HYPERION^2
A toolkit for {meta-, hyper-} heuristic research

Alexander E.I. Brownlee
Division of Computer Science and Mathematics
University of Stirling
Stirling, UK
sbr@cs.stir.ac.uk

Jerry Swan
Division of Computer Science and Mathematics
University of Stirling
Stirling, UK
jsw@cs.stir.ac.uk

Ender Özcan
School of Computer Science
University of Nottingham
Nottingham, UK
ender.ozcan@nottingham.ac.uk

Andrew J. Parkes
School of Computer Science
University of Nottingham
Nottingham, UK
andrew.parkes@nottingham.ac.uk

ABSTRACT

In order for appropriate meta-heuristics to be chosen and tuned for specific problems, it is critical that we better understand the problems themselves and how algorithms solve them. This is particularly important as we seek to automate the process of choosing and tuning algorithms and their operators via hyper-heuristics. If meta-heuristics are viewed as sampling algorithms, they can be classified by the trajectory taken through the search space. We term this trajectory a *trace*. In this paper, we present HYPERION^2, a Java^{TM} framework for meta- and hyper- heuristics which allows the analysis of the trace taken by an algorithm and its constituent components through the search space. Built with the principles of interoperability, generality and efficiency, we intend that this framework will be a useful aid to scientific research in this domain.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*; D.1.5 [**Object-oriented Programming**]

Keywords

Metaheuristics, Hyper-heuristics, analysis, experimental framework, search space

1. INTRODUCTION

Metaheuristics and related methods are reaching maturity as a research domain. An extensive literature covers a large number of approaches, from broad algorithm paradigms to individual operators, and over the past couple of decades metaheuristics have increasingly gained acceptance for application to real-world problems in industry [1]. However, in many respects there is still something of an art in terms of finding the right algorithm for the right problem. Much research still focuses on developing tweaks to existing algorithms and operators to suit new application domains, treating the search process itself as a black box and focussing on metrics such as convergence speed and solution quality in empirical comparisons. Approaches to automating the processes of algorithm selection and tuning have been made, notably hyper-heuristics, which operate in the space of heuristics [2]. However, for such approaches to be effective, it remains critical that we better understand the nature of problems themselves and how algorithms solve them. The ultimate goal is to classify search algorithms and problems [3] so it becomes possible to match a problem with an appropriate algorithm [4].

If metaheuristics are viewed as sampling algorithms [5], they can be classified by the trajectory (which we term a *trace*) that they take through the search space. Understanding the influence of operators on the trace produced by a search will provide essential information for algorithm designers (human or otherwise). Whitebox information about traces (e.g. which bit was flipped, knowing the lineage of individuals in a population) allows the search process to proceed in a more informed manner [6], and can be seen as the principled incorporation of (both problem and solution) domain information in a manner than has proved to be effective (e.g. as evidenced by the tabu search metaheuristic [7]). It is claimed that machine learning approaches are needed to move beyond the 'persistent operator tweaking' that plagues metaheuristic design [8, 9, 10] and options for learning are clearly facilitated by the availability of such trace information. In particular, the incorporation of traces is vital for the creation of an 'interoperable universal signature' for metaheuristics, something which is clearly necessary for automatic hybridization. Further, detailed trace information can be used to generate a proxy for fitness in genetic programming and hyper-heuristics, where traces model algorithm behaviour [10, 11].

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

ACM 978-1-4503-2881-4/14/07 ...$15.00.
http://dx.doi.org/10.1145/2598394.2605687.

It is in this context that we propose HYPERION^2, a Java^{TM} framework to aid scientific research, allowing sound analysis of meta- and hyper- heuristics by providing a general approach for describing the trace taken by an algorithm and its constituent components through the search space.

HYPERION^2 is a research toolkit for non-global optimization that aims to maximize expressive power whilst maintaining ease-of-use. Based on [12], it has evolved through several incarnations and attempts to incorporate the best elements of solution-domain frameworks such as [13], [14] and most notably [15].

