A Genetic Algorithms Approach to ILP

Alireza Tamaddoni-Nezhad and Stephen Muggleton

Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, UK
{atn,shm}@doc.ic.ac.uk

Abstract. In a previous paper we introduced a framework for combining Genetic Algorithms with ILP which included a novel representation for clauses and relevant operators. In this paper we complete the proposed framework by introducing a fast evaluation mechanism. In this evaluation mechanism individuals can be evaluated at genotype level (i.e. bit-strings) without mapping them into corresponding clauses. This is intended to replace the complex task of evaluating clauses (which usually needs repeated theorem proving) with simple bitwise operations. In this paper we also provide an experimental evaluation of the proposed framework. The results suggest that this framework could lead to significantly increased efficiency in problems involving complex target theories.

1 Introduction

Current ILP systems mostly employ deterministic search methods to examine the refinement space of clauses and use different kind of syntactic biases and heuristics (e.g. greedy methods) to cope with the complexity of the search, which otherwise is intractable. Using these biases usually limits the exploration power of the search and may lead to local optima. On the other hand, more powerful search methods are required for dealing with large search spaces (for example in multi-clause learning). Genetic Algorithms (GAs) have great potential for this purpose and it is likely that a combination of ILP and GAs can overcome the limitation of each individual method and can cope with some of the complexities of real-world applications.

In [26] a framework for combining Genetic Algorithms with ILP was introduced and a novel binary representation and relevant operators were discussed. It was shown that the proposed representation has interesting properties in terms of first-order concept learning. For example, it was shown that essential operations on clauses, such as unification and anti-unification [22, 21], can be achieved by simple bitwise operations (e.g. and/or) on binary strings.

In this paper we complete the proposed framework by introducing a fast evaluation mechanism for evaluating individuals at genotype level. In this paper we also explain an implementation of the proposed framework together with an empirical evaluation of the implemented system.

The paper is organised as follows. Section 2 reviews the proposed framework and also provides the definition and theorems which are needed in the next sections. Section 3 introduces a new evaluation mechanism for clauses. In this section we show that evaluating a clause can be done by simple and fast operations. Section 4 introduces stochastic refinement for directing genetic operators. Implementation and evaluation are explained in section 5. Related work and similar systems are discussed in section 6. Finally, section 7 concludes this paper and gives some directions for further research.

2 Representation, Encoding and Operators

In this section, we review the proposed binary encoding for first-order clauses and some of its properties. We also provide a summary of definitions and theorems which will be required in the rest of this paper. Proofs and more details about the representation and operators can be found in [26].

Figure 1.a shows the main idea of the binary representation by a simple example. Consider a clause with n variable occurrences. The relationships between these n variable occurrences can be represented by a graph having n vertices in which there exists an edge between vertices v_i and v_j if ith and jth variable occurrences in the clause represent the same variable. For example variable bindings in clause *p(X,Y):-q(X,Z),r(Z,Y)* represent an undirected graph and this clause can be represented by a binary matrix as shown in Figure 1.a. In this matrix entry m_{ij} is 1 if ith and jth variable occurrences in the clause represent the same variable and m_{ij} is 0 otherwise. This representation has interesting properties which can be exploited by a genetic algorithm for searching the refinement space of a clause.

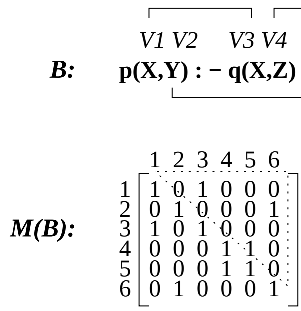
Definition 1 (Binding Set). *Let B and C both be clauses. C is in binding set $\mathcal{B}(B)$ if there exists a variable substitution* [1] *θ such that $C\theta = B$.*

Definition 2 (Binding Matrix). *Suppose B and C are both clauses and there exists a variable substitution θ such that $C\theta = B$. Let C have n variable occurrences representing variables $\langle v_1, v_2, \ldots, v_n \rangle$. The binding matrix of C is an $n \times n$ matrix M in which m_{ij} is 1 if there exist variables v_i, v_j and u such that v_i/u and v_j/u are in θ and m_{ij} is 0 otherwise. We write $M(v_i, v_j) = 1$ if $m_{ij} = 1$ and $M(v_i, v_j) = 0$ if $m_{ij} = 0$.*

