MIXTURE OF EXPERTS LEARNING IN AUTOMATED THEOREM PROVING

by

Cemal Acar Erkek

B.S., Mathematics, Middle East Technical University, 2006

B.S., Computer Engineering, Middle East Technical University, 2006

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2010

MIXTURE OF EXPERTS LEARNING IN AUTOMATED THEOREM PROVING

APPROVED BY:

Assoc.Prof. Tunga Güngör . . . . . .
(Thesis Supervisor)

Assist.Prof. Taylan Cemgil . . . . . .

Assoc.Prof. Borahan Tümer . . . . . .

DATE OF APPROVAL:

ABSTRACT

MIXTURE OF EXPERTS LEARNING IN AUTOMATED THEOREM PROVING

The main challenge of automated theorem proving is to find a way to shorten the search process. Therefore using a good heuristic method is essential. Although there are several heuristics that improve the search techniques, studies show that a single heuristic cannot cope with all type of problems. The nature of theorem proving problems makes it impossible to find the best universal heuristic, since each problem requires a different search approach. Choosing the right heuristic for a given problem is a difficult task even for an human expert. Machine learning techniques were applied successfully to construct a heuristic in several studies. Instead of constructing a heuristic from scratch, we propose to use the mixture of experts technique to combine the existing heuristics and construct a heuristic. Since each problem requires a different approach, our method uses the output data of a similar problem while learning the heuristic for each new problem. The results show that the combined heuristic is better than each individual heuristic used in combination.

ÖZET

OTOMATİK TEOREM İSPATLAMA SİSTEMLERİNDE UZMANLARIN KARIŞIMI YÖNTEMİNİN UYGULANMASI

Otomatik Teorem İspatlama sistemlerinde başlıca zorluk, arama süreçlerini kısaltacak bir yöntem bulmaktır. Arama süreçlerini kısaltmak için, buluşsal yöntemleri kullanmak önemli bir rol oynar. Bu konuda yapılan çalışmaların sonucunda, çesitli buluşsal yöntemlerle başarılı sonuçlar alınmışsa da, tek bir buluşsal yöntemin tüm problem tiplerinin üstesinden gelemediği gösterilmiştir. Her problem tipinin değişik yaklaşım gerektirmesi, evrensel olarak en iyi buluşsal yöntemin bulunmasını imkansızlaştırır. Bir problem için doğru buluşsal yöntemin belirlenmesi, insan uzmanlar için bile çok zordur. Yeni bir buluşsal yöntem geliştirmek için makina öğrenmesi yöntemlerini kullanabiliriz. Bu tezde önerdiğimiz yaklaşım, sıfırdan yeni bir buluşsal yöntem geliştirmek yerine, uzmanların karışımı yöntemini kullanarak, var olan buluşsal yöntemlerin birleşiminden yeni bir buluşsal yöntem geliştirmektir. Her problem yeni bir yaklaşım gerektirdiği için, önerdiğimiz metot benzer problemlerin çözümlerini öğrenme verisi olarak kullanmaktadır. Çalışmamızın sonuçları, birleşik buluşsal yöntemin, karışımda kullanılan tekil buluşsal yöntemlerin her birinden daha iyi olduğunu göstermiştir.

TABLE OF CONTENTS

ABSTRACT iii
ÖZET iv
LIST OF FIGURES vi
LIST OF TABLES vii
LIST OF SYMBOLS/ABBREVIATIONS viii
1. INTRODUCTION 1
1.1. Motivation 2
1.2. Outline 5
2. PREVIOUS WORK 6
2.1. Learning in ATP systems 6
2.2. Clause features 8
2.3. Non-learning heuristics 10
3. MACHINE LEARNING 12
3.1. Machine learning and ANN 12
3.2. Mixture of experts 14
4. APPLICATION TO ATP 16
4.1. General architecture 16
4.2. Numerical features 17
4.3. Expert heuristics 18
4.4. The whole system 20
5. EXPERIMENTS AND RESULTS 23
5.1. Experimental setting 23
5.2. Results 23
5.3. Analysis of used knowledge 26
6. CONCLUSIONS AND FUTURE WORK 29
REFERENCES 30

LIST OF FIGURES

Figure 1.1. Generic ATP algorithm 2

Figure 1.2. Given-clause algorithm 4

Figure 3.1. Definition of discriminant functions 13

Figure 3.2. A typical representation of an ANN 13

Figure 3.3. A typical representation of a MOE 14

Figure 4.1. The system as a whole 17

Figure 4.2. Clause features 18

Figure 4.3. Problem features 18

Figure 4.4. Heuristic function 21

Figure 4.5. Example for areas of expertise 22

Figure 5.1. Results in pie charts 25

Figure 5.2. Number of clauses used in learning (FLD) 27

Figure 5.3. Number of clauses used in learning (GRP) 27

Figure 5.4. Number of clauses used in learning (LCL) 28

LIST OF TABLES

Table 1.1. Theorem proving as a search problem 3

Table 2.1. Example static and dynamic features 9

Table 5.1. Experiment heuristics 24

Table 5.2. Results for all domains 24

Table 5.3. Average time for proofs in seconds 26

LIST OF SYMBOLS/ABBREVIATIONS

$A \rightarrow B$	A implies B
A^T	transpose of A

ANN	Artificial neural network
ATP	Automated theorem proving/prover
RBF	Radial basis function network
MOE	Mixture of experts network
SOS	Set of support
SVM	Support vector machine(s)
TPTP	Thousands of Problems for Theorem Provers

1. INTRODUCTION

The birth of automated theorem proving (ATP) is based on the need of mathematicians for finding a general decision procedure to verify the validity or inconsistency of a mathematical formula [1]. Several mathematicians worked on the subject since 1600s. Today, we know general decision procedures to verify the validity of a mathematical formula which is defined in first order logic, if the formula is valid, otherwise we should not expect the procedure to terminate.

After the introduction of resolution principle, and the evolution of computers, computers became the main tool for ATP systems. Computers are very suitable for ATP since the resolution principle reduces proof procedures into a series of simpler, unintelligent operations, which computers are very fast at performing.

Nowadays, ATP applications are used in several areas. For example, some mathematical theorems which were not proved yet by human effort, are proved by ATP systems. Also, there are several industrial uses of ATP systems such as hardware verification, software verification or ontologies.

Machine learning is also an important area of computer science, where computers are used to solve problems by the analysis of example data or past experience [2]. Machine learning applications are used widely in several applications: natural language processing, medical diagnosis, credit approval in banking, stock market analysis, biometric authentication, game playing, etc. Therefore, we can conclude that most of the machine learning algorithms are mature, widely used and proven to be successful.

A special type of machine learning algorithms, artificial neural networks are also widely used in several machine learning algorithms, including ATP systems. There is a variant of ANN algorithms, which is called mixture of experts (MOE). The effort of this thesis is to integrate MOE algorithm into ATP. Our work proposes a method for using MOE method in the clause selection mechanism of an ATP system. The results are promising, but

further improvements can be achieved by slight modifications.

1.1. Motivation

If some formula G logically follows from other formulae ($F_1, F_2, ..., F_n$), then the formula ($F_1 \wedge ... \wedge F_n \rightarrow G$) is called a theorem [1]. Then we call $F_1, F_2, ..., F_n$ as axioms of the theorem, and G as the conclusion of the theorem. The goal of ATP is to decide whether G logically follows $F_1, F_2, ..., F_n$, and find the logical steps (not always human interpretable) that comes to G assuming $F_1, F_2, ..., F_n$ are given. To achieve this goal, ATP systems (which use resolution method) follow roughly the simplified generic algorithm in Figure 1.1.

1. The theorem is represented in first order logic.
2. The axioms of the theorem are transformed into a set of clauses.
3. The conclusion of the theorem is negated and added to the set of clauses.
4. Two clauses are chosen from the set of clauses.
5. The resolution method is applied to the chosen clauses, which adds new clauses to the set of clauses.
6. Step 4 and 5 are repeated until the false clause (or the empty clause) is produced by the resolution method.

