Back to Basics – Backward Chaining: Expert System Fundamentals

By Dustin Huntington

Introduction

Backward chaining is an incredibly powerful yet widely misunderstood concept, yet it is key to building many types of expert systems. It is particularly well suited for interactive systems that emulate the conversion between a user and a human expert. Backward chaining enables systems to know what question to ask and when. It facilitates the dismantling of complex problems into small, easily defined sections, which the system automatically uses if needed. An understanding of backward chaining is fundamental to building an expert system with the most popular development shells.

Since it is easy to implement, backward chaining is the default method of operation in many expert system tools. If the system must determine the value of a variable, and a rule for deriving that value exists, backward chaining can automatically execute the rule to obtain the value. If this rule requires additional information before it can succeed, then the system can execute additional rules, recursively if necessary. Unfortunately, this simplicity of implementation enables developers to implement systems without fully considering the operation of the inference engine; often negatively affecting performance and efficiency.

Goal Driven, another commonly used term for describing backward chaining, refers to the method used to process the rules. *Data Driven*, the other common approach, is associated with *Forward Chaining*. These terms add confusion; referring to how the inference engine uses the rules, and not to any required system architecture. A backward chaining system can be driven by a block of data supplied at the start – and there are often good reasons to do so. Likewise, a data driven system may appear to interact with the user in a manner similar to a backward chaining system.

Goal Driven System

A goal driven system always has a *Goal List* to attempt to complete. This list, which is fundamental to backward chaining, dynamically adds new goals to its top, pushing the other goals down in the list. A key concept is the system only works on the top goal, which once achieved drops off the list and the next goal becomes the top and active goal. The system is finished once it removes all goals from the list.

The inference engine actively attempts to *achieve* the top goal, which usually requires the determination of a value for a variable. To obtain that value, the inference engine checks the rules to establish if any could derive a value for that variable. This requires an If/Then rule that assigns a value to the variable in the *THEN* part of the rule. If such a rule is found, that rules *IF* portion is tested to determine if it is true.

Determining whether the IF portion is true typically requires data for other variables. Values for these other variables may already be available, making it possible to determine if the rule is true or false. Alternatively, if the value needed to evaluate the rule is unknown, then that variable becomes the new top-level goal and the inference engine looks for the rules that might assign it a value. This is one way the system dynamically adds new goals (variables) to the top of the goal list.

If no rule is available to assign a value to the top-level goal variable, the system asks the user directly. The user's input sets the value of that goal variable, dropping it off the goal list. The next goal in the list becomes the top goal, with this additional information to try to achieve that goal. This process continues with the adding and removing of goals from the list until all goals are gone.

Simple Example

The following example shows how backward chaining adds goals to the goal list, and how it can make a system modular. The sample system helps first-level support staff prioritize support requests by ensuring that certain customers receive priority service and a response within 4 hours.

When building a Backward Chaining system, start with the highest-level rules and add additional detailed rules. At the highest level, the system is one rule:

```
IF
        The customer should receive priority service
THEN
        Call within 4 hours
```

Typically, a command in the expert system defines the initial top-level goal. In this case, it is: “Determine if the response should be within 4 hours.”

Goal List:
1. Determine if the response should be within 4 hours

The system looks through the rules (only 1 rule so far) to find rules with the top goal in the *THEN* part. This rule is tested since it could potentially set the value for the goal.

To determine if the relevant rule is true, and can set a value for the goal, the system must determine whether the *IF* conditions are true. That requires determining whether "The customer should receive priority service”, which becomes the new Top-Level Goal.

Goal List:
1. Determine if the customer is a Priority customer
2. Determine if the response should be within 4 hours

Remember, ONLY the top-level goal matters to the system. The inference engine temporarily stops trying to set a value for the "Respond in 4 hours" goal, and concentrates on the new top Goal, "Priority customer".

Since there are no other rules in the sample system, there is no way of deriving the value so the system must ask the end user. Once the user answers the question, the system knows the value for "The customer should receive priority service", and that goal drops off the Goal list. The Goal list returns to the original goal of determining if the response should be within 4 hours. If the system determines that this is a priority customer, the one rule in the system determines the value for that Goal, and the session is complete. If it cannot determine that this is a priority customer, there are no rules in the system for setting a value for the “respond in 4 hours” variable.

