A Flexible Knowledge Discovery System Using Genetic Programming and Logic Grammars

Man Leung Wong

Department of Computing and Decision Sciences

Lingnan University

Tuen Mun

Hong Kong

mlwong@ln.edu.hk

A Flexible Knowledge Discovery System using Genetic Programming and Logic Grammars

Abstract

As the computing world moves from the information age into the knowledge-base age, it is beneficial to induce knowledge from the information superhighway formed from the Internet and intranet. Knowledge discovery in databases is defined as the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data stored in databases. The knowledge acquired can be expressed in different knowledge representations such as computer programs, first-order logical relations, or Fuzzy Petri Nets (FPNs). In this paper, we present a flexible knowledge discovery system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) that applies genetic programming and logic grammars to learn knowledge in various knowledge representation formalisms. The system is also powerful enough to represent context-sensitive information and domain-dependent knowledge. An experiment is performed to demonstrate that LOGENPRO can discover knowledge represented in FPNs that support fuzzy and approximate reasoning. To evaluate the performance of LOGENPRO in producing good FPNs, the classification accuracy of the fuzzy Petri net induced by LOGENPRO and that of the decision tree generated by C4.5 are compared. Moreover, the performance of LOGENPRO in inducing logic programs from noisy examples is evaluated. A detailed comparison to FOIL, a system that induces logic programs, has been conducted. These experiments demonstrate that LOGENPRO is a promising alternative to other knowledge discovery systems and sometimes is superior for handling noisy and inexact data.

Area: Knowledge Discovery in Databases, Genetic Programming, Logic Grammars, Fuzzy Petri Nets

1. Introduction

As the computing world moves from the information age into the knowledge-base age, it is beneficial to induce knowledge from the information superhighway formed from the Internet and intranet. Knowledge discovery from various data sources of the new information infrastructure is concerned with the non-trivial extraction of implicit, previously unknown, and potentially useful information from them (Fayyad et al. 1996, Frawley et al. 1991, Piatetsky-Shapiro and Frawley 1991). The knowledge acquired can be expressed in different knowledge representation formalisms such as computer programs, first-order logical relations, decision trees, decision lists, production rules, or Petri nets.

Dzeroski and Lavrac have showed that Inductive Logic Programming (ILP) can be used to induce knowledge represented as first-order logical relations (Dzeroski and Lavrac 1993, Dzeroski 1996). Research in ILP can be classified into empirical and interactive ILP (Lavrac and Dzeroski 1994, De Raedt 1992) and several empirical ILP systems, such as CIGOL (Muggleton and Buntine 1988), GOLEM (Muggleton and Feng 1990), FOIL (Quinlan 1990; 1991), and FOCL (Pazzani and Kibler 1992, Pazzani et al. 1991), have been developed recently. For example, FOIL can efficiently learn function-free Horn clauses. 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 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.

Another approach of knowledge discovery is Genetic Programming (GP) that extends traditional Genetic Algorithms (Holland 1975, Goldberg 1989, Davis 1987; 1991) to induce automatically S-expressions in Lisp (Koza 1992; 1994, Koza et al. 1999, Kinnear 1994). Koza (1992) demonstrated that decision trees can be represented as S-expressions in Lisp. In other words, GP can be used to discover knowledge from databases.

The problem of discovering knowledge from databases can be reformulated as a search for a highly fit program in the space of all possible programs (Mitchell 1982). The search space of ILP is determined by the syntax of logic program and the background knowledge. In GP, this space is determined by the syntax of S-expression in Lisp and the sets of terminals and functions. Thus, the search space is fixed once these elements are decided.

In this paper, we present a flexible knowledge discovery system (LOGENPRO), a generalization and extension of GLPS (Wong and Leung 1995), that combine ILP and GP to induce knowledge from databases. LOGENPRO (The LOgic grammar based GENetic PROgramming system) is based on a formalism of logic grammars and it can specify the search space declaratively. The system is powerful enough to represent context-sensitive information and domain-dependent knowledge used to accelerate the learning of knowledge. LOGENPRO is also very flexible and the knowledge acquired can be represented in different knowledge representations such as computer programs, fuzzy Petri nets, first-order logical relations, and/or fuzzy relations (Wong 1998, Wong and Leung 2000).

The next section presents the formalism of logic grammars and the details of LOGENPRO. Petri nets, a directed bipartite graph with a high degree of structural parallelism and pipelining, is an ideal knowledge representation (Peterson 1981). However, Petri nets cannot represent imprecise (fuzzy), incomplete, and uncertain information. Consequently, Chen et al. (1990) proposed the Fuzzy Petri Nets (FPNs) formalism that provides an effective and efficient reasoning procedure to deduce information from inexact knowledge. In section 3, we apply LOGENPRO to induce Fuzzy Petri Nets (FPNs) from information stored in databases. Knowledge discovery systems should be able to induce knowledge from noisy examples. In section 4, we employ LOGENPRO to learn knowledge represented as logic programs from noisy datasets. Finally, the conclusion is presented in the last section.

2. The logic grammars based genetic programming system (LOGENPRO)

The LOgic grammars based GENetic PROgramming system (LOGENPRO) can induce programs in various programming languages such as LISP, C and Prolog. Thus, LOGENPRO must be able to accept grammars of different languages and produce programs in these languages. Most modern programming languages are specified in the notation of BNF (Backus-Naur form) which is a kind of context-free grammar (CFG). However, LOGENPRO is based on logic grammars because CFG is not expressive enough to represent context-sensitive information of the language and domain-dependent knowledge of the target program being induced. Logic grammars are generalizations of CFG. Their expressiveness is much more powerful than that of CFG, but equally amenable to efficient execution. In this paper, logic grammars will be described in a notation similar to that of definite clause grammars (Pereira and Warren 1980). The logic grammar 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 "meaning" of the program.

The terminal symbols, which are 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 symbol. Commas denote concatenation and each grammar rule ends with a full stop.

```
1:start->         [(*], exp(X), exp(X), [)].
2:start->         {member(?x,[X, Y])}, [(*], exp-1(?x),
                  exp-1(?x), [)].
3:start->         {member(?x,[X, Y])}, [(/], exp-1(?x),
                  exp-1(?x), [)].
4:exp(?x)->       [(+ ?x 0)].
5:exp-1(?x)->     {random(0,1,?y)}, [(+ ?x ?y)].
6:exp-1(?x)->     {random(0,1,?y)}, [(- ?x ?y)].
7:exp-1(?x)->     [(+ (- X 11) 12)].
```

Table 1: A logic grammar

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, [X, Y])` in Table 1 instantiates the variable ?x to either X or Y if ?x has not already been instantiated, otherwise it checks whether the value of ?x is either X or Y. If the variable ?y has not been bound, the goal `random(0, 1, ?y)` instantiates ?y to a random floating point number between 0 and 1. Otherwise, the goal checks whether the value of ?y is between 0 and 1. The special non-terminal `start` corresponds to a program of the language.

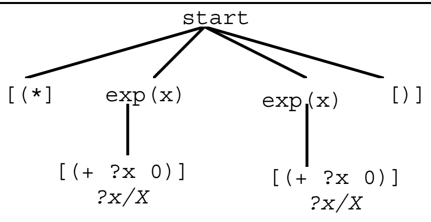
The problem of inducing programs can be reformulated as a search for a highly fit program in the space of all possible programs in the language specified by the logic grammar. In LOGENPRO, populations of programs are genetically bred (Holland 1975, Goldberg 1989, Davis 1987; 1991) using the Darwinian principle of survival and reproduction of the fittest along with genetic operations appropriate for processing 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 high-level algorithm of LOGENPRO is presented in Table 2.

1. Generate an initial population of programs.
2. Execute each program in the current population and assign it a fitness value according to the fitness function
3. If the termination criterion is satisfied, terminate the algorithm. The best program found in the run of the algorithm is designated as the result.
4. Create a new population of programs from the current population by applying the reproduction, crossover, and mutation operations. These operations are applied to programs selected by fitness proportionate or tournament selections.
5. Rename the new population to the current population.
6. Proceed to the next generation by branching back to the step 2.

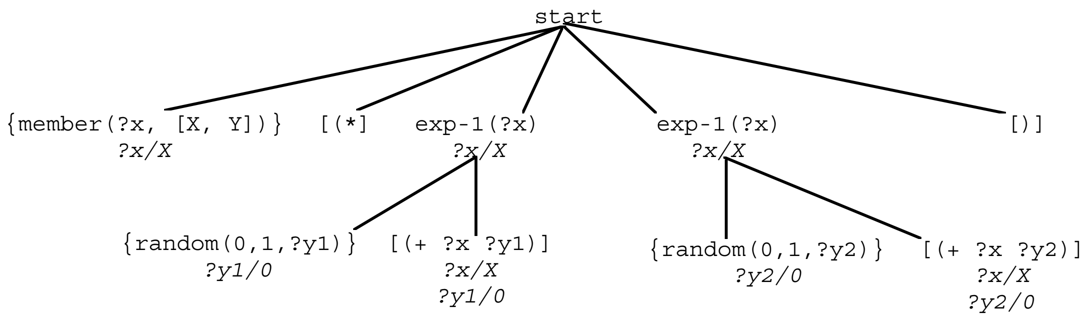
Table 2: A High Level Algorithm of LOGENPRO

The initial programs in generation 0 are normally incorrect and have poor performances. The Darwinian principle of reproduction and survival of the fittest and the genetic operation of sexual crossover 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 one offspring program from the parental programs selected. Logic grammars are used to constraint the offspring programs that can be produced by these genetic operations. 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. 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.

