The OPT4J Tutorials

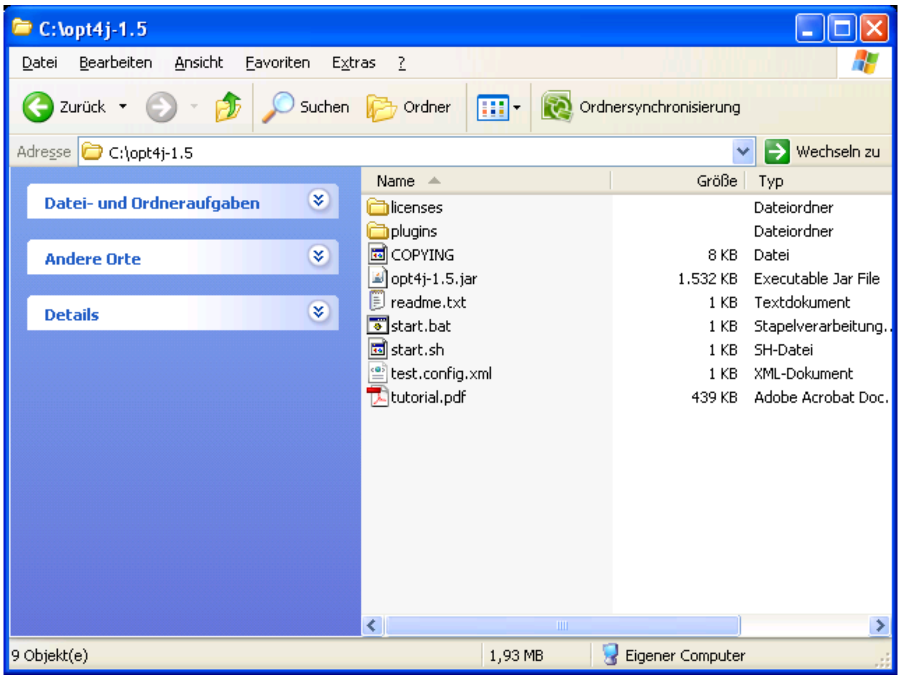
November/19/2008

This tutorial consists of six sections: The first section is an installation guide for OPT4J, followed by a quickstart that introduces the GUI. The third section outlines the structure of OPT4J. Concluding, two examples show how to implement a problem (Traveling Salesman and MinOnes) and one example shows how to implement an optimizer (MutateOptimizer).

1 Installing OPT4J

To install OPT4J, the first step is to ensure that at least Java 6 Runtime Environment (JRE) is installed on your system. (You can use `java -version` on your command line (console) to check the version of your JRE.) If this is not the case, visit `http://java.sun.com`, download, and install the latest JRE for your operating system.

Next, you have to download the OPT4J jar from `http://www.opt4j.org`. Download the latest release (currently, the zip-file **opt4j-1.5.zip**). Unzip all files.

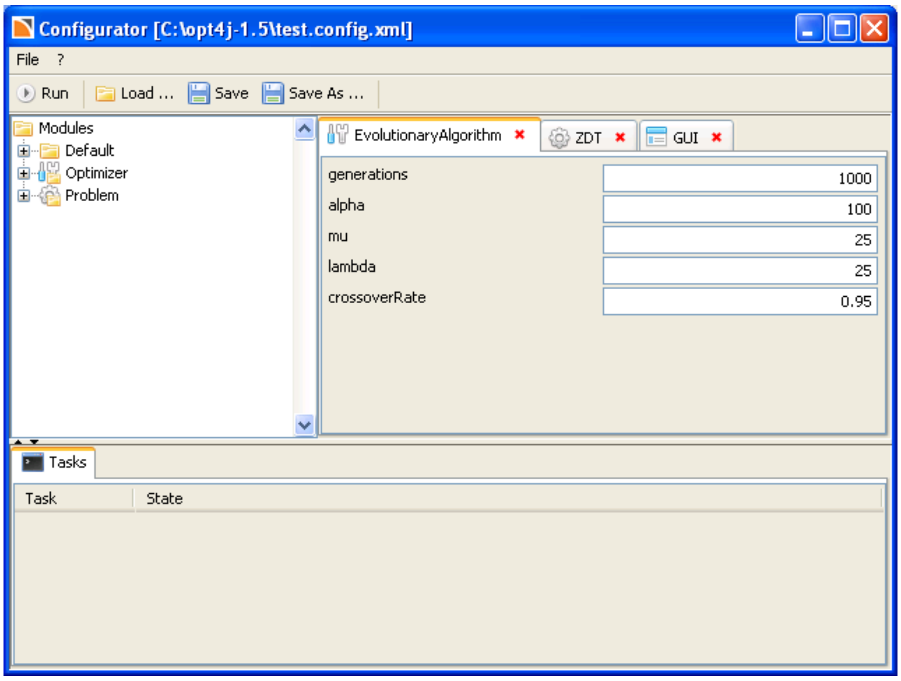


Name	Größe	Typ
licenses		Dateiordner
plugins		Dateiordner
COPYING	8 KB	Datei
opt4j-1.5.jar	1.532 KB	Executable Jar File
readme.txt	1 KB	Textdokument
start.bat	1 KB	Stapelverarbeitung..
start.sh	1 KB	SH-Datei
test.config.xml	1 KB	XML-Dokument
tutorial.pdf	439 KB	Adobe Acrobat Doc.

On Windows systems, you can start the OPT4J configuration GUI with the **start.bat** file (or double-click the **opt4j-1.5.jar** file). On UNIX systems, use **start.sh**. Alternatively, you can use the command `java -jar opt4j-1.5.jar`.

