 Calneighbor

Berechnet die 7 Nachbarschaften eines Punktes sowie deren Abstände zum Zielpunkt. Gespeichert werden diese Nachbarschaften im Array „pointneighborarray“:

```
void calneighbor(point &temppoint, int &distancetorneighbor, point &aimpoint, vector<pointneighbor> &pointneighborarray)
{
    int m=0;
    for(int i=-1;i<=1;i++)
    {
        for(int j=-1;j<=1;j++)
        {
            if(j!=0||i!=0)
            {
                pointneighborarray[m].x=temppoint.x+distancetorneighbor*j;
                pointneighborarray[m].y=temppoint.y+distancetorneighbor*i;
                pointneighborarray[m].distance=(int)
                pow(aimpoint.x-(temppoint.x+distancetorneighbor*j),2.0)+
                pow(aimpoint.y-(temppoint.y+distancetorneighbor*i),2.0);
                m++;
            }
        }
    }
}
```

Um später das Array „pointneighborarray“ den Distanzen entsprechend zu sortieren, muss der Vergleichsoperator „<“ überladen werden. Dies geschieht mit folgendem Codefragment.

```
bool operator< (const pointneighbor &a, const pointneighbor &b)
{return(a.distance<b.distance);}
```

2.3.4 Medpoints

Berechnet den Median der Punkte „pointsinside“ und speichert dies auf den Punkt „randompoint“. Dieser Medianfilter ist nicht streng nach Definition aufgebaut, da er bei einer geraden Anzahl von N Einträgen im Array „pointsinside“ den Wert $N/2-1$ als Median zurückgibt. Weiters wird der Median für die x und y Werte getrennt berechnet.

```
void medpoints(vector<point> &pointsinside, point &randompoint)
{
    vector<int> xarray;
    vector<int> yarray;
    for(int i=0; i<pointsinside.size();i++)
    {
        xarray.push_back(pointsinside[i].x);
        yarray.push_back(pointsinside[i].y);
    }

    sort(xarray.begin(),xarray.end());
    sort(yarray.begin(),yarray.end());

    randompoint.x=xarray[(int)xarray.size()/2];
    randompoint.y=yarray[(int)yarray.size()/2];
}
```

3 Messergebnisse

Im Folgenden wird der Algorithmus auf verschiedene Karten angewendet, wobei die Wegpunkte farblich eingezeichnet wurden. Diese Karten wurden in einer nachgebauten Rescue Arena in der FH Wels erstellt.

