Chapter Fifteen

**Testing and debugging of artificial intelligence programs**

Artificial intelligence programs tend to be big, complicated programs. (If not in this book, in practical applications.) So preparing for good testing and debugging is especially important. Standard software engineering techniques for managing large programs (modularity, top-down decomposition, good naming, separate testing of components, and so on) help, but certain other less common techniques are particularly important to artificial intelligence. These are summarized in Figure 15-1.

**The gold standard**

Artificial intelligence programs try to show intelligent behavior. But as we said in Chapter 1, "intelligence" is a vague word. So it's often hard to see whether an artificial-intelligence program is working properly. Of course, there are obvious mistakes, like a car repair program that concludes your engine is destroyed when the problem is a flat tire. But the less obvious mistakes are more dangerous.

What is usually done is to define an absolute or *gold* standard of behavior against which an artificial-intelligence program can be measured. Often (as for expert systems) a program tries to automate things done by certain humans, so we can just ask these humans whether the program is doing things right. For instance, for auto repair, there are master mechanics; for medical diagnosis, there are doctors that are authorities in their subfield; for scheduling, there are people that schedule all the time, like some secretaries; for finding routes in a city, there are experienced taxi drivers; for solving puzzles, there are expert puzzlers; and for mellowing out, there are California residents. Sometimes there is no single human expert, but different people are expert on different pieces of a problem and we can pool their expertise, as often case with knowledge of how to best accomplish things in a bureaucracy.

Unfortunately humans often hate computers, particularly when they are being replaced by computers without compensating rewards, so we must be extremely careful to make the comparison of computer with human behavior a fair one. It's important to give the humans and the computers the same problems to work on (perhaps not telling the humans that they will be compared to computers and not telling the computers that they will be compared to humans, though computers tend be more broad-minded). Then we compare results and mark any discrepancy as wrong. If our graders don't know themselves which results were by computers and which by humans, only some supervisors then we have a *double-blind* test, a good form of testing often used in medical research.

**Cases**

Once a gold standard is established, it is important to test an artificial intelligence system many ways, or against many situations. These situations are called *cases* or *test cases*; a set of cases used for testing is called a *corpus*. The more complicated the program, the more cases we should run. It is important to test a representative spectrum of cases. It is desirable to have at least one test case for each major program behavior, for each significant combination of inputs, and for each significant kind of output. Since this isn't always possible, judgment is required to reach a compromise.

**Focusing on bugs**

Testing performance on a set of cases creates two categories: cases for which the standard and the program agree, and cases for which they disagree. The second category is the more interesting one. We can examine these program "failures", and try to identify characteristics that distinguish them from "successes".

Several tools can help. A very important one is trace information, the record of reasoning done about a case. Rules that appear often in failure cases are suspect. Search problems provide partial traces in their operator-sequence output. But expert systems usually just give a single answer, and can be considerably enhanced with a trace of what rules gave the answer.

Other useful information is provided by a confusion matrix, something especially helpful with rule-based systems. It summarizes not only when the program went wrong, but how it went wrong. An example is given in Figure 15-2. Program results were grouped into categories and cross-tabulated with the gold standard. Rows represent gold-standard categories and columns represent program-result categories. The entry in row i and column j is the number of times the program gave an answer in category j when the standard gave an answer in category i. (We assume that one of these diagnoses is a default, so the program always reaches one of these four diagnoses.) So a confusion matrix shows which categories tend to be confused with which other categories, and suggests where rule improvements are needed. For instance according to Figure 15-2, the "electrical short" problem is the most frequently misdiagnosed (only 29 out of 55 cases were correctly recognized), suggesting that more rules for electrical shorts or rules with less restrictive conditions are needed. On the other hand, "fuel blockage" is the diagnosis made most frequently misapplied (only 25 out of 45 fuel blockage diagnoses were correct), suggesting that more and stronger conditions in its rules are necessary to filter out mistakes.

**Exploiting pairs of similar cases**

Pairs of very similar cases are helpful in testing and debugging of artificial-intelligence programs. They can localize problems and provide generalizations.

