Department of Mathematics and Computer Science
University of Southern Denmark, Odense

February 28, 2018
Marco Chiarandini

DM865 - Heuristics and Approximation Algorithms

Obligatory Assignment – Part 1, Spring 2018

Deadline: 18th January 2018 at 12:00.

- This is the *first obligatory assignment* of Heuristics and Approximation Algorithms. It will be graded and it will contribute for 50% of the final grade.
- The assignment has to be carried out individually.
- The submission is electronic via `http://valkyrien.imada.sdu.dk/DOApp/`.
- You have to hand in:
  - the source code of your implementation of a heuristic solver. Submit all your files in a `.tgz` archive. Your code will be compiled and run, hence it must comply to the requirements listed in this document.
  - A report that describes the work you have done and presents the results obtained. The document should not exceed 10 pages and must be in PDF format. You cannot list source code. You can write in Danish or in English.
- Make your submissions anonymous. The submission system will rename your files with the identifier that you will see on the page and your identity will be retrieved after the grading has been done.
- Changes to this document after its first publication on December 19 may occur. They will be emphasized in color and if they are major they will be announced via BlackBoard.
- Read all this document before you start to work.
- A starting package containing the instances and the code to read them is available at this link:

`http://www.imada.sdu.dk/~marco/DM841/Files/BP.tgz`

Introduction

The aim of this final assignment is to design, implement and report local search heuristic algorithms for solving

Make sure you have read the whole document before you start to work.

Heuristics for Capacited Vehicle Routing

In vehicle routing problems we are given a set of *transportation requests* and a fleet of *vehicles* are we seek to determine a set of *vehicle routes* to perform all (or some) transportation requests with the fiven vehicle fleet at *minimum cost*; in particular, decide which *vehicle handles which request in which sequence* so that all *vehicle routes* can be *feasibly* executed.
The *capacited vehicle routing problem* (CVRP) is the most studied version of vehicle routing problems.
In the CVRP, the transportation requests consit of the distribution of goods from a single *depot*, denoted as point 0, to a given set of n other points, typically referred to as *customers*, $N = \{1, 2, \ldots, n\}$. The amount that has to be delivered to customer $i \in N$ is the customer's *demand*, which is given by a scalar $q_i \geq 0$, e.g., the weight of the goods to deliver. The *fleet* $K = \{1, 2, \ldots, |K|\}$ is assumed to be *homogeneous*, meaning that $|K|$ vehicles are available at the deport, all have the same capacity $Q > 0$, and are operating at identical costs. A vehicle that services a customer subset $S \subseteq N$ starts at the depot, moves once to each of the customers in S, and finally returns to the depot. A vehicle moving from i to j incurs the *travel cost* c_{ij}.

The given information can be structured using an undirected graph. Let $V = \{0\} \cup N = \{0, 1, \ldots, n\}$ be the set of *vertices* (or nodes). It is convinient to define $q_0 := 0$ for the depot. In the symmetric case, when the cost for moving between i and j does not depend on the direction, i.e., either from i to j or from j to i, the underlying graph $G = (V, E)$ is complete and undirected with edge set $E = \{e = (i, j) = (j, i) : i, j \in V, i \neq j\}$ and edge costs c_{ij} for $\{i, j\} \in E$. Overall, a CVRP instance is uniquely defined by a complete weighted grah $G = (V, E, c_{ij}, q_i)$ together with the size $|K|$ of the of the vehicle fleet K and the vehicle capacity Q.
A *route* (or *tour*) is a sequence $r = (i_0, i_1, i_2, \ldots, i_s, i_{s+1})$ with $i_0 = i_{s+1} = 0$, in which the set $S = \{i_1, \ldots, i_s\} \subseteq N$ of customers is visited. The route r has cost $c(r) = \sum_{p=0}^{s} c_{i_p, i_{p+1}}$. It is *feasible* if the capacity constraint $q(S) := \sum_{i \in S} q_i \leq Q$ holds and no customer is visited more than once, i.e., $i_j \neq i_k$ for all $1 \leq j \leq k \leq s$. In this case, one says that $S \subseteq N$ is a *feasible cluster*. A solution to a CVRP consists of K feasible routes, one for each vehicle $k \in K$. The routes $r_1, r_2, \ldots, r_{|K|}$ and the corresponding clusters $S_1, S_2, \ldots, S_{|K|}$ provide a *feasible solution* to the CVRP if all routes are feasible and the clusters form a partition of N. Hence, the CVRP consists of two interdependent tasks:

(i) the partitioning of the cusotmer set N into feasible clusters $S_1, \ldots, S_{|K|}$;

(ii) the routing of each vehicle $k \in K$ through $\{0\} \cup S_k$.

The Capacited Vehicle Routing Problem (CVRP) is a vehicle routing problem in which a fleet of identical vehicles of finite capacity and located at a central depot must service a set of customers with known demands. The objective is to determine a set of vehicle trips of minimum total cost (sum of travel distance or time), such that each vehicle starts and ends at the depot, each client is visited exactly once and the total demand handled by any vehicle does not exceed its capacity.

