Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Search for sources of gamma radiation

Bc. David Woller

Supervisor: RNDr. Miroslav Kulich, Ph.D.
May 2019

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Woller David**	Personal ID number:	**439600**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Control Engineering**		
Study program:	**Cybernetics and Robotics**		
Branch of study:	**Cybernetics and Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

Search for sources of gamma radiation

Master’s thesis title in Czech:

Hledání zdrojů gama záření

Guidelines:

Consider a limited space in which one or more sources of gamma radiation may be present. The task is to find a plan for a mobile robot to explore the space as quickly as possible and determine the location of all resources. The task is complicated by the fact that the radiation source cannot be detected from one place, but it is necessary to move around it on a circular arc. The student's tasks are the following:
1. To get acquainted with metaheuristic algorithms for routing problems, especially the article [1].
2. Implement the algorithm [1] and compare the implementation properties with the results from the article.
3. Propose a modification of the algorithm [1] for the search of radiation sources and implement this modification.
4. Experimentally evaluate the properties of the implemented algorithm. Describe and discuss the results obtained.

Bibliography / sources:

[1] 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.
[2] R. Martí, P. M. Pardalos, M. G. C. Resende: Handbook of Heuristics. Springer 2018, ISBN 978-3-319-07123-7
[3] M. Gendreau & Jean-Yves Potvin (ed.), 2019. 'Handbook of Metaheuristics,' International Series in Operations Research and Management Science, Springer, edition 3, number 978-3-319-91086-4, December.

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: **31.01.2019** Deadline for master's thesis submission: **24.05.2019**

Assignment valid until:
by the end of summer semester 2019/2020

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

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

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to thank my supervisor, RNDr. Miroslav Kulich, Ph.D., for guidance and my parents for their support throughout my whole life.

Declaration

Abstract

This thesis documents the application of a metaheuristic algorithm GLNS on practical task - search for sources of gamma radiation. An individual source of radiation can be precisely located by a robot carrying a detector if it passes nearby through a trajectory segment, represented by a circular arc. As there can be more than one potential source of radiation and there are many valid arcs for each of them, the trajectory planning task can be after discretization formulated as Generalized Traveling Salesman Problem with m sets (sources) and n vertices (arcs), partitioned into these m sets. The goal is to find a tour, that visits each set exactly once and has a minimum length.

For this purpose, an adaptive large neighborhood search algorithm GLNS is modified and implemented. Additionally, the planner is adapted to plan with cubic curves with predefined minimal curvature as edges, and a procedure for fast estimation of their weights is presented. Moreover, we propose a new intensification procedure called DenseOpt, which serves to improve the quality of the solution obtained from the discrete planner by performing a local search in the continuous space.

Keywords: Generalized Traveling Salesman Problem, Adaptive Large Neighborhood Search, GLNS

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Czech Technical University in Prague
Czech Institute of Informatics, Robotics, and Cybernetics
Jugoslávských partyzánů 1580/3
16000 Prague 6

Abstrakt

Práce se zabývá aplikací metaheuristického algoritmu GLNS v praktické úloze - hledání zdrojů gama záření. Přesná pozice zdroje záření může být určena robotem nesoucím detektor, pokud projede v jeho blízkém okolí po trajektorii odpovídající kruhovému oblouku. V obecném případě je zdrojů radiace více a každý z nich může být detekován průjezdem mnoha různými oblouky, což umožňuje po vhodné diskretizaci formulaci úlohy jako zobecněného problému obchodního cestujícího. Je dáno m množin (zdrojů radiace) a n vrcholů (kruhových oblouků), rozdělených do m množin. Cílem je najít takovou sekvenci vrcholů, aby výsledná trajektorie byla co nejkratší.

Za tímto účelem byl modifikován a implementován algoritmus GLNS, který je založen na heuristickém adaptivním prohledáváním širokého okolí aktuálního řešení problému. Plánovač je rozšířen o možnost plánování s kubickými křivkami jakožto hranami, které navíc dodržují definovanou maximální křivost. Zároveň je představen postup, jak pro účely plánování odhadovat jejich délku, namísto časově náročnější přesné kalkulace. Dále je do algoritmu přidána intenzifikační metoda zvaná DenseOpt, která lokálně optimalizuje výslednou trajektorii získanou diskrétním plánovačem, čehož je docíleno prohledáváním spojitého prostoru přípustných vrcholů v blízkém okolí vrcholů diskrétního řešení.

Klíčová slova: Zobecněný problém obchodního cestujícího

Překlad názvu: Hledání zdrojů gama záření

Contents

Figures

Tables

Chapter 1

Introduction

The ultimate goal of the task motivating this thesis is to detect sources of gamma radiation in a previously unknown area, such as a place of a nuclear accident. The detection is to be carried out as quickly and precisely as possible, with subsequent use of UAV (Unmanned Aerial Vehicle) and UGV (Unmanned Ground Vehicle) [GZ17].

First, the area is mapped by the UAV carrying a photogrammetry multi-sensor system and a gamma detector. The objective of this phase is to quickly build a 3D map of the area surface and to locate regions with potential presence of radiation sources. As the UAV can carry only a lightweight gamma detector with limited accuracy and it also has to keep certain altitude above the terrain, it can only locate radiation sources with precision up to several meters. For more details, see [GL18].

Second, a UGV with more accurate gamma detector is deployed to inspect all previously discovered regions of interest and precisely locate sources of radiation. As the UGV motion is substantially slower than that of the UAV, it is crucial to optimize its trajectory while taking advantage of all of the already acquired information.

A position of a single radiation source can be reliably detected by following a specific trajectory in its close neighborhood. If task-specific conditions described in Section 2.1.2 are met, this partial trajectory is representable by a circular arc. Each region can be covered in many different ways, i.e., many valid circular arcs can be sampled. Besides that, the ordering of regions exploration is not fixed and is also subject to optimization. Therefore, after appropriate discrete coverage of individual regions, the planning problem can be reformulated as the well known NP-hard Generalized Travelling Salesman Problem (GTSP). Given n nodes (circular arcs) divided into m sets (regions of interest), the aim is to find such a trajectory, that passes through exactly one node from each set and is optimal with respect to some criterion, e.g. minimum length. This modified variant of the GTSP with circular arcs as vertices is from now on referred to as the $\mathrm{GTSP}_{\mathrm{arc}}$ and modified GLNS solving it is being called $\mathrm{GLNS}_{\mathrm{arc}}$.

This thesis introduces $\mathrm{GLNS}_{\mathrm{arc}}$ algorithm solving the $\mathrm{GTSP}_{\mathrm{arc}}$, evaluates its performance in several experiments, and proposes two major improvements to the $\mathrm{GLNS}_{\mathrm{arc}}$ functionality. It is structured as follows. Chapter 1 describes the motivation and presents alternative approaches. Chapter 2 is dedicated to the theoretical aspects of this thesis. It provides formal definitions of the GTSP and the $\mathrm{GTSP}_{\mathrm{arc}}$ in Section 2.1. Section 2.2 describes original GLNS and Section 2.3 covers necessary modifications of GLNS towards solving the $\mathrm{GTSP}_{\mathrm{arc}}$ and thus transformation to $\mathrm{GLNS}_{\mathrm{arc}}$. Section 2.4 then introduces proposed improvements of $\mathrm{GLNS}_{\mathrm{arc}}$, such as time-efficient planning with cubic edges and intensification procedure called DenseOpt. Chapter 3 describes experimental setup and presents obtained results. It covers the performance of $\mathrm{GLNS}_{\mathrm{arc}}$ compared to the original GLNS implementation, time demands of planning with cubic edges and efficiency of possible solutions and beneficial effects of DenseOpt. Chapter 4 then summarizes work done, coherently interprets experimental results and proposes areas of future research.

1.1 State of the art

The GTSP is a combinatorial optimization problem extensively studied in operations research with many practical applications, such as location-routing problems, material flow system design, post-box collection, stochastic vehicle routing, arc routing or timetabling [AVS96]. There is no polynomial time known exact solution for this problem and new approaches on how to find acceptable solutions to large problems in reasonable time are still being proposed.
Over the years, various approaches have been developed. Let's start with the optimal solver described in [NB91]. This approach utilizes a Lagrangian relaxation to compute a lower bound on the optimal solution weight and heuristically determines an upper bound. Then, vertices and edges that are guaranteed not to be present in the optimal solution are removed, and the solution is then obtained using a branch-and-bound procedure. A maximal problem successfully solved contains only 104 vertices. However, the experiments complementing the article were carried out in 1991 with corresponding technology.
With the use of branch-and-cut [GT97] algorithm, maximal size of the optimally solved problem later grew to 442 vertices.
As for heuristical non-optimal approaches - there are several solvers using genetic and memetic algorithms. The best solver from this family is called GK [GK10] and it performs well on problems with up to 200 sets (while finding solutions close to those best known in minutes).
Another commonly used approach utilizes reduction of the GTSP to an asymmetric TSP problem instance with the same amount of nodes [NB93] and subsequent solving of this TSP problem with a heuristic TSP solver. This approach was most recently adapted in GLKH solver [Hel15], which combines reduction to TSP with TSP LKH solver [Hel00]. In terms of the quality of the best solution found, GLKH often outperforms GK, although it can be up to two orders slower on the same problem instance. GLKH was tested on problems with up to 17180 sets and 85900 vertices - larger than any of the remaining solvers mentioned.

Finally, there is GLNS [SI17]. GLNS performs adaptive large neighborhood search and is designed directly for the GTSP problems. Thus no reduction to TSP is used. The basic idea of a large neighborhood search is to iteratively apply constructive and destructive heuristics on the current solution, while there can be more than one heuristic of each kind. The search is adaptive, i.e., the previous performance of individual heuristics affects the probability of their selection in future iterations [PR18]. According to [SI17], GLNS outperforms both GK and GLKH, especially on non-metric or non-clustered GTSP instances. However, GLNS is tested on instances of up to 10000 vertices and it is not suitable for significantly larger problems due to memory requirements.

Chapter 2

Methods

2.1 Planning task formulation

This section provides a formal definition of the GTSP and also elaborates on potential problems arising from differences between the practical task of searching for radiation sources and the general, non-specific definition.

Assume a weighted graph $G = (V, E, w)$ and a partition of V into m sets $P_V = \{V_1, ..., V_m\}$, where

- V is a set of n vertices
- $V_i \cap V_j = \emptyset$ for all $i \neq j$
- $\bigcup_{i=1}^{m} V_i = V$
- E is a set of edges such that all vertices are connected, apart from vertices from the same set
- w is a mapping assigning a weight to each edge $w : E \to \mathbb{R}$

Lets also define tour T over graph G as a closed sequence of vertices and edges $T = (v_0, e_0, v_1, e_1, ...v_{m-1}, e_{m-1})$, where each edge connects two consecutive vertices - $e_i = (v_{i-1}, v_i)$ and $e_{m-1} = (v_{m-1}, v_0)$. A set of vertices present in the tour T is denoted as V_T, a set of edges then E_T.

The GTSP objective is to find a tour in G that contains exactly one vertex from each set and has a minimum length, i.e., it minimizes the tour length $w(T)$ defined as

$$w(T) = \sum_{e \in E_T} w(e). \tag{2.1}$$

An example of a solved GTSP instance is shown in Figure 2.1, where vertices of the same color belong to the same set.

2.1.1 Task-specific differences

There are aspects of the planning task solved that prevent us from using the previously given the GTSP formulation and already implemented solvers. Due to the following, the formulation has to be slightly changed, and the solver appropriately modified.

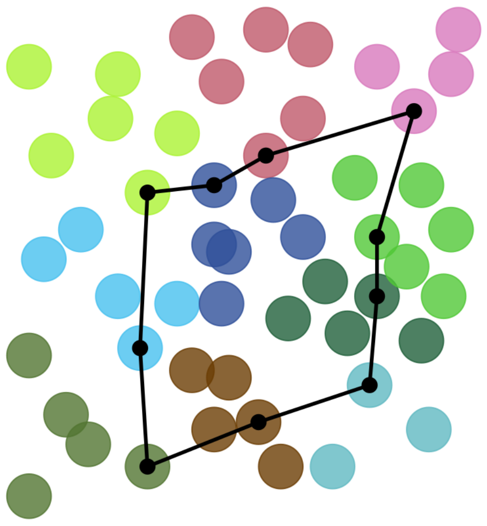


Figure 2.1: Solved GTSP instance

Vertex definition

Contrary to GTSP, where a vertex is a point, a single vertex in the GTSP$_{\mathrm{arc}}$ represents a circular arc - a special trajectory segment such that passing by it allows a precise radiation source detection in a corresponding region of interest. A minimal vertex representation consists of the following parameters:

Symbol	Parameter description
x, y	planar coordinates of circular arc center
r	arc radius
α	angle between x-axis and arc axis
ω	angular size of the arc

Table 2.1: Vertex parameters in the GTSP$_{\mathrm{arc}}$

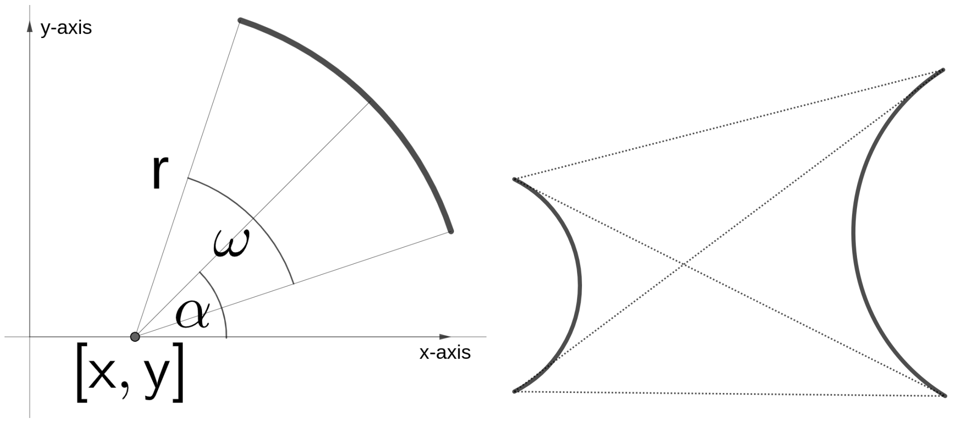
Parameters are also depicted in Figure 2.2a.

