On the Synthesis of Perturbative Heuristics for Multiple Combinatorial Optimisation Domains

Christopher Stone, Emma Hart, Ben Paechter

School of Computing, Edinburgh Napier University, Scotland, UK
{c.stone,e.hart,b.paechter}@napier.ac.uk

Abstract. Hyper-heuristic frameworks, although intended to be cross-domain at the highest level, rely on a set of domain-specific low-level heuristics at lower levels. For some domains, there is a lack of available heuristics, while for novel problems, no heuristics might exist. We address this issue by introducing a novel method, applicable in multiple domains, that constructs new low-level heuristics for a domain. The method uses grammatical evolution to construct iterated local search heuristics: it can be considered cross-domain in that the *same* grammar can evolve heuristics in multiple domains without requiring any modification, assuming that solutions are represented in the same form. We evaluate the method using benchmarks from the travelling-salesman (TSP) and multi-dimensional knapsack (MKP) domain. Comparison to existing methods demonstrates that the approach generates low-level heuristics that out-perform heuristic methods for TSP and are competitive for MKP.

1 Introduction

The *hyper-heuristic* method was first introduced in an attempt to raise the generality at which search methodologies operate [2]. One of the main motivations was to produce a method that was cheaper to implement and easier to use than problem specific special purpose 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. The low-level heuristics are specific to a particular domain, and may be designed by hand, relying on intuition or human-expertise [2], or can be evolved by methods such as Genetic Programming [14]. The success of the high-level heuristic is strongly influenced by the number and the quality of the low-level heuristics available. Given a new problem domain that does not map well to well-studied domains in the literature, it can be challenging to find a suitable set of low-level heuristics to utilise with a hyper-heuristic. Although this can be addressed through evolving new heuristics [1], this process requires in-depth understanding of the problem and effort designing a specialist algorithm to evolve the heuristic. We propose to address this by introducing a method of creating new heuristics that is *cross-domain*, that is, the method can be used without modification to create heuristics in multiple domains, assuming a common problem representation.

As a step towards raising the generality of creating *low-level* heuristics, we focus on domains that can be mapped to a graph-based representation. This includes obvious applications such as routing and scheduling [14], as well as many less obvious ones including packing problems [11] and utility maximisation in complex negotiations[12]. We describe a novel method using grammatical evolution that produces a set of local-search heuristics for solving travelling-salesperson (TSP) problems, and another for multi-dimensional knapsack (MKP) problems. In each case, an identical grammar is used to evolve heuristics that modifies a permutation representing a TSP or MKP problem. The grammar is trained on a small subset of randomly generated instances in each case and shown to produce competitive results on benchmarks when compared to human design heuristics and almost as good as specially design meta-heuristics.

This 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 contributions are threefold: (1) it describes a novel grammar that generates mutation operators that perturb a permutation via partial permutations and inversions; (2) the grammar is trained to produce single instances of new 'move' operators using a *very small set* of randomly generated instances from each problem domain; (3) it demonstrates that competitive results can be obtained from a generic grammar, even when using a representation that is not necessarily considered the most natural for a domain.

2 Background

Hyper-Heuristics are class of algorithms that explore the space of heuristics rather than the space of solutions, and have found application in a broad range of combinatorial optimisation domains [2]. As previously mentioned, the core idea is to create a *generic* algorithm that selects and applies heuristics, separated by a domain-barrier from a subset of low-level domain-specific heuristics. Most initial work focused on development of the generic controlling algorithms [2]. More recent attention has focused on the role of the low-level heuristics themselves. Low-level heuristics fall into two categories [2]. *Constructive* heuristics build a solution from scratch, adding an element at a time, e.g. [14]. On the other hand, *perturbative* heuristics modify an existing solution, e.g. re-ordering elements in a permutation [4] or modifying genes [2].

In many practical domains, hand-designed low-level heuristics are readily available, e.g. [2]. However, a tranche of research has focused on generation of *new* heuristics, typically using methods from Genetic Programming [1],Grammatical Evolution [13] [8] and Memetic Algorithms [6]. Specifically in the domain of *perturbative* heuristics, GP approaches to generating novel local search heuristics for satisfiability testing were proposed by [2]. Grammatical Evolution is applied to evolve new local-search heuristics for 1d-bin packing in [2, 7]. It is also worth mentioning the progress made in cross-domain optimisation thanks to HyFlex [9]: however, note that here the controlling hyper-heuristics are cross-domain but the framework still relies on pools of domain *specific* low-level heuristics.

