Multi Agents Motion Planning and Collision Avoidance

Report of module: Optimization Models and Algorithms

Author: Weijie Qi

Date: 01/2024

Table of Contents

1 Introduction

In recent years, the field of multi-agent path planning in continuous space has gained significant attention due to its relevance in various real-world scenarios. The objective of this research is to address the combined problem of planning optimal paths and generating feasible and smooth trajectories for multiple intelligent agents. This entails ensuring avoidance of collisions with both the map boundaries and obstacles, as well as preventing collisions between the agents themselves.

To achieve this, the proposed approach entails a combination of several optimization-based algorithms. Firstly, the environment's roadmap is constructed using the Quadtree algorithm, which efficiently divides the space into smaller regions to facilitate path planning. Next, the Dijkstra algorithm, a graph-based optimization algorithm, is employed to identify the shortest path from the starting point to the destination for each agent, which serves as a foundation for subsequent motion planning.

To generate smooth and continuous paths between the discrete points along the shortest path, the minimum jerk method is utilized. This technique aims to minimize the jerk, or the rate of change of acceleration, in the generated trajectories. By formulating the motion planning problem as a quadratic optimization problem, the minimum jerk method produces paths that are not only efficient but also smooth and comfortable for the agents.

Furthermore, to prevent collisions between the intelligent agents, the Optimal Reciprocal Collision Avoidance (ORCA) algorithm is implemented. ORCA formulates collision avoidance as a set of linear constraints. By solving this by linear programming, ORCA calculates optimal velocities for each agent, ensuring safe and collision-free navigation. This aspect of the algorithm addresses the motion planning component, ensuring that the generated trajectories are collision-free.

In order to present the contents of the report, the following sections will be covered: Section 2 provides a thorough understanding of the underlying theory and concepts related to multi-agent path planning and motion planning. It will also define the specific problem statement and objectives of the research. Section 3 will detail the various algorithms and methodologies employed in the proposed approach, including the Quadtree algorithm for environment mapping, Dijkstra's algorithm for path planning, the minimum jerk method for generating smooth trajectories, and the ORCA algorithm for collision avoidance. Section 4 will describe the implementation of the proposed approach, while Section 5 will present the experimental results. In Section 6, it will summarize the findings of the research, highlight the contributions made, and discuss future directions for further improvement and research in the field.

Through the exploration of the proposed methodology, this paper aims to shed light on the challenges faced in multi-agent path planning and motion planning, while highlighting the advantages and effectiveness of the combined approach. By studying and addressing these challenges, we can further enhance the capabilities and performance of intelligent systems in real-world scenarios.

2 Background Theory and Problem Definition

2.1 Configuration & Configuration Space

In multi-agent path planning and motion planning, understanding the concept of configuration and configuration space is crucial, providing a solid foundation for subsequent discussions on path planning and motion planning.

Definition 1 *A configuration is a complete specification of all the points of a robot, capturing the position and orientation of the robot's joints or end-effector in its workspace. It provides a concise representation of the robot's state at a specific moment.*

Definition 2 *The configuration space, also known as C-Space, refers to the set of all possible configurations that a robot can attain. It encompasses the entire range of feasible positions and orientations considering the robot's physical constraints.*

Definition 3 *Degrees of freedom (DOF) are the parameters that can be changed to determine different configurations of a robot.*

Fig. 1 provides a visual representation of the definitions discussed earlier. Let's consider our robot, denoted as A, which is a rigid body operating within the workspace W. The workspace W is a subset of the Euclidean space $\mathbb{R}^2$ and is associated with a fixed coordinate frame F_W, with the origin O_W. Similarly, the robot A possesses its own fixed coordinate frame F_A, with the origin O_A. A configuration q of A is a specification of position and orientation of F_A w.r.t. F_W. This configuration is uniquely determined by the triple (x, y, θ), representing the robot's position in the x and y dimensions, as well as its orientation θ. Thus, we can conclude that the robot has a total of three degrees of freedom, encompassing all possible configurations. The configuration space C of A represents the entirety of possible configurations that the robot can assume.

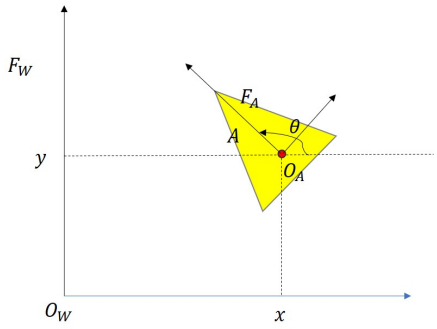


Figure 1

Definition 4 *A configuration, denoted as q, is considered collision-free when the robot positioned at q does not intersect with any obstacles present in the workspace. The collection of all collision-free configurations is referred to as the C_{free} space. Conversely, C_{obs} represents the set of configurations that result in collisions with obstacles.*

The ultimate objective of motion planning is to find a continuous sequence of collision-free configurations that connect an initial configuration to a desired final configuration.

2.2 Exploring and Discretizing C-Spaces

Exploring and discretizing the configuration space is a crucial step in path planning as it allows for the representation of the C-space as a graph or grid. This process is essential in order to effectively navigate and analyze the complex configuration space.

There are two different approaches to explore and discretize C-spaces. One is Sampling-based planning, which involves randomly and incrementally exploring the $C-free$ space and constructing a graph of sampled configurations. Methods such as probabilistic roadmaps (PRM) (Hsu et al. 1998) and rapidly-exploring random trees (RRTs) (LaValle, et al. 2001) fall under this category. The other approach is combinatorial planning, which provides a concise representation of the $C-free$ space in the form of a topological graph (G), capturing the connectivity of the C-space. Methods such as exact cell decomposition, approximate cell decomposition, and maximum clearance are commonly used in combinatorial planning. The combinatorial planning method to discretize C-space offers advantages such as simplicity, efficiency, and scalability in representing the connectivity of the C-space.

After discretizing the $C-free$ space,a roadmap is obtained. It enables us to establish a high-level abstraction of the C-space, which simplifies the search for feasible paths. By applying algorithms such as Dijkstra's algorithm (Frana et al. 2010) or A* (Hart, et al. 1968) search on the roadmap, we can find the shortest collision-free path between a given start and goal configuration. An example of a built roadmap of $C-free$ space is shown in Fig. 2.

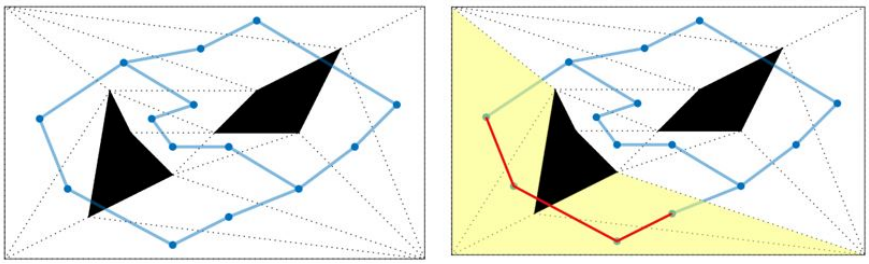


Figure 2

In this roadmap, each vertex (v) represents a configuration in the $C-free$ space, and each edge (e) represents a collision-free path connecting two configurations. Additionally, a channel, depicted as the red lines in Fig. 2, corresponds to a sequence of adjacent cells within the $C-free$ space. This sequence of cells aligns with a path in the connectivity graph, following the edges defined in the graph.

2.3 Path Planning & Motion Planning

Path planning and motion planning are fundamental concepts in the field of robotics and autonomous systems. They are interdependent and both are necessary for successful robot navigation. Path planning provides a high-level abstract representation of the desired trajectory, focuses on defining a sequence of waypoints or configurations that a robot should follow to reach its destination. Motion planning, on the other hand, deals with the generation of robot motions to follow the planned path. It takes into account the kinematic and dynamic constraints of the robot, such as its velocity limits, joint limits, and physical limitations. Motion planning algorithms aim to compute feasible and smooth trajectories that adhere to the planned path while satisfying the robot's constraints.

As shown in Fig. 3, the path composed of red line segments is the result of path planning. However, follow the line directly by connecting the waypoints can lead to a tortuous trajectory, which is not ideal for robot movement. We need to enable the robot to make smooth transitions at every waypoint. This is where motion planning comes into play. It accounts for the specific capabilities and limitations of the robot, ensuring that the generated trajectory is feasible The green curve in the figure is the result of motion planning based on the path points generated by the path planning, which provides a smoother and more practical path that the agent can effectively follow.

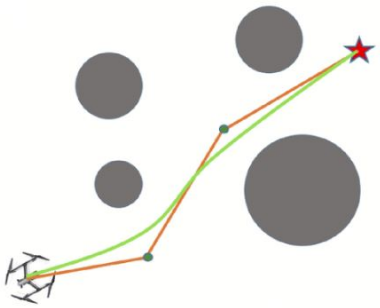


Figure 3

In addition to geometric paths, motion planning also considers time-parametrized trajectories. These trajectories incorporate the element of time, determining how the robot or system should move throughout the path over a certain duration. The time parameterization ensures smooth and controlled motion, taking into account factors such as acceleration, velocity, and jerk.

