Master Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

The Generalized Travelling Deliveryman Problem

Daniel Kubišta

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Study program: Cybernetics and Robotics
May 2025

Acknowledgements

I would like to thank RNDr. Miroslav Kulich, Ph.D., for the insightful discussions and his guidance throughout the majority of my studies.

Declaration

I declare that I have elaborated the master's thesis entitled The Generalized Travelling Deliveryman Problem independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

Prague, 23. May 2025

Abstract

The Mobile Robot Search Problem (MRSP) is a task of searching for an object in a priory known environment. The location of the object is not known. The goal is to find a trajectory that minimizes the expected time to find the object. The discretized variant of the problem can be viewed as a graph theory problem. This thesis approaches the task by representing the MRSP as a novel combinatorial optimization problem named the Generalized Graph Search Problem (GSP) with order-dependent weights, which combines the Generalized Traveling Salesman Problem (GTSP) and the GSP. The novel problem can be tackled by modifying GLNS, a solver designed for solving the GTSP using metaheuristic concepts Adaptive Large Neighborhood Search and Simulated Annealing. With proposed modifications, the solver is able to efficiently solve the Generalized GSP with order-dependent weights, as we demonstrate in the experimental evaluation.

Keywords: metaheuristics, Graph Search Problem, Traveling Deliveryman Problem, Generalized Traveling Salesman Problem

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Intelligent and Mobile Robotics, CIIRC, Jugoslávských partyzánů 1580/3, 160 00 Praha 6

Abstrakt

Mobile Robot Search Problem (MRSP) je plánovací úloha, při které robot hledá předmět v předem známém prostředí. Poloha objektu není známa. Cílem úlohy je naplánovat trajektorii, která minimalizuje očekávaný čas nalezení předmětu. Na diskretizovanou variantu tohoto problému lze pohlížet jako na úlohu z teorie grafů. Tato práce přistupuje k tomuto problému tak, že reprezentuje MRSP jako nový problém nazvaný Generalized Graph Search Problem with order-dependent weight, který kombinuje známé problémy Graph Search Problem (GSP) a Generalized Traveling Salesman Problem (GTSP). Nový problém lze řešit úpravou algoritmu GLNS, navrženého pro řešení GTSP, pomocí metaheuristických konceptů Adaptive Large Neighborhood Search a Simulated Annealing. S navrženými úpravami algoritmu jsme schopni efektivně řešit Generalized GSP with order-dependent weight, jak dokládáme pomocí experimentů.

Klíčová slova: metaheuristika, Graph Search Problem, Traveling Deliveryman Problem, zobecněný problém obchodního cestujícího

Překlad názvu: Zobecněný problém obchodního doručovatele

Contents

Figures

Tables

Chapter 1

Introduction

Continuous improvements of sensing and locomotive capabilities open many opportunities and challenges for mobile robots. One of the challenges is the Mobile Robot Search Problem (MRSP) - a search for an object with unknown location located in a priory known environment. The real life applications of this problem include the Search and Rescue, a task where mobile robots help to localize the survivors of disasters such as fires, earthquakes and floods [4], localization and tracking of animals in the wildlife conservation [3] or robot inspection and monitoring [12].

Due to the complex nature of the task, simplifications are necessary in order to make the problem tractable. A common approach is discretization of the searched space followed by transforming the task into a combinatorial optimization problem. In [23] the problem is solved with transformation into the Traveling Deliveryman Problem (TDP), papers [17, 15] transform the problem to the Graph Search Problem (GSP) and [18, 16] cover the GSP with multiple robots. The most recently [27] proposes the GSP with order-dependent weights to tackle the problem.

This thesis approaches the problem by first turning the problem into the Generalized GSP with order-dependent weights and then solving this graph theory problem using modified GLNS - a solver designed for solving the Generalized Traveling Salesman Problem (GTSP) problem based on the Adaptive Large Neighborhood Search (ALNS) and the Simulated Annealing (SA) metaheuristics.

1.1 State of the art

In this section we present some of the published methods for solving the GTSP, the TDP and the GSP.

GTSP is a popular NP-hard combinatorial optimization problem that was first defined in [37] in 1969, [28] provides an exhaustive list of methods for solving the problem. The survey categorizes them into five categories.

Exact algorithms are able to produce the optimal solution. For NP-hard problems is the search very time consuming due to high complexity of the calculations and large search space, therefore they are usually only applicable on relatively small problems. These algorithms include dynamic programming [37], branch-and-bound [5], integer programming [19] and branch-and-cut [7] methods.

Transformation methods transform the generalized problem to the classical Traveling Salesman Problem (TSP), which is then solved using one of the standard methods for the TSP. A famous Noon and Bean transformation method was introduced in [26]. This method transforms the asymmetric GTSP to the asymmetric TSP while keeping the same number of vertices but significantly increasing the number of edges.

Reduction algorithms represent a class of preprocessing procedures that eliminate edges and vertices that are guaranteed not to be present in the optimal solution. The preprocessing leads to reduction in the size of instances and improved computation time. In paper [9] three reduction algorithms are described.

Approximation methods are polynomial time algorithms that produce an approximation of the optimal solution with demonstrable guarantees on the quality of the solution. The design of these algorithms includes a mathematical proof that the found solution has in the worst case some guaranteed quality. An instance of this methods is poly-logarithmic approximation algorithm, which is applicable on the GTSP and was introduced in [8].

Heuristic and metaheuristic algorithms are algorithms that provide no assurance on optimality nor sub-optimality of the found solution. However, they usually find solutions with reasonably good quality. These algorithms are especially useful on bigger problems, where the exact methods are too computationally expensive and on problems, where the exact algorithms are not known. These algorithms include genetic algorithms [36], ant colony optimization [39], evolution algorithms [10], large neighborhood search [35], iterated local search [32] and variable neighborhood search [11].

TDP is a NP-hard problem closely related to the TSP, in [23] authors recognize two main courses in the research community *exact algorithms* and *heuristic and metaheuristic algorithms*. Early exact algorithms were based upon non-linear integer formulation [1]. Later papers on exact methods propose algorithms utilizing linear integer formulations. These methods provide optimal solution, but are only suitable for small instances due to infeasible computational time. Heuristic and metaheuristic algorithms do not provide any guarantees on the quality of the found solution, however, they still provide reasonably good quality in exchange for better computational time. Most of the methods modified for solving the TDP utilize a combination of Greedy Randomized Adaptive Search Procedure (GRASP), Variable Neighborhood Search (VNS) or Variable Neighborhood Descent (VND) [30, 25].

GSP - a NP-hard problem [31], which extends the TDP, was introduced in [14]. This problem has not received as much attention as the two former problems, however, there is still some relevant research on this problem. In [31, 17] greedy algorithms, which construct a solution by optimizing the immediate utility, are proposed. In [15] a tailored GRASP is developed. In [18] a algorithm based on a combination of GRASP and VND is presented for solving the Multi-robot GSP. In [27] metaheuristics Sequential Variable Neighborhood Descent (SVND) and General Variable Neighborhood Search (GVNS) are used for solving the GSP with order-dependent weights.

1.2 Contributions

This thesis contributes by the following points:

- The Generalized GSP with order-dependent weights is defined as a combination of two graph theory problems: GTSP and GSP.
- Several approximations are examined. Modifications to the approximation methods, used in the classical GSP that can be used in the Generalized GSP are proposed.

- The original GLNS operators are modified specifically to optimize the Generalized GSP. Methods to reduce the time complexity of the heuristic operator evaluation are proposed.
- The performance is experimentally evaluated. Specifically the trade-off between the quality of approximation and computational time is examined. The algorithms are tested on four maps with the size of instances up to 143 clusters and 841 vertices. Finally the modified Effective large neighborhood search heuristic for the Generalized Traveling Salesman Problem (GLNS) is compared to the *Hexaly Optimizer*[1] - "the world's fastest optimization solver for Routing, Scheduling, Packing, and more".

[1]https://www.hexaly.com/hexaly-optimizer

Chapter 2

Mobile Robot Search Problem

In MRSP, a robot navigates through a priory known environment while searching for an object of interest and optimizing the expected time to find the object. In this chapter the problem is described using definitions from [27, 31, 21].

The robot is modeled by a point agent equipped with an omnidirectional sensor with a limited sensing range $r_{\mathcal{V}}$. Robot moves at a constant positive speed and can change its heading instantaneously. Because of the previous assumption, the time needed to follow any trajectory is proportional to the length of the corresponding path.

The environment $\mathcal{W}$ is completely known, static space modeled as a polygon with polygonal obstacles. Obstacles can pose motion and visibility constraints. We assume that all the obstacles that limit the movement of the robot are sensed by its sensor and vice versa. We further assume that the robot can reach any point $\mathbf{y} \in \mathcal{W}$ from any starting position $\mathbf{x}_0 \in \mathcal{W}$. The searched object can be sensed and is located somewhere in the environment with no prior information about its location. Therefore we assume that the probability of an object being in a certain region $\mathcal{R} \subseteq \mathcal{W}$ is evenly distributed along the environment $\mathcal{W}$ and the probability is proportional to the area of the region.

When the robot searches for the object, it moves along a trajectory τ, where $\tau : [0, \infty] \mapsto \mathcal{W}$. The natural objective is to find the object in minimal time. From a probabilistic perspective we want to minimize the expected (average) time to find the object. Therefore the optimization task is to find the trajectory τ^* that minimizes the expected time:

$$\mathbb{E}(T|\tau) = \int_0^\infty t f_{T|\tau}(t) dt \tag{2.1}$$

A model, where the robot does not gather information continuously, but only at discrete time steps leads to the discrete formulation of the expected time. In this model the robot performs measurements at time steps $t \in (t_0, \dots, t_m)$, $t_i < t_{i+1}$, at sensing locations $(\tau(t_0), \dots, \tau(t_m))$. In this time discrete case expected time is computed as follows:

$$\mathbb{E}(T|\tau) = \sum_{i=0}^{m} t_i f_{T|\tau}(t_i) \tag{2.2}$$

Where $f_{T|\tau}(t)$ is the probability that object was first seen in time t. The function being a probability density function, implies that $\int_0^\infty f_{T|\tau}(t)dt = 1$ or $\sum_{i=0}^{m} f_{T|\tau}(t_i) = 1$.

Figure 2.1 shows a motivational example of the MRSP. The figure displays two possible trajectories, both starting from the same point. The one on the left is denoted as τ_1 and the one on the right as τ_2. Using colors, we visualize the evolution of discovered regions of the environment. Regions with lighter colors are discovered earlier that those with darker colors. The length of the trajectory τ_1 is 19.89 and the expected time to find the searched object is $\mathbb{E}(T|\tau_1) = 7.87$. The second trajectory τ_2 is longer, with length 20.39, but the expected time is only $\mathbb{E}(T|\tau_2) = 5.9$. This example shows that a shorter trajectory does not always equal smaller expected time. Intuitively, the robot following the trajectory τ_2 has smaller expected time to find the object, because it discovers the area hidden in the area of inverted U-shape.

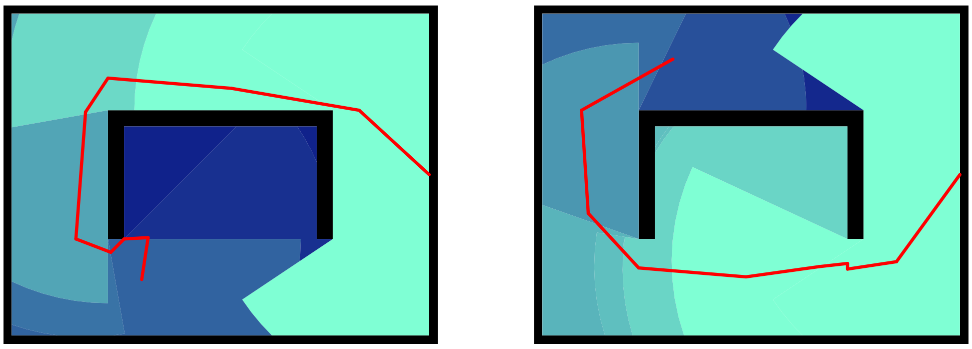

(a) : Trajectory τ_1. **(b) :** Trajectory τ_2.

Figure 2.1: Motivational example.

In order to express the probability density function, we define the *visibility region*. It is a region that the robot senses from some point $\mathbf{x}$ and is denoted

$\mathcal{V}(\mathbf{x})$, formally it is a set of points:

$$\mathcal{V}(\mathbf{x}) = \{\mathbf{y} \in \mathcal{W} | \overline{\mathbf{xy}} \subseteq \mathcal{W} \wedge ||\mathbf{xy}|| \leq r_{\mathcal{V}}\} \tag{2.3}$$

In other words, it is a set of points to which we can construct a straight line from the point $\mathbf{x}$ that does not intersect with any obstacles or environment border and its distance is less than maximal sensing distance of the sensor $r_{\mathcal{V}}$.

The robot moves along the trajectory and performs sensing in discrete time steps, it gathers information about the points that belong to the visibility region associated to its current location. If the object was first seen in time t_k implies that the object is located in the current visibility region but not in any prior visibility region of sensing points along the trajectory. We denote the region $\mathcal{W}^{\text{new}}_{t_k|\tau}$ and define it as follows:

$$\mathcal{W}^{\text{new}}_{t_k|\tau} = \mathcal{V}(\tau(t_k)) \setminus \bigcup_{i=0}^{k-1} \mathcal{V}(\tau(t_i)) \tag{2.4}$$

For more compact notation, we use label $\mathcal{W}^{\text{seen}}_{t_k|\tau} = \bigcup_{i=0}^{k} \mathcal{V}(\tau(t_i))$ to denote the set that has been seen up the time step t_k. We can also express it recursively as:

$$\mathcal{W}^{\text{seen}}_{t_k|\tau} = \mathcal{V}(\tau(t_k)) \cup \mathcal{W}^{\text{seen}}_{t_{k-1}|\tau} = \mathcal{W}^{\text{new}}_{t_k|\tau} \cup \mathcal{W}^{\text{seen}}_{t_{k-1}|\tau} \tag{2.5}$$

As stated earlier, probability of an object being discovered at time step t_k is proportional to the area of newly discovered region of the environment. Because it resides somewhere in the environment, we normalize by the total area.

$$f_{T|\tau}(t_k) = \frac{|\mathcal{W}^{\text{new}}_{t_k|\tau}|}{|\mathcal{W}|} \tag{2.6}$$

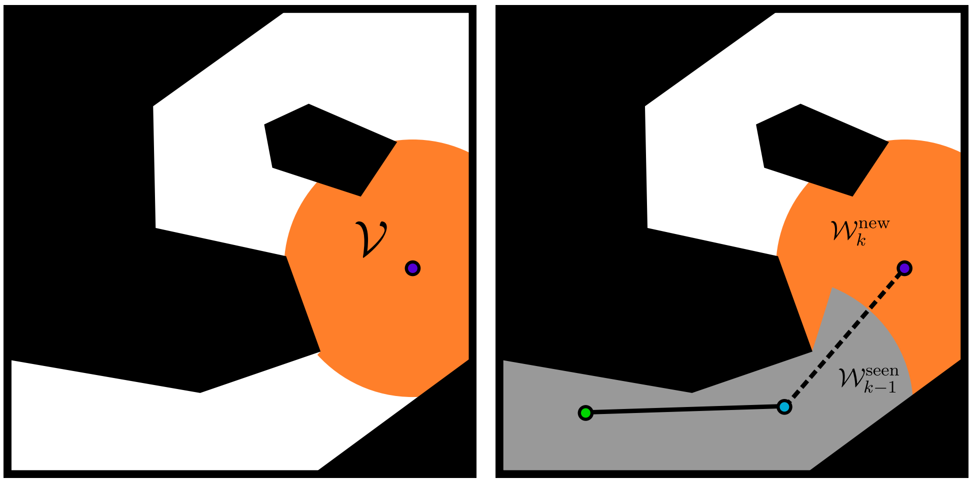
We can notice that the newly discovered regions are mutually disjoint, i.e. $\mathcal{W}^{\text{new}}_{t_i|\tau} \cap \mathcal{W}^{\text{new}}_{t_j|\tau} = \emptyset$ for $i \neq j$ and that $\mathcal{W}^{\text{seen}}_{t_m|\tau} = \mathcal{W}^{\text{new}}_{t_0|\tau} \cup \cdots \cup \mathcal{W}^{\text{new}}_{t_m|\tau}$. Using the properties of mutually disjoint sets we can evaluate the area of seen region as:

$$|\mathcal{W}^{\text{seen}}_{t_m|\tau}| = |\mathcal{W}^{\text{new}}_{t_0|\tau}| + \cdots + |\mathcal{W}^{\text{new}}_{t_m|\tau}| \tag{2.7}$$

By dividing both sides of the equation by $|\mathcal{W}|$, we obtain:

$$\frac{|\mathcal{W}^{\text{seen}}_{t_m|\tau}|}{|\mathcal{W}|} = \frac{|\mathcal{W}^{\text{new}}_{t_0|\tau}|}{|\mathcal{W}|} + \cdots + \frac{|\mathcal{W}^{\text{new}}_{t_m|\tau}|}{|\mathcal{W}|} \tag{2.8}$$

$$= f_{T|\tau}(t_0) + \cdots + f_{T|\tau}(t_m) = 1 \tag{2.9}$$


$\mathcal{V}$
$\mathcal{W}_k^{\text{new}}$
$\mathcal{W}_{k-1}^{\text{seen}}$

(a) : Illustration of a visibility region.

(b) : Ilustration of a newly discovered region.

Figure 2.2: Visual representation of the visibility region and a newly discovered region.

We can see that the property of probability density function $\sum_{i=0}^{m} f_{T|\tau}(t_0) = 1$ is in this case equivalent to the fact that the robot must visually cover the entire environment, i.e. $\mathcal{W}_{t_m|\tau}^{\text{seen}} = \mathcal{W}$.

Figure 2.2 shows a simple example explaining the described terms. The figure on the left shows a visibility region sensed from the blue point. The figure on the right shows the newly discovered region (orange) from the blue dark point. The gray area visualizes the region that was sensed, while following the trajectory, before reaching the dark blue point.

2.1 Related graph theory problems

The TSP is a problem defined on either directed or undirected weighted graph $G = (V, E)$. The objective is to find a cycle that visits all vertices $v \in V$ exactly once, returns to the start vertex and minimizes the cost of the cycle. The generalized variant of the problem extends TSP by grouping the vertices V into disjoint sets, also called clusters. A feasible solution of the GTSP is a cycle visiting all clusters exactly once. The objective is again to minimize the cost of the cycle.

The GSP is a task defined on a complete weighted graph $G = (V, E)$. Each vertex is assigned a weight $w(v)$ and each edge is assigned a cost $d(e)$. The graph is complete, which means that we can address any edge $e \in E$ connecting the vertices $v_i \in V$ and $v_j \in V$ directly by the tuple (v_i, v_j). One of the vertices $v_0 \in V$ is specified as a start depot. The objective is to find a path that starts in v_0, visits all the vertices $v \in V$ exactly once and minimizes the objective function. The objective function is defined as

$$c(\mathbf{x}) = \sum_{i=1}^{n} w(v_{x_i}) \sum_{j=1}^{i} d(v_{x_{j-1}}, v_{x_j}) \tag{2.10}$$

Where $\mathbf{x} = (v_{x_0}, \ldots, v_{x_n})$ defines the order of vertex visits. In [27], a variant specific for mobile search problem is defined. In this variant the weight w is not associated with a single vertex, but instead depends on the current vertex and also the previously visited vertices. The weight corresponds to the probability of detection of the object when visiting v.

2.2 Mobile Robot Search Problem as a graph theory problem

As stated before, there are two key requirements on a selection of a trajectory. First the trajectory must ensure that the object is found during its execution. Second the trajectory must minimize the expected time to find the object. The common approach is to decouple the problem by first generating a set of sampled sensing locations from the environment, and then selecting a sequence of those sensing location, which corresponds to a trajectory that begins in the start location and covers the whole environment while minimizing the expected time.

