Behavior-Based Interactive Multi-Agent Robot Algorithms in Complex Environment

Ji Hyun Kim, Sree Deeksha Bethapuri, Hyeongjin Kim, Mitchell Whalen

October 2023

1 Introduction

In the realm of automation in industrial and large-scale settings, robots are effective in open spaces with automated processes and few constraints, but they have limitations beyond the scope of these simple environments. Robots tend to be less effective when the system is only partially automated or when the environment is more complex (i.e. narrow paths or many obstacles) or unfavorable for robot-robot interaction or human-robot interaction. This project aims to improve upon current multiagent pathfinding methods.

Previous research has separated multiagent pathfinding into two principal approaches: centralized and decentralized modes. The centralized model features a master planner, responsible for organizing the sequence of actions for each agent. Simultaneously, this master planner determines the potential for interference among agents and works to proactively mitigate these conflicts. However, as the number of agents within a multi-robot system increases, the systemic dynamics increase in complexity, rendering the management of such systems increasingly difficult. Furthermore, the reliance on a centralized decision-making unit for the entire system can lead to susceptibility to data flow congestion, particularly in instances of system expansion.

Thus, the decentralized approach has gained popularity. When using the decentralized approach, no singular central control dictates orders; rather, agents engage in inter-agent interactions to collectively arrive at optimal paths. It must be acknowledged, however, that a significant portion of this research has been confined to artificial environments. These environments often exhibit oversimplified grid structures, wherein limited consideration is granted to variations in path dimensions and agent attributes. The resultant lack of authenticity to real-world complexities poses challenges in the practical applicability of these methodologies to many operational contexts.

Notable gaps in current multiagent pathfinding research include a disproportionate emphasis on centralized approaches over decentralized ones, a prevalence of studies in simplified simulated environments that lack real-world complexities, challenges in scalability as the number of agents increases, insufficient consideration of dynamic environmental factors, and a lack of consideration for resource constraints. Collectively, these gaps hinder the application of theoretical advancements to practical, real-world multiagent systems.

To address these gaps, we will be exploring 3 different multi-agent pathfinding algorithms: token passing, reinforcement learning, and adaptive learning. Token passing algorithms involve agents communicating and passing tokens to coordinate their movements efficiently. Reinforcement learning leverages a dynamic approach, allowing agents to learn optimal paths through trial and error Adaptive learning algorithms enable agents to continuously adjust their strategies based on real-time feedback, fostering a responsive and flexible multi-agent system. By investigating these different approaches, we have attempted to increase the understanding of each algorithm's strengths and limitations, ultimately paving the way for a more robust and adaptive multi-agent pathfinding solution.

As we explore these multi-agent pathfinding algorithms, the primary evaluation metric will be the total time taken. The three algorithms will be tested on 4 different environments with 2 levels of size and complexity. The environment variable will be controlled by running all 3 algorithms on the same set of environments. The optimal algorithm will take the least time to accomplish the goal. An algorithm that generalizes well to environments of all sizes and complexity is ideal.

Overall, the A* algorithm with active behavior performed the best in terms of time taken across all 4 levels of the environment. The token passing algorithm fared the second best generally, while the reinforcement learning model lagged. There were minor technical difficulties with the adaptive A* algorithm that we will elaborate upon later. Nonetheless, this paper demonstrates the potential of adaptive and token-passing algorithms to enhance MAPD. Moving forward, we will aim to increase the effectiveness of the adaptive elements. We believe improvements here can not only improve our results in the test environment but adaptive elements can improve the solution's ability to scale to more complex and dynamic environments.

2 Related Work

2.1 Independent, Uncoordinated Algorithm

Centralized approaches such as Conflict-Based Search algorithm (CBS) involve a centralized planner that generates sequences of actions while checking for conflicts between agents. However, this is unsuitable in environments that are yet to be explored or in case the environments keep changing.

On the other hand, in **decentralized approaches**, where a central planner does not exist, agents find collision-free paths themselves. Two state-of-the-art algorithms work in a decentralized way, where all agents are independently moving uncoordinated.

One of them is *PAPO*, which stands for Path and Action Planning with Orientation. This algorithm is designed to generate collision-free paths in multi-agent pickup and delivery tasks within restricted or non-uniform environments. This algorithm especially focuses on restricted path widths and node sizes, which were not often considered in conventional, decentralized approaches. They used uniform models, which made it more difficult to implement in real-world applications [**?**].

The other one is a group of the current variations of *Token Passing (TP)*, where agents pass tokens to each other which contains all the information needed for their path planning, pickup, and delivery. The original Token Passing algorithm is not suitable for multi-agent situations since it does not handle collisions well. However, current work on adding replanning features and adding different parameters within the features has significantly improved the algorithm's performance in a multiagent setting. Specifically, *k-TP and p-TP* parameters focus on dealing with the problem of collision by adding multiple plans or probabilistic approaches in the initial path planning process, guaranteeing the model to be more robust and reliable to handle multiple collisions with less increase in cost, time, and distance [**?**].

2.2 Reinforcement Learning

A paper aiming to use reinforcement learning to address the multiagent pickup and delivery ("MAPD") problem by HyeokSoo Lee and Jongpil Jeong compares the efficacy of the Q-Learning algorithm to the Dyna-Q algorithm. The Q-Learning algorithm "stores the state and action in the Q Table in the form of a Q Value; for action selection, it selects a random value or a maximum value according to the epsilon-greedy value and stores the reward values in the Q Table". The Dyna-Q algorithm is a combination of the Q-Learning algorithm and the Q-Planning algorithm. The Q-Planning algorithm "obtains simulated experience through search control and then changes the policy based on a simulated experience". The Dyna-Q algorithm aims to combine real and simulated experience to build a policy. In this paper, only a single agent was used across several environments, and the paper found that the Q-Learning algorithm was faster than the Dyna-Q algorithm on average, however, the mean final path length for the Dyna-Q algorithm was shorter. Since our primary evaluation metric is total time, we proceeded with using the Q-Learning adapted for multiple agents [**?**].

2.3 Adaptive Learning

A research group from the Julio Godoy Department of Computer Science led by Julio Godoy (2017) reported on the Adaptive Learning Approach for MultiAgent Navigation ("ALAN"). Real-time goal-directed navigation of multiple agents is required in various domains. ALAN is a scalable, adaptive learning approach that allows agents in multi-agent navigation to dynamically adapt their behavior, resulting in faster and collision-free motions compared to existing methods. The approach introduces an offline Markov Chain Monte Carlo method to learn an optimized action space for each environment. ALAN dynamically adapts the motion of agents using the Softmax action selection strategy and a limited time window of rewards.

This paper evaluates the design choices of ALAN, such as the action selection method, the time window length, and the balance between goal progress and politeness, controlled by the coordination factor y in the reward function. The objective is to minimize travel time while ensuring collision-free motion. ALAN outperforms other navigation approaches in terms of travel time [**?**].

Kita and colleagues (2023) were inspired by increasing demand for home deliveries and labor shortages in the delivery field, so they explored using automated robots to solve this problem. This paper proposes an adaptive method of parameter updates with online re-planning to handle MAPF problems. The proposed methods prioritize parameter updates on important edges for path planning with online re-planning of travel paths with updated parameters, resulting in the quick implementation of robust planning for automated delivery services. The proposed method outperforms existing methods regarding the number of conflicts and flowtime [**?**].

