An approach to adaptive inference engine for rules-based consultation systems

An approach to adaptive inference engine for rule-based consultation systems

Von der Fakultät Ingenieurwissenschaften der
Universität Duisburg-Essen
zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation

von

Chinh Phan Cong
aus
Saigon, Vietnam

Referent: Prof. Dr.-Ing. Axel Hunger
Korreferent: Prof. Dr.-Ing. Hans-Dieter Kochs

Tag der mündlichen Prüfung: 17.08.2006

Acknowledgment

I would like to express my sincere thanks to all the people who helped me during the time that I was doing my PhD thesis.

First of all, I would like to express my deepest gratitude to my supervisor Prof. Axel Hunger for his guidance and financial support. I also thank all the members of my dissertation committee, Prof. Kochs, Prof. Krost, Prof. Kruis and Prof. Vogt, who have given me constructive inputs.

For further non-technical support, I appreciate highly all the assistances from Mrs. Elvira Laufenburg, Mr. Dipl.-Ing. Robinson, Mr. M.Sc. Sultan Zahir Khan, Mrs. Dipl.-Ing. Swetlana Khoudiakova, Mr. Bernd Holzke and Mr. Dipl.-Ing. Joachim Zumbraegel.

I am very grateful to my wife, Ha Nguyen Duy Mong for her love and patience throughout my time as a PhD student and especially my little daughter who has always been my best support.

Finally, I would like to thank my parents, Phan Ngoc Thanh and Nguyen Thi Hap, who have been sacrificing all their lives for sake of their children.

Abstract

Today, expert systems are widely used in business, science, engineering, agriculture, manufacturing, medicine, video games, and virtually every other field. In fact, it is difficult to think of a field in which expert systems are not used today (Lozano-Perez T., Kaelbling L., 2003). In the field of consultation, expert system has been applied very early, for example, MYCIN (Buchanan, B. G. and Shortliffe, E. H., 1984) can be seen one of the earliest applications of the expert system. Although there are many commercial products of expert system shells that can be applied to build consultation systems, they have shown some drawbacks:

- Accepting only one determined language for knowledge representation. Therefore, once one decided to use an expert system shell, it is not easy to make a change of it later because of having to edit the whole knowledge base in the new language again.

- Being so passive in comparison with human being. They can only do exactly what the knowledge engineer specified in the knowledge base. Therefore, it requires a lot of efforts for preparing knowledge base

- Using system resource is not optimal, when applying them to consultation systems.

To deal with the above problems, this dissertation is aimed at presenting an approach of an adaptive inference engine for rule-based consultation systems, which is a traditional inference engine with some additional abilities:

- Being able to learn different languages for knowledge representations through training.

- Being able to find out by itself which question should be asked next without any interference from human being. The "*Matching*" can recognize which information is required for the reasoning, then it creates new rules for them and adds the rules into its knowledge base.

- Being able to find out by itself an optimal reasoning strategy (forward chaining, backward chaining or a mixture of both). It does not need any setting from human being for the reasoning strategy. The "*Matching*" determines it from the beginning and at run-time.

- Being able to find out syntax errors in rules. This is the evidence that the inference engine understands its rules and understands what it is doing as well.

- Being able to learn from experience to improve its performance

Beside concepts and implementation of the adaptive inference engine, a mathematical evaluation on the effectiveness of the new matching algorithm is also presented.

Table of Contents

List of Abbreviations ....

Chapter

1. Introduction **1**
- 1.1. General consultative process 1
- 1.2. General structure of an inference engine 2
- 1.3. Drawbacks of present inference engines 4
- 1.4. A model of adaptive inference engine for consultation systems 7

2. An overview of adaptive systems **11**
- 2.1. Adaptive production systems 11
- 2.2. Adaptive agent 12
- 2.3. Adaptive interface 12

3. The basic RETE algorithm **14**
- 3.1. The RETE algorithm 14
- 3.2. Advantages of the RETE algorithm 17
- 3.3. Drawbacks of the RETE algorithm 18

4. Improvements on RETE algorithm **22**

5. The TREAT algorithm **25**

6. Comparison of RETE and TREAT **29**
- 6.1. Adding and deleting a fact 29
- 6.2. Mathematical models 29
  - 6.2.1. TREAT 31
    - 6.2.1.1. Cost of addition 31
    - 6.2.1.2. Memory cost 32
    - 6.2.1.2. Cost of deletion 32
  - 6.2.2. RETE 33
    - 6.2.2.1. Cost of addition 33
    - 6.2.2.2. Memory cost 34
    - 6.2.2.3. Cost of deletion 34

6.3. Summary 35

7. Learnable Reasoning Network Builder 37

7.1. Model of a "Learnable Reasoning Network Builder" 37
7.2. Training process 38
7.3. Creating the internal knowledge base from pattern rules 44
7.4. Diagnosis of syntax errors 50
7.4.1. Sources of syntax errors 50
7.4.2. Diagnosis 50
7.5. Extension of library 52
7.6. Implementation 54
7.5. User interface 56

8. ADINE - An new matching algorithm for rules-based consultation systems. 58

8.1. Observations 58
8.2. Components of reasoning network 59
8.3. Comparison with the RETE network 63
8.4. How do nodes work together? 64
8.5. Forwarding question request 66
8.6. Activating a Beta node 68
8.7. Joining at Memory node 68
8.8. Explanation ability 69
8.9. Optimising the speed of inference engine 70

9. Implementation of ADINE 73

9.1. Data structure of a Memory node 73
9.2. Data structure of an Alpha node 77
9.3. Data structure of a Beta node 78
9.4. Implementation 78
9.4.1. Memory class 78
9.4.2. Alpha node class 80
9.4.3. Beta node class 81
9.4.4. Building reasoning network 84

10. Evaluations on the ADINE 89

11. Mathematical Evaluation 90

10.1. Cost of addition 90
10.2. Cost of deletion 95

10.3. Memory cost 95

12. Learning from experience 99

12.1. What is experience 99

12.2. A model, which supports learning from experience..................................100

12.3. Applying learning from experience to improve the performance 103

12.4. Applying learning from experience to improve the performance 104

13. Conclusions and Future Work 107

11.1. Conclusions 107

11.2. Future Work 107

References 108

List of Abbreviations

ADINE	Adaptive Inference Enigine
AI	Artificial Intelligent
AM	Alpha Memory
BDN	Backward Distance Number
EBL	Explanation-Based Learning
FDN	Forward Distance Number
LHS	Left-Hand Side
RHS	Right-Hand Side
RNB	Reasoning Network Builder
WME	Working Memory Element
XML	Extensible Markup Language

Chapter 1

Introduction

1. General consultative process

From the user's view, a general consultative process consists of three steps:

- Step 1:

A user expresses his desire for a consultation about something. This can be simply a program call from the user, or if the program offers some different consultative services, the user has to choose one.

- Step 2:

The consultation system needs to collect necessary information from the user. Therefore, it asks the user several questions, and the user answers them.

During the conversation, the user can require some explanations from the system. There are some kinds of explanations:

- Terminology-explanation: there may be some strange words or strange concepts, which have been used in the questions and the user does not know or understand them; as a result, he needs an explanation about the meaning of them.

- Why-explanation: the user may want to know why he has to answer a question from the system or why the system needs the information.

The system should be able to give suitable explanations to the user, so that the conversation can continue.

To collect facts from the user, a consultation system normally needs rules which control the process of asking questions. We call these rules question rules. This is a set of rules which specify when to ask what, for example:

If X exists then ask Y
If Y exists then ask Z

- Step 3:

From the collected information, the system will perform the reasoning and give the user some advice.

When receiving the advice from the system, the user may not trust it immediately, and he wants to know how the system has reached the advice, that means the user needs a How-explanation. The system should persuade him of the way that the system has done.

To fulfil the tasks of this third step, the system needs rules to reason out the right advice for the user. Rules for the reasoning process are called main rules. It is the core of a consultation system. While building the knowledge base, which contains question rules and main rules, the knowledge engineer has to follow a determined syntax or language for knowledge representation, which is supported by the inference engine.

2. General structure of an inference engine

The duty of an inference engine is to perform the reasoning process which is based on a knowledge base. There are so many products of expert system shell and they are different from each other internally. However, in my opinion, an inference engine has a common general structure.

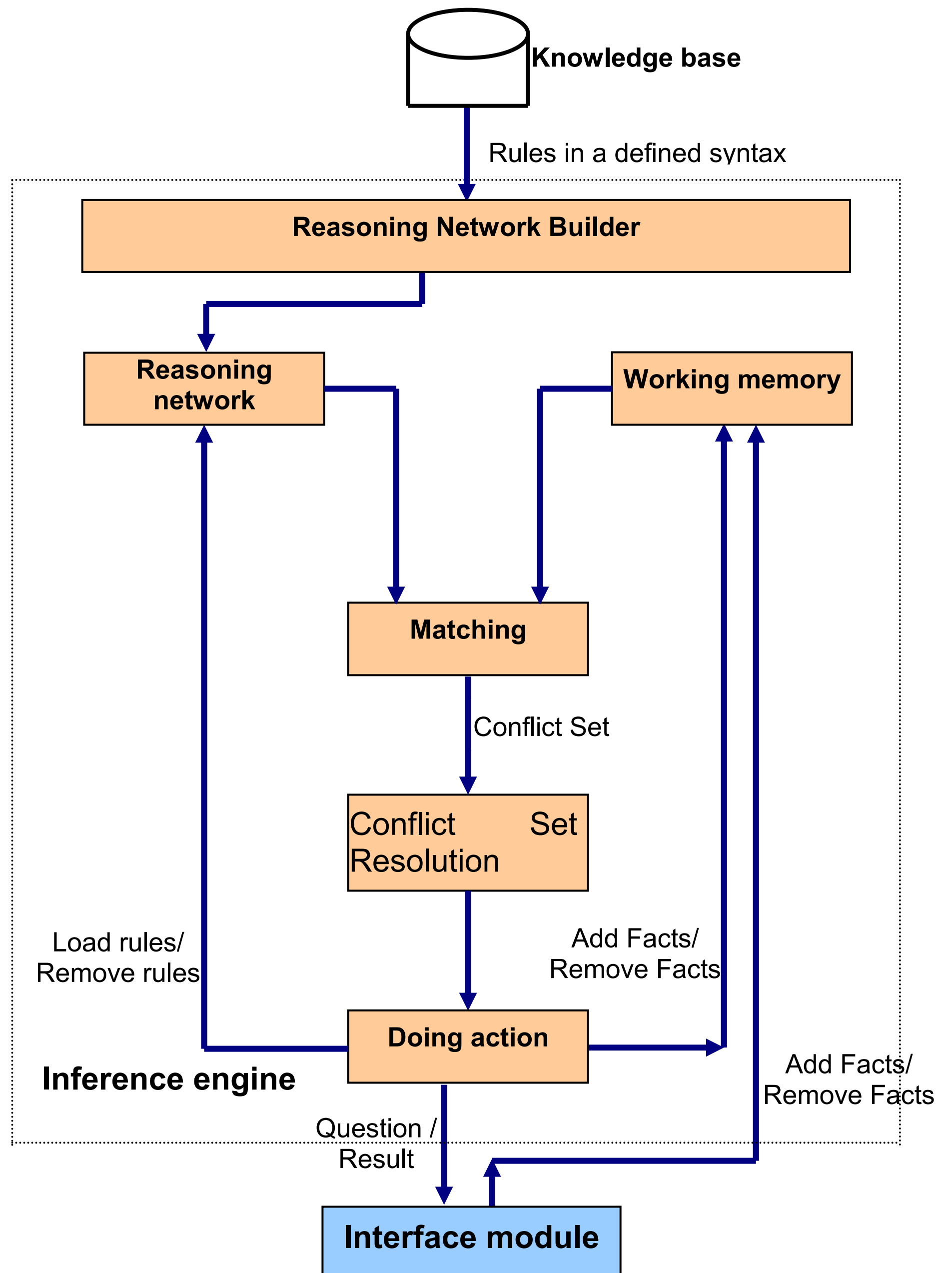


Figure 1.1. General structure of an inference engine

Rules, which are normally saved in a text file or in database, are loaded into memory space of the inference engine through the *"Reasoning Network Builder"*. The *"Reasoning Network Builder"* reads the rules at run-time and organizes them into a certain network called *"Reasoning Network"* that supports the matching. The *"Reasoning Network Builder"* can only process rules in a defined language. For

example, the *"Reasoning Network Builder"* of OPS5 cannot understand rules that are written in CLIPS language.

"Working memory" maintains initial facts and input facts, which have been given by users or which have been added by fired rules.

"Matching" performs constant tests on facts and conditions of rules. Its duty is to find out sets of facts which satisfy complete conditions of a certain rule. These sets of facts are called conflict set.

"Conflict set resolution" performs a selection among fired rules in case that there are several fired rules at the same time. Then the actions of the selected fired rule will be done by the *"Doing action"*.

There are some kinds of action such as sending a question / result to the interface module, adding or removing a fact in/from the working memory, loading or removing rules in rules base, and some actions for control purposes.

3. Drawbacks of the current inference engines

With the current expert system shells, the knowledge engineer has to set up question rules for the question process. Furthermore, the knowledge engineer has to choose a reasoning strategy: backward or forward chaining before building the knowledge base, because the syntax for knowledge representation for backward chaining is normally different from that for forward chaining. From the above reasons, it takes the knowledge engineer more time to prepare the knowledge base.

In comparison with the consultation system, a human adviser needs only the main rules (knowledge base) for the reasoning process. Basing on the main rules, he can find out the question rules by himself. The human adviser can also decide by himself which reasoning strategy (forward or backward chaining) will be suitable. Furthermore, the more customers he has serviced, the more experience he can learn and the better job he can do.

When comparing with a human being, we can see that the current expert systems are too passive. It requires the knowledge engineer to put more knowledge into the knowledge base than the knowledge that a human adviser needs to do the same task. Therefore, the costs of building and maintaining the knowledge base are high. To demonstrate that the ability to find out question rules of the system by itself will help to reduce heavy costs of building and maintaining the knowledge base considerably, let us see an example of a main knowledge base which contains only three simple rules:

Rule 1.
```
If F1 and (F2 or F3)
Then
Advice 1
```

Rule 2.
```
If F2 or F4
Then
F5
```

Rule 3.
```
If F5 and F1 and F6
Then
Advice 2
```

F1, F2, F3, F4, F5, F6 are productions and suppose that f1, f2, f3, f4, f5, f6 are the fact names which are contained in the productions correspondently.

When preparing the knowledge base, the knowledge engineer also has to set question rules for the question process. There are two alternatives

Alternative 1: setting question rules for all fact names: f1, f2, f3, f4, f6

```
If f1 not exists then ask f1
```

```
If f1 exists and f2 not exists then ask f2

If f2 exists and f3 not exists then ask f3

If f3 exists and f4 not exists then ask f4

If f4 exists and f6 not exists then ask f6
```

We do not need to set a question rule for f5, because the fact F5 will be added by the rule 2.

This alternative is simple but not optimal, because question rules may be redundant. For example, after the user has chosen an answer for f2, and the production F2 is true with this answer, the left-hand side (LHS) of the rule 2 is true, so the rule 2 is fired. Therefore, we do not need to ask a question for f4. Furthermore, there is one problem with this alternative: How can the system give an explanation of Why, if the user would like to know why he has to answer the question about f4?

Alternative 2: asking for information, only when it is necessary.

```
If f1 not exists then ask f1

If f1 exists and f2 not exists then ask f2

If F2 then ask f6

If not F2 then ask f4

If F4 and f6 not exists then ask f6
```

To build the set of question rules of the second alternative, the knowledge engineer has to understand the main rules well, and therefore, he has to take time to study at them. In this example, we have only three rules. It is easy to see that this work is more time-

consuming when the main knowledge base contains several dozens or hundreds of rules.

Another drawback of the current inference engines is that each of them is bound to a determined language for knowledge representation. The inference engine of OPS5 cannot handle CLIPS's set of rules. This limits the ability to exchange knowledge base between expert systems. Furthermore, it is not easy to make a change of another expert system, because the whole knowledge base must be edited again in the language of the new expert system.

The above drawbacks have motivated me to develop an adaptive inference engine for rule-based consultation systems, whose structure is presented in the next section.

4. A model of adaptive inference engine for consultation systems

The inference engine should have some intelligent abilities of a human being. These abilities helps the knowledge engineer to reduce the costs of maintaining knowledge base and increase the capability of exchanging knowledge bases between consultation systems. In this dissertation, we will present an adaptive inference engine for rule-based consultation systems, which is an inference engine with some additional abilities:

- Being able to learn different languages for knowledge representations through training.

- Being able to find out by itself which question should be asked next without any interference from human being. The "*Matching*" can recognize which information is required for the reasoning, then it creates new rules for them and adds the rules into its knowledge base.

- Being able to find out by itself an optimal reasoning strategy (forward chaining, backward chaining or a mixture of both). It does not need any setting from human being for the reasoning strategy. The "*Matching*" determines it from the beginning and at run-time.

- Being able to find out syntax errors in rules. This is the evidence that the inference engine understands its rules and understands what it is doing as well.

- Being able to learn from experience to improve its performance

In the following model of an adaptive inference engine, arrows in the brown color and blocks in the pink color are additional or modified components in comparison with a normal inference engine.

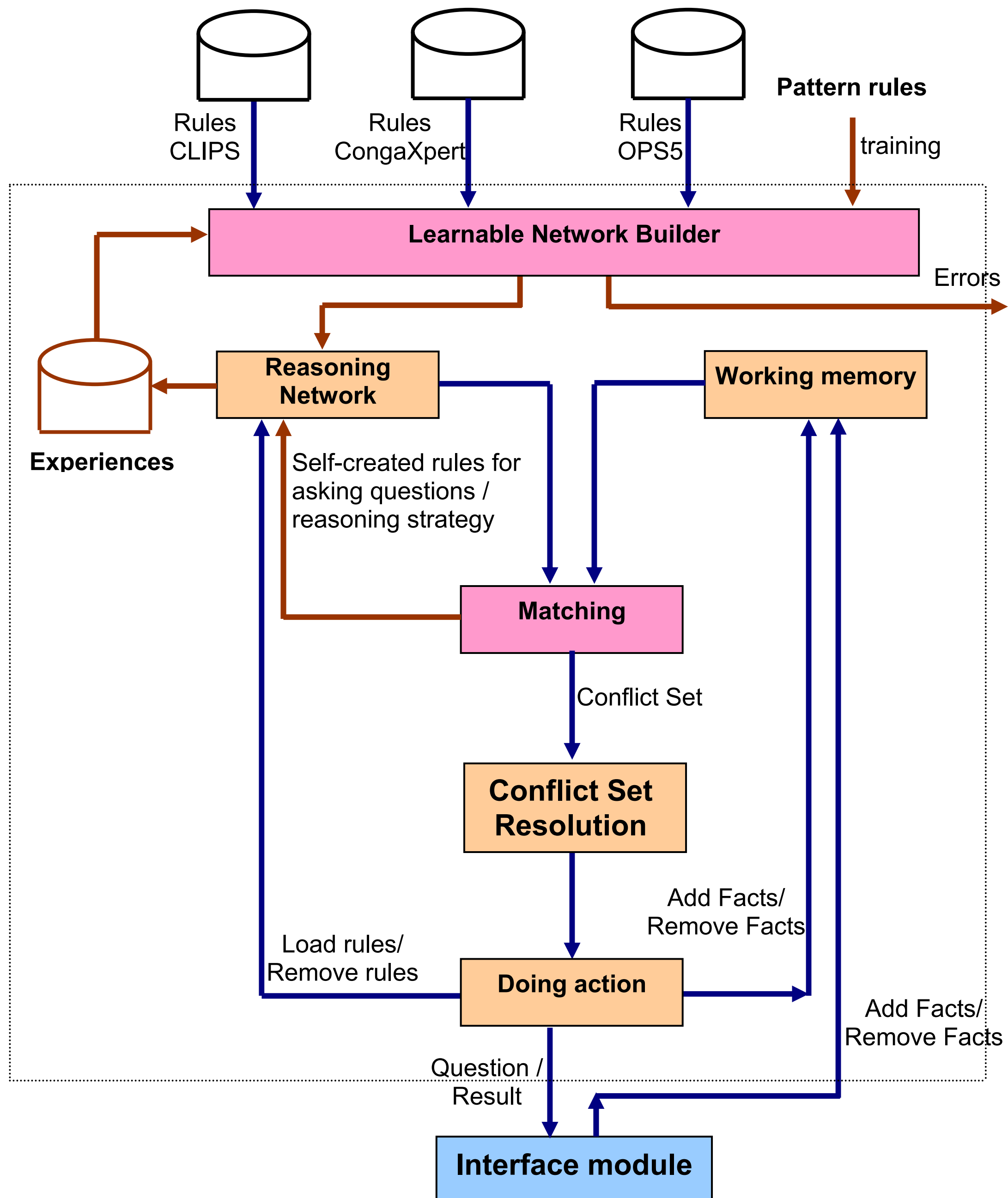


Figure 1.2. A model of adaptive inference engine

The *"Learnable Network Builder"* can handle rules represented in different languages, once it has been trained for them. The training process should be as simple as possible. One good way of training is to use pattern rules. While working, the *"Learnable Network Builder"* can also detect syntax errors. When there is any error, it will stop its work and make an error report.

While matching, the *"Matching"* can create new rules for the question process and choose a suitable reasoning strategy as well. The self-created rules for the question process and for the reasoning strategy will added into the reasoning network. The final reasoning network will be saved as its experience, so that they can be used again in future.

In chapter 2, you can find an overview about adaptive systems. Chapter 3 mentions the basic RETE algorithm, which is the best-known matching algorithm for inference engine. If you know the RETE algorithm, you can omit this chapter. In chapter 4, we discuss works on improvement of RETE. It is continued in chapter 5 with the competitive algorithm TREAT and a comparison between RETE and TREAT is considered in the chapter 6. In chapter 7, 8, 9, concepts and implementations of an adaptive inference engine (ADINE) are surveyed. Chapter 10 summarizes advantages and drawbacks of the adaptive inference engine. Chapter 11 shows a mathematical evaluation on effectiveness of the new matching algorithm. Chapter 12 presents how the system learns by experience and the last Chapter 13 gives the conclusion.

Chapter 2

An overview of adaptive systems

"*Adaptive*" has been considered an expression of intelligence. [PAT94] wrote "*Behaviour is usually considered intelligent when it can be seen to be adaptive*". In literatures of AI, a system is called adaptive, if it is able to improve its goal-achieving competence over time by learning from experiences [TOR90]. There have been so many adaptive systems all over the world. In this chapter, I do not intend to list as many research as possible here, but only try to make an overview by giving some pointers to literature, so that interested readers can make references in this field easily.

1. Adaptive production systems

Traditionally the main purpose of adaptive production systems is to optimise the system speed, when working in environments with dynamic load and resource limit.

[TOR90] introduced an organizational approach to adaptive production systems. It dealt with the load-balancing problem. To achieve adaptive real-time performance in the environment, whose load changes dynamically, an organization of distributed production systems are used, rather than a single monolithic production system. When overloaded, individual production systems decompose themselves to increase the parallelism, and when the load lightens the production systems compose with each other to free hardware resources.

[RAN96] developed an adaptive learning and planning system for transportation scheduling. An adaptive inference engine was mentioned in the sense that its performance could be improved with experience, so that more problems could be solved within a given resource limit. Multiple speedup techniques such as *bounded-overhead success and failure caching*, *explanation-based learning (EBL)*, and a new distributed computing technique called *nagging* were used. The inference engine supported combinations of these techniques to achieve the best performance.