Papers that previously addressed this problem [27, 15, 31, 21] sample the environment with sensing locations $l_0, \ldots, l_n$ such that the union of visibility regions covers the whole environment, i.e.

$$\bigcup_{i=0}^{n} \mathcal{V}(l_i) = \mathcal{W} \tag{2.11}$$

Then they search for a solution that starts in the specified start location, without loss of generality we denote this location l_0, and visits all of the sensing locations while minimizing the expected time. The task to find the optimal order of visits can be viewed as the Graph Search Problem.

We address this problem by taking inspiration from the GTSP and the TSP with neighborhoods. These extensions of the classical TSP assume that the necessary information or reward is not associated only with one certain point, but rather with either a whole region or a set of points. This idea can be utilized in the mobile search problem.

We cover the environment with regions $\mathcal{R}_0, \ldots, \mathcal{R}_m$, such that

$$\bigcup_{i=0}^{m} \mathcal{R}_i = \mathcal{W} \tag{2.12}$$

Each region R_i is then associated with a cluster C_i. A cluster is assigned a set of points, that holds:

$$\mathcal{R}_i \subseteq \mathcal{V}(l), \forall l \in C_i \tag{2.13}$$

Under this condition, to fully cover the environment, it is sufficient to visit only one point from each cluster. Therefore, a feasible solution is a solution that begins in the start location and visits every cluster exactly once. This extension allows us to potentially explore solutions with shorter paths while exploring the same regions of the map.

Now we can formally define the problem as an instance of the Generalized GSP with order-dependent weights. We are given a complete weighted graph $G = (V, E)$. Vertex v_0 is defined as the start vertex. We associate the physical locations with the vertices and denote them $l_i = l(v_i)$. Since the graph is complete, i.e. every pair of distinct vertices is connected by exactly one unique edge, we can address the edge $e \in E$ connecting arbitrary pair of vertices $v_i \in V$ and $v_j \in V$ simply by the tuple (v_i, v_j). The edge cost

$$d(v_i, v_j) : V \times V \mapsto \mathbb{R}_0^+ \tag{2.14}$$

is equivalent to the shortest path from location $l(v_i)$ to $l(v_j)$. Unlike in the classical GSP a weight function is not only a property of a single vertex, but instead is defined as:

$$w(v, L) : V \times V^* \mapsto \mathbb{R}_0^+ \tag{2.15}$$

where V^* is the set of all possible finite sequences of vertices. The weight function represents the area seen from the location associated with the vertex v provided that locations linked with the sequence of vertices L has been already visited, i.e.

$$w(v, L) = |\mathcal{V}(l(v)) \setminus \bigcup_{u \in L} \mathcal{V}(l(u))| \tag{2.16}$$

A function

$$\delta(L) : V^* \mapsto \mathbb{R}_0^+ \tag{2.17}$$

corresponds to the length of a path that visits all the vertices from sequence L in given order, i.e. for a sequence $L = (v_{x_0}, \ldots, v_{x_\ell})$ we write

$$\delta(v_{x_0}, \ldots, v_{x_\ell}) = \sum_{i=1}^{\ell} d(v_{x_{i-1}}, v_{x_i}) \tag{2.18}$$

The vertices are partitioned into mutually disjoint sets $(V_0, \ldots, V_m)$ such that:

1. $v_0 \in V_0, |V_0| = 1$
2. $V_i \cap V_j = \emptyset$ for $i \neq j$
3. $V_0 \cup \cdots \cup V_m = V$

We also refer to these subsets as clusters. A feasible complete solution is a sequence of vertices $L = (v_{x_0}, \ldots, v_{x_m})$, which starts in the start depot, i.e. $v_{x_0} = v_0$ and the solution visits each cluster exactly once. Alternatively we can address the same solution by specifying only the sequence of indices $\mathbf{x} = (x_0, \ldots, x_m)$. The cost of a solution is computed as follows:

$$c(\mathbf{x}) = \sum_{i=1}^{m} w(v_{x_i}, (v_{x_0}, \ldots, v_{x_i-1}))\delta(v_{x_0}, \ldots, v_{x_i}) \tag{2.19}$$

For a more compact notation, we can denote $w_i = w(v_{x_i}, (v_{x_0}, \ldots, v_{x_i-1}))$ and $\delta_i = \delta(v_{x_0}, \ldots, v_{x_i})$. A feasible partial solution starts in the vertex v_0 and visits each cluster at most once.

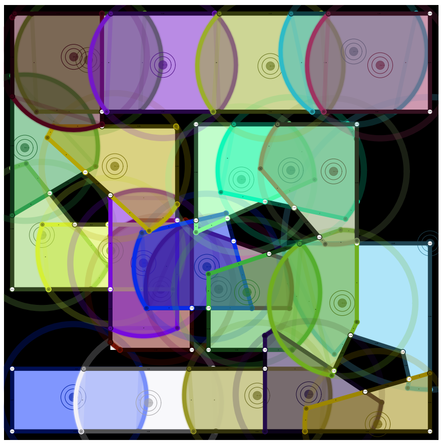
In our model, the robot moves with constant positive speed in arbitrary direction, therefore the time is proportional to the traveled distance, i.e. $t = \mu d$. The probability of discovering an object in a region is proportional to the area of the region, i.e. the weight. Finally we can compare the expected time to the cost function and obtain:

$$\mathbb{E}(T|\tau) = \sum_{i=0}^{m} t_i f_{T|\tau}(t_i) = \tag{2.20}$$

$$= \frac{\mu}{|\mathcal{W}|} \sum_{i=0}^{m} \delta(v_{x_0}, \ldots, v_{x_i}) w(v_{x_i}, (v_{x_0}, \ldots, v_{x_{i-1}})) = \frac{\mu}{|\mathcal{W}|} c(\mathbf{x}) \tag{2.21}$$

Because both the speed μ and the normalizing factor $|\mathcal{W}|$ are independent on the trajectory, they were factored out of the summation. The scalar $\frac{\mu}{|\mathcal{W}|}$ is a positive number and since multiplying a function by positive number does not change the location of a minimum, we see that the task of minimizing $c(\mathbf{x})$ is equivalent to minimizing expected time for given clusters and sensing locations.

Figure 2.3 shows a possible way of covering the environment. The figure on the left shows that map is completely covered with convex regions that were created from circles with radius equal to the half of the sensor range $r_{\mathcal{V}}$. In the right figure we show one of the polygons in violet. Sensing locations within one cluster are located on the border of the polygon. Because all of the sensing locations are located inside or on the border of a virtual circle with radius $r_{\mathcal{V}}/2$, it is certain that the distance from a sensing location to any point from this convex polygon does not exceed the sensing range $r_{\mathcal{V}}$. Because the polygon is convex, we also know that we can construct a line segment between a sensing location and any point in the convex polygon. Line segments lies entirely within the convex polygon. That means the line segment does not interfere with any obstacle. Altogether, we can see that this method of sampling leads fulfills the conditions, we required.


(a) : Coverage of the environment by convex polygons.

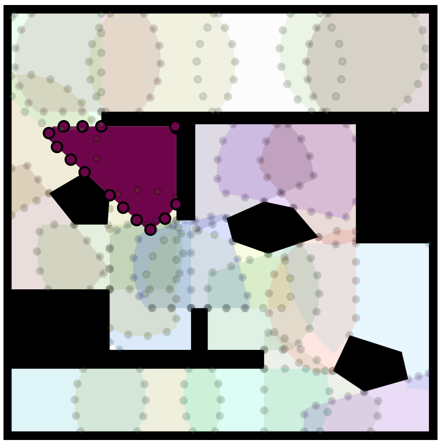
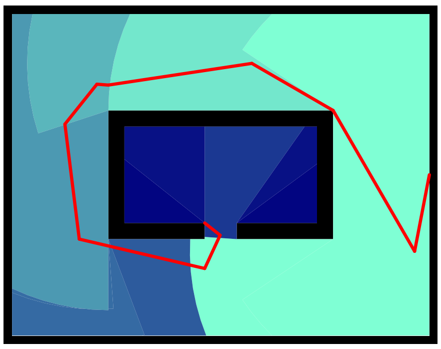

(b) : Sensing location on the border of a convex polygon.

Figure 2.3: Visualization of environment coverage.

Figure 2.4 provides a motivation on why to turn the problem into the Generalized GSP instead of the classical GSP. The figure shows two solutions. In order for a solution to be feasible, all clusters must be visited exactly once. Both displayed solutions visit the clusters in the same order, but choose different vertices. At each cluster both robots discover similar newly seen regions. However because the robot on the right followed more efficient trajectory τ_2, the final expected time is $\mathbb{E}(T|\tau_2) = 8.39$, compared to the trajectory on the left τ_1, with expected time $\mathbb{E}(T|\tau_1) = 9.8$.


(a) : Trajectory τ_1

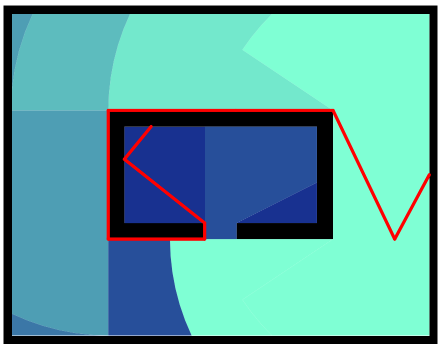

(b) : Trajectory τ_2

Figure 2.4: Comparison of two solutions visiting the clusters in the same order.

Chapter 3

Solution approach

In this chapter, first we describe the main ideas used for solving the problem. Then, we explain the functioning of the GLNS solver, introduced in [35]. Finally we go through all the changes that were done in order to transform the original solver to the solver that tackles mobile search problem.

3.1 Main concepts

Large neighborhood search is a optimization technique, originally proposed to solve vehicle routing problem [33, 34]. This method has some key differences to local search techniques. Local search explores the solution space by slightly modifying previous solution. This slight changes may lead to getting trapped in local minima, i.e. out of the allowed moves, there are none that would lead to improving the solution. This problem is tackled by minima escaping procedures (meta-heuristics), such as simulated annealing or tabu search. Large neighborhood search takes a different approach, instead of small steps it makes larger moves, that are more complex and stronger. Such moves allow the algorithm to reach broader solution space from any point. The size of the neighborhood can be increased when the search seems to stagnate. The basic concept of LNS is called relaxation and re-optimization. In the relaxation[1] step, a partition of a solution is damaged by removing certain solution blocks. In the re-optimization step, the solution is repaired by inserting permissible blocks into the solution, preferably in optimal manner.

[1]Also called destroy.

The original article proposes that selection of what parts to remove is based on relatedness measure. This measure is not strictly defined and depends on specific problem that is being solved, but the idea is that components that play a similar role within the solution are preferred to be selected for removal. Authors argue that if distant component were removed, in re-optimization part, they would likely end up in the same place in the solution as the original.

Adaptive Large Neighborhood Search [29], also originally developed for vehicle routing problems, extends Large Neighborhood Search (LNS) by introducing an adaptive layer to the search procedure. This layer controls the selection of destroy and repair neighborhoods. The selection is stochastic but favors neighborhoods, that yielded better results in the past. Introducing the randomness into the selection of destroy and repair operators causes more diversification and may possibly avoid stagnation in the search. Apart from generating more diverse solutions, the greatest strength is robustness and ability to adapt to specific problems. Thanks to this properties the need for calibration to specific problems is minimal.

Another concept used by GLNS is called Simulated Annealing. It is a optimization technique used for combinatorial problems that exploits an analogy to how metal cools and freezes into minimum energy structure. Compared to greedy search algorithms, SA may accept even the moves that do not improve the solution. The main control parameter is the temperature. The system temperature decreases over time and influences which solutions are explored and which are selected. A non-improving move is selected with a probability given by the following equation:

$$p = e^{-\frac{dc}{\mathcal{T}}} \tag{3.1}$$

where dc is the difference between the cost of trial generated by the move and the original cost. $\mathcal{T}$ is the current temperature of the system and is always a positive number, i.e. $\mathcal{T} > 0$. We consider only non-improving moves, therefore $dc \geq 0$. The formula for computing the probability shows that worse solutions are generally less likely to be accepted and that the overall probability of accepting a non-improving move decreases with decrease of temperature. Important concept in SA is the selection of annealing schedule. It consists of selecting the appropriate initial temperature and the rules for decreasing the system temperature. There are two common strategies for decreasing the temperature - exponential and linear. In the exponential case, every iteration the temperature is multiplied by a coefficient $\alpha \in (0, 1)$. In the linear case, every iteration a constant is taken off of the temperature. The final step is selecting the terminal temperature - the parameter that defines when the search ends. When selecting the mentioned parameters it may be useful to utilize a knowledge about the magnitude of changes in cost that are produced by application of a move in the specific problem [2].

3.2 GLNS

GLNS utilizes the combination of Adaptive Large Neighborhood Search and Simulated Annealing along with other optimization methods for solving the GTSP. An input of the algorithm is a complete graph $G = (V, E)$ and partitioning of the vertices into clusters $P_V = \{V_1, \ldots, V_m\}$. Each edge is assigned a cost $d(e)$. The Algorithm 1 shows the high level overview of the algorithm. The procedure consists of several iterations, called trial, after each trial the weights assigned to removal and insertion heuristics are updated based on their score. In the beginning of each iteration an initial tour is created using construction heuristics, see line 2. The algorithm then enters inner loop, where first removal and insertion heuristics are selected based on the selection weights, see line 5. In the next steps, lines 6 to 9, the number of vertices to remove N_r is uniformly selected from given range, then N_r vertices are removed from the tour using removal heuristic. Finally the tour is repaired by application of insertion heuristic. The tour is then further optimized by local optimization techniques, see line 10. The new tour is then tested whether it is the so far best found solution in this iteration, if so it is stored in a memory. At line 14, acceptance criteria are tested, if successful it is selected as current solution. This inner loop is repeated until stopping criteria, explained later, are met. At the end of each outer loop the selection weights are updated based on performance of individual removal and insertion heuristics.

3.2.1 Insertion heuristics

Insertion heuristics work such that we receive a partial solution, then one by one vertices are added forming a feasible complete solution. The partial tour $T = (V_T, E_T)$ is a cycle in G such that each cluster is visited at most once. We denote clusters present in the partial tour as $P_T \subseteq P_V$. The insertion of a vertex can be divided into two steps, first a cluster is selected and then a vertex belonging to this cluster is inserted into the partial solution at the optimal position. The selection of the cluster to insert depends on the specific heuristics. GLNS implements commonly used insertion heuristics: *nearest*, *farthest*, *random* and *cheapest* [7]. Authors of the GLNS propose another insertion heuristic called *unified*. This insertion heuristic unifies the nearest, farthest and random methods into one. This heuristic optionally introduces randomness into selecting the clusters to be inserted. While keeping the preference to selecting either the nearest or farthest cluster. Experiments

Algorithm 1 GLNS(G, P_V)

```
Input: A GTSP instance $(G, P_V)$
Output: A GTSP tour on $G$
 1: for $i = 1$ to numTrials do
 2:     $T \leftarrow$ initialTour$(G, P_V)$
 3:     $T_{best,i} \leftarrow T$
 4:     repeat
 5:         Select a removal heuristic $R$ and insertion heuristic $I$ using the
            selection weights
 6:         Select the number of vertices to remove, $N_r$, uniformly randomly
            from $\{1, \ldots, N_{max}\}$
 7:         Create a copy of $T$ called $T_{new}$
 8:         Remove $N_r$ vertices from $T_{new}$ using $R$
 9:         For each of the $N_r$ sets not visited by $T_{new}$, insert a vertex into $T_{new}$
            using $I$
10:         Locally re-optimize $T_{new}$
11:         if $c(T_{new}) < c(T_{best,i})$ then
12:             $T_{best,i} \leftarrow T_{new}$
13:         end if
14:         if accept$(T_{new}, T)$ then
15:             $T \leftarrow T_{new}$
16:         end if
17:         Record improvement made by $R$ and $I$
18:     until stop criterion is met
19:     Update selection weights based on improvements of each heuristic over
        trial
20: end for
21: return tour $T_{best,i}$ that attains $\min_i c(T_{best,i})$
```

show that the unified heuristic performs better that the original heuristics alone [35].

An input of the insertion heuristics, described in Algorithm 2, is a partial solution $T = (V_T, E_T)$ First we explain the selection of cluster, in Algorithm 2 at line 1. There are three methods that utilize the distance of a tour to the cluster. A distance to a cluster is computed:

$$\text{dist}(u, V_i) = \min_{v \in V_i}(\min(d(v, u), d(u, v))) \tag{3.2}$$

That means that it is the distance between vertex u and vertex $v \in V_i$ that has the minimal edge cost. In case of asymmetric graph, we consider the smaller edge cost. In the four mentioned heuristics the clusters are selected as follows:

Algorithm 2 Framework of insertion heuristics.

```
Input: A GTSP instance $(G, P_V)$ and a partial tour $T = (V_T, E_T)$ on $G$
Output: An updated partial tour $T$ that visits one additional set.
1: Pick a set $V_i$ in $P_V \setminus P_T$.
2: Find an edge $(x, y) \in E_T$ and vertex $v \in V_i$ that $d(x, v) + d(v, y) - d(x, y)$.
3: Delete the edge $(x, y)$ from $E_T$, add the edges $(x, v)$ and $(v, y)$ to $E_T$, and
   add $v$ to $V_T$.
4: return $T$
```

1. Nearest insert: A cluster V_s that minimizes the distance to the nearest vertex on the partial tour is selected, i.e.

$$V_s = \operatorname*{argmin}_{V_i \in P_V \setminus P_T} \min_{u \in V_T} \operatorname{dist}(u, V_i) \tag{3.3}$$

2. Farthest insert: A cluster V_s that maximizes the distance to the nearest vertex on the partial tour is selected, i.e.

$$V_s = \operatorname*{argmax}_{V_i \in P_V \setminus P_T} \min_{u \in V_T} \operatorname{dist}(u, V_i) \tag{3.4}$$

3. Random insert: A cluster V_s is randomly, with uniform distribution, selected out of the sets that are not present in the partial solution.

4. Cheapest insert: A cluster V_s that contains a vertex v which minimizes the insertion cost is selected, i.e.

$$V_s = \operatorname*{argmin}_{V_i \in P_V \setminus P_T} \min_{v \in V_i, (x,y) \in E_T} d(x, v) + d(v, y) - d(x, y) \tag{3.5}$$

The first three cases can be covered under a general *insertion unified.*

Algorithm 3 Set selection for the unified insertion heuristic.

```
Input: A GTSP instance $(G, P_V)$, a partial tour $T = (V_T, E_T)$ of $G$, and
   $\lambda \in [0, \infty)$
Output: A set $V_i \in P_V \setminus P_T$
1: Randomly select $k \in \{1, \ldots, \ell\}$ according to the unnormalized probability
   mass function $[\lambda^0, \lambda^1, \ldots, \lambda^{\ell-1}]$
2: Pick the set $V_i \in P_V \setminus P_T$ with the $k$th smallest distance $\operatorname{dist}_i$ to the tour
3: return $V_i$
```

Algorithm 3 describes the the procedure of selecting the set using unified heuristic. At line 1, an integer k is randomly selected from ℓ options, where ℓ

is the number of sets that are not present in the partial tour. The probability of selecting k is defined by unnormalized mass function $[\lambda^0, \lambda^1, \ldots, \lambda^{\ell-1}]$. The set $V_i \in P_V \setminus P_T$ with kth smallest distance to the tour is selected. There are three edge cases:

1. $\lambda = 0$, in this case, assuming 0^0 is defined as 1, only the first item in the mass function is non-zero, therefore this case corresponds to the nearest insertion.
2. $\lambda = 1$, in this case all of the items in the mass function are equal to 1 which yields the random insertion.
3. $\lambda = \infty$ corresponds to the farthest insertion.

For intermediate values $\lambda \in (0, 1)$, unified insertion heuristic offers a trade-off between selecting the set closest to the partial tour and uniform random selection. Similarly, in the range $\lambda \in (1, \infty)$ the sets farther from the partial tour are selected with higher probability. Experiments show that the intermediate values perform better compared to the edge cases [35].

The next step, line 2 in the Algorithm 2, is to find an edge (x, y) that will be replaced by a vertex $v \in V_s$ and will replace the insertion cost, i.e. $d(x, v) + d(v, y) - d(x, y)$. The algorithm introduces randomness by additive noise added to the insertion cost. A random sample $r_{v,x,y}$ for each combination of (v, x, y) is generated from uniform distribution $[0, \eta]$. The combination that minimizes $(1 + r_{v,x,y})(d(x, v) + d(v, y) - d(x, y))$ is then selected. This serves as another tool for diversification of the search. After selecting the edge $(x, y) \in E_T$ and the vertex $v \in V_s$ the final step is to remove the edge and insert edges (x, v) and (v, y).