Vertex weighting

As each vertex represents a segment of a robot trajectory, its length influences the total trajectory cost. In the original GTSP implementation described in [SI17], only edges, not vertices, have weights assigned. Therefore this difference must be taken into consideration while implementing various metrics in the algorithm. In GLNS$_{\mathrm{arc}}$, weight w of each vertex is defined as $w = \omega r$.

Vertex connecting

There is no restriction on the direction, in which a vertex is to be passed, therefore connecting two vertices is ambiguous. The original algorithm considers only the possibility, that edge weights are dependent on vertex order



(a) : Parameters definition **(b) :** Vertex connecting

Figure 2.2: Vertices in the GTSP$_{\text{arc}}$

- an edge from a to b might have a different weight than an edge from b to a. However, in the GTSP$_{\text{arc}}$, even with a fixed order, there are still four ways, how to connect two consecutive vertices, as shown in Figure 2.2b. Therefore, GLNS solver has to be modified accordingly.

2.1.2 Vertex validity constraint

While detecting sources of radiation along circular arcs, only certain arcs are of use. The following constraint was provided as a part of this thesis assignment. The source is detectable by passing through a circular segment if the following condition is met:

$$\frac{\frac{I}{2r(r+d)(1-\cos(\frac{l}{2r}-|\theta|))+d^2+h^2}+c_B}{\frac{I}{d^2+h^2}+c_B} < K. \tag{2.2}$$

Parameters are described in Table 2.2.

Symbol	Parameter description
I	intensity of the radiation source
r	radius of circular arc
d	distance from source to the closest point of arc
l	arc length
θ	angle between arc axis and line from arc center to source
h	height of detector above terrain
c_B	background radiation
K	experimentally set constant

Table 2.2: Vertex validity constraint - parameters

If the arcs corresponding to the same radiation source are concentric and a potential source lies close to their center, this constraint limits only the maximal radius of these arcs.

2.2 GLNS description

GLNS is a GTSP solver introduced by S. L. Smith and F. Imeson [SI17] based on an adaptive large neighborhood search. This section gives a brief overview of its functioning, a full description of the algorithm, including thorough performance comparison with other approaches, can be found in [SI17].

2.2.1 Solver overview

GLNS implements an adaptive large neighborhood search, which is a meta-heuristic planning approach based on the iterative application of constructive and destructive procedures on a current solution. Adaptive means, that these procedures are being selected proportionally to their previous performance.

Algorithm 1 GLNS

```
1:  Input: A GTSP instance $(G, P_V)$
2:  Output: GTSP tour $T$
3:  for i = 1 to cold_restarts do
4:      $T \leftarrow$ initial_tour$(G, P_V)$
5:      $T_{best,i} \leftarrow T$
6:      repeat
7:          Select a removal heuristic $R$ and insertion heuristic $I$
8:          Select number of vertices to remove $N_r$
9:          $T_{new} \leftarrow T$
10:         Remove $N_r$ vertices from $T_{new}$ using $R$
11:         Insert $N_r$ vertices to $T_{new}$ using $I$, one from each set not visited
    by $T_{new}$
12:         Locally re-optimize $T_{new}$
13:         if $w(T_{new}) < w(T_{best,i})$ then
14:             $T_{best,i} \leftarrow T_{new}$
15:         end if
16:         if accept$(T_{new}, T)$ then
17:             $T \leftarrow T_{new}$
18:             Record improvement made by $R$ and $I$
19:         end if
20:     until stop criterion met
21:     Update selection weights of heuristics
22: end for
23: return tour $T_{best,i}$ that attains $min_{i \in \{1..\texttt{cold\_restarts}\}} w(T_{best,i})$
```

GLNS pseudocode is given in Algorithm 1. First, an initial random tour is generated (line 4). Then, the following process runs iteratively. A pair of a removal heuristic and an insertion heuristic is selected, according to their selection weights (line 7). These heuristics are then applied to remove and insert N_r vertices (lines 8 to 11), thus creating a modified tour T_{new}. The modified tour T_{new} is subject to local optimization techniques MoveOpt

and ReOpt (line 12) and is consequently accepted or declined, while using standard simulated annealing criterion (line 16). This process repeats until one of the stop criteria is met (line 20). After that, the planner updates selection weights of the heuristics (line 21) and either starts the whole process again in a new cold restart or returns the best tour found overall.
Each cold restart is split into two phases - initial descent and warm restarts. The initial descent ends after a certain fixed number of non-improving iterations. Then, there are several warm restarts, starting with the best solution currently found and with slightly higher simulated annealing temperature (which otherwise decreases with every iteration). This temperature is used in acceptance criterion (line 16), to allow for accepting non-improving tours with a small probability. Each warm restart also ends after a certain fixed number of non-improving iterations.

2.2.2 Initial tour construction

Two approaches are proposed for initial tour construction in GLNS.
Random insertion tour construction is based on random insertion heuristic. First, a randomly selected vertex $v \in V$ is added to T. The following step is then repeated $m-1$ times: a random set V_i is selected from $P_V \setminus P_T$ and such vertex $v_i \in V_i$ is picked and added to T, that minimizes the insertion cost given by Equation 2.3.
Random tour construction creates the initial tour completely randomly. All m sets in P_V are randomly shuffled, and one vertex is picked from each set uniformly randomly.
Random insertion tour construction is preferred in the fast planning mode defined in Section 2.2.7. Otherwise, random tour construction is used.

2.2.3 Insertion heuristics

Insertion heuristic is a function, that takes GTSP problem instance (G, P_V) and a partial tour $T = (V_T, E_T)$ as an input and returns an updated partial tour T visiting one additional set. There are four basic insertion heuristics described in GLNS - nearest, farthest random and cheapest. All of these heuristics follow the same framework - first, a set $V_i \in P_V \setminus P_T$ (i.e., newly visited set) is selected. Then, an edge $(x, y) \in E_T$ and vertex $v \in V_i$ minimizing the insertion cost

$$cost = w(x, v) + w(v, y) - w(x, y) \tag{2.3}$$

are found. Finally, edge (x, y) is deleted from E_T, edges (x, v) and (v, y) are added to E_T and v to V_T. The only difference between the individual heuristics is in the way, how the set V_i is picked. Before specifying that, it is necessary to define set-vertex distance.
Set-vertex distance needs to be precomputed $\forall V_i, i \in \{1, 2..., m\}$ and $\forall u \in V \setminus V_i$ and is given by

$$dist(V_i, u) = min_{v \in V_i}\{min\{w(v, u), w(u, v)\}\}. \tag{2.4}$$

Nearest insertion, picks such a set V_i that contains a vertex v with minimal distance to a vertex from the partial tour T, i.e.

$$V_i = argmin_{V_i \in P_V \setminus P_T} min_{u \in V_T} dist(V_i, u). \tag{2.5}$$

Farthest insertion picks such a set V_i, whose closest vertex to a vertex on partial tour T is at a maximal distance, i.e.

$$V_i = argmax_{V_i \in P_V \setminus P_T} min_{u \in V_T} dist(V_i, u). \tag{2.6}$$

Cheapest insertion picks the set V_i containing the vertex v that minimizes the insertion cost;

$$V_i = argmin_{V_i \in P_V \setminus P_T} min_{v \in V_i, (x,y) \in E_T} \{w(x, v) + w(v, y) - w(x, y)\}. \tag{2.7}$$

In **Random insertion**, set V_i is picked uniformly randomly from $P_V \setminus P_T$. Insertion heuristics are further randomized by adding noise to calculated insertion cost;

$$cost = (1 + rand)(w(x, v) + w(v, y) - w(x, y)), \tag{2.8}$$

where $rand \in [0, \eta]$ is a uniformly randomly sampled number, and η is so-called additive noise.
Set selection in the nearest, farthest and random insertion heuristic can be generalized in a single framework while using an **unnormalized probability mass function** Λ. It is defined as

$$\Lambda = [\lambda^0, \lambda^1, ..., \lambda^{l-1}], \tag{2.9}$$

where l is the number of sets in $P_V \setminus P_T$. To select a set, an index $k \in \{1, ..., l\}$ is randomly selected while using Λ and a set $V_i \in P_V \setminus P_T$ with k-th smallest distance to the tour is picked. Depending on λ, this procedure is equivalent to nearest ($\lambda = 0$), random ($\lambda = 1$) or farthest ($\lambda = \infty$) insertion. Also, by selecting other values of λ, hybrid versions of these heuristics are added to the algorithm.
The bank of insertion heuristics contains the previously listed heuristics with all combinations of predefined values of λ and η, see Table 2.4.

2.2.4 Removal heuristics

Given a tour $T = (v_0, e_0, v_1, e_1, ...v_{m-1}, e_{m-1})$, removal heuristics remove N_r vertices from T and return a closed partial tour. There are four removal heuristics in GLNS: random removal, worst removal, distance removal, and segment removal. First three heuristics are designed for removing one vertex at a time, so to remove N_r vertices, such heuristic is simply applied N_r times.

Worst removal removes such vertex v_j from T that maximizes removal cost

$$r_j = w(e_{j-1}) + w(e_j) - w(v_{j-1}, v_{j+1}). \tag{2.10}$$

Therefore, a vertex resulting in the greatest reduction in tour length is removed, together with edges e_{j-1} and e_j. To make the tour closed again, edge (v_{j-1}, v_{j+1}) is added.
Random removal removes a vertex v_j selected from T uniformly randomly. Again, edges e_{j-1} and e_j are removed and edge (v_{j-1}, v_{j+1}) is added.
Distance removal attempts to remove vertices, that are close to each other. The procedure is described in Algorithm 2. The first vertex to be removed is picked randomly and added to a set called $V_{removed}$ (line 4). The remaining $N_r - 1$ vertices are selected in the following way. For each of them, a vertex v_{seed} is randomly picked from $V_{removed}$ (line 6). Then, the closest vertex to v_{seed} is removed from tour T (line 7) and added to $V_{removed}$.

Algorithm 2 Distance removal

```
 1: Input: GTSP tour $T$, number of vertices to remove $N_r$
 2: Output: Updated GTSP tour $T$
 3: $V_{removed} = \emptyset$
 4: Uniformly randomly remove a vertex $v$ from $T$ and add it to $V_{removed}$
 5: for $i = 1$ to $N_r - 1$ do
 6:     Select vertex $v_{seed}$ from $V_{removed}$, uniformly randomly
 7:     Remove such vertex $v_j$ from $T$, that minimizes $r_j = min\{w(v_{seed}, v_j), w(v_j, v_{seed})\}$
 8:     Add $v_j$ to $V_{removed}$
 9: end for
10: return $T$
```

Each time a vertex v_j is removed from T, edges e_{j-1}, e_j are removed as well and edge (v_{j-1}, v_{j+1}) is added to T.

Segment removal removes a continuous segment of the tour of the length N_r, starting with uniformly randomly selected vertex v_j. All vertices from v_j to v_{j+N_r-1} and all edges between the first and last vertex are removed and an edge from v_{j-1} to v_{j+N_r} is added.
Similarly to the insertion heuristics, distance, random, and worst removals can be generalized using a single framework and the probability mass function Λ defined in Equation 2.9. Let V_T denote a set of vertices present in T. To find a vertex to be removed, an index $k \in \{1, 2, ...|V_T|\}$ is randomly selected, according to $\Lambda = [\lambda^0, \lambda^1, ..., \lambda^{|V_T|-1}]$. Then, the vertex $v_j \in V_T$ with the k-th smallest value r_j is removed, together with removing edges e_{j-1}, e_j and adding edge (v_{j-1}, v_{j+1}).
This way, random removal can be performed by setting $\lambda = 1$, worst removal for $\lambda = \infty$ and closest vertex selection (utilized in distance removal) for $\lambda = 0$, along with custom randomized heuristics for other values.