The specific design goals of HYPERION^2 include:

1. Promoting interoperability via component interfaces. These interfaces are type-parameterized by solution-state representation and objective-value (the latter allowing for multi-objective formulations).
2. Allow rapid prototyping of meta- and hyper- heuristics, with the potential to use the same source code in either case.
3. Provide generic templates for a variety of local search and evolutionary computation algorithms, allowing popular variants to be readily constructed from pre-existing components.
4. Facilitate the construction of novel meta- and hyper-heuristics by hybridization (via interface interoperability) or extension (subtype polymorphism).
5. 'Only pay for what you use' — a design philosophy that attempts to ensure that generality doesn't necessarily imply inefficiency.

In this paper we present an overview of the framework's core interfaces, discuss the key concepts and give some implementation examples including the well-known 'onemax' problem [16], random search and steepest-ascent hillclimbing.

2. RELATED WORK

There are many existing frameworks for construction and analysis of metaheuristics: [17] cites several current examples. The HeuristicLab framework [18] also implements a generic approach to algorithm construction. Two popular frameworks, ECJ [19] and EpochX [20] offer event driven approaches to analyse the progress of evolution, but none offers the level of control, genericity or level of granularity that are present in HYPERION^2.

Besides HYPERION^2 and the original HYPERION, HY-FLEX [21] and hMod [22] are (to the authors' knowledge) the only other existing frameworks for hyper-heuristics. Hy-Flex is concerned with building reusable elements for common problem domains, and recent work [23, 24] has seen the development of additional tools for generating hyper-heuristics in HY-FLEX.

CluPaTra [25, 26] and FloTra [27] are approaches which make use of trajectories in parameter tuning.

3. CORE INTERFACES

The basic interoperability mechanism in HYPERION^2 follows from the observation that a heuristic (equivalently meta- or hyper- heuristic hereafter) is essentially defined by its solution trajectory (*trace*) in the state-space graph. A heuristic h_S over some solution-state representation of type S (e.g. S might be a bitstring, permutation, vector of reals, graph etc) is recursively defined as a function:

$$h_S : Trace_S \rightarrow Trace_S$$

where $Trace_S = (h_S, S)^*$

i.e. a Trace is a list of zero or more (heuristic, solution-state) pairs, with each pair representing a single transition in the state-space.

The implied semantics are that a heuristic h_S takes a trace as input and returns a (possibly extended) trace. Primitive heuristics (such as a bit-flip) extend the trace by a single move, metaheuristics have the potential to extend it by multiple moves. The explicit provision of the input trace means that history-sensitive metaheuristics (e.g. tabu-search, EDA approaches and adaptive variants of simulated annealing) can be implemented with a minimum of additional effort. In addition, the inclusion of previously-applied heuristics in the trace information means that domain-specific information (e.g. knowledge of algebraic relationships between heuristics) can be used to inform metaheuristics in a domain-neutral fashion.

As shown in Listing 1, the corresponding JAVA API is rather more straightforward than the above definition might suggest. Trace<S> is essentially a list of Move<S>, the latter being the pairing of a heuristic with the target state which results from the application of that heuristic, as described above. The additional Random parameter supplied to the apply method of Heuristic allows for the provision of a better random number generator than JAVA's builtin 48-bit linear-congruential implementation, this being essential for fair sampling of large state-spaces.

Other elementary components of the framework are given in Listing 2. ValueFn<S,Value> defines the evaluation function for a state, where Value is a generic type than can include multiobjective representations, with Acceptance and IsFinished defining the interfaces for acceptance and termination criteria respectively.

Another key concept in HYPERION^2 is that of LOCALITY, representing the ubiquitous concept of local search neighbourhood. The HYPERION^2 locality concept is factored into three - ITERABLELOCALITY, GENERATIVELOCALITY and RANDOMACCESSLOCALITY. ITERABLELOCALITY defines some neighborhood of a state, successive elements of which are accessed via the *Iterator* design pattern [28]. GENERATIVELOCALITY provides for the creation of randomly-generated neighbors and RANDOMACCESSLOCALITY allows a neighbor to be accessed via an integer index in O(1) time. The rationale for factoring out these concepts is to reduce the implementation burden for custom neighborhoods. There is explicit support within HYPERION^2 for bit-flip and permutation-swap neighborhoods.

