A cross-domain method for generation of constructive and perturbative heuristics

Christopher Stone, Emma Hart and Ben Paechter

Abstract Hyper-heuristic frameworks, although intended to be cross-domain at the highest level, usually rely on a set of domain-specific low-level heuristics which exist below the domain-barrier and are manipulated by the hyper-heuristic itself. However, for some domains, the number of available heuristics can be very low, while for novel problems, no heuristics might exist at all. We address this issue by describing two general methods for the automated production of constructive and perturbative low-level heuristics. Grammatical evolution is used to evolve low-level heuristics that operate on an 'intermediate" graph-based representation built over partial permutations. As the same grammar can be applied to multiple application domains, assuming they follow this representation, the grammar can be viewed as cross-domain. The method is evaluated on two domains to indicate generality (the Travelling Salesman Problem and Multidimensional Knapsack Problem). Empirical results indicate that the approach can generate both constructive and perturbative heuristics that outperform well-known heuristic methods in a number of cases and are competitive with specialised methods for some instances.

1 Introduction

Hyper-heuristics, "heuristics to choose heuristics", were first introduced in an attempt to raise the generality at which search methodologies operate [4], searching over the space of solvers rather than solutions (as in typical meta-heuristics). Their development was motivated by a desire to produce a method that was cheaper to

Christopher Stone
University of St Andrews, e-mail: cls29@st-andrews.ac.uk

Emma Hart
Edinburgh Napier University, e-mail: e.hart@napier.ac.uk

Ben Paechter
Edinburgh Napier University, e-mail: b.paechter.ac.uk

implement and easier to use than problem-specific, customised methods, while producing solutions of acceptable quality to an end-user in an appropriate time-frame. Specifically, it aimed to address a concern that the practical impact of search-based optimisation techniques in commercial and industrial organisations had not been as great as might have been expected, due to the prevalence of problem-specific or knowledge-intensive techniques, which were inaccessible to the non-expert or expensive to implement.

The canonical hyper-heuristic framework introduces a *domain-barrier* that separates a general algorithm to choose heuristics from a set of low-level heuristics that are specific to a domain, i.e. a particular application class of problem such as bin-packing, or vehicle-routing. The over-riding idea is that switching domains only requires a change in the set of low-level heuristics, with no change to the controlling high-level hyper-heuristic. Clearly, the success of the high-level heuristic is strongly influenced by the number and the quality of the low-level heuristics available. Low-level heuristics incorporating problem-specific knowledge are often designed by hand, relying on intuition or human-expertise [4]. In general, they are also tied to a specific problem-representation, given that they either modify an existing solution (in the case of perturbative heuristics) or create a solution from scratch (in the case of constructive heuristics). Therefore, it is unlikely that low-level heuristics from one domain transfer well (or at all) to another. As a result, when tackling a new domain, a new low-level heuristic set for the domain must be created. This can be often supplied by experts, or in cases where no heuristics are available, it has been shown that new heuristics can be evolved, for example using genetic programming (GP) [2]. However, the latter approach requires an in-depth understanding of a domain in order to select appropriate function and terminal nodes that can be used by the genetic programming algorithm to evolve a heuristic. As such, although at conceptual level, GP can be used to evolve heuristics for any domain, it has to be individually customised to a domain each time it is used.

For new problem domains that do not map well to well-studied domains in the literature, developing an appropriate set of low-level heuristics remains a challenging problem. In this study, we aim to address this challenge by introducing a method of creating new heuristics that is *cross-domain* in the sense that the *generating* method can be used without modification to create heuristics in multiple domains; the heuristics created as a result however are specialised to an individual domain. In other words, we describe a cross-domain generation method of generating domain-specific heuristics.

A cross-domain generation approach must necessarily utilise a common problem representation. We adopt a graph-based representation as it enables a broad spectrum of practical problems to be represented. While this includes obvious applications such as routing and scheduling [26] which have natural representations as graphs, it can also be applied in many less obvious ones including packing problems [22] and utility maximisation in complex negotiations[24] through appropriate manipulation.

We describe two approaches that use grammatical-evolution to generate low-level heuristics for any domain that is represented in graph-form. Each method can

be used *without modification* to generate heuristics for multiple domains. The first approach generates constructive heuristics while the second uses a modified version of the grammar to generate perturbative heuristics. The research lays the foundation for a paradigm shift in designing heuristics for combinatorial optimisation domains in which no heuristics currently exist, or those domains in which hyper-heuristic methods would benefit from additional low-level heuristics. The approach significantly reduces the burden on human experts, as it only requires that the problem can be represented as a graph, with no further specialisation, and does not require a large database of training examples. The method is not expected to compete with domain-specific approaches that have been customised to the idiosyncrasies of a given domain; rather it is intended as a straightforward way of creating new heuristics that have acceptable performance when no other heuristics are available and the expert knowledge need to create them is lacking. The vast majority of hyper-heuristic research is focused on generating algorithms that operate above the domain-barrier — in keeping with the original philosophy of creating generalised methods. In contrast, our approach operates "under" the domain-barrier. However, in keeping with the spirit of hyper-heuristics, our approach is also generalisable, in being able to create low-level heuristics for a diverse range of domains.

