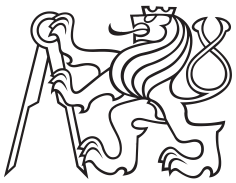
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

A metaheuristics for the Graph Search Problem with order-dependent weights

Bc. Vadym Ostapovych

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Field of study: Open Informatics
Subfield: Software Engineering
May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Ostapovych Vadym** Personal ID number: **483578**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software Engineering**

II. Master’s thesis details

Master’s thesis title in English:

A metaheuristics for the Graph Search Problem with order-dependent weights

Master’s thesis title in Czech:

Metaheuristika pro problém hledání v grafu s váhami závislými na pořadí

Guidelines:

The thesis is motivated by the problem of finding randomly placed target in a given polygonal environment. The current approaches to this problem lead to the discrete Graph Search Problem (GSP), where weights of nodes in the underlying graph are static and correspond to the area seen from the associated location. Nevertheless, this is an approximation of the area NEWLY seen from the location. Ideally, the weights should change based on already visited locations. The student will thus address the GSP with dynamic weights in the following steps:
1) Familiarize yourself with the GSP and its solution methods.
2) Formulate the search task in the polygonal domain as the GSP where the weight of a node depends on already visited nodes.
3) Design and implement a metaheuristics for the formulated task.
4) Experimentally verify the properties of the implemented metaheuristics and compare it for a search problem in a polygonal domain with a method that considers fixed weights.

Bibliography / sources:

[1] Mikula, J., and Kulich, M. (2022). Solving the traveling delivery person problem with limited computational time. Central European Journal of Operations Research, 1–31.
[2] Kulich, M., and Přeučil, L. (2022). Multi-robot search for a stationary object placed in a known environment with a combination of GRASP and VND. International Transactions in Operational Research, 29(2), pp. 805-836.
[3] Kulich, M., Miranda Bront, J. J., and Přeučil, L. (2017). A meta-heuristic based goal-selection strategy for mobile robot search in an unknown environment. Computers & Operations Research, 84.
[4] Mladenovic, N., Urosevic, D., Hanafi, S. (2012). Variable neighborhood search for the travelling deliveryman problem. 4OR pp. 1–17.

Name and workplace of master's thesis supervisor:

RNDr. Miroslav Kulich, Ph.D. Intelligent and Mobile Robotics CIIRC

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **05.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

RNDr. Miroslav Kulich, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I want to express my deepest gratitude to my supervisor, RNDr. Miroslav Kulich, Ph.D., for his exceptional guidance and patience. His support has been essential to completing this work.

I am also profoundly thankful to Ing. Jan Mikula for providing and explaining the code and sharing his expertise in the implementation aspects of the thesis.

Additionally, I sincerely thank the Intelligent and Mobile Robotics team, CIIRC, for granting access to the necessary hardware and for their advice on the presentation aspects of this work.

Finally, I am very grateful to my mom and CTU schoolmates for their unwavering support during challenging times.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within, in accordance with the *Methodical instructions for observing ethical principles in the preparation of university theses*

Prague, May 24, 2024

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s *Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.*

V Praze, 24. května 2024

Abstract

The Graph Search Problem (GSP) and the Traveling Deliveryman Problem (TDP) are variants of the NP-hard Traveling Salesman Problem (TSP). GSP finds application in Mobile Robot Search (MRS), which involves locating a randomly placed target object within a priory known polygonal domain. The solution quality of MRS depends on the weights corresponding to the newly visible area from a given location. The current state-of-the-art method based on GSP treats these weights as static, resulting in approximations. In this thesis, we propose a novel formulation of MRS using GSP with order-dependent weights. Using the combination of various constructive and efficient improving heuristics and the perturbation, we construct Sequential Variable Neighborhood Search (SVND) and General Variable Neighborhood Descent (GVNS) metaheuristics. Based on the experimental evaluation, we select the optimal parameters and components of SVND and GVNS. The results of the experiments show that the proposed metaheuristics outperform the reference method significantly.

Keywords: Mobile Robot Search, Travelling Deliveryman Problem, Graph Search Problem, Metaheuristics, Variable Neighbourhood Search, Local Search

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Intelligent and Mobile Robotics, CIIRC

Abstrakt

Problém hledání v grafu (GSP) a problém doručovatele (TDP) jsou varianty NP-těžkého problému obchodního cestujícího (TSP). GSP nachází uplatnění v problému vyhledávání mobilním robotem (MRS), které zahrnuje lokalizaci náhodně umístěného cílového objektu v předem známém polygonálním prostředí. Kvalita řešení MRS závisí na váhách odpovídajících oblasti nově viditelné z dané pozice. Současná metoda založená na GSP tyto váhy chápe jako statické, což vede k aproximacím. V této práci navrhujeme novou formulaci MRS pomocí GSP s váhami závislými na pořadí. S využitím kombinace různých konstruktivních a efektivních zlepšujících heuristik a perturbace konstruujeme metaheuristiky Sestup ve Variabilní Sekvenci Sousedství (SVND) a Obecné Vyhledávání ve Variabilních Sousedstvích (GVNS). Na základě experimentálního vyhodnocení jsou vybrány optimální parametry a komponenty SVND a GVNS. Výsledky experimentů ukazují, že navržené metaheuristiky poskytují lepší řešení než referenční metoda.

Klíčová slova: Hledání mobilním robotem, Problém cestujícího doručovatele, Problém hledání v grafu, Metaheuristika, Vyhledávání ve variabilních sousedstvích, Lokální vyhledávání

Contents

Figures

Tables

Chapter 1

Introduction

The Traveling Salesman Problem (TSP) is a famous optimization problem that aims to find the shortest possible route to visit each city in a given set exactly once and return to the starting city. Due to its fundamental role in optimizing routes and paths, the TSP and its variations are used in many fields, including astronomy, robotics, biology, and tourism [1]. It has many variations, such as the Traveling Deliveryman Problem (TDP) and the Graph Search Problem (GSP). These problems are considered NP-hard, meaning they are difficult to solve optimally within a reasonable computational time. This puts the main focus of the research community on developing efficient algorithms to quickly find near-optimal solutions without checking every possible solution.
One of the GSP's applications is Mobile Robot Search (MRS), which aims to locate a randomly placed target object within a polygonal domain. MRS is a complex problem; modern approaches thus discretize the environment by representing it as a graph, with the nodes' weights defining the new visible area from that location. The current state-of-the-art method [2] for solving this problem considers the weights static, leading to approximating the newly visible area.
This thesis proposes a new formulation of the GSP with order-dependent, dynamic weights that change based on the areas already explored. We aim to develop metaheuristics for this problem, evaluate their properties experimentally, and compare them with the current state-of-the-art method. While we do not require the proposed approaches to operate in strict real-time, we aim to progress towards that capability. Our approach could serve as a solid foundation for a new formulation of Mobile Robot Search.

1.1 State-of-the-art

The GSP we are studying is the **combinatorial optimization** (CO) problem. According to Papadimitriou and Steiglitz [3], a deterministic CO problem involves finding the best object from a finite or possibly countable infinite set. This object, known as a solution, is typically an integer, a subset, a permutation, or a graph structure. The real-valued cost function of the optimization problem defines the quality of the solution. All possible solutions

to the problem are considered to be in solution space. As summarized by Hansen and Mladenovic [4], the goal of combinatorial optimization is to find a solution whose quality is higher than all possible solutions to the problem or show the absence of a feasible solution within a finite and reasonable time. The **Traveling Deliveryman Problem** (TDP) is a well-known CO problem in the operational research community and has several mathematical formulations [5], [6]. While the Travelling Salesman Problem (TSP) aims to minimize the total travel time from the start to the final node in the graph, TDP shifts to minimizing the total arrival time to each node. Using the postman analogy, TSP focuses on saving time for the postman, while TDP aims to minimize the average waiting time for each customer. According to Blum [7], on a metric space, the TDP is NP-hard like the TSP, but surprisingly, the TDP is harder to solve or approximate. In the literature, it is also known as the Traveling Repairman Problem [8] or Minimum Latency Problem (MLP) [9]. The interesting practical application of TDP is the home delivery of pizzas [6] or disk-head scheduling for optimizing the access and retrieval of data from a disk [7].

Graph Search Problem is considered a weighted TDP, where every customer has an assigned weight, and the postman wants to visit the customers with the highest priority first, minimizing the weighted average waiting time for each customer. Koutsoupias first formulated GSP [10] for applying search in a large network of hypertext documents. The overview of TSP, TDP, and GSP with some approximation schemes for designing routing strategies within a web network can also be found in the work of Ausielo [11]. No further developments of GSP were presented in the related literature except the last works of Kulich [12], [13].

The operational research community follows two main trends in solving TDP: exact and approximation algorithms. The **exact algorithms** aim to find the optimal solution to the problem and are represented by Integer Linear Programming (ILP) and Mixed Linear Programming (MILP) methods. They are usually formulated for The Time-Dependent Traveling Salesman Problem (TDTSP), a generalization of TSP and TDP. They can solve TDTSP using the Branch-Cut-and-Price algorithm to optimality up to 50-60 vertices within several hours [14], [15]. One of the best algorithms for TDTSP proposed by Abeledo [16] solves 107 vertices within 48 hours. The **approximation algorithms** are mostly represented by **metaheuristics**, the general frameworks designed to solve complex optimization problems. Many researchers propose their definitions. We will provide several ones. Osman [17] defines metaheuristics as "An iterative process that directs a subordinate heuristic by cleverly combining various concepts to explore and exploit the search space. Learning strategies are employed to organize information effectively to discover near-optimal solutions efficiently". Another possible definition is taken from Blum [18] "Metaheuristics are high-level strategies for exploring search spaces by using different methods." The overview of different metaheuristics can be found in *Handbook of Metaheuristics* [19] or in [20] describing the working principles and inspiring concepts for stochastic optimization problems.

The introduction for beginners on how to build the Variable Neighborhood Descent (VND) and Variable Neighborhood Search (VNS) metaheuristics with respective definitions can be found in the tutorial of Mladenovic [21]. Salehipour [22] proposed a GRASP metaheuristic for the Traveling Deliveryman Problem (TDP), which employs VND or VNS. Based on this approach, an effective metaheuristic called GILS-RVND was developed by Silva [23]. This method combines GRASP with Iterated Local Search (ILS). The current **state-of-the-art method** is Multi-start-General VNS, the stochastic metaheuristic that uses multiple stopping criteria and iteration limits. It was first presented by Mikula [24] and compared against the existing GILS-RVND in a series of experiments and proved more efficient [2]. The GRASP scheme presented by Kulich [12] mostly represents the GSP metaheuristics.
The applications of GSP for **mobile robot search**, where weights represent newly covered areas in a polygonal environment, come from the work of Kulich and Preucil [25]. Additionally, Sarmiento [26] addresses the problem of mobile robot search in a polygonal domain by modeling the time required to find an object as a random variable influenced by the chosen search path and assuming a uniform probability density function for the object's location. The latest trend in mobile robot exploration is the Watchman Route Problem (WRP) application, where the newly covered areas are not considered, and the goal is to find the shortest route visiting every part of a polygonal environment[27].

1.2 Contributions & Structure

The main contributions of this thesis are as follows:

- We formulated the Graph Search Problem with order-dependent weights to model the Mobile Robot Search Problem in the polygonal domain. The following formulation better describes the discrete version of MRS and allows the development of new methods that lead to more promising solutions.
- We developed a solution approach to the formulated problem utilizing two main metaheuristics: Sequential Variable Neighbourhood Descent (SVND) and General Variable Neighbourhood Search (GVNS). This includes the effective implementation of local search strategies and constructive heuristics for generating initial solutions to the problem.
- We conducted experiments to identify the most effective local search operators, exploration strategies, and metaheuristic parameters and components. Then, we compared our proposed metaheuristics with the current state-of-the-art method in the two experiments with a priority on the solution quality.
- Our experiments showed that the proposed formulation improves Mobile Robot Search performance on various maps by approximately 6-40 %,

with the maximal visibility regions overlapping up to 30-50%. Additionally, we sought to define overlapping regions that reveal the limitations of the reference method, but the current definition only allows for finding the real visibility limit on the maps.

The thesis is organized into five chapters, each explaining the content step-by-step. The Chapter 1 briefly introduces the problem, a literature review, and contributions. The Chapter 2 details the formulation of the Graph Search Problem with order-dependent weights based on Mobile Robot Search in a priory known environment. The Chapter 3 describes the methods used to solve the formulated problem. The Chapter 4 presents the experimental evaluation and analyzes the selected methods and results. The Chapter 5 summarizes the thesis and the potential improvements for the selected approaches.

Chapter 2

Graph Search Problem

This chapter addresses the Graph Search Problem with order-dependent weights, focusing on its practical application in Mobile Robot Search. The first section describes Mobile Robot Search in a priory known environment from a continuous perspective and defines key visibility concepts. The second section formulates the problem from the practical side by discretizing trajectories and environments. The third section merges all the ideas from the previous ones and formulates the robotic problem as the optimization problem.

2.1 Mobile robot search in priory known environment

Assume a scenario when the mobile robot navigates in a priory known environment and searches for the stationary object of interest that is placed randomly. To explain the environmental terms, we refer to [28], [29]. The robot operates within the world $\mathcal{W} \subset R^2$, representing a non-empty connected closed bounded subset of two-dimensional Euclidean space. The region $\mathcal{W}$ denotes the obstacle-free space. The compliment $R^2 \setminus \mathcal{W}$ represents the inaccessible area to the robot with a single outer boundary and one or multiple inner boundaries denoted as holes. The boundary is a simple polygon defined as the closed connected series of pairwise non-intersecting line segments. There isn't an explicit consideration of obstacles that might limit visibility. Instead, the entirety of the set $\mathcal{W}$ represents the area that can be seen through, while $R^2 \setminus \mathcal{W}$ denotes the space that obstructs vision. The environment remains static, and the robot faces two primary constraints

- *Motion.* The robot is confined to moving within the environment $\mathcal{W}$. It cannot cross these limits or traverse over $R^2 \setminus \mathcal{W}$.
- *Visibility.* The robot's visibility is restricted to the interior of the environment $\mathcal{W}$. It cannot see beyond or through the boundaries $R^2 \setminus \mathcal{W}$.

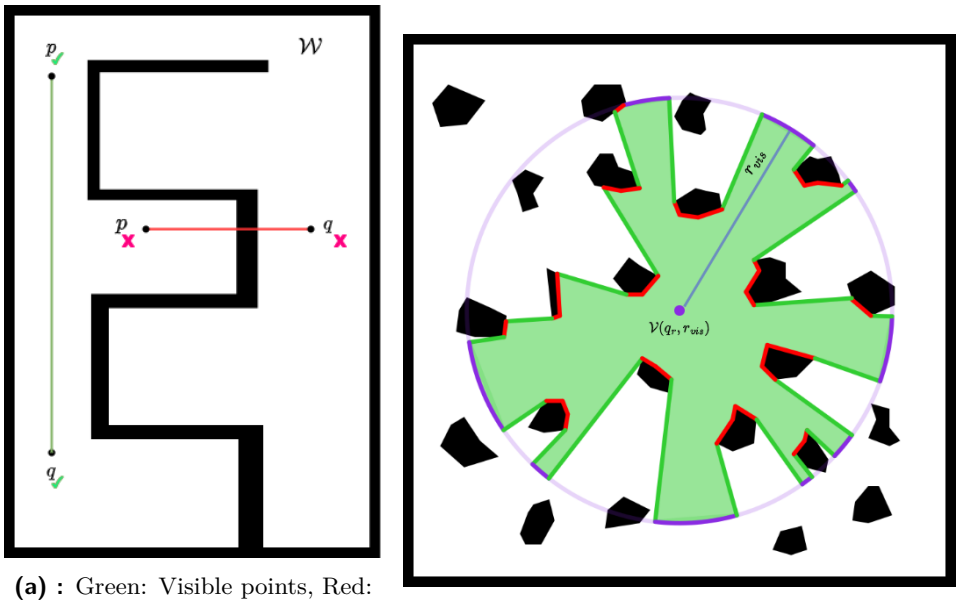
The key visibility definitions of the environment are

- *Points visibility.* Two points $p, q \in \mathcal{W}$ are visible to each other only if their line segment $\overline{pq}$ lies entirely within $\mathcal{W}$, $\overline{pq} \in W$, see Figure 2.1a.

- *Visibility Region.* The set of all points visible from a point p denoted as $V(p) = \{q \in \mathcal{W} \,|\, \overline{pq} \subset \mathcal{W}\}$.
- *Visibility Graph.* A data structure representing visibility relationships within a finite set of points P is defined as the undirected graph $G_P = (V_P, E_P)$, Here, $V_P = P$ denotes the vertex set, and $E_P = \{\{p, q\} \,|\, p, q \in P \wedge p \neq q \wedge \overline{pq} \subset \mathcal{W}\}$ represents the edge set, pairs of the different visible points within the environment $\mathcal{W}$.

From the physical point of view, the robot is the material point. Its configuration includes its 2D coordinates, signifying the center of its footprint within the environment, denoted as $q_r = (x, y)$, $q_r \in \mathcal{W}$. The robot is equipped with a 360-degree sensor, placed in its center q_r with a limited visibility range $r_{vis} \in R^+$, including $r_{vis} = \infty$. The set of points visible by the robot within the distance r_{vis} is defined as the sensing limited visibility region

$$\mathcal{V}(q_r, r_{vis}) = \{p \in \mathcal{W} \,|\, \overline{q_r p} \subset \mathcal{W} \wedge \|\overline{q_r p}\| \leq r_{vis}\} \tag{2.1}$$



(a) : Green: Visible points, Red: Invisible points

(b) : Sensing visibility region limited by r_{vis}

Figure 2.1: Visibility concepts for the environment $\mathcal{W}$

Assume that at time $t = 0$, the robot is positioned at the starting point. It begins to navigate through the environment $\mathcal{W}$ using the obstacle-free continuous trajectory $\tau : \left[0, t^{end}\right] \to Q_\tau \subset \mathcal{W}$. Then, the searched object is considered to be found by the robot during the execution of a trajectory τ when the robot's sensors detect the object $\mathcal{O}$, if there exists $t \in [0, t^{end}]$ such that $\mathcal{O} \in \mathcal{V}(\tau(t), r_{vis})$.
The objective of the Mobile Robot Search is to minimize the expected time of locating the object. To achieve this, we analyze a random variable T

denoting the time taken to detect the target object, with a density probability function $f_{T|\tau}(t)$. This function represents the probability of finding the object at time t, assuming the robot follows a collision-free trajectory $\tau \in [0, t^{end}]$. Minimizing the average time taken across all potential object locations in the environment is more natural, considering the object's location can be anywhere in the environment

$$\mathbb{E}(T \mid \tau) = \int_0^{t^{end}} t^{'} f_{T|\tau}(t^{'})\, dt^{'} \tag{2.2}$$

Then, the goal of Mobile Robot Search is to find a continuous trajectory that minimizes the expected time required to detect the target object

$$\tau^* = \underset{\tau}{\operatorname{argmin}}\, \mathbb{E}(T \mid \tau) \tag{2.3}$$

2.2 Discrete formulation

The continuous formulation of MRS is deemed complex due to its infinite potential object locations and unrestricted robot configurations. This leads to boundless potential exploration trajectories. Hence, simplifying the problem by adopting discrete terms becomes practical.

2.2.1 Trajectory quantization

Let's consider a scenario where a robot follows a trajectory $\tau = [0, t^{end}]$ and senses the environment at discrete times $\mathbf{t} = (t_0 = 0, t_1, ..., t^{end})$, $\forall i \in \{0, 1, .., n-1\}$, $t_i < t_{i+1}$. The locations where the robot performs measurements are sensing locations s_i. Each t_i represents the duration for the robot to traverse to the sensing location s_i after exploring locations s_0 through s_i. We assume no sensing occurs as the robot transitions from s_{i-1} to s_i. After completing the measurement at sensing location s_{i-1}, the entire environment is divided into two parts: the already explored or "covered" denoted as $\mathcal{W}_{i-1}^{cov}$, and the areas yet to be explored or "uncovered", denoted as $\mathcal{W}_{i-1}^{unc}$, see Figure 2.2a. Additionally, we presume that the duration of sensing the environment is zero and that the sensing region within the visibility radius has already been explored at the starting location.

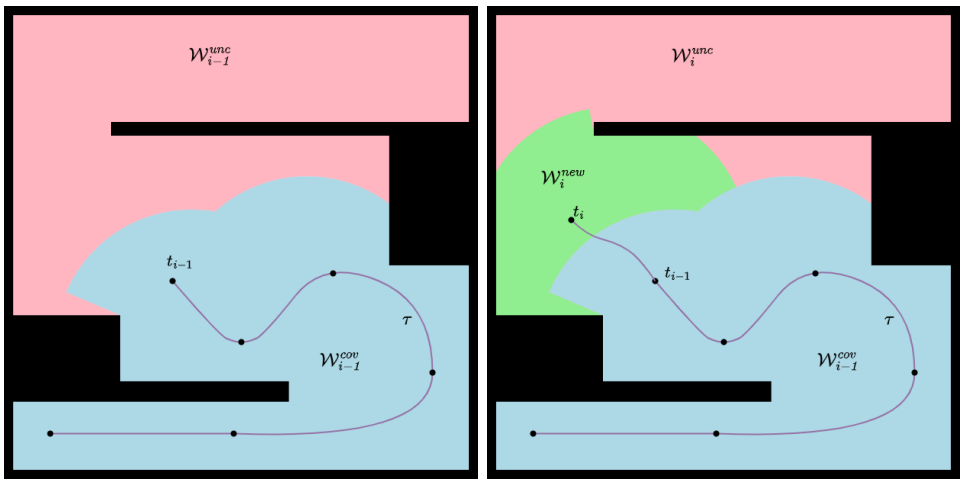
$$\mathcal{W}_0^{unc} = \mathcal{W} \setminus \mathcal{V}(\tau(0), r_{vis}),\ \mathcal{W}_0^{cov} = \mathcal{V}(\tau(0), r_{vis}) \tag{2.4}$$

The robot progresses along its trajectory; in the next step i shown in Figure 2.2b, the robot is at the location s_i and explores the new area of environment that was not explored $\mathcal{W}_i^{new}$.

$$\mathcal{W}_i^{new} = \mathcal{W}_{i-1}^{unc} \setminus \mathcal{V}(\tau(t_i), r_{vis}) \tag{2.5}$$

The update of the covered area in the step i will be

$$\mathcal{W}_i^{cov} = \mathcal{W}_{i-1}^{cov} \cup \mathcal{V}(\tau(t_i), r_{vis}) = \mathcal{W}_{i-1}^{cov} \cup \mathcal{W}_i^{new} \tag{2.6}$$



(a) : Covered (blue) and uncovered (red) area after step $i-1$

(b) : New explored area (green) in the step i

Figure 2.2: Environment split over discrete trajectory

Let's additionally assume that $a(S) = \iint_S, dS$ represents the scalar value associated with the area S. We define the discrete cumulative probability function that represents the likelihood of detecting the object by time t_i as the ratio of the total area covered up to time t_i to the area of the entire environment

$$F_{T|\tau}(t_i) = P(T \leq t_i|\tau) = \frac{a(\mathcal{W}_i^{cov})}{a(\mathcal{W})} \tag{2.7}$$

Alternatively, the discrete probability of finding the object at time t_i can be defined as the relative area of view of the newly explored area at time t_i to the area of the whole environment

$$p(T = t_i|\tau) = \frac{a(\mathcal{W}_i^{new})}{a(\mathcal{W})} = \frac{a\left(\mathcal{W}_{i-1}^{unc} \setminus \mathcal{V}(\tau(t_i), r_{vis})\right)}{a(\mathcal{W})} \tag{2.8}$$

Subsequently, the expected time over quantized trajectory can be computed as the sum of times weighted by the probability of finding the object in each sensing location

$$\mathbb{E}(T\ |\tau) = \sum_{i=1}^{n} t_i\, p(T = t_i|\tau) = \frac{1}{a(\mathcal{W})} \sum_{i=1}^{n} t_i\, a(\mathcal{W}_i^{new}) \tag{2.9}$$

2.2.2 Environment discretization

The quantization of trajectory leads to the implicit discretization of the environment $\mathcal{W}$, which is partitioned into a finite set of possibly intersecting sensing regions limited by a visibility radius r_{vis}. These regions are denoted as $\mathbf{R} = (\mathcal{R}_0, \mathcal{R}_1, ..., \mathcal{R}_n)$, where each region $\mathcal{R}_i$ is centered at a point representing a sensing location $s_i \in \mathcal{W}$. and define the set of points that are visible for the robot when it's located at position s_i, $R_i = \mathcal{V}(s_i, r_{vis})$.

