Hyperion - A Recursive Hyper-heuristic Framework

Jerry Swan, Ender Özcan, Graham Kendall

Automated Scheduling, Optimisation and Planning (ASAP) Research Group,
School of Computer Science, University of Nottingham,
Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, UK.
{jps,exo,gxk}@cs.nott.ac.uk

Abstract. Hyper-heuristics are methodologies used to search the space of heuristics for solving computationally difficult problems. We describe an object-oriented domain analysis for hyper-heuristics that orthogonally decomposes the domain into generative policy components. The framework facilitates the recursive instantiation of hyper-heuristics over hyper-heuristics, allowing further exploration of the possibilities implied by the hyper-heuristic concept. We describe Hyperion, a Java^{TM} class library implementation of this domain analysis.

1 Introduction

The idea of combining the strength of multiple (meta-)heuristics goes back to the 1960s ([1], [2]) with the term *hyper-heuristics* being introduced by Denzinger et al. [3]. There has been recent interest in using hyper-heuristics to tackle combinatorial problems. One approach is to employ heuristics as primitive operators, guided to (and hopefully beyond) local optima by a portfolio of meta-heuristics, with the choice of meta-heuristic to apply at each decision-point being determined by a hyper-heuristic. The underlying idea is that hyper-heuristic activity tends to explore the space of local (and hence hopefully global) optima by using a set of lower-level (meta-)heuristics. There are two main types of hyper-heuristics, categorised by whether they are used for *selecting* or *generating* heuristics (see [4] for the former and [5] for the latter). For further detail on hyper-heuristics the reader is referred to [6], [7], [8] and [9].

We describe an object-oriented domain analysis for hyper-heuristics that orthogonally decomposes the domain into generative policy components [10]. This decomposition yields a generative algorithm framework that facilitates rapid prototyping and allows the components that contribute to an algorithm's success to be identified in a procedural fashion. In addition, we add facilities for recursively aggregating hyper-heuristics via the hierarchical nesting of local search neighborhoods. To the knowledge of the authors, there has been no explicit investigation of the effect of instantiating hyper-heuristics to a depth greater than 2, i.e. instantiating hyper-heuristics over hyper-heuristics (perhaps recursively) rather than simply over meta-heuristics. The facility for nesting algorithms to an arbitrary (and possibly dynamically-determined) depth therefore allows further exploration of the possibilities implied by the hyper-heuristic concept.

2 Domain Analysis

The widespread adoption of design patterns as reusable elements of domain vocabulary has lead to the development of a number of popular local search frameworks (e.g. [11], [12],[13]). Although these offer a diversity of approaches for high-level control, the essential nature of local search is present in some elemental domain concepts (albeit appearing under different names). We present them here in the vocabulary used by Fink and Voß [11] in their generic C++ class library, HotFrame:

State This type parameter represents an element of the solution-space.

ObjectiveFunction A measure of the quality of a State.

Heuristic This interface abstracts the mechanism for transforming an initial State into some other State of (hopefully) superior quality.

Neighborhood This defines some finite neighborhood of a State.

HotFrame also makes use of a NeighborhoodSelectionPolicy, layered upon Neighbourhood and having instances that include random neighbor, best neighbor, and best improving neighbor. Metaheuristics directly supported by HotFrame include iterated local search (from which random search and varieties of hillclimbing can be configured), together with varieties of simulated annealing and tabu search (the latter being configurable with a number of tabu strategies, including static and reactive tabu).

In addition to the identification of ubiquitous domain vocabulary, we were also strongly influenced in our domain decomposition by the approach of Özcan et al. [9], which achieves a highly-modular decomposition of hyper-heuristics as applied to the domain of fixed-length vectors of bits. Özcan et al. describe four separate hyper-heuristic frameworks in which primitive operations and meta-heuristics (in their case a variety of hillclimbers) are conditionally applied in turn. These four frameworks are conceptually parameterized by the choice of primitive operators, meta-heuristics and heuristic selection mechanisms. They also introduce an acceptance policy mechanism with instances that include unconditional acceptance; improving operations only; Metropolis-Hastings probabalistic acceptance of unimproving moves, and a variant of Great-Deluge.

