ENABLING EFFICIENT FLEET COMPOSITION SELECTION THROUGH THE DEVELOPMENT OF A RANK HEURISTIC FOR A BRANCH AND BOUND METHOD

An Undergraduate Honor Thesis

Presented in Partial Fulfillment of the Requirements for Graduation with Honor Distinction in Mechanical and Aerospace Engineering

By

Sun Siyuan

The Ohio State University

May 2023

Thesis Committee

Dr. Stephanie Stockar

Committee Member

Dr. Ali Jhemi

Copyrighted by

Sun Siyuan

2023

Abstract

In the foreseeable future, autonomous mobile robots (AMRs) will become a key enabler for increasing productivity and flexibility in material handling in warehousing facilities, distribution centers and manufacturing systems.

The objective of this research is to develop and validate parametric models of AMRs, develop ranking heuristic using a physics-based algorithm within the framework of the Branch and Bound method, integrate the ranking algorithm into a Fleet Composition Optimization (FCO) tool, and finally conduct simulations under various scenarios to verify the suitability and robustness of the developed tool in a factory equipped with AMRs. Kinematic-based equations are used for computing both energy and time consumption. Multivariate linear regression, a data-driven method, is used for designing the ranking heuristic. The results indicate that the unique physical structures and parameters of each robot are the main factors contributing to differences in energy and time consumption. improvement on reducing computation time was achieved by comparing heuristic-based search and non-heuristic-based search. This research is expected to significantly improve the current nested fleet composition optimization tool by reducing computation time without sacrificing optimality. From a practical perspective, greater efficiency in reducing energy and time costs can be achieved.

Acknowledgments

I would like to sincerely appreciate the guidance and support Dr. Stephanie Stockar and Mithun Goutham provided during my research. Your expertise and mentorship played a vital role in the success of this project. Mithun's contributions were particularly noteworthy, as he played an instrumental role in helping me overcome many of the challenges I faced.

Mithun's extensive coding skills and ability to explain complex concepts in a simple and easy-to-understand manner were critical to the success of my project. He patiently guided me through many difficult concepts that I struggled with and provided invaluable feedback on my work. Without his expertise and dedication, I would not have been able to achieve the level of success that I did.

Dr. Stephanie Stockar also provided valuable guidance and feedback on my project, and her knowledge and expertise in the field were instrumental in helping me develop a strong foundation for my research. Her availability and willingness to assist me throughout the process were greatly appreciated.

Overall, I am truly grateful for the guidance and support provided by Mithun and Dr. Stephanie. Their insights and suggestions were invaluable to my research, and their dedication to helping me succeed was truly appreciated. Thank you both for your contributions to my project.

Table of Contents

Abstract ii

Acknowledgments iv

Table of Contents vi

List of Tables viii

List of Figures ix

Chapter 1. Introduction 1

1.1 Background 1

1.1.1 Implementation of Branch & Bound Method in the Node Sequence Layer 3

1.2 Motivation for a Heuristic 5

1.3 Methodology 5

1.4 Research Goal 6

Chapter 2. Development of Robot Models and Integration with Branch and Bound 7

2.1 Development of Robot Models 7

2.1.1 Updating Robot Dynamics Based on Type & Cargo Mass 8

2.1.2 Equations for Computing Energy and Time Cost of Pure Traversal 9

2.1.3 Equations for Computing Energy & Time Cost of Pure Rotation Case 12

2.1.4 AMR Types and Important Parameters for Equation Calculation 17

2.2 Plot of Energy and Time Consumption for All Types of Robots Under Identical Inputs 19

2.3 Integration of Energy and Time Consumption Within FCO 24

Chapter 3: Multivariate Regression Implemented with Branch and Bound Method 28

3.1: Illustration of Ranking Heuristics and Introduction to Method of Multivariate Linear Regression (MMLR) 28

3.2: Important Parameters Tracking 33

3.2.1: State 1: Remaining Cargo Mass Capacity 33

3.2.2: State 2: Euclidian Distance between Task Location and Robot's Route's Edge. 38
3.3: Implementation of Multivariate Regression Method 39
Chapter 4: Conclusion 42
Chapter 5: Future work 43
Reference 44
Appendix A. Model 46
Analytical Hierarchy Process (AHP) Analysis: 46
Breadth First Search 48
Appendix B. MATLAB Code 49
Appendix C. Methods for Computing Euclidean Distance 61

List of Tables

Table 1 Types of Robots 7
Table 2 Types of Parameters Required for Modeling Energy and Time Consumption ... 18
Table 3 Controlled Variables for Pure Traversal Plotting 19
Table 4 Controlled Variables for Pure Rotation Plotting........ 21
Table 5 Comparison of Two Sets of Max Angular Acceleration 22
Table 6 Structure Array “Robots”........ 25
Table 7 States Specification for Data-Driven Method........ 30
Table 8 Example data of historic Task t costs 30
Table 9 Task assignment for A Simple Testing Scenario........ 34
Table 10 Plot of Spatial Movement for Task [1,3,9] and [2,3,4,7,8]........ 37
Table 11 Sample Euclidean Distance between Each Task Point and Each Edge 39
Table 12 Connotation of each element for the first column of Edge_matrix 63
Table 13 Example of Tabulated results for point-edge distance........ 68

List of Figures

Figure 1 Structure of the Nested Fleet Composition Optimization and Methods 2
Figure 2 Illustration for Simple Factory Layout 3
Figure 3 Branch & Bound Algorithm implemented in the Node Sequence Layer 3
Figure 4 Some of the different drive directions possible with a Mecanum 9
Figure 5 Illustration for a Simplified Trapezoidal Trajectory 18
Figure 6 Energy Consumption of All Types of Robots for Pure Traversal Case 20
Figure 7 Time Consumption for All Types of Robots for Pure Traversal Case 21
Figure 8 Energy Consumption of All Types of Robots for Pure Rotating Case............... 23
Figure 9 Time Consumption of All Types of Robots for Pure Rotating Case................. 23
Figure 10 Time Consumption of All Robots for Pure Rotation Case with Updated Limitation for Maximum Angular Acceleration 24
Figure 11 Illustration of Methods for Storing Energy & Time Cost 26
Figure 12 Schematic of Task Assignment 28
Figure 13 Illustration for Using Data from Past Exploration for Ordering Future Exploration 29
Figure 14 Function of Ranking Heuristic 30
Figure 15 Example of Time Consumption For MMLR 33
Figure 16 Task Visualization for All Feasible Robot ID and Robot Types 35
Figure 17 Visualization for Task [5,10] 35
Figure 18 Plot for Spatial Movement and Time Series for Task [5,10] 36
Figure 19 Evolution of Cargo & Volume Capacity for Robot ID 1 38
Figure 20 (a) 12 tasks assigned to the fleet 41
Figure 21 (b) 13 tasks assigned to the fleet 41
Figure 22 Illustration of the Function of TaskList.TaskCoordinates at Task 1 62
Figure 23 Example of matrix “*Edge_info*” 63
Figure 24 Graphical validation of elements ‘placement in Edge_info matrix 64
Figure 25 Cell array: D_cell_min 66
Figure 26 Overview of Structure Array Solution 67
Figure 27 Overview of Structure Array Solution 67
Figure 28 Task & Route for *solution.bestAssignment* 69

Chapter 1. Introduction

1.1 Background

To enable industry 4.0, assembly lines will likely be substituted with AMRs for enhanced automation and flexibility in material movement [1].The optimal number and type of robots for carrying out specific tasks depend on the material movement needs and facility layout. The objective is to minimize fleet energy usage, task completion time, and purchase cost. Minimizing fleet energy usage, task completion time, and purchase cost is critical for improving operational efficiency, reducing costs, and maximizing profits. To prepare for a future factory that utilizes advanced technologies like autonomous robots, a virtual simulation environment is necessary for testing different fleet compositions, layout designs, and operating procedures. A nested FCO has been developed at the OSU Center for Automotive Research (OSUCAR) as part of a Ford Alliance Project and it will assist with making informed decisions about fleet purchase and implementation while minimizing costs and risks associated with these decisions The structure of the optimizer together with the optimization methods is shown in Figure 1.

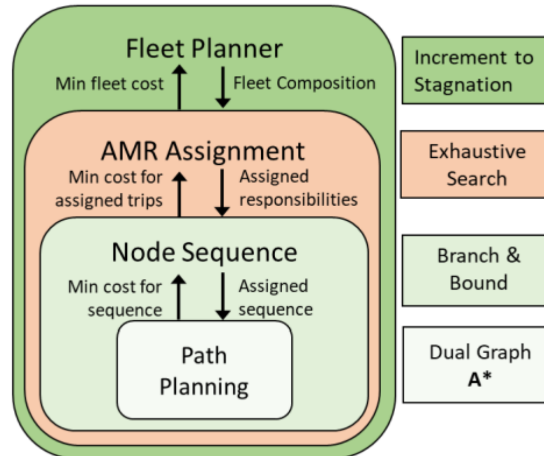


Figure 1 Structure of the Nested Fleet Composition Optimization and Methods

Given a specific cost function that might include total fleet energy, time required, investment cost or a combination of the above metrics, the FCO provides the global optimal solution to the fleet composition problem. However, even for simple factory fleet composition, see in Figure 2, and small number of requirements, the optimization problem is large-scale and solving it with an exhaustive search approach is computationally intractable. For example, with a set of 20 tasks and 5 types of AMRs available, the computation time required to find the optimal fleet composition will be in order of weeks.

At the node sequence layer, the optimization determines the order of nodes that the AMR must visit for minimizing the cost of meeting all pick-up and drop-off locations requirements. For this combinatorial NP-hard optimization problem, where NP stands for nondeterministic polynomial, a Branch & Bound (B&B) algorithm has been implemented.

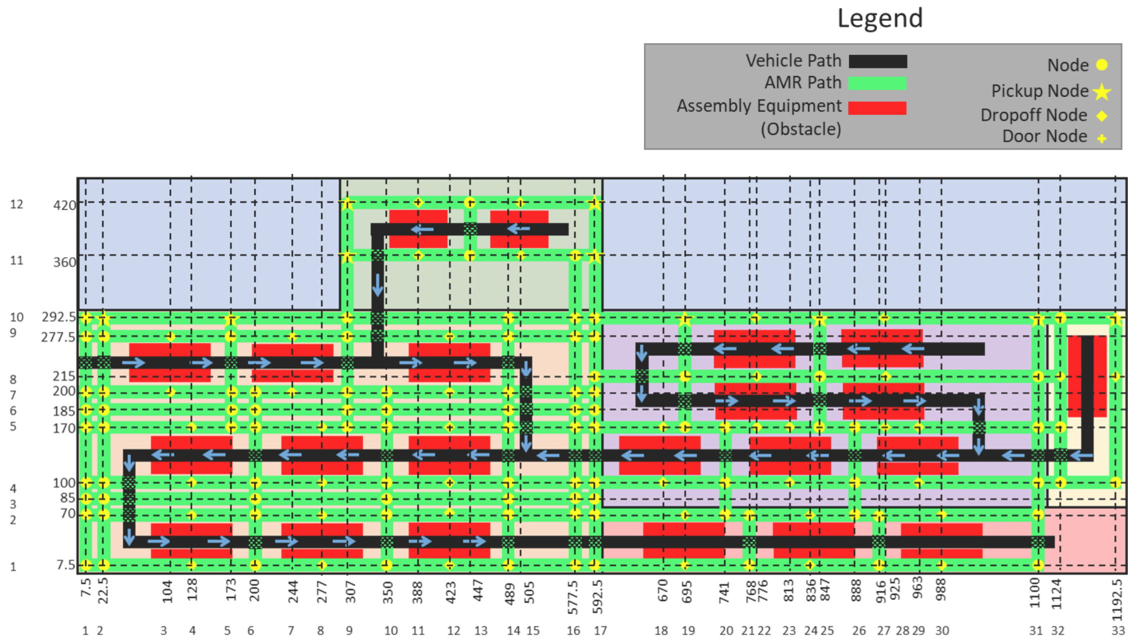


Figure 2 Illustration for Simple Factory Layout

1.1.1 Implementation of Branch & Bound Method in the Node Sequence Layer

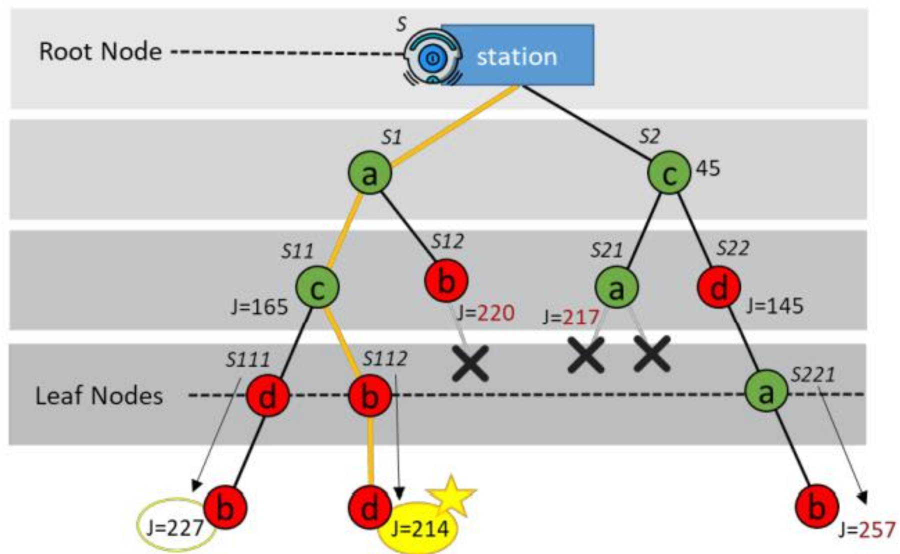


Figure 3 Branch & Bound Algorithm implemented in the Node Sequence Layer

The principle of a B&B algorithm is that an unexplored discrete search subspace can be represented by nodes in a dynamically generated search tree, only branching for further exploration when the candidate node is within upper and lower estimated bounds, thereby discarding nodes that initiate a branch of sub-optimal solutions [3]. At any instance, the incumbent solution is the best solution found thus far by the algorithm and is constantly updated as the algorithm progresses. This serves as the upper bound that eliminates provably suboptimal branches as the search space is explored.

An illustrative example of B&B for the node sequency optimization problem is shown in Figure 3. a factory floor positions a to b, and from c to d. The objective of using a branch and bound method is to solve optimization problems efficiently. To solve this problem, the B&B algorithm starts from an initial feasible solution, shown on the left-hand side of Figure 3, which consists in the sequence (a,c,d,b) associated with a cost of $J_1 = 227$, where J is computed as the sum of fixed cost for working robots and operational cost. This represents an initial candidate solution sequence and forms the first upper bound that can be used to discard any sub-optimal nodes. The algorithm then begins exploring the search space, for example by investigating node b as the second point in the sequence. However, as shown in Figure 3, the sequence (a,b) results in a cost of $J_2 = 220$, which is higher than the previous cost. Since there is no possible solution that would allow to complete the task of visiting all the points with a lower cost than J_1, this branch is cut and none of the associated combination are considered further. The process is then repeated until all the possible combinations are explored. It is therefore clear that, if a near-optimal

initial cost is obtained, the B&B can cut the branches early in the exploration process and therefore find the global optimal solution faster.

