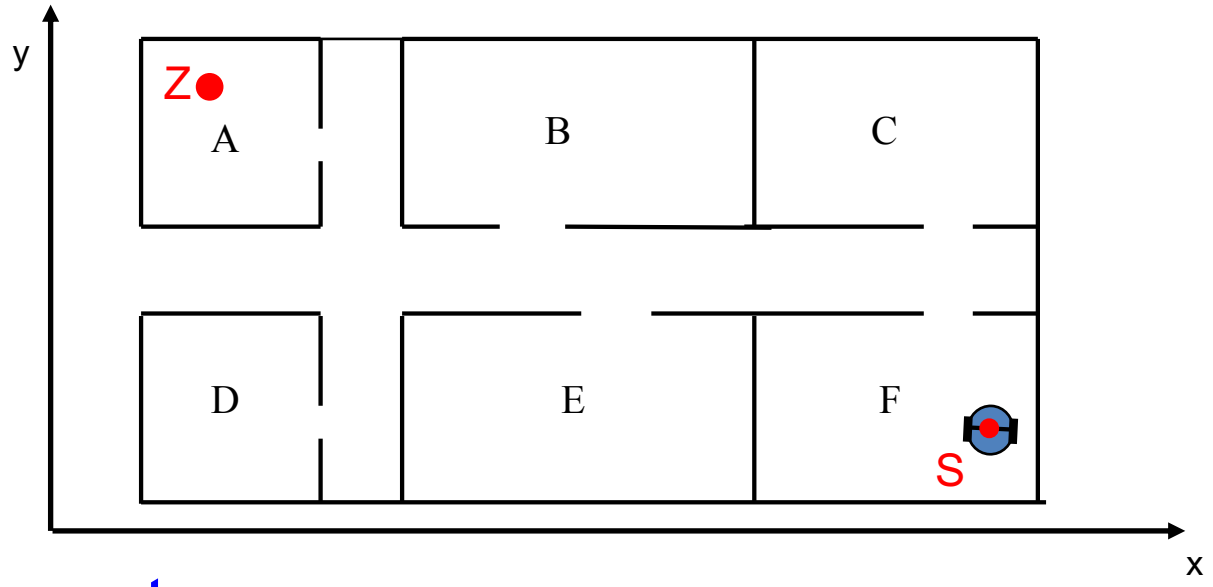
Navigation

- Überblick
  - Ziel, Architektur, Komponenten
- Reaktive Navigation
  - Insekten als Vorbild
  - Braitenberg-Vehikel
  - Zusammenbau elementarer Navigationsbausteine
  - Bug-Algorithmen
  - Histogramm-Verfahren
  - Dynamik-basiertes Verfahren
- Kartenbasierte Navigation
  - Kürzeste Wege in Graphen:
    - A*, Dijkstra-Verfahren und Breitensuche
  - Verfahren mit Belegtheitsgitter
  - Wegekartenverfahren:
    - Voronoi-Diagramme und probabilistische Wegekarten

Problemstellung

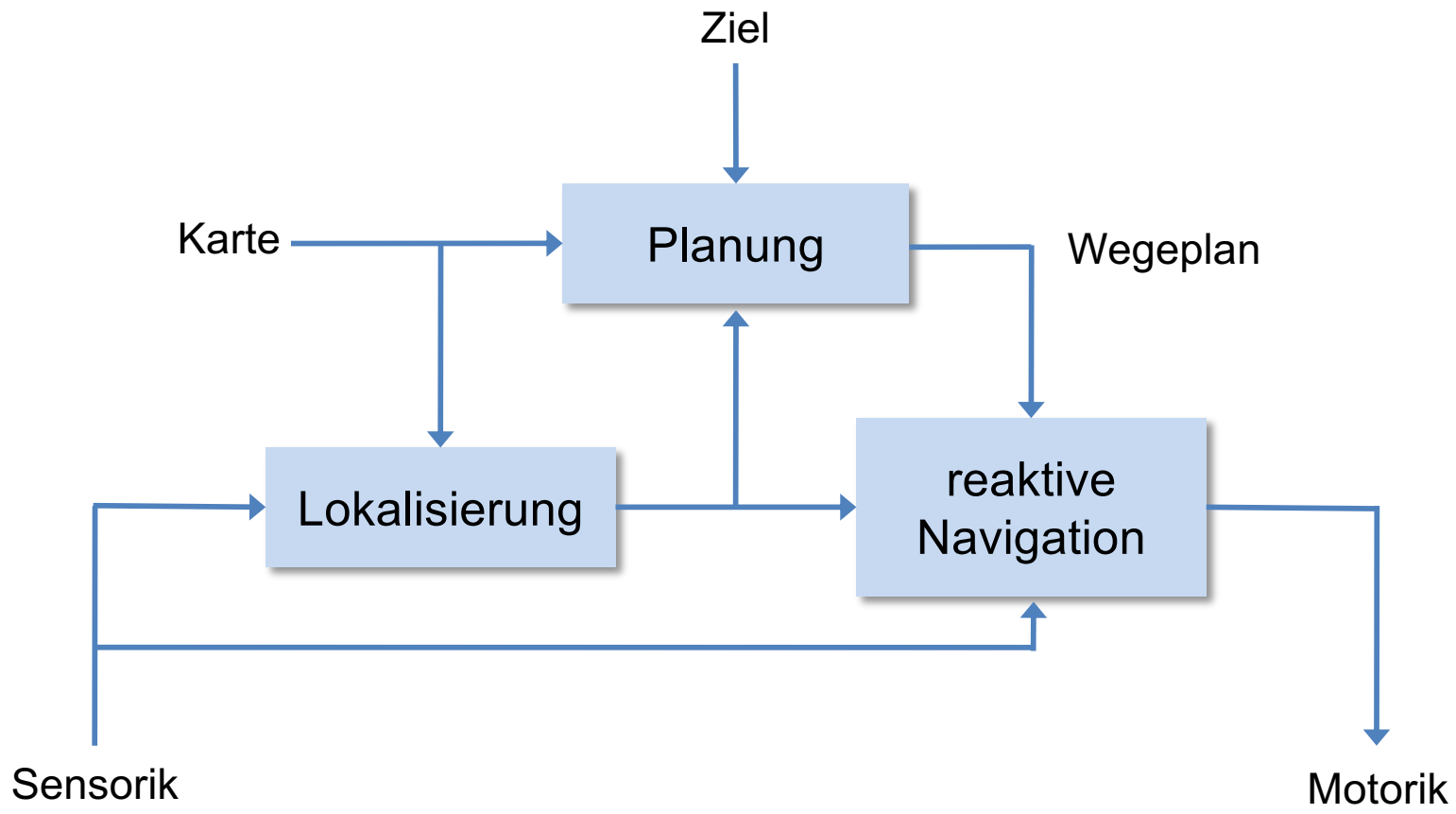
- Bewege Roboter kollisionsfrei von einem Startpunkt S zu einem Zielpunkt Z.



Problemparameter:

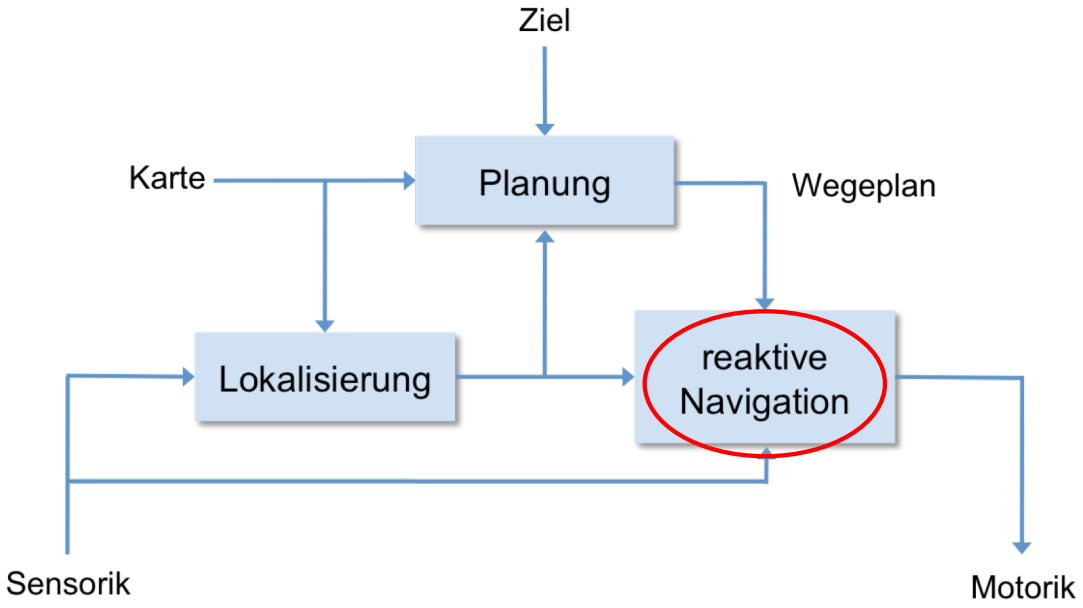
- Roboter:
  - Ausmaße und dynamische Eigenschaften
- Umgebung:
  - Beschaffenheit:
    In/Outdoor, künstlich/natürlich, Land/Wasser, ...
  - dynamische Hindernisse
  - Karte

Typische Architektur und Komponenten



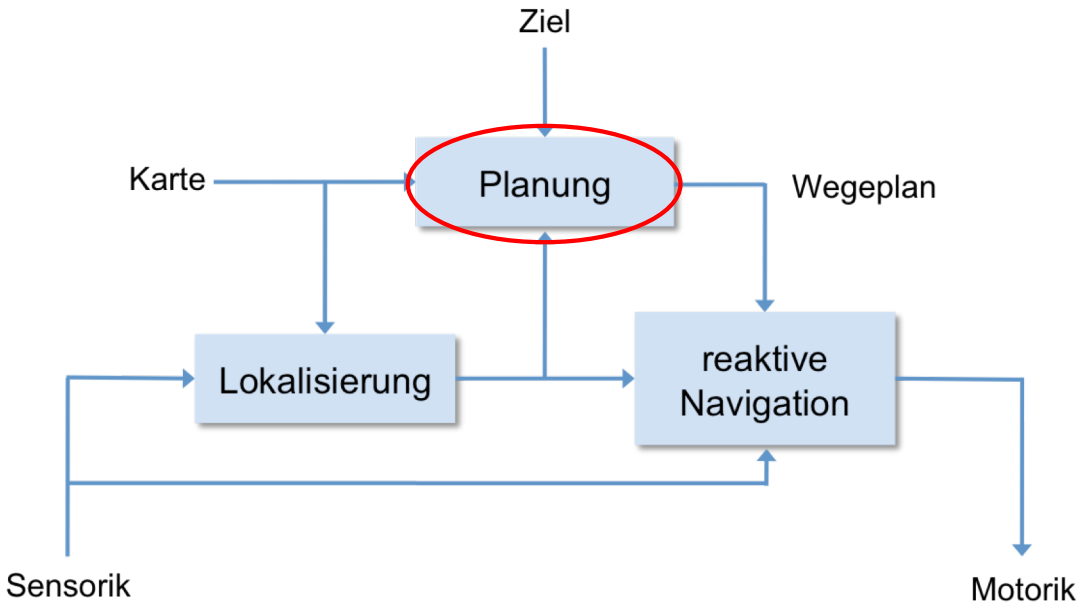
- Zweistufige Architektur:
  - Kartenbasierte Planung eines kollisionsfreien Wegs
  - Ausführen des geplanten Wegs durch reaktive Navigation

Reaktive Navigation



- Führt direkte Steuerung des Roboters durch:
  z.B. durch Setzen der Translations- und Rotationsgeschwindigkeit.
- Sensordaten können gefiltert sein und/oder über ein zeitlich begrenztes Intervall integriert werden: lokale Umgebungskarte.
- Aktuelle Positionsschätzung kann mit einbezogen werden.
- Üblicherweise wird aus einem Wegeplan
  die nächste Zielkoordinate vorgegeben.
- 10 – 100 Hz

Planung

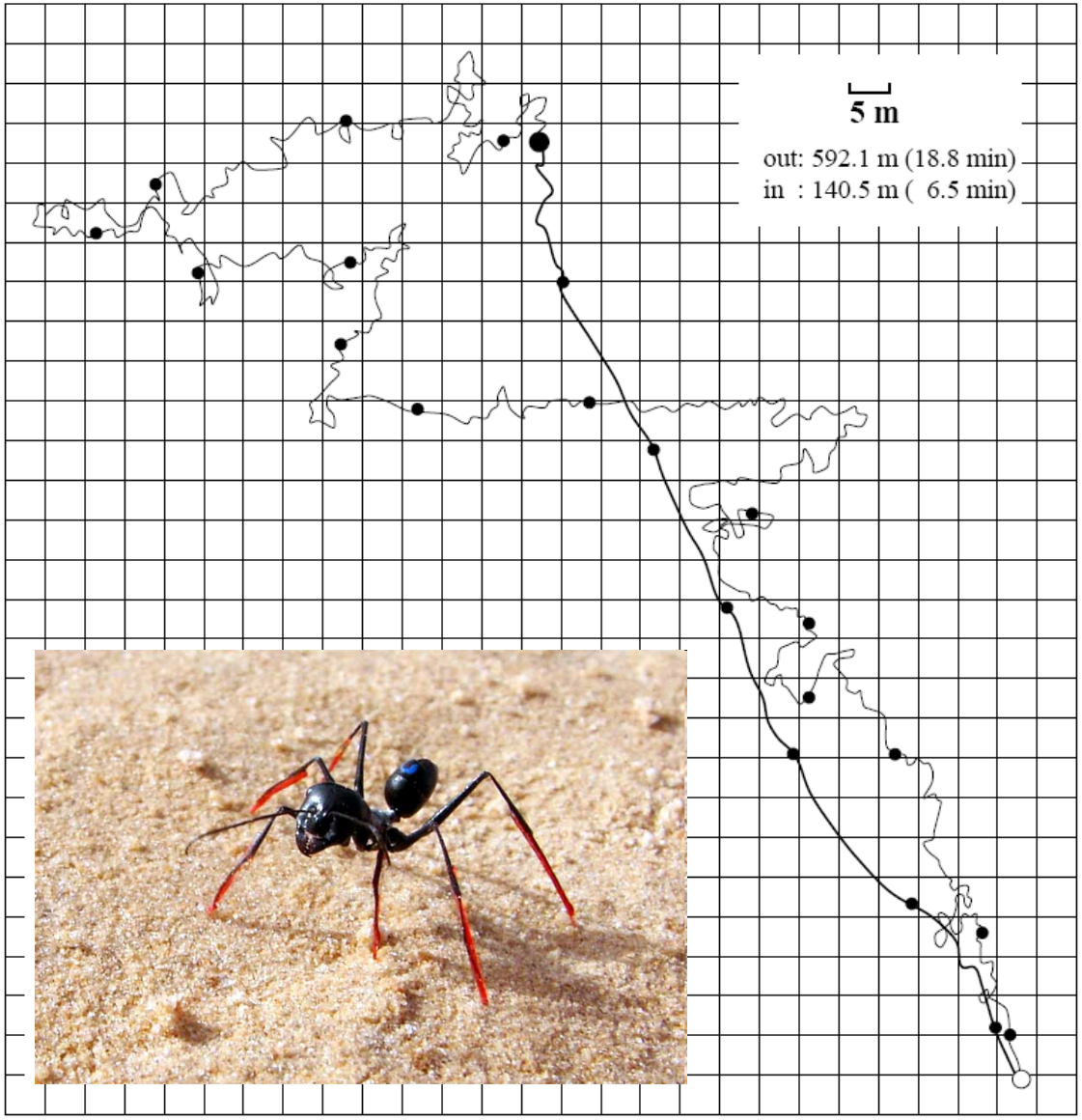


- Erstellung eines kollisionsfreien Wegeplans von der aktuellen Position zur Zielposition.
- Üblicherweise wird ein Plan mit Hilfe einer Karte erstellt.
- Dabei werden nur statische Hindernisse berücksichtigt.
- Meist wird Roboter als punktförmig angenommen
  (Hindernisse lassen sich um Roboterradius aufblasen)
- Meist keine Berücksichtigung der dynamischen Fähigkeiten des Roboters.
- Evtl. Unterstützung einer Planänderung (Sackgassenproblematik)
- ≤ 1 Hz

Navigation

- Überblick
  - Ziel, Architektur, Komponenten
- Reaktive Navigation
  - Insekten als Vorbild
  - Braitenberg-Vehikel
  - Zusammenbau elementarer Navigationsbausteine
  - Bug-Algorithmen
  - Histogramm-Verfahren
  - Dynamik-basiertes Verfahren
- Kartenbasierte Navigation
  - Kürzeste Wege in Graphen:
    - A*, Dijkstra-Verfahren und Breitensuche
  - Verfahren mit Belegtheitsgitter
  - Wegekartenverfahren:
    - Voronoi-Diagramme und probabilistische Wegekarten

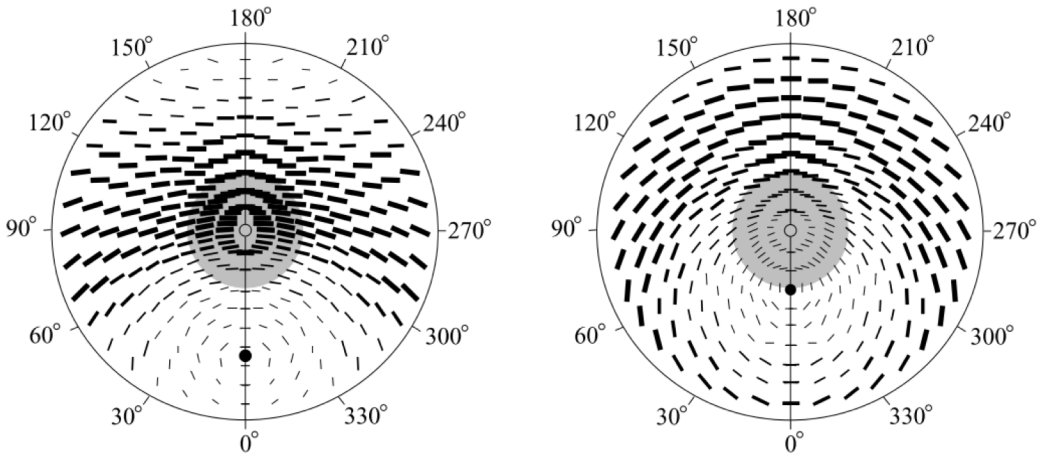
Insekten als Vorbild: Cataglyphis (1)



Wüstenameise (*Cataglyphis*)

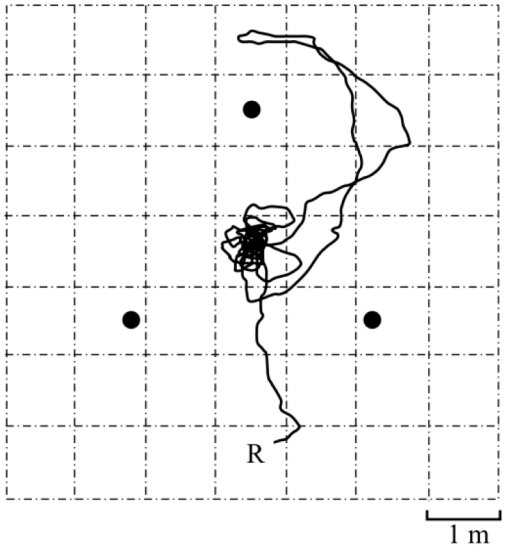
- geht auf Futtersuche nach dem Zufallsprinzip
- findet danach zum bis zu etwa 200 m entfernten Nest zurück.
- keine Pheromonspuren (Wüste!)

Insekten als Vorbild: Cataglyphis (2)



Polarisationsmuster für verschiedene Sonnenstände (schwarzer Punkt)

- Koppelnavigation durch Schrittzähler und Polarisationskompass

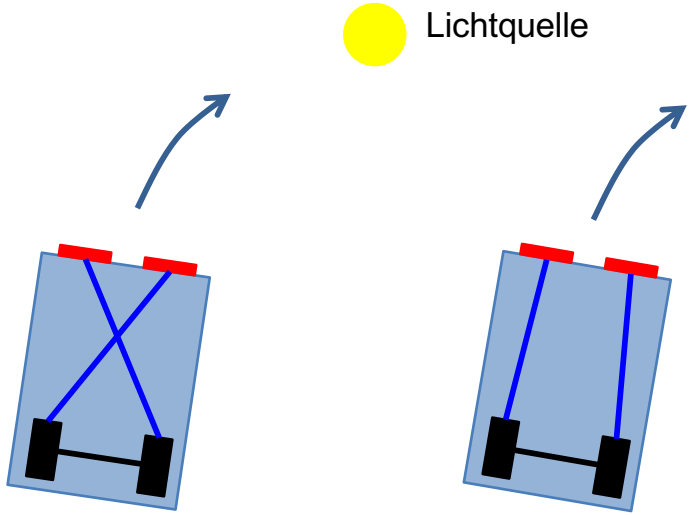


- Visuelle (landmarkenbasierte), lokale Suche im Nestbereich

Bilder aus *Lambrinos et. al. , A mobile robot employing insect strategies for navigation, 1999*.

Braitenberg Vehikel (1)

- Valentino Braitenberg: *Vehicles - Experiments in Synthetic Psychology,* 1984.
- Einfache Roboter die komplexes Verhalten realisieren sollen.
- Sensoren sind direkt mit Motoren gekoppelt
  (ganz allgemein gewichtete Summe)
- Einfache Sensorik:
  Sensoren liefern umgebungsabhängig einen skalaren Wert;
  z.B. Helligkeitssensor, Entfernungssensor, etc.



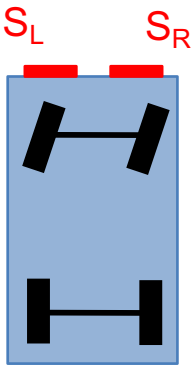
Aggressives Verhalten:
Roboter fährt auf
Lichtquelle zu

Ängstliches Verhalten:
Roboter fährt von
Lichtquelle weg

Simulation eines aggressiven Roboters (1)

- Roboter mit Ackermannsteuerung (Automobil);
  Steuerung der Geschwindigkeit v und des Lenkwinkels γ
- Helligkeitssensor misst Helligkeitswert, der umgekehrt proportional zum Abstand zur Lichtquelle (x_Q,y_Q) angenommen wird.
  Wert wird auf [0,1] normiert.