A particularly interesting case pair is when the standard and the program disagree on one case, but agree on a very similar case. In other words, whenever failing cases are *near misses* to success. Then the reason for the failure must lie in the small difference in characteristics of the two cases, which helps focus on the source of the bug. For example, suppose we have in a rule-based system:

a :- b, c.

Now suppose we test (among others) two cases: case X in which facts **b** and **c** are true, and case Y in which **b**, **c**, and **d** are true. Suppose X is a near miss for conclusion **a**, in that the program (or specifically, this rule) concludes that **a** is true while the gold standard says that **a** is false. The program will also conclude that **a** holds in case Y, and assume furthermore that the standard agrees. Since facts **b** and **c** are present in both X and Y, we can guess that the **d** fact makes the difference. There are other explanations, but that seems the simplest. So to get a better rule-based system, we should rewrite the rule as

a :- b, c, d.

In other words, we figured the original rule was too general, so we fixed it to apply to only one of the two original cases.

Pairs of similar cases can also suggest rule generalizations, a kind of *learning*. Suppose we have in a rule-based system:

a :- b, c, d.

Suppose case Y is as before, with **b**, **c**, and **d** true. Suppose in case Z that **b** and **d** are true but **c** is false. Suppose the gold standard says that Y and Z are both correct cases for conclusion **a**. Then we can guess that the **c** doesn't matter, suggesting the rule:

a :- b, d.

This is a riskier change than the previous "near-miss" one, because we are generalizing a rule to apply to more cases. But it's often reasonable.

When the preceding generalization is too strong, we can make weaker changes too. For the appliance application of Chapter 7, we might have written a rule:

diagnosis('short in cord') :- askif(device\_dead), askif(cord\_frayed).

Suppose we are now told that when there is a frayed cord and the lights just went out, a "short in cord" diagnosis should also be made. This might suggest

diagnosis('short in cord') :- askif(cord\_frayed).

But that's not a smart idea, because a frayed cord is only a weak symptom and can also suggest a break in the cord (that is, a nonconducting cord). Instead, we should use a concept covering both dead devices and the lights going out: a fuse being blown. That suggests:

diagnosis('short in cord') :- diagnosis('fuse blown'), askif(cord\_frayed).

Figure 15-3 tabulates some inferences of this sort. Current research is extending these ideas to build rule-based systems automatically from examples and nonexamples (that is, descriptions of situations that are definitely not examples of particular concepts)) defining appropriate behavior. This is part of the subfield of *automatic programming* within artificial intelligence. The issues are hard--just think how hard it is for you yourself to write a program.

**Composite results**

If running an artificial intelligence program gives a result with components, then evaluating the difference between the program and the gold standard is a little more complicated. If the pieces of the result are independent, then the program and standard can be compared separately for each piece, as for a relaxation application in which the result is separate possibility lists for variables. If pieces of the result are linked in a data structure, then structures must be matched to establish corresponding components before comparison, as for a search application in which we must match program operators with the gold standard operators in corresponding order. In either case, only the pieces that do not match need be studied further.

Here's an example. Suppose for some search problem the gold standard finds the operator sequence

[a,b,c,d,e,f,g,h]

while the program result is

[a,b,r,s,d,c,e,f,h]

Three things seem to be different in the program result: additional operators **r** and **s** have been inserted after action **b**, operators **c** and **d** have been interchanged, and operator **g** has been omitted. We can thus focus our attention on the parts of the search program that affect those things. So we should check the conditions controlling selection of **r** and **s** to see why they were erroneously fulfilled, and check the conditions on **g** to see why it was erroneously omitted; the switch of **c** and **d** suggests a mistake in the priority of those two operators.

**Numbers in comparisons**

Numbers--probabilities, evaluation functions, and cost functions--complicate comparisons of a program and a gold standard. There are three ways to handle them. First, we can use the numbers to pick a "best" program result, and compare this to the gold standard. For a diagnosis expert system that assigns probabilities to diagnoses, we can take the diagnosis with the highest probability; for a search program that finds paths to a goal, we can take the lowest-cost path. Second, we can treat the numbered items as a composite structure as discussed in the last section. For an expert system that concludes that diagnoses **a**, **b**, **c**, and **d** are probable with probabilities 0.9, 0.7, 0.6, and 0.3 respectively, while a gold standard rates them 0.8, 0.5, 0.6, and 0.2 respectively, we can focus the debugging **b** and **c** because their numbers are out of order. Third, we can do statistical analyses; for instance, we can take the sum of the square of the deviations of the program-result numbers from the gold-standard numbers to measure closeness of the program and the standard. We can then quantify any proposed program improvement by this criterion. So it can be used as an evaluation function in a "search" for a better program.

