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

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 e ectiveness 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 e ectiveness 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 optimaliza£ního °e²i£e Hexaly Optimizer vyvíjeného spole£ností Hexaly. Jeho výkonnost byla prov¥°ena na vybraných problémech s více cíli, jako je problém doru£ovatele, prohledávání grafu, zobecn¥ný problém obchodního cestujícího a zobecn¥ný problém prohledávání grafu. Výsledky byly porovnány s výstupy pokro£ilých existujících °e²i£·, jako jsou Ms-GVNS, GILS-RVND a GLNS. P°esnost °e²ení byla zaznamenána do tabulek a graf·, které slouºí k p°ehlednému porovnání efektivity Hexaly Optimizeru a ostatních °e²i£·.

Klí£ová slova: Hexaly Optimizer, problém doru£ovatele, problém prohledávání grafu, zobecn¥ný problém obchodního cestujícího, zobecn¥ný 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 de nition 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 de ned. 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 o ers operators like minimize, maximize, and constraint to de ne 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 ve prede ned 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 les.
- 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 les during the resolution.
- output : for writing results in console or in some les, once the resolution is nished.

Both modes have their built-in variables and functions to help de ne 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 speci cations:

- Intel® Core— i7-8750H CPU (2:20 GHz)
- Kingston 256 GB M.2 SATA SSD
- 16 GB of RAM
- Ubuntu 24.04 LTS

This con guration exceeds the minimum requirements needed for the awless 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 - de nes the time limit for nding the optimal solution
- solFileName - speci es the path to the le 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 speci c 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 prede ned 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 $V = \{v_1; \dots; v_N\}$ 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, $d(E) \in R_0^+$. Let us have a Hamiltonian path in G given by a sequence of vertices $x = (x_0; \dots; x_n)$, 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 $^{x}_{k}$ via x is cumulative and is given by:

$$^{x}_{k} = \sum_{i=1}^{k} d(x_{i-1}; x_i) \tag{2.1}$$

The cost of the whole path through x is then given as:

$$\mathrm{cost}(x) = \sum_{k=1}^{n} {}^{x}_{k} = \sum_{k=1}^{n} \sum_{i=1}^{k} d(x_{i-1}; x_i) \tag{2.2}$$

The solution to TDP is then to find the optimal path x that minimizes the cost:

$$x = \arg\min_{x \in H(\ )} (\mathrm{cost}(x)) \tag{2.3}$$

H is the set of all Hamiltonian paths across $= (G; d; s)$, 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:

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

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 prede ned variable hxSeed The default value of the hxSeed is 0. The solution search process can be modi ed by setting hxSeed to a di erent 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 prede ned integer variable hxTimeBetweenDisplays speci es 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 prede ned 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 de ned 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 eld. 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 le 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 su cient 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):

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

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: $Mean \pm 2\ SE$, 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 $\overline{c_g}$	HO mG_{best} [%]	HO $\bar{t}$ [s]	HO seed $\overline{c_g}$	HO seed mG_{best} [%]	HO seed $\bar{t}$ [s]	MS-GVNS $\overline{c_g}$	MS-GVNS mG_{best} [%]	MS-GVNS $\bar{t}$ [s]	GILS-RVND $\overline{c_g}$	GILS-RVND mG_{best} [%]	GILS-RVND $\bar{t}$ [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 $\overline{c_g}$	HO mG_{best} [%]	HO $\bar{t}$ [s]	HO seed $\overline{c_g}$	HO seed mG_{best} [%]	HO seed $\bar{t}$ [s]	MS-GVNS $\overline{c_g}$	MS-GVNS mG_{best} [%]	MS-GVNS $\bar{t}$ [s]	GILS-RVND $\overline{c_g}$	GILS-RVND mG_{best} [%]	GILS-RVND $\bar{t}$ [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, $d(E) \in R_0^+$. Let us have a Hamiltonian path in the graph G given by a sequence of vertices $x = (x_0, \ldots, x_n)$, 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 ${}^{x}_{k}$ via x is cumulative and is given by formula (2.1). The cost of the path through x is then given as:

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

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 $1 \ldots nbCities$, 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 di erence 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 le and setting the prede ned variable hxSeedlike the TDP program described in Subsection 2.2.1. The output text le 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 modi cations described in subsection 2.2.2 are made. These modi cations are used subsequently to solve GSP. Thus, the output text le 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 nd a valid solution in time before sampling, so the program saves the value 1 to the output le 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 di erent 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 nds a di erent 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 $\mathrm{cost}(x) = \sum_{i=1}^{n} w_i \cdot (N - i) \cdot d(x_{i-1}, x_i)$. 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