One of the contributions of LOGENPRO is that it represents programs in different programming languages appropriately so that the initial population can be generated easily, and that 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 `(* (+ X 0) (+ X 0))` can be generated by LOGENPRO given the logic grammar in Table 1. Its derivation tree is depicted in Figure 1(a). Alternatively, initial programs can be induced by other learning systems such as FOIL (Quinlan 1990) or given by the user. LOGENPRO analyzes each program and creates the corresponding derivation tree. If the language is ambiguous, multiple derivation trees can be generated. LOGENPRO produces only one tree randomly. For example, the program `(* (+ X 0) (+ X 0))` has multiple derivation trees, two of them are shown in Figures 1(a) and 1(b).



(a)



(b)

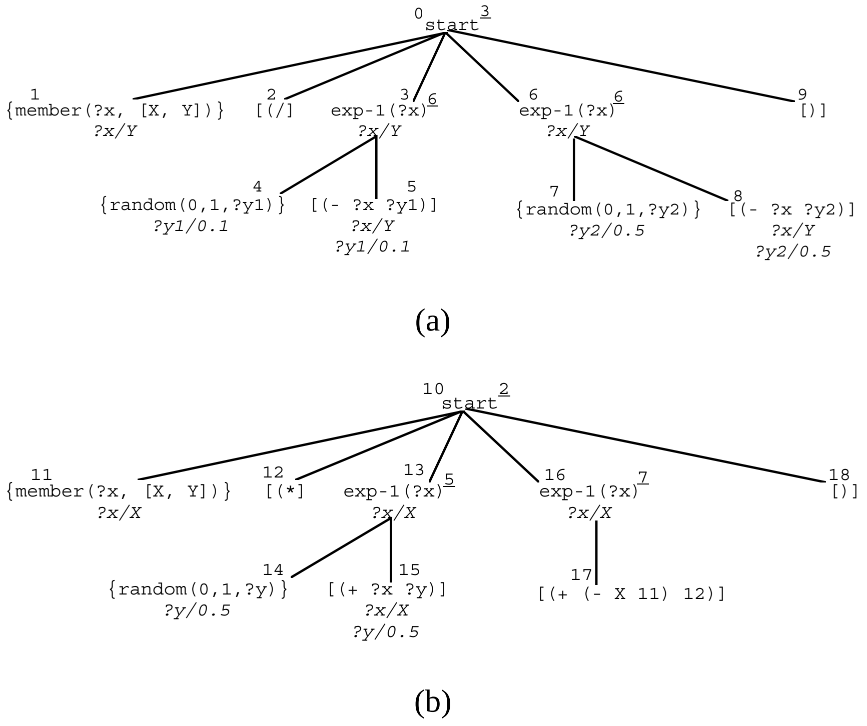
Figure 1: Derivation trees of a program

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. The following algorithm is 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 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 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, complete the crossover algorithm and return the offspring, which is obtained by deleting the selected sub-tree of the primary tree and then

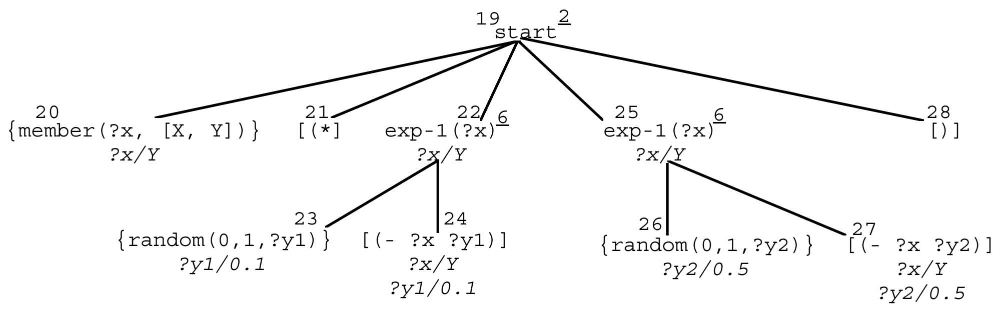
impregnating the selected sub-tree from the secondary tree at the primary crossover point. Otherwise, go to step 1.

Consider two parental programs generated by the grammar in Table 1, the primary program is `(/ (- Y 0.1) (- Y 0.5))` and the secondary program is `(* (+ X 0.5) (+ (- X 11) 12))`. The corresponding derivation trees are depicted in Figures 2(a) and 2(b) respectively. In the Figures, the shadowed numbers identify sub-trees of these derivation trees, while the underlined numbers indicate the grammar rules used in parsing the corresponding sub-trees. For example, if the primary and secondary sub-trees are respectively 2 and 12. The valid offspring `(* (- Y 0.1) (- Y 0.5))` is obtained and its derivation tree is shown in Figure 3(a). It is interesting to find that the sub-tree 19 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 `[(*]`.



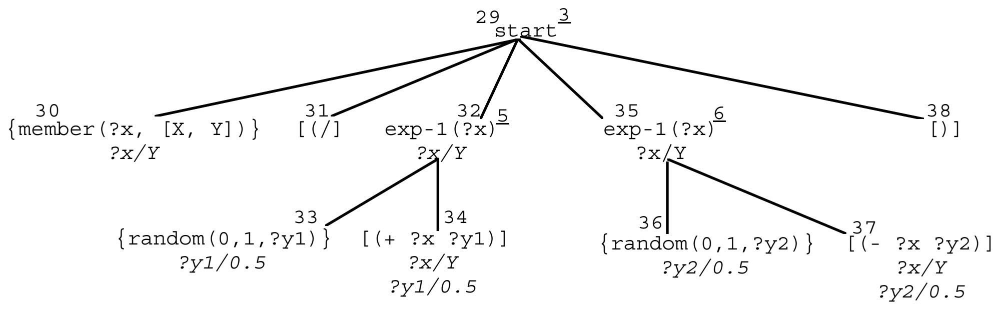
0 start 3
1 {member(?x, [X, Y])} ?x/Y
2 [(/]
3 exp-1(?x) 6 ?x/Y
6 exp-1(?x) 6 ?x/Y
9 [)]
4 {random(0,1,?y1)} ?y1/0.1
5 [(- ?x ?y1)] ?x/Y ?y1/0.1
7 {random(0,1,?y2)} ?y2/0.5
8 [(- ?x ?y2)] ?x/Y ?y2/0.5
(a)
10 start 2
11 {member(?x, [X, Y])} ?x/X
12 [(*]
13 exp-1(?x) 5 ?x/X
16 exp-1(?x) 7 ?x/X
18 [)]
14 {random(0,1,?y)} ?y/0.5
15 [(+ ?x ?y)] ?x/X ?y/0.5
17 [(+ (- X 11) 12)]
(b)

Figure 2: Derivation trees of the parental programs



19 start 2
20 {member(?x, [X, Y])} ?x/Y
21 [(*]
22 exp-1(?x) 6 ?x/Y
25 exp-1(?x) 6 ?x/Y
28 [)]
23 {random(0,1,?y1)} ?y1/0.1
24 [(- ?x ?y1)] ?x/Y ?y1/0.1
26 {random(0,1,?y2)} ?y2/0.5
27 [(- ?x ?y2)] ?x/Y ?y2/0.5

(a)



29 start 3
30 {member(?x, [X, Y])} ?x/Y
31 [(/]
32 exp-1(?x) 5 ?x/Y
35 exp-1(?x) 6 ?x/Y
38 [)]
33 {random(0,1,?y1)} ?y1/0.5
34 [(+ ?x ?y1)] ?x/Y ?y1/0.5
36 {random(0,1,?y2)} ?y2/0.5
37 [(- ?x ?y2)] ?x/Y ?y2/0.5

(b)

Figure 3: Derivation trees of offspring programs produced by crossover

In another example, the primary and secondary sub-trees are 3 and 13 respectively. The valid offspring `(/ (+ Y 0.5) (- Y 0.5))` is produced and the derivation tree is shown in Figure 3(b). It should be emphasized that the constituent from the secondary parent is changed from `(+ X 0.5)` to `(+ Y 0.5)` in the offspring. This must be modified because the logic variable ?x in sub-tree 32 is instantiated to Y in sub-tree 30. 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 16. The sub-tree 16 requires the variable ?x to be instantiated to X, But, ?x must be instantiated to Y in the context of the primary parent. Since X and Y 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 programs are produced and this operation can be achieved effectively and efficiently.

3. Learning Fuzzy Petri Nets

Knowledge-Based Systems (KBS) are definitely one of the major successes of Artificial Intelligence. They demonstrate the value of Artificial Intelligence in terms of practical applications. This statement is proven by many successful implementations of systems in various domains such as MYCIN, PROSPECTOR, XCON, and DENDRAL (Buchanan and Shortliffe 1984, Hayes-Roth et al. 1983).

Knowledge-based systems are defined as intelligent computer applications that solve complicated problems that would normally require extensive human expertise. They simulate the human reasoning process by applying domain knowledge and inference (Stefik 1995). Since they exhibit performance comparable to that of human experts in specific domains, they are employed to perform a variety of extremely complicated tasks that in the past could only be performed by highly trained human experts. Consequently, they have high potential for enormous development.

However, the knowledge acquisition bottleneck (Feigenbaum 1981) greatly obstructs the development of knowledge-based systems because it is time-consuming, error-prone, and expensive to elicit expertise from human experts. Automatic Knowledge Acquisition (AKA) employs various learning techniques to extract knowledge from various data sources such as reference books, documents, and databases (Leung and Wong 1991a; 1991b). It has been demonstrated that automatic knowledge acquisition can significantly reduce the time and cost used in developing knowledge-based systems (Buchanan and Wilkins 1993).

