

---

# Evolutionary Concept Learning

---

Federico Divina and Elena Marchiori

Computer Science Dept.

Vrije Universiteit, Amsterdam, The Netherlands

{divina,elena}@cs.vu.nl

## Abstract

Inductive learning in First-Order Logic (FOL) is a hard task due to both the prohibitive size of the search space and the computational cost of evaluating hypotheses. This paper introduces an evolutionary algorithm for concept learning in (a fragment of) FOL. The algorithm evolves a population of Horn clauses by repeated selection, mutation and optimization of more fit clauses. Its main novelty, with respect to previous approaches, is the use of stochastic search biases for reducing the complexity of the search process and of the clause fitness evaluation. An experimental evaluation of the algorithm indicates its effectiveness in learning short hypotheses of satisfactory accuracy in a short amount of time.

## 1 Introduction

Learning from examples in FOL, also known as Inductive Logic Programming (ILP) (Muggleton & Raedt, 1994), constitutes a central topic in Machine Learning, with relevant applications to problems in complex domains like natural language and molecular computational biology (Muggleton, 1999).

Learning can be viewed as a search problem in the space of all possible hypotheses (Mitchell, 1982). Given a FOL description language used to express possible hypotheses, a background knowledge, a set of positive examples, and a set of negative examples, one has to find a hypothesis which covers all positive examples and none of the negative ones (cf. (Kubat et al., 1998; Mitchell, 1997)).

This problem is NP-hard even if the language to represent hypotheses is propositional logic. When FOL

hypotheses are used, the complexity of searching is combined with the complexity of evaluating hypotheses (Giordana & Saitta, 2000).

Popular FOL learners like FOIL (Quinlan, 1990) and Progol (Muggleton, 1995) adopt a progressive coverage approach. One starts with a training set containing all positive and negative examples, construct a FOL (if-then) rule which covers some of the positive examples, removes the covered positive examples from the training set and continues with the search for the next rule. When the process terminates (after a maximum number of iterations or when all positive examples have been covered), the resulting set of rules is reviewed, e.g., to eliminate redundant rules. These algorithms use different greedy methods as well as heuristics (e.g. information gain) to cope with the complexity of the search.

FOL learners based on genetic algorithms act on more clauses at the same time. Systems like GIL (Janikow, 1993), GLPS (Leung & Wong, 1995) and STEPS (Kennedy & Giraud-Carrier, 1999) use an encoding where a chromosome represents a set of rules. In other GA based systems like SIA01 (Augier et al., 1995), REGAL (Giordana & Neri, 1996), G-NET (Anglano et al., 1998) and DOGMA (Hekanaho, 1998), a chromosome represents a clause. In the latter case a non redundant hypothesis is extracted from the final population at the end of the evolutionary process. Both approaches present advantages and drawbacks. Encoding a whole hypothesis in each chromosome allows an easier control of the genetic search but introduces a large redundancy, that can lead to populations hard to manage and to individuals of enormous size. Encoding one clause in each chromosome allows for cooperation and competition between different clauses hence reduces redundancy, but requires sophisticated strategies, like co-evolution, for coping with the presence in the population of super-individuals.

This paper introduces an evolutionary algorithm which evolves a set of chromosomes representing clauses, where at each iteration fitter chromosomes are selected, mutated, and optimized. The main novelty with respect to previous approaches is the introduction of two stochastic mechanisms for controlling the complexity of the construction, optimization and evaluation of clauses. The first mechanism allows the user to specify the percentage of background knowledge that the algorithm will use, in this way controlling the computational cost of fitness evaluation. The second mechanism allows one to control the greediness of the operators used to mutate and optimize a clause, thus controlling the computational cost of the search.

Furthermore we introduce and test a variant of the Universal Suffrage (US) selection operator ((Giordana & Neri, 1996)), called Weighted Universal Suffrage (WUS) selection operator. The US selection operator is based on the idea that individuals are candidates to be elected, and positive examples are the voters. Every positive example has the same voting power. The idea behind the WUS selection operator, is to give more voting power to examples that are harder to cover. The voting power of examples is adjusted during the computation.