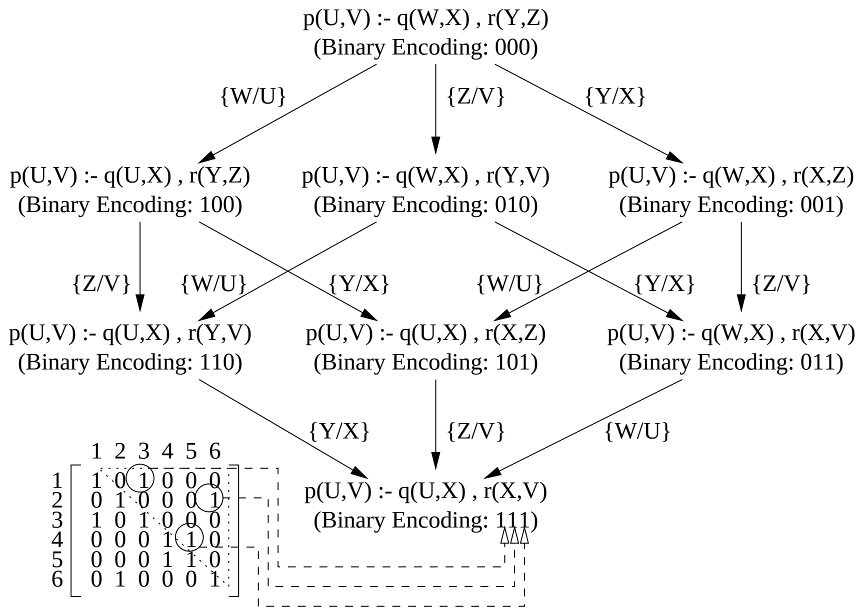
Definition 3 (Normalized Binding Matrix). *Let M be an $n \times n$ binary matrix. M is in the set of normalized binding matrices $\mathcal{M}_n$ if M is symmetric and for each $1 \leq i \leq n$, $1 \leq j \leq n$ and $1 \leq k \leq n$, $m_{ij} = 1$ if $m_{ik} = 1$ and $m_{kj} = 1$.*

Definition 4 (Mapping Function M(C)). *The mapping function $M : \mathcal{B}(B) \rightarrow \mathcal{M}_n$ is defined as follows. Given clause $C \in \mathcal{B}(B)$ with n variable occurrences representing variables $\langle v_1, v_2, \ldots, v_n \rangle$, $M(C)$ is an $n \times n$ binary matrix in which m_{ij} is 1 if variables v_i and v_j are identical and m_{ij} is 0 otherwise.*

[1] Substitution $\theta = \{v_i/u_j\}$ is a variable substitution if all v_i and u_j are variables.



(a) Binding matrix for clause: *p(X,Y):-q(X,Z),r(Z,Y)*



(b) A substitution lattice bounded below by clause *p(X,Y):-q(X,Z),r(Z,Y)* and binary encoding for each clause

Fig. 1. A binary representation for clauses and the relationship between the binary strings and the substitution ordering between clauses.

Definition 5 (Mapping Function C(M)). *The mapping function* $C : \mathcal{M}_n \rightarrow \mathcal{B}(B)$ *is defined as follows. Given a normalized* $n \times n$ *binding matrix* M, $C(M)$ *is a clause in* $\mathcal{B}(B)$ *with* n *variable occurrences* $\langle v_1, v_2, \ldots, v_n \rangle$, *in which variables* v_i *and* v_j *are identical if* m_{ij} *is 1.*

Definition 6 (Matrix Subset). *Let* P *and* Q *be in* $\mathcal{M}_n$. *It is said that* $P \subseteq Q$ *if for each entry* $p_{ij} \in P$ *and* $q_{ij} \in Q$, p_{ij} *is 1 if* q_{ij} *is 1.* $P = Q$ *if* $P \subseteq Q$ *and* $Q \subseteq P$. $P \subset Q$ *if* $P \subseteq Q$ *and* $P \neq Q$.

Definition 7 (Subsumption and Substitution). *Clause* C *subsumes clause* D, $C \succeq D$ *if there exists a (variable) substitution* θ *such that* $C\theta \subseteq D$ *(i.e. every literal in* $C\theta$ *is also in* D*).* C *properly subsumes* D, $C \succ D$ *if* $C \succeq D$ *and* $D \not\succeq C$. *Clause* D *is a substitution of clause* C, $C \sqsupseteq D$ *if there exists a (variable) substitution* θ' *such that* $C\theta' = D$ *(i.e. every literal in* $C\theta'$ *corresponds to a literal in* D*).* D *is a proper instance of* C, $C \sqsupset D$ *if* $C \sqsupseteq D$ *and* $D \not\sqsupseteq C$.

Because of the substitution order between clauses (which is a quasi-order) the search space (or refinement space) can be modelled as a sub-lattice of subsumption [21]. The following theorem represents the relationship between binary matrices and the substitution order of clauses. However, because these theorems also hold for the subsumption order we use the subsumption notation which is more general.

Theorem 1. *For each clause* B *and matrices* M_1 *and* M_2 *in* $\mathcal{M}_n$ *such that* $C(M_1) \in \mathcal{B}(B)$ *and* $C(M_2) \in \mathcal{B}(B)$, $C(M_1) \succ C(M_2)$ *if* $M_1 \subset M_2$.

Example 1. Figure 1.b shows a substitution lattice bounded below by the clause p(X,Y):-q(X,Z),r(Z,Y). This lattice shows the substitution ordering between clauses which are in the same binding set. According to Theorem 1, binding matrix of each clause which subsumes the bottom clause must be a subset of the binding matrix of the bottom clause. Hence, each clause in this search space can be encoded by 3 bits.

The proposed binary representation can be used to encode a substitution lattice (bounded below by a bottom-clause) in a compact way. Moreover, this encoding reduces the redundancy which we usually have in a first-order representation. However, we may still have redundant binary representation for clauses with more than one literal per predicate. To avoid this redundancy, one solution is to re-order literals with the same predicate symbol in such a way that the corresponding binary number is minimal. Using this strategy ensures that the binary encoding for these clauses is not redundant. We assume that this condition is hold in the following definitions and theorems. Another property of the proposed representation is that mgi(most general instance) and lgg(least general generalization) which are also known as unification and anti-unification operations on clauses [22, 21] can be achieved by simple bitwise operations on the binary encoding of clauses. In the following, first we introduce *mgi* and *lgg* operations for clauses.