3.2.2 Removal heuristics

Removal heuristics work such that N_r vertices are removed from a given complete solution. GLNS utilizes *worst*, *random*, *distance* and *segment* removal heuristics. The worst removal and random are unified under *unified* removal heuristic.

Algorithm 4 describes one iteration of the general removal heuristic framework. The procedure starts by selecting a random integer k from the probability mass function, defined by parameter λ, see line 1. Next, at line 2, vertex v_j

corresponding to the kth smallest value r_j is selected. In the unified removal heuristics the value r_j is the cost of removing the corresponding vertex. This framework is also used in the distance removal, where r_j corresponds to the distance. Finally, the selected vertex is removed. We apply this procedure until N_r vertices are removed from the original tour.

Algorithm 4 Removal heuristic framework for a given λ and distance metric r_j.

```
Input: A partial tour $T = (V_T, E_T)$, $\lambda \in [0, \infty)$, and values $r_j$ for each $v_j \in V_T$
Output: A new tour with one vertex removed from $V_T$
1: Randomly select $k \in \{1, \dots, \ell\}$ according to the unnormalized probability mass function $[\lambda^0, \lambda^1, \dots, \lambda^{\ell-1}]$, where $\ell = |V_T|$
2: Pick the vertex $v_j \in V_T$ with the $k$th smallest value $r_j$
3: Remove $v_j$ from $V_T$
4: Remove $(v_{j-1}, v_j)$ and $(v_j, v_{j+1})$ from $E_T$ and add $(v_{j-1}, v_{j+1})$ to $E_T$
5: return $T$
```

The worst removal method can be combined with the random removal into a unified removal, which selects the vertex to be removed. Based on the parameter λ we can decide whether the removal is more towards uniform randomness or selection of the maximal value. When $\lambda = 1$ the unified removal becomes the random removal and for $\lambda = \infty$ it becomes the worst removal.

The next removal insertion is called *distance removal.* The idea is to remove the vertices that are close to each other. It utilizes the idea of relatedness introduced in the original paper about LNS [33], which suggests removing the vertices that are in some aspect similar. In this case, the vertices with low cost of edges connecting the already removed vertices are preferably removed. The removal starts by randomly selecting a vertex from the tour and inserting it in a set of removed vertices V_{removed}. Then in each iteration, the distances r_j are computed as follows:

1. Vertex v_{seed} is randomly selected from the set V_{removed}.
2. For each vertex $v_j \in V_T$ in the tour compute distance

$$r_j = \min(d(v_j, v_{\text{seed}}), d(v_{\text{seed}}, v_j))) \tag{3.6}$$

The values $r_1, \dots, r_\ell$, where $\ell = |V_T|$, are then inputed to the Algorithm 4. Returned vertex is removed from the tour and the procedure is repeated until N_r vertices are removed.

The last implemented removal is called *segment removal.* A vertex from the tour is randomly selected, then N_r following vertices within the tour are selected, i.e. we remove the vertices $v_j, \dots, v_{j+N_r-1}$. If the index of the vertex exceeds the length of the tour, we assume that the indexing wraps around. Motivation of this heuristic is to escape deep local minima by destroying large solution segments.

3.2.3 Initial tour construction

There are two methods used for constructing the initial tour. *Random insertion tour* is a construction method that begins by selecting a vertex and adding it into an empty tour. In each iteration a unified insertion with $\lambda = 1$ is performed, i.e. a random cluster is selected uniformly and then a vertex $v \in V_s$ from this cluster is inserted so that it minimizes the randomized insertion cost:

$$c_{\text{insert}}(v, x, y) = (1 + r_{v,x,y})(d(x, v) + d(v, y) - d(x, y)) \tag{3.7}$$

This is repeated until a complete tour is formed, that is in total $m - 1$ iterations.

The second method, called *Random tour*, starts by selecting a random ordering of the clusters, alternatively, we can say that the clusters are shuffled, then from each cluster a random vertex is selected. Finally the selected vertices are connected forming a complete tour.

It was experimentally shown that the selection of the initial tour construction does not significantly affect the final solution, but the random insertion tour typically converges faster, therefore it seems to be better in cases where time is highly limited [35].

3.2.4 Local optimization

After each iteration of GLNS and before evaluation of the acceptance conditions the tour is attempted to further optimize by applying local optimization techniques that are cyclically repeated for as long as an improvements are found.

Re-optimize vertex in each set. First the order of sets is fixed in accordance to the order of given tour. From each set a vertex is selected so that when the vertices are connected into a tour the cost is minimal. This problem can be solved using graph traversal algorithm such as *breadth-first search* or *depth-first search*. However, this problem can be tackled using *dynamic programming* as was shown in [13]. Which especially for larger problems can significantly reduce the search time.

Move-opt. This operator attempts to optimize the ordering of the cluster. A cluster is randomly uniformly selected from a complete tour. From this cluster a vertex that minimizes the insertion cost is selected and placed to the best such location. This process is repeated N_{move} times. This optimization technique can be viewed as an instance of GLNS where $N_r = 1$.

3.2.5 Acceptance and stopping criteria

As acceptance criterion the standard SA acceptance criterion is used: if the T_{new} improves the solution it is always accepted, if not the probability of acceptance is given by:

$$p = \exp(\frac{c(T) - c(T_{\text{new}})}{\mathcal{T}}) \tag{3.8}$$

For cooling scheme the exponential cooling is used, which decreases the temperature by multiplying it by some scalar γ. The temperature is decreased every iteration of the inner loop. The temperature is reinitialized in the beginning of every outer loop (trial).

Every trial consists of the *initial descent* and several *warm restarts*. In the initial descent phase, if there are no improvements found for certain number of consecutive iterations, the trial switches to the warm restarts phase. In the beginning of each warm restart the temperature is increased, in order to allow for more diversification in the search. Each warm restart ends, when no improvement is found for several consecutive iterations. For more information about the cooling scheme, selection of parameters, and implementation refer to [35].

3.2.6 Choosing insertion and removal heuristics

Both removal and insertion heuristics are selected based on the assigned score. At each iteration of the inner loop, the score for both removal and insertion is evaluated as

$$s = \max(\frac{c(T) - c(T_{\text{new}})}{c(T)}, 0) \tag{3.9}$$

Disallowing negative score is used so that when no improvement is found no penalty is applied. The scores are stored for the specific heuristic and at the end of each trial an they are averaged and the global score is updated as a weighted sum of the original score times ε and the new score times $(1 - \varepsilon)$. The solver contains a bank of insertion heuristics, i.e. cheapest heuristics and unified heuristics along with some value λ and η. Also a bank of removal heuristics, i.e. distance and worst removal along with a value λ, and a the segment removal, which is parameter-less.

3.3 Algorithm modifications

In this section we discuss the modifications to the GLNS solver in order to adjust it to the mobile search problem. We consider two cases - *static weights* and *dynamic weights*. Static weights correspond to solving the MRSP as the GSP, and dynamic weights to the GSP with order-dependent weights. TDP, a special case of GSP, is included in the static weights.

3.3.1 Insertion heuristics

We have to consider that the cost function is different to GTSP, therefore, we need to change the parts of the algorithm where the cost or difference in cost is evaluated. In case of GTSP the difference can be computed by simply adding the cost of edges that connect the inserted vertex to the tour and subtracting the cost of edge that was removed, i.e. $d(x, v) + d(v, y) - d(x, y)$. In case of GSP the calculation is more complex. First the insertion of vertex does affect the contribution of all the following vertices within the solution. The static weights are not affected by their order, but by adding a vertex before, the path length δ is modified. Moreover, in case of dynamic weights, insertion of a vertex may modify the regions discovered by the following vertices and therefore also modify their corresponding weights. One way would be to evaluate the cost before and after the insertion and then to simply find their difference. However, because the evaluation of a solution may be very costly, we use methods that allow to evaluate the difference faster and are presented in the following text.

Assume we are given a partial solution $\mathbf{x} = (x_0, \ldots, x_\ell)$. For more compact notation for a given solution we assume $w(v_{x_i}) = w_i$, $d(v_{x_{i-1}}, v_{x_i}) = d_i$ and $\delta(v_{x_0}, \ldots, v_{x_i}) = \delta_i$. The cost of the original solution is then computed as:

$$c(\mathbf{x}) = \sum_{i=1}^{\ell} w_i \delta_i \tag{3.10}$$

Consider we want to insert a vertex $\widetilde{v}$ with weight $\widetilde{w} = w(\widetilde{v})$ at position k, i.e. between vertices $v_{x_{k-1}}$ and v_{x_k}. Let us denote edge costs $L_{k,1} = d(v_{x_{k-1}}, \widetilde{v})$ and $L_{k,2} = (\widetilde{v}, v_{x_{k-1}})$. The cost of modified solution can be then computed as

$$\begin{aligned} c(\widetilde{\mathbf{x}}) = {} & w_1\delta_1 + \ldots + w_{k-1}\delta_{k-1} + \\ & + \widetilde{w}(\delta_k + L_{k,1}) + \\ & + w_k(\delta_k + L_{k,1} + L_{k,2} - d_k) + \ldots w_\ell(\delta_\ell + L_{k,1} + L_{k,2} - d_k) \end{aligned} \tag{3.11}$$

We can rearrange the equation to:

$$c(\widetilde{\mathbf{x}}) = \sum_{i=1}^{\ell} w_i\delta_i + \widetilde{w}(\delta_k + L_{k,1}) + (L_{k,1} + L_{k,2} - d_k)\sum_{i=k}^{\ell} w_i \tag{3.12}$$

As you can see, the first term is equal to the original, therefore we do not have to evaluate the whole equation but only the difference to the original cost. The second term is only a constant number of operations, but the last term contains summation of $\ell - k + 1$ elements, while this number is less then ℓ, it is still proportional to the number of clusters m. Therefore this modification would not reduce the overall complexity. However, because the term does not depend on the vertex $\widetilde{v}$, when insertion of more vertices is evaluated, the computation of $\sum_{i=k}^{\ell} w_i$ can be done only once and be stored in a memory. Moreover it can be computed for all possible values of k in $O(\ell) = O(m)$. For evaluation of $O(n)$ vertices, we obtain complexity $O(nm)$, compared to $O(nm^2)$, which would be necessary if we evaluated $O(n)$ vertices inserted at $O(m)$ possible positions and each solution evaluated in $O(m)$.

The evaluation of inserting certain vertex at all possible positions can be done using the formula:

$$c(\widetilde{\mathbf{x}}_k) = c(\mathbf{x}) + \widetilde{w}(\delta_k + L_{k,1}) + (L_{k,1} + L_{k,2} - d_k)S_k \tag{3.13}$$

Where S_k is variable that can be precomputed. We define it recursively as $S_{k-1} = S_k + w_{k-1}$, where $S_\ell = w_\ell$.

Figure 3.1 shows a schematic illustration of inserting a vertex in static weights case. We can see that by inserting a vertex, the path length of vertices followed by the inserted vertex may change.

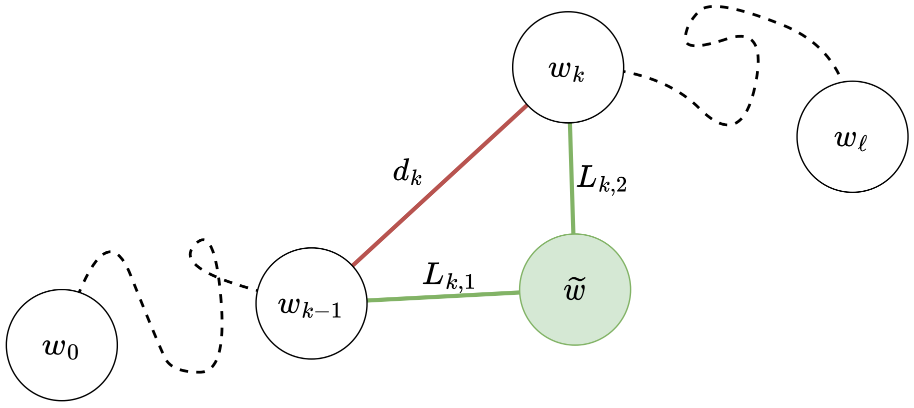


Figure 3.1: Illustration of insertion.

The variant with dynamic weights is even more complicated, because the inserted vertex does not only influence by changing the path lengths to individual vertices but also can change the weights of the following vertices. Let us assume a partial solution $\mathbf{x} = (x_0, \ldots, x_\ell)$. For more compact notation we denote $w_i := w(v_{x_i}, (v_{x_0}, \ldots, v_{x_{\ell-1}}))$, $d_i := d(v_{x_{i-1}}, v_{x_i})$ and $\delta_i := \delta(v_{x_0}, \ldots, v_{x_i})$. The cost of a partial solution is then computed as

$$c(\mathbf{x}) = \sum_{i=1}^{\ell} w_i \delta_i \tag{3.14}$$

For even more compact notation we denote, for fixed partial solution

$$A_k = \mathcal{V}(l(v_{x_k})) \qquad \alpha_k = \bigcup_{i=0}^{k} \mathcal{V}(l(v_{x_k})) \tag{3.15}$$

We denote the original solution extended by inserting the $\widetilde{v}$ to the kth position as $\widetilde{\mathbf{x}}_k = (x_0, \ldots, x_{k-1}, \widetilde{x}, x_k, \ldots, x_\ell)$. Let us start with simplest case of inserting vertex $\widetilde{v}$, with associated visibility region $\mathcal{V}(l(\widetilde{v}))$, to the end of the solution.

$$c(\widetilde{\mathbf{x}}_{\ell+1}) = \sum_{i=1}^{\ell} |A_i \setminus \alpha_{i-1}|\delta_i + |B \setminus \alpha_\ell|(\delta_\ell + L_{\ell+1,1}) \tag{3.16}$$

For other cases when $k \leq n$, the modified cost is

$$\begin{aligned} c(\widetilde{\mathbf{x}}_k) = &\sum_{i=1}^{k-1} |A_i \setminus \alpha_{i-1}|\delta_i + \\ &+ |B \setminus \alpha_{k-1}|(\delta_{k-1} + L_{k,1}) + \\ &+ \sum_{i=k}^{\ell} |A_i \setminus (B \cup \alpha_{i-1})|(\delta_i + L_{k,1} + L_{k,2} - d_k) \end{aligned} \tag{3.17}$$

Unfortunately unlike in the case of static weights, we cannot perform the same simplification. We can, however, subtract the costs of solutions with insertion of the same vertex at adjacent positions k and $k+1$. We obtain

$$\begin{aligned}\Delta = {} & |B \setminus \alpha_{k-1}|(\delta_{k-1} + L_{k,1}) - \\ & - |A_k \setminus \alpha_{k-1}|\delta_k + \\ & + |A_k \setminus (B \cup \alpha_{k-1})|(\delta_k - d_k + L_{k,1} + L_{k,2}) - \\ & - |B \setminus \alpha_k|(\delta_k + L_{k+1,1}) + \\ & + \sum_{i=k+1}^{\ell} |A_i \setminus (B \cup \alpha_{i-1})|(L_{k,1} + L_{k,2} - d_k - L_{k+1,1} - L_{k+1,2} + d_{k+1})\end{aligned} \tag{3.18}$$

In order to examine the time complexity, we need to know the time complexity to perform set operations and to find a size of the set. Because we always use set operations together with the cardinality operation, we can denote the complexity of performing both operations as $O(f)$. The second term can be precalculated for the whole neighborhood in $O(fm)$. First, third and fourth terms need to be calculated for each combination of edge positions and inserted vertex, for fixed inserted vertex it is $\ell + 1 \propto$ [2]m. The last term is a summation of $\ell - k \propto m$ elements. This summation for any k can be evaluated given inserted vertex $\widetilde{v}$ in $\ell \propto m$ steps. When the steps are combined, for a fixed inserted vertex it is complexity $O(fm)$. For the whole neighborhood with number of candidates $\propto n$, we obtain total complexity $O(fmn)$. After preprocessing, Δ can be evaluated using constant number of mathematical operations. Cost of all modified solution $c(\widetilde{\mathbf{x}}_k)$ can be evaluated by first calculating the cost of the original solution extended by inserting $\widetilde{v}$ at the end, and then repeatedly evaluating and adding the Δ, for each k.

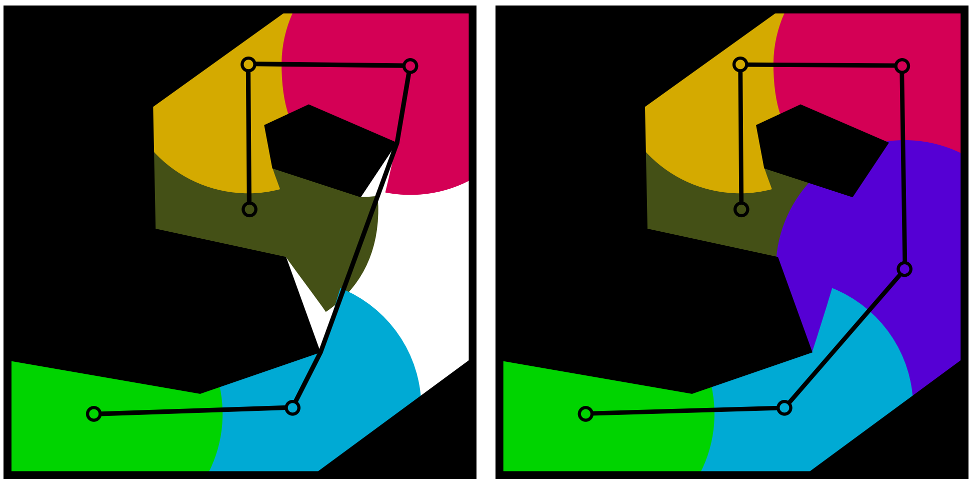
In order to evaluate the cost of inserting a certain vertex at all possible positions, we need to first compute the cost of inserting the vertex at the last position, i.e. $c(\widetilde{\mathbf{x}}_{\ell+1})$. The Δ is equal to the difference in the cost of inserting the vertex at two adjacent positions, i.e. $\Delta = c(\widetilde{\mathbf{x}}_k) - c(\widetilde{\mathbf{x}}_{k+1})$. By repeatedly decreasing k by one, we can evaluate the cost of inserting at all positions, by computing:

$$c(\widetilde{\mathbf{x}}_k) = \Delta + c(\widetilde{\mathbf{x}}_{k+1}) \tag{3.19}$$

where Δ has to be recalculated for every k and $k+1$.

Figure 3.2 shows an illustration of inserting a vertex in the dynamic weights case. Compared to the static weights not only the path lengths of the vertices following the inserted vertex change, but also their associated weight may change. In this example, it is obvious that by inserting the dark blue vertex, the weight of red and dark green vertices changed too.

[2]$\propto$ is a symbol to denote proportional to.



(a) : A partial solution.

(b) : A solution after inserting dark blue vertex.

Figure 3.2: Effect of inserting a vertex in the dynamic weights scenario.

3.3.2 Removal heuristics

The computation of removal cost gets more complicated when using different cost function. In order to decrease the computation complexity, we present formulas that allow us to reduce the time complexity of evaluation of the removal costs. Here it seems only the Generalized GSP with order-independent weights can be simplified to reduce the time complexity.