We show experimentally that the algorithm is able to find hypotheses of satisfactory quality, both with respect to accuracy and simplicity, in a short time.

## 2 The Learning Algorithm

The algorithm considers Horn clauses of the form

$$p(X, Y) \leftarrow r(X, Z), q(Y, a).$$

consisting of atoms whose arguments are either variables (e.g.  $X, Y, Z$ ) or constants (e.g.  $a$ ). The atom  $p(X, Y)$  is called head, and the set of other atoms is called body. The head describes the target concept, and the predicates of the body are in the background knowledge.

The background knowledge contains ground facts (i.e. clauses of the form  $r(a, b) \leftarrow$  . with  $a, b$  constants). The training set contains facts which are true (positive examples) and false (negative examples) for the target predicate. A clause is said to *cover an example* if the theory formed by the clause and the background knowledge logically entails the example.

A clause has a declarative interpretation (universally quantified FOL implication)

$$\forall X, Y, Z (r(X, Z), q(Y, a) \rightarrow p(X, Y))$$

and a procedural one

in order to solve  $p(X, Y)$  solve  $r(X, Z)$  and  $q(Y, a)$ .

Thus a set of clauses forms a logic program, which can directly (in a slightly different syntax) be executed in the programming language Prolog. So the goal of the learning algorithm can be rephrased as finding a logic program that models a given target concept, given a set of training examples and a background knowledge.

The overall algorithm we introduce, called ECL (Evolutionary Concept Learner), is illustrated in pseudo-code in the figure below.

ALGORITHM ECL

```

Sel = positive_examples
repeat
  Select partial Background Knowledge
  Population = Initial_pop
  while (not terminate) do
    Select n chromosomes using Sel
    for each selected chromosome chrom
      Mutate chrom
      Optimize chrom
      Insert chrom in Population
    end for
  end while
  Store Population in Final_Population
  Sel = Sel - { positive examples
               covered by clauses in Population }
until max_iter is reached
Extract final theory from Population

```

In the repeat statement the algorithm constructs iteratively a `Final_population` as the union of `max_iter` populations. At each iteration, part of the background knowledge is chosen using a stochastic search bias described below.

A `Population` is evolved by the repeated application of selection, mutation and optimization (the `while` statement). These operators use only the chosen part of background knowledge.

At each generation of the evolution,  $n$  clauses are selected by means of the Universal Suffrage (US) selection operator (Neri & Saitta, 1995), a powerful selection mechanism used for achieving species formation in GAs for concept learning. US selection chooses randomly a positive example from the set `Sel` of positive examples yet not covered by clauses in the actual `Final_population`, and performs a roulette wheel selection on those clauses of the `Population` which cover that example. If the example is not yet covered by

any clause, a new clause is constructed for that example using a seeding operator. The selected clause is then modified using the mutation and optimization operators, and is inserted in the population.

When the construction of the `Final_population` is completed, a logic program is extracted using a set covering algorithm.

Before presenting the main steps of ECL, we describe the stochastic search biases.

## 2.1 Stochastic Search Biases

ECL uses two stochastic mechanisms, one for selecting part of the background knowledge, and one for selecting the degree of greediness of the operators used in the evolutionary process.

A parameter  $p$  ( $p$  real number in  $(0, 1]$ ) is used in a simple stochastic sampling mechanism which selects an element of the background knowledge with probability  $p$ . In this way the user can limit the cost of the search and fitness evaluation by setting  $p$  to a low value. This because only a part of the background knowledge will be used when assessing the goodness of an individual. This leads to the implicit selection of a subset of the examples (only those examples that can be covered with the partial background knowledge selected will be considered). Individuals will be evaluated on these examples using only the partial background knowledge. In this way an individual can be wrongly evaluated because a subset of the examples is used, and also because those examples can be wrongly classified, in case they are covered using the whole background knowledge, but are not covered using the partial background knowledge. This is different from other mechanisms used for improving the efficiency of fitness evaluation, like (Glover & Sharpe, 1999), (Teller & Andre, 1997), where training set sampling is employed for speeding up the evaluation of individuals.