The process of performing path planning and motion planning to enable the agent to navigate from the initial configuration to the desired final configuration without colliding is hierarchical. The basic framework of this process can be outlined as follows:

- High-level path planning: Algorithms such as A*, Dijkstra, and other similar approaches are employed at the high level to determine an optimal path from the start to the goal configuration. These algorithms typically involve abstracting the configuration space into a roadmap and leveraging graph theory principles to identify discrete optimal path nodes. By navigating through this discretized representation of the environment, these algorithms aim to find the shortest collision-free path.
- High-level motion planning: Once the path is obtained, high-level motion planning techniques, such as Minimum Jerk or Bezier curves, are applied to generate smooth and feasible trajectories along the planned path. These methods take into account the kinematic and dynamic constraints of the agent to ensure that the generated motions are both practical and efficient.
- Low-level motion planning: At the lowest level, low-level obstacle avoidance algorithms like Velocity Obstacles (VO), Reciprocal Velocity Obstacles (RVO), Optimal Reciprocal Collision Avoidance (ORCA), and others are utilized. These algorithms focus on real-time collision avoidance and adjust the agent's motion to avoid obstacles or other agents dynamically.

By following this hierarchical approach, path planning and motion planning work together to enable safe and efficient navigation for the agent.

2.4 Problem Formulation

The problem formulation in this study focuses on the task of coordinating three agents that share a two-dimensional workspace. The workspace is a closed rectangular region, and the agents are constrained to operate within the boundaries of this rectangle. The workspace contains obstacles, which, for simplification purposes, are assumed to be convex polygons. The position and shape of each obstacle can be specified by the user, as well as the initial and goal positions for each agent. The planning algorithms need to adapt to various scenarios based on these user-defined specifications.

Fig. 4 illustrates an example workspace. The black border represents the boundaries of the map, while the blue polygon in the center represents a user-defined obstacle. The user then specifies the starting and target points for each of the three agents, with each agent's starting and target points indicated by solid points of the same color.

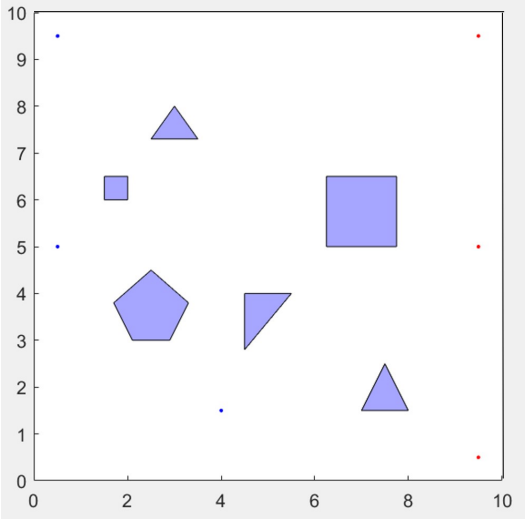


Figure 4

Each agent is represented by a circular shape with a defined radius. A collision between two agents is considered to occur if the distance between the centers of their respective disks is less than the sum of their radii. Additionally, each agent has an orientation that determines its heading direction.

The properties of each agent include its radius, the current position denoted as p_A, and its current velocity denoted as v_A.

By formulating the problem with these components, we establish the foundation for developing planning algorithms that can effectively coordinate the movements of the three agents within the specified workspace. The problem formulation takes into account the workspace boundaries, obstacle representation, agent properties, and collision avoidance criteria, enabling the creation of efficient and safe planning strategies. The objective of these planning algorithms is to generate valid paths for each agent that connect their respective start and goal positions, while ensuring there are no collisions with the workspace boundaries, obstacles, or other agents. This comprehensive approach ensures that the agents can navigate through the workspace in a coordinated manner, achieving their desired configurations without encountering any conflicts.

3 Methodologies and Algorithms

3.1 Quadtree Decomposition

The quadtree algorithm is a hierarchical data structure commonly used in computer graphics and spatial partitioning. It is particularly useful for decomposing and representing complex spaces, such as the configuration space, in a more efficient and organized manner.

The quadtree algorithm starts by defining a bounding box that encompasses the entire configuration space. This bounding box is then recursively divided into four equal quadrants, creating four child nodes. Each child node represents a smaller cell within the configuration space.

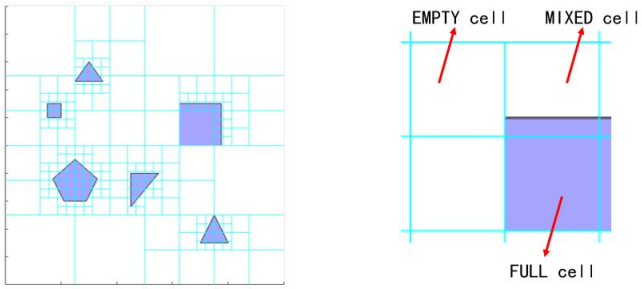


Figure 5: Left: An example of quadtree decomposition. Right: Division of three cell types

During the subdivision process, each cell is classified into one of three types. Firstly, if a cell does not intersect with any obstacles C_{obs}, it is classified as EMPTY. Secondly, if a cell is entirely contained within an obstacle C_{obs}, it is classified as FULL. Lastly, if a cell intersects with some obstacles but is not entirely contained within them, it is classified as MIXED. The process of subdivision and classification continues iteratively for the child nodes associated with MIXED cells. Each MIXED cell is further divided into four quadrants, creating additional child nodes. This recursive subdivision process continues until a desired level of granularity or a termination condition is met.

To apply the quadtree algorithm for modeling the configuration space into a roadmap, the following steps can be followed:

1. Create the root node: Initialize the quadtree with a root node that corresponds to the bounding box of the configuration space.
2. Subdivide the root node: Divide the root node into four quadrants, creating four child nodes that represent smaller regions within the configuration space.
3. Determine termination condition: Define a termination condition, such as a maximum level of subdivision or a minimum size threshold, to stop further subdivision of the quadtree.
4. Evaluate regions: For each child node or quadrant, evaluate the region to determine if it is obstacle-free or contains obstacles. This evaluation can be done using collision detection algorithms or other methods specific to the problem domain.
5. Recurse: If the termination condition is not met, repeat steps 2 to 4 for each child node, subdividing them into four quadrants and evaluating the regions within.

6. Construct the roadmap: Based on the evaluation results, create the roadmap by connecting the obstacle-free regions or cells in the quadtree. Each node or cell in the quadtree represents a configuration, and the connections between nodes represent collision-free paths.

By following these steps, quadtree decomposition decomposes the workspace into a roadmap, which abstractly represents the connectivity of different areas in the work space, and provides convenience for the subsequent path planning.

3.2 Path Planning

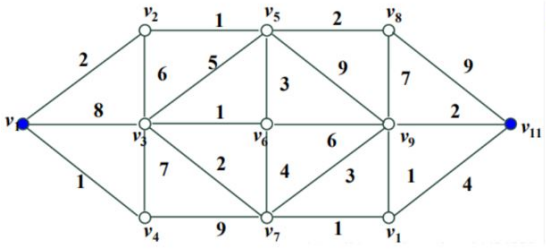
3.2.1 Dijkstra's algorithm

Dijkstra's algorithm is a widely used method for finding the shortest path between two points in a graph. In the context of path planning, after constructing a roadmap, Dijkstra's algorithm can be employed to find the shortest path from the starting point to the destination.

The algorithm begins by assigning a tentative distance value to every node in the graph, which represents the current shortest distance from the starting point. Initially, the distance value for the starting node is set to 0, while all other nodes are assigned a distance of infinity.

Next, the algorithm iteratively selects the node with the smallest tentative distance as the current node. It then examines all the neighboring nodes of the current node and calculates their tentative distances by summing the distance of the current node with the weight of the edge connecting them. If the newly calculated tentative distance is smaller than the current distance value of the neighbor, the distance value is updated.

Fig. 6 illustrates the process of updating distances starting from the source vertex v_1. The shortest distance from v_1 to itself is set to 0. The shortest distance from the nodes directly adjacent to v_1 to v_1 is updated to the value of v_1 plus the weight of the edge connecting v_1 to itself. The distances of the remaining nodes that cannot be directly reached from v_1 are set to infinity.



Node	Path	Distance to v1	Whether is the shortest
V1	V1→V1	0	1
V2	V1→V2	2	0
V3	V1→V3	8	0
V4	V1→V4	1	0
V5		∞	0
V6		∞	0
V7		∞	0
V8		∞	0
V9		∞	0
V10		∞	0
V11		∞	0

Figure 6

Take the minimum from all nodes that are not marked as finding the minimum distance and repeat the steps above. This process continues until the algorithm has visited all the nodes in the graph or until the destination node is reached. At the end of the algorithm, the shortest path from the starting point to the destination is determined by backtracking from the destination node, following the nodes with the lowest distance values.

3.3 Path Smoothing and Motion Planning

Minimum jerk trajectory planning (Kyriakopoulos et al. 1988) is a motion planning algorithm used in robotics and control systems. It aims to generate smooth and natural trajectories for robotic systems by minimizing the jerk, which is the rate of change of acceleration. The algorithm is based on the idea that human movements are characterized by smooth and continuous accelerations, as abrupt changes in acceleration can be uncomfortable and inefficient. Therefore, the minimum jerk algorithm seeks to mimic these natural movements by minimizing the jerk. The algorithm typically takes a set of initial and final conditions, such as positions, velocities, and accelerations, and calculates a trajectory that smoothly connects these points while minimizing jerk. It achieves this by formulating the problem as an optimization task, where the objective is to find the trajectory that minimizes the integral of the jerk over the entire time duration. In this sub-section, we will start by briefly introducing quadratic programming, because it is the cornerstone of our solution to the problem of motion planning

3.3.1 Quadratic Form

A quadratic homogeneous function with n variables $x = x_1, x_2, \dots, x_{n-1}, x_n$ is defined as follows:

$$f(x_1, x_2, \dots, x_n) = a_{11}x_1^2 + a_{22}x_2^2 + \dots + a_{nn}x_n^2 + 2\sum_{i,j=1}^{n} a_{ij}x_i x_j.$$

This function is commonly referred to as a quadratic form.