**Preventive measures**

By current estimates an ounce of prevention is worth 1.07 pounds of fixes, so it's important that an artificial-intelligence programming environment provide software tools to help programmers avoid mistakes in the first place. One big help is predefined expectations about the format of code, as forms or frames in the style of Chapter 12. For rule-based expert systems we can use rule frames, to capture the frequently strong structural similarities between different rules. For instance, rules in a car-repair diagnosis system usually go like this:

(1) localize the problem in a part of the car;   
(2) ask about easily observable symptoms;   
(3) if necessary, ask about symptoms that take a little work to establish (like opening up an object);   
(4) if necessary, ask about experimental actions that need to be done to confirm the diagnosis.

For example:

diagnosis('Dead battery') :-

electrical\_problem, lights\_dead, not(battery\_corroded), battery\_voltage\_low.

A programmer writing a new car-diagnosis rule can be provided with this list of four things as a guide, or warned when departing from this format. Or to be more ambitious, a system could automatically find analogous rules to a new rule a programmer wants to write. Inheritance on a hierarchy of rule-class frames is helpful for this.

**Supporting intuitive debugging**

A gold standard of performance for an artificial intelligence program can be hard to get and costly to use. Early in debugging, a more intuitive approach to performance analysis using *explanation* facilities is better, and this is common in artificial-intelligence development software like expert systems *packages* and *shells*. After running a program, these facilities can report on what your program did, how it reached a particular conclusion, or why it considered a particular line of reasoning and not another. Such explanations can be helpful even when the system is fully debugged; they help people understand the system better, use it more effectively, and have more confidence in its results.

The commonest explanation capabilities are for:

"How" questions: questions as to how the program reached a particular final or intermediate conclusion. When the reasoning is simple, a printout of every step will suffice--just the trace information mentioned in Section 15.3. But when reasoning is complicated, this is too much information. For instance, Prolog trace utilities (see Appendix D.12) generally print out information not only on calling a rule and succeeding with it, but on failing and on backtracking to it. (Try tracing the appliance diagnosis program in Section 7.6 or the flashlight diagnosis problem in Section 11.5, if you have tracing facilities in your Prolog dialect, to see the hundreds of reasoning steps done.) So partial summary of trace information is usually provided in artificial-intelligence environments and packages. For instance, a "how" explanation can list just the "top-level" predicates used to reach a conclusion, and then provide the programmer additional details on demand.

"How not" questions: questions as to why something was *not* concluded. These arise when expectations are violated, and there are two very different situations. First, reasoning found the conclusion false; then the trace information answers the question. Second, no try was made to prove the conclusion; then the program control structure must be studied to determine why different lines of reasoning were pursued instead. For instance, the only rules that could have established the conclusion may occur beyond the point where processing went in a sequence of rules.

"Why ask" questions: questions as to what purpose a question to the user serves. These are only appropriate in expert systems that question a user to get data, like the appliance repair program in Chapter 7 with its **ask** predicate. Instead of answering a question, a user might reply "Why ask that?" and get a description of a higher-level issue the question will help resolve or a description of what the program is working on. As with "how" questions, summarization helps. That is, a user doesn't usually want just the issue (rule left side) immediately responsible for the question but the higher-level issues too.

"Why better" questions: questions as to why one alternative was judged better than another, when alternatives are ranked by numbers (probabilities, evaluation functions, or cost functions). These are hard to answer because they depend on mathematical formulas as well as logical reasoning. One approach is to take the trace record ("how" information) from two comparable conclusions, eliminate everything identical, and try to find one-to-one correspondences between the remaining subconclusions. Explanation can focus on the relative desirability of corresponding subconclusions.