3 Methods

The main methodology was to compare three Multi-Agent path Planning algorithms and check which one works the best in restricted environments. So we chose three algorithms, and implemented them in four different, i.e.; small simple, small complex, large simple, and, large complex environments.

3.1 Independent, Uncoordinated Algorithm

Among these two state-of-the-art algorithms, we will be focusing on *TP-replanning, K-TP,* and *P-TP* algorithms in this research due to their robustness and efficiency in comparison to the other algorithms of its category. The more conventional MAPD algorithms do not adequately address real-world challenges by using non-restricted environments and agents being prone to delays and failures. Whereas, one of the more recent algorithms, PAPO, focuses heavily on a non-uniform environment so that multi-agents can also be deployed in places that keep changing, such as construction sites. However, this is only adequate in outdoor environments, not the environment that this research focuses on, such as small warehouses, stores with narrow paths, or libraries. Therefore, K-TP and p-TP focus more on handling delay to have a more robust solution to MAPF and MAPD problems.

3.2 Reinforcement Learning Algorithm for Multi Agents

Since the Reinforcement learning algorithms are well-suited for learning optimal strategies based on the environment, we decided to implement one where the agents could learn how to find the optimum path by avoiding each other using a reward function.

3.3 A* Algorithm with Active Behavior

Once we started implementing the reinforcement learning algorithm, we came to realize that the learning is not that great in small environments. We reconsidered our initial hypothesis and instead of having a behavior-based algorithm that would require extensive training, we chose a simple A* algorithm and tried to check if adding an active behavior would help better in finding the optimum path and finishing the task faster.

4 Architectural Design

4.1 TP Algorithms

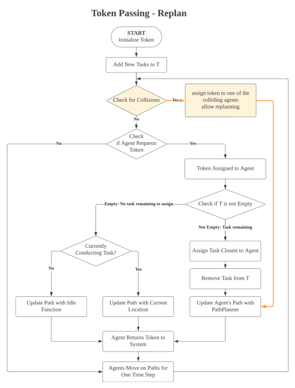

Token Passing - Replan
START
Initialize Token
Add New Tasks to T
Check for Collisions
Yes
assign token to one of the colliding agents allow replanning
No
Check if Agent Requests Token
No
Yes
Token Assigned to Agent
Check if T is not Empty
Empty: No task remaining to assign
Not Empty: Task remaining
Currently Conducting Task?
No
Yes
Assign Task Closest to Agent
Remove Task from T
Update Path with Idle Function
Update Path with Current Location
Update Agent's Path with PathPlanner
Agent Returns Token to System
Agents Move on Paths for One Time Step

Figure 1: TP-Replanning

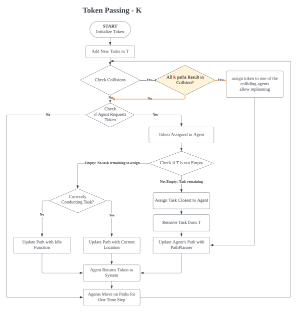

Token Passing - K
START
Initialize Token
Add New Tasks to T
Check Collisions
Yes
All k paths Result in Collsion?
Yes
assign token to one of the colliding agents allow replanning
No
No
Check if Agent Requests Token
No
Yes
Token Assigned to Agent
Check if T is not Empty
Empty: No task remaining to assign
Not Empty: Task remaining
Currently Conducting Task?
No
Yes
Assign Task Closest to Agent
Remove Task from T
Update Path with Idle Function
Update Path with Current Location
Update Agent's Path with PathPlanner
Agent Returns Token to System
Agents Move on Paths for One Time Step

Figure 2: K-TP

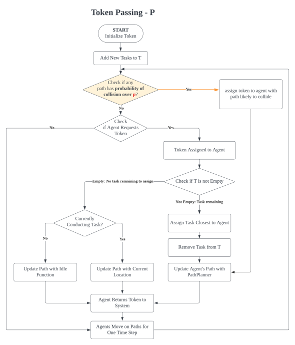

Token Passing - P
START
Initialize Token
Add New Tasks to T
Check if any path has probability of collision over p?
Yes
assign token to agent with path likely to collide
No
Check if Agent Requests Token
No
Yes
Token Assigned to Agent
Check if T is not Empty
Empty: No task remaining to assign
Not Empty: Task remaining
Currently Conducting Task?
No
Yes
Assign Task Closest to Agent
Remove Task from T
Update Path with Idle Function
Update Path with Current Location
Update Agent's Path with PathPlanner
Agent Returns Token to System
Agents Move on Paths for One Time Step

Figure 3: P-TP

Overall Architechcure Figure1 illustrates the structure of three variations of the token-passing algorithm. On the far left is, *Token Passing - Replan*, with the yellow highlighted areas indicating features added to the original Token Passing algorithm to enhance re-planning possible when collisions occur. To take a closer look into that architecture as it is the base for the other variations, it initializes the token that collects all the information needed for the agents to perform tasks. Subsequently, new tasks are appended under 'T' and stored within the tokens. The system then checks for collisions among agents. If there are no collisions, it proceeds to assign the token to a specific agent, either in a prearranged sequence based on their IDs, or dynamically, considering factors like proximity to tasks or whether they are idle or busy. Agents without the token initially remain idle or move to a standby location until they receive the token. Once an agent receives the token, it examines 'T' for remaining tasks. If tasks are present, the agent is allocated the nearest task, which is then removed from 'T'. The agent plans its route to perform the task and updates its planned path on the token. Upon returning the token to the system, all agents move a step along their planned paths. This cycle repeats from the collision check, with immediate token reassignment for replanning if a collision occurs, continuing until all tasks are completed and agents return to their starting positions.

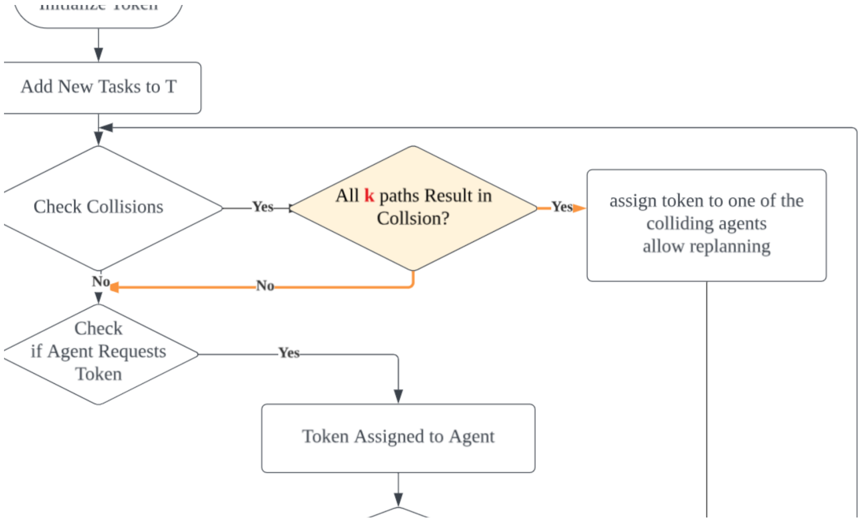

Initialize Token
Add New Tasks to T
Check Collisions
Yes
All k paths Result in Collsion?
Yes
assign token to one of the colliding agents allow replanning
No
No
Check if Agent Requests Token
Yes
Token Assigned to Agent

Figure 4: K-TP

K-TP and P-TP Figure2 differs from figure1 primarily after the collision check as shown in figure4. K-TP agents initially plan 'k' possible paths for task execution, which eliminates the need for immediate replanning in the event of a collision, so that they can resort to a backup pre-planned paths. Consequently, unlike TP-replanning, which is less robust in case a lot of replanning is required, the 'k' parameter introduces an alternative where agents assess the availability of pre-planned paths before the token is reassigned to an agent in collision.

