Cost-Benefit Investigation of a Genetic-Programming Hyperheuristic

Robert E. Keller and Riccardo Poli

Department of Computing and Electronic Systems, University of Essex, UK

Abstract. In previous work, we have introduced an effective, grammar-based, linear Genetic-Programming *hyperheuristic*, i.e., a search heuristic on the space of heuristics. Here we further investigate this approach in the context of search performance and resource utilisation. For the chosen realistic travelling salesperson problems it shows that the hyperheuristic routinely produces metaheuristics that find tours whose lengths are highly competitive with the best results from literature, while population size, genotype size, and run time can be kept very moderate.

1 Introduction

A heuristic is a method that, given a problem, often finds a good solution within acceptable time, while it cannot be shown that a found solution cannot be bad, or that the heuristic will always operate reasonably quickly. A metaheuristic is a heuristic that approaches a problem by employing heuristics. The term *hyperheuristic* [21], see [22] for its origin, refers to a heuristic that explores the space of metaheuristics that approach a given problem.

Over the past few years, hyperheuristics (HH) have increasingly attracted research interest. For example, [7] suggests a method of building low-level heuristics for personnel scheduling, [6] proposes tabu search on the space of heuristics, [9] describes a timetabling application of a hyperheuristic, and [8] suggests simulated annealing as learning strategy for a hyperheuristic. [19] employs Genetic Programming (GP) [2, 15, 16] for evolving Evolutionary Algorithms that are applied to problems of discrete optimisation. For the bin-packing problem, [4] introduces a hyperheuristic that is driven by GP. This system successfully reproduces a human-designed bin-packing method.

While the approaches presented in these papers use fixed, problem-specific languages implying sequential execution of actions, our linear GP hyperheuristic, introduced in [13] and further investigated in [14], uses grammars to obtain independence from a given problem domain and to contribute to guiding the search for a solution to a given problem.

In our previous work we saw that the introduction of a looping construct in one of the investigated grammars proved crucial to the effectiveness of the hyperheuristic: it routinely produced metaheuristics that actually delivered best-known solutions to larger TSP benchmark instances despite the simplicity of the underlying grammar. Also the low-level heuristics, given to the hyperheuristic

N. Monmarché et al. (Eds.): EA 2007, LNCS 4926, pp. 13–24, 2008.

as building material, were basic, showing that a user is only required to provide simple heuristics.

The advantage of the approach is that domain knowledge becomes a free resource for the GP hyperheuristic that does not have to rediscover the provided component heuristics. Moreover, by crafting a grammar appropriately, one can direct evolutionary search towards promising types of metaheuristics.

The demonstrated principle and its real-world effectiveness clearly confirmed the original hope behind hyperheuristics that they can lead to optimisation methods that are more flexible in their application to different practical domains. In this context, the domain-independence of the principle is of particular relevance since a fixed HH that efficiently operates for all domains cannot be designed [24]. This obstacle can be circumvented with our GP hyperheuristic because a decision maker can specialise it for a given problem domain by changing the supporting grammar.

As seen in our previous work, the demands on the user of the hyperheuristic are very modest in terms of sophistication of heuristics to be supplied to the HH. In the present paper, we shall investigate the question whether the HH is also easy on its computing resources, in particular in terms of the sizes of populations and genotypes, and whether obtaining a significant increase in search performance does only require a modest additional investment of resources. To the end of experimenting, we use those grammars from our previous work that have shown most beneficial in guiding the hyperheuristic search.

The paper is organised as follows. In Section 2, we introduce the hyperheuristic in detail. In Section 3, we describe the types of problem used in experiments with the hyperheuristic. In Section 4, we describe the grammars that we then use for experiments described in Section 5. In Section 6, we give a summary and conclusions, while in Section 7 we describe interesting avenues for future work.

2 A Linear-GP Hyperheuristic

Our GP hyperheuristic accepts the definition of the structure of desired metaheuristics for D, an arbitrary, fixed domain of problems. Then, in principle, after changing this description appropriately, one can apply the HH to a different domain.

