Discovering Knowledge from Noisy Databases using Genetic Programming

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 and Engineering
The Chinese University of Hong Kong
ksleung@cse.cuhk.edu.hk

Jack C. Y. Cheng
Department of Orthopaedics and Traumatology
The Chinese University of Hong Kong
jackcheng@cuhk.edu.hk

Abstracts

In data mining, we emphasize the need for learning from huge, incomplete and imperfect data sets (Fayyad et al. 1996, Frawley et al. 1991, Piatetsky-Shapiro and Frawley, 1991). To handle noise in the problem domain, existing learning systems avoid overfitting the imperfect training examples by excluding insignificant patterns. The problem is that these systems use a limiting attribute-value language for representing the training examples and the induced knowledge. Moreover, some important patterns are ignored because they are statistically insignificant. In this paper, we present a framework that combines Genetic Programming (Koza 1992; 1994) and Inductive Logic Programming (Muggleton, 1992) to induce knowledge represented in various knowledge representation formalisms from noisy databases. The framework is based on a formalism of logic grammars and it can specify the search space declaratively. An implementation of the framework, LOGENPRO (The Logic grammar based GENetic PROgramming system), has been developed. The performance of LOGENPRO is evaluated on the chess endgame domain. We compare LOGENPRO with FOIL and other learning systems in detail and find its performance is significantly better than that of the others. This result indicates that the Darwinian principle of natural selection is a plausible noise handling method which can avoid overfitting and identify important patterns at the same time. Moreover, the system is applied to one real-life medical database. The knowledge discovered provides insights to and allows better understanding of the medical domains.

Area: Data mining, Genetic Programming, Evolutionary Computation, Rule Learning

1. Introduction

Data mining is defined as the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data stored in databases (Fayyad et al. 1996, Frawley et al. 1991, Piatetsky-Shapiro and Frawley 1991). The knowledge discovered can be expressed in different knowledge representations such as logic programs, decision trees, decision lists, and production rules.

Two of the approaches in data mining are Inductive Logic Programming (ILP) and Genetic Programming (GP). Dzeroski and Lavrac showed that ILP can be used to induce knowledge represented as logic programs (Dzeroski and Lavrac 1993, Dzeroski 1996). GP (Koza 1992; 1994, Kinnear 1994) extends traditional Genetic Algorithms (Holland 1975, Goldberg 1989, Davis 1987; 1991) to induce automatically S-expressions in Lisp. It performs both exploitation of the most promising solutions and exploration of the search space. It is featured to tackle hard search problems and thus applicable to program induction and data mining.

In this paper, we present a framework that can combine GP and ILP to induce knowledge from databases. We can also specify the search space declaratively. This framework is based on a formalism of logic grammars and is implemented as a data mining system called LOGENPRO (The LOgic grammar based GENetic PROgramming system). The formalism is powerful enough to represent context-sensitive information and domain-dependent knowledge which can be used to accelerate the learning of knowledge. It is also very flexible and the knowledge acquired can be represented in different knowledge representations such as logic programs and production rules (Wong and Leung 1995).

The problem of learning knowledge from huge, incomplete, and imperfect datasets is very important in data mining (Piatetsky-Shapiro and Frawley 1991). The various kinds of imperfections in data are: 1) random noise in training examples and background knowledge; 2) the number of training examples is too small; 3) the distribution of training examples fails to reflect the underlying distribution of instances of the concept being learned; 4) an inappropriate example description language is used: some important characteristics of examples are not represented, and/or irrelevant properties of examples are provided; 5) an inappropriate concept description language is used: it does not contain an exact description of the target concept; and 6) there are missing values in the training examples.

Existing inductive learning systems employ noise-handling mechanisms to cope with the first five kinds of data imperfections. Missing values are usually handled by a separate mechanism. These noise-handling mechanisms are designed to prevent the induced concept from overfitting the imperfect training examples by excluding insignificant patterns (Lavrac and Dzeroski 1994). They include tree pruning in CART (Breiman et al. 1984), rule truncation in AQ15 (Michalski et al.

1986) and significant test in CN2 (Clark and Niblett 1989). However, these mechanisms may ignore some important patterns because they are statistically insignificant.

Moreover, these learning systems use a limiting attribute-value language for representing the training examples and induced knowledge. 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 learning systems such as FOIL (Quinlan 1990; 1991) and mFOIL (Lavrac and Dzeroski 1994) address the issue of learning logic programs from imperfect data. Consequently, we compare LOGENPRO with FOIL and mFOIL in their performance in learning logic programs from noisy databases.

This paper is organized as follows. The next section presents the formalism of logic grammars and the details of LOGENPRO. In section 3, we employ LOGENPRO to combine GP and FOIL to induce knowledge represented as logic programs from noisy datasets. The data mining system has been applied to a real-life medical database. The results are presented in section 4. Finally, a conclusion is given in the section 5.

2. Logic grammars and LOGENPRO

The LOgic grammars based GENetic PROgramming system (LOGENPRO) can induce knowledge represented in various knowledge representation formalisms such as computer programs and production rules. Thus, LOGENPRO must be able to accept grammars of different knowledge representation languages. Most languages are specified in the notation of BNF (Backus-Naur form) which is a kind of context-free grammars (CFGs). However, LOGENPRO is based on logic grammars because CFGs (Hopcroft and Ullman 1979, Lewis and Rapadimitrion 1981) are not expressive enough to represent context-sensitive information for some languages and domain-dependent knowledge of the target knowledge being induced. This section first introduces the formalism of logic grammars followed by the descriptions of LOGENPRO.

2.1. Introduction to logic grammars

Logic grammars are the generalizations of CFGs. Their expressivenesses are much more powerful than those of CFGs, but equally amenable to efficient execution. In this paper, logic grammars are described in a notation similar to that of definite clause grammars (Pereira and Warren 1980, Pereira and Shieber 1987, Sterling and Shapiro 1986). The logic grammar for some simple S-expressions in table 1 will be used throughout this section.

A logic grammar differs from a CFG in that the logic grammar symbols, whether terminal or non-terminal, may include arguments. The arguments can be any term in the grammar. A term is

either a logic variable, a function or a constant. A variable is represented by a question mark ? followed by a string of letters and/or digits. A function is a grammar symbol followed by a bracketed n-tuple of terms and a constant is simply a 0-arity function. Arguments can be used in a logic grammar to enforce context-dependency. Thus, the permissible forms for a constituent may depend on the context in which that constituent occurs in the program. Another application of arguments is to construct tree structures in the course of parsing, such tree structures can provide a representation of the semantics of the program.

```
1:    start       ->    [(*], exp(W), exp(W), exp(W), [)].
2:    start       ->    {member(?x,[W, Z])}, [(*], exp-1(?x),
                        exp-1(?x), exp-1(?x), [)].
3:    start       ->    {member(?x,[W, Z])}, [(+], exp-1(?x),
                        exp-1(?x), exp-1(?x), [)].
4:    exp(?x)     ->    [(/ ?x 1.5)].
5:    exp-1(?x)   ->    {random(1,2,?y)}, [(/ ?x ?y)].
6:    exp-1(?x)   ->    {random(3,4,?y)}, [(- ?x ?y)].
7:    exp-1(W)    ->    [(+ (- W 11) 12)].
```

Table 1: A logic grammar

The terminal symbols enclosed in square brackets correspond to the set of words of the language specified. For example, the terminal `[(- ?x ?y)]` creates the constituent `(- 1.0 2.0)` of a program if `?x` and `?y` are instantiated respectively to 1.0 and 2.0. Non-terminal symbols are similar to literals in Prolog, `exp-1(?x)` in table 1 is an example of non-terminal symbols. Commas denote concatenation and each grammar rule ends with a full stop.

The right-hand side of a grammar rule may contain logic goals and grammar symbols. The goals are pure logical predicates for which logical definitions have been given. They specify the conditions that must be satisfied before the rule can be applied. For example, the goal `member(?x, [W, Z])` in table 1 instantiates the variable `?x` to either `W` or `Z` if `?x` has not been instantiated, otherwise it checks whether the value of `?x` is either `W` or `Z`. If the variable `?y` has not been bound, the goal `random(1, 2, ?y)` instantiates `?y` to a random floating point number between 1 and 2. Otherwise, the goal checks whether the value of `?y` is between 1 and 2.