Lichtquelle Q

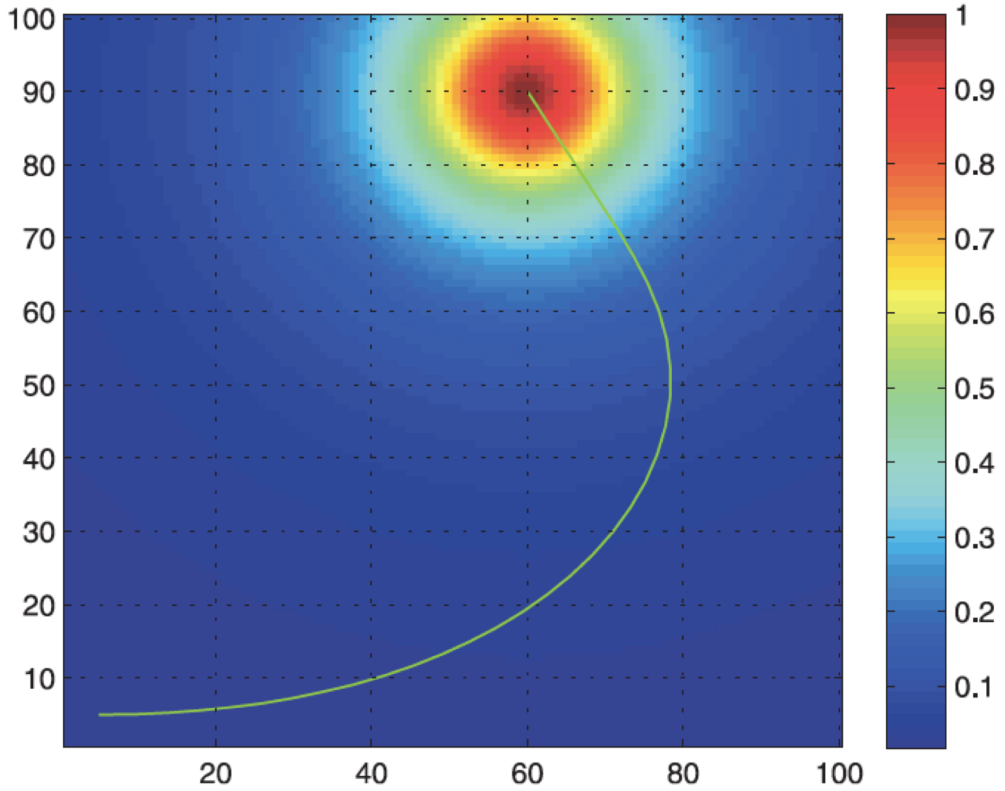


$$s = \frac{k_1}{(x - x_Q)^2 + (y - y_Q)^2 + k_1}$$

$$v = 2 - s_R - s_L$$

$$\gamma = k_2(s_R - s_L)$$

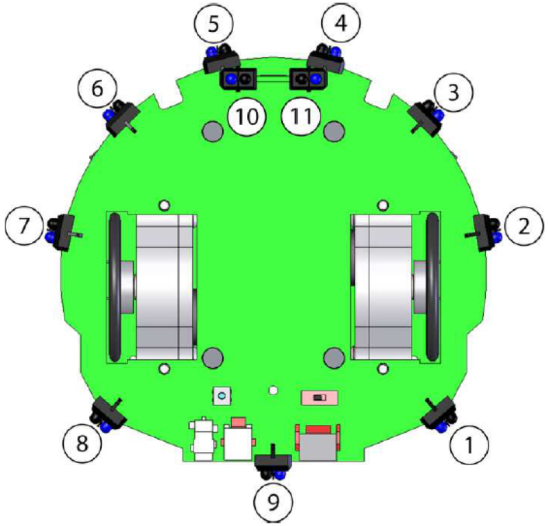
Simulation eines aggressiven Roboters (2)



- Lichtquelle bei Q = (60,90)
- Farbe stellt Entfernung zur Lichtquelle dar.

- Pfad des aggressiven Braitenberg-Vehikels, der zum Helligkeitsmaximum (= Position der Lichtquelle) führt.
- aus [Corke 2011]

Beispiel: Hindernisvermeidung mit Khepera III



IR-Sensoren s_1 bis s_9.

Linker Motor: $LM = \sum_i w_{l,i}\, s_i$

Rechter Motor: $RM = \sum_i w_{r,i}\, s_i$

IR-Sensor	1	2	3	4	5	6	7	8	9
Gewichte $w_{l,i}$ für LM	2	-2	-4	-12	5	2	2	2	4
Gewichte $w_{r,i}$ für RM	2	2	2	5	-12	-4	-2	2	4

rechts vorne

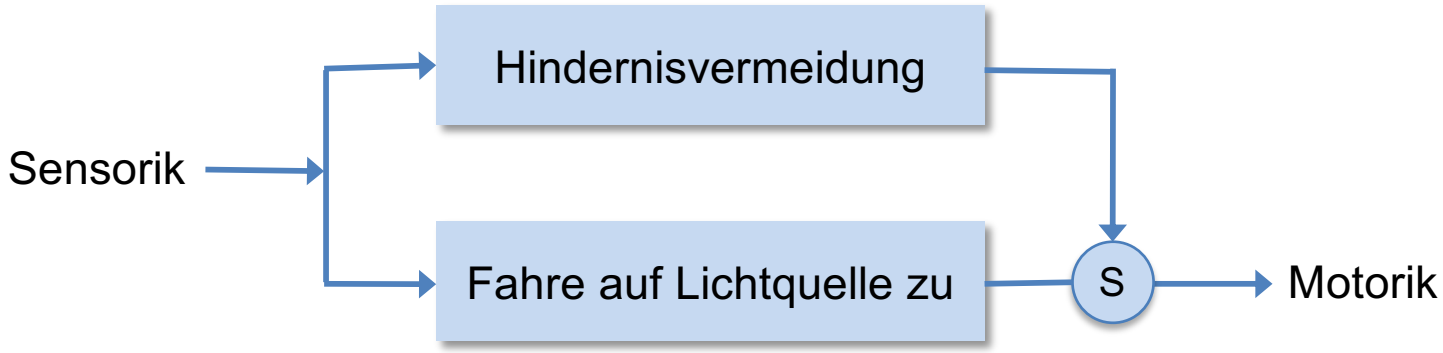
links vorne

Navigation

- Überblick
  - Ziel, Architektur, Komponenten
- Reaktive Navigation
  - Insekten als Vorbild
  - Braitenberg-Vehikel
  - Zusammenbau elementarer Navigationsbausteine
  - Bug-Algorithmen
  - Histogramm-Verfahren
  - Dynamik-basiertes Verfahren
- Kartenbasierte Navigation
  - Kürzeste Wege in Graphen:
    - A*, Dijkstra-Verfahren und Breitensuche
  - Verfahren mit Belegtheitsgitter
  - Wegekartenverfahren:
    - Voronoi-Diagramme und probabilistische Wegekarten

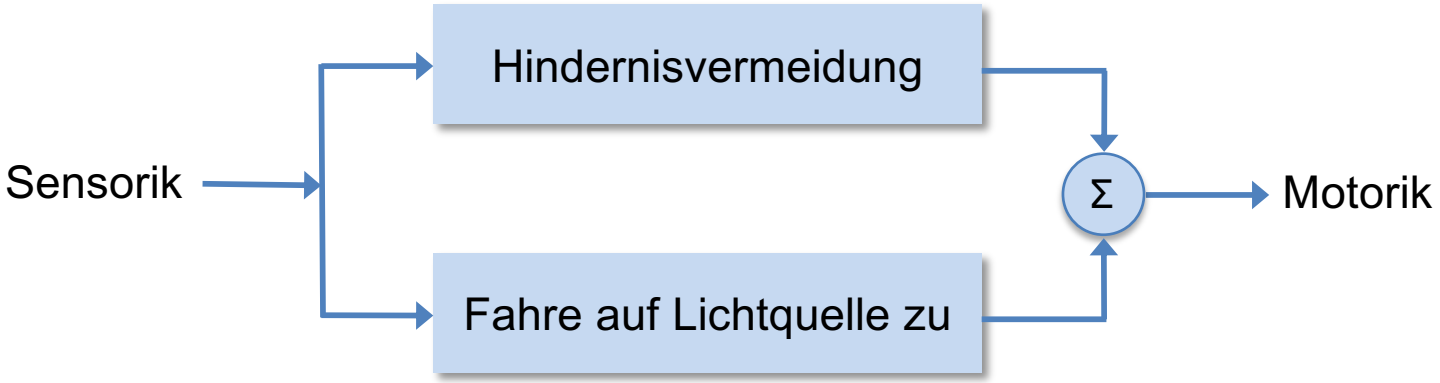
Subsumptionsarchitektur

- Mit einer Subsumptionsarchitektur lassen sich aus mehreren einfachen Verhalten (behaviour) ein komplexeres Verhalten erzielen.
- Dazu werden die Verhalten verschiedenen Ebenen zugeordnet. Die Verhalten der höheren Schichten können die Verhalten der unteren Schichten blockieren (suppress).
- Beispiel für zweischichtiges Verhalten:
  Der Roboter fährt auf die Lichtquelle zu, führt aber eine Hindernisvermeidung durch, sobald ein Hindernis auftaucht.



Fusionierung

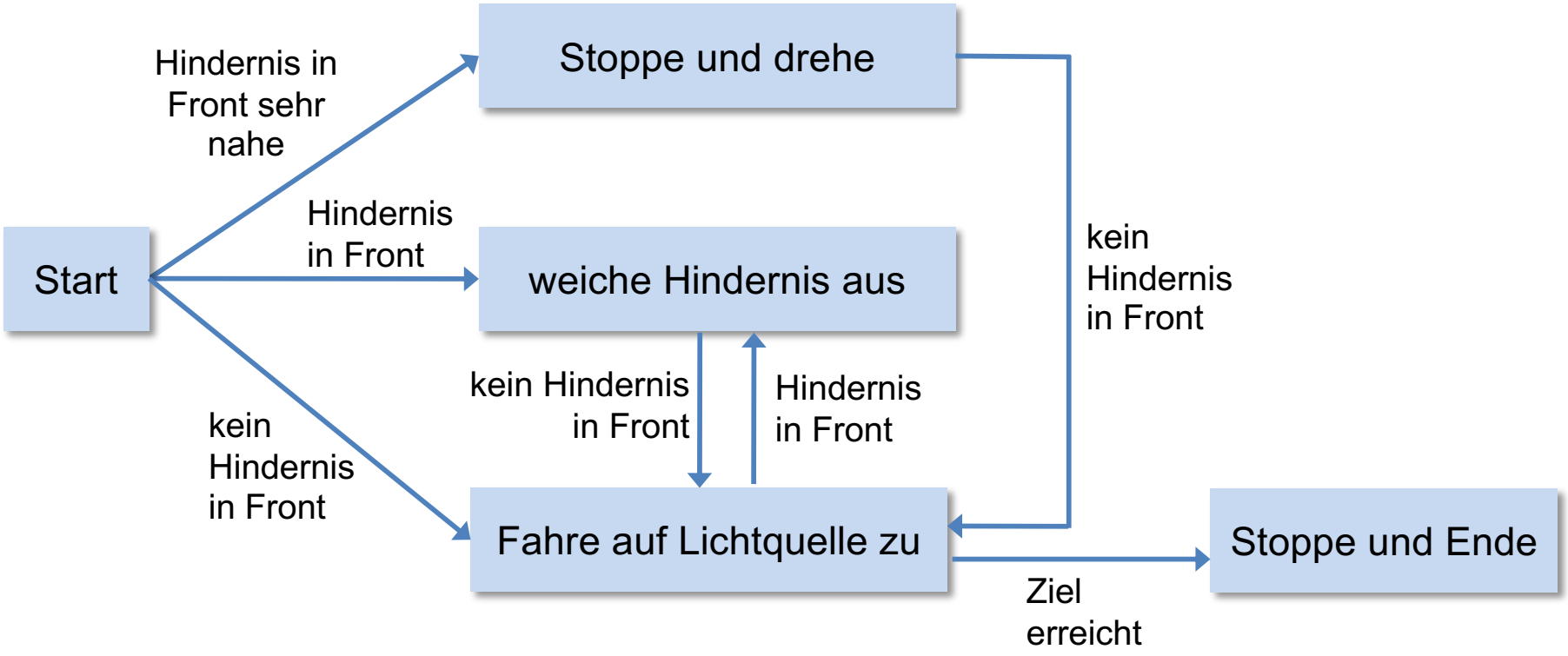
- Die Ausgaben mehrerer parallel laufender Verhalten werden durch eine gewichtete Summe zu einer Ausgabe fusioniert.



- Die Gewichte der Fusionierung kann von den Sensorwerten abhängen.

Endlicher Zustandsautomat

- Jedem Zustand wird ein Verhalten zugeordnet.
- Zustandswechsel wird über Sensorwerte gesteuert.



Navigation

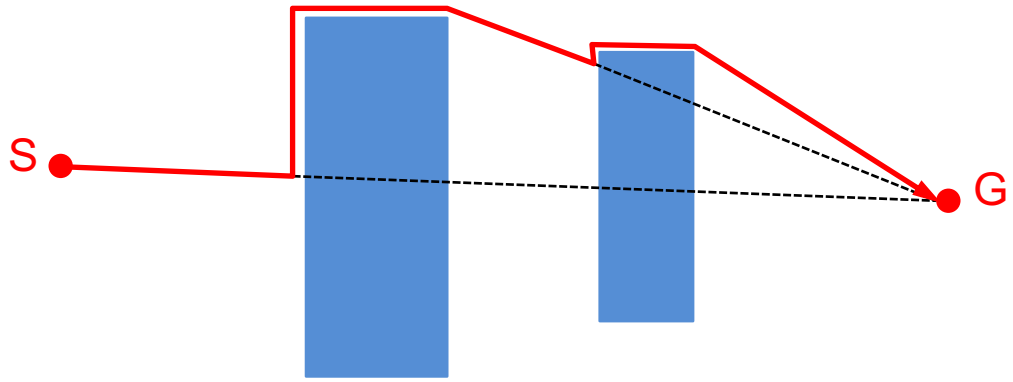
- Überblick
  - Ziel, Architektur, Komponenten
- Reaktive Navigation
  - Insekten als Vorbild
  - Braitenberg-Vehikel
  - Zusammenbau elementarer Navigationsbausteine
  - Bug-Algorithmen
  - Histogramm-Verfahren
  - Dynamik-basiertes Verfahren
- Kartenbasierte Navigation
  - Kürzeste Wege in Graphen:
    - A*, Dijkstra-Verfahren und Breitensuche
  - Verfahren mit Belegtheitsgitter
  - Wegekartenverfahren:
    - Voronoi-Diagramme und probabilistische Wegekarten

Bug Algorithmen

- Durch Insekten inspiriert. Daher der Name.
- Einfache Verhalten:
  - verfolge Wand (links oder rechts herum)
  - gehe auf einer geraden Linie auf Zielpunkt zu
- Einfache Abstandssensorik (oder auch Tastsensor), um Hindernis zu erkennen und Wand zu verfolgen.
- Zielpunkt bekannt.
- Aktuelle Position bekannt.
  Roboter kann damit Abstand und Richtung zum Zielpunkt bestimmen.
  Forderung lässt sich abschwächen.
- Einfache Umgebung: Menge von Polyeder

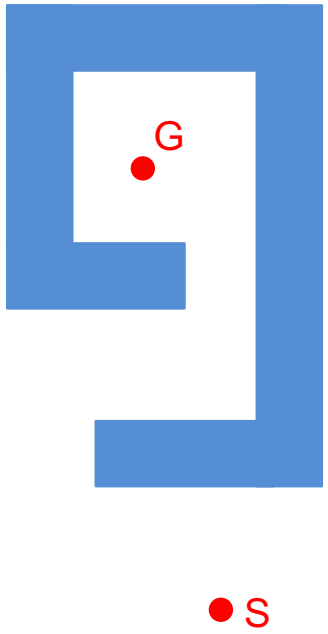
Bug 0 (Naiver Ansatz)

1. Gehe solange auf Ziel zu, bis Ziel erreicht
   oder Berührung mit Hindernis.
   Falls Ziel erreicht, breche mit Erfolg ab.
2. Drehe bei Hindernisberührung nach links (oder rechts)
   und verfolge Wand solange,
   bis Bewegung Richtung Ziel möglich.
   Gehe zu Schritt 1.



Vollständigkeit eines Navigationsverfahren

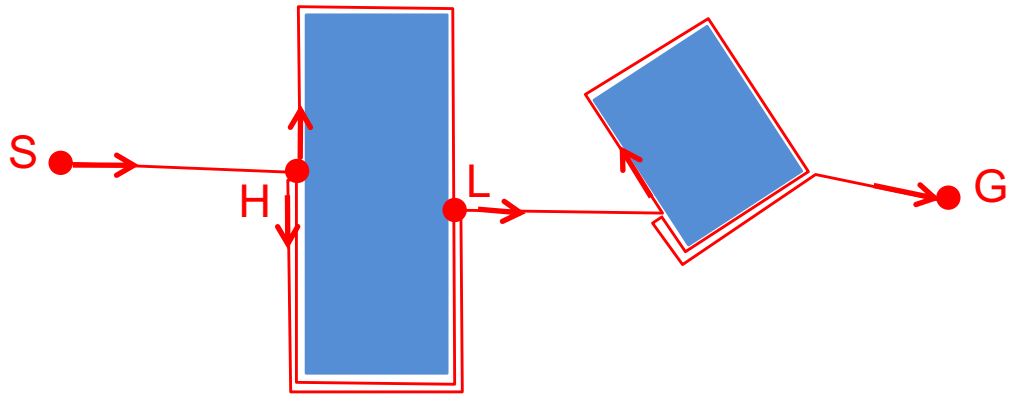
- Ein vollständiges Navigationsverfahren liefert einen Weg von A nach B, falls es einen Weg von A nach B gibt.
- Bug 0 ist nicht vollständig:



- Es gibt offensichtlich einen Weg von S nach G.
- Bug 0 findet aber keinen Weg von S nach G.
- Zeichnen Sie den von Bug 0 gewählten Weg ein.

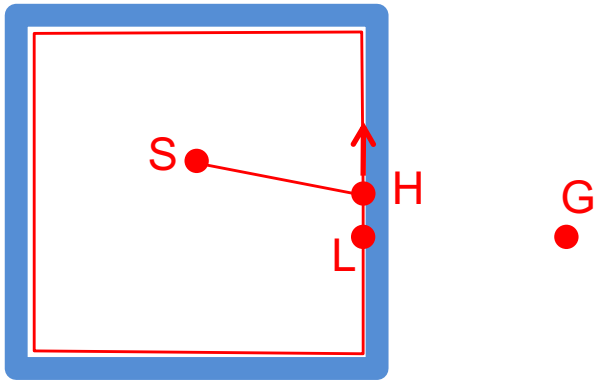
Bug 1

1. Gehe solange auf Ziel zu, bis Ziel erreicht
   oder Berührung mit Hindernis am Punkt H.
   Falls Ziel erreicht, breche mit Erfolg ab.
2. Drehe bei Punkt H (hit point) nach links (oder rechts)
   und verfolge Wand einen kompletten Umfang lang
   (d.h. bis H wieder erreicht wird).
3. Gehe zum Punkt L (leave point) mit geringstem Abstand zum Ziel zurück.
   Falls bei L Richtung zum Ziel blockiert ist, breche mit Misserfolg ab.
   Gehe zu Schritt 1.



Bug 1 bei nicht existierendem Weg

1. Gehe solange auf Ziel zu, bis Ziel erreicht
   oder Berührung mit Hindernis am Punkt H.
   Falls Ziel erreicht, breche mit Erfolg ab.
2. Drehe bei Punkt H nach links (oder rechts)
   und verfolge Wand einen kompletten Umfang lang
   (d.h. bis H wieder erreicht wird).
3. Gehe zum Punkt L mit geringstem Abstand zum Ziel zurück.
   Falls bei L Richtung zum Ziel blockiert ist, breche mit Misserfolg ab.
   Gehe zu Schritt 1.



Analyse von Bug 1

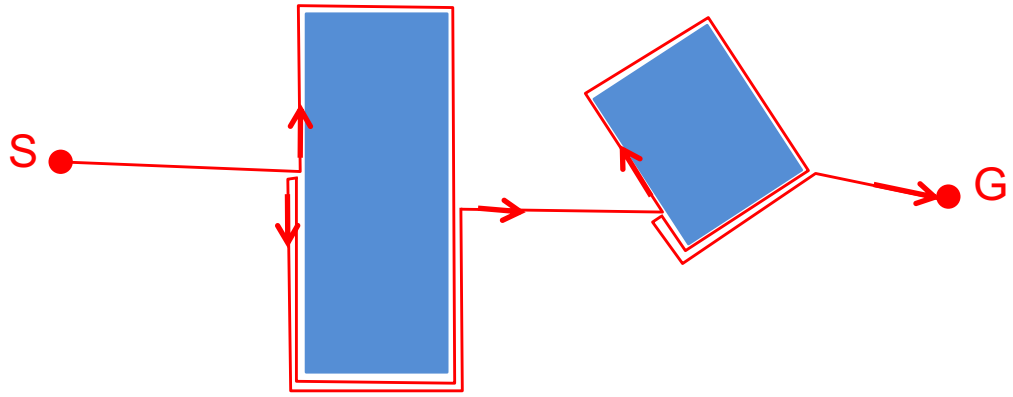
- Bug 1 ist vollständig
- Länge l des gefahrenen Wegs:

$$l \leq d + \frac{3}{2}\sum_{i=1}^{M} p_i$$

d ist der Euklidische Abstand zwischen S und G

p_i ist der Umfang des i-ten Hindernisses

M ist die Anzahl der Hindernisse.



Ein besserer Bug-Algorithmus?

- Bug 0 ist ein gieriger (greedy) Algorithmus:
  gehe direkt auf Ziel zu, sobald möglich.

  Problem: Endlosschleifen in manchen Situationen

- Bug 1 zeichnet sich durch eine
  erschöpfende Suche (exhaustive search) aus:
  schaue Hindernis zuerst komplett an und
  entscheide dann über weiteren Weg.

  Problem: Wege sind in vielen Fällen unnötig lang.

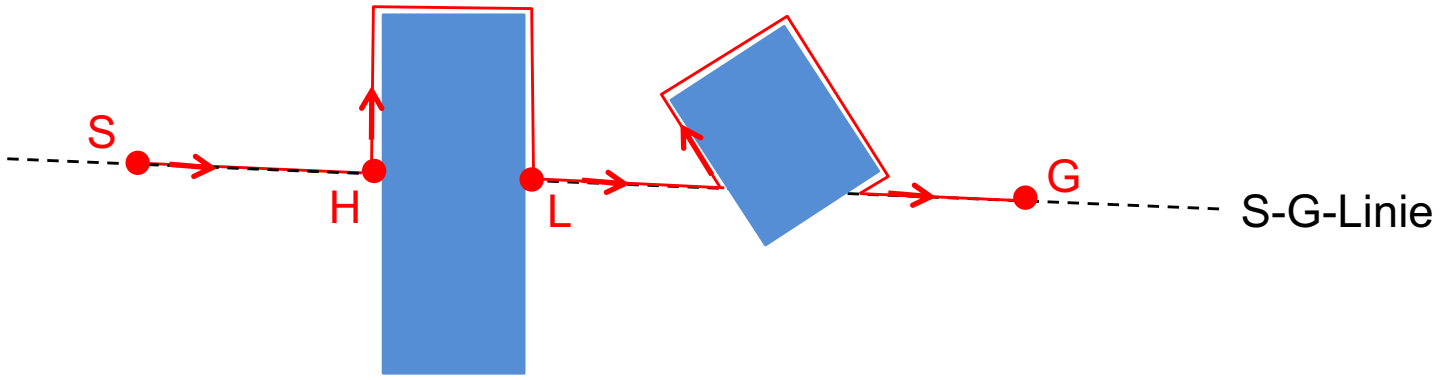
- Bug 2 ist wie Bug 0 ein gieriges Vefahren,
  jedoch werden Endlos-Schleifen vermieden.

Bug 2

1. Gehe auf S-G-Linie solange auf Ziel zu, bis Ziel erreicht oder Berührung mit Hindernis am Punkt H.
   Falls Ziel erreicht, breche mit Erfolg ab.
2. Drehe bei Punkt H nach links (oder rechts) und verfolge Wand bis
   a) Punkt H wieder erreicht wurde oder
   b) ein Punkt L erreicht wurde, der auf der S-G-Linie liegt und näher zum Ziel als der Punkt H liegt und nicht blockiert ist.

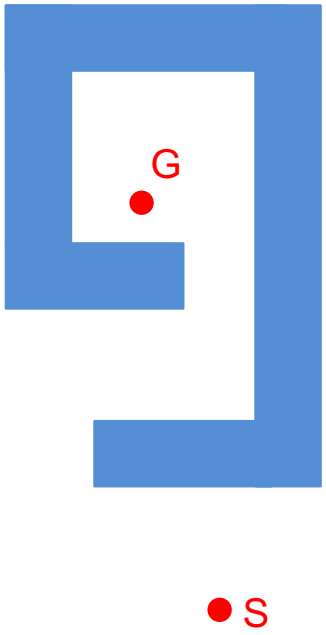
   Im Fall a) breche mit Misserfolg ab.