Figure3also diverges at the same structural point as shown in figure5. This algorithm incorporates a threshold 'p' representing the probability of a collision. When an agent is assigned a task and plans a path, it selects one with a collision probability below the value 'p'. Thus, even in the absence of an actual collision, if the collision probability rise due to other agents' new plans, the affected agent will preemptively request the token at the subsequent step.

4.2 Reinforcement Learning

The first part of setting involves building the environment. This involves creating a map, creating behaviors, and setting rewards. To build the environment, we used tkinker to draw the operation desk, shelves, robots, and targets. The robots' behavior is basically a 2D grid environment, so it consists of simple actions such as freeze, up, down, left right. You can also set the case of random movement and movement by selecting the next action according to the q-table.

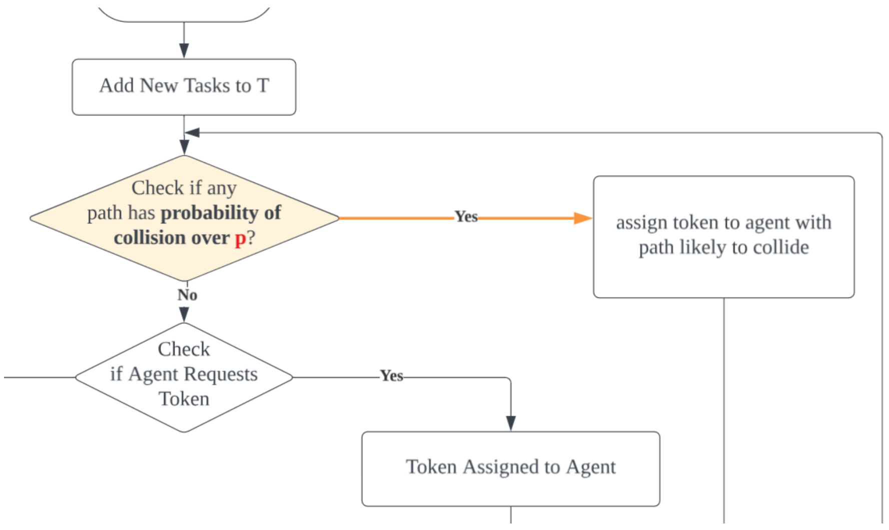

Add New Tasks to T
Check if any path has probability of collision over p?
Yes
assign token to agent with path likely to collide
No
Check if Agent Requests Token
Yes
Token Assigned to Agent

Figure 5: P-TP

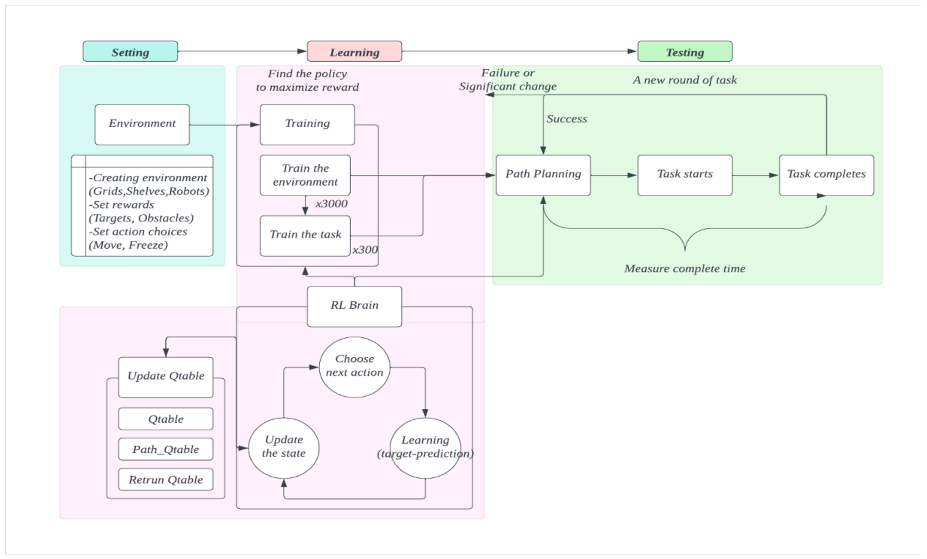


Figure 6: Reinforcement learning

For the rewards, we set the rewards of the obstacles (shelves, people, other robots) that the robots have to avoid while performing the task, and we define the following behaviors as HIT. When they hit the obstacles, they will get -50. We also set the reward for the target that each robot has and the position to get back to its original position as +50, which we define as ARRIVE.

Then there is the learning procedure. We can first train the environment with the raw Qtable and then train the task with the updated Qtable. The training goes through iterations with RL brain updating the Qtable. Specifically, we update a Path$_Qtablethatinitiallymovesrandomlytoagoal, aReturnQtablethatreturnstoitsoriginalposition, anda$

Finally, there is a test phase based on the trained results. We don't modify the Qtable from training, just update it with additional information about the given task to perform the task. If the environment changes significantly or the content of the task does not change significantly, you can proceed to TEST without training, but otherwise, you need to go back to the LEARNING phase to train again.

4.3 A* algorithm with active behavior

The first step in solving a task using the A* algorithm with Active behavior is setting up. In this step, the obstacles, the operation desk, the positions of the robots, and the targets are drawn on a 2D grid. As with the previous algorithm, we configure the algorithms Complex large, complex small, sim v ple large, and simple small. In the calculation phase, we will use the existing algorithm, the A* algorithm, to find the path. At this time, as explained earlier, we cannot add the active behavior function due to technical limitations, but if it is added, the completion time and performance will be the same as A* without impulsivity, so we use the A*algorithm assuming that the active behavior function is added. The optimal path is then executed in the next step, test. If we are given a new task, we will update the new changes in the map and recalculate before performing the task.

5 Formal Design

5.1 TP Algorithms

As outlined in the preceding section, each TP algorithm responds to scenarios where collisions are evident. Reflecting on the specific actions taken by each algorithm under figure7, I will now delve into the intricacies of how each algorithm proceeds with path planning.

At the ninth step with agents 0, 1, and 2 advancing towards their targets, the yellow zone marks where collisions are likely. In this scenario,

- TP: The agents proceed until a potential collision is anticipated near the black obstacle. After they collide, agent obtains the token to replan their path. The strategy concludes one agent waiting for another to pass before proceeding.

- KTP (k=2): As agent 1 and 2 pre-planned alternative paths, they agent 1 goes downward towards its goal, while agent 2 pauses before the yellow zone until agent 1 clears the area.