To the knowledge of the authors, the only other *hyper-heuristic* framework is Hy-flex [14]. In contrast to the solution-domain frameworks above, Hy-flex is concerned with building reusable elements for common *problem domains*, and currently supports modules for SAT; one-dimensional bin-packing; permutation flow-shop and personnel scheduling. In the following sections, we describe Hyperion, a Java^{TM} class library for the hyperheuristic solution-domain that respects the entity relationships that hold between the key domain concepts, generalizes the framework of Özcan et al. and facilitates the hierarchical nesting of meta-heuristics.

3 The HYPERION Hyper-heuristic Framework

We employ object-oriented and generative programming methods [10] to decompose the problem domain, resulting in the key concepts (implemented either directly as classes or generatively via parameterized types) illustrated in Fig. 1-3.

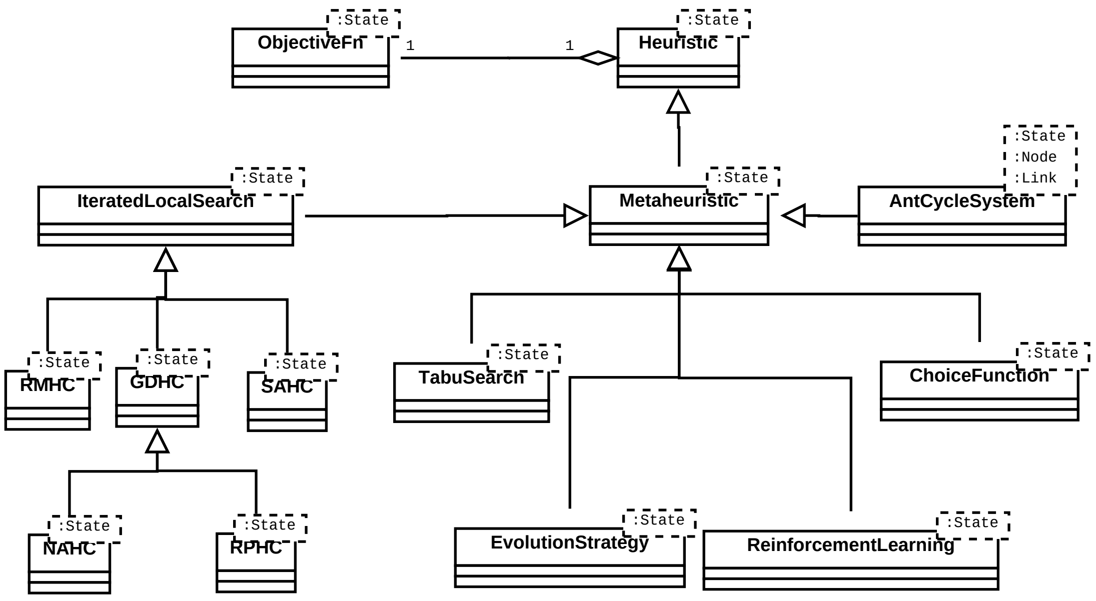

Fig. 1. Main interfaces and concrete meta-heuristics

Figure 1 depicts the heirarchy for HEURISTIC and some of its concrete specialisations. The polymorphic *update* method in the HEURISTIC class represents a single iteration of the algorithm. Formally, the method signature is:

$$update : Transition\langle State\rangle \rightarrow Transition\langle State\rangle$$

where State is a generic type, as denoted by the bracket conventions) and TRANSITION is the generically-typed 5-tuple

$$(from : State, fromValue : \mathbb{R}, operator : Operator, to : State, toValue : \mathbb{R})$$

with *operator* being a descriptor for the operation instance applied. The semantics are that the heuristic should return a result in which the *to* State represents the perturbation of the *from* State of its argument via a single application of the subclass algorithm. In [11], the existence of many-to-one relationships between state-space and objective function and state-space and neighborhood are acknowledged, but for efficiency purposes in the implementation, the STATE concept is actually in one-to-one correspondence with its objective function. Our formulation using explicit “pass-though” of tuples representing transitions in the implied search graph (with their caching of objective values of states) allows us to achieve the desired decoupling of states, objective values and neighborhoods without loss of efficiency. Note that, in the domain of hyper-heuristics, the decoupling of states and neighborhoods is essential, since we need to interoperably consider multiple neighborhoods (perhaps operating at different hierarchical levels) over the same state representation. Figure 2 depicts the heirarchy for Locality,