Definition 8 (mgi and lgg). *Clauses E and F are respectively a common instance and a common generalization of clauses C and D if and only if $C, D \succeq E$ and $F \succeq C, D$. $mgi(C,D)$ and $lgg(C,D)$ are the most general instance and the least general generalization for clauses C and D if and only if for every common instance E and common generalization F it is the case that $mgi(C,D) \succeq E$ and $F \succeq lgg(C,D)$.*

Example 2. In Figure 1.b clause p(U,V):-q(U,X),r(X,Z) is the mgi of clauses p(U,V):-q(U,X),r(Y,Z) and p(U,V):-q(W,X),r(X,Z), and p(U,V):-q(W,X),r(Y,V) is the lgg of clauses p(U,V):-q(U,X),r(Y,V) and p(U,V):-q(W,X),r(X,V).

Definition 9 (Matrix AND). *Let M_1 and M_2 be in $\mathcal{M}_n$. $M = (M_1 \wedge M_2)$ is an $n \times n$ matrix and for each $a_{ij} \in M$, $b_{ij} \in M_1$ and $c_{ij} \in M_2$, $a_{ij} = 1$ if $b_{ij} = 1$ and $c_{ij} = 1$ and $a_{ij} = 0$ otherwise.*

Similar to AND operator, OR operator $(M_1 \vee M_2)$ is constructed by bitwise OR-ing of M_1 and M_2 entries.

Definition 10 (Matrix OR). *Let M_1 and M_2 be in $\mathcal{M}_n$. $M = (M_1 \vee M_2)$ is an $n \times n$ matrix and for each $a_{ij} \in M$, $b_{ij} \in M_1$ and $c_{ij} \in M_2$, $a_{ij} = 1$ if $b_{ij} = 1$ or $c_{ij} = 1$ and $a_{ij} = 0$ otherwise.*

Theorem 2. *For each clause B and matrices M_1, M_2 and M in $\mathcal{M}_n$ such that $C(M_1) \in \mathcal{B}(B)$, $C(M_2) \in \mathcal{B}(B)$ and $C(M) \in \mathcal{B}(B)$, $C(M) = lgg(C(M_1), C(M_2))$ if $M = (M_1 \wedge M_2)$.*

Theorem 3. *For each clause B and matrices M_1, M_2 and M in $\mathcal{M}_n$ such that $C(M_1) \in \mathcal{B}(B)$, $C(M_2) \in \mathcal{B}(B)$ and $C(M) \in \mathcal{B}(B)$, $C(M) = mgi(C(M_1), C(M_2))$ if $M = M_1 \vee M_2$.*

Example 3. In Figure 1.b, lgg and mgi of any two clauses can be obtained by AND-ing and OR-ing of their binary strings.

In summary, the proposed binary representation can be used to encode a substitution lattice bounded below by a bottom clause. Essential operations on clauses can be achieved by bit-wise operation on binary strings. These operations can be considered as task-specific genetic operators which together with conventional genetic operators (i.e. mutation and crossover) can be used to search the refinement space of clauses. This issue will be discussed in section 4. In the next section we show how the properties of the proposed representation can be used to evaluate individuals (i.e. binary strings) at genotype level without mapping them into corresponding clauses.

3 Evaluation Mechanism

The usual way for evaluating a hypothesis in first-order concept learning systems is to repeatedly call a theorem prover (e.g. Prolog interpreter) on training examples to find out positive and negative coverage of the hypothesis. This step is known to be a complex and time-consuming task in first-order concept learning. In the case of genetic-based systems this situation is even worse, because we need to evaluate a population of hypotheses in each generation. This problem is another important difficulty when applying GAs in first-order concept learning.

In this section we introduce a method which is intended to replace the complex task of evaluating clauses with bitwise operations on binary strings. This idea is similar to data-compilation method used by the attribute-based learning system GIL [9]. This system retains binary coverage vectors for all possible features (attributes and values) which can appear in a rule. This introduces a database which can be used for computing the coverage set of each rule by bitwise operations on the coverage vectors of participating features. However, in our case there is no need to maintain such a database. We show that by maintaining the coverage sets for a small number of clauses and by doing bitwise operations we can compute the coverage for other binary strings without mapping them into the corresponding clauses. This property is based on the implicit subsumption order which exists in the binary representation. In the following, first we define the cover-vector approach for representing coverage of a clause on training examples.

Definition 11 (Cover Sets and Cover Vectors). *Let C be a clause and $E^+ = \{e_1^+, e_2^+, \ldots, e_k^+\}$ and $E^- = \{e_1^-, e_2^-, \ldots, e_l^-\}$ be the set of positive and negative training examples respectively* [2]. *e_i^+ is in the positive cover set $\mathcal{P}(C)$ if $C \succeq e_i^+$. Similarly, e_j^- is in the negative cover set $\mathcal{N}(C)$ if $C \succeq e_j^-$. The positive cover vector $\mathcal{PV}(C)$ is a k-bit binary string in which bit i is 1 if $e_i^+ \in \mathcal{P}(C)$ and 0 otherwise. Similarly, the negative cover vector $\mathcal{NV}(C)$ is a l-bit binary string in which bit j is 1 if $e_j^- \in \mathcal{N}(C)$ and 0 otherwise.*