2. Adaptive agent

The word "*adaptive*" has become popular in the literatures of artificial intelligence (AI) since around 1990, when a new wave of research "autonomous agent" emerged. Intelligent agents or autonomous agents represent a modern approach in Artificial Intelligence and deal with software entities, which can act autonomously, communicate with other agents, are goal-oriented, and are using explicit knowledge [WEI99]. They are often used for tasks, which can be hardly solved monolithically and are showing a natural distribution [JEN98]. In these domains agents gain great benefit as they are able to collaborate and solve problems in a distributed manner.

[PAT94], [NIC98] reflected on the state of the art of this new approach. They evaluated distributions of the approach, identified limitations, open issues and future challenges.

[FRA99] presented an adaptive multi-agent system architecture for manufactory. Adaptation was performed through organizational structural change and two learning mechanisms: learning from past experience and learning future agent interactions by simulating future dynamic behaviours.

[STE00] introduced LISYS – an intrusion detection system that monitors network traffic. It simulates human immune system by developing adaptive autonomous agents.

3. Adaptive interface

[MIK97] listed challenges for AI to create adaptive web sites, which can improve their organization and presentation automatically based on user access data. The research set a basis for further development of active interfaces.

[BIL00] suggested a User Modeling for Adaptive and Adaptable Software Systems. The idea is that adaptive systems monitor the user's activity pattern and automatically adjust the interface or the content of interface according to user's preferences, knowledge and skills.

In the field of E-learning, the concept of "*adaptive*" has been applied in Intelligent Tutor Systems. Macro-adaptable systems, which generally attempt to determine learner

aptitudes before the commencement of the educational process and micro-adaptable systems that diagnose different learner characteristics throughout the instruction [BRU03], [PAR03]. For example, macro-adaptable systems are based on a set of personal characteristics:

(a) the intellectual ability of the learner,
(b) the learner cognitive and learning style,
(c) the prior knowledge that the learner has, and
(d) the motivation of the learner.

In the case of micro-adaptable systems the number and the specific characteristics of the mistakes that have been made by the learner during instruction serve as the adaptation criteria. According to these mistakes, the models that perform the micro-adaptations provide for modifications that are related to:

(a) the type (specific content, level of difficulty, etc.) and the amount of multimedia content which is going to be presented to the learner in the following stage of instruction, and
(b) the presentation sequence of the content.

Chapter 3

The basic RETE algorithm

The RETE pattern-matching algorithm [FOR79], [FOR82] is the very efficient algorithm for matching facts against the patterns in rules to determine which rules have had their satisfied conditions. It has been applied in well-known expert system shells such as CLIPS, Jess, Eclipse, Frulekit, OPS5 [JOS05]. Furthermore, the algorithm has been the starting point for a lot of researches, improvements, and discussions in the area of building an inference engine. Because of the above reasons, the RETE pattern-matching algorithm is described in this chapter.

1. The RETE algorithm

On my opinion, starting with an example is a good way to explain the algorithm. Therefore, given a rule and initial working memory

Rule:

(p subject-recognition
(uni-partner (x) in [UKM, UI])
(course (x) (y))
(mark (y) (z) > 0.5)
)
=>
recognition (y)

Initial working memory
(uni-partner UKM)
(uni-partner UI)
(uni-partner TUHCM)
(course UKM, K0001)
(course UI, I0001)
(course TUHCM, H0001)
(mark K0001 0.7)
(mark I0001 0.65)
(mark H0001 0.75)

In human language, the rule can be expressed as follows: if a student, who has learned at one of the partner universities, has passed an final examination for some course, then the course will be recognized at the Duisburg-Essen university. This rule is only used as an example to illustrate the RETE algorithm. It may not have any practical meaning.

As illustrated in Figure 3.1, the algorithm RETE uses a dataflow network to represent the conditions of the productions. The network has three parts:

- The working memory elements (WMEs) are responsible to manage incoming facts and to feed them to corresponding alpha nodes. Hash tables are usually used to index WMEs. In the Figure 3.1, WMEs are the grey rectangles.

- The alpha part contains alpha nodes and alpha memories. The alpha nodes receive incoming facts from working memory elements (WMEs) and perform condition tests on them such as exists, equal, bigger than, smaller than... Which facts passed the condition test within alpha nodes, will be stored in the corresponding alpha memories (AMs).

In the Figure 3.1, alpha nodes are the yellow rectangles and alpha memories are the light yellow ones. The alpha node for the first condition makes the test, if the university partner is UKM or UI. This alpha node receives three facts from the WME uni-partner. Only two of them pass the test and are stored in the corresponding alpha memories.

- The beta part contains beta nodes (or join nodes) and beta memories. Each join node has two inputs, which can come from alpha memory nodes or beta memory nodes. Join nodes perform variable bindings between conditions. Beta memories store partial instantiations of productions, that means combinations of WMEs, which match some but not all of the conditions of a production. These partial instantiations are called tokens and will be fed to successive beta nodes.

In the Figure 3.1, beta nodes are the pink rectangles and beta memories are the light pink ones. The first beta node performs a join of **x** on its two inputs. As we see, the fact `(subject TUHCM, H0001)` was rejected after joining, because there is no fact `uni-partner(TUHCM)` in the other input of the first beta node, although the fact `(subject TUHCM, H0001)` passed the test of the alpha node `subject`.

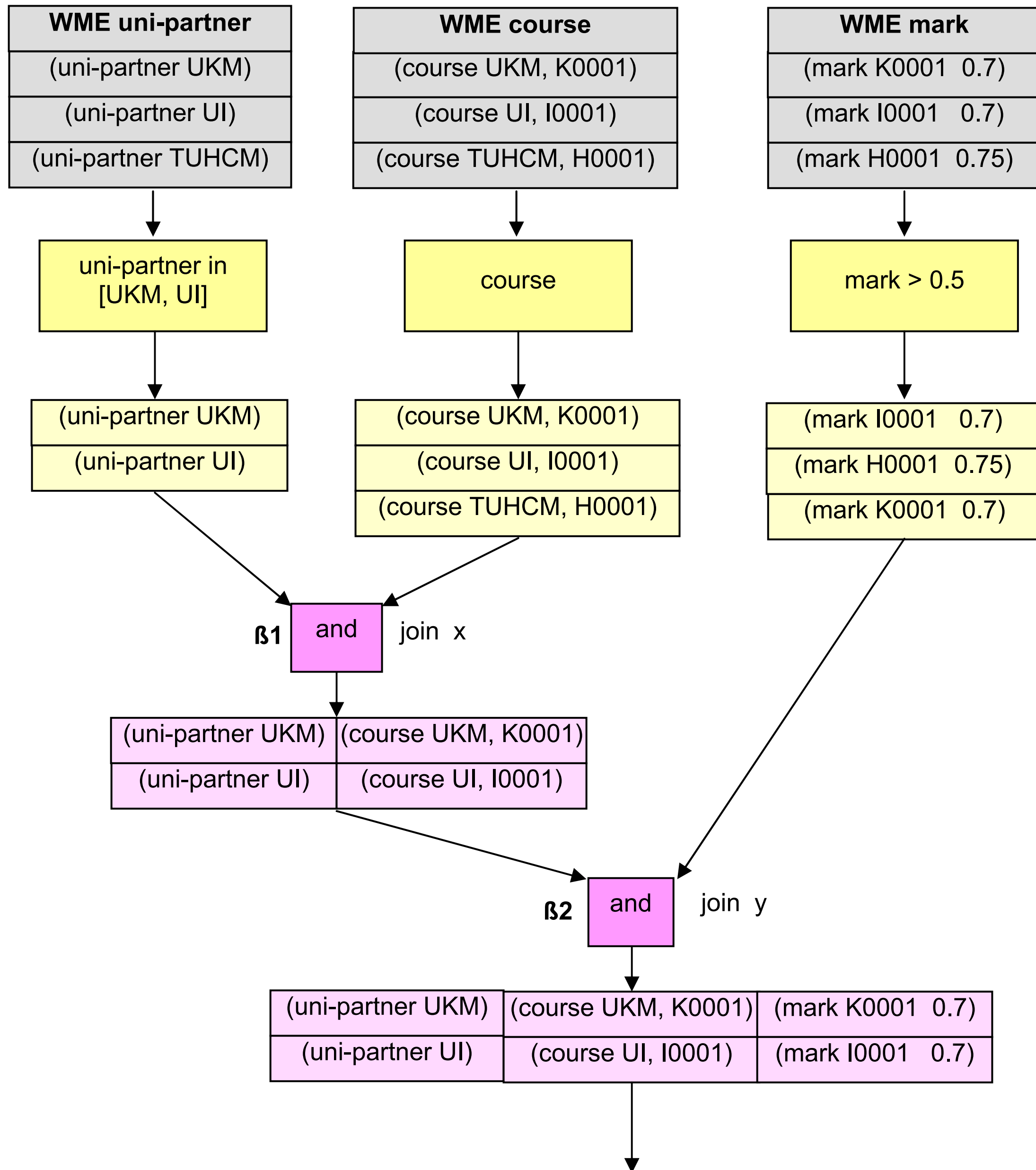


Figure 3.1: An example of RETE network for one rule (production)

Whenever a change is made to working memory elements, these changes are sent through the alpha network and the appropriate alpha memories are updated. Changes can be addition of a new fact or deletion of a fact. These updates are then propagated over to the attached join nodes, activating those nodes. If any new partial instantiations are created, they are added to the appropriate beta memories and then propagated down the beta part of the network, activating other nodes. Whenever the propagation reaches the bottom of the network, it indicates that a production's conditions are completely matched.

The RETE algorithm is not interested in what will be done further after a complete matching reached. Many productions can have complete matching of their conditions at the same time. Therefore the inference engine has to have a mechanism called *conflict resolution strategy*. It is a way of selecting a single complete production matching. Then its corresponding actions will be performed.

2. Advantages of the RETE algorithm

There are two important features of RETE that make it potentially much faster than a naive match algorithm:

- The first one is the use of state saving. The states (results) of the matching process are saved in the alpha and beta memories. Therefore, when a new fact comes, the matching process is just performed on the new fact. RETE does not have to compute on all previous facts again, because it keeps the previous matching results in its alpha and beta memories.

- The second important feature of RETE is the sharing of nodes between productions, which have got similar conditions. At the output of the alpha network, when two or more productions have a common condition, RETE uses a single alpha node for the condition, rather than creating a duplicate one for each production. Moreover, in the beta part of the network, when two or more productions have the same first few conditions, the same node is used to match those conditions. This avoids duplication of matching effort across those productions. To illustrate this point, let us see an example with two following rules:

```
If
(( A and B) and (C and (D or E)) and F)
Then
Action1

If
(A and B and F)
Then
Action2
```

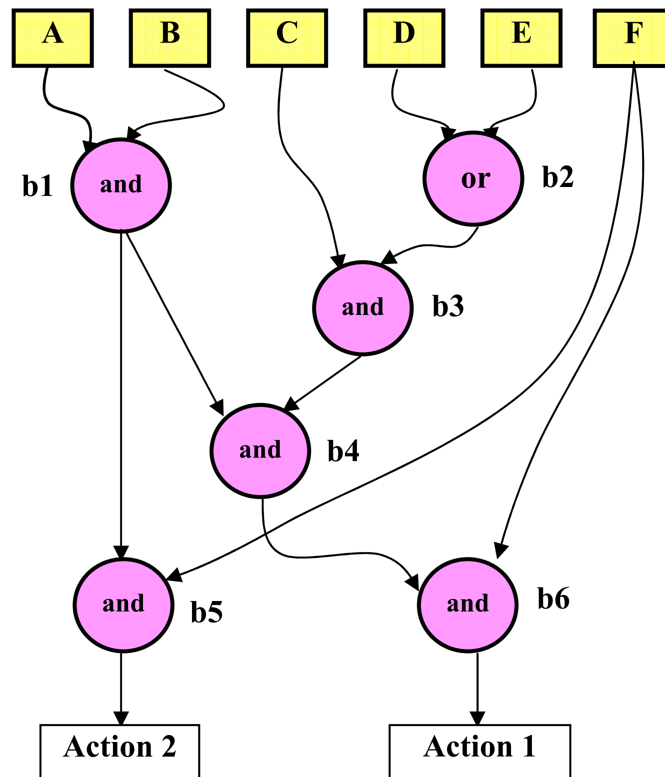


Figure 3.2: sharing of nodes between productions

A, B, C, D, E, F are conditions of rules.

In the Figure 3.2, I concentrate on alpha nodes, beta nodes and sharing of them. I suppose that alpha memories, beta memories are contained inside the alpha nodes and beta nodes. As we see, the alpha node **F** and the beta node **b1** have been shared between two rules.

3. Drawbacks of the RETE algorithm

Although the RETE algorithm has been applied most widely in rule-based systems, but it has also following disadvantages. These have been aims for many improvements, which will be presented in the next section.

- *Waste of memory*: the mechanism of state saving has got the advantage of reducing matching time for new facts. But it results in a waste of memory. For example, in the Figure 3.1 the fact `(course UKM, K0001)` is stored in the working memory, in the alpha memory and all beta memories.

- *Unsuitable for changes*: Because the system state is saved in many different nodes, it costs much time to change the state. For example, when deleting a fact, the fact has to be deleted from all corresponding working memory nodes, alpha memory nodes, and beta memory nodes. When modifying a fact, two actions will be performed: deleting the old fact and adding a new fact. Therefore, RETE is designed for systems where only a small fraction of WMEs change. RETE's state saving would not be very beneficial in systems where most WMEs change each time.

- *Utility problem*: although sharing of nodes between productions, which have got similar conditions is one of advantages of the RETE algorithm, but the system becomes slowly with this mechanism, if it has got so many rules. This problem is called *utility problem* [MIN88b]. This phenomenon happens because of *null-right und null-left activations*. *Right activation* is the activation, which comes from some alpha node. *Left activation* is the activation, which is stimulated by some beta node.

If a WME affects only one production, there is some join node that gets *right activation* when that WME is added to working memory. If a WME affects many productions, there might be only one join node that gets right activation, when the node is shared by the affected productions. In this case, we have only one right activation. But dataflow may propagate down the network from that node, causing left activations of many other nodes. And the number of such left activations can increase with the number of rules in the system. *Null-left activations* occur when a beta node propagates its token to many successive beta nodes, but only one or a few of them will get right-activation later. On the other hand, if a WME affects many productions, there might be many (unshared) join nodes, one for each affected production that gets right-activated. In this case, the number of right activations can increase linearly in the number of rules in the system. *Null-right activations* occur when an alpha node propagates its token to many successive beta nodes, but only one or a few of them will get left-activation later.

- *Unstable performance*: The performance of the RETE reasoning network depends on the order of joining conditions of productions. To illustrate to this problem, let us see an example. We use the rule at the beginning this chapter again. It is easy to see from the Figure 3.1 that

Amount of comparisons at the **ß1** : 2 x 3 = 6

Amount of occupied places at **ß1**'s memory: 4

Amount of comparisons at the **ß2** : 2 x 3 = 6

Amount of occupied places at **ß2**'s memory: 6

and now make a small change that the condition (mark (y) (z) > 0.5) is now placed before the condition (course (x) (y))

Rule:	Initial working memory
(p subject-recognition	(uni-partner UKM)
(uni-partner (x) in [UKM, UI])	(uni-partner UI)
(mark (y) (z) > 0.5)	(uni-partner TUHCM)
(course (x) (y))	(course UKM, K0001)
)	(course UI, I0001)
=>	(course TUHCM, H0001)
recognition (y)	(mark K0001 0.7)
	(mark I0001 0.65)
	(mark H0001 0.75)

We have a new network in the Figure 3.3.

Amount of comparisons at the **ß1** : 2 x 3 = 6

Amount of occupied places at **ß1**'s memory: 12

Amount of comparisons at the **ß2** : 6 x 3 + 2 x 6 = 30

Amount of occupied places at **ß2**'s memory: 6

As we see, although the final result is the same, the amount of comparisons at the ß2 is five times increased, and the amount of occupied places at ß1 is three times increased.

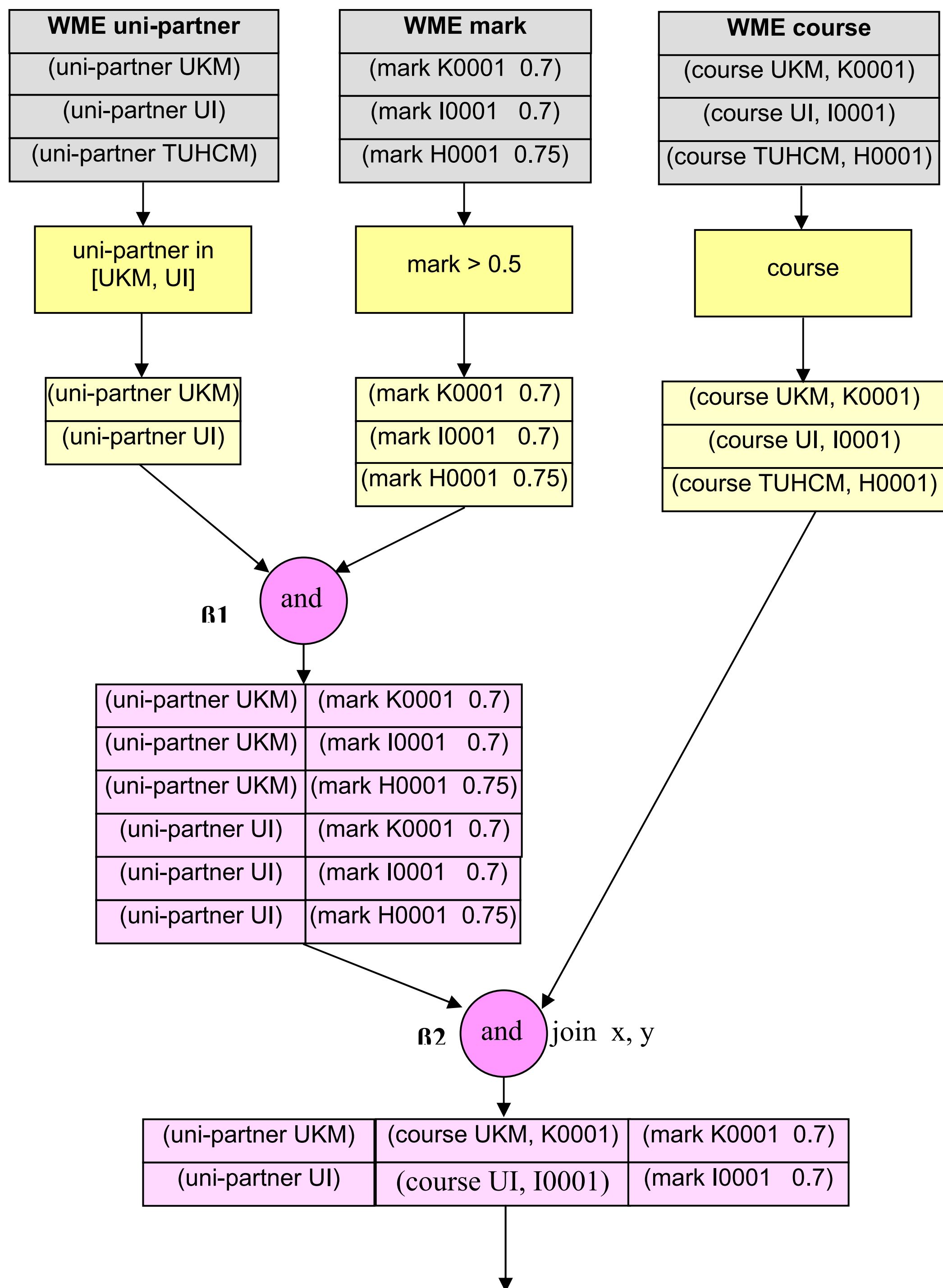


Figure 3.3 Performance depends on the order of conditions

The above drawbacks have been aims for so many researches whose overview is going to be presented in the next chapter.

Chapter 4

Improvements on RETE algorithm

In this section, we discuss researches that have improved the RETE algorithm. Most of them are related to this thesis, but the some others are also mentioned to make an overview and some pointers to the literature.

- A lot of research focused on parallel production systems to improve the overall speed of the inference engine, for example [GUP89] and [STO86]. A good overview of these works was provided in [KUO92]. The parallel algorithms can achieve faster overall speed, only when they run on a special hardware that can offer real parallel computations.

- RETE is considered *unsuitable for changes.* Because RETE saves states in many different nodes; in order to modify a fact, two actions will be performed: deleting the old fact and adding a new fact. [SCH86] extended the RETE to handle the modification of a WME directly. [PER90a] suggested a solution by marking WMEs and tokens as "inactive" instead of deleting them; when a WME is re-added later, inactive tokens are re-activated.

- [MIR 90] suggested the TREAT algorithm and it has been seen a competition to RETE. TREAT concentrates on solving the RETE's two problems: *waste of memory* and *unsuitable for changes*. Detail of the TREAT algorithm will be discussed in the next chapter.

- Some researchers used some form of conflict resolution to speed up the *Matching*, that means knowledge about the particular conflict resolution strategy is integrated into the *Matcher*. A conflict resolution strategy is a way of selecting a single complete production match and using it alone, instead of using all matches. The main idea is to change the definition of the *Matching,* so that its job is to find just only one complete match that would get selected by the conflict resolution strategy, instead of finding all

complete matches. This has been implemented in the lazy match [MIR90] and also by [TZW89].

- There are two general approaches to solve the *Utility problem*:

One approach is simply to reduce the number of rules in the system's knowledge base, via some form of selective learning or forgetting. [MAR93] provides a general framework for analyzing this approach. [MIN88a] gave an idea of discarding learned rules if they turn out to slow down the *Matching* enough to cause an overall system slowdown. Disabling the learning component after some desired or peak performance level has been reached [HOL92]. Learning only certain types of rules that are expected to have low match cost [ETZ93]. Employing statistical approaches to ensure (with high probability) that only rules that actually improve performance get added to the knowledge base [GRA92].

The second approach is to reduce the match cost of individual rules. Many techniques have been developed for this. [TAM91] prevents the formation of individual *expensive rules* that have a combinatorial match cost by restricting the representation that a system uses. Prodigy's compression module [MIN88a] reduces match cost by simplifying the conditions of rules generated by Prodigy/EBL. Static [ETZ90] and Dynamic [PER92] analyze the structure of a problem space to build much simpler rules with lower match cost for many of the same situations as Prodigy/EBL. [CHA89] generalizes or specializes the conditions of search control rules so as to reduce their match cost. A similar approach is taken by [COH90], where a few of the conditions of search control rules are dropped. [KIM93] incorporates extra “search control conditions" into learned rules in order to reduce their match cost. [ROB95] suggested adding right and left unlinking to the RETE. The basic idea is that if we know in advance that a join node's beta memory is empty, then we will arrange for right activations of that join node to be skipped; whenever the alpha memory is empty, we cut the link (the dataflow path) from the beta memory (the one on the left) to the join node.

- In systems with very large working memories, the cost of individual join operations can become expensive, due to large cross-productions being generated. [TAM90] restricted the contents of working memory by using unique-attributes. The RETE

algorithm can then be specialized to take advantage of certain properties of this restriction, yielding the Uni-RETE algorithm [TAM92]. Uni-RETE is significantly faster than RETE, but is not as general-purpose, since it requires strict adherence to the unique-attributes restriction.

- Some researches dealt with the problem that the order in which relations are joined affects performance. There are two approaches: *dynamic ordering* creates join order at the time joins are done. By *static ordering*, the join orders are arranged at the time productions are loaded into system. [SCA86] suggested a static RETE ordering algorithm; its key idea is to join condition elements that are linked to the already joined condition elements. A condition element is said to be linked to the already joined condition elements if the variable in the identifier field of the condition element is bound in the already joined condition elements. TREAT supports both of dynamic ordering and static ordering.

