Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics
Study program: Cybernetics and Robotics

Hexaly Optimizer for Multi-Goal Problems

Hexaly Optimizer pro problémy s více cíli

MASTER THESIS

Author: Bc. Libor Dubský
Supervisor: RNDr. Miroslav Kulich, Ph.D.
May 2025

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Dubský Libor** Personal ID number: **491834**

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:

Hexaly Optimizer for Multi-Goal Problems

Master’s thesis title in Czech:

Hexaly Optimizer pro problémy s více cíli

Guidelines:

Hexaly is a new kind of global optimization solver. It offers nonlinear and set-oriented modeling APIs. The authors claim that the solver relies on innovative proprietary algorithms, and it is faster and more scalable than Mixed-Integer Programming, Constraint Programming, and Nonlinear Programming solvers. The thesis aims to evaluate Hexaly Optimizer in various multi-goal problems. The student will proceed in the following steps:
1. Get acquainted with Hexaly Optimizer and its modeling language.
2. Get acquainted with multi-goal problems, e.g., the Traveling Deliveryman Problem, the Graph Search Problem, the Generalized Traveling Salesman Problem, and the state-of-the-art solvers of these problems.
3. Model selected multi-goal problems in Hexaly Modeller/C++/python.
4. Study the quality of solutions found by Hexaly Optimizer and compare them with solutions found by state-of-the-art solvers. Document and discuss the results.

Bibliography / sources:

[1] https://www.hexaly.com
[2] Mikula, J., and Kulich, M. (2022). Solving the traveling delivery person problem with limited computational time. Central European Journal of Operations Research, 1–31.
[3] 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.
[4] Mikula, J., & Kulich, M. (2022). Towards a Continuous Solution of the d-Visibility Watchman Route Problem in a Polygon With Holes. IEEE Robotics and Automation Letters, vol. 7, no. 3, pp. 5934-5941.
[5] 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.

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: **10.02.2025** Deadline for master's thesis submission: **23.05.2025**

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): Dubský Libor
Personal number: 491834
Programme name: Cybernetics and Robotics

declare that I have elaborated the master's thesis entitled

Hexaly Optimizer for Multi-Goal Problems

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 13.05.2025

Bc. Libor Dubský
.....
student's signature

Acknowledgements

I am grateful to my thesis supervisor RNDr. Miroslav Kulich, Ph.D., for his feedback and valuable advice regarding this work.

Bc. Libor Dubský

Title:	Hexaly Optimizer for Multi-Goal Problems
Author:	Bc. Libor Dubský
Thesis supervisor:	RNDr. Miroslav Kulich, Ph.D.
Abstract:	This thesis deals with evaluating the effectiveness of the Hexaly Optimizer solver developed by Hexaly. Its performance has been examined on selected multi-goal problems, such as the Traveling deliveryman problem, the Graph search problem, the Generalized traveling salesman problem, and the Generalized graph search problem. The results were compared with the outputs of advanced existing solvers such as Ms-GVNS, GILS-RVND, and GLNS. The accuracy of the solutions was recorded in tables and graphs to compare the effectiveness of Hexaly Optimizer and other solvers.
Key words:	Hexaly Optimizer, Traveling deliveryman problem, Graph search problem, Generalized traveling salesman problem, Generalized graph search problem
Název práce:	Hexaly Optimizer pro problémy s více cíli
Autor:	Bc. Libor Dubský
Vedoucí práce:	RNDr. Miroslav Kulich, Ph.D.
Abstrakt:	Tato práce se zabývá zhodnocením efektivity optimalizačního řešiče Hexaly Optimizer vyvíjeného společností Hexaly. Jeho výkonnost byla prověřena na vybraných problémech s více cíli, jako je problém doručovatele, prohledávání grafu, zobecněný problém obchodního cestujícího a zobecněný problém prohledávání grafu. Výsledky byly porovnány s výstupy pokročilých existujících řešičů, jako jsou Ms-GVNS, GILS-RVND a GLNS. Přesnost řešení byla zaznamenána do tabulek a grafů, které slouží k přehlednému porovnání efektivity Hexaly Optimizeru a ostatních řešičů.
Klíčová slova:	Hexaly Optimizer, problém doručovatele, problém prohledávání grafu, zobecněný problém obchodního cestujícího, zobecněný problém prohledávání grafu

Contents

List of abbreviations

HO Hexaly Optimizer

HM Hexaly Modeler

TSP Traveling salesman problem

TDP Traveling deliveryman problem

GSP Graph search problem

GTSP Generalized TSP

GGSP Generalized Graph search problem

Ms-GVNS Multi-Start General Variable Neighborhood Search

GILS-RVND Greedy Iterative Local Search - Reactive Variable Neighborhood Descent

GLNS Generalized large neighborhood search

GLNS_GGSP GLNS for GGSP

CIIRC Czech Institute of Informatics, Robotics and Cybernetics

List of Figures

List of Tables

Introduction

Multi-goal problems are a specific class of mathematical problems designed to find an efficient or optimal sequence of satisfying multiple goals given predefined conditions and constraints. These constraints may often conflict, requiring the search for compromise solutions. Multi-goal problems are commonly encountered in areas such as bioinformatics, finance, planning, logistics, and robotics. However, as the number of objectives increases, the computational complexity of these problems increases exponentially, and finding an optimal solution becomes computationally and time-consuming.

At the Czech Institute of Informatics, Robotics and Cybernetics (CIIRC), these problems are mainly addressed in the context of robotics - for example, in space search or optimizing the transport of objects from one place to another. These problems directly apply to the commercial sphere, where efficient solutions save time, costs, and other resources. For this reason, CIIRC focuses on developing and evaluating the most efficient optimization solvers.

Hexaly Optimizer is a newly developed type of optimization solver developed by Hexaly, a company specializing in mathematical optimization. According to the authors, this solver is faster and more scalable than traditional solvers [1]. This thesis focuses on an experimental comparison of the performance of the Hexaly Optimizer with selected existing methods, namely Ms-GVNS, GILS-RVND [2], and GLNS [3]. The comparison of the quality and speed of the solutions found will be performed on different types of combinatorial problems, specifically:

- Traveling deliveryman problem (TDP)
- Graph search problem (GSP)
- Generalized Traveling salesman problem (GTSP)
- Generalized Graph search problem (GGSP)

We use these problems to assess whether the Hexaly Optimizer offers higher performance and solution quality than established approaches.

The objective and contributions

The main objective of this thesis is to compare the efficiency of the Hexaly Optimizer against other solvers. Partial tasks of this thesis are:

- Acquaint with Hexaly Optimizer and choose a suitable programming language.
- Acquaint with selected multi-goal problems, and the state-of-the-art solvers of these problems.
- Model problems with Hexaly Optimizer to solve them.
- Study the quality of solutions found by Hexaly Optimizer and compare them with solutions found by state-of-the-art solvers.

Chapter 1 focuses on the technical background. Chapter 2 addresses the TDP, Chapter 3 the GSP, Chapter 4 the GTSP, and Chapter 5 the GGSP. Each chapter contains a definition of the problem, a description of its modeling in Hexaly Optimizer, a comparison with other advanced solvers, and a presentation and evaluation of the experimental results. Finally, it summarizes the results obtained, evaluates the performance of Hexaly Optimizer, and suggests possibilities for further development.

Chapter 1

Technical background

1.1 Hexaly Optimizer description

According to [1], HO is a new type of solver. It is implemented in C++ and supports Windows, Linux, and MacOS operating systems. HO can be imported as a library in C++, Python, C#, or Java programs. Also, it can be a standalone executable program coded in Hexaly Modeler (HM). HO binaries are self-contained, so there is no need to install any third-party library to run HO.

To solve a problem using HO, a mathematical model of the problem must be defined. Writing the model as mathematically as possible rather than programmatically is better. If the model is unnecessarily complicated, or if the model uses, for example, a modulo operation for indexing, then the computation time may be increased, or HO may not recognize that it has already found the optimal solution.

1.2 Hexaly Modeler description

According to [1], Hexaly Modeler (HM) is a modeling and programming language. HM offers operators like minimize, maximize, and constraint to define and parametrize the model. It also provides a set of functionalities for programming, for example, loops, conditions, and variables. HM has two modes: main mode and default (classic) mode. In the main mode, HM behaves like an ordinary programming language by declaring the function **main()**, and in the default mode, the program is structured around five predefined functions. These functions are **input()**, **model()**, **param()**, **display()**, **output()** and are executed in this order. The functions are described in [1]:

- **input**: for declaring your data or reading them from files.
- **model**: for declaring your optimization model.
- **param**: for parameterizing the local-search solver before running.
- **display**: for displaying some info in console or in some files during the resolution.
- **output**: for writing results in console or in some files, once the resolution is finished.

Both modes have their built-in variables and functions to help define solving parameters.

1.3 Used hardware and software

We have decided to use the latest version of Hexaly 13.5, which can run on a personal computer. We chose a variant designed for the Linux operating system. To ensure the

smooth running of the application, it is necessary to meet the relevant system requirements [1]:

- Architecture: x64, arm64
- Operating systems: Linux with `libc` 2.28 (or superior) and `libstdc++` 6.0.25 (or superior)
- The system should have at least 4 physical cores for nominal performance

We installed this version on a personal computer with the following hardware and system specifications:

- Intel® Core™ i7-8750H CPU (2.20 GHz)
- Kingston 256 GB M.2 SATA SSD
- 16 GB of RAM
- Ubuntu 24.04 LTS

This configuration exceeds the minimum requirements needed for the flawless operation of the Hexaly 13.5.

1.4 Description of programs with Hexaly Optimizer

All programs with HO are implemented in HM in the default mode. Modeling a problem in HM seemed more straightforward than in other supported programming languages. All programs have a similar set of input parameters. The essential ones are:

- *inFileName* - determines the path to the input instance
- *hxTimeLimit* - defines the time limit for finding the optimal solution
- *solFileName* - specifies the path to the file where the output data is stored

An example of executing the program can look like:

```
hexaly tdp.hxm inFileName=st70.tsp hxTimeLimit=10
   solFileName=sol.txt
```

`hexaly` is an executable tool and `tdp.hxm` is an executable script. The order of the input arguments does not matter. Values are assigned directly to specific variables at execution time based on their names, not their order. Only the presence of all required parameters is required for proper functionality. To create a new variable or to change the value of a predefined variable, you can submit these variables as additional input arguments along with their values to the program before execution.

Chapter 2

The Traveling deliveryman problem

2.1 Definition

The Traveling deliveryman problem (TDP) is a problem where a deliveryman leaves the depot to deliver packages to customers so that everyone is waiting for the shortest possible time. In this problem, it is assumed that the deliveryman carries one package for each customer, so he visits each customer just once and does not need to return to the depot after deliveries. The solution to this problem is a sequence of the orders of customers to minimize the sum of the waiting times of each customer. This is similar to the Traveling salesman problem (TSP), which minimizes the total travel time between all customers. The difference between the ideas of these problems is that TSP, by minimizing travel time, actually seeks the sequence at which the cost to the carrier is the lowest. TDP seeks the sequence at which it satisfies customers regardless of cost by minimizing their waiting time [2]. Both problems solve the same path-finding problem, just with different approaches. The problems are labeled NP-hard for general metric spaces [4] and have various applications in other industries.