To give the definition, one may represent some of D's low-level heuristics or well-known metaheuristics as components of sentences of a language that one describes by a grammar G. In this manner, $\sigma \in L(G)$ defines a metaheuristic for D. Then, any form of grammar-based GP (e.g., [20] [23] [18] [12] [25] [10]), evolving programs from $L(G)$, is a hyperheuristic for D. Here, we describe our HH implementation that is a flavour of linear GP [3].

A metaheuristic is represented as a genotype $g \in L(G)$ with a domain-specific grammar G. T shall designate the set of terminals of G. $L(G) \subset \mathbf{T}^*$, the set of all strings over T. We call a terminal $t \in T$ a *primitive* (and T a *primitive set*) to avoid confusion regarding "terminal" as used in the field of GP. Primitives

Algorithm 1. GP-based HYPERHEURISTIC

given: grammar G, population size p, length l
repeat
  produce next random primitive-sequence $\sigma : |\sigma| = \mathsf{l}$
  EDITING(σ,G) $\rightarrow$ g `genotype`
until p genotypes created
while time available **do**
  Selection: 2-tournament
  Reproduction: Copy winner g into loser's place $\rightarrow g'$
  Exploration: with a given probability
    Mutate copy $g' \rightarrow \delta$
    EDITING(δ,G)$\rightarrow g''$ `genotype`
end while

may represent manually created metaheuristics, low-level heuristics, or parts of them.

The execution of a metaheuristic, g, with $g = \mathsf{i_0}\, i_1 ... i_n$, $i_j \in T$, means the execution of the i_j. This execution constructs a complete structure, s, that is a candidate solution to the given problem. More specifically, s is obtained from an initial, complete structure, $\mathsf{i_0}()$:

$$s = i_n(...(i_1(\ \mathbf{i_0}()\ ))...).$$

All i_j with $j \neq 0$ accept a complete structure as input. All i_j deliver a complete structure as output. In particular, $\mathsf{i_0}$, in some straightforward fashion, delivers an initial, complete structure.

g's fitness shall depend on the quality of s because the execution of g's primitives builds s in the described manner.

At the beginning of a run of the GP hyperheuristic (s. Algorithm 1), given population size p, *initialisation* produces p random primitive-sequences from T^*. All such sequences are of the same length, l. *Mutation* of a genotype $g \in L(G)$ randomly selects a locus, j, of g, and replaces the primitive at that locus, i_j, with a random primitive, $t \in T, t \neq i_j$.

Naturally, both initialisation and mutation may result in a primitive-sequence, $\sigma = \mathsf{i_0}..i_j..i_k \in T^*$, that is not a valid genotype, i.e., $\sigma \notin L(G) \subset T^*$. In this case, the sequence is passed to an operator called *EDITING* that starts reading σ from left to right.

If EDITING reads a primitive, p, that represents a syntax error in its current locus, EDITING replaces it with the *no-operation primitive*, $\mathbf{n}$. These steps are repeated until the last primitive has been processed. Then, either the current σ is in $L(G)$, and EDITING ends, or still $\sigma \notin L(G)$. In the latter case, EDITING keeps repeating the above steps on σ, but this time processing it from right to left. The result is either a $\sigma \in L(G)$ or a σ that consists of $\mathtt{n}$-instances only. In this latter, unlikely case, EDITING then assigns the lowest available fitness value to σ. This way σ will most likely disappear from the population during *tournament selection*.

Note that, although we initialise the population using sequences of a fixed length, l, the application of EDITING effectively leads to a population containing genotypes of variable lengths not longer than l. This variation in genotype size is beneficial, as, in principle, it allows the evolution of parsimonious heuristics.

We actually observed this effect and described it in [14]. It may contribute to saving run time since a shorter genotype may execute faster. In any case, l, the maximally available genotype size, controls the actual genotype sizes, and we shall investigate its influence on search performance later.

3 Problem Domain

To study aspects of resource use in the context of the performance of the GP hyperheuristic, we select the NP-hard set of travelling salesperson problems (TSP) [17].

In its simplest form, a TSP involves finding a minimum-cost *Hamiltonian cycle*, also known as "*tour*", in a given, complete, weighted graph. Let the n nodes of such a graph be numbered from 0 to $n-1$. Then, one describes a tour involving edges (v_0, v_1), (v_1, v_2), ..., (v_{n-1}, v_0) as a permutation $p = (v_0, ..., v_{n-1})$ over $\{0, ..., n-1\}$.

