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

1 Getting started

1. Go to CELENE and download the `step1.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 `step1.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.solver` (you may find the class diagram in Fig. 2 handy)

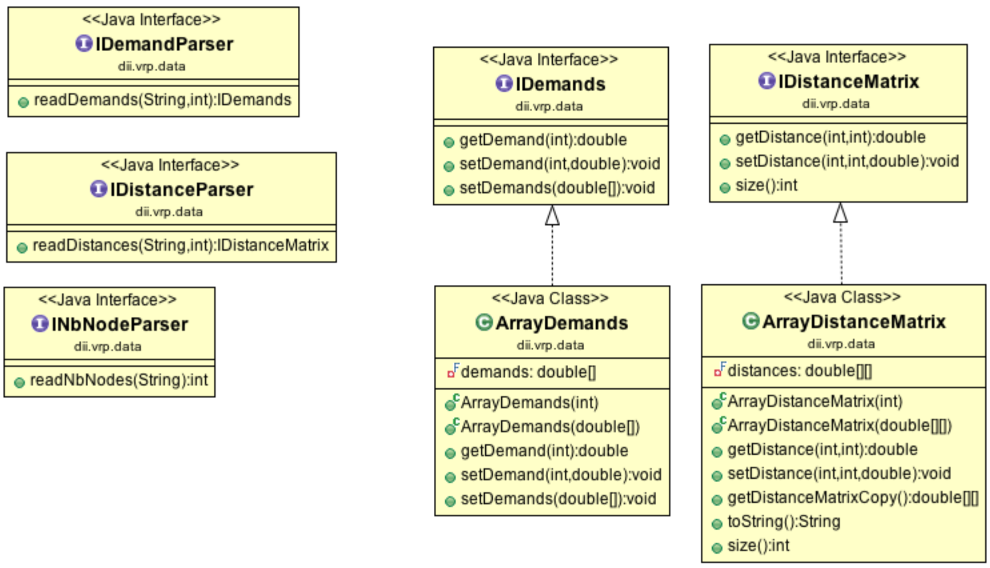

Figure 1: Class Diagram for package `dii.vrp.data`

2 Modeling a TSP solution

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

Listing 1: Solution — TSPSolution.java

```
package dii.vrp.solver;
/**
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 */
public class TSPSolution implements IRoute, ISolution{
```

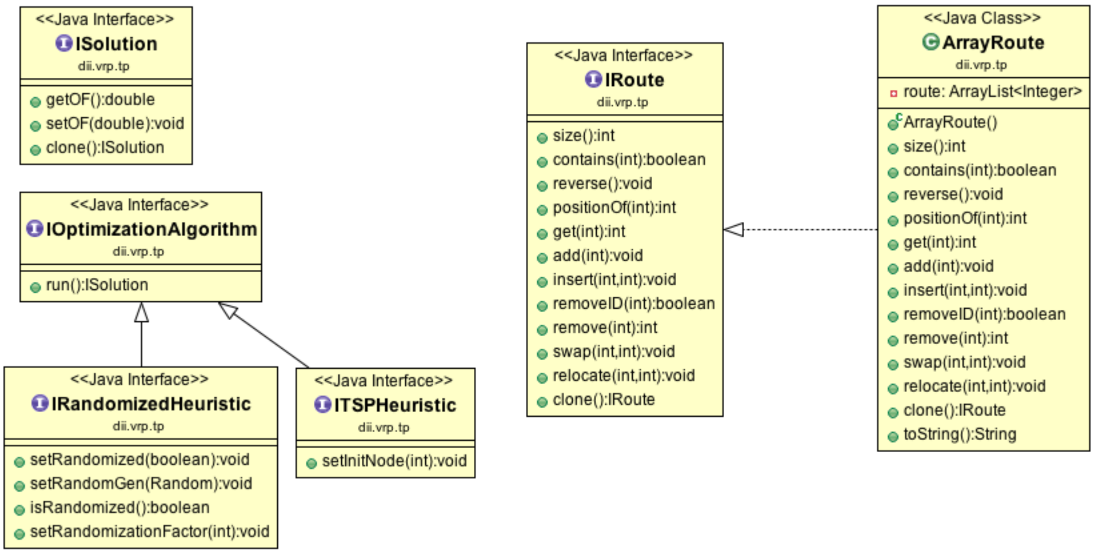

Figure 2: Class Diagram for package `dii.vrp.solver`

```java
/**
 * Stores the route and supports the IRoute interface
 */
private ArrayRoute route;
/**
 * Stores the objective function
 */
private double of=Double.NaN;

public TSPSolution(){
        route=new ArrayRoute();
}

@Override
public double getOF() {
        return this.of;
}

@Override
public void setOF(double of) {
        this.of=of;
}

@Override
public TSPSolution clone() {
        TSPSolution clone=new TSPSolution();
        clone.route=(ArrayRoute)this.route.clone();
        clone.of=this.of;
        return clone;
}

@Override
public int size() {
        return route.size();
}

@Override
public boolean contains(int nodeID) {
        return route.contains(nodeID);
}
```

```
        @Override
        public void reverse() {
                route.reverse();
        }

        @Override
        public int positionOf(int nodeID) {
                return route.positionOf(nodeID);
        }

        @Override
        public int get(int i) {
                return route.get(i);
        }

        @Override
        public void add(int nodeID) {
                route.add(nodeID);
        }

        @Override
        public void insert(int nodeID, int i) {
                route.insert(nodeID, i);
        }

        @Override
        public boolean removeID(int nodeID) {
                return route.removeID(nodeID);
        }

        @Override
        public int remove(int i) {
                return route.remove(i);
        }

        @Override
        public void swap(int i, int j) {
                route.swap(i, j);
        }

        @Override
        public void relocate(int i, int j) {
                route.relocate(i, j);
        }

        @Override
        public String toString(){
                StringBuilder sb=new StringBuilder();
                sb.append(this.of+"|"+this.route.toString());
                return sb.toString();
        }

}
```