Assume we are given a solution $\mathbf{x} = (x_0, \ldots, x_\ell)$. Let us denote $w(v_{x_i}) = w_i$, $d(v_{x_{i-1}}, v_{x_i}) = d_i$ and $\delta(v_{x_0}, \ldots, v_{x_i}) = \delta_i$. The cost is then calculated as

$$c(\mathbf{x}) = \sum_{i=1}^{\ell} w_i \delta_i \tag{3.20}$$

A solution, where the vertex at kth position, i.e. vertex v_{x_k}, was removed solution from the original solution $\mathbf{x}$ is denoted as $\tilde{\mathbf{x}}_k$. Its cost is then computed as:

$$c(\tilde{\mathbf{x}}_k) = \sum_{i=1}^{k-1} w_i \delta_i + \sum_{i=k+1}^{\ell} w_i(\delta_i + L_k - d_k - d_{k-1}) \tag{3.21}$$

Where L_k is the edge cost between the vertices adjacent to the removed vertex, i.e. the vertices $v_{x_{k-1}}$ and $v_{x_{k+1}}$. The formula can be rearranged to the following form:

$$c(\tilde{\mathbf{x}}_k) = \sum_{i=1}^{\ell} w_i \delta_i - w_k \delta_k + (L_k - d_k - d_{k+1}) \sum_{i=k+1}^{\ell} w_i \tag{3.22}$$

It is apparent that the first term is equal to the original cost, the second term is the contribution of the vertex v_{x_k} to the original cost. The last term is a summation of $\ell - k \propto m$ items, which can be precomputed in $O(m)$ as $S_k = S_{k+1} + w_k$, where $S_\ell = w_\ell$. The cost of removal one vertex can is then only a constant number of mathematical operations and is computed as follows:

$$c(\tilde{\mathbf{x}}_k) = c(\mathbf{x}) - w_k\delta_k + (L_k - d_k - d_{k+1})S_{k+1} \tag{3.23}$$

To evaluate the removal for all suitable vertices, we obtain complexity $O(m)$.

Figure 3.3 shows a schematic of removing a vertex from a solution.

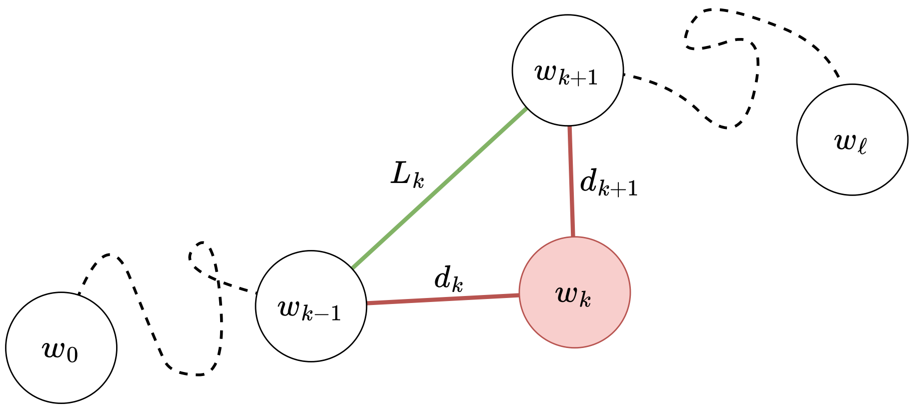


Figure 3.3: Illustration of a vertex removal.

3.3.3 Re-Optimize

In the original Re-Optimize method the optimal solution is found through dynamic programming. Input to this procedure is a complete solution. The idea is that we do not change cluster ordering, but only select a sequence of vertices which minimizes the cost. The original algorithm starts by selecting an arbitrary start cluster, then for each vertex v belonging to the start cluster, we find a minimal tour that starts in the vertex v, traverses over all the cluster in predetermined order and finishes in the same vertex v. For each vertex from the start cluster we obtain a path, finally we simply select the cheapest one as the final solution.

In the Generalized GSP the start vertex is fixed, therefore we need to evaluate minimal tours from the start vertex to all of the vertices in the final cluster, while respecting the cluster ordering. The more important difference lies in the dissimilar cost functions, because direct application of the original procedure does not lead to optimal solution, as it will be shown later.

Figure 3.4 shows a schematic illustration of the Algorithm 5.

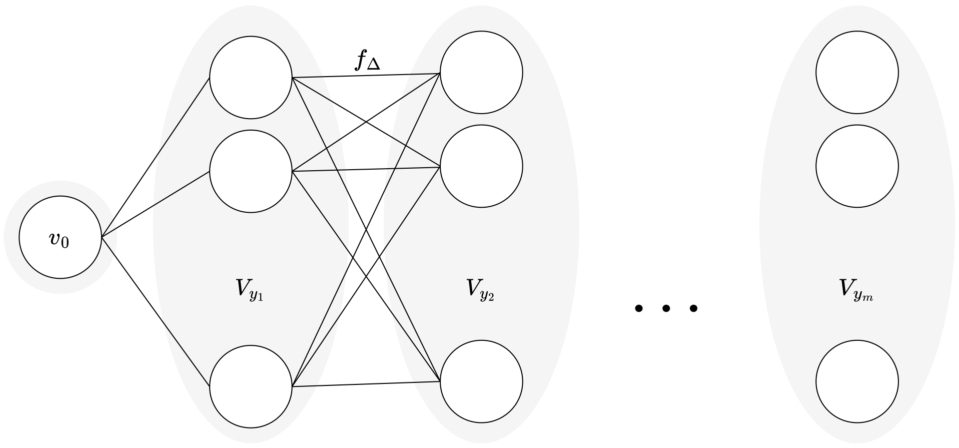


Figure 3.4: Illustration of Re-Optimize using dynamic programming.

The algorithm is described in Algorithm 5. It starts by initialization, f_{min} is a list storing the minimal found cost to each vertex. In the beginning we do not known any path to the vertices except for the start vertex v_0. Initialization is described at lines 1 and 2. At line 3 memory is initialized, this variable is specific for each cost function and its implementation is described later in the corresponding section. The rest of algorithm contains three loops, the most outer one iterates over the clusters in the specified order $(V_{y_0}, \ldots, V_{y_m})$. The two inner loops find all the combinations of vertices from adjacent clusters, described at lines 5 and 6. Inside the loop, we want to evaluate the cost of a path that reaches v_{to} by visiting v_{from}. At line 7, the cost is decomposed to the minimal cost to reach v_{from} and an addition of going from v_{from} to v_{to}. If this candidate path yields better solution, its cost is stored in f_{min} and the helper variables are stored in the memory. The concrete implementation of memory update, described at line 10, is specific to the cost function and is described in designated sections. Finally, the algorithm finishes by selecting the vertex $v \in V_{y_m}$ from the final cluster that has the minimal cost and returning the path that leads to this vertex, see line 13. The way the path is obtained depends on the implementation of the memory, either each vertex stores the previous vertex, then the path is obtained by backtracking back to the start vertex, or alternatively each vertex can store the whole partial solution.

In order to show that this pseudo-code yields the optimal solution for a given fixed order of clusters, we need to show, in accordance to the Bellman's

Algorithm 5 Re-Optimize

```
Input: $G = (V, E)$
Output: Shortest path over ordered clusters $(V_{y_0}, \ldots, V_{y_m})$
 1: $f_{\min}[v_0] \leftarrow 0$
 2: $f_{\min}[v] \leftarrow \infty, \forall v \in V \setminus v_0$
 3: $M[v] \leftarrow$ initializeMemory($v$), $\forall v \in V$
 4: for $i \in (y_1, \ldots, y_m)$ do
 5:   for all $v_{\text{from}} \in V_{i-1}$ do
 6:     for all $v_{\text{to}} \in V_i$ do
 7:       $z \leftarrow f_{\min}[v_{\text{from}}] + f_{\Delta}(v_{\text{from}}, v_{\text{to}})$
 8:       if $z < f_{\min}[v_{\text{to}}]$ then
 9:         $f_{\min}[v_{\text{to}}] \leftarrow z$
10:         $M[v_{\text{to}}] \leftarrow$ updateMemory($v_{\text{from}}, v_{\text{to}}, M$)
11:       end if
12:     end for
13:   end for
14: end for
15: return path of $\underset{v \in V_{y_m}}{\operatorname{argmin}} f_{\min}[v]$
```

principle of optimality, that if there is an optimal path from v_0 to v_{end} that crosses arbitrary vertex v_{mid} that there is not any path from v_0 to v_{mid} what would be cheaper than the path from v_0 to v_{mid} that is contained in the optimal path. If this condition is not fulfilled, the algorithm may select a different path to v_{mid} and the optimal solution may not be found. More formally, we can say that there is an optimal solution $\mathbf{x}^*$ that starts at vertex $v_{x_0} = v_0$, crosses $v_{\min} = v_k$ and ends at vertex $v_{x_m} = v_{\text{end}}$, this solution can be written as:

$$\mathbf{x}^* = (x_0, \ldots, x_{k-1}, x_k, \ldots, x_m) \tag{3.24}$$

Then there is a solution $\mathbf{x}'$ which also starts in $v_{x'_0} = v_0$ and visits $v_{x'_k} = v_{\text{mid}}$ and finishes at $v_{x'_m} = v_{\text{end}}$.

$$\mathbf{x}' = (x'_0, \ldots, x'_{k-1}, x'_k, \ldots, x'_m) \tag{3.25}$$

Up until vertex v_{mid} the paths are different but from $v_{\min}$ until the end of the solution, the paths are equal, i.e.

$$\mathbf{x}' = (x'_0, \ldots, x'_{k-1}, x_k, \ldots, x_m) \tag{3.26}$$

We denote the partial solutions, that is the first $k+1$ items of the solution as $\mathbf{x}^*_p = (x_0, \ldots, x_{k-1}, x_k)$ and $\mathbf{x}'_p = (x'_0, \ldots, x'_{k-1}, x_k)$.

Next, we examine the optimality of the dynamic programming approach for multiple variants of the graph problems. In order to use the dynamic programming, we need to define the cost of a partial solution $f(\mathbf{x}_p)$. We attempt to prove the optimality by contradiction. Our first assumption is that the solution $\mathbf{x}^*$ is optimal i.e.

$$f(\mathbf{x}^*) - f(\mathbf{x}) \leq 0, \forall \mathbf{x} \tag{3.27}$$

We also assume that the partial solution $\mathbf{x}'_p$ of the non-optimal solution $\mathbf{x}'$ minimizes of the partial cost function, i.e.

$$f(\mathbf{x}^*_p) - f(\mathbf{x}'_p) > 0 \tag{3.28}$$

Generalized Traveling Salesman Problem with fixed start

First we examine the simplest scenario - the GTSP. We need to define the cost function of the partial solution. For the cost function that is simply the length of the path, we define:

$$f(\mathbf{x}) = d(x_0, x_1) + \ldots + d(x_{\ell-1}, x_\ell) \tag{3.29}$$

For more compact notation we denote $d_i = d(x_{i-1}, x_i)$ and $d'_i = d(x'_{i-1}, x'_i)$.

First, we evaluate the cost of individual solutions:

$$f(\mathbf{x}^*) = d_1 + \ldots + d_k + d_{k+1} + \ldots + d_m \tag{3.30}$$
$$f(\mathbf{x}') = d'_1 + \ldots + d'_k + d_{k+1} + \ldots + d_m \tag{3.31}$$
$$f(\mathbf{x}^*_p) = d_1 + \ldots + d_k \tag{3.32}$$
$$f(\mathbf{x}'_p) = d'_1 + \ldots + d'_k \tag{3.33}$$

Notice that we replaced d'_i for d_i in terms where $d_i = d'_i$, i.e. $i \in \{k+1, \ldots, m\}$. Now we compare the costs of the complete solutions by finding their difference, we obtain

$$f(\mathbf{x}^*) - f(\mathbf{x}') = d_1 + \ldots + d_k - (d'_1 + \ldots + d'_k) = f(\mathbf{x}^*_p) - f(\mathbf{x}'_p) \tag{3.34}$$

By planting this result to the first assumptions (Equation 3.27) we obtain $f(\mathbf{x}^*_p) - f(\mathbf{x}) \leq 0$. This inequality is, however, apparently in contradiction with second assumption (Equation 3.28), this leads to contradiction. Therefore we have proved that the approach leads to optimal solution.

In the Algorithm 5, we need to define the addition to the cost f_Δ. It can be computed simply as the distance from one vertex to the other, i.e. $f_\Delta(v_{\text{from}}, v_{\text{to}}) = d(v_{\text{from}}, v_{\text{to}})$. Every time we found a better solution, we update the vertex from which we reached v_{to} this way we can backtrack the solution later. In the beginning we can initialize all the previous vertices to some invalid value. Apart from that we do not need to store any other helper variables.

Generalized Traveling Deliveryman Problem

Next we take a look at the Generalized TDP, the setup is the same except for the cost function, which we define as:

$$f(\mathbf{x}_p) = d_1 + \ldots + \sum_{i=1}^{\ell} d_i \tag{3.35}$$

Let us now evaluate the cost functions again

$$f(\mathbf{x}^*) = d_1 + \ldots + \sum_{i=1}^{k} d_i + \sum_{i=1}^{k+1} d_i + \ldots + \sum_{i=1}^{m} d_i \tag{3.36}$$

$$f(\mathbf{x}') = d_1' + \ldots + \sum_{i=1}^{k} d_i' + \left(\sum_{i=1}^{k} d_i' + d_{k+1}\right) + \ldots + \left(\sum_{i=1}^{k} d_i' + \sum_{i=k+1}^{m} d_i\right) \tag{3.37}$$

$$f(\mathbf{x}_p^*) = d_1 + \ldots + \sum_{i=1}^{k} d_i \tag{3.38}$$

$$f(\mathbf{x}_p') = d_1' + \ldots + \sum_{i=1}^{k} d_i' \tag{3.39}$$

Again we plant the results to the first assumption (Equation 3.27) and we obtain:

$$f(\mathbf{x}^*) - f(\mathbf{x}') = f(\mathbf{x}_p^*) - f(\mathbf{x}_p') + (m-k)\left(\sum_{i=1}^{k} d_i - \sum_{i=1}^{k} d_i'\right) \tag{3.40}$$

We can see that in this case we have not received a contradiction because it is not guaranteed that $(m-k)(\sum_{i=1}^{k} d_i - \sum_{i=1}^{k} d_i') \geq 0$, which would cause a contradiction. However, this result gave us an insight how to define a cost of partial solution that would yield optimal results in the algorithm.

Example 3.1. A problem shown in Figure 3.5 serves as a counter example, which shows that with cost of a partial solution defined $f(\mathbf{x}_p) = d_1 + \ldots + \sum_{i=1}^{\ell} d_i$, dynamic programming would not yield the optimal solution. The cost of transition from v_{from} to v_{to} is then defined as $f_\Delta(v_{\text{from}}, v_{\text{to}}, M) = \delta(v_{\text{from}}) + d(v_{\text{from}}, v_{\text{to}})$. Where $\delta(v_{\text{from}})$ is stored in the memory and is equal to the length of the path from the start vertex v_0. This variable is updated as $\delta(v_{\text{to}}) \leftarrow \delta(v_{\text{from}}) + d(v_{\text{from}}, v_{\text{to}})$. In this trivial example, there are only two possible solutions $\mathbf{x}_1 = (v_0, v_1, v_3, v_4)$ and $\mathbf{x}_2 = (v_0, v_2, v_3, v_4)$. Simply by planting the solutions to the definition of cost function for the Generalized TDP, we obtain $c(\mathbf{x}_1) = 10 + 30 + 31 = 71$ and $c(\mathbf{x}_2) = 20 + 21 + 22 = 63$. Therefore the optimal solution is $\mathbf{x}_2$, now we emulate the run of the algorithm to see if it reaches the same result. When the algorithm reaches vertex v_1 it

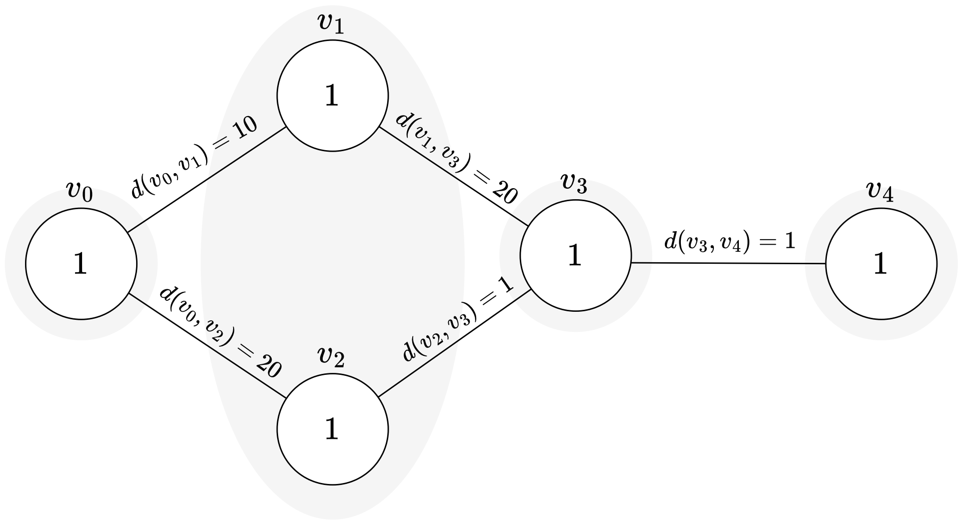


Figure 3.5: Re-Optimize counter example.

stores $f_{\min}[v_1] \leftarrow 10$ and $\delta(v_1) \leftarrow 10$, similarly for vertex v_2, $f_{\min}[v_2] \leftarrow 20$ and $\delta(v_2) \leftarrow 20$. Next the algorithm reaches vertex v_3, there are two ways how to reach this vertex. Therefore it must decide whether chose vertex v_1 with value $z_1 = f_{\min}[v_1] + (\delta(v_1) + d(v_1, v_3)) = 10 + (10 + 20) = 40$, or vertex v_2 with value $z_2 = f_{\min}[v_2] + (\delta(v_2) + d(v_2, v_3)) = 20 + 20 + 1 = 41$. The algorithm must chose the vertex with smaller z, so it chooses vertex v_1. From v_3 there is only one possible choice v_4. We see that the resulting solution is (v_0, v_1, v_3, v_4), that is the solution $\mathbf{x}_1$. However, we have shown that the optimal solution is $\mathbf{x}_2$, therefore we can see that this approach does not generally yield optimal solution.

Note that there is not a strict method how to define a cost of partial solution. The only requirement is that when applied on a complete solution, the value of cost function must be equal. In the previous example we implicitly assumed that the cost of a solution of length $\ell + 1$ is a summation of the first ℓ terms. We show that the cost can be rearranged:

$$c(\mathbf{x}) = d_1 + \ldots + \sum_{i=1}^{m} d_i = md_1 + (m-1)d_2 + \ldots + d_m \tag{3.41}$$

Therefore we define the cost of a partial solution as:

$$f(\mathbf{x}) = md_1 + (m-1)d_2 + \ldots + (m - \ell + 1)d_\ell \tag{3.42}$$

We proceed to the proof with the newly defined cost function, first the individual costs are evaluated

$$f(\mathbf{x}^*) = md_1 + \ldots + (m-k+1)d_k + (m-k)d_{k+1} + \ldots + d_m \tag{3.43}$$
$$f(\mathbf{x}') = md'_1 + \ldots + (m-k+1)d'_k + (m-k)d_{k+1} + \ldots + d_m \tag{3.44}$$
$$f(\mathbf{x}^*_p) = md_1 + \ldots + (m-k+1)d_k \tag{3.45}$$
$$f(\mathbf{x}'_p) = md'_1 + \ldots + (m-k+1)d'_k \tag{3.46}$$

We substitute the evaluated functions to the first assumption (Equation 3.27) and obtain:

$$f(\mathbf{x}^*) - f(\mathbf{x}') = f(\mathbf{x}^*_p) - f(\mathbf{x}'_p) \tag{3.47}$$

We result is $f(\mathbf{x}^*_p) - f(\mathbf{x}'_p) \leq 0$, which again contradicts with the second assumption. Therefore we have proven that using this cost function, the algorithm yields optimal solution. This gives us a hint on how to approach to constructing the cost function for partial solution in more general cases that TDP.

In order to implement this method to the Algorithm 5, we need define the increase in cost f_Δ by traveling from v_{from} to v_{to}. We define the function as:

$$f_\Delta(v_{\text{from}}, v_{\text{to}}, M) = (m - k(v_{\text{to}}) + 1)d(v_{\text{from}}, v_{\text{to}}) \tag{3.48}$$

We need to obtain the value of the integer $k(v_{\text{to}})$, which is the order (counting from 0) of v_{to} within the solution. This value can be stored in the memory and derived by incrementing the $k(v_{\text{from}})$, belonging to v_{from}, by 1. Value of $k(v_0)$ of start vertex v_0 is initialized to 0.

