EvoHyp: A Java Library for Evolutionary Algorithm Hyper-Heuristics Version 1.0

Nelishia Pillay Derrick Beckedahl

March 16, 2017

Contents

Chapter 1

Introduction

EvoHyp is a Java library for evolutionary algorithm hyper-heuristics. It contains the following packages:

- GenAlg - This is a genetic algorithm package that can be used to implement selection constructive and selection perturbative hyper-heuristics.
- DistrGenAlg - This package allows for the genetic algorithm to be distributed over a multicore architecture to reduce the runtime of the hyper-heuristic.
- GenProg - This package provides a genetic programming hyper-heuristic to create new low-level construction heuristics.
- DistrGenProg - This class allows for the genetic programming generation hyper-heuristic to be distributed over a multicore architecture.

The packages are described in the chapters that follow.

Chapter 2

GenAlg Package

This package implements a genetic algorithm selection hyper-heuristic. The package can be used to implement selection constructive or selection perturbative hyper-heuristics. In order to use this package the user has to implement two abstract classes, namely, *InitialSoln* and *ProblemDomain* which form an interface between *GenAlg* and the problem domain. The chapter firstly presents both these classes. The chapter then provides an overview of the genetic algorithm selection hyper-heuristic employed. It then shows how the package can be used to implement a selection constructive and selection perturbative hyper-heuristic.

2.1 *InitialSoln* Class

This is an abstract class which must be implemented by the user. The aim of the class is to create a solution to the problem using a heuristic combination. The class has the following two methods which each concrete implementation of the class must include:

`abstract int fitter(InitialSoln other)`
This method compares the performance of the current *InitialSoln* instance with *other*. A value of 1 is returned if the instance is fitter than *other* and if *other* is fitter a -1 is returned. If both solutions have the same fitness a 0 is returned. The purpose of this method is to allow the user to define the problem specific fitness which depending on the problem domain could be maximized or minimized.

`abstract double getFitness()`
This method returns the *fitness* of the chromosome. The fitness is the objective value of the solution created using the chromosome or a function of the objective value. For example, for the examination timetabling problem the fitness could be the sum of the number of hard constraint and soft con-

straint violations. For the travelling salesman problem the fitness could be the cost of the tour constructed using the chromosome.

`abstract Object getSoln()`
This method returns the solution created using the evolved chromosome. For example, in the case of the travelling salesman problem this could be a string representing a tour, however for the examination timetabling problem this could be an array of examination records storing the period and venue for each examination. The type of the solution is dependent on the problem being solved.

`abstract void setHeuCom(String heuCom)`
This method stores the chromosome, i.e. heuristic combination used to solve the problem. The heuristic combination is a string consisting of characters representing the low-level heuristics.

`abstract String getHeuCom()`
This method returns the chromosome, i.e. heuristic combination, used to construct the solution.

2.2 *ProblemDomain* Class

This is an abstract class which must be implemented by the class for the problem that the hyper-heuristic will be used to solve. The *GenAlg* and *DistrGenAlg* package will access the method *evaluate* via an instance of the concrete implementation of class *ProblemDomain*. The following abstract method must be implemented in the concrete class:

`abstract InitialSoln evaluate(String heuristicComb)`
This method takes the heuristic combination `heuristicComb` evolved by the genetic algorithm in the package *GenAlg* or *DistrGenAlg* as input. The string specified in `heuristicComb` is used to construct a solution to the problem and returns an object of type *InitialSoln*. An instance of the type *InitialSoln* combines the *objective value* or *function of the objective value* that will be used as a fitness value for each chromosome by the genetic algorithm and the solution created using the evolved string of heuristics.

2.3 Genetic Algorithm Hyper-Heuristc

The *GenAlg* package by default implements the generational genetic Algorithm in depicted in algorithm 1.

The algorithm begins by creating an initial population of chromosomes. Each chromosome is a string of characters representing the low-level heuris-

Algorithm 1 Generational genetic algorithm

```
1: Create an initial population
2: repeat
3:     Evaluate the population
4:     Select parents
5:     Apply genetic operators
6: until termination criterion is met
```

tics. Each element of the population is of type *InitialSoln*. The chromosome is stored as part of the individual instance and when the chromosome is evaluated the fitness and solution created using the heuristic combination is added to the instance. The chromosomes are evaluated using the function *evaluate* provided by an instance of the *ProblemDomain* class created by the user to calculate the fitness of each chromosome. Tournament selection is used to select parents to apply genetic operators to. It is invoked each time a parent is needed by a genetic operator. The mutation and crossover operators are used to produce the offspring of each generation. The evolutionary process is terminated when the maximum number of generations has been reached.

