EvoHyp - A Java Toolkit for Evolutionary Algorithm Hyper-Heuristics

Nelishia Pillay
School of Mathematics, Statistics and Computer Science
University of KwaZulu-Natal
KwaZulu-Natal, South Africa
Email: pillayn32@ukzn.ac.za

Derrick Beckedahl
School of Mathematics, Statistics and Computer Science
University of KwaZulu-Natal
KwaZulu-Natal, South Africa
Email: d.beckedahl@gmail.com

***Abstract*—Hyper-heuristics is an emergent technology that has proven to be effective at solving real-world problems. The two main categories of hyper-heuristics are *selection* and *generation*. Selection hyper-heuristics select existing low-level heuristics while generation hyper-heuristics create new heuristics. At the inception of the field single point searches were essentially employed by selection hyper-heuristics, however as the field progressed evolutionary algorithms are becoming more prominent. Evolutionary algorithms, namely, genetic programming, have chiefly been used for generation hyper-heuristics. Implementing evolutionary algorithm hyper-heuristics can be quite a time-consuming task which is daunting for first time researchers and practitioners who want to rather focus on the application domain the hyper-heuristic will be applied to which can be quite complex. This paper presents a Java toolkit for the implementation of evolutionary algorithm hyper-heuristics, namely, *EvoHyp*. *EvoHyp* includes libraries for a genetic algorithm selection hyper-heuristic (*GenAlg*), a genetic programming generation hyper-heuristic (*GenProg*), a distributed version of *GenAlg* (*DistrGenAlg*) and a distributed version of *GenProg* (*DistrGenProg*). The paper describes the libraries and illustrates how they can be used. The ultimate aim is to provide a toolkit which a non-expert in evolutionary algorithm hyper-heuristics can use. The paper concludes with an overview of future extensions of the toolkit.**

I. Introduction

Hyper-heuristics [1] provide an alternative approach to solving problems by exploring a heuristic space rather than a solution space which is typical of most optimization techniques. Exploring a heuristic space which maps to a solution space has been shown to be more effective for solving various real-world problems than searching the solution space directly. Hyper-heuristics either *select* low-level heuristics or *generate* new low-level heuristics. The low-level heuristics can be *constructive* or *perturbative*. Constructive heuristics are used to create solutions to a problem while perturbative heuristics are used to improve an existing solution. Hence, the four types of hyper-heuristics are selection constructive, selection perturbative, generation constructive and generation perturbative.

Selection constructive hyper-heuristics select which constructive heuristic to choose at each point in creating a solution to the problem. Initially, single point searches such as tabu search were employed for selection [2] but as the field developed evolutionary algorithms have proven to be effective [3]. Similarly, selection perturbative hyper-heuristics choose which perturbative heuristic to apply at each point in improving a solution that is created either randomly or using a constructive heuristic. A selection perturbative hyper-heuristic can employ single point or multipoint search to choose low-level perturbative heuristics. In the case of the former the hyper-heuristic is composed of two components, one for *heuristic selection* and the other for *move acceptance*. In the case of selection perturbative hyper-heuristics performing multipoint search, by the nature of multipoint searches such as evolutionary algorithms the approach performs both the processes of heuristic selection and move acceptance and there is no need for separate components. The evolutionary algorithm is applied in the same way as for selection constructive hyper-heuristics with the only difference being that the low-level heuristics are perturbative. Generation hyper-heuristics use genetic programming [4] to create new low-level heuristics. Generation construction hyper-heuristics induce new low-level heuristics comprised of existing low-level heuristics or components thereof, problem attributes and arithmetic and conditional operators. On the other hand generation perturbative hyper-heuristics produce new low-level perturbative heuristics or move operators. However, the field of generation perturbative hyper-heuristics is still in its infancy with initial attempts combining existing perturbative low-level heuristics with conditional operators.

