BEN GURION UNIVERSITY OF THE NEGEV

Faculty Of Engineering Sciences

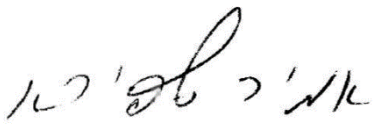
Department Of Mechanical Engineering

Multi-agent algorithm for managing and organizing warehouse packages

23-29

Stav Biton - 208854273

Supervised by: Prof. Amir Shapiro



July 2023

Abstract

Progress and development in the field of logistics have raised the need to optimize the shipping process between the manufacturer and the consumer. One way to optimize the process is to automate sub-processes along the package's route. One of the steps that can be automated is the path the package takes within the warehouse. At this stage, the package is picked up upon arrival at the warehouse and placed on a specific shelf. When it is time for the package to be shipped, it is picked up to be shipped. In this project, the focus is on automating this step.

The primary objective of this project is to ensure the efficient movement of packages within a warehouse by utilizing heterogeneous multi-agent systems i.e. systems comprising various types of robots. This will be achieved by developing a novel algorithm that leans on the principles of existing algorithms such as the Hungarian Algorithm for Multi-Agent Task Allocation (MATA), and Conflict-Based Search (CBS) for Multi-Agent Path Finding (MAPF). These algorithms have been independently tested and proven effective but have not yet been combined in this manner and tested with heterogeneous agents in a 3D environment. The developed algorithm will be evaluated in a simulation environment.

This project reviews the literature and theoretical foundations of MATA and MAPF. In addition, the report provides a thorough analysis of the written algorithm that is utilized for the management of packages within the warehouse and how it is based on existing algorithms. To measure the effectiveness of the algorithm, a simulation environment was developed in Python. A detailed description of this simulation environment is provided along with illustrative examples showing the implementation of the algorithm. The performance of the algorithm is rigorously evaluated in the simulated warehouse environment, and the resulting results and conclusions drawn from running the algorithm in a simulation are discussed and analyzed in detail.

Table of contents

Table of Figures

Table of tables

Table of algorithms

Table of Symbols

Symbol	Description
A	Group of agents from the same type
$\mathbb{A}$	Set up groups of different agents
a	An agent from the 'A' group
a_x	x coordinate of agent
a_y	y coordinate of agent
C	Set of constraints
D	Set of values that the variable can take in constraint satisfaction problems
d	Distance
d_1	Distance
d_2	Distance
$E(G)$	A set of edges
$f(n)$	Cost function
G	Graph
$g(n)$	The cost of the path from the start node to node 'n'
$h(n)$	Heuristic function
j	Number of tasks
k	Number of packages
n	Number of agents
P	Packages
$s(i)$	Agent's source
t	Tasks
t_x	x coordinate task
t_y	y coordinate task
$V(G)$	A finite, nonempty set of vertices
X	Set of variables
π	Action space
ϕ	Storage places on the shelves

1 Introduction

Autonomous robots have the potential to revolutionize the way people work and live, and their use is increasing rapidly in a variety of industries. The use of Multi-Agent Systems (MAS) increases the capabilities that can be performed automatically. In the field of logistics and in particular in warehouses, autonomous MAS can be used to automate the movement of packages, reducing the need for human labor and increasing efficiency. However, coordinating the movement of multiple agents can be a complex task especially in crowded or dynamic environments.

One way to address this challenge is through the use of multi-agent pathfinding algorithms, which allow multiple agents to find the optimal path to a destination with no collisions. These algorithms consider the capabilities and constraints of each robot, as well as the layout of the environment and potential obstacles. In addition to pathfinding, another important aspect of coordinating MAS is task allocation, which involves assigning specific tasks to each agent in a way that maximizes efficiency and minimizes performance time. To date, the integration of these two fields has been studied in a limited manner way the literature as will be presented in Chapter 2, there has not been an examination of the algorithm with the use of heterogeneous agents in 3D environment. The amalgamation of these fields has the potential to significantly enhance the efficiency of the overall process in comparison to previous research.

In this project, several tasks were performed to develop and evaluate an algorithm for managing packages through heterogeneous multi-agent systems in a warehouse environment. First, a literature review was conducted to gain insight into how previous studies have approached task assignment and collision-free path finding. Based on this, relevant algorithms were selected as a foundation and a new algorithm for managing packages in a warehouse was formulated. In addition, a simulation environment was developed to visually test and evaluate the capabilities of the algorithm, and a performance analysis was conducted to evaluate the effectiveness and limitations of the algorithm.

2 Literature Review

The domain of multi-agent motion planning and task allocation is vast and can be categorized in numerous ways. The relevant scope for this research is determining the storage location of each package upon arrival at the warehouse, assigning a robot to transport the package to the designated location, formulating travel routes for each robot within the warehouse and implementing measures to avoid collisions among the robots. In this chapter, a literature review will be conducted on the basis of relevant articles pertaining to the topics of planning a path for a robot and assigning tasks. Additionally, an article that integrates these two fields together will be presented.

2.1 Multi Agent Path Finding (MAPF)

Multi-Agent Pathfinding (MAPF) is the problem of finding paths for multiple agents such that every agent reaches its goal and the agents do not collide [1]. The classical MAPF problem is defined with agents and by tuple $\langle G, s, t \rangle$ where:

- $G = (V, E)$ is undirected graph
- $s : [1, \dots, k] \rightarrow V$ maps an agent to a source vertex
- $t : [1, \dots, k] \rightarrow V$ maps an agent to a target vertex

Time is discretized into time steps. At every time step each robot can do an *action* – *wait* or *move*.

There are two environmental factors that impact the solution and should be considered: the structure of the environment (such as size, shelf location, delivery point, etc.) and the number of robots. There is a decrease in the quantity of packages that can be transfer within a given period of time as the environment becomes increasingly complex. On the other hand, increasing the number of robots can increase the speed at which packages are moved. However, if the number of robots surpasses a certain threshold, the density of space becomes too high, leading to a decrease in the number of packages that can be moved [2].

The MAPF problem is a common challenge in artificial intelligence and is known to be NP-hard, meaning that finding an optimal solution can take exponential time. One common method for solving MAPF with homogeneous agents is Constraint-Based Search (CBS), which has two levels: high-level and low-level. At the high level, a constraint tree (CT) is created with nodes containing time and location constraints for a single agent. The low-level search generates paths for individual agents that are compatible with these constraints. If conflicts between agents are detected, additional nodes with constraints to resolve the conflict are added to the CT. This process is

repeated until all agents have been assigned paths, at which point the search is terminated and the final paths are returned as the solution [3] [4]. The use of the CBS algorithm is not considered appropriate for addressing problems involving a high number of agents.

In order to effectively address the MAPF problem for a large number of agents, several enhancements to the CBS algorithm have been proposed. One such enhancement is Enhanced CBS (ECBS), a complete and bounded-suboptimal variant of the original CBS algorithm. ECBS employs focal search in both the high-level and low-level portions of the algorithm as opposed than best-first search, to achieve suboptimal solutions [5][6]. In addition, to improve the efficiency of finding solutions for a large number of agents, an extension of the ECBS algorithm called Explicit Estimation CBS (EECBS) has been introduced. EECBS incorporates the use of Explicit Estimation Search (EES) at the high-level of the algorithm as an alternative to focal search. Moreover, EECBS includes a number of enhancements to CBS such as the ability to bypass conflicts, prioritize conflicts, apply high-level heuristics and incorporate symmetry reasoning [7]. Both algorithms ECBS and EECBS are not capable of functioning in dynamic environments when the goals of the agents are altered during their movement. Additionally, these algorithms are not equipped to handle cyclical goal assignments and are unable to incorporate new goals for an agent once it has reached its current goal.

The MAPF problem can also be solved using deep learning techniques, such as the use of heuristic search methods. One example of this approach is the PRIMAL algorithm, which combines reinforcement and imitation learning to train fully decentralized policies for agents. These agents are able to reactively plan paths online in a partially observable environment, while exhibiting implicit coordination [8]. PRIMAL has demonstrated good results in low-density environments. To address more complex and constrained worlds PRIMAL2 has been developed [9]. Due to the intricate nature of the PRIMAL 2 algorithm, its execution time is significantly prolonged and as such it is not suitable for use in systems where rapid response and a brief execution time are crucial requirements.

2.2 Multi Agent Task Allocate (MATA)

The Multi-Agent Task Allocation (MATA) problem is a type of optimization problem which known as NP-hard problem even with small numbers of robots and tasks. In this problem, tasks can have varying levels of complexity, and agents can have different capabilities and constraints. The goal of MATA is to optimize the allocation of tasks to agents in order to achieve a desired performance or outcome. This problem

can be found in several fields of MAS, such as: reconnaissance, unmanned search and rescue missions logistics, autonomous exploration etc. [10].

There are various methods for solving the MATA problem, some of which involve communication among the agents and collective behavior, while others rely on centralized decision-making by a main controller. One approach to solving the MATA problem is to employ a distributed version of the Hungarian algorithm, which calculates the optimal task assignments for the agents by determining which robot will execute the task at the lowest cost, with the aim of minimizing the overall cost of task execution [11][12]. An alternative approach is the Market-Based approach, which is predicated on the principle of auctions. This methodology utilizes negotiation processes that are rooted in market theory, with the goal of optimizing an objective function that is based on the utility of robots for performing specific tasks [13]. This method necessitates collaboration among the robots which is not present in the robots utilized within this project.

Additional way to solve the MATA problem is by using deep learning. In MAS with cooperating between agents, agents can act automatically and learn how to communicate with other neighboring agents to allocate tasks and share resources by using deep reinforcement learning. Through learning capabilities, agents can be able to reason conveniently, generate an appropriate policy and make a good decision [14].

2.3 Combined Task Allocation and Path Finding Problem

The integration of MATA and MAPF algorithms is a complex endeavor as these problems are classified as NP-hard. One proposed method for addressing this issue is an optimal solution that involves utilizing the MATA model such that each agent can only perform one task at a time and every task only requires one agent and a MAPF algorithm based on the CBS approach. This combination has been tested on a 2-dimensional grid [15]. This study approaches the problem in a basic manner without considering the problem as a spatial one and disregarding the significance of the location of the package, and the quantity of robots and packages in the warehouse setting.

3 Theoretical background

In this chapter, the relevant theoretical background for understanding the research will be presented. An explanation of data structures and complexity, search algorithms, constraint satisfaction problems and the stages in the Hungarian algorithm will be presented.

3.1 Data structure

Data structures are used to organize and store data in a way that is efficient and easy to access. Some common data structures include arrays, linked lists, stacks, queues, trees, and graphs. Each data structure has its own set of characteristics and is used in different situations depending on the requirements of the problem being solved. In this chapter, graph and tree structures will be explained [16] and the complexity class of those structures will be explained too.