1.2 Motivation for a Heuristic

Using a near-optimal feasible solution as initial cost can reduce the number of explorations by discarding sub-optimal solution branches earlier than with a randomly generated initial exploration [4]. To further optimize the sequence optimization layer, a convex-hull insertion heuristic was implemented to reduce computation time significantly in some cases [5]. This heuristic builds upon the initial cost by iteratively refining the sequence until a near-optimal solution is obtained, thus improving overall fleet composition optimization.

One shortcoming of the existing nested FCO is that an exhaustive search method is used AMR assignment layer, where responsibilities are assigned to each robot. For improving the overall computation time of the FCO, the B&B algorithm can be implemented in this layer. If a near optimal initial guess of the robot to be selected can be provided to the algorithm, significant computation time improvements compare to a brute-force search are expected.

1.3 Methodology

This research aims to develop an efficient correlation of robot attributes (such as maximum speed, maximum payload, and electric driving range) with the objective function at the AMR Assignment level. To achieve this goal, a data-driven approach within the framework of Multi Attribute Decision Making (MADM) is proposed.

Analytical Hierarchy Process (AHP) is initially selected and combined with the pair-wise comparison matrix generated by the Branch & Bound algorithm. The reason for selecting the approach of AHP is because of its outstanding ability for enabling individuals or teams to make complex decisions based on multiple criteria and alternatives.[6-10] However, such method has become outdated since AHP was originally developed for subjective preferences with expert-assigned priorities and needs, and under the inspiration of it, an extension of the approach to accommodate a data-driven method was developed. Specifically, a method of multivariate linear regression is used.

1.4 Research Goal

In this research, equations for modeling energy and time consumptions for five types of different AMRs will be developed and a ranking heuristic for robot selection based on historic exploration data will be generated to speed up the branch & bound algorithms.

Chapter 2. Development of Robot Models and Integration with Branch and Bound

In this chapter, the energy usage and nominal vehicle profiles for five different types of autonomous robot have been developed. These modes are crucial for determining which tasks will be assigned to which robot in the FCO because this decision is based on the comparison of energy and time consumption of each combination. The accuracy of the models is compared against literature data. Finally, the models are integrated in the FCO, such that the library of available robots for selection by the optimizer is increased.

2.1 Development of Robot Models

This study considers the modeling of five types of robots that the FCO will then be able to select as part of the optimal composition of the fleet. The five robot types are summarized in Table 1.

Table 1 Types of Robots

Types	Robots
Type 1	Large Differential Drive Autonomous Cart Puller
Type 2	Large Differential Drive Autonomous Forklift
Type 3	Smaller High-Capacity Omnidirectional Drive Mobile Robots
Type 4	Small Lower Capacity Differential Drive Mobile Robots-Large
Type 5	Small Lower Capacity Differential Drive Mobile Robots-Smaller

2.1.1 Updating Robot Dynamics Based on Type & Cargo Mass

The mass of the robot is given by

$$M_{dynamic} = M_{empty} + M_{cargo} \quad (1)$$

where M_{empty} is the mass of the robot, and M_{cargo} is the mass of the cargo. Then, assuming that the robot is a cuboid, the inertia calculated with respect to the center of gravity is

$$I_{cg} = \frac{1}{12} \cdot M_{dynamic} * (L^2 + W^2) \quad (2)$$

where L and W are the length and width respectively. Then the moment of inertia is obtained using the parallel axis theorem [11]:

$$I_o = I_{cg} + M_{dynamic} \cdot a^2 \quad (3)$$

where a is the distance between driven axle and center of gravity.

For a robot with n_d driven wheels and n_p passive wheels, let $W \in \mathbb{R}^{n_d+n_p}$ be the weight distribution such that $0 < W_i < 1$ and $\sum_{i=1}^{n_d+n_p} W_i = 1$. If C_{rr_i} is the coefficient of rolling resistance at wheel i, then the rolling resistance to be overcome is given by:

$$F_{rr} = \sum_{i=1}^{n_d+n_p} C_{rr_i} \cdot W_i \cdot M_{dynamic} \cdot g \quad (4)$$

In case of differential drive robots where turning involves slipping at the driven wheels, there are additional frictional forces to be overcome, resulting in a moment given by

$$M_f = a \sum_{i=1}^{n_d} \mu_{s_i} \cdot W_i \cdot M_{dynamic} \cdot g \quad (5)$$

where μ_{s_i} is the sliding coefficient at wheel i.

It is worth noting that this model formulation is valid only for 3 types of all the driving directions shown in Figure 4 [13], forward, reverse and rotation. Once these parameters and quantities are found, an energy-based modeling approach is used for calculation.

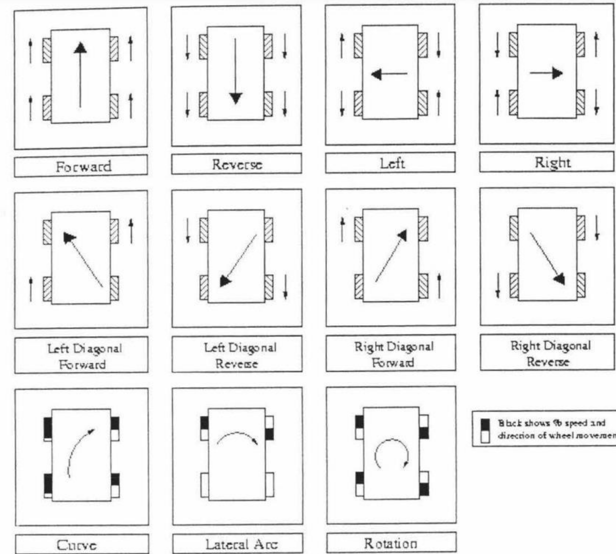


Figure 4 Some of the different drive directions possible with a Mecanum drive system.

2.1.2 Equations for Computing Energy and Time Cost of Pure Traversal

In this section, a simple energy-based model of a generic robot is developed for pure traversal and pure rotational operation, which is consistent with the operation of the five robot types considered in the optimization. The objective of the model is to evaluate the energy consumption and travel time taken by each robot to complete a task.

1. Acceleration phase for pure traversal case

Maximum acceleration between acceleration limit and acceleration calculated through using Newton's second law of motion, $a = F/m$, is given by :

$$a_{max} = \max\left\{MaxAccel, \frac{2 \cdot \frac{\tau_{\max}}{r_{\text{wheel}}} - F_{\text{rr}}}{M_{dynamic}}\right\} \tag{6}$$

where $\tau_{\max}$ stands for maximum torque. Maximum velocity can be obtained by conducting a similar comparison between the maximum velocity limit and the calculated velocity, and it's given by:

$$v_{max} = \max\left\{MaxSpeed, \sqrt{2 \cdot a_{max} \cdot \frac{d}{2}}\right\} \tag{7}$$

By doing so, it can be guaranteed that both calculated velocity and acceleration never exceed the specified limits. Distance traveled for acceleration portion is related by distance, velocity, and acceleration, and it's defined as :

$$d_a = \frac{v_{\max}^2}{2 \cdot a_{max}^2} \tag{8}$$

Torque is computed by using the classic torque formula: moment arm multiplies applied force, is defined as:

$$a_{max} \cdot M_{dynamic} + F_{rr} \tag{9}$$

$$\tau = \frac{1}{2} \cdot r_{wheel} \cdot \left(a_{max} \cdot M_{dynamic} + F_{rr}\right) \tag{10}$$

where the moment arm is defined as the distance between the axis of rotation and the point where the force is applied, which is half of the wheel radius. The applied force is defined as the sum of the rolling resistance force and product of maximum acceleration and dynamic moment.

Energy consumed during acceleration phase is given by:

$$E = \left(a_{max} * M_{dynamic} + F_{rr}\right) * d_a / 0.95 \tag{11}$$

where "0.95" is a numerical scaling factor, representing efficiency.

2. Deceleration phase for pure traversal case:

In order to find acceleration during deceleration phase, a_{max} was changed into negative since in this case, velocity is decreasing which implies a negative acceleration. Torque and energy consumption during deceleration phase is given by:

$$\tau = \frac{1}{2} \cdot r_{wheel} \cdot \left(-a_{max} \cdot M_{dynamic} + F_{rr}\right) \tag{12}$$

$$E = \left(-a_{max} \cdot M_{dynamic} + F_{rr}\right) \cdot d_a \cdot 0.95 \tag{13}$$

where all variables remain the same meaning as above.

Time used during both acceleration and deceleration is given by:

$$T = 2 \cdot \frac{v_{max}}{a_{max}} \tag{14}$$

where it can be interpreted as two times the time required reach the v_{max}, starting from rest and accelerating at a constant rate a_{max}, and then coming to a stop again by decelerating at the same constant rate.

3. Coasting phase for pure traversal case:

Distance traveled during coasting phase (d_c) is defined as :

$$d_c = d - 2 \cdot d_a \tag{15}$$

Where d is total distance, and d_a is the distance traveled during both acceleration and deceleration phase.

Torque during coasting phase is defined as:

$$\tau = \frac{1}{2} \cdot r_{wheel} \cdot F_{rr} \tag{16}$$

Where differs slightly from its counterparts in non-coasting phase and this is directly resulted from the fact that acceleration (deceleration) is zero here, and the applied force in this case will just be rolling resistance force (F_{rr}) alone.

Energy consumed during coasting phase is defined as :

$$E = F_{rr} \cdot d_c \cdot 0.95 \tag{17}$$

Time taken during coasting phase is defined as:

$$T = \frac{d_c}{v_{max}} \tag{18}$$

2.1.3 Equations for Computing Energy & Time Cost of Pure Rotation Case

The rotational force is being defined as:

$$F_w = \frac{\tau_{max}}{r_{wheels}} \tag{19}$$

where it equals to the ratio between maximum torque (τ_{max}) and wheel radius (r_{wheels}), with maximum torque is given as one of robot's parameters.

Rotating moment is given by:

$$M_r = F_w \cdot tw \tag{20}$$

where tw trackwidth.

Then, the maximum acceleration is given by:

$$\alpha_{max} = \min\{MaxAlpha, (M_r - M_f)/I_o\} \tag{21}$$

where a similar comparison conducted for pure traversal case is adopted here. From here, the minimal value between maximum angular acceleration (*MaxAlpha*) and acceleration calculated using classic formula, where $M_r, M_f, and\ I_o$ stands for rotational moment, frictional moment, and moment of inertia, respectively.

Maximum angular velocity is given by:

$$\omega_{max} = \min\left\{MaxOmega, \sqrt{2 \cdot \alpha_{max} \cdot \frac{\theta}{2}}\right\} \tag{22}$$

where it's obtained by the same fashion as α_{max}.

Theta rotated during the acceleration process is given by:

$$\theta_a = \frac{\omega_{max}^2}{2 \cdot \alpha_{max}} \tag{23}$$

Distance traveled during the acceleration process is given by:

$$d_a = \theta_a \cdot \frac{tw}{2} \tag{24}$$

1. <u>Acceleration phase for pure rotational case:</u>

Rotational moment (M_r) is defined as:

$$M_r = \alpha_{max} \cdot I_o + M_f \tag{25}$$

where it's equivalent to the sum of moment of inertia (I_o) and the product of maximum angular acceleration (α_{max}).

Rotational force (F_w) is given by:

$$F_w = \frac{M_r}{tw} \tag{26}$$

where the rotational moment calculated earlier here will be used to divide by trackwidth.

Energy consumption during the acceleration phase is defined as:

$$E = 2 \cdot F_w \cdot \frac{d_a}{0.95} \tag{27}$$

where d_a stands for distance traveled for accelerating process, and 0.95 is the same scaling efficiency stated earlier.

2. <u>Deceleration phase for pure rotational case:</u>

Equation (30-32) covers calculation of energy cost during deceleration process by simply changing the maximum angular acceleration into negative and repeating the same process above.

From here, the rotational moment during deceleration phase is given by:

$$M_r = -\alpha_{max} \cdot I_o + M_f \tag{28}$$

Rotational force (F_w) during deceleration phase is given by:

$$F_w = \frac{M_r}{tw} \tag{29}$$

Energy consumption during deceleration phase is given by:

$$E = 2 \cdot F_w \cdot \frac{d_a}{0.95} \tag{30}$$

Time used during non-coasting (including both acceleration and deceleration) process is given by:

$$T = 2 \cdot \frac{\omega_{max}}{\alpha_{max}} \tag{31}$$

where it's similar to the procedure employed in the pure traversal case, the computation of time consumption during non-coasting phase involves multiplication of two by the ratio between maximum angular velocity (ω_{max}) and maximum angular acceleration (α_{max}).

3. Coasting phase for pure rotational case:

During the coasting phase, the amount of angle rotated (θ_C) is obtained by subtracting the total rotated angle from the amount of angle traveled for both acceleration and deceleration (θ_a), thus necessitating the numerical scaling factor of two. This is clearly depicted in Equation (26), whereby the distance traveled for turning angles during the coasting process can be determined through a similar process. The equations used to calculate crucial parameters, such as M_r, F_w and E, are analogous to those utilized for both the acceleration and deceleration phase, except for the moment created by rotational force (M_r), which is equated to the moment generated by frictional force (M_f). In terms of time consumption, positive angle values are employed to divide the positive maximum angular velocity.

Angle rotated (θ_C) is obtained by:

$$\theta_C = \theta - 2 \cdot \theta_a \tag{32}$$

where θ is total rotated angle and θ_a is the amount of angle traveled for both acceleration and deceleration.

The distance traveled for turning angles during the coasting process, d_c, is given by:

$$d_c = \theta_C \cdot \frac{tw}{2} \tag{33}$$

Rotational moment during coasting phase is given by:

$$M_r = M_f \tag{34}$$

Rotational force is defined as:

$$F_w = \frac{M_r}{tw} \tag{35}$$

Energy consumed during coasting phase is given by:

$$E = 2 \cdot F_w \cdot \frac{d_c}{0.95} \tag{36}$$

where 0.95 is the same scaling factor listed above.

Time used during the coasting phase is defined as:

$$T(\theta > 0) = \frac{\theta_c(\theta_c > 0)}{\omega_{max}(\theta_c > 0)} \tag{37}$$

where only positive angle values are employed to divide the positive maximum angular velocity.

Recall at the end of part 1, "Updating Robot Dynamics Based on Type & Cargo Mass", differences between omnidirectional robots and non-omnidirectional robots for computing two major parameters have been listed for comparison. Additionally, it is also equally important to note that such differences will further affect the calculation for both frictional force (F_L), moment caused by front wheel sliding (M_f) , and eventually for energy consumption and time usage.

I also noticed that approximated energy consumption will become zero during coasting phase, as rotation moment (M_r) become zero as a result of M_f being zero, and this will lead rotational force (F_w) become zero.

Energy consumed by omnidirectional robot during coasting phase is given by:

$$E_{omnicoasting} = 2 \cdot F_w \cdot \frac{d_c}{0.95} \tag{38}$$

where it is evident that the energy cost can be approximated as zero due to rotational force is zero.

2.1.4 AMR Types and Important Parameters for Equation Calculation

Fundamentals of kinematics has been used extensively for developing the equations to describe both energy usage and time take by a robot to traverse a segment of the manufacturing plant [11]. This was accomplished by postulating that the AMR speed follows a prescribed trapezoidal trajectory, as the one shown in Figure 5[13]. The speed trajectory is composed by three segments, namely the acceleration phase, the cruise phase and the deceleration phase, and this provides several advantages when modeling energy consumption for vehicles.

First, it offers a more accurate representation of real-world driving conditions, resulting in a more precise model of energy consumption. Second, analyzing energy consumption across different velocity situations can optimize energy use and identify inefficiencies. Third, customizing the model for different vehicle types and driving conditions can improve its accuracy and usefulness. Overall, this approach provides a more realistic and effective means of identifying opportunities for reducing energy use and improving efficiency in vehicles.