According to [2], the formal definition of the problem is described by a complete undirected graph with N vertices described by G = (V, E), where V = $\{v_1, \ldots, v_N\}$ are the vertices. The vertices are connected by unique edges, defined as $e_{i,j} = (v_i, v_j) \in E$, $v_i \neq v_j$. The edge cost $d(e_{i,j}) = d(v_i, v_j)$ is non-negative, $d(E) \in \mathbb{R}_0^+$. Let us have a Hamiltonian path in G given by a sequence of vertices $\mathbf{x} = (x_0, \ldots, x_n)$, where $x_0 = s$ is the starting point (depot) and $n = N - 1$ is the number of customers. The cost of a path to the k-th vertex $\delta_k^{\mathbf{x}}$ via $\mathbf{x}$ is cumulative and is given by:

$$\delta_k^{\mathbf{x}} = \sum_{i=1}^{k} d(x_{i-1}, x_i) \tag{2.1}$$

The cost of the whole path through $\mathbf{x}$ is then given as:

$$\text{cost}(\mathbf{x}) = \sum_{k=1}^{n} \delta_k^{\mathbf{x}} = \sum_{k=1}^{n} \sum_{i=1}^{k} d(x_{i-1}, x_i) \tag{2.2}$$

The solution to TDP is then to find the optimal path $\mathbf{x}^*$ that minimizes the cost:

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathcal{H}(\pi)}{\arg\min} \left(\text{cost}(\mathbf{x})\right) \tag{2.3}$$

$\mathcal{H}$ is the set of all Hamiltonian paths across $\pi = (G, d, s)$, π is an instance that defines the graph of G, costs d, and the starting vertex s.

2.2 Solution approach

2.2.1 Hexaly Optimizer

The program with HO solves the problem for instances of TSPLIB [5]. The format of TSPLIB is shown in Figure 2.1. From the format, the program loads only the lines `DIMENSION`, `EDGE_WEIGHT_TYPE`, and the `NODE_COORD_SECTION` section, which specify the number of vertices, the method of calculating the edge cost between vertices, and the coordinates of each vertex. The `NODE_COORD_SECTION` contains one line per vertex, where the first number is the index of the vertex followed by its coordinates. In function **input**, the program obtains, from the instances, a distance matrix expressing the cost d of individual transitions between vertices, and $nbCities = N$ determines the number of vertices. The notations of the distance matrix in the instances are different, so we created a module **matrix_modul.hxm** for HM programs that contains functions to obtain distance matrices from any instance in TSPLIB. The module is imported into the program using the command `use matrix_modul as mm`, where `mm` serves as an alias to facilitate calling functions from the module.

```
NAME: st3
TYPE: TSP
COMMENT: 3-city problem
DIMENSION: 3
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1 64 96
2 80 39
3 69 23
EOF
```

Figure 2.1: Example of TSPLIB instance format

To make the path cost minimization more efficient, we modified the formula (2.2) as follows:

$$\begin{aligned}
\text{cost}(\mathbf{x}) &= \sum_{k=1}^{n}\sum_{i=1}^{k} d(x_{i-1}, x_i) = \\
&= (d(x_0, x_1)) + (d(x_0, x_1) + d(x_1, x_2)) + \cdots \\
&\quad + (d(x_0, x_1) + \cdots + d(x_{n-1}, x_n)) = \\
&= n \cdot d(x_0, x_1) + (n-1) \cdot d(x_1, x_2) + \cdots + 1 \cdot d(x_{n-1}, x_n) = \\
&= \sum_{i=0}^{n-1} (n-i) \cdot d(x_i, x_{i+1}) = \\
&= \sum_{i=1}^{n} (N-i) \cdot d(x_{i-1}, x_i) \qquad (2.4)
\end{aligned}$$

The resulting *cost(***x***)* is the same, but the formula is simpler and faster to process.

The main part of the code, function **model**, is shown in Figure 2.2. The figure shows variable *cities*, a modeling array that determines the vertices indices, and the conditions needed to define the problem, such as how many vertices to visit and specifying the index of the starting vertex (depot). The minimized objective function *obj* is defined using the

formula (2.4). The sum function creates an integer variable i that takes values from 1 to $nbCities - 1$. The vertex indices in *cities* change order during minimization.

```
function model {
    // A list variable: cities[i] is the index of the ith city in the
    ↪  tour
    cities <- list(nbCities);

    // All cities must be visited
    constraint count(cities) == nbCities;

    //start point == index
    local start_id = 0;
    constraint cities[0] == start_id;

    // Minimize the total distance
    obj <- sum(1..nbCities, i => (nbCities - i) *
    ↪  distanceMatrix[cities[i - 1]][cities[i]]);

    minimize obj;
}
```

Figure 2.2: TDP, function model code

To modify the solution search, we can change the seed of the pseudo-random generator by adjusting the predefined variable *hxSeed*. The default value of the *hxSeed* is 0. The solution search process can be modified by setting *hxSeed* to a different value, which is provided as an input argument. Furthermore, we want the program to sample how the cost of the optimal path changes depending on the duration of the program run. In the default mode, it is not possible to explicitly program such a thing, but we can bypass it. The predefined integer variable *hxTimeBetweenDisplays* specifies the period after how many seconds the search info is displayed, and the function **display** is called. The default value of *hxTimeBetweenDisplays* is 1. If it is necessary to change *hxTimeBetweenDisplays* or the value of another predefined variable, the variable and the new value can be entered as another input argument when the program is executed. So, in the function **display**, the current path cost is read and stored in an array. For a proper sampling of the path cost, the variables *count_t* and *obj_v_ar* were defined in the **param** function, where *count_t* is initialized with the value 0, and *obj_v_ar* represents an empty array. The display function is shown in Figure 2.3. The second line of the function checks whether the solver has already found a path by checking whether the variable cities contains vertex indices. The third line then checks the validity of the solution found; if the path is marked as `FEASIBLE` or `OPTIMAL`, the total cost of the path is stored in the *obj_v_ar* field. Otherwise, a value of -1 is stored. After adding the current sample, the value of *count_t* is incremented.

The sampled data are then saved to a text file in the format shown in Figure 2.4. The line `measure length` indicates how many samples of the cost have been measured, and below it are the individual samples. Next, below the line `cities order` are the indices of the path from the depot to the last customer. This format is sufficient if we assume that the sampling period is known and the same for all runs.

```
function display(){
    if(cities.value.count() > 0){
        if(hxSolution.status == "FEASIBLE" || hxSolution.status ==
        ↪  "OPTIMAL")
            obj_v_ar[count_t] = obj.value;
        else
            obj_v_ar[count_t] = -1;
        count_t += 1;
    }
}
```

Figure 2.3: function display code

```
measure length: 5
219078 217225 216985 216979 216898

cities order:
1 16 13 12 14 7 6 5 15 8 4 2 3 10 9 11
```

Figure 2.4: TDP, hexaly output text file example

2.2.2 Ms-GVNS and GILS-RVND

Multi-Start General Variable Neighborhood Search (Ms-GVNS) [2] and Greedy Iterative Local Search - Reactive Variable Neighborhood Descent (GILS-RVND) [6] are two approaches to solve the TDP. Ms-GVNS is a multi-search metaheuristic that uses different initial solutions to increase the probability of finding a good result. In each run, the algorithm systematically alternates between various types of local solution modifications, thus expanding the coverage of the space of possible solutions and overcoming stagnation in unfavorable regions by using random perturbation. In contrast, the GILS-RVND approach combines the construction of an initial solution using greedy randomized heuristics with subsequent refinement via iterated local search, where a set of transformation operators is randomly selected from the set of transformation operators in each iteration. With repeated restarts and built-in randomness, the algorithm efficiently explores the solution space and reduces the risk of getting stuck in the local optimum.

We have taken a program of solvers, written in C++, that uses these approaches from [2]. We made modifications so that the program, with a preset period, samples the cost of the path and the times when the sample was made. Furthermore, when the search for the optimal path is finished, all samples and the resulting path are saved in a text file in the format shown in Figure 2.5.

```
Mode: gils-rvnd
lowest weight: 216898
compute time [s]: 5
219078 217225 216985 216979 216898

Times:
1.00001 2.00002 3.00003 4.00005 5

Cities:
1 16 13 12 14 7 6 5 15 8 4 2 3 10 9 11
```

Figure 2.5: TDP, other solvers output text file example

The assumption for this format is the same as for the Hexaly format in Figure 2.4. The period is known and the same for all runs.

2.3 Computational evaluation

To compare the accuracy and speed of the solvers to find optimal paths, we let all solvers run 20 times for each instance for 120 seconds with a sampling period of 1 second. The sampling period determines after how many seconds of solving the path cost is recorded. We did all the runs for HO twice. Once we set $hxSeed = 0$. The second time, we set $hxSeed \neq 0$, where each run had set a different *hxSeed*. We executed the programs with bash scripts, so for a variety of *hxSeed* settings for each run, we used the bash function shuf in the format:

```
min_seed=1
max_seed=2147483647
SEED=$(shuf -i $min_seed-$max_seed -n 1)
```

The shuf function in this configuration generates pseudo-random numbers ranging from *min_seed* to *max_seed*. The seed cannot be negative or greater than the value of *max_seed*. The *SEED* value is loaded into the *hxSeed* variable as an input argument using $. Execution of the program in a bash script might look like:

```
hexaly tdp.hxm inFileName=st70.tsp hxSeed=$SEED hxTimeLimit
    =10 solFileName=sol.txt
```

An output text file with sampled path costs is created at the end of each solver run. From these files, for each solver and instance, we then determine the variables:

- Inst - instance name
- c_{best} - Reference best cost
- $\overline{c_g}$ - Mean of best-found costs
- mG_{best} - Mean percentage gap of best-found costs from c_{best}
- $\bar{t}$ - Mean time of finding the best cost

The values c_{best} are taken from [2]. The Mean percentage gap from c_{best} (mG(t)) represents the time waveform defined over the entire length of the measurement, using the average value Mean(t) at each time point t. The formula determines the value of mG(t):

$$mG(t) = 100 \cdot \frac{\text{Mean}(t) - c_{best}}{c_{best}} \tag{2.5}$$

The value mG_{best} uses the same formula (2.5), but it is a scalar value defining the Mean percentage gap of best-found costs, where the average value of Mean(t) is substituted by $\overline{c_g}$.

All result variables for each solver are shown in Table 2.1. The table shows that HO found the same or worse cost than other solvers. Also, in almost all cases, other solvers found their lowest cost faster than HO. Only for pr107 did HO find the exact solution before Ms-GVNS; the graphs for this instance are shown in Figure 2.8. In some figures, the graphs are zoomed in for a better display of details and do not show the entire time interval of 120 seconds. The figure shows the semi-transparent regions around the graphs, which are approximately 95 % confidence intervals defined as: $Mean \pm 2 \cdot SE$, where SE is the standard error of the mG. The graphs differ from the values in the table because the table shows the mean of the best values from each run, and the graphs show the mean values at a specific time. Therefore, it takes almost 20 seconds for Ms-GVNS to find the optimal solution, as shown in Figure 2.8, while it is 7.150 seconds in the table. Figure 2.6 shows how different the HO and HO seed waveforms can be. The mean waveform of the HO seed is slower and looks like a decreasing exponential function, while the HO has a more steeply changing waveform. Other graphs are in Figures 2.7 and 2.9, where HO seed has a lower end cost than HO. Neither figure is the HO or HO seed better than the other solvers, but it shows that the proper choice of seed is also essential in finding a solution.