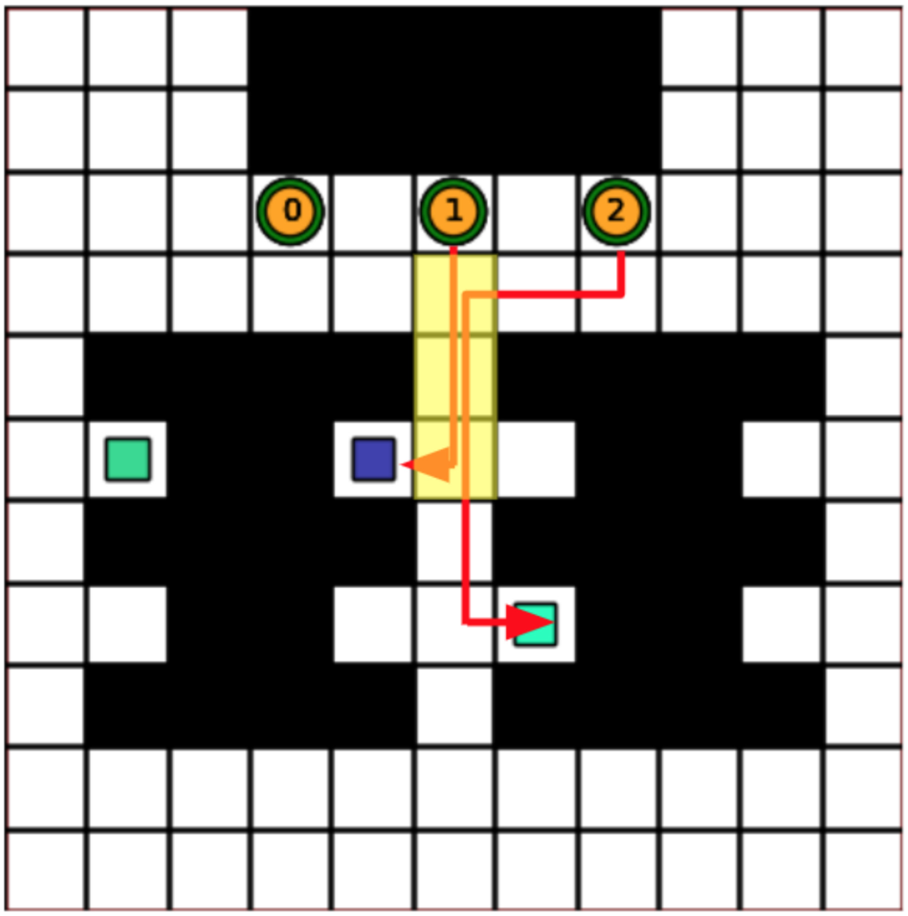


Figure 7: TP algorithm visualization 1

- PTP (p=0.5): Agents calculates that the collision probability surpasses 0.5, agent 2 retraces its steps, avoiding the obstacle entirely, and then resumes its journey to the target.

5.2 Reinforcement Learning

5.2.1 Algorithm

We'll explain in detail how to solve this problem using reinforcement learning. The main idea is to determine a policy that updates the Qtable and selects an action for the state based on the updated Qtable. In general, there are two cases in Q-learning: greedy and -greedy (epsilon greedy). Here, we use -greedy to balance the desire to explore new paths with the desire to utilize previously learned information. Let's see the detailed design in pseudo code.

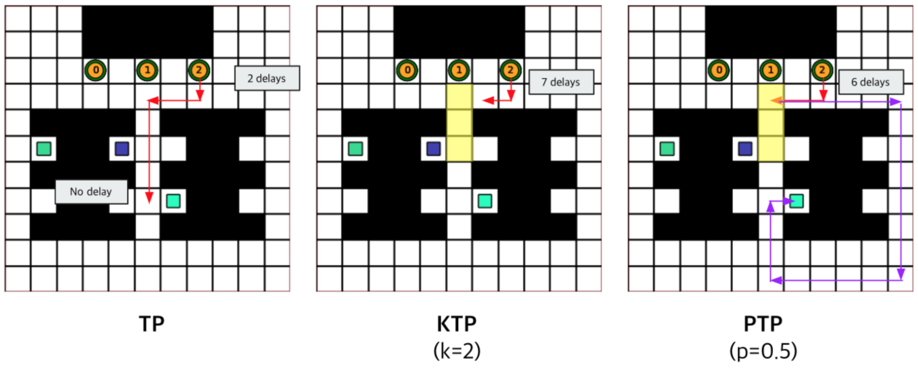


Figure 8: TP algorithm visualization 2

Algorithm 1: Epsilon-Greedy Q-Learning Algorithm

```
Data: $\alpha$: learning rate, $\gamma$: discount factor, $\epsilon$: a small number
Result: A Q-table containing Q(S,A) pairs defining estimated
        optimal policy $\pi$*
/* Initialization                                              */
Initialize Q(s,a) arbitrarily, except Q(terminal,.);
Q(terminal,.) $\leftarrow$ 0;
/* For each step in each episode, we calculate the
   Q-value and update the Q-table                              */
for each episode do
    /* Initialize state S, usually by resetting the
       environment                                             */
    Initialize state S;
    for each step in episode do
        do
            /* Choose action A from S using epsilon-greedy
               policy derived from Q                           */
            A $\leftarrow$ SELECT-ACTION(Q, S, $\epsilon$);
            Take action A, then observe reward R and next state S';
            Q(S, A) $\leftarrow$ Q(S, A) + $\alpha$ [ R + $\gamma \max_a$ Q(S', a) - Q(S, A)];
            S $\leftarrow$ S';
        while S is not terminal;
    end
end
```

Figure 9: RL$_p seudocode$

First, let's take a closer look at how we update the Qtable in each episode. Q(s,a) is the Q-value (expectation of reward) for taking action in state s. is the learning rate, R(s,a) is the reward received, is the discount rate, and maxaQ(s,a) is the maximum possible Q-value in the next state. Q-value Q(s,a) is the value of the expected total reward for choosing action a in a particular state s. The goal of Q-learning is to find the action that maximizes this value. Learning rate is regulating the impact of new information on the existing Q-value: a high learning rate makes new information more important, while a low learning rate makes past data more retained. Reward R(s,a) is the immediate reward an agent receives from the environment when it takes a certain action. For example, encountering an obstacle gives a reward of -50 and reaching a goal gives a reward of +50. We use a table called a Q-table to store and update the expected value of the reward for each state and action combination.

Based on these updated values, the policy is determined by the following formula.

$$\text{Action at time(t)} \begin{cases} \text{random action} & \text{if random number} < \epsilon \\ \arg\max_{a'} Q(s,a') & \text{otherwise} \end{cases}$$

Depending on the value of epsilon, which varies with the degree of learning, we choose a random behavior with some probability and the optimal behavior, argmaxaQ(s,a), with the remaining probability 1-. This approach allows the agent to explore new possibilities while still finding the optimal behavior. In the early stages, it facilitates learning through a variety of experiences, and in the later stages, it allows the agent to make optimal decisions based on these experiences.

5.2.2 Adjustment

Learning rate : In the original code, the learning rate was initially set to 0.03 to explore the entire map, and then the learning rate was gradually reduced after the 500th time to allow for detailed exploration. However, in this experiment, we adjusted the learning rate so that it could be sufficiently trained because we trained each user according to various environments.

Irritation time : I changed the conditions several times to train the model optimally for a given environment. When we changed from the original simple large environment to another environment, the degree of training varied depending on the location of the target. For example, in the complex environment, it took more than 300 iterations to train enough for the environment and task, while in the simple environment, it took more than 100 iterations to achieve performance. To compare the optimal performance, we trained with a different number of repetitions to achieve sufficient performance for each task.

Human intervention: The original code contained human elements in the obstacles. Humans are one of the dynamic obstacles and the robot waits up to

5 seconds when it encounters a human. If after 5 seconds the human has not moved out of the way, the robot will take a step back and explore a new path. In this experiment, for the purpose of comparison with other algorithms, two existing humans were fixed in the corner next to the operation desk to minimize the following human intervention.