Figure 1.1. Generic ATP algorithm

The detailed descriptions of symbolic logic, ATP and resolution method can be found in [1] and [3], therefore we do not explain here the exact details of the principles.

The important point about the above algorithm is that the resolution method is an undirected process, which produces every possible clause from any two clauses. Because, when we produce clauses, we do not exactly know which clauses will help us to reach the empty clause. Therefore, resolution method may produce a lot of clauses, in other words more clauses than it processes. In the loop, after some steps, the set of clauses will be very large, resulting in no solutions (no proof of the given theorem) at a reasonable time. After the introduction of resolution method, several improvements have been applied to make ATP systems faster. Several variants of resolution method are used to limit the number of possible resolutions, therefore producing less number of new clauses. The algorithm

which decides the order of resolutions is improved. Methods for discarding unnecessary and logically equivalent clauses are used.

The order of resolutions (deciding to which clauses the resolution method is to be applied) introduces a search problem whose parameters are defined in Table 1.1.

State	During the search, the current state is the set of clauses to be processed.
Initial state	The initial set of clauses which includes axioms and the negation of the conclusion of the theorem.
Possible actions	At a specific state, the action is to choose two clauses to apply resolution method. The new state is the previous set of clauses and the new clauses that are produced at this resolution step.
Goal test	If any new clause is the false clause (or empty clause), then search is over and the theorem is proved. Alternatively, if all possible resolutions are made without finding the empty clause, then the search is over and the theorem could not be proved.
Path cost	The time spent for each resolution is the estimated cost of each resolution step.

Table 1.1. Theorem proving as a search problem

Actually, all descriptions in this chapter are simplifications of the working principles of ATP systems. An actual ATP system should have a mechanism to keep track of previous resolutions (so it does not repeat the same resolutions), should periodically eliminate unnecessary clauses to shrink the set of clauses, should have an efficient data structure to speed up the search process, etc. One of the methods that ATP systems implement is the set of support strategy [1]. Several ATP systems use that strategy to narrow the possibilities for resolutions. A popular and efficient algorithm used in ATP systems is the given-clause algorithm. This algorithm is defined in [4] and is given in Figure 1.2. This is the algorithm used in the Otter ATP system.

Defining the problem as a search problem allows us to use search algorithms. The given-clause algorithm uses the best-first search method. In the Otter ATP system, the heuris-

There are two main lists of clauses:

usable	: This list contains clauses that are available to make inferences.
set of support (sos)	: This list contains clauses that are waiting to participate in inferences.

The main loop of the search process to find the empty clause is as follows:

1. Let *given_clause* be the best clause in *sos*.
2. Move *given_clause* from *sos* to *usable*.
3. Apply all possible inference rules (resolution or other rules), such that each new clause will have the *given_clause* as one of its parents and members of *usable* as its other parents.
4. If empty clause is found, end the search process.
5. If *sos* is not empty, repeat from step 1.

Figure 1.2. Given-clause algorithm

tic function used to define goodness is the weight (number of symbols) of the clause [4]. It is a simple heuristic and easy to implement. Also, it is a useful heuristic since shorter clauses tend to produce the empty clause faster than longer clauses. But it is limited because if the clauses that will help in the proof are long, we have to process every short clause before long clauses, which possibly increases the number of clauses in the usable list.

We should note an important fact here. At any time in the search process, the given clause is used in inferences with whole usable list. So, if the usable list is larger, there are more possibilities for inference. Therefore, we can conclude that processing a given clause at the beginning of the search takes less time than a given clause in the later stages of the search. A good heuristic function is essential.

Human effort can be and was used for inventing good heuristics. Learning from experience is another useful way for inventing good heuristics ([3, 5]). Machine learning concepts are used in several applications to improve ATP systems [5]. The details will be discussed in

the following chapters.

MOE method is a variant of artificial neural networks. It can be used to improve the current heuristic functions or to find a new heuristic function for the given-clause algorithm (or similar other algorithms whose success depends on choosing a useful clause). [2] tells us that we can use the MOE method to combine multiple learners, so that we could have a better learner than the individual learners which are used in the combination. Our main motivation is that the MOE method can be used in learning, so we can combine the characteristics of learned heuristics or existing conventional heuristics.

1.2. Outline

In the next chapter, we will discuss ATP systems which apply machine learning methods, and other works which the thesis is based on. In the third chapter, the details of our approach will be given. In the fourth chapter, the details of our implementation of the prototype system will be discussed. Then, we will present the experimental work and the results. Finally, we will discuss the results and possible future work.

2. PREVIOUS WORK

The effort to make use of machine learning in ATP systems is not a new idea, but several improvements were needed in artificial intelligence and ATP systems, before we observed convincing results [5]. In recent years, several different learning methods have been applied to ATP systems. We should analyze these methods before explaining the contribution in this thesis.

A general framework for learning ATP system is defined by the following questions [5]. Although some of the answers of these questions overlap, usually all learning ATP systems should deal with these questions.

- Learning phase:
  - i) Whom and what to learn from?
  - ii) What to learn?
  - iii) How to represent and store the learned knowledge?
  - iv) What learning method to use?
- Application phase:
  - i) How to detect applicable knowledge?
  - ii) How to apply knowledge?
  - iii) How to detect and deal with misleading knowledge?
  - iv) How to combine knowledge from different sources?
- Central question:
  - i) Which concepts of similarity are helpful?

2.1. Learning in ATP systems

An important work about the integration of ATP systems and neural networks is [6]. The approach is to learn search-guiding heuristics with a neural network. The learned heuristic (the neural network) is used to evaluate the possible branches of the search tree for deciding the order of the search. Each branch of the search tree represents a clause in the proof. Since neural networks are usually (not always, we will see other cases in this chapter)

suitable for using with numerical input, the clauses should be converted to numerical representations by the help of clause features (e.g. the number of literals in a clause). The training data of the neural network are example proofs, which are proved by the non-heuristic version of the prover. The steps which contribute to the proof are taken as positive training data. Branches that do not contribute, but are close to positive training data (siblings of positive training data) are taken as negative training data. Other data are ignored because the purpose of the method is to learn the nature of the clauses where positive and negative clauses are discriminated. The learned heuristic is better than the non-heuristic version of the prover. The same method was also used in other works [7], and its success was confirmed.

Case-based reasoning methods can be used for adapting search-guiding heuristics [8]. Each previously solved problem and its associated solution is a case for the current (target) problem. Then, the most suitable case is selected according to a similarity concept between problem definitions (axioms and conclusion of the proof problem). Later, a heuristic is configured according to the selected problem (the source problem). The case base is updated according to the success of the source, also the similarity measure can be updated to optimize the selection of cases. Genetic algorithms can also be used to adapt the parameters [9].

Instead of using numerical features, learning with symbolic patterns can be used for improving search-guiding heuristics [10, 11, 12]. Clauses are transformed into symbolic patterns and these patterns are used to construct term space maps. Each pattern is associated with the number of its positive/negative occurrences in previously solved examples. The clauses that contribute to the proof are taken as positive examples and the clauses which are close to the proof steps are taken as negative examples, and other steps are ignored (as in [6]). During the evaluation of a clause in the current proof, the pattern of the clause is calculated and then a bonus/penalty is added to a standard evaluation function according to the associated value in the term space map. The standard evaluation function is used (without bonus/penalty) for patterns which are absent in the term space map. The results are better than the standard heuristic without learning. Other methods are proposed to decide which previous examples will be used to construct the term space map of the current problem, including the features of the problem descriptions (axioms and conclusion of problems) [13]. The bonus/penalty concept is used with numerical features of clauses in [14].

If we have good heuristics at hand, we may wish to choose the most suitable heuristic for the current problem. We can use machine learning methods to decide this suitability [15]. Assuming that we have all of the necessary information about the successes of heuristics for previous examples (i.e. how much time we need for a problem and heuristic pair), then we can suggest a sequence of heuristics according to their expected success. This sequence is calculated according to the similarities between the current problem and the previous example problems. Again, the numerical features of the problem descriptions (axioms and the conclusion) are used to define the similarity. Later, the nearest neighbor algorithm is used to construct the sequence (i.e. if a heuristic is successful with a similar problem, then its expected success is higher). The idea that the method finds a sequence, instead of a single heuristic, is if a heuristic is not successful in the proof, we could try the next one in the sequence.