3.1.1 Graphs

Graphs are sets of objects with no specifically structured relationship. The object in a graph is called a node or a vertex, and the link between the vertex and the node is called an edge. The formal definition of a graph is:

$G = (V, E)$ is graph whose composed of:

- $V(G)$: a finite, nonempty set of vertices
- $E(G)$: a set of edges

Each node can be connected to one or many nodes in the graph in any way. Graphs can be classified into three types: Undirected, Directed and Weighted as shown in Figure 3-1. In an undirected graph, it is possible to move between vertices that are connected by edges regardless of the direction of those edges. In contrast, in a directed graph, movement between nodes is dictated by the direction of the arrows connecting the nodes. In a weighted graph, each step in a particular direction has a specific weight that impacts the algorithm used to guide movement through the graph.

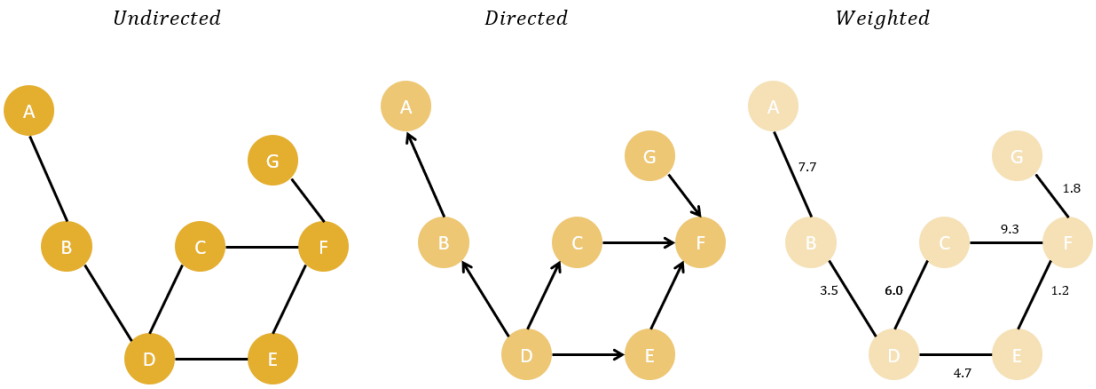


Figure 3-1: Graph Types

3.1.2 Trees

A tree structure is a data structure that consists of a set of nodes organized in a hierarchical manner or equivalently a connected acyclic undirected graph as shown in Figure 3-2. The *root* node is the base of the tree and is typically located at the top of the structure. Each node has one *parent* node and may have multiple *child* nodes that connect with an *edge* to their parent. The children of a single parent node are referred to as *siblings*. A *leaf* node is a node that does not have any children. The *depth of a node* refers to the number of edges between the node and the root node.

Tree structures are commonly used to store and organize data in a hierarchical manner, such as file systems or decision trees. They are efficient at storing and manipulating data but can become unwieldy if the tree becomes too deep or too large.

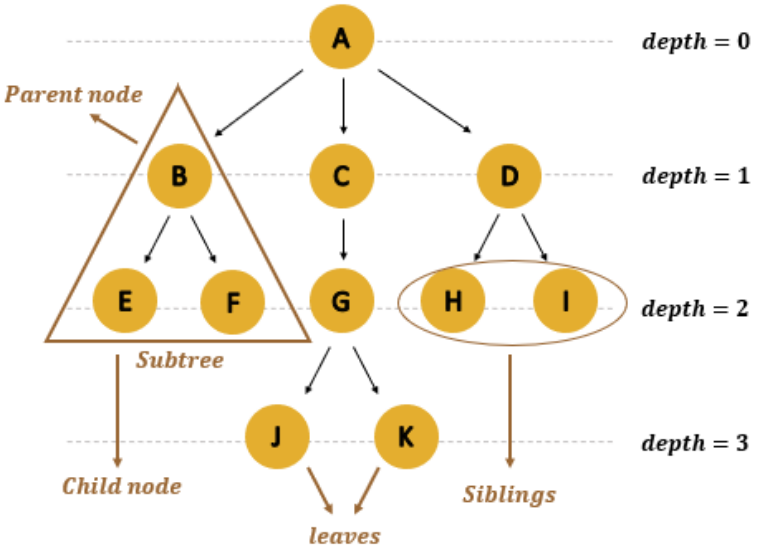


Figure 3-2: Tree Structure

3.2 Complexity

A complexity class is a collection of problems that possess a similar level of complexity. These classes aid researchers in categorizing problems based on the amount of time and space required to both solve and verify solutions. The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer. The space complexity of an algorithm describes how much memory is required for the algorithm to operate [17]. The problem that is present in this research are classified as NP-hard. NP-hard represents problems that are at least as hard as the hardest problems in NP (collection of decision problems that can be solved by a non-deterministic machine in polynomial time). There is no known polynomial time algorithm for solving these problems.

3.3 Search algorithm

To navigate from a starting point to a goal in tree-type and graph-type data structures, search algorithms can be used. This chapter presents the A* (A star) algorithm [18] (more search algorithms are presented in the appendix).

The A* algorithm is a solution to the Single-Source Shortest Path problem for nonnegative edge costs. It's part of a class of Best-First Search (BFS) algorithm those algorithms characterized by expanding the most promising node chosen according to a specified rule. A* determines the optimal path by defining the function $f(n)$

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

Where $g(n)$ is the cost of the path from the start node to node 'n', and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from node 'n' to the goal node. The function $h(n)$ is considered admissible if it is always less than or equal to the actual cost of the lowest-cost path from node n to the goal. If the heuristic function is admissible, the solution to the algorithm will be optimal. There are various methods for designing a heuristic function, and the choice of function can significantly impact the performance of the algorithm. Examples of heuristic functions present in Figure 3-3 those function are Manhattan distance and the Euclidean distance.

Figure 3-3: Euclidean distance and Manhattan distance

3.4 Constraint Satisfaction Problem

Constraint satisfaction problems (CSP) are mathematical problems where one must find states or objects that satisfy a number of constraints or criteria [19]. This problem is defined as a tuple $\langle X, D, C \rangle$ when:

- X is set of variables
- D is a domain of values (i.e., the set of values that the variable can take)
- C is set of constrains

CSP can be solved using various algorithms such as back jumping, forward checking and local search. This type of problems used to model and solve a wide range of problems including scheduling, configuration and optimization problems.

3.5 ABC (Always Better Control) Analysis

The ABC inventory classification method is a systematic approach for categorizing and organizing warehouse products based on their significance, relevance to the organization, economic value, benefits, and rotation generated among other determinants. The objective of this classification method is to prioritize the most critical goods for the company in the warehouse by utilizing the Pareto Principle to segment the goods into three distinct categories (A, B, and C) based on their importance as per the pre-determined criteria [20].

- **Category A products** - These items are characterized by their high turnover rate and strategic importance. They are typically located in the lower areas of the warehouse, with direct and easy access for operators and in close proximity to the shipping docks.
- **Category B products** - These items are characterized by their medium turnover rate. They are typically located in intermediate-height areas of the warehouse, with less direct access than Category A products, but not the least accessible.

- **Category C products** - These items are not considered strategic items. They are characterized by their low turnover rate and are typically located in the highest and least accessible areas of the warehouse, as well as those furthest from the shipping docks.

3.6 Hungarian Algorithm

The Hungarian matching algorithm is a combinatorial optimization algorithm that solves the assignment linear-programming problem in polynomial time. The problem is defined by a $n \times n$ cost matrix, where the i row represents the worker and the j column represents the assignments. The goal is to find an assignment to the workers, such that each task is assigned to one worker and each worker is assigned one task, and that the total cost of assignment is minimum [21].

The steps for implementing the Hungarian algorithm are:

a. Subtract the smallest entry in each row from all other entries in that row.
b. Subtract the smallest entry in each column from all other entries in that column.
c. Draw the fewest number of lines possible on the rows and columns that contain 0 entries.
d. If there are n lines drawn, an optimal assignment of zeros is possible and the algorithm is finished. If the number of lines is less than n, then the optimal number of zeroes is not yet reached and the next step needs to be done.
e. Subtract the smallest entry that is not covered by any line from each row that has not been crossed out and add it to each junction between lines. Return to step 3.

Refer to Figure 3-4 for an example of the implementation of these steps.

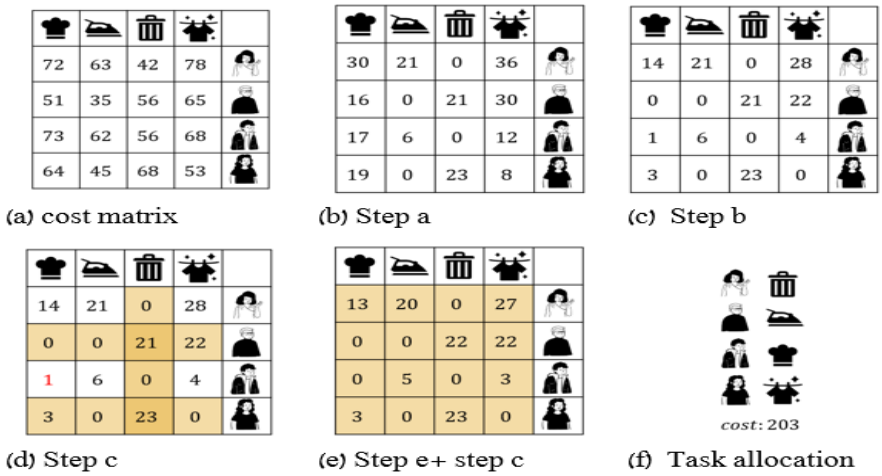


Figure 3-4: Implementing of the Hungarian Algorithm

4 Problem Statement

In recent times, online shopping has become common, and more people order products online. By using robots in logistics warehouses, the shipping process can be automated. To ensure that the robots can perform the required operations inside the warehouse efficiently, they must be assigned tasks while considering all existing tasks and their timing. Route planning for each robot must also be considered to ensure that the routes are the shortest and avoid collisions. The research focuses on combining different principles of existing algorithms to allocate packages and plan routes for each robot in a heterogenic agent group. The solution to the presented problem involves several key steps:

- To conduct a comprehensive literature survey on existing algorithms
- Development of an algorithm based on the literature for heterogeneous agents in three-dimensional environments
- Translate the new algorithm principles into executable code for practical implementation
- Develop a simulation environment for testing and validation
- Perform analysis to assess algorithm efficiency and overall performance

