Multi-Task Code Reuse in Genetic Programming

Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch
Institute of Computing Science, Poznan University of Technology
Piotrowo 2, 60965 Poznań, Poland
{wjaskowski,kkrawiec,bwieloch}@cs.put.poznan.pl

ABSTRACT

We propose a method of knowledge reuse between evolutionary processes that solve different optimization tasks. We define the method in the framework of tree-based genetic programming (GP) and implement it as *code reuse* between GP trees that evolve in parallel in separate populations delegated to particular tasks. The technical means of code reuse is a *crossbreeding* operator which works very similar to standard tree-swapping crossover. We consider two variants of this operator, which differ in the way they handle the incompatibility of terminals between the considered problems. In the experimental part we demonstrate that such code reuse is usually beneficial and leads to success rate improvements when solving the common boolean benchmarks.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods

General Terms

Algorithms

Keywords

Genetic Programming, Code Reuse, Multi-Task Learning

1. INTRODUCTION

An important lesson we learn from Wolpert's 'No free lunch' theorem [24] is that the space of all possible problems (optimization tasks) is vast, even if we fix the number of variables involved in problem definition. An indirect conclusion from that all-important study is that the heuristic search algorithms usually perform better than the random search not only thanks to designer's wit, but also due to the fact that the tasks they usually face are located in a tiny fragment of that space. Thus, such tasks (often identified with real-world tasks) must exhibit some form of similarity. More formally, their mutual similarity has to be on average greater than the mutual similarity of a pair of problems drawn randomly from the space of *all* possible problems.

Particular search algorithms exploit that commonality of problems to a different extent. However, they usually do it only implicitly, by defining some *features* (conditions) a particular problem must possess to be solvable within a given framework. Typical examples of such features are the assumptions that the objective function is differentiable, or that all good solutions are close to each other (a.k.a. global convexity). Addressing and exploiting the issue of mutual problem similarity in a more direct way, by an *explicit* reuse of knowledge between the algorithms that solve different tasks, is much more seldom.

As a consequence, current research is focused on algorithms which 'lifecycle' is limited to solving a single task. Within evolutionary computation framework, such a task typically boils down to an evolutionary run on a particular problem and takes place in perfect isolation from the other experiments. The algorithm is not enabled to refer to the knowledge pertaining to other problems. Because of that, knowledge reuse is still listed among the most challenging issues in machine learning [17].

It has been hypothesized that explicit reuse of knowledge may bring us many benefits, including faster search convergence and lower risk of overfitting in problems that involve dividing data into training set and testing set. This applies in particular to machine learning, where the multi-task learning paradigm has a relatively long history [2]. Following the positive accounts from that domain, we propose here a simple yet effective approach to knowledge reuse for tree-based genetic programming [10]. In our variant, knowledge reuse boils down to *code reuse* that operates between the evolutionary runs that solve different problems in parallel. The technical means for that is a *crossbreeding operator* that crosses over individuals from populations evolving for different problems. As population usually comprises a specific species, we think the term *crossbreeding* reflects well the character of this operator by analogy to its biological meaning—the act of mixing the different species or varieties of animals or plants.

The following Section 2 reviews the related work and motivations for knowledge reuse. Section 3 is the core part of this paper and presents the proposed method of code reuse within the genetic programming paradigm. Section 4 describes the computational experiment, and Section 5 draws conclusions and outlines the future research directions.

2. MOTIVATIONS AND RELATED WORK

In [11], Koza made the key point that the reuse of code is a critical ingredient to scalable automatic programming. In the context of genetic programming, code reuse is often connected with knowledge encapsulation. The canonical result

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.
ACM 978-1-60558-131-6/08/07.

in this field are Automatically Defined Functions, defined by Koza in [10, section 6.5.4]. Since then, more research on encapsulation [20] and code reuse [12] has been done within the genetic programming community. Proposed approaches include reuse of assemblies of parts within the same individual [6], identifying and re-using code fragments based on the frequency of occurrences in the population [7], or explicit expert-driven task decomposition using layered learning [1, 8]. Among other prominent approaches, Rosca and Ballard [21] utilized the genetic code in form of evolved subroutines, Haynes [5] integrated a distributed search of genetic programming-based systems with collective memory, and Galvan Lopez *et al.* [3] reused code using a special encapsulation terminal.