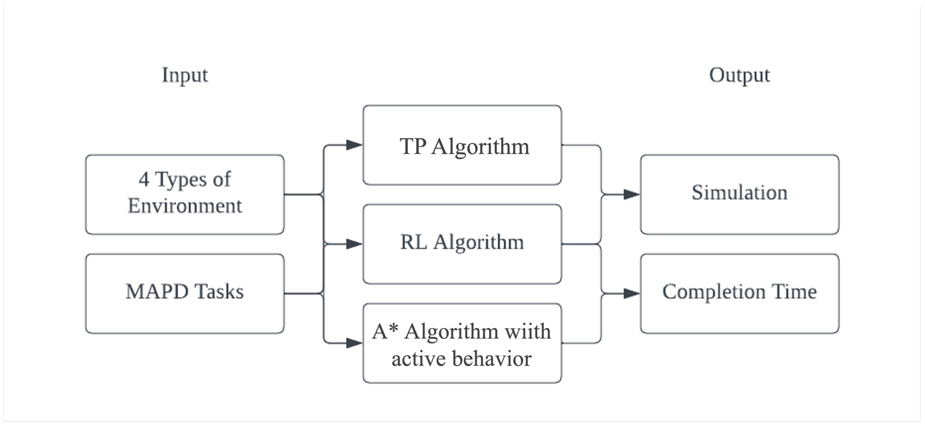
Collision avoidance strategy: The robot uses the Chebyshev distance strategy as an avoidance strategy when it encounters an obstacle. The Chebyshev distance is a metric that considers the maximum of the absolute differences of the Cartesian coordinates. It allows movement in all eight directions - the four cardinal directions plus the four diagonal directions. In scenarios where the robot is capable of moving diagonally in addition to the original four directions, the Chebyshev distance becomes more appropriate. This is because it better represents the shortest path a robot can take when diagonal movements are possible. The Chebyshev distance is effective in environments where diagonal movement does not incur additional cost or time compared to moving in the cardinal directions. It provides a more accurate measure of distance for robots with this enhanced mobility.

Transfer learning: It is inefficient to train by randomly moving and updating the qtable without any information. If you first learn the map through raw data, you can minimize the training for a given task. This is where transfer learning comes in. By transferring what the robots have learned about the environment to the next task and to other robots, the training speed can be reduced while maintaining accuracy. However, in this project, we did not use transfer learning because we aimed to compare the performance of the final selected path in a limited environment.

5.3 A* algorithm with active behavior

The A* algorithm solves pick-up and delivery tasks in warehouses by efficiently finding the shortest path between points, and it does so using a combination of actual travel costs and heuristic estimates. The algorithm calculates the total cost (f-cost), which is the sum of the "actual cost" (g-cost) and the "estimated cost to the goal" (h-cost, heuristic cost) of each node (location). The actual cost is the cost from the starting point to the current node, and the heuristic cost is the estimated cost from the current node to the goal. The algorithm prioritizes exploring the node with the lowest total cost, and the process is repeated to determine the best path to the final goal point. As you can see the cost function in the flowchart below, $f(n)=g(n)+h(n)$, the total cost $f(n)$ for a node n in the A* algorithm is calculated where $g(n)$ represents the actual cost from the start node to the current node n, and $h(n)$ is the heuristic estimated cost from node n to the goal. The paths between these points are the edges, with $g(n)$ representing the travel cost (e.g., distance, time) along these paths. The heuristic $h(n)$ could be a straight-line distance to the goal or a more complex estimate based on warehouse layout and obstacles. A* starts at the initial node and explores neighboring nodes. At each step, it calculates $f(n)$ for each node and prioritizes nodes with the lowest $f(n)$. This process continues until the algorithm reaches the goal. The path with the lowest total cost f(n) is selected as the optimal route. When the existing A* algorithm was implemented in a multi-agent environment, it could not be used because it could not avoid other agents. Therefore, we envisioned a logic that if the robots collided with each other, they would hand over their goals and items to the other robot and receive their opponent's goals and items. Although we could not directly implement it due to technical limitations, we assumed that it would be possible to obtain the same performance as the result of performing the task by passing each other. Therefore, in this project, we presented the flowchart on the right below to show the conceptual introduction and the effect of adding such an active behavior function for easier understanding.

6 Implementation



For our experiments, we created four environments and assigned the same multi-agent pick-up and delivery task to each environment. We prepared and tested the TP algorithm, reinforcement learning, and active adaptive algorithms to perform the task according to these conditions. This allowed us to visualize the work of each algorithm and measure its completion time. The following is a detailed methodology for running each algorithm.

6.1 TP Algorithms

Using the code from the paper, we tailored it to fit our experimental needs.

For the basic setup, we diverged from the original code by designing a standardized environment. We modified the default grid to vary in size and shape of obstacles as previously detailed. We also predetermined all tasks and robots to ensure a consistent comparison, altering the number and location of tasks, and fixing the sequence of task assignments and token distributions, rather than randomizing them.

Regarding parameter values, we opted for two distinct settings for each: one closely aligned with the original TP algorithm and another different. For 'p' values, lower numbers indicate a higher stringency in path selection to avoid encounters with other agents. Higher 'k' values mean agents start with multiple plans, reducing the need for immediate replanning. These specific values were selected after preliminary tests showing that 'k' values beyond 2 and 'p' values below 0.1 did not significantly impact the outcomes.

As for tasks, we slightly modified from the original code, which had agents retrieving items from shelves. Instead, our agents had to place items in designated locations and then return. We adjusted the code so the target points would lighten when task are given, and darken when agents returned to their start points.

In terms of outputs, while the original code recorded each robot's total time, the number of replans, and runtime, our project focused on the total time. Nevertheless, we considered additional metrics such as maximum time taken, number of replans, and delays to make informed choices between the TP variations before contrasting them with others for a comprehensive analysis. The outcomes were graphically represented to illustrate these factors across all parameters in every environment.

As a result, we executed the modified code across four different environments, each tested with five distinct parameters: TP-replan, two 'k' values (1 and 2), and two 'p' values (0.5 and 0.1). For specific instructions and packages used on running the code and its visualization, please refer to the attached Python file and the revised Readme within the code.