HYPERION^2 also supports compile-time checking for minimization or maximization, through the 'curiously recurring template pattern' [29]. This means that any algorithm implementation can be entirely independent of whether a problem is minimization or maximization. More importantly, this will trap 'wrong direction' mixing of maximization and minimization at compile-time. Listing 3 gives the general implementation of this.

```
public interface Heuristic<S>
{
        public Option< Move< BitString > > apply( Trace< BitString > inputTrace, Random rng )
        public Trace<S> iterate( Trace<S> inputTrace, Random rng );
}

public final class Move<S>
{
        public Heuristic<S> getHeuristic() { /* ... */ }
        public S getTargetState() { /* ... */ }

        // Other details omitted ...
}

public final class Trace<S>
{
        public int length() { /* ... */ }
        public Move<S> getMove( int index ) { /* ... */ }

        // Other details omitted ...
}
```

Listing 1: Hyperion^2 Heuristic interface and associated classes.

```
public interface ValueFn<S,Value>
{
        public Value apply( S state );
}

public interface Acceptance<S>
{
        public Probability apply( Trace<S> trace, Move<S> suggestedMove, double temperature );
}

public interface IsFinished<S>
{
        public boolean apply( Trace<S> trace, long numIterations );
}
```

Listing 2: Core Hyperion^2 interfaces.

```
public interface DirectedValue< Derived >
extends Comparable< Derived > {}

public final class Max< Value extends Comparable< Value > >
implements DirectedValue< Max< Value > > {
        private final Value value;

        public Max( Value value ) {
                this.value = value;
        }

        @Override
        public int compareTo( Max< Value > other ) {
                return value.compareTo( other.value );
        }

        // Other details omitted ...
}

public final class Min< Value extends Comparable< Value > >
implements DirectedValue< Min< Value > > {
        private final Value value;

        public Min( Value value ) {
                this.value = value;
        }

        @Override
        public int compareTo( Min< Value > other ) {
                return -value.compareTo( other.value );
        }

        // Other details omitted ...
}
```

Listing 3: Compile-time checking of min vs max functions using CRTP

```
final class OnemaxValueFn
extends ValueFn<BitString, Double> {
        @Override
        public boolean isMinimizing() { return false; }

        @Override
        public Double apply( final BitString x ) {
                return x.cardinality() / (double)x.length();
        }
}

final class UniformMutation
implements Heuristic<BitString> {
        @Override
        public Option< Move<BitString>> apply( Trace<BitString> t, Random random ) {
                final int bitToFlip = random.nextInt( t.getLastState().length() );
                Heuristic<BitString> h = new BitFlip( bitToFlip );
                return h.apply( t, random );
        }
}

final class OnemaxSolutionFound
implements IsFinished<BitString> {
        @Override
        public boolean apply( Trace<BitString> t, long iteration ) {
                return t.getLastState().cardinality() == t.getLastState().length();
        }
}
```

Listing 4: Concrete subclasses for the onemax problem.

```
public static void runRandomSearch( BitString initialState, Comparator< BitString > compareStates, Random random ) {
        IsFinished< BitString > isFinished = new OnemaxSolutionFound();
        RandomAccessLocality< BitString > locality = new BitFlipLocality( random );

        Acceptance< BitString >acceptance = new AllMoves< BitString >();
        Metaheuristic< BitString > meta = new RandomSearch< BitString >( locality, acceptance, isFinished );

        Trace< BitString > result = meta.iterate( Trace.cons( initialState ), random );

        BitString bestState = result.bestState( compareStates );
        System.out.println( "best:" + bestState );
}
```

Listing 5: Invoking a Random-search metaheuristic.

```
public static void runSteepestAscent( BitString initialState, Comparator<BitString> compareStates, Random random ) {
        RandomAccessLocality< BitString > locality = new BitFlipLocality( random );

        Metaheuristic< BitString > meta = new SteepestAscent< BitString >( locality, compareStates );

        Trace< BitString > result = meta.iterate( Trace.cons( initialState ), random );

        BitString bestState = result.bestState( compareStates );
        System.out.println( "best:" + bestState );
}
```

Listing 6: Invoking a Steepest-ascent metaheuristic.

