Report 81-04
Stanford -- KSL

Maxims for Knowledge Engineering.
David R. Barstow, Bruce G. Buchanan,
May 1981

HPP-81-4 May 1981

MAXIMS FOR KNOWLEDGE ENGINEERING

*David R. Barstow**
Bruce G. Buchanan

Abstract

Knowledge engineering is still more of an art than a science. Based on the collected experience of many knowledge engineering projects, certain rules of thumb can be identified. In the list presented here, we have tried to capture the art as it exists at the beginning of the 1980's. (The first such list known to the authors was presented by E. A. Feigenbaum at IJCAI-73 at Stanford. Many of his heuristics still apply.)

*David Barstow is a member of the Schlumberger-Doll Research Laboratory, Ridgefield, CT. This memo is also being published by SDR as AI Memo number 10.

1. Introduction

In the course of building expert systems, knowledge engineers have developed intuitions about how best to proceed, heuristics to keep in mind when building a system for a particular task. The following maxims represent a distillation of some of these intuitions and heuristics. They are not necessarily full of great insight. In many ways, they are similar to well-known guidelines for building other types of software. But we give them here with the hope that they will be helpful to future knowledge engineers.

2. On Task Suitability

Focus on a narrow specialty area that does not involve a lot of common sense knowledge.

Artificial intelligence techniques have not yet progressed to the point where computers are adept at common sense reasoning. And to build a system with expertise in several domains is extremely difficult, since this is likely to involve different paradigms and formalisms.

Select a task that is neither too easy nor too difficult for human experts.

Based on current techniques "too easy" might be defined as "taking a few minutes"; "too hard" might be defined as "requiring a few hours". Hopefullly, our abilities to build expert systems will grow, but for now these rules of thumb are probably right. Don't aim for a program that is an expert in domain D; rather aim for an expert performing task T within domain D. The number of relevant concepts should be bounded and of the order of many hundred.

Define the task very clearly.

This is both obvious and crucial. At the outset, you should be able to describe the nature of the inputs and outputs rather precisely, and the expert should be able to specify many of the important concepts and relations. You should have access to many specific examples of problems and solutions.

Commitment from an articulate expert is essential.

After all, knowledge engineering is a process of rendering human expertise into machine-usable form. You can't hope to do this unless you have a long-term commitment from such an expert.

3. On Building the Mark-I System

Record a detailed protocol of the expert solving one prototypical case.

Many questions can be answered from the protocol later. It provides a list of vocabulary terms and hints about strategies.

Start building a "Mark-I" version of the expert system as soon as the first example is well

understood.

A common error is waiting until the knowledge base is close to complete before programming. Make an initial commitment to a simple representation and a simple control structure. Work with those as vehicles for discovering what new knowledge needs to be added. As a rough guideline, you should plan to have your Mark-I version running within months, rather than years, of the start of your project.

Work intensively with a core set of representative problems.

Pick a half dozen or so target problems including both the input and the desired output. Pick several simple ones that focus on different aspects of the task, so that the resulting system will not be too special-purpose. Get the rules right for these problems before looking at other problems. These target problems can also be used as milestones for the implementation effort.

Identify and separate the parts of the problem that have caused trouble for AI programs in the past.

Don't avoid working on them, but try to keep them out of the Mark-I version. Some such problems are: English understanding and generation, geometric or spatial models, complex temporal or causal relations, modal logic, understanding human intentions, reasoning with imprecise concepts.

Build in mechanisms for indirect reference.

Even though it takes more thought, the benefits of increased flexibility will pay off. For example, refer to members of a class by naming the class concept rather than by listing the members explictly.

Aim for simplicity in the "inference engine".

Simple inference engines are more easily built, permitting you to experiment with the rules quite soon. Besides, the point of knowledge engineering is to make as much knowledge explicit as possible. Knowledge encoded in a complex inference engine is not very explicit.

Don't worry about time and space efficiency in the beginning.

Your primary concern should be for building a set of rules, and an inference engine that can solve the task. If you succeed, you can then worry about efficiency.

Find or build computerized tools to assist in the rule writing process.

For example, a rule editor can be immensely valuable. Especially with LISP-based systems, such tools are not overly complex and generally pay off quickly.

Pay attention to documentation.

Even the Mark-I version needs a one or two page description, so your collaborators can run the program on their own.

Don't wait until the informal rules are perfect before starting to build the system.

This is just another way of saying that you should start implementing early. We are repeating it because it is such a common error. Remember: you haven't got much hope of getting the informal rules exactly right anyway, and trying to implement them is a powerful aid to getting them right.

4. On Extending the Mark-I Version

Build a friendly interface to the system soon after the Mark-I is finished.

Once the Mark-I version has demonstrated the feasibility of the approach, you will want to let other people experiment with it, before building the Mark-II version. A friendly interface will maximize the benefit you get from that experimentation.

