Templar - a framework for template-method hyper-heuristics

Jerry Swan and Nathan Burles

Department of Computer Science,
University of York, York, YO10 5GH, UK.
jerry.swan@cs.york.ac.uk,nathan.burles@york.ac.uk

Abstract. In this work we introduce Templar, a software framework for customising algorithms via the generative technique of template-method hyper-heuristics. We first discuss the need for such an approach, presenting Quicksort as an example. We provide a functional definition of template-method hyper-heuristics, describe how this is implemented by Templar, and show how Templar may be invoked using simple client-code. Finally, we describe experiments using Templar to define a 'hyper-quicksort' with the aim of reducing power consumption—the results demonstrate that the generated algorithm has significantly improved performance on the test set.

Keywords: Genetic Programming, Generative Hyper-Heuristics, Template Method, Energy Profiling, Reduced Power Consumption, Quicksort

1 Introduction

Despite two decades of research, scalability remains an issue for program synthesis via metaheuristics. Although there have been some recent promising results [20], for many problems generative approaches such as Genetic Programming (GP) [9] still work best at the scale of expressions. A greater degree of explicit structure can be imposed by Grammatical Evolution [19], although this approach is known to suffer from a lack of locality [18]. By contrast, human ingenuity has already provided a vast repertoire of specialized algorithms, usually with known asymptotic behaviour. Given these limitations of scale, how can we best use generative approaches to improve upon human-designed algorithms?

One motivating observation is that the performance of many well-known algorithms arises in practice from the inclusion of heuristically-informed decision points and/or *ad hoc* boundary conditions (e.g. the recursion depth at which Musser's widely-used sorting algorithm 'Introsort' switches from Quicksort to Heapsort [15]). The method of hyper-heuristics [4] ('heuristics to select or generate heuristics') performs metaheuristic search over function spaces in which the functions are themselves heuristics. This leads naturally to the idea of a generative hyper-heuristic approach to algorithm design, in which we search for superior versions of the heuristically-informed parts of an algorithm. As described in a recent paper on 'Template Method hyper-heuristics' [27], rather than hope to synthesise a complete algorithm from the 'bottom up', we can instead use the

```
DoubleArray
qsort(arr : DoubleArray, pivotFn : DoubleArray → Double) {
  Double pivot = pivotFn( arr );
  // ^^^ pivotFn can be varied generatively
  return qsort( arr.filter( < pivot ), pivotFn )
      ++ arr.filter( == pivot )
      ++ qsort( arr.filter( > pivot ), pivotFn );
}
```

Listing 1: Quicksort with variant pivot function

'Template Method' Design Pattern [6] and provide an algorithm skeleton (the template) that is parameterized by one or more 'variation points'. Each variation point is permitted to express a family of behaviours, whether constrained merely by the types of its function signature, or more strongly via 'design by contract'. By expressing an algorithmic framework in Template Method terms, we can then use generative techniques (in this case GP) to learn good implementations for the variant parts. By 'good', we mean 'biased towards the distribution to which the algorithm is exposed'. If our algorithms are metaheuristics, an important corollary is that they are not subject to the 'No Free Lunch' theorem [24], since the distribution over problem instances is biased away from uniform by the training set. This approach has been successfully demonstrated in the generation of more effective selection and mutation operators for Genetic Algorithms [25, 26].

1.1 Quicksort - a motivating example

Although Quicksort [7] is of course a generic algorithm (i.e. it can be defined over any partially-ordered type), for simplicity of exposition we consider it to operate on arrays of floating point values, denoted by DoubleArray. Quicksort performance is well-known to be dependent on the choice of pivot, which we can therefore consider as a variation point for the algorithm. The pivot-function can be taken to have signature: pivotFn : DoubleArray → Double, i.e. it returns the choice of pivot value for a given input array. Listing 1 gives the pseudocode for a version of Quicksort that takes the pivot function as an additional argument. By specifying our algorithm in this fashion, we can generate a version that best meets some training criterion specified in client-code, such as robustness against pathological inputs (e.g. hardening against denial-of-service attacks), reduced power consumption, etc.