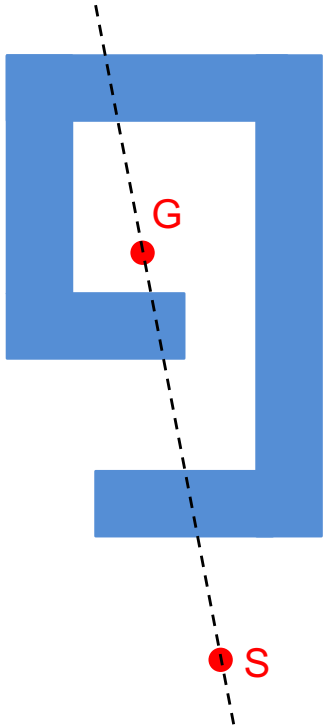
   Im Fall b) mache mit Schritt 1 weiter.



Aufgabe

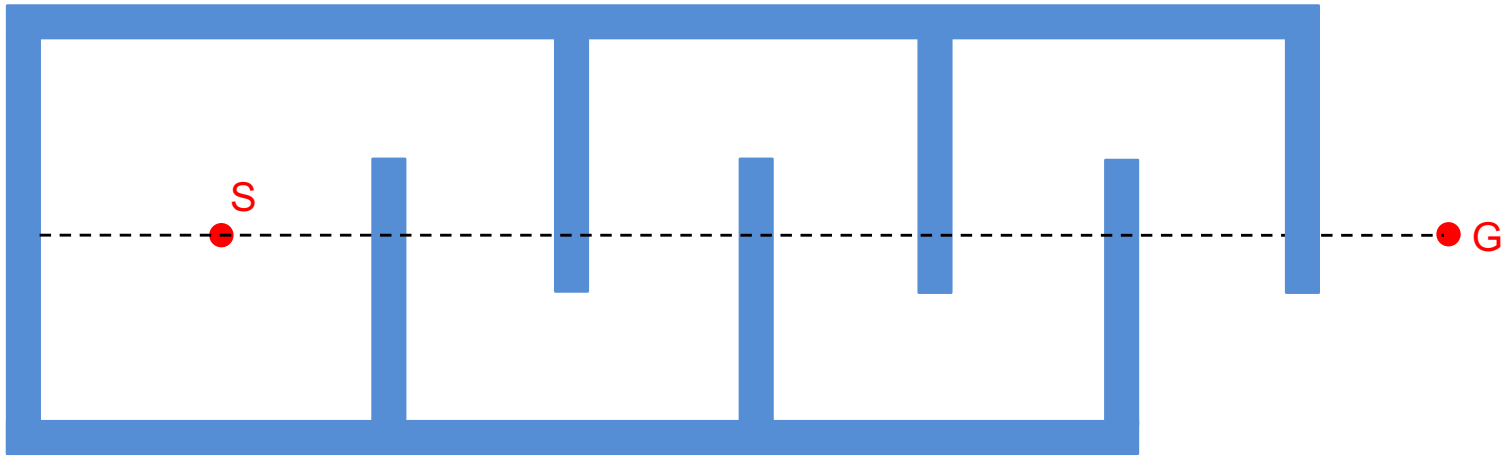
- Führen Sie Bug 1 und Bug 2 in folgendem Szenario aus:



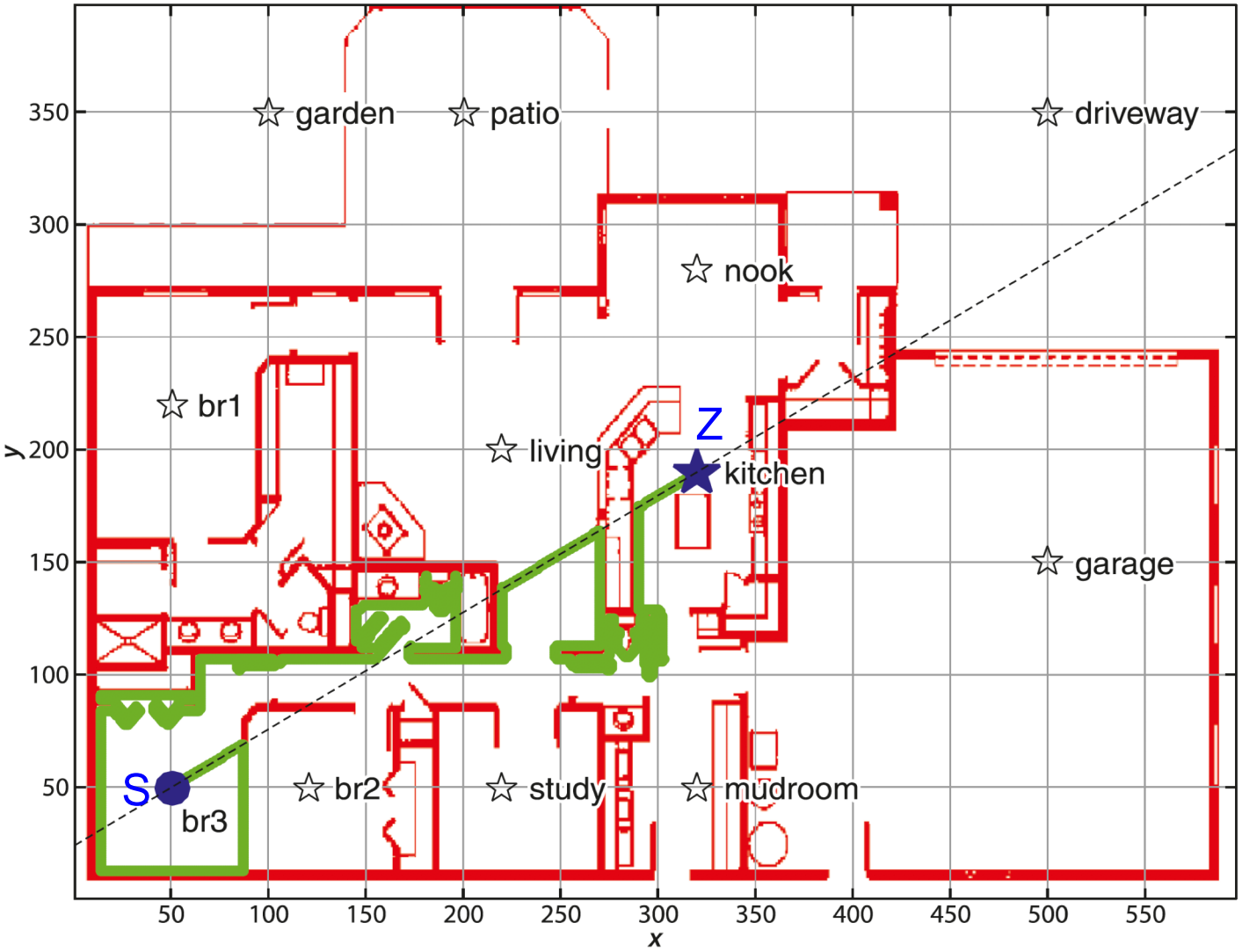


Aufgabe

- Wie verhält sich Bug 2 in folgendem Szenario?



Beispielszenario für Bug 2



Von Bug 2 gewählter Weg in grün. S = Start und Z = Ziel [Corke, 2017]

Analyse von Bug 2

- Bug 2 ist vollständig
- Länge l des gefahrenen Wegs:

$$l \leq d + \frac{1}{2}\sum_{i=1}^{M} n_i p_i$$

d ist der Euklidische Abstand zwischen S und G

p_i ist der Umfang des i-ten Hindernisses

n_i ist die Anzahl der Schittpunkte der S-G-Linie mit dem i-ten Hindernis.

M ist die Anzahl der Hindernisse.

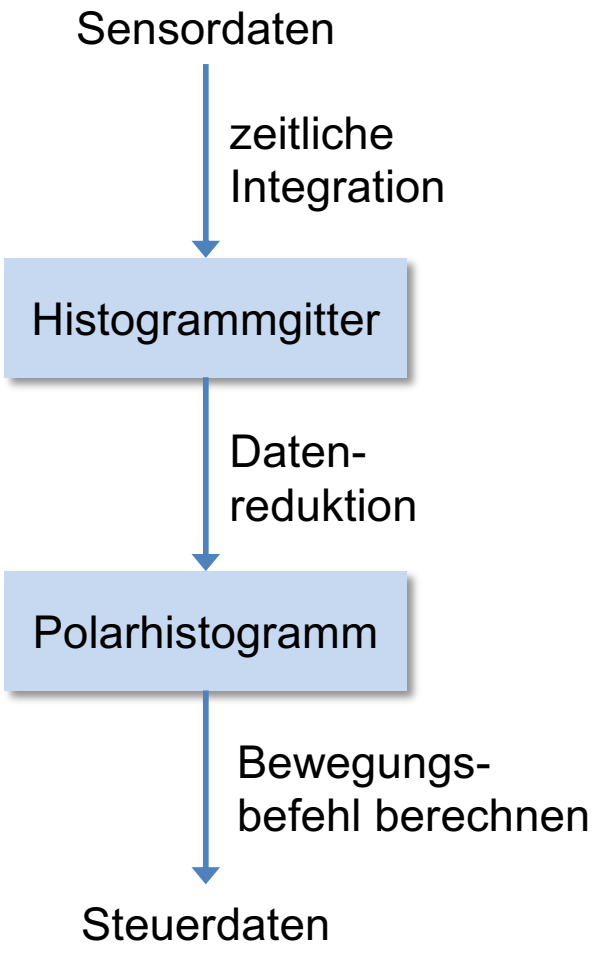
- Begründung für die Obergrenze der Wegelänge:
  die Hälfte der Schnittpunkte der S-G-Linie mit den Hindernissen sind H-Punkte und die andere Hälfte sind L-Punkte.
  Nur für die H-Punkte wird eine Wandverfolgung begonnen, die maximal die Länge des Hindernisumfangs hat.

Navigation

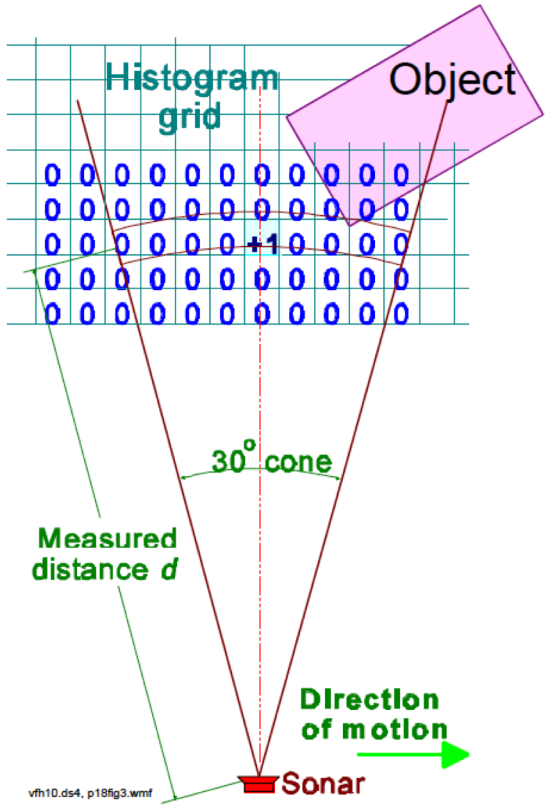
- Überblick
  - Ziel, Architektur, Komponenten
- Reaktive Navigation
  - Insekten als Vorbild
  - Braitenberg-Vehikel
  - Zusammenbau elementarer Navigationsbausteine
  - Bug-Algorithmen
  - Histogramm-Verfahren
  - Dynamik-basiertes Verfahren
- Kartenbasierte Navigation
  - Kürzeste Wege in Graphen:
    - A*, Dijkstra-Verfahren und Breitensuche
  - Verfahren mit Belegtheitsgitter
  - Wegekartenverfahren:
    - Voronoi-Diagramme und probabilistische Wegekarten

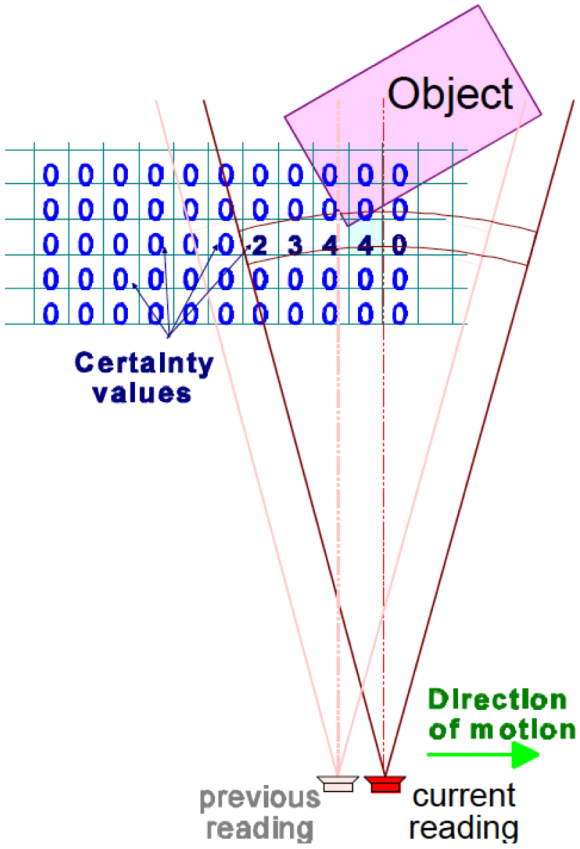
Vector Field Histogram (VFH)

- Borenstein und Koren: VFH 1991.
- Weitere Verbesserungen durch
  Ulrich und Borenstein:
  VFH+ 1998 (Dynamik u. Größe des Roboters)
  VFH* 2000 (= VHF + A*)
- Aufbau einer lokalen Umgebungskarte (Histogrammgitter, histogram grid) durch zeitliche Integration von Sensor-Abstandsinformationen
- Aus der Umgebungskarte wird für die einzelnen Richtungen
  (bzgl. Roboter-Kooordinatensystem) die Hindernisdichte berechnet: Polarhistogramm
- Auf Basis des Polarhistogramms wird Bewegungsbefehl (v, ω) berechnet



Histogrammgitter

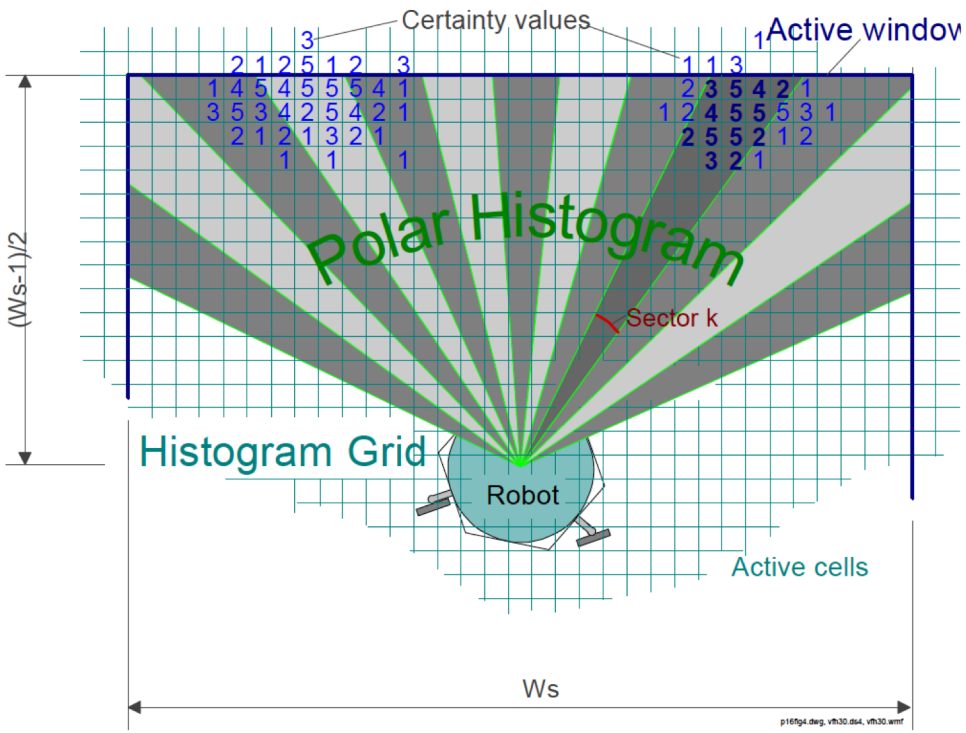




Abbildungen (auch auf den folgenden Seiten) aus [Borenstein und Koren, 1991].

- Für jeden Sensor-Abstandswert wird die entsprechende Zelle des zweidimensionalen Histogrammgitters inkrementiert.
- Sensorwerte werden über einen längeren Zeitraum gesammelt.
- Histogrammgitter wird auf einen lokalen Bereich um den Roboter (z.B. 5m*5m) begrenzt mit einer Zellgröße von 0.1m.

Polar Histogram



- Jede Zelle i des Histogrammgitters hat einen Belegheitswert c_i
- Für jede belegte Zelle i werden Polarkoordinaten d_i, α_i (bzgl. Roboter-KS) berechnet.
- Daraus wird Einflussfaktor m_i bestimmt (a,b Konstante):

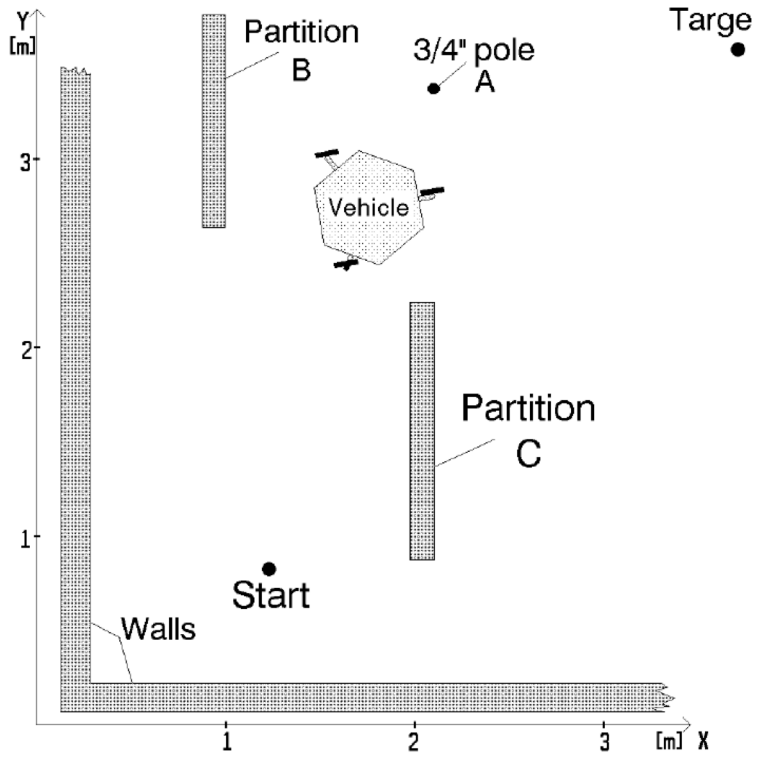
$$m_i = c_i^2(a - bd_i)$$

- Winkel werden in Sektoren s aufgeteilt (z.B. 72 Sektoren mit 5 Grad)
- Für jeden Sektor s wird Belegheitsdichte $h(s)$ ermittelt (Polar-Histogramm)

