Genetic Programming and Deductive-Inductive Learning: a Multi-strategy Approach

Ricardo Aler, Daniel Borrajo, Pedro Isasi
Departamento de Informática, Universidad Carlos III de Madrid
28911 Leganés (Madrid), Spain
email: {aler@inf,dborrajo@ia,isasi@ia}.uc3m.es

Abstract

Genetic Programming (GP) is a machine learning technique that was not conceived to use domain knowledge for generating new candidate solutions. It has been shown that GP can benefit from domain knowledge obtained by other machine learning methods with more powerful heuristics. However, it is not obvious that a combination of GP and a knowledge intensive machine learning method can work better than the knowledge intensive method alone. In this paper we present a multi-strategy approach where an analytical and inductive approach (HAMLET) and an evolutionary technique based on GP (EvoCK) are combined for the task of learning control rules for problem solving in planning. Results show that both methods complement each other, supplying to the other method what the other method lacks and obtaining better results than using each method alone.

1 INTRODUCTION

Genetic Programming (GP) is a machine learning technique based on a search over a huge state space [Koza and Rice, 1991]. Therefore, as any search method, it can be defined in terms of three elements: an initial state, a set of operators, and a heuristic function (called fitness function). GP expands the ideas of Genetic Algorithms by using structured representations (trees). The use of this type of representation is more appropriate for solving symbolic tasks than Genetic Algorithms.

One of such tasks consists on learning control knowledge for problem solving. Problem solving can also be described in terms of a search in another state space than the one of GP. Traditional approaches use domain independent planners for generating plans [Blum and Furst, 1995, Penberthy and Weld, 1992]. PRODIGY, an architecture for planning and learning that uses a means-ends analysis nonlinear planner, is one of such systems [Veloso *et al.*, 1995]. However, planning becomes impractical for large problems. In order to gain efficiency, PRODIGY must be supplied with domain-dependent search control knowledge which can be applied at decision points in the planning reasoning cycle. This control knowledge has the form of control rules, as further explained later on.

In this type of tasks, the use of all available domain knowledge is essential for an efficient learning process. Classically, GP systems have only used domain knowledge for the fitness function. We propose the use of background knowledge coming from the use of a previous learning technique also in another two search elements [Aler *et al.*, 1998a]: first, the initial state will not be created randomly, but using control knowledge learned by another method, HAMLET in this case [Borrajo and Veloso, 1997]. Second, genetic operators will use knowledge in the form of examples, obtained as a sub-product of HAMLET learning process.

In [Aler *et al.*, 1998a] we have shown that GP obtains much better results in planning by using such background knowledge. The purpose of this paper is to show that a multi-strategy approach using GP and HAMLET works better than using each method alone. This multi-strategy approach can be seen as a combination of learning bias from different methods: GP and HAMLET. In this paper, we have used PRODIGY, but in the future other planners such as UCPOP or GRAPHPLAN might be used.

Section 2 explains the role of learning in planning. Section 3 describes our multi-strategy approach for learning in planning. Section 4 describes our experimental setup and the results obtained. Section 5 discusses these results, and presents the conclusions. Finally, Section 6 surveys related work.

2 THE LEARNING TASK

The learning task can be stated as: given a set of traces belonging to problems solved by PRODIGY in a particular planning domain, induce a set of control rules that perform well in that planning domain. Control rules help PRODIGY to make decisions at several points in its search process. If there are no applicable control rules in a decision point, PRODIGY will make a default decision. It has five kinds of decision points:[1]

- Select, prefer or reject a goal from the set of pending goals.
- Select, prefer or reject an operator to achieve a goal.
- Select, prefer or reject a binding for the chosen operator.
- Choose whether to apply an instantiated applicable operator or to subgoal on an unachieved goal.
- Select, prefer or reject an instantiated operator from the set of applicable instantiated operators.

Figure 1 shows an example of a control rule for the blocksworld domain. `current-goal,` and `true-in-state` are meta-predicates. The control rule says that if PRODIGY is working on trying to hold an object, `<object1>`, and this object is on top of another, `<object2>`, in the current state, then PRODIGY should select the operator `UNSTACK` and reject the rest of operators that could achieve the same goal.

```
(control-rule select-operators-unstack
     (if (and (current-goal (holding <object1>))
              (true-in-state (on <object1> <object2>))))
     (then select operator unstack))
```

Figure 1: Example of a control rule for making the decision of what operator to use.