Furthermore, we tried to give solvers instances with many vertices. The program with Ms-GVNS and GILS-RVND solvers had a problem already with an instance containing about 1300 vertices; the program shut down because it ran out of memory. Therefore, we used instances with a range close to 1000 vertices. The program could already process such instances, but it could not sample every second, so we modified the program to start sampling at time zero so that the first value always exists. During data processing, we then filled the empty samples with the value of the previous known sample. The program with HO had neither of those problems.

The results of the runs are shown in Table 2.2, but since we do not have a reference cost for these instances, we replaced the variable c_{best} in the table with the best-found cost from all runs c_{bf}. The table shows that HO found further solutions from the best than for smaller instances. On average, the best-found values were only found in the last 2 seconds before the end of the run, which means that HO may have found a better solution if we had set the run times longer. When we look at the average time to find the best solution, we see that the difference between HO and the other solvers is considerably smaller than in smaller instances. Figures 2.10 and 2.11 show that Ms-GVNS has the best initial cost, HO and HO seed consistently decrease, and GILS-RVND takes longer to approach the optimum. If we executed runs on shorter timescales, HO might find a better solution than GILS-RVND.

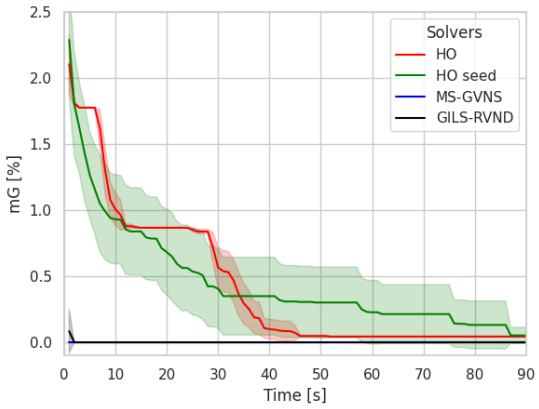

Figure 2.6: Comparison of TDP solvers on the kroD100 instance

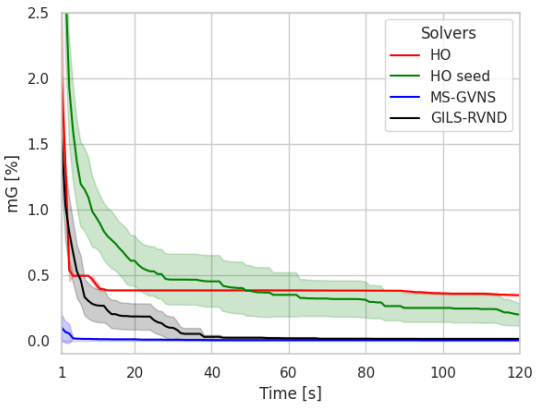

Figure 2.7: Comparison of TDP solvers on the rat195 instance

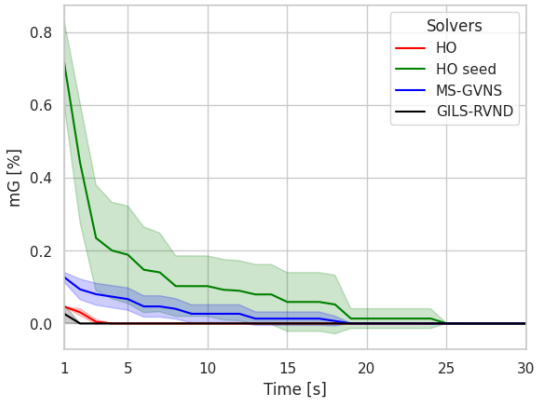

Figure 2.8: Comparison of TDP solvers on the pr107 instance

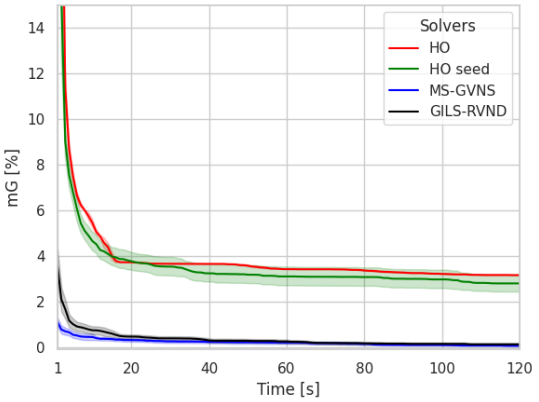

Figure 2.9: Comparison of TDP solvers on the lin318 instance

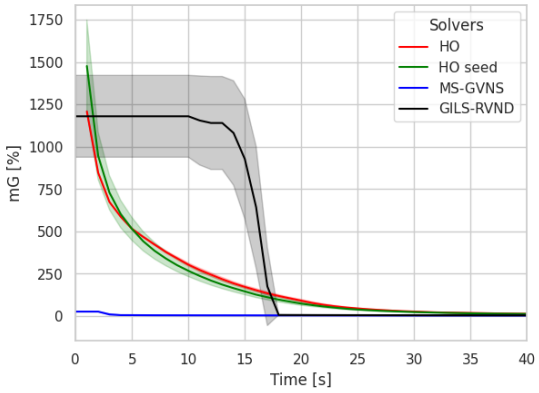

Figure 2.10: Comparison of TDP solvers on the dsj1000 instance

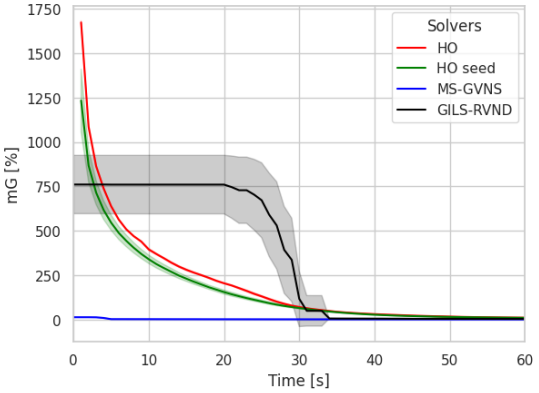

Figure 2.11: Comparison of TDP solvers on the pcb1173 instance

Inst	c_{best}	HO			HO seed			MS-GVNS			GILS-RVND		
		$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]
st70	19710	19710	0.000	5.000	19710	0.000	8.450	19710	0.000	1.000	19710	0.000	1.000
rat99	56573	56573	0.000	109.900	56646	0.129	57.600	56573	0.000	4.400	56573	0.000	1.750
rat195	216154	216906	0.348	100.500	216584	0.199	64.800	216164	0.005	29.250	216186	0.015	42.650
lin318	5569520	5745346	3.157	103.000	5725181	2.795	70.100	5572910	0.061	68.900	5576420	0.124	64.000
pr226	7101223	7101223	0.000	88.950	7102052	0.012	62.150	7101223	0.000	5.200	7101223	0.000	3.600
lin105	586751	586751	0.000	1.150	586751	0.000	12.750	586751	0.000	1.000	586751	0.000	1.000
pr439	17724562	18141665	2.353	98.350	18100071	2.119	92.000	17748090	0.133	61.850	17783436	0.332	62.300
kroD100	951609	952022	0.043	38.250	952002	0.041	36.100	951609	0.000	1.000	951609	0.000	1.050
att532	17449404	17821565	2.133	88.300	17923137	2.715	103.400	17536596	0.500	102.550	17582132	0.761	80.950
pr107	1981991	1981991	0.000	3.100	1981991	0.000	8.000	1981991	0.000	7.150	1981991	0.000	1.200

Table 2.1: Comparison of TDP solvers for TSPLIB instances

Inst	c_{bf}	HO			HO seed			MS-GVNS			GILS-RVND		
		$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]
pr1002	116810740	122707941	5.049	119.100	122153364	4.574	118.350	117964676	0.988	110.900	119272735	2.108	116.050
pcb1173	31464475	32763248	4.128	118.350	32825408	4.325	119.600	31705893	0.767	109.250	32176797	2.264	114.300
u1060	103551726	106447799	2.797	120.000	107757785	4.062	119.250	104852944	1.257	114.550	105964040	2.330	116.350
dsj1000	7799821597	8251460113	5.790	119.150	8115769296	4.051	117.650	7897908070	1.258	110.400	7916169161	1.492	114.600

Table 2.2: Comparison of TDP solvers for larger TSPLIB instances

Chapter 3

The Graph search problem

3.1 Definition

The Graph search problem (GSP) was introduced in [7]. The problem is formulated as TDP, with each vertex having an assigned probability of finding the object. This formulation is more suitable for searching an unknown space. We interpret this formulation to imply that each vertex is assigned a priority value. Informally, the problem is described as the deliveryman minimizing the time to deliver packages to customers, as in TDP, except that some customers have priority over others. Delivery time should still be as low as possible for everyone. The deliveryman determines the order of customers from the product of the priority value and the total travel time to the customer.

The formal definition of GSP is almost the same as the definition of TDP in Section 2.1, a complete undirected graph with N vertices described by G = (V, E). V = $\{v_1, \ldots, v_N\}$ are different vertices, each vertex v_i has a non-negative weight w_i. The vertices are connected by unique edges, defined as $e_{i,j} = (v_i, v_j) \in E$, $v_i \neq v_j$. The edge cost $d(e_{i,j}) = d(v_i, v_j)$ is non-negative, $d(E) \in \mathbb{R}_0^+$. Let us have a Hamiltonian path in the graph G given by a sequence of vertices $\mathbf{x} = (x_0, \ldots, x_n)$, where $x_0 = s$ is the starting point (depot) and $n = N - 1$ is the number of customers. The cost of a path to the k-th vertex $\delta_k^{\mathbf{x}}$ via $\mathbf{x}$ is cumulative and is given by formula (2.1). The cost of the path through $\mathbf{x}$ is then given as:

$$\text{cost}(\mathbf{x}) = \sum_{k=1}^{n} w_k \delta_k^{\mathbf{x}} = \sum_{k=1}^{n} w_k \sum_{i=1}^{k} d(x_{i-1}, x_i) \tag{3.1}$$

The optimal path is then given by formula (2.3).

3.2 Solution approach

3.2.1 Hexaly Optimizer

The program with HO is used to solve the problem for TSPLIB instances and getting the distance matrix the same way as the program for TDP described in Subsection 2.2.1. To solve GSP, we still need to get the vertex weights. The program reads these from a text file, where a random integer is generated on each line.

The function **model** is shown in Figure 3.1. We can see that the codes are not significantly different if we compare Figures 2.2 and 3.1. The most significant differences are in using the *priority_ar* array, which contains the vertex weights, and in the notation of the minimization formula. Formula (3.1) can not be simplified, as did formula (2.2). In the minimized objective function *obj*, two sums with different index range notations exist. The first uses the notation 1...*nbCities*, which denotes the range from 1 to $nbCities - 1$;

the upper bound is not included. The second uses the notation $1..k$, which implies the range from 1 to k inclusive; the upper bound is included. The difference between these notations is in the inclusion or non-inclusion of the upper bound of the index.

```
function model {
    // A list variable: cities[i] is the index of the ith city in the
    ↪  tour
    cities <- list(nbCities);

    // All cities must be visited
    constraint count(cities) == nbCities;

    //start point == index
    local start_id = 0;
    constraint cities[0] == start_id;

    // Minimize the total distance
    obj <- sum(1...nbCities, k => priority_ar[cities[k]] * sum(1..k, i =>
    ↪  distanceMatrix[cities[i - 1]][cities[i]]));

    minimize obj;
}
```

Figure 3.1: GSP, function model code

Since the program is written in HM in the default mode, it handles saving data to a text file and setting the predefined variable *hxSeed* like the TDP program described in Subsection 2.2.1. The output text file has the same format as shown in Figure 2.4, with the same assumptions.