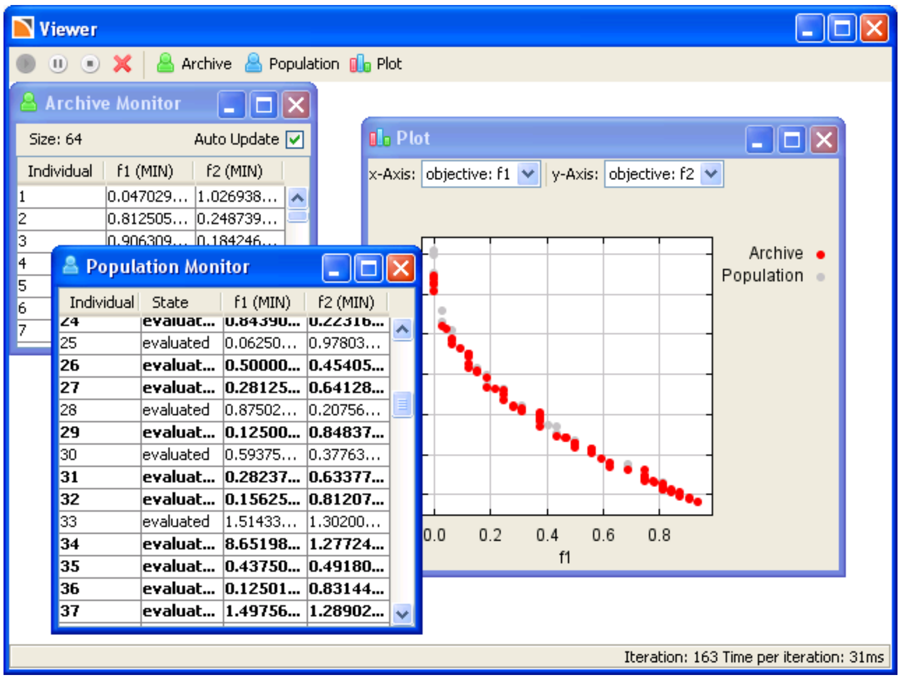
2 Quickstart

For a first test of OPT4J, you can select the following modules or you can load the **test.config.xml** file:


Configurator [C:\opt4j-1.5\test.config.xml]

generations	1000
alpha	100
mu	25
lambda	25
crossoverRate	0.95

Finally, click the *run* button to start the optimization:


Archive Monitor

Size: 64 Auto Update ✓

Individual	f1 (MIN)	f2 (MIN)
1	0.047029…	1.026938…
2	0.812505…	0.248739…
3	0.906309…	0.184246…
4		
5		
6		
7		

Population Monitor

Individual	State	f1 (MIN)	f2 (MIN)
24	**evaluat…**	**0.84390…**	**0.22316…**
25	evaluated	0.06250…	0.97803…
26	**evaluat…**	**0.50000…**	**0.45405…**
27	**evaluat…**	**0.28125…**	**0.64128…**
28	evaluated	0.87502…	0.20756…
29	**evaluat…**	**0.12500…**	**0.84837…**
30	evaluated	0.59375…	0.37763…
31	**evaluat…**	**0.28237…**	**0.63377…**
32	**evaluat…**	**0.15625…**	**0.81207…**
33	evaluated	1.51433…	1.30200…
34	**evaluat…**	**8.65198…**	**1.27724…**
35	**evaluat…**	**0.43750…**	**0.49180…**
36	**evaluat…**	**0.12501…**	**0.83144…**
37	**evaluat…**	**1.49756…**	**1.28902…**

Plot

x-Axis: objective: f1 y-Axis: objective: f2

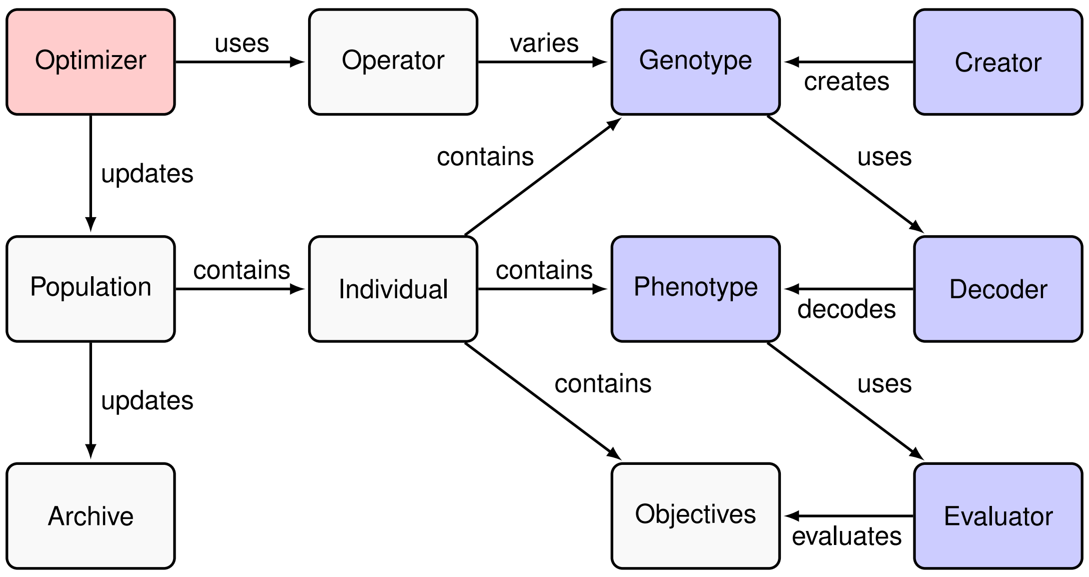
Iteration: 163 Time per iteration: 31ms

The OPT4J configuration files are *XML* files that define the used modules and adjusted properties. Starting these configuration files without the configuration GUI can be done using the command `java -jar opt4j-1.5.jar -s test.config.xml` with **test.config.xml** being the XML configuration. Using the `-s` parameter, you can specify multiple configurations that are executed subsequently.