2.4 Constructors

`GenAlg(long seed, String heuristics)`
The constructor must be passed a seed `seed` for the random number generator and a string `heuristics` comprised of characters representing each of the low-level heuristics.

2.5 Methods

There are two sets of methods available, one to set the parameter values for the genetic algorithm and perform parameter tuning and the second to invoke the genetic algorithm hyper-heuristic. These are discussed in the following sections.

2.5.1 Methods for Setting Parameters

The parameters for the genetic algorithm implemented by the hyper-heuristic can be set by invoking methods to set this individually or by suppling the parameter values in a text file. The following methods can be used to set the parameter values individually:

`void setPopulationSize(int populationSize)`

Sets the population size for the genetic algorithm.

`void setTournamentSize(int tournamentSize)`
Sets the tournament size for the genetic algorithm.

`void setNoOfGenerations(int noOfGenerations)`
Sets the number of generations for the genetic algorithm.

`void setMutationRate(double mutationRate)`
Sets the mutation rate for the genetic algorithm. The rate specifies the percentage of the next generation that will be created using the mutation operator. If the sum of the mutation and crossover rate equals a 100% the reproduction operator is not used. If the sum is not a 100% the remaining percentage of the population is created using reproduction.

`void setCrossoverRate(double crossoverRate)`
Sets the crossover rate for the genetic algorithm. If the sum of the mutation and crossover rate equals a 100% the reproduction operator is not used. If the sum is not a 100% the remaining percentage of the population is created using reproduction.

`void setInitialMaxLength(int initialMaxLength)`
Sets the maximum length of the chromosomes created during initial population generation. The length of chromosomes are randomly selected to be between 1 and this preset maximum length.

`void setOffspringMaxLength(int offspringMaxLength)`

Sets the maximum length of offspring produced by genetic operators. If the offspring produced by a genetic operator exceeds this length the chromosome is pruned to be the substring equal to this maximum length. If there is no limit on the size of offspring produced by the genetic operators set this value to -1.

`void setmutationLength(int mutationLength)`

Sets the mutation length. The mutation operator inserts a substring into the copy of a parent. The length of the substring is randomly selected to be between 1 and the *mutationLength*.

Alternatively, the parameter values can be specified in a text file using the following method:

`void setParameters(String parameterFile)`
Allows the user to set parameter for the genetic algorithm using a text file. The parameters must be specified in the following order, one per line:

populationSize
tournamentSize
noOfGenerations
mutationRate
crossoverRate
initialMaxLength
offspringMaxLength
mutationLength

The following methods can be used to get the parameter values:

`int getPopulationSize()`
Returns the population size.

`int getTournamentSize()`
Returns the tournament size.

`int getnoOfGenerations()`
Returns the number of generations.

`idouble getMutationRate()`
Returns the mutation rate.

`double getCrossoverRate()`
Returns the crossover rate.

`double getReproductionRate()`
Returns the reproduction rate.

`int getInitialMaxLength()`
Returns the maximum chromosome length for initial population generation.

`int getOffspringMaxLength()`
Returns the maximum offspring length for offspring produced by the genetic operators.

2.5.2 Methods for Setting Problem Domain Information

The problem domain specific knowledge that is needed by the hyper-heuristic are the characters representing the low-level heuristics for the problem domain and an instance of the class implementing the *ProblemDomain* abstract class that will create a solution using the evolved heuristic combinations and calculates the objective value of the solution and the fitness of the chromosome. Two *set* methods are used to store this information:

`void setHeuristics(String heuristics)`
This method stores the string comprised of characters, representing the low-level heuristics for the problem domain.

`void setProblem(ProblemDomain problem)`
This method stores an instance of the class implementing the *ProblemDomain* abstract class which will be used to create a solution to the problem using the evolved heuristic combination and calculate the objective value and fitness of the chromosome.

