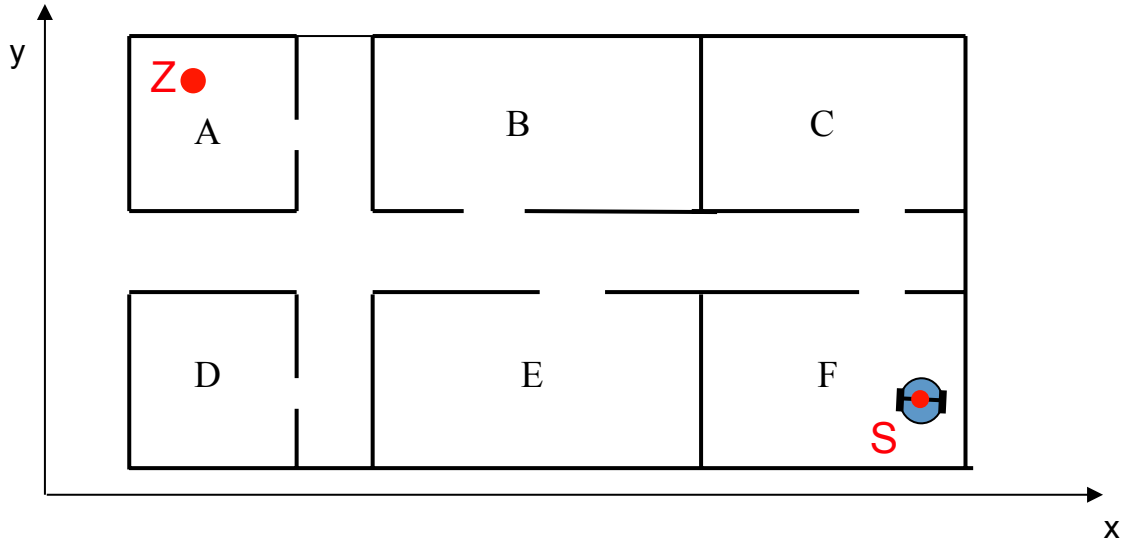
Grundlagen der Pfadplanung

- Überblick
- Kürzeste Wege in Graphen:
  A*, Dijkstra-Verfahren und Breitensuche
- Konfigurations- und Arbeitsraum

Kartenbasierte Navigation

- Suche kürzesten Weg in einer Karte von Start S nach Ziel Z, der dann mittels reaktiver Navigation abgefahren werden kann.
- Kriterien für kürzesten Weg:
  - Weglänge
  - Geschwindigkeit
  - Sicherheit (Abstand von Hindernissen)
  - kinematische Befahrbarkeit



Überblick über Planungsverfahren

- Potentialfeldverfahren:
  Roboter wird als punktförmige Masse behandelt, die unter dem Einfluss eines Potentialfeldes steht.
  Weg wird durch Gradientenabstieg im Potentialfeld gefunden (entspricht dem Weg einer ins Tal rollenden Kugel).

- Wegekartenverfahren:
  Spanne den vom Roboter befahrbaren Bereich durch einen Graphen auf:
  - Sichtbarkeitsgraphen
  - Voronoi-Diagramme
  - Probabilistische Wegekarten (Probabilistic Roadmap Method)
  - RRT-Verfahren (Rapidly-Exploring Random Tree)

- Zellunterteilungsverfahren:
  Zerlege den vom Roboter befahrbaren Bereich in Zellen (ergibt einen Zellnachbarschaftsgraphen)
  - exakte Zellunterteilung (z.B. Trapezzerlegung)
  - approximative Zellunterteilung

