COMP0182 REAL-WORLD MULTI-AGENT SYSTEMS FINAL REPORT

Yifan Cai*, 24202204, University College London

Abstract

This report focuses on developing a multi-agent navigation system using TurtleBot3 robots, integrating advanced algorithms like A*, CBS, RRT*, and NMPC. ArUco markers enable precise localization, while PID control ensures stable motion. A hybrid approach with A* and RRT* optimizes paths, and CBS resolves conflicts for efficient, collision-free navigation. Despite challenges with NMPC computation, the results highlight the potential of combining algorithms for robust multi-agent autonomy.

INTRODUCTION

Autonomous navigation is a pivotal aspect of modern robotics, enabling robots to navigate complex and dynamic environments while avoiding obstacles and achieving predefined goals. This project centers on designing and testing a TurtleBot3-based navigation system capable of autonomously traversing from a start point to a target endpoint while efficiently avoiding obstacles. By utilizing ArUco markers for parameter setup and TurtleBot3 detection, the system enhances spatial precision and aligns simulated environments with real-world coordinates, integrating core robotics concepts like path planning, obstacle avoidance, and real-time decision-making.

The primary aim is to program multiple TurtleBot3 robots to navigate simultaneously from defined starting points to endpoints using YAML-stored simulation coordinates. This project demonstrates the adaptability of autonomous robotic systems in real-world-like environments, offering insights into their potential in logistics, exploration, and collaborative tasks.

This report details the initial tasks that laid the groundwork for the final challenge, explores the algorithms implemented, and presents simulation and experiment results, highlighting the effectiveness and challenges of the solutions.

Literature Review

Effectiveness of pid Controllers in Autonomous Navigation

The Proportional-Integral-Derivative (PID) controller is widely acknowledged as a reliable and adaptable tool for controlling and stabilizing the state variables of various systems[1]. Its fundamental design enables accurate adjustments in dynamic conditions, making it an ideal choice for autonomous navigation systems. The straightforward structure, combined with its strong control performance and ease of deployment, establishes the PID controller as a powerful solution for addressing the intricacies of autonomous robotic systems[2].

In trajectory tracking, PID controllers have proven their effectiveness through their capability to closely approximate target trajectories with exceptional precision[3]. For instance, in the trajectory tracking of a mobile robot platform, the integration of PID control with a dynamic model has been validated through both simulations and experiments, achieving accurate trajectory replication by the robot. These findings underline the controller's ability to adapt dynamically to the robot's motion, ensuring smooth and precise navigation.

PID-based controllers also excel in regulating velocity and improving force control for mobile robots. In demanding situations, such as navigating uphill or downhill, PID controllers enable smooth and stable transitions, minimizing abrupt changes and ensuring effective thrust force management[4]. This ability to handle varying terrains and dynamic conditions underscores the robustness of PID control in autonomous systems.

Additionally, in the realm of autonomous vehicles, PID-based approaches have become integral for steering and path control[2]. Their straightforward implementation and reliability make them particularly well-suited for self-driving cars, where maintaining precise and stable steering is critical[5]. The widespread use of PID controllers in autonomous vehicle steering further emphasizes their dependability and flexibility in overcoming real-world challenges[5].

In conclusion, the PID controller remains a cornerstone of autonomous navigation, providing a straightforward yet highly effective approach for managing dynamic systems. Its role in trajectory tracking, velocity regulation, and path planning has been validated across diverse applications, solidifying its importance in modern robotic systems. Future advancements could focus on hybrid methodologies, integrating PID control with advanced optimization or machine learning techniques, to further enhance its performance in complex and dynamic environments.

Comparative Analysis of A* And RRT* In Robotic Path Planning

A* (A-Star) and RRT* (Rapidly-Exploring Random Tree Star) are two widely used algorithms in robotic path planning, each excelling in different contexts. While A* is a deterministic graph-based search algorithm optimized for finding the shortest path, RRT* is a probabilistic sampling-based algorithm that focuses on efficiently exploring high-dimensional and dynamic spaces while refining paths toward optimality.

A* is celebrated for its accuracy and guarantees of optimality in pathfinding. It works by combining the actual cost from the start node to the current node (g(n)) and a heuristic

* y.cai.24@ucl.ac.uk

estimate of the cost from the current node to the goal (h(n)). The algorithm prioritizes nodes with the lowest total cost ($f(n) = g(n) + h(n)$), ensuring a systematic and efficient search. A* performs exceptionally well in structured, grid-like environments with clearly defined obstacles, as its heuristic guides the search toward the most promising areas, reducing unnecessary computation [6]. However, the algorithm struggles in large or high-dimensional environments due to its exhaustive exploration, leading to significant memory and computational demands.

In contrast, RRT* addresses the limitations of A* by employing a sampling-based approach, which is particularly suited for high-dimensional spaces. The algorithm generates a tree by incrementally adding random samples from the configuration space, connecting them to the nearest existing node while considering constraints such as obstacles [7]. Unlike its predecessor RRT, RRT* incorporates an optimization step that rewires the tree to improve path quality iteratively. This allows RRT* to find asymptotically optimal paths over time, balancing exploration and exploitation. Its probabilistic nature makes it highly adaptable to dynamic and unstructured environments, where deterministic methods like A* may struggle. However, RRT* requires more computation compared to RRT and may produce less direct paths when the sample density is low.

The integration of A* and RRT* into hybrid planning systems leverages their strengths to overcome individual limitations. For instance, A* can provide an initial structured path in constrained areas, while RRT* can refine and extend the path in larger, more dynamic spaces. Such hybrid approaches have been demonstrated to improve the efficiency and feasibility of path planning in real-world applications, including multi-robot systems, autonomous vehicles, and complex robotic arms [8], [9]. These hybrid strategies are particularly useful in environments where the deterministic precision of A* is required for confined spaces, while the flexibility of RRT* can handle open and unpredictable regions.

To conclude, while A* is an optimal and deterministic algorithm suited for structured environments, RRT* excels in high-dimensional, unstructured, and dynamic contexts. Both algorithms play pivotal roles in robotic path planning, with their hybrid usage paving the way for innovative solutions to navigate complex environments effectively. Future work may explore more efficient ways to combine these methods, leveraging the predictability of A* and the adaptability of RRT* to handle increasingly sophisticated robotic challenges.

Individual Contribution

Throughout the project, I played a pivotal role in ensuring the successful completion of each phase. My responsibilities encompassed the weekly setup processes, coding tasks for the final challenge, and developing innovative approaches that enhanced the overall functionality of our system.

1. **Weekly Setup**:

   I meticulously handled the configuration of required software and tools, ensuring a seamless environment for the project. This included the installation and setup of essential platforms such as ROS Noetic, PyBullet, and the integration of the TurtleBot3. I also resolved various compatibility issues and ensured that all team members had access to a well-prepared development environment. In addition, I facilitated the setup of camera calibration using ArUco markers and ensured accurate mapping of simulation and real-world coordinates.

2. **Final Task Implementation**:

   I was solely responsible for implementing the full code for the final task. This involved integrating multiple algorithms, such as CBS with A* and CBS with RRT*, to ensure efficient path planning and conflict resolution for multiple robots. The code was designed to be robust and adaptable, accommodating various edge cases and ensuring smooth transitions between different task stages. I also worked on the integration of threading to enable simultaneous navigation of multiple robots, improving the efficiency and scalability of the system.

3. **Innovative Contributions**:

   In the innovative section, I introduced a hybrid path-planning strategy that combined the strengths of different algorithms for optimal results. For instance, the A* algorithm was utilized for precise navigation in structured environments, while the RRT* algorithm was applied to explore and optimize paths in dynamic or less predictable scenarios.

   I also incorporated the Nonlinear Model Predictive Control (NMPC) framework into the project, particularly for optimizing multi-agent path planning. This involved designing cost functions that balanced trajectory tracking and collision avoidance while respecting dynamic constraints. Although NMPC posed computational challenges when increasing the prediction horizon, I conducted extensive tests and analysis to identify potential improvements for future implementation. This exploration of NMPC demonstrated its potential for real-time control in complex multi-robot systems, laying the groundwork for further refinements.

