الجمهورية الجزائرية الديمقراطية الشعبية

Peoples Democratic Republic of Algeria

وزارة التعليم العالي والبحث العلمي

Ministry of Higher Education and Scientific Research

جامعة سعد دحلب البليدة

SAAD DAHLAB University of BLIDA

كلية التكنولوجيا

Faculty of Technology

قسم الاوتوماتيك والالكتروتقني

Automatic and electrical engineering department

Master's thesis

Specialty : Automatique et informatique industrielle

Presented by

Larachi Feriel

and

Kessi Feriel

Designing a path planning system for an indoor mobile robot operating in an environment with partial knowledge

proposed by: Mr. Bennila Noureddin.

Academic year 2023-2024

Acknowledgements

We thank God the Almighty for having given us the health and the will to begin and

finish this memorial.

First of all, this work would not be as rich and could not have been possible

without the help of the promoter Mr. Bennila Noureddin, we thank him for the quality

of

his exceptional supervision, for his patience, his rigor and his availability during our

preparation of this brief.

Our thanks also go to all our teachers for their generosity and the great patience they

have shown despite their academic and professional responsibilities.

We also thank the jury members for their honor in judging our work.

We thank our parents for the motivation.

We thank our friends for all their sincere friendship over the last five years of study.

Finally, we thank all those who have contributed directly or indirectly to the

development of this work.

Dedication

To our dear mothers:

You have carried for us the care and effort for our education. No dedication

can express all the respect and love we have for you.

You have always trusted us. Find in this work the consolation and witness of patience.

To our dear fathers:

Despite the great responsibilities you assume in your work or as fathers of

families, you have always been close to us, to listen to us, to support us, to follow us

and to encourage us. May this work diminish your suffering and bring you happiness.

To our dear brothers and sisters:

Most of that work is for you. You have always been very helpful to us. We

thank you for all the good that each of you has done for us.

To our families.

To all our friends and classmates, may God preserve our friendship.

To all our teachers from Saad Dahleb Blida University

To all those who have trust us.

Abstract.

Autonomous Mobile Robots (AMRs) are robotic systems designed to navigate environments independently, without human intervention. With their increasing popularity and practical applications, there has been a rapid expansion in research and development in this field.

However, generating and executing efficient trajectory planning remains a significant challenge for these systems.

Our aim with this study is to contribute to the field of path planning by introducing two new methods: the first method is an extension of the Rapidly Exploring Random Tree Star (RRT*) algorithm, which we've named Limiting Tree Expansion, and the second method is a geometric approach inspired by mathematical operations. To validate these proposed methods, we implemented them in a simulation environment. The results demonstrate significant improvements in trajectory quality, with the Limiting Tree Expansion method outperforming the original RRT* algorithm. These promising findings underscore the potential of this optimization technique and pave the way for further research into parallelization to enhance the efficiency and effectiveness of these methods.

Keywords: Autonomous Mobile Robots (AMRs), path planning, Limited Tree Expansion, Geometric method

ملخص

الروبوتات المتنقلة الذاتية (ARMs) هي أنظمة روبوتية مصممة للتنقل في البيئات بشكل مستقل. دون التدخل البشري. ومع تزايد شعبيتها وتطبيقاتها العملية، حدث توسع سريع في البحث والتطوير في هذا المجال. ومع ذلك فإن إنشاء وتنفيذ تخطيط فعال للمسار لا يزال يمثل تحديًا كبيرًا لهذه الأنظمة.

هدفنا من هذه الدراسة هو المساهمة في مجال تخطيط المسار باستخدام طريقتين جديدتين: الأولى مطورة من خوارزمية Rapidly Exploring Random Tree Star (RRT*) قمنا بتسميتها ب Limited Tree Expansion. والطريقة الثانية هندسية مستوحاة من عمليات رياضية. ولتحقيق هذه الطرق المقترحة. قمنا بتنفيذها في بيئة المحاكاة. أظهرت النتائج تحسنا ملحوظا في جودة المسار. حيث تفوقت طريقة Limited Tree Expansionأداء خوارزمية *RRT الاصلية. تسلط هذه النتائج الضوء على إمكانيات التقنية التحسينية وتمهد الطريق لمزيد من البحث في مجال التوازن لتعزيز كفاءة وفعالية هذه الأساليب

الكلمات الرئيسية: الروبوتات المتنقلة الذاتية (ARMs)، تخطيط المسار،Limited Tree Expansion ، الطريقة الهندسية.

Résumer

Les robots mobiles autonomes (RMA) sont des systèmes robotiques conçus pour naviguer dans des environnements de manière indépendante, sans intervention humaine. Avec leur popularité croissante et leurs applications pratiques, il y a eu une expansion rapide de la recherche et du développement dans ce domaine. Cependant, la génération et l'exécution d'une planification de trajectoire efficace restent un défi majeur pour ces systèmes.

Notre objectif avec cette étude est de contribuer au domaine de la planification de trajectoires en introduisant deux nouvelles méthodes : la première méthode est une extension de l'algorithme Rapidly Exploring Random Tree Star (RRT*), que nous avons nommée Expansion Limitée de l'Arbre, et la deuxième méthode est une approche géométrique inspirée par des opérations mathématiques. Pour valider ces méthodes proposées, nous les avons mises en œuvre dans un environnement de simulation. Les résultats montrent des améliorations significatives en termes de qualité de trajectoire, la méthode d'Expansion Limitée de l'Arbre surpassant l'algorithme RRT* original. Ces résultats prometteurs soulignent le potentiel de cette technique d'optimisation et ouvrent la voie à de futures recherches sur la parallélisation visant à améliorer l'efficacité et l'efficience de ces méthodes.

Mots-clés : Robots Mobiles Autonomes, Planification de trajectoire, Expansion Limitée de l’Arbre, Méthode géométrique.

Acronyms and Abbreviations

AMR	Autonomous Mobile Robots
RRT	Rapidly Random Tree
DFS	Depth First search
BFS	Breadth First Search
UAVS	Unnamed Aerial Vehicles
UGVS	Unnamed Gond Vehicles
AUVS	Autonoms Undawater Vhicles
RRT*	Rapidly Random Tree Star
FC. RRT	Flight Cost-Based Rapidly Random Tree
FN-RRT	Fixed Node Rapidly Random Tree
VCS	Version Control Systèmes.
IDE	Integrated Development Environment.
2D	Two-Dimensional

Table of content

List of tables.

List of figures

General Introduction

The field of autonomous mobile robots (AMRs) is rapidly evolving, driven by advancements in technology and increasing demand for automation. These robotic systems are designed to navigate and perform tasks in various environments without human intervention, making them invaluable in numerous applications, from industrial automation to service robotics.

This thesis addresses two critical aspects of AMR navigation: path planning and obstacle avoidance. Path planning involves determining an optimal route from a starting point to a destination, while avoiding obstacles and ensuring efficient movement. Obstacle avoidance focuses on the robot's ability to detect and navigate around obstacles in real-time.

The thesis is structured into three chapters. The first chapter provides a general overview of robotics, with a focus on mobile robots. It classifies path planning methods into two main categories: search methods and construction methods. The search methods discussed include breadth-first search, depth-first search, Dijkstra's algorithm, and the A* algorithm. Construction methods such as cell decomposition, Voronoi diagrams, and visibility graphs are also explored. A comparative analysis of these methods highlights their respective strengths and weaknesses.

The second chapter explore obstacle avoidance, categorizing the techniques into reactive and deliberative methods. Reactive methods, including geographic techniques such as potential fields and algorithms like Bug 1, Bug 2, are discussed. Additionally, deliberative methods with a focus on the RRT* (Rapidly exploring Random Tree Star) algorithm are examined.

The third chapter presents the implementation, analysis, and results of the study. It introduces two new methods: the Limited tree Expansion of RRT* and a geometric method. The implementation of these methods in a simulation environment is detailed, followed by an analysis of their performance. The results demonstrate significant improvements in trajectory quality, surpassing the original RRT* algorithm. These findings underscore the potential of these optimization techniques and suggest directions for future research, including the exploration of parallelization to enhance efficiency and effectiveness.

This thesis aims to contribute to the field of autonomous navigation by providing innovative solutions to the challenges of path planning and obstacle avoidance, ultimately advancing the capabilities of autonomous mobile robots.

Chapter 1

Path planning

1.1 Introduction

Path planning is a critical aspect of robotics and computer science, involving the determination of a viable route from a start point to a destination while avoiding obstacles. This chapter explores key graph search algorithms, including Depth-First Search (DFS), Breadth-First Search (BFS), Dijkstra's Algorithm, and A* Search, which provide various strategies for navigating through graph representations of environments.

Additionally, we explore a graph construction technique like Rapidly Exploring Random Trees (RRT), Voronoi diagrams, visibility graphs, and cell decomposition methods, each offering unique approaches to modeling and solving path planning problems in different types of spaces.

Through these methods, we gain insight into both the theoretical foundations and practical applications of efficient and effective pathfinding.

1.2 Definition of Robotics

Robotic is the intersection of science, engineering and technology it generally refers to anything related to robots or resembling robots in characteristics or behavior. it is hard to give a precise definition. It encompasses the design, construction, operation, and application of robots, and use of machines (robots) to perform tasks done traditionally by human beings. as well as the study of their behavior and the integration of sensory feedback and control in machines. Robotic systems can vary widely in complexity and functionality, from simple industrial robots programmed to perform repetitive tasks to advanced humanoid robots capable of interacting with humans in complex environments. [1]

1.3 Classification of robotics

Robotics covers many different types of classifications which include fixed and mobile robots. Each type has distinct characteristics and applications, making them suitable for different tasks and environments. These two types of robots have very different working

environments and therefore require very different capabilities. [2]

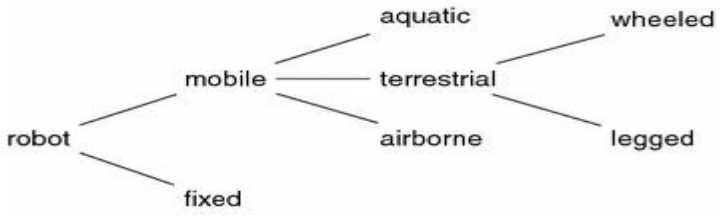


FIgure1. 1:Classification of robots by environment and mechanism interaction

1.3.1 Fixed robots

Fixed robots are stationary robots that are anchored in one location and do not move from their position. They are mostly industrial robotic manipulators that work in well-defined environments adapted for robots. Industrial robots perform specific repetitive tasks such soldering or painting parts in car manufacturing plants. With the improvement of sensors and devices for human-robot interaction, robotic manipulators are increasingly used in less controlled environment such as high-precision surgery.[2]

1.3.2 Mobile robots

Mobile robots are capable of moving through their environment. They can navigate from one location to another, often autonomously. Mobile robots can be classified in two ways: by the environment in which they work and by the device they use to move**.** Examples of different environment mobile robots include:

Polar robots that are designed to traverse icy, uneven environments.

Aerial robots, also known as unmanned aerial vehicles (UAVs) or drones, which fly through the air.

Landor home robots, or unmanned ground vehicles (UGVs), that navigate on dry land or within houses.

Underwater robots, or autonomous underwater vehicles (AUVs), that can direct themselves and travel through water.

Delivery and transportation mobile robots that are designed to move materials and supplies around a work environment. [3]

1.4 Navigation Robots

Navigation robots, a subset of mobile robots, are specifically designed to autonomously navigate through their environment, given partial knowledge about its environment and a goal positions or series of positions, navigation encompasses the ability of the robot to act based on its knowledge and sensor value so as to reach it goal positions as efficiently and as reliably as possible. Two capabilities for navigation are path planning and obstacles avoidance.

1.5 Path planning

Path planning lets an autonomous vehicle, or a robot find the shortest and most obstacle-free path from a start to goal state using different methods.[4]
There are two classifications of this methods:

Graph construction, where nodes are placed and connected vid edges.

Graph search, where the computation of an (optimal) solution is performed.

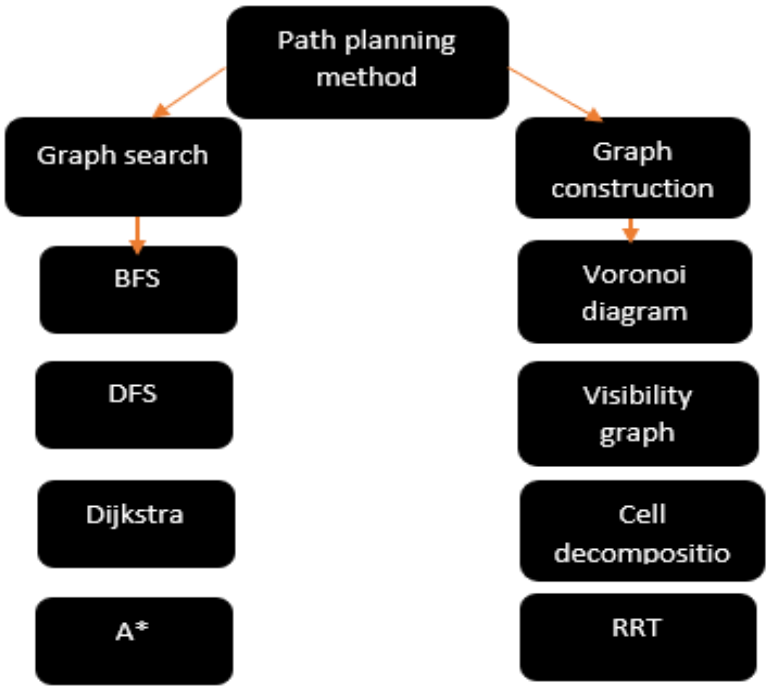


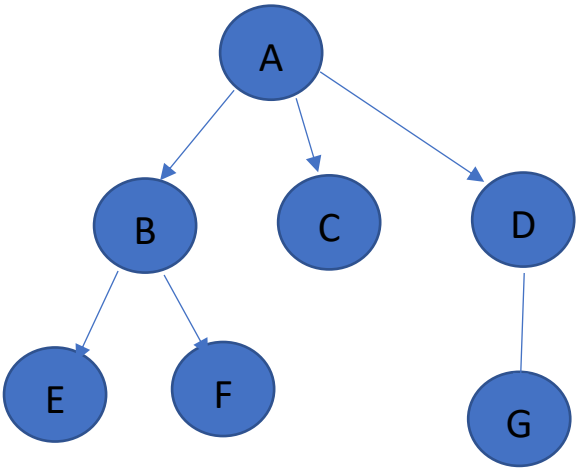
FIgure1. 2:path planning method classification schematic

1.5.1 Graph search method.

Graph search methods are like giving a robot a map with marked paths and helping it find the best way to reach a destination. Imagine each location on the map as a point, and the paths between them as lines. Graph search methods help the robot explore these paths efficiently to find the shortest or best route. They're like guiding the robot through a maze to reach the goal without getting lost. Some common graph search methods include Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's algorithm, and A* algorithm. Each method has its own way of searching through the map to find the optimal path for the robot to follow.[5]

1.5.1.1 Breadth-First Search (BFS)

Breadth-first search or the BFS algorithm systematically explores a graph by visiting all the neighboring nodes of a given level before moving to the next level. The core working principle of BFS is to use a queue to maintain a level-wise exploration order. Initially, the algorithm starts with a source node, enqueues its neighbors, and continues this process until all nodes have been visited. The breadth-first traversal strategy ensures that nodes are visited in increasing order of their distance from the source, allowing for finding the shortest path in unweighted graphs.[6]

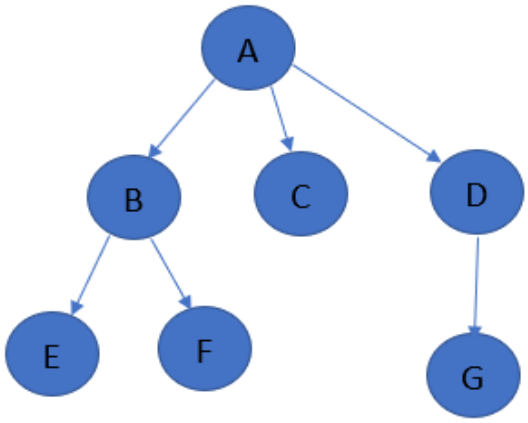


Visited	queue
	A
A	B C D
B	C D E F
C	D E F
D	E F G
E	F G
F	G
G	

FIgure1. 3:Breadth first traversal graph

1.5.1.2 Depth-First Search (DFS)

DFS Unlike BFS in which we explore the nodes breadthwise, in DFS we explore the nodes depth-wise. In DFS we use a stack data structure for storing the nodes being explored. The edges that lead us to unexplored nodes are called ‘discovery edges’ while the edges leading to already visited nodes are called ‘block edges.[7]

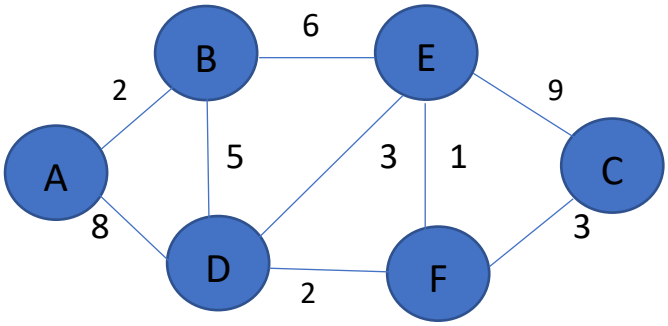


Visited	Stack data
	A
A	B C D
D	B C G
G	B C
C	B
B	E F
F	E
E	

FIgure1. 4:depth first traversal graph

1.5.1.3 Dijkstra algorithm

This algorithm is similar to breadth-first search, except that edge costs may assume any positive value and the search still guarantees solution optimality.
This algorithm expands nodes starting from the start similar to breadth-first search, except that the neighbors of the expanded node are placed in the heap and reordered according to their value, which corresponds to the cost.
Subsequently, the cheapest state on the heap (the top element after reordering) is extracted and expanded. This process continues until the goal node is expanded, or no more nodes remain on the heap. A solution can then be backtracked from the goal to the start. Due to reorder operations on the heap, the time complexity rises [8]



Nodes	Shortest distance	Previous Nodes
A	0	
B	2	A
C	12	F
D	7	B
E	8	B
F	9	D

FIgure1. 5:Dijkstra path finding graph

1.5.1.4 A* algorithm:

the A* search algorithm begins by expanding the start node and placing all of its neighboring nodes onto a priority queue (heap). Unlike Dijkstra's algorithm, which orders nodes based solely on the path cost, A* orders nodes according to a combined cost value, *f*, which includes both the path cost and the heuristic estimate of the remaining distance to the goal.
The combined cost value $f(n)$ for a node n is calculated as:

$$f(n)=g(n)+h(n)$$

where:

$g(n)$ is the actual cost from the start node to node n.