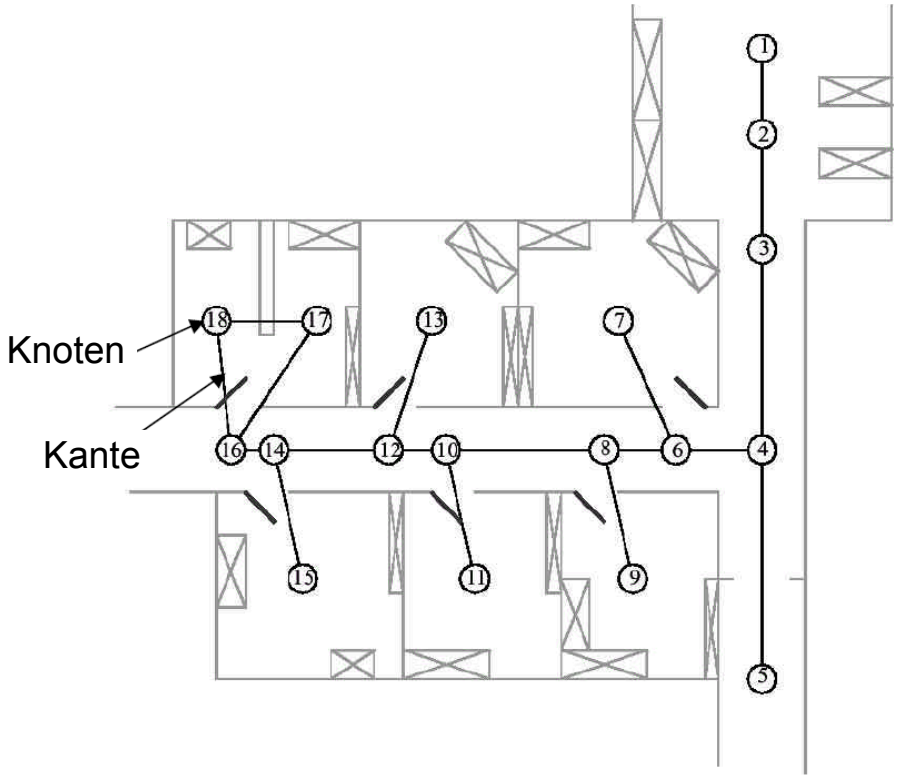
Grundlagen der Pfadplanung

- Überblick
- Kürzeste Wege in Graphen:

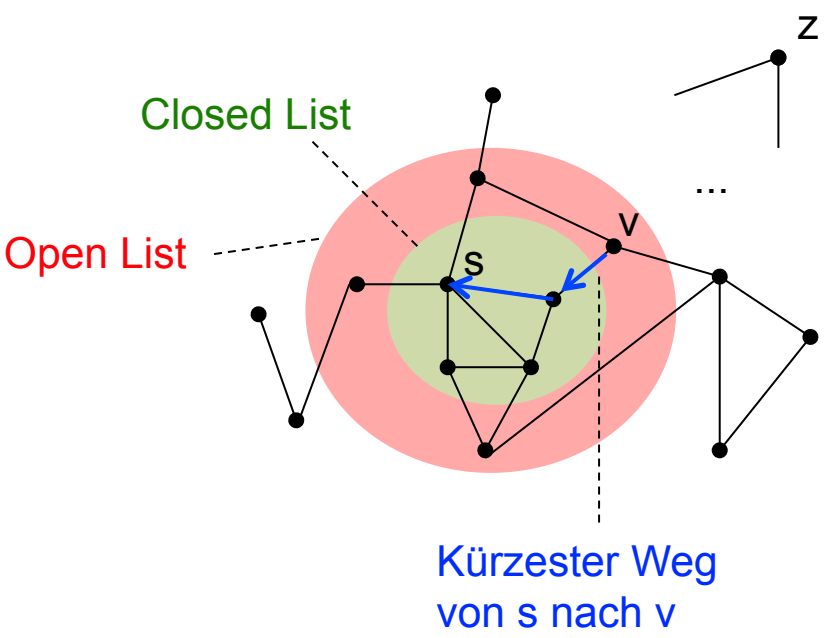
  A*, Dijkstra-Verfahren und Breitensuche
- Konfigurations- und Arbeitsraum

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) ≥ 0 zugeordnet.
- Zusätzlich können die Knoten noch mit geometrischen Informationen wie z.B. (x,y)-Koordinaten versehen sein.



Kürzeste Wege mit A* (1)