[1] HAMLET only generates selection control rules. In this article, GP will look just for that kind of control rules, so that it can be properly compared with HAMLET.

At every decision point, PRODIGY is in a particular search meta-state. Let ME be the set of all possible meta-states. Now, helping PRODIGY to take decisions can be stated as: for each possible decision (for example: `select goal (on x y)`) find a partition of ME into $ME+$ (where the decision should be taken) and $ME-$ (where the decision should not be taken). That is, control rules are actually classification rules: they partition the space of meta-states into those meta-states that belong to a possible decision and those that do not. And this looks like traditional machine learning concept induction, where classification rules have to be induced from a set of examples. In this case, it has the following characteristics:

- Several target concepts have to be learnt from the same data (set of traces). Not only there are different kinds of target concepts associated to each kind of decision (select operator, select goal, etc) but each kind of decision has several associated target concepts. For instance, there will be one target concept of the type select operator for each possible (operator, goal) pair of a particular domain.
- Target concepts will generally be disjunctive (that means that several control rules will be needed to represent a target concept).
- The representation of concepts is relational, so we are dealing with an ILP problem.

Therefore, when using GP, each individual will be a set of control rules, represented as a structure that will be explained in Section 3.2. A GP population is made of several such individuals.

3 A MULTI-STRATEGY APPROACH FOR LEARNING CONTROL KNOWLEDGE

In this section we will describe the architecture of the learning system, and define the learning behavior in terms of its three learning biases.

3.1 ARCHITECTURE OF THE LEARNING SYSTEM

The general architecture of our system consists of five blocks (as also shown in Figure 2). The main blocks are EvoCK ("Evolution of Control Knowledge") and HAMLET.

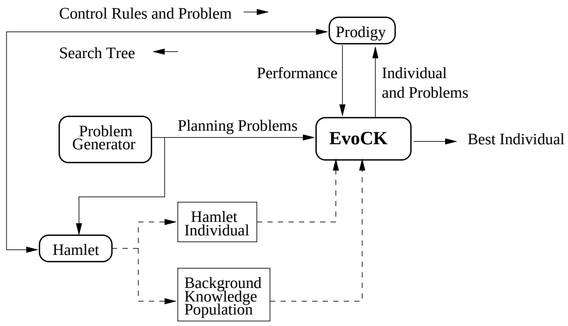

Figure 2: General Architecture of the multi-strategy approach.

EvoCK is the module that implements the GP paradigm adapted for evolving planning control rules. EvoCK is supplied with fitness cases generated by a problem generator. These fitness cases are planning problems generated at random by the problem generator. In order to evaluate individuals from the population with the fitness cases set, EvoCK tells PRODIGY to load the individual and try to solve each one of the fitness cases. Performance of this individual with these fitness cases is returned to EvoCK. HAMLET has a similar relation with PRODIGY and the problem generator but in this case the information returned by PRODIGY is the search tree that HAMLET will use to generalize and refine its control-rules.

EvoCK and HAMLET are weakly coupled in the following way. First, HAMLET is run to learn from a set of randomly generated problems. Then, two of its outputs are used as background knowledge for EvoCK: the set of rules learned by HAMLET ("HAMLET individual") are used to seed the EvoCK initial population. Also, the HAMLET supplies a set of positive examples ("Background Knowledge Population") that will be taken as input by one of the genetic operators (knowledge based crossover [Aler *et al.*, 1998a]). This will be explained in subsection 3.3.

When EvoCK gets to the maximum number of evaluations allowed for learning, it returns its best individual obtained so far. Although not shown in Figure 2, best individuals are tested with a different set of planning problems (also obtained from the problem generator) to check how well they have generalized from the training data.

In the next three sections, we describe the system by explaining its learning biases. These biases are classified following Utgoff [Utgoff, 1986] in language biases, exploration biases and evaluation biases.