On the other hand, the environment can be discretized explicitly and independently on a particular trajectory. The idea is to find the smallest set of feasible sensing locations such that their limited visibility regions will cover the entire environment.

$$\mathcal{W} = \mathcal{R}_0 \cup \mathcal{R}_1 ... \cup \mathcal{R}_n = \bigcup_{i=0}^{n} \mathcal{R}_i \tag{2.10}$$

This discretization allows us to represent the environment as a graph, where the vertices are the sensing locations where the robot uses its sensor to explore the environment, and the edges represent the travel time between these sensing locations, see Figure 2.3. The robot will cover the entire environment if it visits all the sensing locations. Consequently, we can formulate the MRS as the Graph Search Problem to find the path in the graph that will minimize the expected time to visit all the locations, the vertex weights indicate the relative area of view the robot covers at each sensing location, and the edges represent the travel time between locations.

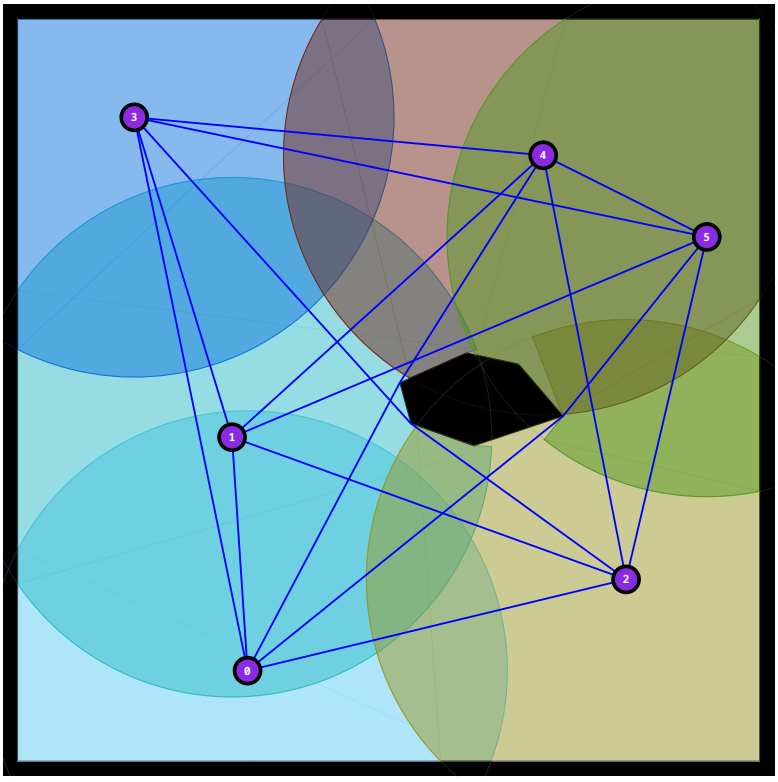
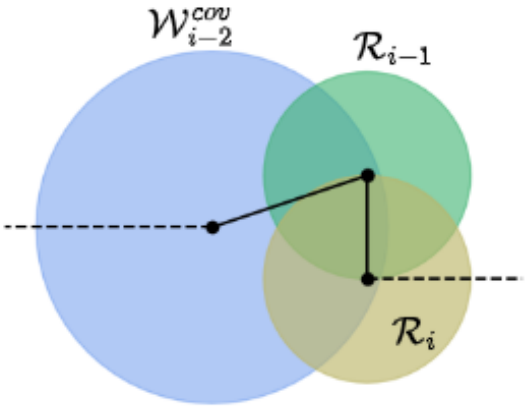


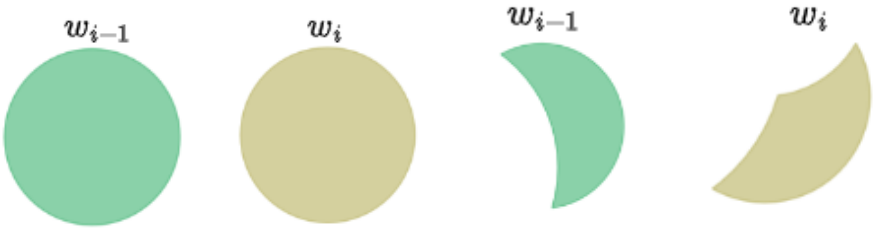
Figure 2.3: Sensing locations in the discretized environment

The visibility regions can intersect, meaning they can cover the same part of the environment. Assume the robot travels from region $\mathcal{R}_{i-1}$ to $\mathcal{R}_i$, see Figure 2.4. The static GSP assumes that the probability of detecting the target object is defined only by the visibility region. In reality, overlapping area $\mathcal{R}_j \cap \mathcal{R}_i, j < i$ was already covered in the previous region $\mathcal{R}_j$. Therefore, the probability of detecting the object in the region $\mathcal{R}_i$ depends on the new

area covered, i.e., the order in which the sensing locations are visited. This motivates the definition of GSP with order-dependent weights.



(a) : Distribution of visibility regions



(b) : Static weights **(c) :** Dynamic weights

Figure 2.4: GSP with static/dynamic weights

2.3 Graph Search Problem with order-dependent weights

In the discrete Mobile Robot Search scenario, we must determine the probability of the target object being present within the interconnected visibility regions through the shortest path graph shown in Figure 2.3. Each vertex in this graph is associated with a weight indicating this probability. GSP was first presented in the context of mobile robot exploration in a priory known environment in [12]. Formally, we define a complete undirected graph $G = (V, E, d, w, s)$ where

- $V = \{v_0, v_1, \ldots, v_n\}$, $|V| = n + 1$ represents the finite set of vertices.
- $E = \{e_0, e_1, \ldots, e_M\}, e_i = e_{kl} = (v_k, v_l)\, \forall v_k, v_l \in V : k \neq l$ stands for the set of edges;
- $d : E \to R^+$ $d(v_i, v_j) \equiv d_{ij}$ denotes the time required for the robot to travel from v_i to v_j.

- $w : V \to [0, 1]$ - vertex weight, probability of detection of the object in the vertex v_i
- s denotes the start vertex, depot.

The subset (V, E, d) is the shortest path graph G_s of the environment, where each vertex v_i is the sensing location, defining a corresponding visibility region $\mathcal{R}_i$ which the robot potentially needs to visit to find the target object.
The robot's route through the environment follows a Hamiltonian path, which includes all possible vertex permutations $\mathbf{x} = (v_0, v_{\pi(1)}, ..., v_{\pi(n)}) = (v_0, x_1, ..., x_n)$, starting from the depot $s = v_0$. Here, x_i represents the vertex at position i along the path from the permutation π. When following the order specified by the path $\mathbf{x}$, the time taken to reach vertex x_i from the depot is determined by summing the costs of the edges defined by this path.

$$\delta_i = d(s, x_i, \mathbf{x}) == \sum_{k=0}^{i-1} d(x_k, x_{k+1}) \tag{2.11}$$

As the robot moves forward, the regions it traverses are subtracted from the unexplored area, a concept commonly referred to as "clipping". Additionally, the newly explored areas add to the weight of the respective vertex, thereby adjusting its importance within the environment.

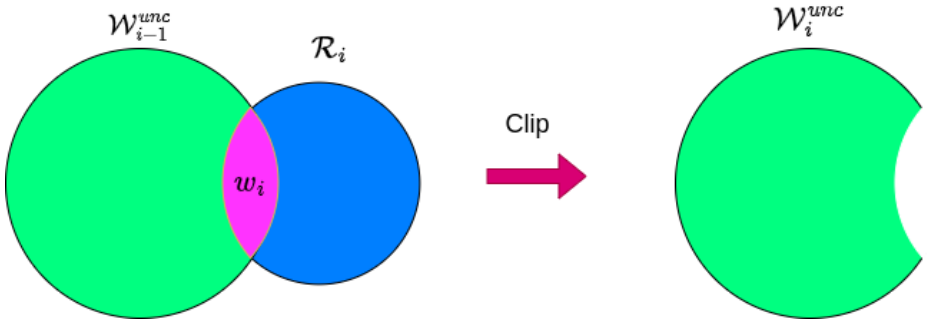


Figure 2.5: Region "clipping" from the path

The uncovered area is dependent on the vertices visited before, defined as

$$\mathcal{W}_{i-1}^{unc} = \mathcal{W} \setminus \bigcup_{k=0}^{i-1} \mathcal{R}_k \tag{2.12}$$

It implies that the weight w_i is also dependent on the sequence of vertices visited before exploring the region $\mathcal{R}_i$

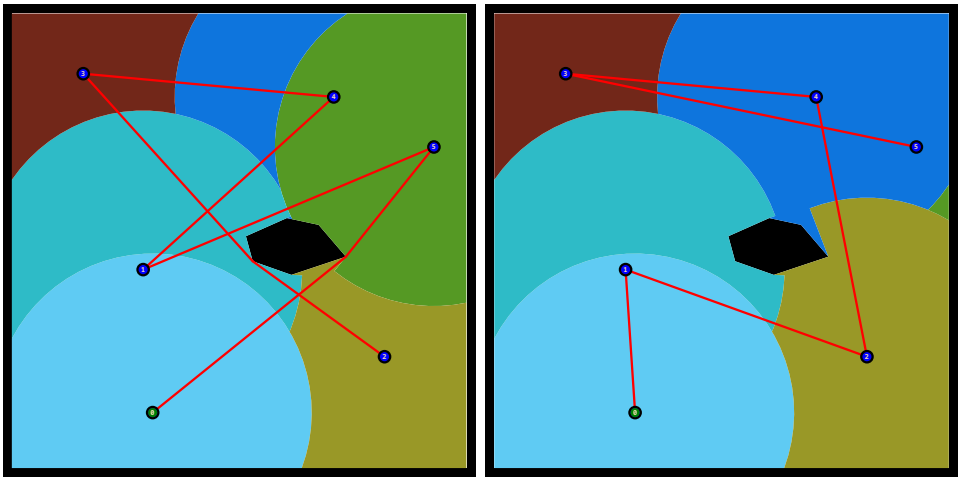
$$w_i = p(\delta_i|\mathbf{x}) = \frac{a(\mathcal{R}_i \cap \mathcal{W}_{i-1}^{unc})}{a(\mathcal{W})} = \frac{a\left(\mathcal{R}_i \cap (\mathcal{W} \setminus \bigcup_{k=0}^{i-1} \mathcal{R}_k)\right)}{a(\mathcal{W})} \tag{2.13}$$

The probability w_i, remains unknown beforehand and for the whole path $\mathbf{x}$ respects the property of cumulative probability distribution function

$$\sum_{i=0}^{n} w_i = 1, \tag{2.14}$$

The order of exploring the graph with the Hamiltonian path is dependent on the selected permutation, which leads us to the new problem, the Graph Search Problem, with order-dependent weights. The expected time for visiting each vertex defines the cost of the solution $\mathbf{x}$

$$\mathbb{E}(T|\ \mathbf{x}) \equiv T^{exp}(\mathbf{x}) = \sum_{i=1}^{n} \delta_i\, w_i = \sum_{i=1}^{n} w_i \sum_{j=1}^{i} d(x_{j-1}, x_j) \tag{2.15}$$



(a) : $T^{exp}(\mathbf{x}) = 28.1$ **(b) :** $T^{exp}(\mathbf{x}) = 15.7$

Figure 2.6: Illustration of GSP with order-dependent weights for the best/worst solution

Then, the task is to find the permutation of vertices starting from v_0 that minimizes the overall expected time with the probability assigned to each vertex in the path

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}}\, T^{exp}(\mathbf{x}) \tag{2.16}$$

Chapter 3

Solution Approach

This chapter outlines approaches for solving the Graph Search Problem with order-dependent weights. It is divided into three sections to address different stages of the solution process systematically. The first section introduces constructive heuristics suitable for generating a complete solution from scratch. The second section delves into improving heuristics, concentrating on enhancing existing solutions. Lastly, the third section covers general-purpose metaheuristics designed to explore potential solutions effectively.

3.1 Constructive heuristics

Constructive heuristics create a good initial solution within a limited timeframe in a step-by-step manner, laying the foundation for further improvement. They start with an empty or partial solution and incrementally construct a complete one.
As the output of the constructive algorithm, we consider the tour $\mathbf{x} = (v_0, x_1, ..., x_n) = (v_0, v_{\pi(1)}, ..., v_{\pi(n)}), v_i \in V$, which represents a possible permutation π of vertices with the fixed v_0, input vertices V are obtained from the distance graph $G = (V, E, d)$ which also considered to be a global parameter. The quality of the tour $\mathbf{x}$ is determined by its expected cost $T^{exp}(\mathbf{x})$, where better solutions correspond to smaller costs.
The general constructive heuristic is described in Algorithm 1. In each step, $i \in \{1, 2, ..., n\}$, the next vertex x_i is selected from the candidate list $\mathcal{CL}$ of unassigned nodes (line 4) based on specific criteria, which vary depending on the heuristic being used. At the end of the iteration, the chosen vertex is removed from the candidate list. The initial solution $\mathbf{x}$ is constructed by iteratively adding unassigned vertices. This process ensures feasibility, as the candidate list is updated sequentially to maintain valid selections.

Algorithm 1: Constructive heuristic

Input: Start vertex v_0.
Output: Initial solution $\mathbf{x} = (x_0, ..., x_n)$.

$\mathbf{x} \leftarrow (v_0)$
$\mathcal{CL} \leftarrow \{v_1, ..., v_n\}$
for $i = 1$ *to* N **do**
  $x_i \leftarrow$ `SelectVertex`$(\mathcal{CL})$
  $\mathbf{x} \leftarrow \mathbf{x} \oplus x_i$
  $\mathcal{CL} \leftarrow \mathcal{CL} \setminus x_i$
return $\mathbf{x}$

3.1.1 Fully random

This heuristics involves creating a random permutation of vertices. In this approach, the next vertex x_i is always chosen uniformly randomly from the candidate list. This means that at each step, any vertex from the candidate list has an equal probability of being selected as the next vertex.
This lack of specific guidance leads to inherent indeterminacy. This variance often results in a substantial standard deviation in the initial solution costs. Consequently, some initial solutions may be distant from the optimum, necessitating additional time for the heuristic to converge towards local or global optima. The heuristics is used in subsequent experiments to evaluate the overall performance of metaheuristics.

3.1.2 Greedy

The following heuristic is inspired by the general greedy scheme for GSP [12]. It extends the current path with the vertex with the lowest value of the deterministic path function. For the incomplete path $\mathbf{x}_i = (x_0, x_1, ..., x_{i-1})$ we consider the time of travel along this path δ_{i-1} and the not visited vertex $v_k \in \mathcal{CL}$, the distance $d(x_{i-1}, v_k)$ between the last vertex in the incomplete path and the unassigned vertex, and its respective weight w_i after the path extension. Then we define five different path criteria and their respective functions:

- Added cost after path extension:

$$f_1(\mathbf{x}_i \oplus v_k) = T^{exp}(\mathbf{x}_i \oplus v_k) - T^{exp}(\mathbf{x}_i) = (\delta_{i-1} + d(x_{i-1}, v_k))\, w_i \quad (3.1)$$

- Distance to the next vertex:

$$f_2(\mathbf{x}_i \oplus v_k) = d(x_{i-1}, v_k) \quad (3.2)$$

- Distance-to-weight ratio:

$$f_3(\mathbf{x}_i \oplus x_k) = \frac{d(x_{i-1}, v_k)}{w_i + 0.01} \quad (3.3)$$

- Total travel time-to-weight ratio:

$$f_4(\mathbf{x}_i \oplus v_k) = \frac{\delta_{i-1} + d(x_{i-1}, v_k)}{w_i + 0.01} \tag{3.4}$$

- Weighted distance:

$$f_5(\mathbf{x}_i \oplus v_k) = d(x_{i-1}, v_k)\, w_i \tag{3.5}$$

With one of the selected functions, we determine the best vertex to append to the path:

$$x_i \leftarrow \underset{v_k}{\operatorname{argmin}}\, f_r(\mathbf{x}_i \oplus v_k)\ ,\ v_k \in \mathcal{CL}, r \in \{1,2,3,4,5\} \tag{3.6}$$

The deterministic nature of the heuristic ensures repeatability, meaning the same input will always produce the same output.

3.1.3 Randomized Greedy

We introduce the randomized greedy algorithm to expand the variability of the generated solutions for the same input instance. This approach integrates stochastic elements, allowing better solutions to be obtained.
To illustrate the concept, at step i of the randomized greedy algorithm, let's denote the list of unassigned vertices as $\{y_1, .., y_{n-i}\} \notin \mathbf{x}$. Then, a sorted candidate list contains the nodes sorted by their path function f_r values.

$$\mathcal{CL} = \{z_i = y_{\pi(i)} : \forall j \in \{i+1, ..., n-i\}, f_r(\mathbf{x} \oplus z_i) \leq f_r(\mathbf{x} \oplus z_j)\} \tag{3.7}$$

Then, for the first $m \ll |\mathcal{CL}|$ candidates, probabilities are assigned for selecting the next vertices, represented by the probability list $\mathcal{PL} = (p_1, p_2, ..., p_m)$, where

$$\sum_{i=1}^{m} p_i = 1,\ p_i \in [0,1],\ \forall i > j,\ p_i > p_j, \tag{3.8}$$

The next vertex $z_{\pi(i)}$ is selected randomly from the first m candidates from the sorted candidate list based on its associated probability p_i. The probability list can be generated randomly at each step of adding a new vertex in the constructive heuristic or provided as the additional constant parameter to the heuristic. In our experiments, we utilized $m = 3$ candidates for the selection with the constant probabilities $\mathcal{PL} = \{p_1, p_2, p_3\}$. These probabilities signify selecting the first-best candidate with a probability of p_1, the second-best with p_2, and the third-best with p_3. If fewer than $m = 3$ unassigned vertices remain, we always opt for the first-best choice. This decision is made because the impact on the overall expected cost is relatively minor towards the end of the path.
For the selection process, we generate a random number $p_{rand} \in [0,1]$ and select the vertex with the smallest cumulative probability that exceeds p_{rand}. Mathematically, this is described as follows:

$$k^* \leftarrow \underset{k}{\operatorname{argmin}}(\sum_{i=1}^{k} p_i \geq p_{rand}),\ k \in \{1, 2, ..., m\} \tag{3.9}$$

$$x_{i+1} \leftarrow z_{k^*} \in \mathcal{CL} \tag{3.10}$$

3.2 Improving heuristics

Unlike constructive heuristics, which build solutions from scratch, improving heuristics starts with an initial solution and iteratively enhances it through local modifications, potentially achieving better-quality solutions.

3.2.1 Local Search

To define the improving heuristics, assume the distance graph $G = (V, E, d)$ and the set $\mathcal{H}$ containing all possible Hamiltonian paths originating from vertex v_0, defining the solution state-space. Given the input tour $\mathbf{x} = (x_0 = v_0, x_1, ..., x_n)$, we introduce the operator $\Phi_{op}(\mathbf{x}, \alpha)$, which generates a new tour $\mathbf{x}' \in \mathcal{H}$. This operator requires a fixed-length set of parameters $\alpha = (\alpha_1, ..., \alpha_p)$, influencing the type of change. Each parameter can take the values $\alpha_i \in \{1, 2, ..., n = |V| - 1\}$, which define the index of a vertex in the sequence $\mathbf{x}$. A single set of parameters can also be considered as a p-dimensional vector of natural numbers less or equal to n: $\alpha \in \mathcal{N}^p_{\leq n}$ Then all possible sets of operator parameters, denoted as U_{op}, form a p-ary Cartesian product over all possible values the parameters can obtain $U_{op} = \alpha_1 \times \alpha_2 ... \times \alpha_p$. A single parameter set $\alpha \in U_{op}$ defines one possible neighbor obtained through the operator while considering all possible sets of parameters defines the set of neighboring tours or the neighborhood of the input tour $N^{\mathbf{x}}_{op} = \Phi_{op}(\mathbf{x}, U_{op})$.

Let's clarify these concepts using the N-queens problem as an example [30]. In this problem, the goal is to place N queens on an N×N chessboard so that no two queens threaten each other. This can be formulated as a CO problem in which the objective is to minimize the number of pairs of queens that attack each other, with the optimal solution being zero such pairs. A solution to this problem is a particular arrangement of queens on the chessboard. The local search operator defines how to explore neighboring solutions from a given input solution. One such operator Φ_{op} could be to take a queen from column i and move it to row j. This action transforms the current board configuration into a neighboring configuration. All feasible transformations for $i, j \in \{1, 2, ..., N\}$ define the neighborhood, blue transformations in Figure 3.1.

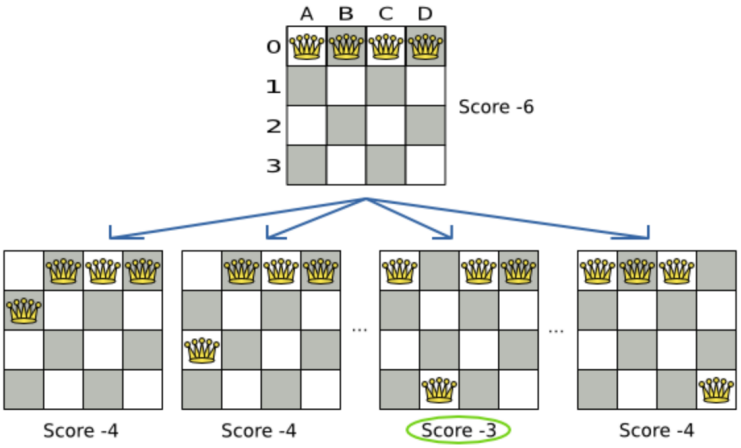


Figure 3.1: N-queens problem

The next important part is that the improving heuristic selects the improved neighbor $\mathbf{x}^{impr} \in N_{op}^{\mathbf{x}}$ with a better cost or returns the input tour if no better tours exist in the neighborhood. Based on [31] [32], we introduce the Local Search Procedure (LSP), see Algorithm 2, with two improvement types $\mathcal{IT}$ that define the deterministic criteria for selecting a better neighbor as the output of the improving heuristic:

- *Best* (Hill Climbing): Selects the best neighbor from the neighborhood.
- *First*: Chooses the first obtained neighbor that is superior to the input tour

The knowledge of all possible operator parameters U_{op} allows us to define the LSP. The Iterative Local Search (ILS), described in Algorithm 3, extends the local search. It iteratively calls the local search procedure for the improved solutions as the input ones until no better solution is found or the stopping criterion is met. The drawback is that it can get stuck in the local minimum.