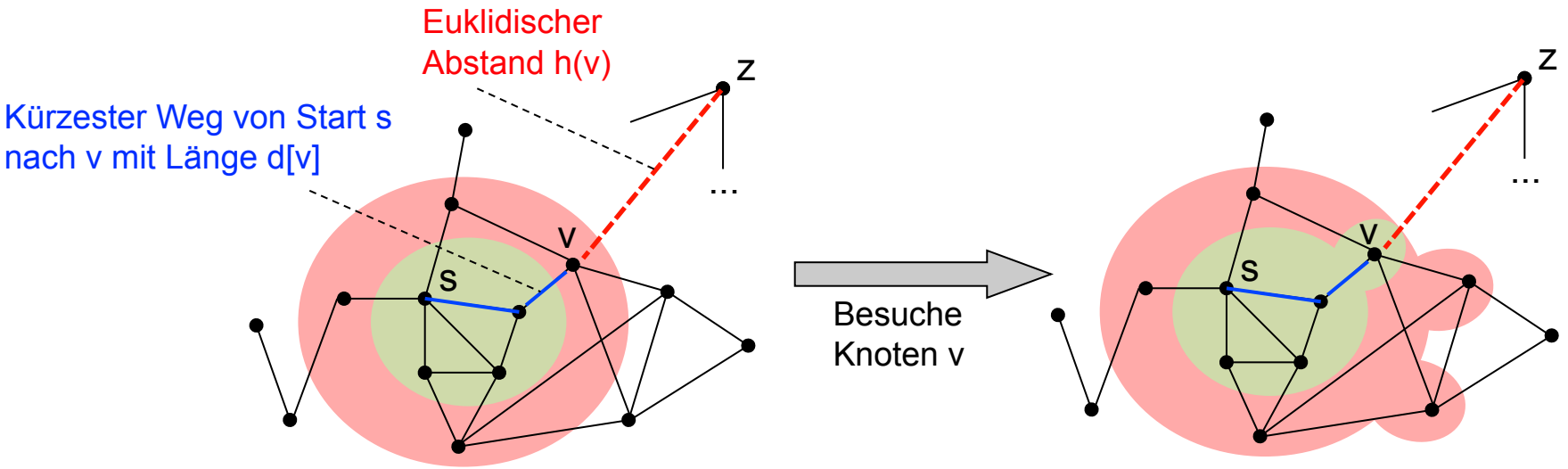


- Suche kürzesten Weg von Startknoten s nach Zielknoten z.
- Closed List:
  Knoten wurden bereits besucht und kürzeste Wege sind bekannt.
- Open List:
  Knoten, die als nächstes besucht werden können.
  Es gibt Werte für kürzeste Wege, die sich aber noch verbessern können.
- Unbekannte Knoten:
  Die restlichen Knoten (in Abb. Knoten mit weissem 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 zum Ziel 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.