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 Method

Our generator makes use of Grammatical Evolution [10] for the production of new heuristics. In particular we specify *one* grammar and this single grammar is used to produce heuristics in two different domains. Our method can be described by three fundamental steps:

- Represent the problem-domain of interest as an ordering problem
- Use Grammatical Evolution to breed heuristics that perturb the order of a solution, using a small training set of examples. The new heuristics are evaluated according their effectiveness as a mutation operator in an iterated local-search algorithm.
- Re-use the evolved heuristics on unseen instances from the same domain

3.1 Grammatical Evolution

Grammatical Evolution (GE) is a population based evolutionary computation approach used to construct sequence of symbols in an arbitrary language defined by a BNF grammar. A BNF Grammar consist of a set of *production rules* composed of terminal and non-terminal nodes. The production rules are used to substitute the non-terminal nodes with other nodes, which can be both non-terminal or terminal nodes, repeatedly until a whole sequence of terminal nodes is composed. Each non terminal node has its own set of production rules. *Codons* (represented as a single integer) specify which specific production rule should be chosen at each step.

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. Our implementation uses the GE library described by Fenton *et al* [5]. This version of GE proved to be accessible, straightforward to reuse, and is the most recent version of GE. A detailed description of the complete implementation can be found in [5]. The code is also open-source and available on *github*[1]. The main implementation details relevant to this work are as follows:

Genome: Fenton's implementation uses a linear genome representation that is encoded as 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 the codon, i.e. $\textit{Selected node} = c \mod n$, 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.

Mutation: An integer flip at the level of the codons is used. One of the codons that has been used for the phenotype is changed each iteration and substituted with a completely new codon.

Crossover: Variable one-point crossover, where the crossing point between 2 individuals is chosen randomly.

Replacement: Generational replacement strategy with elitism 1, i.e one genome is guaranteed to stay in the pool on the next generation.

3.2 Grammar and mechanics of the operator

The operator constructed by our grammar can be thought of as a form of k-opt, that is configurable and includes extra functions to determine where to break a sequence. The formulation and implementation is vertex centric instead of 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 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.1-A and 1-B.

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

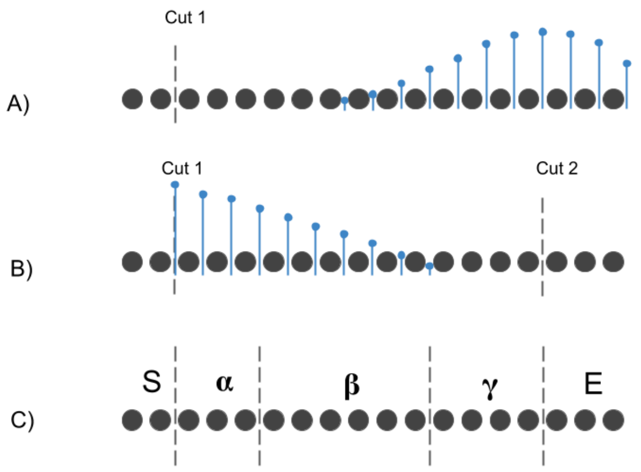


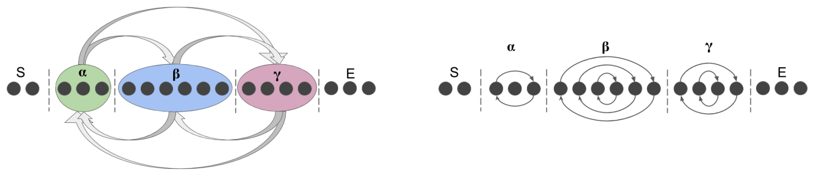
Fig. 1: A) Example of a sequence with one cut and a probability mass function that will decide the loci of the second cut. B) Both cuts now shown C) Final set of subsequences after k-cuts

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. 1-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. 1-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 2-a.

Inversion of the subsequences: The grammar specifies whether the order of each specific subsequence should be reversed or if the reversing should be decided randomly for each subsequence each iteration.