An ideal automatic knowledge acquisition system should be able to extract approximate knowledge from incomplete, incorrect, inconsistent, and inexact information. There are many researches on developing knowledge-based systems (Leung and Lam 1988) that can handle imprecise, incomplete, and uncertain information effectively. Imprecision (fuzziness) occurs when the boundary of a piece of information is not clear cut, e.g. John is quite tall and Mary is very beautiful. Incomplete information means some data normally required are missing. Uncertainty exists if one is not absolutely certain about a piece of information. Fuzzy logic and approximate reasoning (Zimmerman 1986) provide a theoretical foundation to model imprecision and uncertainty in knowledge-based systems. Thus, a good automatic knowledge acquisition system should be able to induce approximate knowledge such as fuzzy production rules from various data sources.

In this section, we present an approach of inducing Fuzzy Petri Nets (FPNs) which represent the fuzzy production rules of knowledge-based systems. We employ LOGENPRO to evolve cellular encodings representing various FPNs (Gruau 1994). The formalism of FPNs is presented in sub-section 3.1. Cellular encodings for FPNs are discussed in the next sub-section. To evaluate the performance of LOGENPRO in producing good FPNs, we compare the classification accuracy of the FPN generated by LOGENPRO and the accuracy of the decision tree induced by C4.5 in sub-section 3.3.

3.1. Fuzzy Petri Nets

A fuzzy knowledge base contains a number of fuzzy production rules that describe knowledge of a specific domain. A fuzzy production rule is a rule that describes the fuzzy relation between two or more propositions. There is a certainty factor (CF) attached to each rule for describing the degree of confidence in the rule. If the value of the certainty factor is larger, there is more confidence that the rule is correct. Some examples of fuzzy production rules are given as follows:

IF Height is tall THEN Weight is heavy WITH CF= 0.7

IF Sex is Male and Age is 20 to 30 and Height is tall
THEN Weight is heave WITH CF = 0.95

Four different types of fuzzy production rules have been identified by Chen et at. (1990):

- Type 1: IF D_{j1} and … and D_{jn} THEN D_k WITH CF = μ_j
- Type 2: IF D_j THEN D_{k1} and … and D_{kn} WITH CF = μ_j
- Type 3: IF D_{j1} or … or D_{jn} THEN D_k WITH CF = μ_j
- Type 4: IF D_j THEN D_{k1} or … or D_{kn} WITH CF = μ_j

where

D_j, D_k,D_{j1}, …, D_{jn}, D_{k1}, …, and D_{kn} are propositions which may contain some fuzzy concepts such as “Height is tall”, “Weight is heavy”, “Temperature is high”, etc. The truth value of each proposition is a real number between zero and one describing the degree of belief of the proposition. μ_j is a real value between zero and one. It represents the certainty factor of the rule.

Since rules of type 4 are not suitable for deduction, they are not allowed to appear in a fuzzy knowledge base.

A Fuzzy Petri Net (FPN) can be used to represent fuzzy production rules of a knowledge base. A FPN contains a finite set of places (**P**) and a finite set of transitions (**T**), where circles represent places, and bars represent transitions. A FPN is a finite bipartite graph where places are linked with transitions which in turn are connected to the output places. Similarly, there are input and output sets of transitions defined for a given place. A source transition is a transition that has no input places. A transition that has no output places is called a sink transition. Each transition is associated with a certainty factor taking values between zero and one. Each place may or may not contain a token associated with a truth value of a proposition. The relationships from places to transitions and from transitions to places are modeled by directed edges. A FPN can be formally defined as an 8-tuple (Chen et al. 1990):

$$\text{FPN} = (\mathbf{P}, \mathbf{T}, \mathbf{D}, I, O, cf, \alpha, \beta)$$

where

- $\mathbf{P} = \{P_1, P_2, \ldots, P_n\}$ is a finite set of places,
- $\mathbf{T} = \{T_1, T_2, \ldots, T_{nm}\}$ is a finite set of transitions,
- $\mathbf{D} = \{D_1, D_2, \ldots, D_n\}$ is a finite set of propositions,
- $\mathbf{P} \cap \mathbf{T} \cap \mathbf{D} = \varnothing$,
- I: $\mathbf{T} \rightarrow \mathbf{P}^{\infty}$ is the input function, a mapping from transitions to bags of places,
- O: $\mathbf{T} \rightarrow \mathbf{P}^{\infty}$ is the output function, a mapping from transitions to bags of places,
- cf: $\mathbf{T} \rightarrow [0, 1]$ is an association function, a mapping from transitions to real values between zero and one,
- α: $\mathbf{P} \rightarrow [0, 1]$ is an association function, a mapping from places to real values between zero and one,
- β: $\mathbf{P} \rightarrow \mathbf{D}$ is an association function, a mapping from places to propositions.

If $P_i \in I(T_j)$, then there is a directed edge from P_i to the transition T_j. If $P_k \in O(T_j)$, then there is a directed edge from the transition T_j to the place P_k. If $cf(T_j)$ is μ_j , then the transition T_j is said to be associated with a real value μ_j. If $\beta(P_i) = D_i$, then the place P_i is said to be associated with the proposition D_i. The four types of fuzzy production rules can be represented by the fuzzy petri nets depicted in Figures 4(a), 4(b), 4(c), and 4(d) respectively (Chen et al. 1990).

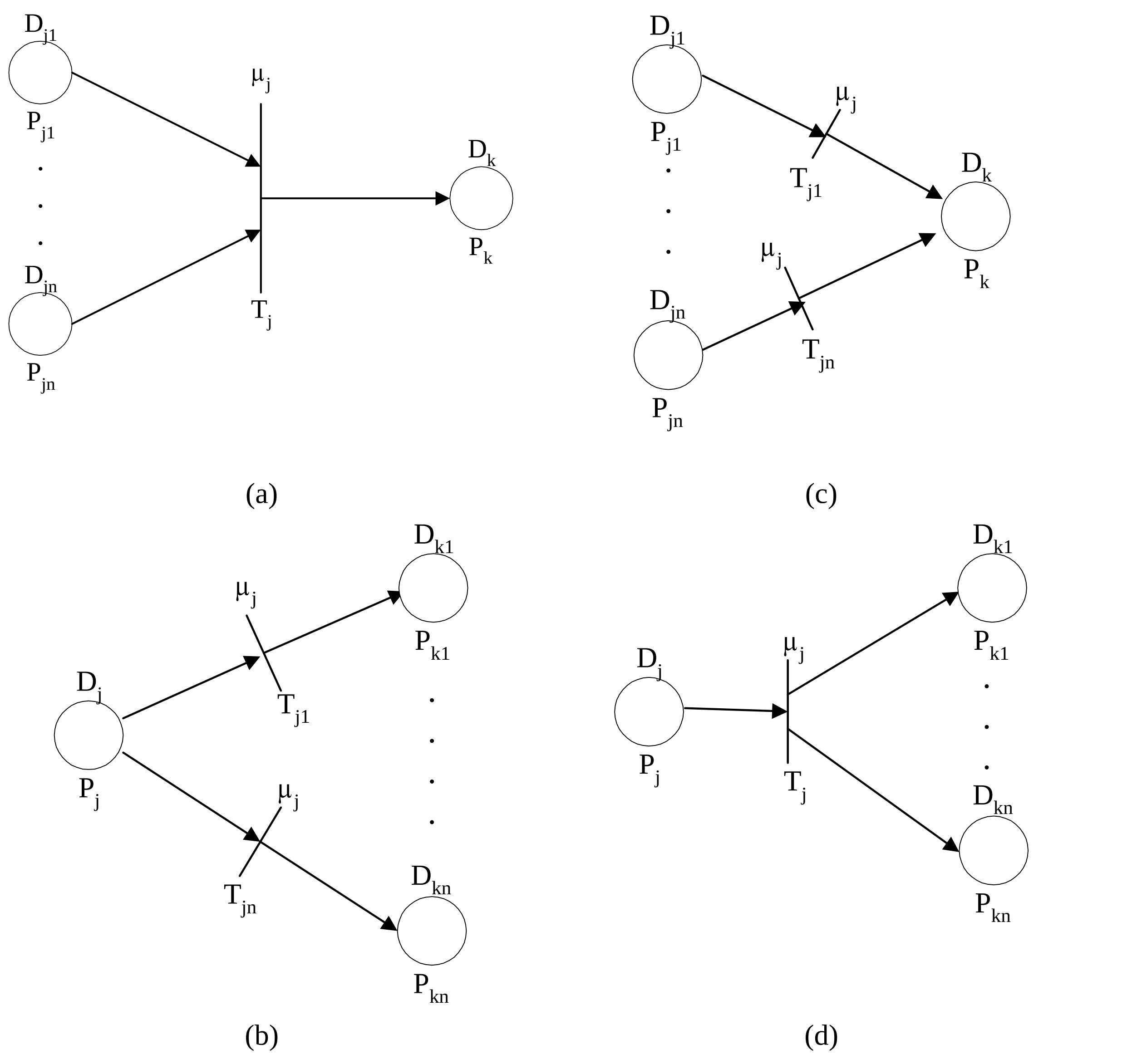


Figure 4: Fuzzy Petri net representations of (a) type 1 rules, (b) type 2 rules, (c) type 3 types, and (d) type 4 rules