Adaptive weights

GLNS maintains a bank of insertion and removal heuristics. Each heuristic is assigned a weight, which is initialized to 1. At each iteration, one insertion and one removal heuristic are selected from the bank, using a roulette wheel selection mechanism according to the weights assigned. With the selected heuristics, the current tour T is destroyed, and a new tour T_{new} is created. The instant score of the heuristics used is evaluated as

$$score = max\{\frac{w(T) - w(T_{new})}{w(T)}, 0\}.$$

At the end of each cold restart, the overall score $score_{trial}$ for each heuristic in that cold restart is obtained by averaging all instant scores.

Then, the weight w of each heuristic is updated as

$$w = \epsilon w + (1 - \epsilon) score_{trial}, \tag{2.11}$$

ϵ being a fixed constant.

2.2.5 Local optimizations

There are two local optimizations present in GLNS - MoveOpt and ReOpt.
MoveOpt attempts to optimize the order of the sets by randomly removing a vertex from the current tour and reinserting such a vertex from the same set, that minimizes the insertion cost. Thus, set order in the tour might be changed.
ReOpt re-optimizes the choice of vertices in all sets, while keeping the set order fixed. It performs a search for the shortest cycle across all sets in the given order. This search can be executed exactly while using an optimal graph search algorithm such as A* with an admissible heuristic. In GLNS$_{\mathrm{arc}}$, a simple breadth-first search is employed.

2.2.6 Acceptance and stopping criteria

- Acceptance criteria

  At each iteration, a new tour T_{new} is created and accepted or discarded. For this purpose, GLNS uses a standard simulated annealing acceptance criterion. Given a temperature $\mathcal{T}$, new tour T_{new} is accepted with probability $min\{exp(\frac{w(T)-w(T_{new})}{\mathcal{T}}), 1\}$, where T is the best tour found in the trial so far. The temperature T is decreased at every iteration as $\mathcal{T} = c\mathcal{T}$, where $c < 1$ is so called cooling rate.

- Stopping criteria

  The GLNS planning process consists of several phases - one or more cold restarts, each of them consisting of an initial descent phase and several warm restarts. Stopping criteria of these phases are as follows.
  Initial descent and warm restarts both end, when the best solution found is not improved for a fixed number of consecutive iterations.
  Planning process ends either after last cold restart, or after meeting one of the alternate stopping criteria: maximum planning time is exceeded or a tour with the weight lower than some given bound is found.

2.2.7 Modes of operation

GLNS utilizes several fixed parameters, all of which are described in Table 2.3. There are three modes of operation proposed in [SI17] - fast, medium, and slow. Each of these modes has different settings in terms of runtime, parameter values, and internally used subroutines. Table 2.4 shows parameter settings for these modes.
Most notably, the individual modes differ in the number of cold restarts, warm restarts, and the number of non-improving iterations needed before terminating current warm restart. Also, the fast mode does not utilize the computationally expensive cheapest insertion heuristic and uses random insertion tour construction, thus starts with greedily created better quality solution than default or slow mode.

Parameter or constant	Description
m	number of sets
Iterations	number of iterations used for cooling rate c determination - c is set so that after this number of iterations, tour with $p_2\%$ higher cost is accepted with the probability of 1/2
Warm iterations - first	after this number of non-improving iterations, warm restart ends
Warm iterations - last	after this number of non-improving iterations, an initial descent or warm restart starting with initial improvement ends
Initial acceptance p_1	constant used for acceptance temperature initialisation; in the first trial, a tour with $p_1\%$ higher cost than the current best is accepted with the probability of 1/2
Final acceptance p_2	constant used for cooling rate c determination
Warm restart acceptance p_3	constant used for acceptance temperature re-initialisation; at the beginning of each warm trial, a tour with $p_3\%$ higher cost than the current best is accepted with the probability of 1/2
Reaction factor ϵ	constant used in updating heuristic weights
Initial tour construction	random tour or random insertion tour construction
Maximum removals	maximal number of vertices to remove in each iteration
λ values	values for building probability functions in generalized heuristics frameworks
Additive noise η	values of additive noise used in insertion heuristics

Table 2.3: GLNS parameters - description

	GLNS mode		
Parameter	Fast	Medium	Slow
Cold restarts	3	5	10
Warm restarts	2	3	5
Iterations	$60m$	$60m$	$150m$
Warm iterations - first	$10m$	$15m$	$25m$
Warm iterations - last	$15m$	$30m$	$50m$
p_1 (%)	5	5	5
p_2 (%)	0.05	0.05	0.05
p_3 (%)	0.5	0.5	0.5
ϵ	0.5	0.5	0.5
Tour init.	Random insert	Random	Random
Max. removals	$\min\{20,\ 0.1m\}$	$\min\{100,\ 0.3m\}$	$0.4m$
Iters. of MoveOpt	$\min\{20,\ 0.1m\}$	$\min\{100,\ 0.3m\}$	$0.4m$
Cheapest insertion	No	Yes	Yes
Insertion λ vals.	$(0, 1/2, 1/\sqrt{2}), 1, \sqrt{2}, 2, \infty)$		
Distance removal λ vals.	$(1/\sqrt{2}, 1, \sqrt{2}, 2, \infty)$		
Worst removal λ vals.	$(1/\sqrt{2}, 1, \sqrt{2}, 2, \infty)$		
Additive noise η vals.	$(0, 0.25, 0.75)$		

Table 2.4: GLNS modes of operation - settings

2.3 Proposed GLNS modifications towards GLNS$_{\text{arc}}$

As described in Section 2.1.1, graph vertex in the GTSP definition corresponds to a circular arc in the GTSP$_{\text{arc}}$. This arc represents a part of a trajectory and has certain properties, that have to be taken into account while employing GLNS to solve the GTSP$_{\text{arc}}$. The main issues arise from the fact, that the arc has a nonzero length and that connecting two vertices is ambiguous. Necessary modifications to the algorithm solving these issues are described in detail in this section.

Vertex duplication

In Section 2.1.1, a vertex was described by the following tuple of parameters - $\langle x, y, r, \alpha, \omega \rangle$. This representation is sufficient for problem formulation but impractical for implementation, as it requires distinguishing between various ways of vertex connecting. Instead of doing that, an additional parameter $sign \in \{\pm 1\}$ is added. This parameter determines, in which direction the vertex is to be passed through (-1 for clockwise passage, +1 for anticlockwise). Naturally, this doubles the total amount of vertices in our GLNS$_{\text{arc}}$ implementation, as each vertex is inserted with both possible *sign* values. On the other hand, the problem with edge connecting is solved, because the original GLNS allows the possibility, that weights $w(v_1, v_2)$ and $w(v_2, v_1)$ differ, which is the case now.

Tour weight

Let $T = (v_0, e_0, v_1, e_1, ..., v_{m-1}, e_{m-1})$ be a tour and $w(T)$ its weight. In GLNS$_{\text{arc}}$, this weight is calculated as

$$w(T) = \sum_{i=0}^{m-1} w(e_i) + \boldsymbol{\sum_{j=0}^{m-1} w(v_j)}, \tag{2.12}$$

where v_i, e_j are vertices and edges from T and $w(v)$, $w(e)$ their respective weights. The bold expression in Equation 2.12 is newly added in GLNS$_{\text{arc}}$.

Cheapest insertion and unified insertions

In all insertion heuristics, the insertion cost of a vertex $v_{new} \in V_i$, $V_i \in P_V \backslash P_T$ is minimized. In cheapest insertion, this cost is minimized to select both V_i and v_{new}, whereas in remaining three unified insertions, V_i is already selected as described in Section 2.2.3 and only v_{new} is sought. In all cases, the insertion cost c_{insert} has to be modified as follows:

$$c_{insert} = w(v_j, v_{new}) + w(v_{new}, v_{j+1}) - w(e_j) + \boldsymbol{w(v_{new})}. \tag{2.13}$$

Here, v_j and v_{j+1} are two consecutive vertices in T before insertion and $w(v_j, v_{new})$, $w(v_{new}, v_{j+1})$ and $w(e_j)$ weights of corresponding edges.

Worst removal

Worst removal removes such vertex $v_j \in V_T$, that maximizes the removal cost c_{remove}. GLNS$_{\text{arc}}$ modification is again rather straightforward:

$$c_{remove} = w(e_{j-1}) + w(e_j) - w(v_{j-1}, v_{j+1}) + \boldsymbol{w(v_j)}. \tag{2.14}$$

ReOpt

Re-Opt subroutine attempts to optimize the choice of vertices while keeping the set order fixed. This is achieved by performing a graph search through all sets, in which only edges between two consecutive sets are considered. When expanding from vertex $x \in V_i$ to vertex $y \in V_{i+1}$, current score in y is calculated as

$$score(y) = score(x) + w(x, y) + \boldsymbol{w(y)}. \tag{2.15}$$

Also, first vertex $a \in V_1$ is initialized with $\boldsymbol{score(a) = w(a)}$, instead of zero.

MoveOpt

MoveOpt subroutine attempts to optimize the sets order by randomly removing a vertex v_i from a tour T and reinserting another vertex v_j from the same set to any position in the tour so that the insertion cost is minimized. This cost c_{insert} is modified the same way as in cheapest and unified insertions, i.e.

$$c_{insert} = w(v_j, v_{new}) + w(v_{new}, v_{j+1}) - w(e_j) + \boldsymbol{w(v_{new})}. \tag{2.16}$$

Remarks

Some parts of GLNS were not formally modified, although the original idea behind them might have changed in GLNS$_{\text{arc}}$ due to task reformulation.
Set-vertex distance from a set V to a vertex u is still defined as

$$dist(V, u) = min_{v_i \in V}(min(w(u, v_i), w(v_i, u))). \tag{2.17}$$

In GLNS$_{\text{arc}}$, these distances are precomputed after vertex duplication. Therefore the value obtained corresponds to the shortest path to or from u to V, no matter the *sign* of u, v (=their orientation) or the edge (u, v) direction.
A different situation arises in **distance removal**, described in Algorithm 2. The motivation is to remove vertices from a current tour T, which are "close to each other" At each iteration, a vertex v_{seed} is selected randomly from the set of already removed vertices $V_{removed}$. The next vertex v_j to be removed from T is obtained as

$$v_j = argmin_{v_j \in T}(min(w(v_{seed}, v_j), w(v_j, v_{seed}))). \tag{2.18}$$

Here, GLNS$_{\text{arc}}$ considers only edges between vertices v_{seed}, v_j and not their oppositely oriented variants available in the GTSP$_{\text{arc}}$ instance, as these variants are not present in T.

2.4 Additional improvements

Apart from modifying GLNS to solve the GTSP$_{\text{arc}}$ and implementing it as GLNS$_{\text{arc}}$, further improvements were made to obtain a more versatile solution. These improvements involve planning with realistically weighted edges (other than primarily used straight lines), precomputing or estimating them in a reasonable time and an the intensification procedure, which further improves the solution obtained for discretely defined GTSP$_{\text{arc}}$ instance.

2.4.1 Cubic interpolation of edges

In the default mode, the GLNS$_{\text{arc}}$ evaluates edge weight from a vertex a to a vertex b as the Cartesian distance between their endpoints, i.e., the edge is interpolated by a straight line. For various reasons, this approach is not very realistic. The resulting trajectory is not smooth - it does not have defined derivatives at every point and thus can't be executed without stopping the robot and rotating in place. Such maneuver can be performed only by a holonomic robot, and the corresponding edge length is no longer proportional to the time required for passing it, as there may be either lot of rotational movement needed or none at all.
As an alternative, it is possible to interpolate edges in GLNS$_{\text{arc}}$ by a single segment of Ferguson cubic curve [YQ99]. This curve is defined by a start point $\boldsymbol{P_0}$, an endpoint $\boldsymbol{P_1}$ and tangent vectors in these points: $\boldsymbol{P_0'}$ and $\boldsymbol{P_1'}$, while the magnitude of these vectors affects the shape, length, and curvature

of the resulting curve. The general formula describing any point on the cubic curve $\boldsymbol{P}(t)$ is

$$\boldsymbol{P}(t) = \boldsymbol{a_0} + \boldsymbol{a_1} t + \boldsymbol{a_2} t^2 + \boldsymbol{a_3} t^3, \tag{2.19}$$

where $t \in \langle 0, 1 \rangle$, $\boldsymbol{a_i} = [a_{ix}, a_{iy}]^T$, $i = 0..3$.
Individual parameter tuples $\boldsymbol{a_0}..\boldsymbol{a_3}$ can be obtained from leading points and tangent vectors as

$$\begin{aligned}
\boldsymbol{a_0} &= \boldsymbol{P_0} \\
\boldsymbol{a_1} &= \boldsymbol{P'_0} \\
\boldsymbol{a_2} &= 3(\boldsymbol{P_1} - \boldsymbol{P_0}) - 2\boldsymbol{P'_0} - \boldsymbol{P'_1} \\
\boldsymbol{a_3} &= 2(\boldsymbol{P_0} - \boldsymbol{P_1}) + \boldsymbol{P'_0} + \boldsymbol{P'_1}
\end{aligned} \tag{2.20}$$

