UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN

CONTROL SYSTEMS ENGINEERING

Path planning algorithms for autonomous navigation of a non-holonomic robot in unstructured environments

Relatore: Prof. BOSCHETTI GIOVANNI

Laureanda: VEZZANI VALERIA

ANNO ACCADEMICO 2022 – 2023

10/10/2023

Ringraziamenti

Questo capitolo vuole essere un piccolo ringraziamento e una dedica a tutti coloro che mi hanno sostenuta e aiutata negli ultimi mesi per la scrittura della tesi. Forse non riuscirò ad essere sintetica.
Prima di tutto, desidero esprimere tutta la mia gratitudine alla mia famiglia: i miei genitori e mio fratello. Sono le persone che, giorno dopo giorno, sono state al mio fianco in modo incondizionato. Hanno saputo ascoltarmi, darmi preziosi consigli e, nei momenti più difficili, sono stati un punto fermo, senza mai arrendersi, affrontando insieme a me ogni sfida. Desidero ringraziarli per il costante sostegno che mi hanno offerto durante questi ultimi 5 anni e per avermi dato l'opportunità di intraprendere il mio percorso universitario, che è stato per me il periodo più straordinario della mia vita.
Voglio ringraziare il mio ragazzo Giovanni. Sei la mia persona, il primo che desidera condividere con me tutte le esperienze, sia quelle positive che negative. Quest'ultimo periodo è stato difficile per me e nonostante tutto ci sei sempre stato e hai saputo sempre tendermi una mano. Ti ringrazio per avermi insegnato la pazienza e la dedizione che ci vogliono per arrivare fino in fondo e per aver sempre creduto in me.
Ringrazio Mariafrancesca che con implacabile costanza riesce sempre ad essere una persona positiva e ottimista e che ha cercato di trasmettermi tutto questo oramai da tanti anni. Ti ringrazio per non aver mai smesso di contagiarmi con il tuo carattere e per essere un'amica splendida.
Ringrazio Carlotta e Agnese, inseparabili amiche di sempre che hanno saputo conciliare leggerezza e serietà nel sapermi aiutare. Vi ringrazio per tutto quello che riusciamo a condividere, vicine e lontane.
Ringrazio il mio gruppo Mango Street: Giovanni, Andrea e Damiano. Siete sempre pieni di una buona dose di follia e umorismo. Grazie per tutte le serate condivise che mi hanno alleggerito questo periodo.
Ringrazio tutti i miei amici di Padova: i miei amici della triennale, della magistrale e i miei

coinquilini. Avete tutti quanti saputo rendere la mia vita lontana da casa sempre piena di piccole avventure e di scherzi indimenticabili. Siete e sarete sempre il capitolo più bello.
Ci tengo tanto a ringrazia le dottoresse che mi hanno seguita negli ultimi mesi: la dott.ssa Porro e la dott.ssa Pinelli. Vi ringrazio infinitamente per la professionalità e la grandissima umanità con le quali mi avete profondamente e concretamente aiutata.
Desidero esprimere la mia sincera gratitudine per il mio relatore, il prof. Giovanni Boschetti, per la sua guida, supporto e disponibilità mostrata durante la stesura di questa tesi.
Ringrazio tutte le persone conosciute all'interno del progetto formativo in E80 Group per essere stati presenti e disponibili ad aiutarmi durante questo percorso coinvolgente.
Grazie ancora a tutti per avermi spronato a finire anche quando io stessa, non lo credevo possibile.

Abstract

Path planning is a crucial aspect of autonomous robot navigation, enabling robots to efficiently and safely navigate through complex environments. This thesis focuses on autonomous navigation for robots in dynamic and uncertain environments. In particular, the project aims to analyze the localization and path planning problems. A fundamental review of the existing literature on path planning algorithms has been carried on. Various factors affecting path planning, such as sensor data fusion, map representation, and motion constraints, are also analyzed. Thanks to the collaboration with E80 Group S.p.A., the project has been developed using ROS (Robot Operating System) on a Clearpath Dingo-O, an indoor mobile robot. To address the challenges posed by unstructured and dynamic environments, ROS follows a combined approach of using a global planner and a local planner. The global planner generates a high-level path, considering the overall environment, while the local planner handles real-time adjustments to avoid moving obstacles and optimize the trajectory. This thesis describes the role of the global planner in a ROS-framework. Performance benchmarking of traditional algorithms like Dijkstra and A*, as well as other techniques, is fundamental in order to understand the limits of these methods. In the end, the Hybrid A* algorithm is introduced as a promising approach for addressing the issues of unstructured environments for autonomous navigation of a non-holonomic robot. The core concepts and implementation details of the algorithm are discussed, emphasizing its ability to efficiently explore continuous state spaces and generate drivable paths.The effectiveness of the proposed path planning algorithms is evaluated through extensive simulations and real-world experiments using the mobile platform. Performance metrics such as path length, execution time, and collision avoidance are analyzed to assess the efficiency and reliability of the algorithms.

Abstract

La pianificazione di percorso è un aspetto cruciale della navigazione autonoma dei robot e consente ai robot di navigare in modo efficiente e sicuro attraverso ambienti complessi. Questa tesi si concentra sulla navigazione autonoma dei robot in ambienti dinamici e incerti. In particolare, il progetto mira ad analizzare i problemi di localizzazione e pianificazione del percorso. È stata condotta una revisione cruciale della letteratura esistente sugli algoritmi di pianificazione di percorso. Sono stati analizzati anche vari fattori che influenzano la pianificazione del percorso, come la fusione dei dati dei sensori, la rappresentazione della mappa e i vincoli di movimento. Grazie alla collaborazione con E80 Group S.p.A., il progetto è stato sviluppato utilizzando ROS (Robot Operating System) su un robot mobile indoor Clearpath Dingo-O. Per affrontare le sfide poste dagli ambienti non strutturati e dinamici, ROS segue un approccio combinato che utilizza un pianificatore globale e un pianificatore locale. Il pianificatore globale genera un percorso ad alto livello, considerando l'ambiente complessivo, mentre il pianificatore locale gestisce gli aggiustamenti in tempo reale per evitare ostacoli in movimento e ottimizzare la traiettoria. Questa tesi descrive il ruolo del pianificatore globale in un framework ROS. La valutazione delle prestazioni di algoritmi tradizionali come Dijkstra e A*, così come di altre tecniche, è fondamentale per comprendere i limiti di questi metodi. Alla fine, viene presentato l'algoritmo Hybrid A* come un approccio promettente per affrontare i problemi degli ambienti non strutturati nella navigazione autonoma di un robot non olonomo. Vengono discusse le concettualizzazioni principali e i dettagli di implementazione dell'algoritmo, sottolineando la sua capacità di esplorare efficacemente spazi di stato continui e generare percorsi praticabili. L'efficacia degli algoritmi proposti per la pianificazione del percorso viene valutata attraverso simulazioni approfondite e esperimenti in ambiente reale utilizzando la piattaforma mobile. Vengono analizzate metriche delle prestazioni come lunghezza del percorso, tempo di esecuzione e evitamento di collisioni per valutare l'efficienza e la affidabilità degli algoritmi.

Contents

Chapter 1

Introduction

Autonomous mobile robots play a pivotal role in both our daily lives and industrial applications, revolutionizing the way we operate and enhancing various aspects of our existence. In everyday life, they have become indispensable in logistics and transportation, facilitating the efficient delivery of goods, reducing traffic congestion, and ensuring timely services. Moreover, in the industrial sector, autonomous robots streamline production processes, improve safety, and enhance productivity. They excel in tasks such as warehouse automation, material handling, and even healthcare, where they assist with patient care and medication delivery. As society continues to advance, the importance of autonomous mobile robots will only grow, making them a transformative force in modern living and industry. A mobile robot can be described as an automated system capable of moving independently. Furthermore, when it is engineered to carry out tasks without human intervention, it achieves full autonomy. Conversely, any involvement from remote assistance or an external tool reduces the level of autonomy exhibited by the mobile robot. These interventions span a spectrum from teleoperation to providing access to environmental data, whether complete or partial, with the source of this information not originating from within the mobile robot itself. In the realm of robotics, the pursuit of autonomous navigation for mobile robots has been a long-standing challenge and a topic of significant research interest. The ability to guide robots through complex and dynamic environments safely and efficiently is paramount for their real-world applicability across a wide spectrum of industries, from logistics and transportation to healthcare and agriculture. One of the key components in achieving this autonomy is the development and implementation of effective path planning algorithms.

Path planning, in essence, involves the task of determining a collision-free path from a robot's initial position to a desired goal location, taking into account the environmental constraints, obstacles, and the robot's own dynamics. It is a crucial aspect of autonomous robotics, as it dictates the robot's ability to make informed decisions and navigate through unstructured and unpredictable surroundings. This thesis embarks on a comprehensive exploration of path planning for mobile robots, with a particular focus on addressing the challenges posed by dynamic and uncertain environments.

1.1 Problem context

The Clearpath Dingo, depicted in 1.1, is a versatile and compact indoor mobile robot and emerges as a capable platform for exploring path planning in dynamic and unstructured environments. This chapter elucidates the integration of the Clearpath Dingo into a case study on path planning, leveraging the power of the Robot Operating System (ROS) and a Light Detection and Ranging (LiDAR) sensor for accurate localization. To facilitate seamless communication and control, ROS was chosen as the underlying software framework. ROS offers a plethora of tools, libraries, and resources for robotics development, making it an indispensable choice for our path planning case study. The Clearpath Dingo was equipped with ROS-compatible hardware, including onboard computers and motor controllers, ensuring tight integration with the ROS ecosystem.

The Clearpath Dingo, complemented by ROS and LiDAR-based localization, offers a formidable

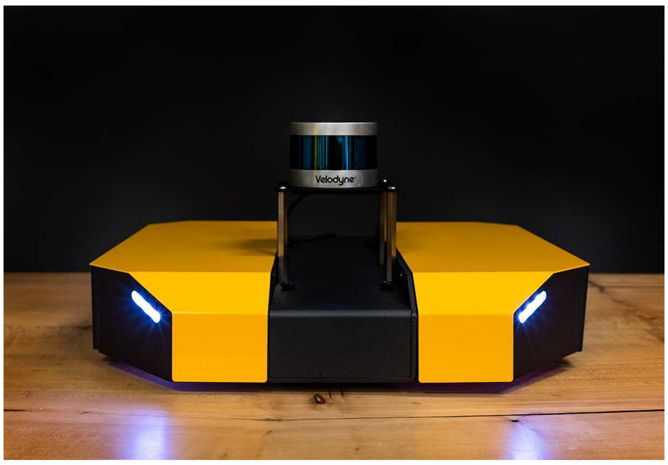


Figure 1.1: Clearpath Dingo platform

platform for exploring path planning in dynamic and uncertain environments. The integration of ROS facilitates seamless communication and control, while the LIDAR sensor ensures accurate perception of the robot's surroundings. The case study showcases the effectiveness of combining global and local planning strategies, with the Hybrid A* algorithm showing promise in addressing the challenges of unstructured environments.

1.2 Content of the thesis

The foundation of our study begins with a comprehensive examination of mobile robots kinematics and the establishment of their mathematical model. Understanding the kinematic constraints, wheel configurations, and motion models is essential for the subsequent stages of our analysis. Building upon this foundation, we embark on a thorough study of the state of the art of path planning algorithms, a critical aspect of autonomous navigation. This comprehensive review allows us to identify and analyze the strengths and limitations of existing path planning techniques, providing valuable insights into their applicability in various scenarios. Leveraging the power of the Robot Operating System (ROS), we then orchestrate the precise movements of the Dingo Clearpath within unstructured environments. This endeavor is further empowered by the integration of a LiDAR sensor, enabling accurate and real-time localization. Our objective is to assess the performance and efficiency of various path planning strategies within this context, ultimately striving for enhanced autonomy and adaptability in complex and dynamic scenarios. In our specific case, we embarked on a comprehensive study to understand the behavior of non-holonomic robots when subjected to state-of-the-art path planning algorithms. Our aim was to address and find effective solutions to the challenges posed by these robots in dynamic and unstructured environments. At the culmination of our research, one path planning algorithm distinctly stood out as particularly promising: the Hybrid A* algorithm. Its efficacy lies in its ability to navigate the intricate dynamics of non-holonomic robots efficiently. By combining elements of traditional A* search with continuous state space exploration, the Hybrid A* algorithm showcases remarkable adaptability. It is well-suited for addressing the complexities posed by unstructured environments, where traditional methods often encounter limitations. This algorithm offers the capability to efficiently generate drivable paths by considering the continuous nature of state spaces, making it a strong contender for optimizing the autonomous navigation of our Clearpath Dingo within challenging terrains. In the forthcoming sections, we delve into the core concepts and implementation details of the Hybrid A* algorithm, shedding light on its potential to provide innovative solutions for our path planning endeavors.

Chapter 2

Literature review

To embark on the journey of designing an effective path planning solution for non-holonomic wheeled robots using the Hybrid A* algorithm, it is crucial to first navigate the existing landscape of knowledge and research. This aims to explore the relevant literature, highlighting the key topics that lay the foundation for our research. We begin by examining the fundamental principles of path planning for wheeled robots, understanding the constraints imposed by their non-holonomic nature, and reviewing classical path planning algorithms that have shaped the field.This comprehensive literature review sets the stage for our theoretical exploration and analysis, offering valuable insights into the state of the art and highlighting the gaps and opportunities that our research seeks to address.

2.1 Wheeled Robots

To achieve robotic locomotion, wheeled mobile robots find extensive utility across various applications. Generally, wheeled robots offer the advantages of enhanced energy efficiency and swifter mobility when compared to alternative locomotion methods, such as legged robots or tracked vehicles. From a control perspective, their uncomplicated mechanisms and reduced stability concerns necessitate less control effort. While they may face challenges traversing rough terrains or uneven surfaces, wheeled mobile robots prove well-suited for a broad spectrum of practical environments[**6**]. In the study of mobile robots, two fundamental models come into play: the kinematic model and the dynamic model. The kinematic model primarily deals with the relationship between wheel speeds and the resulting robot velocities, while the dynamic model delves into how wheel torques translate into robot accelerations. For the scope of this section, we focus exclusively on kinematics and set aside the complexities of dynamics. Addi-

tionally, we make the simplifying assumption that the mobile robots under consideration operate on smooth, flat, and skid-free horizontal surfaces, excluding vehicles like tanks and skid-steered systems. The mobile robot model assumes a single, non-articulated rigid-body chassis, distinct from articulated setups like tractor-trailers. This chassis is defined relative to a fixed spatial reference frame s in the horizontal plane. Wheeled mobile robots fall into two primary categories: omnidirectional and nonholonomic systems. Omnidirectional robots operate without any restrictions on their chassis velocity, denoted as $\dot{q} = (\dot{\theta}, \dot{x}, \dot{y})$. In contrast, nonholonomic robots are subject to a single Pfaffian velocity constraint denoted as $A(q)\dot{q} = 0$.[1] For instance, a car-like robot adheres to this constraint, preventing it from moving directly sideways. Despite this limitation, the car can attain any (θ, x, y) configuration in a plane devoid of obstacles. This velocity constraint cannot be transformed into an equivalent configuration constraint, making it a nonholonomic constraint. The classification of a wheeled mobile robot as omnidirectional or nonholonomic is influenced partly by the type of wheels it employs, as depicted in 2.1. Nonholonomic mobile robots utilize conventional wheels, akin to those on an ordinary car. These wheels rotate about an axle perpendicular to their plane and can be steered by spinning the wheel around an axis perpendicular to the ground at the contact point. The wheels roll without sliding sideways, and this property is the source of the nonholonomic constraint on the robot's chassis. Conversely, omnidirectional wheeled mobile robots typically use either omniwheels or mecanum wheels. Omniwheels resemble regular wheels but have rollers on their outer circumference that spin freely, enabling sideways sliding while the wheel moves forward or backward without slipping. Mecanum wheels offer similar capabilities, with the spin axes of the circumferential rollers positioned differently. The sideways sliding permitted by omniwheels and mecanum wheels eliminates velocity constraints on the robot's chassis.
Notably, omniwheels and mecanum wheels are primarily designed for forward or backward movement and are not intended for steering. Due to the relatively small diameter of their rollers, they perform optimally on hard, flat surfaces. The modeling, motion planning, and control challenges encountered in wheeled mobile robots differ significantly based on whether the robot is omnidirectional or nonholonomic.

2.1.1 Kinematic constraints

In our initial assumptions, we consider a mobile robot consisting of a rigid cart with nondeformable wheels, moving on a horizontal plane. The robot's pose on this plane is defined using

[1]In robot motion planning, a Pfaffian constraint is a set of k linearly independent constraints linear in velocity, i.e., of the form $A(q)\dot{q} = 0$.

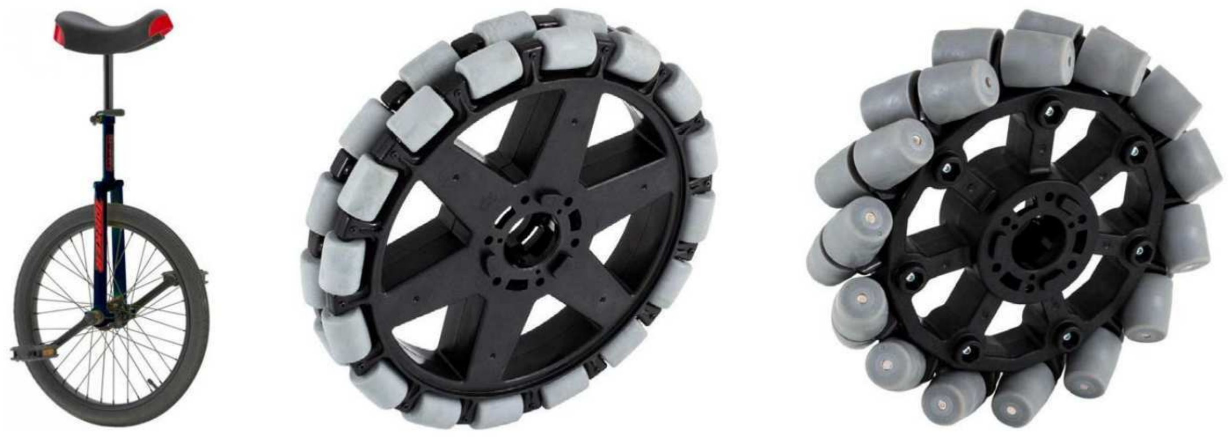


Figure 2.1: Different examples of wheels: (left) standard wheel, (middle) omniwheel, (right) mechanum wheel.

a posture vector

$$q = \begin{bmatrix} \theta \\ x \\ y \end{bmatrix} ; \tag{2.1}$$

where x and y represent the coordinates of a reference point on the cart, and θ indicates the robot's orientation with respect to an inertial frame. We assume that each wheel rotates around its horizontal axis, maintaining a vertical orientation during motion. As already mentioned, we describe two types of idealized wheels: standard and Swedish. In both cases, we simplify the wheel-ground contact to a single point, leading to kinematic constraints that ensure the point of contact remains motionless. For standard wheels, these constraints yield two independent conditions related to wheel velocity, including a non-slip and pure rolling condition. In contrast, Swedish wheels, due to the relative rotation of rollers, result in only one kinematic constraint, with its direction dependent on the wheel's construction.

Holonomic kinematic

Holonomic robots are characterized by their unique kinematic capability, which allows them to move freely and independently in all directions within their workspace. These robots possess full mobility, meaning they can translate and rotate instantly and smoothly without constraints on their motion. Holonomic robots are often equipped with omnidirectional wheels or other specialized mechanisms that enable them to navigate with exceptional agility and precision. This kinematic freedom makes them well-suited for tasks requiring precise maneuvering and dynamic navigation, such as indoor robots, autonomous vehicles, and certain industrial applications.

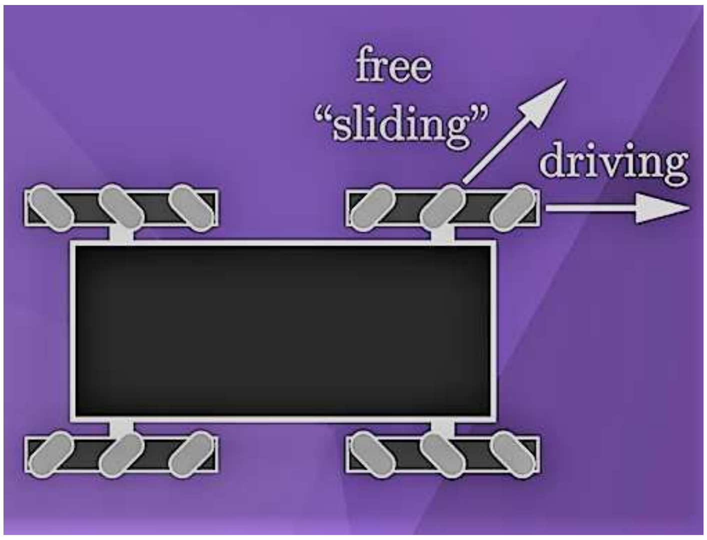


Figure 2.2: Example of a robot equipped with 4 swedish wheels

The Swedish wheels, with their unique design featuring rollers set at specific angles, enable the robot to move in any direction with ease. By understanding how each wheel contributes to the overall motion of the robot, we can derive a comprehensive kinematic model that allows us to control the robot's position and orientation accurately.

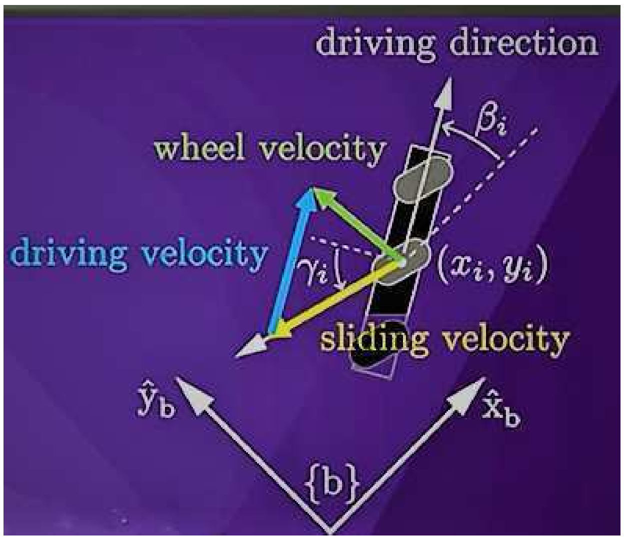


Figure 2.3: Scheme for deriving kinematic

This approach not only simplifies the analysis but also provides valuable insights into the coordinated movements required to achieve desired trajectories and behaviors in these versatile robotic platforms. We start by considering the $-i$ wheel driving speed:

$$u_i = \begin{bmatrix} 1 & tan\gamma_i \end{bmatrix} \begin{bmatrix} cos\beta_i & sin\beta_i \\ -sin\beta_i & cos\beta_i \end{bmatrix} \begin{bmatrix} -y_i & 1 & 0 \\ x_i & 0 & 1 \end{bmatrix} v_b; \tag{2.2}$$

The first term on the right of the equation represents the component in the driving direction. The second term on the right of the Equation 2.2 is the linear velocity at wheel, in the wheel frame. The last term is the linear velocity at wheel, in the chassis frame, namely b. If we rearrange those 3 terms in a single matrix $H(0)$ we obatin the relation depicted in 2.4

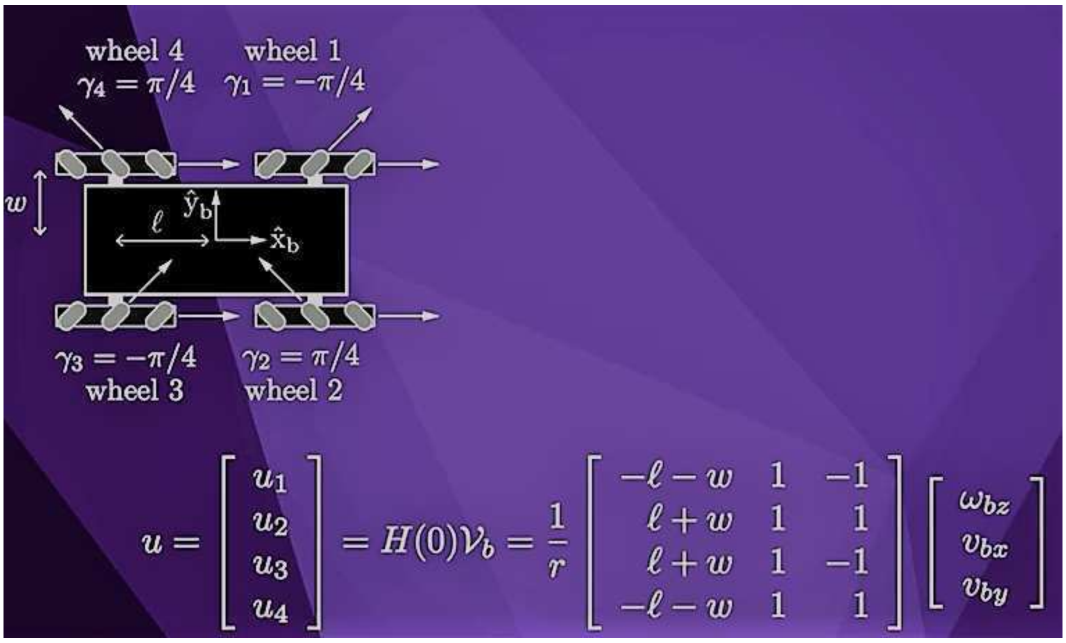

wheel 4
$\gamma_4 = \pi/4$
wheel 1
$\gamma_1 = -\pi/4$
w
ℓ
$\hat{y}_b$
$\hat{x}_b$
$\gamma_3 = -\pi/4$
wheel 3
$\gamma_2 = \pi/4$
wheel 2
$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = H(0)\mathcal{V}_b = \frac{1}{r}\begin{bmatrix} -\ell - w & 1 & -1 \\ \ell + w & 1 & 1 \\ \ell + w & 1 & -1 \\ -\ell - w & 1 & 1 \end{bmatrix}\begin{bmatrix} \omega_{bz} \\ v_{bx} \\ v_{by} \end{bmatrix}$