4. METAHEURISTIC TEMPLATES

A number of local search algorithms are currently provided as an illustration of how to implement and/or customize metaheuristics. These include random search; iterated perturbation; iterated local search; various flavours of tabu search and a simple simulated annealing variant.

As a concrete example, Listings 4-6 give client code for the well-known 'onemax' problem, chosen for its simplicity so that problem-specifics do not obfuscate the details of how metaheuristics are configured. Listing 4 show how the evaluation function and a uniform mutation heuristic are defined. Listings 5 and 6 demonstrate the configuration of random search and steepest-ascent hillclimbing respectively. Although these two listings contain some domain-specific details in order to provide a concrete example, it will hopefully be clear that it is entirely possible to write these kind of templates in a solution-independent manner. As is typical of JAVA, the listings are rather syntactically-verbose, but since many of the interfaces being subclassed consist of a single method, much simplification of client code will be possible when support for lambdas/closures is added in JAVA 8.

5. TRACES

One option available to the algorithm designer is the *level of granularity* at which traces are described. This is of particular importance because, at a hyper-heuristic level, we can consider metaheuristics to be operators [12, 30]. We may therefore wish to compress all the activity of a metaheuristic to a single operator, either to save memory or to explicitly denote it for purposes of subsequent metaheuristic choice (for example, determination of equivalent and inverse operator pairs at a given level of algorithm operation). In the general case of stochastic operators, the designer may elect to have the trace represent either the generalised stochastic operation or else the specific concrete change made to the underlying solution representation.

To illustrate this, we give an example traces for the application of Iterative Local Search on a 5-bit onemax problem. If we elect to record concrete changes, we obtain output of the following form:

$$\begin{aligned}&([01000])\\ \rightarrow&(BitFlip(0), [11000])\\ \rightarrow&(BitFlip(3), [11010])\\ \rightarrow&(BitFlip(4), [11011])\\ \rightarrow&(BitFlip(2), [11111])\end{aligned}$$

where (BitFlip(i) is a mutation applied to a specific variable i.

If the user elects for more coarse-grained information, then the output might appear as follows:

$$\begin{aligned}&([01000]),\\ \rightarrow&(UniformMutation@a1e6661, [11000])\\ \rightarrow&(UniformMutation@a1e6661, [11010])\\ \rightarrow&(UniformMutation@a1e6661, [11011])\\ \rightarrow&(UniformMutation@a1e6661, [11111])\end{aligned}$$

where UniformMutation@a1e6661 is simply the generic uniform mutation operator with no record of the specific bit changed.

It is implicit in the nature of multi-level search methodologies such as hyper-heuristics that it should be possible to make decisions based on operator application at different heirarchical scales, and the mechanism we provide here makes this an explicit part of the API.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented HYPERION^2, a framework to support scientific research in meta- and hyper- heuristics, with a particular focus on mapping the traces taken through the search space by algorithms and their components.

The framework promotes interoperability via component interfaces and is generic enough that meta- and hyper- heuristics can reuse the same source code. Generic templates for a variety of local search algorithms are already implemented, and novel meta- and hyper- heuristics may be constructed by hybridization or extension. The framework follows an 'only pay for what you use' design philosophy, for more efficient concrete implementations and rapid prototyping.

We have described the framework's core interfaces, and given some implementation examples. We have also provided a concrete example of a trace output by the framework. HYPERION^2 is in continuing development, with several remaining priorities.

