Using genetic programming to learn and improve control knowledge

Ricardo Aler [*], Daniel Borrajo, Pedro Isasi

Escuela Politécnica Superior, Universidad 30, 28911 Leganés, Madrid, Spain

Received 28 March 2000; received in revised form 11 April 2001

Abstract

The purpose of this article is to present a multi-strategy approach to learn heuristics for planning. This multi-strategy system, called HAMLET-EVOCK, combines a learning algorithm specialized in planning (HAMLET) and a genetic programming (GP) based system (EVOCK: Evolution of Control Knowledge). Both systems are able to learn heuristics for planning on their own, but both of them have weaknesses. Based on previous experience and some experiments performed in this article, it is hypothesized that HAMLET handicaps are due to its example-driven operators and not having a way to evaluate the usefulness of its control knowledge. It is also hypothesized that even if HAMLET control knowledge is sometimes incorrect, it might be easily correctable. For this purpose, a GP-based stage is added, because of its complementary biases: GP genetic operators are not example-driven and it can use a fitness function to evaluate control knowledge. HAMLET and EVOCK are combined by seeding EVOCK initial population with HAMLET control knowledge. It is also useful for EVOCK to start from a knowledge-rich population instead of a random one. By adding the GP stage to HAMLET, the number of solved problems increases from 58% to 85% in the blocks world and from 50% to 87% in the logistics domain (0% to 38% and 0% to 42% for the hardest instances of problems considered).

Keywords: Speedup learning; Genetic programming; Multi-strategy learning; Planning

1. Introduction

Many interesting problems in Artificial Intelligence can be formulated in terms of search in a state space: optimization, planning, machine learning, etc. In order to explore

[*] Corresponding author
E-mail address: aler@inf uc3m es (R Aler)

efficiently such (usually huge) state-spaces, different Machine Learning (ML) techniques have been devised. Some of them learn global heuristics such as macro-operators [10,19, 32], “chunks” [37], or cases [13,14,41]. Other techniques learn local search heuristics (or control knowledge) [8,26].

All machine learning algorithms have biases that determine how they generalize to unseen instances [38]. Sometimes it is desirable that they possess other learning biases. However, it might be difficult to add those biases if the ML algorithm is very specialized. In this paper we intend to use Evolutionary Computation (EC) techniques to flexibly add those biases to an existing ML technique. EC includes methods such as Genetic Algorithms [12], Classifier Systems [12,34] and Genetic Programming (GP) [20]. EC algorithms explore concept spaces using different biases to other ML methods, as we will discuss throughout the paper. Also, some of the EC techniques learning biases can be declaratively stated in an evaluation function—the fitness function.

In this article we combine an ML and an EC technique to learn symbolic heuristics in a complex search domain: planning. In particular, we propose a two-stage loosely coupled multi-strategy system (HAMLET-EVOCK), that learns control knowledge for PRODIGY4.0, a means-ends bidirectional planner [39]. The first stage—HAMLET—is an incremental deductive-inductive multi-strategy system itself, built by Borrajo and Veloso [5]. The second stage (EVOCK: Evolution of Control Knowledge) is a GP based system whose purpose is to add corrective biases to HAMLET [1]. HAMLET-EVOCK is loosely coupled in the sense that the output of the first stage is just fed into the second stage, by seeding the GP initial population, instead of starting from a random one.

The goal of the HAMLET deductive subcomponent is to obtain approximately correct control knowledge by means of Explanation Based Learning techniques (EBL) [26], while the aim of the inductive subcomponent is to incrementally refine it. However, we have found out that HAMLET does not always generate more accurate control knowledge by observing more and more examples. We suspect that this is due to its example-driven operators (like AQs [24]). This means that the only way HAMLET can refine current control knowledge is by being presented the proper set of examples. Unfortunately, if the example-space is large, it could take HAMLET very long to generate correct control knowledge. Additionally, as HAMLET EBL subcomponent learns from search trees, preferably fully expanded, it has to be trained with simple planning problems. It might be the case that the examples needed to refine the control knowledge do not even exist in that subspace of planning problems.

Our most important guiding hypothesis for this paper is that even though HAMLET can produce incorrect control knowledge, it is only partially incorrect. This means that it could be used with profit by other ML technique which moves through the concept space using complementary biases (this is our second hypothesis). For this purpose, we use GP, whose learning operators (mutation and crossover) do not require examples to modify its candidate hypotheses and therefore it is not limited by significant examples being rare. Our GP based control knowledge learner, EVOCK, is able to use additional non-example-driven operators besides mutation and crossover. As usual, such operators are blind, and therefore EVOCK can also benefit from being combined with an example-driven system.

Also, we use EVOCK fitness function to measure some properties of the control knowledge that cannot be easily assessed by HAMLET. At its best, HAMLET tries to obtain control rules which are as general and correct as possible, but it is not concerned with rules being useful or simple. With respect to usefulness, an incremental system like HAMLET gives equal weight to all training problems whereas in some cases it might be desirable to ignore some of them if that helps to solve many other problems. This is easily achieved by EVOCK using a fitness function that counts the number of problems solved in a given time limit. With respect to simplicity, HAMLET tends to generate a lot of control rules. But having many control rules slows PRODIGY4.0 down. Again, it is very easy to include a term in the fitness function so that the complexity of the control rules is reduced. One could think that simply doing an utility analysis (similar to PRODIGY4.0-EBL [26]) would be enough for solving these problems. As we show with some experiments, in which rules are removed from the set of control rules according to the fitness function, this does not solve completely the problem.

This paper is structured as follows. Section 2 comments on learning heuristics for planning. Section 3 describes in detail HAMLET-EVOCK, our multi-strategy architecture. Empirical results showing that the multi-strategy approach improves on the mono-strategy one will be shown and analyzed in Section 4. This will be followed by Related Work in Section 5, Conclusions in Section 6 and Future Lines of Work in Section 7.

2. The learning task

Classical planning methods always involve search. Applying control knowledge at decision points in a search algorithm can yield far better performance than brute-force search. However, providing the "right" control knowledge is usually very hard for humans for they have to understand the way in which the planner works. The purpose of this article is to automatically learn and improve control knowledge for a classical planning system: PRODIGY4.0. PRODIGY4.0 searches a state-space bidirectionally, from an initial situation S towards the goals G, and vice versa [39]. PRODIGY4.0 uses an augmented STRIPS representation. We could have chosen any other planner for this study, given that all planners (even the most sophisticated ones such as Blackbox [18] or Graphplan [4]) involve some kind of search. We decided to use PRODIGY4.0 for its simplicity for introspection and for having a well integrated control knowledge module.

PRODIGY4.0 follows a four step decision cycle, as shown in Table 1. (1) First, the system must decide whether to apply the applicable planning operators to the current situation S (forward mode), or to work on a goal in G (backward mode). If in forward mode, the first currently applicable operator is applied.[1] If in backward mode, (2) an unachieved goal $g \in G$ must be selected. Then, (3) an operator O able to fulfill g must be chosen. And finally, (4) its unbound variables must be bound. Each one of these four decision points are backtracking points. PRODIGY4.0 allows to define control rules to help the planner to take the right alternative at those points and avoid backtracking. Control rules can select, reject or prefer alternatives at decision points. Fig. 1 shows an example of a control rule for the blocks world domain.

[1] In the latest versions of PRODIGY4 0, this is also a decision point, that determines which ground operator will be applied We do not consider it in this article, though

Table 1
PRODIGY4 0 four step decision cycle

```
Procedure Prodigy-base $(S, G)$

$S, G$ are the current situation and goals, respectively
$g$ is a goal
$O$ is an unground operator
$O_b$ is a ground operator
$b$ is a binding

• (1) Decide whether to work in forward or in backward mode:
    · If in forward mode, apply to $S$ one of the ground operators whose
      preconditions are true in $S$
    · If in backward mode:
          (2) Select a goal $g \in (G - S)$
          (3) Select an operator $O$ that can satisfy $g$
          (4) Select a binding $b$ for grounding $O$  $G \leftarrow G \cup \text{preconds}(O_b)$
```

```
(control-rule select-operators-unstack
     (if (and (current-goal (holding <b1>))
              (true-in-state (on <b1> <b2>))))
     (then select operator unstack))
```

Fig 1 Example of a control rule for selecting the unstack operator in the blocks world If `(holding <o1>)` is to be achieved, and it is currently true that block <b1> is on block <b2>, then the planner should select `UNSTACK` to achieve that goal

This control rule says that if PRODIGY4.0 is working on trying to hold an object, `<b1>`,[2] (this is a goal in G) and this object is on top of another one, `<b2>`, in the current state S, then PRODIGY4.0 should select the operator `UNSTACK` and reject the rest of operators that could achieve the same goal. In terms of search, this means that those successor nodes that are not related to the `UNSTACK` operator should be pruned.

3. HAMLET-EVOCK: A multi-strategy approach

HAMLET-EVOCK is a multi-strategy system made of two different learning systems: HAMLET and EVOCK. HAMLET is an incremental example-driven method for learning control rules for planning, and EVOCK is the module that implements the GP paradigm adapted for searching in the space of sets of control rules. The first two subsections describe HAMLET, GP and the advantages of combining such different paradigms in terms of complementary biases. Section 3.3 describes how they are actually combined. Finally, Sections 3.4, 3.5, and 3.6 explain in detail EVOCK learning biases.