   Finally, I developed a visualization framework that allowed us to effectively compare and evaluate the performance of different algorithms. This framework played a crucial role in understanding the strengths and weaknesses of various approaches, leading to informed decisions for the final implementation.

4. **Problem-Solving and Troubleshooting**:

   During the project, I addressed critical issues such as collisions between multiple robots by experimenting with various strategies, including adjusting speeds and implementing timed delays. These solutions ensured reliable and collision-free navigation, especially in complex multi-robot scenarios.

Overall, my contributions were instrumental in the successful completion of the project, from initial setup to the final

execution. My focus on innovation, meticulous coding, and problem-solving ensured that the system was not only functional but also efficient and scalable for future enhancements.

FUNDAMENTAL TASKING

The project is divided into numerous phases:

1. **Software Configuration**

   The first phase involves the download of numerous software and packages like Ubuntu, Visual Studio Code, ROS Noetic, PyBullet and etc. Additionally, the essential dependencies for ROS are installed as well. The script is subsequently added into `~\.bashrc` for greater convenience.

2. **TurtleBot3 Configuration**

   The TurtleBot3 requires numerous configurations like identifying the IP address of the robot and setting up a development environment. To identify the IP address of the robot, it involves connecting the TurtleBot3's Raspberry Pi to a monitor and keyboard via an HDMI cable. To set up a ROS Noetic Development Environment, we are required to create a folder called `catkin_ws` and git clone the files provided into that folder.

3. **Camera Calibration**

   The Logitech C920 HD Pro camera was connected to our laptop to perform the initialization and calibration process. To calibrate the camera, a checkerboard was used to initialize the camera settings. The process is shown in Fig. 1. To provide a better calibration, the checkerboard needed to be tilted in different directions during the scan.

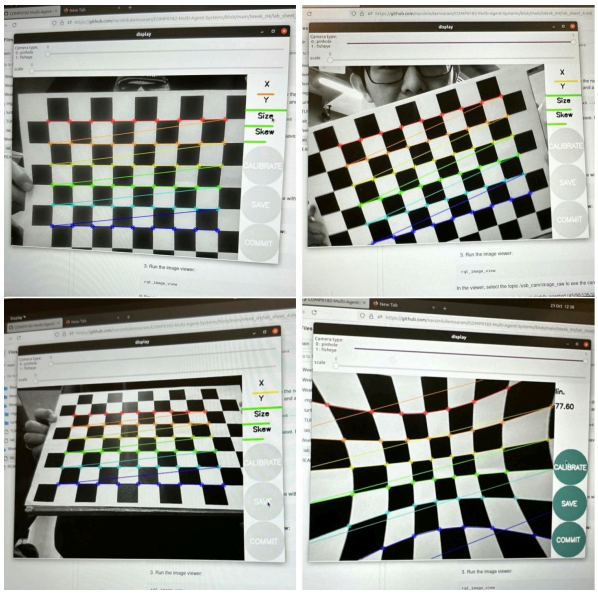

Figure 1: Camera Calibration Process using Checkerboard

4. **Single and Multiple ArUco Markers Initialization**

   This phase utilizes the Logitec camera to detect single and multiple ArUco Markers and program the TurtleBot3 to move towards the ArUco Marker. The position of the TurtleBot3 and the desired goal are represented by the ArUco Marker with two separate identification numbers. We used id100 to represent the position of the TurtleBot3 and id101 to express the position of the goal. We first tested with a single ArUco marker before moving on to multiple ArUco markers.Fig. 2 illustrates the process.

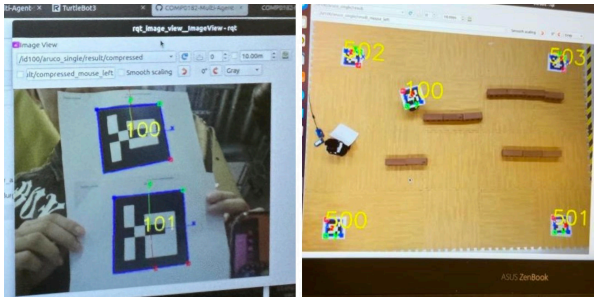

Figure 2: Single & Multiple ArUco Marker Initialization

5. **Mapping and Navigation of Given Environment**

   This phase involves utilizing the LiDAR sensor on the TurtleBot3 to map the given environment. Obstacles have been strategically placed within the environment to define the intended path for the robot. Once the map is generated, the TurtleBot3 can autonomously navigate through the environment by simply designating the desired end goal. Fig. 3 illustrates the map scanned by the TurtleBot3 as well as the position of the desired end goal.

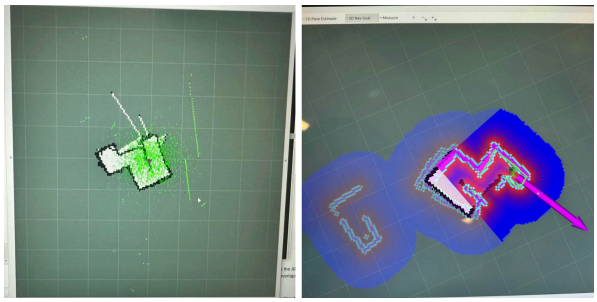

Figure 3: Mapping and Navigation of TurtleBot3 in Given Environment

6. **Single and Multiple Robot(s) Auto Navigation**

   This phase involves the simulation and execution of single and multiple robots where the navigation framework is validated and refined. A single TurtleBot3 is first tested before moving on to multiple agents where the challenge of coordinating them is introduced. Both tests emphasize seamless obstacle avoidance and efficient pathfinding, ensuring each TurtleBot reaches its endpoint without collisions or delays. The report will go on to elaborate more on the simulation process and experimentation process in Sectionref{ref:simulation} and Sectionref{ref:Experiment} respectively.

SIMULATION ALGORITHMS AND INNOVATION

To bridge the gap between theoretical models and real-world execution, a simulation environment is utilized. This simulation provides a controlled space to analyze

and determine the optimal coordinates and path for the TurtleBot3(s). These pre-determined coordinates form the basis for executing the navigation algorithm on actual TurtleBot3s, ensuring a smooth transition from simulation to physical deployment. Through this iterative approach, the project aims to develop a robust and scalable solution for autonomous robotic navigation.

Algorithm for Resolving Conflicts

Conflict-Based Search (CBS) Algorithm

This method is mainly used for multi robot path planning problems. This algorithm aims to find effective paths for multiple robots by resolving conflicts between them, so that each robot can successfully reach its target in a given environment without collision. Specifically, CBS mainly addresses two types of conflicts. One is vertex conflict, where two robots (referred to as multiple robots in this experiment) may occupy the same position at a certain time; The second type is edge conflict, where two robots move along intersecting paths at adjacent time steps, hindering each other. In this experiment, we used this method to avoid conflicts between different robots along predetermined paths.

Algorithm for Low-level Path-Finding

A-Star (A*) Algorithm

The A* algorithm is a widely used graphic search and path planning algorithm, particularly suitable for finding the shortest path from the starting point to the target. It combines the advantages of greedy algorithm and Dijkstra algorithm, which can quickly find solutions while ensuring the optimality of the path.

Its basic principle is to calculate the total cost (cost function) from the starting point to the target, and prioritize the path with the minimum total cost for expansion, gradually searching until the target is found. The total cost function of the A* algorithm is defined as:

$$f(n) = g(n) + h(n) \tag{1}$$

where $g(n)$ is the actual cost from the starting point to the current node, and $h(n)$ is the estimated cost from the current node n to the target node. It selects the node with the smallest $f(n)$ as the current node, removes the open list and adds it to the closed list, and calculates $g(m)$, $h(m)$, and $f(m)$ for each neighboring node. After multiple iterations, it finds the target node and backtracks through the target node to obtain the shortest path.

Algorithm 1 is the pseudocode for A*.