$h(n)$ is the heuristic estimated cost from node n to the goal.

At each step, A* selects the node with the lowest *f*-value from the priority queue. This node is then expanded, meaning its neighbors are evaluated and added to the queue if they haven't been processed or if a cheaper path to them is found.
The process continues, with nodes being selected, expanded, and evaluated based on their *f*-values, until the goal node is reached. Once the goal node is reached, the algorithm backtracks to reconstruct the path from the start node to the goal. [9]

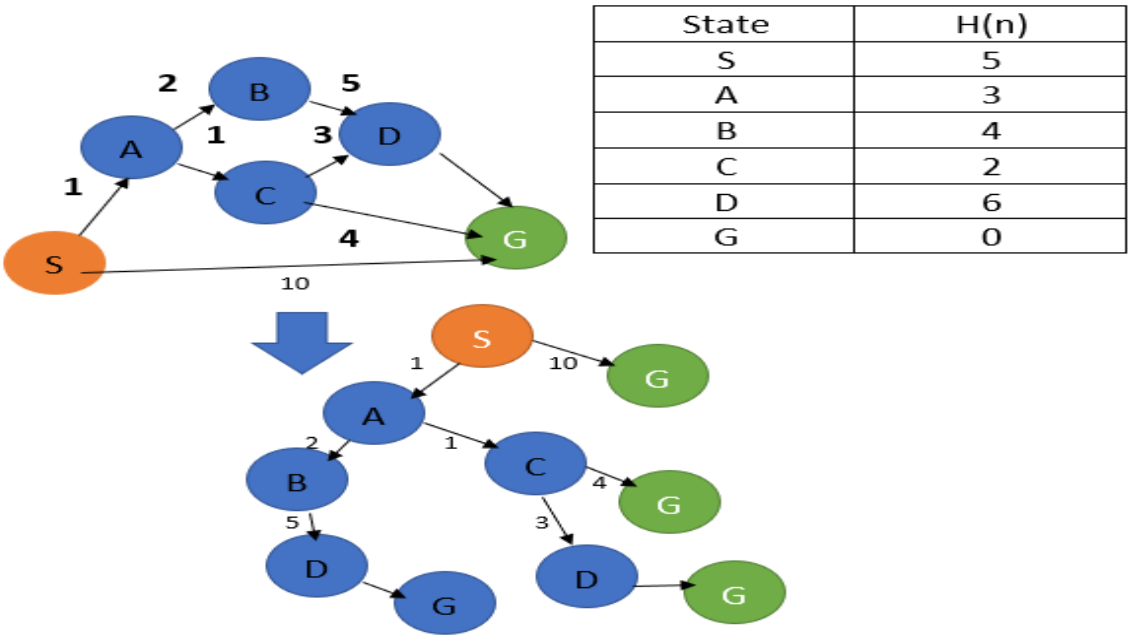


FIgure1. 6:A* Path finding graph.

1.5.2 Graph construction method.

1.5.2.1 Voronoi diagram

The Voronoi diagram algorithm operates by partitioning the environment into distinct regions based on the distances to obstacles. It constructs a graph where nodes represent obstacles, and edges denote lines of equidistant points between neighboring obstacles. These lines delineate Voronoi cells, which define the boundaries of regions within the environment. Leveraging these cells, the Voronoi diagram algorithm devises a path by linking the start and goal locations to cell boundaries, ultimately determining the path that traverses the fewest number of cells.[10]

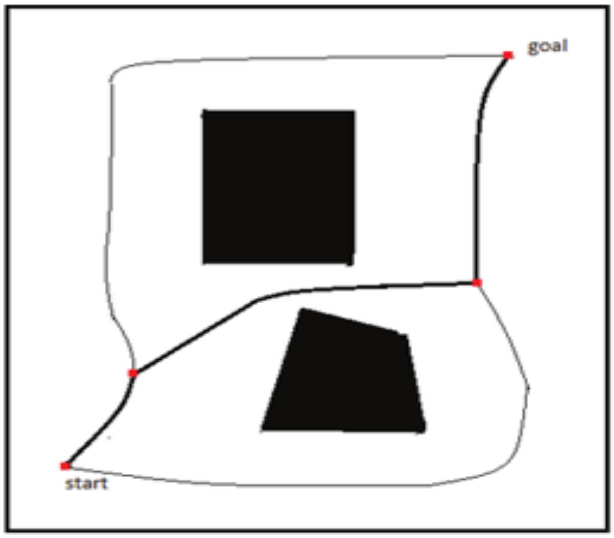


Figure1. 7:Voronoi diagram path finding.

1.5.2.2 Visibility graph

The visibility graph algorithm works by representing obstacles as vertices in a graph, and creating edges between vertices that are visible to each other. The start and goal points are added as vertices, and edges are added between them and any visible vertices. Then, a search algorithm is used to find the shortest path between the start and goal points in the graph [11].

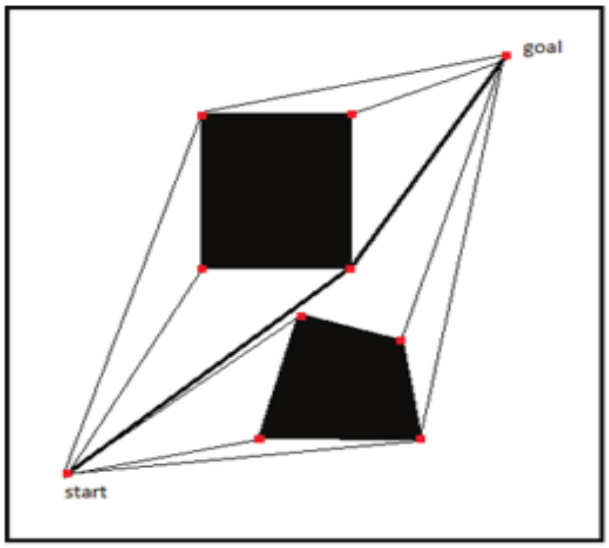


FIgure1. 8:visibility graph method

1.5.2.3 Cell decomposition

Exact cell decomposition is a lossless method, accurately dividing the environment into cells that are either fully free or fully occupied by obstacles. Approximate cell decomposition, on the other hand, uses a grid to represent the environment, where cells can be classified as free, occupied, or mixed, introducing some approximation.
In both methods, a graph is constructed to represent the connectivity between cells, where nodes correspond to the cells and edges represent possible transitions between adjacent cells. The boundaries of these cells are determined by critical geometric features, ensuring accurate representation in the case of exact decomposition.
The primary focus of cell decomposition is that the robot's specific position within a free cell is irrelevant. What matters is the robot's ability to move from one free cell to another. The complexity and computational efficiency of path planning depend on the density and complexity of obstacles in the environment, with more complex environments resulting in a greater number of cells and a more intricate connectivity graph.[12]

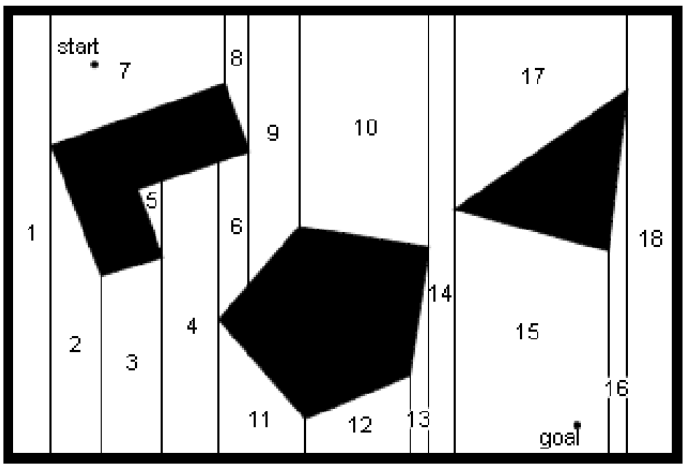


FIgure1. 9:cell decompositions method.

1.5.2.4 Rapidly Exploring Random Tree RRT

The Rapidly Exploring Random Tree (RRT) is an effective algorithm for robotics and path planning, especially useful in high-dimensional and complex environments. It begins by initializing the tree with the robot's starting position as the root node. The algorithm then randomly samples point within the search space. For each sample, it identifies the nearest node in the existing tree and extends a new node towards the sample point by a predefined step size, ensuring the path to this new node does not intersect any obstacles. If the path is collision-free, the new node is added to the tree. This process repeats until either a node reaches the goal region or a specified number of iterations is completed. RRT is particularly advantageous for its ability to efficiently explore large and complex spaces.[13]

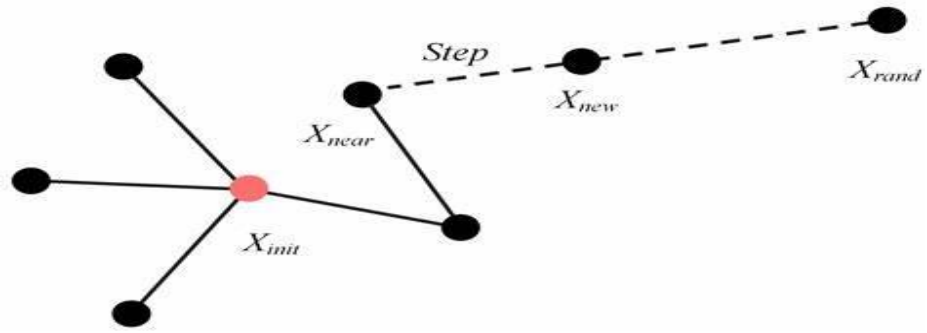


FIgure1. 10:schematic diagram of RRT expansion

1.6 Comparison

Method	Optimal	complete	For complex spaces	Memory efficient	Fast execution	Handles high dimensional spaces
Depth First search (DFS)	No	yes	No	yes	No	No
Breadth First search (BFS)	Yes	yes	No	No	No	No
Dijkstra algorithm	Yes	yes	No	No	No	No
A* search	yes	yes	No	No	Yes	No
RRT	No	yes	No	Yes	Yes	Yes
Cell Decomposition	yes	yes	No	No	No	No
Voronoi diagram	No	yes	No	Yes	Yes	Yes
Visibility graph	yes	yes	Yes	No	Yes	No

Table 1 1:Comparative table of the different method

When comparing path planning methods, we see that each has its own strengths and weaknesses. Some, like A* Search and Dijkstra's Algorithm, are great at finding the shortest path, but they might not work well in tricky spaces. Others, like Rapidly Exploring Random Trees (RRT), handle complex environments better, but they might not always find the absolute best path. Then there are methods like Voronoi Diagrams and Visibility Graphs, which are good at avoiding obstacles but might struggle in really complicated spaces. Picking the right method depends on what you need for your specific situation, like whether you want the shortest path or need to navigate a tricky environment. It's all about finding the right balance to get the job done well.

1.7 Conclusion

In this chapter, we've explored different ways to figure out the best path to get from one place to another, even when there are obstacles in the way.

We learned about simple methods like following paths until we reach the end (DFS) or looking at all possible paths at once (BFS).

We also discovered more advanced techniques like A* and Dijkstra's Algorithm, which are like super-smart ways to find the shortest route. Besides, we looked into how to make maps using methods like drawing lines between points (Voronoi diagrams) and connecting visible points without touching obstacles (visibility graphs). These methods are like special tools that help us understand the environment better.

And now, we're getting ready to dive deeper into a special version of one of these methods called RRT*. It's like adding extra powers to our path-planning skills, so we can handle even trickier spaces. As we keep learning and practicing, we'll become experts at finding our way through all sorts of places, whether it's a maze, a city, or even a virtual world!

2 Chapter 2

Obstacle Avoidance

2.1 Introduction

Obstacle avoidance is a crucial aspect of autonomous robotics, enabling robots to navigate complex environments safely and efficiently.

The methods for obstacle avoidance can be broadly classified into two categories: reactive and deliberative methods. Reactive methods, which involve real-time decision-making based on immediate sensor data, include geometric techniques such as potential fields and Bug algorithms**.**

Also, we will talk about Deliberative methods, which involve more complex planning strategies that consider the entire environment to generate an optimal path, include techniques such as RRT* and its variants.

This exploration of both reactive and deliberative methods will provide a comprehensive understanding of the current strategies in obstacle avoidance, setting the stage for further innovations in autonomous robotic navigation.

2.2 Obstacle avoidance

Obstacle avoidance refers to the techniques and algorithms used in robotics and autonomous systems to detect and navigate around obstacles in their environment. This capability is crucial for the safe and efficient operation of robots, drones, autonomous vehicles, and other systems that operate in dynamic or unstructured environments. They are two methods in obstacle avoidance are shown in figure 2.1.

[14]

.

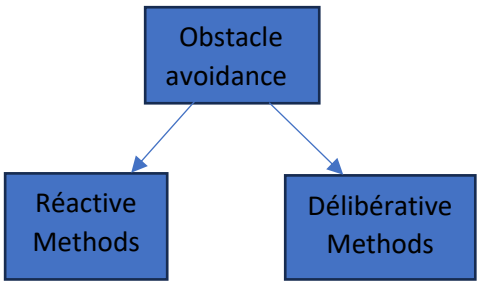


Figure2. 1:classification of obstacle avoidance method

.

2.3 Reactive Methods

These methods rely on immediate sensor data to make real-time decisions to avoid obstacles without a comprehensive plan of the entire environment. Geometric Methods fall under this category as they typically involve real-time adjustments to the robot's path based on the current obstacle layout.

2.3.1 Geometric Methods

are a prominent subset of reactive methods. They use geometric properties and spatial relationships to navigate around obstacles. Common geometric methods include.

2.3.1.1 Potential Fields:

In the artificial potential field, the robot motion is controlled by the attractive force and the repulsive force, the attractive force is generated by the distance and direction to the goal point, whereas the repulsive force is generated by the distance and direction to obstacles. The forces of the robot in the artificial potential field are shown in figure 2.2, where $F2$ is the repulsive force between the obstacle and the robot .$F1$ is the attractive force between the goal point and the robot, and F is the resultant force for controlling the robot. [15]

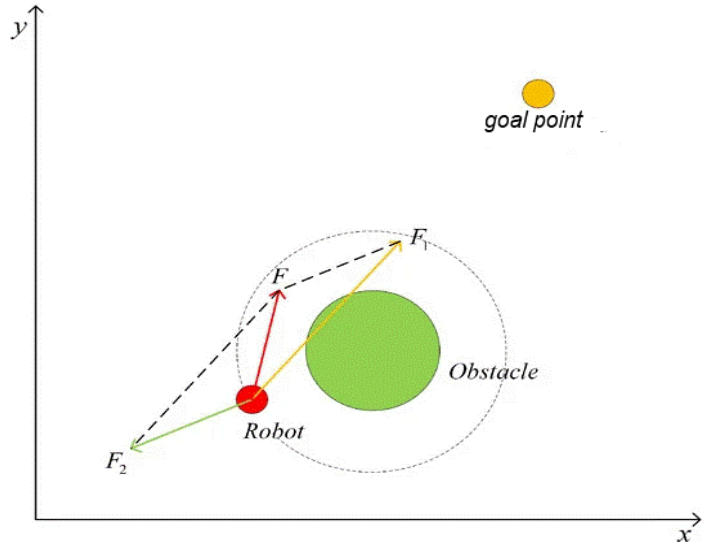


Figure2. 2:attractive force and repulsive force in artificial potential field.

2.3.1.2 Bugs Algorithms

These are simple, rule-based algorithms that handle obstacle avoidance by following the boundary of obstacles.

2.3.1.2.1 Bug 1 Algorithm

The primary goal of Bug1 is to navigate a robot from a starting point to a goal point while avoiding obstacles encountered along the way.

Detailed Steps**:**

-The robot sets off towards the goal along the straight-line path.
Upon hitting an obstacle at point, A (the hit point), the robot switches to boundary-following mode.
-The robot follows the obstacle boundary and continuously monitors its distance to the goal. As it does this, it identifies the point B (the leave point) on the boundary that is -closest to the goal.
-The robot completes one full circuit of the obstacle boundary, ensuring it explores the entire perimeter of the obstacle.
-After returning to the hit point, the robot moves directly to the leave point (point B), which is the closest point to the goal identified during the boundary following.
-From the leave point, the robot continues moving towards the goal. If it encounters another obstacle, the process repeats.[16]

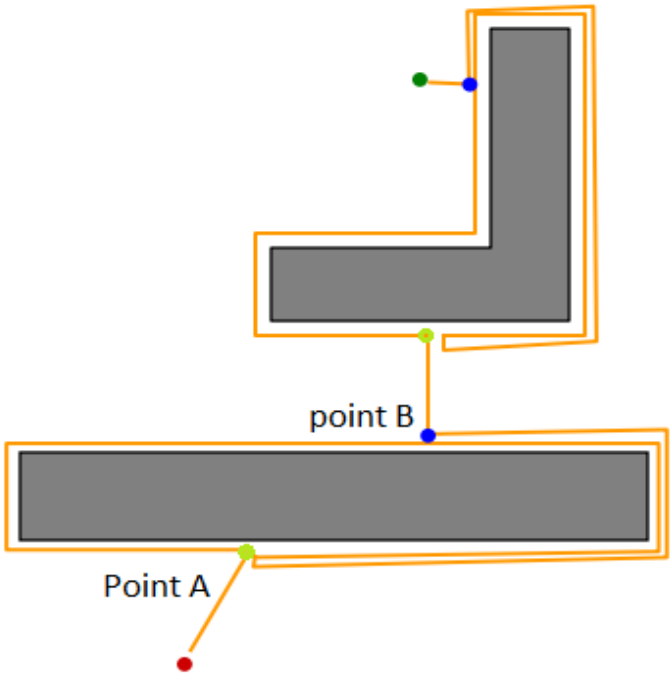


Figure2. 3:presentation of bug 1 algorithm.