3 Implementing the Nearest Neighbor heuristic

3.1 A first approach

Implement a class called `dii.vrp.solver.NNHeuristic` that implements the `dii.vrp.solver. ITSPHeuristic` interface. Provide your class with a `public NaiveNNHeuristic(final IDistanceMatrix distances)` method where argument `distances` holds the distance matrix of the instance to solve. The `dii.vrp.solver.ITSPHeur` method should run the heuristic and return an instance of `dii.vrp.solver.TSPSolution`. For this first approach try the simplest algorithm that comes to your mind.

Implement a class called `dii.vrp.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.data.UY734`. You may find the services provided by class `dii.vrp.utils.EuclideanCalculator` useful to compute the distance matrix.

Listing 2: Solution — NNHeuristic.java

```java
package dii.vrp.solver;

import dii.vrp.data.IDistanceMatrix;

/**
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 */
public class NaiveNNHeuristic implements ITSPHeuristic {
        /**
         * Stores the distance matrix
         */
        private IDistanceMatrix distances;
        /**
         * Stores the departure node. Default value = 0.
         */
        private int initNode=0;

        public NaiveNNHeuristic(final IDistanceMatrix distances){
                this.distances=distances;
        }

        @Override
        public ISolution run() {
                //Compute the number of nodes to route
                int toRoute=distances.size();

                //Initialize node status
                boolean[] nodeStatus = new boolean[toRoute];

                //Initialize the tour
                TSPSolution tour=new TSPSolution();
                tour.add(this.initNode);
                nodeStatus[this.initNode]=true;
                toRoute--;

                //Complete the tour
                while(toRoute>0){
                        //Find nearest neighbor
                        double minDist=Double.MAX_VALUE;
                        int i=tour.get(tour.size()-1);
                        int nn=-1;
                        for(int j=0; j<nodeStatus.length;j++){
                                if(distances.getDistance(i,j)<minDist&&
                                        !nodeStatus[j]&&i!=j){
                                        nn=j;
                                        minDist=distances.getDistance(i, j);
                                }
                        }
                        //Add nearest neighbor to the route
                        tour.add(nn);
                        nodeStatus[nn]=true;
                        toRoute--;
```

```
                }
                //Add return to the initial node
                tour.add(tour.get(0));
                return tour;
        }

        @Override
        public void setInitNode(int i) {
                this.initNode=i;
        }

}
```

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.solver.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.solver.NNHeuristic` and rename it `dii.vrp.solver.NaiveNNHeuristic`. We'll need our naive implementation to conduct some experiments later
3. Refactor class `dii.vrp.solver.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.solver.NNvsNaive`. Study the class to understand the experiment. Run the experiment for different values of `T`. What do you observe?

Listing 3: Solution — NNHeuristic.java

```
package dii.vrp.solver;

import java.util.Random;

import dii.vrp.data.IDistanceMatrix;

/**
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 */
public class NNHeuristic implements ITSPHeuristic{
        /**
         * Stores the distance matrix
         */
        private IDistanceMatrix distances;
        /**
         * Stores the departure node. Default value = 0.
         */
        private int initNode=0;
        /**
         * The nearest neighbor finder
         */
```

```
	private final NNFinder finder;
	/**
	 * The random number generator
	 */
	private Random rnd=null;
	/**
	 * True if the heuristic runs in randomized mode and false otherwise.
	 * Default value = false.
	 */
	private boolean randomized;
	/**
	 * Randomization factor. Default value=1;
	 */
	private int K=1;

	public NNHeuristic(final IDistanceMatrix distances){
		this.distances=distances;
		this.finder=new NNFinder(distances, distances.size());
	}

	@Override
	public ISolution run() {

		//Init ojective function
		double of=0;

		//Initialize random number generator (if needed)
		if(randomized&&rnd==null)
			rnd=new Random();

		//Compute the number of nodes to route
		int toRoute=distances.size();

		//Initialize node status
		boolean[] nodeStatus = new boolean[toRoute];

		//Initialize the tour
		TSPSolution tour=new TSPSolution();
		int initNode=this.initNode;
		if(randomized)
			initNode=1+rnd.nextInt(K);
		tour.add(initNode);
		nodeStatus[initNode]=true;
		toRoute--;

		//Complete the tour
		int k=1;
		while(toRoute>0){
			//Find nearest neighbor
			int i=tour.get(tour.size()-1);
			if(randomized)
				k=1+rnd.nextInt(K);
			int nn=finder.findNN(i, nodeStatus,k);
			nodeStatus[nn]=true;
			tour.add(nn);
			of+=distances.getDistance(tour.get(tour.size()-2),
					tour.get(tour.size()-1));
			toRoute--;
		}
		//Add return to the initial node
		tour.add(tour.get(0));
		of+=distances.getDistance(tour.get(tour.size()-2),
				tour.get(tour.size()-1));
```

```
                tour.setOF(of);
                return tour;
        }

        @Override
        public void setInitNode(int i) {
                this.initNode=i;
        }
}
```

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.solver.NNHeuristic` so it implements the `dii.vrp.solver.IRandomizeHeuristic`. Revise your implementation of the `dii.vrp.solver.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).