The above formula can be expressed as:

$$f = \sum_{i,j=1}^{n} a_{ij}x_i x_j.$$

The matrix representation of this quadratic form is:

$$f = x^T Ax,$$

where A is an $n \times n$ matrix of coefficients.

3.3.2 Quadratic Programming Problem

When the objective function f is quadratic and the constraints are linear, the optimization problem is a quadratic programming problem. It can be expressed as follows:

$$\begin{aligned} \min_x f(x) &= \frac{1}{2}x^T Qx + q^T x \\ \text{s.t. } Ax &= b \\ Gx &\leq h \end{aligned}.$$

Quadratic programming is a convex optimization problem.

3.3.3 Representation of Trajectory and Its Derivatives

In path planning, a series of waypoints can be obtained without time information. However, the trajectory is a function of time t and is often expressed as a polynomial of order n. This can be represented as:

$$\begin{aligned} f(t) &= p_0 + p_1 t + p_2 t^2 + \ldots + p_n t^n \\ &= \textstyle\sum_{i=0}^{n} p_i t^i \end{aligned} .$$

Here, $p_0, p_1, \ldots, p_n$ are the trajectory parameters, which are also the optimization parameters.

Alternatively, we can write $f(t)$ as a vector multiplication:

$$f(t) = \begin{bmatrix} 1 & t & \ldots & t^n \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \end{bmatrix} .$$

The derivatives of the trajectory function represent changes in speed, acceleration, jerk, and other parameters with respect to time. We can express the derivatives as follows:

$$\begin{aligned} \text{vel}(t) = f^{(1)}(t) &= \begin{bmatrix} 0 & 1 & 2t & 3t^2 & \cdots & \frac{n!}{(n-1)!}t^{n-1} \end{bmatrix} \cdot p \\ acc(t) = f^{(2)}(t) &= \begin{bmatrix} 0 & 0 & 2 & 6t & \cdots & \frac{n!}{(n-2)!}t^{n-2} \end{bmatrix} \cdot p \\ \text{jerk}(t) = f^{(3)}(t) &= \begin{bmatrix} 0 & 0 & 0 & 6 & \cdots & \frac{n!}{(n-3)!}t^{n-3} \end{bmatrix} \cdot p \end{aligned} .$$

Here, $p = \begin{bmatrix} p_0 & p_1 & \ldots & p_n \end{bmatrix}^T$ is the parameter vector.

The general formula for the k^{th} derivative of the trajectory is:

$$f^{(k)}(t) = \begin{bmatrix} \overbrace{0 \quad 0 \quad \cdots \quad 0}^{\text{k}} & \overbrace{\frac{(k+0)!}{0!}t^0 \quad \frac{(k+1)!}{1!}t^1 \quad \frac{(k+2)!}{2!}t^2 \quad \cdots \quad \frac{n!}{(n-k)!}t^{n-k}}^{\text{n-k+1}} \end{bmatrix} \cdot p$$

where k represents the order of the derivative.

Complex trajectories often require representation using multiple polynomials. The complete trajectory can be divided into M segments based on time intervals. Each segment can be represented by a polynomial trajectory. For example:

$$f(t) = \begin{cases} p_{1,0} + p_{1,1}t + p_{1,2}t^2 + \cdots + p_{1,n}t^n & T_0 \le t \le T_1 \\ p_{2,0} + p_{2,1}t + p_{2,2}t^2 + \cdots + p_{2,n}t^n & T_1 \le t \le T_2 \\ \vdots & \\ p_{M,0} + p_{M,1}t + p_{M,2}t^2 + \cdots + p_{M,n}t^n & T_{M-1} \le t < T_M \end{cases} .$$

Here, p_{ij} represents the j^{th} parameter of the i^{th} segment of the trajectory.

3.3.4 Determining the Polynomial Order

A jerk is defined as the third derivative of a trajectory function $f(t)$, denoted by $K = 3$. The minimum-jerk approach involves constraining the position, velocity, and acceleration of both the

head and tail, which results in six equality constraints. As a result, the optimization parameter must provide more than six degrees of freedom. Since a fifth-order polynomial consists of six coefficients, it satisfies the minimum requirements. Therefore, we can select a fifth-order polynomial to represent each locus, resulting in the following expression:

$$f(t) = \begin{cases} p_{1,0} + p_{1,1}t + p_{1,2}t^2 + p_{1,3}t^3 + p_{1,4}t^4 + p_{1,5}t^5 & T_0 \leq t \leq T_1 \\ p_{2,0} + p_{2,1}t + p_{2,2}t^2 + p_{2,3}t^3 + p_{2,4}t^4 + p_{2,5}t^5 & T_1 \leq t \leq T_2 \\ \vdots & \\ p_{M,0} + p_{M,1}t + p_{M,2}t^2 + p_{M,3}t^3 + p_{M,4}t^4 + p_{M,5}t^5 & T_{M-1} \leq t < T_M \end{cases}.$$

This expression encompasses M polynomial trajectories, each consisting of $Order + 1 = 6$ coefficients, resulting in a total of $M(Order + 1) = 6M$ coefficients.

3.3.5 Constructing Objective Function

To construct the objective function, let's start by specifying the jerk function, denoted as:

$$\begin{aligned} \text{jerk}(t) = f^{(3)}(t) &= 6p_3 + 24tp_4 + 60t^2p_5 \\ &= \begin{bmatrix} 0 & 0 & 0 & 6 & 24t & 60t^2 \end{bmatrix} \cdot p \end{aligned}$$

To minimize the jerk function, we aim to minimize its integral over the entire time duration. In the minimum-jerk trajectory, we choose to minimize the 2-norm of the jerk function, which can be expressed as:

$$\min_p f^{(3)}(t) = \min_p \sum_{i=1}^{M} \int_{T_{i-1}}^{T_i} \left(f^{(3)}(t)\right)^2 dt$$

Thus, the objective function $J(p)$ is defined as:

$$J(p) = \sum_{i=1}^{M} \int_{T_{i-1}}^{T_i} \left(f^{(3)}(t)\right)^2 dt$$

Let's define $a = \begin{bmatrix} 0 & 0 & 0 & 6 & 24t & 60t^2 \end{bmatrix}^T$. By squaring $f(t)$, we get:

$$\begin{aligned} \left(f^{(3)}(t)\right)^2 &= \left(\begin{bmatrix} 0 & 0 & 0 & 6 & 24t & 60t^2 \end{bmatrix} \cdot p\right)^T \left(\begin{bmatrix} 0 & 0 & 0 & 6 & 24t & 60t^2 \end{bmatrix} \cdot p\right) \\ &= \left(a^T p\right)^T \left(a^T p\right) \\ &= p^T a a^T p \end{aligned}$$

Let $A(t) = aa^T$, then we have:

$$A = aa^T = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 6 \\ 24t \\ 60t^2 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 6 & 24t & 60t^2 \end{bmatrix}$$
$$= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 36 & 144t & 360t^2 \\ 0 & 0 & 0 & 144t & 576t^2 & 1440t^3 \\ 0 & 0 & 0 & 360t^2 & 1440t^3 & 3600t^4 \end{bmatrix}$$

Therefore, the integral of $f^{(3)}(t)$ is:

$$\int_{T_{i-1}}^{T_i} \left(f^{(3)}(t)\right)^2 dt = \int_{T_{i-1}}^{T_i} p^T A(t) p dt \quad = p^T \cdot \int_{T_{i-1}}^{T_i} A(t) dt \cdot p$$

Let the integral of $A(t)$ be the matrix Q, then:

$$Q = \int_{T_{i-1}}^{T_i} A(t) dt = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 36 & 144t & 360t^2 \\ 0 & 0 & 0 & 144t & 576t^2 & 1440t^3 \\ 0 & 0 & 0 & 360t^2 & 1440t^3 & 3600t^4 \end{bmatrix}_{T_{i-1}}^{T_i}$$

The final objective function can be expressed as:

$$\begin{aligned} J(p) &= \sum_{i=1}^{M} \int_{T_{i-1}}^{T_i} \left(f^{(3)}(t)\right)^2 \\ &= \sum_{i=1}^{M} p_i^T Q_i p_i \\ &= \begin{bmatrix} p_1^T & p_2^T & \cdots & p_M^T \end{bmatrix} \begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & \ddots & \\ & & & Q_M \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_M \end{bmatrix} \\ &= p^T Q p \end{aligned}$$

$J(p)$ represents a quadratic problem, which means that a minimum-jerk trajectory optimization problem can be converted to a quadratic programming problem.

3.3.6 Adding Constraints

If obstacles are not considered, there are two main types of constraints: derivative constraints and continuity constraints.

Derivative Constraints: These constraints restrict the initial state and termination state of the trajectory, as well as the start/end position of each trajectory. They use the path points obtained from path planning to constrain the trajectory. The derivative constraints include the following:

- Initial state and termination state constraints for each trajectory segment, represented by:

$$\begin{cases} f^{(k)}(T_0) = x_0^{(k)} \\ f^{(k)}(T_M) = x_M^{(k)} \end{cases}, k = 0, 1, \cdots, K-1 .$$

  Here, T_0 and T_M represent the initial and termination time, and $x_0^{(k)}$ and $x_M^{(k)}$ represent the corresponding states.

- Start position constraint for each trajectory segment, represented by:

$$f_i(T_{i-1}) = x_{i-1}.$$

  Here, i represents the segment index, with $i = 2, 3, \cdots, M$.

For trajectories with $M+1$ waypoints, there are $2K + M - 1$ derivative constraints.