Figure 2.4: Wheels driving speed of a 4-swedish wheel robot

Non-holonomic kinematic

Non-holonomic kinematic constraints are fundamental limitations that restrict the motion of a system, particularly in robotics and vehicle dynamics. Unlike holonomic constraints, which can be integrated to yield a constraint on the configuration of a system, non-holonomic constraints cannot be expressed as such. Instead, they define constraints on the velocities or accelerations of a system, making certain motions impossible or highly constrained. An illustrative example is the constraint on a car-like robot, which cannot move directly sideways due to its wheel configuration. Our focus centers on car-like robots, given their prevalence in the industrial context of E80 Group, where vehicles with this specific kinematic configuration are commonly employed. The car-like robots follow the Ackermann steering geometry. [**Appendix A**] The differential constraints inherent to car-like robots, often stemming from their kinematics and dynamics are crucial considerations, ideally incorporated into the path planning process to ensure compatibility with the robot's limitations. In cases where integrating constraints into planning is impractical, this responsibility may be delegated to the controller, even though with notable challenges. Despite a car's ability to reach any position and orientation in the Euclidean plane (represented as $q = (\theta, x, y)$), its configuration space (C) is confined to $R^2 \times S^1$, meaning it cannot freely translate or rotate but can move forward and backward. Such systems, where there are fewer possible actions than degrees of freedom, are classified as underactuated. A common scenario like parallel parking necessitates both rotation and translation to achieve a specific configuration (q_1) parallel to the initial state (q_0). The constraint preventing sideways movement is expressed as the orthogonal velocity ($v_\perp$) with respect to the vehicle's heading, consistently equaling zero. The perpendicular velocity can be expressed as:

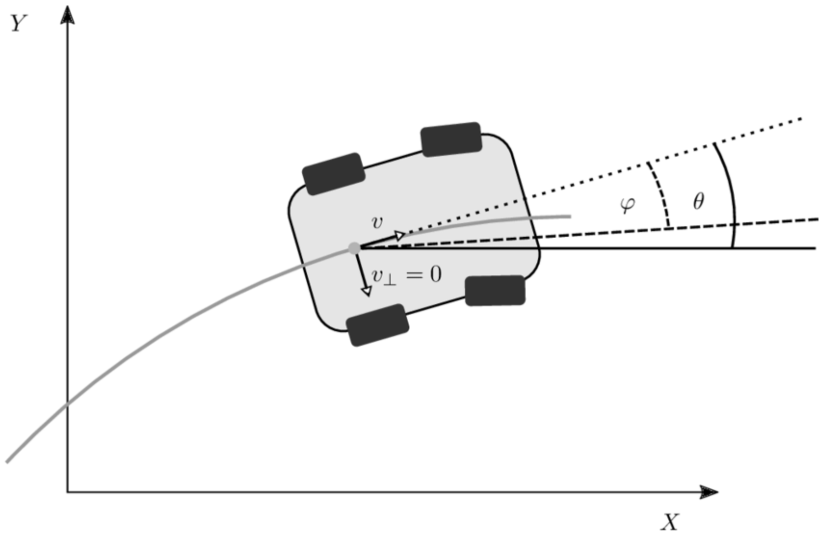


Figure 2.5: Kinematic constraints of a car-like robot

$$v_\perp = \frac{\dot{x}}{cos(\theta - \frac{\pi}{2})}; \tag{2.3}$$

$$v_\perp = \frac{-\dot{y}}{cos(\theta - \frac{\pi}{2})}; \tag{2.4}$$

This leads to the non-holonomic kinemtic constraint which is:

$$\dot{x}cos(\theta) - \dot{y}sin(\theta) = 0; \tag{2.5}$$

2.2 Configuration space

To formulate motion plans for robots, it is essential to define the precise position of the robot. To achieve this, we must provide a detailed specification of the coordinates for every point on the robot, as it is imperative to guarantee that no part of the robot encounters any obstacles. This leads us to ponder some fundamental inquiries: How much data is necessary to fully describe the position of each point on the robot? What is the optimal way to represent this information? What mathematical characteristics underlie these representations? And how can we factor in the presence of obstacles in the robot's environment when planning the robot's trajectory? This section embarks on addressing these critical questions. Initially, we clarify the concept of a robot's configuration and introduce the notion of the configuration space, a pivotal concept in the realm of robot motion planning. We will also touch upon how obstacles within the robot's surroundings impose constraints on the feasible paths. To clarify, we introduce the following definitions.

- A robot system's configuration is a full description of every point's position in that system.
- The configuration space, denoted as C-space, represents all possible system configurations. A configuration is essentially a point in this abstract space.
- We'll use q to represent a specific configuration and Q to refer to the configuration space.
- The number of degrees of freedom in a robot system corresponds to the dimension of the configuration space or the minimum parameters required to specify a configuration.

we can now precisely define the path-planning problem as follows: it involves finding a continuous mapping, denoted as

$$c : [0, 1] \longrightarrow Q; \tag{2.6}$$

This mapping should ensure that no configuration along the path results in a collision between the robot and any obstacle. It is beneficial to explicitly define the group of configurations where such collisions occur. We label this set as the "configuration space obstacle," represented as QO_i. This set encompasses all configurations where the robot intersects with an obstacle.The free space or free configuration space Q_{free} is the set of configurations at which the robot does not intersect any obstacle, i.e.,

$$Q_{free} = Q/(\bigcup_i QO_i); \tag{2.7}$$

With this notation, we define a free path to be a continuous mapping

$$c : [0, 1] \longrightarrow Q_{free}; \tag{2.8}$$

2.3 Path planning algorithms

Path planning algorithms play a pivotal role in the field of robotics and autonomous systems by enabling robots to navigate through complex environments effectively and safely. These algorithms are designed to determine a feasible path from a starting point to a goal location while considering various constraints and obstacles. Path planning encompasses a wide range of techniques, from classical approaches like Dijkstra's algorithm and A* search to more advanced methods like Rapidly-Exploring Random Trees (RRTs) and Probabilistic Roadmaps (PRMs). The choice of algorithm depends on the specific application and the robot's capabilities. Some algorithms focus on optimizing the path for speed and efficiency, while others prioritize collision avoidance and smooth trajectory generation. Path planning algorithms are fundamental in enabling robots to perform tasks such as autonomous navigation, manipulator motion planning, and even coordination in multi-robot systems. As technology advances, the development of increasingly sophisticated and adaptable path planning algorithms continues to be at the forefront of robotics research and innovation. The navigation algorithms are divided into two types: graph-based and sampling based algorithms.

2.3.1 Graph-based algorithms

Motion planners often use graphs to represent the configuration or state space (C-space). A graph consists of nodes (N) and edges (E), where each edge connects two nodes, indicating a valid move between them without obstacles. Graphs can be directed or undirected, weighted or unweighted. Graphs can be represented more compactly as a list of nodes, each with links to its neighbors. To find a motion plan in the free space graph, we search for a path from start to goal. In this section, some of the most widely used graph-based algorithms are presented.

Dijkstra

Dijkstra's algorithm, created by computer scientist Edsger W. Dijkstra in 1956 and published three years later, is a method for discovering the shortest routes between nodes in a graph. This graph can represent various networks, such as road systems. The algorithm comes in several variations. Dijkstra's original algorithm identifies the shortest path between two specific nodes. However, a more commonly used version designates one node as the 'source' and calculates the shortest paths from this source node to all other nodes in the graph, resulting in a shortest-path tree. Specifically, Dijkstra's algorithm chooses the path that minimizes the following function:

$$f(n) = g(n); \tag{2.9}$$

Here, n represents the next node on the path, and $g(n)$ represents the cost of the path from the starting node to n.

The algorithm works as follows:

1. Initialization of all nodes with distance "infinite"; initialization of the starting node with 0
2. Marking of the distance of the starting node as permanent, all other distances as temporarily.
3. Setting of starting node as active.
4. Calculation of the temporary distances of all neighbour nodes of the active node by summing up its distance with the weights of the edges.
5. If such a calculated distance of a node is smaller as the current one, update the distance and set the current node as antecessor. This step is also called update and is Dijkstra's central idea.
6. Setting of the node with the minimal temporary distance as active. Mark its distance as permanent.
7. Repeating of steps 4 to 7 until there aren't any nodes left with a permanent distance, which neighbours still have temporary distances.

As briefly explained above, the 2.6 is the pseudocode representing the required steps to find the path.

A*

In contrast to Dijkstra's algorithm, A* operates as a best-first search algorithm. Best-first search entails exploring nodes within a graph in the direction of the most promising vertex, determined by a predefined rule. This rule involves a heuristic evaluation function denoted as f(n), which relies on the properties of the vertex v. The heuristic function calculates a cost that balances the trade-off between achieving optimality and minimizing computation time. Consequently, the A* algorithm opts for the path that minimizes the evaluation function, expressed as:

$$f(n) = g(n) + h(n); \tag{2.10}$$

In this equation, $g(n)$ signifies the cost of the path from the source vertex s to vertex v, while $h(n)$ represents a heuristic function estimating the cost of the most economical path from s to v. Furthermore, by setting $g(n) = 0$, the algorithm focuses solely on the heuristic aspect, transforming into a greedy best-first search. Conversely, when $h(n) = 0$, the algorithm reverts to Dijkstra's algorithm.

```
Algorithm 1: Djkstra's Algorithm
  Input: graph
  Input: startNode
  Input: targetNode
  Output: Path
  for node in graph do
      node.score := Inf;
      node.visited := false;
  end
  startNode.score := 0;
  while true do
      currentNode := nodeWithLowestScore(graph);
      currentNode.visited := true;
      for nextNode in currentNode.neighbors do
          if nextNode.visited == false then
              newScore := calculateScore(currentNode, nextNode);
              if newScore < nextNode.score then
                  nextNode.score := newScore;
                  nextNode.routeToNode := currentNode;
              end
          end
      end
      if currentNode == targetNode then
          return buildPath(targetNode);
      end
      if nodeWithLowestScore(graph).score == Inf then
          throw NoPathFound;
      end
  end
```

Figure 2.6: Dijkstra's algorithm pseudocode

The steps required to perform A*-search are shortly described:

1. *Open List Creation*: This involves assembling a list of nodes that have been visited but not yet expanded, signifying pending tasks.

2. *Closed List Creation*: This step involves forming a list of nodes that have been both visited and expanded, which means their successors have been explored and included in the open list.

3. *Starting Node Insertion*: The algorithm inserts the starting node into the open list, assigning it a cost value of f(n) = h(n).

4. *Empty Open List Check*: The algorithm checks whether the open list is devoid of nodes. If it is, the algorithm concludes that a solution cannot be found.

5. *Best Node Extraction*: The algorithm selects and extracts the best node to visit, which is determined by having the lowest f(n) value.

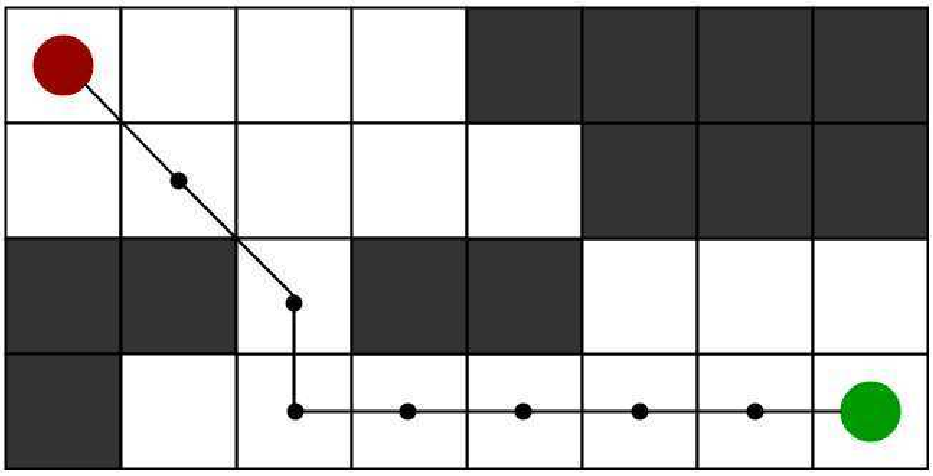


Figure 2.7: A* search

6. *Destination Node Check*: If the extracted node matches the destination node, the algorithm concludes and reports that a solution has been found.

7. *Child Node Examination*: The algorithm proceeds to examine the child nodes of the currently extracted node.

8. *Transfer to Closed List*: Any child nodes that have already been visited and found to be suboptimal are moved from the open list to the closed list.

9. *Remaining Nodes Insertion*: The algorithm inserts the remaining unvisited child nodes into the open list.

10. *Return to Empty Open List Check*: Finally, the algorithm returns to step 4 and repeats the process until a solution is found or it is determined that no solution exists.

Focused D*

The Focused D* optimizes cost updates to minimize state expansions, consequently reducing computational expenses. It is a derived algorithm from the D* [**Appendix B**], which initializes with current environmental knowledge and estimates the costs to reach the goal from each location. We selected to analyse in this thesis the Focused D* algorithm due to its superior performance over the standard D* algorithm.

Through iterative backward and forward passes, it updates these estimates while considering obstacles and changes in costs. D* efficiently adapts to dynamic conditions, making it ideal for real-time robot navigation and exploration. Similar to D*, Focused D* employs a heuristic function similar to A* for propagating cost increases and directing cost reductions efficiently.

```
Algorithm 1: A* Algorithm
Input: graph
Input: startNode
Input: targetNode
Output: Path
for node in graph do
    node score := Inf;
    node.heuristicScore := Inf;
    node.visited := false;
end
startNode.score := 0;
startNode.heuristicScore := 0;
while true do
    currentNode := nodeWithLowestScore(graph);
    currentNode.visited := true;
    for nextNode in currentNode.neighbors do
        if nextNode.visited == false then
            newScore := calculateScore(currentNode, nextNode);
            if newScore < nextNode.score then
                nextNode.score := newScore;
                nextNode.heuristicScore := newScore +
                  calculateHeuristicScore(nextNode, targetNode);
                nextNode.routeToNode := currentNode;
            end
        end
    end
    if currentNode == targetNode then
        return buildPath(targetNode);
    end
    if nodeWithLowestScore(graph).score == Inf then
        throw NoPathFound;
    end
end
```

Figure 2.8: A* algorithm pseudocode

Additionally, it incorporates a biasing function designed to account for the robot's motion between replanning operations, enhancing its adaptability in dynamic environments. The combined result is a notable reduction in runtime, often by a factor of two to three, making it an attractive choice for real-time robotic path planning. This paper embarks on the journey of elucidating the algorithm's underlying intuition, outlining the extensions it introduces, providing a practical example, conducting empirical performance comparisons, and culminating in insightful conclusions. The Focused D* algorithm was developed to address potential errors in the environment map provided to a mobile robot, particularly incomplete maps. It continuously updates the route when map changes occur, resembling the D* algorithm but with improvements. Both algorithms use an open list of candidate states and employ wave propagation to correct path errors caused by map changes. They also use heuristics and cost functions. The key difference is that Focused D* uses a heuristic that concentrates cost calculations toward the robot's goal, making it computationally more efficient while still finding the lowest-cost route in dynamic or incomplete environments. Here the steps followed by the Focused D* algorithm are presented:

1. *Initial Setup*: Begin with an initial environment map. The mobile robot is positioned at the start node.

2. *Map Updates*: As the robot moves along its path, it detects obstacles and updates the map, changing the classification of some nodes from free to impassable. Nodes behind the robot may not be part of the final trajectory and are excluded from consideration.

3. *Heuristic Function*: Define a heuristic function $g(X, R)$, where X represents any state, and R is the current robot position. This function estimates the cost for the robot to move from R to X.

4. *Cost Estimate Function*: Introduce a new function, the cost estimate function $f(X, R) = g(X, R) + h(G, R)$. Here, $f(X, R)$ represents the estimated cost of moving the robot from its current position to the goal position while passing through state X.

5. *Sorting the Open List*: In the Focused D* algorithm, sort the open list of states by their $f(X, R)$ values. If two states have equal f() values, compare their k values, and the state with the lowest k is prioritized.

6. *Optimal Path*: Similar to the D* algorithm, consider the optimal path found when the lowest value in the open list is equal to or greater than the cost of the robot's path.

7. *Cost Spread Focus*: To address errors in the map, focus the cost spread using a procedure. When an error is detected and the robot moves to a new position (e.g., R_0 to R_1), a lower limit for the value of $f(X, R1)$ is defined as $fl(X, R1) = f(X, R1) - g(R1, R0)$. This lower limit assumes the robot moved toward state X. When X is added to the open list considering the value of $fl(X, R1)$, it is evaluated with appropriate $f(X, R1)$ values, and $g(X, R1)$ is updated accordingly. This ensures that X is correctly placed in the open list with the accurate value of $f(X, R1)$.

Lifelong Planning A*

Lifelong Planning A* (LPA*) is a versatile algorithm that can be applied to the same finite search problems as A* while consistently delivering optimal solutions. What sets LPA* apart is its ability to adapt to arbitrary changes in edge costs, making it a valuable tool in dynamic scenarios. Remarkably, LPA* shares a strong algorithmic resemblance to A* but often outperforms it in terms of efficiency, especially in contexts such as route planning, robot control, and symbolic artificial intelligence planning. Beyond its practical advantages, LPA* boasts commendable theoretical properties, making it a valuable asset in various problem-solving domains, including traffic management, networking, and more.

1. *Initialization*: Create a priority queue and set 'g' and 'rhs' values for nodes, while the start node's 'rhs' is set to 0, and it's inserted into the queue.

2. *Compute Shortest Path*: Repeat until the goal is reached or the queue is empty:
   -Process the node with the lowest key in the queue.
   -Update its 'g' value based on 'rhs' if needed.
   -Update its successors and continue.

3. *Node Update (updateNode)*: Recalculate 'rhs' for a node and manage its position in the queue.
   -Calculate 'rhs' as the minimum cost from predecessors.
   -Adjust queue position based on 'g' and 'rhs' values.

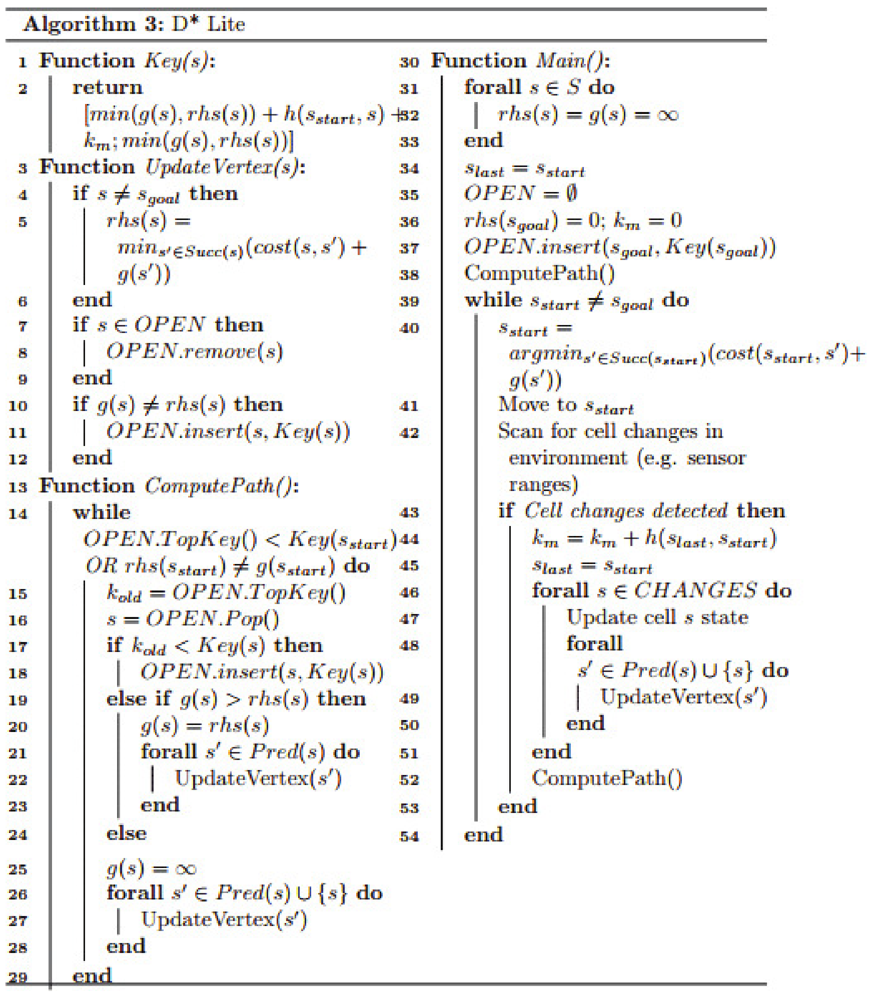
4. *Calculate Key (calculateKey)*: Determine a node's key as a combination of 'g,' 'rhs,' and a heuristic to the goal.

It repeatedly updates the 'g' and 'rhs' values of nodes while maintaining a priority queue to explore nodes with the most promising estimated costs.

```
void main() {
  initialize();
  while (true) {
    computeShortestPath();
    while (!hasCostChanges())
      sleep;
    for (edge : getChangedEdges()) {
        edge.setCost(getNewCost(edge));
        updateNode(edge.endNode);
    }
  }
}

void initialize() {
  queue = new PriorityQueue();
  for (node : getAllNodes()) {
    node.g = INFINITY;
    node.rhs = INFINITY;
  }
  start.rhs = 0;
  queue.insert(start, calculateKey(start));
}

/** Expands the nodes in the priority queue. */
void computeShortestPath() {
  while ((queue.getTopKey() < calculateKey(goal)) || (goal.rhs != goal.g)) {
    node = queue.pop();
    if (node.g > node.rhs) {
      node.g = node.rhs;
    } else {
      node.g = INFINITY;
      updateNode(node);
    }
    for (successor : node.getSuccessors())
      updateNode(successor);
  }
}
/** Recalculates rhs for a node and removes it from the queue.
 * If the node has become locally inconsistent, it is (re-)inserted into the
queue with its new key. */
void updateNode(node) {
  if (node != start) {
    node.rhs = INFINITY;
    for (predecessor: node.getPredecessors())
      node.rhs = min(node.rhs, predecessor.g + predecessor.getCostTo(node));
    if (queue.contains(node))
      queue.remove(node);
    if (node.g != node.rhs)
      queue.insert(node, calculateKey(node));
  }
}

int[] calculateKey(node) {
  return {min(node.g, node.rhs) + node.getHeuristic(goal), min(node.g,
node.rhs)};
}
```

D* Lite

The D*-Lite algorithm, created by Sven Koenig and Maxim Likhachev in 2002, is a renowned path planning technique extensively applied in robotics and artificial intelligence. D* Lite is based on Lifelong Planning A*. Its primary purpose is to solve the challenge of determining the shortest path in environments that undergo alterations. This algorithm builds upon the foundation of the well-established Dijkstra's algorithm, renowned for ensuring the shortest path from an initial node to all other nodes within a static graph. D*-Lite elevates this concept by efficiently recalculating the optimal path whenever environmental changes transpire, rendering it an ideal choice for dynamic scenarios. D* Lite is a variation of the A* algorithm, specifically



```
Algorithm 3: D* Lite
Function Key(s):
    return
        [$min(g(s), rhs(s)) + h(s_{start}, s) + k_m; min(g(s), rhs(s))$]
Function UpdateVertex(s):
    if $s \neq s_{goal}$ then
        $rhs(s) = min_{s' \in Succ(s)}(cost(s, s') + g(s'))$
    end
    if $s \in OPEN$ then
        $OPEN.remove(s)$
    end
    if $g(s) \neq rhs(s)$ then
        $OPEN.insert(s, Key(s))$
    end
Function ComputePath():
    while
      $OPEN.TopKey() < Key(s_{start})$ OR $rhs(s_{start}) \neq g(s_{start})$ do
        $k_{old} = OPEN.TopKey()$
        $s = OPEN.Pop()$
        if $k_{old} < Key(s)$ then
            $OPEN.insert(s, Key(s))$
        else if $g(s) > rhs(s)$ then
            $g(s) = rhs(s)$
            forall $s' \in Pred(s)$ do
                UpdateVertex($s'$)
            end
        else
            $g(s) = \infty$
            forall $s' \in Pred(s) \cup \{s\}$ do
                UpdateVertex($s'$)
            end
    end
Function Main():
    forall $s \in S$ do
        $rhs(s) = g(s) = \infty$
    end
    $s_{last} = s_{start}$
    $OPEN = \emptyset$
    $rhs(s_{goal}) = 0; k_m = 0$
    $OPEN.insert(s_{goal}, Key(s_{goal}))$
    ComputePath()
    while $s_{start} \neq s_{goal}$ do
        $s_{start} = argmin_{s' \in Succ(s_{start})}(cost(s_{start}, s') + g(s'))$
        Move to $s_{start}$
        Scan for cell changes in environment (e.g. sensor ranges)
        if Cell changes detected then
            $k_m = k_m + h(s_{last}, s_{start})$
            $s_{last} = s_{start}$
            forall $s \in CHANGES$ do
                Update cell $s$ state
                forall
                  $s' \in Pred(s) \cup \{s\}$ do
                    UpdateVertex($s'$)
                end
            end
            ComputePath()
        end
    end
```

Figure 2.9: D*-Lite pseudocode

based on the backward LPA* approach. Unlike A* and LPA*, D* Lite conducts its search from the goal to the start positions while estimating distances from the goal. The algorithm initiates its search by executing the 'Compute Shortest Path' function, which operates on a priority queue initially containing only the goal node. Over time, it progressively adds predecessors of each node in the priority queue until reaching the goal node. D* Lite distinguishes itself from

Comparison between D*-Lite and LPA*			
Termination		**Efficiency**	
D*-Lite	**LPA***	**D*-Lite**	**LPA***
-no fixed termination condition -continue to operate when changes occur in the setting	-stops when heuristic values of the start and goal nodes match	-more efficient for frequent localized changes	-lifelong planning -efficient with rare changes

Table 2.1: Characteristics of LPA* and D*-Lite

LPA* by utilizing the '$g()$' estimate. Additionally, it employs 'rhs' values, which are one-step look-ahead values based on the '$g()$' estimates.