3 Understanding OPT4J

3.1 The Interfaces Concept

OPT4J consists of many interfaces while its up to the user has to set the implementations for these interfaces. A schematic overview of the most important OPT4J interfaces and classes can be found in the following figure:



The most important interfaces of the framework are explained in the following.

3.1.1 Optimizer (Singleton)

All *Optimizers* implement the *Optimizer* interface. Usually all *Optimizers* are derived from the *Abstract-Optimizer* that already implements some essential methods.

3.1.2 Creator, Decoder, Evaluator (Singletons)

The *Creator*, *Decoder*, and *Evaluator* interfaces are interfaces for the *Problem*. The *Creator* is responsible for the construction of random *Genotypes*, the *Decoder* converts the *Genotype* into a *Phenotype*, and the *Evaluator* calculates the *Objectives* for a *Phenotype*.

3.1.3 Population (Singleton)

The *Population* is the interface and the implementation in one class. The *Population* contains all current *Individuals*.

3.1.4 Archive (Singleton)

The *Archive* interface is by default bound to the *UnboundedArchive*. The *Archive* contains the best *Individuals* that were found so far.

3.1.5 Copy, Mutate, Neighbor, Crossover (Singletons)

The *Operators* like *Copy*, *Mutate*, *Neighbor*, and *Crossover* are interfaces that are by default bound to generic types that can handle different types of *Genotypes*. *Operators* can change or create *Genotypes*.

3.1.6 Completer (Singleton)

The *Completer* is responsible for decoding and evaluating *Individuals*. By default, the *Completer* is bound to the *SequentialCompleter* that completes *Individuals* subsequently.

3.2 Using the Guice Dependency Injection

OPT4J is using the *Google Guice Dependency Injection* to put everything together. As mentioned, the framework consists of many interfaces and the user has to set the implementations for these interfaces. For instance, the *Optimizer* interface can be implemented by the *EvolutionaryAlgorithm*. Which interface is bound to which implementation is defined in the *Modules*.

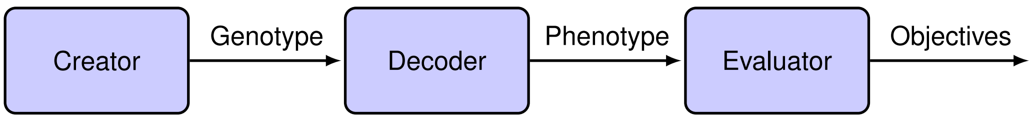
4 Problem Tutorial 1: Traveling Salesman

The Traveling Salesman Problem (http://en.wikipedia.org/wiki/Traveling_salesman_problem) is a classic single objective optimization problem. Given a set of *Cities*, the goal is to find the shortest round trip (*route*) that contains all cities. In this tutorial, it is shown first how to implement the TSP as a *Problem* for OPT4J. Afterwards, a simple visualization for the TSP and how to include it in the OPT4J GUI is presented.

4.1 Problem

To compile the following code, you have to ensure that *opt4j-1.5.jar* is on the classpath.

Put simply, a *Problem* is defined by the *Creator*, *Decoder*, and *Evaluator*. The *Creator* creates a new *Genotype* and passes it to the *Decoder*. The *Decoder* translates the *Genotype* into a *Phenotype*. The *Evaluator* evaluates the *Objectives*.



In the Traveling Salesman Problem, the *Genotype* is a *PermutationGenotype* of the *Cities* representing one round trip (*Route*). The *Phenotype* is such a *Route*. The *Creator*, *Decoder*, and *Evaluator* interfaces are, thus, implemented by the *SalesmanCreator*, the *SalesmanDecoder*, and the *SalesmanEvaluator*.

The ***SalesmanProblem*** contains the problem description. It is defined as a set of 100 *Cities* over the area 100 times 100. The class offers a public method (`getCities()`) to retrieve these *Cities*:

SalesmanProblem.java

```java
public class SalesmanProblem {

  protected Set<City> cities;

  public class City {
    protected final double x;
    protected final double y;

    public City(double x, double y) {
      this.x = x;
      this.y = y;
    }

    public double getX() {
      return x;
    }

    public double getY() {
      return y;
    }
  }
  public SalesmanProblem() {
    cities = new HashSet<City>();

    Random random = new Random(0);

    for (int i = 0; i < 100; i++) {
      final double x = random.nextDouble() * 100;
      final double y = random.nextDouble() * 100;
      final City city = new City(x, y);

      cities.add(city);
    }
  }

  public Set<City> getCities() {
    return cities;
  }
}
```

The ***SalesmanCreator*** has to create *PermutationGenotypes*. Each of them contains all *Cities* in a randomized order. The randomization is done using the `shuffle()` method of the `Collections` class:

SalesmanCreator.java

```
public class SalesmanCreator implements Creator<PermutationGenotype<City>> {

  protected final SalesmanProblem problem;

  @Inject
  public SalesmanCreator(SalesmanProblem problem) {
    this.problem = problem;
  }

  public PermutationGenotype<City> create() {
    PermutationGenotype<City> genotype = new PermutationGenotype<City>();
    for (City city : problem.getCities()) {
      genotype.add(city);
    }

    Collections.shuffle(genotype);

    return genotype;
  }
}
```

Note: The constructor is annotated with `@Inject` which is a Guice Annotation indicating the main constructor. Guice will automatically construct this *Creator* and pass the *SalesmanProblem* into the constructor.

Now, a translation of the *Genotype* to the *Phenotype* has to done by the *Decoder*. In this case, the *Phenotype* is a *Route* representing a round trip. The ***Route*** is given as an ordered *List* of *Cities*.