The construction, mutation and optimization of a clause use four greedy generalization/specialization operators (described later in an apart section). Each greedy operator involves the selection of a set of constants (or of a set of variables). The size of this set can be supplied by the user by setting a corresponding parameter  $Ni$  ( $i = 1, \dots, 4$ ). The elements of the set are then randomly chosen with uniform probability. In this way the user can control the greediness of the operators, where higher values of the parameters imply higher greediness.

Finally ECL uses also a language bias which is commonly employed in ILP systems for limiting explicitly the maximum length of clauses.

These search biases allow one to reduce the cost of both the search and fitness evaluation, but the price to pay may be the impossibility of finding the best clauses.

## 2.2 Fitness and Representation

The quality of a clause  $cl$  is measured by the following fitness function:

$$fitness(cl) = \frac{pos-pos_{cl}}{pos} + w \cdot \frac{neg_{cl}}{neg}$$

The aim of ECL is to evolve clauses with minimum fitness, that is which cover many positive examples and few negative ones. In the above formula  $pos$  and  $neg$  are respectively the total number of positive and negative training examples,  $pos_{cl}$ ,  $neg_{cl}$  are the number of positive and negative examples covered by the clause  $cl$ , and  $w$  is a weight used to favor clauses covering few negative examples. The weight  $w$  is used to deal with skewed distributions of the examples, where a high weight is used when there are much more positive examples than negative ones.

ECL uses a high level representation similar to the one used by SIA01 (Augier et al., 1995), where a clause  $p(X, Y) \leftarrow r(X, Z), q(Y, a)$  is described by the sequence

$$\boxed{p, X, Y}, \boxed{r, X, Z}, \boxed{q, Y, a}$$

This representation is preferred to other GA typical representations, like bit string, because it allows the direct use of ILP operators for generalization and specialization of a clause. Moreover, it does not constraint the length of a chromosome, like e.g in the bitwise representation used in the REGAL and G-NET systems, which requires the user to specify an initial template for the target predicate.

## 2.3 Clause Construction

A clause  $cl$  is constructed when the US selection operator selects a positive example which is not yet covered by any clause in the actual population. This example is used as seed in the following procedure, where  $BK_p$  denotes the chosen part of background knowledge.

1. The selected example becomes the head of the emerging clause;
2. Construct two sets  $A_{cl}$  and  $B_{cl}$ .  $A_{cl}$  consists of all atoms in  $BK_p$  having *at most* one argument which does *not* occur in the head;  $B_{cl}$  contains all elements in  $BK_p \setminus A_{cl}$  having at least one argument occurring in the head.

3. **while**  $length(cl) < l$  and  $A_{cl} \cup B_{cl} \neq \emptyset$ 
  - (a) Randomly select an atom from  $A_{cl}$  and remove it from  $A_{cl}$ . If  $A_{cl}$  is empty then randomly select an atom from  $B_{cl}$  (and remove it from  $B_{cl}$ ). Add the selected atom to the emerging clause  $cl$ .
4. Generalize  $cl$  as much as possible by means of the repeated application of the generalization operator ‘constant into variable’ (described in the next section). Apply this operator to the clause until either its fitness increases or a maximum number of iterations is reached. In the former case, retract the last application of the generalization operator.

In step 3  $l$  is the maximum length of a clause, supplied by the user. If  $l$  was not supplied then the first condition of the *while* cycle is dropped, and no constraint on the length of the clause is imposed.

