PySCIPOpt Exercise: Capacitated Facility Location

3rd IMI-ISM-ZIB Modal Workshop, Tokyo, Japan
Sep. 26, 2018

1 Overview

Facility Location is an entire area of different optimization problems. In this exercise, we consider the well-studied, capacitated variant.

Capacitated facility location denotes the task of connecting customers to facilities in an optimal way, minimizing both the opening costs of the involved facilities and the costs to serve the clients.

More mathematically, let I be the set of customers and J be the set of (potential) facilities. In addition, we use

- $d : I \to \mathbb{R}_+$ the *demand* of each customer
- $M : J \to \mathbb{R}_+$ the *capacity* of each facility
- $f : J \to \mathbb{R}_+$ the *opening costs* of a facility
- $c : I \times J \to \mathbb{R}_+$ the *unit costs* of connecting a customer and a facility.

With those prerequisites, a mixed integer formulation of the capacitated facility location is

$$
\begin{aligned}
\min \quad & \sum_{j \in J} f(j) y_j + \sum_{i \in I, j \in J} c(i,j) x_{i,j} \\
s.t. \quad & \sum_{j \in J} x_{i,j} = d(i) && \forall i \in I \\
& \sum_{i \in I} x_{i,j} \leq M(j) y_j && \forall j \in J \\
& y_j \in \{0,1\} && \forall j \in J \\
& x_{i,j} \geq 0 && \forall i \in I, \forall j \in J
\end{aligned}
\tag{1}
$$

2 Model

2.1 Basic Model creation

All of the necessary code for this exercise goes into the single file "flp_exercise.py". The file already contains the function signature

```
def flp(I, J, d, M, f, c):
    """flp -- model for the capacitated facility location problem
    Parameters:
        - I: set of customers
        - J: set of facilities
        - d[i]: demand for customer i
        - M[j]: capacity of facility j
        - f[j]: fixed cost for using a facility in point j
        - c[i,j]: unit cost of servicing demand point i from facility j
    Returns a model, ready to be solved.
    """
```

All the necessary input data necessary input data are passed as arguments to this function, which should finally declare and return the optimization problem to the script. The following steps are necessary to model the capacitated facility location:

1. creating a new model using the function `Model`.
2. creating variables using the `model.addVar()` function. It is good practice to store the variables in dictionaries `y` and `x` indexed by the facilities and the edges i, j, respectively.

   ```
   y["myvar"] = model.addVar(vtype = "B", name = "variable_one")
   ```

   Objective coefficients are passed by setting the objective function later.
3. creating both types of linear constraints, the demand satisfaction and the capacity restrictions, can be done by using the function `model.addCons()`. Constraints are added as linear or nonlinear expressions of the variables.

   ```
   model.addCons(v + 3 * w + 4 * t <= 6, "example_cons")
   ```

   In this example, all `v`, `t`, and `w` should be variables that have been previously added with `addVar()`.

   It is good practice to wrap these expressions into the `quicksum` function of PySCIPOpt. The `quicksum` function would be useless if it did not accept Python list comprehensions.

   ```
   model.addCons(quicksum(10 * t for t in [u,v,w]) <= 15)
   ```
4. Use the function `model.setObjective()` to set the objective function. This function accepts expressions using `quicksum` exactly like `addCons`.
5. `return` the resulting model. The `data` attribute is used for plotting and the heuristic of the next exercise.

2.2 Running the basic model

Run the model by executing `python flp_exercise.py`. You see the log output of SCIP. What is the number of open facilities in the optimal solution? What are its total costs? How many nodes does SCIP need to optimize this problem? If you have the python package `networkx` installed, a graphical display should open and render the solution as a graph.