We call permutation $(0, 1, ..., n-1)$ the *natural* cycle of the graph. The weight of an edge (i, j) represents the cost of travelling between i and j. Here, we shall interpret this cost as the distance between i and j. Thus, the shorter a tour is, the higher is its quality.

4 Grammars

We describe TSP-specific languages that will support experimenting. To that end, we require a few simple routines, including basic heuristics, that are represented as primitives of terminal sets of the describing grammars.

The primitive `NATURAL` designates the method that creates the natural cycle for a problem.

The low-level heuristic `2-CHANGE` identifies a minimal change of a tour H into a different tour: given two of H's edges $(a, b), (c, d) : a \neq d$, $b \neq c$, `2-CHANGE` replaces them with $(a, c), (b, d)$. When the hyperheuristic is about to call a `2-CHANGE` primitive, it randomly selects two appropriate edges, $(a, b), (c, d)$, as arguments for `2-CHANGE`.

Another primitive, `IF_2-CHANGE`, executes `2-CHANGE` only if this will shorten the tour under construction. As every greedy operator, `IF_2-CHANGE` is a boon and a curse, but its introduction is safe here since there is a randomising counterweight in the form of `2-CHANGE`.

Another low-level heuristic is known as a *3-change*: delete three mutually disjoint edges from a given tour, and reconnect the obtained three paths so that a different tour results. Given this method, we define the heuristic `IF_3-CHANGE`:

```
randomly select edges as arguments for 3-change;
if 3-change betters the cycle for the arguments, execute 3-change.
```

```
metaheuristic       ::= NATURAL
                      | NATURAL search

search              ::= heuristic
                      | heuristic search

heuristic           ::= 2-CHANGE
                      | IF_2-CHANGE
                      | IF_3-CHANGE
                      | IF_NO_IMPROVEMENT
```

Fig. 1. Grammar ThreeChange

```
Preamble

heuristic           ::= 2-CHANGE
                      | IF_2-CHANGE
                      | IF_3-CHANGE
```

Fig. 2. Grammar NoNoImprove

Since `IF_3-CHANGE` introduces a further greedy bias if it is used in combination with `IF_2-CHANGE`, it may or may not be helpful to provide a counter-bias, for instance by occasionally allowing for possibly worsening a tour, such as in the heuristic

`IF_NO_IMPROVEMENT`:
`if none of the latest` $1,000$ `individuals produced has found a better best-so-far tour, execute a 2-change`.

Using the defined primitives, we give grammar *ThreeChange* (s. Figure 1). In further grammars, we shall represent the top two grammar rules, i.e., `metaheuristic` and `search`, by the symbol **Preamble**.

A small language is desirable, as it means a small solution space for the hyperheuristic. To understand, by means of a coming experiment, whether `IF_NO_IMPROVEMENT` does or does not improve the effectiveness of the GP hyperheuristic, we remove it from grammar *ThreeChange*. We call the resulting grammar and its language *NoNoImprove* (s. Figure 2).

So far, only sequential and conditional execution of user-provided heuristics are available to evolved metaheuristics. A loop element is required to complete the set of essential control structures. To that end, we introduce the primitive `REPEAT_UNTIL_IMPROVEMENT` p:
`execute primitive` p `until it has lead to a shorter tour or until it has been executed` ι `times for user-given` ι.

An example for the use of `REPEAT_UNTIL_IMPROVEMENT` in a grammar, *DoTillImprove*, is shown in Figure 3.

```
Preamble

heuristic ::= 2-CHANGE
            | loop IF_2-CHANGE
            | loop IF_3-CHANGE

loop      ::= REPEAT_UNTIL_IMPROVEMENT
            | /* empty */
```

Fig. 3. Grammar DoTillImprove

5 Experiments

5.1 Setup

We number the loci of genotypes, beginning with zero. For the present setup of the hyperheuristic, its random choice of an element from a set shall be uniform. For all experiments, mutation probability (cf. Algorithm 1, step 9) shall be 0.5. The GP-HH shall measure time in terms of the number of offspring individuals produced after creation of the initial individuals (cf. Algorithm 1, step 6).