Chapter 5

The TREAT algorithm

The TREAT algorithm has been seen a competition to the RETE algorithm. Therefore, it is presented here for the purpose of reference. The development of the TREAT algorithm was motivated by three observations on the RETE:

- Beta-memories store the same state redundantly (e.g., see ß1 and ß2 in figure 4.1)
- The Conflict Set contains the information stored in beta-memories
- Deletion is expensive due to the need to remove state stored in beta-memories.

In order to demonstrate the TREAT algorithm, I use the example in chapter 4 again

Rule:

(p subject-recognition
(uni-partner (x) in [UKM, UI])
(course (x) (y))
(mark (y) (z) > 0.5)
)
=>
recognition (y)

Initial working memory:
(uni-partner UKM)
(uni-partner UI)
(uni-partner TUHCM)
(course UKM, K0001)
(course UI, I0001)
(course TUHCM, H0001)
(mark K0001 0.7)
(mark H0001 0.75)

Adding the new fact:
(mark I0001 0.65)

1. Adding a new fact

As in the figure 5.1, TREAT maintains Alpha memories, but does not maintain Beta-memories. Each Alpha memory consists of three partitions: Old-Partition, Add-Partition and Delete-Partition. When a fact is inserted, it is stored in the Add-Partition firstly. Then all conditions of observed rules are matched with facts in the Old-Partition and the new fact in the Add-Partition. Once the matching process is complete, the new fact is moved from the Add-Partition into the Old-Partition.

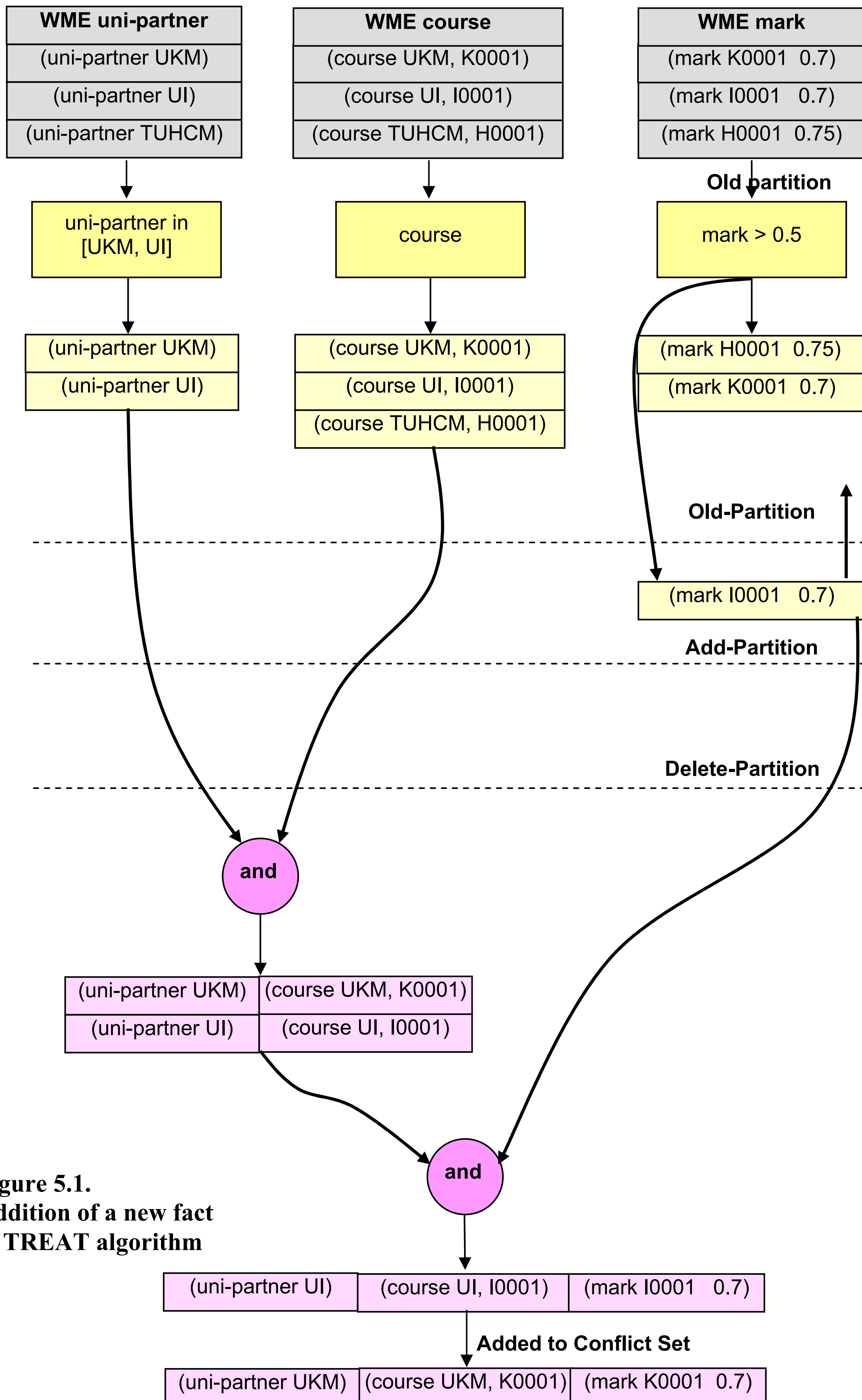


Figure 5.1.
Addition of a new fact in TREAT algorithm

2. Deleting a fact

The removed fact (Mark I0001 0.7) is inserted into the Delete-Partition. The Conflict Set is searched, so that instantiations, which contains (Mark I0001 0.7), are removed. Then (Mark I0001 0.7) is removed from the Old-Partition. At the end, (Mark I0001 0.7) is also removed from the Delete-Partition.

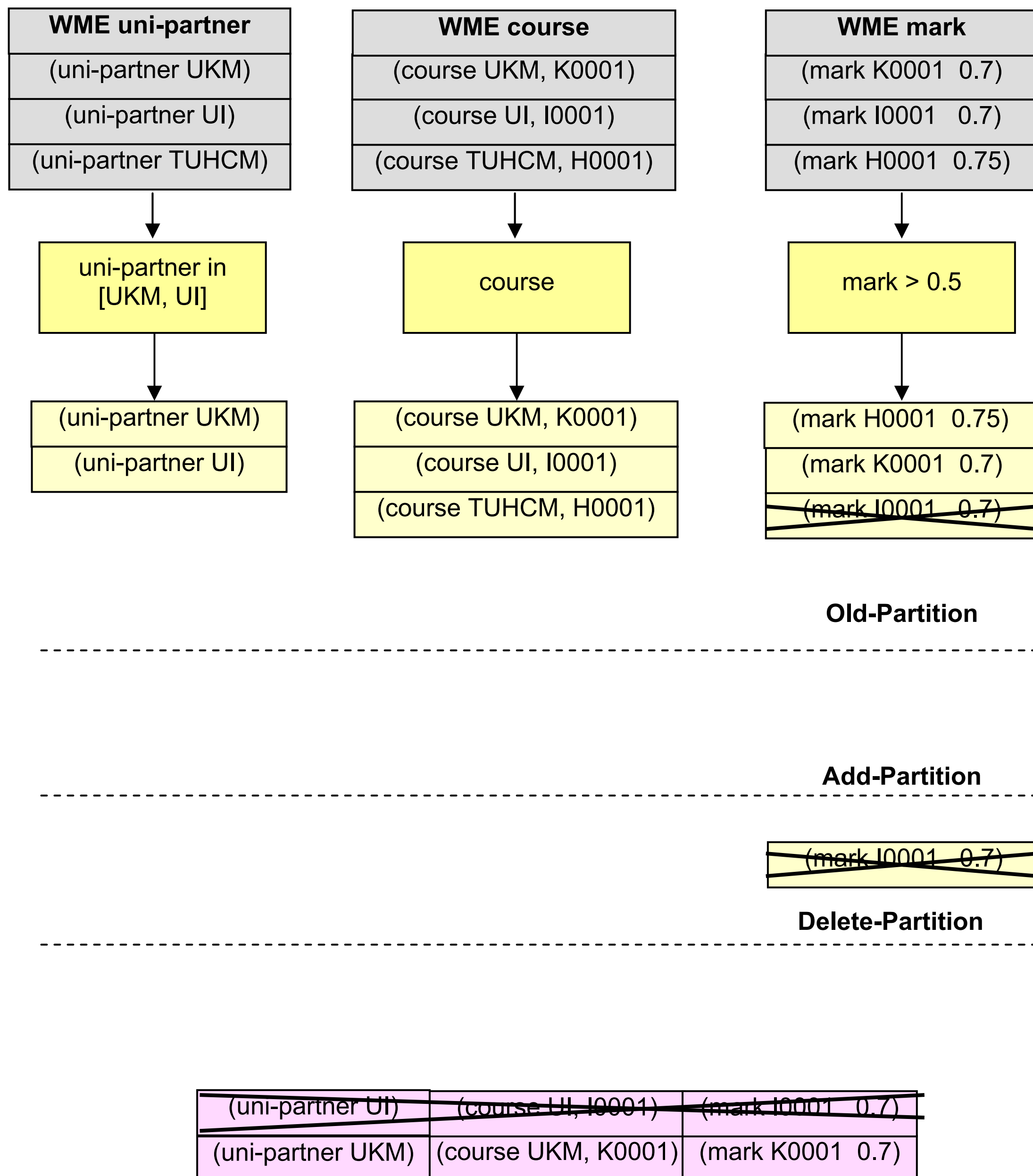


Figure 5.2. Removal of a fact in TREAT algorithm

Chapter 6

Comparison of RETE and TREAT

In this chapter, we compare performance of the two most famous matching algorithms RETE and TREAT.

1. Adding and deleting a fact

Addition and deletion of facts are two main operations in an expert system. In the following table, we make a summary of comparison between RETE and TREAT, when adding and deleting a fact.

	RETE	**TREAT**
Addition	Because previous computations are cached in Beta memories → more quickly	Full computation of all joins is required → more slowly
Deletion	The fact is removed from Alpha memory, Beta memories, and Conflict Set → more slowly	The fact is only removed from Alpha memory, and Conflict Set → more quickly

From the above table, we can conclude that the RETE is suitable for applications, whose main actions are additions of facts. On the contrary, the TREAT will contribute profit, when applications have to process a lot of operations of deleting facts.

2. Mathematical models

[MIR89] provides a simple mathematical model of join processing in order to compare RETE and TREAT processing costs. That analysis is now presented to understand the differences between RETE and TREAT. Because the modeling is a very complex task. For simplification, the following assumptions are made:

- There is no negative conditions, only positive conditions are considered

- Calculations are performed only on a single rule (i.e. join sharing between rules are ignored)

- Reasoning network is of a common form shown in figure 6.1.

- Every fact is delivered to a single Alpha node, and therefore the corresponding token is stored in one and only one Alpha memory node

- Searching a memory to delete a token costs the same as searching a memory to compare and generate a new complex token

- Costs for searching conflict set are ignored

Many of these assumptions are invalidated in practice. However, the mathematical model remains useful for approximate comparison of performances of RETE and TREAT algorithms.

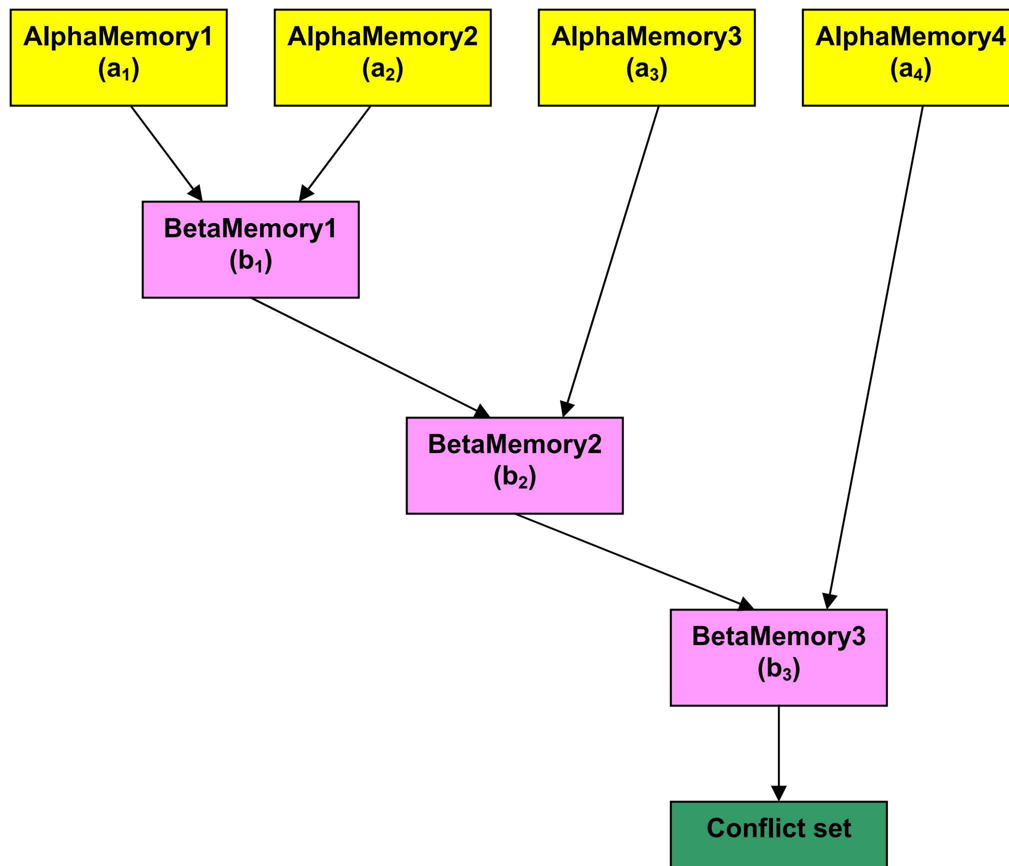


Figure 6.1: Memory nodes of reasoning network

2.1. TREAT

2.1.1. Cost of addition

Whenever a token is added into an Alpha memory, TREAT performs a full join computation on all relevant Alpha memories and Beta memories. The order, in which relations are joined, may be dynamic in TREAT. Therefore, when building mathematical expressions for join processing, we can assume a token enters at any Alpha node, for example at Alpha node 1. In this case, the token must be compared against the content of Alpha memory node 2 ($\mathbf{a_2}$), while joining (see figure 6.1).

$\mathbf{A_{T2}}$ is the number of comparisons required in a network with two Alpha memory nodes.
α_2 , α_3, α_4 , ..., α_n are sizes of nodes $\mathbf{a_2}$, $\mathbf{a_3}$, $\mathbf{a_4}$, ..., $\mathbf{a_n}$ correspondingly.

$\mathbf{A_{T2}} = \alpha_2$
where α_2 is the size of the $\mathbf{a_2}$.

If a network has three Alpha memory nodes, TREAT will join the first two Alpha memory nodes and saves the result into the Beta memory node 1 ($\mathbf{b_1}$). Then it compares the content of $\mathbf{b_1}$ against the content of $\mathbf{a_3}$. The size of $\mathbf{b_1}$ depends on the size of $\mathbf{a_1}$, $\mathbf{a_2}$, and the probability that a token from $\mathbf{a_1}$ matches a token in $\mathbf{a_2}$.

$\mathbf{p_n}$ is the probability that a token from $\mathbf{b_{n-2}}$ matches a token from $\mathbf{a_n}$ (with $n > 2$).
Therefore, the size of $\mathbf{b_1}$ is:

$\mathbf{B_1 = 1p_2\alpha_2}$

The number of comparisons in a network with three Alpha memory nodes is:
$\mathbf{A_{T3}} = A_{T2} + \mathbf{B_1\alpha_3} = \alpha_2 + \mathbf{p_2\alpha_2\alpha_3}$

the size of $\mathbf{b_2}$ is:
$\mathbf{B_2 = B_1p_3\alpha_3 = p_2\alpha_2\ p_3\alpha_3}$

The number of comparisons in a network with four Alpha memory nodes is:

$$\mathbf{A_{T4}} = \mathbf{A_{T3}} + \mathbf{B_2\alpha_4} = \mathbf{A_{T3}} + \mathbf{p_2\alpha_2\, p_3\alpha_3\alpha_4} = \mathbf{A_{T3}} + \boldsymbol{\alpha_4} \sum_{i=2}^{3} p_i \alpha_i$$

$$\mathbf{A_{T4}} = \boldsymbol{\alpha_2} + \mathbf{p_2\alpha_2\alpha_3} + \mathbf{p_2\alpha_2\, p_3\alpha_3\alpha_4} = \boldsymbol{\alpha_2} + \sum_{i=2}^{3} \alpha_{i+1} \prod_{j=2}^{j=i} p_j \alpha_j$$

By induction we have the number of comparisons for a network with **n** Alpha nodes

$$\mathbf{A_{Tn}} = \begin{cases} \alpha_2 & \mathbf{n = 2} \\ \alpha_2 + \sum_{i=2}^{n-1} \alpha_{i+1} \prod_{j=2}^{j=i} p_j \alpha_j & \mathbf{n \geq 2} \end{cases}$$

2.1.2. Memory cost

Because TREAT saves token in its Alpha memories, so memory cost of TREAT is sum of all sizes of Alpha memories. Therefore, memory cost of a network with **n** Alpha nodes:

$$\mathbf{M_{Tn}} = \sum_{i=1}^{n} \alpha_i \qquad \mathbf{n \geq 1}$$

2.1.3. Cost of deletion

When deleting a token, TREAT removes the token from the appropriate Alpha memory. That means TREAT has to compare the deleting token with tokens in the Alpha memory to find out, which the token should be deleted. Hence, the cost of deletion is approximate to the average size of an Alpha memory.

$\mathbf{D_{Tn}}$ is the cost of deleting a token from a network with **n** Alpha nodes:

$$\mathbf{D_{Tn}} = \frac{1}{n} \sum_{i=1}^{n} a_i$$

2.2. RETE

2.2.1. Cost of addition

Before we consider the number of comparison, which is required by RETE when adding a new token, let us begin with determining the size of Beta nodes

$\mathbf{B_1} = \alpha_1 \mathbf{p_2}\alpha_2$

$\mathbf{B_2} = \mathbf{B_1} \mathbf{p_3} \quad = \alpha_1 \mathbf{p_2}\alpha_2 \mathbf{p_3}\alpha_3$

In general, we have the size of the Beta node $\mathbf{B_n}$

$$\mathbf{B_n} = \alpha_1 \prod_{i=2}^{n+1} p_i \alpha_i \qquad \mathbf{n \geq 1}$$

In RETE, at which Alpha node a token enters is important for determining costs of addition or deletion. We call $A_{\mathbf{R(m,n)}}$, the number of comparisons required for a token entering the $\mathbf{m^{th}}$ alpha-memory in a network with **n** alpha-memories:

Because RETE is identical to TREAT if a token enters the network at the first or second alpha-memory, so we have

$\mathbf{A_{R(1,n)} = A_{Tn}}$

$\mathbf{A_{R(2,n)} = A_{Tn}}$

The number of comparisons required when adding a token at $\mathbf{a_3}$ is:

$$\mathbf{A_{R(3,n)} = B_1 + B_1 p_3\alpha_4 + B_1 p_3\alpha_n} \prod_{i=4}^{n-1} p_i \alpha_i$$

In general, we have

$$\mathbf{A_{R(m,n)} = B_{m-2} + B_{m-2}\, p_m\alpha_{m+1} + \ldots + B_{m-2}\, p_m\alpha_n} \prod_{i=m+1}^{n-1} p_i \alpha_i \qquad \mathbf{m \geq 1}$$

$$\mathbf{B_{m-2}} = \alpha_1 \prod_{i=2}^{m-1} p_i \alpha_i \qquad \mathbf{m > 2}$$

$$\mathbf{A_{R(m,n)}} = \frac{\alpha_1\alpha_m}{\alpha_m}\prod_{j=2}^{m-1} p_i\alpha_i + \frac{\alpha_1\alpha_{m+1}}{\alpha_m}\prod_{j=2}^{m} p_i\alpha_i + \ldots + \frac{\alpha_1\alpha_n}{\alpha_m}\prod_{j=2}^{n-1} p_i\alpha_i \qquad \mathbf{m > 2}$$

$$\mathbf{A_{R(m,n)}} = \begin{cases} \mathbf{K_n} & \mathbf{m \leq 2} \\ \sum_{i=m}^{n} \frac{\alpha_1\alpha_i}{\alpha_m}\prod_{j=2}^{i-1} p_i\alpha_i & \mathbf{m > 2} \end{cases}$$

2.2.2. Memory cost

RETE saves tokens in Alpha memories and Beta memories. Therefore, memory costs are sum of sizes of these memories:

$$\mathbf{M_{Rm}} = \sum_{i=1}^{n} \alpha_i + \sum_{i=1}^{n-1} B_i$$

$$\mathbf{M_{Rm}} = \sum_{i=1}^{n} \alpha_i + \alpha_i \sum_{i=1}^{n-1} \prod_{j=2}^{i} p_j\alpha_j$$

2.2.3. Cost of deletion

Miranker assumes that RETE deletion costs the same as RETE addition, because deletion process is an addition process with a minus token essentially.

3. Summary

Cost of adding a token	$\mathbf{A_{Tn}} = \begin{cases} \alpha_2 & \mathbf{n = 2} \\ \alpha_2 + \sum_{i=2}^{n-1} \alpha_{i+1} \prod_{j=2}^{j=i} p_j \alpha_j & \mathbf{n \geq 2} \end{cases}$
	$\mathbf{A_{R(m,n)}} = \begin{cases} \mathbf{K_n} & \mathbf{m \leq 2} \\ \sum_{i=m}^{n} \frac{\alpha_1 \alpha_i}{\alpha_m} \prod_{j=2}^{i-1} p_i \alpha_i & \mathbf{m > 2} \end{cases}$
Cost of deleting a token	$\mathbf{D_{Tn}} = \frac{1}{n} \sum_{i=1}^{n} a_i$
	$\mathbf{D_{R(m,n)}} = \begin{cases} \mathbf{K_n} & \mathbf{m \leq 2} \\ \sum_{i=m}^{n} \frac{\alpha_1 \alpha_i}{\alpha_m} \prod_{j=2}^{i-1} p_i \alpha_i & \mathbf{m > 2} \end{cases}$
Memory cost	$\mathbf{M_{Tn}} = \sum_{i=1}^{n} \alpha_i \qquad \mathbf{n \geq 1}$
	$\mathbf{M_{Rm}} = \sum_{i=1}^{n} \alpha_i + \alpha_i \sum_{i=1}^{n-1} \prod_{j=2}^{i} p_j \alpha_j$

Basing the above result, we can get some following evaluations:

- With RETE, the cost of adding a token tends to smaller, when token enters Alpha nodes behind. The reason is that RETE uses Beta memories to save previous joins and uses them later again. On the contrary, TREAT requires the same cost for each addition, because it doesn't use Beta memories to catch previous results. It calculates all joins again.

- RETE requires the deletion cost and memory code bigger than TREAT. Therefore, in expert systems, where memory is limited or there are lot deletions, TREAT is the best choice. Otherwise, RETE offers the best performance.

In the chapter 9, we would like to present our matching algorithm. An evaluation is also performed basing a mathematical model, which is presented in the chapter 11.