The article provides brings together research previously described in detail in [29, 30]. It provides a synthesised overview of the previous work, while providing some examples of results selected from the original papers to illustrate the salient points. The reader is referred to the original papers for more complete results and more detailed description of the methods.

2 Related Work

While the majority of initial work in the field of hyper-heuristics focused on development of the high-level controlling heuristics[4], more recent attention has focused on the role of the low-level heuristics themselves. Low-level heuristics fall into two categories [4]. *Constructive* heuristics build a solution from scratch, adding an element at a time, e.g. [26] and have been applied in a variety of domains such as personnel scheduling, job-shop problem [31], education timetabling [23] and packing [27]. On the other hand, *perturbative* heuristics modify an existing solution, e.g. re-ordering elements in a permutation [7] or modifying genes [4].

Typical methods to generate constructive heuristics include Genetic Programming (GP) [16] and Grammatical Evolution (GE)[20]. GP constructs trees that, for example, output a number representing an item priority, e.g. for vehicle routing [26], job-shop scheduling [12], TSP [14]. GE is a form of grammar based genetic programming developed for the automatic generation of programs. Differently from GP, it does not apply the evolutionary process directly to a program but on a variable length genome. A mapping process then turns the genome into a program by following grammar rules specified using Backus Naur Form [20]. This approach ensures the creation of syntactically correct programs that then are executed and their fit-

ness function evaluated. GE has already been applied to construct heuristics for the capacitated vehicle routing problem and for the bin packing problem [25]. With respect to generation of *perturbative* heuristics, GP approaches are also common, e.g. for generating novel local search heuristics for satisfiability testing [4]. Grammatical Evolution was applied to evolve new local-search heuristics for 1d-bin packing in [4, 18].

Despite some success in the areas just described, we note that in each case, the function and terminal nodes used in GP or the grammar specification in GE are specifically tailored to a single domain. While clearly specialisation is likely to be beneficial, it can require significant expertise and investment in algorithm design. For a practitioner, such knowledge is unlikely to be available, and for new domains, this may be time-consuming even for an expert. Therefore, we are motivated to design a general-purpose method that is capable — without modification — of producing heuristics in multiple domains. While we do not expect such a generator to compete with specialised heuristics or meta-heuristics, we evaluate whether the approach can be used as a "quick and dirty" method of generating a heuristic that produces an acceptable quality solution in multiple domains.

3 Evaluation Domains and their representation as graphs

In the proposed system we encode all the properties of the problems into a graph embedded in some arbitrary space. Whenever possible, we convert properties into some spatial concept to which we can associate some arbitrary metric. We select two evaluation domains. The first is the *Travelling Salesman Problem* (TSP) [17], one of the most studied problems in combinatorial optimisation, in which a tour passing through exactly n points once must be *minimised*. Due to the fact that it is naturally encoded as an ordering problem represented by a graph in which the cities to visit can be trivially encoded as vertices in 2-D Euclidean space it provides a straightforward baseline for our experiments. The second domain chosen is the *Multidimensional Knapsack Problem* (MKP). This is also well studied with applications in budgeting, packing and cutting problems. In the typical version, the profit from items selected among a collection must be *maximised* while respecting the constraints of a knapsack. In contrast to TSP, knapsack solvers do not typically use a graph-based representation of the problem [6]. However, it can be re-represented in a graph-based formulation as follows.

Assume there is one vertex corresponding to each object, and one vertex for the knapsack. The properties of the vertex can be interpreted as coordinates that determine the location of the vertices in some constraint-profit space. A geometric interpretation of the problem can be intuitively described as follows: when an object is chosen (connected to the knapsack vertex) the properties of the object are added to the knapsack and it is moved in the constraint-profit space. The amount of motion is equal to the values of the object's vector in constraint space and in the direction of the profit space. The configuration of objects connected to the knapsack that

move the knapsack the furthest in profit space without the knapsack crossing the line corresponding to its maximum capacity in any of its constraint dimensions is the best configuration. An example with just one constraint (weight) is drawn in Fig. 1.

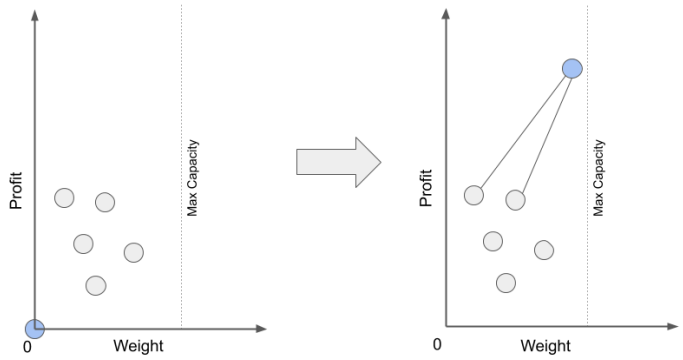


Fig. 1: Geometric interpretation of the knapsack problem simplified to two dimensions (weight constraint and profit). On the left the knapsack at initialisation has 0 weight and 0 profit. As objects are connected to the knapsack it is moved of an amount equivalent to the vectors defined by the sum of the objects.

