IEEE Expert, **10**, no. 5. pp. 68-76

Inducing Logic Programs With Genetic Algorithms: The Genetic Logic Programming System

Man Leung Wong
Department of Computing and Decision Sciences
Lingnan University
Tuen Mun
Hong Kong

mlwong@ln.edu.hk

Kwong Sak Leung
Department of Computer Science
The Chinese University of Hong Kong
Hong Kong

ksleung@cs.cuhk.edu.hk

Abstract

Inductive Logic Programming (**ILP**) integrates the techniques from traditional machine learning and logic programming to construct logic programs from training examples. Most existing systems employ greedy search strategies which may trap the systems in a local maxima. This paper describes a system, called the Genetic Logic Programming System (**GLPS**), that uses Genetic Algorithms (GA) to search for the best program. This novel framework combines the learning power of GA and knowledge representation power of logic programming to overcome the shortcomings of existing paradigms.

A new method is used to represent a logic program as a number of tree structures. This representation facilitates the generation of initial logic programs and other genetic operators. Four applications are used to demonstrate the ability of this approach in inducing various logic programs including the recursive factorial program. Recursive programs are difficult to learn in Genetic Programming (**GP**). This experiment shows the advantage of Genetic Logic Programming (**GLP**) over **GP.**

Only a few existing learning systems can handle noisy training examples, by avoiding overfitting the training examples. However, some important patterns will be ignored. The performance of **GLPS** on learning from noisy examples is evaluated on the chess endgame domain. A systematic method is used to introduce different amounts of noise into the training examples. A detailed comparison with **FOIL** has been performed and the performance of **GLPS** is significantly better than that of **FOIL** by at least 5% at the 99.995% confidence interval at all noise levels. The largest difference even reaches 24%. This encouraging result demonstrates the advantages of our approach over existing ones.

1. Introduction

Currently, there have been increasing interests in systems that induce first order logic programs. Inductive Logic Programming (**ILP**) is the combinations of techniques and interests in inductive concept learning and logic programming. **ILP** is more powerful than traditional inductive learning methods because it uses an expressive first-order logic framework and facilitates the application of background knowledge. Furthermore, **ILP** has a strong theoretical foundation from logic programming and computational learning theory. It also has very impressive applications in knowledge discovery in databases (Lavrac and Dzeroski 1994) and logic program synthesis. However, **ILP** is limited in performing concept learning. Other learning paradigms such as reinforcement learning and strategy learning cannot be achieved by **ILP**.

FOIL (Quinlan 1990) efficiently learns function free Horn clauses, a useful subset of first order predicate logic. It uses a top-down, divide and conquer approach guided by information-based heuristics to produce a concept description that covers all positive examples and excludes all negative examples. **FOCL** (Pazzani and Kibler 1992) extends **FOIL** by integrating inductive and analytic learning in a uniform framework and by allowing different forms of background knowledge to be used in generating function free Horn clauses. **GOLEM** (Muggleton and Feng 1990) learn logic programs by employing inverse resolution. Despite the efficiency of **FOIL**, **FOCL** and **GOLEM**, they are highly dependent on the given vocabulary and the forms of training examples. They cannot extend their vocabulary. In logic programming, inventing new predicates can be treated as creating useful subroutines.

The task of inducing logic programs can be formulated as a search problem (Mitchell 1982) in a concept space of logic programs. Various approaches differ mainly in the search strategies and heuristics used to guide the search. Since the search space is extremely large in logic programming, strong heuristics are required to manage the problem. Most systems are based on a greedy search strategy. The systems generate a sequence of logic programs from general to specific ones (or from specific to general) until a consistent target program is found. Each program in the sequence is obtained by specializing or generalizing the previous one. For example, **FOIL** applies the hill climbing search strategy guided by an information-gain heuristics to search programs from general to specific ones. However, these strategies and heuristics are not always applicable because they may trap the systems in local maxima. In order to overcome this problem, non greedy strategies should be adopted.

An alternate search strategy is Genetic Algorithm (**GA**) which performs parallel searches. Genetic Algorithm performs both exploitation of the most promising solutions and exploration of the search space. It is featured to tackle hard search problems and thus it may be applicable to logic program induction. Since its invention (Holland 1975), GA has proven successful in finding an optimal point in a search space for a wide variety of problems (Goldberg 1989).

A Genetic Programming paradigm (**GP**) extends traditional **GA** to learn computer programs represented as S-expressions of LISP (Koza 1992; 1994). **GP** is a very general and domain-independent search method. It has impressive applications in symbolic regression, learning of control and game playing strategies, evolution of emergent behavior, evolution of subsumption, automatic programming, concept learning, induction of subroutines and hierarchy of a program, and meta-level learning (Koza 1992; 1994, Kinnear 1994, Wong and Leung 1994a; 1994b; 1995). Although it

is very general, it has little theoretical foundation. The shortcomings of **GP** are summarized as follows:

- The semantics of the program created are unclear because (a) the semantics of some primitive functions such as LEFT, RIGHT and MOVE (Koza 1992) are difficult to define. (b) various execution models can be used to execute the programs generated. Thus the semantics of the programs depends on the underlying execution model. It is possible to create two identical programs with different semantics because the underlying execution models are different.
- The underlying execution model must be defined before programs can be created. It means that users must have some ideas of the solutions.
- It is difficult if not impossible to generate recursive programs
- The sub-functions inventing mechanism is restrictive (Koza 1994). In **GP**, the user must decide how many sub-functions (called *ADF* in the **GP**) can be created, the number of formal arguments in each sub-function and whether these sub-functions can invoke one another.
- A special execution model must be used to run programs with iteration. This model imposes a restriction on where iterations can be introduced in the final programs. This requirement implies that the user must know in advance that the programs being found have iteration.

Since **ILP** and **GP** have their own pros and cons, this observation motivates the integration of the two approaches. In this paper, a system called the Genetic Logic Programming System (**GLPS**) is presented. It is a novel framework for combining the search power of Genetic Algorithms and knowledge representation power of first order logic. The shortcomings mentioned above could also be alleviated or eliminated. Currently, **GLPS** can learn function free first order logic programs with constants. Section 2 presents a description of the mechanism used to generate the initial population of programs. One of the genetic operators, crossover, is detailed in section 3. Section 4 presents a high level description of **GLPS**. The results of some sample applications are presented in the section 5. Discussion and conclusion appear in the last section.

2. Reprsentations of logic programs

GLPS uses first order logic and logic programming to represent knowledge and algorithms and can induce logic programs by **GA**. In this section, we present the representation method of logic programs. Let us start by introducing some definitions. A variable is represented by a question mark ? followed by a string of letters and digits. For example ?x is a variable. A function symbol is a letter followed by a string of letters and digits. A predicate symbol is a letter followed by a string of letters and digits. The negation symbol is ~. A term is either a variable, a function or a constant. A function is represented by a function symbol followed by a bracketed n-tuple of terms. A constant is simply a function symbol without any arguments. For example, father(mother(John)) is a function and John is a constant when father, mother and John are function symbols. A predicate symbol immediately followed by a bracketed n-tuple of terms is called an atomic formula. The Genetic Logic Programming System (**GLPS**) allows atomic formula with variables and constants but does not allow them to contain funcaaation symbols.