Route.java

```java
  public class Route extends ArrayList<City> implements Phenotype{}
```

Since the *SalesmanGenotype* is already given as a permutation of all cities, the ***SalesmanDecoder*** is a simple copy operation of the elements of the *PermutationGenotype* to the *Route*.

SalesmanDecoder.java

```java
public class SalesmanDecoder implements
    Decoder<PermutationGenotype<City>, Route> {

  public Route decode(PermutationGenotype<City> genotype) {
    Route route = new Route();
    for (City city : genotype) {
      route.add(city);
    }
    return route;
  }
}
```

The next task is to evaluate the distance of a *Route*. The ***SalesmanEvaluator*** sums up the Euclidean *distances* between each point (city) of the route. The result is returned as the *Objectives* that can consist of the distance *Objective* and and the calculated distance value pair. Note: The *Objectives* can contain multiple *Objective* - *Value* pairs.

SalesmanEvaluator.java

```java
public class SalesmanEvaluator implements Evaluator<Route> {

  Objective distance = new Objective("distance", Sign.MIN);

  public Objectives evaluate(Route route) {
    double dist = 0;
    for (int i = 0; i < route.size(); i++) {
      City one = route.get(i);
      City two = route.get((i + 1) % route.size());
      dist += getEuclideanDistance(one, two);
    }
    Objectives objectives = new Objectives();
    objectives.add(distance, dist);

    return objectives;
  }

  private double getEuclideanDistance(City one, City two) {
    final double x = one.getX() - two.getX();
    final double y = one.getY() - two.getY();
    return Math.sqrt(x * x + y * y);
  }

  public List<Objective> getObjectives() {
    return Arrays.asList(distance);
  }
}
```

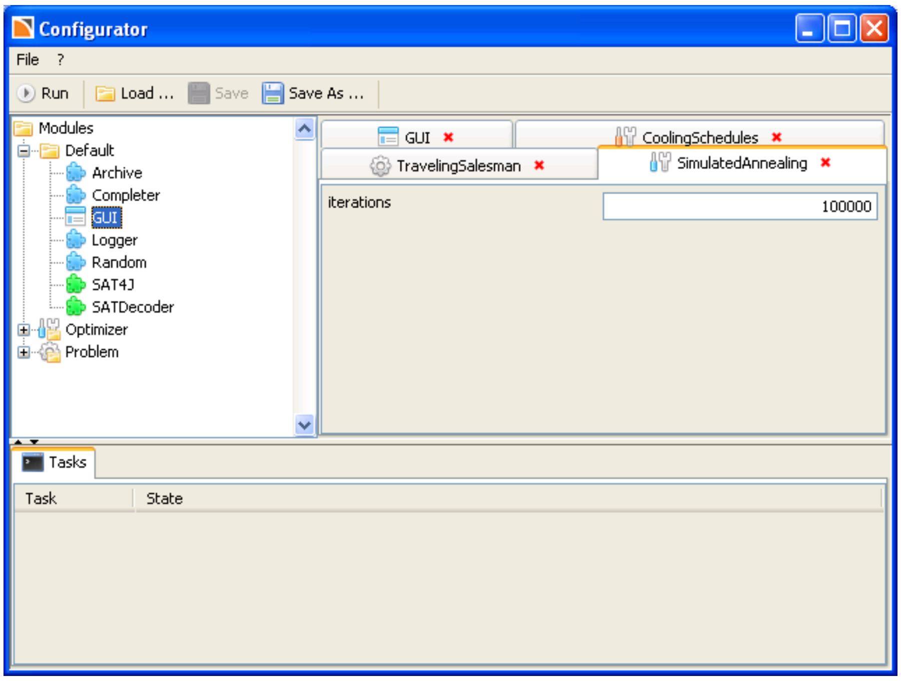
This TSP has a single objective that has to be minimized. Thus, the `distance` *Objective* is the only objective with the name `"distance"` and has to be minimized (`Sign.MIN`).

At this point the *Creator*, *Decoder*, *Evaluator*, *Genotype*, and *Phenotype* interfaces are implemented by the introduced classes for the TSP. Whats left to do is *binding* the implementations to the interfaces. Therefore, we need a so called *module*. Constructing and wiring the implemented classes is accomplished within the Guice framework thats is integrated in OPT4J (`http://code.google.com/p/google-guice/`). The following class is also understandable without knowledge of *Guice*. The ***TravelingSalesmanModule*** is derived from the *ProblemModule* that identifies this as a configurable problem-related OPT4J module.

TravelingSalesmanModule.java

```java
public class TravelingSalesmanModule extends ProblemModule {

  public void configure() {
    bindProblem(SalesmanCreator.class, SalesmanDecoder.class,
        SalesmanEvaluator.class);
  }

}
```

Now we can start the *ModuleViewer* and if the *TravelingSalesmanModule* is on the classpath it will appear in the GUI. You can simply export the classes into one jar and put this into the **plugins** folder of OPT4J. All classes in the **plugins** folder are added automatically to the classpath. You can use the following configuration to start the optimization process.



iterations 100000

4.2 Visualization

Loading the *GUIModule* gives the user the opportunity to monitor the optimization process. In the following, this tutorial will show how we can implement a GUI that displays the current optimal *route*.