[2] Variables are represented within < and >

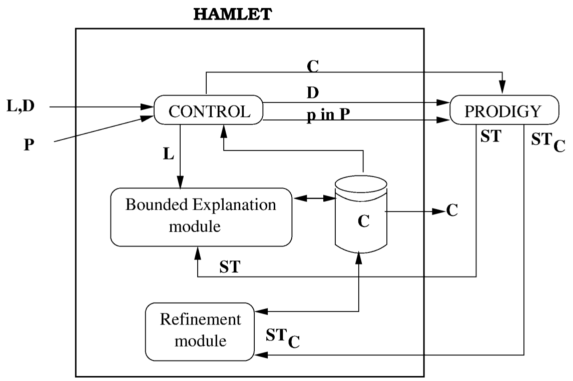


Fig 2 HAMLET high level architecture Inputs to HAMLET are: the domain description D, the set of training problems P, and other learning parameters L HAMLET outputs control knowledge C

3.1. HAMLET

HAMLET is an incremental learning method based on EBL [25] (Explanation Based Learning) and inductive refinement [5]. Fig. 2 shows HAMLET modules and their connection to PRODIGY4.0.

The inputs to HAMLET are a task domain description (D), a set of training problems (P), and other learning-related parameters (L).[3] The output is a set of control rules (C). HAMLET has two main modules: the Bounded Explanation module, and the Refinement module.

For each $p \in P$, HAMLET makes a first call to PRODIGY4.0 with D, p, and the control knowledge C learned in previous cycles (initially C is empty). The Bounded Explanation module analyzes the resulting search tree ST. From the right decisions, this module generates a new set of rules that are added to the previous set C. Then, the Refinement module tries to generalize rules that have the same right hand side. HAMLET calls PRODIGY4.0 again with the control rules learned so far C. After solving the same problem, PRODIGY4.0 generates the search tree ST_C. From the wrong decisions made by the rules, the Bounded Explanation module generates negative examples that are stored for future use by the Refinement module. Also, since the generalized rules might cover previously found negative examples, the Refinement module specializes them to avoid that situation. We refer to [39] for further details about HAMLET.

HAMLET has been tested in several domains: the standard blocks world and the logistics domain [5], some variations of them [3], and other domains such as a process planning domain, a work-flow domain, and a satellite control domain (results not published yet). HAMLET usually helped to improve the efficiency of the base-level problem solver as well as the quality of the solutions. However, HAMLET suffers from some weaknesses.

[3] HAMLET is also able to use a plan quality metric In this article we use the standard quality metric (minimizing the number of operators in the plan) It is not included in Fig 2 to avoid cluttering it

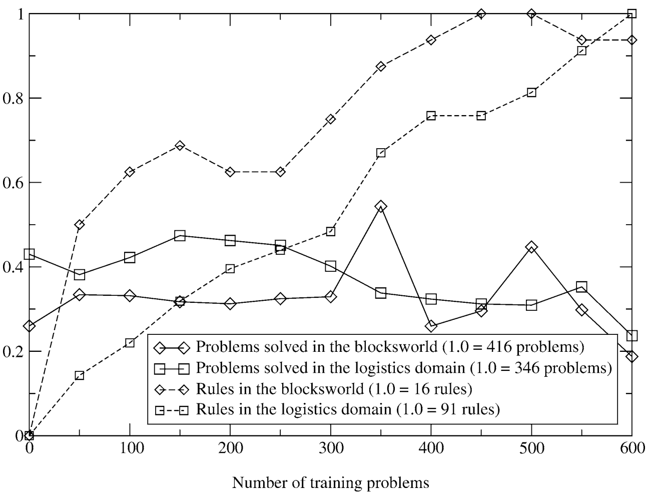


Fig 3 Learning curve for HAMLET in the blocks world and the logistics domain It also displays the evolution of the number of learned rules in both domains

In order to show them graphically, the following experiment was carried out. HAMLET was trained with 600 problems (with one or two goals) in two domains: the blocks world and the logistics domain. Every 50 problems, the rules obtained were tested with 416 and 346 testing problems (in the blocks world and the logistics domain, respectively). Testing problems are much harder (difficulty varies from 5 to 50 goals), to check generality. Fig. 3 displays the evolution of the proportion of testing problems solved and the number of rules learned in both domains.

Fig. 3 shows that, although HAMLET is able to learn control rules that improve accuracy, some training problems can decrease it. Also, even after having seen 600 problems, the best control rules returned by HAMLET are able to solve only 54% of the testing problems in the blocks world, and 47% in the logistics domain, within a reasonable time limit.[4]

We believe that the main reason behind this behavior is due to HAMLET example-driven operators. In order to refine an incorrect control rule, HAMLET assumes that eventually it will find an appropriate set of positive and negative examples. Example-driven operators seem a good heuristic, but given that the potential problem space is huge, the likelihood of finding the appropriate set of problems might be very small in some cases, and learned rules might remain partially correct for a long time. Furthermore, because HAMLET requires preferably fully expanded search trees, learning is limited to simple problems. However, the examples needed to refine the control rules might not even exist in the subspace of simple problems. Additionally, HAMLET—as any other incremental system—has no global picture of the usefulness of the learned control knowledge. It considers all training problems equally, whereas in some cases it might be desirable to ignore some of them if

[4] Here, we use the same time limit as in Section 4 1: $\frac{150}{16}(1 + \text{floor}(\frac{\#\text{goals}}{10}))$

that helps to solve many other problems. The previous analysis should not be considered a complete diagnostic, but a guiding hypothesis for this article.

Fig. 3 also displays the evolution of the number of learned rules. It shows that HAMLET has a strong tendency to increase the number of control rules (up to 16 rules in the blocks world and 91 in the logistics domain) even though this does not always improve the number of testing problems solved. The more rules there are, the longer it takes to evaluate them, so this should be avoided, if possible. Also, it can be observed that there is not a direct correlation of number of learned rules and performance decrease, as usually happens in utility analysis.

We hypothesize that although HAMLET control knowledge is sometimes incorrect, it is only partially incorrect, and could be corrected and improved by means of other learning system with complementary biases. Based on this assumption, we have chosen to combine HAMLET with GP because it possesses the appropriate biases. It is not example-driven and not incremental; and it can evaluate the usefulness and compactness of hypotheses. GP is described in the next section.

3.2. Genetic programming

GP is an Evolutionary Computation (EC) method that has been used for program induction and machine learning [20]. GP searches in the space of computer programs, trying to find a "good enough" computer program according to some metric. GP is a kind of Beam Search [36]. It maintains a set (or population) of many computer programs (or individuals), each one usually represented as a parse tree.

GP search is guided by a heuristic (or *fitness*) function. It measures how well a computer program performs, according to a set of user defined fitness cases. Standard GP uses three search genetic operators: reproduction, mutation, and crossover. Reproduction copies an individual without changes. Mutation replaces a subtree of the individual by another randomly generated one. Crossover swaps two randomly chosen subtrees from two different individuals, as shown in Fig. 4. Table 2 depicts the algorithm followed by steady-state GP, which is the variety used in this work.

One important aspect of the previous algorithm is the selection of individuals for reproduction and for replacement. The most common selection method chooses individuals with a probability proportional to their fitness. Another method—tournament selection—selects randomly a subset of individuals in the population and returns the best one (or the worst, in case of replacement).

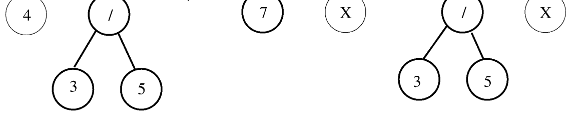


Fig 4 The crossover operator swaps two subtrees randomly chosen

Table 2
Steady state GP algorithm

```
Function steady-state GP ($AO$, $\vec{p}_{AO}$, $f$)

$AO$ contains the available genetic operators
$\vec{p}_{AO}$ is a vector containing the probabilities of selecting each operator
$f$ is the fitness function
$P$ is the genetic population
$G$ is a genetic operator
$I_1$, $I_2$, and $I_3$ are individuals

(1) Create $P$ containing random individuals
(2) Assign a fitness value to each individual in $P$ using $f$
(3) Repeat until some termination criterion is satisfied:
    (a) Select $G$ from $AO$ according to $\vec{p}_{AO}$
    (b) Select $I_1$ from $P$ according to its fitness  IF $G$ is crossover, select also $I_2$  Better individuals are
        more likely to be selected
    (c) Select $I_3$ from $P$ to be replaced  Worse individuals are more likely to be selected
    (d) Apply $G$ to $I_1$ (and to $I_2$ if the operator selected was crossover) and replace $I_3$ with it
```

In contrast to HAMLET, GP follows a generate-and-test approach to modify hypotheses (mutation and crossover), instead of being example-driven. GP is not incremental either. Also, the fitness function can evaluate individuals for usefulness, compactness, etc. On the other hand, EVOCK learning operators do not contain as much knowledge about learning in the planning domain as those of HAMLET. Therefore, GP might also benefit from the combination. Next section explains how this is achieved.

3.3. HAMLET-EVOCK

This section explains how HAMLET and EVOCK are combined in the multi-strategy system HAMLET-EVOCK. Fig. 5 displays its basic architecture. HAMLET, PRODIGY4.0, and the problem generator are preexisting modules.

Training HAMLET-EVOCK occurs in two stages. First, HAMLET learns a set of control rules from a training set randomly generated by the problem generator. Then, they are used to seed EVOCK initial population. EVOCK tries to evolve HAMLET set of control rules by using a second training set (also generated by the problem generator). PRODIGY4.0 is used differently by the two learning systems. HAMLET sends the training problems to PRODIGY4.0, which solves them, and sends back the search trees, from which HAMLET can learn, as explained in Section 3.1. EVOCK uses PRODIGY4.0 to evaluate its individuals (each individual is a set of control rules). For that purpose, individuals are loaded into PRODIGY4.0 and then [PRODIGY4.0 + individual] is run on the training problems (or fitness cases). Performance data such as whether the problems were solved or not, or the time required to solve them, is returned to EVOCK. This is used to determine the worth (fitness) of the individual to guide selection. EVOCK can also learn starting from a random population, as in standard GP. When EVOCK finishes, it returns its best individual.