To put this in context consider the examination timetabling problem [5]. Examinations have to scheduled in timetable periods so as to meet all hard constraints of the problem and minimize the number of soft constraint violations. The constructive heuristics for this problem are used to order the examinations according to difficulty of scheduling and allocate them in this order. An example of a heuristic is the number of available timeslots available on the timetable at a given point in construction to schedule the examination in. This heuristic is referred to as the *saturation degree* and the examinations are sorted in ascending order according to the heuristic and scheduled in order. An example of another constructive heuristic is *largest enrolment*, i.e. the number of students sitting the examination. In this case the examinations are sorted in descending order according to this value. A selection constructive hyper-heuristic selects a heuristic to use at each point in the timetable construction process. Low-level pertubative heuristics for this domain include swapping individual examinations, swapping timetable periods, swapping a subset of examinations, amongst others. Similarly, a selection perturbative hyper-heuristic selects a heuristic to apply at each point of improving a timetable, i.e. reducing the hard and/or soft constraint cost of an initial timetable which is created randomly or using a constructive heuristic. A generation constructive hyper-heuristic will be used to create a new heuristic for the domain of examination timetabling by combining arithmetic operators and conditional operators with problem attributes, e.g. the number of potential clashes an examination can be involved in, the number of students sitting an examination.

Evolutionary algorithms have been employed by both selection and generation hyper-heuristics, however the implementation of evolutionary algorithm hyper-heuristics can be a tedious and daunting task. This paper presents a Java toolkit, *EvoHyp*, for the implementation of evolutionary algorithm hyper-heuristics. This will enable researchers and practitioners to focus on implementation of the problem domain. Existing toolkits for hyper-heuristics such as *HyFlex* and $HYPERION^2$ [6], [7] do not alleviate the user from the task of implementing the evolutionary algorithm hyper-heuristic. *HyFlex* is a Java toolkit that supports the implementation of selection perturbative hyper-heuristics. The toolkit provides methods for creating an initial solution, methods for calculation of the objective function and low-level perturbative heuristics for six combinatorial optimization problems, namely, the maximum satisfiability problem, flow-shop scheduling, one dimensional bin packing, personnel scheduling, travelling salesman and vehicle routing. The user is required to implement the selection perturbative hyper-heuristic to solve the six problems. $HYPERION^2$ allows the user to perform a trace of the implemented metaheuristic or hyper-heuristic in solving a particular problem, enabling the user to understand how the solution is arrived at. Hence, the purpose of *EvoHyp* is different from existing toolkits. The user can focus on the implementation of the application domain and use the evolutionary algorithm hyper-heuristics as a black-box to solve the problem at hand.

We have used *EvoHyp* to successfully implement selection constructive, selection perturbative and generation constructive hyper-heuristics for the university course timetabling and one dimensional bin packing problems. However, this paper focuses on describing the evolutionary algorithms implemented by the toolkit and illustrating how the toolkit can be used. Simple, hypothetical examples are used for this purpose and hence do not require knowledge of the problem domain.

The paper presents an overview of *EvoHyp* in the following section. Section III presents the genetic algorithm selection hyper-heuristic and section IV the genetic programming generation hyper-heuristic. The distributed versions of the genetic algorithm and genetic programming hyper-heuristics are presented in section V. The paper concludes by outlining future extensions of the *EvoHyp* toolkit.

II. Overview of *EvoHyp*

As illustrated in Fig. 1 *EvoHyp* [1] contains four libraries or packages, namely, *GenAlg*, *GenProg*, *DistrGenAlg* and *DistrGenProg*.

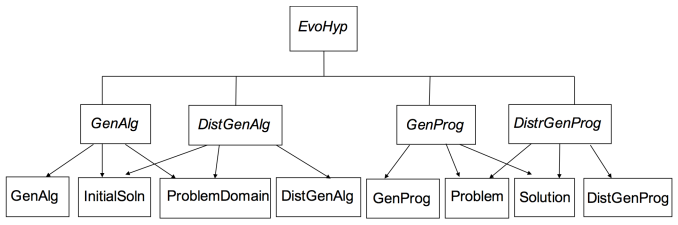

Fig. 1. *EvoHyp* toolkit

The *GenAlg* library allows for the implementation of genetic algorithm selection hyper-heuristics. Selection constructive and selection perturbative hyper-heuristics can be implemented using *GenAlg*. *DistrGenAlg* is the distributed version of *GenAlg*. Implementation of the genetic algorithm is distributed over a multicore architecture. Fig. 1 illustrates the classes in each library that forms the interface between *EvoHyp* and the problem domain. The *GenAlg* library contains the *GenAlg* class which implements the genetic algorithm selection hyper-heuristic. Similarly, *DistGenAlg* contains the *DistrGenAlg* class which implements the distributed genetic algorithm selection hyper-heuristic which is distributed over a multicore architecture. Both packages include the abstract classes *InitialSoln* and *ProblemDomain* which must be implemented by the user to link the problem domain and genetic algorithm hyper-heuristics.