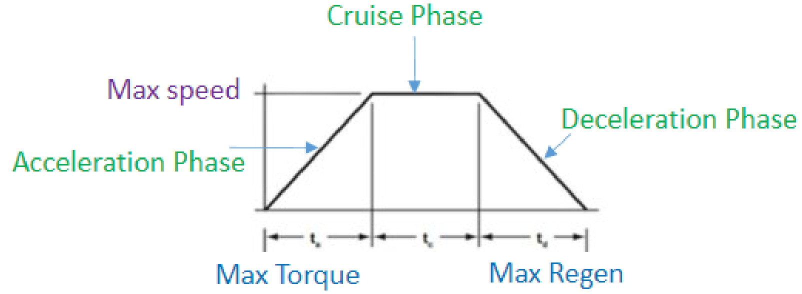


Figure 5 Illustration for a Simplified Trapezoidal Trajectory

Each of the robot parameters defined in Table 3 were provided by the Ford Motor Company for the 5 different types of robots. Based on these parameters, the developed robot models provide the energy and time for each type of robot.

Table 2 Types of Parameters Required for Modeling Energy and Time Consumption

Parameters:			
Physical Constant	Robot Length [m]	Robot Width [m]	Robot Height [m]
	Track Width [m]	Distance between rear axle and the center of gravity [m]	Distance between front axle and the center of gravity [m]
Mass & Volume	Empty Mass [Kg]	Mass Limit [Kg]	Volume Limit [m^3]
Wheel	Number of Driven Wheels	Number of Passive Wheels	Wheel Radius
	Rolling Resistance	Sliding Friction	Weight Distribution
Dynamic Limitation	Maximum Speed [m/s]	Maximum Acceleration [m/s^2]	Max Angular Velocity [rad/s]

	Maximum Angular Acceleration [rad/s^2]		
Motor and Battery	Reduction Ratio	Max Torque [Nm]	Battery Size
	Battery Capacity [kWh]	Motor Efficiency	

To conveniently store the parameters in TABLE REF and use them in the FCO, a structure is created in MATLAB.

2.2 Plot of Energy and Time Consumption for All Types of Robots Under Identical Inputs

The energy consumption and time utilization models for each robot are simulated under both pure traversal and pure rotational case. The parameters used in the simulation for a pure traversal case, across the five robots, are summarized in Table 3.f

Table 3 Controlled Variables for Pure Traversal Plotting

$\theta = 0\ [deg], as\ here\ it\ is\ assumed\ no\ angle\ has\ been\ turned$
$cargo\ mass = 20\ [kg], with\ a\ stepsize\ of\ 0.8\ kg.$
$distance = 10\ [m], with\ a\ stepsize\ of\ 0.5\ m.$

Figure 6 Energy Consumption of All Types of Robots for Pure Traversal Case shows the energy utilization for each of the five considered AMRS when conducting the same maneuver for different cargo mass and travelled distance. Similarly, Figure 7 shows the corresponding time required by the robot to complete the same task. Intuitively, as the

distance between two points increases, or the weight being carried increased, both energy and time increase. Due to the physical limitations of each robot, some have similar characteristic surfaces in both energy and time. For example, the forklift and the puller have the same time profile due to the limitation in the acceleration. Similarly, both differential drive robots also have overlapping surfaces. Moreover, there is an intersection of characteristic surfaces for the fork lifter and omnidirectional robot. Specifically, the omnidirectional robot has a higher energy demand than the fork lifter when the travel distance is small. The opposite trend is observed, however, when the distance between two points increases. This is due to the trapezoidal speed trajectory considered. From Figure 6, it is observed that the omnidirectional robot is always slower than any of the other AMRs. This is due to the lower torque to mass ratio, which impacts the acceleration ability of the system.

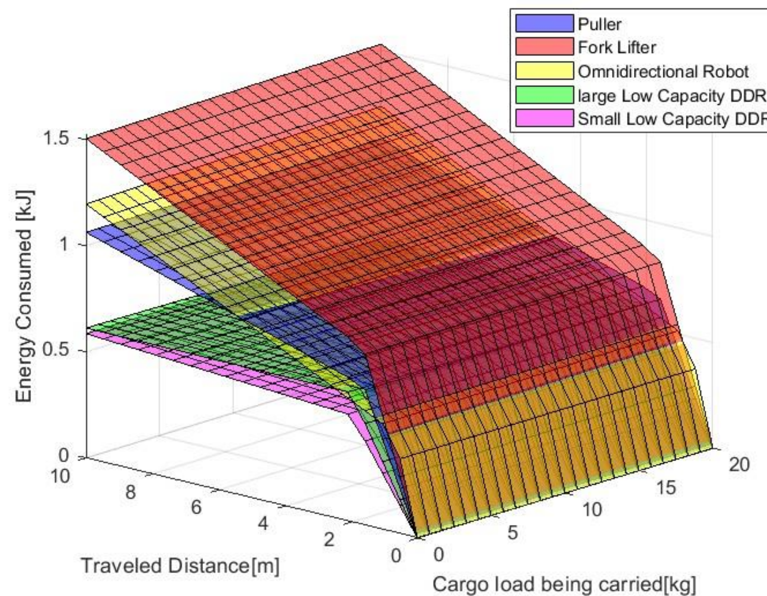


Figure 6 Energy Consumption of All Types of Robots for Pure Traversal Case

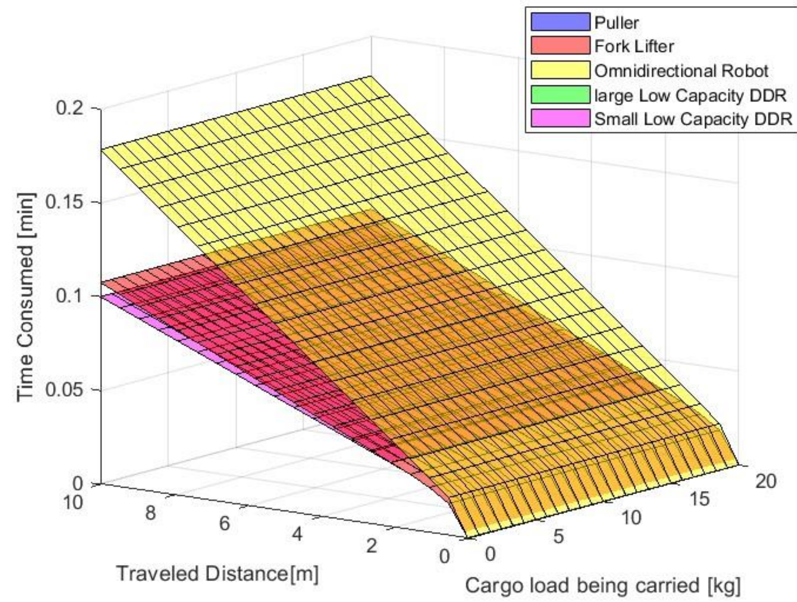


Figure 7 Time Consumption for All Types of Robots for Pure Traversal Case

The parameters used in the simulation for a pure traversal case, across the five robots, are summarized in Table 4

Table 4 Controlled Variables for Pure Rotation Plotting

$\theta = 45\ [deg]$
$cargo\ mass = 20\ [kg], with\ a\ stepsize\ of\ 0.8\ kg.$
$distance = 0\ [m], as\ here\ it\ is\ assumed\ no\ angle\ has\ been\ turned$

Figure 8 shows the energy utilization for each of the five considered AMRS when conducting the same maneuver for different cargo mass and travelled distance. Similarly, Figure 9 shows the corresponding time required by the robot to complete the same task.

It was also observed that as the distance between two points increased or the weight being carried increased, both energy and time requirements for the task also increased. However, due to the specific physical limitations of each robot, certain entities exhibited similar performance characteristics in terms of both energy consumption and time. For example, all AMRs except omnidirectional robot share the same time profile due to the limitation in the acceleration. However, no intersection of characteristic surfaces has been spotted for either of the two types of robots. Particularly, the fork lifter robot has the highest energy demand and omnidirectional robot has the highest demand for time usage.

Another intriguing observation that has been made for Figure 9 is that only two surfaces exist. Yet this again could be explained by the fact that all non-omnidirectional robots' maximum angular acceleration has been set to the same.

Table 5 below indicates the original value and revised value for α_{max}, with revised values are arbitrarily assigned and the sole purpose is to illustrate the overlapping will disappear once different values for maximum accelerations are being used.

Table 5 Comparison of Two Sets of Max Angular Acceleration

	Old α_{max} (rad/s^2)	New α_{max}(rad/s^2)
Puller	10	6.5
Forklift	10	7.5
Smaller Low-Capacity DDR	10	8.5
Larger Low-Capacity DDR	10	9.5

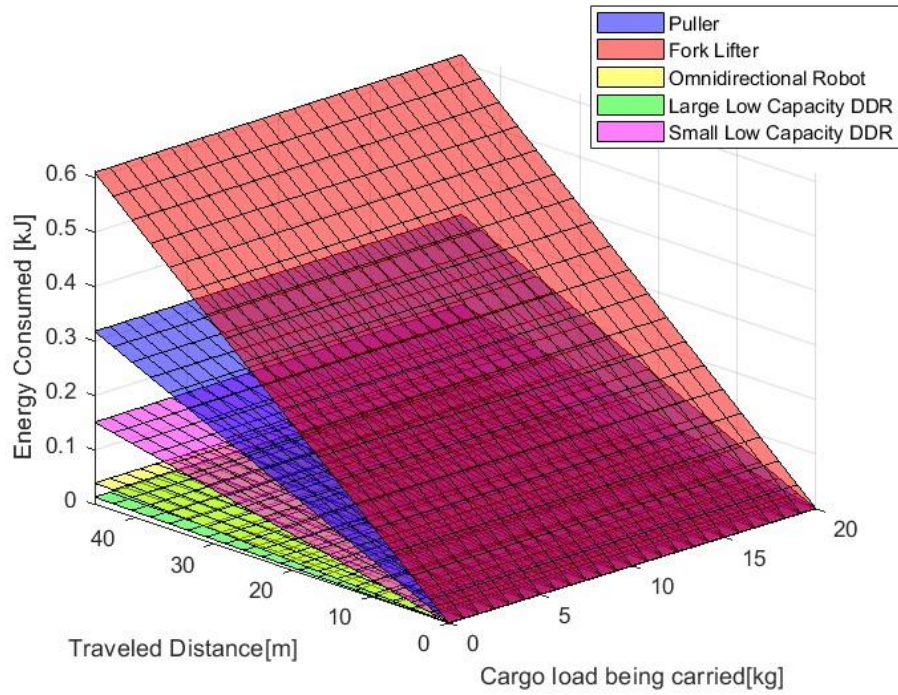


Figure 8 Energy Consumption of All Types of Robots for Pure Rotating Case.

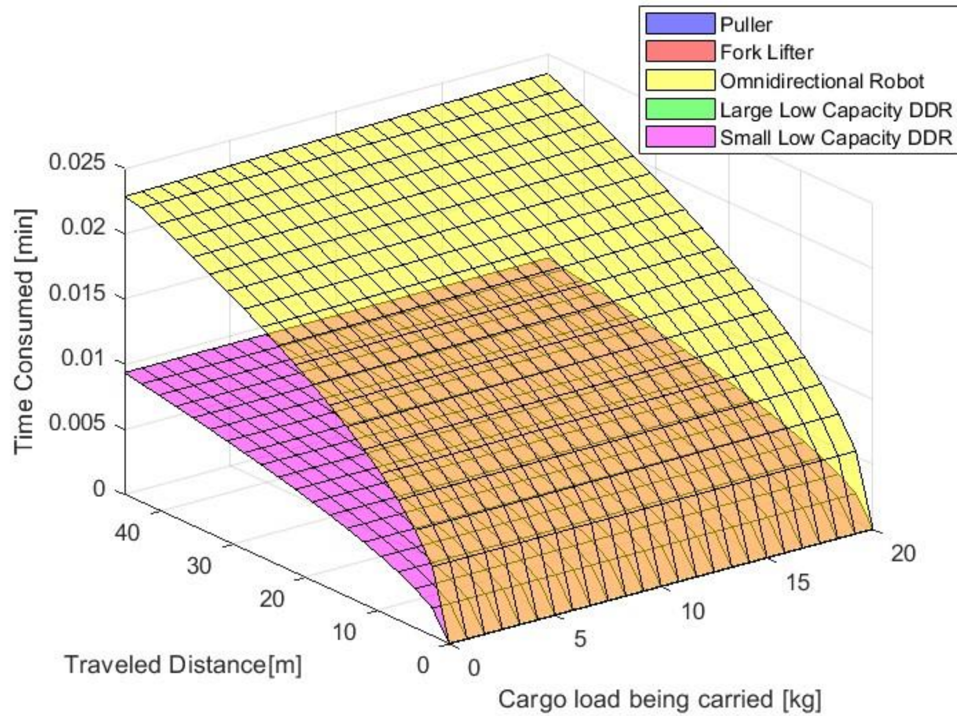


Figure 9 Time Consumption of All Types of Robots for Pure Rotating Case.

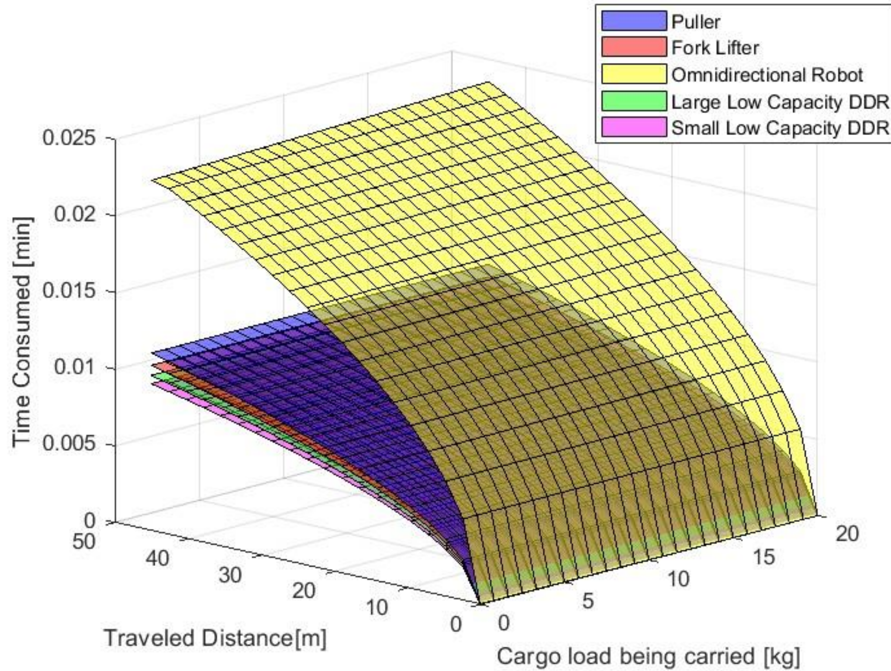


Figure 10 Time Consumption of All Robots for Pure Rotation Case with Updated Limitation for Maximum Angular Acceleration

Through plotting, the issue of surfaces overlapping is indeed addressed, and from Figure 10, Smaller Low-Capacity DDR is proved to be least time-consumptive, and omnidirectional robot still dominates the position for being most time-consumptive type of robot. The same explanation used for the surface overlapping observed in the case of pure traversal case still could be employed here.