6.2 Reinforcement Learning

The code consists of a total of 4 .py files and 12 qtable files. The robots each have a qtable that records their path to the goal, a qtable that records their path from the goal to their initial location, and a qtable that records their combined path.
1) New$_{s}mall_{m}aze.pyisthefilethatconfigurestheenvironment.Itdrawsthegridmap, createsobjectssuchasrobot$
$Thispackageisusedforcreatingstatic, interactive, andanimatedvisualizationsinPython.Weusedittoseetheres$
$PickleisutilizedforserializinganddeserializingPythonobjectstructures.Insimplerterms.Weusedittoloadand$
$AfundamentalpackageforscientificcomputinginPython, NumPyprovidessupportforlarge, multi-$
$dimensionalarraysandmatrices, alongwithacollectionofhigh-levelmathematicalfunctionstooperateonthese$
$Thispackageisessentialfordatamanipulationandanalysis.5)Tkinter : TkinteristhestandardGUI(Graphical$
$TheTimepackageprovidesvarioustime-relatedfunctions.Itisoftenusedformeasuringcompletiontime7)Sys :$
$ThismoduleprovidesaccesstosomevariablesusedormaintainedbythePythoninterpreterandtofunctionsthatint$

6.3 A* algorithm with active behavior

Since we have four different environments, we used four different Python files, each defining its own environment using TKinter. Since A* is suitable for navigation tasks in grid-based environments, all four environments implement active behavioural learning using A*. The robots seek the shortest path using A* and do not have information about each other. The algorithm was used to find the shortest path to the goal using a priority queue that explored potential paths. If the robots collide or come across each other on their way to the goal, they exchange goals based on the distance and proceed ahead. When the robots collide, a function is triggered where the goals (targets) of the robots get exchanged. The algorithm calculates the time taken by the robot to finish the task by capturing the timestamps of start-time and finish-task, which represent the start and end times. We printed this as a calculation in the console for further comparison. The following packages were used in this code. Numpy was used as it is a numerical computation library was used to perform numerical operations, array handling and calculations related to coordinates and positions. The Time package provides various time-related functions. It is used for measuring completion time in this code. Heapq module provided an implementation of priority queue for the A* algorithm to retrieve the element with smallest priority efficiently. Tkinter is a standard GUI toolkit for python was used to create the warehouse environment, including shelves, robots, and targets. Sys module was used to handle differences between python 2 and python 3 when importing the Tkinter Module.

7 Evaluation

In our evaluation chapter, we conducted a comprehensive comparison of various algorithms in the context of warehouse automation, utilizing a set of four distinct map configurations: complex large, complex small, simple large, and simple small. These maps were meticulously designed to test the algorithms under varying conditions while maintaining consistency in essential aspects. Each map, despite differences in layout and written type, shared identical locations for targets, robot starting positions, and obstacles, ensuring a level playing field for algorithm comparison. This setup allowed us to assess how each algorithm adapts to different spatial challenges, ranging from navigating through tight, cluttered spaces in the complex maps to efficiently covering more ground in the simpler, larger maps.

The operational framework of the warehouse simulation involved three robots, each initially stationed at the operation desk, awaiting orders. Upon receiving a command, a robot's task was to navigate to a specific shelf, pick up an item, and return it to the operation desk. With three robots and three items involved in the simulation, this setup provided a dynamic environment for testing the algorithms' efficiency and coordination capabilities. Our primary metric for comparison was the total time taken to complete all tasks, providing a straightforward yet effective measure of each algorithm's performance. To make it happen, we control the speed of robots in each algorithm as moving 1 grid per sec. Additionally, to gain deeper insights into the behavior and interaction of the robots under different algorithms, we recorded and analyzed video footage of the simulations. This visual analysis not only served as a means to corroborate our time-based evaluations but also offered valuable qualitative insights into the algorithms' navigational strategies, collision avoidance mechanisms, and overall coordination efficiency in a realistic warehouse setting.

We didn't consider the calculation time and learning time. Since we want to compare the best performance among algorithms, we gave each algorithm enough time to calculate the path and learn the path. If we include the preparation time as well, it would be difficult to compare all algorithms along the same lines. For example, in the case of RL, how many times training is repeated is important, but if such training time is included, it is difficult to obtain the same results every time and it is difficult to meet the comparison standard. Therefore, we compared only the execution time to compare the performance of the algorithms, although it is difficult to apply them directly to the real environment.

7.1 TP Algorithms

Overall Evaluation In our constrained environment with limited path variations, k-TP and p-TP in general did not yield superior results compared to the baseline TP with replanning. Given the small space, the potential for delay was minimal, diminishing the advanced planning capabilities of k-TP and p-TP unnecessary. The baseline TP algorithm already optimized for collision avoidance effectively, indicating that pursuing paths with even lower collision probabilities was unnecessary in terms of time or path deviation. Essentially, once a certain threshold of collision probability was reached with the baseline, further changes did not lead to better outcomes. Instead, it led to a saturation point where any additional complexity in path planning did not improve the results or even led to worse results by adding complexity in time and distance.

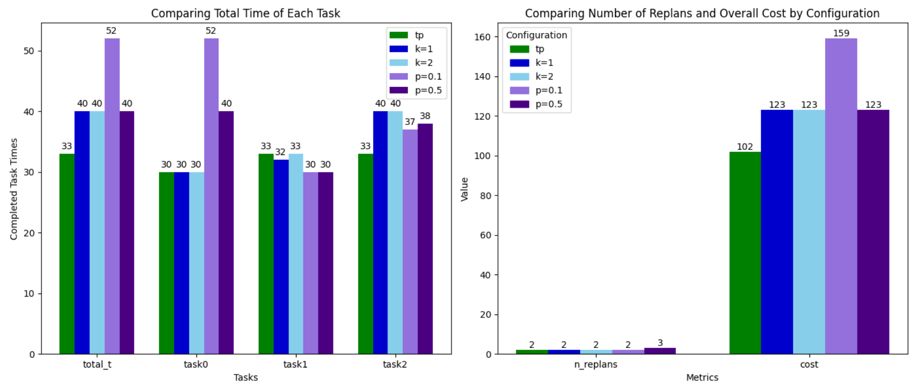


Figure 10: Results of TP Algorithms on large complex environment

Environment: Large Complex

- Total Time: TP-replanning outperformed all other parameters consistently, with p=0.1 demonstrating notably poorer performance.

- Time by Tasks: While most parameters resulted in comparable times for task completion, p=0.1 took a noticeably longer duration to complete task 2.

- Delays and Cost: Most configurations encountered two delays with the exception of p=0.5, which had an additional delay, though this was not substantially different. The cost, however, was significantly lower for TP-replanning and higher for p=0.1.

- Overall: Taking all factors into account, TP-replanning emerged as the most efficient configuration.

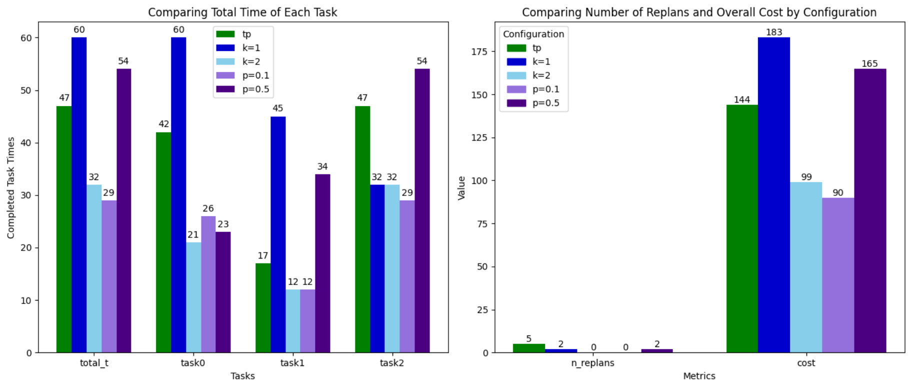


Figure 11: Results of TP Algorithms on small complex environment

Environment: Small Complex

- Total Time: The p=0.1 configuration exhibits the best performance across all configurations, with k=2 closely following. Conversely, k=1 is the least effective.

- Time by Tasks: There is a consistent pattern in the time taken to complete tasks across all configurations, indicating a similar task efficiency.

- Delays and Cost: Both p=0.1 and k=2 configurations, which are the top performers, experienced zero delays, pointing to optimal pathfinding. TP-replanning, on the other hand, underwent replanning five times, suggesting frequent collisions.

