Template Method Hyper-heuristics

John Woodward
Computing Science and Mathematics
University of Stirling
FK9 4LA Scotland UK
john.woodward@cs.stir.ac.uk

Jerry Swan
Computing Science and Mathematics
University of Stirling
FK9 4LA Scotland UK
jerry.swan@cs.stir.ac.uk

1. PROBLEM STATEMENT

The optimization literature is awash with metaphorically-inspired metaheuristics and their subsequent variants and hybridizations. This results in a plethora of methods, with descriptions that are often polluted with the language of the metaphor which inspired them [8]. Within such a fragmented field, the traditional approach of manual 'operator tweaking' makes it difficult to establish the contribution of individual metaheuristic components to the overall success of a methodology.

Irrespective of whether it happens to best the state-of-the-art, such 'tweaking' is so labour-intensive that does relatively little to advance scientific understanding. In order to introduce further structure and rigour, it is therefore desirable to not only to be able to specify entire *families* of metaheuristics (rather than individual metaheuristics), but also be able to generate and test them. In particular, the adoption of a *model agnostic* approach towards the generation of metaheuristics would help to establish which metaheuristic components are useful contributors to a solution.

2. THE SOLUTION

In [7], Krasnogor shows how families of algorithms can be specified in terms of the well-known *template method pattern* [4]. The template-method is a definition of an algorithm 'skeleton' within which the interaction of a collection of abstractly-specified components is orchestrated. Components can be thought of as classes/procedures/functions in object-oriented/procedural functional programming terms. The hard-coded backbone restricts the set of possible algorithms expressed by the template, while the abstract component specification allows for variant implementations within the specified constraints.

Given the requirement to not only specify the members of a family of metaheuristics, but also to *generate* them, the behavioural constraint imposed on the components by the template method is of relevance for two main reasons:

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

ACM 978-1-4503-2881-4/14/07…$15.00.
http://dx.doi.org/10.1145/2598394.2609843 ...$15.00.

Firstly, automatic algorithm synthesis methods such as Genetic Programming (GP) are not yet capable of evolving complete programs from scratch (i.e. those which require nested loops, indexed memory and other control structures such as switch statements). In current practice, GP is best employed to evolve *expressions*, i.e. small mathematical functions or logical expressions. To facilitate the creation of larger programs, the fixed part of a hyper-heuristic template can be seen as providing the orchestrating control structures. A simple example would be with the skeleton providing an iterative procedure (e.g. a for-loop with appropriate bounds), and the variant implementation of the iteration body providing by GP (e.g. the body of the for-loop). This is true of many of the examples in Section 4.

Secondly, Template-Method Hyper-heuristics allows a metaheuristic to be automatically tuned to the set of problems of specific interest. A consequence of the No Free Lunch (NFL) theorems [6] is that no 'universal' optimizer exists, and hence metaheuristics must be specialised on a problem-specific basis. One approach to this to use is meta-learning [5] to adapt a heuristic to the problem instances to which it will be exposed. This employs the traditional machine-learning style of splitting a set of problem instances into disjoint training and testing sets [1] as practiced in [10, 9]. A consequence of NFL is that metaheuristics should not be 'designed' in isolation from a problem domain but designed (and most importantly their performance cited) in the context of some representative problem instances.

As a concrete example of an algorithm template, consider a framwork for a simple evolutionary algorithm. In generative terms, for population P and population history H (the latter being a list of successive populations), we can consider the framework to be parameterized by five variant methods:

$$\begin{aligned} initialization &: Void \rightarrow P \\ selection &: P \times H \rightarrow P \\ variation &: P \times H \rightarrow P \\ succession &: P \times H \rightarrow P \\ termination &: H \rightarrow Bool \end{aligned}$$

In general, we can either elect to search the entire design space defined by this framework (e.g. by simultaneously searching for good versions of all the variant components) or else fix one or more components and generate the others. Section 4 gives examples where the selection operator and mutation operators are evolved.

3. CONSEQUENCES

- 'Tuned to the problem'. If the heuristic is designed

```
procedure evolve
begin
  pop = initialization()
  history = []
  repeat
    parents = selection( pop, history )
    offspring = variation( parents, history )
    pop = succession( offspring, history )
    history = history.append( pop )
  until termination( history )
end
```

in the context of representative problem instances, the resulting heuristic will be expected to perform well on problem instances drawn from a similar probabality distribution as the problem instances in the training phase. Equally there will be a degradation in performance if the training does not reflect the testing [3].