5.2 First Problem

We consider problem `eil51` from [1]. Its dimension is $n = 51$ nodes, its best known solution has a length of 428.87 [11] with natural length of approximately 1,313.47. For a symmetrical TSP instance, the number of tours that are different in human terms equals $(n-1)!/2$.

The evolved metaheuristics operate on permutations of n nodes, so that the size of their search space is $n!$. $n = 51$ gives about 1.6×10^{66} search points and 1.52×10^{64} different tours. Table 1 reports a subset of results from [14] obtained with grammars *ThreeChange* and *NoNoImprove*, the latter lacking the primitive `IF_NO_IMPROVEMENT`.

Here, we comment on an effect that was not in the focus of our previous work: surprisingly, eliminating the non-destructive primitive `IF_NO_IMPROVEMENT` from grammar *ThreeChange* yields better search performance (s. bottom row of table).

A possible explanation of this phenomenon is the much smaller size of the resulting language which is the solution space of the hyperheuristic. This smaller size may at least partially compensate for the loss of `IF_NO_IMPROVEMENT`. For each of both grammars, given its primitive set T and genotype length l, the size of the induced solution space equals

$$|L(G)| = \sum_{i=0}^{l-1} (|T| - 1)^i, \tag{1}$$

since the grammar generates language

Table 1. Performance for eil51 over *ThreeChange* and *NoNoImprove*, 30 runs. Basic parameters: pop.size 100, genotype size 500, offspring 1×10^5, mut. prob. 0.5. "Best" mentions the best value over all runs.

eil51	**Mean best**	SD	Best
Natural length	1,313.47	n.a.	n.a.
ThreeChange	874.96	26.55	810.73
NoNoImprove	798.32	15.98	763.30

$$L(G) = \bigcup_{i=0}^{l-1} \{\mathsf{NATURAL}\} \otimes (T \setminus \{\mathsf{NATURAL}\})^{\otimes i},$$

where $\otimes$ is the Cartesian product and the superscript $^{\otimes i}$ represents the Cartesian product iterated i times. The largest term of the finite geometric series (1) is $(|T|-1)^{l-1}$. So, for grammar $G = \mathit{ThreeChange}$, where $|T| = 5$, and for genotype length $l = 500$, this term equals $4^{499} \approx 2.7\times 10^{300}$. For $G = \mathit{NoNoImprove}$, we obtain merely $3^{499} \approx\ 1.2\times 10^{238}$.

Therefore, in terms of enhancing effectiveness and efficiency of the GP hyperheuristic, it may well be recommendable in general to approach a problem first with a small primitive set for the underlying grammar. This is because every language increases in size, often exponentially, when one adds an element to its primitive set. Should the problem at hand resist solution, one can still incrementally add beneficial primitives.

5.3 Second Problem

While the metaheuristics' search space for eil51 already has a realistic size, next, we consider eil76 [1], a 76-node problem with a size of about 1.9×10^{111} search points.

We use the same basic parameters as given in Table 1, and, from here, grammar *DoTillImprove* with parameter ι for the loop primitive. The best known result from literature [11] is $\alpha = 544.37$, obtained by a highly specialised, manually designed hybrid Genetic Algorithm.

An individual of the GP hyperheuristic that locates a tour whose length is at least as good as α shall be called a *top metaheuristic*.

Table 2 shows results regarding our GP hyperheuristic. The mean best over all runs is well within one percent of α. Our evolved top metaheuristics yield tour lengths that are actually shorter than $\alpha = 544.37$. Unfortunately, [11] does not specify whether α is a rounded value. Thus, we report that our GP hyperheuristic has found an overall best tour length of $\alpha_{HH} = 544.36908$.

In any case, since the hyperheuristic at least finds α, the used parameters are a good starting point for further experiments.

Population size. We ask how the performance of the GP hyperheuristic depends on the population size. Therefore, we vary the population size over several orders of magnitude for different experiments.