In the research cited above, the code was reused only within a *single* evolutionary run. Surprisingly little work has been done in genetic programming to reuse the code between *multiple* tasks. To our knowledge, the first to notice this gap was Seront [22], who investigated code reuse by initializing an evolutionary run with individuals from the concept library consisting of solutions taken from other, similar tasks. He also mentioned the possibility of introducing a special mutation operator that would replace some subtrees in population by subtrees taken from the concept library, in a way similar to our contribution, but did not formalized nor computationally verified it. An example of other approach to reusing the knowledge between different tasks is Kurashige's work on gait generation of six-legged robot [13], where the evolved motion control code is treated as a primitive node in other motion learning task.

In machine learning, the research on issues related to knowledge reuse, i.e., meta-learning, knowledge transfer, and lifelong learning, seem to attract more attention than in evolutionary computation (see [23] for a survey). Among these, the closest machine learning counterpart of the approach presented in this paper is *multitask learning*, meant as simultaneous or sequential solving of a group of learning tasks. Following [2] and [4], we may name several potential advantages of multitask learning: improved generalization, reduced training time, intelligibility of the acquired knowledge, accelerated convergence of the learning process, and reduction of the number of examples required to learn the concept(s). The ability of multitask learning to fulfill some of these expectations was demonstrated, mostly experimentally, in different machine learning scenarios, most of which used artificial neural networks as the underlying learning paradigm [19, 18].

In the field of genetic algorithms, the work done by Louis *et al.* resembles our contribution the most. In their Case Injected Genetic Algorithms (CIGAR) described in [14], the experience is stored in a form of solutions to problems solved earlier ('cases'). When confronted with a new problem, CIGAR evolves a new population of individuals and periodically enriches it with such remembered cases. The experiments demonstrated CIGAR's superiority to genetic algorithm in terms of search convergence. However, CIGAR injects *complete* solutions only and requires the 'donating' task to be finished before starting the 'receiving' task, which makes it significantly different from our approach.

In [9] we proposed a similar method of code reuse to this presented in this paper and evaluated it on a set of complicated pattern recognition problems. The method introduced in this paper has the advantage over the previous one of being parameterless. Also, here we prove the concept of reusing the code on pairs of simple digital logic design tasks which are typical genetic programming benchmarks.

3. CODE REUSE

Genetic programming is well suited to take advantage from code reuse between evolutionary runs solving different problems. The tree-like structure of individuals allows to isolate code fragments in a form of subtrees that can be treated as partial solutions to the problem and/or solutions of some subproblems. The subtrees can then be used as partial solutions for other problems. This property of tree-based genetic programming is exploited by the method proposed in this paper.

Intuitively, code reuse is likely to work for problems that exhibit some *similarity*. However, it is not clear what problem similarity exactly means and how to measure it basing only on the problems definitions. It was argued that there is correlation between problem similarity and solution similarity [14]. Thus, one method to measure the similarity of two problems is to measure the similarity of their solutions, but then the knowledge of similarity is not useful anymore, since we already have the solutions. Other method to assess the similarity between two problems is to somehow compare their search spaces, but, again, it is easier to solve the problem on its own. Therefore, we think that the only reasonable way to assess the similarity between problems is to use one's intuition and domain knowledge or a trial-and-error method.

Although we never know if the code reuse between two particular tasks will work, for some pairs of problems we can definitely say it will not work. The prerequisite for reuse code is that the problems' search spaces have to overlap. For genetic programming, that implies intersection of primitive sets; ideally they should be the same.