Iteration effect: Another component of the grammar is the iteration effect which may associate a specific function that regulate the changein the initial cutting location at each iteration. We have specified four types of effect: *random*, which means that the starting location of the first cut will be random; *oscillate* that makes the starting position move in a wave like manner and returns to the initial loci after a number of iterations; *step* simply moves one step on the right of the previous starting position and finally *none* which has no effect.



(a) Subsequence permutation (b) Subsequence inversion

Fig. 2: Example perturbations of the subsequences produced by the grammar

3.3 Problem Domains and Training Examples

We apply the grammar in two problem domains. The *Travelling Salesman Problem* (TSP) is one of the most studied problems in combinatorial optimisation, in which a tour passing by all points must be *minimised*. Due to the fact that it is naturally encoded as an ordering problem represented by a permutation it plays the role of *base case* for our experiments.

The *Multidimensional Knapsack Problem* (MKP) is another of the most studied problem in combinatorial optimisation with applications in budgeting, packing and cutting problems. In this case the profit from items selected among a collection must be *maximised* while respecting the constraints of the knapsack. This problem is chosen as in its typical form, it is not represented as ordering problem. However, a formulation based on chains and graphs was recently introduced in [15]. The goal here is to demonstrate that the approach can produce acceptable heuristics from a generic representation, without requiring the expert knowledge required to formulate a problem-specific approach.