Simple Example – Adding Clarification

In reality, asking the typical first-level support staff if “The customer should receive priority service” is not reasonable. The staff typically does not have the background or corporate knowledge to answer correctly and consistently. The system needs additional rules to establish this value based on lower level questions, which the intended user can answer correctly and consistently.

The addition of more specific rules, which the inference engine automatically uses, makes the system much more capable and less subjective. Rules specifying when a customer should receive priority service can ask the user more appropriate questions and derive needed information.

In this case, add 3 rules that identify a priority customer:

```
IF
   The customer purchases are over $250,000 per year
THEN
   The customer should receive priority service

IF
   The customer works for a Partner company
THEN
   The customer should receive priority service
IF
   The customer's company has significant growth potential
THEN
   The customer should receive priority service
```

When the system runs, the same initial goal starts the system. The inference engine finds this first rule and tests it, setting the new Top-Level Goal to determine if the customer is a “priority customer.”

The system now has rules to determine if the customer is a priority customer instead of directly asking the user.

The engine tests each rule in order. The first rule found is:

```
IF
   The customer purchases are over $250,000 per year
THEN
   The customer should receive priority service
```

The *IF* condition in that rule becomes the new top-level Goal:

Goal List:
1. Determine if the purchases are over $250,000
2. Determine if the customer is a Priority customer
3. Determine if the response should be within 4 hours

The system automatically searches for any rule that would set a value for the "Purchases over $250,000" variable. Since no such rule exists, the system must obtain the data from the user or an external source such as a database. This is a more reasonable question to ask a user, particularly if they have access to a sales database and can check the sales volume. In practice, an expert system would automatically determine this answer by interfacing directly to external databases.

If the request for sales volume returns the customer's purchases are $20,000, this determines the variable value, the top-level goal on purchase amount is now satisfied, and it drops off the Goal List. It des not matter that the rule that put the goal on the list is false; the system is only working on the top-level goal.

Goal List:
1. Determine if the customer is a Priority customer
2. Determine if the response should be within 4 hours

The next Goal in the list, "Priority Customer" again becomes the Top Level Goal and the associated rule becomes the rule to be tested.

```
IF
    The customer purchases are over $250,000
    per year
THEN
    The customer should receive priority service
```

Based on the value of the customer's purchases of $20,000, the rule can be determined to be false, so it will not fire or indicate anything about the Top-Level Goal variable. This rule is of no value in achieving the top-level goal. However, there is another rule in the system that also provides information on the top-level Goal variable:

```
IF
    The customer works for a Partner company
THEN
    The customer should receive priority service
```

The variable used in the *IF* part of that rule becomes the new Top-Level Goal.

Goal List:

1. Determine if customer is from a Partner company
2. Determine if the customer is a Priority customer
3. Determine if the response should be within 4 hours

There are no other rules that allow the engine to derive "Partner Company", so it asks if the customer works for a Partner company. This is a reasonable question to ask since the number of partner companies is probably reasonably small.

In this example, assume that the customer is not from a Partner company, so it drops that top Goal. After the Partner company rule, the next rule to test is:

```
IF
    The customer's company has significant
    growth potential
THEN
    The customer should receive priority service
```

Since the typical first level support operator may not know the answer, the system needs a few additional rules to explain which customers have "significant growth potential". The Sales Manager could write a few rules describing how to identify a company with significant growth potential. These add the Sales Manager knowledge, on his specific aspects of the problem, to the system. These might be rules such as:

```
IF
    The company is a Fortune 100 company
THEN
    The customer's company has significant
    growth potential

IF
    The company has been a customer for many
    years
THEN
    The customer's company has significant
    growth potential
```

To determine if the customer's company has significant growth potential, the engine automatically calls these new rules which asks the user more objective questions. As mentioned before, in a real system this data might come from other sources such as a database with information on customer companies.