Primitives in genetic programming are typically subdivided into functions and terminals. Within this study we assume that, to participate in and benefit from knowledge reuse, the two problems of interest have to use the same function set. Fortunately, this condition is relatively easy to fulfill as functions usually embody general knowledge and are quite often shared between problems. Terminals, on the contrary, represent function arguments and constants and are problem-dependent to much higher degree than functions. Let us, for example, consider two digital circuit design problems: 6-bit multiplexer problem and 6-bit even-parity. In both problems the number of terminals is the same, but their meaning is different, thus using a subtree from a 6-bit multiplexer solution in a 6-bit even-parity problem should involve changing the meaning of terminals. Imposing the requirement that the set of terminals in both problems should be the same would seriously limit the applicability of code reuse. We also claim that it is unnecessary, as the terminals (function arguments) may be easily substituted without affecting the functionality of the subtree (function body).

In order to make code reuse possible, we introduced a new genetic operator named *crossbreeding*. Crossbreeding swaps subtrees from two individuals coming from evolutionary runs that solve two problems A and B. From the internal perspective of an evolutionary run that solves A, crossbreeding is a mutation (macromutation) operator, as it replaces a subtree in an individual in A by another subtree; the only difference is that the 'imported' genetic material comes from a randomly selected individual from the run that solves B.

Technically, however, crossbreeding is more similar to the tree-swapping crossover because it involves two individuals. The important difference between crossbreeding and standard genetic programming crossover operators is that crossbreeding must swap also some terminals if the terminal sets in the two problems are different.

The terminal substitution in crossbreeding proceeds in the following way. Let us denote the terminals in problem A as $t_1...t_n$ and terminals in problem B as $s_1...s_m$. In general, we cannot assume any semantic (meaningful) correspondence of terminals between these two problems. Therefore when trying to put a subtree from individual in A to individual in B, each terminal t_i must be replaced by some terminal s_j. Our crossbreeding operator performs this replacement change randomly.

In this work, we provide a proof of concept that evolving solutions for two different problems in parallel using the crossbreeding operator that exchanges the genetic material between them may lead to better results than evolving a solution for a single problem. The exchange of genetic material between two evolutionary runs via crossbreeding takes place in every generation, thus the evolutionary runs are synchronized and work in parallel. This setup has the advantage of having only one parameter (the probability with which crossbreeding is engaged), however, other setups are also possible (see e.g. [9]).

4. THE EXPERIMENT

We tested the code reuse on three classes of boolean problems: even-parity, comparator and num-ones. In the k-bit even-parity problem (*even-k*) the goal is to test if the number of 1's in k input bits is even. A solution of the comparator problem (*cmp-k*) returns true if an integer value coded on $k/2$ more significant bits is less than the second integer coded on the other less significant bits. In the third problem called num-ones (*m-ones-k*) the aim is to return true if there are exactly m 1's in the k input bits (regardless of their positions). We run our experiments on nine tasks in total: 3, 4, and 5-bit variants of the even-parity problem (even-3, even-4, and even-5), 4 and 6-bit comparator (cmp-4 and cmp-6), which compare 2 or 3-bits integers respectively, and finally 3, 4, and 5-bit 2-ones (2-ones-3, 2-ones-4, and 3-ones-5), and one 6-bit 3-ones (3-ones-6).

To solve all the above problems we use genetic programming with the same four logical functions: *And*, *Or*, *Nand* and *Nor*. The terminals correspond to the consecutive input bits and their number depends on the dimensionality of the problem.

The control experiments use standard Koza-I-like GP [10] and start from the initial population created using ramped half-and-half operator with ramp from 2 to 6. The population contains 500 individuals and evolves for 200 generations using crossover with probability 0.9 and mutation (substituting subtree with a new random one) with probability 0.1. We use tournament selection of size 7 and lexicographic parsimony pressure [16] that promotes smaller trees if there is no difference in their fitnesses.