In the remainder of this article we describe TEMPLAR, a generic Java™ framework for Template Method hyper-heuristics, and show how it can be used for rapid prototyping. The outline of the article is as follows: in Section 2, we give a functional definition of Template Method hyper-heuristics and describe the corresponding TEMPLAR implementation. Starting from elementary examples, in Section 3 we show how TEMPLAR is configured and invoked from client-code. In Section 4, we detail an experiment with 'hyper-quicksort' to demonstrate the utility of this approach, giving conclusions in Section 5.

```
@FunctionalInterface
// ^ tells Java this can be treated as a lambda function
interface Fun1<Arg, Result> {
  Result apply(Arg arg);
}

@FunctionalInterface
interface Fun2<Arg1, Arg2, Result> {
  Result apply(Arg1 arg1, Arg2 arg2);
}

// We can now use functions as parameters
// and return values:
Fun1<A, C> compose(Fun1<A, B> f, Fun1<B, C> g) {
  return (A x) -> g.apply(f.apply(x));
}
```

Listing 2: Higher-order functions in Java

2 A functional Framework for Template Method Hyper-heuristics

For an algorithm with function signature $I \rightarrow O$, Template Method hyper-heuristics can be described as follows:

1. A list of *variation points* describing the parts of the algorithm to be automatically generated:

$$\text{VP} : (I_1 \rightarrow O_1) \times (I_2 \rightarrow O_2) \times \ldots \times (I_n \rightarrow O_n)$$

2. An *algorithm template* expressing the algorithm skeleton. The template produces a customized version of the algorithm from automatically-generated implementations of the variation points:

$$\text{Template} : \text{VP} \rightarrow (I \rightarrow O)$$

3. A *loss function* to evaluate the customized algorithm on supplied training and testing sets as a function of the difference between actual and expected outputs:

$$\text{LossFn} : (O \times O) \rightarrow V$$

4. An *algorithm factory* that searches the space of variation points to produce an optimized version of the algorithm:

$$\text{Factory} : \text{VP} \times \text{Template} \times \text{LossFn} \rightarrow (I \rightarrow O)$$

Despite the success of applications such as [5], the vast majority of hyper-heuristic publications concern selective hyper-heuristics—there are far fewer on generative approaches. Hoos [8] discusses automated algorithm improvement,

```
interface AlgTemplate<I, O> {
  public Fun1<I, O> makeAlg(ProgramList programs);
}

class AlgFactory<I, O> {
  AlgFactory(GPConfig[] variationPointConfigs,
    AlgTemplate<I, O> template) { ... }

  ProgramList run(FitnessCases<I, O> cases,
    LossFn<O> lossFn) { ... }
}
```

Listing 3: Core TEMPLAR classes

and the benefits this can bring—such as removing the menial work involved in manually experimenting with new algorithms and the improved performance of the resulting algorithms. Unfortunately generative hyper-heuristics are laborious to implement on a per-case basis, but also nontrivial to generalize. There are several reasons for the latter: firstly, generation of the variant programs is typically implemented via GP and is invoked repeatedly by the Factory in the process of the hyper-level search. Unfortunately, popular GP implementations such as ECJ [12] and PushGP [21] prefer to be the 'top' of the system (not least because of their 'configuration file' based approach) and hence attempting to use them for generative hyper-heuristics is not a simple matter. Secondly, the fitness of each variation point depends on the others, and a generic implementation of the complex 'wiring diagram' of dependencies is likely to be offputting to many experimenters. There has recently been some interesting related work using an algorithm configuration tool [10] to play an analogous role to the grammar in Grammatical Evolution [19]. This has been successfully used to automatically generate local search heuristics [13].

3 The TEMPLAR Framework

Functional programming makes intensive use of higher-order functions, i.e. functions which can accept (and significantly, return) other functions. Listing 2 shows how higher-order functions can be defined in Java. As of Java 8, equivalents of Fun1,Fun2 are supported natively as java.util.Function1,Function2 and Lambda functions can be expressed in the concise syntax we use in the program listings[1].