Algorithm 1: A* Algorithm

```
Initialize the open list and closed list
Add the start node to the open list
while the open list is not empty
    Select the node with the lowest f value from the open list (current node)
    Move the current node to the closed list
    if the current node is the goal node then
        Trace back from the goal node to the start node to reconstruct the path
        Return the path
    end
    Generate all valid neighbors of the current node
    for each neighbor node
        if the neighbor is in the closed list then
            Skip to the next neighbor
        end
        Calculate g , h , and f for the neighbor:
            g : Cost from the start node to the neighbor
            h : Heuristic estimate from the neighbor to the goal
            f : g + h
        if the neighbor is not in the open list then
            Add the neighbor to the open list
        else if the new g value is lower then
            Update the neighbor's g and f values
            Update the parent of the neighbor to the current node
        end
    end
end
Return no path found
```

Innovation Part

Rapidly exploring Random Tree (RRT) Algorithm

The RRT algorithm is a sampling-based path planning algorithm, whose core idea is to quickly construct a tree from the starting point that can cover the entire feasible space through random sampling, and continuously expand the tree until the target point is found.

Its basic working principle is to randomly sample a point (which may not be at the target location), and then use a tree (a point on the proposed route) to find the nearest node q_{nearst}, and then expand the tree along the direction from q_{nearest} to q_{rand}. The extension of a tree is usually done in units of a fixed stride Δq, generating a new node $q_{\text{new}}$and adding it to the tree.

$$q_{\text{new}} = q_{\text{nearst}} + \Delta q \cdot \frac{q_{\text{rand}} - q_{\text{nearst}}}{\| q_{\text{rand}} - q_{\text{nearst}} \|} \tag{2}$$

Then iterate multiple times until reaching the target point position. However, in practical operation, it was found that this method has strong randomness and may require a large amount of sampling to obtain an effective path. Moreover, the generated path is often not smooth and needs further optimization. Its path may contain unnecessary bends or uneven turns.

Algorithm 2: RRT Algorithm

```
Initialize the tree with the start node
for i = 1 to iterations:
    Generate a random sample point
    Find the nearest node in the tree
    Steer towards the random point within step size
    if the new node is collision-free:
        Add the new node to the tree
        if the new node is within step size of the goal:
            Return the path from start to the goal
        end
    end
end
Return failure (no path found)
```

Algorithm 2 is the pseudocode for RRT.

RRT* Algorithm

Unlike RRT, the RRT* algorithm takes into account the cost of neighboring nodes within a certain radius of the sampling point. By comparing whether there are better neighboring nodes to replace the original sampling point, the planned route becomes more optimal. It is worth mentioning that the RRT* algorithm introduces several core mechanisms: search radius, rewiring, and cost function. The search radius is defined by the range of neighboring nodes that can be considered around the path point. The cost function $c(x)$ represents the path cost from the starting point $x_{\text{start}}$to any node:

$$c(x) = c\big(x_{\text{parent}}\big) + \text{distance}\left(x_{\text{parent}}, x\right) \tag{3}$$

Among them, x_{parent} represents the parent node of the node.

In rewiring operations, the path structure is optimized by evaluating neighboring nodes with lower costs. For any neighbor node $x_{\text{neighbor}} \in \mathcal{N}(x_{\text{new}})$, if it satisfies

$$\begin{aligned} c(x_{\text{new}}) + \text{distance}\big(x_{\text{new}}, x_{\text{neighbor}}\big) \\ < c\big(x_{\text{neighbor}}\big) \end{aligned} \tag{4}$$

Then update the parent node of x_{neighbor} to x_{new}.

Algorithm 3 is the pseudocode for RRT*.

Nonlinear Model Predictive Control Algorithm

Nonlinear Model Predictive Control (NMPC) is an optimization control method widely used in path planning and control of dynamic systems. In order to move the system from the initial state to the target state, this method adopts a dynamic optimization approach to calculate the optimal control input, which can cope with complex scenarios with nonlinear systems and constraints. The basic steps of this method are to first predict the future state through a dynamic model, then optimize the future control inputs based on the current state, and gradually update the current state through continuous iteration.

Algorithm 3: RRT* Algorithm

```
Initialize the tree with the start node
Set the cost of the start node to 0
for i = 1 to iterations:
    Generate a random sample point
    Find the nearest node in the tree
    Steer towards the random point within step size
    if the new node is collision-free:
        Add the new node to the tree
        Rewire the tree for optimality within the given radius
        if the new node is within step size of the goal:
            Return the optimized path from start to the goal
        end
    end
end
Return failure (no path found)
```

1. System Dynamics Model

   NMPC predicts future states using the system's dynamic model. For a discrete-time system, the dynamics are given by:

   $$x_{k+1} = f(x_k, u_k) \tag{5}$$

   Where:
   - $x_k \in \mathbb{R}^n$: State vector at time k.
   - $u_k \in \mathbb{R}^m$: Control input vector at time k.
   - $f(\cdot)$: Nonlinear function representing the system's behavior.

2. Future State Prediction

   By iteratively applying the system dynamics, NMPC predicts the future states over a time horizon N:

   $$\begin{aligned} x_{k+1} &= f(x_k, u_k), \\ x_{k+2} &= f(x_{k+1}, u_{k+1}), ..., x_{k+N} \\ &= f(x_{k+N-1}, u_{k+N-1}) \end{aligned} \tag{6}$$

   Where N is the prediction horizon length.

3. Optimization Problem

   The core of NMPC is solving an optimization problem to minimize a cost function J. The cost function is typically defined as:

   $$J = \sum_{i=0}^{N-1} \left[\| x_i - x_{\text{ref}} \|_Q^2 + \| u_i \|_R^2 \right] \tag{7}$$

   Where:
   - x_{ref}: Desired reference state.
   - $\| x_i - x_{\text{ref}} \|_Q^2 = (x_i - x_{\text{ref}})^T Q (x_i - x_{\text{ref}})$: Weighted quadratic term penalizing state error.
   - $\| u_i \|_R^2$: Weighted quadratic term penalizing control effort.

- Q and R: Positive-definite weighting matrices for state error and control effort, respectively.

4. Constraints

   The optimization problem must respect the following constraints:

   1. System Dynamics Constraints:
   $$x_{k+1} = f(x_k, u_k), \quad \forall k \in [0, N-1] \tag{8}$$
   2. State Constraints (e.g., physical boundaries):
   $$x_k \in \mathcal{X}, \quad \forall k \in [0, N] \tag{9}$$
   3. Control Input Constraints (e.g., input limits):
   $$u_k \in \mathcal{U}, \quad \forall k \in [0, N-1] \tag{10}$$

5. Control Application

   The solution to the optimization problem is a sequence of control inputs $u_0^*, u_1^*, ..., u_{N-1}^*$. In practice, only the first control input u_0^* is applied to the system:
   $$u_{\text{apply}} = u_0^* \tag{11}$$
   After applying u_0^*, the system state is updated, and the optimization process repeats with the updated state as the new starting point. This process is called receding horizon control.

6. Full NMPC Optimization Problem

   Combining the above, the NMPC optimization problem can be expressed as:
   $$\min_{u_0,\ldots,u_{N-1}} \sum_{i=0}^{N-1} \left[\| x_i - x_{\text{ref}} \|_Q^2 + \| u_i \|_R^2 \right] \tag{12}$$
   Subject to:
   $$\begin{aligned} x_{k+1} &= f(x_k, u_k), \\ x_k \in \mathcal{X}, \quad u_k &\in \mathcal{U}, \quad \forall k \in [0, N-1] \end{aligned} \tag{13}$$

Algorithm 4 is the pseudocode for NMPC.

Algorithm 4: NMPC Algorithm

Function `NMPC(state, goal, obstacles, horizon)`
  Initialize system parameters, constraints, and cost function
  While (not at goal):
    Predict future states over the horizon using the dynamic model
    Formulate the optimization problem to minimize cost:
      Cost includes trajectory tracking and collision avoidance
    Solve optimization to obtain control input `u*`
    Apply the first control input `u_0*` to update the state
    Update state and repeat
Return optimized path

Nonlinear Model Predictive Control (NMPC) in multi-agent path planning is a real-time optimization-based approach that computes collision-free trajectories for multiple agents by predicting their future states over a fixed horizon, minimizing a combined cost of trajectory tracking and collision avoidance, while respecting dynamic, environmental, and interaction constraints to ensure cooperative and efficient navigation.