The main experiments examining the influence of code reuse are a bit more complicated. In this case, for each pair of problems we perform a separate experiment, evolving simultaneously two populations, each of them containing 500 individuals and dedicated to solving one problem from the pair. The only difference in setup with respect to the control experiment is that we replace mutation with crossbreeding (still with probability 10%). The crossbreeding operator produces new individuals for the n-th generation in the *target* population using subtrees taken from individuals from the previous $((n-1)$th) generation of the *source* population. This requires evolving both problems in parallel.

The crossbreeding operator exchanging genetic material between the two tasks has to cope with the possibility of different sets of terminals used in these tasks (in all nine problems the function set is the same). We tested two strategies that handle this problem:

- relabeling only terminals which not exist in the target problem (*relabeling when needed*),
- relabeling randomly all terminals in the transferred subtree (*forced relabeling*).

In this process we do not substitute terminals independently. By relabeling we mean changing *all* original terminals corresponding to the *same* input bit i to a terminal corresponding to a randomly chosen j-th input bit. In other words, all instances of the same terminal from the source problem are replaced by the instances of *one* terminal from the target problem. In this way we preserve the semantic relations between the inputs of the transferred subtree.

Our experiments were implemented in the ECJ framework [15]. The table below summarizes the basic parameters of evolution of a single task:

Parameter name	Control experiment	Code reuse
# generations	200	200
population size	500	500
max tree depth	17	17
tournament size	7	7
crossover rate	90%	90%
mutation rate	10%	-
crossbreeding rate	-	10%

Each experiment was repeated 300 times for different random generator's seeds to provide statistical significance. In Tables 1 and 2 we present the success rate for each task evolved independently (the control experiment, the leftmost table column) and with the other problem in a pair. For instance, the value 0.68 in Table 1 at the intersection of row '2-ones-4' and column '2-ones-5' is the success rate of the former problem when reusing code from the latter problem.

Figures 1 and 2 visualize the gain in success rate for the runs that used code reuse compared to runs without code reuse. The area of circle is proportional to the difference between the success rates of the code reuse experiment and the control experiment. A filled circle means gain, a hollowed one means loss. As we can see, when terminals are always relabeled (Fig. 2), the gain for *m-ones-k* and *cmp-k* problems is in most cases profitable (except the very easy and the very hard problems). Unfortunately, at the same time the problems from the *even-k* family lose the most. This effect is less prominent when terminals are relabeled only when it is necessary (Fig. 1); in that case, code reuse is never statistically worse on the *even-k* problems.

It seems therefore that the 'aggressive' forced relabeling may lead to greater gains of success rate. On the other hand, it is also more risky, as for some problems (*even-k*)

Task	-	2-ones-3	2-ones-4	2-ones-5	3-ones-6	cmp-4	cmp-6	even-3	even-4	even-5	Mean
2-ones-3	0.9900	-	>1.0000	0.9967	0.9833	0.9933	0.9933	>1.0000	0.9967	>1.0000	0.9954
2-ones-4	0.4533	0.4700	-	>0.6800	0.5000	0.4933	0.4800	>0.5233	>0.7800	>0.5733	>0.5625
2-ones-5	0.1033	0.1033	0.1233	-	0.1233	0.1200	0.1100	0.1367	>0.1533	>0.2067	>0.1346
3-ones-6	0.0000	0.0000	0.0000	0.0033	-	0.0000	0.0000	0.0000	>0.0133	0.0000	>0.0021
cmp-4	0.7000	0.7500	0.7433	0.7133	0.7333	-	>0.7633	0.7500	>0.7900	>0.7600	>0.7504
cmp-6	0.3067	<0.2233	0.3067	0.3033	0.2900	>0.4000	-	<0.1933	0.3500	0.3333	0.3000
even-3	0.9900	0.9900	0.9967	0.9967	0.9867	0.9900	0.9967	-	>1.0000	>1.0000	0.9946
even-4	0.7733	0.7633	>0.8400	0.7733	0.7200	0.7433	0.7367	0.7400	-	>0.8667	0.7729
even-5	0.1467	>0.2000	0.1533	0.1900	0.1200	0.1233	0.1333	0.1633	0.1567	-	0.1550

Table 1: The success rates when relabeling terminals only when needed (target tasks in rows, source tasks in columns). '>' and '<' mark statistical significance (t-test, 0.05).

