Projekt: Mobile Robotik DLBROESR01_D

Fallstudie

Studiengang: Angewandte Künstliche Intelligenz

Sven Behrens

Matrikelnummer: 42303511

Prof. Dr. Florian Simroth

7. Februar 2026

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AMR Automated Mobile Robot

C-Space Configuration Space

PRM Probabilistic Roadmap

RRT Rapidly-exploring Random Tree

1 Einleitung

Die Intralogistik befindet sich in einem tiefgreifenden Wandel. Angetrieben durch die steigende Kundenerwartungen an Liefergeschwindigkeit setzen immer mehr Unternehmen auf automatisierte mobile Roboter (Automated Mobile Robot (AMR)) für den innerbetrieblichen Warentransport (Fragapane et al., 2021). Eine zentrale Herausforderung beim Einsatz solcher Systeme stellt die Pfadplanung dar: Der Roboter muss in der Lage sein, kollisionsfrei von einem Startpunkt zu einem Zielpunkt zu navigieren und dabei sowohl statische Hindernisse als auch die eigene Geometrie zu berücksichtigen. Vor diesem Hintergrund wurde im Rahmen des Moduls „Mobile Robotik“ an der IU Internationalen Hochschule ein Softwareprototyp entwickelt, der verschiedene etablierte Pfadplanungsalgorithmen implementiert und vergleichend evaluiert.

Das primäre Projektziel bestand in der Implementierung eines Softwaresystems, das für einen omnidirektionalen Roboter in einer polygonbasierten Lagerumgebung kollisionsfreie Pfade berechnet. Die zentrale Forschungsfrage konzentrierte sich darauf, wie verschiedene Pfadplanungsalgorithmen hinsichtlich Pfadqualität, Rechenzeit und Robustheit in unterschiedlich komplexen Umgebungen performieren. Besondere Aufmerksamkeit galt dabei der korrekten Konstruktion des Konfigurationsraums (Configuration Space (C-Space)), der die Robotergeometrie in die Hindernisdarstellung integriert.

Die methodische Vorgehensweise gliederte sich in mehrere aufeinander aufbauende Phasen. Zunächst wurde eine modulare Softwarearchitektur in Python entwickelt, die eine klare Trennung zwischen Umgebungsmodellierung, Robotergeometrie, Konfigurationsraumberechnung und Pfadplanung vorsieht. Anschließend wurden fünf verschiedene Algorithmen implementiert und verglichen: Die graphbasierten Verfahren A*, Dijkstra und Best-First-Search sowie die samplingbasierten Methoden Rapidly-exploring Random Tree (RRT) und Probabilistic Roadmap (PRM). Die Evaluation erfolgte auf drei Testumgebungen mit steigender Komplexität unter Verwendung von drei unterschiedlichen Robotergeometrien.

Der gewählte Ansatz zeichnet sich durch seine Erweiterbarkeit aus. Neben dem zweidimensionalen C-Space für omnidirektionale Roboter wurde zusätzlich ein dreidimensionaler Konfigurationsraum implementiert, der die Orientierung des Roboters als dritte Dimension berücksichtigt. Dies ermöglicht die Pfadplanung für nicht-holonome Roboter und demonstriert die Skalierbarkeit des entwickelten Systems. Durch die systematische Evaluation verschiedener Algorithmus-Roboter-Umgebungs-Kombinationen wurden quantitative Erkenntnisse gewonnen, die als Entscheidungsgrundlage für den praktischen Einsatz dienen können.

Die vorliegende Fallstudie gliedert sich wie folgt: Nach der Beschreibung der Projektumgebung wird der iterative Entwicklungsansatz erläutert. Die Hauptabschnitte behandeln die 2D- und 3D-Pfadplanung mit ihren jeweiligen Ergebnissen. Abschließend werden die Projektergebnisse kritisch reflektiert und Verbesserungspotenziale aufgezeigt.

2 Hauptteil

2.1 Projektumgebung

Zu Beginn des Projekts wurde ein GitHub-Repository[1] angelegt, um eine nachvollziehbare Versionsverwaltung zu gewährleisten. Anschließend wurde eine virtuelle Python-Umgebung mit `venv` eingerichtet, in der alle projektspezifischen Abhängigkeiten isoliert installiert wurden. Als Implementierungssprache wurde Python gewählt, da der Autor hier über die meiste Erfahrung verfügt.

2.2 Zwei-Phasen-Analyseansatz

Nach dem Einrichten der Projektstruktur wurde die Entwicklung in zwei aufeinander aufbauende Phasen gegliedert. Diese Vorgehensweise ermöglichte es, zunächst die grundlegenden Konzepte der Pfadplanung im einfacheren Fall zu validieren, bevor die Komplexität erhöht wurde.

In der ersten Phase wurde die Pfadplanung für omnidirektionale Roboter implementiert. Bei diesem Robotertyp kann sich der Roboter in jede Richtung bewegen, ohne seine Orientierung ändern zu müssen. Der resultierende Konfigurationsraum ist zweidimensional und umfasst lediglich die Position (x, y) des Roboters.

Die zweite Phase erweiterte das System um die Berücksichtigung der Roboterorientierung. Für nichtholonome Roboter, die sich nicht seitwärts bewegen können, ist die Orientierung θ eine zusätzliche Freiheitsdimension. Der Konfigurationsraum wird dadurch dreidimensional (x, y, θ), was die Komplexität der Kollisionsprüfung und Pfadsuche erheblich steigert. Durch den modularen Aufbau der ersten Phase konnten wesentliche Komponenten wiederverwendet und gezielt erweitert werden.

2.3 2D-Pfadplanung

2.3.1 Umgebungsmodellierung

Die Lagerumgebung wird durch eine rechteckige Grundfläche mit polygonalen Hindernissen modelliert. Jedes Hindernis ist als Liste von Eckpunkten definiert, die ein geschlossenes Polygon bilden. Diese Repräsentation ermöglicht eine flexible Darstellung beliebiger konvexer und konkaver Hindernisformen.

Für die Evaluation wurden drei Testumgebungen mit steigender Komplexität erstellt. Die *easy*-Umgebung umfasst eine Fläche von 20×15 Metern mit zwei rechteckigen Hindernissen, die einen einfachen Umweg erfordern. Die *medium*-Umgebung erweitert die Fläche auf 25×20 Meter und enthält sechs Hindernisse, die korridorartige Strukturen bilden. Die *hard*-Umgebung stellt mit 30×25 Metern und 14 Hindernissen

[1] https://github.com/svenb23/mobile-robotik-pfadplanung

eine labyrinthartige Struktur dar, die deutlich komplexere Pfade erfordert (siehe Fig. 1 im Anhang).

2.3.2 Robotergeometrie

Der Roboter wird als Polygon modelliert, dessen Eckpunkte relativ zum Ursprung definiert sind. Diese Darstellung ermöglicht eine einheitliche Behandlung beliebiger Roboterformen bei der Kollisionsprüfung. Die Methode `at(x, y, theta)` transformiert das Roboterpolygon an eine beliebige Position mit optionaler Rotation.