3.2.2 Ms-GVNS

The program described in [2] contains the Ms-GVNS solver, which can solve TDP and GSP problems. To run Ms-GVNS as a GSP solver, it is necessary not to set an argument specifying the unit vertex weights. Since this is the same program we used to solve TDP, the modifications described in subsection 2.2.2 are made. These modifications are used subsequently to solve GSP. Thus, the output text file has the same format as in Figure 2.5.

3.3 Computational evaluation

For solver comparison, we again chose a setting of 20 runs for each instance for 120 seconds with a sampling period of 1 second. The resulting data is shown in Table 3.1. In the table, we see that HO found a solution that was very far from c_{bf} for some of the smaller instances. The graphs for these instances are shown in Figures 3.2 and 3.5. It is a problem where HO could not find a valid solution in time before sampling, so the program saves the value -1 to the output file as an indication of a non-valid result, and these values are not plotted in the graph. Therefore, the graph looks like HO started looking for a solution with a delay. For the HO seed, the boundary when a valid cost is found is ambiguous because different seeds found solutions at various times. Figure 3.3 shows the cost values of all runs for the att532 instance. We can see that each run finds a different

first valid value at a different time. That is why the waveform looks like the function is gradually increasing and then decreasing. Since the deviation of the HO costs from the c_{bf} cost remained very high at the end of the measurement, we executed an extended run of the HO for the att532 instance, taking one hour, to verify its convergence behavior. The progress of this run is shown in Figure 3.4. The black vertical dashed line indicates the time limit of 120 seconds at which the runs usually ended. The graph shows that the algorithm indeed converges but very slowly. It is not directly apparent from the graph, but a mG value was approximately 13 % at the end of the run.

To see for how large instances this problem occurs, we plotted the graphs for instances lin318, pr226, and rat195 in Figures 3.6, 3.7, and 3.8. Figures 3.6 and 3.7 show that there is still a problem of not finding a solution before sampling. When we look at the graphs in Figure 3.8, we can see that the waveforms look better, and there is no sign of invalid samples. However, when we look at the beginning of each HO run in Figure 3.9, we can see that sometimes the first sample is invalid, and a run is delayed. Since it is only one sample, the rest of the run is not affected as much as the previous instances. From this, we can see that for instances that have less than 200 vertices, HO can find a solution with little or no influence from invalid costs.

Next, we tried to see how the data and waveforms would look for larger instances. We can see from the table that all values are far from c_{bf}, and in one case, HO could not find a solution. The graphs of the instances dsj1000 and pcb1173 are shown in Figures 3.10 and 3.11. In Figure 3.10, we can see the same problem; only the valid solution is found later than in previous instances, and in Figure 3.11, we can see the case when HO was not even able to find a valid solution during the runs.

According to the results of solving GSP and TDP, we assume that the problem of invalid values occurs due to the large number of vertices and the complexity of formula (3.1). For each Multi-Goal problem, there will be some number of vertices when HO can no longer find a solution within 1 second before sampling begins. Regarding the complexity of the formula, if we compare formulas (3.1) and (2.2), we can see that (3.1) has a more complex notation, and we assume that the solver takes longer to process the formula. The weights of the vertices in formula (3.1) shouldn't affect the complexity much because when we were creating the program to solve GSP using HO, we accidentally wrote it with the formula $\text{cost}(\mathbf{x}) = \sum_{i=1}^{n} w_i \cdot (N - i) \cdot d(x_{i-1}, x_i)$. With this formula, the program solved even larger cases than those shown in the table without any delay. Therefore, we believe that finding a solution also depends on the complexity of the minimized formula.