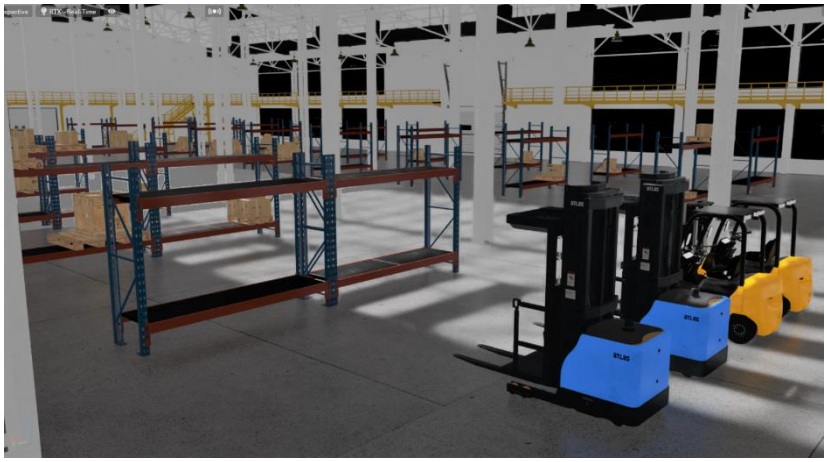


Figure 4-1: Warehouse Simulation with Multi-Agent System in Nvidia-Isaac Sim

5 Multi-Agent Storage and Delivery (MASD)

To ensure efficient and timely collection, storage, and delivery of packages, it is necessary to divide the task into several sub-steps. This includes determining the appropriate storage location for each package within the warehouse, assigning agents to transport the packages to their destinations, and determining the optimal routes for the agents to take in the warehouse, while ensuring that there are no conflicts between agents. In this chapter, the Multi-Agent Storage and Delivery (MASD) algorithm which has been developed as part of this project will be presented. The MASD algorithm is based on the principles of the Hungarian algorithm and CBS and incorporates their key concepts and methodologies.

5.1 Assumptions

- Knowledge of the warehouse structure - two-story shelves arranged in rows and columns.
- Heterogeneous autonomous agents in the warehouse with different capabilities. One type can access only lower shelves, while the other type can access both lower and upper levels.
- Knowledge of packages arriving the next day - their arrival and departure times.
- Knowledge of storage locations within the warehouse, including occupancy status.

5.2 MASD Algorithm Features

This chapter presents the enhancements made to the MASD algorithm and discusses how its features are tailored to address the specific problem at hand.

Firstly, while the CBS algorithm is typically designed for two-dimensional grid environments, the MASD algorithm is specifically tailored to handle the complexities of three-dimensional environments. This adaptation allows for a more accurate and realistic representation of the warehouse setting.

Moreover, considering the heterogeneous nature of the agents involved, the MASD algorithm takes into account that not all agents can perform every task. This aspect poses challenges to the task allocation process, which the MASD algorithm effectively addresses.

Another significant distinction is observed in the continuity of the problem. In CBS problems, each agent has an initial state and goal, with no seamless transition between consecutive tasks. In contrast, the MASD algorithm maintains a continuous flow by assigning new tasks to robots immediately after the completion of their previous tasks, based on their positions in the graph. This ensures a smoother and more efficient task execution process.

In addition, one notable distinction is that in the MASD algorithm, agents return to their starting points instead of remaining at their goal locations, as seen in CBS. This design choice prevents an agent from blocking access to a shelf while waiting for further tasks. By returning to the starting point, the agent ensures that other agents can access the shelves and continue their operations efficiently. This approach enhances the overall accessibility and availability of shelves within the warehouse environment, preventing potential bottlenecks and optimizing the utilization of resources.

5.3 Problem Statement

The input for the MASD problem is $\langle G, \mathcal{A}, P, \phi \rangle$ where:

- $G = (V, E)$ - an undirected graph where V is the vertices and E is the edges.
- $\mathcal{A} = (A_0, \ldots, A_n)$ - groups of n agents $\mathcal{A} = (A_0, \ldots, A_n)$ where each group contains i agents $A_n = (a_0, \ldots, a_i)$.
- $P = (P_0, \ldots, P_k)$ - set of packages that belong to the logistic system of the warehouse i.e. stored or should be stored in the next 24 hours which k is the number of the packages
- ϕ - the storage places on the shelves.

5.4 Algorithm architecture

This chapter provides a comprehensive overview of the algorithm's architecture. As shown in figure 5-1, the initial step involves sorting by time all incoming and outgoing packages scheduled to ship today. Within each specific time interval, a cost calculation is performed to determine the feasibility for each agent to undertake the assigned tasks (see chapter 5.4.2). Subsequently, tasks are allocated to agents using a weight matrix and the Hungarian algorithm. Finally, a collision-free route is meticulously planned for each agent to effectively execute their respective assigned tasks.

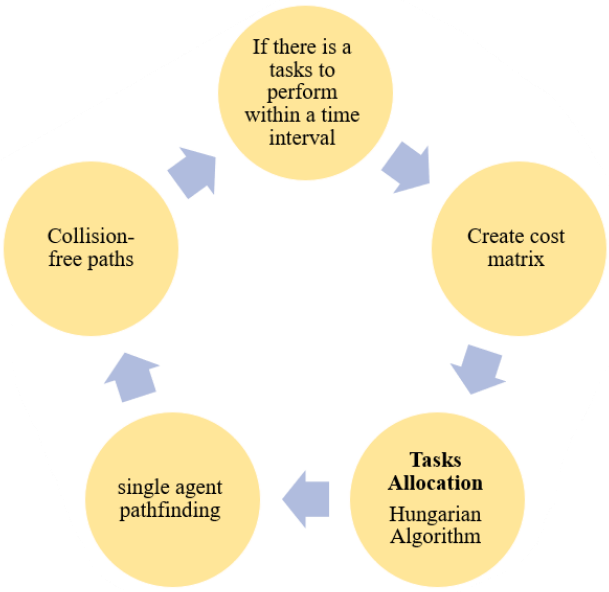


Figure 5-1: Steps of the MASD Algorithm

5.4.1 Package Storage Management System

ABC analysis is utilized to determine the appropriate storage location for packages in the warehouse. This methodology involves categorizing the storage locations within the warehouse into three groups, based on their storage time. As the storage locations are situated closer to the unloading and shipping point, the priority of the location increases. Products that are expected to have a shorter storage time are strategically placed in these higher-priority locations.

ALGORITHM 6-1: MANAGE STORAGE_AREA OF PACKAGES

Input: *Package_list // list of package class which define by Arrival_time, Exit_time and Storage_Area*

Output: *Storage_area // The area location of each package defined*

```
For Package in Package_list
    Storage_time ← package.Arrival_time– package.Exit_time
    If  0 < storage_time ≤ A_lim
        Package.Storage_Area ← 'A'
    Elseif 0 < storage_time ≤ B_lim
        Package.Storage_Area ← 'B'
    Else
        Package.Storage_Area ← 'C'
    End if
End for
```

Algorithm 6-1 illustrates the implementation of ABC analysis. As you can see, by calculating the store time of each package, the category in which the package will be stored is determined.

After categorization, the selection of a particular location within a category is done arbitrarily and it is influenced by the available space and the type of robot that will perform the task. If a category is fully occupied, the package will be placed in the next lower category.

The current division into three categories is relatively rough, the packages can be divided into a larger number of categories if needed to allow for better distribution within the warehouse.

5.4.2 Task Allocation

This chapter provides a comprehensive overview of the methodology employed for Multi-Agent Task Allocation (MATA) within the context of this project. The task assignment for the presented problem can be categorized into two distinct parts: transporting packages from the shipping dock to the shelves and transporting packages from the shelves to the shipping dock. The priority of package transportation is determined based on their available time, giving higher precedence to packages that became available earlier.

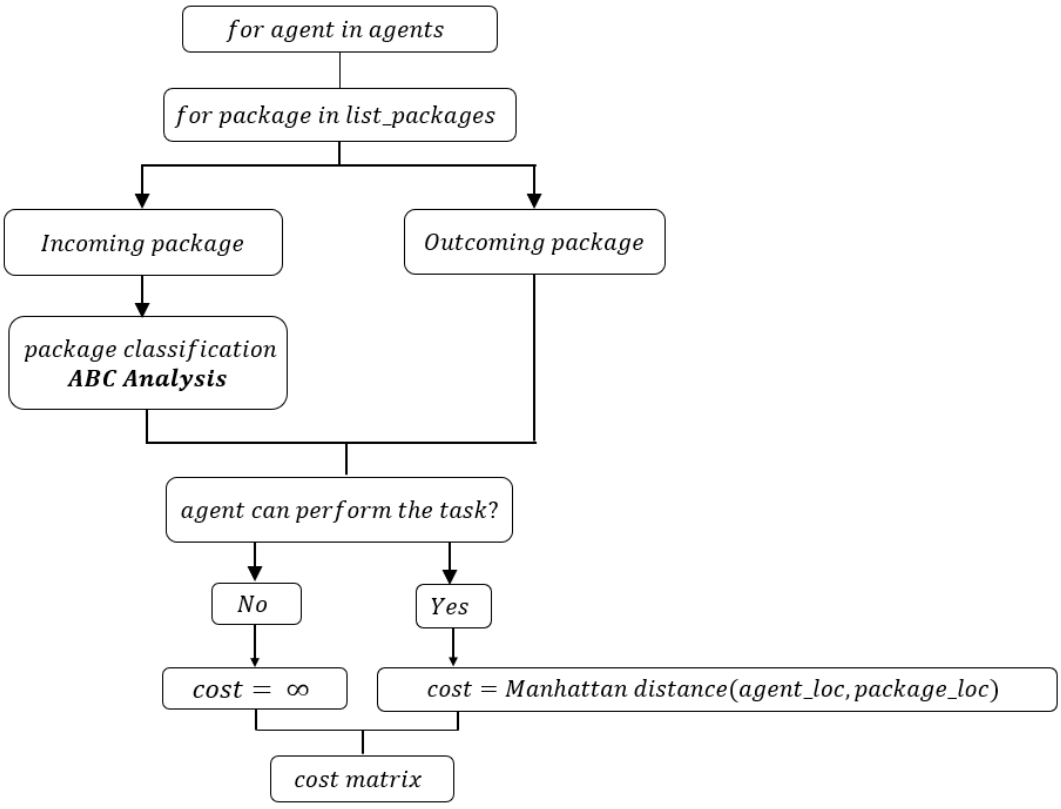


Figure 5-2: Developing a Cost Matrix

During the task assignment phase, the Hungarian algorithm is utilized (as shown in chapter 3.6), which necessitates assigning a price to each agent for executing a specific task. Figure 5-2 depicts the arrangement used to generate the cost matrix. To determine the price for an agent performing a particular task, several considerations are taken into account.

Firstly, it is ascertained whether the package is already stored within the warehouse or if it is an incoming package. For packages already in the warehouse, the destination location for the agent's travel is known. However, in the case of incoming packages, their entry location (i.e., the shipping dock) is known, but the specific location within the warehouse needs to be determined. The location of the package within the warehouse is established using the ABC analysis, which considers the time spent in the warehouse as a crucial factor.