Bug 2 is to enable a robot to navigate towards a goal while avoiding obstacles by following a straightforward strategy:
-The robot initially moves directly towards the goal along a straight line (called the m-line).
-When the robot encounters an obstacle, it marks the point of contact (hit point).
-The robot then follows the boundary of the obstacle in a fixed direction (clockwise or counterclockwise).
-The robot looks for an opportunity to return to the m-line and will leave the boundary when it finds a point on the m-line that is closer to the goal than the hit point.
-The process repeats until the robot reaches the goal or determines that the goal is unreachable.[16]

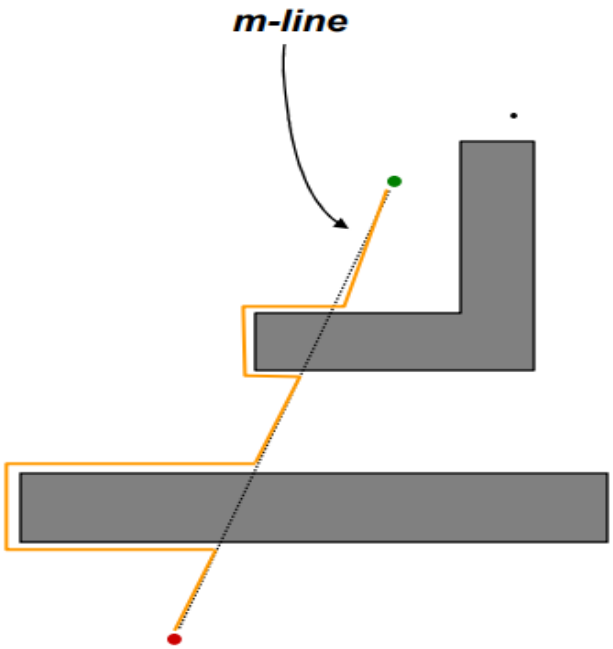


Figure2. 4:presentation of bug 2 Algorithm.

2.4 Deliberative Methods

These methods involve pre-planning a path by considering the entire environment or a significant portion of it. They often ensure global optimization and completeness' (Rapidly exploring Random Tree Star*)* is a deliberative method as it plans paths by considering the overall map and optimizes the path iteratively**.**
RRT* ensures that the path cost converges to the optimal solution as more samples are added, making it particularly effective for complex environments where global optimality is crucial.

2.4.1 RRT* star

RRT* is an optimized version of RRT. As the number of nodes approaches infinity, the RRT* algorithm will provide the shortest possible path to the goal. The basic principle of RRT* is the same as that of RRT, but two key additions to the algorithm yield significantly different results**.**
Firstly, RRT* records the distance traversed by each node relative to its parent node. This is referred to as the cost () of the node. Once the nearest node is found in the graph, a neighborhood of nodes within a fixed radius from the new node is examined. If a node with a lower cost () than the proximal node is found, the least expensive node replaces the proximal node.

The second difference added by RRT* is tree reconnection. Once a node has been connected to the least expensive neighbor, neighbors are re-examined. Neighbors are checked to see if rewiring them to the newly added node will reduce their cost. If the cost does indeed decrease, the neighbor is rewired to the newly added node. This feature makes the path smoother.[17]

The key steps of the RRT* algorithm are as follows:

Step 1: Point Xrandom is randomly generated in the configuration space T.

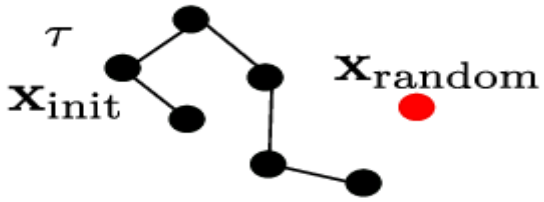


Figure2. 5:shows the sampling of a position x random.

Step 2: Node Xnearest closest to Xrandom is selected from the exploration tree with a connecting line, and a point on this line with a distance r (extension step of the exploration tree) from Xnearest is subsequently taken as a new leaf node, Xnew.

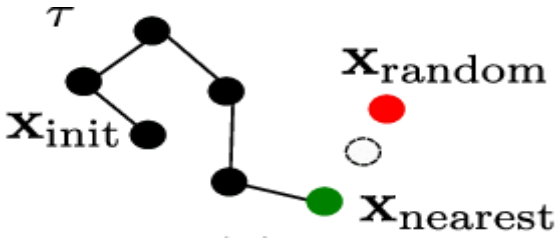


Figure2. 6:the nearest neighbor x nearest of x random is found and x new is generated.

Step 3: The nearest node set Xnear is constructed with the nodes of the exploration tree in a certain range centered on Xnew.

Step 4: Node Xmin is found as the parent of Xnew by traversing Xnearest, through which Xnew is connected to the exploration tree with the minimum path cost.

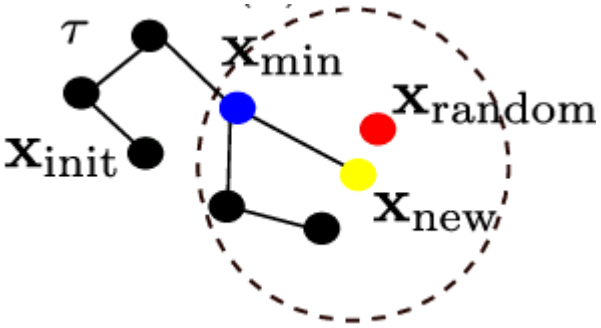


Figure2. 7:represents the choose Parent step.

Step 5: Xnew and Xmin are added to the node set, and the branch connecting them is added to the branch set.

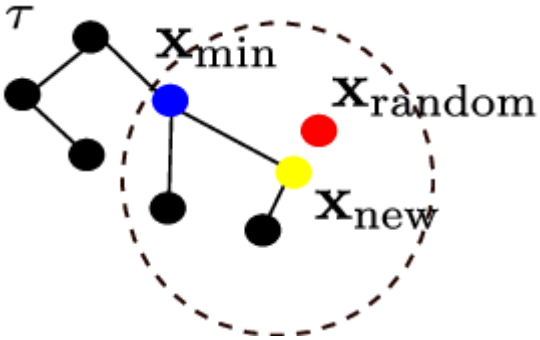


Figure2. 8:depicts the rewire step.

2.4.1.1 RRT* Variants

we will discuss the main contributions of various RRT* (Rapidly- exploring Random Tree Star) variants:

2.4.1.1.1 RRT*-Smart

is an extension version of RRT*, RRT*-Smart randomly searches the state space as RRT* does. Similarly, the first path is found just like the RRT*. Once the first path is found it then optimizes this path by interconnecting the directly visible nodes. This optimized path yields biasing points for intelligent sampling. This process continues as the algorithm progresses and the path keeps on being optimized rapidly.[18]

The RRT*-Smart algorithm contain two key concepts: Intelligent Sampling and Path Optimization

Path Optimization: is based on the concept of the Triangular Inequality, according to the Triangular Inequality, c is always less than the sum of a and b, and hence always gives a shorter path.[18]

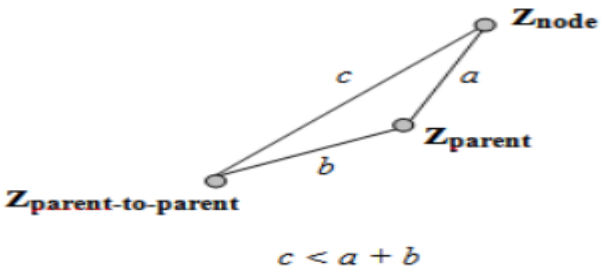


Figure2. 9:presentation of Triangular Inequality

Intelligent Sampling: is to approach optimality by generating the nodes as close as possible to obstacle vertices.[18]

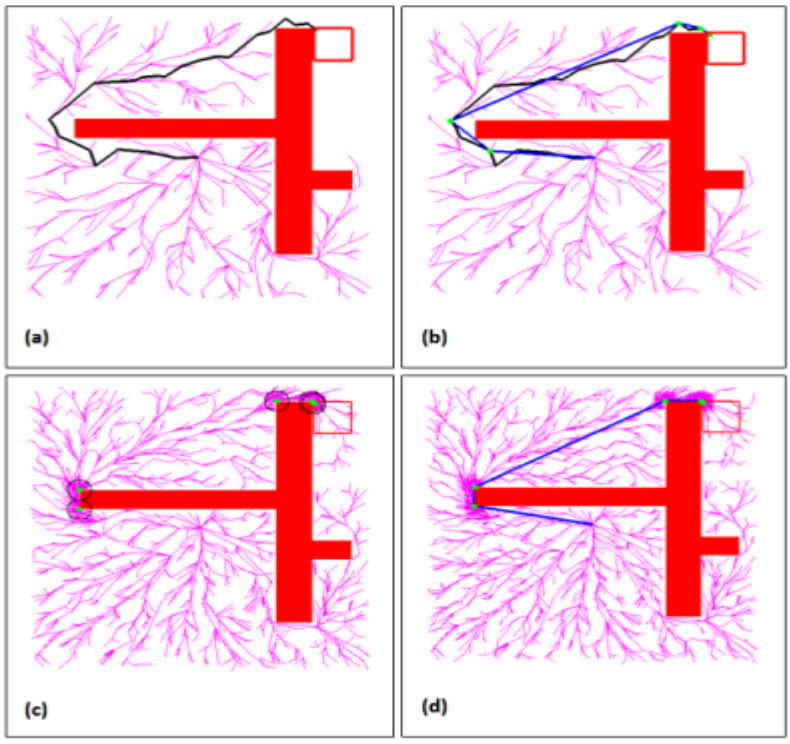


Figure2. 10::(a) First Path given by RRT*, (b) An optimized path (in blue) is shown after the Path Optimization technique is applied on the path shown in (a).,(c) shows clustered samples as a result of biasing towards the beacons (in green),(d) shows the optimum path

2.4.1.1.2 Informed RRT*

is a simple modification to RRT* that demonstrates a clear improvement. It uses heuristics to reduce the planning. once an initial solution is found, all possible improvement is contained in an ellipse defined by the path length the start and the goal. By directly sampling the ellipse, focus the search to only the state that have the possibility of improving the solution.[19]

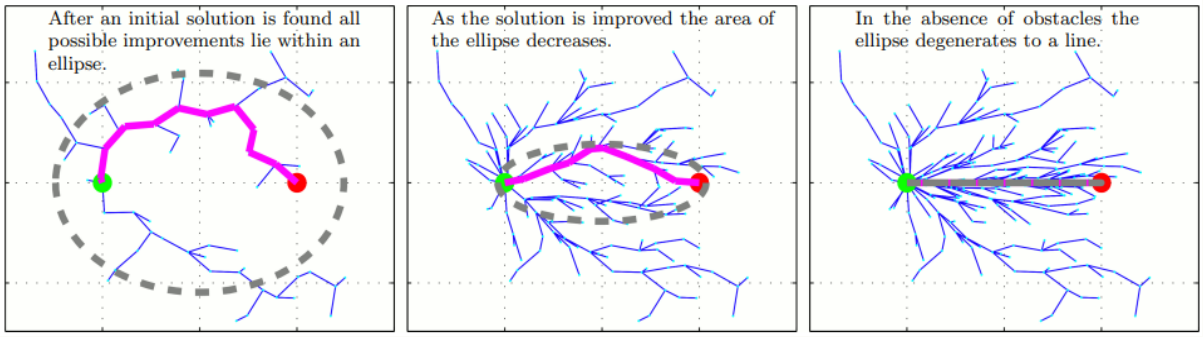


Figure2. 11:objectif of elipse in infirmed RRT* algorithm.

Informed RRT* can find topologically distinct optimal paths more quickly, and in the absence of obstacles can find the optimal solution to within machine zero in finite time. It could be used in combined with other planning technique like path smoothing.[19]

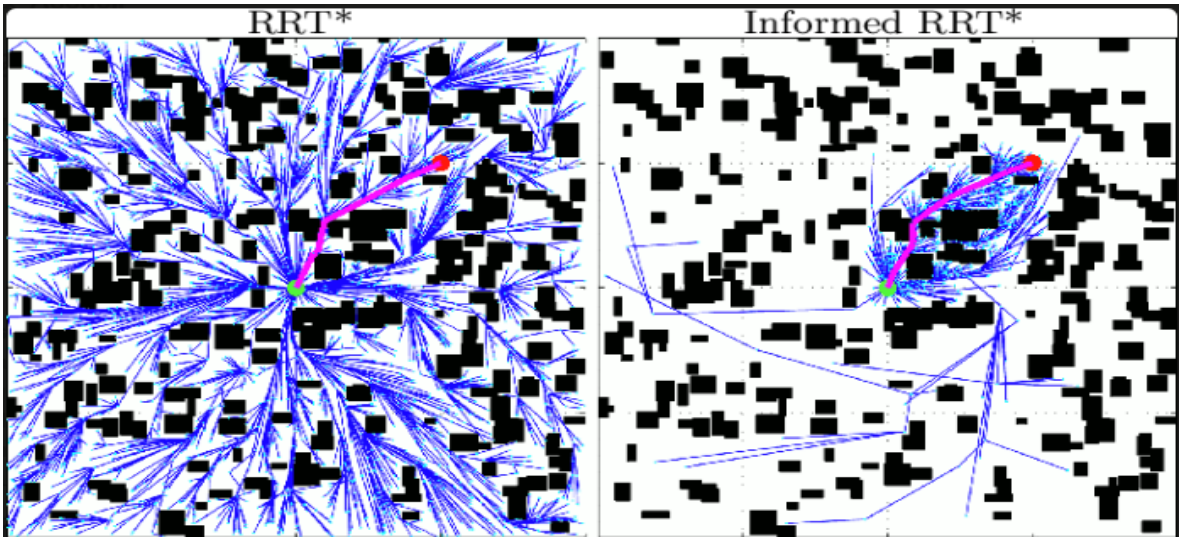


Figure2. 12:Solutions of equivalent cost found by RRT* and Informed RRT*.

2.4.1.1.3 Flight Cost-based-RRT* (FC-RRT*)

is a variant of the RRT* algorithm that incorporates flight cost considerations into the path planning process. This algorithm aims to optimize trajectories for flight applications. Focus on generating flight paths that minimize overall costs while satisfying flight requirements and constraints, making it suitable for applications in drone navigation.[20]

2.4.1.1.4 . Spline-RRT*

is proposed for the coordinated motion of a dual-arm space robot. An initial path for the dual-arm system is obtained by improving the standard RRT* planning algorithm. which samples from two separated inertial spaces to avoid possible self-collisions. Then, quartic splines reparametrize and smooth the generated RRT* path so that the robot can execute it. The motion planning framework based on sampling efficiently discovers a feasible path for the dual-arm space robot.[21]

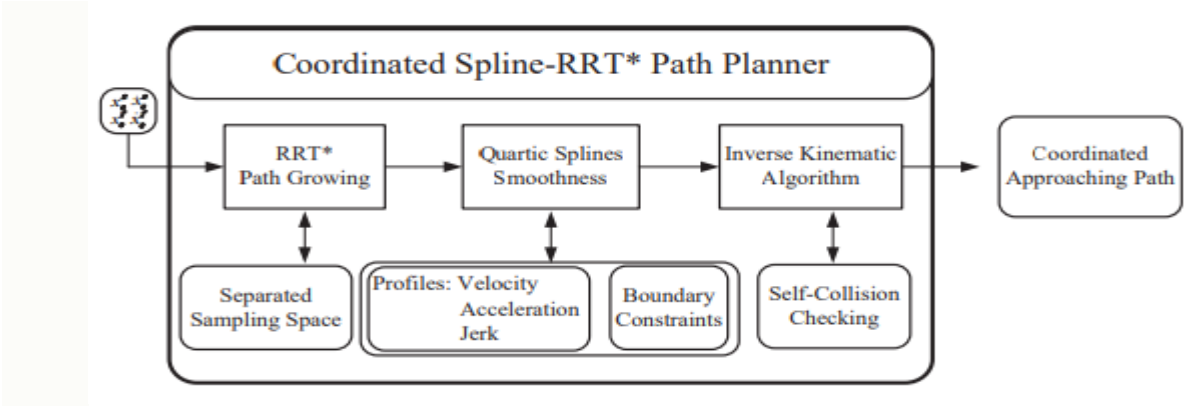


Figure2. 13:Overview of coordinated spline-RRT* planner.

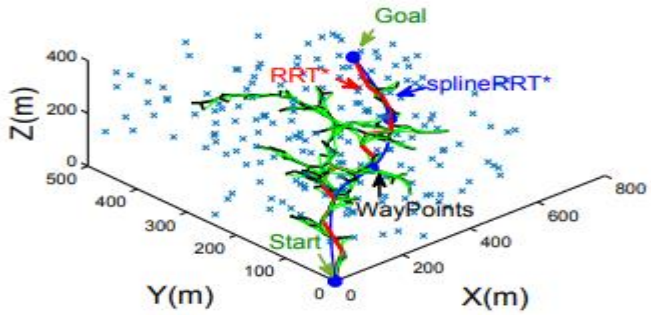


Figure2. 14::Simulation result of spline-RRT* algorithm.

