Learning to Solve Planning Problems Efficiently by Means of Genetic Programming

Ricardo Aler aler@inf.uc3m.es
Department of Computer Science, Universidad Carlos III de Madrid, 28911 Leganés, Madrid, Spain

Daniel Borrajo dborrajo@ia.uc3m.es
Department of Computer Science, Universidad Carlos III de Madrid, 28911 Leganés, Madrid, Spain

Pedro Isasi dborrajo@ia.uc3m.es
Department of Computer Science, Universidad Carlos III de Madrid, 28911 Leganés, Madrid, Spain

---

Abstract

Declarative problem solving, such as planning, poses interesting challenges for Genetic Programming (GP). There have been recent attempts to apply GP to planning that fit two approaches: (a) using GP to search in plan space or (b) to evolve a planner. In this article, we propose to evolve only the heuristics to make a particular planner more efficient. This approach is more feasible than (b) because it does not have to build a planner from scratch but can take advantage of already existing planning systems. It is also more efficient than (a) because once the heuristics have been evolved, they can be used to solve a whole class of different planning problems in a planning domain, instead of running GP for every new planning problem. Empirical results show that our approach (EVOCK) is able to evolve heuristics in two planning domains (the blocks world and the logistics domain) that improve PRODIGY4.0 performance. Additionally, we experiment with a new genetic operator – *Instance-Based Crossover* – that is able to use traces of the base planner as raw genetic material to be injected into the evolving population.

Keywords

Genetic planning, genetic programming, evolving heuristics, planning, search.

1 Introduction

AI Planners aim to achieve a set of goals, starting from an initial state, by using operators that represent the available actions of a task domain. Traditional approaches use domain-independent planners for generating plans (Bonet and Geffner, 1999; Blum and Furst, 1995; Penberthy and Weld, 1992; Veloso et al., 1995). Some recent approaches to planning use Genetic Programming (GP). The genetic planning approach was started by Koza, who evolved a planner that solved a very specific set of problems in the blocks world domain (Koza, 1989, 1992). The ways GP can be applied to planning can be summarized as follows:[1]

[1] We follow a classification proposed by Spector (1994). We have added another item to the classification where our own work is included.

- To evolve a plan. In this context, a plan can be seen as a program that changes the initial state of a planning problem (given as input) into the desired or goal state. Considered as a program, it can be evolved by GP.

- To evolve a planning program for a particular domain. The planner should be, in principle, able to solve a set or a pre-defined subset of the problems in the domain. Taking this idea to the extreme, a truly domain-independent planner could be evolved, although this would require a daunting computer effort and seems currently unfeasible.

- To evolve heuristics to improve the efficiency of an already existing planner, which is able to solve planning problems on its own, but in a very inefficient way.

We will now analyze each alternative in depth.

1.1 Evolving Plans

Handley (1994) used GP to evolve plans for a specific subset of problems in the blocks world domain. Muslea (1997) generalized, extended, and formalized this idea for his SINERGY system and showed how any STRIPS-like planning problem could be translated into an equivalent GP problem. He tested it successfully in several domains offering better performance than UCPOP (Penberthy and Weld, 1992) in difficult problems. Westerberg and Levine (2000) followed a similar approach and also reported good results. An important advantage of this approach is its flexibility: planning operators need not be coded using the STRIPS formalism; they can be arbitrary state-transforming programs. This is also its main drawback, as they cannot use the underlying logical representation of the operators to reason about the world. For instance, the fitness function relies on a continuous measure of closeness between the goals and the current state of the world. If no such measure can be specified, the fitness function will be of little help to GP. This is the case of the "switch goal" mentioned by Handley (a switch can be on or off, but not "close to on").

Also, we believe that this kind of genetic planner would be easily deceived by simplistic fitness functions where closeness measures give a false idea of the actual distance to the desired goal. Consider for instance a robot moving in a labyrinth (the Euclidean distance to the goal position is very misleading) or the classical 8-puzzle. Some deliberative planning approaches based on search (McDermott, 1999; Bonet and Geffner, 1999) have to build complex heuristic functions from the domain information so that good distance estimations to the goal can be made. This shows that the problem is not a trivial one.

A more important problem is that search has to be done every time a problem needs to be solved. As GP is a weak search method and planning is NP-hard (Bylander, 1994), it is not expected that this approach will perform well on very big problems. It is possible though, that the crossover operator is a good heuristic way to explore the space of plans in some domains, although we do not know of any explicit empirical support for this.

1.2 Evolving Domain-Dependent Planners

Koza was the first to follow this approach by evolving a planner that solved a very specific subset of problems in the blocks world domain (Koza, 1992). Spector built a better system that was able to achieve a range of goal conditions from a range of initial conditions (Spector, 1994). However, only problems with three and four blocks were

tested. It seems that evolving a full-blown planner is a hard task. On the other hand, this approach solves a problem the previous one had, because its fitness function evaluates many different fitness cases (i.e., planning problems). Therefore, even if all the information we can get from a single fitness case is 1 or 0 (goal solved or not solved), the fitness function can still have a wide range of values to rank a population of individuals (i.e., $individual_1$ solves 3 out of 300 fitness cases, $individual_2$ 100 of them, etc.). So, many fitness cases may be able to compensate for poor closeness measures. Also, once a planner for a domain has been learned, only a small amount of search should be necessary to solve particular problems in the domain, as opposed to the previous approach. For instance, it is well known that in the classical blocks world domain, no search is required for solving problems of any arbitrary size: all the planner has to do is to move all the blocks to the table and then build the desired towers. Of course, finding optimum paths is another matter altogether (Gupta and Nau, 1992), although there are simple algorithms that can find near-optimal plans in the blocks world (Slaney and Thiebaux, 2001).

1.3 Evolving Planning Heuristics

This is the approach that is described in this article (Aler et al., 1998a; Aler et al., 1998b), which has been implemented in a system called EVOCK (Evolving Control Knowledge). Instead of evolving the whole domain-dependent planner, we start with a domain-independent planner. Domain-independent planning is known to be inefficient because of the unguided search it has to carry out. However, domain-dependent heuristics can be supplied to the planner, so that it makes informed decisions during search (i.e., to prune the search graph).

In this work, we have used GP for evolving such heuristics. We believe that this task should be easier for GP because classical planners are not brute force problem solvers but include powerful domain-independent heuristics (such as means-ends analysis). Therefore, GP has to do only part of the work. Besides, GP can indirectly use the reasoning abilities of such planners, which the plan evolvers cannot do. In practice, this approach is like the "evolving the planner" approach, but only a smaller part of the planner needs to be evolved – the heuristics. Therefore, besides (allegedly) being an easier problem for GP, it enjoys the advantages of the previous approach: once a domain-dependent planning system has been found, obtaining plans for individual problems in that domain should be computationally less expensive than carrying out the whole search process anew for every planning problem, as the plan evolvers must do. The basic intuition here is that problems of different size in a domain can use the same heuristics. For instance, solving a 50-block problem in the blocks world should need about the same heuristics than solving a 49-block problem. In particular, this is useful for solving large problems in the domain, where a pure search process will get bogged down. In previous experiments, for example, it has been shown that simple heuristics scale well in certain domains (Borrajo and Veloso, 1997).

In the GP context, a heuristic is best viewed as follows. Let us suppose that we already have a program P that could benefit from advice given by a function h at some points in its execution. For instance, if P is a planner that mindlessly searches the state space by applying planning operators forward, then P could call h to get some advice about which operator to apply next, instead of applying one at random. h's input is (part of) the internal state of P. If P is the forward planner mentioned before, h would benefit from having as inputs the current planning situation, the desired goal(s), and some additional information about the internal state of P (e.g., what planning situa-

tions or nodes have already been explored). At this point, standard GP could already be applied to the problem of evolving h. Conceptually, it is no different than evolving a function for a wall-following robot or a program for the Santa Fe trail. However, instead of evaluating h directly, $P + h$ has to be evaluated instead. In the GP jargon, P is a wrapper around h. In our work, we have utilized a planning system called PRODIGY4.0 (Veloso et al., 1995), which is much more sophisticated than the random-walk planner mentioned above and allows writing heuristics declaratively.

Additionally, evolving heuristics has a characteristic that might be exploited. Complete domain-independent planners can solve any solvable planning problem, given enough time. Therefore, fitness cases (i.e., training planning problems) can be pre-solved before being supplied to GP for learning. This preprocessing can be useful for extracting some information about the fitness cases that can be used in different ways. For instance, we can know how much memory or how long it takes the domain-independent planner to solve the fitness case. This can be used in the fitness function to compare the performance of an individual to the base planner performance. However, pre-processing the fitness problems offers a more interesting opportunity for evolving heuristics. Once a fitness case has been solved, all the steps and decisions the planner followed when solving it (i.e., the trace) are available. By analyzing this trace, the advice h that the planner would have benefited from for solving this problem can be obtained. Of course, this advice is useful only for this particular fitness case, but it could be used by the GP engine as a starting point to build more general heuristics. In this paper, we study a way to inject such "raw advice" into a GP system, without modifying the GP algorithm substantially. We use the standard crossover operator for this purpose, but the second parent individual is taken from a non-evolving population that contains raw advice heuristics. We call this operator the *instance-based operator* (IBC), because it uses instances previously acquired after analyzing several planning traces.[2]

2 Evolving Planning Heuristics for PRODIGY4.0

As explained before, the goal of this article is to evolve heuristics h for programs P that can use them. In particular, our framework can be used for those programs P that contain *decision* or *backtracking* points. For instance, programs that carry out search in a state-space are very representative of the latter definition.

In a decision point, a program must choose one alternative from a set of them before continuing its execution. Usually, P has little or no information about which alternative is preferable. Depending on the problem, if P makes the wrong decision, it might never find a solution. A decision point can also be a backtracking point. This means that if P made the wrong decision at this point, it will eventually backtrack to it so that another alternative can be tried. In that case, P would have saved time if it had made the right decision from the start. Heuristics can help such programs P in different ways. For this paper, we define a heuristic h as a function that can help a program P to find solutions and/or improve efficiency. More formally, if a program P can choose from a set of alternatives D_p at decision point p, a heuristic function h can be defined for that point as:

- $h : I \to D_p$, where I represents the set of possible internal states of the program P

However, a heuristic function need not return a single decision, but a set of them:

[2] In Aler et al. (1998a), we referred to it as a *knowledge-based operator*, because in a general sense, it uses knowledge previously acquired by another learning tool.

- $h : I \rightarrow D_p^*$, where D_p^* represents the set of all subsets from D_p ($D_p^* = \{X | X \subseteq D_p\}$)

In the latter case, h would make P more efficient if $|h(i)| < |D_p|$ for many $i \in I$. There are other definitions for h that could be used. For instance, a heuristic function could return an ordered set. Also, heuristics can have other purposes besides improving performance, like improving the quality of a solution, which might depend on decisions made at certain decision points. But as this article focuses on the $h : I \rightarrow D_p^*$ kind of heuristics, we will not make the formalism more complex than necessary.

It is not difficult to pose the problem of evolving such heuristics in a GP setting. For instance:

- Individuals: programs that represent a heuristic h for a decision point p. They take as input (i.e., terminals) features that characterize the internal state $i \in I$ of a program P and return a decision (or a subset of them) from D_p.
- Fitness cases: several problems appropriate for P. For instance, if the goal is to solve a Rubik cube, different starting points could be provided. If the goal is to solve blocks world problems, different problems made of pairs of initial situations and goals could be provided.
- Fitness function: if the goal is to solve as many problems as possible, the fitness function could evaluate individuals by counting how many fitness cases are solved when P is helped by the individual. If the goal is to improve the efficiency of P, then the time/space required to solve the fitness cases could be measured. Multi-objective fitness functions could try to achieve both goals at the same time.