The ***SalesmanWidget*** shows the route graphically by double clicking the *route* in the *ArchiveViewer* or right clicking and selecting „show route“:

SalesmanWidget.java

```java
// The SalesmanWidget is an additional feature of this tutorial. It enables a view
// of a single route in the GUI.
public class SalesmanWidget implements IndividualMouseListener {

  protected final ArchiveWidget archiveWidget;
  protected final GUIFrame frame;

  // Panel that paints a single Route (that is the phenotype of an Individual)
  @SuppressWarnings("serial")
  public class MyPanel extends JPanel {

    protected final Individual individual;

    public MyPanel(Individual individual) {
      super();
      this.individual = individual;
      setPreferredSize(new Dimension(208, 208));
    }

    protected void paintComponent(Graphics g) {
      Graphics2D g2d = (Graphics2D) g;
      g2d.setBackground(Color.WHITE);
      g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
          RenderingHints.VALUE_ANTIALIAS_ON);
      g2d.setStroke(new BasicStroke(2f));
      g2d.clearRect(0, 0, 208, 212);

      Route route = (Route) individual.getPhenotype();

      for (int i = 0; i < route.size(); i++) {
        final int j = (i + 1) % route.size();
        City one = route.get(i);
        City two = route.get(j);

        int x1 = (int) (one.getX() * 2) + 4;
        int y1 = (int) (one.getY() * 2) + 4;
        int x2 = (int) (two.getX() * 2) + 4;
        int y2 = (int) (two.getY() * 2) + 4;

        g2d.drawLine(x1, y1, x2, y2);
        g2d.drawOval(x1 - 2, y1 - 2, 4, 4);

      }
    }
  }
```

```java
// Use a custom widget
@WidgetParameters(multiple = true, button = false, open = true, frameTitle = "
    Route", resizable = false, maximizable = false)
protected class MyWidget implements Widget {

  final Individual individual;

  public MyWidget(Individual individual) {
    super();
    this.individual = individual;
  }

  public JPanel getPanel() {
    JPanel panel = new MyPanel(individual);
    return panel;
  }

  public void init(Viewport viewport) {
  }

}

// The route is shown by a double click of a individual in the archive
// monitor panel. Thus we need the ArchiveMonitorPanel and the main
// GUIFrame.
@Inject
public SalesmanWidget(ArchiveWidget archiveWidget, GUIFrame frame) {
  this.archiveWidget = archiveWidget;
  this.frame = frame;
}

// We register this class as a mouse listener of the archive monitor panel.
// This method is called automatically by guice (Inject annotation).
@Inject
public void init() {
  archiveWidget.addIndividualMouseListener(this);
}

// If an individual is double clicked, paint the route.
public void onDoubleClick(Individual individual, Component table, Point p) {
  paintRoute(individual);
}

// If an individual is clicked with the right mouse button, open a popup
// menu that contains the option to paint the route.
public void onPopup(final Individual individual, Component table, Point p) {
  JPopupMenu menu = new JPopupMenu();
  JMenuItem paint = new JMenuItem("show route");
  menu.add(paint);

  paint.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      paintRoute(individual);
    }
  });
  menu.show(table, p.x, p.y);
  }

  // Paint the route: Construct a JInternalFrame, add the MyPanel and add the
  // frame to the desktop of the main GUIFrame.
  protected void paintRoute(Individual individual) {
    Widget widget = new MyWidget(individual);
    frame.addWidget(widget);
  }

}
```

This GUI needs the *ArchiveMonitorPanel* to register the *Listener* for double-click and right-click as well as the *GUIFrame* to get the *Desktop* of the *Frame*.

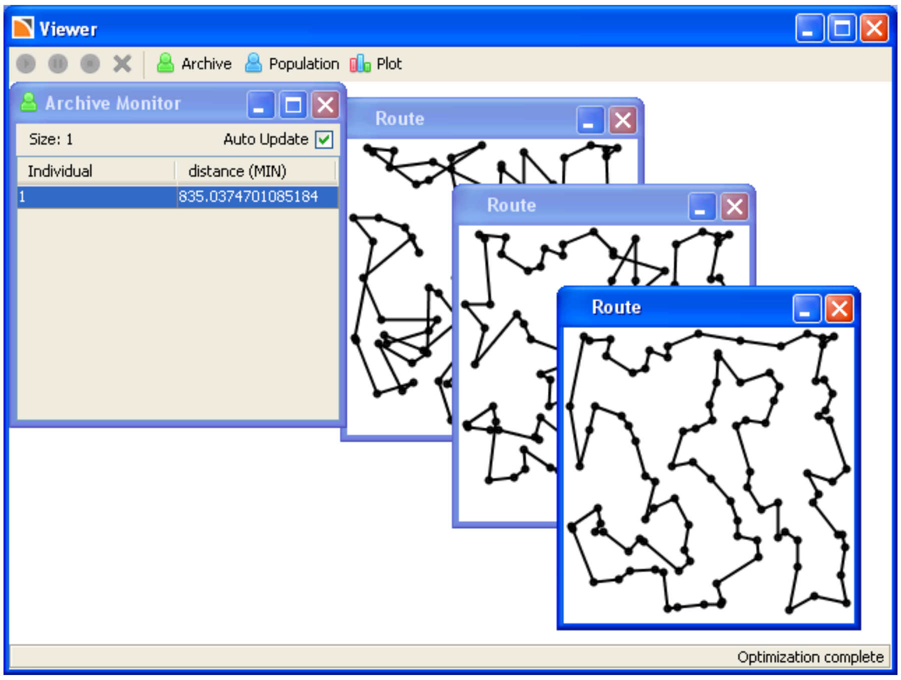
The ***TravelingSalesmanModule*** has to be extended with a single line that tells that the *SalesmanGUI* is loaded as a *singleton* and *independently*. For further information on this topic we recommend to read the *Guice* manual.

TravelingSalesmanModule.java

```
public class TravelingSalesmanModule extends ProblemModule {

  @Name("Use the GUI?")
  boolean useGui = true;

  public boolean isUseGui() {
    return useGui;
  }

  public void setUseGui(boolean useGui) {
    this.useGui = useGui;
  }

  public void configure() {
    bindProblem(SalesmanCreator.class, SalesmanDecoder.class,
        SalesmanEvaluator.class);
    if (useGui) {
      bind(SalesmanWidget.class).asEagerSingleton();
    }
  }

}
```

The ***TravelingSalesmanModule*** has to be extended with a single line that states that the *SalesmanGUI* is loaded as a *singleton* and *independently*. For further information on this topic we recommend to read the *Guice* manual.