Once the location of the package is determined, the price for performing the task is calculated based on the Manhattan distance, as depicted in formula (2):

$$d = |(x_1 - x_2)| + |(y_1 - y_2)| \quad (2)$$

When (x_1, y_1) is the robot location and (x_2, y_2) is the package location.

In situations where an agent is unable to perform a given task, an infinite price is assigned. Subsequently, all the determined prices are incorporated into the cost matrix, forming the basis for subsequent calculations using the Hungarian algorithm.

In cases where the number of available agents exceeds the number of tasks, dummy tasks with a cost of 0 are created to ensure optimal utilization of all agents. However, it is important to note that the algorithm implemented in this project is not designed to handle situations where the number of available agents is less than the number of tasks. The algorithm is specifically designed to find an optimal assignment for performing all the tasks and does not have a mechanism to address scenarios where it is unable to assign all the tasks. Therefore, the algorithm in this project is not susceptible to such situations.

5.4.3 Multi Agent Path Finding (MAPF)

The MAPF approach in this project is based on the Conflict-Based Search (CBS) algorithm, discussed in chapter 2-1. Unlike the CBS algorithm, which focuses on moving agents from an initial state to a single destination state, the approach adopted in this project involves agents navigating through multiple goal points to ensure efficient package movement and reach the intended destination within the warehouse.

Specifically, for each assigned package, an agent's route is planned from its starting point to the package's initial location (either a shelf or a package arrival dock), then to the package's destination location, and finally back to the agent's initial location.

The reason why agents return to their starting points at the end of their routes is to prevent situations where one agent obstructs another from reaching a destination point. For instance, consider a scenario where both levels of a shelf are empty. If the first agent remains in front of the shelf after placing a package on the bottom level, the second agent attempting to place a package on the top level will be unable to access it. Returning to the starting point ensures smooth access for other agents.

The planning of these three routes is accomplished using the A* algorithm as shown in Algorithm 6-2. Algorithm 6-2 describes the following operations: the algorithm initializes by setting the starting point and the goal point. Then, it creates an open list to keep track of the nodes to be explored and a closed list to store the nodes that have been visited. Next, the algorithm calculates the heuristic value for each node, which estimates the cost to reach the goal. The heuristic function is calculated according to the Manhattan distance. It evaluates the cost from the starting node to each neighboring node and updates the parent node and total cost for each neighbor. The algorithm then selects the node with the lowest total cost from the open set and explores its neighbors. At each step, it checks if the goal node has been reached. Once the goal node is reached, the algorithm reconstructs the optimal path from the start to the goal using the parent information stored for each node.

ALGORITHM 6-2: ***A* ALGORITHM***

```
    Input: start node s, goal node t and cost function f(n)=g(n)+h(n)
    Output: Cost-optimal path from s to t, or Ø if no such path exists
1   Closed ← Ø
2   Open ← {s}
3   While (open ≠ Ø)
4       current node = node with minimum f(u) from open
5       Remove current node from open
6       Insert current node into closed
7       If (current node == t) return path(current node)
8       For each neighboring node of the current node:
9           If (neighbor == obstacle || neighbor in Closed) skip it
10          f(neighbor) = g(neighbor) + h(neighbor)
11          Neighbor's parent ← current node //keep track of the algorithm's path
12          Open.append(neighbor)
13      End For
14  End While return No Path Found
```

The path planning in Algorithm 6-2 is individual for each agent and collisions are not considered.

After individually planning the route for each agent, collision detection is performed. The collision check follows the order of the packets, with the agent assigned to the first packet in time among the set of packets processed by the Hungarian algorithm being the first to undergo collision checking. The complete route of this agent is compared with the routes of other agents whose routes have already been planned for the same time. If a collision is detected, alternative routes or waiting in place are considered for that specific agent only, while the previously planned routes remain unchanged. The collision prevention algorithm is detailed in Algorithm 6-3.

ALGORITHM 6 - 3: ***HIGH LEVEL OF CBS***

Input: *MAPF instance*
Root.constraints = Ø
Root.solution = find individual path by low level()
Root.cost = h(Root.solution) || *the heuristic cost base on the heuristic function*
Insert Root to OPEN
While *OPEN not empty* ***do***
    P ← best node from OPEN \\ lowest *solution cost*
    Validate the paths in P until a conflict occurs.
    if *P has no conflict* ***then return*** *P.solution* || *P is goal*
    C ← first conflict (a_i, a_j, v, t) in P || *a_i - agent i ; a_j - agent j ; v - vertex ; t - time*
    For *each agent a_i in C* ***do***
        A ← new node
        A.constriants ← P.constriants + (a_i , s, t)
        A.solution ← P.solution.
        Update A.solution by invoking low-level(a_i)
        A.cost = h(A.solution)
        Insert A to OPEN
    End For
End While

5.5 Performance Analysis

The algorithm is subjected to analysis in terms of optimality, completeness, and complexity. Firstly, the optimality aspect is examined to evaluate the extent to which the algorithm can produce optimal solutions. This involves assessing whether the algorithm guarantees to find the best possible solution according to a given objective function or criteria.

Secondly, the completeness of the algorithm is considered. Completeness refers to the algorithm's ability to find a solution if one exists. It is important to determine whether the algorithm will always terminate and provide a valid solution, or if it may fail to find a solution in certain scenarios. If the algorithm is found to be incomplete, it is necessary to identify the specific conditions or cases in which it fails to find a solution.

Lastly, the complexity of the algorithm is analyzed, focusing on its computational efficiency and resource requirements. This includes assessing the time and space complexity of the algorithm, taking into account factors such as the size of the problem instance and the available computational resources. Understanding the algorithm's complexity aids in evaluating its scalability and suitability for practical implementation.

By conducting a comprehensive analysis of the algorithm in terms of optimality, completeness, and complexity, valuable insights can be gained regarding its performance limitations and potential areas for improvement.

Claim: The algorithm is Suboptimal
Proof: Although the algorithm is base on optimal algorithms, certain sub-processes within it may result in non-optimal paths for the agents from their first task to the last.

The algorithm's sub-optimality stems from several factors. For example, agents upon completing their missions, return to their starting points instead of strategically positioning themselves within the warehouse also they do not check if there is an open task to perform before they return to their initial location. Additionally, the algorithm assumes limited shelf accessibility from a single side, disregarding potential efficiencies that could be achieved if agents could approach shelves from both sides. Moreover, the algorithm overlooks the opportunity to optimize task assignment and resource allocation based on the current inventory of packages in the warehouse. For instance, if there are available agents and a package scheduled for departure the following day, relocating the package closer to the shipping dock can enhance overall efficiency by reducing traveling times.

Claim: The algorithm is Complete

Proof: Every package will successfully reach its destination.

(1) Task Assignment: The algorithm ensures the complete assignment of tasks to agents within the system. Through its design, the algorithm follows a systematic approach to task allocation, ensuring that every package is assigned to an available agent for execution. Even if there are no agents initially available, the algorithm accounts for future agent availability, ensuring that all packages will eventually be assigned to agents.

(2) Route Planning: The algorithm's route planning is based on a conflict-based search (CBS) algorithm, known for its optimality in finding feasible routes. By leveraging CBS principles, the algorithm guarantees the completeness of route planning. If a feasible solution exists, the algorithm will find it, ensuring that each agent is provided with a route to perform their assigned tasks.

Considering these aspects, it can be concluded that the algorithm is complete. It guarantees the assignment of tasks to agents, accounts for future agent availability, and ensures the availability of feasible routes for task execution. Therefore, every package within the system will be successfully assigned to an agent, and agents will be provided with routes to perform their tasks.

Claim: the complexity of the algorithm in the worst- case is exponential in the number of agents and packages

Proof:

(1) The Hungarian algorithm has a time complexity of $O(n^3)$, where n represents the size of the matrix on which the algorithm operates.

(2) The complexity of the CBS algorithm depends on several factors, such as the number of agents, the size of the search space, and the complexity of the individual agent's motion model. The worst-case time complexity of the CBS algorithm is exponential in the number of agents. This means that as the number of agents increases, the time required to find a solution grows exponentially.

The claim that the MASD algorithm's complexity is exponential in the number of agents is supported by the fact that the time complexity of CBS is higher than that of the Hungarian algorithm. Consequently, the MASD algorithm inherits the exponential complexity from CBS.

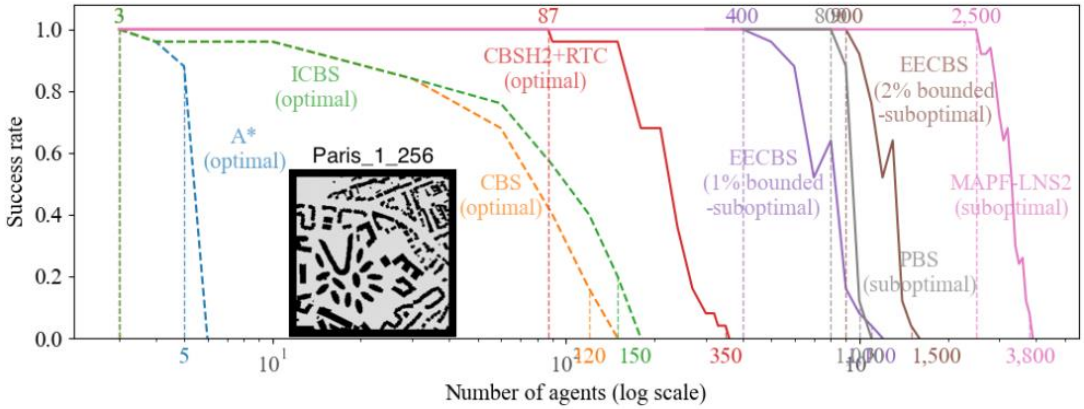


Figure 5-3: Number of agents by success rate for runtime limit of 1 minute [22]

It is important to note that while the worst-case time complexity of the CBS algorithm is exponential in the number of agents, there have been notable improvements to the algorithm. As depicted in Figure 5-3, these enhancements enable the algorithm to handle thousands of agents. Therefore, it is possible to improve the complexity of the MASD algorithm by leveraging these advancements.

The proof that the algorithm's complexity is exponential in the number of packets will be demonstrated in the analysis of the algorithm's results.

6 MASD Algorithm Implementation

In this chapter, the code structure that implements the MASD algorithm will be presented. The code consists of approximately 700 lines, and its structure is depicted in Figure 6-1.