Instead of choosing a heuristic from a set of heuristics, we may combine our heuristics in order to improve the success rate [16]. In this work, two different heuristics are learned, these heuristics are combined and it was found that the success of the combination is better than each one of the heuristics (in some of the tests).

There are several other works that combine ATP with machine learning. ATP may use proof methods, a sequence of inference rules, instead of atomic inferences. Also, proof methods can be learned from similar problems [17]. The inference steps in the positive examples are analyzed to extract proof methods. These proof methods are then applied in the current problem, if preconditions of the methods hold. Data mining methods are used to generate proof tactics, which are similar structures [18]. Finally, reinforcement learning methods can be combined with ATP [19].

2.2. Clause features

If we want to use mixture of experts learning, we need to convert our clauses into compact representations somehow. A common usage is the numerical representation of clauses. Although converting a clause into numerical representation causes some loss of information, numerical representations are powerful because of the following [5]:

- Numerical representations are suitable to express uncertain and inexact knowledge.
- We have a lot of knowledge for the concepts of similarity and distance for numerical representations.
- We have powerful and mature learning methods for numerical representations.

We can use certain numerical features of clauses to convert into numerical representations. This section gives the details of the previous work which uses the features of clauses.

The concepts of static features (which are independent from the state of the proof, can be calculated any time) and dynamic features (which are dependent to the state of the proof, must be calculated during run-time) are given in [6]. Examples are given in Table 2.1. The results of the work shows that static features are more reliable than dynamic features. Also, static features have less overhead since we calculate the feature vector only once when the clause is introduced. Some static features used in that work are number of literals in a clause, number of negative literals in a clause, number of distinct predicates in a clause, number of all variable occurrences in a clause, number of occurrences of constants in a clause, number of functions in a clause, and number of variables connecting two literals.

Static features	number of literals in a clause number of variables in a clause maximum nesting of a clause
Dynamic features	current depth in the proof total number of uses of a clause in the proof so far

Table 2.1. Example static and dynamic features

The Otter ATP system (the system on which we build our prototype implementation) uses number of symbols as the default mechanism for choosing clauses [4].

Also there are features used in similarity detection between proof problems. These features can also be used as clause features. Examples are:

- The number of functions with different arities (unary, binary, ternary) [15]
- Term depth of clauses (nesting) [13]

2.3. Non-learning heuristics

There are several non-learning heuristics currently used in ATP systems. Most of them are successful for several problems. One of the most common non-learning heuristic is the weighted sum of symbols. The definition of this heuristic can be given as:

$$\begin{aligned} weight(x) &= w_v && if\ x\ is\ a\ variable \\ weight(f(t_1,...,t_n)) &= w_f + \sum_{i=1}^{n} weight(t_i) && otherwise \end{aligned} \tag{2.1}$$

where x is term, $f(t_1,...,t_n)$ is a function of terms t_i, w_v is the weight chosen for variables, w_f is the weight chosen for functions, and $weight(clause)$ is the heuristic value of $clause$.

Different values for w_v and w_f can be used. Examples with ($w_f = 1$, $w_v = 1$), ($w_f = 2$, $w_v = 1$) and ($w_f = 1$, $w_v = 2$) are analyzed in [11]. When we use ($w_f = 1$, $w_v = 1$), it is actually counting symbols. It is also Otter's heuristic mechanism [4]. The reasons why this simple approach is successful are given in [11] as:

- Short clauses are more general than long clauses.
- Processing shorter clauses is faster and fewer new clauses are generated, reducing the explosion in the search space.
- Shorter clauses are more likely to generate the empty clause.

We can assign a linear polynomial with coefficients $c_1^f, ..., c_{n_f}^f$ for each function symbol f with order n_f. It is called the linpol heuristic [9]. The linpol heuristic introduces a new problem for determining the coefficients:

$$\begin{aligned} Linpol(x) &= w_v && if\ x\ is\ a\ variable \\ Linpol(f(t_1,...,t_{n_f})) &= w_f + \textstyle\sum_{i=1}^{n_f} c_i^f \times Linpol(t_i) && otherwise \end{aligned} \tag{2.2}$$

The occness heuristic [20] evaluates according to the similarity to the goal (the fact that we are trying to prove). It is used in several studies for comparisons [9, 10, 13, 21, 22]. It is designed to be used in equational theorem proving. The max heuristic is also another

heuristic used in equational theorem proving [9]. It takes the maximum weight of two sides of the equation, as the evaluation value of the whole equation.

3. MACHINE LEARNING

In this chapter, we will give definitions of the machine learning methods that we use in the thesis. In the first section, the definition of machine learning and important details will be discussed. In the second section, MOE will be introduced with its formal definitions.

Our machine learning approach and algorithms are based on [2]. All definitions and formulae are taken from there unless cited otherwise.

3.1. Machine learning and ANN

In [2], machine learning is defined as:

> "Machine learning is programming computers to optimize a performance criterion using example data or past experience. We have a model defined up to some parameters, and learning is the execution of a computer program to optimize the parameters of the model using training data or past experience."

Machine learning techniques can be used in classification applications. In classification, we have some example training data, where each example is a tuple (x_i, y_i) such that x_i is the input vector that represents the individual example i and y_i is the label (or class) assigned to that individual example. Our purpose is to predict the actual label y_j when a new example input x_j is given. For example, in a medical diagnosis application, the input vector may consist of patient's properties (age, gender, medical history), symptoms (body temperature, blood levels, presence of cough, presence of pain), etc. The labels in such an application may be the health status of the patient: healthy, flu, anemia, tuberculosis or other diseases. We can use such an application to assist doctors.

In classification, we try to (implicitly or explicitly) implement discriminant functions for each class. These discriminant functions divide the input space into decision regions. The formal definition of the discriminant functions are given in [2] as in Figure 3.1

$C_i, i = 1, ..., K$ are classes,
$g_i(x)$ are discriminant functions,
$R_i = \{x|g_i(x) = \max_k g_k(x)\}$ are decision regions,
then we assign the class C_i to example x if $g_i(x) = \max_k g_k(x)$,
or equivalently if $x \in R_i$

Figure 3.1. Definition of discriminant functions

In likelihood-based classification, we make assumptions about the densities of classes, and later calculate the posterior probabilities of classes. These posterior probabilities determine discriminant functions (and decision regions). In discriminant-based classification, we do not make assumptions about class densities, but make assumptions about the discriminant functions. In this approach, we do not calculate likelihoods or posterior probabilities, but we estimate the discriminant functions.

Artificial neural networks (ANN) are computational models which take their inspiration from the brain [2]. ANNs are composed of interconnected artificial neurons (small processing units). We can use different types of artificial neural networks in classification tasks. A typical ANN structure can be seen in Figure 3.2.

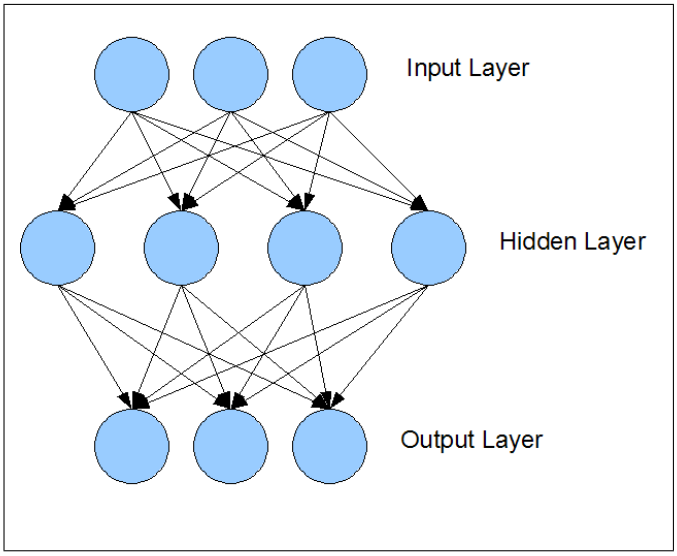


Figure 3.2. A typical representation of an ANN

In an ANN structure, if many hidden units are activated (have non-zero output) for inputs, then it is called a distributed representation. Multilayer perceptrons are examples of

distributed representation. In this kind of networks, input is encoded by the simultaneous activation of many hidden units.

