Guided project - Step 1: Implementing a randomized nearest neighbor heuristic for the TSP

1 Getting started

1. Go to CELENE and download the `tp1.zip` file
2. Create a Java project in your favorite IDE (I will use Eclipse but feel free to use your favorite tool)
3. Import the code in the `tp1.zip` into your project
4. Explore package `dii.vrp.data` (you may find the class diagram in Fig. 1 handy)
5. Explore package `dii.vrp.tp` (you may find the class diagram in Fig. 2 handy)

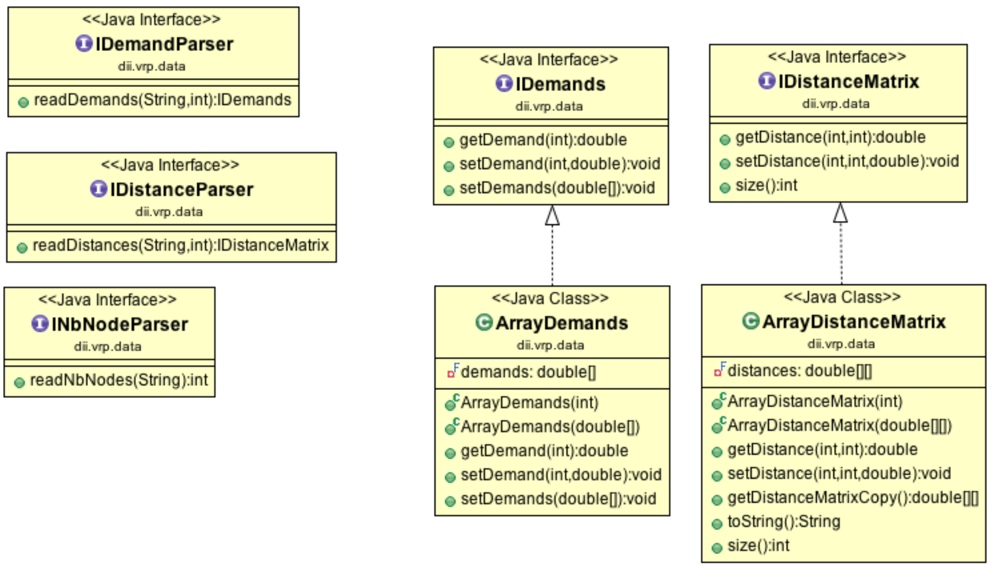

Figure 1: Class Diagram for package `dii.vrp.data`

2 Modeling a TSP solution

Implement a class called `dii.vrp.tp.TSPSolution` that implements the `dii.vrp.tp.IRoute` and `dii.vrp.tp.ISolution` interfaces to model a solution to the TSP (hint: you can use the adapter design pattern to wrap an instance of `dii.vrp.tp.ArrayRoute` and used it to support `IRoute` behavior).

3 Implementing the Nearest Neighbor heuristic

3.1 A first approach

Implement a class called `dii.vrp.tp.TSPSolution` that implements the `dii.vrp.tp.ITSPHeuristic` interface. Provide your class with a `public NaiveNNHeuristic(final IDistanceMatrix distances)`

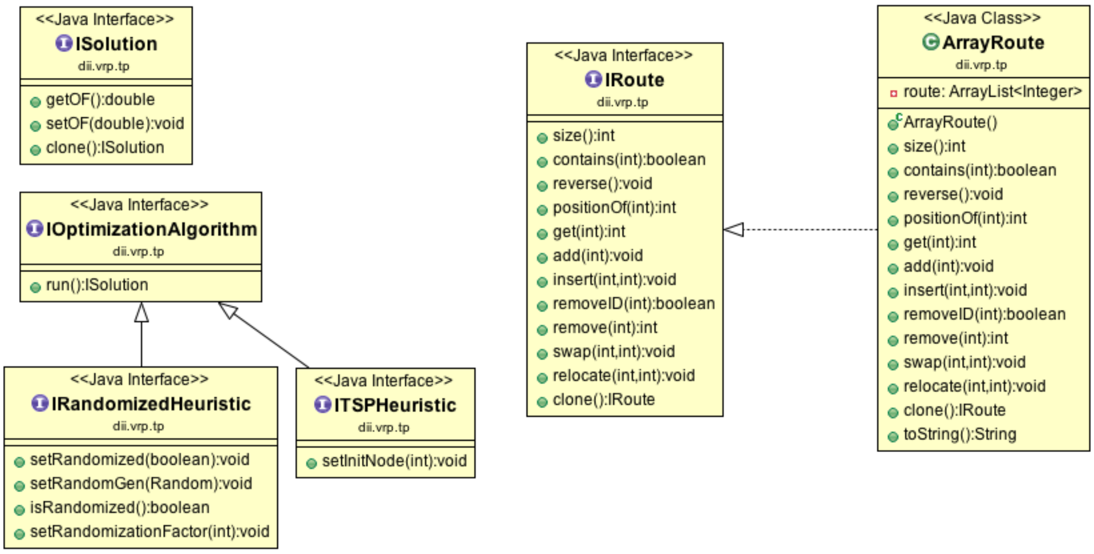

Figure 2: Class Diagram for package `dii.vrp.tp`

method where argument `distances` holds the distance matrix of the instance to solve. The `dii.vrp.tp.ITSPHeuristic#run()` method should run the heuristic and return an instance of `dii.vrp.tp.TSPSolution`. For this first approach try the simplest algorithm that comes to your mind.