Provide some capabilities for examining the rule base and the line of reasoning soon after the Mark-I version is finished.

This is another important way to permit useful experimentation.

Provide a "gripe" facility.

Give your users a way to record their complaints when you are not present. Don't make them learn everything about your operating system's mail facility or text editor.

Keep a library of cases presented to the system.

With each set of modifications, run the system on all the library cases to see if old problems are fixed or if new problems are introduced. After the Mark-I version is running, automate the storage, retrieval, and use of library cases.

5. On Finding and Writing Rules

Don't just talk with the expert, watch him/her doing examples.

The details of the expertise are not always apparent when discussing the task and domain in the abstract. His/her actual rules of are much more visible when he/she is solving concrete problems, and examples give you opportunities to ask specific questions about why some action was (or was not) taken.

Use the terms and methods that the experts use.

The knowledge base will be impossible to modify otherwise.

Look for intermediate level abstractions.

Intermediate level concepts are perhaps the most important tool available for organizing the rule base, both conceptually and computationally. In many cases, these concepts may

not be explicitly mentioned by the expert; instead they must be identified by looking for similarities in the ways that the expert describes different concepts.

If a rule looks big, it is.

The printed version of a formal rule is a good indicator of whether the rule actually encodes a single "chunk" of the expert's knowledge. If it "feels big", then it probably is, and should be broken into several smaller rules.

If several rules are very similar, look for an underlying domain concept.

Often, similarities among rules indicates that there is an important domain concept that hasn't been explicated fully. It may not have a name and the expert might not even be aware of it, but it may be useful to make it explicit. Let the expert name the concept.

If you are tempted to escape from your rule formalism into "pure" code resist the temptation for at least a little while.

The goal of your enterprise is to have an explicit machine-usable encoding of an expert's knowledge. "Pure" code is not usually explicit or machine-understandable enough. On the other hand, if using a little code lets you avoid a problem temporarily, don't be too dogmatic about your formalism.

6. On Maintaining Your Expert's Interest

Engage the expert in the challenge of designing a useful tool.

There will be problems and delays. If you have described the system building enterprise as an intellectual challenge, the expert will be more tolerant and helpful than if you have created unreal expectations about how easy it is.

Give the expert something useful on the way to building a large system.

As you build the system, there will be opportunities to both clarify concepts for the expert and to provide computational tools that are of immediate benefit to him/her. Take advantage of these opportunities.

Insulate the expert, as well as the user, from technical problems.

They should never have to worry about operating systems, terminal modems, programming languages, etc.

Be careful about setting yourself up as an expert.

It is very easy to be deluded into thinking you know more about the domain than you really do. Remember: your expert became one only after years of training and experience.

7. On Building the Mark-II System

Throw away the Mark-I system.

Your goal for the Mark-I system should be to have a machine-usable knowledge base and a much clearer understanding of the task. The Mark-II system can reflect the clearer understanding and you may be able to build computerized tools to help translate the knowledge base into your new formalism.

In the Mark-II (and later) versions, begin to consider generality.

Will the system always be limited to small problems? When the knowledge base grows, will the system bog down? Can the system be generalized to work on large classes of problems? Can the system be made robust enough to be completely "fail soft" and "idiot proof"?

Decide who are the intended users of the ultimate system.

What kind of help do they want? How do they describe the bottlenecks in their work? What is the context in which the program will be used?

Make the I/O to the system appear "natural" to the users.

Stylized "pseudo-English" can be easy to understand and also easy to parse and generate. The form in which answers are displayed should be close to standard textbook forms.

8. On Evaluating the System

Ask early about how you will evaluate the success of your efforts.

Will you collect testimonials? Will you require correct solutions for new problems? Do you expect "Nobel prize" winning performance?

Ask early about how the expert would evaluate the performance of the system.

Does its success depend on solving the problem you are trying to make it solve? If not, then you are not clear enough on the task or the expert may have false expectations. The expert may expect not only the right answer but also the right way of getting the answer.

The user interface is crucial to the ultimate acceptance of the system.

A system that does a superb job may not be accepted if it is too hard to use or too brittle in the hands of our inexperienced user. A little human engineering may easily make the difference between success and failure.

9. General Advice

Exploit redundancy.

Redundant data, hypothesis structures and inference rules can be useful for avoiding problems caused by erroneous or missing information. No conclusion should rely too strongly on a single piece of evidence.

Be familiar with several A.I. paradigms or frameworks.

One of your major problems will be to match your task with an appropriate paradigm. The wrong paradigm can make a task quite hard; the right one can make it much easier.

The process of building an expert system is inherently experimental.

You are experimenting with both the total system framework and with the individual rules. There is really little chance that you can figure out everything beforehand. The only way you can learn the rest is by trying to build a running system.