Domain-dependent knowledge can be represented in logic goals. For example, consider the following grammar rule:

```
a-useful-program  ->    first-component(?X),
                        {is-useful(?X, ?Y)},
                        second-component(?Y).
```

This rule states that a useful program is composed of two components. The first component is generated from the non-terminal `first-component(?X)`. The logic variable `?X` is used to store semantic information about the first component produced. The logic goal then determines whether the first component is useful according to the semantic information stored in `?X`. Domain-

dependent knowledge about which program fragments are useful is represented in the logical definition of this predicate. If the first component is useful, the logic goal `is-useful(?X, ?Y)` is satisfied and some semantic information is stored into the logic variable `?Y`. This information will be used in the non-terminal `second-component(?Y)` to guide the search for a good program fragment as the second component of a useful program.

The special non-terminal `start` corresponds to a program of the language. In table 1, some grammar symbols are shown in bold-face to identify the constituents that cannot be manipulated by genetic operators. For example, the last terminal symbol **`[)]`** of the second rule is revealed in bold-face because every S-expression must be ended with a ‘)’. The number before each rule is a label for later discussions. It is not part of the grammar.

2.2. Representations of programs

One of the fundamental contributions of LOGENPRO is in the representations of programs in different programming languages appropriately so that initial population can be generated easily and the genetic operators such as reproduction, mutation, and crossover can be performed effectively. A program can be represented as a derivation tree that shows how the program has been derived from the logic grammar. LOGENPRO applies deduction to randomly generate programs and their derivation trees in the language declared by the given grammar. These programs form the initial population. For example, the program `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))` can be generated by LOGENPRO given the logic grammar in table 1. It is derived from the following sequence of derivations:

```
start =>    [(*] exp(W) exp(W) exp(W) [)]
      =>    [(*] [(/ W 1.5)] exp(W) exp(W) [)]
      =>    [(*] [(/ W 1.5)] [(/ W 1.5)] exp(W) [)]
      =>    [(*] [(/ W 1.5)] [(/ W 1.5)] [(/ W 1.5)] [)]
      =>    [(* (/ W 1.5) (/ W 1.5) (/ W 1.5))]
```

This sequence of derivations can be represented as the derivation tree depicted in figure 1.

In literature, the terms derivation trees and parse trees are usually used interchangeably. However, we will use the term derivation trees to refer to the tree structures in our framework and the term parse trees to refer to those in GP. The bindings of logic variables are shown in italic font and enclosed in a pair of braces. The sub-trees enclosed in a dashed rectangular are frozen. In other words, they are generated by bold-faced grammar symbols and they cannot be modified by genetic operators.

An advantage of logic grammars is that they specify what is a legal program without any explicit reference to the process of program generation and parsing. Furthermore, a logic grammar can be translated into an efficient logic program that can generate and parse the programs in the

language declared by the logic grammar (Pereira and Warren 1980, Pereira and Shieber 1987, Abramson and Dahl 1989). In other words, the process of program generation and parsing can be achieved by performing deduction using the translated logic program. Consequently, the program generation and analysis mechanisms of LOGENPRO can be implemented using a deduction mechanism based on the logic programs translated from the grammars.

Alternatively, initial programs can be induced by other learning systems such as FOIL (Quinlan 1990; 1991) or given by the user. LOGENPRO analyzes each program and creates the corresponding derivation tree.

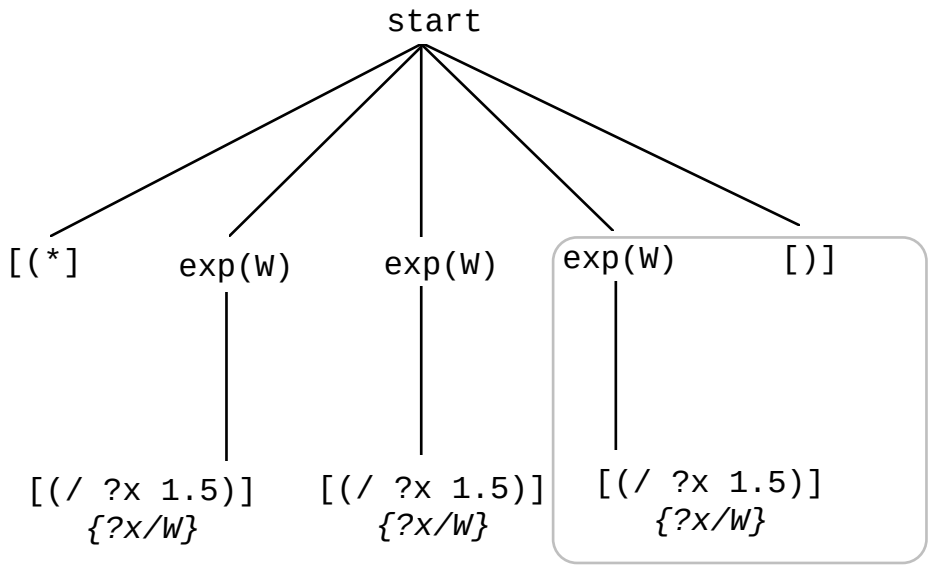

Figure 1: A derivation tree of the S-expression in Lisp `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))`

2.3. Crossover of programs

The crossover is a sexual operation that starts with two parental programs and the corresponding derivation trees. One program is designated as the primary parent and the other one as the secondary parent. Their derivation trees are called the primary and secondary derivation trees respectively. The following steps are used to produce an offspring program:

1. If there are sub-trees in the primary derivation tree that have not been selected previously, select randomly a sub-tree (primary sub-tree) from these sub-trees using a uniform distribution. The root of the selected sub-tree is called the primary crossover point. Otherwise, terminate the algorithm without generating any offspring.

2. Select another sub-tree (secondary sub-tree) in the secondary derivation tree under the constraint that the offspring produced must be valid according to the grammar.
3. If a sub-tree can be found in step 2, create and return the offspring, which is obtained by deleting the primary sub-tree and then inserting the secondary sub-tree at the primary crossover point. Otherwise, go to step 1.

Consider two parental programs generated randomly from the grammar in table 1. The primary parent is `(+ (- Z 3.5) (- Z 3.8) (/ Z 1.5))` and the secondary parent is `(* (/ W 1.5) (+ (- W 11) 12) (- W 3.5))`. The corresponding derivation trees are depicted in figures 2 and 3 respectively. In the figures, the plain numbers identify the sub-trees of these derivation trees, while the underlined numbers indicate the grammar rules used in deducing the corresponding sub-trees.

For example, if the primary and secondary sub-trees are respectively 2 and 15. The valid offspring is `(* (- Z 3.5) (- Z 3.8) (/ Z 1.5))` is obtained and its derivation tree is shown in figure 4. It is interesting to find that the sub-tree 25 has a label 2. This indicates that the sub-tree is generated by the second grammar rule rather than the third rule applied to the primary parent. The second rule must be used because the terminal symbol `[(/]` is changed to `[(*]` and only the second rule can create the terminal `[(*]`.

In another example, the primary and secondary sub-trees are 3 and 16 respectively. The valid offspring `(+ (/ Z 1.5) (- Z 3.8) (/ Z 1.5))` is produced and the derivation tree is shown in figure 5. It should be emphasized that the constituent from the secondary parent is changed from `(/ W 1.5)` to `(/ Z 1.5)` in the offspring. This must be modified because the logic variable `?x` in sub-tree 41 is instantiated to `Z` in sub-tree 39. This example demonstrates the use of logic grammars to enforce contextual-dependency between different constituents of a program.

LOGENPRO disallows the crossover between the primary sub-tree 6 and the secondary sub-tree 19. The sub-tree 19 requires the variable `?x` to be instantiated to `W`, But, `?x` must be instantiated to `Z` in the context of the primary parent. Since `W` and `Z` cannot be unified, these two sub-trees cannot be crossed over.

LOGENPRO has an efficient algorithm to check these conditions before performing any crossover. Thus, only valid offspring are produced and this operation can be achieved effectively and efficiently.

