Genetic Programming of Control Knowledge for Planning

Ricardo Aler, Daniel Borrajo, Pedro Isasi
Universidad Carlos III de Madrid
28911 Leganés (Madrid), España
aler@inf.uc3m.es, dborrajo@ia.uc3m.es, isasi@gaia.uc3m.es
+34-1-624-9418, +34-1-6249459, +34-1-624-9455

Abstract

It is well known that planning for complex domains and non-trivial problems is a difficult task for domain independent planners. Adding domain knowledge to the planner is a simple approach to improve its efficiency for a given domain. In this paper we present EvoCK, a system based on Genetic Programming, which produces control knowledge for a nonlinear domain independent planner (PRODIGY4.0). EvoCK has been tested in two domains (the blocksworld and logistics), and very good experimental results have been obtained both in terms of the number of problems solved and of the quality of the control-rules produced.

1 Introduction

It is well known that planning for complex domains and non-trivial problems is a hard problem.[1] Although much research has gone into building improved planning systems, there is not yet what we call a perfect planner, that is, a planner that solves all the problems of any domain, including the complex ones, expanding only the nodes in the solution path and obtaining optimum plans with respect to different quality measures.

One way to improve planning in a given domain is to learn search control knowledge to guide the planning process. There have been different approaches to acquiring control knowledge for non-trivial (nonlinear) planning. Some of them use analogy [7, 17], others deduction [8, 12], induction [10], and some combine deduction and induction [2, 5].

Fully deductive methods have their weaknesses. For instance, they usually require a complete domain theory. Alternatively, inductive approaches incrementally acquire correct knowledge by observing a large set of problem solving examples. These approaches have strong bias built in their search operators. For instance, HAMLET [2][2] learned control knowledge has the property that, among many different possible generalizations, the most specific generalization is always selected. A learning paradigm with lighter bias might be desirable. Genetic Programming (GP) [9] is such a paradigm.

This article presents a GP based learning system named EvoCK (Evolution of control knowledge). Besides the GP algorithm, EvoCK uses information produced by HAMLET. The reason to supply this information is that GP is a kind of heuristic weak search, in the sense that it does not use background knowledge intensively. Although GP has been shown to perform well on its own [9], adding background knowledge (in this case, coming from HAMLET) to the process should improve its performance. Previous work [1] has shown that this is so.

We have tested our system in two domains (blocksworld and logistics). Our results show that EvoCK can generate control knowledge that performs better than the background knowledge it is supplied with by HAMLET.

The article is divided into six sections. Sections 2 briefly presents PRODIGY4.0 and HAMLET. Section 3 gives a short introduction to GP. Section 4 describes the main ideas about EvoCK. Section 5 shows the experimental results in the blocksworld domain and in a logistics transportation domain. Section 6 discusses those results.

2 PRODIGY4.0 and HAMLET

PRODIGY 4.0 [3] is a nonlinear problem solver. It follows a means-ends analysis backward chaining search procedure reasoning about multiple goals and multiple

[1] In fact, planning is intractable [4].

[2] A deductive-inductive learning system.

alternative operator relevant to the goals [16].[3] The inputs to the problem solver algorithm are:

- Domain theory, $\mathcal{D}$ (or, for short, domain), that includes the set of operators specifying the task knowledge and the object hierarchy;
- Problem, specified in terms of an initial configuration of the world (initial state, $\mathcal{S}$) and a set of goals to be achieved ($\mathcal{G}$); and
- Control knowledge, $\mathcal{C}$, described as a set of control rules, that guides the decision-making process.

The planning/reasoning cycle, involves several decision points, namely:

- to *apply* an operator whose preconditions are satisfied or continue *subgoaling* on an unsolved goal.
- the *goal* to select from the set of pending goals and subgoals;
- the *operator* to choose to achieve a particular goal;
- the *bindings* to choose to instantiate the chosen operator;

Default decisions at all these choices can be directed by explicit control knowledge. Figure 1 sketches the general decision search tree considered by PRODIGY. The decision cycle first encounters (selection of goal, operator, and bindings), followed by the same set of steps, or by applying an operator.