The hope in providing such information to programmers and users is to permit them to use their own intuitions on cases, comparing their reasoning with what the program did. Often artificial intelligence programs try to formalize human intuition, especially expert systems, so an error by a program can "leap out" at the human studying its results.

**Evaluating cooperativeness**

Right answers are not the only criterion we have for evaluating an artificial-intelligence program. We may also care how friendly or cooperative it is, especially if it requires a lot of human interaction. Tools such as questionnaires for measuring this are common in organizations that do considerable software development. Some issues to consider:

--Is the output comprehensible?   
--Does the program use specialized jargon?   
--Can the program paraphrase its questions to the user?   
--Can the program explain its reasoning well?   
--Does the program catch obvious user mistakes?   
--Does the program guess well what the user meant in a mistake?   
--Does the program take what the user says too literally?

**On problems unsuitable for artificial intelligence**

The performance of a program may still be unsatisfactory after a long period of testing and debugging. Despite all the clever things that artificial intelligence techniques can do, they can't solve everything. Sometimes it is hard to know in advance whether a problem is suitable. But some guidelines can be given.

First, a problem suitable for artificial intelligence techniques must be sufficiently well-defined to have clear criteria for success and failure, just as for any other technical method. Otherwise, you can't tell when it's working, and you can't easily test and debug it. For instance, an expert system could create works of art, but it would be hard to evaluate--there's a lot of disagreement between art "experts".

Second, the desired program behavior must be sufficiently stable to permit cost-effective testing and debugging. For instance, an expert system to recommend stock-market investments would be hard to build because investment opportunities are constantly changing ("buy low, sell high" is too general to be useful).

Third, it must be possible to use the program routinely. This is obvious but often overlooked. For instance, most military "battle management" systems are a bad idea because it is too hard to keep them correctly and quickly informed of everything that is happening in a battle (especially with communications equipment itself under attack), though prototype implementations for artificially constructed scenarios might seem to work well.

Fourth, an artificial intelligence system must not be reductionist in a dangerous way. As we mentioned in Chapter 1, reductionism is the extent to which an artificial intelligence program fails to capture subtleties of human behavior it is trying to imitate. A highly reductionist program may be safe when its users thoroughly understand its limitations, but can be very dangerous otherwise. For instance, a significantly reductionist battle management system could order troops to fire on themselves, or be vulnerable to sabotage in surprising ways.

Fifth, artificial intelligence techniques should generally not be used when processing time, storage space, or calculation accuracy is critical. The techniques and programs are complicated and often probabilistic or heuristic. A working artificial-intelligence program can often be optimized or compiled to run faster, take up less space, or make more accurate conclusions, but the main idea of artificial intelligence is to provide "intelligence" in computers first, efficiency later. So for instance a "robot warplane", an autonomous military aircraft controlled by real-time artificial-intelligence software, is not likely to work fast enough because things happen too fast in a battle, a large computer imposes a weight penalty on the aircraft, and high accuracy in maneuvers is essential.

Sixth, an artificial intelligence program must be sufficiently complicated, but not too complicated, so that artificial intelligence techniques do better than competing approaches. For instance, in spite of Chapter 11, it's a poor idea to write a search program to fix flashlights because people can figure it out themselves. But an expert system to replace all doctors isn't reasonable, because there are millions of diseases and millions of symptoms, and just the rule management and indexing problem for such a huge expert system would be mammoth. And, in the United States anyway, a flawed system could be sued for malpractice.

You must study carefully these criteria before writing an artificial-intelligence program. The criteria are on the borderline between the technical and the nontechnical. Should this discourage a programmer from worrying about them? No, not at all. Too often technical people ignore such issues when technical insights about them could be invaluable. Insights about artificial intelligence are particularly important today because so few people understand the field--it's hard to learn, as we said in Chapter 1. But as a new area of research compared to other fields of engineering, artificial intelligence has exciting promises for the future, promises generating considerable enthusiasm. Now that you have read this book you can help fulfill them. Use your knowledge wisely.

**Exercises**

15-1. (A) Give a better gold standard for auto repair than an opinion of an expert mechanic.