Jump Point Search

Jump Point Search (JPS) stands as a noteworthy enhancement to the A* search algorithm, particularly when dealing with uniform-cost grids. Its efficacy lies in the reduction of symmetrical search paths through a process called graph pruning. By intelligently eliminating certain nodes based on informed assumptions about the surroundings of the current node and satisfying specific grid-related conditions, JPS empowers the algorithm to leap across extended distances along straight (horizontal, vertical, and diagonal) lines within the grid, in contrast to A*'s incremental steps from one grid position to the next. Importantly, JPS retains A*'s property of optimality while potentially achieving a remarkable order of magnitude reduction in running time. JPS can be described in terms of two simple pruning rules which are applied recursively during search: one rule is specific to straight steps, the other for diagonal steps. The key intuition in both cases is to prune the set of immediate neighbours around a node by trying to prove that an optimal path (symmetric or otherwise) exists from the parent of the current node to each neighbour and that path does not involve visiting the current node.

1. *Initialization*: Create a priority queue and set the start node's cost to 0.

2. *Exploration Loop*: While there are nodes in the priority queue:
   -Explore the node with the lowest cost.
   -If it's the goal, reconstruct and return the path.
   -Mark the node as explored.

3. *Successor Generation*:
   -Identify "jump points" by pruning nodes that can be skipped.
   -Consider forced neighbors based on specific conditions.

-Detect jump point successors with unique paths.

4. *Recursive Exploration*: Recursively explore nodes in the specified direction, considering jump points.

5. *End of exploration*: when an obstacle, a point of interest, or the goal is encountered.

```
function JumpPointSearch(start, goal):
    openSet = Priority Queue
    initialize start node
    add start node to openSet with priority 0

    while openSet is not empty:
        current = node with lowest priority in openSet
        remove current from openSet

        if current == goal:
            return reconstructPath(current)

        for each successor in IdentifySuccessors(current):
            if successor not in openSet:
                set parent of successor to current
                set g-value of successor
                calculate f-value of successor
                add successor to openSet with priority f-value

    return failure

function IdentifySuccessors(node):
    successors = empty list

    for each valid direction from node:
        neighbor = jump(node, direction)

        if neighbor is walkable:
            successors.append(neighbor)

    return successors

function Jump(node, direction):
    # Recursive function to find jump point
    if node is an obstacle:
        return None

    if node == goal:
        return node

    nextNode = node + direction

    if nextNode is not walkable:
        return None

    # Diagonal case
    if direction is diagonal:
        if Jump(nextNode, diagonal) or Jump(nextNode, horizontal) or Jump(nextNode,    vertical):
            return node

    return Jump(nextNode, direction)

function reconstructPath(node):
    path = []
    while node has a parent:
        add node to path
        node = parent of node
    reverse path
    return path
```

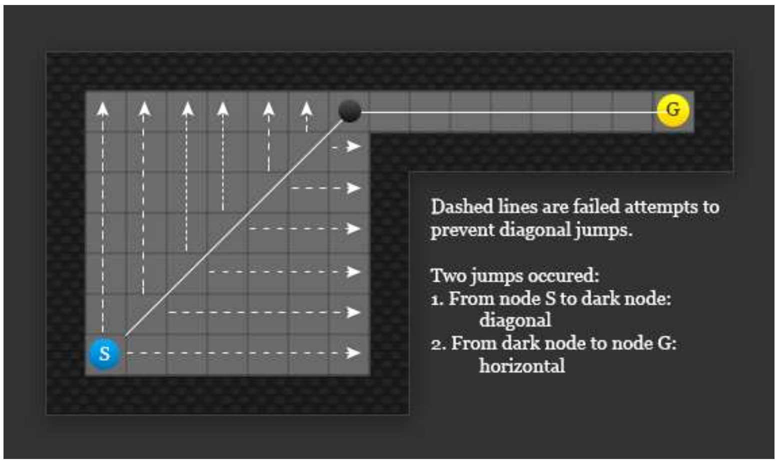

G
S
Dashed lines are failed attempts to prevent diagonal jumps.
Two jumps occured:
1. From node S to dark node: diagonal
2. From dark node to node G: horizontal

Figure 2.10: Example of path planning with JPS algorithm

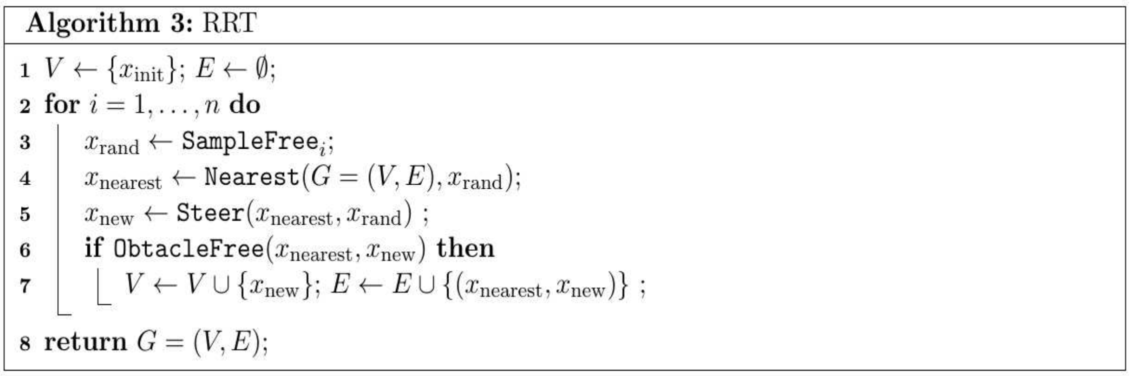
The Jump Point Search algorithm can skip lots of nodes in comparison with A*, which can speed up the operations of the pathfinding algorithm in orders of magnitude. The algorithm works better on uniform-cost grids.

2.3.2 Sampling-based algorithms

These algorithms are widely employed in robotics and motion planning to find feasible paths in complex and high-dimensional spaces. Arguably, two of the most influential motion planning algorithms based on sampling are Probabilistic RoadMaps (PRMs) (Kavraki et al., 1996, 1998) and Rapidly-exploring Random Trees (RRTs) (Kuffner and LaValle, 2000; LaValle and Kuffner, 2001; LaValle, 2006). While both methods share the fundamental concept of connecting randomly sampled points from the state space, they diverge in their approaches to constructing the graph that links these points.

RRT

The Rapidly-exploring Random Tree (RRT) algorithm is an efficient path planning technique designed for navigating unknown environments and complex, non-convex spaces using stochastic search strategies. RRT incrementally constructs a tree-like path by selecting random points within the environment and moving incrementally from the nearest existing node in the tree. This approach efficiently explores free space, making it particularly suited for path planning problems with obstacles and differential constraints. While RRT alone may not solve all planning challenges, it serves as a valuable component that can enhance various planning algorithms. However, one challenge is the potential for the tree to contain many nodes, affecting computational efficiency. In its basic form, RRT constructs a tree structure from the start point to the goal point, making it a versatile tool for optimizing pathfinding in complex scenarios.

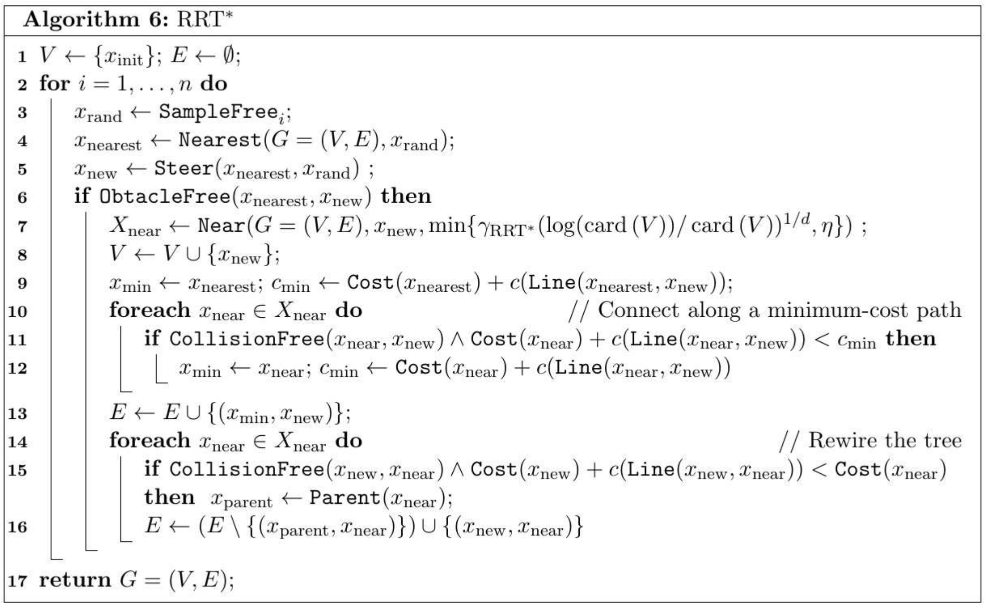


```
Algorithm 3: RRT
$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ do
    $x_{\text{rand}} \leftarrow$ SampleFree$_i$;
    $x_{\text{nearest}} \leftarrow$ Nearest$(G = (V, E), x_{\text{rand}})$;
    $x_{\text{new}} \leftarrow$ Steer$(x_{\text{nearest}}, x_{\text{rand}})$ ;
    if ObtacleFree$(x_{\text{nearest}}, x_{\text{new}})$ then
        $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$ ;
return $G = (V, E)$;
```

Figure 2.11: RRT pseudocode

RRT*

The RRT* (Rapidly-exploring Random Tree Star) algorithm is a path planning technique that excels in complex and high-dimensional spaces. It begins by initializing a tree with the starting configuration and iteratively expands it. In each iteration, it randomly samples a configuration, identifies the nearest node in the existing tree, and computes a feasible path from the nearest node to the sampled configuration. The algorithm then evaluates whether connecting the nearest node to the new node along the path reduces the overall cost. If so, it updates the tree structure to minimize the cost. This process continues until a termination condition is met, often involving reaching the goal state or a maximum number of iterations. RRT* ensures asymptotic



```
Algorithm 6: RRT*
$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ do
    $x_{\text{rand}} \leftarrow$ SampleFree$_i$;
    $x_{\text{nearest}} \leftarrow$ Nearest$(G = (V, E), x_{\text{rand}})$;
    $x_{\text{new}} \leftarrow$ Steer$(x_{\text{nearest}}, x_{\text{rand}})$ ;
    if ObtacleFree$(x_{\text{nearest}}, x_{\text{new}})$ then
        $X_{\text{near}} \leftarrow$ Near$(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}(V))/\text{card}(V))^{1/d}, \eta\})$ ;
        $V \leftarrow V \cup \{x_{\text{new}}\}$;
        $x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow$ Cost$(x_{\text{nearest}}) + c($Line$(x_{\text{nearest}}, x_{\text{new}}))$;
        foreach $x_{\text{near}} \in X_{\text{near}}$ do                    // Connect along a minimum-cost path
            if CollisionFree$(x_{\text{near}}, x_{\text{new}}) \wedge$ Cost$(x_{\text{near}}) + c($Line$(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ then
                $x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow$ Cost$(x_{\text{near}}) + c($Line$(x_{\text{near}}, x_{\text{new}}))$
        $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
        foreach $x_{\text{near}} \in X_{\text{near}}$ do                                    // Rewire the tree
            if CollisionFree$(x_{\text{new}}, x_{\text{near}}) \wedge$ Cost$(x_{\text{new}}) + c($Line$(x_{\text{new}}, x_{\text{near}})) <$ Cost$(x_{\text{near}})$
            then $x_{\text{parent}} \leftarrow$ Parent$(x_{\text{near}})$;
                $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
return $G = (V, E)$;
```

Figure 2.12: RRT* pseudocode

optimality, converging to the optimal path with increasing iterations. Finally, the optimal path is reconstructed by tracing back from the goal node through the tree structure. RRT* is renowned for its efficiency and ability to handle complex environments.

Informed RRT*

Informed RRT* is a variant of the RRT* algorithm that preserves the same probabilistic guarantees for completeness and optimality while enhancing convergence speed and the quality of the final solution[**22**]. This modified approach to RRT* is presented as a straightforward adjustment that has the potential for further extensions by advanced path-planning algorithms. Experimental results demonstrate its superior performance over RRT* in terms of convergence rate, final solution quality, and its ability to navigate challenging passages, all while exhibiting reduced sensitivity to the state dimension and planning problem range. This analysis in 2.13 compares

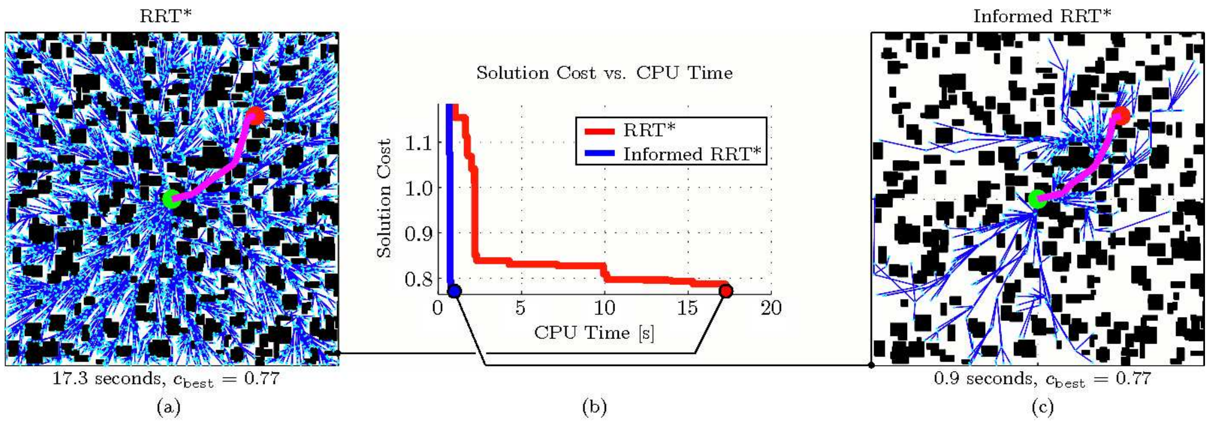


Figure 2.13: Comparison between RRT* and Informed RRT*

the performance of RRT* and Informed RRT* in terms of solution cost and computational time. Both algorithms were executed until they achieved solutions of equal cost. In Figures (a) and (c), we observe the final outcomes, while Figure (b) displays the relationship between solution cost and computational time. Figure (a) vividly demonstrates that RRT* allocates significant computational resources to explore regions of the planning problem that have no potential to enhance the current solution. In contrast, Figure (c) illustrates the targeted and efficient search strategy employed by Informed RRT*. InformedRRT* outperforms RRT* when utilizing an ellipse heuristic, as in 2.14 for several compelling reasons. It employs informed sampling, directing its exploration toward regions with a higher likelihood of reaching the goal, which accelerates convergence and reduces computational time. Additionally, InformedRRT* improves solution quality by prioritizing more efficient paths, particularly beneficial in complex environments. It avoids exhaustive exploration of unpromising regions, making it computationally efficient. Furthermore, InformedRRT* exhibits greater robustness in high-dimensional state spaces and excels in scenarios with narrow passages where RRT* may falter. However, the degree of superiority may vary depending on specific problem settings, including the quality of the heuristic and other algorithmic parameters. The heuristic sampling domain for an R^2 problem

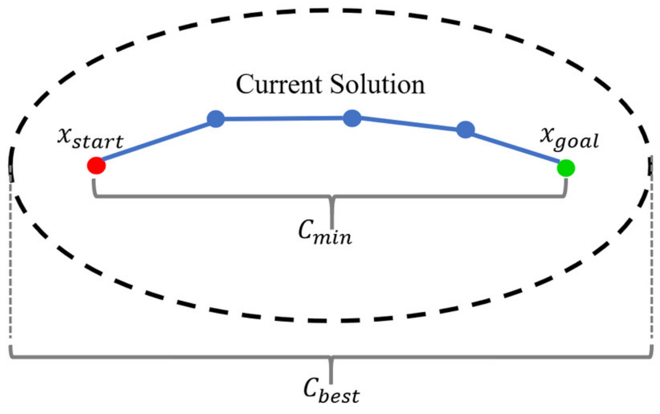


Figure 2.14: Comparison between RRT* and Informed RRT*

aiming to minimize path length, is an ellipse defined by x_{start} and x_{goal} as focal points. The ellipse's shape depends on these points, the theoretical minimum cost (c_{min}) between them, and the current best solution's cost (c_{best}). The ellipse's eccentricity is determined by the ratio of c_{min} to c_{best}.

PRM

The PRM (Probabilistic Roadmap) algorithm is a widely-used approach for motion planning in complex, obstacle-filled environments. It operates by first defining a configuration space and identifying the start and goal positions. Next, it randomly samples configurations within this space, checks for collisions with obstacles, and establishes connections between valid configurations to form a graph. Using this graph, PRM employs a path search algorithm to find a feasible trajectory from the starting point to the goal. While PRM excels in handling high-dimensional spaces and intricate obstacle layouts, its effectiveness may depend on parameter tuning to suit specific environments and requirements These following steps describe the PRM algorithm:

1. *Random Point generation*: The PRM algorithm generates a limited number of random points within a specified area.

2. *Node Clustering*: After a new node is generated, the algorithm clusters nodes into connected components. Each node is associated with a connected component, which needs to be stored for future reference.

3. *Neighbor Collection*: To perform clustering, the algorithm collects all nodes within a fixed radius of the randomly generated node. These neighboring nodes are then ordered by increasing distance or a desired metric.

4. *Cluster Assignment*: The algorithm loops through the ordered neighboring nodes and checks if the randomly generated node is not in the same cluster as the examined node. If this condition is satisfied, the new node is added to the cluster. It's important to note that a node can belong to multiple clusters.

These steps outline how PRM constructs a roadmap, connects the start and goal configurations, and uses a graph search algorithm to determine a path in complex, obstacle-filled environments while maintaining probabilistic completeness. Two important parameters that the user has to choose in this algorithm are the radius and the desired number of nodes. The radius selection used for clustering defines the structure of the roadmap. Choosing an appropriate radius impacts the speed and performance of the roadmap. A larger radius involves examining more neighbors and determining their cluster relationships. The other parameter to set is the desired number of nodes. The PRM algorithm terminates once this number of nodes is generated. Generating too many nodes can extend the time required to create the roadmap, as more time is spent examining regions of neighbors. The number of nodes chosen also affects the connectivity of the graph. Too few nodes can result in a fractured graph, especially in high-density obstacle regions, where clusters may become disconnected from the rest of the roadmap. This can lead to more irregular paths when using a shortest path planning algorithm. RRT is efficient for quick exploration and

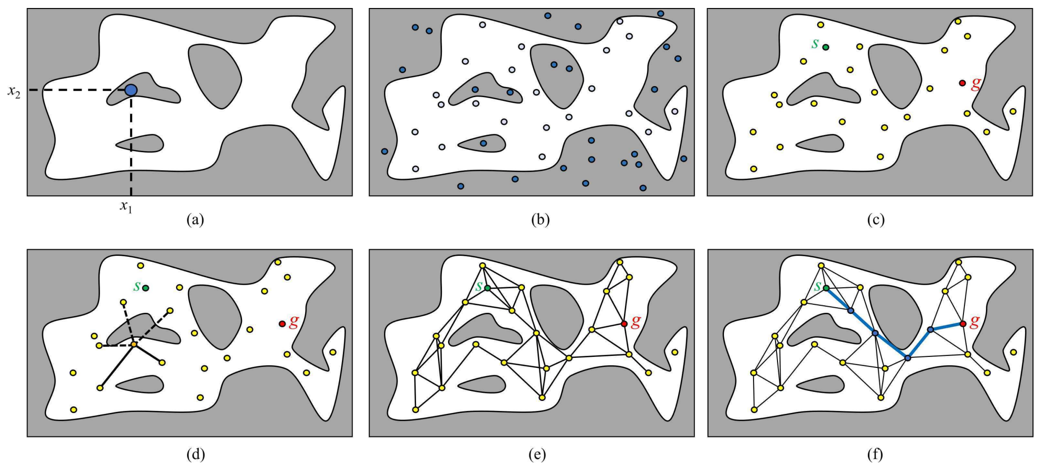


Figure 2.15: PRM algortithm steps

adaptability to dynamic environments but tends to find suboptimal paths. PRM is effective at finding optimal paths and is probabilistically complete but may require more memory and time for exploration.

The choice between RRT and PRM depends on the specific requirements of the robotic system, including the need for optimality, memory constraints, and the nature of the environment (dynamic or static). Some applications may benefit from a hybrid approach that combines the strengths of both algorithms.

Chapter 3

Problem formulation

In the rapidly evolving field of robotics, efficient path planning plays a crucial role in enabling robots to navigate through complex and dynamic environments with precision and intelligence. The Robot Operating System (ROS) has emerged as a powerful platform that not only facilitates the development of robotic systems but also provides comprehensive tools and libraries for path planning. In this chapter, we analyze the key principles, framework, and environment involved in the problem of path planning using ROS. From understanding the fundamental concepts of path planning to harnessing the capabilities of ROS for real-world applications. The goal of this chapter is to shed light on the constraints and challenges faced when applying state-of-the-art path planning algorithms within the context of Robot Operating System for real-world applications. Robotics in practical environments introduces complexities such as sensor noise, dynamic terrain, and moving obstacles, which can significantly impact the efficacy of path planning algorithms. In the next chapter, a possible solution is shown as a potential answer to some of these challenges..

3.1 ROS architecture overview

The Robot Operating System, or ROS, is a flexible and widely adopted framework for building robotic software. ROS follows a modular and distributed architecture, which is one of its key strengths. At its core, ROS employs a peer-to-peer communication model, where different software components, known as nodes, can communicate with each other. These nodes are organized into packages, each encapsulating a specific functionality or module of a robot system. ROS also incorporates a master node that facilitates node discovery and communication within the system. Moreover, ROS provides a wealth of tools and libraries, offering support for various hardware platforms and sensors, making it suitable for a wide range of robotic applications. It promotes code reusability and collaboration by fostering a large and active open-source community that contributes to a vast ecosystem of pre-built packages and libraries. This modular and collaborative architecture empowers robotics developers to efficiently design and implement complex robotic systems, making ROS a central player in the field of robotics research and development. Here we briefly discuss the main building blocks used in ROS:

- **Node**: A node is a lightweight process or executable within a ROS system. Nodes can be thought of as individual software modules that perform distinct functions, such as sensor data processing, control algorithms, or user interfaces.
  - *Publisher*: Nodes can publish data on specific topics. These topics are named channels through which nodes share information. Other nodes can subscribe to these topics to receive and process the data.
  - *Service node*: Nodes can also provide services, which are callable functions or actions that other nodes can request. Service nodes are used for tasks like setting configurations, requesting information, or performing specific actions. - *Subscriber*: Nodes can subscribe to topics to receive data published by other nodes. When a node subscribes to a topic, it will receive any messages published on that topic, allowing for inter-node communication.

- **Topic**: A topic is a named channel through which nodes can publish and subscribe to messages. Topics are used for one-to-many communication, where one node (the publisher) sends data, and multiple nodes (subscribers) can receive and process that data. Each topic is associated with a specific message type. This type defines the structure and content of the messages exchanged on that topic. Nodes that subscribe to a topic must expect and understand the message type to process the data correctly.

- **Message**: A message is a data structure that carries information between nodes. ROS messages are defined in message files and can include various data types like integers, floats, strings, arrays, and custom data types. Messages are used to represent sensor data, commands, status updates, and more.

- **ROS master**: The ROS Master maintains a registry of available topics and their publishers and subscribers. It helps nodes discover and connect to the appropriate topics, facilitating communication within the ROS network.

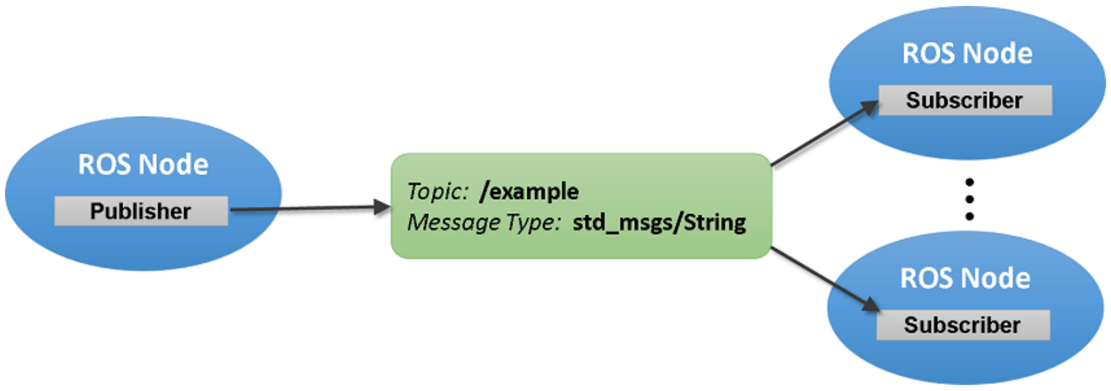


Figure 3.1: Scheme of the architecture of ROS

3.2 Configuration of the robot and setup

This section describes the configuration and setup of the Clearpath Dingo-O robot, which served as the primary platform for our research in autonomous navigation. We provide insights into the robot's hardware specifications, the control system utilized, and the sensor equipment employed for navigation. The Dingo robot is a versatile, agile indoor mobile platform tailored for both research and educational purposes. It offers two drive systems: differential and omnidirectional, along with expandable power and computing options, making it exceptionally adaptable to a diverse array of robotic applications. These applications span autonomous navigation, mobile manipulation, and mapping, showcasing Dingo's multifaceted capabilities. Dingo seamlessly integrates with the ROS (Robot Operating System) and Gazebo simulation environment, and it effortlessly supports a wide assortment of robot sensors and accessories, making it a hassle-free choice for robotic development and experimentation.