Implement a class called `dii.vrp.tp.test.TNNHeuristic` with a `public static main()` method to test your implementation. Use the data for the UY734 instance hard coded in class `dii.vrp.tp.data.UY734`. You may find the services provided by class `dii.vrp.utils.EuclideanCalculator` useful to compute the distance matrix.

3.2 A more refined implementation

Our Nearest Neighbor implementation has a major drawback: it is not computationally efficient. We can improve the efficiency of our heuristic by implementing a speed up technique known as *pre-computed neighbors*. The technique consists in pre-computing for each node the list of its neighbors sorted in ascending order of proximity. When the heuristic needs to find the nearest not-routed neighbor of a node, it searches the list instead of re-computing the distance between the node and all other nodes. In theory this trick does not chance the temporal complexity of the algorithm (why?), but in practice it generates significant speed ups when the heuristic is used repetitively.

1. Complete the implementation of class `dii.vrp.tp.NNFinder`. This class will be responsible for storing the list of neighbors for each node and retrieving the nearest not-routed neighbor of a node whenever the heuristic needs to find it
2. Create a copy of class `dii.vrp.tp.NNHeuristic` and rename it `dii.vrp.tp.NaiveNNHeuristic`. We'll need our naive implementation to conduct some experiments later
3. Refactor class `dii.vrp.tp.NNHeuristic` so it implements the pre-computed neighbors speed up technique
4. Test your implementation using the `dii.vrp.test.TNNHeuristic` class we developed for exercise 3.1
5. Download from CELENE class `dii.vrp.tp.NNvsNaive`. Study the class to understand the experiment. Run the experiment for different values of `T`. What do you observe?

3.3 Randomizing our heuristic

The ultimate goal of our Nearest Neighbor implementation is to embed it into a more complex algorithm for the VRP (we will talk about this in class tomorrow). For that, we need a randomized version of our heuristic.

1. Refactor class `dii.vrp.tp.NNHeuristic` so it implements the `dii.vrp.tp.IRandomizeHeuristic`. Revise your implementation of the `dii.vrp.tp.NNHeuristic#run()` so the heuristic connects at each iteration a random node selected between the first `K` nearest not-routed neighbors of the last node added to the tour (see the slides for chapter 2 for an example).

Routage et modélisation du trafique
Jorge E. Mendoza
Polytech Tours

Guided project - Step 2: Implementing a sequence-first, route-second heuristic

1 Getting started

1. Go to CELENE and download the `step2.zip` file
2. Create a Java project in your favorite IDE (I will use Eclipse but feel free to use your favorite tool)
3. Import the code in the `step2.zip` into your project

2 Implementing a VRP solution

Before implementing our heuristic we are going to need a class modeling a VRP Solution. Complete the implementation of class `dii.vrp.solver.VRPSolution` (HINT: study class `dii.vrp.solver.VRPRoute`).

3 Implementing the split algorithm

As we saw in class, the split algorithm optimally splits a TSP tour into a VRP solution. In this first exercise, we will do a from-the-book implementation of the algorithm based on Prins (2004).

Figure 3 displays the exact same algorithm presented in the Prins paper. Study the algorithm and try to understand it.

Create a class called `dii.vrp.solver.Split` that implements interface `dii.vrp.solver.ISplit` (you can find a template of the class in the project). We are going to code together the algorithm.

Now that we have implicitly built the auxiliary graph and computed the shortest path, we need to extract from the `T` and `P` labels the routes that make up the resulting solution. We also need to *evaluate* the solution (i.e., compute the cost and load for each route, and the total cost of the solution). Implement method `dii.vrp.solver.Split#extractRoutes()` (you can change the method declaration if you need to).

To test our implementation we will use an example from the Prins paper (see Fig. 3). Class `dii.vrp.data.PrinsExample` hard codes the data of the example. Use class `dii.vrp.test.TSplit` to test your implementation. Do you obtain the exact same solution presented in the figure?