3.2 THE LANGUAGE BIAS

Usually, in GP there are no constrains in the structure that is to evolve: any combination of functions and terminals will be valid and crossover points can be taken at any place in the individual. But, in our case, PRODIGY restricts what are valid structures and what are not. For instance, a meta-predicate like `TRUE-IN-STATE`[2] can only be passed as argument a goal like `(on <x> <y>)` but not an operator like `PUT-DOWN`. Other general constrains are imposed by the structure of the rule language itself (if <condition> then <action>, etc). In many cases this problem can be solved by achieving operational closure, that is, by allowing each function to accept any type of result [Koza and Rice, 1991]. However, this is not possible in this case, since PRODIGY fixes the structure of the language for representing control rules and feeding it with non-valid control rules would make it fail.

Therefore, we have chosen to constrain structures to PRODIGY-valid ones (in the literature, such structures are called "constrained structures" [Koza and Rice, 1991] or "strongly typed structures" [Montana, 1995]). In order to achieve it, the following three steps must be followed: create only valid structures, crossover points must be of the same type and mutation operators must take into account the type of the mutation point. The first step is achieved by using an special-purpose production grammar. An example of an individual generated by the grammar might be the one that appears in Figure 3. This individual consists of two control rules for the blocksworld domain. The first one checks whether there is a block with no other blocks on it and if the planner is trying to solve either putting that object on another object or having the robot arm hold a third different object. If both conditions succeed, then the planner will work next in the `(on <object-1> <object-2>)` goal. The other control rule says that if there is an object on the table and the system is trying to bind the `pick-up` operator, then it should be bound to that object.

3.3 THE EXPLORATION BIAS

The exploration bias includes everything related to the search policy: search operators, background knowledge to constrain the search, etc. The system uses

[2]Meta-predicates are functions that have access to PRODIGY meta-state. Therefore they can check whether a condition is true or not in the meta-state. For instance `TRUE-IN-STATE` tests if a particular condition is true in the current planning state

```
(list (rule (and (true-in-state (clear <object-1>))
                 (some-candidate-goals
                   (goals-list (on <object-1> <object-2>))
                               (holding <object-3>)))
            (select-goal (on <object-1> <object-2>)))
      (rule (true-in-state (on-table <object-1>))
            (select-bindings (pick-up-b <object-1> ))))
```

Figure 3: Example of EvoCK individual.

the traditional GP operators (crossover and mutation) and some others specially tailored for the learning task. The whole operator set is:

- **Copy**: reproduction without modification.
- **Xover**: traditional crossover. It takes two constrained structures and produces one constrained structure
- **Changing_mutation**: it chooses a mutation point, and changes the whole subtree by another randomly generated subtree. This mutation is equivalent to **Xover** with a randomly generated individual (as the second parent).
- **Xover_add**: some points in the evolving structure allow for lists of elements of the same kind (as, for instance, lists of goals). In those cases, crossover adds elements to the lists from the other parent, instead of replacing the whole list.
- **Chopping_off_mutation**: in those points where lists of elements of the same kind are allowed, it removes one of the elements.
- **Growing_mutation**: it adds a random subtree at those points where lists of elements of the same type are allowed. It is equivalent to **Xover_add** with a randomly generated individual (as the second parent).

All these operators are simple variations of genetic operators traditionally used in GP. The next two operators are specially tailored for this learning task.

- **Join**: it selects one variable in the control rule (like `<object-1>`) and substitutes it by any other variable in the control rule. The rationale behind this operator is that sometimes there are conditions in a rule that are not related with other conditions by common variables. Sometimes that is undesirable. For instance, if we have a control rule to pick-up an object `<obj1>` when some conditions are true, our experience says that many of those conditions should refer to `<obj1>`. The **join** operator is a simple way of creating these references.
- **Up_the_hierarchy**: objects (the elements to which the planning operators are applied) in PRODIGY are organized in a tree-shaped type hierarchy. For instance, in logistics transportation planning domain, there are trucks and planes, which are both defined as carriers. This genetic operator would take a truck-typed variable in the left hand side of the rule and would substitute all its instances by a carrier-typed variable. Thus, the control rule would become more general.

The related specialization operators (i.e. disjoin and down_the_hierarchy) are not included in the operator pool; we are imposing a strong bias towards generalization. However, the system can still specialize by means of the other generic operators (mutation, etc).