The next three subsections describe EVOCK by explaining its learning biases according to Utgoff [38] (language, exploration, and evaluation biases).

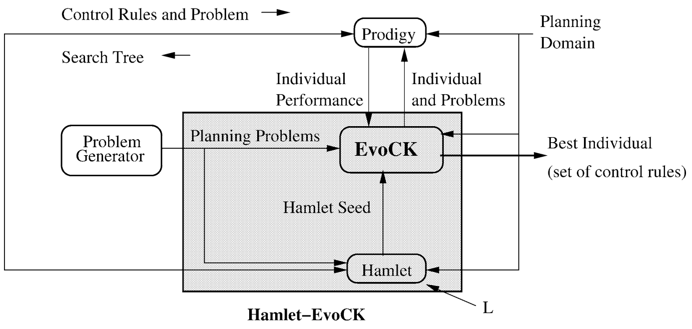


Fig 5 Architecture of HAMLET-EVOCK First, HAMLET learns control rules from problems generated by the problem generator (L are HAMLET learning parameters) HAMLET rules are used to seed EVOCK initial population Then, EVOCK tries to improve them by using new training problems PRODIGY4 0 is used by EVOCK to evaluate individuals

3.4. The language bias

Any learning system is biased by the language that is used to represent candidate hypotheses and examples. In the case of GP, individuals represent the hypotheses and in our case, we decided that each individual represents a set of control rules. Each of the control rules has two parts, the left hand side (LHS) and the right hand side (RHS). The LHS contains a list of conditions that need to be true for the rule to be applied. The RHS specifies the decision to be made. Each condition calls a meta-predicate, which are functions that have access to PRODIGY4.0 internal state. The meta-predicates used in this paper are:

- `true-in-state`(`fact`): it tests if `fact` is true in the current planning situation S.
- `current-goal`(`goal`): it checks if `goal` is the current goal the planner is working on.
- `target-goal`(`goal`): it tests whether `goal` is one of the pending goals in G.
- `some-candidate-goals`(`goal1`, `goal2`, ...): it is equivalent to: `target-goal`(`goal1`) $\vee$ `target-goal`(`goal2`) $\vee \ldots$.

Usually, in GP there are no constraints on the structure to evolve: any combination of functions and terminals is valid. This is called *operational closure* [20]. However, in EVOCK case, PRODIGY4.0 constrains the syntax of the control rules. Also, meta-predicates must be supplied with the right type of arguments. And more importantly, it is a waste of search time to create expressions that are known not to be valid in the first place. Therefore EVOCK individuals are constrained to PRODIGY4.0-valid ones (in the literature, such structures are called *constrained structures* [20] or *strongly typed structures* [27]). Actually, EVOCK individuals are constrained by the same sublanguage than HAMLET (`select`, `apply`, and `sub-goal` rules, and the same meta-predicates). In order to achieve this, only valid individuals must be created in the initial population. Also, applying a genetic operator to valid individuals must produce valid individuals.

Table 3
Domain independent grammar for generating syntactically correct sets of control rules LIST-ROOT-T is the axiom of the grammar

LIST-ROOT-T	$\rightarrow$	(`list` RULE-T) | (`list` RULE-T RULE-T) |
RULE-T	$\rightarrow$	(`rule` AND-T ACTION-T)
AND-T	$\rightarrow$	(`and` METAPRED-T) | (`and` METAPRED-T METAPRED-T) |
METAPRED-T	$\rightarrow$	(`true-in-state` GOAL-T) | (`target-goal` GOAL-T) | (`current-goal` GOAL-T) | (`some-candidate-goals` LIST-OF-GOALS-T)
LIST-OF-GOALS-T	$\rightarrow$	(`list-goal` GOAL-T) | (`list-goal` GOAL-T GOAL-T) |
ACTION-T	$\rightarrow$	(`select-goal` GOAL-T) | (`select-operator` OP-T) | (`select-bindings` BINDINGS-T) | `sub-goal` | `apply`

Table 4
Domain dependent grammar for generating syntactically correct sets of control rules in the logistics domain EvoCK is able to automatically generate this grammar from the domain description

OP-T	$\rightarrow$	`load-truck` | `load-airplane` | `unload-truck` | `unload-airplane` | `drive-truck` | `fly-airplane`
BINDINGS-T	$\rightarrow$	(`load-truck-b` OBJECT-T TRUCK-T LOCATION-T) | (`load-airplane-b` OBJECT-T AIRPLANE-T AIRPORT-T) | (`unload-truck-b` OBJECT-T TRUCK-T LOCATION-T) | (`unload-airplane-b` OBJECT-T AIRPLANE-T AIRPORT-T) | (`drive-truck` TRUCK-T LOCATION-T LOCATION-T) | (`fly-airplane` AIRPLANE-T AIRPORT-T AIRPORT-T)
GOAL-T	$\rightarrow$	(`at-truck` TRUCK-T LOCATION-T) | (`at-obj` OBJECT-T LOCATION-T) | (`inside-truck` OBJECT-T TRUCK-T) | (`inside-airplane` OBJECT-T AIRPLANE-T)

The creation of valid individuals is achieved by using two special-purpose production grammars: domain independent (shown in Table 3) and domain dependent (shown in Table 4). This second grammar is generated on-the-fly by EvoCK for any domain described in PRODIGY4.0 Description Language [6].[5] The grammar shown in Table 4 is for the logistics domain, a well known planning domain described in [40]. In this domain, packages have to be delivered to several locations using trucks (inside a city) or airplanes (between cities).

Terminal symbols in the grammars are displayed in lowercase and non-terminal generative symbols are shown in uppercase. This grammar has production rules that follow the structure $A \rightarrow (F_a A_1 A_2 \ldots)|(F_b B_1 B_2 \ldots)| \ldots$, where F_a is a terminal and $A_i, B_i, \ldots$ are either terminals or non-terminals. They generate lisp sentences such as $(F_a A_1 A_2 \ldots)$. A whole individual is generated by starting with the `LIST-ROOT-T` symbol, and randomly applying the production rules until an individual is completed. As it can be seen, each individual is a list of control rules.

[5] Some non-terminal symbols, like `OBJECT-T`, do not appear in the grammar They only generate terminal symbols like `<object-1>`, `<object-2>`, , which are control rule variables This is also true for `TRUCK-T`, `CARRIER-T`, `AIRPLANE-T`, `LOCATION-T`, `AIRPORT-T`, and `POST-OFFICE-T`

Besides being able to generate valid individuals for the initial population, EVOCK genetic operators need to transform valid individuals into valid individuals. For mutation, the process is similar to creating a whole individual. First, the mutation point is randomly selected (as in standard mutation). Then, EVOCK determines which non-terminal symbol generated the element at the mutation point. This is achieved by checking the right hand side of the productions, and comparing them with the element at the mutation point. For instance, if the element at the mutation point was `and`, then its non-terminal generative symbol would be `AND-T`, as the third production of the grammar in Table 3 indicates. Finally, it replaces the subtree at the mutation point by a randomly grown one by applying the production rules from the non-terminal symbol (`AND-T`, in this case). This mechanism enforces that the new subtree is of the same type than the old one, which is enough in this case to assure correctness.

Constrained crossover works likewise. First, a crossover point is selected randomly in the mother individual. Then, the non-terminal symbol that generated it is determined. Next, a crossover point that was generated with the same symbol is randomly selected in the father individual. Finally, both subtrees are swapped, as in standard crossover, to produce the offspring.

3.5. The exploration bias

The exploration bias deals with the way in which the learning algorithm explores the search space. EVOCK uses the standard GP operators and some others specially tailored for this learning task. All of them use a grammar to always produce correct individuals, as explained in Section 3.4. Besides constrained mutation and crossover, EVOCK uses the following operators:

- *Adding Crossover*: it takes a rule (`RULE-T`) or a goal (`GOAL-T`) from an individual and adds them to another individual at `LIST-ROOT-T` and `LIST-OF-GOALS-T` points, respectively. This operator adds subtrees to individuals, whereas standard crossover replaces them.
- *Growing mutation*: it adds a randomly generated rule (`RULE-T`) or goal (`GOAL-T`) at points generated by `LIST-ROOT-T` and `LIST-OF-GOALS-T`, respectively. It is equivalent to *adding crossover* with a randomly generated individual.
- *Chopping off mutation*: a randomly selected rule or goal is pruned from the individual.
- *Join*: it selects one variable in the control rule (like `<object-1>`) and substitutes it by any other variable in the control rule that already exists. The rationale behind this operator is that sometimes there are conditions in a rule that are not related with other conditions by common variables. This is usually undesirable. For instance, in the blocks world, if there is a control rule to pick-up an object `<obj1>` when some other conditions are true, our experience says that many of those other conditions should refer to `<obj1>` as well. The *join* operator is a simple way of creating these references.

EVOCK can also seed its initial population to focus the search, as explained in Section 3.3.

3.6. 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. This fitness function must reflect the goal of the system, which is to generate an individual that solves as many problems as possible (limited by a timeout or node limit)[6] and faster than PRODIGY4.0. To achieve this goal, a hierarchical fitness function has been devised [1]. A hierarchical function contains several components to be maximized (or minimized). Instead of using a weighted addition of the components to compare different individuals, a hierarchical function compares them using the first component. If there are draws, it uses the second component, and so on. There are other alternatives for multi-criteria optimization such as Pareto (see for instance [21]). We selected hierarchical functions because it is a simpler approach that does not require a big population to store the whole Pareto front. The components of EVOCK hierarchical fitness function are as follows:

(1) *Performance in training problems*: to maximize. It contains three sub-components, ordered as follows:
   (a) *Number of problems solved by the individual expanding strictly fewer nodes than* PRODIGY4.0: to maximize.
   (b) *Number of problems solved by the individual*: to maximize.
   (c) *Total number of nodes expanded by the individual*: to minimize. This is used instead of time because measured time varies slightly even when the system is run in similar conditions. This would make the process non-deterministic and experiments unrepeatable.