From here, $\boldsymbol{P}(t)$ can be expressed as

$$\boldsymbol{P}(t) = (1-3t^2+2t^3)\boldsymbol{P_0} + (3t^2-2t^3)\boldsymbol{P_1} + (t-2t^2+t^3)\boldsymbol{P'_0} + (-t^2+t^3)\boldsymbol{P'_1}. \tag{2.21}$$

Keeping predefined curvature

A cubic curve prescribed by Equation 2.21 is smooth, but it still may be unsuitable for a nonholonomic robot due to excessively sharp turns. Let's assume that the robot has a minimum turning radius of r_{min}. Then, the maximal permissible curvature K_{max} can be then obtained as

$$K_{max} = \frac{1}{r_{min}}. \tag{2.22}$$

The goal is to find the shortest possible edge with the curvature K such that $K < K_{max}$. This can be achieved by adjusting magnitude of vectors $\boldsymbol{P'_0}$, $\boldsymbol{P'_1}$, by finding appropriate scaling parameters k_0 and k_1,

$$\begin{aligned}
\boldsymbol{P'_0} &= k_0 \boldsymbol{P'_{0_{norm}}} \\
\boldsymbol{P'_1} &= k_1 \boldsymbol{P'_{1_{norm}}},
\end{aligned} \tag{2.23}$$

where $\boldsymbol{P'_{0_{norm}}}$, $\boldsymbol{P'_{1_{norm}}}$ are vectors of a unit length.
Now, let's examine the curvature K of a general cubic curve prescribed by Equation 2.21. In 2D, the curvature $K(t)$ of differentiable curve $\boldsymbol{P}(t) = [x(t), y(t)]^T$ can be calculated as follows:

$$K(t) = \frac{|x'(t)y''(t) - y'(t)x''(t)|}{(x'^2(t) + y'^2(t))^{\frac{3}{2}}}. \tag{2.24}$$

After substituting $x(t), y(t)$ by corresponding expressions and evaluating their derivatives as

$$\begin{aligned}
\boldsymbol{P'}(t) = [x'(t), y'(t)]^T &= \boldsymbol{a_1} + 2\boldsymbol{a_2} t + 3\boldsymbol{a_3} t^2 \\
\boldsymbol{P''}(t) = [x''(t), y''(t)]^T &= 2\boldsymbol{a_2} + 6\boldsymbol{a_3} t
\end{aligned} \tag{2.25}$$

the curvature $K(t)$ can be rewritten as

$$
\begin{aligned}
K(t) &= \frac{a}{b} \\
a &= |(2a_{2y} + 6a_{3y}t)(3a_{3x}t^2 + 2a_{2x}t + a_{1x}) \\
&\quad - (2a_{2x} + 6a_{2x}t)(3a_{3y}t^2 + 2a_{2y}t + a_{1y})| \\
b &= ((3a_{3x}t^2 + 2a_{2x}t + a_{1x})^2 + (3a_{3y}t^2 + 2a_{2y}t + a_{1y})^2)^{\frac{3}{2}}
\end{aligned}
\tag{2.26}
$$

Note that $\boldsymbol{a_1} = \boldsymbol{a_1}(k_0)$, $\boldsymbol{a_2} = \boldsymbol{a_2}(k_0, k_1)$ and $\boldsymbol{a_3} = \boldsymbol{a_3}(k_0, k_1)$, thus $K = K(k_0, k_1, t)$.
Now let's examine the curve length L, which is to be minimized. Using a symbolic representation $\boldsymbol{P}(t) = [x(t), y(t)]^T$, it is defined as

$$
L = \int_0^1 \sqrt{(\frac{dx(t)}{dt})^2 + (\frac{dy(t)}{dt})^2 dt}, \tag{2.27}
$$

and after substituting derivatives

$$
L = \int_0^1 \sqrt{(a_{1x} + 2a_{2x}t + 3a_{3x}t^2)^2 + (a_{1y} + 2a_{2y}t + 3a_{3y}t^2)^2 dt}. \tag{2.28}
$$

As the length L is an integral over a definite interval $t \in \langle 0, 1 \rangle$, $L = L(k_0, k_1)$.
Finally, let's proceed to solve the original task of appropriately scaling the cubic curve. Given points $\boldsymbol{P_0}$, $\boldsymbol{P_1}$, unit vectors $\boldsymbol{P_0'}_{\boldsymbol{norm}}$, $\boldsymbol{P_1'}_{\boldsymbol{norm}}$ and the maximal curvature K_{max}, the goal is to minimize $L = L(k_0, k_1)$ given by Equation 2.28, while it must hold that $max(K_{t \in \langle 0,1 \rangle}(t, k_0, k_1)) < K_{max}$. As extrema of Equation 2.26 couldn't be found analytically, this optimization problem is being solved numerically with the use of NLopt nonlinear optimization library [NLo].
In the GLNS$_{\text{arc}}$ implementation, parameters k_0 and k_1 are not treated separately, but as a single parameter $k = k_0 = k_1$. With the optimization algorithm used, tuning these parameters separately yielded less consistent results, occasionally missing a global optimum, thus producing an edge either too curved or too long. The evident drawback of this simplification is that with both parameters tied together, the edge obtained is no longer the shortest cubic curve possible.

2.4.2 Precomputing and estimating edge weights

Another problem is that before GLNS planning run, all edge weights need to be calculated. In case of straight lines, this is not an issue even for large problems, even though the number of edges can rise to n^2. However, previously described process of determining cubic edge parameters is significantly more time consuming, as shown in Figure 3.10. Therefore various approaches addressed in this section were examined so that that edge weights can be determined in a reasonable time.
A basic idea behind speeding up the edge weights determination process is to

1. precompute a sufficiently dense look-up table, mapping various edge shapes to a corresponding weight (with the maximal edge curvature being fixed),
2. estimate edge weights for the current planning problem at the beginning of the planning process instead of calculating them,
3. plan with estimated weights, and
4. after the planning finishes, substitute estimated edge weights in the solution with precisely calculated ones.

Determining edge shape and weight

A cubic edge between two vertices is determined by the points $\boldsymbol{P_0}$, $\boldsymbol{P_1}$ and vectors $\boldsymbol{P_0}'_{\boldsymbol{norm}}$, $\boldsymbol{P_1}'_{\boldsymbol{norm}}$. However, edge position and orientation are not needed in determining its length. If the edge is moved and rotated, so that $\boldsymbol{P_0}$ is in origin and $\boldsymbol{P_0}'_{\boldsymbol{norm}}$ points to $[1, 0]$, no information about the edge shape and length is lost the representation can be narrowed down to $\boldsymbol{P_1}_{\boldsymbol{new}} = [x, y]$ and angle α. Here, $\boldsymbol{P_1}_{\boldsymbol{new}}$ is the new position of $\boldsymbol{P_1}$ and α is the angle determining direction of $\boldsymbol{P_1}'_{\boldsymbol{norm\_new}}$.
Thus, the table with precomputed weights maps only three parameters tuple $[x, y, \alpha]$ to the corresponding weight (with K_{max} fixed).
The table size can be further reduced by considering the symmetry between some edge shapes. For example, the table size is reduced by half only by reflecting over x-axis all such edge shapes, that has $\boldsymbol{P_1}$ lying below the x-axis.

Look-up table sampling

Initially, the look-up table values were precomputed while using a uniform sampling with a fixed step over some predefined range. This yielded unsatisfactory results, because some edge weight estimates were off by several orders of magnitude, thus leading to solutions of no value. Precomputing the table with denser sampling did not solve this issue.
It was discovered that in case of some edge shapes, even a small difference in its parameters $[x, y, \alpha]$ leads to a significant change of its length. Also, interpolating by a single segment of Ferguson cubic curve does not work well for all possible configurations. For example, if $\boldsymbol{P_0'}$ and $\boldsymbol{P_1'}$ both lie on the line from P_0 to P_1 and point in an opposite or identical direction, the resulting curve has de facto infinite length. This is due to the requirement on keeping maximal curvature $K < K_{max}$ and limited flexibility of the curve.

Variable density sampling

The previously described issues were attempted to be solved by precomputing edge weights in a grid with a variable density, depending on the rate of change in a weight. The grid space is 3-dimensional, as the edge shape is defined by $[x, y, \alpha]$. Let's define a grid cell as a cube C in $x \times y \times \alpha$ space, where

$x \in \langle x_{min}, x_{max} \rangle$, $y \in \langle y_{min}, y_{max} \rangle$, $\alpha \in \langle \alpha_{min}, \alpha_{max} \rangle$ and denote its eight vertices (corresponding to edge shapes) as $e_i, i \in \langle 1..8 \rangle$. Also, let's define maximal allowed absolute deviation in edge weight as precision p_{max}. Let's also define a map M, that stores and maps edge shapes e_i to corresponding weights w_i.

Precomputing then proceeds recursively as follows:

1. Start with a cube C, covering the whole edge shape space defined by $\langle x_{min}, x_{max} \rangle$, $\langle y_{min}, y_{max} \rangle$, $\langle \alpha_{min}, \alpha_{max} \rangle$. Map $M = \{\}$.
2. For all vertices c_i of cube C, calculate the corresponding edge weight w_i and add the pair $\langle c_i, w_i \rangle$ to map M.
3. Calculate the maximal absolute deviation p over $c_i, i \in \langle 1..8 \rangle$ from

   $mean(\{c_i | i \in \langle 1..8 \rangle\})$
4. If $p > p_{max}$ and further splitting is possible, split cube C and continue with step 2.

In step 4, the cube is split so that the interval covered by each variable is halved. In the general case, eight sub-cubes are obtained.
It is advisable to set some maximal splitting level for each variable. If a certain resolution is reached, the cube is no longer split in this variable. This prevents issues caused by different scale or even units of the variables - with non-bounded splitting in all three variables, the procedure might be getting excessively accurate in one variable and insufficiently in another, causing M size to grow needlessly.
It might also be necessary to initially cover the space with multiple smaller cubes instead on a single one. By doing that, some initial precision is ensured - if the single initial cube is chosen badly, it may happen than the precision condition initiating further splitting is accidentally met too soon.

Searching the look-up table

To estimate the edge weight, we need to determine its shape by performing the previously described transform, find a shape in the look-up table that is closest to the given one and read the corresponding weight. In the case of the uniformly sampled look-up table with a fixed step in individual parameters, this can be performed in constant time.
With variable density sampling, however, finding the closest tuple of parameters $\langle x, y, \alpha \rangle$ in M is not a trivial task. To perform this operation in a reasonable time, a k-d tree containing edge shapes as nodes is built after loading precomputed M. Edge weight estimation is then performed in 2 steps. First, the closest tuple $\langle x, y, \alpha \rangle$ to the given one is found in the k-d tree, which is an operation with average logarithmic time complexity. Then, the corresponding weight can be read from the map M.

2.4.3 Optimization through intensification - DenseOpt

As explained in Chapter 1, the planning task definition is originally continuous, and each set is discretely covered by a finite number of vertices to allow utilization of GLNS. The basic assumption is, that if vertices are sampled uniformly and densely enough, the solution obtained from GLNS is close to the optimal one, with the same set ordering.
DenseOpt is a newly proposed intensification optimization technique performed on a resulting tour T obtained by GLNS. The idea is to search the close neighborhood of each vertex in T and randomly sample new admissible vertices, not present in the preceding GLNS search formulation $G = (V, E, w)$. If the newly sampled vertex improves the weight of T, it replaces the originally present one. The process is described in Algorithm 3.

Algorithm 3 DenseOpt

```
1: Input: GTSP_arc tour $T = (v_0, e_0, v_1, e_1, ..., v_{m-1}, e_{m-1})$
2: Output: Updated GTSP_arc tour $T$
3: Array $indices = [0, 1, ..., m-1]$
4: for $i = 1$ to $N_d$ do
5:     Uniformly randomly shuffle array $indices$
6:     for $j = 0$ to $m-1$ do
7:         $index = indices[j]$
8:         Vertex $v = v_{index}$ from $T$
9:         for $k = 1$ to $N_s$ do
10:            Sample valid vertex $v_{new}$ close to $v$
11:            if $v_{new}$ improves $w(T)$ then
12:                Replace $v_{index}$ in $T$ by $v_{new}$
13:                Update edges $e_{index-1}$, $e_{index}$ in $T$
14:            end if
15:        end for
16:    end for
17: end for
18: Return $T$
```

The whole tour T is optimized N_d times (line 4). In each iteration, a random order of resampling is created by shuffling array *indices* (line 5). Then, each vertex from the tour T is resampled N_s times (lines 9-10) as v_{new}. Sampling of v_{new} is limited to a predefined range of parameters r, α, and ω. This range is determined by the original density of a set coverage so that v_{new} is sampled anywhere between its closest neighbors. If v_{new} improves tour cost $w(T)$, it is added to T and v_{index} is removed (lines 11-12). Also, edges $e_{index-1}$ and e_{index} are newly generated, so that the newly added v_{new} is connected to the rest of the tour T (line 13). As the vertex v_{index} in T is being replaced continuously, its original position is stored in copy v, so that new vertices v_{new} are sampled in the same region (line 8). Parameter values used were $N_d = 5$, $N_s = 50$. Higher values did not yield better results.