Background knowledge can be introduced to the system in order to restrict the search. So far, we have used two kinds of background knowledge:

- Seeding the initial population with an individual coming from HAMLET.
- The early phase of HAMLET returns a set of positive and negative examples as a sub-product. Positive examples are those where PRODIGY made the right decision in the planning process. These positive examples can be easily transformed into control rules and then into GP individuals. Then, the crossover operator will be able to draw individuals from the background knowledge population instead of the evolving population (this is what we have called "knowledge-based crossover operator" [Aler *et al.*, 1998a]). In that way, background knowledge can be injected into the evolving population.

Finally, we use a steady state GP with a generational gap of 1. 2-tournaments are held for both selection and replacement. This has been shown experimentally to behave well.

3.4 THE EVALUATION BIAS

The evaluation bias concerns the preference criteria used by GP for selecting an individual over another, which is coded as a fitness function. In our case, we devised a hierarchical fitness function that contains the following components [Aler *et al.*, 1998b, Aler *et al.*, 1998a]:

1. **Performance in fitness cases:** to maximize. 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). It will explained later in more detail.
2. **Number of different variables:** to minimize. This fitness component is related to the same bias than the join operator. We want to have as many meta-predicates in the left hand side of the control rules inter-related by common variables as possible.
3. **Number of different true-in-state meta-predicates:** to minimize. The fewer true-in-state meta-predicates, the more general and faster will run the set of control rules.
4. **Number of different goals in some-candidate-goals meta-predicates:** to maximize. This meta-predicate returns true if at least one of its arguments is a candidate goal to be solved by the planner. So, the more goals has some-candidate-goals, in more cases it will be applicable and the more general it will be (although less compact).
5. **Number of different some-candidate-goals:** to maximize. Another way of making a rule more general is to get rid of unnecessary some-candidate-goals checking. This also makes it faster.
6. **Number of control rules:** To minimize. The fewer control rules, the faster will the individual solve the problems.
7. **Individual size (in nodes):** To minimize.

All individuals in the tournament set that have the same score in the first comparison will pass to the second one and so on. The rest will be dropped off the tournament. If more than one individual happen to pass the last comparison, the tournament winner is chosen randomly.

The first criteria, performance in fitness cases, was formerly computed by measuring how many steps of the solution of a given planning problem the individual managed to follow (solutions to all the planning problems were known by EvoCK in advance by letting PRODIGY solve those problems and storing the search trees). However, although we obtained good results, we realized that an individual managing to follow many steps in the solution didn't guarantee that the individual would actually solve the problem. Therefore, we have decided to change it for a set of three new criteria:

- **Number of problems solved** by PRODIGY being guided by the individual with a maximum node limit. To maximize. This node limit is four times the amount of nodes that would be needed to solve the problem if PRODIGY could go straightforward to the solution.
- **Number of problems solved by the individual more efficiently than** PRODIGY **alone**. To maximize. Efficiency in this case means fewer nodes expanded.
- **Total number of nodes expanded by the individual**. To minimize.

In order to test an individual with these new criteria, it has to be loaded into PRODIGY. Then, PRODIGY will be run for each of the planning problems for learning (or fitness cases, in GP terminology). However, complex problems need to be given a high node limit if they are to be solved. As many such evaluations must be performed for each generation, only simple problems can be used for learning (otherwise the fitness function would take too long). This is another bias to take into account.[3] However, [Borrajo and Veloso, 1997] shows empirically that training with simple problems is enough for learning control knowledge useful to solve more complex problems.

4 EXPERIMENTAL RESULTS

In order to test our multi-strategy approach, the following steps were carried out:

1. Hamlet was trained with 400 learning planning problems. Two domains were used: blocksworld and logistics. A set of control rules and a set of positive examples were obtained for each domain. They were used as background knowledge in the next step.
2. EvoCK was trained in the blocksworld and logistics with 192 and 188 learning planning problems

[3] [Aler *et al.*, 1998b, Aler *et al.*, 1998a] was not constrained by this bias.