The core TEMPLAR classes are given in Listing 3. AlgFactory defines the process of hyper-heuristic search for program variants (supplied subclasses offer Iterative Improvement or Genetic Algorithms), whereas the end-user must subclass AlgTemplate in order to generate a new algorithm from a ProgramList containing a generated program for each of the variation points. At the top of Listing 4 is IdentityTemplate, the simplest possible subclass of AlgTemplate. It has no actual algorithm skeleton, i.e. it merely wraps the generated program of its

[1] docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

```
class IdentityTemplate
implements AlgTemplate<Double, Double> {
  public Fun1<Double, Double> makeAlg(ProgramList progs) {
    // Wrap the VP in a function:
    return (Double arg) -> progs.get(0).execute(arg);
  }
}

class CompositionTemplate
implements AlgTemplate<Int, String> {
  Fun1<Int, String> makeAlg(ProgramList progs) {
    Fun1<Int, Double> f = (Int arg) ->
      progs.get(0).execute(arg);
    Fun1<Double, String> g = (Double arg) ->
      progs.get(1).execute(arg);
    // this template composes the two variant programs:
    return compose(f, g);
  }
}
```

Listing 4: Simple AlgTemplate examples

(sole) variation point inside a function and returns it. This is therefore equivalent to a 'standard' (i.e. non-template) generative approach. Of slightly greater utility is CompositionTemplate, in which the algorithm skeleton is the composition of two generated variants f and g to give $f(g(x))$. As can be seen in Listings 4 and 5, using TEMPLAR requires that the end user do only the following:

1. Define an AlgTemplate subclass as described above.
2. Configure GP for each variation point (Listing 5).
3. Invoke TEMPLAR on user-supplied training and testing sets (Listing 5).

The GPConfig class contains all the information (function set, population size, mutation and crossover rates and operators, etc) required to generate programs for each variation point. In Listing 5, the RationalFunctionConfiguration ($\{+, -, *, \%\}$) built-in to TEMPLAR is used. The lossFn parameter determines the fitness of algorithm variants as a function of the difference between actual and expected outputs—here root mean square error (RMS) is used.

In terms of computational expressiveness, TEMPLAR is equivalent to approaches such the Grammatical Evolution or the automated configuration tool *irace* mentioned above. Aside from the issue of lack of locality in such approaches [18], the main advantage we claim for TEMPLAR is that it is more programatically integrated: unlike grammar configuration files or parsed EBNF grammar strings, the entities manipulated by TEMPLAR are all first-class objects and the validity of the resulting program can be enforced to the extent that Java's type-system permits. This approach also brings some other benefits when working with non-trivial datatypes. For example, the algorithm described above operates on values of type Double. For more sophisticated algorithms, it is desirable

```
public static void main(String[] args) {
  // Configure GP for each variation point
  AlgTemplate<Double, Double> template =
      new IdentityTemplate();
  GPConfig[] vpConfigs = new GPConfig[] {
      new RationalFunctionConfig()};
  LossFn<Double> lossFn = new RMSLossFn<Double>();

  // Set-up training and testing sets:
  FitnessCases trainingSet = ...
  FitnessCases testSet = ...

  // Invoke Templar:
  ProgramList bestVPs = Templar.run(template, vpConfigs,
      trainingSet, testSet, lossFn);
  println("best VPs:" + bestVPs);
}
```

Listing 5: Configuring and running Templar

for the generated programs to operate directly on user-defined datatypes (e.g. BitString, Timetable, RoutePlan, AntTrail etc). However, the manual creation of GP nodes for function sets on such custom representations is tedious. Following [11], a FunctionSetGenerator utility is provided by Templar, using reflection to automatically build a function set from the methods defined on *any* Java object.