- 'Algorithm agnostic'. Given a palette of metaheuristic components, the ones that are best suited to the problem instances under consideration will have a greater chance of appearing in the resulting heuristic.
- 'Human competitive performance'. The heuristics produced by this method will produce results which are competitive with already existing human designed heuristics. Indeed, we have the choice of initialzing the search process with already existing heuristics and therefore the design process can be thought of as *improving* algorithms, rather than inventing them from *scratch*.
- 'Decreased human effort'. There is a financial cost involved in developing a heuristic. If we continue to manually design heuristics, this price will increase inline with inflation. However, if we harness automatic design, this price is likely to fall as processor speeds increase roughly with Moore's law. Therefore, if this method of automatic design that we advocate is adopted [2], this monetray cost is expected to fall in the long term.

4. EXAMPLES

Evolving Mutation Operators For Genetic Algorithms Two well-known GA mutation operators on bit-strings are one-point (which alters a single bit in the bit-string) and uniform mutation (which alters all bits in the bit-string with a fixed probability). The well-known GA template can be parameterized by its mutation operator, which can be considered as a procedure with type signature $mutation : \mathbb{B}^n \rightarrow \mathbb{B}^n$. In [9] GP is used to automatically generate a mutation operator that outperforms both one-point and uniform mutation. Key to the success of this approach is setting up the orchestrating template so that operators known to be effective are easily generated by GP or can be encoded easily by a human programmer (e.g. the ability to express one-point and uniform mutation as degenerate cases).

Evolving Selection Operators Similarly, as with the case of mutation abovef selection operators are commonly implemented as either fitness-proportional (which assigns a selection value in proportion to absolute fitness f) or rank selection (which assigns a selection value according to its rank index r in the sorted population of bit-strings). One generalization of these two selection operators is a selection function $selection : \mathbb{R}^2 \rightarrow \mathbb{R}$ which takes as arguments the absolute fitness f and the rank index r and assigns a selection value to each bit-string in the population. Clearly, by returing just the first argument we have fitness-proportional selection, and by returning just the second argument we have rank selection. In [10], GP (with f and r in its terminal set) is used to develop a new selection operator that outperforms both of these two standard selection operators.

5. REFERENCES

[1] Mauro Birattari, Mark Zlochin, and Marco Dorigo. Towards a theory of practice in metaheuristics design: A machine learning perspective. *ITA*, 40(2):353–369, 2006.

[2] Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. Exploring hyper-heuristic methodologies with genetic programming. In Christine L. Mumford and Lakhmi C. Jain, editors, *Computational Intelligence*, volume 1 of *Intelligent Systems Reference Library*, chapter 6, pages 177–201. Springer, 2009.

[3] Edmund K. Burke, Matthew R. Hyde, Graham Kendall, and John Woodward. Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In Dirk Thierens et al., editor, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1559–1565, London, 7-11 July 2007. ACM Press.

[4] 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.

[5] C. Giraud-Carrier and F. Provost. Toward a Justification of Meta-learning: Is the No Free Lunch Theorem a Show-stopper? In *Proceedings of the ICML-2005 Workshop on Meta-learning*, pages 12–19, 2005.

[6] Christian Igel. No free lunch theorems: Limitations and perspectives of metaheuristics, 2014.

[7] N. Krasnogor. Memetic algorithms, 2009.

[8] Kenneth Sörensen. Metaheuristics–the metaphor exposed. 2012.

[9] John R. Woodward and Jerry Swan. The automatic generation of mutation operators for genetic algorithms. In Gisele L. Pappa, John Woodward, Matthew R. Hyde, and Jerry Swan, editors, *GECCO 2012 2nd Workshop on Evolutionary Computation for the Automated Design of Algorithms*, pages 67–74, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[10] John Robert Woodward and Jerry Swan. Automatically designing selection heuristics. In Gisele L. Pappa, Alex A. Freitas, Jerry Swan, and John Woodward, editors, *GECCO 2011 1st workshop on evolutionary computation for designing generic algorithms*, pages 583–590, Dublin, Ireland, 12-16 July 2011. ACM.