Evolving Heuristics for Planning

Ricardo Aler, Daniel Borrajo, Pedro Isasi

Universidad Carlos III de Madrid
Butarque 15
28911 Leganés (Madrid), España
e-mail: {aler@inf,dborrajo@ia,isasi@gaia}.uc3m.es

Abstract. In this paper we describe EvoCK, a new approach to the application of genetic programming (GP) to planning. This approach starts with a traditional AI planner (PRODIGY) and uses GP to acquire control rules to improve its efficiency. We also analyze two ways to introduce domain knowledge acquired by another method (HAMLET) into EvoCK: seeding the initial population and using a new operator (knowledge-based crossover). This operator combines genetic material from both an evolving population and a non-evolving population containing background knowledge. We tested these ideas in the blocksworld domain and obtained excellent results.

1 Introduction

Problem solving aims to achieve a set of goals from an initial state by using operators that represent the different actions available in a task domain. Traditional approaches [16, 14] use domain independent planners for generating plans. Some more innovative approaches to problem solving use genetic programming [9]. This approach was started by Koza [8, 9], who evolved a planner that solved a very specific set of problems in the blocksworld domain. Handley [5–7] used GP to evolve plans for specific problems in the blocksworld domain.[1] Muslea [13] generalized, extended, and formalized this idea, and showed how any planning problem could be translated to an equivalent GP problem. He tested it successfully in several domains.

Spector [15] proposed and analyzed three ways in which GP could be used for planning:

- to evolve a plan for a specific problem in a specific domain (Handley's and Muslea's approach);
- to evolve a partially universal planner in a specific domain (like Koza's approach), that is, a planner that is able to solve problems which have the same initial state but different goals states; and
- to evolve a fully universal planner in a specific domain, "capable of achieving a range of goal conditions from a range of initial conditions" [15].

[1] A similar approach in the GA field was used by Davidor [2] to evolve trajectories for a robot arm.

All the three approaches were tested in the blocksworld by Spector. Moreover, in the third approach, he considered only problems with three blocks, though it was shown that it managed to solve *some* problems with four blocks. It is clear that a more general approach is needed.

In the three approaches considered, it is not always possible to write an informative fitness function. It is very easy in those problems that involve motion, where closeness to a goal can be measured as a distance. But in other cases (like Handley's light switch problem) that is not possible, and the fitness function becomes a count of the number of unsatisfied goals. In problems where there are many goals of this kind, the fitness function becomes informative, like Muslea's briefcase problem. However, in an extreme case, with just one goal to achieve, all the feedback the fitness function could return about the worth of a plan would be just 0 or 1. This is too coarse for GP to get much profit from. Even when writing an informative fitness function is possible, it requires an extra domain knowledge beyond the description of the domain. So far, this knowledge must be supplied by the programmer.[2]

We are interested in applying GP to planning in a general, problem-independent, domain-independent way. So, in this paper we explore a different approach to genetic planning, EvoCK, where traditional approaches and GP are successfully combined.

2 EvoCK Context

Instead of trying to evolve a fully universal domain-independent planner, we start with a traditional domain-independent planner, and see where GP can help. PRODIGY4.0 is such a domain-independent planner; more specifically, it is a means-ends analysis nonlinear planner [16]. However, planning becomes impractical for large problems [1]. At several points in PRODIGY's reasoning cycle, it has no guidance to make a decision, making a weak syntactic-based decision. PRODIGY can be supplied with domain-dependent search knowledge so that its decisions are guided. Here is where GP can be used for planning: instead of evolving a whole planner, we evolve the domain-dependent control knowledge that PRODIGY lacks.

PRODIGY has four kinds of decision points where control knowledge can help to make a better decision in its reasoning cycle:[3]

- Select a goal from a set of pending goals.
- Select an operator to achieve a goal.
- Select a binding for the chosen operator.
- Choose whether to apply the operator with the bindings or to subgoal on an unachieved goal.

[2] Of course, this can be seen from a positive perspective too: it allows us to introduce more domain knowledge into the planner than traditional methods

[3] There are other decision points that can be guided by control knowledge, but we will not use them.

