Multi-Robot Coordination and Path Planning

Nithish Ravikkumar, Prakul Balaji (under guidance of Dr.Felix Orlando)

May 11, 2024

Abstract

This report presents an exploration into the coordination and path planning of multi-robot systems (MRS), focusing primarily on coordination and perception domains. The project aims to implement various multi-robot path planning algorithms and evaluate their performance in a simulated environment.

Contents

Chapter 1

Introduction

1.1 Multi-Robot Systems (MRS)

It has been observed in many instances that multiple robots cooperate to perform complex tasks that would otherwise be impossible for one single powerful robot to accomplish. The fundamental theory behind multi-agent robotics suggests dispatching smaller sub-problems to individual robots in a group and allowing them to interact with each other to find solutions to complex problems. Simple robots can be built and made to cooperate with each other to achieve complex behaviors. It has been observed that multi-robot systems (MRS) are very cost effective as compared to building a single costly robot with all the capabilities. As these systems are usually decentralized, distributed and inherently redundant, they are fault tolerant and improve the reliability and robustness of the system.

1.2 Domains of Multi-Robot Systems

MRS can be broadly categorized into four domains based on their functionality and coordination requirements:

- **Coordination:** Focuses on the synchronization and collaboration among multiple robots to achieve a common goal.Coordination in multi-robot systems is crucial to ensure that robots can work together efficiently without causing conflicts or redundancy.
- **Communication:** Concerned with establishing and maintaining communication links between robots. Communication is essential for enabling robots to share information, coordinate actions, and collaborate effectively in a multi-robot environment.
- **Sensory Perception:** Involves the integration and interpretation of sensory data for environment understanding. This domain focuses on integrating various sensors, such as cameras, lidar, and ultrasonic sensors, to gather environmental data. Advanced algorithms are then used to process and interpret this sensory data, providing robots with the necessary information to make informed decisions and navigate complex environments.
- **Learning:** Learning capabilities enable robots to improve their performance over time by learning from past experiences, making predictions, and adapting to new situations. This involves implementing machine learning algorithms and reinforcement learning techniques that allow robots to recognize patterns, optimize their strategies, and learn new tasks.

For this project, the primary focus is on the **Coordination** and **Sensory Perception** domains.

1.3 Objective

The primary objective of this project is to explore and implement various multi-robot path planning algorithms, and observe the results in a computer simulation. The project aims to:

- Investigate the feasibility and effectiveness of different path planning approaches in a simulated environment.
- Implement a suitable algorithm for collecting particles scattered on the ground while avoiding collisions.

1.4 Scope

The scope of this project encompasses:

- Designing and developing a software-based infrastructure for multi-robot coordination and path planning.
- Implementing and testing multi-robot path planning algorithms focusing on coordination and perception domains.

1.5 Motivation

The motivation behind this project is the growing interest and demand for effective multi-robot coordination and path planning solutions. With the advancements in robotics and AI technologies, there is an increasing need for robust, scalable, and adaptive MRS to tackle complex real-world challenges.

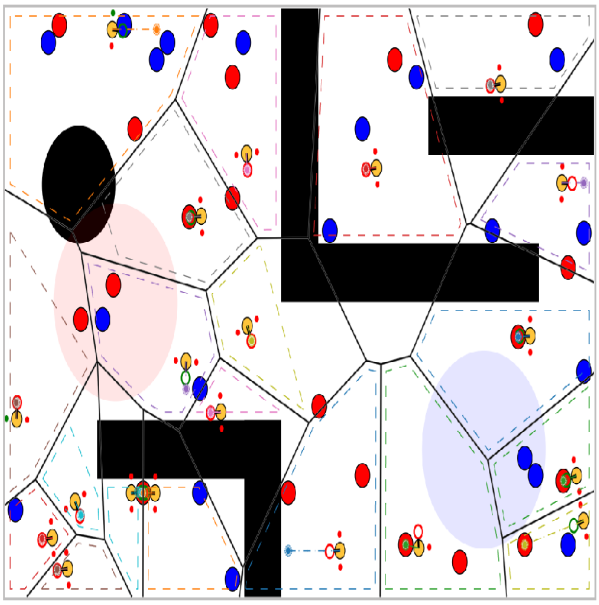

Figure 1.1: Multiple robot coordination

Chapter 2

Multi-Robot Path Planning Algorithms

Path planning is an essential aspect of multi-robot systems, playing a pivotal role in facilitating efficient navigation and coordination among multiple robots operating in a shared environment. To address the challenges inherent in multi-robot path planning, this project delves into the exploration and implementation of four distinct algorithms. Each of these algorithms offers a unique approach, bringing its own set of advantages and disadvantages to the table. In the subsequent sections, we delve deep into these algorithms, shedding light on their methodologies, strengths, and limitations.

2.1 Method 1 -Overhead Camera-Based Multi-Robot Path Planning

2.1.1 Introduction:

The overhead camera-based approach leverages image processing techniques to enable robots to navigate and coordinate within an environment. In a scenario where mobile robots have to navigate through an open field without using a pre-existing map, this method will be useful. The overhead camera will be present in a drone which will be used to capture vertical image of the entire field. By analyzing this image, the program would identify and categorize objects, such as robots, obstacles, and target objects, to facilitate multi-robot path planning.

Functions used:

2.1.2 analyze_image Function

The `analyze_image` function is responsible for processing an overhead image to identify and categorize objects within grids.

Inputs:

- `image_path`: Path to the overhead image
- `obstacle_size_threshold`: Minimum size (in pixels) to consider something an obstacle.

Steps:

1. Read the image.
2. Loop through each grid:
    - Identify target objects based on the green color range.
    - Identify obstacles based on the red color range.
    - Detect Aruco markers for robots.
3. Save the identified objects and their coordinates in a YAML file named "object_data.yaml".

2.1.3 find_and_crop_white_boundary Function

The `find_and_crop_white_boundary` function identifies the white boundary bounding box in an image and saves the cropped region. **Inputs:**

- `original_image_path`: Path to the overhead image.
- `image_path`: Path to save the cropped image.

Steps:

1. Read the image.
2. Convert the image to grayscale.
3. Binarize the image based on a threshold.
4. Find the largest white contour, which represents the boundary.
5. Crop the image based on the bounding rectangle of the contour.
6. Save the cropped image.

2.1.4 CBS (Conflict-Based Search) Algorithm

The CBS algorithm is a two-level search algorithm used for multi-agent pathfinding. At the high level, a search is performed on a Conflict Tree (CT) which is a tree based on conflicts between individual agents. Each node in the CT represents a set of constraints on the motion of the agents. At the low level, fast single-agent searches are performed to satisfy the constraints imposed by the high level CT node. **Classes:**

- `Location`: Represents a location with x and y coordinates.
- `State`: Represents a state with a time and a location.
- `Conflict`: Represents a conflict between agents.
- `VertexConstraint & EdgeConstraint`: Constraints for vertices and edges respectively.
- `Constraints`: Collection of vertex and edge constraints.
- `Environment`: Represents the environment with agents, obstacles, and constraints. It uses A* for pathfinding.
- `HighLevelNode`: Represents a node in the high-level search.
- `CBS`: Implements the CBS algorithm.

Functions:

- `get_neighbors`: Returns neighboring states from a given state.
- `get_first_conflict`: Finds the first conflict in a solution.
- `create_constraints_from_conflict`: Creates constraints based on a conflict.
- `compute_solution`: Computes a solution for the agents using A*.
- `compute_solution_cost`: Calculates the cost of a given solution.