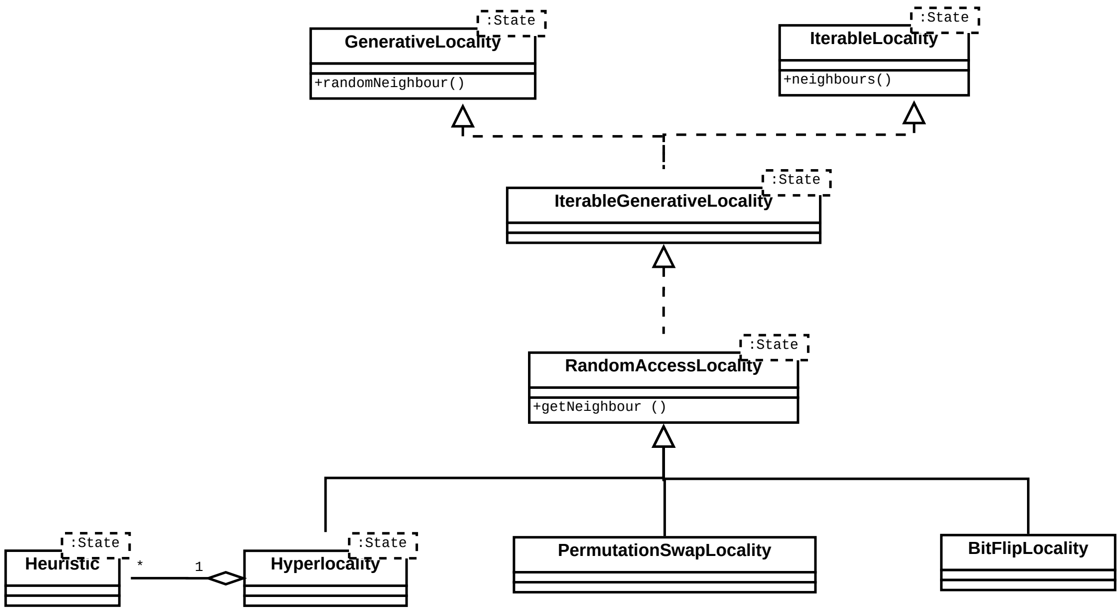

Fig. 2. Abstract and concrete localities

the Hyperion term for the ubiquitous concept of local search neighbourhood. In contrast to the singular HotFrame neighborhood concept, the Hyperion 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 [15], 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 for bit-flip and permutation-swap neighborhoods. By way of example, the interface for BitFlipLocality is given in Listing 1. Hyperion adopts a similar neighborhood selection policy approach to HotFrame, additionally providing stochastic tie-breaking and proportional, rank and tournament

```
public final class BitFlipLocality
extends RandomAccessLocality< BitVector >
{
  public BitFlipLocality( int bitVectorSize )
  {
    /* ... */
  }

  @Override
  public int neighbourhoodSize( Transition< BitVector > t )
  {
    /* ... */
  }

  @Override
  public Transition< BitVector >
  getNeighbour( Transition< BitVector > t, int i )
  {
    /* ... */
  }
}
```

Listing 1. Methods for class BitFlipLocality (implementation details omitted)

selection. We incorporate the acceptance policies of [9] as a generic parameter, and provide the following policies (depicted in Fig. 3):

All Moves (AM) Unconditionally accepts all generated states.

Only Improving (OI) Accepts only states that improve on the objective value of the previously generated state.

Improving and Equal (IE) As OI, but states of equal objective value are also accepted.

Exponential Monte Carlo (EMC) A worsening move is accepted by this policy with the probability of $p_t = e^{-\frac{\Delta f u}{C}}$, where Δf is the change in objective value in the t-th iteration, C is a counter for successive worsening moves and u is the unit time (e.g., in minutes) that measures the duration of the heuristic execution [16].

Simulated Annealing (SA) This policy accepts unimproving states with probability $p_t = e^{-\frac{\Delta f / N}{1 - t/D}}$, where Δf is the change in objective value in the t-th iteration, D is the maximum number of iterations and N is the maximum possible fitness change [17], [18], [19].

Great Deluge (GD) A variant of the algorithm given in [20], this policy accepts states that are improving or equal relative to a dynamically-determined value that is linearly interpolated from initial to optimal (or best-known) values via the iteration count.