Continuity Constraint: This constraint ensures smooth transitions between adjacent trajectories. At the connection point of two trajectories, the continuity constraint requires that the position, speed, and acceleration of the two trajectories are consistent. Specifically, it is represented by the equations:

$$\begin{cases} f_i(T_i) = f_{i+1}(T_i) \\ f_i^{(1)}(T_i) = f_{i+1}^{(1)}(T_i) \\ f_i^{(2)}(T_i) = f_{i+1}^{(2)}(T_i) \end{cases}, i = 1, 2, \cdots, M-1 .$$

These continuity constraints contribute to achieving smooth motion between trajectories.

By incorporating these constraints, trajectory planning can ensure the desired motion while accounting for obstacles and enabling smooth transitions.

3.4 Collision Avoidance

3.4.1 Collision Detection in Velocity Domain

The Velocity Obstacle (VO) algorithm (Fiorini et al. 1998) is a popular motion planning algorithm used for collision avoidance in dynamic environments. The main idea behind the VO algorithm is to transform the collision detection problem from the spatial domain to the velocity domain. By doing so, the algorithm can efficiently compute safe velocities for an agent by considering the relative velocities and positions of other obstacles or agents. It constructs a velocity obstacle for each obstacle, which represents the set of velocities that would result in a collision if the agent continues on its current path.

Consider two circular objects, A and B, at time $t(0)$, each with their respective velocities V(A) and V(B). Object A represents the robot, while object B represents another robot perceived as a moving obstacle.

The first step involves establishing the relative spatial relationship between objects A and B by mapping B into the configuration space of A. This is achieved by reducing A to a point and

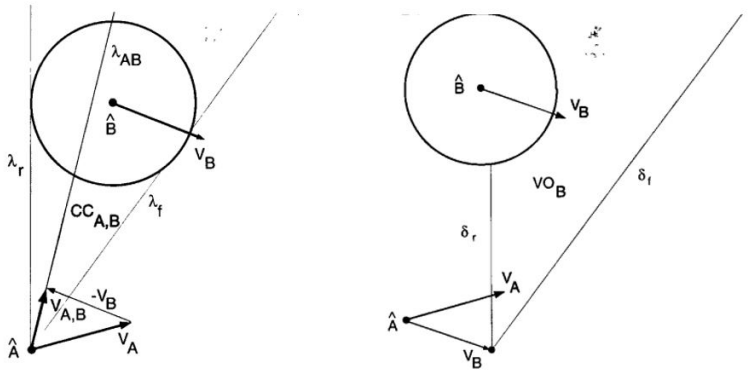


Figure 7

enlarging B to a size of $r_A + r_B$. As illustrated on the left side in Fig. 7, after the transformation, object A becomes particle $\hat{A}$, and object B becomes disk $\hat{B}$ with radius $r_A + r_B$.

Two rays, λ_r and λ_f, are extended from point $\hat{A}$ and tangentially intersect with disk $\hat{B}$. The cone formed by these two rays is referred to as the Collision Cone $CC_{A,B}$.

$$CC_{A,B} = \{\mathbf{v}_{A,B} \mid \lambda_{A,B} \cap \widehat{B} \neq \emptyset\}$$

In this context, $\mathbf{v}_{A,B}$ represents the relative velocity of $\hat{A}$ with respect to $\hat{B}$. It is useful to describe the equivalent velocity set on the absolute velocities of $\hat{A}$. This could be done by computing the Minkowski vector sum of $CC_{A,B}$ and $\mathbf{v}_B$. This operation represents the translation of the collision cone $CC_{A,B}$ by $\mathbf{v}_B$.

$$VO_B = CC_{A.B} \oplus \mathbf{v}_B$$

By now, the VO_B represents the velocity obstacle specific to a particular agent, which in this case is agent B. However, in order to account for avoiding collisions with multiple agents, we take the union of all the individual velocity obstacles.

$$VO = \cup_{i=1}^{m} VO_{B_i}$$

Here,m represents the total number of other agents in the environment. The resulting VO represents the set of velocities that would lead to a collision if the agent continues on its current path, considering the velocities and positions of all other agents.

3.4.2 Reciprocal Velocity Obstacle

In the VO algorithm, each agent considers other agents as obstacles and computes the velocity obstacle to avoid collisions. However, in the Reciprocal Velocity Obstacle (RVO) algorithm (Van et al. 2008), the concept of reciprocity is introduced, and each agent takes into account the motion intentions of other agents while computing the velocity obstacle. This solves the jitter problem of the original VO algorithm, leading to smoother and more efficient movement.

The jitter problem in the VO algorithm stems from the assumption that other agents solely move towards their respective destinations at a fixed speed without taking the same evasive actions. In Fig. 8, for instance, when a collision is predicted, both agents A and B choose velocities that lie outside their respective Velocity Obstacles to avoid the collision. However, since their newly chosen velocities are no longer within the Velocity Obstacle of the current speed, in the subsequent frame, they revert back to their original speeds, leading to a repetitive cycle of velocity adjustments. Consequently, their motion trajectory becomes unnaturally erratic.

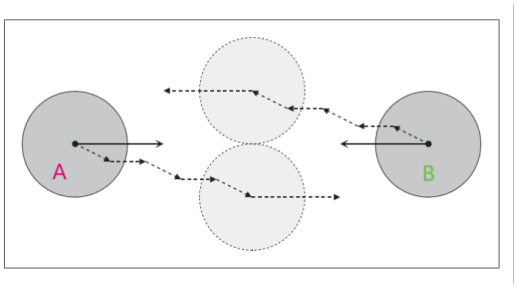


Figure 8

The strategy of RVO algorithm is to assume that other robots also bear the same responsibility for obstacle avoidance. Instead of directly selecting the velocity v'_A outside VO as the new speed, it averages the current velocity v_A and the velocity v'_A outside VO. It defines the Reciprocal Velocity Obstacle as follows:

$$RVO^A_B(\mathbf{v}_B, \mathbf{v}_A) = \{\mathbf{v}'_A | 2\mathbf{v}'_A - \mathbf{v}_A \in VO^A_B(\mathbf{v}_B)\}$$

It is proved that after choosing the new velocities for agent A and agent B by $\mathbf{v}_A$ and $\mathbf{v}_B$ respectively, the old velocity v_A of agent A would fall within the VO of the new velocity of agent B. This implies that after selecting the new velocity, the old velocity becomes invalid and will not be chosen. This is in contrast to the situation when using the original velocity obstacles, where the agent might oscillate between its original velocity and the new velocity, causing repetitive oscillations in its trajectory.

3.4.3 Optimal Reciprocal Collision Avoidance

ORCA (Optimal Reciprocal Collision Avoidance) algorithm (Van et al. 2011) inherits the idea of collision detection in velocity space from the VO algorithm and incorporates the reciprocal measures of RVO.

In scenarios where there is a large number of agents in space, calculating the VO for each agent can be computationally complex. ORCA algorithm simplifies this problem by formulating it as a low-dimensional linear programming task. The basic idea behind ORCA is to compute a set of velocity constraints for each agent, taking into account its current position, velocity, and desired velocity. These constraints define a velocity obstacle, which represents the set of velocities that would result in a collision if the agent were to maintain them. To find a collision-free velocity, ORCA formulates an optimization problem that aims to minimize the deviation from the desired velocity while ensuring that the agent stays within the velocity obstacle. This optimization problem is solved by finding the optimal linear combination of velocity constraints that satisfies the collision avoidance condition. By doing so, ORCA effectively navigates agents through space, avoiding potential collisions.

By incorporating the concept of relativized coordinates from the VO algorithm, the ORCA algorithm introduces the notion of a time window to further formalize the coordinate system of the velocity domain. To achieve this, the ORCA algorithm divides the center and radius of the previous disk $\hat{B}$, by a time window denoted as τ. Consequently, if the current relative velocity $\mathbf{v}_{A,B}$ lies outside of the truncated cone formed by extending two rays from the origin to the disk with centers at $(\mathbf{p}_B - \mathbf{p}_A)/\tau$ and a radius of $(r_A + r_B)/\tau$, it is guaranteed that objects A and B will not collide if they maintain their current velocities within the time window τ.

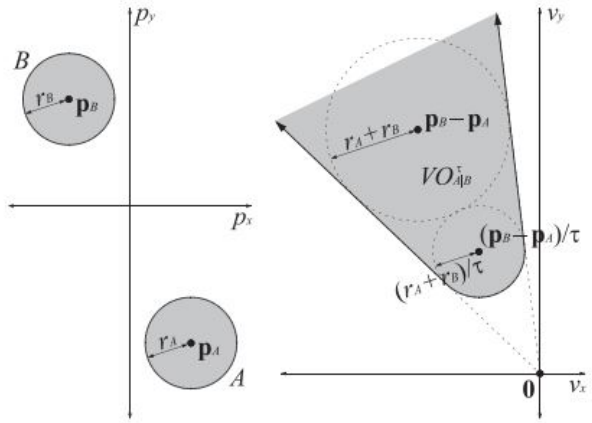


Figure 9

Now the velocity obstacle representing the truncated cone is defined as:

$$VO^{\tau}_{AB} = \{\mathbf{v} | \exists t \in [0, \tau], t\mathbf{v} \in D(\mathbf{p}_B - \mathbf{p}_a, r_A + r_B)\},$$

where VO_{AB} represents the truncated cone that defines the permissible range of velocities for each agent, and $D(\mathbf{p}, r) = \{\mathbf{q} || \mathbf{q} - \mathbf{p}|| < r\}$. By defining the velocity obstacle in this way, it provides sufficient conditions for each agent to be collision-free for at least within this fixed time window in the future.