Theorem 4. *For each clause C_1 and C_2, $\mathcal{P}(mgi(C_1, C_2)) = \mathcal{P}(C_1) \cap \mathcal{P}(C_2)$.*

Proof. Let $e \in \mathcal{P}(mgi(C_1, C_2))$, then according to Definition 11, $mgi(C_1, C_2) \succeq e$. But according to the definition of mgi, $C_1 \succeq mgi(C_1, C_2)$ and $C_2 \succeq mgi(C_1, C_2)$ and therefore $C_1 \succeq e$ and $C_2 \succeq e$. Hence, $e \in \mathcal{P}(C_1)$ and $e \in \mathcal{P}(C_2)$ and therefore $e \in \mathcal{P}(C_1) \cap \mathcal{P}(C_2)$. Hence, $\mathcal{P}(mgi(C_1, C_2)) \subset \mathcal{P}(C_1) \cap \mathcal{P}(C_2)$. Now, let $e \in \mathcal{P}(C_1) \cap \mathcal{P}(C_2)$, then according to Definition 11, $C_1 \succeq e$ and $C_2 \succeq e$. But according to the definition of mgi, $mgi(C_1, C_2) \succeq e$. Hence, $e \in \mathcal{P}(mgi(C_1, C_2))$ and therefore $\mathcal{P}(C_1) \cap \mathcal{P}(C_2) \subset \mathcal{P}(mgi(C_1, C_2))$ and this completes the proof. □

Theorem 5. *For each M, M_1 and M_2 in $\mathcal{M}_n$, $\mathcal{PV}(C(M)) = \mathcal{PV}(C(M_1)) \wedge \mathcal{PV}(C(M_2))$ if $M = M_1 \vee M_2$.*

Proof. Suppose $M = M_1 \vee M_2$. Therefore according to Theorem 3, $C(M) = mgi(C(M_1), C(M_2))$. Then according to Theorem 4 ,$\mathcal{P}(C(M)) = \mathcal{P}(C(M_1)) \cap \mathcal{P}(C(M_2))$. But according to Definition 11, $\mathcal{PV}(C(M)) = \mathcal{PV}(C(M_1)) \wedge \mathcal{PV}(C(M_2))$.
□

[2] Note that the training examples must be prepared for substitution testing with respect to the bottom-clause.

This theorem, which also holds for negative coverage vectors, can be easily extended for n clauses. According to these theorems, positive (or negative) coverage of clauses can be computed by bitwise operations. Hence, the evaluation of each individual is done at genotype level without mapping it into the corresponding phenotype (clause).

Example 4. In Figure 1.b we can compute the coverage vector of any clause provided the coverage vector of individuals 001, 010 and 100. For example: $\mathcal{PV}(C(110)) = \mathcal{PV}(C(100)) \wedge \mathcal{PV}(C(010))$ and $\mathcal{PV}(C(111)) = \mathcal{PV}(C(100)) \wedge \mathcal{PV}(C(010)) \wedge \mathcal{PV}(C(001))$.

4 Stochastic Refinement

According to the theorems in section 2, unification and anti-unification can be done by simple bitwise operations on the binary encoding of clauses. These properties can be used for designing task-specific genetic operators such as generalization and specialization crossover operators. Generalization and specialization are known as the main operations in concept learning methods [28, 16, 15]. In particular, *lgg* and *mgi* are essential in first-order learning. For example, the ILP system Golem [18] which was successfully applied to a wide range real-world applications [2, 4, 19] only uses a *lgg* operator which operates on determinacy restricted clauses.

In addition to the generalization and specialization crossovers mentioned earlier, we can also introduce task-specific mutation operators. In the standard mutation operator we use a fixed probability (mutation-rate) for changing 0 and 1 bits [3]. As shown in section 2, the difference between bits in binding matrices determines the substitution order between clauses. Hence, the substitution distance between clauses increases monotonically with the Hamming distance between the corresponding matrices. We can use this property to set different mutation rates for 0 and 1 bits based on a desirable degree of generalization and specialization. This could lead to a directed mutation operator.

This degree of generalization or specialization (which can be also used for crossover operators) is introduced by probabilities for generalizations and specialization. In a genetic search some criteria such as completeness and consistency of current individuals can be used for setting these probabilities [9, 6].

In first-order concept learning, upward and downward refinement operators are used for generalization and specialization of clauses [21]. In our case, task-specific genetic operators can be interpreted as *stochastic refinement operators* in the context of first-order concept learning.

5 Implementation and Empirical Evaluation

As shown in the previous sections, the proposed framework has great potential for combining GAs and ILP and could lead to an increased performance in a

[3] A random mutation could result in a matrix which is not consistent with Definition 3. This matrix could be normalized using Definition 3.

INPUT: B=Background knowledge, E=Set of examples
1. If $E = \emptyset$ return B
2. Let e be the first example in E
3. Construct the bottom clause ($\perp$) for e
4. $\boxed{\text{Find the best clause H such that } \Box \succeq H \succeq \perp}$
5. Let $B = B \cup H$
6. Let $E' = \{e : e \in E, B \models e\}$
7. Let $E = E - E'$
8. Goto 1

Fig. 2. Progol's Covering Algorithm. In the present implementation, the A^*-like search used in step 4 is replaced by a genetic search.

system which is created based on this framework. In particular, in this section we examine the following two null hypotheses:

Null hypothesis 1: A GA-ILP system based on the proposed framework does not lead to significantly increased efficiency in any problem involving complex target theories.