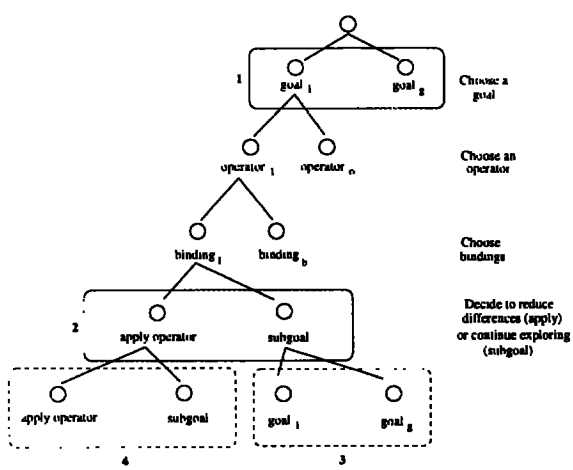

Figure1. Tree of decisions generated by PRODIGY when searching for a solution to a problem. Decisions 3 and 4 enclosed in dashed rectangles are of the same type as 1 and 2, respectively.

[3] PRODIGY4.0 is a successor of the previous linear planner, PRODIGY2.0 [11], and PRODIGY's first nonlinear planner, NOLIMIT [15]. We use the term "nonlinear" for a planner that can fully interleave subplans for different goals.

Although PRODIGY uses powerful domain independent heuristics [13] that guide the decision making process, it is still difficult and costly to characterize when these heuristics are going to succeed or fail. Therefore, learning is used for automatically acquiring control knowledge to *override the default behavior*, so that it guides the planner more efficiently to solutions of good quality.

HAMLET is integrated with the PRODIGY planner. The inputs to HAMLET are a task domain ($\mathcal{D}$), a set of training problems ($\mathcal{P}$), a quality measure (Q) and other learning-related parameters. The output is a set of control rules ($\mathcal{C}$). HAMLET has two main modules: the Bounded Explanation module, and the Refinement module. Figure 2 shows HAMLET's modules and their connection to PRODIGY.

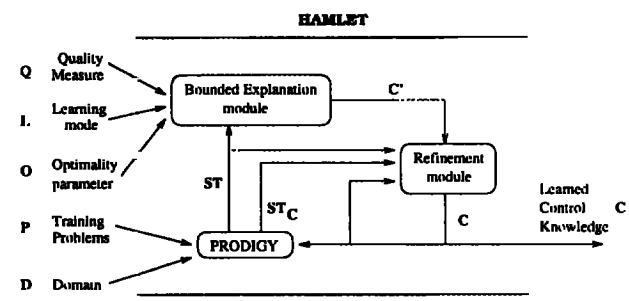

Figure2. HAMLET's high level architecture.

The Bounded Explanation module generates control rules from a PRODIGY search tree. These rules might be overly specific or overly general. The Refinement module solves the problem of being overly specific by generalizing rules when analyzing positive examples. It also replaces overly general rules with more specific ones when it finds situations in which the learned rules lead to wrong decisions. HAMLET gradually learns and refines control rules, in an attempt to converge to a concise set of correct control rules (i.e., rules that are individually neither overly general, nor overly specific). ST and ST_C are planning search trees generated by two calls to PRODIGY's planning algorithm, $\mathcal{C}$ is the set of control rules, and $\mathcal{C}'$ is the new set of control rules learned by the Bounded Explanation module.

HAMLET has been tested in different domains and it has been shown that it improves the efficiency of the base-level problem solver as well as the quality of the solutions [2].

3 Genetic Programming

Genetic Programming is an Evolutionary Computation (EC) method that has been used for program induction and machine learning [9]. GP transforms a population of individuals, having each one of them an associated worth or *fitness*, into a new population by using the Darwinian principle of survival and reproduction of the fittest. In order to create the new generation from the old one, a set of *genetic operators* are used, mainly *crossover* and *mutation*. Even though most of the terminology shows that GP is based on a biological analogy, it can be reformulated in terms of traditional AI Search. According to [14], GP can be seen as a kind of Beam Search.