Für die Evaluation wurden drei unterschiedliche Geometrien implementiert. Der kreisförmige Roboter mit einem Radius von 0,5 Metern wird als 16-Eck approximiert und repräsentiert einen omnidirektionalen Roboter ohne bevorzugte Ausrichtung. Der rechteckige Roboter mit den Abmessungen $0{,}8 \times 0{,}5$ Meter entspricht einem typischen Transportroboter. Der dreieckige Roboter mit einer Basis von 0,8 Metern und einer Höhe von 0,6 Metern zeigt mit seiner Spitze nach vorne, kann sich jedoch als omnidirektionaler Roboter in alle Richtungen bewegen. Diese asymmetrische Form verdeutlicht den Einfluss der Geometrie auf den Konfigurationsraum (siehe Fig. 2 im Anhang).

2.3.3 Konfigurationsraumberechnung

Der Konfigurationsraum (C-Space) transformiert das Pfadplanungsproblem von einem ausgedehnten Roboter zu einem Punktroboter. Dabei werden die Hindernisse um die Robotergeometrie erweitert, sodass eine Kollisionsprüfung nur noch für den Referenzpunkt des Roboters erforderlich ist.

Die Berechnung erfolgt durch Diskretisierung des Arbeitsraums in ein gleichmäßiges Gitter mit konfigurierbarer Auflösung (für die 2D-Evaluation: 0,3 Meter). Für jede Gitterzelle wird das Roboterpolygon an der entsprechenden Position platziert und mittels der Shapely-Bibliothek (Gillies et al., 2024) auf Überschneidungen mit den Hindernissen geprüft. Zusätzlich wird sichergestellt, dass der Roboter vollständig innerhalb der Umgebungsgrenzen liegt. Das Ergebnis ist ein boolesches 2D-Array, in dem besetzte Zellen als `True` und freie Zellen als `False` markiert sind.

Durch diese Vorverarbeitung reduziert sich die Pfadplanung auf eine Graphsuche im diskretisierten Gitter. Die gewählte Auflösung stellt einen Kompromiss zwischen Genauigkeit und Rechenaufwand dar (siehe Fig. 3 im Anhang).

2.3.4 Pfadplanungsalgorithmen

Für die Pfadsuche wurden fünf Algorithmen aus zwei Kategorien implementiert: graphbasierte und samplingbasierte Verfahren.

Die graphbasierten Algorithmen arbeiten auf dem diskretisierten Konfigurationsraum und nutzen die

pathfinding-Bibliothek. A* bewertet jeden Knoten nach der Summe $f(n) = g(n) + h(n)$, wobei $g(n)$ die tatsächlichen Kosten vom Start und $h(n)$ eine heuristische Schätzung zum Ziel darstellt. Der Algorithmus expandiert stets den Knoten mit dem niedrigsten f-Wert und findet garantiert den kürzesten Pfad bei zulässiger Heuristik (Lynch & Park, 2017, S. 312–314). Dijkstra ist ein Spezialfall von A* mit $h(n) = 0$ und expandiert Knoten ausschließlich nach ihren bisherigen Kosten $g(n)$ (Lynch & Park, 2017, S. 314). Dies liefert ebenfalls optimale Pfade, exploriert jedoch mehr Knoten, da keine Richtungsinformation zum Ziel genutzt wird. Best-First-Search verwendet ausschließlich die Heuristik $h(n)$ und ignoriert die bisherigen Kosten. Dadurch findet der Algorithmus schnell einen Pfad zum Ziel, dieser ist jedoch nicht zwangsläufig optimal. Alle drei Algorithmen erlauben diagonale Bewegungen im Gitter und sind vollständig, das heißt sie finden garantiert eine Lösung, sofern eine existiert.

Die samplingbasierten Algorithmen wurden eigenständig implementiert und arbeiten im kontinuierlichen Raum. RRT baut einen Baum ausgehend vom Startpunkt auf. In jeder Iteration wird ein zufälliger Punkt im Raum gesampelt, der nächste Knoten im Baum bestimmt und ein neuer Knoten in Richtung des Samples mit einer festen Schrittweite (0,5 Meter) hinzugefügt. Eine Zielgewichtung von 10% sorgt dafür, dass der Baum gezielt zum Ziel wächst (Lynch & Park, 2017, S. 324–325). PRM arbeitet in zwei Phasen: Zunächst werden 500 zufällige, kollisionsfreie Punkte im Raum verteilt. Anschließend wird jeder Punkt mit seinen 10 nächsten Nachbarn verbunden, sofern die Verbindung kollisionsfrei ist. Die Pfadsuche erfolgt mittels A* auf diesem vorberechneten Graphen (Lynch & Park, 2017, S. 328–330). Beide Verfahren sind probabilistisch vollständig: Mit zunehmender Anzahl an Samples konvergiert die Wahrscheinlichkeit, eine existierende Lösung zu finden, gegen eins. Sie eignen sich besonders für hochdimensionale Konfigurationsräume (siehe Fig. 4 im Anhang).

2.3.5 Evaluation

Die Evaluation erfolgte systematisch über alle Kombinationen aus drei Testumgebungen, drei Robotergeometrien und fünf Algorithmen. Als Metriken wurden die Pfadlänge, die Anzahl der Wegpunkte und die Rechenzeit erfasst.

Die graphbasierten Algorithmen A* und Dijkstra lieferten in allen Testfällen optimale Pfadlängen. Dies bestätigt die theoretische Äquivalenz beider Verfahren bei Verwendung einer zulässigen Heuristik. Best-First-Search erreichte die kürzesten Rechenzeiten, produzierte jedoch um 10–15% längere Pfade, da die Optimierung zugunsten der Geschwindigkeit vernachlässigt wird.

Bei den samplingbasierten Verfahren zeigte RRT eine hohe Zuverlässigkeit und fand in allen Testfällen einen Pfad. Die resultierenden Pfade waren jedoch 15–20% länger als die optimalen Lösungen, was dem explorativen Charakter des Algorithmus entspricht. PRM lieferte in einfachen Umgebungen effiziente Pfade mit wenigen Wegpunkte, scheiterte jedoch in der *hard*-Umgebung bei rechteckiger und dreieckiger

Robotergeometrie. Die zufällige Verteilung der Samples konnte in diesen Fällen keine durchgängige Verbindung durch die engen Korridore herstellen (siehe Fig. 5 im Anhang).

Die Rechenzeiten stiegen erwartungsgemäß mit der Umgebungskomplexität. Während alle Algorithmen in der *easy*-Umgebung unter 0,1 Sekunden benötigten, erreichte RRT in der *hard*-Umgebung Rechenzeiten von bis zu 0,5 Sekunden. Die Robotergeometrie hatte hingegen nur minimalen Einfluss auf die Performance, da die Kollisionsprüfung durch den vorberechneten Konfigurationsraum effizient erfolgt (siehe Table 1 im Anhang).

2.4 3D-Pfadplanung

2.4.1 Umgebungsmodellierung

Für die 3D-Pfadplanung wurde die *hard*-Umgebung aus der 2D-Evaluation wiederverwendet. Diese Wahl ermöglicht einen direkten Vergleich der Ergebnisse und demonstriert die erhöhte Komplexität durch die zusätzliche Orientierungsdimension in einer bereits anspruchsvollen Umgebung.