The aim of this paper is to evolve heuristics for a planning program called PRODIGY4.0. PRODIGY4.0 is a STRIPS-based, domain-independent planning system that carries out bidirectional search in a state space. PRODIGY4.0 inputs are a description of the domain and a planning problem (s, g), where s is the initial situation and g is the goal to be solved. A domain description has two main components:

- a taxonomy of objects in the domain. For instance, in a logistics transportation domain (Veloso, 1994), there can be carriers, locations, and packages. In turn, there can be several types of carriers (ships, planes, trucks, etc.) as well as several types of locations (airports, post-offices, ports, etc.).
- a list of schema operators $a_i \in A$ for the domain. They are described using the STRIPS syntax, although PDL4.0 (PRODIGY4.0 Description Language) allows for more complex logical expressions that involve quantifiers. For instance, in a logistics transportation domain, there could be operators for loading a plane, for unloading it, for moving it to a different location, etc. Schema operators can contain free variables. An schema operator $a_i \in A$ whose free variables have been bound by a *binding* B is called a *grounded operator* and will be represented as $a_{B,i}$.

PRODIGY4.0 output is a plan. Formally, a plan is a sequence of grounded planning operators $a_{B_1,o_1}, a_{B_2,o_2}, \ldots, a_{B_n,o_n}$ that transforms s into another state where the goal g is fulfilled:

$$g(a_{B_n,o_n}(\ldots a_{B_2,o_2}(a_{B_1,o_1}(s)))) = \text{True}$$

In PRODIGY4.0, states s, goals g, and operators a_i are represented using extended-STRIPS. We refer readers to Fikes and Nilsson (1971) for details about STRIPS and

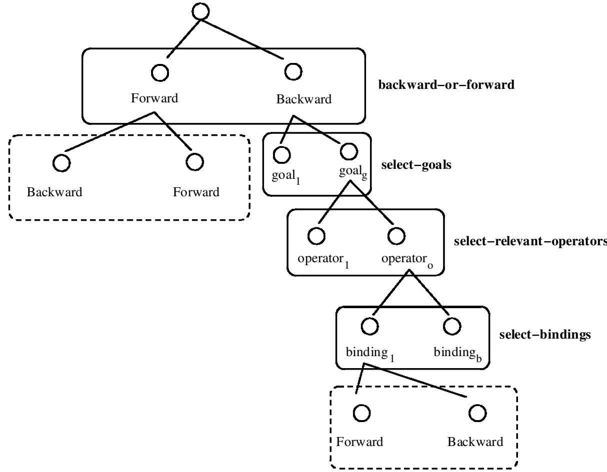


Figure 1: Tree of decisions generated by PRODIGY4.0 when searching for a solution to a problem.

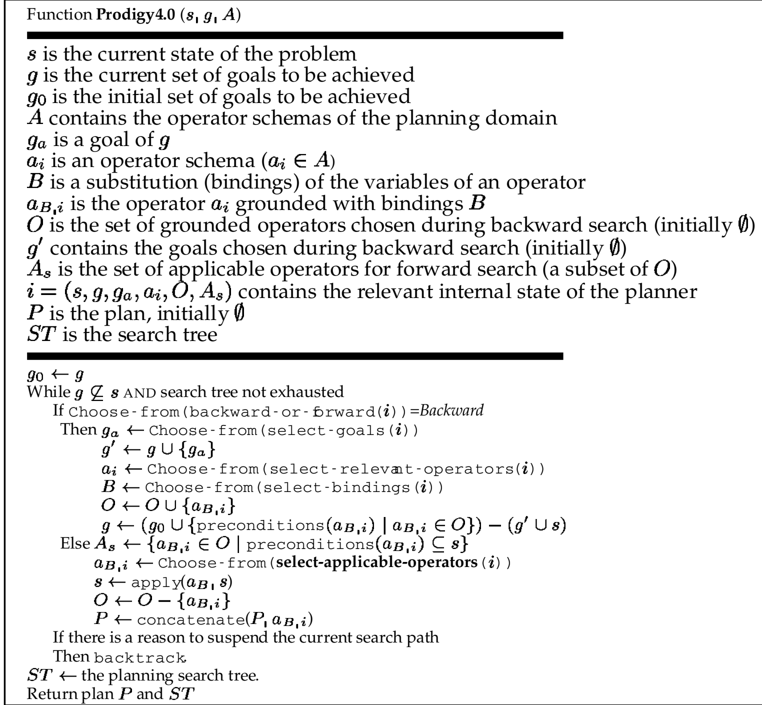
Veloso et al. (1995) for details about PDL4.0, the extended-STRIPS language used by PRODIGY4.0. Here, only a summary will be given. States are represented by means of a list of logical literals. For instance, in the well-known domain blocks world, $\{(\text{on-table } A), (\text{clear } A)\}$ indicates that block A is clear and on the table. Goals are represented likewise. For instance, the previous two literals could be interpreted as requiring that the goal state must fulfill two subgoals: that A must be on the table and that A must be clear. Operator schemas are represented by means of rules. The left hand side contains the (pre)conditions under which the operator can be applied. The right hand side contains the literals that should be added/removed from the current state to make up the new state.

The PRODIGY4.0 algorithm has been extensively described and formalized in Fink and Veloso (1996). The whole algorithm can be seen in Figure 2. Here, we will describe the algorithm in terms of its decision/backtracking points[3] because it is at those points where heuristics can be used to improve search performance, which is the concern of this paper. PRODIGY4.0 decision points are depicted in Figure 1. There are four of them: **backward-or-forward**, **select-goals**, **select-relevant-operators**, and **select-bindings**. It can be seen that every decision point can be represented as a branching point in a tree. Figure 2 also shows PRODIGY4.0 decision points embedded in PRODIGY4.0 algorithm. They are boldfaced and enclosed within the function `Choose-from` that chooses in sequence from a set of possible decisions.

As mentioned before, PRODIGY4.0 performs bidirectional search. This means that at any moment, PRODIGY4.0 can decide to carry out search from either the current state s (forward search) or from the current goal g (backward search). This is the first PRODIGY4.0 decision point, named **backward-or-forward** in Figure 1.

If backward search is selected, PRODIGY4.0 follows the means-ends heuristic: an operator is considered for backward search if it can achieve at least one of the goals in

[3] All PRODIGY4.0 decision points are also backtracking points.



Function **Prodigy4.0** (s, g, A)

s is the current state of the problem
g is the current set of goals to be achieved
g_0 is the initial set of goals to be achieved
A contains the operator schemas of the planning domain
g_a is a goal of g
a_i is an operator schema ($a_i \in A$)
B is a substitution (bindings) of the variables of an operator
$a_{B,i}$ is the operator a_i grounded with bindings B
O is the set of grounded operators chosen during backward search (initially $\emptyset$)
g' contains the goals chosen during backward search (initially $\emptyset$)
A_s is the set of applicable operators for forward search (a subset of O)
$i = (s, g, g_a, a_i, O, A_s)$ contains the relevant internal state of the planner
P is the plan, initially $\emptyset$
ST is the search tree

$g_0 \leftarrow g$
While $g \not\subseteq s$ AND search tree not exhausted
  If `Choose-from(backward-or-forward(`i`))` =*Backward*
   Then $g_a \leftarrow$ `Choose-from(select-goals(`i`))`
     $g' \leftarrow g \cup \{g_a\}$
     $a_i \leftarrow$ `Choose-from(select-relevant-operators(`i`))`
     $B \leftarrow$ `Choose-from(select-bindings(`i`))`
     $O \leftarrow O \cup \{a_{B,i}\}$
     $g \leftarrow (g_0 \cup \{\texttt{preconditions}(a_{B,i}) \mid a_{B,i} \in O\}) - (g' \cup s)$
   Else $A_s \leftarrow \{a_{B,i} \in O \mid \texttt{preconditions}(a_{B,i}) \subseteq s\}$
     $a_{B,i} \leftarrow$ `Choose-from(`**select-applicable-operators**`(`i`))`
     $s \leftarrow \texttt{apply}(a_B, s)$
     $O \leftarrow O - \{a_{B,i}\}$
     $P \leftarrow \texttt{concatenate}(P, a_{B,i})$
  If there is a reason to suspend the current search path
  Then `backtrack`
$ST \leftarrow$ the planning search tree.
Return plan P and ST

Figure 2: PRODIGY4.0 planning algorithm.

g. That is, if $g = \{g_1, g_2, \ldots g_g\}$, then $a_{B,i}$ is considered when its right hand side adds at least one of the g_i. PRODIGY4.0 chooses an $a_{B,i}$ by following a three step process, which corresponds to three different decision points:

1. A goal to be achieved g_a is selected from g (`select-goals` decision point).

2. An operator schema a_i that unifies with g_a is selected from A. a_i is also partially grounded at this point by unifying its right hand side with g_a (`select-relevant-operator` decision point).

3. If some of the variables of a_i are still free, a fully grounded operator $a_{B,i}$ is selected (`select-bindings` decision point).

Once an $a_{B,i}$ has been selected, it is added to the list O that contains all the grounded operators that have been selected during backward search. Also, the current goal g is recomputed by obtaining all the preconditions of the grounded operators in O, adding the initial goals, and removing those goals that are already true in s or that had already been chosen to be solved during backward search (the latter are stored in set g'). More specifically:

$$g \leftarrow (g_0 \cup \{\texttt{preconditions}(a_{B,i}) \mid a_{B,i} \in O\}) - (g' \cup s)$$

```
(control-rule select-operators-unstack
     (if (and (target-goal (on <x> <y>))
              (target-goal (on <y> <z>))))
     (then select goal (on <y> <z>))))
```

Figure 3: Example of a control rule for selecting the right goal.

Then, if the current state s does not fulfill the goal g yet, PRODIGY4.0 algorithm goes back to the first decision point.

For instance, if there were two goals to be achieved {(clear B), (on B A)}, the planner might select one of them (like (clear B)). Then, in order to achieve (clear B), the planner can select the operator unstack(<x>,<y>), which makes the robot arm pick block <x> from block <y> and clears block <y>. Variable <y> must be necessarily bound to B, but a binding has to be chosen for variable <x>. The planner might choose <x>=C. This would complete the three-step process for selecting an $a_{B,i}$, which in this case is unstack(C,B).

If forward search is selected, PRODIGY4.0 has to decide which one of the grounded operators that can be applied to s will actually be applied. PRODIGY4.0 follows the means-ends heuristic: only those operators that can achieve a goal are considered. Those operators have already been stored in O, the set of grounded operators selected during backward search to achieve some of the pending goals. But not all of them are actually applicable, so PRODIGY4.0 computes the set $A_s \subseteq O$, whose preconditions are true in s. Then, it selects one of them and applies it to the current state s. Selecting which operator to apply could also be a decision point, as several operators could be applicable. However, the version of PRODIGY4.0 used in this article carries out this step by using domain-independent heuristics (embedded in the function Choose-from(select-applicable-operators(i)) of Figure 2) that cannot be changed by control knowledge. Thus, we have not used this decision point for this article.