Generalized Graph Search Problem with order-dependent weights

We approach the dynamic weights scenario with the same analogous attitude - we rearrange the formula in the format of weighted sum of distances. For more compact notation we denote $w_i := w(v_{x_i}, (v_{x_0}, \ldots, v_{x_{i-1}}))$. An important insight is that we assumed that the weights sum up to the area of the environment, i.e. $\sum_{i=0}^{m} w_i = |\mathcal{W}|$. Then we can rearrange formula to the following form:

$$c(\mathbf{x}) = \sum_{i=1}^{m} w_i \sum_{j=1}^{i} d_j = w_1 d_1 + w_2(d_1 + d_2) + \ldots + w_m(d_1 + \ldots + d_m) = \tag{3.49}$$
$$= (w_1 + \ldots + w_m)d_1 + (w_2 + \ldots + w_m)d_2 + \ldots + w_m d_m = \tag{3.50}$$
$$= \sum_{i=1}^{m} d_i(|\mathcal{W}| - \sum_{j=0}^{i-1} w_j) \tag{3.51}$$

Therefore we define the cost function of a partial solution as:

$$f(\mathbf{x}_p) = \sum_{i=1}^{\ell} d_i(|\mathcal{W}| - \sum_{j=0}^{i-1} w_j) \tag{3.52}$$

We plant the solution to this function and obtain:

$$\begin{aligned} f(\mathbf{x}^*) = d_1(|\mathcal{W}| - w_0) + \ldots + d_k(|\mathcal{W}| - \sum_{j=0}^{k-1} w_j) + \\ + d_{k+1}(|\mathcal{W}| - \sum_{j=1}^{k} w_j) + \ldots + d_m(|\mathcal{W}| - \sum_{j=0}^{m-1} w_j) \end{aligned} \tag{3.53}$$

$$\begin{aligned} f(\mathbf{x}') = d'_1(|\mathcal{W}| - w'_0) + \ldots + d'_k(|\mathcal{W}| - \sum_{j=0}^{k-1} w'_j) + \\ + d_{k+1}(|\mathcal{W}| - \sum_{j=0}^{k} w'_j) + \ldots + d_m(|\mathcal{W}| - \sum_{j=0}^{m-1} w'_j) \end{aligned} \tag{3.54}$$

$$f(\mathbf{x}^*_p) = d_1(|\mathcal{W}| - w_0) + \ldots + d_k(|\mathcal{W}| - \sum_{j=0}^{k-1} w_j) \tag{3.55}$$

$$f(\mathbf{x}'_p) = d'_1(|\mathcal{W}| - w'_0) + \ldots + d'_k(|\mathcal{W}| - \sum_{j=0}^{k-1} w'_j) \tag{3.56}$$

Notice that unlike in case of distances, where $d_i = d'_i$ for $i \in \{k+1, \ldots, m\}$, we cannot do the same for the weights that is because the weight depends on all the previously visited vertices, which are not the same, i.e. $w_i = w(v_{x_i}, (v_{x_0}, \ldots, v_{x_{i-1}})) \neq w(v_{x'_i}, (v_{x'_0}, \ldots, v_{x'_{i-1}})) = w'_i$. When these formulas are planted in the first assumption (Equation 3.27) we get:

$$f(\mathbf{x}^*) - f(\mathbf{x}') = f(\mathbf{x}^*_p) - f(\mathbf{x}'_p) + \sum_{i=k+1}^{m} d_i \sum_{j=0}^{i-1}(w'_j - w_j) \tag{3.57}$$

The last term is not necessarily greater or equal to zero, therefore we do not come to a contradiction. However, as we show in the experimental part of the text, this method of computing the cost of a partial solution yields significantly better results than a straight forward definition:

$$f(\mathbf{x}_p) = \sum_{i=1}^{\ell} w_i \sum_{i=1}^{j} d_j \tag{3.58}$$

These two methods have different interpretation in the Mobile Robot Search Problem. The latter function can be interpreted as the expected time to discover the object inside the area corresponding to the visited location, while the first method can be interpreted as the expected time to discover the whole area, but after the last location is visited the rest of the environment is discovered instantaneously.

For the implementation in Algorithm 5, we define the addition in cost by traveling from v_{from} to v_{to} as:

$$f_\Delta(v_{\text{from}}, v_{\text{to}}, M) = d(v_{\text{from}}, v_{\text{to}})(|\mathcal{W}| - |\mathcal{S}(v_{\text{from}})|) \tag{3.59}$$

where $\mathcal{S}(v_{\text{from}})$ expresses the region that was seen from the start up until visiting v_{from}. Update of the value is done so that $\mathcal{S}(v_{\text{to}}) \leftarrow \mathcal{S}(v_{\text{from}}) \cup \mathcal{V}(l(v_{\text{to}}))$, i.e it is a union of region seen until v_{from} combined with the region seen from location corresponding to the v_{to}. The start vertex is initialized as $\mathcal{S}(v_0) \leftarrow \mathcal{V}(l(v_0))$.

Generalized Graph Search Problem with order-independent weights

The last scenario we cover are the static weights, where all vertices within one cluster are assigned the same weights, i.e. $w(v_i) = w(v_j)$ for all $v_i \in V_c, v_j \in V_c$. This scenario may not seem useful but we show in the experiments that the best method that uses static weights have this property. An advantage is that we know in advance what is the sum of all the weights $S_w = \sum_{i=0}^{m} w(v_{x_i})$ for any arbitrary complete solution $\mathbf{x} = (x_0, \dots, x_m)$.

Now we proceed in the similar fashion as in the previous cases and rearrange the cost function as follows:

$$\begin{aligned} c(\mathbf{x}) = \sum_{i=1}^{m} d_i \sum_{j=1}^{i} w_j &= d_1 w_1 + (d_1 + d_2) w_2 + (d_1 + \ldots + d_m) w_m = && (3.60)\\ &= d_1(w_1 + \ldots + w_m) + d_2(w_2 + \ldots + w_m) + \ldots + d_m w_m = && (3.61)\\ &= \sum_{i=1}^{m} d_i (S_w - \sum_{j=0}^{i-1} w_j) && (3.62) \end{aligned}$$

Let us then define the cost of partial solution $\mathbf{x}_p = (x_0, \dots, x_\ell)$ as

$$f(\mathbf{x}_p) = \sum_{i=1}^{\ell} d_i (S_w - \sum_{j=0}^{i-1} w_j) \tag{3.63}$$

Now we evaluate the cost functions

$$\begin{aligned} f(\mathbf{x}^*) = d_1(S_w - w_0) + \ldots + d_k(S_w - \sum_{j=0}^{k-1} w_j) + \\ + d_{k+1}(S_w - \sum_{j=0}^{k} w_j) + \ldots + d_m(S_w - \sum_{j=0}^{m-1} w_j) \end{aligned} \tag{3.64}$$

$$\begin{aligned} f(\mathbf{x}') = d'_1(S_w - w_0) + \ldots + d'_k(S_w - \sum_{j=0}^{k-1} w_j) + \\ + d_{k+1}(S_w - \sum_{j=0}^{k} w_j) + \ldots + d_m(S_w - \sum_{j=0}^{m-1} w_j) \end{aligned} \tag{3.65}$$

$$f(\mathbf{x}^*_p) = d_1(S_w - w_0) + \ldots + d_k(S_w - \sum_{j=0}^{k-1} w_j) \tag{3.66}$$

$$f(\mathbf{x}'_p) = d'_1(S_w - w_0) + \ldots + d'_k(S_w - \sum_{j=0}^{k-1} w_j) \tag{3.67}$$

Notice, that it holds that $w_i = w'_i$ for all $i \in \{0, \ldots, m\}$. After planting the results, we obtain

$$f(\mathbf{x}^*) - f(\mathbf{x}') = f(\mathbf{x}^*_p) - f(\mathbf{x}'_p) \tag{3.68}$$

This results leads the result $f(\mathbf{x}^*_p) - f(\mathbf{x}'_p) \leq 0$ which is a contradiction to the second assumption (Equation 3.28). Therefore, we have proved that this methods also finds the optimal solution.

Implementation of this last case in Algorithm 5 has got the increment in cost function caused by going from v_{from} to v_{to} defined as:

$$f_\Delta(v_{\text{from}}, v_{\text{to}}, M) = d(v_{\text{from}}, v_{\text{to}})(S_w - \Sigma(v_{\text{from}})) \tag{3.69}$$

where $\Sigma(v_{\text{from}})$ is the sum of weights summed over vertices visited from the start up until v_{from}. A memory update is done as $\Sigma(v_{\text{to}}) \leftarrow \Sigma(v_{\text{from}}) + w(v_{\text{to}})$. The start vertex is initialized as $\Sigma(v_0) \leftarrow 0$.

3.3.4 Initial solution heuristics

We modify the construction heuristics of the original GLNS so that it accommodates the assumption that any feasible solution starts at vertex v_0. In the random insertion tour, instead of selecting the first vertex, which is inserted to the empty path, randomly. Similarly in the random tour, the cluster V_0

which contains v_0 is fixed to be first, the order of the remaining clusters is then chosen randomly.

Additionally we come up with the generalized version of construction heuristics proposed for the GSP with order-dependent weights in [27]. The construction is initialized by adding the start vertex v_0 to a partial solution L, see line 1. We store the vertices that are possible candidates, i.e. vertices that belong to unvisited clusters, in a set $\mathcal{C}$, initialization is described at line 2. In every iteration of a loop, a vertex v is selected from candidate set based on a criterion function, which is described later. Then, this vertex is added to the partial solution and all the vertices from the cluster that contain v are removed from the candidate set $\mathcal{C}$ (lines 4 - 6). We use a function $g(v) = V_k$ returning the cluster that contains the argument vertex $v \in V_k$.

Algorithm 6 Constructive heuristic

```
1: $L \leftarrow (v_0)$
2: $\mathcal{C} \leftarrow V \setminus v_0$
3: for $i \in (1, \dots, m)$ do
4:     $v \leftarrow \text{selectVertex}(\mathcal{C})$
5:     $L \leftarrow (L, v)$
6:     $\mathcal{C} \leftarrow \mathcal{C} \setminus g(v)$
7: end for
8: return $L$
```

There paper [27] distinguishes two methods *greedy* and *randomized greedy*. Both methods work very similarly - a certain criterion function is evaluated for all the vertices in the candidate set. In case of the greedy method the vertex with the minimal value of criterion function is selected. In case of the randomized greedy method, the candidate vertices are assigned a probability based on their score of criterion function. The vertex is then selected randomly with their assigned probabilities.

Apart from the randomized and stochastic methods of selecting vertices, we can also distinguish different criterion functions. The original paper lists five possible functions to evaluate the candidate vertex u. All of them calculate the function based on the partial solution $L = (v_{x_0}, \dots, v_{x_\ell})$.

- Added cost:

$$f(u, L) = w(u, L)(\delta(L) + d(v_{x_\ell}, u)) \tag{3.70}$$

- Distance to the next vector:

$$f(u, L) = d(v_{x_\ell}, u) \tag{3.71}$$

- Distance to weight ratio

$$f(u, L) = \frac{d(v_{x_\ell}, u)}{w(u, L) + 0.01} \tag{3.72}$$

- Total distance traveled to weight ratio

$$f(u, L) = \frac{\delta(L) + d(v_{x_\ell}, u)}{w(u, L) + 0.01} \tag{3.73}$$

- Weighted distance

$$f(u, L) = w(u, L)d(v_{x_\ell}, u) \tag{3.74}$$

The performance of the construction heuristics is evaluated in the experimental section.

3.4 Weight computation

In this section we discuss the selection of weight function. Previously, we have presented the GSP, where the weights are static, and GSP with order-dependent weights, where the weights change dynamically based on the previously visited vertices. The main advantage of static weights is that the weight does not have to be recalculated, which may be costly, every time we need to find out the cost. Although we have shown, how to reduce the number of weight recalculations in evaluation of operators, it may still be too time expensive. On the other hand, when we are solving the GSP with order-dependent weights the static weights serve only as an approximation. The approximation introduces an error that may not be leveraged even by the quicker computation time. This may vary based on our time budget and complexity of the instance.

3.4.1 Static weights

We consider three variants of static weights. *Unit weights* is a case, where the weights assigned to all vertices equal to the 1, i.e. $w(v_i) = w(v_j) = 1, \forall i, j$. This scenario corresponds to the case where visiting any vertex brings the same amount of information, no matter the order nor the specific location. The scenario, however, is in the Mobile Robot Search Problem not realistic as experiments show that the weights usually decrease within the solution. This case is equivalent to the TDP.

Visibility weights is an approximation where the weights are equal to the area of visibility region with no prior regions discovered. It was experimentally shown [27] that this is a very good approximation for cases, where the regions have small mutual overlap. When the overlap is zero, then this approximation actually yields the real cost.

Greedy weights is an approximation method recently proposed in [22] for solving GSP. The weights are obtained by running a greedy algorithm that constructs a complete solution. The procedure starts in the vertex v_0 and in every iteration vertex is selected and added at the end of a partial solution. The vertex is selected so that it minimizes the ratio of newly discovered area to the total traveled distance. The weight of a vertex is then assigned based on the area of newly discovered region by the vertex, i.e.

$$w(v_{x_i}) = |\mathcal{V}(l(v_{x_i})) \setminus \bigcup_{j=0}^{i-1} \mathcal{V}(l(v_{x_j}))| \tag{3.75}$$

The regions associated with the weights are mutually disjoint and their union forms the whole environment. In [22] this method has outperformed both unit and visibility weights. However, this method cannot be directly used for Generalized GSP. Therefore, we propose two modification suitable for this problem.

In both modifications we assume that we have found the order by a greedy algorithm. In this case, the ordered vertices do not contain all the vertices from V, but only one vertex from each cluster. In the first variant, for all vertices u belonging to the same cluster as v_{x_i}, the weight is assigned as follows:

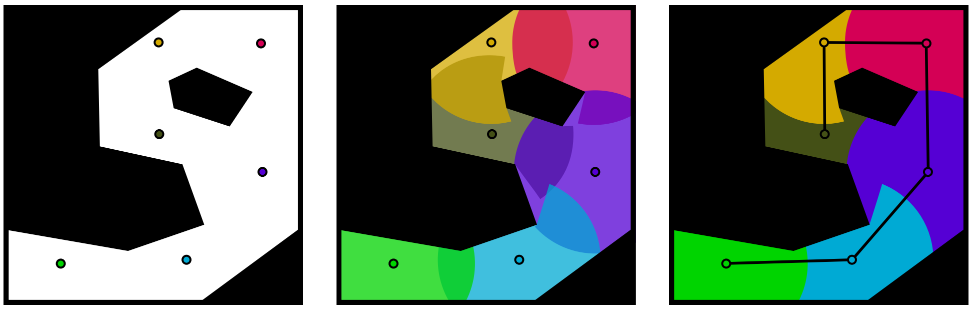
$$w(u) = |\mathcal{V}(l(u)) \setminus \bigcup_{j=0}^{i-1} \mathcal{V}(l(v_{x_j}))| \tag{3.76}$$

In this variant, each vertex is assigned its own weight. In the second variant, for all vertices u from the same cluster as v_{x_i}, the weight is assigned as:

$$w(u) = |\mathcal{V}(l(v_{x_i})) \setminus \bigcup_{j=0}^{i-1} \mathcal{V}(l(v_{x_j}))| \tag{3.77}$$

All the vertices from the same cluster have therefore the same weight. In the experimental section we evaluate performance of both variants. Further, we test a different criteria in the vertex selection of the greedy algorithm.

Figure 3.6 shows an illustration of the different methods of static weights.



(a) : Unit weights. **(b) :** Visibility weights. **(c) :** Greedy weights.

Figure 3.6: Illustration of the static weights.

3.4.2 Dynamic weights

Dynamic weights are used in the Generalized GSP with order dependent weights. In order to evaluate the weights we need a way of performing set operations union and difference. For practical reasons we do not attempt to find analytical solutions but instead the visibility regions are approximated by *polygon approximation* or approximation using *sampling methods*.

Polygonal approximation

Generally the shape of a visibility region composes of straight lines and circular arcs. We can approximate the circular arcs by multiple straight lines and we obtain a polygon. Once we approximate the visibility regions by polygons, the set operations can be then obtained using clipping algorithms. In the experimental setup we utilize `Clipper2`[3] library, which based on Bala Vatti's polygon clipping algorithm [38]. Neither the paper nor the library states the complexity of the algorithm, however, in [20] a new algorithm, which outperforms Vatti's algorithm, is proposed and runs in $O((n+k)\log n)$, where n is total number of edges across all the polygons involved and k is the number of intersections of all polygon edges. Therefore we assume that the implementation `Clipper2` runs at best in $O((n+k)\log n)$.

[3] https://www.angusj.com/clipper2/Docs/Overview.htm

Sampling methods

Sampling methods are used in order to approximate the area of regions by placing samples on the map. Each sample point s belongs to the vertex v if it resides inside of the visibility region of the vertex, i.e. $s \in \mathcal{V}(v)$.

The first method used for sampling is random sampling. The second method is equidistant grid sampling, i.e. the map is split into square grid cells and the samples lie in the center of the square. In an ideal case the map would be a square shaped with no holes. In reality, the map can have arbitrary shape. Generally the map can be wrapped into a rectangle with dimensions $[0, W] \times [0, H]$. We want to cover the map with n_x columns in the x-axis and n_y rows in the y-axis. Assume we want to cover the map with n samples, where the distance between two adjacent samples is a in one of the dimensions. From these requirements we obtain the following:

$$an_x = W \tag{3.78}$$

$$an_y = H \tag{3.79}$$

$$n_x n_y = n \tag{3.80}$$

after rearranging the equations we receive:

$$n_x = \sqrt{\frac{nW}{H}} \qquad n_y = \sqrt{\frac{nH}{W}} \qquad a = \sqrt{\frac{WH}{n}} \tag{3.81}$$

Generally, n_x and n_y are real numbers, but number of rows and columns in our context only makes sense as natural numbers, therefore we round the results to nearest natural number. We then obtain:

$$a\lfloor n_x \rceil + e_x = W \qquad a\lfloor n_y \rceil + e_y = H \tag{3.82}$$

Where e_x, e_y are the errors, $e_x \in [-\frac{a}{2}, \frac{a}{2}]$, $e_y \in [-\frac{a}{2}, \frac{a}{2}]$. The set of all the samples is then

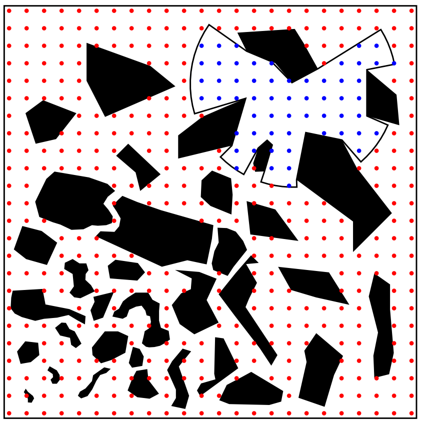
$$\{(ia + \frac{1 + e_x}{2}, ja + \frac{1 + e_y}{2}) | i \in 0, \ldots, \lfloor n_x \rceil - 1, j \in 0, \ldots, \lfloor n_y \rceil - 1\} \tag{3.83}$$

In an environment that contains obstacles, some of the samples may fall into an obstacle. These samples do not bring any information into the approximation and therefore we discard them. Because of the discarded sample, the total of actually used samples would be less than intended. In order to better match the actual amount of samples with the intended amount, we computed the number of samples as follows:

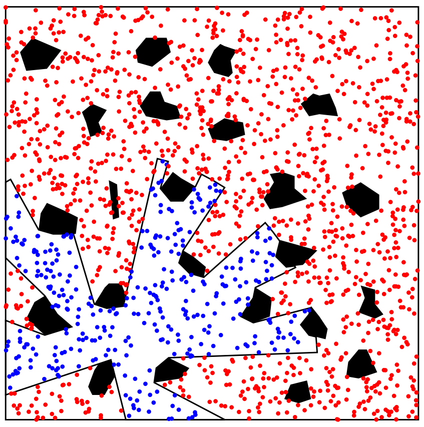
$$n = \frac{WH}{|\mathcal{W}|} n' \tag{3.84}$$

Where $|\mathcal{W}|$ is the area of the environment, WH is the area of rectangle wrapping the environment and n' is the intended number of samples. Although there are no guarantees that the final set of samples have the intended cardinality, empirically we verified that this method produces amount of samples relatively close to the intended amount.