respectively. A population size of 2 was used. Certainly, a population size of 2 is not common in GP. Previous work [Aler *et al.*, 1998a] shows that using a bigger population seems to be good but results are not conclusive: the interaction between population size and seeding the initial population is not properly understood yet. In our case, the seeding individual (coming from HAMLET) is much better than the other initial individuals (randomly generated) therefore two things might happen: first, during the earlier generations the seeding individual would not be selected very often, so some time would be spent evaluating individuals that contain no knowledge. Second, if the seeded individual is much better than the randomly generated individuals, in the long term all members might contain similar genetic information to the seeded individual [Fraser and Rush, 1994]. In this paper a population of 2 has been chosen because in that way, we make sure that genetic operators will always act on individuals which contain knowledge and therefore, the impact of knowledge will be better controlled. In any case, we plan to carry out several experiments that will study the population size-population seeding interaction in detail. Performing crossover in such a small population is not meaningful, so standard crossover is not used in this paper. However, EvoCK can use it in general. Background knowledge from the previous step was used in the two ways described in subsection 3.3. As GP is a stochastic method, several experiments were carried out for each domain: 47 for the blocksworld and 54 for logistics. Each experiment ran for 100.000 evaluations. From each experiment, a set of control rules was obtained.

3. HAMLET was trained in a similar manner than EvoCK. HAMLET started with the sets of control rules obtained in step 1 and refined them with the rest of the learning problems used to train EvoCK. Two sets of control rules were obtained (one for each domain).

4. Finally the sets of control rules obtained by EvoCK and HAMLET were tested with a new set of problems (416 for the blocksworld and 347 for logistics) in the same conditions. Results are shown in Table 1. As EvoCK obtained one set of rules from each experiment, two quantities are shown: the number of problems solved by the best of all sets of control rules (along with the number of control rules for that individual) and the average number of problems solved over all sets.

Table 1: Results for PRODIGY, HAMLET and EvoCK in both the blocksworld and logistics domains.

	% Prob. Solved	Number of Rules	Average % P. Solv.
Blocksworld			
PRODIGY ALONE	21%		
HAMLET SEED	58%	12	
HAMLET	18%	13	
EvoCK (best indiv.)	87%	4	80%
Logistics			
PRODIGY ALONE	43%		
HAMLET SEED	52%	56	
HAMLET	46%	64	
EvoCK (best indiv.)	95%	19	65%

Table 1 shows that when HAMLET tries to refine and improve a set of control rules previously learned (HAMLET seed in Table 1), the percentage of test problems actually solved drops: in the blocksworld it goes from 58% to 18%, in logistics it gets from 52% to 46%. On the other hand, EvoCK improves the set of control rules given as seed for the initial population: 58% to 87% in the blocksworld and 52% to 95% in logistics. Next section comments on these results. It is also noticeable that EvoCK produces individuals with fewer control rules than the seeding individual (12 to 4 control rules in the blocksworld and 56 to 19 in logistics) hence returning more efficient individuals. In order to show that the control rules learned are general and useful for more complex problems, a breakdown of the results are displayed in Tables 2 and 3.

Table 2: Breakdown of the number of testing problems solved in the blocksworld by HAMLET and EvoCK according to the number of goals and of objects).

# Goals	# Objects	PRODIGY	HAMLET seed	HAMLET	EvoCK
50	50	0%	0%	0%	56%
20	50	6%	31%	4%	81%
20	20	6%	27%	4%	69%
10	50	21%	67%	19%	96%
10	20	15%	56%	15%	83%
10	15	31%	48%	15%	85%
5	50	15%	70%	2%	92%
5	20	15%	82%	18%	95%
5	15	40%	82%	35%	98%
5	10	50%	85%	60%	95%

Tables 2 and 3 show a breakdown of the number of problems solved by the different methods in the blocksworld according to problem complexity. This end). Thus, we see that our multi-strategy approach works better than the two methods alone by combining both methods biases.

Table 3: Breakdown of the number of testing problems solved in logistics by HAMLET and EvoCK according to the number of goals and of objects).

# Goals	# Objects	PRODIGY	HAMLET seed	HAMLET	EvoCK
50	50	0%	0%	0%	75%
20	50	3%	0%	0%	100%
20	20	7%	28%	0%	83%
10	50	13%	0%	0%	100%
10	20	20%	53%	33%	100%
10	15	20%	67%	47%	100%
10	10	7%	67%	40%	100%
5	50	42%	0%	0%	100%
5	20	58%	83%	67%	100%
5	15	42%	42%	67%	100%
5	10	25%	58%	67%	100%
5	5	33%	83%	92%	100%
2	50	90%	60%	20%	100%
2	20	90%	100%	100%	100%
2	15	90%	90%	100%	100%
2	10	90%	80%	90%	100%
2	5	100%	100%	100%	100%
2	2	100%	100%	100%	100%
1	50	100%	100%	80%	100%
1	20	90%	100%	100%	100%
1	15	90%	100%	100%	100%
1	10	90%	100%	100%	100%
1	5	100%	100%	100%	100%
1	2	100%	100%	100%	100%

complexity is measured by the number of goals and objects in the planning problem. It is easy to see that EvoCK improves drastically with respect to the initial seed (HAMLET seed) by solving very hard problems. The percentage of testing problems solved for PRODIGY working alone, the initial HAMLET seed and the final HAMLET result are also shown.