Chapter 7

Learnable Reasoning Network Builder

1. A model of "Learnable Reasoning Network Builder"

Duty of a "*Reasoning Network Builder*" is to create a reasoning network from rules, which are represented in a determined language (syntax). In order to service different languages for representations of rules, the "*Reasoning Network Builder*" needs to be able to learn rules' syntaxes once it is trained by pattern rules. We call such a "*Reasoning Network Builder*" the "*Learnable Reasoning Network Builder*". In other words, the "*Learnable Reasoning Network Builder*" is able to digest the training of rules' syntaxes, so that after training it can understand rules and build a reasoning network from them. A model of the "*Learnable Reasoning Network Builder*" is presented here:

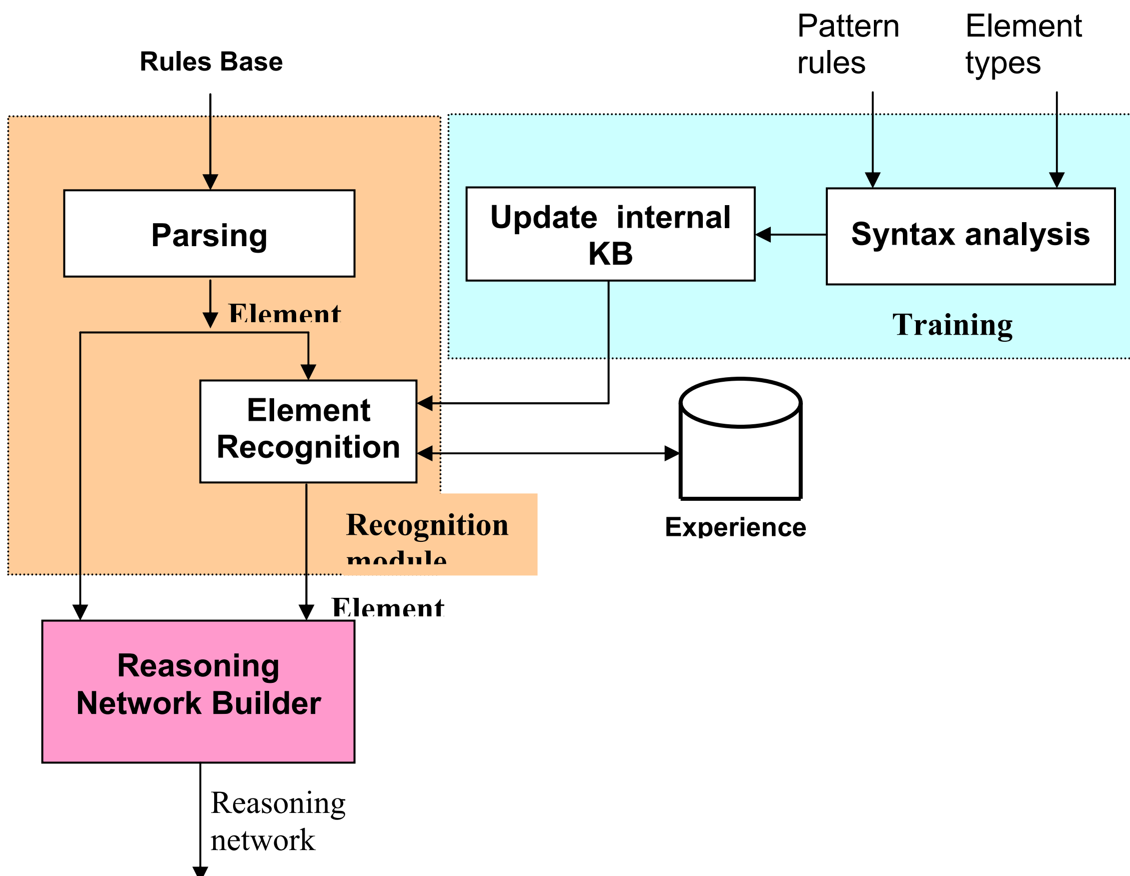


Figure 7.1. Model of a "Learnable Network Builder"

The "*Training module*" accepts trainings from outside. It receives pattern rules, analyses them, and learns their syntaxes. The knowledge of rules' syntaxes is updated into an internal knowledge base while training. The internal knowledge base will be used by the "*Recognition module*" to recognize incoming rules' syntaxes.

Normally rules are saved in the text format. The "*Parsing*" reads the text of every rule, breaks it into elements and sends them to the "*Reasoning Network Builder*". In order to build a reasoning network, the "*Reasoning Network Builder*" needs not only these elements, but also the meaning of them. Therefore, the "*Element Recognition*" is required. The duty of the "*Element Recognition*" is to recognize the types of the elements. In order to perform this duty, the "*Element Recognition*" needs to have the knowledge of rules' syntaxes, which is provided by the "*Training module*". The "*Element Recognition*" bases on this internal knowledge base to fulfill its duty. One important characteristic of the "*Element Recognition*" is that it saves its internal knowledge base as experience for future using. Whenever the system is started, the internal knowledge base is loaded again from the saved experience. In the next section, we discuss about the training process in detail.

In order to increase the speed, the "*Element Recognition*" manages its internal knowledge base according to languages that it has learned. It receives the information about the language from the "*Parsing*", and uses only the part of internal knowledge base for this language when processing the rules.

2. Training process

For the training process, several element types are defined:

Factname

Attribute

Attribute-Value

Variable

Operator

Comparison-Value

Constant

In the training process, one has to specify at least a type for each element in the left-hand side (LHS) of the pattern rule. Then the syntax of the LHS will be learned through creating and updating an internal knowledge base.

For a demonstration, let's see an example with a rule in CLIPS format. This rule is used as a pattern rule for the training process.

```
(defrule income-above-threshold
  (user (name Tom)(income ?i&:(>= ?i 50000)))
  =>
  (assert (result (text "Income above threshold")))
```

The "*Syntax Analysis*" will parse the rule and break it into elements. Then it collects training information about types of elements, which are given by the trainer as follows:

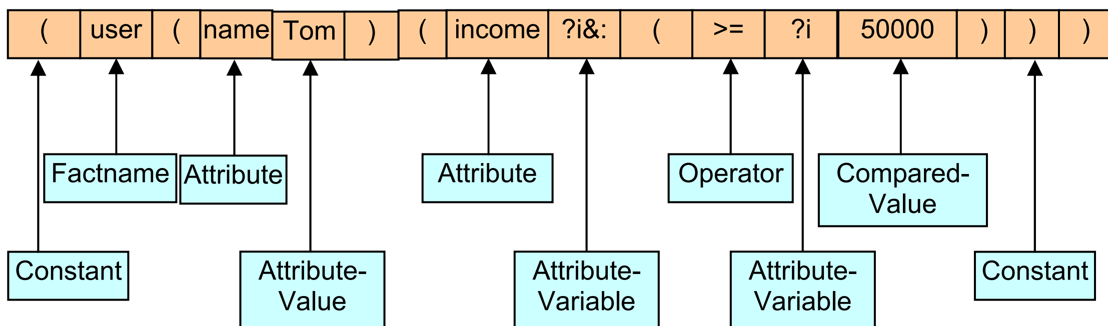


Figure 7.2. Training through specifying the types of elements of rule (rule in CLIPS format)

From the elements and element types, the following rules will be created by the "*Update internal KB*":

Rule 1:

If

The left element of ***X*** is "(" and the left element of **X** is the first element

Then

X has got the type of *Factname*

Rule 2:

If

The left element of ***X*** is "(" and the previous element of the left element of **X** has got the type of *Factname*

Then

X has got the type of *Attribute*

Rule 3:

If

The left element of ***X*** is "(" and the previous element of the left element of **X** is ")"

Then

X has got the type of *Attribute*

Rule 4:

If

The left element of ***X*** has got the type of *Attribute* and the previous element of the left element of **X** is "("

Then

X has got the type of *Attribute-Value*

Rule 5:

If

The left element of ***X*** has got the type of *Operator* and the previous element of the left element of **X** is "("

Then

X has got the type of *Attribute-Value*

Rule 6:

If

The left element of ***X*** is "(" and the previous element of the left element of **X** has got the type of *Attribute-Variable*

Then

X has got the type of *Operator*

Rule 7:

If

The left element of ***X*** has got the type of *Attribute-Value* and the previous element of the left element of **X** has got the type of *Operator*

Then

X has got the type of *Compared-Value*

The above rules are stored in an internal reasoning network, which is presented in the figure 7.4.

Now we give a further training by the same rule but in OPS5 format:

```
(defrule income-above-threshold
  (user ^name  Tom  ^income >= 50000 ^alt 30)
  =>
  (make result ^text "Income above threshold")
```

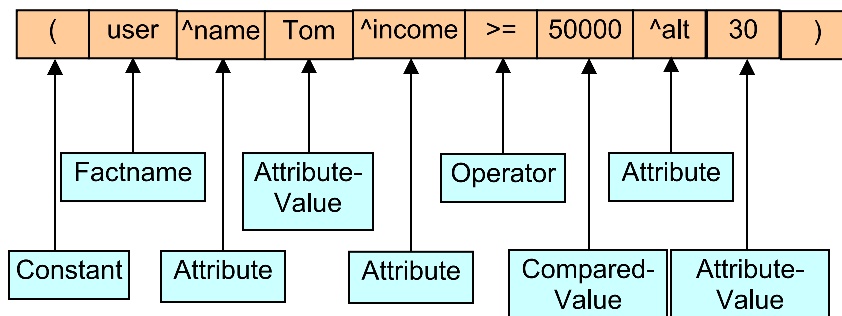


Figure 7.3. Training by specifying the types of elements of rule (rule in OPS5 format)

The following rules will be created from the training process.

Rule 1:

If

The left element of ***X*** is “(“

Then

X has got the type of *Factname*

Rule 2:

If

The left element of ***X*** has got the type of *Fachname*

Then

X has got the type of *Attribute*

Rule 3:

If

The left element of ***X*** has got the type of *Attribute* and ***X*** is not in [>, >=, <, <=, =, <>]

Then

X has got the type of *Attribute-Value*

Rule 4:

If

X is in [>, >=, <, <=, =, <>]

Then

X has got the type of *Operator*

Rule 5:

If

The left element of ***X*** has got the type of *Operator*

Then

X has got the type of *Compared-Value*

Rule 6:

If

The left element of ***X*** has got the type of Compared-Value

Then

X has got the type of *Attribute*

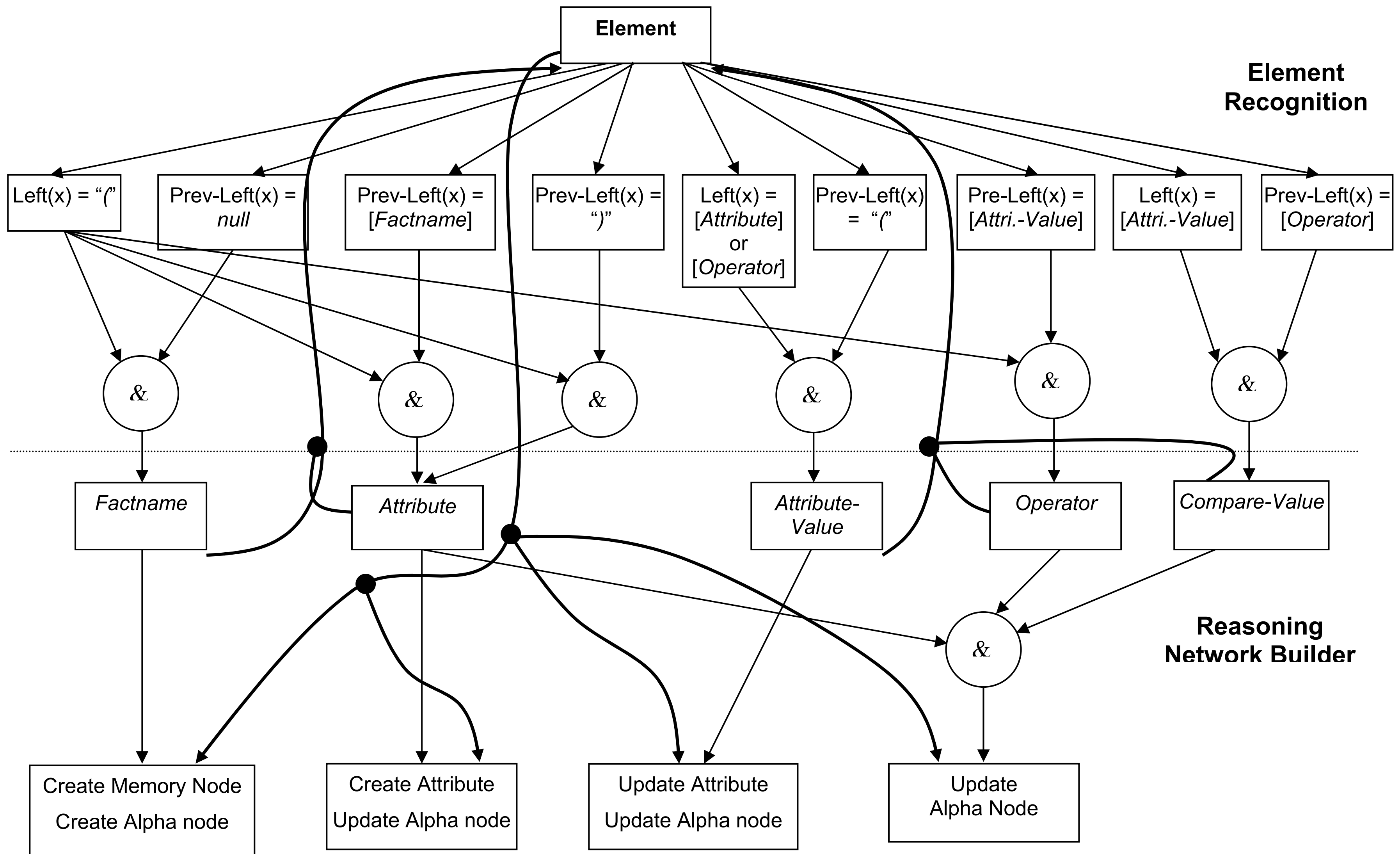


Fig. 7.4: Internal reasoning network created from pattern training rules

The internal reasoning network is built from the patterned rules as in the figure 7.4. It consists of two layers: "*Element Recognition*" and "*Reasoning Network Builder*". The second layer is fixed, but the first layer can be changed according to training rules or pattern rules. The internal reasoning network has got some specific characteristics which are different from the RETE network:

- There is only one Memory node "*Element*" in the first layer "*Element Recognition*". The "*Element*" has got two arrays: the first one contains elements of a rule's LHS which are fed to Alpha nodes. According to which language is used, the Memory node will send elements to corresponding Alpha nodes. This way helps to reduce working time and to avoid errors from language coherences. The second array contains types of elements which correspond to elements in the first array. The second array will be filled by Memory nodes of the second layer "*Reasoning Network Builder*".

- There is no Alpha memory node, because it is not necessary to save elements. When an element satisfies the internal condition of an Alpha node, the Alpha node just sends a token to its Beta nodes, and then it forgets the element.

- Beta nodes of the first layer are connected with Memory nodes of the "*Reasoning Network Builder*" layer. Each Beta node has got a temporary memory. After it received enough tokens, it sends a joined token to its Memory node, after that it removes tokens from its temporary memory. Beta memory nodes do not save the joined tokens.

- After receiving a token, Memory nodes of the "*Reasoning Network Builder*" layer save element types back into the second array of the "*Element*" of the first layer "*Element Recognition*". These Memory nodes have got additional functions for diagnosis of syntax errors which will be presented in the next section 4.

3. Creating the internal knowledge base from pattern rules

From the two above examples, we see that the advantage of the "*Learnable Reasoning Network Builder*" is the ability of creating the internal knowledge base from the pattern rules. In this part, we will explain principles how the internal knowledge base is created.

Basing on the two following observations:

- The locations of elements in the left-hand side (LHS) parts of rules are normally arranged in a determined order that expresses the syntax of the LHS.

- Some elements contain special characters for the recognition purpose. For example, in CLIPS an attribute variable starts with "?", and in OPS5 an attribute starts with "^".

The process of creating the internal knowledge base is as follows:

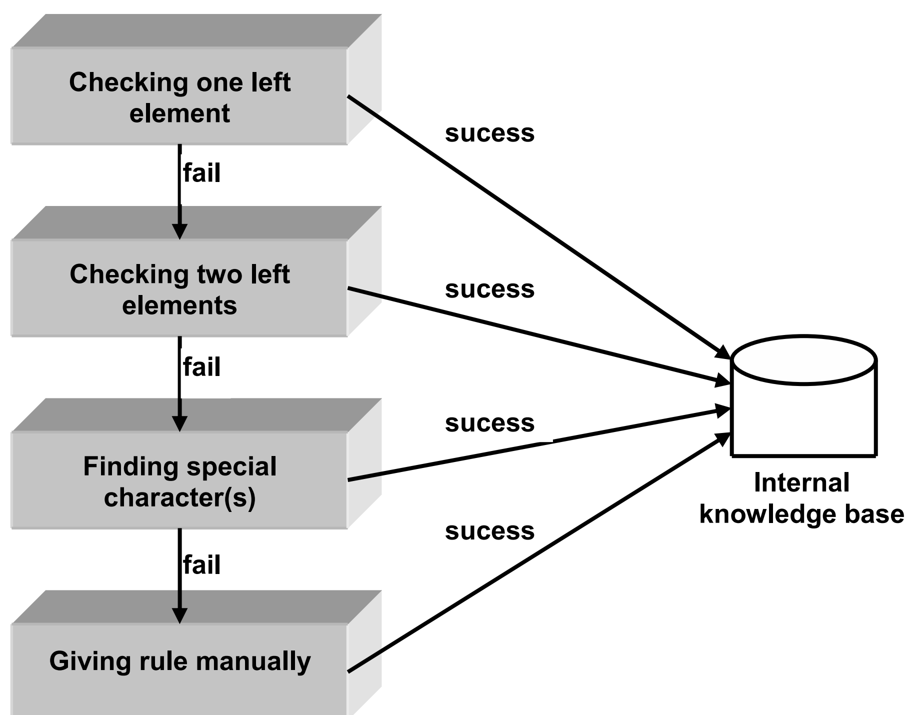


Fig. 7.5: Process of creating the internal knowledge base

Recognition process is performed from left to right of the LHS part. Elements on the left of the observing element have been determined. Elements of the right of the observing element are still unknown. Therefore, only the left elements of the observing element can be used to identify the type of the observing element.

There are 4 steps: "*Checking one left element*", "*Checking two left elements*", "*Finding special character(s)*", "*Giving rule manually*". If a step has been performed successfully, then new rules can be created, and the internal knowledge base will be

updated. If a step failed, the next step will be performed. Now let us see each step in detail.

Checking one left element:

For each observing element, check up all other elements in the LHS of the pattern rule if there is any element whose type is different to the type of the observing element, but the left element of it has got the same type as the type of the observing element. If there is one, the left element of the observing element cannot be used to identify the type of the observing element. Otherwise, a rule is created which bases on the type of the left element to identify the type of the observing element. The following table is used to demonstrate this idea.

	Left(FN)	Left(Attr.)	Left(Attr.-V)	Left(Op)	Left(Comp-V)
Factname(FN)		1			
Attribute(Attr)			**2**	**1**	
Attr.-Value (Atrr.-V)		1			
Operator(Op)					1
Compared-Value (Comp-V)		1			
Constant(Const)	1				

Tab. 7.1: Checking one left element using the example corresponding to the fig. 7.3

Information in the above table is extracted from the pattern rule. For example:

The left element of a Factname-element is a Constant-element (the column **Left(FN)**).

The left element of an Attribute-element can be a Factname-element or an Attribute-Value element or a Compared-Value element (the column **Left(Attr.)**).

The left element of a Compared-Value-element is an Operator-element (the column **Left(Comp-V)**).

The above information can be used to build internal knowledge base, because there is no element whose type is different to the type of the observing element, but its left element has got the same type as the left element of the observing element. Rules can be created from the above information as follows:

Rule 1:
If
The left element of ***X*** is “(“
Then
X has got the type of *Factname*

Rule 2:
If
The left element of ***X*** has got the type of *Fachname*
Then
X has got the type of *Attribute*

Rule 3:
If
The left element of ***X*** has got the type of *Attribute-Value*
Then
X has got the type of *Attribute*

Rule 4:
If
The left element of ***X*** has got the type of *Compared-Value*
Then
X has got the type of *Attribute*

Rule 5:
If
The left element of ***X*** has got the type of *Operator*
Then

But the following information cannot be used to identify the type of the observed element:

Left element of an Attribute-Value element is an Attribute element (the column **Left(Attr.-V)**).

Left element of an Operator element is an Attribute element (the column **Left(Op)**).

Because there are two different elements whose left elements have the same type of *Attribute*. In this case, the second step will be performed further.

Checking two left elements:

The second step does the same as the first step but it bases on two left elements of the observing element. For the above example, now observing elements are elements, which have got the types of *Attribute-Value* and *Operation*.

	Pre-Left(Attr.-V)	Left(Attr.-V)	Pre-Left(Op)	Left(Op)
Factname(FN)	1			
Attribute(Attr)		**2**		**1**
Attr.-Value (Atrr.-V)				
Operator(Op)				
Compared-Value (Comp-V)	**1**		**1**	
Constant(Const)				

Tab. 7.2: Checking two left elements using the same example as Tab. 7.1

From the table we can get out only one rule for the observing element type of *Attribute-Value*:

If

The left element of ***X*** has got the type of *Attribute* and the previous element of the left element has got the type of *Factname*
Then
X has got the type of *Attribute-Value*

But this rule is not general enough. It becomes incorrect if a comparison is performed on the first attribute. So it can be concluded that the training process is not enough, when using only one pattern rule. The system needs more training to differentiate the type of *Attribute-Value* against the type of *Operator*.

Finding special character(s):
The "*Learnable Reasoning Network Builder*" is able to draw rules for element recognition from special characters. It bases on the principle that the beginning or ending of an element may contain specific characters. After several observations, conclusions can be drawn. For example, the following rule can be found out from pattern rules basing on the specific character "^":

If
X starts with ^
Then
X has got the type of *Attribute*

Giving rule manually:
The trainer can give in rules directly. That is the reason why the system can own these rules:

If
X is in [>, >=, <, <=, =, <>]
Then
X has got type of *Operator*

If

The left element of ***X*** has got the type of *Attribute* and ***X*** is not in [>, >=, <, <=, =, <>]
Then
X has got type of *Attribute-Value*

4. Diagnosis of syntax errors

4.1. Sources of syntax errors

Syntax errors can come from several error sources:

(1) The system has not been trained about the syntax of the rule. Therefore the system cannot understand it and the system thinks that there are syntax errors.

(2) Syntax errors in conditions of rules' LHS. They are normally the knowledge engineer's mistakes. For example, a comparison lacks an operator or a compared value.

(3) Syntax errors in conjunction part between conditions within the LHS. This kind of errors can happen with rules that have got complicated LHS. For example, let's see a rule with a complicated LHS:

If
(A and (B or C) or ((D and E) or (F and G)))
Then
….
A, B, C, D, E, F are conditions.

In such a complex rule, brackets play important roles in specifying how to evaluate the LHS expression. The knowledge engineer could make some mistake while editing the rule. He could omit some brackets or logic operators (and, or) and his mistakes result in syntax errors.

4.2. Diagnosis

Basic idea is that using the reasoning network as an effective tool for diagnosis of syntax errors. This idea can be realized by adding new diagnosis functions into nodes

(Memory nodes, Alpha nodes, Beta nodes) in the reasoning network. These nodes and their co-operations will perform the diagnosis while working.