## 2.4 Selection

The US selection operator, first introduced in (Giordana & Neri, 1996), selects clauses in two steps:

1. randomly select  $n$  examples from the positive examples set;
2. for each selected example  $e_i$ ,  $1 \leq i \leq n$ , let  $Cov(e_i)$  be the set of clauses in the current population that cover  $e_i$ . If  $Cov(e_i) \neq \emptyset$ , choose one clause from  $Cov(e_i)$  using a roulette wheel mechanism, where the sector associated with the clause  $c \in Cov(e_i)$  is proportional to the ratio between the fitness of  $c$  and the sum of the fitness of all the clauses occurring in  $Cov(e_i)$ . If  $Cov(e_i) = \emptyset$  create a new clause covering  $e_i$ , using  $e_i$  as a seed.

When introduced, in (Giordana & Neri, 1996), the US selection operator was used in a distributed system, made of various genetic nodes, where each genetic node performs a GA. In that setting, the examples assigned to each node were different, and the training sets changed during the computation. However at the GA level the examples were the same, and had the same probability of being selected.

We propose here the following variant of the US selection, called *Weighted US selection*, where examples have different probability of selection. A weight is associated to each example, where smaller weights are associated to examples harder to cover. Then the random selection used in step 1 of the US selection above is replaced by a selection which takes into account the weights of examples.

More in detail, the weight of an example  $e$  is equal to

$$\frac{|Cov(e)|}{|Pop|}$$

being  $Pop$  the current population and  $Cov(e)$  the set of clauses of  $Pop$  that cover  $e$ . If the population is empty, then every example has the same weight.

The examples are then selected with a roulette wheel mechanism, where the dimension of the sector associated to each examples is inversely proportional to the weight of the example. So the less clauses cover an example, the more chances that example has of being selected. The weights of the examples are updated at every iteration. Once the examples have been selected, the selection of the clauses is made as in the standard US selection operator.

With this mechanism not only uncovered examples are favored, but also examples that are covered by few clauses are favored, having wider sector in the roulette wheel. Examples covered by many clauses are penalized, because easier to cover. In this way the system will focus at each iteration more and more on the harder examples to be covered.

## 2.5 Mutation and Optimization

The mutation consists of the application of one of the four generalization/specialization operators. This operator is chosen as follows. First, a (randomized) test decides whether it will be a generalization or a specialization operator. Next, one of the two operators of the chosen class is randomly applied. The first test is based on the completeness and the consistency of the selected individual. If the individual is consistent with the training set, then it is likely that the individual will be generalized. Otherwise it is more probable that the individual will be specialized. The test decides to generalize a clause  $cl$  with probability

$$p_{gen}(cl) = \frac{1}{2} \left( \frac{pos_{cl} - neg_{cl}}{pos + neg} + \alpha \right)$$

otherwise it decides to specialize the clause. The constant  $\alpha$  is used to slightly bias the decision toward generalization. The probability  $p_{gen}$  is maximal when  $cl$  covers all positive and no negative examples, and it is minimal in the opposite case.

The optimization phase consists of a repeated application of the greedy operators to the selected individual, until either its fitness does not increase or a maximum number of iterations is reached.

The system does not make use of any crossover operator. Experiments with a simple crossover operator,

which uniformly swaps atoms of the body of the two clauses, have been conducted. However the results did not justify its use.

## 2.6 Hypothesis Extraction

The termination condition of the main **while** statement of ECL is met when either all positive examples are covered or a maximum number of iterations is reached. In this case a logic program for the target predicate is extracted from the final population. The theory has to cover as many positive examples as possible, and as few negative ones (notice that at this stage the clauses have been “globally” evaluated, using the complete background knowledge). This problem can be translated into an instance of the weighted set covering problem as follow. Each element  $cl$  of the final population is a column with positive weight equal to

$$weight_{cl} = neg_{cl} + 1$$

and each covered positive example is a row. The columns relative to each positive example are the clauses that cover that example. In this way clauses covering few negative examples are favored. A fast heuristic algorithm ((Marchiori & Steenbeek, 2000)) is applied to this problem instance to find a “best” theory.