Listing 4: Solution — NNHeuristic.java

```
package dii.vrp.solver;

import java.util.Random;
import dii.vrp.data.IDistanceMatrix;
import dii.vrp.solver.NNFinder;

/**
 * Implements a nearest neighbor heuristic.</br>
 * </br>
 * --- DEFAULT BEHAVIOR ---
 * <ol>
 *      <li>{@link #run} method: if the heuristic is running in randomized mode
 * and the random number generator has not been defined (by calling
 * method {@link #setRandomGen(Random)}), the random number generator is
 * initialized with an instance of {@link Random} built using the default
 * constructor.
 * <li> The default randomization factor is K=1.
 * </ol>
 * --- ASSUMPTIONS ---
 * <ol>
 *      <li>The {@link #run} method assumes that the default start node is
 * node 0.
 * </ol>
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 * @since Jan 17, 2016
 *
 */
public class NNHeuristic implements ITSPHeuristic, IRandomizedHeuristic {

        /**
         * The nearest neighbor finder
         */
        private final NNFinder finder;
        /**
         * The number of nodes in the instance
         */
        private final int n;
        /**
```

```
	 * The random number generator
	 */
	private Random rnd=null;
	/**
	 * The starting node
	 */
	private int initNode=0;
	/**
	 * The distance matrix
	 */
	private final IDistanceMatrix distances;

	/**
	 * True if the heuristic runs in randomized mode and false otherwise
	 */
	private boolean randomized=false;
	/**
	 * Randomization factor
	 */
	private int K=1;
	/**
	 * Constructs a new nearest neighbor heuristic
	 * @param finder
	 */
	public NNHeuristic(IDistanceMatrix distances){
		this.finder=new NNFinder(distances, distances.size());
		this.n=distances.size();
		this.distances=distances;
	}

	@Override
	public TSPSolution run() {

		//Init OF
		double of=0;

		//Initialized random number generator (if needed)
		if(randomized&&rnd==null)
			rnd=new Random();

		//Compute nodes to route
		int toRoute=n;

		//Initialize node status
		boolean[] nodeStatus=new boolean[toRoute];

		//Initialize the tour
		TSPSolution tour;
		int init=this.initNode;
		if(randomized)
			init=1+rnd.nextInt(K);
		tour=new TSPSolution();
		tour.add(init);
		nodeStatus[init]=true;
		toRoute--;

		//complete the tour
		int k=1;
		while(toRoute>0){
			//Find the nearest neighbor
			int i=tour.get(tour.size()-1);
			if(randomized)
				k=1+rnd.nextInt(K);
```

```
                        int nn=finder.findNN(i, nodeStatus,k);
                        tour.add(nn);
                        of+=distances.getDistance(tour.get(tour.size()-2),
                                        tour.get(tour.size()-1));
                        nodeStatus[nn]=true;
                        toRoute--;
                }
                tour.add(0);
                of+=distances.getDistance(tour.get(tour.size()-2),
                                tour.get(tour.size()-1));
                tour.setOF(of);
                return tour;
        }

        @Override
        public synchronized void setRandomized(boolean flag) {
                this.randomized=flag;
        }

        @Override
        public synchronized void setRandomGen(Random rnd) {
                this.rnd=rnd;
        }

        @Override
        public synchronized boolean isRandomized() {
                return this.randomized;
        }

        @Override
        public synchronized void setRandomizationFactor(int K) {
                this.K=K;
        }

        @Override
        public void setInitNode(int i) {
                this.initNode=i;
        }

}
```

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`).

Listing 1: Solution — VRPSolution.java

```java
package dii.vrp.solver;

import java.util.ArrayList;
import java.util.List;

/**
 * Models a VRP solution
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 * @since Jan 21, 2016
 *
 */
public class VRPSolution implements ISolution {
	/**
	 * The list of routes
	 */
	private List<IRoute> routes;
	/**
	 * The objective function
	 */
	private double of=Double.NaN;

	public VRPSolution(){
		this.routes=new ArrayList<>();
	}

	@Override
	public double getOF() {
		return this.of;
	}

	@Override
	public void setOF(double of) {
		this.of=of;
	}
```

```
	@Override
	public ISolution clone() {
		VRPSolution clone=new VRPSolution();
		clone.routes=this.cloneRoutes();
		clone.of=this.of;
		return clone;
	}

	/**
	 * Clones the list of routes
	 * @return
	 */
	private List<IRoute> cloneRoutes(){
		List<IRoute> clone=new ArrayList<>();
		for(IRoute r:this.routes)
			clone.add(r.clone());
		return clone;
	}
	/**
	 *
	 * @return the number of routes in the solution
	 */
	public int size(){
		return this.routes.size();
	}
	/**
	 *
	 * @return a copy of the list of routes
	 */
	public List<IRoute> getRoutes(){
		return this.cloneRoutes();
	}

	public void addRoute(final IRoute r){
		this.routes.add(r.clone());
	}

	public void insertRoute(final IRoute r, int i){
		this.routes.set(i, r.clone());
	}

	@Override
	public String toString(){
		StringBuilder sb=new StringBuilder();
		sb.append(this.of+"\n");
		for(IRoute r:routes){
			sb.append(r.toString()+"\n");
		}
		return sb.toString();
	}

}
```