There are other methods that have been applied to the production of control knowledge. Most of them involve acquiring control rules by observing a large set of traces of problems successfully solved by the planner [11, 3]. HAMLET [1] is one of such systems. It learns control rules incrementally, and it has been shown that HAMLET's control rules improve monotonically when more and more traces are supplied. So the question is: why use GP for this task at all? There are two main reasons. First, GP search biases are very different from these methods. If we can combine search biases from two different methods, it is expected that we will get solutions that would not have been obtained by using just one of the methods alone. In our case, that combination can be achieved by introducing domain knowledge into GP obtained by HAMLET when acquiring the control rules. The second advantage is GP's flexibility: very different search biases can be used and changed in GP very easily, without changing the method itself.

3 EvoCK: The System

The main aims of this paper are twofold. First, we want to improve a planning system (PRODIGY4.0) by means of GP-generated control-rules. Second, we want to study the effects of injecting background knowledge coming from a machine learning method into a GP system.

3.1 Overview

The main relations of our system (EvoCK) with the planner (PRODIGY) and the previous learning system (HAMLET) are shown in Figure 1. The generation of control knowledge consists of two learning phases. In the first one (dashed lines in the figure), HAMLET learns from a randomly generated set of problems. For EvoCK purposes, there are two main outputs: a set of control rules, that will configure the initial population of EvoCK; and a set of background knowledge, used as a secondary non-evolving population for EvoCK as explained later. For each problem, PRODIGY also generates a search tree, which is stored by the Search Monitor for later use in the second phase. During the second phase (solid line in the figure), EvoCK evolves to obtain an individual for guiding the search of a solution, using or not using HAMLET outputs.

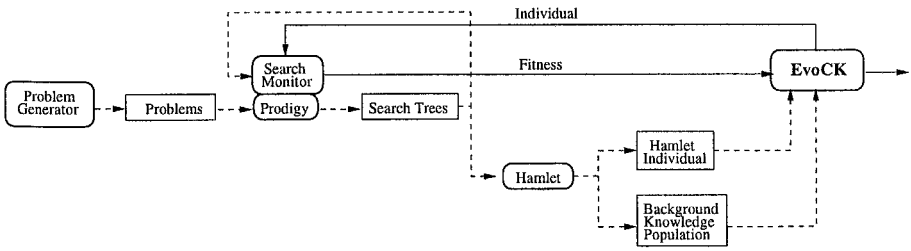

Fig. 1. Relations of EvoCK with HAMLET and PRODIGY.

EVOCK individuals are sets of PRODIGY control rules. One important point is that both initial and genetically generated individuals must be syntactically correct. In the literature, such structures are called "constrained structures" [9] . For that purpose, we use a grammar that describes how individuals must be generated. Following, a EVOCK individual with just one control-rule is shown:

```
if (and (true (clear <obj1>))
        (candidate-goals (on <obj1> <obj2>)) (holding <obj3>))
then (select-goal (on <obj1> <obj2>))
```

The previous individual means that if there is an object A with no objects on it and we are trying to achieve either to have that object A on another object B or to hold another object then we should work next in trying to put object A on object B. Often, individuals have several control-rules. There are four kinds of control-rules, one for each kind of decision PRODIGY makes (see section 2).

EVOCK uses a tournament method for selecting the parents as well as for choosing an individual that offspring will replace. The fitness function is made of several components. One of them (called "performance in fitness cases") tests how well the individual performs when PRODIGY tries to solve the training planning problems when guided by the individual (acting as a set of control rules). The rest of the components test properties of the individual itself, such as the number of variables in the rule, the number of rules, the size in nodes, etc. All individuals in the tournament are compared according to the first component. If there are draws, the second component is used, and so on. The first component ("performance in fitness cases") will be explained in detail in the subsection 3.2.

EVOCK genetic operator set includes the standard set (crossover and mutation), another set to change, add and remove rules from an individual, the knowledge-based crossover (which will be explained in subsection 3.3) and the join operator, which has been specially tailored for control-rules. Its effects won't be analyzed in this paper.