Robot Hardware

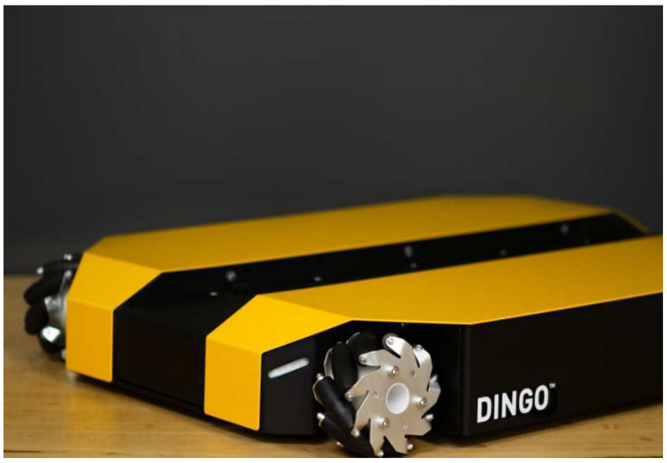


Figure 3.2: Swedish wheels mounted on Dingo-O

- **Swedish wheels**: The Dingo-O is equipped with four Swedish wheels, as depicted in 3.2, allowing for omnidirectional movement. This design grants the robot exceptional maneuverability, enabling it to navigate through intricate spaces with ease.

- **Onboard Computer (Orin)**: To facilitate control and data processing, we utilized an Orin, 3.3 onboard computer. Orin is capable of running ROS, serving as the central hub for interfacing with the robot's various components and executing navigation commands.

Figure 3.3: NVIDIA Jetson AGX Orin

- **Pepperl Fuchs R2000 LiDAR**: Navigation and localization are critical aspects of our research, and the Pepperl Fuchs R2000 LiDAR, 3.4 played a pivotal role. This LiDAR sensor provided accurate environmental mapping and robot localization capabilities, enabling the robot to perceive its surroundings.

Figure 3.4: Pepperl Fuchs R2000 LiDAR

Robot Hardware

- **ROS**: ROS served as the overarching software framework for our robot's control and navigation. It offered a modular and extensible architecture that facilitated communication between various hardware components and the execution of state-of-the-art algorithms.

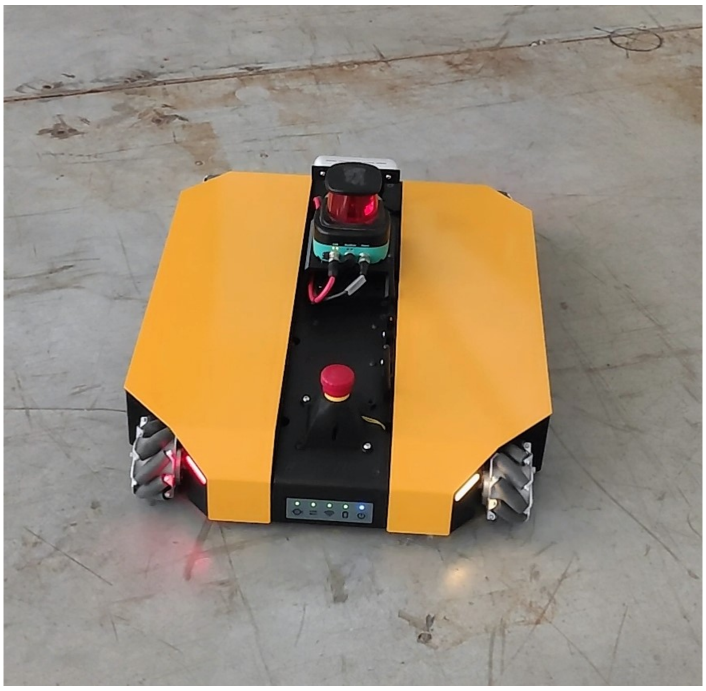


Figure 3.5: Complete Dingo setup

3.3 Navigation stack

The ability to navigate through unstructured environments is a fundamental skill exhibited by intelligent beings, and it stands as a central point of interest in this research endeavor. Navigation, in essence, constitutes a multifaceted task that hinges on the development of an internal representation of the surrounding space. This representation is constructed with the aid of sensors. It serves a dual purpose, concurrently supporting continuous self-localization (the awareness of one's current position, represented as "I am here") and the representation of the intended destination (depicted as "I am going there"). In practical terms, this research investigates and analyzes the 2D navigation stack used in ROS. This stack operates by assimilating information from various sources, including odometry data, sensor input streams, and a specified goal pose. Through a complex yet coherent process, it computes and generates safe velocity commands. These commands are subsequently transmitted to a mobile base, facilitating the robot's movement in a manner that ensures safety while navigating through the dynamic and often unpredictable environments encountered in the course of the research. The Navigation Stack is designed to offer a high degree of versatility, there are three primary hardware prerequisites that govern its applicability:

- **Compatibility with Differential Drive and Holonomic Wheeled Robots**: The Navigation Stack is tailored to function seamlessly with both differential drive and holonomic wheeled robots. It operates under the assumption that control of the mobile base involves the issuance of desired velocity commands, specifying the velocities along the x-axis, y-axis, and angular (theta) velocity.

- **Planar Laser Requirement**: A crucial component is the presence of a planar laser mounted on the mobile base. This laser plays a pivotal role in the processes of map building and localization.

- **Considerations for Robot Shape**: It's worth noting that the Navigation Stack was initially developed with a square robot in mind, thus yielding optimal performance with robots that approximate a square or circular shape. Although it can function with robots of diverse shapes and sizes, larger rectangular robots navigating through tight spaces like doorways may pose certain challenges.

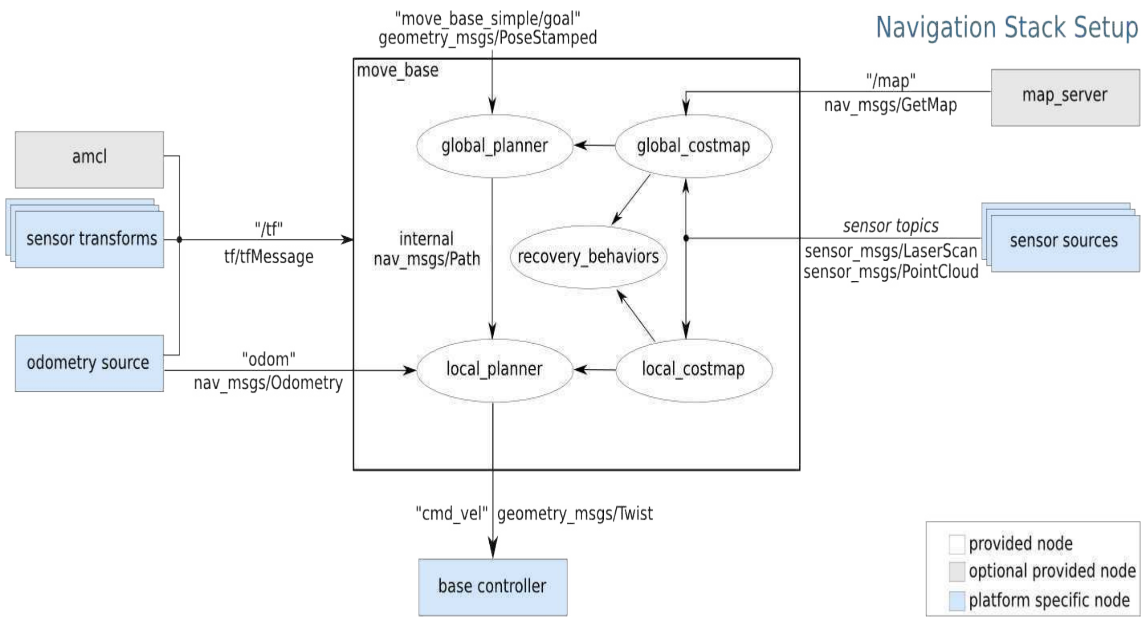


Figure 3.6: Scheme of the main blocks used in navigation

The **move base** node is the central brain of the Navigation Stack in ROS, as depicted in 3.6. It orchestrates the entire navigation process, from setting goals to generating safe trajectories,

obstacle avoidance, and handling dynamic environments. This modular and well-coordinated architecture allows robots to autonomously navigate through complex and unstructured environments effectively.
A brief presentation of the main blocks used in the task of navigating is here listed:

- Global and Local Planning: The $movebase$ node oversees both global and local planning. Global planning involves creating a high-level path from the robot's current position to the desired goal. This is often based on a global costmap that represents the environment. Local planning, on the other hand, deals with obstacle avoidance and fine-tuning the robot's trajectory in real-time using a local costmap.

- Sensor Data Fusion: It integrates data from various sensors, including odometry, laser scans, and possibly other perception sensors. These sensor inputs are used to build and update the costmaps that guide the robot's movement.

- Goal Handling: The $movebase$ node handles goal requests from higher-level systems or user commands. When a new goal is received, it plans a path to reach that goal while avoiding obstacles in the environment.

- Obstacle Avoidance: It continuously monitors the robot's environment using the local costmap and adapts the trajectory to avoid collisions with dynamic obstacles or changes in the environment.

- Feedback and Recovery: $movebase$ provides feedback to higher-level systems and can trigger recovery behaviors if the robot encounters unexpected situations or gets stuck. Recovery behaviors might include rotating in place, backing up, or attempting a different path.

Our primary focus lies in addressing the intricate challenges of the path planning problem, and as such, we will delve extensively into this pivotal aspect. However, it is equally essential to provide a comprehensive understanding of the localization stack. Localization plays a crucial role in enabling a robot to determine its precise position within its environment, a fundamental prerequisite for effective path planning and navigation. Thus, while our central emphasis remains on path planning, we will ensure to provide insightful coverage of the indispensable localization stack to offer a well-rounded perspective on autonomous robot navigation.

3.3.1 Localization

Localization in ROS (Robot Operating System) is a critical component of autonomous robotic systems, enabling robots to determine their precise position and orientation within an environment. This information is essential for accurate navigation, as it allows robots to make informed decisions about their movements, avoiding obstacles and reaching their intended destinations [**1**]. Localization in ROS is typically performed using one of the following methods: odometry-based localization, map-based localization and visual localization. Odometry-Based localization estimates the robot's position by tracking wheel movements, but it can drift over time. Map-Based Localization compares sensor data to a pre-built map, matching features to determine the robot's position. Visual Localization relies on visual cues or landmarks to estimate the robot's position and orientation, often using SLAM techniques for mapping and tracking.

Map-based localization was selected as the method of choice to assess the robot's behavior in

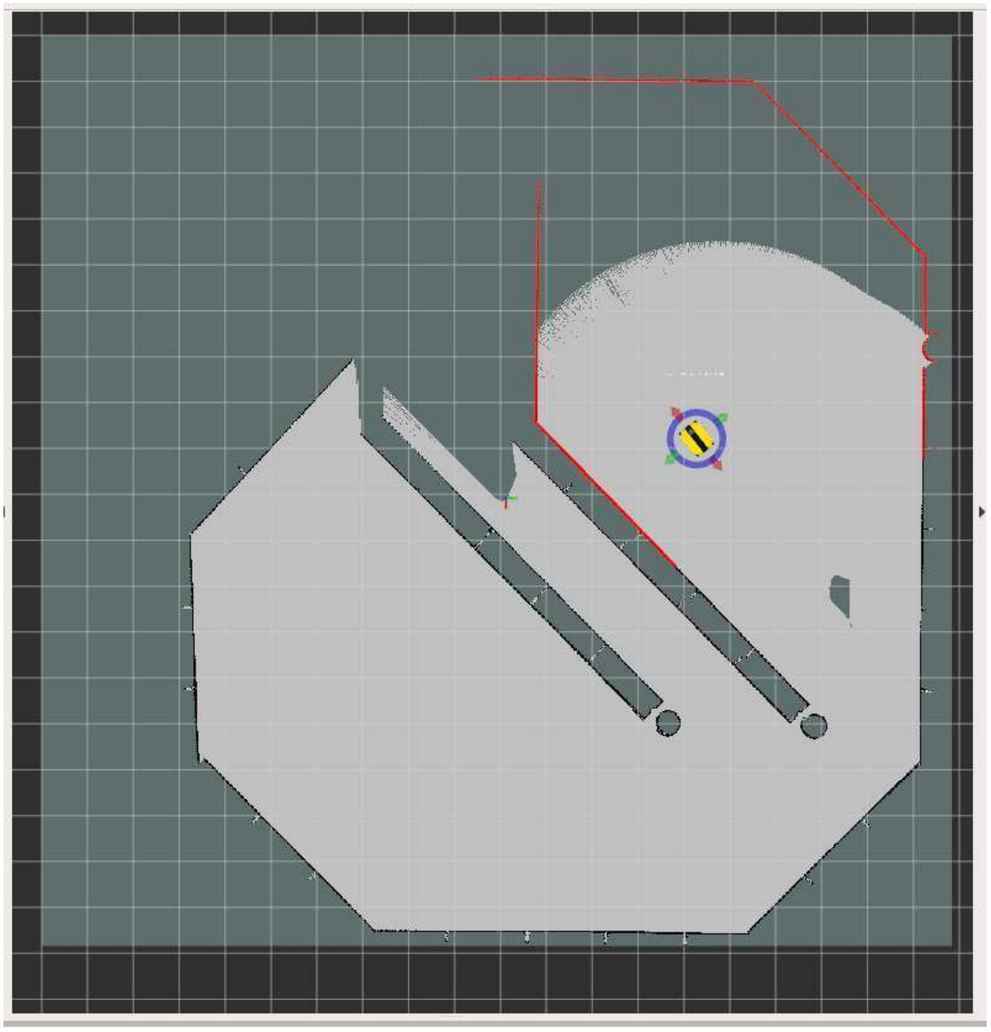


Figure 3.7: Scheme of the main blocks used in navigation

path planning, primarily due to its reliance on a fixed map generated through SLAM (Simultaneous Localization and Mapping). This map, once created as in 3.7 and saved, served as a stable reference during localization tests, aiding in evaluating the performance of the AMCL (Adaptive Monte Carlo Localization) algorithm [**2**]. AMCL is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map. However, over time, a drift in the robot's position estimation became apparent. To rectify this issue, the SnapMAPIcp node within ROS was introduced. This additional node effectively corrected the drift that had occurred when relying solely on the AMCL node for localization, ensuring a more accurate and stable robot pose estimation during navigation tasks. SnapMapICP is a ROS node that improves robot localization with AMCL. It creates a point cloud from the map and matches it to laser scans using ICP. If enough quality matches are found and the robot's pose differs significantly from AMCL's estimate, it sends a new initial pose to AMCL, correcting drift. Parameters control its behavior, and it reinitializes at defined intervals to enhance localization stability.

3.3.2 Navigation

When the $movebase$ is activated on a properly configured robot, its primary objective is to guide the latter towards a desired pose while maintaining a user-defined tolerance [**31**]. In situations where there are no dynamic obstacles, the $movebase$ node will persistently strive to bring the robot within this tolerance of its target pose or, if necessary, communicate failure to the user. To use the move base node in navigation stack, we need to have a global planner and a local planner. There are three global planners that adhere to nav core::BaseGlobal Planner interface: carrot planner, navfn and global planner. One of the commonly used global planners is the Dijkstra's algorithm, which calculates the shortest path in a map to reach the goal while avoiding obstacles. Another popular choice is the A algorithm$*$, which is efficient and can find optimal paths. ROS also provides a flexible framework, allowing users to integrate custom global planners if needed. A prevalent choice for the local planner is the Trajectory Rollout algorithm, which generates local paths for the robot to follow while avoiding obstacles. The Dynamic Window Approach (DWA) is another frequently used method, which focuses on selecting velocities that lead to collision-free navigation. Like global planners, ROS allows for customization and integration of different local planning algorithms to suit specific robot configurations and environments. These global and local planners work in tandem to enable ROS-based robots to navigate autonomously

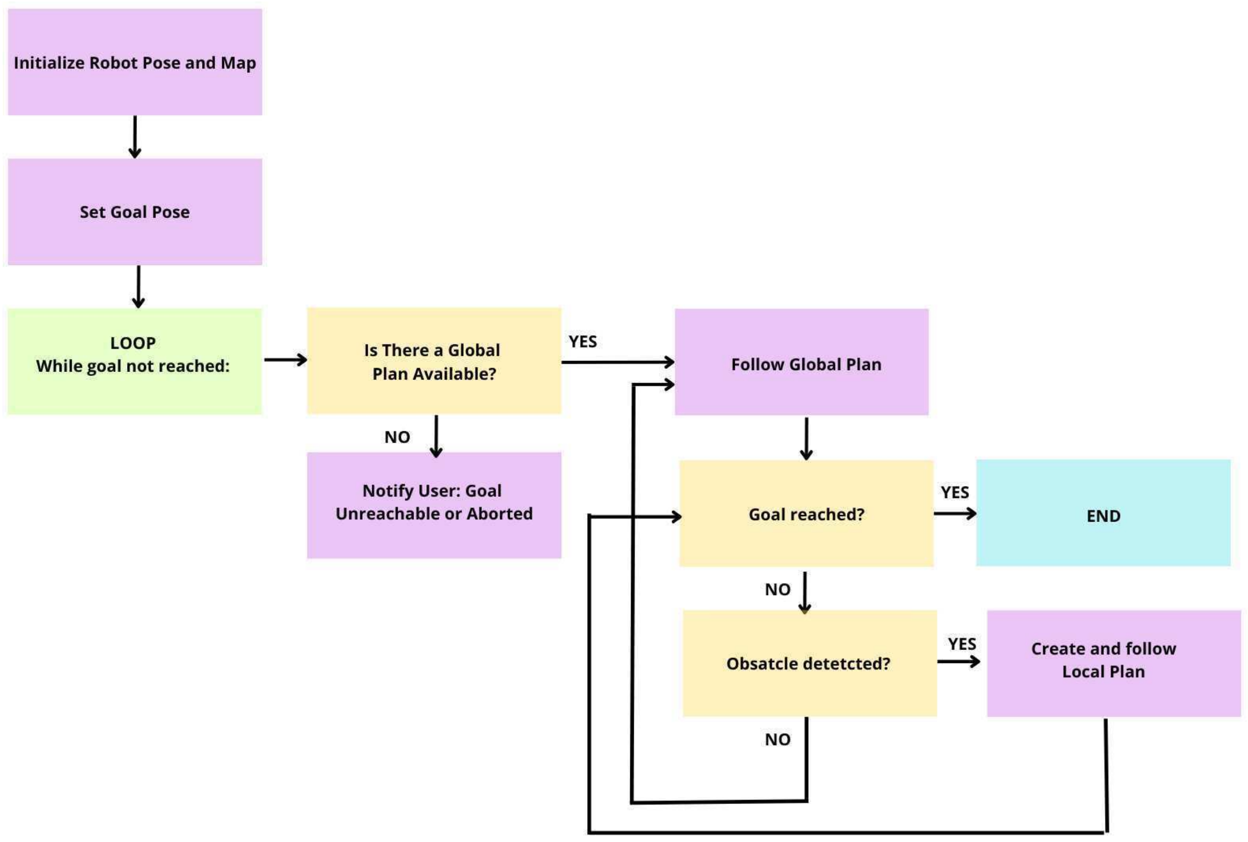

Initialize Robot Pose and Map
Set Goal Pose
LOOP
While goal not reached:
Is There a Global Plan Available?
YES
NO
Notify User: Goal Unreachable or Aborted
Follow Global Plan
Goal reached?
YES
END
NO
Obsatcle detetcted?
YES
Create and follow Local Plan
NO

Figure 3.8: Flowchart of $movebase$ node functioning

in diverse environments, providing adaptability and performance suitable for a wide range of robotic applications.

3.3.3 Global planner

The global planner in ROS operates by first initializing the map of the robot's environment, which includes information about obstacles and free space. Once a user-defined goal pose is provided, the global planner employs pathfinding algorithms like Dijkstra's or A* to compute a high-level path from the robot's current position to the goal. This pathfinding process treats the environment map as a graph, searching for the shortest or most optimal route while considering a cost function that factors in parameters such as distance and terrain difficulty. The output of this computation is a sequence of waypoints, typically represented as poses in terms of position and orientation, providing a planned trajectory for the robot. Importantly, the global planner periodically updates this trajectory to adapt to changes in the environment or new obstacles, ensuring the robot can safely and efficiently navigate towards its goal while avoiding static obstacles.

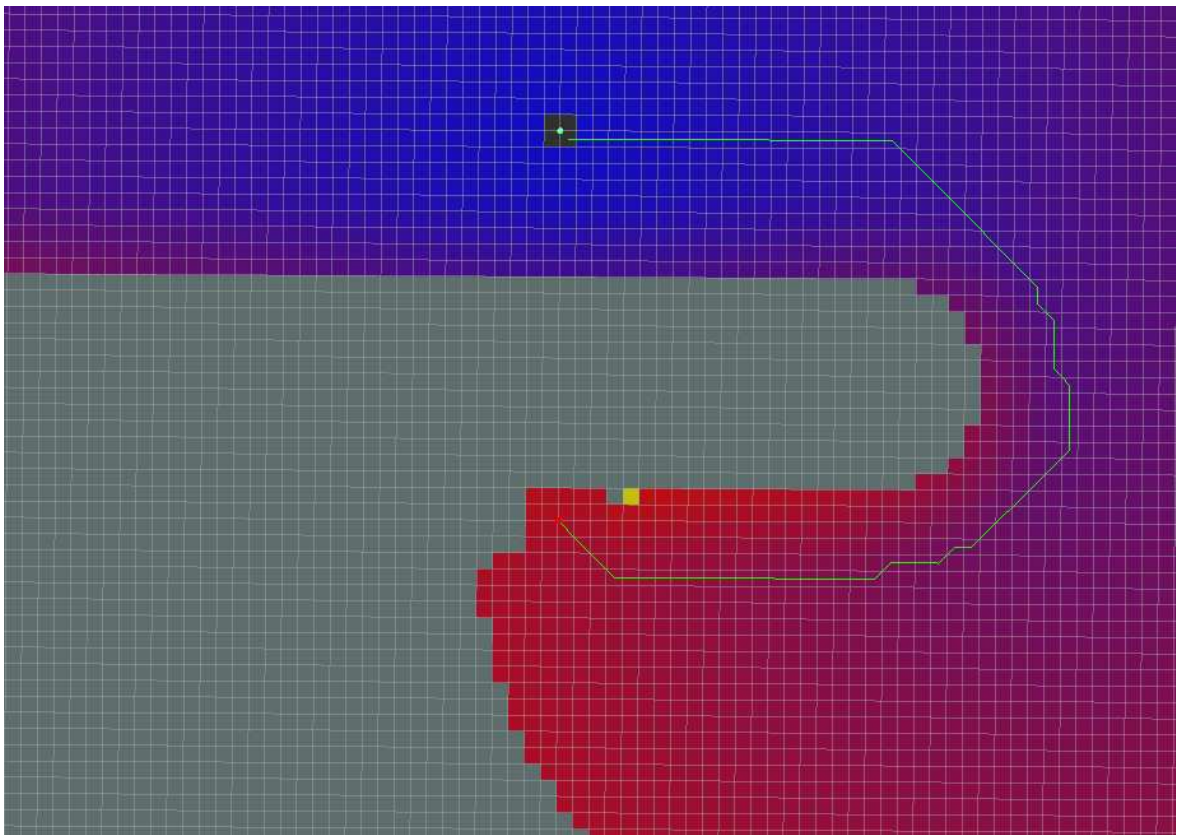


Figure 3.9: Global plan (green) using Dijkstra's algorithm

3.3.4 Local planner

At its core, local planning revolves around the continuous search for an appropriate local path within each control cycle. This involves generating a set of potential trajectories and assessing their viability. Each trajectory is carefully examined to determine if it collides with obstacles, and a rating is assigned to facilitate the selection of the most suitable option. It's crucial to note that the implementation of this principle can vary significantly based on factors such as the robot's physical shape, its actuators, and the specific environment it operates in. Numerous specialized methods exist for trajectory generation and exploring the space of potential trajectories to identify the optimal one. To accommodate this variability, a range of interfaces and classes have been designed to encapsulate these fundamental local planning principles in a generic manner. These abstractions provide a framework that can be tailored to specific requirements. In the Robot Operating System (ROS), there are several local planners that can be used for path planning and obstacle avoidance. Some of the main local planners used in ROS include:

- *base local planner*: This is a widely used local planner in ROS, and it provides a set of local planning algorithms, including Trajectory Rollout and Dynamic Window Approach (DWA), to generate feasible robot trajectories while avoiding obstacles.

- *dwa local planner* The Dynamic Window Approach (DWA) is a specific local planner that focuses on generating feasible trajectories by considering the robot's dynamics, maximum velocity, and acceleration constraints. It is part of the *base local planner* package.

- *teb local planner*: The Timed Elastic Band (TEB) local planner is designed for holonomic and non-holonomic robots. It optimizes trajectories in time-space and considers dynamic constraints, making it suitable for complex and dynamic environments.

- *eband local planner*: The Elastic Band local planner is based on a potential field method and uses an elastic band to generate paths that avoid obstacles while minimizing path length. It is particularly suitable for mobile robots and manipulators.

For our trials, we predominantly relied on the *dwa local planner* and *teb local planner* within the Robot Operating System (ROS) due to their exceptional suitability for addressing the specific characteristics and challenges posed by non-holonomic robots.

dwa local planner

The Dynamic Window Approach (DWA) local planner is a key choice when dealing with non-holonomic robots [**26**]. Non-holonomic robots have limitations on their motion, such as maximum velocity and acceleration constraints, which can significantly impact their ability to navigate effectively. DWA takes these constraints into account during path planning, making it ideal for non-holonomic robots, as in 3.10.

- Velocity Constraints: DWA considers the robot's dynamics, allowing it to calculate feasible trajectories by considering the robot's maximum achievable velocity and acceleration. This ensures that the robot can follow the generated paths more accurately, even in challenging environments.