Table 2. Performance of metaheuristics evolved over language DTI, on problem eil76. 100 runs of GP hyperheuristic. Basic parameters: pop.size 100, genotype size 500, offspring 1×10^6; mut. prob. 0.5. Evolved top metaheuristics at least match effectiveness of hand-crafted Hybrid GA. **P.%**: Mean best or natural length in terms of % of best known result α. All real values rounded off to nearest hundredth.

eil76	Mean best	S.D.	Best	ι	P.%
Nat. length	1,974.71	n.a.	n.a.	n.a.	262.75
DTI	548.99	1.67	**544.37**	2×10^3	0.85
Hybrid GA	n.a.	n.a.	***544.37*** *best known*		n.a.

Table 3. Performance of metaheuristics evolved over language DTI, on problem eil76. 100 runs of GP hyperheuristic for each given population size. Other basic parameters: genotype size 500, offspring 1×10^6, ι 2,000; mut. prob. 0.5. Bottom row gives best known rounded result as found by GP hyperheuristic and Hybrid GA. Over all runs, column *First* gives the mean of the serial number of the first metaheuristic of a run that finds a shortest tour of the run, rounded off to the nearest 1,000, given in the unit of 1,000 individuals.

eil76	Mean best	S.D.	Best	**Pop.size**	P.%	First[k]
Nat. length	1,974.71	n.a.	n.a.	n.a.	262.75	n.a.
	677.53	13.76	614.55	10	24.46	**476**
	548.99	**1.67**	**544.37**	*100*	0.85	627
DTI	**547.48**	1.69	**544.37**	500	**0.57**	845
	552.86	2.55	546.74	1,000	1.56	905
	722.89	29.03	651.82	10,000	32.79	955
GPHH/HGA	n.a.	n.a.	***544.37*** *best known*		n.a.	n.a.

Table 3 shows the results. Starting at p=10,000, smaller population sizes yield better results, up to a point: the drop to p=10 clearly worsens the effectiveness of the GP hyperheuristic. This can be explained by the enhancement of tournament-selection pressure that comes with a smaller population size, which prematurely stalls progress when the pressure becomes too high too early during a run.

We are interested in the *efficiency* of a run of the GP hyperheuristic in terms of the number of individuals it produces before it locates the first of its best individuals.

Table 3 gives these values in its "First" column. While p=10 yields the highest efficiency, the resulting effectiveness (column "Best") is poor. However, p=100 gives best effectiveness (column "Best"), best reliability ("S.D."), second best overall effectiveness ("Mean best"), and second best efficiency ("First"). Thus, an investment in a larger population size is of secondary interest only, as it, at best, yields a marginal improvement in the overall effectiveness, without yielding a better metaheuristic.

Genotype size. Next, we ask for the connection of efficiency and genotype size in the context of effectiveness. To that end, we fix p=100 as it has given best effectiveness, and we shall vary the genotype size, l. For the chosen p-value, Table 3 suggests setting the number of offspring to be produced to 627,000.

Table 4. Runs of GP hyperheuristic for given genotype sizes. Other parameters: population size 100, offspring 627,000; ι 2,000; mut. prob. 0.5. Over all runs done for a given genotype size l, column *Firstbest* gives the mean of the serial number of the top metaheuristic discovered first, if any, else "—"; unit: 1,000 individuals. Column *Runs* gives the number of runs performed for given l. Other details as given in caption of Table 3.

eil76	Mean best	S.D.	Best	**Geno.size**	P.%	Firstbest[k]	Runs
Nat. length	1,974.71	n.a.	n.a.	n.a.	262.75	n.a.	n.a.
	1,314.000	17.55	1,263.400	50	141.38	—	100
	713.601	14.89	677.980	250	31.09	—	100
	642.870	12.4	613.129	300	18.11	—	100
	568.134	4.57	556.451	400	4.37	—	100
DTI	556.043	3.46	545.667	450	2.14	—	55
	548.990	1.67	**544.369**	500	0.85	621	100
	546.531	1.1	**544.369**	600	0.4	494	16
	544.765	0.79	**544.369**	700	0.07	463	10
	544.369	**0.0**	**544.369**	900	0.**0**	**255**	15
GPHH/HGA	n.a.	n.a.	***544.370*** *best known*		n.a.	n.a.	n.a.