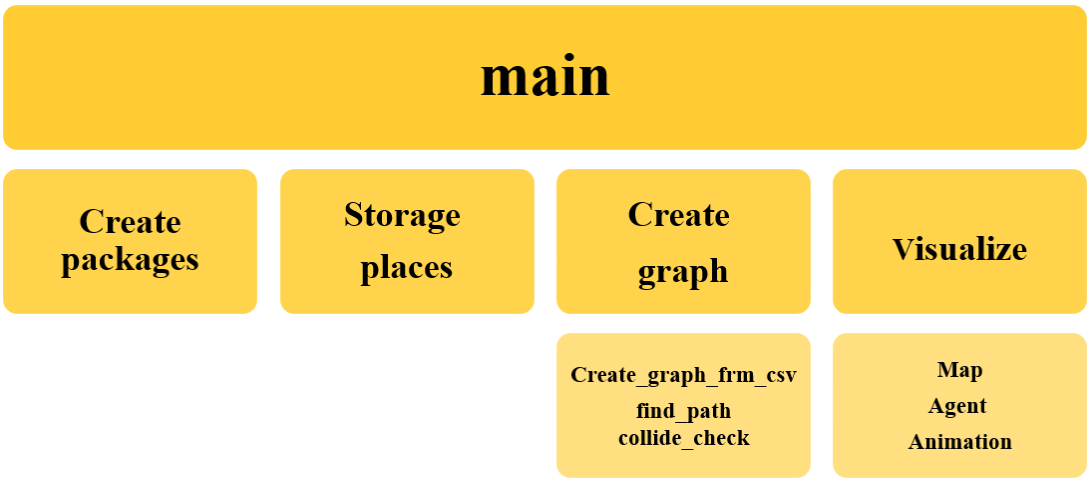


Figure 6-1: Code structure

6.1 Main

This code is a simulation of a warehouse management system. It starts by importing the necessary modules and defining some variables such as: the number of packages, shelves, rows of shelves, the current date, and the number of agents. It then creates a list of agents based on the given number and their capabilities. The code proceeds to create a graph from a CSV file representing the map of the warehouse.

Next, the code creates a list of storage locations in the warehouse, categorized into three types: A, B, and C. It also creates a list of old packages that are already in the warehouse and updates their location data. Then, it filters packages that either arrive or leave the warehouse on the current date and sorts them by their time of arrival or departure.

The code divides the day into 30-minute intervals and processes packages in each interval. For incoming packages, it plans a path for an agent to pick up the package from an entry point and allocates a storage location for it. The agent follows the path to the storage location and returns to its initial position afterwards. For outgoing packages, the code plans a path for an agent to retrieve the package from its storage location, release the storage location, and deliver the package to an exit point. The agent then returns to its initial position.

After processing the packages at each interval, the code converts the agent paths to a format suitable for visualization. It then creates a graphical animation of warehouse and agent movements using the Pygame library. The animation displays the warehouse map, agent paths, and package movements throughout the day.

Finally, the code outputs the agent data in a specific format and displays the animation in a Pygame window.

6.2 Create packages

In this code section, packages are generated randomly based on predefined conditions. The purpose of generating package information in this manner is due to the absence of an actual repository from which to obtain package data. The following steps outline the process of creating k number of packages. Within each designated time frame for package creation, the shipping date and time are determined randomly as follows:

- 0.3k packages are generated with their arrival date set to today, accompanied by a randomly assigned arrival time. These packages are scheduled to depart from the warehouse within the next 30 days.
- 0.3k packages are created, with their arrival dates falling within the 30-day period prior to today. These packages are planned for departure from the warehouse today, with the departure time determined randomly.
- 0.4k packages are generated, with their arrival dates set within the 14-day period preceding today. The departure dates for these packages are scheduled for the next 14 days.

It is worth noting that this approach incorporates randomization in both the shipping dates and shipping times of the packages, ensuring variability and realistic simulations within each time frame.

6.3 Storage places

This code subsection provides information about the storage locations in the warehouse, specifically categorized into three sections: A, B, and C. These sections represent a division of storage locations within the warehouse based on residence times. Area A contains packages with the shortest storage time, while area C is designated for packages with the longest residence time. The code includes functions to calculate the number of rows per section, create storage spaces within the warehouse, and allocate packages based on their arrival and departure times. The functions included in this section are:

- The RowPerSection function determines the number of shelf rows in each section by taking the total number of rows in the warehouse as input and returning the corresponding row counts for sections A, B, and C.
- The storage_spaces function generates storage locations within the warehouse. It creates separate lists (A_list, B_list, C_list) to store the storage locations for each section. The number of rows per section is calculated using the RowPerSection function. The function combines row letters and numbers to create unique storage location identifiers, which are then added to the respective section lists.
- The AllocateOldPackages function is responsible for allocating storage spaces for packages that arrived at the warehouse prior to the current day (referred to as "old packages"). The function determines the time difference between the package's entry and exit times. Based on this difference, packages are assigned to the appropriate storage sections (A, B, or C). The storage_id is randomly selected from the list of available places for storage within the corresponding section, and the package is then added to the allocate_list.
- Similarly, the AllocateNewPackages function handles the allocation of new packages. This function considers an additional parameter which is the ability of an agent to access top shelves. The function assigns the package to a suitable storage space based on its residence time in the warehouse and the availability of agents.

These functions collectively facilitate the management and allocation of storage spaces for packages within the warehouse, taking into account different arrival times, residence times, and agent capabilities.

6.4 Create graph

This code section is responsible for creating a graph used for agent pathfinding search and in addition this code contains functions for searching optimal paths for agents. It includes several functions:

create_graph_from_csv: This function takes a CSV file as input and converts it into a graph representation suitable for route searching by using NetworkX library which is utilized for graph creation and route searching. Figure 2 provides an example of the CSV file structure that is required by this function.

	A	B	C	D	E	F	G	H	I
1	0	0	0	0	0	0	0	0	0
2	0	A10	A20	A30	A40	A50	A60	0	0
3	0	1	1	1	1	1	1	0	0
4	0	1	1	1	1	1	1	0	0
5	0	0	0	0	0	0	0	0	0
6	0	B10	B20	B30	B40	B50	B60	0	0
7	0	1	1	1	1	1	1	0	0
8	0	1	1	1	1	1	1	0	0
9	0	0	0	0	0	0	0	0	0
10	0	C10	C20	C30	C40	C50	C60	0	0
11	0	1	1	1	1	1	1	0	0

Figure 6-2: Example of csv file

find_path: The purpose of this function is to find the shortest path for an agent between its designated destination points along the package route. The A* algorithm is employed to determine the optimal route. The implementation of the A* algorithm within this code snippet utilizes the astar_path function provided by the NetworkX library.

collide_check: This function is responsible for detecting potential collisions between two agents. If no collision is detected, the planned route is added to the agent's travel path. However, in case of a collision, the routes of the colliding agents are passed to the find_path_without_collisions function.

find_path_without_collisions: This function aims to find an alternative route for an agent whose initial planned route collides with other agents' routes. The objective is to minimize the cost while determining an alternative path for the agent.

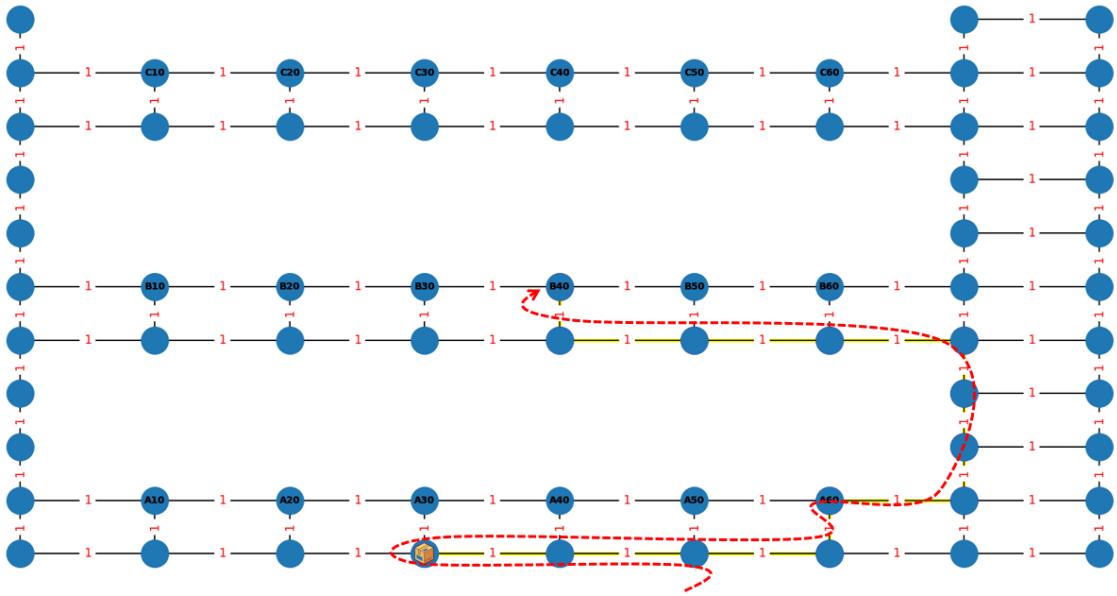


Figure 6-3: Graph Visualization and Agent Path Planning

The visual representation of the graph generated by the code snippet can be seen in Figure 6-3. In this figure, the blue circles represent nodes that the agents can traverse, while the edges indicate the transitions between adjacent blue nodes. The nodes in the graph are labeled with letters and numbers, representing target nodes for storing items on the shelves. The rows are labeled with letters, while the columns are labeled with numbers. When agent intends to place a package on a shelf, he approaches a position in front of the respective shelf and unloads the package. The depicted scenario demonstrates the path finding planning process for an agent: starting from its initial state, moving towards the package arrival point, collecting the package, and finally planning a path towards the package's destination point.

6.5 Visualize

This code is a visualization tool for a multi-agent pathfinding problem. It uses the Pygame library to create a graphical representation of the map and the agents' movements. The code reads a CSV file that contains information about the map and the agents' paths.

- The Map class is responsible for creating the map image from the CSV file. It reads the file and stores the obstacles' positions. The map is drawn on the Pygame window using black rectangles for the obstacles.
- The Agent class represents an agent in the simulation. It parses the agent's path from the input data and stores it as a list of coordinates. The move method updates the agent's position based on the current time and the path. The draw method displays the agent on the Pygame window as a colored circle.
- The Animation class coordinates the main loop of the program. It initializes the map, creates the agents, and handles user input for controlling the animation. The agents' movements are updated based on the elapsed time, and the map and agents are drawn on the window.

7 Simulation

Chapter 6.5 presents a simulation that showcases a multi-agent pathfinding problem. The simulation, developed using Python, offers a visual representation of the problem scenario and demonstrates the movement of agents in a dynamic environment.