15-2. Suppose we have a backward-chaining rule-based diagnosis system written as Prolog rules. How could we modify the rules to automatically provide a trace of rules used along with every diagnosis?

15-3. (A) Suppose we have a rule-based system like the appliance diagnosis program in Chapter 7 for which the order of the rules makes a considerable difference in what conclusion is reached first. Suppose for some diagnosis rule-based system, we have the diagnosis rules for two diagnoses in the wrong order. How could a confusion matrix clue us that this is the case?

15-4. Suppose you wanted to construct a rule-based system automatically from examples and nonexamples of its desired behavior, using the ideas of Figure 15-3. The criteria in the Figure only refer to rules. How would you get an initial set of rules to work on?

15-5. Rule R1 is redundant with respect to rule R2 if any conclusion proved by R1 can also be proved by R2 in the same circumstances.

(a) Explain how R1 can have a different left side than R2 even when it is redundant with respect to R2.

(b) Give a criterion for the redundancy of R1 with respect to R2 that does not require execution of the rules.

15-6. (R,A,E) Discuss the overall suitability of artificial intelligence techniques as the *major* component in the following.

(a) An automatic car that will accelerate and steer on highways without human assistance.

(b) A generator of new cooking recipes.

(c) A "seduction advisor" expert system that advises you how to succeed with members of the opposite sex.

(d) A system to coordinate defense from nuclear missiles.

15-7. (H,P,G) Write a program to automatically build ("learn") a rule-based system from examples and near misses (nonexamples) of left-side predicates. Consider only predicate expressions without variables. Use Figure 15-3 as a guide, and break your problem into the following components:

(1) a "comparer" that takes two inputs, a set of rules (a list of lists) and a description of some situation that is either an example or a near miss (a list of facts). It must compute the differences between the situation and the closest rule, the rule that matches the most facts of the situation. For that rule, it should list differences of four kinds: (a) additions (things that must be added to the rule to match a fact of the situation), (b) deletions (things that must be removed from the rule because they have no counterpart in the situation), (c) bindings (variables in a rule that can be matched to a constant argument of the same-named predicate in the situation), and (d) substitutions (constant arguments in a rule that can be matched to a different constant argument of the same-named predicate in the situation).

(2) An "example handler" that takes the same two inputs as the comparer. It hands these two arguments immediately to the comparer, obtaining the list of differences of the situation from one particular rule. It then modifies the rule appropriately, or creates a new rule, to cover the situation. This done, the new generalized rule may cover anything handled by another rule, and the other rule can be eliminated. The example handler then returns the new set of rules as its result, together with a "gripe amount" indicating how happy it was with the rule modifications it made. If more than one modified set of rules seems equally reasonable, the example handler should permit backtracking into it.

(3) A "near miss handler" analogous to the example handler. It should call the comparer, get back a set of differences between a near miss situation and a rule, and modify (specialize) the rule appropriately so it won't apply to the situation. It should check that its changes don't cause contradictions among the rules. Like the example handler, it should return a new set of rules with a "gripe amount", and should give further answers (if any seem good) on backtracking.

(4) A "top-level searcher" that supplies examples and near misses to the rest of the system, searching over all possible permutations in the order of supplying them. There's a good cost function in the "gripe amounts" returned by the example and near miss handlers, and there's a good evaluation function in the number of rules plus the number of constants in rules. (Fewer rules means better generalization, and fewer constants mean more generality in rule application). This should use the A\* program of Chapter 10, defining the successor function to call the example handler or near miss handler depending on whether the next situation chosen is marked as an example or a near miss.

Try learning rules for some simple diagnosis expert system.

**Miscellaneous exercises covering the entire book**

M-1. (A) Backtracking is important in some very different areas of artificial intelligence. For each of the following, say "yes" or "no" as to whether backtracking is usually necessary to implement them. (All these might use backtracking in Prolog implementations, but which need backtracking no matter what language they are implemented in?) By "backtracking" we mean returning to a processing state from which the current processing state is reachable.