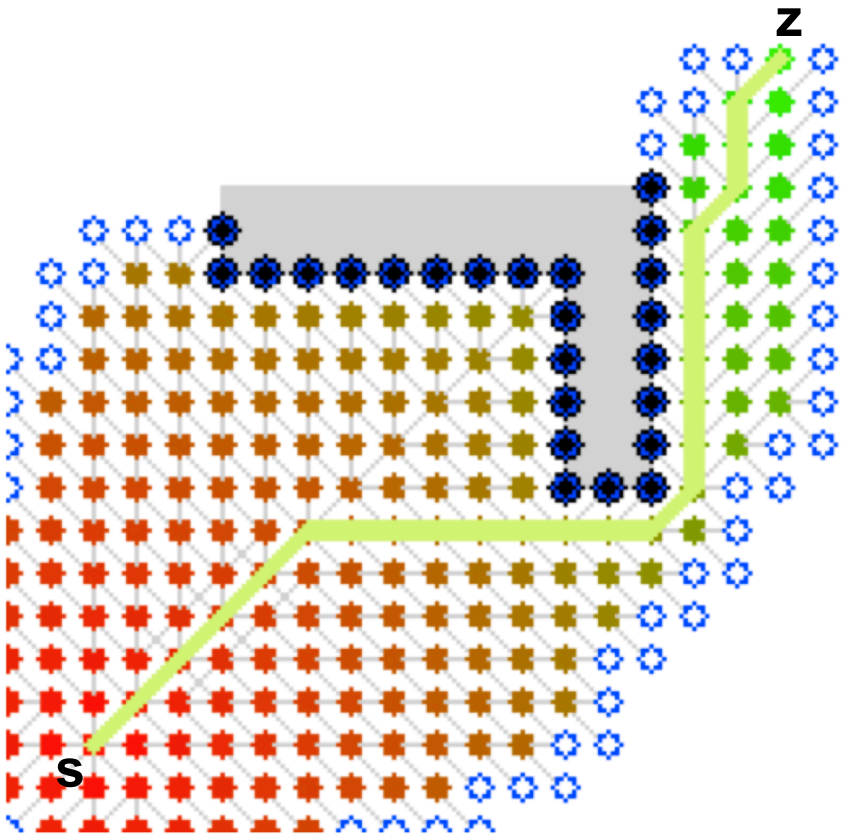
```
void shortestPath (Vertex s, Vertex z, Graph g)
{
        PriorityQueue openList;          // Open List wird als Prioritätsliste realisiert und
                                         // enthält eine Menge von Knoten v mit Prioritätswert d[v] + h(v).
        openList.insert(s, h(s));        // Fügt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

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

Eigenschaften von A* (1)

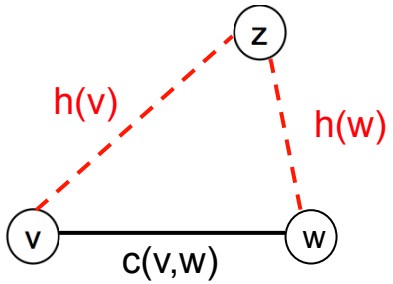
- A* ist vollständig:
  falls ein Weg von S nach Z existiert, wird auch ein Weg gefunden.

- 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 sind monoton.

Heuristiken und Spezialfälle

- Die wichtige Heuristik

  h(v) = Euklidischer Abstand von Knoten v zum Ziel z

  ist zulässig und monoton.

- Die Heuristik

  h(v) = 0 für alle Knoten v

  ist ebenfalls zulässig und monoton. Man erhält damit den Algorithmus von Dijkstra, der kürzeste Wege berechnet ohne Zielorientierung.

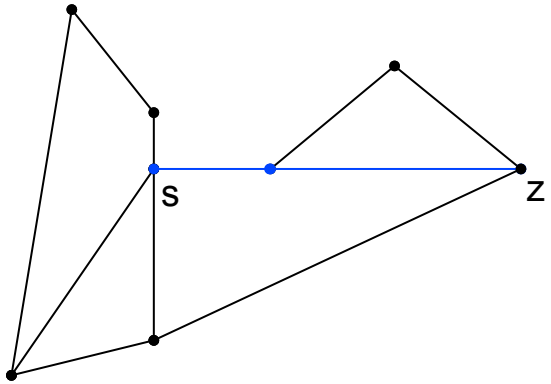