5 DISCUSSION AND CONCLUSIONS

After having experimented both systems (EvoCK and HAMLET) we can draw the following conclusions and comparisons.

- HAMLET does not have a trade-off between correct knowledge and utility of that knowledge. HAMLET manages to learn quite correct knowledge [Borrajo and Veloso, 1997] but sometimes having a lot of correct control rules is not an advantage, because it takes a long time to use it (this is called the utility problem [Minton, 1988]). This explains in part HAMLET bad behavior. On the other hand, our results in [Aler *et al.*, 1998a] show that it is more difficult for GP alone to obtain correct knowledge. However, it is very easy to take into account the utility problem in the fitness function (several of its components press to that end). Thus, we see that our multi-strategy approach works better than the two methods alone by combining both methods biases.

- Another problem that HAMLET has is that as it is a lazy incremental system, in order to refine an incorrect control rule it assumes that eventually it will find an appropriate set of negative examples. Given that the potential problem space is infinite (huge from a computational point of view), the likelihood of finding that appropriate set might be very small. In any case, previous work has shown that in the long run HAMLET tends to converge to the correct knowledge [Borrajo and Veloso, 1997]. Since GP a non-incremental system, it is able to detect negative examples at once by evaluating the whole set of training problems. On the other hand, non-incremental methods are less efficient when learning in complex domains. Again, the complementary aspects of both systems allow to overcome both systems deficiencies.

- Another difference between using GP in this way and more traditional learning techniques is that even using background knowledge, its generalization and specialization operators do not have knowledge about how planning acts. On the contrary, learning techniques such as PRODIGY/EBL [Minton, 1988], or HAMLET "know"[4] how to generalize or specialize in planning domains. GP has no such knowledge, so many of the genetic modifications will not work. Besides, genetic operators are not so constrained by powerful heuristics, so they might get different and new results than those of more traditional methods. Another way to see this is that HAMLET (and many other learning methods) take advantage of the specific-to-general ordering of the control rule space: HAMLET trajectory through the control rule space consists of generalization or specialization steps, in reaction to new examples [Shapiro, 1983]. GP does not take much advantage of this specific-to-general ordering. A mixture of generalizations and specializations are performed at each step in the search. Besides, generalization operators that take advantage of the ordering heuristic are easily added to the operator pool, as our system shows.

- Given that genetic operators do not handle much knowledge, they are faster than classical learning search operators.

[4] Or at least, they have powerful heuristics.

- HAMLET is deterministic: from the same set of training cases, it will always obtain the same set of control rules. On the other hand, GP is stochastic: it can be run several times and obtain different knowledge every time.
- There is a trade-off between understandability and efficiency. HAMLET tends to produce control knowledge which is easier to understand whereas EvoCK control knowledge is more difficult to understand (but more efficient).
- Finally, an important advantage of GP over the rest of learning techniques applied to problem solving is its flexibility. Very different learning biases can be tested without changing the method itself. Following Utgoff's classification [Utgoff, 1986], GP biases are:
  - The language bias can be changed easily. That is not the case with many other learning techniques applied to problem solving, because their search operators depend heavily on the representation language used. For instance, HAMLET only uses a subset of the control rule language allowed by PRODIGY, while GP could use the whole set easily.
  - The exploration bias. GP uses just two task independent operators (crossover and mutation). However, as this paper shows, many possible variations of these operators can be added, as, for instance, task dependent operators (like generalization and specialization).
  - The evaluation bias. In GP, different evaluation biases can be easily combined in the same evaluation function. Also, it is very easy to change from a fitness function to another. In fact, in this paper we have presented a new fitness function that improves previous results obtained using our scheme [Aler *et al.*, 1998a, Aler *et al.*, 1998b].