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.

```
V_0 := 0
for i := 1 to n do V_i := +∞ 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 ≤ W) and (cost ≤ L) then
         //here substring S_i ... 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)

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?

Listing 2: Solution — Split.java

```java
package dii.vrp.solver;

import dii.vrp.data.IDemands;
import dii.vrp.data.IDistanceMatrix;

/**
 * Implements a basic split procedure. The implementation assumes that the
 * {@link TSPSolution} passed to method {@link #split(TSPSolution)} contains
 * the depot at the begining and end of the route.
 *
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 * @since Jan 21, 2016
 *
 */
public class Split implements ISplit {
        /**
         * The distance matrix
```

```
         */
        private final IDistanceMatrix distances;
        /**
         * The customer demands
         */
        private final IDemands demands;
        /**
         * The vehicle's capacity
         */
        private final double Q;

        /**
         *
         * @param distances
         * @param demands
         * @param Q
         */
        public Split(IDistanceMatrix distances, IDemands demands, double Q){
                this.distances=distances;
                this.demands=demands;
                this.Q=Q;
        }

        @Override
        public ISolution split(TSPSolution r) {

                //Define and Initialize labels
                int[] P=new int[r.size()-1];
                //stores the predecessor labels
                double[] V=new double[r.size()-1];
                //stores the shortest path labels

                for(int i=1; i<r.size()-1;i++){
                        V[i]=Double.MAX_VALUE;
                }

                //Build the auxiliary graph and find the shortest path
                for(int i=1;i<r.size();i++){
                        //Evaluate all the arcs starting at node i

                        //Initialize route metrics
                        double load=0;
                        double cost=0;

                        //Build the arcs
                        int j=i;
                        while(load<=Q&&j<r.size()-1){
                                //Compute metrics for the route
                                load+=demands.getDemand(r.get(j));
                                //the load of the route
                                if(i==j)
                                        cost=distances.getDistance(0,r.get(j))+
                                        distances.getDistance(r.get(j),0);
                                else
                                        cost=cost-
                                        distances.getDistance(r.get(j-1),0)+
                                        distances.getDistance(r.get(j-1),
                                                        r.get(j))+
                                        distances.getDistance(r.get(j),0);

                                //Check if the arc (route) is feasible
                                if(load<=Q){
                                        if(V[i-1]+cost<V[j]){
```

```
                                                V[j]=V[i-1]+cost;
                                                P[j]=i-1;
                                        }
                                        j++;
                                }
                        }
                }

                return extractRoutes(P,V,r);
        }
        /**
         * Extracts the routes from the labels, builds a solution,
         * and evaluates the solution
         * @return a solution with the routes in the optimal partition
         * of the TSP tour
         */
        private VRPSolution extractRoutes(int[] P, double[] V, TSPSolution tsp){

                VRPSolution s=new VRPSolution();
                double of=0;
                int head=P.length-1;
                int nodesToRoute=P.length-1;
                while(nodesToRoute>0){
                        int tail=P[head]+1;
                        VRPRoute r=new VRPRoute();
                        r.add(0);
                        double load=0;
                        for(int i=tail;i<=head;i++){
                                int node=tsp.get(i);
                                r.add(node);
                                load+=demands.getDemand(node);
                                nodesToRoute--;
                        }
                        r.add(0);
                        double cost=V[head]-V[P[head]];
                        of+=cost;
                        r.setLoad(load);
                        s.addRoute(r);
                        head=P[head];
                }
                s.setOF(of);
                return s;
        }

}
```

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.

Listing 3: Solution — Split.java

```
package dii.vrp.solver;

public class SFRSHeuristic implements IOptimizationAlgorithm {
        /**
         * The split algorithm
```

```
         */
        private final ISplit s;
        /**
         * The TSP heuristic
         */
        private final ITSPHeuristic h;

        public SFRSHeuristic(ISplit s, ITSPHeuristic h){
                this.s=s;
                this.h=h;
        }

        @Override
        public ISolution run() {
                return s.split(h.run());
        }

}
```

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].

Listing 4: Solution — TSFRSHeuristic.java

```
package dii.vrp.test;

import dii.vrp.data.IDemands;
import dii.vrp.data.IDistanceMatrix;
import dii.vrp.data.VRPREPInstanceReader;
import dii.vrp.solver.ISplit;
import dii.vrp.solver.ITSPHeuristic;
import dii.vrp.solver.NNHeuristic;
import dii.vrp.solver.SFRSHeuristic;
import dii.vrp.solver.Split;

public class TSFRSHeuristic {

        public static void main(String[] args) {
                String file="./data/christofides-et-al-1979-cmt/CMT01.xml";

                //Read data from file
                IDistanceMatrix distances=null;
                IDemands demands=null;
                double Q=Double.NaN;
                try(VRPREPInstanceReader parser=new VRPREPInstanceReader(file))
                {
                        distances=parser.getDistanceMatrix();
                        demands=parser.getDemands();
                        Q=parser.getCapacity("0")
```

[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

```
;                }

                //Set up the heuristic
                ITSPHeuristic rnn=new NNHeuristic(distances);
                ISplit s=new Split(distances,demands,Q);
                SFRSHeuristic h=new SFRSHeuristic(s,rnn);

