Multi-objective optimization and obstacle avoidance strategies for remotely operated vehicles path planning

Kalajžić, Ivana

FACULTY OF SCIENCE, UNIVERSITY OF SPLIT

IVANA KALAJŽIĆ

MULTI-OBJECTIVE OPTIMIZATION AND OBSTACLE AVOIDANCE STRATEGIES FOR REMOTELY OPERATED VEHICLES PATH PLANNING

MASTER'S THESIS

Split, rujan 2024.

FACULTY OF SCIENCE, UNIVERSITY OF SPLIT

DEPARTMENT OF MATHEMATICS

MULTI-OBJECTIVE OPTIMIZATION AND OBSTACLE AVOIDANCE STRATEGIES FOR REMOTELY OPERATED VEHICLES PATH PLANNING

MASTER'S THESIS

Student: Ivana Kalajžić

Mentor: doc. dr. sc. Aljoša Šubašić

Split, September 2024

BASIC DOCUMENTATION CARD

FACULTY OF SCIENCE, UNIVERSITY OF SPLIT

DEPARTMENT OF MATHEMATICS

MASTER'S THESIS

MULTI-OBJECTIVE OPTIMIZATION AND OBSTACLE AVOIDANCE STRATEGIES FOR REMOTELY OPERATED VEHICLES PATH PLANNING

Ivana Kalajžić

Abstract:

This paper explores the optimization of path planning for Remotely Operated Vehicles through the development of a multi-objective optimization model and the implementation of obstacle avoidance strategies.

Key words:

Route Optimization, Weighted Sum Method, Dijkstra Algorithm, Dynamic Trajectory Adjustment.

Specifications:

54 pages, 9 pictures and tables, 3 references, english language

Mentor: *Assistant Professor Aljoša Šubašić*

Čommittee:

Assistant Professor Tanja Vojković

Teaching / research assistant Pavao Radić

Povjerenstvo za diplomski rad je prihvatilo ovaj rad *27.09.2024.*

TEMELJNA DOKUMENTACIJSKA KARTICA

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

SVEUČILIŠTA U SPLITU

ODJEL ZA MATEMATIKU

DIPLOMSKI RAD

VIŠEOBJEKTNA OPTIMIZACIJA I STRATEGIJE ZA IZBJEGAVANJE PREPREKA PRI PLANIRANJU STAZA ZA VOZILA NA DALJINSKO UPRAVLJANJE

Ivana Kalajžić

Sažetak:

U ovom radu bavimo se optimizacijom planiranja staza za vozila na daljinsko upravljanje kroz izgradnju višeobjektnog optimizacijskog modela i kroz implementaciju strategija za izbjegavanje zapreka.

Ključne riječi:

Optimizacija rute, metoda težinske sume, Dijkstrov algoritam, dinamička promjena rute.

Podatci o radu:

broj stranica: 54, broj slika i tablica: 9, broj literaturnih navoda: 3, jezik izvornika: engleski

Mentor: *doc. dr. sc. Aljoša Šubašić*

Članovi povjerenstva:

doc. dr. sc. Tanja Vojković

Asistent Pavao Radić

TEMELJNA DOKUMENTACIJSKA KARTICA

This thesis was approved by a Thesis commettee on *27.09.2024.*

Contents

Contents **vi**

1 Introduction **1**

2 Multi-objective Optimization **4**

2.1 Introduction to Optimization 4

2.2 Single-objective optimization 6

2.2.1 Linear Programming 6

2.3 Multi-objective Optimization 10

2.3.1 Multi-objective Optimization Problem 10

2.3.2 Multi-objective Optimization Solutions 13

2.3.3 Weighted Sum Method 16

2.4 Mathematical Model 18

2.4.1 Data 19

2.4.2 Decision Variables 21

2.4.3 Objective Function 21

2.4.4 Constraints 23

3 Obstacle Avoidance Using Dijkstra Algorithm **25**

3.1 Problem Definition 26

3.2 Dijkstra's Algorithm 31

3.3 Implementation of Dijkstra Algorithm to Obstacle Avoidance Problem 37

3.3.1 Experiment 44

4 Conclusion **47**

Bibliography **49**

Chapter 1

Introduction

Remotely Operated Vehicles (ROVs) have revolutionized underwater exploration and operations, playing a major role in various industries, including marine research, exploration, and intervention. These machines are equipped with advanced sensors and technologies, enabling them to navigate complex underwater environments and carry out tasks that would be hazardous or impossible for human divers. The impact of ROVs extends beyond mere environmental monitoring; they contribute significantly to resource management, and the advancement of underwater industrial constructions.

As ROVs are deployed in immensely challenging environments, the need for effective navigation and obstacle avoidance mechanisms becomes of prime importance. These vehicles often operate in dynamic settings where unexpected obstacles, such as marine life, submarines or underwater waste, can pose significant risks to their mission objectives. Therefore, developing robust algorithms and optimization models that allow ROVs to adapt their trajectories in real-time is necessary for ensuring safe and efficient operations. The integration of obstacle avoidance strategies not only enhances the operational competence of ROVs but also minimizes the risk of damage to both the vehicle and the surrounding environment.

As expected, optimization of path planning plays a crucial role in enhancing the performance of ROVs. In essence, optimization involves finding the best solution to a problem within a defined set of constraints and objectives. In the case of ROVs, this means determining the most efficient path that the vehicle can take while full-filling operational constraints such as energy consumption and mission timelines. The optimization process can be compound, as it must account for multiple objectives, including minimizing travel distance while prioritizing completion of different tasks and minimizing risk caused by environmental changes.

Multi-objective optimization (MOO) techniques are particularly relevant in this context, as they allow for the simultaneous consideration of antagonistic objectives. For instance, an ROV may need to balance the trade-off between minimizing travel time and maximizing the quantity of data collected during its mission or minimizing travel distance while maximizing number of tasks that need to be achieved. The weighted sum method is one approach that can be used to seize such multi-objective problems. By assigning weights to each objective, decision makers can prioritize their goals and explore various trade-offs, ultimately leading to more informed, significant and effective operational strategies.

In the process of finding path for avoiding the obstacle, the application of optimization techniques becomes even more demanding. ROVs must be equipped with software that enable them to detect obstacles in real-time and adjust their paths correspondingly. This requires a dynamic algorithms that can process incoming data from sensors and imaging systems, allowing the ROV to make quick decisions about its trajectory. The Dijkstra Algorithm is one of the widely known shortest path-finding algorithm that can be adapted for use in ROVs to identify the shortest and safest route while avoiding detected obstacles.

Chapter 2

Multi-objective Optimization

2.1 Introduction to Optimization

Optimization is a fundamental concept that is spread through various fields, from engineering and economics to logistics and machine learning. The core of optimization is to find the best possible solution to a problem given a set of constraints and objectives. Formally, an optimization problem involves establishing the optimal value of a function within a given domain. This section provides a general overview of optimization, and single-objective optimization, followed by a more detailed discussion of linear programming, a specific and widely used optimization technique.