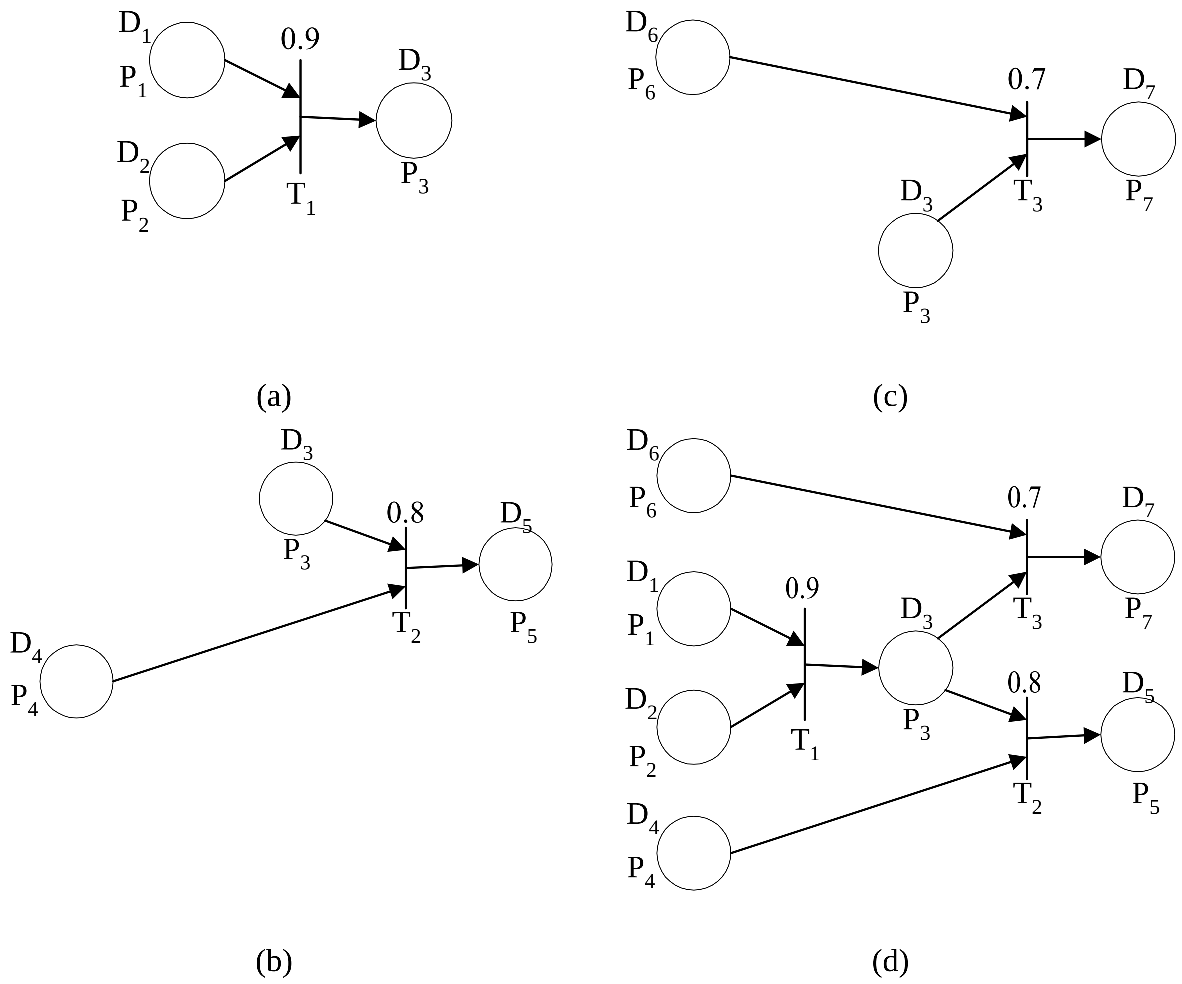
Consider the following fuzzy production rules:

IF D_1 and D_2 THEN D_3 WITH CF = 0.9

IF D_3 and D_4 THEN D_5 WITH CF = 0.8

IF D_3 and D_6 THEN D_7 WITH CF = 0.7

Since these rules are type 1 rules, they are modeled by the fuzzy petri nets given in Figures 5(a), 5(b), and 5(c), respectively. By combining these fuzzy petri nets, the FPN in Figure 5(d) can be obtained.



D_1 P_1 D_2 P_2 0.9 T_1 D_3 P_3
(a)
D_6 P_6 0.7 D_7 P_7 D_3 P_3 T_3
(c)
D_3 P_3 0.8 D_5 P_5 T_2 D_4 P_4
(b)
D_6 P_6 0.7 D_7 T_3 P_7 D_1 P_1 0.9 D_3 P_3 D_2 P_2 T_1 0.8 D_5 T_2 P_5 D_4 P_4
(d)

Figure 5: Fuzzy Petri net representations of (a) the first rule, (b) the second rule, (c) the third rule, and (d) the three rules

The FPN represents a knowledge base containing the three fuzzy production rules. In the FPN,

- $P = \{P_1, P_2, P_3, P_4, P_5, P_6\}$,
- $T = \{T_1, T_2, T_3\}$,
- $D = \{D_1, D_2, D_3, D_4, D_5, D_6, D_7\}$,
- $I(T_1) = \{P_1, P_2\}$, $I(T_2) = \{P_3, P_4\}$, $I(T_3) = \{P_3, P_6\}$

- $O(T_1) = \{P_3\}$, $O(T_2) = \{P_5\}$, $O(T_3) = \{P_7\}$
- $cf(T_1) = 0.9$, $cf(T_2) = 0.8$, $cf(T_3) = 0.7$
- $\beta(P_1) = D_1$, $\beta(P_2) = D_2$, $\beta(P_3) = D_3$, $\beta(P_4) = D_4$, $\beta(P_5) = D_5$, $\beta(P_6) = D_6$, $\beta(P7) = D_7$

3.2. Cellular Encoding for Fuzzy Petri Nets

Cellular encoding is a technique that uses genetic programming to concurrently evolve the architecture of a neural network, together with the weights, thresholds, and biases of the neurons in the neural network (Gruau 1994). In this technique, each individual program in the population is a specification for developing a neural network from a simple embryonic neural network that contains only a single neuron. Genetic programming is applied to evolve programs that create neural networks capable of solving a problem. Since the structure of a FPN is similar to that of a neural network, genetic programming can be used to evolve programs that create complete FPNs from embryonic FPNs.

It can be observed from Figure 5(d) that the places of a FPN can be classified into:

- *Input places*. For example, the places P_1, P_2, P_4, and P_6 are input places. The propositions associated with input places are attribute value pairs such as ‘Height is Tall’, ‘Weight is heavy’, etc.
- *Intermediate places*. For example, the place P_3 is an intermediate place. There is not restriction on the format of propositions associated with intermediate places.
- *Output places*. For example, the places P_5 and P_7 are output places. They represent the various conclusions that can be reached from the FPN.

Since the overall objective of the learning system is to obtain a FPN representing fuzzy production rules for a particular problem, the structure of the obtained FPNs must be restricted, so that only FPNs containing the appropriate numbers of output places can be induced. If there are n different conclusions in the set of production rules, where $n > 2$, the corresponding FPN must contain n output places. Otherwise, only one output place is required if the number of different conclusions is 1 or 2. This constraint can be fulfilled by initializing the embryonic FPN with a specific structure. For example, the embryonic FPN for a complete FPN with one output place is given in Figure 6. The place P_1 and the transition T_1 are modifiable while the output place P_1' is fixed.

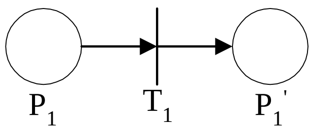


Figure 6: An embryonic FPN for a complete FPN with one output place

The corresponding FPN-constructing program contains two component programs (Table 3):

- the place-manipulating program which specifies how to process the input place P_1,
- the transition-manipulating program which specifies how to process the transition T_1.

```
(progn
  <place-manipulating program>
  <transition-manipulating program>
)
```

Table 3: The structure of a FPN-constructing program

For a complete FPN with n output places, the embryonic FPN has n input places, n transitions, and n output places (Figure 7). The modifiable components are P_1, P_2, …, P_n, T_1, T_2, …, and T_n. On the other hand, the output places P_1', P_2', and P_n' are fixed. The corresponding FPN-constructing program contains 2*n component programs (Table 4).

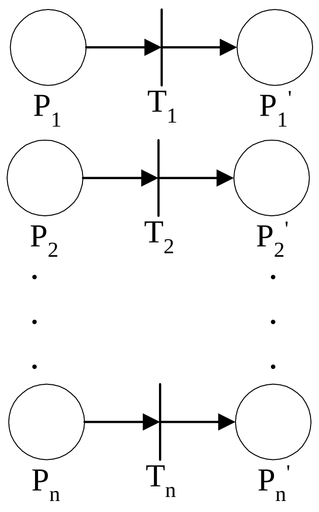


Figure 7: An embryonic FPN for a complete FPN with n output places

```
(progn
  <place-manipulating program 1>
  <transition-manipulating program 1>
  <place-manipulating program 2>
  <transition-manipulating program 2>
          …
          …
  <place-manipulating program n>
  <transition-manipulating program n>
)
```

Table 4: The structure of a FPN-constructing program

3.2.1. Place-manipulating functions

A place-manipulating function initializes the attribute value pair of an input place or inserts additional places and/or transitions in the developing FPN. Since there are two kinds of modifiable places: input and intermediate places, the functions can be classified into two kinds as shown in Table 5.