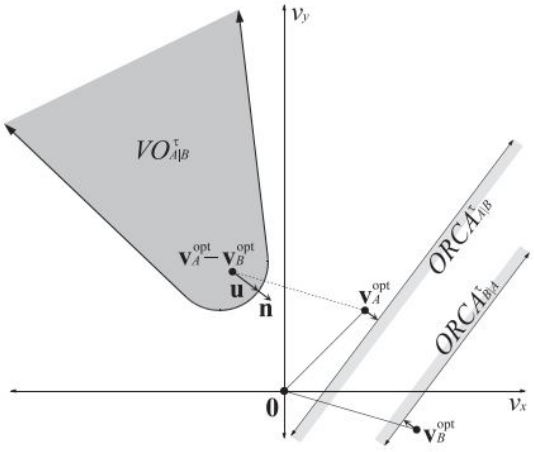


Figure 10

To construct $ORCA^{\tau}_{A|B}$ geometrically, we can follow these steps. Refer to Fig. 10 for visual

representation. Let $\mathbf{u}$ be the vector from $\mathbf{v}_A^{\text{opt}} - \mathbf{v}_B^{\text{opt}}$ to the closest point on the boundary of the truncated cone which representing VO_{AB}:

$$\mathbf{u} = \left(\underset{\mathbf{v} \in \partial VO^{\tau}_{A|B}}{\arg\min} ||\mathbf{v} - \left(\mathbf{v}_A^{\text{opt}} - \mathbf{v}_B^{\text{opt}}\right)|| \right) - \left(\mathbf{v}_A^{\text{opt}} - \mathbf{v}_B^{\text{opt}}\right).$$

Additionally, let $\mathbf{n}$ be the outward normal of the boundary of $VO^{\tau}_{A|B}$ at the point $\mathbf{v}_A^{\text{opt}} - \mathbf{v}_B^{\text{opt}} + \mathbf{u}$. Therefore, $\mathbf{u}$ represents the minimum adjustment required to the relative velocity of agents A and B to avoid a collision within a time window of τ. Inhering the idea from RVO, the responsibility for collision avoidance is shared among the agents. Agent A adjusts its velocity by $\frac{1}{2}\mathbf{u}$ and assumes that agent B takes care of the other half. Consequently, the set $ORCA^{\tau}_{A|B}$, which represents the permissible velocities for agent A, is defined as the half-plane pointing in the direction of $\mathbf{n}$ starting from the point $\mathbf{v}_A^{\text{opt}} + \frac{1}{2}\mathbf{u}$. More formally, we have:

$$ORCA^{\tau}_{A|B} = \left\{ \mathbf{v} \mid \left(\mathbf{v} - \left(\mathbf{v}_A^{\text{opt}} + \frac{1}{2}\mathbf{u} \right) \right) \cdot \mathbf{n} \geq 0 \right\}$$

Similarly as VO, in scenario where collision avoidance with multiple agents are considered, we take the union of all the individual *ORCAs*:

$$ORCA = \cup_{i=1}^{m} ORCA_{A|B_i}$$

To calculate the new velocity $\mathbf{v}_A^{new}$ efficiently, linear programming can be employed, as illustrated in Figure 11. The region $ORCA^{\tau}_{A|B}$ can be considered as a convex area, defined by linear constraints derived from the half-planes of permissible velocities concerning each of the other agents. The optimization objective is to minimize the distance to the preferred velocity $\mathbf{v}_A^{pref}$. Although this objective function may appear quadratic, it still retains the characteristics of linear programming and can be solved using techniques like incremental linear programming.

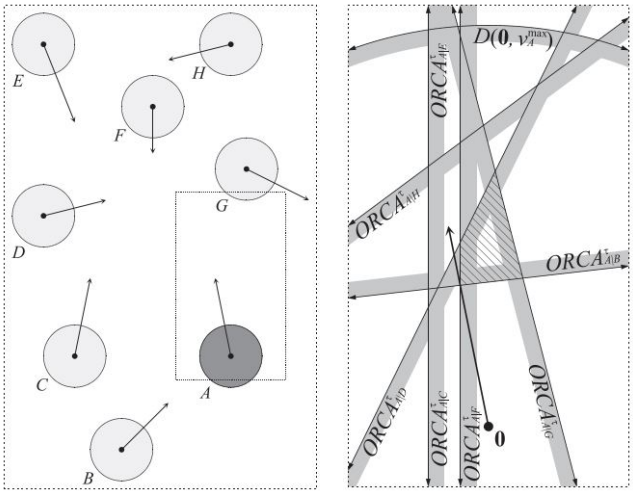


Figure 11

By formulating the problem in this manner, the ORCA algorithm takes advantage of linear programming's computational efficiency to determine the new velocity $\mathbf{v}_A^{new}$ for agent A. The convex nature of $ORCA^{\tau}_{A|B}$, combined with linear constraints and an optimization objective, allows for an effective and practical solution to be obtained.

4 Model Implementation

4.1 Construction of Workspace and Build a Roadmap

The workspace is confined to a two-dimensional area with x ranging from 0 to 10 and y ranging from 0 to 10. Users have the ability to define obstacles, which are polygons. The position of the vertices of these polygons is determined by the user clicking on the mouse.

This code prompts the user to enter the number of obstacles they want to define in the workspace. It then proceeds to obtain the number of vertices for each obstacle and their respective positions by taking user input.

```
n = input('Please enter the number of obstacles: ');
    % Obtain the number and vertex of obstacles through user input
    for i=1:n
        n_i = input(strcat('Please enter the number of vertices of
           the no.', num2str(i),' obstacle: '));
        obstacle = zeros(n_i,2);
        for j=1:n_i
            [x,y] = ginput(1);
            hold on;
            plot(x,y,'b.');
            obstacle(j,:) = [x,y];
        end
        hold on;
        plot(polyshape(obstacle(:,1),obstacle(:,2)),'FaceColor','b'
           );
        obstacles =[obstacles;obstacle];
    end
```

Inside the for loop, the code initializes a matrix called "obstacle" with dimensions $n_i x 2$, where n_i is the number of vertices for the current obstacle. It then iterates through each vertex, using the ginput function to capture the x and y coordinates of the mouse click. Then the obstacle matrix is updated with the current vertex coordinates. This process is repeated for each obstacle specified by the user, resulting in the complete definition and visualization of the obstacles in the workspace.

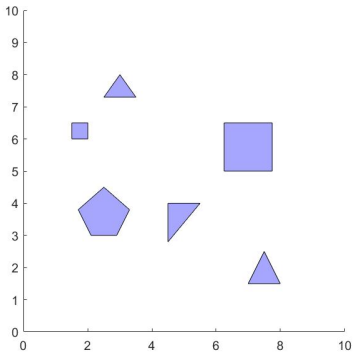


Figure 12: An example of a users-defined workspace

Then, we utilize approximate cell decomposition, specifically quadtree decomposition, to decompose the C-space and build a roadmap out of it, using a graph structure to simplify and abstract the C-space.

```
% 4--------3
% |        |
% |        |
% 1--------2

% g----h----i
% |    |    |
% d----e----f
% |    |    |
% a----b----c

global obstacles; % Access the global variable 'obstacles'
stack = []; % Initialize a stack to store cells for decomposition.
stack = [stack;{cell}]; % Access the global variable 'obstacles'

% Perform quadtree decomposition until the stack is empty.
while ~isempty(stack)
    % Extract the corner points of the current cell from the top of
        the stack.
    x1 = stack{end}(1,1); y1 = stack{end}(1,2);
    x2 = stack{end}(2,1); y2 = stack{end}(2,2);
    x3 = stack{end}(3,1); y3 = stack{end}(3,2);
    x4 = stack{end}(4,1); y4 = stack{end}(4,2);
    stack(end)=[]; % Remove the current cell from the stack.

    % Calculate the corner points of the four sub-cells.
    a = [x1 y1];
    b = [(x1+x2)/2 y1];
    c = [x2 y2];
    d = [x1 (y2+y3)/2];
    e = [(x1+x2)/2 (y2+y3)/2];
    f = [x2 (y2+y3)/2];
    g = [x4 y4];
    h = [(x1+x2)/2 y4];
    i = [x3 y3];

    if y3 - y2 > 0.5 % Check if the height of the current cell is
       greater than the threshold.
        % Plot lines to represent the boundaries between the sub-
           cells.
        plot([h(1),b(1)],[h(2),b(2)],'c');
        plot([d(1),f(1)],[d(2),f(2)],'c');

        % If a sub-cell is occupied, add it to the stack for
           further decomposition.
        cell_1 = [a;b;e;d];
```

```
        occ = cell_checker(cell_1);
        if occ
            stack = [stack;{cell_1}];
        end
        cell_2 = [b;c;f;e];
        occ = cell_checker(cell_2);
        if occ
            stack = [stack;{cell_2}];
        end
        cell_3 = [e;f;i;h];
        occ = cell_checker(cell_3);
        if occ
            stack = [stack;{cell_3}];
        end
        cell_4 = [d;e;h;g];
        occ = cell_checker(cell_4);
        if occ
            stack = [stack;{cell_4}];
        end
    end
```

The function begins by initializing a stack data structure and adding the first element, which is the entire workspace represented by the four corners of it. Then, a while loop is entered, which iterates until the stack is empty. Inside the loop, a cell is extracted from the top of the stack, and subsequently removed from the stack. The code proceeds to decompose the extracted cell into four sub-cells. As shown in the comments of the code, nine corner points produced by the decomposition are denoted as *a* to *i* sequentially. For each sub-cell, the index order of each corner point is also given in the comments at the top of the code. If the height of the sub-cells (y3 - y2) are greater than a precision threshold, which is set as 0.5, the code plots lines to represent the boundaries between the sub-cells. Following this, the code checks each sub-cell for occupancy by invoking the '*cell_checker*' function. If a sub-cell is found to be MIXED, it is added to the stack for further decomposition. The process continues until the stack is empty, resulting in a complete quadtree decomposition of the initial cell.