(2) *Number of different variables*: to minimize. It relates to the same bias as the *join* operator. It is desired to have meta-predicates in the LHS inter-related by common variables.

(3) *Generality/Compactness*: to maximize. Generality and compactness is encouraged by minimizing the number of restrictions in the LHS of the control rules (i.e., the number of `true-in-state` and `some-candidate-goals` (SCG) meta-predicates) and by maximizing the number of arguments of `SCG` (`SCG` is equivalent to a disjunction of meta-predicates. Therefore, the more arguments, the more general a control rule is). Further compactness is achieved by minimizing the number of control rules and the size in nodes of the individual.

Component (1)(a) deserves a longer explanation. Training problems are necessarily easy; otherwise the time limit for fitness evaluation would be too large. However, PRODIGY4.0 is able to solve easy problems without heuristics. Therefore, a good strategy for individuals to score well in the "number of problems solved" component (1)(b) is to do nothing at all and let the planner do all the work. This is very easy to achieve, for instance, by means of control rules with conditions that never match. On the other hand, individuals that do modify the behavior of the planner are likely to be only partially

[6] The node limit is $4N$, where N is the number of nodes required by PRODIGY4 0 to solve the problem without backtracking

correct, and to solve fewer problems than the planner without any heuristics. Therefore, individuals that do nothing will be selected and those that modify the behavior of the planner will not, even though many of the latter might be corrected in the long term. To change this undesirable behavior, we added a first component that forced individuals to do something positive: solving problems using strictly fewer nodes than PRODIGY4.0 (component (1)(a)). Individuals that do nothing will score 0 in this component and will be selected against. However, there is a risk: there can be some individuals that obtain a good score in this component but that solve fewer problems (they score poorly in the second component). This might happen, but in the long term it is expected that, once the first component cannot be improved further, the second component will take the lead and begin to be improved itself.

4. Empirical results

The aim of this section is to describe how HAMLET-EVOCK has been tested. We will first describe how the experiments were carried out. Then, we will show the results and comment on them.

4.1. Experimental setup

First, 400 problems in two planning domains (blocks world and logistics domain) were randomly generated to train HAMLET and obtain the HAMLET seed. Then 192/188 new training problems from the blocks world and the logistics domains, respectively, were randomly generated and used by EVOCK to evolve HAMLET seed. The training sets contain planning problems with 1 and 2 goals and from 2 to 5 objects (blocks in the blocks world and packages in the logistics domain). As GP is stochastic, EVOCK was run 50 times using the same HAMLET seed, but with a different random seed for the random number generator (i.e., 50 GP-runs). Finally, the best set of control rules from each run was evaluated with 416/346 testing problems (blocks world/logistics domain) randomly generated by the same problem generator. Those problems are much more difficult than the ones used for learning, so that scalability of the rules can be checked. The time limit for testing a problem is calculated with the formula $t_{test} = \frac{150}{16}(1 + \text{floor}(\frac{\#\text{goals}}{10}))$ seconds, where #goals is the number of goals in the testing planning problem. Testing was carried out on a 400 MHz Pentium II with 256 MB RAM. This experimental sequence corresponds to the thick line of Fig. 6 and will be referred to as HAMLET-EVOCK.

Obtaining good results from the multi-strategy system is not enough. It is necessary to know what would have happened if all the training problems would have been given to HAMLET, instead of dividing them between the two components of HAMLET-EVOCK. If HAMLET alone produces better results than HAMLET-EVOCK, then the multi-strategy system is obviously not solving HAMLET deficiencies. Therefore, after learning with the first training set, HAMLET was fed with the second one. From them, HAMLET produced a new set of control rules, which was subsequently tested. This second experimental sequence is represented by the thin line of Fig. 6 and it will be referred to as HAMLET-HAMLET because it uses HAMLET in sequence.

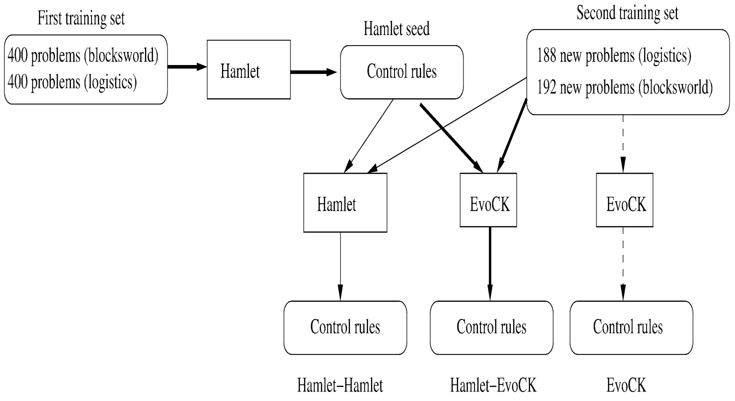


Fig 6 Description of the three different experimental configurations: multi-strategy HAMLET-EVOCK vs monolithic HAMLET-HAMLET and stand-alone EVOCK

Although it is not the focus of this article, another experimental configuration was carried out to compare the effects of seeding EVOCK initial population (that is, HAMLET-EVOCK) with EVOCK starting from a random population, as in standard GP. This third experimental setup (dashed line in Fig. 6) will be referred to as EVOCK, because EVOCK is used as a stand-alone learning system. In both EVOCK and HAMLET-EVOCK, two different configurations were tried, with population sizes of 2 and 300, respectively (the respective tournament set sizes were 2 and 5. No crossover was used in the 2-population). However, the greedy configuration (2-population) significantly outperformed the more exploratory one in both domains.[7] Therefore, in this paper we only report the results for the greedy configuration, so as not to clutter the results section. Both EVOCK and HAMLET-EVOCK were run for a maximum of 100000 evaluations (an evaluation is a call to PRODIGY4.0 to evaluate an individual with a single training problem).

4.2. Results

In this section, results of the three experimental configurations will be discussed and compared. The GP runs for the stand-alone configuration EVOCK and the multi-strategy configuration HAMLET-EVOCK are summarized in Fig. 7. This figure is a cumulative frequency graph, that shows the frequency (y-axis) with which an experimental configuration yields a set of control rules that is able to solve a given percentage of problems or more (x-axis). This can be interpreted as an estimation of the probability an experimental configuration will produce an individual that solves, at least, a given percentage of problems. Results for PRODIGY4.0 and HAMLET seed are also provided for comparison purposes (the dashed and continuous vertical lines in Fig. 7). Given that

[7] It might be possible that larger populations (i e , more exploratory configurations) would give better results if given many more evaluations However, results obtained with 2 populations were so good that we did not try more computationally expensive experiments

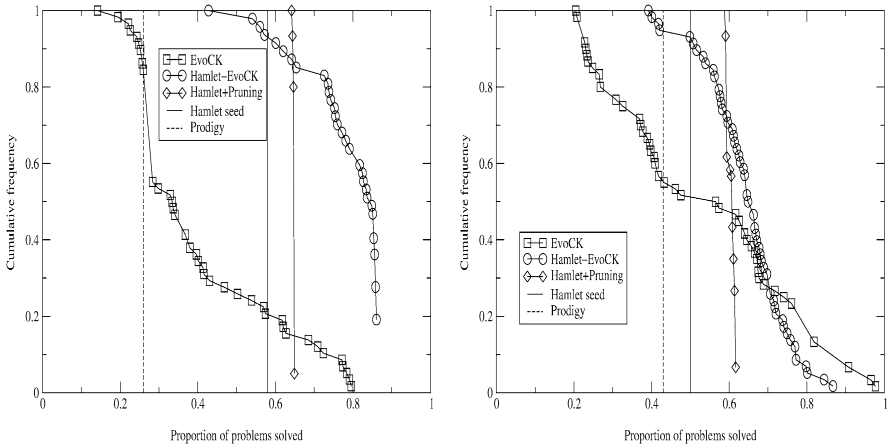


Fig 7 Frequency of GP runs (y-axis) that are able to solve a proportion x of problems (x-axis) or more Results for the blocks world are on the left and for the logistics domain on the right

these two systems are not stochastic, they are represented by vertical lines. The purpose of these results is to visualize the effects of seeding EVOCK initial population with HAMLET seed.

We have used approximate randomization to determine whether the differences between EVOCK and HAMLET-EVOCK shown in Fig. 7 are statistically significant [7]. In particular, for each frequency of solved problems (x-axis of graphics in Fig. 7) we calculated the probability that the differences between the cumulative frequencies of EVOCK and HAMLET-EVOCK are due to chance, at the 5% level. Results are analyzed as follows:

- In the blocks world, the cumulative probability is always greater for HAMLET-EVOCK than for EVOCK (and this is statistically significant). This means that, for instance, it will take HAMLET-EVOCK fewer runs to find an individual that solves 40% of the problems. Also, it is clear that HAMLET-EVOCK improves over the seed supplied by HAMLET, as 90% of the runs do better than HAMLET alone.
- This happens in the logistics domain as well: 93% of the GP-runs improve over HAMLET seed. However, the analysis of the cumulative frequency diagram is not so clear-cut as in the blocks world case. In the range (0%, 62%), HAMLET-EVOCK is able to find good individuals more frequently than EVOCK (and this is statistically significant). However, there are no significant differences in the range (62%, 100%). In fact, the best individuals are found by EVOCK. This might be due to chance, or to the fact that the HAMLET seed in the logistics domain is very large (56 rules), whereas EVOCK best individual contains only 3 rules. Therefore, it might be very time consuming for HAMLET-EVOCK to correct the HAMLET seed. In fact, it only manages to obtain a 21 rule individual. In the blocks world, the HAMLET seed is only 12 rules long, not very far away from the 5 rule best individual created by stand-alone EVOCK.