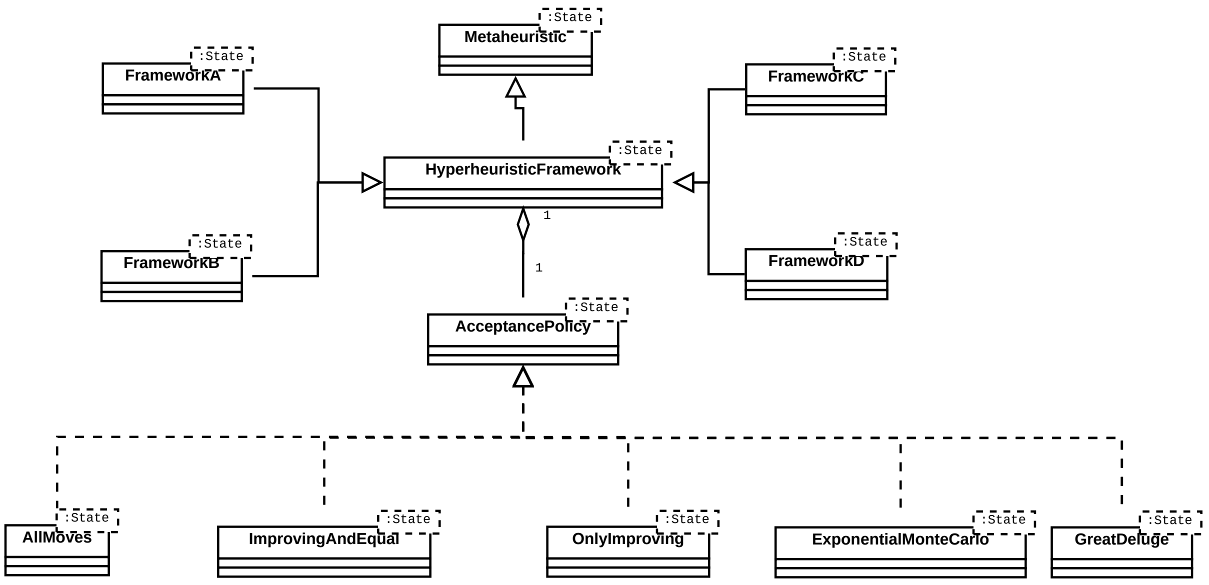

Fig. 3. Frameworks and acceptance policies

The hillclimbing meta-heuristics implemented in HYPERION are combinatorial generalizations of the bitwise hillclimbing variants described in [21]. Each hillclimber iteratively replaces the current solution (conditional upon the acceptance policy) with a solution chosen from the current neighborhood according to a neighbor selection policy. Steepest Ascent Hillcimbing (SAHC) evaluates all neighbors and chooses the one with the best objective value. In Random Mutation Hill Climbing (RMHC), the selection policy is to choose a random neighbor. Generalized Davis Hill Climbing (GDHC) is a generalization of Davis's *random bit climber*, in which successive neighbour selections are determined by successive indices of a permutation function. Next Ascent Hillclimbing (NAHC) is then given by instantiating GDHC with the identity permutation and RPHC is GDHC with a random permutation function.

Other heuristic selection strategies implemented in [21] include Choice Function (CF) [4], Simple Random (SR) and Greedy (GR). CF is directly implemented in HYPERION, SR is equivalent to RMHC and GR may be achieved by instantiating ITERATEDLOCALSEARCH with a BESTNEIGHBOUR selection policy (optionally with stochastic tie-breaking). Other meta-heuristics implemented within HYPERION include Reinforcement Learning [22] [23], Evolutionstrategië [24], Tabu Search [25] and Ant-Cycle System [26]:

ReinforcementLearning (RL) Heuristics are ranked (ranking scores are constrained to a fixed range) with scores increasing or decreasing as a function of the heuristic's performance.

Evolutionstrategië (ES) This is a population-based approach in which the number of mutations applied to offspring is an typically a function of some aspect of parent state.

Tabu Search (TS) This restricts the local search neighbourhood by maintaining a (potentially adaptive) mechanism for identifying prohibited transitions.

Ant Cycle System (ACS) This maintains a graph of *solution components* which is repeatedly traversed by a collection of agents. Components from each traversal are assembled into a complete solution in a problem-specific manner.