The simulation begins by reading map data from a CSV file, which is then used to create a map image. Obstacles within the map are represented by black rectangles. The simulation incorporates multiple agents, each following a predefined path. The agents' movements are smoothly animated based on the provided path data.

To enhance user interaction, the simulation utilizes Pygame's event system. Users can control the animation, adjust the speed of the simulation, and toggle the display of agent IDs.

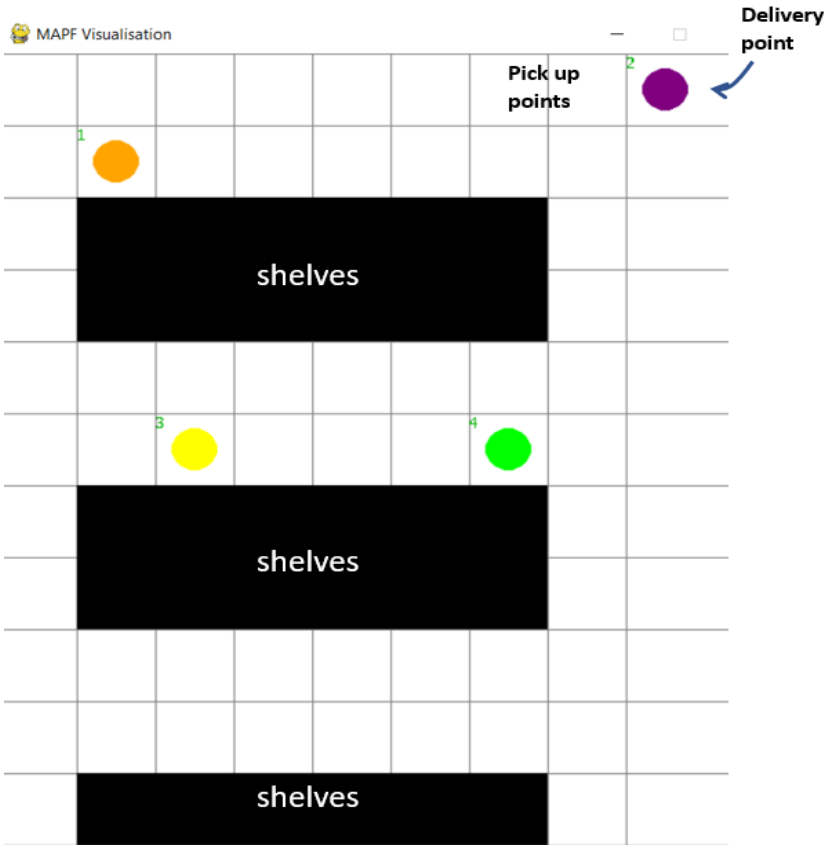


Figure 7-1: Simulation

In Figure 7-1, the visibility of the simulation is shown. While the algorithm is designed for a 3D environment, the simulation is constructed using a 2D mesh. This means that the simulation disregards the z-axis, representing the height of the shelves. Consequently, users may notice that agents can place two packages in the same location, as the simulation lacks vertical distinction.

The network depicted in the figure comprises white squares, which denote accessible nodes for the agents, while the black squares represent shelves. The simulation does not display the initial positions of the agents; the agents begin outside the graph, situated in the upper left corner. There are two marked locations designated as package collection points, where arrived packages are stored, and a single shipping point from which packages depart the warehouse.

Moreover, the agents can only access the shelves from the top row adjacent to them, and not from the bottom row. Essentially, the shelves are only approachable from one side, restricting agent access.

In the figure, the agents are represented by colored circles, with each agent accompanied by a small numerical identifier. These agents possess the ability to navigate freely within the white squares, provided there are no collisions with other agents.

8 Results

In this chapter, the results obtained from running the algorithm will be presented, including a comprehensive example that covers all the steps of the algorithm, ranging from task assignment to path planning and execution, also sensitivity analyses of the algorithm will be conducted to evaluate its robustness and performance under different conditions.

All computations were conducted on a Dell Vostro computer equipped with an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, 2592 Mhz, 6 cores, and 12 logical processors.

8.1 Algorithm demonstration

Figure 8-1 illustrates the initial state for the selected example case. In this case, the first step is to assign agents to tasks, followed by planning the routes for each agent.

The initial state locations are shown in Table 8-1.

Table 8-1: The initial state of the agents and packages

	1	2	3	4
$Agents$	$(0,-1)$	$(1,-1)$	$(2,-1)$	$(3,-1)$
$Packages$	$(7,1)$	$(1,1)$	$(2,1)$	$(2,5)$

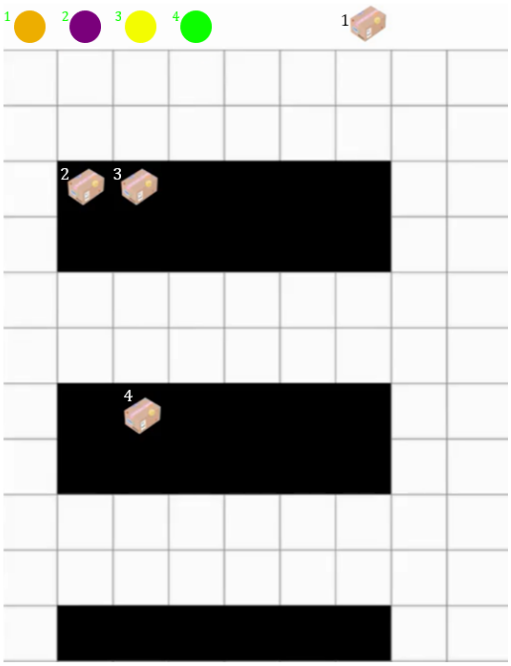


Figure 8-1: The initial state of the example case

In the given case, agents 1 and 2 have the ability to reach the upper shelves, whereas agents 3 and 4 are restricted to the lower shelves. Package 4 is located on the top shelf. The first step is to calculate the cost matrix for the agents. Figure 8-2 displays the resulting cost matrix.

	Package 1	*Package* 2	*Package* 3	*Package* 4
Agent 1	9	3	4	8
Agent 2	8	2	3	7
Agent 3	7	3	2	∞
Agent 4	6	4	3	∞

Figure 8-2: Cost Matrix

After executing all the necessary manipulations based on the Hungarian algorithm, the resulting matrix is as follows:

	Package 1	*Package* 2	*Package* 3	*Package* 4
Agent 1	3	0	1	0
Agent 2	3	0	1	0
Agent 3	2	1	0	∞
Agent 4	0	1	0	∞

Figure 8-3: The cost matrix after implementing the Hungarian algorithm

Based on the resulting matrix, it can be determined that agent 4 will be assigned to package 1, agent 3 will be assigned package 3, and the cost for agents 1 and 2 to perform tasks 2 and 4 is equal. Consequently, the allocation can be done arbitrarily. As a result, agent 1 will be assigned to package 1, while agent 2 will be assigned package 4.

After the tasks have been assigned, the next step is to collect the packages at their designated times. By these times, the routes for the agents are established.

The path for each agent is initially calculated using the A* algorithm without taking into account the paths of other agents. The heuristics for this algorithm are determined using the Manhattan distance. Once the routes are computed, a collision check is performed to identify any potential collisions between agents.

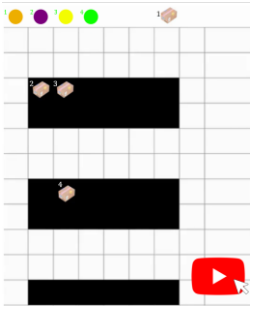


Figure 8-4: Link to YouTube video

As seen in the video linked in Figure 8-4, each agent follows a predetermined path based on their assigned task. A potential collision is evident during the trajectories of Agent 3 and Agent 1. To prevent this scenario, Agent 3 pauses at its current location until Agent 1 passes, ensuring a collision-free path. Once Agent 1 has passed, Agent 3 continues with its assigned task. After completing their respective tasks, all agents return to their starting points before proceeding to perform new tasks.

8.2 Performance Analysis

In this chapter, different parameters and their impact on the algorithm's execution time explored.

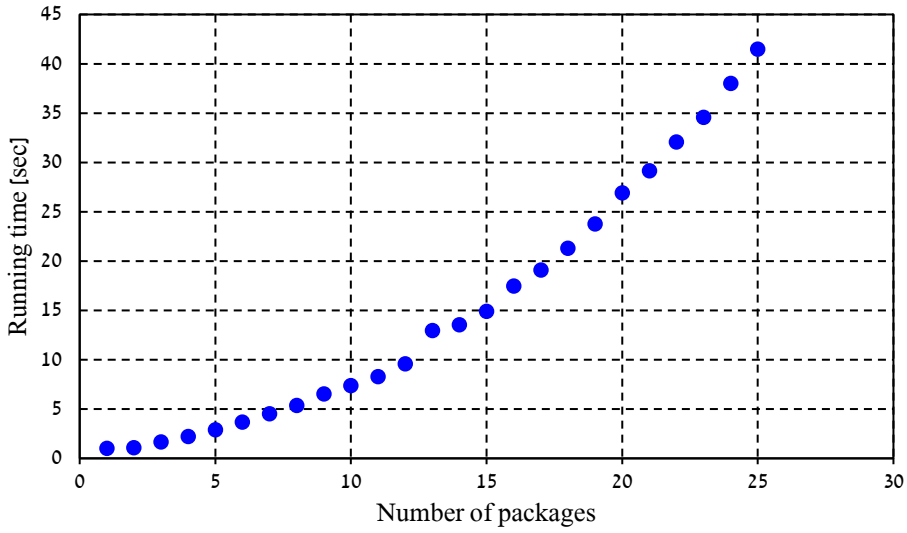


Figure 8-5: The Effect of the Number of Packages on the Running Time

Figure 8-5 depicts the impact of the number of packages on the algorithm's running time. It is evident that as the number of packages in the warehouse increases while keeping the number of agents constant, the running time exhibits exponential growth.