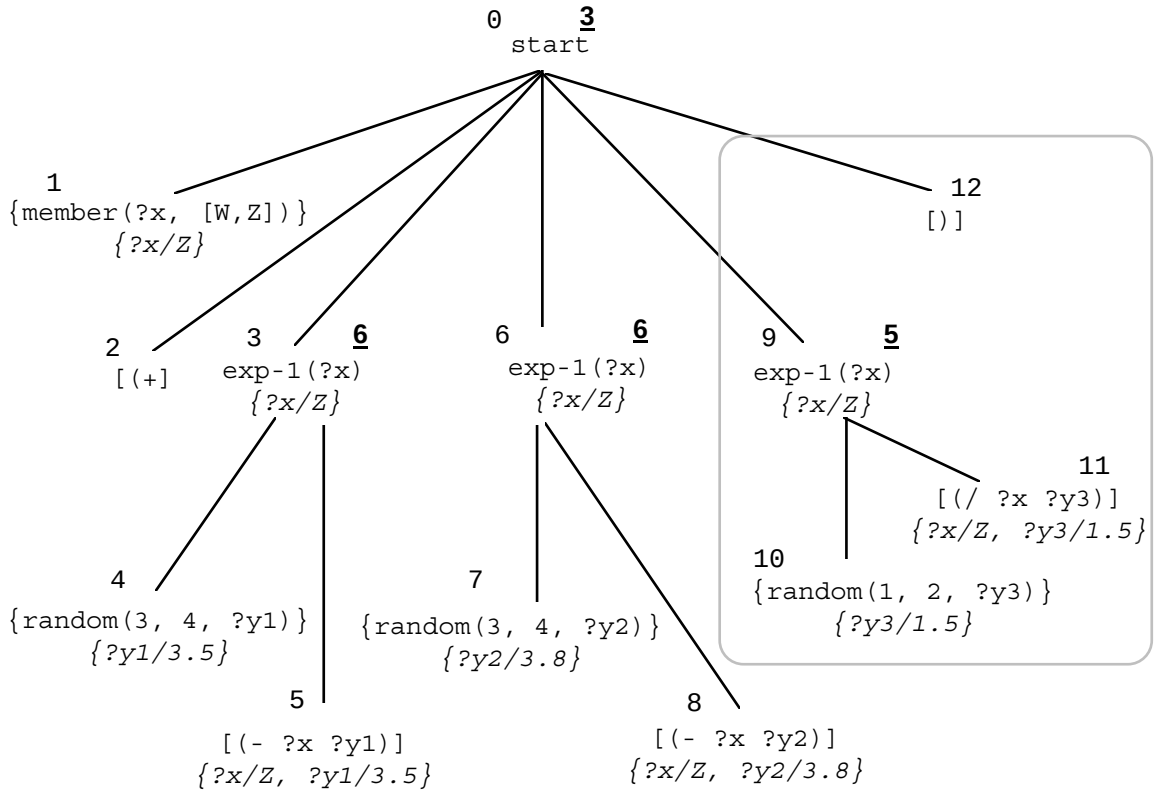

Figure 2: The derivations tree of the primary parental program (+ (- Z 3.5) (- Z 3.8) (/ Z 1.5)).

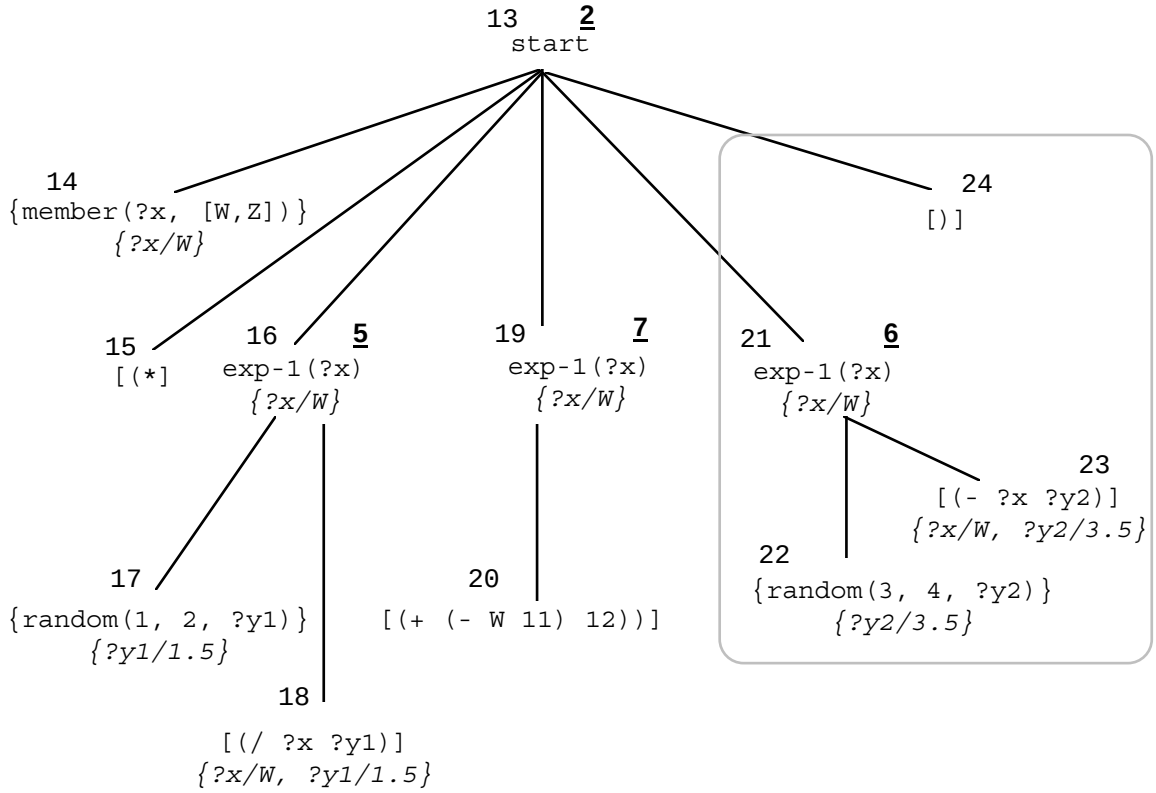

Figure 3: The derivations tree of the secondary parental program `(* (/ W 1.5) (+ (- W 11) 12) (- W 3.5))`.

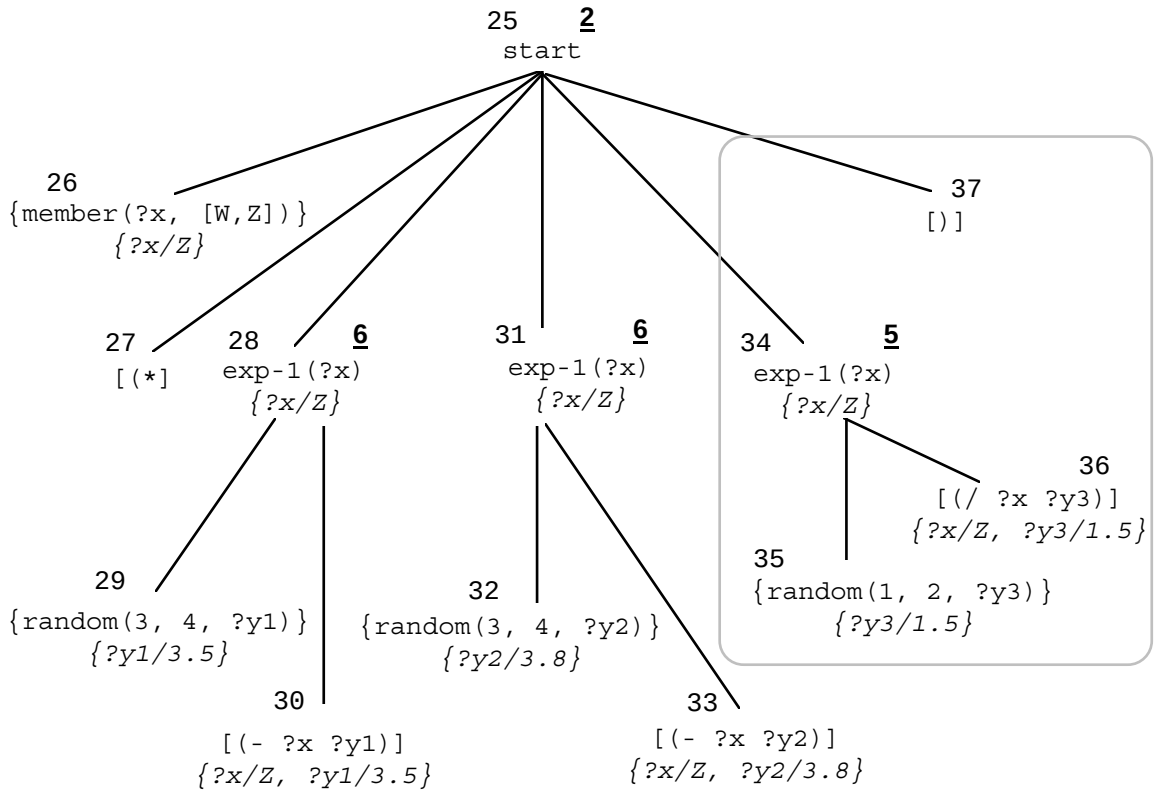

Figure 4: **A derivation tree of the offspring produced by performing crossover between the primary sub-tree 2 of the tree in figure 2 and the secondary sub-tree 15 of the tree in figure 3.**