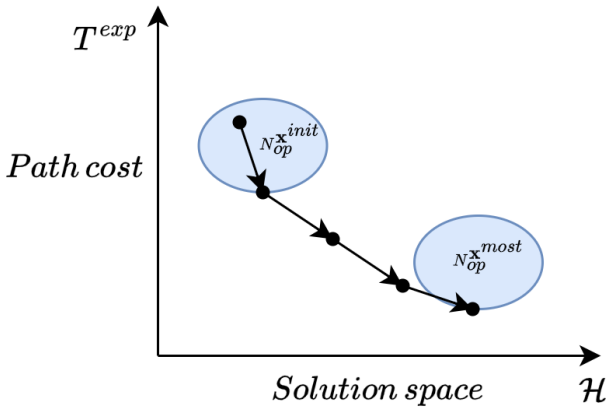


Figure 3.2: ILS: no better neighbour for the final solution $\mathbf{x}^{most}$

Algorithm 2: Local Search Procedure (LSP)

Input: Input tour $\mathbf{x} = (x_0, ..., x_n)$.
Local search operator Φ_{op}
Parameters: Cost function $T^{exp}(\mathbf{x})$,
Improvement type $\mathcal{IT} \in \{Best, First\}$
Output: Improved path x^{impr}

$\mathbf{x}^{impr} \leftarrow \mathbf{x}$
for $\alpha \in U_{op}$ **do**
  $\mathbf{x}^{neigh} \leftarrow \Phi_{op}(\mathbf{x}, \alpha)$
  if $T^{exp}(\mathbf{x}^{neigh}) < T^{exp}(\mathbf{x}^{impr})$ **then**
    $\mathbf{x}^{impr} \leftarrow \mathbf{x}^{neigh}$
    if $\mathcal{IT} = First$ **then**
      break
return $\mathbf{x}^{impr}$

Algorithm 3: Iterative Local Search (ILS)

Input: Input route $\mathbf{x} = (x_0, ..., x_n)$.
Local search operator Φ_{op}
Output: Most improved tour x^{most}.

$\mathbf{x}^{impr}, \mathbf{x}^{most} \leftarrow \mathbf{x}$
$i = 0$
repeat
  $\mathbf{x}^{most} \leftarrow \mathbf{x}^{impr}$
  $\mathbf{x}^{impr} \leftarrow LSP(\mathbf{x}^{most}, \Phi_{op})$
  $i \leftarrow i + 1$
until $\mathbf{x}^{impr} \neq \mathbf{x}^{most}$
return $\mathbf{x}^{\mathbf{most}}$

3.2.2 Basic Operators

For the GSP with order-dependent weights, we employ two local search operators that allow the generation of the different neighborhoods for the input tour. Using them, we define two basic LSPs based on Algorithm 2.

Insert

This operator is inspired as the extension of the swap operator described in [13]. The Insert operator can be described as follows: “Remove the i-th vertex from the route $\mathbf{x}$ and insert it at position j”.

Figure 3.3: Application of the Insert operator

Formally, given an input tour represented by a sequence of cities $\mathbf{x} = (x_0, x_1, ..., x_i, ..., x_j, ..., x_n)$. All possible Insert operator parameters are $\mathcal{U}_{ins} = ((i,j) : i \neq j, (i,j) \in \mathbb{N}^2_{\leq n})$. Then, for the two accepted parameters (i,j), the application of the Insert operator results in a new tour

$$\mathbf{x}' = \Phi_{ins}(\mathbf{x}, (i,j)) = (x_0, x_1, ..., x_{i-1}, x_{i+1}, ..., x_j, x_i, x_{j+1}, ..., x_n) \tag{3.11}$$

The application of the Insert operator is shown in Figure [3.3]: sub-sequence (x_{i-1}, x_i, x_{i+1}) and the edge (x_j, x_{j+1}) are removed from the tour and replaced by the new sub-sequence (x_{i-1}, x_{i+1}) and the edge (x_j, x_i, x_{j+1}). The number of unique neighbors of the Insert operator is defined: remove the element from one of the n positions and place it in one of $n-1$ positions minus $n-1$ same sequences :

$$|N^{\mathbf{x}}_{ins}| = n(n-1) - (n-1) = (n-1)^2 \tag{3.12}$$

2-Opt

Generally, the k-Opt operator is a local search technique that involves removing k edges from the input tour and replacing them with k new edges so that the orientation of the path is preserved [33]. Our work considers a 2-Opt operator that removes exactly 2 non-adjacent edges and adds 2 new edges in the tour.

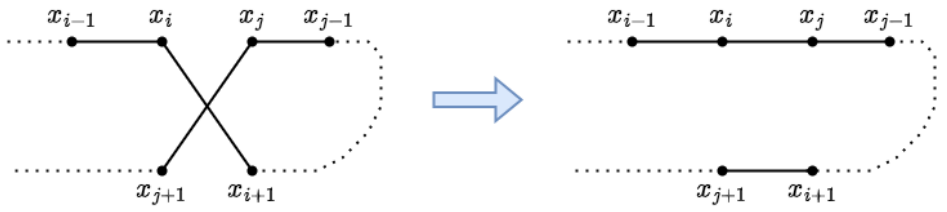


Figure 3.4: Application of the 2-Opt operator

For the input sequence of vertices $\mathbf{x} = (x_0, x_1, ..., x_i, ..., x_j, ..., x_n)$ we define the 2-Opt operator parameters as $\mathcal{U}_{2opt} = ((i,j) : j - i > 1, (i,j) \in \mathbb{N}^2_{\leq n})$.

Then by application of 2-Opt operator for parameters (x_i, x_j) we get a new tour

$$\mathbf{x}' = \Phi_{2opt}(\mathbf{x}, (i,j) = (x_0, x_1, ..., x_i, x_j, x_{j-1}, ..., x_{i+1}, x_{j+1}, ..., x_n) \tag{3.13}$$

In Figure [3.4], we observe that two edges (x_i, x_{i+1}) and (x_j, x_{j+1}) are removed from the path and replaced by edges (x_i, x_j) and (x_{i+1}, x_{j+1}), respectively. This replacement is necessary to maintain the path's orientation. There are no other feasible ways to reconnect the edges while preserving the path's direction. Furthermore, the 2-Opt operator can be conceptualized as reversing the sub-sequence $(x_{i+1}, \ldots, x_j)$ within the input tour $\mathbf{x}$. For the index i we can reverse $n - i$ sub-sequences after:

$$|N_{2opt}| = \sum_{i=1}^{n-1} n - i = \frac{n^2 - n}{2} \tag{3.14}$$

3.2.3 Swap of two consecutive vertices

The drawback of Algorithm 2 is that it requires calculating the cost T^{exp} from Equation 2.15 for every neighbor $\mathbf{x} \in N_{op}^x$, meaning we need to iterate over the tour to compute the weight of each vertex and consequentially the path cost, which takes $\mathcal{O}(n)$. This slows down the algorithm because such calculations need to be performed $|N_{op}^x|$ times to evaluate every neighbor, which leads to the overall complexity of $\mathcal{O}(n \cdot |N_{op}^x|)$. In this section, we propose reducing the cost computation of the Insert and 2-Opt operators using the swap operator, which employs the cost calculation of a neighbor in a constant time.

Operator Definition

We define the swap of two consecutive vertices as "Take the i-th vertex in the tour and swap it with the consecutive neighbor, vertex at the position i+1.". The possible swap operator parameters are $\mathcal{U}_{ins} = ((i) : (x_i) \in \mathbb{N}_{\leq n-1})$. For given a sequence $\mathbf{x} = (x_0, x_1, ..., x_i, x_{i+1}, ..., x_n)$ the result tour is

$$\mathbf{x}' = \Phi_{swap}(\mathbf{x}, (x_i)) = (x_0, x_1, ..., x_{i+1}, x_i, ..., x_n) \tag{3.15}$$

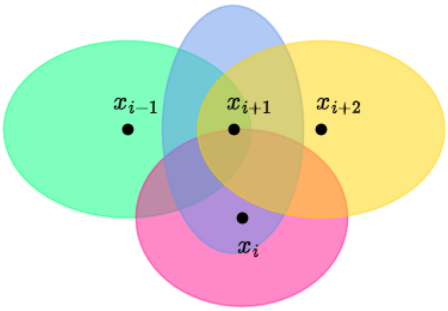
The swap operator is illustrated in Figure 3.5. The partition of path $(x_{i-1}, x_i, x_{i+1}, x_{i+2})$ is removed from the path and replaced with the new partition $(x_{i-1}, x_{i+1}, x_i, x_{i+2})$. The operator's neighborhood size is

$$|N_{swap}^{\mathbf{x}}| = n - 1 \tag{3.16}$$

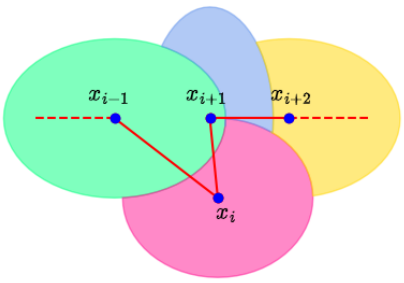
Figure 3.5: Application of the swap operator

Weight Modification

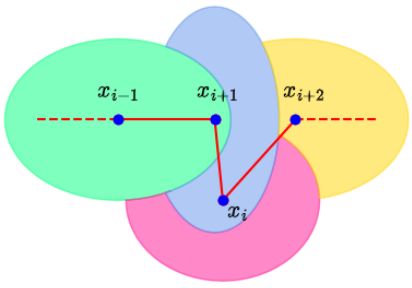
Now, let's examine the weight adjustments when we execute a swap between two consecutive vertices, considering a sequence $\mathbf{x} = (x_0, x_1, ..., x_i, x_{i+1}, ..., x_n)$ and the new swapped sequence $\mathbf{x}' = \Phi_{swap}(\mathbf{x}, x_i)$ illustrated in Figure 3.6.



(a) : Distribution of visibility regions



(b) : Areas contributing to the weights in the initial sequence



(c) : Areas contributing to the weights after the swap of vertices x_i, x_{i+1}

Figure 3.6: Illustration of weights modification for the swap operator

We will assume that $\mathcal{W}_{i-1}^{unc}(\mathbf{x})$ denotes the uncovered environment after visiting $i-1$ vertices over the path $\mathbf{x}$. The area that contributes to the weight of vertex x_i over path $\mathbf{x}$ is denoted $\mathcal{W}_i$. Additionally, we'll relax on computing the relative weight, defined in Equation (2.13), and consider the absolute weight over the path $\mathbf{x}$ obtained by vertex x_i, denoted as $w_i = a(\mathcal{W}_i)$. Since the environment's total area remains constant, we can omit it for simplicity of notation.

Equation (2.12) shows that the uncovered area depends on the visited vertices. This implies that the uncovered area before and after swapped vertices remains the same for both paths

$$\mathcal{W}_{i-1}^{unc}(\mathbf{x}) = \mathcal{W}_{i-1}^{unc}(\mathbf{x}') \tag{3.17}$$

$$\mathcal{W}_{i+1}^{unc}(\mathbf{x}) = (\mathcal{W}_{i-1}^{unc}(\mathbf{x}) \setminus \mathcal{R}_i) \setminus \mathcal{R}_{i+1} = (\mathcal{W}_{i-1}^{unc}(\mathbf{x}') \setminus \mathcal{R}_{i+1}) \setminus \mathcal{R}_i = \mathcal{W}_{i+1}^{unc}(\mathbf{x}') \tag{3.18}$$

When two clip operations are performed in a different order, the resulting uncovered area, denoted as $\mathcal{W}_{i+1}^{unc}(\mathbf{x})$ and equivalent to $\mathcal{W}_{i+1}^{unc}(\mathbf{x}')$, remains unaffected. This is illustrated in Figure 3.7.

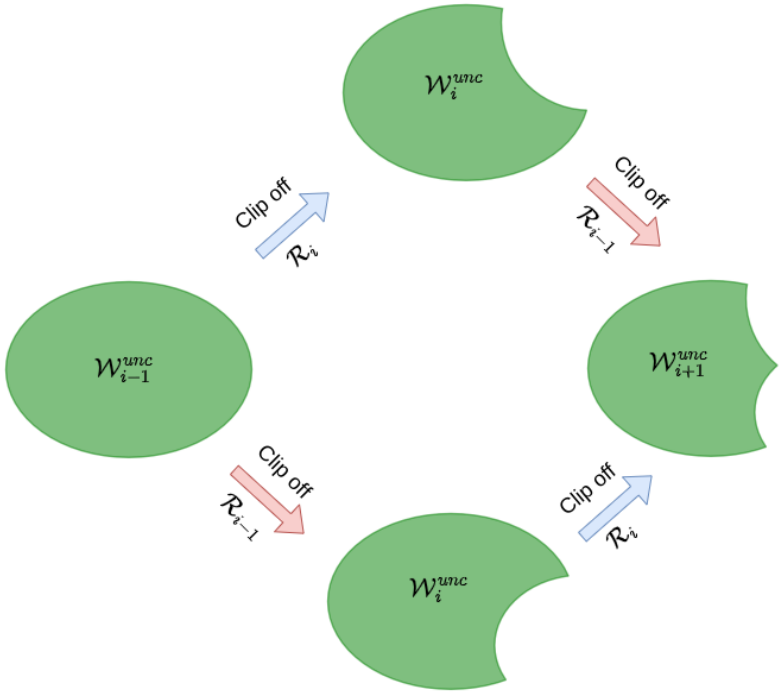


Figure 3.7: Update of uncovered environment for two sequential clips

This allows us to localize problem to the visibility regions $\mathcal{R}_i, \mathcal{R}_{i+1}$ and the uncovered area $\mathcal{W}_{i-1}^{unc} \equiv \mathcal{W}_{i-1}^{unc}(\mathbf{x})$. For clarity, we'll employ the following designations depicted in Figure 3.8:

- $\mathcal{R}_i^{int} = \mathcal{W}_{i-1}^{unc} \cap \mathcal{R}_i \cap \mathcal{R}_{i+1}$ - intersection of uncovered environment and visibility regions.
- $\mathcal{R}_i^{pure} = \mathcal{W}_{i-1}^{unc} \cap \mathcal{R}_i \setminus \mathcal{R}_i^{int}$ - pure intersection of region $\mathcal{R}_i$ with the uncovered environment without intersection with region $\mathcal{R}_{i+1}$.
- $\mathcal{R}_{i+1}^{pure} = \mathcal{W}_{i-1}^{unc} \cap \mathcal{R}_{i+1} \setminus \mathcal{R}_{i+1}^{int}$ - pure intersection for region $\mathcal{R}_{i+1}$

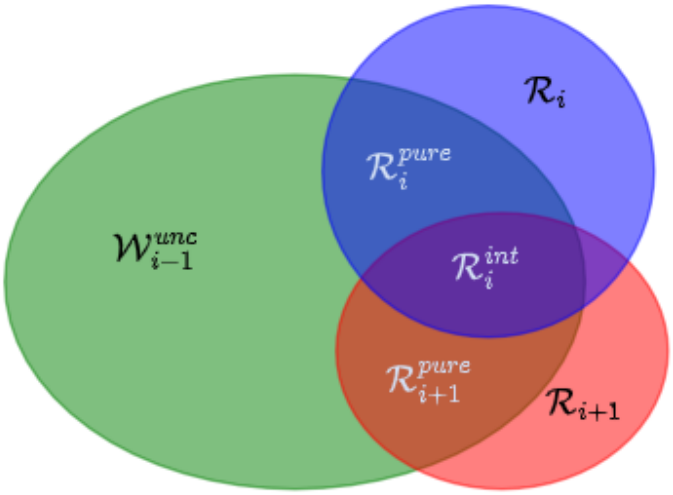


Figure 3.8: Illustration of used notations

Also, we consider for the path $\mathbf{x}$ the vector of precomputed newly covered areas that contribute to the weight of the vertex x_i, $\mathbf{W} = [\mathcal{W}_1, ..., \mathcal{W}_n]$ which can be precomputed in $\mathcal{O}(n)$. All polygon operations are considered to be constant. Then for the path $\mathbf{x}$ the respective weights are

$$w_i = a(\mathcal{W}_i) = a(\mathcal{R}_i^{pure} \cup \mathcal{R}_i^{int}) \tag{3.19}$$
$$w_{i+1} = a(\mathcal{W}_{i+1}) = a(\mathcal{R}_{i+1}^{pure}) \tag{3.20}$$

Due to $\mathcal{R}_i^{pure} \cap \mathcal{R}_i^{int} = \emptyset$ the (3.19) can be rewritten as the sum of absolute areas

$$w_i = a(\mathcal{R}_i^{pure}) + a(R_i^{int}) \tag{3.21}$$

For the new sequence $\mathbf{x}'$ the weights would be

$$w_i' = a(\mathcal{W}_i') = a(\mathcal{R}_i^{pure}) \tag{3.22}$$
$$w_{i+1}' = a(\mathcal{W}_{i+1}') = a(\mathcal{R}_{i+1}^{pure}) + a(\mathcal{R}_i^{int}) \tag{3.23}$$

Based on these equations, we can provide the update rule for the respective areas

$$\mathcal{W}_i^{'} = \mathcal{W}_i \setminus \mathcal{R}_i^{int} \tag{3.24}$$
$$\mathcal{W}_{i+1}^{'} = \mathcal{W}_{i+1} \cup \mathcal{R}_i^{int} \tag{3.25}$$

Let's denote $\Lambda_w \equiv a(R_i^{int})$. Then, the weights update would be

$$w_i' = w_i - \Lambda_w \tag{3.26}$$
$$w_{i+1}' = w_i + \Lambda_w \tag{3.27}$$

From the previous equations, we can see that

$$w_i' + w_{i+1}' = w_i + w_{i+1} \tag{3.28}$$

To update the weight, we need information about the intersection $\mathcal{R}_i^{int}$. In practice, it has been shown that the accuracy of computing this area is compromised by the imprecise nature of polygonal operations, resulting in additional errors in determining its value. Therefore, it's more practical to compute it dynamically using the constant areas of visibility regions. To facilitate this, we introduce another globally precomputed parameter, the set of pair regions intersections $\mathbf{R}_{ij}^{int} = \{\mathcal{R}_i \cap \mathcal{R}_j : i \neq j\}$, which can be precomputed in $\mathcal{O}(n^2)$ time. Using this parameter, the area R_i^{int} can be obtained for the path $\mathbf{x}$ as follows:

$$R_i^{int} = \mathcal{W}_i \cap R_{ij}^{int} \tag{3.29}$$

To summarize, the weights can be updated in constant time, and their update depends on the area R_i^{int} and its numerical value Λ_w. For this purpose, we require the constant global precomputed parameter $\mathbf{R}_{ij}^{int}$ and the path parameter $\mathbf{W}$, which should also be updated to facilitate further swaps on the new route $\mathbf{x}'$.

Cost Difference

To understand how the costs of two paths are related, given that we know how the respective weights of paths $\mathbf{x}$ and $\mathbf{x}'$ can be updated, we aim to find the cost difference between them, assuming that the cost of the current path is already computed. We follow the same derivation process as in the paper [13], where this difference was derived for GSP with static weights. For this purpose, we define the cost difference between two tours obtained with a single swap as:

$$\Delta_{swap}(\mathbf{x}, i) = T^{exp}(\Phi_{swap}(\mathbf{x}, i)) - T^{exp}(\mathbf{x}) = c(\mathbf{x}') - c(\mathbf{x}) \tag{3.30}$$

We also denote $d(x_i, x_j) \equiv d_{i,j}$. Then, the path cost from Equation (2.15), which defines how the i-th weight contributes to the path from the source,

can be rewritten as which weights contribute to the i-th edge over the entire path

$$c(\mathbf{x}) = \sum_{i=1}^{n} w_i \sum_{j=1}^{i} d_{i-1,i} = \sum_{i=1}^{n} d_{i-1,i}\gamma(i) \tag{3.31}$$

where

$$\gamma(i) = \sum_{k=i}^{n} w_i \tag{3.32}$$

is the sum of weights starting from vertex i on the path associated with the edge $d_{i-1,i}$.The swap operator involves removing the three edges (x_{i-1}, x_i), (x_i, x_{i+1}), (x_{i+1}, x_{i+2}) and replacing them with (x_{i-1}, x_{i+1}), $(x_{i+1}x_i)$, (x_i, x_{i+2}), with only the weights w_i and w_{i+1} being altered. Therefore, the tour cost before the swap can be calculated as follows:

$$c(\mathbf{x}) = \sigma_1 + \omega_1 + \omega_2 + \omega_3 + \sigma_2 \tag{3.33}$$

with the terms

$$\sigma_1 = \sum_{k=1}^{i-1} d_{k-1,k}\gamma(k) \tag{3.34}$$

$$\omega_1 = d_{i-1,i}\,\gamma(i) \tag{3.35}$$

$$\omega_2 = d_{i,i+1}\,\gamma(i+1) \tag{3.36}$$

$$\omega_3 = d_{i+1,i+2}\,\gamma(i+2) \tag{3.37}$$

$$\sigma_2 = \sum_{k=i+3}^{n} d_{k-1,k}\,\gamma(k) \tag{3.38}$$

The resulting route cost is

$$c(\mathbf{x}') = \sigma_1 + \omega_1' + \omega_2' + \omega_3' + \sigma_2 \tag{3.39}$$

Each term in this expression can be further defined based on the Equations (3.26), (3.27), (3.36):

$$\omega_1' = d_{i-1,i+1}\,(w_{i+1}' + w_i' + \gamma(i+2)) = \gamma(i)\,d_{i-1,i+1} \tag{3.40}$$

$$\omega_2' = d_{i+1,i}(w_i' + \gamma(i+2)) = d_{i,i+1}(w_i - \Lambda_w - w_{i+1}) + \omega_2 \tag{3.41}$$

$$\omega_3' = \gamma(i+2)\,d_{i,i+2} \tag{3.42}$$

Then, the cost improvement after the swap:

$$\Delta_{swap}(\mathbf{x}, i) = \Lambda_1\gamma(i) + \beta + \Lambda_2\gamma(i+2) \tag{3.43}$$

with the coefficients

$$\Lambda_1 = d_{i-1,i+1} - d_{i-1,i} \tag{3.44}$$

$$\beta = d_{i+1,i}(w_i - \Lambda_w - w_{i+1}) \tag{3.45}$$

$$\Lambda_2 = d_{i,i+2} - d_{i+1,i+2} \tag{3.46}$$

The special case is a swap of the last two vertices: $x_{i+1} = x_n \implies \Lambda_2 = 0$. All required coefficients from the equation are computed in constant time and allow to obtain the cost of the new path

$$T^{exp}(\mathbf{x}') = T^{exp}(\mathbf{x}) + \Delta_{swap}(\mathbf{x}, i) \tag{3.47}$$

We require precomputed $\gamma(i)$ values for all i in the range $[0, n-1]$ to find the cost difference in constant time. For this purpose, we introduce another path parameter: the vector of cumulative weight sums from vertex x_i, denoted as $\mathbf{\Gamma} = [\gamma_0, ..., \gamma_{n-1}]$, where $\gamma_i \equiv \gamma(i)$. It can be precomputed in $\mathcal{O}(n)$ time. To facilitate further swaps on the path $\mathbf{x}'$, this parameter should be updated as follows:

$$\gamma'_i = \gamma_i - w_{i+1} \tag{3.48}$$
$$\gamma'_{i+1} = \gamma_{i+1} + w_i + \Lambda_w \tag{3.49}$$

This approach enables us to perform the cost update after swapping neighboring vertices in constant time.

Algorithm

For the path $\mathbf{x}$, we introduce a set of parameters $\mathcal{X} = (T^{exp}, \mathbf{W}, \mathbf{\Gamma})$, which we need to store and update to perform swaps in constant time. We present the precomputation process in Algorithm 4. The variables T_{exp}, δ, and S_w (line 2) represent the expected cost, time of travel from the starting vertex, and the sum of weights obtained on the path, respectively. Then, the iteration over the input tour $\mathbf{x}$ is performed, where these variables are updated based on newly obtained weight and distance from the previous vertex to the current. The area contributing to the weight (line 8) is obtained in every iteration, and the vertex visibility region is clipped off from the uncovered environment (line 9). The second for loop represents the calculation of the cumulative weight sum from the vertex where we utilize the already computed weight sum S_w (line 17). The swap procedure based on precomputed parameters is presented in Algorithm 5 and based on previously derived equations.