If A is an atomic formula, A and ~A are both literals. A is called a positive literal and ~A is a negative literal. A clause is a finite set of literals, it represents the disjunction of its literals. The clause $\{L_1, L_2, ..., \sim L_i, \sim L_{i+1}, ...\}$ can be represented as $L_1, L_2, ... \leftarrow L_i, L_{i+1}, ...$. A Horn clause is a clause which contains exactly one positive literal. The positive literal in a Horn clause is called the head of the clause while the negative literals are called the body of the clause. A set of clauses represents the conjunction of its clauses. A logic program P is a set of Horn clauses. A rule in a logic program P for the atomic formula L is a collection of Horn clauses each with the head L.

In **GLPS**, populations of logic programs are genetically bred using the Darwinian principle of survival and reproduction of the fittest along with a genetic crossover operation appropriate for mating logic programs. The fundamental difficulty in **GLPS** is to represent logic programs appropriately so that initial population can be generated easily and the genetic operators such as crossover and reproduction can be performed effectively. A logic program can be represented as a forest of AND-OR trees. The leaves of an AND-OR tree are positive or negative literals generated using the predicate symbols and terms of the problem domain. For example, consider the following logic program:

```
C1:   cup(?x) :- insulate_heat(?x), stable(?x), liftable(?x).
C2:   cup(?x) :- paper_cup(?x).

C3:   stable(?x) :- bottom(?x, ?b), flat(?b).
C4:   stable(?x) :- bottom(?x, ?b), concave(?b).
C5:   stable(?x) :- has_support(?x).

C6:   liftable(?x) :- has(?x, ?y), handle(?y).
C7:   liftable(?x) :- small(?x), made_from(?x, ?y),
                      low_density(?y).
```

The predicate symbols are {cup, insulate_heat, stable, liftable, paper_cup, bottom, flat, concave, has_support, has, handle, small, made_from, low_density} and the terms are {?x, ?y, ?b}. This program can be represented as a forest of AND-OR trees (figure 1).

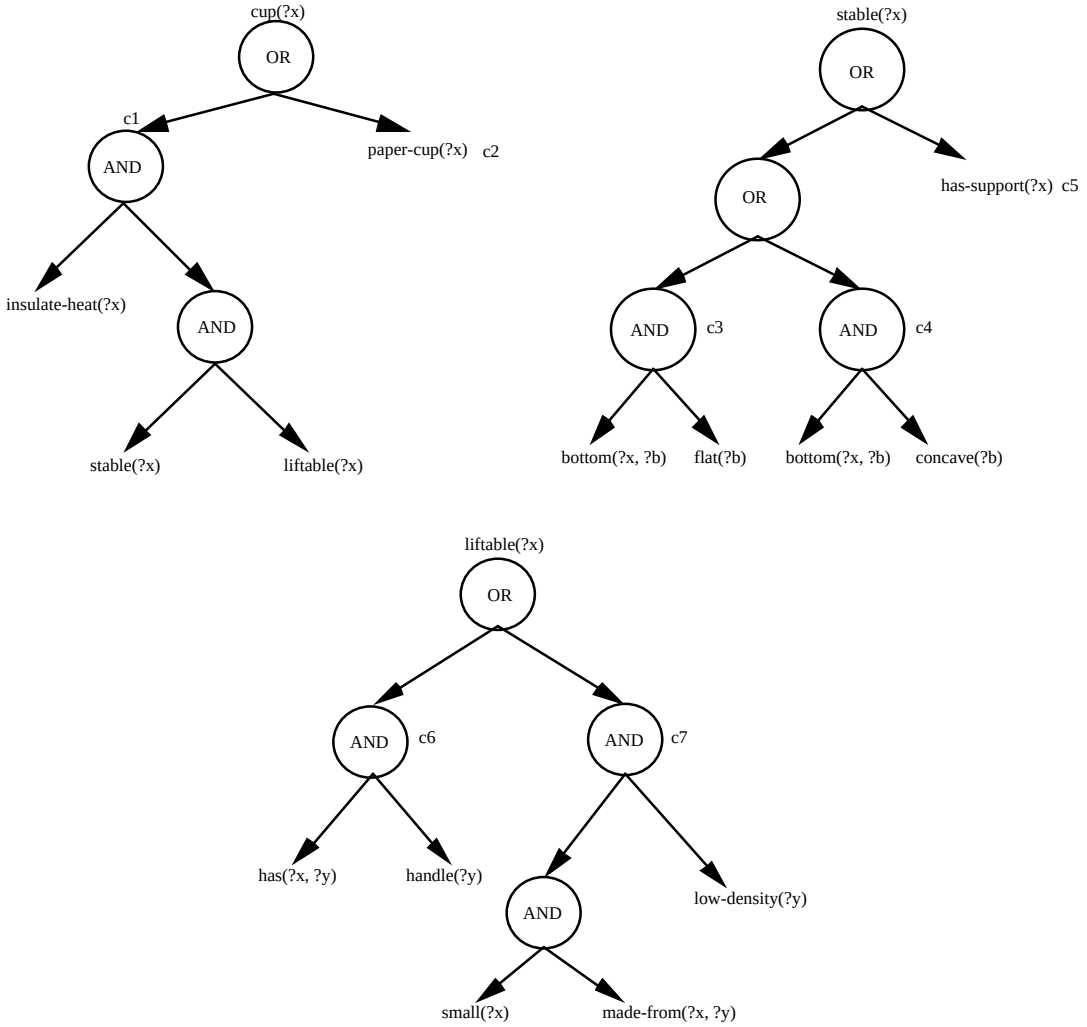


Figure 1: A representation of a program

Since a logic program can be represented as a forest of AND-OR trees, we can randomly generate a forest of AND-OR trees for the program and randomly fill the leaves of these trees with literals of the problem. The high level description of the algorithm used to generate an initial population is depicted in figure 2. For the above example, if the target concept is cup, the sub-concepts are stable and liftable and the terms are {?x, ?y, ?b}, the algorithm generates the following logic programs randomly:

```
C1':  cup(?x) :- bottom(?y), handle(?b).
C2':  cup(?x) :- small(?x), insulate_heat(?y).

C3':  stable(?b) :- cup(?b), paper_cup(?x), flat(?y), flat(?x).

C4':  liftable(?x) :- liftable(?y).
C5':  liftable(?y) :- concave(?y).
```

Alternatively, an initial population of logic programs can be induced by other learning systems, such as a variation of **FOIL** (Quinlan 1990), using a portion of the training examples. Then a forest of AND-OR trees can be generated for each logic program learned. If there are more than one representation for a logic program, one of them will be selected randomly.

Assume the predicate symbols Pred is $\{p_1, p_2, ..., p_n\}$ and the terms are $\{ t_1, t_2, ..., t_m\}$. A special symbol in Pred is the target concept (Target) and some other symbols in Pred are the sub-concepts (Sub) to be learned. If there is no sub-concepts in the target logic programs, Sub is empty. All other predicate symbols represent operational concepts and must be defined by either extensional tuples or built-in operations.