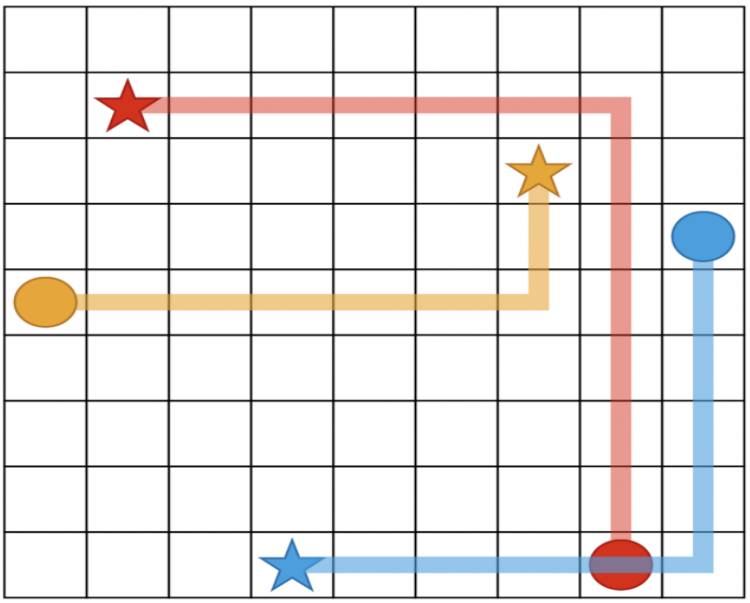

Figure 2.1: CBS - collision avoidance

2.1.5 A* Search Algorithm

A* is a popular pathfinding algorithm used to find the shortest path from a start node to a goal node. It introduces a heuristic into a regular graph-searching algorithm, essentially planning ahead at each step so a more optimal decision is made. **Classes:**

- `AStar`: Implements the A* search algorithm.

Functions:

- `reconstruct_path`: Reconstructs the path from the start to the goal.
- `search`: Searches for a path from the start state to the goal state using A*.

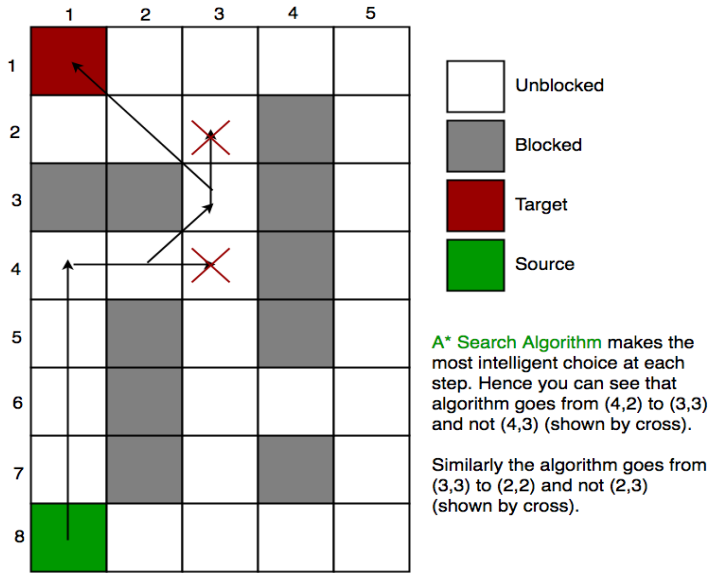

Figure 2.2: A* search algorithm

2.1.6 Hardware implementation of method

We initially aimed to implement a PID controlled 4-wheeled mecanum wheel robot using an Arduino microcontroller. Mecanum wheels allow the robot to move in any direction with the ability to rotate around its axis. The PID control ensures precise control over the robot's movement. This kind of a setup would allow the robot to move in a square n x n grid in straight lines, pick up objects at any cell, and drop them at its destination.

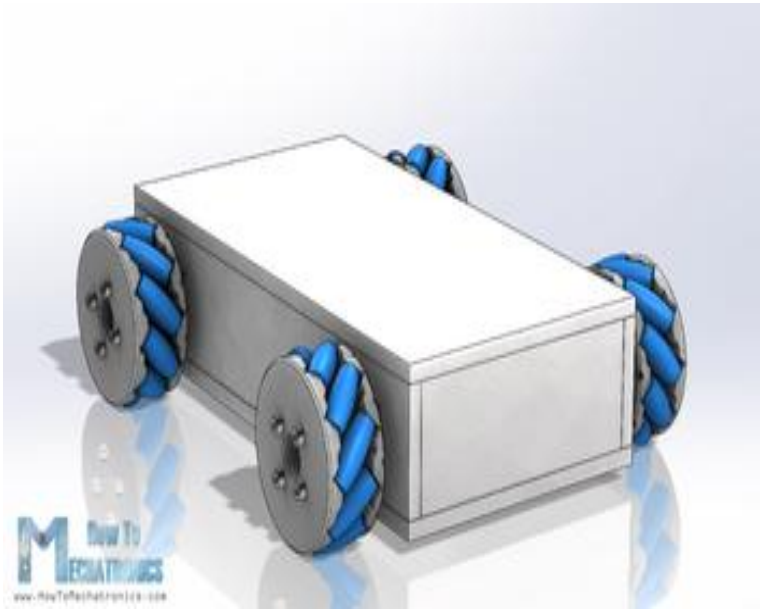

Figure 2.3: Mecanum wheels

Code Implementation

Pin Definitions

```
// Motor pin definitions
#define ENB_FR 8
#define IN_FR1 22
#define IN_FR2 23
// ... (other motor pin definitions)

// Encoder pin definitions
#define ENCODER_PIN_FR_A 2
#define ENCODER_PIN_FR_B 4
// ... (other encoder pin definitions)
```

Setup Function In the setup() function, motor and encoder pins are configured, interrupts are attached to encoder pins, and serial communication is initialized.

```
void setup() {
  // Motor and encoder pin configurations
  // ...

   //Interrupt sequence
  attachInterrupt(digitalPinToInterrupt(ENCODER_PIN_FR_A), updateEncoder, RISING);
  attachInterrupt(digitalPinToInterrupt(ENCODER_PIN_FL_A), updateEncoder, RISING);
  attachInterrupt(digitalPinToInterrupt(ENCODER_PIN_BR_A), updateEncoder, RISING);
  attachInterrupt(digitalPinToInterrupt(ENCODER_PIN_BL_A), updateEncoder, RISING);

  Serial.begin(9600);
}
```

Main Loop The loop() function calculates the target encoder counts based on the direction and updates the motor speeds using PID control until the robot reaches the target counts.

```
void loop() {
```

```
  calculateTargetCounts(direction);

  while (!areEncodersAtTarget()) {
    updateEncoder();
    for (int i = 0; i < 4; i++) {
      motorControlPID(i, targetCountPerWheel[i]);
    }

    delay(sampleTime);
  }
  stop();
}
```