4 Using Grammatical Evolution to evolve low-level heuristics

Grammatical Evolution [19], is a population based approach developing a program through manipulating an integer string (the genotype) which is subsequently mapped to a program (or similar) through the use of a grammar. Given a grammar defined in Backus Naur Form (BNF) containing a terminal Set $< T >$, a non-terminal set $< N >$ and a set of production rules $< P >$, and a start point, an integer sequence can be used to specify a program (a function) by selecting production rules and their expansions (see[19] for a full description). The role of the evolutionary process is then to search the space of integers to find a suitable program.

Here we use the PonyGE3[1] Python implementation of GE [10] to evolve grammars that create constructive and perturbative heuristics. This uses a linear genome representation encoding a list of integers (codons). The mapping between the genotype and the phenotype is actuated by the use of the modulus operator on the value of

[1] https://github.com/PonyGE/PonyGE2

the codon, i.e. *Selected node* = c mod n, where c is the integer value of the codon to be mapped and n is the number of options available in the specific production rule. Offspring are created through one-point crossover followed by mutation that substitutes a single randomly selected codon with a new, randomly chosen value. A generational replacement strategy with elitism of size 1. To evolve constructive heuristics, we use a grammar rich in geometric operations that allows a heuristic to make a choice at each step based on measures over the graph. To evolve perturbative heuristics, we use a grammar that performs three operations: cuts, inversions and permutations. These are described in detail below.

4.1 Evolving Constructive Heuristics

To evolve a new constructive heuristic that builds a solution from scratch, GE is used to evolve a ranking function that ranks candidate vertices. The ranking function is iteratively applied to select the next vertex to be added to the partial solution. Assuming there are n available vertices for a domain, then in the TSP case, exactly $k = n$ vertices must be selected, while in the knapsack domain, $k<n$ are selected.

In the case of a TSP solution, each successive vertex defines the next city to visit. In the case of MKP, each successive vertex of the chain is used to select the next item to be placed in the knapsack as shown in figure 2.

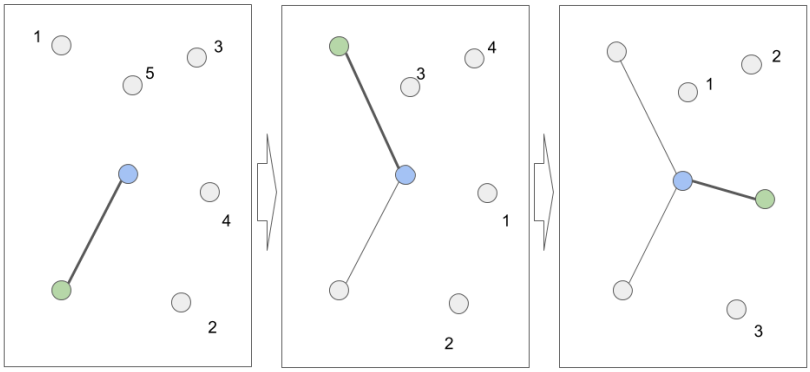


Fig. 2: Constructing a solution for the MKP. Blue node is the knapsack. Green node is the last appended vertex

The set of production rules used is given in figure 3. Definitions of the terminal nodes used to define nodes can be found in [28]. Note the same *grammar* is used to generate constructive heuristics for both the TSP and Knapsack domains.

```
<exp> → <exp><arithmetic_op><exp> |
        protected_division(<exp>,<exp>) |
        root(<exp>) |
        log(<exp>) |
        <c><c>.<c><c>|
        <trig> |
        <graph_function> |
        <info> |
        <constant>
<c>  →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<arithmetic_op>  → + | * | -
<trig> → sine(<exp>) |
        cosine(<exp>) |
        tangent(<exp>)
```

```
<graph_function> → distance(vertex, <metric>) |
                   kd_angle_leg(vertex) |
                   estimated_graph_complexity |
                   hull_area
                   longest_edge |
                   distance_to_v0(vertex, <metric>) |
                   vec_max(<vector>) |
                   vec_min(<vector>) |
                   elements_sum(<vector>)
<vector>  →  v0_difference(vertex) |
             chain_delta_vectorsum |
             v0
<metric>  →  euclidean | cosine distance
<constant> → π | ε
<info>    →  chain_length | vertices_num
```

Fig. 3: Complete grammar for generation of *constructive* heuristics

4.2 Evolving Perturbative Heuristics

In this case, we use GE to evolve a Python program that takes a sequence (i.e a permutation) as an input and returns a modified version of the same sequence (permutation) with the same length. The production rules of the grammar are shown in figure 4. As above, the PonyGE2 implementation is used to evolve a program, which in this case modifies an existing solution. Definitions of the terminal nodes used in the production rules can be found in [**?**].

