TU Wien
Institute for Logic and Computation
Algorithms and Complexity Group

Heuristic Optimization Techniques, WS 2025

Programming Exercise: General Information

v1, 2025-07-22

The programming exercise consists of two assignments. This document contains information valid for these assignments, please read it carefully. If you have questions, do not hesitate to contact us either after the lectures or via `heuopt@ac.tuwien.ac.at`.

1 The Selective Capacitated Fair Pickup and Delivery Problem

We study the *Selective Capacitated Fair Pickup and Delivery Problem (SCF-PDP)*, where the goal is to design fair and feasible routes for a subset of n customer requests. Each customer needs transportation of a certain amount of goods from a specific pickup location to a corresponding drop-off location.

The problem is modeled on a complete directed graph $G = (V, A)$, where:

- The node set V includes the vehicle depot, as well as all pickup and drop-off locations associated with customer requests.
- The arc set $A = \{(u, v) : u, v \in V, u \neq v\}$ represents the fastest travel routes between each pair of locations.
- Each arc $(u, v) \in A$ is associated with a travel distance $a_{u,v}$. The distance is calculated as the euclidean distance between the locations rounded up to the next integer. With the x and y coordinates of a location u being denoted as x_u and y_u. The distance is thus calculated as:

$$a_{u,v} = \lceil \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2} \rceil$$

There are n customer requests denoted by $CR = \{1, \dots, n\}$. Each request $i \in CR$ is defined by a pickup location $v_i^{\uparrow} \in V$ and a drop-off location $v_i^{\downarrow} \in V$. We assume that each request concerns the transportation of a certain amount of goods $c_i \in \mathbb{N}$.

Rather than serving all requests, the problem focuses on satisfying only a subset of them. The required number of requests to be fulfilled is given by a fixed parameter γ, where $\gamma \leq n$ and a solution is required to fulfill at least γ requests.

Requests are served by a fleet K of n_K identical vehicles starting from the *depot*, each with a maximum capacity of C.

A feasible solution consists of a route R_k for each vehicle $k \in K$, defined as an ordered sequence of pickup and drop-off locations with the i-th stop of a route being $R_{k,i}$. Each route must fulfill the following properties:

- The vehicle capacity must never be exceeded at any point along the route.
- Each served request must be handled in its entirety by a single vehicle.
- At least γ requests must be served across all vehicles.

The objective is to minimize a combination of the total travel time and a fairness measure among the vehicle routes. To simplify notation we write $(u, v) \in R_k$ to describe a pair of consecutive locations of stops in R_k. The total travel duration of vehicle k is described as follows where $a_{depot,R_{k,1}}$ is the inital travel time from the depot to the first stop and $a_{R_{k,|R_k|},depot}$ is the final travel time from the last stop back to the depot:

$$d(R_k) = a_{depot,R_{k,1}} + a_{R_{k,|R_k|},depot} + \sum_{(u,v)\in R_k} a_{u,v}$$

We define the fairness among all routes using the Jain fairness measure, which is defined as

$$J(R) = \frac{\left(\sum_{k\in K} d(R_k)\right)^2}{n \cdot \sum_{k\in K} d(R_k)^2}.$$

The overall objective function to be minimized is

$$\sum_{k\in K} d(R_k) + \rho \cdot (1 - J(R)).$$

where ρ is a weighting parameter that controls the trade-off between total travel time and fairness across vehicle routes.

2 Instances & Solution Format

A problem instance is given as plain ASCII file and contains in the first line, separated by a space, the number of requests n, the number of vehilces n_K, the capacity C, the minimum number of requests to be served γ, and the fairness weight ρ.

The second part of the file separated by the string `#demands` contains the demands of the requests.