2.4.1.1.5 rapidly exploring random tree-fixed node (RRT*FN)

limits the maximum number of nodes, with the same sampling, node expansion and parent node selection methods of the RRT*algorithm. by using a fixed set of nodes sampled at the beginning of the algorithm. This number of nodes remains constant all the execution. this method prevents infinite tree growth and saves memory. It includes heuristic sampling, dichotomy greedy expansion, two additional expansions, local environment sampling boundary expansion and triangle inequality path optimization Therefore, when the number of nodes reaches the maximum value, the vs.-RRT*FN algorithm forcibly removes some leaf nodes to free up memory. it involves projecting points onto lines, checking for intersections with obstacles, and creating paths with perpendicular points around obstacles to find optimal routes from a start point to a goal point. [22]

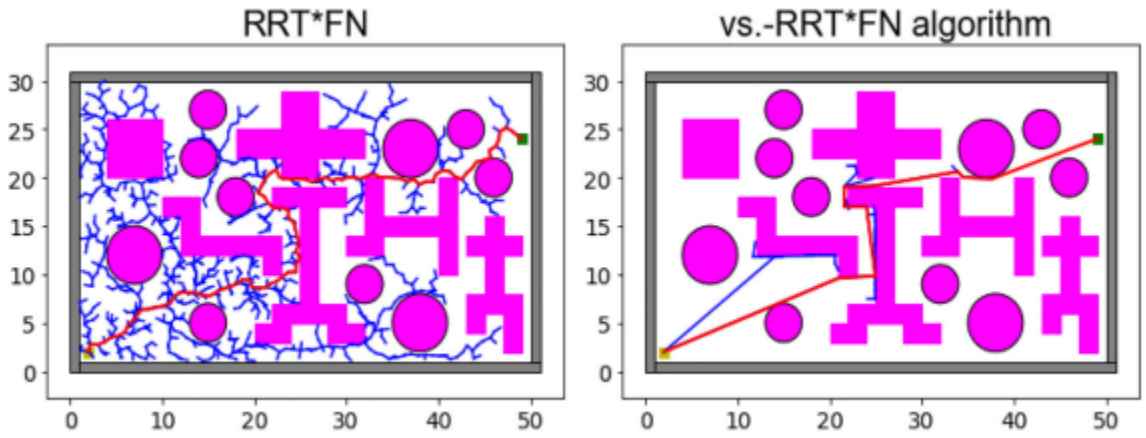


Figure2. 15:Simulation result of RRT*FN and VS-RRT*-FN algorithm.

2.5 Conclusion

the exploration of obstacle avoidance in autonomous robotics encompasses both reactive methods, like potential fields and Bug algorithms, and deliberative methods, including RRT* and its variants.

This dual approach provides a robust framework for navigating complex environments, ensuring both real-time responsiveness and strategic planning. Moving forward, the development of new geometric methods and novel variants of existing algorithms will continue to advance the field, enhancing the capabilities of autonomous systems.

In the following chapter, we will advance the discussion by developing a novel variant of the RRT* algorithm, aiming to improve its efficiency and performance in specific scenarios. Additionally, we will introduce a new geometric method for obstacle avoidance, enhancing the toolkit available for real-time robotic navigation. This exploration will not only highlight the innovative strategies in obstacle avoidance but also contribute to the ongoing evolution of autonomous robotic systems.

3 Chapter 3

Simulation, Analysis, and Result

3.1 Introduction

In this chapter, we focus on the implementation, analysis, and results of two our method: the Limited tree expansion RRT* and a geometric method concept-based approach. The Limited Expansion Tree RRT* is an optimized variant of the classic RRT* algorithm. On the other hand, the geometric concept-based method utilizes geometric principles to create safe and optimized paths in complex environments.

First, we will provide a detailed description of these two approaches, highlighting the theoretical principles and the innovations they bring. Then, we will explore the implementation of the algorithms, explaining the various functions and libraries used in our Python development environment, Finally, we will analyze the performance of both methods in terms of generated paths, computation time, and environmental complexity.

We will conclude this chapter with a detailed comparison of the obtained results, highlighting the advantages and limitation of each approach.

3.2 Limited tree expansion RRT* Methods

RRT* (Rapidly exploring Random Trees Star) is a motion planning algorithm used in robotics. It continuously expands a tree structure to find near-optimal paths from the start position to the goal, helping robots move around efficiently.

While the RRT* algorithm offers significant advantages in terms of finding near-optimal paths efficiently in complex environments, it also has some limitations and disadvantages:

One drawback is the process of iteratively expanding tree by sampling, connecting, and extending nodes can be time-consuming, particularly in environments with complex geometry or obstacles. As the tree grows. the algorithm may require more iterations to explore the entire configuration space, leading to longer planning times.

In our project, we've introduced a new approach to address this issue, several strategies can be employed to limit the expansion of the RRT* tree and focus the search on the most promising areas.

3.2.1 Design of Limited tree expansion RRT*method

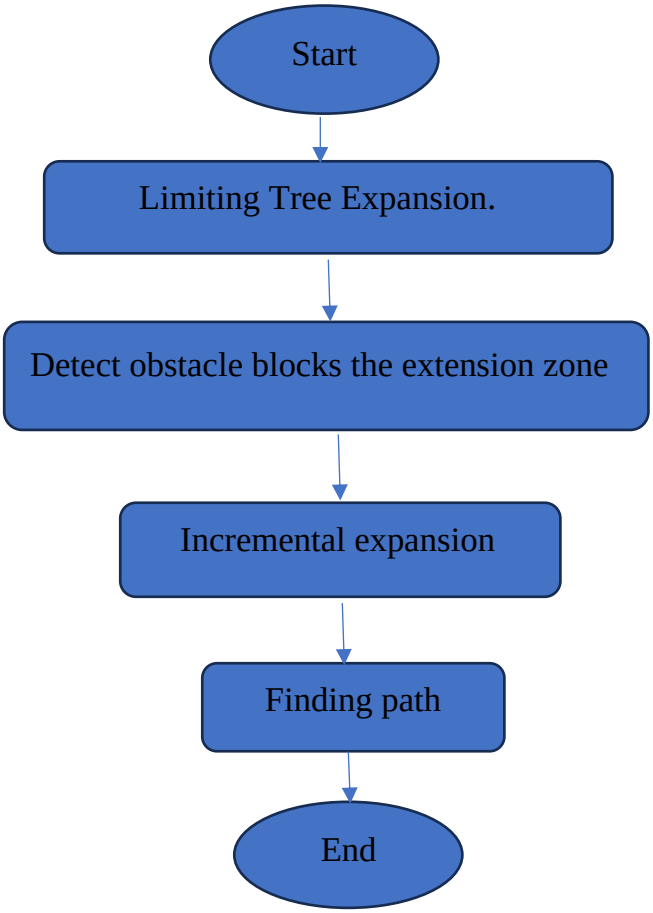


Figure3. 1:general architecture

3.2.1.1 Limiting Tree Expansion

Instead of allowing the RRT* tree to grow without bounds, the algorithm should confine the tree expansion within the region defined by the x start and x goal coordinates. This helps to focus the search on the relevant state space and avoid unnecessary exploration in areas that are unlikely to improve the solution. By setting appropriate limits on the tree's growth, the algorithm can efficiently navigate towards the goal while avoiding wasting computational resources on distant regions.as show in figure 3.2:

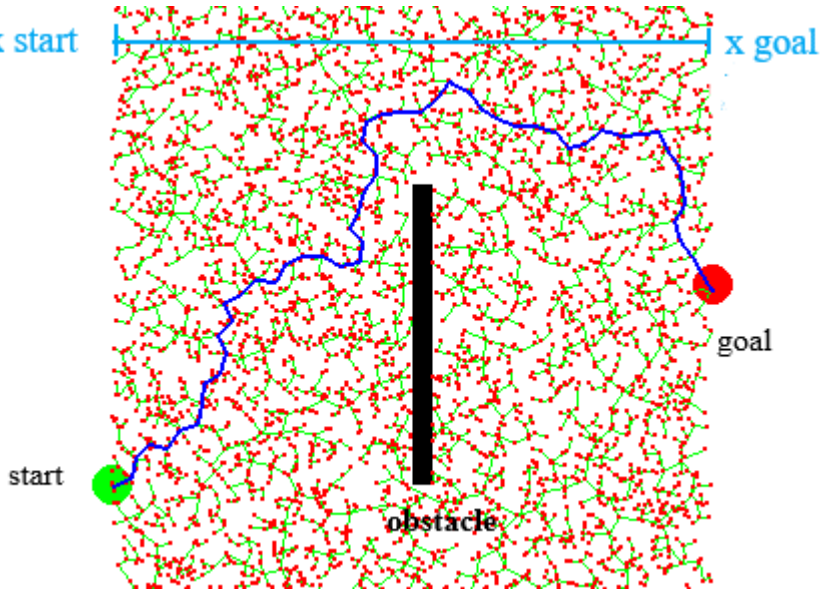


Figure3. 2:presentation of the Exploration Bounds of the tree.

Expanding on this approach, while confining the extension of the Rapidly exploring Random Tree (RRT*) algorithm between the coordinates of x start and x goal proves successful in many cases, encountering obstacles within this restricted zone presents a challenge. However, by dynamically adjusting the extension area, the algorithm can adapt and overcome such obstacles:

3.2.1.2 Obstacle Detection

When an obstacle blocks the extension zone between x start and x goal as shown in figure 3.3., the algorithm detects this obstruction during the tree expansion process. This detection triggers a deviation from the standard extension procedure, signaling the need for an alternative approach to find a feasible path.

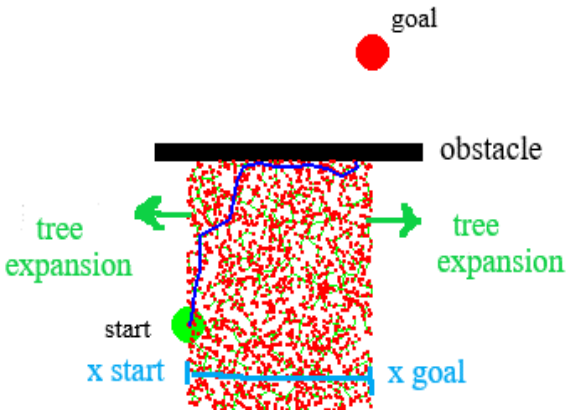


Figure3. 3:obstacle blocks the extension zone .

3.2.1.3 Incremental expansion

When the algorithm encounters an obstacle that blocks the direct path between the start and goal points, it responds by incrementally expanding the area it searches for a viable route. Rather than being limited to the original bounds, the algorithm dynamically grows the exploration space to find alternative pathways that can navigate around the obstacle.

3.2.1.4 Finding path

once the algorithm identifies a potential path. it stops expanding the search tree any further as shown in figure 3. 4. By halting the extension process at this point, the algorithm indicates that it has found a feasible route from the starting point to the goal.

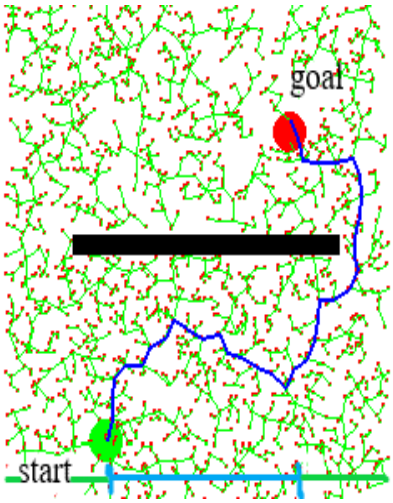


Figure3. 4:finding the path.

3.2.2 Objective of Limiting Tree Expansion RRT*

In the realm of pathfinding algorithms, our method of confining tree expansion within specified start and goal coordinates emerges as a powerful technique. This approach, aimed at simplifying the exploration process, offers a multitude of benefits ranging from reduced complexity to improved path quality and clearer visualization:

Reduced Complexity

Limiting the tree expansion helps in reducing the computational complexity associated with exploring the entire problem space. The algorithm focuses on the specific area where the optimal path is more likely to be found.

Faster Convergence

By limiting the search space, the RRT* algorithm can converge to a solution more quickly This targeted approach allows the algorithm to rapidly discover feasible solutions, improving its overall efficiency and speed.

Improved Path Quality

the algorithm can potentially generate paths of higher quality. Since the exploration is focused on the relevant part of the problem space, the resulting paths are more likely to be optimal. This targeted approach allows the algorithm to find solutions that are more efficient and effective.

Clear Visualization

By confining the tree's growth, the visualization becomes significantly clearer. This allows for a simple observation of how the algorithm navigates the space, which areas it traverses, and how it constructs paths leading to the goal.

3.2.3 RRT* Algorithm VS Limited Tree Expansion RRT*

In the field of path planning algorithms, the strategic choice between the exploration of the RRT* algorithm and our concept of limiting tree expansion presents a pivotal decision point. Both approaches offer unique advantages, shaping the efficiency and effectiveness of pathfinding solutions.

3.2.3.1 RRT* Algorithm

The RRT* algorithm is a powerful tool for probabilistic path planning, known for its ability to construct a tree that gradually extends towards the goal while optimizing the path by minimizing overall cost. This is achieved by iteratively sampling random points in the configuration space and connecting them to the nearest node in the tree.

3.2.3.2 Limited Tree Expansion RRT*

To address the potential drawbacks of the RRT* algorithm, we proposed the concept of limiting tree expansion. By confining the growth of the search tree within the region defined by the x start and x goal coordinates, this approach focuses the exploration on the most relevant areas of the configuration space.

3.2.3.3 Comparison

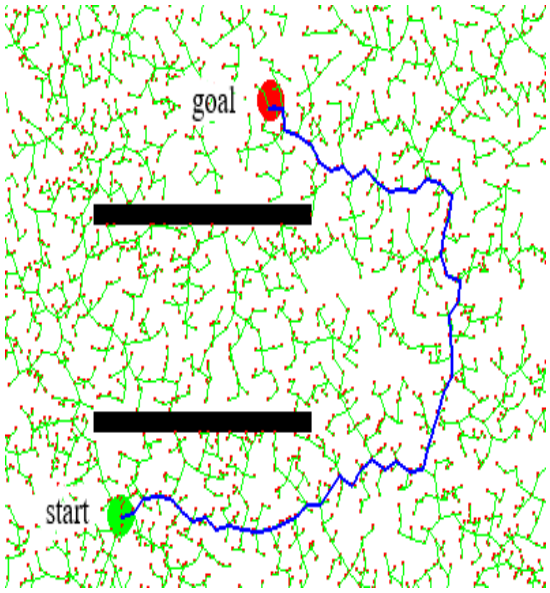


Figure3. 5:presentation of RRT* map.

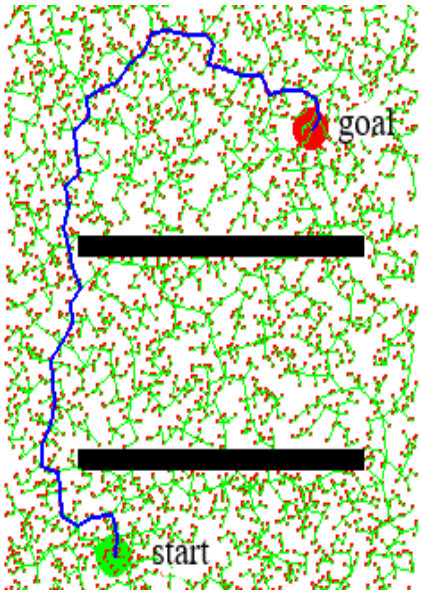


Figure3. 6:presentation of Limited Tree Expansion map

Limited Tree Expansion RRT* was compared to RRT* on a variety of simple planning problems (figure 3.5, figure 3.6), Simple problems were used to test specific challenges.

The RRT* algorithm and limiting tree expansion differ in their exploration scope, complexity, optimality, path quality, and visualization. RRT* explores the entire configuration space, whereas limiting tree expansion confines the search to a specific region, reducing complexity and improving efficiency. While RRT* strives for optimality, our concept prioritizes expedient convergence. Both approaches aim for high-quality paths but limiting tree expansion achieve this more efficiently and more optimality. Confining tree expansion also enhances visualization clarity, aiding in easier interpretation and debugging.

in the term of faster convergence, we compared RRT* and limited tree expansion as the figure explains:

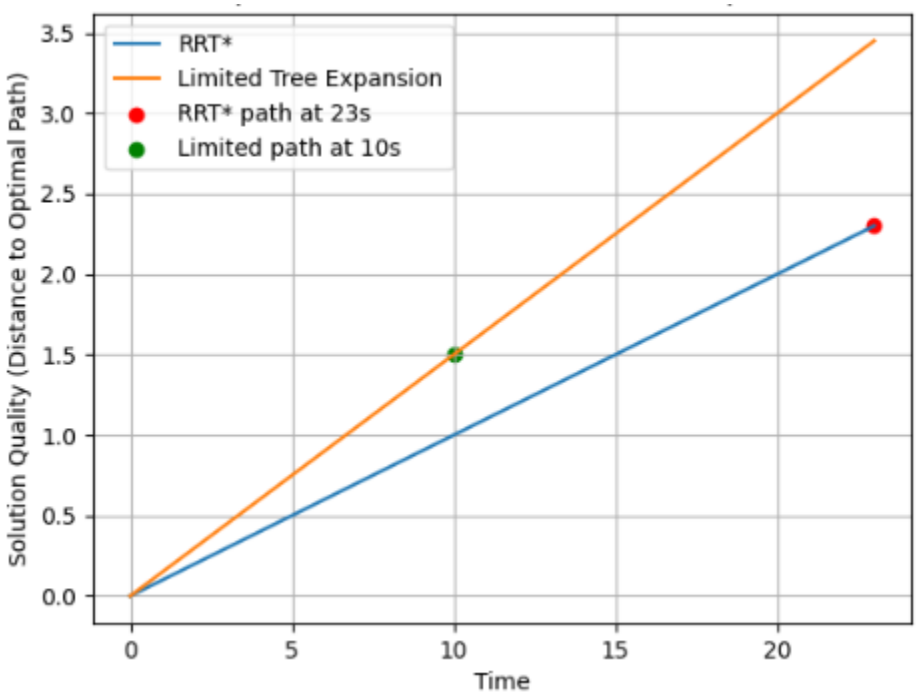


Figure3. 7:comparison of RRT* and limited tree expansion.