The operator constructed by the grammar can be thought of as a form of k-opt [13] that is configurable and includes extra functions that determine where to break a sequence. However, the formulation and implementation is *vertex* centric rather than *edge* centric. The mechanics of the algorithm are as follows:

Number of cuts: This determines in how many places a sequence will be cut creating $(k-1)$ subsequences where k is the number of cuts. The number of possible loci of the cuts is equal to $n+1$, where n is the number of vertices (the sequence can be cut both before the first element and after the last element).

Location of cuts: The grammar associates a strategy to each cut that will determine the location of the specific cut. A strategy may contain a reference location

```
<op>              →  addCut(<loci_ref>,<distance>)
                     <c>
                     <ExtraCuts>
                     Iteration_effect(<motion>,<loci_computation>)
                     permutation(<perm_behaviour>)
                     inversions(<inv_behaviour>)
<ExtraCuts>       →  ∅ | <c> | <c><c> | <c><c><c>
<c>               →  addCut(<loci_ref>,<distance>)
                     isInverted(<invert>)
                     permutationFactor(<r>)
<motion>          →  'random' | 'oscillate' | 'steps' | 'none'
<loci_ref>        →  'none' | 'left' | 'right' | 'limit'
<distance>        →  'linear' | 'negative_binomial',(<r>,<p>)
<r>               →  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
<p>               →  0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7
<loci_computation> →  'once' | 'always'
<perm_behaviour>  →  'fixed' | 'random'
<inv_behaviour>   →  'fixed' | 'random'
<invert>          →  0 | 1
```

Fig. 4: Complete grammar for generation of *perturbative* heuristics

such as the ends of the sequence or subsequence, a specific place in the sequences or a random location. The reference can be used together with a probability distribution that determines the chances of any given location to be the place of the next cut. These probability distributions *de facto* regulate the length of each subsequence. Two probability distributions can be selected by the grammar: a discretised triangular distribution and a negative binomial distribution. An example can be seen in fig.5-A and 5-B.

After the cutting phase the subsequences are given symbols with S being always the leftmost subsequence and E being the rightmost subsequence such as in fig. 5-C. The start and end sequences (S,E) are never altered by the evolved operator which only acts on the sequences labelled α-β in fig. 5-C. Note that subsequences may be empty. This can happen if the leftmost cut is on the left of the first element (leaving S empty), if the rightmost cut is after the last element (leaving E empty) or if two different cuts are applied in the same place.

Permutation of the subsequence: After cutting the sequence the subsequences becomes the units of a new sequence. The grammar can specify if the subsequence will be reordered to a specific permutation (including the identity, i.e no change) or to a random permutation. An example can be seen in 6-a.

5 Methodology

We conduct experiments in the two domains described above. An experiment consists of two phases: a *training* phase in which GE is used to evolve a heuristic on

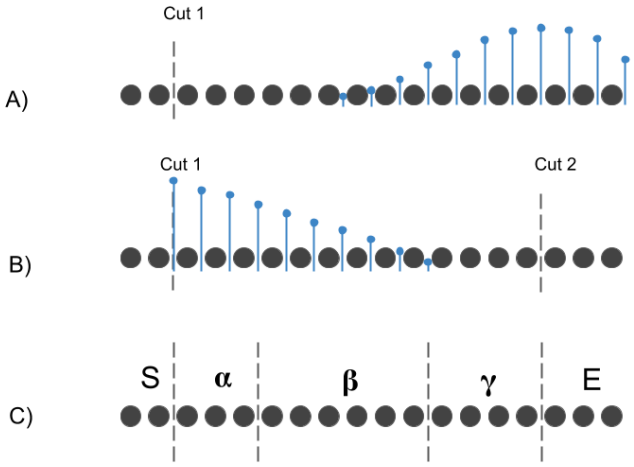
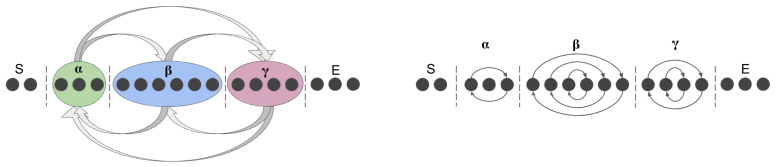


Fig. 5: A) Example of a sequence with one cut and a probability mass function that will decide the loci of the second cut. B) Both cuts now shown C) Final set of subsequences after k-cuts



(a) Subsequence permutation (b) Subsequence inversion

Fig. 6: Example perturbations of the subsequences produced by the grammar

small training set of randomly generated instances. This is followed by a *test* phase in which heuristics evolved from multiple runs of the GE algorithm are evaluated on a test set of benchmark instances taken from the literature. The next two sections describe the procedures by which the training data is generated for both constructive and perturbative approaches, and provide the details of the experimental set up in each case.