2.4.2 Robotergeometrie

Als Robotergeometrie wurde ausschließlich ein rechteckiger Roboter mit den Abmessungen $1{,}0 \times 0{,}5$ Meter verwendet. Die 2D-Evaluation zeigte, dass die Robotergeometrie nur minimalen Einfluss auf den Algorithmusvergleich hat, weshalb für die 3D-Pfadplanung auf die Variation der Roboterformen verzichtet wurde.

2.4.3 Konfigurationsraumberechnung

Der dreidimensionale Konfigurationsraum erweitert die 2D-Repräsentation um die Orientierung θ als dritte Dimension. Die Diskretisierung erfolgt mit einer räumlichen Auflösung von 0,5 Metern und 12 Winkelschritten, was einer Winkelauflösung von 30° entspricht.

Für jede Kombination aus Position (x, y) und Orientierung θ wird das rotierte Roboterpolygon auf Kollisionen mit Hindernissen und Umgebungsgrenzen geprüft. Die Berechnung ist rechenintensiver als im 2D-Fall, da für jeden Gitterpunkt zwölf Orientierungen evaluiert werden müssen. Das Ergebnis ist ein dreidimensionales boolesches Array, das die Befahrbarkeit jeder Konfiguration angibt.

2.4.4 Pfadplanungsalgorithmen

Für die 3D-Pfadplanung wurden vier Algorithmen implementiert: A*, Dijkstra, RRT und PRM. Die graphbasierten Verfahren A* und Dijkstra operieren auf dem 3D-Gitter mit einer erweiterten Nachbarschaftsdefinition:

Neben der 8-Nachbarschaft in der xy-Ebene werden für jede Position drei Optionen für die Orientierung berücksichtigt (unverändert, +30°, -30°).

Die samplingbasierten Verfahren RRT und PRM wurden für den dreidimensionalen Raum (x, y, θ) angepasst. Bei der Bewegung zwischen zwei Konfigurationen wird sowohl die räumliche Distanz als auch die Winkeländerung interpoliert und auf Kollisionsfreiheit geprüft.

2.4.5 Evaluation

Die Evaluation wurde mit einem Startpunkt bei $(1, 1, 0)$ und einem Zielpunkt bei $(7, 2, 180)$ durchgeführt. Diese Konfiguration erfordert nicht nur eine räumliche Navigation durch die Hindernisse, sondern auch eine Drehung des Roboters um 180°.

A* und Dijkstra fanden optimale Pfade mit einer Länge von 66,5 Metern bei Rechenzeiten von 0,74 bzw. 0,68 Sekunden. RRT benötigte mit 8,69 Sekunden deutlich mehr Rechenzeit und lieferte einen längeren Pfad von 79,3 Metern. PRM erzielte mit 72,7 Metern und 0,22 Sekunden einen guten Kompromiss zwischen Pfadqualität und Rechenzeit.

Obwohl samplingbasierte Verfahren theoretisch besser mit höheren Dimensionen skalieren (Petrović, 2018, S. 1), zeigt sich dieser Vorteil bei nur drei Dimensionen noch nicht deutlich. Die graphbasierten Verfahren profitieren hier von der noch handhabbaren Gittergröße. In strukturierten Umgebungen mit engen Korridoren haben samplingbasierte Verfahren zudem Schwierigkeiten, da zufällige Samples selten die schmalen Durchgänge treffen (siehe Fig. 6 und Table 2 im Anhang).

3 Fazit

3.1 Zielerreichung und Projektergebnisse

Das primäre Projektziel, ein Softwaresystem zur kollisionsfreien Pfadplanung für mobile Roboter in polygonbasierten Lagerumgebungen zu entwickeln, wurde vollständig erreicht. Die implementierte Lösung ermöglicht die Berechnung von Pfaden sowohl für omnidirektionale Roboter im 2D-Konfigurationsraum als auch für orientierungsabhängige Roboter im 3D-Konfigurationsraum.

Die systematische Evaluation von fünf Algorithmen auf drei Testumgebungen mit unterschiedlichen Robotergeometrien lieferte quantitative Erkenntnisse über die Stärken und Schwächen der verschiedenen Ansätze. A* und Dijkstra erwiesen sich als zuverlässige Verfahren für optimale Pfade, während Best-First-Search bei Zeitkritikalität Vorteile bietet. Die samplingbasierten Verfahren RRT und PRM zeigten ihre Eignung für komplexere Szenarien, offenbarten jedoch Schwächen in Umgebungen mit engen Korridoren.

3.2 Kritische Reflexion

Die gewählte Diskretisierung des Konfigurationsraums stellt einen Kompromiss dar. Eine feinere Auflösung würde präzisere Pfade ermöglichen, erhöht jedoch den Speicherbedarf und die Rechenzeit erheblich. Besonders im 3D-Fall mit zwölf Winkelschritten ist die Auflösung relativ grob, was zu suboptimalen Pfaden führen kann.

Die Evaluation basiert auf einzelnen Durchläufen pro Konfiguration. Da RRT und PRM auf Zufallszahlen basieren, können die Ergebnisse bei wiederholter Ausführung variieren. Eine statistisch robustere Evaluation würde Mehrfachdurchläufe mit Mittelwertbildung erfordern.

Die Testumgebungen sind statisch und repräsentieren idealisierte Lagerhallen. Reale Einsatzszenarien umfassen dynamische Hindernisse, Unsicherheiten in der Lokalisierung und Sensorrauschen, die in dieser Arbeit nicht berücksichtigt wurden.

3.3 Verbesserungspotenziale und Optimierungsansätze

Die Pfadqualität könnte durch Nachbearbeitung verbessert werden. Pfadglättungsalgorithmen würden die kantigen Trajektorien der gitterbasierten Verfahren in fahrbare Kurven umwandeln. Für die samplingbasierten Verfahren existieren Varianten wie RRT* und PRM*, die asymptotisch optimale Pfade garantieren (Karaman & Frazzoli, 2011, S. 1).

Die Rechenzeit ließe sich durch adaptive Auflösung reduzieren. In freien Bereichen genügt eine grobe Diskretisierung, während in der Nähe von Hindernissen eine feinere Auflösung sinnvoll wäre. Hierarchische Ansätze könnten zunächst auf grobem Gitter planen und nur relevante Bereiche verfeinern.

Eine Erweiterung auf höherdimensionale Konfigurationsräume wäre ebenfalls denkbar. Roboterarme mit sechs oder mehr Freiheitsgraden erfordern entsprechend viele Dimensionen. In solchen Szenarien würden die samplingbasierten Verfahren ihre Stärken gegenüber gitterbasierten Methoden deutlicher ausspielen können.

Neben den implementierten Algorithmen existieren weitere Ansätze, die das System ergänzen könnten. Die Potentialfeld-Methode modelliert das Ziel als anziehende und Hindernisse als abstoßende Kräfte, wodurch der Roboter entlang des Gradientenabstiegs zum Ziel navigiert. Trajektorienoptimierungsverfahren wie CHOMP (Ratliff et al., 2009) oder TrajOpt (Schulman et al., 2014) könnten die gefundenen Pfade hinsichtlich Glätte und Energieeffizienz nachoptimieren.