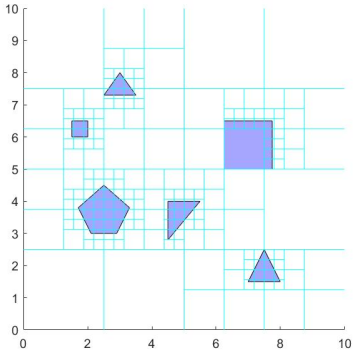


Figure 13: The result of quadtree decomposition

The next step is to generate a graph based on the set of EMPTY cells. The graph represents

the connections between the EMPTY cells.

```
% Loop through each empty cell
for i = 1:len
    e_cell = empty{i};
    center = [(e_cell(1,1)+e_cell(2,1))/2 (e_cell(1,2)+e_cell(4,2))/2];
    graph(i,1) = {center};
    % Expand each cell slightly so that adjacent cells have overlapping parts
    e_cell(1,:) = e_cell(1,:)-0.01;
    e_cell(2,1) = e_cell(2,1)+0.01;e_cell(2,2) = e_cell(2,2)-0.01;
    e_cell(3,:) = e_cell(3,:)+0.01;
    e_cell(4,1) = e_cell(4,1)-0.01;e_cell(4,2) = e_cell(4,2)+0.01;
    poly = [poly; {polyshape(e_cell(:,1),e_cell(:,2))}];
end

% Loop through each cell in the graph and check for intersections
for i = 1:len-1
    plot(graph{i,1}(1),graph{i,1}(2),'.','MarkerSize',8,'MarkerEdgeColor','k');
    j = i+1;
    while j <= len
        t = intersect(poly{i},poly{j});
        if t.NumRegions
            graph{i,2} = [graph{i,2};j];
            graph{j,2} = [graph{j,2};i];
            plot([graph{i,1}(1),graph{j,1}(1)],[graph{i,1}(2),graph{j,1}(2)],'Color',[0.5 0.5 0.5],'LineWidth',0.5);
            d = norm(graph{i,1}-graph{j,1});
            graph{i,3} = [graph{i,3};d];
            graph{j,3} = [graph{j,3};d];
        end
        j = j+1;
    end
end
```

As shown in the above code, it loops through each empty cell. For each empty cell, it calculates the center coordinates and stores them in the graph array in the first column. Then, it slightly expands each cell so that adjacent cells have overlapping parts. This is done to facilitate intersection checking later. The expanded cells are used to create polygon objects, which are stored in the poly variable. Next, it loops through each cell in the graph and checks for intersections. If two cells intersect, their indices are stored in the second column of the graph array. It plots a line segment connecting the intersecting cells representing the edge of the graph. Additionally, it calculates and stores the distance between the centers of the two cells in the graph array in the first column. Result of graph building is shown in Fig. 14.

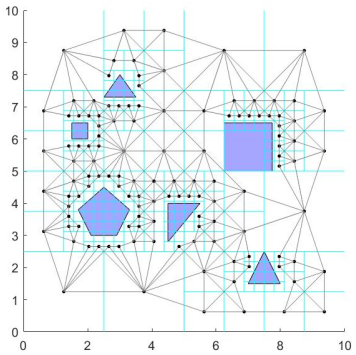


Figure 14: The built graph

4.2 Path Planning using Dijkstra

After the configuration space is abstracted into a graph, Dijkstra, as a graph search algorithm, can be utilized to find a path from the starting point to the target point within the configuration space.

We begin by defining the output variable 'path', which will be used to store the resulting shortest path. We also create the variable 'unseenNodes' with n rows and 2 columns. The first column is initialized with values ranging from 1 to n, representing the indices of each cell. The second column is initially set to infinity. Instead of directly using the number of rows, we index the cells based on the elements in the first column. We then locate the cell that contains the starting point and obtain its index, set the value in the second column of the corresponding row in 'unseenNodes' to 0, indicating the distance from the starting point to that cell. Additionally, fill the index-th value of the variable shortest_distance variable as 0.

```
while ~isempty(unseenNodes)
    [~,I] = min(unseenNodes(:,2)); % sort all the distance in
       unseen nodes
    i = unseenNodes(I,1); % get the index of the minimun node in
       graph
    unseenNodes(I,:) = []; % delete this node from unseenNodes
    n_child = size(graph{i,2},1); % get the number of childs of
       this node
    for c = 1:n_child
        child = graph{i,2}(c);
        new = shortest_distance(i) + graph{i,3}(c);
        if shortest_distance(child) > new
            shortest_distance(child) = new;
            c_i = find(unseenNodes(:,1)==child);
            unseenNodes(c_i,2) = new;
            path(child) = i;
        end
    end
end
```

Next, it enters a while loop shown above that iterates until 'unseenNodes' is empty. In each

iteration, it finds the node with the minimum distance in ‘unseenNodes‘, removes it from ‘unseenNodes‘, and traverses all its child nodes. For each child node, it calculates the new distance from the starting point to the child node through the current node. If the new distance is smaller than the current shortest distance of the child node, it updates the shortest distance of the child node, updates the distance in ‘unseenNodes‘, and stores the index of the current node in the ‘path‘ variable, indicating that the child node can be reached through the current node. When ‘unseenNodes‘ becomes empty, the shortest path to all nodes has been computed, and it is stored in the ‘path‘ variable.

```
i = path(target_cell);
fastest_connect = [target_cell;fastest_connect];
while i ~= start_cell
    fastest_connect = [i;fastest_connect];
    i = path(i);
end
fastest_connect = [i;fastest_connect];
```

The code above backtracks the shortest path using the path matrix *path* and the index of the target cell. It starts from the target cell and iteratively traces back, storing the indices of the passed cells in the *fastest_connect* array. Then, it retrieves the center point coordinates of each cell from the graph array and stores the key points along the path in the *pivot_points* array to obtain the complete path.

Finally, the code can visualize the path using the plotting code to plots the lines between the key points as red lines to display the entire path:

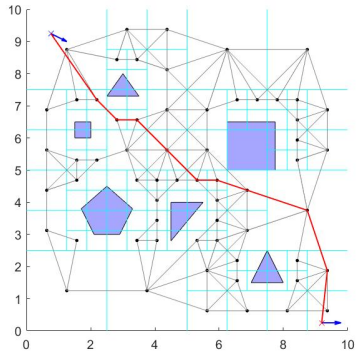


Figure 15: The found path (red lines).

4.3 Motion planing based on Minimum Jerk

Next, given a set of waypoints output by path planning algorithm, a minimum jerk trajectory will be generated while minimizing the rate of change of acceleration.

We start by defining the jerk order K (which is the 3rd derivative) and calculating the polynomial order n_order based on the jerk order. It also calculates the number of segments in the trajectory M and the dimension of the matrix $Q(N)$. And, initializes empty arrays *Pos* and *Vel* to store the positions and velocities of the trajectory, which are also the final output from this mumimum jerk algorithm.

```
K = 3; % jerk is the 3rd derivative
n_order = 2 * K - 1; % Polynomial order
M = length(path) - 1; % The number of segments of the trajectory
N = M * (n_order + 1); % The dimension of matrix Q
```

We then calculates the distance between each consecutive point in the path and normalizes it to obtain relative times. These relative times are then used to generate a time vector T that represents the time intervals between points. This is followed by iterating over each segment of the trajectory ($k = 0 : M - 1$) and generating a set of time points t within each segment. These time points are evenly spaced within the interval defined by $T(k + 1)$ and $T(k + 2)$.

```
distance = sqrt(sum(diff(path).^2, 2));
relative_times = cumsum(distance / sum(distance)) * 20;
T = [0;relative_times];
for k = 0:M-1
    t = linspace(T(k+1)+(T(k + 2)-T(k+1))/20, T(k + 2), 20);
    Pos = [Pos, t];
    Vel = [Vel, t];
end
```