- Obstacle Avoidance: DWA excels at obstacle avoidance. It evaluates multiple potential trajectories and selects the one with the lowest risk of collision. This is crucial for non-holonomic robots, which may have difficulties making sharp maneuvers to avoid obstacles.

teb local planner

The Timed Elastic Band (TEB) local planner is another excellent choice for non-holonomic robots. It offers unique advantages, especially when navigating in complex, dynamic environments [**33**].

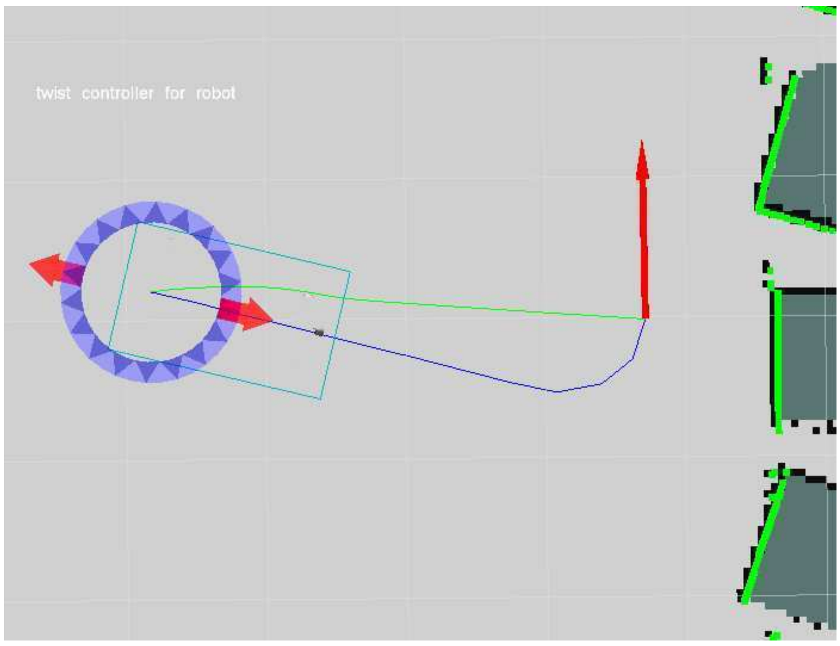


Figure 3.10: Global plan (blue), local plan obtained by using DWA approach (green)

- Dynamic Environment Handling: TEB optimizes trajectories in time-space, which is beneficial for non-holonomic robots operating in dynamic surroundings. It can adjust paths and timings to account for moving obstacles or changing conditions, making it particularly robust.

- Holonomic and Non-Holonomic Support: TEB is versatile and can accommodate both holonomic and non-holonomic robots. This adaptability allows you to use it across different robot platforms while maintaining high-quality navigation performance.

In summary, the selection of the *dwa local planner* and *teb local planner* for your trials was driven by their ability to address the specific challenges posed by non-holonomic robots. These planners take into account the robot's dynamics, velocity constraints, and obstacle avoidance needs, making them valuable tools for achieving precise and reliable navigation results in a variety of dynamic environments.

3.4 Limitations of current approach

The ROS navigation stack is a powerful tool for enabling autonomous navigation in a wide range of environments. However, when the transition is made from controlled, structured environments to real-world industrial settings, certain challenges and limitations become apparent. One of the critical limitations pertains to the repeatability of navigation, and this limitation primarily arises due to the unpredictable behavior of the local planner.

1. **Unpredictable Local Planner Behavior**: In industrial environments, robots often encounter dynamic and cluttered spaces with various obstacles and machinery. The local planner, responsible for real-time obstacle avoidance and trajectory generation, may struggle to exhibit consistent and predictable behavior in such complex scenarios. The local planner's responses can vary depending on the specific arrangement of obstacles, sensor noise, and other dynamic factors, making it difficult to achieve precise and repeatable movements.

2. **Repeatability Challenges**: In industrial automation and manufacturing, repeatability is a fundamental requirement. Robots must execute tasks with a high degree of accuracy, returning to the same positions and orientations reliably. The inherent unpredictability of the local planner can hinder this repeatability, leading to inconsistent performance in repetitive tasks, which is a critical concern in settings where precision is paramount.

3. **Customization Challenges**: To mitigate the limitations, industrial users often resort to customizing and fine-tuning the local planner's parameters. While this approach can improve performance to some extent, it requires in-depth expertise and substantial effort, which may not always be practical in industrial settings with rapidly changing requirements.

4. **Safety Considerations**: The unpredictable behavior of the local planner can also raise safety concerns, as robots may occasionally make unexpected movements when trying to avoid obstacles. In industrial environments, ensuring the safety of human operators and adjacent equipment is of utmost importance.

5. **Alternative Solutions**: In response to these challenges, some industrial applications opt for alternative navigation solutions, including specialized motion planning algorithms, external sensors (e.g., vision systems), and more deterministic control methods. These approaches provide more control over robot behavior but may require additional integration efforts.

Even when "limiting" the local planner to follow the global planner, this results in a poor navigation. In fact, the global planner typically generates a sequence of positions as a path without taking into account the robot's orientation or its kinematic constraints. In practice, industrial robots have specific limitations regarding their motion, such as maximum joint velocities, acceleration, and payload considerations. When the global planner fails to consider these constraints, it may produce paths that are geometrically sound but impractical for the robot to follow precisely. Moreover, industrial robots often require agility and precise maneuvering, especially when navigating through constrained spaces or when performing tasks that demand intricate movements. The global planner's focus on minimizing distance may not align with the robot's ability to make sharp turns or adjustments, leading to paths that are suboptimal for the robot's maneuverability. As a result, when industrial robots attempt to follow paths generated solely by the global planner, they may exhibit erratic behavior, struggle with precise positioning, or even fail to execute the path altogether. This disconnect between global path planning and local execution can lead to inefficiencies, safety concerns, and challenges in achieving the required level of repeatability and precision that are critical in industrial applications. To address these concerns, industrial navigation often involves a combination of custom local planning, sensor feedback, and careful consideration of the robot's kinematics and constraints to ensure that the generated paths are not only optimal in theory but also practical and navigable in the real-world industrial environment.

Chapter 4

Hybrid A* Algorithm

In the context of the industrial landscape at E80 Group, the imperative need for an advanced path planning algorithm became evident as the decision to transition away from the Robot Operating System (ROS) was made. While ROS had been instrumental in facilitating research and development, the industrial vehicles deployed were non-holonomic car-like robots, presenting unique challenges for navigation. Traditional algorithms such as Dijkstra or A*, while effective in certain scenarios, yielded paths with sharp and edgy turns that were ill-suited for the precise and safe navigation demanded in industrial environments. Moreover, to ensure seamless operations and avoid costly replanning actions by a "local planner," it was fundamental to find an algorithm capable of simultaneously accommodating the kinematic constraints and the unique shape of the robot. This quest marked a pivotal shift in the pursuit of efficient and streamlined operations at E80 Group, calling for the development of a versatile path planning solution tailored to the specific needs of the industrial context.

4.1 Overview and Algorithmic Description

Hybrid A* is a path planning algorithm designed for vehicles with continuous state spaces, making it well-suited for smooth and efficient navigation, particularly in complex, real-world environments. It starts by initializing the start and goal states and utilizes cost and heuristic functions to guide its search. The algorithm explores a discretized state lattice, efficiently evaluating potential states and prioritizing them based on cost. Once the goal is reached, Hybrid A* reconstructs an optimal path, considering vehicle dynamics and constraints, ensuring feasible and smooth trajectories. Its key strengths lie in its ability to handle continuous state spaces, efficiently explore them, and generate paths tailored to a vehicle's kinematic constraints, making it invaluable for autonomous vehicles and robots operating in dynamic and intricate surroundings.

4.2 Hybrid A* search

The Hybrid A* algorithm demonstrated its effectiveness in the DARPA Urban Challenge, a robotics competition orchestrated by the U.S. Government back in 2007. In the subsequent years, Dolgov et al. provided valuable insights into the algorithm in their publications. The behavior of the Hybrid A* algorithm closely resembles that of the A* algorithm. However, the critical distinction lies in the fact that state transitions occur in continuous space, as already de-

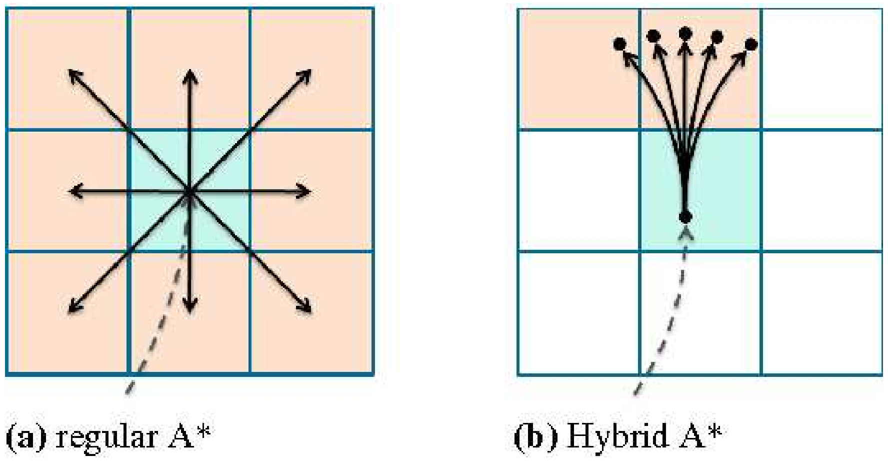


Figure 4.1: Different node expansions between A* and Hybrid A*

scribed, rather than in a discrete manner. One of the most significant shortcomings of earlier approaches to path planning for non-holonomic robots is that the resulting paths are discrete, often leading to non-executable trajectories due to abrupt changes in direction. The Hybrid A* search implicitly constructs the graph on a discretized grid, allowing vertices to access any continuous point on the grid. To manage the infinite nature of a continuous search space, the algorithm employs grid cell discretization, thus constraining the growth of the graph. Since transitions between vertices lack a predefined structure, it readily accommodates the non-holonomic nature of state transitions. Typically, the search space is three-dimensional, encompassing the state space X comprising x, y, and θ, forming a discretized cuboid. The base of this cuboid represents the x, y position, while the height denotes the heading θ of a vertex. Further in the description are outlined the steps involved in the Hybrid A* search [**4**]. Similar to the conventional A* search, it begins by establishing empty sets O and C, along with setting the predecessor state of the initial state to null. The initial state is then placed on the open list. The while loop starts, continuing until either the open list is empty or the goal state is reached. If the currently expanded

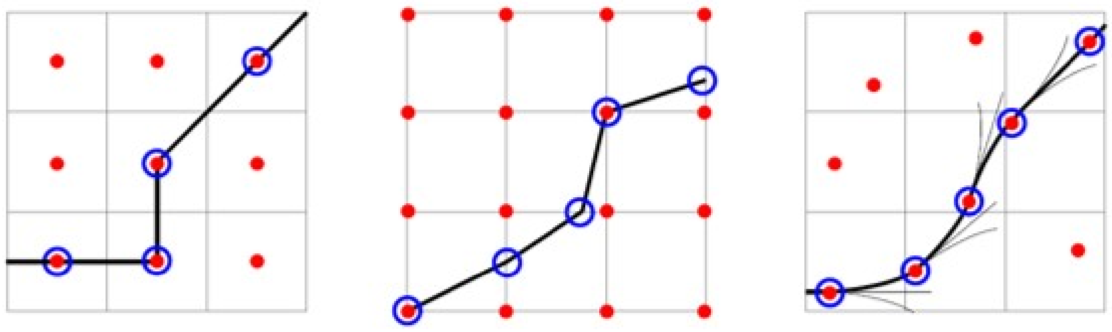


Figure 4.2: Comparison between A* (left), Field D* (centre) and HA* (right)

vertex is not the goal vertex, new successors are generated for all available actions $u \in U(x)$. If the successor is not already in C, the algorithm calculates the cost-so-far for the vertex. If the vertex is not in O or if the cost-so-far is smaller than the cost for a vertex with the same index in O, the successor is assigned a pointer to its predecessor, and the cost-so-far and cost-to-come are updated. Subsequently, the vertex is either pushed onto the open list or its key is reduced using the new value $f(xsucc)$. It's worth emphasizing that even though Hybrid A* rounds the state to prune similar branches, the expansion always occurs from the actual state value rather than the rounded one. In this section the search used in Hybrid A* algorithm is analyzed and deeply studied. The schematic description is given:

1. *Initialization*: The algorithm begins by initializing the start and goal states. These states are represented in continuous space but are converted into grid coordinates to work within the discrete grid-based map. The start node is assigned a COST of 0. Two dictionaries, namely openList and ClosedList, are created to keep track of explored and unexplored states.

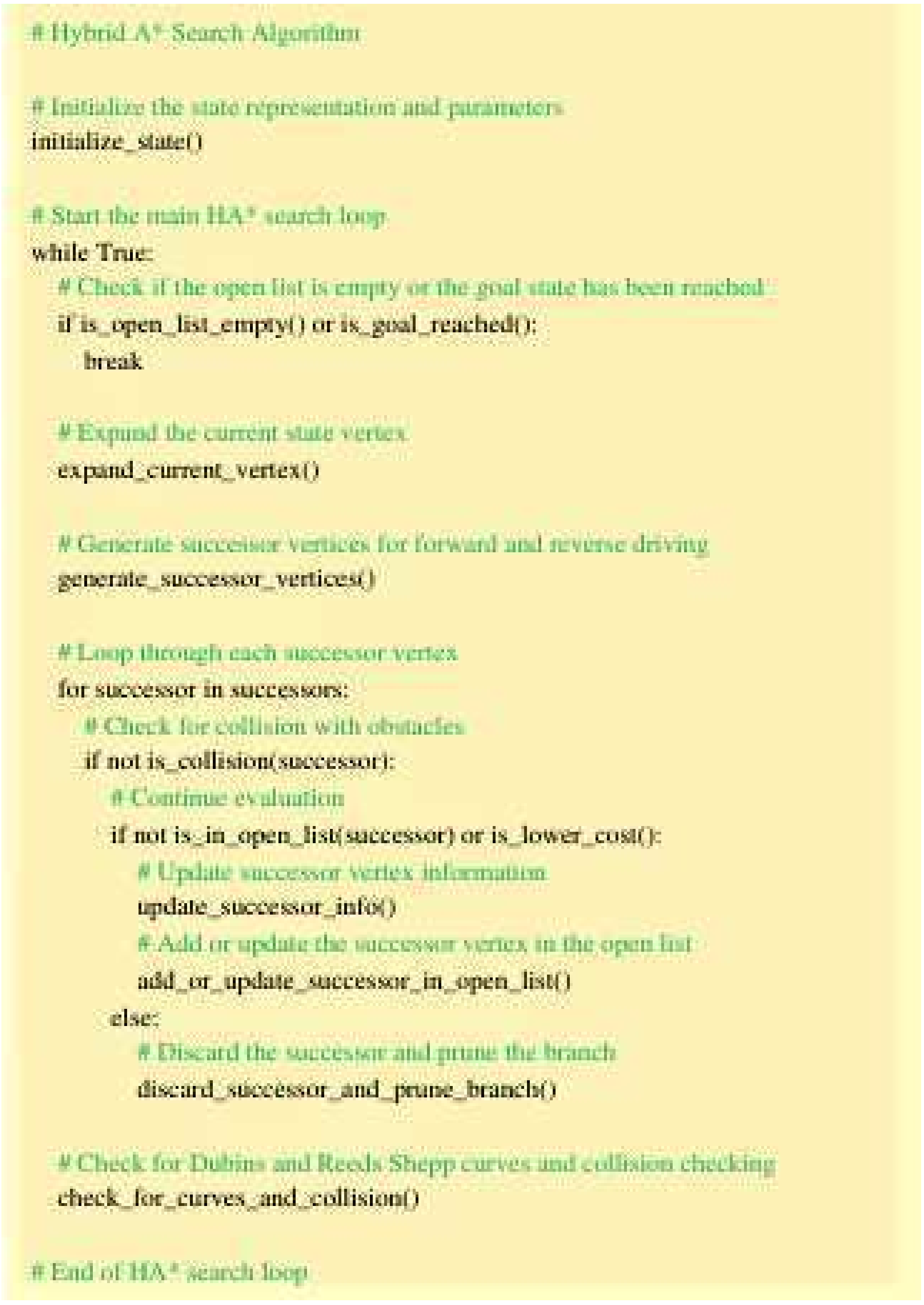
2. *Heuristic Estimation*: To aid in the search process, the algorithm employs a heuristic function called $CALC\ DISTANCE\ HEURISTIC$. This function calculates an estimate of the cost from the goal for all non-obstacle nodes on the map. It does so by expanding nodes starting from the goal and assigning a cost based on the distance to the goal. This information is used to guide the search.

3. *Priority Queue (PQ) Initialization*: A priority queue (PQ) is initialized to efficiently explore states. The start node is inserted into the openList.

4. *Search Loop*: The core of the algorithm is a search loop that continues until a path is found or determined to be infeasible. In each iteration of the loop: - The node with the lowest combined cost is 'popped' from the PQ, and its cost and index are obtained. - If

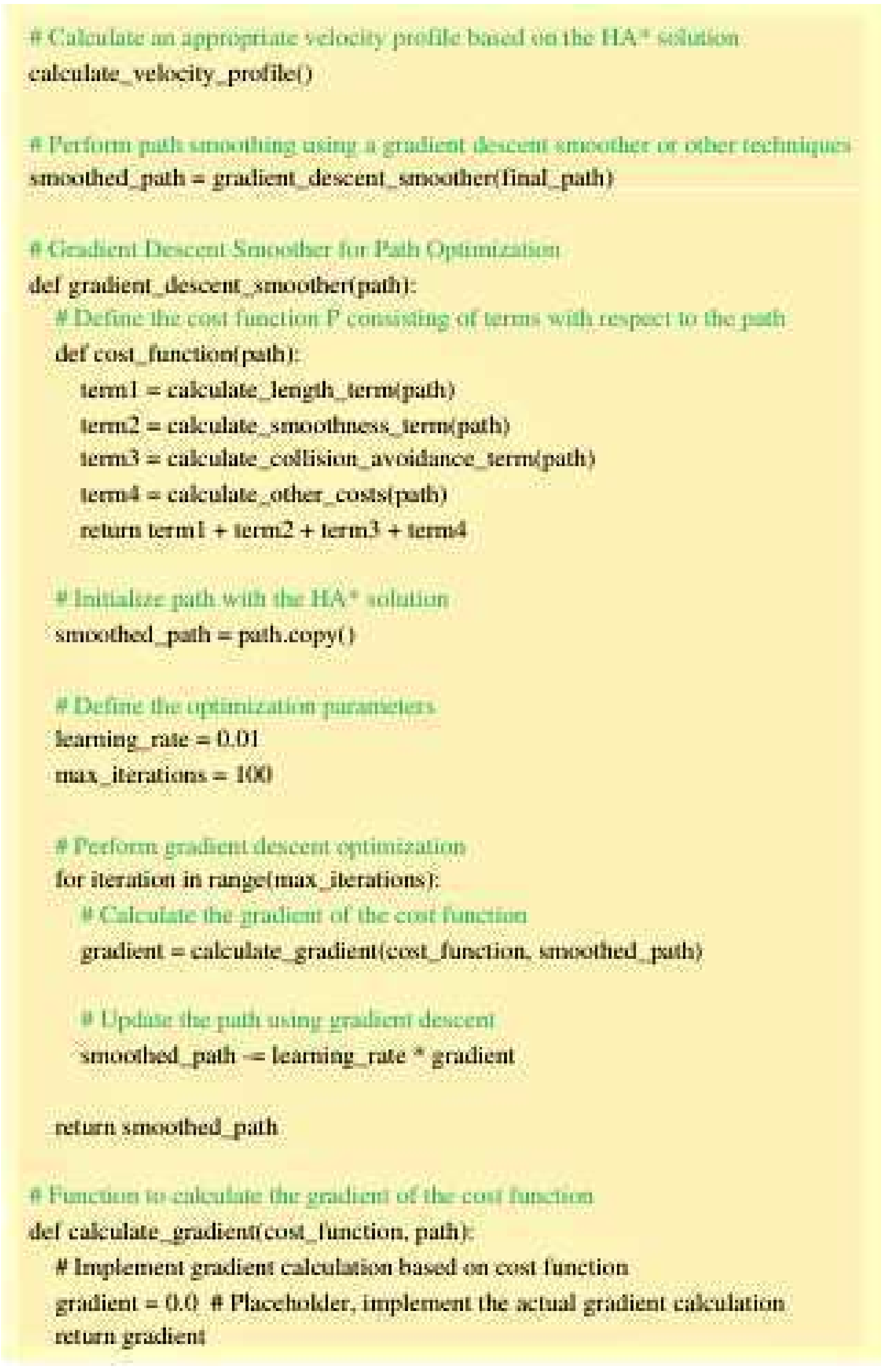
the popped node's index is already in the openList, it is removed from there and added to the ClosedList. - The algorithm identifies a promising node by exploring whether a feasible path to the goal can be found. This is done using the $UPDATE\ NODE\ WITH\ ANALYTIC\ EXPANSION$ function, which tracks whether a feasible path can be constructed from the current node. If successful, it calculates the cost of the path considering direction changes and other factors. - If a promising node is found, it is treated as a potential solution. If not, the search continues. - Neighbor Exploration: The algorithm explores neighboring nodes in the continuous space. The $GET\ NEIGHBORS$ function returns neighboring nodes, considering various steering angles to explore different directions. For each direction and steering angle, it calculates the next node and checks for collisions. It also computes the cost associated with changing direction and updates the node's cost based on the motion performed.

5. *Termination*: The search loop terminates when a path to the goal is found or when it's determined that no feasible path exists.



```
# Hybrid A* Search Algorithm

# Initialize the state representation and parameters
initialize_state()

# Start the main HA* search loop
while True:
    # Check if the open list is empty or the goal state has been reached
    if is_open_list_empty() or is_goal_reached():
        break

    # Expand the current state vertex
    expand_current_vertex()

    # Generate successor vertices for forward and reverse driving
    generate_successor_vertices()

    # Loop through each successor vertex
    for successor in successors:
        # Check for collision with obstacles
        if not is_collision(successor):
            # Continue evaluation
            if not is_in_open_list(successor) or is_lower_cost():
                # Update successor vertex information
                update_successor_info()
                # Add or update the successor vertex in the open list
                add_or_update_successor_in_open_list()
            else:
                # Discard the successor and prune the branch
                discard_successor_and_prune_branch()

    # Check for Dubins and Reeds Shepp curves and collision checking
    check_for_curves_and_collision()

# End of HA* search loop
```

Figure 4.3: Pseudocode of the Hybrid A* algorithm



```
# Calculate an appropriate velocity profile based on the HA* solution
calculate_velocity_profile()

# Perform path smoothing using a gradient descent smoother or other techniques
smoothed_path = gradient_descent_smoother(final_path)

# Gradient Descent Smoother for Path Optimization
def gradient_descent_smoother(path):
  # Define the cost function P consisting of terms with respect to the path
  def cost_function(path):
    term1 = calculate_length_term(path)
    term2 = calculate_smoothness_term(path)
    term3 = calculate_collision_avoidance_term(path)
    term4 = calculate_other_costs(path)
    return term1 + term2 + term3 + term4

  # Initialize path with the HA* solution
  smoothed_path = path.copy()

  # Define the optimization parameters
  learning_rate = 0.01
  max_iterations = 100

  # Perform gradient descent optimization
  for iteration in range(max_iterations):
    # Calculate the gradient of the cost function
    gradient = calculate_gradient(cost_function, smoothed_path)

    # Update the path using gradient descent
    smoothed_path -= learning_rate * gradient

  return smoothed_path

# Function to calculate the gradient of the cost function
def calculate_gradient(cost_function, path):
  # Implement gradient calculation based on cost function
  gradient = 0.0  # Placeholder, implement the actual gradient calculation
  return gradient
```

Figure 4.4: Pseudocode of the Hybrid A* algorithm

4.3 Reeds-Shepp curves

In 1990, Reeds and Shepp tackled a problem that appeared similar to the one Dubins addressed in the appendix. They devised a solution for calculating paths with an upper bound on curvature while considering that a car could move both forwards and backward. This was a significant advancement, as it allowed vehicles to reverse during path planning.The shortest paths for a car with a reverse gear, sometimes called the Reeds–Shepp car in honor of the mathematicians who first studied the problem, use only straightline segments and minimum-turning-radius arcs. Using the notation C for a minimum-turning-radius arc, C_a for an arc of angle a, S for a straight-line segment, and | for a cusp (a reversal of the linear velocity). The Reeds-Shepp solution aims to find a path with a maximum of 2 cusps (points where the curvature changes suddenly due to reversing). Among the numerous possible paths, the one with the minimum length is considered the solution. Like Dubins Curves, Reeds-Shepp curves are composed of curved and straight segments, but they accommodate the added complexity of reversing. Paths generated using Reeds-Shepp curves typically consist of at most five segments, following a pattern like CCSCC (C for curve and S for straight), depending on the specific requirements of the path. This advancement in path planning has been especially valuable in robotics and autonomous vehicle navigation, allowing for more versatile and efficient maneuvers. In the context of the

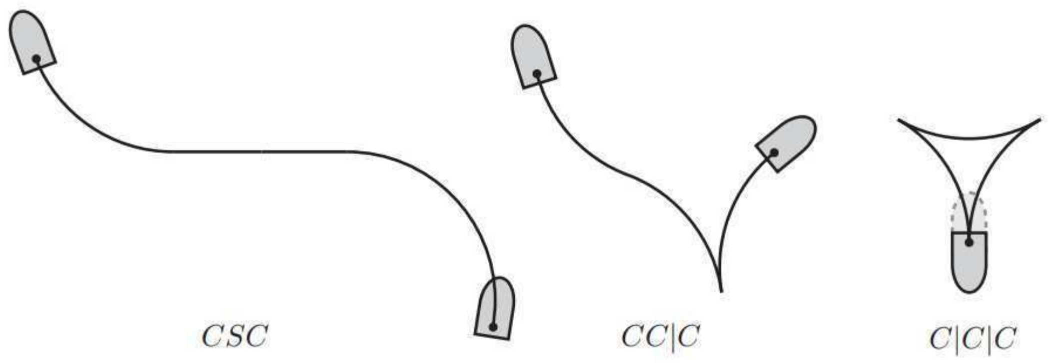


Figure 4.5: Examples of shortest paths for a car with a reverse gear.