	Name	**Description**	**Number of arguments**
Input place	sp1	Sequential division	3
	pp1	Parallel division	2
	init	Initialize the attribute value pair	2
Intermediate place	sp2	Sequential division	3
	pp2	Parallel division	3
	stop	Terminate the modification	0

Table 5: Place-manipulating functions

Consider the developing FPN depicted in Figure 8, the sp1 function applies on the input place P_1 will create the new FPN given in Figure 9. It inserts a new transition T_4 and a new intermediate place $P_1^{'}$ into the FPN. The first argument of the sp1 function is a place-manipulating program that applies on the place P_1. The second argument is a transition-manipulating program that modifies the transition T_4. The third argument is a place-manipulating program that handles the intermediate place $P_1^{'}$.

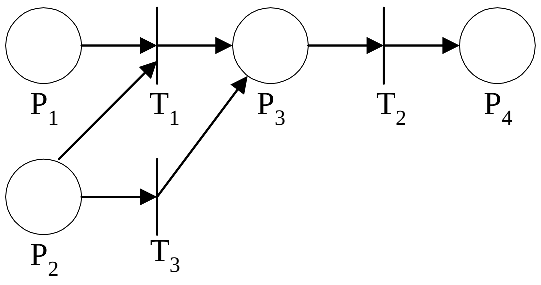


Figure 8: A developing FPN

The pp1 function applies on the input place P_1 of Figure 8 will create the new FPN depicted in Figure 10. It adds a new input place $P_1^{'}$ into the FPN. The first and the second arguments of the function are place-manipulating programs that apply on the input places P_1 and $P_1^{'}$ respectively.

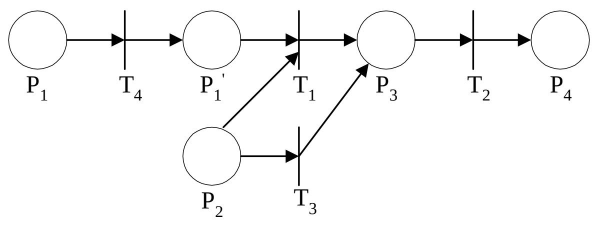


Figure 9: A new FPN created by the sp1 function

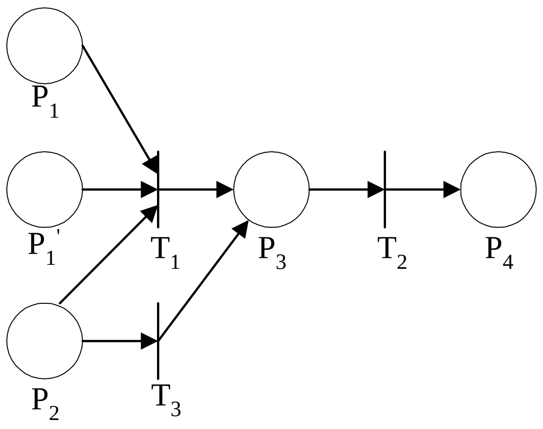


Figure 10: A new FPN created by the pp1 function

The init function initializes the attribute value pair of an input place. The first and the second arguments of the function are respectively the attribute and the value of the pair.

The sp2 function applies on the intermediate place P_3 of Figure 8 will produce the new FPN shown in Figure 11. It adds a new transition T_4 and a new intermediate place $P_3^{'}$. The first argument of the function is a place-manipulating program that handles the intermediate place P_3. The second argument is a transition-manipulating program that modifies the transition T_4. The last argument is a place-manipulating program that changes the intermediate place $P_3^{'}$

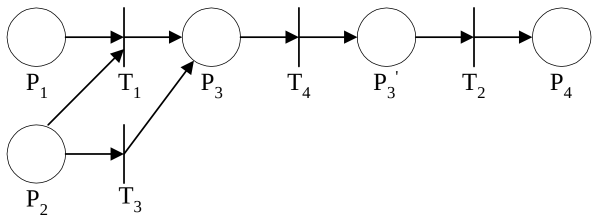


Figure 11: A new FPN created by the sp2 function

The pp2 function applies on the intermediate place P_3 of Figure 8 will create the new FPN depicted in Figure 12. The function first generates a new intermediate place $P_3^{'}$ and a new transition $T_1^{'}$. Subsequently, it connects the places P_1 and P_2 to $T_1^{'}$. Finally, the function connects $T_1^{'}$ to $P_3^{'}$ and attaches $P_3^{'}$ to T_2. The first argument of the function is a place-manipulating program that modifies P_3. The second one is a transition-manipulating program that operates on $T_1^{'}$. The last one is a place-manipulating program that handles the intermediate place $P_3^{'}$.

Finally, the stop function specifies that the corresponding intermediate place will not be modified any more.

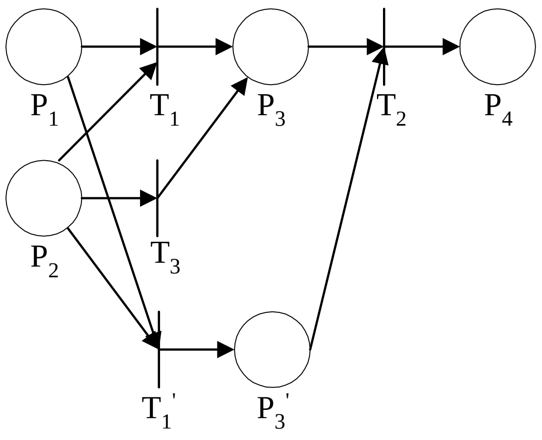


Figure 12: A new FPN created by the pp2 function

3.2.2. Transition-manipulating functions

The five transition-manipulating functions are summarized in Table 6.

Name	Description	Number of arguments
st	sequential division	3
pt	parallel division	2
cut	remove one of the incoming edges	2
setcf	set the certainty factor	1

Table 6: Transition-manipulating functions

Consider the developing FPN shown in Figure 8. The st function will produce the new FPN depicted in Figure 13 when applied on the transition T_2. It creates a new intermediate place P_5 and a new transition T_2'. The first argument of the function is a transition-manipulating program that operates on the transition T_2. The second one is a place-manipulating program that modifies the intermediate place P_5. The last one is a transition-manipulating program that changes the transition T_2'.

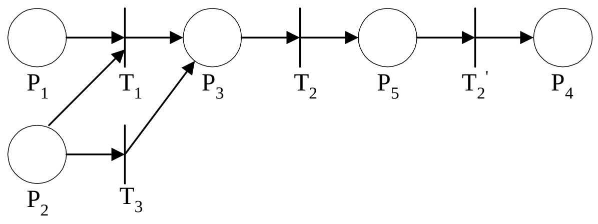


Figure 13: A new FPN created by the st function

A new FPN (Figure 14) will be produced if the function pt operates on the transition T_2 of Figure 8. A new transition T_2' is inserted. The first and the second arguments of the function are transition-manipulating programs applying on the transitions T_2 and T_2' respectively.

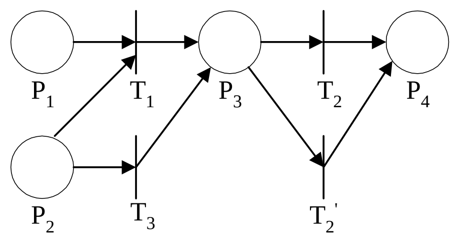


Figure 14: A new FPN created by the pt function

The cut function removes one of the incoming edges of a transition if there are more than one incoming edge. Otherwise, it does not modify the FPN. The first argument of the function is a transition-manipulating program that operates on the transition and the second argument identifies the incoming edge to be removed. Assume that the number of incoming edges of the transition is I and the second argument of the function is N where N > 1. The function first calculates the value (I mod N). The first incoming edge is removed if the calculated value is 0.. Otherwise, the corresponding incoming edge is removed. For example, the new FPN (Figure 15) will be produced if the cut function operates on the transition T_1 of Figure 8 with the value of the second argument equal to 5.

Finally, the setcf function takes one argument cf and initializes the certainty factor of the transition to cf.

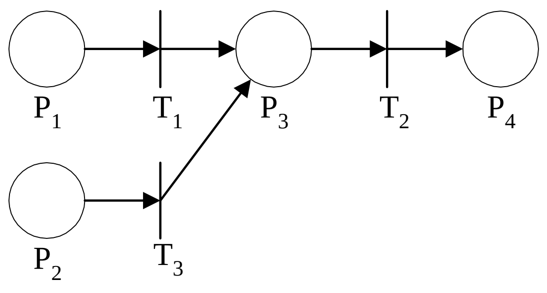


Figure 15: A new FPN created by the cut function

3.2.3. Generate Fuzzy Petri Nets

In the previous two sub-sections, we discussed the place-manipulating and transition-manipulating functions that operate on a developing FPN. In this sub-section, we give an example to demonstrate the procedure of generating a complete FPN from an embryonic FPN as shown in Figure 6.

Assume that we expect to develop a knowledge-based system which can help medical doctors to determine whether a patient should be observed by considering their height, weight, and age. The knowledge base of this system is composed of a number of fuzzy production rules which are represented as a FPN. This FPN can be obtained by executing the following FPN-constructing program on the embryonic FPN given in Figure 16,

```
(prong
  (pp1 (pp1 (sp1 (pp1 (init height tall) (init weight heavy))
                 (setcf 0.8) (stop))
            (init age old))
       (init age young))
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
)
```

The place-manipulating program,

```
  (pp1 (pp1 (sp1 (pp1 (init height tall) (init weight heavy))
                 (setcf 0.8) (stop))
            (init age old))
       (init age young))
```

specifies how to process the place P_1 of Figure 16, and the transition-manipulating program,

```
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
```

specifies how to process the transition T_1 of Figure 16.