3.2 Performance in fitness cases

The "performance in fitness cases" measure deserves a longer explanation. It uses the search trees obtained from PRODIGY in Figure 1. Control rules allow PRODIGY to prune the search tree, so an immediate performance function could be the number of nodes expanded by PRODIGY when trying to solve a problem. The better an individual (control rule) would do, the fewer nodes would PRODIGY expand. However, this performance measure would discriminate poorly between bad individuals: they would just exhaust the maximum time allocated to fitness evaluation and return exactly the same performance measure. Therefore, the performance function must be made both more efficient and more informative. Following paragraphs explain how this is done.

In the first phase, all of the fitness cases are supplied to PRODIGY. Then, it will produce several solutions in the form of a set of paths in the search tree,

from the initial state to the state where the goals are satisfied. Solutions can be sorted according to a quality criterion. We retain only the best solutions of the set produced by PRODIGY. The set of best solutions is just a part of the search tree, where all leaf-nodes are goal states of the problem to be solved (and the root is the initial state). Let us call it the "best solutions tree."

The performance of an individual C (a set of control rules) over a set of problems P_i is measured as:

$$PF(C) = \left(\sum_i \frac{S_{C,i}}{S_{C,i} + E_i} \right) / N \tag{1}$$

For each P_i, we calculate the number of steps $S_{C,i}$ that PRODIGY, being guided by C, manages to follow in the best solutions tree. Then we divide it by the total number of steps C should have followed. E_i is the expected number of steps left to get to a leaf node. It is "expected" because not all paths leading from a given node to a leaf node will have the same length, so the *expected length* is averaged over all of them. N is the number of problems EvoCK is being trained with.

This performance function turns out to be much more tractable and informative.

3.3 The knowledge-based crossover operator

This is a crossover operator that is very useful if it is possible to represent background (or domain knowledge) as GP individuals, that is, individuals that would represent approximations to a solution, a partial solution, a good building block for a solution, or a good starting point to get to a solution. In some cases, we could just seed the initial population with those individuals and let it evolve. But we want to explore another possibility: using those individuals as a source from where a genetic operator might get some profit.

In our case, when HAMLET is in the process of inducing control rules, it generates a subproduct that consists of all the successful and failed decisions that PRODIGY made when solving a problem. These decisions were made in each of the four kinds of decision points we saw in section 2. It is straightforward to convert those decisions to a control rule format and then to a EvoCK individual. Those decisions are too specific, and dependent of the point where that decision was made. But they seemed like a good source of materials for a crossover operator. That is exactly what the knowledge-based crossover is: a standard crossover but, instead of choosing the second parent from the population, it draws it from a population that is known to be a good source of genetic material.

4 Experimental Results

We carried out several empirical tests to study two ways to inject domain knowledge coming from HAMLET into EvoCK:

- To introduce control rules generated by HAMLET into the initial population ("HAMLET individual" in Figure 1).
- To use the knowledge-based crossover operator ("Background Knowledge Population" in Figure 1).

Therefore, there are four categories of experiments:

1. **RSRM:** Random initial population (that is, a random seed) and random mutation.
2. **RSKC:** Random seed and knowledge-based crossover.
3. **HSRM:** Initial population seeded by HAMLET (HAMLET seed) and random mutation.
4. **HSKC:** Initial population seeded by HAMLET and knowledge-based crossover.

Each category is divided into two configurations:

- EvoCK with a population of two individuals. As the selection method is a tournament, our genetic search amounts to some sort of hill climbing in this case. Besides the knowledge-based crossover operator, no other crossover operators will be used, just mutation.
- EvoCK with a population of 50 individuals. Even with this small population size, in GP terms, we obtained quite impressive results. This configuration is identified by prefixing **P** to the name. Thus, **PHSKC** will identify the configuration with the following characteristics: HAMLET seed, knowledge-based crossover and a population of 50.

Therefore, we have eight sets of experiments (or configurations): RSRM, RSKC, HSRM, HSKC, PRSRM, PRSKC, PHSRM and PHSKC.

We use a steady state GP with a generational gap of 1. Tournaments are held for both selection and replacement. The number of evaluations is limited to 100,000 which amounts to about 242 generations. 410 planning cases randomly generated for the blocksworld domain were used as fitness cases. They contain problems of various degrees of difficulty (their number of goals ranging from 1 to 5 and their number of objects ranging from 2 to 10).

