TEMPLAR - a framework for Template Method Hyper-heuristics
IEEE CEC, Sendai June 2015

jerry.swan@york.ac.uk
nathan.burles@york.ac.uk

Motivation - 1

Scalability remains an issue for program synthesis:

- We don't yet know how to generate sizeable algorithms from scratch.
- **Formal** approaches require a strong *mathematical background*.
- **Generative** approaches such as *GP* still work best at the scale of *expressions* (though some recent promising results [8]).

Motivation - 2

- *Human ingenuity* **already** provides a vast repertoire of *specialized algorithms*, usually with known asymptotic behaviour.
- So how can we best use **generative techniques** to **improve** upon **human-designed algorithms**?
- The approach discussed here is the application of the *Template Method* ‘Design Pattern’ [2] to *generative hyper-heuristics* (‘heuristics to generate heuristics’ [1]).

The Template Method Pattern

- The **'Template Method' Design Pattern** [2] divides an algorithm into a *fixed skeleton* with one or more *variant* parts.
- The *fixed* parts *orchestrate the behaviour* of the *variant* parts.
- Running Example: Quicksort performance depends on the quality of the *pivot*, so we can treat the *pivot function* as a *variant part*:

```
DoubleArray qsort(DoubleArray arr) {
    double pivot = pivotFn(arr);
    return
      qsort( arr.filter(< pivot), depth+1 )
     ++ arr.filter(== pivot)
     ++ qsort( arr.filter(> pivot), depth+1 );
}
```

Template Method Hyper-heuristics [13]

- 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* [11], 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 [14, 12].

A framework for generative hyper-heuristics

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 : $VP \times Template \times Fitness \rightarrow (I \rightarrow O)$.

Why a Framework? - 1

There are **many papers** on *selective* HH, but not so many for *generative*. 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 [4] and PushGP [9] *expect to be the 'top' of the system* . . .
- . . . hence are not easy to use for generative hyper-heuristics.

Why a Framework? - 2

- 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, support for cross-validation should be built-in.

Interlude - higher-order functions in Java

```
interface Fun1<Arg, Result> {
  Result apply(Arg arg);
}
interface Proc1<Arg> implements Fun1<Arg, Void> {
  void 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 (Int x) -> g( f(x) );
}
```

Core TEMPLAR classes

```
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) { ... }
}
```

Simple example - ‘Identity’ template

Executes the generated program for the (sole) variation point:

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

Using TEMPLAR

- Write algorithm template (previous slide).
- Set up the algorithm-specifics:

```
AlgTemplate<Double, Double> template =
  new IdentityTemplate();
GPConfig[] vpConfigs =
  { new RationalFunctionConfig(); }
FitnessCases trainingSet = ...
FitnessCases testSet = ...
```

- Invoke TEMPLAR:

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

- That's *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 = (Int arg) -> progs.get(0).execute(arg);
      g = (Double arg) -> progs.get(1).execute(arg);
      // this template just composes
      // the two variant programs ...
      return compose(f,g);
  };
};
```

HyperQuicksort

- We'll create 'HyperQuicksort' in 6 easy steps.
- The same process will work for *any* algorithm you wish to optimize!

HyperQuicksort - Step 1

- Start with regular Quicksort.

```
DoubleArray
qsort( DoubleArray arr, int depth ) {
    double pivot =
      median(arr.first,arr[arr.length/2],arr.last);
    return
      qsort( arr.filter(< pivot), depth+1 )
     ++ arr.filter(== pivot)
     ++ qsort( arr.filter(> pivot), depth+1 );
}
```

HyperQuicksort - Step 2

- Identify the variant part(s).

```
DoubleArray
qsort( DoubleArray arr, int depth ) {
    double pivot =
      median(arr.first ,arr[arr.length/2],arr.last);
    return
      qsort( arr.filter(< pivot), depth+1 )
     ++ arr.filter(== pivot)
     ++ qsort( arr.filter(> pivot), depth+1 );
}
```

HyperQuicksort - Step 3

- Factor out the variant part(s).