Null hypothesis 2: The increased efficiency in Null hypothesis 1 cannot be achieved without substantial decrease of accuracy.

To test these hypotheses, we employed the proposed framework to combine Inverse Entailment in CProgol4.4 with a genetic algorithm. CProgol is an ILP system which develops first-order hypotheses from examples and background knowledge. Figure 2 shows the set covering algorithm used in CProgol. CProgol uses Inverse Entailment [17] to construct the most specific clause (or the bottom-clause) for each example and then searches for the best clause H which subsumes this bottom-clause ($\Box \succeq H \succeq \perp$). This introduces a subsumption lattice bounded below by the bottom-clause ($\perp$). The standard CProgol starts from the empty clause ($\Box$) and uses an A^*-like algorithm for searching this bounded subsumption lattice (step 4). In the present implementation, this lattice is searched by a genetic search. The details for CProgol's A^*-like search, refinement operator and algorithm for building the bottom-clause can be found in [17].

As shown in section 2, $\mathcal{B}(\perp)$ represents a substitution lattice bounded below by the bottom-clause. We used a genetic algorithm together with the binary encoding for clauses (as described in section 2) to evolve a randomly generated population of binary strings in which each individual corresponds to a member of $\mathcal{B}(\perp)$. Because of simple representation and straightforward operators any standard genetic algorithm can be used for this purpose. We used a *Simple Genetic Algorithm(SGA)* [7] and modified it to suit the representation introduced in this paper. This genetic search evolves a population of hypotheses which all subsume the bottom-clause.

In the following, we explain the material and methods used in our experimentation and then discuss the results.

for $i = 1$ **to** 100 **do**
  for $j = 1$ **to** 5 **do**
    Generate a random target concept T_{ij} with complexity $5 \times j$
    Generate 2×100 random training and test examples for concept T_{ij}
    Execute Progol on the training examples using the A^* search
      N_{ij} =number of evaluations before finding hypothesis C_{ij}
      A_{ij} =predictive accuracy of C_{ij} on the test examples
    Execute Progol on the training examples using the genetic search
      N'_{ij} =number of evaluations before finding hypothesis C'_{ij}
      A'_{ij} =predictive accuracy of C'_{ij} on the test examples
  end
end
for $j = 1$ **to** 5 **do**
  Plot average and standard error of N_{ij} and N'_{ij} versus $5 \times j$ ($i \in [1..100]$)
for $j = 1$ **to** 5 **do**
  Plot average and standard error of A_{ij} and A'_{ij} versus $5 \times j$ ($i \in [1..100]$)

Fig. 3. Experimental method.

Material and methods In this experiment, we compare the performance of the A^*-like search and the genetic search in learning concepts with different complexities. Figure 3 shows the experimental method used in this experiment. In this experiment we measure the average performance of the genetic search and the A^* search on 100 different runs. In each run, target concepts with complexities between 5 to 25 are generated. For each target concept, a fixed number (i.e. 100) of examples are generated both for training and testing. After generating random examples, Progol is executed on the training example using the A^* search and the genetic search. For each iteration of the loop the following parameters are recorded: N, the number of evaluations before finding a single clause C which is complete and consistent with respect to the training examples and A, the predictive accuracy of C on the test examples. The average and standard error of these parameters are then plotted against the complexity of the target concepts.

As mentioned before, in this experiment we needed to generate random concepts with a given complexity as well as random training and test example for each concept for measuring the predictive accuracy of the induced hypotheses. For this purpose, we have used a concept generator program in which the concept description language is determined by a Stochastic Logic Program(SLP) [20]. In the present experiment, we have used a concept description language similar to one used in the Michalski's trains problem [14]. Table 1 shows part of the train description language used in this experiment. This program defines the space of all possible carriages. A random train is defined as a list of carriages sampled from this program. In this experiment, the complexity of target concept is measured by the number of specific features which describe the target concept. This

Table 1. Part of a stochastic logic program for generating random trains. This program defines the space of all possible carriages. A random train is defined as a list of carriages sampled from this program.

```
carriage(Shape,Length,Double,Roof,Wheels,Load) :-
        shape(Length,Shape),
        double(Length,Shape,Double),
        roof(Length,Shape,Roof),
        wheels(Length,Wheels),
        load(Length,Load).

shape(long,rectangle).  shape(short,rectangle).  shape(short,ellipse).
shape(short,hexagon).   shape(short,u_shaped).   shape(short,bucket).

double(short,rectangle,double).      double(long,rectangle,not_double).
double(short,rectangle,not_double).  double(short,ellipse,not_double).
double(short,hexagon,not_double).    double(short,u_shaped,not_double).
double(short,bucket,not_double).

roof(short,ellipse,arc).     roof(short,hexagon,flat).
roof(long,rectangle,none).   roof(long,rectangle,flat).
roof(long,rectangle,jagged). roof(short,rectangle,none).
roof(short,rectangle,flat).  roof(short,rectangle,peaked).
roof(short,u_shaped,none).   roof(short,u_shaped,flat).
roof(short,u_shaped,peaked). roof(short,bucket,none).
roof(short,bucket,flat).     roof(short,bucket,peaked).

wheels(short,2).  wheels(long,2).  wheels(long,3).

load(short,l(circle,1)).    load(short,l(diamond,1)).
load(short,l(hexagon,1)).   load(short,l(rectangle,1)).
load(short,l(triangle,1)).  load(short,l(utriangle,1)).
load(short,l(circle,2)).    load(short,l(diamond,2)).
load(short,l(hexagon,2)).   load(short,l(rectangle,2)).
load(short,l(triangle,2)).  load(short,l(utriangle,2)).
load(long,l(circle,1)).     load(long,l(diamond,1)).
load(long,l(hexagon,1)).    load(long,l(rectangle,1)).
load(long,l(triangle,1)).   load(long,l(utriangle,1)).
load(long,l(circle,2)).     load(long,l(diamond,2)).
load(long,l(hexagon,2)).    load(long,l(rectangle,2)).
load(long,l(triangle,2)).   load(long,l(utriangle,2)).
load(long,l(circle,3)).     load(long,l(diamond,3)).
load(long,l(hexagon,3)).    load(long,l(rectangle,3)).
load(long,l(triangle,3)).   load(long,l(utriangle,3)).
```