4 Hyper-quicksort

By following the steps described above, it is a simple matter to create hyper-heuristic versions of any algorithm. In this section, we describe how to create a 'hyper-quicksort'. Listing 6 gives the complete client code for this. Java is an unnecessarily verbose language and unlike languages such as C++ which can reduce syntactic clutter by using typedef to create a type alias, there is no explicit support in Java for this. Something similar can, however, be achieved by creating an appropriately named subclass, as has been done here with PivotFn.

It is well-known that Quicksort does not perform well on certain pathological distributions (e.g. nearly-sorted or reverse-sorted input) [14]. To demonstrate the effectiveness of our approach, we used as input a 'pipeorgan' distribution, in which the values in the input array increase monotonically until some randomly-specified index, then decrease monotonically. Quicksort is known to behave poorly against data drawn from this distribution, so the case study we present could be considered as a simple example of 'hardening' software against a denial-of-service attack. An example is given in Fig. 1.

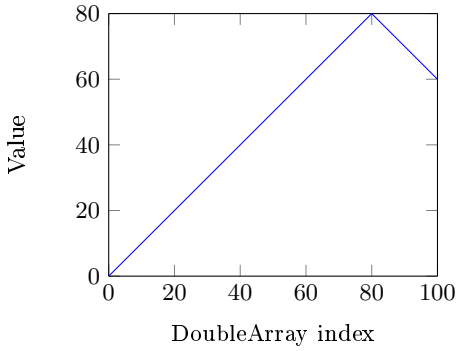

Fig. 1. Example of 'pipeorgan' training set for hyper-quicksort

4.1 Optimising for Energy Reduction

Measuring the power consumption of a computer can be performed using a number of methods, such as reading the data from an uninterruptible power supply, or the use of an electronic watt-meter. Recently, software-based alternatives have become available that use power models in conjunction with timings and system information such as CPU utilisation in order to provide reasonably accurate estimations [2]. These hardware and software tools can be very useful for informing users of their power consumption, or as a course-grained overview of energy used by an application. Unfortunately their precision and accuracy are too low to be suitable for use in automated software improvement—competing algorithms would need to be run an inordinate number of times to obtain a single measurement, essentially making the hyper-heuristic intractable. Although software-based tools can only provide estimates of the power consumed, the important factor is relative consistency between competing solutions. The use of a more accurate software measure is thus acceptable, such as the Wattch [3] and JALEN [16] tools. Wattch is a cycle-level simulator that has been used successfully with GP (e.g. [23]), however it requires a parameterised model of the processor and does not support Java. JALEN is a more recent alternative that targets Java, and can calculate an estimate of the power consumption by monitoring the execution time alongside system resources such as processor utilisation. Due to its simplicity and target language, in this work we have chosen to use JALEN to determine the fitness of competing algorithms.

4.2 Experimentation and Results

The experimental setup was as follows: the GP metaheuristic was configured with a population size of 100, 200 generations per run, an initial tree depth of 2 and a max tree depth of 4. These values were determined empirically as a reasonable trade-off between solution quality and execution time of the hyper-heuristic. All other GP parameters and operators were as the EpochX 1.4 defaults. An iterative-improvement hyper-heuristic was used on top of a GP metaheuristic,

```
// 1. Define an AlgTemplate subclass:

// 'typedef' for PivotFn to increase readability:
@FunctionalInterface
abstract class PivotFn
extends Fun2<DoubleArray, Int, Double> {
  Double apply(DoubleArray a, Int recursionDepth);
}

class QuicksortTemplate
implements AlgTemplate<DoubleArray, Int> {
  Fun1<DoubleArray, Int> makeAlg(ProgramList progs) -> {
    PivotFn pivotFn = (DoubleArray a, Int recursionDepth) -> {
      int progResult = programs.get(0).execute(a.size(),
          recursionDepth);
      int numSamples = Math.min(Math.abs(progResult),
          a.size());
      return median(randomSample(a, numSamples));
    };
    return (DoubleArray arg) -> Quicksort.sort(arg, pivotFn);
  }
}

Int quicksort(DoubleArray a, PivotFn pivotFn);
// ^ instrumented to return fitness (e.g. power consumed)

// 2. Configure GP to generate pivotFn VP:
List<Var> vars = {Var("size"), Var("recursionDepth")};
List<Node> funcSet = {IfFn(), LessFn(), AddFn(), ...};
GPParams params = ... // crossover, selection, etc
GPConfig vpConfigs = {new GPConfig(funcSet, vars, params)};

// 3. Invoke Templar
AlgTemplate<Double, Double> template =
    new QuicksortTemplate();
FitnessCases trainingSet = ...
FitnessCases testSet = ...
Templar.trainAndTest(template, vpConfigs, trainingSet,
    testSet, new RMSLossFn<Double>());
```