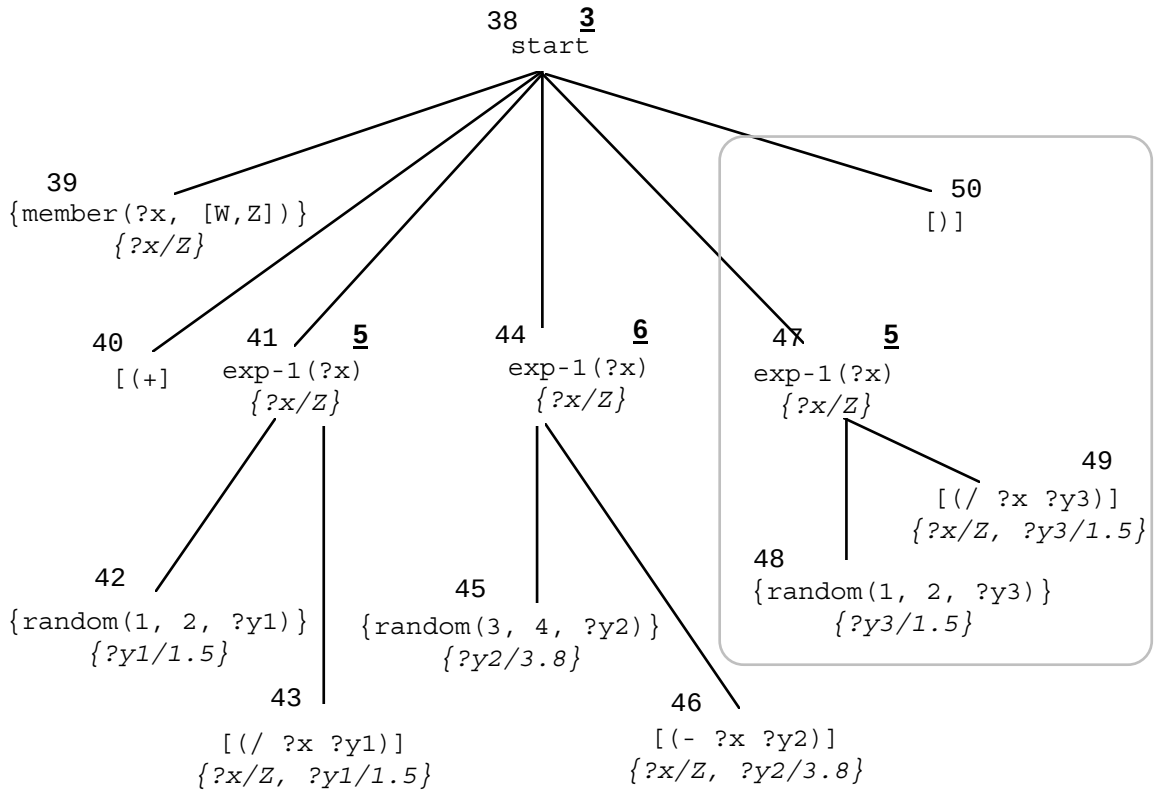

Figure 5: A derivation tree of the offspring produced by performing crossover between the primary sub-tree 3 of the tree in figure 2 and the secondary sub-tree 16 of the tree in figure 3.

2.4. Mutation of programs

The mutation operation in LOGENPRO introduces random modifications to programs in the population. A program in the population is selected as the parental program. The selection is based on various methods such as fitness proportionate and tournament selections. The following steps are used to produce an offspring program:

1. If there are sub-trees in the derivation tree of the parental program that have not been selected previously, select randomly a sub-tree from these sub-trees using a uniform distribution. The root of the selected sub-tree is called the mutation point. Otherwise, terminate the algorithm without generating any offspring.
2. Generate a new derivation tree using the deduction mechanism produced by LOGENPRO. The new derivation tree is created under the constraint that the offspring produced must be valid according to the grammar.

3. If a new derivation-tree can be found in step 2, create and return the offspring, which is obtained by deleting the selected sub-tree and then inserting the new derivation tree at the mutation point. Otherwise, go to step 1.

For example, assume that the program being mutated is `(+ (- Z 3.5) (- Z 3.8) (/ Z 1.5))` and the corresponding derivation tree is depicted in figure 2. If the sub-tree 3, MUTATED-SUB-TREE, is selected to be modified and the root of the MUTATED-SUB-TREE is designated as the MUTATE-POINT. Then a new derivation tree, NEW-SUB-TREE, for the S-expression `(/ Z 1.9)` can be obtained from the non-terminal symbol `exp-1(Z)` using the fifth rule of the grammar. The derivation tree is shown in figure 6. A new offspring is obtained by duplicating the genetic materials of its parental derivation tree, followed by deleting the MUTATED-SUB-TREE from the duplication, and then pasting the NEW-SUB-TREE at the MUTATE-POINT. The derivation tree of the offspring `(+ (/ Z 1.9) (- Z 3.8) (/ Z 1.5))` can be found in figure 7.

LOGENPRO has an efficient implementation of the mutation algorithm. Moreover, an inference engine has been developed for deducing derivation trees (or programs) from a given logic grammar. Thus, only valid mutations can be performed and this operation can be achieved effectively and efficiently.

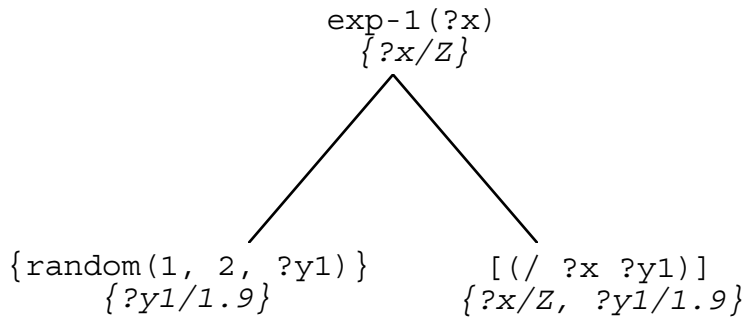

Figure 6: A derivation tree generated from the non-terminal exp-1(Z)

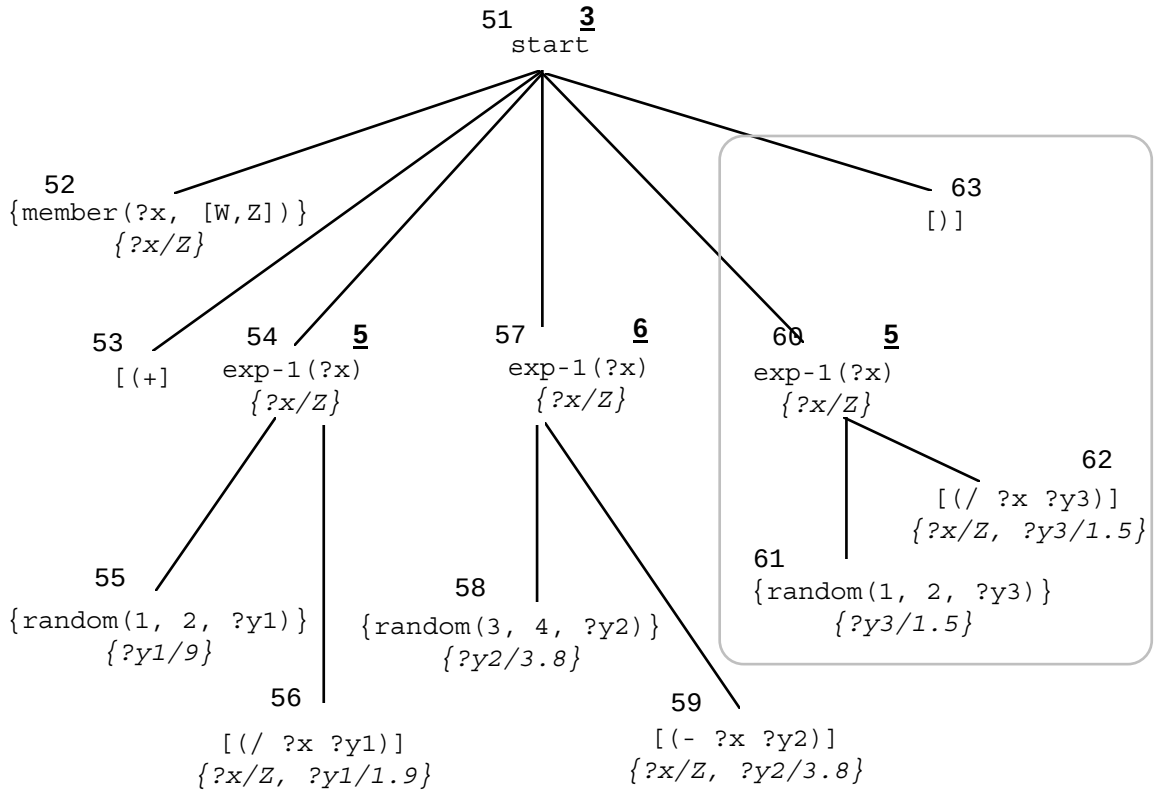

Figure 7: A derivation tree of the offspring produced by performing mutation of the tree in figure 6 at the sub-tree 2