A set of heuristics is evolved in each domain, using a set of example training instances in each case. It is well known that having better training instances leads to better outcomes [2]. However, as the ultimate goal of this work is produce a system that can produce acceptable heuristics in an unknown domain in which good training examples might not be available (or in an existing domain in which we cannot predict characteristics of future problems) we synthesise a random set of training instances in each case. Parameters of the synthesisers are given in table 1. 5 TSP instances are synthesised using a uniform random distribution. Each instance has 100 cities placed in a 2D Euclidean plane. For MKP, each of 5 instances has 100 objects with 10 constraints. Each constraint is a sample from a uniform random distribution between 0 and 100. The profits of each object are taken from a normal distribution with mean equal to the sum of the constraints and standard deviation 50. The constraints of the knapsack are sampled from a normal distribution with mean 2500 and standard deviation 300. 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.

```
<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. 3: Grammar used to produce the local search operator

4 Experiments

Training Phase: One-point local-search heuristics are generated using an off-line learning approach. The system is applied *separately* to each domain, but uses an *identical* grammar in both. At each iteration of the GE, each heuristic in the population is applied within a hill-climbing algorithm to each of the 5 training instances starting from an randomly initialised solution. The hill-climber runs for x iterations with an improvement only acceptance criteria. For TSP, $x = 1000$ and for MKP, $x = 2500$ (based on initial experimentation). The fitness at the end-point is averaged over the 5 instances and assigned to the heuristic (i.e. distance for TSP and profit for MKP). Experiments are repeated in each domain 10 times, with a new set of 5 problems generated for each run. The best performing heuristic from each run is retained, creating an ensemble of 10 heuristics as a result. All the parameters of the synthesisers are give in table 1a while the GE parameters are in table 1b.

Testing Phase: The generated ensemble is tested on benchmark instances from the literature. For TSP, we use 19 problems taken from the TSPlib. MKP heuristics are tested on at total of 54 problems from 6 benchmark datasets from the OR-library. Each of the 10 heuristics is applied 5 times to each problem for 10^5 iterations, starting from 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.

For TSP, we compare the results with 50 runs per instance of a classic two opt algorithm[2], chosen as a commonly used example of high-performing local-search

[2] using the R package TSPLIB

heuristic. For MKP, the vast majority of published results use meta-heuristic approaches. We compare with two approaches from [3], 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.

Parameter	Value
Number of cities	100
Cities distribution type	Uniform
Cities distribution range	0-100
Number of objects	100
Number of constraints	10
Object constraints distribution	Uniform
Object constraints range	0-100
Object profit distribution	Normal
Object profit mean	Sum of constraints
Object profit deviation	50
Knapsack constraints dist.	Normal
Knapsack constraints mean	2500
Knapsack constraints deviation	300

(a) Problem synthesisers

Parameter	Value
Generations	80
Population	100
Mutation	int flip
Crossover Prob.	0.80
Crossover type	one point
Max initial tree	10
Max tree depth	17
Replacement	generational
Tournament size	2

(b) Grammatical Evolution

Table 1: Experimental Parameters

5 Results and Analysis

We refer to our algorithm as *HHGE* in all reported results. Table 3 shows the best, worst and median performance of the evolved heuristics and the two-opt based algorithm for TSP. With the exception of a single case, the evolved heuristics perform better in term of best, worst and median results. For each instance, we apply a Wilcoxon Rank-sum test on the 50 pairs of samples, and provide a p-value in the rightmost column. Improvements are statistically significant at the 5% level in all cases.

Results for MKP are reported in table 2, averaged over 10 heuristics in each case. Note that despite the simplistic nature of our approach — a hill-climber

	HHGE					CBPSO		SACRO-BPSO	
Instance	Best	Worst	Average	Median	Optima	Best	Average	Best	*Average*
hp1	3418	3385	**3410.56**	3418	3418	3418	3403.9	3418	*3413.38*
hp2	3186	2997	3171.54	3186	3186	3186	3173.61	3186	*3184.74*
pb1	3090	3057	**3083.32**	3090	3090	3090	3079.74	3090	*3086.78*
pb2	3186	3114	**3179.88**	3186	3186	3186	3171.55	3186	*3186*
pb4	95168	90961	93515.54	93897	95168	95168	94863.67	95168	*95168*
pb5	2139	2085	**2120.06**	2130.5	2139	2139	2135.6	2139	*2139*
pb6	776	641	733.12	735.5	776	776	758.26	776	*776*
pb7	1035	983	1018.9	1025	1035	1035	1021.95	1035	*1035*
pet2	87061	78574	85409.32	87061	87061	-	-	-	-
pet3	4015	3165	3955.8	4015	4015	-	-	-	-
pet4	6120	5440	6040.2	6110	6120	-	-	-	-