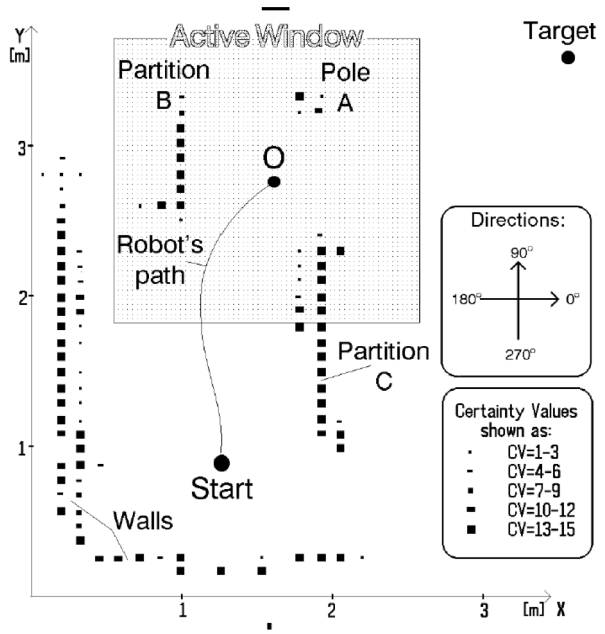
$$h(s) = \sum_{Zelle\ i\ in\ s} m_i$$

- Polarhistogramm wird noch geglättet

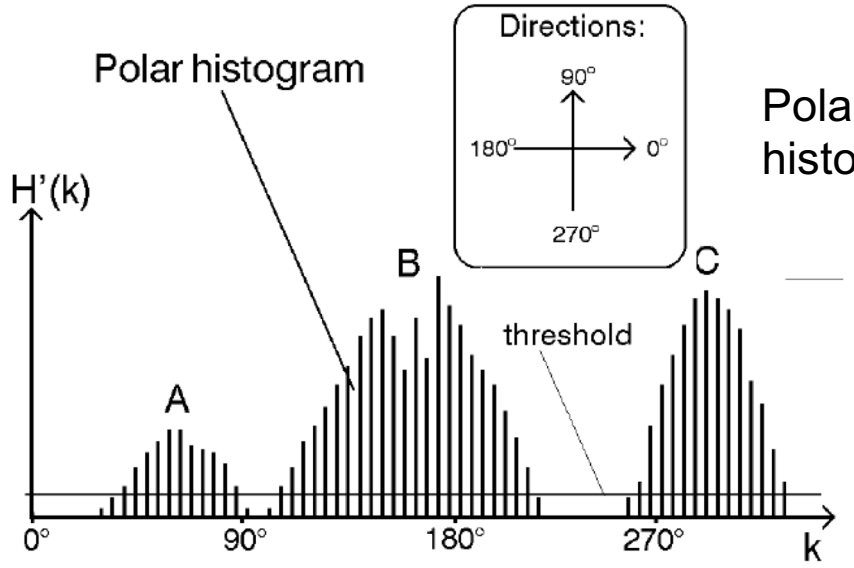
Typisches Szenario



Roboter und Umgebung



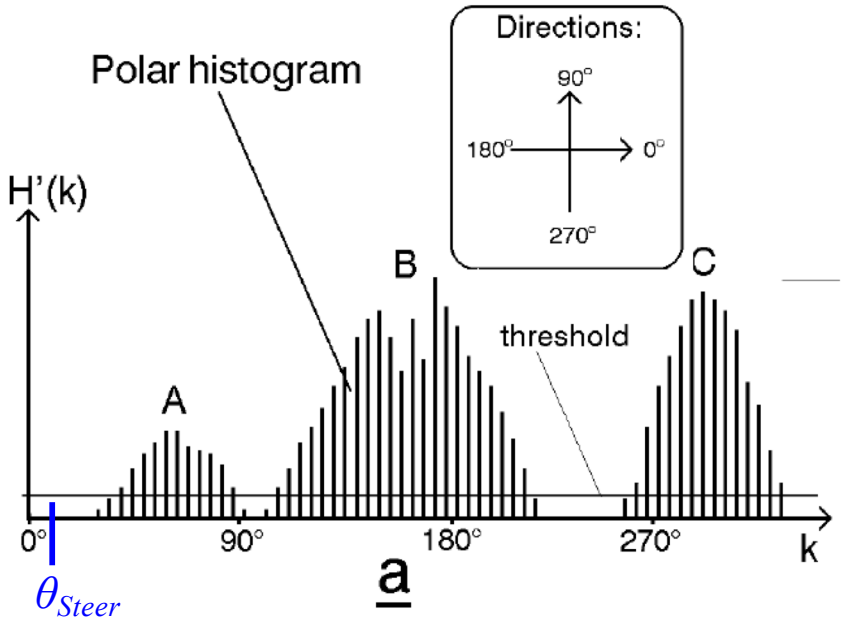
Histogramm-gitter

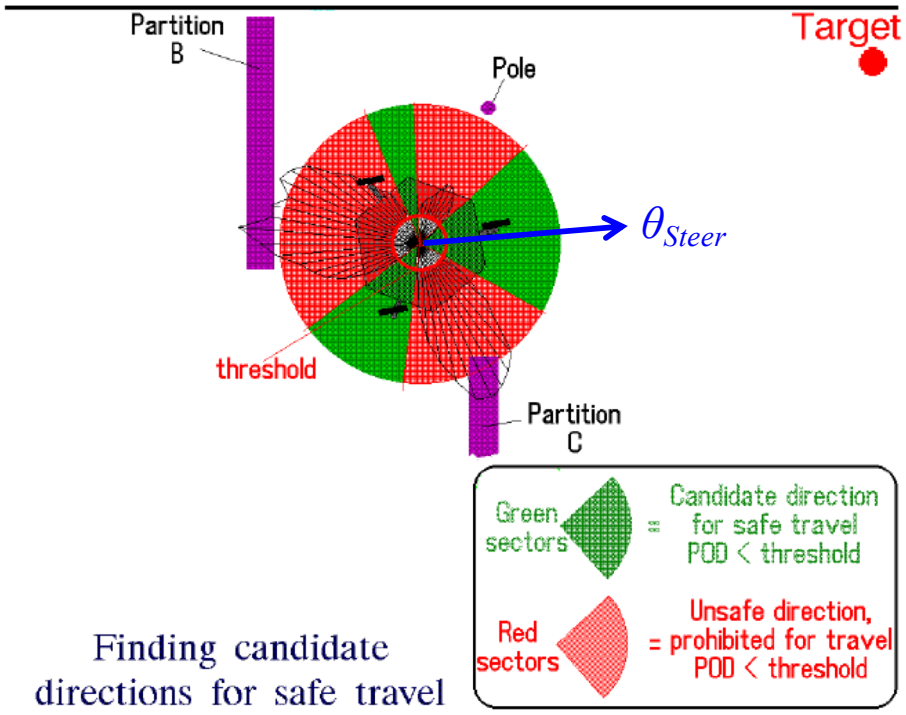


Polar-histogramm

Berechnung der Bewegungsrichtung

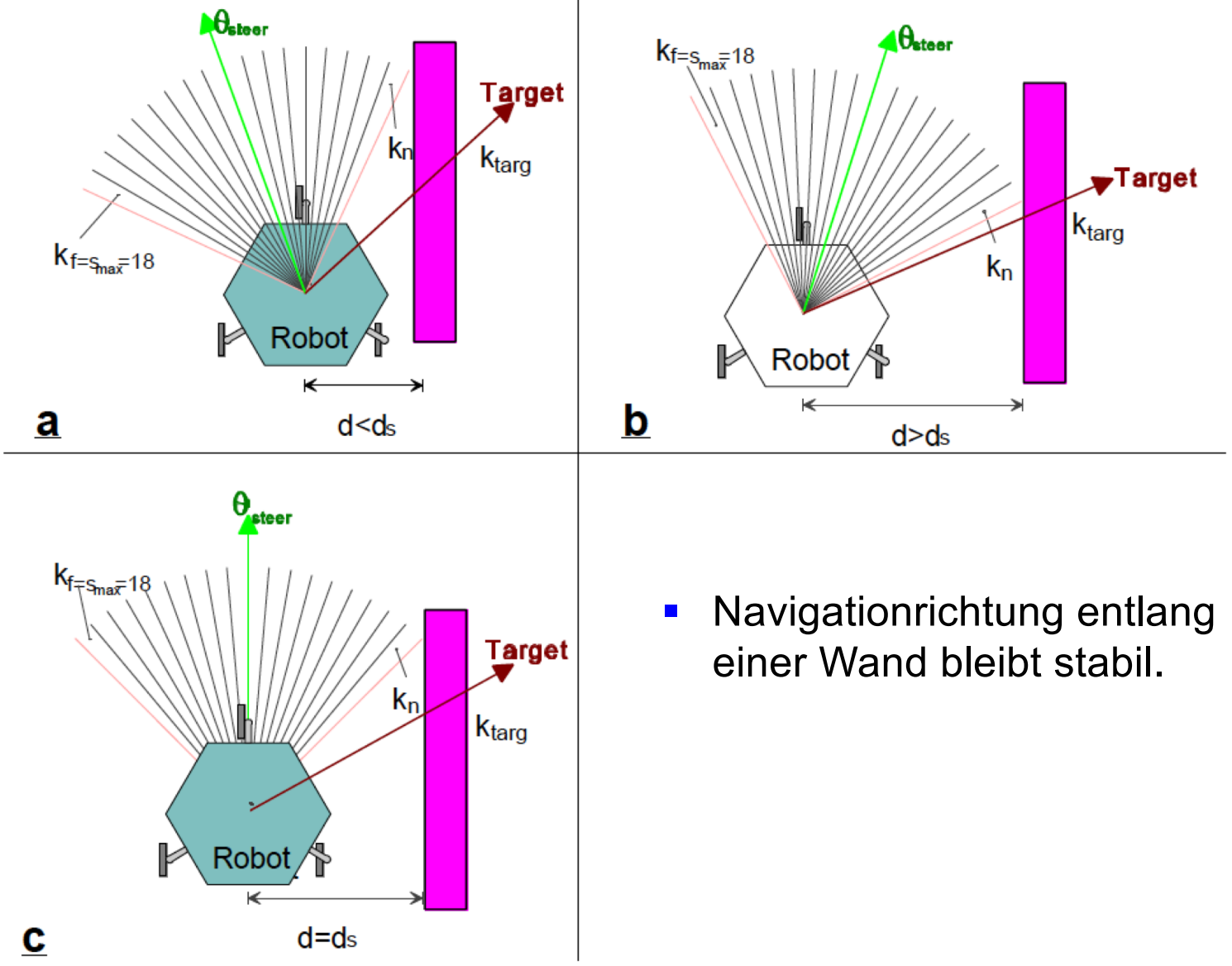
- Im Polarhistogramm werden alle Täler (lokale Minima) betrachtet, deren Histogrammwerte unterhalt einer bestimmten Schwelle (threshold) liegen.
- Es wird dajenige Tal gewählt, das am besten zur Ziel-Richtung passt.
- Als Bewegunsgrichtung θ_{Steer} wird die Talmitte gewählt.
  Jedoch Begrenzung bei breiten Tälern.





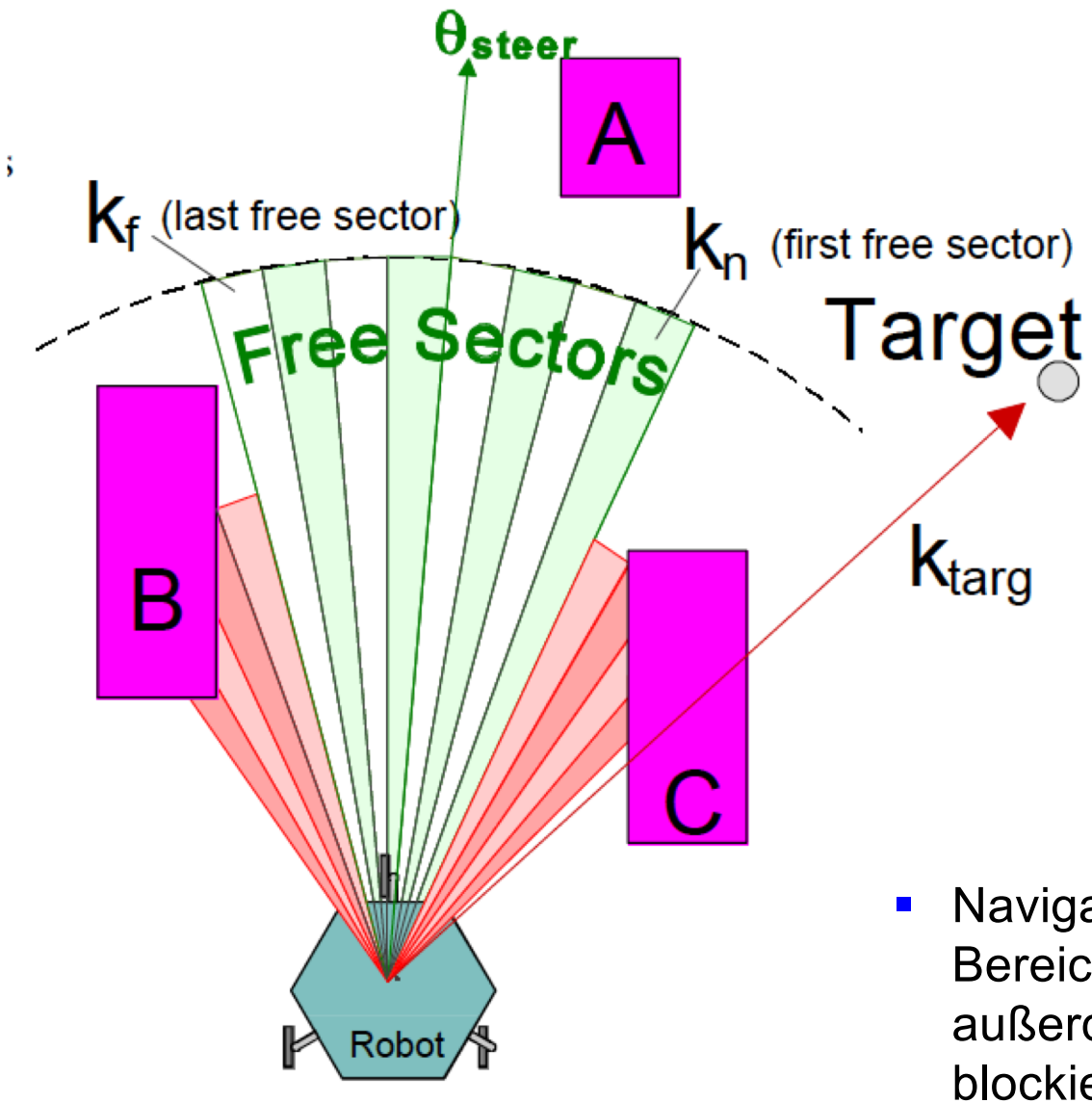
Finding candidate directions for safe travel

Berechnung der Bewegungsrichtung - Beispiel



- Navigationrichtung entlang einer Wand bleibt stabil.

Berechnung der Bewegungsrichtung - Beispiel



- Navigation in einem engen Bereich. Die Zielrichtung ist außerdem durch ein Hindernis blockiert.

Berechnung des Bewegungsbefehls

- Rotationsgeschwindigkeit ω wird proportional zur Abweichung zwischen tatsächlicher Ausrichtung θ und gewünschter Ausrichtung θ_{Steer} (begrenzt auf ω_{max}) gesetzt:

$$\omega = \min(\omega_{\max}, k * (\theta_{steer} - \theta))$$

- Translationsgeschwindigkeit v wird umgekehrt proportional zur Hindernisdichte $h(\theta_{Steer})$ in gewünschter Richtung und umgekehrt proportional zur Rotationsgeschwindigkeit ω gesetzt. Bei freier Geradeaus-Fahrt wird Maximalgeschwindigkeit gewählt.

$$v = v_{\max}\left(1 - \frac{h(\theta_{steer})}{h_0}\right)\left(1 - \frac{\omega}{\omega_{\max}}\right)$$

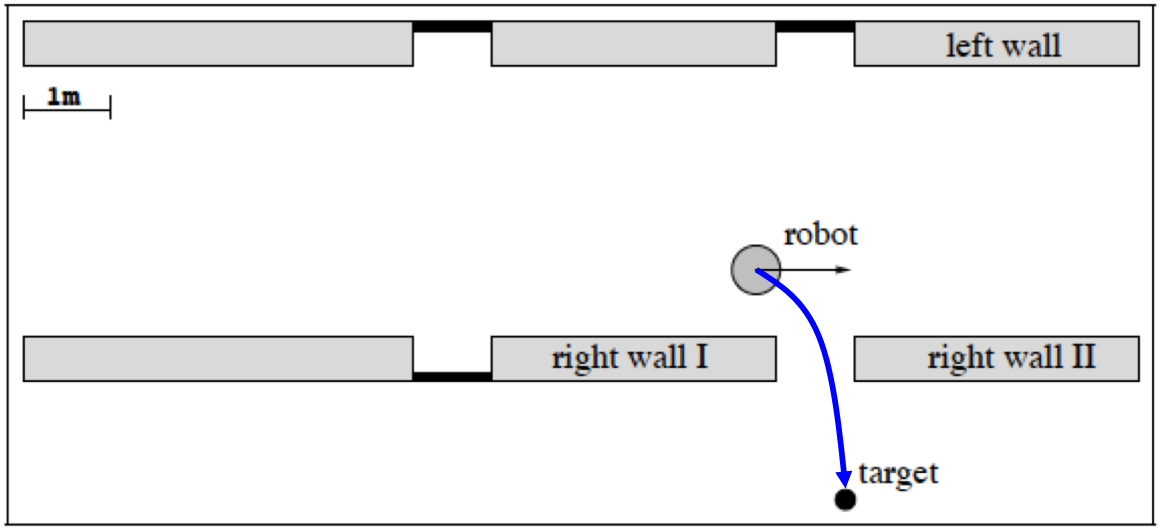
- Verfahren wurde im Praxiseinsatz mit ω_{max} = 120 Grad/sec und v_{max} = 0.8 m/sec erfolgreich getestet.

Navigation

- Überblick
  - Ziel, Architektur, Komponenten
- Reaktive Navigation
  - Insekten als Vorbild
  - Braitenberg-Vehikel
  - Zusammenbau elementarer Navigationsbausteine
  - Bug-Algorithmen
  - Histogramm-Verfahren
  - Dynamik-basiertes Verfahren
- Kartenbasierte Navigation
  - Kürzeste Wege in Graphen:
    - A*, Dijkstra-Verfahren und Breitensuche
  - Verfahren mit Belegtheitsgitter
  - Wegekartenverfahren:
    - Voronoi-Diagramme und probabilistische Wegekarten

Dynamic Window Approach (DWA)

- Fox, Burgard und Thrun,
  The Dynamic Window Approach to Collision Avoidance, 1997.
- Auswahl des Steuerbefehls (v, ω) unter Berücksichtigung der dynamischen Fähigkeiten des Roboters (Beschleunigung und Winkelbeschleunigung).



alle Abbildungen auch auf den folgenden Seiten aus
[Fox, Burgard u. Thrun, 1997]

Nur bei bestimmten Geschwindigkeiten und dynamischen Fähigkeiten ist es sinnvoll, den Roboter direkt zum Ziel zu steuern.

DWA-Verfahren

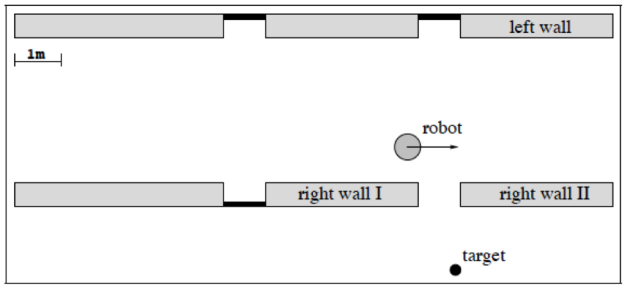
1. Festlegung des Suchraums für eine Zielfunktion:

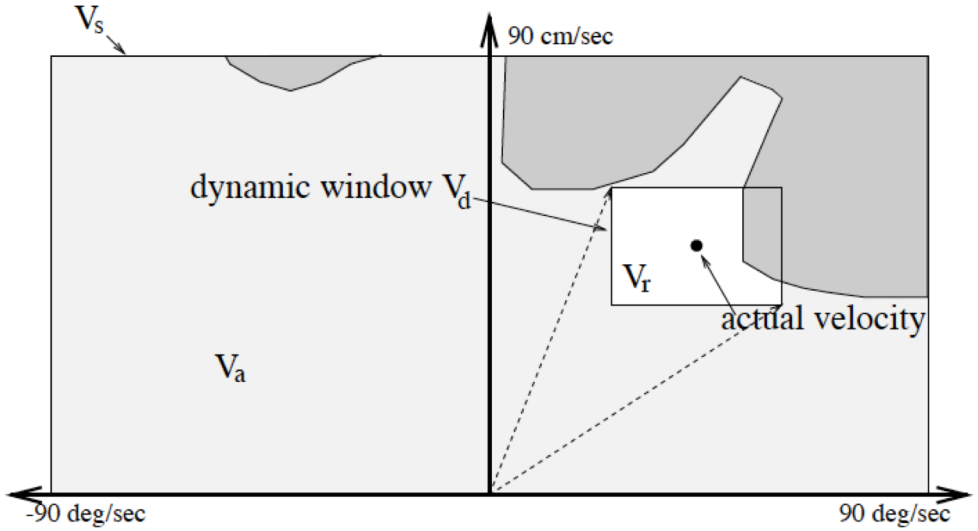
   Es werden alle Steuerbefehle (v, ω) betrachtet, die aufgrund der kinematischen Fähigkeiten des Roboters (Beschleunigung und Winkelbeschleunigung) möglich und aufgrund der Hindernisse erlaubt (kollisionsfrei) sind.

2. Auswahl des bestmöglichen Steuerbefehls (v, ω) mit einer dreiteiligen Zielfunktion:
   - gute Zielausrichtung
   - weite Entfernung von Hindernissen
   - hohe Geschwindigkeit

Suchraum

- Festlegung der kinematisch möglichen und der zulässigen (d.h. kollisionsfreien) Steuerbefehle (v, ω)





V_s = Bereich mit allen möglichen Steuerbefehlen.

V_d = kinematisch mögliche Steuerbefehle.

V_a = zulässige Steuerbefehle (d.h. kollisionsfrei).

- Suchraum für Zielfunktion:

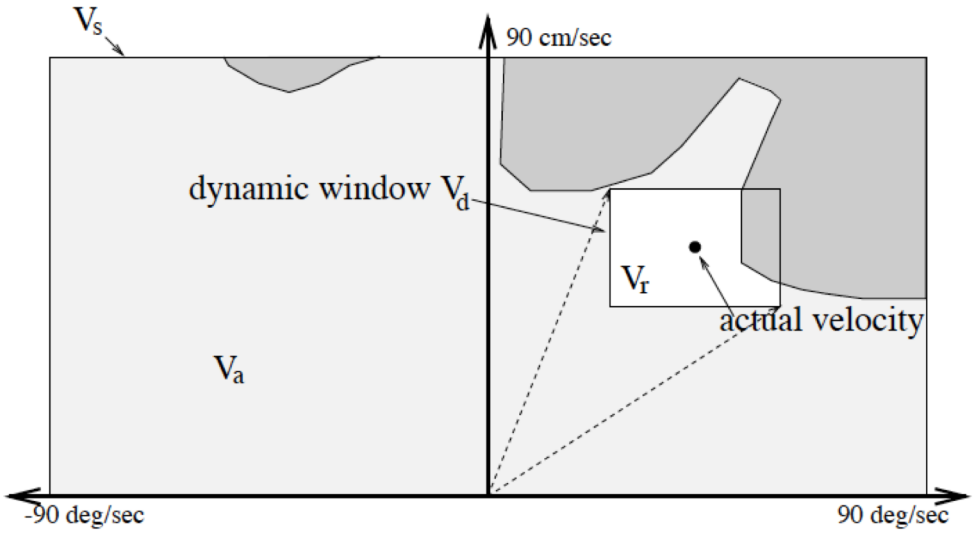
$$V_r = V_s \cap V_d \cap V_a$$

Kinematisch mögliche Steuerbefehle

- V_d umfasst alle Geschwindigkeiten, die im nächsten Zeitschritt T erreichbar sind.
- Dabei wird von einer maximal möglichen Beschleunigung a und Winkelbeschleunigung α des Roboters ausgegangen.

$$V_d = \{(v,\omega) / v_a - aT \le v \le v_a + aT \text{ und } \omega_a - \alpha T \le \omega \le \omega_a + \alpha T\}$$

- Dabei ist v_a , ω_a die aktuelle Geschwindigkeit bzw. Winkelgeschwindigkeit.

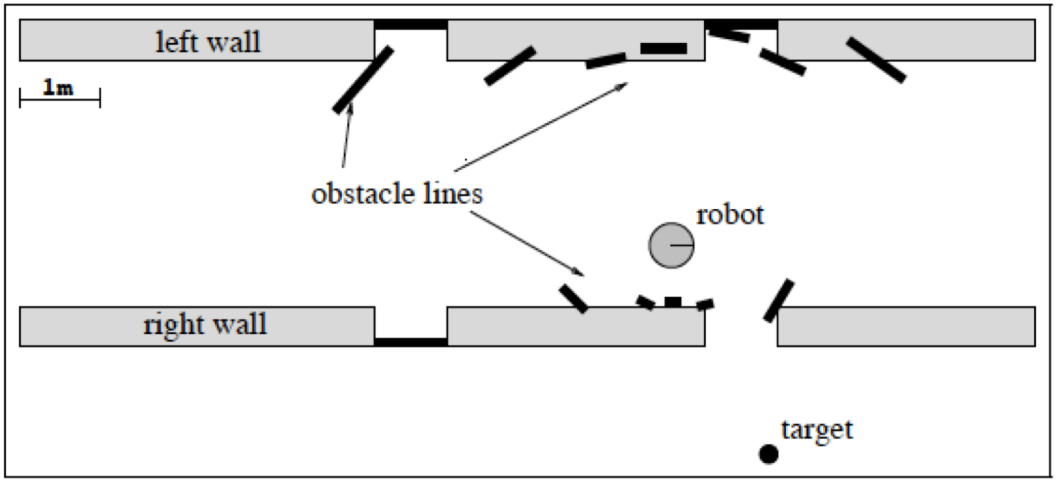


Zulässige Steuerbefehle

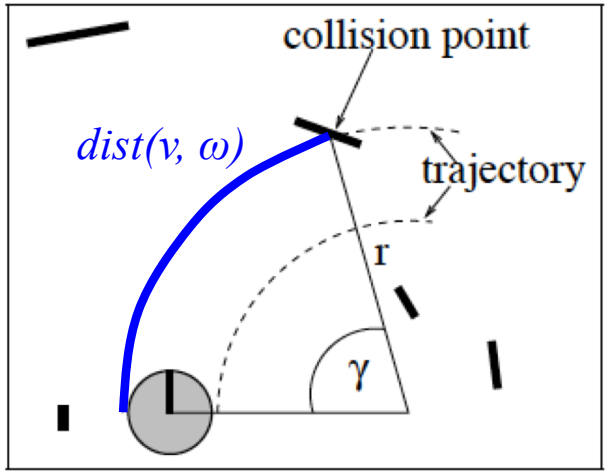
- Ein Steuerbefehl (v, ω) wird als zulässig definiert, wenn der Roboter beim Fahren des Kreisbogens, der durch (v, ω) definiert ist, rechtzeitig vor dem nächsten Hindernis bremsen kann.
- Dazu wird definiert:

  $dist(v, \omega)$ = Abstand zum nächsten Hindernis,
  beim Befahren des durch (v, ω) definierten Kreisbogens.

- Minimialer Bremsweg muss dann ≤ $dist(v, \omega)$ sein.