Table 2. Control parameters and evaluation function used by the genetic search in the present experimentation.

Population size ($popsize$): 30
Probability of mutation (p_m): 0.0333
Probability of crossover (p_c): $1 - 0.8 \times f$
Probability of lgg (p_{lgg}): $0.8 \times f$
($f = \frac{f(C_1)+f(C_2)}{2}$ where C1 and C2 are parental clauses in crossover and lgg)

Evaluation function: $f(C) = \alpha \frac{e^+(C)}{(E^+ + \beta * e^-(C))} + (1-\alpha)(1 - \frac{c(C)+h(C)}{c_{max}+h_{max}})$
where $\alpha = 0.8$ and $\beta = 0.5$
E^+, E^-: total number of positive and negative examples
$e^+(C), e^-(C)$: number of positive and negative examples covered by clause C
$c(C)$: length of clause C
$h(C)$: number of further literals to complete clause C (as defined in [17])

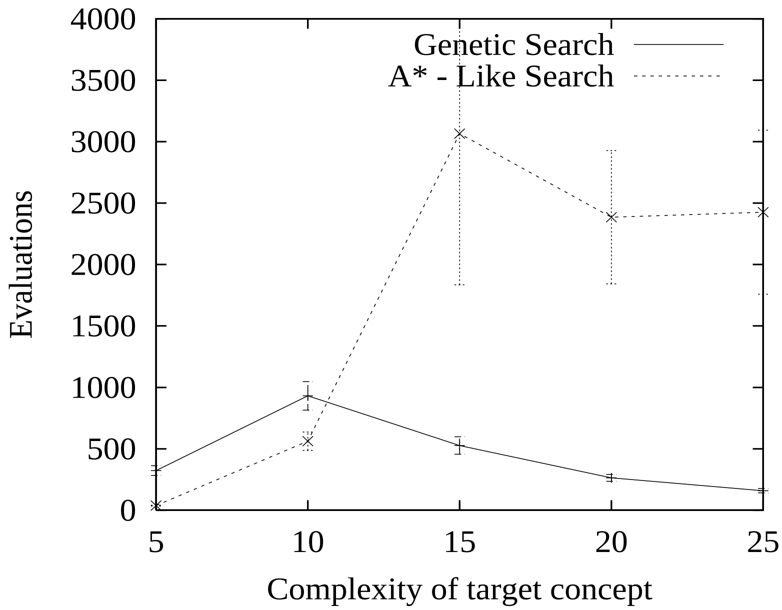
is determined by the number of specific carriages in the target train and number of properties for each carriage (see Table 1).

The evaluation function and control parameters which are used by the genetic search are shown in Table 2. The evaluation function uses similar criteria to the ones used in the evaluation function of the A^*-like search of CProgol. The probability setting for generalisation used in this experiment is similar to one used in [6].

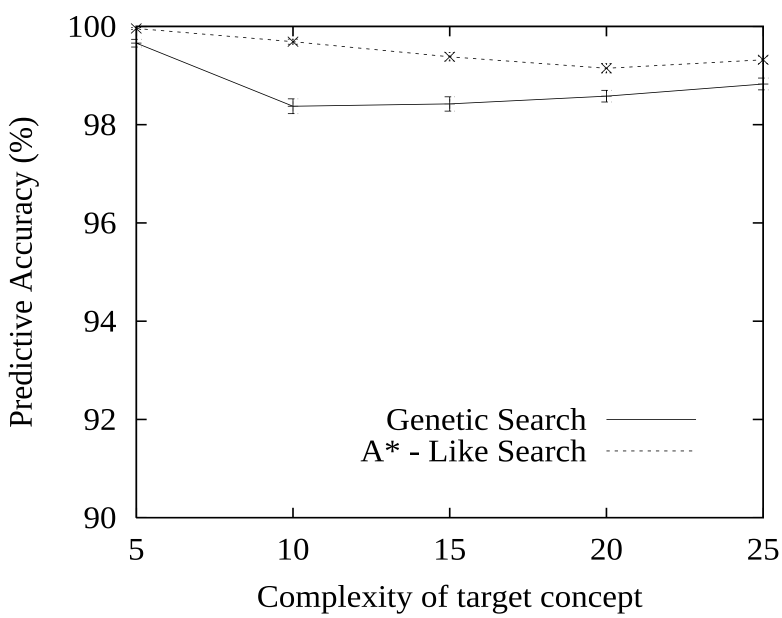
Results and discussion Figure 4.a compares the average number of clauses evaluated by the genetic search and the A^*-like search in learning concepts with different complexities. The vertical axis shows the average number of the explored nodes before finding a complete and consistent hypothesis. The horizontal axis shows the complexity of target concept which is measured by the number of conditions in the target concept. This figure shows that in this experiment the A^*-like search exhibits a better performance in learning concepts with small complexities (e.g. less than 10). However, the number of evaluations by the A^*-like search grows very sharply in the complexity of target concept and varies between 38 and 3065.