SIMULATION PROCESS

The simulation of this experiment was conducted on a 10×10 grid, which is a multi-robot simulation system based on PyBullet. Its process is to plan the initial and destination positions of multiple robots through underlying algorithms (A* algorithm is given by default), avoiding preset obstacles. The planned paths are then stored in YAML files as path points. However, there may be conflicts during the simultaneous movement of multiple robots, so the CBS algorithm is introduced to avoid possible conflicts and achieve the simultaneous movement of multiple robots and ultimately reach their respective target points. In order to simulate the changes during the motion process, PyBullet is introduced to visualize the motion process. It should be noted that there is no unique algorithm for the underlying design path. In this experiment, we compared other methods to observe their different motion performances.

It should be noted at the beginning that both the A* algorithm and our innovative RRT&RRT* algorithm are based on the optimal path planned under the given starting point, target point, and preset obstacle information. However, due to the introduction of additional fetch points in the task, these fetch points may not necessarily appear on the optimal path planned by the A* algorithm (i.e., the introduction of fetch points tends to be more biased towards paths that are farther away). In order to meet the requirements of reaching the fetch point and using the A* algorithm, we divide the entire navigation process into two parts: the first stage is to control the robot to navigate from the starting point to the given fetch point, and the second stage is to control the robot to move from the fetch point to the destination point. The two stages will determine the algorithm to be used separately, which can meet the requirements of the underlying pathfinding algorithm without affecting the task of reaching the fetch point.

An Algorithm Comparator Visualized through Two-dimensional Trajectories

Firstly, before starting the simulation of a single robot and multiple robots, we need to discuss the visualization comparison of the robot's movement trajectory during the defined observation simulation process. The reason for introducing this algorithm is that due to the small size of the environment and the relatively narrow effective path reserved for obstacles under a fixed placement strategy, the differences in paths generated by different algorithms are not significant. If we only observe the pseudo materialized environment built by PyBullet, it is difficult to clearly and intuitively distinguish the differences between them, which

will pose a cognitive barrier to the authenticity of the encoding. Therefore, we have designed a mapping method that generates trajectories by visually observing the lines connecting each path point in the YAML file. The general drawing logic is as Algorithm 5.

Algorithm 5: RRT Path Plotting Algorithm

Function
  Create a figure and axis
  if obstacles exist:
    Plot obstacles and boundary edges
  Create and display a grid
  if start or goal exists:
    Plot start (green) and goal (red)
  if path exists:
    Plot path as blue line with points
  if tree exists:
    Plot tree nodes as black dots
  Add labels, title, legend, save and display the plot

Single Robot Simulation Process

Refactoring the Navigation Method to Adapt to the Fetch Points

As mentioned earlier, we use the A* algorithm to navigate a single robot in both stages (i.e. from the starting point to the fetch point, and then from the fetch point to the target point). For this, we need to modify the `final_challenge.py` file, as it involves connecting two stages to ensure the smoothness of the robot throughout the entire movement process. We have modified the navigation method to set the current position as the initial point for the next stage after completing the first stage, completing the navigation of the true target point. The basic logic is as follows:

```
def navigation(agent, goal, schedule, goal2,
schedule2):
    """
    Set velocity for robots to follow the path
in the schedule.
    Args:
     agents: array containing the boxID for
each agent
    schedule: dictionary with boxID as key and
path to the goal as list for each robot.
     index: index of the current position in
the path.
    Returns:
    Leftwheel and rightwheel velocity.
  """
basePos
= p.getBasePositionAndOrientation(agent)
while(not checkPosWithBias(basePos[0], goal,
dis_th)):
# ... The navigation process in the original
navigation method
basePos
= p.getBasePositionAndOrientation(agent)
while(not checkPosWithBias(basePos[0], goal2,
dis_th)):
# ... The same procedure in the previous
navigation method
```

It should be noted that the schedule parameter in the above method expresses the path points stored in the YAML file generated by the algorithm in advance. Therefore, it is necessary to generate separate paths for the initial and target points (based on different stages) corresponding to the env file information used to generate YAML files in advance. Then, during the execution of the main function, the YAML files used to store the path points are extracted and passed as parameters to the modified navigation method. This effectively completes the calling and encapsulation of different Python files. The basic logic is as follows:

```
from cbs import cbs_a_star, cbs_rrt,
                              cbs_rrt_star

  # ... previous initialize procedure
          cbs_a_star.main("final_challenge/
env_2.yaml", "cbs_output_fetch_1.yaml",1).
  # we can change different algorithms
  schedule =
  read_output("cbs_output_fetch_1.yaml")
  _, goals2, env_loaded =
read_input("final_challenge/
env_2_stage.yaml",env_loaded)
          cbs_a_star.main("final_challenge/
env_2_stage.yaml",
  "cbs_output_fetch_2.yaml",2)
  # we can change different algorithms
                                  schedule2
= read_output("cbs_output_fetch_2.yaml")
    run(agents,  goals,  schedule,  goals2,
schedule2)
```

Addition of PID Control

PID controller is one of the most commonly used control algorithms in automatic control systems, consisting of three parts: proportional (P), integral (I), and derivative (D). It adjusts the control input to gradually approach the target value of the system output. The control signal output formula of PID controller is:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \quad (14)$$

Due to the widespread explanation of such formulas, the meanings of the parameters will not be elaborated here. It should be mentioned that in our actual operation, we did not strictly follow the PID formula to achieve smooth operation

adjustment, but used P+D control. There are three reasons for doing so:

- This is because the integral term (I) in PID control accumulates errors, requires additional memory and computational overhead, and may cause delays in the controller's response.
- The current task mainly focuses on the robot navigating along the path point to the target point, which is usually close, and each step of the path planning is predefined.
- In practical operation, we need to avoid collisions with obstacles as much as possible, and the instability of the car operation poses a challenge to the already narrow space. Therefore, the simplified P+D control directly handles errors by updating the target angle and proportion adjustment in real time, avoiding unnecessary oscillations caused by integration.

Our specific approach to implementing PID for this is:

- Proportional control (part p)

$$\nu_{\text{linear}} = K_1 \cdot \cos(\theta) \tag{15}$$

$$\nu_{\text{angular}} = K_2 \cdot \theta \tag{16}$$

Among them, K_1represents the proportional coefficient of linear velocity, K_2represents the proportional coefficient of angular velocity, and θ represents the angle difference between the current robot orientation and the direction of the target point.

- Differential Control (Part D):

$$\theta = \theta_{\text{goal}} - \theta_{\text{orientation}} \tag{17}$$

The adjustment of θ takes into account the orientation change of the robot's current position, thereby reducing significant oscillations.

Visualization

Fig. 4 compares path planning approaches with and without the inclusion of fetch points. The left image represents a direct path generated using the A* algorithm, where the robot moves from the start point (green dot) to the goal point (red dot) without any intermediate stops. This path is efficient and avoids obstacles (gray blocks) but does not account for tasks requiring intermediate fetch points.

In contrast, the middle and right images together form a complete path that includes a fetch point. The middle image illustrates the first stage, where the robot navigates from the start point to the fetch point using the A* algorithm. The right image shows the second stage, where the robot moves from the fetch point to the goal point, with CBS resolving conflicts and optimizing the trajectory for multi-agent scenarios.

By incorporating the fetch point, the path becomes longer but fulfills task-specific requirements, such as stopping at intermediate locations. This two-stage approach demonstrates the flexibility and adaptability of combining different algorithms like A* and CBS for multi-agent path planning tasks.