Therefore, there are four heuristic functions to be learned:

- **backward-or-forward**: $I \rightarrow \{\text{backward}, \text{forward}\}^*$, where I is the set of possible internal states of PRODIGY4.0
- **select-goal:** $I \rightarrow g^*$, where g contains the list of currently pending goals
- **select-relevant-operator:** $I \rightarrow A^*_{g_a}$, where g_a is the goal that was selected and $A_{g_a} \subseteq A$ is the set of all operators that add a literal that can be unified with goal g_a
- **select-bindings:** $I \rightarrow B^*_{g_a,a_i}$, where $a_i \in A$ is the operator schema that was selected to achieve goal g_a and B_{g_a,a_i} is the set of possible bindings for operator schema a_i when it is used to solve goal g_a

The four functions return a set of possible decisions, that will be tried in sequence by the function Choose-from (see Figure 2) after every backtracking. However, the order in which they are tried is determined by PRODIGY4.0, not by the heuristics themselves.[4]

[4] PRODIGY4.0 heuristics can also return ordered sets of decisions, but they have not been used in this paper. Some examples of those heuristics can be seen in Aler et al. (2001).

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

Figure 4: Example of a control rule for selecting the unstack operator.

PRODIGY4.0 provides a language to program heuristics by means of control rules. Each rule has a left hand side and a right hand side. In the left hand side, conditions under which the heuristic should be applied are described. Those conditions refer to the internal state of PRODIGY4.0 ($i \in I$). PRODIGY4.0 internal state is made of the current state s, the current pending goals g, the selected goal g_a, the selected operator a_i, the list of grounded operators chosen during backward search O, and the list of applicable grounded operators A_s in the current state s. In short, the internal state is (s, g, g_a, a_i, O, A_s).[5] PRODIGY4.0 internal state can be accessed via predefined functions called *meta-predicates*. Although PRODIGY4.0 provides some standard meta-predicates, additional ones can be defined by the user if necessary.

The right hand side of a control rule contains the decision that must be made under the conditions of the left hand side. A control rule is actually a template that could have free variables. If this is the case, PRODIGY4.0 grounds the control rule in all possible ways, and therefore, a single control rule can return a list of possible decisions.

Figure 3 shows an example of a control rule for the blocks world domain. This control rule says that if there are two pending goals of the form `(on <x> <y>)` and `(on <y> <z>)`, then the latter should be preferred to the former, since it is better to work on the bottom of a tower. If g contains many `on` goals, the variables of this control rule can be bound in many different ways. Each grounding of the control rule will give different advice (the goal to be selected). Therefore, as explained before, the result of a heuristic containing this single rule can be a set of goals to be selected (i.e., a set of alternative decisions).

Figure 4 shows another example of a control rule for the blocks world domain. This control rule says that if PRODIGY4.0 is working on trying to hold an object <`object1`>, and this object is on top of another one <`object2`> 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 use other operators than `UNSTACK` should be pruned.

Within this framework, the goal of GP is to automatically learn the four heuristic functions (`select-goal`, `select-relevant-operator`, `select-bindings`, and `backward-or-forward`). These can be coded as control rules for selecting goals, operators, bindings, and to determine whether PRODIGY4.0 should search forward or backward, respectively.[6] Therefore, every GP individual is a list of control rules. There are still many details to address before GP can be applied to this problem in practice. These will be described in Section 4.

[5] Here, only the part of the internal state that heuristics actually use will be considered.

[6] Besides rules for selecting, PRODIGY4.0 permits rules for preferring and rejecting, although they will not be used in this article. See Aler et al. (2001) for details.

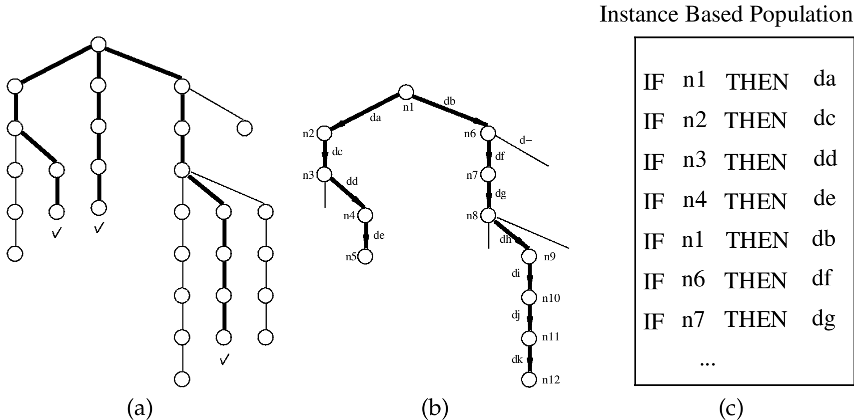


Figure 5: Analyzing the trace tree. (a) Whole trace tree (solution paths in thick lines). (b) Trace tree pruned by HAMLET. (c) Population made of the instances extracted from the pruned tree.

3 The Instance-Based Crossover (IBC) Operator

As mentioned in Section 1, it is possible to automatically obtain traces from PRODIGY4.0 (and from other programs with decision/backtracking points). In PRODIGY4.0, the nodes in the trace contain the internal state $i \in I$ of the program at any time, and the branches represent the decisions the algorithm actually made. By analyzing these traces, it is possible to determine when the algorithm made a right decision that led directly to the solution, and when it did not and had to backtrack. From this information, a heuristic function h could be written so that, if the algorithm was confronted with exactly the same problem, then it would make the correct decision. Of course, these heuristics are too specific to be of any use. The challenge here is to use them to bias GP so that it will find better solutions with less effort without changing the basic GP algorithm. Here, we propose the following three step process:

1. Analyze the traces and generate specific heuristics from the traces
2. Create a population – the instance-based population (IB population) – whose individuals are the specific heuristics generated in step 1
3. Use the crossover operator to inject individuals from the IB population (or parts of them) into the evolving population

Once a fitness case (a planning problem) has been supplied to PRODIGY4.0 , and PRODIGY4.0 has solved it, a trace (a search tree) is available for analysis. Analysis is meant to determine when PRODIGY4.0 made the right decision, the one that led to a success node in a leaf of the trace tree. This is achieved by marking those nodes in the path to a goal node from the root of the tree.

Figure 5(a) shows these marked paths with a thick line. It is possible that two or more different paths lead to a solution, like Figure 5(a) shows (thick lines). In some cases it would be better to prefer one to the other. For instance, as every path corre-

```
(if
     (and (CURRENT-GOAL (HOLDING D))
          (TRUE-IN-STATE (ARM-EMPTY))
          (TRUE-IN-STATE (CLEAR C))
          (TRUE-IN-STATE (CLEAR D))
          (TRUE-IN-STATE (ON C B))
          (TRUE-IN-STATE (ON D A))
          (TRUE-IN-STATE (ON-TABLE B))
          (TRUE-IN-STATE (ON-TABLE A))
          (SOME-CANDIDATE-GOALS NIL)
          )
  (SELECT OPERATORS UNSTACK))
```

Figure 6: Example of a control rule for selecting the `unstack` operator.

sponds to a plan, and plans can be measured with quality metrics,[7] some paths might be preferred to others. At the end, only the best paths will be left on the tree to learn from. In this case, only two of the three possible solution paths have been considered, as Figure 5(b) shows.

When the "good" paths have been selected, as in Figure 5(b), it is possible to generate heuristics valid for that trace tree. For instance, if n_1 is a node in one of those paths, n_2 is the next one along the path, and d_a is a decision that was made by planner to get from n_1 to n_2, then one could learn that $h(i(n_1)) = d_a$. Of course, d_a is a correct decision, because only the correct decisions are left in the pruned trace tree. There is an algorithm called HAMLET (Borrajo and Veloso, 1997) that already performs the selection of correct paths described here. Therefore, instead of reprogramming the whole algorithm, we have taken HAMLET sub-product and converted it to EVOCK internal format. HAMLET selects those paths corresponding to the shortest plans, in terms of number of operators in the plan. A newer version of HAMLET is able to select paths according to arbitrary quality measures, but we have not used it here (Borrajo et al., 2001).

These generated heuristics can be represented in exactly the same way as control rules, like the one in Figure 6. This means that they can be converted to GP individuals and crossed over with other individuals. They could possibly be used in many different ways. In this article, they are stored in a non-evolving population, and the standard crossover operator can be used to inject those individuals, or parts of them, into the evolving population as illustrated in Figure 7. This is achieved by using an individual from the evolving population as first parent for crossover and an individual from the non-evolving population as second parent. The latter can be done because both the members of the evolving population and the ones in the IB population are GP individuals. The root of the offspring always comes from the evolving parent. Initially, it was expected that injecting parts of these individuals into the main population could be useful, because it would add useful code that could be crossed over and mutated by the genetic operators. Whether this is true or not will be empirically tested in Section 5.

In this article, only heuristics that follow the solution path have been used to fill the IBC non-evolving population. However, heuristics to avoid walking away from the solution path might also be useful. For instance, in Figure 5, node n_6 might leave

[7] Quality metrics measure properties of a plan. The most common one is the number of operators in the plan (i.e., the length of the plan). But other properties like execution time, economical cost of executing the plan, etc. can be measured (Pérez and Carbonell, 1994; Borrajo et al., 2001).

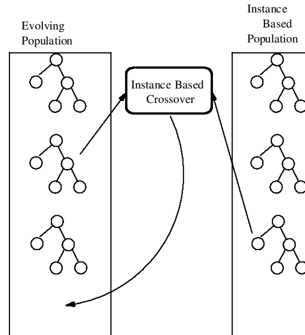


Figure 7: Instance-based crossover between the evolving population and a population containing specific heuristic individuals.

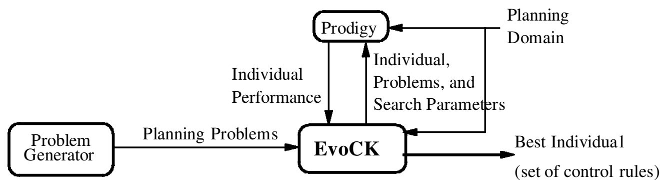


Figure 8: EvoCK relations with other systems.

the right path if decision $d-$ is made. Therefore, individuals including such specific heuristics could be included in the non-evolving population. However, we have not used such rejection heuristics in this paper.

The main weakness of the IBC operator is that in order to use it, the planner must be able to solve many planning problems on its own (without heuristics). This is possible only if these planning problems are easy enough. So, there is a hidden assumption here: specific heuristics obtained by solving simple problems must suffice for the IBC to work.

4 EvoCK

EvoCK is the GP-based tool that implements the ideas described in previous sections. It searches the space of PRODIGY4.0 planning heuristics, and it is able to use the IBC. As the IBC is optional, from now on, we will refer to the system as IBC-EvoCK when the IBC is used and autonomous-EvoCK when it is not. There are some details about EvoCK that will be discussed next. Subsection 4.1 describes EvoCK relations with other systems. Subsections 4.2 and 4.3 discuss how EvoCK individuals are generated and genetically modified, respectively. Subsection 4.4 describes the fitness function.

4.1 EvoCK Architecture

Figure 8 shows how EvoCK is related to two other important external systems: the base planner (PRODIGY4.0) and the random problem generator.

EvoCK receives the fitness cases (the training planning problems) from a generator that is able to create planning problems at random. Currently, this generator has

to be designed for each planning domain. The link between it and EvoCK is off-line: a human decides what kind of problems should be generated for training and testing, which are subsequently stored in a file to be read by EvoCK. PRODIGY4.0 is used by EvoCK to evaluate individuals. Individuals are heuristics for planning, and more specifically, each individual is a list of control rules. The interface between EvoCK and PRODIGY4.0 has two parts:

- The EvoCK → PRODIGY4.0 communication: EvoCK sends PRODIGY4.0 an individual to be evaluated, and the planning problem it should be evaluated with. EvoCK also imposes a time limit and can specify to PRODIGY4.0 other search parameters.
- The PRODIGY4.0 → EvoCK communication: PRODIGY4.0 returns to EvoCK information about whether it was able to solve the problem by using the individual as guiding heuristics, how long it took, and the number of nodes that were expanded. This information is used by EvoCK to compute the fitness of the individual.