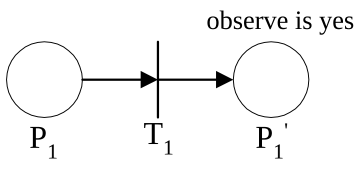
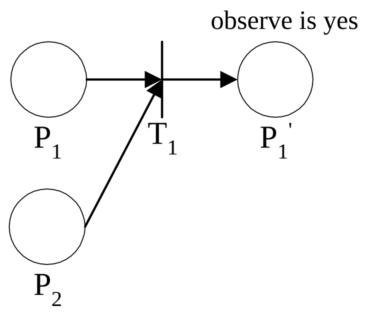


Figure 16: The embryonic FPN for the problem

The place-manipulating program is executed first. The pp1 function creates the new FPN depicted in Figure 17. The first argument of the function is the place-manipulating program,

```
  (pp1 (sp1 (pp1 (init height tall) (init weight heavy))
            (setcf 0.8) (stop))
       (init age old))
```

that will be applied on the input place P_1. The second argument is another place-manipulating program `(init age young)` that will be applied on the input place P_2.



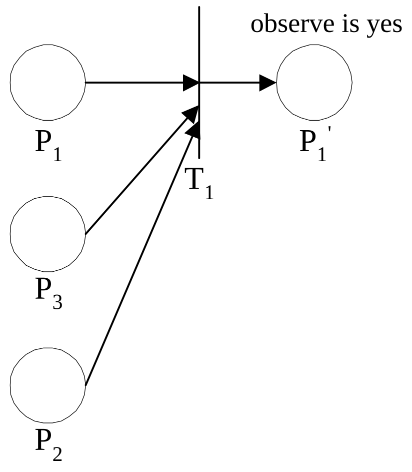
```
(prong
  (pp1 (pp1 (sp1 (pp1 (init height tall)
                      (init weight heavy))
                 (setcf 0.8) (stop))
            (init age old))
       (init age young))
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
)
```

Figure 17: The developing FPN (on the left) generated by apply the pp1 function (the shaded program fragment on the right) on the place P_1 of Figure 16

The place manipulating function pp1 is then operated on the place P_1 of Figure 17. It generates the FPN shown in Figure 18. The first argument of the function is the place-manipulating program

```
  (sp1 (pp1 (init height tall) (init weight heavy))
       (setcf 0.8) (stop))
```

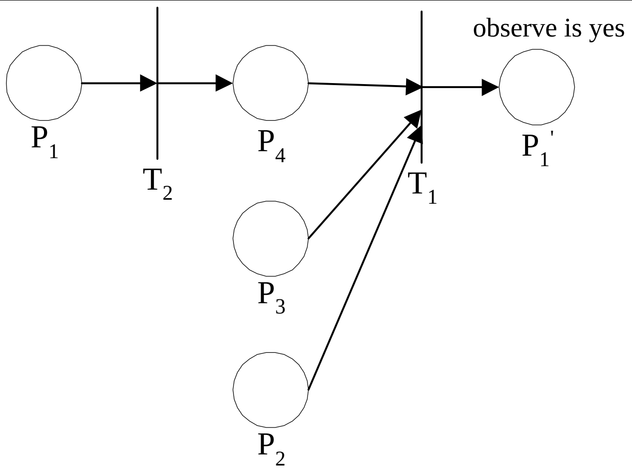
that will be applied on the input place P_1. The second argument is another place-manipulating program `(init age old)` that will be applied on the input place P_3.



```
(prong
  (pp1 (pp1 (sp1 (pp1 (init height tall)
                      (init weight heavy))
                 (setcf 0.8) (stop))
            (init age old))
       (init age young))
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
)
```

Figure 18: The developing FPN (on the left) generated by apply the pp1 function (the underlined program fragment on the right) on the place P_1 of Figure 17

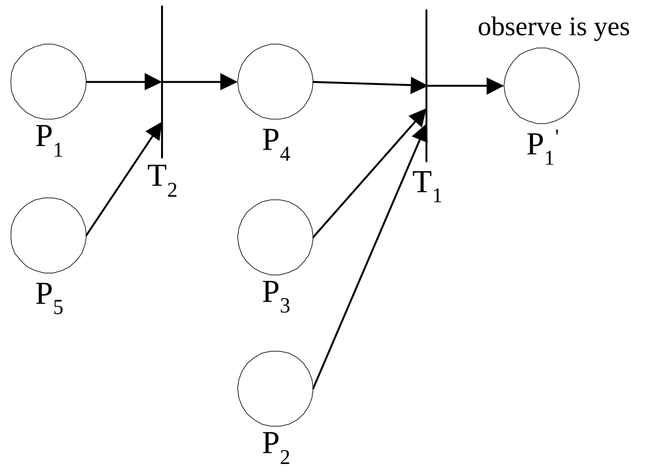
The place manipulating function sp1 is then operated on the place P_1 of Figure 18. It produces the FPN shown in Figure 19. The first argument of the function is the place-manipulating program `(pp1 (init height tall) (init weight heavy))` that will be applied on the input place P_1. The second argument is a transition-manipulating program `(setcf 0.8)` that will be applied on the new transition T_2. The third argument is a place-manipulating program `(stop)` that will be applied on the place P_4.



```
(prong
  (pp1 (pp1 (sp1 (pp1 (init height tall)
                      (init weight heavy))
                 (setcf 0.8) (stop))
            (init age old))
       (init age young))
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
)
```

Figure 19: The developing FPN (on the left) generated by apply the sp1 function (the underlined program fragment on the right) on the place P_1 of Figure 18

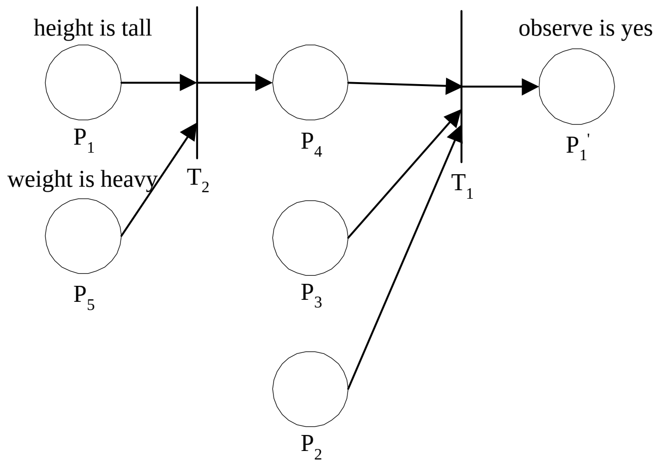
The place manipulating function pp1 is then operated on the place P_1 of Figure 19 and a new FPN is generated (Figure 20). The first argument of the function is the place-manipulating program `(init height tall)` that will be applied on the input place P_1. The second argument is a place-manipulating program `(init weight heavy)` that will be applied on the new input place P_5.



```
(prong
  (pp1 (pp1 (sp1 (pp1 (init height tall)
                      (init weight heavy))
                 (setcf 0.8) (stop))
            (init age old))
       (init age young))
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
)
```

Figure 20: The developing FPN (on the left) generated by apply the pp1 function (the underlined program fragment on the right) on the place P_1 of Figure 19

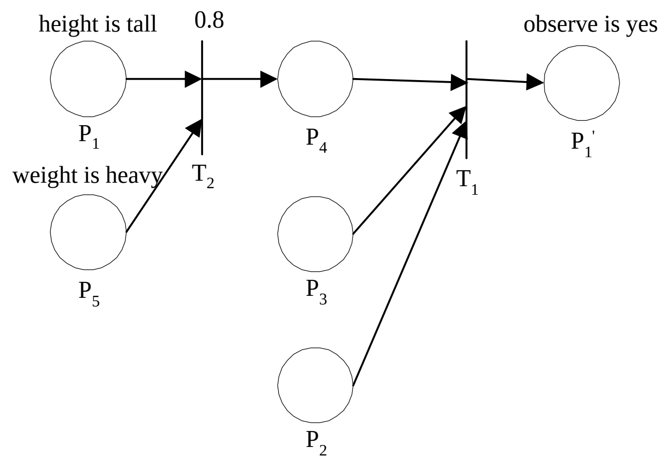
The program `(init height tall)` is applied on the input place P_1 and then the program `(init weight heavy)` is operated on the place P_5. A new FPN is generated (Figure 21).



```
(prong
  (pp1 (pp1 (sp1 (pp1 (init height tall)
                      (init weight heavy))
                 (setcf 0.8) (stop))
            (init age old))
       (init age young))
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
)
```

Figure 21: The developing FPN (on the left) generated by apply the init function (the underlined program fragments on the right) on the places P_1 and P_5 of Figure 20

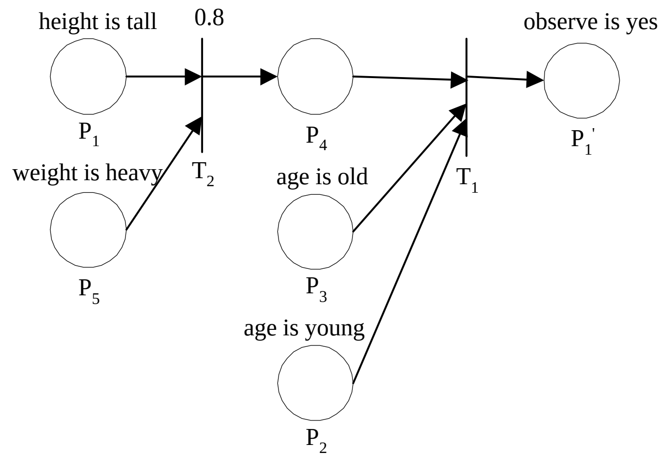
The program `(setcf 0.8)` is applied on the transition T_2 and then the program `(stop)` is operated on the place P_4. A new FPN is generated (Figure 22).



```
(prong
  (pp1 (pp1 (sp1 (pp1 (init height tall)
                      (init weight heavy))
                 (setcf 0.8) (stop))
            (init age old))
       (init age young))
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
)
```

Figure 22: The developing FPN (on the left) generated by apply the setcf function on the transition T_2 and the stop function on the place P_4 (the underlined program fragments on the right) of Figure 21

The program `(init age old)` is applied on the place P_3 and then the program `(init age young)` is operated on the place P_2. These programs produce a new FPN depicted in Figure 23.