Optimization problems can be commonly categorized into several types based on the nature of the objective function, the constraints, and the decision variables. The considerable categories include linear, nonlinear, integer, combinatorial, and dynamic optimization.

Let's begin by defining the most general form of the optimization problem.

Definition 2.1.1. An optimization problem can be expressed as:

$$\min_{x \in X} f(x)$$

where $f(x)$ is the objective function to be minimized, and X is the feasible region defined by a set of constraints.
The feasible region X is the set of all x that satisfy the constraints of the problem.

An optimization problem does not only handle minimization of an objective function, but can also be used to maximize an objective function. Therefore, sometimes our goal is to find not just the lowest value but also the highest value of the objective function.

To approach both types of problems in a consistent way, we can use a simple mathematical equivalence relation:

$$\max f(x) \Leftrightarrow \min(-f(x))$$

This principle shows that any problem where we need to maximize a function can be turned into a minimization problem by taking the negative of the function, and the other way round. Therefore, most optimization problems are commonly formulated as minimization problems for simpleness and convenience.

2.2 Single-objective optimization

Single-objective optimization (SOO) can be described as a type of optimization problem where a single objective function is used. Meaning, the focus is on optimizing only one of the criteria whilst following the given constraints. Let's take a look at a special type of single-objective optimization in which the objective function and constraints are linear.

2.2.1 Linear Programming

Linear programming (LP) is a powerful mathematical method for determining the best end result in a mathematical model whose requirements are represented by linear relationships. It has numerous applications in various industries, including manufacturing, transportation, finance, and military planning.

Definition 2.2.1. A linear programming problem is formulated as:

$$\min_{x \in \mathbb{R}^n} \mathbf{c}^T \mathbf{x}$$

subject to:

$$\mathbf{A}\mathbf{x} \leq \mathbf{b}$$

$$\mathbf{x} \geq 0$$

where $\mathbf{c}$ is an n-dimensional vector of coefficients for the objective function, $\mathbf{A}$ is an $m \times n$ matrix of coefficients for the constraints, $\mathbf{b}$ is an m-dimensional vector of constraint bounds, and $\mathbf{x}$ is the vector of decision variables.

As we can see from definition above, in linear programming,the objective function and the constraints are linear. This means that the objective function is a linear combination of decision variables. Analogously, the constraints are linear equations or inequalities involving the decision variables. The feasible region, defined by the intersection of linear constraints, forms a convex polytop. This convexity suggestes that any local optimum is also a global optimum, which is a significant advantage in finding the best solution efficiently. Linear programming benefits from a well known theoretical premise and efficient algorithms, such as the Simplex method and Interior-point methods, which make solving these problems relatively straightforward [1].

Example 2.2.1 (Production Optimization)**.** In a factory, two different products X_1 and X_2 are being manufactured by three machines M_1, M_2, and M_3. Each machine can be used for a limited amount of time. Machine M_1 has a maximum working time of 10 hours, machine M_2 is restricted to 20 hours, and machine M_3 is limited to 12 hours to avoid excessive wear. The production times of each product on each machine range: product X_1 requires 1 hour on machine M_1, 1 hour on machine M_2, and 3 hours on machine M_3; product X_2 requires 1 hour on machine M_1, 4 hours on machine M_2, and 1 hour on machine M_3. The objective is to maximize the combined time of usage of all three machines.

Every production decision must follow the constraints on the available time. In particular, we have:

$$x_1 + x_2 \leq 10$$

$$x_1 + 4x_2 \leq 20$$

$$3x_1 + x_2 \leq 12$$

where x_1 and x_2 denote the production levels. The combined production time of all three machines is:

$$f(x_1, x_2) = 5x_1 + 6x_2.$$

Thus, the problem in compact notation has the form:

$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0, \end{aligned}$$

where

$$\mathbf{c}^T = [5, 6],$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 4 \\ 3 & 1 \end{bmatrix},$$

$$\mathbf{b} = \begin{bmatrix} 10 \\ 20 \\ 12 \end{bmatrix}.$$

Definition 2.2.2. Any vector $\mathbf{x}$ that yields the minimum value of the objective function $\mathbf{c}^T\mathbf{x}$ over the set of vectors satisfying the constraints $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq 0$, is said to be an **optimal feasible solution**.

Definition 2.2.3. An m-element subset B of $\{1, \ldots, n\}$ is said to be a **basis** (with respect to matrix A) if the columns of A indexed by the elements in B are linearly independent.

We say that $\mathbf{x}^* \in \mathbb{R}^n$ is a **basic solution** to the system $A\mathbf{x} = \mathbf{b}$ if there exists a basis B such that

(i) $A\mathbf{x}^* = \mathbf{b}$;

(ii) $x_j^* = 0$ for all $j \notin B$.

Definition 2.2.4. An optimal feasible solution that is basic is said to be an **optimal basic feasible solution**.

Theorem 2.2.1 (Fundamental Theorem of LP)**.** Consider a linear program in standard form.

(i) If there exists a feasible solution, then there exists a basic feasible solution;

(ii) If there exists an optimal feasible solution, then there exists an optimal basic feasible solution.

2.3 Multi-objective Optimization

In the last few years, the field of multi-objective optimization has gained significant attention across various domains, including engineering, finance, and environmental management. Multi-objective optimization involves optimization of two or more objectives simultaneously, which is a common scenario in real-life applications.

2.3.1 Multi-objective Optimization Problem

Similarly to a single-objective optimization problem, the multi-objective optimization problem may contain a number of constraints which any feasible solution (including all optimal solutions) must satisfy. Therefor, any multi-objective optimization problem can be represented by the following general mathematical model:

$$
\begin{aligned}
\min \qquad & \mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x})]^T \\
\text{subject to} \qquad & g_i(\mathbf{x}) \geq 0, \quad i = 1, 2, \ldots, p \\
& h_j(\mathbf{x}) = 0, \quad j = 1, 2, \ldots, q \\
& x_i^{(\min)} \leq x_i \leq x_i^{(\max)}, \quad i = 1, 2, \ldots, n \\
& \mathbf{x} = [x_1, x_2, \ldots, x_n]^T \in \mathcal{Q}
\end{aligned}
$$

where m is the number of objective functions, $\mathcal{Q}$ is the n-dimensional search space defined by the lower bounds $\mathbf{x}^{(\min)} = [x_1^{(\min)}, x_2^{(\min)}, \ldots, x_n^{(\min)}]^T$ and upper bounds $\mathbf{x}^{(\max)} = [x_1^{(\max)}, x_2^{(\max)}, \ldots, x_n^{(\max)}]^T$ of decision variables $\mathbf{x}$. The constraints $g_i(\mathbf{x}) \geq 0$ and $h_j(\mathbf{x}) = 0$ represent p inequality constraints and q equality constraints, respectively. If $p = q = 0$, the problem simplifies to an unconstrained multi-objective optimization problem.