                //Run the heuristic
                System.out.println(h.run());
        }

}
```

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?

Listing 5: Solution — TRandomizedSFRSHeuristic.java

```
package dii.vrp.test;

import java.util.Random;

import dii.vrp.data.IDemands;
import dii.vrp.data.IDistanceMatrix;
import dii.vrp.data.VRPREPInstanceReader;
import dii.vrp.tp.ISolution;
import dii.vrp.tp.ISplit;
import dii.vrp.tp.NNHeuristic;
import dii.vrp.tp.SFRSHeuristic;
import dii.vrp.tp.Split;

/**
 * Runs a randomized sequence-first, route-second heuristic for the CVRP
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 * @since Jan 22, 2016
 *
 */
public class TRandomizedSFRSHeuristic {

        /**
         * Runs the experiment
         * @param args nothing
         */
        public static void main(String[] args){

                //Parameters
                int T=10000;
                //Iterations
                String file= "./data/christofides-et-al-1979-cmt/CMT01.xml";
                //Instance

                //Read data from the instance file
                IDistanceMatrix distances=null;
                IDemands demands=null;
                double Q=Double.NaN;
```

```
                try(VRPREPInstanceReader parser=new VRPREPInstanceReader(file)){
                        distances=parser.getDistanceMatrix();
                        demands=parser.getDemands();
                        Q=parser.getCapacity("0");
                }

                //Initialie the sequence-first, route-second heuristic
                NNHeuristic rnn=new NNHeuristic(distances);
                rnn.setRandomized(true);
                rnn.setRandomGen(new Random(1));
                rnn.setRandomizationFactor(3);
                ISplit split=new Split(distances, demands, Q);
                SFRSHeuristic h=new SFRSHeuristic(rnn, split);
                ISolution best=null;
                for(int t=1; t<=T;t++){
                        ISolution s=h.run();
                        if(best==null)
                                best=s;
                        if(s.getOF()<best.getOF()){
                                best=s;
                                System.out.println("***␣NEW␣BEST␣SOLUTION␣"
                                                + "FOUND␣***");
                        }
                }

                //Run the heuristic and report the results
                System.out.println(best);
        }

}
```

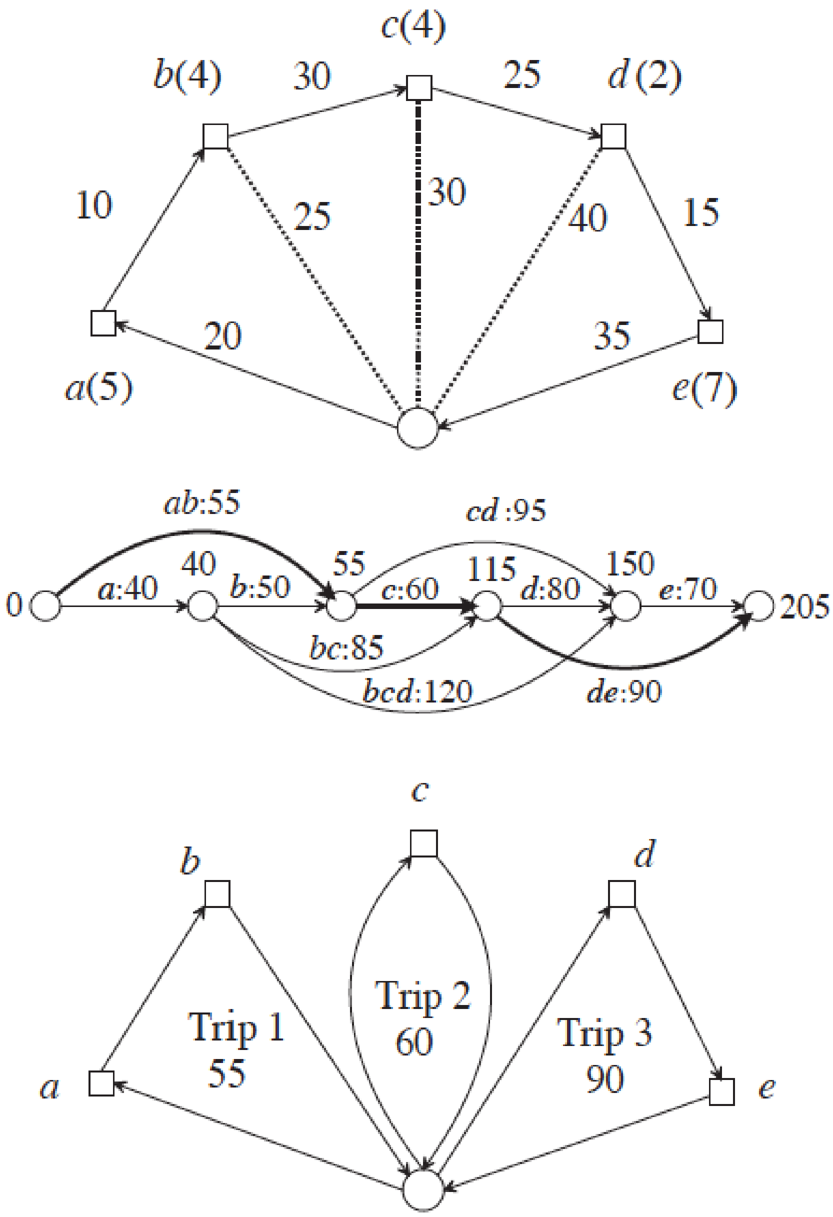

Figure 2: Example from Prins (2004)

Routage et modélisation du trafique
Jorge E. Mendoza
Polytech Tours

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.

Listing 1: Solution — Relocate.java