This architecture is quite flexible in the sense that PRODIGY4.0 could be substituted quite easily by another planner with similar inputs and outputs. The most important requirement for the base planner is that it can use heuristics and that they can be loaded into the system easily. Unfortunately, not all planners comply with this requirement (this is one important reason that led us to choose PRODIGY4.0), but there are some that do. For instance, we believe that EvoCK could be applied to current implementations of planners like UCPOP. EvoCK is quite independent of the syntax used to define the heuristics (which is planner-dependent) because it uses a grammar to constrain the individuals that can be changed by hand for new planners (this grammar is explained in Subsection 4.2). However, this grammar is partially built from the domain description, which is written using PRODIGY4.0 domain description language (PDL4.0). Other planners describe domains differently, so this part would have to be reprogrammed so that EvoCK can be applied to other planners.[8]

4.2 EvoCK Individuals

EvoCK individuals are sets of control rules, represented in an internal language very close to the actual PRODIGY4.0 control rules. It is typical in GP to assume operational closure (Koza, 1992). However, in our case, PRODIGY4.0 forces a quite restricted syntax. If heuristics are not syntactically correct, then the planner will fail. In this paper, we have used constrained structures (Koza, 1992) (also similar to strongly typed structures (Montana, 1995)). In order to work with these structures, three aspects must be considered: only valid structures must be created; crossover points must be of the same type; and mutation operators must take into account the type of the mutation point. We have achieved these three points by using a special purpose grammar. This grammar has production rules that follow the structure $A \rightarrow (F_a A_1 A_2 \ldots) | (F_b B_1 B_2 \ldots) | \ldots$. The grammar is partly domain dependent and is automatically built by EvoCK from the domain description. Table 1 shows the grammar that would be built for the blocks world domain.

The grammar is used to create S-expressions, which are the internal language to represent individuals. For instance, the fourth rule of Table 1 specifies that the

[8]However, PDL4.0 is very similar to PDDL, which is used by several modern planners, so the reprogramming effort would not be too great.

Table 1: Grammar for generating syntactically correct sets of control rules in the blocks world.

LIST-ROOT-T	→	(list RULE-T) | (list RULE-T RULE-T) | ...
RULE-T	→	(rule AND-T ACTION-T)
AND-T	→	(and METAPRED-T) | (and METAPRED-T METAPRED-T) | ...
METAPRED-T	→	(true-in-state GOAL-T) | (target-goal GOAL-T) |
		(current-goal GOAL-T) | (some-candidate-goals LIST-OF-GOALS-T)
LIST-OF-GOALS-T	→	(list-goal GOAL-T) | (list-goal GOAL-T GOAL-T) | ...
ACTION-T	→	(select-goal GOAL-T) | (select-operator OP-T) |
		(select-bindings BINDINGS-T) | sub-goal | apply
OBJECT	→	$< object-1 >$ | $< object-2 >$ | ...
OP-T	→	pickup | put-down | stack | unstack
BINDINGS-T	→	(pick-up-b OBJECT) | (put-down-b OBJECT)
		| (stack-b OBJECT OBJECT) | (unstack-b OBJECT OBJECT)
GOAL-T	→	(clear OBJECT) | (on-table OBJECT)
		| (arm-empty) | (holding OBJECT) | (on OBJECT OBJECT)

```
(list
  (rule
    (and
      (current-goal (clear <a>))
      (true-in-state (clear <b>)))
    (prefer-operator unstack put-down)))
```

Figure 9: Example of EvoCK control rule.

METAPRED-T generative symbol can be expanded into four different alternatives. Each alternative corresponds to a different meta-predicate. Notice also that every meta-predicate has arguments that can be expanded further (they are represented by the generative symbols in uppercase) and that they are of different types: GOAL-T and LIST-OF-GOALS-T. This way, meta-predicates will always have the right type of arguments.[9]

By applying the grammar recursively starting from the symbol LIST-ROOT-T, it is possible to create correct new individuals. Figure 9 displays an example of EvoCK control rule generated by the grammar.

It is also possible to create parts of individuals, as required by the mutation operator, by starting from any other generative symbols. The crossover operator also uses the grammar. In order to cross over two individuals, a random point is selected in the first parent. Let us suppose that the symbol at the random point is true-in-state. Then, the generative symbol that generated the symbol at the random point is determined by means of the grammar (METAPRED-T, in this example). Finally, a random crossover point containing a symbol generated by the same generative symbol is selected in the second parent. Then, the two subtrees are exchanged, as is done in the standard crossover operator. As the two of them have been generated by the same generative symbol, it is ensured that the resulting individual will be syntactically correct.

Additionally, individuals have to be protected before being sent to Prodigy4.0 for evaluation by adding some meta-predicates in the left hand side of the control rules of the individuals. This is detailed in Table 2.

Finally, an additional protection is required. Prodigy4.0 requires that after match-

[9]See Aler et al. (2001) for examples of different grammars that can be used by EvoCK for Prodigy4.0.

Table 2: Additional protections required by the control rules.

Type of rule	Protection
select goal	A meta-predicate `target-goal` is added to check that the goal to be selected in the right hand side is actually one of the pending goals.
select operator	A meta-predicate `appropriate-operator` is added to ensure that the operator that will be selected by the control rule can achieve the current goal.
select bindings	A meta-predicate `current-operator` is added to ensure that the variables in the bindings to be selected by the rule are appropriate for the operator chosen by PRODIGY4.0.

ing a control rule, all variables are bound. To ensure the no free variables restriction, a meta-predicate `type-of-object(<variable>, type)` is added for every variable in the control rule. This meta-predicate binds a `<variable>` in all possible ways allowed by the `type` in case the variable is free. This requires that every variable has an associated type. This is achieved by coding the type in the variable's name. For instance, the variable `<carrier-1>` would require the meta-predicate `type-of-object(<carrier-1>, carrier)` to the control rule.

As explained in Section 2, control rules can access PRODIGY4.0 internal state by using meta-predicates. In the present work, evolved individuals can only access the current state s, the current pending goals g, and the currently selected goal $g_a \in g$. To do this, the individuals use the following meta-predicates:

- s is accessed by meta-predicate `true-in-state(literal)`. It determines if the `literal` unifies with a literal of the current state s or not.
- g is accessed by meta-predicate `target-goal(literal)` and `some-candidate-goals(literal1 , literal2,...)`. `target-goal` tests whether the `literal` unifies with one of the unachieved goals in g. `some-candidate-goals` determines if at least one of a list of literals unifies with an unachieved goal in g.
- g_a (the goal that the planner has selected) is accessed by meta-predicate `current-goal(literal)`. It indicates whether the `literal` unifies with g_a.

4.3 EVOCK Genetic Operators

The system uses the traditional GP operators (crossover and mutation), in addition to the IBC operator. However, as every EVOCK individual is a list of rules, some specially tailored genetic operators have been added. Some of them are similar to Koza's architecture altering operators (Koza, 1995; Koza and Andre, 1995; Bennett III, 1996). These additional genetic operators are:

- **Adding Crossover**: this operator is able to add rules and conditions to an individual from another individual.

- **Growing Mutation**: this operator can grow random rules and conditions to an individual.
- **Chopping Off Mutation**: it removes rules and conditions from individuals at random points.
- **Join**: It takes two different variables in a control rule and makes them the same variable. For instance, if we have 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>`. The **join** operator is a shortcut to create these references.
- **Hierarchy Generalization**: as explained previously, objects in PRODIGY4.0 are organized in a hierarchy or taxonomy. For instance, in a 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 a rule (like `<truck-1>`) and substitute all its instances by a carrier-typed variable (like `<carrier-1-1>`). Thus, the control rule would become more general.

The related operators (i.e., disjoin and hierarchy specialization) are not included in the operator set because we believe that join and hierarchy generalization are good biases for EVOCK. In any case, the effects of disjoin and hierarchy specialization can be achieved by means of the rest of the genetic operators.

4.4 Fitness Function

EVOCK fitness function evaluates every individual by loading its heuristics into PRODIGY4.0 and running the planner with several fitness cases (the training planning problems). There are many aspects of an individual that must be assessed. Therefore, the fitness function contains several objectives or components. EVOCK uses tournament selection and evaluates individuals by means of a lexicographical ordering of the fitness components (see for instance Koza et al. (1999)). This is also called a hierarchical fitness function. Pareto techniques could have been used, as in Langdon (1995), but our current approach works quite well and does not require keeping the whole Pareto front. EVOCK fitness function contains three main components in the following order:

1. **Performance in fitness cases** (to maximize). Its goal is to force EVOCK individuals to solve as many fitness cases as efficiently as possible. It includes three subcomponents that will be explained later in more detail.
2. **Number of different variables** (to minimize). It is computed as follows:
   (a) Let $\{r_{i,1}, r_{i,2}, \ldots, r_{i,k}\}$ be the control rules of individual i.
   (b) $\forall j = 1..k$ compute $\text{CV}(r_{i,j})$, where the CV function calculates the number of different variables in a control rule.
   (c) $n_i = \sum_{j=1}^{k} \text{CV}(r_{i,j})$, where n_i is the number of different variables for individual i.

   This fitness component is related to the same bias as the join operator.
3. **Compacity/generality components** (to maximize). This includes several other components to encourage compacity and generality of the individuals. They will be explained later.

The main fitness component (performance in fitness cases) is computed using three sub-components:

- C_1: **Number of problems solved more efficiently than** PRODIGY4.0 **alone** (to maximize). Efficiency in this case means fewer nodes expanded.
- C_2: **Number of problems solved** (to maximize).
- C_3: **Number of nodes expanded** (to minimize).

C_1, C_2, and C_3 deserve a longer explanation. The goal of EVOCK is to find individuals that solve as many problems as efficiently as possible. This is directly taken care of by the C_2 and C_3 subcomponents.[10] However, an additional fitness component C_1 had to be added to solve the following problem.

In order to train EVOCK, many fitness cases (planning problems) are required. Therefore, the timeout for each fitness case must be small. This implies that the fitness cases must be easy to solve; otherwise, C_2 would almost always be zero. However, PRODIGY4.0 is able to solve easy problems without heuristics. Therefore, a good strategy for individuals to score well in C_2 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 (like checking whether the planner is trying to achieve two different goals at the same time, which is impossible). Individuals that do modify the behavior of the planner are likely to be only partially 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 some of the latter might be only partially wrong and might be corrected in the long term. To change this undesirable behavior, we added a first component C_1 that forced individuals to do something positive: solving problems using fewer nodes than PRODIGY4.0. By means of this component, those individuals that do nothing would be selected against, because they do not change the behavior of the planner.

However, there is a risk: there can be some individuals that solve some problems expanding fewer nodes than PRODIGY4.0 alone (for instance, $C_1 = 20\%$), but they actually solve very few of the training problems ($C_2 = 30\%$). This might happen, but in the long term, it is expected that once C_1 cannot be improved further, C_2 will begin to improve. Empirical examples of this behavior will be shown in Section 5.

The compacity/generality component also needs a longer explanation. It includes five subcomponents, ordered as follows:

1. G_1 : **Number of `true-in-state` meta-predicates** (to minimize). The more tests on s there are in a control rule, the less general it is, so this component should be minimized. $G_{1,i}$ is computed as $\sum_{j=1}^{k}$ `n-tis`$(r_{i,j})$, where k is the number of control rules of individual i, and `n-tis` calculates the number of `true-in-state` meta-predicates in a control rule.