Example 2.3.1 (Multi-Objective Production Optimization)**.** In a factory, two different products X_1 and X_2 are being manufactured by three machines M_1, M_2, and M_3. Each machine can be used for a limited amount of time. Machine M_1 has a maximum working time of 15 hours, machine M_2 is restricted to 10 hours, and machine M_3 is limited to 12 hours. Production times of each product on each machine are given in Table 1. The objective is to maximize the combined production time of utilization of all three machines as well as to maximize the profit generated by the production.

Machine	X_1 (hours)	X_2 (hours)
M_1	2	3
M_2	4	1
M_3	3	2

Table 2.1: Production times for X_1 and X_2 on machines M_1, M_2, and M_3

The time constraints for each machine are:

$$\begin{aligned} 2x_1 + 3x_2 &\leq 15 \qquad \text{(Machine } M_1 \text{ time constraint)} \\ 4x_1 + x_2 &\leq 10 \qquad \text{(Machine } M_2 \text{ time constraint)} \\ 3x_1 + 2x_2 &\leq 12 \qquad \text{(Machine } M_3 \text{ time constraint)} \end{aligned}$$

The combined production time of all three machines is given by the function:

$$f_1(x_1, x_2) = 2x_1 + 3x_2 + 4x_1 + x_2 + 3x_1 + 2x_2 = 9x_1 + 6x_2$$

Production of product X_1 bring profit of 5 and production of product X_2 bring profit of 7. Hence, the profit generated by producing these products is:

$$f_2(x_1, x_2) = 5x_1 + 7x_2$$

Thus, the problem in compact notation has the form:

$$\begin{aligned} \text{maximize} \qquad & [f_1(\mathbf{x}), f_2(\mathbf{x})]^T \\ \text{subject to} \qquad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & x_1, x_2 \geq 0, \end{aligned}$$

where:

$$f_1(x_1, x_2) = 9x_1 + 6x_2,$$

$$f_2(x_1, x_2) = 5x_1 + 7x_2$$

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 4 & 1 \\ 3 & 2 \end{bmatrix},$$

$$\mathbf{b} = \begin{bmatrix} 15 \\ 10 \\ 12 \end{bmatrix},$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

2.3.2 Multi-objective Optimization Solutions

In multi-objective optimization problems, the challenge lies in defining the solutions. From a mathematical standpoint, there is not a single solution of multi-objective problem but rather a set of solutions. In 1951, Koopmans introduced the concept of Pareto efficiency, which describes the solution set under partial order rather than total order. A solution is considered Pareto optimal if no other solution can improve one objective without degrading another. This concept is crucial in decision-making processes where multiple criteria must be considered.

Definition 2.3.1 (Feasible Solution)**.** A solution vector $\mathbf{x} \in \mathcal{Q}$ is defined as a feasible solution if it satisfies all the inequality and equality constraints for $i = 1, 2, \ldots, p$ and $j = 1, 2, \ldots, q$. Otherwise, it is an infeasible solution.

All feasible solutions constitute the feasible domain $\mathcal{U}$, and all infeasible solutions constitute the infeasible domain $\mathcal{U}'$. Clearly, $\mathcal{U} \cup \mathcal{U}' = \mathcal{Q}$, where $\mathcal{U} \subseteq \mathcal{Q}$ and $\mathcal{U}' \subseteq \mathcal{Q}$.

In other words, feasible solution is one that meets all the constraints imposed by the problem. The feasible domain is the set of all such solutions, while the infeasible domain is the set of solutions that do not meet the constraints.

In the decision variable space (space of all possible values of decision variables), a solution $\mathbf{a}$ is said to dominate another solution $\mathbf{b}$ if $\mathbf{a}$ is no worse than $\mathbf{b}$ in all objectives and strictly better in at least one objective.

Definition 2.3.2 (Decision Variable Domination)**.** For two vectors $\mathbf{a} = [a_1, a_2, \ldots, a_n]^T$ and $\mathbf{b} = [b_1, b_2, \ldots, b_n]^T$ in the decision variable space, $\mathbf{a}$ is said to dominate $\mathbf{b}$ (denoted as $\mathbf{a} \prec \mathbf{b}$) if:

(i) $\forall i \in \{1, 2, \ldots, m\}\ f_i(\mathbf{a}) \leq f_i(\mathbf{b})$

(ii) $\exists j \in \{1, 2, \ldots, m\}\ f_j(\mathbf{a}) < f_j(\mathbf{b})$

In the objective function space, a point $\mathbf{g}$ dominates another point $\mathbf{h}$ if $\mathbf{g}$ is no worse than $\mathbf{h}$ in all objectives and strictly better in at least one objective.

Definition 2.3.3 (Objective Function Domination)**.** For two vectors $\mathbf{g} = [g_1, g_2, \ldots, g_m]^T$ and $\mathbf{h} = [h_1, h_2, \ldots, h_m]^T$ in the objective function space, $\mathbf{g}$ is said to dominate $\mathbf{h}$ (denoted as $\mathbf{g} \prec \mathbf{h}$) if:

(i) $\forall i \in \{1, 2, \ldots, m\}$, $g_i \leq h_i$

(ii) $\exists j \in \{1, 2, \ldots, m\}$, $g_j < h_j$

Definition 2.3.4 (Pareto Optimal Solution)**.** A vector $\mathbf{x}^* = [x_1^*, x_2^*, \ldots, x_n^*]^T \in \mathcal{Q}$ is a Pareto optimal solution if:

$$\forall \mathbf{x} \in \mathcal{Q},\, \mathbf{x} \neq \mathbf{x}^* \Rightarrow \mathbf{f}(\mathbf{x}) \not\prec \mathbf{f}(\mathbf{x}^*)$$

The set of all Pareto optimal solutions is called the Pareto optimal set, denoted as $\mathcal{PS}^*$.

In other words, a Pareto optimal solution is one where no other solution in the feasible domain can improve any objective without causing a degradation in at least one other objective.

Definition 2.3.5 (Pareto Optimal Front)**.** The Pareto optimal set represented in the objective function space is called the Pareto optimal front, denoted as:

$$\mathcal{PF}^* = \{\mathbf{f}(\mathbf{x}) \,|\, \mathbf{x} \in \mathcal{PS}^*\}$$

The Pareto optimal front is the set of objective vectors corresponding to the Pareto optimal solutions. It displays the trade-offs between different objectives in the objective function space.

Multi-objective optimization methods focuses on finding solutions that are as close as possible to the Pareto optimal front and are uniformly distributed to ensure equity. Such methods should show good convergence and diversity. Additionally, the solutions should be numerous to ensure a wide range of options for decision-makers. Once the Pareto optimal set is found, decision-makers can select the concluding solution based on specific optimization problems or personal preferences. A diverse and sizeable set of solutions allows for better comparison and selection according to various criteria and fondness.

2.3.3 Weighted Sum Method

One of the most simple and commonly used approaches to tackle multi-objective optimization problems is the weighted sum method.

This method transforms the multi-objective problem into a single-objective problem by allocating weights to each objective function, considering their relative importance. The weighted sum of the objectives is then optimized, allowing for the observation of different trade-offs by varying the weights. This approach is particularly appealing due to its simplicity and ease of implementation.