- Overall: The p-TP configuration with p=0.1 is the superior performer, signifying that in environments with a high potential for collisions, a stricter probability threshold is beneficial for avoiding such incidents.

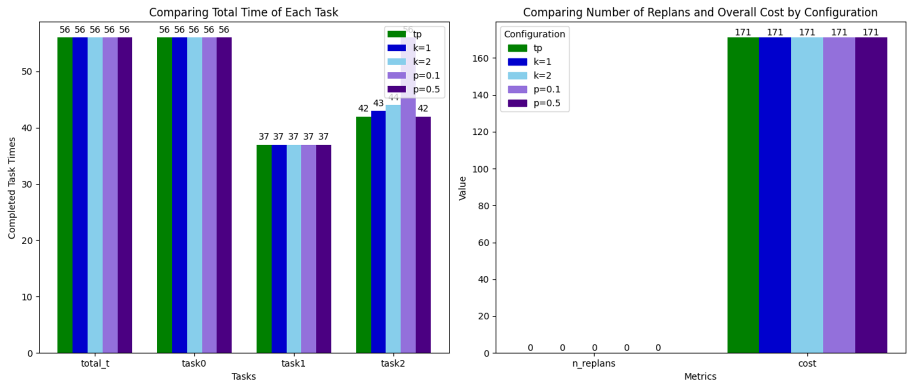


Figure 12: Results of TP Algorithms on large simple environment

Environment: Large Simple

- Total Time: All configurations yield identical total completion times, suggesting that the environmental simplicity negates the need for more complex algorithmic variations.
- Time by Tasks: The completion times for individual tasks are consistent among the first two tasks for all configurations. The only deviation occurs in the final task, where p=0.1 incurs a notably higher time investment.
- Delays and Cost: There are no replans required for any configuration, and the incurred costs do not vary between configurations.
- Overall: Given these observations, it is evident that fine-tuning parameters offers no advantage in this simple environment. Consequently, TP-replanning, serving as the baseline algorithm, is deemed optimal due to its comparative simplicity and equivalency in performance outcomes.

Environment: Small Simple

- Total Time: TP-replanning and p=0.5 are tied for the shortest total time, while p=0.1 takes the longest.
- Time by Tasks: The time to complete individual tasks is consistent across configurations, following a similar trend.
- Delays and Cost: There are minimal replans for most configurations, and costs mirror the trends observed in the total time taken.
- Overall: TP-replanning and p=0.5 emerge as the most optimal configurations in terms of total time, with p=0.1 lagging behind. This suggests that while p=0.5 manages to achieve efficiency comparable to TP-replanning, the stricter avoidance criteria of p=0.1 may result in unnecessary delays and increased cost.

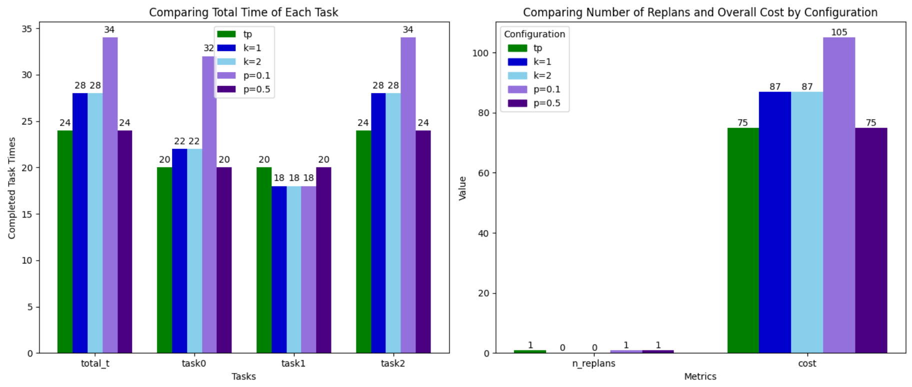


Figure 13: Results of TP Algorithms on small simple environment

7.2 Reinforcement Learning

Evaluation adjustment in RL, all of the existing code is written to return to its original position after arriving at the target location, but we've changed this so that it can return immediately. Also, since the training time was taking too long when measuring the time, we set the time to 0.2 seconds to move one space and adjusted the measured time to match the same speed as the other algorithms. However, we found it difficult to scale from a simple 0.2-second product to a 1-second product. This is because each robot is not following a predetermined path, but is using information that is updated in real time to find its way around, which takes time to compute. We tried to use one gird per one second speed, but the results took way too long to see the results. Therefore we choose an alternative way. Since we have a simulation of each training, we could choose the last simulation and see the how many grids that robots moved. We measured the completion time by calculating how many grids that the last arriving robot moved and how long it waited when other robots were moving. We did it There is a chance that it would take more time to update the state and choose next action, so we decided to leave it as a margin of error.

7.3 A* algorithm with active behavior

The evaluation of the A* algorithm with active behavior was tricky as we started with an assumption that the robots will be exchanging goals when they meet or collide with each other at any point. The robot moved one block per second for this simulation as well to keep the results synchronous. This algorithm gave the best result in a simple small environment, and the next best was a simple large environment. where the robots didn't have to collide at any point. In the other two, the robots collide but still have a better result than the other two algorithms. This is due to the reason that they don't wait for the other robot to finish the task before moving.

7.4 Simulation analysis

8 Results

8.1 Time completion

	TP	Reinforcement Learning	A* algorithm with active behavior
Complex Large	33 (tp)	53	27
Complex Small	29 (p=0.1)	28	17
Simple Large	56 (tp)	94	15
Simple Small	24 (tp)	40	11

Complex Large : In Complex Large environment, A* with active behavior outperforms both TP and RL algorithms, completing tasks in only 27 seconds compared to 33 and 53 seconds, respectively. The Reinforcement Learning (RL) algorithm takes longer, finishing in 53 seconds, indicating a higher complexity handling challenge. This showcases A*'s superiority in managing complex, large-scale tasks efficiently. Complex Small :In the Complex Small environment, the A* algorithm again excels with the fastest completion time of 17 seconds. RL is competitive in this scenario, with a close completion time of 28 seconds, demonstrating good performance in smaller complex environments. The TP algorithm, while slower at 29 seconds, still shows reasonable efficiency. Simple Large : A* with active behavior significantly outperforms in Simple Large environments, completing tasks in only 15 seconds. The TP algorithm takes considerably longer, at 56 seconds, indicating a less efficient approach in simpler, larger spaces. RL also struggles the most in this category, with a completion time of 94 seconds. This suggests that while A* efficiently handles simple, large tasks, TP and RL may face challenges in scaling up in simpler environments. Simple Small : A* maintains its lead in the Simple Small environment with an impressive 11 seconds completion time. The TP algorithm shows better performance than in larger environments, finishing tasks in 24 seconds. RL takes notably longer at 40 seconds, suggesting challenges in efficiency even in simpler, smaller settings as well.

8.2 Simulation analysis

In reinforcement learning we observed 1 time collision with another agent in complex large environment, 3 times collision in complex small environment, 2 times collision with other agents in simple large, 3 times collision with other agents and 2 times collision with shelves in simple small even though we trained the model fully. Also it shows they even though we train the model fully, if there was an error during training, we could only know about the error after the implementation. And even if we choose another path or wait to avoid a collision, we can conclude that reinforcement learning is not suitable in this environment because it already shows the lowest performance. Also, as looking at the simulation results of the A* algorithm, we found see that bottlenecks occur. In this case, we need an algorithm that can predict and resolve these bottlenecks rather than relying on the active behavior of individual agents. In such a crowded situation, it is possible that the performance of the algorithm will be worse than other algorithms.