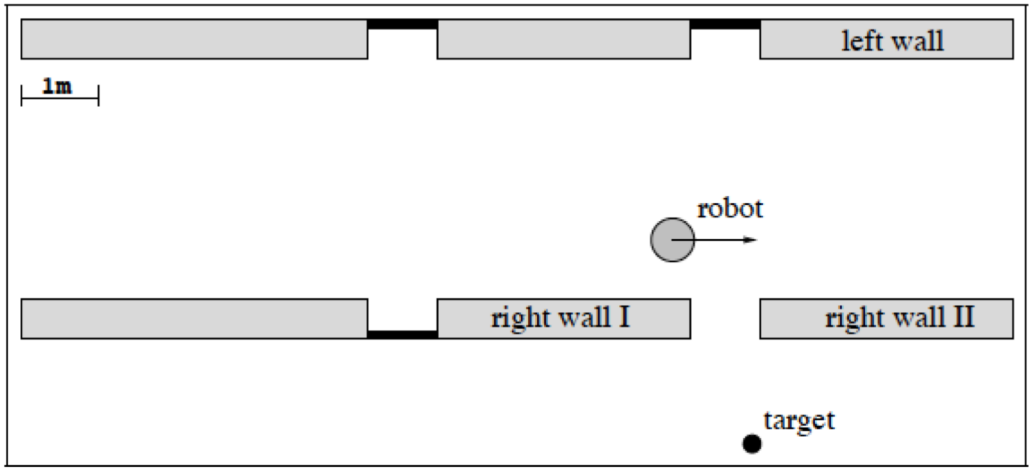


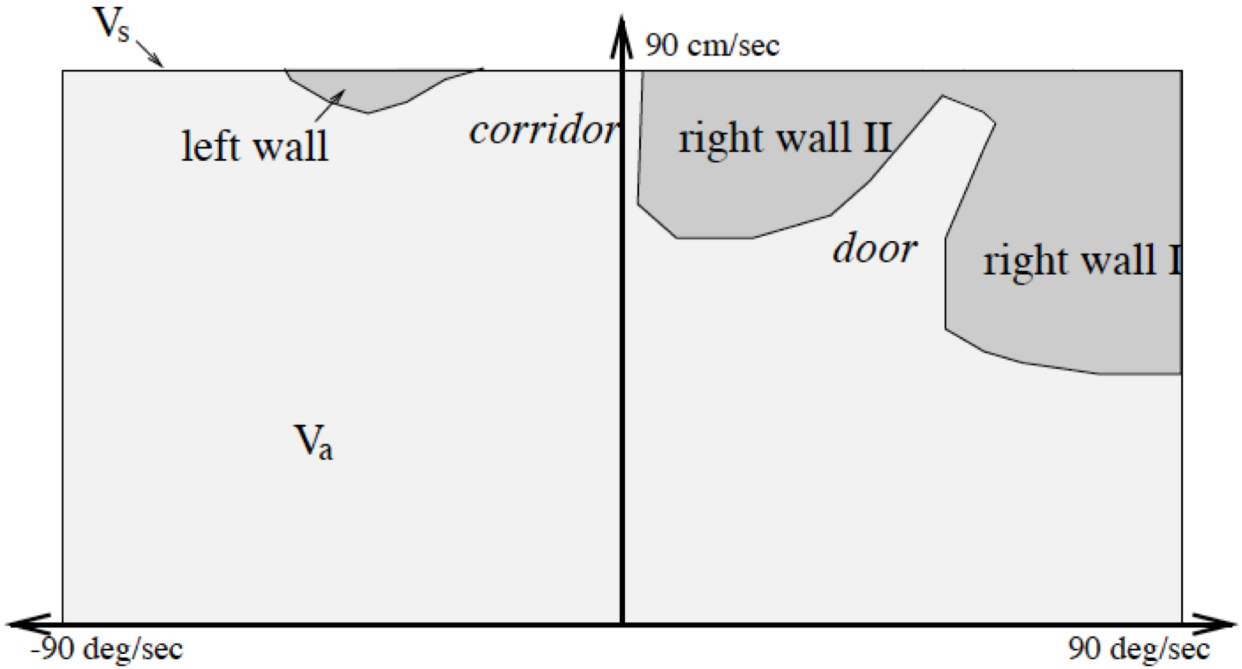
Umgebung mit Hindernispunkten



Durch Steuerbefehl (v, ω)
ist Kreisbogen mit
Radius $r = v/\omega$ definiert.

Beispiel für zulässige Steuerbefehle





- Zulässige Steuerbefehle sind hellgrau dargestellt.
- Nicht zulässige dunkelgrau.
- Maximale Beschleunigung ist dabei 0.5m/sec^2 und 60° /sec^2

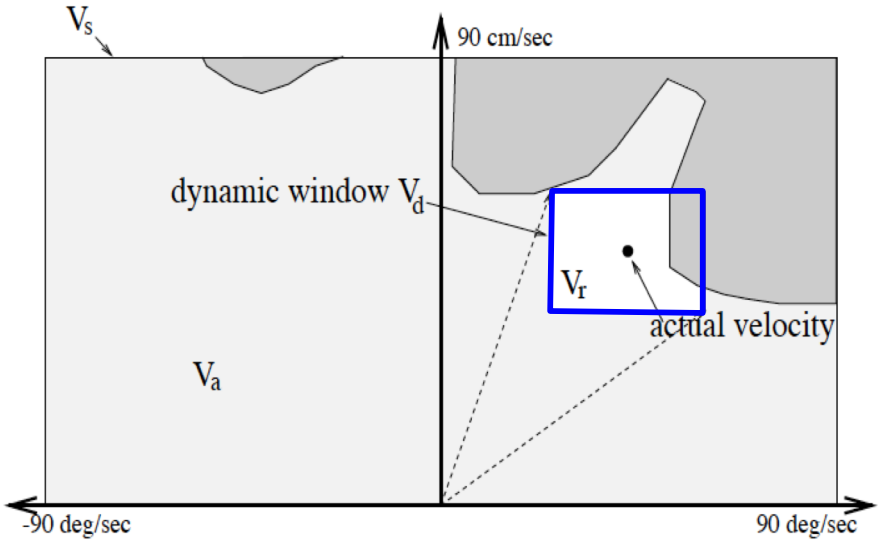
Zielfunktion

- Auf jeden Steuerbefehl im diskretisiertem Suchraum wird folgende Zielfunktion angewandt.

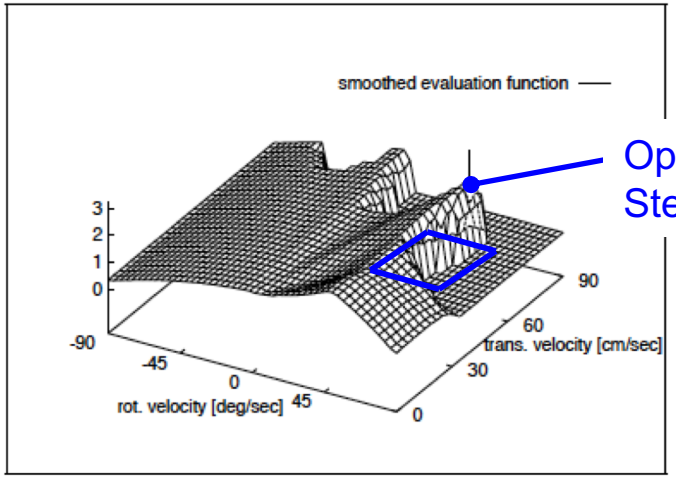
$$G(v, \omega) = a*heading(v, \omega) + b*dist(v, \omega) + c*velocity(v, \omega)$$

$heading(v, \omega)$	Zielausrichtung
$dist(v, \omega)$	Hindernisentfernung
$velocity(v, \omega) = v$	Geschwindigkeit

- Die Zielfunktion G wird geglättet und Steuerbefehl mit maximalem Zielwert gewählt.



Suchraum

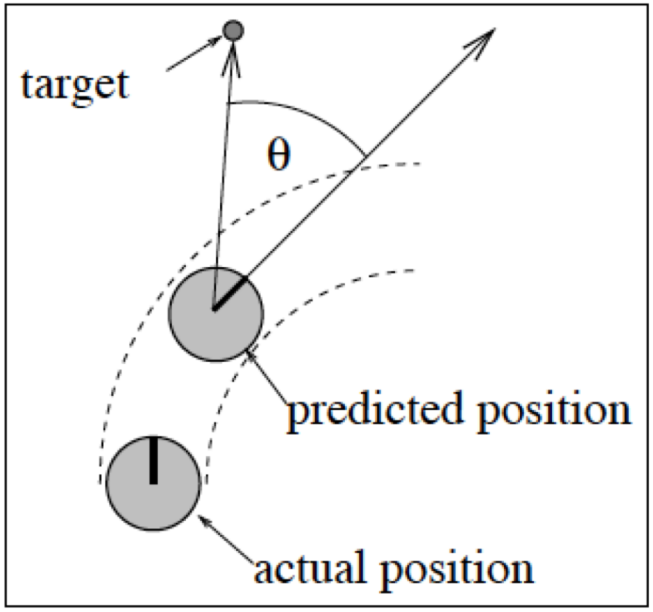


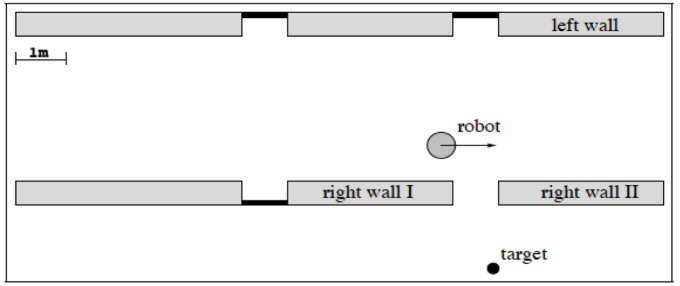
Zielfunktion G für kompletten Bereich V_S

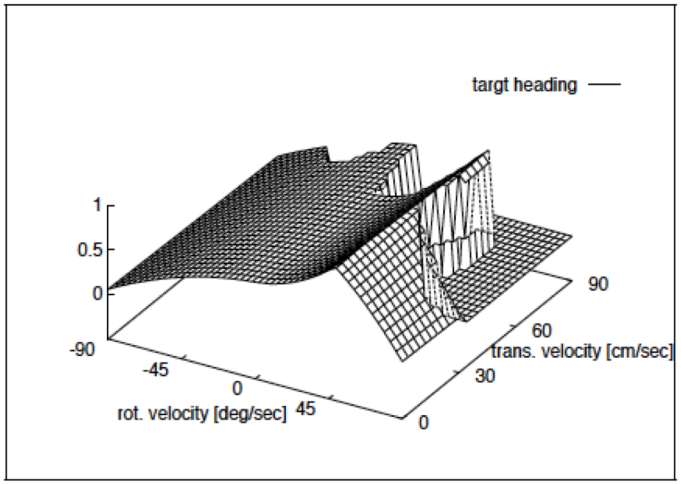
Zielfunktion - heading

heading(v, ω) =

Abweichung der Ausrichtung des Roboters von der Zielausrichtung, nachdem der Roboter den Steuerbefehl *(v, ω)* ein Zeitschritt T lang durchgeführt hat.



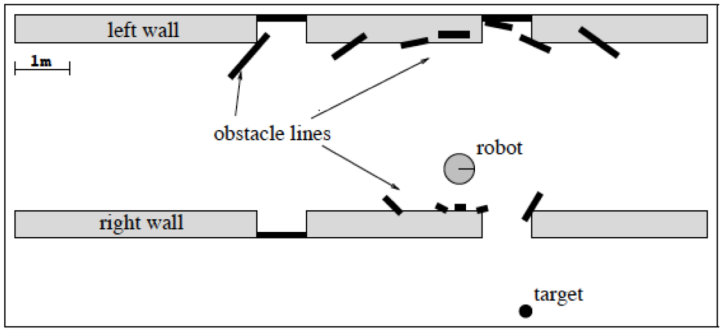




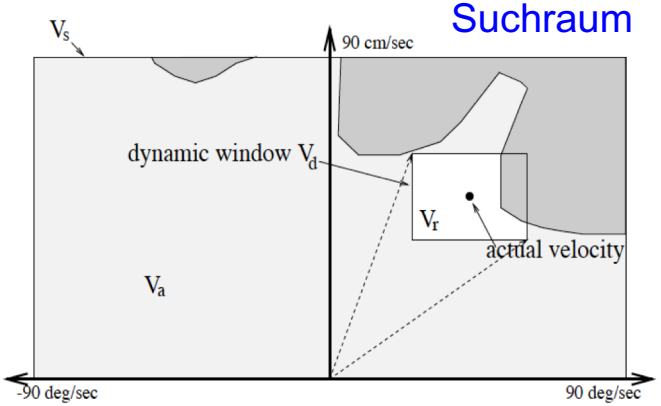
heading(v, ω)

DWA Beispiel

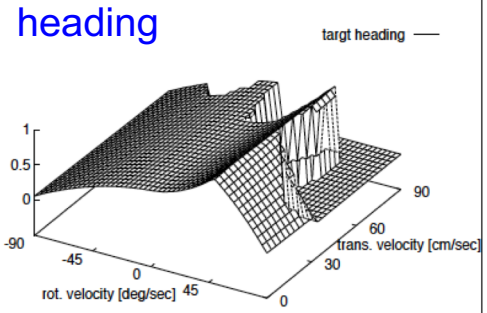
Umgebung mit Hindernissen



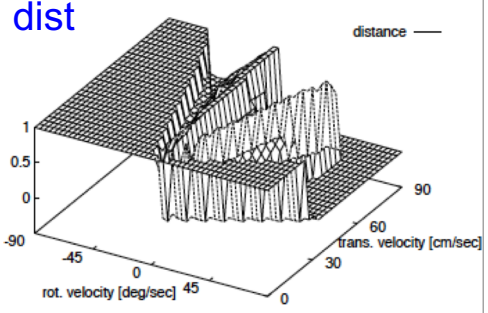
Suchraum



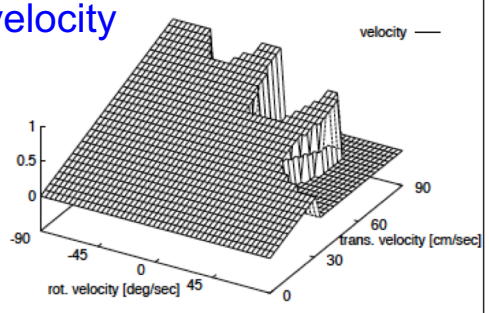
heading



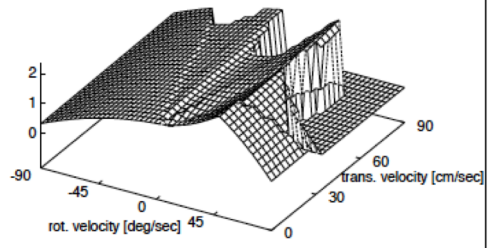
dist



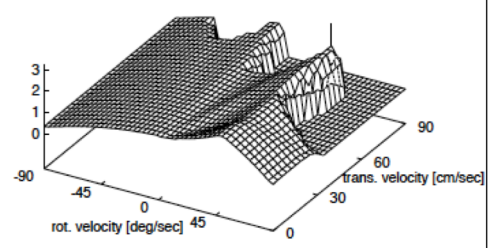
velocity



Zielfunktion G

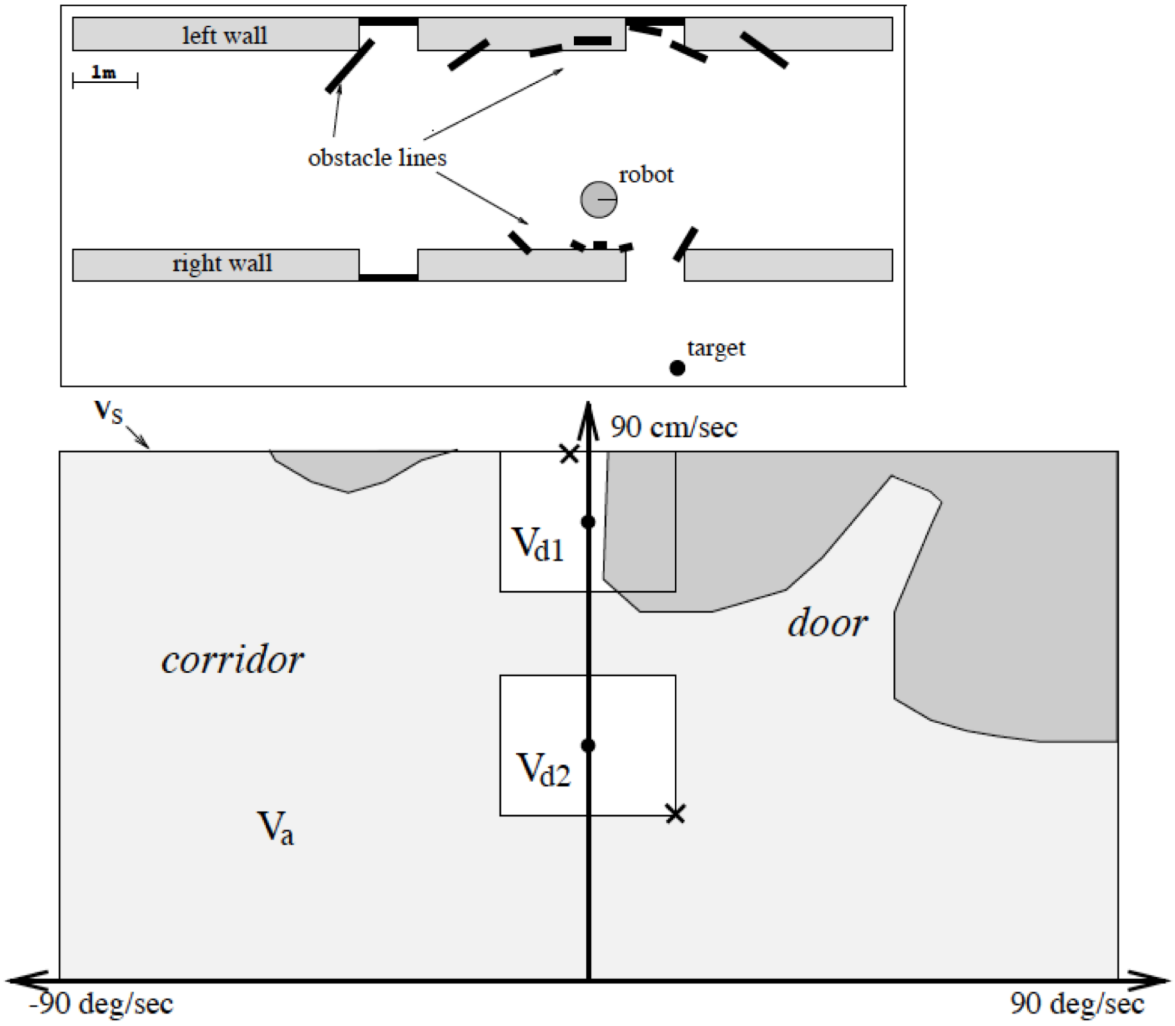


Geglättete Zielfunktion



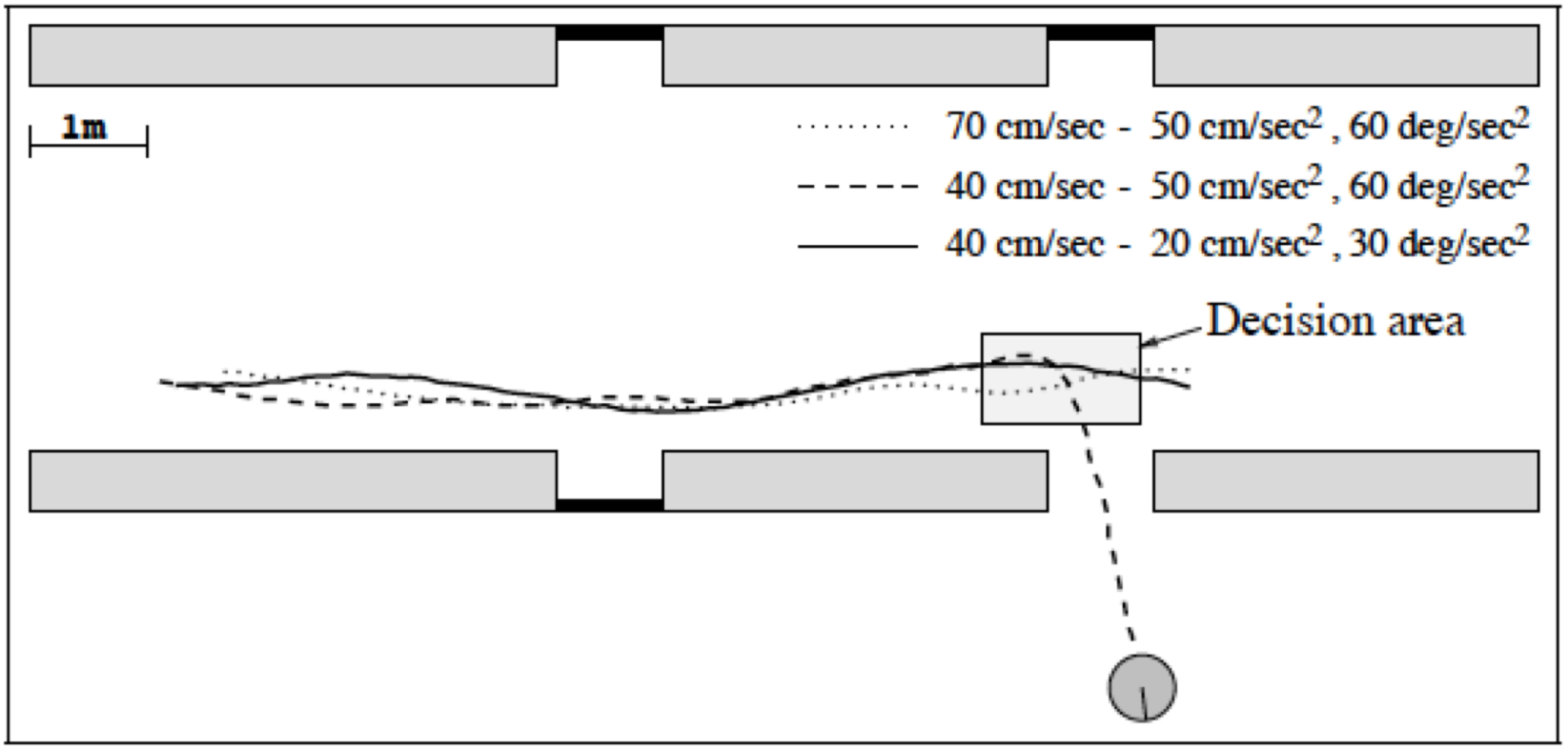
G = 2.0*heading
+ 0.2*dist
+ 0.2*velocity

DWA bei unterschiedlichen Geschwindigkeiten

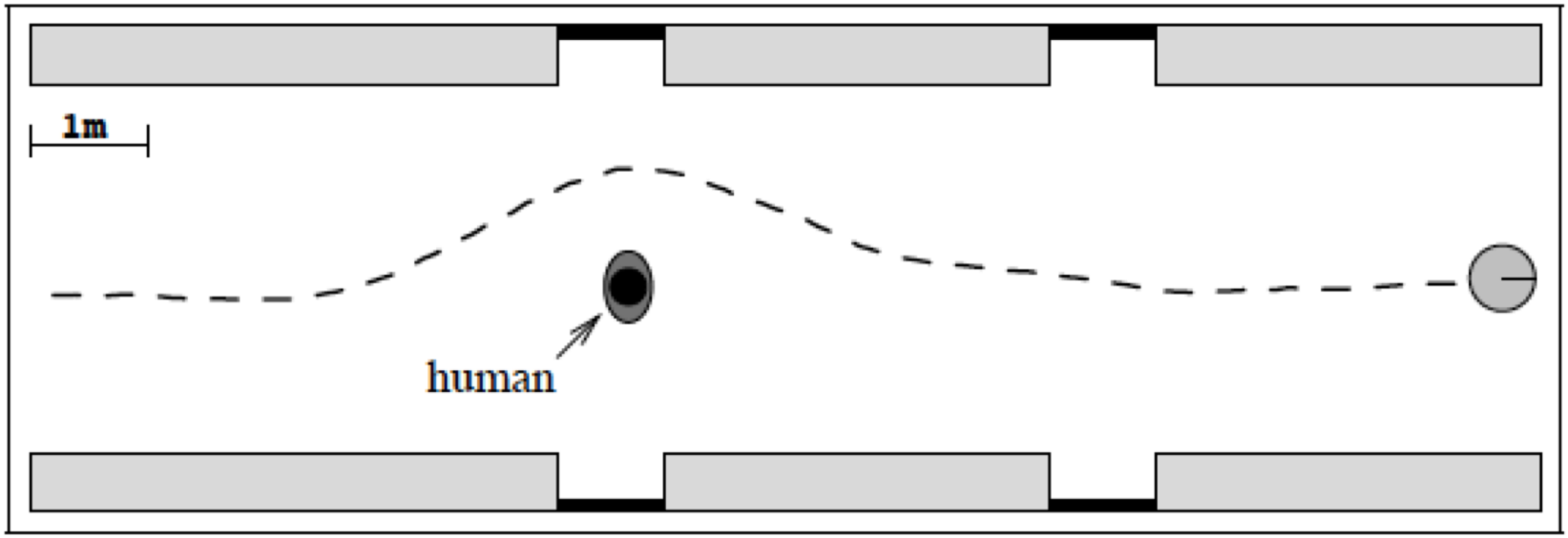


- Dynamik-Fenster bei 0.75 m/sec und 0.4 m/sec
- Berechneter optimaler Steuerbefehl durch x dargestellt.