We may choose to have a local representation instead. In this kind of networks, one or a few hidden units are active (have non-zero output) at the same time. Examples of this kind of network are radial basis function networks (RBF) and MOE.

3.2. Mixture of experts

The general structure of the MOE network can be seen in Figure 3.3.

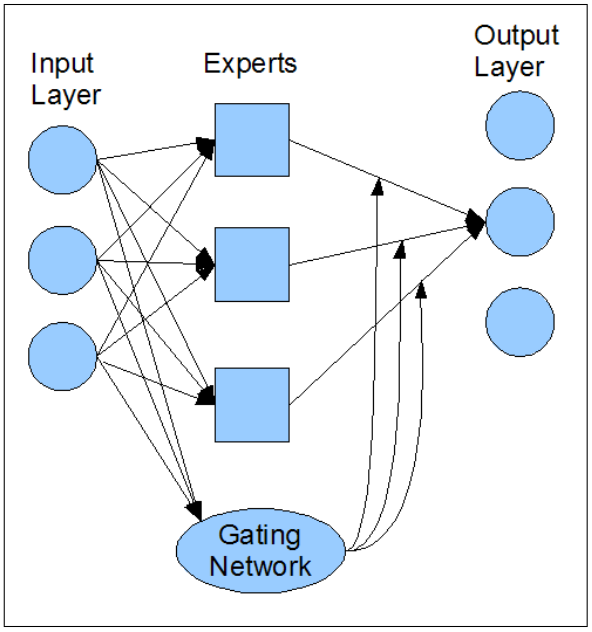


Figure 3.3. A typical representation of a MOE

There are two different perspectives for MOE. In the first case, MOE can be seen as an extension of RBF, where hidden layer outputs are not constants but linear models. In the second perspective, MOE can be seen as a general architecture for combining multiple experts, where the experts may not be linear or learning and the gating may not be linear. In this thesis, we will use the second perspective. We will give the details of this perspective in this section. All other details can be found in [2].

The idea behind MOE is to combine multiple experts to achieve success better than

each individual expert. This task is done by taking a weighted average of the expert outputs:

$$y^t = \sum_{h=1}^{H} g_h(x^t) \cdot w_h(x^t) \tag{3.1}$$

where y^t is the output of MOE for input x^t, $w_h(x^t)$ is the output of expert h, $g_h(x^t)$ is the weight of expert h, x^t is the current input for the system. As we can see from the above formula, both the weights and expert outputs depend on the input. These weights are determined by the gating network. We can use softmax gating for the gating network:

$$g_h(x^t) = \frac{\exp[m_h^T \cdot x^t]}{\sum_{l=1}^{H} \exp[m_l^T \cdot x^t]} \tag{3.2}$$

where the vector m_h defines a hyperplane for expert h , and H is the number of experts. This function fulfills a classification task for input vector x^t. For each given x^t, g_h determines if x^t is in the expertise region of expert h defined by m_h. The denominator of the formula ensures that $\sum g_h = 1$, and for some j, $g_j > 0$.

MOEs are trained with backpropagation algorithm. The update rule for m_h after each individual instance x^t is:

$$\Delta m_h = \eta[r^t - y^t] \cdot [w_h(x^t) - y^t] \cdot g_h(x^t) \cdot x^t \tag{3.3}$$

where r^t is the desired output for the training example x^t, and η is the learning factor.

It is worth pointing here that we describe the cooperative MOE method, where all experts cooperate for the output of the system. There is also a variation which is the competitive MOE method where units are forced to be separated. [2] remarks that the cooperative method is more accurate while the competitive method makes learning faster. We prefer the cooperative MOE in this thesis.

4. APPLICATION TO ATP

In this chapter, we describe our application of MOE method to ATP systems. We discuss the general design of the method and their realizations in our prototype system.

4.1. General architecture

In applying machine learning, our training data will be the output of the system, which indicates that we will use the proof steps of previous problems to solve new problems. Initially, since there is no training, we must use the outputs of problems solved with conventional heuristics.

A successful proof has two types of clauses: clauses that contribute to the solution (positive examples) and clauses that do not contribute to the solution (negative examples). Usually, a proof has much more negative examples than positive examples. If we include all positive and negative examples, negative examples will dominate the learning process. So we have to reduce the number of negative examples. [5] suggests to take negative examples which are close to positive examples. In a similar manner, we consider negative examples which are two steps away from positive examples in the proof tree.

If a proof has very few steps, then we will not have enough data for a successful training. So, we do not use solutions with very small number of proof steps in training. The positive and negative examples are converted into their numerical representations as described in Section 4.2. These numerical data are used to train the MOE. We train the network until the coefficients are stable. After this process, the problem definition of the proof and the coefficients are kept in a knowledge base. In our examples, all training sessions are very fast (takes less than 1 s.), so compared to the proof sessions, the total time of the training sessions is negligible.

When a new proof problem arrives, the problem definition of the new problem is compared with the previously solved problems and the knowledge of the most similar problem

is applied to the new problem. This concept, which is called instance-based learning, is successfully used in [15]. To determine the similarity, each problem is converted into numerical representations (problem features). Then, similarity is calculated as the Euclidean distance between these feature vectors. The feature set used in the application is given in Section 4.2.

The MOE is initialized with the coefficients taken from the most similar problem. The output of this MOE is used as the heuristic function in the given-clause algorithm. The system is visualized in Figure 4.1.

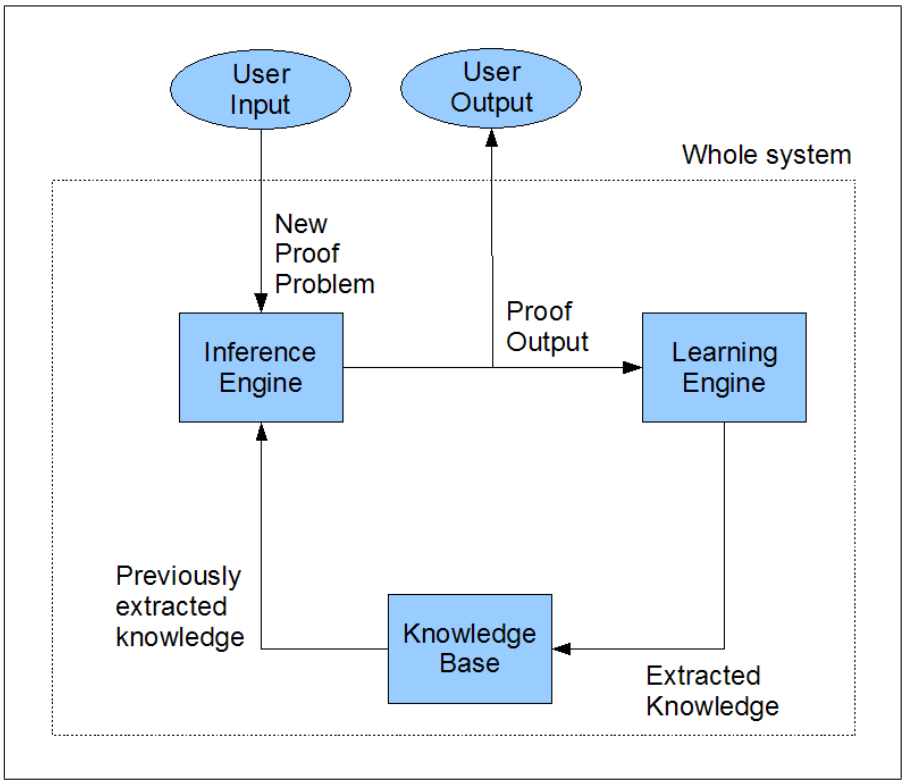


Figure 4.1. The system as a whole

4.2. Numerical features

In our application, we convert clauses into numerical representations since ANNs are commonly used with numerical input. We do not use dynamic features since static features are more simple and more reliable (see Section 2.2). Each clause is converted into a vector of features given in Figure 4.2. Then these vectors are used in the training of the MOE (specifically in the gating network).