  Das Dijkstra-Verfahren kann, auch wenn der Zielknoten erreicht wurde, fortgesetzt werden. Dann werden alle kürzesten Wege mit Startknoten s gefunden.

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

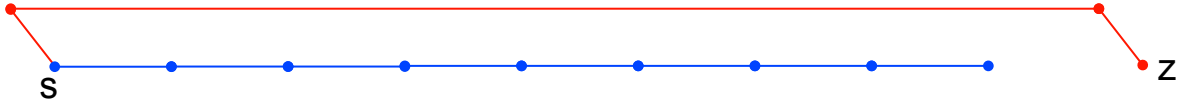
  Statt einer Prioritätsliste kann dann eine einfache Schlange (Queue) verwendet werden.

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.

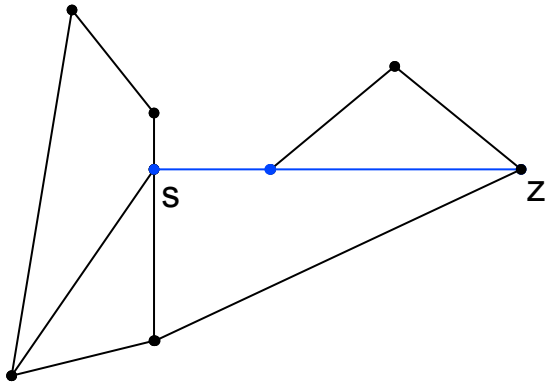


1. A* geht direkt auf das Ziel z zu, läuft aber in eine Sackgasse (blau).
2. A* besucht nun einen Weg, der sich anfangs vom Ziel entfernt und daher nicht früher betrachtet wurde (rot).

Komplexität von A* (2)

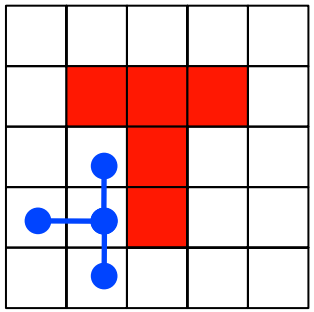
- Laufzeit im schlechtesten Fall:
  - Heuristik bringt keinen Vorteil (siehe Sackgassen-Beispiel).
  - Kein frühzeitiger Abbruch, weil das Ziel gefunden wurde. (Fast) jeder Knoten v wird besucht.
  - Ist der Graph dünn besetzt und wird eine indizierte Prioritätsliste eingesetzt, dann ist $T = O(n \log(n))$,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