3.4 Ausblick

Für einen produktiven Einsatz des entwickelten Systems wären weitere Entwicklungsschritte erforderlich. Die Integration mit realer Sensorik, insbesondere LIDAR und Kamerasystemen, würde eine dynamische Aktualisierung der Umgebungskarte ermöglichen. Dadurch könnte der Roboter auf unvorhergesehene Hindernisse reagieren und seine Pfade in Echtzeit anpassen.

Ein weiterer wichtiger Aspekt ist die Koordination mehrerer Roboter in derselben Umgebung. In modernen Lagerhäusern operieren häufig Flotten von AMR gleichzeitig, was eine kollisionsfreie Pfadplanung unter Berücksichtigung der anderen Roboter erfordert. Hierfür existieren dezentrale und zentrale Ansätze, die auf den implementierten Einzelroboter-Algorithmen aufbauen könnten.

Schließlich bieten maschinelle Lernverfahren vielversprechende Möglichkeiten zur Beschleunigung der Pfadplanung. Neuronale Netze könnten trainiert werden, um gute Initiallösungen für die Optimierung zu liefern oder die Heuristik von A* zu verbessern. Solche hybriden Ansätze kombinieren die Zuverlässigkeit klassischer Algorithmen mit der Effizienz datengetriebener Methoden.

Projektrepository

Der vollständige Quellcode ist im GitHub-Repository verfügbar: https://github.com/svenb23/mobile-robotik-pfadplanung

Literatur

Fragapane, G., De Koster, R., Sgarbossa, F., & Strandhagen, J. O. (2021). Planning and control of autonomous mobile robots for intralogistics: Literature review and research agenda. *European Journal of Operational Research*, *294*(2), 405. https://doi.org/10.1016/j.ejor.2021.01.019

Gillies, S., et al. (2024). *Shapely* (Version 2.0) [[Python-Bibliothek]]. https://shapely.readthedocs.io/

Karaman, S., & Frazzoli, E. (2011). Sampling-based Algorithms for Optimal Motion Planning. *The International Journal of Robotics Research*, *30*(7), 846–894. https://doi.org/10.1177/0278364911406761

Lynch, K. M., & Park, F. C. (2017). *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press.

Petrović, L. (2018). Motion planning in high-dimensional spaces [Vorab-Onlinepublikation]. *arXiv*. https://arxiv.org/abs/1806.07457

Ratliff, N., Zucker, M., Bagnell, J. A., & Srinivasa, S. (2009). CHOMP: Gradient Optimization Techniques for Efficient Motion Planning. *Robotics: Science and Systems*. https://doi.org/10.1109/ROBOT.2009.5152817

Schulman, J., Duan, Y., Ho, J., Lee, A., Awwal, I., Bradlow, H., Pan, J., Patil, S., Goldberg, K., & Abbeel, P. (2014). Motion Planning with Sequential Convex Optimization and Convex Collision Checking. *The International Journal of Robotics Research*, *33*(9), 1251–1270. https://doi.org/10.1177/0278364914528132

Verzeichnis der Anhänge

Anhang

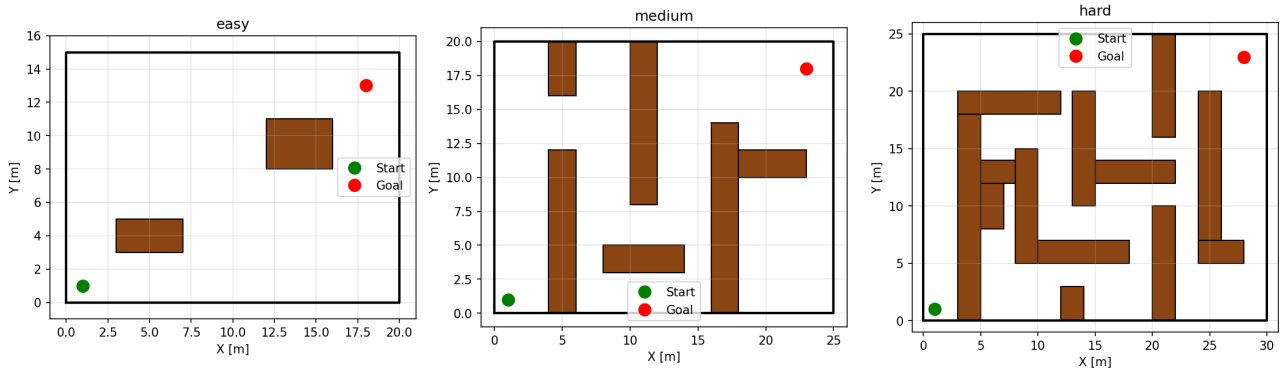

Abbildung 1: Übersicht der drei Testumgebungen: easy (links), medium (Mitte), hard (rechts)

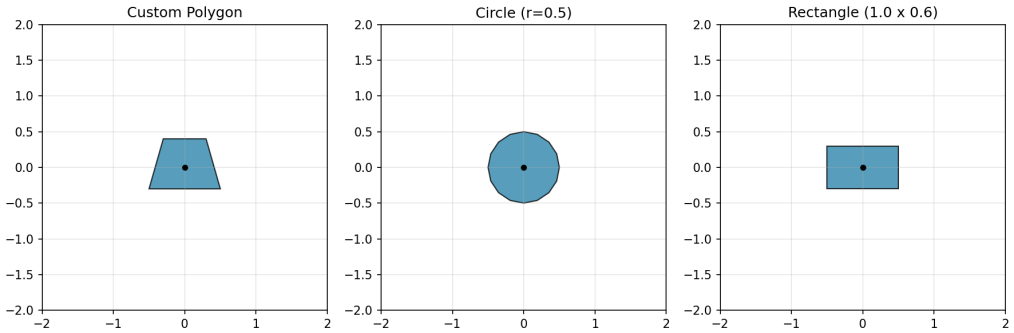

Abbildung 2: Die drei Robotergeometrien: Kreis (links), Rechteck (Mitte), Dreieck (rechts)

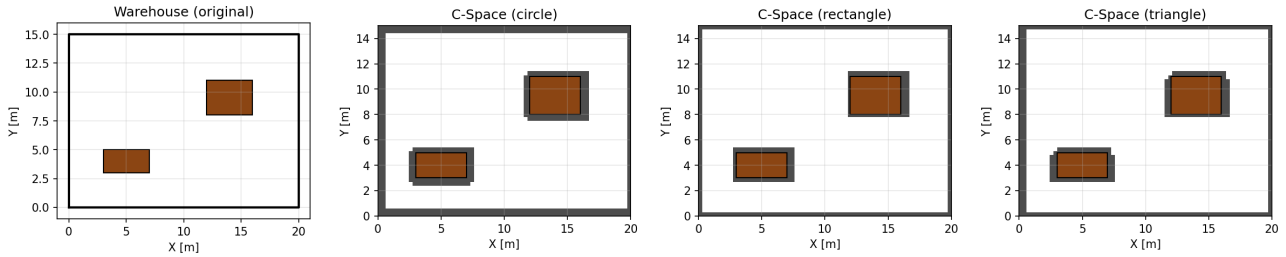

Abbildung 3: Konfigurationsräume für verschiedene Robotergeometrien: Die grauen Bereiche zeigen die erweiterten Hindernisse