Task	-	2-ones-3	2-ones-4	2-ones-5	3-ones-6	cmp-4	cmp-6	even-3	even-4	even-5	Mean
2-ones-3	0.9900	-	>1.0000	0.9967	0.9867	0.9900	0.9967	>1.0000	>1.0000	0.9900	0.9950
2-ones-4	0.4533	>0.5233	-	>0.7000	>0.5333	>0.5333	0.4967	0.5067	>0.7767	>0.6200	>0.5863
2-ones-5	0.1033	0.1367	0.1367	-	0.1267	0.1233	0.1400	0.1233	0.1433	>0.2367	>0.1458
3-ones-6	0.0000	0.0000	0.0033	0.0033	-	0.0000	0.0000	0.0000	0.0000	0.0033	>0.0012
cmp-4	0.7000	0.7433	0.7367	>0.7933	>0.7733	-	>0.7767	>0.7933	>0.7967	>0.7900	>0.7754
cmp-6	0.3067	>0.4133	>0.4400	>0.4333	>0.4100	>0.4600	-	>0.4067	>0.3967	>0.4667	>0.4283
even-3	0.9900	0.9933	0.9900	0.9900	0.9767	0.9867	0.9900	-	0.9933	0.9867	0.9883
even-4	0.7733	0.7167	0.7967	0.7367	<0.6100	<0.6800	<0.6800	0.7367	-	0.8233	<0.7225
even-5	0.1467	<0.0633	0.1033	<0.0833	<0.0700	<0.0767	<0.0600	0.1167	0.1500	-	<0.0904

Table 2: The success rates with terminals always relabeled (target tasks in rows, source tasks in columns). '>' and '<' mark statistical significance (t-test, 0.05).

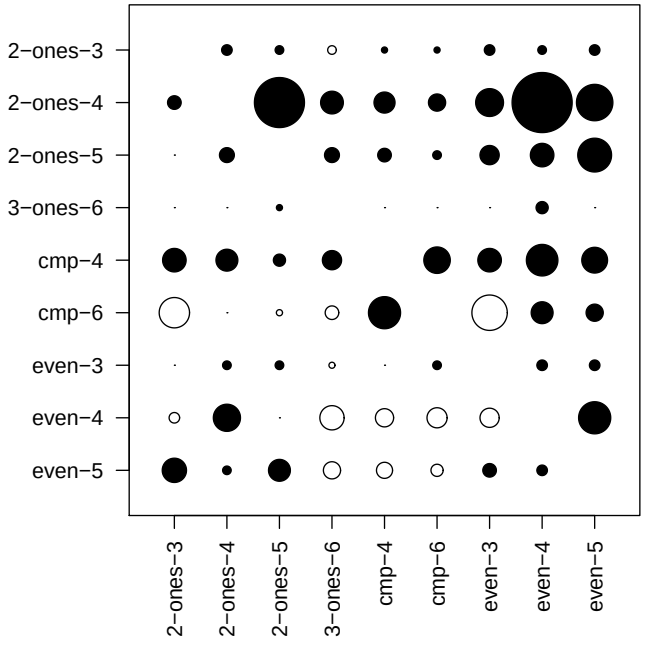

Figure 1: The gain in success rate resulting from code reuse (terminals relabeled only when needed)—target tasks in rows, source tasks in columns. Filled and empty circles indicate gains and losses, respectively.