Hybrid A* algorithm, the incorporation of Dubin or Reeds-Shepp curves plays a pivotal role in enhancing the efficiency and accuracy of path planning for autonomous vehicles. Dubin curves [**Appendix C**], characterized by their simplicity and limited maneuverability, are often employed when vehicle dynamics are relatively straightforward, making them a suitable choice for scenarios where turning radius and speed are constrained. On the other hand, Reeds-Shepp curves, with their ability to model more complex vehicle dynamics and account for reverse gear, are preferred in situations where precise path tracking, reverse maneuvers, and obstacle avoidance are paramount. By seamlessly integrating these curve models into the Hybrid A* algorithm, it becomes capable of generating optimal paths that adhere to the vehicle's kinematic constraints while navigating through diverse environments, making it a valuable tool in the realm of autonomous navigation and robotics.

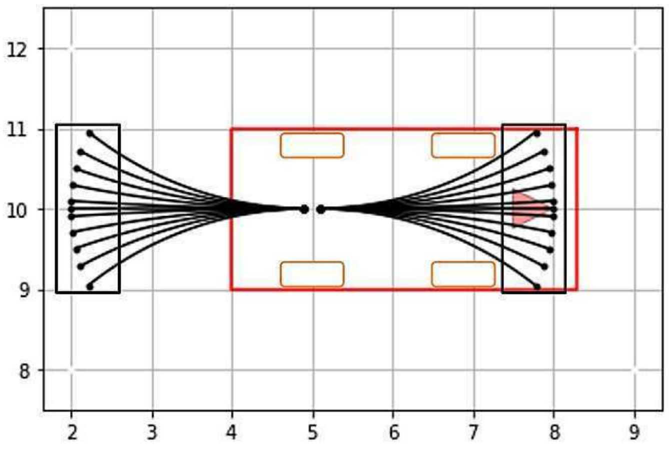
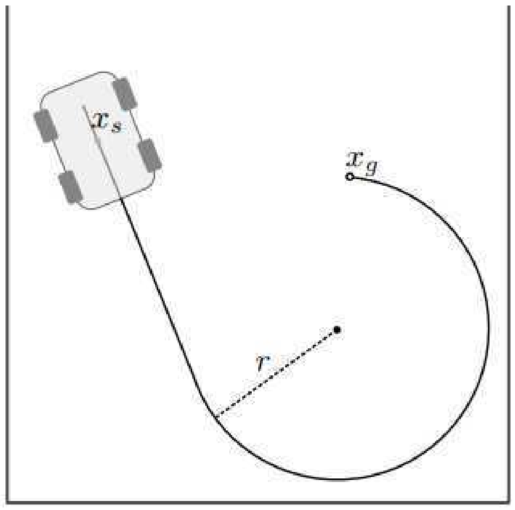


Figure 4.6: Successor selection in continuous domain when robot can move backward

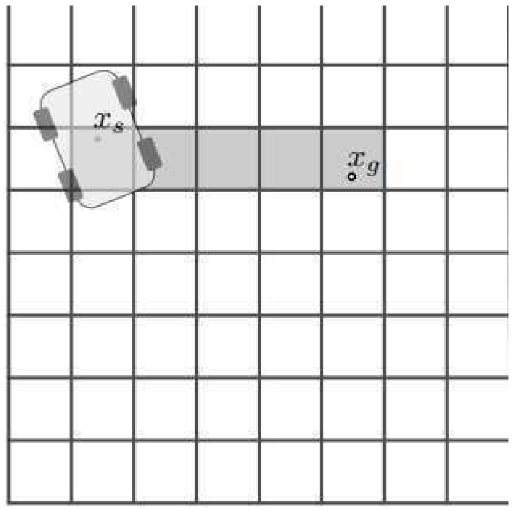
4.4 Heuristics

In order to identify the most efficient path, it's crucial for the search process to follow a systematic approach. The primary distinguishing factor among various search algorithms lies in how they expand vertices. To prevent the wasteful exploration of unpromising areas within a graph, the search should be as well-informed as possible, expanding only those nodes with the potential to belong to the optimal path. When the search incorporates information that results in the omission of expanding a particular node, and consequently fails to find the optimal path, it compromises admissibility. An ideal heuristic would furnish the true cost associated with a vertex. Heuristics serve as a valuable tool for approximating solutions, particularly in cases where computational power limitations necessitate a substantial reduction in the search space. Given that a finite amount of time allows for only a finite number of computations, this constraint is not surprising but remains a significant impediment for problems that exhibit exponential growth in search depth. During the exploration of a graph, the search must make decisions about which vertex to expand and which edge to follow. Information aimed at addressing this question is referred to as a heuristic. Such a heuristic may rely on cost estimates between the current vertex and the goal vertex. A heuristic essentially functions as a tool that provides essential information, expediting the algorithm's convergence toward the desired goal. However, it's worth noting that only an admissible heuristic can lead to optimal outcomes. While the ultimate aim is to generate viable solutions that approach optimality, it is crucial to leverage A*'s nature as an informed search algorithm. This involves implementing heuristics that expedite the algorithm's convergence towards a solution. In the case of HA*, it utilizes estimates from two distinct heuristics. Both of these heuristics are admissible, and for any given state, HA* selects the maximum value between them. These two heuristics address very different aspects of the problem.

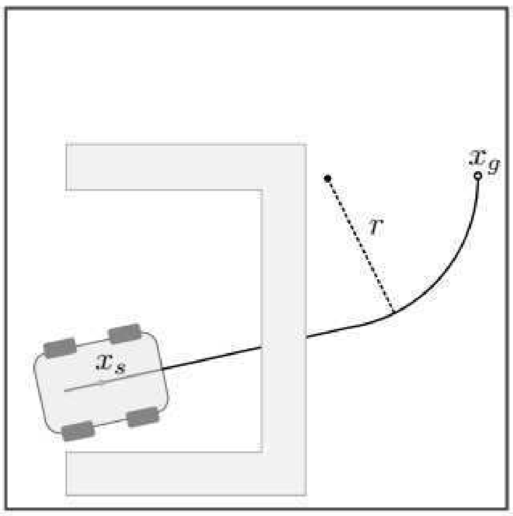
Constrained Heuristic (Accounting for Vehicle Characteristics): The constrained



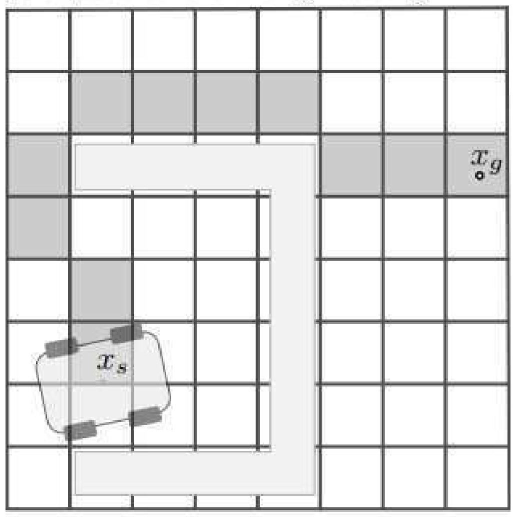
(a) The constrained heuristic accounting for the goal heading.



(b) The unconstrained heuristic heavily underestimating the cost of the path, due to the wrong heading.



(c) The constrained heuristic heavily underestimating the cost of the path, due to ignoring obstacles.



(d) The unconstrained heuristic accounting for obstacles and dead ends.

Figure 4.7: An evaluation of the advantages offered by both the constrained and unconstrained heuristics.

heuristic takes into consideration the unique characteristics of the vehicle while disregarding environmental factors. It typically employs Dubins or Reeds-Shepp curves, introduced before, as suitable candidates. These curves represent paths of minimal length while adhering to upper bound curvature constraints, both for forward and forward/backward driving cars, respectively. This heuristic considers the current heading and turning radius, ensuring that the vehicle approaches the goal with the correct orientation, especially when it nears the goal. For efficiency, this heuristic can be precomputed and stored in a lookup table since it does not involve obstacles and, thus, doesn't rely on environmental information. However, it primarily improves performance and doesn't significantly impact the quality of the solution, so a lookup table may not always be implemented. Given that both Dubins and Reeds-Shepp curves represent minimal paths, this heuristic is clearly admissible.

Unconstrained Heuristic (Focusing on Obstacles):
The unconstrained heuristic, in contrast, disregards the vehicle's characteristics and concentrates solely on obstacle avoidance. It estimates the shortest distance between the goal node and the currently expanded vertex. This distance is calculated using standard A* search in two dimensions (x, y position) with an Euclidean distance heuristic. Notably, the two-dimensional A* search employs the current vertex as the goal and the goal vertex of the HA* search as the starting point. This is advantageous because the closed list of the A* search stores all shortest distances g(x) to the goal and can thus serve as a lookup table, eliminating the need for a new search as HA* progresses. The unconstrained heuristic guides the vehicle away from dead ends and aids in navigating around U-shaped obstacles. Since HA* can potentially reach any point within a cell, the unconstrained heuristic needs to be adjusted by the absolute difference between the continuous coordinates of the current and goal vertices.

4.5 Path smoothing

The paths generated by the hybrid-state A* algorithm often fall short of optimality and require further enhancement. In our empirical observations, we have found that these paths are indeed navigable, but they may exhibit unnatural swerves, demanding excessive steering input. Consequently, we have devised a two-stage optimization process to refine the output of the hybrid-state A* algorithm. In the initial stage of optimization, we formulate a non-linear optimization program that focuses on the coordinates of the path's vertices. This optimization aims to improve both the length and smoothness of the solution. We employ the conjugate-gradient (CG) descent technique, known for its efficiency in numerical optimization. The primary goal of this first optimization stage is to enhance the smoothness of the path by adjusting the positions of its vertices while preserving the path's original discretization. However, the resulting path from the first stage may still have a coarse discretization, making it unsuitable for precise control of a physical vehicle (typically with a granularity of approximately 0.5 meters). To address this, we proceed to a second stage, where we perform non-parametric interpolation on the output of the first stage. This interpolation is accomplished through another iteration of the conjugate-gradient method. As a result, the interpolated paths possess a higher-resolution discretization, typically in the range of 5 to 10 centimeters, making them well-suited for the smooth and precise control of the robot.

To achieve this objective, a gradient descent smoother can be employed, which seeks to minimize P. P is comprised of the following four terms, and the minimization is carried out with respect to the path.

$$P = P_{obs} + P_{cur} + P_{smo} + P_{vor}; \tag{4.1}$$

Further elaboration on these four terms is provided in the description, with each term being explained in the context of its particular purpose.

Obstacle Term

$$P_{obs} = w_{obs} \sum_{i=1}^{N} \sigma_{obs}(|x_i - o_i| - d_{obs}); \tag{4.2}$$

This term imposes penalties for collisions with obstacles. It applies to all vertices xi within a range defined as $|x_i - o_i| \leq d_{obs}$, with the cost P_{vor} being calculated based on the distance to the nearest obstacle. Here, x_i represents the x, y-position of a path vertex, while o_i signifies the location of the nearest obstacle to x_i. The threshold d_{obs} determines the maximum distance over which obstacles can affect the path's cost. To impose a more significant penalty as the path approaches obstacles, σ_{obs} is employed as a quadratic penalty function. The weight w_{obs} is utilized to modulate the extent of its impact on the path's alterations.

Curvature Term

$$P_{cur} = w_{cur} \sum_{i=1}^{N-1} \sigma_{cur}(\frac{\Delta\Phi_i}{|\Delta x_i|} - k_{max}); \tag{4.3}$$

In order to ensure driveability the curvature term upper-bounds the instantaneous curvature of the path at every vertex. It is defined when the term inside parenthesis is > 0. The displacement vector at the vertex x_i is defined as $\Delta x_i = x_i x_{i_1}$. The change in tangential angle at a vertex can be expressed by $\Delta\Phi_i$. The maximum allowable curvature is denoted by k_{max}. Deviations from the maximum allowable curvature are penalized with a quadratic penalty function σ_{cur}. The curvature weight w_{cur} controls the impact on the change of the path.

Smoothness Term

$$P_{smo} = w_{smo} \sum_{i=1}^{N} (\Delta x_{i+1} - \Delta x_i)^2; \tag{4.4}$$

The smoothness term assesses the displacement vectors between vertices, effectively assigning a cost to vertices that exhibit uneven spacing or abrupt changes in direction. The parameter

w_{smo} signifies the weight of the smoothness term and, consequently, its influence on altering the path.

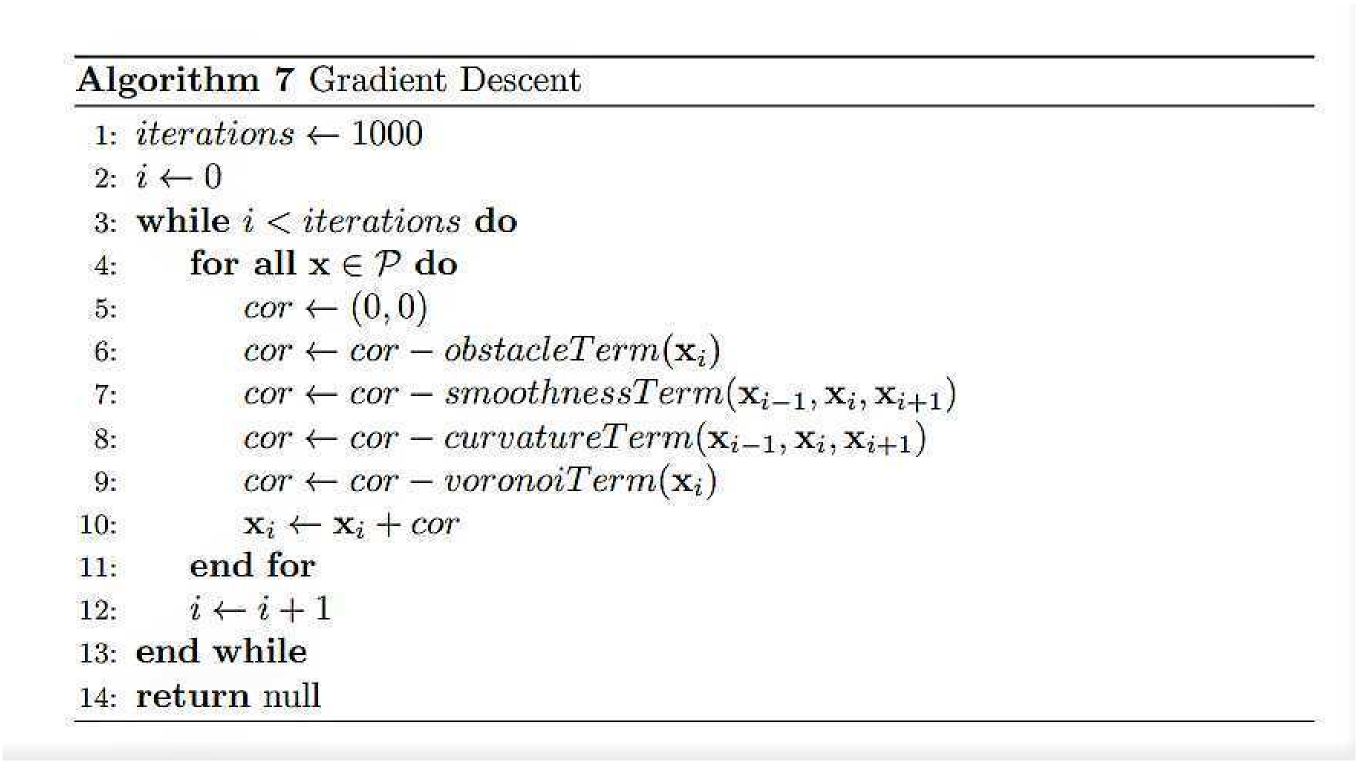
Voronoi Term

$$P_{vor} = w_{vor} \sum_{i=1}^{N} (\frac{\alpha}{\alpha + d_{obs}(x,y)})(\frac{d_{vor}(x,y)}{d_{obs} + d_{vor}(x,y)})(\frac{(d_{obs}(x,y) - d_{vor})^2}{d_{vor}^2}); \tag{4.5}$$

This term influences the path to steer clear of obstacles. When dobs is less than or equal to d_{vor}, the cost P_{vor} comes into play, taking into account the node's position within the Voronoi field. Here, d_{obs} represents the positive distance to the nearest obstacle, d_{edg} is the positive distance to the closest GVD edge, and d_{vor} denotes the maximum distance at which obstacles influence the Voronoi potential. The parameter α, which is greater than zero, governs the rate at which the field diminishes, while w_{vor}, the Voronoi weight, determines its impact on the path.

Gradient Descent

The gradient descent technique is an optimization algorithm that leverages the function's gradient to seek a local minimum. It advances iteratively with step sizes proportional to the negative gradient: $\Delta x = -\bigtriangledown f(x)$. Instead of the common practice of using the absolute gradient value as a stopping criterion, usually a predetermined number of iterations is adopted to maintain consistent runtime efficiency.



Algorithm 7 Gradient Descent

```
1: $iterations \leftarrow 1000$
2: $i \leftarrow 0$
3: while $i < iterations$ do
4:     for all $\mathbf{x} \in \mathcal{P}$ do
5:         $cor \leftarrow (0,0)$
6:         $cor \leftarrow cor - obstacleTerm(\mathbf{x}_i)$
7:         $cor \leftarrow cor - smoothnessTerm(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{x}_{i+1})$
8:         $cor \leftarrow cor - curvatureTerm(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{x}_{i+1})$
9:         $cor \leftarrow cor - voronoiTerm(\mathbf{x}_i)$
10:        $\mathbf{x}_i \leftarrow \mathbf{x}_i + cor$
11:    end for
12:    $i \leftarrow i + 1$
13: end while
14: return null
```

Figure 4.8: Pseudocode of the Gradient Descent.

Chapter 5

Applications and Case Studies

Unfortunately, due to time constraints and certain inconveniences encountered during my internship with the E80 Group, we were unable to conduct testing of the Hybrid A* algorithm on the Dingo. However, to provide valuable insights and results regarding the application of this algorithm, we have included some illustrative cases for reference. Therefore, in the final chapters of this thesis, we transition from the theoretical underpinnings of the Hybrid A* algorithm to its practical applications and case studies. This chapter is subdivided into four comprehensive papers, each offering a deeper understanding of how Hybrid A* can be harnessed and enhanced in various real-world scenarios and simulations.

5.1 Autonomous Navigation in Unstructured Environments

5.1.1 Off-Road Autonomous Vehicles

In the realm of off-road autonomous vehicles, Hybrid A* has emerged as a game-changer. We delve into the successful deployment of Hybrid A* in rugged terrains such as deserts, forests, and agricultural fields. Through detailed case studies, we illustrate how this algorithm empowers robots and vehicles to navigate through challenging landscapes while ensuring safety and efficiency. The algorithm's adaptability to diverse terrains and its ability to find feasible paths amidst uneven topography are highlighted. In the paper *"Application of Hybrid A* to an Autonomous Mobile Robot for Path Planning in Unstructured Outdoor Environments"* [**15**], the authors presented the HA* applied to a nonholonomic mobile outdoor robot in order to plan near optimal paths in mostly unknown and potentially intricate environments. The paper presents an upgraded version of the HA* algorithm in order to address the specific challenges of the real-world environment. In fact, the authors considered and handles some of the limitations of the scenario and incorporate them into heuristics for the algorithm. The main challenges addressed are:

- Handling Waypoints Sequences
- Terrain Characteristics
- Waypoints Beyond the Local Map
- Wall Following for Large Obstacles

This paper describes the application of the Hybrid A* algorithm in path planning for autonomous mobile robots in unstructured outdoor environments. The algorithm considers vehicle kinematic constraints and surface conditions, ensuring generated paths are feasible and drivable. Notably, it can explicitly plan through multiple waypoints, guaranteeing drivability, and employs cost functions and heuristics to accommodate constraints like on-road preferences. Additionally, the algorithm efficiently handles waypoints lying beyond the local map using suitable heuristics, enhancing its adaptability to complex real-world scenarios. In 5.1, the results of

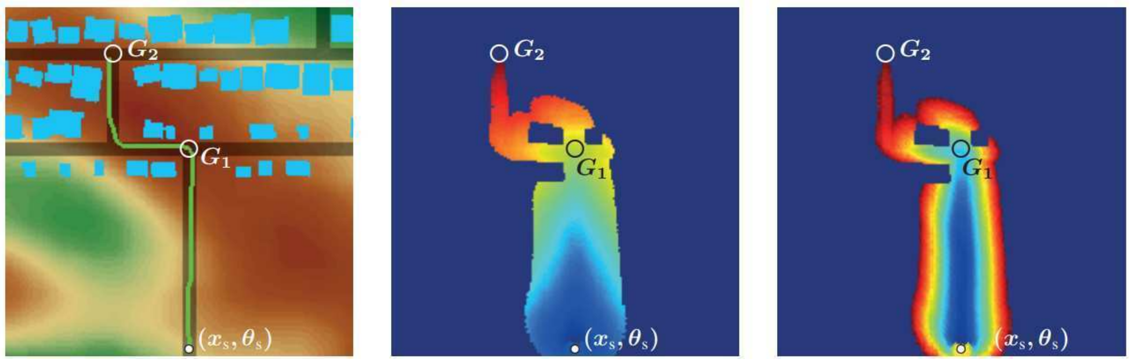


Figure 5.1: The path planned within an urban setting (left), actual minimum cost $min\theta$ g (center), and predicted minimum cost $min\theta$ f (right).

the planning process in an urban environment are shown. The left figure displays the generated path, which remains on the road thanks to the incorporation of an off-road penalty in the path costs, avoiding shortcuts through off-road areas. Furthermore, the Hybrid A* algorithm prioritizes the road network exploration, as evidenced by Figure 5's center, illustrating the actual costs (g), and the right, illustrating the predicted costs (f). The rightmost figure clearly demonstrates that the cell expansion during the path search to reach G2 is guided through the goal region G1 of the first waypoint.

5.1.2 Indoor Environments

Indoor environments pose unique challenges for autonomous navigation due to constrained spaces and dynamic obstacles. In this subsection, we explore how Hybrid A* is utilized in scenarios such as warehouse automation and indoor robotics. Practical examples and simulations demonstrate how the algorithm aids in path planning within tight confines, enabling robots to operate efficiently in industrial and home environments. In the study *"Improved Analytic Expansions in Hybrid A-Star Path Planning for Non-Holonomic Robots"* [**17**], the Hybrid A* algorithm for non-holonomic robots in indoor environments is explored. The investigation uncovers issues in different algorithm phases, such as RS curves being generated too close to obstacles and routes containing unnecessary turns. To address these challenges, the study proposes generating multiple RS curves with varying curvature values and using a cost function to select the safest path among them. Additionally, a fine-tuning approach is suggested to reduce unnecessary turns without complex smoothing techniques. These improvements enhance the algorithm's safety performance for indoor robot applications. Experimental demonstrations validate the effectiveness of the proposed method in simulations, resulting in smoother routes with lower collision risks and reduced turning points compared to conventional methods. Although the computation time increases moderately, the study highlights the potential of Hybrid A* in indoor autonomous driving scenarios, where safety and efficiency are paramount. In simulations of the Hybrid A* algorithm in indoor environments, issues arose with non-smooth segments in the path generated by the forward search (blue lines) and the proximity of RS path results (green lines) to walls, particularly at corners. This proximity raised concerns about potential collisions due to measurement errors or motor control errors. To address these problems, the study aims to enhance the algorithm's analytic expansions by leveraging a unique property of RS curves. Achieving a safer route for a car-like robot involves adjusting the curvature value. Multiple paths are generated with varying curvatures, and the selection of a safer path from the start to the goal state is crucial. To evaluate each path, an objective function is proposed, comprising two cost functions.

$$G = \sigma_1 \cdot v + \sigma_2 \cdot m; \tag{5.1}$$

The first function, represented by 'v,' calculates the cost of the Voronoi field, emphasizing safer distances from obstacles. The second function, denoted as 'm,' assesses the cost of movement along the RS path, considering factors like path length, steering angles, and steer switching. By combining these functions with appropriate weights (σ_1 and σ_2), a balance is struck between selecting a safer but potentially longer path or a faster but riskier one for the car-like robot.

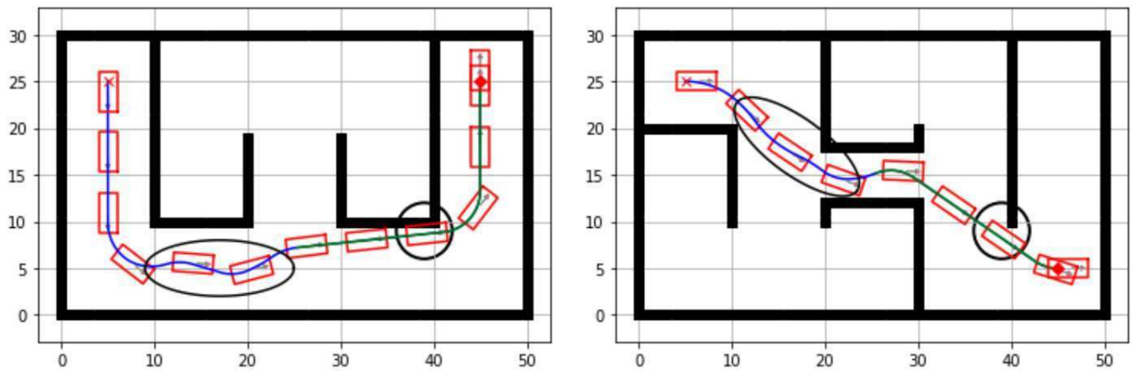


Figure 5.2: Original Hybrid A-star path planning in simulations. The black ellipses show the critical part of the generated RS curves.