9 Discussion

9.1 Real World Application

In the real world, finding an optimal multi-agent pickup and delivery solution for robots maximizes the efficiency of resource utilization which can reduce costs and increase productivity. Building off of the A* algorithm with active behavior could lead to reduced traffic congestion, improved delivery services, and greater scalability of autonomous systems in a variety of settings beyond typical industrial ones. Thus, an optimal multi-agent pathfinding solution for robots with goals holds the potential for a breakthrough in autonomous systems by making algorithms more adaptive, efficient, and applicable across diverse scenarios.

However, there were some limitations with this project and its ability to produce results directly applicable to the real world. First, although we attempted to construct environments of varying complexity, these environments were rather static. For example, obstacles, goals, and the number of agents were all static. For this project to be useful it must be able to handle to scale up its computational complexity and applicability to dynamic environments. Furthermore, there is a trade-off between optimality and performance in many cases. This can especially be seen with the reinforcement learning algorithm which has to balance exploration and exploitation. But for all pathfinding algorithms, one of the key challenges is striking a balance between true optimality and performance (in this case total time).

9.2 Strength and limitation of TP

Variations of the TP algorithm has its strength having a straightforward design, where editing parameters is easy to bolster robustness against delays or collisions. This simplicity makes them an appealing option for real-world scenarios characterized by static environments and straightforward robotic tasks.

Our observations affirm that these algorithms excel in managing collisions, making them ideal for settings where multiple robots operate concurrently with sufficient leeway for route adjustments.

Nonetheless, there are constraints. The robots' reliance on token information may limit efficiency gains achievable through real-time inter-robot communication. Moreover, while the TP algorithm considers replanning time in its calculations, actual deployment may reveal extended durations if the computational hardware is suboptimal or if the communication network among warehouse robots is inadequately configured. These factors could prolong the expected time for replanning and task execution, affecting the overall system performance.

9.3 Strength and limitation of RL

Training it to find the most optimal path is very difficult. It is certainly possible to train it to solve a given task, but whether it is the most effective way to do so requires further validation. Given the trade-off between exploration and exploitation, even if there is a lot of exploration at the beginning, as you continue to iterate, you will tend to go down the path you have learned. While this is a helpful direction for learning, it is very unlikely to find the optimal path if one becomes familiar with the learned path before discovering other effective paths. To solve this problem, existing algorithms apply a high learning rate at first to increase exploration, and then reduce the learning rate as it passes, but this does not solve the problem completely.

The second is the problem of learning rate and exploration. In complex environments, when the target is located in a corner or in a narrow gap, setting the learning rate too high will not explore the remote path even though it may be more effective. This problem also occurs when the learning rate is lowered to 0.001, making the training time too long to lower it further. This can be partially solved by setting different levels of rewards for different environments and updating rewards for different regions and stages, but it can make it difficult to adapt to changing conditions in the future. Third, it was possible to adapt to small changes and form an optimal path based on the trained qtable, but for large changes, new training was required. For example, if the number of items increases, the location changes significantly, or the obstacles change, new training is required. A new algorithm would be needed to determine whether to retrain or continue as is, and timely execution would be difficult in such cases. This is especially challenging in complex environments where there are many changes in the location and number of obstacles and targets. To solve this problem, reinforcement learning that can accommodate feedback is being researched.

As a direction for future research, it is crucial to explore methodologies that integrate feedback mechanisms into the reinforcement learning process. This could involve developing systems that not only validate the outcomes of the learning phase but also incorporate real-time feedback during training. Such an approach would allow for the continuous refinement and adaptation of learning algorithms, ensuring they remain effective and relevant in dynamically changing environments. By integrating feedback loops, both post-training and in-training, the learning process can become more responsive to new information and challenges, leading to more robust and efficient learning outcomes.

9.4 Strength and limitation of A*algorithm with active behavior

Our initial analysis with two robots in a single lane gave a good result, but as we increased the number of robots to three, the robots just collided with each other and continued going in the same lane at the same time. This is partly due to the environment. The ideal environment for the robots to exchange goals would be a single lane where the robots need to cross each other to finish their assigned task. but in most of our environments, they don't have to cross each other; they travel the blocks together sometimes as the tasks are not in different directions. But despite the mixed up result, we realised that having an active behaviour would definitely make a difference for the robots. And simple algorithms like A* can be enough to achieve the goal instead of reinforcement algorithms for small environemnts.

10 Conclusion

In this study, we compared different algorithms that can be used for multi-agent pick up and delivery tasks. Especially in relatively small, complex, and changing environments, such as libraries or grocery stores, the existing algorithms used in big warehouses are not suitable and adaptive methods are needed. Therefore, to find an algorithm that is more adaptive to other robots or humans in such environments, we built four types of different environments (complex large, complex small, simple large, and simple small) and compared the performance of TP algorithm, RL, and A* with active behavior by analyzing the completion time and simulation. Our results clearly show that The TP algorithm shows better results than RL in almost all situations, which suggests that the TP algorithm is better at implementing adaptive behavior in complex situations. In particular, RL has many limitations in that it needs to be rewarded and retrained to adapt to changing conditions in order to properly train for complex environments. By analyzing the simulation, we could learn about the limitations of each algorithm and future direction. In our simulations, we found that RL tended to keep making errors and training on them, and over time, it stuck to the path it had learned, even if it was inefficient. The A* algorithm with active behavior was also not perfect. It showed signs of poor performance when bottlenecks occurred or when many robots crowded into an area. To sum up, TP and RL, which are widely used in large warehouses, have been identified as difficult to adaptively apply in relatively small and complex environments, and a simple A* algorithms with active behavior characteristics can perform better, suggesting the need to apply active behavior to existing algorithms in the future. These results open the door for other research projects that will focus on adaptive algorithms for complex and collaborative settings in order to maximize the interaction between robot-robot and human-robot and performance.

References

[1] Tomoki Yamauchi, Yuki Miyashita, and Toshiharu Sugawara. Efficient path and action planning method for multi-agent pickup and delivery tasks under environmental constraints. *SN Computer Science*, 4(1):83, 2022.

[2] Giacomo Lodigiani, Nicola Basilico, and Francesco Amigoni. Robust multi-agent pickup and delivery with delays, 2023.

[3] HyeokSoo Lee and Jongpil Jeong. Mobile robot path optimization technique based on reinforcement learning algorithm in warehouse environment. *Applied Sciences*, 11(3), 2021.

[4] Karamouzas Ioannis Godoy, Julio, Stephen J. Guy, and Maria Gini. Adaptive learning for multi-agent navigation. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multi-Agent Systems*, pages 1577–1585. International Foundation for Autonomous Agents and Multi-Agent Systems, 2015.

[5] Atsuyoshi Kita, Nobuhiro Suenari, Masashi Okada, and Tadahiro Taniguchi. Online re-planning and adaptive parameter update for multi-agent path finding with stochastic travel times, 2023.

[6] Issa Zidane, Khalil Ali Khalil Ibrahim, Wavefront and A-Star Algorithms for Mobile Robot Path Planning, Conference: International Conference on Advanced Intelligent Systems and Informatics, 2018

[7 baeldung , Epsilon-Greedy Q-learning,https://www.baeldung.com/cs/epsilon-greedy-q-learning, March 24, 2023]