2.5.3 Methods for the Genetic Algorithm Hyper-Heuristic

The genetic algorithm hyper-heuristic is invoked using the following method:

```
InitialSoln evolve()
```

This method implements the generational genetic algorithm depicted in section 2.3. The algorithm evolves a population of heuristic combinations over a set number of generations. Tournament selection is used to choose parents. The *mutation* and *crossover* operators are used to create the offspring of each generation. The mutation operator replaces the heuristic at the mutation point with a newly created substring of heuristics. The substring is created by randomly selecting heuristics from the set provided. The crossover operator randomly selects crossover points in a copy of each of the parents. The parents are crossed over at these points producing two offspring of variable length.

2.6 Implementing a Selection Hyper-Heuristic

The *GenAlg* package contains three *jar* files that must be included as libraries in the project using *GenAlg* to implement a selection hyper-heuristic:

- *Problemdomain* - an abstract class which must be extended to define the problem domain.

- *InitialSoln* - an abstract class which must be extended to define an initial solution for the problem domain.
- *GenAlg* - the class implementing a genetic algorithm selection hyper-heuristic.

An example project *SolveProblem* illustrates how *EvoHyp* can be used to solve a combinatorial optimization problem. In order to use the *GenAlg* class to create an initial solution to a combinatorial optimization problem the user must:

- Implement a class to extend the *ProblemDomain* which implements an *evaluate* method to create an instance of the class extended *InitialSoln*. This is the *ComOptProb* class in the *SolveProblem* project.
- Implement a class that extends the *InitialSoln* class. This class should create a solution to the problem using a heuristic combination passed to the class. The function to calculate the objective value and fitness of the solution created using the heuristic combination should be included in this class. This class is *ComOptSoln* in the *SolveProblem* project. In creating a solution to the problem using the heuristic combination, the low-level heuristics must be implemented by the user.

The *ComOptProb* class is listed below:

```
package solveproblem;

//Import statements
import problemdomain.*;
import initialsoln.*;

public class ComOptProb extends ProblemDomain
{
 public ComOptSoln evaluate(String heuristicComb)
 {
   ComOptSoln soln = new ComOptSoln();
   soln.setHeuCom(heuristicComb);
   soln.createSoln();

   return soln;
  }
}
```

The `ComOptProb` class `extends` the abstract class `ProblemDomain`. The abstract method `evaluate` must be implemented. The concrete implementation of the *InitialSoln* class, `ComOptSoln` implements a method, `createSoln`

to create a solution to the problem using the heuristic combination. The `evaluate` class creates an instance of `ComOptSoln` and passes it the heuristic combination. The solution created using the heuristic combination and the corresponding fitness is returned in the instance `soln`. The *ComOptSoln* class is listed below:

```
package solveproblem;

//Import statements
import initialsoln.*;

public class ComOptSoln extends InitialSoln
{
 //Data elements

 private String heuristicComb;
 private double fitness;
 String initSoln[];

//Methods

 public double getFitness()
 {
   return fitness;
 }

 public void setHeuCom(String heuristicComb)
 {
   this.heuristicComb=heuristicComb;
 }

public String getHeuCom()
{
  return heuristicComb;
}

public String[] getSoln()
{
    return initSoln;
}

 public int fitter(InitialSoln other)
 {
   if(fitness < other.getFitness())
```

```
    return 1;
   else if (fitness > other.getFitness() )
    return -1;
   else
    return 0;
 }

 public void createSoln()
 {

   String temp[]={"This"," is"," a"," solution",
                  " created"," using ", heuristicComb};
   initSoln=temp;

   fitness=temp.length+Math.random();
 }
}
```

The `createSoln` method creates a solution to the problem using the heuristic combination `heuristicComb`. In the example this is a simple method which creates a *String* array to indicate that a solution has been created using the heuristic combination. The solution created can be of any type, in this case it is a *String* array.

The `fitter` method is used by the *GenAlg* class to compare to elements of the population to see which is fitter. Thus, the method takes in another instance of type *InitialSoln* and compares its fitness with that of the current instance. In the example a lower fitness indicates a fitter individual. Hence, a 1 is returned in the case that the fitness of the current instance is less than the other instance, indicating that the current instance is fitter.