2.3 Integration of Energy and Time Consumption Within FCO

This section will be used for showing the adopted method for having previously analyzed kinematic based equations integrated into the currently existing FCO.

At this stage, energy and time costs for carrying out a specific task by each robot can be

computed, given the constrained conditions and parameters. However, a convenient method to store all the calculated energy and time costs becomes crucial.

To achieve this, a structure array “Robotics” is created, which has multiple fields, see Table 6listed as below:

Table 6 Structure Array “Robots”

Name	**Field(s)**	**Subfield**
Robots	mass Limit	
	volume Limit	
	battery Limit	
	charge Power	
	Kerb Weight	
	cost Matrix	Energy Cost Matrix Time Cost Matrix

From here, six fields are being included, and energy & time cost are being stored in the form of matrix, secured as two subfields within the field “cost matrix”. A graphical illustration explaining how energy and time consumption being stored are listed below, see in Figure 11 Illustration of Methods for Storing Energy & Time Cost.

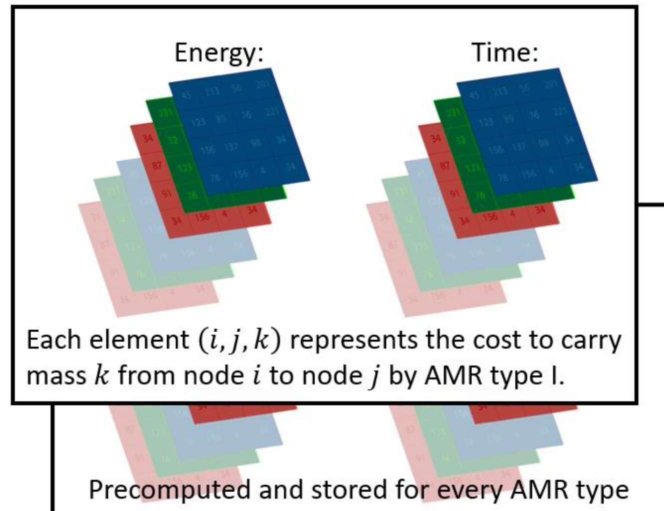


Figure 11 Illustration of Methods for Storing Energy & Time Cost

From Figure 11, as denoted on the graph itself, both energy and time cost are being stored as a 3-dimensional array, also commonly known as "3D matrix" for every AMR type. For n tasks, all possible payloads are first obtained by $\binom{n}{k}$ for $k \in \{1,2 \cdots n\}$, and this yields the mass vector = $[m_1\ m_2\ m_3 \ldots \ldots m_p]$, with p representing a user defined vector length, that represents all realizations of payload.

In the present study, a cost analysis of AMR fleet composition optimization was conducted, with a particular focus on the energy and time requirements of transporting mass K from node *i* to node j by each specific AMR type. To achieve this, a 3-dimensional array was utilized, with each value representing the cost associated with a given layer and specific AMR type. The lookup table was restricted to include only stations, charge stations, and task locations, as all possible realizations of payload were considered, and no interpolation was necessary. This pre-computation significantly

reduced computation time. As the present study was entirely code analysis-based, a simplified and visually comprehensible methodology was employed to integrate the energy and time cost analyses into the FCO. By doing so, the logical flow of the process was reinforced, and the results were made more accessible. To achieve this purpose, a simplified pseudocode written in MATLAB has been provided for comprehension, see in Appendix B: Section 2.3.

Chapter 3: Multivariate Regression Implemented with Branch and Bound Method

This chapter introduces the multivariate linear regression method and its implementation within the Task Assignment B&B algorithm.

3.1: Illustration of Ranking Heuristics and Introduction to Method of Multivariate Linear Regression (MMLR)

Consider the following schematic of the task assignment tree in Figure 12, where the highlighted branch signifies the decision for having both tasks T_1 and T_2 assigned to robot r_2 and task T_3 to robot r_3.

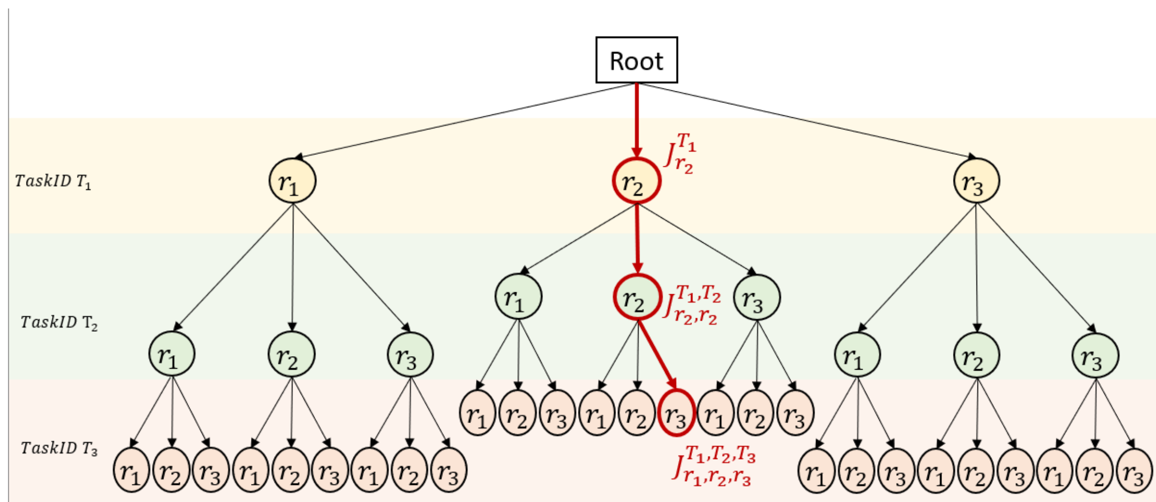


Figure 12 Schematic of Task Assignment

The implemented branching order uses the 'Breadth First' approach, which evaluates the costs of children nodes first and then branches to them in a cheapest-first order. This has a significant effect on how the incumbent solution is updated and is limited by being able to only access information about costs of robot assignment at that branching decision, as

opposed to using information about the historic performance of the robot for all the remaining tasks.

For the highlighted branching decision of Figure 12, where task T_3 is to be assigned to robot r_1, r_2 or r_3, costs associated with branching selections exist in the highlighted red box of Figure 13. By utilizing this information, it is expected that a more sophisticated data-driven branching order can be developed, instead of a breadth-first approach. Implementing such an approach is expected to lead to greater efficiency, especially when the number of robots and tasks increases.

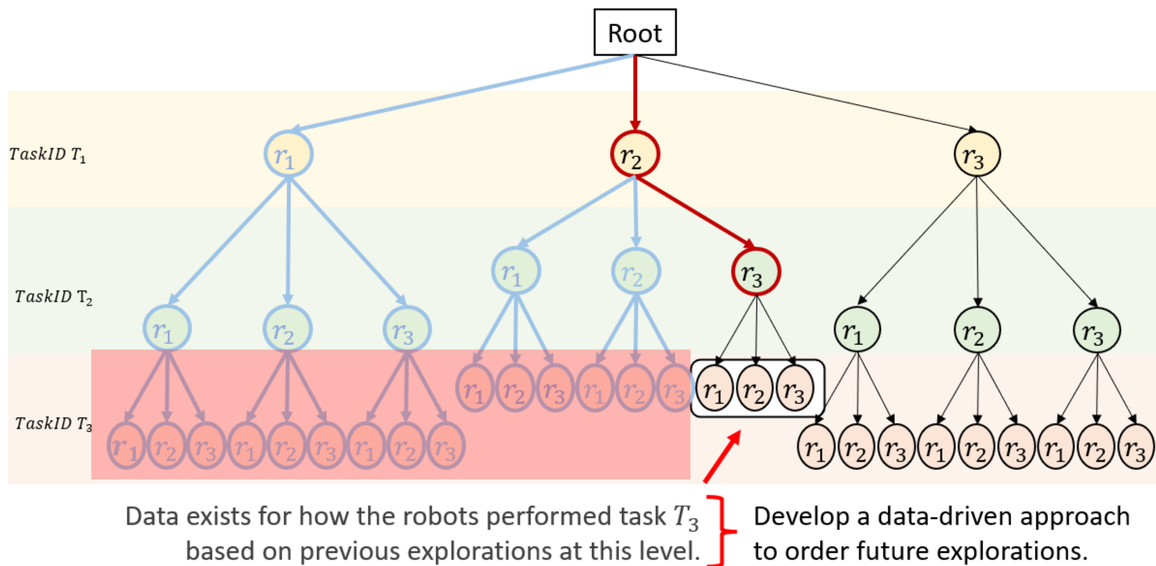


Figure 13 Illustration for Using Data from Past Exploration for Ordering Future Exploration

The objective of the heuristic is to rank the available robots r_1, r_2, r_3 for priority in exploration when being assigned task T. For instance, Figure 14 illustrates a worthy question, such that which robot should be assigned to task 2 after having task 1 assigned to robot 2. Should we explore robot 3 for the first one?

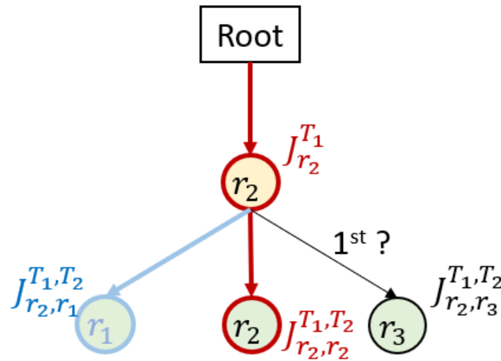


Figure 14 Function of Ranking Heuristic

Table 7 States Specification for Data-Driven Method

State 1:	Remaining cargo mass capacity after finishing previous exploration.
State 2:	Total Euclidean distance between robot's route and remaining task positions

At every branching instance, the fleet state includes details of each robot's minimum remaining cargo capacity and Euclidean distance (between each task and all route's edges) at the end. As an example of the previous explorations, Table 8 shows the state of robots $\boldsymbol{r_1}, \boldsymbol{r_2}$ and $\boldsymbol{r_3}$ and the resulting leaf node cost of branching to those robots. The decision to branch out into one of the available robots is highlighted. By using this information that relates the states of the robots with the costs associated with assigning the task T to a robot, a data-driven approach can be used to predict the costs of each type of robot when a new fleet state is encountered.

Table 8 Example data of historic Task t costs

	r_1 State 1 (kg)	r_1 State 2 (m)	r_2 State 1 (kg)	r_2 State 2 (m)	r_3 State 1 (kg)	r_3 State 2 (m)	Task T Robot	Cost at Leaf Node (kJ)
Exploration 25	13.2	246	2.4	68	3.7	141	r_1	3243

Exploration 26	13.2	246	2.4	68	3.7	141	r_2	1724
Exploration 27	13.2	246	2.4	68	3.7	141	r_3	2425
Exploration 28	12.7	252	2.7	48	3.2	203	r_1	3453

Since the ranking heuristic must be implemented within a large-scale optimization problem that requires significant computational resources, an additional objective is that the data-driven approach must be computationally efficient. The overhead associated with ranking the robots must be a negligible computational expense that can be used every time a branching of robots is to be ordered. For this reason, predicting the cost of the branching decision using a neural network is ruled out. A multi-variate linear regression approach is selected for computational efficiency as detailed below.

Consider the case where there are r_p robots in the fleet and as explorations are being conducted in the task assignment search space, the states as defined in Table 8 are tracked for each robot at some task level t. If n branching instances were conducted in the past and a total of m fleet states were tracked at each exploration, then the ordered states of the fleet form the $X \in \mathbb{R}^{n \times m}$ matrix.

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \tag{39}$$

Similarly, let the matrix J_t capture the cost at the leaf node for assigning each robot type at each exploration.

$$J_t = \begin{bmatrix} J_{1r_1} & J_{1r_2} & \cdots & J_{1r_p} \\ J_{2r_1} & J_{2r_2} & \cdots & J_{2r_p} \\ \vdots & \vdots & \ddots & \vdots \\ J_{nr_1} & J_{nr_2} & \cdots & J_{nr_p} \end{bmatrix} \tag{40}$$

The objective of multivariate regression is to find a matrix W_t that best approximates the relation between the fleet state and the cost of assigning a robot, i.e, XW_t approximates J_t. This is achieved by finding W_t that minimizing the difference between XW_t and J_t, defined as

$$g(W) = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \begin{bmatrix} w_{1r_1} & w_{1r_2} & \cdots & w_{1r_p} \\ w_{2r_1} & w_{2r_2} & \cdots & w_{2r_p} \\ \vdots & \vdots & \ddots & \vdots \\ w_{mr_1} & w_{mr_2} & \cdots & w_{mr_p} \end{bmatrix} - \begin{bmatrix} J_{1r_1} & J_{1r_2} & \cdots & J_{1r_p} \\ J_{2r_1} & J_{2r_2} & \cdots & J_{2r_p} \\ \vdots & \vdots & \ddots & \vdots \\ J_{nr_1} & J_{nr_2} & \cdots & J_{nr_p} \end{bmatrix} \tag{41}$$

This can be achieved by minimizing $g(W)$ or $||g(W)||^2$by choosing W_t appropriately. By differentiating $||g(W)||^2$with respect to W, the best fitting matrix $\widehat{W_t}$ is found:

$$\widehat{W_t} = argmin\|g(W_t)\|^2 = (X^T X)^{-1} X^T J_t \tag{42}$$

Thus, for a new fleet state $\bar{x}$, the cost for assigning available robots to task t can be efficiently approximated as $\overline{J_t} = \bar{x}\widehat{W_t}$. While the obtained costs may not be accurate, it is expected that the ranking of the robots for the task can be selected based on the approximated costs.

The approach is also confirmed to be computationally efficient as illustrated in Figure 15, where, for randomly generated matrices J_t and X matrices, the computation time to find matrix W_t is shown to be negligible. To limit memory demand, only the last 20 explorations are stored for each task.

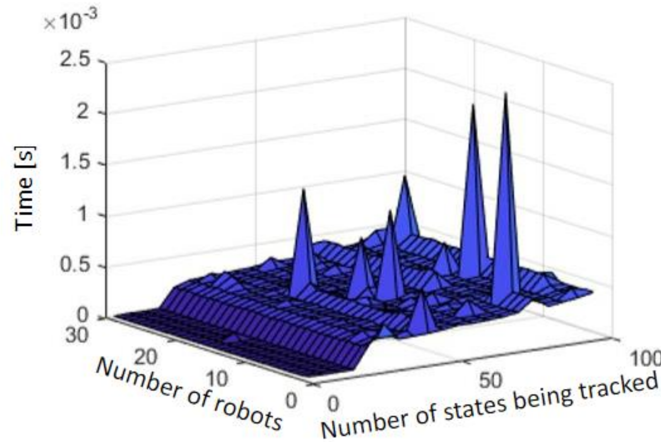


Figure 15 Example of Time Consumption For MMLR

3.2: Important Parameters Tracking

Section 3.1 introduces the construction of the X matrix, involves the use of two states, as indicated in Table 7. This section will delineate the process of acquiring states 1 and 2, as well as elucidate the significance of both states.

3.2.1: State 1: Remaining Cargo Mass Capacity

Graphical illustrations pertinent towards the evolution of remaining cargo capacity and volume capacity have been achieved. Specific tasks have been assigned to individual robots based on previous analyses of their energy and time consumption.

An example of testing scenario with detailed task assignments is listed in Table 9.