The relative values of σ_1 and σ_2 determine the robot's preference between safety and speed. Following adjustments to the motion primitives without exhaustive search, they determined that a value of 1.5 for the motion primitive yielded satisfactory results. As depicted in 5.3, the

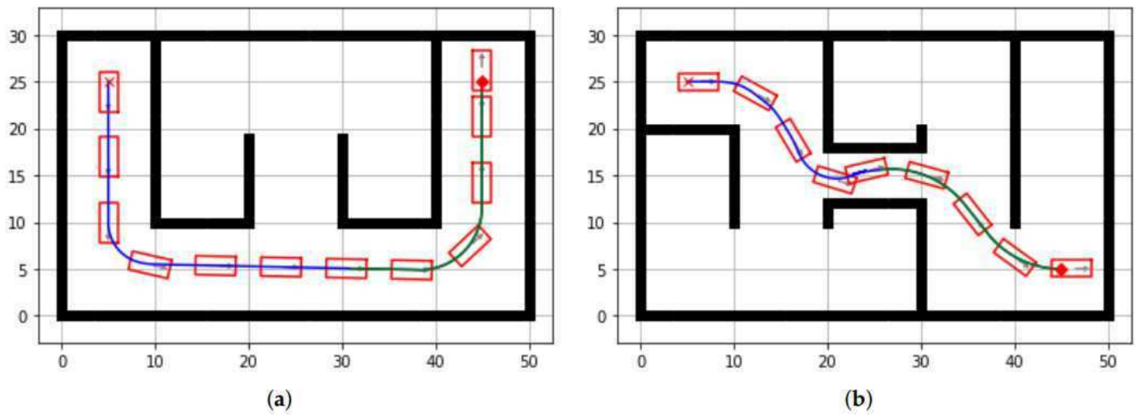


Figure 5.3: Tuning the improved hybrid A-star path planning.

tuning process has notably enhanced the algorithm's performance by ensuring a safe distance from walls and eliminating unnecessary swerves in the path.

5.2 Safety and efficiency

The academic paper *"Improved Hybrid A-Star Algorithm for Path Planning in Autonomous Parking System Based on Multi-Stage Dynamic Optimization"* [**19**] focused on enhancing path planning in autonomous parking systems. It introduces the popularity of autonomous parking and the use of the Hybrid A-star algorithm for path planning due to its simplicity and practicality. The paper proposes two significant improvements: a safety-enhanced design that considers path safety by incorporating Voronoi field potential and an efficiency-enhanced design that employs multi-stage dynamic optimization. Simulation experiments confirm that these enhancements result in safer paths, with greater distance from obstacles, and significantly improve search efficiency in terms of time and space. This modified version of the Hybird A* algorithm aims to achieve two fundemental conditions: safety and efficiency. To verify the proposed improved algorithm, they preset a scenario illustrated in 5.4. The parking area features two entrances. The black lines represent the parking lot's perimeters, while the orange lines mark the boundaries of the accessible parking spaces. All of these lines are regarded as obstacles that the vehicle must steer clear of to prevent collisions.

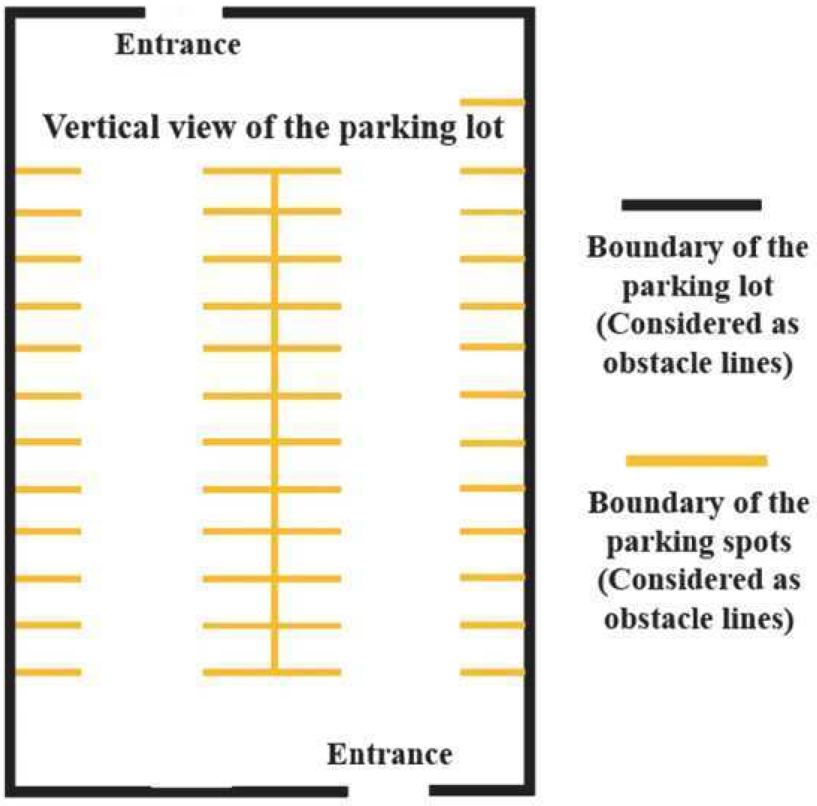

Entrance
Vertical view of the parking lot
Entrance
Boundary of the parking lot (Considered as obstacle lines)
Boundary of the parking spots (Considered as obstacle lines)

Figure 5.4: Layout of the parking lot.

Safety-enhanced Design

The primary concern is to improve the safety of the generated trajectories. The existing algorithm is capable of producing smooth paths that satisfy the vehicle's non-holonomic constraints but may not guarantee a safe distance from obstacles. To address this issue, the authors introduce a safety term based on the Voronoi field, which considers the trade-off between path length and proximity to obstacles. The Voronoi field is defined mathematically and offers several advantages, including scalability and ease of navigation through narrow openings. The overall cost function is then modified to incorporate this safety term alongside motion and heuristic costs. This safety-enhanced design aims to produce safer trajectories in complex environments with multiple obstacles, without compromising maneuverability in narrow spaces, such as parking lots.

Efficiency-enhanced Design

The conventional hybrid A-star algorithm, while effective, can be time-consuming and computationally intensive when dealing with such obstacles. To address this, the authors propose a multi-stage planning framework based on the hybrid A-star algorithm, incorporating safety enhancements. In this approach, parking path planning is divided into two stages. The first stage focuses on planning a path from the entrance of the parking lot to a midpoint near the target parking spot, which is relatively easier as it involves mainly road driving without many obstacles. The second stage deals with planning a more complex path from the midpoint to the target parking spot while avoiding intersections with surrounding border lines and adhering to the vehicle's non-holonomic constraints. To enable this two-stage approach, an available list of middle points around the target parking spot is constructed. These middle points are selected dynamically through an optimization framework during the first planning stage. Once the first stage is completed, the second stage becomes more efficient since it is guaranteed to find a collision-free path from the middle point to the target parking spot. The method introduces a neighboring interest area around the target state, and the state space is discretized with resolution parameters to create a discrete state set. Then, collision-free RS (Reeds-Shepp) paths from each state in the set to the target state are checked, and the states that can be connected by such paths are added to the admissible set. The dynamic selection of the best middle point is done through an optimization scheme, where a cost function considers four cost subitems, each with specific objectives. These subitems include measures of Voronoi field potential, the gradient of this potential with respect to the heading angle, the integral of potential along the RS path, and the difference between heading angles. These subitems guide the selection of the middle point to stay away from obstacles, align with the direction of the drivable road, find safer paths, and accelerate the first planning stage. The method offers flexibility for various cost functions to be used in practical engineering applications under this dynamically optimized multi-stage planning framework, allowing for customization based on specific intentions and requirements. In essence, this approach aims to make parking path planning more efficient and safe by breaking it into two stages, dynamically selecting intermediate points, and using a cost-based optimization framework to guide the planning process.

Testing

Initially, the safety-enhanced design is implemented, prioritizing path safety in autonomous driving scenarios. Subsequently, the efficiency-enhanced design is deployed to improve the efficiency of path planning. The results of both these design approaches are illustrated in the accompanying figures.

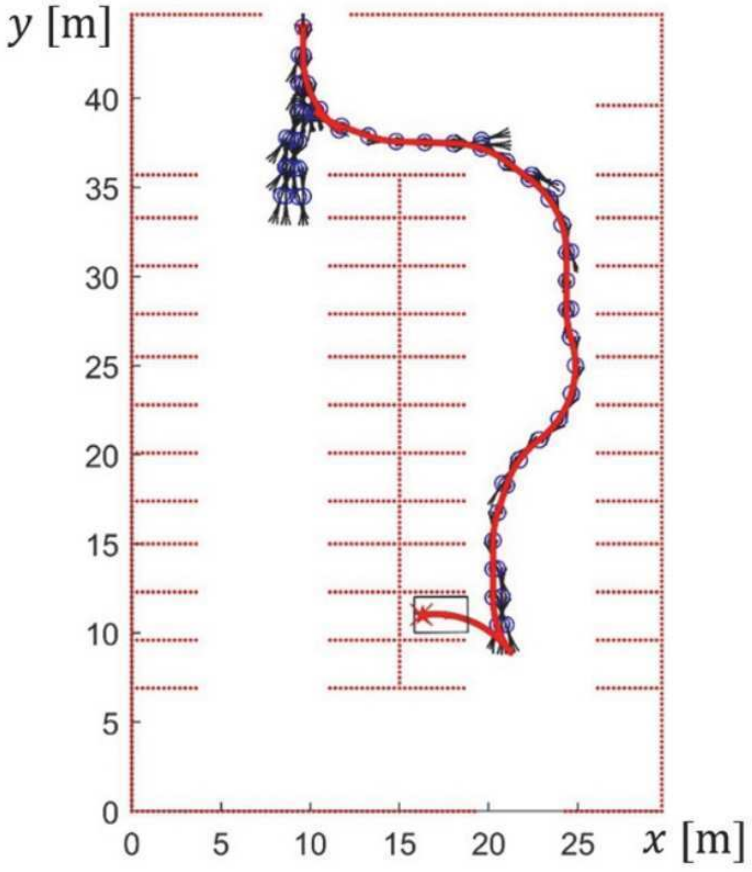
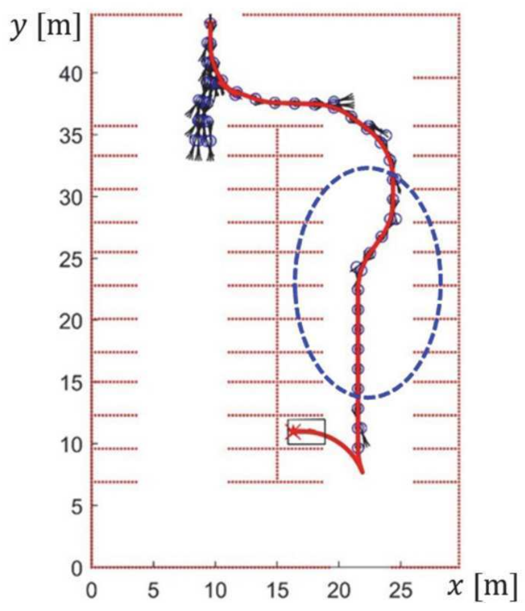
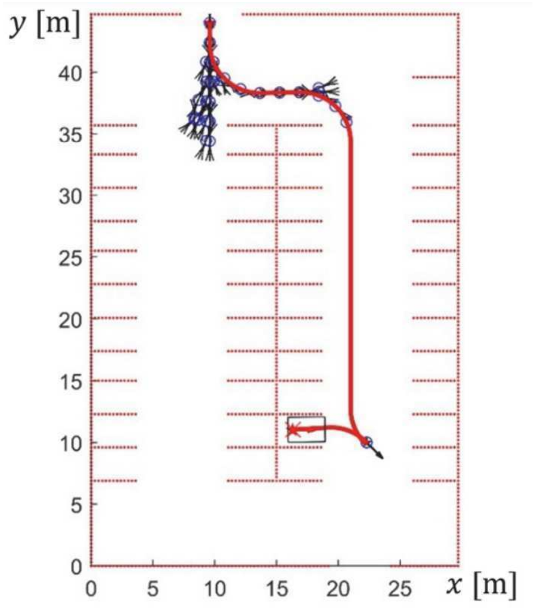


Figure 5.5: Searching results of conventional hybrid A-star algorithm.

In contrast to the conventional hybrid A-star algorithm, the enhanced algorithm they propose not only produces a safer path, maintaining a greater distance from obstacles, but also notably reduces the complexity of the search process, resulting in improved efficiency. While deploying the improved algorithm does require some additional pre-processing work, it's worth noting that these efforts can be leveraged repeatedly for subsequent use.



(a) Searching results of the improved hybrid A-star algorithm with only safety-enhanced design.



(b) Searching results of the improved hybrid A-star algorithm with both safety-enhanced design and efficiency-enhanced design.

Figure 5.6: Results of both enhancing designs

5.3 Multi-robot system

Multi-robot path planning is a challenging yet crucial aspect of robotics and autonomous systems, where multiple robots must navigate through complex environments while avoiding collisions and optimizing their paths. Leveraging the Hybrid A* algorithm has proven to be highly successful in addressing this intricate task. This advanced algorithm not only ensures the efficient planning of individual robot trajectories but also orchestrates their coordinated movements, allowing multiple robots to navigate seamlessly through unstructured environments while adhering to kinematic constraints, avoiding obstacles, and optimizing their collective paths. This breakthrough in multi-robot path planning showcases the versatility and effectiveness of the Hybrid A* algorithm in real-world applications, paving the way for enhanced robotic cooperation and efficiency in various domains. The article *"A divide-and-conquer control strategy with decentralized control barrier function for luggage trolley transportation by collaborative robots"* [**16**]focuses on addressing complex constraints in a transportation system consisting of a luggage trolley queue and two collaborative robots (a leader and a follower). To tackle this challenging task, a divide-and-conquer control strategy is proposed, aiming to manage various constraints separately. The system's complexity arises from the need to handle nonholonomic constraints due to the robots' differential structures while maintaining a stable formation between them. Traditional path planning algorithms like basic A* are unsuitable

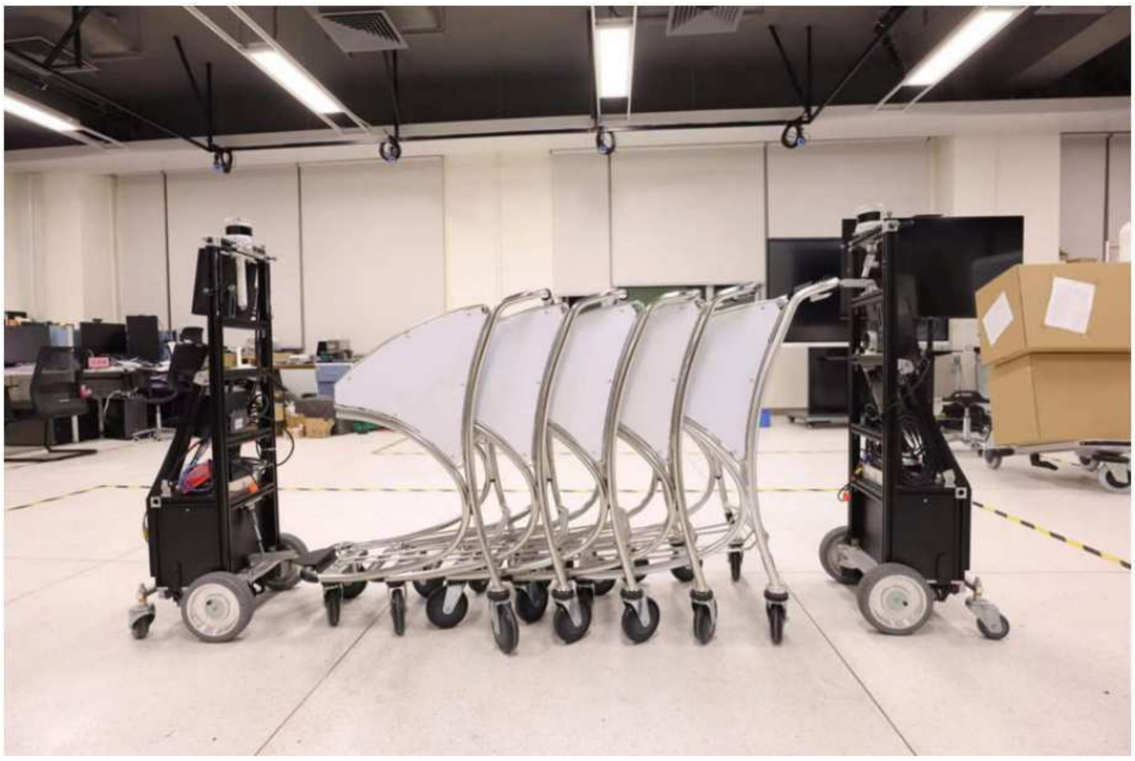


Figure 5.7: The luggage trolley transportation system

for this task, leading the authors to adopt the Hybrid A* algorithm for global path planning. The proposed strategy employs a hierarchical approach, involving a high-level global planner to generate smooth paths and a low-level controller to track these paths while preserving the formation. Extensive simulation and physical experiments confirm the effectiveness of this approach, demonstrating the ability to smoothly transport the luggage trolley queue while maintaining formation and adhering to nonholonomic constraints.

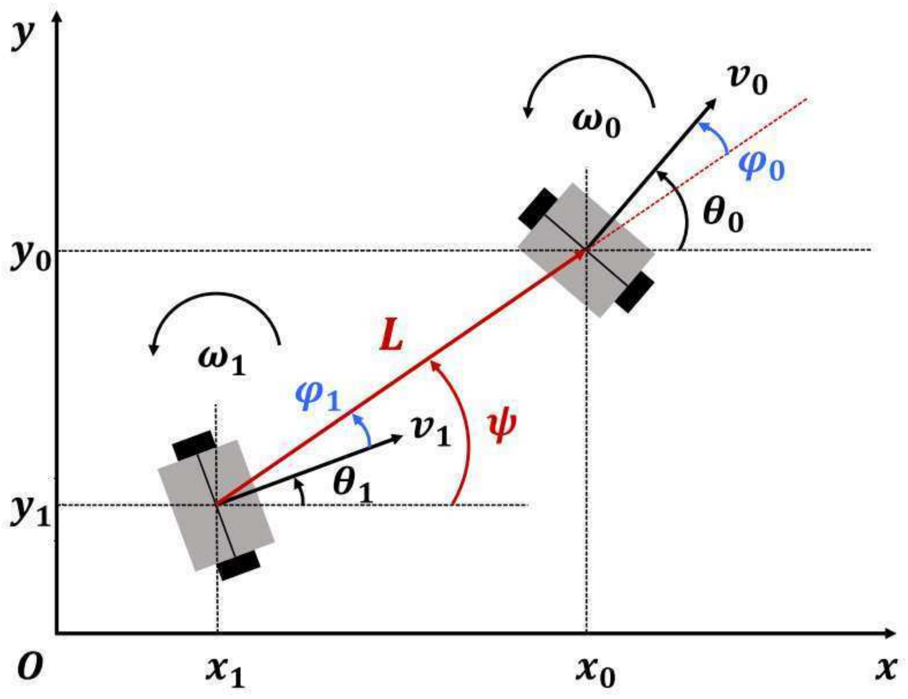


Figure 5.8: The kinematic model incorporates nonholonomic constraints, and the system comprises two differential robots, designated as 0 (leading) and 1 (following), respectively. Additionally, the illustration includes representations of the inter-robot distance (L), the system's yaw angle (ψ), and the steering angles of the robots relative to the system (ϕ_0 and ϕ_1).

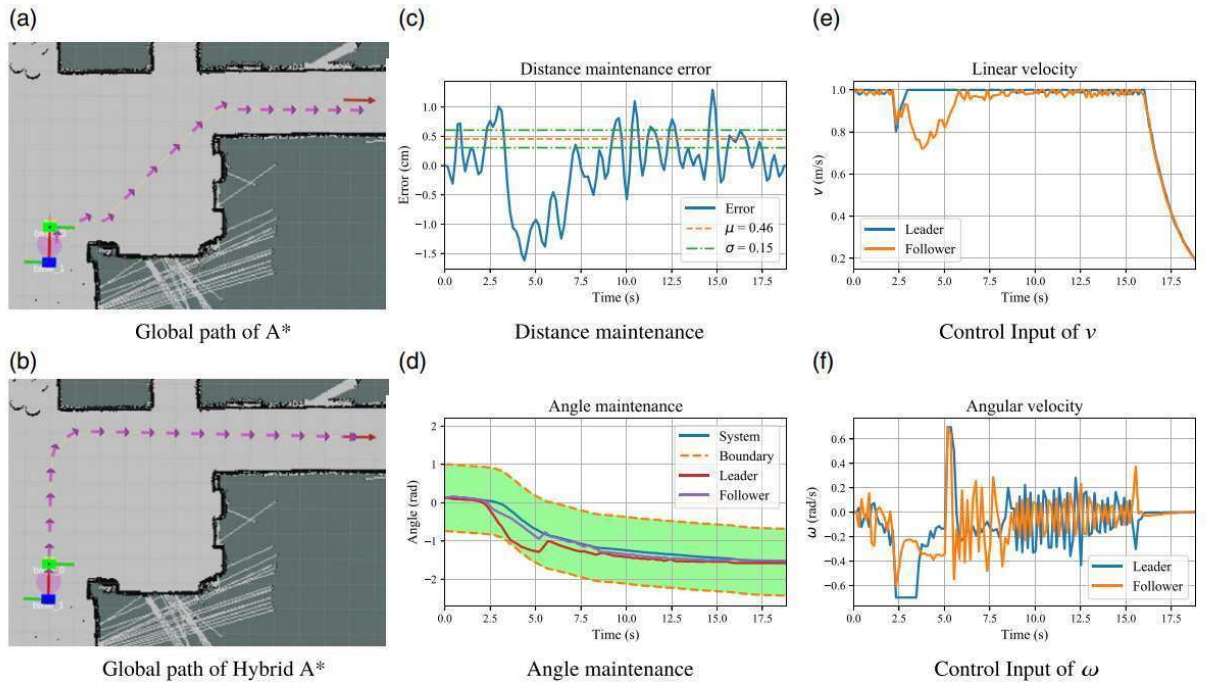


Figure 5.9: Results from the simulation experiments include distance maintenance statistics, represented by mean and standard deviation values, and angle maintenance, where the safe region is visually depicted with a boundary and shaded in green.

Chapter 6

Challenges and Future Directions

In this thesis we have explored a spectrum of state-of-the-art path planning algorithms with a particular emphasis on the Hybrid A* algorithm. While the research conducted thus far has provided valuable insights into path planning for non-holonomic robots operating in dynamic and unstructured environments, several noteworthy challenges and promising future directions have emerged. One of the key challenges lies in real-time implementation and optimization of path planning algorithms to ensure their practicality in dynamic, real-world scenarios. Additionally, enhancing the adaptability of these algorithms to varying environmental conditions and uncertainties remains a critical research area. Looking ahead, it is essential to consider the context of the E80 Group's future goal, which involves developing path planning algorithms for multi-robot systems. This endeavor introduces a unique set of challenges and difficulties. Coordination, communication, and collision avoidance among multiple robots will become paramount concerns. Designing algorithms that efficiently distribute tasks and ensure cooperation among robots while avoiding conflicts is a complex undertaking. Scalability and the ability to handle a growing number of robots in a seamless manner will also be a significant challenge. Moreover, the incorporation of machine learning and swarm intelligence techniques to improve the collective decision-making of multi-robot systems presents exciting opportunities and research avenues. Thus, the future of path planning in the E80 company and similar applications holds the potential for groundbreaking developments but necessitates addressing the complexities associated with multi-robot coordination and navigation in unstructured environments.

6.1 Limitations of the Hybrid A* Algorithm

The Hybrid A* algorithm, while a powerful tool for path planning in various scenarios, does have some limitations:

- **High Computational Demands**: Hybrid A* can be computationally expensive, particularly in environments with many obstacles or complex terrain. The discretization of the state space and the generation of a hybrid state graph can result in a large number of nodes to explore, leading to increased computation time.
- **Grid Resolution**: The quality of the generated path depends on the resolution of the grid used to discretize the environment. If the grid resolution is too coarse, the algorithm might miss narrow paths or fail to account for intricate obstacle shapes. On the other hand, increasing the resolution can significantly increase computational requirements.
- **Memory Consumption**: Storing the hybrid state graph can be memory-intensive, especially in large and cluttered environments. This can limit the algorithm's applicability on resource-constrained platforms.
- **Handling Dynamic Environments**: Hybrid A* assumes that the environment is static during planning. Adapting it to dynamic environments, where obstacles or conditions change over time, is a non-trivial challenge and often requires real-time updates and re-planning.
- **Smoothness of Paths**: The paths generated by Hybrid A* may not always be the smoothest, which can be a limitation for applications where path smoothness is crucial, such as in controlling certain types of robots or vehicles.
- **Limited Sensor Models**: The algorithm relies on accurate sensor data to build the state space representation. Inaccuracies or limitations in sensor data can lead to incorrect path planning outcomes.
- **Scalability**: While Hybrid A* works well for single-robot systems, extending it to multi-robot scenarios can be challenging, particularly in scenarios with high robot density and complex interactions.

Despite these limitations, Hybrid A* remains a valuable choice for path planning in many contexts. Addressing these limitations often involves a trade-off between computational complexity and path quality, and researchers continue to work on improving the algorithm's efficiency and adaptability to a wide range of real-world scenarios.

6.2 Future Research Opportunities

The field of path planning algorithms for autonomous robots is continuously evolving, presenting several exciting future opportunities:

- **Multi-Robot Coordination**: As multi-robot systems become more prevalent, developing efficient algorithms for cooperative path planning and coordination among multiple robots will be crucial. These algorithms will need to account for both inter-robot communication and collision avoidance.

- **Uncertainty Handling**: Path planning algorithms that can effectively handle uncertainty, whether it's due to sensor noise, environmental variability, or perception limitations, will be essential for robust autonomous navigation.