However, it has its limitations, such as the potential to overlook non-convex regions of the Pareto front and the challenge of selecting appropriate weights that accurately represent the decision-maker's preferences.

Mathematically, general formulation of the weighted sum method for a multi-objective optimization problem can be expressed as follows:

$$\begin{aligned}
\min \quad & f(\mathbf{x}) = \sum_{m=1}^{M} w_m f_m(\mathbf{x}) \\
\text{subject to} \quad & g_j(\mathbf{x}) \geq 0, \quad j = 1, 2, \ldots, J \\
& h_k(\mathbf{x}) = 0, \quad k = 1, 2, \ldots, K \\
& x_i^{(L)} \leq x_i \leq x_i^{(U)}, \quad i = 1, 2, \ldots, n
\end{aligned}$$

where:

(i) $\mathbf{x}$ is the vector of decision variables.

(ii) $f_m(\mathbf{x})$ is the m-th objective function.

(iii) w_m is the weight assigned to the m-th objective function, with $w_m \geq 0$ and $\sum_{m=1}^{M} w_m = 1$.

(iv) $g_j(\mathbf{x})$ are the inequality constraints.

(v) $h_k(\mathbf{x})$ are the equality constraints.

(vi) $x_i^{(L)}$ and $x_i^{(U)}$ are the lower and upper bounds on the decision variables.

The weights w_m are user-supplied and represent the priority or importance of each objective function. This allows for a customized approach depending on the importance of each objective and exploring different trade-offs between the objectives.

One of the key strengths of this method lies in its simplicity and flexibility. The weighted sum method is apparent and easy to implement. It converts a multi-objective problem into a single-objective problem, which can be solved using standard optimization techniques.

On the other hand, choosing appropriate weights to obtain a desired Pareto-optimal solution can be challenging. The solution is sensitive to the choice of weights, and inappropriate weights may lead to sub-optimal solutions. Additionally, in cases where the objective space is non-convex, the weighted sum method may fail to find certain Pareto-optimal solutions. This is because the method relies on linear combinations of the objectives, which may not capture the true trade-offs in a non-convex space [2].

2.4 Mathematical Model

In this section we are constructing a model to determine the optimal route for a Remotely Operated Vehicle (ROV) in an offline setting, considering various factors that impact the route choice.

The ROV's mission demand visiting a series of stations, each with a specific priority for visitation, while taking into account the distance between stations and the corresponding environmental risks. The objective is to minimize the total distance traveled, prioritize stations based on their urgency (with 1 being the highest priority and 5 the lowest) and precedence (some stations need to be visited before others, because of the nature of the tasks executed in stations), and manage risks related to certain stations.

This must be achieved within a set of constraints to ensure an efficient and feasible route.

By utilizing the before-mentioned weighted sum method, we combine these objectives into a single, flexible objective function. The constants α, β, and γ play a influential role in balancing the different objectives, allowing for a modifying approach depending on the specific mission requirements.

The model's design is not only practical but also highly customized, with the ability to adjust the weights to explore different trade-offs and optimize the ROV's route according to the decision-maker's preferences. This foundation prepare the way for further purification and application in real-world scenarios, where the balance between these competing objectives is critical to the success of ROV missions.

By minimizing the total distance, prioritizing stations with higher urgency, while also taking into account the risks associated with environmental conditions, this model provides a comprehensive and flexible tool for offline ROV route optimization.

2.4.1 Data

To build our model, we first need to lay out and formalize the data that will be used. Let us assume we have gathered the necessary information and now introduce the notation that will represent the data:

(i) B: set of all stations.

(ii) $S = |B| < \infty$: total number of stations.

(iii) N: minimum number of stations required to visit (0.5 S).

(iv) d_{ij}: distance between station i and station j.

(v) p_i: priority of station i, lower values indicate higher priority.

(vi) P: set of precedence ordered pairs (i, j) such that beacon i must be visited before beacon j

(vii) r_i: the risk of extreme weather conditions at station i

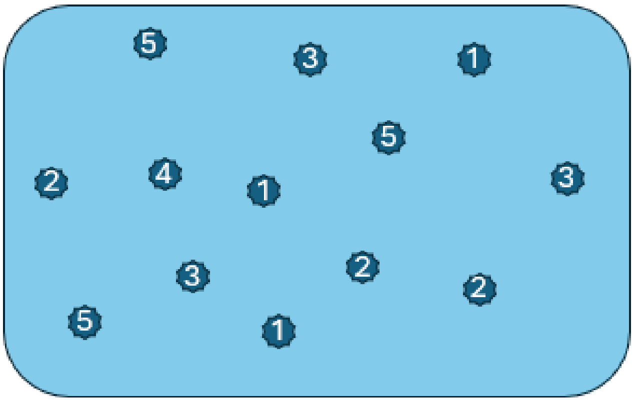


Figure 2.1: Example of data: Search area with station ranked by visitation priority

2.4.2 Decision Variables

The decision variables x_{ij} and u_i are essential components of the model, with each serving a well defined purpose in the optimization process:

(i) x_{ij}: binary variable indicating if the route goes from station i to station j.

(ii) u_i: auxiliary variables for sub-tour elimination.

2.4.3 Objective Function

Applying the weighted sum method, the objective function in the model is expressed as:

$$\alpha \sum_{i \in B} \sum_{j \in B} d_{ij} \cdot x_{ij} + \beta \sum_{i \in B} \sum_{j \in B} p_i \cdot x_{ij} + \gamma \sum_{i \in B} \sum_{j \in B} r_i \cdot x_{ij}$$

Here is an explanation of each part of the objective function:

Term

$$\sum_{i \in B} \sum_{j \in B} d_{ij} \cdot x_{ij}$$

focuses on minimizing the total distance in the route. Here, d_{ij} represents the distance between station i and station j, and x_{ij} is a binary variable indicating whether the route from i to j is taken (1 if taken, 0 otherwise).

Term

$$\sum_{i \in B} \sum_{j \in B} p_i \cdot x_{ij}$$

incorporates the priority or urgency of each station into the optimization. Here, p_i represents the priority of station i, with lower values indicating higher priority.

Finally, term

$$\sum_{i \in B} \sum_{j \in B} r_i \cdot x_{ij}$$

accounts for additional factors represented by r_i, which include various environmental risks associated with visiting station i. The product $r_i \cdot x_{ij}$ incorporates these risks into the route planning, ensuring that the model considers potential challenges or disadvantages when determining the optimal path. Stations situated in environmentally sensitive or hazardous areas might have higher r_i values to account for potential environmental impacts.

The formulation of objective function effectively combines the three objectives—minimizing distance, prioritizing important stations, and minimizing risk—into a single objective. The constants α, β, and γ can be adjusted to achieve the preferred balance between these competing objectives, with the constraint that their sum must equal one. This adjustment allows for an analysis of how their values impact the overall objective function value.