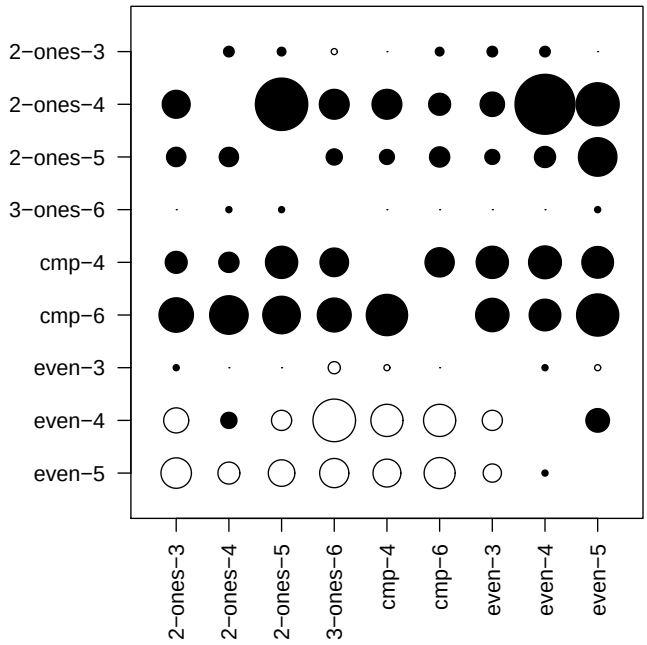

Figure 2: The gain in success rate resulting from code reuse (terminals always relabeled)—target tasks in rows, source tasks in columns. Filled and empty circles indicate gains and losses, respectively.

it never brings significant profit and sometimes leads to significant deterioration. On the contrary, the more moderate technique of relabeling the terminals when needed produces less spectacular gains but fails only twice: for the task pair (cmp-6, 2-ones-3) and for the task pair (cmp-6, even-3) (cf. Table 1). The overall rate of failure is therefore 2 in 72 (the total number of task pairs), i.e., less than 3%. In other words, assuming our pool of problems is representative enough, an experimenter's risk of losing due to the use of our code reuse method is less than 1 in 30. The analogously defined rate of gain amounts to 19 in 72, over 26%. For the forced relabeling, these figures amount to 11% and 33%, respectively.

These estimates are however only partially valid, as it may be easily seen that entire families of target tasks are in general less likely to benefit from code reuse, no matter what the source task is. It seems for instance that *any* code imported into an *even-k* task is detrimental for it. This may be due to the fact that in our setup the crossbreeding operator *replaces* the mutation operator. We hypothesize therefore that crossbreeding does not provide for enough genotypic variation. However, more evidence is needed to verify this supposition.

5. CONCLUSIONS

In this paper, we presented and verified experimentally a straightforward method of code reuse between different simultaneously solved tasks. The results obtained by solving nine different problems using this technique are encouraging and thought-provoking.

The major conclusion is that the crossbreeding operator as a tool of knowledge reuse has usually positive impact on the performance of the evolutionary runs delegated to involved problems. The 'aggresive' forced relabeling of terminals increases the chance of outperforming the standard approach without code reuse, but unfortunately involves significant risk of failure. The technique of relabeling when needed is more 'reserved', offering slightly smaller chance of success and almost negligible risk of failure. In both cases however, the probability of success rate improvement is much higher than the probability of success rate deterioration.

Though this study has preliminary character, the promising results prompt us to examine also other possible settings and scenarios of applying code reuse in GP. One of such settings, mentioned in the previous section, would involve using a mutation operator together with crossbreeding. This would diversify the population and intensify the exploration of the search space, which seems to be not intense enough for some problems (this is probably the reason why code reuse fails on the *even-k* problems). Another possibility is to test the scenario when the source population contains well-developed solutions, instead of transferring the genetic code from individuals which are still evolving and may have poor quality. Finally, the method may be elegantly extended to more than two tasks that participate in the code reuse process in parallel.

In a more general perspective, we demonstrated that the paradigm of genetic programming, thanks to symbolic representation of solutions and the ability of abstraction from a specific context, offers an excellent platform for knowledge transfer. Furthermore, analogously to evolving evolvability, this could lead to the concept of *evolving reusability*, where the objective for the evolutionary process would be to evolve individuals' encoding that promotes code reusability.