Table 4 collects the results. Already for modest values of l, such as 400, the GP hyperheuristic produces competitive metaheuristics. Higher values, still in the same order of magnitude, increase all aspects of performance, such as efficiency ("First best") and effectiveness. l=900 even guarantees the best known result for every observed run. Thus, clearly, if one considers making an additional investment of memory, it should be spent on the genotype size.

Iteration number. Finally, we are interested in the question whether, for p=100, l=500, 1×10^6 offspring, and mutation probability 0.5 (cf. Table 3), at a reasonable expense of more run time per metaheuristic, the hyperheuristic can clearly improve its overall effectiveness. For $\iota = 2,000$, on average, the hyperheuristic produces 97 metaheuristics per second. To approach our question, we run the HH for $\iota = 15,000$.

We find that the mean best's P.% value drops to 0.26, less than a third of its previous value, 0.85. On average, the GP hyperheuristic still produces 56 metaheuristics per second. Thus, the run-time increase by factor $1/56/(1/97) \approx 1.7$ has more than tripled the overall effectiveness of the hyperheuristic. Also, while we consider only 30 runs, a very low standard deviation (0.97) indicates reliable search behaviour.

6 Summary and Conclusions

We have investigated our domain-independent, linear GP hyperheuristic (HH) [14, 13] with respect to its demand for computing resources in the context of its effectiveness and efficiency. The HH produces metaheuristics from a user-given language, employing provided heuristics. We experimented on this approach, using the domain of travelling-salesperson problems. To this end we provided the

hyperheuristic with elementary heuristics for this domain and with a progression of simple grammars.

On the used, realistic benchmark problems, it shows that the GP hyperheuristic shows excellent competitiveness, yielding best known tour lengths usually only produced by specialised, sophisticated, man-made solvers initialised with selected tours as good starting points.

We observed that one can increase efficiency and effectiveness of the hyperheuristic by making only modest additional investments of population size, genotype size, and production time of an evolved metaheuristic. Favourable scalability may well be a common property of GP-based hyperheuristics over different problem domains, since [5] reports related results for a different challenge.

Also, regarding our GP hyperheuristic, we noted that decreasing the size of the primitive set of the underlying grammar may help solve a problem. We argued that this is at least due to the resulting, exponentially smaller solution space facing the hyperheuristic. It may thus be beneficial to start out with a small primitive set, before incrementally adding primitives if solution quality stays unacceptable.

In our experiments with the TSP domain, it was important to approach a problem with a medium-sized population when using tournament selection, as neither small nor high selection pressure appeared beneficial. It is also useful, if run time is acceptable and memory available, to rather increase the genotype size instead of the population size. These two approaches may also work well for problems other than TSP, but further research is needed to confirm this.

We conclude that, in addition to asking for only little domain knowledge of its user, the GP hyperheuristic, while being competitive, is also undemanding in terms of computing resources.

7 Further Work

In the future we intend to test the presented GP hyperheuristic on further real-world problems from several domains, again comparing the produced metaheuristics to domain-specific algorithms.

Also, it would be interesting to explore what happens if one breaks up low-level heuristics into their components and represents them as primitives. In this way, in principle, the hyperheuristic would be able to produce even more novel and powerful metaheuristics.

Furthermore, on the level of search guidance, we intend to have the GP hyperheuristic collect and use information on the topology of the search space of an underlying problem.

Acknowledgements

We would like to acknowledge financial support from EPSRC (grants EP/C523377/1 and EP/C523385/1) and to thank the anonymous reviewers for helpful comments.