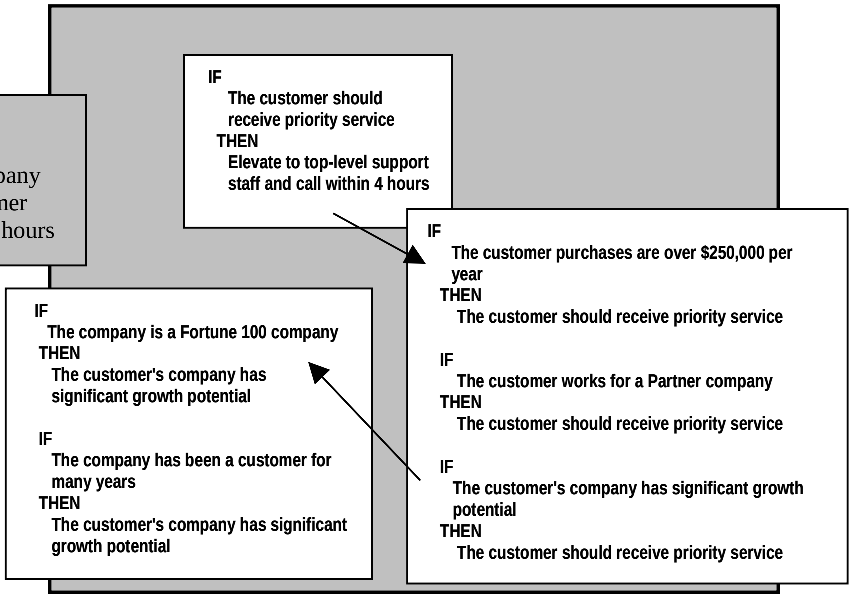
IF
The customer should receive priority service
THEN
Elevate to top-level support staff and call within 4 hours
IF
The customer purchases are over $250,000 per year
THEN
The customer should receive priority service
IF
The customer works for a Partner company
THEN
The customer should receive priority service
IF
The customer's company has significant growth potential
THEN
The customer should receive priority service
IF
The company is a Fortune 100 company
THEN
The customer's company has significant growth potential
IF
The company has been a customer for many years
THEN
The customer's company has significant growth potential

Fig. 1 How individual rules call other blocks of rules

Backward Chaining enables the decomposition of complex problems into smaller modules. By starting with the highest-level description that solves the problem, this quickly leads to the creation of a working system. If the questions asked by the system are not at the appropriate level for the intended end user, adding additional rules enables deriving the information using simpler questions.

A system can be started with high level rules describing the decision-making process and expanded to whatever level is required by adding blocks of rules that cover specific decision details.

The system can reuse rule blocks in multiple places. For example, if multiple places in the system must know if a customer should receive priority service, perhaps to determine what method to use for product shipments, the inference engine automatically invokes the same rule block to derive the value.

In the future, if there are other criteria to determine if a customer is a "priority customer", simply add another rule. The inference engine automatically calls and tests this new rule in any relevant situation, making it very easy to add rules and expand a system.

Comparison of Inference Engines to Traditional Programming

A backward chaining inference engine makes system development and maintenance much easier. When first exposed to IF/THEN rule logic, they are often confused with the simple IF/THEN statements of computer languages such as C and BASIC. However, the inference engine is fundamentally very different and much more powerful.

A BASIC program, for example, allows nested IF/THEN blocks. However, if a program needs to reuse a complex or deeply nested IF/TTHEN relationship in another section of the program, it must duplicate the code, or make it a function. A standard program cannot simply call the necessary section of the computer code just because it exists in the system – yet this is exactly what the inference engine does.

During backward chaining, if any rule assigns a value to variable X, that rule is automatically available whenever other rules being tested need a value of X. Rules can be physically located anywhere in the system, and there is no explicit linking of rules. Having two rules use the same variable is all that the inference engine needs to link them.

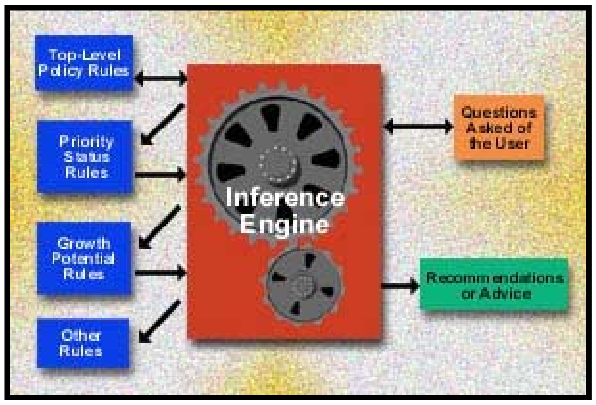

Fig. 2 The role of the Inference Engine in Backward Chaining

This rather "free-form" nature of the rules makes development very simple. Provide the IF/THEN rules necessary to make a decision and tell the system what to derive, and the inference engine does the rest. The engine asks questions in a focused manner, and only asks the relevant questions that it cannot derive from other rules. It does not ask unnecessary questions as often seen in traditional programming.