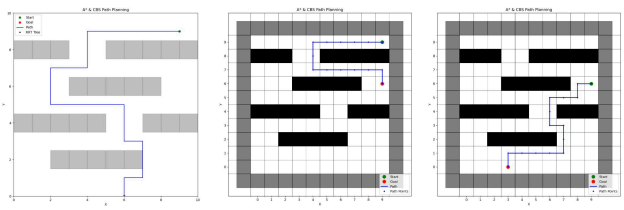

Figure 4: Comparison of Astar algorithm for paths with or without fetch points

Multi Robot Simulation Process (CBS + A)*

Statement of a* File and Modification of CBS File

In the code implementation process, we package the A* algorithm as a complete `AStar` class, where the final route backtracking is achieved through the `reconstruct_path` method, and the search method is used to search for the designated robot path. The specific initialization and function calling of the AStar algorithm are carried out in `CBS.py`, and the way points return is generated based on this algorithm from the initial point to the destination point. The key code is:

```
class Environment(object):
def    __init__(self,    dimension,    agents,
obstacles):

#    Omitting    the    initialization    of
configurations   for   obstacles   and   robot
environments

self.a_star = AStar(self)

# ...
def compute_solution(self):
# Route finding for each turtlerobot
    solution = {}
    for agent in self.agent_dict.keys():
        self.constraints =
       self.constraint_dict.setdefault(agent,
Constraints())
      local_solution =
      self.a_star.search(agent)
      if not local_solution:
          return False
      solution.update({agent:local_solution})
    return solution
```

Reorganize the Yaml Files and Visualization from Pybullet

To generate YAML files for waypoints, it is necessary to include the configuration of the robot's initial points, fetch points (intermediate goals), and final target points along with the corresponding obstacle environment configuration. Since the robot's tasks are divided into two phases —moving from the initial point to the fetch point, and then from the fetch point to the final target point—the `env.yaml` file needs to be split into two separate files to

reflect these distinct phases.
In the first phase:

- Turtlebot 1 starts at (9, 9) and moves to the fetch point at (9, 6).
- Turtlebot 2 starts at (0, 9) and moves to the fetch point at (0, 6).

In the second phase:

- Turtlebot 1 starts at the fetch point (9, 6) and moves to the final target point (3, 0).
- Turtlebot 2 starts at the fetch point (0, 6) and moves to the final target point (6, 0).

The two separate `env.yaml` files are structured as follows:

```yaml
agents:
  - goal:
    - 9
    - 6
    name: 1
    start:
    - 9
    - 9
  - goal:
    - 0
    - 6
    name: 2
    start:
    - 0
    - 9
map: ...
```

```yaml
agents:
  - goal:
    - 3
    - 0
    name: 1
    start:
    - 9
    - 6
  - goal:
    - 6
    - 0
    name: 2
    start:
    - 0
    - 6
map: ...
```

After generating the YAML files of the waypoints, we need to simulate the process of turtlebot motion. Therefore, in the `final_challenge.py` file, we need to modify the path of the generated YAML file in order to perform simulation.

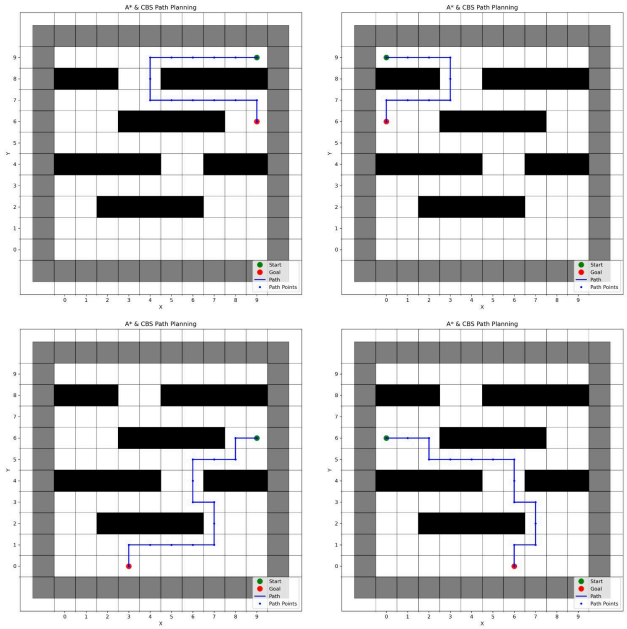

Figure 5: Path Display of A* Algorithm on Multiple Robots

Multi Robot Simulation Process (CBS+RRT/ RRT)*

Statement of RRT File and Modification of CBS File

Based on the theory of RRT algorithm introduced in the previous text, we have packaged the RRT algorithm into a complete class. The corresponding methods in this class include randomly generating sampling points, finding the planned path node closest to the sampling point, and attempting to extend the planned path node towards the sampling point, checking the effectiveness of the path, checking whether it has reached the target, and backtracking the complete path. It is worth noting that when attempting to expand the path towards the sampling point, we attempted to measure the length of the step_size from the starting point to the sampling point.

$$\text{direction} = \left(\frac{x_2 - x_1}{d}, \frac{y_2 - y_1}{d}\right) \tag{18}$$

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{19}$$

$$(x', y') = \left(x_1 + \text{direction}_x \cdot \text{step}_{\text{size}}, y_1 + \text{direction}_y \cdot \text{step}_{\text{size}}^2\right) \tag{20}$$

In addition, when checking whether the attempted path collides with obstacles, we used the cross product method for verification. In summary, we will attempt to cross product the coordinates of the starting and ending points of the path with the four corners of the obstacle one by one.

$$\text{val} = (q_y - p_y)(r_x - q_x) - (q_x - p_x)(r_y - q_y) \tag{21}$$

If val is 0, it expresses that the three points are collinear, indicating that the route collides with obstacles.

For the path finding calculation of RRT algorithm in CBS, the corresponding initialization and calling of search method are similar to A* algorithm, and will not be further elaborated here.

Improvement of RRT* Algorithm Compared to RRT Algorithm

Similarly, based on the original RRT algorithm, the RRTStar algorithm has more advantages in pathfinding strategies. The RRTstar algorithm has been briefly introduced in the previous text. Compared with the general RRT algorithm, the RRTStar algorithm quickly finds feasible paths and continuously optimizes them through rewiring operations. It searches for a path with lower cost near the random sampling point found by RRT to connect the established path point with the sampling point, making the generated path smoother. We achieve this through the following logic:

```
def rewire(self, new_node):
    for neighbor in self.tree:
        if self.distance(neighbor, new_node)
< self.search_radius:
                new_cost = new_node.cost +
self.distance(new_node, neighbor)
```

```
            if new_cost < neighbor.cost:
                neighbor.cost = new_cost
                   self.parent_map[neighbor]
= new_node
```

Mixed Simulation (A* + RRT*)

We have already discussed the impact of introducing fetch points on the generation paths of different algorithms in the previous text. Therefore, we propose a hybrid simulation method that uses fetch points to simulate two different stages using different algorithms. In our final group presentation, we use the A* algorithm in the first stage and the RRT* algorithm in the second stage under the simulation of the real environment. We have elaborated on the specific connection methods for the two stages in the previous text. It is worth noting that since we control multiple robots to start simulation simultaneously, we have restructured our human run method and used Python threads to operate them simultaneously.

```
def run(agents, goals, schedule, goals2,
schedule2):
"""
    Set up loop to publish leftwheel and
rightwheel velocity for each robot to reach
goal position.
  Args:
  agents: array containing the boxID for each
agent
  schedule: dictionary with boxID as key and
path to the goal as list for each robot.
  goals: dictionary with boxID as the key and
the corresponding goal positions as values
"""
  threads = []
  for agent in agents:
      t = threading.Thread(target=navigation,
args=(agent, goals[agent], schedule[agent],
goals2[agent], schedule2[agent]))
      threads.append(t) \ t.start()
  for t in threads:
      t.join()
```

Visualization

Fig. 6 illustrates a simulation environment created using PyBullet. The environment represents a maze-like structure with walls depicted in white and the floor arranged in a checkered pattern. In the center of the image, a small blue robot can be observed navigating the maze. A distinctive white square marker is placed on top of the robot, indicating the integration of a fetch action. This fetch marker represents the fetch point or task assigned to the robot as part of its navigation. The addition of this feature demonstrates the capability to perform fetch actions within the simulation, enhancing the robot's functionality and showcasing its ability to complete complex multi-stage tasks.