- In order to cope with the syntax error of type (1) (see the section 4.1), the Memory node of the first layer "*Element Recognition*" is equipped a new function as follows:

After feeding an observed element to Alpha nodes, it checks its second array which contains the types of elements. If it doesn't receive a type for the observing element, it can conclude that the reasoning network of "*Element Recognition*" cannot recognize the observed element, or the syntax of the rule has not been trained. The Memory node can perform the check on several elements to make sure the conclusion is correct.

- The second type of syntax errors comes from the knowledge engineer's mistakes. The nature of the "*Element Recognition*" is to base on left elements to recognize the type of the observing element. But this way works well only under an assumption that elements are in a correct order or the syntax is correct.

What happens if there are syntax errors? For example, normally an element of type *Attribute* stands on the right of the element of type *Factname*. But while editing a rule, because of a mistake, the knowledge engineer omitted the element of type *Attribute*. Now the element of type *Attribute-Value* stands directly on the right of the element of type *Factname*. In this case, the element of type *Attribute-Value* will be recognized as the one of type *Attribute*, because its left element has got the determined type of *Fachname.*

This problem has been solved when we set some checkpoints in the Memory nodes of the second layer "*Reasoning Network Builder*". These Memory nodes try to learn characteristics of their types, for example:

- In OPS5 elements with the type of *Attribute* start with the letter "^"
- In CLIPS elements with the type of Variable start with the letter "?"
- Elements with the type of *Operator* can be among [>, >=, <, <=, =, <>]

When these Memory nodes receive a token, they will check if this token is believable basing on their experiences. If it is believable, it sends the token further. If not, it sends out an error message.

- The solution for the syntax errors of the type (3) bases on the observation that this type of errors causes Alpha nodes or Beta nodes to lose connections with other Beta nodes within its rule. Therefore we put some check functions into Alpha nodes and Beta nodes as follows

- If an Alpha node doesn't have any connection with a Beta node, which belongs to the same rule as the Alpha node, then it sends out an error message.

- If a Beta node doesn't have any connection with a Beta node or an Action node, which belongs to the same rule as the Beta node, then it sends out an error message.

5. Extension of library

Although the adaptive inference engine can learn the syntax of rules but how about functions, which are called from the rules' right-hand side (RHS) and which perform comparative operations in the condition part of the LHS. Existing languages for knowledge representation provide sets of defined functions in theirs library. Besides that they offer users the possibility to define their own functions used in rules' RHS. On my opinion, the way, which existing languages have been following, has got several drawbacks:

- It doesn't always meet users' needs entirely. For example, some user may need some special comparative operators for conditions in the LHS of his knowledge base, and these operators may be not provided by existing language.

- It is redundant because it provides too many functions, but not all users can take full advantages of them.

- Although users can define their own functions for rules' RHS in their knowledge base, but because of limitations of languages, users cannot do any thing that they want.

For example, database connection, communication over network. Furthermore, the user-defined functions are compiled at run-time. So system's performance will be affected negatively when using many user-defined functions of this way.

The above drawbacks motivated me to present a solution as follows:

- A library, which contains common functions for comparative operators and actions, is provided as usual.

- Users can use Java language to develop their own library containing necessary functions and integrate them into system easily. By this way users can exploit all advantages of the Java language to extend the system's library. The system's library has been also developed with Java, therefore users can import and use it when programming.

To provide users with this possibility, we defined two standard interfaces and users have to use these interfaces when developing their own library.

```
public interface Operator {
  public boolean Checking(Object[] arg);
}
```

```
public interface Action {
  public boolean DoingAction(Object[] arg);
}
```

For example, a user wants to develop an operator "`equal`", which checks equality of two input parameters of String. The class `equal.java` looks like as follows:

```
public class equal implements Operator {
  public boolean Checking(Object[] arg){
     String param1 = (String)arg[0];
     String param2 = (String)arg[1];
```

```
    return param1.equals(param2);
  }
}
```

Now the inference engine can call this class by using the Java's function `loadClass`:

```
Try
{
    Operator op =
    Operator)this.getClass().getClassLoader().loadClass("equal").newInstance();
  arg[0] = (String)"test";
  arg[1] = (String)"test";
  result = op.Checking(arg);
}
catch(IllegalAccessException ille){
        System.out.print("Illegal Exception");
}
catch(InstantiationException ille){
        System.out.print("Instantiation Exception");
}
catch(ClassNotFoundException ce){
        System.out.print("class not found 1");}
```

6. Implementation

To create internal rules, it is necessary to find out which element types can be recognized by their left element types. For those elements, internal rules can be created easily. However, there may be some elements, which cannot be recognized by their left neighbors. In these cases, a process of finding out special letters at the beginning or at the end of the elements is executed. If some special letters are found, they can be used as distinguishing features and internal rules can be created basing on them. In the worst case, that means, the program cannot recognize an element, it is required to input the rule from outside/human. We would like to show the following pseudo-code, which implements this algorithm:

We assume that

A[n] is the array which contains element types of the LHS of the pattern rule.

A1[n] is the result of shifting elements of the array A[n] one step to the right.

A2[n] is the result of shifting elements of the array A[n] two steps to the right.

C[n] is an array which contains element indexes of the suspicious elements

R[n] is an array which contains created internal rules

```
void Create_Internal_KnowledgeBase{
For i from 1 to n {
  For j from (i+1) to n {
    If  (A1[i] equals to A1[j]) and (A2[i] equals to A2[j]) and (A[i] not equals to A[j])
        then put i, j into C;
  }
}

For i from 1 to n {
  If C not contains i then Create_InternalRule1(i);
  Else if (Finding_SpecialLetter(i)) then Create_InternalRule2(i);
  Else  Input_Rule(i)}
}
}
```

The indexes of elements, which cannot be recognized by their left neighbors, are saved in the array C. For elements, whose indexes are not saved in C, internal rules are created by the function Create_InternalRule1(i). For elements, which belong to C, the function Finding_SpecialLetter(i) is called firstly. If it returns an positive integer, then internal rules can be created by the function Create_InternalRule2(i). In another cases, the function Input_Rules(i) will get the rules from outside.

```
String CreateRule1(i){
return "if  Left(X) = " + A1[i] + "and Pre_left(X) = " + A2[i]  + then X has got the type of " + A[i] + " "
```

```
}

String CreateRule2(i){
        String specialLetter = getSpecialLetter(Finding_SpecialLetter(i));
        return "if X contains " + specialLetter + "at the position " +
                Finding_SpecialLetter(i) +  then X has got the type of " + A[i] + " "
}
```

7. User interface

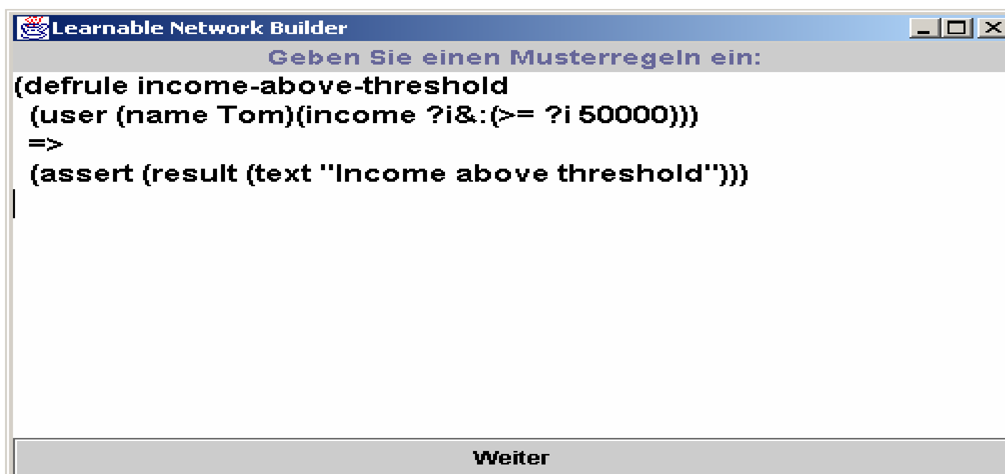


Fig. 7.6: The user interface for giving in training rules

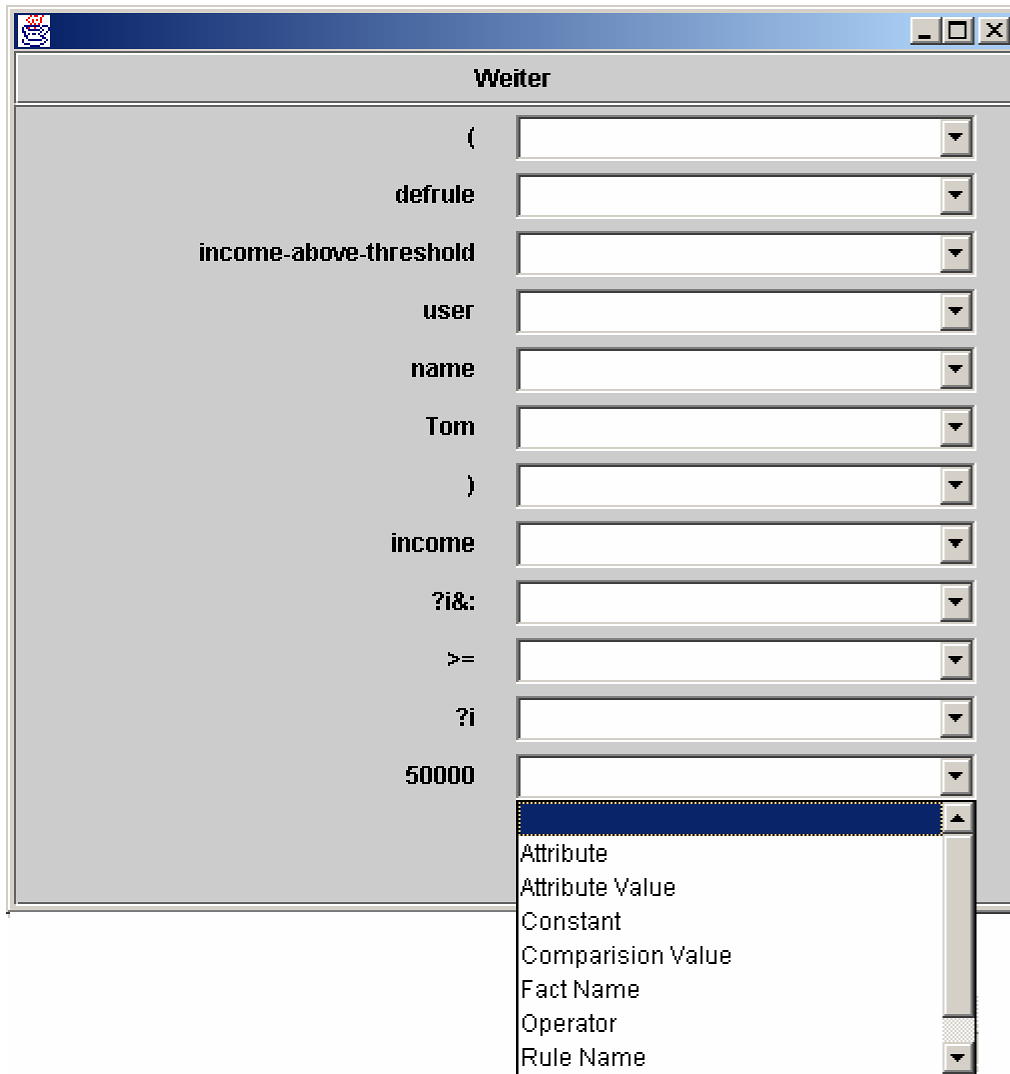


Fig. 7.7: Specifying types for elements

Chapter 8

ADINE - A new matching algorithm for rule-based consultation systems

1. Observations

The essence of my algorithm is to exploit the following observations:

- *Passive Beta node*: in the RETE network, the main duty of a Beta node is to join its tokens together into a new token and sends the new token to its successive Beta nodes. If a Beta node has not received enough two tokens yet, it does nothing. In this case, it is just waiting for tokens. On my opinion, Beta nodes should be more active. An *active Beta node* will send a request for the second token, which it is expecting, after it has received the first one.

- *Answers-relationship*: in the question–answer phase of a consultative session, a new answer often has a relationship with some old answer of a previous question. For example, the answer for the question: “At which ***university*** have you studied?” has a relationship with the answer of the previous question: “In which ***country*** have you studied?“ Because the ***university*** belongs to the ***country***. Furthermore, this relationship reflects binding or joining of variables in rules. If a LHS of a rule contains two facts ***country*** and ***university***, then there must be a joining between these two facts in the rule. For example, let us see a rule:

```
(defrule demo1
   (country (country-id ?cid)(country-name ?cname))
   (university (country-id ?cid)(uni-name ?uname))
  =>
  (assert (summary   (country-name ?cname)
                     (uni-name ?uname) )))
```

The variable “**?cid**“ binds facts “**country**” and “**university**” together.

Basing on this observation, we came to the idea that the joining can be performed at Memory node, where facts are stored. The joining at Beta node is not necessary anymore. As a result, we can spare Beta memories, Alpha memories and also increase the system performance.

2. Components of the reasoning network

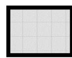
For an easier presentation, let us begin with an example of two following rules:

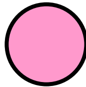
```
If
(( F1 and F2) and (F3 and (F4 or F5) and F6)
Then
Action1

If
(F1 and F2 and F5)
Then
Action2
```

F1, F2, F3, F4, F5, F6 are conditions.

The reasoning network for these rules is built as follows:

 Memory node

 Beta node

Alpha node

"&" stands for the logic operator "and"

"| |" stands for the logic operator "or"

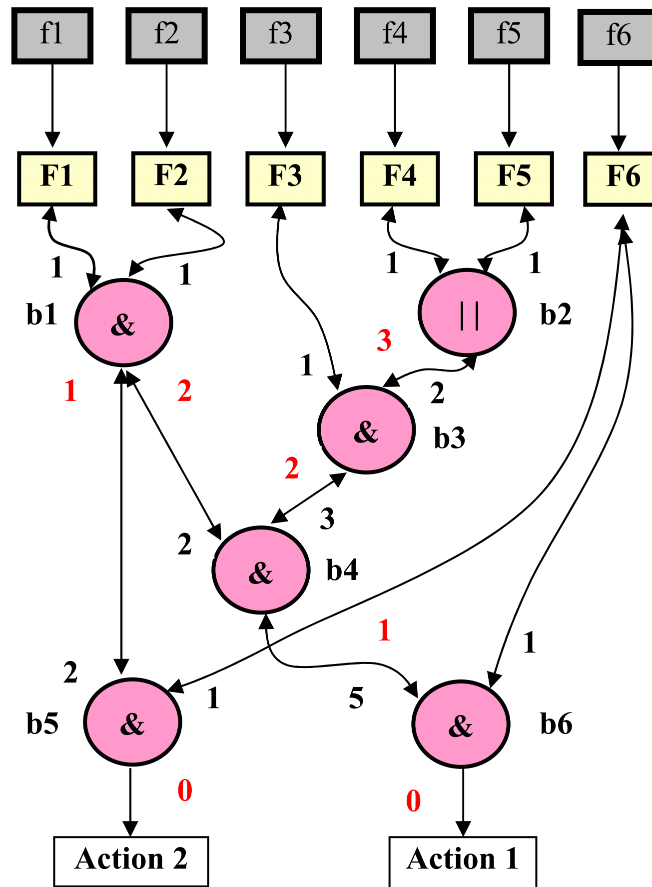


Figure 8.1: reasoning network

We use the structure of the RETE reasoning network as basic infrastructure for my algorithm. That means there are also Memory nodes, Alpha nodes, Beta nodes and connections between them in the reasoning network. The differences are that how these nodes operate in cooperation with the others.

The tasks of a Memory node:

- Two Partitions are used to save facts: Success-Partition and Fail-Partition. Facts, which satisfy some internal condition of some Alpha node, will be saved in the Success-Partition. Facts, which do not satisfy any internal condition of any Alpha node, are saved in the Fail-Partition.

- When all answers (facts) of a question do not satisfy any internal condition of any Alpha node, the Memory node will inform the Control Unit about that by sending a signal "*Fail*" to the Control Unit.

- Providing their Alpha nodes with facts that it is keeping up.

- Receiving question requests from Alpha nodes and putting them into a question queue if it can not satisfy those requests, that means if the Memory node does not own facts, which the Alpha nodes are expecting.

- Performing joining. In our algorithm, joining is not performed at Beta node, but at Memory node. We will present this topic in the section 6.

The tasks of an Alpha node:

- Receiving facts from its Memory node and checking whether they satisfy its internal conditions. If it is satisfied, the Alpha node will inform the Memory node about it and send a token to its successive Beta nodes. A token contains the Identifier of the sending node.

- When receiving a question request from some Beta node, if there has not been any fact that has satisfied its internal conditions, the Alpha node will transfer these requests to its Memory node. Otherwise, it sends a token to the Beta node.

The tasks of a Beta node:

- Saving Ids of rules, which the Beta node belongs to. A Beta node can belong to many different rules, therefore it has to save all Ids of those rules.

- Receiving tokens from Alpha nodes and Beta nodes. If it has got enough tokens on hands, it will send a new token to the successive Beta node, which has got the smallest forward distance number (FDN) and has not received any token from it yet. The new token contains the identifier of the sending Beta node.

In the figure 8.1, the forward distance numbers are the red numbers. That is the distance from a Beta node to the last Beta node of the corresponding rule. For example, **b5**, **b6** are the last Beta nodes, which have a forward distance number of 0. The Beta

node **b4** has got the forward distance number of 1 and so on. That means the choice is action-oriented. The chosen Beta node is the one that is nearest to an action node.

- If a Beta node has got several successive Beta nodes, then after sending a new token to a successive Beta node, it saves its address in the *Active-BetaNode-Stack* that the Control Unit will access later.

- When receiving a question request from another Beta node, it tries to find the shortest way to an Alpha node, and forwards the question request upward to the Alpha node.

In order to find the shortest way, backward distance numbers are used. In the figure 8.1, they are black numbers at each input line of Beta nodes. Each Beta node has got two input lines. Each input line will be granted a number, which expresses the distance from the Beta node to some Alpha node. The input lines, which come directly from Alpha nodes, will have got the backward distance number of 1. For example, the Beta node **b2** has got two input lines and each line has a backward distance number of 1. So the output line of **b2** will have a backward distance number of 2, which is a sum of backward distance numbers of the two input lines.

With these backward distance numbers, a Beta node can find the shortest way to reach an Alpha node. For example, **b3** has got one input line with the backward distance number of 1 and one input line with the backward distance number of 2, so the shortest way is the line to the Alpha node **C** with the backward distance number of 1.

- If a Beta node is the last node of a rule, after it has got enough tokens in hand, it sends a token to an Action node.

The tasks of an Action node:

- Performing commands in the action part of the fired rule. If the action part does not contain the command "*add-fact*", that means the action part contains a conclusion or a result. In this case, the Action node will send a signal "*Success*" to the Control Unit, so that the Control Unit may start the process of looking for an alternative result.

The tasks of the Control Unit:

- Serving question requests from Memory nodes. The Control Unit will send requests to the Interface module, receives coming facts from it and delivers them to the Memory nodes.

- Serving requests of Why- and How-explanation. We will explain this function later.

- When receiving the signal "*Success*" from an Action node, it saves the rule-id, which the Action node belongs to, in the *Fired-Rules-List* then it selects a Beta node from the *Active-BetaNode-Stack* and activates the Beta node. The purpose for this work is to find an alternative result, if it is possible. In the section 6, we are going to discuss about how to choose a Beta node to fire.

3. Comparison with the RETE network

This network is similar to the RETE network, however there are some differences:

- In the RETE network, there is only one direction of data flow, from top to bottom. In this network, there are two directions of data flow, from top to bottom for transmitting tokens like in RETE network, from bottom to top for transmitting question requests.

- The nodes in this network are more active. If they receive a request and cannot satisfy it, they will forward it to the other.

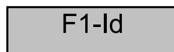
- In the RETE network, joining is performed at Beta nodes. In this network, joining is performed at Memory nodes.

- In the RETE network, there are a lot of redundancies, because an instance can be stored in different locations (Memory node, Alpha memory node, Beta memory node). In this network, instances are stored only in Memory nodes. There is no joining at Beta nodes. Therefore it is unnecessary to store instances in Alpha memory nodes and Beta memory nodes.

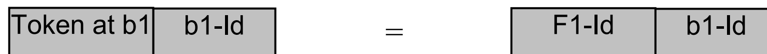
The essential difference between two networks will be illustrated in the next part.

4. How the nodes work together? - An example

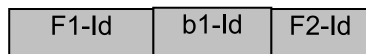
Let's assume that a fact for the Memory node **f1** is inserted. The Memory node **f1** will save the fact into its Temporality-Partition and forwards the fact to its Alpha nodes. In this case, the Alpha node **F1** will receive the fact, and checks it against its internal conditions. **F1** informs **f1** about the result of this checking. If no internal condition has been satisfied, **f1** saves the fact into its Fail-Partition. Otherwise, **f1** will save the fact into its Success-Partition. Then the Alpha node **F1** sends a token to one of its successive Beta nodes, which has got the smallest forward distance number (FDN). In this case, it is the Beta node **b1**. The token contains the Id of the sending node **F1**:



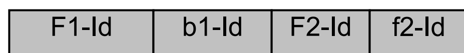
The Beta node **b1** receives a token from **F1**, it recognises that it needs also a token from the Alpha node **F2**. Therefore **b1** sends a question request to the Alpha node **F2**. The question request has the following format:



The Alpha node **F2** will append its Id to the question request and sends it to the Memory node **f2.**



The Memory node **f2** will also append its Id to the question request and put the question request into a question queue, because **f2** has not owned any fact for the Alpha node **F2** in its Success-Partition yet.



The Control Unit will take the question request from the question queue, and sends a request to the Interface module. After receiving an answer (fact) from the Interface module, the Control Unit will send it to the Memory node **f2**.

The answer has following format:

F1-Id	b1-Id	F2-Id	fact

The Memory node **f2** takes the Id of the Alpha node **F2** away, and sends the fact to the Alpha node **F2**.

F1-Id	b1-Id	fact

The Alpha node **F2** will get the fact, checks it against its condition. If the condition is satisfied, it informs the Memory node **f2** about that. **F2** will move the fact from its Temporary-Partition to its Success-Partition, saves **F1-Id** into the list of Beta nodes, which sent a request to it, then the **F2** sends a token to **b1.**

The Beta node **b1** has got two tokens on hand, it sends a new token to one of its successive Beta nodes, which has got the smallest forward distance number. In this case, it is **b5**. The new token is the Id of **b1**.

b1-id

If **b5** receives a token from the Alpha node **F6**, it will put an action request into the conflict set. The action request contains the Id of the last Beta node and the rule-id, which the last Beta node belongs to:

b5-Id	Rule-Id

After serving **b5**, if **b5** do not get a token, that means, no fact at the Memory node **f6** satisfied the internal conditions of **F6**, **F6** informs **f6** about that. The Memory node **f6** will send a message of fail to **b1**. Afterward, **b1** selects another Beta node from its Beta node list, which has got also the smallest forward distance number except **b5**. In this case, **b4** is chosen. Now **b4** sends a question request to **b3**. **b3** will find the shortest way to reach an Alpha node basing on backward distance number (BDN). The way to the Alpha node **F3** is the best one, because of the smallest backward distance number of **1**. The progress is continuous…

5. Forwarding question request

As we discussed above, Beta nodes forward question requests upwards to the Beta nodes, which have got the smallest BDN. In this section, we complete this strategy to guarantee that the question-answer process can be always performed successfully.