Figure 4.b compares the predictive accuracy of the induced hypotheses by the genetic search and the A^*-like search. This figure shows that the predictive accuracy of the genetic search is slightly lower (less than 2%) than the predictive accuracy of the A^*-like search.

In summary, these graphs suggest that in the random trains problem the genetic search can lead to significantly increased efficiency for learning complex target trains and this can be achieved without substantial decrease of accuracy. These graphs also suggest that the performance of the genetic search is less dependent on the complexity of hypotheses, whereas A^*-like search shows a great dependency on this factor. The results of this experiment are consistent with the



(a)



(b)

Fig. 4. Performance of the genetic search and A^*-like search in learning concepts with different complexities. a) average number of clauses evaluated by each search method and b) predictive accuracy of the induced hypotheses versus the complexity of concepts.

fact that the actual completeness and complexity exhibited by the standard A^*-like search of CProgol depends upon the order of literals in the bottom clause and upon the complexity of the hypothesis. In contrast, the genetic search is less dependent on the complexity of the hypotheses and is not affected by the order of literals in the bottom clause.

6 Related Work

One main difficulty in applying conventional GAs in first-order domain is related to the formulation of first-order hypotheses into bit-strings. GA-SMART [6] was the first relation learning system which tackled this problem by restricting concept description language and introducing a language template. A template in GA-SMART is a fixed length CNF formula which must be defined by the user. Mapping a formula into bit-string is done by setting the corresponding bits to represent the occurrences of predicates in the formula. The main problem of this method is that the number of conjuncts in the template grows combinatorially with the number of predicates. REGAL [5], DOGMA [8] and G-NET [1] follow the same basic idea as GA-SMART and employ a user-defined template for mapping first-order rules into bit strings. However, instead of using a standard representation, a template in these systems is a conjunction of internally disjunctive predicates. This leads to some difficulties, for example in representing continuous attributes. Other systems including GILP [27], GLPS [13], LOGENPRO [29], STEPS [11] and EVIL [23] use hierarchical representations rather than fixed length bit-strings. These systems evolve a population of logic programs in a Genetic Programming (GP) [12] manner. Even though some of the above mentioned systems use background knowledge for generating the initial population or seeding the population, most of these systems cannot benefit from intentional background knowledge in the same way as in usual first-order learning systems. This is mainly because in most cases genetic search has been used as the only learning mechanism in the system.

In our proposed framework, encoding of hypotheses is based on a most specific (or bottom) clause which is constructed according to the background knowledge and training examples. This bottom-clause can be automatically constructed using logic-based methods such as Inverse Entailment. Moreover, as shown in section 2 and section 3, the proposed encoding and operators can be interpreted in well known first-order logic terms.

7 Conclusions and Further Work

In this paper we have introduced a framework for combining first-order concept learning with GAs by introducing a novel encoding for clauses, relevant genetic operators and a fast evaluation mechanism. Empirical results suggest that the proposed framework could lead to significantly increased efficiency in problems involving complex target theories and this can be achieved without substantial decrease of accuracy.

The present implementation could be improved in many ways. A natural improvement might be using more sophisticated genetic algorithms rather than a simple genetic algorithm. For example the greedy cover set algorithm of CProgol, which repeatedly generalizes examples, could be replaced by a parallel genetic algorithm. The task-specific genetic operators can be used to guide the genetic search towards the interesting areas of the search space by specialization and/or generalization as it is done in usual concept learning systems. The fast evaluation mechanism can be used to compensate for the natural computation cost of a genetic algorithm and could lead to a high performance genetic search. In the current approach, the occurrence of atoms in a clause is not considered in the binary encoding of the clause and inactive atoms (e.g. unconnected predicates) are filtered from the induced hypotheses. Alternatively, the presence or absence of atoms in each clause can be encoded as a part of the binary representation of the clause. Finally, more experiments are required to evaluate the proposed framework in real-world domains.

The framework proposed in this paper is an opportunity not only to utilize the benefits of these two different paradigms (i.e. ILP and GAs) in a hybrid system but also to study some common issues with an analogical view. In the following we review some of these issues which could be considered as further research.

Hybrid search. It has been argued [7] that GAs are a weak method without the guarantee of optimality. In other words, GAs sort out interesting area of a space quickly without the guarantees of more convergent process. In ILP problems, which known local but convergent methods exist, the idea of a hybrid GA [3] is natural. In this scheme the search is started using a GA to sort out the interesting hills in the problem. Once the GA locates the best regions, a locally convergent search is used to climb the local peaks. In this way, one can combine the globality and parallelism of the GA with the more convergent behaviour of the local search (i.e. the ILP techniques). In this hybrid scheme, the GA performs a global adaptive search of the space of possible hypotheses and then an ILP algorithm locally refines an initial estimate provided by the GA. Theory revision in ILP could be relevant in designing a local search in this scheme. This hybrid scheme also provides an opportunity to study the interaction between the computational models of 'evolution' and 'learning'.