Chapter 3

Experimental results

GLNS$_{\mathrm{arc}}$ implementation performance and all the additional improvements were tested on various instances. This Chapter describes the performed experiments and discusses the obtained results. It is structured as follows: Section 3.1 compares the performance of the first version of GLNS$_{\mathrm{arc}}$ with the original GLNS implementation on a set of GTSP instances. Section 3.2 examines performance of GLNS$_{\mathrm{arc}}$ after adaptation to the GTSP$_{\mathrm{arc}}$ with the use of four custom generated datasets. Each of these datasets enables to inspect influence of different trends, such as an increase in m, n or randomness in sets coverage by vertices. Section 3.2.3 covers issues linked to planning with cubic edges - namely weights precomputing time requirements and efficiency of weight estimation. Finally, Section 3.2.4 evaluates the beneficial effects of DenseOpt and studies its effects when applied on a solution with cubic edges, that was obtained from planning with roughly estimated edge weights.
All measured times further presented were obtained with Intel Core i7-7500U CPU (2.70 GHz), while running on a single core with 5.8 GB of RAM available to the process.

3.1 Comparison with original GLNS implementation

The original GLNS solver [SI17] is implemented in the Julia language. The GTSP$_{\mathrm{arc}}$ solver is written in C++, and it was first implemented as a GLNS solver, which was then transformed into the GTSP$_{\mathrm{arc}}$ solver by applying necessary modifications described in Section 2.3. This approach made it easy to asses the functionality and performance of the initial C++ implementation. Table 3.1 compares running times and best tour weights obtained by the two GLNS implementations; both running in the fast mode. Both implementations were tested on a subset of small GTSP instances provided in [OdLM13].
These results show that both implementations return practically the same results in terms of the best tour weight. Julia implementation stores edge weights in integers, thus resulting weights are not identical. As for the running times, C++ implementation is faster by one order for smaller problems, but by up to one order slower for problems with a higher number of sets m.

	Problem size		Time (s)		Weight	
Problem name	m	n	C++	Julia	C++	Julia
`5eil51`	5	51	0.04	1.04	74.99	75
`10eil51`	10	51	0.11	1.06	144.78	145
`15eil51`	15	51	0.30	1.07	203.24	201
`5berlin52`	5	52	0.03	1.04	2,065.90	2,065
`10berlin52`	10	52	0.10	1.05	3,223.39	3,223
`15berlin52`	15	52	0.27	1.08	4,689.99	4,691
`5st70`	5	70	0.03	1.27	102.74	103
`10st70`	10	70	0.09	1.19	212.18	211
`15st70`	15	70	0.27	1.42	249.76	248
`5eil76`	5	76	0.03	1.23	73.86	74
`10eil76`	10	76	0.10	1.18	127.39	127
`15eil76`	15	76	0.29	1.31	185.81	186
`5pr76`	5	76	0.03	1.11	19,245.10	19,246
`10pr76`	10	76	0.09	1.45	35,794.90	35,795
`15pr76`	15	76	0.27	1.33	52,154.80	52,156
`10rat99`	10	99	0.10	1.59	325.03	324
`25rat99`	25	99	1.21	1.46	508.79	509
`50rat99`	50	99	10.69	2.10	816.78	814
`25kroA100`	25	100	1.14	1.51	10,430.40	10,429
`50kroA100`	50	100	10.83	1.95	15,943.60	15,944
`10kroB100`	10	100	0.12	1.35	5,921.22	5,920
`50kroB100`	50	100	10.78	1.76	15,843.90	15,842
`25eil101`	25	101	1.34	1.67	238.68	237
`50eil101`	50	101	10.35	2.01	403.37	396
`25lin105`	25	105	0.81	1.60	8,226.05	8,223
`50lin105`	50	105	11.14	2.21	11,296.30	11,294
`75lin105`	75	105	43.04	4.10	13,135.00	13,134

Table 3.1: C++ vs. Julia GLNS implementation

The slower performance of Julia on smaller problems may be due to the fact, that a Julia program is compiled at runtime ('just in time' compilation). In the case of the small problems, the compilation accounts for most of the runtime. The C++ performance was analyzed using CLion built-in profiler, and it has shown, that the most expensive operations attribute to two GLNS$_{\text{arc}}$ components. First of them is the BFS search used in ReOpt local optimization. The second is utilization of C++ STL containers in general. The analysis indicates, that not the most suitable containers were always selected, as querying various structures or modifying them is more time consuming, than would be adequate. For example, set-vertex distances are stored in an std::map container, which maps a pair $\langle set.id, vertex.id\rangle$ to a distance value. Reading distance from this map has $\mathcal{O}(log(p))$ time complexity, where p is the number of pairs in the map. If a hashmap was implemented instead of the map, this operation could be performed in constant time. Issues like this, that were identified during performing experiments, were not fixed on the go so that all of the experiments could be compared with each other. As shown later, even the current GLNS$_{\text{arc}}$ implementation is capable of solving GTSP$_{\text{arc}}$ instances of the same size as the original GLNS implementation. Also, when planning with precisely computed cubic edges, the planning times of the current GLNS$_{\text{arc}}$ implementation are negligible compared to the duration of computing edge weights.

3.2 GLNS$_{\text{arc}}$ performance

GLNS$_{\text{arc}}$ performance is evaluated on three different datasets (`variable_n`, `variable_m`, `variable_m_n`), all of them being derived from the same default problem `100_12000`. Each dataset is designed to capture different trend - either the influence of change in the number of vertices n, in the number of sets m or both. The default problem is visualized in Figure 3.7 and its parameters are given in Table 3.2. This problem contains 100 identical sets, each of them containing 120 oriented and concentric vertices. There are 5 possible values of r, 12 values of α, 2 values for $sign$ and ω is fixed to $\pi/3$. Making all possible combinations of these four parameters produces 120 vertices. The sets are placed in space randomly, with minimal distance between individual sets being limited to 45, and the size of the space covered is limited to 550 in both dimensions.

Parameter	Value or range
m	100
n	12000
No. of vertices per set	120
Vertices α values	$\{0, \pi/6, ..., 11\pi/6\}$
Vertices ω values	$\pi/3$
Vertices r values	{5, 10, 15, 20, 25}

Table 3.2: Default problem `100_12000` parameters

All problems were generated by removing vertices or sets from the default problem - for details, see Table 3.3. Vertices were removed randomly from each set, while sets were removed according to their distance to the point $[0, 0]$ (farthest to closest).

Dataset	`variable_m`	`variable_n`	`variable_m_n`
No. of problems	17	24	20
m values	{20, 25,...,100}	50	{5, 10,..., 100}
Vertices per set	$2400/m$	{5, 10,...,120}	{25, 30,..., 120}

Table 3.3: Datasets description

The plots in Figures 3.2, 3.3 and 3.4 show GLNS$_{\text{arc}}$ performance on the generated datasets in terms of the planning time and the number of iterations needed. All tests were carried out in the fast mode.

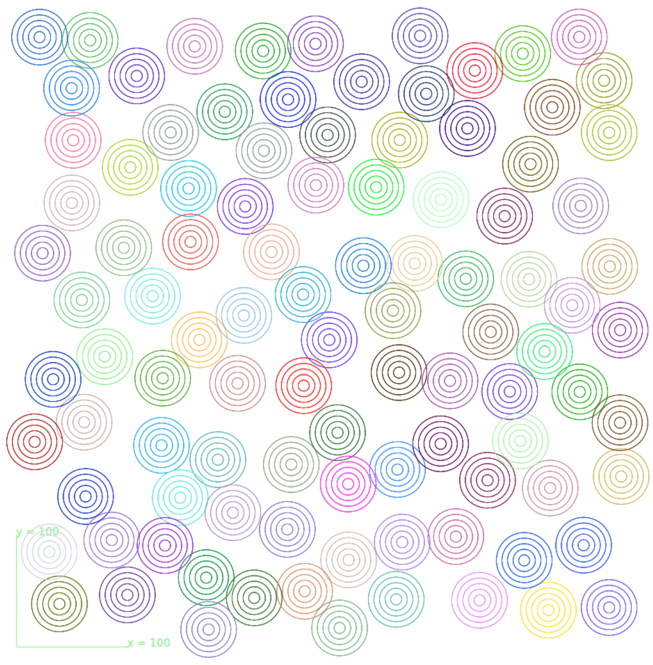
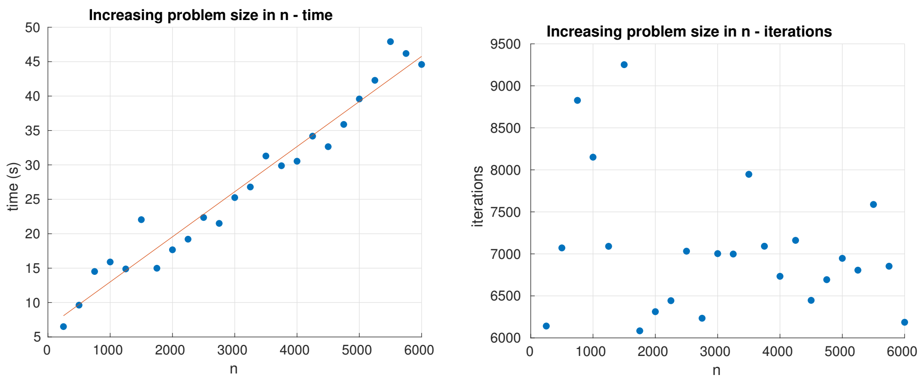


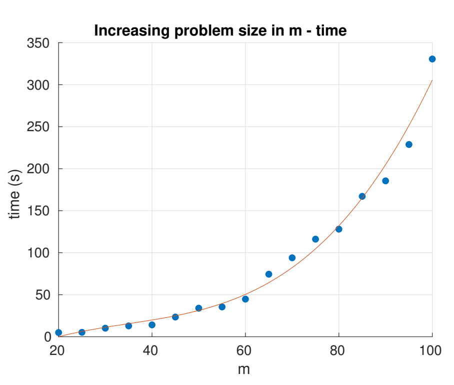
Figure 3.1: Default problem `100_12000`

According to [SI17], GLNS runtime in the fast mode is $\mathcal{O}(mn)$. Figure 3.2a shows linear dependence of the runtime on n with m fixed. However, Figure 3.3a indicates polynomial relation between m and the runtime. Therefore, GLNS$_{\text{arc}}$ implementation is probably suboptimal. Figure 3.4a then confirms, that the time complexity while increasing both m and n is polynomial.
As for the number of iterations - GLNS warm restarts end after reaching a fixed number of non-improving iterations (for details see Table 2.4), which is a multiple of m. Figure 3.2b does not indicate any evident relation between n and number of iterations, when m is fixed. Figures 3.3b and 3.4b document, that number of iterations indeed increases linearly relative to m.

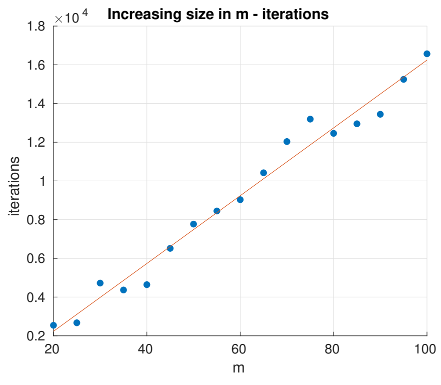


(a) : `variable_n` - time **(b) :** `variable_n` - iterations

Figure 3.2: GLNS$_{\text{arc}}$ performance on `variable_n` dataset

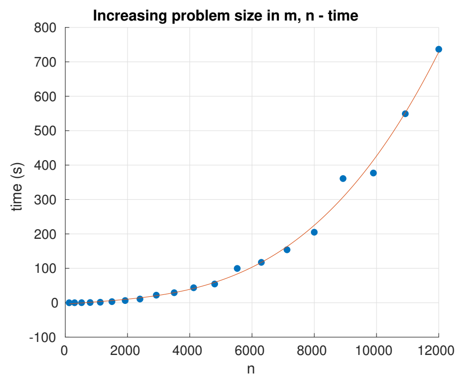


(a) : `variable_m` - time

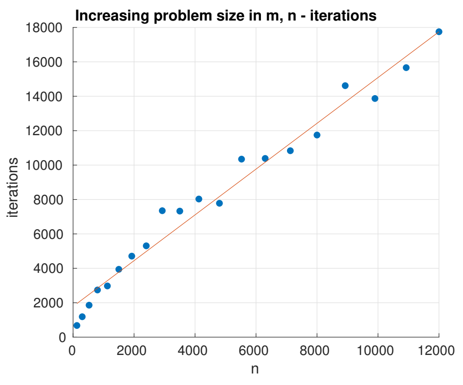


(b) : `variable_m` - iterations

Figure 3.3: GLNS$_{\text{arc}}$ performance on `variable_m` dataset



(a) : `variable_m_n` - time



(b) : `variable_m_n` - iterations

Figure 3.4: GLNS$_{\text{arc}}$ performance on `variable_m_n` dataset

3.2.1 Progress in quality of solution during runtime

The GLNS$_{\text{arc}}$ runtime is mainly influenced by m (determining the number of non-improving iterations before exiting each warm restart) and the mode of operation (determining most notably number of cold restarts). The behavior of GLNS$_{\text{arc}}$ in the fast mode was observed in detail on three problems: small with 25 sets, medium with 50 sets, and large with 100 sets. Each of these problems contains 120 vertices per a set. Figure 3.5 displays the current best weight in separate trials, while peaks in the plots correspond to cold restarts of trials.
Figure 3.6 then shows the progress of the best weight over all cold trials, while the complete runs are in the left plots and the second halves of the runs are in the right plots. The data shown were obtained by averaging 50 planner runs for each problem. The complete runs show that the biggest improvement is achieved during the first cold trial, no matter the problem size. The detailed plots of the second halves of the planning processes show

ongoing best weights improvement, but the total improvement in the second half of the planning is insignificant compared to the first cold trial.