The key advantage is that the parameters of the next sequence obtained with the swap are derived from the swap procedure, eliminating the need for recomputation. For instance, we apply the swap operator k times using the recurrent form:

$$\mathbf{x}^{(k)} = \Phi^{(k)}_{swap}(\mathbf{x}) \tag{3.50}$$

A naive implementation would require recomputing the expected cost every time we generate the next neighbor, resulting in a $\mathcal{O}(nk)$ complexity. However, the new approach reduces the complexity to $\mathcal{O}(n + k)$ because the swap procedure allows us to obtain the cost of the new path in constant time, requiring only one precomputation for the initial sequence. This approach yields significantly faster computations, particularly in cases where $k \gg n$.

Algorithm 4: Precompute Path Parameters (*PrecomputeParameters*)

Input: Tour $\mathbf{x} = (x_0, ..., x_n)$.
Parameters: Visible environment $\mathcal{W}$,
Time of travel $\delta_{i,j}$ between vertices x_i, x_j
Output: Path parameters $\mathcal{X}$.

$\mathbf{W}, \mathbf{\Gamma} \leftarrow []$
$T^{exp}, \delta, S_w \leftarrow 0$
$\mathcal{W}^{unc} \leftarrow \mathcal{W}$
for $i = 0$ *to* N **do**
  $t \leftarrow 0$
  if $i \neq 0$ **then**
    $t \leftarrow \delta_{i-1,i}$
  $\mathcal{W}_i \leftarrow \mathcal{R}_i \cap \mathcal{W}^{unc}$
  $W^{unc} \leftarrow W^{unc} \setminus \mathcal{R}_i$
  $\mathbf{W} \leftarrow \mathbf{W} \oplus \mathcal{W}_i$
  $w_i \leftarrow \frac{a(\mathcal{W}_i)}{a(\mathcal{W})}$
  $T^{exp} \leftarrow \delta \cdot w_i$
  $S_w \leftarrow S_w + w_i$
  $\delta \leftarrow \delta + t$
for $i = 0$ *to* $N - 1$ **do**
  $\gamma_i \leftarrow S_w - w_i$
  $S_w \leftarrow S_w - w_i$
  $\Gamma \leftarrow \Gamma \oplus \gamma_i$
return $(T^{exp}, \mathbf{W}, \mathbf{\Gamma})$

Algorithm 5: Swap Procedure (*PerformSwap*)

Input: Index of vertex to swap i, tour $\mathbf{x}$, path parameters $\mathcal{X} = (T^{exp}, \mathbf{\Gamma}, \mathbf{W})$.
Output: Improved path x^{impr}.

$\mathcal{W}_i, \mathcal{W}_{i+1} \leftarrow \mathbf{W}_{i,i+1}$
Compute $\Lambda_w, \mathcal{R}_i^{int}$ using Equation 3.29
$\gamma_i, \gamma_{i+2} \leftarrow \mathbf{\Gamma}_{i,i+2}$
Calculate Δ_{swap} based on Equation (3.43) and coefficients from Equations (3.44), (3.46), (3.46)
Update the path parameters $\mathcal{X}$ based on Equations (3.47) (3.24), (3.25) (3.48),(3.49)
Swap elements x_i, x_{i+1} in the tour $\mathbf{x}$
return $\mathbf{x}, \mathcal{X}$

State-Space

We can generate all possible sequence permutations by recurrently swapping two consecutive vertices from Equation (3.50). The provided Figure 3.9 illustrates the state space of all paths $\mathcal{H}$, i.e., permutations for the sequence $\mathbf{x}^{\mathbf{init}} = (0, 1, 2, 3, 4)$ and starting vertex 0.

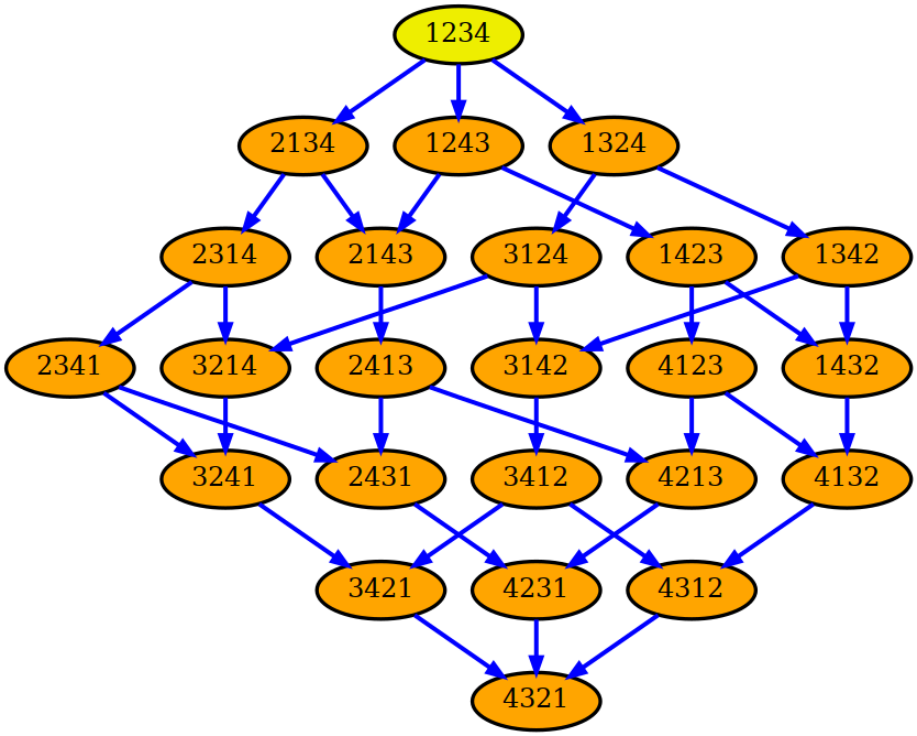


Figure 3.9: Swap state-space for n = 4

We can conceptualize the state-space as the graph $G = (V_s, E_s, \mathbf{x}^{init})$, where each permutation $\mathbf{x} \in \mathcal{H}$ and $\mathbf{x} \in V_s$ is a node in a directed graph, with the initial sequence serving as the root. Traversing along edges E_s, representing the swaps to perform, allows us to reach neighboring sequences. The distance between sequences, denoted by $h(\mathbf{x}^{\mathbf{init}}, \mathbf{x}), \mathbf{x}^{neigh} \in V_s$, reflects the depth of the graph and number of swaps to perform. For example, between the initial sequence and its reversed counterpart, the graph's depth is 6, indicating the necessity of 6 swaps to achieve this transformation. As the size of the initial sequence increases, the corresponding state space also expands.
This implies that we can determine the expected cost and other parameters of the neighbors of any node in the graph in $h(\mathbf{x}^{\mathbf{init}}, \mathbf{x})$ time. Additionally, since the entire state space can be generated from the initial sequence, and all the neighborhoods obtained by other operators lie within this state space, we have the potential to explore all sequences generated by other operators. This can be achieved by performing predefined recurrent swaps of two consecutive vertices starting from the initial one.

3.2.4 Proposed local search procedures

The swap state-space can also be defined for a specific depth h as $N^h_{swap} = \{\Phi^{(k)}_{swap}(\mathbf{x}, (i)) : i \in \mathbb{N}_{\leq n}, k \in \mathbb{N}_{\leq h}\}$. For the maximal depth h_{max}, $N^{h_{max}}_{swap} = \mathcal{H}$. In simpler terms, these neighborhoods represent the unique paths starting from the initial sequence $\mathbf{x}$ and extending to a depth h. As we move along these paths, we generate the permutations.
Suppose an operator *op* defines the neighborhood as $N^{\mathbf{x}}_{op}$. Our goal is to find a swap neighborhood $N^h_{swap} \equiv N^{op}_{swap}$ such that $N^{\mathbf{x}}_{op} \subseteq N^{op}_{swap}$. We aim to find tours obtained by the swap that include all permutations the operator *op* can achieve. This does not mean that all neighbors in N^{op}_{swap} are also in the neighborhood of the operator $N^{\mathbf{x}}_{op}$, which implies $|N^{op}_{swap}| \geq |N^{\mathbf{x}}_{op}|$.
If we understand how to generate the neighborhood for the operator *op* using swaps, we can propose a local search procedure that determines the expected cost of each neighbor from path parameters $\mathcal{X}$ on the fly, given that the swap of two neighboring vertices has constant complexity. The maximum computational speedup is achieved when $|N^{op}_{swap}| = |N^{\mathbf{x}}_{op}|$.

Swap-Based Insert

For the Insert operator neighborhood, the objective is to develop a modification mechanism that explores the entire Insert neighborhood. The green nodes on the Figure [3.10] represent the permutations within the Insert neighborhood

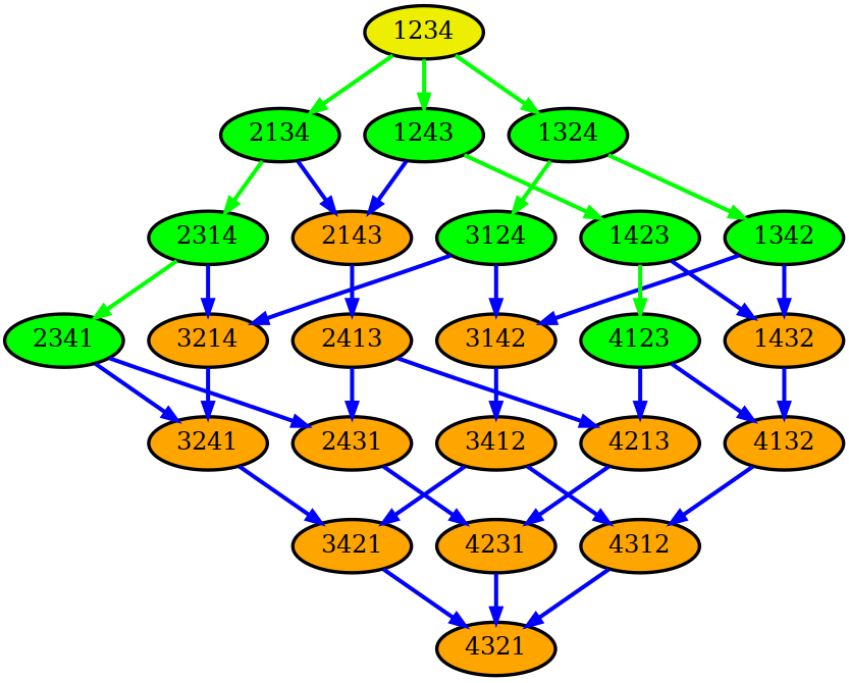


Figure 3.10: Illustration of the Insert neighborhood in state-space

We propose the following approach for the sequence $\mathbf{x}^{init}$, see Figure [3.11]:

- Swap the initial sequence pairs $x_{frw} = x_j, x_{bckw} = x_{j+1}, \forall j = 1, ..., n-1$. Obtain a new sequence $\mathbf{x}' = \Phi_{swap}(\mathbf{x}, j)$ (green nodes).
- Bubble element x_{frw} in the forward direction to the last position in the sequence $\mathbf{x}'$ using swaps (grey nodes).
- Bubble x_{bckw} in the backward direction to the first position in the sequence $\mathbf{x}'$. (purple nodes).

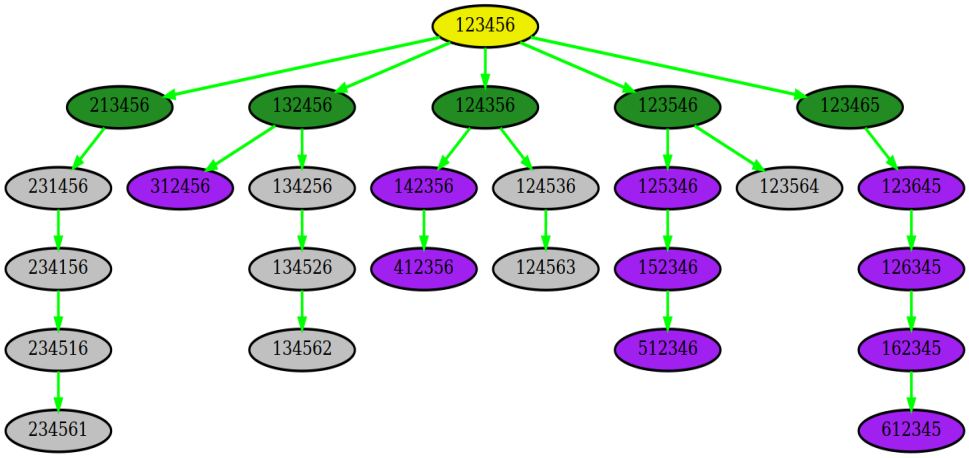


Figure 3.11: Generation of the Insert neighborhood

The total number of initial swaps $n-1$ and the number of further bubblings $n-1$ define the neighborhood size of the Swap-Based Insert operator

$$|N_{swap}^{ins}| = (n-1)^2 \tag{3.51}$$

This allows us to achieve the complexity of LSP, see Algorithm 6 of $\mathcal{O}(n^2)$ representing a substantial reduction compared to the naive approach of $\mathcal{O}(n^3)$.

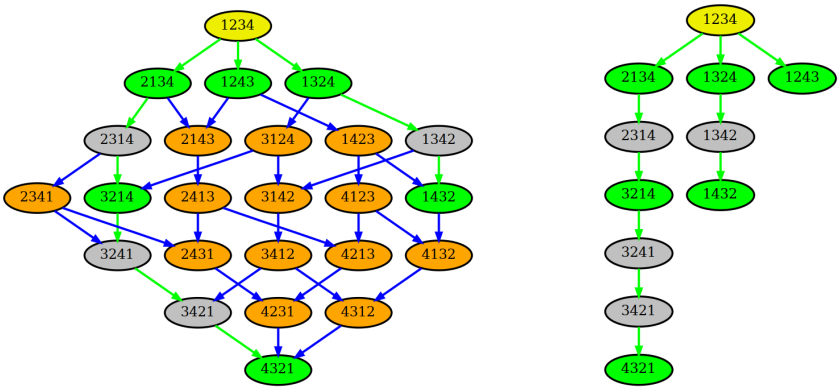
Algorithm 6: LSP Swap-Based Insert

Input: Input path $\mathbf{x} = (x_0, x_1, ...x_n)$
Parameters: Improvement type $\mathcal{IT} \in \{Best, First\}$
Output: Improved path $\mathbf{x}^{impr}$

$\mathbf{x}, \mathcal{X} \leftarrow PrecomputeParameters(\mathbf{x})$
$\mathbf{x}^{impr}, \mathcal{X}^{impr} \leftarrow \mathbf{x}, \mathcal{X}$
for $i = 1$ *to* $N - 1$ **do**
  `// Initial swap`
  $\mathbf{x}^{init}, \mathcal{X}^{init} \leftarrow PerformSwap(i, \mathbf{x}, \mathcal{X})$
  if $cost(\mathcal{X}^{init}) < cost(\mathcal{X}^{impr})$ **then**
    $\mathbf{x}^{impr}, \mathcal{X}^{impr} \leftarrow \mathbf{x}^{init}, \mathcal{X}^{init}$
    if $\mathcal{IT} = First$ **then**
      return $\mathbf{x}^{impr}$
  $\mathbf{x}^{frw}, \mathcal{X}^{frw} \leftarrow \mathbf{x}^{init}, \mathcal{X}^{init}$
  `// Do forward swaps`
  for $j = i + 1$ *to* $N - 1$ **do**
    $\mathbf{x}^{neigh}, \mathcal{X}^{neigh} \leftarrow PerformSwap(j, \mathbf{x}^{frw}, \mathcal{X}^{frw})$
    if $cost(\mathcal{X}^{neigh}) < cost(\mathcal{X}^{impr})$ **then**
      $\mathbf{x}^{impr}, \mathcal{X}^{impr} \leftarrow \mathbf{x}^{neigh}, \mathcal{X}^{neigh}$
      if $\mathcal{IT} = First$ **then**
        return $\mathbf{x}^{impr}$
    $\mathbf{x}^{frw}, \mathcal{X}^{frw} \leftarrow \mathbf{x}^{neigh}, \mathcal{X}^{neigh}$
  $\mathbf{x}^{bckw}, \mathcal{X}^{bckw} \leftarrow \mathbf{x}^{init}, \mathcal{X}^{init}$
  `// Do backward swaps`
  for $j = i - 2$ *to* 1 *with step* -1 **do**
    $\mathbf{x}^{neigh}, \mathcal{X}^{neigh} \leftarrow PerformSwap(j, \mathbf{x}^{bckw}, \mathcal{X}^{bckw})$
    if $cost(\mathcal{X}^{neigh}) < cost(\mathcal{X}^{impr})$ **then**
      $\mathbf{x}^{impr}, \mathcal{X}^{impr} \leftarrow \mathbf{x}^{neigh}, \mathcal{X}^{neigh}$
      if $\mathcal{IT} = First$ **then**
        return $\mathbf{x}^{impr}$
    $\mathbf{x}^{bckw}, \mathcal{X}^{bckw} \leftarrow \mathbf{x}^{neigh}, \mathcal{X}^{neigh}$
return $\mathbf{x}^{impr}$

Swap-Based 2-Opt

Figure 3.12a shows the distribution of 2-Opt permutations in the swap state-space. We can not directly reach the target 2-Opt neighbors (green), meaning we need to perform the intermediate steps and obtain the sequences not part of the 2-Opt neighborhood (grey).



(a) : 2-Opt operator neighborhood in the state-space

(b) : Different depths of the 2-Opt operator

Figure 3.12: Swap-Based 2-Opt

For the initial sequence $\mathbf{x} = (x_0, x_i, ...x_j, ..., x_n)$, a single application of the 2-Opt operator reverses the sub-sequence $(x_{i+1}, ..., x_j)$. For the position i, the maximal length of such sub-sequence is $d(i) = n - i$. To reverse a sub-sequence of length d, we need to perform a total of $\sum_{j=1}^{d(i)} j$ swaps: the first element goes to the end with $d(i)$ swaps, the second with $d(i) - 1$ swaps, and so on. The maximal length of the sub-sequence we can reverse for the whole initial sequence is when $i = 1$, and it defines the maximal depth of the 2-Opt operator obtained with swaps.

$$h_{max}^{2opt} = d(1) = \sum_{j=1}^{n} j = \frac{n(n-1)}{2} \tag{3.52}$$

The intermediate permutations occur when we reverse sub-sequences. The number of neighbors obtained using Swap-Based 2-Opt is defined by the number of reversals needed for each position from 1 to $n - 1$.

$$|N_{swap}^{2opt}| = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i} j = \sum_{j=1}^{n-1} j \cdot (n-j) = \frac{n^3 - n}{6} \tag{3.53}$$

To provide a visual illustration of how the complexity of the Swap-Based 2-Opt operator grows, we draw the graphs of some complexity functions that grow asymptotically close 3.13.

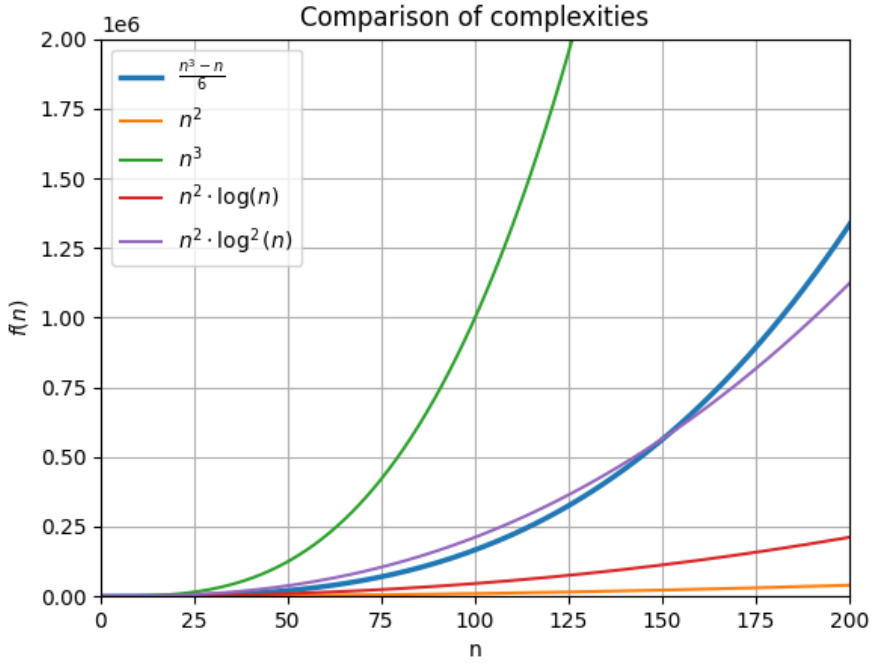


Figure 3.13: Illustration of 2-Opt complexity

2-Opt for the small instances of less than 20 nodes grows asymptotically the same as $n^2 \cdot log(n)$. For the bigger instances of less than 175 nodes, it behaves similarly to $n^2 \cdot log^2(n)$. Although the upper bound asymptotical complexity remains the same $\mathcal{O}(n^3)$ the number of operations required to evaluate all neighbors of the swap-based 2-Opt is less than in basic 2-Opt $n \cdot |N^{2opt}_{swap}| < |N^{\mathbf{x}}_{2opt}|$
The proposed LSP is described in Algorithm 7. We also introduce an additional parameter, the 2-Opt depth, which can range from 1 to h^{2opt}_{max}. This parameter reduces the maximal path length in the state space. This parameter is set to h^{2opt}_{max} by default. From Figure 3.12, we observe that the intermediate swaps also increase as the depth of 2-Opt increases. In other words, obtaining a 2-Opt neighbor distant from the initial sequence requires more additional swaps. Further, we conducted the experiments to identify the optimal depth of 2-Opt that reduces the number of swap operations.

Algorithm 7: Swap-Based 2-Opt

Input: Input path $\mathbf{x} = (x_0, x_1, ...x_n)$
Parameters: Maximal depth 2-Opt depth h_{max}
,improvement type $\mathcal{IT} \in \{Best, First\}$
Output: Improved path $\mathbf{x}^{impr}$

$\mathbf{x}, \mathcal{X} \leftarrow PrecomputeParameters(\mathbf{x})$
$\mathbf{x}^{impr}, \mathcal{X}^{impr} \leftarrow \mathbf{x}, \mathcal{X}$
`// Sub-sequence to reverse`
for $i = 1$ *to* $N - 1$ **do**
  $\mathbf{x}^{cur}, \mathcal{X}^{cur} \leftarrow \mathbf{x}, \mathcal{X}$
  $h_{cur} \leftarrow 0$
  `// Reverse distance`
  for $d = i + 1$ *to* N **do**
    `// Backward swaps`
    for $j = d - 1$ *to* $j \geq i$ *with step* -1 **do**
      if $h_{cur} > h_{max}$ **then**
        break
      $\mathbf{x}^{neigh}, \mathcal{X}^{neigh} \leftarrow PerformSwap(j, \mathbf{x}^{cur}, \mathcal{X}^{cur})$
      if $cost(\mathcal{X}^{neigh}) < cost(\mathcal{X}^{impr})$ **then**
        $\mathbf{x}^{impr}, \mathcal{X}^{impr} \leftarrow \mathbf{x}^{neigh}, \mathcal{X}^{neigh}$
        if $\mathcal{IT} = First$ **then**
          return $\mathbf{x}^{impr}$
      $\mathbf{x}^{cur}, \mathcal{X}^{cur} \leftarrow \mathbf{x}^{neigh}, \mathcal{X}^{neigh}$
      $h_{cur} \leftarrow h_{cur} + 1$
return $\mathbf{x}^{impr}$

Balas-Simonetti Neighbourhood

The following is an improving heuristic that doesn't relate to swap-based procedures and was inspired by the Balas-Simonetti neighborhood first proposed for solving the TSP with precedence constraints[34]. According to [35], for a given integer $k \geq 2$, the Balas-Simonetti (BS) neighborhood $N^{\mathbf{x}}_{BSk}$ allows the restricted permutation of the vertices relative to a given path, i.e., it permits the movement of any vertex up to k positions forward or backward, ensuring that precedence is maintained. More precisely, for a given tour $\mathbf{x} = (x_0, x_1, \ldots, x_n)$, another tour $\mathbf{x}' = (x_0, x_{\pi(1)}, \ldots, x_{\pi(n)})$, defined by permutation π, is in the neighborhood $N^{\mathbf{x}}_{BSk}$ if

$$i + k \leq j \implies \pi(i) < \pi(j), \forall i, j \in \{1, ..., n\} \tag{3.54}$$

The larger value of k offers more flexibility, allowing the neighborhoods to be nested.