Each question request, which originated from some Beta node, can be sent to some Memory node successfully through the strategy of finding the shortest path to an Alpha node basing on BDNs. But what happens, if the user's answer does not satisfy the internal condition of the Alpha node on the found path. In this case, the system got an unsatisfied answer from the user. We may ask ourselves, whether there may be another path, which leads to another Alpha node and another Memory node and as a result, the user will receive another question. But with this question the system may get a satisfied answer from the user? When building a strategy for forwarding question request, we have to consider this factor to make sure that an optimal path can always be found. Therefore, the complete strategy is as follows:

When a Beta node receives a question request, its reaction depends on which type of logic operator (and, or) the Beta node has got.

- A Beta node of the type "*and*" will append its Id to the question request and send it upwards to the node, which has the smallest backward distance number.

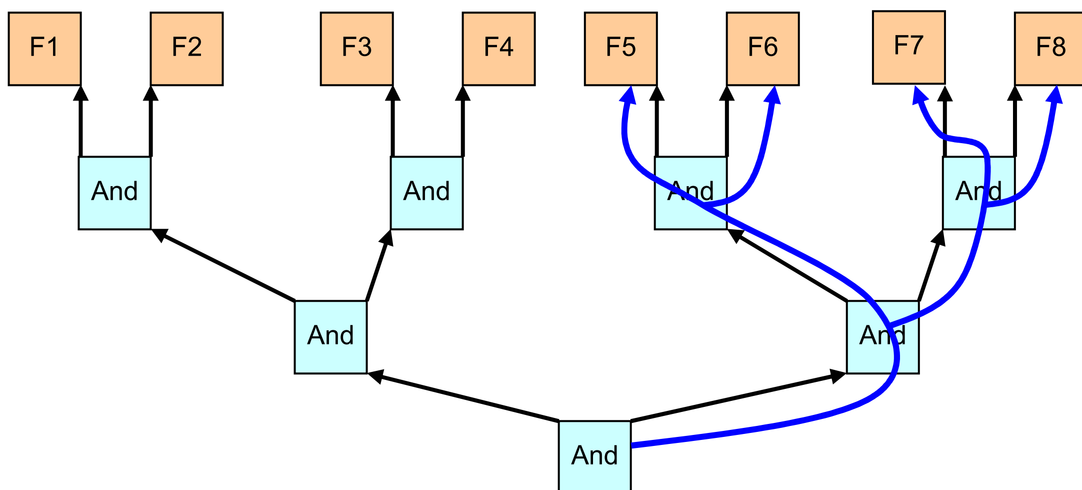


Figure 8.2: In all paths there are only Beta nodes of the type "*and*"

In the Figure 8.2, there are 4 different paths for a question request, which begins from the lowest Beta node and directs towards different Alpha nodes F5, F6, F7, F8. In these paths, there are only Beta nodes of the type "*and*". Any of them can be chosen, because it gives the same result.

- A Beta node of the type "*or*" will save the original question request, then it creates a new question request and sends it upwards to the node, which has got the smallest BDN. If it receives a token for its question request, it will send a token downwards to the original requesting Beta node, which is the owner of the original question request, basing on the original question request. Otherwise, it creates a new question request again and sends it upwards to the second node (Remember, each Beta node has got only two inputs). After that, if it receives an announcement from some Memory node that the user's answer does not satisfy internal condition of the Alpha node in the found path, it will send a signal "Fail" to the Control Unit.

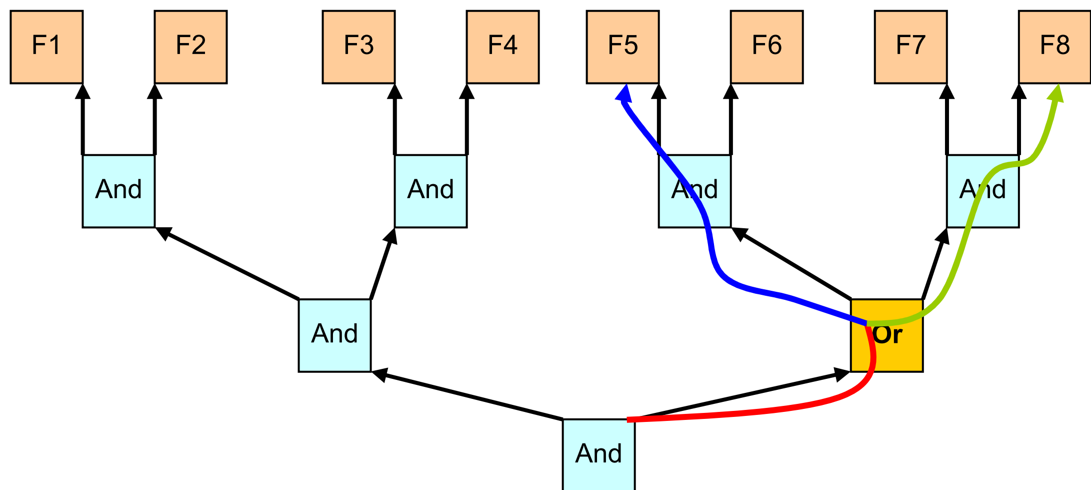


Figure 8.3: There is one Beta node of the type "*or*" on the way to Alpha nodes

In the figure 8.3, the Beta node of type "*or*" replaces the original question request (the red line) with a new question request, which will be forwarded to F5 on the blue line. If it does not get a token for this new question request, it creates a new question request again, and sends it on the green line. At the end, if it still does not get a token for its question request, it informs the Control Unit about the failure. Otherwise, it sends a token downwards towards the original requesting Beta node.

6. Activating a Beta node

When receiving the signal "*Fail*" from a Memory node or the signal "*Success*" from an Action node, the Control Unit will get a Beta node address from the *Active-BetaNode-Stack* and activates the Beta node.

The activated Beta node will choose a Beta node among its successive Beta nodes to send a token to. There are some criteria for the selection:

- The Beta node has not received any token from the activated Beta node yet
- The Beta node does not belong to any rule in the *Fire-Rules-List*
- The Beta node has got the smallest FDN.

7. Joining at Memory node

In my algorithm, Beta nodes do not perform joins. The joins are performed at Memory nodes. This new feature helps to reduce redundancies in Beta memories. In order to understand how a Memory node does the joins, let's begin with the process of creating a question as follows:

(1) A Memory node receives a question request from an Alpha node. Normally the question request is to aim at acquiring values of a determined attribute of the Memory node.

(2) The Memory node goes to its Initiation-Partition to get instances of the requested attribute. If there are father instances of the attribute in the Success-Partition, the Memory node gets instances in turn, which belong to the father instances. The instances will be used as answer possibilities for the question. Then it sends the instances together with the requested attribute and the father instances to Interface module.

(3) The Interface module creates a question corresponding to the requested attribute and provides the user with answer possibilities for his choice. The user can only choose one or several answers from the answer possibilities, which are classified according to their father instances.

(4) After receiving answers from the user, the Interface module delivers the requested attribute, the user's answers as the attribute's instances, and its father instances, to the Memory node.

(5) The Memory node sends the user's answers (instances) to the Alpha node, which sent the question request. The Alpha node will inform the Memory node about which instances have passed its internal condition(s). Then the Memory node saves them into the "Success-Partition" according to their father instances. That means, the Memory node performs joins between satisfied instances to their father instances.

The instances, which did not pass the internal condition(s) of the Alpha node, will be saved in the Fail-Partition.

From (2), we can see that the joining starts, when the Memory node gets instances (from the Initiation-Partition), which belong to father instances of the requested attribute. So the user can only choose his answers from these instances. The joining ends after (5) is performed. From this reason, it is not necessary to perform joins at Beta nodes anymore.

It will be illustrated more, when we present structure of a Memory node and connections between them as well in the next chapter.

8. Explanation ability

- Why-explanation:

The user may want to know why he has to answer a question from the system or why the system needs the asking information. To respond to a request of why-explanation, some following information is needed:

- What facts has the user given in?
- Which rules are being observed?

The Control Unit can provide the explanation module with this information. Because for each question, the Control Unit knows the Id of the requesting Beta node, which is contained in the question request. From it, the Control Unit knows which rules the

request Beta node belongs to, because each Beta node keeps its rule-ids in its memory. Furthermore, the Control Unit keeps track all previous question requests and corresponding Memory nodes. From these, it can get information about facts, which the user has given in.

- How-explanation:

After receiving advice from the system, the user may not trust it immediately, and he/she wants to know how the system has reached these advices. The system should persuade him of the way that the system has done, that means the user needs a How-explanation. To answer a request of How-explanation, the "Explanation" module needs some information:

- What facts has the user given in?
- Which rule was used to reach the result?

Remember, a Beta node puts an action request into conflict set in following format.

Its Id	Rule-id

The Rule-id identifies which rule was used to reach the result (the answer for the second question).

The first question can be answered by the Control Unit because the Control Unit keeps track all previous question requests and corresponding Memory nodes. From these, it can get information about facts, which the user has given in. Therefore the "Explanation" module can get enough information from the Control Unit to do its task.

9. Optimising the speed of inference engine

There are some ways to improve the speed of inference engine:

- Using the distance number. We presented how to use it above. The main idea is to find the shortest way so that a question request reaches a Memory node as soon as possible.

- The Ids of nodes are saved in each question request. Therefore, it economizes much time for the return way. If not, nodes have to broadcast tokens as in RETE algorithm. It is to avoid the problem of "null right activations" [ROB95]

- One important ability of human being is the ability to learn from experience. The inference engine should have this ability, so that it can improve its performance itself.

The Control Unit can observe the process of performing a question request. If it is successful, an experience can be saved. An experience is simply the question request, which was performed successfully.

For example, after performing a question request from the Beta node **b4** successfully, the Control Unit can save the question request as an experience:

b4-Token	Id of b4	Id of b3	Id of F3	Id of f3

This experience means, when **b4** has received a token from **b1**, then a question request as above should be performed by the Control Unit. From now on, **b4** do not have to initialize a question request as before. This way can save much time for the next time.

- One important ability of human being is the ability to predict what will happen and then he can prepare himself for it. So when it happens, he can react more quickly. If an inference engine has got this ability, it can improve its performance itself.

In stage 2 of the consultative process, an inference engine does nothing while the user is choosing an answer for the received question. Normally a user needs at least 1 second to read the question and to answer it. The inference engine can economize this duration to make plans. There are only two possibilities: the coming fact will satisfy the internal condition of the Alpha node or not. So the inference engine can prepare itself as follows:

- If the coming fact satisfies the internal condition of the Alpha node, the requesting Beta node will receive a token, and then sends a token to its successive Beta nodes. Applying principles of choosing a question request from Beta nodes, the Control Unit

can choose one from these successive Beta nodes to serve first. Then the process of creating a question request can be performed by the chosen node, if it doesn't have any experience. So the Control Unit will receive a question request before the expecting fact comes.

- If the coming fact does not satisfy the internal condition of the Alpha node, the Control Unit will choose the second request of previous cycle from some Beta node to serve.

These works can be prepared or done before the user's answer comes. This helps the inference engine to react more quickly when a fact comes

Chapter 9

Implementation of ADINE

1. Data structure of a Memory node

Data structure of a Memory node consists of three partitions: Initiation-Partition, Success-Partition, and Fail-Partition.

- Initiation-Partition contains initial facts. Normally these facts are used in the question-answer phase to provide users with answer possibilities corresponding to questions.
- Success-Partition maintains facts, which have passed constant tests in some Alpha node.
- Fail-Partition keeps facts, which have not satisfied constant test of any Alpha node.

The general structure of a Partition is as follows:

FactName				
AttributeName 1		AttributeName 2		...
LinkedFactNam	LinkedAttribute	LinkedFactNam	LinkedAttribute	...
FatherInstance	FatherInstance	FatherInstance	FatherInstance	...
Attr.-Value 1	Attr.-Value 1	Attr.-Value 1	Attr.-Value 1	...
....	....	....	....	...
Attr.-Value n	Attr.-Value n	Attr.-Value n	Attr.-Value n	...
ListOfAlphaNodeId				

Figure 9.1: Structure of a Partition of Memory node

Each Memory node represents a type of fact, which can consist of many attributes. Each attribute may be linked to an attribute of another fact. The link represents the join between two attributes. Therefore, a "*LinkedFactName*" and a "*LinkedAttribute*" are reserved to store the names of a fact and an attribute that are referred to by the attribute. The fact and the attribute, which are referred by another attribute, are called *source fact* and *source attribute*.

Each attribute can own many values, we call them instances of the attribute. The Memory node has to maintain a father instance for each attribute value. Father instances are instances of the *source attributes*. A pair of father instance and attribute value presents a link between this attribute and the *source attribute* of the *source fact*.

Each Memory node maintains a list of Alpha nodes “*ListOfAlphaNodeId*”, which the Memory node will send user’s answers (instances) to.

A demonstration is presented in the next page.

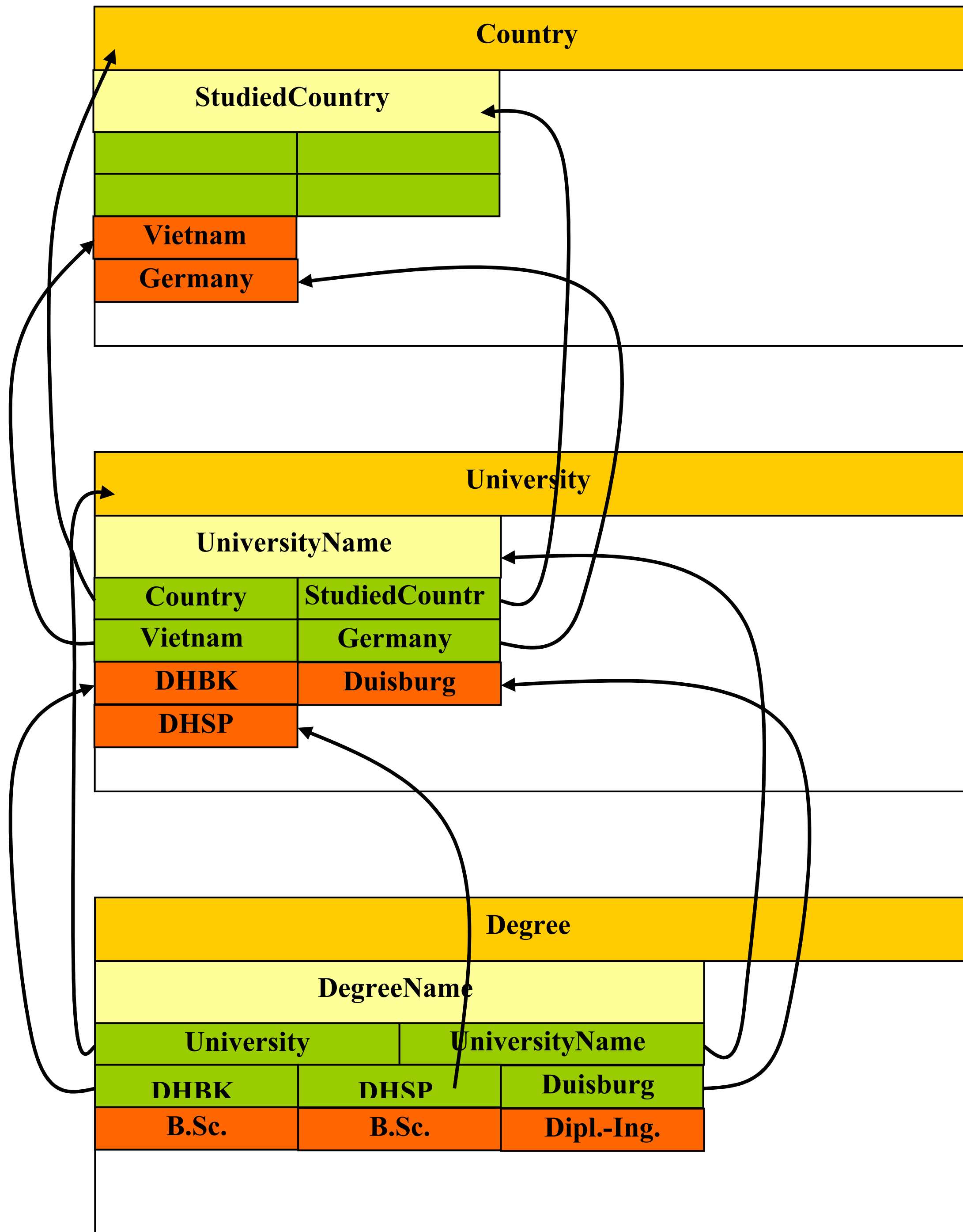


Figure 9.2: A demonstration of Memory node

In this demonstration, we have three Memory nodes, which presents three facts: "*Country*", "*University*" and "*Degree*". Each of them has got an attribute "*StudiedCountry*", "*University*", "*Degree*" correspondingly.

For the attribute "*StudiedCountry*" of the Memory node "*Country*", there is no "*LinkedFactName*" and "*LinkedAttribute*", because this attribute does not refer to any other attribute. Therefore, there is no "*FatherInstance*" for its instances.

The attribute "*UniversityName*" of the Memory node "*University*" refers to the attribute "*StudiedCountry*" of the Memory node "*Country*". For this reason, instances of the attribute "*StudiedCountry*" will be father instances of instances of the attribute "*UniversityName*". For example, "DHBK", "DHSP" have got the same father instance of "*Vietnam*". The another instance "*Duisburg*" has got the father instance of "*Germany*".

It is similar for the Memory node "*Degree*". Its sole attribute "*DegreeName*" refers to the attribute "*UniversityName*" of the Memory node "*University*". Therefore, instances of the attribute "*UniversityName*" such as "*DHBK*", "*DHSP*", "*Duisburg*" are father instances of instances of the attribute "*DegreeName*".

Although each Memory node in this demonstration has got only one attribute, but in general there is no limitation for the amount of attributes in a Memory node.

The key advantage of this structure is the support of joining at Memory node. In the figure 9.2, links between attributes within Success-Partitions of Memory nodes represent joins between variables across conditions within rules.

2. Data structure of an Alpha node

AlphaNodeId		ListOfRuleId	
AttributeName 1	AttributeName 2	AttributeName 3	....
Operator	Operator	Operator	....
ComparedValue	ComparedValue	ComparedValue	....
MemoryNodeId			
BetaNodeId 1		FDN	
....		....	
BetaNodeId n		FDN	
Status			

Figure 9.3: Data structure of an Alpha node

For an easy maintenance, each Alpha node is allocated an identifier (Id). Rule-Ids should be saved in the "*ListOfRuleId*" of the Alpha node, so that it can support the explanation ability of the system.

The main duty of an Alpha node is to verify whether coming facts satisfy its internal conditions. In general, an internal condition can be applied on every attribute. Therefore, an operator and a compared value are reserved for each attribute. When there is at least a fact, whose attribute values satisfy all of internal conditions, the "*Status*" will be set to "*Passed*", otherwise it is set to "*Fail*". When an Alpha node receives a question request from a Beta node, it checks its "*Status*" again. If it is "*Passed*", the Alpha node does not forward the question request to Memory node anymore, but it sends a token to the requested Beta node.

Each Alpha node has to communicate with the Memory node, which it received an attribute values from. Therefore it has to save the address of the Memory node in its memory. Each Alpha node can have several connections to Beta nodes. Therefore, it has to also save Ids of them in "*BetaNodeId n*".

3. Data structure of a Beta node

BetaNodeId	ListOfRuleId	Token
LeftInput	Type	BDN
RightInput	Type	BDN
BetaNodeId 1		FDN
....		....
BetaNodeId n		FDN

Figure 9.3: Data structure of an Alpha node

Each Beta node has got two inputs, which are from either Beta nodes or Alpha nodes, which we call *father nodes* for an easy reference. Beta node keeps the addresses of these input nodes in its data. In order to differentiate inputs, we call them "*LeftInput*" and "*RightInput*". The "*Type*" indicates whether the input node is an Alpha node or a Beta node. Backward Distance Number "*BDN*" helps the Beta node to find down the shortest way to reach an Alpha node. "*Token*" contains either true or false, which indicates whether both of two input nodes own tokens or not.

In our algorithm, the Beta node does not perform joining. After receiving enough tokens (two tokens for a Beta node of type "*and*", may be one token for a Beta node of type "*or*"), the Beta node sends its identifier as a token to its successive Beta nodes, whose addresses are stored in its data.

4. Implementation

In this section, we only aim at describing main ideas of the implementation. Three classes MemoryNode, AlphaNode and BetaNode will be introduced basing on the above analyses. These classes are main components of the reasoning network, which is presented in the section 4.4.

4.1. Memory class

```
public class MemoryNode {
```

```
  private  String FactName;
  private  Vector AttributeNames = new Vector();
  private  Vector AttributeValues = new Vector();
  private  Vector LinkedFactNames = new Vector();
  private  Vector LinkedFactValues = new Vector();
  private  Vector LinkedAttributeNames = new Vector();
  private  Vector LinkedAttributeValues = new Vector();

  private  Vector AlphaNodes = new Vector();
  private  AlphaNode anode;

  MemoryNode(String FactName, Vector AttributeNames){
      this.FactName = FactName;
      this.AttributeNames = AttributeNames;
  }

  public void SetLink(String Attribute, String LinkedFact,
String  LinkedAttribute,  String  LinkedFactValue,  String
LinkedAttributeValue){
…
}

  public void AddAlphaNode(AlphaNode a){
      AlphaNodes.addElement(a);
  }

  public void AddNewFact(String Value){
   …
  }

  private void SendValueToAlphaNode(String Value){
      Enumeration v = AlphaNodes.elements() ;
```

```
      for (; v.hasMoreElements() ;) {
           AlphaNode a = (AlphaNode)v.nextElement();
           a.ReceivingFactFromMemoryNode(this, Value);
      }
  }
}
```

4.2. Alpha Node class

```
public class AlphaNode{

private String AlphaNodeId;
private Vector ListOfRulesId = new Vector();
private Vector Attributes = new Vector();
private Vector operators = new Vector();
private Vector ComparedValues = new Vector();
private MemoryNode mnode;
private Vector BetaNodes = new Vector();
private Vector FDNs = new Vector();
private boolean Status = false;
private BetaNode req_bnode;

AlphaNode(String AlphaNodeId, MemoryNode m, String RuleId)
{
   this.AlphaNodeId = AlphaNodeId;
   this.mnode = m;
   m.AddAlphaNodes(this);
   ListOfRulesId.addElement(RuleId);
}

public void AddBetaNode(BetaNode b){
   BetaNodes.addElement(b);
}
```

```
public void ReceivingFactFromMemoryNode(MemoryNode sender,
String Fact){
  boolean result = false;
  this.mnode = sender;
  result = Check_Condition(Fact);
  ConfirmToMemoryNode(result);
  if (result){
     SendTokenToBetaNodes();
  }
}

public String GetAlphaNodeId(){
  return AlphaNodeId;
}

private void ConfirmToMemoryNode(boolean result){
   this.mnode.ConfirmFromAlphaNode(result);
}

private void SendTokenToBetaNodes(){
   int i = FindingSmallestFDN(FDNs);
   BetaNode b = BetaNodes.elementAt(i);
   b.ReceivingTokenFromAlphaNode(true);
}

public void ForwardRequest(BetaNode b){
   this.req_bnode = b;
   mnode.ForwardRequest(this);
}
}
```

4.3. Beta node class

```
public class BetaNode {
```

```
  private String BetaNodeId;
  private Vector BetaNodes = new Vector();
  private Vector FDNs = new Vector();
  private BetaNode LeftInput1;
  private BetaNode RightInput1;
  private AlphaNode LeftInput2;
  private AlphaNode RightInput2;
  private int BDN1;
  private int BDN2;
  private String LogicOperator;
  private String Token = "";

  private BetaNode RequestNode;
  private String type;

  public BetaNode(String id, AlphaNode a1, AlphaNode a2,
String op) {
      this.BetaNodeId = id;
      this.LeftInput2 = a1;
      this.RightInput2 = a2;
      this.LogicOperator = op;
      this.BDN1 = 0;
      this.BDN2 = 0;
      type = "b1";
  }

  public BetaNode(String id, AlphaNode a1, BetaNode b1, int
BDN, String op) {
      this.BetaNodeId = id;
      this.LeftInput2 = a1;
      this.RightInput1 = b1;
      this.LogicOperator = op;
      this.BDN1 = 0;
```

```
        this.BDN2 = BDN;
        type = "b2";
    }

    public  BetaNode(String  id,  BetaNode  b1,  int  BDN,
AlphaNode a1, String op) {
        this.BetaNodeId = id;
        this.LeftInput1 = b1;
        this.RightInput2 = a1;
        this.LogicOperator = op;
        this.BDN1 = BDN;
        this.BDN2 = 0;
        type = "b3";
    }

    public  BetaNode(String  id,  BetaNode  b1,  int  BDN1,
BetaNode b2, int BDN2, String op) {
        this.BetaNodeId = id;
        this.LeftInput1 = b1;
        this.RightInput1 = b2;
        this.LogicOperator = op;
        this.BDN1 = BDN1;
        this.BDN2 = BDN2;
        type = "b4";
    }

    public int getMinBDN(){
        if (type.equals("b4"))
            return min(this.BDN1, this.BDN2);
        if (type.equals("b3")) return 0;
        if (type.equals("b2")) return 0;
        if (type.equals("b1")) return 0;
```

```
  }

  private void ReceivingToken(String type, Object obj){
      SetStatus();
      if (this.Status.equals("token")) {
          SendingToken();
      }
      else
          SendingRequest();
  }

  private void SendingToken(){
      int i = GetMinFDN();
      BetaNode b = (BetaNode)BetaNodes.elementAt(i);
      b.ReceivingToken(this.type, this);
  }



  private void SendingRequest(){
  }

  private void ReceivingRequest(BetaNode b){

      if (this.Status.equals("token"))
          b.ReceivingToken(this.type, this);
      else{
          this.RequestNode = b;
          SendingRequest();
      }
  }
}
```

4.4. Building the reasoning network

After we have discussed about the system and data structures of nodes in reasoning network, now we would like to cooperate all things together to see how a reasoning network can be built from rules. But first of all, let us make three assumptions:

- The system has been trained by pattern rules, so that the "*Element Recognition*" can understand the language that is used to represent the knowledge base.