2.5. The evolution process of LOGENPRO

The problem of inducing programs can be reformulated as a search for a highly fit program in the space of all possible programs in the language specified by the logic grammar. In LOGENPRO, populations of programs are genetically bred (Goldberg 1989) using the Darwinian principle of survival and reproduction of the fittest along with genetic operations appropriate for creating programs. LOGENPRO starts with an initial population of programs generated randomly, induced by other learning systems, or provided by the user. Logic grammars provide declarative descriptions of the valid programs that can appear in the initial population. A fitness function must be defined by the user to evaluate the fitness values of the programs. Typically, each program is run over a set of fitness cases and the fitness function estimates its fitness by performing some statistical operations (e.g. average) to the values returned by this program.

The initial programs in generation 0 are normally incorrect and have poor performance. However, some programs in the population will be fitter than others. Fitness of each program in the generation is estimated and the following process is iterated over many generations until the termination criterion is satisfied. The reproduction, sexual crossover, and asexual mutation are used

to create new generation of programs from the current one. The reproduction involves selecting a program from the current generation and allowing it to survive by copying it into the next generation. Either fitness proportionate or tournament selection can be used.

The crossover is used to create a single offspring program from two parental programs selected. Mutation creates a modified offspring program from a parental program selected. Unlike crossover, the offspring program is usually similar to the parent program. Logic grammars are used to constraint the offspring programs that can be produced by these genetic operations.

This algorithm will produce populations of programs which tend to exhibit increasing average of fitness. LOGENPRO returns the best program found in any generation of a run as the result.

3. Learning logic program from imperfect data

In this section, we describe the application of LOGENPRO to learn logic programs from noisy and imperfect training examples. Empirical comparisons of LOGENPRO with FOIL (the publicly available version of FOIL, version 6.0 , is used in this experiment) and with mFOIL (Lavrac and Dzeroski 1994) in the domain of learning illegal chess endgame positions from noisy examples are presented.

mFOIL is based on FOIL that has adapted several features from CN2 (Clark and Niblett 1989), such as the use of the Laplace and m-estimate as a search heuristics and the use of significance testing as a stopping criterion. Moreover, mFOIL uses beam search and can apply mode and type information to reduce the search space. The parameters that can be set by a user are: 1) the beam width, 2) the search heuristics, 3) the value of m if m-estimate is used as the search heuristics, and 4) the significance threshold used in the significance test. A number of different instances of mFOIL have been tested on the chess endgame problem. Their parameter values are summarized in table 2.

	beam width	heuristics	m	significance threshold
mFOIL1	5	m-estimate	0.01	0
mFOIL2	10	m-estimate	0.01	0
mFOIL3	5	m-estimate	0.01	6.35
mFOIL4	10	m-estimate	32	0

Table 2: The parameter values of different instances of mFOIL examined in this section.

In this section, LOGENPRO employs a variation of FOIL to find the initial population of logic programs. Thus, it uses the same noise-handling mechanism of FOIL. The variation is called BEAM-FOIL because it uses a beam search method rather than the greedy search strategy of FOIL.

BEAM-FOIL produces a number of different logic programs when it terminates and the best program among them is the solution of the problem. The logic programs created by BEAM-FOIL are used by LOGENPRO to initialize the first generation. In order to study the effects of the genetic operations performed by LOGENPRO on the initial programs provided by BEAM-FOIL, a comparison between them is also discussed.

The chess endgame problem is presented in the following sub-section. The experimental setup is detailed in sub-section 3.2. We compare LOGENPRO with other learning systems in the subsequent sub-sections.

3.1. The chess endgame problem

The chess endgame problem is a benchmark problem in the field of data mining for evaluating performance of data mining systems (Dzeroski and Lavrac 1993). In the problem, 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.

There are 11000 examples in the dataset (3576 positive and 7424 negative examples). Muggleton et al. (1989) used smaller datasets to evaluate the performances of CIGOL and DUCE for the chess endgame problem. There were five small sets of 100 examples each and five large sets of 1000 examples each. In other words, there were 5500 examples in total. Each of the sets was used as a training set. The induced programs obtained from a small training set was tested on the 5000 examples from the large sets, the programs obtained from each large training set was tested on the remaining 4500 examples.

3.2. The setup of experiments

In each experiment of the ten experiments performed, the training set contains 1000 examples (336 positive and 664 negative examples) and the disjoint testing set has 10000 examples (3240 positive and 6760 negative examples). These training and testing sets are selected from the dataset using different seeds for the random number generator.

Different amounts of noise are introduced into the training examples in order to study the performances of different systems in learning logic programs from noisy environment. To introduce n% of noise into argument X of the training 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 $1-\{[(1-n\%)+n\%*\frac{1}{8}]^6*(1-n\%)\}$. For each 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. Logic programs are then induced from the training set using LOGENPRO, FOIL, different instances of mFOIL, and BEAM-FOIL. Finally, the classification accuracy of the learned logic programs is estimated on the testing set. For BEAM-FOIL, the size of beam is ten and thus ten logic programs are returned. The best one among the programs returned is designated as the solution of BEAM-FOIL.

LOGENPRO uses the logic grammar in table 3 to solve the problem. In the grammar, `[adjacent(?x, ?y)]` and `[less_than(?x, ?y)]` are terminal symbols. The logic goal `member(?x, [WKf, WKr, WRf, WRr, BKf, BKr])` will instantiate logic variable `?x` of the grammar to either `WKf`, `WKr`, `WRf`, `WRr`, `BKf`, or `BKr`, the logic variables of the target logic program.

The population size for LOGENPRO is 10 and the maximum number of generations is 50. In fact, different population sizes 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 evolutionary algorithms using the proposed framework.

```
start       ->    clauses.
clauses     ->    clauses, clauses.
clauses     ->    clause.
clause      ->    consq, [:-], antes, [.].
consq       ->    [illegal(WKf, WKr, WRf, WRf, BKf, BKr)].
antes       ->    antes, [,], antes.
antes       ->    ante.
ante        ->    {member(?x,[WKf, WKr, WRf, WRf, BKf, BKr])},
                  {member(?y,[WKf, WKr, WRf, WRf, BKf, Bkr])},
                  literal(?x, ?y).
literal(?x, ?y)   ->    [?x = ?y].
literal(?x, ?y)   ->    [ ~ ?x = ?y].
literal(?x, ?y)   ->    [ adjacent(?x, ?y) ].
literal(?x, ?y)   ->    [ ~ adjacent(?x, ?y) ].
literal(?x, ?y)   ->    [ less_than(?x, ?y) ].
literal(?x, ?y)   ->    [ ~ less_than(?x, ?y) ].
```

Table 3: The logic grammar for the chess endgame problem.

For concept learning (DeJong et al. 1993, Janikow 1993, Greene and Smith 1993), each individual logic program in the population can be evaluated in terms of how well it covers positive examples and excludes negative examples. Thus, the fitness functions for concept learning problems calculate this measurement. 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 method of estimating fitness 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 value, a logic program that deduces everything are negative will have very good fitness. Thus, in this case, the fitness function should find a weighted sum of the total numbers of misclassified positive and negative examples.