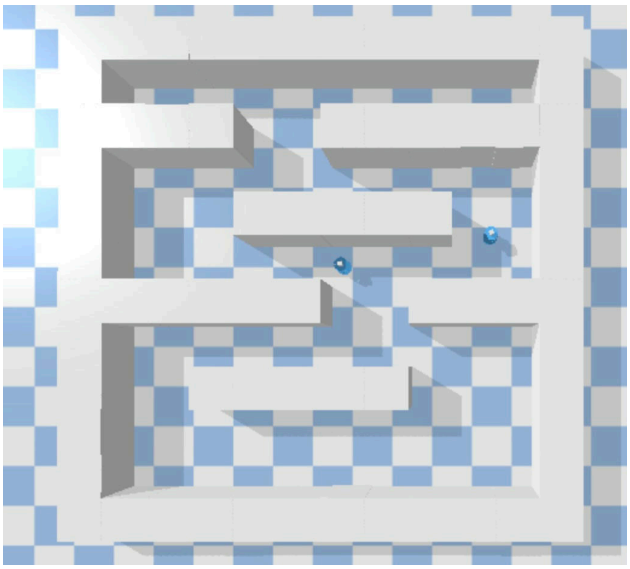

Figure 6: Simulation Fetch the Boxes

Fig. 7 illustrates the performance of the RRT algorithm for multi-robot path planning. It generates feasible paths quickly with a sparser exploration tree, making it computationally lightweight and faster in execution. However, the paths produced are suboptimal, often containing unnecessary detours and abrupt bends, which can increase travel time and reduce overall efficiency. This makes RRT suitable for scenarios where real-time, quick path generation is a priority, but precision and path smoothness are secondary considerations.

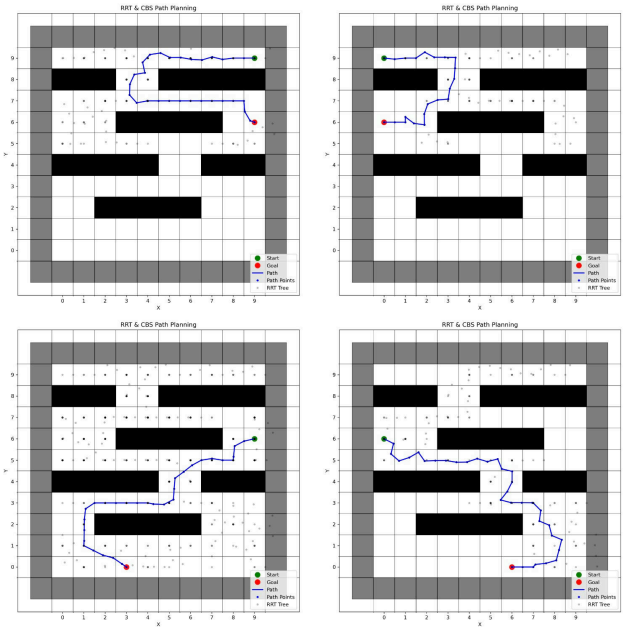

Figure 7: Path Display of RRT Algorithm

Fig. 8 showcases the results of the RRT* algorithm, which improves upon RRT by optimizing path quality through cost-based evaluation and rewiring. The exploration tree is denser, allowing for smoother and more efficient paths with fewer unnecessary turns. While this

results in significantly better path quality and navigation efficiency, it comes with a trade-off: higher computational requirements and longer planning times. RRT* is more appropriate for tasks that prioritize optimality and smooth movement over real-time responsiveness, especially in complex environments.

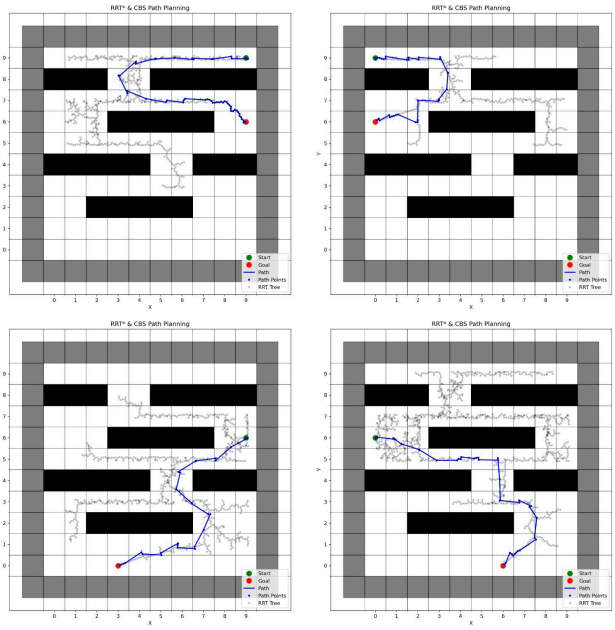

Figure 8: Path Display of RRT* Algorithm

Multi Robot Simulation Process (CBS+NMPC)

When using Nonlinear Model Predictive Control (NMPC) for multi-agent path planning, increasing the prediction horizon (horizon) often leads to higher computational demands. This increased computation can result in longer waiting times between control updates, causing noticeable delays or sluggish movement of the agents.

This Fig. 9 represents the path planning using Nonlinear Model Predictive Control (NMPC) with a prediction horizon of 4. The two robots are depicted in green, starting at different initial positions and aiming to navigate through a grid-based maze to their respective goals. The blue blocks represent obstacles, while the dashed lines (orange and blue) indicate the planned trajectories for the two robots.

However, both robots appear to stop midway instead of reaching their respective goal positions.I have put significant effort into analyzing this issue but still haven't fully understood the underlying reasons. When I increased the horizon to 8 to potentially improve the path planning results, the computational load grew significantly. After waiting for 3 hours, no feasible solution was obtained, and the robots even moved out of the 10x10 grid map unexpectedly.

Despite these challenges, I remain committed to addressing this issue. In the future, I will continue to refine the implementation and optimize the computational efficiency to ensure the system performs as expected. This process involves exploring advanced optimization techniques, improving model constraints, and potentially reducing the computational complexity of the NMPC framework to achieve reliable path planning for multi-agent systems.

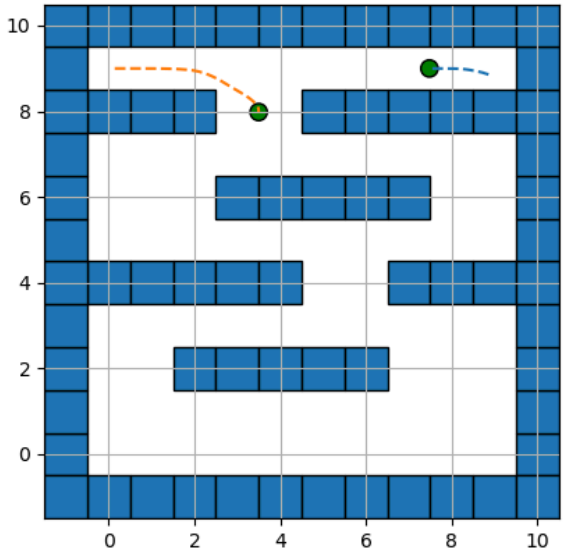

Figure 9: Path Display of NMPC (Horizon = 4)

REAL WORLD PROCESS

To establish the connection between our remote laptop and the TurtleBot3, SSH is used. SSH allows secure communication between our laptop and the TurtleBot3 over a network, enabling us to send commands, retrieve data, and monitor processes running on the robot. To connect the TurtleBot3(s) to our remote laptop, we perform the following steps:

- Connecting a Single TurtleBot3
  ```
  ssh ubuntu@192.168.0.219
  ```
- Connecting Multiple TurtleBot3
  ```
  ssh ubuntu@192.168.0.219
  ssh ubuntu@192.168.0.240
  ```

After connecting the TurtleBot3(s) to our remote laptop, we are required to connect the camera to our laptop and execute the necessary actions to get it in operation. To connect the camera, these are the set of commands that need to be performed:

(1)

```
# run the ros
roscore
```

(2)

```
# run the multiple aruco finder
roslaunch auto_aruco_marker_finde \
multiple_aruco_marker_finder.launch
```

(3)

```
#set video4 don't auto focus
uvcdynctrl --device=/dev/video4 \
--set='Focus, Automatic Continuous' 0
```