Listing 6: Client-code for Hyper-quicksort

with a stopping condition of 100 iterations. The training set contained 70 cases, where each case consisted of 100 arrays to be sorted, each with size 100. The function to be minimized was the energy used to perform the Quicksort, using a modified version of the Jalen tool[2].

[2] In order to provide the highest possible accuracy, Jalen was modified to run from within the Java Virtual Machine, rather than as an external Java Agent

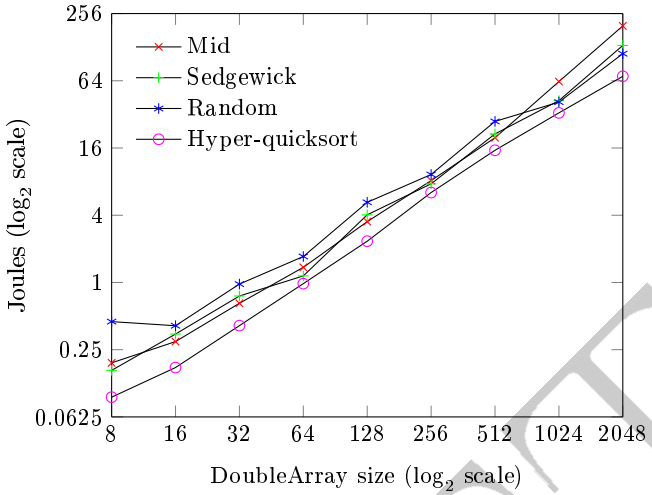

Fig. 2. Energy required to sort 1000 'pipeorgan' arrays using various pivot functions

As described above, the pivot function is the sole variation point, and the input variables for the generated programs are 'array size' and 'recursion depth'. The template for the evolved GP function is configured as follows: the Oracular pivot value of an array of values is its median. Here, we use GP to generate the function GP(arraySize, recursionDepth) $\rightarrow$ numSamplePoints and take as pivot the median of numSamplePoints randomly-chosen array elements. Although Quicksort is defined on anything with a partial order, note that this particular method only works when the array values are numeric. A number of different functions were generated by different runs of the GP, however commonly-observed amongst the fitter functions was recursionDepth. This suggests that as the recursion depth increases (and array size decreases), performance is improved by increasing the number of samples.

In Fig. 2, the performance is compared with three well-known pivot functions: 'Middle index', 'Random index', and 'Sedgewick' (the latter returning the median of the first, middle, and last elements). In this testing, JALEN was used to calculate the energy required to sort 1000 arrays of varying lengths using each of the pivot functions. This test was performed 100 times, and Fig. 2 presents the mean results. Hyper-quicksort can be seen to outperform all of the alternatives. In order to verify that the results are significant, we used the non-parametric Mann-Whitney U-test [1]. For each array length, $2^3 - 2^{11}$, we ran the U-test comparing the set of results obtained using hyper-quicksort to each of the alternative pivots in turn. As shown in Table 1, in all of the cases it was clear that the distributions of the results were significantly different, with $p < 0.05$.