In this experiment, the fitness function of LOGENPRO evaluates the number of training examples misclassified by each individual in the population. Since LOGENPRO is a probabilistic system, five runs of each experiment are performed and the average of the classification accuracy of these five runs is returned as the classification accuracy of LOGENPRO for the particular experiment. In other words, fifty runs of LOGENPRO have been performed in total. The average execution time of LOGENPRO is 1 hour 43 minutes on a Sun Sparc Workstation. The results of these systems are summarized in table 4. The performances of these systems are compared using the one-tailed paired *t*-test with 0.05% level of significance. The sizes of logic programs induced by these learning systems are summarized in table 5.

	Noise Level						
	0.00	**0.05**	**0.10**	**0.15**	**0.20**	**0.30**	**0.40**
LOGENPRO (Average)	0.996	0.983	0.960	0.938	0.855	0.733	0.670
Variance	0.00E+00	7.74E-06	2.96E-04	7.85E-04	2.57E-03	2.47E-03	1.44E-04
FOIL (Average)	0.996	0.898	0.819	0.761	0.693	0.596	0.529
Variance	0.00E+00	5.07E-04	6.56E-04	5.15E-04	5.30E-04	3.35E-04	3.11E-04
BEAM-FOIL (Average)	0.996	0.802	0.757	0.744	0.724	0.685	0.674
Variance	0.00E+00	7.07E-04	1.62E-04	1.88E-04	2.00E-04	1.40E-04	1.04E-04
mFOIL1 (Average)	0.985	0.883	0.845	0.815	0.785	0.719	0.685
Variance	0.00E+00	5.15E-05	7.29E-05	3.12E-04	2.15E-04	1.39E-04	1.30E-04
mFOIL2 (Average)	0.985	0.932	0.888	0.842	0.798	0.713	0.680
Variance	0.00E+00	7.47E-05	9.16E-05	9.26E-04	3.09E-04	1.41E-04	3.05E-04
mFOIL3 (Average)	0.896	0.836	0.805	0.771	0.723	0.677	0.676
Variance	1.97E-16	7.83E-04	1.05E-04	1.89E-04	9.81E-04	7.74E-06	0.00E+00
mFOIL4 (Average)	0.985	0.985	0.880	0.806	0.740	0.692	0.668
Variance	0.00E+00	4.05E-06	7.85E-03	5.14E-03	2.14E-03	3.72E-04	2.86E-04

Table 4: The averages and variances of accuracy of LOGENPRO, FOIL, BEAM-FOIL, and different instances of mFOIL at different noise levels.

	Noise Level						
	0.00	**0.05**	**0.10**	**0.15**	**0.20**	**0.30**	**0.40**
LOGENPRO (#clauses)	4.000	9.540	8.960	8.620	6.680	4.220	2.540
#literals/clause	1.50	2.59	2.94	3.20	3.40	4.39	4.98
FOIL (#clauses)	4.000	35.100	48.000	48.700	56.200	59.800	71.300
#literals/clause	1.50	3.65	4.44	4.73	5.06	5.23	5.40
BEAM-FOIL (#clauses)	4.000	5.000	4.400	4.200	4.000	3.500	2.800
#literals/clause	1.50	3.75	3.93	4.17	4.63	5.25	6.07
mFOIL1 (#clauses)	3.000	31.900	35.700	31.100	28.300	18.100	15.700
#literals/clause	2.00	3.07	3.20	3.18	3.42	3.34	3.57
mFOIL2 (#clauses)	3.000	48.800	50.600	48.200	44.500	41.400	34.900
#literals/clause	1.67	3.18	3.33	3.44	3.57	3.62	3.70
mFOIL3 (#clauses)	2.000	12.400	10.400	7.300	3.300	0.100	0.000
#literals/clause	1.50	2.68	3.10	3.02	3.46	4.00	0.00
mFOIL4 (#clauses)	3.000	3.000	2.400	1.800	1.200	1.200	11.200
#literals/clause	1.67	1.73	1.80	2.15	2.00	1.46	3.55

Table 5: The sizes of logic programs induced by LOGENPRO, FOIL, BEAM-FOIL, and different instances of mFOIL at different noise levels.

3.3. Comparison of LOGENPRO with FOIL

The classification accuracy of both systems degrades seriously as the noise level increases (figure 8). The classification accuracy of LOGENPRO decreases smoothly when the noise level is on or below 0.15. It reduces from 0.996 to 0.938, a 5.8% decrement. There are sudden drops of accuracy when the noise level is between 0.15 and 0.40. It falls from 0.938 to 0.670, a 28.5% reduction. The accuracy of FOIL decreases rapidly when the noise level is on or below 0.20. It drops from 0.996 to

0.693, a 30.4% reduction. The decrease slightly slows down between the noise levels of 0.20 and 0.40. It drops from 0.693 to 0.529, a 23.7% reduction.

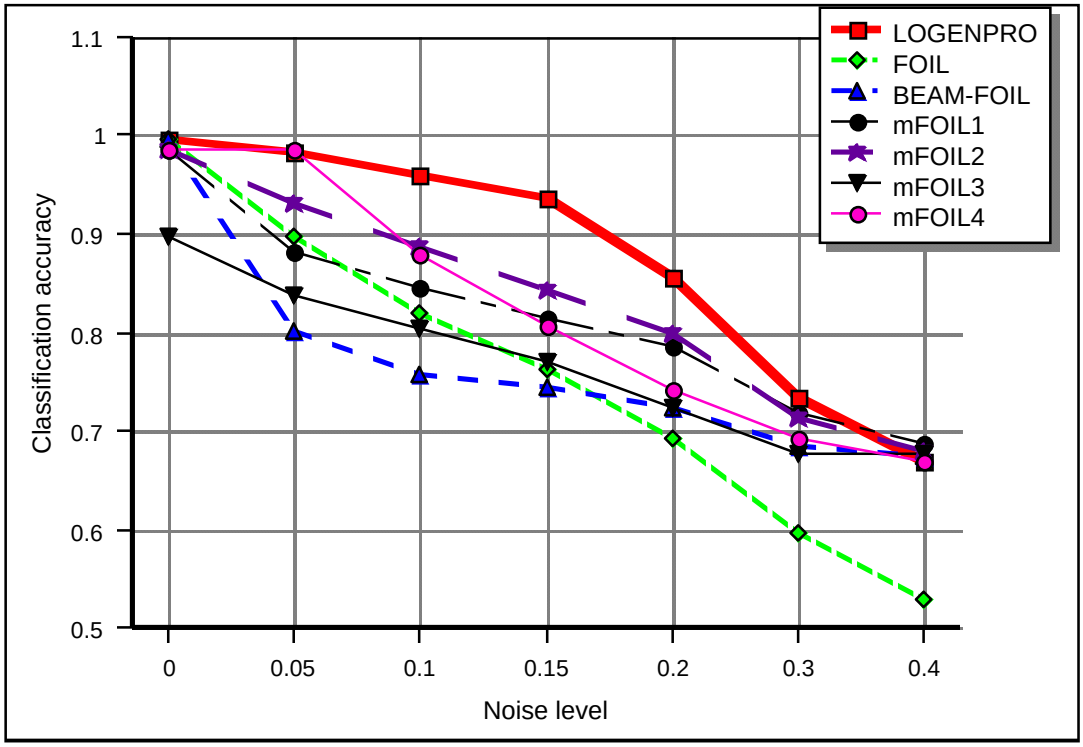

Figure 8: Comparison between LOGENPRO, FOIL, BEAM-FOIL, mFOIL1, mFOIL2, mFOIL3 and mFOIL4

The results are statistically evaluated using the one-tailed paired *t*-test. For each noise level, the classification accuracy is compared to test the null hypothesis against the alternative hypothesis. The null hypothesis states that the difference in accuracy is zero at the 0.05% level of significance. On the other hand, the alternative hypothesis declares that the difference is greater than zero at the 0.05% level of significance. The t-statistics are listed as follows:

Noise Level	0.00	0.05	0.10	0.15	0.20	0.30	0.40
t-statistics	NA	12.59	17.78	19.33	14.17	8.07	26.82

The t-statistics at the 0.00 noise level is not available because the variances are very small (near zero). The t-statistics at the 0.05 noise level is 12.59 which is greater than the critical value of 4.78. Thus, we can reject the null hypothesis and assert that the classification accuracy of LOGENPRO is higher than that of FOIL. Similarly, the classification accuracy of LOGENPRO at the noise levels between 0.05 and 0.40 is significantly higher than that of FOIL. The largest difference reaches 0.177 at the 0.15 noise level. The average number of induced clauses and the average number of literals per clause show that LOGENPRO generates compact and

comprehensive logic programs even at the high noise levels. On the other hand, the complexity of the logic programs learned by FOIL increases when the noise level increase. In other words, FOIL overfits noise in the dataset.