4 Implementing a sequence-first, route-second heuristic

We now have all the ingredients to "cook" a sequence-first, route-second heuristic, namely, a TSP heuristic and a split procedure. Implement a class called `dii.vrp.solver.SFRSHeuristic` that implements interface `dii.vrp.solver.IOptimizationAlgorithm`. Implement a constructor `public SFRSHeuristic(final ITSPHeuristic h, final ISplit s)` so client classes can pass a reference to the objects responsible for building TSP solutions and splitting those solutions into VRPSolutions.

```
$V_0$ := 0
for $i$ := 1 to $n$ do $V_i$ := $+\infty$ endfor
for $i$ := 1 to $n$ do
   $load$ := 0; $cost$ := 0; $j$ := $i$
   repeat
      $load$ := $load$ + $q_{S_j}$
      if $i = j$ then
         $cost$ := $c_{0,S_j} + d_{S_j} + c_{S_j,0}$
      else
         $cost$ := $cost - c_{S_{j-1},0} + c_{S_{j-1},S_j} + d_{S_j} + c_{S_j,0}$
      endif
      if ($load \leq W$) and ($cost \leq L$) then
         //here substring $S_i \dots S_j$ corresponds to arc $(i-1, j)$ in $H$
         if $V_{i-1} + cost < V_j$ then
            $V_j := V_{i-1} + cost$
            $P_j := i - 1$
         endif
         $j$ := $j + 1$
      endif
   until ($j > n$) or ($load > W$) or ($cost > L$)
enfor.
```

Figure 1: The split algorithm as presented in Prins (2004)

5 Testing our heuristic

We are now going to test our heuristic on standard instances from the literature. Go to `www.vrp-rep.org` and download the CMT dataset. Decompress the `.zip` file and put the file in the `./data/CMT` folder of your Java project. The instance files are formatted using the VRP-REP instance specification (see `http://www.vrp-rep.org/faq.html#œspecification`). Class `dii.vrp.data.VRPREPInstanceReader` implements a parser for VRP-REP-formatted files. Take a quick look at the parser.

Implement a class called `dii.vrp.test.TSFRSHeuristic` with a `main()` method to test our implementation. Use any of the 14 instances in the CMT set to test our code (e.g., CMT01.xml). You can find the best known solutions for those instances at `http://www.vrp-rep.org/solutions.html`. How does our algorithm compare to the state-of-the-art approaches?[1].

6 Implementing a randomized sequence-first, route-second heuristic

Based on our `dii.vrp.test.TSFRSHeuristic` write a class that i) randomizes our sequence-first, route-second heuristic (HINT you can do that in three lines of code after line 44) and ii) runs the randomized heuristic T times and reports only the best solution found. Try with different values of T and randomization factors. Can you get closer to the best known solution?

[1]some of the instances have a distance constraint. Since we did not implement this constraint our results on those instances are not comparable with results from the literature