GenProg allows for implementation of genetic programming generation hyper-heuristics. Given the infancy of the research on generation perturbative hyper-heuristics at this point, a toolkit for generation perturbative hyper-heuristics is not of benefit and more research in this area is needed before a library can be made available. Hence, *GenProg* only caters for the implementation of generation constructive hyper-heuristics. *DistrGenProg* is the distributed version of *GenProg* which distributes the genetic programming algorithm over a multicore architecture.

The following sections describe the evolutionary algorithms provided by each of the libraries and illustrate how the libraries can be used.

III. *GenAlg*

This section describes the *GenAlg* library. The library can be used to implement selection constructive or selection perturbative hyper-heuristics. Section III-A describes the genetic algorithm hyper-heuristic and section III-B illustrates how the library can be used.

A. Genetic Algorithm Hyper-Heuristic

The *GenAlg* library implements the generational genetic algorithm [8] depicted in Algorithm 1.

[1]http://titancs.ukzn.ac.za/EvoHyp.aspx

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
```

Each chromosome is a combination of characters, with each character representing a low-level constructive or perturbative heuristic. For example, *efab*, where *e*, *f*, *a* and *b* represent low-level heuristics. The heuristics are applied from left to right, to either construct a solution in the case of selection constructive hyper-heuristics or improve a solution in the case of selection perturbative hyper-heuristics. The user must specify the characters representing the low-level heuristics. In the case of selection constructive hyper-heuristics the user will use the heuristic combination produced by the hyper-heuristic to create a solution to the problem by applying the heuristic represented by each character in the combination to create the next part of the solution. For example, for the examination timetabling problem each character in the combination corresponds to a heuristic to select the next examination to schedule. Similarly, in the case of selection perturbative hyper-heuristics each character in the heuristic combination represents a pertubative heuristic applied to improve the solution, e.g. swap two randomly selected examinations. Each chromosome is created by randomly selecting characters representing low-level heuristics until a specified length is reached. The length of each chromosome is randomly chosen to be between 1 and a maximum chromosome length specified by the user.

Each chromosome is passed to a method defined by the user as part of the problem domain, namely, `evaluate` to calculate its fitness. Tournament selection [8] is used to select parents which the mutation and crossover operators are applied to.

The mutation operator randomly selects a mutation point and the character at that position in the chromosome is replaced with a randomly created substring. The length of the substring is randomly chosen to be in the range 1 to the maximum length specified by the user as the mutation length. The crossover operator randomly selects two points in each of the parents and the parent chromosomes are crossed over at these points to produce two offspring. The fitter of the two offspring is returned as the result of the operator. If the user has specified a limit on the offspring size, the heuristic combination is pruned to be the maximum specified offspring length. The population of each generation is created by applying mutation and crossover as specified by the genetic operator application rates provided by the user. For example, if the mutation rate is 40% and the crossover rate 60%, 40% of the population will be created using mutation and the remaining 60% using crossover.

The generational genetic algorithm terminates when a maximum number of generations is reached.

The following section illustrates how *GenAlg* can be used to solve a problem.

B. Using GenAlg

GenAlg enables the user to focus on implementing the problem domain. Fig. 1 illustrates the interaction between *GenAlg* and the problem domain.

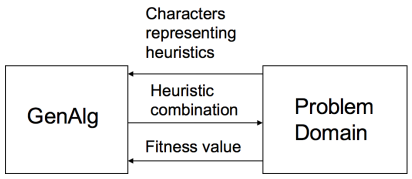

Fig. 2. Interaction between *GenAlg* and the problem domain

As is illustrated in the figure, the user needs to provide a list of characters representing the low-level constructive or perturbative heuristics. The genetic algorithm evolves a population of heuristic combinations. The user is required to implement the problem domain including methods for constructing a solution or improving a solution using the heuristic combination, determining the fitness of a heuristic combination and determining which of two heuristic combinations are fitter. In implementing the problem domain the user has to implement two abstract classes, namely, *ProblemDomain* and *InitialSolution*.

The steps involved in implementing a genetic algorithm hyper-heuristic using *GenAlg* are:

1) Implement the abstract class *InitialSolution* for the problem to be solved.
2) Implement the abstract class *ProblemDomain* for the problem to be solved.
3) Use the class *GenAlg* to implement a genetic algorithm selection hyper-heuristic to solve the problem.

These steps are described below.

Implementing the *InitialSoln* class

The purpose of this class is to create a solution using the heuristic combination. The class, e.g. *MySoln* `extends` *InitialSoln*. In addition to the methods for solving the problem the *InitialSoln* class must implement the following methods for the particular problem domain:

- `public int fitter(InitialSoln other)` - This method must be overridden in the concrete implementation. If the heuristic combination in the current instantiation is fitter than `other` a value of 1 must be returned, if the fitness is the same a value of 0 must be returned and if `other` is fitter a value of -1 must be returned. An example of an implementation of the `fitter` function is :

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

In this example `fitness` is a data element of the class.

- `abstract public double getFitness()` - This is an abstract class that must be implemented. It essentially returns the fitness of the heuristic combination.
- `abstract public Object getSoln()` - This method must return the solution created using the heuristic combination. Note that the return type is `Object` so the user can define the type of the solution created using the heuristic combination. For example, in solving the university course timetabling problem the solution may be a `Timetable`, however for the travelling salesman problem the solution would be a `Tour`.
- `abstract public void setHeuCom(String heuCom)` - This is a set method that stores the heuristic combination `heuCom`.
- `abstract public String getHeuCom()` - This method returns the heuristic combination `heuCom`.

Implementing the *ProblemDomain* class

The *ProblemDomain* class serves as an interface between the *InitialSoln* and *GenAlg* class. The concrete class `extends` *ProblemDomain*. At least the following abstract method `evaluate` must be implemented in the concrete class:

```
public InitialSoln evaluate(String heuCom)
```

This method must evaluate the heuristic combination (*heuCom*) by using it to create a solution and calculating the fitness of the heuristic combination. The methods for creating a solution can be included in the implementation of this class or the implementation of *InitialSoln*. An example of an`evaluate` method is depicted below:

```
public ComOptSoln evaluate(String heuCom)
{
  ComOptSoln soln = new ComOptSoln();
  soln.setHeuCom(heuCom);
  soln.createSoln();

  return soln;
}
```

ComOptSoln is the concrete implementation of the abstract class *InitialSoln*. The class is not part of the *EvoHyp* library, it is an example of a concrete instance of the *InitialSoln* class which will be provided by the user for the particular problem domain and as such represents a combinatorial optimization problem solution (ComOptSoln). In the example the method for creating the solution using the heuristic combination is included in the *ComOptSoln* class. The `evaluate` method calls the set method to store the heuristic combination in the instance of *ComOptSoln* and evokes the method to create the solution using the heuristic combination.

Using the *GenAlg* class

In order to implement a genetic algorithm selection hyper-heuristic an instance of the *GenAlg* class must be created. A seed and string of characters representing the heuristics must be passed to an instantiation of the class via the constructor: `public GenAlg(long seed, String heuristics)`.

The user is required to specify the parameter values to be used by the genetic algorithm, namely, population size, number of generations, tournament size, initial chromosome length, mutation length and offspring length. If there is no limit on the size of the offspring produced by the genetic operators a -1 must be specified for the offspring length. The parameter values can be set by means of a file containing the parameter values or setting each parameter individually using the set methods provided by *GenAlg*.

GenAlg uses the `evaluate` method defined in the implementation of the abstract method *ProblemDomain* to calculate the fitness of each element of the population. An instance of the concrete implementation of the *ProblemDomain* class must be passed to the *GenAlg* class using the `setProblem` method in the *GenAlg* class: `public void setProblem(ProblemDomain problem)`. To implement the genetic algorithm hyper-heuristic the `evolve` method from the *GenAlg* class must be evoked: `public InitialSoln evolve()`. This method returns the best chromsome evolved over a genetic algorithm run of *n* generations. The method returns an instance of *InitialSoln*. Suppose that *ComOptSoln* is a concrete implementation of *InitialSoln* and *ComOptProb* is a concrete implementation of the *ProblemDomain* class, an example of code to implement a genetic algorithm selection hyper-heuristic is listed below:

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

System.out.println("Best Solution");
System.out.println("--------------");
System.out.println(solution.getFitness());
System.out.println(solution.getHeuCom());
displaySolution(solution.getSoln());
```

The current system time is used as the seed. Three heuristics are used for this problem represented by *s*, *l* and *w*. The parameters for the genetic algorithm are specified in the file *Parameters.txt*. An instance of *ComOptProb* is created and passed to an instance of *GenAlg*. The `evolve` method is evoked to implement the selection hyper-heuristic and the returns the best performing heuristic combination and corresponding solution in `solution` which in this case is an instance of *ComOptSoln*. The methods `getFitness`, `getHeuCom` and`getSoln` are used to get the fitness, heuristic combination and solution respectively.