6 RELATED WORK

There have been different approaches to acquire control knowledge for non-trivial (non-linear) problem solving. Some of them use analogy [Kambhampati, 1989, Veloso and Carbonell, 1993], others pure deduction [Katukam and Kambhampati, 1994, Minton and Zweben, 1993], pure induction [Leckie and Zukerman, 1991], and some combine deduction and induction [Borrajo and Veloso, 1997, Estlin and Mooney, 1996]. The main difference with our approach is that they did not combine incremental knowledge intensive and non-incremental methods (GP).

Some innovative approaches to problem solving use genetic programming [Koza, 1992]. This approach was started by Koza [Koza, 1989, Koza, 1992], who evolved a planner that solved a very specific set of problems in the blocksworld domain. Handley [Handley, 1994] used GP to evolve plans for specific problems in the blocksworld domain. Muslea [Muslea, 1997] 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 [Spector, 1994] proposed and analyzed several ways in which GP could be used for planning. The main difference with our approach is that they used GP to search in the plans space.

References

[Aler *et al.*, 1998a] Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Evolving heuristics for planning. In *Proceedings of the Seventh Annual Conference on Evolutionary Programming*, Lecture Notes in Artificial Intelligence, San Diego, CA, March 1998. Springer-Verlag.

[Aler *et al.*, 1998b] Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Genetic programming of control knowledge for planning. In Reid Simmons, Manuela Veloso, and Stephen Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, Pittsburgh, PA, June 1998.

[Blum and Furst, 1995] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. In Chris S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI-95*, volume 2, pages 1636–1642, Montral, Canada, August 1995. Morgan Kaufmann.

[Borrajo and Veloso, 1997] 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.

[Estlin and Mooney, 1996] 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.

[Fraser and Rush, 1994] Adam P. Fraser and Jon R. Rush. Putting **ink** into a **bir**o: A discussion of problem domain knowledge for evolutiaonary robotics. In *AISB Workshop, Evolutionary Computing*, April 11th-13th 1994.

[Handley, 1994] 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.

[Kambhampati, 1989] Subbarao Kambhampati. *Flexible Reuse and Modification in Hierarchical Planning: A Validation Structure Based Approach.* PhD thesis, Computer Vision Laboratory, Center for Automation Research, University of Maryland, College Park, MD, 1989.

[Katukam and Kambhampati, 1994] Suresh Katukam and Subbarao Kambhampati. Learning explanation-based search control rules for partial order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 582–587, Seattle, WA, 1994. AAAI Press.

[Koza and Rice, 1991] John R. Koza and James P. Rice. Genetic generation of both the weights and architecture for a neural network. In *Proceedings of IJCNN-91*, volume II, pages 397–404, 1991.

[Koza, 1989] 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.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[Leckie and Zukerman, 1991] C. Leckie and I. Zukerman. Learning search control rules for planning: An inductive approach. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 422–426, Evanston, IL, 1991. Morgan Kaufmann.

[Minton and Zweben, 1993] Steven Minton and Monte Zweben. Learning, planning and scheduling: An overview. In Steven Minton, editor, *Machine Learning Methods for Planning*, chapter 8. Morgan Kaufmann, 1993.

[Minton, 1988] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* Kluwer Academic Publishers, Boston, MA, 1988.

[Montana, 1995] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[Muslea, 1997] Ion Muslea. SINERGY: A linear planner based on genetic programming. In Sam Steel, editor, *Recent Advances in AI Planning. 4th European Conference on Planning, ECP'97*, number LNAI 1348 in Lecture Notes in Artificial Intelligence, pages 312–324, Toulouse, France, September 1997. Springer-Verlag.

[Penberthy and Weld, 1992] 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.

[Shapiro, 1983] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[Spector, 1994] 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.

[Utgoff, 1986] Paul Utgoff. *Machine Learning: An Artificial Intelligence Approach*, volume II, chapter Shift of Bias for Inductive Concept Learning, pages 107–148. Morgan Kaufmann, Los Altos, CA, 1986.

[Veloso and Carbonell, 1993] Manuela M. Veloso and Jaime G. Carbonell. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning*, 10(3):249–278, March 1993.

[Veloso *et al.*, 1995] 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.