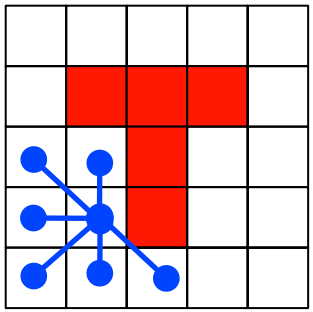


Beispiel: A* in einem Belegtheitsgitter (1)

- Ein Belegtheitsgitter kann als (dünn besetzter) 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.



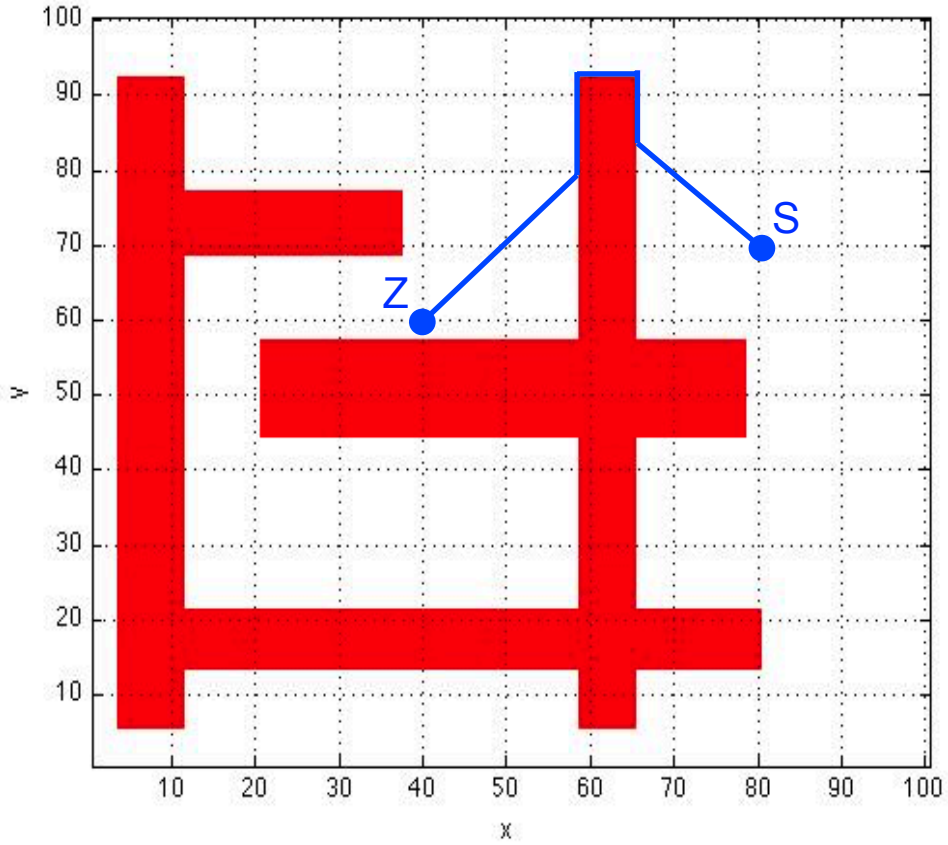
4-Nachbarschaft



8-Nachbarschaft

Beispiel: A* in einem Belegtheitsgitter (2)

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



aus [Corke 2011]

Grundlagen der Pfadplanung

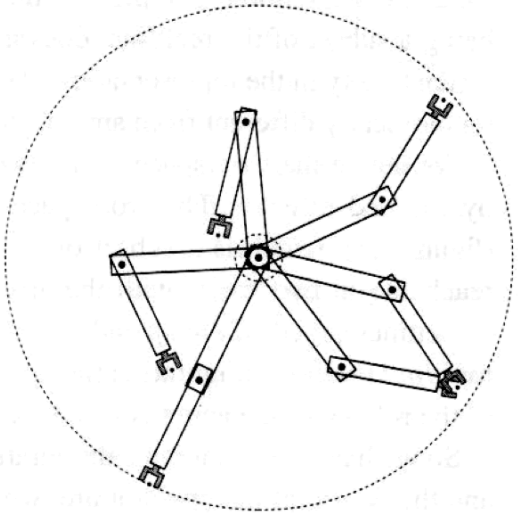
- Überblick
- Kürzeste Wege in Graphen:

  A*, Dijkstra-Verfahren und Breitensuche
- Konfigurations- und Arbeitsraum

Arbeitsraum

Arbeitsraum (workspace) $\mathcal{W}$:

- Raum, in dem sich der Roboter bewegen kann.
- Bei uns: $\mathcal{W} \subseteq R^2$.
  Auch möglich: $\mathcal{W} \subseteq R^3$.



Arbeitsraum für einen zweigliedrigen Greifarm.

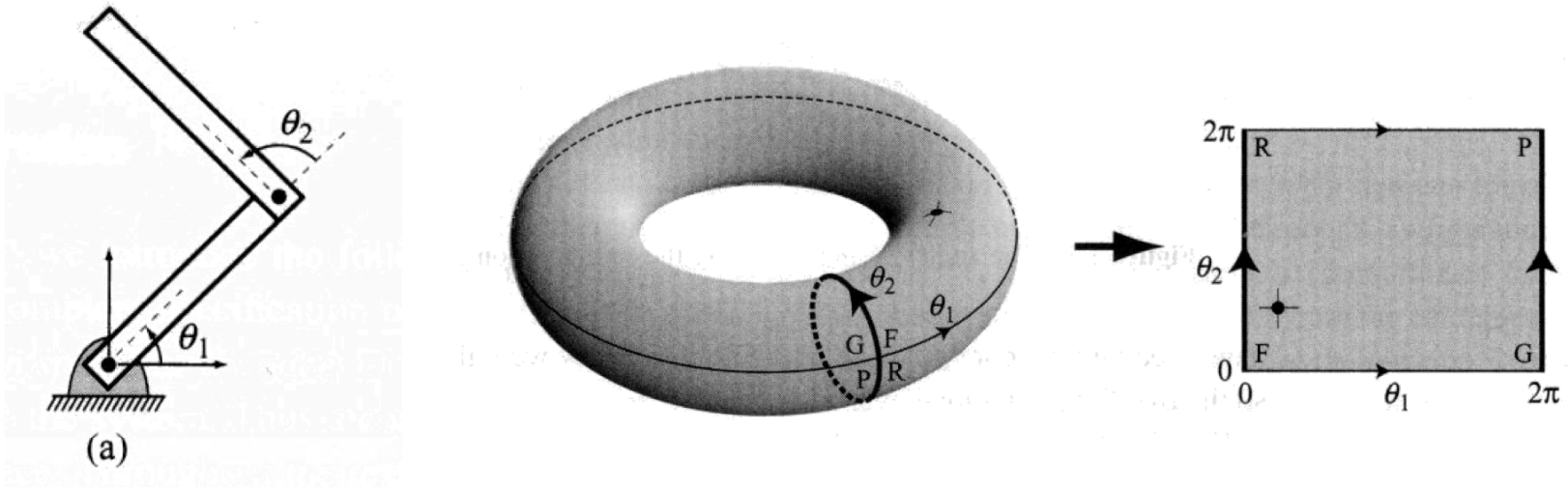
Der Arbeitsraum ist ein Ring (d.h. Kreis mit ausgeschnittenem Kreis im Mittelpunkt)

Aus [Choset 2004].

Konfigurationsraum

Konfigurationsraum (configraution space) *C:*