```
package dii.vrp.solver;

import dii.vrp.data.IDemands;
import dii.vrp.data.IDistanceMatrix;
```

```
/**
 * Implements the relocate neighborhood for the VRP
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 * @since Jan 25, 2016
 *
 */
public class Relocate implements INeighborhood {

        /**
         * Internal calss modeling a relocation movement
         * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
         * @version %I%, %G%
         * @since Jan 25, 2016
         */
        private class Relocation{
                /**
                 * The route from where the node is extracted
                 */
                int rOut;
                /**
                 * The position in route <code>rOut</code> from where the node
                 * is extracted
                 */
                int iOut;
                /**
                 * The delta in the cost of route <code>rOut</code>
                 */
                double deltaOFROut;
                /**
                 * The route where the node is to be inserted
                 */
                int rIn;
                /**
                 * The position in route <code>rIn</code> at which the node
                 * is to be inserted
                 */
                int iIn;
                /**
                 * The delta in the cost of route <code>rIn</code>
                 */
                double deltaOFRIn;
                /**
                 * The demand of the relocated node
                 */
                double demand;
                /**
                 * Construcs a new Relocate move
                 * @param rOut
                 * @param iOut
                 * @param deltaOFROut
                 * @param rIn
                 * @param iIn
                 * @param deltaOFRIn
                 * @param demand
                 */
                public Relocation(int rOut, int iOut, double deltaOFROut,
                                int rIn, int iIn, double deltaOFRIn,
                                double demand){

                        this.rOut=rOut;
                        this.iOut=iOut;
                        this.deltaOFROut=deltaOFROut;
```

```
                this.rIn=rIn;
                this.iIn=iIn;
                this.deltaOFRIn=deltaOFRIn;

                this.demand=demand;

            }
    }

    /**
     * The exploration strategy. Default value
     * {@link ExplorationStrategy#BEST_IMPROVEMENT}.
     */
    private ExplorationStrategy strategy=ExplorationStrategy.BEST_IMPROVEMENT;
    /**
     * The distance matrix
     */
    private final IDistanceMatrix distances;
    /**
     * The customer demands
     */
    private final IDemands demands;
    /**
     * The capacity of the truck
     */
    private final double Q;
    /**
     * Tolerance
     */
    private final double epsilon=0.00001;

    /**
     * Construcs a new Relocate neighborhood
     * @param distances the distance matrix
     * @param demands the customer demands
     * @param Q the capacity of the trucks
     */
    public Relocate(IDistanceMatrix distances, IDemands demands, double Q){
            this.distances=distances;
            this.demands=demands;
            this.Q=Q;
    }

    @Override
    public void setExplorationStrategy(ExplorationStrategy strategy) {
            this.strategy=strategy;
    }

    @Override
    public ExplorationStrategy getStrategy() {
            return this.strategy;
    }

    @Override
    public VRPSolution explore(final ISolution s) {

            //Set best solution found
            //Make a copy of the initial solution
            VRPSolution best=(VRPSolution) s.clone();

            //Initilize auxiliary variables
            //The best relocate move we have found so far
```

```
        Relocation bestMove=null;
        //The best OF improvement we have found
        double bestDelta=0;
        //The last OF delta we have evaluated.
        //Improving moves have a negative delta.
        double delta=Double.NaN;
        //The ID of the node we are trying to relocate
        int node=-1;
        //The savings
        double savings=Double.NaN;
        //The cost
        double cost=Double.NaN;
        //The demand of the node we are trying to relocate
        double demand=Double.NaN;
        //The load of the last considered route
        double load=Double.NaN;

        //Check all possible movements
        for(int route=0;route<best.size();route++){
                //We assume the depot appears at the beginning
                //and end of each route
          for(int position=1;position<best.size(route)-1;
            position++){
                //The node to relocate
                node=best.getNode(route, position);
                demand=demands.getDemand(node);
                savings=this.getSavings(best, route, position);
                //Try re-locating the node at each possible position
                for(int r=0;r<best.size()-1;r++){
                        load=best.getLoad(r);
                        //If the move is feasible in terms of the
                        //capacity constraint
                        if(load+demand<Q||route==r)
                                for(int i=1; i<best.size(r)-1;
                                        i++){
                                //the relocation leads to the same solution
                                cost=route==r&&
                                  (i-position==1||i-position==0)?
                                Double.MAX_VALUE:this.getCost(best, r, i, node);
                                delta=cost-savings;
                                //If the delta in the objective function
                                //is better than the best delta
                                if(delta<bestDelta-epsilon)
                                  if(this.strategy==
                                        ExplorationStrategy.FIRST_IMPROVEMENT)
                                        return this.executeMove(
                                         new Relocation(route,position,savings,r,
                                                i,cost,demand),best);
                                  else{
                                    bestMove=new Relocation(route,position,
                                          savings,r,i,cost,demand);
                                        bestDelta=delta;
                                  }
                                }
                        }
                }
        }
        return this.executeMove(bestMove,best);
}
/**
 * Executes a relocate move on a solution
 * @param m the movement to execute
 * @param s the solution to which the move is to executed
```