2. G_2 : **Number of arguments in `some-candidate-goals` meta-predicates** (to maximize). It happens that `some-candidate-goals(goal1, goal2, ...)` is equivalent to:

   `target-goal(goal1)` $\vee$ `target-goal (goal2)`$\vee \ldots$

[10] It would have been better to use planner time instead of nodes expanded by PRODIGY4.0 to measure efficiency. However, measured time depends slightly on external factors and makes experiments unreproducible.

Table 3: EvoCK tableau summarizing its main parameters.

Objective:	Find a set of control rules that optimizes the hierarchical fitness function defined previously
Architecture of the individual Terminal/Function set Fitness cases	Defined by a grammar, which depends on the planning domain (see Table 1) Terminal symbols of a grammar (see Table 1) 188 (blocks world) 192 (logistics domain). Types of training problems: (2,5), (2,4), (2,3), (2,2), (1,5), (1,4), (1,3), (1,2)
Fitness	A hierarchical function with three main components (performance, number of different variables, compacity/generality). It has been defined previously.
Wrapper	PRODIGY4.0
Parameters	M=2 and M=300, Evaluations=100000
Selection method	2-tournament (M=2) and 5-tournament (M=300)
Success predicate	None

Therefore, increasing the number of arguments for `some-candidate-goals` meta-predicates increases the generality of control rules (i.e., they can be applied to more situations). G_2 is computed as $\sum_{j=1}^{k} \texttt{n-arg-scg}(r_{i,j})$, where `n-arg-scg` computes the number of arguments in the `some-candidate-goals` meta-predicates of control rule $r_{i,j}$.

3. G_3: **Number of `some-candidate-goals` meta-predicates** (to minimize). The purpose of this component is to minimize the number of checkings on g. $G_{3,i}$ is computed as $\sum_{1}^{k} \texttt{n-scg}(r_{i,j})$, where k is the number of control rules of individual i, and `n-scg` calculates the number of `some-candidate-goals` meta-predicates in a control rule.

4. G_4: **Number of control rules** (to minimize). This component presses towards compact descriptions of the heuristics, which are faster to evaluate.

5. G_5: **Size in nodes** (to minimize). This component prefers individuals with fewer nodes.

Finally, we use a steady-state GP with a generational gap of 1 and tournament selection for both selection and replacement. We experimented with two population sizes: 2 and 300. A 2-population with a 2-tournament is roughly equivalent to hill climbing, and it was initially used as a baseline to compare results with larger populations. It turned out that hill climbing was best for the domains we have used. All genetic operators described in Subsection 4.3 have the same probability of being applied. EvoCK main parameters are summarized in Table 3.

5 Empirical Study

The aim of this section is to provide an empirical evaluation of EvoCK. The first subsection describes the experimental setup. The second one evaluates the heuristics obtained by EvoCK. The third subsection discusses the learning curves.

5.1 Experimental Setup

The main purpose of this study is to determine how well a GP-based system can perform when applied to finding good planning heuristics. Second, the effects of the IBC will be shown and analyzed. For that purpose, four different experimental configurations have been studied: autonomous-EvoCK, autonomous-EvoCK (300), IBC-EvoCK , and IBC-EvoCK (300). Autonomous-EvoCK is a 2-population pure GP (no IBC is used), whereas IBC-EvoCK uses the IBC operator instead of the standard and growing mutation operators. When a population larger than 2 is used, the size of this population is shown between parentheses (300 in this case). 2-tournaments were used for 2-populations and 5-tournaments for 300-populations. No crossover operators, except the IBC, were used with 2-populations for obvious reasons.

Both configurations have been trained and evaluated in two planning domains: the blocks world and the logistics transportation domain. Both of these domains have been extensively used in planning research. The blocks world is a simple domain, but classical planners find it difficult to solve, especially when problems include many blocks. The logistics domain is a more complicated domain based on a real problem where packages have to be delivered to different places in different cities using different carriers (planes and trucks, in this case). For both domains, we use the definition that comes in the PRODIGY4.0 distribution.[11]

Both autonomous-EvoCK and IBC-EvoCK were trained with 188 and 192 fitness cases from the blocks world and the logistics domain, respectively. Both sets of fitness cases contain the following kind of problems: (2,5), (2,4), (2,3), (2,2), (1,5), (1,4), (1,3), (1,2), where the first number represents the number of goals and the second the number of objects (blocks in the blocks world and packages in the logistics domain).

In the blocks world, (#n,#g) problems are obtained by first generating two random blocks world states s_0 and s_f with #n blocks and then translating them into STRIPS format (a list of literals). s_0 is the initial state of the problem. To build the goal g, #g literals are chosen from s_f and they are shuffled. The result is a random problem (s_0, g) in the blocks world.

In the logistics domain, (#n,#g) problems depend mainly on the parameter #n, which is the number of packages. Every problem has #n packages, #n airplanes, and #n cities. Every city has an airport, a post-office, and a truck. As in the blocks world, two states s_0 and s_f are generated. In both of them, the truck of a city can be either at the airport or at the post-office (this is randomly determined). The #n airplanes are randomly distributed between the #n airports. Finally, the #n packages are randomly distributed between the #n airports, the #n post-offices, the #n airplanes, and the #n trucks. Both states s_0 and s_f are converted into STRIPS format. A random problem in the logistics domain is a pair (s_0, g), where g is obtained from s_f by choosing #g literals and shuffling them.

IBC-EvoCK requires an IB population. Therefore, solutions to 400 additional planning problems were processed by HAMLET, and these specific heuristics extracted

[11] The standard PRODIGY4.0 distribution can be obtained from the project homepage at http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/project/prodigy/Web/prodigy-home html.

Table 4: Distribution of testing problems for the blocks world.

Blocks world			
# Goals	# Objects	# Problems	% Problems
50	50	16	4%
20	50	48	12%
20	20	48	12%
10	50	48	12%
10	20	48	12%
10	15	48	12%
5	50	40	10%
5	20	40	10%
5	15	40	10%
5	10	40	10%

by HAMLET were converted to IBC-EVOCK format and stored in the non-evolving population. Both systems ran for 100000 evaluations, an evaluation being a call to PRODIGY4.0 to evaluate a single fitness case (i.e., a planning problem). Each configuration was run 58 times starting with different random seeds. After every GP-run, the best individual was obtained and evaluated with a mixture of easy, difficult, and very difficult problems (up to 50 goals and 50 blocks/packages) to determine whether the obtained heuristics are general and scale well. There were 416 testing problems in the blocks world and 348 in the logistics domain. Tables 4 and 5 show how the test problems are distributed. Testing problems have the same profile as learning problems.

The time limit for testing a problem is calculated with Equation 1:

$$t_{test} = \frac{150}{16}(1 + \text{floor}(\frac{\#\text{goals}}{10})) \text{ seconds,} \quad (1)$$

where #goals is the number of goals in the testing planning problem. A time limit is necessary so that testing is done in a reasonable time. In any case, we have observed that this time limit is more than enough for individuals to show their worth. Testing was carried out on a 400MHz Pentium II with 256MB RAM.

5.2 Test Results

The GP runs for autonomous-EVOCK and IBC-EVOCK are summarized in Figure 10. This figure displays a cumulative frequency graph. It shows the frequency with which an experimental setup yields a set of control rules (y-axis) that is able to solve a certain percentage of the testing problems (x-axis). For instance, point (0.4, 0.3) on Figure 10(a) means that if we run autonomous-EVOCK 100 times, about 40 of them are expected to find an individual that solves at least 30% of the testing problems. Results for PRODIGY4.0 alone (i.e., with no heuristics) are also provided for comparison purposes (the dashed vertical line in Figure 10).

The first result is that autonomous-EVOCK is able to obtain individuals that solve a high percentage of the testing problems in both domains. In these particular runs, the maximum is 80% in the blocks world and almost 100% in the logistics domain. Also, the individuals found are very compact and fast to evaluate (3 rules in both the blocks world and the logistics domain). In addition, autonomous-EVOCK best individuals are able to solve very difficult problems. In the blocks world, the best individual solves 5 (out of 16) of the 50 goal/50 object testcases. In the logistics domain, the best individual solves 37 (out of 48) of the 50 goal/50 package testcases. Tables 6 and 7

Table 5: Distribution of testing problems for the logistics domain.

Logistics			
# Goals	# Objects	# Problems	% Problems
50	50	48	14%
20	50	30	9%
20	20	30	9%
10	50	15	4%
10	20	15	4%
10	15	15	4%
10	10	15	4%
5	50	12	3%
5	20	12	3%
5	15	12	3%
5	10	12	3%
5	5	12	3%
2	50	10	3%
2	20	10	3%
2	15	10	3%
2	10	10	3%
2	5	10	3%
2	2	10	3%
1	50	10	3%
1	20	10	3%
1	15	10	3%
1	10	10	3%
1	5	10	3%
1	2	10	3%

show a complete breakdown of the results for the blocks world and the logistics domain, respectively. PRODIGY4.0 is unable to solve any of the difficult problems within the same time limit. Therefore the heuristics evolved by autonomous-EVOCK are very good when compared to the base planner. This is also true of other planners. The hardest instances of problems in the blocks world (50 goals and 50 objects) were tested with GRAPHPLAN (Blum and Furst, 1995) and UCPOP.[12] None of them managed to solve any of the problems within the 100 seconds time limit. We did so to show that solving "hard" planning problems is difficult for different planners.

Figure 10 also shows results for the IBC operator. Results for the blocks world differ from those of the logistics domain. In the blocks world, autonomous-EVOCK and IBC-EVOCK are both able to obtain individuals that solve 80% of the problems. Unfortunately, the IBC does not seem to be able to supply good genetic material to reach better individuals than autonomous-EVOCK. The actual effect of the IBC is to bias IBC-EVOCK towards a local minima at about 51%. For this value, IBC-EVOCK does better than autonomous-EVOCK but gives no improvement for individuals that solve more than 51% of the problems (i.e., IBC-EVOCK finds 51% individuals more frequently: 0.44 vs. 0.25, but better individuals are more frequently found by autonomous-EVOCK). However, the IBC operator does improve the learning rate in this domain, as should be expected when some knowledge is injected into the population, as will be shown in Subection 5.4. Results for both configurations with the larger population

[12] We have used the Lisp implementations of the University of Washington. UCPOP can be found at http://www.cs.washington.edu/ai/ucpop.html, and sensory GRAPHPLAN can be found at http://www.cs.washington.edu/ai/sgp.html.

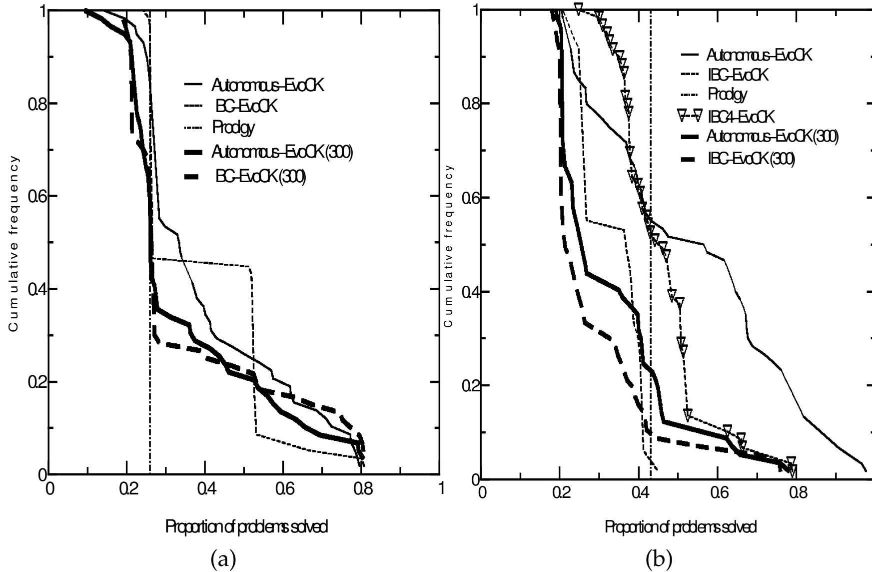


Figure 10: Frequency of GP-runs (y-axis) that are able to solve a proportion x of problems (x-axis) or more. (a) Results for the blocks world. (b) Results for the logistics domain.