Having demonstrated significance, it is important that the effect size is also measured [1]. For this we used Vargha and Delaney's non-parametric $\hat{A}_{12}$ statistic [22] for each of the sets of results above. This test returns the probability that using algorithm A provides higher values than using algorithm B. We wish

Array size	Middle index J	Middle index p	Middle index e	Sedgewick J	Sedgewick p	Sedgewick e	Hyper-quicksort (J)
8	0.191	7.46e-32	0.981	0.163	8.37e-32	0.980	0.094
16	0.296	1.20e-30	0.971	0.345	1.25e-31	0.979	0.173
32	0.651	8.13e-32	0.980	0.757	7.25e-32	0.981	0.410
64	1.366	4.80e-33	0.990	1.145	1.68e-30	0.970	0.976
128	3.505	4.80e-33	0.990	4.034	4.14e-33	0.991	2.341
256	8.175	4.14e-33	0.991	7.646	3.41e-32	0.983	6.387
512	19.777	4.33e-34	0.998	21.391	3.62e-34	0.999	15.268
1024	62.961	2.52e-34	1.000	42.508	6.44e-33	0.989	33.012
2048	198.438	2.52e-34	1.000	132.663	2.52e-34	1.000	70.234

Array size	Random index J	Random index p	Random index e
8	0.446	8.37e-32	0.980
16	0.410	8.37e-32	0.980
32	0.967	4.80e-33	0.990
64	1.708	4.80e-33	0.990
128	5.221	2.52e-34	1.000
256	9.269	8.87e-34	0.996
512	27.685	2.52e-34	1.000
1024	41.245	3.61e-32	0.983
2048	111.894	3.47e-33	0.991

Table 1. Energy (J) required to sort 1000 'pipeorgan' arrays using various pivot functions, as well as the p-values (p) and effect size measures (e) comparing hyper-quicksort to the alternative pivot functions.

to minimize the energy used, and so in each case algorithm B is hyper-quicksort, and algorithm A the alternative—the results of the $\hat{A}_{12}$ statistic therefore give the probability that using the alternative pivot function will use more energy than hyper-quicksort. Vargha and Delaney suggest the following guidelines for interpreting the effect size: 0.5 indicates that there is no benefit to either algorithm; 0.56 indicates a small difference; 0.64 indicates a medium difference; and 0.71 indicates a large difference. As would be expected, there is some range in the effect sizes across the various array sizes and alternative pivot functions—from 0.970 to 1.000. The results are presented in full in Table 1, and show definitively that the Quicksort variant generated by Templar provides improved performance compared to common pivot functions, with respect to the energy required to sort families of data drawn from a pathological 'pipeorgan' distribution.

5 Conclusion and Future Work

We have introduced Templar, a framework that supports 'Template Method Hyper-heuristics' [27], by which any algorithm can be parameterized by a collection of variant subroutines to be generated automatically. By judicious choice of subroutines (and their associated function signatures), the framework can make effective use of Genetic Programming to tune the algorithm to some target dis-

tribution. Since the supporting algorithm skeleton can be arbitarily complex, this allows greater scalability than can be achieved by the naïve appliciation of Genetic Programming. In implementation terms, TEMPLAR makes creation of generative hyper-heuristics a more procedural matter of GP parameter tuning. We described how to create a 'Hyper-quicksort' and showed the effectiveness of the approach by optimizing for power consumption. Regarding future work, TEMPLAR currently uses EpochX [17] as the GP backend. It would also be desirable to support PushGP and ECJ as alternatives. This may be difficult because of their 'configuration file' based-approach, but if successful would allow direct comparison of their performance as a generative hyper-heuristic mechanism.

Although this approach has proven successful, the imprecision of the power measurement approach used in our experiments imposes an undesirable constraint on experimentation—algorithms must be run numerous times within a single measurement. This greatly increases the time required to run the hyper-heuristic. In future work it would be advantageous if a bytecode or opcode power model was developed for use with an execution trace. Such an implementation would allow highly precise modelling of power consumption, reducing the time required for optimizations and potentially improving results.