GP consists of three main components: a population of individuals (or candidate solutions), a *fitness* measure (or heuristic function) and a set of genetic operators. A genetic operator is a function that takes one (or two) candidate solution and generates another candidate solution. Three genetic operators are generally used in GP: reproduction, mutation and crossover. Reproduction just copies an old candidate solution into a new one without changes. Mutation takes one candidate solution and generates another one by changing it slightly. crossover takes two candidate solutions and generates a new one by exchanging subtrees randomly chosen from the parents. A simple GP algorithm known as steady-state is described below[4]:

1. Create an initial population of random candidate solutions.
2. Assign a fitness value to each individual in the population using the fitness measure (or heuristic function).
3. Repeat until some termination criterion is satisfied:
   (a) Select a genetic operator G from the set of available operators (usually, reproduction, crossover or mutation). Each operator has a fixed probability of being selected.
   (b) Select one individual I_1 (or two, if the operator selected was crossover) from the population according to its fitness. Selection is stochastic. Better individuals will have a better chance of being selected.
   (c) Select one individual from the population to be replaced (I_3). Worse individuals will have a better chance of being selected.
   (d) Apply the genetic operator previously selected to I_1 (and I_2 if the operator selected was crossover) and replace I_3 with it.

[4] It must be noted that this is not the standard GP, as well known as generational GP, but it is the basic algorithm we have used in our work.

The main difference between GP and other EC techniques is that in GP, the structure of the individual that undergoes evolution is a parse tree. Parse trees allow to represent easily symbolic structures such as computer programs, induction trees or, like in our case, control knowledge. Other EC methods, such as genetic algorithms (GA) [6], would require to codify symbolic control knowledge as a bit string. Besides, standard GA genetic operators wouldn't respect the parse tree structure. This is the main reason GP has been chosen as the paradigm for this work.

4 EvoCK

EvoCK is a system based on the GP [9] paradigm with two differences: it has a wider set of genetic operators and it can use background knowledge previously obtained by HAMLET.

The main relations of EvoCK with the planner (PRODIGY) and HAMLET is shown in Figure 3. 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 that can be used by some of EvoCK genetic operators. For each problem, PRODIGY also generates a search tree, which is stored by the Search Monitor for later use in the second phase[5]. During the second phase (solid line in the figure), EvoCK evolves an individual for guiding PRODIGY in the search of a solution, using or not HAMLET outputs.

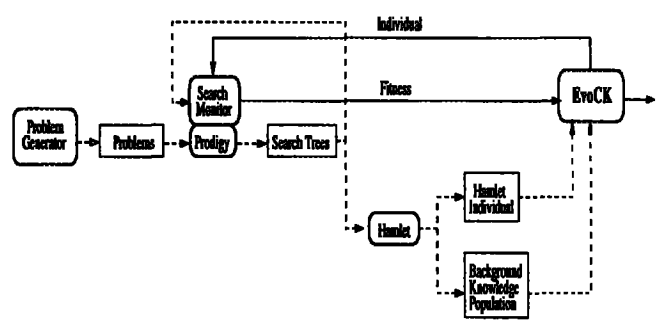

Figure3. Relations of EvoCK with HAMLET and PRODIGY.

[5] If comparing HAMLET and EvoCK is not the goal, different problems can be used by both methods. This is the case in this article

4.1 EvoCK individuals

EvoCK individuals are sets of PRODIGY control rules (specially represented to be manipulated by genetic operators). In this article we use only a subset of the control rules that can be written using PRODIGY4.0 control rule language. There are four kinds of control rules that EvoCK can learn. Each one corresponds to a decision point in the reasoning cycle of PRODIGY. Therefore, there are control rules for:

- decide either to apply an *operator* for achieving a *goal* or to subgoal on an unachieved *goal*;
- select an unachieved *goal*;
- select an *operator* to achieve some *goal*; and
- select bindings for an *operator* when trying to achieve a *goal*.

Initially, EvoCK creates a random population of control rules. Those initial random individuals are created according to a generation grammar that considers all possible operators, goals, and conditions (called meta-predicates) that can appear in the control rule language. Alternatively, EvoCK can use the set of control rules learned by HAMLET to seed the initial population.

An example of an individual generated by this grammar might be:

```
(list
    (rule (and (true-in-state (clear <object-1>))
               (some-candidate-goals
                    ((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>))))
```

In this example, there are two rules; one for selecting the next goal the planner will work on, and the other for selecting the bindings the operator `pick-up` will use (whatever is contained in variable `<object-1>` in this case). These rules can be read as:

```
if there is an object A with no objects on it and
  it is trying to achieve either and
  to have that object A on another object B or
     to hold another object
then it should work next in trying
  to put object A on object B

if there is an object on the table and
  it previously selected operator pick-up
then we should pick up that object
```

The antecedent of each rule is described as a conjunctive set of tests.[6]. Tests are expressed by means of a set of predicates such as:

- *true-in-state* (<expression>): determines whether an expression is true in the current state.
- *current-goal* (<expression>): tests whether *goal* is the current goal the planner is trying to achieve.
- *some-candidate-goals* (<list of goals>): tests whether any of the goals in *list-of-goals* is an unachieved goal.