2.4.4 Constraints

Constraints are given by:

$$\sum_{i\in B}\sum_{j\in B} x_{ij} \geq N \tag{2.1}$$

$$\sum_{j\in B} x_{ij} \leq 1 \quad \forall i \in B \tag{2.2}$$

$$\sum_{i\in B} x_{ij} \leq 1 \quad \forall j \in B \tag{2.3}$$

$$u_i - u_j + N \cdot x_{ij} \leq N - 1 \quad \forall i, j \in B,\ i \neq j \tag{2.4}$$

$$u_i \geq 0 \quad \forall i \in B \tag{2.5}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in B \tag{2.6}$$

$$p_i \in \{1, 2, 3, 4, 5\} \quad \forall i \in B \tag{2.7}$$

$$u_i \leq u_j \quad \forall (i, j) \in P \tag{2.8}$$

$$\alpha + \beta + \gamma = 1 \tag{2.9}$$

- (2.1): Visit at least 50% of the stations
- (2.2): Each visited station must be exited exactly once
- (2.3): Each visited station must be entered exactly once
- (2.4): Miller-Tucker-Zemlin sub-tour elimination constraint
- (2.5): This constraint ensures that the values of are appropriate for eliminating sub-tours
- (2.6): constraint on the variable's values
- (2.7): constraint on the variable's values

- (2.8): ensures that the auxiliary variable for station i is less than or equal to that of station j, effectively enforcing the visit order
- (2.9): constraint on weights of Weighted sum method

Chapter 3

Obstacle Avoidance Using Dijkstra Algorithm

Now, we are addressing the problem of obstacle avoidance for a remotely operated vehicle in a 3D underwater environment. In the context of ROVs performing underwater navigation, an effective and fast online obstacle avoidance mechanism is crucial for maintaining safe and logical travel. This process is particularly significant when the ROV, which follows a pre-planned offline path, encounters unexpected obstacles in its environment. The main objective of this problem is to dynamically adjust the ROV's trajectory to navigate around such obstacles while ensuring that the detour is minimal and reasonable, and that ROV returns to the pre-planned path seamlessly. This mechanism becomes active when the ROV detects an obstacle, which is identified through image processing techniques.

3.1 Problem Definition

The ROV is tasked with following a predetermined path that optimally visits several stations or waypoints. However, the path planning conducted offline does not account for unforeseen obstacles that might suddenly appear in the ROV's path during its operation. To address this, the ROV must employ an obstacle avoidance mechanism that can adapt to real-time changes in the environment.

The ROV is equipped with image-capturing technology that provides continuous visual data of its surroundings. Let's suppose that at discrete intervals, denoted as Δt, the ROV updates its memory with new image data. The image processing system analyzes these images to detect the presence of obstacles in environment. If an obstacle is detected, it is, generally, represented as a polygon with n sides, which approximates the shape and broadness of the obstacle. This polygonal representation allows for a precise definition of the obstacle's location and dimensions.

The operational procedure of obstacle avoidance mechanism involves the following steps:

1. The ROV relies exclusively on images captured from its onboard sensors to perceive its surroundings. At discrete time intervals, denoted as Δt, the ROV updates its environmental memory with new data gathered through these images.

2. When an image processing indicates the presence of an obstacle, the ROV initiates an obstacle avoidance protocol. This involves analyz-

Figure 3.1: Data is gathered from image processing and used for obstacle avoidance strategies

ing the captured image to identify and locate the obstacle. When an captured image undergoes a detection algorithm of a standard training model for detection tasks, a n-sided polygon is constructed on the resultant image indicating presence of an obstacle. Therefore, the result of this image processing procedure is a polygon with n sides that approximates the obstacle's shape and position in the environment. Vertices of the n - sided polygon are 3-dimensional points that will be an input data of graph search algorithm for obstacle avoidance. Meaning, at each time step Δt, a graph $G(V, E)$ is updated where:

(i) V represents the set of vertices (3D coordinates) of the polygons.

(ii) E represents the edges, which are the Euclidean distances between these vertices.

Since the positions of the vertices are updated at each time step, continuously, the graph is dynamic and needs to be reconstructed continually.

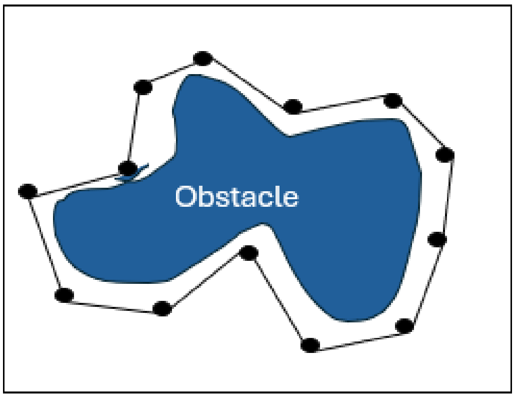


Figure 3.2: Polygon with sides that approximates the obstacle's shape

To ensure accurate obstacle avoidance, we need to address the fact that not every vertex of the polygon will be used as a node in the graph due to limitations in the captured image.Specifically, the image may not encompass the entirety of the obstacle, leading to incomplete data. Vertices of the polygon located along the periphery of the captured image are particularly problematic. These vertices suggest that the captured image does not cover the entire extent of the obstacle, indicating that the obstacle likely extends beyond the edges of the captured picture.

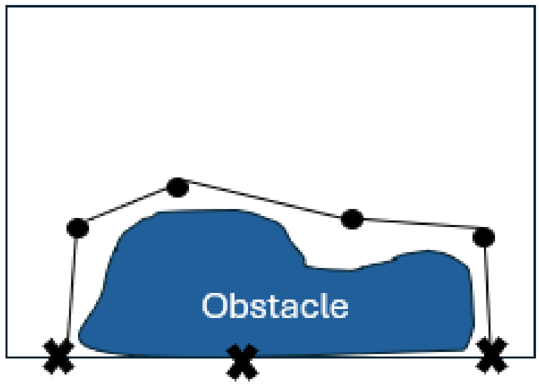


Figure 3.3: Vertices of the polygon located close to the periphery of the captured image are disregarded

Therefore, these vertices that are on or closed to the periphery of the captured image are disregarded in the graph construction process. This is because including them could lead to an inaccurate representation of the obstacle's shape and position. When vertices on the edge of the image are included, there is a risk that the polygon's boundaries are not correctly aligned with the actual obstacle. Since these edge vertices are likely to be outside the true obstacle's boundary, their inclusion could cause the graph to misrepresent the obstacle's location and shape.

The primary concern with incorporating such edge vertices is that it may lead to a situation where the ROV's path planning algorithm does not accurately avoid the obstacle. Specifically, if the edges associated with these vertices extend beyond the actual edge of the image, the ROV might be directed towards areas where the obstacle is actually present but not captured. Consequently, this could result in potential collisions between the ROV and the obstacle, undermining the effectiveness of the obstacle avoidance strategy. Therefore, to ensure a safe and accurate navigation, these peripheral vertices are excluded from the graph to prevent any misleading conclusions about the obstacle's extent and to maintain a reliable path planning system.