Helper Functions

- `calculateTargetCounts(int dir)`: Calculates target encoder counts based on the direction.
- `areEncodersAtTarget()`: Checks if all encoder counts have reached the target counts.
- `updateEncoder()`: Updates the encoder counts using interrupts.
- `motorControlPID(int motorIndex, long targetCount)`: Implements PID control for each motor.
- `Motion functions (forward(), backward(), rotateL(), rotateR(), stop())`: Control the robot's motion based on the calculated PID control signal.

```
void updateEncoder() {
  vector<int> l[4] ={ENCODER_PIN_FR_B,ENCODER_PIN_FL_B,ENCODER_PIN_BR_B,ENCODER_PIN_BL_B};
  for (i in l){
  int b = digitalRead(i);
  if (b > 0) {
    encoderCount[getIndex(vector<int> l, int i)]+=1;
  }
  else {
    encoderCount[getIndex(vector<int> l, int i))]-=1;
  }
  }
}

////////////////////////////

void motorControlPID(int motorIndex, long targetCount, int direction) {

  double error[4] = targetCount - encoderCount;
  motorIntegral[motorIndex] += error;
  double derivative = ((error - motorError[motorIndex]) * 1000) / sampleTime;
  motorError[motorIndex] = error;
  double controlSignal = Kp * error + Ki * motorIntegral[motorIndex] + Kd * derivative;


  for (int i =0, i<4, i++ ) {

   // Control motor based on control signal
  if ((abs(targetCount[i]-encoderCount[i]))<(0.03*(targetCount[i]))){
    stop();
  }
  else if (controlSignal[i] > 0){
    analogWrite(ENB, controlSignal[i]); // Set PWM value for speed control
    digitalWrite(IN3, HIGH); // Rotate motor in one direction
    digitalWrite(IN4, LOW);
    EnableMotion(int direction);

  }
  else if (controlSignal[i] < 0) {
    analogWrite(ENB, abs(controlSignal[i])); // Set PWM value for speed control (positive value)
    digitalWrite(IN3, LOW); // Rotate motor in the other direction
    digitalWrite(IN4, HIGH);
```

```
    EnableMotion(int direction);
  }
  else {
    stop();
  }

  prevError = error;

  // Sampling time delay
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= sampleTime) {
    previousMillis = currentMillis;
  }
  }
}
////////////////////////////////

void forward(){
  analogWrite(ENB_FR,speed);
  analogWrite(ENB_FL,speed);
  analogWrite(ENB_BR,speed);
  analogWrite(ENB_BL,speed);

  digitalWrite(IN_FR1,LOW);
  digitalWrite(IN_FR2,HIGH);
  digitalWrite(IN_FL1,LOW);
  digitalWrite(IN_FL2,HIGH);
  digitalWrite(IN_BR1,LOW);
  digitalWrite(IN_BR2,HIGH);
  digitalWrite(IN_BL1,LOW);
  digitalWrite(IN_BL2,HIGH);

  delay(1000);
  }


void backward(targetCountPerWheel){
  analogWrite(ENB_FR,speed);
  analogWrite(ENB_FL,speed);
  analogWrite(ENB_BR,speed);
  analogWrite(ENB_BL,speed);

  digitalWrite(IN_FR1,HIGH);
  digitalWrite(IN_FR2,LOW);
  digitalWrite(IN_FL1,HIGH);
  digitalWrite(IN_FL2,LOW);
  digitalWrite(IN_BR1,HIGH);
  digitalWrite(IN_BR2,LOW);
  digitalWrite(IN_BL1,HIGH);
  digitalWrite(IN_BL2,LOW);

  delay(1000);
  }

void rotateL(targetCountPerWheel){
  analogWrite(ENB_FR,speed);
  analogWrite(ENB_FL,speed);
  analogWrite(ENB_BR,speed);
  analogWrite(ENB_BL,speed);

  digitalWrite(IN_FR1,LOW);
  digitalWrite(IN_FR2,HIGH);
  digitalWrite(IN_FL1,HIGH);
  digitalWrite(IN_FL2,LOW);
  digitalWrite(IN_BR1,LOW);
  digitalWrite(IN_BR2,HIGH);
```

```
  digitalWrite(IN_BL1,HIGH);
  digitalWrite(IN_BL2,LOW);

  delay(1000);
  }

void rotateR(targetCountPerWheel){
  analogWrite(ENB_FR,speed);
  analogWrite(ENB_FL,speed);
  analogWrite(ENB_BR,speed);
  analogWrite(ENB_BL,speed);

  digitalWrite(IN_FR1,HIGH);
  digitalWrite(IN_FR2,LOW);
  digitalWrite(IN_FL1,LOW);
  digitalWrite(IN_FL2,HIGH);
  digitalWrite(IN_BR1,HIGH);
  digitalWrite(IN_BR2,LOW);
  digitalWrite(IN_BL1,LOW);
  digitalWrite(IN_BL2,HIGH);

  delay(1000);
  }


void stop(){
  analogWrite(ENB_FR,0);
  analogWrite(ENB_FL,0);
  analogWrite(ENB_BR,0);
  analogWrite(ENB_BL,0);

  digitalWrite(IN_FR1,HIGH);
  digitalWrite(IN_FR2,HIGH);
  digitalWrite(IN_FL1,HIGH);
  digitalWrite(IN_FL2,HIGH);
  digitalWrite(IN_BR1,HIGH);
  digitalWrite(IN_BR2,HIGH);
  digitalWrite(IN_BL1,HIGH);
  digitalWrite(IN_BL2,HIGH);
}
```

2.1.7 Summary:

The overhead camera-based approach employs image processing techniques to identify robots, obstacles, and target objects within an environment. The `analyze_image` function processes overhead images to categorize objects and saves their locations in a YAML file. The CBS algorithm manages multi-agent pathfinding by resolving conflicts between agents using A* for individual pathfinding. Then, this solution can be implemented using a mecanum wheel based robot. Overall, this method provides a comprehensive system for multi-agent pathfinding in an environment observed by an overhead camera, facilitating coordinated and efficient robot navigation.

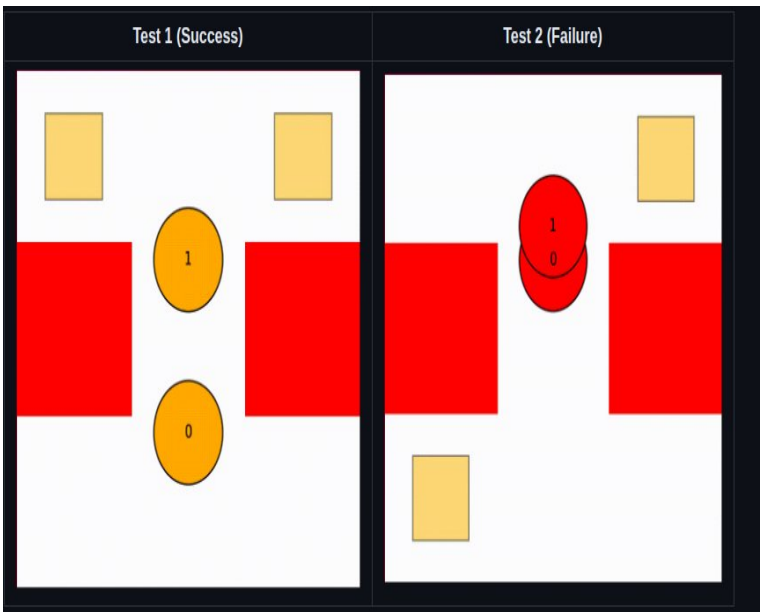

Figure 2.4: Pass and fail scenarios for mapping

2.2 Method 2 - Ultrasound SLAM-Based Multi-Robot Path Planning

2.2.1 Introduction:

Ultrasound-based Simultaneous Localization and Mapping (SLAM) has gained significant attention in the robotics community for its ability to enable robots to navigate and map unknown environments. This method utilizes ultrasound sensors to measure distances to obstacles, creating a digital map of the environment while tracking the robot's location within it. In this section, we delve into the implementation and analysis of Ultrasound SLAM for multi-robot path planning.