Navigationspfade für verschiedene dyn. Bedingungen

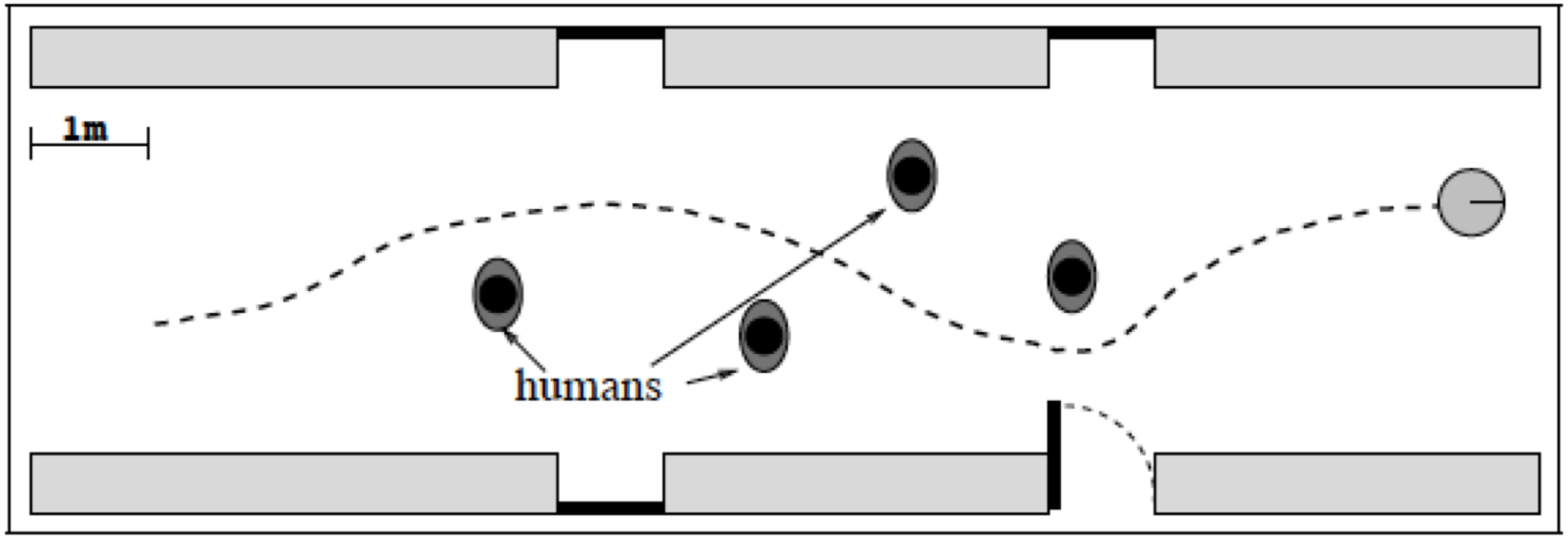


Navigationspfad mit dynamischem Hindernis



- Geschwindigkeit bei Umfahrung der Person: 0.55 m/sec
- Durchschnittgeschwindigkeit: 0.72 m/sec

Navigationspfad mit mehreren Hindernissen



- Geschwindigkeit bei Umfahrung der Personen maximal 0.95 m/sec
- Geschwindigkeit bei Engstelle an der Türe: 0.2 m/sec
- Durchschnittgeschwindigkeit: 0.65 m/sec

Navigation

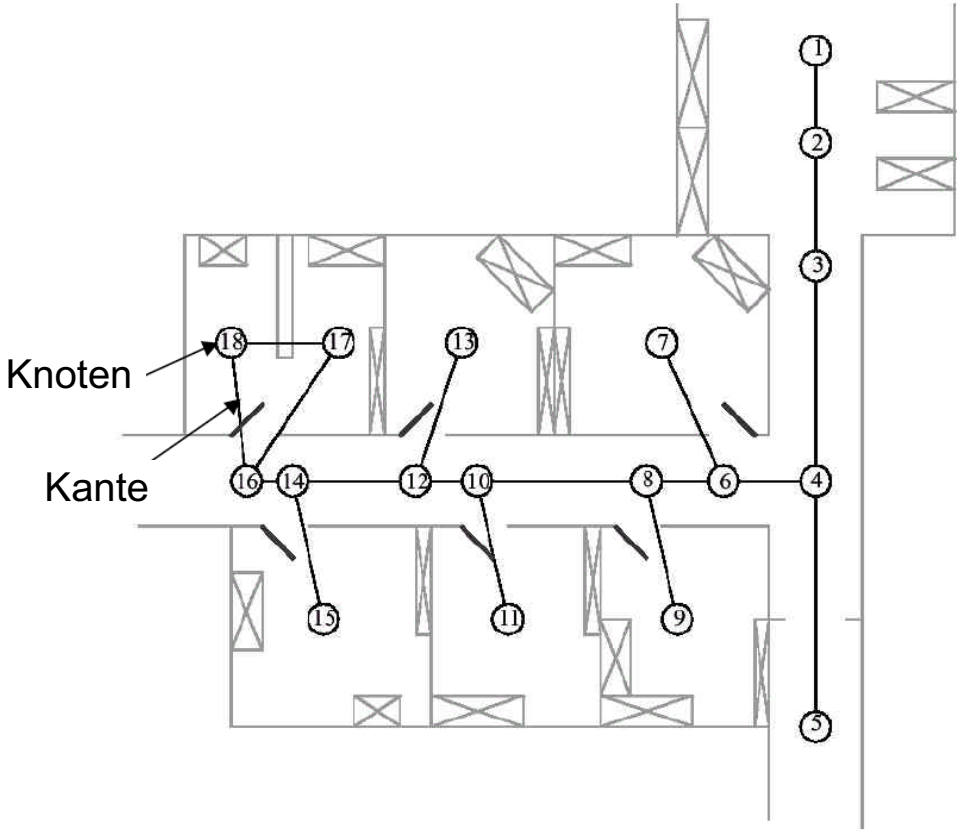
- Überblick
  - Ziel, Architektur, Komponenten
- Reaktive Navigation
  - Insekten als Vorbild
  - Braitenberg-Vehikel
  - Zusammenbau elementarer Navigationsbausteine
  - Bug-Algorithmen
  - Histogramm-Verfahren
  - Dynamik-basiertes Verfahren
- Kartenbasierte Navigation
  - Kürzeste Wege in Graphen:
    - A*, Dijkstra-Verfahren und Breitensuche
  - Verfahren mit Belegtheitsgitter
  - Wegekartenverfahren:
    - Voronoi-Diagramme und probabilistische Wegekarten

Kartenbasierte Navigation

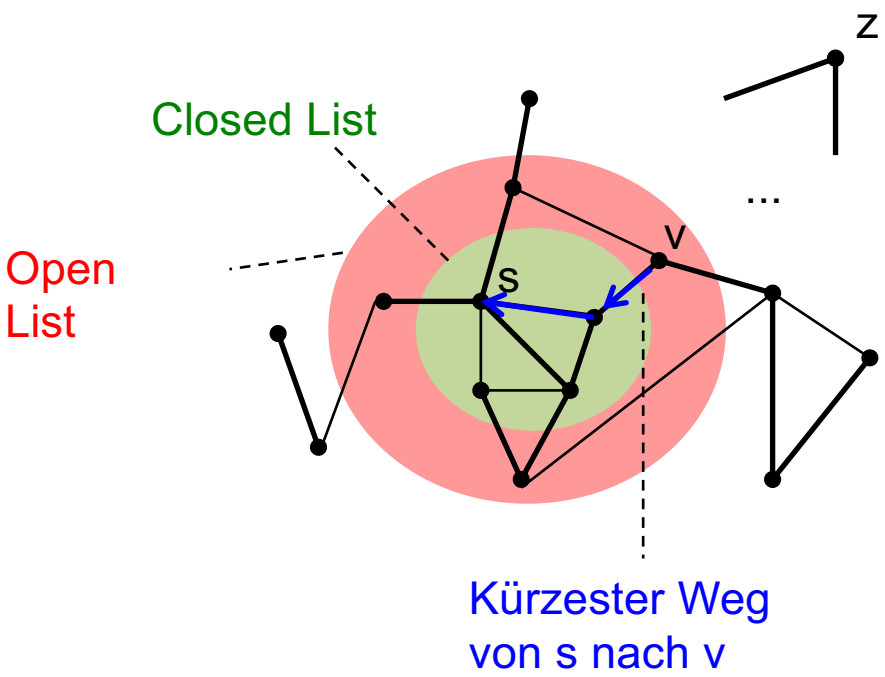
- Suche kürzesten Weg in einer Karte,
  der dann mittels reaktiver Navigation abgefahren wird.
- Kriterien für kürzesten Weg:
  - Weglänge
  - Geschwindigkeit
  - Sicherheit (Abstand von Hindernissen)
  - kinematische Befahrbarkeit
- Wichtige Modelle für Karten:
  - Graphen
  - Belegtheitsgitter

Graphen

- Graph besteht
  aus einer Menge V
  von Knoten (verteces)
  und einer Menge E
  von Kanten (edges).
- Knoten u und v, die einfach gegenseitig erreichbar sind, sind durch eine Kante (u,v) verbunden.
- Jeder Kante (u,v) sind Kosten c(u,v) zugeordnet.
- Zusätzlich können die Knoten noch mit geometrischen Informationen wie z.B. (x,y)-Koordinaten versehen sein.



Kürzeste Wege mit A* (1)

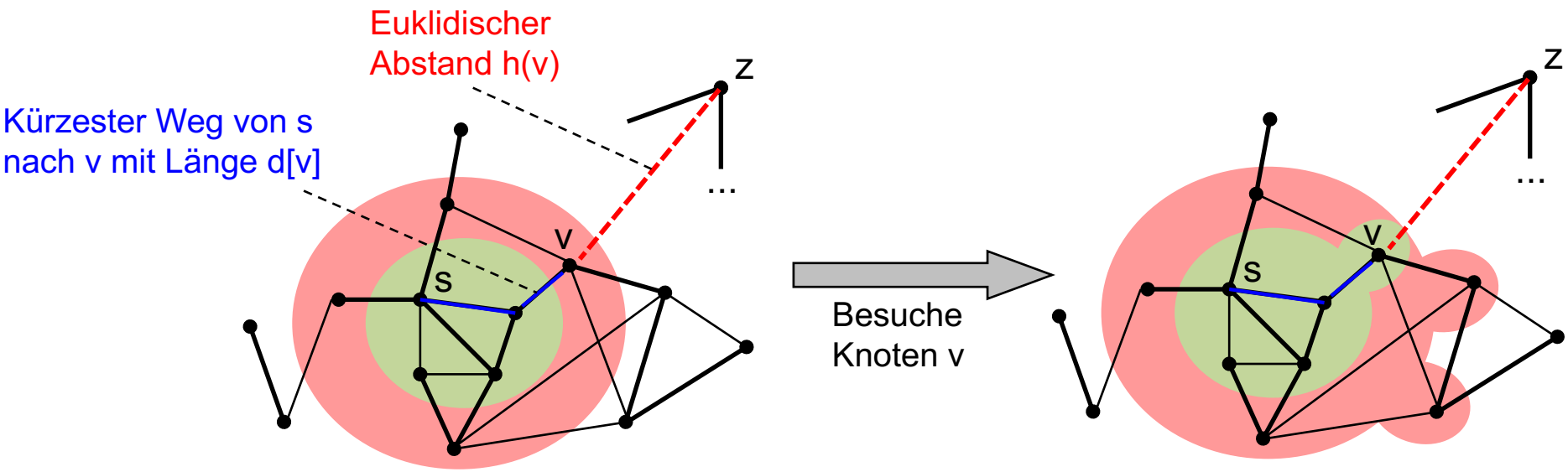


- Suche kürzesten Weg von Startknoten s nach Zielknoten z.
- Closed List:
  Knoten wurden bereits besucht und kürzeste Wege sind bekannt.
- Open List:
  Knoten, die als nächstes besucht werden können.
  Es gibt Schätzwerte für kürzeste Wege, die sich aber noch ändern können.
- Unbekannte Knoten:
  Die restlichen Knoten (Knoten mit weisser Hintergrund) sind noch unbekannt.

- Speicherung der kürzesten Wege:

  d[v] = Länge des kürzesten Wegs von s nach v
  p[v] = Vorgänger auf dem kürzesten Weg von s nach v

Kürzeste Wege mit A* (2)



- aus der Open List wird derjenige Knoten v besucht, für den der folgende Wert minimal ist:

  d[v] + h(v)

- h(v) ist eine Heuristik, die die noch unbekannte Weglänge von v nach z abschätzt. Sehr häufig wird der Euklidische Abstand gewählt.

- v kommt zur Closed List dazu. Es werden alle Nachbarn w von v betrachtet.
  Falls w unbekannt ist, dann kommt w zur Open List dazu.
  Falls w schon in der Open List ist, dann wird überprüft, ob sich d[w] verbessert.
  d[w] und p[w] werden neu gesetzt bzw. aktualisiert.

A*-Algorithmus

Sucht kürzesten Weg von Startknoten s nach Zielknoten z im Graph g.

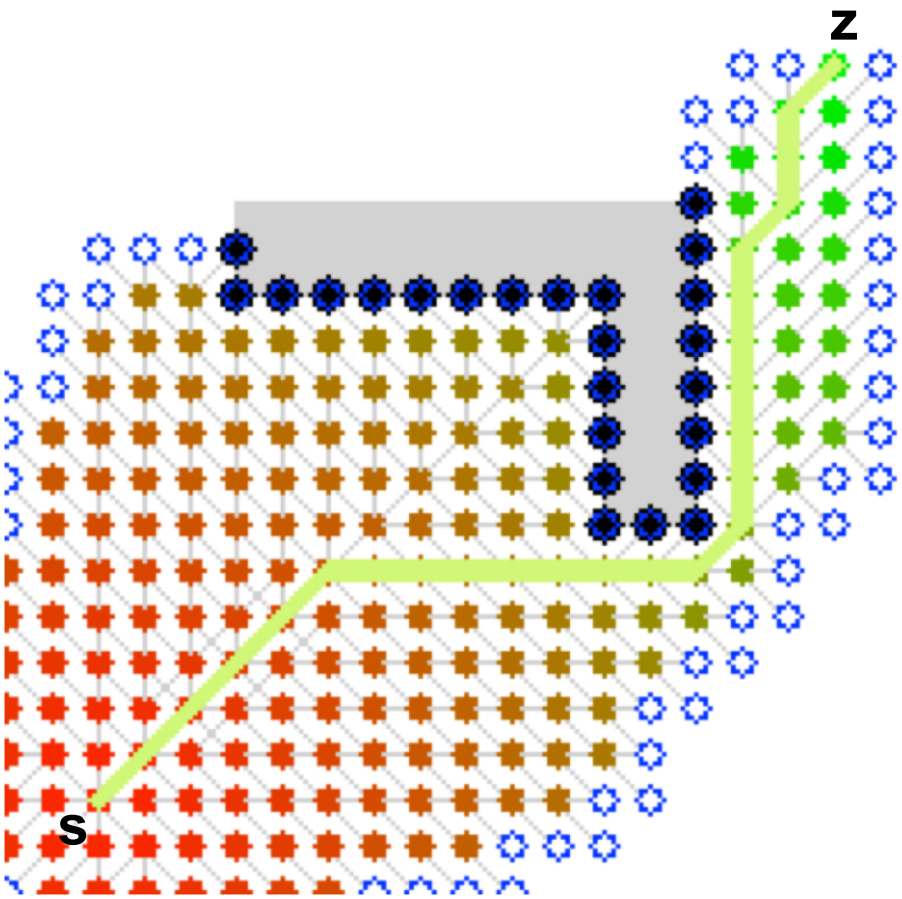
```
void shortestPath (Vertex s, Vertex z, Graph g)
{
        PriorityQueue openList;          // Open List wird als Prioritätsliste realisiert und
                                         // enthält eine Menge von Knoten v mit Prioritätswert d[v] + h(v).
        openList.insert(s, h(s));        // Fügt Startknoten s mit Prioritätswert h(s) ein.
        d[s] = 0;

        while (! openList.empty() ) {
            v = openList.delMin();       // Knoten v wird besucht; kürzester Weg ist jetzt bekannt.
            if (v == z)                  // Ziel erreicht.
                breche Suche erfolgreich ab;
            for (jeden Nachbarn w von v ) {
                if (d[w] ist noch undefiniert) {      // Knoten w ist noch unbekannt.
                    d[w] = d[v] + c(v,w);
                    p[w] = v;
                    openList.insert(w, d[w]+h(w));
                }
                else if (d[v] + c(v,w) < d[w]) {      // d[w] kann verbessert werden mit Knoten v als Vorgänger.
                    d[w] = d[v] + c(v,w);
                    p[w] = v;
                    openList.changePriority(w, d[w]+h(w));
                }
            }
        }
}
```

Implementierungshinweise

- Für die Open List wird eine indizierte Prioritätsliste verwendet
  mit effizienten Realisierungen (d.h. O(log(n)) für die Operationen:
  - delMin()
  - insert(v, prio)
  - changePriority(v, prio)
- d[v] und p[v] können mit Hilfe eines Map-Datentyps realisiert werden.
- In der vorliegenden Algorithmus-Variante ist die Closed List
  (= Menge der besuchten Knoten) nicht explizit realisiert.
  Sie ist aber implizit gegeben:
  jeder Knoten v, für den d[v] definiert ist und
  der nicht in der Open List ist, ist in der Closed List.
- In machen A*-Implementierungen wird die Closed List auch
  explizit als Set-Datentyp realisiert.
  Sobald ein Knoten v aus der Open List mit delMin() entfernt wird,
  wird v in die Closed List hinzugefügt.

Beispiel



https://de.wikipedia.org/wiki/A*-Algorithmus,
User:Subh83

- Blau umrandete Knoten:
  Open List
- Farbige Knoten:
  Closed List (besuchte Knoten)
  Farbwert spiegelt Distanz zu Startknoten s wider.
- Restliche Knoten:
  nicht dargestellt.

Eigenschaften von A* (1)

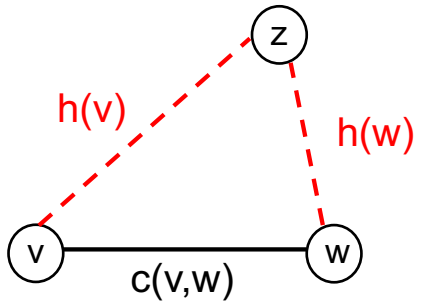
- A* ist vollständig:
  falls ein Weg von S nach Z existiert, wird auch ein Weg gefunden.

- A* ist optimal:
  der gefundene Weg ist ein kürzester Weg.

- Voraussetzung dafür sind zwei Bedingungen an die Heuristik h:

  (1) h ist zulässig:
  $0 \leq h(v) \leq d(v,z)$ für alle Knoten v.
  Dabei ist d(v,z) die Länge eines kürzesten Wegs von v zum Ziel z.
  D.h. h unterschätzt die Länge des kürzesten Weges von v nach z.

  (2) h ist monoton:
  $h(v) \leq c(v,w) + h(w)$ für alle Knoten v, w



Eigenschaften von A* (2)

- Eine monotone Heuristik gewährleistet, dass entlang jeden von A* berechneten Wegs (Datenstruktur p) die Prioritätswerte monoton steigend sind.
- Damit müssen die Prioritätswerte der besuchten Knoten über die Zeit aufgetragen monoton steigend sein. Folgende Schleife würde also steigende Prioritätswerte ausgeben:

```
while (! openList.empty() ) {
        v = openList.delMin();     // besuche Knoten v
        gibt Prioritätswert von v aus: d[v] + h(v)

        // ...
}
```

- Daher ist es überflüssig, einen bereits besuchten Knoten nochmals zu besuchen.
- Der Forderung einer monotonen Heuristik kann auch wegfallen.
  Dann muss der A*-Algorithmus allerdings so umformuliert werden,
  dass auch bereits besuchte Knoten nochmals besucht werden können.
  Das würde jedoch die Laufzeit verschlechtern.
- Die wichtigsten Heuristiken wie Euklidischer Abstand sind monoton.

Dijkstra-Algorithmus und Breitensuche

- Die Heuristik

  h(v) = 0 für alle Knoten v

  ist zulässig und monoton.

  Man erhält damit den Algorithmus von Dijkstra, der kürzeste Wege berechnet ohne Zielorientierung.

  Das Dijkstra-Verfahren kann, auch wenn der Zielknoten erreicht wurde, fortgesetzt werden. Dann werden alle kürzesten Wege mit Startknoten s gefunden.

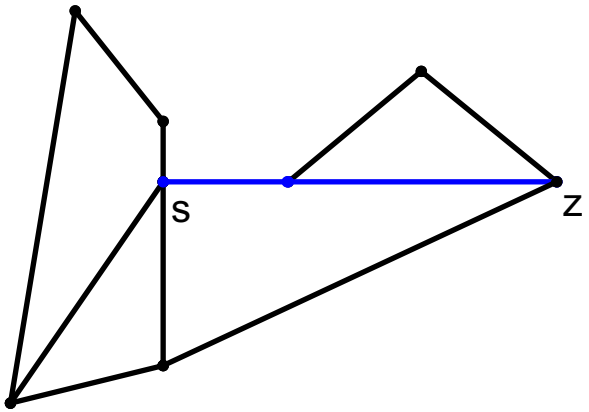
- Falls h(v) = 0 und alle Kanten das Gewicht 1 haben (d.h. c(u,v) = 1), ergibt sich die Breitensuche:
  zuerst werden die Knoten besucht, die über eine Kante direkt erreichbar sind, dann die Knoten die über 2 Kanten erreichbar sind, etc.

  Statt einer Prioritätsliste kann dann eine einfache Schlange (Queue) verwendet werden.

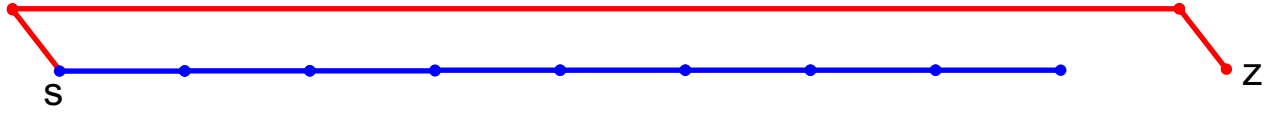
Komplexität von A* (1)

- Die Laufzeit von A* hängt ganz wesentlich davon ab, wie gut die Heuristik zur Graphstruktur passt.