To test best of run individuals, 416 test problems were used. Their number of goals ranging from 5 to 50 and their number of objects ranging from 2 to 50. The testing cases are extremely harder than the fitness cases, its purpose being to check whether GP has generalized well. About 10 experiments were carried out for each experimental configuration, obtaining a best-of-run individual from each experiment.

The blocksworld domain consist of blocks that can be picked up by a robot arm, stocked on other blocks or put down on a table. A problem in this domain consists of an initial state, that is, a configuration of blocks, and a final state to be reached. The goal state is composed of several predicates (or goals) to be fulfilled by the planner. A plan that transforms the initial state in the final state is a solution to the problem. The more goals and blocks are included in a problem, the more difficult it is to solve that problem. For instance, problems

with 3 goals and 4 blocks are easily solved by PRODIGY, but problems with 50 goals and 50 objects are seldom solved (this will be seen later in Table 2).

Table 1 summarizes the results obtained from the different configurations. The first four columns show data related to the best of the best individuals obtained by each configuration: the generation at which the individual was obtained, the number (and percentage) of testing problems solved by it (P. Solved), and the number of control rules (size). Column "averages" shows the average number of testing problems solved. It has been obtained by averaging results from the several best-of-run individuals belonging to each experimental configuration. Results for PRODIGY working alone and results for PRODIGY using control-rules generated by HAMLET are also shown.

	Gen.	P. solved	Size	Averages
PRODIGY		**86** (21%)		
HAMLET		**238** (58%)	**12**	
RSRM	236	**242** (59%)	**5**	99
PRSRM	195	**331** (81%)	**1**	161
RSKC	212	**196** (48%)	**4**	165
PRSKC	89	**331** (81%)	**1**	166
HSRM	202	**320** (78%)	**4**	229
PHSRM	154	**358** (87%)	**3**	276
HSKC	199	**358** (87%)	**3**	307
PHSKC	237	**358** (87%)	**4**	248

Table 1. This table shows the results obtained by the best individual produced by each of the configurations. "P. Solved" is the number of testing problems solved by said individuals.

5 Discussion

First, Table 1 shows clearly that EVOCK improves on PRODIGY even when it uses no background knowledge. PRODIGY alone is able to solve 86 test problems only, whereas RSRM managed to solve 242 test problems. It even manages to perform better than HAMLET. PRSRM results are even better, outperforming RSRM. However, better results are obtained when using background knowledge: PHSRM, HSKC and PHSKC solve 358 problems. This is better than the seed supplied by HAMLET, which solves 238 problems only. Table 2 shows that this is a qualitative improvement, because HSKC (and PHSKC) manages to solve problems of a degree of difficulty (50 goals and 50 objects) that HAMLET control-rules alone were unable to cope with in the allocated time. Also, (P)HSKC improves on HAMLET results on all categories of problems, except by a small percentage in the (5,10) category.

# Goals	#Objects	%PRODIGY	%HAMLET	%EVOCK
50	50	0	0	53
20	50	6	31	79
20	20	6	27	65
10	50	21	66	95
10	20	11	54	80
10	15	30	46	82
5	50	15	68	80
5	20	15	82	94
5	15	40	82	98
5	10	50	84	83

Table 2. This table shows the percentage of testing problems solved by the three systems: PRODIGY, HAMLET and EVOCK for each category of problems.

Second, EVOCK individuals tend to be much smaller than HAMLET ones (even when they are better). Table 1 shows that even though HSKC best individual performs better than HAMLET one, it only has 3 control rules (vs. 12 in HAMLET). Smaller individuals will be interpreted faster by PRODIGY. This results show how HAMLET search biases (which result in the HAMLET generated control-rules) and EVOCK search biases (the parsimony component in the fitness function) can be successfully combined.

Third, the use of the knowledge-based crossover operator makes a significant difference. To see whether the results are consistent in most of the runs, average results will be analyzed (see Table 1). RSKC consistently beats RSRM results (165 vs. 99 on average), while HSKC is also better than HSRM. PRSKC improvement with respect to PRSRM is less noticeable (166 vs. 161 on average) and there is no improvement at all in PHSKC on PHSRM results (248 vs. 276 on average). This latter anomaly is a matter for further research. Seeding the initial population with a good individual (HAMLET generated control rules) makes also a significant difference: all HAMLET-seeded configurations beat the corresponding randomly seeded ones on average.