Finally, we use the function "quadprog", which is a built-in optimization function in Matlab, to solve the quadratic programming problems and get the coefficient matrix *p_coff*. Within each segment, the code calculates the position and velocity at each time point using the polynomial coefficients *p_coff* obtained from the quadratic programming optimization problem. The position is calculated using a polynomial function of degree 5, and the velocity is calculated using the derivative of the position function. These are calculated independently for the x and y axes.

```
for d = 1:2
    x = path(:,d); % Extract x or y coordinates
    Q = zeros(N, N); % Initialize Q matrix
    for k = 1:M
        Qk = getQk(T(k), T(k + 1)); % Calculate Qk matrix for each
           segment
        Q(6 * (k - 1) + 1:6 * k, 6 * (k - 1) + 1:6 * k) = Qk; %
           Assign Qk to Q matrix
    end
    Q = 2 * Q; % The standard objective function is 1/2xTQx + qTx,
       so Q need to multiply by 2

    A0 = zeros(2 * K + M - 1, N);
    b0 = zeros(1,size(A0,1));
    % Add the state constraints for the first and last points (
       including position, velocity, acceleration)
    for k = 0:K-1
        for i = k:5
            c = 1;
            for j = 0:k-1
                c = c * (i - j);
            end
```

```
            A0(1 + k*2, i+1) = c * T(1)^(i - k);
            A0(2 + k*2, (M - 1) * 6 +i+1) = c * T(M+1)^(i - k);
        end
    end
    b0(1) = x(1); % Set initial position constraint
    b0(2) = x(M + 1); % Set final position constraint
    % Add an initial position constraint for each trajectory
    for m = 1:M-1
        for i = 0:5
            A0(6 + m, m * 6 + i+1) = T(m + 1)^i;
        end
        b0(6 + m) = x(m + 1);
    end
    A1 = zeros((M - 1) * 3, N);
    b1 = zeros(1,size(A1,1));
    for m = 0:M - 2
        for k = 0:2
            for i = 0:5
                c = 1;
                for j = 0:k-1
                    c = c * (i - j);
                end
                index = m * 3 + k;
                A1(index+1, m * 6 + i+1) = c * T(m + 2)^(i - k);
                A1(index+1, (m + 1)* 6 + i+1) = -c * T(m + 2)^(i -
                   k);
            end
        end
    end
    A = [A0; A1];
    b = [b0, b1];
    p_coff = quadprog(Q, zeros(1,N), zeros(size(A)),zeros(size(b)),
       A, b);
    pos = [];
    vel = [];
    for k = 0:M-1
        t = linspace(T(k+1)+(T(k + 2)-T(k+1))/20, T(k + 2), 20);
        t_pos = [t.^0; t.^1; t.^2; t.^3; t.^4; t.^5];
        t_vel = [t*0; t.^0; 2 * t.^1; 3 * t.^2; 4 * t.^3; 5 * t
           .^4];
        coef = p_coff(k*6+1 : (k+1)*6);
        p = coef' * t_pos;
        v = coef' * t_vel;
        pos = [pos, p];
        vel = [vel, v];
    end
    Pos = [Pos; pos];
    Vel = [Vel; vel];
end
```

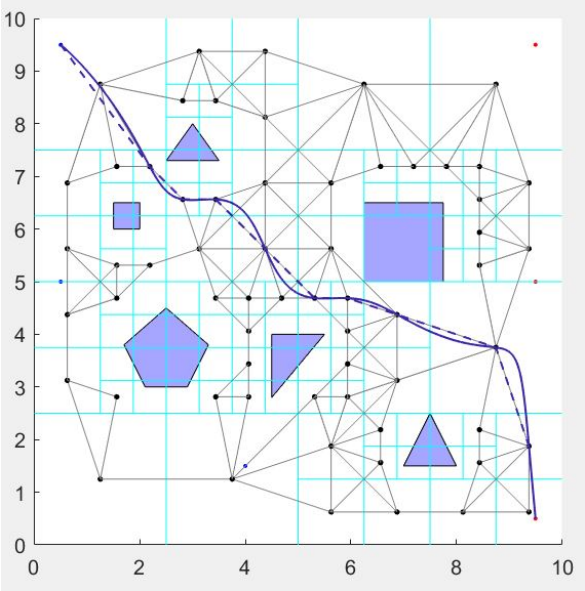


Figure 16: The smooth path after motion planning (solid blue lines).

4.4 Collision Avoidance based on Optimal Reciprocal Collision Avoidance

ORCA is a low-level motion planning algorithm that is continuously called during the execution of motion. After obtaining the trajectories of each agent through the minimum jerk algorithm, the agents start to move. Before assigning velocities to each agent in each frame, the ORCA algorithm is called to check if this set of velocities will cause any conflicts between agents in a short term. If there are no conflicts, ORCA outputs the original velocities and the agents move accordingly. If conflicts are detected, ORCA modifies the velocities to avoid the conflicts.

ORCA begins by transforming the coordinate system into the velocity domain, determining the position vector p_hat_B and the speed vector r_hat_B of other agents with respect to the current agent. The difference in speed between agents v_AB is then computed, and the function FindUN is called to calculate the adjustment speed $\mathbf{u}$ and its direction normal vector $\mathbf{n}$.

```
p_hat_B = (oth_Pos(i,:) - Pos)/deltaT;
r_hat_B = 2*r/deltaT;
v_AB = Vel - oth_Vel(i,:);

[c,u,n] = FindUN(p_hat_B',r_hat_B,v_AB');
```

In the FindUN function, projection is performed within a truncated cone following the steps of the ORCA algorithm. The truncated cone is bounded by a portion of the circumference and two rays (depicted as the red part in Fig. 17). To determine whether to project onto the disk or the ray, we can check the line connecting the velocity and the center of the circle intersects the circumference or the ray. For simplicity, let $\mathbf{v}$ represent $\frac{\mathbf{p}_B - \mathbf{p}_A}{\tau}$ and let r represent $\frac{r_A + r_B}{\tau}$ here. The orange dashed line represents the two tangential points connecting the circle and the ray. The intersection between the orange dashed line and the line from the origin to the center of the circle is referred to as the dividing_center point. The coordinates of the dividing_center are obtained using the concept of similar triangles. The $\mathbf{v}$ that does not cross the orange dashed

line falls into the subsequent circumferential projection category, while the **v** that crosses the orange dashed line falls into the subsequent radiographic projection category.

This is achieved by calculating the vector dividing_center (the black solid line in Fig. 17). The length of the green line segment in the figure can be expressed as $r * \cos\theta = |p| * \cos^2\theta$, where $\cos\theta = r/p$. The dividing_center can be obtained as $dividing\_center = p - p * \cos^2\theta = p(1 - \cos^2\theta)$. We then create a vector $v - dividing\_center$ and calculate the dot product to determine whether the angle between it and vector **v** is greater than 90 degrees. If it is greater than 90 degrees, it means that **v** does not cross the orange dashed line.

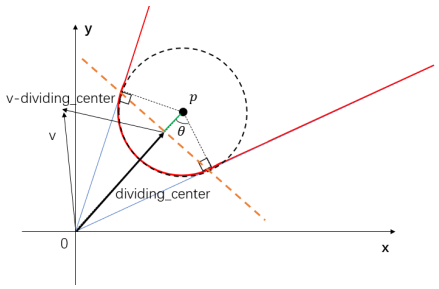


Figure 17

If **v** is located in front of the orange line, then the vector $\vec{w} = \vec{v} - \vec{p}$ is located in front of the orange line, then the vector **n**, which can be obtained by normalizing **w**: $\vec{u} = r * \vec{n} - \vec{w}$.

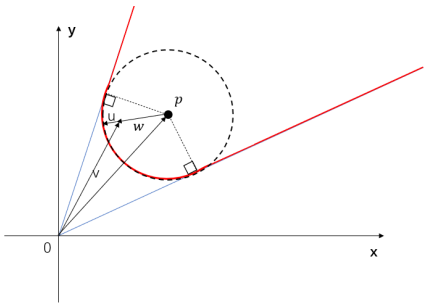


Figure 18

The functions mentioned above with regards to computing **u** and **n** are implemented using the following code snippet:

```
function [c,u,n] = FindUN(p,r,v)

% to decide whether to project onto the disk truncating the VO or
   onto the sides
dividing_center = p * (1 - r^2 / dot(p,p));
if dot(v - dividing_center, dividing_center) < 0 % project onto the
    disk
    w = v - p;
    n = w/norm(w);
    u = n*r - w; % if v is outside of VO, u is useless but compute
       it anyway
    c = sqrt(sum((p - v).^2))-r < 0 ; % will collide in the future
else % project onto the line
```

```
    leg_len = sqrt(dot(p,p)-r^2);
    direction_r = sign(det([p, v])) * r;
    rotation = [leg_len,-direction_r;direction_r,leg_len]/norm(p);
    side = rotation * p/norm(p); % already normalized
    n = [side(2);-side(1)];
    if direction_r > 0
        n = -n;
        c = det([v,side]) > 0;
    else
        c = det([v,side]) < 0;
    end
    u = side * dot(v,side) - v;
end
end
```

The half plane of ORCA is represented in Hessian normal form (point-normal form) (depicted by the red arrow in Fig. 19). The point $v + \frac{1}{2}u$ represents the starting point of the arrow, which we'll refer to as line.point, and the normal line is **n**, which we'll refer to as line.direction, indicating the direction of the half plane.

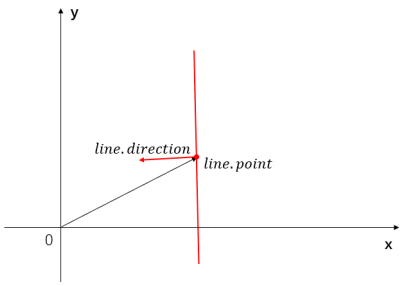


Figure 19

The area enclosed by the ORCA half plane of multiple obstacles is depicted as the dashed line in Fig. 20. If the velocity **v** is located outside this area, it is necessary to adjust **v** to the nearest $\mathbf{v}_{new}$ within the permissible region in order to avoid collision. This is not a typical linear programming problem because the cost function is not a linear equation. What is actually minimized is the length of velocity change $|v - v_{new}|$, represented by the dashed orange line "d" in the figure. Once "d" is determined, $v + d = v_{new}$ represents the final collision-free velocity.

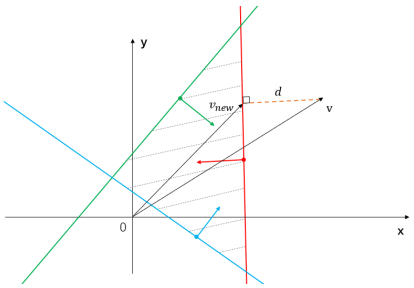


Figure 20

We utilize the incremental linear programming algorithm to adjust **v** in accordance with the conformance constraints, one by one, based on the half plane where **v** does not conform to the constraints. We identify the first half plane that renders **v** non-compliant with the constraint (the order of identification is not significant). This half plane is represented by the red dot normal in Fig. 21. Firstly, we calculate the distance between the intersection point of the plane and its corresponding normal line. This distance is divided into left_distance and right_distance. The point normals of the other planes are denoted as oth_point and oth_direction.