```java
	 * @return a modified and up-to-date solution
	 */
	private VRPSolution executeMove(Relocation m,VRPSolution s){

		if(m==null)
			return null;

		//Make the move
		if(m.rOut==m.rIn&&m.iOut<m.iIn)
			s.insert(s.remove(m.rOut, m.iOut), m.rIn, m.iIn-1);
		else
			s.insert(s.remove(m.rOut, m.iOut), m.rIn, m.iIn);
		//Update the cost
		s.setCost(m.rOut,s.getCost(m.rOut)-m.deltaOFROut);
		s.setCost(m.rIn,s.getCost(m.rIn)+m.deltaOFRIn);
		s.setOF(s.getOF()-m.deltaOFROut+m.deltaOFRIn);
		//Update load
		if(m.rIn!=m.rOut){
			s.setLoad(m.rOut,s.getLoad(m.rOut)-m.demand);
			s.setLoad(m.rIn,s.getLoad(m.rIn)+m.demand);
		}
		s.removeRoutesBysize(2);
		return s;
	}

	/**
	 * Computes the saving generated by extracting the node in
	 * position <code>i</code> from route <code>r</code> in solution
	 * <code>s</code>
	 * @param s the solution
	 * @param r the route
	 * @param i the position of the node to extract
	 * @return the saving generated by extracting the node in
	 * position <code>i</code> from route <code>r</code> in solution
	 * <code>s</code>
	 */
	private double getSavings(final VRPSolution s,int r, int i){
		int node=s.getNode(r, i);
		int pIOut=s.getNode(r, i-1);
		int sIOut=s.getNode(r, i+1);
		return distances.getDistance(pIOut,node)+
				distances.getDistance(node,sIOut)-
				distances.getDistance(pIOut,sIOut);
	}
	/**
	 * Computes the cost generated by inserting node <code>node</code>
	 * into position <code>i</code> of route <code>r<code> in solution
	 * <code>s</code>
	 * @param s the solution
	 * @param r the route
	 * @param i the position in the route
	 * @param node the node to insert
	 * @return the cost generated by inserting node <code>node</code>
	 * into position <code>i</code> of route <code>r<code> in solution
	 * <code>s</code>
	 */
	private double getCost(final VRPSolution s, int r, int i, int node){
		int pIIn=s.getNode(r, i-1);
		int sIIn=s.getNode(r, i);
		return distances.getDistance(pIIn,node)+
				distances.getDistance(node,sIIn)-
				distances.getDistance(pIIn,sIIn);
	}
```

```
}
```

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).

Listing 2: Solution — TRelocate.java

```java
package dii.vrp.test;

import dii.vrp.data.CMT01;
import dii.vrp.data.IDemands;
import dii.vrp.data.IDistanceMatrix;
import dii.vrp.data.VRPREPInstanceReader;
import dii.vrp.solver.CVRPRouteEvaluator;
import dii.vrp.solver.INeighborhood;
import dii.vrp.solver.Relocate;
import dii.vrp.solver.VRPSolution;

public class TRelocate {

        public static void main(String[] args) {

                //Parameters
                //Instance
                String file= "./data/christofides-et-al-1979-cmt/CMT01.xml";

                //Read data from an instance file
                IDistanceMatrix distances=null;
                IDemands demands=null;
                double Q=Double.NaN;
                try(VRPREPInstanceReader parser=new VRPREPInstanceReader(file)){
                        distances=parser.getDistanceMatrix();
                        demands=parser.getDemands();
                        Q=parser.getCapacity("0");
                }

                //Create a route evaluator
                CVRPRouteEvaluator evaluator=new CVRPRouteEvaluator(distances,
                                demands);

                //Create the optimal solution for instance CMT01
                VRPSolution s=CMT01.getS1(evaluator);

                System.out.println(s);

                //Set up neighborhood
                INeighborhood n=new Relocate(distances, demands, Q);

                //Search the neighborhood
                s=(VRPSolution) n.explore(s);

                System.out.println(s);

        }

}
```

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.

Listing 3: Solution — TRelocate.java

```java
package dii.vrp.solver;

import java.util.ArrayList;
/**
 * Implemenst a variable neighborhood descent
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 * @since Jan 25, 2016
 *
 */
public class VND implements ILocalSearch {
        /**
         * The list of neighborhoods
         */
        private final ArrayList<INeighborhood> neighborhoods;
        /**
         * Construcs a new variable neighborhood descend algorithm
         * @param neighborhood the neighborhood in position 1 of the list
         */
        public VND(final INeighborhood neighborhood){
                neighborhoods=new ArrayList<INeighborhood>();
                neighborhoods.add(neighborhood);
        }
        /**
         * Adds a new neighborhood
         * @param neigborhood the neighborhood to add
         */
        public void addNeighborhood(final INeighborhood neigborhood){
                neighborhoods.add(neigborhood);
        }

        @Override
        public ISolution run(final ISolution initSol,
                        final OptimizationCriterion sense) {
                ISolution best=initSol.clone();
                for(int k=0; k<neighborhoods.size(); k++){
                        boolean stop=false;
                        while(!stop){
                                INeighborhood n=neighborhoods.get(k);
                                ISolution s=n.explore(best.clone());
                                //The neighborhood exploration was sucessfull
                                if(s==null)
                                        stop=true;
                                else
                                  if((sense==OptimizationCriterion.MINIMIZATION
                                   &&s.getOF()<best.getOF())||(sense==
                                   OptimizationCriterion.MAXIMIZATION
                                   &&s.getOF()>best.getOF())){
                                        //Update best solution
                                        best=s;
                                        k=0;
                                   }
                        }
```