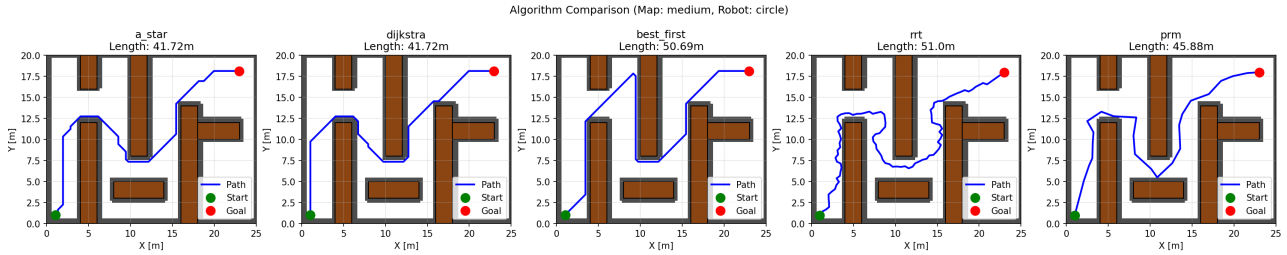

Abbildung 4: Vergleich der Pfadplanungsalgorithmen: Unterschiedliche Strategien führen zu verschiedenen Pfadverläufen

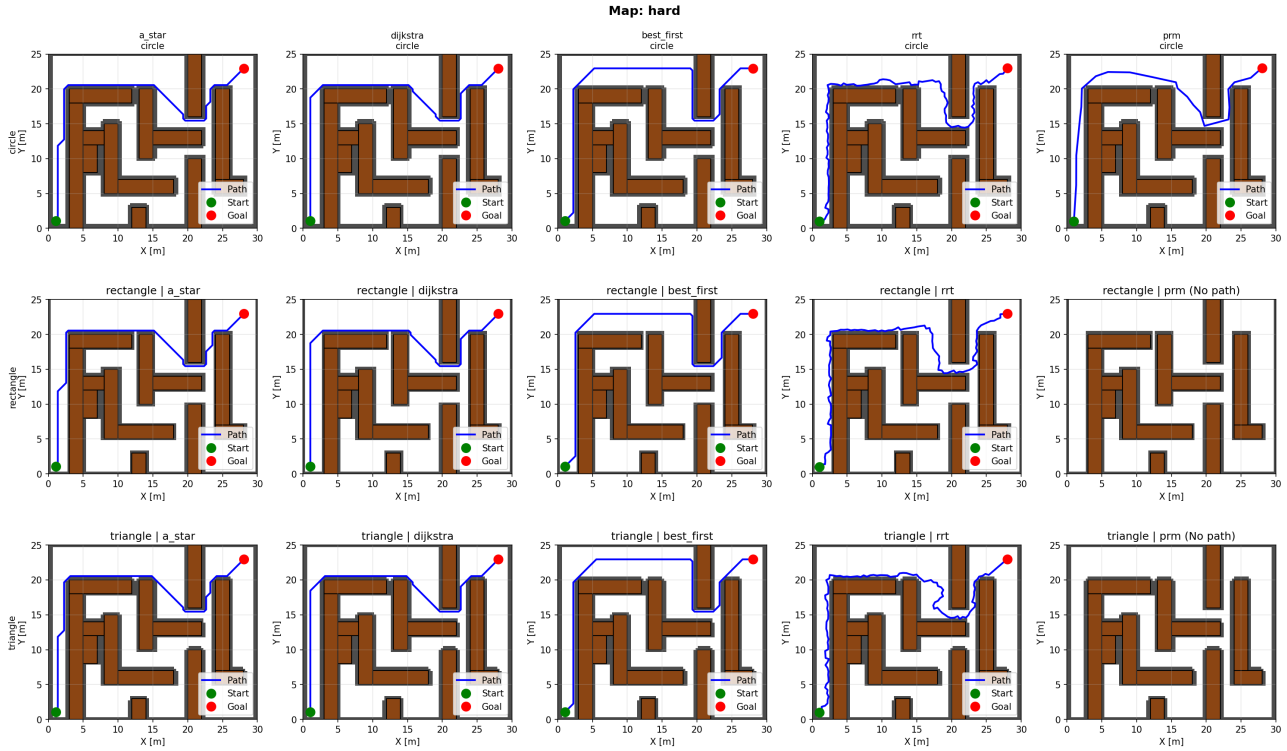

Abbildung 5: Evaluationsergebnisse in der *hard*-Umgebung: PRM scheitert bei rechteckiger und dreieckiger Robotergeometrie (rechte Spalte, mittlere und untere Zeile)

Tabelle 1: Evaluationsergebnisse der 2D-Pfadplanung (kreisförmiger Roboter)

Umgebung	Algorithmus	Pfadlänge [m]	Wegpunkte	Zeit [s]
easy	A*	23,65	67	0,010
easy	Dijkstra	23,65	67	0,017
easy	Best-First	24,60	71	0,004
easy	RRT	28,18	58	0,018
easy	PRM	23,31	21	0,106
medium	A*	41,72	121	0,020
medium	Dijkstra	41,72	121	0,019
medium	Best-First	50,69	148	0,010
medium	RRT	51,00	104	0,125
medium	PRM	45,88	25	0,067
hard	A*	53,12	164	0,022
hard	Dijkstra	53,12	164	0,023
hard	Best-First	58,80	185	0,012
hard	RRT	62,02	126	0,414
hard	PRM	55,04	22	0,045

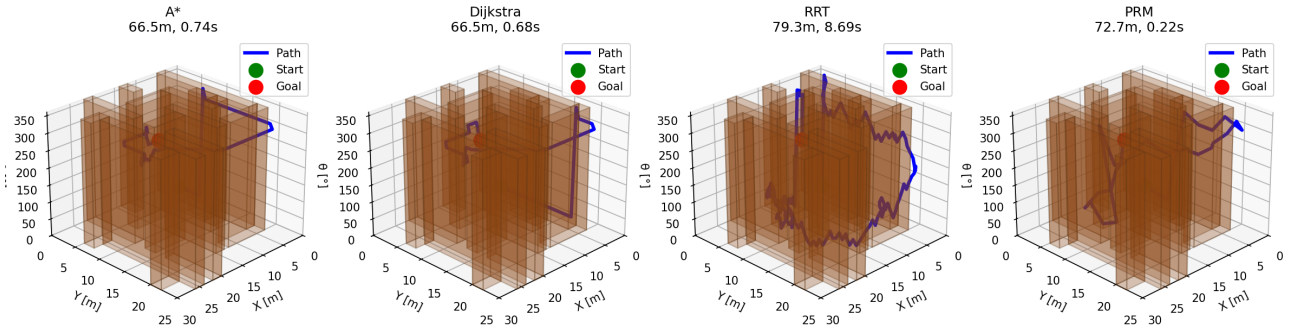

Abbildung 6: 3D-Konfigurationsraum mit Pfaden der vier Algorithmen: Die Hindernisse erstrecken sich als vertikale Wände durch alle Orientierungen