The result is the following:



Size: 1 Auto Update ✓

Individual	distance (MIN)
1	835.0374701085184

Optimization complete

5 Problem Tutorial 2: MinOnes Optimization of 3-SAT

Many optimization problems with a discrete search space consist of binary variables. However, in some cases not all representations of these variables are *feasible*. Common approaches deteriorate the objectives if the solution is not feasible. As a matter of fact, in case that the number of feasible solutions is much lower than the number of infeasible solutions, the optimization process is more focused on the search of feasible solutions than optimizing the objectives.

The *SAT-Decoding* package (*org.opt4j.sat*) in OPT4J includes our latest research on this topic. The specialized *Decoder* gives the user the opportunity to define constraints that have to be fulfilled to obtain a feasible solution. Thus, the Decoder takes over the task to deliver feasible solutions and, thus, allows the optimizer to deal with feasible solutions only.

As an example for the SAT-Decoding we will use the following example: Given is a vector with a fixed number of binary (*0/1*) variables. The objective is to minimize the number of *1s* of this vector. The solution of this problem is quite simple: Set all variables to *0*. However, in order to make this problem more difficult, we introduce the following condition: A solution is only considered feasible if the variables satisfy a set of *constraints*. The following example is used to illustrate this problem:

```
minimize w + x + y + z
subject to:
        w + x + y > 0 (constraint 1)
        y + z > 0     (constraint 2)
        w + z > 0     (constraint 3)
        x + y + z > 0 (constraint 4)
```

Now `(w=1,x=1,y=0,z=1)` is a feasible solution that fulfills all constraints and the objective is 3. On the other hand, for `(w=0,x=0,y=0,z=0)`, the objective is 0 but this solution is not feasible and therefore worthless. A good solution that fulfills all constraints would be `(w=0,x=1,y=0,z=1)`with the objective value 2.

In general, there is no simple algorithm that delivers feasible solutions for a set of linear constraints with binary variables since this problem is known to be *NP-complete*. However, the SAT-Decoding technique utilizes a *Pseudo-Boolean solver* to obtain feasible solutions only.

With this background knowledge, the following tutorial gives a short introduction how to use the SAT-Decoding package. First, the solution vector is created as the *Phenotype* of this problem. The variables here are integers. However, one solution is put into the **Result.java** which is a simple map from the variables (integers) to Boolean values.

```java
public class Result extends HashMap<Integer, Boolean> implements Phenotype{}
```

The ***MinOnesEvaluator*** simply counts the number of *1s*.

MinOnesEvaluator.java

```java
public class MinOnesEvaluator implements Evaluator<Result> {

  Objective ones = new Objective("ones", Sign.MIN);

  public Objectives evaluate(Result result) {

    int value = 0;
    for (boolean v : result.values()) {
      if (v) {
        value++;
      }
    }

    Objectives objectives = new Objectives();
    objectives.add(ones, value);

    return objectives;
  }

  public List<Objective> getObjectives() {
    return Arrays.asList(ones);
  }
}
```

To use the SAT-Decoding, we have to derive our *Decoder* from the *AbstractSATDecoder*. In fact, the *AbstractSATDecoder* is a *Decoder* and *Creator* in a single class such that we only have to specify the constraints and the rules how a feasible solution of these constraints is translated into our *Phenotype*. The ***MinOnesDecoder*** is derived from the *AbstractSATDecoder* with the generic values *Genotype* (which we don't have to care about since we want the *AbstractSATDecoder* to handle the *Creator* task) and *Result* that is the *Phenotype* of our problem.

MinOnesDecoder.java

```java
public class MinOnesDecoder extends AbstractSATDecoder<Genotype, Result> {

  @Inject
  public MinOnesDecoder(SATManager manager, Random random) {
    super(manager, random);
  }

  // Here you can set the constraints of your problem. In our case we will
  // randomly generate a problem with 3SAT problem (3 literals per clause)
  // with 1000 variables and 1000 clauses. This problem is known to be
  // NP-complete. However, we hope that there exists at least one feasible
  // solution (and with the seed 0 of random it does ;) ).
  @Override
  public void init(Set<Constraint> constraints) {

    Random random = new Random(0);

    for (int i = 0; i < 1000; i++) {
      Clause clause = new Clause();
      HashSet<Integer> vars = new HashSet<Integer>();
      do {
        vars.add(random.nextInt(1000));
      } while (vars.size() < 3);

      for (int n : vars) {
        clause.add(new Literal(n, random.nextBoolean()));
      }

      constraints.add(clause);
    }

  }

  @Override
  public Result convertModel(Model model) {

    Result result = new Result();

    for (int i = 0; i < 1000; i++) {
      if (model.get(i) == null || model.get(i) == false) {
        result.put(i, false);
      } else {
        result.put(i, true);
      }
    }

    return result;
  }

}
```

The constructor takes the *SATManager* and *Rand* which we pass directly into the constructor of the *AbstractSATDecoder*. The `@Inject` Annotation tells *Guice* that this is the main constructor to use. However, since *Guice* handles the construction of the objects, we don't have to care about where to obtain the classes *SATManager* and *Rand*.

The constraints are set within the `init(Set<Constraint> constraints)` method. This method is overridden from the *AbstractSATDecoder* and called once when the *Decoder* is initialized. We generate a random 3-SAT (http://en.wikipedia.org/wiki/3SAT) problem with 1000 constraints from 1000 variables.

The `convertModel(Model model)` method is overridden from the *AbstractSATDecoder* and responsible for converting the feasible solutions (*Models*) into the *Phenotype* or *Result*, respectively. Note that the *Model* can contain a *null* for a variable. A *null* indicates that this is a don't care variable under the current solution and we can set this variable to *false* since we are trying to minimize the *1s*.