References

[1] http://www.iwr.uni-heidelberg.de/groups/comopt/software/tsplib95/tsp/

[2] Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: On the Automatic Evolution of Computer Programs and its Applications. In: Genetic Programming – An Introduction, Morgan Kaufmann, San Francisco, CA, USA (1998)

[3] Brameier, M., Banzhaf, W.: Linear Genetic Programming. In: vol. 1, Genetic and Evolutionary Computation, Springer, Heidelberg (2006)

[4] Burke, E., Hyde, M., Kendall, G.: Evolving bin packing heuristics with genetic programming. In: Runarsson, T.P., Beyer, H.-G., Burke, E.K., Merelo-Guervós, J.J., Whitley, L.D., Yao, X. (eds.) PPSN 2006. LNCS, vol. 4193, pp. 860–869. Springer, Heidelberg (2006)

[5] Burke, E., Hyde, M., Kendall, G., Woodward, J.: Scalability of evolved on line bin packing heuristics. In: Proceedings of Congress on Evolutionary Computation (CEC 2007) (2007)

[6] Burke, E., Kendall, G., Soubeiga, E.: A tabu-search hyperheuristic for timetabling and rostering. Journal of Heuristics 9(6), 451–470 (2003)

[7] Chakhlevitch, K., Cowling, P.: Choosing the fittest subset of low level heuristics in a hyperheuristic framework. In: Raidl, G.R., Gottlieb, J. (eds.) EvoCOP 2005. LNCS, vol. 3448, pp. 23–33. Springer, Heidelberg (2005)

[8] Dowsland, K., Soubeiga, E., Burke, E.: A simulated annealing hyper-heuristic for determining shipper sizes. European Journal of Operational Research 179(3), 759–774 (2007)

[9] Gaw, A., Rattadilok, P., Kwan, R.: Distributed choice function hyper-heuristics for timetabling and scheduling. In: Burke, E.K., Trick, M.A. (eds.) PATAT 2004. LNCS, vol. 3616, pp. 495–497. Springer, Heidelberg (2005)

[10] Janikow, C.Z.: Constrained genetic programming. In: Hussain, T.S. (ed.) Advanced Grammar Techniques Within Genetic Programming and Evolutionary Computation, Orlando, Florida, USA, 13 July 1999, pp. 80–82 (1999)

[11] Jayalakshmi, G., Sathiamoorthy, S., Rajaram, R.: An hybrid genetic algorithm — a new approach to solve traveling salesman problem. International Journal of Computational Engineering Science 2(2), 339–355 (2001)

[12] Keller, R.E., Banzhaf, W.: The evolution of genetic code on a hard problem. In: Spector, L., Langdon, W.B., Wu, A., Voigt, H.-M., Gen, M. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), San Francisco, CA, July 7–11, 2001. pp. 50–66, Morgan Kaufmann, San Francisco, CA (2001)

[13] Keller, R.E., Poli, R.: Linear genetic programming of metaheuristics. In: GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation, July 7-11, 2007, ACM Press, London (2007)

[14] Keller, R.E., Poli, R.: Linear genetic programming of parsimonious metaheuristics. In: Proceedings of Congress on Evolutionary Computation (CEC 2007), Swissotel The Stamford, Singapore September 25-28 (2007)

[15] Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)

[16] Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer, Heidelberg (2002)

[17] Lawler, E., Lenstra, J., Kan, A.R., Shmoys, D. (eds.): The Travelling Salesman Problem. Wiley, Chichester (1985)

[18] Montana, D.J.: Strongly typed genetic programming. Evolutionary Computation 3(2), 199–230 (1995)
[19] Oltean, M.: Evolving evolutionary algorithms using linear genetic programming. Evolutionary Computation 13(3), 387–410 (Fall 2005)
[20] O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language. Genetic programming, vol. 4. Kluwer Academic Publishers, Dordrecht (2003)
[21] Ross, P.: Hyperheuristics. In: Burke, E., Kendall, G. (eds.) Search Methodologies, pp. 529–556. Springer, New York, Berlin (2005)
[22] Soubeiga, E.: Development and application of hyper-heuristics to personnel scheduling. PhD thesis, Computer Science, University of Nottingham (2003)
[23] Whigham, P.A.: Search bias, language bias, and genetic programming. In: Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L. (eds.) Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, July 28–31, 1996, pp. 230–237, MIT Press, Cambridge (1996)
[24] Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation 1(1), 67–82 (1997)
[25] Wong, M.L., Leung, K.S.: Applying logic grammars to induce sub-functions in genetic programming. In: 1995 IEEE Conference on Evolutionary Computation, Perth, Australia, November 29 - 1 December, 1995, vol. 2, pp. 737–740. IEEE Press, Los Alamitos (1995)