Hint SCIP provides even more detailed statistics on the solution process. In order to access them after SCIP has solved the model, uncomment the following line in the main block of the script.

```
#model.printStatistics()
```

2.3 Extending the model

The initial LP relaxation can be made stronger. As a matter of fact, one can easily formulate upper bounds for the x variables. Infact, because of the non-negative costs, none of the x variables will exceed the demand of its associated customer. Infact, one can even make this stronger: $x_{i,j} \leq d(i)y_j$ for all $i \in I$ and $j \in J$. Such inequalities are also called variable bound inequalities in SCIP as the upper bound of $x_{i,j}$ varies with the values of the y_j's.

Add these additional inequalities to the model. By how much does the value of the initial root LP relaxation change by the tighter formulation?

3 Heuristic

After the modeling has been finished, it is a natural second step to customize the solution process to the model at hand. SCIP's plugin based system allows for extensions to all main components of the solution process. For the sake of this exercise, we extend the primal heuristics of SCIP by an additional greedy heuristic.

The file "flp_exercise.py" already contains the class `GreedyHeur` with a functional, yet empty class method `heurexec`. The greedy algorithm should be completely implemented within this method. The heuristic has already been added (included) to the model object, such that calling optimize triggers the execution of the greedy heuristic at the specified timing, i.e., every time a search node has been selected, but not processed yet.

```
class GreedyHeur(Heur):

    def heurexec(self, heurtiming, nodeinfeasible):
        """execution callback of the greedy heuristic

        Parameters:
        -self: the heuristic itself, with the FLP model as attribute
        -heurtiming: timing during the search process
        -nodeinfeasible: flag to indicate whether this node is already infeasible.
        """

        if nodeinfeasible:
            return {"result" : SCIP_DIDNOTRUN}

        sol = self.model.createSol(self)
        x, y, M, d, c, I = self.model.data

        if False:
            accepted = self.model.trySol(sol)
        else:
            accepted = False

        if accepted:
            return {"result": SCIP_RESULT.FOUNDSOL}
        else:
            return {"result": SCIP_RESULT.DIDNOTFIND}

if __name__ == "__main__":
    I, J, d, M, f, c = make_data()
    model = flp(I, J, d, M, f, c)
    model.includeHeur(GreedyHeur(),
        "greedyfacility",
        "greedy␣heuristic␣for␣capacitated␣facility␣location",
        "Y",
        timingmask = SCIP_HEURTIMING.BEFORENODE)
```

We suggest to implement a primal heuristic that implements the following

simple greedy procedure.

1. Set $d'(i) = d(i)$ for all $i \in I$, $M'(j) = M(j)$ for all $j \in J$.
2. Select the facility $j^* \in J$ that provides the highest capacity/cost ratio.
3. Set $y_{j^*} = 1$.
4. Choose a customer i^* with some residual demand d' with the cheapest connection cost c_{i^*,j^*}.
5. Set $x_{i^*,j^*} = \min\{d'(i^*), M'(j^*)\}$.
6. Decrease $d'(i*)$ and $M'(j*)$ $by x_{i^*,j^*}$.
7. Repeat 3. and 4. until j^* is saturated, or no customer is left.
8. Remove j^* from the set of facilities.
9. Go to 1.

Task Implement the algorithm, and comment on its success.

Hints

- near the end of the method, the variable `accepted` is currently set to `FALSE`. Drop this `if`-condition to actually test the created solution for feasibility.
- all necessary data has already been unpacked from the model data object. This need not be modified.
- The function for setting solution values is `model.setSolVal(sol, var, value)` The `sol` object has already been created for you
- You may encounter output that reports a violated constraint or variable bound. This violation output is expected for this heuristic, but does not render the solving process infeasible. Can you think of a reason why this output appears?

Extension The heuristic is included with a heuristic frequency of 1, which means that it is run at every node of the search. The algorithm does not use local information, yet. Hence, the greedy procedure is the same at every node. You may consider local bounds for the y_j variables to split the facilities into the subsets of open, closed, and undecided facilities. Therefore, you need to query the local lower and upper bounds of the variables. The respective methods are `var.getLbLocal()` and `var.getUbLocal()`. How do you use this information to modify the algorithm?