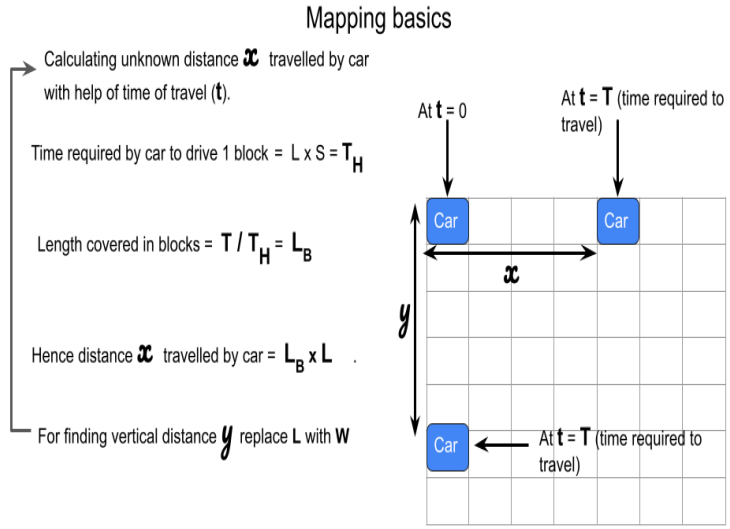

Figure 2.5: Ultrasound SLAM mapping

Methodology:

2.2.2 Sensor input

Each robot in the multi-robot system is equipped with an ultrasound sensor responsible for collecting distance measurements to nearby obstacles. The sensor emits ultrasound waves and calculates the distance based on the time taken for the waves to return after hitting an object. This data is crucial for building an accurate digital map of the environment.

2.2.3 Map Creation

The collected ultrasound sensor readings are continuously processed to build a detailed digital map of the environment. This map includes information about the location of obstacles, the position of the robots, and other relevant features. As the robots move through the environment, the map is updated in real-time to reflect any changes in the surroundings.

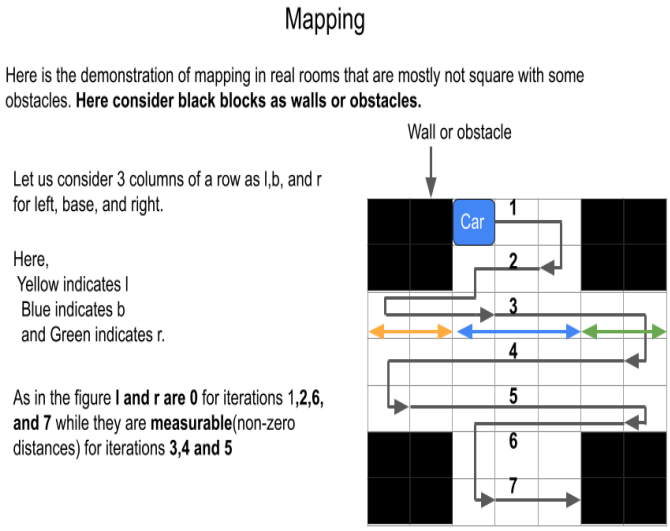

Figure 2.6: Map creation

2.2.4 Path Planning

The digital map generated through Ultrasound SLAM serves as input to the path planning algorithm. This algorithm calculates optimal paths for each robot to navigate from their current locations to their respective destinations. The paths are designed to avoid collisions with obstacles and other robots, ensuring safe and efficient navigation.

2.2.5 Robot Movement

Based on the calculated paths, each robot receives movement instructions to execute the planned trajectories. The robots adjust their speeds and directions according to the paths, facilitating coordinated movement and efficient object collection in the shared environment.

2.3 Implementation (Code Explanation):

Initialization

```
void setup() {
  servo.attach(10);
  servo.write(80);
  initializeUS();
}
```

The setup function initializes the ultrasound sensor and servo motor, which is used for sensor positioning.

Main Loop

```
void loop() {
  // Robot movement and sensor readings
}
```

The main loop controls the robot's movements and reads ultrasound sensor data to update the digital map.

Ultrasound Sensor Reading

```
int readUS() {
  // Ultrasound sensor reading code
}
```

The readUS function reads the distance from the ultrasound sensor and converts it to centimeters.

Movement Functions

```
void move_f() {
  // Forward movement code
}

void turn_r() {
  // Right turn code
}

void turn_l() {
  // Left turn code
}
```

These functions control the robot's movement based on sensor readings and path planning instructions.

Map Storage

```
void addReading(int i){
  // Sensor reading storage code
}

void writeMap(){
  // Map writing code
}

void readMap(){
  // Map reading code
}
```

These functions handle the storage and retrieval of ultrasound sensor readings, which are used to build and update the digital map.

2.3.1 Advantages and Disadvantages:

Advantages:

- Robust environment perception using ultrasound sensors.
- Real-time digital map generation and updating.
- Efficient path planning for safe and coordinated navigation.

Disadvantages:

- Limited range and field of view of ultrasound sensors.
- Vulnerable to sensor noise and inaccuracies.
- Complexity in calibration and setup for optimal performance.

2.3.2 Conclusion:

Ultrasound SLAM-based multi-robot path planning offers a promising approach for enabling intelligent and autonomous multi-robot systems in shared environments. Despite its challenges, such as sensor limitations and calibration complexities, the method's robust environment perception and real-time map updating capabilities make it a viable solution for various applications. Future work may focus on enhancing the system's robustness and scalability to handle larger and more complex environments effectively.

2.4 Method 3 - Clustering + CL-CBS Method for Multi-Robot Path Planning

2.4.1 Introduction:

The Clustering + CL-CBS approach combines particle clustering to group points of interest and the CL-CBS algorithm for collision-free path planning of multiple robots. This method optimizes robot movement by efficiently allocating clusters and generating collision-free paths.

Functions and files: This approach integrates three pivotal components: Particle Clustering, Path Planning, and Visualization. This method optimizes task allocation, path generation, and provides an animated representation for enhanced understanding and analysis.

2.4.2 Particle Clustering

The Particle Clustering component employs the AgglomerativeClustering algorithm to cluster particles based on their proximity. Agglomerative Hierarchical Clustering is a method used to cluster data points based on their similarity. In the context of multi-robot path planning, this technique is employed to group together particles or obstacles in the environment to facilitate efficient task allocation and collision avoidance.

Input: A set of particles X generated within a 20.0×20.0 map.
Output: n clusters, corresponding to the number of robots.

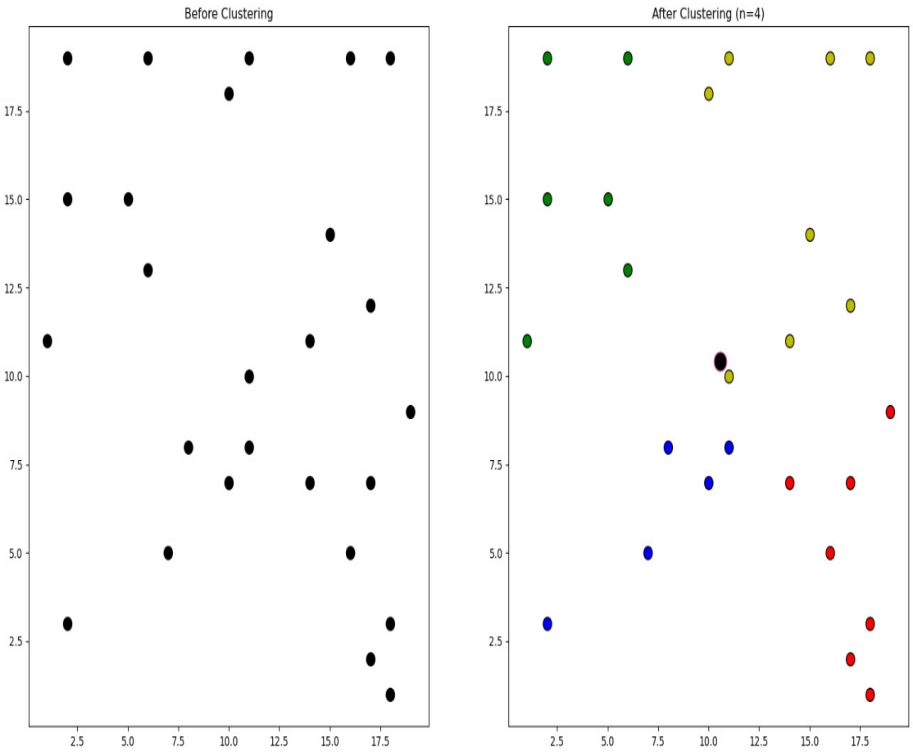

Figure 2.7: Before and After Clustering

Algorithm Steps:

- **Distance Calculation:** The `dist_calc` function computes the distance between points.
- **Agglomerative Clustering:** Utilize the AgglomerativeClustering algorithm to cluster particles into n clusters.
- **Cluster Center Calculation:** Determine the center of mass (com) for each cluster.
- **Merge Clusters:** Iteratively merge the two closest clusters based on a specified linkage criterion (e.g., average linkage, single linkage, complete linkage).
- **Stop Criteria:** Continue merging clusters until a specified number of clusters is reached or until all clusters are merged into a single cluster.
- **Assign Robots:** Assign each cluster to a robot and define their start and end positions.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn import datasets

X = np.unique(np.random.randint(10, size=(20,2)),axis=0)
n=2
clustering = AgglomerativeClustering(n_clusters=n).fit(X)
labels = clustering.labels_
d={0:[],1:[]}


print(labels)

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)

# Plot result

# Black removed and is used for noise instead.
unique_labels = set(labels)
print(unique_labels)
colors = ['y', 'b', 'g', 'r']

for i in range(len(X)):
    d[labels[i]].append(X[i])
    col=colors[labels[i]]
    plt.plot(X[i][0], X[i][1], 'o', markerfacecolor=col,
         markeredgecolor='k',
         markersize=6)

print(len(d[0]),len(d[1]))

plt.title('number of clusters: %d' % n_clusters_)
plt.show()
```

Role in Collision Avoidance:

- **Task Allocation:** Once particles or obstacles are clustered, each cluster can be assigned to a robot, reducing the number of collision points and facilitating smoother navigation.
- **Collision Prediction:** Clustering enables better prediction and avoidance of potential collisions by considering the collective movement of clusters rather than individual particles.

2.4.3 Visualization

Visualization employs matplotlib to depict robot and object movements over the map.

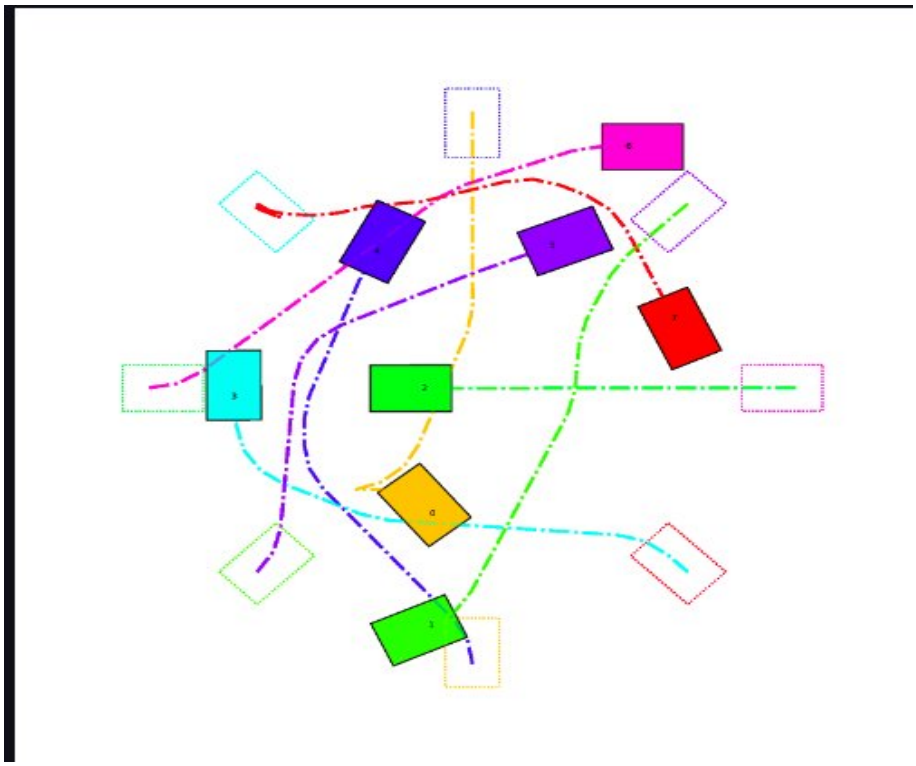

Figure 2.8: CBS visualization

Input: Map details, robot schedules, and object removal times.
Output: Animated visualization illustrating robot movements over map. However, objects are not visible

2.4.4 cl_cbs_algorithm.cpp

This file implements the CL-CBS algorithm for multi-agent pathfinding considering conflicts and constraints.

Structs:

- `State`: Represents the robot's position, orientation, and other parameters.
- `Environment`: Defines environment constraints, obstacles, and parameters.

Conflict Detection:

- `Conflict Identification:` The algorithm identifies conflicts between robots, such as collisions or overlapping paths.
- `Conflict Resolution:` Based on the identified conflicts, the algorithm creates constraints to resolve them, ensuring collision-free paths for each robot.

Path Planning

- `A* Search Integration*:` The code integrates the A* search algorithm to find optimal paths for individual robots while respecting the constraints.
- `Solution Computation:` It computes a solution that satisfies all constraints and minimizes conflicts among the robots.

Output

- `Success Flag:` The algorithm returns a Boolean flag indicating whether a collision-free path was found.
- `Path List:` If successful, the algorithm returns a list of waypoints representing the path for each robot.

Steps:

1. **Particle Clustering:** Clusters particles based on proximity and assigns them to robots.
2. **Path Planning with CL-CBS:** Generates collision-free paths for robots considering constraints and conflicts.

2.4.5 Advantages and Disadvantages:

Advantages

1. **Task Allocation:** Efficiently groups particles into clusters, allowing for effective task allocation among robots.
2. **Conflict Resolution:** Uses the CL-CBS algorithm for conflict resolution, ensuring collision-free paths for multiple robots.

Disadvantages

1. **Computational Complexity:** Clustering and path planning algorithms can be computationally intensive, especially with a large number of particles and robots.
2. **Parameter Sensitivity:** Performance may be sensitive to parameters such as cluster size and constraint definitions.

2.4.6 Summary:

The Clustering + CL-CBS method synergizes particle clustering and the CL-CBS algorithm for optimized multi-robot path planning. The `particle_clustering.py` file clusters particles, the `cl_cbs_algorithm.cpp` file implements the CL-CBS algorithm for path planning, and the `robot_movement.py` file handles the robot movement execution. Together, these components enable efficient coordination and movement of multiple robots in complex environments.

2.5 Method 4- Clustering and Collision Avoidance: Agglomerative Hierarchical Clustering

In the previous approach, we had used the clustering method in combination with CL-CBS. The path-finding method used in this method is not a traditional path-finding algorithm like A*, Dijkstra's, or RRT (Rapidly-exploring Random Tree). Instead, it uses a heuristic-based approach combined with clustering and optimization techniques, based on Agglomerative clustering.

Algorithm:

Here's a breakdown of the path-finding method used-

1. **Clustering:** The code initially clusters the data points representing potential waypoints for the robots using Agglomerative Clustering. This step groups nearby waypoints into clusters, simplifying the path-finding process.
2. **Heuristic Path Planning:** After clustering, each cluster's center of mass is calculated, and robots are assigned to move between these centers of mass. This step simplifies the path planning by reducing the number of waypoints to navigate through.
3. **Path Optimization:** Once the initial paths are assigned, the code optimizes them using a heuristic-based approach. It iteratively explores different configurations of paths by swapping waypoints within the paths while evaluating the total distance traveled. This optimization aims to minimize the total distance traveled by the robots.
4. **Final Approach:** After the main paths are optimized, a final approach to the goal is handled, ensuring that robots reach their final destinations accurately.

Overall, the approach combines clustering for waypoint simplification, heuristic-based path planning, and optimization techniques to find efficient paths for the robots. This method is tailored to the specific problem domain and constraints of the application, such as collision avoidance and path length optimization.

2.5.1 Code implementation:

This code is built on the code used for the previous approach. Additional functions for path planning has been implemented in this version.

clustering:

```
def dist_calc(o,di):
    td=0
    for i in range(len(o)-1):
        td+=di[o[i]][o[i+1]]
    return td

figure,axis=plt.subplots(1,2)
robots={}
map_size=20.0
X = np.unique(np.random.randint(1,map_size, size=(30,2)),axis=0)
n=4
start=[np.array([0.0,0.0]),np.array([map_size,map_size]),np.array([0.0,map_size]),np.array([map_size,0.0]),np.array
clustering = AgglomerativeClustering(n_clusters=n).fit(X)
labels = clustering.labels_
d={}
for i in range(n):
    d[i]=[]



# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)

# Plot result
```

```
# Black removed and is used for noise instead.
unique_labels = set(labels)

colors = ['y', 'b', 'g', 'r','m']

for i in range(len(X)):
    d[labels[i]].append(X[i])
    col=colors[labels[i]]
    axis[0].plot(X[i][0], X[i][1], 'o', markerfacecolor='k',
         markeredgecolor='k',
         markersize=10)
    axis[1].plot(X[i][0], X[i][1], 'o', markerfacecolor=col,
         markeredgecolor='k',
         markersize=10)


com=[]
for i in range(len(d)):
    c=(sum(d[i])/len(d[i]))
    com.append(c)

overall_com=sum(com)/len(com)
axis[1].plot(overall_com[0],overall_com[1],'o',markerfacecolor='k',markersize=15,label='Final Goal')

robots['agents']=[]
incomplete=list(range(n))
for m in range(n):
    min_sd=10*sqrt(2)
    start_position=start[m]
    end_position=com[incomplete[0]]
    k=incomplete[0]
    for p in incomplete:
       sd=np.linalg.norm(start_position-com[p])
       if sd<min_sd:
           min_sd=sd
           end_position=com[p]
           k=p

    incomplete.remove(k)
    points=d[k]
    points.insert(0,start_position)
    points.append(end_position)
    dist_list=[]
    points_order=[]

    for i in range(len(points)):
        dist=[]
        for j in range(len(points)):
            if j<i:
                dist.append(dist_list[j][i])
            else:
                dist.append(np.linalg.norm(points[j]-points[i]))
        dist_list.append(dist)

    pseudo_order=list(range(len(points)))
    final_order=deepcopy(pseudo_order)
    min_dist=dist_calc(pseudo_order,dist_list)
    for i in range(1,len(points)-2):
        for j in range(i+1,len(points)-1):
            pseudo_order[i],pseudo_order[j]=pseudo_order[j],pseudo_order[i]
            total_dist=dist_calc(pseudo_order,dist_list)
            if total_dist<min_dist:
                min_dist=total_dist
                final_order=deepcopy(pseudo_order)
            pseudo_order[i],pseudo_order[j]=pseudo_order[j],pseudo_order[i]
```

```
    for f in final_order[1:-1]:
        points_order.append(points[f].tolist())
```

Path Planning:

```
def calc_yaw(direc):
    if direc[0]==0:
        return direc[1]*1.57
    elif direc[1]==0:
        if direc[0]>0:
            return 0.0
        else:
            return 3.14
    elif direc[0]<0:
        if direc[1]<0:
            return atan(direc[1]/direc[0])-3.14
        else:
            return atan(direc[1]/direc[0])+3.14
    else:
        return atan(direc[1]/direc[0])

with open('output.yaml','r') as f:
    try:
        param = yaml.load(f, Loader=yaml.FullLoader)
    except yaml.YAMLError as exc:
        print(exc)

v=0.5
time_interval=1
n_robots=len(param['agents'])

start=[]
com=[]
objects=[]
for i in param['agents']:
    start.append(np.array(i['start'][0:2]))
    com.append(np.array(i['goal']))
    object_arrays=[]
    for j in i['objects']:
        object_arrays.append(np.array([float(j[0]),float(j[1])]))
    object_arrays.append(np.array(i['goal']))
    objects.append(object_arrays)
final_goal=sum(com)/len(com)

goal_reached=[0]*n_robots
output={}
object_remove={}
output['schedule']=dict()
time_list=[]
t=0
current_position=copy.deepcopy(start)
dimension=param['map']['dimensions']

for i in range(n_robots):
    time_list.append([])
    init_yaw=atan(((dimension[1]/2)-current_position[i][1])/((dimension[0]/2)-current_position[i][0]))
    time_list[i].append({'t':t,'x':current_position[i].tolist()[0],'y':current_position[i].tolist()[1],'yaw':init_y
    object_remove['agent'+str(i)]={}


object_no=[-1]*4
while goal_reached!=[1]*n_robots:
```

```
    t+=time_interval
    calc_position=[]
    yaw_list=[]
    object_change=[]
    for i in range(n_robots):

        position=current_position[i]
        if goal_reached[i]==1:
            calc_position.append(position)
            yaw_list.append(time_list[i][-1]['yaw'])
            object_change.append(0)
            continue

        current_object=objects[i][object_no[i]]

        if t==time_interval or np.linalg.norm(current_object-position)<0.01:
            if object_no[i]!=len(objects[i])-1:
                new_object=objects[i][object_no[i]+1]
            else:
                new_object=final_goal

            direction=(new_object-position)/np.linalg.norm(new_object-position)
            calc_position.append(position)
            object_change.append(1)

            yaw_list.append(calc_yaw(direction.tolist()))

            continue

        if np.linalg.norm(current_object-position)<v*time_interval:
            calc_position.append(current_object)
            yaw_list.append(time_list[i][-1]['yaw'])
            #time_list.append({'t':t,'x':current_position[0],'y':current_position[1]})
            object_change.append(0)
            continue

        direction=(current_object-position)/np.linalg.norm(current_object-position)
        yaw_list.append(calc_yaw(direction.tolist()))
        calc_position.append(position+v*direction*time_interval)
        object_change.append(0)
        #time_list.append({'t':t,'x':current_position[0],'y':current_position[1]})
        '''if np.linalg.norm(current_object-current_position)<0.01:
            object_change.append(1)
        else:
            object_change.append(0)'''