- **Human-Robot Interaction**: Developing path planning algorithms that consider human-robot interaction and take into account human intentions and safety preferences will be essential for robots coexisting with humans in shared spaces.

- **Energy Efficiency**: Energy-efficient path planning algorithms will be crucial for resource-constrained robots, such as drones and mobile robots, extending their operational capabilities and reducing environmental impact.

- **Machine Learning Integration**: The integration of machine learning techniques, particularly deep reinforcement learning and neural networks, can enhance path planning by allowing robots to learn and adapt to complex and dynamic environments. End-to-end learning for path planning and decision-making is an emerging area of research.

- **Autonomous Exploration**: Path planning for autonomous exploration in unknown or partially known environments, including underwater exploration, planetary exploration, and search and rescue missions, remains a significant research area.

- **Ethical Considerations**: Addressing ethical considerations and guidelines for path planning, especially in situations where robots must make decisions that involve ethical and moral implications, such as autonomous vehicles.

In summary, the future of path planning for autonomous robots holds immense promise, with opportunities ranging from advanced machine learning integration to addressing the complexities of multi-robot coordination and human interaction. These advancements will be instrumental in enabling robots to navigate and operate effectively in a wide range of environments and applications.

Appendix A

Ackermann steering geometry

The Ackermann steering geometry represents a specific configuration of mechanical linkages within the steering system of an automobile or any other vehicle. Its primary purpose is to address the challenge that arises when the wheels on the inner and outer sides of a turn must follow paths with differing radii. This innovative steering arrangement can be attributed to Georg Lankensperger, a skilled carriage builder from Munich, who first conceptualized it in 1816. Subsequently, Rudolph Ackermann (1764–1834), Lankensperger's representative in England, secured a patent for this design in 1818, specifically for horse-drawn carriages. Interestingly, there is historical speculation that Erasmus Darwin might have laid claim to an earlier version of this steering system dating back to 1758, motivated by a personal injury resulting from a carriage overturning. Consider a low-speed cornering manoeuvre, where all tyres are in pure rolling condition, and there is no vehicle sliding present. As the vehicle travels along a curved path, all four tyres follow unique trajectories around a shared turn centre, as defined by the blue arcs, in 1. The different curvature radii mean that to avoid sliding, the steering geometry must

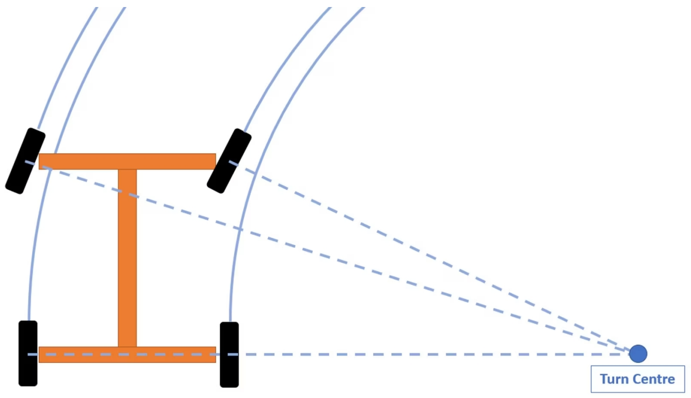


Figure 1: Simplified depiction of Ackermann Steering configuration.

steer the inside front tyre at a larger angle than the outside front. Ackermann Steering refers to the geometric configuration that allows both front wheels to be steered at the appropriate angle to avoid tyre sliding. Let's consider the optimal configuration. Upon turning, it becomes apparent that the inner wheel needs to pivot at a more pronounced angle compared to the outer wheel. In the accompanying diagram, in 2:

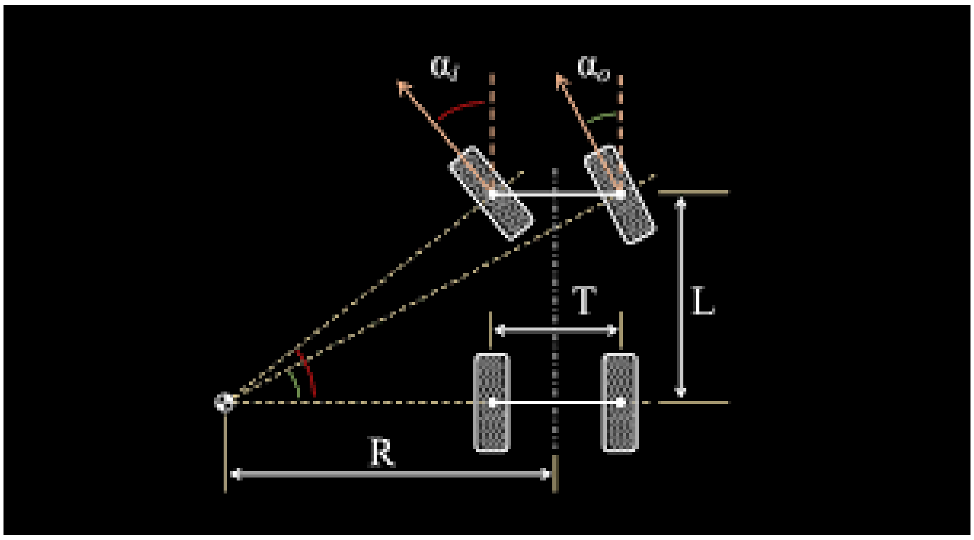


Figure 2: Scheme of the Ackermann model

- L represents the wheelbase of the vehicle, denoting the distance between its two axles.
- T signifies the track, representing the distance between the center lines of each tire.
- R stands for the turn radius as perceived from the vehicle's centerline.
- α_i corresponds to the angle of the inner wheel relative to the straight-ahead position.
- α_o represents the angle of the outer wheel relative to the straight-ahead position.

If we make certain assumptions, such as maintaining a constant speed (neglecting weight transfer and external forces), disregarding body roll or suspension effects, and considering only the front wheels for steering, we can employ basic trigonometry to determine the optimal wheel angles:

$$\alpha_i = tan^{-1}(\frac{L}{R - \frac{T}{2}}); \tag{1}$$

$$\alpha_o = tan^{-1}(\frac{L}{R + \frac{T}{2}}); \tag{2}$$

Ackerman steering solves most of the problems of turntable steering: The space required (fore-and-aft travel) by each wheel is significantly reduced, and the moment arm transmiting back imperfections in the road is reduced.

Appendix B

D* Algorithm

It was originally developed by Anthony Stentz at Carnegie Mellon University. D* stands for "Dynamic A*," and it is an improvement upon the A* algorithm, which is also used for pathfinding. The key feature of D* is its ability to efficiently update the path when there are changes in the environment or the robot's position. It is particularly well-suited for scenarios where the robot needs to recompute its path in real-time as new information becomes available. D* uses a heuristic to estimate the cost of reaching the goal from a given position and iteratively updates this estimate as it explores the map.
Basic Idea:
- The fundamental idea behind D* is to iteratively update the path as new information becomes available or the robot's position changes. This is in contrast to static path planning algorithms like A* that plan a path once and assume a fixed environment. D* relies on cost maps that represent the terrain or environment. These maps assign costs to different cells or locations in the environment, indicating the desirability of traversing those areas. It maintains a path from the start to the goal, which can be updated as needed. Like A*, D* uses a heuristic function to estimate the cost from any cell to the goal. The core of the Original D* algorithm revolves around two primary functions:

- **PROCESS-STATE**: This function computes the optimal path costs to reach the goal.
- **MODIFY COST**: It is responsible for altering the cost function of arcs, denoted as c(), and introducing changes to states listed in the OPEN set.

These functions rely on a set of variables as follows:
• t(x): A tag variable that assumes the value NEW if state x has never appeared in the OPEN list, OPEN if it is currently listed, and CLOSED if it has been removed from the OPEN list.
• b(x): A backpointer from state x to the next state, denoted as y.

• c(x, y): Represents the cost of traversing an arc from state y to state x. When this value is positive, it signifies that states x and y are neighbors.
• h(x): This variable represents the path cost estimate, approximating the sum of arc costs from state x to the goal (G).
• k(x): The key variable classifies a state x within the OPEN list as either a RAISE state, indicating information about path cost increases, or a LOWER state, indicating information about path cost reductions. The variable k_min represents the minimum value among all k(x) values, while k_old represents the state of k_min prior to the most recent removal of a state from the OPEN list.

In the PROCESS-STATE function, the state x with the lowest k() value is removed from the OPEN list. If x is a LOWER state i.e,(k(x) = h(x)), its path cost is optimal since h(x) is equal to the previous $k_min.$ Cost changes are propagated to each neighbor y that has a backpointer to x, regardless of whether the new cost is greater or less than the old cost. These states, being descendants of x, are influenced by any change in the path cost of x. The backpointer of y is adjusted as necessary to establish a monotonic sequence. All neighbors receiving a new path cost are placed in the OPEN list, allowing them to propagate cost changes to their respective neighbors. For RAISE states, whose path cost may not be optimal, their optimal neighbors are examined before propagating cost changes to their descendants. If x can lower the path cost of a state that is not an immediate descendant, it is placed back on the OPEN list to prevent the creation of closed loops in backpointers. If a suboptimal neighbor can reduce the path cost of x, that neighbor is also placed back on the OPEN list. Consequently, the update is 'postponed' until the neighbor attains an optimal path cost. In the MODIFY-COST function, the arc cost function is updated with a new value. Since the path cost for state y is expected to change, state x is placed on the OPEN list. When x is expanded via PROCESS-STATE, it computes a new $h(y) = h(x) + c(x, y)$ and adds y to the OPEN list. Subsequent state expansions transmit cost changes to the descendants of y. The MAIN algorithm demonstrates the use of PROCESS-STATE and MODIFY-COST to navigate the robot from state S through the environment to G via an optimal route. It initializes by setting t() to NEW for all states, setting h(G) to 0, and placing G on the OPEN list. The algorithm repeatedly calls PROCESS-STATE (line 7) until it either finds an initial path to the robot's states (t(S) = CLOSED) or concludes that no path exists (val = NO-VAL and t(S) = NEW). The robot then follows the backpointers in the sequence R until it reaches the goal or detects a discrepancy between the sensor measurement of an arc cost s() and the stored arc cost c(), which can result from obstacles. Such discrepancies can occur anywhere, not just within the R sequence. MODIFY-COST is utilized to rectify the arc cost function c() and includes affected states in the OPEN list. The function returns GOAL-REACHED if the goal is reached and NO-PATH if the goal is unreachable. To emphasize the presence of two new sub-functions: LESS(a, b), which returns True if $a < b$ and False otherwise, and COST(x), which returns h(x) for state x. In summary, the D* algorithm is a dynamic path planning method that allows robots to adapt their paths in real-time as they navigate through changing environments or when their positions are uncertain. It achieves this through iterative updates to the path and cost estimates based on updated information from the environment.

Algorithm 3.3 Original D* Algorithm

```
procedure MAIN($S, G$)
    for $\forall$ state x in the graph
        t(x) = NEW
    INSERT(G,0)
    val = 0
    while t(S) $\neq$ CLOSED and val $\neq$ NO-VAL
        val = PROCESS-STATE()
    if t(S) = NEW
        return NO-PATH
    R = S
    while R $\neq$ G
        for $\forall$ (x,y) such that s(x,y) $\neq$ c(x,y)
            val = MODIFY-COST(x, y, s(x,y))
        while LESS(val ,COST(R)) and val $\neq$ NO-VAL
            val = PROCESS-STATE()
        R = b(R)
    return GOAL-REACHED
```

```
procedure PROCESS-STATE(())
    x = MIN-STATE()                    ▷ return the OPEN state with minimum k()
    if x = NULL
        return NO-VAL
    $k_{old}$ = k(x)
    DELETE(x)                          ▷ delete x from OPEN list and sets t(x) = CLOSED
    if $k_{old}$< h(x)
        for $\forall$ neighbour y of x
            if t(y) $\neq$ NEW and h(y) $\leq$ $k_{old}$ and
                h(x) > h(y) + c(y,x)
                b(x) = y
                h(x) = h(y) + c(y,x)
    if $k_{old}$ = h(x)
        for $\forall$ neighbour y of x
            if t(y) = NEW or
                (b(y) = x and h(y) $\neq$ h(x) + c(x,y)) or
                (b(y) $\neq$ x and h(y) > h(x) + c(x,y))
                b(y) = x
                INSERT(y, h(x)+c(x,y))         ▷ change h(y) with the second given
                input and inserts/repositions y on the OPEN list
    else
        for $\forall$ neighbour y of x
            if t(y) = NEW or
                (b(y) = x and h(y) $\neq$ h(x) + c(x,y))
                b(y) = x
                INSERT(y, h(x)+c(x,y))
            else
                if b(y) $\neq$ NEW and h(y) > h(x) + c(x,y) and
                    t(x) = CLOSED
                    INSERT(x, h(x))
                else
                    if b(y) $\neq$ NEW and h(x) > h(y) + c(y,x) and
                        t(y) = CLOSED and h(y) > $k_{old}$
                        INSERT(y, h(y))
    return MIN-VAL()                   ▷ returns $k_{min}$ for the OPEN list
procedure MODIFY-COST($x, y, cval$)
    c(x,y) = cval
    if t(x) = CLOSED
        INSERT(x, h(x))
    return MIN-VAL()                   ▷ returns $k_{min}$ for the OPEN list
```

Figure 3: Pseudocode of Original D* algorithm

Appendix C

Dubins curve

A Dubins curve is a path that connects two points in a two-dimensional space while respecting certain constraints. These constraints typically include a minimum turning radius or curvature for the path, which is particularly important in scenarios involving vehicles with limited turning capabilities, such as cars or aircraft. There are three basic types of Dubins curves:

- LSL (Left-Straight-Left): This path starts with a left turn, followed by a straight segment, and ends with another left turn.
- RSR (Right-Straight-Right): Similar to LSL, but it starts with a right turn, has a straight segment in the middle, and ends with a right turn.
- LSR (Left-Straight-Right): This path begins with a left turn, followed by a straight segment, and ends with a right turn.

Dubins curves are commonly used in robotics and autonomous vehicle navigation to find the shortest path between two points while adhering to the vehicle's constraints. They have applications in various fields, including aerial vehicle flight planning, ground vehicle path planning, and even in modeling the motion of biological organisms. The simplicity and optimality of Dubins curves make them a valuable tool in motion planning, especially when it's important to minimize travel distance and obey specific constraints. It was demonstrated that when considering the Dubins car, the shortest path between any two configurations can always be expressed as a combination of at most three basic motion primitives. Each of these motion primitives enacts a constant action over a specific period of time. Furthermore, to traverse these shortest paths, only three distinct actions are required, represented as $u \in -1, 0, 1$.

The primitives and their corresponding symbols are visually depicted below.

Symbol	Steering: u
S	0
L	1
R	-1

Specifically, the S primitive propels the car straight ahead, while the L and R primitives execute the sharpest possible left and right turns, respectively. Utilizing these symbols, any conceivable

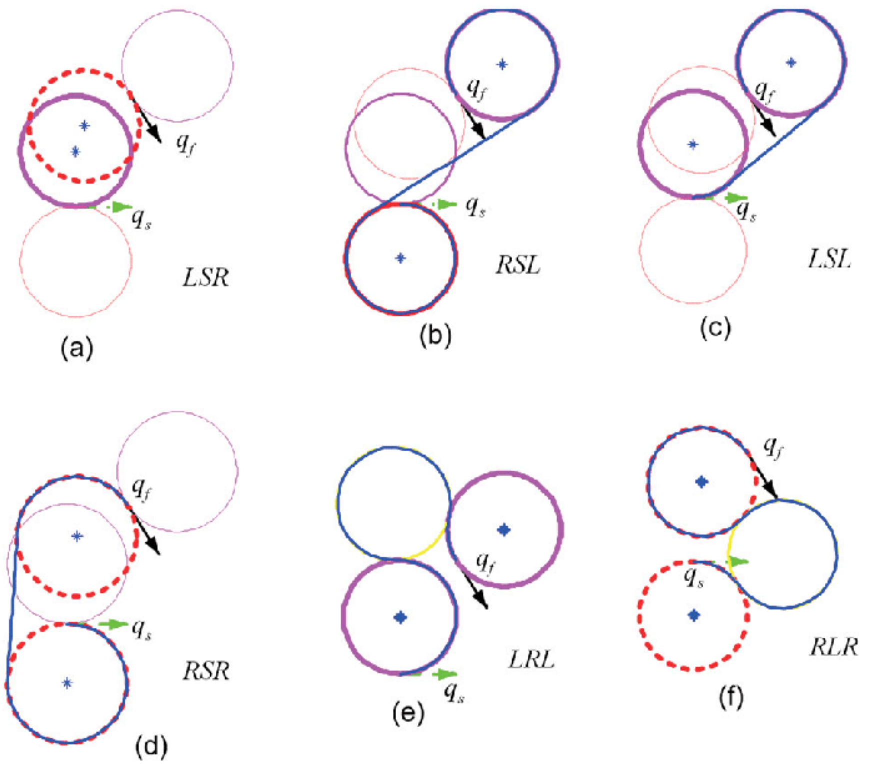


Figure 4: Main types of Dubins' paths

shortest path can be denoted as a sequence of three symbols, signifying the order in which these primitives are applied. Such a sequence is referred to as a "word." It's important to note that consecutive primitives of the same type can be merged into a single primitive, making two consecutive primitives of the same type unnecessary. Given this insight, there are ten potential words of length three. However, Dubins' analysis revealed that only the following six words are potentially optimal:

$$LRL, RLR, LSL, LSR, RSL, RSR \tag{3}$$

The shortest path between any two configurations can always be characterized by one of these words. These are called the Dubins curves. o provide a more precise description. It's essential to specify the duration of each primitive. For the L or R primitives, we can use subscripts to indicate the total amount of rotation accumulated during their application. Similarly, for the S primitive, a subscript can denote the total distance traveled. With these subscripts, we can more accurately characterize Dubins curves as:

$$L_\alpha, R_\beta, L_\gamma, ; R_\alpha, L_\beta, R_\gamma, ; R_\alpha, S_d, L_\gamma, ; R_\alpha, S_d, R_\gamma \tag{4}$$

where α and γ are constrained to the interval $[0, 2\pi)$, β falls within the range $(\pi, 2\pi)$, and d is greater than or equal to zero. It is crucial noting that β must be greater than π (if it is less, then some other word becomes the optimal choice). To enhance clarity and group together qualitatively similar paths, a compressed form of these words can be introduced, which will prove especially valuable when discussing Reeds-Shepp curves, as there are 46 of them, compared to the 6 Dubins curves. Let's denote a symbol as C, representing either R or L. Using this symbol, the six words used before can be compressed into just two base words:

$$CCC, CSC \tag{5}$$

In this compressed form, it's important to remember that two consecutive Cs must always be filled in by distinct turns (RR and LL are not allowed as subsequences). In this compressed notation, the base words can be precisely specified as:

$$C_\alpha, C_\beta, C_\gamma, ; C_\alpha, S_d, C_\gamma \tag{6}$$

with α and γ confined to the interval $[0, 2\pi)$, β in the range $(\pi, 2\pi)$, and d greater than or equal to zero.

Bibliography

[1] Webster, J.G., Huang, S. and Dissanayake, G. (2016). *Robot Localization: An Introduction*. In *Wiley Encyclopedia of Electrical and Electronics Engineering*, J.G. Webster (Ed.).

[2] Wallace Pereira Neves dos Reis, Guilherme José da Silva, Orides Morandin Junior, et al. (2021). *Extended Analysis on Tuning The Parameters of Adaptive Monte Carlo Localization ROS Package in an Automated Guided Vehicle.* Preprint Version 1. Available at Research Square.

[3] C. Roesmann, W. Feiten, T. Woesch, F. Hoffmann, and T. Bertram (2012). *Trajectory Modification Considering Dynamic Constraints of Autonomous Robots.* In *ROBOTIK 2012; 7th German Conference on Robotics*, Munich, Germany, 2012.

[4] K. Kurzer, (2016)*Path Planning in Unstructured Environments: A Real-time Hybrid A* Implementation for Fast and Deterministic Path Generation for the KTH Research Concept Vehicle*. Retrieved from https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-198534

[5] Portugal, David; Araújo, André; and Couceiro, Micael (2021). *Improving the Robustness of a Service Robot for Continuous Indoor Monitoring: An Incremental Approach.* From *International Journal of Advanced Robotic Systems*.

[6] *Introduction to Mobile Robotics.* Slides adopted from: Wolfram Burgard, Cyrill Stachniss, Maren Bennewitz, Kai Arras, and *Probabilistic Robotics* Book.

[7] *Probabilistic Robotics. Probabilistic Motion and Sensor Models.* Slides adopted from: Wolfram Burgard, Cyrill Stachniss, Maren Bennewitz, Kai Arras, and *Probabilistic Robotics* Book.

[8] Zammit, Christian; van Kampen, Erik-Jan (2018). Delft University of Technology. *Comparison between A* and RRT Algorithms for UAV Path Planning.* Zammit, Christian; van Kampen, Erik-Jan. Published in *Proceedings of the 2018 AIAA Guidance, Navigation, and Control Conference*.

[9] Anthony (Tony) Stentz. (1995). *The Focused D* Algorithm for Real-Time Replanning*. In *Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI '95)*, pp. 1652–1659.

[10] Lu, J.; , Y.; , Q.; Liu, Y.; Lu, J. (2023). *Jump Point Search Algorithm. Encyclopedia.* Available online: https://encyclopedia.pub/entry/24246

[11] Dolgov, D.; Thrun, S.; Montemerlo, M.; Diebel, J. (2010). Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments. *The International Journal of Robotics Research.*

[12] David Ferguson and Anthony (Tony) Stentz. (2005). *Field D*: An Interpolation-based Path Planner and Replanner*. In *Proceedings of 12th International Symposium on Robotics Research (ISRR '05)* pp. 239–253.

[13] Karaman, S.; Frazzoli, E. (2011). Sampling-based Algorithms for Optimal Motion Planning. *The International Journal of Robotics Research.*

[14] Robotic Systems (draft) of Kris Hauser notes from courses at Indiana University, Duke University, and University of Illinois at Urbana-Champaign, Section III. MOTION PLANNING.

[15] Janko Petereit, Thomas Emter, Christian W. Frey. *Application of Hybrid A* to an Autonomous Mobile Robot for Path Planning in Unstructured Outdoor Environments.* Fraunhofer Institute of Optronics, System Technologies and Image Exploitation IOSB, Karlsruhe, Germany;

[16] Gao, Xuheng; Luan, Hao; Xia, Bingyi; Zhao, Ziqi; Wang, Jiankun; Meng, Max Q.-H. (2023). *A Divide-and-Conquer Control Strategy with Decentralized Control Barrier Function for Luggage Trolley Transportation by Collaborative Robots. Robotica*, 1–16.

[17] Dang, C.V.; Ahn, H.; Lee, D.S.; Lee, S.C. (2022). *Improved Analytic Expansions in Hybrid A-Star Path Planning for Non-Holonomic Robots. Appl. Sci..*

[18] Kevin M. Lynch and Frank C. Park. (2017). *Modern Robotics: Mechanics, Planning, and Control.*

[19] Meng, T., Yang, T., Huang, J., et al. (2023). *Improved Hybrid A-Star Algorithm for Path Planning in Autonomous Parking System Based on Multi-Stage Dynamic Optimization. Int. J. Automot. Technol.*

[20] Christoph Rösmann, Frank Hoffmann, Torsten Bertram. (2017). Integrated Online Trajectory Planning and Optimization in Distinctive Topologies. *Robotics and Autonomous Systems*, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889016300495

[21] ROS (Robot Operating System) https://www.ros.org/

[22] Jonathan D. Gammell, Siddhartha Srinivasa, and Timothy D. Barfoot. (2014) *Informed RRT*: Optimal Incremental Path Planning Focused through an Admissible Ellipsoidal Heuristic.* In *Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*

[23] Robin R. Murphy. (2000). *Introduction to AI Robotics*. A Bradford Book, The MIT Press, Cambridge, Massachusetts, London, England.

[24] Roland Siegwart, Illah R. Nourbakhsh, and Davide Scaramuzza. (2004). *Introduction to Autonomous Mobile Robots*, second edition. The MIT Press, Cambridge, Massachusetts, London, England.

[25] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A. Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementations*. The MIT Press.

[26] D. Fox, W. Burgard, and S. Thrun, (1997) *The Dynamic Window Approach to Collision Avoidance* in *IEEE Robotics & Automation Magazine*, vol. 4, no. 1, pp. 23-33.

[27] H. Quang, T. Manh, C. Manh, P. Tin, M. Van, N. Tien Kiem, and D. Nguyen Duc, *An Approach to Design Navigation System for Omnidirectional Mobile Robot Based on ROS*, *International Journal of Mechanical Engineering and Robotics Research*, pp. 1502-1508

[28] Bartłomiej Cybulski, A. Wegierska, Grzegorz Granosik ,(2019) *Accuracy Comparison of Navigation Local Planners on ROS-based Mobile Robot* Conference: 2019 *12th International Workshop on Robot Motion and Control*

[29] Xuexi Zhang, Jiajun Lai, Dongliang Xu, Huaijun Li, and Minyue Fu, (2020), *2D Lidar-Based SLAM and Path Planning for Indoor Rescue Using Mobile Robots*

[30] Y. Abdelrasoul, A. B. S. H. Saman, P. Sebastian, (2016), *A quantitative study of tuning ROS gmapping parameters and their effect on performing indoor 2D SLAM* 2nd IEEE International Symposium on Robotics and Manufacturing Automation (ROMA).

[31] Kaiyu Zheng, (2016), *ROS Navigation Tuning Guide*.

[32] Looi, C. Z., Ng, D. W. K. (2021). *A Study on the Effect of Parameters for ROS Motion Planner and Navigation System for Indoor Robot. International Journal of Electrical and Computer Engineering Research*

[33] Pablo Marín, Ahmed Hussein, David Martín Gómez, Arturo de la Escalera. Journal of Advanced Transportation, (2018), *Global and Local Path Planning Study in a ROS-Based Research Platform for Autonomous Vehicles.* Pages 1-10.