## 3 Generalization/Specialization Operators

A clause  $cl$  is generalized either by deleting an atom from the body of the clause or by replacing (all occurrences of) a constant with a variable. Dually,  $cl$  is specialized by either adding an atom to the body of  $cl$ , or by replacing (all occurrences of) a variable with a constant.

The four operators utilize four parameters  $N_1, \dots, N_4$ , respectively, in their definition, and a gain function. When applied to operator  $\tau$  and clause  $cl$ , the gain function yields the difference between the clause fitness before and after the application of  $\tau$ :

$$gain(cl, \tau) = fitness(cl) - fitness(\tau(cl))$$

The four operators are defined below.

### 3.1 Atom Deletion

Consider the set  $Atm$  of  $N_1$  atoms of the body of  $cl$  randomly chosen. For each  $A$  in  $Atm$ , compute  $gain(cl, -A)$ , the gain of  $cl$  when  $A$  is deleted from  $cl$ .

Choose an atom  $A$  yielding the highest gain  $gain(cl, -A)$  (ties are randomly broken), and generalize  $cl$  by deleting  $A$  from its body.

Insert the deleted atom  $A$  in a list  $D_{cl}$  associated with  $cl$  containing atoms which have been deleted from  $cl$ . Atoms from this list may be added to the clause during the evolutionary process by means of a specialization operator.

### 3.2 Constant into Variable

Consider the set  $Var$  of variables present in  $cl$  plus a new variable. Consider also the set  $Con$  consisting of  $N_2$  constants of  $cl$  randomly chosen.

For each  $a$  in  $Con$  and each  $X$  in  $Var$ , compute  $gain(cl, \{a/X\})$ , the gain of  $cl$  when all occurrences of  $a$  are replaced by  $X$ .

Choose a substitution  $\{a/X\}$  yielding the highest gain (ties are randomly broken), and generalize  $cl$  by applying  $\{a/X\}$ .

### 3.3 Atom Addition

Consider the set  $Atm$  consisting of  $N_3$  atoms of  $B_{cl}$  (list built at initialization time) and of  $N_3$  atoms of  $D_{cl}$ , all randomly chosen.

For each  $A$  in  $Atm$  compute  $gain(cl, +A)$ , the gain of  $cl$  when  $A$  is added to the body of  $cl$ .

Choose an atom  $A$  yielding the highest gain  $gain(cl, +A)$  (ties are randomly broken), and specialize  $cl$  by adding  $A$  to its body.

Remove  $A$  from its original list ( $B_{cl}$  or  $D_{cl}$ ).

### 3.4 Variable into Constant

Consider the set  $Con$  consisting of  $N_4$  constants (of the problem language) randomly chosen, and a variable  $X$  of  $cl$  randomly chosen.

For each  $a$  in  $Con$ , compute  $gain(cl, \{X/a\})$ , the gain of  $cl$  when all occurrences of  $X$  are replaced by  $a$ .

Choose a substitution  $\{X/a\}$  yielding the highest gain (ties are randomly broken), and specialize  $cl$  by replacing all occurrences of  $X$  with  $a$ .

## 4 Experimental Evaluation

We consider three datasets for experimenting with ECL: the vote, credit and mutagenesis dataset, respectively. The three dataset are public domain datasets.

The vote dataset contains votes for each of the U.S. House of Representatives Congressmen on the sixteen key votes. The problem is learning a concept for distinguishing between democratic and republican congressmen. The dataset consists in 435 instances, of which 267 are examples of democrats, and 168 are republicans.

The credit dataset concerns credit card applications. The problem consists in learning when to allow a subject to have a credit card or not. There are 690 instances, of which 307 are positive instances and 383 are negative instances. Each instance is described by fifteen attributes. These first two datasets are taken from (Blake & Merz, 1998).