(300) are also shown (thick lines). It can be seen that larger populations seem to perform slightly worse than the simple hill-climbing approach.

In the logistics domain, testing results for the IBC-EvoCK are very bad. Almost none of the individuals is able to improve PRODIGY4.0! When the IB population for the logistics domain was examined, it was found out that the control rules stored there had many more conditions than in the blocks world. Also, it seemed that autonomous-EvoCK spent most of its time pruning useless condition branches from the individuals. Those branches had previously been injected by the IBC. It was hypothesized that as it seems successful individuals in this domain are simple, IBC-EvoCK found it very difficult to profitably use the genetic material of the IB population. Therefore, it was decided to prune them randomly to a manageable size (not more than 4 conditions) before being used by the IBC. Results are labelled in Figure 10(b) as IBC4-EvoCK. Results are much better, but they do not go as far as autonomous-EvoCK. Results for autonomous-EvoCK (300) and IBC-EvoCK (300) are also shown. It is noticeable that autonomous-EvoCK (300) does much worse than autonomous-EvoCK. However, IBC-EvoCK (300) does better than its small-size relative IBC-EvoCK. This is reasonable, because IBC-EvoCK has 300 initial random individuals that can be recombined and pruned and does not depend so much on the IBC to generate new good individuals. On the other hand, IBC-EvoCK only has 2 initial random individuals and the IBC operator. Everytime the IBC is used, big chunks are added to the current individual. Then, the IBC-EvoCK spends most of its time pruning the resulting individual. This effect will be shown in Subection 5.4.

In order to determine how significant the curves of Figure 10 are, we have used an approximate randomization test (Cohen, 1995). A non-parametric test has been

Table 6: Breakdown of the number of testing problems solved in the blocks world by autonomous-EVOCK according to the number of goals and of objects/packages.

# Goals	# Objects	PRODIGY4.0	autonomous-EVOCK
50	50	0%	31%
20	50	8%	71%
20	20	10%	65%
10	50	21%	83%
10	20	25%	75%
10	15	40%	77%
5	50	18%	88%
5	20	20%	92%
5	15	42%	95%
5	10	65%	95%

Table 7: Breakdown of the number of testing problems solved in the logistics domain by autonomous-EVOCK according to the number of goals and of objects/packages.

# Goals	# Objects	PRODIGY4.0	autonomous-EVOCK
50	50	0%	77%
20	50	3%	97%
20	20	7%	100%
10	50	13%	100%
10	20	20%	100%
10	15	20%	100%
10	10	7%	100%
5	50	42%	100%
5	20	58%	100%
5	15	42%	100%
5	10	25%	100%
5	5	33%	100%

used because no assumptions need to be made about the distribution properties, as a t-test requires. It works as follows. $f(S_A, x)$ and $f(S_B, x)$ are the cumulative frequency curves of Figure 10 for configurations A and B, respectively (A could be autonomous-EVOCK and B could be autonomous-EVOCK (300), for instance). S_A is a set containing one item for every A-configuration GP-run, each item being the frequency of test problems solved by the best individual of that GP-run. Likewise for S_B. We want to know whether the difference $d_{x_i}(S_A, S_B) = |f(S_A, x_i) - f(S_B, x_i)|$ at x_i is significant. For instance, if assuming that both S_A and S_B have been drawn from the same underlying distribution, the probability of obtaining such difference is large, then that difference could not be considered significant. On the contrary, if the probability is very small (in our case, less or equal to 0.01), the hypothesis that the difference has been obtained by mere chance, is rejected. In order to do this, new sets S_A^j and S_B^j are obtained by sampling from $S = S_A \cup S_B$, and $d_{x_i}^j = d_{x_i}(S_A^j, S_B^j)$ is calculated. If this process is iterated many times (for many j's, 100 in our case), a sample $S_{d_{x_i}}$ made of the $d_{x_i}^j$'s can be obtained. Then, $S_{d_{x_i}}$ is sorted and the critic value $d_{x_i}^{c99}$ that bounds the lowest 1% of $S_{d_{x_i}}$ is determined. If the actual difference $d_{x_i}(S_A, S_B)$ is smaller or equal to $d_{x_i}^{c99}$, then the probability of obtaining such a small difference by mere chance is smaller than 0.01,

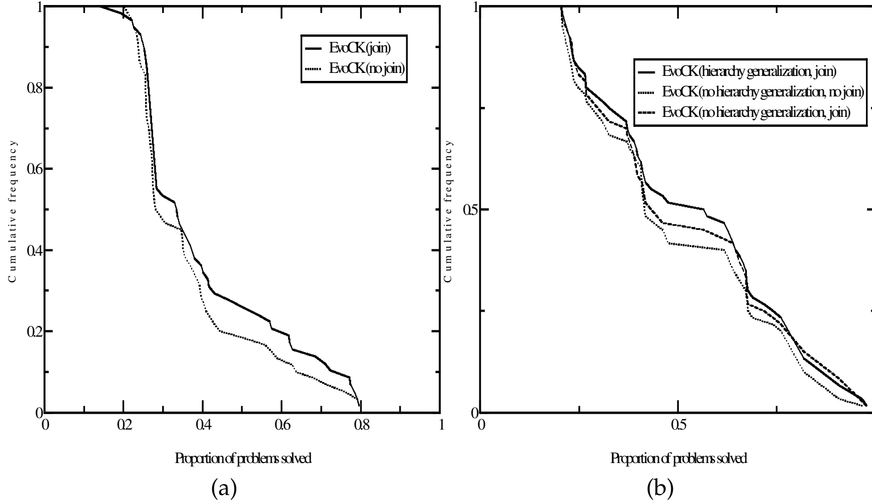


Figure 11: The effects of the Join and Hierarchy Generalization operators in (a) the blocks world and (b) the logistics domain.

and that hypothesis is rejected.

Applying the previous framework, we find that the differences between autonomous-EvoCK and autonomous-EvoCK (300) in the blocks world (see Figure 10(a)) are not too significant, which is what is actually observed in the figure. According to the randomization test, there are only significant differences in small ranges at (0.22-0.25).[13] With respect to the IBC in the blocks world, the randomization test confirms that the large differences observed in Figure 10(a) between autonomous-EvoCK and IBC-EvoCK are also significant. (More specifically, differences in the range (0.51-0.62), except in small subranges (range (0.52-0.53), and points 0.59 and 0.61).)

In the logistics domain, only the differences between autonomous-EvoCK and IBC4-EvoCK were checked for significance (see Figure 10(b)). The randomization test shows that differences are significant in the ranges (0.22-0.36) and (0.52-0.82). Again, the test confirms what is observed in the figure.

5.3 The Join and Hierarchy Generalization Operators

In this section, we analyze the effects of two non-standard EvoCK genetic operators (Join and Hierarchy Generalization (HG)). In the blocks world, there is no hierarchy of objects, therefore only two configurations were tested: EvoCK with Join and EvoCK without Join. Results are summarized in Figure 11(a). In the logistics domain, these two configurations were tested and compared to the standard one (Join, HG). Results are summarized in Figure 11(b). The randomization test shows that there are no significant differences between curves at the 1% level. Therefore, the intuitions underlying those operators have no noticeable positive or negative effect in the chosen domains.

[13] Those ranges are approximate, as the minimum step between different x_i's is 0.02.

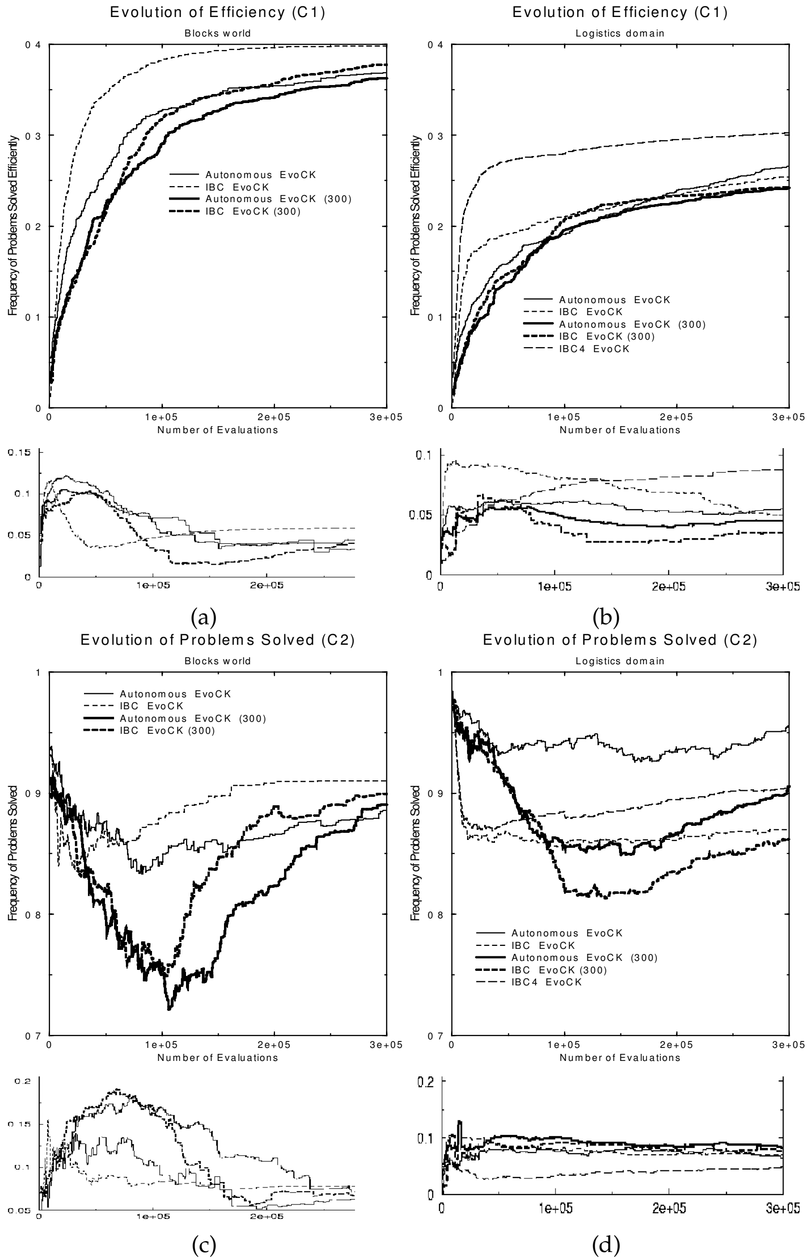


Figure 12: Evolution of components C_1 ((a) blocks world, (b) logistics domain) and C_2 ((c) blocks world, (d) logistics domain). Large plots are averages, small plots are standard deviations.