Table 9 Task assignment for A Simple Testing Scenario

Types of Best Assignments	Robot Type	Assigned Task
1	Large Differential Drive Autonomous Cart Puller (Type 1)	[5,10]
2	Large Differential Drive Autonomous Cart Puller (Type 1)	[1,3,9]
7	Small Lower Capacity Differential Drive Mobile Robots-Large (Type 4)	[2,4,6,7,8]

Note here number under the column of “Assigned Task” represents their order from a calculated list of tasks by using given information.

Figure 16 shows a detailed illustration for each of the ten tasks on a map, with each task being represented by an arrow. For each arrow, its head’s coordinate stands for the location where cargo being dropped off and its tail’s coordinate stands for the location where cargo being picked up.

On Figure 16’s right part, for example, row 1 stands for in order to complete task 1, the robot has to pick up a 180 kg cargo and drop off within the indicated time range (37min-43min in this case.) This provided a direct visualization of time consumption at each stage for assigned robots to finish tasks.

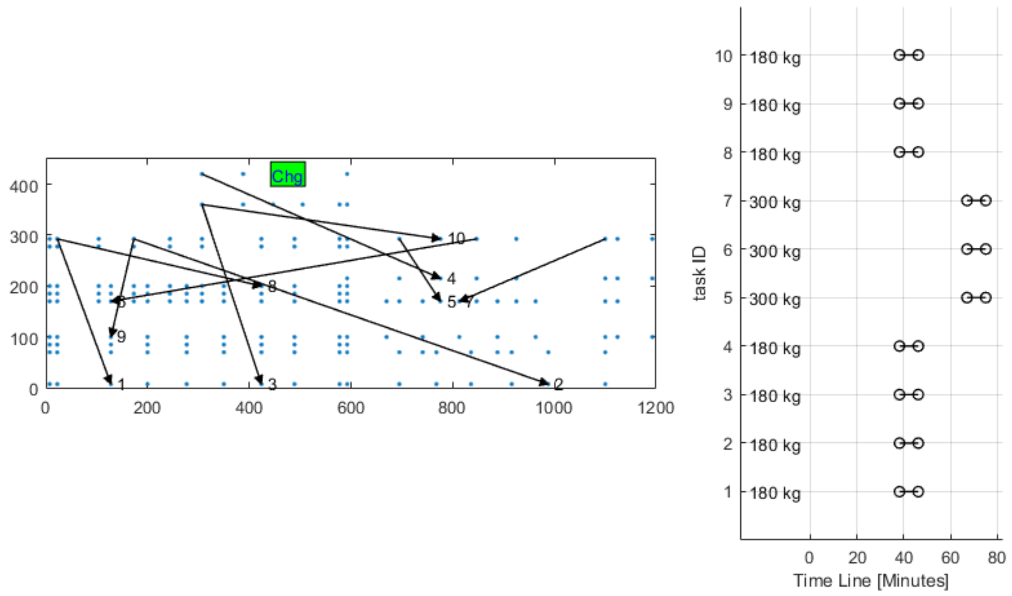


Figure 16 Task Visualization for All Feasible Robot ID and Robot Types

Here the first case, task [5,10] will be used for simple illustration. Figure 17should be interpreted in the same manner as Figure 16, except being highlighted in yellow.

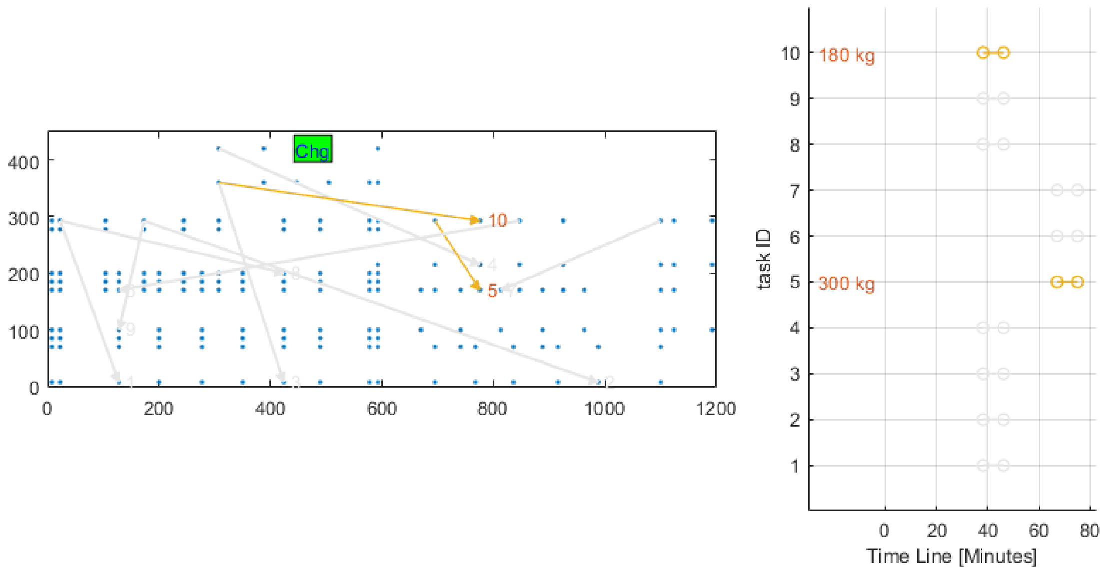


Figure 17 Visualization for Task [5,10]

The left-hand side of Figure 18 is the plot of robots' spatial movement, and it depicts the route traveled by an assigned robot, where directions are marked by tiny arrows. From there, it can be concluded that in order to finish task [5,10], the robot would have to follow a certain visiting sequence:

1. pickup point of task 10 (tail of arrow 10)
2. pickup point of task 5 (tail of arrow 5)
3. drop-off point of task 10 (head of arrow 10)
4. drop-off point of task 5 (tail of arrow 5)

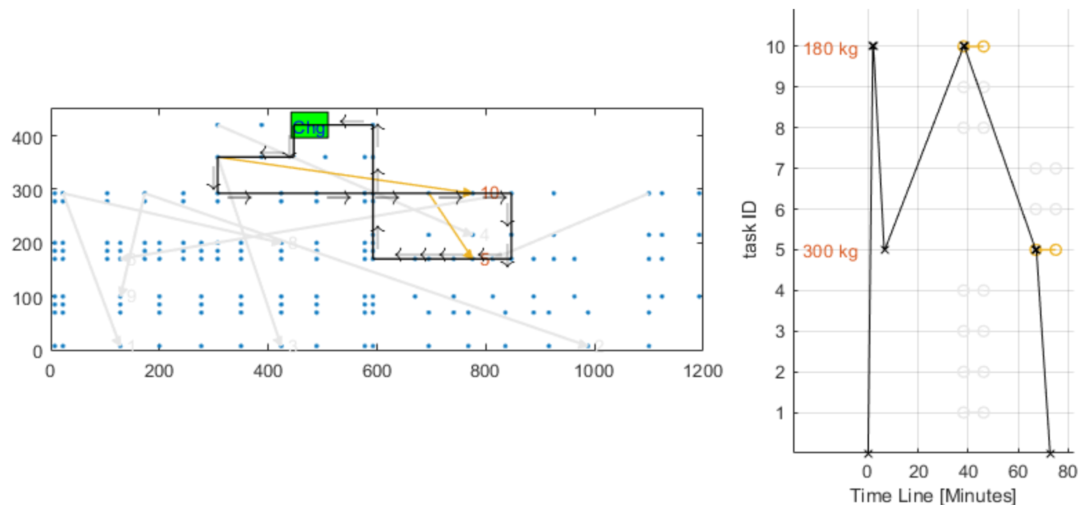
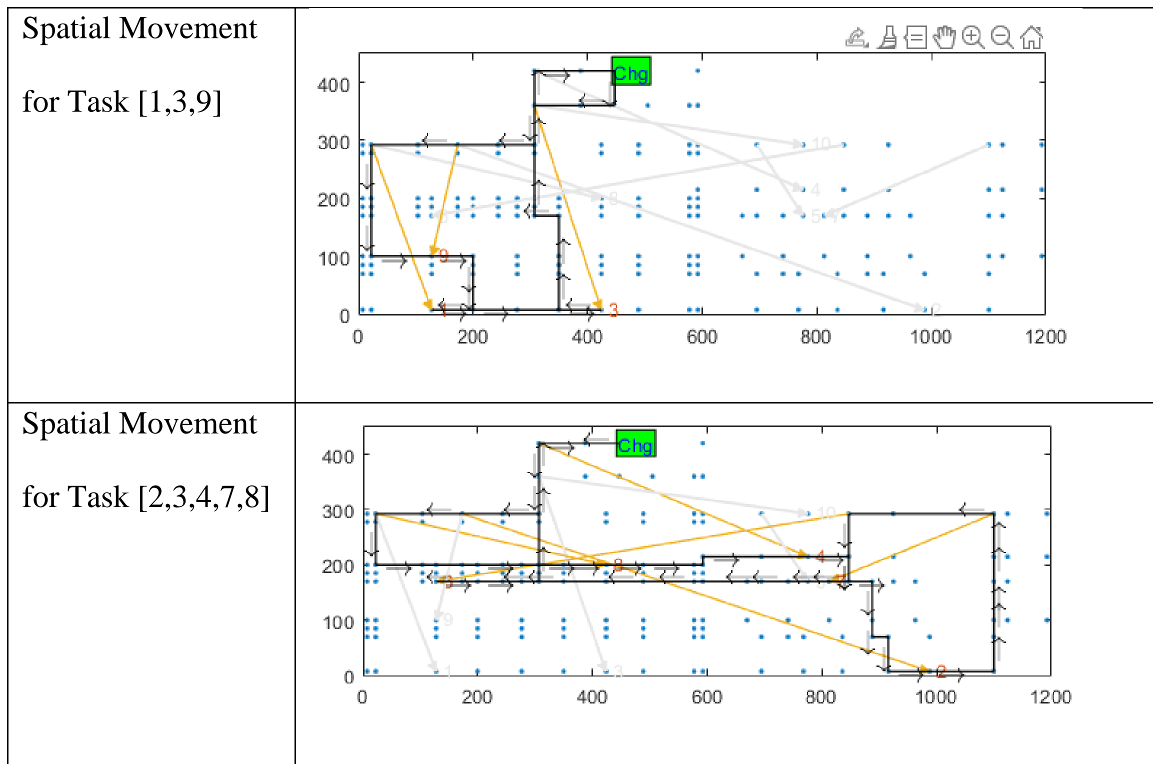


Figure 18 Plot for Spatial Movement and Time Series for Task [5,10]

Such a sequence is validated by the time plot on the right-hand side, where cross represents pick up and "O" represents drop off. indicates the situation of the second case, which a similar layout is shown below.

In Table 10, spatial plot for the task case [1,3,9] and [2,3,4,7,8] is shown, and from here, it's evident that the complexity of traveled route increased as the number of tasks increased.

Table 10 Plot of Spatial Movement for Task [1,3,9] and [2,3,4,7,8]



Back to case of task [5,10], plots of the evolution of remaining cargo capacity and volume capacity be helpful for introducing a visual understanding of shear amount of available capacity, and volume capacity. Such plots were generated through storing each robot's distinctive mass limits and volume limits and had it subtracted by originally defined mass and volume from task list.

The remaining cargo capacity and volume capacity for carrying out assigned tasks [5,10] is shown in Figure 19, and both volume and cargo capacity have been stored to their original value, and such a phenomenon is reasonably explained by the fact that all cargo have been dropped off at drop-off locations. Recall in section 3.1, only remaining cargo

mass capacity after finishing previous exploration will be used for constructing the X matrix, and volume capacity evolution is plotted for visualization only.

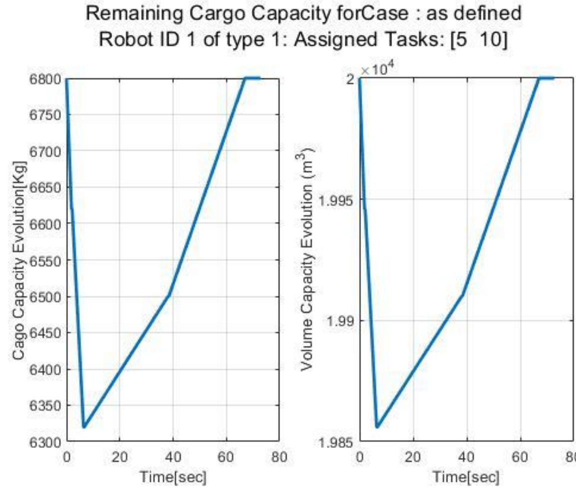


Figure 19 Evolution of Cargo & Volume Capacity for Robot ID 1

3.2.2: State 2: Euclidian Distance between Task Location and Robot's Route's Edge.

Another required information for using the multivariate linear regression method is the shortest distance between each assigned task and all edges of its corresponding task's route, also known as the Euclidean distance.

In section 3.2.2, a method for computing the shortest Euclidean distance [14] between the location of a task (pick up or drop off) and edge of route traveled by robots is presented. A detailed solving procedure is included in Appendix.C. Through running this procedure under the same tasks [5,10] and route traveled for task [5,10], Table 11 was obtained.

From Table 11, state 2 can be computed as sum of shortest distance between robot's traveled route and all remaining task locations. In Table 11, all remaining tasks points are highlighted in light blue, and task [5,10] are highlighted in light red.

Table 11 Sample Euclidean Distance between Each Task Point and Each Edge

The shortest distance between point 1 and all edges is 284.5, with edge index is 3
The shortest distance between point 2 and all edges is 336.5501, with edge index is 3
The shortest distance between point 3 and all edges is 134, with edge index is 3
The shortest distance between point 4 and all edges is 215.1447, with edge index is 12
The shortest distance between point 5 and all edges is 0, with edge index is 1
The shortest distance between point 6 and all edges is 234.0908, with edge index is 17
The shortest distance between point 7 and all edges is 0, with edge index is 1
The shortest distance between point 8 and all edges is 38.0088, with edge index is 13
The shortest distance between point 9 and all edges is 0, with edge index is 6
The shortest distance between point 10 and all edges is 0, with edge index is 13
The shortest distance between point 11 and all edges is 0, with edge index is 10
The shortest distance between point 12 and all edges is 216.9038, with edge index is 3
The shortest distance between point 13 and all edges is 253, with edge index is 10
The shortest distance between point 14 and all edges is 0, with edge index is 14
The shortest distance between point 15 and all edges is 284.5, with edge index is 3
The shortest distance between point 16 and all edges is 92.5, with edge index is 5
The shortest distance between point 17 and all edges is 134, with edge index is 3
The shortest distance between point 18 and all edges is 262.8636, with edge index is 3
The shortest distance between point 19 and all edges is 0, with edge index is 1
The shortest distance between point 20 and all edges is 0, with edge index is 8

3.3: Implementation of Multivariate Regression Method

At each branching instance, for each robot in the fleet, the remaining cargo capacity after having completed the previously assigned tasks and the total Euclidean distance to the remaining pickup and delivery locations is found. These form the states of the fleet at the branching instance. From each branch where a robot has been assigned the task, after the B&B algorithm progresses until the leaf nodes, the costs that are found are then saved as

the result of that task assignment. The fleet state and the costs are saved as the explorations proceed and after sufficient data has been collected, multivariate regression can be used to compute the approximate costs for future exploration. The obtained costs are used to order the robots for exploration, forming the ranking heuristic.

A computational experiment is created in which a fleet of 5 robots were assigned varying number of tasks to be completed. The previously implemented B&B algorithm that used a breadth first search is compared with the proposed method that uses the ranking heuristic.