1. Number of literals
2. Number of negative literals
3. Number of distinct predicate symbols
4. Number of constants
5. Number of functions
6. Number of variables
7. Number of variables connecting literals
8. Total number of symbols
9. Number of unary functions
10. Number of binary functions
11. Number of ternary functions
12. Maximum nesting
13. Maximum weight of literals

Figure 4.2. Clause features

We also use numerical features of problem definitions to determine the similarity between problems. The features used in the similarity concept are given in Figure 4.3.

1. Number of axioms
2. Average term depth of the axioms
3. Standard deviation of the term depth of axioms
4. A vector describing the distance of function arities in the signature of the problem
5. Number of distinct predicates

Figure 4.3. Problem features

4.3. Expert heuristics

We use clause heuristics as the experts of the system. The flexibility of MOE method allows us to use different types of heuristics together (conventional non-learning heuristics and learning heuristics). As an extreme case, we can also combine human experts but it would not be practical since both the learning phase and the application phase would be

difficult and slower because of human interaction.

In the preliminary work of this thesis, we encountered a problem when combining heuristics with different scales. The most common heuristics are symbol weighting heuristics in ATP systems (see Section 2.3). These heuristics count the symbols in clauses (by giving weights for symbols) and choose the clause with minimum weight. Different heuristics may assign different weights, resulting different output intervals for different heuristics. But this is not the only case, we may have an adaptive heuristic (for example multilayer perceptron), which may give its output in $[0, 1]$ interval, while choosing the clause with maximum output. In some cases, this problem will cause some of the combined heuristics to drop because of very small outputs or very small weights (gating outputs). We can see this fact in the hypothetical example in the following paragraph. In preliminary experiments, this problem reduces our system to one of the heuristics, so we cannot improve the system by learning.

Assume we have these heuristics combined:

- $h_1(clause) \in [0, 1]$
- $h_2(clause) \in [0, 100]$

At the earlier stages of learning, when weights are close, outputs of h_2 will be dominant in the output. Therefore y will be close to w_2, so $(w_2 - y)$ will be close to 0. Also, $(w_1 - y)$ will be significantly greater than 0. So, changes in m_1 will be faster, changes in m_2 will be slower since Equation 3.3 has the term $(w_h - y)$ (see also Equation 3.2). This fact spoils the learning at the earlier stages. Assume that earlier stages of our learning session goes well despite this fact. At the later stages, the weights of h_1 (in other words g_1) should be higher, and g_2 should be lower in order to balance the contribution of heuristics (if g_1 is small, then the contribution of h_1 will not be observable in the output). So, changes in m_1 will be faster, changes in m_2 will be slower since Equation 3.3 has the term g_h. In both cases, the learning is corrupted, and the system drops to one of the heuristics combined, no useful learning is achieved. Using heuristics with different scales should be avoided. We should normalize the outputs of these heuristics.

We need a way to normalize heuristics. Maybe the most simple method would be 0-

1 normalization (where all data will be mapped into $[0, 1]$ interval). But for some type of heuristics, there may be no lower bound or upper bound. For example for the simple symbol counting heuristic, the lower bound is 1, but there is no upper bound for the number of symbols in a clause. So, we come up with a simple but useful method: we filter the outputs of heuristics with perceptrons.

Perceptron is a basic structure which can be used as a linear classifier. It is the building block of an ANN. It seperates the input space with a hyperplane. It is defined with these equations:

$$o = w^T \cdot x \tag{4.1}$$

$$y = sigmoid(o) \tag{4.2}$$

$$sigmoid(t) = \frac{1}{1 + e^{-t}} \tag{4.3}$$

where y is the output of the perceptron, x is the input of the perceptron, w defines the hyperplane of the perceptron. See [2] for further details.

These perceptrons are trained as classifiers for seperating useful clauses and useless clauses from the outputs of heuristics. In other words, we calculate the posterior probabilty of being a useful clause from the heuristic output of that clause. After training, we treat the combinations of the heuristics and the perceptrons as the experts of MOE. This method ensures that our expert outputs are in $[0, 1]$ interval.

4.4. The whole system

In this section, we will give a summary of our implementation, so the system can be understood as a whole.

Our design is built on clause heuristics, which are called experts in our perspective. Since we allow any kind of heuristic, simply we assume that a heuristic gives its output in the interval $(-\infty, \infty)$. The perceptrons in our first level of MOE transform our heuristic

outputs to probability models also ensure that the outputs are in $[0, 1]$. At this stage, we interpret the output as a probabilistic prediction. These steps can be visualized as Figure 4.4 (heuristic input is visualized as one dimensional for simplicity; it could be any number of dimensions).

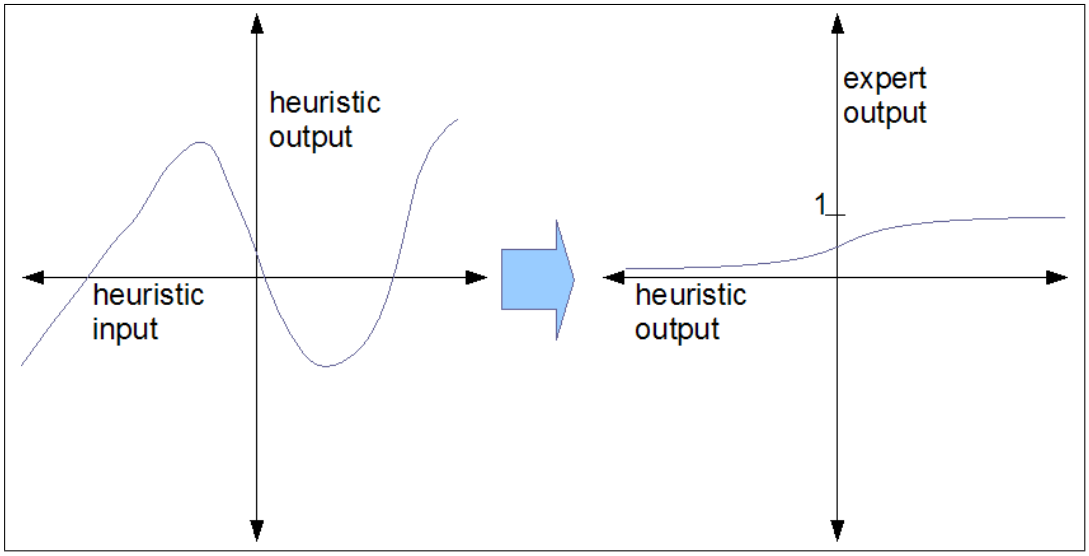


Figure 4.4. Heuristic function

Our heuristic functions are successful (give accurate output) for some of the clauses. One heuristic may distinguish the useful clauses, where another one may fail. Our assumption is that we can learn the subsets (subregions) of the clause domain where any heuristic is more successful than others. The gating network is responsible for this task. The outputs of the gating network (weights of experts for that input) determine the effect of experts for the combined output. So, for any clause (mathematically, a vector in the clause domain), one heuristic is dominant for the combined output. See Figure 4.5 for an example; the clause domain is given in two dimensional for simplicity in this example. It is 13 dimensional in the actual application.

In our gating structure, we use the soft-max gating method, so that our gating structure gives soft weights to the experts, instead of giving one for the dominant expert, and zeros for the others. The system output is a weighted sum of the experts, where the sum of the weights is one.