Tabelle 2: Evaluationsergebnisse der 3D-Pfadplanung (rechteckiger Roboter, *hard*-Umgebung)

Algorithmus	Pfadlänge [m]	Wegpunkte	Zeit [s]
A*	66,50	–	0,74
Dijkstra	66,50	–	0,68
RRT	79,30	–	8,69
PRM	72,70	–	0,22

Quellcode

robot.py – Robotergeometrie

```python
import numpy as np
from shapely.geometry import Polygon
from typing import List, Tuple

class Robot:
    """Robot geometry defined by polygon vertices centered at origin."""

    def __init__(self, vertices: List[Tuple[float, float]]):
        self.vertices = vertices
        self.polygon = Polygon(vertices)

    def at(self, x: float, y: float, theta: float = 0) -> Polygon:
        """Return robot polygon at position (x,y) with rotation theta."""
        if theta == 0:
            transformed = [(vx + x, vy + y) for vx, vy in self.vertices]
        else:
            cos_t, sin_t = np.cos(theta), np.sin(theta)
            transformed = [
                (vx * cos_t - vy * sin_t + x, vx * sin_t + vy * cos_t + y)
                for vx, vy in self.vertices
            ]
        return Polygon(transformed)

    @classmethod
    def rectangle(cls, width: float, height: float) -> 'Robot':
        w, h = width / 2, height / 2
        return cls([(-w, -h), (w, -h), (w, h), (-w, h)])

    @classmethod
    def circle(cls, radius: float, n_points: int = 16) -> 'Robot':
        angles = np.linspace(0, 2 * np.pi, n_points, endpoint=False)
        vertices = [(radius * np.cos(a), radius * np.sin(a)) for a in angles
    ]
        return cls(vertices)

    @classmethod
```

```
    def triangle(cls, base: float, height: float) -> 'Robot':
        return cls([
            (height / 2, 0),
            (-height / 2, -base / 2),
            (-height / 2, base / 2),
        ])
```

warehouse.py – Lagerumgebung

```
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon as MplPolygon
from shapely.geometry import Polygon, Point
from typing import List, Tuple

class Warehouse:
    """Warehouse environment with polygon obstacles."""

    def __init__(self, width: float, height: float):
        self.width = width
        self.height = height
        self.obstacles: List[Tuple[Polygon, List, str]] = []

    def add(self, vertices: List[Tuple[float, float]], name: str = "") ->
    None:
        polygon = Polygon(vertices)
        self.obstacles.append((polygon, vertices, name))

    def is_free(self, point: Tuple[float, float]) -> bool:
        x, y = point
        if not (0 < x < self.width and 0 < y < self.height):
            return False
        for polygon, _, _ in self.obstacles:
            if polygon.contains(Point(point)):
                return False
        return True

    def get_polygons(self) -> List[Polygon]:
        return [polygon for polygon, _, _ in self.obstacles]
```

cspace.py – 2D-Konfigurationsraum

```
import numpy as np
from shapely.geometry import Polygon
from typing import Tuple

class CSpace:
    """2D Configuration space (x, y) for omnidirectional robots."""

    def __init__(self, warehouse, robot, resolution: float = 0.5):
        self.warehouse = warehouse
        self.robot = robot
        self.resolution = resolution
        self.cols = int(warehouse.width / resolution)
        self.rows = int(warehouse.height / resolution)
        self.grid = np.zeros((self.rows, self.cols), dtype=bool)
        self._compute()

    def _compute(self):
        obstacles = self.warehouse.get_polygons()
        for row in range(self.rows):
            for col in range(self.cols):
                x = (col + 0.5) * self.resolution
                y = (row + 0.5) * self.resolution
                robot_poly = self.robot.at(x, y)
                if not self._inside_bounds(robot_poly):
                    self.grid[row, col] = True
                    continue
                for obs in obstacles:
                    if robot_poly.intersects(obs):
                        self.grid[row, col] = True
                        break

    def _inside_bounds(self, polygon: Polygon) -> bool:
        minx, miny, maxx, maxy = polygon.bounds
        return minx >= 0 and miny >= 0 and \
               maxx <= self.warehouse.width and maxy <= self.warehouse.
    height

    def is_free(self, x: float, y: float) -> bool:
        col = int(x / self.resolution)
```

```
        row = int(y / self.resolution)
        if 0 <= row < self.rows and 0 <= col < self.cols:
            return not self.grid[row, col]
        return False
```

pathfinder.py – Pfadplanungsalgorithmen (Auszug)

```
import numpy as np
from pathfinding.core.grid import Grid
from pathfinding.core.diagonal_movement import DiagonalMovement
from pathfinding.finder.a_star import AStarFinder
from pathfinding.finder.dijkstra import DijkstraFinder
from pathfinding.finder.best_first import BestFirst
from typing import List, Tuple, Optional
import random

class Pathfinder:
    """2D Pathfinding on CSpace grid with multiple algorithms."""

    def __init__(self, cspace, algorithm: str = "a_star"):
        self.cspace = cspace
        self.algorithm = algorithm
        self._init_finder()

    def _find_rrt(self, start, goal, max_iter=5000, step_size=0.5):
        """RRT (Rapidly-exploring Random Tree) algorithm."""
        tree = {start: None}
        for _ in range(max_iter):
            if random.random() < 0.1:
                sample = goal
            else:
                sample = (random.uniform(0, self.cspace.warehouse.width),
                          random.uniform(0, self.cspace.warehouse.height))
            nearest = min(tree.keys(), key=lambda n: self._dist(n, sample))
            direction = np.array(sample) - np.array(nearest)
            dist = np.linalg.norm(direction)
            if dist < 1e-6:
                continue
            direction = direction / dist
            new_point = tuple(np.array(nearest) + direction * min(step_size,
```

```
    dist))
            if self._line_free(nearest, new_point):
                tree[new_point] = nearest
                if self._dist(new_point, goal) < step_size and \
                   self._line_free(new_point, goal):
                    tree[goal] = new_point
                    return self._reconstruct_path(tree, goal)
        return None

    def _find_prm(self, start, goal, n_samples=500, k_neighbors=10):
        """PRM (Probabilistic Roadmap) algorithm."""
        samples = [start, goal]
        for _ in range(n_samples):
            point = (random.uniform(0, self.cspace.warehouse.width),
                     random.uniform(0, self.cspace.warehouse.height))
            if self.cspace.is_free(point[0], point[1]):
                samples.append(point)
        edges = {s: [] for s in samples}
        for s in samples:
            neighbors = sorted(samples, key=lambda n: self._dist(s, n))[1:
    k_neighbors+1]
            for n in neighbors:
                if self._line_free(s, n):
                    edges[s].append(n)
                    edges[n].append(s)
        return self._astar_roadmap(start, goal, edges)
```