The evolution of the best-found solutions are shown in Figure 20 and Figure 21for 12 and 13 number of tasks to be completed. The red dotted line corresponds to the case where the multivariate linear regression method was employed, and the blue line represents the scenario where the breadth first search was used. It is clear that by utilizing the previous information about costs obtained from the branching decision, it is possible to find better solutions quicker than with the breadth first search method.

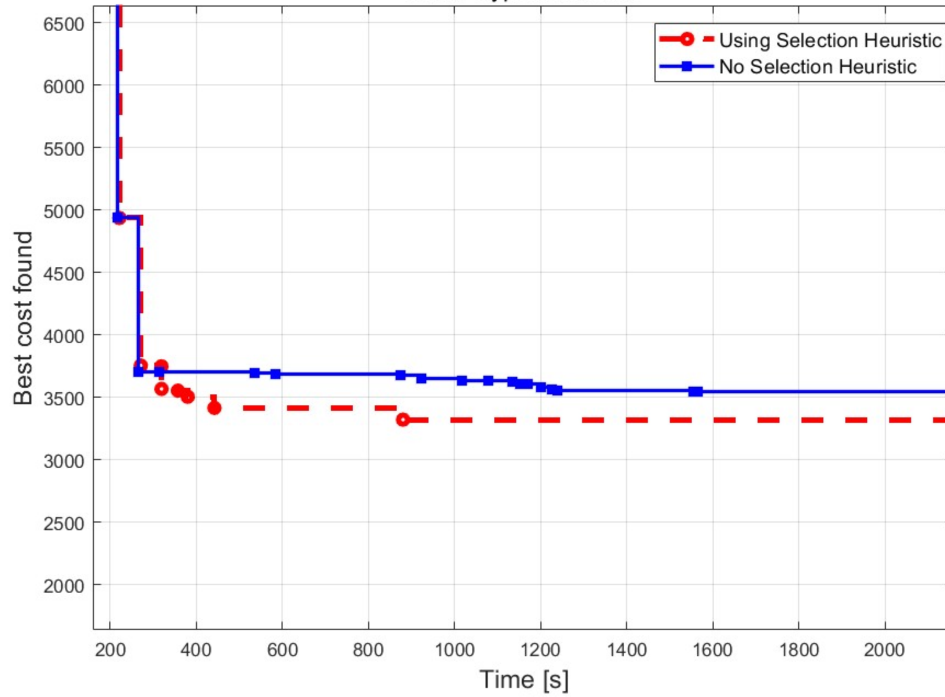


Figure 20 (a) 12 tasks assigned to the fleet

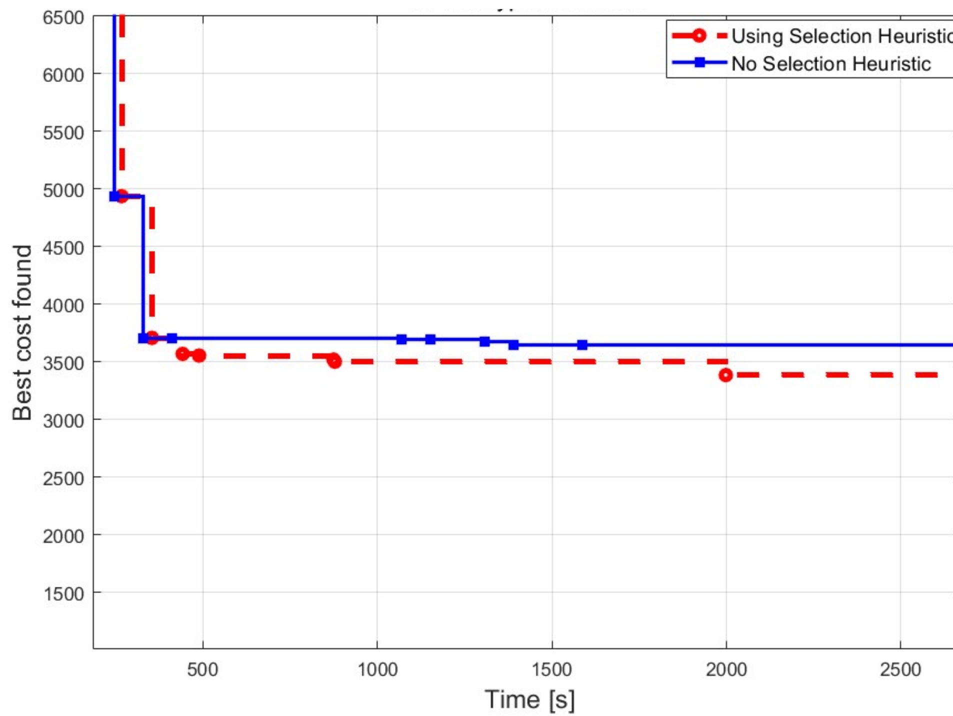


Figure 21 (b) 13 tasks assigned to the fleet

Chapter 4: Conclusion

As part of this research, 5 different robot models were developed to capture the energy and time consumption of different types of autonomous mobile robots for both traversal and pivoting movements. The energy and time provided by these models were included in a large-scale optimization problem with the objective of minimizing energy costs when the fleet of robots completes assigned tasks.

This optimization problem was previously solved using a breadth-first search within a Branch & Bound algorithm that ordered the exploration of tasks using a cheapest-first approach. A data-driven ranking heuristic was developed that uses leaf-node costs from previous explorations to approximate the solution cost when assigning a task to a robot. After successful implementation to order task exploration, computational experiments showed that the developed data-driven approach was able to find better solutions to the problem quicker than the cheapest-first approach.

Chapter 5: Future work

The fleet composition optimizer tackles a large-scale optimization problem, which has been developed for a high-performance computing environment to achieve fast computation results. By utilizing multiple processing cores, the algorithm runs much faster, enhancing the efficiency of the optimization process. Although the developed ranking heuristic has been implemented successfully in a single-core version, adapting it to the parallel processing framework requires some adjustments in the architecture.

Specifically, it is necessary to ensure that the information obtained from each processing core can be accessed and utilized by every processor. This adjustment ensures the availability of cost data for the multivariate regression method to maximize the benefits of this approach across processors. This is expected to provide significant benefits in terms of time and cost savings, and overall computational effectiveness.

Reference

[1] Fragapane, Giuseppe, et al. "Increasing flexibility and productivity in Industry 4.0 production networks with autonomous mobile robots and smart intralogistics." Annals of operations research (2020): 1-19.

[2] Goutham, M., Boyle, S., Menon, M., Mohan, S., Garrow, S., Stockar, S. “Path Planning through a Waypoint Sequence" IEEE Robotics and Automation Letters.

[3] Clausen, Jens. "Branch and bound algorithms-principles and examples." Department of Computer Science, University of Copenhagen (1999): 1-30.

[4] Nilsson, Christian. "Heuristics for the traveling salesman problem." Linkoping University 38 (2003): 00085-9.

[5] Goutham, M., Boyle, S., Menon, M., Mohan, S., Garrow, S., Stockar, S. “Adapting convex-hull based TSP heuristics for the Vehicle Routing Problem." IEEE Transactions on Intelligent Transportation Systems.

[6] Petrov, "Multi-criteria selection of industrial robots: modelling users' preferences in combined AHP-Entropy-TOPSIS," 2022 5th International Conference on Computing and Informatics (ICCI), 2022, pp. 126-131, doi: 10.1109/ICCI54321.2022.9756084.

[7] Racz, Sever-Gabriel, et al. "Mobile Robots—AHP-Based Actuation Solution Selection and Comparison between Mecanum Wheel Drive and Differential Drive with Regard to Dynamic Loads." Machines 10.10 (2022): 886

[8] Breaz, Radu Eugen, Octavian Bologa, and Sever Gabriel Racz. "Selecting industrial robots for milling applications using AHP." Procedia computer science 122 (2017): 346-353.

[9] Zhang, Nuo, et al. "Labelling robots selection based on AHP and TOPSIS." UPB Scientific Bulletin, Series D: Mechanical Engineering 82.3 (2020): 29-40.

[10] Horňáková, Natália, et al. "AHP method application in selection of appropriate material handling equipment in selected industrial enterprise." Wireless Networks 27.3 (2021): 1683-1691.

[11] Abdulghany, A. R. (1970, January 1). Generalization of parallel axis theorem for rotational inertia. American Association of Physics Teachers. Retrieved April 1, 2023, from https://aapt.scitation.org/doi/10.1119/1.4994835

[12] Martins, Felipe & Sarcinelli-Filho, Mário & Carelli, Ricardo. (2017). A Velocity-Based Dynamic Model and Its Properties for Differential Drive Mobile Robots. Journal of Intelligent & Robotic Systems. 85. 10.1007/s10846-016-0381-9.

[13] Phillips, J.G., 2000. Mechatronic design and construction of an intelligent mobile robot for educational purposes. Master of Technology Thesis, Massey University, Palmerston North, New Zealand, pp: 150

[14] Sunday, D. (2001). Practical Geometry Algorithm. Geometry Algorithms Home. Retrieved March 11, 2023, from http://www.geomalgorithms.com/index.html

Appendix A. Model

Analytical Hierarchy Process (AHP) Analysis:

Analysis was done for the AHP method is primarily conducted through coding via MATLAB.

```
% Subject:            Analytical Hierarchy Process
% Date:               9/6/2022
% Name:               Sun Siyuan
%
% Notes:
%Logically, Consistency ratio (CR) should be calculated at first place as the
%calculation of matrix will be meaningful only when it's reasonably
%consistent, such that CR < 0.1
%However, here we will be calculating criteria weights first and then
%perform consistency test.

clc
clear
%input matrix
%This is just an example for a pair-wise comparison matrix. In our actual research, we
don't have the leisure for having expert assigned criteria weights for each specific
attributes (for example, efficiency, battery size, capacity, weight, and max speed). The
lack of such values prohibited us from developing a pair-wise comparison matrix. Thus,
another method is being considered, which is the data-driven method that utilizes data
from past exploration.
A=[1     5    4    7;
   0.2    1    0.5  3;
   0.25   2    1    3;
   0.14 0.33 0.33  1]
Sum_A=sum(A,1)%Sum up all elements on each column
n=size(A,1)%find the size of our matrix
Norm_A=A./repmat(Sum_A,n,1)%Find the normalized pairwise matrix A
CW_A=mean(Norm_A,2) %Calculate Criteria Weights
%calculating the Consistency:
WS_A=A*CW_A %Calculating Weighted Sum Value
Ratio=(WS_A)./(CW_A)
Lamda=mean(Ratio)
Lamda_max=max(Lamda) %lamda max
Consistency_Index=(Lamda_max-n)/(n-1)

if n==1
```

```
RI=0
elseif n==2
    RI=0
elseif n==3
    RI=0.58
elseif n==4
    RI=0.9
elseif n==5
    RI=1.12
elseif n==6
    RI=1.24
elseif n==7
    RI=1.32
elseif n==8
    RI=1.41
elseif n==9
    RI=1.45
elseif n==10
    RI=1.49
else
    fprintf('Something went wrong, please do check your matrix.')
end

    Consistency_ratio=Consistency_Index/RI %calculate consistency ratio
    %determine if it can pass the consistency test
    if Consistency_ratio < 0.1
        fprintf('This matrix is reasonably consistent.')
    else
        fprintf('Further adjustments is required in order to have matrix being consistent')
    end
```

Breadth First Search

Major attributes of breadth first search here are listed in the box below as well as their implications.

Pro	Con
Completeness: BFS guarantees that all nodes at a distance k from the source vertex will be visited before any node at distance k+1 is visited. Therefore, if a solution exists at a certain depth in the graph, BFS will find it.	**Space complexity**: BFS requires a lot of memory to keep track of the visited nodes and the nodes in the queue. This can be a problem for very large graphs.
Shortest path: BFS is guaranteed to find the shortest path between the source vertex and any other vertex in an unweighted graph.	**Time complexity**: The worst-case time complexity of BFS is $O(\|V\| + \|E\|)$, where $\|V\|$ is the number of vertices and $\|E\|$ is the number of edges in the graph. This can be slow for very dense graphs.
Implementation simplicity: BFS is easy to implement and can be used as a building block for more complex algorithms.	**Not optimal for weighted graphs**: BFS does not guarantee the shortest path in a weighted graph. For weighted graphs, Dijkstra's algorithm or A* algorithm are more appropriate.
Finding all connected components: BFS can be used to find all the connected components in an undirected graph.	**Multiple solutions**: BFS can find multiple solutions if they exist at the same depth, which may not be desirable.

Appendix B. MATLAB Code

```
function: p = robot_parameters (type)
 for i = 1: 5
          if type =1
         store all parameters for type 1 robot as fields for structure array p
         elseif type = 2
        store all parameters for type 2 robot as fields for structure array p
         elseif type = 3
         store all parameters for type 3 robot as fields for structure array p
         elseif type = 4
         store all parameters for type 4 robot as fields for structure array p
         else
         store all parameters for type 5 robot as fields for structure array p
end
end
```

Section 2.3:

File name: *LauchFile.m*

```
%% Optimizer Parameters:
%Initialize and name folders where file sharing occurs:
%Import Problem Parameters:
         Import number of tasks, number of available robot types, and maximum number
         of robots for each available type.
% Initialization & Problem Parameters
         Map = definition_map (2) %Generating structure array "Map" through calling
         user defined function "definition_map", with 2 stands for the map type defined by
         Ford.
         [TaskList] = definition_tasks(2,numTasks,Map); %Generating structure array
         "TaskList" through calling user defined function "definition_tasks". "2" here is
         the map type defined by ford.
         [Map, Robots] = definition_robots(Map, TaskList); %Results from the previous
         two line are being used as inputs for another user defined function
         "definition_robots" to update current "Map" structure and generate new "Robots"
         structure, which has been briefed in the beginning part of section 2.3.

Function  [Map, Robots] = definition_robots(Map, TaskList);
```

```
Generate path and distance through calling a user-defined function
"pathcostMatrix" where it takes same input as definition_robots.
for i=1:size (number of available robot types)
        loaded= robot_parameter(i)  %User defined function "robot_parameter"
        has been defined in section 2.1.1, and here all corresponding value for
        robot type i is stored to variable "loaded ".
        %Extract and store all variables into desired structure array "Robots".
        %Such that:
        Robots(i).massLimit      = loaded.massLimit              % mg
        Robots(i).volumeLimit = loaded.volumeLimit;              % m^3
        Robots(i).batterySize    = loaded.BatteryCapacity;       % kWh
        Robots(i).chargePower = 2*Robots(i).batterySize;         % kW
        Robots(i).kerbWeight   = loaded.EmptyMass;               % kg

        %Variable Initialization:
        cargoMass=0;
        energyMatrix =zero(size(path)); timeMatrix=zero(size(path));
        for i= 1: size(path,1)
            for i= 1: size(path,2)
                pathDistance = distance{i, j} % "distance" is a user defined
        function which computes the shortest path distance between nodes pairs in
        a digraph, and results are stored as variable "pathDistance".
                [E,T]=robot_cost (pathDistance, 0, i, cargoMass)
                energyMatrix=sum(E);
                timeMatrix = sum(T);
            end
        end
        %Store computed matrix into desired structure array "Robots"
        Robot(i).costMatrix.Energy = energyMatrix;
        Robot(i).costMatrix.Time    = timeMatrix;
 end
end
```

Values for energy cost and time cost are calculated through calling the user-defined function "robot_cost", which its algorithm is entirely based on the kinematics-dependent

equations shown in section (2.1.2)-(2.1.4). Likewise, better is a simplified pseudocode written for " *robot_cost":*

```
Function  [E, T] = robot_cost(Distance, Theta, Type, cargoMass)
            Units Conversion and taking absolute value
            Make sure both Distance and Theta vector shared the same length
% Covering Section 2.1.1: Updating Robot Dynamics based on Cargo Mass
          obj.params= robot_parameters(Type);  %Importing parameters through calling
user defined function "robot_parameter", and having it stored in a structure array "obj".
          Calculate M_dynamic, I_cg, I_0, and F_rr
          if type = 3 % Number "3" here representing the type of omnidirectional robot
             Calculate μ and N
          else          % For non-omnidirectional robot
             Calculate μ and N
          end
             Calculate F_L and M_f

% Covering Section 2.1.2: Equation Set for Traversal Traveling Portion
% Initialization for Energy and Time Cost
          E = 0; T = 0;
          if any(Distance>0)
             Calculate maximum velocity, acceleration and distance traveled for non-
          coasting phase
          end

% Covering Section 2.1.3: Equation Set for Pure Rotation Portion
if type == 3
   if any(Theta>0)
      Calculate Rotating force F_w, rotating moment M_r, maximum angular acceleration
      α_max, maximum angular velocity ω_max, and angle rotated for non-coasting phase
      θ_a
      Calculate energy and time cost for acceleration, deceleration and coasting    phase.
else
    if any(Theta>0)
```

Calculate Rotating force F_w, rotating moment M_r, maximum angular acceleration α_{max}, maximum angular velocity ω_{max}, and angle rotated for non-coasting phase θ_a.
Calculate energy and time cost for acceleration, deceleration and coasting phase.
end
Units conversion for E & T
end (end function)

Equations for the above can be found from section 2.1.2 to section 2.1.3. At this stage, all variables listed in table 7 all have been computed and systematically stored inside the structure array "Robots" for data extraction.