pet5	12400	12090	12363.1	12400	12400	-	-	-	-
pet6	10618	10107	10592.1	10604	10618	-	-	-	-
pet7	16537	15683	16504.48	16537	16537	-	-	-	-
sento1	7772	7491	**7706.92**	7749.5	7772	7772	7635.72	7772	*7769.48*
sento2	8722	8614	**8691.02**	8704	8722	8722	8668.47	8722	*8722*
weing1	141278	135673	140619.36	141278	141278	141278	141226.8	141278	*141278*
weing2	130883	118035	128542.94	130712	130883	130883	130759.8	130883	*130883*
weing3	95677	77897	93099.5	94908	95677	95677	95503.93	95677	*95676.39*
weing4	119337	100734	117811.56	119337	119337	119337	119294.2	119337	*119337*
weing5	98796	78155	95912	98475.5	98796	98796	98710.4	98796	*98796*
weing6	130623	117715	129452.56	130233	130623	130623	130531.3	130623	*130623*
weing7	1095382	1088277	**1093583.14**	1093595	1095445	1095382	1084172	1095382	*1094349*
weing8	624319	525663	**606175.12**	613070	624319	624319	597190.6	624319	*622079.9*
weish01	4554	4298	4494.34	4530	4554	4554	4548.55	4554	*4554*
weish02	4536	4164	4485.12	4536	4536	4536	4531.88	4536	*4536*
weish03	4115	3707	3963.08	3985	4115	4115	4105.79	4115	*4115*
weish04	4561	3921	4385.5	4455	4561	4561	4552.41	4561	*4561*
weish05	4514	3754	4265.56	4479.5	4514	4514	4505.89	4514	*4514*
weish06	5557	5238	5503.16	5538	5557	5557	5533.79	5557	*5553.75*
weish07	5567	5230	5496.56	5542	5567	5567	5547.83	5567	*5567*
weish08	5605	5276	5534.82	5597.5	5605	5605	5596.16	5605	*5605*
weish09	5246	4626	5062.24	5128	5246	5246	5232.99	5246	*5246*
weish10	6339	5986	6244.82	6314	6339	6339	6271.84	6339	*6339*
weish11	5643	5192	5522.18	5631.5	5643	5643	5532.15	5643	*5643*
weish12	6339	5951	6217.14	6322.5	6339	6339	6231.5	6339	*6339*
weish13	6159	5780	6032.28	6056	6159	6159	6120.38	6159	*6159*
weish14	6954	6581	6827.9	6852	6954	6954	6837.77	6954	*6954*
weish15	7486	7113	**7391**	7445.5	7486	7486	7324.55	7486	*7486*
weish16	7289	6902	7154.82	7159.5	7289	7289	7288.7	7289	*7288.7*
weish17	8633	8506	**8609**	8633	8633	8633	8547.71	8633	*8633*
weish18	9580	9310	**9527**	9560.5	9580	9580	9480.86	9580	*9578.46*
weish19	7698	7272	7505.3	7527	7698	7698	7528.55	7698	*7698*
weish20	9450	9117	**9381.32**	9430	9450	9450	9332.11	9450	*9450*
weish21	9074	8655	**8972.9**	9025	9074	9074	8948.22	9074	*9074*
weish22	8947	8466	**8814.7**	8871	8947	8947	8774.2	8947	*8936.92*
weish23	8344	7809	**8202.06**	8217.5	8344	8344	8165	8344	*8344*
weish24	10220	9923	**10154.54**	10185.5	10220	10220	10106.28	10220	*10219.7*
weish25	9939	9667	**9872.48**	9909.5	9939	9939	9826.57	9939	*9939*
weish26	9584	9175	**9434.92**	9473	9584	9584	9313.87	9584	*9584*
weish27	9819	9244	**9652.3**	9671	9819	9819	9607.54	9819	*9819*
weish28	9492	8970	**9328.52**	9347.5	9492	9492	9123.26	9492	*9492*
weish29	9410	8794	**9217.28**	9279	9410	9410	9025.5	9410	*9410*
weish30	11191	10960	**11135.64**	11161	11191	11191	10987.21	11191	*11190.12*

Table 2: Generated heuristics vs specialised meta-heuristics from [3]. Highlighted values for HHGE indicate where it outperforms CBPSO. SACRO-BPSO performs best in all instances

	HHGE			2-opt			
	Best	**Worst**	**Median**	**Best**	**Worst**	**Median**	**Ranksum p-value**
berlin52	7793	8825	8170	7741	9388	8310	0.0033
ch130	6418	7108	6722	6488	7444	6984	0.0030
d198	16256	17033	16651	16400	18213	17291	$\ll 0.001$
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	$\ll 0.001$
kroA150	28109	31473	29344	29223	31994	30509	$\ll 0.001$
kroA200	31470	34528	32634	31828	35170	32893	0.0005
kroB150	27028	30283	28767	28114	30941	29134	$\ll 0.001$
kroB200	31315	35319	33029	31509	35077	33422	0.0455
kroC100	21418	24353	22885	22953	25503	23977	$\ll 0.001$
kroD100	21817	24405	23233	22772	26428	23430	$\ll 0.001$
kroE100	22660	25509	24178	23012	26695	24216	0.0021
lin105	14675	16965	15642	14966	17057	16191	$\ll 0.001$
pr107	45547	50313	47560	47597	51932	50002	0.0001
pr144	58847	68722	61534	59058	67272	64660	0.0002
pr152	75615	81458	78073	77307	81850	79964	$\ll 0.001$
pr226	81811	96484	86244	83566	101582	91512	0.0021
u159	44826	51353	47461	45297	51505	48124	0.1276

Table 3: Comparison between evolved heuristics and classic two-opt. For each instance we compute the Wilcoxon Rank-sum test using 50 pairs of samples

with an evolved mutation operator — our approach out-performs CBSPO in 22 out of 54 instances when considering average performance[3]. SACRO-BPSO (currently the best available meta-heuristic) performs better across the board, as expected.

In table 4 we compare the Average Success Rate (ASR) across all instances group by dataset against the results presented by [3] on 2 versions of SACRO algorithms and an additional fish-swarm method. In [3], ASR is calculated as the number of times the global optima was found for each instance divided by the number of trials. For HHGE, 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. It can be seen that the results are comparable to those of specialised algorithms, and in fact outperform these methods on Weing and HP sets.

6 Conclusions