(4)

```
# show the GUI of camera screen
rosrun rqt_gui rqt_gui
```

The first step is to run command (1) as it sets up the communication infrastructure for the TurtleBot3 system. Secondly, command (2) is run to start detecting ArUco markers that are used for localizing the robot and mapping the between simulation and real-world coordinates. Before proceeding to run command (3), it is necessary to configure the `multiple_aruco_marker_finder.launch` file by adding the required ArUco IDs, such as 100, 101, 102, 103, 500, 501, 502, 503, and 504. This ensures that the system is properly set up to detect and identify the specified markers.

```
# add different id in the file
<group ns="id100">
<include
file="$(find auto_aruco_marker_finder)
/launch/aruco_marker_finder.launch">.
    <arg name="markerId" value="100"/>
</include>
</group>
<group ns="id101">...</group>
<group ns="id102">...</group>
<group ns="id103">...</group>
<group ns="id500">...</group>
<group ns="id501">...</group>
<group ns="id502">...</group>
<group ns="id503">...</group>
<group ns="id504">...</group>
```

Thirdly, we run command (3) to turn off the autofocus of the camera as it would affect the ability of the camera in picking up the ArUco markers placed at the corners of the camera vision. Lastly, command (4) helps to open a graphical interface that allows us to see the camera view and the ArUco markers placed within the view of the camera. Figure Fig. 10 shows the commands in the Terminal as well as the interface which displays the camera view.

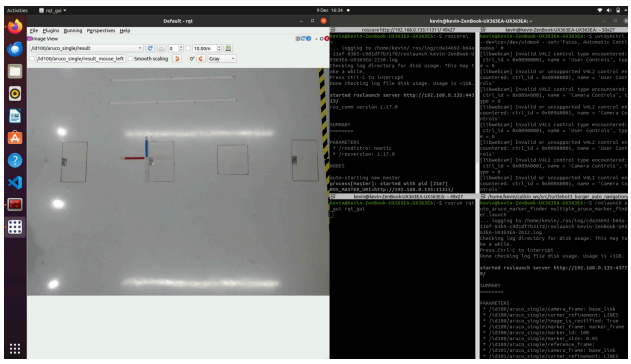

Figure 10: Commands executed in Terminal and Camera View from Interface

Single Agent Task

The navigation of a single agent starts with its initial localization using ArUco markers. The corners of the real-world environment are identified by specific IDs: id500 (top left corner), id501 (top right corner), id502 (bottom right corner), and id503 (bottom left corner). The agent itself is represented by id101, while the fetch point is marked by id102. To align the real-world coordinate system with the simulation map, the coordinates were adjusted from (0,10) to (-1,10), ensuring accurate positional alignment between the two systems. The TurtleBot3 is programmed to follow a series of predefined waypoints: beginning at the start point, proceeding to a designated fetch point, and ultimately arriving at the goal position. These waypoints, initially defined in simulation coordinates, are converted into real-world coordinates using a scaling matrix.

The navigation logic embedded in the code systematically guides the robot to each waypoint by verifying its current position and orientation before progressing to the next one. A tolerance of 0.05 meters is established, allowing the robot to consider a waypoint "reached" if it comes within this range, after which it advances to the subsequent waypoint. The linear velocity is used to drive the robot forward or backward, reducing the distance to the waypoint, while angular velocity ensures proper alignment with the target direction.

A PID controller was integrated into the system to enhance the robot's navigational precision and stability. This controller dynamically adjusts the robot's velocity based on error values in both linear and angular directions. By incorporating the PID controller, the robot is able to navigate efficiently with minimal overshooting or deviation, even in dynamic conditions. For instance, when the robot reorients itself towards the fetch point, the controller smooths out abrupt turns by finely tuning the angular velocity.

Additionally, the robot's speed is carefully managed to optimize efficiency and maintain control stability. The linear velocity is capped at 0.2 meters per second to prevent overshooting waypoints or compromising localization accuracy. Similarly, the angular velocity is limited to 1.0 radian per second to avoid sudden rotations that could cause the robot to deviate from its path. When the robot is far from a waypoint, its speed increases to reduce travel time. However, as it approaches a waypoint, its speed gradually decreases. This adaptive speed regulation, governed by the PID controller, plays a crucial role in achieving smooth and reliable navigation.

After running the simulation for the single agent, the simulated path coordinates are recorded and saved in the bash `cbs_output.yaml` file. The agent then autonomously navigates based on the provided coordinates. Executing auto-navigation for a single agent is significantly more straightforward since all other elements in the environment remain stationary. The A* algorithm was applied to generate the path planning for the TurtleBot3. Through extensive testing and adjustments, we successfully programmed the single agent to follow the designated path without any collisions during both attempts in the final showcase. The smooth trajectory achieved by the robot was largely attributable to the PID controller, which proved instrumental during the final demonstration.

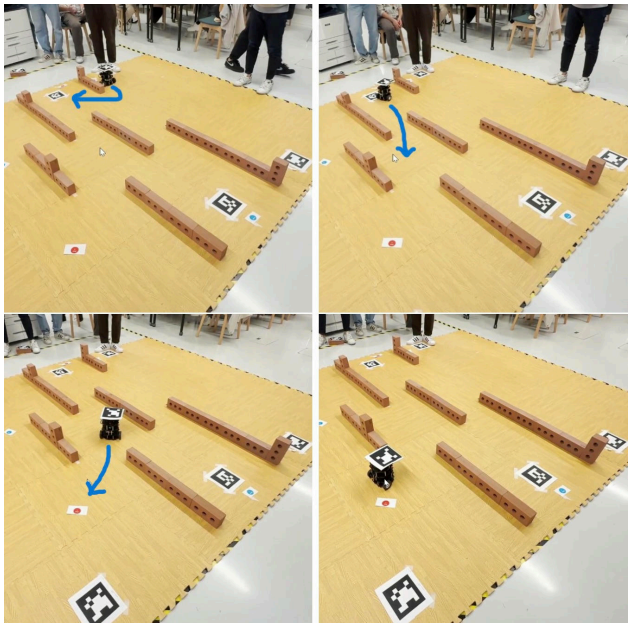

Figure 11: Auto Navigation of Single TurtleBot3 from Start point -> Fetch point -> End Point

Multi-Agents Task

The code used to execute multiple agents closely resembles the one designed for a single agent, with only a few adjustments made to the algorithm used for path planning of both TurtleBot3 robots. In the multi-agent implementation, we incorporated the threading method to enable simultaneous navigation of multiple robots, each following its designated path. Without threading, the navigation process would operate sequentially, requiring one robot to complete its path entirely before the next robot could begin.

```
def run(agents, ids, coordinates):
  """
      Set up loop to publish leftwheel and
rightwheel velocity for each robot to reach
goal position.
    Args:
     agents: array containing the boxID for
each agent
    schedule: dictionary with boxID as key and
path to the goal as list for each robot.
    goals: dictionary with boxID as the key and
the corresponding goal positions as values
  """
  threads = []
  for i in range(len(agents)):
      t = threading.Thread(target=navigation,
args=(agents[i], ids[i], coordinates[i]))
      threads.append(t)
      t.start()
  for t in threads:
      t.join()
```

The primary benefit of utilizing threading is its scalability to accommodate more than two robots, as each additional robot simply introduces another thread. Furthermore, each thread functions independently, managing its robot's navigation logic autonomously. Threads also share resources like the ROS Master without interfering with each other's operations, enhancing overall efficiency in terms of both time and resource utilization.