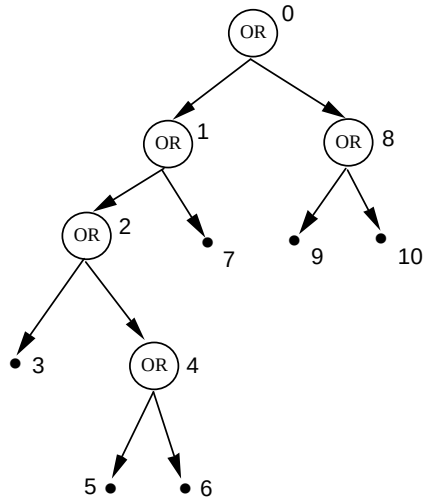
```
Let Depth be an input parameter that specifies the maximum depth of
  the AND-OR trees created.
Let Balance be an input parameter that controls whether balance or
  unbalance AND-OR trees will be generated.
Let All-concepts be the union of Target and Sub.

For all concepts in All-concepts do
  1.  Create an AND-OR tree for the current concept.
  2.  The leaves of the AND-OR tree are selected from literals in the
      domain.
  3.  Store the AND-OR tree as a rule in the logic program.
```

Figure 2: Algorithm for generating an initial population randomly

3. Crossover of logic programs

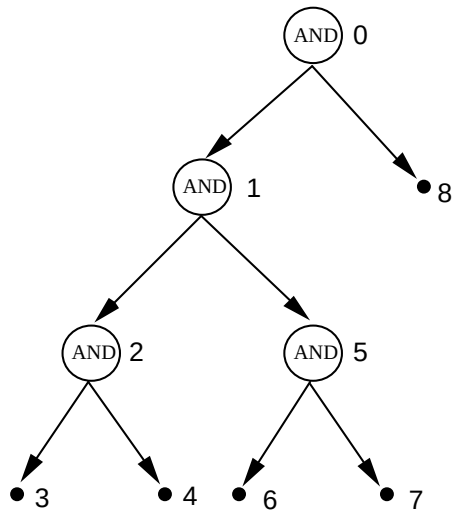
We can apply crossover to the omponents of a logic program including the whole logic program, the rules, the clauses and the antecedent literals. In **GLPS**, the terms of literals cannot be exchanged. Thus crossover components are referred to by a list of numbers. The list can have at most three elements:

1. {} refers to the whole logic program.
2. {m} refers to the mth rule in the program. A rule has one or more clauses.
3. {m, n'} refers to a clause or a number of clauses of the mth rule in the program where n' is a node number of the corresponding sub-tree. For instance, let the mth rule has N_m clauses which are arranged in an OR-tree as follows:



Each leaf in the tree represents a clause. In the example, the tree has six clauses, i.e. $N_m = 6$. There are 11 nodes in the tree, and the number of nodes is denoted by N'_m. n' in the list {m, n'} is between 0 and N'_{m-1}. Thus, { m, n'} represents a clause if n' corresponds to a leaf node. It refers to a set of clauses if n' corresponds to an internal node in the tree.

4. {m, n, l'} refers to a literal or a set of literals of the nth clause of the mth rule where l' is also a node number of the corresponding sub-tree. For example, let the clause has $L_{m,n}$ antecedent literals. These literals are arranged in an AND-tree as follows:



Each leaf in the tree represents an antecedent literal and there are 5 antecedent literals, i.e. $L_{m,\ n} = 5$. Let the number of nodes in an AND tree be $L'_{m,n}$ which is 9 for the above tree. The third number in {m, n, l'} can have value between 0 and $L'_{m,\ n-1}$. {m, n, l'} represents a literal if l' refers to a leaf node. It is a set of literals if l' refers to an internal node.

There are four kinds of crossover points represented by the above lists of numbers. Two crossover points are compatible if their representations (i.e. lists) have the same number of elements. In **GLPS**, crossover between two parental programs can only occur at compatible crossover points. Consider the following logic program, $Prog_1$, represented in Horn clauses:

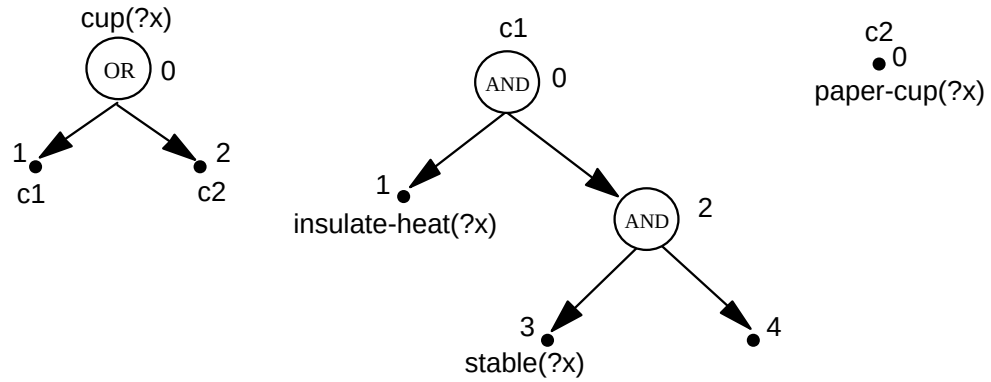
```
C1:   cup(?x) :- insulate_heat(?x), stable(?x), liftable(?x)
C2:   cup(?x) :- paper_cup(?x)

C3:   stable(?x) :- bottom(?x, ?b), flat(?b)
C4:   stable(?x) :- bottom(?x, ?b), concave(?b)
C5:   stable(?x) :- has_support(?x)

C6:   liftable(?x) :- has(?x, ?y), handle(?y)
C7:   liftable(?x) :- small(?x), made_from(?x, ?y),
                      low_density(?y)
```

and the following logic program, $Prog_2$:

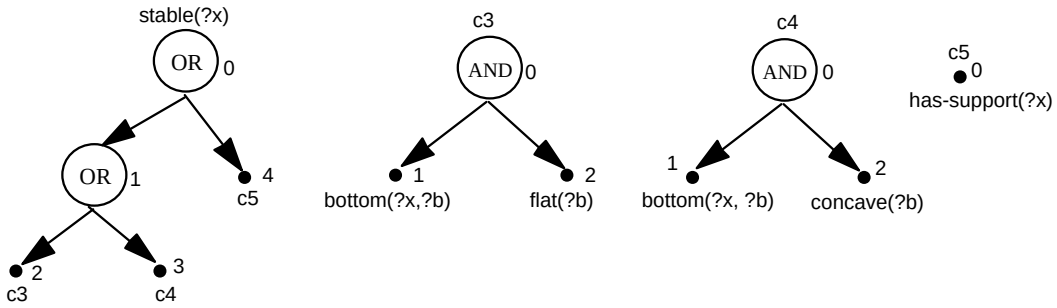
```
C1':  cup(?x) :- insulate_heat(?x), stable(?x)

C2':  stable(?x) :- bottom(?x, ?b), flat(?b), concave(?b),
                    has_support(?x)
```

The AND-OR trees of $Prog_1$ and $Prog_2$ are depicted respectively in figures 3 and 4.