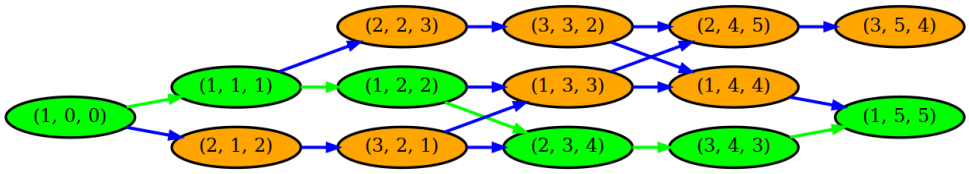


Figure 3.14: BS2 neighbourhood for the initial path $(x_0, x_1, x_2, x_3, x_4, x_5)$

Thus far, the best-improved neighbor for Insert and 2-Opt has been achieved through sequential evaluation of all neighbors. However, for the Balas-Simonetti neighborhood, this approach is not ineffective because the neighborhood size is exponential, $\Omega(\frac{k-1}{e})^{n-1}$ [36]. Consequently, the optimization approach is used, which is based on finding the shortest path in the auxiliary graph G_k.
Figure 3.14 illustrates the auxiliary graph G_k for $k = 2$, where each node is defined by the triple $(l, i, \pi(i))$: layer, position, and permutation of the position. For example, traversing along the green nodes results in the valid BS2 permutation $(x_0, x_1, x_2, x_4, x_3, x_5)$. Building such a graph takes $\mathcal{O}(n\,k(k+1)2^{k-2})$ and finding the shortest path in the graph for the fixed k is linear to the tour length.
The implementation of the auxiliary graph for $k = 2, 3, 4$ is provided in the article [36]. Based on this paper, we construct three auxiliary graphs for $k = 2, 3, 4$ respectively and consider the construction process independent of the respective local search procedure BSk. In LSP of BSk, we call the Dijkstra algorithm that uses the criterion from Equation 3.1 to take the next minimum vertex. The found shortest path is considered to be the best-improved neighbor. The procedure BSk is part of the iterative local search.

3.3 Metaheuristics

The drawback of improving heuristics is that they often get stuck in local minima, failing to reach the global optimal solution. Metaheuristics are typically proposed to address this limitation. One of the most popular approaches is Variable Neighborhood Search (VNS), which focuses on improving a single solution rather than generating multiple ones and selecting the best among them. The core concept of VNS lies in efficiently exploring different neighborhood structures to escape local minima and aim for global optimality. VNS recognizes that local minima in one neighborhood may not hold in others, yet they tend to be relatively close across various neighborhoods [31]. VNS effectively balances exploration and exploitation to converge toward good-quality solutions by iteratively refining solutions and prioritizing global optimization.

3.3.1 Sequential Variable Neighbourhood Descent

Variable Neighborhood Descent (VND) is a version of VNS explores several neighborhood structures deterministically. Its success is based on the fact that different neighborhood structures usually contain distinct local minima [37]. Therefore, systematically transitioning between various neighborhood structures helps prevent entrapment in local optima. There are various versions of VND, but we will focus on a basic sequential approach (SVND) denoted in Algorithm 8, which can be considered a solid starting point for further extension. In this method, several neighborhood structures are ordered in a list and examined one after another according to the established order.
Let $\mathcal{N} = \{\Phi_1, ..., \Phi_{l_{max}}\}$be a list of local search procedures defining the operators' neighborhoods. Then, the neighborhoods are explored iteratively, one after another, and the order of their examination is determined by the order of operators in the list. To facilitate this, we introduce the Neighbourhood Change procedure, outlined in Algorithm 9. Its purpose is to guide the variable neighborhood search by determining the next neighborhood to explore and deciding whether to accept a solution as a new incumbent solution. When an improvement occurs in the incumbent solution within a neighborhood structure, the SVND procedure resumes searching in the first neighborhood structure (according to the defined order) of the new incumbent solution. The process terminates if the incumbent solution cannot be improved concerning any of the maximum neighborhood structures.

Algorithm 8: Sequential Variable Neighbourhood Search (*SVND*)

Input: Initial solution $\mathbf{x}^{init}$
List of local search procedures $\mathcal{N} = \{\Phi_1, ..., \Phi_{l_{max}}\}$
Parameters: Time limit t_{max}
Output: The most optimized path $\mathbf{x}^{most}$

$\mathbf{x}^{most} \leftarrow \mathbf{x}^{init}$
$l \leftarrow 1$
$t \leftarrow$ CpuTime()
repeat
  $\mathbf{x}^{impr} \leftarrow$ LocalSearch($\mathbf{x}^{most}, \Phi_l$)
  $l, \mathbf{x}^{most} \leftarrow$ NeighbourhoodChange($\mathbf{x}^{impr}, \mathbf{x}^{most}, l$)
  $t \leftarrow t -$ CpuTime()
until $(l < l_{max})$ *and* $(t < t_{max})$

Algorithm 9: Neighbourhood change (*NeighbourhoodChange*)

Input: Improved solution within the current neighborhood $\mathbf{x}^{impr}$.
The most optimized path $\mathbf{x}^{most}$.
Current neighborhood index l
Output: Next neighbourhood index $k$to explore .
The most optimized path $\mathbf{x}^{most}$

if $\mathbf{x}^{most} \neq \mathbf{x}^{impr}$ **then**
  $k \leftarrow 1$
  $\mathbf{x}^{most} \leftarrow \mathbf{x}^{impr}$
else
  $k \leftarrow l + 1$
return k, $\mathbf{x}^{most}$

3.3.2 General Variable Neighbourhood Search

For a more sophisticated approach capable of attaining the global minimum, we utilized the General Variable Neighborhood Search (GVNS) represented in Algorithm 10 due to its reported effectiveness in various applications [38], [39]. In this metaheuristics, the iterated local search is substituted with SVND, and stochastic perturbations are introduced to enable the discovery of global minimum solutions.

Algorithm 10: General Variable Neighbourhood Search ($GVNS$)

Input: Initial solution $\mathbf{x}^{init}$.
List of local search procedures $\mathcal{N} = \{\Phi_1, ..., \Phi_{l_{max}}\}$. Maximal perturbation strength k_{max}
Parameters: Time limit t_{max}.
Output: The most optimized path $\mathbf{x}^{most}$

$\mathbf{x}^{most} \leftarrow \mathbf{x}^{init}$
$k \leftarrow 1$
$t \leftarrow$ CpuTime()
repeat
  $\mathbf{x}^{pert} \leftarrow Perturbate(\mathbf{x}^{most}, k)$
  $\mathbf{x}^{impr} \leftarrow SVND(\mathbf{x}^{pert}, \mathcal{N})$
  $k, \mathbf{x}^{most} \leftarrow NeighbourhoodChange(\mathbf{x}^{impr}, \mathbf{x}^{most}, k)$
  $t \leftarrow t -$ CpuTime()
until $(k < k_{max})$ *and* $(t < t_{max})$

Perturbation

The purpose of perturbation is to diversify the search and prevent the local search algorithm from getting stuck in the local optimum. By introducing perturbation, we aim to explore various regions of the solution space. While this may lead to obtaining a worse solution initially, the subsequent local search phase enables exploration of the neighborhood around this solution, potentially leading to an improved outcome.

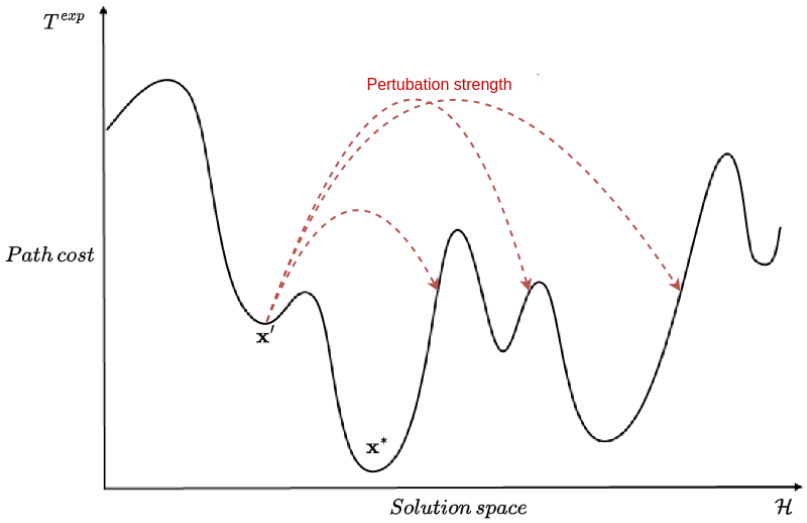


Figure 3.15: Perturbation strength

The classical perturbation operator is the double-bridge [40], a variation of the 4-opt operator. While the classical 4-opt operator removes four edges from the current tour and has many ways of adding new edges, the double-bridge employs a unique way of tour construction. Let's assume we have a tour $\mathbf{x} = (x_0, x_1, ..., x_n)$. Then, we select the 4 different breakpoints $b_1 < b_2 < b_3 < b_4$ and remove the edges between b_i and its successor, denoted as $scs(b_i)$, creating the segments $\mathbf{s} = (A, B, C, D, E)$ where each segment is the nonempty subset of vertices. The classic double-bridge is defined for the cyclic Hamiltonian path, so the segment E, A is considered a single segment with the vertex x_0. But in our problem, the path is not cyclic, so we assume that the edge x_n, x_0 is absent. The double bridge connects the segments $\mathbf{s}$ in such a way that the orientation of the path remains and creates a new path $\mathbf{x}^{pert} = (A, D, C, B, E)$, which illustrated in Figure 3.16.

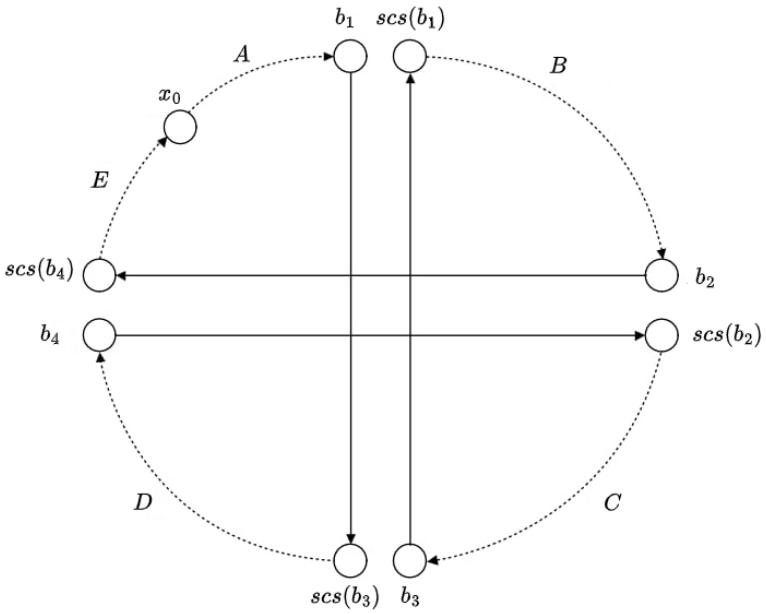


Figure 3.16: Double-bridge

The double bridge is advantageous [41] because it removes just four arcs from a tour without employing reverse operations, implying that the new path is close to the original one. Additionally, its modifications are difficult for other operators to reverse, minimizing the likelihood of reverting to the previously perturbed local minimum.

Algorithm 11: k-Double-bridge (*k-DoubleBridge*)

Input: Sequence of vertices $\mathbf{x} = (x_0, ...x_n)$
Perturbation strength k
Output: k-perturbated sequence of vertices $\mathbf{x}^{pert} = (x_0, ...x_n)$

$\mathbf{x}^{pert} \leftarrow \mathbf{x}$
for $m = 0$ *to* k **do**
  Select randomly b_1, b_2, b_3, b_4 such that $0 \leq b_1 < b_2 < b_3 < b_4 \leq n$
  arcs $\leftarrow \{\{0, b_1\}, \{b_1 + 1, b_2\}, \{b_2 + 1, b_3\}, \{b_3 + 1, b_4\}, \{b_4 + 1, n\}\}$
  $\mathbf{x}^{orig} \leftarrow \mathbf{x}^{pert}$
  $\mathbf{x}^{orig} \leftarrow ()$
  `// Append the path segments`
  for *arc in arcs* **do**
    $\mathbf{x}^{pert} \leftarrow \mathbf{x}^{pert} \oplus \mathbf{x}^{orig}[arc]$
return $\mathbf{x}^{pert}$

We propose the extension based on the k-double bridge approach, considered one of the most effective solution mutation operators [33] and detailed in Algorithm 11. It involves iteratively applying the double bridge operation on the tour formed by the previous application of the double bridge. The value of k determines the strength of perturbation. Increasing k leads to more iterations of the double bridge operation, causing the path to deviate further from the original tour, illustrated in Figure 3.15.

Chapter 4

Experimental evaluation

This chapter provides the experimental assessment of the proposed methods. The first section describes the setup used for our experiments. The second section identifies the best local search procedures for building the metaheuristics described. The final section compares the proposed metaheuristics with the reference method.

4.1 Setup

4.1.1 Software & Hardware

The primary codebase is written in C++17 and undergoes thorough testing within the *Visis-Planner* environment [1]. This environment utilizes 2D graphics libraries *Clipper* [2], *Triangle* [3], and *Robust geometric predicates* [4]. To conduct experiments, we employed Python scripts to execute the C++ main program with different input arguments and gathered data about the performance of our algorithms. The key metrics include time, measured using the high-resolution clock integrated into C++, and cost, computed according to the equation (2.15). The first two sections, aiming to construct our metaheuristics, were measured on the Lenovo Legion 5 running on Ubuntu 23.04 with hardware AMD Ryzen 7, 3.8 GHz, and 16 GB of RAM. The last section, aiming to compare the proposed methods with the reference one, was measured on the laboratory PC running on Debian 5.0 with CPU Intel Core i7-7700, 3.6 GHz, and 32 GB of RAM.

4.1.2 Maps overview

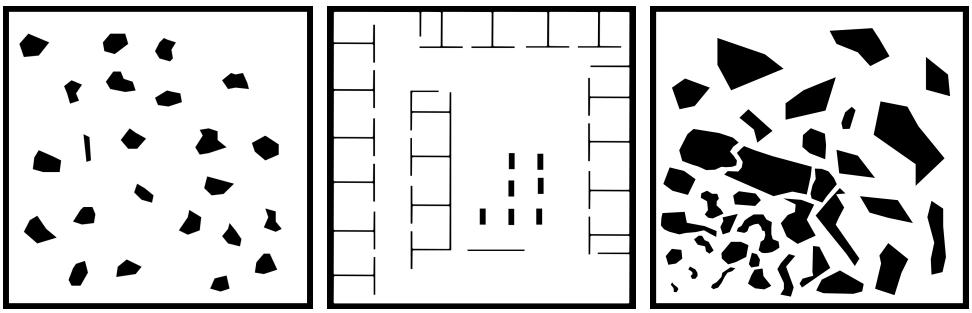
We will use three maps illustrated in Figure 4.1 for our experiments. We chose them because they are among the most challenging for the weight approximation for the reference method with static weights [12].

[1] `https://gitlab.ciirc.cvut.cz/mission-planning/visis-planner`. The planner was developed by the team Intelligent Mobile Robotics, CIIRC, CTU in Prague.

[2] `https://angusj.com/clipper2/Docs/Overview.htm`, polygon clipping operations, and related geometric algorithms.

[3] `https://www.cs.cmu.edu/~quake/triangle.html`, triangular mesh generation.

[4] `https://github.com/dengwirda/robust-predicate`, for geometry primitives.



(a) : *potholes*, [20 x 20] **(b) :** *jari-huge*, [21 x 23] **(c) :** *large*, [40 x 40]

Figure 4.1: Maps used for measurements and their sizes

The generation of respective visibility regions and their distance relationships are based on the dual sampling described in [29]. In our experiment, we will work with the number of vertices in the graph, which defines the problem size. However, we cannot directly generate a specific number of vertices for the map because the number of vertices is inversely dependent on the chosen visibility radius, which can potentially be infinite. For illustration, we generated the number of vertices for the visibility radius ranging from 2 to 12 with a step size of 0.5 for all maps, as provided in Figure 4.2. Beyond a certain visibility radius, we cannot generate fewer vertices, establishing the minimal number of nodes we can use for our maps: 15 for *potholes*, 35 for *jari-huge*, and 50 for *large*, respectively. We should mention that increasing the precision in the visibility radius can lead to rounding errors in region generations. In most of our experiments, we will assume that the visibility radius has been predetermined and focus solely on the number of nodes within this radius.

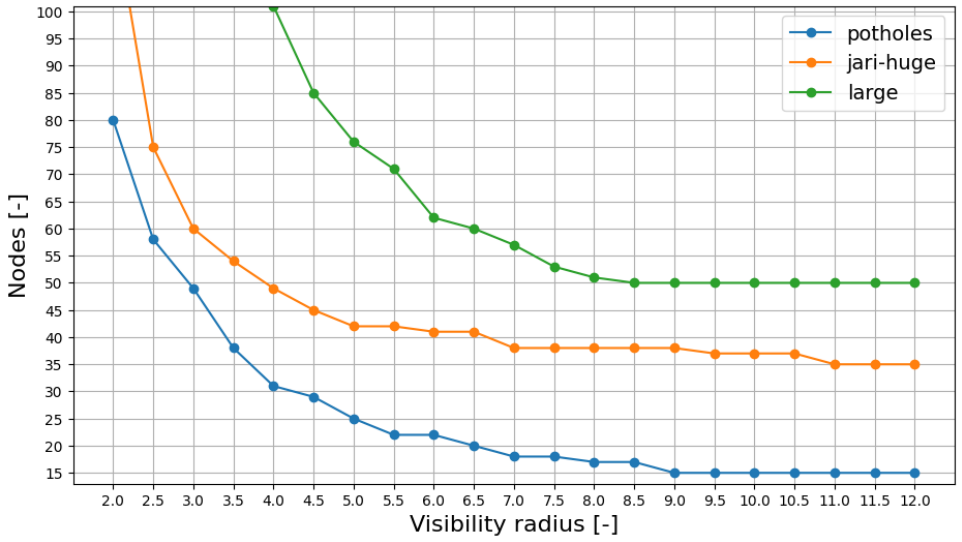


Figure 4.2: Nodes dependency on visibility radius

4.2 Local Search Procedures

These experiments aim to select the most effective local search procedures to develop our SVND and GVNS metaheuristics further. We have chosen a single map *potholes* and varied the number of nodes from 15 to 50 with a step size 5. Also, we tested all our local search procedures on the same initial solutions, which were generated using a randomized greedy algorithm with the criterion from Equation (3.1) and probabilities $p_1 = 0.66, p_2 = 0.22, p_3 = 0.12$. For every node count, we created 20 initial solutions. This method comprehensively evaluates the algorithm's performance across a spectrum of input scenarios.

4.2.1 Precomputation complexity

Our local search requires global precomputation, which is not factored into the time evaluation of local search procedures. We consider the intersection of polygon areas and the construction of Balas-Simonetti auxiliary graphs as precomputation processes that need to be executed once for subsequent applications of their respective local search procedures. Figure 4.3 illustrates that the precomputation process is swift, typically measured in tens of milliseconds for small problem sizes. Additionally, we observe minimal variation in the mean construction time of Balas-Simonetti graphs across different values of $k = 2, 3, 4$. It's important to note that polygon operations may not remain constant in real-world scenarios. The complexity of polygons can increase with the number of nodes in the graph, impacting the required time.

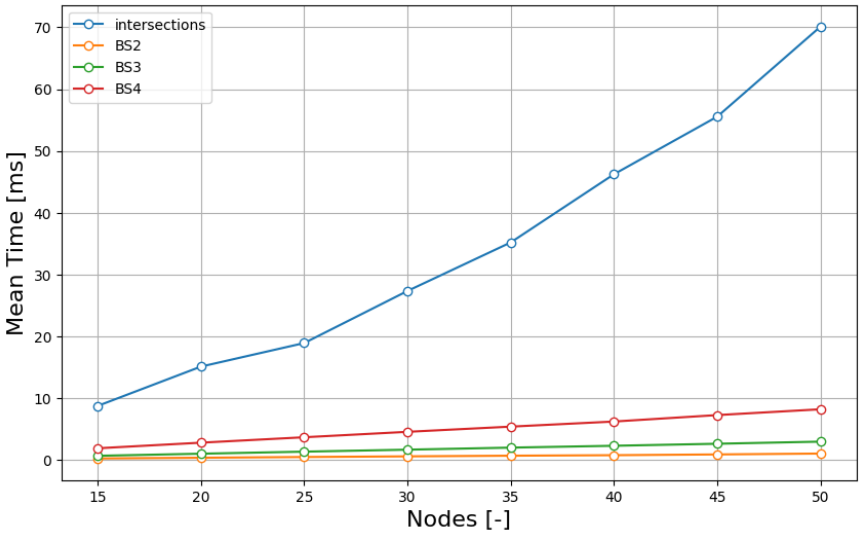
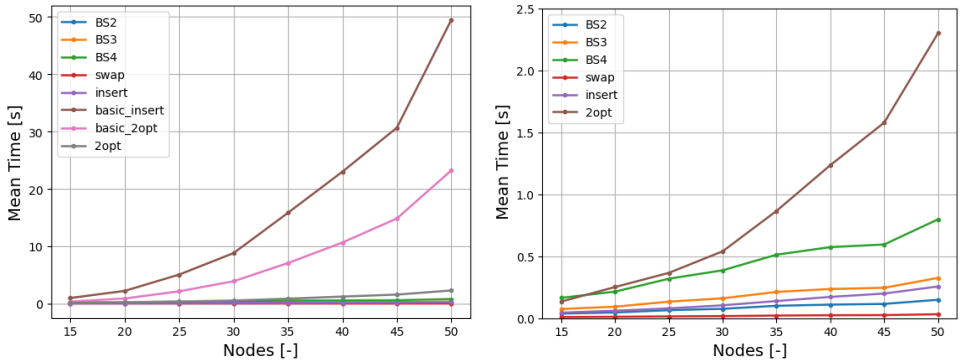


Figure 4.3: Real precomputation time

4.2.2 All operators time

Now, we can focus on comparing all proposed operators. For this experiment, we will execute one local search procedure with the best improvement with all our operators and measure the mean time achieved by them



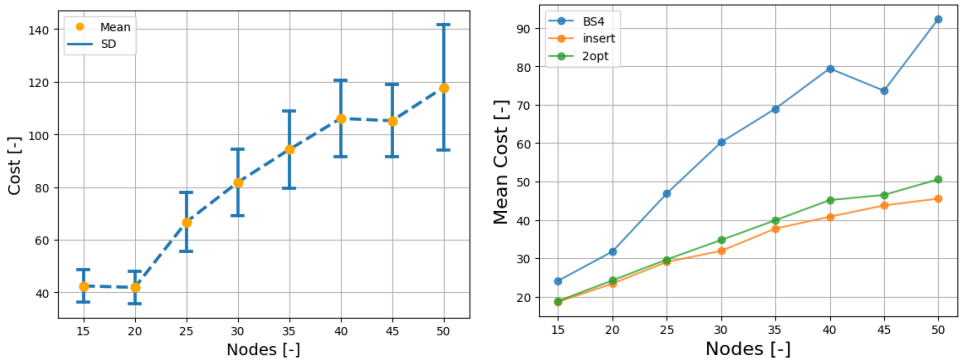
(a) : All operators **(b) :** Fast operators

Figure 4.4: Single iteration of best improvement

As seen in Figure, 4.4, the basic Insert and 2-Opt operators (*basic_insert*, *basic_2-opt*) exhibit a significant slowdown compared to swap-based counterparts (*insert*, *2opt*), validating the soundness of our selected approach. We choose BS4, swap-based Insert, and 2-Opt among the remaining operators for further application. We select BS4 because it can potentially lead to more promising solutions due to its larger neighborhood, and we don't observe a significant slowdown for BS4 compared to BS2 and BS3. The swap operator is not considered separately because it's already part of the insert operator, and the insert itself covers the neighborhood of the swap.