The graph compares the performance of the RRT* algorithm with limited tree expansion concerning the distance to the optimal path over time.

The plot for RRT* (in blue) shows a linear increase in the distance to the optimal path over time. This indicates that RRT* is approaching the optimal path gradually.

The plot for limited tree expansion (in orange) also depicts an increase in the distance to the optimal path over time. so, it increases at a faster rate compared to RRT*, suggesting that limited tree expansion converges towards the optimal path more quickly.

The red and green scatter points indicate the paths provided by RRT* and limited tree expansion, respectively. The red point represents the path provided by RRT* at 23 seconds, while the green point represents the path provided by limited tree expansion at 10 seconds.

Overall, the graph allows for a visual comparison of how RRT* and limited tree expansion algorithms progress towards the optimal path over time. Limited tree expansion demonstrates a faster convergence towards the optimal path compared to RRT*.

3.3 Geometric Methods

Geometric methods are fundamental in the field of path planning, particularly in robotics and autonomous systems. These methods leverage the principles of geometry to design efficient and effective paths for navigating through an environment.

Geometric methods often aim to find the most efficient path between two points. This efficiency is typically measured in terms of distance, time, or energy consumption.

In our project we developed new geometric method designed to be computationally efficient. we used well-defined mathematical properties and structures (such as rectangles and vertex) to reduce the complexity of the path planning problem.

3.3.1 Global Architecture

Step 1: Start by establishing a straight-line segment between the starting point and the goal point.

Step 2 : Obstacle Détection and Analysais :

Check for obstacles intersecting with the initial straight-line segment.

Step 3: Perpendicular Distance Calculation:

-Calculate perpendicular distances from the vertices of detected obstacles to the initial line.

-Focus on vertices above or below the line, depending on the orientation of the obstacle.

Step 4: Identification of Critical Turning Points:

-Identify the longest perpendicular distances from vertices above and below the line.

-Choose the shortest of these distances as the nearest point of deviation from the straight path.

-Mark this chosen vertex as a critical turning point with a circle.

Step 5: Path Refinement:

-Redraw the line segment the marked critical turning point to the goal .

-Repeat the obstacle detection, perpendicular distance calculation, and critical turning point identification process along the new line segment.

Step 6: Iterative Refinement:

-Continue iterating this process until the line segment from the last marked critical turning point to the goal point is obstacle-free.

Step 7: Path Completion:

-Once a clear path is established from the start to the goal without any obstacles, the pathfinding process is complete.

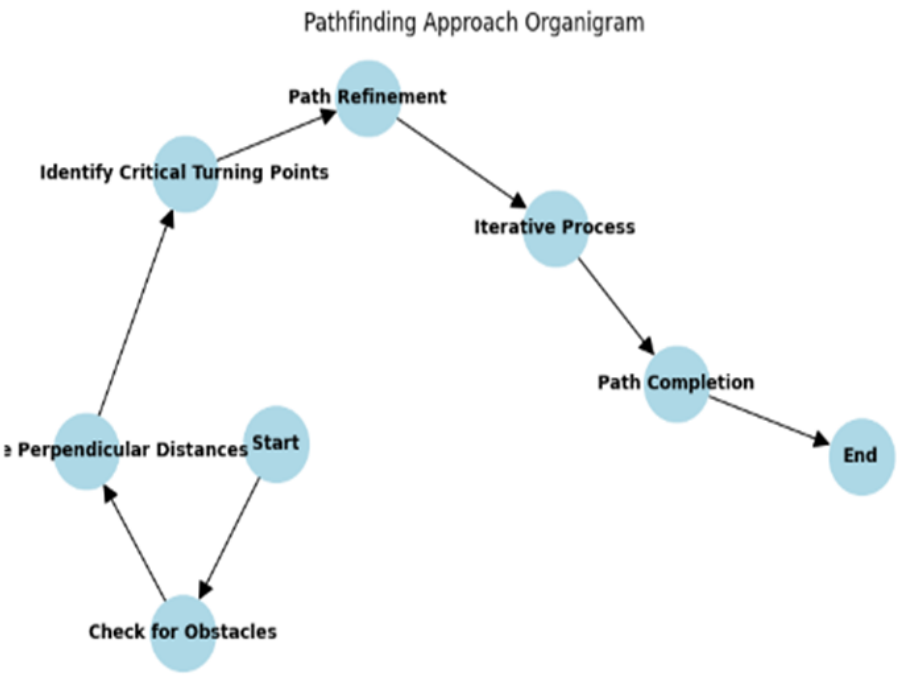


Figure3. 8:Global architecture

3.3.2 Overview of the Concept

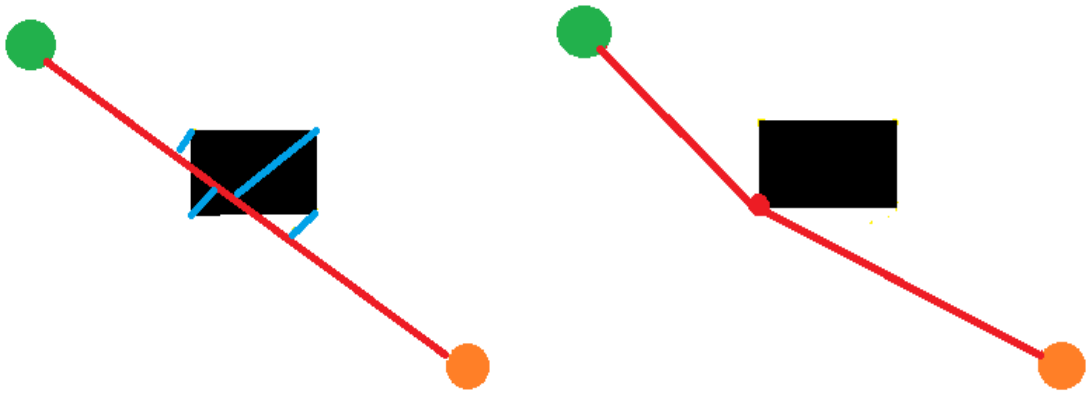
Our approach begins by drawing an initial straight line between the starting point and the goal point.

This line serves as the basis for our pathfinding method. We then examine this line to check for any obstacles that intersect it.

When an obstacle is detected, we proceed to calculate the perpendicular distances from the vertices of the obstacle to our initial line.

Specifically, we focus on the vertices that are either above or below the line. We measure the lengths of these perpendiculars and identify the longest perpendicular distance from the vertices above the line and the longest perpendicular distance from the vertices below the line. Out of these two longest perpendiculars, we select the one with the shortest length, ensuring we choose the nearest point of deviation from our straight path. This chosen vertex is marked with a circle to indicate a critical turning point in our path. Following this, we redraw the line segment from the starting point to this newly marked vertex and repeat the process: checking for further obstacles along

the new line segment, calculating perpendicular distances, selecting the most significant deviation point, and marking it. This iterative process continues until the line segment from our last marked vertex to the goal point is free of obstacles, thereby constructing a clear and navigable path from the start to the goal.



a-presentation of the perpendiculars b-presentation of the new line

Figure3. 9:presentation of the method.

Purpose of Expansion

The expansion ensures that the pathfinding algorithm considers a safety margin around each obstacle. This is crucial for scenarios where the robot has a physical size that needs to be accommodated to avoid collisions. By expanding the obstacles, the algorithm can create a path that maintains a safe distance from the edges of the obstacles.

Expanding the obstacle involves adjusting its boundaries to create a larger rectangle. By shifting the left boundary leftward and the top boundary upward each by the expansion amount, and increasing both the width and height

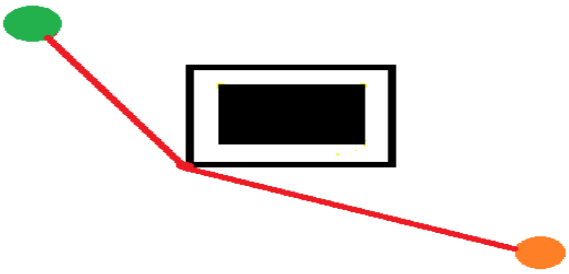


Figure3. 10:Expansion of the obstacle.

3.3.3 Geometrics properties

The Geometrics properties and calculations ensure that the pathfinding algorithm effectively navigates around obstacles by finding optimal points of deviation and iteratively constructing a collision-free path.

The Euclidean distance

The distance function calculates the Euclidean distance between two points. The formula is:

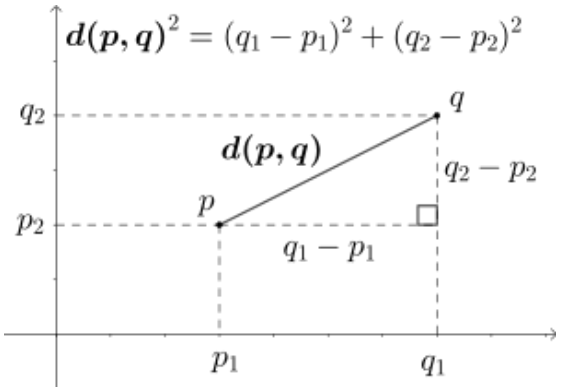


Figure3. 11:Euclidean distance.

the Euclidean distance helps:

- in finding the closest point on a line segment to a given point.

-Find the longest and shortest perpendicular vertices of obstacles.

-Finding the shortest and longest distance from a point to a line.

-When a path segment intersects an obstacle, projecting the obstacle's vertices onto the path helps determine which points to use to navigate around the obstacle.

Victor Projection

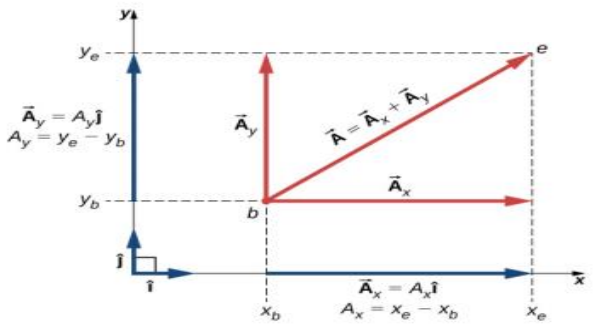


Figure3. 12:vector projection.

-Helps for Normalized the Direction Vector from start point to goal point and Direction of perpendiculars.

3.3.4 The Objectives of the idea

Create a straight-line path between the start and goal points.to Provides a quick estimate of the path length and a framework for further refinement.

Identify obstacles that intersect the initial path to Ensures that the algorithm can effectively navigate around obstacles.

Measure perpendicular distances from obstacle vertices to the initial path to Identifies key points where the path needs to deviate to avoid obstacles.

Choose the most efficient deviation points based on the calculated distances to Minimizes the overall path deviation, ensuring efficient navigation.

Continuously adjust the path by adding new turning points until the path is clear of obstacles to Creates a smooth, obstacle-free path from the start to the goal point.

Provide a visual representation of the pathfinding algorithm's progress to Enhances understanding and debugging of the algorithm through clear visualization.

Develop a path that is both efficient in terms of distance and clear of any obstacles to Facilitates reliable and effective navigation in various environments.

By meeting these objectives, the pathfinding algorithm aims to deliver a reliable solution for autonomous navigation, ensuring that the path from the start to the goal is as direct and obstacle-free as possible.

3.3.5 Theoretical Analysis.

Time Complexity Analysis.

For each obstacle, check if it intersects the line Complexity per Obstacle Intersection checks can typically be done in C (1) time per obstacle. If there are n obstacles, the complexity is C(n).

Vertices per Obstacle Assume each obstacle has v vertices on average. Distance Calculation Computing the perpendicular distance for each vertex involves basic arithmetic operations, resulting in C (1) time per vertex. The Complexity is for n obstacles with v vertices each, the complexity is C(nv).

Finding the longest perpendicular distances from above and below the line involves comparing distances for each vertex. The Complexity As each comparison is C (1), the complexity for all vertices is C(nv).

Iterations, In the worst case, this process may need to be repeated for each obstacle. The complexity per iteration is C(nv). If k iterations are needed, the overall complexity is C(knv)

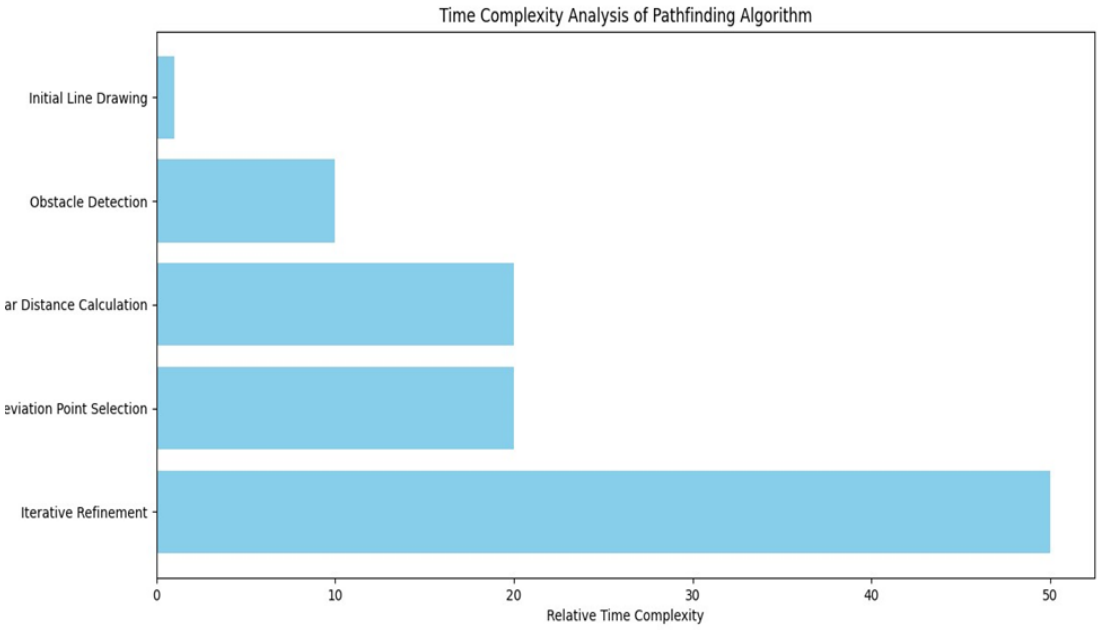


Figure3. 13:Histogram representing time complexity analysis of pathfinding.

Space Complexity Analysis

Path Storage: Store the list of deviation points. In the worst case, there might be as many deviation points as there are obstacle Storing the vertices of each obstacle, requiring a space.

Path Optimality By selecting the nearest critical deviation point, the approach ensures minimal deviation at each step.

Robustness

Obstacle Handling: Efficiently handles a variety of polygonal obstacles, provided they have well-defined vertices.

Dynamic Environments: This static pathfinding method does not adapt to dynamic changes. Modifications are required for real-time obstacle updates.

Path Optimality

Local Optimality: By selecting the nearest critical deviation point, the approach ensures minimal deviation at each step.

Global Optimality: The algorithm does not guarantee a globally optimal path, as it focuses on local optimizations. It may result in a longer overall path if locally optimal points do not align well globally.

Completeness

Solution Guarantee: If a path exists, the iterative process should eventually find it by repeatedly refining the path and avoiding detected obstacles.

Exploration Coverage: The method systematically checks and adjusts for obstacles, ensuring thorough exploration along the initial path line.

Implementation Complexity

Ease of Implementation: The algorithm is straightforward to implement, involving basic geometric calculations and iterative refinement.

Computational Requirements: Suitable for environments with moderate complexity; may require optimization for high-density obstacle fields or very large environments.

3.3.6 Analyzing the Key Attributes of the Algorithm

Simplicity: The method starts with a straightforward straight-line path between the start and goal points, which serves as the basis for further refinement. This simplicity makes the algorithm easy to understand and implement.

Efficiency: By initially drawing a straight line, the algorithm provides a quick estimation of the path length. This estimation can be valuable, especially in real-time applications where quick decisions are required.

Adaptability: The algorithm adapts to obstacles by iteratively refining the path based on detected intersections. This adaptability allows it to navigate complex environments with both vertical and horizontal obstacles effectively.

Optimization: By selecting critical turning points along the initial straight line, the algorithm optimizes the path by choosing the nearest deviation point. This optimization minimizes the number of calculations needed to reach the goal, leading to efficient pathfinding.

Clear Path Construction: The iterative process of refining the path ensures that the final path from the start to the goal is clear and navigable, avoiding obstacles effectively. This clear path construction is essential for successful navigation in obstacle-rich environments.

Versatility: The method is versatile and can be applied to various pathfinding scenarios, including robotics, game development, and logistics planning. Its simplicity and adaptability make it suitable for different applications.

Incremental Improvement: The algorithm iteratively improves the path by detecting obstacles, calculating perpendicular distances, and selecting optimal deviation points. This incremental improvement ensures that the final path is well-suited for navigation.

the radar chart in figure highlight that the pathfinding method excels in simplicity, clear path construction, and efficiency, while also being highly versatile and capable of incremental improvement. There is a moderate strength in adaptability and optimization, indicating potential areas for further enhancement. This visual representation underscores the method's effectiveness and suitability for various pathfinding tasks.

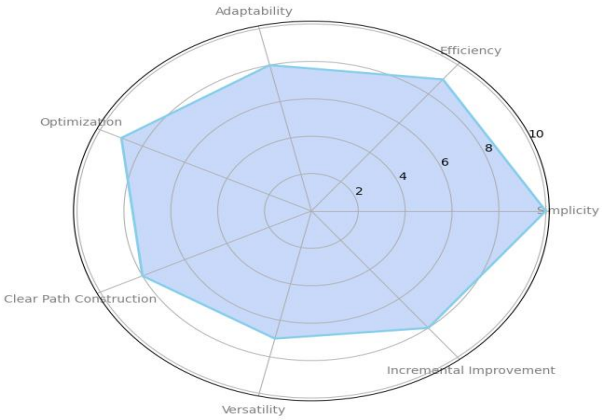


Figure3. 14:radar chart the strengths and weaknesses of the method across different attributes.