The following code illustrates how the genetic algorithm selection hyper-heuristic implemented in *GenAlg* can be evoked to solve a combinatorial optimization problem:

```
package solveproblem;

//Import statements
import genalg.*;
import initialsoln.*;


public class SolveProblem
{
 static public void solve()
 {
  ComOptProb problem = new ComOptProb();
```

```
  long seed = System.currentTimeMillis();
  String heuristics=new String("abc");
  GenAlg schh = new GenAlg(seed,heuristics);
  schh.setParameters("Parameters.txt");
  schh.setProblem(problem);
  InitialSoln solution= schh.evolve();

  System.out.println("Best Solution");
  System.out.println("---------------");
  System.out.println("Fitness: "+solution.getFitness());
  System.out.println("Heuristic combination: "
                                  +solution.getHeuCom());
  System.out.println("Solution: ");
  displaySolution(solution.getSoln());
  }
}
```

An instance of the concrete implementation of *ProblemDomain* is firstly created. An instantiation of *GenAlg* takes a seed for the random number generator and a string of characters representing each of the heuristics as input. The parameters for the genetic algorithm are then set using the `setParameters` method. An instance of the problem domain is passed to the instance of the *GenAlg* class. The method `evolve` implements the selection hyper-heuristic to solve the combinatorial optimization problem. The `evolve` method returns an instance of the type *InitialSolution* which contains the best performing heuristic combination, its fitness and the solution created using the heuristic combination.

In this way a selection constructive hyper-heuristic or a selection perturbative hyper-heuristic can be implemented. The only difference in the implementation for both hyper-heuristics is the `createSoln` method implemented by the user in the concrete implementation of the *InitialSoln* class. In the case of a selection constructive hyper-heuristic this method must use the heuristic combination to create a solution. The low-level heuristics must also be implemented by the user. In the case of a selection perturbative hyper-heuristic, the user must implement methods to create an initial solution to the problem and improve the solution using the heuristic combination.

Chapter 3

GenProg Package

This chapter describes the *GenProg* library which implements a generation constructive hyper-heuristic. This package can be used to create constructive heuristics for combinatorial optimization problems. The heuristics are either *arithmetic functions* or *arithmetic rules*. If both cases the evolved heuristic evaluates to a numerical value. In constructing a solution the heuristic value can be calculated for each entity to be allocated to create a solution and these can be allocated in some order depending on the heuristic value. The problem domain must be implemented by the user using the *Solution* and *Problem* classes. The chapter firstly describes these classes. Details of the genetic programming generation constructive hyper-heuristic are then provided followed by an overview of the *Evaluator* library that provides an interpreter for the evolved heuristics which can be used to calculate the value of the heuristic.

3.1 *Solution* Class

This is an abstract class which must be implemented by the user. The aim of the class is to create a solution to the problem using the heuristic. The class has the following two methods which each concrete implementation of the class must include:

`abstract int fitter(Solution other)`
This method compares the performance of the current *Solution* instance with *other*. A value of 1 is returned if the instance is fitter than *other* and if *other* is fitter a -1 is returned. If both solutions have the same fitness a 0 is returned. The purpose of this method is to allow the user to define the problem specific fitness which depending on the problem domain could be maximized or minimized.

`abstract double getFitness()`

This method returns the *fitness* of the chromosome. The fitness is the objective value of the solution created using the chromosome or a function of the objective value. For example, for the examination timetabling problem the fitness could be the sum of the number of hard constraint and soft constraint violations. For the travelling salesman problem the fitness could be the cost of the tour constructed using the chromosome.

`abstract Object getSoln()`
This method returns the solution created using the evolved heuristic. For example, in the case of the travelling salesman problem this could be a string representing a tour, however for the examination timetabling problem this could be an array of examination records storing the period and venue for each examination. The type of the solution is dependent on the problem being solved.

`abstract void setHeuristic(Other heu)`
This method stores the evolved heuristic to be used to solve the problem. The heuristic can be an arithmetic function or rule. The heuristic is defined to be of type *Object* so it can be of any type. In the *GenProg* library the evolved heuristics are defined to be of type *Node*.

`abstract Object getHeuristic()`
This method returns the heuristic used to construct the solution.

3.2 *Problem* Class

This is an abstract class which must be implemented by the class for the problem that the hyper-heuristic will be used to create a heuristic for. The *GenProg* and *DistrGenprog* package will access the method *evaluate* via an instance of the concrete implementation of class *Problem*. The following abstract method must be implemented in the concrete class:

`abstract Solution evaluate(Object heuristic)`
This method takes the heuristic `heuristic` evolved by the genetic programming algorithm in the package *GenProg* as input. The heuristic `heuristic` is used to construct a solution to the problem and returns an object of type *Solution*. The instances of the type *Solution* combines the *objective value* or *function of the objective value* that will be used as a fitness value for each parse tree by the genetic programming algorithm and the solution created using the heuristic.