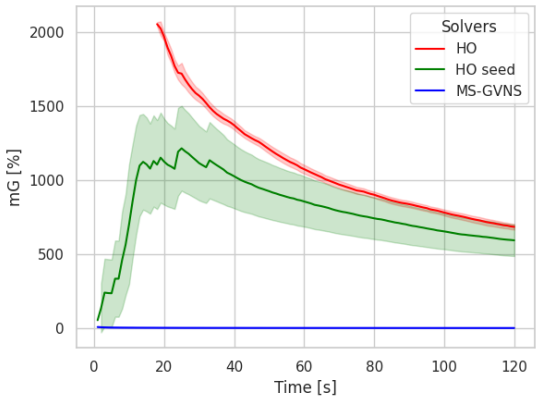

Figure 3.2: Comparison of GSP solvers on the att532 instance

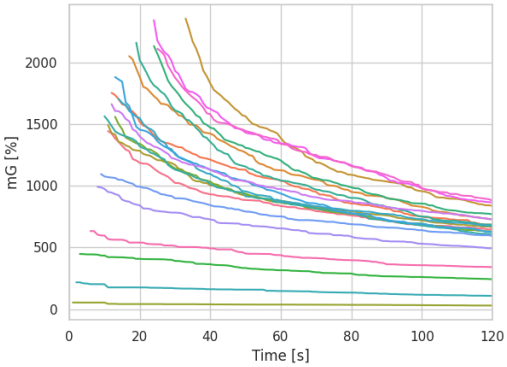

Figure 3.3: GSP, HO seed separated runs of the att532 instance

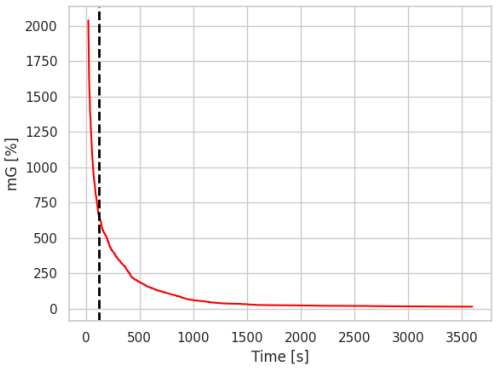

Figure 3.4: GSP, HO single 1 h run of the att532 instance

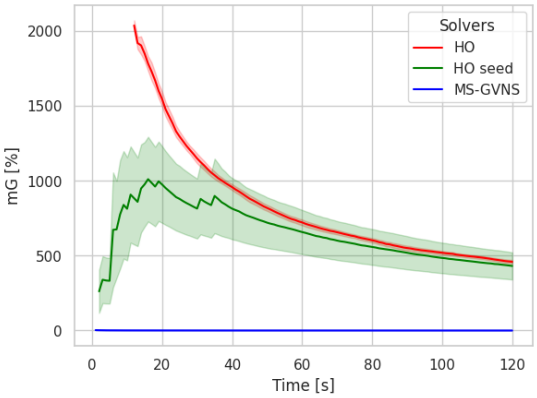

Figure 3.5: Comparison of GSP solvers on the pr439 instance

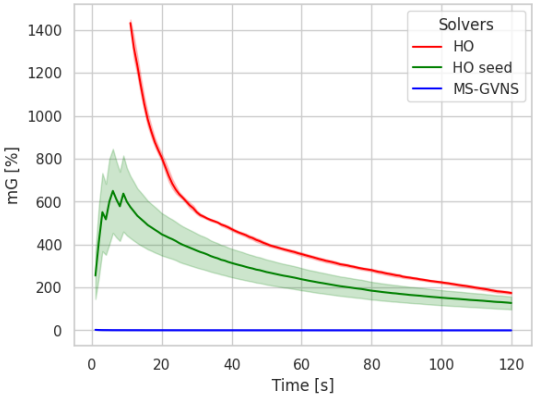

Figure 3.6: Comparison of GSP solvers on the lin318 instance

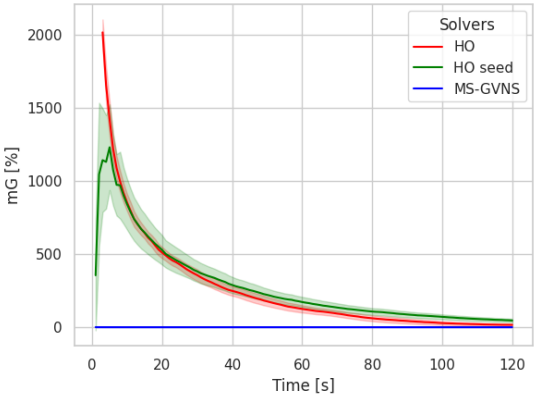

Figure 3.7: Comparison of GSP solvers on the pr226 instance

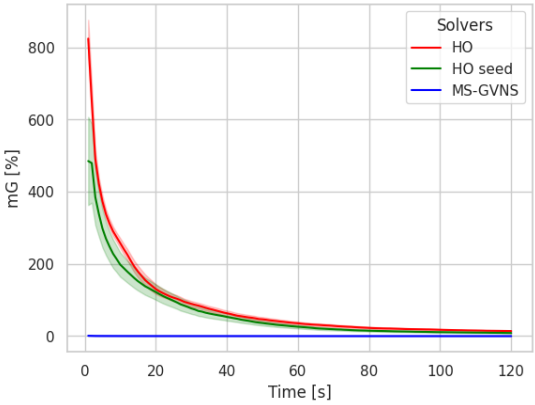

Figure 3.8: Comparison of GSP solvers on the rat195 instance

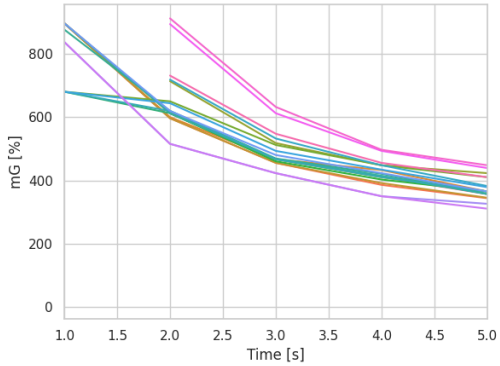

Figure 3.9: GSP, HO separated runs of the rat195 instance

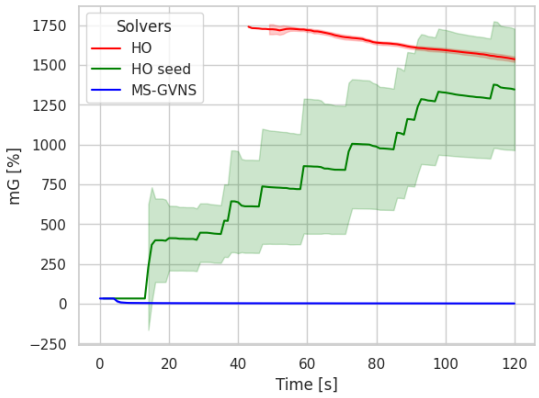

Figure 3.10: Comparison of GSP solvers on the dsj1000 instance

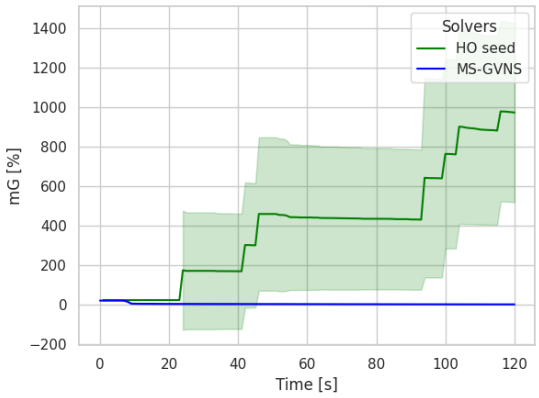

Figure 3.11: Comparison of GSP solvers on the pcb1173 instance

Inst	c_{bf}	HO			HO seed			MS-GVNS		
		$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]
st70	102473	102553	0.078	61.950	103218	0.727	59.100	102473	0.000	1.000
rat99	269138	269301	0.061	39.450	274474	1.983	77.650	269138	0.000	1.000
rat195	1107983	1263779	14.061	119.750	1203406	8.612	118.350	1108368	0.035	55.750
lin318	30647368	84036495	174.205	120.000	69746888	127.579	119.700	30704324	0.186	74.900
pr226	39644390	45965619	15.945	119.450	57786614	45.762	119.550	39651579	0.018	53.150
lin105	3323092	3339524	0.494	103.650	3359310	1.090	80.150	3323092	0.000	3.350
pr439	98853941	552629006	459.036	120.000	525081240	431.169	119.950	99162323	0.312	76.700
kroD100	4794046	4837983	0.916	114.950	4872780	1.642	87.800	4794046	0.000	1.000
att532	92834887	727192155	683.318	119.800	642826152	592.440	119.900	93241920	0.438	109.800
pr107	10038865	10072487	0.335	89.550	10077896	0.389	77.350	10038865	0.000	1.000
pr1002	622963153	6988958438	1021.890	119.650	9016005418	1347.277	119.550	629515052	1.052	113.900
pcb1173	162810159	—	—	—	1747544459	973.363	110.000	164526072	1.054	111.300
u1060	563381902	9262798412	1544.142	119.850	8167241404	1349.681	113.278	567490790	0.729	113.900
dsj1000	42160373602	689760265379	1536.039	119.550	609630525248	1345.980	118.944	42579949421	0.995	112.200

Table 3.1: Comparison of GSP solversfor TSPLIB instances

Chapter 4

The Generalized TSP

4.1 Definition

The Generalized TSP (GTSP) [3] is an extension of TSP. TSP Searches for the shortest path in which a traveling salesman has to visit all customers just once and return to the starting point. Meanwhile, in GTSP, customers are divided into non-overlapping groups (sets), and the shortest path is searched, in which the traveling salesperson visits just one customer from each set once and forms a closed loop. How a GTSP solution could look like is shown in Figure 4.1.

The formal definition of the GTSP is described by a complete undirected graph with N vertices described by G = (V, E). V = $\{v_1, \ldots, v_N\}$ are different vertices, they are distributed in m sets S = $\{S_1, \ldots, S_m\}$, $S_i \neq \emptyset$, $S_i \cap S_j = \emptyset$, $i \neq j$ and $\cup_{i=1}^{m} S_i = V$. The vertices of the set S_i are denoted as $\{s_1^i, \ldots, s_{m_i}^i\}$ [8]. The vertices are connected by unique edges, defined as $e_{i,j} = (v_i, v_j) \in E$, $v_i \neq v_j$. The edge cost $d(e_{i,j}) = d(v_i, v_j)$ is non-negative, $d(E) \in \mathbb{R}_0^+$. The shortest path is determined by the formula:

$$\begin{aligned} \arg\min_{\sigma \in \Sigma} L &= \left(\sum_{i=1}^{m-1} d(s^{\sigma_i}, s^{\sigma_{i+1}}) \right) + d(s^{\sigma_m}, s^{\sigma_1}) \\ \text{subject to:} & \\ & \sigma = (\sigma_1, \ldots, \sigma_m) \in \Sigma, 1 \leq \sigma_i \leq m, \sigma_i \neq \sigma_j, i \neq j \\ & s^{\sigma_i} \in S_{\sigma_i}, S_{\sigma_i} = \{s_1^{\sigma_i}, \ldots, s_{m_{\sigma_i}}^{\sigma_i}\}, S_{\sigma_i} \in S \end{aligned} \tag{4.1}$$

σ is a permutation of the indices of sets S, σ_i is a i-th index of the sets S in the path, Σ is the set of all permutations, S_{σ_i} is the σ_i-th set, and s^{σ_i} is a vertex from the S_{σ_i} set.

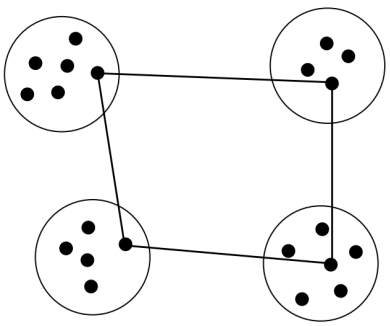

Figure 4.1: Illustration of the GTSP solution [8]

4.2 Solution approach

4.2.1 Hexaly Optimizer

The program with HO solves GTSP for instances from the GTSP-LIB and LARGE-LIB libraries. GTSP-LIB was found on [9] and LARGE-LIB on [10]. Both libraries are based on TSPLIB, but the instances additionally contain sections where vertices are divided into sets. The format of the instance is shown in Figure 4.2. The figure shows that compared to the TSPLIB format, only the `GTSP_SETS` line and the `GTSP_SET_SECTION` section, which specify the number of sets and the distribution of vertex indices into individual sets, are extra. In the `GTSP_SET_SECTION` section, the first number indicates the index of the set, followed by the individual vertex indices belonging to that set, and a value of -1 indicates the end of the set. The program in function **input** extracts the distance matrix, $nbCities = N$, $nbSets = m$ and the *sets_ar* array, which contains the indices of vertices in each set.

```
NAME: st3
TYPE: GTSP
COMMENT: 3-city problem
DIMENSION: 3
GTSP_SETS: 2
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1 64 96
2 80 39
3 69 23
GTSP_SET_SECTION:
1 2 3 -1
2 1 -1
EOF
```

Figure 4.2: Example of GTSP-LIB and LARGE-LIB instance format

The function **model** is shown in Figure 4.3. The for loop with *sets_ar* creates conditions that declare that the program can only visit one vertex from each set. The following condition specifies that the program can visit only as many vertices as there are sets. A formula for the variable L from (4.1) was used for the solution search. The formula was modified to use the indices $i - 1$ and i instead of the indices i and $i + 1$. The program handles writing to a text file in the same way as the program in subsection 2.2.1, and the format of the output text file is shown in Figure 2.4.

4.2.2 GLNS

Generalized large neighborhood search (GLNS) is the method for solving GTSP described in [3]. GLNS uses the Adaptive Large Neighborhood Search (ALNS) framework. The algorithm iteratively modifies the current solution using destruction and construction heuristics, the selection of which is adaptively guided by their past success, thus enabling efficient solution space exploration even for difficult instances.

The original GLNS code is written in Julia. We decided to use a modified version from [11], which is written in C++. The code is implemented only as a submodule, so we had to write a new program that uses this submodule. We created functions for the program to process data from the instance file by rebuilding functions from the **matrix_modul.hxm** module into C++. The GLNS module contains a pseudo-random number generator, and we

```
function model {
    // A list variable: cities[i] is the index of the ith city in the
    ↪  tour
    cities <- list(nbCities);

    // visit only one city from each set
    for [s in sets_ar]
        constraint sum[i in s](contains(cities, i)) == 1;

    // Visit only as many cities as there are sets
    constraint count(cities) == nbSets;

    // Minimize the total distance
    obj <- sum(1...nbSets, i => distanceMatrix[cities[i - 1]][cities[i]])
            + distanceMatrix[cities[nbSets - 1]][cities[0]];

    minimize obj;

}
```

Figure 4.3: GTSP, function model code

have modified it so that the generator's seed is set according to the current time each time the program is executed. We get more variability in the results of the runs with this setting. We also modified the GLNS module to produce an output text file in the format shown in Figure 2.5 when the path search is finished, with one difference. The GLNS module has only one solver but three modes: Fast, Middle, and Slow. Modes set the optimization intensity in the GLNS through parameters specifying the number of iterations, trials, the size of solution interventions, and local optimizations. Thus, the user can choose between a faster run with lower accuracy (Fast) and a more computationally intensive but higher quality solution (Slow). So, in the `Mode:` line, instead of the solver used, the settings used will be written.

4.3 Computational evaluation

For solver comparison, we again chose a setting of 20 runs for each instance for 120 seconds with a sampling period of 1 second. The resulting data is shown in Table 4.1. The values c_{best} are taken from [3], and there are instances of GTSP-LIB and LARGE-LIB. GLNS was executed in two modes. Each mode, like HO, was run 20 times for each instance. The first time for GLNS was in Slow mode, and the second time was in Fast mode, so we could compare the difference between solutions that should be the most accurate and fastest.

In the table, we can see that for instances from GTSP-LIB, both solvers found costs close to the optimal cost. We can even see cases where the solvers found a better solution than the reference. This may be because most of these instances are in `GEO` format, for which we computed the distance matrices using the function from [1], which is different from the pseudo-code described in TSPLIB documentation [12], which we used to make functions for almost all types of instances. Both codes create a matrix for the same type of instances, so the distance matrices should be practically the same; the difference may be due to the use of different earth radii. However, the 45tsp225 instance is of type `EUC_2D`,

and both solvers found better costs than the reference, which we do not know how to explain because we followed the documentation [12] for such instances and other instances of type `EUC_2D`, like 46pr226 or 64lin318, we got the reference cost at best. The graphs of the 45tsp225 instance are shown in Figure 4.4. The figure shows that all the solvers found costs lower than the reference very early on. GLNS has some termination conditions because the figure shows how the GLNS fast run ended after less than 10 seconds of running.

More graphs are shown in Figures 4.5, 4.6, and 4.7. Figure 4.5 shows the case where all solvers found the reference cost in the first second, and these are valid values for all solvers. Figure 4.6 shows the graphs of instance 35si175, which has a similar number of sets and vertices as instance 32u159, but here, the solvers took longer to find the reference cost, and the HO took considerably longer. This is most likely due to how the vertices are distributed in the sets. Figure 4.7 shows the graphs of instance 89rbg443, which is the instance where all the solvers found the solution furthest from the reference cost, but both HO and HO seed found a better solution and faster than GLNS slow and GLNS fast.

Furthermore, we tested how HO and GLNS manage large instances. The program with GLNS does not have memory problems like the program with Ms-GVNS and GILS-RVND so that it can manage much larger instances. For large instances, we can better see the dependency between the number of vertices and the size of the sets. The dependency can be seen in the table for the 31C3k and 10C10k instances. Their final cost and time of finding a solution differ from the rest of the LARGE-LIB instances. The instances have many vertices but few sets, so finding a solution for them is easier. The differences in the time of finding a solution can be seen in Figures 4.8 and 4.9; for 10C1k, it took at most 2 seconds, and for GLNS fast, it took less than a second. From Figure 4.9, we can see that for large instances with a large number of sets, GLNS is not as fast as HO and that GLNS has a delayed first sample. This means GLNS can not sample on time. In addition, we can see that the program with GLNS exceeded the set limit of 120 seconds, which means that the current program does not expect to process such large instances. For runs with large instances, we have noticed that programs take longer to load and process data from instances before they start looking for a solution. This preparation time is not included in tables or graphs and is most noticeable for large instances. Therefore, we measured how long it takes a program with HO and a program with GLNS to process instance 2370rl11849 before the programs are ready to record the first sample. Both programs started to resolve the GTSP approximately 40 seconds after execution. Hence, these delays may not be considered when comparing solvers, as they are similar.