Since ES is population-based, there is no entirely satisfactory way for it implement the single-solution-based *update* method. We have elected to achieve this by returning the best population member encountered so far and treating the input *from* state as a hint for conditionally reseeding the population. TS is parameterized by a TABUPOLICY in a similar manner to HOTFRAME, since design investigation of a variety of alternative tabu policy signatures revealed that the HOTFRAME approach was the most loosely-coupled of all the alternatives considered. For each of these meta-heuristics, except ACS, the neighborhood is specified via a LOCALITY parameter. In [9], hillclimbers feature as both meta-heuristics and hyper-heuristics, but are implemented separately in each case. By contrast, HYPERION facilitates the creation of hyper-heuristics from existing meta-heuristics via the HYPERLOCALITY specialization of RANDOMACCESSLOCALITY. By adapting a sequence of heuristics into a locality, a HYPERLOCALITY (listing 2) allows the same algorithm implementation to be used in either case. Listing 3 shows the use of HYPERLOCALITY to recursively instantiate a collection of hillclimbers.

The four frameworks described by Özcan et al. are shown in Fig. 3 in the context of HYPERION and the detail of their internal operation is given in Fig. 4. In these frameworks, primitive heuristics and hillclimbers (or more generally in HYPERION, meta- or hyper- heuristics) can be partitioned into separate groups. If we denote the application of a framework-selected primitive heuristic by h, a framework-selected higher-order (i.e. meta- or hyper-) heuristic by H and a predetermined higher-order heuristic by P, then the operation of a single invocation of the *update* method on these these frameworks can be described by the following grammar:

$$\begin{aligned} F_A &::= h|H \\ F_B &::= hP|H \\ F_C &::= hP \\ F_D &::= hH \end{aligned}$$

The underlying idea is that this pattern of interaction between primitive and higher-order heuristics will promote solution diversity [9].

```
public final class Hyperlocality< State >
extends RandomAccessLocality< State >
{
  private List< Metaheuristic< State > >  meta-heuristics;

  public Hyperlocality( List< Metaheuristic< State > > mh )
  {
    this.meta-heuristics = mh;
  }

  @Override
  public Transition< State >
  getNeighbour( Transition< State > t, int index )
  {
    return meta-heuristics.get( index ).update( t );
  }

  @Override
  public int neighbourhoodSize( Transition< State > s )
  {
    return meta-heuristics.size();
  }
}
```

Listing 2. Methods for class Hyperlocality

```
public final class HyperHillclimbers
{
  public static < State >
  List< Metaheuristic< State > >
  instantiate( RandomAccessLocality< State > locality ,
      int recursionDepth )
  {
    if( recursionDepth < 0 )
      throw new IllegalArgumentException();
    else if( recursionDepth == 0 )
      return getHillclimbers( locality );
    else
    {
      List< Metaheuristic< State > > lm = instantiate(
          locality , recursionDepth - 1 );
      return getHillclimbers( new Hyperlocality< State >( lm )
           );
    }
  }

  ////////////////////////////////

  private static < State >
  List< Metaheuristic< State > >
  getHillclimbers( RandomAccessLocality< State > locality )
  {
    List< Metaheuristic< State >
      > result = new ArrayList< Metaheuristic< State > >();
    result.add( new SAHC< State >( locality ) );
    result.add( new RMHC< State >( locality ) );
    result.add( new NAHC< State >( locality ) );
    result.add( new RPHC< State >( locality ) );
    return result;
  }
}
```