`abstract void setAttribs(String attribs)`
This method is an abstract method which must be implemented by the

user to specify the problem attributes that should be used in the heuristic. These are specified as a string of characters, `attribs`, with each character representing a problem attribute.

3.3 Genetic Programming Hyper-Heuristic

The *GenProg* package implements a genetic programming generation constructive hyper-heuristic to create constructive heuristics for combinatorial optimization problems. The genetic programming algorithm is the generational algorithm depicted in algorithm 1.

The evolved heuristics can be arithmetic functions or arithmetic rules. The terminal set is essentially the set of characters provided by the user representing problem attributes. In the case of arithmetic functions the function set consists of the standard arithmetic operators, namely, addition, subtraction, multiplication and division. The division operator is a protected operator which returns a value of 1 if the denominator is 0. In the case of arithmetic rules the function set also includes an *if-then-else* operator and relational operators, $<, >, <=, >=, ==, != $.

Each element of the population is a parse tree representing a heuristic. The *grow* method is used to create each element of the population by randomly selecting elements from the function and terminal sets. The root of each tree is a function. At the maximum depth permitted the elements from the terminal set are selected. At all other levels of the tree elements are selected from both the function and terminal sets. A parse tree is of type *Node*. Each element of the population is of type *Solution* and is an instance of the concrete implementation of this abstract class. The instance stores the heuristic, its fitness and the solution created using the heuristic.

Each element of the population is evaluated by using it to create a solution. The `evaluate` method in the instance of the concrete implementation of the *Problem* class is evoked for this purpose. The corresponding solution and fitness is stored in the instance representing the element of the population. Tournament selection is used to select parents which the genetic operators are applied to. It is invoked each time a genetic operator needs a parent.

The standard genetic programming reproduction, mutation and crossover operators are used to create the offspring of each generation. The mutation operator randomly selects a mutation point in an individual and the subtree rooted at that point is replaced with a randomly created subtree. Crossover selects a crossover point in each of the parents and the subtrees rooted at these points are swapped.

3.4 *Evaluator* Class

The package includes the class *Evaluator* to interpret heuristics created by *GenProg*. In the concrete implementation of the abstract class *Solution* the user will define a method to create a solution to the problem using the heuristic. The heuristic of type *Node* is passed from the *GenProg* class to the problem domain implementation via the `evaluate` method defined for the problem domain. The method to create a solution must then create a solution using the heuristic. The heuristic has to be interpreted using the current values of the attributes. The *Evaluator* class provides for this via the following methods:

`Evaluator(String attributes,double attributeVals[])`
This is the constructor which takes a string comprised of characters representing each of the problem attributes `attributes` and the corresponding values `attributeVals` as input.

`double eval(Node op)`
This method evaluates the heuristic `op` using the values in the array `attributeVals`. The numerical value that is calculated by the heuristic is returned.

3.5 Constructors

`GenProg(long seed,String attributes,int heuType)`
The constructor must be passed a seed `seed` for the random number generator, a string `attributes` comprised of characters representing each of the low-level heuristics and an integer flag *heuType* indicating whether an arithmetic function or rule should be evolved. A value of 0 for *heuType* indicate that a function should be evolved and a 1 that an arithmetic rule should be evolved.

3.6 Methods

There are two sets of methods available, one to set the parameter values for the genetic programming algorithm and perform parameter tuning and the second to invoke the genetic programming hyper-heuristic. These are discussed in the following sections.

3.6.1 Methods for Setting Parameters

The parameters for the genetic programming algorithm implemented by the hyper-heuristic can be set by invoking methods to set this individually or

by suppling the parameter values in a text file. The following methods can be used to set the parameter values individually:

`void setPopulationSize(int populationSize)`
Sets the population size for the genetic programming algorithm.

`void setTournamentSize(int tournamentSize)`
Sets the tournament size for the genetic programming algorithm.

`void setNoOfGenerations(int noOfGenerations)`
Sets the number of generations for the genetic programming algorithm.