It could be thought that a simpler integration of HAMLET and GP would perform equivalently. After all, HAMLET has a way of creating rules, but no way of evaluating which ones to keep. EVOCK has both: constrained genetic operators for the former and the fitness function for the latter. What would happen if EVOCK fitness function was used to remove non-useful HAMLET seed rules?. It is possible that HAMLET is after all a better control rule generator, if only its control rules were filtered somehow, like the utility analysis carried out in [26]. In order to test this simpler approach, we have tried another configuration of the multi-strategy system where EVOCK can either prune rules or add rules that were already present in the HAMLET seed (no other genetic operators are allowed).[8] Addition of rules provides a sort of backtracking mechanism, to test whether a control rule that had already been removed has some good effects once other rules have been removed. The effects of HAMLET-EVOCK with the restricted set of genetic operators have been labeled 'HAMLET + pruning' in Fig. 7. In both domains, the cumulative curve is almost vertical, which means that in most runs HAMLET-EVOCK got an equivalent individual. This restricted multi-strategy configuration is able to improve HAMLET seed. If only the best individuals are taken into account (which are not very different from the worst ones), the HAMLET seed is improved by 7% in the blocks world and by 12% in the logistics domain (see also Table 5). These results allow to draw three important conclusions:

- The HAMLET seed contains quite good control rules and therefore, it makes sense to use them as seed for HAMLET-EVOCK.
- The HAMLET seed contains incorrect control rules (once removed, performance increases). In fact, in the logistics domain, the main source of power is pruning. The best individual from HAMLET + pruning solves 62% of the testing problems, and after that value, differences between EVOCK and HAMLET-EVOCK do not differ significantly (see Fig. 7).
- However, mere pruning does not get as good results as full-fledged HAMLET-EVOCK. The "creative" aspects of GP are required as well (mutation, crossover, etc.).

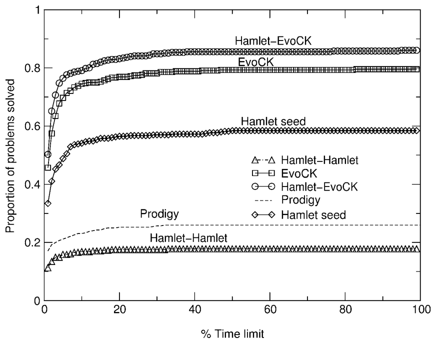
In summary, HAMLET-EVOCK is able to improve significantly HAMLET seed and to find good individuals (although not always the best ones) more frequently than EVOCK alone. But it is interesting to know whether HAMLET alone scores better when given all the training examples (i.e., the HAMLET-HAMLET configuration). The learning curve in Section 3.1 (Fig. 3) already hints that this will not necessarily improve HAMLET control rules.

Table 5 shows that when HAMLET tries to refine and improve a set of control rules previously learned by itself (HAMLET seed in Table 5), the percentage of test problems solved actually drops: in the blocks world it falls from 58 to 18%, in the logistics domain it gets from 50 to 47%. This is a confirmation of the oscillatory behavior displayed in Section 3.1 (Fig. 3). On the other hand, HAMLET-EVOCK improves the set of control rules given as seed for the initial population: 58–85% in the blocks world and 50–87% in the logistics domain. Therefore, it seems that the multi-strategy system does better than the

[8] Actually, it was one of the reviewers who suggested this experiment

Table 5
Results for PRODIGY4 0, HAMLET seed, HAMLET-HAMLET, EVOCK (random seed), HAMLET-EVOCK, and HAMLET + pruning in the blocks world and logistics domains

	Blocks world		Logistics domain	
	% Problems solved	Number of rules	% Problems solved	Number of rules
PRODIGY4 0	26%		43%	
HAMLET seed	58%	12	50%	56
HAMLET-HAMLET	18%	13	47%	64
EVOCK	80%	3	98%	3
HAMLET-EVOCK (best ind)	85%	5	87%	21
HAMLET+Pruning (best ind)	65%	5	62%	16



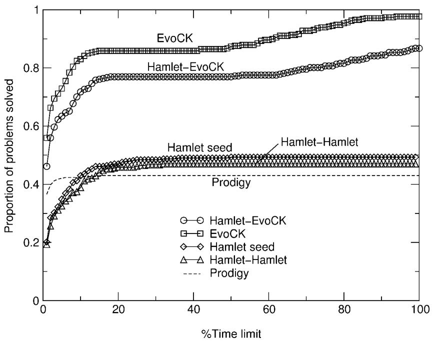


Fig 8 Proportion of problems solved when varying the testing time limit from 0 to 100% Results for the blocks world are on the left and for the logistics domain on the right

mono-strategy one. It is also noticeable that HAMLET-EVOCK produces individuals with fewer control rules than the seeding individual (12 to 5 control rules in the blocks world and 56 to 21 in the logistics domain) hence returning more efficient individuals. On the other hand, HAMLET-HAMLET always increases the number of control rules: from 12 to 13 in the blocks world and from 56 to 64 in the logistics domain. Thus, HAMLET-EVOCK provides the sought compactness bias to HAMLET.

Segre et al. [33] critique an arbitrary choice of a time limit for testing. Fig. 8 displays how the proportion of problems solved increases when the allocated testing time goes from 0 to 100% of the maximum testing time limit t_{test}. If we had observed that the HAMLET seed or HAMLET-HAMLET solved more and more problems as the allocated time increased, and this did not happen for HAMLET-EVOCK, it should have been concluded that HAMLET would eventually outperform the multi-strategy system. However, Fig. 8 shows that this is not the case. Therefore, the reason of HAMLET-HAMLET drop in performance is not that it generates clever but slow to evaluate rules, but that they are not very correct rules.

In order to show that the set of control rules learned are general and useful for more complex problems, a breakdown of the results is displayed in Tables 6 and 7. Problem

Table 6
Breakdown of the number of testing problems solved in the blocks world by HAMLET and EVOCK according to the number of goals and objects

# Goals-# Objects	PRODIGY4 0	HAMLET seed	HAMLET-HAMLET	HAMLET-EVOCK	EVOCK
50-50 (4%)	0%	0%	0%	38%	31%
20-50 (11%)	8%	33%	4%	77%	71%
20-20 (11%)	10%	29%	4%	73%	65%
10-50 (11%)	21%	60%	19%	94%	83%
10-20 (11%)	25%	56%	15%	85%	75%
10-15 (11%)	40%	48%	15%	85%	77%
05-50 (10%)	18%	70%	2%	92%	88%
05-20 (10%)	20%	90%	18%	98%	92%
05-15 (10%)	42%	88%	38%	98%	95%
05-10 (10%)	65%	88%	60%	95%	95%

Table 7
Breakdown of the number of testing problems solved in logistics domain by HAMLET and EVOCK according to the number of goals and objects

# Goals-# Objects	PRODIGY4 0	HAMLET seed	HAMLET-HAMLET	HAMLET-EVOCK	EVOCK
50-50 (14%)	0%	0%	0%	42%	77%
20-50 (9%)	3%	0%	0%	87%	97%
20-20 (8%)	7%	7%	0%	90%	100%
10-50 (4%)	13%	0%	0%	80%	100%
10-20 (4%)	20%	53%	33%	80%	100%
10-15 (4%)	20%	67%	40%	87%	100%
10-10 (4%)	7%	67%	40%	80%	100%
05-50 (3%)	42%	0%	0%	83%	100%
05-20 (3%)	58%	75%	67%	83%	100%
05-15 (3%)	42%	42%	58%	67%	100%
05-10 (3%)	25%	58%	67%	67%	100%
05-05 (3%)	33%	83%	92%	100%	100%
02-50 (3%)	90%	40%	40%	90%	100%

complexity is measured by the number of goals and objects in the planning problem. This is shown in the first column of Tables 6 and 7 (the quantity in parentheses represents the percentage of problems of that difficulty level there are in the testing set). Usually, the more goals and objects a problem has, the more difficult it is for the planner to solve it. The last assertion can be easily checked in Table 6, where results for PRODIGY4.0 are supplied. For instance, PRODIGY4.0 solves none of the 50-50 problems. HAMLET-EVOCK improves drastically with respect to the initial seed (HAMLET seed) by solving very hard problems (50 goals and 50 objects). The percentage of testing problems solved for PRODIGY4.0 working alone and HAMLET-HAMLET results are also shown.[9]

[9] In the logistics domain, results for the 2-20, 2-15, 2-10, 2-5, 2-2, 1-50, 1-20, 1-15, 1-10, 1-5, and 1-2 types are not shown because they are not very interesting They comprise about 33% of the testing problems EVOCK solves all of them and the other configurations solve most of them (percentages are always greater than 80%, and very often greater than 90%)

4.3. Learning curves

In this section, we study experimentally how the performance of EVOCK and HAMLET-EVOCK changes when three important parameters are modified: the number of evaluations, the number of training examples, and the quality of the HAMLET seed. Due to GP being rather time expensive, these learning curves contain only three points (three cumulative frequency curves, actually) corresponding to three different parameter values. One of the three points represents the cumulative curve obtained from the default parameters that was already discussed in Fig. 7 of Section 4.2. The other two have been generated from 30 GP-runs each. This is true for all the graphs of this section. It is not easy to quantify whether one cumulative frequency curve is "better" than another. Here, we have decided to carry out an approximate randomization test like in Section 4.2. One curve is considered better than another if it has a significantly (0.05 level) higher probability of obtaining an individual that solves at least $x\%$ of the testing problems. x is the percentage of problems solved by HAMLET + pruning (this has been discussed previously in Table 5). $x = 0.65$ for the blocks world and $x = 0.62$ for the logistics domain. We have considered this a reasonably non-arbitrary way of focusing on the interesting part of the distributions (their upper tail).