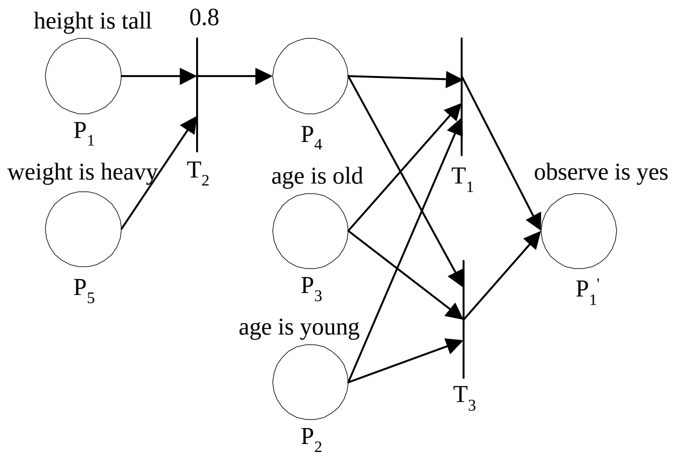
```
(prong
  (pp1 (pp1 (sp1 (pp1 (init height tall)
                      (init weight heavy))
                 (setcf 0.8) (stop))
            (init age old))
       (init age young))
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
)
```

Figure 23: The developing FPN (on the left) generated by apply the init function on the places P_3 and P_2 (the underlined program fragments on the right) of Figure 22

The transition-manipulating program,

```
(pt (cut (setcf 0.9) 2)
    (cut (setcf 0.4) 1))
```

is then operated on the transition T_1 of Figure 23. The pt function creates the new FPN illustrated in Figure 24. The first argument of the function is the transition-manipulating program
`(cut (setcf 0.9) 2)` that will be applied on the transition T_1. The second argument is another transition-manipulating program `(cut (setcf 0.4) 1)` that will be applied on the new transition T_3.



```
(prong
  (pp1 (pp1 (sp1 (pp1 (init height tall)
                      (init weight heavy))
                (setcf 0.8) (stop))
           (init age old))
       (init age young))
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
)
```

Figure 24: The developing FPN (on the left) generated by apply the pt function on the transition T_1 (the underlined program fragment on the right) of Figure 23

When the program `(cut (setcf 0.9) 2)` is operated on the transition T_1, the arc from the place P_2 is deleted first, and then the certainty factor of the transition T_1 is set to 0.9. After that the program `(cut (setcf 0.4) 1)` is applied on the transition T_3. The arc from the place P_4 is removed, and then the certainty factor of the transition T_3 is set to 0.4. Since the FPN-constructing program finishes, the final FPN is generated and is illustrated in Figure 25. This FPN represents the following fuzzy production rules:

IF Height is tall and Weight is heavy THEN D_4 WITH CF= 0.8

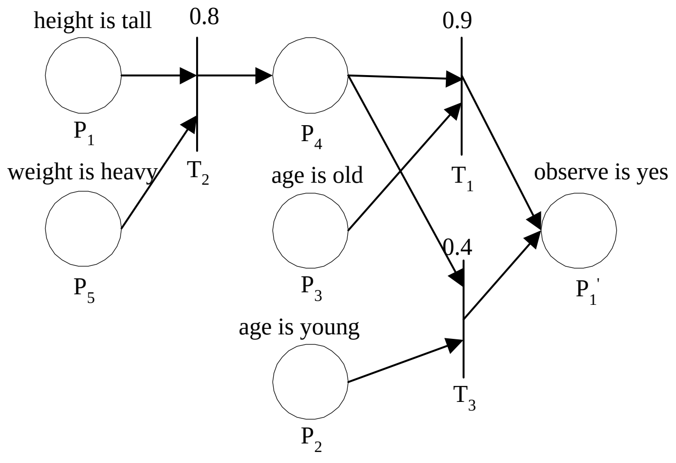
IF Age is old and D_4 THEN Observe is yes WITH CF = 0.9

IF Age is young and D_4 THEN Observe is yes WITH CF = 0.4

where

D_4 is a proposition associated with the place P_4. This proposition contains new fuzzy concepts.

One interesting feature of this approach for generating fuzzy Petri nets is that new intermediate fuzzy concepts can be produced. These concepts can improve the comprehensibility and the reasoning efficiency of the whole knowledge base by arranging the rules in the knowledge base in a hierarchical structure.



```
(prong
  (pp1 (pp1 (sp1 (pp1 (init height tall)
                      (init weight heavy))
                 (setcf 0.8) (stop))
            (init age old))
       (init age young))
  (pt (cut (setcf 0.9) 2)
      (cut (setcf 0.4) 1))
)
```

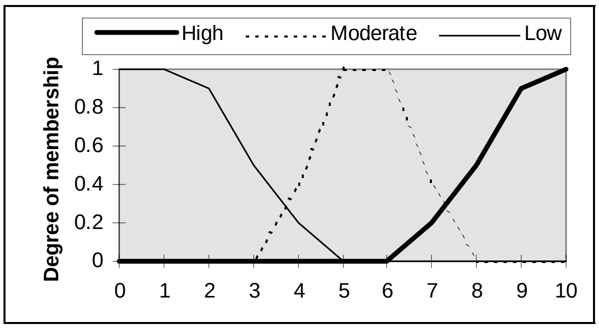
Figure 25: The complete FPN (on the left) generated by apply the cut and setcf functions on the transitions T_1 and T_3 (the underlined program fragment on the right) of Figure 24

3.3. Experiment

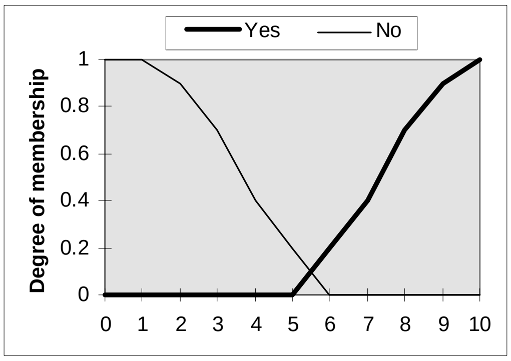
In this experiment, the performance of LOGENPRO in inducing FPN from training examples is evaluated using the malignant tumor classification program. The domain includes a set of 699 breast cancer patients, of which 458 have no malignancy and the remainder have confirmed malignancies. Each patient is described by nine features with real-values between 1 and 10. The features are classified into two categories:

- features that describe fuzzy concepts with 3 fuzzy terms ‘High’, ‘Moderate’, and ‘Low’,
- features that specify fuzzy concepts with 2 fuzzy terms ‘yes’ and ‘no’.

The membership functions of these fuzzy terms are depicted in Figure 26. The features, the attribute names, and the corresponding numbers of fuzzy terms are summarized in Table 7. The problem is to determine whether the tumors are benign or malignant from these cancer patients.



(a)



(b)

Figure 26: Membership functions of (a) 'High', 'Moderate', and 'Low', (b) 'Yes' and 'No'

feature	attribute name	number
clump thickness	clump	3
uniformity of cell size	cell-size	3
uniformity of cell shape	cell-shape	3
marginal adhesion	marginal-adhesion	3
single epithelial cell size	SE-cell-size	3
bare nuclei	nuclei-bare	2
bland chromatin	chromatin-bland	2
normal nucleoli	nucleoli	2
mitoses	mitoses	2

Table 7: Features and attribute names

To induce a program using LOGENPRO, we have to determine the logic grammar, fitness cases, fitness function, and termination criterion. The logic grammar for this problem is given in Table 8.

```
1:start     ->    [(progn], place1, tran, [)].
2:place1    ->    [(sp1], place1, tran, place2, [)].
3:place1    ->    [(pp1], place1, place1, [)].
4:place1    ->    [(init], attribute(?t), value(?t), [)].
5:place2    ->    [(], [sp2], place2, tran, place2, [)].
6:place2    ->    [(], [pp2], place2, tran, place2, [)].
7:place2    ->    [(stop)].
8:tran      ->    [(st], tran, place2, tran, [)].
9:tran      ->    [(pt], tran, tran, [)].
10:tran     ->    [(cut], tran, no, [)].
11:tran     ->    [(setcf], cf, [)].
12:cf       ->    {generate-cf(?x)}, [?x].
13:no       ->    { member(?x, [0, 1, 2, 3, 4]) }, [?x].
14:attribute(three) ->  { member(?x,[clump, cell-size, cell-shape,
                          marginal-adhesion, SE-cell-size]) }, [?x].
15:attribute(two) ->    { member(?x, [nuclei-bare, chromatin-bland,
                          nucleoli, mitoses])}, [?x].
16:value(three)   ->    { member(?x, [High, Moderate, Low]) }, [?x].
17:value(two)     ->    { member(?x, [Yes, No]) }, [?x].
```

Table 8: A logic grammar for the malignant tumor classification problem

The first rule of this grammar specifies that a FPN-constructing program has two components: a input-place-manipulating program and a transition-manipulating program. Rules 2, 3, and 4 specify the structures of input-place-manipulating programs. Rules 5, 6, and 7 declare the structures of intermediate-place-manipulating programs. The syntax of transition-manipulating programs is defined in rules 8, 9, 10, and 11. The logical goal `generate-cf(?x)` in rule 12 instantiates the variable `?x` to a random floating point number between 0 and 1. The goal `member(?x, [0, 1, 2, 3, 4])` in rule 13 instantiates `?x` to an integer between 0 and 4. Rules 14 and 15 specify the two categories of attributes while rules 16 and 17 define the different fuzzy terms.