Section 3.2.2:

1. ***Algorithm for finding Edge_info***

```
for i = 1: number of best assignments
    if task is empty; end;
      update Children of assigned task
      update robot Parameters based on
      update assigned Task ID
      compute (*update) node sequence and create sequence vector
      generate plot of base figure
     calling "staticImage_spatial" to get Edge_info;
    Edge_info{i}=Edge_info;
end
```

where:

Number of best assignments: a subfield from a structure array "solution," derived from previous coding work. It contains critical information, such as cost and assigned tasks for the type of robot of each best solution. In this case, the total number of best solutions is 8, and therefore, 8 iterations can be expected.

Node sequence: A structure array that contains valuable information, such as best cost, worst cost, and best sequence, and it will be used for one of the inputs for the user defined function: *StaticImage_spatial.*

StaticImage_spatial: A user defined function takes input of robot parameters, node sequence, task list, map and output the Edge_info matrix.

Edge_info: A matrix that contains all required information to compute one single edge for analyzing the tour path. Edge_info' s size will be m x n, where m= 4, and n = number of edges at that specific task. The reason m = 4 is because it takes 4 points in total to form two points' coordinates. An example of such is listed in the picture below:

2. *Function for "dist_point_to_line_segment"*

```
Function: dist = dist_point_to_line_segment(point, v1, v2)

% Calculate distance from point to line segment defined by v1 and v2
% point: a 2-element vector representing the point (x, y)
% v1: a 2-element vector representing the start of the line segment (x1,  y1)
% v2: a 2-element vector representing the end of the line segment (x2, y2)
% Returns:
% dist: the distance from point to the line segment
% Reference: http://geomalgorithms.com/a02-_lines.html
% calculate the length and direction of the line segment
        len = norm(v2 - v1);
        dir = (v2 - v1) / len;
% calculate the vector from v1 to the point
        vec = point - v1;
% calculate the projection of vec onto the line segment
        proj = dot(vec, dir);
% clamp the projection to lie within the line segment
        proj = max(0, min(proj, len));
% calculate the point on the line segment closest to the point
        closest = v1 + proj * dir;
% calculate the distance from the point to the closest point on the line segment
        dist = norm(point - closest);
end
```

3. Algorithm for constructing D matrix.

```
D = zeros (number of points, number of edges) %Initialize an empty D matrix:
for f = 1: size(D,1)
      for g = 1: size(D,2)
            v1 = V1(g, :); %V1 stores all starting points coordinates
                  if g == number of edges
                      v2 = V2(1, :); %V2 stores all ending points coordinates
                  else
                      v2 = V2 (g+1, :);
                  end
                  p = TaskList.TaskCoordinates(f, :);
                 d = dist_point_to_line_segment (p, v1, v2) %Compute shortest distance
                D (f, g) = d %Entering calculated distance value into empty D shell
       end
  end
```

4. Algorithm for integration into previous work

```
%Initialize empty cell structure:
D_cell = cell (1, number of best assignment);
D_cell_min = cell (1, number of best assignment);
Edge_index_cell= cell (1, number of best assignment));

for i = 1: number of best solutions
           .
           .
           .
                 D_cell{i}=D;
           %Find shortest distance between each point and all edges and display
                [D_min, Edge_index]=min (D, [],2); %D with both each row's minimum
           value and corresponding index found
                Edge_index_cell{i}=Edge_index;
               D_cell_min{i}=D_min;
end
```

Section 3.3:

1. ***Logistic for recursive function "BnB_TaskAssignment"***

```
function
outState = BnB_TaskAssignment(inState,  Fleet, Robots, TaskList,
remainingTaskIDs,opSettings,Map)
% Recursive code implementation of the Branch & Bound algorithm for finding
% optimal task assignment.
%   This implements a depth first search, and only saves the best cost &
%   sequence. It also saves the worst cost for the MCTS.

%%
deltaRobots = Fleet.deltaRobots;
TaskID_next = remainingTaskIDs(1);

%% Find and sort valid expansion, and Find minimum Distance and remaining mass
capacity here:
[childrenRobotIDs,ChildrenCost,AllRobotIDs,xVector] =
validSortedExpansions(inState,TaskID_next,Robots,deltaRobots,TaskList, opSettings,
Map); % Function in this one is where equations for calculating 2 states being
integrated!!
     yVector = NaN(1,length(AllRobotIDs));   %Initialize. Do not sort.


 %Initiate Search
if childrenRobotIDs is not empty
     outstate = instate
  for i = 1: length( childrenRobotIDs)
     outState.AssignmentSoFar(childrenRobotIDs(i)).assignedTasks
=  [inState.AssignmentSoFar(childrenRobotIDs(i)).assignedTasks, TaskID_next];
     outState.AssignmentSoFar(childrenRobotIDs(i)).cost = ChildrenCost(i);
[fixedCostWorkingRobots,workingRobots] = fixedCost(outState,AllRobotIDs,Robots);

%compute fixed cost
      if fixedCostWorkingRobots < outState.bestCost
         if there's >= 1 delta robot
           [outState] =
nonDeltaOpCost (outState, TaskList, workingRobots, deltaRobots, Robots, opSettings);
       operationalCost = sum([outState.AssignmentSoFar.Cost]); %operational cost
computing
       outState.costSoFar = fixedCostWorkingRobots + operationalCost;
```

```
        outState = par_updateBestSolution(outState,opSettings); % Check shared folders
to find best cost (MCTS & FCO)
            if  outState.costSoFar < outState.bestCost
              if TaskID_next < TaskList.numTasks
                 if verbosity > 0, printData(...), end
               outstate = BnB_TaskAssignment(.........)  %recursive entry
               bestAssignment = outState.bestAssignment;
               bestCost = outState.bestCost;
               outState = inState;
               outState.bestAssignment = bestAssignment;
               outState.bestCost = bestCost;
              else %If TaskID_next >= TaskList.numTasks
              bestAssignment = outState.AssignmentSoFar;
              bestCost = outState.costSoFar;
              parsave_FCO(opSettings.FCOcache, bestCost, bestAssignment)
              if opSettings.verbosity >0,
              outState.bestCost = bestCost; printData(outState,'improved'), end
              outState = inState;
              outState.bestCost = bestCost;
              outState.bestAssignment = bestAssignment;
             end
           else   % if cost so far >  best cost
              bestAssignment = outState.bestAssignment;
              bestCost = outState.bestCost;
              outState = inState;
              outState.bestAssignment = bestAssignment;
              outState.bestCost = bestCost;
              continue
           end
       else % no delta robot
             if TaskID_next < TaskList.numTasks % Recursive re-entry
               if opSettings.verbosity >0,
                 printData(outState,'noDelta_intermediate')
               end
             outState = BnB_TaskAssignment(..., remainingTaskIDs(2:end), ...);
             %Recursive Entry for no delta robot
             bestAssignment = outState.bestAssignment;
             bestCost = outState.bestCost;
```

```
                        outState = inState;
                        outState.bestCost = bestCost;
                        outState.bestAssignment = bestAssignment;
                        else
                        if opSettings.verbosity >0, printData(outState,'noDelta_leaf'), end
                        bestAssignment = outState.bestAssignment;
                        bestCost = outState.bestCost;
                        outState = inState;
                        outState.bestCost = bestCost;
                        outState.bestAssignment = bestAssignment;
                        end
                end
            else % fixed cost > or = best cost
                if opSettings.verbosity >0, printData(outState,'existenceFutile'), end
                bestAssignment = outState.bestAssignment;
                bestCost = outState.bestCost;
                outState = inState;
                outState.bestCost = bestCost;
                outState.bestAssignment = bestAssignment;
            end
    end
else
No valid children
end
```

2. Function for "ValidSortedExpansions"

```
function
[childrenRobotIDs,ChildrenCost,AllRobotIDs, xVector] =
validSortedExpansions(inState,TaskID_next,Robots,deltaRobots,TaskList, opSettings,
Map)

% Find valid Expansions & Only delete robots that are completely incapable of that task
checkMassLimit=[Robots.massLimit]>=
TaskList.mass(TaskID_next)*ones(size(Robots));

checkVolLimit=[Robots.volumeLimit]>=
TaskList.volume(TaskID_next)*ones(size(Robots));

cargoLimitationsOK =  checkMassLimit & checkVolLimit;
```

```
AllRobotIDs=1:size(inState.AssignmentSoFar,2);%The fields make up the Robot IDs
childrenRobotIDs =
AllRobotIDs(ismember([inState.AssignmentSoFar.Type],find(cargoLimitationsOK ==
1)));

% Time based deletions(as we don't want disqualified robots)
        pickupTimeStart =  TaskList.timeStart(TaskID_next);
        dropoffTimeEnd = TaskList.timeEnd(TaskID_next+TaskList.numTasks);
        deleteChildren = zeros(1,length(childrenRobotIDs));
  for i = 1:length(childrenRobotIDs)  %Remove any time-infeasible pickups
      robotType = inState.AssignmentSoFar(childrenRobotIDs(i)).Type;
      initPos = TaskID_next + 1;
      % Add 1, because station is at 1st index of cost matrix
      finPos = TaskID_next+ TaskList.numTasks + 1;
      % Add 1, because station is at 1st index of cost matrix
        minTimeToMove = Robots(robotType).costMatrix.time(initPos,finPos);
        if minTimeToMove + 2*TaskList.exchangeTime > dropoffTimeEnd -
pickupTimeStart
           deleteChildren(i)=1;
        end
  end

     childrenRobotIDs(logical(deleteChildren)) = [];

%% Calculate costs to sort children:
ChildrenCost        = NaN(1,length(childrenRobotIDs));
addedemploymentCost = NaN(1,length(childrenRobotIDs));

xVector = NaN(1,2*length(AllRobotIDs));    %This is never sorted! Maintain robot IDs
here
  %Initialize
  for i = 1:length(childrenRobotIDs) %Doing this before the Initiate Search section
allows cheapest-first ordering
      rID = childrenRobotIDs(i);
        if ismember(rID,deltaRobots) % Only compute costs if the child robot is a delta
robot
robotParameters = Robots(inState.AssignmentSoFar(rID).Type);
ChildrenassignedTasks = [inState.AssignmentSoFar(rID).assignedTasks, TaskID_next];
assignedsubTaskIDs = [ChildrenassignedTasks , ChildrenassignedTasks +
TaskList.numTasks];
opSettings.tic2 = tic;
nodeSeq = BnB_NodeSequence([],robotParameters,assignedsubTaskIDs, TaskList,
opSettings);
```

```
ChildrenCost(i) = nodeSeq.bestCost;
            if ~isempty(nodeSeq.bestSeq)
[dist2RemTasks,cargoCapRemaining]
=insertionStates(nodeSeq,TaskList,Map,ChildrenassignedTasks,robotParameters);
xVector(2*(rID-1)+1:2*(rID-1)+2) = [dist2RemTasks,cargoCapRemaining];
            else
              xVector(2*(rID-1)+1:2*(rID-1)+2) = [NaN,NaN];  %Only 2 states per robot
<< remove HARDCODING LATER
            end
         else
            continue   % The other robot costs are computed only if useful
         end
         addedemploymentCost(i) = ChildrenCost(i) + robotParameters.purchaseCost;
   end
% Cheapest First Ranking is done considering the cost of employment
% Reported costs are operational costs
   [~,Index] = sort(addedemploymentCost);
      childrenRobotIDs = childrenRobotIDs(Index);
      ChildrenCost = ChildrenCost(Index);
End (end of function "ValidSortedExpansion")
```

3. **Function for "InsertionStates"**

```
%This function shows detailed work of how previously developed algorithm for
computing 2 states is used for generating the X matrix in "validSortedExpansion".
[dist2RemTasks,cargoCapRemaining] =
insertionStates(nodeSeq,TaskList,Map,ChildrenassignedTasks,robotParameters)
   taskcoordinate_x=[];taskcoordinate_y=[];
TaskRemaining = setdiff([1:TaskList.numTasks],ChildrenassignedTasks);
%setdiff(A,B)-->find data in A that is not in B
   for k=1:length(TaskRemaining)
      taskcoordinate_x = [taskcoordinate_x
Map.Corner_Coordinates(TaskList.position(k),1)
Map.Corner_Coordinates(TaskList.position(k+TaskList.numTasks),1)];
      taskcoordinate_y = [taskcoordinate_y
Map.Corner_Coordinates(TaskList.position(k),2)
Map.Corner_Coordinates(TaskList.position(k+TaskList.numTasks),2)];
   end
   TaskList.TaskCoordinates = [taskcoordinate_x' taskcoordinate_y']; %matrix two
columns representing x and y coordinates.

    edge_info = staticImage_spatial(robotParameters,nodeSeq,TaskList,Map);
    %Edge_info's each column contains all necessary info to compute the edge
    Shortest_Distance = EuclideanDistance(TaskList,edge_info);
```