We have presented a method based on grammatical evolution for generating perturbative low-level heuristics for multiple problem domains that is cross-domain: the same grammar generates heuristics for a domain that can be represented as an ordering problem. The method was demonstrated on two specific domains, TSP (a natural ordering problem) and MKP. We have compared the synthesised heuristics with a specialised human-designed heuristic in the TSP domain where the synthesised heuristic outperformed the well-known 2-opt heuristic. In the

[3] We do not provide statistical significance information as the PSO results, which are reported directly from [3], use a population based approach and vastly different number of evaluations

Problem Set	Instances	ASR			
		IbAFSA	BPSOTVAC	CBPSOTVAC	**HHGE**
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 with latest specialised meta-heuristics (PSO) from the literature: a fish-swarm algorithm IbAFSA and the two most recent SACRO algorithms, results taken directly from [3]

MKP domain, we compared the generated heuristics against two of the latest specialised meta-heuristics. The heuristics outperform one of these methods, and are at least comparable to the best method. We also note that the ensemble of 10 generated heuristics demonstrate high success rates in finding known optima when each heuristic is applied several times.

The approach represents the first steps towards increasing the cross-domain nature of hyper-heuristics: current approaches tend to focus on the high-level hyper-heuristic as cross-domain, while relying on specialised low-level heuristics below the domain barrier. Our approach extends existing work by also making methods for the automated generation of low-level heuristics cross-domain, without requiring specialist human-expertise. The proposed approach is applicable to a subset of domains that can be represented as ordering problems. While we believe this subset is large, it clearly does not include all domains. However, the same approach could be generalised to develop a portfolio of modifiable grammars, each addressing a broad class of problems.

Recall that in each case, HHGE was trained using a very small, uniformly generated set of instances, and in the case of MKP, applied to a non-typical representation, yet still provides acceptable results. We believe this fits with the original intention of hyper-heuristics, i.e. to provide quick and acceptable solutions to a range of problems with minimal effort. Although specialised representations and large sets of specialised training instances undoubtedly have their place in producing very high-quality results when required, these results demonstrate that a specialised representation is not *strictly* necessary and can be off-set by an appropriate move-operator.

Reproducibility

The code used for the experiments and for the analysis of the results is available at https://github.com/c-stone2099/HHGE-PPSN2018

References

1. Mohamed Bader-El-Den and Riccardo Poli. Generating sat local-search heuristics using a gp hyper-heuristic framework. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 37–49. Springer, 2007.
2. Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.
3. Mingchang Chih. Self-adaptive check and repair operator-based particle swarm optimization for the multidimensional knapsack problem. *Applied Soft Computing*, 26:378–389, 2015.
4. Peter Cowling, Graham Kendall, and Eric Soubeiga. A hyperheuristic approach to scheduling a sales summit. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 176–190. Springer, 2000.
5. Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. Ponyge2: Grammatical evolution in python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1194–1201. ACM, 2017.
6. Natalio Krasnogor and Steven Gustafson. A study on the use of"self-generation"in memetic algorithms. *Natural Computing*, 3(1):53–76, 2004.
7. Franco Mascia, Manuel López-Ibánez, Jérémie Dubois-Lacoste, and Thomas Stützle. 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*, pages 321–334. Springer, 2013.
8. Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & operations research*, 51:190–199, 2014.
9. Gabriela Ochoa, Matthew Hyde, Tim Curtois, Jose A Vazquez-Rodriguez, James Walker, Michel Gendreau, Graham Kendall, Barry McCollum, Andrew J Parkes, Sanja Petrovic, et al. Hyflex: A benchmark framework for cross-domain heuristic search. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 136–147. Springer, 2012.
10. Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.
11. Ulrich Pferschy and Joachim Schauer. The knapsack problem with conflict graphs. *J. Graph Algorithms Appl.*, 13(2):233–249, 2009.
12. Valentin Robu, DJA Somefun, and Johannes A La Poutré. Modeling complex multi-issue negotiations using utility graphs. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 280–287. ACM, 2005.
13. Nasser R Sabar, Masri Ayob, Graham Kendall, and Rong Qu. Grammatical evolution hyper-heuristic for combinatorial optimization problems. *strategies*, 3:4, 2012.
14. Kevin Sim and Emma Hart. A combined generative and selective hyper-heuristic for the vehicle routing problem. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 1093–1100. ACM, 2016.
15. Christopher Stone, Emma Hart, and Ben Paechter. 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*, pages 578–593. Springer, 2018.