Finally, we have to write a ***MinOnesModule*** for this *Problem*, cf. Tutorial 1.

MinOnesModule.java

```
public class MinOnesModule extends ProblemModule {

  public void configure() {
    bindProblem(MinOnesDecoder.class, MinOnesDecoder.class,
        MinOnesEvaluator.class);
  }
}
```

Since the *MinOnesDecoder* is a *Decoder* and *Creator* in one class, we first bind it as a Singleton and then bind the *Creator* and *Decoder* to the *MinOnesDecoder* that is now the same object for both interfaces.

6 Optimizer Tutorial

This Tutorial shows how to write an *Optimizer* that is based on *mutation*. The presented *Optimizer* has a population size of 100 and creates 25 offspring *Individuals* from 25 parents *Individuals* each generation by a *Mutate* operation. Afterwards, the worst *Individuals* are sorted out by a *Selector*. Fortunately, OPT4J already offers the *Mutate* operator as well as the *Selector*, making this *Optimizer* quite simple to implement.

We recommend to derive each *Optimizer* from the *AbstractOptimizer* class. Additional to the classes that the *AbstractOptimizer* needs, our ***MutateOptimizer*** needs a *Copy* and *Mutate* operator as well as a *Selector* and the number of iterations.

MutateOptimizer.java

```java
public class MutateOptimizer extends AbstractOptimizer {

  protected final Mutate mutate;

  protected final Copy copy;

  protected final Selector selector;

  @Retention(RUNTIME)
  @BindingAnnotation
  protected @interface Iterations {
  }

  protected final int iterations;

  public static final int POPSIZE = 100;

  public static final int OFFSIZE = 25;

  @Inject
  public MutateOptimizer(Population population, Archive archive,
      IndividualBuilder individualBuilder, Completer completer,
      Control control, Selector selector, Mutate mutate, Copy copy,
      @Iterations int iterations) {
    super(population, archive, individualBuilder, completer, control);
    this.mutate = mutate;
    this.copy = copy;
    this.selector = selector;
    this.iterations = iterations;
  }

  public void optimize() throws TerminationException, StopException {
    selector.init(OFFSIZE + POPSIZE);

    for (int i = 0; i < 100; i++) {
      population.add(individualBuilder.build());
    }

    nextIteration();

    for (int i = 0; i < iterations; i++) {
      Collection<Individual> parents = selector.getParents(OFFSIZE,
          population);

      for (Individual parent : parents) {
        Genotype genotype = copy.copy(parent.getGenotype());
        mutate.mutate(genotype);

        Individual child = individualBuilder.build(genotype);
        population.add(child);
      }

      completer.complete(population);

      Collection<Individual> lames = selector.getLames(OFFSIZE,
          population);
      population.removeAll(lames);

      nextIteration();

    }

  }

}
```

The constructor expects all needed objects for the *AbstractOptimizer* and our *Optimizer*. Our *MutateOptimizer* needs the *Copy* and *Mutate* operator, the *Selector*, and the number of *iterations*. The `int iterations` is annotated with the user defined Annotation `@Iterations` to allow *Guice* to bind this constant value (also see the following *Module*).

The optimization is done in the `optimize()` method. The Selector is initialized with the maximum size of 125. Initially, the *Population* is filled with 100 randomly generated *Individuals*. After creating this initial *Population*, we can indicate that one iteration has passed by calling the method `nextIteration()` implemented by the *AbstractOptimizer*. The method `nextIteration()` automatically evaluates all *Individuals* and updates the *Archive*.

The following optimization process takes place in the for-loop. The *Selector* selects the parent *Individuals*. The *Genotype* of each *Individual* is copied and mutated. A new offspring *Individual* is built by the *IndividualBuilder* and added to the *Population*. Now, we call the *Completer* `complete()` method to evaluate all *Individuals*. The current size of the *Population* is 125, thus, 25 low quality *Individuals* have to be removed from the *Population*. This is done by the truncate method of the *Selector*. Finally, the method `nextIteration()` from the *AbstractOptimizer* has to be called for each completed iteration.

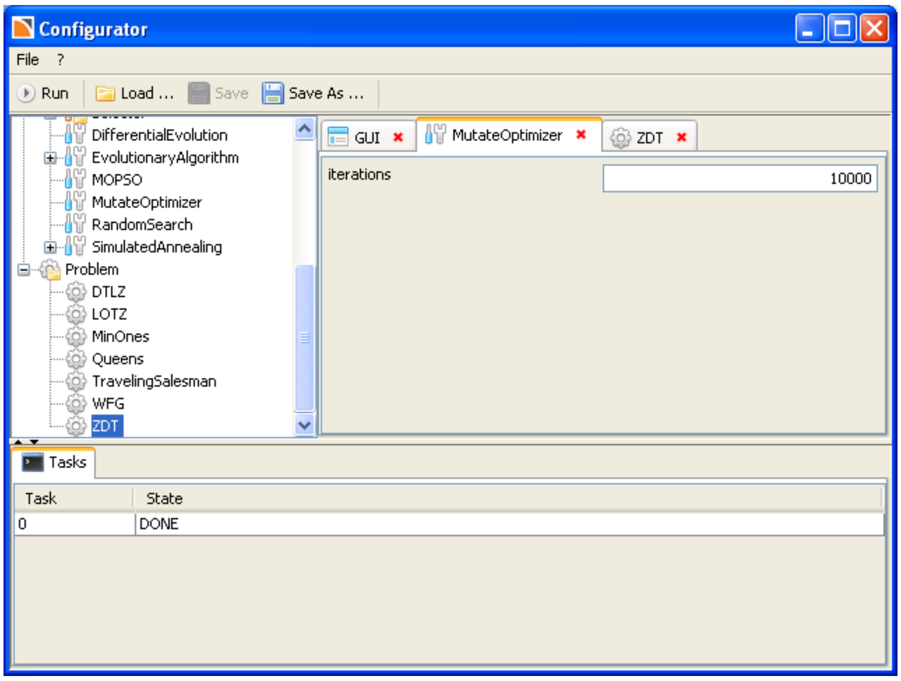
Next, we show how to write a *Module* for the *MutateOptimizer*. The ***MutateOptimizerModule*** has to be derived from the *PropertyModule* and it is recommended to implement the *OptimizerModule* to indicate that we bind an *Optimizer*.