    for i in range(n_robots):
        s='agent'+str(i)
        current_position[i]=calc_position[i]
        time_list[i].append({'t':t,'x':current_position[i].tolist()[0],'y':current_position[i].tolist()[1],'yaw':yaw
        if object_change[i]:
            if -1<object_no[i]<(len(objects[i])-1):
                object_remove[s][t]=objects[i][object_no[i]].tolist()
            object_no[i]+=1
            if object_no[i]==len(objects[i]):
                goal_reached[i]=1
else:
    final_goal_reached=[0]*n_robots
    while final_goal_reached!=[1]*n_robots:
        t+=time_interval
        for i in range(n_robots):
            position=current_position[i]
            if final_goal_reached[i]==1:
                continue

            direction=(final_goal-position)/np.linalg.norm(final_goal-position)
```

```
            yaw=calc_yaw(direction.tolist())
            if np.linalg.norm(final_goal-position)<3*v*time_interval:
                current_position[i]=final_goal-2*v*time_interval*direction
                time_list[i].append({'t':t,'x':current_position[i].tolist()[0],'y':current_position[i].tolist()[1],'
                final_goal_reached[i]=1
                continue

            current_position[i]+=v*direction*time_interval
            time_list[i].append({'t':t,'x':current_position[i].tolist()[0],'y':current_position[i].tolist()[1],'yaw'
```

2.5.2 Visualization

Here, we have used a similar visualization technique as the previous method, albeit with a few small differences.

Input:

- Map details: Information about the map including boundaries and obstacles.
- Robot schedules: Schedules dictating the movement of robots over time.
- Object removal times: Times at which objects need to be removed from the environment.

Output:

- Animated visualization: An animation illustrating the movements of robots and the removal of objects over time.

Visualization Steps:

1. **Map and Agents Initialization:**
   - Set map boundaries: Define the boundaries of the map.
   - Obstacles: Determine the positions of obstacles within the map.
   - Agent positions: Initialize the starting positions of agents (robots) within the map.
2. **Animation Setup:**
   - Utilize `matplotlib.animation.FuncAnimation`: Set up the animation framework using the `FuncAnimation` class from Matplotlib.
3. **Animation Functions:**
   - Initialization function (`init_func`): Prepare the initial state of the animation by adding patches for map boundaries, obstacles, and agent positions.
   - Animation function (`animate_func`): Update the animation for each frame. This involves updating the positions of agents based on their schedules, handling object removals, and checking for collisions between agents.
4. **Collision Detection:**
   - Highlight collisions between robots: Implement logic to detect collisions between robots during the animation. If a collision occurs, highlight the colliding agents to provide visual feedback.

```
import yaml
import matplotlib
# matplotlib.use("Agg")
from matplotlib.patches import Circle, Rectangle, Arrow
from matplotlib.collections import PatchCollection
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import animation
import matplotlib.animation as manimation
import argparse
import math

Colors = ['yellow', 'blue', 'green', 'red','yellow']
```

```
size=0.5
framespermove=3

class Animation:
  def __init__(self, map, schedule, object_remove):
    self.map = map
    self.schedule = schedule
    self.object_remove=object_remove
    self.combined_schedule = {}
    self.combined_schedule.update(self.schedule["schedule"])

    aspect = map["map"]["dimensions"][0] / map["map"]["dimensions"][1]

    self.fig = plt.figure(frameon=False, figsize=(4 * aspect, 4))
    self.ax = self.fig.add_subplot(111, aspect='equal')
    self.fig.subplots_adjust(left=0,right=1,bottom=0,top=1, wspace=None, hspace=None)
    # self.ax.set_frame_on(False)

    self.patches = []
    self.artists = []
    self.object_list=[]

    self.objects={}
    self.object_names={}
    self.object_remove_dict=self.object_remove['object']
    self.agent_object={}
    self.agents = dict()
    self.agent_names = dict()
    # create boundary patch
    xmin = -0.5
    ymin = -0.5
    xmax = map["map"]["dimensions"][0] + 0.5
    ymax = map["map"]["dimensions"][1] + 0.5

    # self.ax.relim()
    plt.xlim(xmin, xmax)
    plt.ylim(ymin, ymax)
    # self.ax.set_xticks([])
    # self.ax.set_yticks([])
    # plt.axis('off')
    # self.ax.axis('tight')
    # self.ax.axis('off')

    self.patches.append(Rectangle((xmin, ymin), xmax - xmin, ymax - ymin, facecolor='none',
        edgecolor='red'))
    '''for o in map["map"]["obstacles"]:
      x, y = o[0], o[1]
      self.patches.append(Rectangle((x - 0.5, y - 0.5), 1, 1, facecolor='red', edgecolor='red'))'''

    # create agents:
    self.T = 0
    # draw goals first
    '''for d, i in zip(map["agents"], range(0, len(map["agents"]))):
      self.patches.append(Rectangle((d["goal"][0] - 0.25, d["goal"][1] - 0.25), 0.5, 0.5,
          facecolor=Colors[0], edgecolor='black', alpha=0.5))'''
    self.c=0
    for d, i in zip(map["agents"], range(0, len(map["agents"]))):
      for j in d['objects']:
        self.objects[str(self.c)]=(Circle((j[0], j[1]), 0.1, facecolor=Colors[i], edgecolor='black'))
        self.patches.append(self.objects[str(self.c)])
        self.object_names[str(self.c)]=self.ax.text(j[0],j[1], '')
        self.artists.append(self.object_names[str(self.c)])
        self.object_list.append(j)
        self.c+=1

      name = d["name"]
```

```
    dist = (size/2)*math.sqrt(2)
    angle = d['start'][2]+math.atan(1)
    self.agents[name] = Rectangle((d["start"][0]-(dist*math.cos(angle)),
        d["start"][1]-(dist*math.sin(angle))),0.5,0.5,d['start'][2],facecolor='orange',
        edgecolor='black')
    self.agents[name].original_face_color = 'orange'
    self.patches.append(self.agents[name])
    self.T = max(self.T, schedule["schedule"][name][-1]["t"])
    self.agent_names[name] = self.ax.text(d["start"][0], d["start"][1], '')
    self.agent_names[name].set_horizontalalignment('center')
    self.agent_names[name].set_verticalalignment('center')
    self.artists.append(self.agent_names[name])

  # self.ax.set_axis_off()
  # self.fig.axes[0].set_visible(False)
  # self.fig.axes.get_yaxis().set_visible(False)

  # self.fig.tight_layout()

  self.anim = animation.FuncAnimation(self.fig, self.animate_func,
                          init_func=self.init_func,
                          frames=int(self.T+1)*framespermove,
                          interval=50,
                          repeat=False,
                          blit=True)

def save(self, file_name, speed):
  self.anim.save(
    file_name,
    "ffmpeg",
    fps=10 * speed,
    dpi=200),
    # savefig_kwargs={"pad_inches": 0, "bbox_inches": "tight"})

def show(self):
  plt.show()

def init_func(self):
  for p in self.patches:
    self.ax.add_patch(p)
  for a in self.artists:
    self.ax.add_artist(a)
  return self.patches + self.artists

def animate_func(self, i):
  '''if i in self.object_remove_dict:
    for ob in self.object_remove_dict[i]:
      c=self.object_list.index(ob)
      self.objects[str(c)].center=(-1,-1)
      self.object_names[str(c)].set_position((-1,-1))'''
  for agent_name, agent in self.combined_schedule.items():
    j=i/framespermove
    pos = self.getState(i/framespermove, agent)
    d = (size/2)*math.sqrt(2)
    angle = pos[2]+math.atan(1)
    p = [pos[0]-(d*math.cos(angle)), pos[1]-(d*math.sin(angle))]
    self.agents[agent_name].set_xy(p)
    self.agents[agent_name].angle = pos[2] / math.pi * 180
    self.agent_names[agent_name].set_position(p)

    if agent_name not in self.agent_object:
      self.agent_object[agent_name]=[]

    if j in self.object_remove_dict[agent_name]:
      c=self.object_list.index(self.object_remove_dict[agent_name][j])
      self.agent_object[agent_name].append(str(c))
```