Furthermore, not every new node in an updated graph will be connected to all the previous nodes. For every new update, nodes that are vertices of the same polygon will be connected in a way that every node is connected to its neighbors nodes. Also, all nodes from the new update will be connected to the all the nodes in only the forerunner update. Reason for this is to ensure logical form of the graph and so the result of search algorithm is valid and sensible.

3. Upon detecting an obstacle, the ROV's path is no longer aligned with the offline pre-planned route. Consequently, a graph search algorithm is employed to determine a new path around the obstacle. The search begins at the point where the ROV strays from the pre-planned trajectory, establishing this point as the start node of the graph for shortest path finding algorithm.

4. Once the ROV has successfully navigated around the obstacle and no further obstacles are detected, it resumes its journey towards the subsequent station or waypoint as defined in the original pre-planned path.

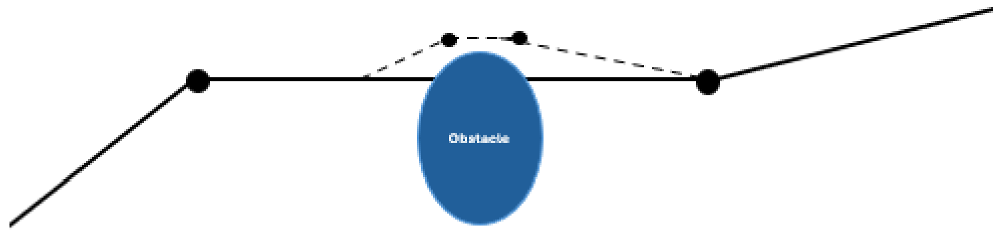


Figure 3.4: Obstacle avoidance process

3.2 Dijkstra's Algorithm

Dijkstra's algorithm is a fundamental and well known algorithm in computer science, named after Dutch computer scientist Edsger W. Dijkstra, who first published it in 1959. It is used to find the shortest path from a starting node (often referred to as the "source" node) to all other nodes in a graph with weighted edges.

Dijkstra's algorithm has several significant characteristics that make it particularly effective in solving specific types of problems. The algorithm implements a greedy approach, meaning it makes the optimal choice at each step with the goal of finding the global optimum. This characteristic ensures that once the shortest path to a node is pined down, it remains unchanged, contributing to the algorithm's overall efficiency and speed.

Another critical aspect of Dijkstra's algorithm is its demand for non-negative edge weights. The algorithm assumes that once a path to a node has been established with a certain cost, no shorter path will be discovered later. Negative weights would compromise this assumption, potentially leading to incorrect results and making the algorithm unsuitable for graphs with such weights.

The time complexity of Dijkstra's algorithm varies depending on the data structures used. In its simplest form, when implemented with arrays, the algorithm has a time complexity of $O(V^2)$, where V represents the number of nodes. Nevertheless, by utilizing more advanced data structures like Fibonacci heaps, the time complexity can be reduced to $O(V \log V + E)$, where E denotes the number of edges. This improvement makes the algorithm more suitable for greater graphs.

Dijkstra's algorithm has a wide range of implementations across various fields. In computer networks, it plays a crucial role in network routing protocols, determining the shortest path for data to travel across routers. In Geographic Information Systems (GIS), the algorithm is integral to mapping software, helping to find the shortest path between locations, such as in GPS navigation systems. The algorithm is also widely used in robotics and artificial intelligence for path-finding, enabling robots to navigate through environments efficiently. Furthermore, in telecommunications, Dijkstra's algorithm is employed to optimize the routing of signals across complex networks, ensuring efficient communication pathways [3].

Dijkstra's Algorithm steps:

1. Set the distances of all nodes as ∞ except for the source node, which is set to 0.
2. Mark all nodes as non-visited, including the source node.
3. While there are non-visited nodes:
   (a) Set the non-visited node with the smallest current distance as the current node C.

(b) For each neighbor N of the current node C:

i. Calculate the potential new distance through C as:

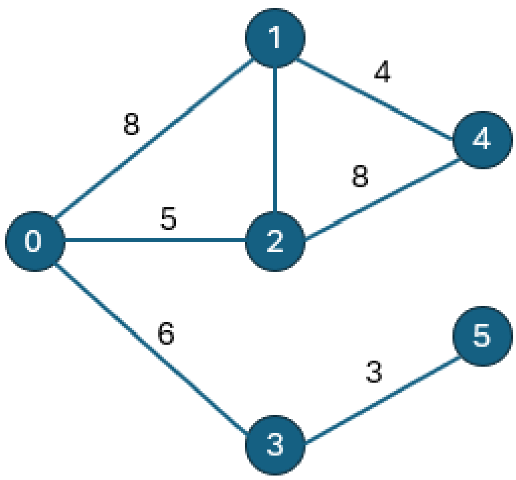
$$\text{new distance} = \text{current distance of } C + \text{weight of edge } CN$$

ii. If this new distance is smaller than the current distance of N, update the distance of N.

(c) Mark the current node C as visited.

4. Repeat the process in step 3. until all nodes are marked as visited.

Example 3.2.1 (Finding the Shortest Path using Dijkstra's Algorithm)**.** To illustrate how Dijkstra's Algorithm works, let's consider the following example of the graph whose nodes are denoted as numbers and weights written above the edges:



To find the shortest path from the source node (0) to all other nodes in the weighted graph using Dijkstra's Algorithm, we perform the following steps.

First, initialize the distance table by setting the distance of all nodes as ∞, except for the source node, which is set to 0. This gives the initial distances as follows:

Distance to 0: 0,

Distance to 1: ∞,

Distance to 2: ∞,

Distance to 3: ∞,

Distance to 4: ∞,

Distance to 5: ∞.

All nodes are initially marked as non-visited.

We then proceed by iterating through the non-visited nodes, selecting the node with the smallest current distance as the current node and updating the distances of its neighbors. The process continues until all nodes are visited.

In the first iteration, the current node is source node 0 because it has the smallest distance (0). We update the distances of its neighbors (nodes 1, 2, and 3). The new distance to node 1 is calculated as

$$0 + 8 = 8,$$

to node 2 as

$$0 + 5 = 5,$$

and to node 3 as

$$0 + 6 = 6.$$

The updated distances after this iteration are:

Distance to 1: 8,

Distance to 2: 5,

Distance to 3: 6.

Node 0 is then marked as visited.

In the second iteration, node 2 is set as the current node as it has the smallest distance among non-visited nodes. We then update the distances of its neighbors (nodes 0, 1, 3, and 4). The new distance to node 4 is calculated as

$$5 + 8 = 13.$$

Since the current distances to vertices 1 and 3 are smaller than their potential new distances, they are not updated. The updated distance to node 4 becomes 13:

Distance to 4: 13.

Node2 is marked as visited.

In the third iteration, node 3 is set as the current node, having the smallest distance among non-visited nodes. The distance to its neighbor, node 5, is updated to

$$6 + 3 = 9.$$

The updated distances are:

Distance to 5: 9. Node 3 is marked as visited.