4.2.3 Fast operators cost

Next, we run the same experiment with a fully iterated local search with the best improvement until no improved solution occurs, and we compute the mean cost for a single node instance. Figure 4.5 shows that our improving heuristics provide significant cost improvement compared to the initial solution.



(a) : Initial solution distribution **(b) :** Iterated local search results

Figure 4.5: Fast operators cost comparison

We can consolidate the findings from Figures 4.5 and 4.4 into Table 4.1, where the numbers 1, 2, and 3 represent the performance ranking of each operator in terms of both cost and time. Specifically, 1 indicates the best performance, while 3 signifies the worst.

Operator	Cost	Time
Insert	1	1
2-Opt	2	3
BS4	3	2

Table 4.1: Operators ranking

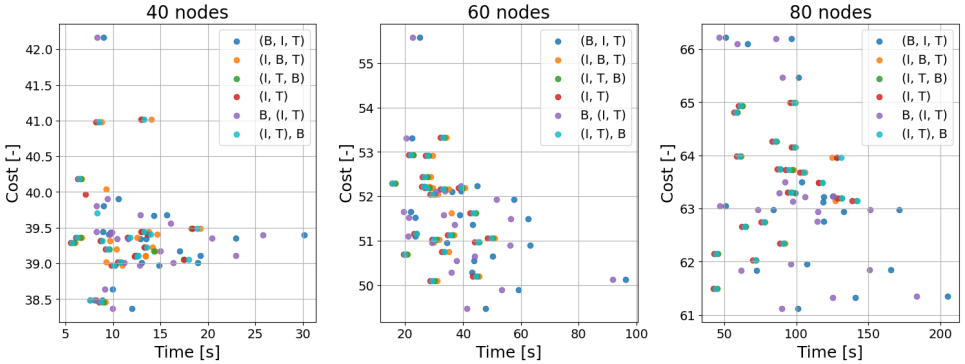
4.3 SVND, GVNS construction

This section focuses on building the SVND and GVNS metaheuristics based on the selected fast local search procedures. SVND is a subpart of GVNS, so properly constructing SVND allows us to obtain the desired behavior of GVNS. For these experiments, we considered two maps, *potholes* and *jari-huge*, with larger problem sizes of 40, 60, and 80 nodes, respectively. In the first three experiments, we used 20 initial solutions per node instance generated using the same randomized greedy approach as in the previous section.

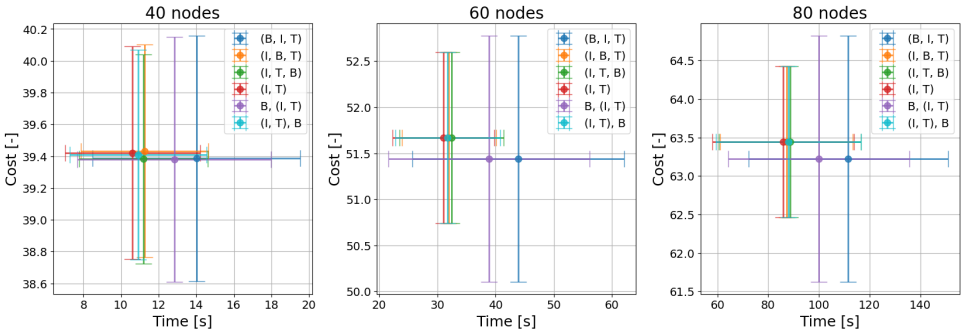
4.3.1 BS4 impact

The first step is to identify a list of local search procedures that can be utilized for our SVND. The basic are the Insert and 2-Opt operators and the list (Insert, 2-Opt), where the Insert operator precedes 2-Opt because it yields the fastest and most optimized ILS solutions.
In this experiment, we aim to assess the impact of the BS4 operator within the context of our SVND structure. We denote Insert operator as I, 2-Opt as

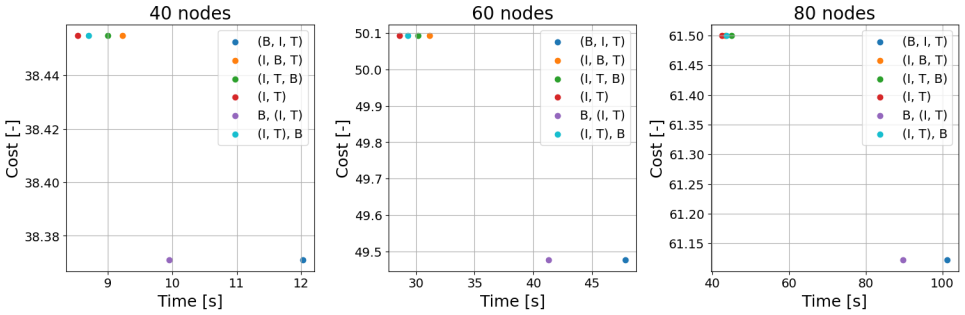
T, and BS4 as B. We will compare our basic SVND structure, consisting of the operators Insert and 2-Opt (I, T), with five variations incorporating BS4: (B, I, T); (I, B, T); (I, T, B); (I, T), B; B, (I, T). The first three variations represent different placements of BS4 within the SVND operators list. In the fourth variation, we execute BS4 once using full ILS on the Best improvement criterion, followed by the basic (Insert, 2-Opt) SVND. In the fifth variation, the basic (Insert, 2-Opt) SVND is initially applied, and then ILS is performed with BS4. Both Insert and 2-Opt operators are evaluated using the best improvement criterion on the map *potholes*.



(a) : All measured points



(b) : Mean, standard deviation of cost and time



(c) : Points with minimal cost

Figure 4.6: Impact of BS4 for the different SVND structures

We found that the BS4 operator improves cost only when positioned before our Insert and 2-Opt operators. In other arrangements, it didn't notably affect the outcome and sometimes even caused a slight slowdown; see Figures 4.6b, 4.6c. Placing BS4 at the beginning resulted in cost improvement and a noticeable slowdown, especially with 80 nodes.
Initially, BS4 can find better solutions, but convergence takes more time when followed by the Insert operator. This behavior can be attributed to the nature of BS4 neighborhoods, as it is constrained by the specific positions on which the vertex can be placed. In contrast, Insert allows the placement of any vertex in any position. The value of k for Balas-Simonetti increases the variability of vertices placements. However, this experiment for BS4 showed a significant slowdown for only marginal improvement. Hence, we opted to stick with the basic SVND structure of Insert and 2-Opt.

4.3.2 2-Opt Depth

The next step is to limit the depth of 2-Opt, as excessive neighborhood exploration may not always yield improvements; it simplifies the routine without affecting the quality of the solutions [42]. To achieve this, we adjust the maximum depth percent of 2-Opt, meaning that if 2-Opt exceeds this depth, it stops searching further. Figure 4.7 shows how the number of explored neighbors evolves with the increasing depth percentage for a single local search procedure of 2-Opt with the Best improvement. Both graphs are identical, confirming the Equation (3.52) that the number of visited neighbors in 2-Opt stays consistent for the same number of input nodes and the same depth.

For this experiment, we used two maps, *potholes* and *jari-huge*, to measure SVND with different depth percentages using the Best Improvement strategy for our operators. Setting the depth percentage in SVND to 0 means we only apply the Insert operator, with no influence from the 2-Opt operator.

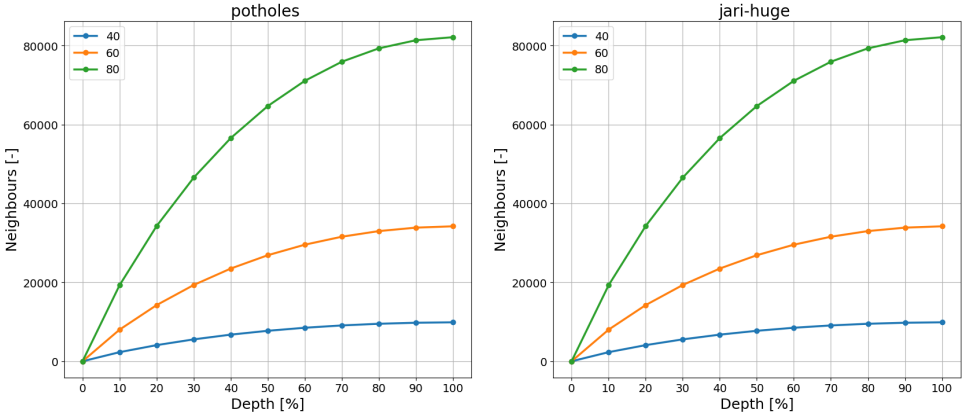
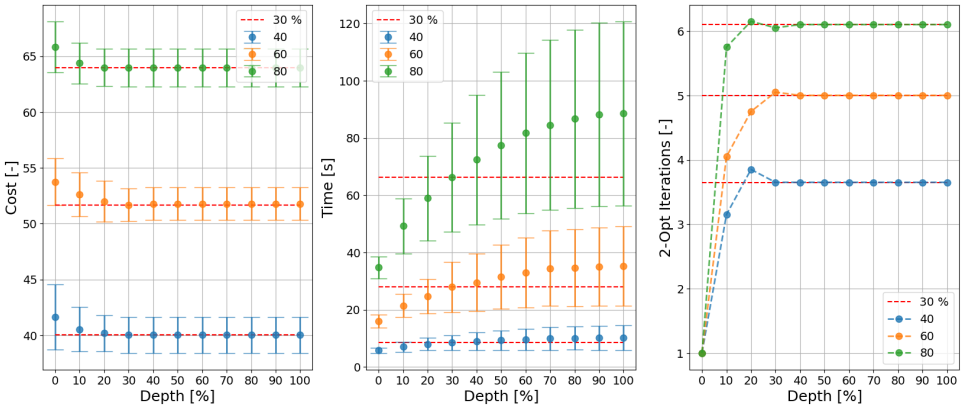


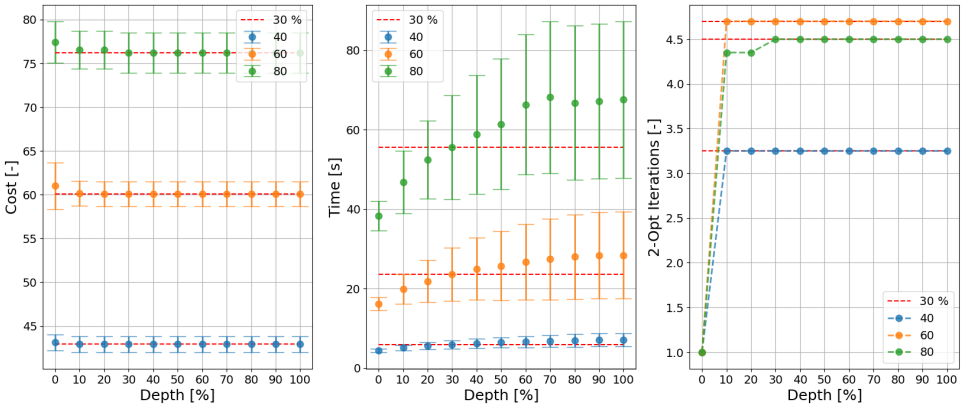
Figure 4.7: Exploration of neighbors by 2-Opt relative to depth percent

Based on our measured data in Figure 4.8, we determined that the optimal depth percentage for 2-Opt is 30 percent of the maximal depth. We observe no significant cost improvements beyond this depth for SVND. Limiting the

2-Opt depth doesn't reduce the number of iterations executed by 2-Opt during SVND but replaces those extra 2-Opt iterations with the less time-consuming Insert operator. This substitution maintains solution quality while improving time performance.



(a) : *potholes*

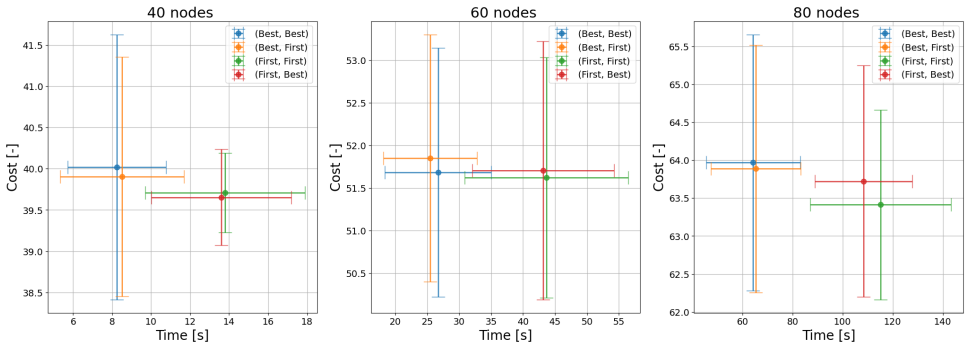


(b) : *jari-huge*

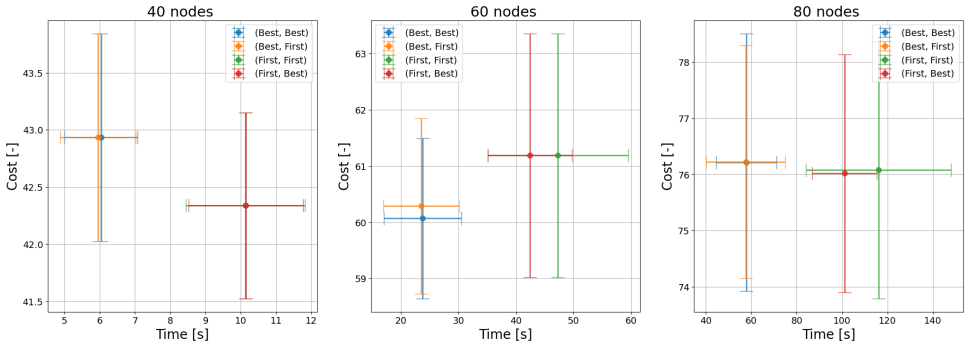
Figure 4.8: Measurements of SVND for different 2-Opt depths

4.3.3 Best-first improvement

We're now examining the improvement criteria we can apply to our local search procedures within SVND [32]. These procedures can operate under either the Best or First improvement strategy. To explore this, we're considering four pairs of improvements for specific local search operators: $(Best, Best)$, $(First, First)$, $(Best, First)$, $(First, Best)$. The first improvement strategy in each pair pertains to the Insert operator, while the second relates to 2-Opt. We'll use these four pairs to run our SVND on the same input instance as in the previous experiment.



(a) : *potholes*



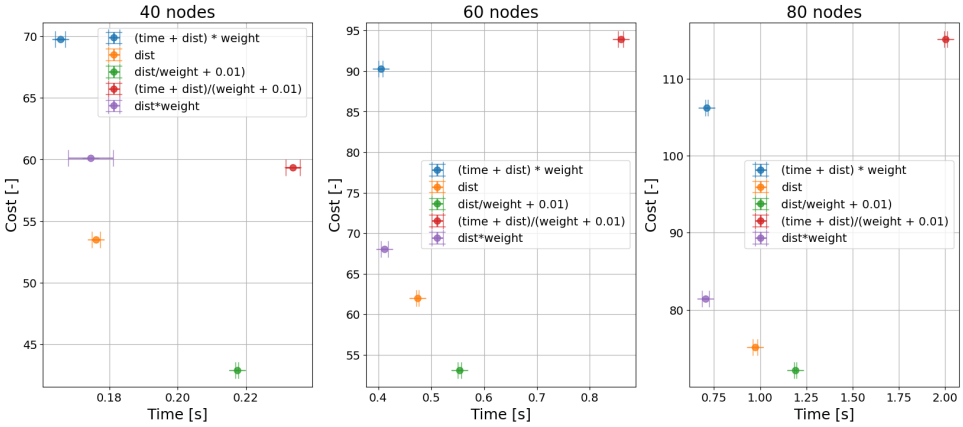
(b) : *jari-huge*

Figure 4.9: Best-first improvements comparison

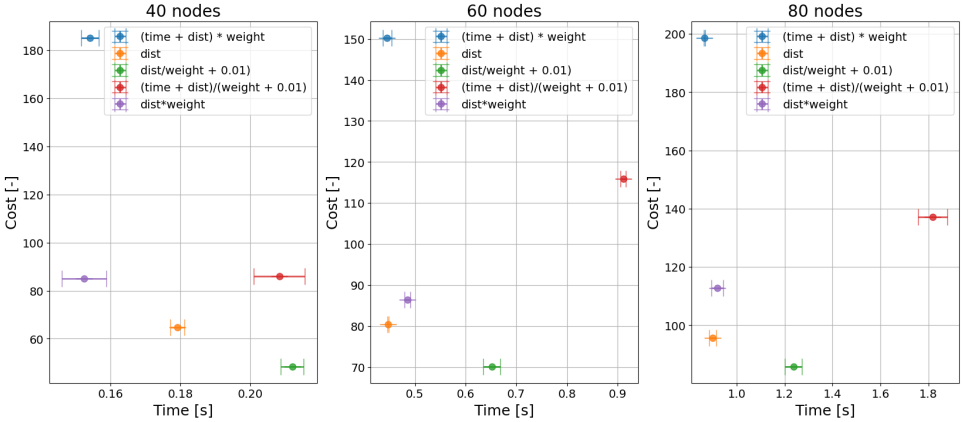
Figure 4.9 demonstrates a noticeable slowdown in processing time when opting for the First improvement strategy for the Insert operator without yielding significant cost improvements. The most time-optimal criterion for Insert is the Best improvement strategy, which promotes faster convergence. With the depth reduced to 30 percent in the previous experiment, we notice that the $Best$ and $First$ improvements for 2-Opt exhibit similar behavior. Hence, we can choose between $(Best, First)$ and $(Best, Best)$, as they provide similar cost and time performance. Subsequently, we decided to proceed with $(Best, Best)$ for further experiments.

4.3.4 SVND initial solution

In our earlier experiments, the initial solution was generated using the greedy criterion from Equation (3.1). We aim to determine the criterion we will use to generate both greedy and randomized initial solutions for SVND in the final experiments. For that, we conducted five runs of SVND using all our greedy criteria and assessed the overall cost and time performance.



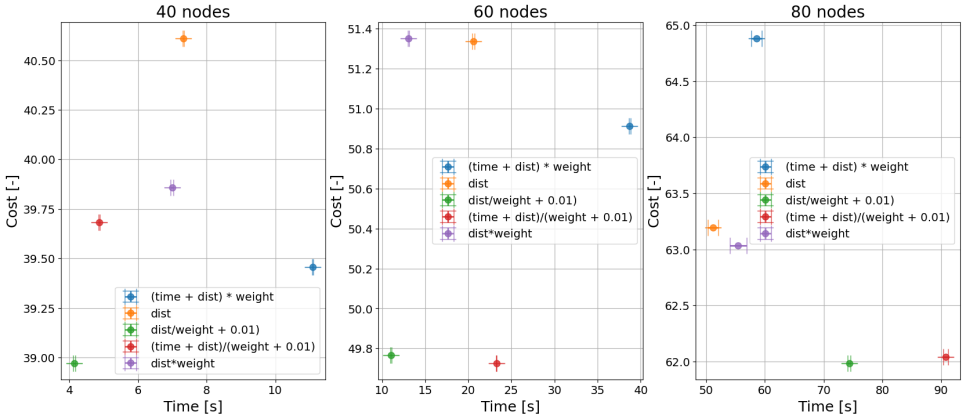
(a) : *potholes*



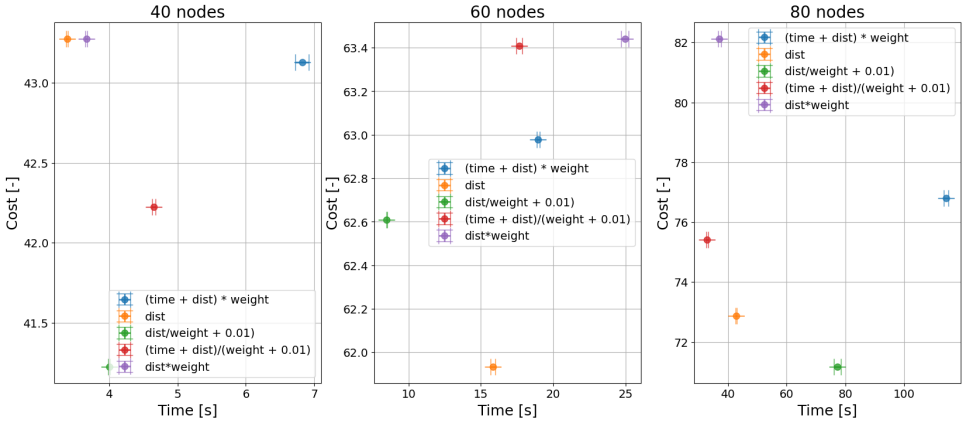
(b) : *jari-huge*

Figure 4.10: Initial solution cost/time distribution for various criteria

The fact that the initial cost remains consistent across all runs validates the correctness of our greedy criteria, as there was no deviation observed in Figure 4.10. After applying SVND, the greedy initial solution based on the criterion from Equation (3.3) consistently maintained its position among the best or at the top in cost, denoted in Figure 4.11. Additionally, SVND demonstrated efficient convergence with this criterion, requiring minimal time. Consequently, we have opted to continue utilizing the criterion from Equation (3.3) for our greedy and randomized greedy constructive heuristics.



(a) : potholes

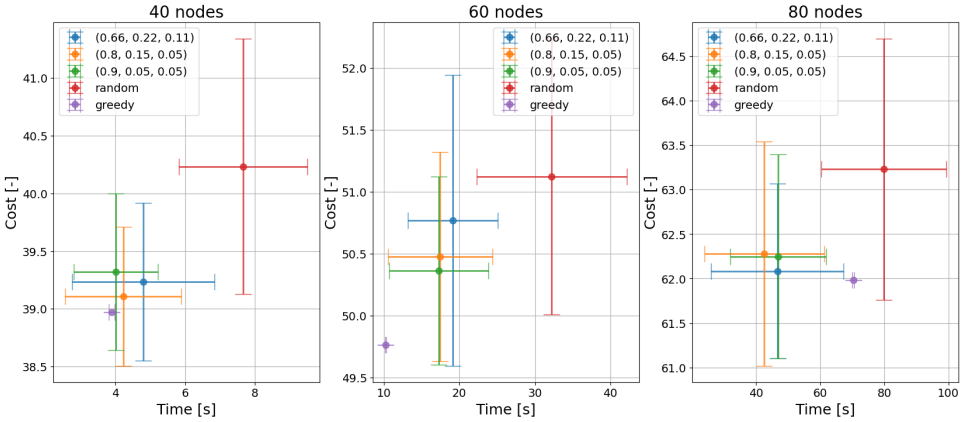


(b) : jari-huge

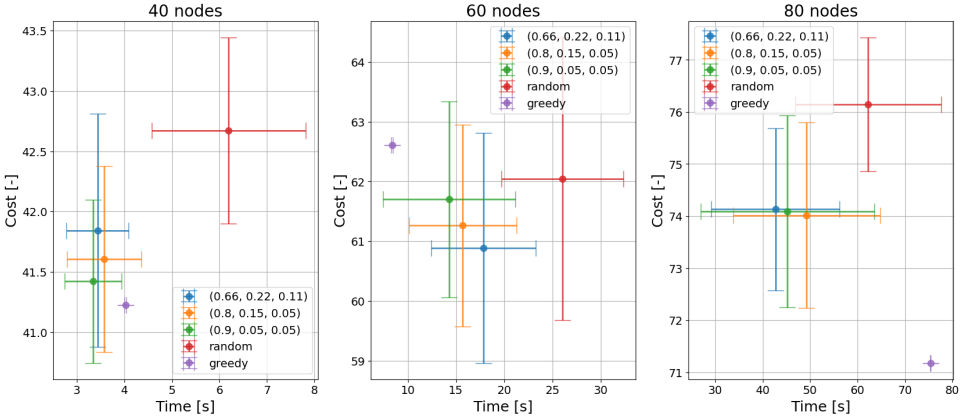
Figure 4.11: Cost/time distribution after applying SVND with various greedy criteria

Next, we attempted to generate randomized greedy solutions based on the selected greedy criterion. To achieve this, we created 30 randomized greedy initial solutions using three different probability sets: $(0.66, 0.22, 0.12)$, $(0.8, 0.15, 0.05)$, and $(0.9, 0.05, 0.05)$. Also, we generated 30 fully random initial solutions and the deterministic greedy one. Then, we compared all the solution types after the application of SVND.