Fourth, experiments with a population size of 50 tend to outperform (on average) pseudo hill-climbing. This suggests that either crossover or bigger population sizes are having a noticeable effect (there is an anomaly in the fourth set of experiments, though: HSKC beats PHSKC).

6 A New Perspective for Injecting Background Knowledge into GP

We believe to have identified a possible new way to inject domain knowledge (or background knowledge) into GP. There are many ways in which it is possible to give background knowledge to a GA. For instance, seeding the initial population with good individuals, adapting the genetic operators to the problem at hand (or creating new ones specially tailored for the task), etc. We ourselves have used

those methods in this article (HAMLET seed, the knowledge-based crossover and the join operator). Also, in GP it is possible to use constrains (to constraint the evolving structures [9, 12]) and therefore, they reduce the search space. And of course, it is always possible to select the function and terminals most appropriate to the domain or to the problem.

We think our work points to a new direction: do not evolve the whole program, evolve only that part of the program that you do not know how to program, or that could be improved. In our case, we started with a program (PRODIGY, a domain independent planner), we saw that there were some parts of the program that could be improved (PRODIGY's four decision points) and we let GP evolve those parts. But what was actually evolving was a population of whole planners, because the fitness function was computed from the performance of the entire planner, not only of the control rules. PRODIGY was designed with that sort of decomposition of the planning process in mind. But we believe that this idea could be applied to other problems as well. For instance, if a programmer considers that a problem solution needs two nested loops, s/he could just fix that in the program and leave the rest blank. Then, GP would fill in the blanks.

7 Conclusions

The major contribution of this paper is to show that Genetic Programming can be successfully applied in a new way to traditional planning problems, by means of acquiring control rules for a domain independent planner. For that purpose, we have studied two ways in which domain knowledge can be injected into GP: by seeding the initial population with a good individual coming from another learning method (HAMLET) and by means of the knowledge-based crossover operator. Both of them work very well in this case. Although knowledge-based crossover is specially adequate for this task, we believe it could be used in other GP problems as well.

Along the way, we have realized we were also introducing domain knowledge into GP implicitly, by fixing part of the program that was to evolve (PRODIGY), and evolving the rest (PRODIGY's decision points). We believe that this idea, obvious as it seems, could be exploited in many other GP problems.

We have shown that different biases from different search methods (HAMLET and EvoCK, in this case) can be combined successfully. This allowed us to obtain very good and very small and efficient individuals from a Hamlet seed.

References

1. Daniel Borrajo and Manuela Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning*, 11(1-5):371–405, February 1997.
2. Y. Davidor. *Genetic Algorithms and Robotics.* The IEEE Computer Society Press, 1992.

3. Tara A. Estlin and Raymond Mooney. Hybrid learning of search control for partial-order planning. In *New Directions in AI Planning*, pages 115–128. IOS Press, 1996.
4. Oren Etzioni. Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–301, 1993.
5. S. Handley. The automatic generation of plans for a mobile robot via genetic programming with automatically defined functions. In *Proceedings of the Fifth Workshop on Neural Networks: An International Conference on Computational Intelligence: Neural Networks, Fuzzy Systems, Evolutionary Programming, and Virtual Reality*, 1991.
6. S. Handley. The genetic planner: The automatic generation of plans for a mobile robot via genetic programming. In *Proceedings of the Eighth IEEE International Symposium on Intelligent Control*, 1993.
7. Simon G. Handley. The automatic generations of plans for a mobile robot via genetic programming with automatically defined functions. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 18, pages 391–407. MIT Press, 1994.
8. J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774. Morgan Kaufmann, 20-25 August 1989.
9. John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
10. J.R. Koza. *Genetic Programming II*. The MIT Press, 1994.
11. Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, Boston, MA, 1988.
12. David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.
13. Ion Muslea. SINERGY: A linear planner based on genetic programming. In *Fourth European Conference on Planning*, Toulouse, France, 24–26 September 1997.
14. J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of KR-92*, pages 103–114, 1992.
15. L. Spector. Genetic programming and AI planning systems. In *Proceedings of Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.
16. Manuela Veloso, Jaime Carbonell, Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical AI*, 7:81–120, 1995.