```
      for ob in self.agent_object[agent_name]:
        self.objects[ob].center=(pos[0]+((size/2)*math.cos(pos[2])),pos[1]+((size/2)*math.sin(pos[2])))
        self.object_names[ob].set_position(p)



    # reset all colors
    for _,agent in self.agents.items():
      agent.set_facecolor(agent.original_face_color)

    # check drive-drive collisions
    '''agents_array = [agent for _,agent in self.agents.items()]
    for i in range(0, len(agents_array)):
      for j in range(i+1, len(agents_array)):
        d1 = agents_array[i]
        d2 = agents_array[j]
        pos1 = np.array(d1.center)
        pos2 = np.array(d2.center)
        if np.linalg.norm(pos1 - pos2) < 0.5:
          d1.set_facecolor('red')
          d2.set_facecolor('red')
          print("COLLISION! (agent-agent) ({}, {})".format(i, j))'''

    return self.patches + self.artists


  def getState(self, t, d):
    idx = 0
    while idx < len(d) and d[idx]["t"] < t:
      idx += 1
    if idx == 0:
      return np.array([float(d[0]["x"]), float(d[0]["y"]), float(d[0]["yaw"])])
    elif idx < len(d):
      yawLast = float(d[idx-1]["yaw"])
      yawNext = float(d[idx]["yaw"])
      if (yawLast - yawNext) > math.pi:
        yawLast = yawLast - 2 * math.pi
      elif (yawNext - yawLast) > math.pi:
        yawLast = yawLast + 2 * math.pi
      posLast = np.array([float(d[idx-1]["x"]), float(d[idx-1]["y"]), yawLast])
      posNext = np.array([float(d[idx]["x"]), float(d[idx]["y"]), yawNext])
    else:
      return np.array([float(d[-1]["x"]), float(d[-1]["y"]), float(d[-1]["yaw"])])
    dt = d[idx]["t"] - d[idx-1]["t"]
    t = (t - d[idx-1]["t"]) / dt
    pos = (posNext - posLast) * t + posLast
    return pos



if __name__ == "__main__":
  '''
  parser = argparse.ArgumentParser()
  parser.add_argument("map", help="input file containing map")
  parser.add_argument("schedule", help="schedule for agents")
  parser.add_argument('--video', dest='video', default=None, help="output video file (or leave empty
      to show on screen)")
  parser.add_argument("--speed", type=int, default=1, help="speedup-factor")
  args = parser.parse_args()
  '''
  with open('output.yaml','r') as map_file:
    map = yaml.load(map_file, Loader=yaml.FullLoader)

  with open('schedule.yaml') as states_file:
    schedule = yaml.load(states_file, Loader=yaml.FullLoader)
```

```
with open('object.yaml') as object_file:
  object_remove = yaml.load(object_file, Loader=yaml.FullLoader)

animation = Animation(map, schedule, object_remove)


animation.show()
```

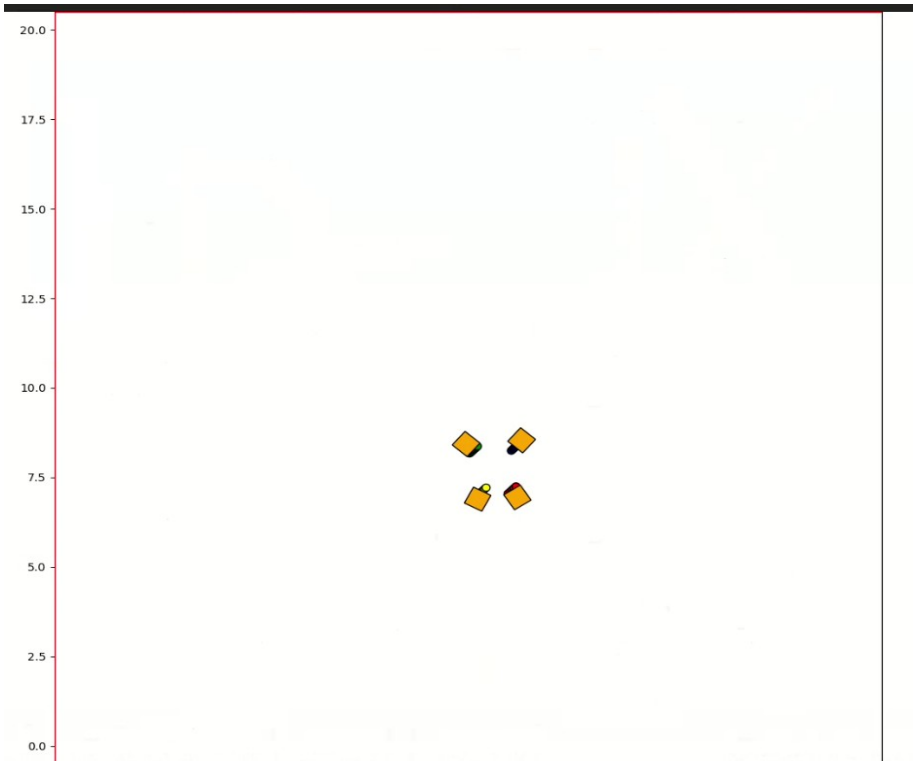

Figure 2.9: Final Result

2.5.3 Summary

- **Particle Clustering:** Groups particles into clusters, assigning them to robots.
- **Path Planning:** Generates collision-free paths for robots.
- **Visualization:** Offers an animated representation of robot movements and object removals.

Method 4 integrates Particle Clustering, Path Planning, and Visualization to efficiently allocate tasks, compute collision-free paths, and provide an intuitive animated visualization for analysis.

Chapter 3

Conclusion

In conclusion, our project has evaluated four distinct methods for multi-robot path planning and coordination. Among these methods, Method 1 - Overhead Camera-Based Multi-Robot Path Planning and Method 4 - Clustering and Collision Avoidance: Agglomerative Hierarchical Clustering have demonstrated optimal performance. Method 1 excels in environments requiring motion along coordinate directions, although it has limitations in supporting diagonal motion. On the other hand, Method 4 offers robustness and flexibility, supporting various types of motion, making it an ideal choice for hardware implementation, such as in a two-wheel differential drive robot.

However, Methods 2 and 3 exhibited certain limitations due to issues in their code integration, leading to potential errors and reduced reliability. These methods require further refinement to enhance their robustness and effectiveness in real-world applications.

In future work, we aim to focus on enhancing the robustness of the less-performing methods and exploring the integration of machine learning techniques to further improve the adaptability and efficiency of the selected methods in dynamic environments.

Chapter 4

References

- **Github repository on Multi-Agent path planning in Python:** https://github.com/atb033/multi_agent_path_planning
- **Github repository on Car-like Conflict-Based Search (CL-CBS):** https://github.com/APRIL-ZJU/CL-CBS
- **Github repository on Ultrasonic-SLAM:** https://github.com/PatelVatsalB21/Ultrasonic-SLAM
- **Conflict-Based Search For Optimal Multi-Agent Path Finding:** https://people.engr.tamu.edu/guni/Papers/CBS-AAAI12.pdf
- **SLAM Research summarization paper:** https://people.eecs.berkeley.edu/~pabbeel/cs287-fa09/readings/Durrant-Whyte_Bailey_SLAM-tutorial-I.pdf