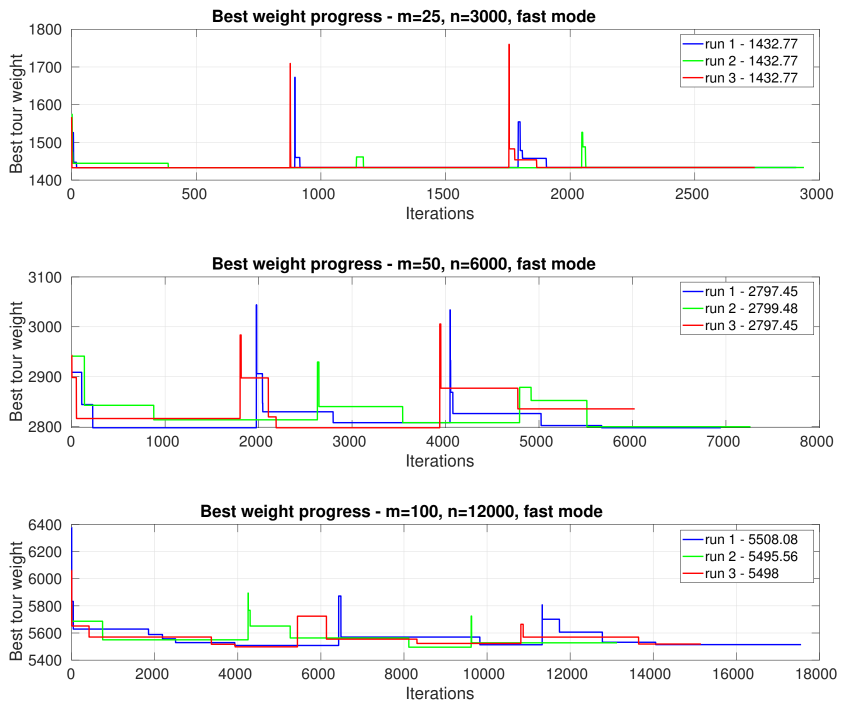


Figure 3.5: Best weight progress in individual trials

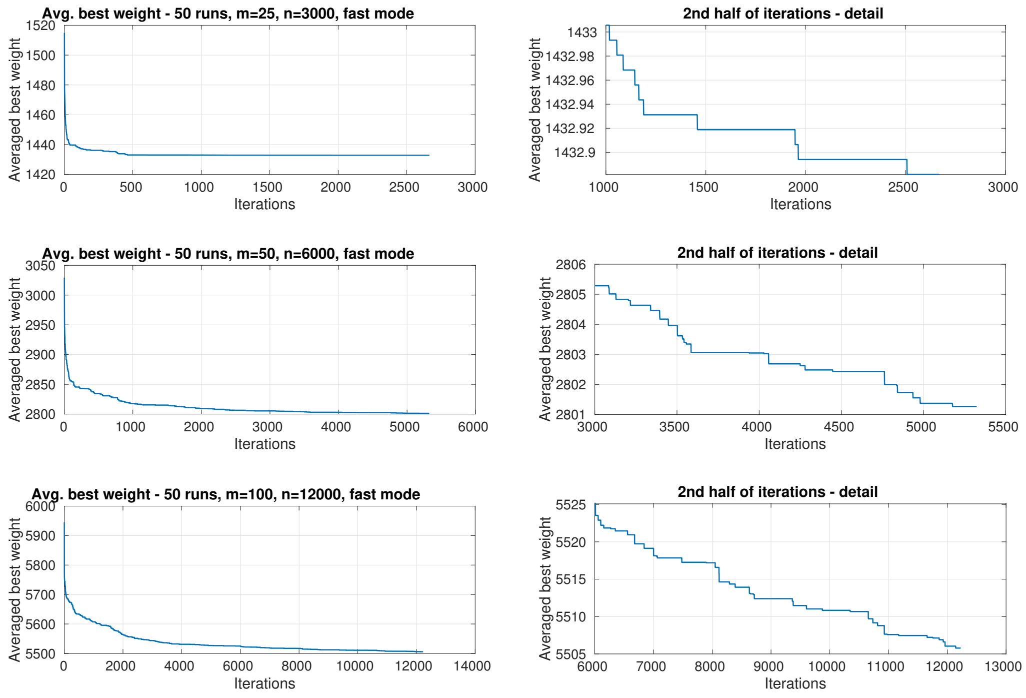


Figure 3.6: Best weight progress - 50 runs averaged

These results suggest that the fast mode is sufficient for all three problem sizes tested and that solving problems of similar size in medium or slow mode would probably yield negligible improvement compared to fast mode. However, all three problems contain clustered and non-overlapping sets with vertices uniformly sampled in the close neighborhood of the potential source of radiation. If the motivating task wouldn't imply such configuration, fast mode results for similarly sized but structurally different problems might be less satisfactory.

3.2.2 Solving a large problem

According to [SI17], GLNS was not tested on problems significantly larger than 10000 vertices. The stated reason for this is that GLNS always stores edge weights in a complete distance matrix and that for large problems, the memory requirements start to be too high.

GLNS$_{\text{arc}}$ was successfully tested on a problem with 200 sets and 24000 vertices. Detailed results are shown in Table 3.4. In GLNS$_{\text{arc}}$ C++ implementation, the distance matrix stores edge weights as 8-byte doubles. For a problem with 24000 vertices, the size of the distance matrix exceeds 4.5 GB. Therefore the memory requirements are indeed a limiting factor.

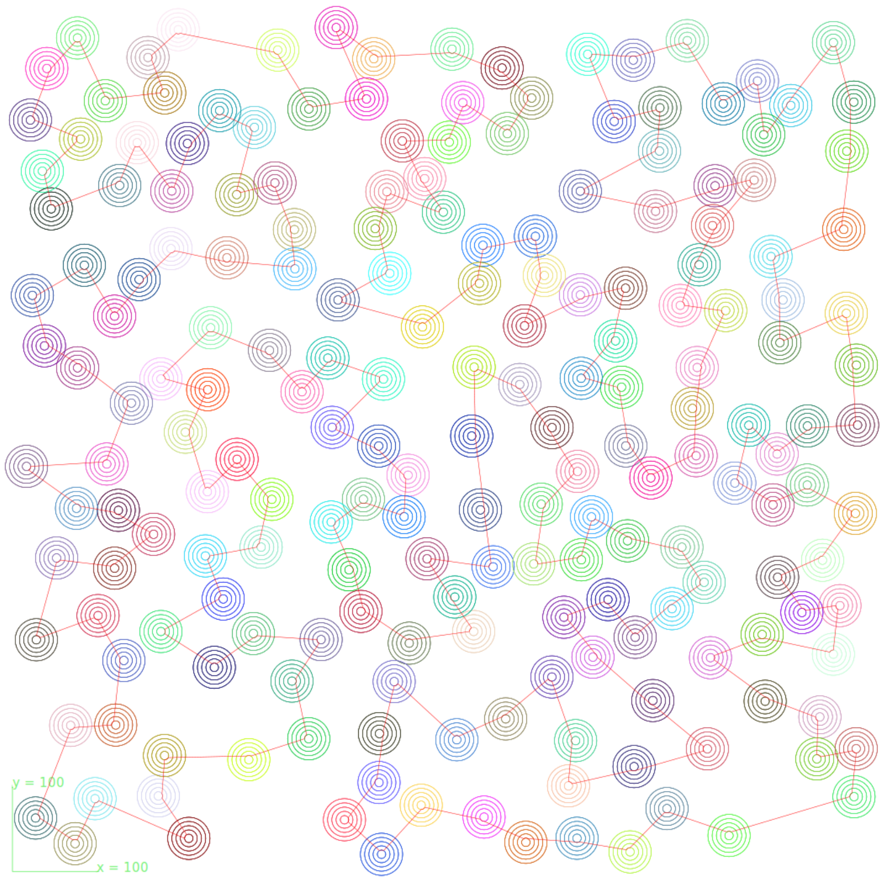


Figure 3.7: Solved large problem 200_24000

Attribute	Value
Problem name	200_24000
Problem size	$m = 200$, $n = 24000$
Planner mode	fast
Planning time	4h 23min 35s
Iterations	37264
Best weight	13367.4
Best weight after DenseOpt	13354.7

Table 3.4: Solving large problem - results

3.2.3 Planning with cubic edges

Planning with cubic edges instead of straight lines is closer to real-world problems. Figure 3.8 illustrates the difference between optimal trajectories for these two edge types. The trajectory using cubic edges has maximal curvature $K_{max} = \frac{1}{r_{min}} = 0.2$ and is smooth.

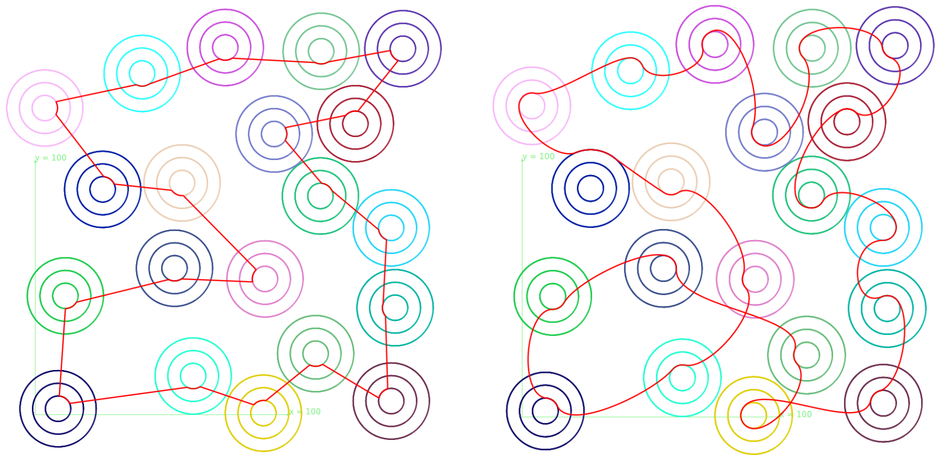


Figure 3.8: Straight line vs. cubic curve edges

Computing edge weights at runtime

GLNS$_{\text{arc}}$ computes a matrix of edge weights between all possible pairs of vertices before planning. Once computed, edge type does not affect the planning process, so strictly speaking, the planning time is not affected by the edge type. However, the computing time of cubic edge weights is significantly higher than for straight lines, and unfortunately, all or vast majority of weights is indeed queried during the planning process, so postponing weights computing until they are needed is not an option.
Figure 3.10 compares time requirements of weights precomputing for both edge types. There measurements are performed on dataset `variable_m_n_v2`. The number of sets in this datasets increases from 5 to 100 by 5 and the number of vertices by 120, as each set contains 120 vertices. The dependence

of the total precomputing time on n is quadratic and can be estimated as

$$t_{precomp} = \frac{n(n - \frac{n}{m})c}{2}, \tag{3.1}$$

where c is a time constant determined by the edge type. It is assumed that all sets are the same size, and that edge weight does not depend on its direction. On the hardware used, the value of this constant evaluated to 52 ns for straight lines and 0.47 ms for cubic curves. This leads to the conclusion that even small problems cannot be solved with cubic edges in a reasonable time due to the time demands of edge weights precomputing (assuming that the weights are not known in advance). To illustrate this issue further - a medium sized problem with 60 sets and 7200 was solved (see Figure 3.9 and Table 3.5 for detailed results). While planning in the fast mode finished in 82 seconds, precomputing all edge weights consumed more than 6 hours.