Keys:

Blue Line: Represents the pathfinding method’s performance on each attribute.

Shaded Blue Area: Visualizes the overall performance; a larger area indicates stronger overall performance.

Concentric Circles: Provide the scale for each attribute, with the outermost circle typically representing the highest score (10).

Axes Labels: Indicate the attributes being measured.

3.4 Visual representation and simulation

3.4.1 Software development environment

PyCharm 2024.1

PyCharm is an Integrated Development Environment (IDE) used for programming in Python. It provides code analysis, a graphical debugger, an integrated unit tester,

integration with version control systems (VCSes), and supports web development with Django. PyCharm is developed by the Czech company JetBrains.

PyCharm is a powerful and comprehensive IDE that greatly simplifies the development of Python applications, from simple scripts to complex web and data projects.[23]

Figure3. 15:PyCharm logo

3.4.2 Programming language

Starting with the research of the programming language, which will be used to develop the idea, is usually the first thing to do. After taking a deep and careful general look at many programming languages, Python was the one catching the interest.

python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms and can be freely distributed.

Python is a general-purpose programming language started by Guido van Rossum that became very popular very quickly, mainly because of its simplicity and code readability. It enables the programmer to express ideas in fewer lines of code without reducing readability.[24]

Figure3. 16:python logo.

Why python?

Due to the above reasons, Python was chosen as Programming Language. To summarize it all

Python is robust.

It is solid and powerful. Python has a relatively small quantity of lines of code, which makes it less prone to issues, easier to debug, and more maintainable. It is also very fast.

Python is flexible.

Because it was not created to answer a specific need, Python is not driven by templates or specific APIs, therefore, well suited to the rapid development of all kinds of applications.

Python is easy to learn and use.

"Python, in particular, emerges as a nearly ideal candidate for a first programming language", says John M. Zelle, in the Department of Mathematics, Computer Science, and Physics at Wartburg College in Iowa.

Python is free.

Since Python is an open-source programming language, it immediately reduces upfront project costs by leveraging Python in the development projects.

Python in robotics.

In robotics, programming, simulation, and visualization are an essential factor in facilitating and simplifying complex matters. In our experience, the Python programming language has been instrumental and powerful.

3.4.3 Visualization.

Visualizing the process of our program is crucial for understanding its behavior and effectiveness. Pygame serves as the perfect tool for this purpose, offering a versatile framework for creating interactive graphical interfaces. By integrating Pygame into our program, we unlock the ability to dynamically display various elements.

Pygame is a powerful Python library specifically designed for creating and developing multimedia applications and games. It provides a robust set of tools and functionalities for handling graphics, sound, input devices, and other multimedia components, making it a popular choice among game developers.[25]

Figure3. 17:Pygame logo.

In our code we utilize Pygame to visualize the Limited Tree Expansion RRT* path planning algorithm in action. Let's break down how Pygame is utilized in the code:

Pygame is used to draw various elements on the screen. Circles representing the start and goal positions are drawn using. **pygame. draw. circle ()**, while rectangles representing obstacles are drawn using **pygame. draw. rect ()**. Lines representing the RRT* tree and the final path are drawn using **pygame. draw. line ()**
and pygame. draw. Lines () respectively.

Pygame is used to handle events such as quitting the application. The **pygame.event.get ()** function is used to retrieve and handle events, such as the user closing the window.

After drawing all the elements, the screen is updated using **pygame. display. Flip ()** to reflect the changes.

Pygame is utilized in conjunction with custom classes (Node and RRTMap) to

represent nodes in the RRT* tree and to draw the map with obstacles, start, goal, tree, and path.

Pygame serves as the backbone for creating the graphical interface and visualizing our concept in real-time, making it easier to understand and debug the algorithm's behavior.

3.4.4 The libraries used.

In our program, we used three libraries: the pygame library, which we mentioned earlier, the math library, and the random library, to implement an RRT* (Rapidly exploring Random Tree Star) path planning algorithm.

pygame: Used for creating a graphical user interface to visualize the path planning process. It provides functions for drawing shapes, handling events like quitting the application, and updating the display.

math: Utilized for mathematical operations required in the path planning algorithm, such as calculating distances between points and performing geometric calculations for steering towards a new point.

random: Employed for generating random points within a specified range, which is essential for the random sampling strategy in the RRT* algorithm.

3.5 Simulation

3.5.1 Pygame Initialization

3.5.1.1 Creation of the environment

We created the scene using the Pygame library to establish the 2D environment and visualize the path planning process. This environment serves as a representation of a laboratory within our department.

3.5.1.2 Creating obstacles

In our program, obstacles are introduced as rectangular entities within the 2D environment depicted by the Pygame window. These obstacles, akin to the walls of a laboratory, are defined by their positions (x coordinate, y coordinate) and dimensions (width, height). They remain static throughout the program's execution, serving as permanent fixtures in the environment and influencing the path planning process to avoid collisions. These obstacles, representing the walls of a lab, are stored in a list structure:

```
pygame.Rect(750, 50, 10, 500),
```

Here is the general view of the scene.

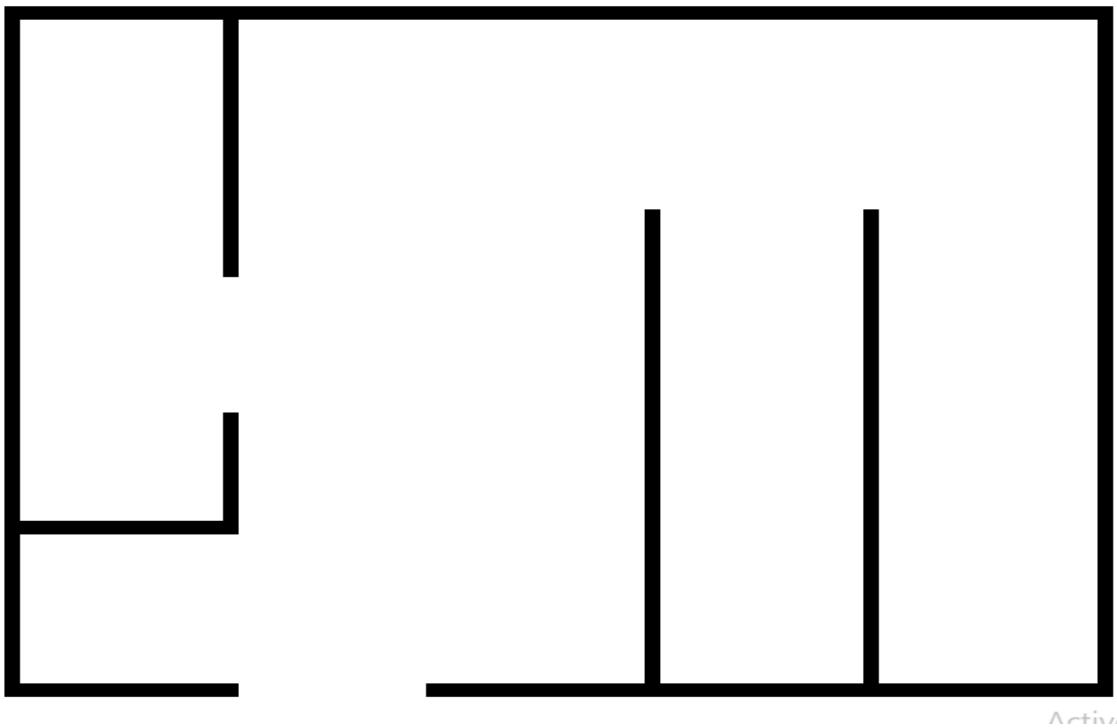
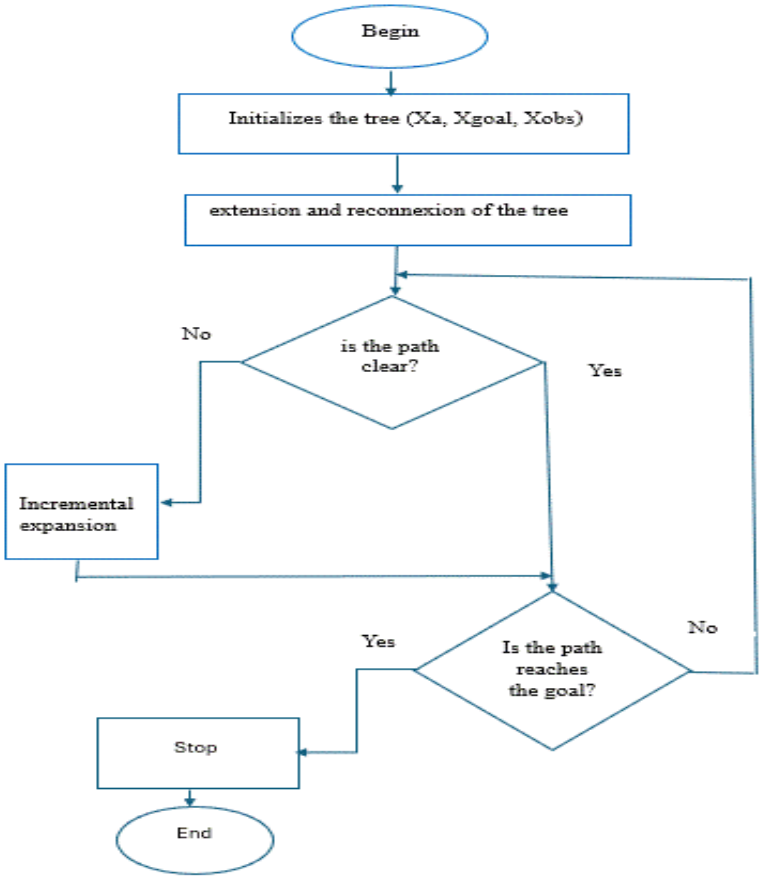


Figure3. 18:presentation of the general view of a lab.

3.5.2 Limited tree expansion.

3.5.2.1 Algorithm logigram



3.5.2.2 Utility Functions.

3.5.2.2.1 **New functions:**

random_point_within_zone

```
def random_point_within_zone(start, goal):
    min_x = min(start.x, goal.x)
    max_x = max(start.x, goal.x)
    return Node(random.uniform(min_x, max_x), random.uniform( a: 0, HEIGHT))
```

This function is designed to generate a random point within the zone defined by the x-coordinates of two given point: start and goal, for limited the expansion of the tree

is_path_clear.

```
def is_path_clear(from_node, to_node, obstacles):
    for obstacle in obstacles:
        if obstacle.collidepoint(to_node.x, to_node.y):
            return False
    return True
```

This function checks if there are any obstacles along a path from one node to another node. it checks If any obstacle is found to be blocking the path, or if there is a collision detected.

random_point.

```
def random_point(start, goal, offset):
    min_x = min(start.x, goal.x) - offset
    max_x = max(start.x, goal.x) + offset
    return Node(random.uniform(min_x, max_x), random.uniform( a: 0, HEIGHT))
```

this function generates a random point within an expanded search space to avoid obstacles. The expansion is achieved by adjusting the range of x-coordinates based on the offset value, ensuring that the random point is not too close to obstacles that might block the direct path from start to goal.

3.5.2.2.2 Tree function

nearest_node: find and return the node in the tree that is closest to the specified point, using the Euclidean distance as the measure of closeness.

Steer: to steer toward a new point

rrt_star: function to find a path:

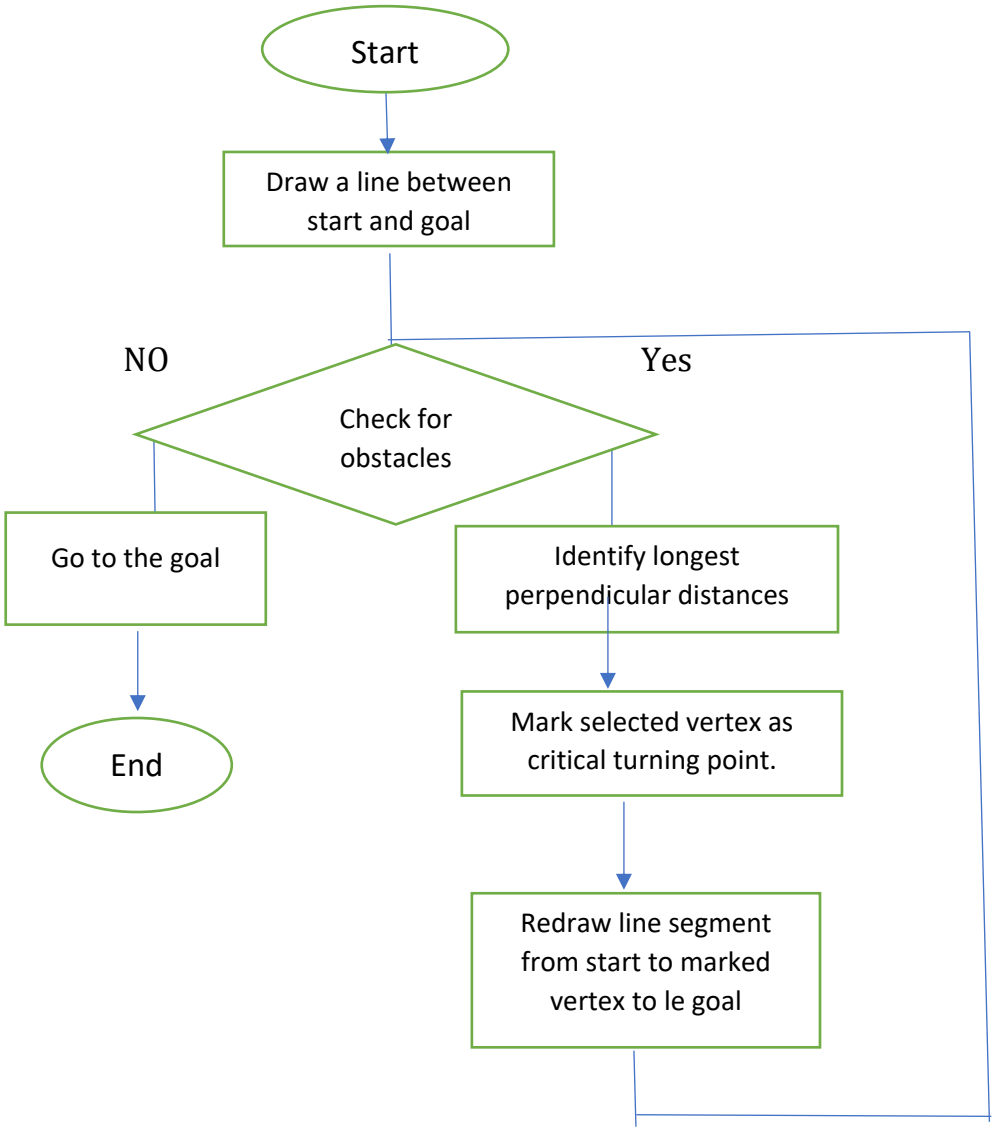
- Find the node closest to the goal.

-Build the path from the nearest node to the start.

- Reverse the path to start at the beginning.

3.5.3 Geometric Method

3.5.3.1 Algorithm logigram



3.5.3.2 Utility Functions

distance: Computes the Euclidean distance between two points.

```
def distance(point1, point2):
    return math.sqrt((point2[0] - point1[0]) ** 2 + (point2[1] - point1[1]) ** 2)
```

project_point_onto_line: Projects a point onto a line segment.

```
def project_point_onto_line(point, line_start, line_end):
    dx = line_end[0] - line_start[0]
    dy = line_end[1] - line_start[1]
    magnitude = math.sqrt(dx ** 2 + dy ** 2)
    ux = dx / magnitude
    uy = dy / magnitude
    vx = point[0] - line_start[0]
    vy = point[1] - line_start[1]
    projection_length = vx * ux + vy * uy
    projected_point = (
        line_start[0] + projection_length * ux,
        line_start[1] + projection_length * uy,
    )
    return projected_point
```

does_line_intersect_obstacle: Checks if a line intersects with any obstacle.

```
def does_line_intersect_obstacle(line_start, line_end, obstacle):
    return obstacle.clipline((line_start, line_end)) != ()
```

The function **does_line_intersect_obstacle** checks whether a line segment defined by its start and end points intersects with a given obstacle. It utilizes the **clipline** method of the obstacle object to perform this check. It calls the **clipline** method of the obstacle object, passing the start and end points of the line segment as a tuple.

If the **clipline** method returns an empty tuple, it means there's no intersection, and the function returns **False**. Otherwise, it returns **True**, indicating an intersection.

find_longest_perpendicular_to and find_shortest_perpendicular_bottom: Find the longest perpendicular distances from obstacle vertices to the initial straight line.

```
def find_longest_perpendicular_top(line_start, line_end, obstacle):
    top_vertices = [vertex for vertex in [obstacle.topleft, obstacle.topright]
    longest_perpendicular_vertex = None
    longest_perpendicular_distance = -float('inf')
    for vertex in top_vertices:
        projected_point = project_point_onto_line(vertex, line_start, line_end)
        perp_distance = distance(vertex, projected_point)
        if perp_distance > longest_perpendicular_distance:
            longest_perpendicular_distance = perp_distance
            longest_perpendicular_vertex = vertex
    return longest_perpendicular_vertex
```

Pathfinding Functions.

create_path_with_perpendiculars: Constructs a path by finding perpendicular vertices on intersecting obstacles.

3.6 Statistical Analysis of Path Efficiency

in the preceding chapter, we examined the contrast between the RRT star and Limited Tree Expansion algorithm. Now, in this section we'll extend our analysis to encompass three diverse environments, each environment was carefully selected to represent different challenges. enabling a thorough examination of how the algorithms navigates obstacles, adapts to changes, and optimizes its pathfinding strategy.

Through this comprehensive analysis, our objective was to elucidate the unique strengths and weaknesses of each algorithm, RRT*, limited tree expansion, and the geometric method. By subjecting these algorithms to various environments, we evaluate the variations in complexity, path length, and execution time for real-world applications across diverse settings. This comparative approach allowed us to evaluate not only their individual capabilities but also their relative advantages and limitations in addressing different challenges.