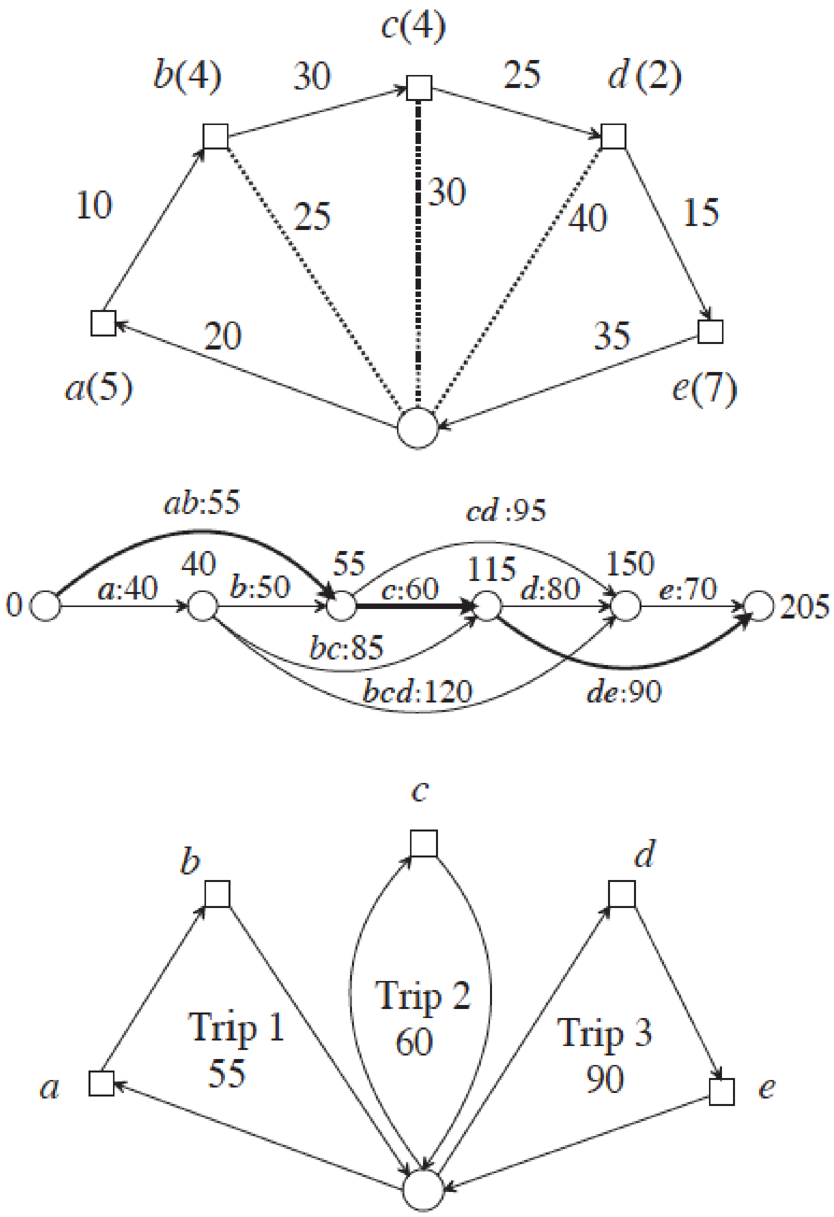

Figure 2: Example from Prins (2004)

Guided project - Step 3: Implementing a simple GRASP for the VRP

1 Getting started

1. Go to CELENE and download the `step3.zip` file
2. Create a Java project in your favorite IDE (I will use Eclipse but feel free to use your favorite tool)
3. Import the code in the `step3.zip` into your project

2 Introduction

As we study in class, the Greedy Randomized Adaptive Search Procedure (or GRASP) is a metaheuristic framework that combines a randomized heuristic and a local search procedure. We are going to use the blocks we built in the last two sessions to put together a simple GRASP for the VRP. We now have:

1. A randomized nearest neighbor heuristic
2. A split produce for the VRP
3. A randomized sequence-first, cluster-second heuristic (= 1 + 2)

Building block 3 can play the role of the randomized heuristic in our GRASP. We now need to develop a local search procedure. Our local search procedure will be a variable neighborhood descent (VND). Because of time constraints we will only implement one neighborhood (relocate), but our implementation will be flexible enough to include as many neighborhoods as we want.

3 Implementing the relocate neighborhood

The relocate move extracts a node from its current location and tries to insert it in every possible position in the solution. Before we start implementing our neighborhood, we first need to do some code reading. Study interface `dii.vrp.solver.INeighborhood` and enumeration `dii.vrp.solver.ExplorationStrategy`. Now create a class called `dii.vrp.solver.Relocate` that implements interface `dii.vrp.solver.INeighborhood`. Give your class a constructor `public Relocate(IDistanceMatrix distances, IDemands demands, double Q)`.

Implement an *internal private class* called `dii.vrp.solver.Relocate#Relocation` modeling a relocate move (since the class is private you do not need to implement accessors). What information about a move shall we store in a `Relocation` object?

Now we are going to implement the `Relocate#explore(ISolution)` method together.

To test our implementation create a class called `dii.vrp.test.TRelocate` with a `main()` method that explores the Relocate neighborhood starting from an *almost-optimal-solution* for instance CMT01 (use method `dii.vrp.data.CMT01#getS1()` to build the solution).

4 Implementing a VND

Now that we have a neighborhood for our problem, we need to implement a local search procedure. Create a class called `dii.vrp.solver.VND` that implements interface `dii.vrp.solver.ILocalSearch`. Add a constructor `public VND(final INeighborhood neighborhood)` to your class. This constructor ensures that your VND has, at least, one neighborhood.

To test your implementation create a class called `dii.vrp.test.TVND` similar to `dii.vrp.test.TRelocate` that runs your VND on the almost-optimal-solution for instance CMT01 retrieved by method `dii.vrp.data.CMT01#getS2()`.

5 Implementing a GRASP

Now that we have a randomized heuristic and a local search procedure, we are ready to implement our GRASP. Create a class called `dii.vrp.solver.GRASP` that implements interface `dii.vrp.solver.ILocalSearch`. Add a constructor `GRASP(IRandomizedHeuristic initSolGenerator, ILocalSearch localSearch, int iterations)` to your class.

To test your GRASP implement a class called `dii.vrp.test.TGRASP` that runs your GRASP using an instance of your `dii.vrp.solver.SFRSHeuristic` as a randomized heuristic (do not forget to set the random mode on and set a randomization factor greater than 1) and your `dii.vrp.solver.VND` as a local search procedure.

How do your results compare to those obtained by the `dii.vrp.solver.SFRSHeuristic` in the experiment ran with class `dii.vrp.test.TSFRSHeuristic`? Does the local search contribute to the accuracy of the method?.