Rule for cup(?x) - The first rule of the program



Rule for stable(?x) - The second rule of the program



Rule for liftable(?) - The third rule of the program

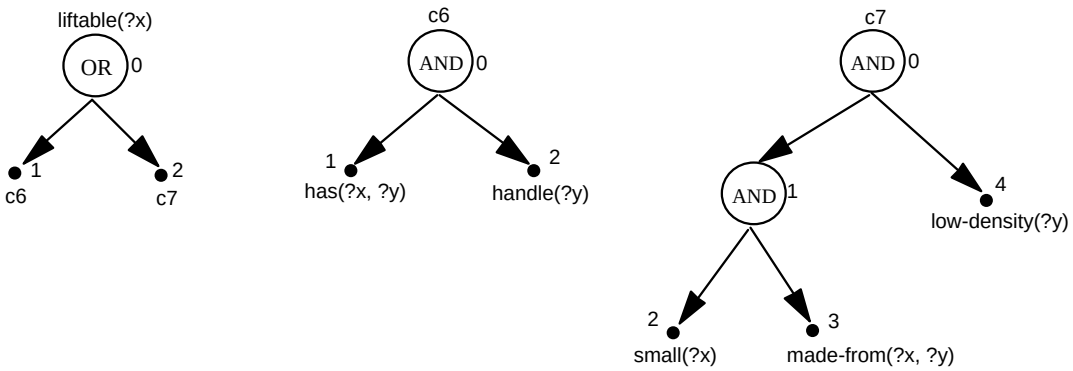
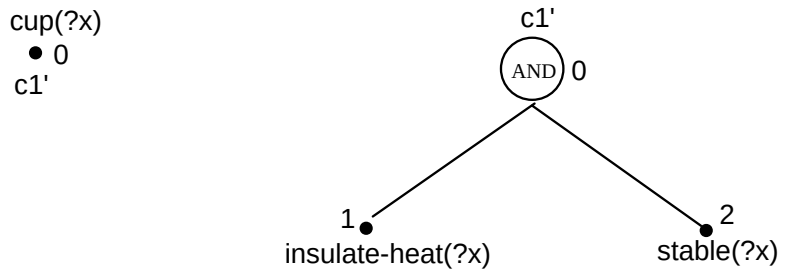


Figure 3: The And-Or tree of program $Prog_1$

Rule for cup(?x) - The first rule of the program



Rule for stable(?x) - The second rule of the program

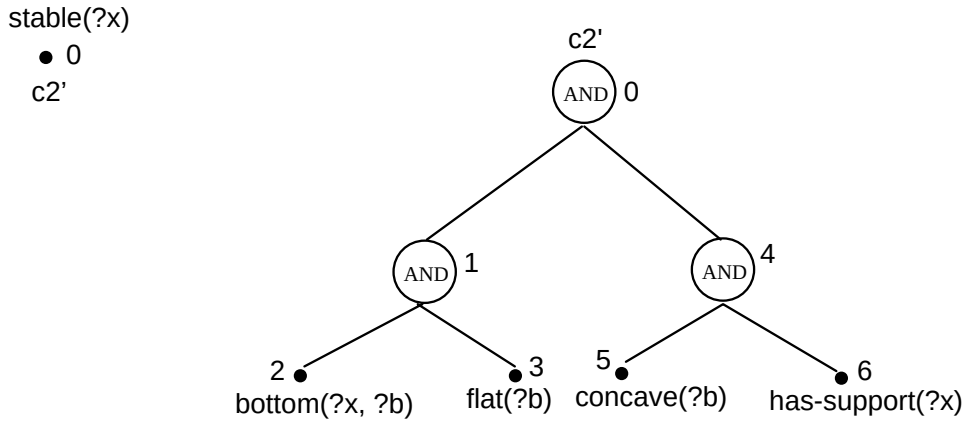


Figure 4: The And-Or tree of program $Prog_2$

If the crossover points are empty lists {}, the offspring are identical to their parents and the crossover operation degenerates into reproduction. Thus, **GLPS** has no independent reproduction operation. Reproduction is emulated by crossover at node 0. There is a parameter P_0 which controls the probability of reproduction.

The parameter P_1 controls the probability of a list with only one element being generated. For instance, if the crossover points are {2} and {2}, the offspring are:

```
C1:   cup(?x) :- insulate_heat(?x), stable(?x), liftable(?x)
C2:   cup(?x) :- paper_cup(?x)

C2':  stable(?x) :- bottom(?x, ?b), flat(?b),
                    concave(?b), has_support(?x)

C6:   liftable(?x) :- has(?x, ?y), handle(?y)
C7:   liftable(?x) :- small(?x), made_from(?x, ?y),
                      low_density(?y)
```

and

```
C1':  cup(?x) :- insulate_heat(?x), stable(?x)

C3:   stable(?x) :- bottom(?x, ?b), flat(?b)
C4:   stable(?x) :- bottom(?x, ?b), concave(?b)
C5:   stable(?x) :- has_support(?x)
```

The parameter P_2 determines the probability that a list of two elements is generated. If the crossover points are {2, 1} for $Prog_1$ and {2, 0} for $Prog_2$, the offspring are:

```
C1:   cup(?x) :- insulate_heat(?x), stable(?x), liftable(?x)
C2:   cup(?x) :- paper_cup(?x)

C2':  stable(?x) :- bottom(?x, ?b), flat(?b),
                    concave(?b), has_support(?x)
C5:   stable(?x) :- has_support(?x)

C6:   liftable(?x) :- has(?x, ?y), handle(?y)
C7:   liftable(?x) :- small(?x), made_from(?x, ?y),
                      low_density(?y)
```

and

```
C1':  cup(?x) :- insulate_heat(?x), stable(?x)

C3:   stable(?x) :- bottom(?x, ?b), flat(?b)
C4:   stable(?x) :- bottom(?x, ?b), concave(?b)
```

The parameter P_3 determines the probability that a list of three elements is created. If the crossover points are {2, 3, 0} for $Prog_1$ and {2, 0, 1} for $Prog_2$, the offspring are:

```
C1:   cup(?x) :- insulate_heat(?x), stable(?x), liftable(?x)
C2:   cup(?x) :- paper_cup(?x)

C3:   stable(?x) :- bottom(?x, ?b), flat(?b)
C4:   stable(?x) :- bottom(?x, ?b), flat(?b)
C5:   stable(?x) :- has_support(?x)

C6:   liftable(?x) :- has(?x, ?y), handle(?y)
C7:   liftable(?x) :- small(?x), made_from(?x, ?y),
                      low_density(?y)
```

and

```
C1':  cup(?x) :- insulate_heat(?x), stable(?x)

C2':  stable(?x) :- bottom(?x, ?b), concave(?b),
                    concave(?b), has_support(?x)
```

Hence, the crossover operation has many effects depending on the crossover points and only generates syntically valid logic programs.

4. The Genetic Logic Programming System (GLPS)

This section presents the evolutionary process performed by **GLPS**. It starts with an initial population of first-order concepts generated randomly, induced by other learning systems, or provided by the user. The initial logic programs are composed of the predicate symbols, the terms and the atomic formulas of the problem domain. An atomic formula can be defined extensionally as a list of tuples for which the formula is true or intensionally as a set of Horn clauses that can compute whether the formula is true. Intensional atomic formulas can also be standard built-in formulas that perform arithmetic, input/output and logical functions etc.