cspace3d.py – 3D-Konfigurationsraum

```
import numpy as np
from shapely.geometry import Polygon
from typing import Tuple

class CSpace3D:
    """3D Configuration space (x, y, theta) for non-holonomic robots."""

    def __init__(self, warehouse, robot, resolution: float = 0.5, n_angles:
    int = 12):
        self.warehouse = warehouse
        self.robot = robot
```

```
        self.resolution = resolution
        self.n_angles = n_angles
        self.angle_resolution = 2 * np.pi / n_angles
        self.cols = int(warehouse.width / resolution)
        self.rows = int(warehouse.height / resolution)
        self.grid = np.zeros((self.rows, self.cols, n_angles), dtype=bool)
        self._compute()

    def _compute(self):
        obstacles = self.warehouse.get_polygons()
        angles = np.linspace(0, 2 * np.pi, self.n_angles, endpoint=False)
        for row in range(self.rows):
            for col in range(self.cols):
                x = (col + 0.5) * self.resolution
                y = (row + 0.5) * self.resolution
                for k, theta in enumerate(angles):
                    robot_poly = self.robot.at(x, y, theta)
                    if not self._inside_bounds(robot_poly):
                        self.grid[row, col, k] = True
                        continue
                    for obs in obstacles:
                        if robot_poly.intersects(obs):
                            self.grid[row, col, k] = True
                            break

    def is_free(self, x: float, y: float, theta: float) -> bool:
        col = int(x / self.resolution)
        row = int(y / self.resolution)
        k = int((theta % (2 * np.pi)) / self.angle_resolution) % self.
    n_angles
        if 0 <= row < self.rows and 0 <= col < self.cols:
            return not self.grid[row, col, k]
        return False
```

pathfinder3d.py – A* für 3D-Konfigurationsraum

```
import numpy as np
from typing import List, Tuple, Optional
import heapq

```

```
class Pathfinder3D:
    """A* pathfinding in 3D configuration space."""

    def __init__(self, cspace3d):
        self.cspace = cspace3d

    def find(self, start, goal) -> Optional[List[Tuple[float, float, float]]]:
        start_cell = self.cspace.to_grid(*start)
        goal_cell = self.cspace.to_grid(*goal)
        if self.cspace.grid[start_cell] or self.cspace.grid[goal_cell]:
            return None
        open_set = [(0, start_cell)]
        came_from = {}
        g_score = {start_cell: 0}
        neighbors_xy = [(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1),(0,0)]
        neighbors_theta = [-1, 0, 1]

        while open_set:
            _, current = heapq.heappop(open_set)
            if current == goal_cell:
                path = [current]
                while current in came_from:
                    current = came_from[current]
                    path.append(current)
                return [self.cspace.to_world(*cell) for cell in path[::-1]]
            row, col, k = current
            for dr, dc in neighbors_xy:
                for dk in neighbors_theta:
                    if dr == 0 and dc == 0 and dk == 0:
                        continue
                    nr, nc = row + dr, col + dc
                    nk = (k + dk) % self.cspace.n_angles
                    if not (0 <= nr < self.cspace.rows and 0 <= nc < self.cspace.cols):
                        continue
                    if self.cspace.grid[nr, nc, nk]:
                        continue
                    neighbor = (nr, nc, nk)
```

```
                    spatial_dist = np.sqrt(dr**2 + dc**2) * self.cspace.
    resolution
                    rotation_cost = abs(dk) * self.cspace.angle_resolution *
     0.5
                    tentative_g = g_score[current] + spatial_dist +
    rotation_cost
                    if neighbor not in g_score or tentative_g < g_score[
    neighbor]:
                        came_from[neighbor] = current
                        g_score[neighbor] = tentative_g
                        h = np.sqrt((nr-goal_cell[0])**2 + (nc-goal_cell[1])
    **2) * \
                            self.cspace.resolution
                        heapq.heappush(open_set, (tentative_g + h, neighbor)
    )
        return None
```

pathfinder3d_dijkstra.py – Dijkstra für 3D-Konfigurationsraum

```
import numpy as np
from typing import List, Tuple, Optional
import heapq

class Pathfinder3D_Dijkstra:
    """Dijkstra pathfinding in 3D configuration space."""

    def __init__(self, cspace3d):
        self.cspace = cspace3d

    def find(self, start: Tuple[float, float, float],
             goal: Tuple[float, float, float]) -> Optional[List[Tuple[float,
    float, float]]]:
        """Find optimal path from start to goal using Dijkstra (no heuristic
    )."""
        start_cell = self.cspace.to_grid(*start)
        goal_cell = self.cspace.to_grid(*goal)

        if self.cspace.grid[start_cell] or self.cspace.grid[goal_cell]:
            return None

```

```
        open_set = [(0, start_cell)]
        came_from = {}
        g_score = {start_cell: 0}

        # 8-connectivity in xy + 3 options for theta
        neighbors_xy = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1)
    , (1, 0), (1, 1), (0, 0)]
        neighbors_theta = [-1, 0, 1]

        while open_set:
            current_g, current = heapq.heappop(open_set)

            if current == goal_cell:
                # Reconstruct path
                path = [current]
                while current in came_from:
                    current = came_from[current]
                    path.append(current)
                return [self.cspace.to_world(*cell) for cell in path[::-1]]

            if current_g > g_score.get(current, float('inf')):
                continue

            row, col, k = current

            for dr, dc in neighbors_xy:
                for dk in neighbors_theta:
                    if dr == 0 and dc == 0 and dk == 0:
                        continue

                    nr = row + dr
                    nc = col + dc
                    nk = (k + dk) % self.cspace.n_angles

                    if not (0 <= nr < self.cspace.rows and 0 <= nc < self.
    cspace.cols):
                        continue
                    if self.cspace.grid[nr, nc, nk]:
                        continue

```

```
                    neighbor = (nr, nc, nk)
                    # Cost = spatial distance + rotation cost
                    spatial_dist = np.sqrt(dr**2 + dc**2) * self.cspace.
    resolution
                    rotation_cost = abs(dk) * self.cspace.angle_resolution *
     0.5
                    tentative_g = g_score[current] + spatial_dist +
    rotation_cost

                    if neighbor not in g_score or tentative_g < g_score[
    neighbor]:
                        came_from[neighbor] = current
                        g_score[neighbor] = tentative_g
                        heapq.heappush(open_set, (tentative_g, neighbor))

        return None
```

