Hyper-quicksort: energy efficient sorting via the TEMPLAR framework for Template Method Hyper-heuristics

jerry.swan@cs.stir.ac.uk
nathan.burles@york.ac.uk
Motivation

**Scalability** remains an issue for program synthesis:

- We don’t yet know how to generate sizeable algorithms from scratch.
- *Generative* approaches such as *GP* still work best at the scale of *expressions* (though some recent promising results [6]).
- *Formal* approaches require a strong *mathematical background*.
- ...but *human ingenuity already* provides a vast repertoire of *specialized algorithms*, usually with known asymptotic behaviour.

Given these limitations, how can we best use *generative hyper-heuristics* to *improve* upon *human-designed algorithms*?
The ‘Template Method’ Design Pattern [1] divides an algorithm into a fixed skeleton with one or more variant parts. The fixed parts orchestrate the behaviour of the variant parts. Example: Quicksort performance depends on the quality of the pivot, so we can treat the pivot function as a variant part:

```java
DoubleArray qsort(DoubleArray arr) {
    double pivot = pivotFn(arr);
    // ^^^ pivotFn can be varied generatively
    return qsort(arr.filter(< pivot))
    ++ arr.filter(== pivot)
    ++ qsort(arr.filter(> pivot));
}
```
Template Method Hyper-heuristics [10]

- So if we can express an algorithmic framework in template method terms, then we can *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*, this means that they are *not subject to the ‘No Free Lunch’ theorem* [8], since the distribution over problem instances is *biased away from uniform* by the training set.
- Successfully demonstrated this approach to learn more effective GA selection and mutation operators [11, 9].
Generative hyper-heuristics can be specified by:

- A list of **variation points** describing the parts of the algorithm to be automatically generated.
- An **algorithm template** expressing the algorithm skeleton. The template produces a *customized version of the algorithm* from *automatically-generated implementations* of the variation points.
- A **fitness function** to evaluate the customized algorithm.
- An **algorithm factory** that *searches the space of variation points* to produce an *optimized version of the algorithm*. 
A functional description

For algorithm with function signature \( I \rightarrow O \):

- \( VP : (I_1 \rightarrow O_1) \times (I_2 \rightarrow O_2) \times \ldots \times (I_n \rightarrow O_n) \).
- Template : \( VP \rightarrow (I \rightarrow O) \).
- Fitness : \( (I \rightarrow O) \rightarrow V \).
- Factory : \( VP \times \text{Template} \times \text{Fitness} \rightarrow (I \rightarrow O) \).
Why a Framework?

Generative HH are *laborious to implement* on a per-case basis, but *non-trivial to generalize*:

- The Factory is typically implemented via GP and is invoked repeatedly . . .
- . . . but popular GP implementations such as ECJ [3] and PushGP [7] *expect to be the ‘top’ of the system* . . .
- . . . hence are not easy to use for generative hyper-heuristics.
- Fitness of one VP depends on the other VPs, so *some fiddly software engineering is required* to enable ‘dependency inversion’.
- *Heterogeneous signatures of VPs* needs special handling to retain any notion of type-safety.
- To prevent overfitting, *cross-validation should be built-in to the fitness function by default.*
Interlude - higher-order functions in Java

```java
interface Fun1<Arg, Result> {
    Result apply(Arg arg);
}

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

// We can then use functions as parameters
// and return values:
Fun1<Int, String> compose(Fun1<Int, Double> f, Fun1<Double, String> g) {
    return new Fun1<Int, String>() {
        String apply(Int arg) {
            return g.apply(f.apply(arg));
        }
    };
}
```
Core `TEMPLAR` classes

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

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

    ProgramList run(FitnessCases<I, O> cases,
                     LossFn<O> lossFn) {
        ...
    }
}
```
Trivial example - ‘Identity’ template

Just executes the generated program for the (sole) variation point:

class IdentityTemplate implements AlgTemplate<Double, Double> {

    public Fun1<Double, Double>
    makeAlg(ProgramList progs) {
        // Wrap the VP in a function:
        return new Fun1<Double, Double>() {
            Double apply(Double arg) {
                return progs.get(0).execute(arg);
            }
        };
    }
}
The end-user only needs to do this\(^1\):

```java
// 1. Define an AlgTemplate subclass (previous slide).
// 2. Set up the algorithm-specifics:
AlgTemplate<Double, Double> template = new IdentityTemplate();
GPConfig[] vpConfigs = {new RationalFunctionConfig();}
FitnessCases trainingSet = ...
FitnessCases testSet = ...