Figs. 9(a) and (b) show how the cumulative frequency graphs for EVOCK and HAMLET-EVOCK change when the number of evaluations increases from 33333 to 100000. Differences for EVOCK are not significant, which means that EVOCK had already converged at 33333 evaluations, and that no significant over-fitting occurs by allowing more evaluations. In the case of HAMLET-EVOCK, there are significant differences for the (66666, 33333) curves. Probably, HAMLET-EVOCK needs enough evaluations to correct the often complex HAMLET seed. Fig. 10 displays the same information for the logistics domain. In this case, EVOCK benefits from increasing the number of evaluations ((66666, 33333) are significantly different), although it seems to be converging at 100000 evaluations (no significant differences for the (100000, 66666) curves), as shown in Fig. 10(a). HAMLET-EVOCK also improves as evaluations increase, as shown in Fig. 10(b)

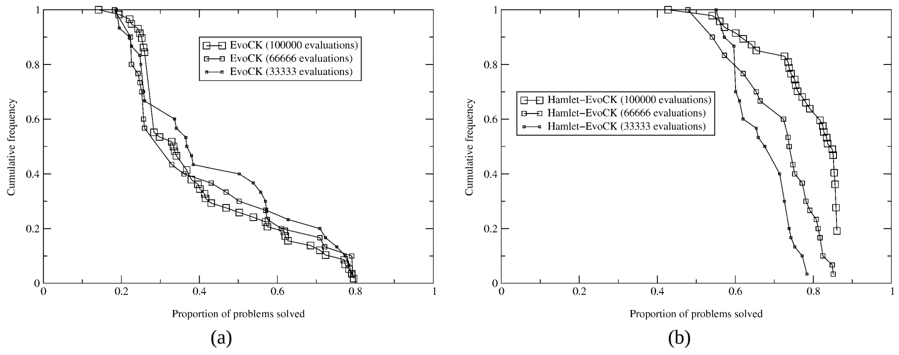
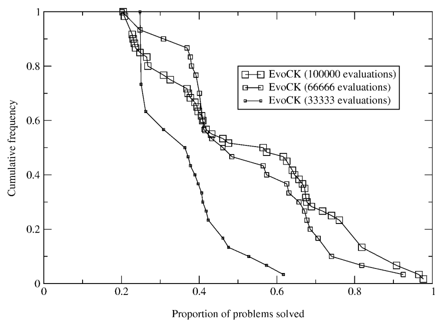
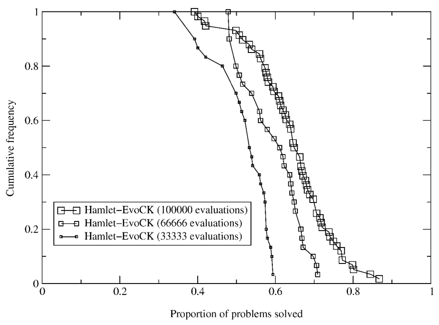


Fig 9 Cumulative frequency graphs of EVOCK (a) and HAMLET-EVOCK (b) for 33333, 66666, and 100000 evaluations, respectively (blocks world domain)

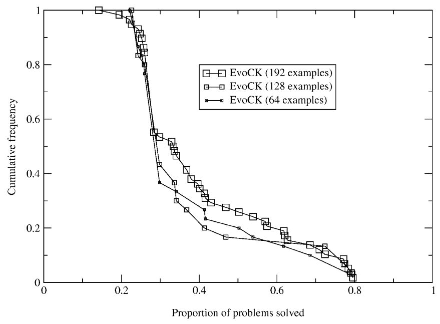


(a)

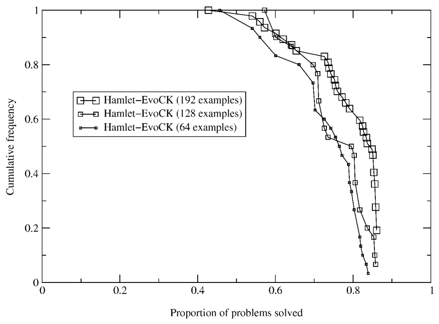


(b)

Fig 10 Cumulative frequency graphs of EVOCK (a) and HAMLET-EVOCK (b) for 33333, 66666, and 100000 evaluations, respectively (logistics domain)



(a)



(b)

Fig 11 Cumulative frequency graphs of EVOCK (a) and HAMLET-EVOCK (b) for 64, 128, and 192 examples, respectively (blocks world domain)

(significant differences for (66666, 33333)). The HAMLET seed in the logistics domain is even larger than in the blocks world, so this result makes sense.

Figs. 11 and 12 display the cumulative frequency graphs when the number of training problems changes from 1/3 of the original training data, to 2/3, and 3/3, in the blocks world and the logistics domain, respectively. The number of evaluations has been reduced proportionally to keep evaluations per problem constant. Otherwise, decreasing the number of training problems to be processed for each individual would increase the number of GP generations and therefore, the number of applications of the genetic operators. In that case, it would be uncertain whether performance is different because fewer training problems were used, or because genetic operators were applied more times. Fig. 11(a) shows that EVOCK gives about the same result for 1/3, 2/3, and 3/3 of the examples. Indeed, approximate randomization shows that these differences are not significant. This means that EVOCK had already converged with only 1/3 of the training examples. This is the case also for HAMLET-EVOCK (see Fig. 11(b)). In the logistics domain, EVOCK

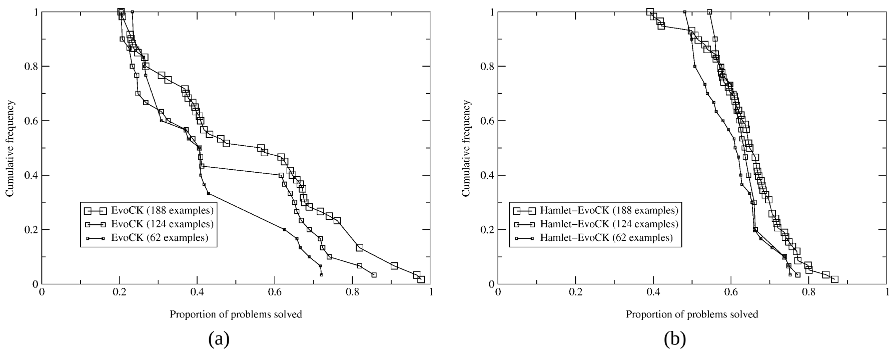


Fig 12 Cumulative frequency graphs of EVOCK (a) and HAMLET-EVOCK (b) for 62, 124, and 188 examples, respectively (logistics domain)

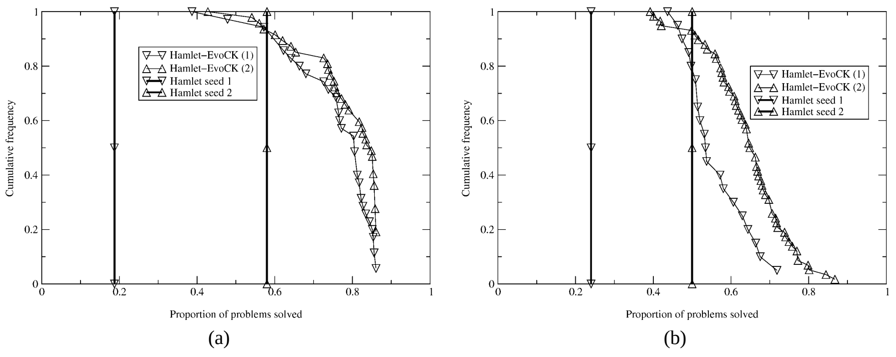


Fig 13 Cumulative frequency graphs of HAMLET-EVOCK starting with two different seeds in the blocks world (a) and the logistics domain (b), respectively HAMLET-EVOCK (1) is the result of seeding EVOCK with HAMLET seed 1 Likewise for HAMLET-EVOCK (2)

seems to improve as the number of examples increases, as Fig. 12(a) shows (differences are significant for the (124, 62) curves). HAMLET-EVOCK has almost already converged with only 1/3 of the training data (Fig. 12(b)). Differences are significant only for the (188, 62) curves.

It is also interesting to observe how HAMLET-EVOCK depends on the number of problems solved by the HAMLET seed. Figs. 13(a) and (b) display the cumulative frequencies of HAMLET-EVOCK having been seeded with two different seeds, in the blocks world and the logistics domain, respectively. HAMLET seed 2 refers to the seed that has been used so far. HAMLET seed 1 refers to the worst HAMLET set of control rules obtained in Fig. 3. In the blocks world, HAMLET-EVOCK obtains a similar result with both seeds (no significant differences), although HAMLET seed 1 is much worse than HAMLET seed 2. In the logistics domain, HAMLET-EVOCK results look worse for the bad seed (differences are significant). However, they are not as bad as HAMLET seed 1 might suggest. It seems that HAMLET-EVOCK is sensible to the quality of the seed, although not strongly so. This piece of evidence confirms one of the assumptions of this article: even though HAMLET can generate quite inaccurate rules, they are only partially incorrect and can be corrected by GP. It is also reasonable that HAMLET-EVOCK finds more difficult to correct the HAMLET seeds of the logistics domain, because they are bigger than in the blocks world.

