6.871 PROJECT CAUTIONS

There are a number of standard things that can go wrong in the design and task selection phase of 6.871 projects. To help keep you out of trouble, review the following list of potential problems, and then avoid them! If you have any doubt as to whether you're falling into one or more of these traps, you probably are. In that case, *please* come talk with any of the course staff; we can very often remedy the situation by making small changes, especially if you come see us early enough.

As a consequence of having cautioned you about these problems, we will be unforgiving when grading the projects if your program still manages to exhibit one of these.

A Composite Bad Example

Ever interested in getting rich, Ben Bitdiddle decided to build a financial advisor expert system that interviewed you about your financial situation and then recommended one or more investment ideas. Ben decided that the program would be extremely powerful if it had a large number of different investment ideas. He assembled a list of ideas organized by life stage (5 stages, from starting out to retiring), investment horizon (6 values, from 1 month to 10 years), and industry (10 values, including computers, automobile, foodstuffs, etc.). He then boasted that his impressive system had 300 different ideas, and therefore must be very smart.

In keeping with all the latest trends, Ben also decided to interface his system to the stock quote information available on the World Wide Web using `Java±` (a soon-to-be released version of a kind of sort of dialect of Java); he also built an interface to a PC program that permitted electronic investment via Merrill Lynch.

In an early run of the program Ben was troubled by a lack of smoothness in the interaction with the user. Unlike a human interviewer, who would proceed methodically through the issues, starting with investment horizon, then life stages, then industry, the system asked questions in an order that depended only on the rule it happened to be using. Ben eventually figured out a way around this.

What's wrong with this? Several things:

1. The Database Lookup Bug; aka The Shallow Inference Bug

Unfortunately, all the rules in Ben's knowledge base were of the same general form, looking like: `if life-stage of person is young-professional, and investment horizon of person is 3-5 years, and preferred industry of person is computers, then suggest investing in Microsoft`. Note that he had 5 life stages, 6 horizons, and 10 industries, and 5*6**10 = 300. In effect the system simply looked up one idea in each of 300 different categories. Dull. Very dull.

Remedy: Make the reasoning *deeper* instead of *broader*. Rather than 300 investments selected in a single reasoning step each, take a dozen ideas and develop a set of rules that's much deeper. Instead of asking what the person's investment horizon is, *figure it out*, i.e., write rules capable of determining what it ought to be. Doing that might involve knowing what their savings goals are (e.g., buy a house, educate kids, retire comfortably). Don't just ask, write rules that figure it out. In general, always try to write rules (or create other knowledge representations) that figure it out, rather than ask. Apply this principle recursively, and you'll end up with a system with a reasonably deep chain of reasoning, one that works from more basic data, and hence is both "smarter" and more useful.

2. The Eyewash Bug

Ben will spend about 90% of his programming time trying to get his expert system tool to interface with the World Wide Web browser and Java, and talk to the electronic investment program. Don't bother. Such things make for nice slides and good interfaces are very important, but they're

just not what the particular course is about. Instead of actually connecting via the Web, have the system simply print out the relevant URL; instead of having it interface to the electronic investment program, have it dump out the relevant information in a file, and then if you like, let the user enter that info into the PC program by hand.

First make the program smart; second, make the program smart; for our purposes everything else is unimportant.

3. The Dialog Bug

Don't worry about whether your system asks questions in an order that looks smart. The job of the program is to be smart in its reasoning, and that doesn't necessarily mean it will ask questions in an order that is familiar (and hence seems smart). A doctor program, for example, might start with some very obscure questions, whereas a human doctor might start general and work toward specifics. The important task here is to make the program smart in what it knows (ie, the diagnoses it can do); user inteface issues are secondary for now. They do matter in real use, but it overloads the project to try to accomplish all of these things in a single term.

4. The Traditional Program Bug

Ben decides to redesign the program after he sees a brochure from a local investment firm that offers a neat little flowchart for selecting from among a group of investments. This is particularly handy because Ben is a good programmer and knows how to turn a flowchart into a working program very quickly.

The problem here is that Ben has missed the idea of a knowledge-based system. In our slogan-like fashion, the idea in KBS is to tell the program what to know, not what to do. Traditional programming is about telling the program what to do, i.e., what procedures to follow (e.g., the flowchart). Knowledge-based systems focus on giving the program the relevant knowledge (tell it what to do), and leaving it to the program to apply the knowledge as it sees fit.

One guideline here is to ask yourself if you can easily identify a knowledge base and an inference engine in your system. If you simply print out your knowledge base (in the same form that the program uses it), will anyone else be able to answer the question, What does your program know? If not, you probably have a problem.

(Another variant on this bug is the *Decision Tree Bug*: writing a decision tree, instead of building a knowledge base.)

5. The Optimization/Linear Programming Bug

Ben decides to redesign the program, and figures out that investment can be viewed in terms of optimizing risk and return. He gets all the relevant information, dumps it into an optimizer, and prints out investment suggestions.

This may be a perfectly good way to build an investment advisor; it just doesn't have anything to do with this class. Be wary of any problem statement that involves coming up with an answer that is optimal in some sense. Our job is to capture the informal knowledge that people have that allows them to find "good enough" solutions to problems. Where problems are appropriately solvable with some manner of optimizer, fine; use it. We're focusing on those problems where optimization techniques are unavailable.

Another variant on this bug shows up in programs whose job is to find the best route through some search space, e.g., the best route from Boston to NY. This is basically a search and optimization problem. One remedy is to turn it into an expert system problem by redefining the task. For example, build a program intended to help someone pick a place to vacation, depending on his or her interests, family makeup, etc.

Please come and talk to us if you have any concerns about your project.