Programmers might look at the IF/THEN rules in a simple system, such as those demonstrating the above concepts, and feel they could program a similar system in a few lines of Visual Basic. For very simple systems, that is true. However, if the system grows even a modest amount, the problem rapidly becomes very complex to program using traditional techniques. It requires far more than just nested IF/THEN statements to handle cases where there are multiple sources to derive a fact, multiple uses of the same rules, or many levels of derivation that may depend dynamically on user input.

Traditional code can rapidly becomes very complicated when handling any of these situations. It becomes even more complex when adding new rules and maintaining the system. Adding a single new rule can have ripple effects across the entire system and this is considerably more complex when the programmer has not seen the code for a while or someone other than the original programmer is maintaining it. Implementing a new heuristic using traditional programming is often quite difficult and if not added correctly, often has side effects that are difficult to detect and fix.

For significantly complex systems, the most efficient approach is separating the rules from the actual program code and handling the rules more as data. By writing a program to process the rules as data, this allows changes to the rules without changing the program. Changes to the programs that process the rules do not necessarily affect the rules, and changes to the rules usually do not affect the program. This partitions the data from the program and greatly reduces the effects of the changes in one on the other, and the corresponding testing when there are rule changes. This is, in effect, an Inference Engine – though perhaps not a full featured one.

Working with an existing inference engine is far easier and much more productive. Using a proven inference engine that has already been tested and should be relatively error free is similar to using a well-utilized programming library. Programmers will benefit by testing done by those that have already worked with the inference engine on previous projects. In addition, many of the debugging tools that simplify rule-level debug are already in the inference engine where traditional programming environments assist with code level, not rule-level debug. Attempting to build a robust knowledge automation system by traditional programming techniques is typically much more expensive and far less likely to succeed or be maintainable.

Forward Chaining

As mentioned earlier, many inference engines use Forward Chaining. Some expert systems support hybrid approaches where the basic system uses forward chaining, but allows backward chaining to derive needed values. This combination provides the best of both approaches and is often very effective.

Conceptually, forward chaining is much simpler than backward chaining. The system simply tests the rules in the order that they occur, so rule order is crucial. If the system needs a variable to determine if a rule is true or false, and the value of that variable is unknown, the system immediately asks the user for the value without any attempt to derive its value. If it is determined that a rule is true, the assignments in the rules THEN part add data to what the system knows and can use in subsequent rules. If a rule is determined to be false, it discards that rule.

Forward chaining systems are data driven since the system simply processes a set of data by the rules with no specific defined Goal. Forward chaining is often faster than backward chaining since it does not have the overhead of dynamically determining which rule to activate, but it does not exclude blocks of rules and logic that are not actually needed. Forward chaining asks less focused questions and is not as good an emulation of a human interaction with an expert. Unlike backward chaining systems, the order of questions in a forward chaining system is very dependent on rule order.

Programmers often apply a data driven concept to systems, such as monitoring systems, where a set of data is available at the start of a session. The system applies the rule logic to the data to produce results, but the order in which the system uses the data does not matter. Programmers often use forward chaining for these types of systems. Backward chaining is often a better choice for problems that benefit by being modularized or handled from top-level logic down.

Backward chaining systems are often easier to develop than forward chaining. A human expert intuitively thinks: "The cause could be X. To determine that, I need to know the value of Y, but to find Y, I first need to know Z". This type of logic is similar to one that a backward chaining system produces from rules.

Summary

Backward chaining is an effective method for building any type of system from knowledge management, to help desk, to diagnostic, to decision support. With a clear understanding of how backward chaining operates, systems can be modularized and rapidly constructed. These systems can easily combine the expert knowledge of multiple individuals into a single coherent system. Separating domain-knowledge (rules) from the program (inference engine) greatly reduces the amount of work required to create an effective program. Using an existing inference engine eliminates the need to develop a major part of the program from scratch.