`void setMutationRate(double mutationRate)`
Sets the mutation rate for the genetic programming algorithm. The rate specifies the percentage of the next generation that will be created using the mutation operator. If the sum of the mutation and crossover rate equals a 100% the reproduction operator is not used. If the sum is not a 100% the remaining percentage of the population is created using reproduction.

`void setCrossoverRate(double crossoverRate)`
Sets the crossover rate for the genetic algorithm. If the sum of the mutation and crossover rate equals a 100% the reproduction operator is not used. If the sum is not a 100% the remaining percentage of the population is created using reproduction.

`void setMaxDepth(int maxDepth)`
Sets the maximum depth of the parse trees created during initial population generation.

`void setOffspringDepth(int offspringDepth)`

Sets the maximum depth of offspring produced by genetic operators. If the offspring produced by a genetic operator exceeds this depth the parse tree is pruned by replacing the function nodes at the maximum depth by terminals. If there is no limit on the size of offspring produced by the genetic operators set this value to -1.

`void setmutationDepth(int mutationDepth)`

Sets the maximum depth of the new subtree created by the mutation operator to replace the subtree at the mutation point.

Alternatively, the parameter values can be specified in a text file using the following method:

`void setParameters(String parameterFile)`
Allows the user to set parameters for the genetic programming algorithm using a text file. The parameters must be specified in the following order, one per line:

populationSize
tournamentSize
noOfGenerations
mutationRate
crossoverRate
maxDepth
offspringDepth
mutationDepth

The following methods can be used to get the parameter values:

`int getPopulationSize()`
Returns the population size.

`int getTournamentSize()`
Returns the tournament size.

`int getnoOfGenerations()`
Returns the number of generations.

`idouble getMutationRate()`
Returns the mutation rate.

`double getCrossoverRate()`
Returns the crossover rate.

`double getReproductionRate()`
Returns the reproduction rate.

`int getMaxDepth()`
Returns the maximum chromosome length for initial population generation.

`int getOffspringDepth()`
Returns the maximum offspring length for offspring produced by the genetic operators.

`int getMutationDepth()`
Returns the maximum mutation depth for subtrees created by the mutation

operator.

3.6.2 Methods for Setting Problem Domain Information

The problem domain specific knowledge that is needed by the hyper-heuristic is an instance of the class implementing the *ProblemDomain* abstract class that will create a solution using the evolved heuristic and calculate the objective value of the solution and the fitness of the parse tree. The following method is used to store this information:

`void setProblem(Problem problem)`
This method stores an instance of the class implementing the *Problem* abstract class which will be used to create a solution to the problem using the evolved heuristic and calculate the fitness of the parse tree.

3.6.3 Methods for the Genetic Programming Hyper-Heuristic

The genetic programming hyper-heuristic is invoked using the following method:

`Solution evolve()`
This method implements the generational genetic programming algorithm depicted in section 3.3. The algorithm evolves a population of heuristics over a set number of generations. Tournament selection is used to choose parents. The *reproduction*, *mutation* and *crossover* operators are used to create the offspring of each generation.

3.7 Implementing a Generation Constructive Hyper-Heuristic

The *GenProg* package contains three *jar* files that must be included as libraries in the project using *GenProg* to implement a generation constructive hyper-heuristic. These libraries provide the following classes:

- *Problem* - an abstract class which must be extended to define the problem domain.
- *Solution* - an abstract class which must be extended to define an initial solution for the problem domain.
- *GenProg* - the class implementing a genetic programming algorithm generation constructive hyper-heuristic.

- *Evaluator* - this class provides an interpreter to interpret the heuristic evolved by *GenProg* given the attribute values.

An example project *CreateHeuristic* illustrates how *GenProg* can be used to create a heuristic for a combinatorial optimization problem. In order to use the *GenProg* class to induce a heuristic for a combinatorial optimization problem the user must:

- Implement a class to extend the *Problem* which implements an *evaluate* method to create an instance of the class extending *Solution*. This is the *ComOptProb* class in the *CreateHeuristic* project.
- Implement a class that extends the *Solution* class. This class should create a solution to the problem using a heuristic combination passed to the class. The function to calculate the objective value and fitness of the solution created using the heuristic combination should be included in this class. This class is *ComOptSoln* in the *CreateHeuristic* project.