The expertise regions, mentioned above, are learned from previous examples. The

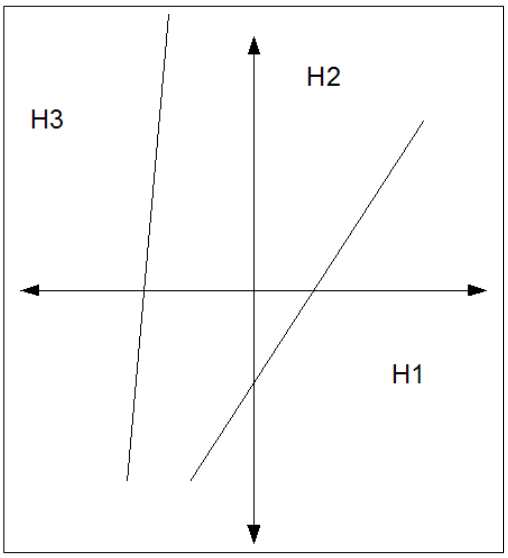


Figure 4.5. Example for areas of expertise

clauses are extracted from the output of a previous proof, with labels “useful” and “not useful”. Then, most of the unsuccessful clauses are filtered out as mentioned in Section 4.1. The rest of the clauses are used to train the MOE according to the rules in Section 3.2. The results of training are recorded in the knowledge base to be used for future problems.

Another assumption is that the outputs of a similar problem will be useful for the target problem (see Section 2.1). We used this approach to determine the knowledge to be used. For the new problem, its problem features are calculated and the most similar problem is chosen from the knowledge base, to assign the parameters of the MOE. In our application, Otter’s inner mechanisms are manipulated to use such a network for the clause selection mechanism.

5. EXPERIMENTS AND RESULTS

5.1. Experimental setting

In our experiments, we implemented the proposed method on top of the Otter ATP system [4]. Otter is a popular ATP system and it was used widely for comparison of ATP systems. We modified the clause selection mechanism of Otter to use the MOE. The other mechanisms of Otter were kept the same so that we can isolate the effect of the clause selection mechanism in the results.

Our learning engine is partly programmed in PHP (for the extraction of clauses from proof outputs) and in C (for the training of the MOE from the extracted clauses). The inference engine of Otter was manipulated to use the knowledge from previous examples (as trained MOE). Otter is programmed in C, so as our extension.

The experiments were done on an Intel Pentium 4 1.7 Ghz Ubuntu Linux computer. In all of the tests, a moderate time limit was given to the prover to prevent running indefinitely if it does not find a solution. Also, in later stages of proving, the efficiency of the prover reduces since both the set of support and the set of usable clauses expand quickly. If the prover does not find a solution until the time limit, the prover is stopped. This time limit is three minutes in our experiments. We used the TPTP (Thousands of Problems for Theorem Provers) library in the tests [23]. We used problems without equality, which are defined in clause normal form.

In the experiments, we combined three simple heuristics. Also, for comparison, we used two hypothetical heuristics. These heuristics are defined in Figure 5.1.

5.2. Results

We tested the proposed system in the following domains from the TPTP library: FLD, GRP, and LCL. The results are given in Figure 5.2 (percentages are rounded to integral

H_1	$w_v = 1, w_f = 1$ in Equation 1, the default heuristic of Otter
H_2	Maximum nesting level of the clause
H_3	$w_v = 1, w_f = 2$ in Equation 1
H_{best}	Hypothetical heuristic, which chooses the best from H_i where $i = 1, 2, 3$ for a specific problem
H_{worst}	Hypothetical heuristic, which chooses the worst from H_i where $i = 1, 2, 3$ for a specific problem
H_{moe}	MOE heuristic (the proposed heuristic in this paper)

Table 5.1. Experiment heuristics

values), also in Figure 5.1 as pie charts.

	FLD	GRP	LCL
Both H_{moe} and H_{best} succeeded	51%	61%	97%
Both H_{moe} and H_{best} failed	45%	27%	0
H_{moe} succeeded but H_{best} failed	3%	2%	0
H_{moe} failed but H_{best} succeeded	1%	10%	3%
H_{moe} is faster than H_{best}	14%	5%	16%
H_{moe} is as fast as H_{best}	22%	53%	57%
H_{moe} is faster than H_{worst} but slower than H_{best}	6%	2%	19%
H_{moe} is slower than H_{worst}	9%	1%	5%
Total number of problems (100%)	161	127	300

Table 5.2. Results for all domains

The proposed system finds a solution for seven problems (one per cent of all problems) where all other heuristics fail. In 76 problems (13 per cent of all problems), it finds a solution faster than all of the combined heuristics. In 274 problems (47 per cent), the success of H_{moe} is as fast as H_{best}, in other words, H_{moe} has a score equal to the best of the heuristics for a specific problem. In total, for about 61 per cent of all problems, our system gives at least equal or more successful results with H_{best}. So, we can conclude that the system has gained abilities beyond the combined heuristics for these problems.

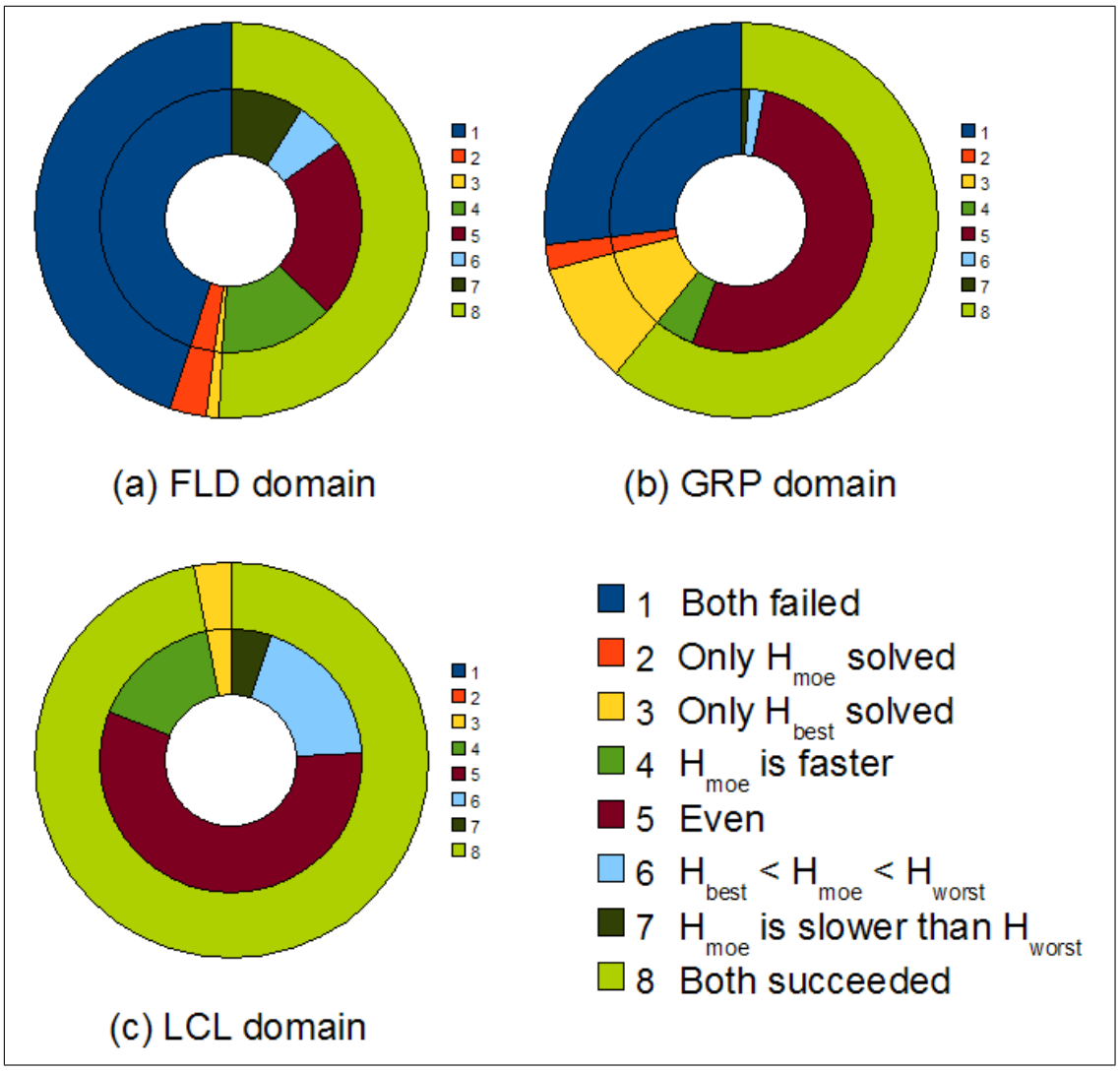


Figure 5.1. Results in pie charts

But, four per cent of the problems cannot be solved by our system, although they are solved with the combined heuristics. In addition, in 11 per cent of the problems, our proposed system finds a solution, but it is slower than H_{best}; in five per cent of the problems, it is even worse than H_{worst}. There are two possibilities for these negative results: the loss of information due to numerical representations and the similarity approach we used.