4.4. Plan quality considerations

It has been shown that HAMLET-EVOCK is able to improve PRODIGY4.0 and HAMLET solvability horizons. However, although HAMLET learns from best quality (shortest) solutions,[10] EVOCK does not have a definite bias towards that end. Therefore, it would be interesting to know whether HAMLET-EVOCK and EVOCK take advantage of a tradeoff between solving more problems and plan quality. We also want to know whether HAMLET seed is able to bias HAMLET-EVOCK towards finding control rules that improve plan quality.

Table 8 tries to answer that question. Columns labeled as “better”, “equal”, and “worse” give the number and percentage of testing problems where the quality of HAMLET-EVOCK plans is better, equal or worse, respectively, to those of PRODIGY4.0, HAMLET seed, and EVOCK. Only those problems that can be solved by both HAMLET-EVOCK and the other system to be compared with are considered. Finally $\frac{\bar{q}_{H-E}-\bar{q}_B}{\bar{q}_B}$% displays the average improvement in quality of HAMLET-EVOCK with respect to the other three configurations ($B =$ PRODIGY4.0, HAMLET seed, or EVOCK). $\bar{q}_{H-E}$ is the average quality of those problems solved by both HAMLET-EVOCK and system B. $\bar{q}_B$ is the corresponding value for system B. Negative values for the latter column indicate HAMLET-EVOCK is better than B and positive values that it is worse, in terms of plan quality. Column “Range” displays the minimum and maximum average quality improvement values along with the type of problems where such value was reached.

- HAMLET-EVOCK vs. PRODIGY4.0: in the blocks world, HAMLET-EVOCK does not improve the quality with respect to PRODIGY4.0 (no plan is shorter than those of PRODIGY4.0), but it does in the logistics domain: 23% of the problems have a shorter solution than PRODIGY4.0, and only 7% of them are longer. Solutions are 2% shorter on average. For problems with 10 goals and 15 objects, the reduction is remarkable: 22% (see column “Range” in the same Table 8). Therefore, in this case HAMLET-EVOCK is not taking advantage of a solvability/quality tradeoff.
- HAMLET-EVOCK vs. HAMLET seed: in the blocks world, HAMLET-EVOCK provides longer solutions than the initial seed in 14% of the plans, and only 4% of them are shorter. On average, solutions are 3% longer. This effect is not so acute in the logistics domain, but there is no improvement on HAMLET seed in terms of quality (0%).

[10] Plan quality can be measured in different ways: plan execution time, economic cost, etc [31] Here, we use a typical quality metric, which is the number of planning operators in the plan (i e , the length of the plan; the shorter, the better)

Table 8
Comparison of HAMLET-EVOCKwith PRODIGY4 0, HAMLET seed, and EVOCK in terms of plan quality

	Better	Equal	Worse	$\frac{\bar{q}_{H-E}-\bar{q}_B}{q_B}$%	Range
			HAMLET-EVOCK vs PRODIGY4 0		
Blocks world	0 (0%)	106 (98%)	2 (2%)	0%	All $<$ 1%
Log domain	173 (23%)	519 (70%)	52 (7%)	-2%	-22% (10-15) to 0% (1-15)
			HAMLET-EVOCK vs HAMLET seed		
Blocks world	9 (4%)	193 (82%)	32 (14%)	+3%	-2% (5-10) to +12% (20-20)
Log domain	27 (16%)	128 (74%)	18 (10%)	0%	-6% (20-20) to +5% (5-20)
			HAMLET-EVOCK vs EVOCK		
Blocks world	7 (2%)	305 (93%)	15 (5%)	0%	0% (5-50) to +1% (50-50)
Log domain	102 (34%)	172 (57%)	15 (9%)	-2%	-11% (2-10) to +4% (10-50)

Therefore, it seems that HAMLET-EVOCK is able to solve more problems than the HAMLET seed (see previous results of Fig. 7) by sometimes increasing plan length.

- HAMLET-EVOCK vs. EVOCK: it seems that seeding EVOCK has no noticeable effect in the blocks world (0% average improvement). However, it does in the logistics domain: solutions are 2% shorter and 34% of the problems have a shorter solution with HAMLET-EVOCK than its non-seeded counterpart, and only 9% of the problems have a longer solution. Therefore, although in the logistics domain the individual that solves more problems was found by EVOCK, HAMLET-EVOCK manages to give shorter solutions. The multi-strategy configuration shows his worth again.

In summary, HAMLET-EVOCK can sometimes increase the number of problems solved by decreasing plan quality, but this is not necessarily so. Also, the HAMLET seed bias HAMLET-EVOCK conveniently with respect to plan quality.

5. Related work

There have been different approaches to acquire control knowledge for non-trivial (non-linear) planning. Some of them use analogy [15,41], others pure deduction (EBL) [16, 17,26], pure induction [23], and some combine deduction (EBL) and induction like HAMLET [5] and SCOPE [8,9] (EBL + ILP). However, they do not use genetic search as a component for the multi-strategy system. In particular, Estlin offers many results that could be used for comparison purposes with our approach [8]. Results in the logistics domain are particularly well suited. Unfortunately, they use a different base planner (UCPOP [30] is a partial order planner, whereas PRODIGY4.0 is a total order planner) and it is not clear how problems containing many packages (logistics domain) are generated. Different machines to ours were also used. However, a rough comparison can be done by comparing results for the hardest problems. Estlin reports that SCOPE solves 24% of the 50-goal 50-package problems in the logistics domain when given a time limit of 500 seconds on a Ultra Enterprise 5000. SCOPE requires 165 seconds on average per problem solved. On the other hand, EVOCK and HAMLET-EVOCK are able to solve 77 and 42% of the 50-goal 50-package problems, respectively, when testing time is limited to about 56 seconds (in a Pentium II 400 MHz). Notice also that 50-50 problems are very hard for the base planner PRODIGY4.0 (it solves none of them within the time limit as shown in Table 6). This is also true for UCPOP. All in all, it seems that HAMLET-EVOCK is doing better than SCOPE in this domain. It would be interesting to see if SCOPE benefits from a GP stage add-on, in the same way than HAMLET does.

Some innovative approaches to planning use genetic programming [20]. This approach was started by Koza [20], who evolved a planner that solved a very specific set of problems in the blocks world domain. Handley [11] used GP to evolve plans for specific problems in the blocks world domain. Muslea [28] 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 [35] also analyzed these two basic approaches for GP-planning. The main difference with our approach is that they used GP to search in the space of plans or planners, and we are exploring the space of heuristics for a planner. We believe this is a better approach, because searching in the space of plans can not be very efficient when large problems are considered. And searching in the space of full-blown planners seems like a daunting task. On the other hand, searching for just the heuristics is a more feasible task and once they have been found, they reduce the amount of search needed to solve future planning problems.

In the Genetic Algorithm field, there are two main approaches named Pittsburg [34] and Michigan [12] that have been applied for learning rules. In the Michigan approach, each individual is a rule. Therefore, it is not well suited for disjunctive concepts (consisting of several of these rules), except if additional techniques like 'niching' and 'species formation' are used [22]. Also, a method for extracting the solution from the population must be used (like the Universal Suffrage operator in [29]). In the Pittsburgh approach, each individual is a set of rules. Therefore each individual can represent a whole disjunctive concept. Its main disadvantage is that it requires more space. On the other hand, the search method is just a simple Genetic Algorithm. Our work follows the Pittsburgh approach but we have used the symbolic variable length structures of GP.

In this work, we use a grammar both to generate the initial random population and to check individuals for syntactic correctness. Whigham [42] also uses grammars, although the representation is different. His individuals are the actual parse trees, whose non-leave nodes contain the generative terms of the grammar. That is, only the leaves of the tree represent the actual individual. The rest of the nodes are the grammar non-terminal nodes used to generate the individual. This has the advantage that in order to cross two individuals, the system only has to check that both crossing points come from the same non-terminal, whereas in EVOCK case, nodes have to be looked up in the grammar. But Whigham's approach requires to build the actual individual before each evaluation whereas EVOCK does not need to. Another difference with his work is that EVOCK grammar is built on-the-fly for each different planning domain.

6. Conclusions

In this article, we have described a multi-strategy approach, HAMLET-EVOCK, that learns control rules for a planning system (PRODIGY4.0). HAMLET is able to learn control knowledge on its own. However, HAMLET generates too many control rules, which do not always generalize well. This led us to combine it with EVOCK (a GP-based system), to overcome HAMLET problems. Our work is based in three assumptions:

- HAMLET produces incorrect control knowledge mainly because of two of its biases:
  - Its example-driven operators, which require the proper examples to refine the rules and generate correct control knowledge. Those examples might be very rare because of the size of the example-space, or might not even exist in the problem subspace used for learning.
  - Its incremental nature does not permit a global picture of the usefulness of the acquired control knowledge. HAMLET considers equally all training problems, whereas in some cases it would be useful to ignore some of them so that many others can be solved.
- HAMLET incorrect control knowledge is only partially incorrect (i.e., it can be corrected).
- HAMLET incorrect control knowledge can be corrected and improved by combining it with another ML system with complementary biases (this is the multi-strategy assumption).

GP possesses the right biases, according to our previous hypotheses:

- Contrary to HAMLET, EVOCK genetic operators are not example-driven. Mutation and crossover do not depend on actual examples to change and correct individuals. Of course, in some cases it could be more difficult for EVOCK to find the right mutation than for HAMLET to get the right example, but this is not the case of the domains tested in this article.
- GP can easily evaluate usefulness and compactness of control knowledge by means of its fitness function.