3.6.1 Limited tree expansion method

We introduced three distinct environments to observe how the limited tree expansion algorithm executes and reacts within each setting:

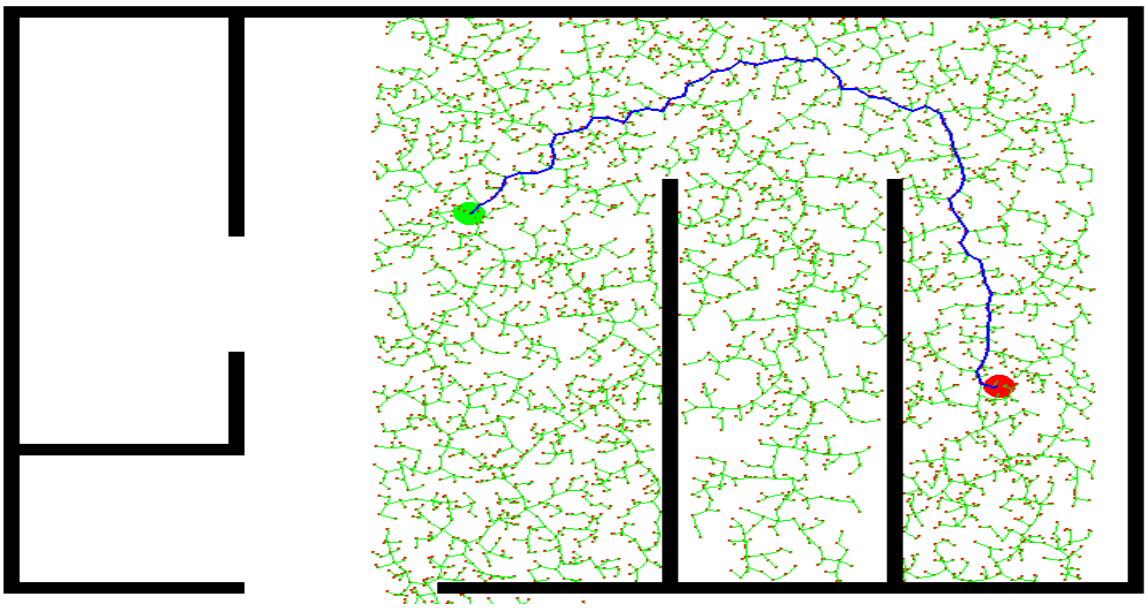


Figure3. 19:finding path by limited tree expansion method in lab 1.

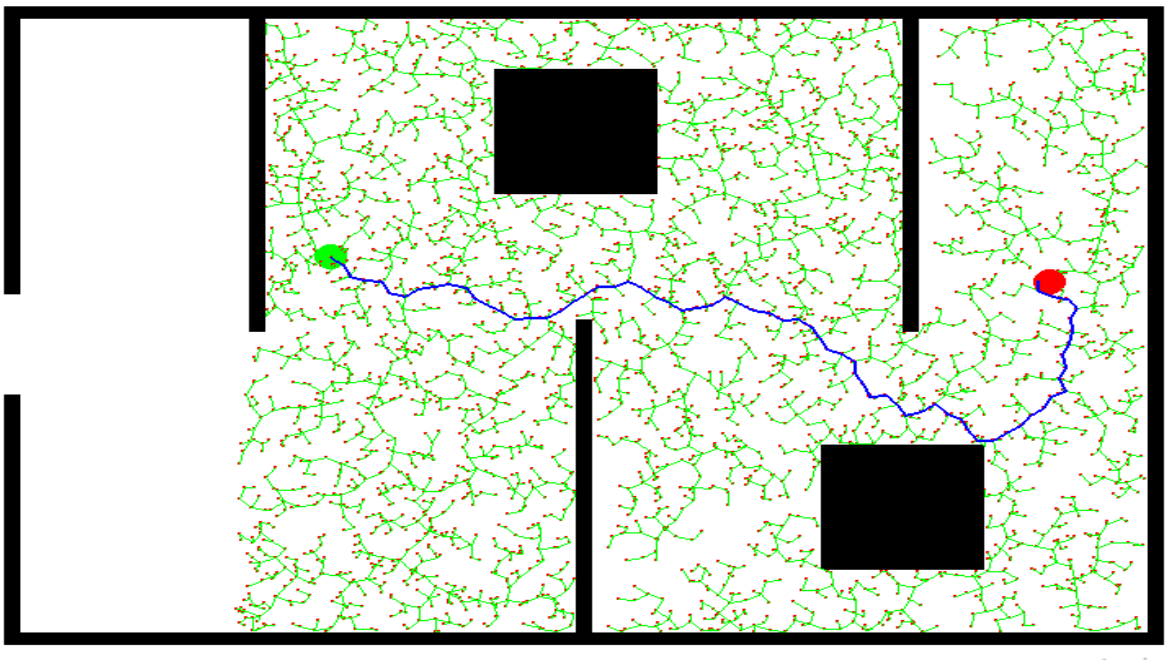


Figure3. 20:finding path by limited tree expansion method in lab 2.

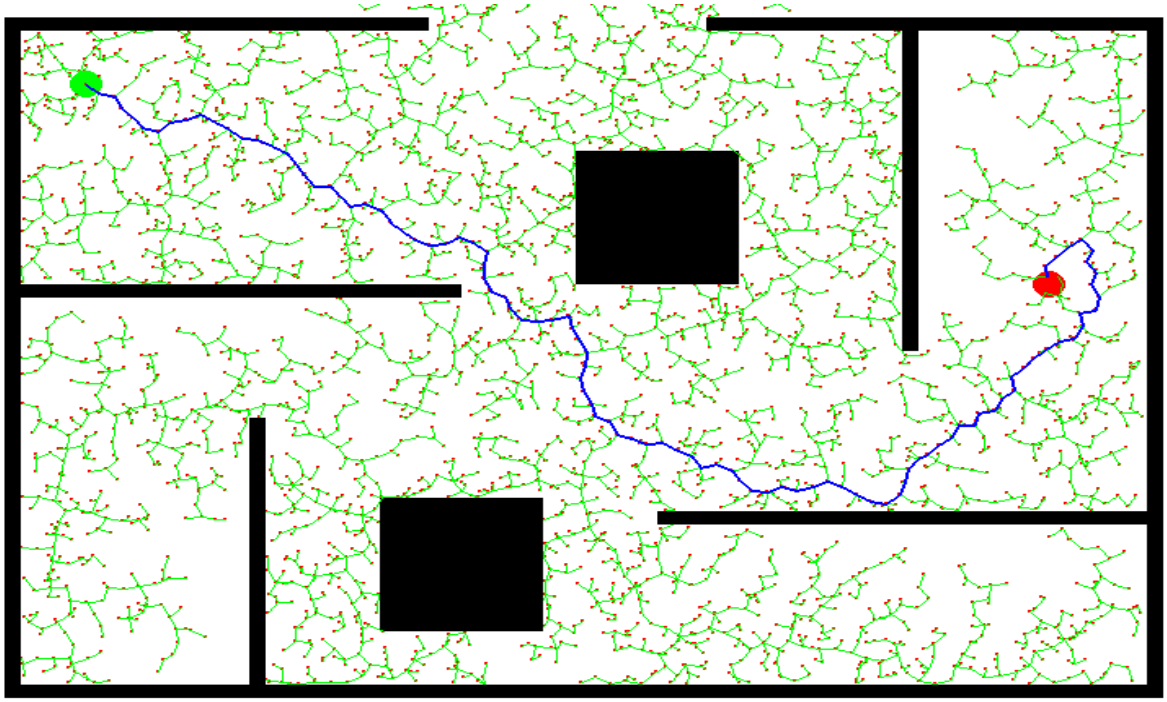


Figure3. 21:finding path by limited tree expansion method in lab 3.

3.6.1.1.1 Complexity in multiple environments.

In this study, we examine Limited tree expansion method in diverse environments to compare their performance.

we evaluated in three distinct environments to assess its efficacy in path planning from the start point (green circle) to the goal point (red circle) as show above in figure 3.19, figure 3.20 and figure 3.21. Despite encountering complexity within the environments, we observed that this method consistently provided a viable path in all cases.

3.6.1.2 path length.

the path length in the limited tree expansion method refers to the total distance covered along the generated path. It is computed by summing the lengths of all line segments connecting consecutive points in the path. The length of each segment is calculated using the Euclidean distance based on the coordinates of the points in the path. This measure helps evaluate the effectiveness of the path generated by this method in navigating from the start node to the goal node while avoiding obstacles in the environment. We evaluated the path length under the three cases of the environment, with the findings presented in the table 3.1:

	Path length (m)
Lab 1	**8.29**
Lab 2	**8.95**
Lab 3	**11.29**

Table 3 1:path length of limited tree expansion method

3.6.1.3 Execution time.

Execution time refers to the duration it takes for the program to complete its execution, The time taken by the limited tree expansion algorithm to find the path from the start of the algorithm to the completion of the pathfinding. We evaluated the execution time under the three cases of the environment, with the findings presented in the table:

	Execution time (S)
Lab 1	**8.29**
Lab 2	**8.75**
Lab 3	**9.82**

Table 3 2: Execution time of limited tree expansion method

Upon comparing the results of path length and execution time, it becomes evident that as the path length extends, so does the duration of execution.

3.6.2 Geometric method

We introduced three distinct environments to observe how the geometric method executes and reacts within each setting.

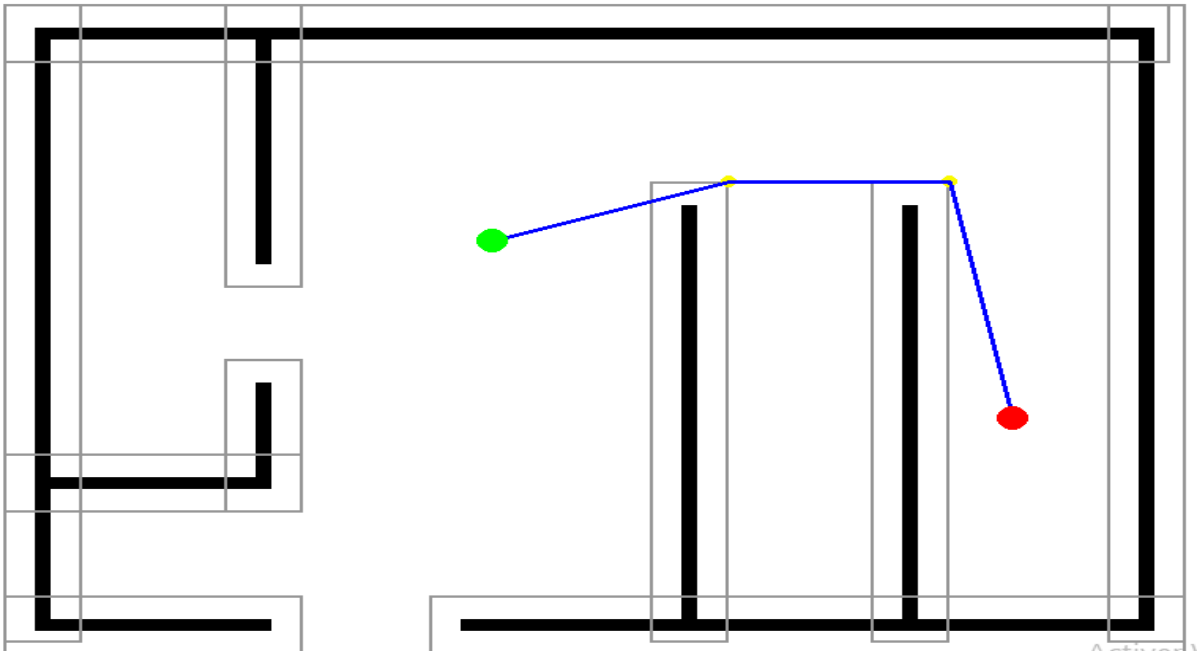


Figure3. 22:finding path by Geometric method in lab 1.

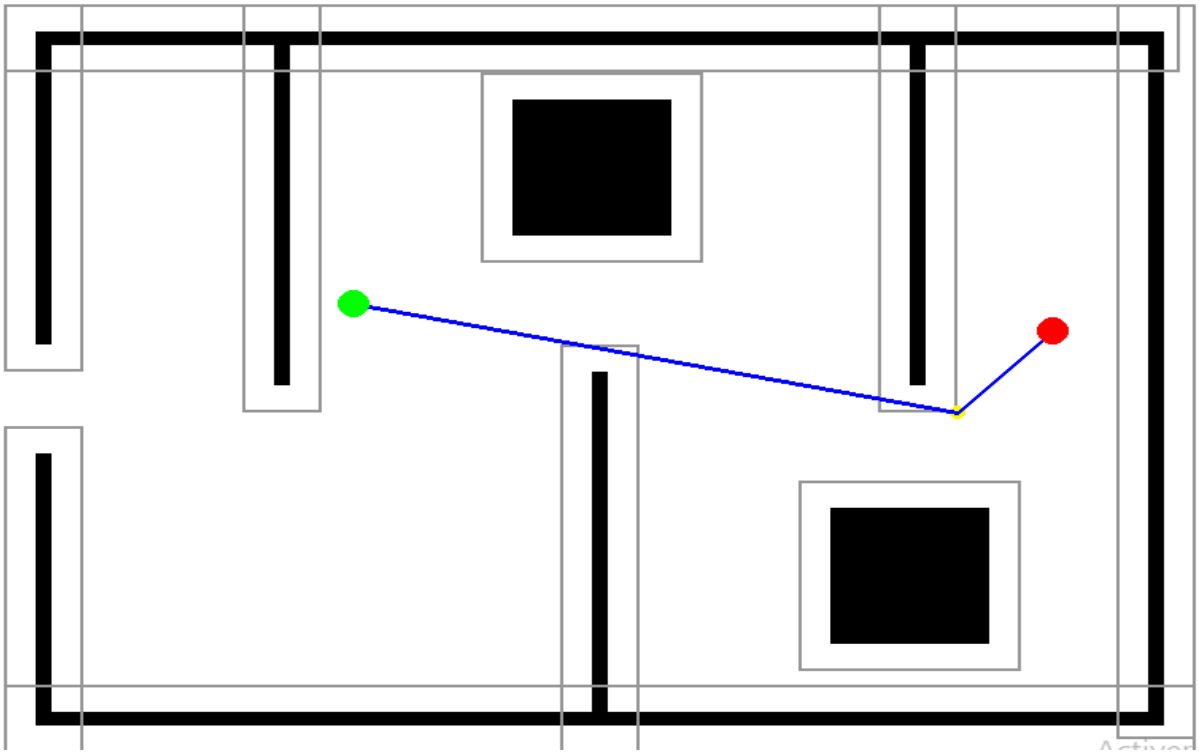


Figure3. 23:finding path by Geometric method in lab 2.

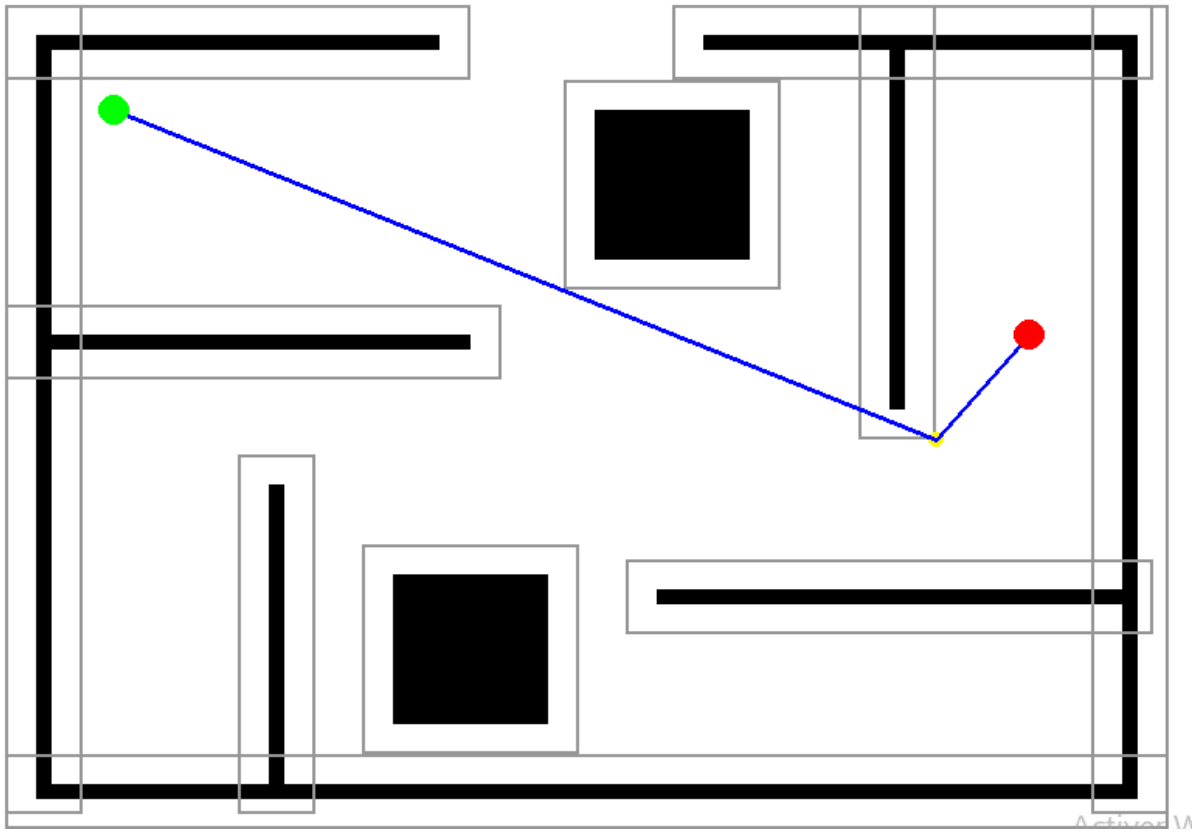


Figure3. 24:finding path by Geometric method in lab 3.