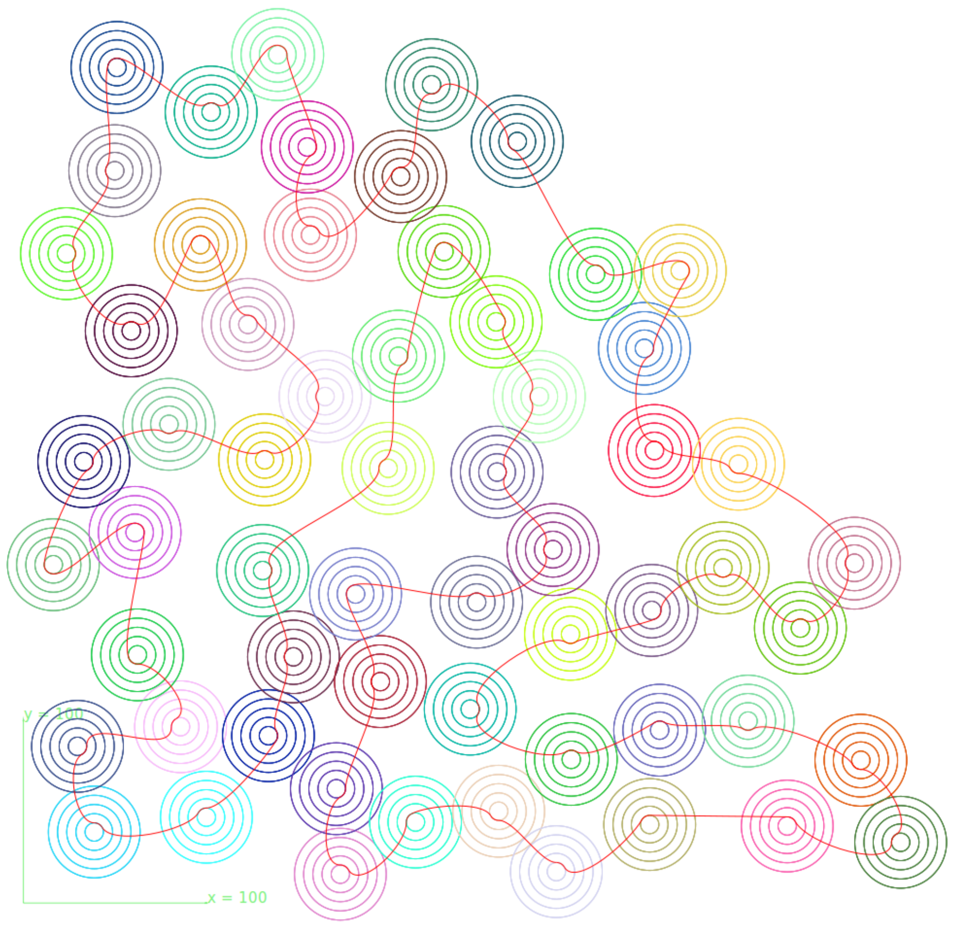
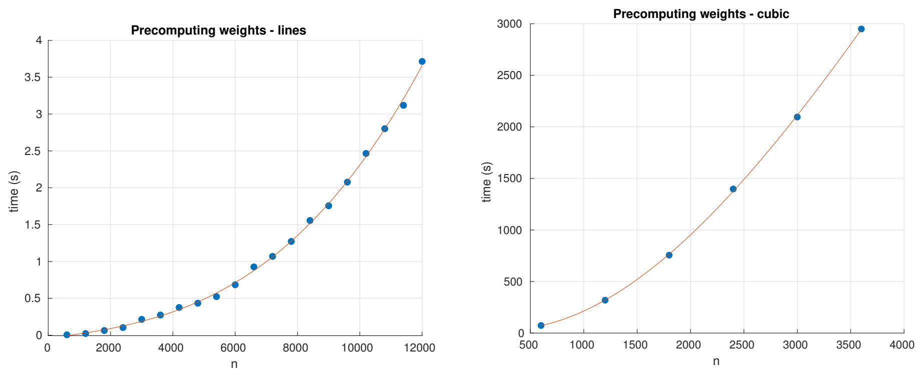


Figure 3.9: Solved problem 60_7200 with cubic edges

Attribute	Value
Problem name	60_7200
Problem size	$m = 60$, $n = 7200$
Planner mode	fast
Edges precomputing time	6h 39 min 30s
Planning time	77 s
Iterations	7911
Best weight	3824
Best weight after DenseOpt	3809

Table 3.5: Solving problem 60_7200 with cubic edges - results



(a) : Straight lines **(b) :** Cubic curves

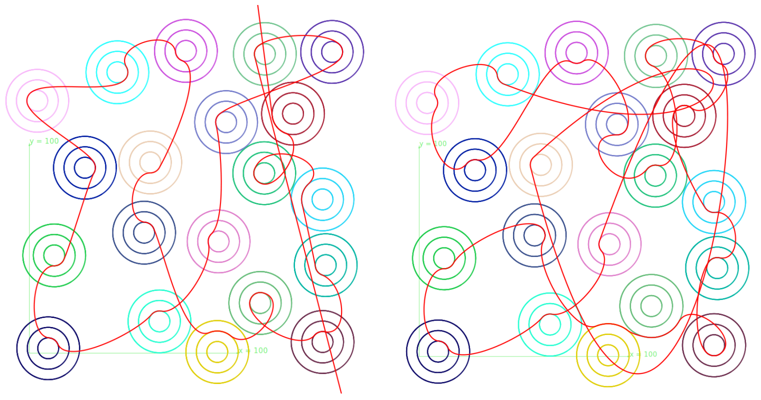
Figure 3.10: Edge weight precomputing - line vs. cubic

Precomputing and estimating edge weights

To solve the issue of disproportionate time requirements of cubic edges weight computing compared to planning time needed, various approaches to estimating edge weights instead of precise computation during runtime, introduced in Section 2.4.2, were tested.

First attempts utilized a table of weights that covers the space of edge shape parameters α, x, y uniformly. An example of a solution, where planning was based on weights obtained from this table is shown in Figure 3.11a. Parameters of the table used are given in Table 3.6. Even though the table covers the space large only enough to solve the problem shown and it is quite big in terms of memory requirements, some edge weights were estimated poorly, as the solution with precisely computed weights shown in Figure 3.8b differs significantly.

In the following attempt, variably dense table described in Table 3.6 was used. The solution obtained from planning with edge weights estimated from this table is shown in Figure 3.11b. Unlike in the previous attempt, there is no extremely underestimated edge going beyond the figure boundaries, but overall, the estimation precision is still poor, and the solution obtained is



(a) : Uniformly dense table **(b) :** Variably dense table

Figure 3.11: Edge weight estimation - uniformly vs. variably dense table

unsatisfactory. Also, as the table is not sampled uniformly, searching for the closest precomputed shape is no longer a constant time operation.

Precomputed shapes are stored in a k-d tree, and NN (nearest neighbor) search must be performed for every estimated edge. For the problem shown, which has 20 sets and 1440 vertices, searching the k-d tree for NN and reading corresponding weights took 149 s, while estimating 1968499 edges (1421 edges were out of range, and their weight was computed precisely). That is about 0.075 ms per edge, which is circa 10 times faster than computing the weight precisely. This value is, however, dependent on the size s of a precomputed table stored in the k-d tree. Searching k-d tree has $\mathcal{O}(log(s))$ time complexity, whereas precise weights computing is $\mathcal{O}(n^2)$.

In conclusion, the approaches elaborated in this section do accelerate edge weights precomputing significantly, but planning with the estimated weights does not lead to useful solutions. The next section provides a solution to this problem, but at the cost of relaxing problematic vertices position.

Parameter	Uniformly sampled table	Variably sampled table
x, y range	$\langle -200, 200\rangle$	$\langle -200, 200\rangle$
x, y resolution	5	$\langle 1, 10\rangle$
α range	$\langle 0, 2\pi\rangle$	$\langle 0, 2\pi\rangle$
α resolution	$\pi/180$	$\langle \pi/180, \pi/36\rangle$
No. of weights	3682561	4524275
Size	117 MB	144 MB
Min. weight precision	undefined	10

Table 3.6: Precomputed tables of weights - parameters

3.2.4 DenseOpt

Application of the DenseOpt intensification procedure was tested on two previously used datasets - `variable_m_n` and `variable_m_n_v2`.

Problem	Best tour weight	After denseOpt	Rel. improvement (%)
5_125	287.44	265.84	7.5
10_300	644.85	612.38	5.0
15_525	878.54	835.43	4.9
20_800	1092.70	1044.86	4.4
25_1125	1450.32	1390.82	4.1
30_1500	1682.23	1607.52	4.4
35_1925	1973.60	1883.96	4.5
40_2400	2262.74	2172.86	4.0
45_2925	2557.01	2464.20	3.6
50_3500	2827.73	2717.00	3.9
55_4125	3091.62	2971.75	3.9
60_4800	3393.69	3273.88	3.5
65_5525	3638.18	3491.25	4.0
70_6300	3861.62	3713.05	3.8
75_7125	4076.86	3934.63	3.5
80_8000	4410.73	4265.44	3.3
85_8925	4660.72	4496.22	3.5
90_9900	4971.73	4794.31	3.6
95_10925	5233.00	5050.15	3.5
100_12000	5476.30	5274.32	3.7

Table 3.7: DenseOpt performance on `variable_m_n` dataset

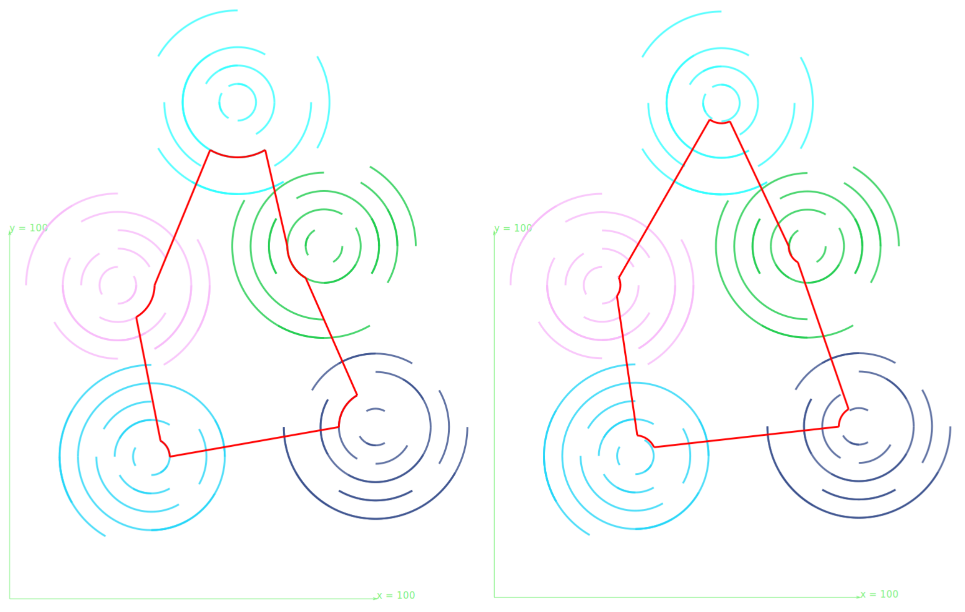
Problem	Best tour weight	After DenseOpt	Rel. improvement (%)
5_600	275.36	265.76	3.5
10_1200	635.13	612.26	3.6
15_1800	863.83	835.71	3.3
20_2400	1078.15	1043.16	3.2
25_3000	1432.77	1391.03	2.9
30_3600	1662.15	1608.06	3.3
35_4200	1948.42	1883.90	3.3
40_4800	2240.87	2169.33	3.2
45_5400	2535.35	2450.95	3.3
50_6000	2799.19	2705.22	3.4
55_6600	3074.44	2973.17	3.3
60_7200	3360.68	3250.42	3.3
65_7800	3626.75	3498.26	3.5
70_8400	3860.59	3712.77	3.8
75_9000	4104.47	3954.58	3.7
80_9600	4406.75	4270.81	3.1
85_10200	4672.13	4512.84	3.4
90_10800	4966.85	4775.25	3.9
95_11400	5211.59	5015.80	3.8
100_12000	5504.73	5290.52	3.9

Table 3.8: DenseOpt performance on `variable_m_n_v2` dataset

The difference between them is, that problems in the first dataset contain sets with a variable number of vertices, depending on the problem size, whereas problems in the second dataset always have 120 vertices per a set, evenly distributed in terms of α and r. In these experiments, GLNS$_{\text{arc}}$ was planning with straight lines as edges. DenseOpt is performed for a total number of 5 iterations after GLNS$_{\text{arc}}$ planning finished. In each iteration, 50 vertices are uniformly randomly sampled in the close neighborhood of each vertex present in the best tour found (in a random order). For a vertex v with parameters r and α, this neighbourhood is limited to intervals $\langle max(r-5,5), min(r+5,25)\rangle$ and $\langle \alpha - \pi/6, \alpha + \pi/6\rangle$. These ranges correspond to the density of vertex sampling in both datasets. Parameter ω is kept constant, and r is limited to interval $\langle 5, 25\rangle$, again in correspondence with the datasets.