- Stellt ein Graph befahrbare Wege in einer Umgebung dar, dann führt der Euklidische Abstand als Heuristik in der Regel zu einer kurzen Laufzeit von A*.



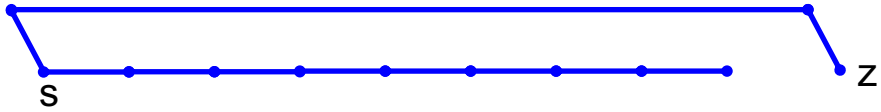
- Das muss aber nicht so sein! Man stelle sich Sackgassen oder ein Labyrinth vor.



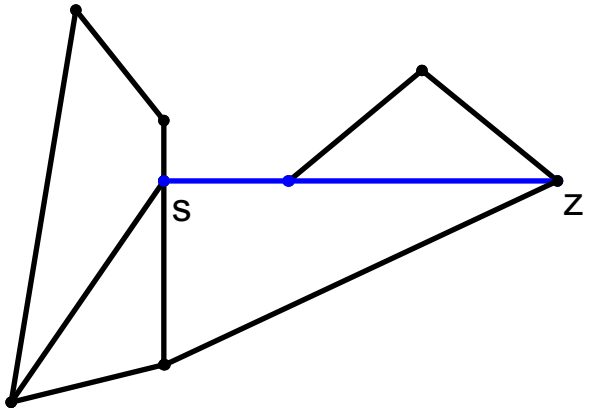
1. A* geht direkt auf das Ziel z zu, läuft aber in eine Sackgasse (blau).
2. A* besucht nun einen Weg, der sich anfangs vom Ziel entfernt und daher nicht früher betrachtet wurde (rot).

Komplexität von A* (2)

- Laufzeit im schlechtesten Fall:
  - Heuristik bringt keinen Vorteil (siehe Sackgassen-Beispiel).
  - Kein frühzeitiger Abbruch, weil das Ziel gefunden wurde. (Fast) jeder Knoten v wird besucht.
  - Ist der Graph dünn besetzt und wird eine indizierte Prioritätsliste eingesetzt, dann ist T = O(n log(n)), wobei n die Anzahl der Knoten ist.



- Laufzeit im besten Fall:
  - Heuristik ist optimal.
  - Es werden nur solche Knoten v besucht, die für den kürzesten Weg relevant sind.
  - Ist der Graph dünn besetzt, dann gilt T = O(d log(d)), wobei d die Anzahl der Knoten des gefundenen kürzesten Weges ist.

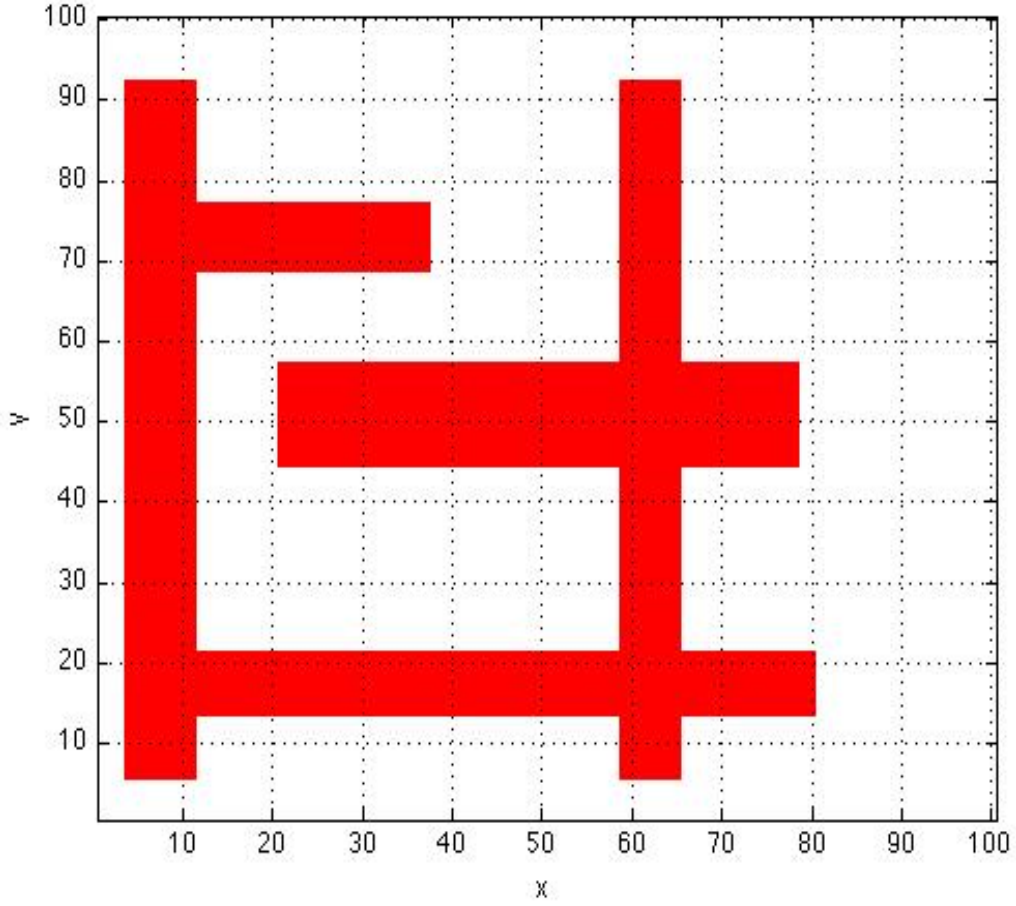


Navigation

- Überblick
  - Ziel, Architektur, Komponenten
- Reaktive Navigation
  - Insekten als Vorbild
  - Braitenberg-Vehikel
  - Zusammenbau elementarer Navigationsbausteine
  - Bug-Algorithmen
  - Histogramm-Verfahren
  - Dynamik-basiertes Verfahren
- Kartenbasierte Navigation
  - Kürzeste Wege in Graphen:
    - A*, Dijkstra-Verfahren und Breitensuche
  - Verfahren mit Belegtheitsgitter
  - Wegekartenverfahren:
    - Voronoi-Diagramme und probabilistische Wegekarten

Belegtheitsgitter

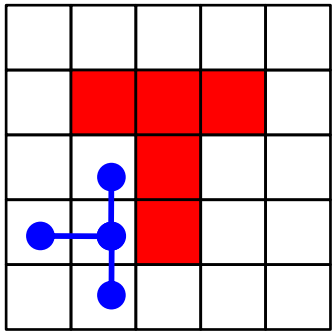
- Umgebung wird in ein Gitter von Zellen mit gleicher Größe aufgeteilt.
- Eine Zelle ist belegt, wenn sie durch ein Hindernis geschnitten wird.



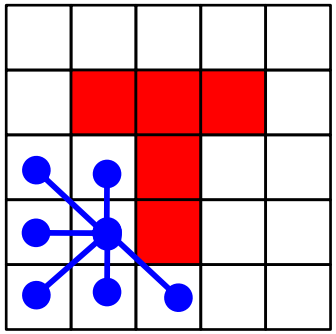
Gitter mit
100*100 Zellen

Belegtheitsgitter als Graph

- Ein Belegtheitsgitter kann als Graph aufgefasst werden.
- Jede freie Zelle bildet einen Knoten.
- 4-Nachbarschaft: jede horizontal bzw. vertikal benachbarte freie Zelle ergibt eine Kante.
- 8-Nachbarschaft: auch diagonal benachbarte freie Zellen ergeben eine Kante.
- Vertikale und horizontale Kanten können mit Kosten 1 und diagonale Kanten mit Kosten $\sqrt{2}$ belegt werden.



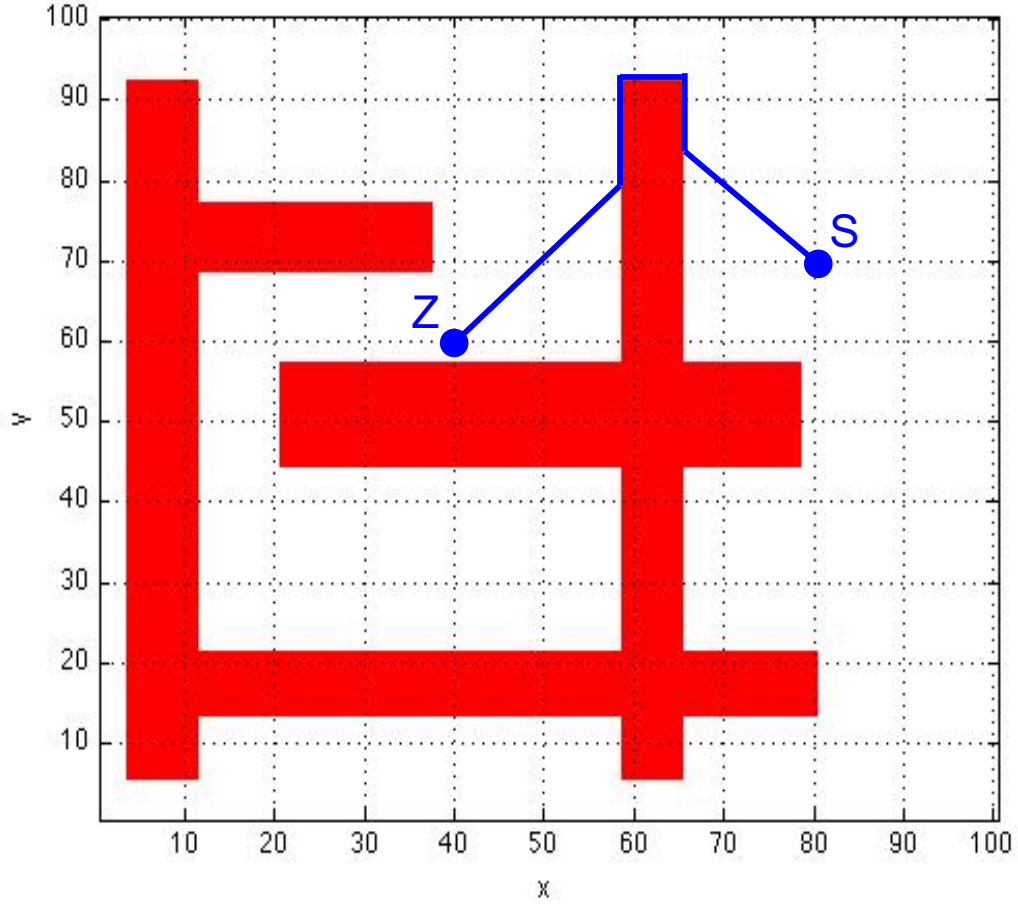
4-Nachbarschaft



8-Nachbarschaft

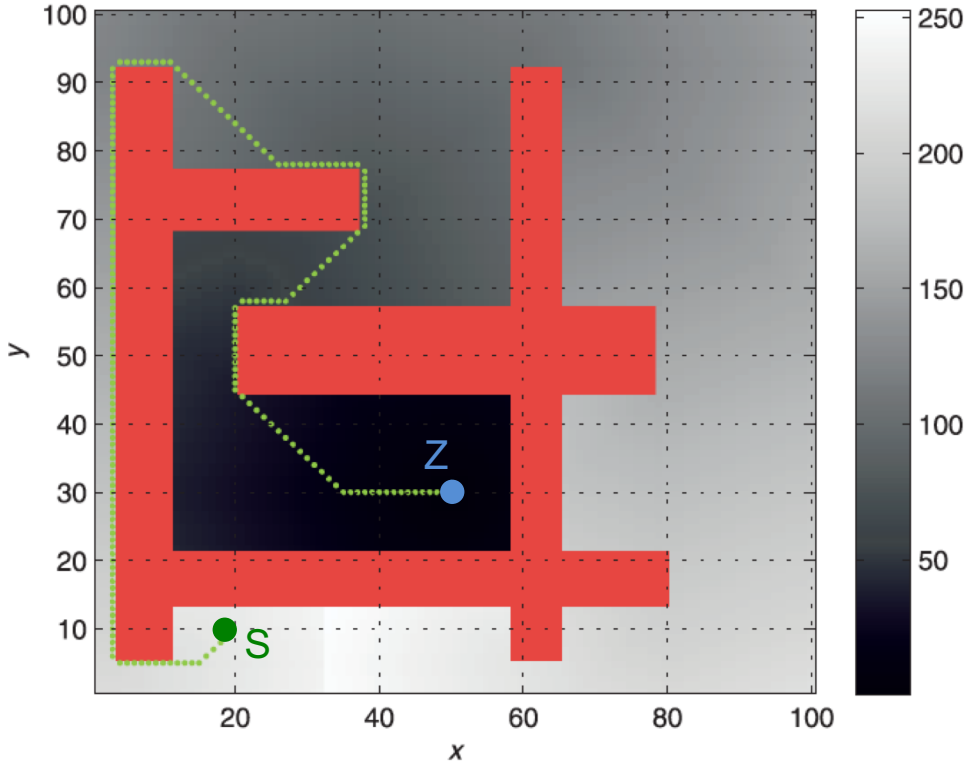
Gitterbasiertes Planungsverfahren

- Fasse Belegtheitsgitter als Graph auf und berechene mit A*-Verfahren kürzesten Weg von Start nach Ziel.



Gradientenplaner

- Fasse Belegtheitsgitter als Graph auf und berechne mit Dijkstra-Verfahren alle kürzesten Wege vom Ziel nach allen anderen Knoten und trage die berechneten kürzesten Weglängen d in die entsprechenden Gitterzellen ein.
- Gesuchter Weg ergibt sich dann, indem bei einem beliebigen Startknoten S begonnen wird und immer zu derjenigen Nachbarzellen mit geringstem d-Wert gegangen wird (Gradientenabstieg).

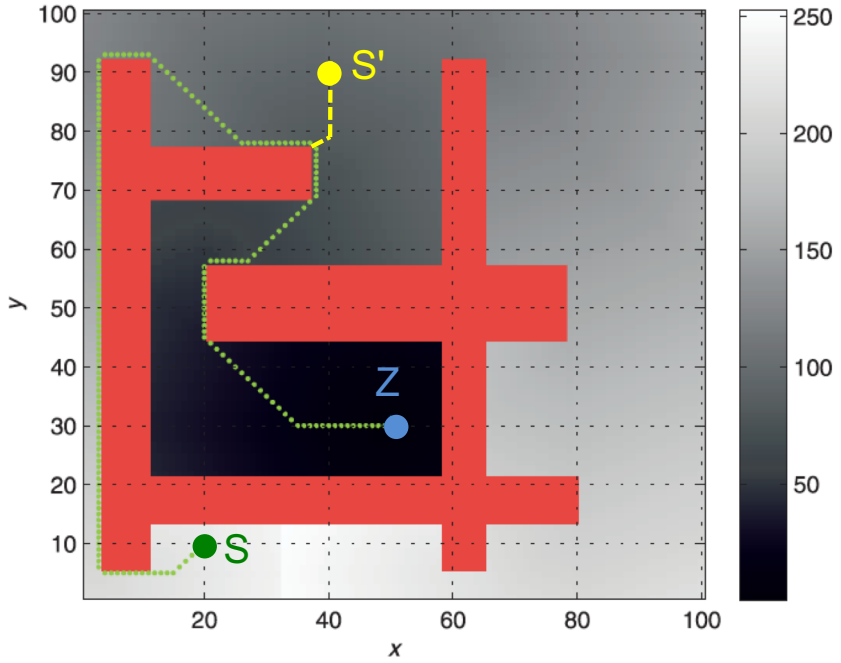


[Corke, 2011]

- Belegtheitsgitter mit 8-Nachbarschaft.
- Grauwert stellt die berechneten kürzesten Weglängen dar.

Bemerkungen

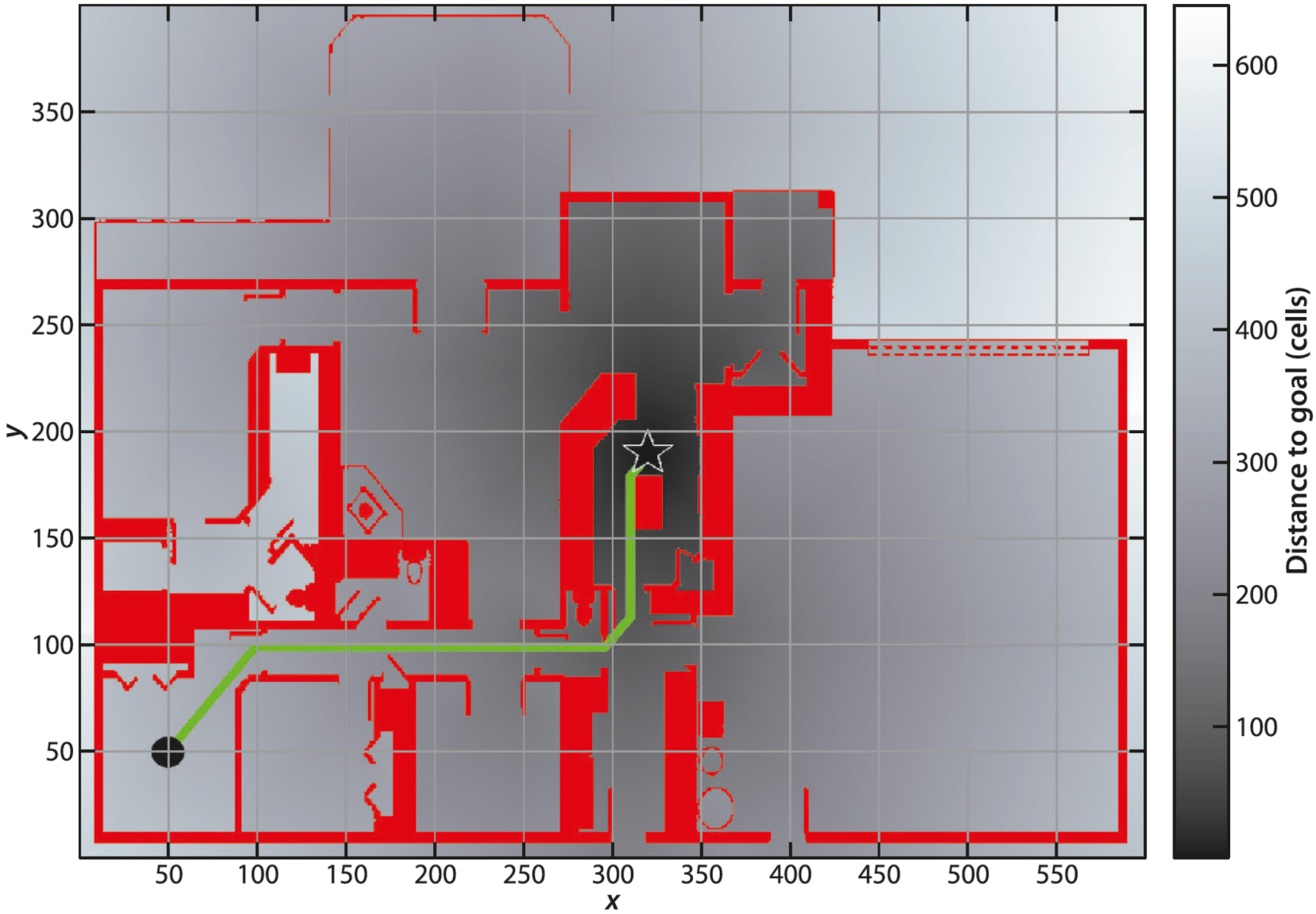
- Sobald die kürzesten Wege berechnet wurden, ist für jeden Startknoten (!) der kürzeste Weg unmittelbar durch Gradientabstieg gegeben.
- Vorteil ist: bei einer Abweichung vom geplanten Weg steht eine Neuplanung des Wegs sofort zur Verfügung. Das Ziel darf sich dabei jedoch nicht ändern.



[Corke, 2011]

- Zusätzlich können Zellen noch mit Kosten versehen werden, die vom Abstand zum nächsten Hindernis abhängen. Damit werden bei der Pfadplanung zwei Ziele verfolgt: Sicherheit der Wege (d.h. größere Abstände von Hindernissen) und ihre Länge.

Wohnungsszenario mit Gradientenplaner



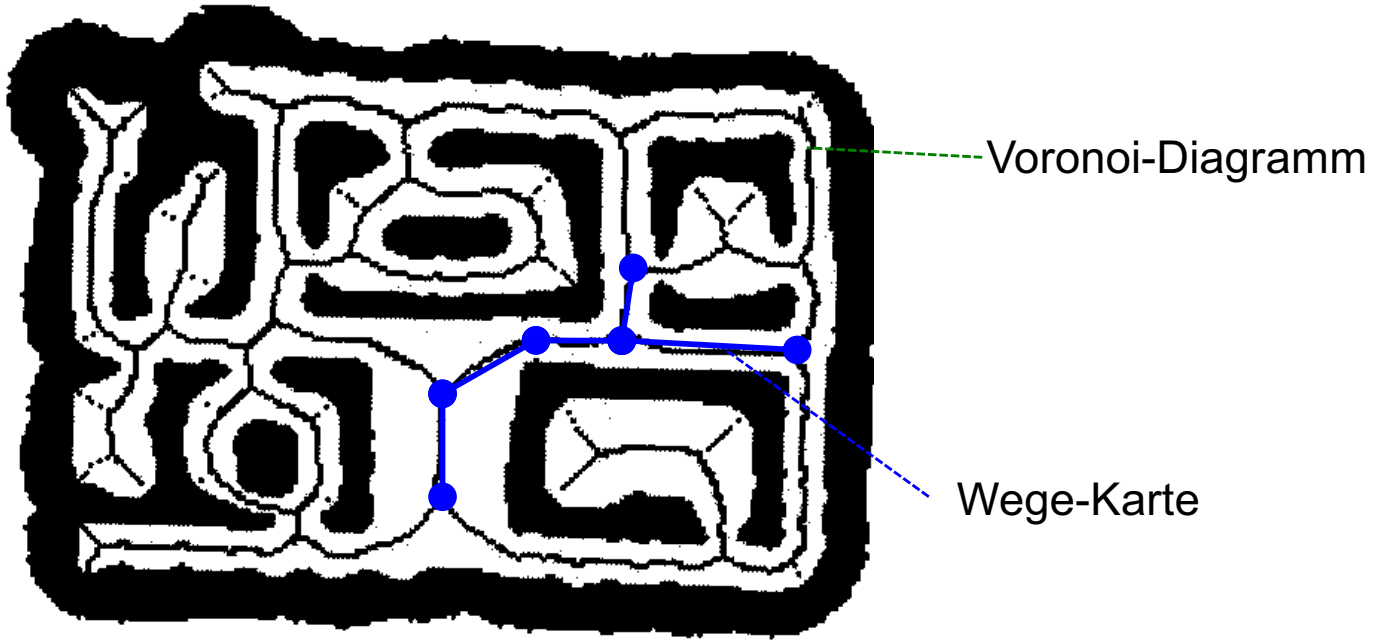
[Corke, 2017]

Navigation

- Überblick
  - Ziel, Architektur, Komponenten
- Reaktive Navigation
  - Insekten als Vorbild
  - Braitenberg-Vehikel
  - Zusammenbau elementarer Navigationsbausteine
  - Bug-Algorithmen
  - Histogramm-Verfahren
  - Dynamik-basiertes Verfahren
- Kartenbasierte Navigation
  - Kürzeste Wege in Graphen:
    - A*, Dijkstra-Verfahren und Breitensuche
  - Verfahren mit Belegtheitsgitter
  - Wegekartenverfahren:
    - Voronoi-Diagramme und probabilistische Wegekarten

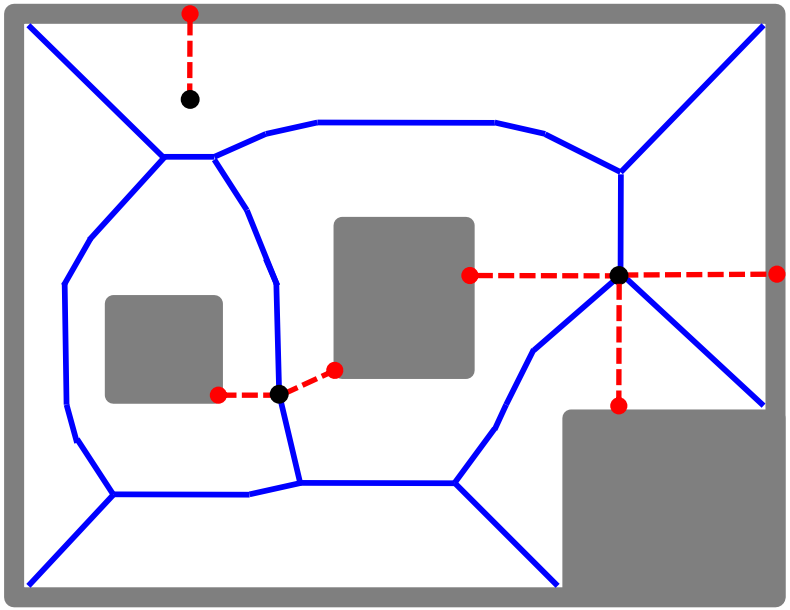
Ziel der Voronoi-Diagramme

- In frei befahrbare Umgebung werden Knoten und Verbindungen gelegt (Voronoi-Diagramm), die möglichst weit weg von den Hindernissen sind.
- Abstrahiere Voronoi-Diagramm zu Graph (Wegekarte), und führe Navigationsplanung auf Graphenalgorithmen zurück.