The average time spent for problems (including the problems with no solution) is given in Figure 5.3. The average time of H_{moe} is greater than H_{best} as expected. But for GRP and LCL domains, we realize that one of the heuristics has an average time less than H_{moe}. In detailed analysis, we realize that, for some problems, although one of the heuristics gives a solution quickly, our system cannot learn that, and gives a solution close to the results of

other heuristics (these problems are given as *H_{moe} is faster than H_{worst} but slower than H_{best}* in domain result tables). These individual problems should be carefully analyzed to find out the problems in the whole system.

Domain Heuristic	H_1	H_2	H_3	H_{moe}	H_{best}
FLD	106	106	106	94	91
GRP	60	93	60	71	59
LCL	93	60	71	60	59

Table 5.3. Average time for proofs in seconds

In general, the results show that the proposed system is promising. Although, for some problems, the performance of the system is behind the performance of H_{best}, constructing such perfect H_{best} heuristics is impossible and will be subject to the same problems as the problems of similarity and numerical representations. To further improve the proposed system, we should extract the individual negative results and analyze them for the similarity concept. Improvements in the similarity approach will directly affect the performance.

5.3. Analysis of used knowledge

In our learning scenario, we choose a source problem (with our similarity concept) for each target problem. But, solutions of source problems contain different number of positive and negative clauses. It is certain that if we do not have enough positive and negative clauses, learning is not possible. In the system proposed in this thesis, the lowest limit is 35 clauses (solutions with less clauses are discarded). To verify this limit we have analyzed the effect of the number of clauses on the success rate of our learned heuristic. This analysis is given in Figure 5.2 - 5.4

The figures show that our limit is sufficient for training. Above that limit, the number of clauses does not have an impact on the success rate of the learned heuristic.

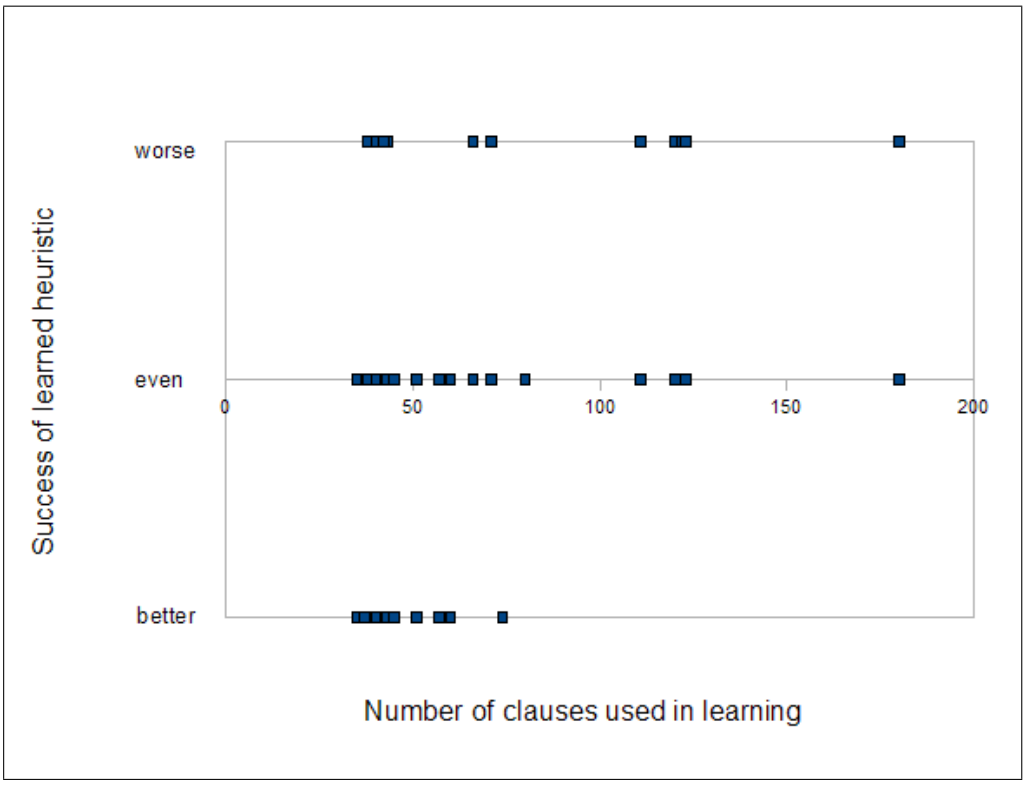


Figure 5.2. Number of clauses used in learning (FLD)

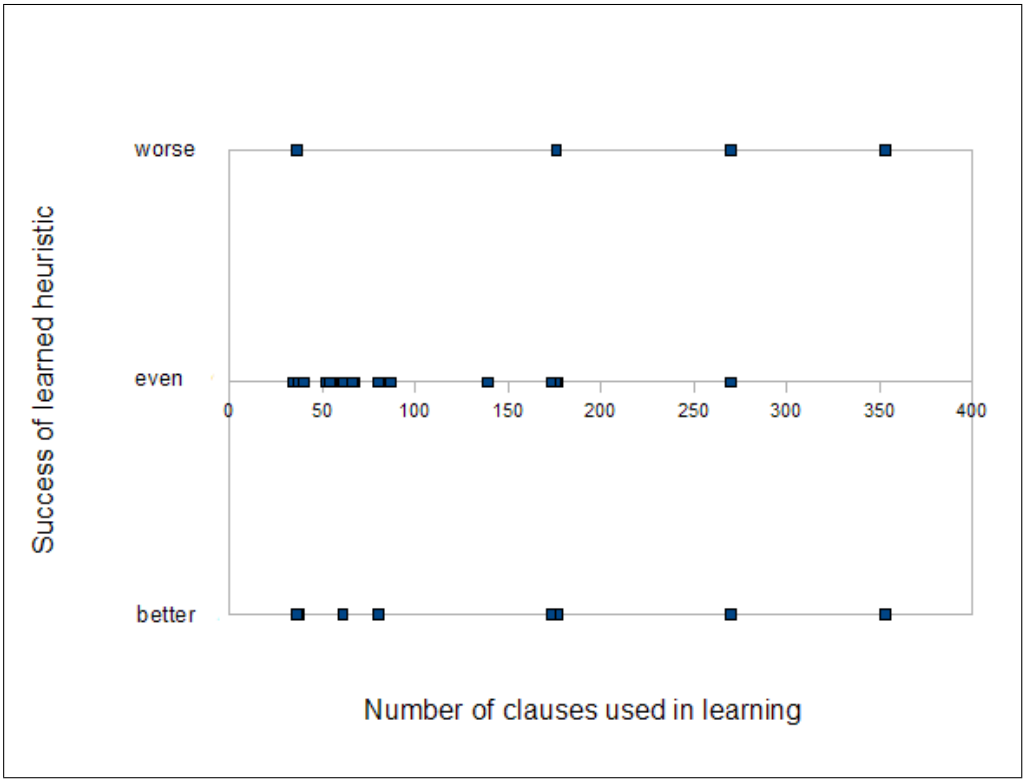


Figure 5.3. Number of clauses used in learning (GRP)