- Die Konfigration eines Roboters ist eine komplette Beschreibung der Lage seiner für die Bewegung relevanten Teile, so dass die Lage des Roboters im Arbeitsraum eindeutig festgelegt ist.



Torusförmiger Konfigurationsraum für einen zweigliedrigen Greifarm.

Man beachte, dass in der rechten Darstellung die Kanten FR und GP bzw. FG und RP verbunden sind. Aus [Choset 2004].

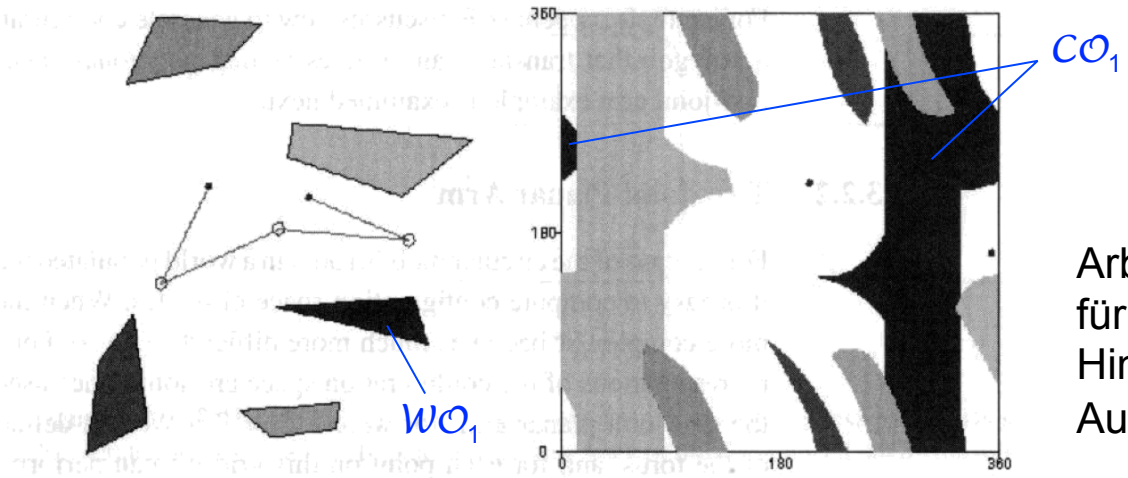
Hindernisse

Hindernisse im Arbeitsraum:

- Arbeitsraum enthält Hindernisse (obstacles).
- i-tes Hindernis: $\mathcal{WO}_i$
- Freier Arbeitsraum: $\mathcal{W}_{free} = \mathcal{W} \setminus \bigcup_i \mathcal{WO}_i$

Hindernisse im Konfigurationsraum:

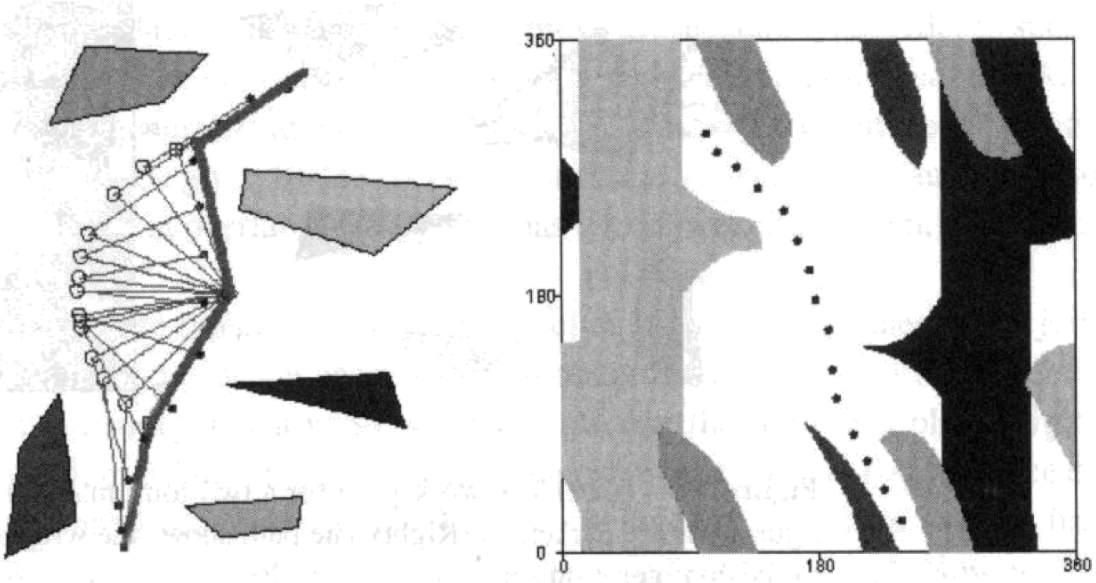
- Für c∈$\mathcal{C}$ sei R(c) der vom Roboter im Arbeitsraum eingenommene Raum.
- i-tes Hindernis im Konfigurationsraum: $\mathcal{CO}_i = \{c \in \mathcal{C} \mid R(c) \cap \mathcal{WO}_i \neq \emptyset\}$
- Freier Konfigurationsraum: $\mathcal{C}_{free} = \mathcal{C} \setminus \bigcup_i \mathcal{CO}_i$