IV. *GenProg*

The *GenProg* library can be used to implement generation constructive hyper-heuristics. The following section describes the genetic programming hyper-heuristic and section IV-B illustrates how *GenProg* can be used for a problem domain.

A. *Genetic Programming Hyper-Heuristic*

Genetic programming is used to induce new low-level constructive heuristics. The generational algorithm depicted in Algorithm 1 is implemented. However, each element of the population is a parse tree representing a heuristic.

The evolved heuristic can be an arithmetic function or an arithmetic rule. In the case of the an arithmetic function, the function set is comprised of the standard arithmetic functions namely, addition, subtraction, multiplication and division. The division operator is protected division which returns a value of 1 if the denominator is zero. In addition to these operators, the function set for generating heuristic functions include an *if-then-else* operator and the relational operators, less than, greater than, less than equal to, greater than equal to, equal to and not equal to. The relational operators are only used in creating the first argument of the *if-then-else* operator. The terminals are problem attributes specified by the user. The user has to specify a list of characters representing the attributes for the problem.

The *grow* method [4] is used to create each element of the initial population. The root of the tree is randomly selected from the function set. The nodes at the maximum initial depth specified by the user are randomly selected from the terminal set. The nodes at the remaining depths are randomly selected from both the function and terminal sets.

The fitness of each heuristic is calculated by passing the heuristic to the method defined by the user as part of the problem domain, `evaluate` to use the heuristic to create a solution to the problem. Tournament selection is used to choose parents to apply the genetic operators to.

Mutation and crossover are used to create the offspring of each generation. The standard mutation operator is used [4]. This operator randomly selects a mutation point in the parent. The subtree rooted at the mutation point is replaced by a randomly created subtree. The depth of the subtree is limited to the maximum mutation depth specified by the user. The crossover operator randomly chooses two crossover points in both of the parents and the subtrees rooted at these points are swapped to produce two offspring. If the offspring produced by the genetic operators exceed the maximum offspring depth specified by the user the offspring are pruned by replacing function nodes with randomly selected terminals at the maximum permitted depth. As in the case of *GenAlg* the population of each generation is created by applying mutation and crossover as specified by the genetic operator application rates provided by the user. For example, if the mutation rate is 40% and the crossover rate 60%, 40% of the population will be created using mutation and the remaining 60% using crossover.

The generational genetic programming algorithm terminates when the number of generations specified by the user has been reached.

The following section describes how *GenProg* can be applied to a problem domain.

B. *Using GenProg*

Fig. 2 illustrates the interaction between the *GenProg* implementation and the problem domain implementation.

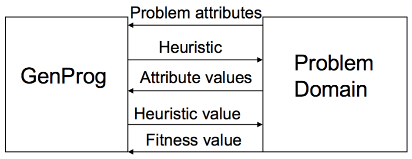

Fig. 3. Interaction between *GenProg* and the problem domain

The user is required to specify a list of characters representing the attributes of the problem. *GenProg* creates low-level constructive heuristics of type *Node*. As part of the implementation of the problem domain the user has to implement methods that use a heuristic evolved by *GenProg* to solve the problem and calculate the fitness of the evolved heuristic. The *GenProg* library provides the *Evaluate* class to calculate the value the heuristic evaluates to given the attribute values by the user. The fitness of each heuristic is calculated using the `evaluate` method defined by the user as part of the problem domain. The fitness of the heuristic is calculated by using it to create a solution to the problem. Concrete implementations of the abstract classes *Solution* and *Problem* must be implemented as part of the problem domain by the user.

The steps involved in implementing a generation constructive hyper-heuristic using *GenProg* are:

- Implement the abstract class *Solution* for the problem to be solved.
- Implement the abstract class *Problem* for the problem to be solved.
- Use *GenProg* to implement a generation constructive hyper-heuristic to create a heuristic for the problem domain.

These steps are described below.