3.6.2.1 Complexity in multiple environments:

In this study, we examine Geometric method in diverse environments to compare their performance.

we evaluated in three different environments to measure its effectiveness in path planning from the start point (green circle) to the goal point (red circle) as show above in figure 3.22, figure 3.23 and figure 3.24. With escalating complexity in the environment, the method became less efficient in successfully generating a path.

3.6.2.2 path length.

Path length, in the geometric methods, refers to the cumulative distance covered along a route between two points in a space, typically measured along the curve representing the path. This measurement involves calculating the sum of distances between consecutive points or segments along the path, utilizing geometric principles such as the Euclidean distance in Cartesian coordinates. We evaluated the path length under the three cases of the environment, with the findings presented in the table:

	Path length (m)
Lab 1	**6.28**
Lab 2	**5.91**
Lab 3	**8.32**

Table 3 3:path length of geometric method

3.6.2.3 Execution time

Execution time signifies the duration it takes for the geometric method to complete its tasks. reflecting the time taken for each step and the cumulative duration of the entire process. We evaluated the execution time under the three cases of the environment, with the findings presented in the table:

	Execution time (S)
Lab 1	**0.75**
Lab 2	**0.67**
Lab 3	**0.88**

Table 3 4:Execution time of geometric method

3.6.3 RRT* algorithm

We introduced three distinct environments to observe how the RRT* algorithm executes and reacts within each setting.

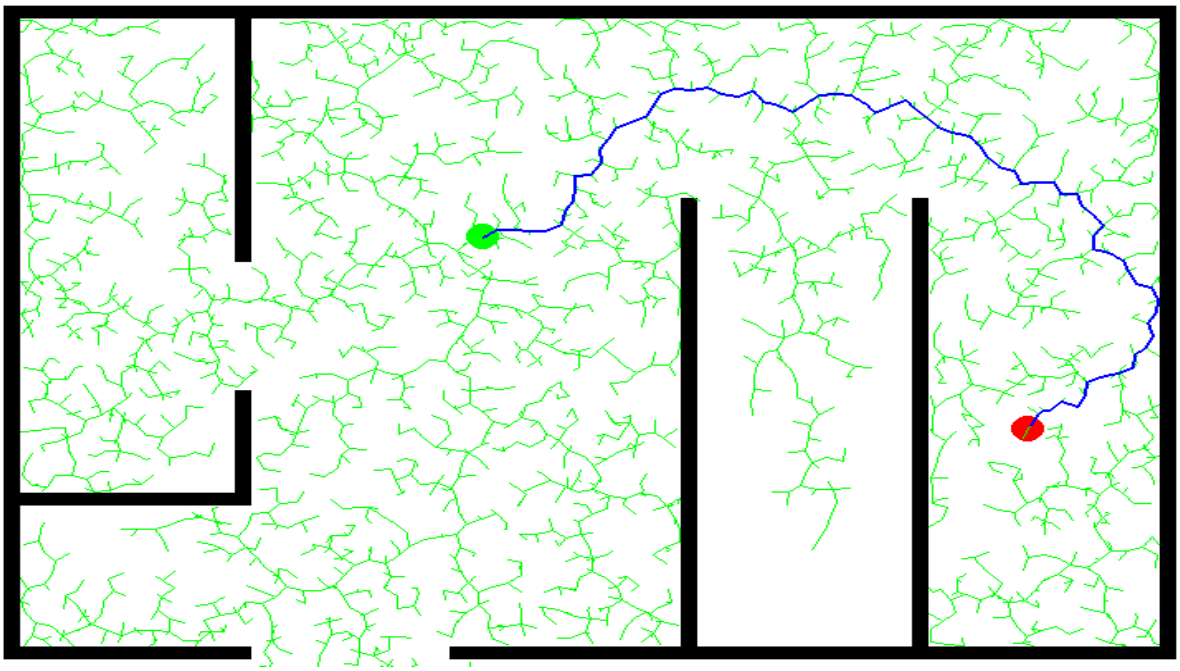


Figure3. 25:finding path by RRT* algorithm in lab1.

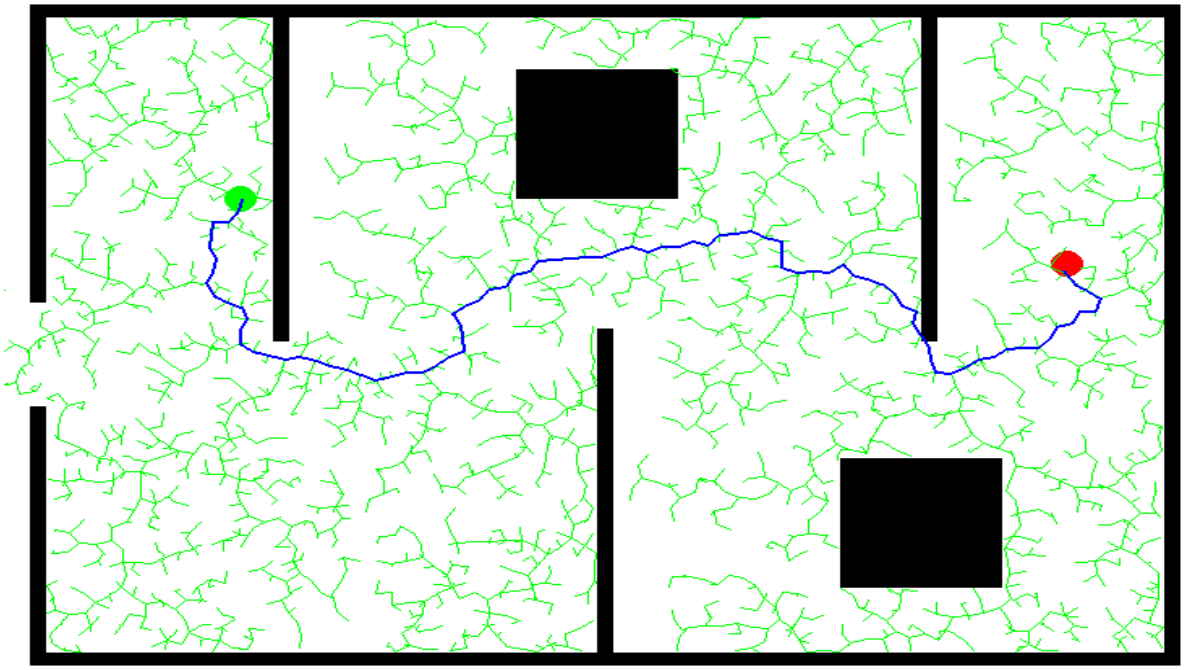


Figure3. 26:finding path by RRT*algorithm in lab 2.

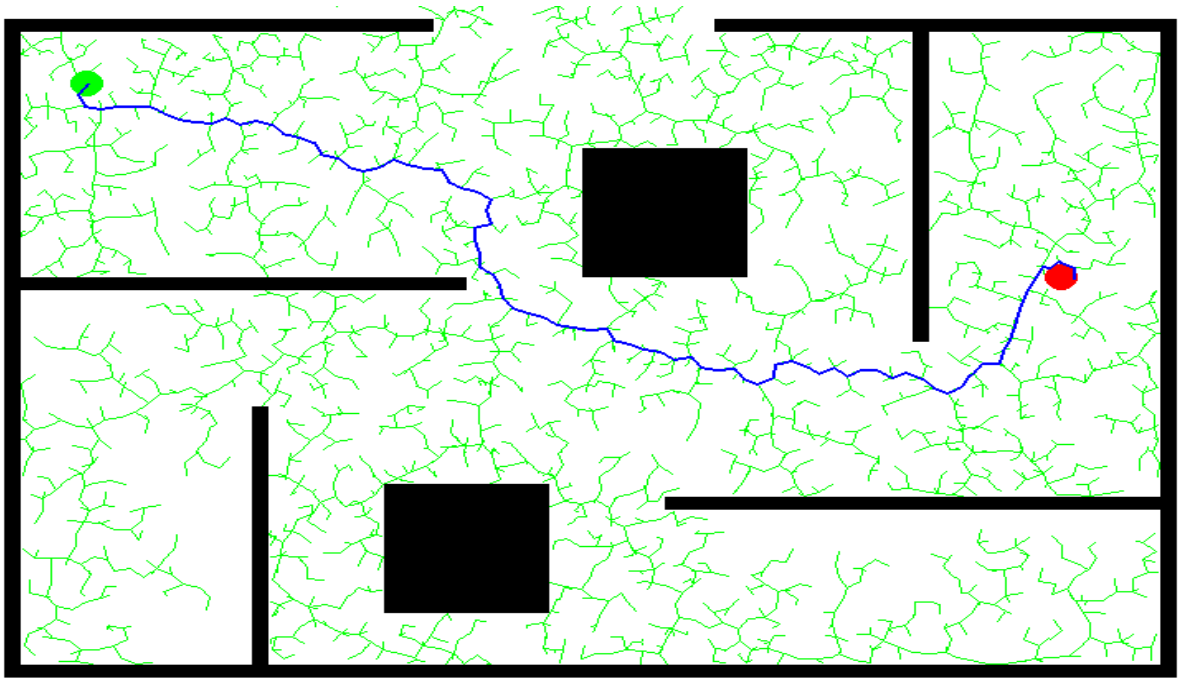


Figure3. 27:finding path by RRT* algorithm in lab 3.

3.6.3.1 Complexity in multiple environments.

In this study, we evaluated RRT*algorithm in three distinct environments to assess its efficacy in path planning from the start point (green circle) to the goal point (red circle) as show above in figure 3.25, figure 3.26 and figure 3.27. Despite encountering complexity within the environments, we observed that this algorithm consistently provided a viable path in all cases.

3.6.3.2 path length.

path length is the sum of the Euclidean distances between consecutive nodes along the path traversed by the RRT* algorithm, from the starting point to the goal point. We evaluated the path length under the three cases of the environment, with the findings presented in the table 3.5:

	Path length (m)
Lab 1	**12.19**
Lab 2	**9.2**
Lab 3	**12.35**

Table 3 5:path length of RRT* algorithm

3.6.3.3 Execution time

Execution time refers to the duration it takes for the program to complete its execution, The time taken by RRT* algorithm to find the path from the start of the algorithm to the completion of the pathfinding. We evaluated the execution time under the three cases of the environment, with the findings presented in the table below:

	Execution time (S)
Lab 1	**10.27**
Lab 2	**9.27**
Lab 3	**13.5**

Table 3 6:Execution time of RRT*algorithm

3.6.4 Results

in this section, we have conducted a comparison of all the outcomes obtained through the two methods concerning both path length generated, and the time taken for the execution of processes in one table.

	Limited tree expansion		Geometric method		RRT* algorithm	
	Path length (m).	Execution time (s)	Path length (m).	Execution time (s)	Path length (m).	Execution time (s)
Lab1	**8.29**	**8.29**	**6.28**	**0.75**	**12.19**	**10.27**
Lab 2	**8.95**	**8.57**	**5.91**	**0.67**	**9.2**	**9.27**
Lab 3	**11.29**	**9.82**	**8.32**	**0.88**	**12.35**	**13.5**

Table 3 7:comparison between limited tree expansion, geometric method and RRT* algorithm

The results obtained from comparing the two methods reveal that the geometric approach excels in optimizing path length compared to the limited tree expansion method. Additionally, in terms of time efficiency, the geometric method demonstrates a faster path-finding capability compared to the limited tree expansion approach.

Geometric methods offer several advantages over tree-based approaches. Firstly, they often involve direct calculations of distances between points, facilitating more precise path determination. Additionally, geometric methods typically operate in continuous space, allowing for smoother and more accurate path interpolation, Furthermore, these

methods frequently employ efficient search techniques, which enable rapid convergence to optimal or near-optimal solutions.

The limited tree expansion method demonstrates inferior performance compared to geometric methods in path planning for several reasons. Firstly, it relies on discrete sampling of the search space, leading to suboptimal paths, especially in environments with complex geometries or obstacles. Moreover, the discrete nature of limited tree expansion methods makes them prone to getting trapped in local minima, hindering exploration of alternative paths efficiently. Furthermore, these methods require fine-tuning of parameters such as step size, tree expansion strategy, which can be challenging and time-consuming, and may not always yield optimal path results. Finally, limited tree expansion methods may struggle with accurately interpolating paths between sampled points, resulting in jagged or suboptimal trajectories, particularly in environments with intricate geometries or sharp turns.

On the other hand, the Limited Tree Expansion method proves to be more efficient compared to the RRT* algorithm due to its shorter execution time. By strategically expanding the tree only within the vicinity of the start and goal points, and incrementing expansion only when encountering obstacles, this method significantly reduces the time required to find the optimal path.

Finally, each method has its own set of advantages and limitations, making them suitable for different applications and environments. The RRT* algorithm excels in finding optimal paths in complex and dynamic environments but comes with higher computational costs. The Limited Tree Expansion method offers a balance between efficiency and optimality, making it suitable for challenges requiring faster execution times. The Geometric method, while simpler and faster in certain contexts, may lack the robustness and adaptability needed for more challenging environments. Thus, the choice of method depends on the specific requirements and characteristics of the given challenges.

3.7 Conclusion

In conclusion, this chapter has demonstrated the capabilities and limitations of the Limited Tree expansion RRT* and geometric concept-based methods for path planning and obstacle avoidance. The performance analysis revealed that the Limited Expansion Tree RRT* excels in environments where path quality and computational efficiency are paramount, thanks to its ability to limit unnecessary expansion of the search tree. In contrast, the geometric method stands out for its simplicity and efficiency in structured environments where obstacles can be represented geometrically.

Quantitative comparisons showed that each method has its optimal application domains, depending on the complexity of the environment and the specific mission requirements. The obtained results provide a solid basis for choosing the most appropriate method based on operational constraints and mission objectives. These findings also open up future improvement opportunities, such as combining both approaches to leverage their respective advantages and overcome their limitations. Thus, this chapter has not only presented a thorough evaluation of two distinct methods but also provided valuable insights for the future development of robust and efficient navigation solution

General Conclusion

The methods presented in this thesis represent significant advancements in the field of autonomous mobile robot (AMR) navigation. By introducing enhanced variants of the Rapidly exploring Random Tree Star (RRT*) algorithm, such as the Limited Expansion Tree RRT*, along with our new geometric method, we have demonstrated notable improvements in trajectory quality. These methods outperform the original RRT* algorithm by providing more efficient and optimized trajectories, which are crucial for autonomous navigation in complex environments.

One of the key advantages of these new approaches is their ability to balance trajectory planning efficiency with computational complexity, making robots more responsive and adaptable to dynamic changes in their environment. Additionally, the use of geometric techniques allows for better obstacle handling and the planning of safer and more precise paths.

The results obtained highlight the considerable potential of these methods to enhance AMR capabilities. In the future, the development of these techniques can benefit from the integration of parallelization and advanced algorithmic optimization. This will not only improve the efficiency and effectiveness of these methods but also make them more robust and adaptable to a wider range of real-world scenarios.

In conclusion, the innovations presented in this thesis open new avenues for autonomous mobile robot navigation. They establish a solid foundation for future research aimed at fully exploiting the potential of AMRs in various industrial, commercial, and service applications, thereby contributing to the ongoing advancement of autonomous robotics

References

[1] https://www.britannica.com/technology/robotics

[2] https://link.springer.com/chapter/10.1007/978-3-319-62533-1_1

[3] What is a mobile robot? Definition from WhatIs.com. (Techtarget.com)

[4] Path Planning - MATLAB & Simulink (mathworks.com)

[5] https://memgraph.com/blog/graph-search-algorithms-developers-guide

[6] https://memgraph.com/blog/graph-search-algorithms-developers-guide

[7] Depth First Search (DFS) C++ Program To Traverse A Graph Or Tree (softwaretestinghelp.com)

[8] Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction (freecodecamp.org)

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

[10] Aurenhammer, F., 1991. Voronoi diagrams-a survey of a fundamental geometric data structure. ACM Computing Surveys (CSUR), 23(3), pp.345-405.

[11] Lozano-Pérez, T. and Wesley, M.A., 1979. An algorithm for planning collision-free paths among polyhedral obstacles. Communications of the ACM, 22(10), pp.560-570.

[12] Methodology for Path Planning and Optimization of Mobile Robots: A Review Mohd. Nayab

[13] https://www.cs.cmu.edu/~motionplanning/lecture/lec20.pdf

[14] [11] https://en.wikipedia.org/wiki/Obstacle_avoidance

[15] https://www.sciencedirect.com/science/article/pii/S0019057823000769

[16] Robotic Motion Planning: Bug Algorithms Robotics Institute 16-735 http://www.cs.cmu.edu/~motion

[17] Sertac Karaman, Emilio Frazzoli. Sampling-based Algorithms for Optimal Motion Planning

[18] Fahad Islam1,2, Jauwairia Nasir1,2 , Usman Malik1 , Yasar Ayaz1 and Osman Hasan. RRT*-Smart: Rapid convergence implementation of RRT* towards optimal solution

[19] Jonathan D. Gammell1 , Siddhartha S. Srinivasa2 , and Timothy D. Barfoot1. Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic

[20] Yicong Guo 1 , Xiaoxiong Liu 1,2,*, Xuhang Liu 1 , Yue Yang 1 and Weiguo Zhang 1,2. FC-RRT*: An Improved Path Planning Algorithm for UAV in 3D Complex Environment

[21] Min Yu, ∗ Jianjin Luo, Mingming Wang, Dengwei Gao. Spline-RRT*: Ccoordinated Motion Planning of Dual-Arm Space Robot

[22] Jianyou Qi1 · Qingni Yuan1,2 · Chen Wang1 · Xiaoying Du1 · Feilong Du1,2 · Ao Ren1. Path planning and collision avoidance based on the RRT*FN framework for a robotic manipulator in various scenarios.

[23] PyCharm at Python US 2024: Engage, Learn, and Celebrate! | The PyCharm Blog (jetbrains.com)

[24] https://www.python.org/doc/essays/blurb/

[25] https://datascientest.com/pygame-tout-savoir