The original HYPERION [12] implementation additionally provided generic templates for Ant-Cycle System, Evolutionary Strategies and Genetic Algorithms. In order to align these with the trace-based extensions of HYPERION^2, it will be necessary to introduce some form of 'multi-trace' support (i.e. each member of the population having its own trace). Furthermore, this would allow extension to multi-objective problems, with each member of a Pareto front having a corresponding trace. Given the 'only pay for what you use' design criterion, it remains to be discussed how best to achieve this. One option is to use a shared workspace in the manner of [30] as the repository for multi-traces, with their presence being at the option of the client-programmer.

7. REFERENCES

[1] E. K. Burke, G. Kendall, Search methodologies, Springer, 2005.

[2] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, J. R. Woodward, A classification of hyper-heuristic approaches, in: M. Gendreau, J.-Y. Potvin (Eds.), Handbook of Metaheuristics, Vol. 146 of International Series in Operations Research and Management Science, Springer US, 2010, pp. 449–468.

[3] J. R. Woodward, J. Swan, Why classifying search algorithms is essential, in: Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on, Vol. 1, IEEE, 2010, pp. 285–289.

[4] P. Ross, S. Schulenburg, J. G. Marín-Blázquez, E. Hart, Hyper-heuristics: Learning to combine simple heuristics in bin-packing problems, in: Proceedings of the Genetic and Evolutionary Computation Conference 2002, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002, pp. 942–948.

[5] S. Luke, Essentials of Metaheuristics, 2nd Edition, Lulu, 2013, available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[6] A. J. Parkes, Combined Blackbox and AlgebRaic Architecture (CBRA), in: Proc. of PATAT 2010 (extended abstract), 2010, pp. 535–538.

[7] F. Glover, Tabu Search - Part I, INFORMS Journal on Computing 1 (3) (1989) 190–206.

[8] J. R. Woodward, J. Swan, The automatic generation of mutation operators for genetic algorithms, in: Proceedings of the 14th Genetic and Evolutionary Computation Conference Companion, GECCO Companion '12, ACM, New York, NY, USA, 2012, pp. 67–74. doi:10.1145/2330784.2330796.

[9] J. R. Woodward, J. Swan, Automatically designing selection heuristics, in: Proceedings of the 13th Conference Companion on Genetic and Evolutionary Computation, GECCO '11, ACM, New York, NY, USA, 2011, pp. 583–590. doi:10.1145/2001858.2002052.

[10] K. Krawiec, J. Swan, Pattern-guided genetic programming, in: Proceeding of the 15th Genetic and Evolutionary Computation Conference, GECCO '13, ACM, New York, NY, USA, 2013, pp. 949–956. doi:10.1145/2463372.2463496.

[11] K. Krawiec, J. Swan, Guiding evolutionary learning by searching for regularities in behavioral trajectories: A case for representation agnosticism, in: 2013 AAAI Fall Symposium Series, 2013, pp. 41–46.

[12] J. Swan, E. Özcan, G. Kendall, Hyperion: a recursive hyper-heuristic framework, in: Proceedings of the 5th international conference on Learning and Intelligent Optimization, LION'05, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 616–630.

[13] C. Voudouris, R. Dorne, D. Lesaint, A. Liret, iOpt: A Software Toolkit for Heuristic Search Methods, in: T. Walsh (Ed.), Principles and Practice of Constraint Programming CP 2001, Vol. 2239 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2001, pp. 716–729.

[14] L. D. Gaspero, A. Schaerf, Easylocal++: An Object-oriented Framework for the flexible design of Local-Search Algorithms, Softw., Pract. Exper. 33 (8) (2003) 733–765.

[15] A. Fink, S. Voß, Hotframe: A heuristic optimization framework, in: S. Voß, D. Woodruff (Eds.), Optimization Software Class Libraries, OR/CS Interfaces Series, Kluwer Academic Publishers, Boston, 2002, pp. 81–154.