Listing 3. Recursive instantiation of hyper-hillclimbers

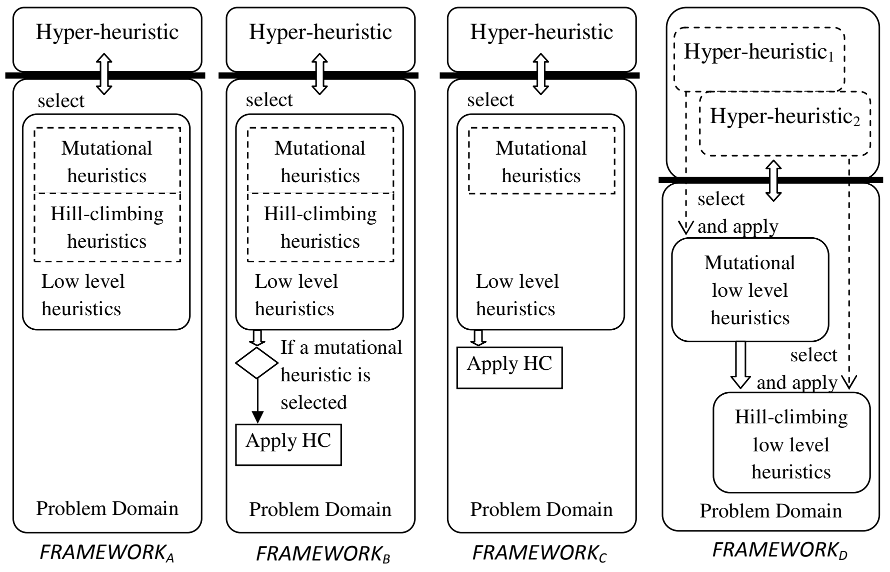

Fig. 4. Internal operation of top-level frameworks

3.1 Design-space of Hyper-heuristics

In [8], Burke et al. describe a design space for hyper-heuristics that has two orthogonal dimensions. The first dimension represents selection versus generation and the second the source of feedback during learning (online,offline or none). Both dimensions are further partitioned by the nature of the search space (constructive or pertubative). If we instantiate Hyperion with State taken to be some representation of solution state S, then this corresponds to selective hyper-heuristics. If instead we take State to be some type representing the mapping $S \to S$, then this corresponds to generative hyper-heuristics. The only explicitly constructive heuristic implemented in Hyperion is ACS, which is additionally parameterized by Node and Link types, representing the vertices and edges of the graph of partial solutions traversed by the agents of the ACS. If we employ ACS as a hyperheuristic over some complete solution state, then a path in the graph of partial solutions corresponds to a sequence of lower-level heuristics and an adaptor function is used to yield the resulting complete solution state via by the sequential application of these heuristics to the *from* state. In general therefore, heuristics may be considered as constructive or perturbative as required, employing adaptors as necessary for interoperability with other solution representations. By virtue of this modularity of decomposition, Hyperion facilitates a wide variety of hyperheuristic strategies. In particular, the approaches adopted in [27], [28] and [29] may all be considered as specific configurations of Hyperion components.

3.2 Application to SAT

We illustrate the use of the framework classes via application to the well-known boolean satisfiability problem (SAT). The palette of meta-heuristics is obtained from some class MyMetaheuristics, which is identical to code for the hyperhill-climbers described in listing 3, together with an instantiation of simulated annealing that has a geometric annealing schedule in which the parameters are dynamically determined by sampling the state-space [30]. The client-code for applying 'Framework A' to the SAT domain using a simple heuristic measure of the number of unsatisfied clauses is given in listing 4. Table 1 gives the average heuristic values obtained from 100 applications of this framework to the first 20 instances of the 3-SAT UF20-91 SATLIB problem set (`http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`) [31]. All instances have 20 variables and 91 clauses and are known to be satisfiable. RPHC can be seen to give better average performance in all cases, but if we consider the percentage of cases that are actually solved (as given in the bottom row of Table 1), we see that SAHC converges in the highest number of cases and RPHC gives the second worst performance.

Problem instance	RPHC	RMHC	SAHC	SA	NAHC
uf20-01.cnf	2.96	4.36	6.85	8.42	11.54
uf20-02.cnf	3.14	4.16	6.08	7.09	9.84
uf20-03.cnf	3.37	4.99	7.56	9.24	12.35
uf20-04.cnf	3.23	5.27	8.11	10.0	13.67
uf20-05.cnf	3.84	5.75	9.13	10.97	15.32
uf20-06.cnf	3.29	5.0	7.35	9.02	12.37
uf20-07.cnf	2.91	3.93	5.55	6.78	9.07
uf20-08.cnf	3.07	4.7	6.89	8.32	11.2
uf20-09.cnf	3.07	4.71	6.84	8.41	11.88
uf20-010.cnf	2.76	4.19	6.16	7.49	10.23
uf20-011.cnf	2.9	3.76	5.84	6.81	9.88
uf20-012.cnf	2.2	2.82	4.38	4.96	7.33
uf20-013.cnf	3.42	5.24	7.82	9.79	13.17
uf20-014.cnf	2.92	4.4	6.68	8.2	11.29
uf20-015.cnf	2.67	3.71	5.49	6.49	9.2
uf20-016.cnf	2.82	4.12	6.26	7.53	10.34
uf20-017.cnf	2.55	3.94	5.75	7.2	9.76
uf20-018.cnf	3.56	5.65	8.64	10.69	14.6
uf20-019.cnf	3.28	5.26	7.85	9.67	13.17
uf20-020.cnf	3.21	4.66	7.06	8.54	12.0
percentage solved	3.7%	8.55%	10.35%	8.25%	2.9%