Parallel search. In addition to the well-known implicit parallelism [4], GAs are naturally suitable for parallel implementation [25]. This makes it easier to scale-up GA-based systems and to benefit from the computational power of parallel and distributed hardware. Hence, in a GA-based ILP system a reasonable attempt is to parallelize the search. Moreover, there are some situations in ILP methods where parallel processing could be useful. For example, the greedy

[4] It has been shown [7] that even though in each generation we perform computation proportional to the size of the population (n) we get useful processing of much more schemata ($O(n^3)$) in parallel with no special bookkeeping or memory other than the population itself.

set covering algorithm which repeatedly generalizes training examples, could be replaced by a parallel co-evolutionary procedure [1].

Learning clausal theories. In the current approach each individual in the population stands for a single clause. The final solution consists of clauses each corresponding to an iteration of the cover set algorithm. An alternative is to induce the whole clausal theory at once. In a GA-based system this requires encoding of clausal theories as bit-strings which are regarded as individuals in the population. This problem is similar to Pittsburgh approach [24, 10] in which each individual encodes a set of production rules. To be able to learn clausal theories, the proposed binary encoding must be extended to represent multiple clauses.

Acknowledgements

We would like to thank the anonymous reviewers for their comments and constructive criticism. The first author was supported by an ILPnet2 travel funding to attend ILP02 conference.

References

1. C. Anglano, A. Giordana, G. Lo Bello, and L. Saitta. An experimental evaluation of coevolutive concept learning. pages 19–27. Morgan Kaufmann, 1998.
2. I. Bratko, S. Muggleton, and A. Varsek. Learning qualitative models of dynamic systems. In *Proceedings of the Eighth International Machine Learning Workshop*, San Mateo, Ca, 1991. Morgan-Kaufmann.
3. L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.
4. C. Feng. Inducing temporal fault dignostic rules from a qualitative model. In S. Muggleton, editor, *Inductive Logic Programming.* Academic Press, London, 1992.
5. A. Giordana and F. Neri. Search-intensive concept induction. *Evolutionary Computation Journal*, 3(4):375–416, 1996.
6. A. Giordana and C. Sale. Learning structured concepts using genetic algorithms. pages 169–178. Morgan Kaufmann, 1992.
7. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, Reading, MA, 1989.
8. J. Hekanaho. Dogma: A ga-based relational learner. pages 205–214. Springer-Verlag, 1998.
9. C. Z. Janikow. A knowledge-intensive genetic algorithm for supervised learning. *Machine Learning*, 13:189–228, 1993.
10. Kenneth A. De Jong, William M. Spears, and Diana F. Gordon. Using genetic algorithms for concept learning. *Machine Learning*, 13:161–188, 1993.
11. C. J. Kennedy and C. Giraud-Carrier. An evolutionary approach to concept learning with structured data. In *Proceedings of the fourth International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 1–6. Springer Verlag, April 1999.
12. J. R. Koza. *Genetic Programming.* MIT Press, Cambridge, MA, 1991.

13. K. S. Leung and M. L. Wong. Genetic logic programming and applications. *IEEE Expert*, 10(5):68–76, 1995.
14. R.S. Michalski. Pattern recognition as rule-guided inductive inference. In *Proceedings of IEEE Trans. on Pattern Analysis and Machine Intelligence*, pages 349–361, 1980.
15. T. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.
16. T.M. Mitchell. Generalisation as search. *Artificial Intelligence*, 18:203–226, 1982.
17. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
18. S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, 1990. Ohmsha.
19. S. Muggleton, R. King, and M. Sternberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5(7):647–657, 1992.
20. S.H. Muggleton. Stochastic logic programs. In L. de Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.
21. S-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag, Berlin, 1997. LNAI 1228.
22. G.D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, Edinburgh, 1969.
23. P. G. K. Reiser and P. J. Riddle. Evolving logic programs to classify chess-endgame positions. In C. Newton, editor, *Second Asia-Pacific Conference on Simulated Evolution and Learning*, Canberra, Australia, 1998.
24. S F Smith. Flexible learning of problem solving heuristics through adaptive search. In *Proc. 8th Int. Joint Conf. on A.I.*, pages 422–425, 1983.
25. J. Stender. *Parallel Genetic Algorithms: Theory and Practice*. IOS Press, Amsterdam, 1993.
26. A. Tamaddoni-Nezhad and S. H. Muggleton. Searching the subsumption lattice by a genetic algorithm. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, pages 243–252. Springer-Verlag, 2000.
27. A. Varšek. *Inductive Logic Programming with Genetic Algorithms*. PhD thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Ljubljana, Slovenia, 1993.
28. P. H. Winston. *Learning Structural Descriptions from Examples*. Phd thesis, MIT, Cambridge, Massachusetts, January 1970.
29. M. L. Wong and K. S. Leung. Evolutionary program induction directed by logic grammars. *Evolutionary Computation*, 5(2):143–180, 1997.