5.1 Training Phase

A set of 10 random instances is synthesised in order to train the heuristic generator in each case. The reader is referred to [29, 30] for an exact description of each synthesiser and the associated parameters in each domain, however a brief outline is repeated here.

For TSP, a set of instances each containing n cities are generated using a uniform-random distribution on a 2D plane. For MKP, instances have m objects with k constraints each. Each constraint is a sample from a uniform random distribution with a specified range o. The profits of each object are taken from a normal distribution with mean equal to the sum of the constraints and fixed standard deviation sd_p. The constraints of the knapsack are sampled from a normal distribution[2] with mean c and standard deviation sd_c.

During evolution of constructive heuristics, the fitness of a heuristic on the training set in both domains is calculated as the *median* fitness returned from the set of training instances. When evolving perturbative heuristics, the fitness evaluation consists of applying the heuristics as the move-operator within a hill-climbing algorithm to each of the training instances starting from an randomly initialised solution. The hill-climber runs for x iterations with an improvement only acceptance criteria. The fitness at the end-point is averaged over the 5 instances and assigned to the heuristic.

All experiments are repeated in each domain 10 times, with a new set of training instances generated for each run. All parameters of the PonyGE algorithm to evolve the heuristics using the specified grammars can be found in [29, 30]. The best performing heuristic from each run is retained, creating an ensemble of 10 heuristics as a result to be used in the test phase.

5.2 Test Phase

For the purposes of evaluating the evolved heuristics, we select an indicative set of benchmarks from the each domain. The evolved heuristics are compared to well-known approaches from the literature in each case. The work in [29, 30] provides extensive evaluation results over large numbers of benchmarks. In this chapter for clarity we select representative results for presentation.

In the TSP domain, we provide results on 5 benchmarks instances taken from the TSPLib[3]. In the case of *constructive* heuristics, we compare results to the known optimal, as well as two well known constructive heuristics from the literature, the nearest-neighbour heuristic [17] and the MST heuristic [17]. Both these heuristics are deterministic. For evaluation of the perturbative heuristics, we use the same

[2] We recognise that real-instances are unlikely to be uniformly distributed: our implementation therefore represents the worst-case scenario in which the system can be evolved

[3] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

instances and compare to the best known perturbative heuristic 2-opt [8], using the R TSPLIB implementation[4]. As this is a stochastic method, it is repeated 50 times.

In the MKP domain, constructive heuristics are compared to a greedy depth first search algorithm [15] which also constructs solutions. As the vast majority of published results in this domain use meta-heuristic approaches, there are no obvious heuristic methods to use a comparison, Therefore we compare with two meta-heuristic approaches from [5], the Chaotic Binary Particle Swarm Optimisation with Time Varying Acceleration Coefficient (CBPSO), and an improved version of this algorithm that includes a self-adaptive check and repair operator (SACRO CBPSO), the most recent and highest-performing methods in MKP optimisation. Both algorithms use problem specific knowledge: a penalty function in the former, and a utility ratio estimation function in the latter, with a binary representation for their solution. Both are allocated a considerably larger evaluation budget than our experiments. The heuristics evolved using our approach would not be expected to outperform these approaches — however, we wish to investigate whether the approach can produce solutions within reasonable range of known optima that would be acceptable to a practitioner requiring a quick solution.

In all testing experiments, the 10 heuristics in the ensemble created in the training phase are applied to each test instance. In the constructive case, each heuristic is applied once to each test instance to construct a solution. In the perturbative domain, each of the 10 evolved heuristics are applied 5 times to a randomly initialised solution using an improvement only acceptance criteria (hill-climber). We record the average performance of each heuristic over 5 runs, as well as the best, and the worst. These values were selected following a minimal amount of empirical investigation, motivated by the desire to develop a system that could quickly generate a set of new heuristics (hence choice of a small ensemble), and to quickly provide a reasonable estimate of heuristic fitness (hence a small number of replications).

6 Results

We present the results first for the experiments in which constructive heuristics were evolved, followed by those for the evolution of perturbative heuristics. Recall from above that two grammars are specified, one for constructive heuristics and one for perturbative heuristics. The constructive grammar is used without modification to generate heuristics for two distinct domains (TSP and MKP). Similarly, the perturbative grammar is also used without modification in both domains.

[4] https://cran.r-project.org/web/packages/TSP/TSP.pdf

6.1 Constructive Heuristics

The results in table 1 show a comparison of the best, worse and median fitness of the evolved heuristics on each of the TSP test instances. As explained above, these are compared to the deterministic values obtained from a single run of the human designed heuristics. The best evolved heuristic is better than both human heuristics in all 5 cases, while in 3 out of 5 instances the median performance of the evolved heuristics is better than both the simple human heuristics.