// 3. Invoke TEMPLAR:
ProgramList bestVPs = Templar.trainAndTest(template, vpConfigs,
trainingSet, testSet,
new RMSLossFn<Double>();
println("best VPs: " + bestVPs);
```

\(^1\)These examples describe all the code you need to write.
Next simplest example - Composition Template

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

- Just follow the above steps for *any* algorithm you wish to optimize.
- We’ll see how easy it is to create ‘Hyper-quicksort’ ...
abstract class PivotFn
extends Fun2<DoubleArray, Integer, Double>
{
    Double apply(DoubleArray a, Integer recursionDepth);
}

class SedgewickPivotFn extends PivotFn {
    // counts the case of sorted
    // (or reverse-sorted) input
    Double apply(DoubleArray a, Integer recursionDepth){
        return median(a.first, a[a.length/2], a.last);
    }
}

Int quicksort(DoubleArray a, PivotFn pivotFn);
// ^ instrumented to return some measure
// of pivotFn fitness (e.g. max recursion depth)
class QuicksortTemplate
implements AlgTemplate<DoubleArray, Int> {
    Fun1<DoubleArray, Int> makeAlg(ProgramList progs) -> {
        PivotFn pivotFn = (DoubleArray a, Int recursionDepth) -> {
            int progResult = progs[0].execute(a.size, recursionDepth);
            int numSamples = min(abs(progResult), a.size);
            return median(randomSample(a, numSamples));
        };
        return (DoubleArray arg) -> Quicksort.sort(arg, pivotFn);
    }
}
HyperQuicksort - Top Level

// 1. Define an AlgTemplate subclass (previous slide).
// 2. Configure GP to generate pivotFn VP:
List<Var> vars = {Var("size"), Var("recursionDepth")};
List<Node> funcSet = {IFn(), 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>());
Wait - there’s more . . .

- Manual creation of GP nodes for function sets on custom solution representations (e.g. Timetable, RoutePlan, AntTrail etc) is tedious.
- Following [2], Templar.FunctionSetGenerator uses reflection to **automatically build a function set** from any Java object.
- By this means, a hyper-heuristic for *Iterated Local Search over bitstrings* was up and running from scratch **in under 20 minutes**
- By following the above steps, it’s quick and easy to create a template for **your favourite algorithm here**.
- All you need now is **lots of CPU time** . . .
Experiment - Setup

- EpochX for GP.
  - Monitors execution time and processor utilisation to estimate power consumption.
  - Non-deterministic (e.g. other processes), and accuracy limited by platform (up to nanosecond).
  - Multiple arrays need to be sorted for each measurement (100).
  - Oracular pivot function.
Experiment - Pipeorgan Distribution [4]

**Training set** size: 70 (* 100).

**Testing set** size: 100 (at 9 different array lengths, * 1000)._
Results

![Graph showing results of various sorting algorithms. The x-axis represents DoubleArray size (log₂ scale), and the y-axis represents Joules (log₂ scale). The graph compares Mid, Sedgewick, Random, and Hyper-Quicksort methods.](image-url)
## Results - Table

<table>
<thead>
<tr>
<th>Array size</th>
<th>Middle index</th>
<th>Sedgewick</th>
<th>Hyper-quicksort (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>J</td>
<td>p</td>
<td>e</td>
</tr>
<tr>
<td>8</td>
<td>0.191</td>
<td>7.46e-32</td>
<td>0.981</td>
</tr>
<tr>
<td>16</td>
<td>0.296</td>
<td>1.20e-30</td>
<td>0.971</td>
</tr>
<tr>
<td>32</td>
<td>0.651</td>
<td>8.13e-32</td>
<td>0.980</td>
</tr>
<tr>
<td>64</td>
<td>1.366</td>
<td>4.80e-33</td>
<td>0.990</td>
</tr>
<tr>
<td>128</td>
<td>3.505</td>
<td>4.80e-33</td>
<td>0.990</td>
</tr>
<tr>
<td>256</td>
<td>8.175</td>
<td>4.14e-33</td>
<td>0.991</td>
</tr>
<tr>
<td>512</td>
<td>19.777</td>
<td>4.33e-34</td>
<td>0.998</td>
</tr>
<tr>
<td>1024</td>
<td>62.961</td>
<td>2.52e-34</td>
<td>1.000</td>
</tr>
<tr>
<td>2048</td>
<td>198.438</td>
<td>2.52e-34</td>
<td>1.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array size</th>
<th>Random index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>J</td>
</tr>
<tr>
<td>8</td>
<td>0.446</td>
</tr>
<tr>
<td>16</td>
<td>0.410</td>
</tr>
<tr>
<td>32</td>
<td>0.967</td>
</tr>
<tr>
<td>64</td>
<td>1.708</td>
</tr>
<tr>
<td>128</td>
<td>5.221</td>
</tr>
<tr>
<td>256</td>
<td>9.269</td>
</tr>
<tr>
<td>512</td>
<td>27.685</td>
</tr>
<tr>
<td>1024</td>
<td>41.245</td>
</tr>
<tr>
<td>2048</td>
<td>111.894</td>
</tr>
</tbody>
</table>
Experiment - Conclusions

- P-values (Mann-Whitney U-test) and effect sizes (Vargha-Delaney $\hat{A}_{12}$) clearly show Hyper-quicksort provides significant improvement on pipeorgan distributions.
- Intermediate results showed that minimal recursion doesn’t always equate to minimal power consumption, as pivot function becomes more demanding.
- Imprecision and non-determinism of power measurement imposes time constraints on experimentation.
Conclusion and Future Work

- Algorithms can be decomposed into *templates* consisting of a fixed skeleton and a collection of variant components.
- By judicious choice of function signatures, we can use generative methods (GP etc) to create variant components that are tuned to some target distribution.
- In implementation terms, **TEMLAR** makes generative HH for any algorithm a matter of GP parameter tuning.
- New methods of power consumption modelling are in development...
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

*Design patterns: elements of reusable object-oriented software.*


Simon M. Lucas.

References II

Sean Luke, Liviu Panait, Gabriel Balan, and Et.

M. Douglas McIlroy.
A killer adversary for quicksort.

Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier.
Runtime monitoring of software energy hotspots.


References V

John R. Woodward and Jerry Swan.
Template method hyper-heuristics.

John Robert Woodward and Jerry Swan.
Automatically designing selection heuristics.
In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO ’11, pages 583–590, New York, NY, USA, 2011. ACM.