3.4. Comparison of LOGENPRO with BEAM-FOIL

The results of the one-tailed paired *t*-test are listed as follows:

Noise Level	0.00	0.05	0.10	0.15	0.20	0.30	0.40
t-statistics	NA	22.20	33.82	21.91	9.19	3.26	-0.81

The t-statistics at the 0.00 noise level is not available because the variances are very small (near zero). The classification accuracy of LOGENPRO at the noise levels between 0.05 and 0.20 is significantly higher than that of BEAM-FOIL. At the noise level of 0.30, the accuracy of LOGENPRO is higher than that of BEAM-FOIL, but the difference is not significant. On the other hand, the accuracy of BEAM-FOIL at the noise level of 0.40 is higher than that of LOGENPRO, but the difference is insignificant. This comparison indicates that the genetic operations of LOGENPRO can actually improve the logic programs generated by other learning systems such as BEAM-FOIL. The sizes of logic programs induced by BEAM-FOIL show that BEAM-FOIL over-generalizes at the high noise levels.

3.5. Comparison of LOGENPRO with mFOIL1

We compare LOGENPRO with mFOIL1 to mFOIL4 one by one in this and the following sub-sections. The parameters of this instance are presented in table 2. Lavrac and Dzeroski (1994) compare the performances of mFOIL1 with FOIL2.0, a version of FOIL, for the chess endgame problem using the smaller dataset described in sub-section 2.1. They find that mFOIL1 outperforms FOIL2.0 at all noise levels. Our results depicted in figure 8 are inconsistent with those obtained by Lavrac and Dzeroski. We find that FOIL outperforms mFOIL1 at the noise levels of 0.0 and 0.05. On the other hand, mFOIL1 has better performance when the noise level is on or over 0.1. The inconsistency may be explained because we employ an improved version of FOIL, FOIL6.0, and larger sets of training and testing examples. The results of the one-tailed paired *t*-test between LOGENPRO and mFOIL1 are listed as follows:

Noise Level	0.00	0.05	0.10	0.15	0.20	0.30	0.40
t-statistics	3.03E+08	35.38	17.29	14.98	5.15	1.11	-3.37

The classification accuracy of LOGENPRO at the noise levels between 0.0 and 0.20 is significantly higher than that of mFOIL1. At the noise level of 0.30, the accuracy of LOGENPRO

is higher than that of mFOIL1 by about 0.014, but the difference is not significant. On the other hand, the accuracy of mFOIL1 at the noise level of 0.40 is higher than that of LOGENPRO, the difference is insignificant.

3.6. Comparison of LOGENPRO with mFOIL2

The results of the one-tailed paired *t*-test between LOGENPRO and mFOIL2 are listed as follows:

Noise Level	0.00	0.05	0.10	0.15	0.20	0.30	0.40
t-statistics	3.03E+08	21.59	13.05	9.95	4.37	1.23	-1.65

The classification accuracy of LOGENPRO at the noise levels between 0.0 and 0.15 is significantly higher than that of mFOIL2. At the noise levels of 0.20 and 0.30, the accuracy of LOGENPRO is higher than that of mFOIL2, but the differences are not significant. On the other hand, the accuracy of mFOIL2 at the noise level of 0.40 is higher than that of LOGENPRO, but the difference is insignificant.

3.7. Comparison of LOGENPRO with mFOIL3

The accuracy of mFOIL3 at the noise levels of 0.00, 0.30, and 0.40 is not acceptable. By comparing mFOIL3 with mFOIL1 (figure 8), we can conclude that the significance threshold for noise-handling affects the performance of mFOIL severely (see table 2). The results of the one-tailed paired *t*-test between LOGENPRO and mFOIL3 are listed as follows:

Noise Level	0.00	0.05	0.10	0.15	0.20	0.30	0.40
t-statistics	NA	16.99	22.29	16.44	8.12	3.65	-1.66

The t-statistics at the 0.00 noise level is not available because the variances are very small (near zero). The classification accuracy of LOGENPRO at the noise levels between 0.05 and 0.40 is significantly higher than that of mFOIL3.

3.8. Comparison of LOGENPRO with mFOIL4

The results of the one-tailed paired *t*-test between LOGENPRO and mFOIL4 are listed as follows:

Noise Level	0.00	0.05	0.10	0.15	0.20	0.30	0.40
t-statistics	2.22E+08	-1.45	2.77	6.37	8.00	2.20	0.24

The classification accuracy of LOGENPRO at the noise levels 0.00, 0.15 and 0.20 is significantly higher than that of mFOIL4. The sizes of the logic programs learned by mFOIL4

illustrate that mFOIL4 over-generalizes at the noise levels between 0.10 and 0.30. On the other hand, mFOIL4 overfits the noise in the dataset at the 0.40 noise level.

3.9. Discussion

In this section, we employ LOGENPRO to combine evolutionary algorithms and BEAM-FOIL, to learn logic programs. The performance of LOGENPRO in a noisy domain has been evaluated by using the chess endgame problem. Detailed comparisons between LOGENPRO and other ILP systems have been conducted. It is found that LOGENPRO outperforms these ILP systems significantly at most noise levels. These results are surprising because the LOGENPRO uses the same noise-handling mechanism of FOIL by initializing the population with programs created by BEAM-FOIL.

One possible explanation of the better performance of LOGENPRO 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 finds interesting and useful patterns from these noisy examples.

4. Learning rules from the fracture database

The data mining system has been applied to a real-life medical database consisting of children with limb fractures, admitted to the Prince of Wales Hospital of Hong Kong in the period 1984-1996. This data can provide information for the analysis of children fracture patterns. This database has 6500 records and 8 attributes. The attributes are listed in table 6.

From the database, we expect to learn knowledge about these attributes. The medical expert provides extra knowledge on how the rules should be formulated. He suggests that the attributes can be divided into three time stages: a diagnosis is first given to the patient, then an operation is performed, and after that the patient stays in the hospital. This knowledge leads to three kinds of rules. Firstly, sex, age and admission date are the possible causes of diagnosis. Secondly, these three attributes and diagnosis are the possible causes of operation and surgeon. Thirdly, length of stay has all other attributes as the possible causes. A grammar (see Appendix A) is written to specify these three kinds of rules. In this experiment, we have used a population size of 300 to run for 50 generations.

Name	Type	Description	Possible Value
Sex	Nominal	Sex	'M' or 'F'
Age	Numeric	Age	Between 0 to 16 years old
Admday	Date	Admission date	Between year 1984 to 1996; Divided into four parts:Day, Month, Year and Weekday
Stay	Numeric	Length of staying in hospital	Between 0 to 1000 days Discretized into 18 non-uniform ranges.
Diagnosis	Nominal	Diagnosis of fracture	10 different values, based on the location of fracture
Operation	Nominal	Operation	'CR' (Simple Closed Reduction), 'CR+K-wire' (Closed Reduction with K-wire), 'CR+POP' (Closed Reduction with POP), 'OR' (Open Reduction), or Null (no operation)
Surgeon	Nominal	Surgeon	One of 61 surgeons or Null if no operation
Side	Nominal	Side of fracture	'Left', 'Right', 'Both' or 'Missing'

Table 6: Attributes in the fracture database

One of the two interesting rules discovered about diagnosis is given below:

```
If age is between 2 and 5, then diagnosis is Humerus.
```

LOGENPRO finds that humerus fracture is the most common fracture for children between 2 and 5 years old, while radius fracture is the most common fracture for boys between 11 and 13.

Eight interesting rules about operation are found. One of them is presented as follows:

```
If age is between 0 and 7 and admission year is between 1988 and 1993
and diagnosis is Radius, then operation is CR+POP.
```

These rules suggest that radius and ulna fractures are usually treated with CR+POP (i.e. plaster). Operation is usually not needed for tibia fracture. Open reductions are more common for elder children with age larger than 11, while young children with age lower than 7 have a higher chance of not needing operations. We do not find any interesting rules about surgeons, as the surgeons for operation are more or less randomly distributed in the database.

Seven interesting rules about length of stay are found. One of them is:

```
If admission year is between 1985 and 1996
and diagnosis is Femur , then stay is more than 8 days.
```

The rules about the length of stay suggest that Femur and Tibia fractures are serious injuries and have to stay longer in hospital. If open reduction is used, the patient requires longer time to recover because the wound has been cut open for operation. If no operation is needed, it is likely that the patient can return home within one day. Relatively, radius fracture requires a shorter time for recovery.