Figure 3.7a shows an example of the equidistant sampling and Figure 3.7b shows an example of the random sampling.



(a) : Equidistant sampling on the map *large* with 425 samples



(b) : Random sampling on map the *potholes* with 1600 samples.

Figure 3.7: Two examples of possible sampling, with one visibility region highlighted in each figure.

Chapter 4

Experiments

In this chapter we provide experimental evaluation of the modified GLNS solver. The overall goal is to evaluate the performance of the solver on the Generalized GSP with order-dependent weights and examine the quality of presented the approximations and their trade-off with computational cost.

4.1 Software and hardware

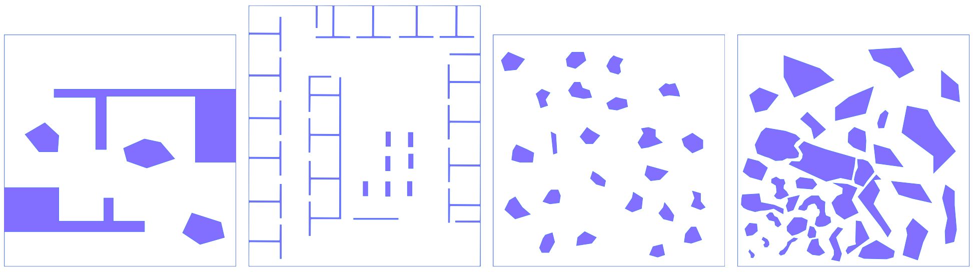
The solver was implemented in `C++` and is based on the implementation of the original GLNS by Jan Mikula[1]. The experiments were carried out on the Lenovo ThinkPad P51 with 32.0 GiB memory, processor Intel® Core™ i7-7820HQ CPU @ 2.90GHz $\times$ 8 and Mesa Intel® HD Graphics 630 (KBL GT2) graphics.

4.2 Instances

We carried out the experiments on four maps, shown in Figure 4.1. The maps differ in their size, shape, density of obstacles, complexity of obstacles and other parameters. For each map we generated five different instances by dual

[1] https://orcid.org/0000-0003-3404-8742

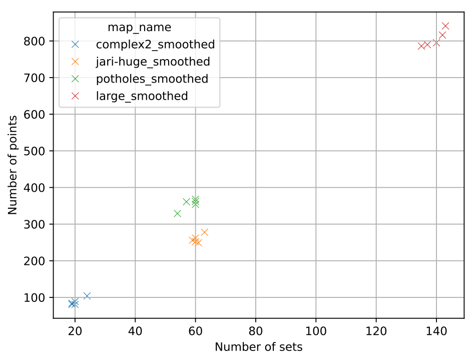
sampling method, described in [24]. The sampling generates a set of convex polygons that completely cover the environment. Each of the generated polygons corresponds to a cluster. Sensing locations belonging to a certain cluster are constructed by sampling the border of the corresponding polygon. The described instance generation was done using *Visis*[2] framework with visibility radius of a robot set to 10.



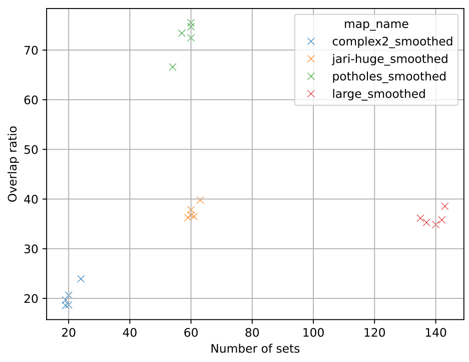
(a) : Map *complex2*. **(b) :** Map *jari-huge*. **(c) :** Map *potholes*. **(d) :** Map *large*.

Figure 4.1: Maps used for experiments.

The instances range from problems with 19 clusters and 81 vertices up to problems with 143 clusters and 841 vertices. Figure 4.2a shows the size of the individual instances. Figure 4.2b shows the overlap of visibility regions, computed as the sum of areas of all visibility regions divided by the area of the environment.



(a) : Relation between number of sets and points.



(b) : Relation between number of sets and overlap retio.

Figure 4.2: Properties of the instances used for the experiments.

[2]gitlab.ciirc.cvut.cz/mission-planning/visis-planner

4.3 Metrics

In most of the experiments, we examine the evolution of solution's quality in time. The goal of the MRSP is to minimize the expected time. Therefore we measure cost of the solution, i.e. the expected time to find an object. We also measure the elapsed computational time displayed on the x-axis. We set the time budget to 60 seconds, as it is sufficiently long time interval, where we can see a convergence of the search.

In order to make the results more interpretable, instead of showing the absolute value of the objective function, we show the cost's gap, i.e. the relative difference of the cost to the best found solution. With gap, we can easier see, how does the solution compare to the best known solution percentage-wise. The gap of a solution $\mathbf{x}$ is computed as follows:

$$c_{\text{gap}}(\mathbf{x}) = 100\frac{c(\mathbf{x}) - c_{\min}}{c_{\min}}[\%] \tag{4.1}$$

where $c(\mathbf{x})$ is the cost of solution $\mathbf{x}$ and $c_{\min}$ is the minimal value of cost function found across all experiments for a specific instance. Along with the averaged run shown on the y-axis, we also show a standard deviation error bars. For normally distributed data, around 68% of the data lie inside of the error bars.

Note that in order to measure the expected time we may need to find the intersections of two shapes made up of straight lines and circular arcs. For practical reasons we do not find the intersections analytically but instead we approximate the shapes using polygons. Therefore measured expected time is not exactly equal to the real expected time, but for our purposes the error is negligible.

4.4 Cost modes

In the first experiment, our goal was to explore the dissimilarities in the cost functions. While the definitions of costs differ, all of them, in essence, minimize the traveled distance. In this experiment, we kept the original GLNS algorithm with a modification that instead of using the length of a tour as a cost function, one of the tested cost function is used. The examined cost functions and approximations were described in the theoretical part of this thesis. We distinguish the following cost functions:

- *PathLen* is simply a sum of distances, i.e. the total traveled distance.
- *ExpectedTime* is the reference method to compute the expected time, i.e. uses polygonal approximation. The weights are order dependent.
- *ExpectedTimeGridMap* uses sampling method for weight computation. In this experiment the map was sampled with 400 samples equidistantly.
- *ExpectedTimeVisibilityWeights* is an approximation of the expected time using the area of visibility region.
- *ExpectedTimeUnitWeights* approximates all the weights by 1.
- *ExpectedTimeGreedyWeights* approximates the expected time with static weights as described in Section 3.4.1. For the initialization we use greedy distance construction heuristic.

In this experiment we ran the solver in total 720 times, that is for a unique combination of map and cost function 30 runs. Figure 4.3 shows that across all maps *ExpectedTimeGridMap* yields the best solutions. However, we could argue that without limited time budget the *ExpectedTime* would eventually converge to same or even better result, as the objective criterion is actually the same as the cost function. Figure 4.4 demonstrates it by comparing the *ExpectedTime* and the *ExpectedTimeGridMap* on the *large* map. Instead of using elapsed time on x-axis, we use iterations. It is apparent that both cost modes behave very similarly. Out of the cost modes that are not order-dependent. There is not a clear winner, but greedy weights seem to outperform the rest in most of the cases. Notice that in Figure 4.3 actually increases with increased time. This effect happens, because the objective cost, displayed in the graphs, is not equal to the cost functions used by the the solver. Clearly, optimizing one cost function may worsen the quality computed using a different cost function.

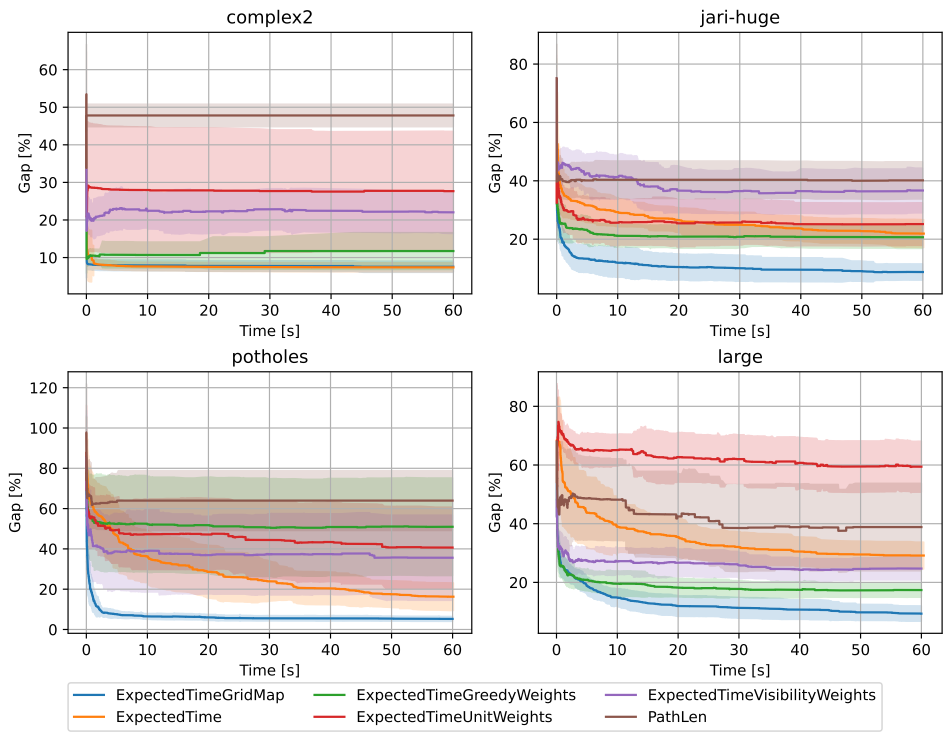


Figure 4.3: Results of experiments with the original solver with replaced cost functions.

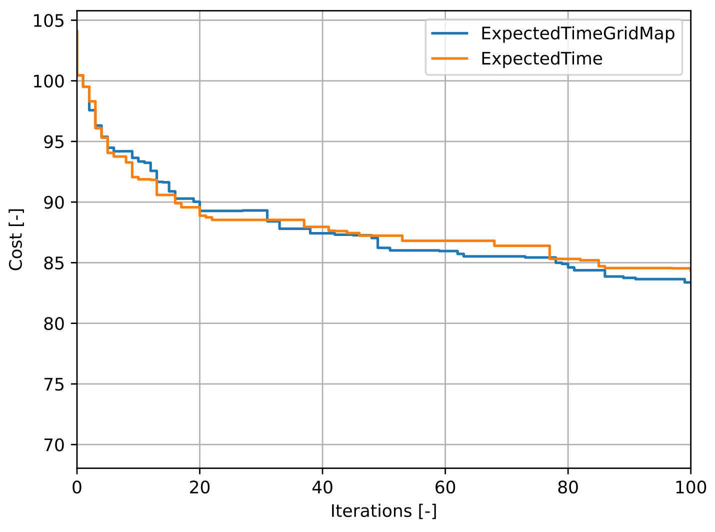


Figure 4.4: The first 100 iterations of the search on the map *large*.

4.5 Operator selection

Since the previous experiments showed that the order-dependent approximation of the objective function using sampling methods seems to offer the most reasonable trade-off between computational time and quality of solution, we continue this next experiment with the same cost function.

In the theoretical section we have shown modifications to the original GLNS that allow adaptation to minimize the cost specific for the Generalized GSP with order-dependent weights, i.e. the expected time. Since the usage of dynamic weights increases the computational complexity caused by the necessity to recalculate the weights using the set operations, we want to evaluate the influence of the specific operators on the quality and time. Implementation of the GLNS can be done so that one iteration of search is divided into five steps.

The experiment is set up so that the we start with the original solver, then for each part of the experiment one of the steps of the original procedure is replaced with the corresponding step but adjusted for optimization of different cost function. The options are:

- *None* refers to none of the order-dependent weights specific operators, i.e. the original solver.
- *Removal* refers to the order-dependent weights specific removal heuristics.
- *SelectSet* refers to the the order-dependent weights specific set selection within the insertion heuristics.
- *InsertVer* refers to the the order-dependent weights specific vertex insertion within the insertion heuristics.
- *ReOpt2* refers to the the order-dependent weights specific Re-Optimize local optimization.
- *MoveOpt* refers to the the order-dependent weights specific MoveOpt local optimization.

Figure 4.5 shows the effect of individual scenarios. We can see that *InsertVer*, *ReOpt2* and *MoveOpt* alone outperform the solver without cost function specific operators. Out of the local optimization methods *MoveOpt* seems to outperform *ReOpt2*. On the map *complex2*, *ReOpt2* method performs significantly worse than *MoveOpt* and seem to got stuck in a local optima. This result is not that surprising as Re-Optimize cannot change the order of clusters and therefore may be more prone to plateau during the search. On the other hand *SelectSet* performed the worst, which may be caused by the computational expanse of evaluating the insertion cost of all possible clusters. There were total of 600 runs of the solver, with 25 runs for a unique combination of a map and an option.

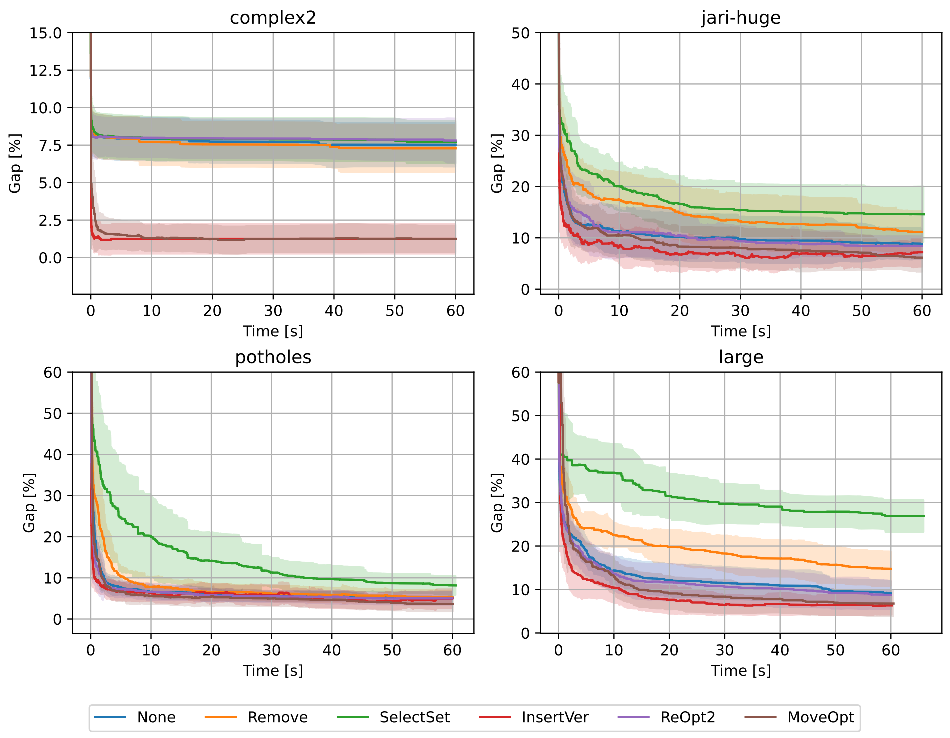


Figure 4.5: Influence of including a specific operator.

The next experiment is similar to the previous one, but now we started with a solver that has all parts of the implementation adjusted to minimizing the expected time with dynamic weights. This option is denoted *All*. The remaining options examine the situation, where one block of the solver is replaced with the original implementations. We denote these *NotRemove*, *NotSelectSet*, *NotInsertVer*, *NotReopt* and *NotMoveopt*.

Figure 4.6 displays that all of the methods performed similarly, except for *NotInsertVer*, which performed the worst. An interesting result is shown for the map *large*, where the search stagnated at a very bad quality of solution. In this experiment there were also 600 runs in total and 25 runs per unique tuple map and option.

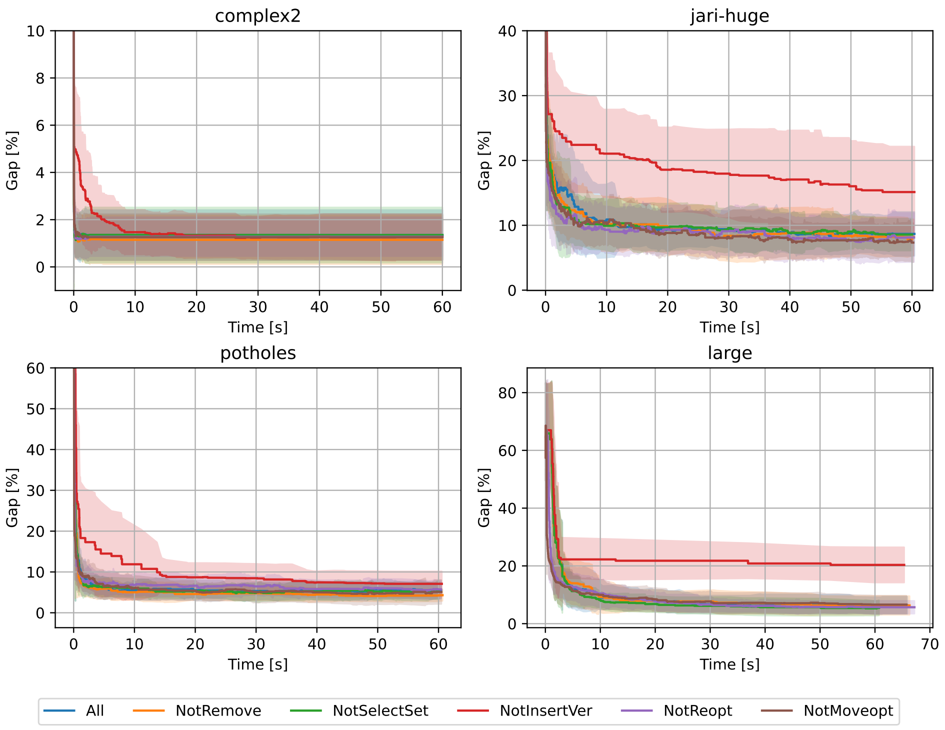


Figure 4.6: Influence of not using a specific operator.

Our conclusion from these two experiments is that the most crucial piece of the solver is the insertion of vertices, followed by local optimization operators. The cost function specific removal heuristics and set selection do not seem to be worth the increased computational complexity.

4.6 Sampling method parameters

As we have shown in the previous experiments approximation based on sampling seems to be the best approximation candidate. In those experiments we used settings with 400 equidistant samples. Now we examine the effect of different sampling strategies and number of samples. The sampling strategies were explained in the theoretical part of this thesis. Our experiment consists of testing two sampling strategies *equidistant* and *random*, and five values of the number of samples 100, 200, 400, 800 and 1600. We have tested combinations of all the parameters. Figure 4.7 shows the influence of the sampling strategy. In the processing of this experiment, we combined all of the possible values of the number of samples. This experiment consisted of 400 runs of the solver, with 10 runs per unique combination of map, number of samples and sampling strategy. It is apparent that on all of the maps, equidistant sampling outperformed the random sampling in the quality of solution, the time required to converge seems to be independent on the sampling strategy.