The *ComOptProb* class is listed below:

```
package createheuristic;

import solution.*;
import problem.*;

public class ComOptProb extends Problem
{
    //Data elements

    private String attributes;

    //Methods

    public Solution evaluate(Object heuristic)
    {
      ComOptSoln soln = new ComOptSoln();
      soln.setHeuristic(heuristic);
      soln.createSoln(attributes);

     return soln;
   }
}
```

The `ComOptProb` class `extends` the abstract class `Problem`. The abstract method `evaluate` must be implemented. The concrete implementation of the *Solution* class, `ComOptSoln` implements a method, `createSoln` to create

a solution to the problem using the heuristic. The `evaluate` method creates an instance of `ComOptSoln` and passes it the heuristic combination. The solution created using the heuristic combination and the corresponding fitness is returned in the instance `soln`. The *ComOptSoln* class is listed below:

```
package createheuristic;

//Import statements
import genprog.*;
import java.util.Arrays;
import solution.*;
import java.util.Random;
import java.util.ArrayList;

public class ComOptSoln extends Solution
{
  //Data elements

  private Entity entities[];
  private double fitness;
  private Node heuristic;
  private ArrayList soln;
  private String attributes;

 public ComOptSoln()
 {
   initEntities();
 }

 public void setHeuristic(Object heuristic)
 {

    this.heuristic=(Node)heuristic;
 }

 public Object getHeuristic()
 {

   return heuristic;
 }
```

```
public Object getSoln()
{

   return soln;
}

public double getFitness()
{

  return fitness;

}

public int fitter(Solution other)
{
  if(other.getFitness() < fitness)
   return -1;
  else if (other.getFitness() > fitness)
   return 1;
  else
   return 0;
}

public void initEntities()
{

  Random ranGen= new Random();

  entities=new Entity[3];

  for(int count=0; count < 3;++count)
  {
    entities[count]=new Entity();
    double attribs[]={ranGen.nextInt(10)+1,ranGen.nextInt(10)+1,
                      ranGen.nextInt(10)+1};
    entities[count].setAttribs(attribs);
    attribs=null;
  }//endForCount
}
```

```
private void calcHeuristics()
{

  for(int count=0;count < entities.length;++count)
  {
   if(entities[count].getHeuristic()!=Double.MAX_VALUE)
   {
    Evaluator eval = new Evaluator(attributes,
                                   entities[count].getAttribs());
    entities[count].setHeuristic(eval.eval(heuristic));
   }
  }//endForCount

}

public void createSoln(String attributes)
{

  this.attributes=attributes;
  calcHeuristics();
  Arrays.sort(entities);
  while(entities[0].getHeuristic() != Double.MAX_VALUE)
  {

   if(soln==null)
    soln=new ArrayList();

   soln.add(entities[0]);
   updateEntities();
   calcHeuristics();
   Arrays.sort(entities);

  }//endwhile

  Entity entity = (Entity) soln.get(0);
  double a[]=entity.getAttribs();
  fitness=a[0]+a[1]+a[2];
}
```

```
private void updateEntities()
{
    entities[0].setHeuristic(Double.MAX_VALUE);
    for(int count=1;count < entities.length;++count)
    {
      double attribs[]=entities[count].getAttribs();
      --attribs[1];
      entities[count].setAttribs(attribs);

    }//endfor
 }
}
```

The *Entity* class is used to illustrate the use of the *GenProg* library. An instance of this class represents an entity that the evolved heuristic is used to select. For example, in the examination timetabling problem this would be an examination while in the one dimensional bin-packing problem this would be a bin. This class contains a data element that is a double array. A solution is created by sorting the instances of the *Entity* class according to the heuristic value and allocating them in order to create a solution which is essentially an `ArrayList` of instances of the *Entity* class. Each time an entity is allocated the heuristics of the remaining entities are recalculated and the list of entities to be allocated is resorted. As can be seen from the `createSoln` method in creating a solution the heuristic values of the entities are calculated and the list of entities are sorted according to this value. The first entity in the sorted list is allocated to the solution. The heuristic values are recalculated for the remaining entities based on the entity just allocated and the process is repeated until all the entities have been allocated.

The `calcHeuristic` method is used to calculate the heuristic value for each entity. This is done by using the *Evaluator* class provided as part of *GenProg*. The `eval` method calculates the value of the heuristic given the set of attribute values.