(a) backward chaining

(b) forward chaining

(c) best-first search

(d) answering long queries efficiently by exploiting dependencies

(e) slot inheritance

(f) resolution among members of a set of clauses without variables

M-2. Lattices (directed graphs without cycles) are important in many different ways in artificial intelligence. They're important not only as a data structure inside the computer, but as a way of explaining what the computer is doing. Explain in 40 words or less what they are useful for in each of the following areas.

(a) search

(b) frames

(c) compiled rule-based systems, in a way different from uncompiled rule-based systems

(d) resolution theorem-proving

M-3. (R,A) Give the letter for the best match to each item in the first list from the choices in the second list. Each item should be matched to one and only one item. (Hint: doublecheck your answers by re-solving from bottom to top, and then by matching the second list to the first list.)

Horn clause

generate-and-test

constraint

forward chaining

default

depth-first control structure

intension

difference table

near miss

Skolem function

caching

dependency

multiple inheritance

decision lattice

Bayes's Rule

heuristic

hierarchical reasoning

agenda

a. a way of reusing results

b. a weak general-purpose rule or fact

c. contradictory reasoning is a danger

d. a theorem

e. example: prefer investigating further the last thing discovered

f. "or" with "not" in front of every expression but one

g. a fast implementation of rule-based systems

h. recommends an operator

i. example: the Prolog query **?- a(X), b(X), c(X).**

j. used for existential quantifiers

k. lists candidates

l. a description of means-ends analysis

m. a predicate expression

n. valuable in debugging all artificial intelligence systems

o. reasoning about implications of facts

p. the abstract meaning of something

q. requires a stack

r. when two predicate expressions have some of the same variables

M-4. (R,A) Consider an expert system built to advise Americans which income tax forms and schedules to fill out. There are different ways satisfying the laws, and some are better for the taxpayer than others. (It matters how much a taxpayer earns and how much of their income is of a certain type. The taxpayer wants the possibility that costs the least money.) Using answers to some simple questions, an expert system could make recommendations without having to ask for a complete financial history. But its conclusions will be necessarily imperfect without such complete information.

(a) Which of the following techniques is most appropriate for this problem? Explain why.

1. and-or-not lattice   
2. dependency-based backtracking   
3. confusion-matrix conflict-resolution   
4. breadth-first resolution strategy

(b) Which of the following techniques is *least* useful for this problem? Explain why not.

1. virtual facts   
2. inference   
3. depth-first control structure   
4. concurrency

(c) Which of the following Prolog features is *least* useful in implementing an expert system in Prolog for this problem? Explain.

(a) **asserta** or **assertz**   
(b) **repeat**   
(c) **write**   
(d) **>** [greater-than sign]

M-5. (A) Consider the problem of foot placement for a 6-legged robot. Suppose the robot has a vision system that studies the ground in terms of squares one square meter in area. For each such square the vision system (which you don't need to consider) decides whether that square is a good one to place a leg on. Suppose only 20 percent of the squares are suitable on the average, and suppose the robot can only extend its legs a certain distance forward. Suppose the vision system only examines squares within a rectangle centered in front of the robot, extending from the front of the robot to ten meters ahead, and ten meters off to the left, and ten meters off to the right. Consider the problem of how to use that information to plan a good sequence of foot placements to the centers of squares, keeping the robot moving ahead in an approximately straight line. The robot is not in a hurry but it doesn't want to ever back up.

(a) Explain which of these search strategies is best for this problem and why.

1. best-first search   
2. weight-first search   
3. breadth-first search   
4. branch-and-bound search

(b) Which of the following are also helpful in step planning? Explain why.

1. means-ends analysis   
2. caching   
3. both generate-and-test and caching

(c) Which of the following Prolog features is *least* useful for implementing this step-planning (not the vision system) in Prolog? Explain why.

1. **is**   
2. **assertz** (as opposed to **asserta**)   
3. **cut** (!)   
4. **repeat**

M-6. Many artificial intelligence applications require representation of causation. Suppose we use the Prolog predicate **causes(X,Y)**, where **X** and **Y** are names of events such that event **X** causes event **Y** to happen. **X** is assumed to be sufficient for **Y** to happen (so that any time **X** happens **Y** must necessarily happen too), but **X** is not necessary for **Y** to happen (**Y** may have other causes).