5.4 Learning Curves

The purpose of this section is twofold. First, it shows how the hierarchical fitness function behaves in both domains. Second, the effects of the IBC are assessed by analyzing the evolution of the learning curves. The number of evaluations has been extended from 100000 to 300000 to have a better view of the learning curves. The large plots of Figure 12 show the average best-of-generation[14] evolution of the first C_1 and second C_2 components of the fitness function. The evolution of the standard deviation is also displayed in the small plots beneath the average plots in the same figure. C_1 is the number of problems solved expanding fewer nodes than PRODIGY4.0, and C_2 is the total number of problems solved. Let us analyze the evolution of C_1 and C_2 for the blocks world (see Figures 12(a) and (c)). C_1 increases monotonically, whereas C_2 decreases initially but tends to recover later on. This is what was expected in Subsection 5.2. The 300-population configurations (thick lines) evolve more slowly than their 2-population counterparts (thin lines), although the final result is quite close. This can be seen more clearly in Figure 12(c). This behavior occurs in the logistics domain as well (see Figures 12(b) and (d)). With respect to the evolution of the standard deviation (small plots in Figure 12), there are no large differences for autonomous-EVOCK (2 and 300) configurations in both the blocks world and the logistics domain.

Figure 12 displays the effects of the IBC during learning as well. In the blocks world, Figures 12(a) and (c) show that the IBC improves the learning rate of C_1 and C_2 when compared to autonomous-EVOCK, for both population sizes (2 and 300). In the logistics domain (Figures 12(b) and (d)), this effect is less noticeable in C_1 and non-existent in C_2: C_2 goes downhill and never actually recovers. This effect is even larger for the 300-population configuration. One possible cause of this poor performance was that the left hand side of the individuals of the IB population had many meta-predicates, some of which were probably irrelevant and the system spent too much time pruning them. To solve this problem, individuals from the IB population were truncated to four conditions maximum as explained before. Curves for this truncated configuration are also displayed in Figures 12(b) and (d), again under the name of IBC4-EVOCK. It significantly improves the learning rate for C_1 and slightly for C_2, but it is still worse than autonomous-EVOCK for C_2, which helps to explain why IBC4-EVOCK does worse than autonomous-EVOCK with the testing problems. With respect to the evolution of the standard deviations, the IBC configurations seem to keep the search more focused in the blocks world (Figures 12(a) and (c)), but the opposite happens in the logistics domain (Figures 12(b) and (d)).

Figure 13 sheds some light on the IBC inner workings. It displays the evolution of the number of rules (G_4). The first noticeable effect is that G_4 grows more slowly for IBC-EVOCK than for autonomous-EVOCK. There is a strong bias to decrease the size of the individuals. Therefore, if we consider only the hill climbing configurations, the only reason for GP to add new rules is because they improve one or more of the most important components (C_1 and C_2). Otherwise, the number of rules either keeps stable or decreases. That G_4 grows slowly for IBC-EVOCK means that control rules in the IB population are frequently not useful (i.e., they do not improve C_1 or C_2). The same phenomenon can be observed in the logistics domain (see Figure 13(b)). However, once the individuals are pre-truncated (IBC4-EVOCK), GP is able to add useful rules frequently. This somewhat confirms the hypothesis that the individuals in the logistics domain IB population contain too many meta-predicates in their left hand side, making

[14] That is, the evolution of the best-of-generation individuals averaged for all GP-runs.

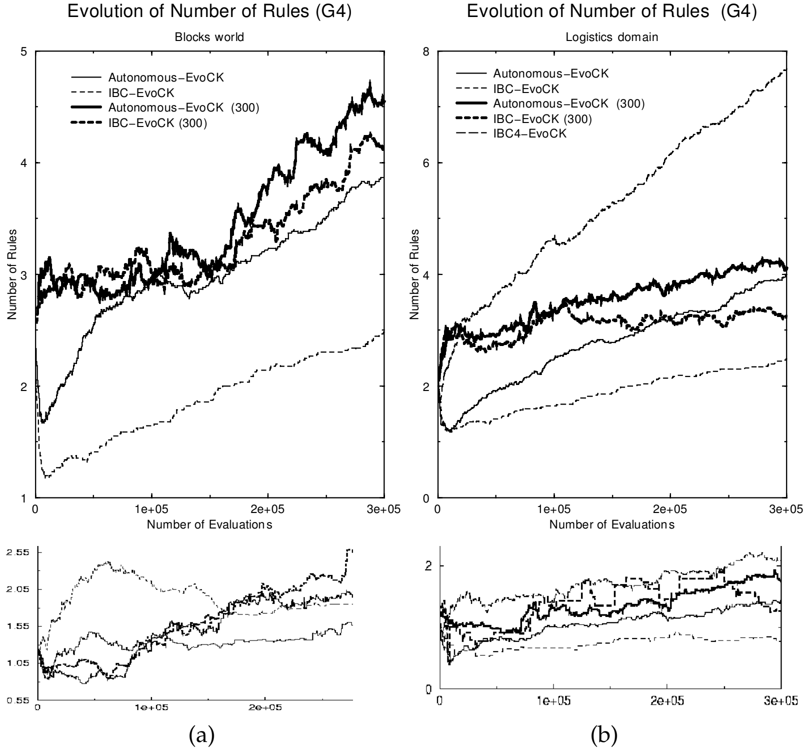


Figure 13: Evolution of component G_4 ((a) blocks world, (b) logistics domain). Large plots are averages, small plots are standard deviations.

it almost impossible to add new rules into the evolving population.

5.5 Discussion

The empirical study of the two previous subsections suggests the following remarks. First, EVOCK seems a very promising approach to planning. Results in two domains (blocks world and the logistics domain) are good: some compact heuristics are evolved that improve PRODIGY4.0 performance.

The second result concerns the IBC. Results are different for the two domains tested. In the blocks world, the IBC improves the learning rate of the GP-runs, but does not generate heuristics better than the ones produced by the non-IBC configuration. In fact, the best individuals are generated more frequently by the non-IBC configuration. It seems that the IBC leads the GP evolution to a local minimum. In the logistics domain, results by the IBC are very bad. The analysis of the learning curves suggests that the individuals in the IB population are too large because the logistics domain is more complex and requires many more predicates to describe the current planning situation (e.g., it is necessary to use predicates for the position of each one of the trucks, the planes, and the packages). Figure 14 shows such a highly complex individual.

If such individuals are pre-truncated before being used by the IBC, results im-

```
(control-rule r2
  (if (and (current-goal (at-obj ob1 a2))
           (prior-goal ((at-obj ob1 a2)))
           (true-in-state (same-city a2 po2))
           (true-in-state (same-city po2 a2))
           (true-in-state (same-city a1 po1))
           (true-in-state (same-city po1 a1))
           (true-in-state (same-city a0 po0))
           (true-in-state (same-city po0 a0))
           (true-in-state (loc-at a2 c2))
           (true-in-state (loc-at po2 c2))
           (true-in-state (loc-at a1 c1))
           (true-in-state (loc-at po1 c1))
           (true-in-state (loc-at a0 c0))
           (true-in-state (loc-at po0 c0))
           (true-in-state (part-of tr2 c2))
           (true-in-state (part-of tr1 c1))
           (true-in-state (part-of tr0 c0))
           (true-in-state (at-truck tr2 a2))
           (true-in-state (at-truck tr1 po1))
           (true-in-state (at-truck tr0 a0))
           (true-in-state (inside-truck ob0 tr1))
           (true-in-state (inside-truck ob1 tr0))
           (true-in-state (inside-truck ob2 tr1))
           (true-in-state (at-airplane pl0 a2))
           (true-in-state (at-airplane pl1 a0))
           (some-candidate-goals nil)
  (then select operators unload-truck))))
```

Figure 14: Logistics individual in the IB population.

prove, but the non-IBC configuration is still better. Some other experiments were carried out, including one where the IBC was used only 25% of times and rules were truncated to only two `true-in-state` meta-predicates. No noticeable improvement was observed. Of course, it might happen that if the GP parameters are fine-tuned and the planning domain is suitable, the IBC could be a successful operator. However, the IBC, as it stands, it is not the general and simple operator we were after. Yet, the IBC shows some positive effects (improved learning rate in the blocks world) that are worth investigating. Currently, we believe that a more clever preprocessing of the individuals in the IB population – probably involving other machine learning techniques – could be the key to successfully using the IBC.

6 Related Work

In addition to the GP approaches mentioned in Section 1, there are several machine learning approaches that learn planning heuristics. The basic approaches use analogy (Kambhampati, 1989; Veloso and Carbonell, 1993), Explanation Based Learning (EBL) (Katukam and Kambhampati, 1994; Minton and Zweben, 1993; Kambhampati, 1999), and induction (Leckie and Zukerman, 1998). Some approaches combine EBL and induction, like HAMLET (Borrajo and Veloso, 1997); or EBL and Inductive Logic Programming, like SCOPE (Estlin and Mooney, 1997; Estlin, 1998). None of them use a GP-based approach like the one we have used in this article.

In this work, we use a grammar both to generate the initial random population and to check individuals for syntactic correctness. Whigham (1996) also uses grammars, although the representation is different. Whigham's individuals are the actual parse trees, whose non-leaf 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 generative 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 are the same generative node, whereas in EvoCK's case, nodes have to be looked up in the grammar to determine whether they have been generated by the same generative symbol, and therefore can be used as crossover points. But Whigham's approach requires that the actual individuals are built before each evaluation, whereas EvoCK does not need to. Another difference with his work is that EvoCK's grammar is built on-the-fly for each different planning domain. There are several other researchers that take advantage of the flexibility of the grammar approach and use them for different purposes. For instance, Freeman (1998) uses grammars to obtain linear encodings for GP. Wong and Leung (1995, 1997) provide a very complete framework for using grammars with GP. They can be used to evolve programs in any language or for data mining. Their grammar approach has many more features than the ones that EvoCK requires.

We are not aware of other work where new material is obtained by a non-GP algorithm and injected into the population via the crossover operator, like the IBC. However, there is some work where such material is injected by seeding the initial population: Aler et al. (1998b) (the seed is obtained by HAMLET) and Langdon and Nordin (2000) (the seed is obtained by C4.5 (Quinlan, 1993)). Most of the work deals with injecting material coming from previous generations or previous GP-runs. For instance, during a GP-run, GLiB (Angeline and Pollack, 1992) can inject back previously stored parameterized subtrees to increase diversity. Louis and Zhang (1999) use worthy individuals obtained during past runs of genetic algorithms to build a case-based (CB) population. This is later used to both seed future initial populations and replace bad individuals during future GA-runs. In their work to evolve a Robocup team, Andre and Teller (1999) inject non-optimal basic subroutines as ADF's (automaticall defined functions) in the initial population. Finally, O'Reilly and Oppacher (1995) hybridize GP and hill climbing. During the hill-climbing phase, the current solution is crossed over with individuals randomly selected from the GP population or drawn from a pool of fittest individuals, with good results.

O'Reilly has studied the impact of using hill-climbing and simulated annealing on GP hierarchical variable length representations and concluded that they are feasible to solve typical GP problems (O'Reilly and Oppacher, 1994, 1996). Their work is very relevant, as our most successful experimental configurations resorted to a hill-climbing strategy and outperformed the population-based one.

7 Conclusions and Future Work

The main contribution of this article is to show that GP, when suitably adapted, can be used for evolving heuristics for planning. This approach is more feasible and efficient than current GP approaches to planning. It is more feasible than using GP to build a planner because it does not have to build a planner from scratch but can take advantage of the base planner, which can itself use powerful heuristics. It is more efficient than using GP to search plan space because once the heuristics have been learned, they can be used to solve a whole class of different planning problems in a planning domain, instead of running GP for every new planning problem. The basic intuition here is that problems of different size in a domain can use the same heuristics. For instance, solving a 50-block problem in the blocks world should need approximately the same heuristics as solving a 49-block problem. And perhaps these heuristics can be useful for much larger problems as well. We have adapted GP to heuristic learning by

adding new operators that are more appropriate to work with heuristics represented as sets of control rules. The resulting system – EVOCK – obtains good results in two widely used planning domains: the blocks world and the logistics domain. In these two domains, PRODIGY4.0 cannot solve a reasonable number of problems, specifically those with many goals and objects. We show how this behavior can be improved with a GP approach. The problems we are solving in testing time are hard enough that other modern planners, like UCPOP and GRAPHPLAN, cannot solve them either.