We think that utilization of knowledge reuse will be the key to solving the really sophisticated and challenging problems in the future. The research areas of multi-task problem solving and knowledge reuse seem to be underexploited, which is puzzling, given the encouraging results that may be obtained using even straightforward means, as we have demonstrated in this study.

6. ACKNOWLEDGMENT

This research has been supported by the Ministry of Science and Higher Education grant # N N519 3505 33.

7. REFERENCES

[1] A. Bajurnow and V. Ciesielski. Layered learning for evolving goal scoring behavior in soccer players. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1828–1835, Portland, Oregon, 20-23 June 2004. IEEE Press.

[2] R. Caruana. Multitask learning. *Mach. Learn.*, 28(1):41–75, 1997.

[3] E. Galvan Lopez, R. Poli, and C. A. Coello Coello. Reusing code in genetic programming. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 359–368, Coimbra, Portugal, 5-7 Apr. 2004. Springer-Verlag.

[4] J. Ghosn and Y. Bengio. Bias learning, knowledge sharing. *ijcnn*, 01:1009, 2000.

[5] T. Haynes. On-line adaptation of search via knowledge reuse. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 156–161, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[6] G. S. Hornby and J. B. Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artif. Life*, 8(3):223–246, 2002.

[7] D. Howard. Modularization by multi-run frequency driven subtree encapsulation. In R. L. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practise*, chapter 10, pages 155–172. Kluwer, 2003.

[8] W. H. Hsu, S. J. Harmon, E. Rodriguez, and C. Zhong. Empirical comparison of incremental reuse strategies in genetic programming for keep-away soccer. In M. Keijzer, editor, *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, Seattle, Washington, USA, 26 July 2004.

[9] W. Jaśkowski, K. Krawiec, and B. Wieloch. Knowledge reuse in genetic programming applied to visual learning. In D. Thierens, editor, *Genetic and Evolutionary Computation Conference GECCO*, pages 1790–1797. Association for Computing Machinery, 2007.

[10] J. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.

[11] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[12] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. Reuse, parameterized reuse, and hierarchical reuse of substructures in evolving electrical circuits using genetic programming. In T. H. *et al.*, editor, *Proceedings of International Conference on Evolvable Systems: From Biology to Hardware (ICES-96)*, volume 1259 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[13] K. Kurashige, T. Fukuda, and H. Hoshino. Reusing primitive and acquired motion knowledge for gait generation of a six-legged robot using genetic programming. *Journal of Intelligent and Robotic Systems*, 38(1):121–134, Sept. 2003.

[14] S. Louis and J. McDonnell. Learning with case-injected genetic algorithms. *Evolutionary Computation, IEEE Transactions on*, 8(4):316–328, 2004.

[15] S. Luke. ECJ evolutionary computation system, 2002. (http://cs.gmu.edu/ eclab/projects/ecj/).

[16] S. Luke and L. Panait. Lexicographic parsimony pressure. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[17] T. M. Mitchell. The discipline of machine learning. Technical Report CMU-ML-06-108, Machine Learning Department, Carnegie Mellon University, July 2006.

[18] J. O'Sullivan and S. Thrun. A robot that improves its ability to learn, 1995.

[19] L. Y. Pratt, J. Mostow, and C. A. Kamm. Direct Transfer of Learned Information among Neural Networks. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 584–589. AAAI, July 1991.

[20] S. C. Roberts, D. Howard, and J. R. Koza. Evolving modules in genetic programming by subtree encapsulation. In J. F. M. *et al.*, editor, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 160–175. Springer-Verlag, 2001.

[21] J. P. Rosca and D. H. Ballard. Discovery of subroutines in genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996.

[22] G. Seront. External concepts reuse in genetic programming. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 94–98, MIT, Cambridge, MA, USA, 10–12 Nov. 1995. AAAI.

[23] R. Vilalta and Y. Drissi. A perspective view and survey of meta-learning. *Artif. Intell. Rev.*, 18(2):77–95, 2002.

[24] D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.