The results have been evaluated by the medical expert. The rules provide interesting patterns that were not recognized before. The analysis gives an overview of the important epidemiological and demographic data of the fractures in children. It clearly demonstrated the treatment pattern and rules of decision making. It can provide a good monitor of the change of

pattern of management and the epidemiology if the data mining process is continued longitudinally over the years. It also helps to provide the information for setting up a knowledge-based instruction system to help young doctors in training to learn the rules in diagnosis and treatment.

5. Conclusion

We have proposed a framework for combining Genetic Programming and Inductive Logic Programming. This framework is based on a formalism of logic grammars. To implement the framework, a system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) has been developed. The formalism can represent context-sensitive information and domain-dependent knowledge. The formalism is also very flexible and the knowledge learned can be represented in various knowledge representations such as logic programs and production rules. LOGENPRO has been tested on some learning tasks. These experiments and the results demonstrate that LOGENPRO is a promising system for inducing knowledge from databases.

Since the framework is very flexible, different representations employed by other learning systems can be specified easily. It facilitates the integration of LOGENPRO with the latter. One approach is to incorporate the search operators of other systems into LOGENPRO. These operators include information guided hill-climbing (Quinlan 1990; 1991), explanation-based generalization (DeJong and Mooney 1986, Mitchell et. al. 1986, Ellman 1989), explanation-based specialization (Minton 1989) and inverse resolution (Muggleton 1992). LOGENPRO can also invoke these systems as front-ends to generates the initial population. The advantage is that we can quickly find important and meaningful components (genetic materials) and embody these components into the initial population. Moreover, it has been found that LOGENPRO, when combined with other learning systems, has superior performance in learning logic programs from imperfect data as 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 LOGENPRO because it uses genetic algorithms as its primary learning mechanism.

We have described how to combine LOGENPRO and a variation of FOIL, BEAM-FOIL, in learning logic programs. The initial population of logic programs is provided by BEAM-FOIL**.** The performance of LOGENPRO in inducing logic programs from imperfect training examples is evaluated using the chess endgame problem. A detailed comparison to FOIL, BEAM-FOIL, and mFOIL has been conducted. It is found that LOGENPRO outperforms the other systems significantly in this domain.

The system has been applied to one real-life medical database. The results can provide interesting knowledge as well as suggest refinements to the existing knowledge. The system automatically uncovered knowledge about the age effect on fracture, the relationship between diagnoses and operations, and the effect of diagnoses and operations on lengths of staying in the hospital.

In conclusion, LOGENPRO, which is a grammar driven genetic based system, has been demonstrated to be a promising tool for knowledge discovery in a noisy environment.

Acknowledgments

The research is sponsored by the RGC Earmarked research grant of UGC reference number CUHK 486/95E.

Reference

Abramson, H. and Dahl, V. (1989). *Logic Grammars*. Berlin: Springer-Verlag.

Breimen, L., Friedman, J. H., Olshen, R. A. and Stone, C. J. (1984). *Classification and Regression Trees*. Belmont: Wadsworth.

Clark, P. and Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*; **3**, 261-283.

Davis, L. (1987). Genetic Algorithms and Simulated Annealing. London: Pitman.

Davis, L. (1991). *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.

DeJong, G. F. and Mooney, R. (1986). Explanation-Based Learning: An Alternative View. *Machine Learning*, **1**, 145-176.

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

Dzeroski, S. (1996). Inductive Logic Programming and Knowledge Discovery in Databases. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (eds.), *Advances in Knowledge Discovery in Data Mining*, pp. 117-152. Menlo Park, CA: AAAI Press.

Dzeroski, S. and Lavrac, N. (1993). Inductive Learning in Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering*, **5,** 939-949.

Ellman, T. (1989). Explanation-Based Learning: A Survey of Programs and Perspectives, *ACM Computing Surveys*, **21**, 163-222.

Fayyad, U. M., Piatetsky-Shapiro, G., and Smyth, P. (1996). From Data Mining to Knowledge Discovery: An Overview. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (eds.), *Advances in Knowledge Discovery in Data Mining*, pp. 1-34. Menlo Park, CA: AAAI Press.

Frawley, W., Piatetsky-Shapiro, G., and Matheus, C. (1991). Knowledge Discovery in Databases: an Overview. In G. Piatetsky-Shapiro and W. Frawley (eds.), *Knowledge Discovery in Databases*, pp. 1-27. Menlo Park, CA: AAAI Press.

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

Greene, D. P. and Smith, S. F. (1993). Competition-Based Induction of Decision Models from Examples, *Machine Learning*, **13**, 229-257.

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

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation*. MA: Addison-Wesley.

Janikow, C. Z. (1993). A Knowledge-Intensive Genetic Algorithm for Supervised Learning, *Machine Learning*, **13**, 189-228.

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.

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

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

Lewis, H. R. and Rapadimitrion, C. H. (1981). *Elements of the theory of computation*. NJ: Prentice Hall.

Michalski, R. S., Mozetic, I., Hong, J. and Lavrac, N. (1986). The multi-purpose incremental learning system AQ15 and its testing application on tree medical domains. In *Proceedings of the National Conference on Artificial Intelligence*, 1041-1045. San Mateo, CA: Morgan Kaufmann.

Minton, S (1989). Learning Search Control Knowledge: An Explanation-Based Approach. Boston: Kluwer Academic.

Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T. (1986). Explanation-Based Generalization: A Unifying View, *Machine Learning*, **1**, 47-80.

Muggletion, S., editor (1992). *Inductive Logic Programming*. London: Academic Press.

Muggleton, S., Bain, M., Hayes-Michie, J., and Michie, D. (1989). An experimental comparison of human and machine learning formalisms. In *Proceedings of the Sixth International Machine Learning Workshop*, 113-188. CA: Morgan Kaufmann

Pereira, F. C. N. and Warren, D. H. D. (1980). Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks *Artificial Intelligence*, **13**, 231-278.

Pereira, F. C. N. and Shieber, S. M. (1987). *Prolog and Natural-Language Analysis*. CA: CSLI.

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**, 239-266.

Quinlan, J. R. (1991). Determinate Literals in Inductive Logic Programming, In *Proceedings of the Eighth International Workshop on Machine Learning*, 442-446. CA: Morgan Kaufmann.

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. MA: MIT Press.

Wong, M. L. and Leung, K. S. (1995a). An Induction System that Learns Programs in different Programming Languages using Genetic Programming and Logic Grammars. In *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence*. pp. 380-387. CA: IEEE Computer Society Press.

Appendix A: The logic grammar for the fracture database

This grammar is not completely listed. The grammar rules for the other attribute descriptors are similar to the grammar rules 14 - 25

```
1:    start       ->    rule1.
2:    start       ->    rule2.
3:    start       ->    rule3.
4:    rule1       ->    [if], antes1, [, then], consq1, [.].
5:    rule2       ->    [if], antes1, [and], antes2, [, then], consq2 [.].
6:    rule3       ->    [if], antes1, [and], antes2, [and], antes3,
                        [, then], consq3, [.].
7:    antes1      ->    sex1, [and], age1, [and], admday1.
8:    antes2      ->    diagnosis1.
9:    antes3      ->    operation1, [and], surgeon1.
10:   consq1      ->    diagnosis_descriptor.
11:   consq2      ->    operation_descriptor.
12:   consq2      ->    surgeon_descriptor.
13:   consq3      ->    stay_descriptor.
14:   sex1        ->    [any].
15:   sex1        ->    sex_descriptor.
16:   sex_descriptor    ->    {sex_const(?x)}, [sex = ?x].
17:   admday1           ->    [any].
18:   admday1           ->    admday_descriptor.
19:   admday_descriptor ->    {day_const(?x)}, {day_const(?y)},
                               [admission day between ?x and ?y].
20:   admday_descriptor ->    {month_const(?x)}, {month_const(?y)},
                               [admission month between ?x and ?y].
21:   admday_descriptor ->    {year_const(?x)}, {year_const(?y)},
                               [admission year between ?x and ?y].
22:   admday_descriptor ->    {weekday_const(?x)}, {weekday_const(?y)},
                               [admission weekday between ?x and ?y].
23:   diagnosis1        ->    [any].
24:   diagnosis1        ->    diagnosis_descriptor.
25:   diagnosis_descriptor    ->    {disgnosis_const(?x)},
                                     [diagnosis is ?x].
      …
      …
```