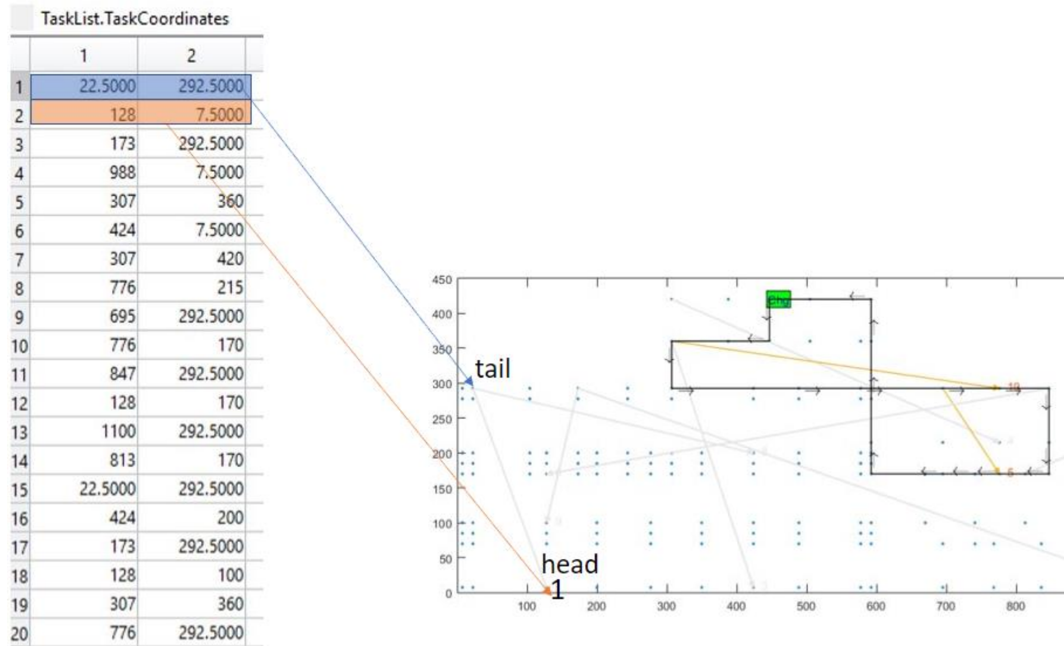
```
    %Calling UserDefined Function for finding Shortest Distance
  D_min = min(Shortest_Distance,[],2);
  dist2RemTasks = sum(D_min,1);
% Here the Sum of shortest distance between each task location and all edges for the
current i, this should return only 1 number.
        %--------State Remaining Cargo (& Volume Capacity)
        sequence = [nodeSeq.bestSeq];
        cargoCapRemaining = robotParameters.massLimit;
        volumeCapRemaining= robotParameters.volumeLimit;
        %It's calculated yet won't be used.
         for j = 1:length(sequence)
            cargoCapRemaining = [cargoCapRemaining cargoCapRemaining(end)-
TaskList.mass(sequence(j))];
%             volumeCapRemaining =[volumeCapRemaining
volumeCapRemaining(end)-TaskList.volume(sequence(j))];
         end
        cargoCapRemaining = min(cargoCapRemaining);
%          volumeCapRemaining = min(volumeCapRemaining);
end
```

Appendix C. Methods for Computing Euclidean Distance

This research often involves complex scenarios for task arrangements, hence a straightforward way for analysis and computation is desired. Appendix C delineates a detailed procedure for finding the state 2, Euclidean distance, from given tasks and traveled route.

Each task can be interpreted as an arrow, with its head representing the drop-off location and its tail representing the pick-up location. Two variables, "taskcoordinate_x" and "taskcoordinate_y," were created to store the x and y coordinates of all points, respectively. In the current testing scenario, covered in Figure 16, there are 10 tasks under investigation, resulting in a total of 20 points, since each task comprises a head and a tail.

For simplicity, coordinates are stored within the structure array called “TaskList.” Below is an example provided to demonstrate this: In the table on the left-hand side, number 22.5 and 292.5 stand for the coordinate of the tail of the arrow with number 128 and 7.5 stand for the head of arrow.



TaskList.TaskCoordinates

	1	2
1	22.5000	292.5000
2	128	7.5000
3	173	292.5000
4	988	7.5000
5	307	360
6	424	7.5000
7	307	420
8	776	215
9	695	292.5000
10	776	170
11	847	292.5000
12	128	170
13	1100	292.5000
14	813	170
15	22.5000	292.5000
16	424	200
17	173	292.5000
18	128	100
19	307	360
20	776	292.5000

Figure 22 Illustration of the Function of TaskList.TaskCoordinates at Task 1

To analyze the robots' tour path, it's important to gather key information about the route they took. Conceptually, this is approached by collecting information for each edge, formed by two adjacent points, and having them stored in a matrix, called "*Edge_info*", for future use. To achieve this, a simple for-loop has been developed and a pseudocode has been demonstrated below for better comprehension, and code is listed in **Appendix B.: Section 3.2.2:Algorithm for creating *Edge_info*** Figure 23 is the generated example matrix for *Edge_info*:

edge_info | solution | solution.bestAssignment | Map | Map.Corner_Coordinates

4x22 double

	1	2	3	4	5	6	7	8
1	447	447	307	307	307	489	577.5000	[illegible]
2	420	360	360	360	292.5000	292.5000	292.5000	292.5
3	447	307	307	307	489	577.5000	695	[illegible]
4	360	360	360	292.5000	292.5000	292.5000	292.5000	292.5
5								

Figure 23 Example of matrix “*Edge_info*”

This matrix contains all the required information to compute one single edge for analyzing the tour path. Edge_info’ s size will be m x n, where m equals 4, and n equals number of edges at that specific task. The reason m is 4 is because it takes 4 points in total to form two points’ coordinates.

In Table 12, it’s evident to conclude that 22 edges can be expected at this specific scenario since the n here is 22, or we have 22 edges for the current best solution’s route. Meanwhile, the number of rows can be comprehended as the x and y coordinates for both starting and ending points for each edge, respectively.

Detailed information for thoroughly understanding the first edge by inspecting the Edge_info matrix is listed in Table 12:

Table 12 Connotation of each element for the first column of Edge_matrix

Edge_info (1,1) = 447	x coordinates of starting point for edge 1
Edge_info (2,1) = 420	y coordinates of starting point for edge 1
Edge_info (3,1) = 447	x coordinates of ending point for edge 1
Edge_info (4,1) = 360	Y coordinates of ending point for edge 1

The aforementioned information was substantiated by means of meticulous inspection of the plot depicted below, wherein the red line corresponds to the first edge, while the blue line represents the second.

On this basis, it was reasonably inferred that the edge's coordinates are accurately stored as values from the matrix that correspond with the values from the axes of the cartesian coordinate system, shown in Figure 24.

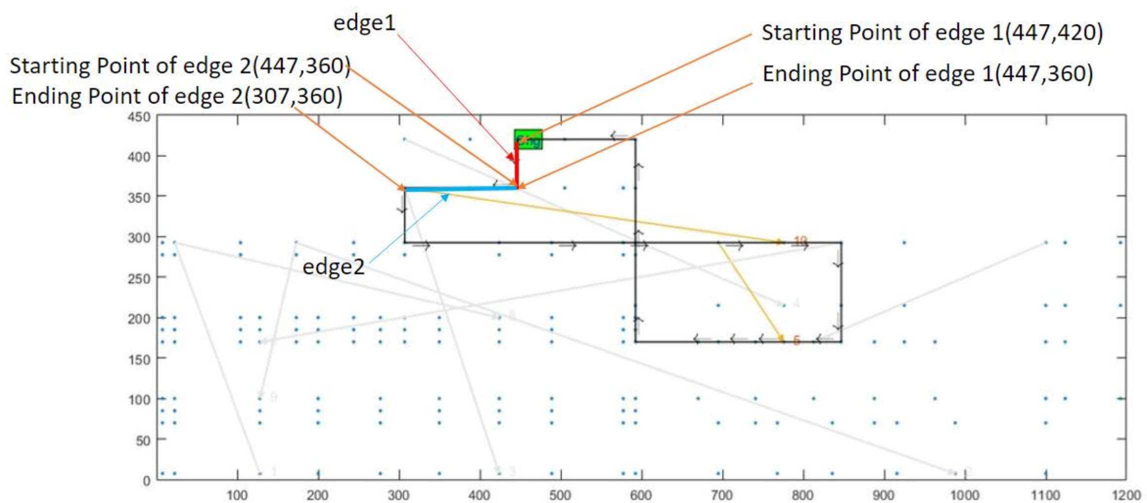


Figure 24 Graphical validation of elements 'placement in Edge_info matrix

The next step is to separated *Edge_info* matrix into two matrices, V_1 *and* V_2, with V_1 contains all starting points' coordinates and V_2 contains all ending points' coordinates, and it serves a sole purpose of better data arrangement.

At this point, each edge has been expressed in a vector form, and I computed the distance between each point and each edge. Analysis upon "Edge_info" has been thoroughly conducted, and to compute the shortest distance, information regarding points is still required. Recall from previous part, we have all tasks' (pick up & drop off) location

stored inside a structure array, TaskList.TaskCoordinates. At this specific scenario, the matrix "TaskCoordinates" has a size of 20 by 2 and this is corresponding to the total number of tasks being equal to 10.

To compute the shortest Euclidean Distance between each task point and each edge, a user-defined function, "*dist_point_to_line_segment*", has been developed for achieving this goal. This user defined function is used to calculate the distance between a point and a line segment using the Euclidean distance formula and it will first calculate the projection of the point onto the line defined by the line segment, and then calculates the distance between the projected point and the original point using the Euclidean distance formula. Inputs along with outputs are listed below, and a simplified pseudocode is provided in **Appendix B: function for "dist_point_to_line_segment"**

Inputs for this function are the following:

1. point: a 2-element vector representing the point (x, y)
2. edge 1: a 2-element vector representing the start of the line segment (x1, y1)
3. edge 2: a 2-element vector representing the end of the line segment (x2, y2)

Outputs for this function are the following:

1. distance: the distance from point to the line segment

Six main calculating procedures of this function are listed below:

1. calculate the length and direction of the line segment.
2. calculate the vector from v1 to the point.
3. calculate the projection of vectors onto the line segment.
4. clamp the projection to lie within the line segment.

5. calculate the point on the line segment closest to the point.

6. calculate the distance from the point to the closest point on the line segment.

Each of the calculated shortest distance will be stored in a matrix, named "*D*", and *D* matrix's size is dictated by the number of points and number of edges. With that been said, in this best solution example, the *D* matrix thus will have a size of 20 by 22, and each *D* matrix element, say *D (a, b)* will be interpreted as the shortest distance between a_{th} point and b_{th} edge. Again, this is accomplished through a simple setup of for-loop, and the corresponding pseudocode is attached in Appendix B. Section 4.2: Algorithm for constructing D matrix.

At this point, a 20 by 22 *D* matrix is successfully calculated. However, what's even better is to filter out the most valuable information from the *D* matrix as well as index of which specific edge for each best solution's route has the shortest distance with each task points. More importantly, this part will be integrated into the first for-loop shown in section 4.2, and the details are listed in ***Appendix B.: Algorithm for integration into previous work.*** Through integrating and running the code above, D_cell_min can provide with valuable information as shown in Figure 25.

D_cell_min

1x8 cell

	1	2	3	4	5	6	7	8
1	20x1 double	20x1 double	[]	[]	[]	[]	20x1 double	[]

Figure 25 Cell array: D_cell_min

One might be wondering why D_cell_min{3,4,5,6,8} are empty, and it is entirely due to the initial testing cases from given condition, or also known as the computed "solution"

array. Recall in Figure 26, detailed task list has only been listed for the first, second and seventh robot ID, and the reason the rest are not being listed is because such information is simply not given. Content of structure array “solution” is shown in both Figure 26 and Figure 27:

solution | D_cell_min

1x1 struct with 2 fields

Field	Value
bestCost	3.4100e+03
bestAssignme...	1x8 struct

Figure 26 Overview of Structure Array Solution

solution | D_cell_min | solution.bestAssignment

solution.bestAssignment

Fields	Type	assignedTasks	cost
1	1	[5,10]	55.1244
2	1	[1,3,9]	76.5989
3	2	[]	0
4	2	[]	0
5	3	[]	0
6	4	[]	0
7	4	[2,4,6,7,8]	28.2644
8	5	[]	0

Figure 27 Overview of Structure Array Solution

From Figure 27, one can conclude that no task(s) have been assigned for subfield 3,4,5,6 and 8. Henceforth, this should explain the emptiness one spotted in D_cell_min{3,4,5,6,8}. To gain a more comprehensive understanding of the findings, the task point, corresponding edge, and their Euclidean distance were correlated and presented in tabular format. Specifically, this was achieved by analyzing the results of the solution.bestAssignment field for field 7. In this particular case, there were a total of 61 edges and 20 task points, which remained constant throughout the analysis. The detailed results are given in Table 13.

Table 13 Example of Tabulated results for point-edge distance

The shortest distance between point 1 and all edges is 0, with edge index is 6
The shortest distance between point 2 and all edges is 162.5, with edge index is 52
The shortest distance between point 3 and all edges is 0, with edge index is 4
The shortest distance between point 4 and all edges is 0, with edge index is 29
The shortest distance between point 5 and all edges is 0, with edge index is 2
The shortest distance between point 6 and all edges is 162.5, with edge index is 48
The shortest distance between point 7 and all edges is 0, with edge index is 1
The shortest distance between point 8 and all edges is 0, with edge index is 20
The shortest distance between point 9 and all edges is 77.5, with edge index is 21
The shortest distance between point 10 and all edges is 0, with edge index is 44
The shortest distance between point 11 and all edges is 0, with edge index is 38
The shortest distance between point 12 and all edges is 0, with edge index is 52
The shortest distance between point 13 and all edges is 0, with edge index is 36
The shortest distance between point 14 and all edges is 0, with edge index is 42
The shortest distance between point 15 and all edges is 0, with edge index is 6
The shortest distance between point 16 and all edges is 0, with edge index is 14
The shortest distance between point 17 and all edges is 0, with edge index is 4
The shortest distance between point 18 and all edges is 70, with edge index is 52
The shortest distance between point 19 and all edges is 0, with edge index is 2
The shortest distance between point 20 and all edges is 71, with edge index is 38

Upon careful examination of the Table 13 listed above, it becomes readily notable that multiple shortest distance between points and edges are zero. While such an observation may instill a sense of concern, it is crucial to note that this phenomenon is directly contributed by the superimposition of many points (each task’s head and tail) on the route’s edges, see in the Figure 28.

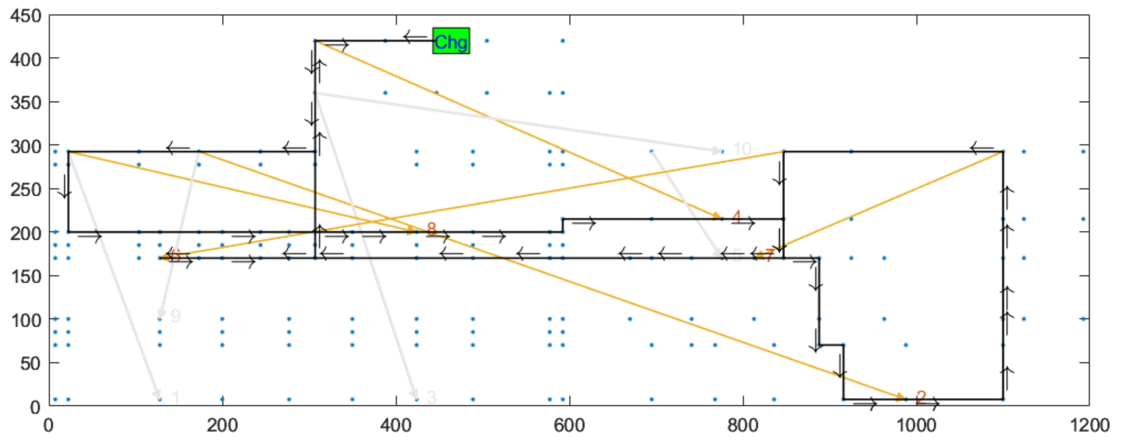


Figure 28 Task & Route for *solution.bestAssignment*