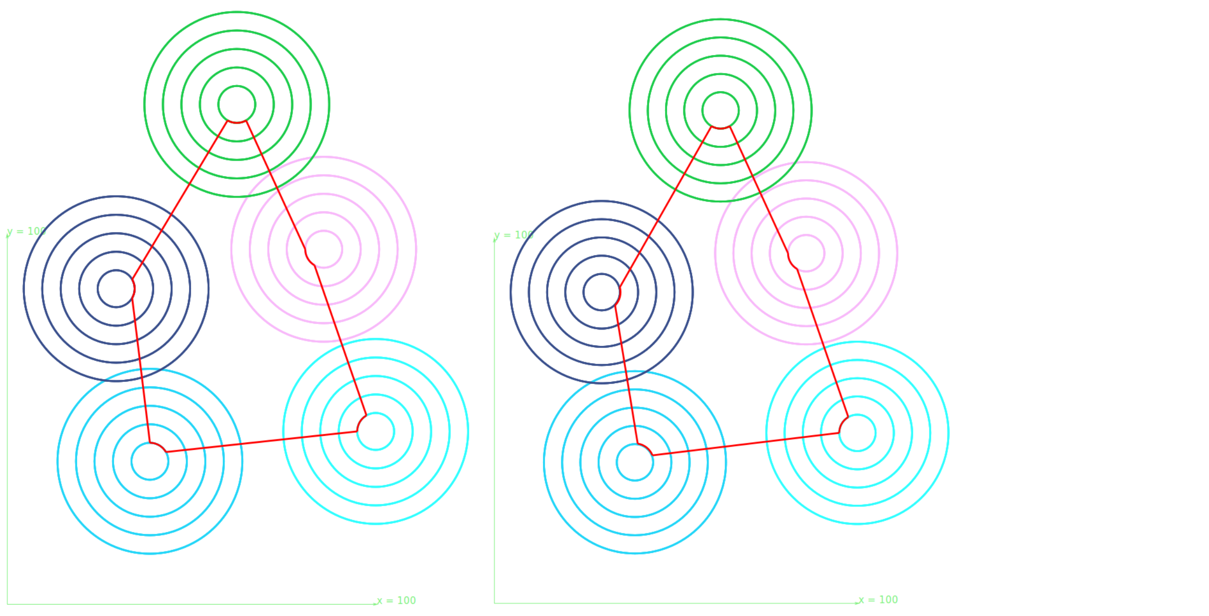
Table 3.7 shows performance of DenseOpt on the `variable_m_n` dataset. The relative improvement was calculated as $\frac{B-A}{B}$, where B is the best tour weight before performing DenseOpt, and A is the weight of the same tour after DenseOpt. Here, it varies from 3% to 7%, while better results are obtained on smaller problems. Table 3.8 then shows performance on the `variable_m_n_v2` dataset, where the improvement is consistently about 3%.

Examples of how DenseOpt affects the solution are visualized in Figures 3.12 and 3.13. In case of problem `5_125` DenseOpt converges to the shortest possible arcs. Sets in this problem contain randomly placed vertices.



(a) : `5_150` optimal plan **(b) :** `5_150` after DenseOpt

Figure 3.12: Effect of DenseOpt on `5_125`



(a) : `5_600` optimal plan   **(b) :** `5_600` after DenseOpt

Figure 3.13: Effect of DenseOpt on `5_600`

As for the second problem `5_600`, vertices with the smallest possible radius are already present in the optimal path returned by GLNS$_{\text{arc}}$, and DenseOpt only slightly rotates these. This problem contains a larger number of vertices that are displaced uniformly. Improvement in the solution weight is visually almost imperceptible.

This observation might explain the better performance of DenseOpt on the `variable_m_n` dataset, that contains problems similar to `20_150`. Universally, shorter vertices seem to be preferable. If they are already present in the problem solved and are sampled sufficiently densely, DenseOpt adjusts only their α value, thus rotates them.

Applying DenseOpt after cubic edges estimation

Planning with estimated weights described in Section 3.2.3 does not yield satisfying results. After the estimated weights in the resulting tour are replaced by precisely computed edges, the solution turns out to be often useless due to a large difference between its real weight and the estimated one, as shown in Figure 3.11. However, even though the real edge weight is often estimated poorly, its presence in the solution indicates, that there should be a better alternative in its close neighborhood. The assumption is that performing DenseOpt after planning with estimated weights could converge to it, thus leading to an acceptable solution.

DenseOpt influence on tour weight after planning with precisely computed edge weights is presented in Table 3.9. Due to the rapidly growing time requirements of edge weight computing, only 4 smallest problems are included,

	$w(T)$				Avg. time (s)	
Problem	Min	Max	Mean	St. dev.	Precomputing	Planning
5_125	265.71	266.10	265.96	0.15	5.16	0.01
10_300	612.34	616.98	613.86	2.12	37.83	0.07
5_600	265.69	265.82	265.74	0.05	128.05	0.08
10_1200	612.35	617.28	615.96	1.89	580.84	0.44

Table 3.9: Tour weights after DenseOpt - precise planning

	$w(T)$				Avg. time (s)	
Problem	Min	Max	Mean	St. dev.	Estimating	Planning
5_125	265.75	266.20	265.96	0.14	11.54	0.33
10_300	612.29	616.96	612.90	1.43	15.46	0.38
5_600	265.75	265.88	265.79	0.04	24.07	0.35
10_1200	612.27	612.57	612.40	0.09	79.60	0.75

Table 3.10: Tour weights after DenseOpt - planning with estimated weights

and each of them is solved only 10 times. Table 3.10 then presents results of performing DenseOpt on the same problems, but after planning with estimated edge weights. Edge weights estimation was performed while using a variably sampled table with parameters given in Table 3.6. Mean values of final tour weights differ very little, and even other observed properties such as minimal weight, maximal weight, and standard deviation across all runs stay within a similar range.
This leads to the conclusion that the planning process with cubic edges can indeed be significantly accelerated without decreasing the solution quality by using a table of precomputed edge weights, given that the solution is then optimized by DenseOpt. A drawback of this approach is that before solving any problem a sufficiently large and dense table of weights must be available. Also, the table is valid only for a single predefined value of maximal edge curvature K_{max}.

Chapter 4
Conclusion

This thesis is motivated by a practical task, whose goal is to search for sources of gamma radiation while using a UAV and a UGV. The objective of the UAV is to quickly estimate the approximate location of the sources of radiation. These locations are then to be determined precisely by the UGV, and the task solved here is to find an optimal or at least as good as a possible trajectory to do so. Each source of radiation can be precisely located by passing through a circular arc trajectory segment in its close neighborhood, while these segments can be connected arbitrarily. This task can be after suitable discretization formulated as the GTSP with certain modifications caused by using circular arcs as vertices and is referred to as the GTSP_{arc}.
The state of the art GLNS algorithm was successfully modified to solve the GTSP_{arc}, and this modified version is being called GLNS_{arc}. GLNS_{arc} was first implemented as a GTSP solver, and its performance was compared with the original GLNS implementation on various GTSP instances. Results of these tests show that GLNS_{arc} finds solutions of the same quality as GLNS and therefore is likely to be implemented correctly.
After adapting to solve the GTSP_{arc}, GLNS_{arc} was tested on four newly generated GTSP_{arc} datasets, and its performance was analyzed in detail. Relationship between separate problem characteristics such as problem size, number of sets m or number of vertices n and computation time needed were experimentally assessed on these datasets, containing problems of up to 100 sets and 12000 vertices. Also, a large problem of 200 sets and 24000 vertices was successfully solved. According to [SI17], original GTSP implementation was not tested on problems significantly larger than 10000 vertices, so solving a problem with 24000 vertices is a success. It was also shown, that for the GTSP_{arc} instances based on the practical task, GLNS_{arc} can be utilized in the fast mode, as slower modes of operation do not further improve the quality of the solution. This may be due to the fact, that all of these problems contain clustered, non-overlapping sets.
After successfully implementing GLNS_{arc}, two major improvements were added. The first one is the option to plan with cubic curves as edges instead of straight lines. These curves are generated so that the predefined maximal trajectory curvature (corresponding to the minimal robot turning radius) is always kept. Precomputing of these cubic edge weights is $\mathcal{O}(n^2)$ time consum-

ing and significantly increases total planning time. Therefore, a procedure of estimating their weights from the precomputed variably dense weight table is proposed and tested. If the resulting tour with estimated edge weights is subject to the proposed intensification procedure, acceptable solutions with cubic edges are obtained.

Another improvement added is the intensification method called DenseOpt. The $\mathrm{GTSP}_{\mathrm{arc}}$ formulation and $\mathrm{GLNS}_{\mathrm{arc}}$ approach are discrete, but the motivating task is, in fact, continuous. This can be utilized to obtain a better quality solution without breaking constraints on vertex validity. DenseOpt takes the $\mathrm{GLNS}_{\mathrm{arc}}$ final solution as an input, performs a randomized local search around each vertex and potentially samples, and adds newly generated valid vertices to the tour if they improve the total tour cost. In other words, $\mathrm{GLNS}_{\mathrm{arc}}$ determines a set ordering, and approximate the position of the best vertex from each set and DenseOpt further refines this solution. The experimental results have shown, that improvement of the total tour weight is between 3-7%, depending mainly on the problem size and distance between individual sets.

In conclusion, the assignment has been fulfilled, as the motivating planning problem can be successfully solved by $\mathrm{GLNS}_{\mathrm{arc}}$. Planning with cubic edges was added so that the trajectories generated are suitable even for non-holonomic robot and DenseOpt intensification compensates for the imprecision caused by problem discretization.

Further work can lie in researching how to sample vertices so that the area of potential sources of radiation covered by one set of vertices is the largest possible. Datasets generated for purposes of testing $\mathrm{GLNS}_{\mathrm{arc}}$ contain only sets with concentric vertices, without further examination of how large area such a set covers or how to maximize this area.

Also, the proposed procedure of estimating edge weights from a precomputed table is limited by this table size, because satisfactory edge weight estimation requires a sufficiently dense table, whose size increase rapidly with the problem space size. On the other hand, precise computation of weights proved to be time-consuming. Therefore, it would be beneficial to determine a formula for estimating the cubic edge weight with a certain maximal curvature, that would be reasonably accurate, yet fast to compute.

Finally, the quality of the $\mathrm{GLNS}_{\mathrm{arc}}$ solution is limited by the density of vertex sampling, as the original problem is, in fact, continuous. This is partially compensated by DenseOpt local intensification. However, it might be feasible to implement a continuous version of $\mathrm{GLNS}_{\mathrm{arc}}$. Individual sets would be defined only by constraints on vertices, insertion heuristics, removal heuristics, and other mechanisms would be adapted accordingly, and the high-level principles of adaptive large neighborhood search would remain the same.

Bibliography

[AVS96] Gilbert Laporte, Ardavan Asef-Vaziri and Chelliah Sriskandarajah, *Some applications of the generalized travelling salesman problem*, The Journal of the Operational Research Society **47** (1996), 1461–1467.

[GK10] Gregory Gutin and Daniel Karapetyan, *A memetic algorithm for the generalized traveling salesman problem*, Natural Computing **1** (2010), 47–60.

[GL18] Petr Gabrlik and Tomas Lazna, *Simulation of gamma radiation mapping using an unmanned aerial system*, 15th IFAC Conference on Programmable Devices and Embedded Systems PDeS 2018 **51** (2018), 240–254.

[GT97] Matteo Fischetti, Juan José Salazar González and Paolo Toth, *A branch-and-cut algorithm for the symmetric generalized traveling salesman problem*, Operations Research **45** (1997), 327–494.

[GZ17] Tomas Lazna, Tomas Jilek, Petr Gabrlik and Ludek Zalud, *Multi-robotic area exploration for environmental protection*, Lecture Notes in Computer Science **10444** (2017), 240–254.

[Hel00] Keld Helsgaun, *An effective implementation of the lin–kernighan traveling salesman heuristic*, European Journal of Operational Research **126** (2000), 106–130.

[Hel15] ______, *Solving the equality generalized traveling salesman problem using the lin–kernighan–helsgaun algorithm*, Mathematical Programming Computation **7** (2015), 269–287.

[NB91] Charles E. Noon and James C. Bean, *A lagrangian based approach for the asymmetric generalized traveling salesman problem*, Operations Research **39** (1991), 528–687.

[NB93] ______, *An efficient transformation of the generalized traveling salesman problem*, INFOR: Information Systems and Operational Research **31** (1993), no. 1, 39–44.

[NLo] NLopt, *Nonlinear Optimization library*, https://nlopt.readthedocs.io.

[OdLM13] M. Mestria, L. S. Ochi and S. de Lima Martins, *Grasp with path relinking for the symmetric euclidean clustered traveling salesman problem*, Computers Operations Research **40** (2013), 3218–3229.

[PR18] Rafael Martí, Panos M. Pardalos and Mauricio G. C. Resende, *Handbook of heuristics*, Springer, 2018.

[SI17] Stephen L. Smith and Frank Imeson, *GLNS: An effective large neighborhood search heuristic for the generalized traveling salesman problem*, Computers and Operations Research **8** (2017), 1–19.

[YQ99] J. Ye and R. Qu, *Fairing of parametric cubic splines*, Mathematical And Computer Modelling **30** (1999), 121–131.

Appendix A

Content of the attached CD

Filename or directory	Description
GLNS	C++ GTSP solver
modified_GLNS	C++ GTSP_{arc} solver - GLNS_{arc}
data	GTSP_{arc} problem datasets
results	GLNS_{arc} test results
weights	precomputed edge weights tables
DP.pdf	text of this thesis
readme.txt	detailed data description, code instructions etc.