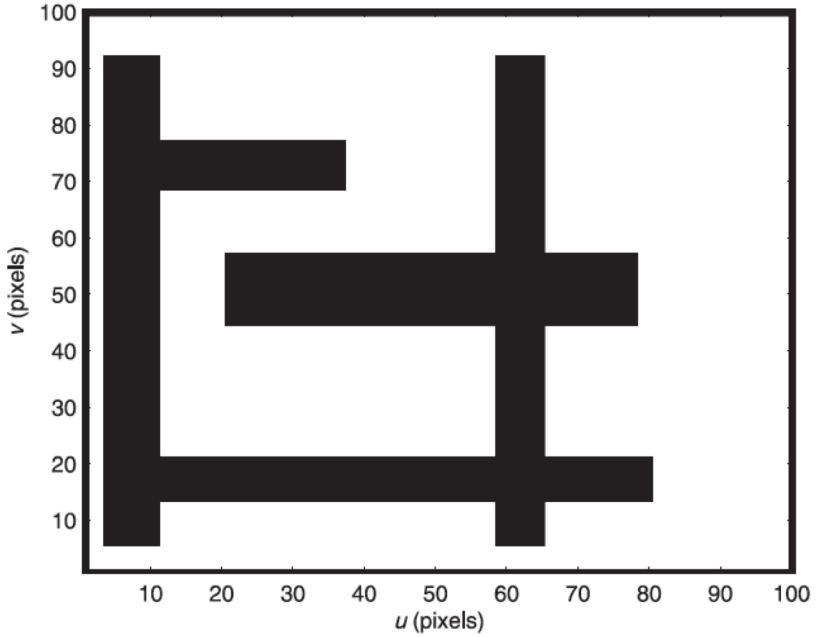
Definition der Voronoi-Diagramme

- Zu einem Punkt x der freien Umgebung heisst ein nächstgelegener Hindernispunkt auch Basispunkt.
- Es kann zu einem Punkt mehrere Basispunkte geben, die dann alle gleich weit entfernt sind.
- Ein Voronoi-Diagramm besteht aus allen Punkten, die wenigstens 2 Basispunkte besitzen.
- Punkte mit echt mehr als 2 Basispunkten heissen auch Voronoi-Ecken.



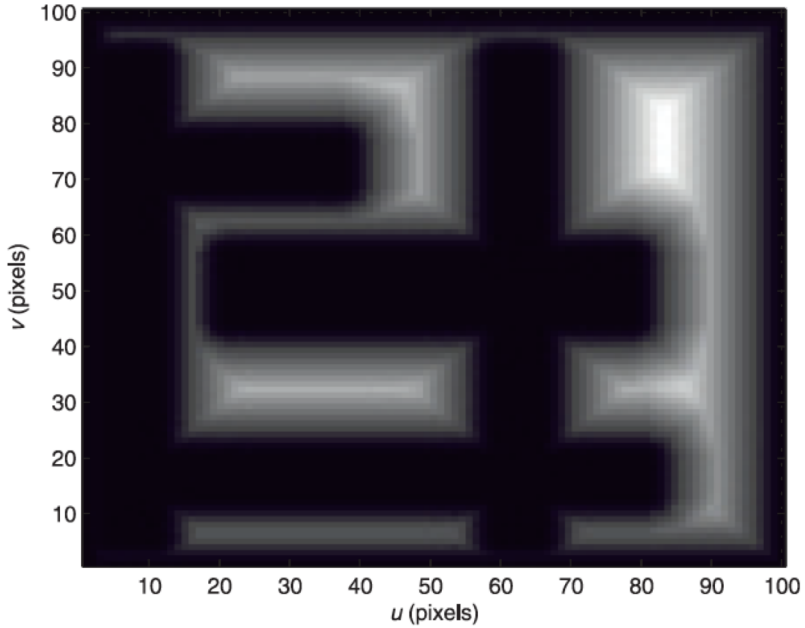
Approximative Berechnung der Voronoi-Diagramme (1)

- Erstelle zur Umgebung ein Belegtheitsgitter.
- Ermittle alle Randzellen. Randzellen sind Zellen, die zu einem Hindernisrand gehören.
- Starte nun das Dijkstra-Verfahren, wobei Open List mit allen Randzellen initialisiert wird. Der d-Wert der Randzellen wird auf 0 gesetzt.
- Das Dijkstra-Verfahren berechnet nun für alle Zellen den Abstand zu den nächstgelegenen Hindernissen.



[Corke, 2011]

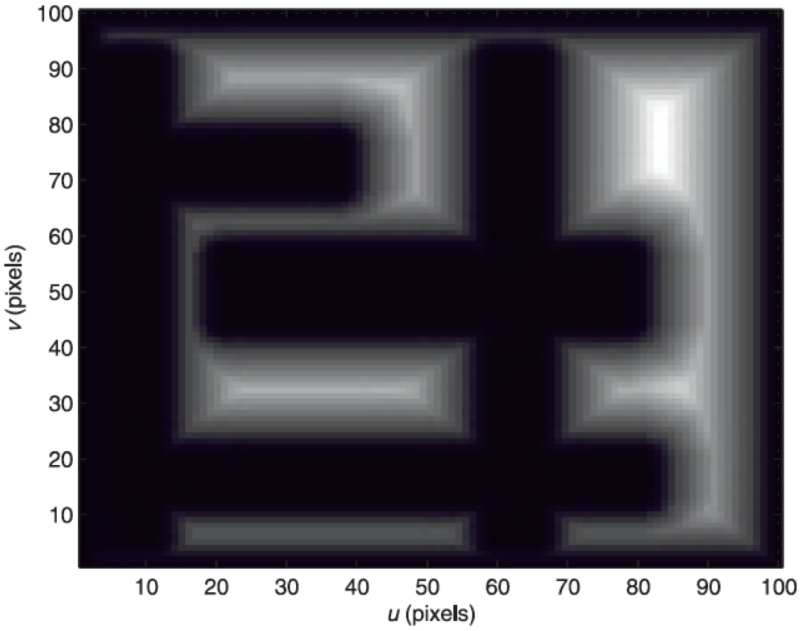
Belegtheitsgitter



Gitter mit Entfernungswerten zu den nächstgelegenen Hindernissen

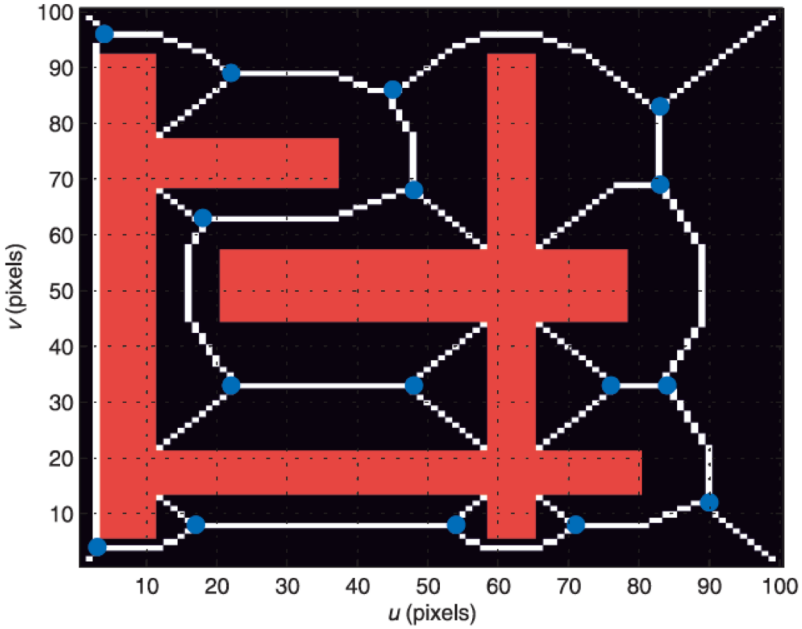
Approximative Berechnung der Voronoi-Diagramme (2)

- Das Voronoi-Diagramm besteht aus allen Zellen, deren d-Wert ein lokales Maximum bilden.
- Voronoi-Diagramm kann zu Graph abstrahiert werden. Nehme Voronoi-Ecken als Knoten und Verbindungen als Kanten. Führe evtl. Zwischenknoten ein.



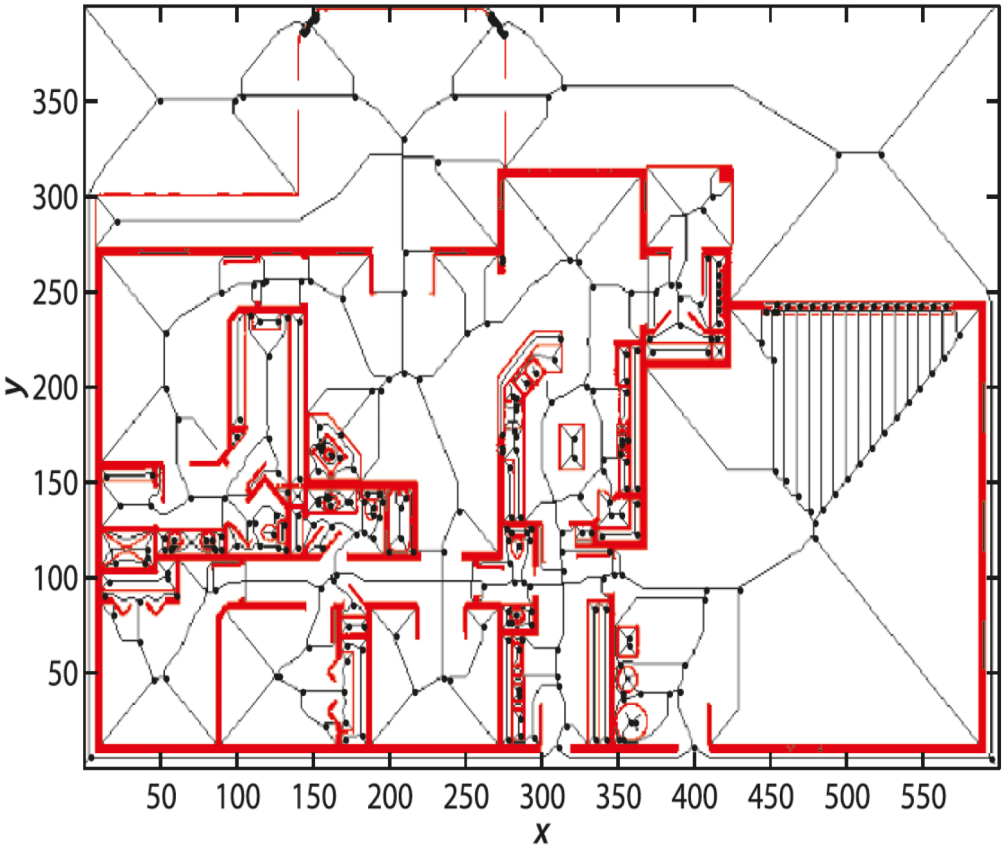
[Corke, 2011]

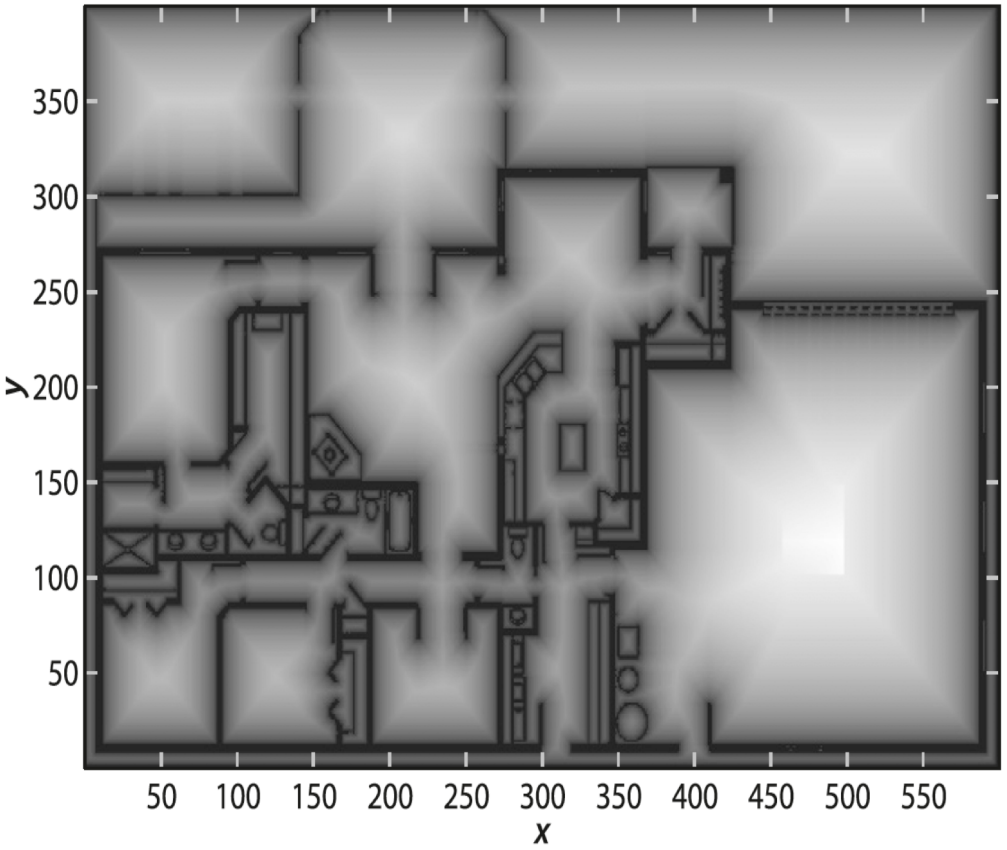
Gitter mit Entfernungswerten zu den nächstgelegenen Hindernissen



Voronoi-Diagramm. Voronoi-Ecken sind blau dargestellt. Bei Überführung in ein Graph können noch Zwischenknoten eingebaut werden.

Wohnungsszenario und Voronoi-Diagramm

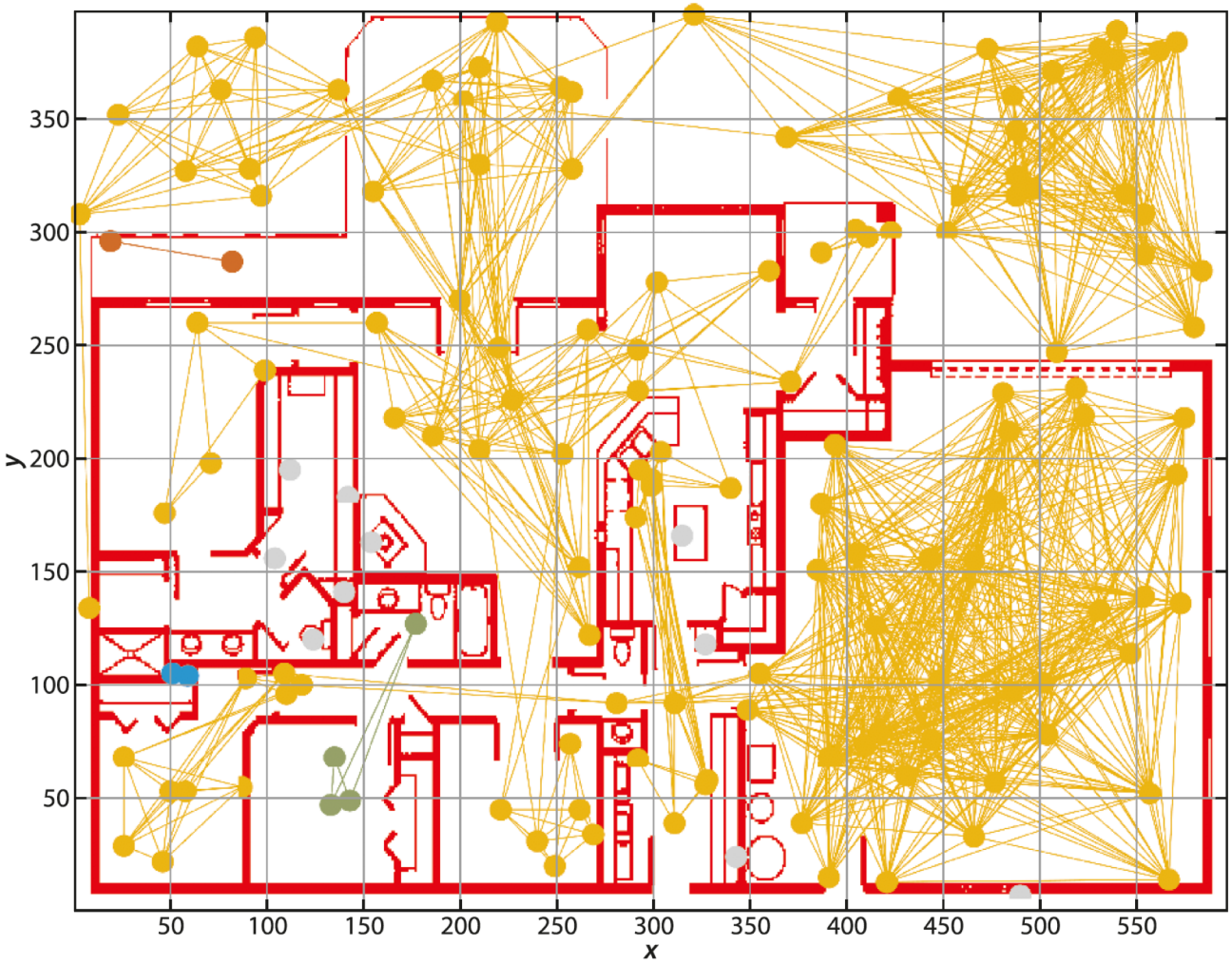




[Corke, 2017]

Probabilistische Wegekarten

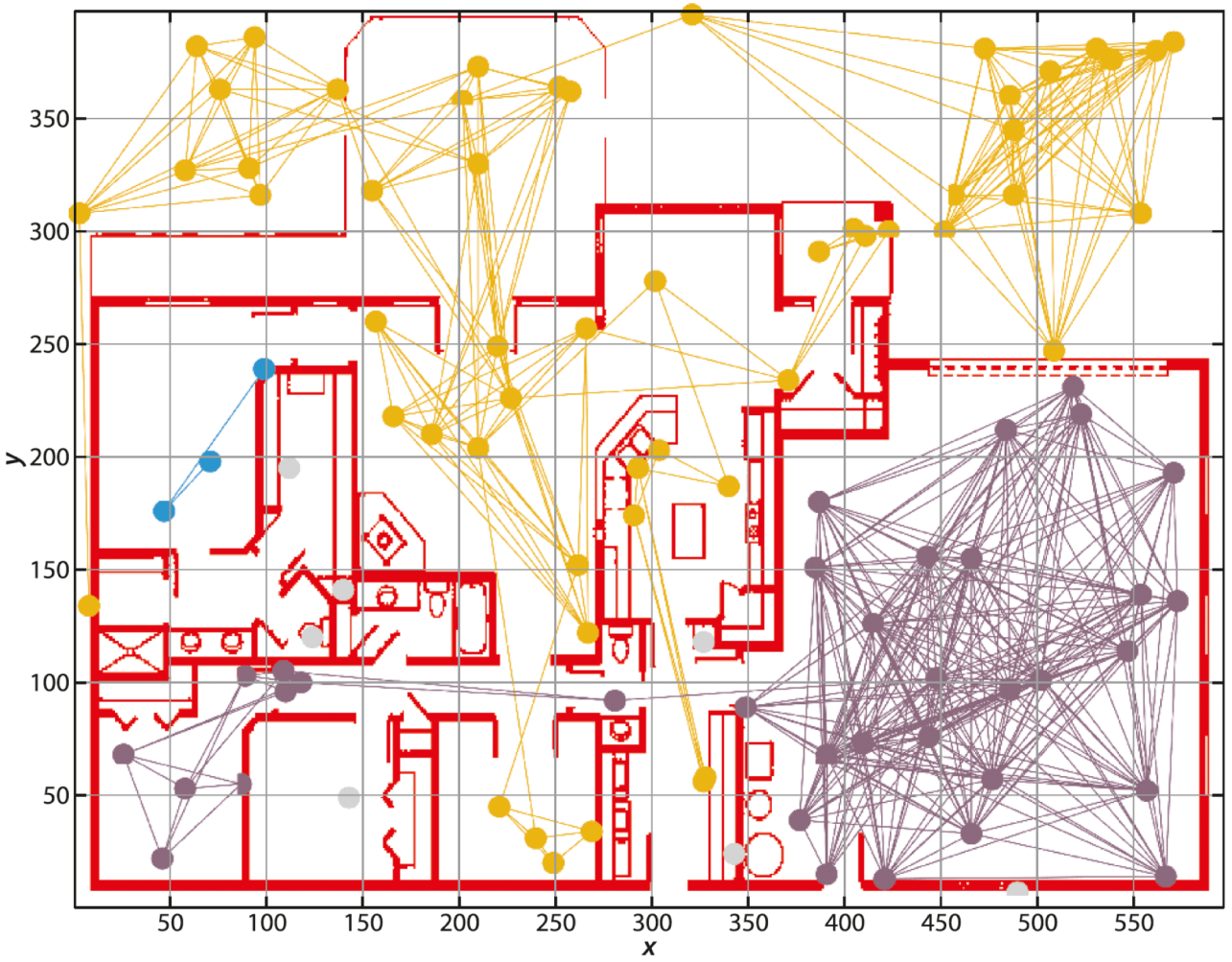
- Beim PRM-Verfahren (Probabilistic Roadmap Method) werden zufällig Punkte auf den Freiflächen generiert.
- Punkte werden mit allen sichtbaren Nachbar-knoten, die innerhalb einer bestimmten Entfernung liegen, mit einer Kante verbunden.
- In den PRM-Karten lassen sich mit den üblichen Graphen-algorithmen Wege finden.



PRM-Karte mit 150 Punkten aus [Corke, 2017]

Bemerkungen

- Das PRM-Verfahren führt im Gegensatz zu Voronoi-Diagramm bei großen Freiflächen zu einer wesentlich besseren Abdeckung.
- Graph deckt bei zu wenig Knoten möglicherweise befahrbaren Bereich nicht hinreichend ab
- Das Verfahren kann dadurch unvollständig werden oder suboptimale Wege generieren.



PRM-Karte mit 100 Punkten aus [Corke, 2017].
Es gibt mehrere isolierte Bereiche.