From Figure 4.12, the randomized greedy approach, based on the selected criterion, enables us to obtain better solutions compared to the greedy one potentially. Furthermore, the solutions obtained through randomized greedy exhibit less variability than those obtained through full randomization. However, the choice of probability set is heavily influenced by the map and the number of nodes. After careful consideration, we have decided to use the probability set $(0.8, 0.15, 0.05)$ to randomize our greedy solutions. This set strikes a balance between providing less variability compared to the first set and not overly reducing it compared to the third set



(a) : *potholes*

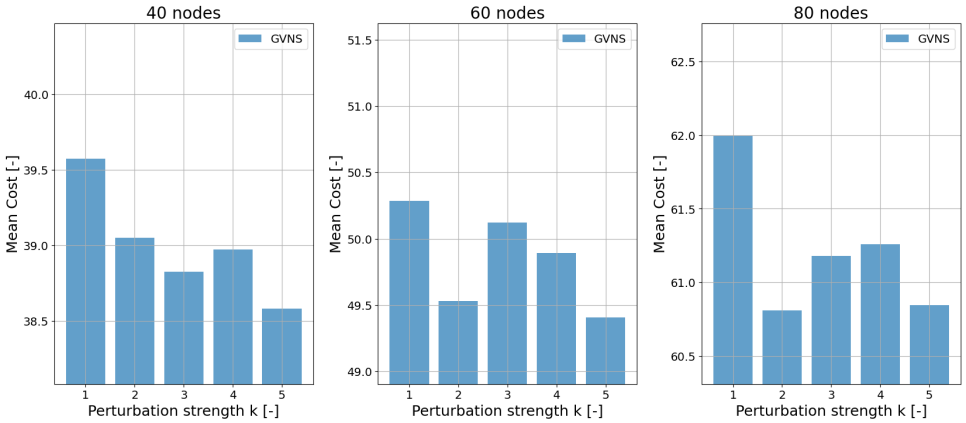


(b) : *jari-huge*

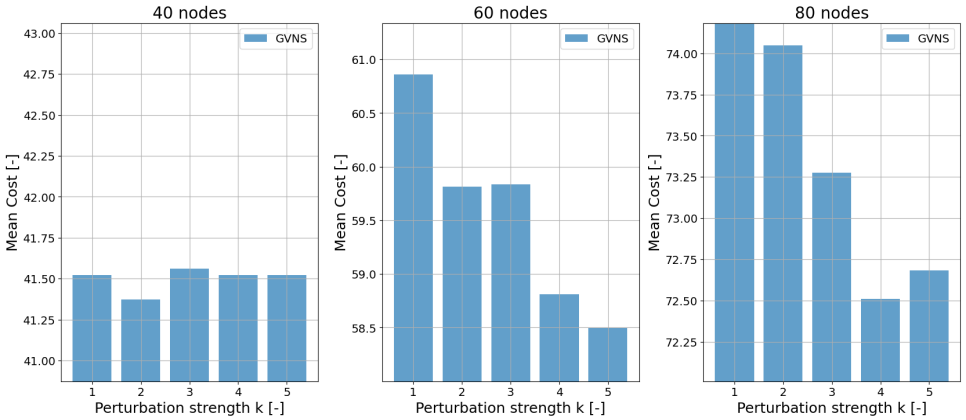
Figure 4.12: Cost/time distribution for different probabilities of randomized greedy after application of SVND

4.3.5 GVNS perturbation

For the initial solution of our GVNS, we consider the deterministic greedy one; the randomization is achieved using perturbation, potentially leading to better solutions. To achieve this, we must select the perturbation strength for the proposed k-double bridge. We have decided to maintain a perturbation strength of 5 and evaluate whether this choice results in better solutions. To assess this, we executed our GVNS algorithm 10 times on each node instance with different perturbation strengths.



(a) : *potholes*



(b) : *jari-huge*

Figure 4.13: Perturbation evaluation

The results in Figure 4.13 confirm that the mean cost decreases with the growing perturbation strength. Therefore, we've decided to continue with a perturbation strength of 5 and prioritize the cost performance of GVNS over time.

4.4 Reference method comparison

In the final experiments, we compare the proposed metaheuristics with the reference method on all three maps. For each input instance, SVND will be executed on three types of initial solutions: full random (30 times), greedy (5 times), and randomized greedy (30 times). Additionally, GVNS with a perturbation strength of 5 will be run 10 times using the deterministic greedy as the initial solution. No time or iteration limits are considered for these experiments. The initial solution time construction is included in the result time of the method.

The cost comparison is done using the mean and minimum cost gaps. The cost gap (CG) is defined as the relative improvement of the method's cost, $cost(m, n)$ compared to the reference $cost(ref, n)$ over n runs:

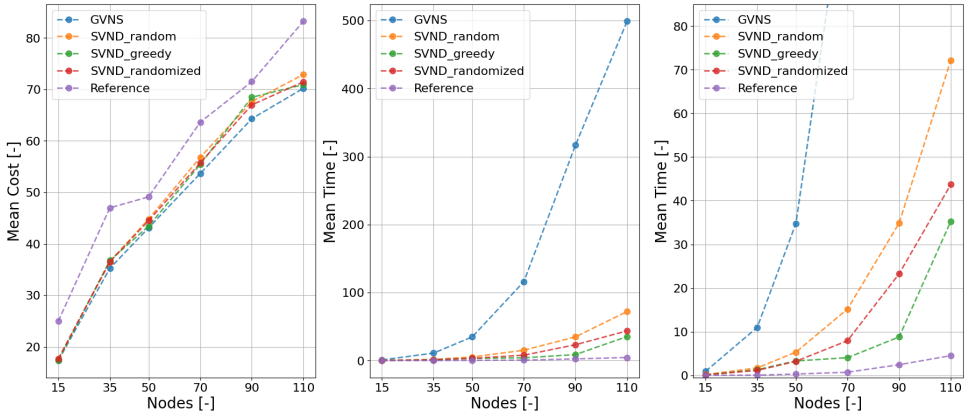
$$CG(n) = 100\frac{cost(m, n) - cost(ref, n)}{cost(ref, n)}[\%] \tag{4.1}$$

For the mean cost gap, we compare the reference method's mean cost to the proposed method's mean cost, evaluating the average improvement. For the minimum cost gap, we compare the lowest cost obtained by the reference method to the lowest cost obtained by the proposed method, evaluating the best results achieved by each method.

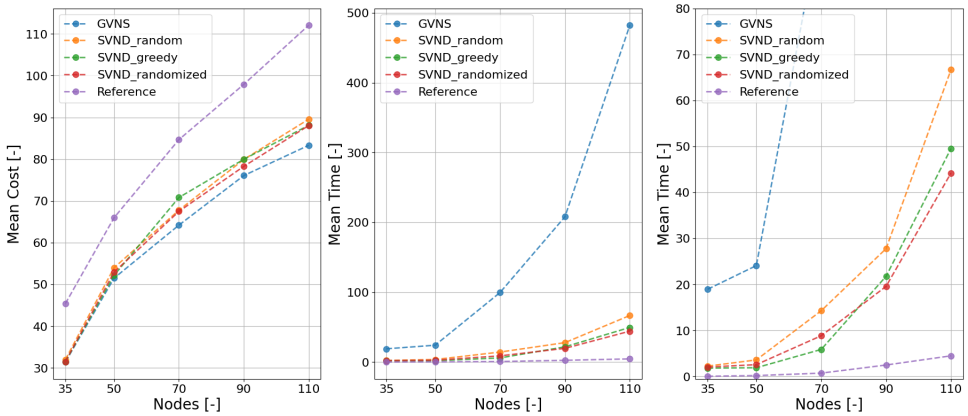
4.4.1 Different instances all maps

In this experiment, we repeat the methodology from the previous section, but this time, we include a wider range of node instances. This includes scenarios with the minimal number of nodes that can be generated on this map and instances with a larger number of nodes:

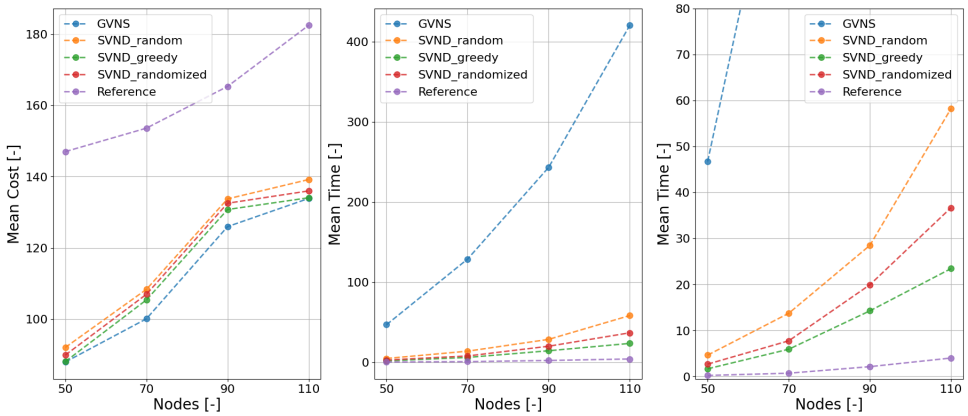
- *potholes* - $\{15, 35, 50, 70, 90, 110\}$.
- *jari-huge* - $\{35, 50, 70, 90, 110\}$.
- *large* - $\{50, 70, 90, 110\}$.



(a) : *potholes*

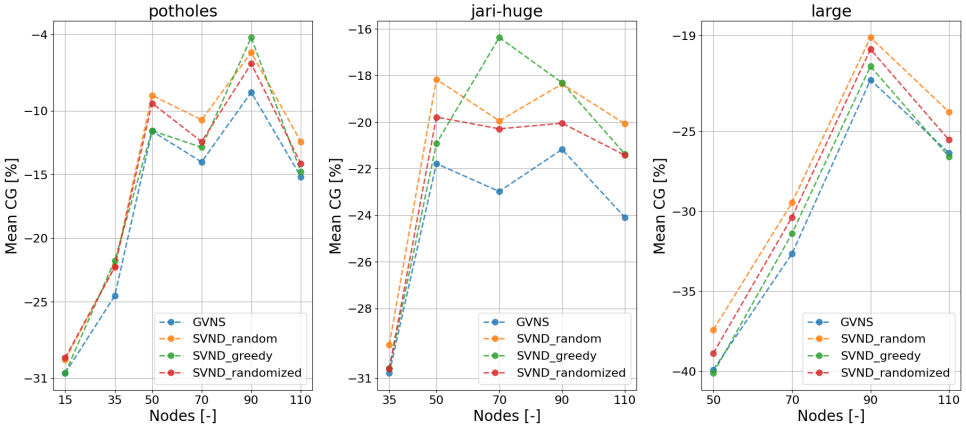


(b) : *jari-huge*

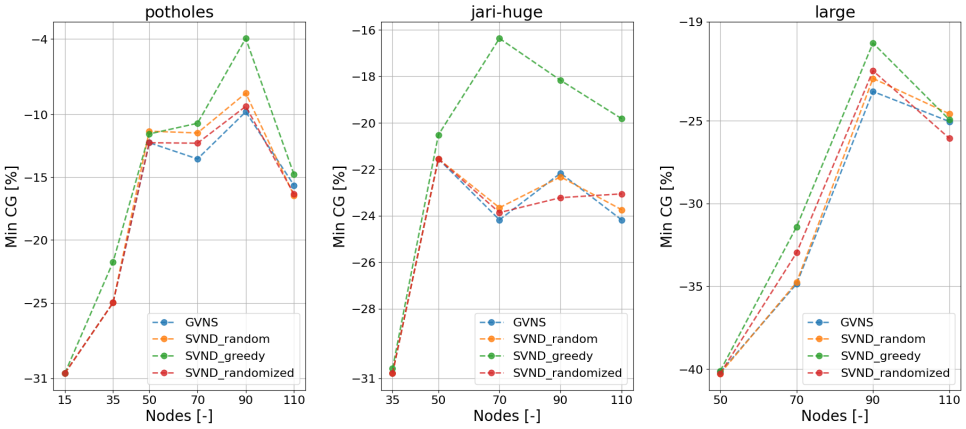


(c) : *large*

Figure 4.14: Comparison of SVND and GVNS on all maps across different node instances: left column shows cost, middle illustrates time, and right is the cut of the middle



(a) : Mean



(b) : Min

Figure 4.15: Cost Gaps for different instances on all maps

From the results of the experiment shown in Figure 4.14 and the cost gaps depicted in Figure 4.15, we observe that our proposed metaheuristics perform better in terms of cost compared to the reference method for all node instances. The cost improvement is quite dependent on the map. The mean CG for *potholes* ranges from 4-31%, *jari-huge* from 16-31%, and *large* from 20-40%. It's interesting to note that the biggest improvements are achieved for the minimal number of nodes on each map. The larger intersections of visibility regions can explain this compared to instances with more nodes.

The minimal (CG) shows that the worst best solutions, compared to the best solutions of the reference method, are obtained with the greedy SVND: 4% for *potholes* with 90 nodes, 16% for *jari-huge* with 70 nodes, and 20% for *large* with 90 nodes. This indicates that the deterministic nature of the initial solution restricts the metaheuristic from obtaining more promising solutions. At the same time, GVNS achieves the best mean cost gaps, but the difference in cost gap between randomized and fully random SVND is about 1-3% across all maps.

For the maximal measured number of 110 nodes, the mean time of GVNS is 400-500 seconds, highlighting challenges for using this metaheuristic with

an even larger number of nodes. On the other hand, SVND converges in about 20-80 seconds for the same 110 nodes, which is also relatively slow compared to the reference method, which takes 3-5 seconds. However, the average time for the fully random SVND is 60-70 seconds, higher than the randomized greedy approach at 30-50 seconds. This can be explained by the more variable initial solutions, which take more time to converge because they are far from the local optimum.

Based on these observations, we propose the following ranking of cost and time performance methods summarized in Table 4.2

Method	Cost	Time
GVNS	1	4
Full random SVND	2-3	3
Greedy SVND	4	1
Randomized greedy SVND	2-3	2

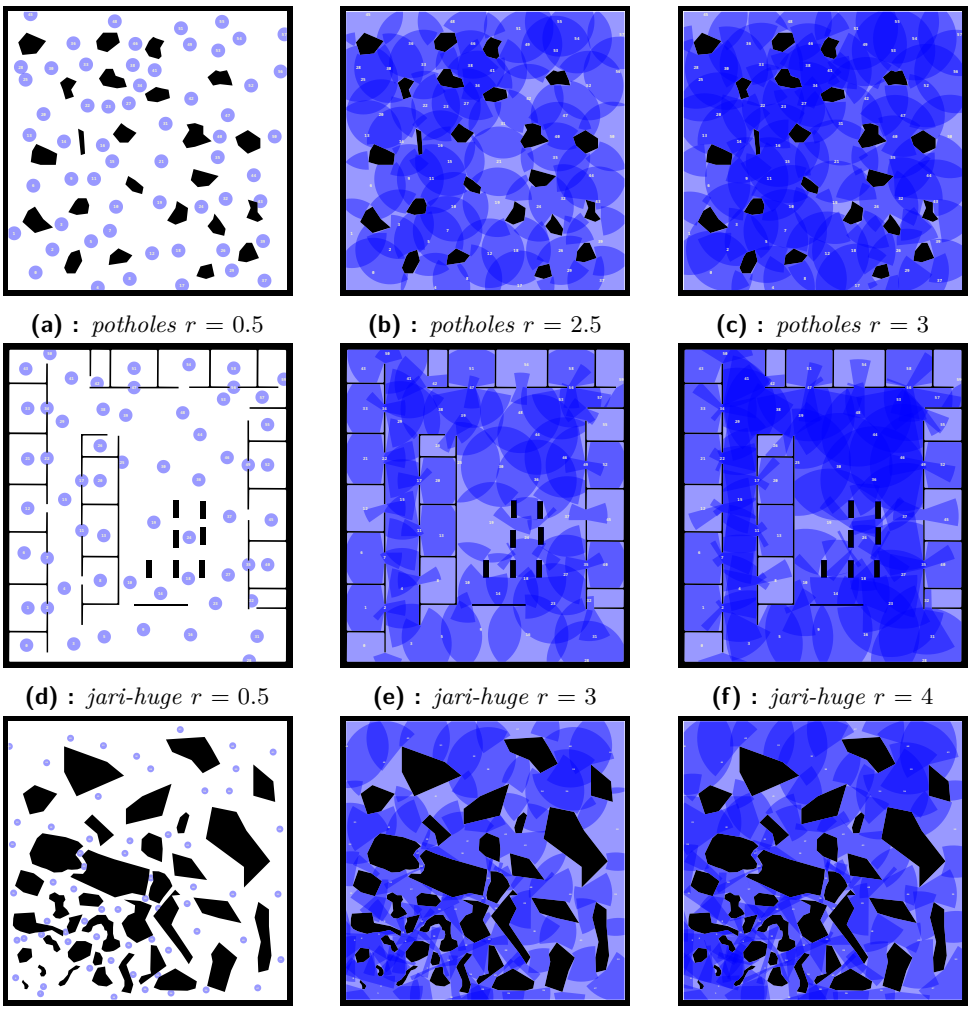
Table 4.2: Ranking of SVND and GVNS for different instances experiment

4.4.2 Extended visibility radius

The next experiment aims to assess the performance of the proposed methods for different measurements of visibility regions overlapping. The hypothesis is that greater overlapping leads to larger weight errors in the reference method.

We selected one visibility radius for each map and generated approximately 60 nodes. Next, we scaled the generated visibility regions with radii ranging from 0.5 to the size of the map's diagonal. The distances between the regions remain unchanged. A radius of 0.5 simulates no overlapping, resulting in GSP with static weights, see4.16. The map's diagonal size is the theoretical visibility limit because increasing the radius beyond this point has no practical effect, as the map's borders constrain it. The selected visibility radius for generation, number of nodes, and chosen visibility limit for each map are summarized below:

- *potholes*: (2.5, 58, 30).
- *jari-huge*: (3, 60, 30).
- *large*: (6, 62, 60).



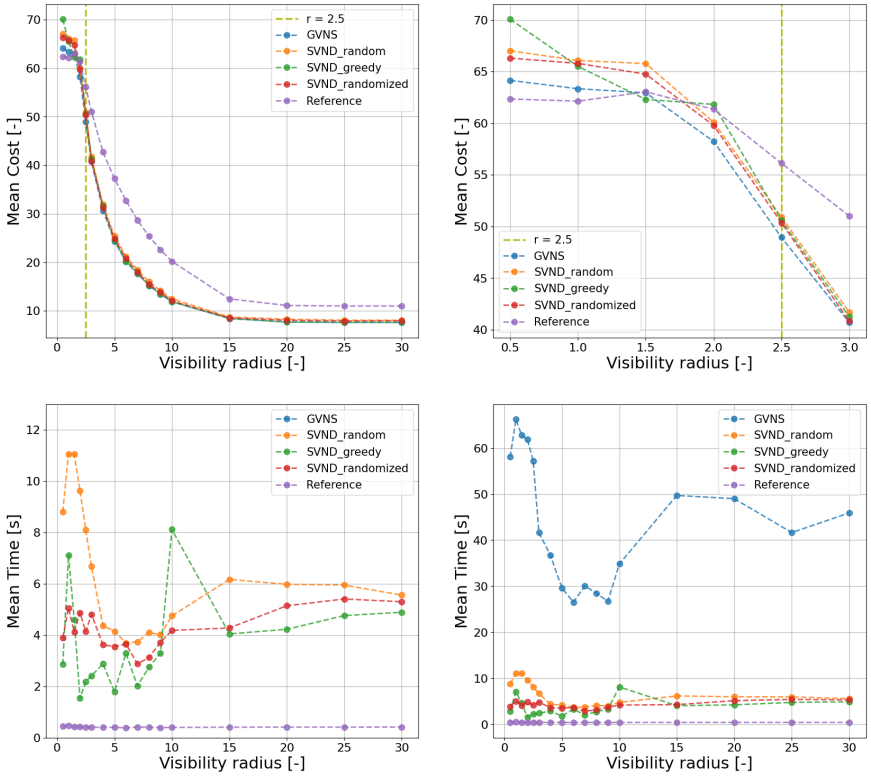
(a) : *potholes* $r = 0.5$ **(b) :** *potholes* $r = 2.5$ **(c) :** *potholes* $r = 3$

(d) : *jari-huge* $r = 0.5$ **(e) :** *jari-huge* $r = 3$ **(f) :** *jari-huge* $r = 4$

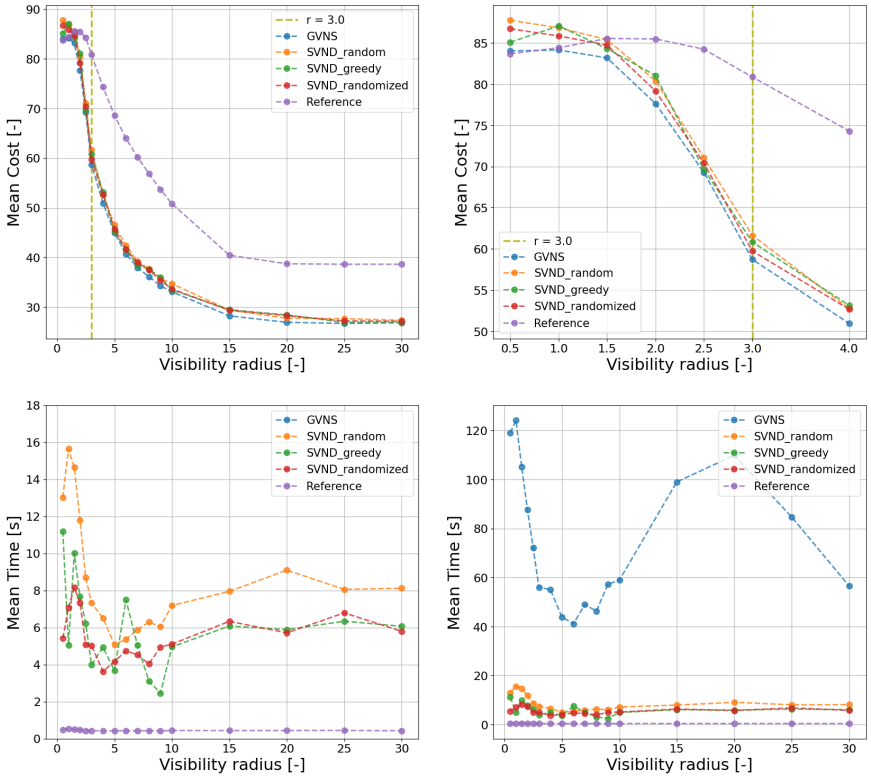
(g) : *large* $r = 0.5$ **(h) :** *large* $r = 6$ **(i) :** *large* $r = 7$

Figure 4.16: Illustration of visibility regions overlapping for different scaled visibility radii

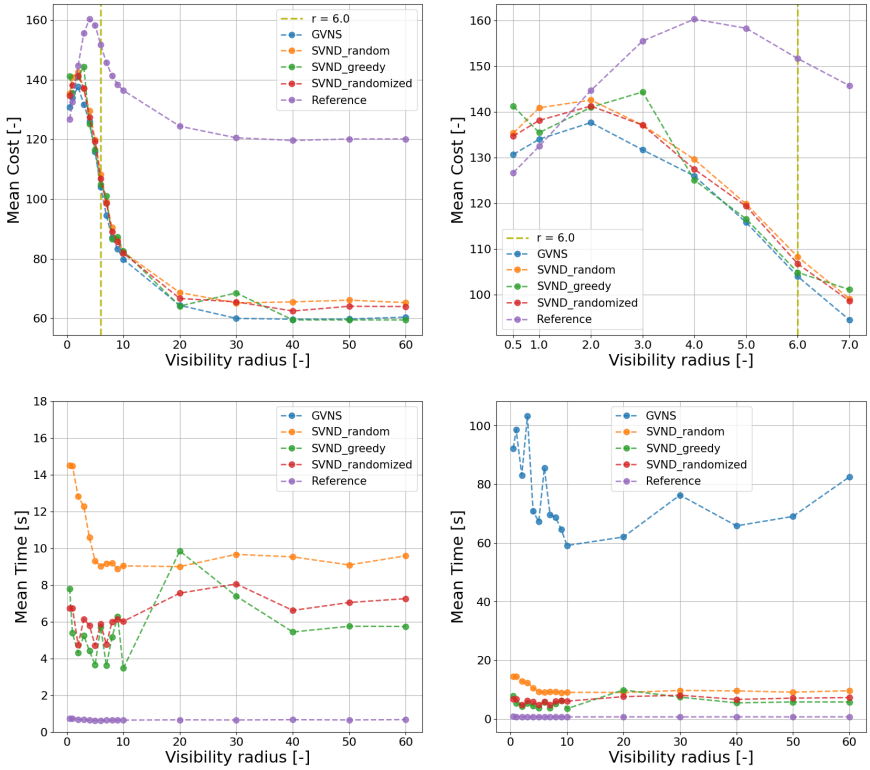
We measured the proposed metaheuristics for each scaled visibility radius using the same initial solutions and their respective number of runs as in the previous experiment. Due to the imprecision of polygon operations for the scaled visibility radius, we empirically set a threshold of 10^{-5} for our local search procedures. During the local search procedure, if a neighbor with a better cost is found, we update the current best solution only if the absolute cost difference $|\Delta_{cost}|$ between the neighbor and the current best is greater than the threshold. Otherwise, we will consider them the same solution.