The set of patients is divided into the training fitness cases and the testing example. There are 300 fitness cases (200 benign patients and 100 malignant patients). During each generation, 500 individual FPN-constructing programs of the population are executed to create 500 FPNs. Each FPN is used to classify the fitness cases. A fitness case is classified as benign if the FPN returns a degree of belief greater than 0.5. Otherwise, the case is classified as malignant. The fitness value of a FPN-constructing program is the number of misclassified fitness cases for the corresponding FPN. LOGENPRO terminates and returns the best program found during the run if the maximum number of generations (which is 50 in this experiment) is reached or a good FPN-constructing program is found. A program is good if it generates a FPN that classifies correctly all fitness cases.

The FPN generated by the best program is further evaluated by using the 399 testing examples (258 benign patients and 141 malignant patients). The classification accuracy is 94.98%. The same training set is given to C4.5 (Quinlan 1992) to induce a decision tree. The accuracy of the induced decision tree, 93.2%, is obtained by using the same testing set. This experiment shows that the induced fuzzy Petri net performs better than the decision tree induced by C4.5.

4. Learning logic program

In knowledge discovery from databases, we emphasize the need for learning from huge, incomplete, and imperfect datasets (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 the 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 relation learning systems such as FOIL (Quinlan 1990; 1991) and mFOIL (Lavrac and Dzeroski 1994) address the issue of learning 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 (a system that induces logic programs) in the domain of learning illegal chess endgame positions from noisy examples are presented.

In this domain, the target predicate `illegal(WKf, WKr, WRf, WRr, BKf, BKr)` states whether the position where the white king is at (WKf, WKr), the white rook is at (WRf, WRr) and the black king is at (BKf, BKr), is an illegal 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. LOGENPRO uses the logic grammar in Table 9 for this problem. In this 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 `?X` to either `WKf`, `WKr`, `WRf`, `WRr`, `BKf`, or `BKr`. The logic variables of the target logic program are `WKf`, `WKr`, `WRf`, `WRr`, `BKf`, and `BKr`.

```
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 9: The logic grammar for the chess endgame problem

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 amount of noise is introduced into the training examples in order to study the performances of both systems in learning programs 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. 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 noise introduced 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 and FOIL. Finally, the classification accuracy of the learned programs is estimated on the testing set. For LOGENPRO, the initial population of programs are induced by a variation of FOIL using a portion of the training examples. The population size is 10 and the maximum number of generations for each experiment is 50. Since LOGENPRO is a non-deterministic learning system, the process is repeated for five times on the same training set and the average of the five results is reported as the classification accuracy of LOGENPRO.

Many runs of the above experiments are performed on different training examples. The average results of ten runs are summarized in Figure 27. The performances of both systems are compared using paired t-test. From this experiment, the classification accuracy of both systems degrades seriously as the noise level increases. Nevertheless, the classification accuracy of LOGENPRO is better than that of FOIL by at least 5% 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. 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 can finds interesting and useful patterns from these noisy examples. The experiments demonstrate that LOGENPRO is a promising alternative to other famous inductive logic programming systems and sometimes is superior for handling noisy data.

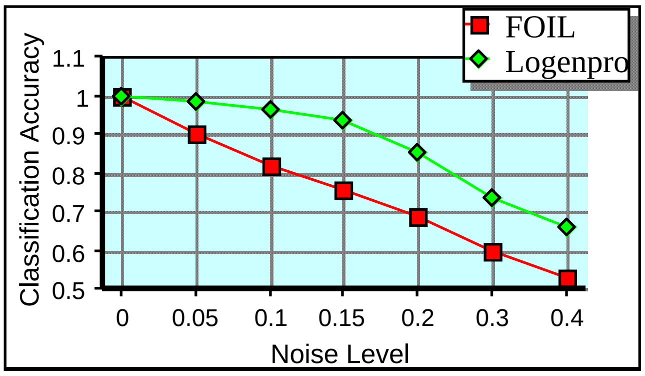


Figure 27: Comparison between LOGENPRO and FOIL

5. Conclusion

In this paper, we have presented a flexible knowledge discovery system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) that combines Genetic Programming and Inductive Logic Programming. It is based on a formalism of logic grammars. The system can learn programs in various programming languages and represent context-sensitive information and domain-dependent knowledge.

An experiment is performed to demonstrate that LOGENPRO can induce appropriate Fuzzy Petri Nets (FPN) representing a number of fuzzy production rules from information stored in databases. The performance of the FPN induced by LOGENPRO and the decision tree produced by C4.5 is compared using the malignant tumor classification problem.

The performance of LOGENPRO in inducing logic programs from imperfect training examples is evaluated using the chess endgame problem. A detailed comparison to FOIL has been conducted. This experiment demonstrates that LOGENPRO is a promising alternative to other inductive logic programming systems and sometimes is superior for handling noisy data.

These experiments show that LOGENPRO is a flexible knowledge discovery system because it can induce knowledge represented in various knowledge representation formalisms such as fuzzy Petri nets and logic programs. For future work, we plan to perform experiments on other learning problems to demonstrate that the system is applicable in different domains.

Acknowledgments

The research is sponsored by the direct research grant (reference number RES-001/199) of Lingnan University, Hong Kong.

References

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

Buchanan, B. G. and Shortliffe, E. H., Eds. (1984). *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, MA: Addison-Wesley.

Buchanan, B. G. and Wilkins, D. C., Eds. (1993). *Readings in Knowledge Acquisition and Learning: automating the construction and improvement of expert systems*. CA: Morgan Kaufmann

Chen, S. M., Ke, J. S., and Chang, J. F. (1990). Knowledge Representation Using Fuzzy Petri Nets. IEEE Transactions on Knowledge and Data Engineering, 2, pp. 311-319.

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

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

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

De Raedt, L. (1992). *Interactive Theory Revision: An Inductive Logic Programming Approach*. London: Academic Press

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.

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.

Feigenbaum, E. A. (1981). Expert systems in the 1980's. In A. Bond (ed.), *State of the Art Report on Machine Intelligent*. Maidenhead: Pergamon-Infotech.

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.

Gruau, F. (1994). Genetic Micro Programming of Neural Networks. In K. E. Kinnear, Jr. (Ed.) *Advances in Genetic Programming*. MA: MIT Press.

Hayes-Roth, F., Waterman, D. A. and Lenat, D. B. Eds. (1983). *Building Expert Systems*. Reading, MA: Addison-Wesley.

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

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

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. 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., Bennett, F. H. III, Andre, D., and Keane, M. A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann.

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

Leung, K. S. and Lam, W. (1988). Fuzzy concepts in expert systems. *IEEE Computer*, **21**, pp. 43-56.

Leung, K. S. and Wong, M. L. (1991a). Inducing and refining rule-based knowledge from inexact examples, *Knowledge Acquisition*. **3**, pp. 291-315.

Leung, K. S. and Wong, M. L. (1991b). Automatic refinement of Knowledge Bases with Fuzzy Rules. *Knowledge-Based Systems*, **4**, pp. 231-246.

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.

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

Muggleton, S. and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, 339-352. CA: Morgan Kaufmann

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

Pazzani, M., Brunk, C. A. and Silverstein, G. (1991). A Knowledge-Intensive Approach to Learning Relational Concepts, in *Proceedings of the Eighth International Workshop on Machine Learning*, 432-436, CA: Morgan Kaufmann.

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

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.

Peterson, J. L. (1981). Petri Net Theory and Modeling of Systems. NJ: Prentice Hall.

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

Quinlan, J. R. (1992). C4.5: Programs for Machine Learning. CA: Morgan Kaufmann.

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.

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

Stefik, M. (1995). *Introduction to Knowledge Systems*. San Francisco, CA: Morgan Kaufmann.

Wong, M. L. (1998). An Adaptive Knowledge Acquisition System using Generic Genetic Programming. *Expert Systems with Applications*, **15**, no. 1, pp.47-58.

Wong, M. L. and Leung, K. S. (2000). Data Mining Using Grammar Based Genetic Programming and Applications. Kluwer Academic Publishers.

Wong, M. L. and Leung, K. S. (1995). Inducing Logic Programs with Genetic Algorithms: The Genetic Logic Programming System, *IEEE Expert*, **9**, no. 5. pp. 68-76.

Zimmermann, H. J. (1986). *Fuzzy Set Theory and its Applications*. Dordrecht: Kluwer-Nijhoff.