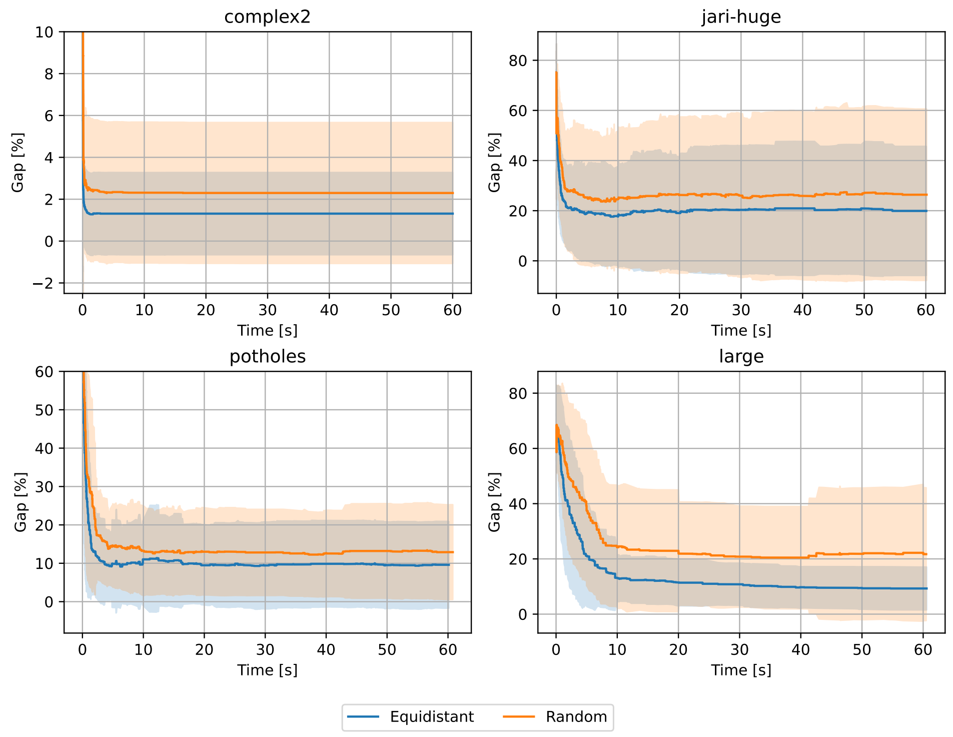


Figure 4.7: Effect of sampling strategies.

Figure 4.8 shows the effect of sampling density. Because the the former experiment showed that random sampling performs worse, we only show the results of equidistant sampling. In accordance to expectations, with increasing number of samples, the best solution found by the solver is better, while the time to converge increases. From the experiment, we imply that the most suitable choice for the number of samples is somewhere between 400 and 800 samples. 1600 samples do not significantly improve the quality of best found solution and moreover it delays the convergence time. However, the selection depends on the specific instance, more complex maps with narrow passages and complex obstacles may require higher sampling density. Important finding is that we did not see a need for increasing sampling density with an increase in the size of the map, in fact on the biggest map *large* the 400 samples found the solution with cost very close to the cost of solutions found by sampling with 800 and 1600 samples.

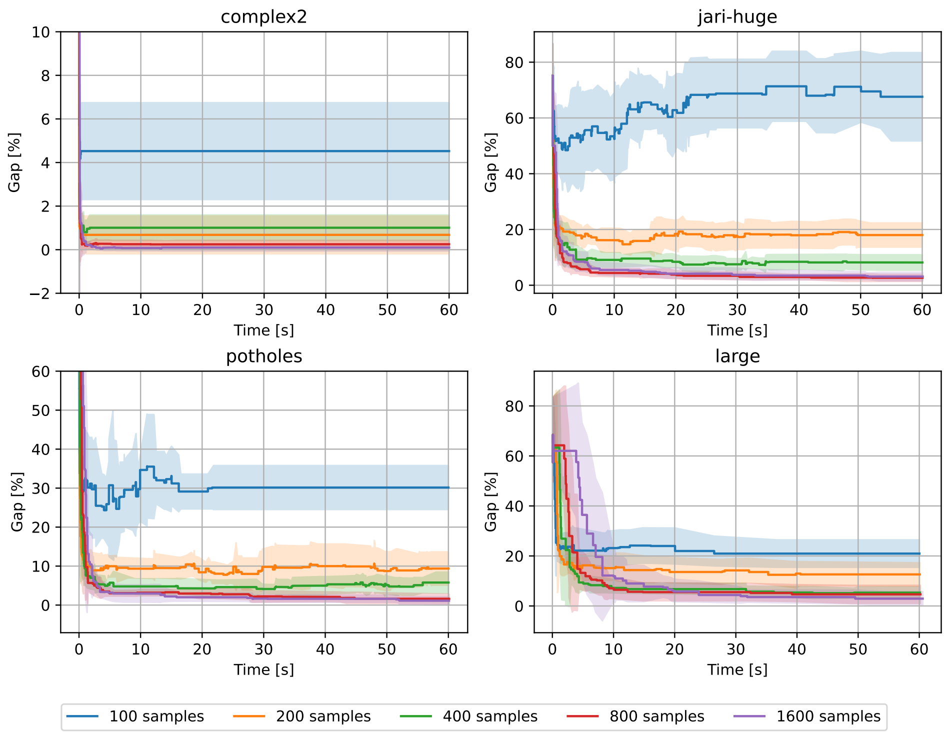
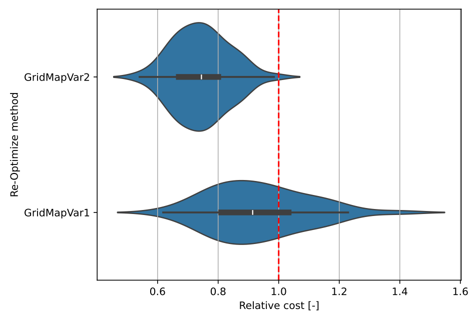


Figure 4.8: Effect of different number of samples in the sampling approximation.

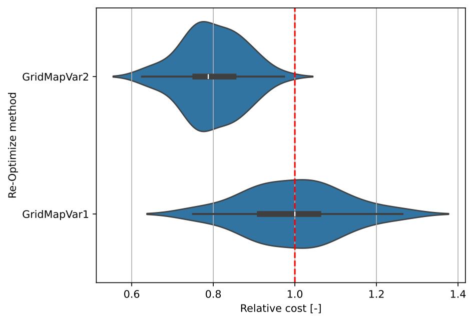
4.7 Re-Optimize method

In this section we test the performance of two variants of Re-Optimize operator tailored to the Generalized GSP with order-dependent weights.

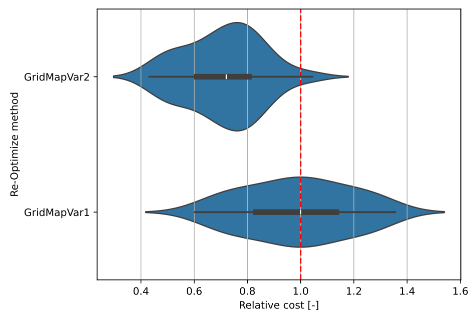
- *GridMapVar1* refers to the Re-Optimize method using dynamic programming with the more straight forward definition of the transition function defined in the Equation 3.58.
- *GridMapVar2* refers to the Re-Optimize method using dynamic programming with the transition function defined by the Equation 3.52. This function is inspired by the method used in the case of the Generalized TDP, where it yields the optimal solution.



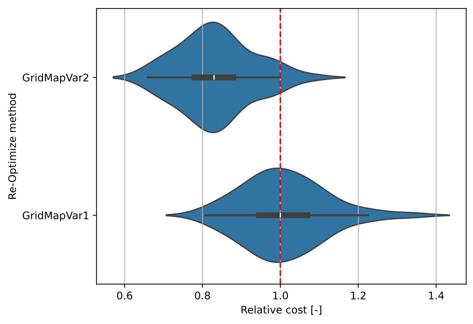
(a) : Map *complex2*.



(b) : Map *jari-huge*.



(c) : Map *potholes*.



(d) : Map *large*.

Figure 4.9: Comparison of Re-Optimize methods.

In this experiment we constructed a random solutions, then both variants of Re-Optimize were applied on the solutions. Figure 4.9 shows violin plots with the quality of resulting solutions. The measured quality was normalized by the median cost of the initial random solutions. It is apparent that the *GridMapVar2* exceeds the improvement on all the maps. Furthermore the

GridMapVar1 improved the initial solution on average only on the simplest map *complex2*. The Re-Optimize is implemented so that if no improvement is found, the original solution is returned, therefore it is possible that the solutions generated by *GridMapVar1* were in fact worse that the initial solutions.

4.8 Greedy weights

In this experiment we examine the performance of the two suggested greedy weight variants. In the theoretical part we proposed two modifications to the original greedy weights approximation used in the GSP. We denote the variant, where each vertex has its own weight, as *Vertex*. This variant is described by the Equation 3.75. The other variant, where all vertices within the same cluster share the same weight, is denoted as *Set* and is described by the Equation 3.77.

Note that the weights assigned to the individual vertices are determined by the initial solution constructed in the beginning of every iteration of the GLNS algorithm. In order to explore the effect of the initial solution on the final cost, we experimented with all five of the deterministic greedy initial solution heuristics described in Subsection 3.3.4. In the first part of the experiment, we tested the performance of both variants of greedy. All runs independent on the initial solution heuristic were combined. There were 100 runs of the solver for every combination of a map and a greedy weights variant. Figure 4.10 shows the evolution of solution quality in time. It is apparent the the *Set* variant exceeds the *Vertex* variant across all maps.

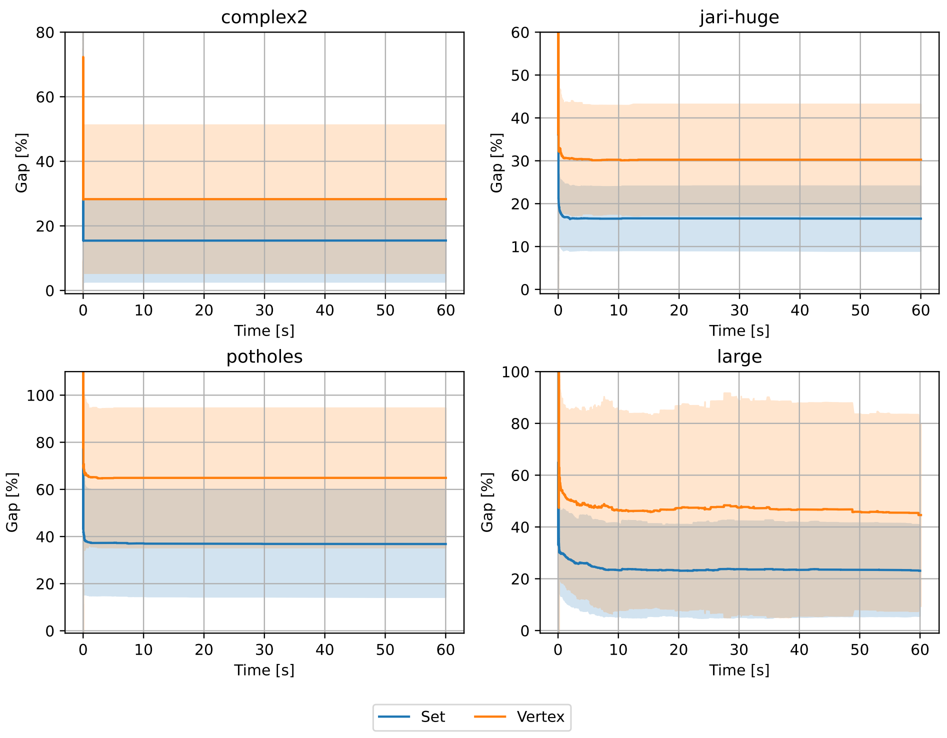


Figure 4.10: The comparison of different variants of greedy weights.

Another view of the same experiment is provided by the Figure 4.11. The figure is a scatter plot, where each point corresponds to one search. In this figure we only display the runs of *Set* variant. The value on x-axis is the cost of the initial solution and the value on y-axis is the final cost obtained by the search. The figure also displays an orange line, which interpolates the data using linear regression. The black dashed line is a unit slope linear function passing through the origin. This line separates the plane into two parts. The points above this line actually correspond to search runs, where the solution got worse. Such situation is not pleasant, but can happen because the cost computed using greedy weights is only an approximation of the real cost. Apart from that, we can see that there does not seem to be a significant correlation between the initial solution and a final solution.

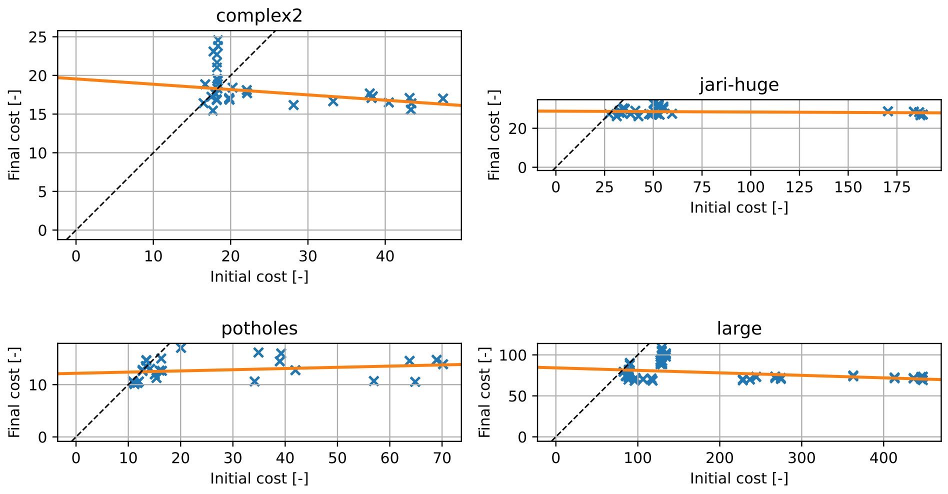


Figure 4.11: The comparison of the initial cost and the final cost.

Next, we examined a combination of greedy weights and sampling method approximation. In the previous experiments, we observed that these two methods performed the best. With greedy weights being faster at convergence and sampling method finding better final solution. In Figure 4.12 three solver settings are compared. A modified solver that uses both greedy weights specific operators and cost function is denoted as *NoGridmap*. The next two modifications have greedy weights specific operators but use cost function computed using sampling. They are denoted as *Gridmap* and *GridmapRnd*. While *Gridmap* uses *GreedyDist* method for initialization of greedy weights, *GridmapRnd* uses *RndGreedyDist* initialization. We can see that both options that use the combination of method yielded better results. There were in total 540 runs of the solver and 45 runs for every combination of a map and an option.

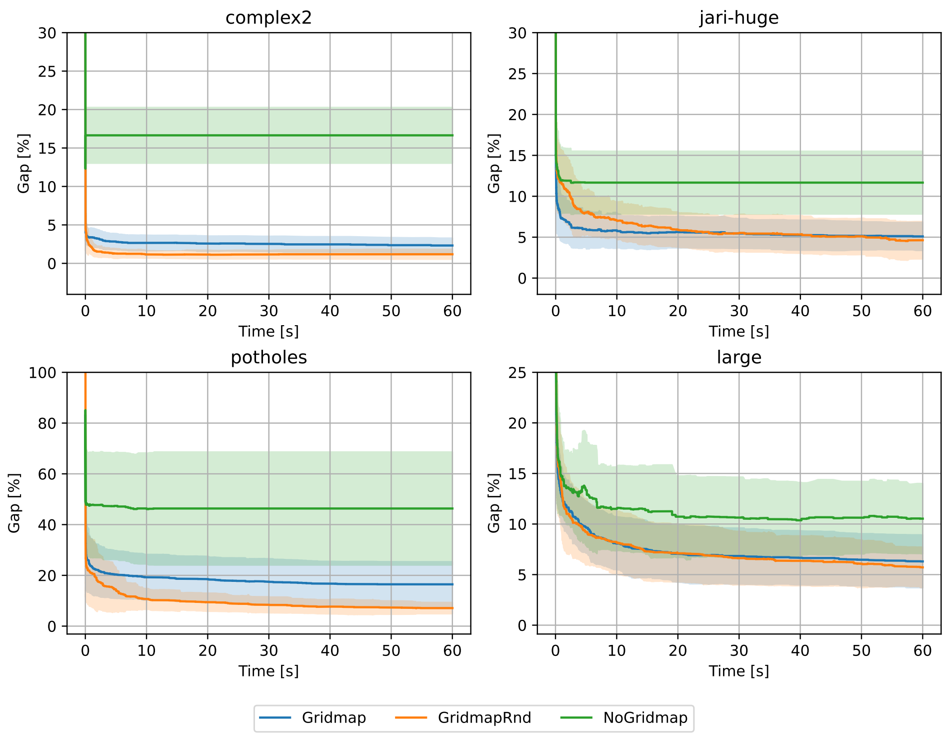


Figure 4.12: The effect of combining greedy weights with sampling approximation.

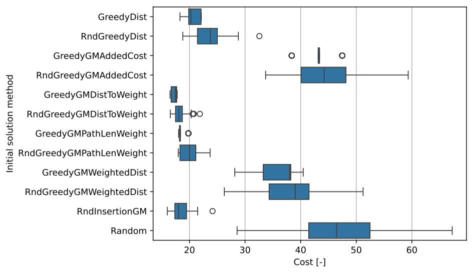
4.9 Initial solution heuristics

In this experiment we tested the quality of initial solutions created by the presented construction heuristics. While a better initial solution does not always equal a better final solution, it is reasonable to start the search with a sufficiently good solution. Because the GLNS constructs the initial solution several times within the search, it is also appropriate to utilize a construction method, which generates diverse selection of initial solutions. With more diverse initial solutions it is more likely that broader part of the search space can be discovered. We distinguish 12 construction heuristics

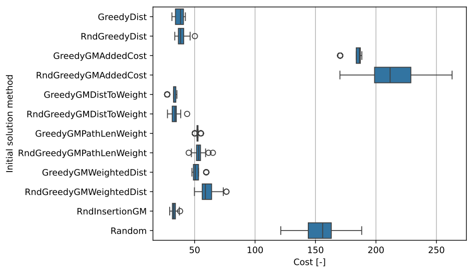
- *GreedyDist* refers to the method minimizing the distance to the next vertex described by Equation 3.71, *RndGreedyDist* is its randomized variant.
- *GreedyGMAddedCost* refers to the method minimizing the added cost by appending selected vertex described by Equation 3.70. The added cost is computed using sampling approximation with 400 equidistant samples. *RndGreedyAddedCost* is its randomized variant.
- *GreedyGMDistToWeight* refers to the method minimizing distance to the next vertex to the area discovered by that vertex ratio, the formula is described by Equation 3.72. The distance to weight ratio is computed using sampling approximation with 400 equidistant samples. *RndGreedyDistToWeight* is its randomized variant.
- *GreedyGMPathLenWeight* refers to the method minimizing the total traveled distance to discovered area ratio, described by Equation 3.70. The function is evaluated using sampling approximation with 400 equidistant samples. *RndGreedyGMPathLenWeight* is its randomized variant.
- *GreedyGMWeightedDist* refers to the method minimizing the distance to the next vertex times the area discovered by this vertex, the formula is described by Equation 3.74. The weighted distance is computed using sampling approximation with 400 equidistant samples. *RndGreedyGMWeightedDist* is its randomized variant.
- *RndInsertionGM* refers to the method random insertion tour. The necessary weight computation is obtained sampling approximation with 400 equidistant samples.
- *Random* refers simply to the random tour.

For the randomized version, we sort the candidates by their greedy objective function. A vertex is selected from the top three candidates with probabilities

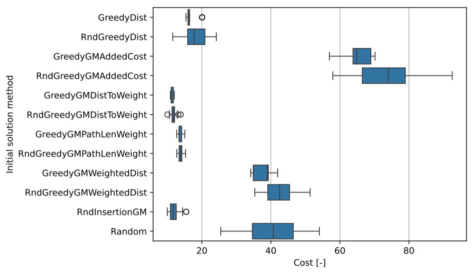
$p_1 = 0.66, p_2 = 0.22$ and $p_3 = 0.12$. When there is less then three candidates, the best one is selected deterministically.



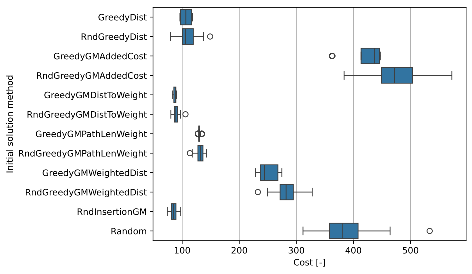
(a) : Map *complex2*.



(b) : Map *jari-huge*.



(c) : Map *potholes*.



(d) : Map *large*.

Figure 4.13: Box plots of the quality of initial solutions obtained by different methods.