For concept learning (De Jong et al 1993), each individual logic program in the population is measured in terms of how well it covers positive examples and excludes negative examples. This measure is the fitness function of **GLPS**. Typically, each logic program is run over a number of training examples so that its fitness is measured as the total number of misclassified positive and negative examples. Sometimes, if the distribution of positive and negative examples is extremely uneven, this fitness function is not good enough to focus the search. For example, assume that there are 2 positive and 10000 negative examples, if the number of misclassified examples is used as the fitness function, a logic program that deduces everything are negative will have very good fitness. Thus, in this case, the fitness function should be a weighted sum of the total numbers of misclassified positive and negative examples. **GLPS** can also learn logic programs computing arithmetic functions such as square root or factorial. In this case, the fitness function calculates the difference between the outputs found by the logic program and the results of the target arithmetic function.

The initial logic programs in generation 0 are normally incorrect and have poor performances. However, some individuals in the population will be fitter than others. The Darwinian principle of reproduction and survival of the fittest and the genetic operation of sexual crossover are used to create new offspring population of programs from the current population. The reproduction operation involves selecting a program from the current population of programs and allowing it to survive by copying it into the new population. The selection is based on either fitness (fitness proportionate selection) or tournament (tournament selection).

The genetic process of crossover is used to create two offspring programs from the parental programs selected by either fitness proportionate or tournament selection. The parental programs are usually of different sizes and structures and the offspring programs are composed of the clauses and the literals from their parents. These offspring programs are typically of different sizes and structures from their parents. The new generation replaces the old generation after the reproduction and crossover operations are performed on the old generation. The fitness of each program in the new generation is estimated and the above process is iterated over many generations until the termination criterion is satisfied.

The algorithm will produce populations of programs which tend to exhibit increasing average fitness in producing correct answers for the training examples. **GLPS** returns the best logic program found in any generation of a run as the result. A high level description of **GLPS** is presented in figure 5.

Assume that the problem domain has a set of n predicate symbols Pred = $\{p_1, p_2, ..., p_n\}$ and a set of m terms Terms = $\{t_1, t_2, ..., t_m\}$. These predicate symbols and terms can be used to generate various positive and negative literals.

1. Generate an initial population of logic programs that is composed of the predicate symbols Pred, the terms Terms and atomic formulas of the domain.
2. Execute each logic program in the population and assign it a fitness value according to how well it covers positive examples and excludes negative examples.
3. Create a new population of logic programs by applying the two primary genetic operations: reproduction and crossover. The operations are applied to logic programs in the population selected by fitness proportionate or tournament selections.
4. If the termination criterion is not satisfied, go to step 2.
5. The single best program in the population produced during the run is designated as the result of the run of genetic programming.

Figure 5: High level description of GLPS

5. Applications

A preliminary implementation of **GLPS** is completed. It is implemented in CLOS (Common Lisp Object System). It has been tested on various CLOS implementations and different hardware platforms including CMU Common Lisp on a SparcStation, Lucid Common Lisp on a DecStation and MCL on a Macintosh.

Four applications of **GLPS** on learning are given below as demonstrations, namely, the Winston's arch problem, the modified Quinlan's network reachability problem, the factorial problem and the chess-endgame problem. Five runs are performed for the first three problems and fifty runs are preformed for the last problem. The parameters P_0, P_1, P_2 and P_3 are 0.0, 0.1, 0.3 and 0.6 respectively. The maximum number of generations of each run is 50 for the first two problems, 20 for the third problem and 50 for the last problem.

5.1. The Winston's arch problem

In this learning task, the objective is to learn the nature of arches from examples (Winston 1975). The domain has several operational relations as follows:

1) supports(?A, ?B) -- ?A supports ?B
2) left-of(?A, ?B) -- ?A is on the left of ?B
3) touches(?A, ?B) -- ?A touches ?B
4) brick(?A) -- ?A is a brick
5) wedge(?A) -- ?A is a wedge
6) parallel-piped(?A) -- ?A is a brick or a wedge.

The non-operational relation arch(?A, ?B, ?C) contains all tuples <?A, ?B, ?C> that form an arch with lintel ?A. There are 2 positive and 1726 negative training examples. Since the number of negative examples is much larger than that of positive examples, the standardized fitness is the weighted number of misclassified examples. Each misclassified positive example has a weight of 863 while the negative one has a

weight of 1. The predicate symbols are the operational and non-operational predicates described. The terms are {?A, ?B, ?C} and the population size is 1000. The maximum number of generations is 50. **GLPS** can find a near correct program within 2 generations. One of the best programs induced is:

```
arch(?A, ?B, ?C) :- left-of(?C, ?B), wedge(?C)
arch(?A, ?B, ?C) :- left-of(?B, ?C), supports(?B, ?A)
```

The standard solution of this problem is:

```
arch(?A, ?B, ?C) :- left-of(?B, ?C), supports(?B, ?A),
                    ~touches(?B, ?C)
```

and it is similar to the second clause of the program induced. Figure 6 delineates the best, average and worst standardized fitnesses for increasing generations.

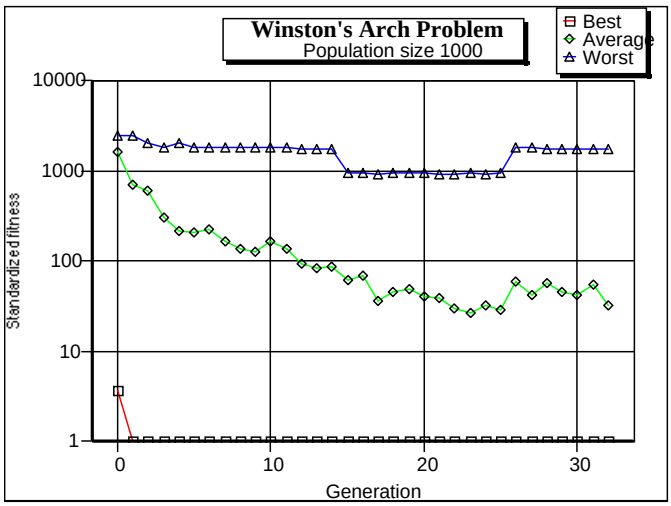
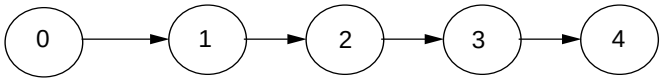


Figure 6: Performance for the Winston's Arch problem

5.2. The modified Quinlan's network reachability problem

The network reachability problem is originally proposed by Quinlan (Quinlan 1990), the domain involves a directional network such as the one depicted as follows:



The structural information of the network is the literal linked-to(?x, ?y) denoting that node ?x is directly linked to node ?y. The extension of linked-to(?x, ?y) is {<0, 1>, <1, 2>, <2, 3>, <3, 4>}. Here, the learning task is to induce a logic program that determines whether a node ?x can reach another node ?y. This problem can also be formulated as finding the intensional definition of the relation can-reach(?x, ?y) given its extension. Its extensional definition is {<0, 1>, <0, 2>, <0, 3>, <0, 4>, <1, 2>, <1, 3>, <1, 4>, <2, 3>, <2, 4>, <3, 4>}. The tuples of this relation are the positive training examples and {<0, 0>, <1, 0>, <1, 1>, <2, 0>, <2, 1>, <2, 2>, <3, 0>, <3, 1>, <3, 2>, <3, 3>, <4, 0>, <4, 1>, <4, 2>, <4, 3>, <4, 4>} are the negative examples.