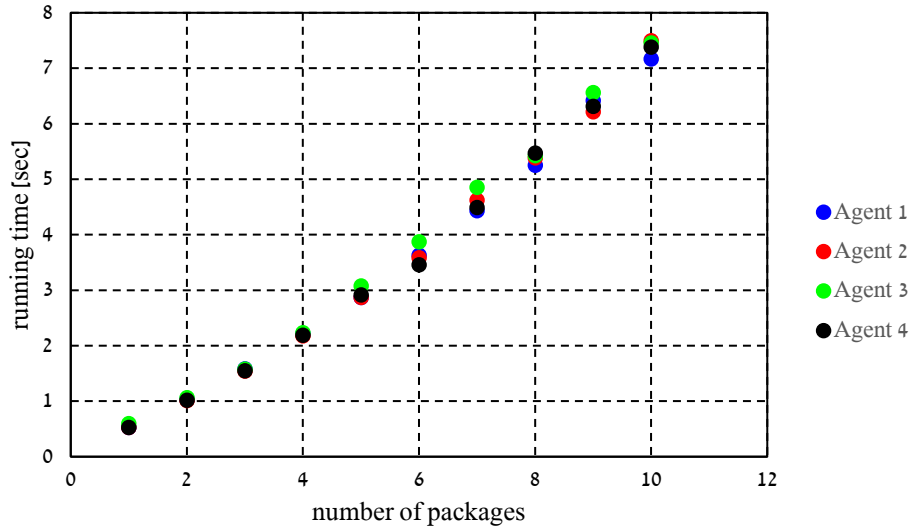


Figure 8-6: The Effect of the Number of Agents on the Running Time

In the given problem, the agents occupy a maximum of 8% of the graph area. Figure 8-6 indicates that the number of agents does not have a significant impact on the running time when considering 1-4 agents in a 9x11 graph. However, it cannot be definitively concluded that this observation applies universally, as larger warehouses with more agents, which occupy a larger percentage of the graph, need to be

examined to make a conclusive determination regarding the effect on the running time.

9 Conclusions

In conclusion, the algorithm offers several advantages in task assignment and route planning for agents within a 3D environment. It effectively handles the optimization of task assignments, collision-free route planning, handling heterogeneous agents, and ensuring continuity of tasks

However, the algorithm has certain limitations that need to be addressed. One drawback is that agents return to their initial state after each task which can lead to inefficient utilization of available time.

Additionally, the algorithm′s optimality is not guaranteed, indicating room for improvement in terms of efficiency.

Furthermore, it is important to note that the algorithm has primarily been tested on small warehouses. Its performance in more complex scenarios, such as dealing with traffic congestion or an increased number of agents, remains uncertain.

Moreover, it is evident that the running time of the algorithm increases exponentially with an increase in the number of packets. This suggests the need for further optimization when handling larger quantities of packages.

In conclusion, while the algorithm offers notable advantages, it also faces certain limitations that should be considered for future enhancements and evaluations in larger-scale scenarios.

10 Summary

In summary, the developed algorithm presents an innovative approach to effectively manage the movement of packages within a warehouse, addressing the challenges of task allocation and multi-agent pathfinding. The algorithm builds upon well-established principles and techniques found in the literature, ensuring a solid foundation for its design. To evaluate the algorithm′s capabilities and performance, a simulation environment was created using Python code. This environment served as a platform for conducting extensive experiments, enabling thorough testing and analysis of the algorithm′s functionality. The results obtained from the experiments and the analysis of the algorithm shed light on the algorithm′s strengths and limitations. Despite being classified as sub-optimal, the algorithm still yielded impressive results

when applied to small warehouses that incorporate heterogeneous agents in three-dimensional environments. This algorithm represents a significant contribution to the field of warehouse management, providing an effective solution for package movement in dynamic and heterogeneous environments.

11 Economic estimation

The estimations of the costs are represented in the next table:

Table 11-1: Economic estimation table

Product	Quantity	Cost per unit [$]	Cost [$]
Powerful computer with graphic processor	1	3,000	**3000**
Working time	96 hours x 10 months	20	**19,200**
Total cost			**22,200**

12 Further research

This study presents the development of an algorithm for optimizing the movement of packages within warehouses, based on a comprehensive literature review. Additionally, a simulation environment was constructed to evaluate the algorithm's performance. The findings indicate that the algorithm can be effectively applied in small warehouses, providing optimal results. This study serves as a foundational infrastructure for further exploration in the following year. Future research endeavors will focus on adapting the algorithm to larger work environments, where challenges such as traffic congestion may arise. Moreover, efforts will be directed toward addressing the constraints identified in the initial version of the algorithm. This includes identifying enhanced waiting areas for agents within the warehouse and enabling access to shelves from both sides. By minimizing these constraints, the algorithm can further optimize package movement and improve overall efficiency within the warehouse. Additionally, the algorithm will be subjected to testing across different warehouse environments, diverse agent types, and various package scenarios to ensure its robustness and applicability.

12.1 Gantt board

Tasks	2023					2024						
	8	9	10	11	12	1	2	3	4	5	6	7
writing code for 3D simulation												
Implementation of the code in the new simulation system												
Increasing the warehouse and examining the code capabilities												
Improved package location												
Access to shelves from both directions												
Improving agent location												
Performing experiments on different environments												
Algorithm sensitivity analysis												
extra time for delays												
writing the thesis												

Figure 12-1: Gantt board

12.2 Task list:

1. Writing code for 3D simulation

   1.1 Research and gather resources on existing 3D simulation frameworks and libraries.

   1.2 Define the required functionality and specifications for the 3D simulation code.

   1.3 Design the architecture and data structures for the simulation code.

2. Implementation of the code in the new simulation system

   2.1 Adapt the existing code to integrate seamlessly with the new simulation system.

   2.2 Perform thorough testing to verify the functionality and compatibility of the implemented code.

   2.3 Fine-tune and optimize the code to ensure efficient and reliable simulation performance.

3. Increasing the warehouse and examining the code capabilities

   3.1 Determine the desired expansion size and dimensions for the warehouse.

   3.2 Conduct extensive testing and evaluation to assess the code's performance and scalability in the larger warehouse setting.

   3.3 Identify any potential issues or limitations that may arise due to the increased warehouse size and make necessary adjustments.

4. Improved package location

4.1 Explore different approaches for optimizing package placement within the warehouse.

4.2 Implement enhancements to the code to improve the efficiency and effectiveness of package location.

4.3 Validate the improvements through testing and performance evaluation, comparing them with the previous package location strategy.

5. Access to shelves from both directions

5.1 Modify the code to incorporate the new functionality for bidirectional shelf access.

5.2 Verify the correct implementation through testing, ensuring agents can effectively access shelves from either side.

6. Improving agent location.

6.1 Implement mechanisms to prevent agent collisions and optimize their paths.

6.2 Validate the improvements through extensive testing and performance evaluation, comparing them to the previous agent location approach.

7. Performing experiments on different environments

7.1 Select a range of diverse warehouse environments to serve as test cases.

7.2 Configure the simulation code to accommodate the chosen environments and their specific characteristics.

7.3 Conduct experiments, executing simulations in each environment and collecting relevant data.

7.4 Analyze and interpret the results to gain insights into the algorithm's performance across different environments.

8. Algorithm sensitivity analysis

8.1 Execute the sensitivity analysis, observing and documenting the algorithm's performance under different conditions.

8.2 Analyze the collected data to determine the algorithm's robustness and identify areas for further improvement.

9. Writing the thesis

13 Supervisor review

This project report sets the foundation for warehouse automation. It introduces an innovative approach in the field of logistics by addressing the problem from the perspective of package paths rather than robot paths. The project successfully achieved its objectives as outlined in the final project, and its outcomes will serve as the foundation for the Master's thesis. Throughout this endeavor, Stav demonstrated significant progress and adhered to her research plan with commendable skill and accomplishments. I deeply appreciate her abilities and achievements thus far.

14 Bibliography

[1] R. Stern *et al.*, "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks," vol. 10, no. 1, pp. 151–158, Sep. 2021, doi: 10.1609/socs.v10i1.18510.

[2] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. K. S. Kumar, and S. Koenig, "Lifelong Multi-Agent Path Finding in Large-Scale Warehouses," vol. 35, no. 13, pp. 11272–11281, May 2021, doi: 10.1609/aaai.v35i13.17344

[3] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," vol. 219, pp. 40–66, Feb. 2015, doi: 10.1016/j.artint.2014.11.006.

[4] A. Felner *et al.*, "Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding," vol. 28, pp. 83–87, Jun. 2018, doi: 10.1609/icaps.v28i1.13883.

[5] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. K. S. Kumar, and S. Koenig, "Lifelong Multi-Agent Path Finding in Large-Scale Warehouses," vol. 35, no. 13, pp. 11272–11281, May 2021, doi: 10.1609/aaai.v35i13.17344

[6] T. Huang, S. Koenig, and B. Dilkina, "Learning to Resolve Conflicts for Multi-Agent Path Finding with Conflict-Based Search," vol. 35, no. 13, pp. 11246–11253, May 2021, doi: 10.1609/aaai.v35i13.17341.

[7] J. Li, W. Ruml, and S. Koenig, "EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding," vol. 35, no. 14, pp. 12353–12362, May 2021, doi: 10.1609/aaai.v35i14.17466.

[8] G. Sartoretti *et al.*, "PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning," vol. 4, no. 3, pp. 2378–2385, Jul. 2019, doi: 10.1109/LRA.2019.2903261.

[9] M. Damani, Z. Luo, E. Wenzel, and G. Sartoretti, "PRIMAL 2: Pathfinding Via Reinforcement and Imitation Multi-Agent Learning - Lifelong," vol. 6, no. 2, pp. 2666–2673, Apr. 2021, doi: 10.1109/LRA.2021.3062803.

[10] F. Zitouni, S. Harous, and R. Maamri, "A Distributed Approach to the Multi-Robot Task Allocation Problem Using the Consensus-Based Bundle Algorithm and Ant Colony System," vol. 8, pp. 27479–27494, 2020, doi: 10.1109/ACCESS.2020.2971585.

[11] S. Giordani, M. Lujak, and F. Martinelli, "A Distributed Algorithm for the Multi-Robot Task Allocation Problem." Springer Berlin Heidelberg, p. 721, 2010, doi: 10.1007/978-3-642-13022-9_72

[12] H. W. Kuhn, ‘‘The hungarian method for the assignment problem,’’ Naval Res. Logistics Quart., vol. 2, nos. 1–2, pp. 83–97, 1955.

[13] A. Khamis, A. Hussein, and A. Elmogy, “Multi-robot Task Allocation: A Review of the State-of-the-Art,” vol. 604. Springer International Publishing, p. 31, 2015, doi: 10.1007/978-3-319-18299-5_2.

[14] D. Ben Noureddine, A. Gharbi, and S. Ben Ahmed, “Multi-agent Deep Reinforcement Learning for Task Allocation in Dynamic Environment.” SCITEPRESS - Science and Technology Publications, 2017, doi: 10.5220/0006393400170026.

[15] C. Henkel, J. Abbenseth, and M. Toussaint, “An Optimal Algorithm to Solve the Combined Task Allocation and Path Finding Problem,” Nov. 2019, pp. 4140–4146, doi: 10.1109/IROS40897.2019.8968096.

[16] W. Smith, *Everyday data structures*, 1st ed. Birmingham, England ; Packt, 2017.

[17] Umng, 2022, " Types of Complexity Classes | P, NP, CoNP, NP hard and NP complete" https://www.geeksforgeeks.org/types-of-complexity-classes-p-np-conp-np-hard-and-np-complete/

[18] S. Edelkamp and S. Schrödl, *Heuristic search : theory and applications*. Amsterdam: Morgan Kaufmann, 2012.