Next, node 1 is selected as the current node. The distance to its neighbor, node 4, is updated to $8 + 4 = 12$, which is smaller than the current

distance of 13. Thus, the distance to vertex 4 is updated:
Distance to 4: 12.

Node 1 is marked as visited.

In the fifth iteration, node 5 is the current node with a distance of 9. However, all its neighbors are already visited, so no further updates are necessary. Node 5 is marked as visited.

Finally, in the sixth iteration, node 4 is selected as the current node with a distance of 12. Again, all its neighbors are already visited, so no further updates are needed and node 4 is marked as visited.

At this point, all nodes have been visited, and the shortest distances from vertex 0 to all other vertices have been determined.

The final shortest distances from vertex 0 to all other vertices are:
Distance to 0: 0,
Distance to 1: 8,
Distance to 2: 5,
Distance to 3: 6,
Distance to 4: 12,
Distance to 5: 9.

3.3 Implementation of Dijkstra Algorithm to Obstacle Avoidance Problem

In the real-time underwater navigation, efficiently avoiding obstacles is crucial for the safe operation of ROVs. The dynamic nature of the underwater environment demands a vigorous algorithm capable of calculating and recalculating paths as new obstacles are detected. Dijkstra's algorithm, famous for its effectiveness in finding the shortest paths in a graph, is particularly well-suited for this task. By integrating Dijkstra's algorithm with the ROV's obstacle detection system, we can ensure that the vehicle can dynamically adjust its path, avoiding obstacles while minimizing detours. The following section details the implementation of Dijkstra's algorithm specifically tailored to address the obstacle avoidance problem in a 3D underwater environment.

The first step in the obstacle avoidance process involves updating the graph that represents the ROV's environment. As described in problem definition, this graph consists of nodes corresponding to points in 3D space and edges representing the Euclidean distances between these points. The vertices are updated dynamically based on the data provided by image processing, which are designed to detect and approximate obstacles. When a new set of points is identified, representing the vertices of an obstacle, the graph needs to be updated to include these points.

The `updateGraph` algorithm begins by connecting all the vertices of the newly detected obstacle to its neighbors. This is done by iterating through the list of new vertices and calculating the Euclidean distances between vertex with previous and following vertex, thereby adding the corresponding edges to the graph. In other words, the current update are connected in a way that every vertex, except for the last one, is connected to the subsequent one and every vertex, except the first one, is connected to the previous one. Finally, first and the last vertex are connected. After connecting the vertices of the current update, the algorithm proceeds to connect these new vertices with the vertices of the previous update. This ensures that the graph remains sensibly connected, accounting for all obstacles that the ROV has encountered so far. By continually updating the graph in this manner, the algorithm maintains an accurate and up-to-date representation of the ROV's environment, which is crucial for the path finding process. The graph is undirected, meaning that the distance from point u to point v is the same as the distance from v to u, which is why the distance is stored symmetrically in the graph matrix.

Algorithm 1 Update Graph

```
Require: graph matrix of size MAX_POINTS × MAX_POINTS
Require: points array of all points
Require: setSize array of input sizes
Require: inputNumber number of updates
Require: totalPoints size of array points
for u ← totalPoints - setSize[inputNumber] +1 to totalPoints do
    if u > then totalPoints - setSize[inputNumber] +1    ▷ Connect to the previous node (if not the first node)
        graph[u][u-1] ← euclideanDistance(points[u], points[u-1])
        graph[u-1][u] ← graph[u][u-1]    ▷ Graph is undirected
    end if
    if u < totalPoints then    ▷ Connect to the following node (if not the last node)

        graph[u][u+1] ← euclideanDistance(points[u], points[u+1])
        graph[u+1][u] ← graph[u][u+1]    ▷ Graph is undirected
    end if
end for
Connect first and last node
for u ← totalPoints - setSize[inputNumber] - setSize[inputNumber - 1] to totalPoints - setSize[inputNumber] do
    for v ← totalPoints - setSize[inputNumber] +1 to totalPoints do
        graph[u][v] ← euclideanDistance(points[u], points[v])    ▷ Connect all nodes in the current update with previous update
        graph[v][u] ← graph[u][v]    ▷ Graph is undirected
    end for
end for
```

Once the graph is updated, the next step is to find the shortest path from the ROV's current location to one of the nodes in the latest update. This is accomplished using Dijkstra's algorithm, which is well-suited for finding the shortest path in a weighted graph.

The algorithm starts by initializing the distance to all nodes in the graph as infinite, except for the starting node, which is set to zero. This initialization reflects the fact that initially, the shortest path to any node is unknown, except for the starting node itself. A set called `sptSet` (Shortest Path Tree Set) is used to keep track of the nodes that have been visited, ensuring that each node is only visited once.

The core of Dijkstra's algorithm is its greedy approach: at each step, the algorithm selects the non-visited node with the smallest known distance from the starting node and explores its neighbors. For each neighbor, the algorithm calculates the potential new distance by adding the distance from the current node to the weight of the edge connecting the current node to the neighbor. If this new distance is smaller than the currently known distance to the neighbor, the algorithm updates the neighbor's distance and records the current node as its predecessor.

The process repeats until all distances of the goal nodes(nodes from the latest update) are calculated. The result is a list of distances from the starting node to the goal nodes in the graph, with the shortest path to each node being determined by following the recorded predecessors back from the goal node to the starting node.

Algorithm 2 Dijkstra Algorithm

```
Require: graph matrix of size MAX_POINTS × MAX_POINTS
Require: totalPoints number of nodes in the graph
Require: currentLocation source node
Require: inputSet array of goal nodes
Initialize distances as array of size totalPoints with all elements set to ∞
distances[currentLocation] ← 0    ▷ Distance to the source is zero
Initialize sptSet as array of size totalPoints with all elements set to false
Initialize previous as array of size totalPoints with all elements set to null
for count ← 0 to totalPoints - 1 do
    u ← vertex with minimum dist not in sptSet
    sptSet[u] ← true    ▷ Mark vertex u as processed
    if u is in inputSet then
        break    ▷ Stop if we reach a goal node
    end if
    for v ← 0 to totalPoints - 1 do
        if not sptSet[v] and graph[u][v] ≠ 0 and distances[u] ≠ ∞ and distances[u] + graph[u][v] < distances[v] then
            distances[v] ← distances[u] + graph[u][v]
            previous[v] ← u    ▷ Track the predecessor of v
        end if
    end for
end for
Initialize goalNode as node in inputSet with the minimum distances
Initialize path as an empty list
currentNode ← goalNode    ▷ Start with the goal node found
while currentNode is not null do
    insert(currentNode, path)    ▷ Insert currentNode at the beginning of the path
    currentNode ← previous[currentNode]
end while
return path ▷ Return the list of nodes forming the shortest path to a goal node
```

In a dynamic environment, where the ROV continually encounters new obstacles, the graph and shortest path need to be updated regularly. The `graph-update` algorithm combines the graph updating and shortest path calculation into a single process, allowing the ROV to adjust its path in real-time as it navigates through the underwater environment.