Implementing the *Solution* class

The purpose of this class is to create a solution using the heuristic. The class stores the heuristic and the solution created using it. The concrete class must `extend` *Solution*. In implementing the *Solution* abstract class the following methods should be implemented:

- `abstract public int fitter(Solution other)` - This method determines whether the heuristic in the current instantiation is fitter than *other* or not. If it is fitter a value of 1 is returned. If *other* is fitter a value of -1 is returned, otherwise a value of 0 is returned.
- `abstract public double getFitness()` - Returns the fitness of the heuristic.
- `abstract public Object getSoln()` - Returns the solution created using the heuristic. The type of the solution is *Object* and can be defined to be any type required by the user.
- `abstract public void setHeuristic(Object heuristic)` - Sets the heuristic produced by *GenProg* to be used to solve the problem.
- `abstract public Object getHeuristic()` - This method returns the heuristic to solve the problem.

Implementing the *Problem* class

The *Problem* class is essentially an interface between the concrete implementation of the *Solution* class and the *GenProg* class. The *Problem* class requires the following abstract methods to be implemented:

- `public Solution evaluate(Object heuristic)` - This method takes the heuristic as input and must use the heuristic to solve the problem at hand and return the solution as an instance of the type *Solution*. The type of the heuristic has been left as *Object* for possible future expansion, however in the current version of *GenProg* heuristics are of the type *Node*. The package provides the *Evaluator* class which provides the user with methods to interpret the heuristic given attribute values and obtain the heuristic value. An example of the `evaluate` function is depicted below:

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

  return soln;
}
```

  In this example the methods for creating a solution are implemented in the concrete implementation of the *Solution* class, *ComOptSoln*. The class is not part of the *EvoHyp* library, it is an example of a concrete instance of the *Solution* class which will be provided by the user for the particular problem domain and as such represents a combinatorial optimization problem solution (ComOptSoln).The `evaluate` method creates an instance of the *ComOptSoln* class and passes it the heuristic to solve the problem. The `createSoln` method in the *ComOptSoln* class is evoked to create a solution using the heuristic and the solution is returned.
- `public void setAttribs(String attribs)` - This method allows the user to set the attributes for the problem domain. The user is required to specify a string of characters, with each character representing an attribute.

Using *GenProg*

The genetic programming generation constructive hyper-heuristic is implemented by first creating an instance of the *GenProg* class. The *GenProg* constructor takes three arguments: `public GenProg(long seed,String attributes,int heuType)`. The first argument is the seed value to be used by the random number generator, the second the problem attributes in the form of a string with each character representing an attribute and the last argument is an integer value indicating whether an arithmetic function or arithmetic rule should be evolved. If the last argument is 0 an arithmetic function is evolved and if it is 1 an arithmetic rule.

As in the case of *GenAlg* the parameters for the genetic programming algorithm need to be set using either the `setParameters` method and supplying a file with the parameters or setting these individually using the set method for each parameter provided by *GenProg*. The parameter values that must be set include population size, number of generations, tournament size, initial tree depth, mutation depth, crossover application rate, mutation application rate and maximum offspring depth. If there is no limit on the depth of the offspring created by the genetic operators, the value of the offspring depth must be set to be -1.

GenProg calculates the fitness of each element of the population by calling the `evaluate` method from an instance of the concrete implementation of the *Problem* class. This instance must be passed to the *GenProg* class using the `setProblem` method: `public void setProblem(Problem problem)`.

The `evolve` method must be evoked to implement the generation constructive hyper-heuristic to evolve a heuristic for the problem domain: `public Solution evolve()`. This method returns an instance of type *Solution* which stores both the heuristic and the solution created using it. Example code for implementing a generation constructive hyper-heuristic is included below:

```
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(sol.getFitness());
System.out.print("Heuristic: ");
printInd((Node)sol.getHeuristic());
```