MutateOptimizerModule.java

```
public class MutateOptimizerModule extends OptimizerModule {

  protected int iterations = 1000;
  public int getIterations() {
    return iterations;
  }

  public void setIterations(int iterations) {
    this.iterations = iterations;
  }

  public void configure() {

    bindOptimizer(MutateOptimizer.class);
    bindConstant(Iterations.class).to(iterations);

  }

}
```

The *Optimizer* is bound to our *MutateOptimizer* in the *Singleton Scope* since we only want a single instance of the *Optimizer*. Now, the *MutateOptimizer* has a constant in the constructor, the *iterations*. The *Module* itself has an `int iterations` property with the corresponding getter and setter method. The `iterations` value is bound as constant and *Guice* is informed about the Annotation `Iterations` such that this value is correct bound in the constructor of the *MutateOptimizer*.

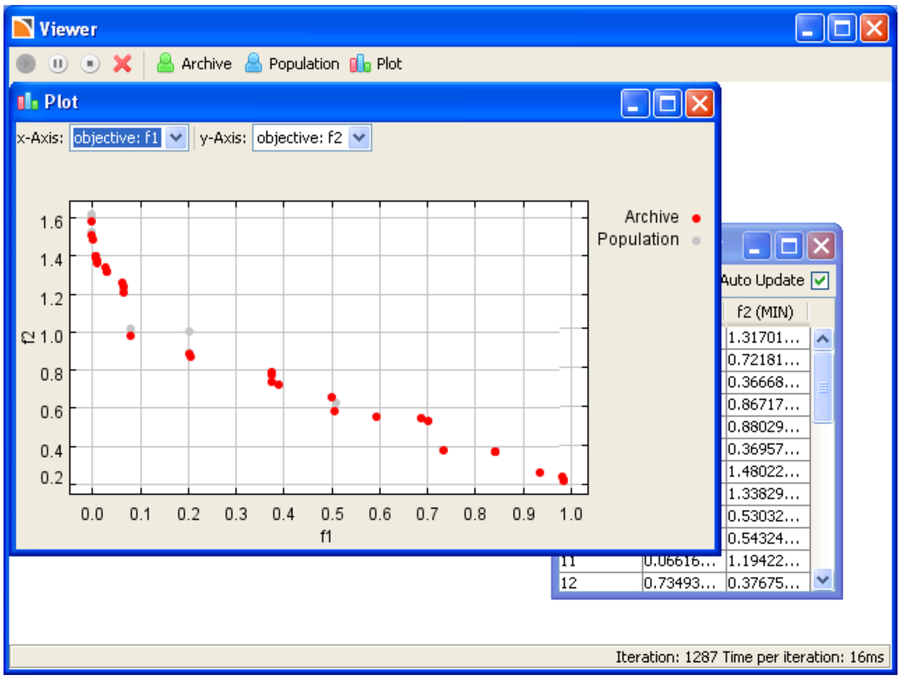
Now, we can test our new *Optimizer Module*:



iterations 10000

Task	State
0	DONE

And after starting the optimization:



x-Axis: objective: f1 y-Axis: objective: f2

Auto Update ✓

		f2 (MIN)
		1.31701...
		0.72181...
		0.36668...
		0.86717...
		0.88029...
		0.36957...
		1.48022...
		1.33829...
		0.53032...
		0.54324...
11	0.06616...	1.19422...
12	0.73493...	0.37675...

Iteration: 1287 Time per iteration: 16ms

7 Integration

This section describes how OPT4J can be embedded into other programs by starting it from the command line or directly from Java.

7.1 Start Optimization from Command Line

To start an optimization without the configurator directly from the command you need a valid configuration XML file. A configuration file can be created with the configurator by saving a configuration. In order to start a configuration `config.xml` directly from the command line, you simply use to `-s` option: `java -jar opt4j-1.5.jar -s config.xml`.

7.2 Start Optimization from Java

The following code snippet shows how to optimize the first DTLZ test function with an Evolutionary Algorithm from Java and afterwards to obtain the best found solutions:

```java
EvolutionaryAlgorithmModule ea = new EvolutionaryAlgorithmModule();
ea.setGenerations(500);
ea.setAlpha(100);

DTLZModule dtlz = new DTLZModule();
dtlz.setFunction(DTLZModule.Function.DTLZ1);

GUIModule gui = new GUIModule();
gui.setCloseOnStop(true);

Collection<Module> modules = new ArrayList<Module>();
modules.add(ea);
modules.add(dtlz);
modules.add(gui);

Opt4JTask task = new Opt4JTask(false);
task.init(modules);

try {
  task.execute();
  Archive archive = task.getInstance(Archive.class);

  for(Individual individual: archive){
    //...
  }

} catch (Exception e) {
  e.printStackTrace();
} finally {
  task.close();
}
```

First, you have to fill a collection with the desired modules for the optimization. The `Opt4JTask` is constructed with the parameter **`false`** that indicates that the task is closed manually and not automatically once the optimization stops. The `Opt4JTask` is initialized with the modules. Now, it is possible to execute the task (which is blocking in this case, but you can also start it in a separate thread) and once the optimization is finished you can obtain the `Archive` to iterate over the best solutions. After closing the task, obtaining instances like the `Archive` is not possible anymore.