The `fitter` method is used by the *GenProg* class to compare to elements of the population to see which is fitter. Thus, the method takes in another instance of type *Solution* and compares its fitness with that of the current instance. In the example a lower fitness indicates a fitter individual. Hence, a 1 is returned in the case that the fitness of the current instance is less than the other instance, indicating that the current instance is fitter.

The following code illustrates how the genetic programming algorithm generation hyper-heuristic implemented in *GenProg* can be evoked to solve a combinatorial optimization problem:

```
package createheuristic;
```

```
//Import statements
import genprog.*;
import Solution.*;
import Problem.*;
import java.util.ArrayList;

public class CreateHeuristic
{
 static public void solve()
 {
  ComOptProb problem = new ComOptProb();
  long seed = System.currentTimeMillis();
  String attribs=new String("abc");
  problem.setAttribs(attribs);
  GenProg gchh = new GenProg(seed,attribs,1);
  gchh.setParameters("Parameters.txt");
  gchh.setProblem(problem);
  ComOptSoln sol= (ComOptSoln)gchh.evolve();

  System.out.println("Best Solution");
  System.out.println("---------------");
  System.out.println("Fitness: "+sol.getFitness());
  System.out.print("Heuristic: ");
  printInd((Node)sol.getHeuristic());
 }

 static private void printInd(Node root)
 {
    System.out.print(root.getLabel());
    for (int count=0; count < root.getArity();++count)
    {
      printInd(root.getChild(count));
    }//endfor_count
```

An instance of the concrete implementation of *Problem* is firstly created. An instantiation of *GenProg* takes a seed for the random number generator, a string of characters representing each of the problem attributes and an integer flag indicating whether an arithmetic function or rules must be evolved as input. In the example an arithmetic rules is required. The parameters for the genetic programming algorithm are then set using the `setParameters` method. An instance of the problem domain is passed to the instance of the *GenProg* class. The method `evolve` implements the generation constructive hyper-heuristic to solve the combinatorial optimization problem.

The `evolve` method returns an instance of the type *Solution* which contains the best performing heuristic, its fitness and the solution created using the heuristic.

Chapter 4

The *DistrGenAlg* and *DistrGenProg* Packages

The distributed versions of *GenAlg* and *GenProg* are *DistrGenAlg* and *DistrGenProg*. These distributed versions distribute the implementation of the generational genetic algorithm and genetic programming algorithm over a multicore architecture as follows:

- The initial population is divided into subpopulations by dividing the population size by the number of available cores. Each subpopulation is created and evaluated on a separate core.
- The regneration process, i.e. selection of parents and creation of offspring, of each generation is distributed over a number of cores. The population to be created is divided into subpopulation by dividing the population size by the number of cores. The offspring for each subpopulation are created and evaluated on separate cores.

DistrGenAlg and *DistrGenProg* are used in the same way as *GenAlg* and *GenProg* in implementing genetic algorithm selection hyper-heuristics and genetic programming generation constructive hyper-heuristics respectively. The only difference is in creating an instance of these classes the number of cores to use must be specified:

`GenAlg(long seed, String heuristics,int noOfCores)`
The constructor must be passed a seed `seed` for the random number generator, a string `heuristics` comprised of characters representing each of the low-level heuristics and the number of cores the algorithm must be distributed over `noOfCores`.

`GenProg(long seed,String attributes,int heuType, int noOfCores)`
The constructor must be passed a seed `seed` for the random number generator, a string `attributes` comprised of characters representing each of the low-level heuristics, an integer flag *heuType* indicating whether a an arithmetic function or rule should be evolved and the number of cores over which the implementation of the algorithm must be distributed `noOfCores`. A value of 0 for *heuType* indicate that a function should be evolved and a 1 that an arithmetic rule should be evolved.

Chapter 5

EvoHyp Website and Contact

The *EvoHyp* website can be accessed via http://titancs.ukzn.ac.za/EvoHyp.aspx. The website provides the following:

- Jar files for the *GenAlg* package:
  - *ProblemDomain* class
  - *InitialSoln* class
  - *GenAlg* class
- Jar files for the *GenProg* package:
  - *Problem* class
  - *Solution* class
  - *GenProg* class
- Jar file for *DistGenAlg* class.
- Jar file for the *DistrGenProg* class.
- Java files for the *SolveProblem* example.
- Java files for the *CreateHeuristic* example.
- Documentation and papers.

If you have any queries, comments or suggestions please contact us at *evohyp@gmail.com*.