HAMLET and EVOCK have been coupled in a simple fashion: a set of control rules generated by HAMLET are used to seed EVOCK initial population, which in turn, will apply its own biases to the HAMLET seed. HAMLET-EVOCK has been tested in two planning domains: the blocks world and the logistics domain. The success of the system supports our guiding hypotheses. Empirical results show that HAMLET benefits from the multi-strategy approach. EVOCK also benefits from it because it generates good individuals more frequently when seeded than by starting from a random population. However, in the logistics domain, HAMLET-EVOCK is unable to produce control rules better than the best ones generated by stand-alone EVOCK, although the results are not significantly worse either. A plausible reason, supported by several experiments, is that in this domain, the best individuals are small, and the sets of control rules generated by HAMLET tend to be very large. Therefore, it takes a lot of effort to correct the HAMLET seed, and it is just as profitable to start from scratch. This might also be true of other domains.

7. Future lines of work

- HAMLET and EVOCK biases could be integrated more tightly. For instance, HAMLET could use directly the genetic operators. Or the other way around, EVOCK could take advantage of HAMLET learning operators.
- Learning from simple problems (that allow to expand completely PRODIGY4.0 search tree) has proven to be a good assumption in the domains tested in this article. Other domains might require more complex examples to learn. In that case, partially expanded trees could be used for learning. In an extreme case, both HAMLET and EVOCK could learn from a single solution path (i.e., no search tree, only the solution path) by using what we have called white box fitness functions (there are some preliminary results in [2]). Solution paths can be generated directly, more efficiently, without calling the base planer. However, this is likely to have other effects on the quality and speed of learning, which we would have to study.
- We believe that learning control knowledge to guide the planner towards good quality solutions (not only in terms of solution length) would not be very difficult to add to our multi-strategy approach (although finding good quality solutions is much more difficult than just solving a problem).
- EVOCK flexibility could be used further. Currently, EVOCK explores the space of select-heuristics, but it could be very easily changed to explore the space of prefer or reject heuristics. Likewise, considering negated left hand side conditions would be very easy. All that is needed is to use another language bias by changing the generating grammar. Many other machine learning (including HAMLET) would require substantial modifications to make these changes.

Acknowledgements

We thank the reviewers for their very useful comments. The suggested additional experiments helped considerably to clarify HAMLET handicaps and HAMLET-EVOCK behavior.

References

[1] R Aler, D Borrajo, P Isasi, Genetic programming and deductive-inductive learning: A multistrategy approach, in: J Shavlik (Ed), Proc 15th International Conference on Machine Learning, ICML'98, Madison, WI, 1998, pp 10–18

[2] R Aler, D Borrajo, P Isasi, GP fitness functions to evolve heuristics for planning, in: M Middendorf (Ed), Evolutionary Methods for AI Planning, Las Vegas, NV, 2000, pp 189–195

[3] R Aler, D Borrajo, P Isasi, Knowledge representation issues in control knowledge learning, in: Proc 17th International Conference on Machine Learning, Morgan Kaufmann, San Francisco, CA, 2000, pp 1–8

[4] A L Blum, M L Furst, Fast planning through planning graph analysis, in: C S Mellish (Ed), Proc IJCAI-95, Montreal, Quebec, Vol 2, Morgan Kaufmann, San Mateo, CA, 1995, pp 1636–1642

[5] D Borrajo, M Veloso, Lazy incremental learning of control knowledge for efficiently obtaining quality plans, Artificial Intelligence Rev (Special Issue on Lazy Learning) 11 (1–5) (1997) 371–405

[6] J G Carbonell, J Blythe, O Etzioni, Y Gil, R Joseph, D Kahn, C Knoblock, S Minton, A Pérez, S Reilly, M M Veloso, X Wang, PRODIGY4 0: The manual and tutorial, Technical Report CMU-CS-92-150, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992
[7] P R Cohen, Empirical Methods for Artificial Intelligence, MIT Press, Cambridge, MA, 1995
[8] T A Estlin, Using multi-strategy learning to improve planning efficiency and quality, PhD Thesis, Department of Computer Sciences, University of Texas at Austin, TX, 1998
[9] T A Estlin, R J Mooney, Learning to improve both efficiency and quality of planning, in: Proc IJCAI-97, Nagoya, Japan, Morgan Kaufmann, San Mateo, CA, 1997, pp 1227–1233
[10] R E Fikes, P E Hart, N J Nilsson, Learning and executing generalized robot plans, Artificial Intelligence 3 (4) (1972) 251–288
[11] S G Handley, The automatic generations of plans for a mobile robot via genetic programming with automatically defined functions, in: K E Kinnear Jr (Ed), Advances in Genetic Programming, MIT Press, Cambridge, MA, 1994, pp 391–407, Chapter 18
[12] J H Holland, Adaptation in Natural and Artificial Systems, The University of Michigan Press, Ann Arbor, MI, 1975
[13] L Ihrig, S Kambhampati, Design and implementation of a replay framework based on a partial order planner, in: Proc AAAI-96, Portland, OR, 1996, pp 849–854
[14] L Ihrig, S Kambhampati, Storing and indexing plan derivations through explanation-based analysis of retrieval failures, J Artificial Intelligence Res 7 (1997) 161–198
[15] S 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
[16] S Kambhampati, Improving Graphplan’s search with EBL and DDB techniques, in: Proc IJCAI-99, Stockholm, Sweden, Vol 2, 1999
[17] S Katukam, S Kambhampati, Learning explanation-based search control rules for partial order planning, in: Proc AAAI-94, Seattle, WA, 1994, pp 582–587
[18] H Kautz, B Selman, Unifying SAT-based and graph-based planning, in: J Minker (Ed), Workshop on Logic-Based Artificial Intelligence, Washington, DC, Computer Science Department, University of Maryland, College Park, MD, 1999, pp 14–16
[19] R E Korf, Macro-operators: A weak method for learning, Artificial Intelligence 26 (1985) 35–77
[20] J R Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, 1992
[21] W B Langdon, Pareto, population partitioning, price and genetic programming, Research Note RN/95/29, University College London, Gower Street, London, 1995
[22] P Langley, Elements of Machine Learning, Morgan Kaufmann, San Mateo, CA, 1996
[23] C Leckie, I Zukerman, Learning search control rules for planning: An inductive approach, in: Proc 8th International Workshop on Machine Learning, Evanston, IL, Morgan Kaufmann, San Mateo, CA, 1991, pp 422–426
[24] R S Michalski, A theory and methodology of inductive learning, Artificial Intelligence 20 (1983) 111–161
[25] S Minton, J G Carbonell, C A Knoblock, D R Kuokka, O Etzioni, Y Gil, Explanation-based learning: Optimizing problem solving performance through experience, Artificial Intelligence 40 (1989) 63–118
[26] S Minton, M Zweben, Learning, planning and scheduling: An overview, in: S Minton (Ed), Machine Learning Methods for Planning, Morgan Kaufmann, San Mateo, CA, 1993, Chapter 8
[27] D J Montana, Strongly typed genetic programming, Evolutionary Computation 3 (2) (1995) 199–230
[28] I Muslea, SINERGY: A linear planner based on genetic programming, in: S Steel (Ed), Recent Advances in AI Planning 4th European Conference on Planning, ECP’97, Toulouse, France, in: Lecture Notes in Artificial Intelligence, Vol 1348, Springer, Berlin, 1997, pp 312–324
[29] F Neri, First order logic concept learning by means of a distributed genetic algorithm, PhD Thesis, Dipartimento di Informática, Universitá di Torino, Torino, Italy, 1997
[30] J S Penberthy, D S Weld, UCPOP: A sound, complete, partial order planner for ADL, in: Proc KR-92, Cambridge, MA, 1992, pp 103–114
[31] M A Pérez, J G Carbonell, Control knowledge to improve plan quality, in: Proc 2nd International Conference on AI Planning Systems, Chicago, IL, 1994, pp 323–328

[32] B.W. Porter, D.F. Kibler, Experimental goal regression: A method for learning problem-solving heuristics, Machine Learning 1 (3) (1986) 249–285.

[33] A. Segre, C. Elkan, A. Russell, Technical note: A critical look at experimental evaluations of EBL, Machine Learning 6 (2) (1991) 183–195.

[34] S. Smith, Flexible learning of problem solving heuristics through adaptive search, in: Proc. IJCAI-83, Karlsruhe, Germany, 1983, pp. 422–425.

[35] L. Spector, Genetic programming and AI planning systems, in: Proc. AAAI-94, Seattle, WA, AAAI Press/MIT Press, 1994.

[36] W.A. Tackett, Recombination, selection, and the genetic construction of Computer Programs, PhD Thesis, University of Southern California, Department of Electrical Engineering Systems, 1994.

[37] M. Tambe, A. Newell, P.S. Rosenbloom, The problem of expensive chunks and its solution by restricting expressiveness, Machine Learning 5 (3) (1990) 299–348.

[38] P. Utgoff, Shift of bias for inductive concept learning, in: Machine Learning: An Artificial Intelligence Approach, Vol. II, Morgan Kaufmann, Los Altos, CA, 1986, pp. 107–148.

[39] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, J. Blythe, Integrating planning and learning: The PRODIGY architecture, J. Experiment. Theoret. Artificial Intelligence 7 (1995) 81–120.

[40] M.M. Veloso, Learning by analogical reasoning in general problem solving, PhD Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.

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

[42] P.A. Whigham, Evolving a program defined by a formal grammar, in: Proc. 4th International Conference on Neural Information Processing—The Annual Conference of the Asian Pacific Neural Network Assembly (ICONIP'97), Dunedin, New Zealand, 1997, pp. 456–459.