The results for the MKP tests are show in table 2 which compares the performance of the evolved heuristics to a greedy constructive heuristic (run 10 times), where the heuristic try to fit each object in the knapsack if there is sufficient space using a different ordering each time and to random construction. The best of evolved heuristics outperforms the greedy (deterministic) heuristic in 4 out of 7 instances. It is also of note that in 2 cases, the global optima is obtained by the best heuristics. The worst performing evolved heuristics outperform the greedy heuristic in 3 out of 7 instances. Statistical analysis using a Wilcoxon rank test fails to reject the null hypothesis that constructiveGE produces the same results as the greedy construction method at a significance level of 0.05 for all instances.

Instance name	Optima	NN	MST	constructive-GE		
				Median	Best	Worst
berlin52	7542	8868	10404	9196	**8452**	*10515*
ch130	6110	7575	8277	**7501**	**6942**	*8469*
eil101	629	826	846	**803**	**736**	897
eil51	426	521	605	547	**451**	*620*
eil76	538	700	739	**652**	**603**	678

Table 1: Results obtained from generating reusable constructive heuristics for the TSP domain

Instance	optima	rand			greedy			constructive-GE		
		worst	median	best	worst	median	best	worst	median	best
mknap1-1	3800	100	1700	3100	1200	2700	**3800**	**1800**	**3300**	**3800**
mknap1-2	8706.1	1482.1	5059	8687.5	4340.7	6504.8	8650.1	4212.4	**7059.8**	**8706.1**
mknap1-3	4015	985	2235	2860	1895	**3325**	**3765**	**2390**	2480	3725
mknap1-4	6120	1320	3240	5820	2460	**3525**	5390	**2480**	3020	**5640**
mknap1-5	12400	3770	7770	10340	**7590**	**8990**	**11550**	6855	8150	10510
mknap1-6	10618	4286	6566	9770	**7400**	**8032**	**10345**	7238	7641	9352
mknap1-7	16537	5661	9509	12769	**8770**	**12363**	15330	8335	10887	**15668**

Table 2: Results obtained from generating reusable constructive heuristics for the MKP domain

7 Results: Perturbative Heuristics

Table 3 shows the best, worst and median performance of the evolved heuristics and the two-opt based algorithm for TSP. The median result obtained by the evolved heuristic improved on 2-opt in each case, with the evolved heuristics finding the best single result on 4 out of 5 instances. A Wilcoxon Rank-sum test applied to compare the two treatments on each instance enables us to reject the null-hypothesis in each case, i.e improvements are statistically significant at the 5% level.

In table 4 we present results in the MKP instances. The table compares the Average Success Rate (ASR) across all instances grouped by dataset against the results presented by [5] on 2 versions of SACRO algorithms and an additional fish-swarm method. The results given in the table are taken directly from this paper. In [5], ASR is calculated as the number of times the global optima was found for each instance divided by the number of trials. For *perturbativeGE*, we define a trial as successful if at least one of the 10 heuristics found the optima in the trial, and repeat this 5 times. Despite the fact that the our new perturbative heuristics have no domain-specific information and are simplistic compared to the specialised metaheuristic methods we compare to, it can be seen that the results are comparable to those of specialised algorithms. In fact, perturbativeGE outperforms the specialised methods on Weing dataset.

	perturbativeGE			2-opt			Rank sum p-value
	Best	**Worst**	**Median**	**Best**	**Worst**	**Median**	
berlin52	7793	8825	**8170**	**7741**	9388	8310	0.0033
ch130	**6418**	7108	**6722**	6488	7444	6984	0.0030
eil101	**674**	739	**702**	680	749	709	0.0073
eil51	**435**	484	**456**	442	494	473	$\ll 0.001$
eil76	**563**	616	**593**	583	628	611	$\ll 0.001$

Table 3: Results obtained from generating perturbative heuristics for use in the TSP domain.

Problem Set	Instances	Average Success Rate			
		IbAFSA	BPSOTVAC	CBPSOTVAC	**perturbativeGE**
Sento	2	1.000	0.9100	0.9100	0.90
Weing	8	0.7875	0.7825	0.7838	0.80
Weish	30	0.9844	0.9450	0.9520	0.907
Hp	2	0.9833	0.8000	0.8600	1.00
Pb	6	1.000	0.9617	0.9517	0.967
Pet	6	*na*	*na*	*na*	1.00

Table 4: Comparison of results obtained from the perturbative GE algorithm with the latest specialised meta-heuristics from the literature: a fish-swarm algorithm IbAFSA and the two most recent SACRO algorithms, results taken directly from [5]. *na* indicates that no results were provided in [5] on this dataset

8 Discussion

In this article, we set out a method of tackling the issue of generating new low-level heuristics that could be used across the domain-barrier in a hyper-heuristic. Unlike previous approaches to low-level heuristic generation that are customised to a particular domain, the results presented above show that a *single grammar* can be used to generate heuristics for two different domains, without modification. [5]. This is made possible by assuming a common representation of a problem, in this case as a graph. While this may appear somewhat restrictive, it is clear that many practical problems can be represented in this form. In fact several renowned industrial applications have used graph representations such as Google's page rank algorithm [21], Amazon and Netflix recommendation system[3], Drug Discovery [1], General Electric's power distribution system[9] and GSM mobile phone network frequency assignment[11].