(a) : *potholes*

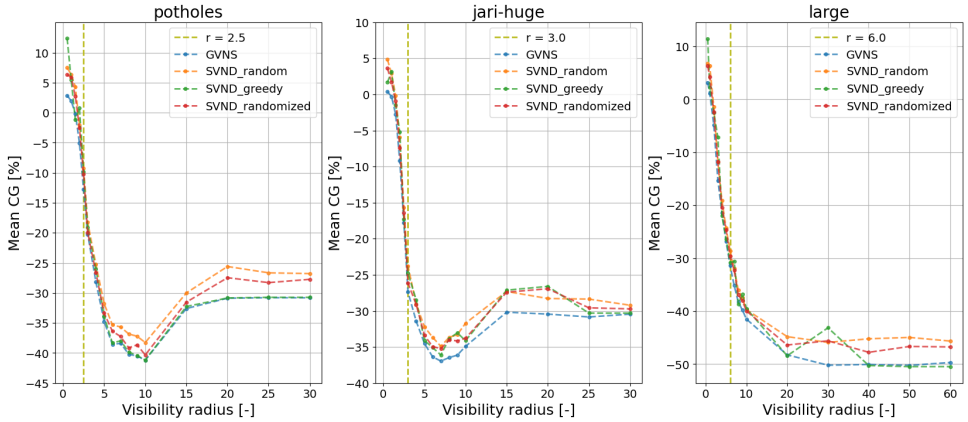


(b) : *jari-huge*

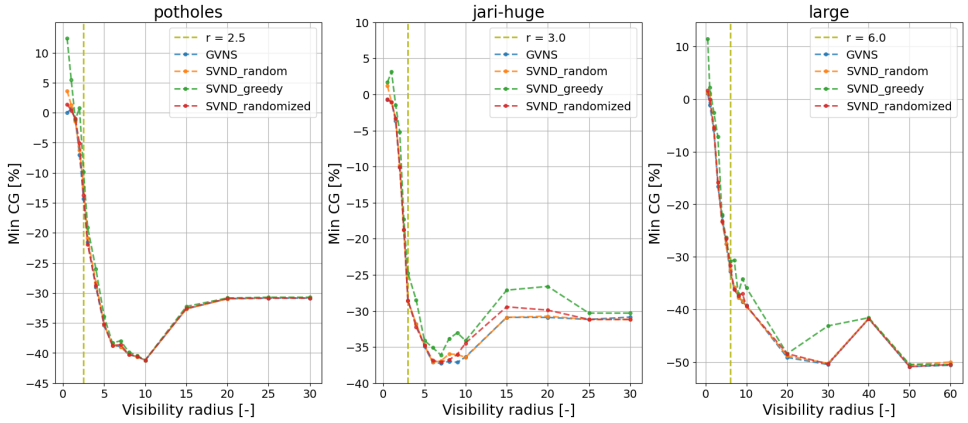


(c) : *large*

Figure 4.17: Cost and time comparison for SVND and GVNS for scaled visibility radii. The first row of the subfigure shows: the first column - cost, the second column - cost for the scaled radii near the generated one. The second row of the subfigure: cut of the time and the time.



(a) : Mean



(b) : Min

Figure 4.18: Cost gaps for scaled visibility radii

The results of the experiment, shown in Figures 4.17 and 4.18, confirm that as the scale of visibility regions increases, the Cost Gap (CG) also increases, and the proposed methods perform better. For the maximal overlapping scenarios, the CGs converge to the following values: *potholes* and *jari-huge* around 30%, and *large* about 50%.

Interestingly, on smaller maps, we observe even higher CGs: for *potholes* with a scaled visibility radius of 10, the CG is about 40%, and for *jari-huge* with a radius of 7, the CG is about 35%. This can be attributed to specific properties of these maps.

However, it is notable that the Cost Gap is worse compared to the reference method for non-overlapping regions (radius = 0.5), with a CG of about 0-10%. Theoretically, the GSP with order-dependent weights should not perform worse than the GSP with static weights. We propose that this discrepancy is unrelated to the weights of visibility regions, and the reference method is inherently stronger for non-overlapping regions because it is more advanced from the solution approach perspective.

In the end, we aim to derive the map property that defines overlapping regions, which sheds light on the performance of GSP with order-dependent weights. We define the visibility regions overlapping $\mathcal{OVR}$ for visibility regions $(\mathcal{R}_0, ..., \mathcal{R}_n)$ as

$$\mathcal{OVR} = a(\bigcup_{i \neq j} \mathcal{R}_i \cap \mathcal{R}_j \setminus \mathcal{R}_0) \tag{4.2}$$

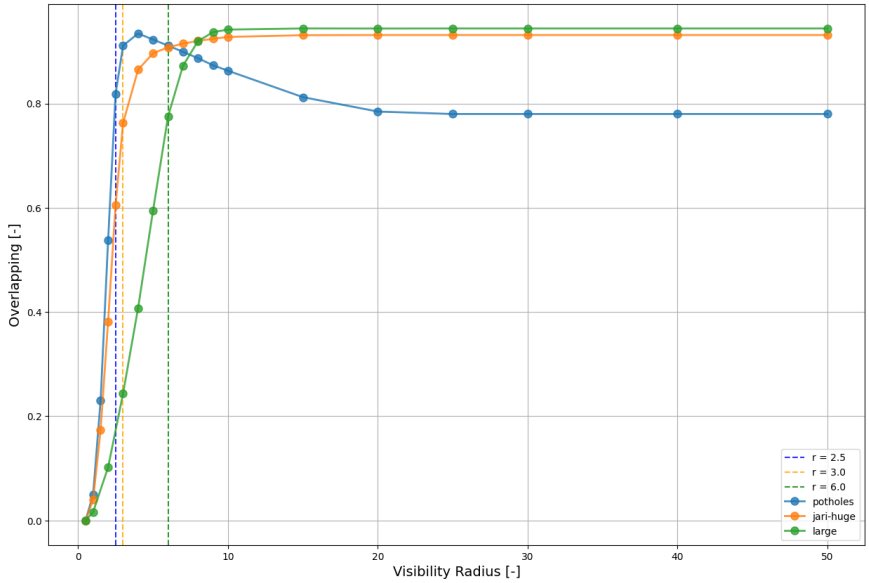


Figure 4.19: Visibility regions overlapping for different maps

Theoretically, the following coefficient defines the total area of intersecting parts of regions without including the starting $\mathcal{R}_0$, as the starting region's weight doesn't directly contribute to any edge on the path. Figure 4.19 shows that for the maps *jari-huge* and *large*, the overlapping saturates without any peaks. This can be explained by the fact that the starting region is limited by some obstacles, preventing it from covering more of the environment. As the area of other regions expands, the degree of overlapping also increases. We propose that initially, the obstacles are located at a distance of 4, but they are not very dense. As the scaling increases, the scaled starting region can overcome these obstacles and cover more of the environment. Saturation occurs when the scaled $\mathcal{R}_0$ faces the map's outer boundaries.

However, the proposed overlapping definition doesn't fully explain why we observe better cost gaps before reaching the visibility limit on the maps *potholes* and *jari-huge*. The proposed coefficient only helps identify the visibility limit, which, in reality, is smaller than the size of the map's diagonal. It can determined by observing when this coefficient value saturates rather than applying metaheuristics to multiple input solutions across different scaled visibility radii.

Chapter 5

Conclusion

In this work, we proposed a new approach for Mobile Robot Search in the polygonal domain by introducing the problem as the Graph Search Problem with order-dependent weights, described in Chapter 2. This formulation allows the improvement of the existing methods for solving MRS.
In Chapter 3, we presented a comprehensive solution approach, systematically divided into several key steps. In Section 3.1, we introduced three constructive heuristics, full random, greedy, and randomized greedy, that facilitate the initial construction of solutions from scratch. We proposed five distinct criteria to enhance the effectiveness of the greedy heuristics, enabling the generation of various deterministic initial solutions. These heuristics and criteria collectively form a robust foundation for the initial solution phase. In Section 3.2, we introduced improving heuristics based on the Iterated Local Search. We considered two basic local search operators: Insert and 2-Opt. The naive implementation of their local search procedures has a computational complexity of $\mathcal{O}(n^3)$. We introduced an additional operator that swaps two consecutive vertices in the path to address this. It was demonstrated that the path cost update after performing the swap can be computed in constant time by precomputing certain path parameters and updating them accordingly. This innovation allowed us to define a swap state-space where each possible neighboring permutation can be obtained through the recurrent application of swapping two neighboring vertices. We proposed effective Swap-Based Insert and 2-Opt procedures based on the swap state-space. The complexity of the basic Insert was reduced to $\mathcal{O}(n^2)$. While the upper bound complexity of Swap-Based 2-Opt remains the same, the number of operations required to explore the neighborhood is significantly reduced to $\frac{n^3-n}{6}$: for large instances (up to 175 nodes) the complexity is $n^2 \log^2(n)$, making the swap-based 2-Opt more efficient than the basic 2-Opt. Also, we proposed the additional Balas-Simonetti (BSk) for $k = 2, 3, 4$ operator that allows us to find the best-improved neighbor dynamically. In Section 3.3, we presented two metaheuristics: Sequential Variable Neighborhood Descent and General Variable Neighborhood Search with double-bridge application for solution perturbation. These metaheuristics further enhance optimization by systematically exploring and escaping local optima.
In Chapter 4, a series of experiments were conducted to evaluate the correct-

ness and efficacy of the chosen approach. From the experiments detailed in Section 4.2, three local search procedures were selected for integration into our SVND: Swap-Based Insert, Swap-Based 2-Opt, and the Balas-Simonetti operator for $k = 4$ (BS4). However, based on the results from Section 4.3, it became evident that the impact of BS4 was relatively minor compared to the other operators. Consequently, it was omitted from the list of operators utilized in SVND. Furthermore, an optimal 2-Opt depth of approximately 30% was determined, enabling the removal of unpromising neighbors and further reducing computational complexity. Evaluation of the *Best* and *First* improvement strategies for our operators within SVND led to selecting the *Best* for both. Subsequently, upon constructing SVND, we determined that the criterion from Equation 3.3 is best for the greedy constructive heuristics. Additionally, a probability set of $(0.85, 0.15, 0.05)$ was established for the randomization of deterministic greedy. For the General Variable Neighborhood Search (GVNS), a perturbation strength of 5 was selected, with results proving that increasing perturbation strength led to better solutions. In Section 4.4, we compared SVND employing three different initial solution types (fully random, deterministic, and randomized greedy) and GVNS with the state-of-the-art method in two experiments. First, we compared the cost and time performance on different input instances and obtained cost gap improvements of 6-40 %, depending on the map and the node instances. In the second experiment, we compared the methods for increasing overlapping of visibility regions. The cost improvement for 60 nodes was about 30-50 % for the maximal scaled visibility radii. Also, we defined the visibility regions overlapping that allows understanding when the visibility limit occurs for different scaling of regions
Overall, the proposed metaheuristics perform better than the reference method, with the exception of the non-overlapping visibility regions leading to static weights, which the reference method solves more efficiently due to a more advanced solution approach. The new formulation of Mobile Robot Search for GSP with order-dependent weights allows for an improved solution approach and can be considered a solid starting point for further research.

5.1 Future work and improvements

We will highlight the following possible extensions to the project.

- Extension of the Balas-Simonetti neighborhood to larger values of k utilizing A^* for finding the shortest path and evaluation of different greedy criteria for that purpose. This neighborhood can be applied as an intensification procedure in SVND/GVNS or as part of the constructive heuristics.
- Investigation into potential time improvements of the metaheuristics by implementing time or iteration limits for the ILS procedures, reducing the perturbation strength for GVNS, and introducing multi-start GVNS with multiple stopping criteria for GSP with order-dependent weights.

- Further exploration of overlapping, with a focus on proposing a new definition that better describes the properties of the maps.
- Continued research into impreciseness in polygon operations and experimentation with different values of thresholds in our local search procedures.

Bibliography

[1] Wikipedia, "Travelling salesman problem — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/wiki/Travelling_salesman_problem`, 2024. Accessed: 2024-05-08.

[2] J. Mikula and M. Kulich, "Solving the traveling delivery person problem with limited computational time," *Central European Journal of Operations Research*, vol. 30, pp. 1451–1481, Dec 2022.

[3] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[4] P. Hansen, N. Mladenovic, and J. Moreno-Perez, "Variable neighborhood search: Methods and applications," *4OR*, vol. 175, pp. 367–407, 02 2010.

[5] I. Mendez-Diaz, P. Zabala, and A. Lucena, "A new formulation for the traveling deliveryman problem," *Discrete Applied Mathematics*, vol. 156, no. 17, pp. 3223–3237, 2008.

[6] M. Fischetti, G. Laporte, and S. Martello, "The delivery man problem and cumulative matroids," *Operations Research*, vol. 41, pp. 1055–1064, 12 1993.

[7] A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan, and M. Sudan, "The minimum latency problem," in *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pp. 163–171, 1994.

[8] M. Bruni, P. Beraldi, and S. Khodaparasti, "A heuristic approach for the k-traveling repairman problem with profits under uncertainty," *Electronic Notes in Discrete Mathematics*, vol. 69, pp. 221–228, 2018. Joint EURO/ALIO International Conference 2018 on Applied Combinatorial Optimization (EURO/ALIO 2018).

[9] T. Bulhoes, R. Sadykov, and E. Uchoa, "A branch-and-price algorithm for the minimum latency problem," *Computers & Operations Research*, vol. 93, pp. 66–78, 2018.

[10] E. Koutsoupias, C. Papadimitriou, and M. Yannakakis, "Searching a fixed graph," in *Automata, Languages and Programming: 23rd International Colloquium, ICALP'96 Paderborn, Germany, July 8–12, 1996 Proceedings 23*, pp. 280–289, Springer, 1996.

[11] G. Ausiello, S. Leonardi, and A. Marchetti-Spaccamela, "On salesmen, repairmen, spiders, and other traveling agents," vol. 1767, pp. 1–16, 03 2000.

[12] M. Kulich, J. J. Miranda Bront, and L. Přeučil, "A meta-heuristic based goal-selection strategy for mobile robot search in an unknown environment," *Computers & Operations Research*, vol. 84, Aug. 2017.

[13] M. Kulich and L. Přeučil, "Multi-robot search for a stationary object placed in a known environment with a combination of grasp and vnd," *International Transactions in Operational Research*, vol. 29, pp. 805–836, Apr. 2022.

[14] L.-P. Bigras, M. Gamache, and G. Savard, "The time-dependent traveling salesman problem and single machine scheduling problems with sequence dependent setup times," *Discrete Optimization*, vol. 5, no. 4, pp. 685–699, 2008.

[15] B. Ha-Bang, "An efficient two-phase metaheuristic algorithm for the time dependent traveling salesman problem," *RAIRO - Operations Research*, vol. 53, 01 2019.

[16] H. Abeledo, R. Fukasawa, A. Pessoa, and E. Uchoa, "The time dependent traveling salesman problem: Polyhedra and branch-cut-and-price algorithm," vol. 5, pp. 202–213, 05 2010.

[17] I. Osman and G. Laporte, "Metaheuristics: A bibliography," *Annals of Operational Research*, vol. 63, pp. 513–628, 10 1996.

[18] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, pp. 268–308, 01 2001.

[19] M. Gendreau and J.-Y. Potvin, *Handbook of Metaheuristics*. Springer Publishing Company, Incorporated, 2nd ed., 2010.

[20] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr, "A survey on metaheuristics for stochastic combinatorial optimization," *Natural Computing*, vol. 8, pp. 239–287, 2009.

[21] N. Mladenovic, "A tutorial on variable neighborhood search," 01 2004.

[22] A. Salehipour, K. Sörensen, P. Goos, and O. Bräysy, "Efficient grasp+vnd and grasp+vns metaheuristics for the traveling repairman problem," *4OR*, vol. 9, pp. 189–209, 06 2011.

[23] M. Silva, A. Subramanian, T. Vidal, and L. Ochi, "A simple and effective metaheuristic for the minimum latency problem," *European Journal of Operational Research*, vol. 221, pp. 513–520, 09 2012.

[24] J. Mikula, "The traveling deliveryman problem in a specific application scenario: Meta-heuristic vs. ilp approach." `https://cw.fel.cvut.cz/b222/_media/courses/ko/traveling_deliveryman.pdf`. Accessed: 2024-05-10.

[25] M. Kulich, L. Přeučil, and J. J. Miranda Bront, "Single robot search for a stationary object in an unknown environment," in *ICRA2014: Proceedings of 2014 IEEE International Conference on Robotics and Automation*, (Piscataway, US), IEEE, 2014.

[26] A. Sarmiento, R. Murrieta-Cid, and S. Hutchinson, "A multi-robot strategy for rapidly searching a polygonal environment," in *Advances in Artificial Intelligence–IBERAMIA 2004: 9th Ibero-American Conference on AI, Puebla, Mexico, November 22-26, 2004. Proceedings 9*, pp. 484–493, Springer, 2004.

[27] J. Mikula and M. Kulich, "Towards a continuous solution of the d-visibility watchman route problem in a polygon with holes," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 5934–5941, 2022.

[28] J. Mikula, "Search for a static object in a known environment." `http://hdl.handle.net/10467/92871`, 2021.

[29] J. Mikula and M. Kulich, "Optimizing mesh to improve the triangular expansion algorithm for computing visibility regions," *SN Computer Science*, vol. 5, p. 262, Feb 2024.

[30] C. Gonzalez, "Using AI to solve the N-Queens Problem." `https://medium.com/@carlosgonzalez_39141/using-ai-to-solve-the-n-queens-problem-2a5a9cc5c84c`.

[31] P. Hansen and N. Mladenovic, *Variable Neighborhood Search Methods*, vol. 22, pp. 3978–. 01 2009.

[32] P. Hansen and N. Mladenović, "First vs. best improvement: An empirical study," *Discrete Applied Mathematics*, vol. 154, no. 5, pp. 802–817, 2006.

[33] A. Blazinskas, "Combining 2-opt , 3-opt and 4-opt with k-swap-kick perturbations for the traveling salesman problem," 2011.

[34] E. Balas, "New classes of efficiently solvable generalized traveling salesman problems," *Annals of Operations Research*, vol. 86, 01 1999.

[35] J. Schmidt and S. Irnich, "New neighborhoods and an iterated local search algorithm for the generalized traveling salesman problem," *EURO Journal on Computational Optimization*, vol. 10, p. 100029, 05 2022.

[36] E. Balas and N. Simonetti, "Linear time dynamic-programming algorithms for new classes of restricted tsps: A computational study," *INFORMS Journal on Computing*, vol. 13, 10 2000.

[37] A. Duarte, J. Sánchez-Oro, N. Mladenović, R. Todosijević, R. Martí, P. M. Pardalos, and M. G. C. Resende, "Variable neighborhood descent.," 2018.

[38] N. Mladenovic, R. Todosijević, and D. Urosevic, "An efficient general variable neighborhood search for large travelling salesman problem with time windows," *YUJOR. Yugoslav Journal of Operations Research*, vol. 23, 01 2013.

[39] P. Hansen and N. Mladenović, "Variable neighborhood search: Principles and applications," *European Journal of Operational Research*, vol. 130, no. 3, pp. 449–467, 2001.

[40] S. Lin and B. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations Research*, vol. 21, pp. 498–516, 1973.

[41] F. Carrabs, J.-F. Cordeau, and G. Laporte, "Variable neighborhood search for the pickup and delivery traveling salesman problem with lifo loading," *INFORMS Journal on Computing*, vol. 19, 04 2015.

[42] D. Karapetyan and G. Gutin, "Lin-kernighan heuristic adaptations for the generalized traveling salesman problem," *European Journal of Operational Research*, vol. 208, pp. 221–232, 02 2011.

Appendix A

A.1 Additional Software

We used the following AI tools: *ChatGPT*[1], *Grammarly* [2], *Github Copilot*[3] according to *Methodical Guideline No. 5/2023* [4].

A.2 Attachments

- **thesis_latex.zip**: The .zip file containing the LaTeX files of the thesis.
- **scripts.zip**: The .zip file with the Python scripts used for the research and running of our methods in the experiments.
- **graphs.zip**: The .zip file containing graphs from the experimental section of the thesis.
- **visis_codes.zip**[5]: The source codes of the implemented methods.

[1] https://chat.openai.com/chat
[2] https://www.grammarly.com/
[3] https://github.com/features/copilot
[4] https://www.cvut.cz/sites/default/files/content/d1dc93cd-5894-4521-b799-c7e715d3c59e/cs/20240130-metodicky-pokyn-c-52023.pdf
[5] https://gitlab.ciirc.cvut.cz/mission-planning/visis-planner/-/tree/dsearch