The third part of the file separated by the string `#request locations` contains for all locations their x and y coordinates, starting with the depot and followed by each request's pickup and drop-off location.

n n_K C γ ρ
`# demands`
c_1 c_2 ... c_n
`# request locations`
x_{depot} y_{depot}
$x_{v_1^\uparrow}$ $y_{v_1^\uparrow}$
$x_{v_2^\uparrow}$ $y_{v_2^\uparrow}$
...
$x_{v_n^\uparrow}$ $y_{v_n^\uparrow}$
$x_{v_1^\downarrow}$ $y_{v_1^\downarrow}$
$x_{v_2^\downarrow}$ $y_{v_2^\downarrow}$
...
$x_{v_n^\downarrow}$ $y_{v_n^\downarrow}$

To calculate the value $d(R_k)$ of a simple example route $R_k = [1, 11]$ in an instance with $n = 10$ requests the expression would be:

$$d(R_k) = a_{depot,1} + a_{1,11} + a_{11,depot}$$

The first line of the solution file contains the name of the instance file (without path or file extension) the solution belongs to. Following are the stops of each vehicle. Each line represents one vehicle route starting and ending at the depot. We only note the indexes of the request locations.

```
filename
```
$R_{1,1}\ R_{1,2}\ \ldots\ R_{1,|R_1|}$
$R_{2,1}\ R_{2,2}\ \ldots\ R_{2,|R_2|}$
$\ldots$
$R_{n_K,1}\ R_{n_K,2}\ \ldots\ R_{n_K,|R_{n_K}|}$

An example solution of an instance with 10 requests and 2 vehicles could look like this:

```
example_instance
1 11 2 3 13 4 14 12
5 6 7 8 9 10 15 16 17 18 19 20
```

3 Reports

For each programming exercise you are expected to hand in a concise report via TUWEL containing (if not otherwise specified) at least:

- A description of the implemented algorithms (the adaptions, problem-specific aspects, parameters etc., not the general procedure).
- Experimental setup (machine, tested algorithm configurations).
- Best objective values and runtimes plus (mean/std. deviation for randomized algorithms over multiple runs) for each published instance and algorithm. Infeasible solutions must be excluded from these calculations.
- Do not use excessive runtimes for your algorithms, limit the maximum runtime to, e.g., 15 minutes per instance on the machine you use.
- Use the instances of various sizes provided in TUWEL. Use the `training_instances` to tune the parameters of your different algorithms. Report the results on the instances in `test_instances` in your report.

What we do not want:

- Multithreading and multiprocessing, GPU usage – use only single CPU threads.
- Repetition of the problem description.

4 Solution & Source Code Submission

Hand in your best solutions for each instance and each algorithm **and** a zip-archive of your source code in TUWEL before the deadline. Make sure that the reported best solutions and the uploaded solutions match.

We will provide a competition server where you can upload solutions for the competition instances. The uploaded solutions are then checked for correctness and, if okay, entered in a ranking table. The ranking table shows information about group rankings (best three groups per instance & algorithm) and solution values to give you an estimate of your algorithms performance in comparison to your colleagues' algorithms. Your ranking does not influence your grade. However, the finally best three groups will win small prizes!

5 Development Environment & AC Group's Cluster

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

It is also possible to use the AC group's computing cluster:
Login using ssh on `USERNAME@eowyn.ac.tuwien.ac.at` or `USERNAME@behemoth.ac.tuwien.ac.at`. Both machines run `Ubuntu 18.04.6 LTS` and provide you with `Julia 1.11`, a `gcc 7.5.0` toolchain, `Java openJDK 11.`, as well as `R 4.4.3` with `irace 4.2.0` and `Python 3.13`. You may install other programming languages or language versions or software packages on your own in your home directory but take care to stay in your 5GB disk quota limit.

A possible starting point may be the following open source framework maintained by our group, which provides generic implementations of VNS, GRASP, LNS etc. and examples for the TSP, MAXSAT, and graph coloring for the Julia programming language:

- `https://github.com/ac-tuwien/MHLib.jl`

The usage of any other suitable packages, e.g., for handling graphs or visualization purposes, also is allowed.

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

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 done by `qdel <job_id>`. You can find additional information under `https://www.ac.tuwien.ac.at/students/compute-cluster/`.