Table 1. Average heuristic values obtained over 100 runs of 3-SAT instances

```
package hyperion.benchmarks.sat;

public final class RunSAT
{
  static final int NUM_ITERATIONS = 100000;
  static final int HYPERHEURISTIC_NESTING_LEVEL = 0;
  // ^ nesting level 0 instantiates _meta_ heuristics

  public static void main( String [] args ) throws IOException
  {
    String fileName = "resources/uf20-91/uf20-0102.cnf";
    CNF cnf = ReadCNF.readDIMACS( fileName );
    ObjectiveFn<BitVector> heuristicFn = new
        NumUnsatisfiedClauses( cnf );
    List< Metaheuristic<BitVector> > hyperheuristics =
      MyMetaheuristics.instantiate(
      new BitFlipLocality( cnf.getNumVariables() ),
        HYPERHEURISTIC_NESTING_LEVEL );

    BitVector initial = new BitVector( cnf.getNumVariables());
    AcceptancePolicy<BitVector> acceptance = new AllMoves<
        BitVector >();
    for( Metaheuristic<BitVector> alg : hyperheuristics )
    {
      FrameworkA<BitVector> framework = new FrameworkA<
          BitVector>(
          alg,
          acceptance,
          NUM_ITERATIONS );

      BitVector result= framework.apply(initial, heuristicFn);
      int value = heuristicFn.valueOf( result );
      System.out.println("alg:" + alg + ",value:" + value);
    }
  }
}
```

Listing 4. Client code for SAT solver

4 Conclusion and Future Work

We have presented an object-oriented analysis of the hyper-heuristic domain, incorporating generic versions of the decomposition given in [9] to produce a Java^{TM} implementation (available from `http://hyperion-java.sourceforge.net`) that recursively aggregates local search neighborhoods to generate hyper-heuristics from meta-heuristics without the necessity for source-code duplication. In addition, it is possible to combinatorially instantiate hyper-heuristics from collections of policy components, with the additional possibility that instantiation can recurse over available meta-heuristics to some dynamically-determined depth.

Recursion is thus of value as a facility for source code re-use. In addition, by altering the given examples of recursive instantiation to make a stochastic choice of lower-level (hyper-)heuristics, Hyperion can also be considered as a generation mechanism for *strongly-typed genetic programming* [32] in the domain of hyper-heuristics. Future work includes an investigation of the effect of recursion depth in the context of building-blocks in 'hierarchical iff' functions [33]. There are also a number of aspects of the current framework implementation that we believe could be improved upon. As discussed above, single-state and population-based meta-heuristics do not interoperate in an entirely satisfactory manner. A more loosely-coupled scheme for mediating interactions between heuristics is currently under development. Another significant improvement would be a change in the level of abstraction from that of local search neighborhoods to local search *frames*, the analogy being with stack frames in a conventional programming language. A frame encapsulates an algorithm instantiated over a locality and comes equipped with a parameter schema detailing not only the set of permissible parameter values but also other information pertinent to searching the parameter space (e.g. whether first or second derivatives exist for a parameter).

References

1. H. Fisher, G. L. Thompson, Probabilistic learning combinations of local job-shop scheduling rules, in: J. F. Muth, G. L. Thompson (Eds.), Industrial Scheduling, Prentice-Hall, Inc, New Jersey, 1963, pp. 225–251.
2. W. Crowston, F. Glover, G. Thompson, J. Trawick, Probabilistic and parameter learning combinations of local job shop scheduling rules, ONR Research Memorandum, GSIA, Carnegie Mellon University, Pittsburgh, 117 (1963).
3. J. Denzinger, M. Fuchs, M. Fuchs, High Performance ATP Systems by combining several AI Methods, in: Proceedings of the 4th Asia-Pacific Conference on SEAL, IJCAI, 1997, pp. 102–107.
4. P. I. Cowling, G. Kendall, E. Soubeiga, A Hyperheuristic approach to Scheduling a Sales Summit, in: PATAT '00: Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling III, Springer-Verlag, London, UK, 2001, pp. 176–190.
5. E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, J. R. Woodward, Exploring Hyper-heuristic Methodologies with Genetic Programming, in:

J. Kacprzyk, L. C. Jain, C. L. Mumford, L. C. Jain (Eds.), Computational Intelligence, Vol. 1 of Intelligent Systems Reference Library, Springer Berlin Heidelberg, 2009, pp. 177–201.
6. P. Ross, Hyper-heuristics, in: E. K. Burke, G. Kendall (Eds.), Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, Springer, 2005, Ch. 17, pp. 529–556.
7. E. K. Burke, E. Hart, G. Kendall, J. Newall, P. Ross, S. Schulenburg, Hyper-heuristics: An emerging direction in modern search technology, in: F. Glover, G. Kochenberger (Eds.), Handbook of Metaheuristics, Kluwer, 2003, pp. 457–474.
8. 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.
9. E. Özcan, B. Bilgin, E. E. Korkmaz, A comprehensive analysis of hyper-heuristics, Intell. Data Anal. 12 (1) (2008) 3–23.
10. K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley Professional, 2000.
11. 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.
12. 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.
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. 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`
15. 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.
16. M. Ayob, G. Kendall, A monte carlo hyper-heuristic to optimise component placement sequencing for multi head placement machine, in: Proceedings of the International Conference on Intelligent Technologies (InTech'03), Chiang Mai, Thailand, 2003, pp. 132–141.
17. S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, Optimization by simulated annealing, Science 220 (1983) 671–680.
18. R. Bai, G. Kendall, An investigation of automated planograms using a simulated annealing based hyper-heuristics, in: T. Ibaraki, K. Nonobe, M. Yagiura (Eds.), Metaheuristics: Progress as Real Problem Solver, Springer, 2005, pp. 87–108.
19. E. Burke, G. Kendall, M. Misir, E. Özcan, Monte carlo hyper-heuristics for examination timetabling, Annals of Operations Research (2010) 1–1810.1007/s10479-010-0782-2.
20. G. Dueck, New optimization heuristics: The great deluge algorithm and the record-to record travel, Journal of Computational Physics 104 (1993) 86–92.

21. M. Mitchell, J. H. Holland, When will a genetic algorithm outperform hill climbing?, in: Proceedings of the 5th International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, p. 647.
22. L. P. Kaelbling, M. L. Littman, A. P. Moore, Reinforcement learning: A survey, J. Artif. Intell. Res. (JAIR) 4 (1996) 237–285.
23. E. Özcan, M. Misir, G. Ochoa, E. Burke, A reinforcement learning - great-deluge hyper-heuristic for examination timetabling, International Journal of Applied Metaheuristic Computing (2010) 39–59.
24. M. Herdy, Application of the evolutionsstrategie to discrete optimization problems, in: H.-P. Schwefel, R. Männer (Eds.), Parallel Problem Solving from Nature I, Vol. 496 of Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 188–192.
25. F. Glover, Tabu Search - Part I, INFORMS Journal on Computing 1 (3) (1989) 190–206.
26. M. Dorigo, T. Stützle, Ant Colony Optimization, MIT Press, 2004.
27. J. C. Ortiz-Bayliss, E. Özcan, A. J. Parkes, H. Terashima-Marin, Mapping the performance of heuristics for constraint satisfaction, 2010, pp. 1–8.
28. M. Hyde, E. Özcan, E. K. Burke, Multilevel search for evolving the acceptance criteria of a hyper-heuristic, in: Proceedings of the 4th Multidisciplinary Int. conf. on Scheduling: Theory and Applications, 2009, pp. 798–801.
29. E. Ersoy, E. Özcan, c. Uyar, Memetic algorithms and hyperhill-climbers, in: P. Baptiste, G. Kendall, A. M. Kordon, F. Sourd (Eds.), 3rd Multidisciplinary Int. Conf. On Scheduling: Theory and Applications, 2007, pp. 159–166.
30. S. White, Concepts of scale in simulated annealing, in: Proc. Int'l Conf. on Computer Design, 1984, pp. 646–651.
31. H. H. Hoos, T. Stützle, SATLIB: An online resource for research on SAT, In: I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000, SATLIB is available online at www.satlib.org (2000).
32. D. J. Montana, Strongly typed genetic programming, Evolutionary Computation 3 (2) (1995) 199–230.
33. D. Iclanzan, D. Dumitrescu, Overcoming hierarchical difficulty by hill-climbing the building block structure, in: GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM, New York, NY, USA, 2007, pp. 1256–1263.