4.2 EvoCK genetic operators

The system uses the standard GP operators (reproduction, crossover and mutation) and some others specially tailored for the learning task. All the operators have been extended to preserve correct syntax of the control knowledge. The whole operator set is:

- **Copy**: reproduction without modification.
- **Xover**: traditional crossover. It takes two individuals (parse trees) and exchanges two of their subtrees, taking care that correct syntax is preserved. Besides preserving correct syntax, this operator differs from traditional crossover in that it only produces one individual.
- **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).
- **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

[6] Some of the tests are, in fact, disjunctive, such as *some-candidate-goals*.

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 the 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).

4.3 Background knowledge

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

- Seed the initial population with an individual coming from HAMLET, instead of using a random generated population.
- The early phase of HAMLET returns a set of positive and negative examples as a subproduct. Positive examples are those where PRODIGY made the right decision. These positive examples can be easily transformed into control rules and then into GP individuals. As we recently saw, mutation operators are equivalent to crossover with a random individual. When using this kind of background knowledge, individuals coming from the positive examples population are used as the second parent, instead of using a random individual. For more details about this technique we refer to [1].

4.4 EvoCK fitness function

EvoCK fitness function is made of different components, each one measuring a different aspect of the individual. Those aspects are:

1. **Performance in planning problems** (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 is measured by counting how many steps of the best-quality[7] solution the individual manages to follow before stepping out of the solution path, and dividing it by the total number of steps in the solution. This can be done because PRODIGY expanded and stored previously the search tree for all the learning problems.
2. **Number of different variables** (to minimize): By minimizing the number of variables in each control rule, the different conditions in the left hand side will become linked to each other by common variables. We have seen empirically that this is desirable.
3. **Number of different true-in-state metapredicates** (to minimize): Minimizing this component makes the individual more general and more efficient (it takes time to evaluate each metapredicate).
4. **Number of different goals in some-candidate-goals metapredicates** (To maximize): this metapredicate 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 some-candidate-goals metapredicates** (to minimize): another way of making a rule more general is to get rid of unnecessary some-candidate-goals tests.
6. **Number of rules** (To minimize): by minimizing this component, the individual will become more general and more efficient.
7. **Individual size (in nodes)** (To minimize): this term takes into account simplicity of the individual not already accounted for by the other terms.

Selecting an individual in the population by means of this fitness function is achieved by means of tournament selection in the following way: a set of individuals is drawn randomly from the population and then each component of the fitness function is calculated for each individual. Then, components are compared one by one, starting from the performance in planning problems. If only one of the individuals is better than the rest, that individual is chosen for reproduction. If there are draws, the second component is used to decide, and so on. If by the final component there are still draws, one of the finalist is chosen randomly.

5 Experimental results

We conducted a series of experiments to test our system. Two different domains were used: the blocksworld

[7] In this paper, the quality criterion we use is the number of operators in the solution.

and logistics. EvoCK is a stochastic method, so it will usually produce different solutions every time it is run. For this paper, EvoCK has been run several times and only the best results will be commented upon. Considerations of the learning effort involved compared to other methods (like HAMLET) are left for future work.

We followed the following procedure:

1. HAMLET was trained with 400 randomly generated planning problems (in both the blocksworld and logistics).
2. A set of control-rules was obtained.
3. EvoCK was trained with 410 randomly generated planning problems in the blocksworld domain and with 452 in the logistics domain. They contained 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). Tournament size was 2. Two population sizes were used (2 and 50) in different experiments. Also, on some experiments background knowledge was used both to seed the initial population and for the knowledge-based crossover operators. In this paper, we only present the best individuals found with any configuration. In a previous paper, the reader can see a more detailed comparison of different experimental setups from a GP perspective and its results for the blocksworld only [1].
4. Several sets of control-rules were obtained.
5. Then a set of test problems (416 for the blocksworld and 357 for logistics) was randomly created and used to test PRODIGY alone, PRODIGY being guided by HAMLET control-rules and PRODIGY using control-rules generated by EvoCK. Those problems were much more harder than the ones used to train EvoCK, their purpose being to check whether EvoCK had generalized well.