[16] M. Mitchell, An Introduction to Genetic Algorithms, The MIT Press, 1998.

[17] J. A. Parejo, A. Ruiz-Cortés, S. Lozano, P. Fernandez, Metaheuristic optimization frameworks: a survey and benchmarking, Soft Computing 16 (3) (2012) 527–561.

[18] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, M. Affenzeller, Advanced Methods and Applications in Computational Intelligence, Vol. 6 of Topics in Intelligent Engineering and Informatics, Springer, 2014, Ch. Architecture and Design of the HeuristicLab Optimization Environment, pp. 197–261. URL `http://link.springer.com/chapter/10.1007/978-3-319-01436-4_10`

[19] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, E. Popovici, S. K, J. Harrison, J. Bassett, R. Hubley, A. Chircop, H. W. Compton J, S. Donnelly, B. Jamil, J. O'Beirne, ECJ: A Java based evolutionary computation research system (2009). URL `http://cs.gmu.edu/~eclab/projects/ecj/`

[20] F. Otero, T. Castle, C. Johnson, EpochX: Genetic programming in java with statistics and event monitoring, in: Proceedings of the 14th genetic and

evolutionary computation conference companion, ACM, 2012, pp. 93–100.

[21] E. K. Burke, T. Curtois, M. Hyde, G. Kendall, G. Ochoa, S. Petrovic, J. A. Vazquez-Rodriguez, HyFlex: A Flexible Framework for the Design and Analysis of Hyper-heuristics, in: Multidisciplinary International Scheduling Conference (MISTA 2009), Dublin, Ireland, Dublin, Ireland, 2009, pp. 790–797. URL http://www.asap.cs.nott.ac.uk/publications/pdf/MISTA09HyFlex.pdf

[22] E. Urra, D. Cabrera-Paniagua, C. Cubillos, Towards an object-oriented pattern proposal for heuristic structures of diverse abstraction levels, in: Workshop on Agents and Collaborative Systems (WACS), School of Computer Engineering, Catholic University of Temuco, 2013.
URL jcc2013.inf.uct.cl

[23] A. Elyasaf, M. Sipper, HH-evolver: a system for domain-specific, hyper-heuristic evolution, in: Proceeding of the 15th conference companion on Genetic and evolutionary computation conference companion, ACM, 2013, pp. 1285–1292.

[24] H. K. Cora, H. T. Uyar, A. Ş. Etaner-Uyar, HH-DSL: a domain specific language for selection hyper-heuristics, in: Proceeding of the 15th Genetic and evolutionary computation conference companion, ACM, 2013, pp. 1317–1324.

[25] Lindawati, H. Lau, D. Lo, Instance-based parameter tuning via search trajectory similarity clustering, in: C. Coello (Ed.), Learning and Intelligent Optimization, Vol. 6683 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 131–145. doi:10.1007/978-3-642-25566-3_10.

[26] Lindawati, H. Lau, F. Zhu, Instance-specific parameter tuning via constraint-based clustering, in: Proc. of the 1st Int. Workshop on COmbining COnstraint solving with MIning and LEarning(CoCoMile' 12), joint with ECAI, Montpellier, France, 2012.

[27] L. Lindawati, F. ZHU, H. C. LAU, FloTra: Flower-shape trajectory mining for instance-specific parameter tuning, 10th edition of the Metaheuristics International Conference (MIC 2013), 2013.

[28] E. Gamma, R. Helm, R. E. Johnson, J. M. Vlissides, Design patterns: Abstraction and reuse of object-oriented design, in: ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming, Springer-Verlag, London, UK, 1993, pp. 406–431.

[29] J. O. Coplien, Curiously recurring template patterns, C++ Rep. 7 (2) (1995) 24–27.
URL http://dl.acm.org/citation.cfm?id=229227.229229

[30] J. Swan, J. Woodward, E. Özcan, G. Kendall, E. Burke, Searching the hyper-heuristic design space, Cognitive Computation 6 (1) (2014) 66–73.