The algorithm begins by initializing the graph with the ROV's current location and then enters a loop where it continuously reads input from the ROV's sensors. Each time a new obstacle is detected, the graph is updated with the new points, and Dijkstra's algorithm is invoked to recalculate the shortest path from the ROV's current location to the next target. The newly calculated path is then used to guide the ROV around the obstacle.

If no new obstacles are detected, the algorithm re-initializes its data structures, preparing for the next obstacle avoidance process. This re-initialization is crucial because it ensures that the algorithm does not retain outdated information from previous avoidance, which could lead to incorrect path calculations and overloaded memory.

By dynamically updating the graph and recalculating the shortest path, the algorithm ensures that the ROV can adapt to changes in its environment and navigate around obstacles in an efficient way.

Algorithm 3 Dynamic Graph Update and Shortest Path Calculation

```
Require: Initial 3D point currentLocation    ▷ Point where ROV deviates from pre-planned trajectory and a source node of the graph
Require: Array of 3D points inputSet
  Initialize points as a null array of size MAX_POINTS    ▷ Array to hold the points from the inputs
  Initialize graph as a null matrix of size MAX_POINTS × MAX_POINTS
  Initialize setSize as a null array of size MAX_POINTS    ▷ Array to hold the sizes of each input set
  Initialize inputNumber as integer    ▷ Counter for the number of non-empty inputs
  Initialize totalPoints as integer
  totalPoints ← 1
  setSize[0] ← 1
  inputNumber ← 0
  while True do
      Read inputSet
      Remove noncompliant points from inputSet
      Read currentLocation
      points[0] ← currentLocation
      if inputSet is not empty then
          inputNumber ← inputNumber + 1
          setSize[inputNumber] ← number of points in inputSet
          Update points with inputSet
          totalPoints ← totalPoints + setSize[inputNumber]
          Call updateGraph(graph, points, setSizes, inputNumber, totalPoints)
          Call dijkstra(graph, totalPoints, currentLocation)
          newPath[MAX_POINTS] ← dijkstra(graph, totalPoints, currentLocation)
          Print newPath
      end if
      Reinitialize all data structures    ▷ No obstacle or avoidance process is done
      points[0] ← currentLocation
      totalPoints ← 1
      setSize[0] ← 1
      inputNumber ← 0
      sleep(Δ t)
  end while
```

3.3.1 Experiment

The algorithm from the section before is now tested on synthetic data, which consists of 3D point that represent current location of the ROV and multiple sets of points that mimic the vertices of the polygon that approximates the obstacle in multiple captured images, as explained before.

Input:

{0.0, 0.0, 0.0}

{ ∅,

{ {1, 3, 0}, {1, -3, 0}, {1.5, 1, 3}, {1.5, -2, 2.5} },

{ {7, 6, 3.5}, {7.5, -2.7, 5.5}, {7.5, 2, 5}, {7, -6.7, -3.1} },

{ {12.1, 2, 7.3}, {12.4, 3.6, 5.5}, {13, -4.6, 5.4} },

∅ }

Here we label input points and visualize the data and graph.

O=(0.0,0.0,0.0),

A=(1,3,0),

B=(1,-3,0),

C=(1.5,1,3),

D=(1.5,-2,2.5),

F=(7,6,3.5),

G=(7.5,-2.7,5.5),

H=(7.5,2,5),

J=(7,-6.7,-3.1),

K=(12.1,2,7.3),

L=(13, -4.6, 5.4),

I=(12.4, 3.6, 5.5).

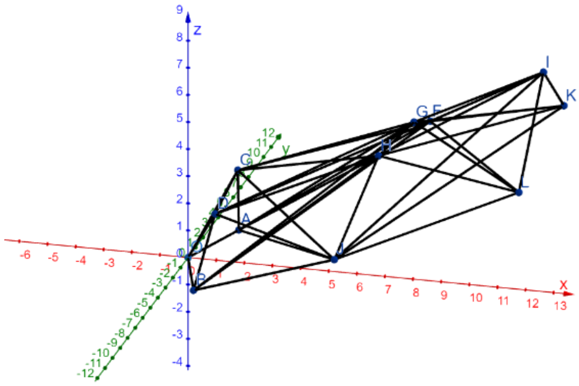


Figure 3.5: Data represented in 3D space

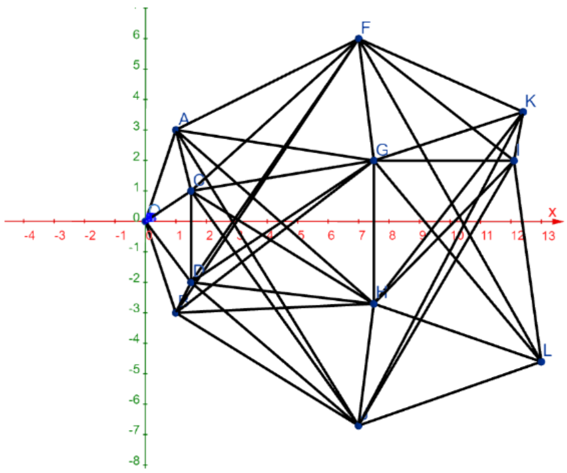


Figure 3.6: Floor plan of data

Output:

(0.00, 0.00, 0.00)

(1.00, 3.00, 0.00)

(7.00, 6.00, 3.50)

(12.10, 2.00, 7.30)

Time taken: 0.000058 seconds

Chapter 4

Conclusion

In summary, the integration of optimization techniques into the operational framework of Remotely Operated Vehicles (ROVs) is essential for enhancing their effectiveness and safety in complex underwater environments. As ROVs continue to play a essential role in various industries, the development of advanced optimization models that address multiple objectives will be crucial for their future success. By employing methods such as the weighted sum method, decision-makers can tailor ROV operations to meet specific mission goals, ensuring that these vehicles can adapt to unforeseen challenges while maximizing their operational efficiency.

Moreover, the implementation of advanced obstacle avoidance algorithms, such as those based on Dijkstra's Algorithm, highlights the importance of real-time data processing. As technology continues to advance, the potential for ROVs to operate autonomously and efficiently in challenging underwater scenarios will only increase, paving the way for new discoveries and innovations.

Ultimately, the ongoing research and development in optimization techniques for ROVs not only enhance their operational capabilities but also contribute to the broader understanding of marine environments. By ensuring that ROVs can navigate safely and effectively, we can unlock new opportunities for exploration, conservation, and data collection, thereby making a lasting impact on our understanding of the underwater world.

Bibliography

[1] Edwin K.Chong, Stanidlaw H. Zak, *An Introduction to Optimization*, John Wiley & Sons, Inc., 2001, pages 255-271

[2] Sebastien Verel, *An Introduction to Multiobjective Optimization*, 2020

[3] *What is Dijkstra's Algorithm? Introduction to Dijkstra's Shortest Path Algorithm*, last updated: 09 May, 2024, URL: https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/, last checked: 2024-08-21