Additionally, we have experimented with a new genetic operator – the Instance-Based Crossover (IBC) – that is able to use traces of the base planner as raw genetic material to be injected to the evolving population. This is achieved by initially solving some planning problems, obtaining the traces the planning system provides, analyzing these traces to produce heuristics that could help to solve similar problems, and converting them to GP individuals. The IBC is unable to improve testing results and gets worse results in the logistics domains. However, some of its positive effects (improved learning rate in the blocks world) are interesting enough to deserve further investigation. However, the IBC, as it stands, it is not the general and simple operator we were after. Currently, we believe that a more sophisticated preprocessing of the individuals in the IB population – probably involving other machine learning techniques – could be the key to successful use of the IBC. Also, some more clever mechanism to decide when and in what proportion IB individuals should be injected into the population is needed.

One of the chief difficulties in using our system is the large number of fitness cases needed to evolve heuristics. In order to evaluate each individual with each fitness case, a call to the planner PRODIGY4.0 must be made. It would be useful if EvoCK could determine which fitness cases are more appropriate at different stages in the evolutionary process. We believe that co-evolution techniques (Berlanga, 2000) and dynamic training subset selection policies (Gathercole and Ross, 1994) could be used for that purpose.

In this article, we have used a multi-objective technique known as a hierarchical fitness function. This forced us to add an additional fitness component to measure efficiency so that the system would find useful new rules. However, optimizing this first component might lead the system to fall into local minima for the second component and never recover. In the experiments we have carried out, this behavior only occurs with some of the IBC configurations in one of the domains. Perhaps using Pareto optimization techniques, where selection of the actual best individual is deferred until the end of the run, could be used to solve this problem. However, maintaining the Pareto front would require more computer resources. Therefore, we will try to reduce fitness evaluation effort before attempting to use Pareto techniques.

Finally, although we have focused on STRIPS-planning, we would like to extend our approach to evolving heuristics for other search problems. Actually, GP itself can be considered as a search process. For instance, many of the problems addressed by the GP community belong to the symbolic regression variety. This kind of problem can be reformulated as a search in the space of regression programs. Search is what GP does, but in a generic search scenario, the search operators need not be (only) the genetic operators. The goal of such search is to find a regression program that minimizes error in a predefined set of training cases. Now, many such symbolic regression problems can be varied from simpler problems to more difficult problems, just like in the blocks world, problem difficulty can go from 3 blocks to 50 blocks. Actually, all of the problems described in Koza (1994) are of this kind. Problems like even-parity, the bumblebee, etc. are rather classes of problems of different difficulties. It may be possi-

ble to learn heuristics using the simpler problems as fitness cases that could be used to guide the search through program space to solve tougher problems, which is what we have done in this article to solve planning problems. For instance, heuristics learned to find programs like even-parity-3 and -4 might be useful to find programs for even-parity-10. Of course, finding generic programs is more difficult than finding plans, and unfortunately, there are no powerful planners for finding even-parity-x programs.

References

Aler, R., Borrajo, D., and Isasi, P. (1998a). Evolving heuristics for planning. *Lecture Notes in Computer Science*, 1447:745–754.

Aler, R., Borrajo, D., and Isasi, P. (1998b). Genetic programming and deductive-inductive learning: A multistrategy approach. In Shavlik, J., editor, *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 10–18, Morgan Kaufmann, San Francisco, California.

Aler, R., Borrajo, D., and Isasi, P. (2001). Grammars for learning control knowledge with gp. In Greenwood, G., editor, *Proceedings of the Conference on Evolutionary Computation*, pages 1220–1227, IEEE Press, Piscataway, New Jersey.

Andre, D. and Teller, A. (1999). Evolving Team Darwin United. *Lecture Notes in Computer Science*, 1604:346–351.

Angeline, P. J. and Pollack, J. B. (1992). The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241, Lawrence Erlbaum, Hillsdale, New Jersey.

Bennett III, F. H. (1996). Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In Koza, J. R. et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 30–38, MIT Press, Cambridge, Massachusetts.

Berlanga, A. (2000). *Dinámica evolutiva/involutiva para la obtención de soluciones generales en computación evolutiva (Evolutive/Involutive Dynamics for obtaining general solutions in evolutionary computing)*. Ph.D. thesis, Computer Science, Universidad Carlos III de Madrid, Madrid, Spain.

Blum, A. L. and Furst, M. L. (1995). Fast planning through planning graph analysis. In Mellish, C. S., editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1636–1642, Morgan Kaufmann, San Francisco, California.

Bonet, B. and Geffner, H. (1999). Planning as heuristic search: New results. *Lecture Notes in Computer Science*, 1809:360–372.

Borrajo, D. and Veloso, M. (1997). Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning*, 11(1-5):371–405.

Borrajo, D., Vegas, S., and Veloso, M. (2001). Quality-based learning for planning. In *Working notes of the International Joint Conference on Artificial Intelligence Workshop on Planning with Resources*, pages 9–17, IJCAI Press.

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204.

Cohen, P. R. (1995). *Empirical Methods for Artificial Intelligence*. Massachusetts Institute of Technology, Cambridge, Massachusetts.

Estlin, T. A. (1998). *Using Multi-Strategy Learning to Improve Planning Efficiency and Quality*. Ph.D. thesis, Department of Computer Sciences, University of Texas, Austin, Texas.

Estlin, T. A. and Mooney, R. J. (1997). Learning to improve both efficiency and quality of planning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1227–1233, Morgan Kaufmann, San Francisco, California.

Fikes, R. and Nilsson, N. (1971). A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

Fink, E. and Veloso, M. (1996). Formalizing the PRODIGY planning algorithm. In Ghallab, M. and Milani, A., editors, *New Directions in AI Planning*, pages 261–272. IOS Press, Amsterdam, The Netherlands.

Freeman, J. J. (1998). A linear representation for GP using context free grammars. In Koza, J. R. et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 72–77, Morgan Kaufmann, San Francisco, California.

Gathercole, C. and Ross, P. (1994). Some training subset selection methods for supervised learning in genetic programming. *Lecture Notes in Computer Science*, 866:312–321.

Gupta, N. and Nau, D. (1992). On the complexity of blocks-world planning. *Artificial Intelligence*, 56:223–254.

Handley, S. G. (1994). The automatic generations of plans for a mobile robot via genetic programming with automatically defined functions. In Kinnear, K. E., editor, *Advances in Genetic Programming*, chapter 18, pages 391–407. MIT Press, Cambridge, Massachusetts.

Kambhampati, S. (1989). *Flexible Reuse and Modification in Hierarchical Planning: A Validation Structure Based Approach*. Ph.D. thesis, Computer Vision Laboratory, Center for Automation Research, University of Maryland, College Park, Maryland.

Kambhampati, S. (1999). Improving Graphplan's search with EBL & DDB techniques. In Dean, T., editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 982–987, Morgan Kaufmann, San Francisco, California.

Katukam, S. and Kambhampati, S. (1994). Learning explanation-based search control rules for partial order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 582–587, AAAI Press, Menlo Park, California.

Koza, J. R. (1989). Hierarchical genetic algorithms operating on populations of computer programs. In Sridharan, N. S., editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, volume 1, pages 768–774. Morgan Kaufmann, San Francisco, California.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts.

Koza, J. R. (1995). Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. In McDonnell, J. R., Reynolds, R. G., and Fogel, D. B., editors, *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 695–717, MIT Press, Cambridge, Massachusetts.

Koza, J. R. and Andre, D. (1995). Evolution of both the architecture and the sequence of work-performing steps of a computer program using genetic programming with architecture-altering operations. In Siegel, E. V. and Koza, J. R., editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 50–60, AAAI press, Menlo Park, California.

Koza, J. R. et al. (1999). *Genetic Programming III*. Morgan Kaufmann, San Francisco, California.

Langdon, W. B. (1995). Pareto, population partitioning, price and genetic programming. Research Note RN/95/29, University College London, London, UK.

Langdon, W. B. and Nordin, J. P. (2000). Seeding GP populations. *Lecture Notes in Computer Science*, 1802:304–315.

Leckie, C. and Zukerman, I. (1998). Inductive learning of search control rules for planning. *Artificial Intelligence*, 1(2):63–98.

Louis, S. J. and Zhang, Y. (1999). A sequential similarity metric for case injected genetic algorithms applied to TSPs. In Banzhaf, W. et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 377–384, Morgan Kaufmann, San Francisco, California.

McDermott, D. (1999). Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109:1–361.

Minton, S. and Zweben, M. (1993). Learning, planning and scheduling: An overview. In Minton, S., editor, *Machine Learning Methods for Planning*, chapter 8. Morgan Kaufmann, San Francisco, California.

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

Muslea, I. (1997). SINERGY: A linear planner based on genetic programming. In Steel, S., editor, *Recent Advances in AI Planning. Fourth European Conference on Planning*, number 1348 in Lecture Notes in Artificial Intelligence, pages 312–324, Springer-Verlag, Berlin, Germany.

O'Reilly, U.-M. and Oppacher, F. (1994). Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. Technical Report 94-04-021, Santa Fe Institute, Santa Fe, New Mexico.

O'Reilly, U.-M. and Oppacher, F. (1995). Hybridized crossover-based search techniques for program discovery. Technical Report 95-02-007, Santa Fe Institute, Santa Fe, New Mexico.

O'Reilly, U.-M. and Oppacher, F. (1996). A comparative analysis of GP. In Angeline, P. J. and Kinnear, K. E., editors, *Advances in Genetic Programming 2*, chapter 2, pages 23–44. MIT Press, Cambridge, Massachusetts.

Penberthy, J. S. and Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In Nebel, B., Rich, C., and Swartout, W., editor, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114, Morgan Kaufmann, San Francisco, California.

Pérez, M. A. and Carbonell, J. G. (1994). Control knowledge to improve plan quality. In Hammond, K. J., editor, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 323–328, AAAI Press, Menlo Park, California.

Quinlan, R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, California.

Slaney, J. and Thiebaux, S. (2001). Blocks world revisited. *Artificial Intelligence*, 125:119–153.

Spector, L. (1994). Genetic programming and AI planning systems. In *Proceedings of Twelfth National Conference on Artificial Intelligence*, pages 1329–1334, MIT Press, Cambridge, Massachusetts.

Veloso, M. (1994). *Planning and Learning by Analogical Reasoning*. Springer Verlag, Berlin, Germany.

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

Veloso, M. et al. (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical AI*, 7:81–120.

Westerberg, C. H. and Levine, J. (2000). Genplan: Combining genetic programming and planning. In *Proceedings of Nineteenth Workshop of the UK Planning and Scheduling Special Interest Group*, pages 270–281.

Whigham, P. A. (1996). Search bias, language bias, and genetic programming. In Koza, J. R. et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 230–237, MIT Press, Cambridge, Massachusetts.

Wong, M. L. and Leung, K. S. (1995). Combining genetic programming and inductive logic programming using logic grammars. In *1995 IEEE Conference on Evolutionary Computation*, volume 2, pages 733–736, IEEE Press, Piscataway, New Jersey.

Wong, M. L. and Leung, K. S. (1997). Evolutionary program induction directed by logic grammars. *Evolutionary Computation*, 5(2):143–180.