The mutagenesis dataset comes from the field of organic chemistry, and concerns the problem of learning the mutagenic activity of nitroaromatic compounds. These compounds occur in automobile exhaust fumes and are also common intermediates in the synthesis of many thousands of industrial compounds (Debnath et al., 1991). Highly mutagenic nitroaromatics have been found to be carcinogenic (Ashby & Tennant, 1991). The concept to learn is expressed by the predicate  $active(C)$ , which states that compound  $C$  has mutagenic activity. The dataset originates from (Debnath et al., 1991).

The parameter settings used in the experiments are given in Table 1.

|            | Vote      | Credit    | Mutagenesis |
|------------|-----------|-----------|-------------|
| pop_size   | 80        | 20        | 50          |
| mut_rate   | 1         | 1         | 1           |
| n          | 10        | 2         | 7           |
| max_gen    | 5         | 30        | 10          |
| max_iter   | 2         | 10        | 10          |
| N(1,2,3,4) | (3,6,2,5) | (2,5,2,5) | (4,8,2,8)   |
| p          | 0.1       | 0.2       | 0.1         |
| l          | 4         | 4         | 8           |

Table 1: Parameter settings: pop\_size = maximum size of population, mut\_rate = mutation rate, n = number of selected clauses, max\_gen = maximum number of GA generations, max\_iter = maximum number of iterations, N(1,2,3,4) = parameters of the four greedy operators, p = parameter of BK selection, l = maximum length of a clause

These values have been obtained after a few experiments on the training sets, with the constraint that a run of ECL would take at most 1 hour. As expected, the values found depend on the specific dataset. Unfortunately, we were unable to find general rules that

could explain the choice of these parameters. This is in general a challenging problem (Eiben et al., 1999), and we are actually investigating methods for the on-line adaptation of parameters.

The evaluation method used is ten-fold cross validation. Each data set is divided in ten disjoint sets of similar size; one of these sets is used as test set, and the union of the remaining nine forms the training set. Then ECL is run on the training set and it outputs a logic program, whose performance on new examples is assessed using the test set.

This process is repeated 10 times, using each time a different set as test set. The average of all the results is taken as final evaluation measure for ECL.

We consider three performance measures:

- efficiency: the running time of the algorithm on the training set for finding the logic program;
- simplicity: the number of clauses of the logic program;
- accuracy: the proportion of examples in the test set which have been correctly classified by the resulting logic program.

| System | Vote         | Credit       | Mutagenesis  |
|--------|--------------|--------------|--------------|
| G-NET  | 0.95 (0.032) | 0.84 (0.044) | 0.91 (0.079) |
| C4.5   | 0.95 (0.030) | 0.86 (0.033) | n.a.         |
| Progol | -            | -            | 0.80 (0.030) |
| ECL    | 0.94 (0.023) | 0.79 (0.072) | 0.87 (0.056) |

Table 2: Accuracy results obtained using ten-fold cross validation. Standard deviation is given between brackets.

Results obtained by ECL are compared to results obtained by three of the most effective concept learners based on different approaches in table 2. C4.5 is based on decision trees, Progol employs a progressive coverage method, and G-NET is a distributed co-evolutionary genetic algorithm.

The results for the first three systems are taken from (Anglano et al., 1998), while the result of Progol is taken from (Srinivasan et al., 1994). All the results were obtained using the same ten-fold cross validation. On the vote dataset the results obtained by ECL are comparable to those obtained with the other three systems. The results on the credit dataset are worse than those of G-NET and C4.5.