References

1. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. and Reliab. 24(3), 219–250 (2014)
2. Bekaroo, G., Bokhoree, C., Pattinson, C.: Power measurement of computers: analysis of the effectiveness of the software based approach. Int. J. Emerg. Technol. Adv. Eng. 4(5), 755–762 (2014)
3. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: a framework for architectural-level power analysis and optimizations. In: 27th Annual International Symposium on Computer Architecture, ISCA '00. pp. 83–94. ACM (2000)
4. Burke, E.K., Hyde, M.R., Kendall, G., Ochoa, G., Özcan, E., Woodward, J.R.: A classification of hyper-heuristic approaches. In: Gendreau, M., Potvin, J.Y. (eds.) Handbook of Metaheuristics, International Series in Operations Research and Management Science, vol. 146, pp. 449–468. Springer, New York (2010)
5. Burke, E.K., Hyde, M.R., Kendall, G., Woodward, J.R.: Automating the packing heuristic design process with genetic programming. Evol. Comput. 20(1), 63–89 (2012)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1995)
7. Hoare, C.A.R.: Algorithm 64: Quicksort. Commun. ACM 4(7), 321 (1961)
8. Hoos, H.H.: Computer-aided design of high-performance algorithms. Tech. Rep. TR-2008-16, University of British Columbia, Department of Computer Science (2008)
9. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
10. López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M.: The irace package, iterated racing for automatic algorithm configuration. Tech. Rep. TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium (2011)

11. Lucas, S.M.: Exploiting reflection in object oriented genetic programming. In: Keijzer, M., O'Reilly, U.M., Lucas, S., Costa, E., Soule, T. (eds.) EuroGP 2004, LNCS, vol. 3003, pp. 369–378. Springer, Heidelberg (2004)
12. Luke, S.: The ECJ Owner's Manual, zeroth edn., online version 0.2 edn. (Oct 2010), `http://www.cs.gmu.edu/~eclab/projects/ecj`
13. Mascia, F., López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T.: Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. Comput. Oper. Res. 51, 190–199 (2014)
14. McIlroy, M.D.: A killer adversary for quicksort. Softw. Pract. Exp. 29(4), 341–344 (1999)
15. Musser, D.R.: Introspective sorting and selection algorithms. Softw. Pract. Exp. 27(8), 983–993 (1997)
16. Noureddine, A., Bourdon, A., Rouvoy, R., Seinturier, L.: Runtime monitoring of software energy hotspots. In: 27th IEEE/ACM International Conference on Automated Software Engineering 2012. pp. 160–169. IEEE (2012)
17. Otero, F., Castle, T., Johnson, C.: EpochX: genetic programming in Java with statistics and event monitoring. In: GECCO '12. pp. 93–100. ACM, New York (2012)
18. Rothlauf, F., Oetzel, M.: On the locality of grammatical evolution. In: Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A. (eds.) EuroGP 2006, LNCS, vol. 3905, pp. 320–330. Springer, Heidelberg (2006)
19. Ryan, C., Collins, J.J., O'Neill, M.O.: Grammatical evolution: evolving programs for an arbitrary language. In: EuroGP 1998, LNCS, vol. 1391, pp. 83–96. Springer, Heidelberg (1998)
20. Spector, L., Harrington, K., Helmuth, T.: Tag-based modularity in tree-based genetic programming. In: GECCO 2012. pp. 815–822. ACM, New York (2012)
21. Spector, L., Klein, J., Keijzer, M.: The Push3 execution stack and the evolution of control. In: GECCO 2005. pp. 1689–1696. ACM, New York (2005)
22. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. 25(2), 101–132 (2000)
23. White, D.R.: Genetic programming for low-resource systems. Ph.D. thesis, University of York (2009)
24. Woodward, J.R., Swan, J.: Why classifying search algorithms is essential. In: International Conference on Progress in Informatics and Computing (2010)
25. Woodward, J.R., Swan, J.: Automatically designing selection heuristics. In: GECCO 2011. pp. 583–590. ACM, New York (2011)
26. Woodward, J.R., Swan, J.: The automatic generation of mutation operators for genetic algorithms. In: GECCO 2012. pp. 67–74. ACM, New York (2012)
27. Woodward, J.R., Swan, J.: Template method hyper-heuristics. In: GECCO 2014. pp. 1437–1438. ACM, New York (2014)