The path planning for multiple robots was achieved through the implementation of two distinct algorithms. For the initial segment of the path, from the start point to the fetch point, the *CBS with A** algorithm was employed, whereas the second segment, from the fetch point to the endpoint, utilized the *CBS with RRT** algorithm. After extensive experimentation and evaluation, we concluded that combining paths generated by these two algorithms resulted in the fastest and most effective trajectory. Consequently, both algorithms were integrated into the final implementation. Since these algorithms produced separate path segments, we employed the append function to merge them into a unified, smooth trajectory for each robot. This approach ensured seamless transitions between the two segments for efficient navigation.

```
# Read and transform waypoints from the YAML
file
coordinates1 =
read_and_transform_waypoints("./
cbs_output_1_fetch.yaml", matrix)
coordinates2 =
read_and_transform_waypoints("./
cbs_output_rrt_2.yaml", matrix)
pose102 =
rospy.wait_for_message(f'/id102/
aruco_single/pose', PoseStamped, timeout=10)
pose103 =
rospy.wait_for_message(f'/id103/
aruco_single/pose', PoseStamped, timeout=5)

coordinates2[0].insert(0,
[pose102.pose.position.x,
pose102.pose.position.y])

coordinates2[1].insert(0,
[pose103.pose.position.x,
pose103.pose.position.y])

coordinates_full = []
coordinate3 =
coordinates1[0] + (coordinates2[0])
coordinate4 =
coordinates1[1] + (coordinates2[1])

coordinates_full.append(coordinate3)
coordinates_full.append(coordinate4)
```

The waypoints had to be appended accurately to ensure smooth transitions without abrupt changes in position or orientation during execution. The trajectories generated by CBS with A* (coordinates1) and CBS with RRT* (coordinates2) were seamlessly concatenated. This integration created a unified and continuous trajectory for each robot, which was subsequently added to the `coordinates_full` list for execution.

Combining the paths derived from separate algorithms offers several notable benefits. First, it enables us to capitalize on the individual strengths of both algorithms. Second, the combined paths can be adaptively adjusted based on changes in the robot's environment. Lastly, this approach effectively addresses complex navigation challenges. Fig. 12 illustrates the paths followed during the autonomous navigation of multiple robots.

One of the major challenges we encountered was avoiding collisions between the two robots after leaving the fetch point and heading toward their final goal points. We experimented with various strategies, such as increasing the speed of one robot while reducing the speed of the other. Despite these attempts, the collision issue persisted. Ultimately, we opted to pause one robot at the fetch point for four seconds before resuming its movement toward the endpoint. Among all the methods tested, this approach proved to be the most reliable, effectively preventing collisions between the two robots.

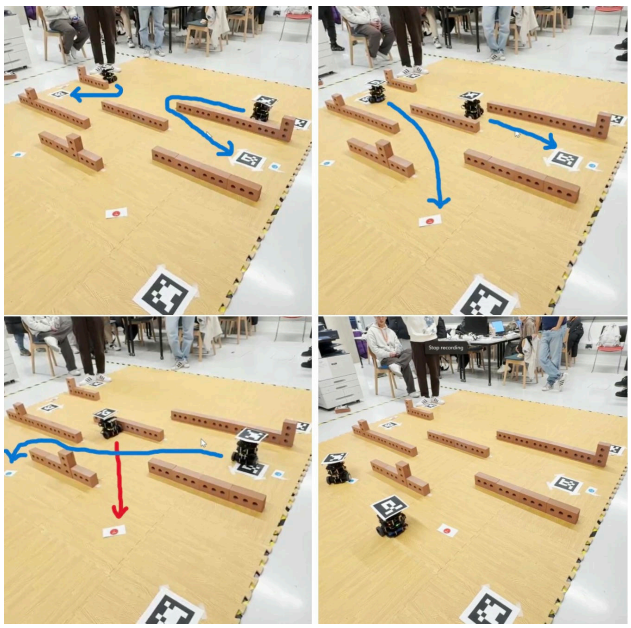

Figure 12: Auto Navigation of Multiple TurtleBot3 from Start point -> Fetch point -> End Point

DISCUSSION AND CONCLUSION

The project successfully explored the implementation of various path planning algorithms for autonomous navigation of TurtleBot3 robots in both simulated and real-world environments. Through the integration of algorithms such as A*, CBS, RRT*, and NMPC, we were able to assess their effectiveness in single-agent and multi-agent scenarios, highlighting the strengths and limitations of each approach.

In the simulation phase, the A* algorithm demonstrated efficiency in generating optimal paths in structured environments with static obstacles. Its deterministic nature ensured reliable navigation for single-agent tasks. However, when scaling to multi-agent systems, the A* algorithm alone was insufficient due to the potential for conflicts between agents. The incorporation of the Conflict-Based Search (CBS) algorithm addressed this limitation by effectively resolving conflicts through high-level and low-level planning. The CBS algorithm, combined with A*, enabled multiple robots to navigate simultaneously without collisions, although it introduced increased computational complexity.

The RRT* algorithm was employed to enhance path optimization, particularly in the second stage of navigation involving fetch points. RRT* provided smoother and more natural paths by continuously optimizing the tree of possible paths. Its probabilistic nature allowed for better exploration of the search space, which was advantageous in dynamic or less structured environments. The hybrid approach of using A* for the initial path segment and RRT* for the subsequent segment proved effective, leveraging the strengths of both algorithms to achieve efficient and collision-free navigation.

Despite these successes, the implementation of Nonlinear Model Predictive Control (NMPC) presented challenges. While NMPC has the potential to optimize multi-agent path planning by predicting future states and minimizing a combined cost function, it proved computationally intensive for our application. Increasing the prediction horizon to improve path planning results led to excessive computation times and impractical delays, rendering NMPC unsuitable for real-time control in our multi-agent system. This highlights the need for further optimization or alternative approaches to reduce computational demands when using NMPC in similar contexts.

In the real-world implementation, the use of ArUco markers for localization and mapping proved effective. The robots were able to accurately interpret their positions and navigate accordingly. The integration of a PID controller significantly improved the robots' navigational precision and stability, reducing overshooting and ensuring smooth transitions between waypoints. The threading method facilitated simultaneous multi-agent navigation, although it introduced challenges in coordinating the agents to prevent collisions. Implementing a delay for one of the robots at the fetch point emerged as a practical solution to mitigate collision risks.

The project demonstrated the feasibility of combining multiple path planning algorithms to enhance autonomous navigation in both simulation and real-world environments. The use of A* and CBS provided a solid foundation for multi-agent path planning, while the integration of RRT* contributed to smoother and more optimized paths. The challenges encountered with NMPC underscore the importance of considering computational efficiency in real-time applications.

Future work could focus on optimizing NMPC or exploring alternative predictive control methods that offer a balance between computational demands and performance. Additionally, further refinement of multi-agent coordination strategies, such as dynamic speed adjustment or more sophisticated conflict resolution mechanisms, could enhance the scalability and robustness of the system. Expanding the system to accommodate more agents and testing in more complex environments would also provide

valuable insights into the practical limitations and potentials of the implemented algorithms.

Overall, the project contributes to the advancement of autonomous multi-agent systems by showcasing effective strategies for path planning and navigation, highlighting both successes and areas for improvement. The lessons learned lay a foundation for future research and development in the field of robotic navigation and coordination.

APPENDIX

This project repository, containing all source codes, configuration files, and documentation, is available on GitHub at the following link: **`COMP0182 Real-world Multi-agent Systems Repository`**

REFERENCES

[1] R. Karthik, R. Menaka, P. Kishore, R. Aswin, and C. Vikram, "Dual mode PID controller for path planning of encoder less mobile robots in warehouse environment", *IEEE Access*, vol. 12, no. , pp. 21634–21646, 2024, doi: `10.1109/ACCESS.2024.3363898`.

[2] A. Visioli, *Practical PID control*. Springer Science & Business Media, 2006.

[3] A. J. Moshayedi, A. Abbasi, L. Liao, and S. Li, "Path planning and trajectory tracking of a wheeled mobile robot using bio-inspired optimization algorithms and PID control".

[4] A. Souliman, A. Joukhadar, H. Alturbeh, and J. F. Whidborne, "Real time control of multi-agent mobile robots with intelligent collision avoidance system", in *2013 Science and Information Conference*, 2013. pp. 93–98.

[5] S. I. Khather, M. A. Ibrahim, and M. H. Ibrahim, "PID controller for a bearing angle control in Self-driving vehicles", *Journal of Robotics and Control (JRC)*, vol. 5, no. 3, pp. 647–654, 2024.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths", *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968, doi: `10.1109/TSSC.1968.300136`.

[7] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning", *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011, doi: `10.1177/0278364911406761`.

[8] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[9] I. A. Sucan, M. Moll, and L. E. Kavraki, "The open motion planning library", *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012.