In this experiment, the predicate symbols are can-reach and linked-to. The symbol can-reach represents the target concept while linked-to is an operational concept. The terms are {?x, ?y, ?z}. The population size is 1000. The standardized fitness is the total number of misclassified training examples. The maximum number of generations is 50. **GLPS** can find a perfect program that covers all positive examples while excludes all negative ones within a few generations. One program found is:

```
can-reach(?x, ?y) :- linked-to(?z, ?y), linked-to(?x, ?z)
can-reach(?x, ?y) :- linked-to(?x, ?y), linked-to(?x, ?z)
can-reach(?x, ?y) :- can-reach(?x, ?z), can-reach(?z, ?y)
```

This program can be simplified to

```
can-reach(?x, ?y) :- linked-to(?x, ?z), linked-to(?z, ?y)
can-reach(?x, ?y) :- linked-to(?x, ?y)
can-reach(?x, ?y) :- can-reach(?x, ?z), can-reach(?z, ?y)
```

The first clause of this program declares that a node ?x can reach a node ?y if there is another node ?z that directly connects them. The second clause declares that a node ?x can reach a node ?y if they are directly connected. The third clause is recursive. It expresses that a node ?x can reach a node ?y if there is another node ?z, such that ?z is reachable from ?x and ?y is reachable from ?z. In fact, this program is semantically equivalent to the standard solution

```
can-reach(?x, ?y) :- linked-to(?x, ?y)
can-reach(?x, ?y) :- linked-to(?x, ?z), can-reach(?z, ?y)
```

This experiment demonstrates that **GLPS** can learn recursive program naturally and effectively. Recursive functions are difficult to learn in Koza's **GP** (Koza, 1992), this experiment shows the advantage of Genetic Logic Programming (**GLP**) over **GP**. Figure 7 depicts the best, average and worst standardized fitnesses for increasing generations.

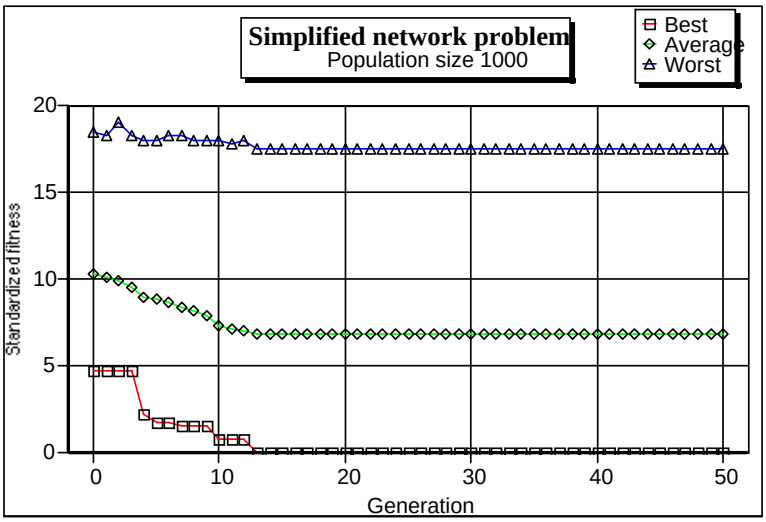


Figure 7: Performance for the modified network reachability problem

5.3. The factorial problem

This experiment learns the relation factorial(?A, ?B) where ?B is the factorial of ?A. The predicate symbols are factorial, plus and multiplication. The symbol factorial represents the target concept while plus and multiplication are built-in predicates used to perform arithmetic operations. The literal plus(?x, ?y, ?z) finds the sum of ?x and ?y and assigns the output to ?z if ?x and ?y are instantiated and ?z is not instantiated. It finds the difference of ?z and ?x and assigns the result to ?y if ?x and ?z are instantiated and ?y is not instantiated. It calculated the difference of ?z and ?y and assigns the output to ?x if ?z and ?y are instantiated and ?x is not instantiated. If ?x, ?y and ?z are all instantiated, the literal plus(?x, ?y, ?z) is satisfied if the sum of ?x and ?y is equal to ?z. The literal is not satisfied if more than one variable is not instantiated.

The literal multiplication(?x, ?y, ?z) finds the product of ?x and ?y and assigns the output to ?z if ?x and ?y are instantiated and ?z is not instantiated. It divides ?z by ?x and assigns the result to ?y if ?x and ?z are instantiated and ?y is not instantiated. It divides ?z by ?y and assigns the output to ?x if ?z and ?y are instantiated and ?x is not instantiated. If ?x, ?y and ?z are all instantiated, the literal multiplication(?x, ?y, ?z) is satisfied if the product of ?x and ?y is equal to ?z. The literal is not satisfied if more than one variable is not instantiated.

The terms are {0, 1, 2, ?w, ?x, ?y, ?z}. The population size is 1000 and the maximum number of generations is 20. The standardized fitness of a program is defined as follows:

$$\sum_i \min[1,\ abs\ (\frac{prog\_factorial(i) - factorial(i)}{factorial(i)})]$$

```
where i is the input value;
      factorial(i) returns the correct result for the input i;
and
      prog_factorial(i) returns the result of the logic program
      for the input i
```

In this experiment, we use five fitness cases for i from 0 to 4. Since the search space of this problem is extremely large, a number of incorrect initial clauses are used to create the initial population of programs. An individual program contains a random subset of clauses from these incorrect initial clauses. The clauses are as follows:

```
factorial(0, 1)   :- plus(1, 1, 2).
factorial(1, 1)   :- plus(1, 1, 2).
factorial(?x, ?y) :- plus(?z, 1, ?x), plus(?x, ?y ?z).
factorial(?x, ?y) :- plus(?z, ?x, ?y) factorial(?z, ?w),
                     multiplication(?w, ?x, ?y).
factorial(1, 1)   :- plus(1, 1, 2), multiplication(?x, ?x, ?y).
factorial(?x, ?y) :- plus(?z, 1, ?x),
                     multiplication(?z, ?z, ?w),
                     multiplication(?w, ?x, ?y).
factorial(?x, ?y) :- factorial(?z, ?w),
                     multiplication(?w, ?x, ?y),
                     multiplication(?x, ?y, ?z).
factorial(?x, ?y) :- plus(?x, ?x, ?w),
                     multiplication(?w, ?w, ?z),
                     multiplication(?z, ?x, ?y).
factorial(?x, ?y) :- multiplication(?x, ?x, ?w),
                     factorial(?w, ?z), plus(?z, ?x, ?y).
```