```
                }
                return best;
        }

}
```

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()`.

Listing 4: Solution — TVND.java

```
package dii.vrp.test;

import dii.vrp.data.CMT01;
import dii.vrp.data.IDemands;
import dii.vrp.data.IDistanceMatrix;
import dii.vrp.data.VRPREPInstanceReader;
import dii.vrp.solver.CVRPRouteEvaluator;
import dii.vrp.solver.OptimizationCriterion;
import dii.vrp.solver.Relocate;
import dii.vrp.solver.VND;
import dii.vrp.solver.VRPSolution;

public class TVND {

        public static void main(String[] args) {

                //Parameters
                //Instance
                String file= "./data/christofides-et-al-1979-cmt/CMT01.xml";

                //Read data from an instance file
                IDistanceMatrix distances=null;
                IDemands demands=null;
                double Q=Double.NaN;
                try(VRPREPInstanceReader parser=new VRPREPInstanceReader(file)){
                        distances=parser.getDistanceMatrix();
                        demands=parser.getDemands();
                        Q=parser.getCapacity("0");
                }

                //Create a route evaluator
                CVRPRouteEvaluator evaluator=new CVRPRouteEvaluator(distances,
                                demands);

                //Create the optimal solution for instance CMT01
                VRPSolution s=CMT01.getS2(evaluator);

                System.out.println(s);

                //Set up the VND
                VND ls=new VND(new Relocate(distances, demands, Q));

                //Search the neighborhood
                s=(VRPSolution)ls.run(s,OptimizationCriterion.MINIMIZATION);

                System.out.println(s);

        }

}
```

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.

Listing 5: Solution — GRASP.java

```
package dii.vrp.solver;

/**
 * Implements a simple GRASP algorithm
 * @author Jorge E. Mendoza (dev@jorge-mendoza.com)
 * @version %I%, %G%
 * @since Jan 19, 2016
 *
 */
public class GRASP implements ILocalSearch{

	/**
	 * The heuristic responsible for generating initial solutions
	 */
	private final IRandomizedHeuristic h;
	/**
	 * The local search procedure
	 */
	private final ILocalSearch ls;
	/**
	 * The number of iterations
	 */
	private final int T;
	/**
	 * Constructs a new GRASP heuristic
	 * @param initSolGenerator the heuristic responsible for generating
	 * initial solutions
	 * @param localSearch the local search procedure
	 * @param iterations the number of iterations
	 */
	public GRASP(IRandomizedHeuristic initSolGenerator, ILocalSearch
			localSearch, int iterations){
		h=initSolGenerator;
		this.ls=localSearch;
		this.T=iterations;
	}

	@Override
	public ISolution run(final ISolution initSol,
			final OptimizationCriterion sense) {
		ISolution best;
		if(initSol==null)
			best=h.run();
		else
			best=initSol.clone();
		for(int t=1;t<=T;t++){
			ISolution s=h.run();
			s=ls.run(s.clone(),sense);
			if(sense==OptimizationCriterion.MINIMIZATION
					&&s.getOF()<best.getOF()||
					sense==OptimizationCriterion.MAXIMIZATION
					&&s.getOF()>best.getOF()
					)
```

```java
                                best=s;
                }
                return best;
        }

}
```

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.

Listing 6: Solution — TGRASP.java

```java
package dii.vrp.test;


import java.util.Random;

import dii.vrp.data.IDemands;
import dii.vrp.data.IDistanceMatrix;
import dii.vrp.data.VRPREPInstanceReader;
import dii.vrp.solver.ExplorationStrategy;
import dii.vrp.solver.GRASP;
import dii.vrp.solver.NNHeuristic;
import dii.vrp.solver.OptimizationCriterion;
import dii.vrp.solver.Relocate;
import dii.vrp.solver.SFRSHeuristic;
import dii.vrp.solver.Split;
import dii.vrp.solver.VND;
import dii.vrp.solver.VRPSolution;

public class TGRASP {

        public static void main(String[] args) {

                //Parameters
                //Instance
                String file= "./data/christofides-et-al-1979-cmt/CMT01.xml";
                int T=1000;

                //Read data from an instance file
                IDistanceMatrix distances=null;
                IDemands demands=null;
                double Q=Double.NaN;
                try(VRPREPInstanceReader parser=new VRPREPInstanceReader(file)){
                        distances=parser.getDistanceMatrix();
                        demands=parser.getDemands();
                        Q=parser.getCapacity("0");
                }

                //Set up GRASP
                NNHeuristic nn=new NNHeuristic(distances);
                Split split=new Split(distances, demands, Q);
                SFRSHeuristic h=new SFRSHeuristic(nn, split);
                h.setRandomized(true);
                h.setRandomizationFactor(3);
                h.setRandomGen(new Random(1));
                Relocate relocate=new Relocate(distances, demands, Q);
                relocate.setExplorationStrategy(
                                ExplorationStrategy.FIRST_IMPROVEMENT);
                VND vnd=new VND(relocate);
                GRASP grasp=new GRASP(h, vnd, T);
```

```
                //Run GRASP
                VRPSolution s=(VRPSolution)grasp.run(null,
                                OptimizationCriterion.MINIMIZATION);

                //Report solution
                System.out.println(s);

        }

}
```

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?.