- The system has not owned any experience yet.

- Each Memory node, Alpha node, Beta node corresponds to a well-defined class, which contains functions as specified in the chapter 8, section 2, and contains data structures as discussed in the section 1, 2, 3 of this chapter.

- Because each node have to store identifiers (Ids) of its input nodes, while building the reasoning network, three stacks **memory-stack, alpha-stack, beta-stack** are used to save Ids of Memory nodes, Alpha nodes, Beta nodes correspondingly, so that each node can retrieve Ids of its input nodes easily.

- A hash table **variable-ht** is used to save pairs of variables and corresponding attributes. Normally, variables are used to express variable links.

- Three hash tables **memory-address, alpha-address, beta-address** are used to save Ids and addresses of corresponding nodes.

When building a reasoning network, the system will perform following steps:

- Basing on the element types, which are the results from the "*Element Recognition*", nodes such as Memory nodes, Alpha nodes or Beta nodes are created correspondingly.

- After creating a Memory node, the "*Reasoning Network Builder*" (RNB) puts the Id of the new Memory node into the **memory-stack** and puts a pair of the Id and the address of the new Memory node into the hash table **memory-address**.

- After creating an Alpha node, the RNB puts the Id of the new Alpha node into the **alpha-stack**, and puts a pair of the Id and the address of the new Alpha node into the hash table **alpha-address**. Then it gets one element (Id of a Memory node) from the **memory-stack**, saves it into the field "*MemoryNodeId*" of the new Alpha node, and set the "*Backward Distance Number"* **BDN** to 0.

Basing on the hash table **memory-address**, the RNB can retrieve the address of the Memory node corresponding to the Id saved in the field "*MemoryNodeId*". The Id of the new Alpha node is also appended to the "**ListOfAlphaNodeId**" in the retrieved Memory node.

- After creating a Beta node, the RNB puts the Id of the new Beta node into the **beta-stack**, and puts a pair of the Id and the address of the new Beta node into the hash table **beta-address,** then perform following steps:

If (there are two Ids of alpha nodes in the alpha-stack)
{
    Get two Ids of alpha nodes from the alpha-stack
    Save Id of the new beta node into two alpha nodes (*BetaNodeId*)
    Save Ids of two alpha nodes into the new beta node (*LeftInput, RightInput*)
    Get two BDNs from the two alpha nodes, add **1** to each of them and save them into the new beta node at appropriate places
}

If (there is one Id of an old alpha node in the alpha-stack and one Id of a old beta node in beta-stack)
{
    Get Id of the old alpha nodes from alpha-stack
    Get Id of the old beta nodes from beta-stack
    Save Id of the new beta node into the old alpha node and the old beta node
    Save Id of the old alpha node and the old beta node into the new beta node (*LeftInput, RightInput*)
    Get two BDNs from the old alpha node and old beta node, add **1** to each of them

```
        and save them into the new beta node at appropriate places
}

If (there are two Ids of beta nodes in beta-stack)
{
        Get two Ids of beta nodes from beta-stack
        Save Id of the new beta node into the two old beta nodes
        Save Id of the two old beta nodes into the new beta node (LeftInput, RightInput)
        Get two BDNs from the two beta nodes, add 1 to each of them and save them
        into the new beta node at appropriate places
}
```

- When the processing element has got the type of variable, the BDN searches in the hash table **variable-ht** for an identical variable. If the variable exists already, it builds a link between an attribute of the present Memory node and an attribute of the old Memory node, whose name has been found in the hash table **variable-ht**. That means, it saves the name of the old Memory node and its attribute into the fields "*LinkedFactName*" and "*LinkedAttribute*" of the new Memory node.

- Inserting Forward Distance Numbers (FDNs) into Beta nodes: after an Action node is created, it sends a message to its input Beta nodes. The message contains Id of the sender and the smallest FDN plus 1. At the moment, the sender is the new Action node, which has got the smallest FDN of 0.

When receiving the message, the Beta node saves the FDN into the appropriate place, then it creates a message, which consists of its Id and its smallest FDN plus 1, and sends the message to its input nodes. Each Beta node may contain several FDNs that correspond to its output nodes. It will choose the smallest FDN, adds 1 to it and appends it into the message.

We can see that the process of building a reasoning network costs much time. Therefore, the system should do it only one time. Then it saves the resultant reasoning network as its experience. At the next time of a new initiation, if there is no change in

knowledge base, the saved experience will be used to rebuild the reasoning network. In chapter 12, we will discuss about saving experience.

Chapter 10

Evaluations on the ADINE

The ADINE has got also advantages, which the RETE algorithm has got. In addition, ADINE algorithm improves some drawbacks of the RETE algorithm as following:

Waste of memory: ADINE saves facts only in Memory nodes. Beta memories and Alpha memories store only dummy tokens to indicate that there are real tokens in the Success-Partitions of Memory nodes. Therefore, there are no multiple versions of facts that are stored in Memory nodes, Alpha nodes and Beta nodes. It avoids the problem of waste of memory.

Utility problem: *Null-right activations* do not happen with ADINE algorithm. Because each right activation respond to a question request that originated from a Beta node. Although *Null-left activations* can happen, its negative affect is very smaller than one in the RETE algorithm, because ADINE algorithm does not perform the joining in Beta nodes.

Unstable performance: The essence of this problem is that the RETE algorithm performs joins at Beta nodes blind. That means, before a token comes, the Beta node does not know how to join the new token with the old ones. Therefore, a searching process is carried out before joining, and how long the searching process takes depends on the order of conditions of productions. On the contrary, the ADINE algorithm performs the joining at Memory nodes actively. That means, before a fact comes it knows which fact should join to the new one, because it controls the process of asking questions totally.

Unsuitable for changes: the ADINE has got also this drawback like the RETE. With consultation systems this drawback is acceptable, because changes happen seldom. Facts, which are given by users, are changed only when there have been mistakes while choosing answers.

Chapter 11

Mathematical Evaluation

In this chapter, we would like to give mathematical evaluations on my algorithm that was presented in the chapter 8. The evaluations are performed through specifying the costs of adding a new fact, deleting a fact and memory costs.

1. Cost of addition

Before building a mathematical model for ADINE, we would like to review shortly what happens when a token enters an Alpha node:

- The Alpha node checks the token against its internal conditions. If it is satisfied, the Alpha node informs the appropriate Memory node about the success. The Memory node performs a join by looking in its memory for a father instance, which identifies with the father instance of the new fact, so that it can save the new fact at a right place in its memory, which corresponds to the father instance. The cost of this operation depends on the how many father instances the father Memory node has got. In other words, it is the size of the Success-Partition of the father Memory node. We can calculate the average size of a Success-Partition, and assume that this average size is the cost of joining at a Memory node approximately.

$$\mathbf{J} = \frac{1}{k}\sum_{i=1}^{k} m_i \qquad \textbf{(1)}$$

In which

k is the number of Memory nodes

$\mathbf{m_i}$ is the size of the Success-Partition of the $\mathbf{i^{th}}$ Memory node

- Then the Alpha node sends a token to its Beta nodes. There is no join at Beta nodes. At this time, if two input nodes of a Beta node have got tokens, the Beta node sends a token to one of its successive Beta nodes, which have the smallest FDN.

Otherwise, it sends a question request upwards towards some Alpha node, which is responsible for its other token. The cost of sending a question request to an Alpha node $\mathbf{Q_i}$ is the cost of finding a path from the Beta node to an Alpha node, when starting at the input, which the Beta node is expecting a token from. Our algorithm provides the ability of finding the shortest way to an Alpha node basing on BDNs. Furthermore, because of the ability of learning from experience, the Beta node does this search only one time and saves the path in its memory for next usages. We can assume that $\mathbf{Q_i} = 0$, because after giving in rules, one should run the system for testing normally. Through the testing process, the inference engine can learn from experience and paths for question requests are found and saved for next usages.

The process of sending token is similar to the one in RETE. Therefore, we can use the formula for RETE with some modifications.

By RETE the cost of adding a token at the $\mathbf{m}^{\mathbf{th}}$ Alpha node in a network with $\mathbf{n}$ Alpha nodes

$$\mathbf{A_{R(m,n)} = B_{m-2} + B_{m-1}\alpha_{m+1} + \ldots + B_{n-2}\alpha_n} \qquad \mathbf{m > 2} \qquad \mathbf{(2)}$$

$\mathbf{B_i}$ is the size of the Beta node $\mathbf{i}$

Call α_r is the average of α_i, $\mathbf{B_r}$ is the average of $\mathbf{B_i}$

$$\alpha_r = \frac{1}{n}\sum_{i=1}^{n} \alpha_i \qquad \mathbf{(3)}$$

$$\mathbf{B_r} = \frac{1}{n}\sum_{i=1}^{n} B_i \qquad \mathbf{(4)}$$

Now (2) becomes

$$\mathbf{A_{R(m,n)} \approx B_r + (n - m)\, B_r\alpha_r} \qquad \mathbf{(5)}$$

By ADINE, the size of Alpha memories and Beta memories are equal to either 1 or 0, because these memories contain maximum of one dummy token to indicate that there are real tokens in the Success-Partitions of Memory nodes, otherwise they do not keep any token.

$\alpha_i = 0$ if there is no real token in Success-Partitions **(6)**

$\alpha_i = 1$ if there are real tokens in Success-Partitions

and

$B_i = 0$ if there is no real token in Success-Partitions **(7)**

$B_i = 1$ if there are real tokens in Success-Partitions

By ADINE, the average size of Alpha memories and Beta memories:

$$\alpha_a = \frac{1}{n}\sum_{i=1}^{n}\alpha_i \quad (8)$$

$$B_a = \frac{1}{n}\sum_{i=1}^{n}B_i \quad (9)$$

From **(6), (7)** , we have $0 \le \alpha_a, B_a \le 1$

RETE saves real tokens in Alpha memories and Beta memories. Therefore in general, the average sizes of Alpha memories and Beta memories by RETE are bigger than ones by ADINE

$\alpha_a \le \alpha_r$ **(10)**

$B_a \le B_r$ **(11)**

$A_{A(m,n)}$ is the cost of adding a token by ADINE when a token enters the mth Alpha node in a network with **n** Alpha nodes.

$$\mathbf{A_{A(m,\,n)} = B_a + (n - m)B_a\alpha_a + J} \qquad \textbf{(12)}$$

ADINE saves all instances, which have passed internal conditions of Alpha nodes in Success-Partitions of Memory nodes. **J** is the average size of Success-Partitions that expresses the average cost of joining for each cycle. In comparison with RETE, we have $\mathbf{J = \alpha_r}$, because RETE saves passed instances in its Alpha nodes.

$$\mathbf{F = A_{R(m,\,n)} - A_{A(m,\,n)} = (B_r - B_a) + (n - m)(B_r\alpha_r - B_a\alpha_a) - \alpha_r} \qquad \textbf{(13)}$$

$$\mathbf{F \geq 0 \Leftrightarrow n(B_r\alpha_r - B_a\alpha_a) \geq m\,(B_r\alpha_r - B_a\alpha_a) + \alpha_r - B_r + B_a} \qquad \textbf{(14)}$$

- If $\mathbf{(B_r\alpha_r - B_a\alpha_a) = 0}$ then **(14)** becomes

$$\mathbf{\alpha_r - B_r + B_a \leq 0} \quad \textbf{(14*)}$$

Because $\mathbf{\alpha_r \geq B_r}$ and $\mathbf{B_a \geq 0}$, **(14*)** is only satisfied, when $\mathbf{\alpha_r = B_r = B_a = 0}$. In this case, $\mathbf{F = 0}$

- If $\mathbf{(B_r\alpha_r - B_a\alpha_a) > 0}$

$$F \geq \mathbf{0} \Leftrightarrow \mathbf{m} \leq \frac{n(B_r\alpha_r - B_a\alpha_a) - (\alpha_r - B_r + B_a)}{B_r\alpha_r - B_a\alpha_a} = n - \frac{\alpha_r - B_r + B_a}{B_r\alpha_r - B_a\alpha_a} \equiv \mathbf{n - M} \qquad \textbf{(15)}$$

Under the condition that $\mathbf{(B_r\alpha_r - B_a\alpha_a) > 0}$, ADINE's performance is better than the RETE's one ($\mathbf{F \geq 0}$), when a fact is fed to an Alpha node m, with $\mathbf{1 \leq m \leq n - M}$ and

$$M \equiv \frac{\alpha_r - B_r + B_a}{B_r\alpha_r - B_a\alpha_a} \leq \frac{\alpha_r}{B_r\alpha_r - B_a\alpha_a} \qquad \textbf{(16)} \qquad \text{because } \mathbf{B_a \leq B_r}$$

Now, let us find an upper boundary for M.

$$\frac{\alpha_r}{B_r\alpha_r - B_a\alpha_a} \le 1 \qquad \textbf{(17)}$$

$$\Leftrightarrow \alpha_r \le B_r\alpha_r - B_a\alpha_a$$
$$\Leftrightarrow \alpha_r(B_r - 1) - B_a\alpha_a \ge 0$$

The left side of the above inequality can be considered as follows

$$\alpha_r(B_r - 1) - B_a\alpha_a \ge \alpha_r(B_r - 1) - B_a\alpha_r \ge 0$$

$$\Leftrightarrow \alpha_r(B_r - B_a - 1) \ge 0$$
$$\Leftrightarrow (B_r - B_a - 1) \ge 0$$
$$\Leftrightarrow B_r - B_a \ge 1 \qquad \textbf{(19)}$$

(19) is satisfied, that means the average size of a RETE's Beta node is at least 1 byte bigger than the average size of a ADINE's Beta node. This condition is normally satisfied, because the RETE network has to spend a lot of memory spaces to save joined tokens in Beta memory nodes, whereas the ADINE does not.

If **(19)** is satisfied, then **(17)** is also satisfied and as a result M **≤ 1**

$\Rightarrow$ **n –M ≥ n –1**

We can conclude that the ADINE's performance is better than the RETE's one (**F ≥ 0)** at least when **1 ≤ m ≤ n - 1**. Therefore, only when a token comes into the last Alpha node, the RETE has got a better performance.

Let us see an example. Suppose that

$\boldsymbol{\alpha_a = B_a = 1}$,

$\boldsymbol{\alpha_r = B_r = 2}$

m $\le \frac{3n-1}{3}$

n	m
2	m < 1,67 → m = **1**
3	m < 2,67 → m = **1, 2**
4	m < 3,67 → m = **1, 2, 3**

From the above table, only when a token enters the last Alpha node, the RETE's cost is smaller than the ADINE's cost. At other Alpha nodes, ADINE has smaller cost than RETE. Therefore, in total, by adding a token the ADINE proves a better performance than the RETE and of course also better than the TREAT.

2. Cost of deletion

The process of deleting a token can be seen a process of adding a negative token. Therefore, the cost is the same like adding a new token.

$$\mathbf{D_{A(m,n)} = B_a + (n - m)B_a\alpha_a + J}$$

3. Memory cost

$$\mathbf{M_A} = \sum_{i=1}^{n}\alpha_i + \sum_{i=1}^{n-1}B_i = \alpha_s + B_s \qquad \mathbf{n \geq 1} \qquad \mathbf{(1)}$$

α_s is sum of sizes of all Alpha nodes. Each Alpha node owns at most one token, which indicates that there are successful instances in Success-Partitions. We need only one byte for this token, because the token only contains either **true** or **false**.

$\mathbf{B_s}$ is sum of sizes of all Beta nodes. Each Beta node spends a at most one token of one byte to indicate that there are successful joined instances in the Success-Partitions, two bytes for two BDNs , **k** bytes for FDNs. **k** is the average number of successive Beta nodes that a Beta node has got.

In principle, the memory cost has to involve size of Memory nodes. But they are ignored in mathematical expressions of my algorithm and RETE, because the total size of Memory nodes is the same in both algorithms and our purpose is just to compare memory cost of both algorithms.

In a network with **n** Alpha nodes as in figure 6.1, there are **(n-1)** Beta nodes. The memory cost is calculated as follows:

$$\mathbf{M_{An}} = \sum_{i=1}^{n} \alpha_{Ai} + \sum_{i=2}^{n} P_{Ai} + (2+k)(n-1) + (n-1)d_A \qquad \mathbf{n \geq 1} \qquad \mathbf{(2)}$$

$\mathbf{P_{Ai}}$ is the probability that two input nodes of the $\mathbf{i^{th}}$ Beta node have got tokens. If two input nodes of a Beta node (type *"and"*) have got tokens, the Beta node owns also a token and then it sends a copy of it to its successive Beta nodes.

Each Beta node spends several bytes to save a path for its question request. That is the path from the Beta node to its nearest Alpha node. $\mathbf{d_A}$ is the average length of these paths.

α_{Ai} is the size of the $\mathbf{i^{th}}$ Alpha node

Remember that the memory cost by RETE is as follows:

$$\mathbf{M_{Rn}} = \sum_{i=1}^{n} \alpha_{Ri} + \sum_{i=1}^{n-1} B_{Ri}$$

Now we calculate difference between the two above memory costs

$$\mathbf{M_{Rn} - M_{An}} = \sum_{i=1}^{n} \alpha_{Ri} + \sum_{i=1}^{n-1} B_{Ri} - \sum_{i=2}^{n} P_{Ai} - (2+k)(n-1) - (n-1)d_A - \sum_{i=1}^{n} \alpha_{Ai}$$

$$= \mathbf{n\alpha_R + (n-1) B_R - (n-1) P_A - (2+k)(n-1) - (n-1)d_A - n\alpha_A} \qquad \mathbf{(3)}$$

α_R is average value of α_{Ri}

$\mathbf{B_R}$ is average value of $\mathbf{B_{Ri}}$

$\mathbf{P_A}$ is average value of $\mathbf{p_{Ai}}$, $\mathbf{0 \leq P_A \leq 1}$

α_A is average value of α_{Ai} , $\mathbf{0 \leq \alpha_A \leq 1}$

Normally a fact is expressed or saved in the following common format of text:

Factname (Attribute_1 Value_1) (Attribute_2 Value_2)… (Attribute_t Value_t) .

So at least a fact consists of three elements: a fact name, an attribute and a value. The length of a fact is sum of lengths of its elements. Assume L is the average length of elements within a fact, the minimum length of a fact: $\mathbf{L_f = 3L}$

In RETE, Alpha nodes save facts, which passed its condition. Call **m** average number of facts in an Alpha node, the average minimum size of an Alpha node is equal to $\mathbf{mL_f} = \mathbf{3mL}$.

In RETE, a token, which is saved in a Beta node, consists of at least two facts, because Beta node has performed a join between two facts. Call **h** the average number of tokens in a Beta node, so average minimum size of a Beta node is $\mathbf{2hL_f = 6hL}$.

$\mathbf{M_{Rn} - M_{An} \geq 3nmL + 6h(n - 1)L - (n - 1) - (2 + k)(n - 1) - (n - 1)d_A - n \equiv D_{RA}}$

$$\mathbf{D_{RA} = (3mn + 6h(n - 1))L - (n-1)(1 + 2 + k + d_A + 1) - 1} \qquad \mathbf{(4)}$$

$$\mathbf{D_{RA} \geq 0} \Leftrightarrow (3mn + 6h(n-1))L \geq (n-1)(4 + k + d_A) + 1$$

$$\Leftrightarrow L \geq \frac{(n-1)(4 + k + d_A) + 1}{3mn + 6h(n-1)} \qquad \mathbf{(5)}$$

With $\mathbf{m \geq 1}$ and $\mathbf{h \geq 1}$, we have

$$\frac{(n-1)(4 + k + d_A) + 1}{3mn + 6h(n-1)} \leq \frac{(n-1)(4 + k + d_A) + 1}{3n + 6(n-1)} = \frac{(4 + k + d_A)n - (3 + k + d_A)}{9n - 6} \equiv L_s \quad \mathbf{(6)}$$

Because $\lim_{n \to \infty} L_s = \frac{4 + k + d_A}{9}$

We can conclude that if the average length of elements within a fact $L \geq \frac{4 + k + d_A}{9}$, then the ADINE's memory costs are smaller than the RETE's memory costs.

If we assume that the reasoning network is of a common form shown in figure 6.1 (chapter 6), then we have $k = 1$ and $d_A = 1$. In this case, L must be bigger than $5/9 \cong 0,6$ so that the ADINE's memory costs are lower than the RETE's memory costs. This condition is always satisfied, because one has to use at least one letter to represent an element. Therefore, L is normally bigger than 1.

Chapter 12

Learning from experiences

1. What is experience?

We are very familiar to the word "experience", because we have learned already a lot of experiences in our life. But what is experience? Before answering this question, let us examine an example. One can find in a cookbook how to cook the spaghetti and can follow it easily. But while cooking, everyone may obtain some new knowledge. Someone supposes that, it needs more butter than in the recipe, but the other thinks that a little less butter is better. Although their opinions can be different, but all of them believe that it makes their spaghetti more delicious with their own modified recipe and they remember the modified recipe as their own experience.