Figure 4.13 displays performance of the individual construction methods using box plot graph. We can see that across all the maps, both deterministic and randomized versions of *GreedyWeightedDist* performed the worst. Both versions of *GreedyWeightedDist* performed, except for one map, also very poorly. In both of these methods, the greedy criterion is computed such that the area of discovered region is multiplied by some sort of distance. The criterion can be interpreted as minimizing both the distance and the area of discovered region. Therefore vertices that discover less area are preferred, which is in conflict with what we have observed in the best quality solutions. When the deterministic and randomized variants are compared, we can see that on average the deterministic variants yield better solutions and their deviation is smaller, which may imply less diverse initial solutions. In this experiments, we do not show computational time, because it was negligible in comparison to the total length of the search.

4.10 The selected solver settings

Based on the knowledge obtain from the previous experiments, we believe that modified solver performs the best in two modes:

- Mode A is suitable for scenarios where the time budget is highly limited. This mode is composed from operators specific for greedy weights and cost computation using sampling approximation.
- Mode B is suitable for scenarios, where the time is not that critical. This mode uses the original set selection and vertex removal, but vertex insertion and local optimization procedures modified for the sampling approximation.

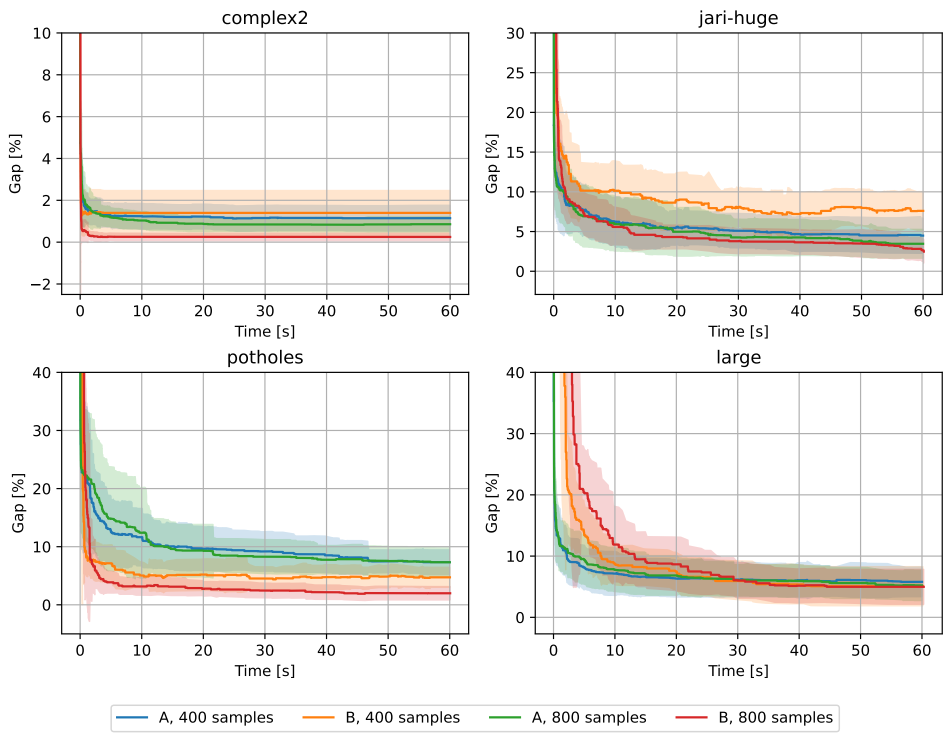
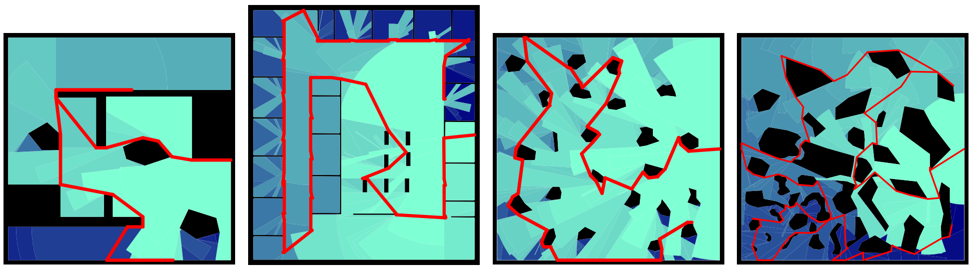


Figure 4.14: The selected solver settings.

Figure 4.14 displays the results of the two modes with number of samples 400 and 800. During this experiment the solver was run in total 400 times, with 25 runs per unique combination of map, mode and the number of samples. The figure shows that in the early stages of the search mode A performs better. In the later stages, mode B with 800 samples clearly outperforms all other combinations of parameters. To our surprise both variants of mode A

surpassed the mode B with 400 samples in the quality of the final solutions. Apart from that, the results correspond to our expectations.



(a) : Map *complex2*. **(b) :** Map *jari-huge*. **(c) :** Map *potholes*. **(d) :** Map *large*.

Figure 4.15: Selected visualized solutions.

From this experiment, we conclude that both modes perform in accordance to our intentions and expectations. The number of samples should be chosen based on the complexity of a map and our time budget.

For illustration, in Figure 4.15, we provide examples of solutions found during the experiments. The trajectory is visualized with the red curve, the discovered regions are displayed using different colors. The region discovered earlier in the search have lighter tone of blue and the regions discovered later have darker tone of blue.

4.11 Comparison with Hexaly Optimizer

In this final experiment we compare the modified GLNS to the Hexaly Optimizer - a general purpose optimizer, which claims to be the world's fastest optimization solver for routing, scheduling, packing, and more. We have decided to make a comparison to this solver, because it is the main topic of a concurrent thesis by Libor Dubský [6]. From our instances, we generated a dataset of problems. In this case, we experimented only with the order-independent weights, more specifically the visibility weights. Libor Dubský wrote a model for the Hexaly Optimizer and provided us with the results. Both solvers ran 50 times for each map.

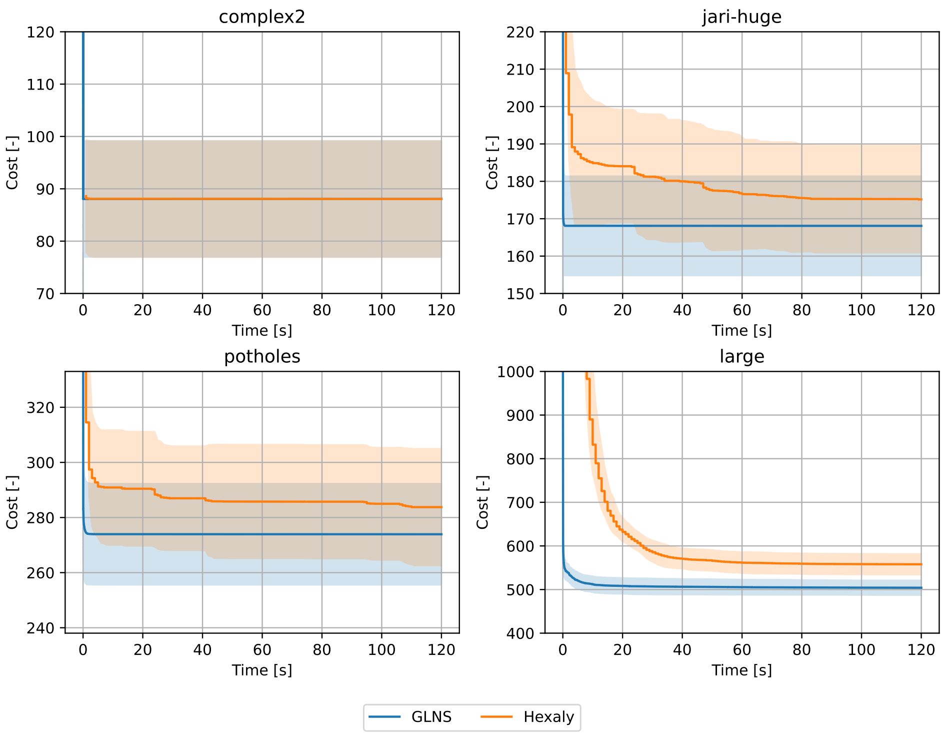


Figure 4.16: Comparison to the Hexaly Optimizer.

Figure 4.16 shows a comparison of the two solvers. Hexaly Optimizer provides the solution every second, therefore the corresponding graphs start at 1 second. It is apparent that the GLNS outperforms the Hexaly Optimizer in both the quality of the best found solution and the speed of convergence. Although it should not be much of a surprise that the solver specifically designed for solving the Generalized GSP surpasses the general purpose solver, this experiment serves as a check that we successfully modified the GLNS so that it can tackle the Generalized Graph Search Problems.

Chapter 5

Conclusion and future work

In this work, we have defined the Generalized GSP with order-dependent weights as a combination of two combinatorial optimization problems GSP and GTSP with order-dependent weights. In the theoretical part, we presented GLNS - a solver using ALNS metaheuristics to tackle the GTSP. We have shown the modifications necessary to be done in order to solve the newly defined problem. We have also defined formulas for a more efficient evaluation of heuristic operators. These modifications have been implemented in C++ to an existing GLNS solver and tested experimentally. The experiments have shown a trade-off between quality of a solution and computational time of different heuristic operators, based on specific approximation methods. We identified the most successful heuristic operators and their appropriate combinations. The most successful operators were based on greedy weights and sampling approximations. Finally, we compared the modified GLNS solver to a general-purpose Hexaly Optimizer. The experiments show that the modified GLNS surpassed Hexaly Optimizer in both the quality of the solution and the computational time.

During work on this thesis the main concern was the high expense of recalculating a weight based on previously visited vertices. This high expense was the motivation for using the approximation methods that were studied in this work. Although we have seen some of the approximation to be very successful, we believe that there is still room for improvement and further research. In the implementation and experiments, we kept heuristic operators, based on different approximations, separate. This allowed us to clearly see the effects of different approximation methods. However, in order to fully utilize the adaptivity of the GLNS, more operators could be used in the bank of methods at the same time. The solver would then itself decide, which heuristic operators and which approximations to use based on their performance. This should lead to better robustness and independence of the specificity of an instance. Moreover, some of the studied operators are computationally inexpensive but not as powerful as other more expensive but stronger operators. The adaptive scheme of GLNS could then select inexpensive operators earlier in the search and later use more powerful operators. This could lead to even better trade-off between fast convergence of the search and quality of the final solution. Because we have mainly focused on approximation methods and their incorporation into the existing solver, there is a room for implementation of some other neighborhood structures, which could be used as a local optimization methods.

Appendix A

Acronyms

ALNS	Adaptive Large Neighborhood Search
GLNS	Effective large neighborhood search heuristic for the Generalized Traveling Salesman Problem
GRASP	Greedy Randomized Adaptive Search Procedure
GSP	Graph Search Problem
GTSP	Generalized Traveling Salesman Problem
GVNS	General Variable Neighborhood Search
LNS	Large Neighborhood Search
MRSP	Mobile Robot Search Problem
SA	Simulated Annealing
SVND	Sequential Variable Neighborhood Descent
TDP	Traveling Deliveryman Problem
TSP	Traveling Salesman Problem
VND	Variable Neighborhood Descent
VNS	Variable Neighborhood Search

Appendix B

Bibliography

[1] L. Bianco, A. Mingozzi, and S. Ricciardelli. The traveling salesman problem with cumulative costs. *Networks*, 23(2):81–91, 1993.

[2] F. Busetti. *Simulated annealing overview.* 05 2001.

[3] F. Chen, H. V. Nguyen, D. A. Taggart, K. Falkner, S. H. Rezatofighi, and D. C. Ranasinghe. Conservationbots: Autonomous aerial robot for fast robust wildlife tracking in complex terrains. *Journal of Field Robotics*, 41(2):443–469, Nov. 2023.

[4] G. D. Cubber, D. Doroftei, K. Rudin, K. Berns, A. Matos, D. Serrano, J. Sanchez, S. Govindaraj, J. Bedkowski, R. Roda, E. Silva, and S. Ourevitch. Introduction to the use of robotic tools for search and rescue. In *Search and Rescue Robotics*, chapter 1. IntechOpen, Rijeka, 2017.

[5] V. Dimitrijević, M. Milosavljević, and M. Marković. A branch and bound algorithm for solving a generalized traveling salesman problem. *Publikacije Elektrotehničkog fakulteta. Serija Matematika*, (7):31–35, 1996.

[6] L. Dubský. Hexaly optimizer for multi-goal problems. Master's thesis in progress, Department of Cybernetics, Czech Technical University in Prague, 2025.

[7] M. Fischetti, J. J. Salazar González, and P. Toth. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. *Operations Research*, 45(3):378–394, 1997.

[8] N. Garg, G. Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the group steiner tree problem. *Journal of Algorithms*, 37(1):66–84, 2000.

[9] D. Gutin, Gregory ; Karapetyan. Generalized traveling salesman problem reduction algorithms. *Algorithmic Operations Research*, 4(2):144–154, 2009.

[10] Z. Hao, H. Huang, and R. Cai. *Bio-inspired Algorithms for TSP and Generalized TSP*. INTECH Open Access Publisher, 2008.

[11] B. Hu and G. R. Raidl. Effective neighborhood structures for the generalized traveling salesman problem. In J. van Hemert and C. Cotta, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 36–47, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[12] D. Huamanchahua, D. Yalli-Villa, A. Bello-Merlo, and J. Macuri-Vasquez. Ground robots for inspection and monitoring: A state-of-the-art review. In *2021 IEEE 12th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 0768–0774, 2021.

[13] O. Jellouli. Intelligent dynamic programming for the generalised travelling salesman problem. In *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236)*, volume 4, pages 2765–2768 vol.4, 2001.

[14] E. Koutsoupias, C. Papadimitriou, and M. Yannakakis. Searching a fixed graph. In F. Meyer and B. Monien, editors, *Automata, Languages and Programming*, pages 280–289, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[15] 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*, 84:178–187, 2017.

[16] M. Kulich and L. Přeučil. Multirobot search for a stationary object placed in a known environment with a combination of grasp and vnd. *International Transactions in Operational Research*, 29(2):805–836, 2022.

[17] M. Kulich, L. Přeučil, and J. J. M. Bront. Single robot search for a stationary object in an unknown environment. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5830–5835, 2014.

[18] M. Kulich, L. Přeučil, and J. J. Miranda Bront. On multi-robot search for a stationary object. In *2017 European Conference on Mobile Robots (ECMR)*, pages 1–6, 2017.

[19] G. Laporte and Y. N. and. Generalized travelling salesman problem through n sets of nodes: An integer programming approach. *INFOR: Information Systems and Operational Research*, 21(1):61–75, 1983.

[20] F. Martínez, A. J. Rueda, and F. R. Feito. A new algorithm for computing boolean operations on polygons. *Computers & Geosciences*, 35(6):1177–1185, 2009.

[21] J. Mikula. Search for a static object in a known environment. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, January 2021. Supervisor: RNDr. Miroslav Kulich, Ph.D.

[22] J. Mikula. Anytime metaheuristic framework for global route optimization in expected-time mobile search. Manuscript in review, 2025.

[23] J. Mikula and M. Kulich. Solving the traveling delivery person problem with limited computational time. *Central European Journal of Operations Research*, 30(4):1451–1481, 2022.

[24] J. Mikula and M. Kulich. Optimizing mesh to improve the triangular expansion algorithm for computing visibility regions. *SN Computer Science*, 5, 02 2024.

[25] N. Mladenovic, D. Urosevic, and S. Hanafi. Variable neighborhood search for the travelling deliveryman problem. *4OR*, 11, 03 2013.

[26] C. E. Noon and J. C. Bean. A lagrangian based approach for the asymmetric generalized traveling salesman problem. *Operations Research*, 39(4):623–632, 1991.

[27] V. Ostapovych. A metaheuristics for the graph search problem with order-dependent weights. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, May 2024. Supervisor: RNDr. Miroslav Kulich, Ph.D.

[28] P. C. Pop, O. Cosma, C. Sabo, and C. P. Sitar. A comprehensive survey on the generalized traveling salesman problem. *European Journal of Operational Research*, 314(3):819–835, 2024.

[29] S. Ropke and D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40:455–472, 11 2006.

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

[31] A. Sarmiento, R. Murrieta, and S. Hutchinson. An efficient strategy for rapidly finding an object in a polygonal world. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, volume 2, pages 1153–1158 vol.2, 2003.

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

[33] P. Shaw. A new local search algorithm providing high quality solutions to vehicle routing problems. 1997.

[34] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In M. Maher and J.-F. Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[35] S. L. Smith and F. Imeson. Glns: An effective large neighborhood search heuristic for the generalized traveling salesman problem. *Computers & Operations Research*, 87:1–19, 2017.

[36] L. V. Snyder and M. S. Daskin. A random-key genetic algorithm for the generalized traveling salesman problem. *European Journal of Operational Research*, 174(1):38–53, 2006.

[37] S. S. Srivastava, S. Kumar, R. C. Garg, and P. Sen. Generalized travelling salesman problem through n sets of nodes. *CORS Journal*, 7(2):97–101, 1969. Accessed: 2025-05-22.

[38] B. R. Vatti. A generic solution to polygon clipping. *Commun. ACM*, 35(7):56–63, July 1992.

[39] J. Yang, X. Shi, M. Marchese, and Y. Liang. An ant colony optimization method for generalized tsp problem. *Progress in Natural Science*, 18(11):1417–1422, 2008.

Appendix C

Attached files

The attached folder contains the solver along with some example instances. The folder has got the following structure. Please refer to the `README.md` for more information.

```
/
├── conda/
├── data/
├── example_instances/
├── experiments/
├── visis/
├── build.bash
└── README.md
```

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Kubišta Daniel**	Personal ID number:	**492344**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Cybernetics and Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

The Generalized Travelling Deliveryman Problem

Master’s thesis title in Czech:

Zobecněný problém obchodního doručovatele

Guidelines:

In the Generalized Traveling Salesman Problem (GTSP), we are given a set of cities grouped into possibly intersecting clusters. The objective is to find a closed path of minimum cost that visits at least one city in each cluster. The Traveling Deliveryman Problem (TDP), on the other hand, asks for a path visiting all cities exactly once minimizing the sum of waiting times in all cities. In the thesis, a new problem combining GTSP a TDP will be studied. The student will proceed in the following steps:

1. Familiarize yourself with the GLNS method [4] for solving the Generalized Traveling Salesman Problem.
2. Familiarize yourself with methods for solving the Traveling Deliveryman Problem [2,3,5,6].
3. Design a method for solving the Generalized Traveling Deliveryman Problem based on the methods mentioned above.
4. Perform experimental verification of the proposed method. Document and discuss the results.

Bibliography / sources:

[1] Mikula, J., and Kulich, M. (2022). Towards a continuous solution of the d-visibility watchman route problem in a polygon with holes. IEEE Robotics and Automation Letters, 7(3), 5934–5941. https://doi.org/10.1109/LRA.2022.3159824
[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. https://doi.org/10.1111/itor.12794
[3] Mikula, J., and Kulich, M. (2022). Solving the traveling delivery person problem with limited computational time. Central European Journal of Operations Research, 1–31. https://doi.org/10.1007/S10100-021-00793-Y
[4] Stephen L. Smith, Frank Imeson, GLNS: An effective large neighborhood search heuristic for the Generalized Traveling Salesman Problem, Computers & Operations Research, Volume 87, 2017, Pages 1-19, ISSN 0305-0548, https://doi.org/10.1016/j.cor.2017.05.010.
[5] Mualla Gonca Avci, Mustafa Avci, An adaptive large neighborhood search approach for multiple traveling repairman problem with profits, Computers & Operations Research, Volume 111, 2019, Pages 367-385, ISSN 0305-0548, https://doi.org/10.1016/j.cor.2019.07.012.
[6] Cédric Pralet, Iterated Maximum Large Neighborhood Search for the Traveling Salesman Problem with Time Windows and its Time-dependent Version, Computers & Operations Research, Volume 150, 2023, 106078, ISSN 0305-0548, https://doi.org/10.1016/j.cor.2022.106078.

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: **29.01.2025** Deadline for master's thesis submission: ___________

Assignment valid until: **20.09.2026**

prof. Dr. Ing. Jan Kybic 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

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s):	Kubišta Daniel
Personal number:	492344
Programme name:	Cybernetics and Robotics

declare that I have elaborated the master’s thesis entitled

The Generalized Travelling Deliveryman Problem

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor’s and Continuing Master’s Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 23.05.2025

Bc. Daniel Kubišta
....
student's signature