Table 3 presents the results obtained on the three datasets using the parameter  $p$  set to one. This means

| Dataset     | Accuracy     | Efficiency | Simplicity |
|-------------|--------------|------------|------------|
| Vote        | 0.94 (0.033) | 66 min     | 5.89       |
| Credit      | 0.71 (0.074) | 224min     | 41.1       |
| Mutagenesis | 0.81 (0.089) | 81 min     | 16         |

Table 3: Results obtained by ECL using the same parameters shown in table 1, but here  $p$  is set to 1, so that the whole background knowledge is used.

| Dataset            | Efficiency | Simplicity |
|--------------------|------------|------------|
| Vote (ECL)         | 29 minutes | 5          |
| Vote (G-NET)       | -          | 2          |
| Credit (ECL)       | 50 minutes | 5          |
| Credit (G-NET)     | -          | 14         |
| Mutagenesis (ECL)  | 10 minutes | 4          |
| Mutagenesis (GNET) | few hours  | 3          |

Table 4: Efficiency = average running time, Simplicity = average number of clauses for ECL, and of disjuncts for G-NET.

that the whole background knowledge will be used. The other parameters are the ones defined in table 1. It can be seen that the results are not better than the results shown in table 2, especially in the mutagenesis dataset. This is probably due to overfitting that can take place when too much information about the problem to tackle is present. Moreover, as expected, the system slows down sensibly when using the whole background knowledge.

Table 4 shows the average time taken by a run on each dataset. Table 4 shows that even if in some cases other systems outperform ECL, ECL is able to find, in a short amount of time, a simple result with a satisfactory accuracy. For instance, on the mutagenesis dataset ECL is able to find a simple logic program with 4 clauses in just 10 minutes (on the average). In contrast, as mentioned in (Anglano et al., 1998), G-NET, which is a distributed system working on a cluster of workstations, needs few hours for finding a theory of comparable simplicity, which is not much more accurate. Unfortunately detailed results on the execution time of G-NET were not available, and also replicating the experiments resulted not possible, due to the impossibility to install the system<sup>1</sup>.

#### 4.1 US vs. Weighted US Selection

Table 5 reports some results in which the US and the WUS selection operators are compared. It can be seen

<sup>1</sup>Thanks to F. Neri for his support.

| Operator    | WUS   | US    |
|-------------|-------|-------|
| Vote        | 0.941 | 0.882 |
| Credit      | 0.790 | 0.795 |
| Mutagenesis | 0.872 | 0.860 |

Table 5: Average accuracy results obtained on the three dataset, with US and weighted US selection operator.

seen that the use of the WUS selection operator improves the accuracy of the system for the vote and the mutagenesis datasets. In particular in the vote dataset the difference is evident. For the credit dataset the use of the WUS selection operator does not lead to any improvement. Even if the results of the experiments do not indicate a dramatic benefit of the WUS selection operator over the US one, it does not affect the efficiency of the system hence it can be used as alternative selection mechanism.

## 5 Conclusion

In this paper we presented a concept learning algorithm based on evolutionary computation, which incorporates novel simple parametric mechanisms for controlling the cost of searching the hypotheses space and the cost of fitness evaluation. We also introduced a variant of the US selection operator, called Weighted US selection operator.

The algorithm can be used profitably for exploring efficiently a new learning problem to get a first rough idea of possible simple models of the target concept, or for experimenting with a range of different search strategies at the same time, including random search and hill climbing as bounds of the range, which can be obtained from ECL by setting appropriately the bias search parameters.

The search biases of ECL assume a uniform distribution of the data used for selection. This does not reflect reality in many learning problems. We are actually investigating alternative stochastic sampling mechanisms for selecting the portion of background knowledge, which would take into account the estimated importance of each element (e.g. fact of the background knowledge) according to some evaluation measure obtained from the examples in the training set.