So on my opinion experience is the knowledge
- that one obtained while/after working or interacting with the around environment,
- that can help him to improve his performance, and
- that can be different from one to one.

While working one can get a lot of new knowledge, but one call it experience only if one can use it to improve his performance in the future.

The process that one obtains experiences usually happens as following:
- One is doing something and he must ***understand what*** **he is** ***doing***
- he ***observes*** his works and finds out some ***disadvantages***, which should be eliminated or improved.
- he ***finds out a way*** **to** ***improve the work*** and ***remembers it as experience*** for the next usage.

The italic, bold words express important factors in the process of learning from experiences of the human being. Basing on these factors, we tries to find out a general model of learning from experiences, which can be applied in the computer world.

2. A model, which supports learning from experiences

In the literature of Artificial Intelligence (AI), there are a lot of kinds of machine learning, for example Supervised learning, Clustering, Reinforcement (Lozano-Perez T., 2003)… This classification bases on the kinds of problems, which machine learning copes with. On my opinion, we can divide them into 2 categories: action-oriented learning and result-oriented learning.

Action-oriented learning:

Some good examples (Lozano-Perez T., 2003) for this category are

- learning how to pronounce words
- learning how to throw a ball

The main character for this category is that at the beginning, the system or the program owns only primitive actions to reach a goal or to solve a problem. Then the program follows a progress of "Learn from Experiences" repeatedly. During this progress, basing on given input/output pairs, the program modifies its actions by itself, so that the quality of its outputs will be improved. In this case, the experiences are the modified actions.

To build a learning system of this category, one needs to build a mathematical model, which expresses,

- What system's behaviors will be changed?,
- How to implement these changes?,
- How to evaluate them?,
- How to employ prior knowledge/experience?…

These factors depend on which specific problem that the system has to solve. In the AI world, the problems of machine learning are very diversified. So development of a common model of learning for this category can be performed only in a general level. In

this section we do not intend to go further in this category of machine learning, because our aim is to improve the performance of our Web-based rules-based consultation system through learning from experiences, and the kind of its machine learning belongs to the second category, which is presenting in next part.

Result-oriented learning:

The example, which was presented in the introduction part, is the typical one for this kind of learning. As we known, the goal of the action-oriented learning is to improve the quality of the results/output values through modifying internal actions of the system. Otherwise, the goal of result-oriented learning is to improve the speed of program through saving and reusing input-output pairs as experiences. In this case, the program owns fixed actions to solve its problem. The ability that its actions can be improved during working is not supported. The result-oriented learning can be applied only when:

- the program needs much time to perform its actions
- the amount of its input values is limited.

Experiences are pairs of input/output values. So when an input value comes in, the program is checking, whether the corresponding calculated output value exists or whether it has experience with this input value. If yes, the program does not perform its actions on the coming input value, but just takes from its experiences the corresponding output value and sends it out.

In next part, we concentrate on building a general model for this kind of learning from experiences.

Basing on the above analyses, we suggest a model for computer programs, which can support the intelligent ability of learning from result-oriented experiences by itself. See the figure 11.1.

Let us assume that we have a program, which consists of three modules A, B, C. A module is a collection of functions, which fulfill a determined task together. The "Control Unit" plays the role as the head of the program. It has got the knowledge about the modules, and decides when a module is allowed to perform its task.

After a module receives the signal of "enable" from the Control Unit, it is allowed to run its functions. The Control Unit also lets the module know where to get its input data through the "input-addr.", and where to store its output data through the "output-addr.". After finishing its task, the module sends the signal of "finished" to the Control Unit.

The Control Unit can measure the time that each module needs to do its task. The working time of a module begins when the Control Unit sends the signal of "enable" to the module, and it ends when the module sends the signal of "finished" to the Control unit. If it is long, a process of learning experience (LE) can be applied to the module. From then on, before the module is enabled to work, the Control Unit let the LE check if the corresponding experience exists. If yes, the current module is bypassed, the LE puts the experience into the output address, and the next module will be enabled to work. If no, the current module is allowed runs, and its output result will be saved by LE as an experience, which corresponds to its input(s), for the next usage.

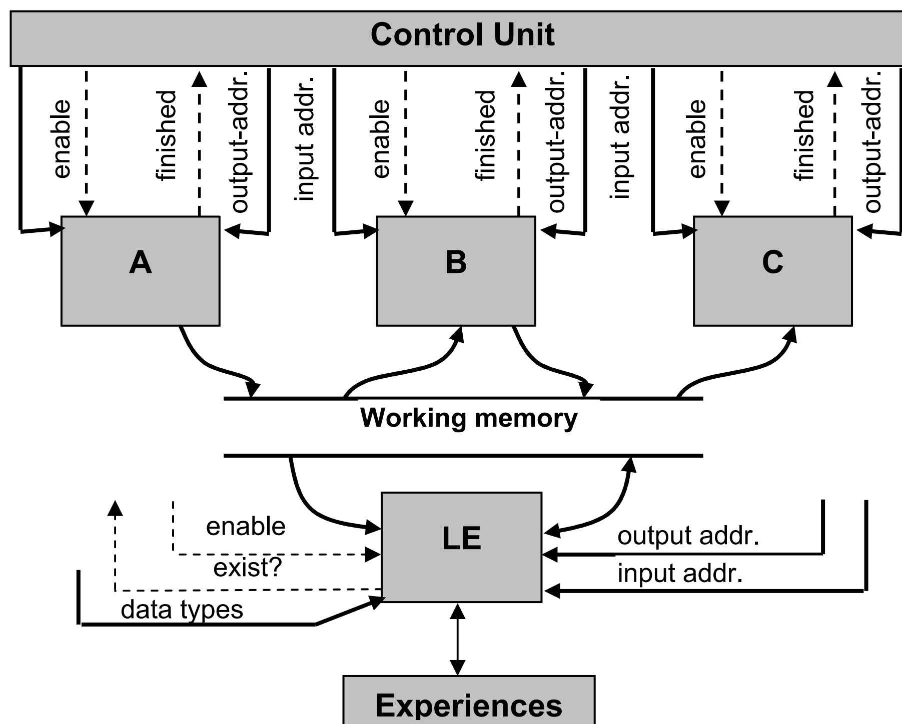


Figure 11.1: a general model of learning from experience

With this model, the Control Unit can learn from experiences as a human being, because:

- It has knowledge about what it is doing. It knows which modules should be run next.
- It can measure the working times of modules and finds out which module is working slowly and should be improved.
- It can save experience for the next usage in the future. It will improve the performance of the observed modules.

We can apply this model into individual modules to increase the ability of learning from experiences. That means there are also a Control Unit and a Learning Experiences unit in each module. The inside LE has a higher priority than the outside LE.

An experience is always related with a context. The context is something to specify when to use this experience. Inputs of a module can be seen as the context for experiences, which correspond to its outputs.

These experiences are result-oriented. So when saving experiences, the LE is only interested in the result and do not care about how to reach the result. So that LE can save experiences, the Control Unit has to give it the knowledge of data types, which inputs and outputs of the observed module use. The LE should have the ability to save and to recover experiences, which can be in the complex data types such as class or a network of classes.

3. Applying learning from experience to improve the performance

Basing on the above ideas, we would like to present a model for my inference engine, which can learn from result-oriented experience.

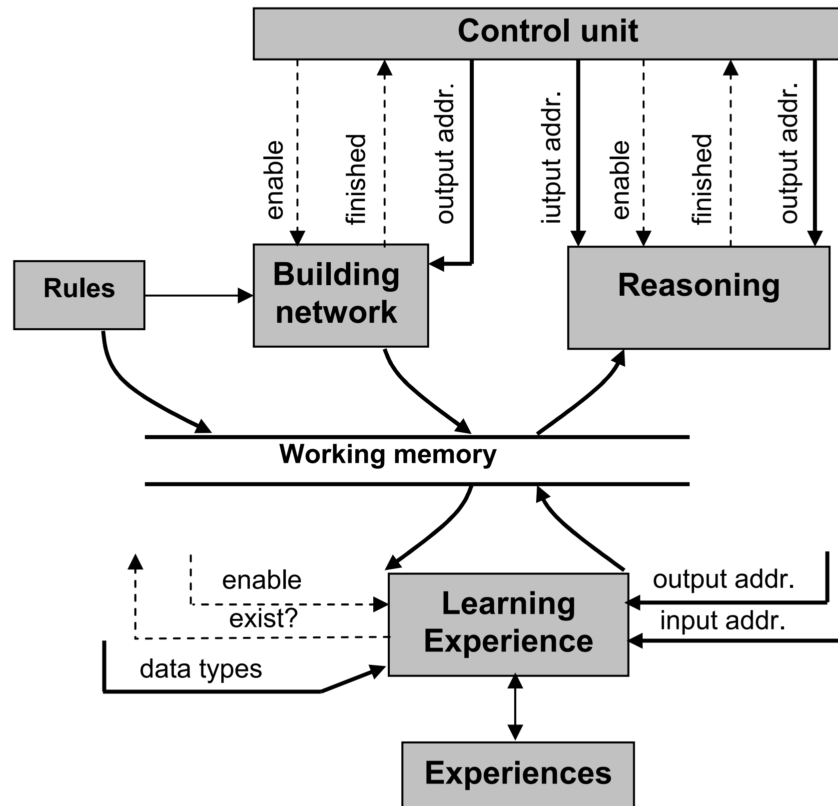


Figure 11.2: Learning from experiences in adaptive inference engine

The first duty of the Control Unit is to find down which module is working slowly and then to apply the "*Learning Experience*" on the module. In this case, the module, whose performance should be improved, is the "*Building Network*" module.

At the beginning, there is no experience. Therefore the reasoning network will be built from original rules by the module "*Building network*". After the reasoning network has been built, experience will be saved by the "*Learning Experience*" (LE).

On next time of starting, if there is not any change on the original rules, the experience will be used to build the reasoning network. If the original rules have been modified, experience must be updated.

4. Implementation

In my implementation of an adaptive inference engine, experiences are mainly nodes of the reasoning network. The Extensible Markup Language (XML) is used as a medium to save and transfer experience.

For illustration, we would like to show you an example, how to save an instance of the class MemoryNode, which is presented in the chapter 9:

```
<Class>
<Type>Memory</Type>
<Factname>Factname</Factname>

<AttributeNames>
     <Type>Vector</Type>
     <Value> value1</Value>
     <Value> value2</Value>
     <Value> value3</Value>
     <Value> value4</Value>
</AttributeNames>

<AttributeValues>
     <Type>Vector</Type>
     <Value> value1</Value>
     <Value> value2</Value>
     <Value> value3</Value>
     <Value> value4</Value>
</AttributeValues>

< LinkedFactNames >
     <Type>Vector</Type>
     <Value> value1</Value>
</ LinkedFactNames >

< LinkedFactValues >
     <Type>Vector</Type>
     <Value> value1</Value>
</ LinkedFactValues >

< LinkedAttributeNames >
     <Type>Vector</Type>
```

```
     <Value> value1</Value>
     <Value> value2</Value>
</ LinkedAttributeNames >

< LinkedAttributeValues >
     <Type>Vector</Type>
     <Value> value1</Value>
     <Value> value2</Value>
</ LinkedAttributeValues >

< AlphaNodes >
     <Type>Vector</Type>
     <Value> value1</Value>
     <Value> value2</Value>
     <Value> value3</Value>
</ AlphaNodes>

</Class>
```

It is easy to see that XML offers the ability to save any kind of data of a class. Therefore, we can utilize this ability to save all nodes of the reasoning network as system's experience. In the next start, the reasoning network will be recovered from the experience. It spares the cost of parsing rules from the beginning.

Chapter 13

Conclusion and Future Work

1. Conclusion

The concepts and implementation of an adaptive inference engine for consultation systems have been presented. The inference engine has got important following features:

- Being able to be trained to adapt to knowledge bases that can be prepared in different languages. This feature enables a consultation system to extend its knowledge base easily through integrating external knowledge bases.

- Being able to find out which questions should be asked and suitable reasoning strategy during consultation. This feature helps to reduce the cost of maintaining knowledge base considerably.

- Problems with optimising speed and memory using have been considered. Learning from experience, forecasting future works and new matching algorithm are the keys to solve the problems.

2. Future Work

- The adaptive inference engine can be further developed to serve reasoning uncertainly.

- The adaptive inference engine can be extended to apply to Intelligent Tutor Systems.

References

[ACH92] Acharya, A.; Tambe, M. (1992). Collection-oriented match: Scaling up the data in production systems. Technical Report CMU-CS-92-218, School of Computer Science, Carnegie Mellon University.

[BIL00] Bill, K. (2000). User Modelling for Adaptive and Adaptable Software Systems. Published by Department of Computer Science, University of Maryland , USA

[BRU03] Brusilovsky, P. (2003). Developing adaptive educational hypermedia systems: From design models to authoring tools. In T. Murray, S. Blessing, & S. Ainsworth (Eds.), *Authoring tools for advanced technology learning environment*. Dordrecht, The Netherlands: Kluwer Academic.

[BUC84] Buchanan, B. G.; Shortliffe, E. H. (1984). Rule-based expert systems: the MYCIN experiments of the Stanford Heuristic Programming Project.

[CHA.89] Chase, M. P.; Zweben, M.; Piazza, R. L.; Burger, J. D.; Maglio, P. P.; Hirsh, H. (1989). Approximating learned search control knowledge. In Proceedings of the Sixth International Workshop on Machine Learning, pages 218-220.

[COH90] Cohen, W. W. (1990). Learning approximate control rules of high utility. In Proceedings of the Seventh International Conference on Machine Learning, pages 268-276.

[ETZ90] Etzioni, O. (1990). A Structural Theory of Search Control. PhD thesis, School of Computer Science, Carnegie Mellon University.

[ETZ93] Etzioni, O. (1993). A structural theory of explanation-based learning.

Artificial Intelligence, 60(1):93-139.

[FOR79] Forgy, C. L. (1979). On the Efficient Implementation of Production Systems. PhD thesis, Computer Science Department, Carnegie Mellon University.

[FOR81] Forgy, C. L. (1981). OPS5 user's manual. Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University.

[FOR82] Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence, 19(1):17-37.

[FRA99] Francisco, M.; Weiming, S.; Douglas, H. N. (1999). MetaMorph: An Adaptive Agent-Based Architecture for Intelligent Manufacturing. International Journal of Production Research, 37(10), 2159-2174.

[GEO01] George, D.; Andriana P. (2001).Machine Learning In Medical Applications. In Lecture Notes in Computer Science, volume 2049, pages 300-307

[GHA81] Ghallab, M. (1981). Decision trees for optimising pattern-matching algorithms in production systems. In Proceedings of the Seventh International Joint Conference on Artificial Intelligence, pages 310-312.

[GRA92] Gratch, J.; DeJong, G. (1992). COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In Proceedings of the Tenth National Conference on Artificial Intelligence, pages 235-240.

[GRE92] Greiner, R.; Jurisica, I. (1992). A statistical approach to solving the EBL utility problem. In Proceedings of the Tenth National Conference on Artificial Intelligence, pages 241-248.

[GUP89] Gupta, A.; Forgy, C.; Newell, A. (1989). High-speed implementations of rule-based systems. ACM Transactions on Computer Systems, 7(2):119-

146.

[HAU00] Haun, M. Wissensbasierte Systeme : eine praxisorientierte Einführung (Expert-Verlag, 2000)

[HOL92] Holder, L. B. (1992). Empirical analysis of the general utility problem in machine learning. In Proceedings of the Tenth National Conference on Artificial Intelligence, pages 249-254.

[JEN98] Jennings, N.R.; Wooldridge, M. (Ed.) (1998). Agent Technology: Foundations, Applications, and Markets. Springer-Verlag, ISBN 3-540-63591-2, Berlin

[JOS05] Joseph G.; Garry R. (2005). Expert Systems Principles and Programming. Thomson Course Technology.

[KIM93] Kim, J.; Rosenbloom, P. S. (1993). Constraining learning with search control. In Proceedings of the Tenth International Conference on Machine Learning, pages 174-181.

[KUO92] Kuo, S.; Moldovan, D. (1992). The state of the art in parallel production systems. Journal of Parallel and Distributed Computing, 15:1-26.

[LIC89] Lichtman, Z. L.; Chester, D. L. (1989). A family of cuts for production systems. In Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence, pages 353-357.

[LOZ03] Lozano-Perez T.; Kaelbling L.(2003). Artificial Intelligence (Massachusetts Institute of Technology)

[MAR88] Markovitch, S.; Scott, P. D. (1988). The role of forgetting in learning. In Proceedings of the Fifth International Conference on Machine Learning, pages 459-465.

[MAR93] Markovitch, S.; Scott, P. D. (1993). Information _ltering: Selection mechanisms in learning systems. Machine Learning, 10(2):113-151.

[MCD78] McDermott, J.; Forgy, C. (1978). Production system conflict resolution strategies. In Waterman, D. A. and Hayes-Roth, F., editors, Pattern-Directed Inference Systems, pages 177-199. Academic Press, New York.

[MIK97] Mike, P. (1997). Adaptive Web Sites: an AI Challenge. In Proceeding of IJCAI 1997, pages 16-23.

[MIN88a] Minton, S. (1988a). Learning Effective Search Control Knowledge: An Explanation-Based Approach. Kluwer Academic Publishers, Boston, MA.

[MIN88b] Minton, S. (1988b). Quantitative results concerning the utility of explanation-based learning. In Proceedings of the Seventh National Conference on Artificial Intelligence, pages 564-569.

[MIR90] Miranker, D. P. (1990a). TREAT: A New and Efficient Match Algorithm for AI Production Systems. Morgan Kaufmann, San Mateo, CA.

[MIR90b] Miranker, D. P.; Brant, D. A.; Lofaso, B.; Gadbois, D. (1990b). On the performance of lazy matching in production systems. In Proceedings of the Eighth National Conference on Artificial Intelligence, pages 685-692.

[NIC98] Nicholas R. J.; Katia S.; Michael W.(1998). A Roadmap of Agent Research and Development. Journal of Autonomous Agents and Multi-Agent Systems, volume 1, pages 7-38.

[PAR03] Park, O.; Lee, J. (2003). Adaptive instructional systems. In D.H. Jonassen (Ed.), *Handbook of research for educational communications and technology*, (pages 651-685). Mahwah, NJ: Lawrence Erlbaum.

[PAT94] Pattie M. (1994). Modelling Adaptive Autonomous Agents. Journal of Artificial Life, volume 1, number 9

[PEI00] Pei W. (2000). Logic of Learning (Internet Resource, http://citeseer.ist.psu.edu/wang00logic.html)

[PER88] Perlin, M. W. (1988). Reducing computation by unifying inference with user interface. Technical Report CMU-CS-88-150, School of Computer Science, Carnegie Mellon University.

[PER89] Perlin, M.; Debaud, J.-M. (1989). Match box: fine-grained parallelism at the match level. In Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence, pages 428-434.

[PER90a] Perlin, M. (1990a). Scaffolding the Rete network. In Proceedings of the Second IEEE International Conference on Tools for Artificial Intelligence, pages 378-385.

[PER90b] Perlin, M. (1990b). Topologically traversing the Rete network. Applied Artificial Intelligence, 4(3):155-177.

[PER91a] Perlin, M. (1991a). Incremental binding-space match: The linearized MatchBox algorithm. In Proceedings of the Third International Conference on Tools for Artificial Intelligence, pages 468-477.

[PER91b] Perlin, M. W. (1991b). Automating the Construction of Efficient Artificial Intelligence Algorithms. PhD thesis, School of Computer Science, Carnegie Mellon University.

[PER92] Pérez, M. A. ; Etzioni, O. (1992). DYNAMIC: A new role for training problems in EBL. In Proceedings of the Ninth International Conference on Machine Learning, pages 367-372.

[PER92] Perlin, M. (1992). Constraint satisfaction for production system match. In Proceedings of the Fourth IEEE International Conference on Tools with Artificial Intelligence, pages 28-35.

[PHA05] Phan C. C.; Hunger A. (2005). Self-adaptable Inference Engine. In Proceeding of the 4th WSEAS (World Scientific and Engineering Academy and Society) International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED).

[PHA05] Phan C., C.; Hunger, A. (2005). Self-adaptable Inference Engine. In Journal of the 4th WSEAS (World Scientific and Engineering Academy and Society) International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED)

[PHA05] Phan C., C.; Hunger, A. (2005). Learning by experiences in a Web-based rules-based consultation system as a possibility to improve its performance. In Proceeding of the 9th IASTED (International Association of Science and Technology for Development) International Conference on Artificial Intelligence and Soft Computing (ASC)

[RAN96] Randall J.; Alberto M. S.; David S.(1996). The Peaks and Valleys of ALPS: an Adaptive Learning and Planning System for Transportation Scheduling. In Proceeding of International Conference on AI Planning Systems(AIPS-96)

[ROS05] Rosu, M.; Phan C., C.; Hunger, A. (2005). An Intelligent Web Based Dialogue For Rule Based Expert System - An Ontological Approach. In Proceeding of the ICAI'05 (International Conference on Artificial Intelligence)

[SCA86] Scales, D. J. (1986). Efficient matching algorithms for the Soar/Ops5 production system. Technical Report KSL-86-47, Knowledge Systems Laboratory, Department of computer Science, Stanford University.

[SCH86] Schor, M. I.; Daly, T. P.; Lee, H. S.; Tibbitts, B. R. (1986). Advances in Rete pattern matching. In Proceedings of the Fifth National Conference on Artificial Intelligence, pages 226-232.

[STE00] Steven A. H.; Stephanie F. (2000). Architecture for an Artificial Immune System. In journal of Evolutionary Computation, volume 8, number 4, pages 443-473

[STO86] Stolfo, S. J.; Miranker, D. P. (1986). The DADO production system machine. Journal of Parallel and Distributed Computing, 3:269-296.

[TAM88] Tambe, M.; Kalp, D.; Gupta, A.; Forgy, C.; Milnes, B.; Newell, A. (1988). Soar/PSME: Investigating match parallelism in a learning production system. In Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems, pages 146-160.

[TAM90] Tambe, M.; Newell, A.; Rosenbloom, P. S. (1990). The problem of expensive chunks and its solution by restricting expressiveness. Machine Learning, 5(3):299-348.

[TAM91a] Tambe, M. (1991a). Eliminating Combinatorics from Production Match. PhD thesis, School of Computer Science, Carnegie Mellon University.

[TAM91b] Tambe, M.; Kalp, D.; Rosenbloom, P. (1991b). Uni-Rete: Specializing the Rete match algorithm for the unique-attribute representation. Technical Report CMU-CS-91-180, School of Computer Science, Carnegie Mellon University.

[TAM92] Tambe, M.; Kalp, D.; and Rosenbloom, P. S. (1992). An efficient algorithm for production systems with linear-time match. In Proceedings of the Fourth IEEE International Conference on Tools with Artificial Intelligence, pages 36-43.

[TAM94] Tambe, M.; Rosenbloom, P. S. (1994). Investigating production system representations for non-combinatorial match. Artificial Intelligence, 68(1):155-199.

[TOR90] Toru I.; Makoto Y.; Les G. (1990). An Organizational Approach to Adaptive Production Systems. In Proceeding of the National Conference on Artificial Intelligence, pages 52-58.

[TZV89] Tzvieli, A.; Cunningham, S. J. (1989). Super-imposing a network structure on a production system to improve performance. In Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence, pages 345-352.

[WAT] Waterman, D. A.; Hayes-Roth, F. Editors, Pattern-Directed Inference Systems, pages 177-199. Academic Press, New York.

[WEI99] Weiss, G. (Ed.) (1999). Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence, The MIT Press, ISBN 0-262-23203-0, Cambridge, Massachusetts