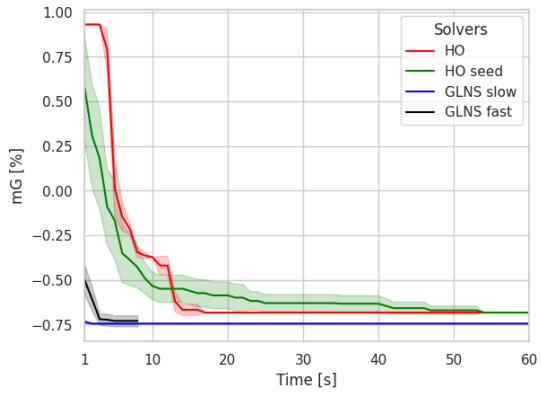

Figure 4.4: Comparison of GTSP solvers on the 45tsp225 instance

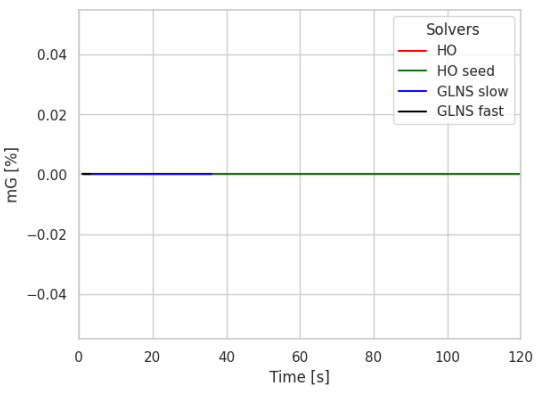

Figure 4.5: Comparison of GTSP solvers on the 32u159 instance

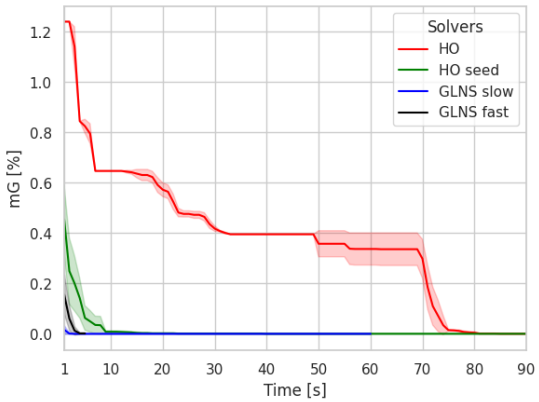

Figure 4.6: Comparison of GTSP solvers on the 35si175 instance

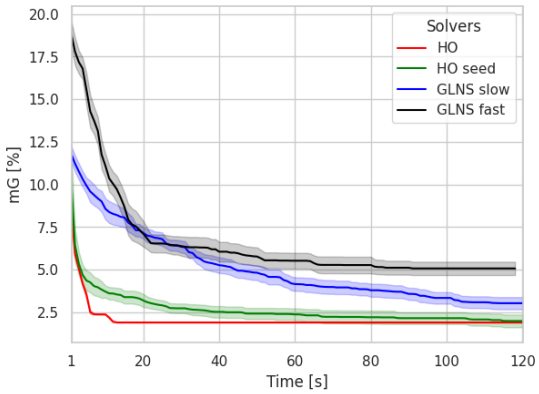

Figure 4.7: Comparison of GTSP solvers on the 89rbg443 instance

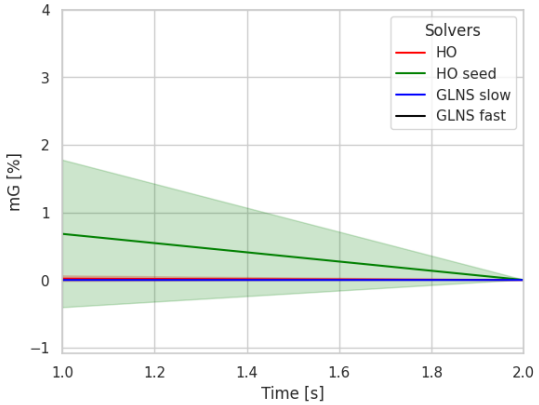

Figure 4.8: Comparison of GTSP solvers on the 10C1k instance

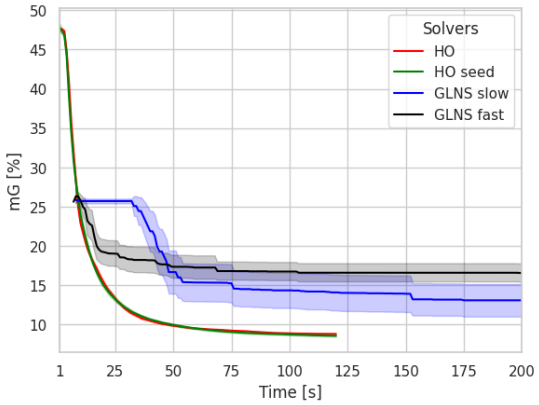

Figure 4.9: Comparison of GTSP solvers on the 2370rl11849 instance

4.3.1 Solver quality by instance type

The complexity of finding a GTSP solution is determined by the number of vertices and the number of sets and vertices in each set. Therefore, we decided to analyze the instances from Table 4.1 and compare the efficiency of the HO and GLNS solvers in terms of the difficulty of these instances. The asymptotic complexity of GTSP is given by the formula:

$$O\left(m! \cdot \prod_{i=1}^{m} |S_i|\right) \tag{4.2}$$

m is the number of sets and $|S_i|$ is the number of vertices of the set S_i. The asymptotic complexity of the problem would take very high values in a direct calculation, making it difficult to interpret and compare. To obtain more straightforward results, we decided to logarithm the complexity and get the formula:

$$O\left(\log\left(m! \cdot \prod_{i=1}^{m} |S_i|\right)\right) = O\left(\sum_{j=1}^{m} \log(j) + \sum_{i=1}^{m} \log(|S_i|)\right) \tag{4.3}$$

We created 3D scatter plots based on the number of vertices, the number of sets, and the logarithmic asymptotic time complexity, showing which solver is more efficient for each

instance. The plots are shown in Figures 4.10 and 4.11. In these plots, we compare which solver found the better solution; if both solvers achieved the same result, the decision was based on the time to find the solution. Figure 4.10 shows that for GTSP-LIB instances, the GLNS slow almost always achieves better results than HO. However, we can also observe that HO can find a better solution for very small or, on the contrary, larger instances. Figure 4.11 shows the instances from LARGE-LIB, where we can see that when the number of vertices is large, and the number of sets is small, GLNS fast performs better. However, when the number of sets increases, HO seed becomes more efficient.

From this, we could say that GLNS fast is better for instances with fewer sets but more vertices. GLNS slow seems more efficient for instances with more sets, but the number of vertices is not that large. HO is then good for instances with larger sets and vertices, and with the right seed setting, HO seed can be even better than HO.

Inst	c_{best}	HO			HO seed			GLNS slow			GLNS fast		
		$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]
39rat195	854	854	0.000	5.600	854	0.000	8.900	854	0.000	1.000	854	0.000	1.050
40d198	10557	10557	0.000	8.350	10557	0.000	28.750	10557	0.000	1.000	10557	0.000	1.950
41gr202	23301	23256	-0.193	4.150	23256	-0.193	7.250	23256	-0.193	1.000	23256	-0.193	1.350
35si175	5564	5564	0.000	75.850	5564	0.000	11.300	5564	0.000	1.250	5564	0.002	2.450
89pcb442	21657	21698	0.189	57.200	21749	0.426	67.050	21687	0.139	51.400	21745	0.407	37.450
46gr229	71972	71870	-0.142	23.600	71870	-0.142	8.750	71870	-0.142	1.050	71870	-0.142	2.050
53gil262	1013	1013	0.000	54.600	1014	0.123	45.350	1013	0.000	1.250	1013	0.000	3.200
45tsp225	1612	1601	-0.682	13.050	1601	-0.682	17.350	1600	-0.744	1.150	1600	-0.729	2.300
32u159	22664	22664	0.000	1.000	22664	0.000	1.000	22664	0.000	1.000	22664	0.000	1.000
88pr439	60099	60099	0.000	40.150	60141	0.069	57.150	60099	0.000	23.700	60102	0.005	38.150
80rd400	6361	6453	1.446	94.250	6381	0.311	52.200	6361	0.000	11.400	6362	0.017	21.000
89rbg443	632	644	1.899	12.200	645	1.986	56.400	651	3.030	93.150	664	5.071	53.250
72rbg358	693	693	0.000	2.100	693	0.022	26.600	693	0.036	60.650	697	0.519	21.350
60pr299	22615	22635	0.088	57.150	22623	0.036	39.000	22615	0.000	4.050	22620	0.021	10.650
64lin318	20765	20765	0.000	113.200	20787	0.104	35.600	20765	0.000	2.350	20765	0.000	9.150
53pr264	29549	29549	0.000	8.000	29549	0.000	3.200	29549	0.000	1.000	29549	0.001	4.200
36brg180	4420	4420	0.000	1.000	4420	0.000	1.000	4420	0.000	1.050	4420	0.000	1.200
87gr431	101946	101797	-0.146	45.200	102036	0.088	69.450	101780	-0.163	24.050	101823	-0.121	36.500
40kroA200	13406	13406	0.000	4.000	13406	0.000	6.700	13406	0.000	1.000	13408	0.018	2.400
84fl417	9651	9651	0.000	8.100	9651	0.000	10.250	9651	0.000	1.200	9654	0.031	32.500
45ts225	68340	68340	0.000	34.800	68349	0.013	27.100	68340	0.000	1.950	68352	0.018	3.500
40kroB200	13111	13117	0.046	16.250	13112	0.011	49.450	13111	0.000	1.000	13111	0.000	2.400
31pr152	51576	51576	0.000	4.050	51576	0.000	5.350	51576	0.000	1.000	51576	0.000	1.050
35ftv170	1205	1205	0.000	1.000	1205	0.000	2.550	1205	0.000	1.000	1205	0.000	1.050
46pr226	64007	64007	0.000	1.000	64007	0.000	1.000	64007	0.000	1.000	64007	0.000	1.050
81rbg403	1170	1170	0.000	12.400	1170	0.000	18.400	1170	0.000	1.200	1170	0.000	12.750
56a280	1079	1079	0.000	38.650	1079	0.009	38.950	1079	0.000	2.400	1079	0.000	3.650
65rbg323	471	477	1.274	52.650	473	0.467	54.050	472	0.127	61.250	476	0.977	14.350
1183rl5915	309243	331158	7.087	49.700	331004	7.037	104.950	338001	9.299	78.800	348490	12.691	55.850
2370rl11849	427996	465720	8.814	119.800	464703	8.577	119.800	484028	13.092	110.450	498926	16.573	51.000
10C1k	2522585	2522585	0.000	1.050	2522585	0.000	1.250	2522580	-0.000	1.150	2522580	-0.000	0.200
100C10k	6158999	6454877	4.804	109.450	6338535	2.915	66.400	6466194	4.988	112.550	6217627	0.952	81.200
31C3k	3553142	3553142	0.000	12.950	3553142	0.000	21.800	3558437	0.149	48.500	3553552	0.012	9.400