```
DoubleArray
qsort(DoubleArray arr, int depth, PivotFn pivotFn) {
    double pivot = pivotFn(arr, depth);
    return
      qsort( arr.filter(< pivot), depth+1 )
     ++ arr.filter(== pivot)
     ++ qsort( arr.filter(> pivot), depth+1 );
}
```

HyperQuicksort - Step 4

- Create a ‘generative’ version of each variant

```
PivotFn
makePivotFn(GPProgram variant) {

// Wrap the GP program in Java code
// to make a PivotFn
return (DoubleArray a,int depth) -> {
  int progResult = variant.exec(a.length,depth);
  int numSamples = min(abs(progResult),a.length);
  return median(randomlySample(a, numSamples));
};
}
```

HyperQuicksort - Step 5: Write the algorithm template

```
class QuicksortTemplate implements AlgTemplate<
    DoubleArray> {

  public Proc<DoubleArray>
  makeAlg(ProgramList variants) {

   // 1. Make a PivotFn from the GPProgram:
   PivotFn pivotFn = makePivotFn(variants.get(0));

    // 2. Use this PivotFn to make a custom Quicksort:
    return (DoubleArray a) -> Quicksort.sort(a, pivotFn);
    // ^ return a version of Quicksort which uses
    // this PivotFn. This is then evaluated
    // by the hyper-heuristic layer.
  }
}
```

HyperQuicksort - Step 5: Write the algorithm template

```
class QuicksortTemplate implements AlgTemplate<
    DoubleArray> {

  public Proc<DoubleArray>
  makeAlg(ProgramList variants) {

   // 1. Make a PivotFn from the GPProgram:
   PivotFn pivotFn = makePivotFn(variants.get(0));

    // 2. Use this PivotFn to make a custom Quicksort:
    return (DoubleArray a) -> Quicksort.sort(a, pivotFn);
    // ^ return a version of Quicksort which uses
    // this generated PivotFn. This is then evaluated
    // by the hyper-heuristic layer.
  }
}
```

HyperQuicksort - Step 6

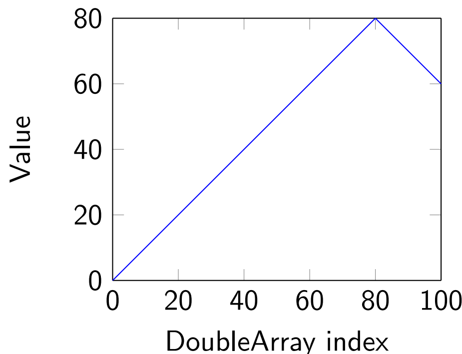
- Configure GP to generate the pivotFn

```
List<Var> vars = {Var("arraySize"),Var("depth")};
List<Node> funcSet = {IfFn(),LessFn(),AddFn(),...};
GPParams params = ...; // crossover, selection etc
GPConfig vpConfigs={new GPConfig(funcSet,vars,
    params);}
```

- and Invoke Templar