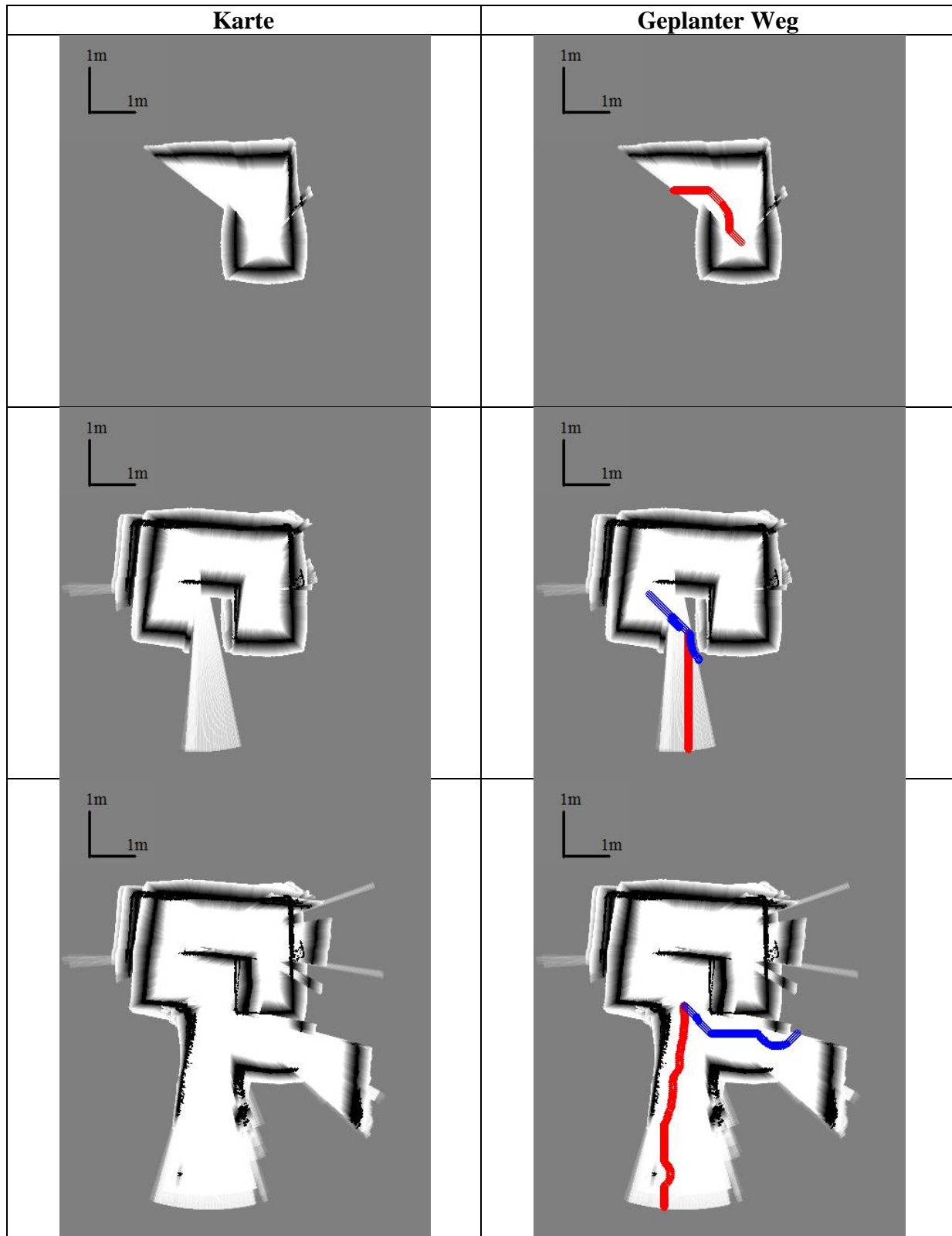


Abbildung 9 (Messergebnisse Pfadplanung)

4 Fazit und Ausblick

Mit Hilfe von mehreren Tests und Versuchen wurde gezeigt, dass der Algorithmus stabil läuft. Dies ist auch aus Messergebnissen im vorherigen Abschnitt ersichtlich. Voraussetzung für diese Stabilität ist eine Karte mit sehr geringen Fehlern. Zur Vorbeugung dieser Fehler in der Karte darf der Roboter durch das Gelände nur sehr langsam bewegt werden. Speziell bei Drehungen des Roboters, kann dessen Orientierung nicht immer zufriedenstellend berechnet werden und so treten Fälle auf, die mit dem Pfadplanungsalgorithmus nicht behandelt werden können.

Ein Potential für Fehler kann der Start des Roboters mit sich bringen. Dabei weiß der Roboter kurz nach dem Einschalten noch nichts bzw. sehr wenig über seine Umgebung. Daraus ergibt sich auch, dass zu diesem Zeitpunkt eine Pfadplanung noch nicht stattfinden kann. Als Abhilfe dafür kann ein Befehl an den Roboter sein, zuerst eine 360° Drehung durchzuführen, um die unmittelbare Umgebung zu scannen, und erst danach die Pfadplanung durchzuführen. Hat der Roboter nach einer gewissen Zeit ein größeres Gebiet exploriert, treten immer mehr mögliche Zielpunkte auf. Dies lässt sich auf die zunehmenden Fehler durch das Mapping und auf die größere Karte zurückführen. Sind mehrere mögliche Zielpunkte vorhanden ist auch mit einer Zunahme der Berechnungsdauer für den Pfad zu rechnen.

Zur Verkürzung der Berechnungsdauer kann versucht werden, in einem nächsten Schritt den Code recheneffizienter zu programmieren. Mit dieser Version des Codes wurde versucht ein nachvollziehbares Programm zu erstellen, welches für zukünftige Anwendungen als Ausgangspunkt herangezogen werden kann.

Eine weitere mögliche Verbesserung, welche sich in der Praxis bewähren könnte, ist es nicht den kürzesten Weg als Pfad zu benutzen, sondern jenen Weg, bei dem der Roboter die wenigsten und kleinsten Drehungen ausführen muss. Dies würde den Vorteil mit sich bringen, dass bei kleineren Drehungen der Fehler, der dabei auftritt, minimiert wird.

Im nächsten Schritt können die berechneten Wegpunkte als Eingangsgrößen für die Steuerung herangezogen werden. Dazu wird eine gewisse Anzahl von Wegpunkten ausgewählt. Zwischen den jeweiligen Wegpunkten wird zuerst ein Steuerbefehl für die Drehung an den Roboter geschickt und danach ein Steuerbefehl für die Vorwärtsbewegung. So gelangt der Roboter von einem Wegpunkt zum nächsten, bis er an dem Zielpunkt angekommen ist und ein neuer Pfad berechnet werden muss.

5 Literaturverzeichnis

[Thrun, 2005] Sebastian Thrun, Wolfram Burgard, Dieter Fox: „Probabilistic robotics“, MIT Press, 2005

[Kneidinger 2009] Harald Kneidinger, „2D-Mapping für einen Roboter der Rescue League“, Bachelorarbeit, 2009