## References

- Anglano, C., Giordana, A., Bello, G. L., & nza Saitta, L. (1998). An experimental evaluation of coevolutionary

- concept learning. *Proc. 15th International Conf. on Machine Learning* (pp. 19–27). Morgan Kaufmann, San Francisco, CA.
- Ashby, J., & Tennant, R. (1991). Definitive Relationships among chemical structure, carcinogenicity and mutagenicity for 301 chemicals tested by the U.S. NTP. *Mutation Research*, 257, 229–306.
- Augier, S., Venturini, G., & Kodratoff, Y. (1995). Learning first order logic rules with a genetic algorithm. *The First International Conference on Knowledge Discovery and Data Mining* (pp. 21–26). Montreal, Canada: AAAI Press.
- Blake, C., & Merz, C. (1998). UCI repository of machine learning databases.
- Debnath, A., de Compadre, R. L., Debnath, G., Schusterman, A., & Hansch, C. (1991). Structure-Activity Relationship of Mutagenic Aromatic and Heteroaromatic Nitro Compounds. Correlation with molecular orbital energies and hydrophobicity. *Journal of Medical Chemistry*, 34(2), 786–797.
- Eiben, A. E., Hinterding, R., & Michalewicz, Z. (1999). Parameter control in Evolutionary Algorithms. *IEEE-EC*, 3, 124.
- Giordana, A., & Neri, F. (1996). Search-intensive concept induction. *Evolutionary Computation*, 3, 375–416.
- Giordana, A., & Saitta, L. (2000). Phase Transitions in Relational Learning. *Machine Learning*, 41(2), 217–251.
- Glover, R., & Sharpe, P. (1999). Efficient GA based techniques classification. *Applied Intelligence*, 11(3), 277–289.
- Hekanaho, J. (1998). DOGMA: a GA based relational learner. *Proceedings of the 8th International Conference on Inductive Logic Programming* (pp. 205–214). Springer Verlag.
- Janikow, C. (1993). A knowledge intensive genetic algorithm for supervised learning. *Machine Learning*, 13, 198–228.
- Kennedy, C. J., & Giraud-Carrier, C. (1999). A depth controlling strategy for strongly typed evolutionary programming. *GECCO 1999: Proceedings of the First Annual Conference* (pp. 1–6). Morgan Kaufmann.
- Kubat, M., Bratko, I., & Michalski, R. (1998). A review of Machine Learning Methods. In R. Michalski, I. Bratko and M. Kubat (Eds.), *Machine learning and data mining*. Chichester: John Wiley and Sons Ltd.
- Leung, K., & Wong, M. (1995). Genetic logic programming and applications. *IEEE Expert*, 10(5), 68–76.
- Marchiori, E., & Steenbeek, A. (2000). An Evolutionary Algorithm for Large Scale Set Covering Problems with Application to Airline Crew Scheduling. *Real World Applications of Evolutionary Computing*. Springer (pp. 367–381). Springer-Verlag.
- Mitchell, T. (1982). Generalization as search. *Artificial Intelligence*, 18, 203–226.
- Mitchell, T. (1997). *Machine learning*. Series in Computer Science. McGraw-Hill.
- Muggleton, S. (1995). Inverse entailment and PROLOG. *New Generation Computing*, 13, 245–286.
- Muggleton, S. (1999). Inductive logic programming: issues, results and the challenge of learning language in logic. *Artificial Intelligence*, 114, 283–296.
- Muggleton, S., & Raedt, L. D. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19–20, 669–679.
- Neri, F., & Saitta, L. (1995). Analysis of genetic algorithms evolution under pure selection. *Proceedings of the Sixth International Conference on Genetic Algorithms* (pp. 32–39). Morgan Kaufmann, San Francisco, CA.
- Quinlan, J. (1990). Learning logical definition from relations. *Machine Learning*, 5, 239–266.
- Srinivasan, A., Muggleton, S., King, R., & Sternberg, M. (1994). Mutagenesis: Ilp experiments in a non-determinate biological domain. *Proceedings of the 4th International Workshop on Inductive Logic Programming* (pp. 217–232). Gesellschaft für Mathematik und Datenverarbeitung MBH.
- Teller, A., & Andre, D. (1997). Automatically choosing the number of fitness cases: The rational allocation of trials. *Genetic Programming 1997: Proceedings of the Second Annual Conference* (pp. 321–328). Stanford University, CA, USA: Morgan Kaufmann.