```
AlgTemplate<DoubleArray> template = new
    QuicksortTemplate();
FitnessCases trainingSet = ...
FitnessCases testSet = ...
Templar.trainAndTest(template,vpConfigs,trainingSet
    ,testSet,new MyFitnessFn<DoubleArray>());
```

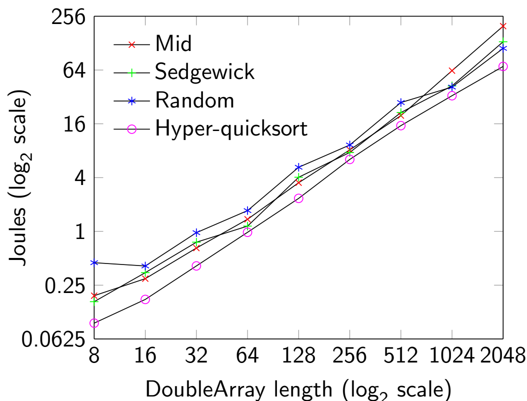
Dataset - ‘Pathological’ Pipeorgan Distribution [5]



Fitness measure - Energy consumption

- The JALEN [6] was used to measure energy consumption.
- Estimates power consumption as a function of execution time and processor utilisation.
- JALEN normally runs as external process.
- In order to obtain greater accuracy, we created a custom version invocable from within a host JVM.

Results



Training set size: 70 (each containing 100 arrays, of length 100)
Testing set size: 100 (each containing 1000 arrays of lengths 8, . . . , 2048

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 [3], 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* . . .

Conclusion and Future Work - 1

- 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.
- Currently uses EpochX [7] as the GP backend. Although fiddly (because of their 'configuration file' approach), it *should* be possible to support ECJ and PushGP as alternatives, allowing direct comparison of performance . . .

Conclusion and Future Work - 2

- Facilitates *Embedded Dynamic Improvement Programming*:
  - Embeddable within a mainstream programming language (in this case Java).
  - Supported by the *type-system* of that language, i.e. can operate on *objects* as *first class entities*.
- 'Offline' training can actually take place:
  - At the point of embedding.
  - Lazily (i.e. on first use).
  - Periodically or asynchronously.

References I

E.K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward.
A Classification of Hyper-heuristic Approaches, chapter Handbook of Meta-Heuristics, pages 449–468.
Kluwer, 2010.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design patterns: elements of reusable object-oriented software.

Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

References II

Simon M. Lucas.
Exploiting reflection in object oriented genetic programming.
In Maarten Keijzer, Una-May O'Reilly, Simon Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming*, volume 3003 of *Lecture Notes in Computer Science*, pages 369–378. Springer Berlin Heidelberg, 2004.

Sean Luke, Liviu Panait, Gabriel Balan, and Et.
ECJ 16: A Java-based Evolutionary Computation Research System, 2007.

M. Douglas McIlroy.
A killer adversary for quicksort.
Softw., Pract. Exper., 29(4):341–344, 1999.

References III

Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier.
Runtime monitoring of software energy hotspots.
In *27th IEEE/ACM Int. Conf. on Autom. Softw. Eng. 2012*, pages 160–169. IEEE, 2012.

Fernando Otero, Tom Castle, and Colin Johnson.
Epochx: Genetic programming in java with statistics and event monitoring.
In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, pages 93–100, New York, NY, USA, 2012. ACM.

References IV

Lee Spector, Kyle Harrington, and Thomas Helmuth.
Tag-based modularity in tree-based genetic programming.
In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 815–822, New York, NY, USA, 2012. ACM.

Lee Spector, Jon Klein, and Maarten Keijzer.
The push3 execution stack and the evolution of control.
In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on*

References V

Genetic and evolutionary computation, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

Jerry Swan and Nathan Burles.
Templar: a framework for template-method hyper-heuristics.
In Penousal Machado, Malcolm I. Heywood, James McDermott, Mauro Castelli, Pablo Garcia-Sanchez, Paolo Burelli, Sebastian Risi, and Kevin Sim, editors, *18th European Conference on Genetic Programming*, volume 9025 of *LNCS*, pages 205–216, Copenhagen, 8-10 April 2015. Springer.

John Woodward and Jerry Swan.
Why classifying search algorithms is essential.
In *2010 International Conference on Progress in Informatics and Computing. (PIC-2010)*, 2010.

References VI

John R. Woodward and Jerry Swan.
The automatic generation of mutation operators for genetic algorithms.
In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO Companion '12, pages 67–74, New York, NY, USA, 2012. ACM.

John R. Woodward and Jerry Swan.
Template method hyper-heuristics.
In *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion*, GECCO Comp '14, pages 1437–1438, New York, NY, USA, 2014. ACM.

References VII

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.