Motivated by the general challenge of designing heuristics for new domains in which few examples of instances are available, we trained the two heuristic generators (one for constructive heuristics, one for perturbative heuristics) on very small training sets containing randomly synthesised instances. Despite this, the evolved heuristics are shown to capable of outperforming well-known simple heuristics on many benchmark instances in the TSP domain and find comparable results to specialised meta-heuristics in the MKP domain. Therefore, it is reasonable to consider the method *cross-domain* for the set of problem domains that can be represented in graph form. The grammar used to evolve the heuristics is exclusively composed of geometric properties and functions that are not tied to any specific problem domain, and we only implement graph-manipulation techniques. Figure 7 shows examples of a constructive heuristic evolved in each of the two domains that returns a priority for each vertex (with the highest priority vertex then being added to the solution). The heuristics exploit geometric properties of the graphs. For instance, in the MKP domain, the *cosine* distance appears in 97% of evolved heuristics with the *Euclidean* distance measure only appearing in 17%: in contrast, in TSP, the *Euclidean* distance metric appears in 100% of evolved heuristics, with the *cosine* metric only appearing in 12%.

Clearly, there remains considerable scope for training using much larger instance sets, or non-random instances. In this respect, our results are representative of the worst-case performance of the system. As noted at the start, the motivation behind the research is not to develop heuristics that outperform very specialised methods — there will always be a trade-off between specialist and generalist heuristics in terms of performance. However, to be useful, the heuristics should have acceptable performance. The results presented clearly demonstrate that this is the case.

[5] note that different grammars are required depending on whether one wishes to generate constructive or perturbative heuristics

```
a) psqrt(distance_to_v0(vertex,'cosine'))-
sin(plog(exp(distance_osms(vertex,'cosine'))))

b)distance_osms(vertex,'euclidean')-
exp(cos(distance_to_v0(vertex,'euclidean')))
```

Fig. 7: Examples of evolved constructive heuristics for a) MKP and b) TSP domain. Each heuristic returns a priority for a vertex; the vertex with the highest priority is chosen. For example the MKP heuristic is interpreted as '*The priority of the current vertex is equal to: the root of the cosine distance between the current vertex and the container vertex minus the sine of the distance between the last vertex of the chain and the current vertex' etc.*

9 Conclusion

The article has presented a method that demonstrates it is possible to go below the hyper-heuristic domain-barrier and use a cross-domain grammar, without modification, to generate new heuristics for each of two separate domains. The generation method is cross-domain, while the heuristics it produces are specialised to each individual domain considered. The article synthesises previous work which was reported in [29, 30] where expanded results and analysis can found. Given that a graph-based representation enables a rich and diverse set of application domains to be represented, the approach augments existing hyper-heuristic methods, particularly in being able to provide a source of low-level heuristics for new types of problem classes for which low-level heuristics are unavailable. Importantly, it removes the need for expertise in either heuristic design or the domain itself. While of course domain-expertise will result in the creation of high-quality heuristics, we have demonstrated that in fact our approach is able to generate high-performing heuristics that are comparable to (and occasionally better than) existing methods.

Many improvements are possible to the method itself. This includes expanding the components of the grammar that currently uses only a fraction of the possible geometric information derivable from a graph, and extensive tuning of the parameters of the approach. The ability to generate new heuristics for a domain also opens up the ability of improving existing hyper-heuristic methods that operate *above* the domain-barrier by extending the set of heuristics available for selection.

References

1. Amigó, J., Gálvez, J., Villar, V.: A review on molecular topology: applying graph theory to drug discovery and design. Naturwissenschaften (2009)
2. Bader-El-Den, M., Poli, R.: Generating sat local-search heuristics using a gp hyper-heuristic framework. In: International Conference on Artificial Evolution (Evolution Artificielle), pp. 37–49. Springer (2007)