During one of the runs, the correct logic program is induced in the eighth generation. The program is

```
factorial(0, 1)   :- plus(1, 1, 2).
factorial(?x, ?y) :- factorial(?z, ?w),
                     multiplication(?w, ?x, ?y),
                     multiplication(?x, ?y, ?z).
factorial(?x, ?y) :- plus(?z, ?x, ?y), factorial(?z, ?w),
                     multiplication(?w, ?x, ?y).
factorial(0, 1)   :- multiplication(?w, 0, 1).
factorial(?x, ?y) :- multiplication(?w, ?x, ?y),
                     multiplication(?w, ?x, ?y),
                     multiplication(?x, ?y, ?z).
factorial(1, 1)   :- plus(1, 1, 2), multiplication(?x, ?x, ?y).
factorial(?x, ?y) :- plus(?z, 1, ?x), factorial(?z, ?w),
                     multiplication(?w, ?x, ?y).
factorial(?x, ?y) :- plus(?z, 1, ?x), plus(?x, ?y, ?z).
```

In this program, the first term of the factorial literal should be an instantiated input value and the second term should be the output result. Some clauses in this program can be eliminated because they contain arithmetic literals with more than one un-instantiated variable. For example, the third clause can be removed. Thus, the program is simplified to

```
factorial(0, 1)    :- plus(1, 1, 2).
factorial(0, 1)    :- multiplication(?w, 0, 1).
factorial(?x, ?y)  :- plus(?z, 1, ?x), factorial(?z, ?w),
                      multiplication(?w, ?x, ?y).
factorial(?x, ?y)  :- plus(?z, 1, ?x), plus(?x, ?y, ?z).
```

Since the second clause in the simplified program cannot be satisfied in every situation, it is removed from the program too. Although the last clause is incorrect, it will not be used during execution, so it is eliminated too. The final program is

```
factorial(0, 1)    :- plus(1, 1, 2).
factorial(?x, ?y)  :- plus(?z, 1, ?x), factorial(?z, ?w),
                      multiplication(?w, ?x, ?y).
```

which is a correct logic program to find the factorial of a number. Figure 8 depicts the best, average and worst standardized fitnesses against increasing generations.

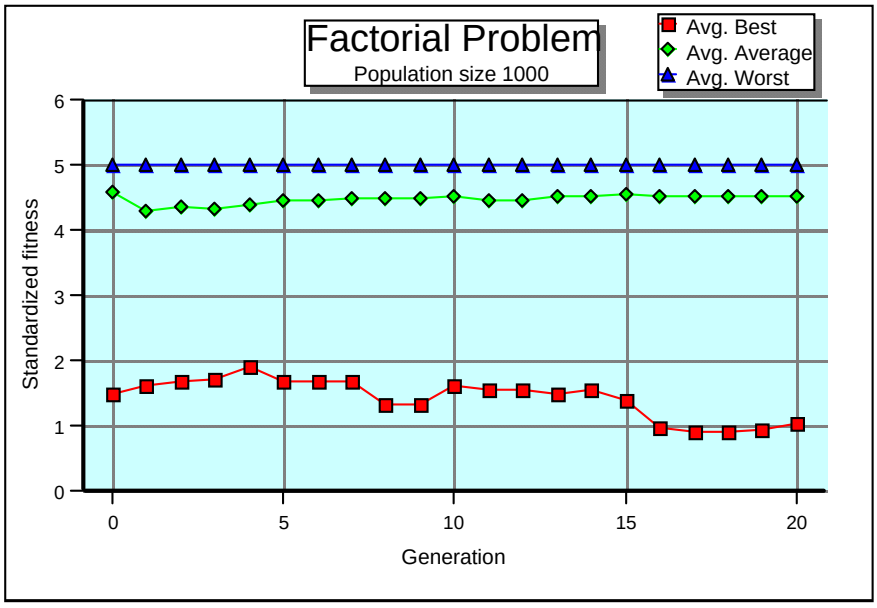


Figure 8: Performance for the factorial problem

5.4. Learning program from imperfect data

In knowledge discovery from databases, we emphasize the need for learning from huge, incomplete and imperfect data sets (Piatetsky-Shapiro and Frawley 1991). Existing inductive learning systems employ noise-handling mechanisms to cope with different kinds of data imperfections such as noise, insufficiently covered example space, inappropriate description language and missing values in the training examples (Lavrac and Dzeroski 1994). These mechanisms include tree pruning, rule truncation, and significant test.

However, most existing learning systems use attribute-value language for representing the training examples and induced knowledge and allow a finite number of objects in the universe of discourse. This representation limits them to learn only propositional descriptions in which concepts are described in terms of values of a fixed number of attributes. Currently, only a few relation learning systems such as **FOIL** address the issue of learning from imperfect data. This experiment describes

the application of **GLPS** to learn logic programs from noisy examples. An empirical comparison of **GLPS** and **FOIL6** (a version of **FOIL**) in the domain of learning illegal chess endgame positions from noisy examples is conducted.

In **FOIL**, the noise handling mechanism is the encoding length restriction. The idea is that the number of bits required to encode the clause should never exceed the total number of bits needed to indicate explicitly the positive training examples covered by the clause. Thus, if a clause covers r positive examples out of n examples in the training set, the number of bits available to encode the clause is $\log_2(n) + \log_2(\binom{n}{r})$. If there are no bits available for adding another literal, but the clause has more than 85% accuracy, it is retained in the induced set of clauses, otherwise it is deleted. This heuristic avoids overfitting the training examples because insignificant literals are excluded from clauses of the inducing concept. The acquired concept description is thus smaller, simpler, more accurate and more comprehensible. Lavrac and Dzeroski (Lavrac and Dzeroski 1994) argued that the encoding length restriction has two deficiencies. In exact domains, it sometimes prevent **FOIL** from learning complete description. In noisy domains, it generates very specific clauses. In this experiment, **GLPS** employs a variation of **FOIL** to find the initial population of logic programs. Thus, it uses the same noise handling mechanism of **FOIL**.

In the chess-endgame, the setup is white king and rook versus black king (Quinlan 1990). The target concept illegal(?WKf, ?WKr, ?WRf, ?WRr, ?BKf, ?BKr) states whether the positions where the white king at (?WKf, ?WKr), the white rook at (?WRf, ?WRf) and the black king at (?BKf, ?BKr) are not a legal white-to-move position. The background knowledge is represented by two predicates, adjacent(?X, ?Y) and less_than(?W, ?Z), indicating that rank/file ?X is adjacent to rank/file ?Y and rank/file ?W is less than rank/file ?Z respectively. The training set contains 1000 examples (336 positive and 664 negative examples). The testing set has 10000 examples (3240 positive and 6760 negative examples).

Different amounts of noise are introduced into the training examples in order to study the performances of both systems in learning concepts from noisy environment. To introduce n% of noise into argument ?X of the examples, the value of argument ?X is replaced by a random value of the same type from a uniform distribution, independent to noise in other arguments. For the class variable, n% positive examples are labeled as negative ones while n% negatives examples are labeled as positive ones. Noise in an argument is not necessarily incorrect because it is chosen randomly, it is possible that the correct argument value is selected. In contrast, noise in classification implies that this example is incorrect. Thus, the probability for an example to be incorrect is $1 - \{[(1-n\%) + n\% * \frac{1}{8}]^6 * (1-n\%)\}$. In this experiment, the percentages of introduced noise are 5%, 10%, 15%, 20%, 30% and 40%. Thus, the probabilities for an example to be noisy are respectively 27.36%, 48.04%, 63.46%, 74.78%, 88.74% and 95.47%. Background knowledge and testing examples are not corrupted with noise.