Table 4.1: Comparison of GTSP solvers for GTSP-LIB and LARGE-LIB instances

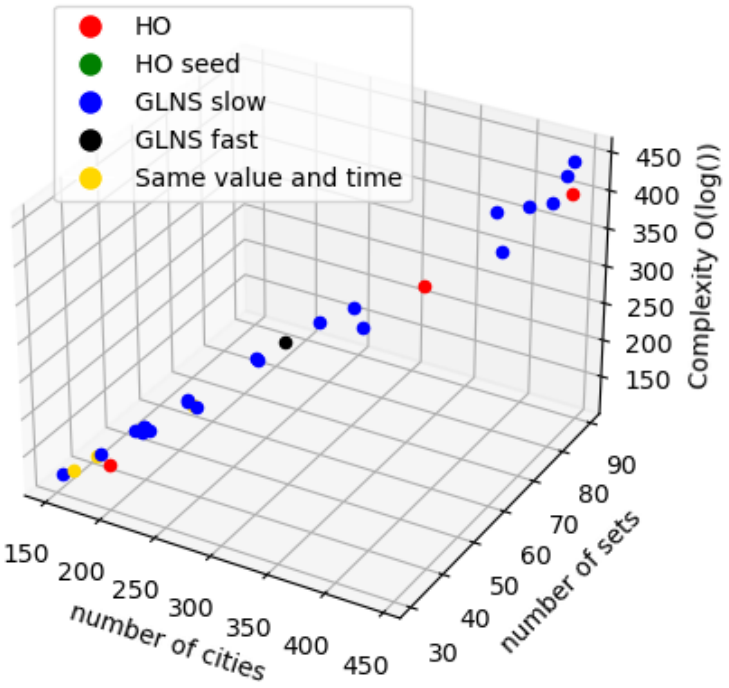

Figure 4.10: 3D scatter plot of GTSP-LIB instances

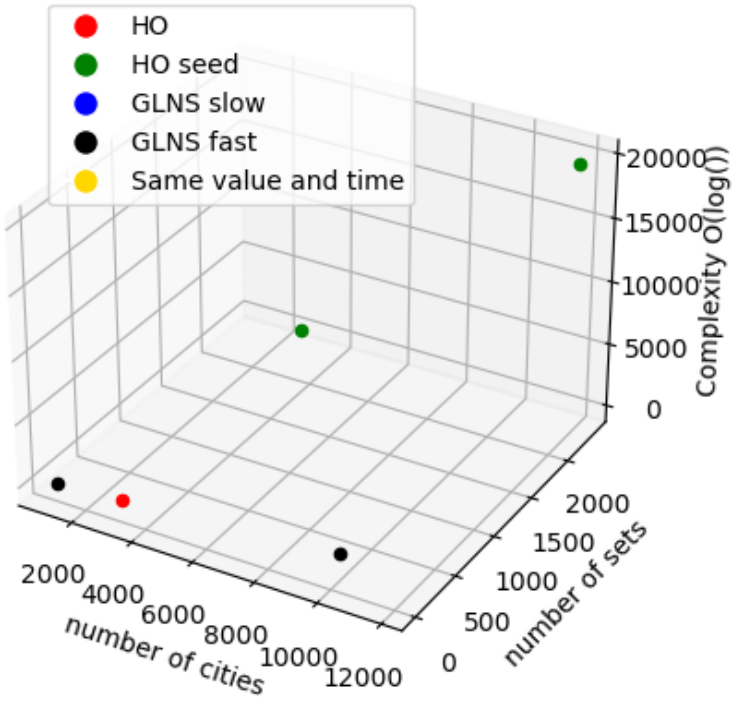

Figure 4.11: 3D scatter plot of LARGE-LIB instances

Chapter 5

The Generalized Graph search problem

5.1 Definition

The Generalized Graph search problem (GGSP) is an extension of GSP. In this problem, the goal of the deliveryman is to find a path that minimizes total delivery time and considers customers' priorities, as in GSP. However, customers are divided into disjoint sets, and the deliveryman visits just one customer from each set, corresponding to the GTSP principle.

The formal definition of GGSP is a combination of definitions from Sections 3.1 and 4.1, a complete undirected graph with N vertices described by G = (V, E). V = $\{v_1, \ldots, v_N\}$ are different vertices, each vertex v_i has a non-negative weight w_i, and they are distributed in M sets S = $\{S_0, \ldots, S_{M-1}\}$, $S_i \neq \emptyset$, $S_i \cap S_j = \emptyset$, $i \neq j$ and $\cup_{i=0}^{M-1} S_i = V$. The set S_0 contains the starting point (depot), and the number of customer sets to be visited is $m = M - 1$. The vertices of the set S_i are denoted as $\{s_0^i, \ldots, s_{m_i}^i\}$. The vertices are connected by unique edges, defined as $e_{i,j} = (v_i, v_j) \in E$, $v_i \neq v_j$. The edge cost $d(e_{i,j}) = d(v_i, v_j)$ is non-negative, $d(E) \in \mathbb{R}_0^+$. The shortest path is determined by the formula:

$$\begin{aligned} &\arg\min_{\sigma \in \Sigma} L = \sum_{k=1}^{m} w_k \delta_k^{\mathbf{x}} = \sum_{k=1}^{m} w_k \sum_{i=1}^{k} d(s^{\sigma_{i-1}}, s^{\sigma_i}) \\ &\text{subject to:} \\ &\qquad \sigma = (\sigma_0, \ldots, \sigma_m) \in \Sigma, 1 \leq \sigma_i \leq m, \sigma_i \neq \sigma_j, i \neq j \\ &\qquad s^{\sigma_i} \in S_{\sigma_i}, S_{\sigma_i} = \{s_0^{\sigma_i}, \ldots, s_{m_{\sigma_i}}^{\sigma_i}\}, S_{\sigma_i} \in S \end{aligned} \tag{5.1}$$

σ is a permutation of the indices of sets S, σ_i is a i-th index of the sets S in the path, Σ is the set of all permutations, S_{σ_i} is the σ_i-th set, and s^{σ_i} is a vertex from the S_{σ_i} set. For the set S_0, s^{σ_0} is always the vertex, which is the depot.

5.2 Solution approach

5.2.1 Hexaly Optimizer

The program with HO solves GGSP for instances similar to GTSP-LIB. The difference is that there is a section with weights for all points in the instance under the list of sets. To process these instances, we slightly modified the function **input** so that the vertex weights are read from the instance, not the adjacent text file. These instances are specially created for this program.

The function **model** is shown in Figure 5.1. We see that there are the same conditions as for GTSP. We used the L formula from (5.1) for the search, which we modified to use the indices $i-1$ and i instead of i and $i+1$.

```
function model {
    // A list variable: cities[i] is the index of the ith city in the
    ↪  tour
    cities <- list(nbCities);

    // Visit only one city from each set
    for [s in sets_ar]
        constraint sum[i in s](contains(cities, i)) == 1;

    // Visit only as many cities as there are sets
    constraint count(cities) == nbSets;

    //start point == index
    local start_id = 0;
    constraint cities[0] == start_id;

    // Minimize the total distance
    obj <- sum(1...nbSets, k => priority_ar[cities[k]] * sum(1..k, i =>
    ↪  weight_matrix[cities[i - 1]][cities[i]]));

    minimize obj;
}
```

Figure 5.1: GGSP, function model code

The program creates an output text file like the program in subsection 2.2.1, and the format of the output text file is shown in Figure 2.4.

5.2.2 GLNS_GGSP

GLNS for GGSP (GLNS_GGSP) is a solver developed in [13]. The solver is a modified version of the GLNS algorithm, adapted to solve the GGSP problem and possibly its variants with dynamic weights. This solver, like the GLNS described in subsection 4.2.2, allows execution in Fast, Medium, and Slow modes. Based on an agreement, the author provided randomly generated input instances with fixed weights and, subsequently, the results of the runs. The instances provided are those stored in **ggsp_instances.zip**. The results were delivered as CSV files, which we converted into a format shown in Figure 2.5 for further processing.

5.3 Computational evaluation

For solver comparison, we chose a setting of 10 runs of 120 seconds with a sampling period of 1 second. We only got 10 runs per instance, executed in the Medium mode, from the author of GLNS_GGSP, and we adapted our analysis accordingly. Next, the GLNS_GGSP runs did not have a fixed sampling time, so we then converted the results to a per-second sampling format by filling the empty samples with the previously known value, as in section 2.3, when the program could not sample every second.

The results of the runs are shown in Table 5.1. The found costs are real numbers because the weights used are also real numbers. From the table, we see that HO found lower costs for instances with fewer points and sets. However, as the number of vertices and sets increases, the GLNS_GGSP produces better results. Graphs for smaller instances are shown in Figures 5.2 and 5.3. We can see that HO found the best solution in less than 10 seconds; HO seed is a bit slower but has a better initial cost. Figures 5.4, 5.5, and 5.6 show that as the number of sets and vertices increases, the HO and HO seed can not find the best solution within the time limit anymore. In some cases, the HO seed found a valid solution after a delay, leading to an increase in the average cost value in the graph, as in Section 3.3. From the graphs, it may look like the GLNS_GGSP waveforms are constant. It is indeed the case for smaller instances, as most samples were sampled during the first second of the run, and the cost was already constant for the rest of the run. From these quickly recorded samples, only the lowest value was selected for our format. The cost decrease was recorded for the larger instances, but the HO is often started with a much larger initial value. Therefore, the decrease is not visible in the graph.

Since the speed of finding a solution in this case depends on multiple parameters, we create the same 3D scatter plot for the instances as for GTSP in Subsection 4.3.1, using the formula (4.3). The plot is shown in Figure 5.7. In the figure, we can see that the instances are divided into three groups, and only in the one with the lowest difficulty did HO find better results than the GLNS_GGSP. Due to the limited range of instances, we cannot be certain if the HO method would achieve better results with larger instances as it did for the GTSP problem, but based on the graphs for GGSP, we would conclude that with larger instances, the delay due to invalid samples would be more pronounced, similar to the GSP case. The minimization formula for GGSP is practically the same as for GSP, so with a sufficiently large number of sets, we expect the problem of invalid samples to recur.