pathfinder3d_rrt.py – RRT für 3D-Konfigurationsraum

```
import numpy as np
from typing import List, Tuple, Optional
import random

class Pathfinder3D_RRT:
    """RRT (Rapidly-exploring Random Tree) in 3D configuration space."""

    def __init__(self, cspace3d):
        self.cspace = cspace3d

    def find(self, start: Tuple[float, float, float],
             goal: Tuple[float, float, float],
             max_iter: int = 10000, step_size: float = 0.5,
             angle_step: float = 0.3) -> Optional[List[Tuple[float, float,
    float]]]:
        """Find path using RRT algorithm."""
        tree = {start: None}

        for _ in range(max_iter):
            # Sample random point (10% bias towards goal)
            if random.random() < 0.1:
```

```
                sample = goal
            else:
                sample = (
                    random.uniform(0, self.cspace.warehouse.width),
                    random.uniform(0, self.cspace.warehouse.height),
                    random.uniform(0, 2 * np.pi)
                )

            # Find nearest node in tree
            nearest = min(tree.keys(), key=lambda n: self._dist(n, sample))

            # Steer towards sample
            direction = np.array(sample[:2]) - np.array(nearest[:2])
            dist = np.linalg.norm(direction)
            if dist < 1e-6:
                new_x, new_y = nearest[0], nearest[1]
            else:
                direction = direction / dist
                new_pos = np.array(nearest[:2]) + direction * min(step_size,
 dist)
                new_x, new_y = new_pos[0], new_pos[1]

            # Interpolate angle
            angle_diff = sample[2] - nearest[2]
            angle_diff = (angle_diff + np.pi) % (2 * np.pi) - np.pi
            new_theta = nearest[2] + np.sign(angle_diff) * min(abs(
angle_diff), angle_step)
            new_theta = new_theta % (2 * np.pi)

            new_point = (new_x, new_y, new_theta)

            # Add to tree if path is collision-free
            if self._line_free(nearest, new_point):
                tree[new_point] = nearest

                # Check if goal is reachable
                if self._dist(new_point, goal) < step_size:
                    if self._line_free(new_point, goal):
                        tree[goal] = new_point
                        return self._reconstruct(tree, goal)
```

```
        return None

    def _dist(self, a, b):
        """Distance metric combining spatial and angular distance."""
        spatial = np.sqrt((a[0] - b[0])**2 + (a[1] - b[1])**2)
        angle_diff = abs(a[2] - b[2])
        angle_diff = min(angle_diff, 2 * np.pi - angle_diff)
        return spatial + angle_diff * 0.5

    def _line_free(self, a, b, steps: int = 10):
        """Check if straight line between two configurations is collision-free."""
        for i in range(steps + 1):
            t = i / steps
            x = a[0] + t * (b[0] - a[0])
            y = a[1] + t * (b[1] - a[1])
            angle_diff = b[2] - a[2]
            angle_diff = (angle_diff + np.pi) % (2 * np.pi) - np.pi
            theta = (a[2] + t * angle_diff) % (2 * np.pi)
            if not self.cspace.is_free(x, y, theta):
                return False
        return True

    def _reconstruct(self, tree, goal):
        """Reconstruct path from tree."""
        path = [goal]
        current = goal
        while tree[current] is not None:
            current = tree[current]
            path.append(current)
        return path[::-1]
```

pathfinder3d_prm.py – PRM für 3D-Konfigurationsraum

```
import numpy as np
from typing import List, Tuple, Optional
import random
import heapq

```

```python
class Pathfinder3D_PRM:
    """PRM (Probabilistic Roadmap) in 3D configuration space."""

    def __init__(self, cspace3d):
        self.cspace = cspace3d

    def find(self, start: Tuple[float, float, float],
             goal: Tuple[float, float, float],
             n_samples: int = 1000, k_neighbors: int = 15) -> Optional[List[
    Tuple[float, float, float]]]:
        """Find path using PRM algorithm."""
        # Sample random collision-free configurations
        samples = [start, goal]
        for _ in range(n_samples):
            point = (
                random.uniform(0, self.cspace.warehouse.width),
                random.uniform(0, self.cspace.warehouse.height),
                random.uniform(0, 2 * np.pi)
            )
            if self.cspace.is_free(*point):
                samples.append(point)

        # Build roadmap by connecting k-nearest neighbors
        edges = {s: [] for s in samples}
        for s in samples:
            neighbors = sorted(samples, key=lambda n: self._dist(s, n))[1:
    k_neighbors+1]
            for n in neighbors:
                if self._line_free(s, n):
                    edges[s].append(n)
                    edges[n].append(s)

        # Search roadmap using A*
        return self._astar(start, goal, edges)

    def _astar(self, start, goal, edges):
        """A* search on the roadmap."""
        open_set = [(0, start)]
        came_from = {}
        g_score = {start: 0}
```

```
44
45          while open_set:
46              _, current = heapq.heappop(open_set)
47
48              if current == goal:
49                  path = [current]
50                  while current in came_from:
51                      current = came_from[current]
52                      path.append(current)
53                  return path[::-1]
54
55              for neighbor in edges.get(current, []):
56                  tentative_g = g_score[current] + self._dist(current,
    neighbor)
57                  if neighbor not in g_score or tentative_g < g_score[neighbor
    ]:
58                      came_from[neighbor] = current
59                      g_score[neighbor] = tentative_g
60                      f = tentative_g + self._dist(neighbor, goal)
61                      heapq.heappush(open_set, (f, neighbor))
62
63          return None
64
65      def _dist(self, a, b):
66          """Distance metric combining spatial and angular distance."""
67          spatial = np.sqrt((a[0] - b[0])**2 + (a[1] - b[1])**2)
68          angle_diff = abs(a[2] - b[2])
69          angle_diff = min(angle_diff, 2 * np.pi - angle_diff)
70          return spatial + angle_diff * 0.5
71
72      def _line_free(self, a, b, steps: int = 10):
73          """Check if straight line between two configurations is collision-
    free."""
74          for i in range(steps + 1):
75              t = i / steps
76              x = a[0] + t * (b[0] - a[0])
77              y = a[1] + t * (b[1] - a[1])
78              angle_diff = b[2] - a[2]
79              angle_diff = (angle_diff + np.pi) % (2 * np.pi) - np.pi
80              theta = (a[2] + t * angle_diff) % (2 * np.pi)
```

```
            if not self.cspace.is_free(x, y, theta):
                return False
        return True
```

maps.py – Kartendefinitionen

```
MAPS = {
    "easy": {
        "width": 20,
        "height": 15,
        "obstacles": [
            [(3, 3), (7, 3), (7, 5), (3, 5)],
            [(12, 8), (16, 8), (16, 11), (12, 11)],
        ]
    },

    "medium": {
        "width": 25,
        "height": 20,
        "obstacles": [
            [(4, 0), (6, 0), (6, 12), (4, 12)],
            [(4, 16), (6, 16), (6, 20), (4, 20)],
            [(10, 8), (12, 8), (12, 20), (10, 20)],
            [(16, 0), (18, 0), (18, 14), (16, 14)],
            [(8, 3), (14, 3), (14, 5), (8, 5)],
            [(18, 10), (23, 10), (23, 12), (18, 12)],
        ]
    },

    "hard": {
        "width": 30,
        "height": 25,
        "obstacles": [
            [(3, 0), (5, 0), (5, 18), (3, 18)],
            [(3, 18), (12, 18), (12, 20), (3, 20)],
            [(8, 5), (10, 5), (10, 15), (8, 15)],
            [(10, 5), (18, 5), (18, 7), (10, 7)],
            [(13, 10), (15, 10), (15, 20), (13, 20)],
            [(15, 12), (22, 12), (22, 14), (15, 14)],
            [(20, 0), (22, 0), (22, 10), (20, 10)],
```

```
            [(20, 16), (22, 16), (22, 25), (20, 25)],
            [(24, 5), (28, 5), (28, 7), (24, 7)],
            [(24, 7), (26, 7), (26, 25), (24, 25)],
            [(12, 0), (14, 0), (14, 3), (12, 3)],
            [(5, 8), (7, 8), (7, 12), (5, 12)],
            [(5, 12), (8, 12), (8, 14), (5, 14)],
            [(26, 20), (30, 20), (30, 22), (26, 22)],
        ]
    },
}
```