A chosen level of noise is first introduced in the training set. First-order logic programs are then induced from the training set using **GLPS** and **FOIL6**. Finally, the classification accuracy of the learned logic programs is estimated on the testing set. For **GLPS**, the parameters P_0, P_1, P_2 and P_3 are 0.0, 0.1, 0.3 and 0.6 respectively. The

population size is 10 and the maximum number of generations for each experiment is 50. In fact, different population sizes (e.g. 100 and 500) have been tried and the results are still satisfactory even for a very small population. This observation is interesting and it demonstrates the advantage of combining inductive logic programming and genetic programming using the proposed framework. The fitness function evaluates the number of training examples misclassified by each individual in the population. Fifty runs of the above experiments are performed on different training examples. The results of the two systems are summarized in figure 9.

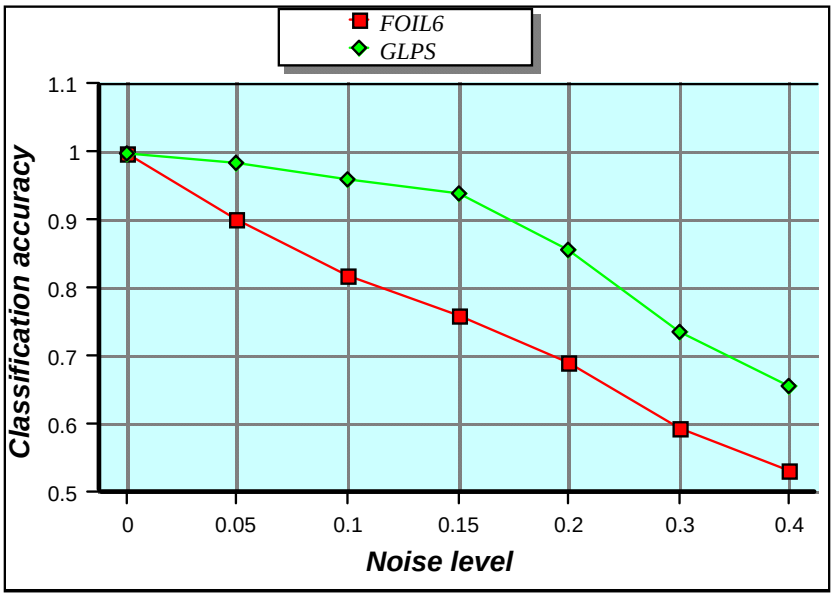


Figure 9: Comparison between GLPS and FOIL6

From this experiment, the classification accuracy of both systems degrades seriously as the noise level increases. The results were statistically evaluated using a paired *t*-test. For each noise level, they are compared to determine if their difference in accuracy is statistically significant at the 99.995% confidence interval. The classification accuracy of **GLPS** is better than that of **FOIL6**. The differences are significant at the 99.995% confidence interval at all noise levels (except the noise level of 0%). The largest difference reaches 24% at the 20% noise level. This result is surprising because both systems use the same noise handling mechanism. One possible explanation of the better performance of **GLPS** is that the Darwinian principle of survival and reproduction of the fittest is a good noise handling method. It avoids overfitting noisy examples, but at the same time, it can finds interesting and useful patterns from these noisy examples.

6. Conclusion

We have proposed a framework for inducing logic programs using genetic algorithms. A preliminary implementation of the framework has been developed and it has been tested on the following learning tasks: the Winston's arch problem, the modified Quinlan's network reachability problem, the factorial problem and the chess-endgame

problem. The experiments demonstrate that **GLPS** is a promising alternative to other famous inductive logic programming systems.

Since **GLPS** uses the same representation of other inductive logic programming systems, it is possible to combine **GLPS** with these systems. One approach is to incorporate their search operators into **GLPS**. These operators are information guided hill-climbing, explanation-based generation, explanation-based specialization and inverse resolution. **GLPS** can also invoke these systems as front-ends to generates the initial population. The advantage is that they can quickly find important and meaningful components (genetic materials) and embody these components into the initial population. Moreover, it has been found that **GLPS**, when combined with other learning systems, has superior performance in learning logic programs from imperfect data as demonstrated in the chess-endgame problem. The Darwinian principle of survival and selection of the fittest is a plausible noise handling method which can avoid overfitting and identify important patterns simultaneously. This superior noise handling ability is intrinsically embedded in **GLPS** because it uses genetic algorithms as its primary learning mechanism.

We have described how to combine **GLPS** and **FOIL** in learning first-order concepts. The initial population of logic programs is provided by a variation of **FOIL.** The performance of **GLPS** in a noisy domain has been evaluated by using the chess endgame problem. A detailed comparison to **FOIL6** (a version of **FOIL**) has been performed. It is found that **GLPS** outperforms **FOIL6** significantly in this domain. This result is very encouraging and we plan to combine **GLPS** with other learning systems such as **GOLEM** (Muggletion and Feng 1990) and **LINUS** (Lavrax and Dzeroski 1994). Another important future work is to study how to induce new literals or subroutines automatically.

Reference

De Jong, K. A., Spears, W. M. and Gordon, D. F. (1993). Using Genetic Algorithms for Concept Learning, *Machine Learning*, **13**, pp. 161-188.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor, MI: The University of Michigan Press.

Kinnear, K. E. Jr., editor (1994). *Advances in Genetic Programming*. Cambridge, MA: MIT Press.

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

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

Lavrac, N. and Dzeroski, S. (1994). I*nductive Logic Programming: Techniques and Applications*. London: Ellis Horword.

Mitchell, T. M. (1982). Generalization as Search; *Artificial Intelligence*, **18**, pp. 203-226.

Muggleton, S. and Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pp. 1-14.

Pazzani, M. and Kibler, D. (1992). The utility of knowledge in Inductive learning. *Machine Learning*, **9**, pp. 57-94.

Piatetsky-Shapiro, G. and Frawley, W. J. (1991). *Knowledge Discovery in Databases*. Menlo Park, CA: AAAI Press.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, **5**, pp. 239-266.

Winston, P. H. (1975). Learning structural descriptions from examples. In P. H. Winston (ed.), *The psychology of computer vision*. New York: McGraw-Hill.

Wong, M. L. and Leung, K. S. (1994a). Inductive Logic Programming Using Genetic Algorithms. In J. W. Brahan and G. E. Lasker (Eds.), *Advances in Artificial Intelligence - Theory and Application II*, 119-124. I.I.A.S., Ontario.

Wong, M. L. and Leung, K. S. (1994b). Learning First-order Relations from Noisy Databases using Genetic Algorithms. In *Proceedings of the Second Singapore International Conference on Intelligent Systems*, B159-164.

Wong, M. L. and Leung, K. S. (1995). An adaptive Inductive Logic Programming system using Genetic Programming. In *Proceedings of the Fourth Annual Conference on Evolutionary Programming*. MA: MIT Press.