TU Wien
Institute for Logic and Computation
Algorithms and Complexity Group

Heuristic Optimization Techniques, WS 2019

Programming Exercise: General Information

v1, 2019-10-23

The programming exercise consists of two assignments. This document contains information valid for both assignments. Please read it carefully and contact us by e-mail to `heuopt@ac.tuwien.ac.at`, if you have any questions.

1 Problem description

We consider the *Traveling Salesperson Problem with Target Distances for Multiple Drivers* (TSPTDMD), an adaption of the classical TSP. We are given a non-negatively integer weighted, undirected graph $G = (V, E, d)$, drivers $C = \{c_1, \ldots, c_k\}$, and a desired driver's travel distance $L \in \mathbb{N}_0$, where V are the graph's vertices, E its edges, and $d\colon E \to \mathbb{N}_0$ the edge weight function. The goal is to find a Hamiltonian cycle $H = (e_1, \ldots, e_n)$ and a dependent assignment of the edges of the cycle to the drivers $\varphi\colon E(H) \to C$, so that the root mean squared driver's deviation from the fixed desired travel distance is minimized:

$$\min_{H,f} \mathrm{RMSE}(H, f) = \sqrt{\frac{1}{k}\sum_{i=1}^{k}\left(L - \sum_{e\in\varphi^{-1}(c_i)} d_e\right)^2} \tag{1}$$

Remark: In your implementation, use integers by directly optimizing on the squared error, **not** floats, to avoid numerical issues, especially during delta-evaluation. Convert to the real objective function only for the output.

2 Handling of infeasible solutions

Since the graph is not necessarily complete, it will be difficult for some metaheuristics to always find a feasible solution. To handle these difficulties a common method is penalization. Infeasible solutions are considered feasible during the execution of the algorithm but have an objective value greater than any feasible solution. To achieve this in our problem, we can make our graph complete by introducing edges with large constant weights. Such kind of constant is called big-M. In our case this means that non-existing edges can be used in the solutions, but have weights $M \in \mathbb{N}$.

3 Instances & solution format

As usual, $n := |V|, m := |E|$. The instances are plain ASCII files with filename `<basename>.txt` and either provide the graph as an edge list or as points in the Euclidean plane.

3.1 Edge list format

The format for the edge list looks like:

```
EDGELIST
<n> <m> <k> <L>
<vertex_1_of_edge_1> <vertex_2_of_edge_1> <weight_of_edge_1>
...
<vertex_1_of_edge_m> <vertex_2_of_edge_m> <weight_of_edge_m>
```

A simple example file looks like:

```
EDGELIST
5  8  2  4
0  1  2
0  2  1
0  3  3
1  2  2
1  3  1
2  3  1
2  4  1
3  4  2
```

3.2 Coordinate format

The format using 2D coordinates looks like the following, implying a complete graph:

```
COORDS
<n> <k> <L>
<vertex_1_coord_x> <vertex_1_coord_y>
...
<vertex_n_coord_x> <vertex_n_coord_y>
```

A simple example file looks like:

```
COORDS
7  3  5
0  0
0  5
-2  2
7  4
1  3
20  30
21 -7
```

The weights are given by the Euclidean distance between two vertices rounded to the nearest integer, when halfway rounded to the larger integer.

3.3 Solution format

Vertices are identified by their indices starting with 0: $V = \{0, \ldots, n-1\}$. A solution is described by:

1. Its corresponding instance, the first line is the basename of the instance.
2. A permutation of V given as a single line with whitespace between the vertex indices.
3. Assignments of the edges between the vertices given a single line with whitespace between the driver numbers, where driver numbers also start from zero.

An example is:

```
some_prefix_n5_m8_k2_L4_1
0 3 4 2 1
 0 1 1 0 1
```

Since the graph is not necessarily complete, only the subset of permutations corresponds to valid solutions where there is an edge between every subsequent vertex and from the last vertex to the first.

4 Reports

For each programming exercise we require you to hand in a short report (about 4-6 pages) via TUWEL containing (if not otherwise specified) at least:

- A description of the implemented algorithms (the adaptions, selected options, etc.—not the general procedure)
- Experimental setup (machine, tested algorithm configurations)
- Best objective values and runtimes (+ mean/std. dev. for randomized algorithms over multiple runs) for all published instances and algorithms. Infeasible solutions must be excluded from this calculations. Furthermore list the number of runs which returned feasible/infeasible solutions for all published instances and algorithms.
- Do not use excessive runtimes for your algorithms, limit the maximum CPU time (not wall clock time!) to 15 minutes per instance.

What we do not want:

- Multithreading – use only single-threading
- UML diagrams (or any other form of source code description)
- Repetition of the problem description

5 Solution & source code submission

We require you to hand in your best solutions for each instance and each algorithm **and** an archive of your source code in TUWEL before the deadline. Make sure that the reported best solutions and the uploaded solutions match. The upload can be repeated multiple times—only the best solutions are shown. If one of your algorithms yields infeasible solutions only, still upload the one with the least missing edges.

The uploaded solutions are then checked for correctness and entered in the ranking table. The ranking tables shows information about group rankings (best three groups per instance & algorithm) and solution values to give you an estimate of your algorithms performance. Your ranking does not influence your grade.

6 Development environment & AC cluster

You are free to use any programming language and development environment you like.

It is also possible to use the AC computing cluster. Login using ssh on `USERNAME@eowyn.ac.tuwien.ac.at` or `USERNAME@behemoth.ac.tuwien.ac.at`. Both machines run `Ubuntu 18.04 LTS` and provide you with a `gcc 7.4.0` toolchain, `Java 1.8`, and `Python3.6/3.7`. External libraries required by your program should be installed in your home directory and the environment variables should be set accordingly.

Do not run heavy compute jobs directly on behemoth or eowyn but instead submit jobs to the cluster.

Before submitting a command to the computing cluster create an executable e.g. a bash script setting up your environment and invoking your program. It is possible to supply additional command line arguments to your program. To submit a command to the cluster use:

```
qsub -l h_rt=00:15:00 [QSUB_ARGS] COMMAND [CMD_ARGS]
```

The `qsub` command is a command for the Sun Grid Engine and the command above will submit your script with a maximum runtime of 15 minutes (hard) to the correct cluster nodes. Information about your running/pending jobs can be queried via `qstat`. Sometimes you might want to delete (possible) wrongly submitted jobs. This can be easily done by typing `qdel <job_id>`. You can find additional information under:
`https://www.ac.tuwien.ac.at/students/compute-cluster/`