(a) Is **causes** transitive? Explain.

(b) Does **causes** inherit with respect to **part\_of**? If so, in which direction? Explain.

(c) Consider an alternative representation of causation in Prolog as a rule with the "**:-**" symbol. For example:

event(lights\_come\_on) :- event(turn\_wall\_switch\_to\_on).

or better yet:

event(lights\_come\_on) :- event(turn\_wall\_switch\_to\_on),

not(bulbs\_burned\_out), not(wiring\_defective), not(switch\_broken).

But give a major disadvantage of representing causation this way as a rule instead of with **causes(X,Y)** facts. (Give a "deep" problem; don't consider efficiency.)

M-7. (E) The conjunctions "and" and "or" are very important in artificial intelligence. Consider the conjunction "but".

(a) Suppose we let experts building an expert system use the word "but" in writing rules. How should we translate it into something a programming language like Prolog can handle? Be sure your answer covers most of the usual meaning of "but".

(b) Vocabulary variety makes computer output more comfortable for humans. Discuss when a computer could use "but" in natural language output from a program. In particular, consider a query-answering program. Compare with your answer to part (a).

(c) Would a "but" feature, in addition to "and" and "or", be useful in a logic programming language like Prolog?

M-8. (E) Discuss the possible application of artificial intelligence techniques to the following two problems:

(a) You are in a submarine and have various sensors. You want to guess the identity of all moving objects up to the limits of your sensor range.   
(b) You are positioning army units for a possible battle. You have various kinds of personnel, and various kinds of weapons. You want to position units and equipment in a way that gives you the most tactical advantage.

For each problem, discuss which of the following specific techniques apply:

automatic backtracking, generate-and-test, inheritance, defaults, rule-based systems, backward chaining, forward chaining, fixed-order conflict resolution, focus-of-attention conflict resolution, probabilities, depth-first search, breadth-first search, best-first search, A\* search, means-ends analysis, frames, relaxation, dependency-based backtracking, resolution

M-9. (H,E) Consider the problem of building an automatic "commercial censor" system that will determine when a television is receiving a commercial, and will turn off the audio and video until the commercial is finished. (This would be nice when videotaping.) 95% accuracy within two seconds after a commercial starts or ends is desired. In addition to the frequency analyzer and the low-level vision processor mentioned, assume you have a fast computer with one million bytes of random-access memory and ten million bytes of disk storage. Conduct experiments (involving observation of at least 200 commercials) with a real television to help answer the following questions. An adequate answer to this problem will be 2000 words total.

(a) Categorize clues available from processing of the audio signal. Assume frequency-spectrum calculation hardware (technically, a discrete Fourier transform processor) is available.

(b) Categorize clues available from processing of the video signal. Assume a low-level vision processor (that groups homogeneous two-dimensional regions of a single picture, and calculates their average brightness and color, but slowly).

(c) Categorize clues available from correlation of the results of parts (a) and (b) with an accurate clock connected to the system.

(d) Describe a good control structure (or control structures) for an artificial intelligence system to do this job. Be sure to include whether backward, forward, or hybrid chaining is desirable; and whether depth-first, breadth-first, best-first, or A\* searching is desirable.

(e) Suggest good uncertainty combination methods for such an artificial intelligence system. Are numbers a good idea? In particular, suppose each clue from parts (a), (b), and (c) is summarized by a single probability: how would you combine these three?

(f) Where might learning techniques help to automatically improve system performance with time? In particular, discuss how learning from examples can be done. Describe exactly what is learned: how it would be represented in the computer, how it would be modified upon new data, and so on. Remember, artificial intelligence methods are mostly nonnumeric, so your learning must be mostly nonnumeric.

(g) Describe how you would fairly validate the 95% accuracy claim. When would you run the system, how long, and how many times? Run some simple (not necessarily thorough) tests on a real television of some of the ideas you've developed previously. Explain how you could also use these testing procedures in debugging.

Some hints: often students are too superficial in their answers to essay questions like these, so err on the side of writing too much rather than writing too little. Don't give definitions of terms and other things that are easy to look up.