Table 1 shows the results obtained by testing PRODIGY alone, the set of control-rules produced by HAMLET and a couple of the best individuals obtained by EvoCK during the experiments, for each domain (EvoCK-1 to EvoCK-4). Column **%P.Sol** tells the percentage of testing problems solved by control knowledge returned by each system. The rest of the columns show different syntactic aspects of the control rules obtained. **#Vars** informs of the number of variables in the rules, **#TIS** counts the number of tests performed on the current state by the control-rules, **#Goals** and **#SCG** tell the number of goals inside of *some-candidate-goals* metapredicates and the number of *some-candidate-goals* metapredicates themselves, respectively. Finally **#Rules** displays how many rules the set of control-rules has. Table 2 shows a breakdown for EvoCK individual EvoCK-1. And finally, so that the reader can see an individual evolved by EvoCK, table 3 shows EvoCK-2. Individuals for the logistics are much more complex than this one.

	% P.Sol	#Vars	#TIS	#Goals	#SCG	#Rules
Blocksworld						
PRODIGY	21%					
HAMLET	58%	39	25	28	12	12
EvoCK-1	81%	1	0	0	0	1
EvoCK-2	87%	6	3	0	0	3
Logistics						
PRODIGY	43%					
HAMLET	52%	401	164	118	56	56
EvoCK-3	60%	3	1	0	0	1
EvoCK-4	57%	198	79	59	32	34

Table1. Results for PRODIGY, HAMLET and EvoCK in both the blocksworld and logistics.

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

Table2. Solved-problems breakdown (percentage) for EvoCK-2 (87% problems solved)

6 Discussion

First, it is clear that PRODIGY using EvoCK generated control rules works better than PRODIGY alone in both domains, in terms of number of testing problems solved. Results are much better for the blocksworld (EvoCK-1) where we get a 60% differences between EvoCK and PRODIGY working alone. Differences in the logistics domain are smaller (EvoCK-3): 17% for EvoCK-PRODIGY, the reason being that logistics is a much more complex domain than the blocksworld in terms of learning, since there are more predicates and many more interactions between goals.

```
rule: if current-operator unstack
        true-in-state
          (on <object-2> <object-3>)
      then select-bindings
          (unstack-b <object-2> <object-3>)

rule: if current-goal
          (clear <object-2>)
        true-in-state
          (on <object-1> <object-2>)
      then select-operator unstack

rule: if current-goal
          (holding <object-2>)
        true-in-state
          (on <object-2> <object-3>)
      select-operator unstack
```

Table3. Individual EvoCK-2. 87% problems solved

Second, the objective of this paper is not to compare EvoCK with HAMLET, but to present an integrated method of a combined analytical and inductive learning technique with a genetic-based one. The results show that allowing a genetic evolution of the output generated by HAMLET greatly improves the number of solved problems, the differences being 29% and 8% for the blocksworld and logistics respectively. Table 2 shows a breakdown per number of goals and number of objects for individual EvoCK-2. It clearly shows that there is a very noticeable improvement on HAMLET results, even for very hard problems (50 goals and 50 objects).

Third, EvoCK obtains very slim and efficient control rules. For instance, in the blocksworld, HAMLET got a set of 12 control rules, whereas the best of EvoCK individuals (EvoCK-1) has only 3. Each of the rules is also simpler: HAMLET has 3.25 true-in-state predicates per rule on average whilst EvoCK-1 has only 2. Table 1 confirms this result for logistics as well.

References

1. 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.
2. 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.
3. Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. Prodigy: An integrated architecture for planning and learning. In K. VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990. Also Technical Report CMU-CS-89-189.
4. David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.
5. 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.
6. J.H. Holland. *Adaptation in Natural and Artificial Systems*. The University of MIchigan Press, 1975.
7. 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.
8. 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.
9. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.
10. 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.
11. Steven Minton, Craig A. Knoblock, Dan R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, 1989.
12. 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.
13. P. Stone, M. Veloso, and J. Blythe. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems.*, pages 164–169.
14. Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, 1994.
15. M. M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, 1989.
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.
17. 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.