Arbeits- und Konfigurationsraum für einen zweigliedrigen Greifarm. Hindernisse sind grau dargestellt. Aus [Choset 2004].

Wegeplanung im Konfigurationsraum

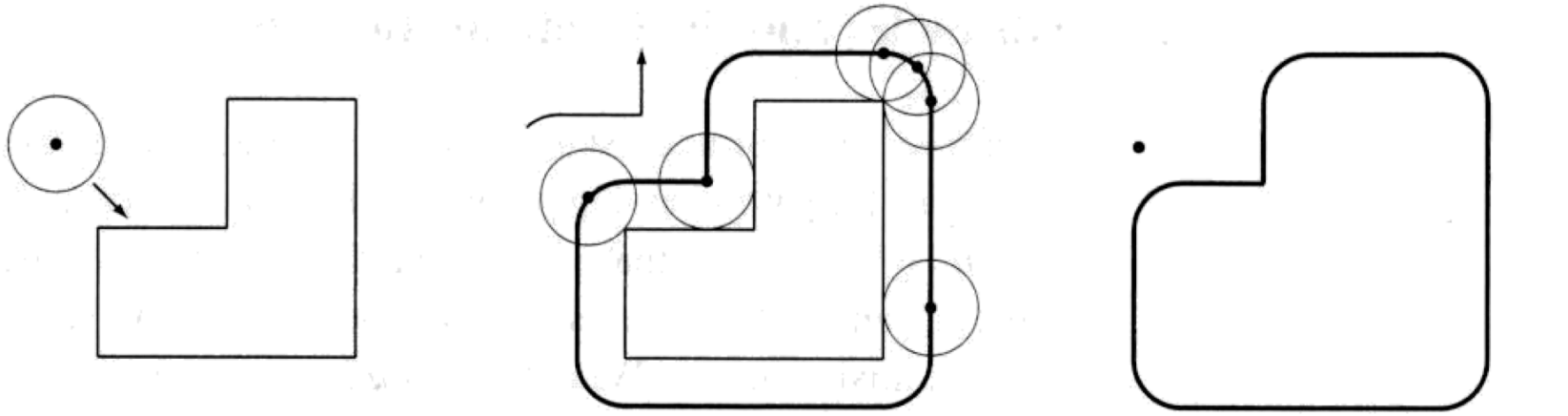
- ein freier Weg von Start s∈$\mathcal{W}$ nach Ziel z∈$\mathcal{W}$ wird definiert als stetige Abbildung p : [0,1] → C_{free} , wobei
  - p(0) = c_s die Startkonfiguration und
  - p(1) = c_z die Zielkonfiguration ist.



Weg eines zweigliedrigen Greifarms im Arbeits- bzw. Konfigurationsraum. Man beachte die Stetigkeit des Weges im Konfigurationsraum (rechts). Aus [Choset 2004].

Kreisförmige mobile Roboter

- In vielen Fällen kann der Roboter als kreisförmig angesehen werden.
- Der Konfigurationsraum ergibt sich dann durch „Aufblasen“ der Hindernisse im Arbeitsraum um den Roboterradius r.

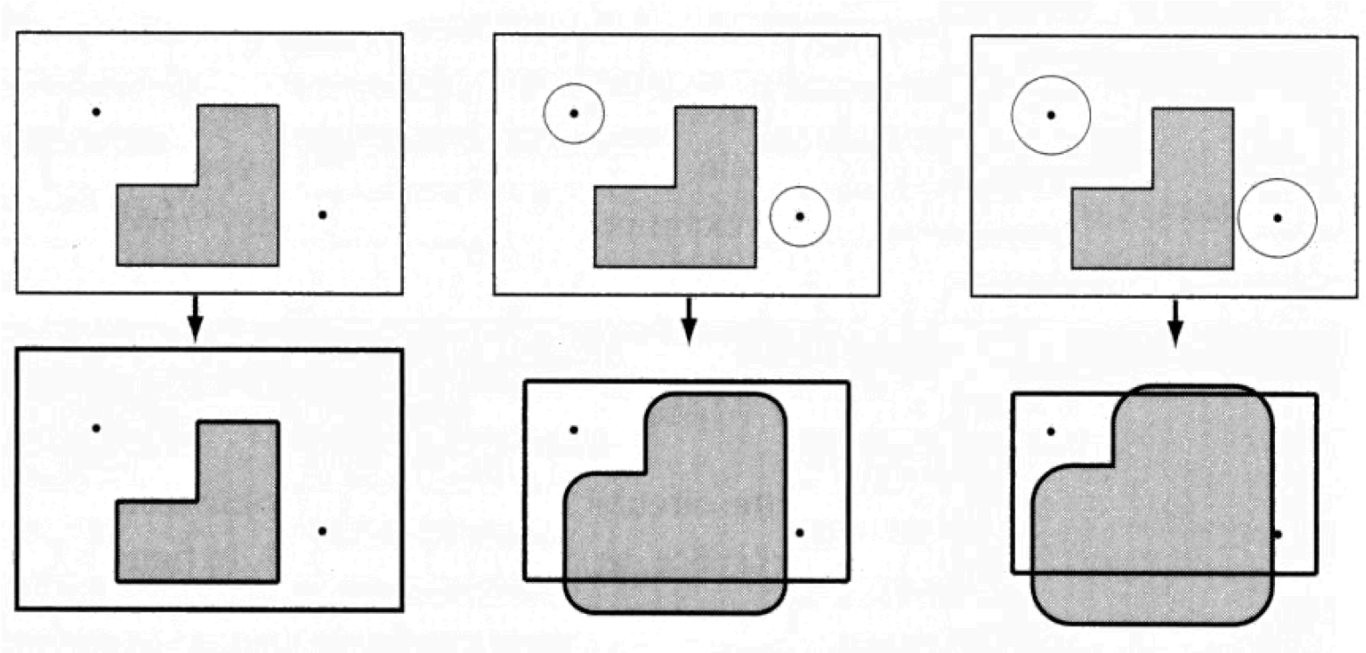


Arbeitsraum

Konfigurationsraum

Punktförmige mobile Roboter

Wir wollen für diese Vorlesung einen punktförmigen Roboter annehmen. Damit fallen Arbeitsraum und Konfigurationsraum zusammen.



Arbeitsraum

Konfigruations-raum

Punktförmiger Roboter

Kreisförmige Roboter