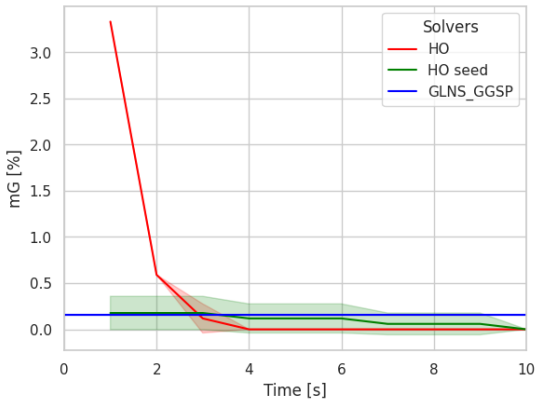

Figure 5.2: Comparison of GGSP solvers on the 19x81 instance

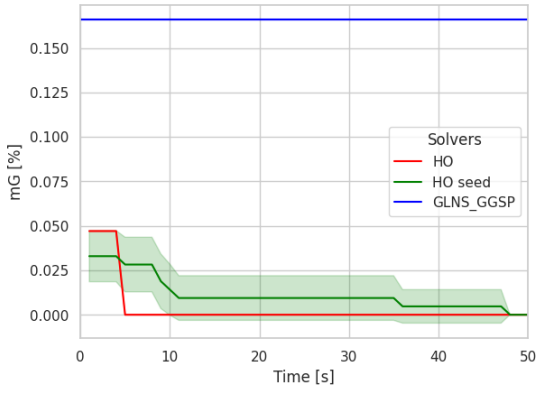

Figure 5.3: Comparison of GGSP solvers on the 24x104 instance

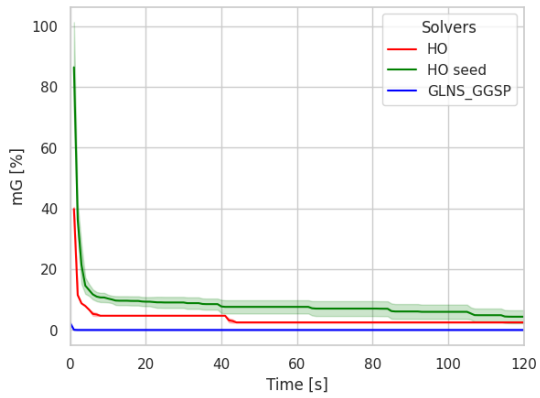

Figure 5.4: Comparison of GGSP solvers on the 54x329 instance

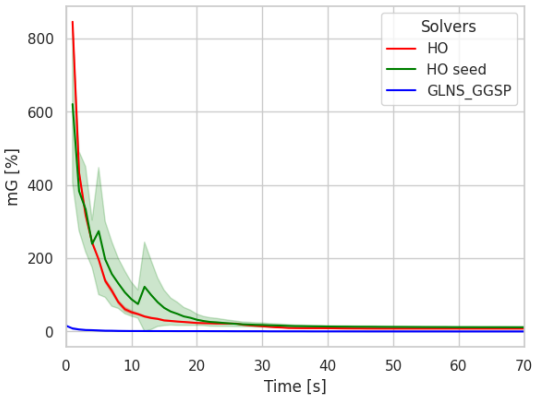

Figure 5.5: Comparison of GGSP solvers on the 135x786 instance

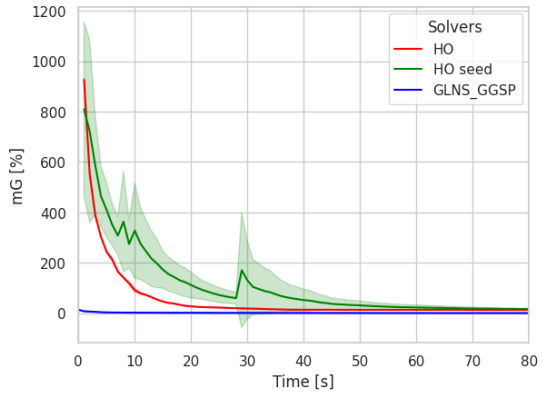

Figure 5.6: Comparison of GGSP solvers on the 143x841 instance

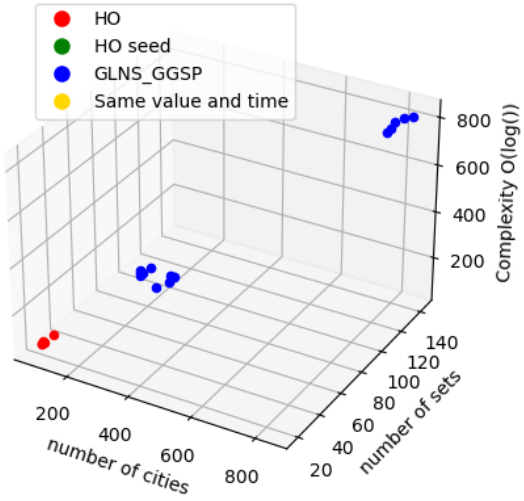

Figure 5.7: 3D scatter plot of the instances

Inst	c_{bf}	HO			HO seed			GLNS_GGSP		
		$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]	$\overline{c_g}$ —	mG_{best} [%]	$\overline{t}$ [s]
19x81	8022.933	8022.933	0.000	3.200	8022.933	0.000	2.800	8035.240	0.153	0.000
19x84	8029.302	8029.302	0.000	1.000	8029.302	0.000	2.000	8039.880	0.132	0.000
20x81	9061.211	9061.211	0.000	1.000	9061.211	0.000	4.000	9072.120	0.120	0.000
20x89	8061.127	8061.127	0.000	1.000	8061.127	0.000	2.200	8076.100	0.186	0.000
24x104	10851.308	10851.308	0.000	5.000	10851.308	0.000	13.100	10869.300	0.166	0.000
59x256	15815.850	16051.428	1.490	70.800	16063.512	1.566	63.000	15861.400	0.288	0.300
60x252	16150.130	16298.810	0.921	84.500	16590.827	2.729	52.300	16170.300	0.125	1.100
61x249	18782.100	19241.898	2.448	70.300	19242.331	2.450	52.600	18782.100	0.000	1.000
60x262	15414.000	16796.730	8.971	50.100	16091.295	4.394	59.800	15414.000	0.000	0.900
63x278	18045.800	19191.292	6.348	68.800	18637.170	3.277	69.000	18045.800	0.000	1.100
54x329	24794.900	25410.334	2.482	42.600	25874.405	4.354	73.300	24794.900	0.000	1.200
57x361	26020.347	26799.862	2.996	108.800	26734.410	2.744	59.200	26047.500	0.104	1.200
60x354	28559.104	29200.949	2.247	28.800	29969.204	4.937	70.900	28568.700	0.034	1.300
60x362	29904.000	31483.898	5.283	28.600	31141.710	4.139	45.600	29904.000	0.000	2.100
60x368	28099.000	28987.619	3.162	96.600	29328.510	4.376	45.500	28099.000	0.000	2.600
135x786	49098.000	52927.410	7.800	68.300	54573.676	11.153	98.200	49138.040	0.082	38.800
137x790	48761.400	55595.547	14.015	114.000	54357.393	11.476	89.400	48776.400	0.031	48.900
140x795	50645.900	54556.797	7.722	118.300	55657.009	9.894	108.900	50814.780	0.333	40.100
142x816	50195.800	55827.285	11.219	90.200	56358.471	12.277	106.800	50447.760	0.502	35.800
143x841	53404.100	60049.154	12.443	108.700	60092.544	12.524	115.700	53519.470	0.216	48.000

Table 5.1: Comparison of GGSP solvers

Conclusion

In this thesis, we experimentally evaluate the performance of the Hexaly Optimizer (HO) solver in solving multi-goal problems, namely TDP, GSP, GTSP, and GGSP. The objective was to compare the quality and speed of the solutions found by HO with the outputs of advanced existing solvers such as Ms-GVNS, GILS-RVND, and GLNS.

The results show that HO can find quality solutions, especially for smaller instances. In the case of GTSP, it achieved better results than the reference solver for both small and large instances, especially when the random seed of the pseudo-random number generator is set appropriately. For GGSP, it succeeded for the smallest instances tested. The modeling of the problems in the Hexaly Modeler was intuitive, and individual programs could be easily adapted to different problems. The HO programs also reliably handled the loading of large instances and always met the specified time limit for finding a solution.

On the other hand, although HO has provided competitive solutions, it has rarely achieved optimal results. In most cases, finding an optimal or near-optimal solution took significantly longer than other solvers. Also, selecting an unsuitable random seed could decrease the quality or speed of the found solution compared to the default setting. For problems with more complex minimization formulas (such as nested sums), like GSP and GGSP, the limitations of HO became more pronounced - the computation slowed down with increasing instance size and, in some cases, failed to find a valid solution within the time limit. These difficulties generally show up sooner for problems with more challenging optimization formulas.

Overall, HO can be described as a flexible optimization tool that offers an alternative to classical heuristic solvers. Its main advantages include intuitive modeling using the Hexaly Modeler language, good scalability, and versatility when applied to optimization problems. Although, in many cases, it does not achieve the best results in terms of solution quality or computation speed, it has proven to be a robust and reliable solver. Given that it is a commercially developing product, further development can be expected, and it may surpass some of the referee solvers in terms of performance in the future. Due to these features, HO can become a valuable tool for solving combinatorial optimization problems.

Further research could focus on using the HO in other supported programming languages, such as C++ or Python, and try to implement it in existing programs. It could also be beneficial to investigate whether it is possible to implement formulas with variable weights into HO and evaluate its effectiveness in solving multi-goal problems.

Bibliography

1. HEXALY. *Hexaly Website* [online]. [N.d.]. [visited on 2025-03-11]. Available from: https://www.hexaly.com.
2. MIKULA, Jan; KULICH, Miroslav. Solving the traveling delivery person problem with limited computational time. *Central European Journal of Operations Research*. 2022, pp. 1–31. ISSN 16139178. Available from DOI: 10.1007/S10100-021-00793-Y.
3. SMITH, Stephen L.; IMESON, Frank. GLNS: An effective large neighborhood search heuristic for the Generalized Traveling Salesman Problem. *Computers & Operations Research*. 2017, vol. 87, pp. 1–19. ISSN 0305-0548. Available from DOI: 10.1016/J.COR.2017.05.010.
4. SAHNI, Sartaj; GONZALEZ, Teofilo. P-complete approximation problems. J ACM 23(3): 555-565. *J. ACM*. 1976, vol. 23, pp. 555–565. Available from DOI: 10.1145/321958.321975.
5. REINELT, Gerhard. TSPLIB - A Traveling Salesman Problem Library. *ORSA Journal on Computing*. 1991, vol. 3, no. 4, pp. 376–384. Available from DOI: 10.1287/ijoc.3.4.376.
6. SILVA, Marcos M.; SUBRAMANIAN, Anand; VIDAL, Thibaut; OCHI, Luiz Satoru. A simple and effective metaheuristic for the Minimum Latency Problem. *European Journal of Operational Research*. 2012, vol. 221, no. 3, pp. 513–520. Available from DOI: 10.1016/j.ejor.2012.03.044.
7. KOUTSOUPIAS, Elias; PAPADIMITRIOU, Christos; YANNAKAKIS, Mihalis. Searching a fixed graph. In: MEYER, Friedhelm; MONIEN, Burkhard (eds.). *Automata, Languages and Programming*. Springer Berlin Heidelberg, 1996, vol. 1099, pp. 280–289. Lecture Notes in Computer Science.
8. FAIGL, Jan. *Data Collection Planning – Multi-Goal Planning* [Lecture 07, B4M36UIR – Artificial Intelligence in Robotics, Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague]. 2024. [visited on 2025-04-05]. Available from: https://cw.fel.cvut.cz/b241/_media/courses/uir/lectures/b4m36uir-lec07-slides.pdf. University lecture.
9. ZVEROVITCH, Alexei. *GTSPLIB* [https://www.cs.rhul.ac.uk/home/zvero/GTSPLIB/]. [N.d.]. [visited on 2025-04-05].
10. *GLNS Solver for the Generalized Traveling Salesman Problem* [online]. University of Waterloo, 2024. [visited on 2025-04-05]. Available from: https://ece.uwaterloo.ca/~sl2smith/GLNS/.
11. MIKULA, Jan. *GLNS for GTSP* [online]. 2025. [visited on 2025-02-20]. Available from: https://gitlab.ciirc.cvut.cz/mission-planning/glns-for-gtsp.
12. REINELT, Gerhard. *TSPLIB95* [online]. [visited on 2025-03-30]. Tech. rep. Institut für Informatik, Universität Heidelberg. Available from: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf.

13. KUBIŠTA, Daniel. *The Generalized Travelling Deliveryman Problem*. 2025. Master's thesis in progress, Department of Cybernetics, Czech Technical University in Prague.

Appendix

matrix_modul.hxm - Hexaly module for creating distance matrices for all types of TSPLIB instances.

GGSP_instances.zip - GGSP instances for Hexaly program