```
System.out.println();
System.out.println("Solution: ");
displaySolution((ArrayList)sol.getSoln());
```

In this example the concrete implementation of the *Solution* class is *ComOptSoln*. An instance of the *GenProg* class `gchh` is created and passed a seed, the problem attributes, the parameters via a file and an instance of the concrete implementation of *Problem*, `ComOptProb`. The current system time is used as the seed and the problem attributes are *a*, *b* and *c*. The 1 passed as the last argument to the constructor indicates that *GenProg* must generate the heuristic as an arithmetic rule. The `evolve` method returns a solution of type `ComOptSoln` which includes both the solution to the problem and the best performing heuristic used to create it.

During the evolution process the *GenProg* class uses the `evaluate` method to calculate the fitness of each heuristic, i.e. the element of the population. The heuristic of type `Node` is passed to the `evaluate` method and must be used to solve the problem. This heuristic is then passed to the concrete implementation of the *Solution* class, in the example *comOptSoln*, which contains the methods to create a solution to the problem using the heuristic. In order to use the heuristic to solve the problem it must be interpreted with the current values of the attributes provided by the user. The *GenProg* library provides the *Evaluator* class for this purpose. The class provides the following methods:

- `public Evaluator(String attributes,double attributeVals[]` - The constructor for the class requires the user to specify the attributes and values of the attributes.
- `public double eval(Node op)` - This method calculates the heuristic value given the heuristic produced by *GenProg*. This allows the user to evaluate the heuristic with attribute values externally as part of the problem domain.

In the above example this class is instantiated in the instance of the *ComOptSoln* class to create a solution. The heuristic of type `Node` is passed to the instance of *ComOptSoln* class via the `evaluate` method as indicated in the code above for the `evaluate` method. The method in the instance of the *ComOptSoln* which solves the problem passes the heuristic to an instance of the *Evaluator* class and evokes the `eval` method to get the corresponding heuristic value for the heuristic.

V. *DistrGenAlg* AND *DistrGenProg*

As the runtimes for evolutionary algorithm hyper-heuristics can be high for certain problem domains, the toolkit provides a distributed version of *GenAlg* and *GenProg*. In both the distributed versions the implementation of the evolutionary algorithm is distributed over a multicore architecture as follows:

- The task of creating the initial population is distributed over the number of specified cores. Subpopulations are created and evaluated on each core.
- The process of regeneration is distributed over the available cores. Subpopulations are created on each core by applying genetic operators and evaluating the offspring.

The implementation of the hyper-heuristics are the same as described in sections III and IV for *GenAlg* and *GenProg* respectively with the only difference being that the number of available cores to use must be specified when instantiating an instance of each class:

```
public DistrGenAlg(long seed, String
heuristics,int noOfCores)

public GenProg(long seed,String attributes,int
heuType)
```

VI. CONCLUSION

This paper has presented a Java toolkit for the implementation of selection constructive, selection perturbative and generation constructive evolutionary algorithm hyper-heuristics. The evolutionary algorithms for the hyper-heursitcs have been presented and how hyper-heuristics can be implemented using the toolkit has been illustrated. Future extensions of the toolkit will be to include a steady-state genetic algorithm option for *GenAlg* and *DistGenAlg*. While genetic programming has chiefly been used in generation hyper-heuristics as the field is advancing grammatical evolution, a variation of genetic programming, is gaining popularity. *EvoHyp* will be extended to include a grammatical evolution option for *GenProg*. The ultimate aim of the *EvoHyp* toolkit is to provide non-expert researchers and practitioners with a tool that does not require detailed knowledge of evolutionary algorithm hyper-heuristics to use. However, in the current version the user is still required to provide parameter values for the evolutionary algorithms and hence perform parameter tuning. Automated parameter tuning of the evolutionary algorithms will be incorporated into future versions *EvoHyp*. Finally, as the field of generation perturbative hyper-heuristics develops further, the toolkit will be extended to include genetic programming generation perturbative hyper-heuristics.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their helpful comments to improve the paper.

REFERENCES

[1] E. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and R. Qu, "Hyper-heuristics: a survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, 2013.

[2] E. K. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu, "A graph-based hyper-heuristic for educational timetabling problems," *European Journal of Operational Research*, vol. 176, pp. 177–192, 2007.

[3] N. Pillay, "Evolving hyper-heuristics for the uncapacited examination timetabling problem," *Journal of the Operational Research Society*, vol. 63, pp. 47–58, 2012.

[4] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, 1st ed. MIT, 1992.

[5] N. Pillay, "A review of hyper-heuristics for educational timetabling," *Annals of Operations Research*, vol. 239, no. 1, pp. 3–38, 2016.

[6] P. Ryser-Welch and J. F. Miller, "A review of hyper-heuristic frameworks," in *Proceedings of the Evo20 Workshop, AISB 2014*, 2014.

[7] N. Pillay, “Tutorial: An overview of evolutionary algorithm hyper-heuristics,” 2015 IEEE Congress on Evolutionary Computation (CEC 2015), http://www.cs.usm.maine.edu/ congdon/Conferences/CEC2015/Pillay.CEC2015.tutorial.pdf, May 2015.

[8] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing, 1989.