Using the test instances described below, you have submit a report and a source code that address the following tasks:

- Determine an easy-to-calculate lower bound K_{LB} to the number of vehicles needed to satisfy the demand of all customers. Report in a table of your final text document the lower bounds thus found for each given instance. (Another way to obtain a lower bound, which is a bit harder to calculate, is by solving a problem addressed in one of the examples of Gecode and described in the manual [MPG, part C] – you will need this to address point 3 below.)
- Model the problem in constraint programming terms and describe the model in the report. Ignore for the moment the objective of minimizing the total length of the routes and focus on feasibility only.
- Implement the model at point 2. above in Gecode. Let K^* be the minimum number of vehicles needed cover the demand of all customers of an instance. Determine this number for all instances given and compare it with K_{LB}. Report in a table the comparison with the lower bounds found in point 1.
- Modify your model to minimize the total length of the routes while using the number of vehicles given in the name of the instance. Report the results in a table. Indicate whether the instance is solved to optimality or not and report meaningful measures to describe the effort needed to find the solutions. If the solution found is not optimal within a given time limit, report the best solution found so far.
- Improve your model by fine tuning some parameters of your choice and present the analysis with numerical results and discussion. The parameters can be: branching rules, type of consistency, redundant constraints, alternative models, symmetry breaking.
- Study the use of random restarts. Describe the restart strategy and the configuration you chose and find out whether random restarts improve or not the performance of the solver. See sec. 9.4 from [MPG] for instructions on how to do random restart. Start your implementation from the best model found so far, but make sure that a restart is worth by ensuring that at each restart a modified problem is solved. Consider using no-goods, shared heuristics or random choices or different heuristics at each restart.
- Implement a large neighborhood search to improve the quality of the best solutions found in the largest instances. Describe your attempt in the report.

There is no predefined time limit. You can choose to run your experiments with the time limit that best fit with your computation resources (eg, time left from deadline, number of cores and machines, etc). As always the final results count. Performance can be assessed in several ways (max size of the instances solved, computation time, solution quality, etc.) It is up to you to identify meaningful measures and comment on performance.
All instances are *minimization* problems. The ones that will be used to test your solvers are listed in Table **??**.

Practicalities

Associated to this document there is a starter package `vrp.tgz` containing the test instances and some initial code to read the instances, output a solution and produce a graphical view of the instances and solutions in SVG format (see example in Figure **??**).

Instances In the directory `data/augerat/P` there is one main set of instances of varying size, from 16 to 101 customers. This is the set P from [**?**]. In addition, you are given a small instance with 10 costumers `toy.xml`. The toy instance and a solution for it is visualized in Figure **??**. All instances are in the same XML format. The format allows you to inspect the instance if you need it. The names of the instances use the following scheme: `P-n[NNN]-k[KK].xml`, where `NNN` is the number of costumers and `KK` is the number of vehicles to use.

The program Enter in the directory `src/` and edit the makefile to make sure that the path to the Gecode library is correct. Compile the program via `make` and execute

```
$ xyz ../data/toy.xml
```

This will read the instance and use a stored solution for it. It will print in the standard output some relevant data of the instance and produce a file `routes.svg` which can be used to visualize the instance and the solution with any browser. For example in Linux:

```
$ firefox routes.svg
```

In OSX you can use the command `open`.
If parameters have to be passed by command line, they must be set before the name of the instance file. For example:

```
$ xyz -time 60000 ../data/toy.xml
```

The source files `tinyxml2` contain the library to read XML files. You do not need to edit these files. The files `Data` contain the class `Input` and `Output`. The class `Input` contains the relevant data of the instance, precisely:

- `int decimals` this parameter defines how to approximate the calculation of the length of the arcs. You should not need it;
- `const char* name;` the name of the instance;
- `bool symmetric;` whether the arcs are symmetric on not. For the given instances the graphs are undirected and complete. In the general case the graph may also be directed and incomplete;
- `vector<_vehicle> vehicles;` The set of vehicles with a type, a departure node, an arrival node and the capacity. You will not need to use the type in this assignment;
- `vector<int> depots;` a list of nodes that represent depots. In the given instances there is only one depot. Note that this is not necessarily the node 0.
- `map<int, _node> nodes;` The set of nodes. The key of the map is the identifier of the node. The structure `_node` contains the id, the type, the demand, and the coordinates cx and cy. The type defines whether the node is a depot (0) or not ($\neq 0$);
- `unordered_map<arcKey, _arc, arcKeyHash, arcKeyEqual> arcs;` The set of arcs. The map has as key a pair of numbers (tail,head) representing the nodes of the arc. The structure `_arc` defines the tail (starting node), the head (ending node) and the length;

The class `Output` contains `map<int, list<int> > routes`, that is, a set of lists of costumers representing the routes. Each of these lists must start and end with the depot. The class also contains the function to plot the instance and the overwriting of the output operator to write the solution in the format required in the standard output.
You will need to modify the function `setSolution(Output & o)` to process the solution found by your model and write it to `Otuput::routes`. The function is then called by in the `main()` function:

```
out.getSolution();
cout << out << endl;
out.draw();
```