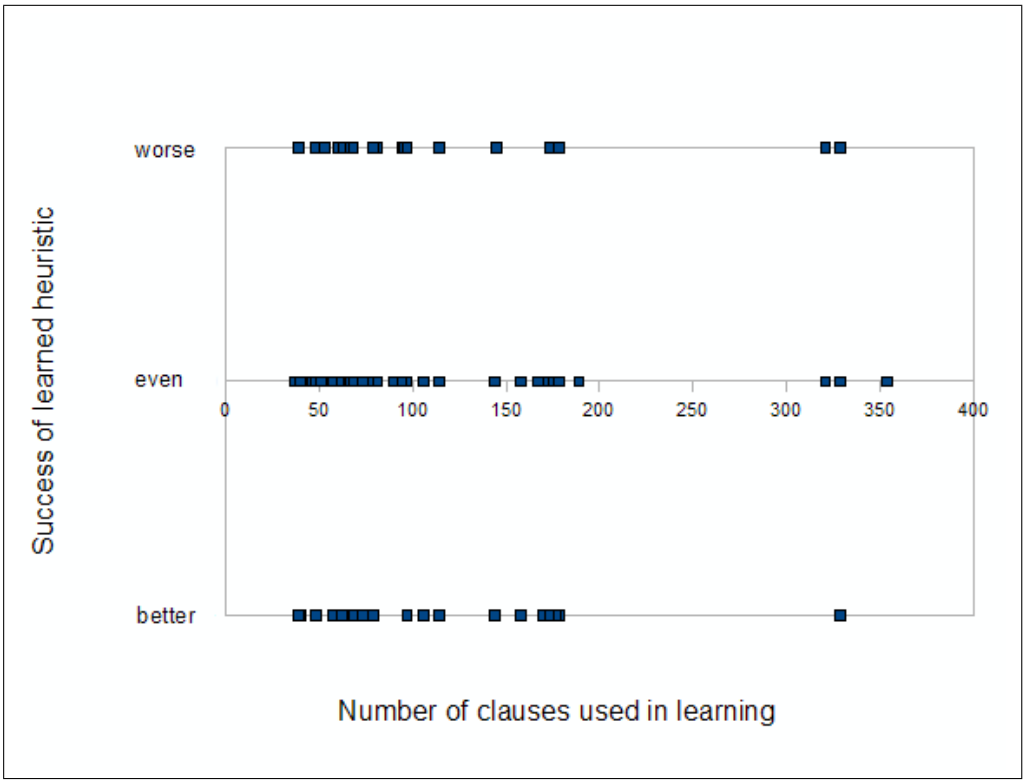


Figure 5.4. Number of clauses used in learning (LCL)

6. CONCLUSIONS AND FUTURE WORK

Although the proposed work is very promising, it has several drawbacks that have to be addressed. There are three main points that need improvement: similarity approach, experts, structure of the network.

First of all, the similarity approach needs revising. Although this concept is used in several works, it does not deal with semantic similarities between problems, but only with syntactic similarities. A more advanced similarity approach will be a valuable addition in the system. Also, our implementation simply chooses the most similar problem from the database, but does not check if the chosen problem is "similar enough". The system should have a mechanism for dealing with these problems that do not have applicable knowledge in their neighborhood in the similarity domain. For these types of problems, the system should drop to a default safe heuristic.

The experts used in the system are very primitive but they are proven to be successful for lots of problems. Other experts can be added to the system for improvement. We do not need to worry about the individual success rate of additional experts, since the gating structure will drop unsuccessful experts by evolving to a very small weight. Additional experts can be learning experts as well. The system will easily be integrated with experts which use back-propagation without any modification. Other learning methods will require a modification in the learning scheme.

Another promising approach is using symbolic representations in ANNs. This approach also can be combined with the current work. The gating structure of MOE can be modified to use symbolic representations [24, 25]. Although our network structure is the default structure of MOE, other topologies can be used in the network structure as well.

Our learning scheme uses the results of plain Otter for learning in the experiments. It does not have an incremental learning model. If we feed the system with its own outputs, we may eliminate more useless clauses with the additional information gained.

REFERENCES

1. Chang, C. and R. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press New York, 1973.

2. Alpaydin, E., *Introduction To Machine Learning*, MIT, 2004.

3. Russell, S., P. Norvig, J. Canny, J. Malik, and D. Edwards, *Artificial Intelligence: A Modern Approach*, Prentice Hall, NJ, 1995.

4. McCune, W., *OTTER 3.3 Reference Manual*, Argonne National Laboratory, Technical Memorandum No.263, 2003.

5. Denzinger, J., M. Fuchs, C. Goller, and S. Schulz, "Learning from Previous Proof Experience: A Survey", *Technical Report AR-99-4, Fakultät für Informatik der Technischen Universität München*, 1999.

6. Suttner, C. and W. Ertel, "Automatic Acquisition of Search Guiding Heuristics", *Proc. of International Conference on Automated Deduction*, pp. 470–484, 1990.

7. Goller, C., "A Connectionist Control Component for the Theorem Prover SETHEO", *Proc. ECAI94 Workshop W14: Combining Symbolic and Connectionist Processing*, pp. 88–93, 1994.

8. Fuchs, M. and M. Fuchs, "Applying Case-Based Reasoning to Automated Deduction", *Proc. of International Conference on Case-Based Reasoning*, pp. 23–32, 1997.

9. Fuchs, M., "Learning Proof Heuristics by Adapting Parameters", *Proc. of International Conference on Machine Learning*, pp. 235–243, 1995.

10. Schulz, S., "Term Space Mapping for DISCOUNT", *Proc. of CADE-15 Workshop on Using AI methods in Deduction*, 1998.

11. Schulz, S., *Learning Search Control Knowledge for Equational Deduction*, PhD Dissertation, Vol. 230, IOS Press, 2000.

12. Schulz, S., "Learning Search Control Knowledge for Equational Theorem Proving", *Proc. of KI*, 2001.

13. Schulz, S. and F. Brandt, "Using Term Space Maps to Capture Search Control Knowledge in Equational Theorem Proving", *Proc. of FLAIRS*, pp. 244–248, 1999.

14. Fuchs, M., "Feature-Based Learning of Search-Guiding Heuristics for Theorem Proving", *AI Communications*, Vol. 11, No. 3, pp. 175–189, 1998.

15. Fuchs, M., "Automatic Selection of Search-Guiding Heuristics", *Proc. of FLAIRS*, pp. 1–5, 1997.

16. Fuchs, M., "Experiments in the Heuristic Use of Past Proof Experience", *Proc. of CADE-13*, Vol. New Brunswick, LNAI, 1104, pp. 523–537, 1996.

17. Silver, B., J. Richardson, A. Smaill, A. Stevens, and A. Bundy, "Cooperating Reasoning Processes: More than just the Sum of their Parts", *Proc. of IJCAI*, pp. 2–11, 2007.

18. Duncan, H., A. Bundy, J. Levine, A. Storkey, and M. Pollet, "The Use of Data-Mining for the Automatic Formation of Tactics", *Proc. of IJCAR*, pp. 61–71, 2004.

19. Liu, Q., Y. Gao, Z. Cui, W. Yao, and Z. Chen, "An Tableau Automated Theorem Proving Method Using Logical Reinforcement Learning", *Proc. of ISICA*, Vol. LNCS, 4683, p. 262, 2007.

20. Denzinger, J. and M. Fuchs, "Goal Oriented Equational Theorem Proving Using Team Work", *Proc. of KI*, Vol. LNCS, 861, pp. 343–343, 1994.

21. Denzinger, J. and S. Schulz, "Automatic Acquisition of Search Control Knowledge from Multiple Proof Attempts", *Information and Computation*, Vol. 162, No. 1-2, pp. 59–79, 2000.

22. Denzinger, J. and S. Schulz, "Learning Domain Knowledge to Improve Theorem Proving", *Proc. of Cade-13*, 1996.

23. Sutcliffe, G., "The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0", *Journal of Automated Reasoning*, Vol. 43, No. 4, pp. 337–362, 2009.

24. Goller, C., *A Connectionist Approach for Learning Search-Control Heuristics for Automated Deduction Systems*, PhD Dissertation, Technical University of Munich, 1999.

25. Kfichler, A. and C. Goller, "Inductive Learning in Symbolic Domains Using Structure-Driven Recurrent Neural Networks", *Proc. of KI*, Vol. LNCS, 1137, pp. 183–197, 1996.