[19] Milos Simic, 2022, How to Solve Constraint Satisfaction Problems https://www.baeldung.com/cs/csp

[20] Abby Jenkins, 2020, ABC Analysis in Inventory Management: Benefits & Best Practices https://www.netsuite.com/portal/resource/articles/inventory-management/abc-inventory-analysis.shtml

[21] Lei Mao, 2021, " Hungarian Matching Algorithm" https://leimao.github.io/blog/Hungarian-Matching-Algorithm/

[22] Foundations of Multi-Agent Path Finding, Jiaoyang Li, https://jiaoyangli.me/research/mapf/

15 Appendix

15.1 Search Algorithm

15.1.1 Breadth First Search (BFS)

Breadth First Search algorithm is used to locate a path in a graph or tree data structure. It begins at a specific starting point at graph or at the root of the tree and visits all neighboring nodes of the selected node. It then proceeds to visit all other nodes, proceeding level by level through the given structure as shown in Figure 13-1. The algorithm terminates when all nodes have been traversed and marked.

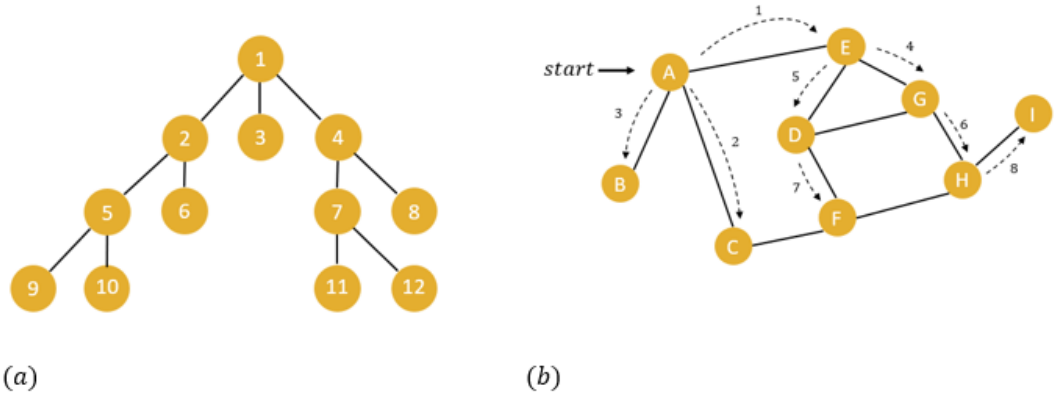


Figure 15-1: BFS algorithm search order in structure: (a) tree (b) graph

One advantage of this algorithm is that it can find the optimal route if there are multiple paths leading to the goal. However, if the route to the goal is unique , the efficiency of the algorithm may not be as high because it will continue to traverse and mark all nodes even after the goal has been reached.

15.1.2 Depth First Search (DFS)

The Depth First Search (DFS) algorithm is a method used to locate a path in a graph or tree data structure. It begins at a specific starting point at graph or at the root of the tree and explores branches as far as possible before backtracking. There are two ways to implement the DFS algorithm: using a last in first out (LIFO) stack or using recursion.

In the stack implementation, the algorithm begins at the starting node and adds it to a stack. It then checks if the current node is the goal. If it is not, the algorithm adds the children of the current node to the stack and marks them as visited. The algorithm then repeats this process, checking the node at the head of the stack for being the goal. If it is not, the algorithm adds its children to the stack and marks them as visited. This process continues until the stack is empty and all the node are marked. An example to the stack implementation shown in Figure 13-2.

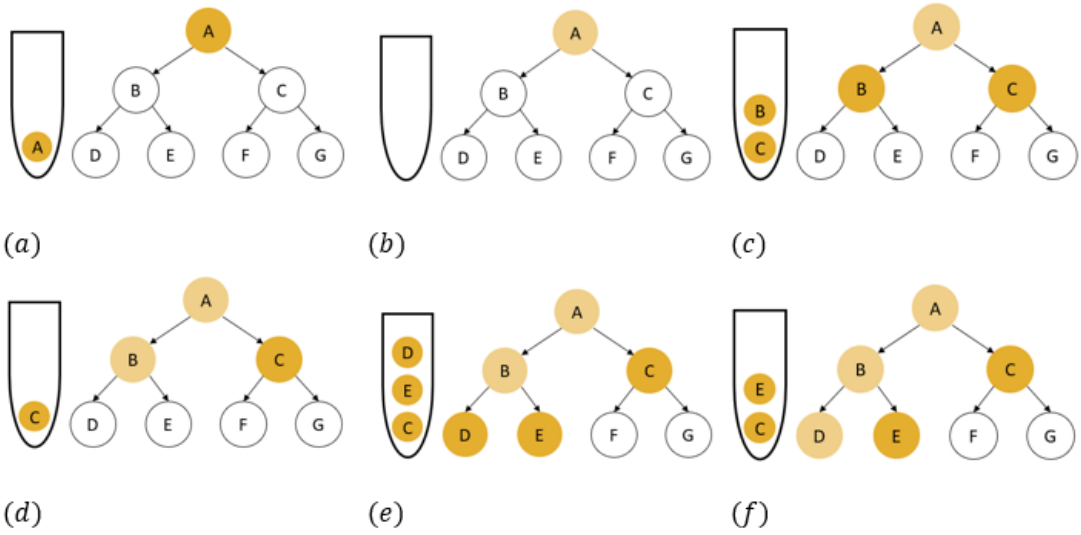


Figure 15-2: Steps description in DFS algorithm at stack implementation

In the recursive implementation, the algorithm begins at the root of the tree and traverses to the depth of the tree until it reaches a leaf node. It then backtracks to the parent of the leaf node and checks if it has any more children that have not been visited. The algorithm continues this process until all nodes have been traversed. An example to the recursive implementation shown in Figure 13-3.

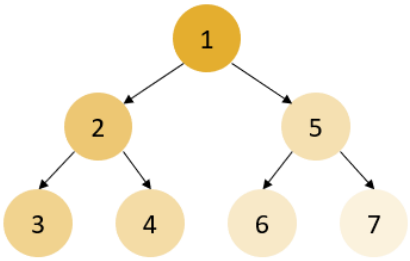


Figure 15-3: Steps description in DFS algorithm at recursive implementation

DFS algorithm is simple and easy to implement, but it may not be suitable for traversing trees that are infinitely large or impractical to completely traverse due to its exhaustive nature

15.1.3 complexity

Such as present in chapter 2.2 there are more complexity classes that exist. The difficulty level of those class shown in Figure 13-4:

- P – It is the collection of decision problems that can be solved by a deterministic machine in polynomial time.

A deterministic problem is a problem where the outcome or solution is uniquely determined by the input, and the same input will always produce the same output (e.g. equation).

- NP - It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.
  A non-deterministic problem is a problem where the outcome or solution is not uniquely determined by the input and multiple solutions or outcomes can be produced for the same input (e.g. the Traveling Salesman Problem (TSP)).
- NP-complete - it is a subset of NP-hard problems and represents the hardest problems in NP. Any NP-complete problem can be transformed into any other NP-complete problem in polynomial time.

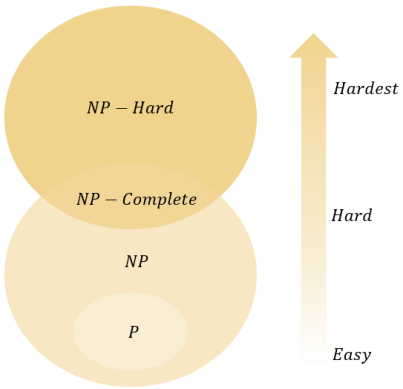


Figure 15-4: Problem classification

15.2 Code

The code snippets for this algorithm can be found on GitHub, providing a detailed view of the implementation.

תקציר

בשנים האחרונות תחום הלוגיסטיקה התפתח והתקדם בצורה משמעותית. התקדמות זו העלתה את הצורך לייעל את תהליך השילוח בין היצרן לצרכן. אחת הדרכים לייעל את התהליך היא לבצע אוטומציה של תהליכי משנה לאורך מסלול החבילה. אחד השלבים במחסן לוגיסטי המשמש לאחסון, שניתן לבצע בו אוטומציה, הוא הנתיב שהחבילה עוברת בתוך המחסן. בשלב זה אוספים את החבילה עם ההגעה למחסן ומניחים אותה על מדף ספציפי. כשמגיע הזמן משלוח החבילה, היא נאספת למשלוח. פרויקט זה, מתמקד באוטומציה של שלב זה.

המטרה העיקרית של פרויקט זה הינה ליעל תנועת חבילות בתוך מחסן באמצעות מערכת הטרוגנית מרובות סוכנים, כלומר, מערכת הכוללת סוגים שונים של רובוטים. מטרה זו תושג על ידי פיתוח אלגוריתם חדש הנשען על עקרונות של אלגוריתמים קיימים כגון: האלגוריתם ההונגרי עבור בעיית הקצאת משימות למספר סוכנים, וחיפוש מבוסס-קונפליקטים עבור מציאת מסלול עבור מערכות מרובות סוכנים. אלגוריתמים אלו נבדקו באופן עצמאי והוכחו כיעילים אך טרם שולבו ונבדקו עם סוכנים הטרוגניים בסביבה תלת ממדית. האלגוריתם שפותח יוערך בסביבת סימולציה.

בפרויקט זה תבוצע סקירה של הספרות הרלוונטית והכרה עם היסודות התיאורטיים הקשורים לתחומי הקצאת משימות ותכנון מסלולים לקבוצות סוכנים. כמו כן, הדוח מספק הסבר מקיף עבור האלגוריתם החדש שנכתב ומציג את החידושים בו וכן את העקרונות עליהם הוא מתבסס. בנוסף , תוצג סביבת סימולציה ייעודית אשר פותחה ב-Python לשם בחינת יכולות האלגוריתם ותוצאותיו. תיאור מפורט של סביבת סימולציה זו מסופק יחד עם דוגמאות מאוירות המציגות את יישום האלגוריתם. ביצועי האלגוריתם ויכולותיו מוערכים בקפדנות, והתוצאות והמסקנות המתקבלות מהפעלת האלגוריתם בסימולציה נדונות ומנותחות בפירוט.

אוניברסיטת בן גוריון בנגב

הפקולטה למדעי ההנדסה

המחלקה להנדסת מכונות

אלגוריתם לניהול וארגון חבילות במחסן על-ידי קבוצת סוכנים

23-29

נכתב על ידי:

סתיו ביטון 208854273

מנחה: פרופ' אמיר שפירא

יולי 2023 תמוז תשפ״ג