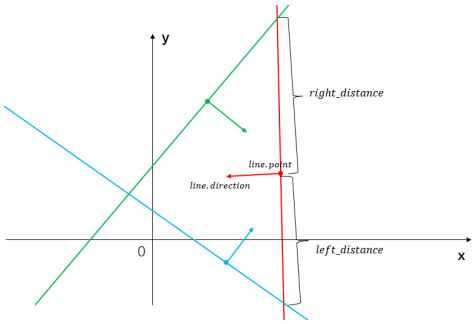


Figure 21

Let's consider the calculation of right_distance at the intersection point with the green plane as an example. First, we need to determine the distance from line_point to the green plane, which is represented by the dashed orange line in Fig. 22. This distance is equivalent to the length of the projection of line_point-oth_point onto the green dot normal, i.e point(oth_direction line_point-oth_point). Since both line_diretion and oth_direction are unit vetors, and $\sin\alpha$ is equal to $det(line\_direction, oth\_diretion)$, we can calculate right_distance as follows:

$$right\_distance = \frac{dot(oth\_direction, line\_point - oth\_point)}{det(line\_direction, oth\_diretion)}$$

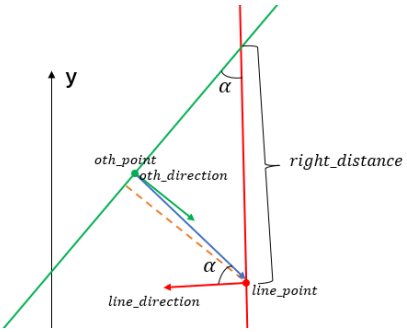


Figure 22

proj_len is calculated as the dot product of the unit vector obtained by rotating line_direction 90 degrees clockwise with **v** as illustrated in Fig. 25. The final step involves checking whether proj_len exceeds the boundaries set by right_distance and left_distance. To ensure that proj_len falls within these boundaries, it needs to be clamped or constrained to the range defined by right_distance and left_distance.

The functions mentioned above, pertaining to projection and incremental linear programming, implemented using the following code snippet:

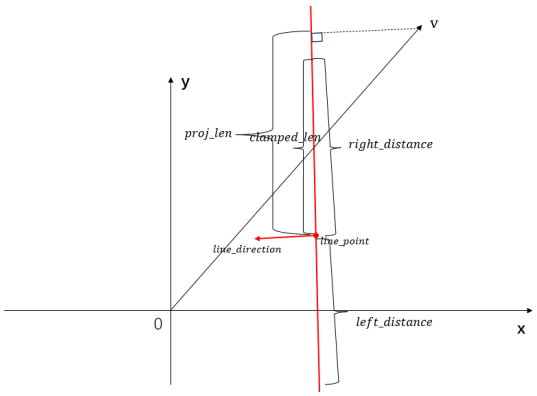


Figure 23

```
function [c,u,n] = FindUN(p,r,v)

% to decide whether to project onto the disk function [left_dist,
   right_dist] = line_halfplane_intersect(line_point,line_direction
   ,oth_point,oth_diretion)

left_dist = -inf;
right_dist = inf;
for i = 1:size(oth_point,1)
    num = dot(oth_diretion(i,:),line_point - oth_point(i,:));
    den = det([line_direction', oth_diretion']);
    offset = num / den;
    if den > 0
        % Point of intersection is to the right.
        right_dist = min(right_dist, offset);
    else
        % Point of intersection is to the left.
        left_dist = max(left_dist, offset);
    end
end
end

function Vel = point_line_project(line_point,line_direction, Vel,
   left_dist, right_dist)

line_dir = [line_direction(2), -line_direction(1)];
proj_len = dot(Vel - line_point, line_dir);
clamped_len = max(min(proj_len, right_dist), left_dist);
Vel = line_point + line_dir .* clamped_len;
end
```

The final trajectory after incorporating ORCA for obstacle avoidance is as follows:

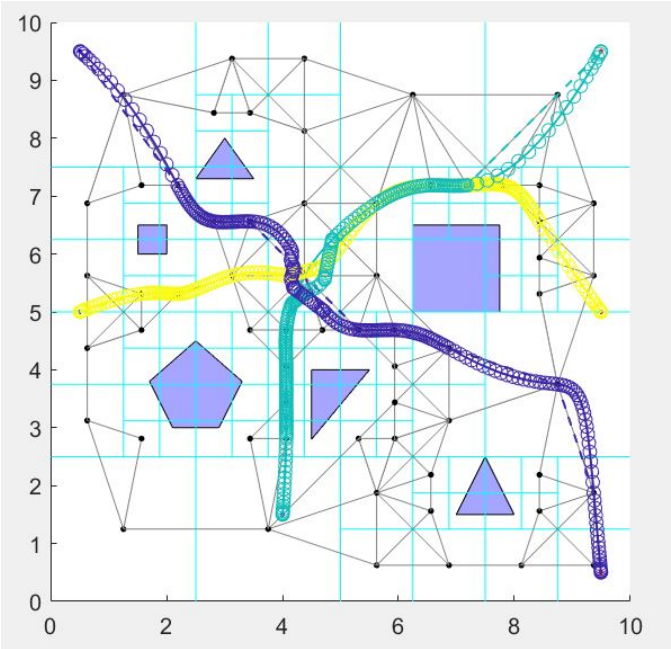


Figure 24

For reference, the trajectory without the inclusion of ORCA appears as follows, all three agents simultaneously move towards the middle part of the map, resulting in a collision:

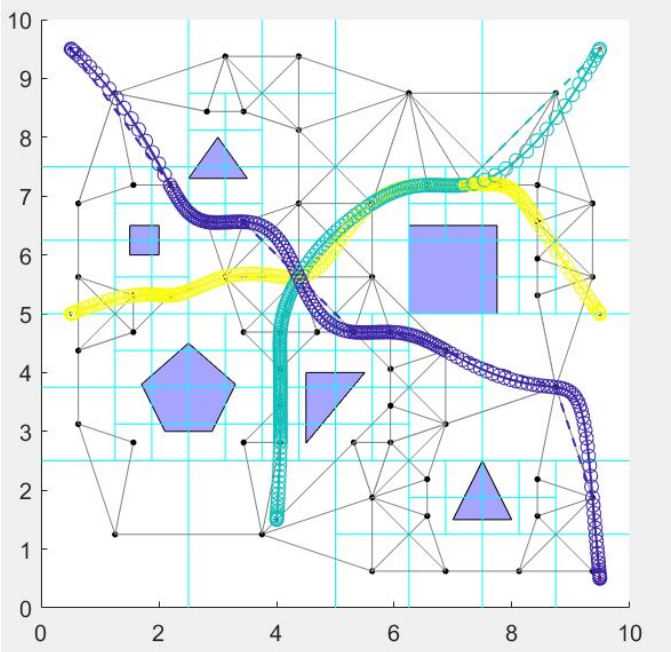


Figure 25

5 Results

The experiments conducted in this report utilized the model implementation described in the previous section. The results obtained from these experiments are presented below.

The path planning algorithm successfully calculated the shortest path from the starting point to the target point within the configuration space. The Dijkstra algorithm was used to search for a path in the graph representation of the configuration space. The resulting path was visualized by plotting lines between key points.

The minimum jerk algorithm was used to generate a smooth trajectory based on the waypoints obtained from the path planning algorithm. The algorithm minimized the rate of change of acceleration to ensure smooth motion. The resulting trajectory was visualized by plotting the positions and velocities of the trajectory.

The Optimal Reciprocal Collision Avoidance (ORCA) algorithm was applied to avoid collisions between multiple agents in real-time. The algorithm continuously checked for conflicts and adjusted the velocities of the agents to avoid collisions. It can be observed that the agents successfully avoided collisions and reached their respective target points.

Overall, the experiments demonstrated the effectiveness of the implemented model in achieving path planning, motion planning, and collision avoidance in a simulated environment. The model successfully generated optimal paths, smooth trajectories, and avoided collisions between multiple agents. These results showcase the potential of the model for various applications, such as autonomous navigation and robotics.

6 Introduction

Fiorini, P., & Shiller, Z. (1998). Motion planning in dynamic environments using velocity obstacles. The international journal of robotics research, 17(7), 760-772.

Frana, P. L., & Misa, T. J. (2010). An interview with edsger w. dijkstra. Communications of the ACM, 53(8), 41-47.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics, 4(2), 100-107.

Hsu, D., Kavraki, L. E., Latombe, J. C., Motwani, R., & Sorkin, S. (1998, March). On finding narrow passages with probabilistic roadmap planners. In Robotics: the algorithmic perspective: 1998 workshop on the algorithmic foundations of robotics (pp. 141-154).

Kyriakopoulos, K. J., & Saridis, G. N. (1988, April). Minimum jerk path generation. In Proceedings. 1988 IEEE international conference on robotics and automation (pp. 364-369). IEEE.

LaValle, S. M., Kuffner, J. J., & Donald, B. R. (2001). Rapidly-exploring random trees: Progress and prospects. Algorithmic and computational robotics: new directions, 5, 293-308.

Van den Berg, J., Lin, M., & Manocha, D. (2008, May). Reciprocal velocity obstacles for real-time multi-agent navigation. In 2008 IEEE international conference on robotics and automation (pp. 1928-1935). Ieee.

Van Den Berg, J., Guy, S. J., Lin, M., & Manocha, D. (2011, August). Reciprocal n-body collision avoidance. In Robotics Research: The 14th International Symposium ISRR (pp. 3-19). Berlin, Heidelberg: Springer Berlin Heidelberg.