3. Bogers, T.: Movie recommendation using random walks over the contextual graph. Proc. of the 2nd Intl. Workshop on Context-Aware (2010). URL http://ids.csom.umn.edu/faculty/gedas/cars2010/bogers-cars-2010.pdf
4. Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Qu, R.: Hyper-heuristics: A survey of the state of the art. Journal of the Operational Research Society **64**(12), 1695–1724 (2013)
5. Chih, M.: Self-adaptive check and repair operator-based particle swarm optimization for the multidimensional knapsack problem. Applied Soft Computing **26**, 378–389 (2015)
6. Chu, P.C., Beasley, J.E.: A genetic algorithm for the multidimensional knapsack problem. Journal of heuristics **4**(1), 63–86 (1998)
7. Cowling, P., Kendall, G., Soubeiga, E.: A hyperheuristic approach to scheduling a sales summit. In: International Conference on the Practice and Theory of Automated Timetabling, pp. 176–190. Springer (2000)
8. Croes, G.A.: A method for solving traveling-salesman problems. Operations research **6**(6), 791–812 (1958)
9. Elgerd, O., Happ, H.: Electric Energy Systems Theory: An Introduction. IEEE Transactions on Systems, Man, and (1972)
10. Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., Hemberg, E., O'Neill, M.: PonyGE2. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '17, pp. 1194–1201. ACM Press, New York, New York, USA (2017). DOI 10.1145/3067695.3082469. URL http://dx.doi.org/10.1145/3067695.3082469
11. Gamst, A.: Application of graph theoretical methods to GSM radio network planning. Circuits and Systems, 1991., IEEE International (1991)
12. Hart, E., Sim, K.: A hyper-heuristic ensemble method for static job-shop scheduling. Evolutionary computation **24**(4), 609–635 (2016)
13. Helsgaun, K.: General k-opt submoves for the lin–kernighan tsp heuristic. Mathematical Programming Computation **1**(2-3), 119–163 (2009)
14. Keller, R.E., Poli, R.: Linear genetic programming of parsimonious metaheuristics. In: 2007 IEEE Congress on Evolutionary Computation, pp. 4508–4515. IEEE (2007). DOI 10.1109/CEC.2007.4425062. URL http://ieeexplore.ieee.org/document/4425062/
15. Knuth, D.E.: The art of computer programming: sorting and searching, vol. 3. Pearson Education (1998)
16. Koza, J.: Genetic programming: on the programming of computers by means of natural selection (1992)
17. Laporte, G.: A concise guide to the traveling salesman problem. Journal of the Operational Research Society **61**(1), 35–40 (2010)
18. Mascia, F., López-Ibánez, M., Dubois-Lacoste, J., Stützle, T.: From grammars to parameters: Automatic iterated greedy design for the permutation flow-shop problem with weighted tardiness. In: International Conference on Learning and Intelligent Optimization, pp. 321–334. Springer (2013)
19. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Transactions on Evolutionary Computation **5**(4), 349–358 (2001)
20. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Springer (2003). DOI 10.1007/978-1-4615-0447-4_4
21. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the web. (1999). URL http://ilpubs.stanford.edu:8090/422
22. Pferschy, U., Schauer, J.: The knapsack problem with conflict graphs. J. Graph Algorithms Appl. **13**(2), 233–249 (2009)
23. Pillay, N., Banzhaf, W.: A study of heuristic combinations for hyper-heuristic systems for the uncapacitated examination timetabling problem. European Journal of Operational Research **197**(2), 482–491 (2009)
24. Robu, V., Somefun, D., La Poutré, J.A.: Modeling complex multi-issue negotiations using utility graphs. In: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, pp. 280–287. ACM (2005)

25. Sabar, N.R., Ayob, M., Kendall, G., Qu, R.: Grammatical Evolution Hyper-Heuristic for Combinatorial Optimization Problems. IEEE Transactions on Evolutionary Computation **17**(6), 840–861 (2013). DOI 10.1109/TEVC.2013.2281527
26. Sim, K., Hart, E.: A combined generative and selective hyper-heuristic for the vehicle routing problem. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, pp. 1093–1100. ACM (2016)
27. Sim, K., Hart, E., Paechter, B.: A hyper-heuristic classifier for one dimensional bin packing problems: Improving classification accuracy by attribute evolution. In: International Conference on Parallel Problem Solving from Nature, pp. 348–357. Springer (2012)
28. Stone, C., Hart, E., Paechter, B.: Automatic generation of constructive heuristics for multiple types of combinatorial optimisation problems with grammatical evolution and geometric graphs. In: International Conference on the Applications of Evolutionary Computation (EvoStar), pp. 578–593. Springer (2018)
29. Stone, C., Hart, E., Paechter, B.: Automatic generation of constructive heuristics for multiple types ofcombinatorial optimisation problems with grammatical evolution andgeometric graphs. In: K. Sim, P. Kaufmann (eds.) Applications of Evolutionary Computation, pp. 578–593. Springer International Publishing (2018)
30. Stone, C., Hart, E., Paechter, B.: On the synthesis of perturbative heuristics for multiple combinatorial optimisation domains. In: International Conference on Parallel Problem Solving from Nature, pp. 170–182. Springer (2018)
31. Tay, J.C., Ho, N.B.: Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. Computers & Industrial Engineering **54**(3), 453–473 (2008)